LA-UR-98-2968
Approved for public release;
distribution is unlimited.

Title: **Efficient Sort-Last Rendering Using Compression-Based Image Compositing**

Author(s): James Ahrens
James Painter

Submitted to:

http://lib-www.lanl.gov/cgi-bin/getfile?00783056.pdf

Efficient Sort-Last Rendering Using Compression-Based Image Compositing

James Ahrens James Painter

Advanced Computing Laboratory
Los Alamos National Laboratory

Abstract

State of the art scientific simulations are currently working with data set sizes on the order of a billion cells. Parallel rendering is a promising approach for interactively visualizing multiple isosurface variables from data sets of this magnitude. In sort-last rendering, each processor creates a depth buffered image of its assigned objects. All processors' images are composited together to create a final result. Improving the efficiency of this compositing step is key to interactive parallel rendering. This paper presents a compression-based image compositing algorithm which can provide significant savings in both communication and compositing costs.

1 Introduction

Parallel polygon rendering techniques are a promising approach to visualizing the massive scientific data sets generated by parallel simulations. For example, scientists working on the United States Department of Energy's Accelerated Strategic Computing Initiative (ASCI) project are currently generating data sets on the order of a billion cells and are projected to generate order of magnitude increases in size every two years. Interactive techniques are critical to understanding these data sets. We are creating a parallel visualization software framework to interactively visualize massive data sets such as those produced by the Department of Energy's ASCI and Grand Challenge programs. The framework makes use of existing commercial and freeware visualization and graphics software including the Visualization Toolkit, OpenGL and Mesa. A key advantage of using this approach is the leverage gained by using the serial algorithms in these packages. For example, the Visualization Toolkit contains imaging, visualization and graphics libraries. Within the framework, independent processes execute serial visualization algorithms on partitioned subsets of a massive data set, communicating boundary elements when necessary. Each process uses Mesa, for software rendering, or OpenGL, for hardware accelerated rendering. A final image compositing step is executed which merges these images into a result image using a sort-last rendering algorithm. In sort-last rendering, each processor creates a depth buffered image of its assigned objects. A compositing algorithm inputs two images and iterates through pixel pairs (i.e. two pixels with corresponding locations in the input images).

These pixels are read and compared and the pixel closer to the viewer is output to the result image[5]. A significant disadvantage of sort-last algorithms is the amount of image data which must be exchanged during compositing. This paper describes a compositing scheme that communicates compressed images in order to reduce message traffic. In addition, a compositing scheme is presented that composites these compressed images. When compositing compressed images this algorithm can be faster than standard pixel-by-pixel compositing method. Using a compressed image representation can improve the performance of many sort-last compositing methods. This paper include results for binary tree compositing on a shared memory multiprocessor.

2 Related Work

Previous work has explored the use of compression to speed up image compositing. Lacroute and Levoy [2] explored the use of run-length encoding and compressed compositing for a serial volume renderer. This paper promotes the use of run-length encoding and compositing as a general technique applicable to many sort-last parallel image compositing algorithms. Ma et. al. [4] suggest tracking a bounding rectangle of non-blank pixels for each image and only compositing pixels within the intersection of these rectangles. Lee et. al.[3] also used this technique as part of a parallel pipelining image compositing algorithm. The authors report a degradation in performance of 20 to 50 percent for their test data when the bounding box optimization is not used. We expect to achieve better compression ratios from run-length compression and compositing than bounding box optimization because the run-length algorithm can compress pixels over the entire image instead of only outside a bounding box. Another compositing algorithm, distributed snooping for pixel merging by Cox and Hanrahan [1], reduces message traffic by forwarding depth buffer information from a process to all other processes and compositing this information locally. Each forwarding of the depth buffer reduces the number of pixels in the remaining depth buffers that need to be forwarded. Values are forwarded with an annotation of their x,y position in the image. Run-length encoding and compositing provides a possibly superior alternative to sending x,y positions, since explicit location information does not need to be sent with each pixel.

3 A Compression-based Image Compositing Algorithm

The key idea of this work is the representation and manipulation of images as run-length encoded objects. Run-length encoding is a lossless compression technique. Adjacent pixels that have the same value are represented as a single instance of the value along with a count of the number of identical pixels. In this paper, a pixel value is represented using depth, red, green, blue and count fields.

A compression algorithm that creates this representation is now described: the algorithm stores a pixel for comparison, initially this stored pixel is the first element. Iterating through the pixels of the image, the algorithm compares the stored pixel to the current one. If their values are equal, the stored pixel's count is incremented. Otherwise the stored pixel is output into the resulting image and the stored pixel's value is set to the current value with a count field of 1. This algorithm runs in $O(n)$ where n is the total number of

pixels in the input image.

An algorithm for compositing compressed images is now described: in the standard pixel compositing algorithm, two images are input and a result image is output. The standard algorithm iterates thorough pixel pairs (i.e. two pixels with corresponding locations in the input images). These pixels are read and compared and the pixel closer to the viewer is output to the result image. When the images to be composited contain runs, two additional cases need to be considered. The first is the case in which one image contains a run and the other image does not. In this case, a pixel is extracted from the run, the run count is decremented and the comparison continues as usual. In the second case, when both images contain runs, the runs of pixels can be composited together avoiding pixel-by-pixel comparison. The length of the runs that can be composited is equal to the smaller of the two run counts. This count is subtracted from the greater of the runs count and the remaining run is left as part of its input image. Then the pixel runs are compared as usual and the closer run is output.

As the output of the compositing algorithm is generated it can either be re-compressed or uncompressed. To recompress the output, the compression algorithm is run incrementally, storing and creating runs of pixels as they are generated. To uncompress the output image the algorithm notes if a generated output pixel contains a run. If it does then the run is uncompressed immediately into the result image. A useful optimization to the compressed composite algorithm is a test to decide whether to uncompress the output data. For an image without much compression it is more efficient to use the standard compositing algorithm. Thus, the compressed composite algorithm could generate uncompressed images and use the standard algorithm when it is more efficient to do so. The standard and compressed compositing algorithms run in $O(n)$ where n is the total number of pixels in the input images.

The amount of data compression obtained from the compression and compressed compositing algorithm is dependent on the number of the constant runs of pixels in their input images. Runs can result from background pixels as well as constant pixel values across polygon scanlines. The compressed compositing algorithm creates image sizes of n in the worst case and a single compressed pixel in the best case.

4 Application to Binary Tree Compositing

To test the performance of the compressed compositing algorithm two versions of a binary tree compositing algorithm were created. The first version uses the standard compositing algorithm and the second uses the compressed compositing algorithm. In the binary tree algorithm each process starts with an initial depth buffered image. A tree communication pattern is created between the processes. Each parent process receives a depth buffer image from each of their two children processes. These images are composited and the result is sent to the process's parent. A final result is available from the process that is the root of the binary tree. The run-time of the binary-tree compositing algorithm is based on the number of levels in compositing tree, which is $\log_2 P$, where P is the number of processes. For the compressed compositing algorithm each local image is first compressed with a cost of $O(n)$. At each level, pixel data is communicated and composited with a cost of $O(n)$ for both the standard and compressed compositing algorithms. In summary, the binary tree version of the standard compositing algorithm runs in $O(n \log P)$ and the binary tree version of the compressed compositing algorithm also runs in $O(n \log P)$. This analysis shows that the

difference between the run times of the standard and compressed compositing algorithms is dependent upon constant factors. These constant factors are identified in the equations below:

For the standard compositing algorithm the cost is:

$$(C_{send}n + C_{composite}n)\log_2 P$$

- n is the total number of pixels in the input image.
- P is the total number of processes.
- C_{send} is the constant factor associated with sending data.
- $C_{composite}$ is the constant factor associated with the standard compositing algorithm.

For the compressed compositing algorithm the cost is:

$$\max_{proc=0}^{P-1}(C_{compress}n + C_{uncompress}n + \sum_{level=0}^{(\log_2 P)-1}(C_{send}(n-m) + C_{compressed\_composite}(n-o)))$$

- $C_{compress}$ is the constant factor associated with initial image compression algorithm.
- $C_{uncompress}$ is the constant factor associated with decompression of the final result. Note that the uncompress step actually occurs within the compressed compositing algorithm but is identified separately in this analysis for clarity.
- $C_{compressed\_composite}$ is the constant factor associated with the compressed compositing algorithm.
- m is the total number of compressed pixels minus the number of runs needed to represent these pixels for node $(level, \lfloor \frac{proc}{2^{level}} \rfloor)$. The indexing scheme for nodes in the binary tree is $(level, id)$. Level 0 is the leaf level and levels are incrementally assigned up to level $\log_2 P$ which is the root level. On each level, nodes are assigned ids incrementally starting at 0 and increasing to $\lfloor \frac{(P-1)}{2^{level}} \rfloor$. An example of tree indexing for P = 4 is shown in Figure 1.
- o is length of overlap in the runs minus the number of overlapping runs from node $(level, \lfloor \frac{proc}{2^{(level+1)}} \rfloor * 2)$ and node $(level, \lfloor \frac{proc}{2^{(level+1)}} \rfloor * 2 + 1)$. Note that a single run on one node can overlap with many runs on the another node. Each overlap is counted.

A number of observations can be made based on this analysis. Notice that the run time of the standard compositing algorithm is only dependent upon constant factors, image size and number of processors. In addition the compressed compositing algorithm is also a data dependent algorithm providing performance improvements based on the how well the input images can be compressed and composited along the worst

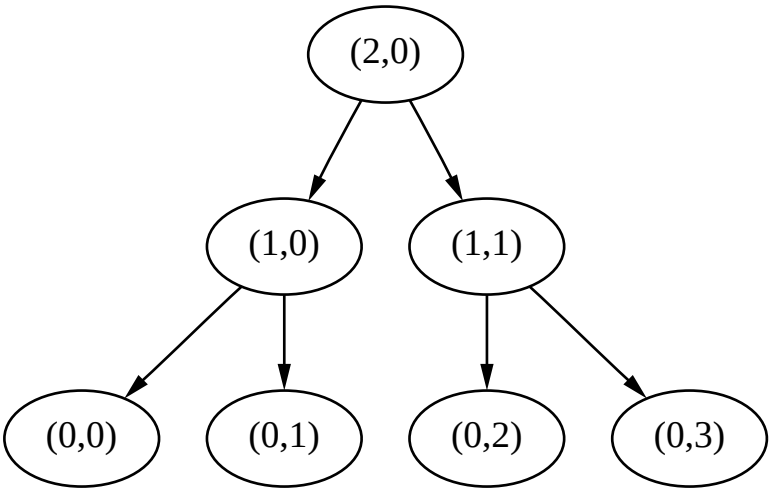

Figure 1: Example of tree indexing for P = 4

case path in the binary compositing tree. In limit, when all images are fully compressible (i.e. m and o are equal to $n-1$) the run time of the compressed compositing algorithm reduces to $O(n+\log p)$.

It is interesting to compare the cost equations. Notice that the compressed compositing algorithm pays an extra cost of $C_{compress}n$ to initially compress the input data and $C_{uncompress}n$ to uncompress the final result over the standard algorithm. Thus for small number of processes, P, the standard algorithm may provide better performance than the compressed algorithm. This is shown to be the case empirically in the next section in Table 6. Also notice that the constant factors ($C_{composite}$ and $C_{compressed\_composite}$) for the composite steps are different for the standard and compressed compositing algorithms. The constant factor for the compressed algorithm is greater than the standard algorithm because the source code for the standard algorithm can be better optimized by the compiler since it is a simple loop with a single conditional. The compressed compositing algorithm contains multiple conditionals and cannot be optimized as well.

Finally, notice that if the cost of communication dominates over the computation (i.e. C_{send} is significantly greater than $C_{compress}$, $C_{uncompress}$ and $C_{compressed\_composite}$) and the input images are amenable to compression then the compressed compositing algorithm is the algorithm of choice since it can reduce communication time by sending less data.

5 A Performance Study

The binary tree compositing algorithm was executed on an SGI Origin 2000 symmetric multiprocessor (SMP) with 64 processors, 16 Gigabytes of total memory and 8 Megabytes of secondary cache per processor. A shared memory message passing library called ACLMPL [6] is used to pass messages between processes. In order to test the performance of binary tree compositing algorithms, tests were run using the parallel visualization framework described briefly in the introduction. A program to visualize a medical

data set in parallel was written. A volume of CT (Computed Tomography) data (93 slices which are 64x64 images) is partitioned by slices to each process. Each process reads its slices and create two isosurfaces, one representing skin and the other bone. These isosurfaces are then rendered and the resulting images are composited together. Figure 2 shows a standard and an exploded view of the slices to be partitioned to 8 processes for a face-on view of the data set.

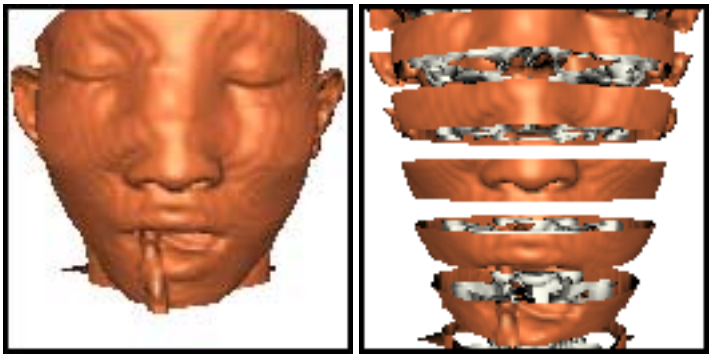

Figure 2: Standard and exploded view of slices

All tests where run on an image size of 1023x1023 pixels on 32 processes unless otherwise noted. We used an image size of 1023x1023 in order to avoid the cache conflicts that result with power of 2 array sizes on the Origin 2000s. Two views of the CT data set where run, view 0, provides a face-on view of the data set. Table 1 presents the execution time in seconds and speedup ratios for a set of four images with different zoom factors. The resulting images are shown in Figure 3. Table 2 presents the total data size transferred and compression ratios achieved.

Distance from Object - view 0	Far	Closer1	Closer2	Near
Compressed Compositing	0.20 (4.5:1)	0.29 (3.1:1)	0.55 (1.7:1)	0.84 (1.1:1)
Standard Compositing	0.90	0.91	0.94	0.93

Table 1: Compositing time in seconds and speedup ratios

Distance from Object - view 0	Far	Closer1	Closer2	Near
Compressed Compositing	127,225 (255:1)	720,980 (45:1)	2,859,607 (11:1)	6,096,363 (5:1)
Standard Compositing	32,442,399	32,442,399	32,442,399	32,442,399

Table 2: Total data transferred in long integers (8 bytes) and compression ratios

Notice that using run-length encoding can improve performance on an SMP by up to a factor of 4 depending on the contents of the composited images. Also notice the significant total compression ratios. These ratios show that using compressed compositing will have an even more significant impact in environments in which communication costs (i.e. bandwidth limitations) dominate over computation. Examples include cross-box communication on the Origins and workstation clusters[1].

[1]The compression ratio of 255 to 1 is due to the implementation. The value of count is limited due to the number of bits used to

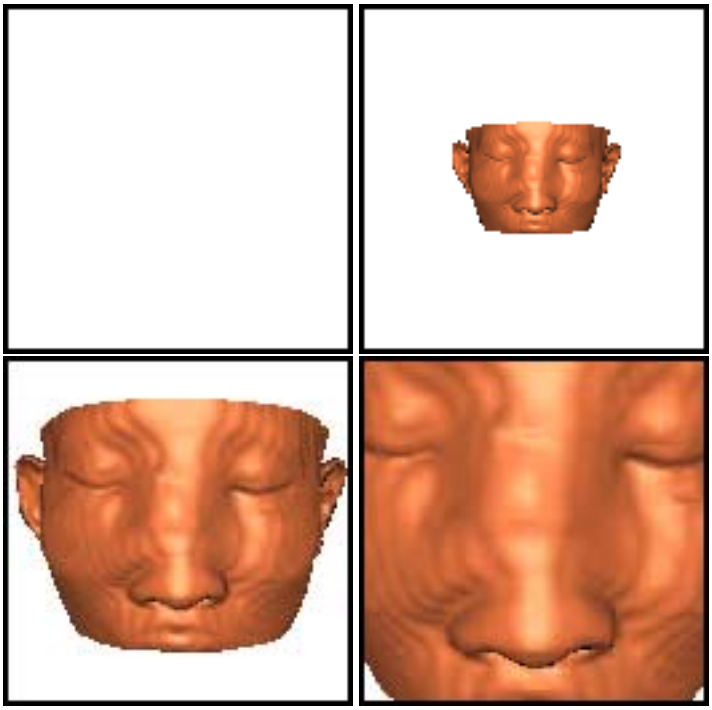

Figure 3: Result Images for view 0 for Far, Closer1, Closer2, Near (images arranged from left to right, top to bottom)

Results for a second view, view 1, of the CT data set are given in Table 3 and Table 4. The second view looks up thorough the head. Figure 4 shows these generated results.

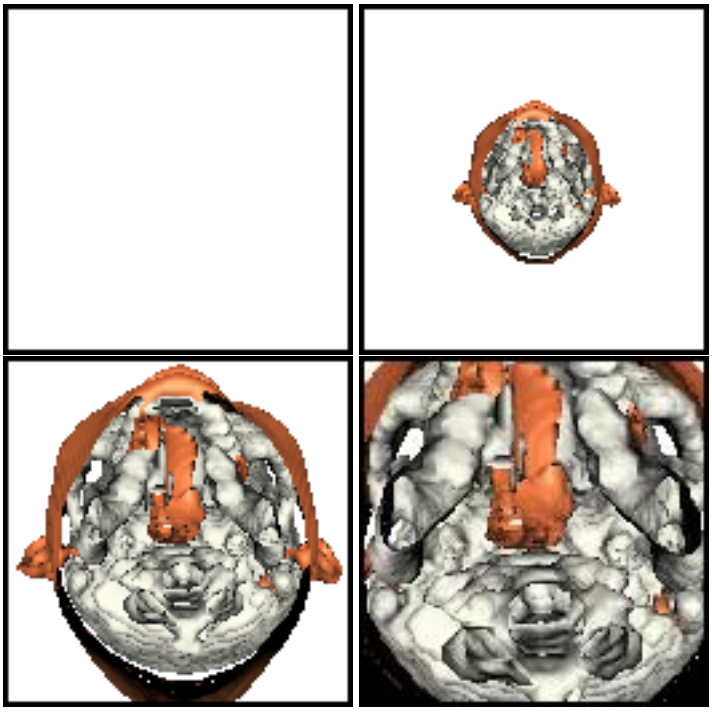

Figure 4: Result Images for view 0 for Far, Closer1, Closer2, Near

Distance from Object - view 1	Far	Closer1	Closer2	Near
Compressed Compositing	0.20 (4.5:1)	0.39 (2.4:1)	0.95 (1.1:1)	1.35 (0.7:1)
Standard Compositing	0.91	0.95	1.00	1.01

Table 3: Compositing time in seconds and speedup ratios

One important different between view 0 and view 1 is the viewing angle on the partitioned slices. In view 0, processes were assigned slices that project on only a small portion of the result image, whereas in view 1 each process displays a full cross section of the data set. Therefore the compression ratios and resulting execution time improvements are not as dramatic as with view 0. View 0, can be thought of as a representative best case for this data set and view 1 as a representative worst case, with the performance characteristics for other views falling between these endpoints.

More details of the execution time of the procedures executed by the binary tree algorithm are provided in Table 5. Notice that on an SMP the time to communicate and composite are of the same order of magnitude. This is because message passing simply require a memory copy within an SMP and this is similar to computation done by the compositing algorithm. Because of low communication costs, the gain from compressed

represent the field (in this case, 8 bits). The compression algorithm is amended as follows: An additional test checks whether the count is greater than the allowable size. If the limit is reached the run is written and a new run is started.

Distance from Object - view 1	Far	Closer1	Closer2	Near
Compressed Compositing	127,255 (255:1)	1,630,332 (20:1)	5,747,078 (6:1)	10,051,629 (3:1)
Standard Compositing	32,442,399	32,442,399	32,442,399	32,442,399

Table 4: Total data transferred in long integers (8 bytes) and compression ratios

Closer2 - view 0	Compress Time	Communication Time	Composite Time	Total Time
Compressed Compositing	0.13	0.10	0.10	0.55
Standard Compositing	0.00	0.47	0.35	0.94

Table 5: Time in seconds for procedures within compositing algorithms

compositing is smaller on an SMP architecture than it would be on a distributed memory architecture. In spite of this, we were still able to obtain up to a factor of four improvement.

In order to test the scalability of the algorithm we ran the algorithm on 4 to 32 processors of the SGI Origin. The results are shown in Table 6. Notice that the performance of the compressed compositing algorithm provides a speedup when compared to the standard algorithm as the number of processes increases. This is because as the number of processes grows so does the number of communications. The compressed compositing algorithm reduces the cost of these communciations by compressing the data to be sent.

Closer2 - view 0	4 processes	8 processes	16 processes	32 processes
Compressed Compositing	0.41	0.50	0.55	0.55
Standard Compositing	0.38	0.56	0.75	0.92

Table 6: Total binary tree composite execution time in seconds for varying number of processes

6 Conclusions and Future Work

The paper presents a general technique to reduce the time required to communicate and composite images. Results show up to a factor of four improvement in execution time and create compression ratios ranging from 255:1 to 3:1 on the tested images. In the future we will use and test the compressed compositing algorithm as part of other sort-last algorithms such as binary swap and Cox's pixel merging algorithm. In addition, we would like to test these algorithms on a distributed cluster of workstations. Another area for future exploration consists of modifying an existing renderer to generate compressed results directly, thus avoiding the creation and re-compression of a full sized image. We are also interested in using this technique as part of a remote image delivery system for distance visualization since it can produce compressed final images directly.

7 Acknowledgments

We would like to thank the other members of the ACL Visualization Team, Patrick McCormick and Allen McPherson for their support and advice. We acknowledge the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 87545. This work was performed on computing resources located at this facility.

References

[1] M. Cox and P. Hanrahan. A distributed snooping algorithm for pixel merging. *IEEE Parallel and Distributed Technology*, 1994.

[2] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *SIGGRAPH 94 Conference Proceedings*, 1994.

[3] T. Lee, C. Raghavendra, and J. Nicholas. Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 1996.

[4] K.L. Ma, J. Painter, C. Hansen, and M. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics*, 1994.

[5] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics*, 1994.

[6] J. Painter, P. McCormick, M. Krogh, C. Hansen, and G. Verdiere. ACLMPL: Portable and efficient message passing for MPPs. *EPFL Supercomputing Review*, 1995.