Scalable Ray-Casted Volume Rendering

Roba Binyahib[*]
University of Oregon

Research Advisor: Hank Childs[†]
University of Oregon
Lawrence Berkeley Nat'l Lab

Abstract

Computational power has been increasing tremendously in recent years, resulting in an increase in data size and complexity. Volume rendering is an important method for visualizing such data, as it provides insight over the entire data set. However, traditional volume rendering techniques are not sufficient to handle such data because they are too large to fit in the memory of a single computer. Using a distributed system to visualize massive data improves the performance. That said, while parallelization has its benefits, it also creates challenges. There are two main approaches for parallel volume rendering: image order and object order. While these methods have advantages, they fail to achieve load balance in some cases. With our work, we present a hybrid parallel volume rendering algorithm that guarantees good performance and maintains load balance, since at each step of our hybrid approach the execution time is bounded. We compare our algorithm with the object order approach and demonstrate that our algorithm achieves performance and data scalability in cases where object order fails.

1 Introduction

The use of computer simulation continues to increase. Scientists use simulations to predict and understand the structure and behavior of important systems. For example, a simulation can be used to study reaction of chemicals, bacterial reproduction, or stars exploding. These simulations generate 3D data that contains critical and complex details that must be understood. Visualization is often used to present this data to provide insight to researchers. It is a way of simplifying the exploration of complex data, while maintaining accuracy.

Performance and scalability are important considerations for visualization algorithms. Traditionally, simulations on supercomputers have stored data at regular intervals and separate programs have loaded this data and visualized it, so called *post hoc* processing. In this paradigm, performance and scalability make interactivity possible, which is often a key factor in enabling insights. That being said, the push towards exascale is changing the balance of supercomputers. In particular, the rate at which supercomputers can generate data is going up much faster than the rate at which supercomputers can store and load data. As a result, visualization routines are increasingly being run *in situ*, meaning that these routines are run at the same time as the simulation, and they can access the data directly, instead of through a file system. In the *in situ* paradigm, failing to achieve performance and scalability can slow down the simulation code, making the entire supercomputer less efficient.

Volume rendering is an essential visualization algorithm. The basic idea behind volume rendering is to use a combination of color and transparency to allow all parts of a three-dimensional volume to be visible in a single image. Further, while many visualization algorithms transform data to create surfaces and then use traditional computer graphics approaches to render the surfaces, volume rendering incorporates the rendering portion directly into its algorithms. As a result, the performance characteristics of volume rendering are different as compared to most other visualization algorithms.

Parallel volume rendering is frequently used to visualize very large data sets that typically do not fit in the memory of a single computer. The two most studied approaches for parallel volume rendering are object-order (where the parallelization begins by iterating over cells from a mesh) and image-order (where the parallelization begins by iterating over pixels on the screen). While both of these approaches have been used successfully, they can become highly inefficient for some volume rendering workloads, in particular 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 slow down significantly, which has negative ramifications on both *post hoc* and *in situ* processing.

Childs et al. [3] introduced an algorithm in 2006 that showed promising results. Their algorithm improves on the object-order approach by detecting conditions for imbalance and adapting its execution. However, their study was lacking in several significant ways: (1) the benefits of the algorithm were not demonstrated by means of a comparison with the object-order approach, (2) it consumed significant amounts of memory, (3) it was written for single core CPUs, and (4) it considered scalability only up to 400 cores.

With our study, we revisit Childs et al.'s algorithm, making adaptations for modern supercomputers. Specifically, we introduce enhancements that minimize memory usage and work on many-core architectures, and we consider workloads that demonstrate the benefit of the approach over traditional methods. After the Directed Research Project, I will pursue the fourth limitation of their prior study (scalability up to only 400 cores), when I intern at Argonne National Laboratory and run experiments on their supercomputer. Ultimately, we will pursue a journal publication for this project, as an extension of the original work by Childs et al.

The contribution of our study is an improved algorithm beyond that of Childs et al. Specifically, our use of partial composites dramatically reduces memory usage, and our incorporation of many-core processing leads to a more efficient hybrid parallel solution. We demonstrate these benefits by running a performance study, and varying parameters such as data set, camera position, concurrency, image size, and transfer function. We compare the results with the traditional object-order approach. Our findings are that the new algorithm incurs some overhead and that it can be slightly slower with some visualization workloads. However, for workloads with high load imbalance, we find that our new algorithm is highly superior to the traditional approach.

2 Background and Related Work

2.1 Volume Rendering

Different visualization techniques can be used depending on the data structure and the purpose of visualization. There are two categories of 3D data visualization, surface rendering and volume rendering [2, 14, 30]. Surface rendering presents a surface of the data. This approach hides the interior of the data and can provide a limited insight. While volume rendering present the data as a whole, and different layers of the data can be visualized. Volume rendering is used when the transparency of the data is important (e.g., fog

[*]e-mail: roba@cs.uoregon.edu
[†]e-mail:hank@uoregon.edu

and human anatomy). For example, when visualizing a CT scan of a human head, if the purpose is to visualize one layer (the bone layer), using surface rendering would be sufficient. But if the goal is to present the different layers, for instance skin, bones, tissues, etc., then volume rendering would be superior.

Volume rendering produces 2D images of 3D volumetric data without extracting geometry explicitly. The main idea of volume rendering is to map a scalar field to color and opacity as an RGBA (red, green, blue, alpha) value using a transfer function. The transfer function maps numerical values within a certain range into RGBA values. It can be a curve, a linear function, or a random table.

There are two ways of performing volume rendering: image based and object based. In an image based approach, the algorithm loops over the pixels and computes RGBA values. The final RGBA value for a pixel is calculated by accumulating the RGBA values along that ray. While in an object-based approach, the algorithm loops over the cells and determines the impacted pixels. The RGBA value of a pixel is calculated by accumulating the contribution of different cells. While there are several volume rendering techniques, we focus on the ray casting technique in our algorithm.

2.1.1 Ray Casting

Ray casting [10] is an image based technique for volume rendering. It is a commonly used approach due to its simplicity and the quality of the results. In this technique, a ray is cast for each pixel into the volume. Then the data is sampled along the ray, and for each sample an RGBA value is computed using the transfer function. These values are accumulated to produce the final RGBA value of the pixel. The accumulation process can be performed either in a front-to-back order or in a back-to-front order. Our algorithm uses a front-to-back order, which uses the following equation for blending the final RGBA value:

$$C = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1 - A_i) \tag{1}$$

Where C is the RGBA value of the pixel, C_i is the color of the current scalar value at sample i, n is the number of samples along the ray, and A_i is the opacity at sample i.

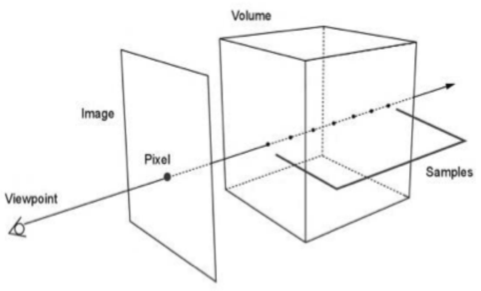

Figure 1: A single ray is cast into the volume [1].

2.2 Parallel Volume Rendering

Volume rendering is computationally expensive and it is challenging to achieve interactive frame rates. Several solutions use GPUs to perform volume rendering in parallel [11, 27]. But with tremendous increases in computational power, simulations are generating data sets that exceed the capacity of a single machine's memory. Thus, distributed memory solutions are increasingly needed. The main idea is to distribute work among different processors, perform some operation in parallel, then gather results into one image. There are two approaches for performing parallel volume rendering [4, 22]: image order and object order.

2.2.1 Image Order

The image pixels are distributed among processors and data are copied to each processor. Each processor samples the cells that contribute to its tile and apply ray casting to generate a sub-image. The sub-images are collected into one processor that acts as the server and the final image is produced. While image order is a ray coherent algorithm and thus samples along the ray can be composited without extra cost, the method has several disadvantages:

- When one tile contains more cells than the other tiles, then one processor performs more work (load imbalance)
- A cell can be covering more than one tile of the image resulting in replicating data over several processors

Moloney et al. [20] provides a solution to achieve data scaling using image order. Data is divided into equal blocks and each processor only loads the blocks that contributes to its view, thus reducing the amount of loaded data. Although this solution increases data scalability, load imbalance might still occurs when one tile has larger cells than others.

Image order approach has also been used for multi-projector rendering systems. Samanta et al. [28] render the volume onto multiple screens. Each processor is responsible for a projector. The problem occurs when the data assigned to one screen contains larger cells, resulting in one processor performing more work than others. To solve this, the authors distribute virtual tiles that can be of any shape and size. The distribution of these virtual tiles depends on the size and complexity of the data contributing to the tile. Processors sample and render their virtual tiles while maintaining load balance. Next, the resulting sub-images are exchanged depending on the actual screens assignments. Each processor displays its part of the scene.

2.2.2 Object Order

Data are divided into blocks and distributed among processors. Each processor samples its own cells, and the generated samples need to be arranged in a front-to-back order. Rays are composited producing the final image.

Object order has the advantage of distributing data among processors, which guarantees data scalability. Several solutions have been proposed. Ma et al. [16] present an algorithm that divides the data into sub-volumes distributed among processors, and each processor applies the ray-casting algorithm to its data. A parallel compositing is used to merge the results of each process into a final image. Wylie et al. [31] present an object order approach to achieve high rendering rates. Their solution divides the data between processors, then each processor performs rasterization on its data and stores the result in a frame buffer. Next, the frame buffer values are arranged in front-to-back order by performing a z-buffer comparison for each pixel. Finally, pixels are composited and the final image is generated. Strengert et al. [29] extend the advantage of data scalability by using hierarchical compression. They used a single GPU wavelet compression by Guthe et al. [9] to reduce the size of data blocks.

While the object order approach has high data scalability, it has some drawbacks.

- Ordering samples in front-to-back (i.e, image compositing) is the most expensive step of the object order and might become the bottleneck
- When dealing with unstructured data, one processor can own larger cells than the others. Or camera view might cover a region of the data that is owned by one processors. Both of these scenarios result in an unequal amount of work (load imbalance).

The previous methods [16, 29, 31] could lead to a load imbalance in cases similar to those mentioned above. Marchesin et al. [18] presents a solution to guarantee load balance. The data is divided into blocks and an estimation step of the rendering cost is performed. The transfer function and camera view are used to discard any blocks that are out of the camera view or has no opacity. The remaining blocks are then distributed among nodes and each node renders its data. Finally, binary swap [17] is used to composite the final image.

Ma [15] uses a graph-based data-partitioning software package to achieve equal sub-volume distribution among processors. While this approach maintains load balance for most cases, it fails to achieve load balance in cases where the camera is inside the volume. Muller et al. [23] demonstrated a dynamic load balancing technique, where the balance of each processor is calculated while sampling the cells. Blocks of data are transferred between processors, moving blocks to processors with less workload. The drawback of their method is the cost of moving data between processors. In addition it might still fail to maintain equal workload among processors, if the camera zoomed into a particular cell. This results in a heavier workload for one processor.

Our work solves this problem by deferring sampling of cells with large number of samples and distributes the workload among processors in a later phase.

2.2.3 Image Compositing

The final step in the object order approach is image compositing. This step orders the samples along each ray in front-to-back order. Image compositing is the most expensive step in the object order approach. Several techniques have been developed to perform image compositing and reduce its cost.

Direct send [5] is the simplest method, where all processors communicate with each other to acquire the required samples. Each processor owns a tile of the image and all other processors send the contributing samples to the owner processor. Another approach is binary swap [17]. 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 smaller by half in each round. In each round processors communicate in pairs, and processors swap pairs. Although, binary swap reduces network conjunction, it requires the number of processors to a be power of two and it has a synchronization overhead after each round. Direct send offers flexibility in terms of number of processors and has been used often in prior work [5, 26]. A modified version of the binary swap was implemented by Yu et al. [32] to overcome the limitation of the of number of processors. The algorithm allows any number of processors to be used and perform the communication in multiple rounds. These processors are divided into groups of twos or threes, and each group exchange data using the send direct approach. At the next round current image tiles are divided, and processors owning the same tiles on different groups exchange data. Another approach that combines the flexibility of the direct send and the performance of binary swap is radix-k [25]. This method divides the communication into multiple rounds, and defines a group size for each round k_i. The multiplication of the group sizes of all rounds is equal to N, where N is the number of processors. At each round, the current image tile is divided into k_i partitions. Processors of each group exchange the samples using a direct send method, with each processor owning a region of the current image tile.

2.3 Load Balancing in Parallel Systems

Load balancing is the process of maintaining equal amounts of work over all available processors at all times. Achieving load balance is the biggest challenge when performing parallel volume rendering. Load imbalance impacts the performance of the algorithm, since the execution time is determined by the time of the slowest processor. Additionally, load imbalance results in wasting resources that are being held. A hybrid solution that combines image order and object order approach can lead to load balance and better performance.

Childs et al. [3] used a hybrid approach to implement a highly scalable volume rendering algorithm. Their paper explored a predecessor idea to our own approach. Their algorithm consists of three steps. In the first step, the data is distributed among processors using the object order method. They use a cell classification step to determine the size of the cell. Small cells are sampled and large cells are deferred into a later stage. After sampling the small cells, the bounding box of each large cell is used along with the camera view to assign the large cells to processors. The second step is exchanging data (large cells and samples) using a direct send approach [5]. In the third step, each processor samples the large cells that it owns in an image order manner and composite its pixels to generate a sub-image. The sub-images are collected and the final image is generated. Our solution is similar to the previous work, in its three main phases. However, we reduce the memory cost by using partial composites, which is discussed in section 3.2. In addition, our study compares object order with hybrid. Finally, we use multi-threading for sampling and partial compositing to enhance the performance.

Another hybrid solution was demonstrated by Montani et al. [21]. Nodes are grouped into clusters and the image order approach is used to assign a tile for each cluster. The data is copied to each cluster and data are distributed among its nodes in an object order approach. Their solution helps to achieve data scalability by using object order on the nodes level and reduces the amount of data replication. Load imbalance can happen at the cluster level if one tile of the image has larger cells, causing a group of nodes to do more work. It might also occur among the nodes of a cluster, if the data assigned to one processor has a larger region of the tile than other processors. In our solution, we use a hybrid approach along with cell classification to maintain load balance.

2.4 Unstructured Data and Volume Rendering

An unstructured mesh [12] represents different cell types and sizes in an arbitrary order. Unlike structured data, cells connectivity is explicitly specified. There are some cases where unstructured meshes enable more accurate representation of the data. Since unstructured data does not have an indexing approach, volume rendering complexity increases. Both Ma [15] and Garrity [8] computed cell connectivity as a pre-step that is used to traverse points along a ray. The pre-step also determines the exterior faces (i.e, faces that are not shared between different cells). Ma [15] presented a solution for parallel volume rendering on unstructured data. Their solution used hierarchical data structure (HDS) to store face-node, cell-face, and cell-node relationships. Each processor performed the cell connectivity pre-step. Next, the camera view is used to exclude cells that are outside the camera view. Each processor applies the ray-casting algorithm to its data. The ray enters the volume from an exterior face and uses the connectivity information to determine the next cell. When the ray intersects with another exterior face, it exits the volume. Later, image pixels are assigned to different processors and HDS is used to exchange rays. Rays are composited to produce the final image. Max et al. [19] used a slicing based approach to render unstructured data. For each cell, three slices are generated perpendicular to the three X, Y, and Z axes. The camera view determines which of these slices is used. The value of each sample is computed by interpolating the cell vertices. This scalar value is used as 1D texture coordinates to obtain the color. Slices are rendered depending on their distance from the viewing direction in a back to front order. Colors are blended to generate the final pixel color.

Our solution adapts a many-core sampler implemented by Larsen et al. [13], where cells are sampled in parallel using multithreading. The samples of each cell are stored in a shared buffer. The index of each sample is calculated depending on its screen space coordinates (x, y and z). In the compositing stage the samples of each ray are 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.

3 Algorithm

This section presents our solution, which extends the approach by Childs et al. [3]. A hybrid approach is used to solve the problems associated with object order and image order techniques, and achieve load balance. The algorithm compares the number of samples per cell with a given threshold to determine if a cell is a small cell (SC) or large cell (LC). This cell assessment step is discussed further in section 3.1. This step results in two sampling steps: one is performed in object order and the other in image order. Sampling large cells is deferred to the image order phase to maintain load balance and decrease execution time. In addition, we use partial composites to reduce the memory and communication cost. This concept is discussed in section 3.2.

Our algorithm consist of three main phases:

1. Object Order Phase
   Data are distributed among processors and each processor samples its small cells.
2. Exchange Phase
   The algorithm assigns image pixels to the different processors. Using the pixel assignment and the cell screen space coordinates, the destination processor for each cell is determined and data are exchanged. Partial composites are used to reduce the size of exchanged data, which is discussed in more details in section 3.2.
3. Image Order 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 presented in Figure 2.

3.1 Cell Assessment

The first step of the algorithm is to exclude cells that are outside the camera view. This is done by considering the camera view, the coordinates of the cells and the number of samples per ray. Our hybrid approach also uses this information to classify cells as small or large by comparing the number of samples with a given threshold. This classification is used to achieve load balance by deferring the sampling of large cells. The value of this step rises when one of the processors has more samples than the rest. Meaning there is a processor performing more work, while other processors are idle. This scenario can occur in two different cases. The first takes place when the camera is zoomed into a specific region of that data, which is owned by one processor. The second occurs when rendering unstructured data with different cell sizes, which may result in assigning large cells into one processor. Both cases result in load imbalance, causing the execution time to increase. To achieve load balance, sampling large cells is deferred to the image order phase, thus distributing the cost of sampling among different processors.

3.2 Partial Composites

Partial composites is a technique to reduce memory and communication cost. It reduces the size by compositing a group of floating points into 24 bytes representing color, opacity and depth information. The transfer function is used to assign a color and opacity

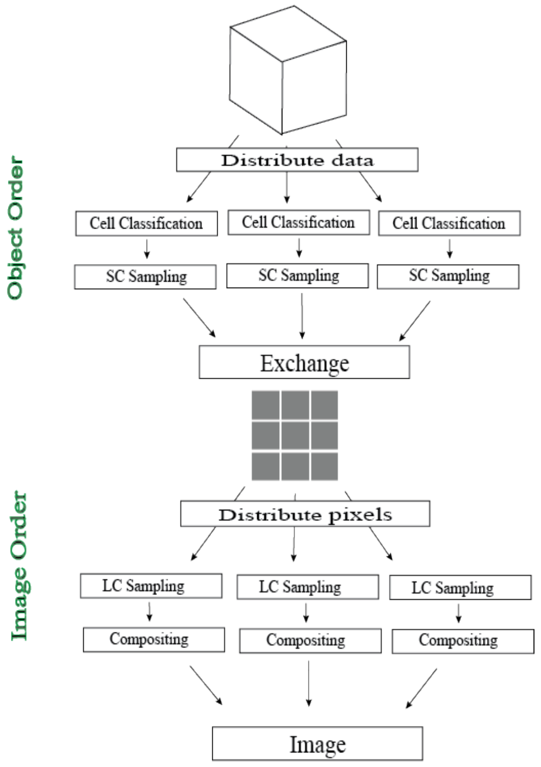

Figure 2: The algorithm pipeline highlight the three main phases, object order, exchange, and image order phase.

to each sample. Next, these colors are composited in front-to-back order using the following equation:

$$C = \sum_{i=0}^{n} C_i * Al_i \prod_{j=0}^{i-1} (1 - O_i) \tag{2}$$

Where C is the RGB value of the partial composite, C_i is the color of the current sample, Al_i is the opacity of the sample, n is the number of samples along the ray and O_i is the accumulated opacity.

Consider an example. Let $P0$ be the first processor, Z be the depth of the samples, n be the number of samples per ray, and the value of each sample be a floating number of size 4 bytes. If $n = 1000$ and $P0$ has the samples for $Z = 3$ to $Z = 104$, that will cost $100 * 4 = 400$ bytes. Compositing these samples into a color RGB and opacity A will result in storing 4 floating point values. Two additional integers are used to keep track of 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. $Z_e = 104$). This will result in 4 floating point values and 2 integer point values instead of a 100 thus reducing the cost from 400 bytes to 24 bytes, which benefits both communication and memory.

3.3 Object Order Phase

Each processor classifies its cells into small and large cells and samples its small cells.

3.3.1 Small Cell Processing

This step contains two sub-stages: 1) sampling small cells and 2) generating partial composites along each ray. We adapt a multithreaded sampler implemented by Larsen et al.[13] to perform the sampling. Parallelism is applied by working on different cells simultaneously. Cells are sampled and the values are stored in a sample buffer.

Next, partial composites are generated for each pixel by working on different pixels simultaneously. For each group of continuous samples along the z axis, samples are composited into a partial composite, as described in section 3.2.

3.4 Exchange Phase

In this phase, two types of data are being exchanged: 1) large cells and 2) partial composites. In preparation for the image order phase, image tiles are assigned to different processors and each processor receives the data contributing to its tile.

Processors determine the destination of the data and pack the messages into a buffer. Then, processors perform the exchange using **Message Passing Interface** (MPI) [7]. Each processor receives its data and unpacks the messages into a defined structure (partial composite or large cell).

3.5 Image Order Phase

After the exchange phase, each processor has all the data contributing to its tile of the image. This phase employs three consecutive sub-phases.

3.5.1 Large Cell Processing

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 uses its tile coordinates to sample only the contributing part of the cell. This reduces redundant sampling that can be performed by different processors. After sampling, partial composites are generated and stored. Now each processor has the data it needs to generate its sub-image.

3.5.2 Compositing

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, as described in section 3.2, are used to order the partial composites. Finally, these partials are combined in a front-to-back order using the following equation:

$$C = \sum_{i=0}^{np} C_i \prod_{j=0}^{i-1} (1 - O_i) \tag{3}$$

Where C is the RGB value of the pixel, C_i is the color of the current partial, np is the number of partial composites along the ray and O_i 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 2.

3.5.3 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.

4 ANALYSIS

4.1 Performance

In this section we compare the performance of the algorithm between the Object Order (OO) and our Hybrid approach (HB).

We define the following abbreviations, which we use in the rest of the paper:

- C_i: The number of cells owned by processor i
- SPC: The number of samples per each cell
- SC_i: The number of small cells owned by processor i.
- S: The number of samples per ray
- PC_i: The number of partial composites owned by by processor i
- LC_i: The number of large cells owned by by processor i
- P: The total number of pixels
- N: The total number of processors

The first step for both algorithms is cells assessment discussed in section 3.1. The time of this step is bounded by (C_i). The second step is sampling, and the performance of this step varies between the two approaches. In the OO, all cells are sampled, thus the performance depends on the number of cells owned by the processor (C_i) and also the number of samples per each cell (SPC). While in the HB only small cells are sampled and thus the time is bounded by the number of small cells owned by the processor (SC_i). Since the number of samples for each sampled cell is small, it does not have an impact on performance. Partial compositing time is proportional tp (S) for OO, while for HB the execution time is bounded by the number of small cells. Both approaches exchange partial composites (PC_i). HB has an additional cost in the exchange phase, which is the cost of exchanging large cells (LC_i). Thus the cost of the exchange phase is lower for OO than HB. HB also has two additional steps: large cell sampling and another partial composites for processing these large cells. As mentioned in section 3.5.1, these two steps are performed in image order. Thus the performance depends on the number of pixels (P) multiplied by the number of samples per ray divided by the number of processors (N). Finally, both approaches perform the final composite producing the final image and the cost is the same for both.

Table 1, summarizes the performance of each step for the two approaches.

Table 1: Performance of Object Order and Hybrid approaches

Steps	OO	HB
Cells Assessment	$O(C_i)$	$O(C_i)$
SC Sampling	$O(C_i) + O(SPC)$	$O(SC_i)$
Partial Composite	$O(S)$	$O(SC_i)$
Exchange	$\sum_{i=0}^{n} PC_i$	$\sum_{i=0}^{n} PC_i + \sum_{i=0}^{n} LC_i$
LC Sampling	–	$O(\frac{P*S}{N})$
Partial Composite	–	$O(\frac{P*S}{N})$
Final Compositing	$O(\frac{P}{N})$	$O(\frac{P}{N})$

4.2 Benefits and Drawbacks

Both approaches have pros and cons, and their three main differences are listed in Table 2. The first difference is that OO does not perform large cell exchange. The second difference is that OO has fewer partial composites to exchange due to its sampling strategy. In OO, all cells are sampled and thus a processor might be able to composite all the samples in a ray into one partial composite. While in HB, small and large cells are sampled in different steps, producing more partial composites along the ray. The third difference is load imbalance: OO approach can have a high load imbalance but HB can prevent load imbalance. Although the first two points are advantages for object order and they result in lower exchange cost than hybrid, using hybrid can result in lower execution time and load imbalance in some cases. Examples include, if the camera is zoomed-in to one of the regions owned by one processor, or dealing it is an unstructured mesh has an equal cell sizes. In these cases, one processor is performing most of the work, resulting in high load imbalance and long execution time. Thus, using a hybrid approach and paying the additional cost of exchange, can result in much lower execution time and better load balance.

Table 2: Differences between Object Order and Hybrid

	OO	HB
No LC Exchange	+	−
Fewer PC Exchanged	+	−
Handling Load imbalance	−	+

5 Experiment Overview

This section describes the details of our experiment.

5.1 Factors

We explored the following four factors in our study.

- Algorithm: The main purpose of our algorithm is to achieve load balance and reduce execution time We compare the performance and load balance of our hybrid approach with the object order method.
- Data set: Aligned with our motivation of implementing a parallel volume rendering for large data sets, we test the performance using different data sizes.
- Parallelism: Our algorithm is designed to achieve high performance and load balance on a distributed system, so we test the performance scaling on two machines.
- Camera position: Our algorithm uses the camera view in the sampling step. Thus we study the impact of camera position on load balance and execution time. For each position the execution time is measured for the exchange step, the sampling step, and the total execution time.
- Image Size: The image size affects both sampling and compositing. We measure the execution time for different image sizes.
- Transfer function: The algorithm excludes cells that have zero opacity value. We study the effect of different opacity values on the execution time.

5.2 Experiment Configuration

Our experiment vary the six factors and test them by running 66 experiments in six phases.

- Algorithm (2 options)
- Parallelism (8 options)
- Data set (4 options)
- Camera position (3 options)
- Image size (4 options)
- Transfer function (5 options)

Each of these factors and their options are discussed in the following subsections.

5.2.1 Algorithm

We performed our experiment using hybrid and object order methods.

- Hybrid: The algorithm described in section 3.
- Object order: The algorithm described in section 2.2.2.

5.2.2 Parallelism

We test the performance scaling of our algorithm varying the number of nodes as the following: 1, 2, 3, 4, 8, 12, 16, 32, 64, and 128 nodes. We used Twelve threads per node, and 1 MPI task per node.

5.2.3 Data set

- Enzo-1M: A cosmology data set from the Enzo simulation code [24]. 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.
- Nek-50M: An unstructured mesh that contains 50M tetrahedrons from a Nek5000 [6] thermal hydraulics simulation.

Figure 3, shows a rendering of both data sets.

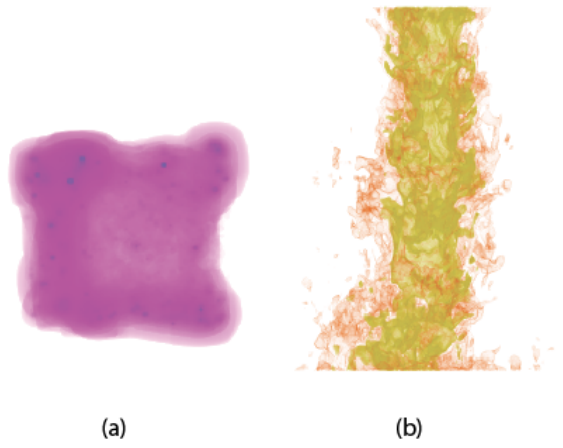

Figure 3: The data set used in the experiment, (a)Enzo, (b)Nek.

5.2.4 Camera position

We compare the performance and load balance of hybrid and object order methods for three different camera positions. The variation of camera positions demonstrates the points discussed in section 4.

- Zoom-in
- Mid-zoom
- zoom-out

Figure 4, shows the different camera positions.

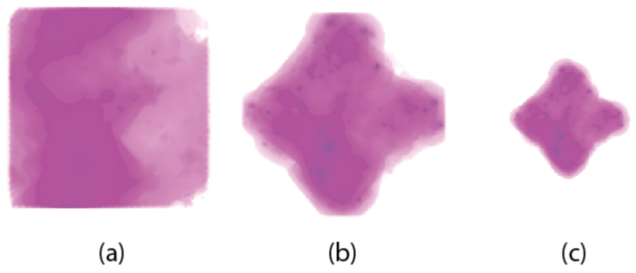

Figure 4: The three camera positions, (a)zoom-in, (b)mid-zoom and (c)zoom-out

5.2.5 Image Size

Four image sizes are used to measure the impact of the size on the execution time for both methods.

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

5.2.6 Transfer function

The opacity of the transfer function varies as the following:

- Very Dense (highest opacity)
- Dense
- Middle
- Light
- Very Light (lowest opacity)

The rendering of the different transfer functions is presented in figure 5.3.

5.3 Performance Measurements

We measure the execution time (in seconds), and we calculate the load imbalance for each test with the following equation:

$$\text{Load imbalance} = \frac{\mathrm{T_s}}{\mathrm{TA}}$$

Where T_s is the execution time of the slowest processor, and T_A is the average execution time of overall processors.

For some of the experiments, we also present the time for different phases of the algorithm.

We run our experiments on following the two machines:

- Alaska: A University of Oregon machine. Four cluster nodes each has an Intel Xeon E5-1650v3 CPU running at 3.5GHz, containing 6 physical cores and 12 threads.
- Edison: One of NERSC's machines. Each node contains two "Ivy Bridge" CPUs running at 2.4 GHz, each CPU containing 12 physical cores and 24 threads.

6 Results

In this section, we present and analyze the results of our experiment.

6.1 Phase 1: Base Case

Our base case tests the performance and load balance of our hybrid algorithm and compares it with the object order method.

This phase use the following configuration:

- Enzo-1M data set
- 4 nodes, 1 MPI task per node
- 12 cores per node for multi-threading
- Machine: Alaska
- Fixed camera position (zoom-in)
- Image size 1024×1024
- 1000 samples per ray
- Medium opacity for transfer function

Table 3 presents the time (in seconds) for each step and load imbalance for both algorithms.

The object order approach suffered from load imbalance when the camera view is focused on the data of one processor. We test this case here and show that our hybrid approach achieves load balance. The results presented in Table 3 demonstrate the advantages of the hybrid approach over object order. Since object order samples all the cells in one phase (SC processing), one processor is

Table 3: A comparison of the performance of hybrid and object order approach.

	OO	HB
SC Processing	6.41	0.100
Exchange LC	–	0.11
LC Processing	–	1.60
Exchange PC	0.12	0.09
Compositing	0.004	0.005
Collect	0.003	0.003
Other	0.05	0.48
Total Time	6.59	2.40
Load imbalance	3.47	1.12

taking 6.41 seconds while others are much faster. In our hybrid approach, the processor only processes small cells and defers processing large cells into the image order phase (LC processing). This results in reducing the sampling time from 6.41 to 1.7 seconds (total of both sampling steps). While the hybrid approach has an extra exchange cost, the total execution time is less than the object order. In addition it reduces the load imbalance from 3.47 to 1.12.

6.2 Phase 2: Distributed Memory Scalability

This phase use the following configuration:

- Enzo-1M data set
- 12 cores per node for multi threading
- Machine: Alaska and Edison
- Two camera position: zoom-in and mid-zoom
- Image size 1024×1024
- 1000 samples per ray
- Medium opacity for transfer function

Table 4: Distributed Memory Alaska mid-zoom

	1	2	3	4
OO Time	7.15	4.80	3.87	2.98
OO Speedup	1	1.48	1.84	2.39
HB Time	7.15	4.82	3.91	3.10
HB Speedup	1	1.48	1.82	2.30

Table 4 presents the execution time and the speedup for both approaches for a mid-zoom camera position. The speedup for both methods is comparable. In this mid-zoom case, some of the cells are large and thus they are sampled in the image order phase. While others are small and sampled in the object order phase.

Table 5: Distributed Memory Alaska zoom-in

	1	2	3	4
OO Time	6.49	6.49	6.71	6.59
OO Speedup	1	1.00	0.96	0.98
HB Time	6.49	3.46	2.67	2.40
HB Speedup	1	1.87	2.43	2.70

The results of Table 5 demonstrate one of the cases mentioned earlier in section 3.3, where the camera is zoomed into the volume. The results shows that object order is taking much more time in the zoom-in camera position because there is one processor always doing more work. Adding more processors harms the algorithm since the workload is not being distributed and the cost of exchanging partials increases.

Table 6 shows the results of running on a higher scale concurrency by running on Edison. Although the algorithm outperforms

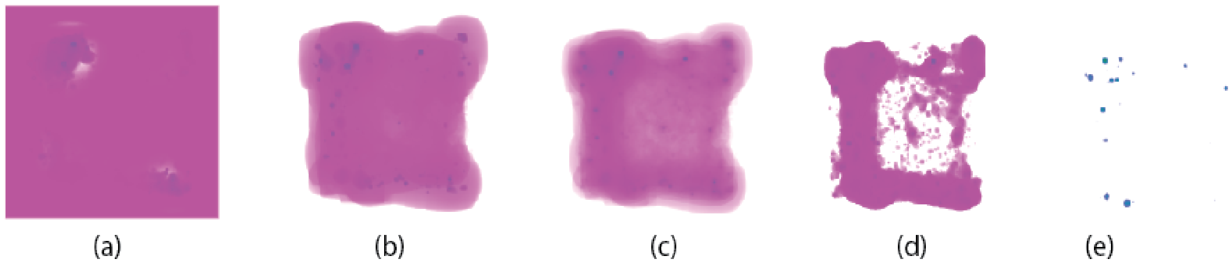

Figure 5: Different transfer function opacity, starting with (a) the highest opacity to (e) the lowest.

Table 6: Distributed Memory Edison zoom-in

Num of proc	OO Time	OO Speedup	HB Time	HB Speedup
1	8.97	1	8.97	1
2	8.40	1.06	4.33	2.06
4	8.45	1.06	2.84	3.15
8	6.03	1.48	1.89	4.73
16	3.53	2.53	1.09	8.19
32	3.86	2.32	0.84	10.64
64	4.29	2.08	0.52	17.01
128	4.11	2.17	0.38	23.52

the object order approach at small scale, it is designed to achieve even better performance at large scale. We plan to take this experiment further and run on more resources in the coming months at Argonne National Laboratory. We expect to achieve even better results at larger scale.

6.3 Phase 3: Camera Position

This phase use the following configuration:

- Enzo-1M data set
- 4 nodes, 1 MPI task per node
- 12 cores per node for multi threading
- Machine: Alaska
- Image size 1024×1024
- 1000 samples per ray
- Medium opacity for transfer function

Table 7: Camera Position

	Zoom-In		Mid-Zoom		Zoom-Out	
	OO	HB	OO	HB	OO	HB
Exchange Time	0.12	0.20	0.20	0.35	0.14	0.14
Sampling Time	6.41	1.7	2.76	2.59	1.05	1.05
Total Time	6.59	2.40	2.98	3.10	1.25	1.25
Load Imbalance	3.47	1.12	1.37	1.04	1.13	1.12

The results presented in Table 7 align with the discussion in section 4.2. The table shows that for the zoom-in case the load imbalance is high for object order. Our hybrid approach reduces the load imbalance and execution time. While in cases where the camera is mid-zoom and load imbalance is low, paying the extra exchange cost in hybrid result in higher execution time even thought it reduces the load imbalance. Finally, in the zoom-out case both approaches give comparable results since the number of samples per cell is small and thus all cells are sampled in the first phase.

6.4 Phase 4: Data Set

This phase use the following configuration:

- 4 nodes, 1 MPI task per node
- 12 cores per node for multi threading
- Machine: Alaska
- Image size 1024×1024
- 1000 samples per ray
- Medium opacity for transfer function

Table 8: Data Set

Data Set	#Cells	OO	HB	Camera
Enzo-1M	1.25M	6.59	2.40	near
Enzo-10M	10.2M	7.31	5.72	near
Enzo-80M	83.9M	7.88	7.83	near
Nek	50M	5.36	4.31	near

Table 8 demonstrates that using hybrid approach reduces the execution time for both small and large data sets. As the number of cells increases for the Enzo data set, the size of the cells become smaller. This explains the decrease in the difference between the performance of the hybrid and object order. In the smaller version Enzo-1M, the cells are larger so the hybrid algorithm defer sampling these cells into the image order approach. While in the larger versions, the size of cells gets smaller so most of the cells are sampled in the object order phase of our algorithm.

6.5 Phase 5: Image Size

This phase use the following configuration:

- Enzo-1M data set
- 4 nodes, 1 MPI task per node
- 12 cores per node for multi threading
- Machine: Alaska
- Fixed camera position (zoom-in)
- 1000 samples per ray
- Medium opacity for transfer function

Table 9 shows that our hybrid approach still maintains a lower execution time than object order when the image size increases. The image size information is used in the sampling step, and since our hybrid approach distributes the cost of sampling among processors

Table 9: Image Size

Image Size	OO	HB
100×100	0.15	0.24
200×200	0.41	0.35
500×500	1.41	0.75
1024×1024	6.59	2.40

it achieve better performance. As the size of the image increases the number of samples per cell increases, which means more larger cells. 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 algorithm samples most of the cells in the object order phase. With the increase of the image size the hybrid approach samples more cells in the image order phase.

6.6 Phase 6: Transfer Function

This phase use the following configuration:

- 4 nodes, 1 MPI task per node
- 12 cores per node for multi threading
- Machine: Alaska
- Fixed camera position (zoom-in)
- Image size 1024×1024
- 1000 samples per ray

Table 10: Transfer Function

Transfer Function	OO	HB
Most Dense (highest opacity)	7.78	3.20
Dense	7.51	3.06
Middle	6.59	2.40
Light	5.15	2.18
Most Light (lowest opacity)	5.05	2.10

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 10 shows that both approaches increase proportionally as the opacity increases.

7 Conclusions and Future Work

We implemented a scalable ray casting algorithm that achieves load balance and reduces execution time. These are the main contributions of our algorithm:

- We improve load balance even for cases where the camera is inside the volume
- We compare our hybrid approach with the object order approach and demonstrate that our method achieves higher performance and load balance
- Our algorithm reduces communication and memory cost by using partial composites
- We adapt a many-core sampler to increase the performance

Our algorithm is intended for high scale rendering, but we performed our experiments on the resources we have currently. As a future work, we plan to study the performance of our algorithm on a higher scale using Argonne National Laboratory resources. We expect the performance to increase at higher scale. We also plan to extend our algorithm to include more image compositing methods.

References

[1] U. Ayachit, B. Geveci, K. Moreland, J. Patchett, and J. Ahrens. The ParaView Visualization Application. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, pages 383–400. Oct. 2012.

[2] E. Bethel, H. Childs, and C. Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2012.

[3] H. Childs, M. Duchaineau, and K.-L. Ma. A scalable, hybrid scheme for volume rendering massive data sets. pages 153–161, 2006.

[4] 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*, EGPGV '07, pages 29–36, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[5] 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*, EGPGV '07, pages 29–36, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[6] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier. nek5000 Web page, 2008. http://nek5000.mcs.anl.gov.

[7] M. P. Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[8] M. P. Garrity. Raytracing irregular volume data. In *Proceedings of the 1990 Workshop on Volume Visualization*, VVS '90, pages 35–40, New York, NY, USA, 1990. ACM.

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

[10] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[11] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 38–, Washington, DC, USA, 2003. IEEE Computer Society.

[12] H. Kutluca, T. M. Kurç, and C. Aykanat. Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids. *J. Supercomput.*, 15(1):51–93, Jan. 2000.

[13] 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)*, pages 53–62, Cagliari, Italy, May 2015.

[14] M. Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, May 1988.

[15] K.-L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *Proceedings of the IEEE Symposium on Parallel Rendering*, PRS '95, pages 23–30, New York, NY, USA, 1995. ACM.

[16] K. L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. A data distributed, parallel algorithm for ray-traced volume rendering. In *Proceedings of the 1993 Symposium on Parallel Rendering*, PRS '93, pages 15–22, New York, NY, USA, 1993. ACM.

[17] 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 1994.

[18] 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*, EGPGV '06, pages 43–50, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

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

[20] B. Moloney, M. Ament, D. Weiskopf, and T. Moller. Sort-first parallel volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1164–1177, Aug 2011.

[21] C. Montani, R. Perego, and R. Scopigno. Parallel rendering of volumetric data set on distributed-memory architectures. *Concurrency: Practice and Experience*, 5(2):153–167, 1993.

[22] C. Mueller. The sort-first rendering architecture for high-performance graphics. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, I3D '95, pages 75–ff., New York, NY, USA, 1995. ACM.

[23] 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*,

EGPGV '06, pages 59–67, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[24] 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.

[25] 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*, pages 1–10, Nov 2009.

[26] 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*, EGPGV '08, pages 73–80, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[27] D. Ruijters and A. Vilanova. Optimizing gpu volume rendering. 2006.

[28] 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*, HWWS '99, pages 107–116, New York, NY, USA, 1999. ACM.

[29] 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*, EGPGV '04, pages 41–48, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[30] A. C. Telea. *Data Visualization: Principles and Practice, Second Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2014.

[31] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland. Scalable rendering on pc clusters. *IEEE Computer Graphics and Applications*, 21(4):62–69, Jul 2001.

[32] 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*, SC '08, pages 48:1–48:11, Piscataway, NJ, USA, 2008. IEEE Press.