Optimizing Multi-Image Sort-Last Parallel Rendering

Matthew Larsen *
Lawrence Livermore Nat'l Lab
University of Oregon

Kenneth Moreland †
Sandia Nat'l Lab

Chris R. Johnson ‡
University of Utah

Hank Childs §
Lawrence Berkeley Nat'l Lab
University of Oregon

ABSTRACT

Sort-last parallel rendering can be improved by considering the rendering of multiple images at a time. Most parallel rendering algorithms consider the generation of only a single image. This makes sense when performing interactive rendering where the parameters of each rendering are not known until the previous rendering completes. However, *in situ* visualization often generates multiple images that do not need to be created sequentially. In this paper we present a simple and effective approach to improving parallel image generation throughput by amortizing the load and overhead among multiple image renders. Additionally, we validate our approach by conducting a performance study exploring the achievable speed-ups in a variety of image-based *in situ* use cases and rendering workloads. On average, our approach shows a 1.5 to 3.7 fold improvement in performance, and in some cases, shows a 10 fold improvement.

1 INTRODUCTION

In this paper, we explore new algorithms for multi-image sort-last parallel rendering, which is becoming increasingly common for scientific visualization. The algorithm frequently used for this problem has evolved from approaches from interactive exploration within visualization tools. However, with multi-image rendering, the desired images are all known prior to rendering, which opens the door to improve efficiency.

In a traditional scientific visualization setting, the list of images to generate are not known a priori. End users interact with visualization software and view the imagery produced from their direction, iteratively exploring their data by repeatedly selecting and rendering. End users want to see surfaces or volumes from a specific position, the visualization software renders the data from that position, and the end users view it. This process is repeated as the user navigates to new camera positions. As a result, visualization software renders images one at a time in response to user guidance.

This "render images one at a time" model is likely not efficient for sort-last parallel rendering. Sort-last parallel rendering frequently has load imbalance, and it is difficult to achieve efficient rendering without taking extra steps (i.e., rebalancing geometry, etc.) when rendering just a single image. The traditional model for sort-last parallel rendering consists of two phases — rendering (or sub-image generation) and compositing — and the total time for sort-last parallel rendering is the sum of the times for these two phases. In the rendering phase, each processing element generates the image from its portion of the larger data set. The resulting image contains depth information (i.e., the z-buffer), so that the compositing phase can combine the sub-images into a single image, as if all data was rendered on a single processing element. The compositing phase occurs after the rendering phase, since all sub-images are needed to generate the final picture. This compositing phase has been the subject of several research efforts to understand the most efficient communication patterns for combining the sub-images[14, 19, 7, 20].

There are multiple sources of load imbalance with sort-last parallel rendering. For the rendering phase, the amount of geometry on each processing element will almost certainly be varied, and maybe significantly so. For example, when rendering the results of an isosurface, one processing element may have many triangles (if the field was complex in the spatial region it owned), and another processing element may have zero triangles (if the field was homogeneous in its region). Further, rendering time is often affected by how many pixels need to be updated (fill rate). Therefore, a second source of load imbalance for rendering stems from camera position. That is, two processing elements may have the same amount of data to render, but one may take much longer to render if the camera is zoomed in on its spatial region. Further, for the compositing phase, the time to execute is much longer than the time it takes to compare pixels from other processing elements. Specifically, network transfers have latencies that affect performance. Additionally, it is difficult to ensure that all processing elements have meaningful work to perform, although algorithms such as binary swap and radix-k aim to balance work as effectively as possible.

Recently, a new paradigm for *in situ* processing has emerged that makes heavy use of sort-last parallel rendering. The idea behind this paradigm is to eschew saving of meshes and fields, and, instead, to generate many images for a given time slice while the simulation is running. These images come from many camera positions and may consider different sets of geometry as well. Further, the images can be "explorable," meaning that they could contain information aside from colors, such as scalar fields, or view normals so that coloring and lighting can be done interactively in a post hoc fashion. Several researchers have explored the idea of multi-image *in situ* rendering, but in this paper we refer to this use case as Cinema-style rendering, in reference to the Cinema software being developed following this paradigm. Cinema-style rendering opens the door for new approaches with sort-last parallel rendering. If R_i is the task of doing rendering for image i and C_i is the task of doing compositing for image i, then the only necessary constraint for parallel rendering is that R_i takes place before C_i for all i.

However, the traditional model imposes more constraints, namely a sequencing of $R_0, C_0, R_1, C_1, \ldots, R_N, C_N$. That said, sequencing for Cinema-style rendering can be different, such as $R_0, R_1, \ldots, R_N, C_0, C_1, \ldots, C_N$. Further, compositing phases can be combined to save on latencies, i.e., replacing $C_0, C_1, \ldots, C_N$ with one compositing algorithm (*MC*) that takes multiple images as input. With this work, we hypothesized that changing the order of rendering tasks could lead to significant savings in overall rendering times. To evaluate our hypothesis, we designed a study that compared several algorithms (i.e., orderings of R_i's and C_i's) over workloads that varied camera configurations, number of images to produce, amount of geometry and distribution, and concurrency. We found that reordering rendering phases for multi-image rendering was consistently better. Specific improvements varied heavily based on workload and algorithm, with many tests achieving an average speed-up of 2.3x, and others achieving as much as 10.0x speed-up and more. We believe the result of this study is impact-

*e-mail: larsen30@llnl.gov
†e-mail: kmorel@sandia.gov
‡e-mail: crj@sci.utah.edu
§email: hank@cs.uoregon.edu

ful and is applicable anytime multiple images are rendered. In the Cinema-style rendering paradigm, the simulation code will allocate finite time to visualization routines to generate its imagery. Therefore, rendering images more quickly can be beneficial either by allowing the *in situ* rendering routines to fit within the time budget or by allowing more images to be captured within the time budget, thus better preserving the state of the simulation.

2 Related Work

In this section, we discuss the related work of image-based *in situ* and image compositing.

2.1 Image-Based In Situ

An unsurprisingly common operation for *in situ* visualization is the rendering of images. However, one oft noted shortcoming of *in situ* rendering is that the rendering parameters must be established a priori, and interactive visualization is unavailable [5]. In response to this shortcoming, several researchers propose generating additional information, often in the form of multiple images, to recapture some of the exploratory process.

Chen et al. [4] note that if there is a finite space of camera position and visualization operations, then it may be feasible to precalculate all of the images before hand. Likewise, Kageyama and Yamada [8] propose creating a spherical shell of camera positions around the data object to be interactively viewed with a special movie player later. Tikhonova et al. [21] provide an alternate "explorable images" approach where images are saved with extra layers of information that provide some level of interactivity. However, interactions like camera movements are limited, so the explorable images technique must be combined with the other approaches using multiple images for a truly interactive experience.

Cinema [2, 18] is a generalization on this concept of generating multiple images to recapture interactivity. With Cinema, any visualization parameter can become explorable by providing multiple images rendered with different parameter values. Rendering multiple camera angles is typical, but other parameters such as isosurface value can be expressed as well.

Cinema provides the motivation for our work. When producing Cinema databases, *in situ* visualization generates many images, possibly hundreds, for a given time step. Our work sees an opportunity to optimize sort-last parallel rendering while generating many pre-defined images as in the case of a Cinema database.

2.2 Image Compositing

Sort-last parallel rendering has been the preferred method of rendering in distributed-memory environments because, unlike other parallel rendering modes, it scales very well with respect to the amount of geometry being rendered [22]. Consequently, it is utilized by industry standard visualization tools such as ParaView [1] and VisIt [6]. In this paradigm, data are distributed across ranks, and each rank is responsible for rendering the data it owns. The resulting images are then composited together into a single image of the entire data set using depth information. The sort-last approach naturally lends itself to *in situ* use cases since simulations discretize problems spatially, and sort-last rendering generally does not require redistribution.

Sort-last parallel rendering is an extremely well studied problem. This research generally focuses on the image compositing step. One of the earliest image compositing algorithms that is still commonly used today is binary swap [14]. Binary swap is a parallel divide and conquer approach that is simple to implement but efficient and load balanced. The compositing algorithm used for the work in this paper is radix-k [19]. Radix-k is a derivation of binary swap that achieves better efficiency by overlapping the data communication and the computation required for compositing. A more thorough review of sort-last parallel rendering algorithms, including other compositing algorithms, is given by Peterka and Ma [20].

Parallel image compositing can be further accelerated by incorporating optimizations such as active-pixel encoding and image interlacing. The details of the image compositing algorithm and the relevant optimizations employed are summarized by Moreland et al. [17]. The implementation is encapsulated in the IceT image compositing library [15], which is made publicly available. Finally, Grosset [7] et al. addressed load balancing issues in the rendering phase by scheduling compositing so that the data from the longest running ranks are only required at the latter stages of the reduction.

3 Algorithms for Multi-Image Sort-Last Parallel Rendering

We consider four algorithms. The first algorithm is the traditional approach that comes from applying sort-last parallel rendering repeatedly. However, since all desired camera positions and rendering configurations are known prior to Cinema-style rendering, it is possible to reorder the rendering and compositing steps. The final three algorithms take advantage of this property and also consider combined compositing steps. For each algorithm, we consider the equation governing the time complexity to generate n images. We use the following terms to define our equations:

- T is the total time to generate n images
- $R_{p,i}$ is the time for rank p to render its i^{th} sub-image ($1 \leq i \leq n$)
- C_i is the time to composite image i ($1 \leq i \leq n$), assuming that all ranks have completed rendering their inputs
- $C_{1:n}$ is the time to composite a single large image composed of n sub-images and to collect all n sub-images to the first rank element
- $C^*_{1:n}$ is the time to composite a single large image composed of n sub-images and to leave the n sub-images scattered across the ranks

3.1 Algorithm 1: Traditional Approach

The traditional approach is to render images one at a time, based on user guidance. This approach stems from the usage where camera positions are created on demand as the user manipulates the view to explore the data set, and the main concern is the ability to deliver high frame rates in the requested order. However, although this is the approach that has been used for multi-image rendering in the Cinema-style paradigm so far, we believe it may not be the most efficient approach. We include this algorithm in our study as a reference, so that we can compare it with algorithms designed for multi-image rendering. With the traditional approach, all ranks must render their images before compositing can begin. As a result, there is an implicit barrier between the two phases of the sort-last rendering process, and the total render time is limited by the slowest rank. Cores sit idle as they wait to proceed past the barrier, and this inefficiency is inherent to approaches where full interactivity is paramount. In the multi-image setting, these effects are exacerbated as an image database is generated. Equation 1 represents the time to generate n images with Algorithm 1:

$$T = \sum_{i=1}^{n} (max_p(R_{p,i}) + C_i) \tag{1}$$

3.2 Algorithm 2: Render-Then-Composite

This method generates n images by first rendering all sub-images and then compositing n images. The key difference is that each rank renders its images independently of other ranks, which reduces the number of implicit barriers from n to 1. Equation 2 represents the time to generate n images with Algorithm 2:

$$T = max_p(\sum_{i=1}^{n} R_{p,i}) + \sum_{i=1}^{n} C_i \quad (2)$$

In this model, rendering time is bound by the slowest rank, but it is the slowest rank to render all of its sub-images. Algorithm 2 contrasts with Algorithm 1, where the rendering time is bound by the slowest rank for each individual image, which can change from image to image.

3.3 Algorithm 3: Render-Then-Multi-Composite

Algorithm 3 extends Algorithm 2 by using a multi-composite step. That is, rather than applying a compositing algorithm n times to generate n images, the multi-composite combines the n images into one (very large) image, and then applies compositing on this single image. Our intuition suggests that compositing n images sequentially will add additional inefficiencies, and that compositing them as one large image will remove these inefficiencies by amortizing the overhead.

$$T = max_p(\sum_{i=1}^{n} R_{p,i}) + C_{1:n} \quad (3)$$

3.4 Algorithm 4: Render-Then-No-Gather-Multi-Composite

Parallel compositing hierarchically sub-divides computational work among all participating processors. When all computation is complete, contiguous portions of the final image exist on all processors, and the final image is assembled through a final gathering step. With multi-image rendering, the images are not immediately displayed, so this gather step is not needed. Instead, we only need to save the resulting images to a parallel file system. With this strategy, individual images can be left in place, and then each rank can save the images it owns to the parallel file system. The authors of IceT anticipated this use case and included necessary functionality inside the API. Further, the IceT implementation allows for the creation of "tiles" corresponding to whole images, meaning that images will not straddle ranks, and thus there is no need for additional communication between ranks. Equation 4 represents the time to generate n images with Algorithm 4:

$$T = max_p(\sum_{i=1}^{n} R_{p,i}) + C^*_{1:n} \quad (4)$$

4 Study Overview

Our study was designed to explore the opportunities for improvement within sort-last parallel rendering. In this section, we provide details on our software implementation, study factors, and architectures for testing.

4.1 Software Implementation

For our *in situ* infrastructure, we used Strawman [11], a lightweight library that comes integrated with three physics simulations. We used two out of the three simulations available: Lulesh [9], a hydrodynamics code using unstructured hexahedrons, and Kripke [10], a deterministic neutron transport code using a structured grid. We integrated Strawman with VTK-m [16], which is a many-core version of VTK targeting performance portability across heterogeneous architectures. We used VTK-m's existing isosurface and external faces filters, but we added a slice filter to support this study. For rendering, we used ray-tracing, since recent research has indicated that rendering with ray-tracing is faster than rasterization when BVH build costs are amortized [12]. Further, the ray-tracer in VTK-m has been demonstrated to be competitive with industry standards, specifically within a factor of two or better [16, 13]. Finally, we used IceT for image compositing, since it is widely used and has demonstrated good scaling at massive concurrency.

4.2 Study Phases and Factors

In our study, we varied the following factors:

- Algorithm (4 options)
- Camera configuration (3 options)
- Image counts (5 options)
- Image type (2 options)
- Rendering workload (4 options)
- Concurrency (9 options)

To manage the number of configuration combinations, we divided the study into two phases. In phase 1, we tested the cross-product of all testing options except concurrency, resulting in 480 tests. In phase 2, we held camera configuration, image count, and rendering workload constant, and varied concurrency, algorithm, and image type, resulting in 72 tests. Therefore, we conducted 552 total tests between both phases. The details of the factors are discussed in the following subsections.

4.2.1 Algorithm

We use the four algorithms described in Section 3. They are:

- Algorithm 1: Traditional approach (see Section 3.1)
- Algorithm 2: Render-then-composite (see Section 3.2)
- Algorithm 3: Render-then-multi-composite (see Section 3.3)
- Algorithm 4: Render-then-no-gather-multi-composite (see Section 3.4)

4.2.2 Camera Configurations

Camera configuration is closely tied to performance. If all camera positions share a focal point, then the rank that contains that focal point will likely do significant rendering work for each position, which limits speed-ups of Algorithms 2 to 4. Therefore, we wanted to consider multiple arrangements of camera positions. When considering this factor, we anticipate one finding will be that some choices of camera positions are not suitable for significant speed-up with multi-image sort-last parallel rendering, and that others are.

The three camera configurations we consider are:

- Cinema: The Cinema specification uses a spherical camera that assigns positions based on regular divisions of phi and theta. In our study, we chose to use a Fibonacci spiral, which allowed us to assign an arbitrary number of camera positions and avoid clustering the camera position around the poles. Camera distance and field-of-view were chosen to minimize the number of background pixels.
- Surface Reconstruction: Images with depth information can be treated as point clouds by surface reconstruction algorithms [3] in order to rebuild the original surface from the compressed representation. We reduced the camera field-of-view proportionally to the number of camera positions, which increases the resolution of the point cloud and reduces redundant information by minimizing image overlap. Further, this camera setup exposes greater rendering work imbalances (i.e., fewer ranks will have any work for a single frame), which will allows us to better explore the space of all possible configurations.
- Inverse Cinema: This configuration places the camera at the sphere center, and the view direction points towards locations on the surface of the sphere. These directions correspond to the same points as the Cinema configuration camera positions. The inverse Cinema configuration allows the user to move the camera view direction instead of the position.

4.2.3 Image Counts

Image count refers to the number of images rendered. We consider 5 image counts in our study, ranging from 20 to 100 images in increments of 20. We consider this factor because we hypothesize that larger image counts create opportunities for larger savings compared to the traditional technique.

4.2.4 Image Type

Images can be either Static or Explorable. Static images contain only color data, and so interactivity is limited to changing the camera position. Explorable images contain different types of data, for example scalars and view normals, which allows for changing color maps and lighting parameters when doing post hoc exploration in a database viewer.

We consider both image types in our study:

- Static: represented with an RGBA format using 32 bits for each pixel, which reduces compositing time due to less communication overhead.
- Explorable: represented with 128 bits for each pixel, which increases compositing time due to additional communication overhead.

A resolution of 1024^2 was used for both image types.

4.2.5 Rendering Workload

In a distributed memory simulation using tightly-coupled *in situ*, the rendering workload per rank varies depending on the current time step and the type of visualization operation that generates the geometry. On the one hand, the rendering workload could be completely balanced (i.e., each rank contains the same amount of geometry), and, on the other hand, the rendering workload could be highly imbalanced (i.e., only a subset of ranks have any geometry). The set of all possible workloads falls somewhere in between these two cases, and to explore this space, we used four workloads that are representative over the spectrum:

- Balanced: each rank renders the external faces of the rank's domain.
- Imbalanced: only the external faces of the entire data set are rendered.
- Slice: the entire data set is sliced along the xy, xz, and yz axis planes
- Isosurface: generated from an isovalue with approximately 65% of the ranks with rendering work.

The geometry for the balanced, imbalanced, and slice workloads is created from Lulesh, which is run on 512 ranks with a total data set size of 2048^3, a total of 8.5 billion elements, and each rank contains 256^3 unstructured hexahedrons. The balanced workload renders 393,216 faces per rank (i.e., 768K triangles), and the imbalanced workload renders a total of 25.1 million faces (i.e., 50.2M triangle) spread across 224 ranks. The geometry for the isosurface was created using Kripke running on 512 ranks with a total data set size of 1024^3 and resulted in 73M total triangles.

4.2.6 Concurrency

Since IceT does not use hybrid-parallelism, we ran our rendering and visualization using one MPI rank per core. For phase 1, we used a total of 512 MPI ranks. For phase 2, we ran 9 options, ranging between 512 (8^3) and 4096 (16^3) MPI ranks. We were only able to run with MPI rank counts that were perfect cubes, due to constraints from Lulesh.

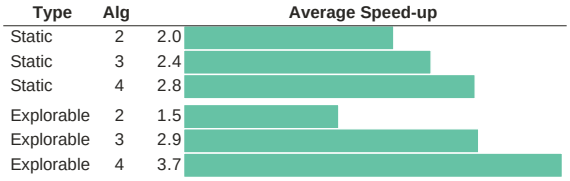

Figure 1: The speed-ups of each algorithm averaged over all image counts and camera configurations.

4.3 Hardware Architecture

Phase 1 and phase 2 of our study used different machines, due to machine availability; we had access to a small machine where we could run many jobs (phase 1) and a large machine where we could run few jobs (phase 2). In terms of hardware specifics:

- Phase 1 ran on Surface, a visualization cluster at Lawrence Livermore National Laboratory. Each node has two Intel Xeon E5-2670 running at 2.6 GHz with 16 total cores and 256 GB of memory per node.
- Phase 2 ran on Cab, a large capacity machine also at Lawrence Livermore National Laboratory. Each node has same configuration as Surface but with 32 GB of memory per node.

5 Results

In this section we discuss the results of the two phases of our study.

5.1 Phase 1

For the first phase of the study, we hold concurrency constant and vary all other study options. We begin by presenting the overall speed-up of Algorithms 2 to 4 compared to the traditional approach (Section 5.1.1). Then, we analyze the individual components of the speed-ups of sort-last parallel rendering — rendering (Section 5.1.2) and compositing (Section 5.1.3) — to explain the reasons behind the overall speed-ups.

5.1.1 Overall Speedups

For each of the image formats, Figure 1 shows the average speed-ups over Algorithm 1 for all camera configurations and image counts, and Figure 2 shows the overall speed-ups for all factors of the study configuration when rendering 100 images. For brevity, we omitted the other image counts from the figure since those results are presented in detail when we analyze the individual components. Each algorithm incrementally introduces new improvements over the traditional method. Algorithm 2 renders all images at once to reduce rendering imbalances, while the compositing stage is identical to the traditional method. Algorithm 3 improves compositing efficiency by treating all renderings as a single image, and Algorithm 4 eliminates the final gathering of images to a single process. In almost all cases, each successive algorithm improved the speed-up over the previous one.

Overall, the speed-ups over the traditional methods ranged between 1.0x (i.e., no improvement) and 10.2x (i.e., ran more than 10 times faster), with the Imbalanced workload providing the most significant improvements, and on average, the speed-ups ranged between 1.5x and 3.7x. The Cinema camera configuration varied between 1.4x and 1.9x with the Static image type, and it varied between 1.0x and 3.5x with the Explorable image type. The Inverse Cinema and Surface Reconstruction camera configurations

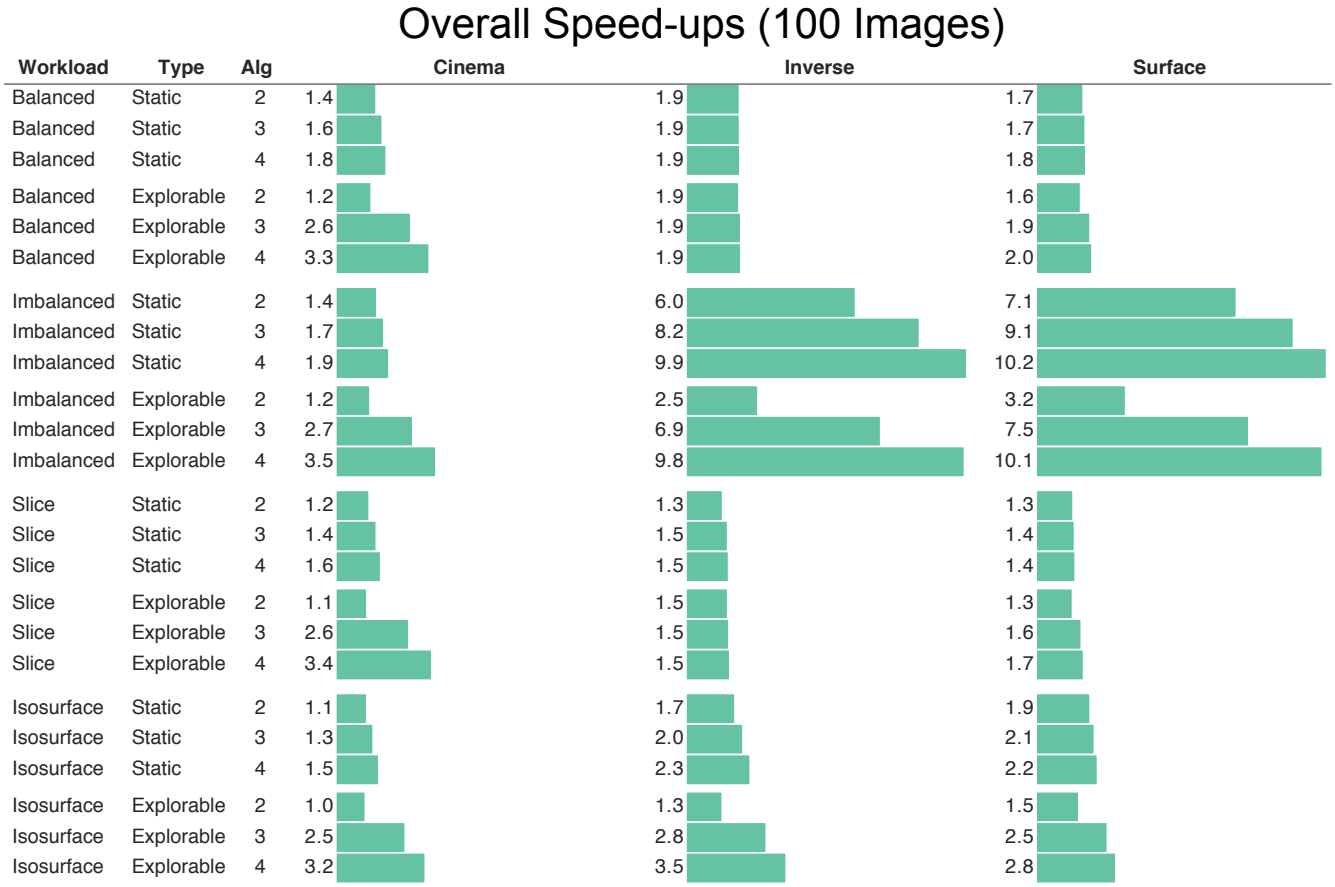

Figure 2: Summary of the overall speed-ups compared to Algorithm 1 for all configurations at 100 images. As an example of how to interpret the figure, the entry of 1.4 in the first row means that balanced workload with the Static image type and the camera positions defined by Cinema is 1.4x faster with Algorithm 2 than it is with Algorithm 1.

Ratio of Compositing Over Rendering

Wrkld	Type	Algo	Cinema	Inverse	Surface
Iso	S	1	0.52	0.25	0.11
		2	0.59	0.46	0.20
		3	0.32	0.25	0.11
		4	0.15	0.12	0.05
	E	1	2.36	1.17	0.53
		2	2.60	2.21	1.03
		3	0.53	0.46	0.21
		4	0.19	0.17	0.09

Table 1: An example of the impact of on the ratio of time spent compositing over rendering with the isosurface workload at 100 images. Values greater than 1.0 means that more time is spent compositing, and values less than 1.0 mean that more time is spent rendering. For image type, **S** denotes the Static and **E** denotes the Explorable.

had speed-ups ranging between 1.3x and 10.2x. Excluding the Imbalanced workload, which had the largest speed-ups, both configurations had speed-ups between 1.3x and 3.5x.

The overall speed-ups in the study varied based on the relative speed-ups of each phase. Table 1 provides an example of the ratio of time spent rendering and compositing for the Isosurface workload. Compositing times for the Static image representation were approximately four times faster than for the Explorable image type, due to shear data size. Within each image data representation, compositing times for the different algorithms generally stayed relatively uniform, although there were variations due to different rendering workloads. For the Explorable image type, rendering times tended to be equal to compositing times for Algorithms 1 and 2, but rendering began to more heavily influence overall times as compositing times became faster with Algorithms 3 and 4. For the Static image type, rendering times always played a more important role in overall time.

5.1.2 Rendering

The speed-ups across all workloads and camera configurations are summarized in Figure 4. The speed-ups range between 1.1x and 9.0x. Note that Algorithms 2 to 4 operate identically for the rendering phase, so they are reported jointly. Figure 3 plots the *max* terms in the equations for Algorithm 1 and Algorithms 2 through 4. In this section, we discuss the speed-ups of the various workloads and the effects of the different camera configurations.

To simplify discussion, we define two terms:

- LR_1: the set of limiting ranks for Algorithm 1, which are ranks that have the highest rendering workload for each frame.
- LR_2: the single limiting rank for Algorithms 2 to 4, which is the rank that has the highest rendering workload over all frames.

Balanced The speed-ups for the Balanced workload ranged between 1.6x and 1.9x, and remained unchanged as the number of

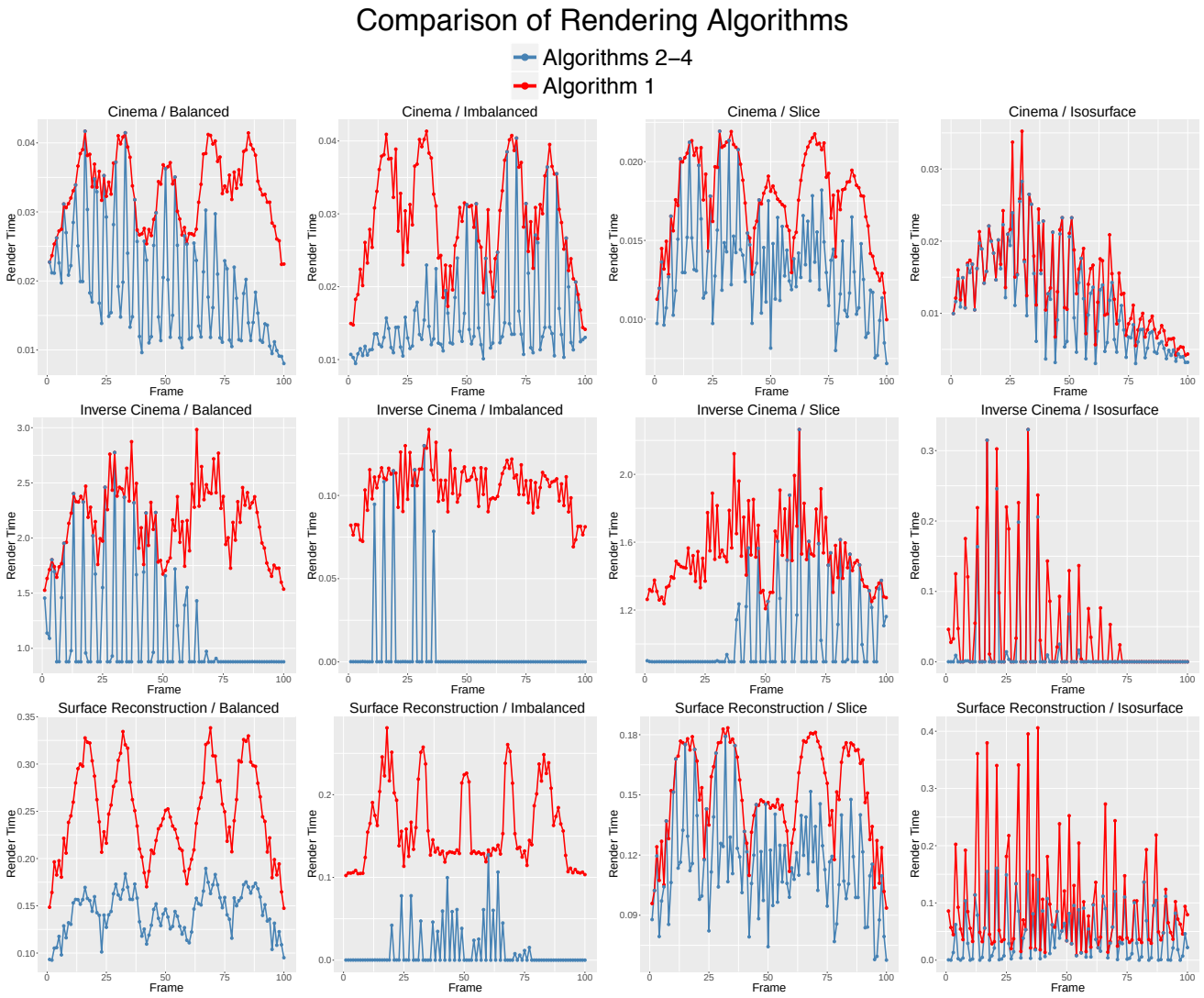

Figure 3: A comparison of rendering times between Algorithm 1 and Algorithms 2-4 described in section. The y-axis is rendering time, where lower is better, and the x-axis is frame number. In the above figures, the red line is the slowest render time for each frame (Algorithm 1 rendering time), and the slowest rank can change from frame to frame. The blue line is the slowest rank over all frames (Algorithms 2-4 rendering times), and each data point is that rank's render time. From top to bottom, the rows represent the different camera configurations of Cinema, Inverse Cinema, and Surface Reconstruction. From left to right, the columns represent the Balanced, Imbalanced, Slice, and Isosurface rendering workloads. The difference in areas under the two curves is the maximum potential difference between the traditional approach and ours with respect to rendering. Further, the ratio of the two areas indicates the relative speed-up.

images increased. The effect of each camera configuration can be seen in the first column of Figure 3.

In the Cinema configuration, each rank contains an equal amount of geometry, and ranks directly in front of the camera have the largest amount of work. For any given image, LR_1 is always directly in front of the camera. LR_2 is only sometimes in front of the camera, since the camera could be focused on the other side of the data set. Because LR_2 has a much lower rendering workload while not in front of the camera, on average we observed a 1.6x speed-up.

The Inverse Cinema configuration had the largest speed-up of the three configurations at 1.9x with the Balanced rendering workload. Here the camera is at the center of the data set looking out, and the limiting ranks are at the center of the data set in front of the camera. Of the four ranks surrounding the center point of the data set, each is in front of the camera about a quarter of the time. Our method averaged out the work imbalance and was able to produce almost twice the number of images than did Algorithm 1 in the same amount of time.

The Surface Reconstruction camera configuration achieved a 1.7x speed-up over Algorithm 1. With a tight camera field of view, only ranks directly in front of the camera, at the center of the data set, and on the far side of the data set had any rendering work for a given camera position. LR_1 was always on the surface of the data set. LR_2 was located at the center of the data set, and it had consistent work over all camera positions. Thus LR_1 and LR_2 are never the same for Surface Reconstruction, in contrast to the other camera configurations.

Imbalanced The Imbalanced workload offered the largest opportunity for rendering time improvements and the speed-ups ranged from 1.6x to 9.0x. The summary of the limiting factors is found in the second column of Figure 3.

The Cinema camera configuration showed the lowest speed-ups

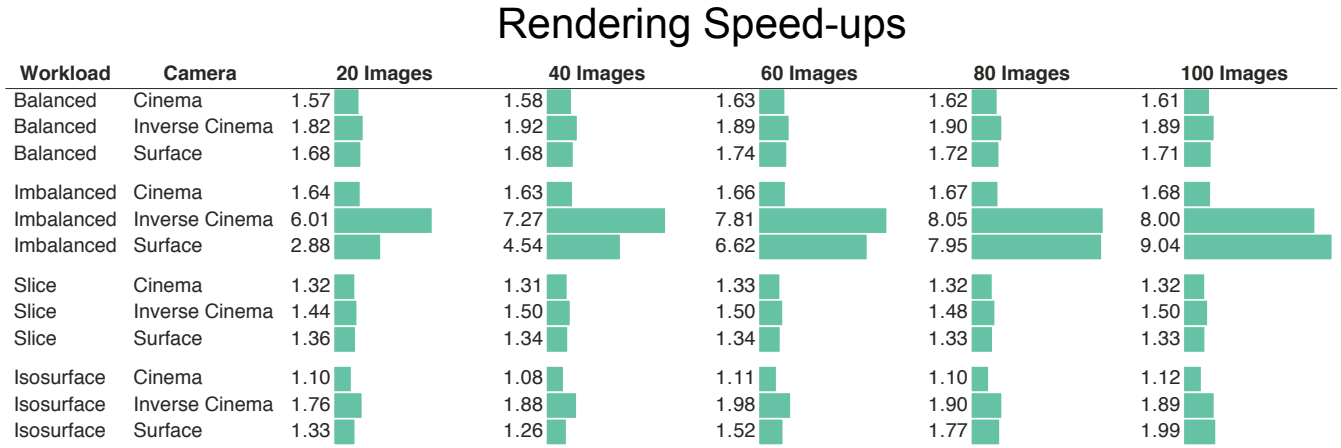

Figure 4: The rendering speed-ups for Algorithms 2-4 compared to Algorithm 1 for each image count, workload, and camera configuration. The rendering times for each of Algorithm 2, 3, and 4 are identical, so they are reported as a single number for each test.

at around 1.6x. Since the entire data set is in view of the camera from any position, all ranks with geometry always performed some amount of work, and the opportunity to improve on Algorithm 1 is limited. Similar to the Balanced workload, LR_1 was directly in front of the camera. The speed-up did improve as the image count increased, but the increase was negligible for the image counts in our study.

The Inverse Cinema and Surface Reconstruction camera configurations achieved speed-ups over 8x. For each image, only a portion of the data set is within the view of the camera, which enabled large speed-ups over Algorithm 1. Further, the observed speed-up steadily increased as the image counts increased. The Surface Reconstruction configuration amplified the work imbalance since the field of view narrowed as the image count increased. Consequently, the speed-up was proportional to the image count.

Slice The Slice workload achieved the most modest speed-up of the four workloads, ranging from 1.3x to 1.5x. The summary of the limiting factors is in the third column of Figure 3.

The slice along the xy, xz, and zy axis planes produces regular amounts of geometry on both the inside and outside of the data set. All three camera configurations exhibited similar limiting ranks that alternated between ranks at the center and on the outside. Additionally, the Cinema and Surface Reconstruction configuration bottleneck produced near identical plots in Figure 3 that resulted in similar achievable speed-ups.

Isosurface The Isosurface workload achieved speed-ups between 1.1x and 2.0x. The summary of the limiting factors is in the fourth column of Figure 3.

The achievable speed-up of this workload was heavily impacted by the triangle distribution and camera configuration. The isosurface contained approximately 75M triangles, and 65% of all ranks contained geometry. The majority of ranks had between 10K and 250K triangles. However, eight ranks had between 500K and 1.9M triangles, and these ranks dominated rendering time over most images.

In the Cinema camera configuration, the two curves in Figure 3 nearly mirror each other, meaning the longest running rank and the per image maximum were mostly the same rank. That is, the ranks with 1.9M triangles were always in view of the camera and were almost always the bottleneck, which limited the speed-up to 1.1x. With the Inverse Cinema configuration, the ranks with 1.9M triangles were only within view of the camera for a few frames, and the work imbalance was able to average out, resulting in a speed-up of 2.0x. The Surface Reconstruction camera configuration was limited by a rank with a smaller amount of triangles at the center of the data set, and that rank was almost always within the view of the camera.

The speed-ups of both the Inverse Cinema and Surface Reconstruction increased as the number of images increased. This increase indicates that much higher speed-ups are possible, and that the observed speed-ups were limited by the particular data set and isovalue value.

5.1.3 Compositing

The speed-ups for the different camera configurations of the study are summarized in Figures 5, 6, and 7. In the figures, we included only Algorithms 3 and 4, since Algorithm 2 improves only rendering performance. Overall, we observed speed-ups of 1.3x to 22.2x, depending on the exact configuration. Among the different configurations, the speed-ups naturally varied the greatest between the image data sizes and image compositing algorithm, although the rendering workload indirectly had an impact on compositing time.

IceT performs image compression to minimize the communication and processing overhead during compositing. When images are sent to IceT, the depth buffer is scanned to determine exactly how many pixels each rank contains, and images are compressed using run-length encoding. The rendering workload and camera configuration determine the exact amount of compositing work, and these differences produced minor variations in the speed-ups of our method in the compositing stage.

Other than the reduced communication overhead, several other factors contributed to the increased compositing efficiency of a single large image. When compositing a single image, it is more likely to have a pixel imbalance across ranks, leaving many portions of the reduction tree without work, but when compositing many images simultaneously, it is much more likely to have better distribution of work at the base of the tree, which increases compositing efficiency. For Algorithm 4, the final gather of the image fragments is omitted because we have no need to display the image immediately, and we just write out the pieces to the parallel file system. As concurrency gets larger, the overhead of this step can quickly become the bottleneck of sort-last parallel rendering.

Static Images The Static image type contains only color information, which is 32 Bytes per pixel. While the resulting image database is explorable (i.e., changing camera position by loading image-after-image), color maps and lighting values are static. As

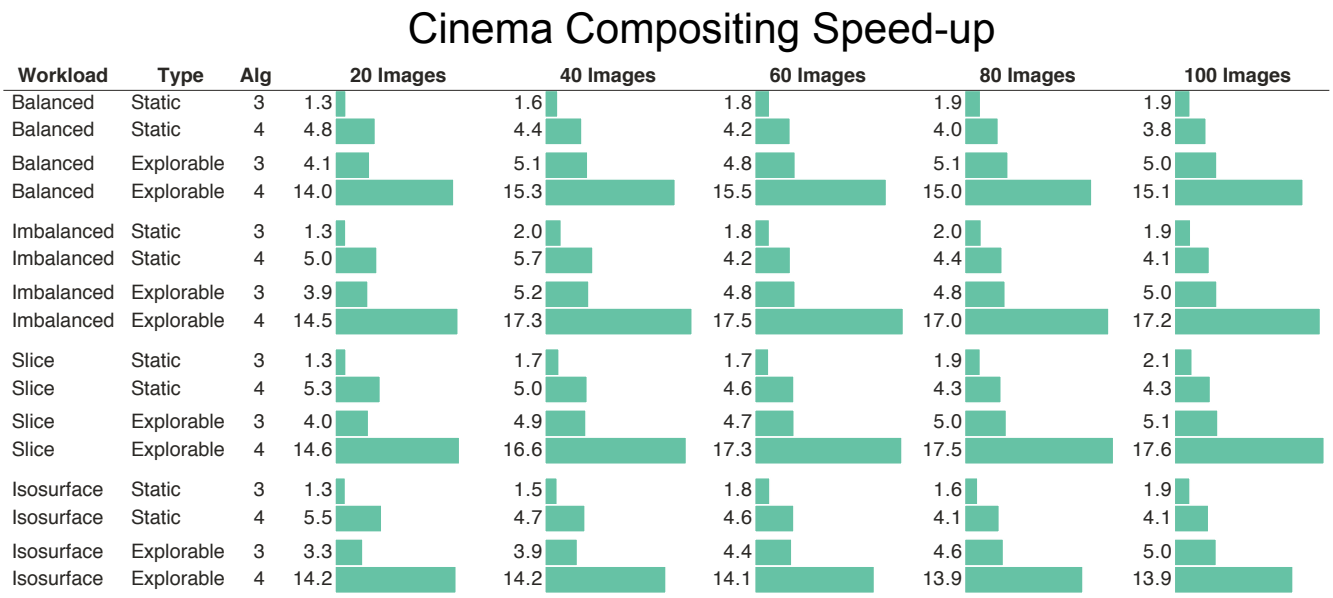

Figure 5: Summary of the compositing speed-up with the Cinema camera configuration.

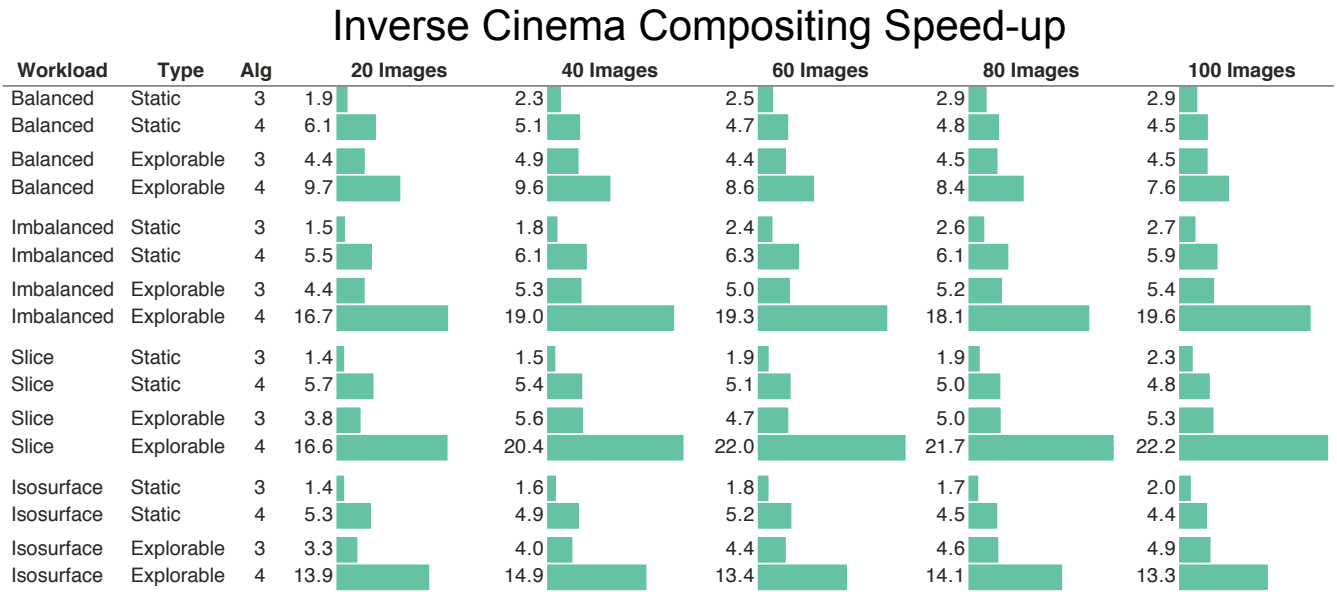

Figure 6: Summary of the compositing speed-up with the Inverse Cinema camera configuration.

a result, there is less data movement and communication overhead, which in turn leads to modest compositing speed-ups.

For Algorithm 3, the Static image type achieved speed-ups of 1.3x to 2.9x. In all cases, the speed-up increased as the number of images became higher, so we can achieve even greater image throughput at higher image counts. Conversely, Algorithm 4 experienced the greatest speed-ups at lower image counts, although the Surface Reconstruction and Imbalanced workload remained relatively flat for all image counts. With Algorithm 4, the speed-ups ranged between 3.8x and 6.1x.

Explorable Images The Explorable image type produces a database that contains scalar field and lighting values, which enables users to dynamically change color maps and lighting during exploration. Since the Explorable image type contains more data, improvements in efficiency lead to greater speed-ups over Algorithm 1. For Algorithm 3, the speed-ups ranged between 3.9x and 5.3x, and for Algorithm 4, the speed-ups varied between 5.6x and 22.2x.

5.2 Phase 2

For the second phase of our study, we held camera configuration, rendering workload, and image count constant, and varied all other study options. We ran the Cinema camera configuration, Imbalanced rendering workload, and both the Static and Explorable image types. Additionally, we reduced the per MPI rank simulation domain size to 32^3 zones per node, and we ran the study on Cab,

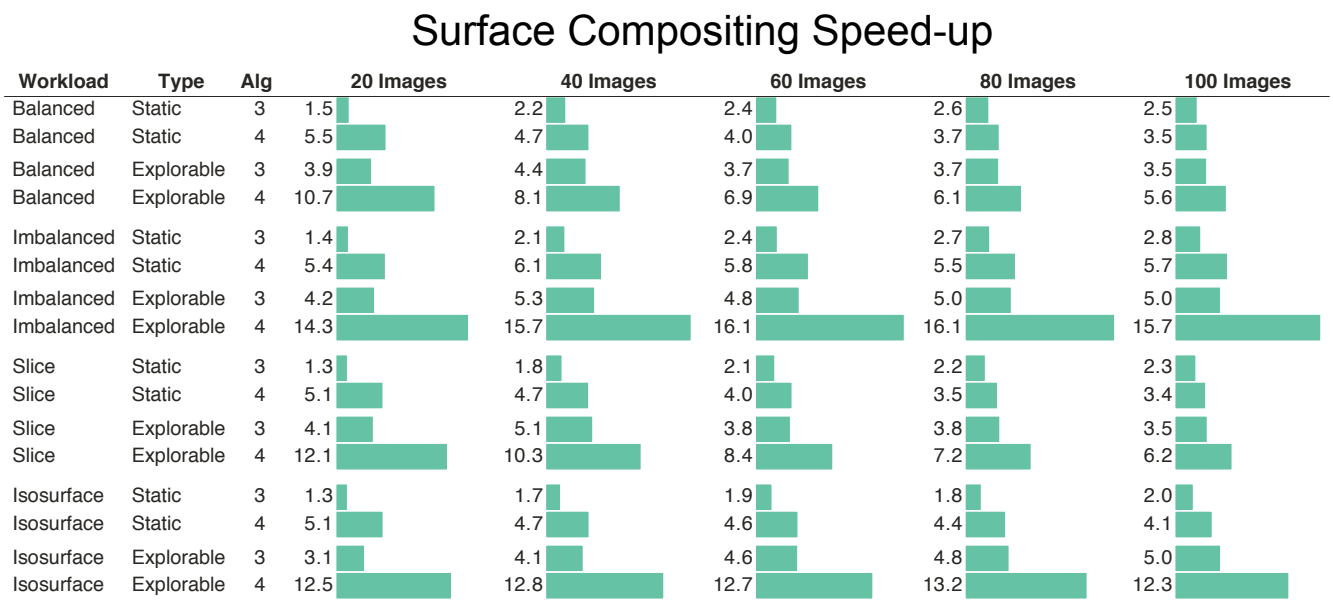

Figure 7: Summary of the compositing speed-up with the Surface Reconstruction camera configuration.

	MPI Ranks				
A/I	512	1000	1728	3375	4096
1/S	1.0 (1.2)	1.0 (1.5)	1.0 (3.7)	1.0 (15.5)	1.0 (21.3)
3/S	2.3 (0.4)	2.5 (0.4)	4.2 (0.5)	13.1 (0.6)	16.8 (0.6)
4/S	2.6 (0.2)	2.8 (0.2)	4.7 (0.3)	15.3 (0.4)	11.8 (0.6)
1/E	1.0 (4.4)	1.0 (3.9)	1.0 (3.3)	1.0 (11.4)	1.0 (14.2)
3/E	4.1 (0.7)	3.4 (0.8)	2.8 (0.8)	6.9 (1.1)	7.8 (1.0)
4/E	5.6 (0.3)	4.6 (0.3)	3.8 (0.3)	9.4 (0.5)	11.5 (0.7)

Table 2: Scaling of rendering and compositing using the Cinema camera configuration, 40 images, and the Imbalanced rendering workload using 32^3 zones per rank. **A** denotes algorithm and **I** denotes image type (**S** for Static and **E** for Explorable). The upper portion of the table contains the Static image type and the bottom portion contains the Explorable image type. Each entry shows the speed-up over Algorithm 1; in parentheses is the ratio of compositing time over rendering time. Algorithm 2 was omitted from the table since it did not provide any additional information.

which has more nodes than the Surface machine.

Table 2 summarizes the results of the second phase. Algorithm 2 did not show significant improvements for this configuration, so we omitted the results from the table. The tables show the speed-up over Algorithm 1; in parentheses is the ratio of the time spent compositing and rendering. As concurrency increased, compositing time quickly dominated rendering time for Algorithm 1. The final compositing times for both the Static and Explorable image types are over 9.0s, but for Algorithms 3 and 4, the maximum compositing time never exceeds 0.65s for either image type. Overall,the speed-ups increased with concurrency. At 512 MPI ranks, the speed-ups ranged between 2.3x and 5.6x, and at 4096 MPI ranks, they varied between 7.8x and 16.8x.

Communication overhead increases as the number of MPI ranks becomes larger. Within compositing, the communication cost is composed of two stages: round communication and a final gather. Round communication occurs during the intermediate stages of compositing, where pieces of the images are exchanged between processors, and the number of rounds is proportional to the number of participating ranks. After the rounds are complete, the final image is distributed in pieces throughout the participating MPI ranks. Finally, the pieces of the complete image are gathered to a single rank and reassembled.

Algorithm 1 performs both stages for every image. Algorithm 3 performs compositing in a single step for all images, after which it performs a single gather. Finally, algorithm 4 omits the final gather step of algorithms 1 and 3. Thus, the overall communication cost for algorithm 1 is much higher than algorithms 3 and 4. Consequently, the compositing growth rate of algorithm 1 is significantly higher than algorithms 3 and 4.

Rendering times trend in the opposite direction. With the balanced workload, the amount of geometry per rank stays constant, but as concurrency increases, there is a decrease in the number of pixels each rank contributes to the final image. That is, at higher concurrency, ranks have less rendering work. Consequently, the compositing cost grows faster than the rendering cost.

5.3 Memory Usage Trade-Off

With *in situ* visualization, memory is a constrained resource, and the memory trade-off between algorithm 1 and algorithms 2-4 is straight-forward. A single Static image consumes 4MB of memory at a resolution of 1024^2, and at the same resolution, an Explorable image consumes 16MB, i.e. four times the amount of the Static image. If we renderer 10 images, then algorithm 1 only uses the memory of a single image, but algorithms 2-4 consume 10x more memory. Using one MPI rank per node, the memory requirements of algorithms 2-4 would likely be acceptable, given the performance gains they offer, but using one MPI rank per core, the memory requirements would likely be unreasonable.

Given a memory budget, image resolution, and image format, it is simple to calculate the memory usage. If a simulation can accommodate that memory budget, then the algorithms presented in this work can, on average, either render 1.5 to 3.7 times faster, giving the simulation more time to advance, or render 1.5 to 3.7 more images, creating a larger image database. Ultimately, users will have to make these trade-offs based on what is appropriate for their use case.

6 Conclusion

We have described a straight-forward method that increases the efficiency of multi-image sort-last parallel rendering over the traditional approach. We found that our new algorithms were helpful in all cases, with average performance increases ranging from 1.5x to 3.7x. In some configurations, the speed-ups were more than 10x, which in turn would allow for Cinema-style rendering to either cause less burden on the simulation code, or to save 10x more images. In other configurations, the effects were more modest. Further, we believe our results are applicable anytime multiple images are rendered, not just in a Cinema-style use case. In terms of future work, we believe more study should be placed on how to choose rendering workloads that deliver consistently good results (i.e., more 10x speed-ups and less 1.5x speed-ups). Camera configuration will be an important factor in achieving this goal. We believe that modifying the camera layouts to de-emphasize the center portion of the data set would significantly reduce bottlenecks. However, user studies are needed to understand which camera configurations would be acceptable to end users, which we felt was out of scope for this study. Another direction of future work is to overlap rendering and compositing. We decided not to pursue this strategy since we felt that a key finding of this study is that a simple change in practice can make a significant difference.

Acknowledgements

Some of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was supported in part by the The SciDAC Institute of Scalable Data Management, Analysis and Visualization, funded by the DOE Office of Science through the Office of Advanced Scientific Computing Research. Hank Childs is grateful for support from the DOE Early Career Award, Contract No. DE-SC0010652, Program Manager Lucy Nowell.

References

[1] J. Ahrens, B. Geveci, and C. Law. Paraview: An end-user tool for large data visualization. In *Visualization Handbook*. Elesvier, 2005. ISBN 978-0123875822.

[2] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 424–434. IEEE Press, 2014.

[3] M. Berger, A. Tagliasacchi, L. Seversky, P. Alliez, J. Levine, A. Sharf, and C. Silva. State of the art in surface reconstruction from point clouds. In *EUROGRAPHICS star reports*, volume 1, pages 161–185, 2014.

[4] J. Chen, I. Yoon, and W. Bethel. Interactive, internet delivery of visualization via structured prerendered multiresolution imagery. *IEEE Transactions on Visualization and Computer Graphics*, 14(2):302–312, 2008.

[5] 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.

[6] H. Childs et al. A contract based system for large data visualization. In *VIS 05. IEEE Visualization, 2005.*, pages 191–198. IEEE, 2005.

[7] A. V. P. Grosset, A. Knoll, and C. Hansen. Dynamically Scheduled Region-Based Image Compositing. In E. Gobbetti and W. Bethel, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2016.

[8] A. Kageyama and T. Yamada. An approach to exascale visualization: Interactive viewing of in-situ visualization. *Computer Physics Communications*, 185(1):79–85, 2014.

[9] I. Karlin, J. McGraw, J. Keasler, and B. Still. Tuning the lulesh miniapp for current and future hardware. In *Nuclear Explosive Code Development Conference Proceedings (NECDC12)*, 2012.

[10] A. Kunen, T. Bailey, and P. Brown. Kripke-a massively parallel transport mini-app. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2015.

[11] M. Larsen, E. Brugger, H. Childs, J. Eliot, K. Griffin, and C. Harrison. Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 30–35. ACM, 2015.

[12] M. Larsen, C. Harrison, J. Kress, D. Pugmire, J. Meredith, and H. Childs. Performance Modeling of In Situ Rendering. In *The International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, Nov. 2016. (to appear).

[13] M. Larsen, J. S. Meredith, P. A. Navrátil, and H. Childs. Ray tracing within a data parallel framework. In *2015 IEEE Pacific Visualization Symposium (PacificVis)*, pages 279–286. IEEE, 2015.

[14] 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.

[15] K. Moreland. IceT users' guide and reference, version 2.1. Technical Report SAND2011-5011, Sandia National Laboratories, August 2011.

[16] K. Moreland et al. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications*, 36(3):48–58, 2016.

[17] 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*, page 25. ACM, 2011.

[18] P. O'Leary, J. Ahrens, S. Jourdain, S. Wittenburg, D. H. Rogers, and M. Petersen. Cinema image-based in situ analysis and visualization of MPAS-ocean simulations. *Parallel Computing*, 55:43–48, 2016.

[19] 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.

[20] T. Peterka and K.-L. Ma. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, chapter Parallel Image Compositing Methods, pages 71–89. CRC Press/Francis–Taylor Group, 2012. ISBN 978-1439875728.

[21] A. Tikhonova, C. D. Correa, and K.-L. Ma. Visualization by proxy: A novel framework for deferred interaction with volume data. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1551–1559, November/December 2010. DOI 10.1109/TVCG.2010.215.

[22] 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.