Comparing Binary-Swap Algorithms for Odd Factors of Processes

Kenneth Moreland*
Sandia National Laboratories

Abstract

A key component of most large-scale rendering systems is a parallel image compositing algorithm, and the most commonly used compositing algorithms are binary swap and its variants. Although shown to be very efficient, one of the classic limitations of binary swap is that it only works on a number of processes that is a perfect power of 2. Multiple variations of binary swap have been independently introduced to overcome this limitation and handle process counts that have factors that are not 2. To date, few of these approaches have been directly compared against each other, making it unclear which approach is best. This paper presents a fresh implementation of each of these methods using a common software framework to make them directly comparable. These methods to run binary swap with odd factors are directly compared. The results show that some simple compositing approaches work as well or better than more complex algorithms that are more difficult to implement.

Index Terms: Computing methodologies—Computer graphics—Rendering; Computing methodologies—Parallel computing methodologies—Parallel algorithms—Massively parallel algorithms

1 Introduction

Parallel rendering is critical for large-scale scientific visualization. Broadly speaking, there are two general approaches to render data in a distributed parallel system. The first approach is to distribute the geometry such that each process can completely render a subregion of the screen (known as *sort-first* rendering [12]). The second approach is to have each process render a full image with partial data and then combine (a.k.a. composite or reduce) these to a single, complete image (known as *sort-last* rendering [12]). It has long been shown that for large parallel jobs, sort-last provides much better scalability [16, 29]. The efficiency of sort-last parallel rendering with image compositing has been demonstrated in many systems [3, 13, 21, 22].

The efficiency of sort-last parallel rendering depends on the ability to composite the images generated by each process into a single image. One of the most well known algorithms, and one still used commonly to date, is binary swap [11]. Binary swap is popular because it is straightforward to implement and has good scaling behavior in terms of data transfer and number of iterations [21].

One natural problem with binary swap is that because it iteratively divides processors into two groups, and these groups need to be the same size, it only works well when the number of processes is a perfect power of 2. Multiple variations of binary swap have been independently introduced to overcome this limitation, but few have previously been directly compared with each other. This paper consolidates this research by implementing each of these binary swap algorithms in a common code base and running a direct comparison across them. This direct comparison draws some surprising results about the comparative performance of each.

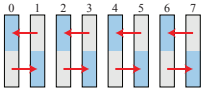
*e-mail: kmorel@sandia.gov

2 Compositing Algorithms

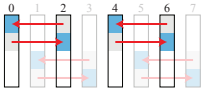
This section reviews the compositing algorithms used in this paper. Note that the compositing techniques outline here have been presented in some form in previous literature. However, this is the first time all these algorithms have been brought together and directly compared in a common code base.

2.1 Base Binary Swap

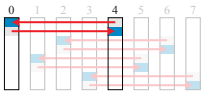
Binary swap is first introduced by Ma et al. [11]. It is a straightforward divide-and-conquer algorithm where at each phase each process exchanges pixels with another process. During the exchange, each process offloads half of its pixels, which are blended into the remaining images in the paired process.



Binary swap occurs on a series of iterations. In the first iteration, processes group in adjacent pairs. Pairs exchange halves of their image, as shown at right.



For the next iteration, all the process holding the same image group together, and then repeat the swapping operation. In the example at right with 8 processes, highlighted processes 0, 2, 4, and 6 group together and split their remaining images. (The remaining processes form their own group and similarly subdivide.)

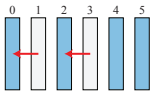


Binary swap iterations continue until each process has a unique portion of the image. In our example with 8 processes, binary swap completes after the third iteration. In general, binary swap takes $\log_2 p$ iterations, where p is the number of processes.

2.2 Binary Swap Variations for Odd Process Factors

Binary swap has many desirable qualities. The number of iterations grows slowly with respect to the number of processes. Also, each iteration halves the amount of work required per process. In total, each process will send, receive, and operate on no more than the number of pixels in a single image. However, a well known limitation is that binary swap only works when the number of processes is a perfect power of 2. If there are any odd factors, then at some iteration there will be an odd number of processes in a group, and they will not all be able to pair up properly. In response, there are numerous modifications to binary swap to handle odd factors.

2.2.1 Naive



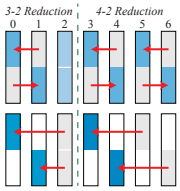
The most obvious way to manage odd factors is to find the largest power of 2 less than the number of processes and then offload images from any processes that do not fit in a group of that size. In the example at right, we have 6 processes. The largest power of 2 less than 6 is 4, so two processes send their images away and drop out of the communication. The remaining 4 processes do a normal binary swap as outlined in Section 2.1.

This method has been given multiple names such as *reduced* [31] and *folding* [13]. To avoid confusion from the other techniques (many of which can be thought of as "reducing" or "folding"), this paper refers to this technique as *naive*.

2.2.2 234 Composite

One of the big drawbacks of the naive technique is that it adds another iteration to the compositing. It also requires the transfer of full images, which potentially doubles the amount of data transferred or computed by each process. Nonaka et al. [18, 19] propose *234 composite* to roll the reduction into the first iteration of the binary swap algorithm.

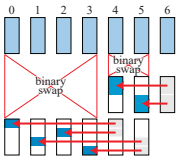
234 composite does this using a combination of *3-2 reduction* and *4-2 reduction* operations. These operations take a group of 3 and 4 processes, respectively, and reduce them to a group of 2. The resulting 2 processes each end with half an image much like a standard binary-swap pair.



In this example of 7 processes, there is 1 group for a 3-2 reduction and 1 group for a 4-2 reduction. After the operations are complete, there are 4 remaining processes in a state equivalent to have running the first step of binary swap. The remainder of the algorithm follows that of binary swap.

2.2.3 Telescoping

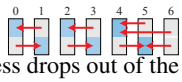
Moreland et al. [13] propose the *telescoping* algorithm. Like the naive algorithm, telescoping reduces the problem to groups that are a power of 2. However, rather than reduce the size of the groups before compositing starts, telescoping performs this reduction after compositing primarily finishes. It does this by breaking the processes into a series of groups that are the largest powers of 2 that it can make. It then runs binary swap on all such groups simultaneously. When compositing finishes, the smaller groups transfer their images to the larger groups.



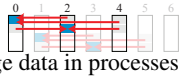
In the example at right, 7 processes are split into a group of 4, a group of 2, and a group of 1 (all perfect powers of 2). Binary swap is run concurrently and independently on each group. The group of size 1 splits its image and sends it to the size 2 group. Subsequently, the size 2 group splits its images and sends them to the size 4 group (thus collapsing the images in a telescoping manner). Since smaller groups have less work, we can expect the smaller groups to finish more quickly and have its data in transit while the larger groups finish.

2.2.4 Remainder

Rather than attempt to reduce the number of processes to a power of 2, another approach is to modify the binary swap algorithm to manage the situation where a group cannot be evenly divided. A simple approach, which we will refer to as *remainder*, simply rolls the image from any remainder when dividing the processes by employing a single 3-2 reduction to absorb the remainder. This is roughly equivalent to the reduction technique proposed by Rabenseifner and Träff [23] although, to the author's knowledge, this has not been directly applied to parallel rendering. (Also, the implementation used in this paper actually uses the overlap variant of 3-2 reduction as proposed by Nonaka [18].)



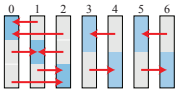
In this example of 7 processes to the right, the processes are divided by employing a 3-2 reduction on the last 3 processes. The last process drops out of the computation at this point.



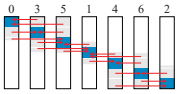
The next (final) iteration has groups of 3. These again are managed by a 3-2 reduction. The highlighted processes 0, 2, and 4 blend image data in processes 0 and 2 while dropping process 4.

2.2.5 2-3 Swap

All the previous algorithms for odd factors of processes require processes to go idle at some point during the computation. Yu et al. [31] describe *2-3 swap*, which can perform image compositing on any number of processes while balancing the computation among all processes at all iterations. It does so by dividing processes into groups of 2 or 3 and likewise dividing images by 2 and 3.



For example, the first step of a 7 process composite, shown at right, splits the processes in a group that does a 3-way swap and 2 more groups that do a 2-way swap. The details on when 2-3 swap does a 2-way vs. 3-way swap is complicated and requires precomputing a compositing tree. For details, see Yu's paper [31].



After the first step it is notable that the image partitions of processes do not line up as they do for binary swap. 2-3 swap manages this by interlacing the processes that are regrouped and redividing evenly. (Note the reordering of processes in this example.) The interlacing ensures that each process receives only pixels that it started with.

2.3 Alternatives to Binary Swap

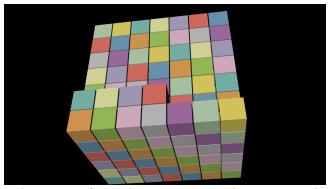
Finally, we quickly review some alternatives to binary swap that are viable but not considered in this paper.

Direct send [17] is a simple algorithm that assigns a piece of the image to each process, and then each process sends all its pieces directly to the responsible process. Direct send minimizes latency because all data reaches its destination in one step, but the number of messages required to do this grows quadratically with respect to the number of processes. Thus, the technique is unsuitable for large scales [13]. An often cited but seldom implemented variant of direct send is *Scheduled Linear Image Compositing* (SLIC) [25], which optimizes pixel assignment by assigning non-contiguous image pieces.

The *parallel pipeline* method [9] establishes processes in a linear chain where image data are passed down the chain. Parallel pipeline has a long latency as images must be passed down the entire pipeline, but it has been shown that the regular message passing can be used to optimize for physical hardware connections [28]. *Rotate tiling* [10] is a variant of parallel pipeline that establishes multiple pipelines to reduce latency.

Radix-k [20] combines binary swap and direct send by allowing each iteration of binary swap to group into any size rather than just 2 and uses a direct send in that subgroup to swap data around. Because it can divide processes in any way, it is not limited to powers of 2 like binary swap. However, because the performance of radix-k can vary based on factors chosen [5, 13], it would be interesting to explore ways to adjust available factors. However, this is beyond the scope of this paper and left for future work.

3 Experiments



For these experiments, versions of binary swap and the variants described in Section 2.2 were implemented in a common code base. All the algorithms use a common infrastructure for rendering and image data structures. This test infrastructure renders a simple scene where each process renders an opaque box like that shown here.

The experiments were run on the Sky Bridge cluster at Sandia National Laboratories [24]. Sky Bridge is a water cooled Cray capacity cluster with 1,848 nodes (although at most 512 nodes were used at any one time during these experiments). Each node contains 2 2.6 GHz Intel Xeon E5-2670 processors, each with 8 cores (16 cores total). The nodes are connected with an Infiniband interconnect. The experiments were run in "virtual node" mode where each core ran a separate MPI process (except where specified in Section 3.3).

Each experiment has 20 trials (rendered frames). Each trial is performed from random camera rotations around the geometry (al-

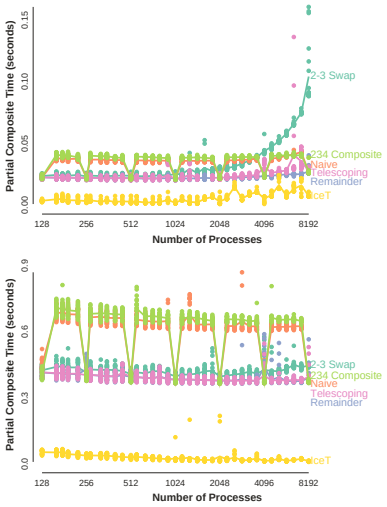
though these random locations are consistent across experiments by using the same pseudorandom number generator seed). Rendering times for two different image sizes are reported: HDTV (1920×1080) and 8K UHD (7680×4320).

The compositing engaged active pixel encoding for compression (except where specified in Section 3.2). The times reported here are specifically the time to divide the image and blend pixels. This leaves the final composited image divided across many MPI processes. The time to gather the image pieces is not reported except where discussed in Section 3.4.1. The time to map the geometry to pixels is not reported as this time is independent from the compositing algorithm.

3.1 Algorithm Comparison and Scaling

The first experiment performs a scaling study of the behavior of the binary swap algorithm with the 5 variations discussed in Section 2.2. The tests include runs on 128 processes (8 real nodes) up to 8192 processes (512 real nodes). It is impractical to run an experiment on every possible number of processes. Instead, jobs are run on every $2^{i/6}$ processes, rounded to the nearest integer. This hits all jobs sizes that are a perfect power of 2 and 5 in between.

The results of these experiments are shown at right. The top plot shows results for HDTV images whereas the bottom plot shows results for 8K UHD images. Each trial (20 per experiment) is drawn as a dot. Experiments with little deviation in the trials have their dots drawn on top of each other. A trend line is drawn for the average values. The plots are labeled as "Partial Composite Time" because these times include only splitting the images and blending pixels. The fully composited image is left in pieces scattered across the MPI processes. Section 3.4.1 discusses the collection of these pieces.

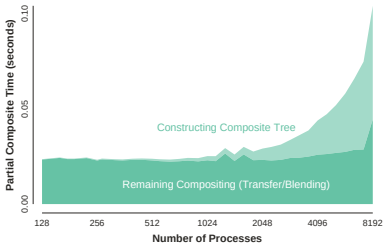


A first observation about the data is to note that the naive algorithm pays a significant but consistent penalty for running on a number of processes that is not a power of 2. This is consistent with the findings of Yu et al. [31].

A second observation is that the 234 composite algorithm follows a similar pattern as the naive algorithm. Although Nonaka et al. [18] suggest that the combination of reduction into the first step of binary swap will make 234 composite faster than naive, these measurements do not show an improvement.

A third observation is that the 2-3 swap algorithm behaves well up to 1024 processes (which is as large as was measured by Yu et al. [31]), but the performance starts to deteriorate on larger sizes, particularly for the HDTV images. This appears to be caused by the time taken to build the compositing tree, which is complex for 2-3 swap and has to take into account all the processes.

The plot at right demonstrates the fraction of time spent in building the 2-3 compositing tree. The top lighter shade shows the time spent building the compositing tree whereas the bottom darker shade shows the remainder of the algorithm (transferring data, blending colors, etc.).



A fourth observation is that both the telescoping and remainder versions of binary swap perform very well throughout the entirety of the experiments. This is somewhat counterintuitive for the remainder algorithm, which, much like the poor performing naive, and 234 composite algorithms, lets processes go idle. However, unlike the naive and 234 composite algorithms, remainder delays letting processes go idle until as late as possible.

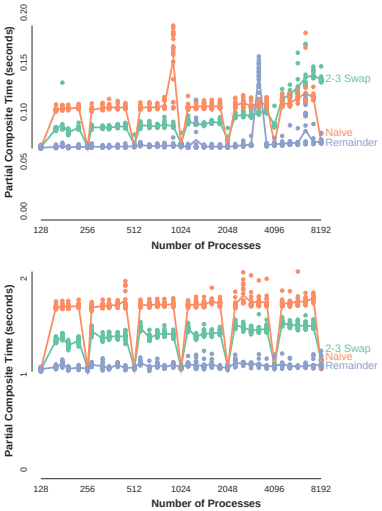
A fifth observation is that all of the binary swap algorithms implemented for this paper perform less well than the IceT software. IceT uses a combination of radix-k and telescoping, but the real performance gain is in its high-speed blending and management of memory to reduce allocation and messages. In contrast, this paper's implementation sacrifices some efficiency for readability of code and ease of implementation to facilitate comparisons like those in this paper.

3.2 No Image Compression

The compositing reported in Section 3.1 compresses images during compositing using run lengths of active and inactive pixels (sometimes called *active pixel encoding*). Any practical system should employ active pixel encoding or something like it as previous work has shown dramatic improvements in compositing time [1,15,26,30]. However, active pixel encoding can change the performance behavior of the compositing. The overall time will increase and decrease depending on the effectiveness of the compression. Image compression can also add load imbalance.

To verify that active pixel encoding is not artificially skewing the previous results, a second set of runs replicates the experiments with image compression turned off. This set of experiments only includes the naive, 2-3 swap, and remainder algorithms. 234 composite and telescoping were not run to save time because their performance is similar to naive and remainder, respectively. IceT was not run because it always compresses the data.

The results are shown at right. The top plot shows results for HDTV images whereas the bottom plot shows results for 8K UHD images. As before, the measurements for all trials are shown as dots drawn on an average trend line.

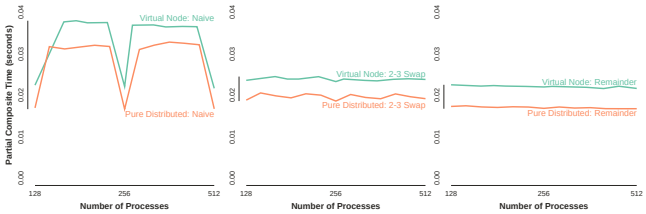


A first observation is that the behavior of the three algorithms is roughly consistent, proportionally speaking, with the behavior using active pixel encoding with the exception of 2-3 swap. Both the naive and 2-3 swap algorithms pay a large penalty when not using a perfect power of 2 number of processes (although 2-3 swap has about half the penalty). The remainder algorithm's performance is consistent and is the best performer in all cases.

A second observation is that without compression, the compositing takes about 2.5 to 4 times as long, which verifies the statements about the utility of active pixel encoding during compositing.

3.3 Virtual Node vs. Pure Distributed

To maximize the scaling in the aforementioned experiments, runs were performed in a *virtual node* mode where each core on a node has its own MPI process. In essence, each core is treated as a separate node even though they technically share resources. One consequence of the virtual node setup is that the network behavior is expected to be more heterogeneous. Data transfers between two virtual nodes on the same physical node are expected to be much faster than transfers between virtual nodes on two different physical nodes. To ensure that the conclusions we draw are not invalid for different network configurations, the experiments are also run in a *pure distributed* mode where only a single MPI process is run on each node (and consequently only one core is used).



The plots above compare the performance of compositing when run on the same number of nodes in virtual node mode and pure distributed mode when using the naive, 2-3 swap, and remainder algorithms (in the left, center, and right plots, respectively). The average times for the trials of each experiment are given. A similar run for 8K UHD images was also run and gave similar results, but it is not displayed here for lack of space.

An interesting observation is that the pure distributed mode runs are consistently faster than the virtual node mode. This is counterintuitive as many of the connections in virtual node mode are faster than those in pure distributed mode. Most likely the slowdown is due to the cores in virtual node mode sharing the same network interface controller (NIC) and having to divide the bandwidth accordingly. This measurement suggests that overall rendering throughput will be maximized by leveraging shared-memory local rendering, of which there are many choices [2, 6, 7, 14, 27], to reduce the number of MPI processes on each physical node.

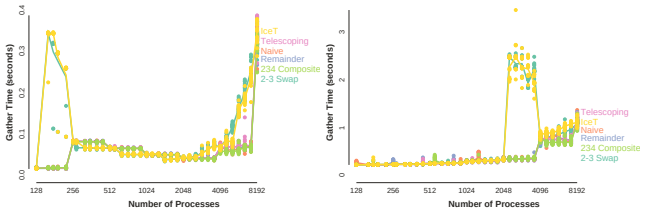
Apart from the consistent overhead of running in virtual node mode, we can see that the characteristics of the behavior are similar for both modes. Thus, we can conclude that the findings drawn from our virtual node scaling can be applied to other job launching patterns.

3.4 Aberrant Readings

The results presented in the previous sections have been down-selected for clarity. For completeness, this section documents some of the more unusual or inconsistent measurements taken.

3.4.1 Inconsistent Gather Times

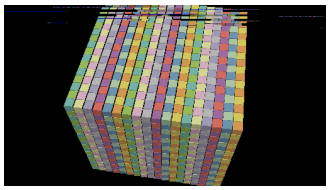
The previously reported compositing times include the process up to the point where the resulting image is divided among multiple processes. However, in any practical system, the images must be gathered together for display. It is known that this gathering can take a significant portion of the compositing time [8, 13, 19, 23].



The plots above show the time spent while using MPI_Gather and MPI_Gatherv for HDTV images (left) and 8K UHD images (right). The measurements for all trials are shown as dots drawn on an average trend line. The cluster used for these experiments is giving wildly different performance for the gather operations that seems to have little to do with the compositing algorithms analyzed in this paper, which is why these times are not incorporated into the other times given.

These times suggest that a different mechanism for collection is needed. Nonaka et al. [19] has success by adding padding and process reordering to replace MPI_Gatherv with MPI_Gather. However, in this case it might be better to not rely on MPI-provided collectives at all.

3.4.2 Errors in 2-3 Swap



Several checks were run during the experiments to ensure the correctness of the compositing algorithms. For most of the runs there were no problems. However, during a few of the 2-3 swap runs bad pixels were detected at the top of the images like that shown here. It appears to be an indexing problem caused by a race condition that cannot be easily replicated.

Although some of the images are incorrect, it seems unlikely that the errors are causing a significant impact in the recorded timings. Furthermore, since 2-3 swap is not one of the better performing algorithms, there seems little reason to suffer the aggravation to fix the problem.

3.4.3 System Slowdowns

During the experiments it was noticed that some of the runs gave anomalously long run times. These appear to be intermittent issues with the testing cluster. Several of these experiments were re-run and found that the extra run time was in fact anomalous. However, some of the less egregious anomalies are left in the results (such as with the spikes seen in the plots of Section 3.2).

4 Conclusions

This paper presents a comprehensive comparative study of many parallel image compositing algorithms based on binary swap. Never before have all these algorithms been directly compared, and the results challenge some previously accepted notions.

The clear and surprise winner of the comparison is the remainder version of binary swap. Remainder is simple to implement (the implementation used in these experiments has less lines of code than even the naive algorithm) and is consistently a top performer. In contrast, the implementation of 234 composite is over twice as long as remainder's (according to cloc [4]) but seldom performs better than naive. The implementation for 2-3 swap is almost 4 times as long as remainder, and, in the authors subjective experience, much more difficult to implement.

Also surprisingly, the simple but effective remainder technique does not seem to be used at all in previous literature of parallel rendering. Ultimately, the parallel rendering research community should stop attempting to find complex solutions to manage undesirable factors but rather use a simple technique to absorb the remaining images when using factors that do not divide processes evenly.

One of the biggest contributions of this work is a collection of sort-last image compositing algorithms with a consistent implementation and software framework, which makes direct comparisons like those provided in this paper possible. To encourage and enable other researchers to make similar studies (as well as verify or dispute the results of this paper) all the software used in this study is being made publicly available. The software, as well as all the raw timing logs collected during the study, are posted at

`http://www.kennethmoreland.com/scalable-rendering/#LDAV2018`

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number 14-017566.

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND 2018-9004 C

REFERENCES

[1] J. Ahrens and J. Painter. Efficient sort-last rendering using compression-based image compositing. In *Second Eurographics Workshop on Parallel Graphics and Visualization*, September 1998.

[2] B. Cherniak. OpenSWR: A scalable high-performance software rasterizer for scivis. Presentation, *Intel HPC Developer Conference*, Nov. 1015.

[3] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. H. Weber, and E. W. Bethel. Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications*, 30(3):22–31, May/June 2010. doi: 10.1109/MCG.2010.51

[4] A. Danial. cloc: Count lines of code. `https://github.com/AlDanial/cloc`.

[5] W. Kendall, T. Peterka, J. Huang, H.-W. Shen, and R. Ross. Accelerating and benchmarking radix-k image compositing at large scale. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, May 2010.

[6] A. Knoll, I. Wald, P. Navratil, A. Bowen, K. Reda, M. E. Papka, and K. Gaither. RBF volume ray casting on multicore and manycore CPUs. *Computer Graphics Forum (Proceedings of EuroVis)*, 33(3):71–80, June 2014. doi: 10.1111/cgf.12363

[7] M. Larsen, J. S. Meredith, P. A. Navratil, and H. Childs. Ray tracing within a data parallel framework. In *IEEE Pacific Visualization Symposium (PacificVis)*, pp. 279–286, April 2015. doi: 10.1109/PACIFICVIS.2015.7156388

[8] M. Larsen, K. Moreland, C. Johnson, and H. Childs. Optimizing multi-image sort-last parallel rendering. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, Oct. 2016. doi: 10.1109/LDAV.2016.7874308

[9] T.-Y. Lee, C. Raghavendra, and J. B. Nicholas. Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217, Sept. 1996. doi: 10.1109/2945.537304

[10] C.-F. Lin, S.-K. Liao, Y.-C. Chung, and D.-L. Yang. A rotate-tiling image compositing method for sort-last parallel volume rendering systems on distributed memory multicomputers. *Journal of Information Science and Engineering*, 20(4):643–664, 2004.

[11] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. A data distributed, parallel algorithm for ray-traced volume rendering. In *Proceedings of the 1993 Symposium on Parallel Rendering*, pp. 15–22, 1993. doi: 10.1145/166181.166183

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

[13] K. Moreland, W. Kendall, T. Peterka, and J. Huang. An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, November 2011. doi: 10.1145/2063384.2063417

[14] K. Moreland, C. Sewell, W. Usher, L.-T. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci. VTK-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications*, 36(3):48–58, May/June 2016. doi: 10.1109/MCG.2016.48

[15] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 85–92, October 2001.

[16] C. Mueller. The sort-first rendering architecture for high-performance graphics. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pp. 75–84, 1995. doi: 10.1145/199404.199417

[17] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Proceedings of the 1993 Symposium on Parallel Rendering*, pp. 97–104, 1993. doi: 10.1145/166181.166196

[18] J. Nonaka, K. Ono, and M. Fujita. 234 scheduling of 3-2 and 2-1 eliminations for parallel image compositing using non-power-of-two number of processes. In *International Conference on High Performance Computing & Simulation (HPCS)*, July 2015. doi: 10.1109/HPCSim.2015.7237071

[19] J. Nonaka, K. Ono, and M. Fujita. 234Compositor: A flexible parallel image compositing framework for massively parallel visualization environments. *Future Generation Computer Systems*, 82:647–655, 2018. doi: 10.1016/j.future.2017.02.011

[20] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, November 2009. doi: 10.1145/1654059.1654064

[21] T. Peterka and K.-L. Ma. *High Performance Visualization: Enabling Extreme Scale Insight*, chap. Parallel Image Compositing Methods, pp. 71–89. CRC Press, 2013.

[22] T. Peterka, H. Yu, R. Ross, K.-L. Ma, and R. Latham. End-to-end study of parallel volume rendering on the IBM Blue Gene/P. In *Proceedings of ICPP '09*, pp. 566–573, September 2009. doi: 10.1109/ICPP.2009.27

[23] R. Rabenseifner and J. L. Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)*, vol. 3241 of *Lecture Notes in Computer Science*, pp. 36–46, 2004. doi: 10.1007/978-3-540-30218-6_13

[24] N. Singer. Sandia turns on sky bridge supercomputer. *Sandia Lab News*, December 12 2014. `http://www.sandia.gov/news/publications/labnews/archive/14-12-12.html#3`.

[25] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett. SLIC: Scheduled linear image compositing for parallel volume rendering. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG 2003)*, pp. 33–40, October 2003.

[26] A. Takeuchi, F. Ino, and K. Hagihara. An improvement on binary-swap compositing for sort-last parallel rendering. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, pp. 996–1002, 2003. doi: 10.1145/952532.952728

[27] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: A kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG)*, 33(4), July 2014. doi: 10.1145/2601097.2601199

[28] Q. Wu, J. Gao, Z. Chen, and M. Zhu. Pipelining parallel image compositing and delivery for efficient remote visualization. In *Journal of Parallel and Distributed Computing*, pp. 230–238, March 2009. doi: 10.1016/j.jpdc.2008.11.004

[29] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland. Scalable rendering on PC clusters. *IEEE Computer Graphics and Applications*, 21(4):62–70, July/August 2001.

[30] D.-L. Yang, J.-C. Yu, and Y.-C. Chung. Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. In *1999 International Conference on Parallel Processing*, pp. 200–207, 1999. doi: 10.1109/ICPP.1999.797405

[31] H. Yu, C. Wang, and K.-L. Ma. Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008. doi: 10.1109/SC.2008.5219060