Histogram-Based I/O Optimization for Visualizing Large-Scale Data

Yuan Hong
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210-1277
hongy@cse.ohio-state.edu

Tom Peterka
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439
tpeterka@mcs.anl.gov

Han-Wei Shen
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210-1277
hwshen@cse.ohio-state.edu

ABSTRACT

We present an I/O optimization method for parallel volume rendering based on visibility and spatial locality. The combined metric is used to organize the file layout of the dataset on a parallel file system. This reduces the number of small, noncontiguous I/O operations and improves load balance among I/O servers. The net result is reduced I/O time. Since large-scale visualization is data-intensive, overall visualization performance improves using this method. This paper explains the preprocessing of data blocks to compute feature vectors and the storage organization based on them. Run-time performance is analyzed with a variety of transfer functions, view directions, system scales, and datasets. Our results show significant performance gains over file layouts based on space-filling curves.

Categories and Subject Descriptors

E.1 [**DATA STRUCTURES**]: Distributed data structures

General Terms

Histogram Parallel I/O

1. INTRODUCTION

The overall performance of large-scale parallel algorithms is bound by the cost of data movement. Frequent I/O accesses are needed to load data that is too large to fit in memory, or that consists of multiple time-steps. As I/O bandwidth saturates, scientists cannot visualize large-scale results fast enough, and timely results are critical to many applications such as on-demand query and real-time visualization.

Examples of sparse data traversal such as visibility culling can be used to optimize large-scale data exploration, but sparse traversal can produce a large number of small I/O operations if not accompanied by I/O organization. There has been little research to address this problem so far. Space-filling curves such as the Hilbert curve, Z-curve and others can ameliorate this situation, although their original purpose is to balance working load among processors. In our research we found that the effectiveness of space-filling curves is limited because they only consider spatial locality, without taking visibility into account. Their overall effectiveness diminishes as the number of blocks per processor decreases; hence, they do not scale well with high numbers of processors.

In this paper we present a method to optimize I/O performance for parallel volume rendering by using a more complex metric that considers both visibility and spatial locality. Moreover, the technique presented in this paper also considers the data access patterns resulting from visibility culling and optimizes the file layout accordingly. We first perform visibility preprocessing on data blocks independent of view conditions and transfer functions. We then organize the data blocks in storage based primarily on the visibility feature vector, and secondarily on spatial locality when blocks have the same visibility feature vector. By integrating visibility culling with I/O organization, our method helps reduce I/O time and balances load among I/O servers.

We conducted a series of experiments that show that our method scales better than space-filling curves with the number of processors. The tests show that the performance improves over a large number of view directions, transfer functions, and time steps. Our contributions are reduced I/O time during rendering, independent of transfer functions and view parameters. We also provide heuristics about tuning parameters of our algorithm such as the block size, stripe size, and number of view samples.

2. RELATED WORK

Related literature includes visibility culling for rendering, out-of-core methods related to storage optimization, and collective I/O optimization.

2.1 Visibility Culling for Rendering

Early ray termination is a simple method for culling occluded data in volume rendering. When accumulating opacity from front to back, a ray can be terminated prematurely once a threshold opacity is attained. No further data points are visible beyond that point, so they are not considered. The effect of early ray termination is limited in a parallel environment. In [9], the efficiency of early ray termination drops by approximately 30% as the number of processes scales up.

More effective culling approaches for parallel volume algorithms exist. Ma and Crockett [6] studied parallel rendering of unstructured grid data using cell projections. To further improve performance, the authors culled portions of the volume early in the visualization pipeline. In [17], hierarchical occlusion maps (HOM) were

UltraVis Workshop '09 Portland, Oregon USA

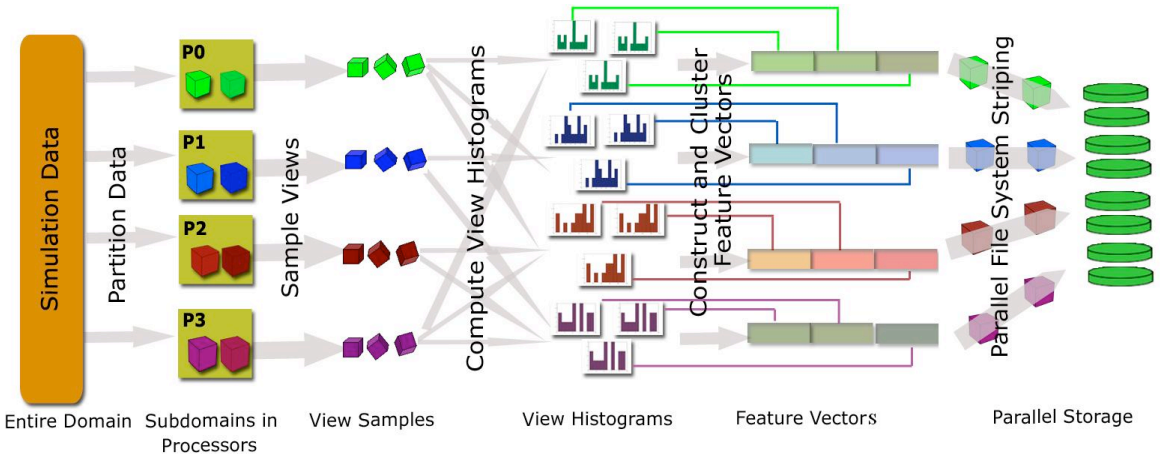


Figure 1: The overview of our algorithm

introduced for visibility culling on complex models with high depth complexity, using hierarchies in object space and image space. CPU-based culling methods were also presented in [2]. Liu et al. [5] described a progressive view-dependent isosurface extraction algorithm. This approach determines visible voxels by casting a small number of viewing rays and then propagating the visibility information up from these seed voxels to obtain the full visibility for the volume. Gao et al. [1] proposed a scalable visibility culling method based on plenoptic opacity functions (POFs). A POF performs well if the transfer functions are known or can be derived from a small set of base transfer functions. The authors assume that the transfer functions can be expressed by a linear combination of the corresponding bases. But in some cases, as in [16], transfer functions can be generated from nonlinear combinations of the existing transfer functions as well.

2.2 Out-of-core Methods

In [12], authors reviewed several principle out-of-core algorithms for scientific visualization. Among the research on out-of-core algorithms, Pascucci and Frank [8] defined hierarchical indices over very large regular grids, leading to efficient disk layout. Their approach is based on the use of the Lebesgue curve for defining the data layout and indexing. They demonstrated their approach in a progressive slicing of very large multi-resolution datasets. Isenburg et al. [4] described a streaming format for polygon meshes to replace offline mesh formats for large datasets. It is an input and output format that processes meshes in a streaming, or pipelined fashion.

2.3 Collective I/O Optimization

Besides being an important computational tool, parallelism is the basic vehicle for accelerating I/O throughput. Parallel application programs access parallel file systems through MPI-IO calls such as MPI_ File_read_all. In order to reduce the number of disk accesses, techniques such as two-phase I/O, data sieving and list I/O aggregate smaller accesses into a single larger access [10, 14].

The tools that are built into a parallel I/O system cannot guarantee good I/O performance alone. Intelligent algorithms that organize data effectively are also needed. File layouts and supporting systems are tuned in [3] to achieve higher performance. Wang et al. [15] present a profile-guided greedy partitioning algorithm to parallelize I/O access for file-intensive applications on cluster-based systems. Smirni et al. [13] analyzed the I/O behavior of scientific applications and their interactions with the file system. They concluded that tuning of file system policy parameters to match I/O demands can significantly increase I/O throughput.

3. METHOD

3.1 Overview

Our method has five steps, and Figure 1 shows the overall process. The first step is to divide the data into uniform-size blocks and allocate blocks to processors. In steps 2 and 3, multiple view directions are sampled and a view histogram is constructed for each view of each block. In the fourth step, feature vectors are constructed for each block by concatenating the view histograms. Clustering is performed on these feature vectors in order to group blocks with similar visibility together. The last step is to organize the data blocks into storage based on the clustering results. Within a cluster, blocks are further organized by spatial locality. All five steps are performed in preprocessing, before running the volume rendering. They can be performed efficiently in parallel. Figure 2 shows the preprocessing time on the IBM Blue Gene/P (BG/P) for one of our test datasets: a supernova that has a size of 276GB and 1024 sampled views. The total preprocessing time scales with the number of processors.

The steps in Figure 1 are explained in more detail in the following subsections.

3.2 View Histogram

A view histogram is the signature of a block's visibility for a particular view direction. Two blocks with similar view histograms will have similar visibility in this view direction irrespective of transfer function. The view histogram is computed once during preprocessing and is used with multiple transfer functions during volume rendering. The view histogram is computed as follows.

Each processor has a number of data blocks in its subdomain, and we use the term "target block" to indicate the current block in question. For each pixel in the projection of the target block, a ray is cast through the blocks in front of the target block. A histogram is constructed from the frequency of data values along this ray. These histograms are averaged over all of the pixels in the projection, resulting in a single view histogram for each sampled view direction.

A block's visibility in a given view is determined by the accumu-

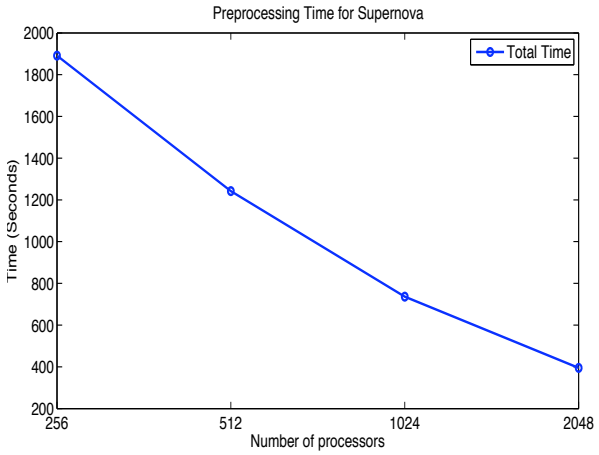


Figure 2: The total preprocessing time for supernova data

lated data from the front blocks, so we can calculate the histograms along the casting rays in front-to-back order and stop when we reach the target block. Equation 1 (Max [7]) expresses accumulated opacity along a viewing ray and shows that the final accumulated opacity does not depend on the sampling order.

$$\alpha = 1 - \prod_{i=1}^{m} (1 - \overline{\alpha}(S_i))^{k_i} \quad (1)$$

where m is the number of unique values sampled along the ray, $\overline{\alpha}(S_i))$ is the corresponding opacity for the scalar value S_i, and k_i is the number of sample points that have the scalar value S_i.

We apply a similar approach to computing view histograms in parallel. Each block is handled independently by the processor that owns it and the local histograms are then communicated among processors to build view histograms. Recall that the view histogram is only for a single view direction, but in the following, we describe how the view histograms for multiple views can be used to construct a block's visibility feature vector.

3.3 Visibility Feature Vector

A visibility feature vector is the descriptor of the data values in the block for all sampled view directions. In our implementation, view directions are taken at uniform points on a view sphere. Subsection 3.3.1 describes how to determine an appropriate number of sample views to use. The resulting histograms are concatenated into a feature vector for one block, and each element of the feature vector is one of the view histograms. This is repeated for all blocks in the data. Blocks with similar feature vectors will have similar visibility in most view directions, and Subsection 3.3.2 explains how to determine similarity among feature vectors.

3.3.1 View Direction Sampling

This section examines how to determine an appropriate number of view directions to sample. We use an iterative method that samples increasing numbers of views and terminates when the variance across view directions is small enough. In one iteration, view histograms are computed for the sampled views, and the variance is calculated across all the views for each bin in the histogram. The variances are summed across all bins in the histogram and averaged across all the blocks. This yields a single variance for the iteration, which represents the average histogram variance. The number of view directions taken from the view sphere quadruples with each iteration and the process repeats. When the change in variance is below a threshold, the process stops. In general, the number of view samples is data-dependent. In our test datasets, the variance stabilized after at most five iterations, or 1024 view samples.

Figure 3 shows the variance with respect to the number of sampled views for one of our test datasets. The variance increase is less than 10^{-5} when the number of sampled views equals 256. Sampling more views is unnecessary for this dataset.

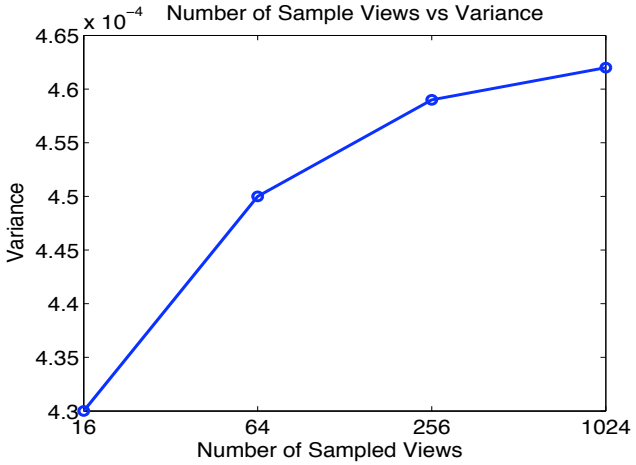


Figure 3: The variance as a function of the number of sampled views for Viswoman dataset. The variance changes slowly after the number of sampled views approaches 256.

3.3.2 Distance Metrics

Next we examine how to cluster similar feature vectors together; in particular, what distance metric should be used to do so. To compute the difference between two view histograms, the traditional Euclidean distance does not suffice because it is a flat metric that does not consider the shape of the vectors, although the vector shape influences the computed visibility. Instead, we use the Earth Mover's Distance (EMD) [11] so that the difference between two vectors accounts for the distribution as well as the actual histogram values. When using EMD, the ground distance matrix, C, must be defined.

For two feature vectors $V_1 = [p_1, p_2, ..., p_n]$ and $V_2 = [q_1, q_2, ..., q_n]$ p_i and q_j represent the number of samples in bins i and j, respectively. C is a matrix with dimensions $n \times n$. Each entry C_{ij} is the ground distance from p_i in V_1 to q_j in V_2. Formally, the ground distance matrix is defined as:

$$C_{ij} = \frac{||p_i - q_j||}{w_{i,j}} \quad (2)$$

where $||.||$ is the L-2 norm operator, and $w_{i,j}$ is the weight for adjusting the bin index difference between bin i and bin j: the larger the difference between i and j, the smaller the value of $w_{i,j}$. We use a nonlinear mapping to calculate $w_{i,j} = e^{-\frac{(i-j)^2}{\sigma^2}}$, where σ is a user-defined width to control the distance between two bins.

3.4 Organizing Data into Storage

Our layout optimization is designed for a parallel file system such as PVFS that distributes data in a file across multiple servers. We will reorder blocks in the data file based on similar feature vectors and stripe them in a round-robin manner across file servers.

Subsection 3.4.1 discusses how similar feature vectors are clustered using the distance metric defined earlier, and Subsection 3.4.2 discusses the ordering of clusters in the data file. The effect of stripe size is examined in Section 4.5.

3.4.1 Meanshift Clustering

Meanshift clustering was used to group blocks based on their feature vectors. Recall that the visibility feature vector has a format of $[h_1, h_2, ..., h_n]$, where each entry of the vector, h_i, is a visibility histogram. To group the visibility feature vectors, we apply the EMD metric described in Section 3.3.2. Given two visibility feature vectors $[h_1, h_2, ..., h_n]$ and $[g_1, g_2, ..., g_n]$, where n is the length of two vectors, the distance is calculated as:

$$D = \sqrt{\sum_{i=1}^{n} EMD(h_i, g_i)^2} \quad (3)$$

where EMD is the Earth Mover Distance.

3.4.2 Data Organization

By grouping blocks with similar values of D we can classify data blocks based upon their visibility and spatial locality. After the clustering process described in section 3.4.1 is completed, blocks that have similar visibility, thus most likely to be accessed together, are placed in a cluster. We keep a data structure for each block that stores three variables: an index that describes the block's spatial location, a cluster ID indicating which cluster the block belongs to, and an EMD from the block to the center of the cluster in the spatial domain. We also keep a table structure to map the blocks' spatial index to their physical position in the file layout.

Data blocks in close proximity or having similar visibility have a higher probability to be accessed together. To order blocks in the file, we first arrange them based on their EMD to the cluster center; then, for those blocks that have approximately the same distance to the cluster center, we arrange them based on their spatial position.

After the relative positions of blocks in a cluster are determined, they are copied into PVFS. Data blocks in a cluster are striped along I/O servers. Data blocks in the same cluster are likely to be accessed together, and striping these blocks across all servers can improve load balance, avoid network congestion, and improve I/O performance per server.

4. VOLUME RENDERING PERFORMANCE RESULTS

We tested the effect of our preprocessing method on the performance of parallel volume rendering. We examined the scalability of the I/O time and end-to-end time in three different datasets under different view directions, transfer functions, and time-steps.

4.1 Datasets and Testing Environment

All our tests were run on an IBM Blue Gene/P supercomputer at the Argonne National Laboratory. The $512 \times 512 \times 1728$ Visible Woman (VisWoman) dataset from the National Library of Medicine, the $2048 \times 2048 \times 1920$ Richtmyer-Meshkov Instability (RMI) dataset from the Lawrence Livermore National Laboratory and the $3456 \times 3456 \times 3456$ supernova dataset from UC Davis were used in our tests. The Viswoman dataset consists of 2-byte short integers, RMI consists of 1-byte values, and supernova consists of 4-byte floating-point values. We supersampled the 1102th time step of the supernova (originally 432^3) data to 3456^3. The Viswoman volume was partitioned into $110,592$ blocks of size 16^3. The RMI volume was partitioned into $245,760$ blocks of size 32^3, and the supernova dataset was partitioned to $10,077,696$ blocks with 16^3 block size. The resultant supernova data is about 276GB, when including a ghost boundary surrounding each block. The block sizes were chosen as described in Section 4.4. All files are stored in PVFS with 16 I/O servers. The Viswoman dataset is a static dataset. RMI and supernova are time-varying datasets with approximately 1000 time-steps each.

Multiple view directions were uniformly sampled on a view sphere. Four different file layouts were used to test the I/O performance. The first was the canonical layout (raw data) where the data blocks were stored as they appeared from the simulation. The second and third were space-filling curves: Hilbert curve and Z-curve. The last was the new layout using our visibility-based clustering. We tested multiple transfer functions in each of the file layouts.

4.2 Scalability of I/O Time

Figure 4 shows the comparisons of I/O performance between the four file layouts as the number of processors increases. Figure 4(a) is the Viswoman dataset, Figure 4(b) is the 16th time-step of RMI and Figure 4(c) is the supernova dataset.

For each file layout, data were volume rendered in 256 randomly selected view directions using the same transfer function. The average I/O time is plotted. Figure 4(a) and 4(b) show that the Z-curve layout does not improve much compared to the raw data file layout; 4(b) and 4(c) show that the Hilbert curve performs better than the Z-curve and raw data layout. Our histogram-optimized file layout has the best performance overall. In Figure 4(c) at 4096 processors, the histogram-optimized layout performs up to 62% better than the other file layouts. Similar performance improvements were observed using other test transfer functions (Section 4.8) and using different block sizes (Section 4.4). The I/O performance for our histogram-optimized layout scales well out to 4096 processors, while the space-filling curves approach the raw data performance at that scale.

4.3 Scalability of End-to-end Time

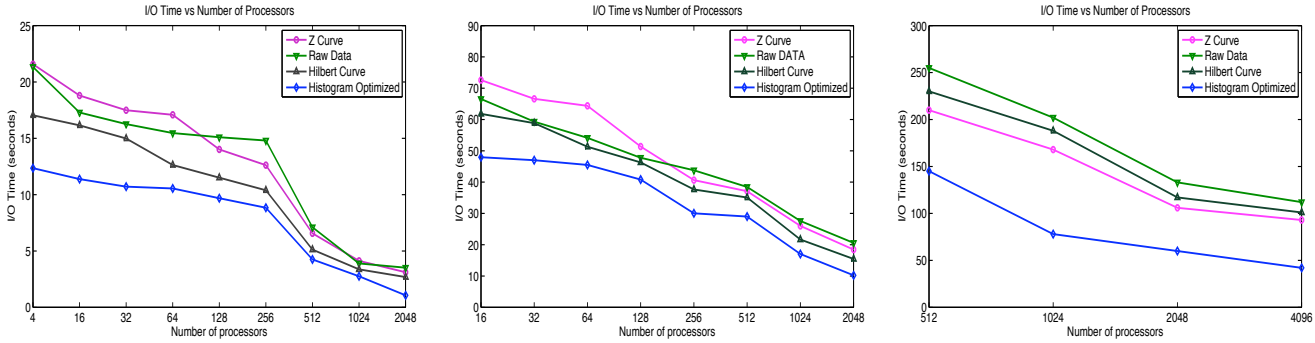
Figure 5 illustrates end-to-end time for the supernova dataset, which is broken down into I/O time, rendering time and image compositing time. All tests used the same direct volume renderer with the same transfer function, and the image size is 1024×1024. Figure 5 shows that I/O is the main bottleneck. The histogram-optimized layout has a smaller percentage of I/O time, compared with the other two space-filling curve layouts, especially with a larger number of processors. For example, at 4096 processors, the end-to-end time is 43% faster than the other two layouts. This is a similar percentage as in Section 4.2 because the end-to-end time is mainly dictated by I/O.

# of Processors	I/O	Rendering	Compositing	Total
64	4.37	1.02	1.2	6.59
128	3.66	0.46	0.8	4.92
256	3.43	0.33	0.8	4.56
512	1.77	0.20	0.6	2.57
1024	0.91	0.12	0.5	1.53

Table 1: Performance (in seconds) for volume rendering Viswoman.

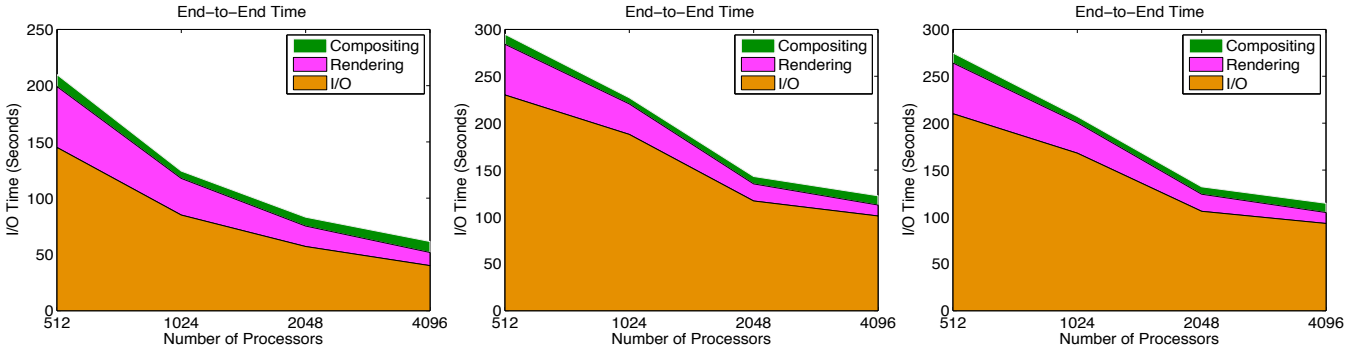
Table 1 shows the performance results using different numbers of processors for the Viswoman dataset. The overall times are shorter than those in Figure 5(a) because the Viswoman data is smaller than the supernova, but the fraction of time spent on I/O, rendering, and compositing is similar.

4.4 Effect of Block Size



(a) Viswoman (b) RMI (c) Supernova

Figure 4: I/O time comparisons for three datasets with increasing sizes. In all three datasets, our histogram-optimized method scales better than the other methods tested.



(a) Histogram optimized (b) Hilbert Curve (c) Z Curve

Figure 5: End-to-end time comparisons for three layouts for the supernova dataset. In all three layouts, our histogram-optimized method has smaller portion of I/O time than the other methods.

The data block size plays an important role in the performance of our algorithm. The determination of block size is a trade-off between the number of I/O operations and the size of data read. We selected block sizes that are multiples of the read buffer size. For instance, our MPI_IO implementation has a default buffer size of 16KB, and we choose block sizes of 16^3 ($16^3 \times 2$ bytes per voxel = 8KB) for viswoman, and block size of 32^3 ($32^3 \times 1$ byte per voxel = 32KB) for RMI.

The choice of which multiple of system buffer size to use is found empirically. In Figure 6, two tests were conducted to find the proper block size. Figure 6(a) shows that in our histogram-optimized method, the block size of 16^3 has the best I/O performance over different number of processors, compared with other block sizes (for the Viswoman dataset). The optimal block size is independent of process count. Figure 6(b) shows that the histogram-optimized method improves I/O time over other methods at all block sizes, but finding the optimal block size is necessary in order to optimize the actual I/O time.

4.5 Effect of Stripe Size

In PVFS, files are striped along multiple I/O servers with a default stripe size. In our study, we found that the stripe size, if not properly chosen, can degrade the overall I/O performance by breaking the block continuity. We observed that the best I/O performance was achieved when the stripe size approached the average size of a cluster, computed as described in Section 3.4. Figure 7 shows the I/O time as a function of stripe size for Viswoman. This dataset has $110,592$ blocks with a size of 16^3, clustered into about 1000 clusters. The average number blocks in each cluster is 110. So we expect the optimal stripe size to be 1 MB, which is close to $110 \times 16^3 \times 2$ bytes $= 901,120$ bytes. Figure 7 shows that the I/O time as a function of stripe size for Viswoman supports our hypothesis.

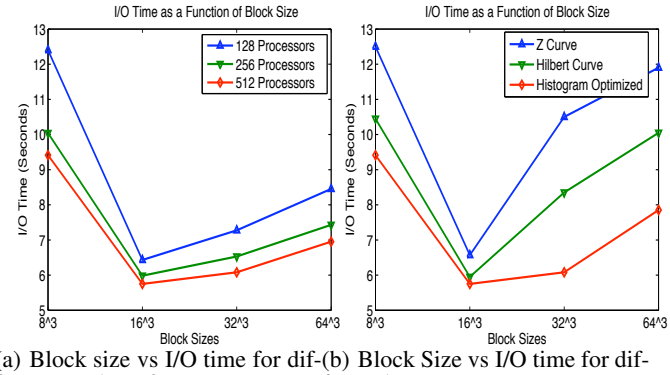
4.6 Effect of View Directions

Figure 8 shows the standard deviations of I/O time across view directions for the histogram-optimized layout in RMI. 256 view directions selected at random were volume rendered, and standard deviations of I/O times were calculated over these view directions. This procedure was repeated with different numbers of processors. Figure 8 shows that the deviation in I/O time is less than 0.8 for the histogram-optimized layout while it is as high as 1.6 for the Hilbert curve layout. This test confirms that the clustering metric we used in preprocessing accurately represents a variety of view conditions encountered at run-time.

4.7 Effect of Time-steps

Figure 9 compares I/O time between 64 time steps for the RMI dataset. The histogram-optimized layout and the Hilbert curve layout were used.

The result shows the optimized file layout has lower I/O time consistently across multiple time-steps. This is an important con-



(a) Block size vs I/O time for different number of processors

(b) Block Size vs I/O time for different layouts

Figure 6: The determination of block sizes for the Viswoman dataset. Left: A block size of 16^3 has the best I/O performance, irrespective of process count. Right: The histogram-optimized method is better than space-filling curves at all block sizes.

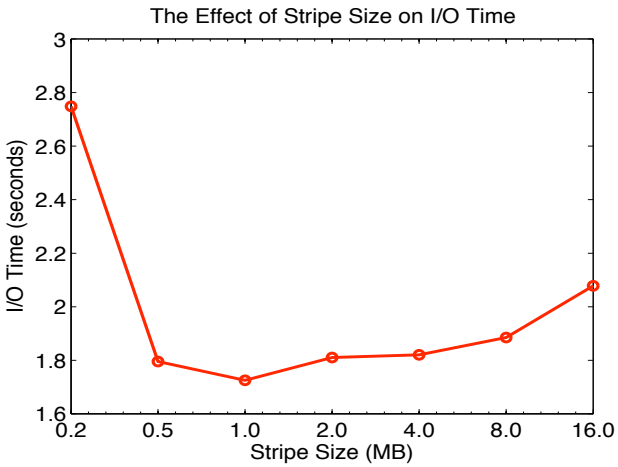


Figure 7: The I/O time vs. stripe size for Viswoman dataset.

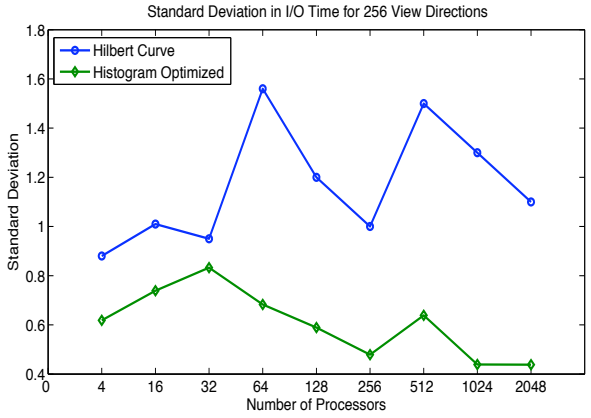


Figure 8: Standard deviation of I/O time across 256 view directions demonstrates that the histogram-optimized layout performs consistently over a variety of view conditions.

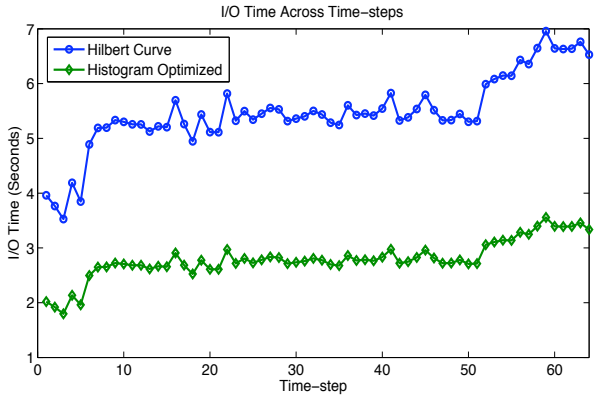


Figure 9: Results from rendering 64 time-steps of the RMI dataset with 512 processors. The I/O time for histogram-optimized layout and Hilbert curve layout are plotted.

sideration when dealing with time-varying data, because the method must perform well across the entire dataset, not just at selected time-steps.

4.8 Effect of Transfer Function

Figure 10 compares the histogram-optimized layout and Hilbert curve layout for the supernova dataset. Multiple transfer functions were applied to render objects with distinct appearance, ranging from solid to semi-transparent, with single and multiple components ranging from small to large sizes. To generate the transfer functions, we randomly selected 2 or 3 modes from the resolution of the transfer function and constructed Gaussian curves for each mode by randomly setting the curve parameters. The corresponding rendered images are also shown in Figure 10. Over all of the transfer functions that we tested, the histogram-optimized file layout has better performance than the Hilbert curve layout.

5. CONCLUSIONS AND FUTURE WORK

This paper introduces a histogram-based I/O optimization for parallel volume rendering. Histograms based on visibility are precomputed for each block from a set of sample views, and feature vectors are constructed from them. Then, the data blocks are organized in storage according to the feature vectors and spatial locality. Experimental results to test the I/O performance during volume rendering showed that the metric is effective in predicting the I/O pattern and reducing the I/O time, independent of transfer functions and view directions. We demonstrated scalability, and offered heuristics toward setting the block size and stripe size.

We plan to investigate the following issues in the future. Our selection of sample views during preprocessing and during volume rendering were taken at the same zoom level. Zooming in or out at run-time can change the visibility culling dramatically from that of preprocessing. Multilevel view sampling based on resolutions is a possible solution to solve this problem. Each level of view sampling corresponds the specific resolution and can be used to interpolate new levels.

Another current limitation is the dimensionality of transfer functions. At present, our method assumes a 1D transfer function. In the future, we plan to extend our algorithm to higher dimensional transfer functions and design a more compact representation to store the resulting histograms.

We plan to also test our method on different parallel file systems such as Lustre. Since our method does not assume a particular

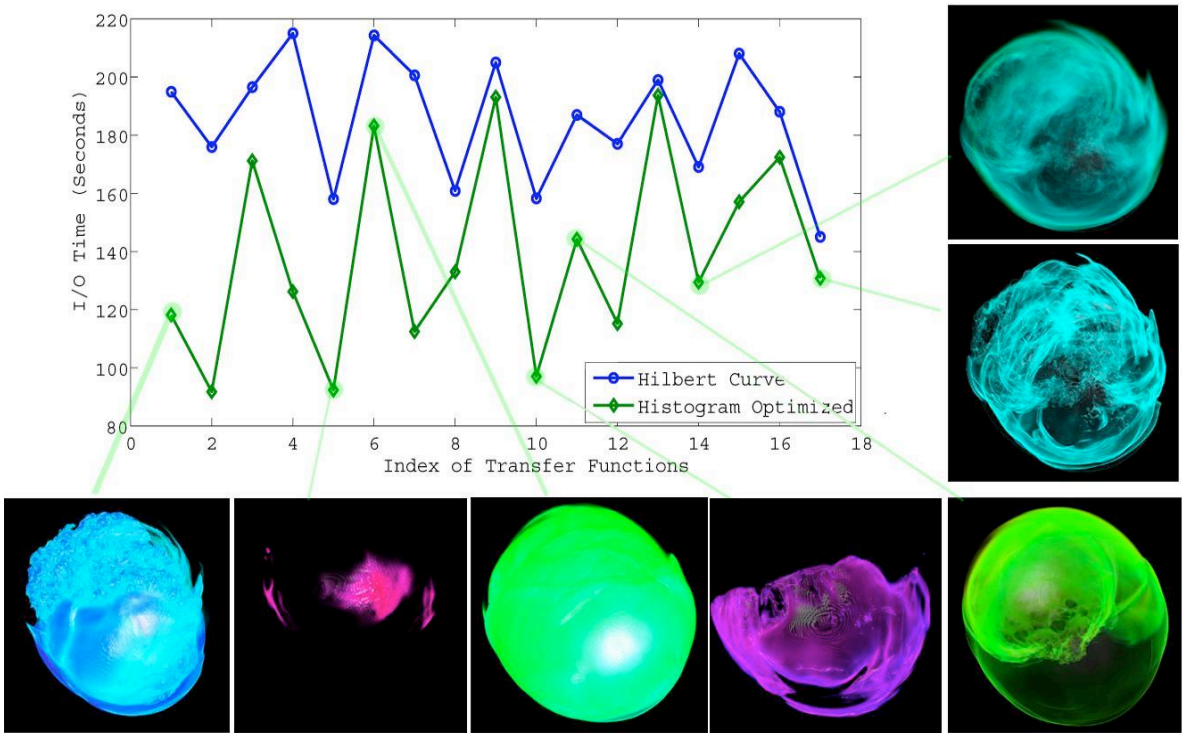


Figure 10: The I/O time comparisons between the histogram-optimized file layout and the Hilbert curve file layout for supernova dataset. Over the multiple transfer functions tested, histogram-optimized file layout has a better I/O performance.

architecture, we expect to see similar performance gains on other systems.

6. ACKNOWLEDGMENT

This work was supported by the U.S. Department of Energy under Contract DE-AC02-06CH11357.

7. REFERENCES

[1] J. Gao, J. Huang, H.-W. Shen, and J. A. Kohl. Visibility culling using plenoptic opacity functions for large volume visualization. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 45, Washington, DC, USA, 2003. IEEE Computer Society.

[2] S. Grimm, S. Bruckner, A. Kanitsar, and M. E. Gröller. Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. In *VG '04: Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*, pages 1–8, Washington, DC, USA, 2004. IEEE Computer Society.

[3] F. Isaila and W. F. Tichy. Clusterfile: a flexible physical layout parallel file system. *Concurrency and Computation: Practice and Experience*, 15:653–679, 2003.

[4] M. Isenburg and P. Lindstrom. Streaming meshes. In *Proc. IEEE Visualization 2005*, pages 231–238, 2005.

[5] Z. Liu, A. Finkelstein, and K. Li. Progressive view-dependent isosurface propagation. In *the Joint EurographicsąIEEE TCVG Symposium on Visualization 01'*, 2001.

[6] K.-L. MA and T. Crockett. A scalable, cell-projection volume rendering algorithm for 3d unstructured data. In *1997 Symposium on Parallel Rendering, IEEE CS Press*, pages 95–104, 1997.

[7] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[8] V. Pascucci and R. J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 2–2, New York, NY, USA, 2001. ACM.

[9] T. Peterka, R. Ross, H. Yu, K.-L. Ma, W. Kenall, and J. Huang. Assessing improvements in the parallel volume rendering pipeline at large scale. In *Proc. SC 08 Ultrascale Visualization Workshop*, Austin TX, 2008.

[10] J.-P. Prost, R. Treumann, B. Jia, R. Hedges, and A. Koniges. Mpi-io/gpfs, an optimized implementation of mpi-io on top of gpfs. In *ACM/IEEE SuperComputing '01*, pages 17–17, 2001.

[11] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. In *IEEE International Conference on Computer Vision,98*, pages 59–66, jan 1998.

[12] C. Silva, Y.-J. Chiang, W. Correa, J. El-Sana, and P. Lindstrom. *Out-of-Core Algorithms for Scientific Visualization and Computer Graphics*. 2003. http://www.sci.utah.edu/ csilva/papers/silva-et-al-2003.pdf.

[13] E. Smirni, C. L. Elford, A. J. Lavery, and A. A. Chien. Algorithmic influences on i/o access patterns and parallel file system performance. In *ICPADS '97: Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, pages 794–801, Washington, DC, USA, 1997. IEEE

Computer Society.

[14] R. Thakur, W. Gropp, and E. Lusk. Optimizing noncontiguous accesses in mpi-io. *Parallel Computing*, 28:83–105, 2002.

[15] Y. Wang and D. Kaeli. Profile guided i/o partitioning. In *ACM International Conference on Supercompuing '03*, jun 2003.

[16] Y. Wu, H. Qu, H. Zhou, and M. Chan. Transfer function fusing. In *IEEE Visualization '06*, 2006.

[17] H. Zhang, D. Manocha, T. H. Kenneth, and E. H. Iii. Visibility culling using hierarchical occlusion maps. In *In Proc. of ACM SIGGRAPH*, pages 77–88, 1997.