LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-JC-152963-REV-1

A Multi-Layered Image Cache for Scientific Visualization

E. LaMar

June 26, 2003

Conference on Visualization and Data Analysis 2004,
San Jose, California, January 18-22, 2004

Updated October 14, 2003

A Multi-Layered Image Cache for Scientific Visualization

Eric LaMar

ABSTRACT

We introduce a multi-layered image cache system that is designed to work with a pool of rendering engines to facilitate an interactive, frameless, asynchronous rendering environment. Our system decouples the rendering from the display of imagery. Therefore, it decouples render frequency and resolution from display frequency and resolution, and allows asynchronous transmission of imagery instead of the compute–send cycle of standard parallel systems. It also allows local, incremental refinement of imagery without requiring all imagery to be re-rendered. Images are placed in fixed position in camera (vs. world) space to eliminate occlusion artifacts. Display quality is improved by increasing the number of images. Interactivity is improved by decreasing the number of images.

keywords: image cache, impostors, scientific visualization, multi-resolution techniques, hierarchical techniques, parallel techniques.

1. INTRODUCTION

In this paper, we consider the problem of providing a simple and general solution to the problem of scaling the performance of a graphics system with respect to large variations of the input data set and the resources available. Given a basic rendering engine, it is a major task to provide scalability of the engine's performance with respect to (i) size of the input dataset, (ii) resolution of the output display and (iii) amount of computational resources available (including parallel and distributed processing units, memory, and hardware acceleration). In a complex system, this task may also be replicated several times as several rendering engines are used concurrently (such as slicing, volume rendering, and iso-contouring). Previous parallel rendering techniques are based on a compute–send–display cycle that must be executed strictly within the time elapsed between the display of two frames. That is, while the parallel rendering engines are producing images, the network is not being used. When rendering of the images is complete, all of the render engines synchronize; the slowest engine therefore determines the overall time that is used to produce images. The previously idle network then becomes saturated with traffic as the images are all sent at the same time. The network is being used very inefficiently as it alternates between disuse and over use.

Caching and reusing imagery over several display cycles relaxes the requirement that all images be re-rendered every frame. Moreover, the images can be sent continuously, thereby spreading the load over time and utilizing the communications resources more efficiently. This generation of images without a strict sense of a frame is called "frameless rendering."

There is a large body of research on using impostor images generated off-line to approximate large scenes that cannot be rendered interactively. However, existing techniques are unable to meet the demands of scientific visualization in that they expect static scenes, and require significant preprocessing time and user assistance to place impostors. Scientific visualization methods require user-controlled, run-time parameters that can significantly alter the visualization results, *i.e.*, transfer-functions for volume visualization, iso-value for iso-contouring, or temporal movement for time-varying datasets. All of these require adaptive refinement and dynamic updating of the imagery.

Outline of the approach. We have developed our Multi-Layered Image Caching (MLIC) system to cache rendered images and to address issues specific to the use of impostors for scientific visualization. The components of our system include a display engine, an image database, and a set of rendering engines. Our system uses a special decomposition of space about the viewpoint: the first is to refine the space into a set of non-occluding (with respect to the viewpoint) polyhedra. These polyhedra are then refined by an individual k-D tree to allow for adaptivity. As the viewpoint moves, so do the polyhedra and k-D trees move.

Each node of the k-D trees has an associated image produced from the data enclosed by that node; the images have specific spatial position and extent.

The images all have the same pixel resolution, so low-resolution regions are covered by fewer images with larger spatial extent, and high-resolution regions are covered by more images with smaller spatial extent.

The images can overlay each other with respect to the viewpoint, so they include both color and opacity information. The images cover the space around the viewpoint, although only those currently visible in the camera's view frustum are displayed. The images do not move with respect to the viewpoint (or each other), so do not include depth information.

The rendering engines receive requests to generate new images. Upon completion, they send an acknowledgment to the display engine and send images to the image database.

By combining an image database, a display engine that displays only what is available, and the explicit notification of render requests and acknowledgments, the MLIC system accomplishes the decoupling of image display rates from image generation rates. The decoupling

of display resolution from rendering resolution is accomplished by images covering different spatial extents. Local refinement of data does not require re-rendering all images. Instead, only those images that intersect the refinement region need to be re-rendered.

As the images are held at fixed positions with respect to the viewpoint, a translation of the viewpoint through the data means that images have also moved from where they were rendered. These images should be re-rendered; however, if the translation is small, they can be used to approximate the correct image. Hence, if the rate at which images can be re-rendered is slow, and the viewpoint is moving slowly, not all images need be re-rendered each display cycle.

The display engine displays images from the image database. It iterates through the set of polyhedra that decompose the space about the viewpoint. For each polyhedra in the view frustum, traverse the associated k-D tree, displaying images that meet the current display requirements. If an image does not meet display requirements (*i.e.*, it was generated when the viewpoint was an a different location or it covers too large a spatial extent), the display engine sends a request to the render engines to generate one or more new images. When these new images are available, the display engine displays them.

Main Results. We have implemented a parallel shared memory and distributed MPI versions of our MLIC system, on an SGI Origin3000 and a Linux cluster, respectively. Our intent is to provide interactive rendering of large datasets and linear scaling on the number of rendering engines. Both implementations show linear scaling for some modest-sized (512^3) datasets.

We currently use VTK for rendering because it supports a very large array of rendering modalities. However, none of the modalities is very fast, and VTK cannot seem to handle datasets larger than 512^3.

Both implementations use simple FIFO queues for render requests and acknowledgments. When moving through a dataset quickly and displaying it at high resolution (*i.e.*, displaying a large number of images), the rendering engines are not able to re-render images from the current viewpoint quickly enough to produce a consistent view of the data (*i.e.*, all the images are from slightly different locations). This can make it difficult to navigate through the dataset. Prioritizing re-render requests to provide a consistent view for the very center of the display, while letting the periphery be less consistent, could make navigation easier. Similarly, predicting the viewpoint location would allow generation of images that will be correct.

2. RELATED WORK

In this section we discuss related works in the areas of parallel and distributed rendering, image-based rendering techniques, multiresolution data structures, specialized hardware, and other related techniques.

2.1. Parallel and Distributed Rendering

Ma *etal.*[1] introduces Binary-Swap composition to reduce the total amount of imagery sent over the network of a parallel machine when performing parallel, distributed rendering. The data are distributed as thin slabs that cover the viewport. Instead of each node transmitting its entire image to one common node to be composited, the images are composited in a binary tree hierarchy. Progressively smaller portions of the images are swapped and composited, until each processor contains a P/N portion of the final image, where P is number of pixels in the final image and N is the number of processors. The N partial images are sent to the display buffer. They observe that communication takes a small portion of the overall time used to generate a single, complete image, when using up to 512 processors. This method, however, requires a hypercube topology to efficiently implement non-interfering image-swaps. The authors implemented their system on a Connection Machines CM5 (which has a hyper-cube topology and very high bi-section bandwidth), and did not discuss issues or possible problems with other communication topologies.

Parker *etal.*[2] discuss parallel isosurface computation on an Origin2000. They observe that while ray-tracing has a high per-instance cost, it is extremely parallelizable. They use a two-level hierarchy to enable skipping of empty space, and observe near-linear scaling, and a 10 frames-per-second visualization, on the 1GB Visible Woman dataset. Their technique relies on the data to be a regular, Cartesian grid, for efficient traversal and iso-contour tests.

Ahrens *etal.*[3] discuss an on going project to parallelize the general data-path of Kitware's Visualization Toolkit,[4] with the goal of visualizing extremely large datasets on large, parallel machines, using VTK's pipeline. The current MPI extensions to VTK only parallelize individual filter modules. Their system shows reasonable, but sub-linear, scaling. What is more promising in this work is that VTK is a popular, portable package for scientific visualization, with support for a large number of rendering modalities.

Wald *etal.*[5] discuss the parallelization of a highly optimized ray-tracer on a cluster of PCs. Scenes with millions of triangles can be ray-traced at several frames per a second. They show linear speed-up up to 16 nodes, at which point the network connection on the display node becomes saturated. They observe that this configuration can compute 5 millions rays per a second. The

scene data is replicated on all nodes; thus, the system does not scale with respect to data.

2.2. Image-Based Rendering Techniques

Light-Field Techniques. There are several techniques that can create a novel view of an object of scene from a set of pre-computed, pre-recorded views. Gortler *etal.*[6] and Levoy *etal.*[7] are among the first to demonstrate the efficacy of these "light-field" techniques (so called because they attempt to represent all of the possible light rays in a scene, a 5D function). Both techniques use simplifying assumptions to reduce the dimensionality of the function and make the problem tractable. The images can either be synthetic or of a real-world object. A new image can be generated for a new viewpoint by blending images from near-by viewpoints. Both techniques can easily be implemented on contemporary graphics hardware. However, neither technique is able to handle large differences between images, as these techniques are intentionally oblivious of the underlaying representation. The resulting images can be blurry.

Layered Images. Several software techniques have been designed decompose a scene into layers, then display the layers from viewpoints close to the original. Mueller *etal.*[8] allow transparent volume visualization, but must keep track of all composited depth values, so the system can place the layers such that no gaps appear when viewing the layers from new viewpoints. Schaufler[9] uses multiple layers to render fully opaque models. Gaps are avoided by overlapping the spatial extents of the layers such that the images overlap by several pixels. Layers are re-rendered just before a gap is predicted to appear. His prediction mechanism only characterizes error with respect to camera translations and not with camera rotations about the model. Shade *etal.*[10] discuss techniques to use layers of images (with color and depth) to approximate complex objects. Images are reprojected on a pixel-by-pixel basis and require a well-defined depth value; this this technique is useful only for fully opaque objects.

Impostors. A significant amount of work[11–15] has been done on the use and pre-processing of impostors for viewing extremely large CAD datasets. Typical applications are architecture walk-throughs, either of individual buildings, or of whole cities or city districts. Significant preprocessing effort (with user assistance) is required to find good locations to place impostors and to segment the model, with respect to the impostor's location, to near and far sets. Far geometry is rendered and cached with that impostor. At run-time, near geometry is rendered directly and the far geometry is approximated with the cached imagery. These datasets are static and are intended to be visualized many times, so it is reasonable to spend a large amount of preprocessing time to accelerate this rendering.

Multi-Resolution Data Structures

Octrees. Shekar *etal.*[16] discuss an octree-based iso-surface decimation approach. A seed-cell and surface tracking technique is used to extract the initial surface. The geometry is then progressively decimated and stored at different levels of the octree. They also describe how to patch the surface between different levels of detail without cracks or introducing more triangles.

LaMar *etal.*[17, 18] introduce multiresolution volume visualization and multiresolution cutting plane techniques. The data volume is first decomposed by an octree into a hierarchy of approximations. The original data is stored at the leaf nodes, and approximations are stored in the interior nodes. The root node contains the coarsest approximation. To generate an approximation to render, the octree is traversed from root to leaf nodes, adding nodes to the approximation, until error criteria or rendering budgets are met. The approximation is then rendered. These techniques only work with rectilinear datasets.

Adaptive Mesh Refinement (AMR). AMR[19] is a technique used in CFD (computation fluid dynamics) simulations where interesting phenomena vary significantly in spatial extent. Compared to "regular" refinement approaches (*i.e.*, octrees), AMR more closely resembles a "soup of nested grids." This "soup" is refined around regions of interest and coarsened elsewhere. This refinement/coarsening can occur each time step.

Bethel *etal.*[20] discuss the Visipult system, a distributed volume visualization system for time varying AMR datasets. Imagery is produced at the brick-level and is, therefore, not independent of the data decomposition.

Weber *etal.* discuss iso-surface extraction[21] and volume visualization[22] on AMR grids, with the emphasis of maintaining C^0 continuity across boundaries of bricks of different resolutions. For iso-surfacing, continuity requires computing a dual grid of the AMR grid, computing a set of "stitching" cells, then extracting geometry from the grid voxels and "stitching" cells. A similar technique is used for volume visualization, but without explicitly constructing of a dual grid of "stitching" cells.

Specialized Hardware

Numerous specialized hardware solutions have been proposed to parallelize rendering of large datasets or to drive large display walls.

There are several research projects[23–25] that have explored using COTS (common off the shelf) desktop machines with specialized interconnects to improve rendering speeds. The first two use a $M \times N$ cross-bar switch to connect M PCs to N display devices. The latter connects N machines in a daisy chain. The first ([23]) has a nice notion of adaptivity with respect to image size and location on the output device, but is only a software simulation. The latter two have a very limited notion of adaptivity with respect to image size and location. All of these can scale if only one of render size (M), display size (N), or data size is scaled up, but to not scale if all are increased.

The SGI Origin SMPs (symmetric multi-processor) systems, equipped with InfiniteReality[26] rendering engines, provide a general solution to parallel rendering. While these machines have a extremely fast, general purpose interconnect fabric, the InfiniteReality engine is significantly out-performed by newer cards, and the Origin can hold only a a limited number of InfinteReality engines. The PixelFlow[27] is a very specialized machine that uses a deeply pipelined compositing network connected to a set of rendering engines. Data is distributed over the rendering engines, which compute a full frame, and ship the frame to the compositing network.

All of the hardware techniques show reasonable speed-ups for small numbers of rendering engines, displays, and datasets, but extremely expensive and very specialized. These systems have an explicit notion of frames and are not tolerant of delays or stalls in rendering engines. Two recent commercial products, SGI's InfinitePerformance[28] and HP's SV6,[29] use a larger number of rendering nodes, connected by a compositing network. While neither are COTS, both are more commodity-and component-oriented than prior offerings by SGI and HP.

Other Related Techniques

The Tapestry project by Simmons *etal.*[30] renders a scene by drawing a set of gouraud-shaded triangles where the color and depth of the vertices are calculated by a ray-tracer. Triangles are refined if they are physically large or have large changes in color or depth. As the user moves, sample points (vertices) that become occluded are removed from the mesh. Their technique does not handle transparent volumes as it assumes opaque surfaces and maintains only one layer of sample points.

The MLIC Technique

While our technique relies on a set of parallel processes to quickly produces images, it does not require an explicit notion of frames, thus avoiding the communications loading of the compositing phase of traditional parallel rendering techniques. Our technique also uses impostors - cached images. However, it only caches runtime generated imagery - there are (and can be) no preprocessed images. The rendering engines can use any form multiresolution representation or any kind of rendering technique "under the hood" without affecting the design or operation of any other component of the system, as the interface is extremely simple. The image database is a hierarchical data structure, but we use the term "multi-layered" to differentiate it from multiresolution rendering or datasets. Our system requires not rely on any special hardware, outside of a generic parallel machine, so it does not have the cost consideration of specialized hardware solutions. However, the faster the network and processors, the better our system will fair. Our system requires no preprocessing of the data, nor generation of any imagery before exploring a dataset. All cached imagery is imagery produced during the exploration of the dataset.

3. THE MLIC SYSTEM

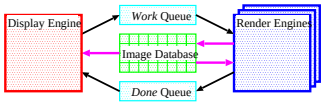


Figure 1. The conceptual MLIC system structure.

Figure 1 shows the conceptual structure of our MLIC system. The display engine (shown in red) displays images from the image database (shown in green). It sends requests for images via the *work* queue (shown in cyan) to the render engines (shown in blue). Upon completing an image, the render engines sends the image to the image database, and sends an acknowledgment via the *done* queue (also shown in cyan). The purple lines show image movement, and the black lines show movement of requests and acknowledgments.

3.1. Foundations

Understanding the design of the MLIC requires understanding the method and reason for decomposing the space about the view-point. We draw a distinction between images and k-D nodes, how the image database and the k-D tree decomposition of space are traversed, and the operations required to maintain both.

3.1.1. View Dependent Space Decomposition

One issue with prior techniques is that densely placed images that move with respect to the view-point commonly experience occlusion artifacts. Images that are supposed to be adjacent to each other can appear to move apart, allowing a hole to form where structures that are supposed to be hidden are exposed, or adjacent images move over and occlude each other, as shown in Figure 2. While these problems have been addressed

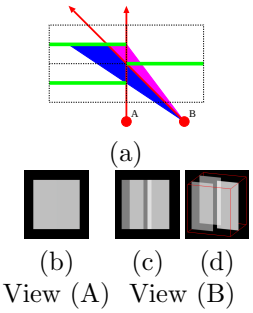


Figure 2. Occlusion artifacts occur when images generated to be correct from view-point *A* are viewed from view-point *B*. In (a), view-points are red, images are green, the hole is blue, and the occlusion is magenta. (b)-(d) shows (a) from the view-point; the two left images have opacities of 50% (which composite to opacity of 75%), and the right image has an opacity of 75%.

in a very basic way for opaque structures, there are no prior techniques to handle transparent structures.

Our solution is to decompose the space around the view-point into a set of convex, non-occluding (with respect to the view-point) polyhedra, which are then refined by a k-D tree (see Samet[31]) to allow for adaptivity. The k-D tree based refinement of the polyhedra is constrained to avoid these occlusion artifacts. Images are placed at the nodes of the k-D tree and move with the view-point.

The implementations discussed in this paper uses a cube, centered about the view-point, decomposed with six pyramids, where the tops meet at the view-point and the bases form the six faces of the cube. Other configurations are possible. Figure 3 shows a cube, centered around the view-point (shown as a red disk) decomposed into six polyhedra. One polyhedra (shown in purple) is then refined twice by a k-D tree. The tree on the right shows the parent-child relationships between the nodes.

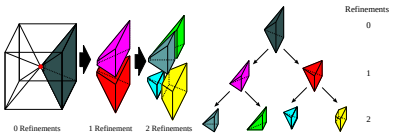


Figure 3. Example of the decomposition of a cube about the view-point (the red circle) into six polyhedra (pyramids). (a) shows the spatial decomposition, while (b) shows the k-D tree representation of the decomposition. The gray pyramid is refined twice.

The images are placed at fixed positions with respect to the view-point, so must be re-rendered when the view-point translates; they don't need to be re-rendered if the view-point's orientation or view-frustum change.

Figure 4 shows a 2D and 3D example of a Multi-Level Image Cache implemented using a square/cube basis. For figure 4(a), the red disk at the center is the view-point. The red lines show the view frustum. Black lines show the base pyramid decomposing of the space; blue lines show the k-D tree refinement of each pyramid; and green lines show individual images. The images are shown slightly smaller than their physical extent (delimited by blue and black lines) to emphasize that they are independent entities. Note that the different pyramids have different levels of refinement. Figure 4(b) shows the pyramids in different colors, and scaled down by 10% to show the individual nodes. All pyramids have been refined twice.

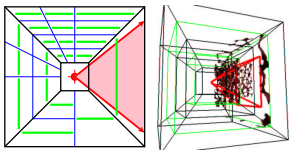


Figure 4. Example of a MLIC implemented (a) on a square in 2D and (b) on a cube in 3D. In both, the view-point and view-frustum is illustrated with red pyramid. Figure (a) shows that the different polyhedra can have different degrees of refinement by a k-D tree. Figure (b) shows all six polyhedra with one front/back refinement.

The distance to the outer face of a pyramid corresponds to the far-clipping plane of a viewing-frustum. Increasing and decreasing the distance to the far-clipping plane increases and decreases the spatial extent covered by the cache.

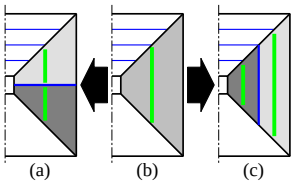


Figure 5. A k-D tree node can be refined along two orientations in 2D (three in 3D). (b) is the parent; (a) shows a left/right or top/bottom split, while (c) shows a front/back split.

The k-D tree refinement planes are either parallel to the cube face or pass though the origin of the cube, so there are no occlusion artifacts (compared to Figure 2); *i.e.*, images abut the boundaries of a k-D node and their end-points do not move with respect to the view-point. This is shown in Figure 5. Figure 5(b) shows the parent image to be refined; Figure 5(a) shows the refinement of the parent image parent image into left and right images. This can also be see in Figure 3(a) with the gray to red/purple split or the red to gray/green split. Figure

5(c) shows the refinement of the into front and back images (with respect to the view-point), also shown in Figure 3(a) with the red to cyan/yellow split.

Images vs. k-D Nodes. One distinction that we need to make between images and k-D nodes. While there is a one-to-one correspondence of images in the image database to nodes in the k-D tree decomposition of the polyhedra about the view-point, they are conceptual different entities.

The k-D tree decomposes space into a set of sub-regions; in our system, all of these regions are hexahedral trapezoids. The image associated with a node of the k-D trees contains the rendered view of the data enclosed by that k-D node. If the view-point translates, the k-D node is not changed, but the image may change. Also, if data viewing parameters are changed, *e.g.* the transfer function or iso-contour value, the image will change but the k-D node will remain unchanged.

Secondly, the display engine may only display a small portion of the images in the image database. While there might be a higher-resolution decomposition of some region, the display engine, for reasons of rendering budget, or some other, may choose not to show all of the images in the image database. At the same time, there is a difference between the refinement that the display engine performs to display some portion of the image database, and the refinement of the images in the image database itself: the former modifies what is displayed from, and the latter modifies what is contained in, the image database.

Image Reuse. Figure 6 shows where images would be reused after rotating the view-frustum. In Figure 6(a) the first frame, a set of images in rendered. The blue and green images are rendered for the first frame. In the second frame (Figure (b)), as the view-frustum turns right, an additional set of images in now visible. Those shown in green are rendered in both frames. Images shown in blue are not visible in the second frame, and may be deleted (if running out of cache space). The purple images in the second frame are now visible; they will be placed in a *work* queue to be rendered if the images are invalid. The middle row shows a rendering of a iso-surface rendering of a Trabecular bone dataset; the initial view-point position is shown in the left image (c) and the view-frustum has turned to the right in the right image (d). The bottom row shows a side view of the MLIC refinement, with the view-point and view-frustum pyramid shown in red. Image boundary color corresponds to the faces of the cube. Notice that the images outlined in cyan no longer appear in the right image, and new images appear on the right side of the image.

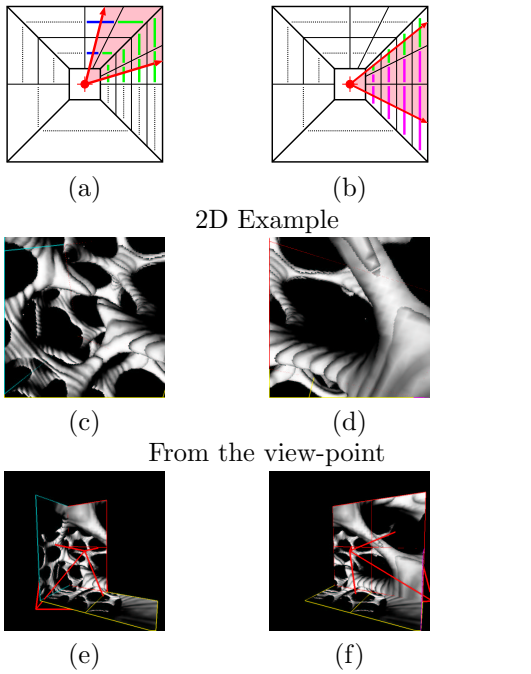


Figure 6. Reusing images as the view-frustum turns to the right.

All images have the same resolution. A k-D tree node is refined by replacing it with a left/right, top/bottom, or front/back child, where each child node's images contains a copy of the corresponding region of the parent node's image. The new images are marked for future re-rendering. Images are coarsened by removing the child nodes and replacing the parent's imagery by a filtered version of the child node's images.

Rotating the view-frustum direction does not invalidate any of the images, but newly exposed images may require rendering, and possibly refinement.

Figures 7 and 8 shows examples of refining the image database and k-D tree. Figure 7 shows three levels of refinement of the image database and k-D tree, demonstration left/right and top/bottom splits. The first column (Figures 7(a,d,g)) shows a single image with no refinement. The second column (Figures 7(b,e,h)) shows a left/right split, and the third column (Figures 7(c,g,i)) shows a further top/bottom split. The first row (images 7(a)-(c)) show the scene from the view-point. The second row (images 7(d)-(f)) show the scene from outside the view-point; the red lines show the borders of the k-D nodes and the images. The third row (images 7(g)-(i)) show the individual images in the image database. Notice that the sub-images of image 7(h) are stretched; this is because they cover less space left-to-right than top-to-bottom.

Figure 8 shows two levels of refinement of the image database and k-D tree, demonstration front/back

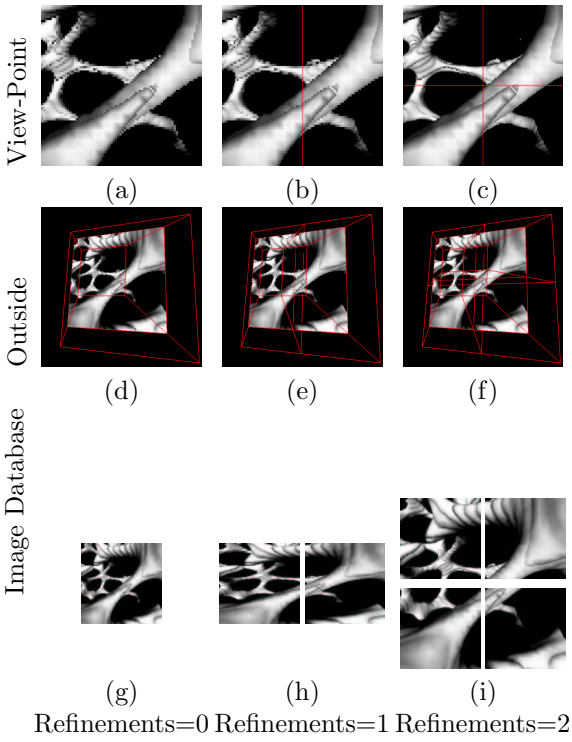


Figure 7. Three levels of refinement, showing left/right and top/bottom splits of the image database and k-D nodes.

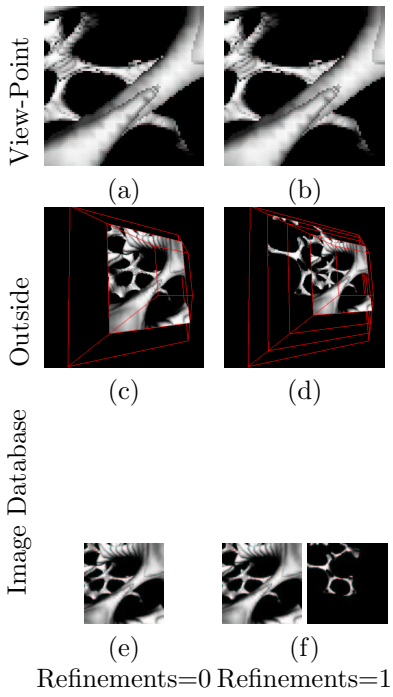


Figure 8. Two levels of refinement, showing a top/bottom split of the image database and k-D nodes.

split. The first column (Figures 8(a,c,e)) shows a single image with no refinement. The second column (Figures 8(b,d,f)) shows the front/back split. The first row (images 8(a)-(b)) show the scene from the view-point. The second row (images 8(c)-(d)) show the scene from outside the view-point; the red lines show the borders of the k-D nodes and the images. The third row (images 8(e)-(f)) show the individual images in the image database.

3.1.2. Displaying the Image Database.

The space about the view-point is decomposed by polyhedra, then refined by k-D trees. Displaying the image database involves visited all the polyhedra, and traversing the accompanying k-D trees, in an top-down manner (with respect to the k-D trees) and back-to-front (with respect to the view-point). At each k-D tree node, the image is examined; If the image is current, and it is too coarse, and it has children, then its children are visited. If the image is invalid, then the refinement stops. The image is displayed, and it is tagged to be *rendered*. If the imagery is invalid, and has children, and the children have current imagery, the image is drawn, and the image is tagged to be *merged*. If the imagery is current, and it is too coarse, it is drawn and it is tagged to be *refined*. If the imagery is current, and it is not too coarse, it is simply displayed.

As we noted earlier, the set of images displayed is potentially a much smaller set than exists in the image database. For example, if user is exploring a dataset and examining many small features without moving the view-point, once an image is rendered and cached, it does not need to be re-rendered. Thus, it is possible to display significantly more detail. However, if the user moves quickly to another, distant view-point, all of the images will become invalid, and it is better to use a smaller number of images, updated more frequently, to provide the user with proper feedback on their movement through the dataset.

3.1.3. Image Database Operations.

Three kinds of operations can be applied to images in the image database and nodes in the k-D tree: *refine*, *render*, and *merge*. The image-level operations are to create new images, either by re-rendering the data, or by reusing some previously computed images. The k-D tree level operations are to add or prune nodes from the tree.

The *refine* operation refines a region: it adds child nodes to the k-D tree node which represents that region, allocates images the image database, and computes images for the child nodes by using just the image of the parent node. Figure 3 shows this for the k-D tree: the initial gray pyramid is refined into red and purple nodes. A image can be split (with respect to the view-point)

into a left/right pair (Figure 3, red into cyan and yellow nodes), top/below pair (Figure 3, the gray into red and purple nodes), or front/back pair (Figure 3, purple node into light gray and green nodes). Since all images have the same number of pixels, when a left/right or top/bottom pair is refined, the effect is to double to number of pixels in the direction of the split.

Images are displayed in back to front order with blending to affect an composite operation (see,[32] the *OVER* operator). To break a image into a front/back image pair, we need to set the values in the front/back pairs such that when they are composited together, they produce the same result as displaying the parent image. Hence, to refine a image, we must compute an inverse to the compositing operation. The compositing operation is defined as follows:

$$\begin{aligned} K &= C_0 + C_1(1-\alpha_0) \\ \gamma &= \alpha_0 + \alpha_1(1-\alpha_0) \end{aligned}$$

Where $C_{1,2}$ and $\alpha_{1,2}$ are the input opacity-weighted color and opacity values, respectively, and K and γ are the output opacity-weighted color and opacity values, respectively. Note that the compositing operation performs the same calculation on each of the red, green, and blue channels. K and γ correspond to the color and opacity of the parent image, and $C_{1,2}$ and $\alpha_{1,2}$ correspond to the color and opacity of the child images. We also simplify by assuming the front/back imagery is the same, e.g., $C = C_1 = C_2$ and $\alpha = \alpha_0 = \alpha_1$.

Solving for C, given K:

$$K = C + (1-\alpha)C = C(2-a)$$

thus

$$C = \frac{K}{(2-\alpha)}$$

And solving for α, given γ:

$$\gamma = \alpha + \alpha(1-\alpha) = 1 - (1-\alpha)^2$$

thus

$$\alpha = 1 - \sqrt{1-\gamma}$$

The *merge* operation coarsens a region by collapsing a left/right, top/bottom, or front/back pair into their parent node, and to create a lower-resolution image for their parent node. This is used when the view-point has not changed, but the region is less important (*e.g.*, the user has turned the view frustum away). The left/right and top/bottom pairs are produced by low-pass filtering or sub-sampling, depending on which filtering mode used in the original rendering. Merging a front/back pair is simply compositing the front and back pairs together.

The *render* operation generates an image from the data contained within the spatial extent of a k-D node. The render engine is given a dataset at startup, and receives requests to render specific nodes. At the end of rendering, the image is sent to the image database.

4. SYSTEM IMPLEMENTATION

We have implemented two versions of our MLIC system. The first is designed to run on a SMP (Symmetric Multi-processor) machine using shared memory for all communication and the image database. The second version is designed to run on a large distributed-memory parallel machine using MPI (Message Passing Interface)[33] for communication.

There are several engines in these two systems, most of which are used in both versions. The fundamental difference between the distributed, message passing version and the shared memory version, is the "dispatch engine", a module for interfacing with MPI.

All of the engines are written to be independent modules, and are not aware of running on shared or distributed memory machines. We will briefly discuss the individual engines before discussing the shared memory and message passing versions of MLIC.

4.1. Shared Memory

Shared memory is implemented through the Unix *mmap* function call. On 32-bit machines, the maximum size of a shared memory segment available through this mechanism is 2GB. We have not found this limitation to be a problem.

4.2. Image Database

```
enum { TileSize = 128 };
struct Pixel { unsigned char r, g, b, a; };
typedef Pixel PxImg[TileSize][TileSize];
class KdNode
{
  public:
  BOOL iscopy;   // TRUE when refine or merge, FALSE when generate
  BOOL inqueue;  // TRUE when placed in done queue until remove work
  int nodeid;    // Identity of this node
  int parentiid; // Identity of this node's parent node
  int leftid;    // Identity of this node's left child node
  int rightid;   // Identity of this node's right child node
  int kd_dir;   // The X/Y/Z direction of the split
  int depth;     // The depth of this node in the tree
  float pos[3];  // The view-point where the image was rendered
  PxImg image;   // The Image
};
KdNode ImageDb[];
```

Figure 9. The Image Database is a large array of *KdNode* structures.

The *image database* is shown in figure 9, and is stored in the shared memory segment. The first N positions of the database are the root nodes of the N polyhedra that decompose the space about the view-point. Since the current implementation uses a cube, the first six position are the root nodes for the six faces of the cube. If we were to use another basis, say an octohedron (which would be decomposed into eight tetrahedron), the first eight positions would correspond to the eight tetrahedron.

The *inqueue* flag is set to true when a image is first added to the *work* queue, and set to false when it is

removed from the *done* queue. This flag used is to prevent a image from being added more than once to the queue, but can be used to monitor progress of the systems. For example, the images can be outlined with different colors to reflect this flag.

The *iscopy* flag is set to true if the image was generated through a *refine* or *merge* operation, and false if the image was generated by a *render* operation. If the display engine encounters an image with the *iscopy* flag set to true, the image is displayed, but the render engine adds the image to the *work* queue, requesting a *render* operation on the image.

The *nodeid*, *parentid*, *leftid*, and *rightid* fields are the identifiers (array subscripts) of the current node, it's parent node, and it's child nodes. The *parentid* field is zero for a root node, and *leftid* and *rightid* fields are both zero for leaf nodes.

The *pos* field records the location of the view-point when the image was rendered. This field is currently used to test if the image is *current*, *i.e.*, *pos* == current-view-point, or *old*, *i.e.*, *pos* != current-view-point. A mechanism to prioritize updating images could use the difference between this field and the current view-point to weight the priority.

4.3. *Work* and *Done* Queues

The *work* and *done* queues are implemented as fixed-sized circular queues, and are stored in the shared memory segment. Images that are tagged for one of the maintenance operations are placed (rather, a pointer to the image) in the *work* queue. When the operation is complete, the image(s) are placed in to *done* queue. Engines that access these structures attach to this shared memory segment.

4.4. Engines

There are three basic engines in the MLIC system: *display*, *render*, and *dispatch*. The first two are found in both implementations, while the latter is only found in the distributed, MPI version.

4.4.1. Display Engine

The display engine displays images, and maintains all fields, except for *image*, in the image database. Image operations are requested by the display engine by placing the requests into the *work* queue (and setting the image's *inqueue* flag). When the acknowledgments are received, *i.e.*, read from the *done* queue, the display engine updates the *iscopy* and *inqueue* fields.

We have not separated the image database maintenance functionality into a separate process, as the work is very limited. When implementing the shared memory version, we moved this functionality to the display engine because is significantly eased the contention on the image database structures. The display engine never modifies images, and the render engines never modify any but the images.

4.4.2. Render Engine

The *render* engine performs the image operations and does not modify (in the SMP version) any other fields in the image database. It reads requests from the *work* queue, performs the operation, writes the image(s) to the image database, and send an acknowledgment via then *done* queue.

4.4.3. Dispatch Engine

The dispatch engine communicates via the shared memory queues with the display engine and via MPI messages with the render engines. It maintains a queue of currently available render engines. It reads requests from the *work* queue, selects and removes the first entry in the available render engines queue, and sends the render request to that engine. When the acknowledgment comes back, it writes the acknowledgment to the *done* queue, and the image(s) to the image database, and add the render engine back to the available render engines queue. All of the MPI messages sent and received are done asynchronously since the render engines can service the requests at different rates.

4.5. Shared Memory Implementation

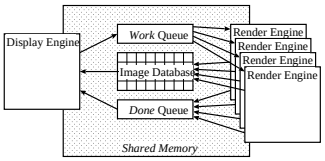


Figure 10. Shared memory architecture.

The shared memory version of MLIC uses a large, shared memory segment that include the image database and the *work* and *done* queues. The display engine and render engines all attach directly to this shared memory segment; see figure 10.

The engines are written to be processes, not threads. There is no explicit notion of assignment of engine processes to particular processors; it is up to the OS load-balancing mechanism to spread the processes out as appropriated.

4.6. MPI Implementation

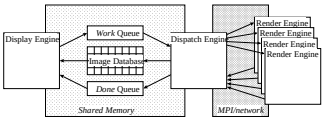


Figure 11. MPI architecture.

The MPI version of MLIC also uses a large, shared memory segment for the image database and *work* and *done* queues. However, there is a third, "dispatch" engine, that moves the requests and acknowledgments from the *work* and *done* queues and the MPI message queues; see figure 11. The display engine is the same code for the shared memory and MPI versions of MLIC.

5. MLIC SYSTEM PERFORMANCE

We have tested the shared memory and MPI versions of our MLIC on two different kinds of machines.

5.1. Test Platforms

5.1.1. Origin3000.

The Origin3000 is a SMP (symmetric multi-processor) machine from SGI, with NUMA (non-uniform memory access) shared memory. The system we used has 48 250MHz MIPS R10K processors and 40GB of memory. The image database and communications mechanisms reside in a large shared memory segment (implemented via *mmap*), with direct access by the viewer and renderer processes. The images are rendered, then copied directly to the image database. We use the software-based ray-casting iso-surfacing and volume visualization engines on this platform, as there are too few graphics pipes to perform a realistic scalability study using them. We report only basic scalability studies for this machine. This simpler implementation allowed us study the behavior of the MLIC system and interaction of the components without the complexity and overhead of message-passing and synchronizing distributed processes. We have tested only the shared memory version of MLIC on this system.

5.1.2. Linux Cluster.

Our cluster is a set of 64 Linux boxes running the LLNL CHAOS Linux kernel.[34] Each box has two 2.5GHz Pentium Xeon processors, 2GB of memory, a nVidia GeForce ti4600 graphics card, and a single Quadrics[35] Network Card. The interconnect fabric is a two-level Quadrics network. The bidirectional bandwidth claimed by the manufacturer is 340MB/sec. We have tested only the MPI version of MLIC on this machine.

5.2. Dataset

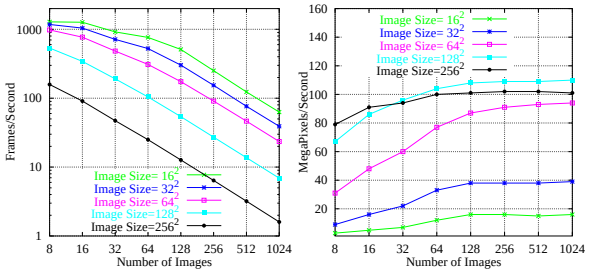
5.2.1. Trabecular Bone Dataset.

We use a Trabecular bone dataset for our scalability studies. It has the nice property of being fairly "open," in that uniformly covers the full extent of the volume. Also, we can visually confirm that (1) the decomposition of data space is correct, and (2) the current status of individual images. The dataset is an extremely high-resolution scan of the spongy material inside of bones; the original 540^3 dataset is $1\,cm^3$. We use 256^3 and 512^3 versions of this larger volume.

5.3. Scalability Study.

We study the absolute performance and speed-up characters of the MLIC system on the Origin3000 and Linux cluster, and we show the effect of image size vs. the number of images displayed on display performance. We measure the sustained communications bandwidth of the Linux cluster. We show brief results for the Origin3000 and compare some of these results against the Linux cluster. We show the effect of image size and the number of processors on image rendering performance on the Linux cluster.

5.3.1. Display Rates



(a) Frames/Second (b) Mega-pixels/Second

	Image Size									
	16^2		32^2		64^2		128^2		256^2	
Images	Frame Rate	Fill Rate	Frame Rate	Fill Rate	Frame Rate	Fill Rate	Frame Rate	Fill Rate	Frame Rate	Fill Rate
8	1282	3	1177	9	980	31	535	67	158	79
16	1266	5	1042	16	769	48	343	86	90.9	91
32	917	7	714	22	483	60	191	96	47.2	94
64	763	12	526	33	310	77	104	104	25.0	100
128	510	16	302	38	174	87	54.0	108	12.7	101
256	251	16	154	38	90.9	91	27.2	109	6.4	102
512	124	15	76.3	38	46.5	93	13.7	109	3.2	102
1024	63.0	16	39.1	39	23.5	94	6.9	110	1.6	101

(c)

Figure 12. Display Rates as a function of total number images displayed and image size. The window size is 500^2 pixels. The limiting factor is fill-rate, at about 110 mega-pixels per a second.

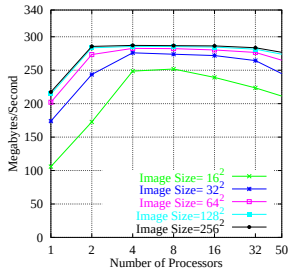
Figure 12 compares the rates for rendering 8 to 1024 images for images of size 16^2 to 256^2 pixels on a window of 500^2 pixels. The images are downloaded as textures, then mapped to a polygon for rendering. Nearest-neighbor filter was used, and mip-mapping was turned off. The "Frame Rate" column reports the number of complete frames rendered per a second. The "Fill Rate" column reports the pixel fill rate in mega-pixels per a second, which is the number of images per a second times the number of pixels per an image. The pixel-fill rates is the best measure of the render capability of the display engine. The numbers reported here, however, reflect both download *and* fill rates because we want measure the worst-case performance.

The fill rate tops out at about 110 mega-pixels per a second. The optimum size tile seems to be 128^2 pixels, and not 256^2. The decrease at 256^2 has to do with

the ratio of pixel to texel size: on contemporary graphics cards pixel-fill performance decreases quickly as the texel size becomes smaller than the pixel size. The images cover fairly small portions of the window, so a 256^2 image has very poor cache locality.

Our current implementation of the display engine does not cache the textures on the graphics card (*e.g.*, it does not use OpenGL's texture-object functionality). They are downloaded each time that they are displayed.

5.3.2. Total Communications Bandwidth on the Linux Cluster.



(a)

Image Size	Number of Processors						
	1	2	4	8	16	32	50
16^2	39	74	106	115	110	95	54
32^2	106	173	249	252	239	224	206
64^2	174	244	276	274	272	264	237
128^2	202	273	283	282	280	277	260
256^2	214	283	286	285	285	281	271
512^2	217	286	287	287	286	284	274

(b)

Figure 13. Total Communications Bandwidth on the Linux Cluster as a function of image size and number of processors. The numbers shown are Megabytes per second.

This test measures the total effective bandwidth of the Linux cluster for different image sizes and numbers of processors, given the communications behavior of the MLIC system. The observed peak rate of 285MB/second, shown in figure 13, and is very close to the manufacturer's claimed peak rate of 340MB/second.

The test is set up to have the display engine request the rendering of $10,000$ images. The rendering engines receive the requests, and immediately send back an empty image. On average, each rendering engine will receive $10,000/N$ requests, where N is the number of processors. Each image contains 16 bytes of header and a RGBA image, for a total size of $(imagesize)^2 \times 4 + 16$ bytes. The Megabytes per second was computed by dividing the total number of bytes sent (total size $\times 10,000$), by the elapsed time from the first request sent by the display engine to the last image received by the display engine. This factors in all of the overhead of MPI, and the sending of the request messages. We do not, however, count the number of bytes in the render request structures because the number of images per second that could be sent *back* to the image database is the only figure of interest.

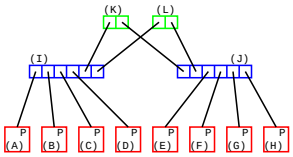


Figure 14. A possible Quadrics network configuration, with eight processors and two levels of switches.

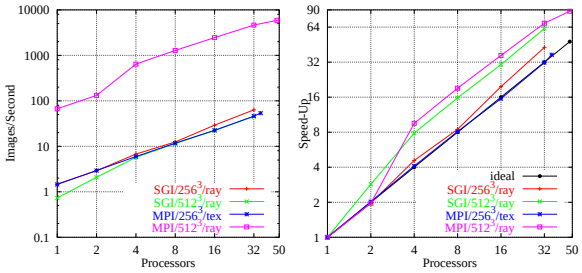
The reason that the total capacity decreases from 32 to 50 processors is congestion. However, understanding this congestion requires understanding the network topology of the Linux cluster. The cluster uses a two-level Quadrics switch network (see[35]), with eight switches at the highest-level, connected to eight low-level switches (with 8 links to processors and 8 links to high-level switches), for a total of 64 nodes (and 128 processors).

Figure 14 shows a simplified network with eight processors (A-H, in red), two low-level switches (I & J, in blue) and two high-level switches (K & L, in green). Links are shown as black lines. The low-level switches have six ports, four to processors, and two to high-level switches. Any packet send between nodes on different low-level switches must pass through the high-level switches. For example, if (A) sends a packet to node (E), it must pass through both low-level switches (I & J) and one of the two high-level switches (K or L). However, if node (A) sends a packet to node (B), the packet must only pass through the (low-level) switch (I).

Processors are allocated sequentially to a job. Requesting five processors (see Figure 14) will result in four allocated processors, (A) to (D) on the low-level switch (I), and the fifth, (E), on the second switch (J). If processor (A) is running the display engine (and the others are running render engines), then only images from the fifth processor (E) must travel through a high-level switch (K or L) back to the display engine on (A). Images sent from rendering engines on processors (B) to (D) must only pass through the low-level switch (I). If the display engine is running on the fifth processor, (E), and processors (A) to (D) are running render engines, then all of the images sent from processors (A) to (D) must all travel through the two high-level switches (K) or (L). However, since there are only two high-level switches, only two render engines can send images at a time.

5.3.3. Comparing the Origin3000 vs. the Linux Cluster.

This study compares the absolute render rates and speed-ups on the Origin3000 and Linux cluster. We con-

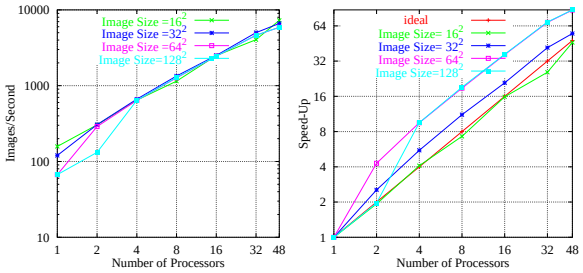


(a) Images/Second (b) Speed-up

	(A) SGI/256^3/ry		(B) SGI/512^3/ry		(C) MPI/256^3/tx		(D) MPI/512^3/ry	
Procs	Img/s	S-U	Img/s	S-U	Img/s	S-U	Img/s	S-U
1	1.48	1.00	0.73	1.00	1.45	1.00	67	1.00
2	2.91	1.97	2.09	2.86	2.93	2.02	132	1.96
4	6.76	4.57	5.74	7.86	5.95	4.10	641	9.54
8	12.5	8.47	11.5	15.8	11.8	8.11	1286	19.2
16	29.2	19.7	22.3	30.5	22.5	15.5	2456	36.6
32	63.3	42.8	45.4	62.2	46.0	31.7	4637	69.1
36					53.7	37.1		
50							5885	87.6

(c)

Figure 15. Comparing the Origin3000 vs. the Linux cluster showing images generated per a second and speed-ups using two Trabecular bone datasets. Image Size is 128^2.



(a) Images/Second (b) Speed-up

	Image Size							
	16^2		32^2		64^2		128^2	
Procs	Img/s	S-U	Img/s	S-U	Img/s	S-U	Img/s	S-U
1	158	1.00	120	1.00	67	1.00	67	1.00
2	304	1.93	306	2.54	289	4.29	132	1.96
4	644	4.09	665	5.53	641	9.51	641	9.54
8	1148	7.28	1341	11.2	1268	18.8	1286	19.2
16	2499	15.9	2508	20.9	2450	36.3	2456	36.6
32	4049	25.7	5010	41.7	4625	68.5	4637	69.1
48	7272	46.1	6619	55.0	5895	87.4	5885	87.6

(c)

Figure 16. Scalability as a function of Image Size vs. Number of Processors collected over the last minute of a four-minute run. (a) shows the rate at which images are processed per a second. (b) shows the speed-up. (c) shows the raw numbers.

ducted only basic scalability studies on the Origin3000.

Figure 15, columns (A) and (B), shows results using software, iso-surface, ray-casting on the Origin3000, for the 256^3 and 512^3 Trabecular bone datasets, respectively. Figure 15, column (C), shows results of hardware texture volume visualization using the 256^3 dataset, while column (D) shows the results of software, iso-surface ray-casting using the 512^3 dataset, on the Linux cluster. The image size for all of these runs is 128^2 pixels.

Notice that for both runs with dataset size of 256^3 exhibit linear scalability, while both with dataset size of 512^3 exhibit super-linear up to four processors, and linear thereafter. We are not sure why the super-linear behavior occurs. Since each processor in both implementations have their own, local copies of the dataset, there should be no super-linear scaling due a distributed dataset fitting into memory caches.

Notice that the two MPI studies use more than 32 processors, but different in the number. This variation is due to the stability and availability of the Linux cluster.

5.3.4. Scalability as a function of Image Size vs. Number of Processors.

This test examines the sustained rate of image generation and transmission on the distributed version of the MLIC implementation, see figure 16. The numbers reported are the sustained rates over the last minute of a four-minute run. The reason that we only measure the last minute is to let the system settle into normal operation. The system takes a minute or two to refine the images to a point where we can reasonably measure of a one-processor run versus a fifty-processor run. Under normal operations, a user would not immediately ask for extremely high-resolution imagery, but only ask for a few images, which can be rendered very quickly. Notice that all the runs follow the same basic linear curve. Several of the runs are super-linear up to 4 or 8 processors (rendering engines), and are linear there after. We do not have a good explanation for this apparent super-linear speed-up; however, it seems to be a consistent trend here and in experiments discussed above.

Notice that the capacity of the system does not scale with respect to the image size. This shows that the overhead of the other operations (image read-back, communication, etc.) are large, and are the dominate cost for small image sizes.

Also notice that the speed-up from 32 to 50 processors, while linear, has a smaller slope than the speed-up from 4 to 32 processors. This can be explained by observing that there is a general reduction in bandwidth, starting at 32 processors, as discussed in section 5.3.2.

5.3.5. Data Scalability

Our second test (see section 5.3.3) shows that there is basic scalability on the size of the data. However, both datasets are too small, in general, and our tests not comprehensive to make any strong statements about the scalability with respect to dataset size. These two

datasets are simply ordered by X, Y, then Z, so extremely large datasets will start to show sub-linear performance due to poor memory locality.

5.4. Rendering Requirements vs. Capacity

How many processors and what network bandwidth is necessary? Rather than just discuss the performance of the system from the back-end forward, what are the requirements at the front-end and does the back-end meet them?

If we assume a window of 1024^2 pixels, driven by a display engine, how many images it is reasonable to display? If we use a image size of 128^2 pixels, the window could be tiled by a 8×8 array of images. If, however, we wish to avoid aliasing artifacts, we should use a 4×4 array of images, where each pixel in an image projects to (at least) a 2×2 array on pixels on the window. This means that the largest dataset that should be displayed on a 1024^2 window is 512^3. Notice that we do not say $512^2 \times X$, where X is very large. If a dataset can be viewed from any arbitrary direction, then the largest dimension of the dataset must be 512 or less to satisfy the sampling requirements. This is also not to say that a larger dataset cannot be handled – only that the *currently displayed* region of the dataset cannot be over 512^3 voxels.

If we assume that an image is generated from a cubic subregion of the dataset, then a 512^3 dataset is covered by a $4^3 = 64$ array of images.

If we want each image to be refreshed 30 times a second, then the system must be able to process $30 \times 64 = 1920$ images per a second. From section 5.3.4, this can be met by only 16 processors.

If we allow aliasing artifacts, we then use $8^3 = 512$ images, or $512 \times 30 = 15360$ images per a second. This cannot be met by any configuration, but if we reduce the frame rate to 10 fps, or 5120 images a second, this can just be met.

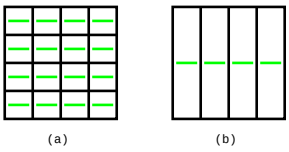


Figure 17. Rendering slabs instead of cubes to reduce the number of images rendered.

Another way to reduce the image requirements and keep the higher display frame rate is to render slab instead of cubes – render very deep regions, with respect to the view-point. For example, see figure 17: instead of rendering a 512^3 dataset with sixteen 128^2 pixels images that represent the sixteen 128^3 blocks of the larger dataset (Figure 17(a), simply render the same 128^2 image, but covering the full depth of 512 (Figure 17(b)).

5.5. Bottlenecks

There are two points of limited bandwidth. With a network bandwidth capacity of 285MB/second, the maximum number of pixels received a second at the display engine is 72.12 megapixels/second. The rendering speed tops out a 110 megapixels/second, which is significantly more than the network's 71 Mpx/second ($= 285MB/s/RGBA$). This is a fairly simplistic distillation of the system bottlenecks, but it is clear from our studies that there are aspects of the system performance that we do not fully understand.

5.6. Observations and Synthesis

The core set of tests show that the system is scalable. However, it is clear that there is a limit to the scalability due to the configuration and capacity of the Quadrics network on our Linux cluster.

For the distributed, MPI, implementation, the primary limitation is the bandwidth to the image database from the rendering engines. The image database is stored in the same machine as the display engine, so all image traffic arrives over a single network connection. There are many possible solutions; the two at the top of our list are compressing the image stream, and using a distributed image cache. Our first solution is compressing the image stream. While compression is not scalable in the strict sense of the word, it should reduce the traffic by a reasonable percentage. The question is how keep the extra computational effort from taking more time than transmitting the uncompressed image itself. Careful consideration will be necessary to make this efficient. Our second solution is to use a distributed image cache. That is, use multiple meta-display engines, each with its own image database. Render engines send images to one (possibly, more) of these meta-display engines. These meta-display engines when send a partially composited result to the final render engine. Several issues arise: (1) How does one decompose the image space to limit the number of different display engines that an image must be sent to? (2) How does one decompose the space about the viewpoint to balance the load on the display engines. (3) How efficiently can the output of these separate display engines be sent to a meta-display engine; that is, how does one minimize the latency to the final display.

The rendering engine will be reworked to allow exploring larger datasets, at which point we will conduct a set of tests to measure scalability of MLIC with respect to data. However, as discussed above, we do not believe that there will be any problems or surprises in accomplishing this.

6. CONCLUSIONS AND FUTURE DIRECTIONS.

Conclusions. We have introduced our Multi-Level Image Caching system as an approach for scaling generic

rendering on a parallel, distributed computing system. The MLIC system maintains a set of images for interactive display while a set of parallel engines render new images. Our system has successfully decoupled rendering rates from display rates, allowing for fairly slow image generation, while providing interactive display of imagery. The system experiences linear scaling on an Origin3000 SMP using 32 processors and on a Linux-cluster using 50 nodes.

Future Directions. We plan to incorporate multi-resolution render of multi-resolution data sources into the rendering engine, extending the work by Pascucci *etal.*.[36] Their tests show good render rates for for datasets up to 2048^3 voxels. Their system, in theory, should allow rendering of datasets much larger than 2048^3.

We plan to implement a priority-based scheduler that incorporates error- and view-driven criteria, and temporal prediction and approximation for time-varying data, extending work by Nuber *etal.*.[37, 38] We plan to augment the system to handle multi-resolution, time-varying datasets, and to explore the use of a distributed image cache on distributed memory machines to ease the bandwidth requirements to the image database node. We plan to explore the use of compression and distributing the image database in situations where the system is communications-bound.

Acknowledgments

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes. This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

REFERENCES

1. K. Ma, J. Painter, C. Hansen, and M. Krogh, "Parallel volume rendering using binary-swap compositing," *IEEE Computer Graphics and Applications* **14**(4), pp. 59–68, 1994.
2. S. Parker, M. Parker, Y. Livnat, P. Sloan, C. Hansen, and P. Shirley, "Interactive ray tracing for volume visualization," *IEEE TVCG* **5**, pp. 238–250, /1999.
3. J. Ahrens, C. Law, W. Schroeder, K. Martin, and M. Papka, "A parallel approach for efficiently visualizing extremely large, Time-Varying Datasets," tech. rep., LANL, 2000.
4. Kitware. Kitware, Inc. VTK, the Visualization Toolkit, http://www.kitware.com.
5. I. Wald, C. Benthin, P. Slusallek, T. Kollig, and A. Keller, "Interactive Global Illumination using Fast Ray Tracing," in *Proceedings of the 13th Eurographics Workshop on Rendering (RENDERING TECHNIQUES-02)*, pp. 15–24, Eurographics Association, June 26–28 2002.
6. S. Gortler, R. Grzeszczuk, R. Szeliski, and M. Cohen, "The Lumigraph," in *Siggraph 1996*, pp. 43–54, Aug. 4–9 1996.
7. M. Levoy and P. Hanraham, "Light Field Rendering," in *Siggraph 1996*, pp. 31–42, Aug. 4–9 1996.
8. K. Mueller, N. Shareef, J. Huang, and R. Crawfis, "IBR-Assisted Volume Rendering," in *Hot Topics, Vis 1999*, pp. 1–4, Oct. 1999.
9. G. Schaufler, "Per-Object Image Warping with Layered Impostors," in *Rendering Techniques 1998*, pp. 145–156, 1998.
10. J. Shade, S. Gortler, L. He, and R. Szeliski, "Layered depth images," in *Siggraph 1998*, pp. 231–242, July 1998.
11. M. Carrozzino, F. Tecchia, C. Falcioni, and M. Bergamasco, "Image caching algorithms and strategies for real time rendering of complex virtual environments," in *Afrigraph 2001*, pp. 65–74, Nov. 5–7 2001.
12. B. Chen, J. Swan, II., E. Kuo, and A. Kaufman, "LOD-Sprite Technique for Accelerated Terrain Rendering," in *IEEE Visualization 1999*, pp. 291–298, Oct. 1999.
13. G. Schaufler and W. Stürzlinger, "A three-dimensional image cache for virtual reality," in *Eurographics 1996*, 1996.
14. J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder, "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments," in *Siggraph 1996*, pp. 75–82, Aug. 4-9 1996.
15. X. Decoret, F. Sillion, G. Schaufler, and J. Dorsey, "Multi-layered impostors for accelerated rendering," *Computer Graphics Forum* **18**, pp. 61–73, Sept. 1999.
16. R. Shekhar, E. Fayyad, R. Yagel, and J. Cornhill, "Octree-Based Decimation of Marching Cubes Surfaces," in *Visualization 1996*, R. Yagel and G. M. Nielson, eds., pp. 335–344, IEEE, (Los Alamitos), Oct. 27–Nov. 1 1996.
17. E. LaMar, K. Joy, and B. Hamann, "Multi-Resolution Techniques for Interactive Hardware Texturing-based Volume Visualization," in *IEEE Visualization '99*, pp. 355–361, 25-29 Oct. 1999.
18. E. LaMar, M. Duchaineau, B. Hamann, and K. Joy, "Multiresolution Techniques for Interactive Texturing-based Rendering of Arbitrarily Oriented Cutting-Planes," in *Data Visualization 2000*, pp. 105–114, 29–30 May 2000.
19. G. Bryan, "Fluids in the Universe: Adaptive Mesh Refinement in Cosmology," *Computing in Science and Engineering* **1**, pp. 46–53, Mar./Apr. 1999.

20. W. Bethel, J. Shalf, S. Lau, D. Gunter, J. Lee, B. Tierney, V. Beckner, J. Brandt, D. Evensky, H. Chen, G. Pavel, J. Olsen, and B. Bodtker, "Visapult - Using High-speed WANs and Network Data Caches to Enable Remote and Distributed Visualization," in *Super Computing 2000*, pp. 118–119, Nov. 4-10 2000.
21. G. Weber, O. Kreylos, T. Ligocki, J. Shalf, H. Hagen, B. Hamann, and K. Joy, "Extraction of Crack-free Isosurfaces from Adaptive Mesh Refinement Data," in *VisSym*, pp. 25–34, May 28–31 2000.
22. G. Weber, O. Kreylos, T. Ligocki, J. Shalf, H. Hagen, B. Hamann, K. Joy, and K. Ma, "High-quality Volume Rendering of Adaptive Mesh Refinement Data," in *Proceedings of 6th International Fall Workshop Vision, Modeling, and Visualization*, pp. 121–128, 522, Nov. 21–23 2001.
23. W. Blanke, C. Bajaj, D. Fussell, and X. Zhang, "The Metabuffer: A Scalable Multiresolution Multidisplay 3D Graphics System Using Commodity Rendering Engines," Technical Report TR2000-16, University of Texas at Austin, 2000.
24. G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan, "Lightning-2: A High-Performance Display Subsystem for PC Clusters," in *Siggraph 2001*, pp. 141–148, Aug. 12–17 2001.
25. S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heirich, "Scalable Interactive Volume Rendering Using Off-the-Shelf Components," in *IEEE PVG 2001*, pp. 115–121, 158, Oct. 2001.
26. J. Montrym, D. Baum, D. Dignam, and C. Migdal, "InfiniteReality: A Real-Time Graphics System," in *Siggraph 1997*, pp. 293–302, Aug. 3–8 1997.
27. J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover, "PixelFlow: The Realization," in *Graphics Hardware Symposium*, pp. 57–68, 1997.
28. I. SGI. InfinitePerfomance, http://www.sgi.com.
29. HP. Visualization Center SV6, http://www.hp.com.
30. M. Simmons and C. Séquin, "Tapestry: A Dynamic Mesh-based Display Representation for Interactive Rendering," in *Rendering Techniques 2000*, pp. 329–340, June 2000.
31. H. Samet, *Applications of Spatial Data Structures*, Addison-Wesley, 1990.
32. T. Porter and T. Duff, "Compositing Digital Images," in *Siggraph 1984*, pp. 253–259, July 1984.
33. MPI. The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.
34. LLNL. Lawrence Livermore National Laboratory, http://www.llnl.gov/linux/chaos.
35. Quadrics. Quadrics, Inc. http://www.quadrics.com.
36. V. Pascucci and R. Frank, "Global Static Indexing for Real-time Exploration of Very Large Regular Grids," in *Proceedings of Super Computing CD*, Nov. 10–16 2001.
37. C. Nuber, E. LaMar, K. Joy, and B. Hamann, "Error-Based Temporal Cache and Reuse," in *Geometrical Methods for Scientific Visualization, to appear*, ?, ed., pp. ?–?, Springer-Verlag, 2003.
38. C. Nuber, E. LaMar, V. Pascucci, B. Hamann, and K. Joy, "Using Graphs for Fast Error Term Approximation of Time-varying Datasets," in *Data Visualization 2003*, pp. ?–?, EUROGRAPHICS/IEEE, Springer-Verlag, ?–? May 2003.