Title
SLIC: Scheduled Linear Image Compositing for Parallel Vollume Rendering

Authors
Lum, Eric
Ma, Kwan-Liu
Ahrens, James
et al.

Publication Date
2003

Peer reviewed

SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering

Aleksander Stompel Kwan-Liu Ma Eric B. Lum
University of California at Davis
{stompel,ma,lume}@cs.ucdavis.edu

James Ahrens John Patchett
Los Alamos National Laboratory
{ahrens,patchett}@lanl.gov

Abstract

Parallel volume rendering offers a feasible solution to the large data visualization problem by distributing both the data and rendering calculations among multiple computers connected by a network. In sort-last parallel volume rendering, each processor generates an image of its assigned subvolume, which is blended together with other images to derive the final image. Improving the efficiency of this compositing step, which requires interprocesssor communication, is the key to scalable, interactive rendering. The recent trend of using hardware-accelerated volume rendering demands further acceleration of the image compositing step. This paper presents a new optimized parallel image compositing algorithm and its performance on a PC cluster. Our test results show that this new algorithm offers significant savings over previous algorithms in both communication and compositing costs. On a 64-node PC cluster with a 100BaseT network interconnect, we can achieve interactive rendering rates for images at resolutions up to 1024×1024 pixels at several frames per second.

CR Categories: C.4 [Computer Systems Organization]: Performance of Systems—; F.2.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity—Nonnumerical Algorithms and Problems; I.3.2 [Computer Graphics]: Graphics Systems—Distributed graphics

Keywords: high-performance computing, image compositing, parallel rendering, PC clusters, visualization, volume rendering

1 Introduction

Many scientific and medical investigations can produce high-resolution volume data sets that cannot be rendered at interactive rates on a single computer. A viable solution is to make use of a cluster of PCs to distribute both the volume data and the rendering calculations. Sort-last parallel volume rendering [Molnar et al. 1994] requires a final compositing step which involves inter-processor communication. Several parallel image compositing algorithms have been developed to address this need [Ma et al. 1994; Lee et al. 1996; Ahrens and Painter 1998; Yang et al. 2001]. These algorithms worked quite well until recently as the increased use of hardware accelerated rendering and the increased interest in building low-cost graphics clusters demand more efficient software algorithms. Hardware-accelerated rendering enables realtime rendering rates but in a parallel rendering setting the compositing step, if done in software, could become the performance bottleneck. That is, the scalability of the parallel rendering is determined by the parallel compositing, rather than the rendering calculations. Using a high-speed network interconnect and hardware compositing devices [Moll et al. 1999; Gordon et al. 2001; Muraki et al. 2001] is an effective but very expensive solution. This paper presents a new parallel image compositing algorithm designed with the goal to achieve significant improvement over previous algorithms. Our performance study on a 64-node PC cluster shows that with this new algorithm it is possible to use a low-cost network interconnect to build high-performance volume graphics PC clusters.

2 Parallel Image Compositing

Among parallel image compositing algorithms developed for sort-last distributed parallel volume rendering, the direct send, binary swap, and parallel pipeline algorithms are representative. Except direct send, most of the previous image compositing algorithms were designed for and only work well on specific types of network interconnect. In this section, we briefly review these three image compositing algorithms and recent development of compositing hardware devices.

2.1 Software Algorithms

All image compositing algorithms are concerned with partitioning the image space for compositing job assignments, delivering pixels, each of which stores RGBA values, from rendering of each subvolume to the designated compositing nodes, and computing final pixel values by compositing multiple pixels in depth order. Each compositing node is assigned some image areas to perform the corresponding compositing task. The image space assignments should be done in such a way that overloading a few particular compositing nodes will never happen. Rendering nodes produce pixels which are delivered to the compositing nodes responsible for the image areas to which these pixels contribute. In most settings, each processor switches between being the rendering node and the compositing node.

The simplest compositing technique is the direct send method, which has each processor send pixels directly to the processor responsible for compositing them. This ap-

proach has been used in [Hsu 1993; Neumann 1994; Ma and Interrante 1997] because its simplicity and ease of implementation. With direct send compositing, in the worst case there are $n(n-1)$ messages to be exchanged among n compositing nodes. For low-bandwidth networks care should be taken so that fewer nodes send messages to the same node at the same time.

Lee et al. [Lee et al. 1996] introduce a parallel compositing algorithm to avoid link contention on a mesh network. For an $m \times l$ mesh, compositing is first performed along the column direction and then along the row direction (or vice versa) in a pipelined fashion such that a total of $m + l - 2$ steps are taken. For each direction, the images (as well as their z-buffers for polygon rendering) are split into subimages each which is forwarded to the a processor on the same row (or column) in turn. Link contention is avoided because there is never two processors sending messages to the same processor at the same time. Further optimization can be obtained by using boundary box, pixel forwarding, static load balancing, and task switching, as discussed in [Lee et al. 1996].

A conceptually simple method is binary tree compositing which pairs up processors in order of compositing. One processor in each pair sends its image data to the other one for compositing so each disjoint pair of processor produces a new partial-image. Then the processors holding the new partial-images pair up for the next level compositing. Continuing this fashion, after $\log n$ stages where n is the total number of processors used, the final image is obtained. The problem with binary tree compositing is that at each stage of compositing half of the processors become idle. Finally, at the top of the compositing tree, only one processor is active, doing the final compositing for the entire image. When a massively parallel computer with hundreds to thousands of processors is used, compositing this fashion would become a serious bottleneck.

Ma, et al [Ma et al. 1994] improve binary tree compositing by keeping all processors actively participating in the whole course of a binary tree compositing process. The key idea is that in each stage rather than having only one node from each pair composite the whole image plane, the image plane is split into two pieces, and each node takes responsibility for one of the two pieces. Since a swapping of the pieces between the two nodes is needed the algorithm is called binary swap. In the early stages each node needs to composite a large portion of the image area, but the portion becomes smaller further up the compositing tree. Then after exactly $\log n$ stages, compositing is completed and each node holds $\frac{1}{n}$ of the final image. Binary swap can also better exploit parallelism. Especially, when a tree or hypercube network is used, it can take advantage of nearest neighbor communication. Since as the compositing proceeds the image each processor handles becomes smaller, it is beneficial to swap large images during earlier stages of the compositing between processors that are physically next to each other. Similarly, additional optimizations such as bounding box, compression [Ahrens and Painter 1998], etc., can be incorporated into binary swap. However, like any binary-tree based methods, binary swap is limited to using power of two processors.

2.2 Hardware Designs

Several specialized hardware architectures and devices have been developed to support real-time image compositing for demanding graphics applications using a cluster of graphics-enhanced PCs. Both Sepia [Moll et al. 1999], lightning-2 [Gordon et al. 2001], and Metabuffer [Blanke et al. 2000] were developed for the construction of large display subsystems for distributed clusters. Sepia is a commodity-based architecture implemented by custom PCI cards connected to a high speed network for image acquisition, compositing, and display. It supports pipelined associative blending operations in a sort-last configuration. The second generation of Sepia [Lombeyda et al. 2001] incorporates a high speed network interface, Servernet-2. Lightning-2 is a hardware system that employs scanline based pixel mapping and provides a DVI-to-DVI interface which delivers pixel data from graphics accelerators to remote tiled displays. It scales in both the number of rendering nodes and the number of displays supported, and allows any pixel data generated from any node to be dynamically mapped to any location on any display. Metabuffer, based on a mesh interconnect, is similar to Lightning-2 in supporting a rich set of viewport mappings but it also offers multiresolution support. Another compositing hardware design is based on binary-tree compositing [Muraki et al. 2001] resulting in reduced circuitry and scalable performance. While some of these hardware solutions have become commercially available they can be prohibitively expensive for the building of larger systems.

3 Scheduled Linear Image Compositing

SLIC is essentially a highly optimized direct send method. The optimizations are achieved by refining the direct send method based on the following observations. First, image space partitioning for compositing tasks is crucial to load balancing. A simple way to ensure load balancing without runtime overheads is to statically assign each processor image areas scattering the whole image space. An example is scanline interleaving. In this way, each processor's compositing load becomes less view dependent. In addition, fine-grain partitioning, generally giving more flexibility in load distribution, should be used. Second, after local rendering is done by each processor, there are three types of pixels: background pixels, pixels in the nonoverlapping areas, and pixels in the overlapping areas. Background pixels can be ignored. Pixels in the nonoverlapping areas can be delivered directly to the host or display device. Only the pixels in the overlapping areas need to be sent to the processors responsible for compositing the corresponding areas. Figure 1 shows pixels classification as a result of a particular projection. Once pixels are classified, an optimized compositing schedule for all processors and respective assignments can be computed. Note that each processor is assigned within the image space it renders into. With direct send or binary swap, a processor could be assigned compositing regions that it was not involved with in rendering, which results in additional sends. Lastly, it is generally true that communication is more expensive than computation. This is the case even more so since we aim to develop a low-cost solution such that expensive network interconnect is not required. It is thus desirable to perform additional computation for minimizing communication as much as possible.

To simplify the presentation of the SLIC and performance study, we assume rendering and compositing are completely separated. That is, the rest of the discussion is independent of the rendering method used. However, we do assume that a block partitioning scheme, as suggested in [Neumann 1994], is used to distribute the volume data among the nodes, and that an unambiguous, front-to-back order of the blocks can be straightforwardly determined. Consequently, each node

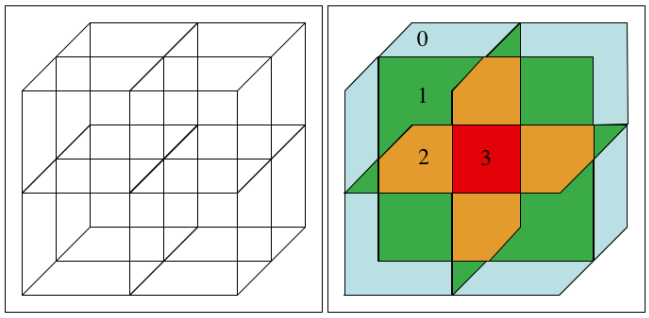


Figure 1: Left: Projection of the bounding boxes of eight subvolumes. Right: The image space is partitioned into areas according to the number of overlaps. In addition to the background, there are areas with no overlap (blue), one overlap (green), two overlaps (orange), three overlaps (red), etc.

starts with some pixels that correspond to the projection of the block of volume data the node renders, and ends with some image data to be delivered to the host node. Each node needs to perform three tasks: computing a compositing schedule, exchanging pixels with other nodes according to the schedule, and finally compositing the pixels. We use the word 'schedule' not in a temporal sense, but rather as a list that indicates how the task of compositing is distributed.

3.1 Computing the Compositing Schedule

Each node first computes a schedule independent of other nodes. The schedule is determined based on the overlapping relations between the projection of the local blocks of volume and the projection of other blocks. Because a regular data partitioning is used, each node also knows the exact projection of nonlocal blocks. Specifically, each node performs the following steps:

1. project corner vertices of each block based on the current view and constructing its convex hull,

2. traverse through the overlapped convex hulls in scanline order to identify compositing tasks in terms of spans, and

3. assign each span to a node in an interleaving fashion.

The convex hull defines the exact projected area of the block in the image space. A scanline algorithm similar to polygon scan-conversion is then used to process the edges of the overlapped convex hulls, as shown in Figure 2. Note that each node only needs to scan the projected bounding edges of the local blocks. In Figure 3, the color area shows the result of applying the scanline algorithm. However, the projected bounding edges of nonlocal blocks are used to determine the number of overlaps.

The edges that each scanline intersects break the scanline into multiple spans, which can be classified into: background spans, no-overlap spans, one-overlap spans, two-overlap spans, etc. Background spans are never generated. No-overlap spans are sent directly to the host node. The rest of the spans are either kept locally or delivered to other nodes by following Step 3. In our current implementation, the node assignment is determined by $(((x + y)\times p)$ `mod` $n)$ where x and y are the coordinates of the starting position

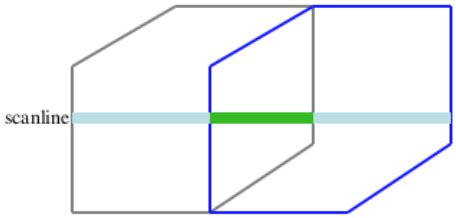


Figure 2: Scanline-order processing of the bounding edges of blocks to generate spans of compositing tasks for a two-block case. The green span is generated twice, once by the node rendering the left block and the other by the node rendering the right block.

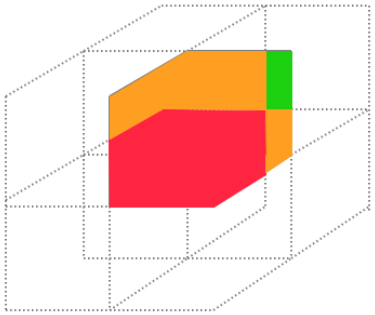


Figure 3: To compute a schedule, each node only needs to process the projected area of its local block (color zones) but the projected bounding edges of nonlocal blocks (gray dotted lines) are also needed to compute the number of overlaps.

of the span, p is a large prime number, and n is the total number of compositing nodes used. Figure 4 shows the distribution of the compositing tasks to eight processors for the rendering of a CT mouse dataset. The pixels in the gray area are directly sent to the host computer because there is no overlap. The same color spans are delivered to the same processor.

The scanline-based algorithm works because the bounding edges break the scanlines in exactly the same way across all nodes. As a result, if two spans created by different nodes would overlap, they must completely overlap; that is, the two spans have the same starting and end screen positions, as illustrated in Figure 5. Because all blocks are presorted in depth order, each span can be assigned a compositing order, which simplifies the actual compositing calculations. Other information stored with each span includes a sequence of RGBA values and the starting and end screen coordinates of the span.

This preprocessing step to compute a compositing schedule for each view introduces very low overhead, generally under 10 milliseconds for up to 64 processors with a 1 GHz Pentium III CPU. When more nodes are used, since the projected image area of local blocks decreases the cost does not increase. So in general the cost of computing the schedule is independent of the number of nodes used. With the resulting schedule, the total amount of data that must be sent over the entire network to accomplish the compositing task is minimized.

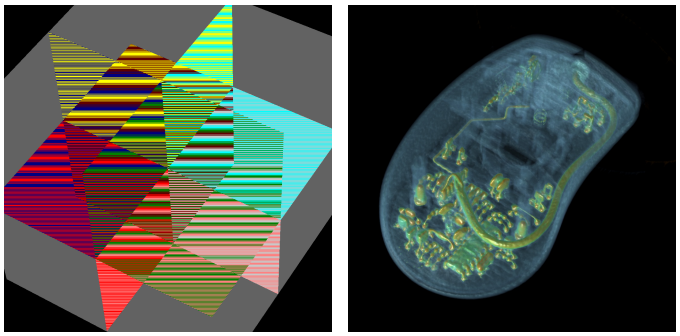


Figure 4: The left image illustrates the distribution of the compositing tasks to eight processors for parallel volume rendering of a CT mouse dataset. The pixels in the gray area are directly sent to the host computer because there is no overlap. The resulting image is shown on the right.

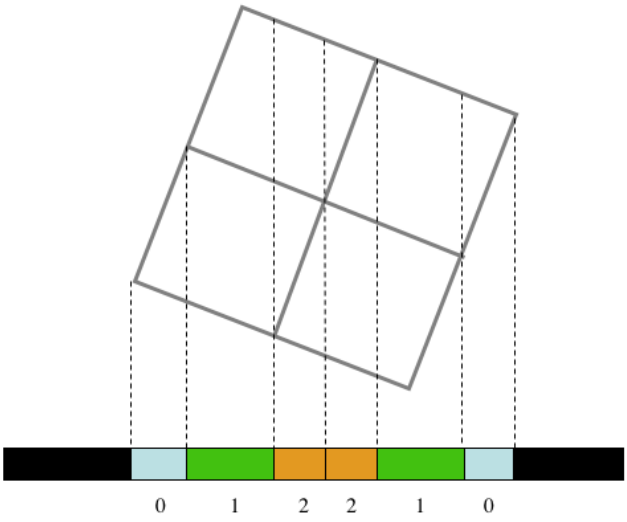


Figure 5: In 2D, the four blocks are processed by four different nodes but the spans mapping to the same image area must completely overlap because all nodes compute the compositing schedule by using the same set of projected bounding edges. For example, since there are two overlaps in the volume space corresponding to the yellow area, the same size spans are created independently by three different nodes. According to SLIC, one of the nodes will be assigned to composite these three spans. Note that 0 1 2 2 1 0 are number of overlaps.

3.2 Comparisons with Other Algorithms

Providing a block data distribution is used, the approximate depth overlap per pixel is $n^{(1/3)}$ where n is the number of processing nodes used. Consequently, if p is the total number of pixels in the final image there are a total of $n^{(1/3)}p$ image pixels and the approximate number of pixels per node is $n^{(-2/3)}p$. With direct send, Neumann [Neumann 1994] assumes that the image space subdivision is done in an interleaved fashion to ensure load balancing. The result is that the pixels each node produces are distributed among all n nodes, and about $(n^{(-2/3)}p)/(n-1) = n^{(1/3)}p(1-1/n)$ pixels must be transmitted. Since we do not make any assumption on the image space partitioning, on average the pixels each node produces are distributed among fewer nodes, about $n^{1/3}$. Consequently, each node needs to send approximately $(n^{(-2/3)}p(1-p^{-(1/3)})$ pixels to some other nodes, and thus the total number of pixels to be transmitted is $n^{1/3}p(1-n^{-(1/3)}))$. Asymptotically, this is comparable to both direct send's $n^{1/3}p(1-1/n)$ [Neumann 1994] and binary swap's $2.43n^{1/3}p$ [Ma et al. 1994].

Direct send could require sending $n(n-1)$ messages while binary swap sends exactly $n \log n$ messages. SLIC requires transmitting about $n^{(4/3)}$ (with a small constant factor) messages. This number is derived based on the observation that in SLIC the spans generated by each node are assigned to only those nodes sharing the same projected area of the local block. When 1024 or fewer processors are used, SLIC is comparable to binary swap for the number of messages a node must send to distribute the spans. However, SLIC is optimized to reduce the number of pixels that must be transmitted. Therefore, when the network is bandwidth limited, SLIC would perform better, especially for large images. When the network is latency limited, the compositing performance is determined more by the number messages that must be sent.

4 Test Results

We have done a set of tests on a 64-node PC cluster operated at the Los Alamos National Laboratory. Each node has a single 1GHz CPU, 512MB RAM, and a 100BaseT interconnect. Our objective was to study both the behavior and performance of the algorithm with a focus on the 100BaseT network interconnect because our strong interest in deriving a low-cost solution. Two data sets were used for our study. As shown in Figure 6 one is a CT scan of a Microsoft mouse with $205\times205\times259$ voxels, and the other is a confocal microscopic ganglion dataset with $600\times800\times129$ voxels. The size of a data set has very little impact on the performance of the image composition. Our study isolates the image composition timing, so we chose these smaller data sets to reduce the overall time needed to perform a large number of tests.

Figure 7 presents average compositing time of rendering from 36 different view points using from 2 to 64 processors for three different image sizes (512^2, 1024^2, and 2048^2 pixels). SLIC, direct send, and binary swap were tested to compare. The time plotted for SLIC includes including the time to compute the schedule, distribute the spans, compute the alpha compositing, and deliver the image data to the host computer. As the graphs show, SLIC outperforms both direct send and binary swap in all cases. Independent of the number of processors used, with SLIC the compositing time stays about the same, unlike using direct send or binary swap.

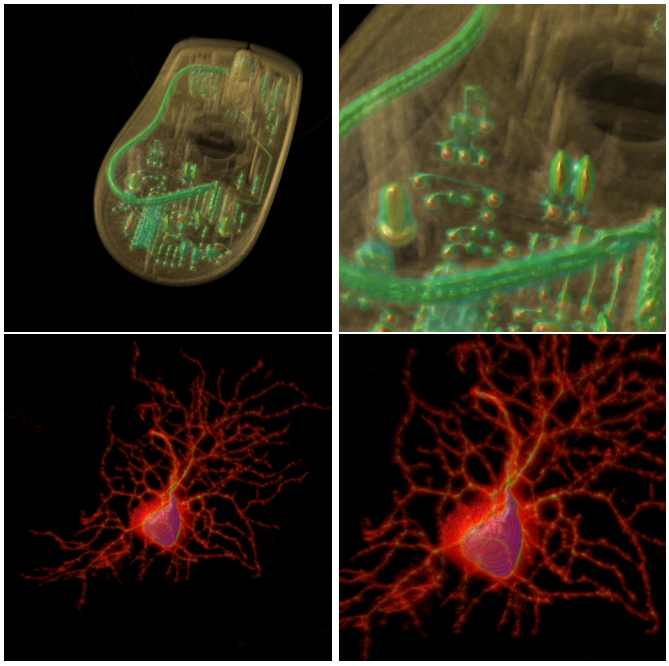


Figure 6: Top: CT mouse data, 205×205×259 voxels. Bottom: Confocal microscopic ganglion data, 600×129×800 voxels.

Table 1: Average compositing time (in seconds) with SLIC for rendering the mouse data.

#nodes	256×256	512×512	1024×1024	2048×2048
2	0.03	0.120	0.474	1.887
4	0.03	0.141	0.552	1.984
8	0.03	0.104	0.541	1.594
16	0.03	0.105	0.742	1.764
32	0.03	0.149	0.677	1.618
64	0.04	0.097	0.792	1.632

Table 1 shows the compositing cost using SLIC for four different image sizes. For rendering small images, SLIC allows for 10-30 frames per second. For 1024×1024, about 1-2 frames per second can still be achieved. These numbers also show that with SLIC the compositing cost does not increase as more processors are used. On the other hand, SLIC does not scale on the 100BaseT network interconnect.

Figure 8 compares the performance of SLIC with the other two compositing methods with the optimization that run-length encoding of transparent image regions is used. That is, the image data that is entirely transparent are not transmitted. This optimization does improve the performance of SLIC since it makes use of image segments that are tightly bound to the projected volume to begin with. As the graphs show, SLIC still outperforms direct send and binary swap in all cases.

Figure 9 shows timing results of using SLIC and both the optimized and unoptimized versions of direct send and binary swap for rendering from six different views. Timing results are plotted from left to right on the graph for rendering using increasingly close-up views. For the closest views,

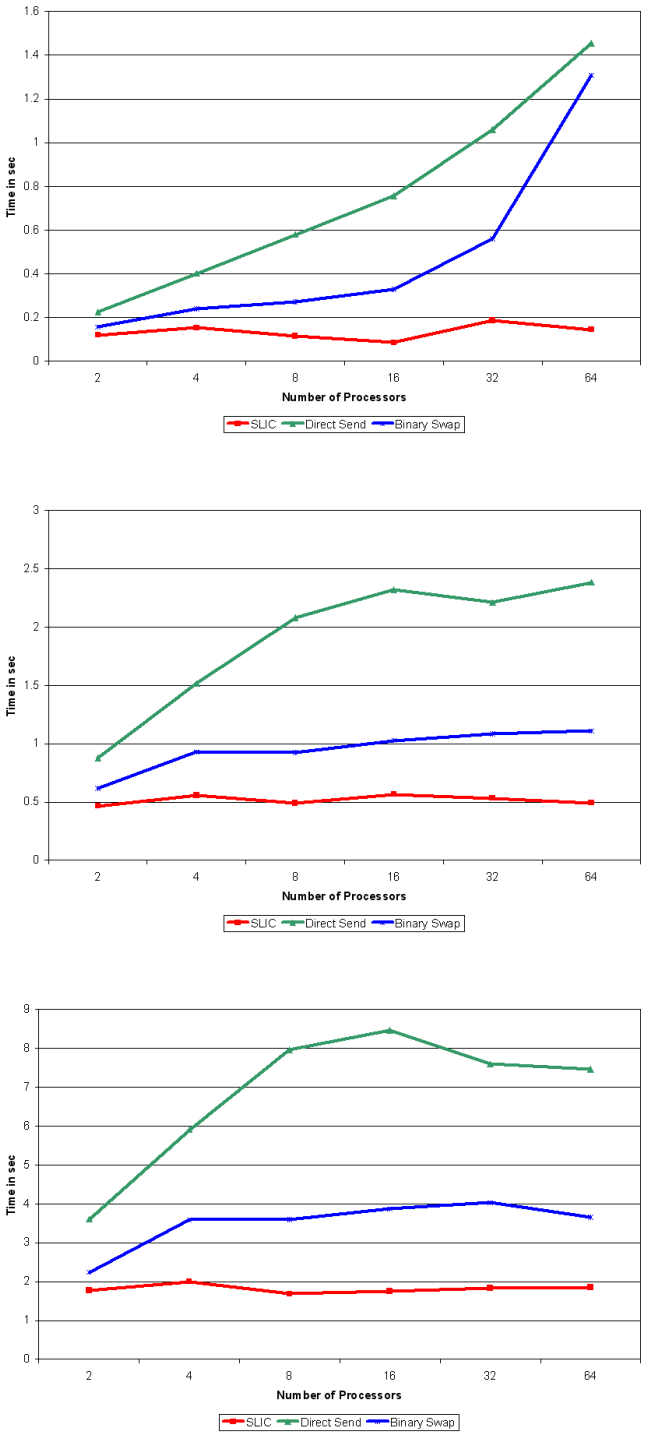
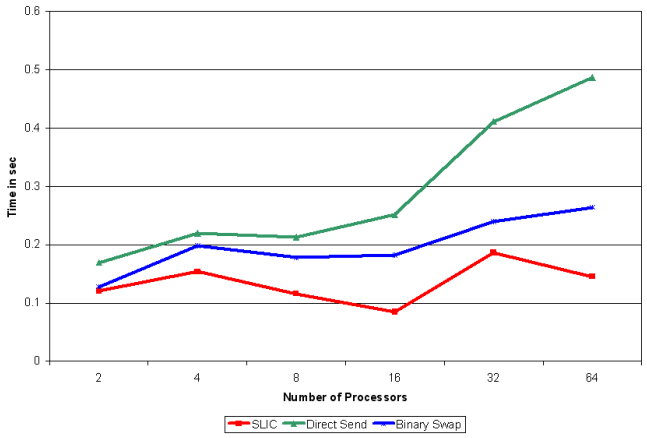
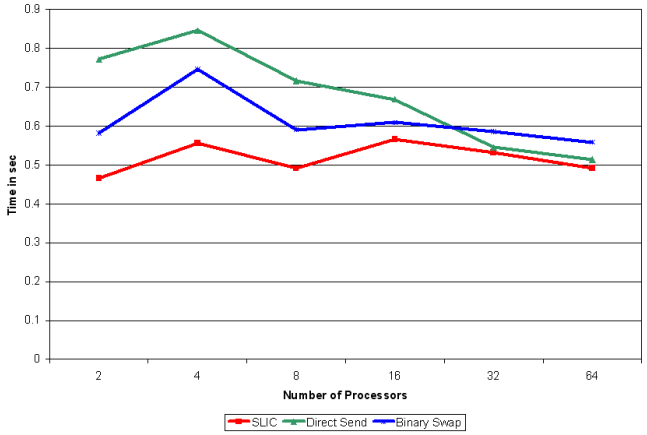


Figure 7: Comparing the performance of SLIC, direct send, and binary swap. Each graph plots the compositing time for images of resolutions at (top) 512×512, (middle) 1024×1024, and (bottom) 2048×2048 pixels using each compositing method. The time plotted is the average time of rendering from 36 different views, and includes the time to compute the schedule, deliver the spans, calculate the alpha compositing, and deliver the image data to the host. Up to 64 PCs were used for compositing.





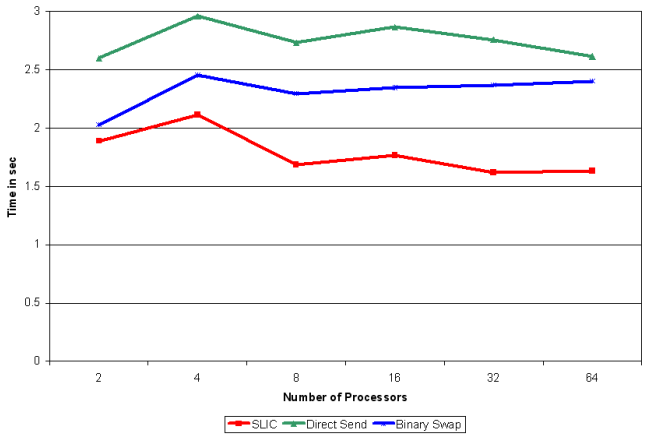
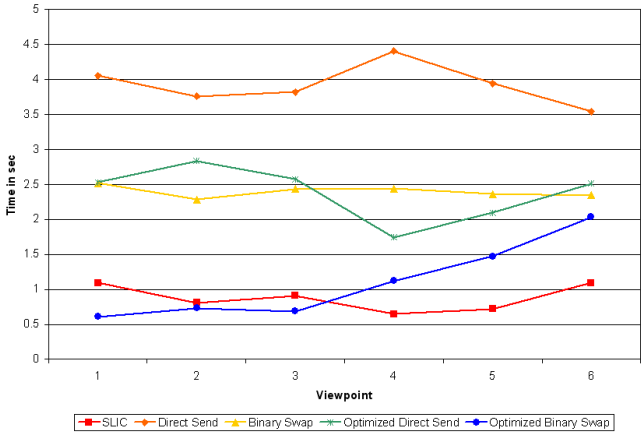


Figure 8: Comparing the performance of SLIC, direct send, and binary swap. Runlength encoding was used for direct send and binary swap. Each graph plots the compositing time for images of resolutions at (top) 512×512, (middle) 1024×1024, and (bottom) 2048×2048 pixels using each compositing method. Up to 64 PCs were used for compositing.



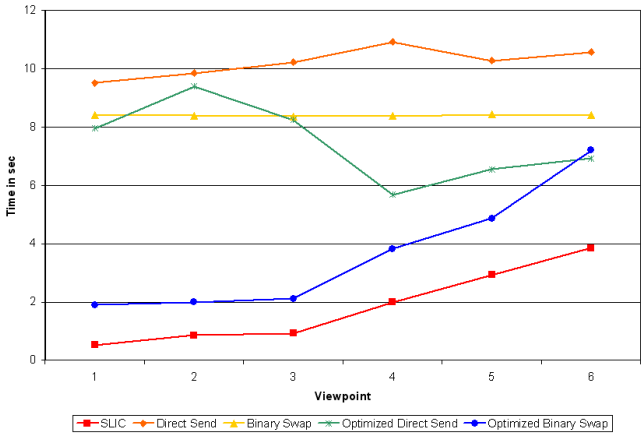


Figure 9: Compositing costs for each of six selected views using different compositing methods on 64 processors. Top: 1024×1024 pixels. Bottom: 2048×2048 pixels.

since there is little or no empty space, more pixels must be transmitted so the cost of compositing using SLIC increases.

In the 1024×1024 pixels case (top graph), for the first three views, the optimized binary swap (with runlength encoding) performs slightly better than SLIC because message overhead is the bottleneck. For close-up views, SLIC yields better performance. For the very large images like 2048×2048 pixels, SLIC is better for all cases.

The favorable performance of SLIC is due to the use of a compositing schedule, which can be computed quickly by each node independently. As shown in Figure 10, the cost is very small (several milliseconds), even with a large number of processors.

Figure 11 displays the average number of spans each node produces for each of 120 different views, and Figure 12 shows the total number of spans generated for each view. It is clear the results are highly view dependent. While the difference in the number of spans that must be processed can be quite large the difference in the overall compositing time is negligible.

Finally, Figure 13 shows the compositing time breakdown for each of the 32 processors for rendering the ganglion data to 512×512 pixels from a particular view. The cost of transferring the spans dominate the overall cost, which suggests that a faster network such as Myrinet would help to signifi-

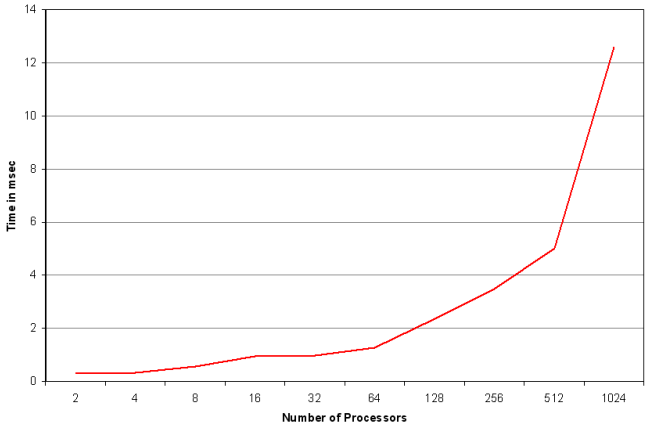


Figure 10: The time (in milliseconds) to compute the compositing schedule for different numbers of processors. Note that we can derive these numbers without actually using a large cluster because computing the compositing schedule is independent of the rendering nor the data size or content. The cost for each processor size would be the same for the same image resolution, in this case 512×512, and the same view point.

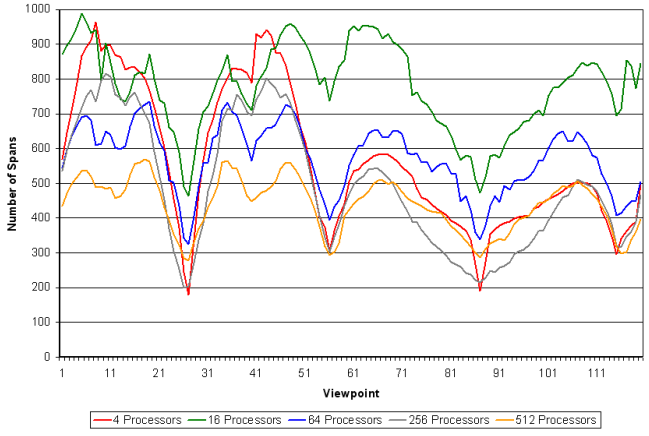


Figure 11: Average number of spans generated by a node for each of 120 different views. Note that we can derive these numbers because computing the schedule (and thus the number of spans) is independent of the rendering and volume data properties.

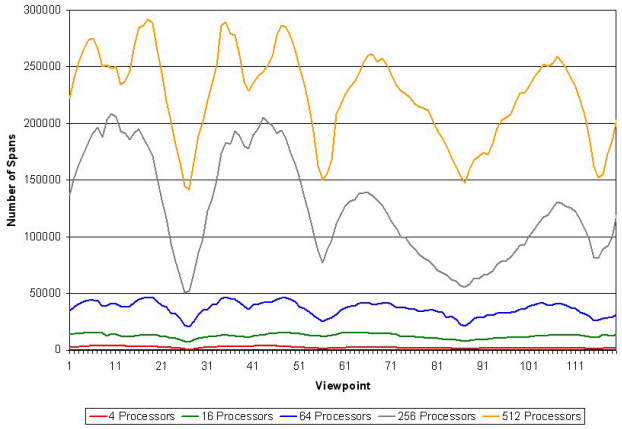


Figure 12: Total number of spans generated by all processors for each of 120 different views.

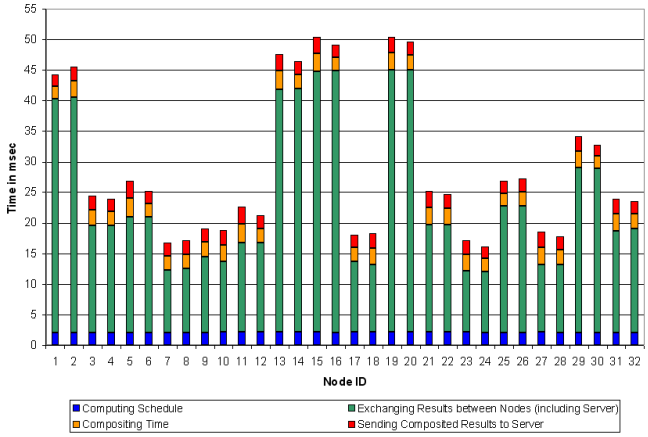


Figure 13: Overall compositing time breakdown for each of the 32 processors for rendering the ganglion data to 512×512 pixels from a particular view.

cantly reduce the compositing cost.

SLIC outperforms the optimized direct send and binary swap methods when high image resolutions are desired. Figure 14 displays the estimated "cross-over" line where SLIC outperforms the other optimized methods for the rendering of the mouse data . Without using runlength encoding for direct send and binary swap, the line stays between 300×300 and 500×500 suggesting that SLIC is favorable for any image resolution above 500×500.

5 Conclusions

We have presented a new image compositing algorithm, SLIC, optimized for reducing pixel coimmunication by using a compositing schedule computed on-the-fly. The cost of computing the schedule is very small and only depends on the image resolution and the number of processors used. Unlike binary swap, SLIC is not limited to using a number of processors that is a power of two. Furthermore, SLIC is especially efficient for the rendering of large images, which are required for the increasingly used large display spaces.

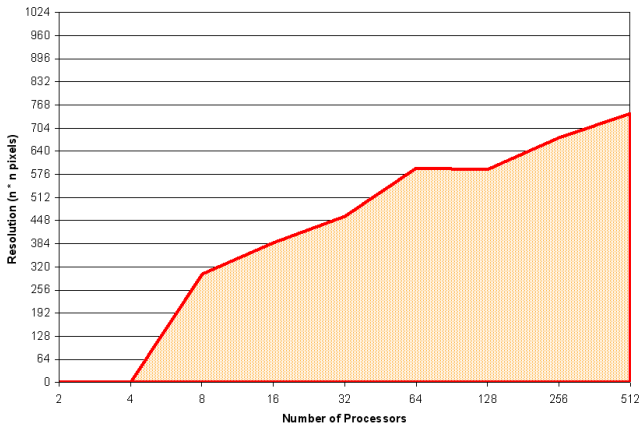


Figure 14: Estimated "cross-over" performance line for SLIC when compared with other methods. The shade area represents resolutions for which the optimized direct send or binary swap (with compression) outperforms SLIC.

According to our test results, it is clear that now we can build low-cost graphics PC clusters without using expensive network interconnect. However, those realtime graphics applications requiring above 30 frames per second display rates would still have to adopt hardware support. We have also tested SLIC on large clusters using up to 512 processors, and the test results show that the compositing cost stays almost constant even as more processors are used.

Acknowledgments

This work has been sponsored in part by the U.S. National Science Foundation under contracts ACI 9983641 (PECASE award), ACI 9982251 (the LSSDSV program), and ACI 0222991; the U.S. Department of Energy under Memorandum Agreements No. DE-FC02-01ER41202 (SciDAC program) and No. B523578; the National Institute of Health through the Human Brain Project, and a United States Department of Education Government Assistance in Areas of National Need (DOE-GAANN) grant P200A980307. The authors would especially like to thank John Owens at UCD and the anonymous reviewers for their comments on this paper.

References

Ahrens, J., and Painter, J. 1998. Efficient sort-last rendering using compression-based image compositing. In *Proceedings of the 2nd Eurographics Workshop on Parallel Graphics and Visualization*, 145–151.

Blanke, W. J., Fussell, D. S., Bajaj, C., and Zhang, X. 2000. The metabuffer: A scalable multiresolution multidisplay 3-d graphics system using commodity rendering engines. Tech. Rep. Technical Report No. 2000-16, University of Texas at Austin.

Gordon, S., Eldridge, M., Patterson, D., Webb, A., Berman, S., Levy, R. Caywood, C., Taverira, M., Hunt, S., and Hanrahan, P. 2001. A high performance display subsystem for PC clusters. In *Proceedings of SIGGRAPH 2001*, 141–148.

Hsu, W. M. 1993. Segmented ray casting for data parallel volume rendering. In *Proceedings of 1993 Parallel Rendering Symposium*, 7–14.

Lee, T.-Y., Raghavendra, C. S., and Nicholas, J. B. 1996. Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2*, 3, 202–217.

Lombeyda, S., Moll, L., Shand, M., Breen, D., and Heirich, A. 2001. Scalable interactive volume rendering using off-the-shelf components. In *Proceedings of 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, 115–121.

Ma, K.-L., and Interrante, V. 1997. Extracting Feature Lines from 3D Unstructured Grids. In *Proceeding of Visualization '97 Conference (to appear)*.

Ma, K.-L., Painter, J., Hansen, C., and Krogh, M. 1994. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications 14*, 4 (July), 59–67.

Moll, L., Heirich, A., and Shand, M. 1999. Sepia: Scalable 3-d compositing using PCI pamette. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 146–155.

Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. 1994. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (July), 23–32.

Muraki, S., Ogata, M., Ma, K.-L., Koshizuka, K., Kajihara, K., Liu, X., Nagano, Y., and Shimokawa, K. 2001. Next generation visual supercomputing using PC cluster with volume graphics hardware devices. In *Proceedings of Supercomputing 2001 Conference*.

Neumann, U. 1994. Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications 14*, 4 (July), 49–58.

Yang, D.-L., Yu, J.-C., and Chung, Y.-C. 2001. Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *The Journal of Supercomputing 18*, 2 (February), 201–220.