SAND2011-2482C

Sort-Last Smackdown!

Kenneth Moreland*
Sandia National Laboratories

Wesley Kendall†
University of Tennessee, Knoxville

Tom Peterka‡
Argonne National Laboratory

Jian Huang§
University of Tennessee, Knoxville

ABSTRACT

The only proven method for performing distributed-memory parallel rendering at large scales, tens of thousands of nodes, is a class of algorithms called sort last. The fundamental operation of sort-last parallel rendering is an image composite, which combines a collection of images generated independently on each node into a single blended image. Over the years numerous image compositing algorithms have been proposed as well as several enhancements and rendering modes to these core algorithms. However, the testing of these image compositing algorithms has been with an arbitrary set of enhancements, if any are applied at all. In this paper we take a leading production-quality image-compositing framework, IceT, and use it as a testing framework for the leading image compositing algorithms of today. The compositing enhancements provided by IceT, including some introduced in this paper, are employed in our measurements. IceT also provides different compositing mechanisms for different rendering environments such as opaque surface versus volume rendering and fixed point versus floating point color representations. These variations are also considered in our analysis. To understand the behavior of these algorithms at vary large scale, we run tests on up to 64K cores of the Intrepid BlueGene/P at Argonne National Laboratories.

Index Terms: I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing

1 INTRODUCTION

The demands of parallel rendering continue to grow as visualization is applied to ever larger scientific data. Early efforts have satisfied the need of parallel rendering on specialized visualization clusters containing hundreds of nodes. Because of recent constraints in building specialized visualization clusters [4], recent research focuses on performing visualization directly on the same supercomputing architectures driving simulation. Whereas previous research considers rendering on the order of hundreds of processes, recent efforts use over ten thousand processes [5]. Furthermore, in an attempted to get around bottlenecks introduced by file I/O, it is now more commonplace to have visualization run *in-situ* with simulation [8,11,27,30], meaning parallel rendering will soon run on over a hundred thousand processes.

These increased demands on parallel rendering have spawned a resurgence in parallel rendering research. Recent studies investigate the scaling of parallel rendering algorithms [19], the creation of new image compositing algorithms [18, 31], and new compositing enhancements [7]. Although each of these studies improve the state of the art in parallel rendering, all involve locally built algorithm implementations that contain some arbitrary subset of enhancements and that may or may not be publicly available. The consequence is that it is difficult to repeat the experiments, to compare the results with each other, and to apply the algorithms to production software.

The intention of this work is to bridge the gap between independent parallel rendering algorithm development and practical application by bringing together multiple algorithms and enhancements together in a production quality parallel rendering library. More specifically, this paper provides the following.

- An introduction to a sort-last rendering framework named *IceT* that is general purpose, production quality, and fully optimized.
- Describe an optimization to the compositing order for radix-k and direct send that minimizes the number of copies of non-overlapping pixels.
- A new method of *image interlacing* that requires no additional image copying to reconstruct the final image.
- The *telescoping* algorithm, which can be applied to an existing image compositing algorithm that works best on powers of two, such as binary swap, to run efficiently on any number of processes.
- An investigation of the collection operation required at the end of compositing and adjustments to the compositing algorithms to minimize it.
- A comparison of the ever popular binary-swap algorithm with the newer radix-k algorithm on leadership-class high-performance computers. These tests are performed with every enhancement one should expect in production quality parallel rendering as well as with a variety of rendering modes that can be encountered in production software.

2 PREVIOUS WORK

Although many aspects of parallel rendering have changed since the sorting classification of parallel rendering algorithms was introduced [12], these classifications are still used today because they accurately characterize and predict the scaling performance of these algorithms. When rendering on a hundred or more distributed nodes, the most efficient class of algorithm is sort-last because it scales extremely well with respect to the number of processes and size of the geometry being rendered and because the main contributing factor to its overhead, the size of the image being rendered, is fixed by the display that we are using [28].

The main characteristics of sort-last parallel rendering is that geometry is statically partitioned; processes each independently render images using only their local partition, and these images are *composited* together by blending or comparing pixels. Consequently, it is the behavior of this compositing operation that determines the overall efficiency of sort-last parallel rendering.

*e-mail: kmorel@sandia.gov
†e-mail: kendall@cs.utk.edu
‡e-mail: tpeterka@mcs.anl.gov
§e-mail: huangj@eecs.utk.edu

2.1 Basic Parallel Compositing Algorithms

Over the years researchers have designed several variations of the image compositing algorithm. One of the oldest and simplest algorithms that is still in wide use is direct send [15, 16]. Direct send assigns each process a unique partition of the image to be rendered. After the local geometry is rendered, each process sends each pixel fragment directly to the process responsible for compositing it. Each process then collects pixel fragments from all other processes and combines them to form its partition of the image. Although direct send is efficient in the amount of data it transfers, the number of messages it generates grows quadratically with the number of processes. Thus, for large numbers of processes the network can get overwhelmed by many small messages.

One of the most popular image compositing algorithms is binary swap [9, 10]. Binary swap executes in rounds. During a round, each process pairs up with another process, the image is split in half, the paired processes exchange image halves, and each process composites the pixel fragments for the half of the image it received. After $\log_2 n$ rounds, where n is the number of processes, each process holds a unique fully-composited partition of the image. Binary swap uses fewer messages than direct send: $n \log_2 n$ total messages with only n messages at a time (assuming minimal overlap between rounds). Because the bisection bandwidth of a cluster interconnect generally grows with respect to the number of nodes, the bandwidth requirements on the network remain relatively fixed.

One of the problems with sort last is that it requires a number of processes equal to a power of two. The simplest solution in dealing with other process counts is to *fold* the images into a group of the correct size. Create the largest group possible with a power of two, and then send the image data from those processes outside the group to a process inside the group. Those processes outside the group sit idle while those inside the group continue on to composite the image. This approach has inefficiencies because processes have to sit idle during most of the computation. The 2-3 swap algorithm [31] takes a different approach. It relaxes binary swap such that processes can be grouped into pairs of two (like binary swap) or sets of three (unlike binary swap). Using these groups of two or three, 2-3 swap can decompose any number of processes into groups, and in this way all processes can take part in compositing at all times.

Radix-k [18] is a combination of binary swap and direct send. Radix-k first factors the number of processes into a series of what are called k values, which need not be prime. In a sequence of rounds, one per k value, radix-k partitions the processes into groups of size k and performs a direct send within each group. The next round recurses into processes with the same partition until all k values are used and each process has a unique partition. When it has one round with a k value equal to the number of processes, radix-k is equivalent to direct send. When it has $\log_2 n$ rounds with all k values equal to two, radix-k is equivalent to binary swap.

Radix-k improves on binary swap by overlapping data transfers with computation. When receiving data from multiple processes, which happens whenever k is greater than two, radix-k can begin compositing pixels as soon as the first message is received while other messages are still in transit. Yet radix-k retains binary swap's ability to limit the total number of messages sent. Radix-k is also able to handle process groups that are not powers of two because the k value for each round can be any factor. That said, if the number of processes factors into large prime numbers, the performance can degrade to that of direct send.

2.2 Compositing Enhancements

A naïve implementation of sort-last image compositing will consider every pixel fragment from every process participating. However, in almost all practical use cases the geometry is, or at least can be, partitioned spatially. When each process has geometry in a confined spatial region, there is a great deal of empty space in the original rendered images. A pragmatic image compositing algorithm takes advantage of these empty spaces in two ways. First, the pixels in these empty regions will be removed from communication, thus making better use of available network bandwidth. Second, the empty regions are not considered in the composite operation, which reduces the overall computation performed.

There are two standard approaches for tracking the "active" pixels (those that have been rendered to) and "inactive" pixels (those over empty regions). The first method is to track bounding boxes around geometry. Typically, a box around the geometry in each process is projected to screen space to define the region of pixels that likely have geometry rendered to them. (The boxes are often expanded to axis aligned bounding boxes to simplify management.) Only the pixels in this region are read, transferred, and composited. Ma et al. [10] show that in the common case tracking bounding boxes reduces the total number of pixels transmitted from $O(np)$ to $O(n^{1/3}p)$, where n and p are the number of processes and pixels, respectively.

[IF THE PAPER IS LONG WE CAN SHORTEN IT BY REMOVING THE DISCUSSION ON ACTIVE PIXELS.]

The second approach for tracking active and inactive pixels is to use run-length encoding [2]. A generic run-length encoder will look for run lengths of any repeated value. However, when compositing images the active pixels tend to have run lengths of 1, so run-length encoding can actually hurt in these regions. Thus, a better approach is to use *active pixel encoding*, which classifies the pixels as either active or inactive and provide run lengths for continuous regions of any active pixels. Moreland et al. [14] show that this encoding is both effective and never adds to the data size even in the worst pathological cases. Active pixel encoding improves on region boxes by tightly identifying active and inactive pixels. There is a greater overhead incurred by searching through the image for inactive pixels, but this overhead is mitigated by considering the bounding boxes during the initial encoding [29].

Although active pixel encoding almost always improves the performance of compositing, it does introduce an issue of load balancing. As images are partitioned, some regions will have more active pixels than others. By balancing the active pixels assigned to regions, the parallel compositing becomes better load balanced and performance can improve even further.

The most straightforward way of balancing active pixels is to *interlace* the images [12, 26].[1] An image is interlaced by rearranging regions of pixels, commonly scanlines, in a different order. This reordering of pixels is designed such that when the images are later partitioned, each partition gets pixels from all over the images. Consequently, regions with many active pixels are distributed to all the partitions.

The SLIC algorithm [25] integrates the direct-send algorithm with inactive pixel skipping and image interlacing. It finds areas of geometry overlap by projecting bounding boxes to screen space. SLIC then breaks scanlines by areas of overlap and uses a simple hashing function to assign these scanline fragments to processes. The hash function provides load balancing and the tracking of overlap limits the total number of messages to $O(n^{4/3})$, where n is the number of processes, which is better than the original direct send but worse than binary swap or radix-k.

One problem with image interlacing is that the pixels in the fully composited region must be rearranged once again into the correct order. This added overhead can remove the performance gains of the load balancing. To get around this problem, Kendall et al. [7] propose a method in which the partitioning for the radix-k algorithm is adjusted so that each partition has the same amount of ac-

[1] Other literature uses the term interleave, but we feel the word interlace is more descriptive.

tive pixels. Although Kendall's algorithm improves load balancing, it also adds overhead in readjusting partitions for consistency amongst all processes.

Most sort last algorithms rely on a static partitioning of the data to produce empty regions of the data. Hybrid algorithms [22] also use dynamic partitioning of the data to collect geometry by screen region based on the current projection. Hybrid algorithms reduce the compositing time at the expense of redistributing geometry, which means the effectiveness of the technique is dependent on the amount of geometry being rendered. Other approaches propose ensuring empty space regions using static partitions with replication [21].

3 Testing Environment

The Image Composition Engine for Tiles (IceT) is a high-performance sort-last parallel rendering library [13]. Although originally created to capture sort-last rendering algorithms for tiled displays [14], IceT also works effectively for smaller single image displays.

IceT contains several image compositing algorithms, and its internal architecture makes it straightforward to add new algorithms. It also optimizes the compositing process by tracking the projection of geometry and compressing images through run-length encoding. IceT also supports multiple rendering modes allowing both color blending for volume rendering and z-buffer comparisons for opaque geometries.

IceT is used in multiple production products like ParaView [24] and VisIt [1] and has been used to achieve record-breaking rendering rates [6]. As such, IceT is an excellent code base for creating, testing, and comparing image compositing algorithms. It already contains routines for efficiently capturing, compressing, and compositing images. It also contains efficient existing algorithms to which to compare new ones. Furthermore, any optimizations or new algorithms added to IceT can be applied to existing production software.

The experiments we run for this paper are encapsulated in IceT's testing suite under the SimpleTiming test. This test evenly partitions volume-wise a cube of space amongst processes. Each process renders a hexahedron filling the space it is assigned as a proxy geometry for the rendering. We use this proxy rendering to simplify compiling and porting, which should be particularly useful for anyone wishing to repeat these experiments. In any case, the rendering time is discarded as we are interested only in the compositing overhead. Figure 1 shows an example of images rendered by SimpleTiming. For each SimpleTiming test we render 101 frames at pseudorandom viewpoints, always using the same seed for consistency between experiments. The time for the first frame is thrown out of any average because it contains added overhead of memory allocations not included in subsequent frames.

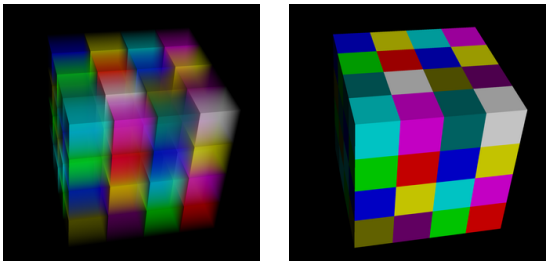


Figure 1: Examples of images rendered in our experiments.

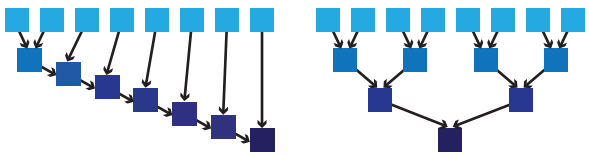
Most of the experiments reported in this paper were run on Argonne National Laboratory's Intrepid Blue Gene/P computer [23]. Intrepid comprises a total of 40,960 nodes, each containing four cores. Each experiment was run in one of two modes. The first mode, *Shared Memory Processors* (SMP), runs a single MPI process on each Intrepid node. The intention of the mode is to run multiple threads to use all four cores, but in our experiments we run a single thread using only one core. The second mode, *Virtual Node* (VN), runs four MPI processes on each Intrepid node. It treats each core on the node as a distributed memory processes even though it is possible to share memory. Data transfers amongst the processes within a single node still require explicit MPI memory passing although the underlying MPI layer bypasses the network infrastructure in this case. We consider both running modes because both are commonly used today and each has differing demands on the underlying subsystems.

Supplemental experiments were run on Sandia National Laboratories' Red Sky computer. Red Sky comprises a total of 2,279 nodes, each containing dual 2.93 GHz quad core Nehalem X5570 processors (8 cores in all). Nodes are connected by a QDR InfiniBand network. All experiments on Red Sky use all eight cores of each node (the equivalent of VN mode on Intrepid). Red Sky has faster processors than Intrepid but a slower interconnect, which changes the dynamics and optimizations of parallel algorithms run on them.

4 Compositing Order

During our integration of radix-k into IceT, we discovered that the compositing order of incoming images could make a significant performance difference. In our initial implementation of radix-k, we got dramatically different results than those reported by Kendall et al. [7]. Rather than getting improved performance with radix-k, we saw worse performance. Deeper inspection revealed that although our radix-k implementation was properly overlapping communication with compositing computations, the computations took longer with larger values of k.

This increase in compositing time is caused by a change in the order images are composited together. The order in which images are composited within a round is not specified in the original radix-k algorithm; however, generally images are composited in the order they are received and accumulated in a single buffer, as demonstrated in Figure 2a. The issue is that composited images grow with respect to the non-overlapping pixels in each image. Pixels that do not overlap are simply copied to the output. In the example of compositing images for a radix-k round of $k = 8$ given in Figure 2a, non-overlapping pixels in the leftmost images are copied up to seven times before the final image is produced. In contrast, binary swap performs the equivalent composites in a tree-like order as shown in Figure 2b, and no pixel needs to be copied more than three times.



(a) Accumulative Order (b) Tree Order

Figure 2: Two possible orders for compositing eight images. Boxes represent images and arrows indicate how two images are composited together to form a third image.

Given this observation, we made two independent improvements to the radix-k algorithm. The first improvement speeds-up the compositing computation. Specifically, run lengths of non-overlapping pixels to be copied are collected and copied in blocks rather than independently cast and copy each pixel value, as was done before.

The second improvement constrains the compositing to follow the specific tree composite order. That is, rather than composite an incoming image whenever possible, force the compositing to happen in an order like that in Figure 2b. This constraint may cause radix-k to wait longer for incoming images, but the overhead is more than compensated by the improved rendering performance.

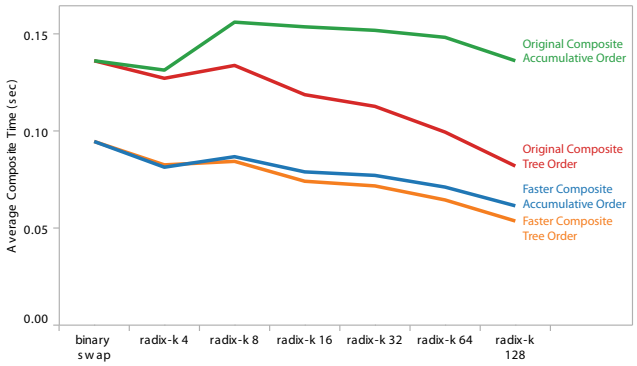


Figure 3: Comparative performance of radix-k with improved compositing computation and changing the order of compositing. All runs were performed on 256 nodes of Intrepid in SMP mode generating images with 2048×2048 pixels and transparent blending.

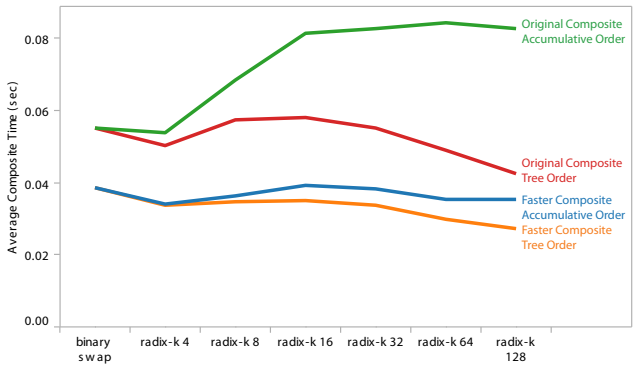


Figure 4: Comparative performance of radix-k with improved compositing computation and changing the order of compositing. All runs were performed on 2048 nodes of Intrepid in SMP mode generating images with 2048×2048 pixels and transparent blending.

Results of these performance improvements are independently shown in Figures 3 and 4. The improvements in compositing computation lower the overall compositing time and reduce the pixel-copying overhead of larger k values. The change in composite ordering removes the extra overhead of pixel-copying and maximizes the performance of larger k values.

5 Minimal-Copy Image Interlacing

The major problem encountered with sparse parallel image compositing is that it introduces load inbalance, which limits the improvements attained from compressing images. A straightforward approach to balance the compositing work is to interlace the images [12, 26]. Interlacing basically shuffles the pixels in the image such that any region of the image with more compositing work is divided and distributed amongst the processes.

Interlacing incurs overhead in two places during parallel compositing. The first overhead is the shuffling of pixels before any compositing or message transfers take place. This overhead tends to be low because it occurs when images are their most sparse and the work is distributed amongst all the processes. The second overhead is the reshuffling after compositing completes to restore the proper order of the pixels. This second suffling is substantial as it happens when images are at their most full and the maximum amount of pixels must be copied. Furthermore, because pixels must be shuffled over the entire image, this reshuffling must happen after image data is collected on a single process, which means the reshuffling itself is poorly load balanced.

Here we provide an interlacing algorithm that completely avoids the needs for this second reshuffling. The algorithm is based on the simple observation that at the completion of either binary swap or radix-k, each process contains a partition of the final image. If we arrange our initial shuffling such that each of the partitions remain a contiguous set of pixels, then we do not need the final reshuffling at all.

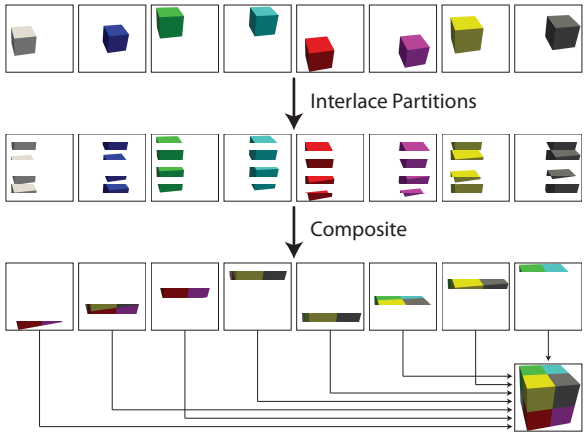
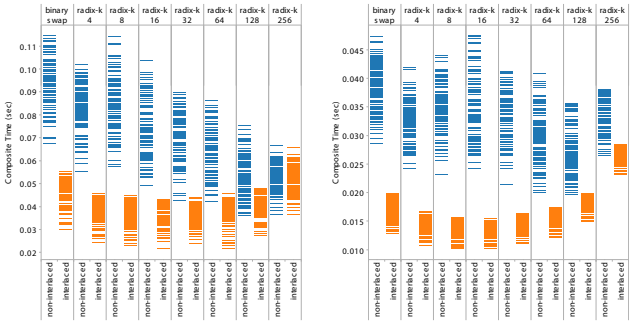


Figure 5: Pixel shuffling in minimal-copy image interlacing.

Our minimal-copy image interlacing is demonstrated in Figure 5. Rather than picking arbitrary partitions, such as scan lines, to interlace, our interlacing uses the partitions that binary swap or radix-k will create anyway. The partitions are reordered by reversing the bits of their indices, which creates a van der Corpt sequence to maximize the distance between adjacent blocks. The image with the interlaced block is composited as normal. Each resulting image partition is already intact, it is only the implicit offsets that need to be adjusted.



(a) 256 processes. (b) 2048 processes.

Figure 6: Comparative performance of radix-k with and without image interlacing. The composite time for each frame is represented as mark in the plot. All runs were on Intrepid in SMP mode.

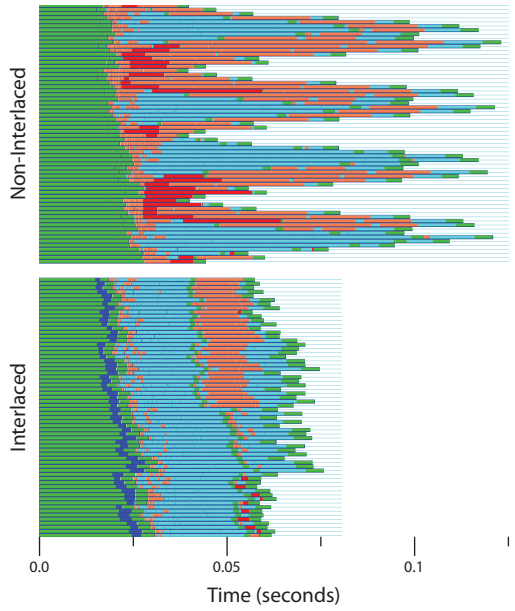


Figure 7: Jumpshot logs demonstrating the effect of minimal-copy image interlacing. The cyan color denotes the blending operation whereas the salmon and red indicate communication and waiting on messages. Green represents the encoding and splitting of images, and the dark blue is the time spent interlacing.

Figure 6 compares the compositing performance with and without image interlacing. Image interlacing both reduces the overall time to composite and reduces the variability between different viewports. Figure 7 compares Jumpshot logs of compositing with and without image interlacing on 64 processes using radix-k with $k = 16$. With image interlacing, the compositing work is better balanced and less time is spent waiting for messages.

6 Telescoping Compositing

A well known problem with binary swap is its inability to deal well with processor counts that are a power of two. A simple and common technique to apply binary swap to an arbitrary count of processes is folding. Folding finds the largest count of processes that is a power of two and forms a group of that size. Any process outside of this group (of which there are always fewer than inside the group) sends its entire image to one process inside the group where it is blended with the local image. The processes inside the power-of-two group composite normally while the remaining processes remain idle. Yu et al. [31] show that there is a performance penalty for folding.

2-3 swap [31] augments binary swap by grouping processes in either twos or threes in each round rather than exclusively twos. Because groups of two and three divide images differently, intermediate steps between rounds repartition images and send data to ensure that all processes have image partitions on the same boundaries. Radix-k also supports process counts that are not a power of two by forming groups that are any factor of the process count. This approach works well if the process count has small factors, but is inefficient for counts with large factors (as demonstrated by the results given in Figure 9, which concur with observations of Peterka et al. [19] for direct send).

We provide an alternative algorithm called telescoping for supporting arbitrary process counts. Our telescoping algorithm provides a simpler indexing and partitioning scheme than 2-3 swap and can be applied to most base parallel compositing algorithms. We demonstrate telescoping to both binary swap and radix-k. Telescoping works similar to folding except instead of sending images before compositing, images are sent afterward to minimize idle time.

Telescoping works as follows. [SHOULD THERE BE A FORMAL ALGORITHM LISTED?] At the onset, the algorithm finds the largest process subgroup that is a power of two. Of the remaining processes, it again finds the largest power-of-two subgroup. This is repeated until all processes belong to some group that has a power of two size. Then the algorithm independently and concurrently runs a parallel image compositing (such as binary swap or radix-k) on each group.

We now assume that each instance of the parallel image compositing results in an image evenly partitioned amongst all processes. Most parallel image compositing algorithms (including binary swap and radix-k) satisfy this criterion. To complete the image compositing, processes in the smaller groups send their partitions to those in the next larger group, starting with the smallest group. The second smallest group receives image data from the smallest, blends this data to its local image data, and sends the results to the third smallest group. This continues until the largest group receives and blends image data from the second largest, at which point the composite is complete. Figure 8 demonstrates the communication pattern of the telescoping algorithm.

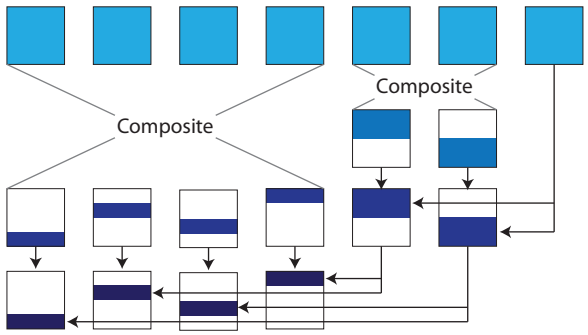


Figure 8: Augmenting an algorithm to composite on a number of processes that is not a power of two.

Although the sending of image data across groups happens sequentially (from smallest to largest), the overhead is minimal. In binary swap and radix-k, smaller process groups have fewer rounds and therefore usually finish earlier. Thus, by the time the largest groups finish their "local" composite, the image data from the next smallest group is usually waiting in an MPI buffer.

Telescoping is most efficient (and easiest to implement) when the partition boundaries of a process group of one size align with those of other process groups. The image partitioning algorithm in IceT ensures this consistent partitioning by first computing the regions for partitions of the largest group and then combining regions for smaller groups. A second criterion of running telescoping efficiently is that it is necessary for a smaller group to know where each partition will be in the larger group. As this partition indexing is a necessary part of the implementation of binary swap or radix-k, determining the same indices for telescoping is trivial. It is also trivial to use telescoping with the minimal-copy image interlacing described in Section 5 since the resulting partitions are the same as those for compositing the largest process group.

Figures 9 and 10 report the performance of binary swap and radix-k on Intrepid with and without telescoping. Unlike the other data given in this paper, these measurements come from the averaging of 10 frames rather than 100 due to the large number of measurements we took. Figure 9 demonstrates that the original radix-k performs well for some process counts but abysmally for others. Figure 10 shows an overhead for folding binary swap analogous to

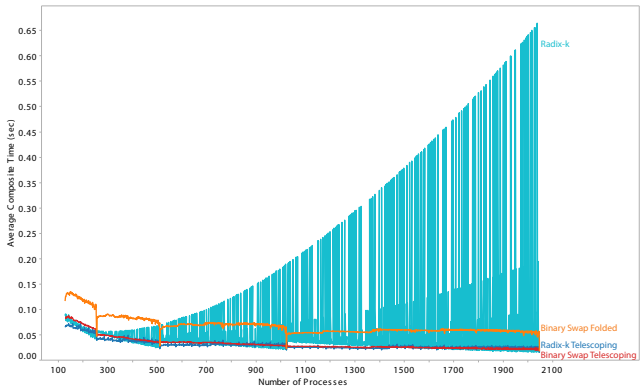


Figure 9: Comparison of telescoping and non-telescoping versions of binary swap and radix-k (favoring $k = 32$) on Intrepid in SMP mode.

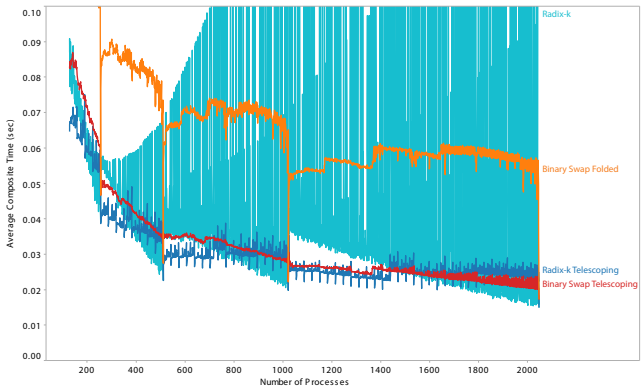


Figure 10: Comparison of telescoping and non-telescoping versions of binary swap and radix-k (favoring $k = 32$) on Intrepid in SMP mode. The vertical axis is scaled to see detail in the telescoping algorithms.

that reported by Yu et al. [31]. Telescoping makes both algorithms perform reasonably consistent for all process counts.

Figure 11 shows the data for the same experiment scheduled on Intrepid in VN mode to larger process counts (and again using 10 frames per measurement). Values for radix-k without telescoping

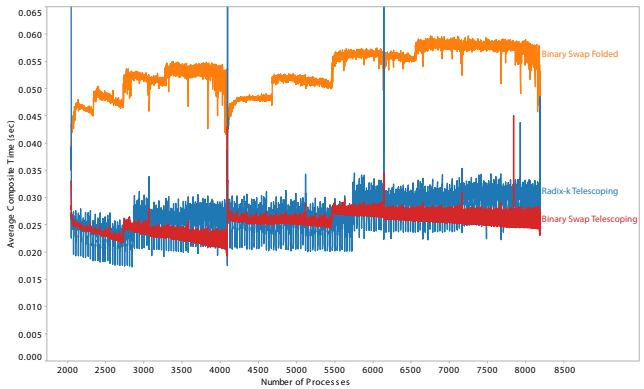


Figure 11: Comparison of telescoping and non-telescoping versions of binary swap and telescoping version of radix-k (favoring $k = 32$) on Intrepid in VN mode.

are not shown because the frame times are too long to measure in the amount of processor time we have available. We did, however, record times for the largest process counts where we expect the worst behavior. Our longest average measurement with unaltered radix-k is 12.88 seconds per frame for 8191 processes (which is, unsurprisingly, the largest prime number of processes for which we ran). The same compositing using the telescoping version of radix-k took only about 0.05 seconds per frame.

We also note some telescoping radix-k measurements in VN mode that are anomalous compared to other process counts (although consistent for all frames of that run). Note the blue spikes in Figure 11. The largest such spike is 0.55 seconds per frame at 4097 processes. We are not sure why these spike occur, but we observe that they happen when a small process group has to send images to a much larger process group. Since these spikes only occur with radix-k and in VN mode, we suspect that this telescoping communication is happening at the same time as the larger group's radix-k communication, and the two interfere with each other.

7 Image Collection

As previously mentioned, parallel compositing algorithms finish with image data partitioned amongst the processes. Although the compositing is technically finished at this point, it is of little practical use if it is split into thousands of pieces and distributed across nodes. Real use cases require the images to be collected in some way. For example, ParaView collects the image to the root node and then sends it over a socket to a GUI on the user's client desktop [3].

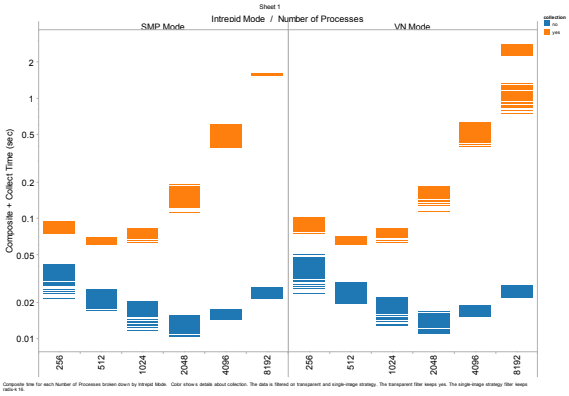


Figure 12: Compositing times on Intrepid when considering time to collect image fragments into single image. All times are given for binary swap.

Because the collection of the image is so important in practice, it is proper to also measure its effect when scaling the image compositing algorithm. Figure 12 compares compositing times with and without image collection using an MPI_Gatherv operation. Although the numbers given here are only given for binary swap, the variance between versions is minor compared to the overhead of collection.

Clearly the MPI_Gatherv collect is not scaling as well with respect to the rest of the compositing algorithm. To make the collection more efficient, we propose limiting the number of partitions created in the parallel compositing algorithm with a simple change to the binary swap and radix-k algorithms. The algorithms proceed in rounds as before. As long as the total number of partitions created remains under a specified threshold, images are split and swapped as normal. However, when this threshold of partitions is reached, images are no longer split. Instead, one process collects

all the image data from other processes in the group. The collection process continues while the other processes drop out. [FIGURE DEMONSTRATING THIS?]

Because this change causes processes to sit idle, it makes the compositing less efficient but potentially makes the collection more efficient. Collecting rather than splitting images is only useful to the point where the improvements in collection outweigh the added inefficiencies in compositing. We should also note that limiting the number of partitions created limits the number of partitions used in the minimal-copy image interlacing described in Section 5, but since our reduced image partitions only contain a few scan lines, the effect is minimal.

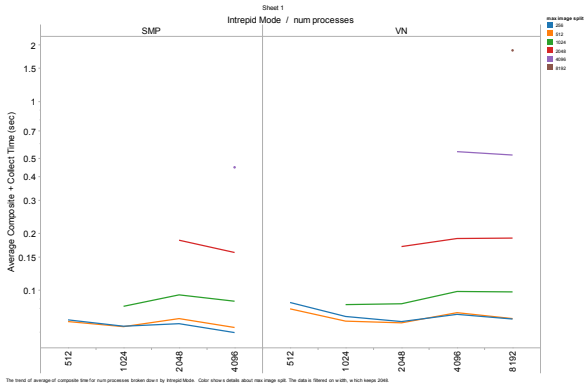


Figure 13: Compositing times on Intrepid when considering the time to collect image fragments into a single image and the total number of partitions is limited by collecting within rounds rather than splitting. All times are given for binary swap.

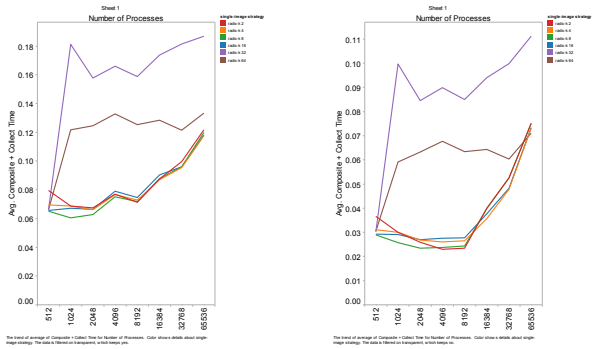
Figure 13 demonstrates the effects of limiting the number of partitions on the binary swap algorithm (radix-k with various values for k is measured in the following section). Due to limits in processor allocation, we have not tested SMP mode with 8192 nodes, but other measurements suggest that the results will be comparable to VN mode with 8192 cores.

Our best composite + collect measurements in Figure 12 occur at 512 partitions, so we limited the maximum number of partitions to 256 and up. Limiting the number of partitions to collect clearly benefits the overall time for larger process counts. Our optimal number of partitions is in the 256–512 range.

8 Scaling of Binary Swap and Radix-k

For our final experiment, we observe the rendering system with all the improvements discussed in this paper (except telescoping because we only considered powers of two) scaled up to massive process counts. Figure 14 summarizes the results. Keep in mind that these timings include collecting image partitions into a single image, a necessary but expensive operation that is often overlooked. All runs come from Intrepid scheduled in VN mode. For all runs we set the maximum number of partitions to 512 although the actual number of partitions is smaller with values of k that do not factor 512 evenly.

For most runs, radix-k with $k = 32$ and $k = 64$ are significantly slower than the others. This is not an effect of the radix-k algorithm itself but rather the maximum number of partitions that we used. For example, two rounds of radix-k with $k = 32$ create $32 \times 32 = 1024$ partitions, which is above our threshold. Thus, the partition threshold is actually throttled back to 32, which results in slower compositing that is not compensated by faster collecting.



(a) Transparent Geometry (b) Opaque Geometry

Figure 14: Performance of binary swap and several versions of radix-k on Intrepid up to 65,536 cores. Transparent rendering uses 4 floats (16 bytes) per pixel, and opaque rendering uses 4 bytes + 1 float (8 bytes) per pixel.

Our results show an increase in overall time for the largest process counts. This increase is primarily caused by an increase in collection times despite our limits on the total number of partitions. [PLOT DEMONSTRATING THIS.] Nevertheless, we are able to completely composite and collect an image on 65,536 cores in less than 0.12 seconds for transparent images (using floating point colors) and in less than 0.075 seconds for opaque images (using 8-bit fixed point colors and floating point depths).

9 Conclusions

In this paper we describe several optimizations to image compositing for sort-last parallel rendering. We also demonstrate our completed system on some of the largest process counts to date. Our findings show that image compositing continues to be a viable parallel rendering option on the largest computers today. These data also suggest a path for future research.

The design of new fundamental compositing algorithms in addition to binary swap, radix-k, and others is probably unnecessary. In our observations, the performance difference between binary swap and the various factorings of radix-k are small compared to the other optimizations of the system such as sparse pixel encoding, load balancing, and image collection. In fact, we find image collection to be the largest overhead currently in our rendering system. Addressing image collection is one of the most promising avenues of future research.

Another fruitful area of research is better methods to take advantage of multi-core processors. Although it is reasonable to ignore the shared memory between the four cores on Intrepid, future computers will have many more cores per node. Some introductory work has analyzed the behavior of image compositing in shared-memory architectures [17, 20], but further refinement is required to take advantage of the hybrid distributed memory plus shared memory architecture of large systems and to evolve the compositing as architectures and rendering algorithms change.

Acknowledgements

Funding for this work was provided by the SciDAC Institute for Ultrascale Visualization and and by the Advanced Simulation and Computing Program of the National Nuclear Security Adiminstration.

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lock-

heed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

REFERENCES

[1] VisIt user's manual. Technical Report UCRL-SM-220449, Lawrence Livermore National Laboratory, October 2005.

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

[3] A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, and J. Farve. Remote large data visualization in the ParaView framework. In *Eurographics Parallel Graphics and Visualization 2006*, pages 163–170, May 2006.

[4] H. Childs. Architectural challenges and solutions for petascale postprocessing. *Journal of Physics: Conference Series*, 78(012012), 2007. DOI=10.1088/1742-6596/78/1/012012.

[5] 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.

[6] D. Higham and B. Wylie. Sandia National Labs achieves breakthrough performance using NVIDIA technology for scientific visualization. Press Release, March 2005. NVIDIA Corporation, http://www.nvidia.com/object/IO_19962.html.

[7] 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.

[8] K.-L. Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, November/December 2009. DOI=10.1109/MCG.2009.120.

[9] 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*, pages 15–22, 1993. DOI=10.1145/166181.166183.

[10] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, July/August 1994. DOI=10.1109/38.291532.

[11] K.-L. Ma, C. Wang, H. Yu, K. Moreland, J. Huang, and R. Ross. Next-generation visualization technologies: Enabling discoveries at extreme scale. *SciDAC Review*, (12):12–21, Spring 2009.

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

[13] K. Moreland. IceT users' guide and reference, version 2.0. Technical Report SAND2010-7451, Sandia National Laboratories, January 2011.

[14] 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*, pages 85–92, October 2001.

[15] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Proceedings of the 1993 Symposium on Parallel Rendering*, pages 97–104, 1993. DOI=10.1145/166181.166196.

[16] U. Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications*, 14(4):49–58, July 1994. DOI=10.1109/38.291531.

[17] B. Nouanesengsy, J. Ahrens, J. Woodring, and H.-W. Shen. Revisiting parallel rendering for shared memory machines. In *Eurographics Symposium on Parallel Graphics and Visualization 2011*, April 2011.

[18] 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.

[19] 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 *International Conference on Parallel Processing (ICPP '09)*, pages 566–573, September 2009. DOI=10.1109/ICPP.2009.27.

[20] E. Reinhard and C. Hansen. A comparison of parallel compositing techniques on shared memory architectures. In *Proceedings of the Third Eurographics Workshop on Parallel Graphics and Visualization*, pages 115–123, September 2000.

[21] R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with k-way replication. In *2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pages 75–84, October 2001.

[22] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 97–108, 2000.

[23] C. Sosa and B. Knudson. *IBM System Blue Gene Solution: Blue Gene/P Application Development*. IBM Redbooks, fourth edition, August 2009. ISBN 0738433330.

[24] A. H. Squillacote. *The ParaView Guide: A Parallel Visualization Application*. Kitware Inc., 2007. ISBN 1-930934-21-1.

[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)*, pages 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*, pages 996–1002, 2003. DOI=10.1145/952532.952728.

[27] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron. From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[28] 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.

[29] 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*, pages 200–207, 1999.

[30] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, May/June 2010. DOI=10.1109/MCG.2010.55.

[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=10.1145/1413370.1413419.