Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

Accelerating and Benchmarking Radix-k Image Compositing at Large Scale

Wesley Kendall[1], Tom Peterka[2], Jian Huang[1], Han-Wei Shen[3], and Robert Ross[2]

[1]The University of Tennessee, Knoxville, TN
[2]Argonne National Laboratory, Argonne, IL
[3]The Ohio State University, Columbus, OH

Abstract

Radix-k was introduced in 2009 as a configurable image compositing algorithm. The ability to tune it by selecting k-values allows it to benefit more from pixel reduction and compression optimizations than its predecessors. This paper describes such optimizations in Radix-k, analyzes their effects, and demonstrates improved performance and scalability. In addition to bounding and run-length encoding pixels, k-value selection and load balance are regulated at run-time. Performance is systematically analyzed for an array of process counts, image sizes, and HPC and graphics clusters. Analyses are performed using compositing of synthetic images and also in the context of a complete volume renderer and scientific data. We demonstrate increased performance over binary swap and show that 64 megapixels can be composited at rates of 0.08 seconds, or 12.5 frames per second, at 32 K processes.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Hardware Architecture]: Parallel processing—I.3.2 [Graphics Systems]: Distributed / network graphics—I.3.7 [Three-Dimensional Graphics and Realism]: Raytracing—I.3.8 [Applications]: Image compositing—Volume rendering

1. Introduction

Data-parallel visualization algorithms are a critical ingredient of large-scale scientific computation. Just as simulations scale to hundreds of thousands of processes and compute tens of billions of grid points per time-step, visualization must be performed at commensurate scale in order to accurately represent the fidelity of the computed data. In sort-last parallel rendering, this requires parallel compositing of images that are tens of megapixels in size, from tens of thousands of processes.

Radix-k is a relatively new, configurable algorithm for parallel image compositing. Its configurations include binary swap, direct-send, and many points in between. One primary distinction of Radix-k when compared to other algorithms is its ability to overlap communication and computation. As a result of this, it can be tuned to underlying hardware support for parallel communication channels and offer superior performance over other methods. Optimizations that bound and compress pixels can allow Radix-k to achieve even better performance and scalability.

In this paper, we demonstrate a new capability for tuning the Radix-k algorithm to various architectures along with benchmarking performance and scalability. The results of our analyses show significant improvements over binary swap, the *de facto* standard for image compositing. Along with tuning to specific architectures, we supplement Radix-k with an advanced compression implementation and a new load-balancing method, allowing for faster compositing and better scalability on diverse architectures. Tests utilized synthetic images where we could control the bounded pixel regions, and a parallel volume renderer with actual scientific data.

Our contributions are (1) augmenting Radix-k with an efficient run-length encoding scheme that compresses message length and accelerates compositing, (2) determining target k-values for various HPC and cluster architectures using these improvements, (3) developing and testing a new algorithm for load balancing, and (4) demonstrating improved performance and scalability on Blue Gene/P, Cray XT5, and graphics clusters in a parallel volume rendering application.

2. Background

In sort-last parallel rendering [MCEF94], data are partitioned among processes, rendering occurs locally at each process, and the resulting images are depth-sorted or composited at the end. Cavin et al. [CMF05] surveyed methods for sort-last compositing and analyzed relative theoretical performance. Compositing algorithms traditionally are categorized as direct-send [Hsu93, Neu94, MI97], binary swap [MPHK94], and parallel pipeline [LRN96]. Traditionally used in distributed memory architectures, they have also been tested in shared memory machines [RH00]. Direct-send and parallel pipeline methods have been eclipsed by binary swap, although hybrid combinations of direct-send and binary swap are being studied. Nonaka et al. [NOM08] combined binary swap with direct-send and binary tree to improve performance. Yu et al. [YWM08] extended binary swap compositing to non-power-of-two numbers of processes in 2-3 swap compositing by communicating in groups of two and three members.

The Radix-k algorithm [PGR*09] allows the amount of communication concurrency to be tuned to the architecture. By selecting appropriate k-values (the number of messaging partners in a group), it can offer more parallelism than binary swap while avoiding message contention that can occur at large numbers of processes with direct-send. Radix-k does not incur a penalty for non-power-of-two processes, unlike the 2-3 swap extension to binary swap. Through careful implementation of nonblocking communication, Radix-k overlaps the compositing with communication as much as possible. The benefits, however, depend on the underlying hardware support; when an architecture does not manage concurrent messages or overlapped computation effectively, it is better to select smaller k-values, closer to binary swap.

Direct-send and binary swap can be optimized by scheduling communication and by exploiting image characteristics such as spatial locality and sparseness. The SLIC algorithm [SML*03] schedules communication in direct-send. Run-length encoding (RLE) images before compositing achieves lossless compression. For example, Ahrens et al. [AP98] reported an average RLE compression ratio of approximately 20:1 for intermediate zoom levels.

Using bounding boxes to identify "active" pixels is a practical way to reduce message size [MWP01]. Neumann [Neu93] estimated that the active image size for each process could be approximated by

$$\frac{i}{p^{2/3}} \tag{1}$$

where i is the image size and p is the number of processes. Yang et al. [YYC01] analyzed bounding boxes and RLE optimization for binary swap and Takeuchi et al. [TIH03] implemented binary swap with bounding boxes, interleaved splitting, and RLE. Takeuchi et al. began with the local image computed by Equation 1, and they reported RLE compression ratios of approximately 7:1.

Image compositing can be implemented in parallel rendering using libraries such as IceT [MWP01]. Chromium [HHN*02] is another system that supports compositing across cluster nodes and tiled displays. In addition to volume rendering, these libraries support polygon rendering by including a depth value per pixel and modifying the compositing operator [MAF07].

3. Implementing Improvements to Radix-k

A complete description of the original, unoptimized Radix-k algorithm can be found in [PGR*09]. Briefly, compositing occurs in rounds, each round containing a number of groups that operate independently. Binary swap uses groups of size two and direct-send uses one group for all processes. In Radix-k, groups can be factored into distinct sizes of k for each round.

For example, 64 processes can be composited with k-values of [2, 2, 2, 2, 2, 2] (binary swap), or with a single round of k = [64] (direct-send), or something in between such as k = [8, 8]. Unlike kd-partitioning in some binary swap implementations [MPHK94], the image is partitioned by scanlines. Within a round, each group of k participants performs a direct-send message pattern local to that group, such that each participant takes ownership of 1/k of the current image being composited by that group. This design philosophy avoids network contention that may occur in direct-send, while being able to saturate the network more than binary swap. We discuss our implementation of pixel compression, automatic k-value selection, and load balancing below.

3.1. Accelerations

By delineating pixels with bounding boxes, compositing can be accelerated by operating on only the pixels that were produced by a local node, rather than on the entire image. As the number of compositing rounds grows, however, the union of bounding boxes from earlier rounds can accumulate many unnecessary pixels. This can be avoided by run-length encoding (RLE) the images.

Typical RLE implementations store a count of contiguous identical pixels in scanlines, followed by the pixel value, in the same buffer [AP98]. This approach has several drawbacks. Transfer functions often create few contiguous identical pixels, hindering the amount of compression and adding overhead. Taking subregions of an encoded image, a task that is required in each round, is costly when one buffer has both index and pixel values because it requires additional copying. Also, byte alignment issues can arise when pixels and counts are different sizes and do not align on word boundaries.

We designed a different implementation to avoid these drawbacks. In our implementation, RLE is used to compress only empty areas and the bounding box information is utilized to accelerate this step. Two separate buffers are used: one for storing alternating counts of empty and nonempty pixels in scanlines, and one for storing the nonempty pixels. Using this implementation, a subset of an encoded image can be taken by first constructing a new index to represent the subset, and then by assigning a pointer to the proper location in the pixel buffer of the fully encoded image. This way, unnecessary memory copy and allocation are avoided.

Similar to [AP98, MWP01], images remain encoded throughout compositing, but we do not iterate through each pixel of the encoded images. Overlapping areas are first computed using the index of both images. By doing this, complete chunks of nonoverlapping areas are copied to the final image, and only the overlapping nonempty pixels are visited when compositing.

3.2. K-value Selection

The choice of k-values plays a vital role in maximizing communication and compositing overlap. The k-values depend on the machine architecture (network topology, bandwidth, latency), number of processes, and image size. Results from [PGR*09] showed that k-values of eight performed well on Blue Gene/P for various cases using unoptimized Radix-k. Compression, however, changes the performance of these k-values.

We tested a wide array of process counts and image sizes to discover "target" k-values for numerous architectures. The target k-value represents the optimal size of a group for any round. We used these results in our implementation as a lookup table for k-value selection. We envision similar testing can be automated to appropriately select k-values when installed on new architectures.

3.3. Load Balancing

While reducing the total amount of compositing work, bounding boxes and compression can cause work distribution among processes to become uneven. Research has been conducted to load balance binary swap by interleaving scanlines among processes [TIH03]. One central reason exists why we chose not to implement this method: when the image is completely composited, processes do not contain contiguous portions of the image. If the image is to be written in parallel, either an extra global communication step to create contiguous portions is introduced or there are many more disk writes and seeks when writing the image. Both of these have severe performance implications. We describe a new method for load balancing within and across groups that keeps the image distributed in contiguous portions.

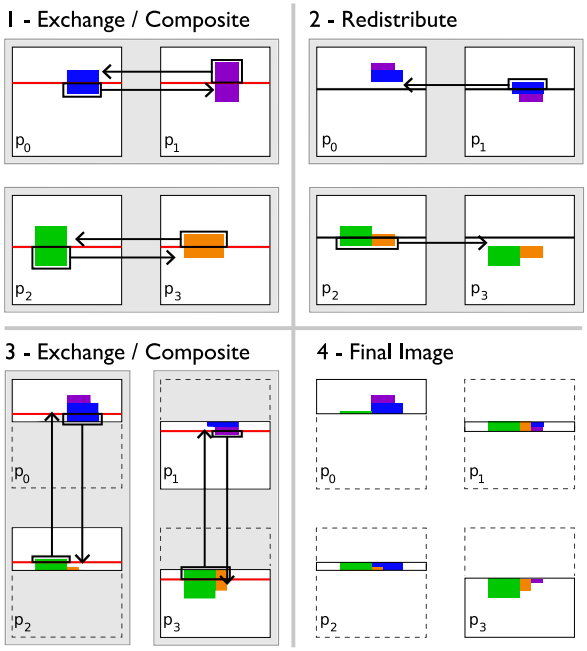


Figure 1: Illustration of load balancing using four processes and k-values of two. In the first step, groups (enclosed by the gray areas) compute a local partition (red lines) of their image such that group members perform equal compositing. In the second step, all processes compute a global partition (black lines) to ensure that groups in the third step will operate on matching regions. Some group members have to redistribute composited areas because of this. The third step operates like the first, and the final image is shown in the fourth step.

Figure 1 provides a simple illustration of our load balancing algorithm using four processes and group sizes of two. The boxes represent the local image on each process, and the gray areas around the boxes indicate the groups. We describe partitions of images as either local or global. A local partition is computed by a group of processes and a global partition is computed by every process. Red lines are used to denote local partitions and black lines indicate global partitions.

In the first step, groups compute a local partition of the image such that the area encompassed by the bounding boxes will be evenly divided among group members. The respective regions are exchanged and the compositing operator is performed. The composited subimages are shown in the second step.

In the second step, all processes must agree on a global partition of the image such that groups in the third step will operate on matching regions. The global partition is created by dividing the area covered by the bounding boxes across all processes, ensuring that later intergroup compositing

workloads will be balanced. Because of this new partition, processes may have to redistribute parts of their composited subimages to other members in the group. For example, in the second step, p_1 has to give the region above the partition to p_0. This is because p_1 is only operating on the lower region in the third step. Similarly, p_0 is operating on the upper region in the third step and needs this portion of data for correct compositing to occur.

The third step operates like the first one. The groups compute a local partition of the image such that group members perform equal compositing work. The respective regions are exchanged and the compositing operator is performed to obtain the final image shown in the fourth step.

Local and global partitions are computed using the same algorithm with different inputs: the former uses bounding boxes of the group while the latter uses bounding boxes of all processes. A simple method for determining the best partition is to compute a count of the number of bounding boxes that overlap each pixel. An equal division of the area covered by bounding boxes could then be determined by equally dividing the prefix sum of all the counts. This algorithm is $O(wh)$ where w and h are the width and the height of the image. Our approach computes a count of the widths of the bounding boxes that overlap each row of the image. We can then compute a prefix sum of the counts and estimate the best partition. For cases when estimating the division of a single row, the minimum and maximum column values of the overlapping bounding boxes of the row are taken into account. This algorithm is $O(h)$ and showed little difference in load imbalance when compared to the $O(wh)$ approach.

Our method assumes that each process has one or more bounding boxes containing nonempty data. This assumption is reasonable but does introduce unequal work when bounding boxes contain empty pixels. Our method also requires a collective gathering (*MPI_Allgather*) of the bounding boxes of all processes. This communication step occurs only once, and the overhead was negligible in all of our tests.

3.4. Test Environment

For testing, synthetic images were used for tuning k-values to architectures (Section 4) and load balancing (Section 5). These are checkerboard patterns that are slightly offset from one process to another and have partial transparency for each pixel. The number of bounded pixels per process was determined using Equation 1, and the location of the bounded area was determined randomly. Our synthetic image benchmark also has the ability to perform zooming. Test results for the synthetic patterns were compared with a serial version of the compositing code to verify correctness. In Section 6, a scientific dataset was used instead of synthetic images. The dataset is from a simulation of the standing accretion shock instability in a core-collapse supernova [Iri06]. In all cases, a four-byte floating-point value (0.0 - 1.0) for each of the four R,G,B,A channels represented each pixel. Image sizes varied from 4 to 64 megapixels. We tested power-of-two processes to compare against binary swap; however, [PGR*09] showed that Radix-k performs similarly at other process counts.

The volume renderer for the scientific dataset is a software ray caster that uses kd-partitioning to divide the data in object space. It has no acceleration schemes such as early ray termination or load balancing. The renderer stores bounding box information for the image at each process. We note that our Radix-k enhancements also apply to ray casters that use multiple bounding boxes.

Our tests were conducted on two platforms at the Argonne National Laboratory and two platforms at the Oak Ridge National Laboratory. At Argonne, the IBM Blue Gene/P *Intrepid* contains 40,960 nodes consisting of quad-core 850 MHz IBM PowerPC processors. The nodes are connected in a 3D torus topology. Our tests were conducted in SMP mode, that is, one process per node. *Eureka* is a graphics cluster that contains 100 nodes consisting of two quad-core 2 GHz Intel Xeon processors and two nVidia FX5600 graphics cards. The compute nodes are connected by a 10 Gb/s Myrinet switching fabric.

At Oak Ridge, the Cray XT5 *Jaguar*, currently the fastest supercomputer in the world, consists of 18,688 nodes that contain two hex-core 2.6 GHz AMD Opteron processors. The nodes are connected in a 3D torus topology. *Lens* is a 32 node visualization and analysis cluster. Each node contains four quad-core 2.3 GHz AMD Opteron processors. The nodes are connected by a 20 Gb/s DDR Infiniband network.

4. Tuning to Architectures

We begin with an analysis of the target k-values to use for various architectures. Our goal is to have a single target k-value chosen by a lookup table based on number of processes and image size, so that the algorithm can automatically determine the k-values for each round. This analysis is performed with the bounding box and RLE optimizations using synthetic images.

We tested image sizes of 4, 8, 16, and 32 megapixels five times each with random bounding box locations and selected the k-value with the lowest time. We used our synthetic checkerboard images and bounding box sizes that were determined by Equation 1. We tested k-values of 2, 4, 8, 16, 32, 64, and 128.

To quantify the overlap of communication with compositing computation, we used

$$O = \frac{Computation}{(CommWait + Computation)} \tag{2}$$

where *CommWait* represents the time spent waiting during communication and *Computation* is the time spent performing the compositing operator. Figure 2 provides an illustration of these. O is used to estimate the overlap of

p \ i	4	8	16	32
8	8	8	8	8
16	16	16	16	16
32	32	32	32	32
64	64	64	64	64
128	64	128	128	128
256	64	128	128	128
512	64	128	128	128
1 K	64	128	128	128
2 K	32	128	128	128
4 K	32	32	32	32
8 K	32	32	32	32
16 K	32	32	32	32
32 K	32	32	32	32

(a) Intrepid

p \ i	4	8	16	32
8	4	4	4	4
16	4	4	4	4
32	16	8	16	16
64	16	16	16	16
128	8	8	8	8
256	16	8	8	8
512	16	32	8	8
1 K	64	32	32	8
2 K	8	32	32	32
4 K	8	16	32	64
8 K	4	8	32	64
16 K	4	32	32	8
32 K	16	16	64	8

(b) Jaguar

p \ i	4	8	16	32
8	8	8	8	8
16	16	16	16	16
32	32	32	32	32
64	16	32	16	32
128	64	64	64	64

(c) Lens

p \ i	4	8	16	32
8	8	8	8	8
16	8	16	16	16
32	32	32	32	32
64	32	32	32	32
128	32	64	64	32

(d) Eureka

Table 1: Target k-values for (a) Intrepid, (b) Jaguar, (c) Lens, and (d) Eureka where p is the number of processes and i is the image size in megapixels. The target k-values represent the best group sizes to use when compositing and vary with different image sizes and process counts. These tables were encoded into our Radix-k implementation and used in all experiments.

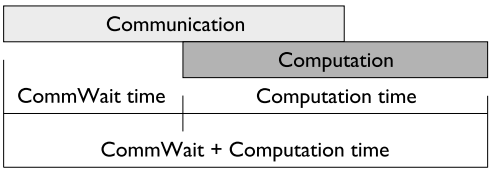


Figure 2: Illustration of the *CommWait* and *Computation* parameters in Equation 2. These parameters are useful in estimating the overlap of communication and computation when compositing.

communication with computation. Higher O values represent greater overlap efficiency. Along with O, we computed the relative compression ratio by dividing the amount of data that would have been transferred without RLE by the amount transferred with RLE.

Results for the target k-values on Intrepid are shown in Table 1a. The average O for all the target k-values was 0.58, meaning there was significant overlap between compositing and communication, thus allowing us to go to higher k-values. Since there are six bidirectional links per node, communication and compositing can be overlapped to a high degree. When scaling to four racks or more ($p >= 4$ K), we observed compression ratios of 145:1 and presumed that the smaller k-values are better for higher latency overhead and smaller messages. At $p \leq 64$, direct-send performed the best, that is, $k = p$. Binary swap ($k = 2, 2, \ldots$) was never the optimal algorithm on Intrepid.

Jaguar results appear in Table 1b. The average O for all the target k-values was 0.22, meaning there was less overlap between communication and computation; thus, Jaguar was not performing as well at higher k-values compared to Intrepid. Both machines have 3D tori, but Jaguar's 6 bidirectional links must support 12 cores. In addition, Jaguar's interconnect is shared between all running jobs, unlike Blue Gene/P which dedicates a partition to one job. There is also no guarantee on Jaguar about the physical placement of nodes with respect to each other, meaning a partition can be fragmented or have a suboptimal shape. In general, using a target k-value of 8 or 16 gave good performance. Unlike Intrepid, there were no immediate patterns recognized when scaling to many cabinets. Direct-send and binary swap were never optimal.

Lens results can be found in Table 1c. The average O for all target k-values was 0.29. Lens has 16 cores per node, allowing more intranode communication and less network contention. We believe this was the primary reason why the O value of Lens was greater than that of Jaguar. Direct-send was the optimal algorithm in almost every case.

Eureka results are shown in Table 1d. The average O for all target k-values was 0.10. The results showed that direct-send is the optimal algorithm in almost every case. We credit this to higher compression ratios that were occurring with higher k-values. For example, direct-send was showing a compression ratio of 25:1 compared to the 10:1 compression ratio of binary swap.

We encoded these tables for Intrepid, Jaguar, Lens, and Eureka into our algorithm and selected k-values automatically from the tables for the remainder of our tests. We performed a nearest-neighbor selection in the table when using untested parameters. When process counts were not divisible by the target k-value, we determined the factor closest to the target k-value.

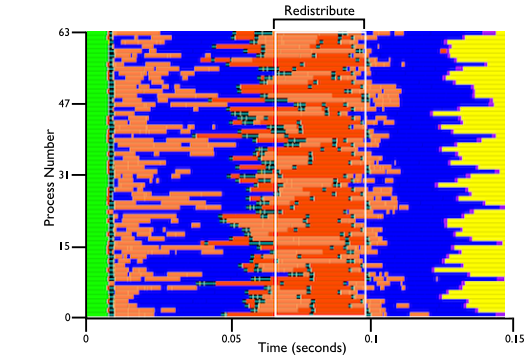
5. Load Balancing

We provide an example of how our load balancing method affects the computation workload across processes. Figure 3 shows a comparison of Jumpshot logs using the synthetic benchmark on Intrepid with (3a) and without (3b) load balancing for 64 processes and a target k-value of 8. Jumpshot is a profiling tool [CGL08] that shows time on the horizontal axis, and each process is a row in the image. The blue areas denote the computation while salmon and red regions are communication. Green represents the encoding of the images and yellow represents idle time. The load-balanced upper image shows a more even distribution of compositing across processes. Figure 3a has a white rectangle around the region where the redistribution step occurred during load balancing. This corresponds to step two of Figure 1. It is evident this step is quite expensive, and as we will see, it does not always pay off. The goal of our analysis is to determine for how many processes, and for what level of imbalance, our load balancing algorithm reduces the total compositing time.

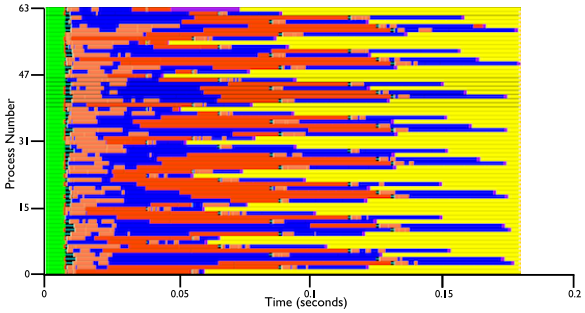
We tested load balancing of the synthetic benchmark on each architecture at various process counts with an image size of 32 megapixels. We used *zoom* factors of 1.0 and 0.5 in our tests. Each test was averaged over five random bounding box locations. We used the target k-values found from Section 4 for each architecture. Figure 4 shows the improvement factor (time without load balancing divided by time with load balancing) that was obtained in this experiment.

Intrepid (4a) benefited from using our load balancing method at $p < 1$ K. An improvement factor of over 2.5 is shown in some cases when $zoom = 0.5$. The redistribution step, however, dominated the overall benefits of balanced compositing computation at $p \geq 1$ K. Similarly, Jaguar (4a) showed reduced performance when scaling to higher numbers of processes. The performance gains of using load balancing on Jaguar were not as apparent as Intrepid. This correlates with Jaguar having lower k-values than Intrepid, causing more rounds of compositing, thus more expensive redistribution steps.

Lens (4b) showed over double increase in performance in some circumstances, and the redistribution step is avoided in many cases because $k = p$. On the other hand, Eureka (4b) results were mixed. Although Lens and Eureka have similar switched networks, the network bandwidth on Lens is twice that of Eureka's, which helped accommodate the additional communication needed for load balancing.



(a) With Load Balancing



(b) Without Load Balancing

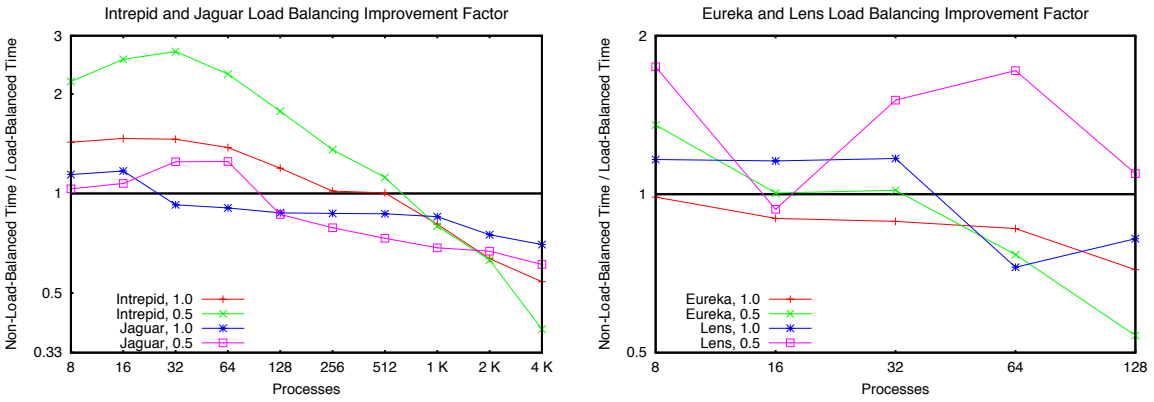
Figure 3: Jumpshot logs showing the synthetic benchmark (a) with and (b) without load balancing using 64 processes. Time is plotted on the x-axis and process numbers are plotted on the y-axis. The blue color denotes the compositing operator, while salmon and red indicate communication and waiting on messages. Green represents the encoding of the images and yellow represents idle time. The white rectangle in (a) encompasses the redistribution step explained in Figure 1.

We quantify when to use load balancing with

$$L = \frac{\sigma_{Work}/\mu_{Work}}{r} \tag{3}$$

where σ_{Work} and μ_{Work} are the standard deviation and mean of work across all processes in the first compositing round, and r is the number of rounds. We only considered the work in the first round in Equation 3 because the first round is most sensitive to imbalance. L is used to decide when to turn load balancing on and off, and higher L values indicate when our load balancing method should give us better performance. Dividing by r compensates for the fact that load balancing is more costly for more rounds.

We also tested other variants of the algorithm that avoided multiple redistribution steps. One variant was to simply turn off load balancing after the first round. The majority of the results showed highly skewed intergroup load imbalance in subsequent rounds and generally gave poorer results.



(a) Intrepid and Jaguar

(b) Eureka and Lens

Figure 4: Improvement factor of using load balancing when compared to no load balancing (log-log scale). This test was conducted on a 32 megapixel image in the synthetic benchmark using *zoom* factors of 1.0 and 0.5. Improvement factor is calculated by dividing the time without load balancing by the time with load balancing. A factor below one indicates that using no load balancing resulted in better performance.

Another variant that avoided the redistribution step was using one global partition of the image that divided all bounding boxes evenly. This also often resulted in skewed workload across groups and generally resulted in worse performance.

In Radix-k, we completely turned off load balancing at $p > 512$ because of the results from Figure 4. Images are so finely partitioned at scales greater than 512 processes that compositing is more communication-bound rather than computation-bound. When $p \leq 512$, we turned on load balancing above certain thresholds of L in our implementation. We found $L > 0.5$ to fit most of the results for Intrepid, Jaguar, and Lens. $L > 0.8$ was used for Eureka.

6. Volume Rendering Large Images

We tested Radix-k in a volume renderer and rendered images of a supernova core collapse simulation. We used three *zoom* levels, depicted in Figure 5. We tested strong scaling and performed a comparison against binary swap on all architectures. The previously discussed optimizations were used in our Radix-k implementation. Timing results included the entire compositing algorithm, including encoding and decoding of images.

6.1. Scalability

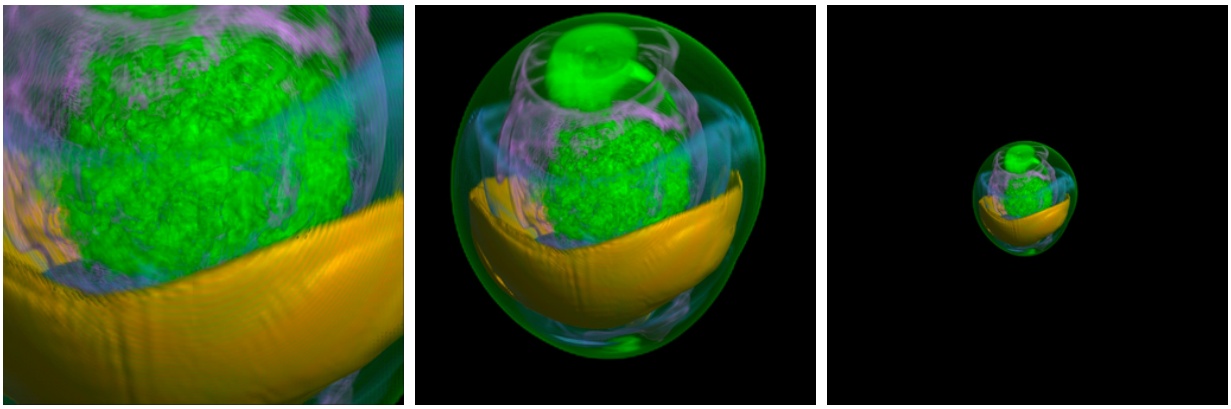
Strong scaling tests were conducted using a 64 megapixel image size with the three scenarios depicted in Figure 5. Results are shown in Figure 6 for the three *zoom* factors. K-values were automatically selected using the nearest value in Table 1 for the number of processes and image size, and load balancing was turned on based on the rules derived in Section 5.

Intrepid (6a) showed decreased time at more processes in most cases. When load balancing was turned on, we observed an improvement factor of over 1.5 in most cases when compared to no load balancing. Jaguar (6a) showed similar behavior to Intrepid. At our intermediate *zoom* factor of 1.5, we were compositing the 64 megapixel image in 0.08 seconds (12.5 FPS) at 32 K processes, a near-interactive frame rate. When load balancing was turned on, an improvement factor of over 1.5 was gained in most cases when compared to no load balancing.

Eureka (6b) was the only system that did not show decreased time at larger process counts. We credit this to the poor O values that were observed when doing the initial k-value testing in Section 4 and conclude the network was a bottleneck; however, further benchmarking is needed to verify this. Lens (6b) showed modest decreased compositing time under most circumstances. For the cases when load balancing was turned on, an improvement factor of over 1.7 was obtained many times when compared to not using load balancing.

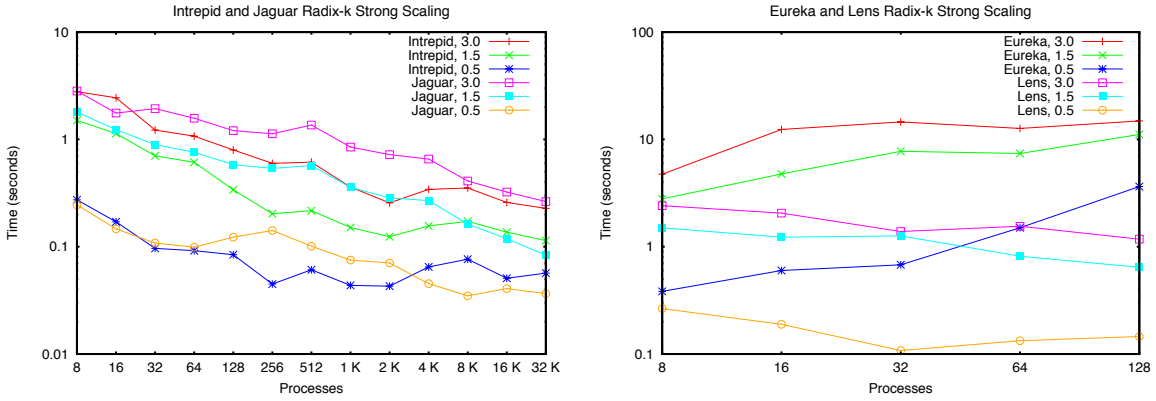
6.2. Comparison to Binary Swap

For all architectures, we compared the improvement factor of Radix-k versus binary swap (binary swap time divided by Radix-k time) for the three scenarios depicted in Figure 5 and 64 megapixel image sizes. It is important to note that our binary swap used k-values of two and the same RLE implementation as Radix-k. Figure 7 shows the results.



(a) $zoom = 3.0$ (b) $zoom = 1.5$ (c) $zoom = 0.5$

Figure 5: Volume rendering of one time step of a core-collapse supernova simulation at various *zoom* levels. We used these three scenarios in our volume rendering experiments.



(a) Intrepid and Jaguar (b) Eureka and Lens

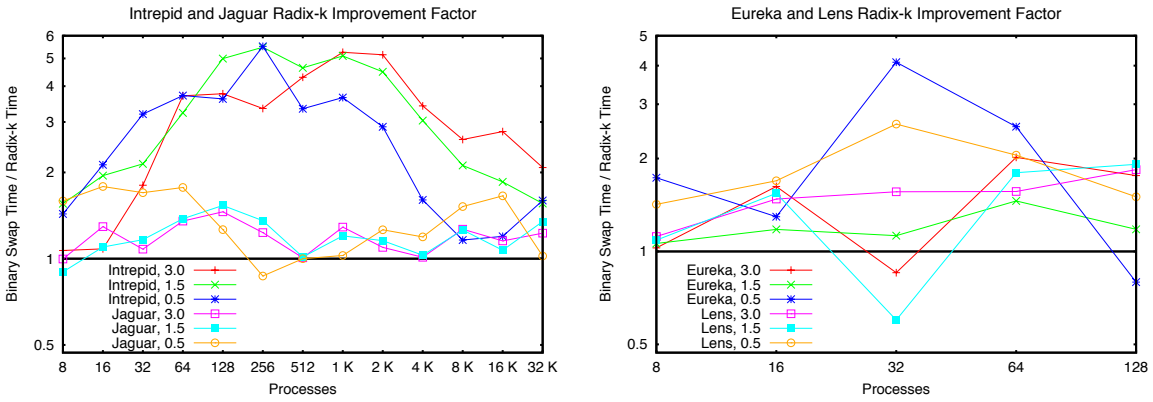
Figure 6: Strong scaling of Radix-k (log-log scale). The experiment was conducted using our volume renderer on a 64 megapixel image size with the three *zoom* levels depicted in Figure 5.

Binary swap outperformed Radix-k in a few experiments, suggesting that our target k-values were not always optimal for the renderings. The results, however, showed improved performance over binary swap at almost all scales. For many results, compression ratios were up to ten times better than binary swap, further helping provide the improved performance.

An improvement factor of over 1.5 was obtained in most cases on the visualization clusters (7b). Jaguar (7a) showed modest improvement at most scales, but improvement factors of over 1.5 were noticed in some situations. We surmise that this is because smaller k-values were being used. On Intrepid (7a), Radix-k was over five times as fast as binary swap in many experiments. This result showed the benefits of using higher k-values on architectures like Blue Gene/P that have many network interconnects per processor.

7. Summary

In the past, compositing at many processes did not scale well. Increasing the number of processes resulted in either a flat or slowly increasing compositing time. At tens of thousands of processes, this performance degradation can be severe [PYR*09]. Most of our results showed decreased time at larger process counts when each process started with the same complete image size. Such scalability out to 64 megapixel image size and 32 K cores is needed in order for parallel visualization to keep pace with the growing scale of simulation data.



(a) Intrepid and Jaguar

(b) Eureka and Lens

Figure 7: Improvement factor of Radix-k over binary swap (log-log scale). The experiment was conducted using our volume renderer on a 64 megapixel image size with the three *zoom* levels depicted in Figure 5. Improvement factor is calculated by dividing binary swap time by Radix-k time. A factor below one indicates that binary swap performed better.

The computational power of today comes in many variations. We have shown that the improvements to the Radix-k algorithm can be geared towards a wide variety of these computational architectures. Graphics clusters, often containing ordinary networks, benefit from the additional compression that comes with using RLE and high k-values in Radix-k. On the other hand, cutting-edge machines benefit from having multiple network interconnects that can overlap computation with the compressed messages to higher degrees.

The improvements to Radix-k, which include our efficient RLE scheme and new load balancing method, outperformed optimized binary swap at even higher factors than reported in [PGR*09]. We have also shown that we can automate the process of determining a proper configuration for any architecture, a process that is needed to place Radix-k in production use to further make impacts in small- and large-scale visualization applications.

In our current and future work, we are continuing to prepare Radix-k for production use. To do so, we plan to implement polygon compositing with individual depth values for each pixel, and to benchmark Radix-k with volume renderers that produce multiple bounding boxes per process. For better scalability of load balancing, we also want to research methods that balance communication along with computation, and we would like to compare our method with the interleaved scanline algorithm presented in [TIH03]. Another area for future study is for time-varying volume rendering where we can use occlusion information from previous time steps to ease the compositing and communication workload. Our primary future goal is to implement Radix-k in production image compositing libraries such as IceT that are used in mainstream visualization packages like ParaView and VisIt.

8. Acknowledgment

Funding for this work is primarily through the Institute of Ultra-Scale Visualization (`http://www.ultravis.org`) under the auspices of the SciDAC program within the U.S. Department of Energy (DOE). This research used resources of the National Center for Computational Science (NCCS) at Oak Ridge National Laboratory (ORNL), which is managed by UT-Battelle, LLC, for DOE under Contract DE-AC05-00OR22725. We gratefully acknowledge the use of the resources of the Leadership Computing Facilities at Argonne National Laboratory and Oak Ridge National Laboratory. We would also like to thank John Blondin and Tony Mezzacappa for their dataset and the generous support of ORNL's visualization task force led by Sean Ahern.

Research was supported in part by a DOE Early Career PI grant awarded to Jian Huang (No. DE-FG02-04ER25610) and by NSF grants CNS-0437508 and ACI-0329323. This work was also supported by the Office of Advanced Scientific Computing Research, Office of Science, DOE, under Contract DE-AC02-06CH11357 and DE-FC02-06ER25777.

References

[AP98] AHRENS J., PAINTER J.: Efficient sort-last rendering using compression-based image compositing. In *Proc. Eurographics Parallel Graphics and Visualization Symposium 2008* (Bristol, United Kingdom, 1998).

[CGL08] CHAN A., GROPP W., LUSK E.: An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming 16*, 2-3 (2008), 155–165.

[CMF05] CAVIN X., MION C., FIBOIS A.: Cots cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In *Proc. IEEE Visualization 2005* (2005), pp. 111–118.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics 21*, 3 (2002), 693–702.

[Hsu93] HSU W. M.: Segmented ray casting for data parallel volume rendering. In *Proc. 1993 Parallel Rendering Symposium* (San Jose, CA, 1993), ACM, pp. 7–14.

[Iri06] IRION R.: The terascale supernova initiative: Modeling the first instance of a star's death. *SciDAC Review 2*, 1 (2006), 26–37.

[LRN96] LEE T.-Y., RAGHAVENDRA C. S., NICHOLAS J. B.: Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (1996), 202–217.

[MAF07] MORELAND K., AVILA L., FISK L. A.: Parallel unstructured volume rendering in paraview. In *Proc. IS&T SPIE Visualization and Data Analysis 2007* (San Jose, CA, 2007).

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32.

[MI97] MA K.-L., INTERRANTE V.: Extracting feature lines from 3d unstructured grids. In *Proc. IEEE Visualization 1997* (Phoenix, AZ, 1997), pp. 285–292.

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications 14*, 4 (1994), 59–68.

[MWP01] MORELAND K., WYLIE B., PAVLAKOS C.: Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *PVG 2001: Proc. IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics* (Piscataway, NJ, USA, 2001), IEEE Press, pp. 85–92.

[Neu93] NEUMANN U.: Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Proc. 1993 Parallel Rendering Symposium* (San Jose, CA, 1993), pp. 97–104.

[Neu94] NEUMANN U.: Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications 14*, 4 (1994), 49–58.

[NOM08] NONAKA J., ONO K., MIYACHI H.: Theoretical and practical performance and scalability analyses of binary-swap image composition method on ibm blue gene/l. In *Proc. 2008 International Workshop on Super Visualization (unpublished manuscript)* (Kos, Greece, 2008).

[PGR*09] PETERKA T., GOODELL D., ROSS R., SHEN H.-W., THAKUR R.: A configurable algorithm for parallel image-compositing applications. In *Proc. SC 2009* (Portland OR, 2009).

[PYR*09] PETERKA T., YU H., ROSS R., MA K.-L., LATHAM R.: End-to-end study of parallel volume rendering on the ibm blue gene/p. In *Proc. ICPP 2009* (Vienna, Austria, 2009).

[RH00] REINHARD E., HANSEN C.: A comparison of parallel compositing techniques on shared memory architectures. In *Proc. Third Eurographics Workshop on Parallel Graphics and Visualization* (2000), pp. 115–123.

[SML*03] STOMPEL A., MA K.-L., LUM E. B., AHRENS J., PATCHETT J.: Slic: Scheduled linear image compositing for parallel volume rendering. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Seattle, WA, 2003), pp. 33–40.

[TIH03] TAKEUCHI A., INO F., HAGIHARA K.: An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Computing 29*, 11-12 (2003), 1745–1762.

[YWM08] YU H., WANG C., MA K.-L.: Massively parallel volume rendering using 2-3 swap image compositing. In *Proc. SC 2008* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 1–11.

[YYC01] YANG D.-L., YU J.-C., CHUNG Y.-C.: Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *Journal of Supercomputing 18*, 2 (2001), 201–220.