Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays

Kenneth Moreland
kmorel@sandia.gov

Brian Wylie
bnwylie@sandia.gov

Constantine Pavlakos
cjpavla@sandia.gov

Sandia National Laboratories

Abstract

Due to the impressive price-performance of today's PC-based graphics accelerator cards, Sandia National Laboratories is attempting to use PC clusters to render extremely large data sets in interactive applications. This paper describes a sort-last parallel rendering system running on a PC cluster that is capable of rendering enormous amounts of geometry onto high-resolution tile displays by taking advantage of the spatial coherency that is inherent in our data. Furthermore, it is capable of scaling to larger sized input data or higher resolution displays by increasing the size of the cluster. Our prototype is now capable of rendering 120 million triangles per second on a 12 mega-pixel display.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems-Distributed/network graphics.

Keywords: Parallel Rendering, Sort-Last, Compositing, PC-Cluster, Tile Display.

1. Introduction

The Department of Energy's Accelerated Strategic Computing Initiative (ASCI) is producing computations of a scale and complexity that are unprecedented [5, 19]. High fidelity simulations, at high spatial and temporal resolution, are needed to achieve the necessary confidence in simulation results. The ability to visualize the enormous data sets produced by such simulations is well beyond the current capabilities of a single-pipe graphics machine. Our needs require interactive rendering of several hundred million polygons, which can only be achieved by applying parallel techniques. We are, however, willing to sacrifice the ability to render at real-time rendering rates for the ability to work with hundreds of millions of polygons at a time.

The focus of our work has been to develop highly scalable rendering techniques. Highly scalable techniques will be necessary to address projected rendering performance targets, which are as high as 19 billion polygons per second on displays of 64 million pixels in 2004 [19]. As a part of a broader effort in ASCI's Visual Interactive Environment for Weapons Simulations (VIEWS) program, Sandia National Laboratories (SNL) is exploring the development of cluster-based rendering systems to address these extreme data sets. The intent is to leverage widely available commodity graphics cards and workstations in lieu of traditional, expensive, specialized graphics systems.

Rendering on cluster computers requires specialized parallel rendering algorithms. Parallel rendering algorithms, regardless of hardware architecture, fall into three categories first proposed by Molnar [14], sort-first, sort-middle, and sort-last, based on how the geometric primitives are sorted from object space to screen space. Both sort-first and sort-middle strategies require primitives to be rasterized by the rendering processors responsible for the screen space in which each primitive lies. To achieve this as viewpoints change arbitrarily, either some of the primitives must be transferred in between frames, or the data must be replicated on all processors. Neither of these strategies can scale to SNL's extreme data sets.

In contrast, the sort-last strategy combines images after rasterization occurs. Sort-last allows each geometric primitive to be rendered on any processor. Hence, each processor only needs to hold a fraction of the primitives, and the primitives never need to be transferred between processors. In fact, parallel rendering using sort-last yields a faster polygon per second rendering rate as data sets get larger. However, unlike the sort-first and sort-middle strategies, the performance of sort-last parallel rendering drops sharply as the resolution of the display increases. Yet we feel we can address this issues. Our goal is to drive multiple tile displays with frame rates that are comparable to a single tile display using a sort-last-based parallel algorithm that scales appropriately with large data sets.

2. Related Work

A substantial amount of work pre-exists in this area, especially with regard to software implementations on parallel computers [2, 8, 12, 15, 21, 22, 23]. As with our work, these efforts have been largely motivated by visualization of big data, with an emphasis on demonstrating scalability across significant numbers of compute processors. However, these software-based efforts have yielded relatively modest raw performance results when compared with hardware rendering rates. Recent efforts have provided parallel rendering using commodity hardware at close to aggregate hardware rendering rates [24], but at moderate display resolutions. Others have designed highly specialized parallel graphics hardware, such as the PixelFlow system [4], that scales and is capable of delivering extensive raw performance, but such systems have not yet proven to be commercially viable. Still others are developing parallel hardware to be used in conjunction with commodity graphics hardware such as Sepia [13], the Metabuffer [1], and Lighting-2 [20].

The desire to drive large, high-resolution tiled displays has recently become an additional motivation for building parallel rendering systems. ASCI partners, including Princeton University [16] and Stanford University [6, 7], as well as the ASCI labs themselves [17], are pursuing the implementation of such systems. Both Stanford and Princeton have implemented scaleable display systems using PC-based graphics clusters.

3. Overview of Approach

Traditional approaches to the sort-last rendering problem require each processor to render an image of the same size as the final display. However, our current target output resolution is 12 million pixels and growing. Producing an image of this size is well beyond the capabilities of commodity graphics hardware. Buffer space for images of this size, almost 100MB for each image, can be taxing even for memory available to a PC's CPU. Furthermore, given that we are using a distributed memory parallel-computing environment, the image will likely need to be split and distributed among several display processors before it can be viewed.

In fact, because commodity graphics hardware is incapable of generating high-resolution images, many high-resolution displays do not take input from a single graphics engine. Instead, they take the input of several graphics engines and display them in a tiled grid [9, 17]. A tiled display such as this is our target output, so it makes sense to follow a similar convention in our parallel rendering. Rather than render a single high-resolution image, each processor generates images for the tiles that make up the display. This may require each processor to render several images for each frame: one image for each tile on which it displays data. Just as the image for each tile is rendered individually, each tile image is composed individually and drawn onto the appropriate portion of the display. In short, our approach is to use N processors to render and compose a large geometry with T different projections, one for each display tile.

4. System Organization

SNL has built the VIEWS Data Visualization Corridor, shown in figure 15, to service our demanding visualization needs. The Corridor includes a 4x4, 16-tile display, which is expandable up to 48 tiles (12x4). The tiled display uses an array of rear projectors with each array element connected to a single node in our graphics cluster, known as "RiCky." RiCky is comprised of 64 Compaq 750 nodes. Each of these nodes is equipped with 512MB of main memory and 4x AGP versions of nVidia's GeForce^{TM} 256 graphics chipset. Switched fast Ethernet interfaces on each node in RiCky provide an administrative and general purpose TCP/IP channel to the cluster. The message passing and parallel communications interconnect between the nodes is composed of a high-speed system area network. As of spring 2001, this network is based on Compaq/Tandem's Servernet II VIA-based hardware. We found this system area network to yield a peak throughput of 95 MB/s from point to point using VIA protocols, but the peak bandwidth dropped to about 70 MB/s when using our implementation of MPI. An upgrade to Myrinet 2000 is scheduled for this summer. Total cost for the 64-node graphics cluster was approximately $500K. In this paper, a reference to a "processor" is considered a unit like the nodes in our cluster containing a general purpose CPU and dedicated rendering and network hardware.

The system is designed to allow any number of N processors to contribute to rendering T images for a tile display as long as $N \geq T$. Thus, some processors will be responsible for displaying fully composed images on a viewable display tile while other processors will be solely responsible for rendering and composing images. However, the system does not make a clear distinction between these display processors and those that do not output an image to the user. Because it takes so little time to display an image on a tile, we can achieve much better load balancing by letting the display processors render and compose images along

with the rest of the processors. Only after the rendering and composing of images is complete are the display processors differentiated by drawing the final image on the screen.
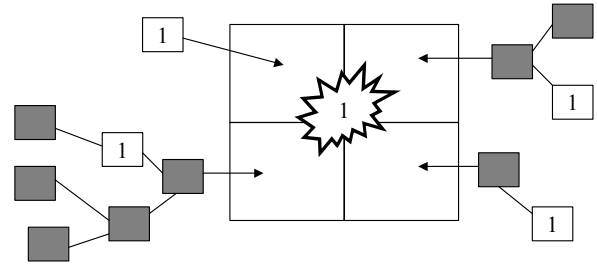

Figure 1: Processor 1's geometry spans four tiles and contributes to the compositing networks of each.

The system software is intended to draw polygons that are evenly distributed amongst all the processors. Like other sort-last systems, the system does not need to redistribute any of these polygons to draw different viewpoints. To draw an image on the multiple tile screen, each processor first determines which tiles its polygons will be drawn on using bounding box information and the current transformation matrices. The geometry on processor 1 shown in figure 1 is projected onto four tiles. Thus, the processor must render four images, one for each tile. Rather than render and cache all the images up front, which can take a substantial amount of memory, the images are rendered through callbacks on an as-needed basis.

The projection information is scattered amongst the processors, and a compositing network for each of the tiles is collectively built. In figure 1, because processor 1 generates image data for each of the four tiles shown, it must participate in the compositing network for each tile. The load balancing of the system depends a great deal on the strategy for building these networks and performing the composition in parallel. For experimental purposes, the composition function was implemented as a swappable component. In the following sections, we describe several methods of composing multiple images in parallel and some optimizations that can be made on the process.

5. Composition Strategies

Sort-last rendering strategies such as those described by Ma [10] and Mitra [12] are designed to generate a single, full-resolution image. However, our approach is to generate several lower-resolution images for use in a tile display. Thus, we require other methods to generate these multiple images.

Our data sets come from simulation codes run on massively parallel machines. For basic efficiency, these codes must work to keep cell elements spatially coherent within each processor and amongst adjoining processors [3]. When geometry such as an isosurface is extracted from the simulation's final model, the geometry contained by each processor also tends to be spatially coherent. Even geometries that do not initially have good spatial locality can be preprocessed through a straightforward algorithm that groups geometric primitives into three-dimensional regions.

When spatially coherent, the geometry of each processor is typically projected onto a fraction of the total viewing area. Thus as a rule, each processor will not have to render every tile because many of the tiles will not have any geometry rendered into them. This reduction of images significantly reduces the total amount of work that needs to be done to render tiles. We wish our composition strategies to take advantage of this.

5.1. Serial

A simple approach to compose T images for a tile display is to serially run a composition algorithm for a single display T times. Because the composition is independently run T times, we can expect it to take T times as long to complete. However, the *serial* strategy does not take advantage of any spatial decomposition of the geometry. Processors perform no valuable work when composing images it does not render. It was built mostly for comparison for other composition strategies. Any other strategy should perform much better than the *serial* strategy in the average case and perform as well as the *serial* strategy in the worst case.

5.2. Virtual Trees

Our *virtual trees* strategy is derived from the binary tree algorithm for composing a single image [12]. In each step of the binary tree algorithm, half of the participating processors relinquish their entire image and drop out of the computation while the other half receive one image, perform a depth comparison to their own image, and reiterate. We observed that if we were running several binary tree compositions in parallel, these freed processors could join other binary tree computations.
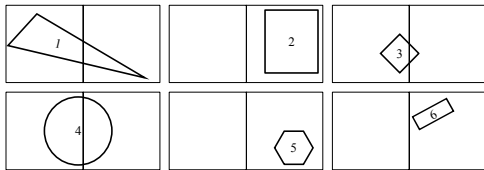
(a) Six processing nodes (numbered one through six) independently generate the above images on two tiles. Some processors generate two images, one for each tile.

Stage 1
Stage 2
Stage 3
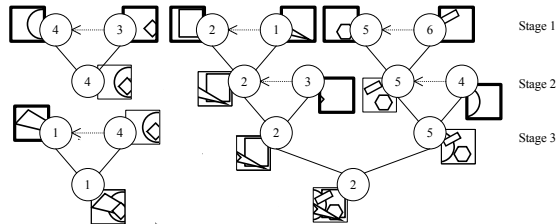
(b) Two virtual trees used to compose two complete tile images. The nodes are the numbered circles with their current image for the given composed tile next to them. At the leaves, each node has an image they have just rendered, shown in bold. As stated before, some nodes must render more than one image.

Figure 2: Example of *virtual trees* parallel composition.

The strategy works by creating a "virtual" tree for each display tile. Contained in each tree are processors that have rendered an image for that display tile. The algorithm proceeds much like the binary tree composition algorithm except that the processors float among the trees, helping with the composition, as they become available. Figure 2 shows an example of *virtual trees* that may be used to compose the six objects shown. The composition for each image proceeds as a binary tree composition except that processors will float their participation between the trees. In particular, notice that processor 4 takes part in the left tree in stage 1, then floats to take part in the right tree in stage 2, and returns to take part in the left tree in stage 3. When necessary, the processor must keep track of multiple images belonging to several virtual trees.

Properly scheduling when and where each processor will participate in each tree is vital to ensure that the depth of each tree is low and the load utilization is high. The scheduling algorithm we used is based on the observation that processors that still have many images to contribute should attempt to send an image to free its resources for other trees. Likewise, processors with few images to contribute should attempt to receive images to continue to be included in the compositions. Thus, the scheduling algorithm first sorts the processors keyed on how many images each contain. It then pairs senders and receivers, starting with the processes with the most images as senders and those with the fewest images as receivers. The images are then transferred and depth compared. The whole process is repeated until there remains only one image for each display tile. (Note that this scheduling is not demonstrated in figure 2 to better demonstrate the properties of *virtual trees*.)

We had moderate success with the *virtual trees* strategy. The algorithm composed images for a 16-tile display much faster than the *serial* strategy. However, the weakness of the *virtual trees* strategy is that eventually processors must drop out of the computation regardless of how good the scheduling. When composing the 16 tile images on our rendering cluster, the last stage could only have 16 processors participating in the composition while the other 48 processors remained idle. While the load balancing of the strategy was basically good until the last two or three stages of the algorithm, these were the stages where our compression techniques (described in section 6.2) were least effective and poor load balancing hit us hardest.

5.3. Tile Split and Delegate

Next, we tried to develop a simpler strategy that involved all the processors throughout the computation. Our strategy is an extension of the direct send algorithm described by Ma [10]. We assign each processor to one section of a single tile. Each processor then collects image data pertinent to its section from other processors and depth compares them. The tile sections are then combined to form a complete image for each tile. The processes are load balanced by assigning tile sections in such a way that each processor receives the same amount of pixel data to compose. This balance can be created by splitting tiles which have more images rendered in them into smaller pieces.

Figure 3 shows an example of using *tile split and delegate* to compose the same six objects shown in figure 2a. The *tile split* strategy first assigns each processor to be involved in the composition of a single display tile. The number of processors assigned to each display is proportional to the number of images that need to be composed for that tile. That is, if more processors generated images for a given tile than the other tiles, more processors will take part in composing the given tile. In this example, there are three images generated for the left tile and six images generated for the right tile. Because there are twice as many images in the right tile, twice as many processors are assigned to that tile. Hence, the right tile gets four processors assigned to it while the left tile only gets two.

After a set of processors is assigned to a given tile, the tile is split evenly amongst the processors in this set. Each processor is solely responsible for the portion of the screen assigned to it. Then, each processor renders each applicable image, splits it, and sends each piece directly to the processor responsible for composing that piece. Figure 3c shows how the two images generated by processor 1 get split up and sent to the responsible processors.
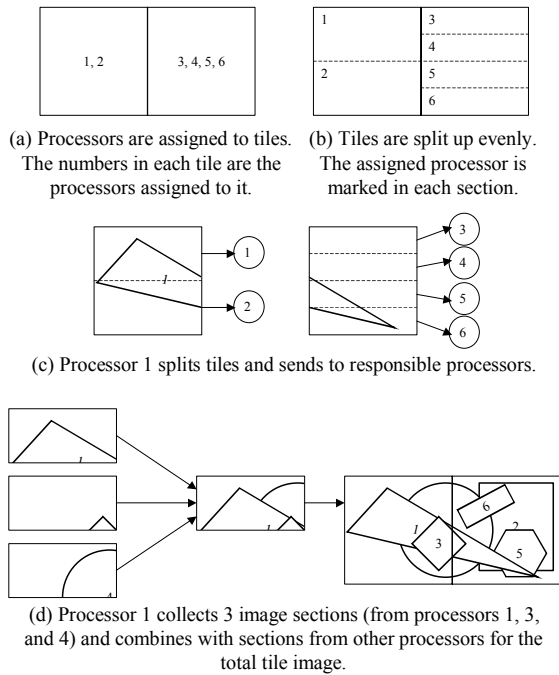
(a) Processors are assigned to tiles. The numbers in each tile are the processors assigned to it.

(b) Tiles are split up evenly. The assigned processor is marked in each section.

(c) Processor 1 splits tiles and sends to responsible processors.

(d) Processor 1 collects 3 image sections (from processors 1, 3, and 4) and combines with sections from other processors for the total tile image.

Figure 3: Example of *tile split and delegate* parallel composition.

The PCs in our cluster, like all standard PCs, have separate hardware for graphics and networking. We take advantage of this by asynchronously sending the first image while rendering the second image. Exercising both the graphics and networking hardware at the same time rather than letting one go idle makes better use of the system resources and reduces the overall processing time.

While rendering and sending images, each processor is also asynchronously receiving image sections from processors it knows will be rendering to the tile. As each processor receives incoming image fragments, it performs depth buffer comparisons on them. Once the processor has finished comparing all incoming image fragments, it sends its fully composed screen portion to the display processor. The display processor then pieces together the fragments before drawing the final image for the user. Figure 3d shows which image sections processor 1 receives, what they look like after they are depth-compared, and how the resulting segment fits within the tile.

The *tile split and delegate* strategy worked very well in the average case. It consistently performs better than the *virtual trees* strategy. However, the *tile split* strategy can require a large amount of message passing. In the degenerate case, where every processor renders images for every tile, the *tile split* strategy requires all-to-all communication within the processors, generating $O(N^2)$ messages. Furthermore, these messages are not generated in any predefined sequence. In some cases, the *tile split* strategy generated so many messages it caused flow control problems with our Servernet II interconnect. Therefore, we began looking for other strategies that could be load balanced as well as *tile split*, but did not generate as many messages.

5.4. Reduce to Single Tile

Our next attempt at parallel composition was to reduce the problem to that of composing a single image in the same manor as traditional sort-last parallel rendering systems. We observed that

if each processor contains at most one image, then each processor could take part in the composition of that image without affecting the composition of any other images that may be happening in parallel. Of course, we need to do some image transfers first to ensure that each processor only has one tile image.

Before composition begins, each processor holds between zero and T images for separate tiles. The goal is for each processor to hold exactly one image. The *reduce to single tile* strategy first designates each processor to hold an image for a particular tile. Much like the *tile split and delegate* strategy, more processors are assigned to tiles with more images rendered in them. The processors are then scheduled to receive images for the tile they have been assigned, and send all the images for tiles they have not been assigned. The transfers are scheduled in such a way that all processors assigned to a given tile receive about the same amount of images. As processors accept incoming images, they are depth compared to generate a single image.

Once all of these transfers are complete, each processor should have exactly one image. Now each processor performs the binary swap algorithm [10, 12] with the other processors that hold images for the same tile. Mitra has shown that binary swap's running time asymptotically approaches a constant as the number of processors is increased [12], so each composition completes at about the same time.
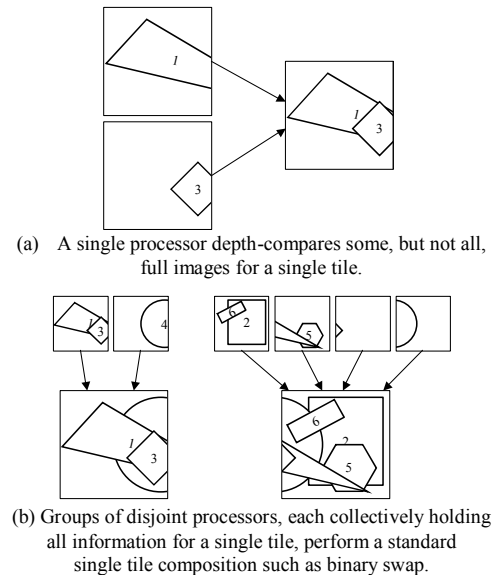

(a)   A single processor depth-compares some, but not all, full images for a single tile.

(b) Groups of disjoint processors, each collectively holding all information for a single tile, perform a standard single tile composition such as binary swap.

Figure 4: Example of *reduce to single tile* parallel composition.

Consider a *reduce to single tile* parallel composition working on six processors that are rendering the same six objects shown in figure 2a. The processors are assigned to tiles in the same manner as in the *tile split and delegate* strategy as shown in figure 3a. Each processor then receives a subset of the images for that tile. Figure 4a shows how one of the processors for the left tile may collect two of the images for that tile. Figure 4b shows the partially composed images contained by the two disjoint groups of processors. Each group collectively holds all the data for a tile image. The groups perform a traditional single image composition to produce the two tile images also shown in Figure 4b.

The *reduce* strategy gives us load balancing that is nearly as good as the *tile split and delegate* strategy. Furthermore, it produces many fewer messages: $O(N*T + N \log N)$ where N is the number of processors and T is the number of tiles. The *reduce*

strategy was not nearly as taxing for the interconnect switches as the *tile split* strategy. Our Servernet II interconnect has no problem handling the message passing of the *reduce* strategy. Moreover, because the growth rate of messages does not have an N^2 term in it, we expect it to scale to large clusters better.

6. Optimizations

In addition to developing efficient strategies for parallel composition of multiple images, we were constantly looking for other techniques that would help maximize the performance of our system. This section describes general optimizations that are applicable to the system when it is running any of the composition strategies described above. The optimizations focus mainly on reducing the overhead of generating multiple tiles and reducing network traffic.

6.1. Bucketing

Consider a set of polygons that lies within four tiles as shown in figure 5. Only a small subset of polygons actually lies within the upper right quadrant. Without first testing the polygons to determine in which tiles they lie, it is
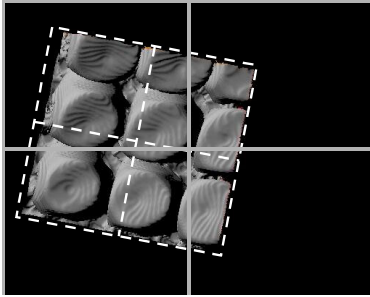
Figure 5: Using buckets on object straddling tile boundaries.

necessary to feed every polygon to the graphics hardware four times (once for each tile image to be drawn) and let the hardware clip unnecessary polygons. While this is certainly a correct solution, it incurs a heavy penalty in the geometry processing in the graphics pipeline.

To reduce the total amount of polygons sent to the graphics hardware, we estimated which polygons could be ignored with bucketing. Before rendering begins, each processor's polygons are grouped into several 3D regions called buckets. This bucketing only occurs at initialization when the data is read from disk. Before each tile image is rendered, the buckets are tested to determine which lie in the tile. Only the polygons in these buckets are rendered. Now if all the polygons in figure 5 are split into the four buckets defined by the dotted lines, only one fourth of the polygons will need to be rendered in the upper right tile. This number can be decreased further by making larger numbers of smaller buckets. However, the more buckets used, the greater the overhead in determining screen projections.

We found that using even a moderate amount of buckets could cut our rendering time in half if the geometry straddled several tiles. With our very large data set, we found that the spatial coherency of the geometry was fine grained enough to base our buckets around the layout of the data rather than to sort the data into new buckets.

6.2. Active Pixel Encoding

Because each processor renders only a fraction of the total geometry, the geometry often occupies only a fraction of the screen space in some or all of the tiles in which it lies. Consequently, the initial images distributed between processors at the beginning of composition often have a significant amount of blank space within them. Explicitly sending this information

between processors is a waste of bandwidth. Transferring sparse image data rather than full image data is a well-known way to reduce network overhead [14]. So far, our best method to do this has been with active pixel encoding.

The sparse image data is sent via alternating run lengths of "active" pixels, pixels that contain geometry information, and "inactive" pixels, pixels that have no polygons drawn on them. The active pixel run length is followed by pairs of color and depth values. The inactive pixels are not accompanied by any color or depth information. The depth information is assumed to be of maximum depth, and the color values are ignored since they contain no geometry information.

There are many other ways to encode sparse images and reduce data redundancy. However, we are particularly enamored with our active pixel encoding for this application because it exhibits all of the following properties:

- Fast encoding. Image encoding requires each pixel to be visited exactly once. Each visit includes a single depth buffer comparison, a single addition, and at most one copy.
- Free decoding. Processors typically perform a depth comparison as soon as they receive incoming data. The depth comparison can be done directly against an image that is still encoded in sparse form. In fact, the depth comparison can skip the comparisons for the inactive pixels. Thus doing depth comparisons against encoded images is often faster than against unencoded images.
- Effective compression. During the early stages of composition when the largest images must be transferred, the sparse data is commonly less than one fifth the size of the original data.
- Good worst case behavior. No image will ever grow by more than a few bytes of header information. Images that have geometry drawn on every pixel will only have one run length. Even images that alternate between active and inactive status for every pixel, and hence have a run length for every pixel, do not grow when encoded. The number of bytes required to record two run lengths is equal to the number of bytes saved by not recording color and depth information for a single inactive pixel. Thus, there is no penalty for recording run lengths of size one.

6.3. Floating Viewport

Consider the geometry shown in figure 6 that projects onto a screen space that fits within a single tile but is moved in the horizontal and vertical directions so that it straddles four tiles. If the system limits itself to projecting onto physical tiles, the processor must render and read back four

Figure 6: Using floating viewport on object straddling tile boundaries.

images; although it could generate a single image that contains the entire geometry with the exact same pixel spacing. Instead of rendering four tiles, the system can float the viewport in the global display to the space straddling the tiles. That is, the system may
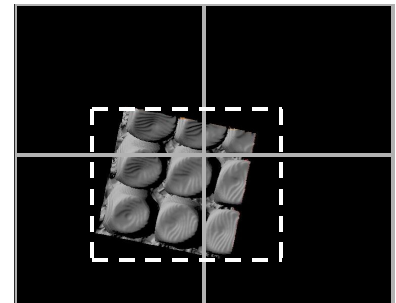
project the geometry to the space shown by the dotted line in figure 6 and split the resulting image back into pieces that can be displayed directly on each tile. Hence, the system does not need to render any polygon more than once, and the frame buffer is read back one time instead of four.

When a processor's geometry fits within the floating viewport, it can cut the rendering time dramatically. This is most likely to happen when the number of tiles is small compared to the number of processors and the spatial coherency of the data is good.

7. Experimental Results

The system and algorithms described above have been implemented in C. The system was then tested on the rendering cluster described in section 4 running Windows 2000. The goal of these tests was to demonstrate the characteristics of the system, compare the algorithms described in section 5, and determine the feasibility of using a sort-last parallel rendering system on a PC cluster to perform interactive rendering onto multiple tile displays.

Four data sets were used to test the system. Three are sets of triangles with various random distributions. In the first, shown in figure 11, all triangles are distributed linearly throughout the viewable area. This serves as worst-case scenario: each processor has triangles projected onto every tile. The second data set, shown in figure 12, has triangles placed in a Gaussian distribution, where every processor has its triangles centered at a different, randomly chosen point. The Gaussian distribution allows the triangles within each processor to be spatially coherent while still allowing irregular overlapping. The third data set, shown in figure 13, has the triangles for each processor contained in non-overlapping boxes. This creates a perfect separation of triangles between processors and maximizes the spatial coherency within each processor. The images in figures 11, 12, and 13 are drawn with each processor rendering its triangles in a single color to show how the polygons are distributed amongst the processors. The forth data set, shown in figure 14, is a 469 million triangle isosurface generated from a large turbulence simulation provided by Lawrence Livermore National Laboratories [11]. No pre-processing is done on this data set. The distribution of polygons amongst processors and the bucketing of polygons within processors (see section 6.1) are based entirely on the organization of the original data set. That is, all spatial decomposition is inherited from the original simulation.

Table 1 shows run time statistics when using the parallel composition strategies described in section 5 on these four data sets. The times required to render polygons, read and write images to and from the graphics cards, compress images, and compare images in each frame are given. The overall amount of data transferred between frames and the frame rate are also given. The "serial," "v tree," "split," and "reduce" strategies are those described in sections 5.1, 5.2, 5.3, and 5.4 respectively. The measurements were taken by a simple application that displays the given data and rotates it about the Y-axis. The application was run on all 64 nodes of our rendering cluster and displayed on 16 tiles laid out as a 4x4 grid. The total resolution of this display is 12 mega-pixels. The *tile split and delegate* strategy failed to compose the linear distribution of triangles. We believe this is caused by flow control problems in our Servernet II interconnect generated by the all-to-all communication of the *tile split* strategy.
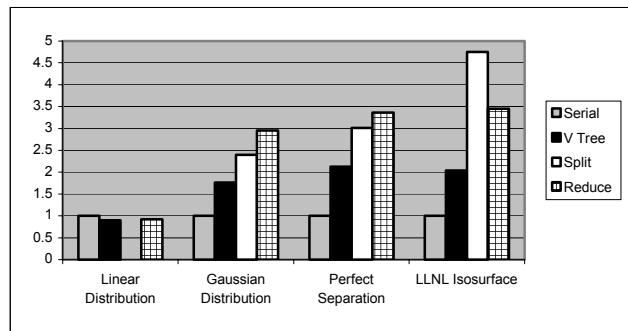
7.1. Comparison of Strategies

Figure 7: Relative frame rates for composition strategies.

Figure 7 shows the relative frame rates that each composition strategy required for each data set. The relative frame rate is the frame rate measured for the given strategy divided by the frame rate of performing a sort-last composition on each tile individually (i.e., using the *serial* strategy). As expected, when the data is not spatially decomposed, as is the case with the "linear distribution" data set, the *serial* strategy is optimal. However, for data sets that are spatially decomposed, any of the other strategies are a much better alternative.

	Strategy	Render Time (ms)	Read Time (ms)	Write Time (ms)	Compress Time (ms)	Z Compare Time (ms)	Net Usage (Mbytes)	Frame Rate (Hz)
Linear Distribution	Serial	94.9	638.5	10.0	436.5	269.3	2638	0.269
	V Tree	93.8	603.6	9.9	435.4	430.5	2598	0.241
	Split	—	—	—	—	—	—	—
	Reduce	94.8	554.1	9.9	359.5	363.3	2046	0.247
Gaussian Distribution	Serial	79.3	514.1	9.9	307.3	62.0	491	0.370
	V Tree	78.1	357.8	10.1	184.8	127.0	386	0.652
	Split	78.5	505.0	10.2	238.6	43.0	238	0.886
	Reduce	78.3	313.1	9.9	140.1	109.5	296	1.093
Perfect Separation	Serial	76.3	487.4	9.9	317.3	78.9	562	0.309
	V Tree	76.3	380.8	10.0	201.3	131.0	464	0.657
	Split	76.6	483.4	10.3	235.2	56.1	279	0.930
	Reduce	76.8	357.1	9.9	163.4	95.3	383	1.040
LLNL Isosurface	Serial	2229.2	430.8	9.9	292.6	41.6	340	0.076
	V Tree	2416.3	332.0	10	188.1	96.8	236	0.155
	Split	2424.1	423.7	10.3	205.4	30.0	156	0.361
	Reduce	2416.7	254.9	9.8	119.3	78.9	213	0.262

Table 1: Statistics for rendering each data set under each composition strategy.

The results show that both the *tile split* and the *reduce* strategies performed surprisingly well composing the Lawrence Livermore National Laboratories data set. The *tile split* strategy ran 4.75 times as fast as the *serial* strategy, and the *reduce* strategy ran 3.45 times as fast. In fact, we were pleasantly surprised to find that the performance boost for real data with spatial sorting inherited from a parallel simulation was better than that for our fabricated data with "perfect" separation. The data suggests that these methods can effectively take advantage of the same spatial locality requirements for efficient parallel simulations.

7.2. Tradeoffs
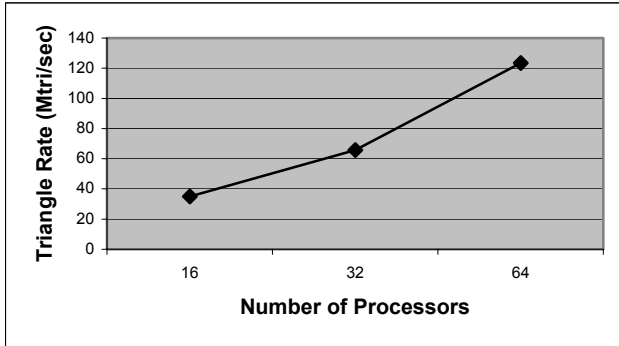
Figure 8: Measured triangle rates while rendering the LLNL isosurface data set on 16 tiles. Triangle rates are measured in millions of triangles per second.

Figure 8 shows a plot of the rendering speeds of the system while rendering the Lawrence Livermore National Laboratories data set on a 16-tile display using the *reduce* strategy. The number of nodes used in rendering and composing is varied to determine its effect on the system performance. Notice that each time the number of render processors is doubled the triangle rate is also nearly doubled.
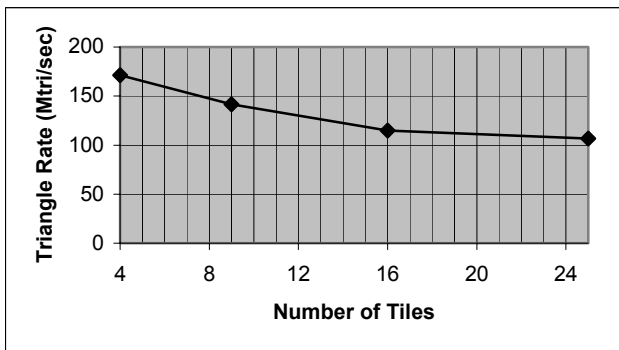

Figure 9: Measured triangle rates while rendering the LLNL isosurface data set using 64 processing nodes.

Figure 9 shows a plot of the rendering speeds of the system while rendering the same data set. However, instead of varying the number of processing nodes, it varies the number of tiles in the display. As can be expected, the rendering performance is negatively affected by increased display sizes, but not by as much as might be expected by a traditional sort-last composition. Quadrupling the display by increasing the number of tiles from 4 to 16 only drops the rendering rate by 33%.

These two graphs suggest a tradeoff that the system provides. The aggregate computational power of the processors

can be directed toward the display resolution or the rendering rate. Increasing the resolution of the display will drop the rendering speed, but the speed can be increased again by adding more processors. In effect, we are trading display resolution for rendering speed.

7.3. Comparison to Other Parallel Cluster Rendering Systems

Typical software implementations of parallel rendering systems using a sort-last approach can render large data sets as fast or even faster than the system described here, but they cannot handle the large output display. Consider the sort-last parallel renderer described by Wylie [24]. Using the same hardware and LLNL isosurface data described in this paper, Wylie was able to generate a 1024 x 768 image in about 1.5 seconds. In order to render the images used in the benchmarks in table 1, it would need to be run 16 times, taking a total of 24 seconds. That is over six times as long as it took our system using the *reduce* strategy.
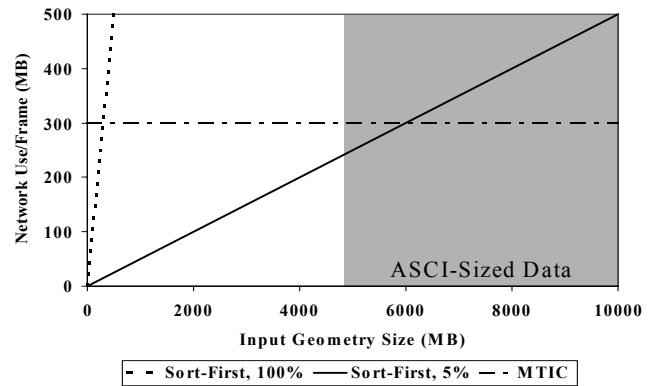

Figure 10: The network bandwidth required for rendering large input geometries.

Typical software implementations of parallel rendering systems using a sort-first approach can potentially render frames at much higher rates, but they cannot handle large input data sets. Figure 10 compares the amount of network bandwidth required for a sort-first system that needs to transfer every polygon (such as WireGL [7]), a sort-first system that needs to transfer about 5% of the polygons, and the multiple tile image compositor described in this paper. The sort-first system that transfers 100% of the polygons requires much more bandwidth than our system for ASCI-sized data sets. A sort-first system that transfers only 5% of the polygons will be using about the same amount of network bandwidth as our system with our large data sets. However, a sort-first system of this type probably achieves this reduced network bandwidth by taking advantage of frame-to-frame coherency. On our hardware, it takes over 1 second just to render our data in parallel on all 64 nodes with perfect load balancing. In an interactive application with a 1 second refresh, the user is likely to significantly change the viewing angle between refreshes. Thus, we may expect little frame-to-frame coherency, and the sort-first system will begin to consume much more network bandwidth.

8. Conclusions

In this paper, we have demonstrated the use of a sort-last parallel rendering system that is capable of rendering to large tile displays. Because the system does not replicate input data amongst processors or transfer input data between processors, it

can handle continually growing data set sizes. Furthermore, we can support ever-growing display resolutions with larger rendering clusters.

We have demonstrated several composition algorithms that are designed to create the multiple images required for a tile display. Our running tests show that some of these algorithms can generate these images in one quarter of the time required for previous sort-last composition algorithms. We also described some effective optimization techniques that can be used to help render and compose these images in an interactive application.

As can be expected, we noticed a processing overhead for rendering to larger displays. However, we feel that the system's extra computational needs as the display resolution grows are quite reasonable. Furthermore, this extra computational load can be recovered by adding more rendering nodes to the cluster. We feel we have demonstrated that the system is a viable driver for visualizing the extremely large data sets generated by Sandia National Laboratories and the other DOE laboratories.

9. Future Work

We expect to continue to explore the use of sort-last rendering techniques for use on tile display systems. We also anticipate continued optimization of our current software and possible consideration of hybrid sorting schemes.

We are also looking to adapt our current software to existing APIs to make porting applications easier. Our most likely candidate is the Visualization Toolkit [18]. Implementing a new renderer in the Visualization Toolkit is straightforward, and its organization complements the callback structure of our system nicely.

We also note with respect to table 1 that one of the bottlenecks of the system, like many other software-based sort-last systems, is the time required to read back the frame buffers. The Metabuffer [1] and Lightning-2 [20] technologies attempt to circumvent this cost by reading the frame buffers from DVI outputs. Unfortunately, current 3D graphics cards do not have a direct path from their depth buffer to their DVI output. Once this issue has been resolved and these technologies improve, we hope the techniques discussed in this paper can be combined with DVI reading hardware for a significant speedup.

10. Acknowledgements

11. References

[1] Blanke, W. et al. The Metabuffer: A Scalable Multiresolution Multidisplay 3-D Graphics System Using Commodity Rendering Engines. Technical Report TR2000-16, Department of Computer Science, University of Texas at Austin, 2000.

[2] Crockett, T. W. and Orloff, T. A MIMD Rendering Algorithm for Distributed Memory Architectures. *1993 Parallel Rendering Symposium Proceedings*, pages 35-42. IEEE. October 1993.

[3] Devine, K. D. et al. Design of Dynamic Load-Balancing Tools for Parallel Applications. *Proceedings of the International Conference on Supercomputing*, Sante Fe, May 2000.

[4] Eyles, J. et al. PixelFlow: The Realization, *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57-68. Los Angeles, CA, August 1997.

[5] Heermann, P. Production Visualization for the ASCII One TeraFLOPS Machine. *Proceedings of Visualization '98*, pages 459-462. IEEE, October 1998.

[6] Humphreys, G. and Hanrahan, P. A Distributed Graphics System for Large Tiled Displays. Proceedings of *Visualization '99*, pages 215-223. IEEE, October 1999.

[7] Humphreys, G. et al. Distributed Rendering for Scalable Displays. *Proceedings of SC2000*. IEEE, 2000.

[8] Lee, T. et al. Image Composition Methods for Sort-Last Polygon Rendering on 2-D Mesh Architectures. *1995 Parallel Rendering Symposium Proceedings*, pages 55-62. IEEE. October 1995.

[9] Li, K. et al. Building and Using A Scalable Display Wall System. *IEEE Computer Graphics and Applications*, pages 29-37. IEEE Computer Society, July/August 2000.

[10] Ma, K. et al. Parallel Volume Rendering Using Binary-Swap Image Composition. *IEEE Computer Graphics and Applications*, pages 59-68. IEEE Computer Society, July 1994.

[11] Mirin, A. Performance of Large-Scale Scientific Applications on the IBM ASCI Blue-Pacific System. *Proceedings of the Ninth SIAM Conference of Parallel Processing for Scientific Computing*, CD-ROM. SIAM Philadephia. March 1999.

[12] Mitra, T. and Chiueh, T. Implementation and Evaluation of the Parallel Mesa Library. *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. December 1998.

[13] Moll, L., Heirich, A., and Shand, M. Sepia: Scalable 3D Compositing Using PCI Pamette. *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '99)*, April 1999.

[14] Molnar, S. et al. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, pages 23-32. IEEE Computer Society, July 1994.

[15] Samanta, R. et al. Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. Interlaken, Switzerland, August 2000.

[16] Samanta, R. et al. Load Balancing for Multi-Projector Rendering Systems. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. August, 1999.

[17] Schikore, D. et al. High-resolution multi-projector display walls and applications. *IEEE Computer Graphics and Applications*, pages 38-44. IEEE Computer Society, July/August 2000.

[18] Schroeder, W. et al. *The Visualization Toolkit 2nd Edition*. Prentice Hall, Upper Saddle River, NJ. 1998.

[19] Smith, P. H. and van Rosendale, J. Data and Visualization Corridors, *Report on the 1998 DVC Workshop Series*. Caltech, 1998.

[20] Stoll, G. et al. Lightning-2: A High-Performance Display Subsystem for PC Clusters. *Computer Graphics (Proceedings of SIGGRAPH 01)*, August 2001.

[21] Udeshi, T. and Hansen, C. Parallel Multipipe Rendering for Very Large Isosurface Visualization. *Joint EUROGRAPHICS – IEEE TCCG Symposium on Visualization*. 1999.

[22] Whitman, S. A Load Balanced SIMD Polygon Renderer. *1995 Parallel Rendering Symposium Proceedings*, pages 63-69. IEEE. October 1995.

[23] Whitman, S. et al. A Task Adaptive Parallel Graphics Renderer. *1993 Parallel Rendering Symposium Proceedings*, pages 27-34. July 1994.

[24] Wylie, B. et al. Scalable Rendering on PC Clusters. To appear in *IEEE Computer Graphics and Applications*. IEEE Computer Society. July/August 2001.
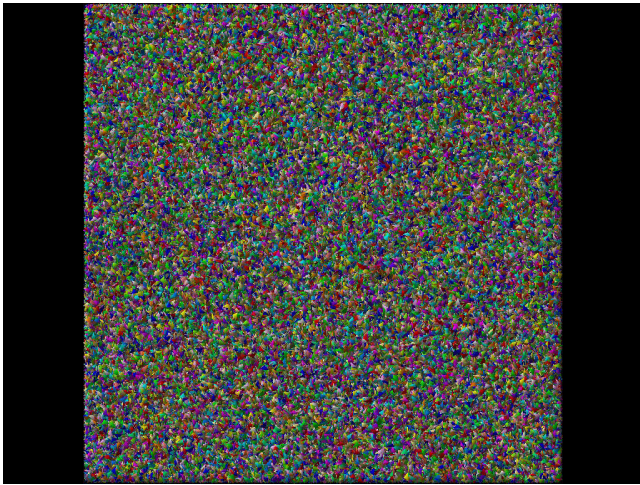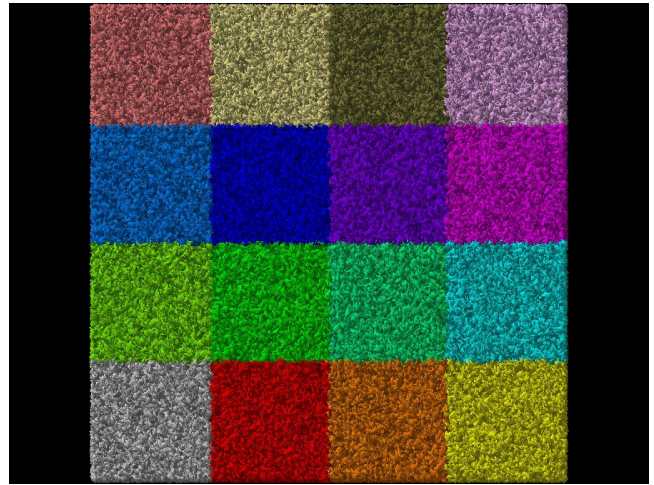
Figure 11: Linear distribution of random triangles.

Figure 13: A distribution of random triangles with (almost) perfect separation (i.e. spatial decomposition).
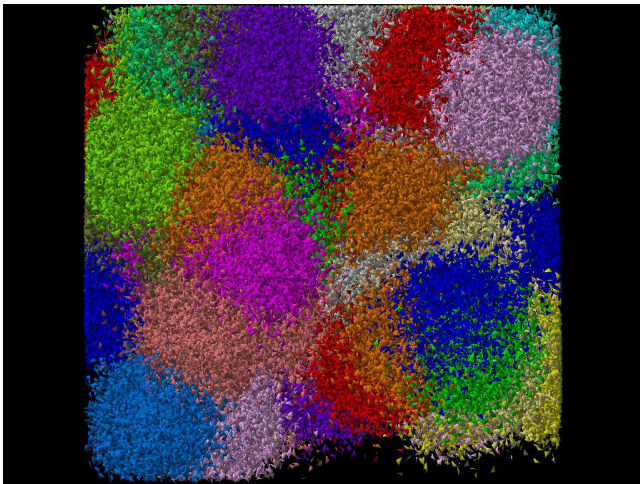

Figure 12: Gaussian distribution of random triangles.
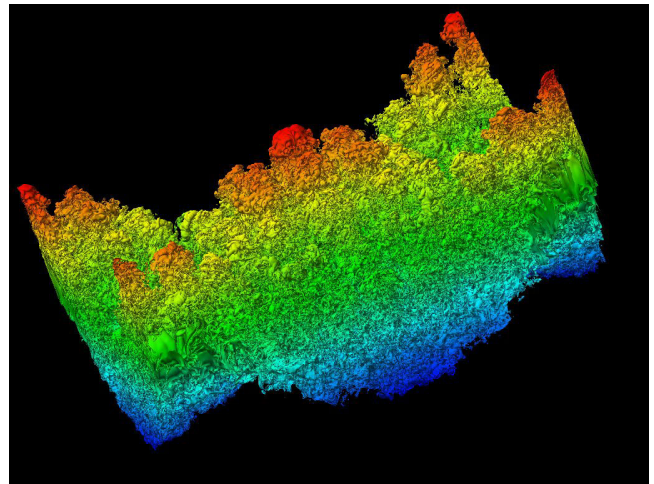

Figure 14: 469 million triangle isosurface from Lawrence Livermore National Laboratories. Images covered by LLNL: UCRL-MI-142527 Rev 1.
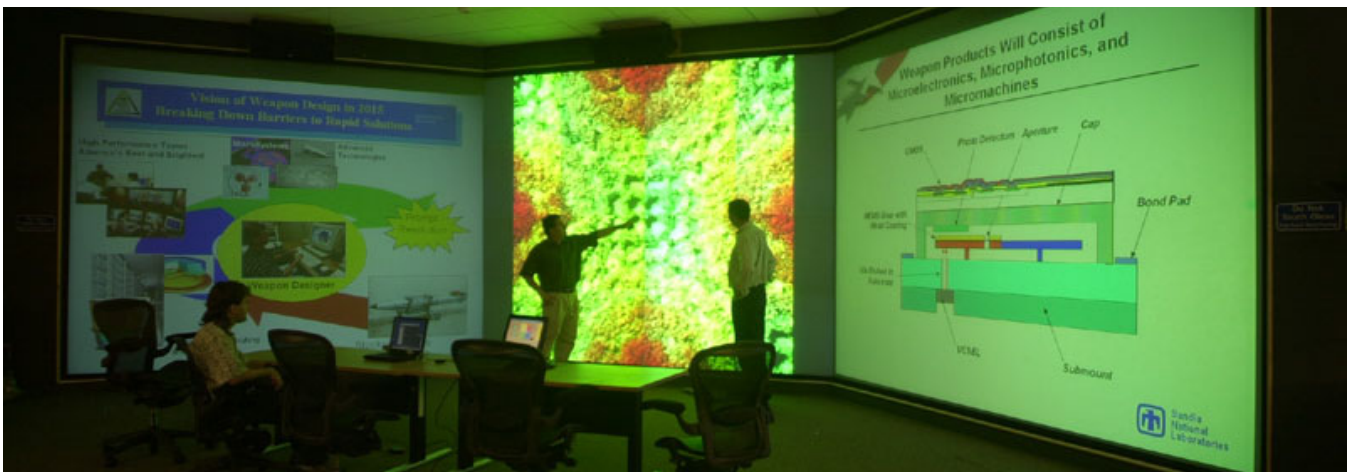

Figure 15: Sandia National Laboratories' VIEWS Data Visualization Corridor. The image of the LLNL isosurface displayed in the center is 5120 pixels wide by 4096 pixels high.

Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays
Kenneth Moreland, Brian Wylie, Constantine Pavlakos