A Scalable Hybrid Scheme for Ray-Casting of Unstructured Volume Data

Roba Binyahib, Tom Peterka, Matthew Larsen, Kwan-Liu Ma, and Hank Childs

Abstract—We present an algorithm for parallel volume rendering that is a hybrid between classical object order and image order techniques. The algorithm operates on unstructured grids (and structured ones), and thus can deal with block boundaries interleaving in complex ways. It also deals effectively with cases that are prone to load imbalance, i.e., cases where cell sizes differ dramatically, either because of the nature of the input data, or because of the effects of the camera transformation. The algorithm divides work over resources such that each phase of its processing is bounded in the amount of computation it can perform. We demonstrate its efficacy through a series of studies, varying over camera position, data set size, transfer function, image size, and processor count. At its biggest, our experiments scaled up to 8,192 processors and operated on data sets with more than one billion cells. In total, we find that our hybrid algorithm performs well in all cases. This is because our algorithm naturally adapts its computation based on workload, and can operate like either an object order technique or an image order technique in scenarios where those techniques are efficient.

Index Terms—Volume rendering, parallel visualization, large scale visualization.

1 Introduction

Computational power has been increasing tremendously in recent years, allowing scientists to simulate larger time-varying 3D physical phenomena at higher resolutions, leading to larger data sets to analyze and visualize. Volume rendering is an important method for visualizing such data, as it provides insight into the entire data set. Application areas that regularly make use of volume rendering include astrophysics, biomedical, combustion, and nuclear reactor simulations, among others. However, when data sets become so large that they can not fit in the memory of a single computer, serial volume rendering techniques are insufficient. Parallel volume rendering is frequently used to visualize these data sets. The two most studied approaches for parallel volume rendering are object order (parallelization over input data, also known as sort-last) and image order (parallelization over output pixels, also known as sort-first). While both of these approaches have been used successfully, they can become highly inefficient for some volume rendering workloads, particularly when the cell sizes in the mesh vary greatly or when the camera position emphasizes some regions of the scene over others. In these cases, performance can decrease significantly, which has negative ramifications on both post hoc and in situ processing.

In 2006, Childs et al. [1] devised a parallel volume rendering approach for unstructured meshes that was a hybrid method between object-order and image-order techniques. While their approach was novel and interesting, the work was lacking in three major respects: (1) their evaluation was insufficient to demonstrate the necessity of their technique, (2) the technique required unnecessarily large amounts of memory to store intermediate results, which affected both timing information and led to crashes, and (3) no consideration was taken for many-core architectures. With our work, we address these three shortcomings. Specifically, we perform an evaluation that considers multiple camera positions, data sets, and scales. The resulting cross-product of workloads clearly demonstrates the benefit of the hybrid approach in a way that was not done in their original work. We also introduce the concept of partial composites to their original algorithm as a way to reduce memory footprint. Finally, we add support for many-core architectures. This support is important since many-core processors are already widely deployed on current supercomputers, and it is expected to become even more dominant in the future. Further, the balance of computing and communication changes in the context of many-core computing, requiring new evaluation in this context. This study contains this new evaluation. Overall, the results presented in this work demonstrate that the hybrid approach (i.e., between object-order and image-order) outperforms traditional approaches in most cases. Specifically, it is superior on workloads that have previously led to load imbalance conditions, and equally good for the remaining workloads.

- *R. Binyahib and H. Childs are with the University of Oregon E-mail: roba, hank@cs.uoregon.edu*
- *T. Peterka is with Argonne National Laboratory E-mail: tpeterka@mcs.anl.gov*
- *M. Larsen is with Lawrence Livermore National Laboratory E-mail: larsen30@llnl.gov*
- *K.-L. Ma is with the University of California at Davis E-mail: ma@cs.ucdavis.edu*

Manuscript received September 7, 2017; revised January 19, 2018.

2 Background and Related Work

2.1 Distributed-Memory Parallel Volume Rendering Techniques

For both image order and object order distributed-memory parallel volume rendering, the main idea is to distribute work among different processors, perform some operations in parallel, and then gather results into one image. We organize our discussion of related work on techniques around

these two approaches. We also discuss image compositing, as it is an important step for the object order approach.

2.1.1 Image Order

With the image order approach, pixels are partitioned among processors. The partitioning creates groups of consecutive pixels, also known as tiles. Each processor begins by loading the cells that contribute to its tile. Then each processor generates the portion of the image corresponding to its tile, by operating on the cells that it loaded. The portion of the image produced by each processor is also known as a sub-image. The sub-images from all the processors are then collected onto one processor to produce the final image. The image order technique has two main disadvantages with respect to distributed-memory parallelism:

- Image order does not guarantee equal cell distribution among processors. When one tile contains more cells than the other tiles, its corresponding processor performs more work, resulting in load imbalance.
- Cells can cover more than one tile of the image, resulting in replicated loading of these cells.

One interesting use of the image order approach was for multi-projector rendering systems [2]. In this work, the authors rendered the volume onto multiple screens. Each processor was responsible for a projector. With this approach, a problem could occur when the data assigned to one screen contained a large number of cells, resulting in one processor performing more work than the others. To solve this problem, the authors distributed virtual tiles that could be of any shape or size. The distribution of these virtual tiles depended on the size and complexity of the data contributing to the tile. This allowed processors to sample and render their virtual tiles while maintaining load balance. Erol et al. [3] presented a dynamic load balancing approach based on cross-segment load balancing. Their solution divided the workload into segments and used previous rendering times to assign more processors to segments that had higher workloads.

2.1.2 Object Order

With the object order approach, data is divided into blocks among processors. Each processor volume renders its own cells independently of the other processors. Then the contributions from all the processors are composited together to produce a final image.

There are multiple approaches for object order volume rendering. With our approach, we assume the data is partitioned by the simulation code, and all data is resident in main memory. These assumptions are often necessary for in situ processing. However, other approaches consider repartitioning of the data to improve load balance and also do not require all data to be resident in main memory. These latter approaches enable techniques such as data compression [4], multi-resolution, and out of core processing [5].

With the type of object order processing we consider (no repartitioning and all data in main memory), there are two main disadvantages with respect to distributed-memory parallelism:

- Data exchange between processors (e.g., image compositing) can become a bottleneck [6].
- Uneven work distribution might occur. For example, when dealing with unstructured data, one processor can own larger cells than the others, which can mean more work. Another example is when the camera view focuses on a region of the data that is owned by one processor. Both of these scenarios result in an unequal amount of work (load imbalance).

Several object order solutions have been proposed. Marchesin et al. [7] presented a solution to guarantee load balance. They divided data into blocks and performed an estimation step of the rendering cost. The transfer function and camera view were used to discard any blocks that were out of the camera view or that had no opacity. The remaining blocks were then distributed among processors, and each node rendered its data. Finally, binary swap [8] was used to composite the final image. Strengert et al. [9] used hierarchical compression for their texture-based volume rendering. They incorporated a wavelet compression technique by Guthe et al. [10] to reduce the size of the data blocks.

Ma [11] used a graph-based data-partitioning software package to achieve equal sub-volume distribution among processors. While this approach maintained load balance for most cases, it failed to achieve load balance in cases where the camera was inside the volume. Ma et al. [12] presented a solution that uses round robin cells assignment to achieve approximate load balance by performing interleaved cell partitioning. In most cases cells that are spatially connected have similar sizes, thus assigning connected cells to different processors helps to balance the workload. This distribution method also helps in cases where the camera is focused on a region of the data. For each pixel, they stored the different ray partitions using a linked list. Processors render cells of the same image space at the same time using cell-projection to allow early merge of ray partitions. Image space regions are calculated using a k-d tree, and all processors have the same k-d tree communicated by a host processor.

Steiner et al. [13] presented a dynamic load balancing technique that used a work package pulling mechanism within the Equalizer [14] framework. The server divided the work into equal packages and inserted them into a queue. When a client completed the work for its assigned packages, it requested more work from the server. They compared their method to the traditional method within Equalizer, also known as Load Equalizer. Their solution showed improved load balance for the object order approach.

Muller et al. [15] demonstrated a dynamic load balancing technique, where the balance of each processor was calculated while sampling the cells. Blocks of data were transferred between processors to balance work. The drawback of their method was the cost of moving data between processors. In addition, it may still fail to maintain equal workloads among processors, if the camera is zoomed into a particular region. This would result in a heavier workload for one processor.

Our approach solves this problem by deferring processing of large cells, and distributes the workload among processors in a later phase.

2.1.3 Image Compositing

The final step in the object order approach is image compositing. This step orders data in the correct depth order. Image compositing can be the most expensive step in the object order approach [6]. As a result, several techniques have been developed to perform image compositing and reduce its cost.

Direct send [16] is the simplest method, where all processors communicate with each other. Image tiles are assigned to processors, and processors exchange data. Direct send has been used often in prior work [16], [17].

Another image compositing approach is binary swap [8]. This method requires the number of processors to be a power of two. Binary swap divides the communication between processors over different rounds. The number of rounds is equal to $log_2(N)$, where N is the number of processors. The size of the exchanged tiles gets reduced by half in each round. Processors communicate in pairs, and then swap pairs. Although binary swap reduces network congestion, it requires the number of processors to a be a power of two, and it has a synchronization overhead after each round. A modified version, 2-3 swap, was implemented by Yu et al. [18] to overcome the limitation that the number of processors had to be a power of two. Their algorithm allows any number of processors to be used and performs the communication in multiple rounds. These processors are divided into groups of twos or threes, and each group exchanges data using the direct send approach.

Peterka et al. proposed an approach [19], known as radix-k, that combines the flexibility of direct send and the performance of binary swap. This method divides the communication into multiple rounds, and defines a group size k_i, for each round, where i is the current round. For this algorithm, the product of all k_i must be equal to N, where N is the number of processors. Processors within each group exchange data using a direct send method, with each processor owning a region of the current image.

2.2 Hybrid Distributed-Memory Parallel Volume Rendering

Traditional distributed-memory parallel volume rendering techniques can suffer from load imbalance. Load imbalance impacts the performance of the algorithm since the execution time is determined by the time of the slowest processor. As a result, resources are wasted. Using a hybrid solution that combines object order and image order approaches can reduce load imbalance by ensuring equal distribution of work and thus better performance.

An initial hybrid approach was developed by Montani et al. [20]. In their approach, processors were grouped into clusters, and the image order approach was used to assign a tile for each cluster. The data was copied to each cluster and distributed among its processors using the object order approach. Their solution helped to achieve data scalability by using object order at the node level while reducing the amount of data replication. Load imbalance could happen at the cluster level if one tile of the image had larger cells, causing a group of processors to do more work. It might also occur among the processors of a cluster if the data assigned to one processor had a larger region of the tile than other processors.

Childs et al. [1] developed another hybrid approach. This approach is the basis for our own work. Their algorithm classified cells as small or large, depending on the number of samples per cell. The algorithm had three phases: small cell sampling, exchange, and large cell sampling. With their algorithm, data was first distributed among processors using the object order approach. Small cells were sampled and large cells were deferred into a later stage. Second, data was exchanged, specifically large cells and samples. Next, each processor sampled the large cells that it owned using an image order approach and composited its pixels to generate a sub-image. Our solution is similar to this previous work, in its three main phases. However, we extend their algorithm to take advantage of many-core architectures and we also significantly reduce their memory costs.

2.3 Unstructured Data and Volume Rendering

Unstructured meshes create particular difficulties for volume rendering since their cell sizes can vary dramatically (leading to load imbalance), since they can present interleaving conditions that affect compositing ordering, and since they do not have a native layout and ordering (like structured grids). As a result, there have been several works that have addressed volume rendering of unstructured grids specifically.

Ma [11] computed cell connectivity as a pre-step that was used to traverse points along a ray. The pre-step also determined the exterior faces (i.e., faces that are not shared between different cells). He used a hierarchical data structure (HDS) to store face-node, cell-face, and cell-node relationships. Their algorithm worked as follows. Each processor performed the cell connectivity pre-step. Next, cells that are outside the camera view were excluded. Each processor applied the ray-casting algorithm to its data. Each ray entered the volume from an exterior face and used the connectivity information to determine the next cell. When a ray intersected a second exterior face, it exited the volume. Later, image pixels were assigned to different processors and HDS was used to exchange rays. Rays were composited to produce the final image.

Max et al. [21] used a slicing based approach to render unstructured data. For each cell, three slices were generated perpendicular to the X, Y, and Z axes. The camera view determined which of these slices was used. The value of each sample was computed by interpolating the cell vertices. Each corresponding scalar value was used as a 1D texture coordinate to obtain the color. Slices were rendered depending on their distance from the viewing direction in a back-to-front order. Colors were blended to generate the final pixel color.

Our solution adapts a many-core sampler (EAVL) implemented by Larsen et al. [22], where cells were sampled in parallel using multi-threading. The samples of each cell were stored in a shared buffer. The index of each sample was calculated depending on its screen space coordinates (x, y, and z). In the compositing stage, the samples of each ray were combined to produce the final color. We incorporated their routine to work within a distributed system since their routine was designed for a single node only.

3 Algorithm

This section presents our distributed-memory parallel volume rendering algorithm, which extends the algorithm by Childs et al. [1]. The algorithm works on both structured and unstructured grids, but it is designed with unstructured grids in mind, since this mesh type is prone to load imbalance issues. To prevent load imbalance, we use a hybrid between object order and image order approaches. Our algorithm performs ray-casting, evaluating samples along each ray. However, it combines the contributions of samples from different processors in a way that ensures the correct picture. Specifically, when an unstructured mesh is partitioned into blocks and distributed over processors, and when block boundaries interleave in complex ways, then we create the correct picture even when a ray enters and exits a pair of blocks multiple times (see Figure 1).

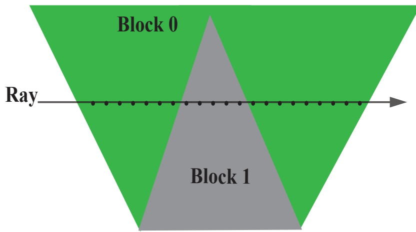


Fig. 1: A ray going through two interleaving blocks. The green color indicates the first block (Block 0) and the gray color indicates the second block (Block 1). Our algorithm is able to correctly order the contributions from the first span through Block 0, then the contributions from Block 1, and then the second span through Block 0.

Conceptually, our algorithm creates a buffer that stores all samples along all rays. Such an approach is typically not advisable in a serial setting, since this buffer will likely have a billion samples or more (e.g., 1024×1024 pixels and 1000 samples along each pixel's ray). However, it is more viable to store this buffer in a distributed-memory parallel setting, since there is access to more memory. That said, a key difference between our work and the predecessor work by Childs et al. [1] is that we combine consecutive samples (into so-called "partial composites") to minimize memory usage, while their work allocated memory for every sample (although distributed across processors). Our predecessor work, which has been deployed in VisIt [23] for the past decade, has proven to be prone to excessive memory usage, prompting this improvement.

Our algorithm has two sampling phases: one is performed in an object order setting and the other is performed in an image order setting. For each cell, we compare the estimated number of samples per cell with a given threshold. We refer to cells with fewer samples than the threshold as "small," and those with more samples as "large." Small cells are sampled in the first sampling phase (object order) and large cells are sampled in the second sampling phase (image order). If the threshold is equal to one, the hybrid approach classifies all cells to be large and behaves exactly like the image order approach. On the other hand, if the threshold is large, then the hybrid approach classifies all cells to be small and behaves like the object order approach (see Figure 2). The hybrid approach bounds the amount of work done by each processor in each phase, promoting load balance.

Fig. 2: For extreme choices of threshold value, our hybrid algorithm can act as either an object order and image order volume rendering algorithm. For threshold values between these extremes, it acts as a new algorithm with characteristics of both.

Our algorithm works as follows:

1) Object Order Sampling Phase
   Data is distributed among processors and each processor samples its small cells.
2) Exchange Phase
   The algorithm partitions image pixels over processors. The processors exchange the results of the first sampling phase among themselves, as well as the large cells that were not sampled.
3) Image Order Sampling Phase
   Each processor samples the large cells that contribute to its pixels. For each pixel, the samples are composited, producing a sub-image.

These phases are illustrated in Figure 3.

In the remainder of this section, we describe the algorithm in more depth. We begin by describing the foundational concepts of partial composites in Section 3.1. Section 3.2 through 3.4 describe the object order sampling phase, the exchange phase, and the image order phase, respectively.

3.1 Partial Composites

3.1.1 General Description

Partial composites refers to the technique to reduce memory and communication cost. This technique reduces the memory size by compositing a group of consecutive samples into color, opacity, and depth information. Partial composites are calculated as follows. First, the transfer function is used to assign a color and opacity to each sample. Next, these colors are composited in front-to-back order using the following equation:

$$C_{out} = C + (1 - \alpha) * C_{in} * \alpha_{in} \tag{1}$$

Where C_{out} is the RGBA value of the output partial composite, C is the color of the current partial composite, C_{in} is the color of the current sample, α is the accumulated opacity, and α_{in} is the opacity of the current sample.

3.1.2 Implementation

Without partial composites, the cost of storing and exchanging samples can be high. Assuming that each sample is represented by a floating point value (4 bytes), then:

$$Cost_{bytes} = 4 * SPR * P \tag{2}$$

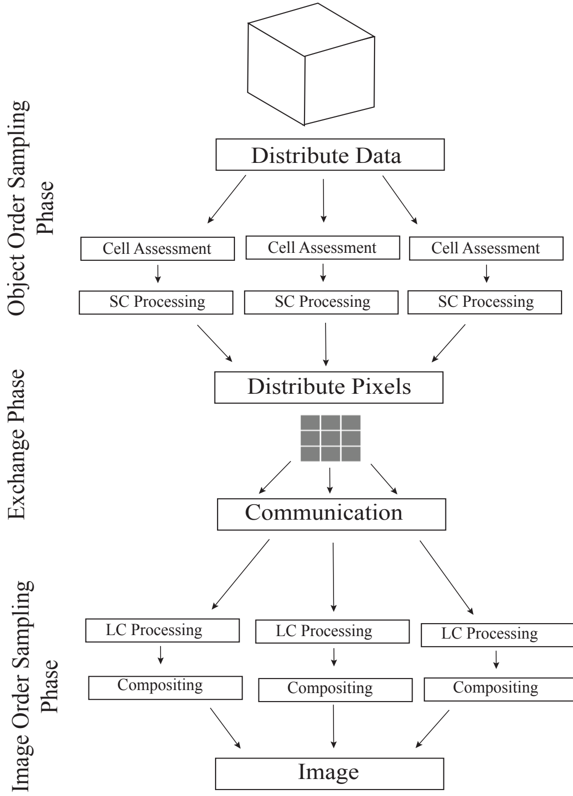


Fig. 3: The algorithm pipeline highlights the three main phases: object order, exchange, and image order. SC denotes small cells, and LC denotes large cells.

Where $Cost_{bytes}$ is the cost in bytes, SPR is the number of samples per pixel, and P is the number of pixels. For an image with a million pixels and 1000 samples per ray, the cost will be 4 GB. But partial composites combine consecutive samples into color, opacity, and depth information, which uses 24 bytes no matter how many samples are in a consecutive run. Revising the example with a million pixels and 1000 samples per ray, the cost with partial composites can drop to as low as 24 MB.

Consider the following example of representing consecutive samples with a partial composite. Assume the first sample in the sequence, denoted as Z_s, is located at the third sample along the ray. Further assume that the last sample in the sequence, denoted as Z_e, is located at the 102nd index along the ray. Then Z_s = *3* and Z_e = *102*. This sequence contains 100 samples, and, assuming four bytes per sample, would occupy 400 bytes overall. Compositing these samples into a color *RGB* and opacity *A* (represented as C_{out} in the previous equation), will result in 4 floating point values (RGBA). Two additional integers are used to track the depth information. The first additional integer (Z_s) stores the depth of the first value in the partial composite (i.e., Z_s = *3*). The second (Z_e) stores the depth of the final sample in the partial composite (i.e., Z_e = *102*). This will result in 4 floating point values and 2 integer point values instead of 100 thus reducing the storage from 400 bytes to 24 bytes, benefiting both communication and memory.

The final pixel color is obtained by accumulating the partial composites after sorting them in the correct z-depth order by using the depth information (Z_s and Z_e) for each partial composite. For example, assume a processor has three partial composites after the exchange, PC_1, PC_2, and PC_3, with the following depth information respectively: ($Z_{1s} = 501$, $Z_{1e} = 1000$), ($Z_{2s} = 0$, $Z_{2e} = 2$) and ($Z_{3s} = 3$, $Z_{3e} = 500$). These partial composites will be sorted front-to-back in the following order: PC_2, PC_3 and then PC_1.

3.2 Object Order Sampling Phase

3.2.1 Distribute Data

The first step of the algorithm is to distribute data over processors. Our implementation supports two options, reading the data from a single file or reading multiple decomposed blocks. In the first case when the data is not decomposed, each processor reads $\frac{C}{P}$ cells, where *C* is the number of cells and *P* is the number of processors. In the second case, each processor reads a block.

3.2.2 Cell Assessment

In this step, the algorithm evaluates the properties of each cell. We estimate the number of samples per cell by considering its bounding box in image space. Our estimate is based on the product of two factors: first, the number of pixels covered by the bounding box, and second, the number of samples in depth overlapping the bounding box. This overestimates the possible number of samples. Next, we classify cells as small or large by comparing the estimated number of samples with a given threshold.

3.2.3 Small Cell Processing

This step contains two sub-stages: 1) sampling small cells and 2) generating partial composites along each ray. To perform the sampling, we adapt a multi-threaded implementation by Larsen et al. [22]. Parallelism is achieved by working on different cells simultaneously. As the cells are sampled, their values are stored in the sample buffer.

Next, partial composites are generated for each pixel. Parallelization occurs by working on different pixels simultaneously. For each group of continuous samples along the depth-axis, samples are composited into a partial composite.

3.3 Exchange Phase

3.3.1 Distribute Pixels

In preparation for the image order phase, image tiles are assigned to different processors, and each processor receives the data contributing to its tile. Image pixels are distributed among processors by dividing the image into tiles. The size of the tile assigned to each processor is $(W/\sqrt{N}) * (H/\sqrt{N})$, where W is the width of the image, H is the height of the image, and N is the number of processors. This simple scheme was sufficient for our study but can be adapted easily in the future.

3.3.2 Communication

In this phase, two types of data are being exchanged: 1) large cells and 2) partial composites. After destination processors are determined, the data is packed into a buffer. Then, processors perform the exchange in a direct send approach using the Message Passing Interface (MPI) [24]. Each processor receives its data and unpacks the messages into a defined structure (partial composite or large cell).

We experimented with different data exchange methods and found direct send to perform the best for our algorithm.

3.4 Image Order Sampling Phase

3.4.1 Large Cell Processing

In this sub-phase large cells are sampled, and then partial composites are generated. Both of these operations are performed using multi-threading. While sampling large cells, each processor restricts its sampling to its tile boundaries so that only the contributing part of the cell is considered. This removes redundant sampling that could occur over different processors. At this point, each processor has the data it needs to generate its sub-image.

3.4.2 Compositing

In this step, each processor orders and combines its partial composites to produce the final color of the pixel. The depth information, stored as Z_s and Z_e (described in Section 3.1), is used to order the partial composites. Finally, these partials are combined in a front-to-back order using the following equation:

$$C_{out} = C_{in} + (1 - \alpha) * C \tag{3}$$

Where C_{out} is the RGB value of the pixel, C_{in} is the color of the current partial composite, and α is the accumulated opacity. This equation combines the colors of the partial composites computed earlier, so it does not need to take the opacity of each sample into consideration, as it was calculated already in Equation 1.

3.4.3 Collecting Final Image

Image tiles are collected to produce the final image. One processor acts as the host, and all other processors send their tiles to the host. The host processor receives the tiles and arranges them according to the assignment of each processor.

3.5 Lighting

In the present study, the volume renderings do not employ a lighting model. That said, the hybrid approach for parallel volume rendering has been previously extended to perform lighting [23], by treating the gradient of the underlying scalar field as the normals for a Phong lighting model. If this study were to have included lighting, then the impacts would depend on how the gradients were obtained. Calculating gradients during sampling would increase execution time, while using a pre-calculated gradient would increase memory footprint and data exchanges.

4 Analysis of Complexity

4.1 Performance

In this section, we compare the performance of the three approaches, i.e., object order, image order, and hybrid. We analyze each phase considering computational complexity, and the analysis is summarized in Table 1. We define the following abbreviations, which we use throughout the rest of the paper:

- OO: Object Order approach.
- IO: Image Order approach.
- HB: HyBrid approach.
- P: the total number of pixels.
- N: the total number of processors.
- SPR: the number of samples per ray.
- C_i: the number of cells owned by processor i.
- SC_i: the number of small cells owned by processor i.
- LC_i: the number of large cells owned by processor i.
- PC_i: the number of partial composites owned by processor i.
- SPC_{ij}: the estimated number of samples for cell j owned by processor i.

As mentioned in Section 3, our approach has three main phases: object order sampling, exchange, and image order sampling. Our analysis in this section considers all three approaches within the context of these three phases. For the object order approach, all cells would be sampled in the object order sampling phase, and no work occurs during the image order sampling phase. For the image order approach, there are two cases. The first case is when the data is small and it is replicated on each node. In this case, only the image order sampling phase of our model occurs. The second case is when the data is large and is distributed across different nodes, which is representative of the case when operating in situ. In this case, nodes exchange data and thus both the exchange phase and the image order sampling phase are performed. For our analysis, we assume this latter case.

4.1.1 Object Order Sampling

In the object order sampling phase, the performance varies between the approaches. In the hybrid approach, only small cells are sampled. Further, because the cells are small, the number of samples for each cell is fixed, and therefore bounded by a constant. Thus the time for this phase is proportional to the number of small cells owned by the processor (SC_i). In the object order approach, all cells are sampled. Thus the performance of processor i depends on the number of cells owned by that processor (C_i) and on the number of samples for each cell (SPC_{ij}).

Since the object order approach samples all the cells, load imbalance can happen if the number of samples varies between processors. This can happen when the camera is zoomed into one of the regions owned by one processor, or when dealing with an unstructured mesh that has unequal cell sizes. In these cases, the object order approach will have unequal distribution of samples, resulting in load imbalance and increased execution time. The hybrid approach avoids that by deferring sampling of large cells, thus preventing any processor from doing significantly more work than expected.

TABLE 1: Analysis of a processor's execution time for the object order, image order, and hybrid approaches. The † symbol denotes the possibility of extreme load imbalance. For the hybrid approach, the work performed by each processor is bounded, and load imbalance only occurs when some processors do less work than their upper limit.

Steps	Object Order	Hybrid	Image Order
Object Order Sampling	$O(C_i) + O(\sum_{j=0} SPC_{ij})$†	$O(SC_i)$	–
Exchange	$\sum_{i=0}^{n} PC_i$	$\sum_{i=0}^{n} PC_i + \sum_{i=0}^{n} LC_i$	$\sum_{i=0}^{n} C_i$
Image Order Sampling	–	$O(\frac{P * SPR}{N})$	$O(Ci) + O(\frac{P * SPR}{N})$†

4.1.2 Exchange

While both the object order and hybrid approaches exchange partial composites (PC_i), the hybrid approach has an additional cost in the exchange phase, which is the cost of exchanging large cells (LC_i). Both the object order and hybrid approaches have a fixed data exchange, since the amount of data exchanged is dependent only on the number of samples and image size and not dependent on the size of data. For the image order approach, the data is distributed across the nodes, therefore the image order approach exchanges cells (C_i). When the number of cells is very large, the image order approach may have high exchange time.

4.1.3 Image Order Sampling

In the image order sampling phase, the performance varies between the approaches. In the hybrid approach, the number of cells is smaller than the number of samples, so the number of cells it processes is bounded. This number is bounded by the potential number of samples in its tile, specifically the number of pixels (P) multiplied by the number of samples per ray (SPR) divided by the number of processors (N). In the image order approach, the possible number of samples to extract is the same as the hybrid approach. However, in the image order approach, processors might have different numbers of cells, which can result in load imbalance. For example, when the camera is zoomed out, a processor might have a part of the image that has more cells contributing to it. Using the image order approach for this case results in unequal distribution of cells per processor since some processors may have more data in their tile.

4.1.4 Summary

The performance for ray-casting is proportional to the number of cells and the number of samples. The object order approach has a bound on the number of cells per processor, but it does not guarantee equal distribution of samples per processor. And the image order approach has a bound on the number of samples per processor, but it does not guarantee equal distribution of cells per processor. Therefore, both the object order and the image order approaches are susceptible to load imbalance. The hybrid approach, on the other hand, addresses these limitations by ensuring equal distribution of both cells and sampling per processor.

5 Study Overview

This section describes the details of our study, which is composed of two main parts:

- Algorithm Comparison
  In this part, we compare the performance and load imbalance of our hybrid approach against both object and image order approaches using different factors, which are described in the following subsection.
- Distributed Memory Scalability
  In this part, we test scalability since our algorithm is designed to achieve high performance and load balance on a distributed system.

The configurations for the first part of our study are described in Subsection 5.1, and the configurations for the second part are in Subsection 5.2.

5.1 Algorithm Comparison Factors

The first part of our study is composed of four phases. Each phase varies one of four factors, while holding the other three constant. The four factors are:

- Camera position (4 options)
- Data set (24 options)
- Image size (4 options)
- Transfer function (5 options)

In total, we considered 37 (= 4 + 24 + 4 + 5) configurations in this part of our study. For each configuration, we ran three experiments, one for each algorithm, meaning 111 experiments overall. We ran the experiments on 256 nodes, with 1 MPI task per node and 12 threads of on-node parallelism. Each of these factors and their options is discussed in the following subsections.

5.1.1 Camera Position

As discussed in Section 4, varying the camera position can have a high impact on algorithm performance. We consider four camera positions:

- Zoom out
- Mid zoom
- Zoom in
- Inside

Figure 5 shows a data set at these camera positions.

For each position, we measure the execution time for the exchange step, the sampling step, and the total execution time. We then compare performance and load balance across algorithm and camera position.

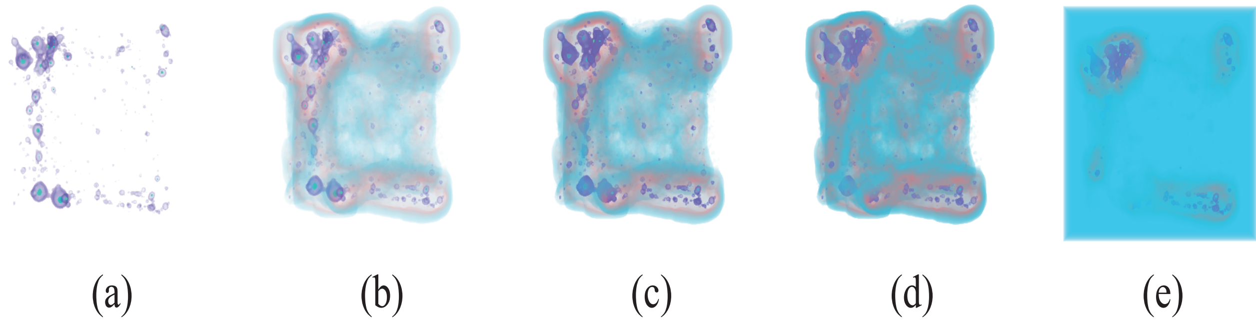


Fig. 4: The five settings for transfer function opacity in our study: (a) Almost Transparent, (b) Light, (c) Middle, (d) Dense, (e) Nearly Opaque.

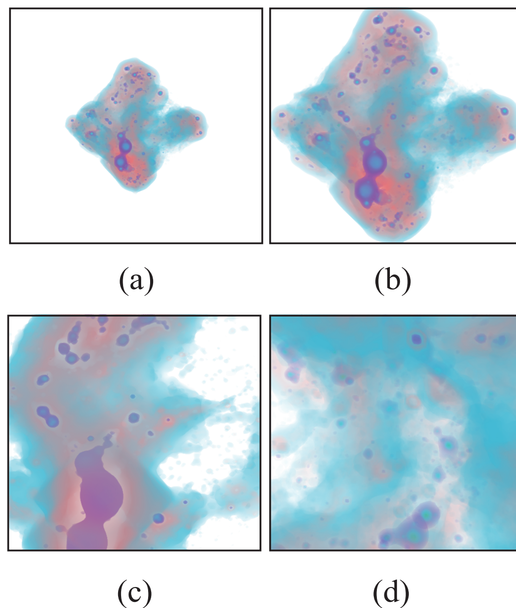


Fig. 5: The four settings for camera positions in our study: (a) zoom out, (b) mid zoom, (c) zoom in, and (d) inside.

5.1.2 Data Set

We test the performance of our algorithm using different sizes of two data sets: Enzo and Ice. As the data set size increases, the number of cells increases, which mean the sizes of the cells decrease. This impacts our algorithm since small cells are sampled in a different sampling phase than large cells. The data sets we consider are:

- Enzo-1M: A cosmology data set from the Enzo simulation code [25]. The data set was originally on a rectilinear grid but was mapped to a tetrahedral grid.
- Enzo-10M: A 10.2M tetrahedron version of Enzo-1M.
- Enzo-80M: An 83.9M tetrahedron version of Enzo-1M.
- Enzo-1B: A 1 billion tetrahedron version of Enzo-1M.
- Ice-6M: A simulation of a high speed train and the surrounding conditions. This data set is tetrahedral, and its 6M tetrahedrons significantly vary in size.
- Ice-1B: A 1 billion tetrahedron version of Ice-6M.

For each of the six data sets, we test four camera positions, zoom out, mid zoom, zoom in and inside, meaning this phase has 24 options overall.

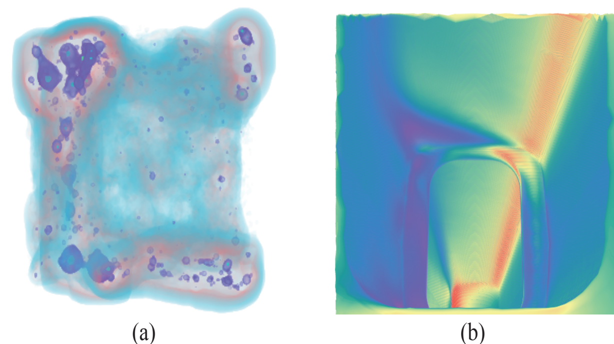


Fig. 6: The two data sets used in our study: (a) Enzo and (b) Ice.

5.1.3 Image Size

The image size affects both sampling and compositing. Four image sizes are considered:

- 100×100
- 200×200
- 500×500
- 1024×1024

5.1.4 Transfer Function

As all three approaches exclude cells that have zero opacity, we study the effect of different opacity values on the execution time. We picked five transfer functions with varying opacities and denote them as:

- Almost Transparent
- Light
- Middle
- Dense

- Nearly Opaque

Volume renderings using these transfer functions are plotted in Figure 4.

5.2 Distributed Memory Scalability

In the second section of our study, we test the performance scaling of our algorithm by varying the number of nodes as follows: 1, 2, 3, 4, 8, 12, 16, 32, 64, 128, 256, 512, and 1024. We use 8 MPI tasks per node resulting in a maximum of 8192 MPI tasks. We test the scalability for two camera positions: zoom in and zoom out. For both positions we use the Middle opacity transfer function, an image size of 1024×1024, and 1000 samples per ray. For the zoom in camera position we test the Enzo-1M data set, and for the zoom out camera position, we test the Enzo-10M data set.

5.3 Hardware Used

- Edison: We performed the first part of our experiments on Edison, a machine at Lawrence Berkeley National Laboratory's NERSC facility. It contains 5,586 nodes, with each node running two Intel "Ivy Bridge" processors at 2.4 GHz. There are 12 cores and 64 GB of memory per node.
- Vesta: We performed the second part of our experiments on Vesta, a supercomputer at Argonne National Laboratory. It contains 2,048 nodes, with each node running a "PowerPC A2" processor at 1.6 GHz. There are 16 cores and 16 GB of memory per node.

5.4 Performance Measurement

We measure the non-idle execution time for every phase on every processor.

From these measurements, we can derive load imbalance. Load imbalance affects the performance because the execution time is determined by the time of the slowest processor. In addition, load imbalance results in wasted resources that are idle. Let T_p be the total non-idle execution time for processor P, and T_s is the total non-idle execution time of the slowest processor. We define load imbalance with the following equation:

$$\text{Load imbalance} = \frac{\mathrm{T_s}}{\sum_{0<\mathrm{p}<\mathrm{N}} \mathrm{T_p}/\mathrm{N}}$$

6 Results

We compare the performance of our hybrid approach with the object order and image order approaches. In Section 6.1, we present the first part of our study, i.e., parallelism at 256 nodes while varying other factors. In Section 6.2, we present the second part of our study, i.e., the scalability of the three approaches.

6.1 Algorithm Comparison

Our comparison happens in phases, with each phase varying one of the factors described in Section 5.1.

6.1.1 Camera Position

In this phase we vary the camera position, using the following configuration:

- Data set: Enzo-1M data set
- Nodes: 256 nodes, 1 MPI task per node
- Cores: 12 cores per node for multi-threading
- Machine: Edison
- Image size: 1024×1024
- Samples per ray: 1000
- Transfer function: Middle

The results of this phase are presented in Table 2.

For the zoom out camera position, the hybrid approach is faster than the image order approach because it reduces load imbalance. Further, the performance of the hybrid approach is equal to the object order approach. The hybrid approach produces comparable results to the object order approach since the number of samples per cell is small and thus all cells are sampled in the first sampling phase. In this case, the sampling for the hybrid approach is identical to that of the object order approach. For the mid zoom camera position, the hybrid approach is closer to the object order approach, which means most of the sampling is performed in the first phase (object order sampling phase). Although the hybrid approach has a higher execution time than the object order and image order approaches due to the extra work it is performing (two sampling phases and two exchange operations), it has better load balance. The extra operations of the hybrid approach pay off when the size of the data is large, as presented later in Table 4.

For both the zoom in and the inside camera position, the situation reverses. The hybrid approach is faster than the object order approach because it reduces load imbalance, while it performs similarly to the image order approach.

For all three camera positions, the hybrid approach maintains good performance due to its low load imbalance. The hybrid approach achieves this by adapting its behavior according to the configuration. The two other approaches, however, can suffer from significant load imbalance. When the camera is zoomed into the data set, the object order approach has unequal work distribution, which leads to a high execution time. As the camera moves further from the data, the execution time for the image order approach increases, since there are more cells to exchange (i.e., not just a small subset that is visible as a result of zooming in). In addition, the distribution of cells per tile becomes less balanced, resulting in unequal work distribution among processors. The hybrid approach maintains good performance and low load imbalance in this case, because the work performed by each processor is bounded at each step of the algorithm (see Section 4).

6.1.2 Data Set

In this phase we vary the data set, using the following configuration:

- Nodes: 256 nodes, 1 MPI task per node
- Cores: 12 cores per node for multi-threading
- Machine: Edison
- Image size: 1024×1024

TABLE 2: Results using 256 nodes, and varying the camera position.

	Zoom out		
	OO	HB	IO
Exchange	0.043s	0.043s	0.054s
Sampling	0.179s	0.179s	0.38s
Total	0.23s	0.23s	0.461s
Load Imbalance	1.31x	1.31x	1.531x
	Mid zoom		
	OO	HB	IO
Exchange	0.054s	0.069s	0.097s
Sampling	0.375s	0.44s	0.023s
Total	0.440s	0.521	0.164s
Load Imbalance	1.258x	1.167x	1.228x
	Zoom in		
	OO	HB	IO
Exchange	0.057s	0.013s	0.013s
Sampling	0.86s	0.089s	0.089s
Total	0.927s	0.141s	0.141s
Load Imbalance	1.44x	1.08x	1.08x
	Inside		
	OO	HB	IO
Exchange	0.47s	0.18s	0.18s
Sampling	4.65s	0.11s	0.11s
Total	5.12s	0.32s	0.32s
Load Imbalance	2.59x	1.10x	1.10x

- Samples per ray: 1000
- Transfer function: Middle

The results of this phase demonstrate that using the hybrid approach reduces the execution time for both small and large data sets. This improved performance can be understood by considering the number of samples per cell. As the number of cells increases, the size of the cells becomes smaller, which means less samples per cell.

When the camera is far from the data set, the number of small cells increases. For the zoom out camera position, the hybrid approach samples all cells in the first sampling phase (object order sampling phase), as presented in Table 3. The results from Table 4 show that the difference between the

TABLE 3: Results using 256 nodes, and varying data sets for the zoom out camera position.

Data Set	Camera	OO	HB	IO
Enzo-1M	zoom out	0.23s	0.23s	0.461s
Enzo-10M	zoom out	0.257s	0.251s	0.568s
Enzo-80M	zoom out	0.275s	0.275s	1.508s
Enzo-1B	zoom out	0.777s	0.773s	12.575s
Ice-6M	zoom out	0.246s	0.247s	0.487s
Ice-1B	zoom out	0.795s	0.792s	12.799s

TABLE 4: Results using 256 nodes, and varying data sets for the mid-zoom camera position.

Data Set	Camera	OO	HB	IO
Enzo-1M	mid-zoom	0.440s	0.521s	0.164s
Enzo-10M	mid-zoom	0.443s	0.457s	0.330s
Enzo-80M	mid-zoom	0.450s	0.425s	1.07s
Enzo-1B	mid-zoom	0.875s	0.888s	9.947s
Ice-6M	mid-zoom	0.530s	0.413s	0.290s
Ice-1B	mid-zoom	1.055s	0.891s	9.610s

TABLE 5: Results using 256 nodes, and varying data sets for the zoom in camera position.

Data Set	Camera	OO	HB	IO
Enzo-1M	zoom in	0.927s	0.141s	0.141s
Enzo-10M	zoom in	0.661s	0.457s	0.314s
Enzo-80M	zoom in	0.529s	0.529s	0.601s
Enzo-1B	zoom in	0.921s	0.922s	2.741s
Ice-6M	zoom in	1.095s	0.300s	0.302s
Ice-1B	zoom in	0.940s	0.936s	2.831s

TABLE 6: Results using 256 nodes, and varying data sets when the camera is inside the data.

Data Set	Camera	OO	HB	IO
Enzo-1M	inside	5.12s	0.32s	0.32s
Enzo-10M	inside	6.71s	0.73s	0.75s
Enzo-80M	inside	5.19s	0.98s	1.85s
Enzo-1B	inside	7.80s	1.27s	2.98s
Ice-6M	inside	6.47s	0.652s	0.651s
Ice-1B	inside	9.083s	1.301s	3.056s

execution time for the hybrid and object order approaches for the mid-zoom position is larger than the zoom out position. The hybrid approach takes more time when the data size is small due to the extra cost mentioned previously (two sampling phases and two exchange operations). This cost pays off when the size of the data gets larger (Enzo-80M, Enzo-1B, and Ice-1B). Finally, the modest increase in the execution time for the Ice data and object order approach compared to the Enzo data is because the mid-zoom camera position for the Ice data is somewhat closer than that for the Enzo data.

Table 5 shows the results for the zoom in camera position. As the camera moves closer to the data, the number of large cells increases. This explains the decrease in the difference between the performance of the hybrid and image order approaches and the increase in the difference between the performance of the hybrid and object order approaches. In the smallest data set, Enzo-1M, the cells are larger, so the hybrid approach defers sampling of these cells into the second sampling phase (image order sampling phase). As the data set gets larger, the cells sizes get smaller, so most of the cells are sampled in the object order sampling phase of our algorithm. Therefore, the execution time for the object order approach decreases as the number of cells increases, because the distribution of work among processors gets more balanced. The same case can be noticed for the Ice-6M and Ice-1B. For the object order approach, the distribution of cells for Enzo data is more balanced than the Ice data. Thus the execution time for the Ice data is higher than the Enzo data.

Table 6 presents the results of the three approaches when the camera is inside the data. The object order approach consistently performs poorly due to load imbalance. While the image order approach achieves good performance when the size of the data is small, as the data size increases the performance gets worse due to the time of exchanging cells. On the other hand, the hybrid approach achieves good performance and load balance for all data sizes.

6.1.3 *Image Size*

In this phase we vary the image size, using the following configuration:

- Data set: Enzo-1M data set
- Nodes: 256 nodes, 1 MPI task per node
- Cores: 12 cores per node for multi-threading
- Machine: Edison
- Camera position: Zoom in
- Samples per ray: 1000
- Transfer function: Middle

TABLE 7: Results using 256 nodes, and varying the image size.

Image Size	OO	HB	IO
100×100	0.017s	0.066s	0.066s
200×200	0.063s	0.101s	0.101s
500×500	0.91s	0.124s	0.124s
1024×1024	0.927s	0.141s	0.141s

Table 7 shows that the impact of varying the image size on the three algorithms. As the size of the image increases, the number of samples per cell increases, which means the number of larger cells increases. This explains the increase in the difference between the object order execution time and the hybrid execution time. When the size of the image is small, there are few large cells, and thus the hybrid approach samples most of the cells in the object order sampling phase. With the increase in the image size, the hybrid approach samples more cells in the image order sampling phase.

6.1.4 *Transfer Function*

In this phase we vary the transfer function, using the following configuration:

- Data set: Enzo-1M data set
- Nodes: 256 nodes, 1 MPI task per node
- Cores: 12 cores per node for multi-threading
- Machine: Edison
- Camera position: Zoom in
- Image size: 1024×1024
- Samples per ray: 1000

The opacity of the transfer function determines the amount of visible data. A more dense transfer function results in computing more samples, thus higher execution time. Table 8 shows that the influence of transfer function on the execution time is small, and that all three approaches increase proportionally as the opacity increases.

6.2 Distributed Memory Scalability

In this section, we study the scalability of our algorithm.

This phase uses the following configuration:

- Data set: Enzo-1M and Enzo-10M data sets
- MPI tasks: 8 MPI task per node
- Cores: 1 per node, no multi-threading
- Machine: Vesta

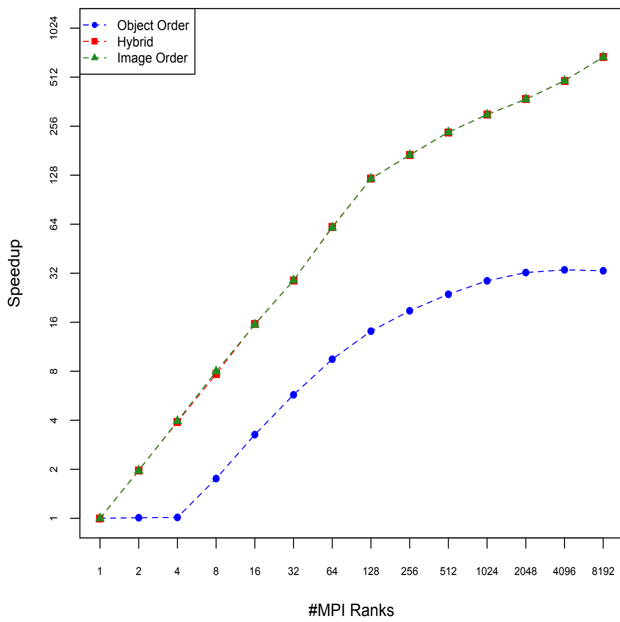


Fig. 7: Scalability of the three algorithms with the Enzo-1M data set and the zoom in camera position.

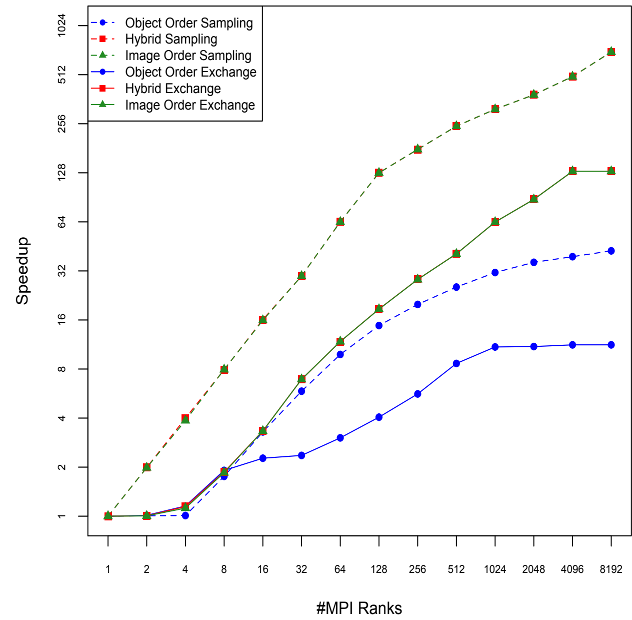


Fig. 8: Scalability of sampling and exchange for the three algorithms with the Enzo-1M data set and the zoom in camera position. This figure looks at the same results as Figure 7, but instead presents the information on a per phase bases.

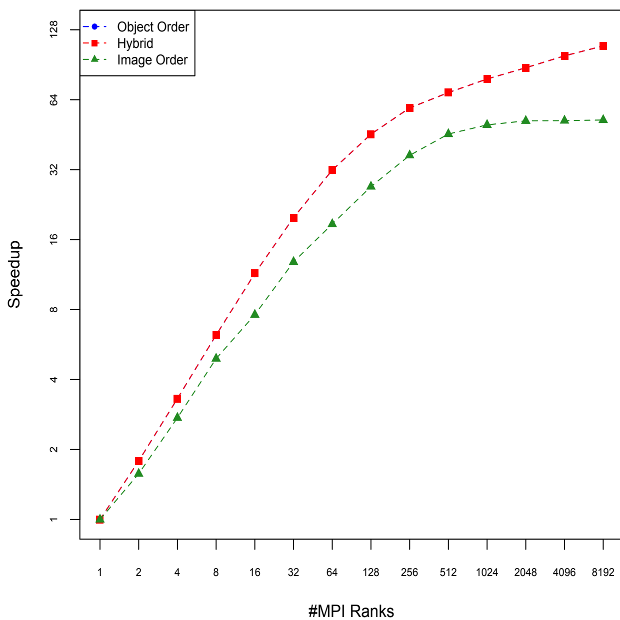


Fig. 9: Scalability of the three algorithms with the Enzo-10M data set and the zoom out camera position.

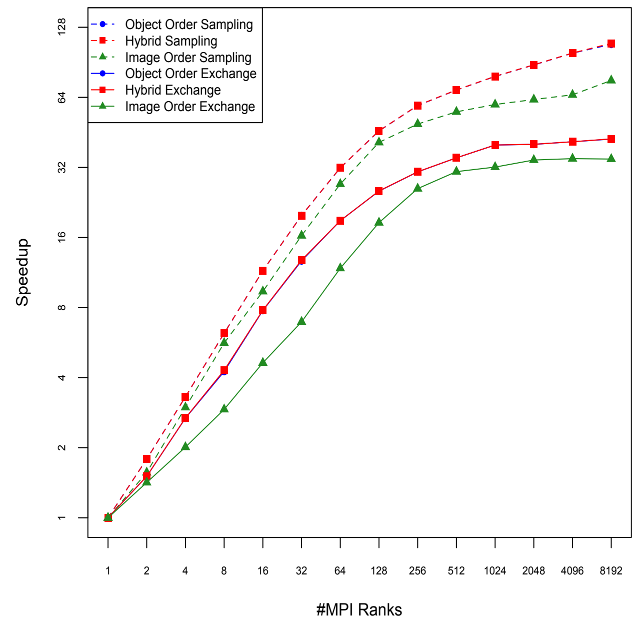


Fig. 10: Scalability of sampling and exchange for the three algorithms with the Enzo-10M data set and the zoom out camera position. This figure looks at the same results as Figure 9, but instead presents the information on a per phase bases.

TABLE 8: Results using 256 nodes, and varying the opacity of the transfer function.

Transfer Function	OO	HB	IO
Almost Transparent	0.885s	0.098s	0.10s
Light	0.915s	0.129s	0.128s
Middle	0.927s	0.141s	0.141s
Dense	0.934s	0.151s	0.152s
Nearly Opaque	0.938s	0.155s	0.155s

- Camera positions: zoom in and zoom out
- Image size: 1024×1024
- Samples per ray: 1000
- Transfer function: Middle

Figure 7 presents the results for the zoomed in camera position. In this case, the object order approach is taking much more time because there is an unequal distribution of work among processors. Adding more processors is not helpful for the algorithm, since the workload is not being distributed evenly and the cost of exchanging partial composites increases. Since the hybrid approach is equivalent to the image order approach in the zoom-in camera position case, both approaches have similar scaling rate. Figure 8 presents the individual scalability curves for the sampling and the exchange steps. While the sampling step has a better scaling rate than the exchange step for all three algorithms, the sampling scaling rate reduces when reaching a large number of processors. This happens because the amount of actual sampling computation reduces, and the fixed time of the initialization step dominates the time of the total sampling step.

Figure 9 demonstrates the case where the camera is zoomed out. This experiment uses the Enzo-10M data set to test the scaling of the three algorithms. As mentioned earlier in Section 5.1.2, as the number of cells increases the size of the cells becomes smaller. The Enzo-10M data set is enough to make the number of samples per cell below our threshold for the hybrid algorithm. The hybrid approach samples all the cells in the first sampling phase (object order sampling phase), thus the hybrid approach is effectively equal to the object order approach. Since the camera is zoomed out, some tiles have more cells, which means the number of cells across tiles is not equal. Using the image order approach will then result in unequal distribution of cells, which will create load imbalance. As the number of processors increases, the image order approach will have many processors with empty tiles. Processors owning the middle of the image will have too many cells, while other processors will own empty tiles. Another factor that reduces the performance of the image order approach is the cost of exchanging cells. Figure 10 presents the individual scalability curves for the sampling and the exchange steps. Similar to the previous case, the sampling step has a better scaling rate than the exchange step, and the scaling rate reduces when reaching a large number of processors.

6.3 Summary of Findings

The evaluation showed that our hybrid algorithm performed well in all cases. In particular, the algorithm adapts its performance to meet the better of either object order or image order approaches. Further, in our most extreme test case (one billion cells with the camera inside the data set), our algorithm was faster than either of the classic approaches, since it avoided their individual pitfalls. In all, we feel that this evaluation demonstrates that the hybrid algorithm is a better choice for distributed-memory parallel volume rendering of unstructured data. Specifically, we feel that it is a better choice for production visualization tools. This is because production visualization tools must face a variety of use cases (including those that lead to load imbalance with traditional approaches), and it is very desirable in this setting to have an algorithm naturally adapts to achieve optimal performance without guidance from end users.

7 Conclusions and Future Work

The contribution of this paper is three-fold: (1) we introduced improvements over predecessor work that address memory footprint issues, (2) we evaluated the algorithm in a significantly more complete way, and (3) we modernized the algorithm to consider many-core architectures. As discussed in the summary of findings, our algorithm has superior performance to traditional approaches on workloads that are prone to load imbalance, and equivalent performance on all other workloads.

In terms of future work, we plan to integrate our algorithm into the VisIt visualization software. Our predecessor work was integrated into VisIt, but would crash with regularity due to excessive memory usage. We also plan to port our on node computations to VTK-m [26], since that software technology is now favored over EAVL [27]. Finally, we plan to further investigate parallel communication patterns at high concurrency. While Direct Send performed best up to 8192 MPI ranks, we believe higher ranks may well justify more sophisticated approaches. We also plan to study the impact of lighting models on performance.

Acknowledgments

This work was supported by Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357, Program Manager Lucy Nowell. Hank Childs is grateful for support from the DOE Early Career Award, Contract No. DE-SC0010652, Program Manager Lucy Nowell. Some of this work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-74473). Thanks for Dr. Christoph Garth at the University of Kaiserslautern for providing the Ice dataset, which was originally generated by Markus Rtten (markus.ruetten@dlr.de) at the German Aerospace Research Center, Gttingen.

REFERENCES

[1] H. Childs, M. Duchaineau, and K.-L. Ma, "A scalable, hybrid scheme for volume rendering massive data sets," pp. 153–161, 2006. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV06/153-161

[2] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh, "Load balancing for multi-projector rendering systems," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, ser. HWWS '99. New York, NY, USA: ACM, 1999, pp. 107–116. [Online]. Available: http://doi.acm.org/10.1145/311534.311584

[3] F. Erol, S. Eilemann, and R. Pajarola, "Cross-Segment Load Balancing in Parallel Rendering," in *Eurographics Symposium on Parallel Graphics and Visualization*, T. Kuhlen, R. Pajarola, and K. Zhou, Eds. The Eurographics Association, 2011.

[4] M. Balsa Rodríguez, E. Gobbetti, J. Iglesias Guitián, M. Makhinya, F. Marton, R. Pajarola, and S. Suter, "State-of-the-art in compressed gpu-based direct volume rendering," *Comput. Graph. Forum*, vol. 33, no. 6, pp. 77–100, Sep. 2014. [Online]. Available: https://doi.org/10.1111/cgf.12280

[5] J. Beyer, M. Hadwiger, and H. Pfister, "State-of-the-art in gpu-based large-scale volume visualization," *Comput. Graph. Forum*, vol. 34, no. 8, pp. 13–37, Dec. 2015. [Online]. Available: https://doi.org/10.1111/cgf.12605

[6] E. W. Bethel, H. Childs, and C. Hansen, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, 1st ed. Chapman & Hall/CRC, 2012.

[7] S. Marchesin, C. Mongenet, and J.-M. Dischler, "Dynamic load balancing for parallel volume rendering," in *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '06. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006, pp. 43–50. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV06/043-050

[8] 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*, vol. 14, no. 4, pp. 59–68, July 1994.

[9] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl, "Hierarchical visualization and compression of large volume datasets using gpu clusters," in *Proceedings of the 5th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004, pp. 41–48. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV04/041-048

[10] S. Guthe, M. Wand, J. Gonser, and W. Strasser, "Interactive rendering of large volume data sets," in *Visualization, 2002. VIS 2002. IEEE*, Nov 2002, pp. 53–60.

[11] K.-L. Ma, "Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures," in *Proceedings of the IEEE Symposium on Parallel Rendering*, ser. PRS '95. New York, NY, USA: ACM, 1995, pp. 23–30. [Online]. Available: http://doi.acm.org/10.1145/218327.218333

[12] K.-L. Ma and T. W. Crockett, "A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data," in *In Proceedings of 1997 Symposium on Parallel Rendering*, pp. 95–104.

[13] D. Steiner, E. G. Paredes, S. Eilemann, and R. Pajarola, "Dynamic work packages in parallel rendering," in *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization*, ser. EGPGV '16. Goslar Germany, Germany: Eurographics Association, 2016, pp. 89–98. [Online]. Available: https://doi.org/10.2312/pgv.20161185

[14] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A scalable parallel rendering framework," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 3, pp. 436–452, May 2009.

[15] C. Müller, M. Strengert, and T. Ertl, "Optimized volume raycasting for graphics-hardware-based cluster systems," in *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '06. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006, pp. 59–67. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV06/059-066

[16] S. Eilemann and R. Pajarola, "Direct send compositing for parallel sort-last rendering," in *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 29–36. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV07/029-036

[17] T. Peterka, H. Yu, R. Ross, and K.-L. Ma, "Parallel volume rendering on the ibm blue gene/p," in *Proceedings of the 8th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 73–80. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV08/073-080

[18] H. Yu, C. Wang, and K.-L. Ma, "Massively parallel volume rendering using 2-3 swap image compositing," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 48:1–48:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413419

[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*, Nov 2009, pp. 1–10.

[20] C. Montani, R. Perego, and R. Scopigno, "Parallel rendering of volumetric data set on distributed-memory architectures," *Concurrency: Practice and Experience*, vol. 5, no. 2, pp. 153–167, 1993.

[21] N. Max, P. Williams, C. Silva, and R. Cook, "Volume rendering for curvilinear and unstructured grids," in *Computer Graphics International, 2003. Proceedings*. IEEE, 2003, pp. 210–215.

[22] M. Larsen, S. Labasan, P. Navrátil, J. Meredith, and H. Childs, "Volume Rendering Via Data-Parallel Primitives," in *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, Cagliari, Italy, May 2015, pp. 53–62.

[23] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, K. Bonnell, M. Miller, G. H. Weber, C. Harrison, D. Pugmire *et al.*, "Visit: An end-user tool for visualizing and analyzing very large data," Oak Ridge National Laboratory (ORNL), Tech. Rep., 2011.

[24] "Mpi: A message passing interface," in *Supercomputing '93. Proceedings*, Nov 1993, pp. 878–883.

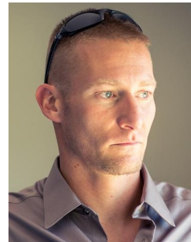
[25] B. W. O'Shea, G. Bryan, J. Bordner, M. L. Norman, T. Abel, R. Harkness, and A. Kritsuk, "Introducing Enzo, an AMR Cosmology Application," *ArXiv Astrophysics e-prints*, Mar. 2004.

[26] K. Moreland, C. Sewell, W. Usher, L. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci, "VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures," *IEEE Computer Graphics and Applications (CG&A)*, vol. 36, no. 3, pp. 48–58, May/Jun. 2016.

[27] J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros, "EAVL: The Extreme-scale Analysis and Visualization Library," in *Eurographics Symposium on Parallel Graphics and Visualization*, H. Childs, T. Kuhlen, and F. Marton, Eds. The Eurographics Association, 2012.

Roba Binyahib is a Ph.D. student in Computer Science at the University of Oregon. She is a member of the CDUX research group, supervised by Hank Childs. Before joining the University of Oregon, she received her Master's degree in Computer Science from King Abdullah University of Science and Technology, Saudi Arabia. Her research interests include high performance computing, scientific visualization, and large scale parallel rendering.

Tom Peterka is a computer scientist at Argonne National Laboratory, fellow at the Computation Institute of the University of Chicago, adjunct assistant professor at the University of Illinois at Chicago, and fellow at the Northwestern Argonne Institute for Science and Engineering. His research interests are in large-scale parallelism for in situ analysis of scientific data. His work has led to three best paper awards and publications in ACM SIGGRAPH, IEEE VR, IEEE TVCG, and ACM/IEEE SC, among others. Peterka received his PhD in computer science from the University of Illinois at Chicago, and he currently works actively in several DOE- and NSF-funded projects.



Matthew Larsen is a computer scientist at Lawrence Livermore National Laboratory. He received his Ph.D. in computer science from the University of Oregon in 2016. He is the primary developer for ALPINE's Ascent in situ library, as well as a key contributor to VTK-m and VisIt. Matthew's research interests include rendering for visualization, performance modeling for visualization, and many-core architectures.

Kwan-Liu Ma is a professor of computer science at the University of California, Davis, where he leads VIDI Labs and directs the UC Davis Center for Visualization. He received his PhD degree in computer science from the University of Utah in 1993. Professor Ma presently serves as the AEIC of IEEE CG&A. He is a fellow of IEEE.

Hank Childs is an Associate Professor in the Computer and Information Science Department at the University of Oregon. He received his Ph.D. in computer science from the University of California at Davis in 2006. Hank's research focuses on scientific visualization, high performance computing, and the intersection of the two.