Histogram-based Visibility Culling in Visualizing Large Volume Data

Yuan Hong and Han-Wei Shen
the Ohio State University

ABSTRACT

Visibility culling can be used to increase the run time performance when visualizing large volume data sets. Currently, most of the existing visibility culling algorithms depend on pre-specified transfer functions, which makes it difficult to perform general visibility preprocessing and run time visibility culling for applications running on parallel machines. In this paper, we present a novel data-centric visibility culling algorithm that is independent of any particular transfer function and can quickly estimate each data block's visibility at run time for a given view. The main concept in the algorithm is the Perview Sample Histogram extracted from the original volume data in a preprocessing step. These histogram profiles can be used to build a compact linear model to quickly adapt to the run time transfer function and calculate the block-wise visibilities. We notice that, for large volume data sets, I/O is often the bottleneck of the entire rendering pipeline. To improve the I/O performance, the Perview Histogram can be used to construct a visibility feature vector for each data block, which in turn will be used to group the data blocks that are likely to be accessed together. We show that the histogram profile is effective in predicting the pattern of block data I/O.

1 INTRODUCTION

While visualization has been proven effective to analyze data from many scientific and medical applications, the sheer size of data often handicaps the usability of many visualization tools due to the large resource requirement in CPU cycles, memory, storage, and network bandwidth. Among the many techniques available to handle large data sets, visibility culling, a technique that avoids processing invisible data blocks, is one of the most frequently used approaches. Although there have been intensive efforts in the past to develop visibility culling algorithms for volume rendering, two issues still remain to be addressed. First, since the result of volume rendering directly depends on the input transfer function, it is difficult to perform any visibility analysis without knowing the transfer function or a family of transfer functions [6, 7] that will be used at run time. The second challenge is that, as the size of data continues to increase, I/O starts to become a major bottleneck in the visualization pipeline. It is important to avoid introducing extra overhead when reading only the visible data blocks. We have observed that when we only load the visible portions of the data, a large number of small I/O requests are often issued, which will slow down the overall I/O performance. For this reason, it is better to reorganize the layout of the data blocks based on their visibility patterns.

Previously researchers have not paid enough attention to the issues related to improving the I/O performance by taking into account the data set's visibility pattern. There was also no work on visibility preprocessing for volume rendering that is completely independent of run time transfer functions. The latter is important because computing the visibility and re-organizing the file layout based on certain transfer functions do not guarantee satisfactory results for general cases. In this paper, we propose a novel data-centric visibility culling method, which is totally independent of any particular transfer function. Based on this algorithm, We can study the optimal data layout in the file by exploiting the visibility pattern of the blocks in the data set. The algorithms we developed are targeted at parallel visualization of large scientific and medical datasets.

The basic algorithm for visibility culling in the context of volume rendering is early ray termination [17]. For parallel and out-of-core applications, the main goal of visibility culling is to know the exact set of visible blocks before the actual data loading and rendering take place. In spite of the simple idea behind volume visibility culling, one challenge is its dependency to the transfer function, which usually can be changed at run time. This transfer function dependency makes it difficult to perform useful visibility preprocessing without being limited to a small number of transfer functions known in advance. Previously, researchers [6, 7] have proposed to calculate the block-wise opacity information based on a set of given opacity transfer functions at a preprocessing stage. This auxiliary opacity information for data blocks is then distributed to the parallel processors along with the original volume data. During the data loading and rendering phase, the visibility is estimated by looking up those auxiliary information. This type of methods work well only if the run-time transfer functions come from simple linear combinations of the transfer functions that are already known. However, it is impractical to assume that the run time transfer functions to be used by the user are always pre-determined.

To address this problem, in this paper we present a novel approach for visibility culling that is independent of any specific transfer functions. Our method introduces a new concept called *Perview Sampled Histogram* which represents a 1D histogram collected from samples along view-dependent sampling rays. Direct volume rendering essentially consists of two components: one is the process of view-dependent sampling through the volume, which is data-centric, and the other is to look up a transfer function to classify the sample values into colors and opacities. In our algorithm, in a preprocessing stage, we compile the view-dependent samples and collect the ray-based histograms to construct abstract statistical descriptors, called Perview Sampled Histograms, for each data block. At run-time, with an arbitrary input transfer function, the Perview Sampled Histograms can be used to construct a compact linear model to efficiently compute the visibility of each data block. Our algorithm is data-centric because the Perview Sampled Histograms are not subject to any transfer function and derived only from the data.

The key research motivation of our work is based on the observation that ray-based spatial coherence is not unusual in volume data, especially in small local volume blocks. The rays passing through each block usually show local similarities in the histogram generated from the samples. We call this *histogram similarity*, which is used as a metric to cluster the rays within each block. The Perview Sampled Histogram is constructed to describe the clustered histograms. We show that the histograms can be used to compute a block's opacity quickly at run time without needing to render the data. With the help of data-centric visibility preprocessing, it becomes possible to optimize I/O performance based on the visibility pattern of data blocks without depending on any transfer functions. In this paper we use the Perview Sampled Histograms calculated at the preprocessing stage to construct a visibility feature vector for each data block, which in turn is used as the clustering metric to identify data blocks that are likely to be accessed together. The clustering result is used to re-organize the layout of the data file stored in a Parallel File System so that data needed for a given view will be accessed from contiguous regions. Our visibility-assisted

I/O optimization method is data-centric and therefore not subject to any specific transfer functions.

In the remainder of this paper, we first briefly discuss the related work of large scale volume visualization and visibility culling in Section 2. The details of Perview sampled Histogram, run-time visibility culling algorithm and its IO application are then presented in Section 3. Visibility culling results and the IO performance improvements are shown in Section 4. In Section 5, we summarize our contribution and conclude with a discussion of possible future work.

2 Related Work

Previously there have been various research works proposed for visibility culling. In the context of volume visualization, most of them require prior knowledge about the transfer functions. Very little work has been done to perform visibility analysis based on the scalar data only. In this section, we review some of the previous work on visibility culling and also its relations to I/O.

2.1 Visibility Culling

Many parallel algorithms have been designed to utilize large scale parallelism to accelerate rendering of larger volume data. In [20] Ma *et al.* presented the Binary-Swap algorithm for image compositing that has an efficient $O(logN)$ complexity. Ma and Crockett [19] studied parallel rendering of unstructured grid data using cell projections. To further improve the performance, visibility culling is used to eliminate the occluded portion of the volume early in the visualization pipeline. Visibility culling was introduced in view-dependent rendering of isosurfaces in [18, 22]. Authors in [11] developed a visibility-assisted parallel splatting algorithm for volume datasets with moderate to heavy occlusion. To accelerate parallel isosurface extraction, Gao and Shen [4, 5] proposed a progressive visibility culling method that efficiently eliminates invisible isosurface triangles to achieve satisfactory parallel speedups. Guthe and Strasser [8, 25] applied visibility test to multiresolution volume rendering which considers the change of transfer function. Liu *et al.* [31] 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 information for the volume.

Recently, Gao *et al.* [6]proposed a highly-scalable visibility culling method based on Plenoptic Opacity Functions (POFs). The extended version of POF to time-varying data was presented in [7]. POF performs well if the transfer functions are known or can be derived from a small set of base transfer functions. They assume that the transfer functions to use can be expressed by a linear combination of the corresponding bases. But in some cases, as described in [29], transfer functions can be generated from some non-linear combinations of the existing transfer functions. In our work, we do not rely on pre-existing transfer functions.

2.2 Visibility IO

While visibility culling can accelerate the rendering speed, special care needs to be taken to minimize the I/O cost when large datasets are concerned. Since visibility culling usually breaks the whole data into many non-contiguous data blocks that will be accessed at a time, additional optimization from the applications is needed to ensure a good overall I/O performance.

In [1], a Parallel Virtual File System (PVFS) is used to implement *list I/O* to enhance performance of non-contiguous I/O. [26] proposes the idea of *collective caching*, which coordinates the application processes to manage cache data and achieve cache coherence in the PVFS system. Worringen *et al.* [14] present *listless I/O*, that can be incorporated into MPI-IO implementation as an interface, thusly improving the performance of non-contiguous accesses. To achieve higher performance, file layouts and supporting system are changed accordingly in [12]. Clusterfile is a parallel file system provides parallel file access on a cluster of computers. The authors in [12] introduce a file partitioning model that has been used in the design of Clusterfile. The model uses a data representation that is optimized for multidimensional array partitioning while still allowing arbitrary partitions. To effectively exploit parallel resources, Wang *et al.* [27] present a new profile-guided greedy partitioning algorithm to parallelize I/O access for file-intensive applications run on cluster-based systems.

Relatively little attention has been paid to the I/O issue for parallel visualization algorithms. Yu *et al.* [9] presented an I/O solution for visualization of time-varying volume data in a parallel and distributed computing environment. They built a rendering model to calculate the number of I/O nodes necessary for keeping the rendering pipeline full. Yang *et al.* designed an application-specific file system that transparently maximizes the overlap between disk I/O and computation without requiring application modifications [30, 3]. I/O patterns related to visibility have some particular features: they are highly dependent on the transfer functions. The resulting I/O accessing may contains many non-contiguous pieces. In our paper we utilize our fast visibility approximation to estimate the possible accessing patterns and optimize the I/O speed.

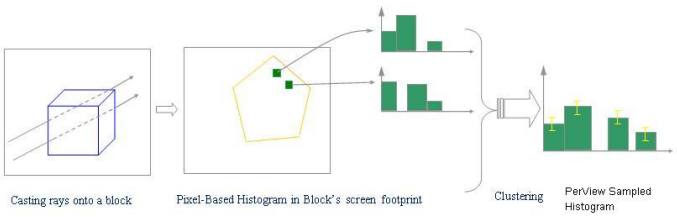


Figure 1: For each view, A histogram is created from the samples along each ray marching through a data block. For each green pixel in the screen footprint there is a sampled histogram created. These sampled histograms are clustered into one or multiple Perview Sampled Histograms to represent the data block in this view direction.

3 Algorithm

Our visibility culling algorithm contains two major components: one is a preprocessing step, described in Section 3.1 and Section 3.2, that extracts the essential information from the data set for efficient run time visibility culling. The other component is the run time algorithm, described in Section 3.3, that performs visibility culling. Our algorithm is based on a novel concept called *Perview Sample Histogram*, collected at a preprocessing step. Fig. 1 shows the basic idea of Perview Sampled Histograms: First histograms are collected along the sample rays cast from the image plane through each data block. These *Sampled Histograms* are processed to obtain compact *Perview Sampled Histograms* that represent the signatures of a data block in one view direction. The details of our algorithm is described in the following sections.

3.1 Ray-based Sampled Histogram

In [21], the accumulated opacity along a viewing ray can be approximated as:

$$\alpha = 1 - \prod_{i=1}^{n} (1 - \alpha(S_i)) \quad (1)$$

where α is the accumulated opacity along the ray, $\alpha(S_i)$ represents the accumulated opacity value within each ray segment S_i,

and n represents the number of segments along the ray. In practice, with a step size of 1.0, we usually approximate $\alpha(S_i)$ by looking up the transfer function using the value sampled at the i-th sample point. Assuming we place the values of the samples along a ray into a histogram, Eqn. 1 can be rewritten as:

$$\alpha = 1 - \prod_{i=1}^{m} (1 - \overline{\alpha}(S_i))^{k_i} \quad (2)$$

where m is the number of bins in the histogram, $\overline{\alpha}(S_i))$ is the corresponding opacity for the scalar value represented by the i-th bin of the histogram, and k_i is the number of sampled voxels placed in the i-th bin. From Eqn. 2, we can see that the accumulated opacity for each ray can be derived from the histogram collected from the samples when combined with a given transfer function. The histogram alone, however, is independent of the transfer function. We call this histogram the *Ray-based Sampled Histogram* which is stored as a histogram vector. A histogram vector has a format of $[w_1, w_2, ..., w_m]$, where w_i shows how many sample points have values that fall into the i-th bin of the histogram.

The idea behind the Ray-based Sampled Histogram is that even though in the preprocessing stage we do not know yet what transfer function will be used by the volume renderer, the histogram of samples along each ray can be collected in a preprocessing stage. This histogram later can be used to quickly compute the accumulated opacity at run time once a transfer function is given. To create the histogram, all we need to know is the resolution of the transfer function, i.e., how many bins will be used for the transfer function to represent the range of the scalar values for a volume data set. The same number of bins will be used to create the histogram. A Ray-based Sampled Histogram is sampled per view for each sampling ray cast from the image plane. The same process is done for all sample views around every block. The required memory storage, however, is not as large as it seems since in most cases only a few bins have non-zero occurrences while many other bins are empty. This is because for a small data block, for example at a resolution of $32 \times 32 \times 32$, in most cases due to spatial coherence, the variance of the scalar values in the data block is quite low. The same situation happens to the gradients too.

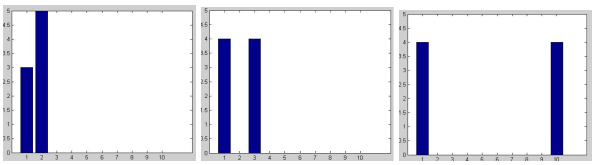
In our experiments, the number of bins needed for each histogram is small, usually 3 to 5. Therefore, a compact version of the Ray-based Sampled Histogram can be used. For now, for a given view one histogram will be created for each sampling ray. In the next section, we describe how to combine the histograms among rays to make the representation more compact.

3.2 Perview Sampled Histogram

The Ray-based Sampled Histograms created for different rays in a given view are often very similar to each other due to the value coherence in the data block. To take advantage of this, given the Sampled Histograms from all rays in a given view direction, we apply mean-shift clustering [2] to group them and use a *Perview Sampled Histogram* to describe each of the clustered groups. The Perview Sampled Histogram is an abstract representation of original data and can be used at run-time to perform visibility culling(see details in Section 3.3).

The reason to choose Mean-Shift is that it is a non-parametric feature space analysis technique. The most attractive feature for mean-shift clustering algorithm is its robustness: it does not require prior knowledge of the appropriate number of clusters to use. Our assumption on clustering is that similar ray-based histograms will produce similar accumulated opacity values since in most cases, the transfer functions are smooth. Our goal to cluster the sampled histograms and use the representative perview histogram for the clustered groups is to reduce the storage requirement.

To compute the difference between two histograms, the traditional Euclidean Distance metric will not work well since it is a flat



(a) bin pattern 1 (b) bin pattern 2 (c) bin pattern 3

Figure 2: different bins patterns. Fig. 2(c) has a different bin pattern from the first two so it should not be clustered together with them although the three histograms share the same number of occurrences.

metric, without considering the shape of the vectors. To address the issue, we use the Earth Mover's Distance(EMD)[28] as the distance metric so that the distance calculation will take the bin patterns into considerations. Figure 2 shows 3 histograms with 2 different bin patterns. All these three histograms have the same total population. For example, histogram in Fig. 2(a) has 3 samples in bin #1 and 5 in bin #2; histogram in Fig. 2(b) has 4 samples in bin #1 and 4 in bin #3; Each of the two histograms has a total of 8 samples and their non-empty bins are close to each other which means they are similar to each other. In the rendering process, similar histograms are most likely to give approximately similar optical properties, due to the smoothness of volume data. So it is more reasonable to group them into the same cluster. The histogram in Fig. 2(c) has a non-empty bin #10, which makes its bin pattern different from the first two. So it is possible that the third histogram will result in a different optical appearance from the first two. If Euclidean distance metric is used, all three histograms will be clustered together since the distance between every two of them is the same, which is the case we want to avoid. Our goal is to group the histograms in Figure 2(a) and Figure 2(b) together but not the histogram in Figure 2(c). The underlying idea is that the closer of the bins in histogram are, the more likely they will be assigned with similar opacities.

The Earth Mover's Distance (EMD) is a method to evaluate the dissimilarity between two multi-dimensional distributions in certain feature space where a distance measure between two single features, called the ground distance matrix C, is given. Since the Ray-based Sampled Histograms have different vector sizes, EMD is very suitable to be used as a metric to compare the histogram bin patterns.

To fit EMD to our need, we redefine the ground distance matrix C as follows: For 2 sampled histograms $V_1 = [p_{n_1}, p_{n_2}, ..., p_{n_l}]$ and $V_2 = [q_{m_1}, q_{m_2}, ..., q_{m_h}]$ where l and h are the length of the histograms; p_{n_i} and q_{m_j} represent the number of samples in bin n_i and m_j, the ground distance matrix C is a matrix with dimensions $n \times m$. Each entry C_{ij} is defined as the ground distance from p_{n_i} in V_1 to q_{m_j} in V_2. Formally, ground distance matrix C is defined as:

$$C_{ij} = \frac{||p_{n_i} - q_{m_j}||}{w_{n_i,m_j}} \quad (3)$$

where $||.||$ is the norm operator and $w_{i,j}$ is the weight for adjusting the distance between bin n_i and bin m_j: the larger the difference between n_i and m_j, the smaller the value of $w_{i,j}$ is. We use a non-linear mapping to calculate $w_{i,j} = e^{-\frac{(n_i - m_j)^2}{\sigma^2}}$, where σ is the user-defined width to control the distance between 2 bins; n_i and m_j are the bin index of the histogram. Computing the EMD is based on a solution to the well-known transportation problem [10] which can be formalized as a linear programming problem: feature vectors are classified into supplier and consumer and for each supplier-consumer pair, the cost of transporting a single unit of goods is given by the ground distance matrix C. The transportation problem is then to find the least-expensive flow of goods from the suppliers

to the consumers that satisfies the consumers' demand [10].

After clustering we obtain one or more histogram groups, each of which is described by a *Perview Sampled Histogram*. In our experiments, most of the data blocks after the clustering, produce only one such histogram group for one view direction. We treat these Perview Sampled Histograms as the abstract representations of the data blocks for that view direction. To store them, the following information are recorded:

1. Number of the non-empty bins n and the indices of those bins. Since the histogram vector is sparse we only need to save non-empty bins;

2. N_{max} : Maximal number of total samples in the histograms from the group;

3. N_{min} : Minimal number of total samples in the histograms from the group;

4. M_{max_i} : Maximal number of samples for each individual histogram bin in the clustered group;

5. M_{min_i} : Minimal number of samples for each individual histogram bin in the clustered group;

For example, the Perview Sampled Histogram to describe the histograms in Fig. 2(a) and Fig. 2(b) contains the following information. These two histograms in this cluster have the same total number of samples, e.g. $N_{max} = N_{min} = 8$; there are 3 distinct non-empty bins in these two histograms; the histogram populations for bin #1 range from the value 3 to 4, bin #2 from 0 to 5 and bin #3 from 0 to 4. As we can see, the number of non-empty bins in the Perview Sampled Histogram usually is greater than that of its individual members in the cluster because our clustering strategy is that, when a bin has a non-zero value in one of the clustered members, this bin should be recorded, e.g. adding a pair of M_{max_i} and M_{min_i}. More information can be added to describe the Perview Sampled Histogram but it will require larger storage space. In this paper the above information is sufficient for visibility computation. A Perview Sampled Histogram has a tuple-format $[n, N_{max}, N_{min}, M_{max_1}, M_{min_1}, ..., M_{max_n}, M_{min_n}]$. The Perview Sampled Histogram will be used to solve a linear programming problem at run-time, which is discussed in the next section.

3.3 Run-Time Visibility Calculation

At run time, to estimate whether a block is visible from a given view direction, we first estimate the minimum opacity of each data block, and then composite the opacities from the blocks together in a front to back order according to the view direction. A block is considered invisible if the accumulated opacity from the blocks in front of it is close to 1.0 or larger than a threshold. This is a conservative approach because many rays going through a block may accumulate a much higher opacity. Nevertheless, it makes sure that no wrong result will be generated, i.e., no visible blocks will be classified as invisible. In the previous work [6, 7], the minimum opacities are pre-calculated based on Eqn. 2 for each block with some presumed transfer functions. In this paper, we use the Perview Sampled Histogram to represent a cluster of sampled histograms for a block at a given view, so that no prior knowledge about the run time transfer functions is required. With the Perview Sampled Histogram, the problem of finding the minimal opacity is equivalent to finding one sampled histogram from the cluster that will produce the minimal opacity for a given run-time transfer function. Since the exact information of each sampled histogram is merged into the Perview Sampled Histogram, the problem of finding a histogram with the minimal opacity is transformed to an optimization problem based on certain given constraints.

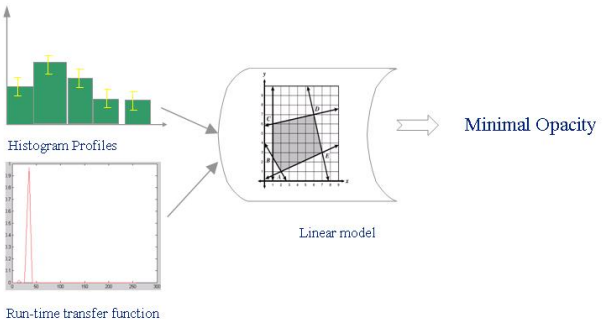


Figure 3: Perview Sampled Histogram and run-time transfer function are input to create a linear programming problem. After solving this linear programming problem the maximal value f should be converted back to the minimal value of α. Visibility determination is implemented by checking the accumulated α. The process is done along the view direction.

To tackle the problem of finding the minimal opacity, we use a linear programming model in order to avoid brute-forcedly checking all the possibilities or introducing dependency to specific transfer functions. To do this, we first rewrite Eqn. 2 as:

$$1 - \alpha = \prod_{i=1}^{m} (1 - \overline{\alpha}(S_i))^{k_i} \tag{4}$$

If we apply the log function to both sides and let $f = log(1-\alpha)$, then we have:

$$f = log(1-\alpha) = \sum_{i=1}^{m} log(1 - \overline{\alpha}(S_i))k_i = \sum_{i=1}^{m} c_i k_i \tag{5}$$

where $c_i = log(1 - \overline{\alpha}(S_i))$, and can be computed when the transfer function is known at run time. Since Eqn. 5 is a linear combination of variable k_i which is non-negative, we can build a linear programming model to calculate the maximum value of f in the above equation, which is equivalent to finding the minimal value of α in Eqn. 2.

The linear programming model is defined as follow:

$$\text{Objective function : Maximize f}$$
$$\text{Subject to :} k_i \geq 0 \text{ i=1...n}$$
$$N_{min} \leq \sum_{i=1}^{n} k_i \leq N_{max}$$
$$M_{min_i} \leq k_i \leq M_{max_i} \text{ i=1...n}$$
$$\textit{Other user-defined constraints}$$

The above linear programming problem can be constructed from the pre-computed Perview Sampled Histograms with a given run-time transfer function. The histograms are used to formulate the constraints for the linear problem, and the run-time transfer function is used to calculate the coefficients c_i. Given a run-time transfer function, c_i only needs to be computed once as long as the transfer function stays the same. The complexity of computing c_i is linear, which is proportional to the number of bins in the transfer function. The first entry of the Perview Histogram is used to set up the number of k_i in the linear programming model; the following entries set up the range of each k_i in the constraints. The goal of this linear programming problem is to find out the maximum value of f, thereby finding the minimal value of α in Eqn. 2.

Fig. 3 shows the run-time process to estimate the block-wise visibilities. At run-time, along the view direction, each data block calculates its minimal opacity by solving the linear programming problem constructed from the corresponding Perview Sampled Histograms and the run-time transfer function. Extra constraints can be added if prior knowledge of the data is available. These constraints depict a feasible region for the linear programming problem. Our method is to search the boundary of the feasible region in the linear model to acquire the max f. There are trade-offs between the number of the constraints and the final maximal value of f: if more constraints are used, the resulting f_{max} is closer to the exact one; On the other hand, the storage requirement becomes larger. However, some heuristics can be utilized to reduce the storage. For instance, in many scientific datasets there exist a large amount of uniform data blocks. For those uniform blocks there is no need to calculate the histograms.

The Simplex Tableau algorithm is used to solve the linear model by constructing an admissible solution at each vertex of the polyhedra, and walking along the edges of the polyhedron to find vertices with higher values of the objective function until the optimum is reached. The above linear programming problem can be solved in polynomial time $O(h)$ where h is related to the number of constraints. Since the Perview Sampled Histograms are created for small 3D data blocks, the number of the constraints are limited.

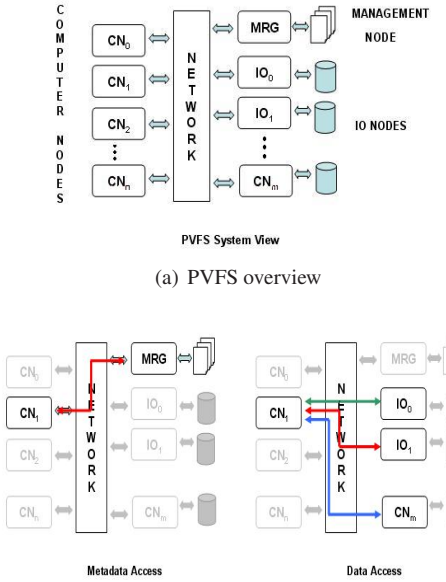
3.4 I/O Optimization

In this section we discuss the use of Perview Sampled Histograms to improve I/O performance. As the size of data continues to increase, I/O is becoming one major bottleneck in the visualization pipeline. There are two sources of inefficiency. One is that loading large files from the storage devices would naturally take longer. The other is that even when we only load the data that is necessary to create the visualization, such as the visible data blocks, the raw data can be scattered across the entire disk so a large number of small I/O requests will be issued. These so-called non-contiguous I/O requests can cause extra overheads and further slow down the I/O performance.

In this work, we aim to improve the overall I/O performance related to visualizing large volume data sets by taking advantage of visibility. Our algorithm is based on the *Parallel Virtual File System (PVFS)*. PVFS has been recently used in a rendering system [9] to provide global user accesses across different processors or clusters. Its goal is to provide a higher and scalable performance compared with the traditional methods where the user has to subdivide the data files in advance and distribute them to different cluster nodes' local storage. The main idea of our algorithm is to use the Perview Sampled Histograms calculated in the preprocessing stage to build *Perview Visibility Vectors* that are metrics to identify data blocks that have similar visibility properties. Our method is to create a more efficient data distribution strategy to groups together data blocks that are more likely to be accessed together to avoid non-contiguous I/O requests. Our approach is described in detail in Section 3.4.2. In the following, we first briefly introduce the main concept of PVFS.

3.4.1 PVFS and Non-Contiguous I/O

The Parallel Virtual File System (PVFS) is an Open Source parallel file system, which is a type of distributed file system that distributes data in a file across multiple servers and allows for concurrent access by multiple tasks in a parallel application. PVFS was designed for use in large scale cluster computers. Fig. 4 shows an overall view of a standard PVFS and its data migration flows. In PVFS, data are striped along each I/O server in a round-robin manner. If the useful or visible data blocks are scattered sparsely in the large dataset, there exist two I/O related problems if only these visible or useful data blocks are loaded. First, these visible or useful blocks might be stripped into one or a few I/O servers, hence produce I/O congestions during the rendering time. Second, these visible or useful blocks may also scatter very sparsely in an individual I/O server, which will increase the seeking time for external storage devices. Fig. 5 displays two I/O access patterns in two I/O servers for the viswoman data set stored in PVFS2, where the data $(512 \times 512 \times 1728)$ is distributed to 32 I/O servers with the strip size equal to 1K bytes. Fig. 5 shows that, even in PVFS, the data I/O access patterns can be very sparse and generate many non-contiguous I/O requests to the I/O servers.



(a) PVFS overview

(b) Data Migration

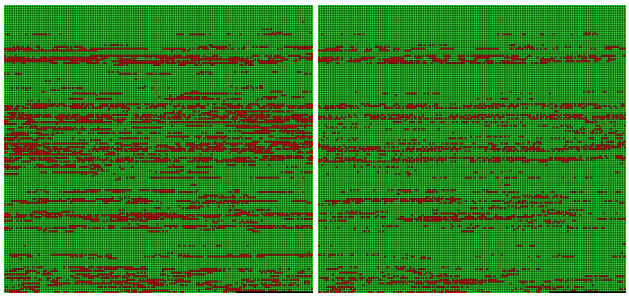
Figure 4: PVFS overview and data migration.

Many previous system research works have focused on non-contiguous I/O issues. For example, MPI has implemented collective I/O functions such as *MPI_File_read_all* to ameliorate the I/O overheads. The main idea behind it is to utilize 2-*phase loading* and *data sieving* which load more data blocks than necessary and *sieve* the useful ones out [24, 23, 13]; In this way fewer I/O operations are needed and thus better performances can be obtained. *ROMIO* implements a similar idea in their fundamental I/O modules and uses the List-I/O data structure to buffer the individual I/O requests and sends them together in a later time [15].

In spite of all the existing system support, the performance of applications still degrade significantly as the number of non-contiguous I/O requests increase, which are not unusual in scientific visualizations applications. Generally, there are up to 70 to 80 percent of data blocks that are transparent, and hence meaningless to load, and among those meaningful blocks, half of them can be invisible after visibility culling. Therefore, more steps can be taken to further reduce the I/O overhead. Our strategy is to utilize the Perview Sampled Histograms to construct a *visibility feature vector* for each data block. We use these visibility feature vectors as signatures to identify the data blocks that have similar access patterns: either visible or invisible. The following Section discusses our approach in detail.

3.4.2 Visibility Feature Vector

As mentioned previously, in the preprocessing stage we calculate the Perview Sampled Histogram, which can be used to compute



(a) I/O Pattern in the #2 IO server (b) IO Pattern for in the #15 IO server

Figure 5: Sparse I/O access patterns In Viswoman($512 \times 512 \times 1728$) with data block size $16 \times 16 \times 16$. The sequential data file is reorganized into a 2D block square representation. The green blocks represent the data strips in the I/O server and the red ones represent the visible/useful data strips that should be loaded. Data access patterns in two servers are shown.

the minimum opacity for each data block from a given view. To improve the I/O performance and minimize the number of non-contiguous I/O requests, we want to group together the data blocks that share similar visibility characteristics and place them together in the data file. To achieve this, we cluster data blocks based on their *Accumulated* Sampled Histograms. The Accumulated Sampled Histogram is the histogram created from the samples generated by a viewing ray from the image plane through all the blocks before the ray reaches the block in question. The idea behind is that if two blocks have similar Accumulated Sampled Histograms, they should have similar visibility since the Sampled Histogram is used to estimate the accumulated opacity before the ray reaches the block.

Given a view direction, there are two possible ways to obtain the accumulated histogram for each data block. The first is to cast rays through the volume and collect the sampled voxels along the rays to build the accumulated histograms; The other way is to take advantage of the pre-calculated Perview Sampled Histograms from the individual data blocks by combining them along the view direction, which is an approximation of the first approach. In this paper we use the second method to minimize the computational cost. For each data block, we add up the Perview Sampled Histograms of the blocks which are spatially located in front of the current block. The result is a block-wise accumulated histogram for the current block. The whole process repeats for each view direction.

With the Accumulated Histograms generated for a set of sampled views, for each data block we create a *Visibility Feature Vector*. A visibility feature vector is a n-dimensional tuple in the format of $[h_1, h_2, ..., h_n]$, where each dimension represents a sampled view direction. The entry h_i is the corresponding Accumulated Histogram with respect to the i-th view direction. The purpose of the visibility feature vector is to keep an overall description of a block's visibility status for all possible view directions. Each block has a visibility feature vector, based on which it is possible to compare the visibility characteristics among multiple blocks. The block-wise visibility feature vector is constructed based on the data only and not depending on run time transfer functions. By clustering the data blocks based on the visibility feature vectors, it is possible to identify those blocks with approximately the same visibility status, and hence the same usage pattern.

We apply the clustering technique described in Section 3.2. Since each entry of a visibility feature vector is a vector instead of a scalar value, the distance metric is defined as the EMD metric. 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 in following:

$$D = \sqrt{\sum_{i=1}^{n} EMD(h_i, g_i)^2} \tag{6}$$

where EMD is the Earth Mover Distance. Eqn. 6 first calculates the EMD between each pair of the vector entries, then normalize them into a scalar value D which is used as the clustering metric. By grouping blocks with close values of D we can classify data blocks based upon their visibility status. The overall process to build the visibility feature vector and clustering is listed blow in Algorithm 1.

Algorithm 1 Visibility Feature Vector Algorithm

```
for each view direction P do
    for each block i in the order of view direction P do
        Accumulate histograms of blocks in front of block i
        Add up to a histogram vector v_i
    end for
end for
for each pair of v_i and v_j do
    D = sqrt(sum EMD(v_i, v_j)^2)
    clustering based on D
end for
```

When coping data file into the PVFS, data blocks are stripped along various I/O servers in terms of clusters: the blocks within the same cluster are distributed contiguously into a sequence of I/O servers. This new layout strategy has the advantage that, since the data blocks in the same cluster are likely to be accessed together, distributing the I/O load to all servers can avoid imbalanced traffic congestion in a few I/O servers or multiple non-contiguous IO requests in a single IO server. Our experimental tests showed that the new file layout can greatly reduce the I/O cost.

4 Results

4.1 Visibility Culling

In this section, we present experimental results to study the effectiveness of our algorithm. All our tests were run on an IBM Blue Gene/L supercomputer at the Argonne National Laboratory, which has up to 2048 700 MHz PowerPC 440 embedded processors, each with a double-pipeline-double-precision Floating Point Unit (FPU), a cache sub-system with built-in DRAM controller and the logic to support multiple communication sub-systems. The dual FPUs give each BlueGene/L node a theoretical peak performance of 5.6 GFLOPS (gigaFLOPS). The $512 \times 512 \times 1728$ VisibleWoman dataset from the National Library of Medicine and the $1024 \times 1024 \times 960$ Richtmyer-Mevhkov Instability (RMI) simulation dataset from the Lawrence Livermore National Laboratory were used in our tests. In our experiments, the Viswoman volume is partitioned into $110,592$ ($16 \times 16 \times 16$ = 4096 voxels) volume blocks and the RMI volume is partitioned into $245,760$ ($16 \times 16 \times 16$ = 4096 voxels) blocks. All volume data are stored in the PVFS2 with the strip size of 2K and 25 I/O servers. All of our experiments were conducted using 64 processors.

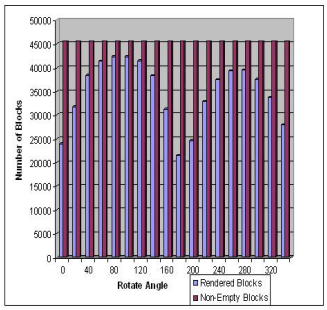
To conduct our experiments, we implemented a benchmark visibility culling program to calculate the exact visibility of each block for a given transfer function. The goal of this benchmark is to verify the correctness and effectiveness of our algorithm. In the benchmark implementation, blocks are rendered in a front-to-back order. When a block is visited, its visibility is checked by looking up the composited opacity from the previously rendered blocks in its screen projection area. The visibility of each block from this

benchmark test is collected for later use. In our experiments, all data blocks are classified into 3 classes: a *transparent block* is the block that has a nearly zero overall opacity; an *visible block* is the block that has some contribution to the final image and hence cannot be removed; and a *invisible block* is the block that is occluded by others and thus can be culled away to improve rendering performance. The transparent blocks are determined directly by the transfer function and not dependent on the view direction. But visible and invisible blocks depend on both the transfer function and the view direction. To precisely evaluate the benefit of visibility culling, we do not consider the transparent blocks in the rendering pipeline, since they can be trivially rejected by checking the variation of the block's minimal and maximal opacity or using the value histogram [16].

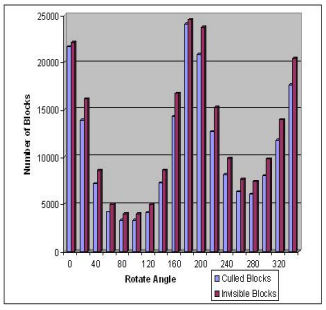
In visibility culling, *false positives* are the invisible blocks that are wrongly classified as visible. On the contrary, *false negatives* are the visible blocks classified as invisible, which will cause incorrect images being generated. The effect of false positive is to introduce unnecessary overhead since invisible blocks contribute nothing to the final image. The goal of our culling algorithm is to reduce the number of false positives while preventing any false negative to happen.

All of our experiments were conducted in the following way. First we run the benchmark program to obtain the precise visibility information for all the data blocks with respect to a specific transfer function; then we run our histogram-based culling algorithm using the same transfer function. The culling results from our algorithm are compared with the benchmark results.

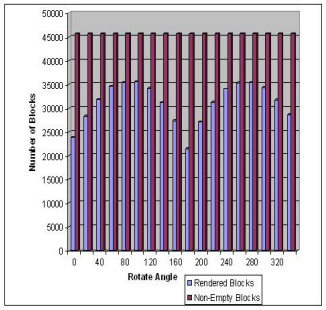
4.1.1 VisWoman Dataset



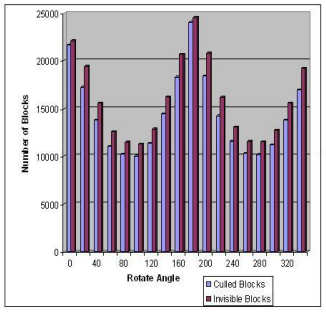
(a) Total rendered vs Total Non-empty



(b) Total Culled vs Total Invisible

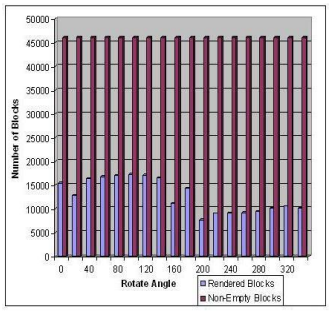


(c) Total rendered vs Total Non-empty

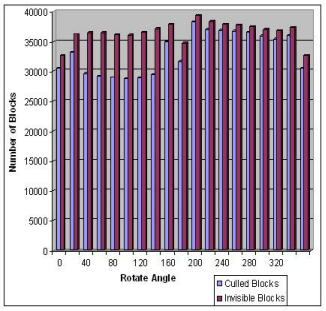


(d) Total Culled vs Total Invisible

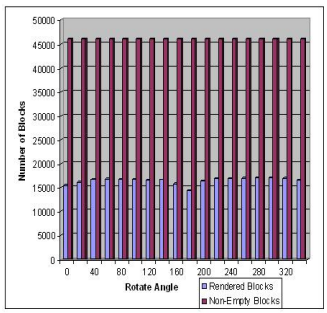
Figure 6: Results from the visibility culling tests using the Viswoman($512 \times 512 \times 1728$) data set with the data block size $16 \times 16 \times 16$. The numbers of data blocks that are rendered and culled are compared with their corresponding benchmark results. The View direction was rotated along the X and Y axes and the data was rendered once from each direction. One of the rendered image using this transfer function is shown in Fig. 10(a).



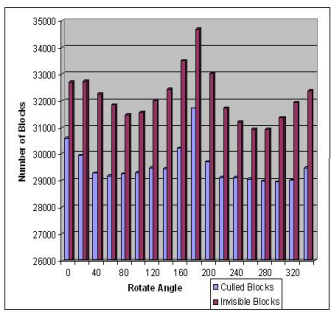
(a) Total rendered vs Total Non-empty



(b) Total Culled vs Total Invisible



(c) Total rendered vs Total Non-empty



(d) Total Culled vs Total Invisible

Figure 7: Results from the visibility culling tests using the Viswoman($512 \times 512 \times 1728$) data set with the data block size $16 \times 16 \times 16$. The numbers of data blocks that are rendered and culled are compared with their corresponding benchmark results. The view direction was rotated along the X and Y axes and the data was rendered once from each direction. One of the rendered image using this transfer function is shown in Fig. 10(b).

Fig. 6, 7, 8 show results from a series of experiments on the VisWoman with three different transfer function. Different view directions by rotating the data around X axis and Y axis are used in the tests. The corresponding images rendered using the transfer functions are shown in Fig. 10. The rendered image using the transfer function in Fig. 6 is shown in Fig. 10(a), which has more semi-transparent voxels. In Fig. 6(a) and Fig. 6(b), the view directions are sampled along the X axis with an 18 degree increment. In Fig. 6(c) and Fig. 6(d), the view directions are sampled with the same increment but along the Y axis. For each view direction the results of our culling algorithm are compared with that in the benchmark. Fig. 6(a) compares the number of data blocks that were rendered after applying our histogram-based visibility culling algorithm and the total number of non-empty data blocks. The total number of non-empty data blocks equals to the sum of the numbers of visible and invisible blocks. Eighteen tests were conducted with different view directions to verify if our culling method is stable. Fig. 6(b) compares the number of blocks culled by our algorithm with the number of actual *invisible* blocks. The difference between these two are due to the false positives. From Fig. 6(b) we can clearly see that our culling method can remove around 80% of the invisible blocks and this culling rate is stable in different view directions. Fig. 7 shows the culling result using another transfer function to test if our culling method can perform well when the transfer function changes. The image rendered from this transfer function is shown in Fig .10(b). In our test, the view directions were rotated along X axis in Fig. 7(a),7(b) and Y axis in Fig. 7(c),7(d), both with a 18 degree increment. Similarly to Fig. 6, the number of the rendered data blocks and culled blocks are compared with the benchmark results. Form Fig. 6 and Fig. 7, we can see that the culling rate of

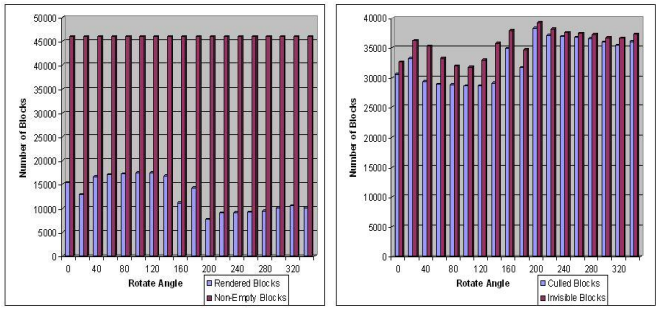
our method is stable, independent of the run time transfer function and view directions. We further tested our algorithm with another transfer function. Fig. 10(c) is the rendered image, which shows the flesh combined with bones in the VisVoman data. The comparison results are drawn in Fig. 8(a)-8(d). From Fig. 8(b) and 8(d) we can easily find the culling rate is around 82% and stable as well.

Table 1 shows the average visibility culling time for three different tests: each test has a specific transfer function with six different view directions (VD1-VD6). The visibility culling time includes reading pre-computed sampled histograms; estimating the visibility for each block and finally communicate to gather the visibility information.

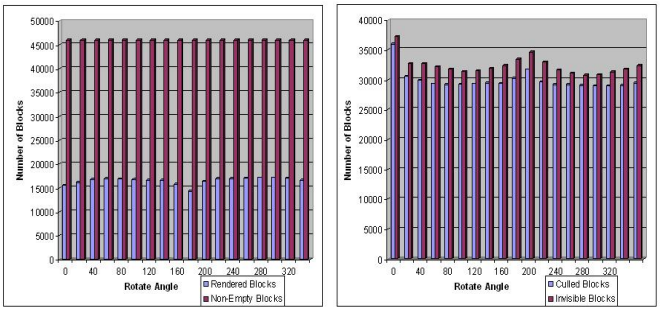
Test with different TF	VD1	VD2	VD3	VD4	VD5	VD6
1	0.13	0.25	0.37	0.23	0.31	0.25
2	0.14	0.31	0.49	0.21	0.43	0.17
3	0.20	0.35	0.21	0.11	0.29	0.15

Table 1: The average time (in seconds) to perform visibility culling in different tests

Table 1 shows that our histogram-based culling has relatively low overheads. It is possible to further speed up the culling process. Since there are many uniform data blocks with similar voxel values within, to these blocks, the culling procedures can be replaced with a quick transfer function lookup and a multiplication of the block's depth along the view direction.

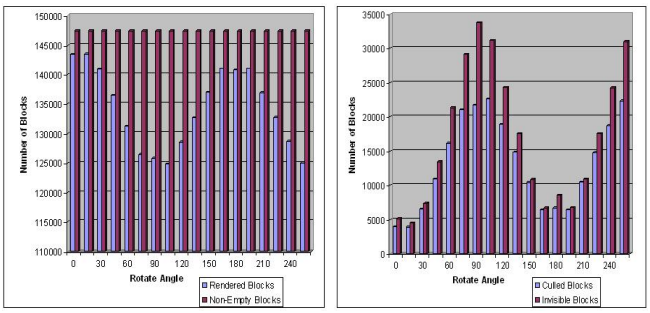


(a) Total rendered vs Total Non-empty (b) Total Culled vs Total Invisible



(c) Total rendered vs Total Non-empty (d) Total Culled vs Total Invisible

Figure 8: Results from the visibility culling tests using the Viswoman($512 \times 512 \times 1728$) data set with the data block size $16 \times 16 \times 16$. The numbers of data blocks that are rendered and culled are compared with their corresponding benchmark results. The view direction was rotated along the X and Y axes and the data was rendered once from each direction. One of the rendered image using this transfer function is shown in Fig. 10(c).



(a) Total rendered vs Total Non-empty (b) Total Culled vs Total Invisible

Figure 9: Results from the visibility culling tests using the 228th time step of the RMI($1024 \times 1024 \times 960$) data set with data block size $16 \times 16 \times 16$. The number of data blocks that are rendered and culled are compared with their corresponding benchmark results. The view direction was rotated along the axis [225 0 0] with Y increasing 15 degree each step. The data was rendered once from each direction. One of the rendered image using this transfer function is shown in Fig. 10(d).

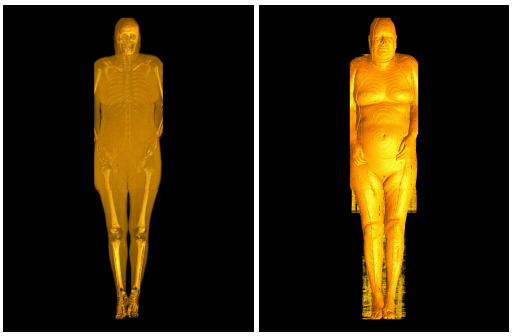
4.1.2 RMI Dataset

The 228th time step of RMI dataset was used to test our algorithm. Scientific simulation datasets have some different natures compared to medical datasets in that they sometime produce large potions of uniform data blocks. Moreover, the boundaries of regions are often not so obvious in scientific data as in medical dataset. In our experiments, we compared the result of visibility culling with the benchmarks, as illustrated in Fig. 9. The number of data blocks that were rendered and culled are compared with the corresponding benchmark numbers. As shown in Fig. 9 the data set has a total of 245760 data blocks, out of which 147439 are non-empty blocks. This is also equal to the sum of the visible blocks and invisible blocks. The remaining 98321 data blocks are transparent. It can be seen that the number of blocks that can be possibly culled is smaller compared with the VisWoman dataset. The culling rate was around 77% and remained stable which is similar to the case in the VisWoman dataset.

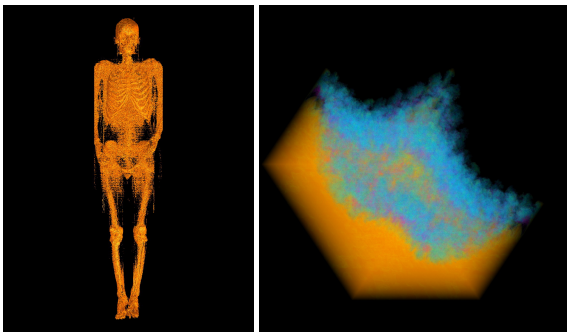
4.2 I/O Performance

For the Viswoman dataset we computed the Visibility Feature Vectors described in Section 3.4.2. The sampled view directions were rotated along the X axis and Y axis with a 20 degree increment in each step, which produced a total of 64 sampled view directions. Based on the Visibility Feature Vectors a clustering process was conducted to group those blocks with similar feature vectors. Our I/O strategy is to place those data blocks in the same cluster contiguously in the logical file layout since members in the same cluster should have similar visibility characteristics, hence similar usage patterns. When a data file is stored in a PVFS, the system will strip the data in a round-robin manner to several I/O servers. An I/O request for a chunk of logical contiguous data blocks will cost much less than several individual I/O requests for non-contiguous data blocks. The latest PVFS has optimizations that can combine individual I/O requests into a larger one if the logical file addresses of the requested data blocks are contiguous. In volume rendering, visibility culling can substantially speed up the process. Therefore, the layout method described in Section 3.4.2 can optimize both the rendering and I/O speed.

We conducted a series of experiments to test the I/O performance of our algorithm. Two different file layouts were tested on the VisWoman dataset: one was the original layout where the data blocks were stored sequentially according to their spatial locations, and



(a) TF1 for Viswoman (b) TF2 for Viswoman

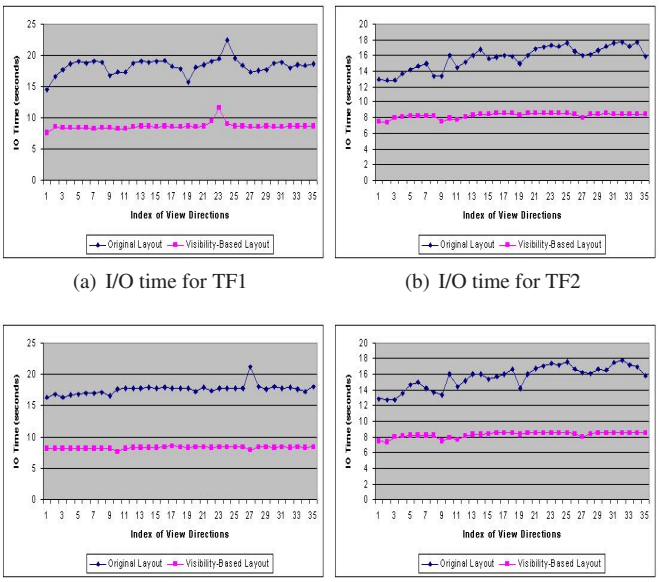


(c) TF3 for Viswoman (d) TF1 for RMI

Figure 10: Rendered images in Viswoman and RMI for Different transfer functions.

the other was the new layout using our visibility-based clustering. To ensure that the I/O performance improvement was stable under different the transfer functions, 4 transfer functions were used for each layout, and 35 view directions were sampled to test with each transfer function. During the run-time rendering process, after visibility culling, each processor reads the needed data blocks by sending individual I/O requests. The PVFS is then exploited to optimize these I/O requests. In our implementation we first use MPI_File_set_view() to adjust individual process's view of the data in the file. Then we call MPI_File_read_all() to utilize the noncontiguous I/O optimizations provided in PVFS2. For each view direction, the I/O time was collected for each processor and the average time was calculated as the metric to compare between the two layouts. We set up the data file in PVFS using 32 I/O servers with 3 strip sizes: 1024, 4096 and 8192 bytes to further explore the relationships of the PVFS performance and our layout algorithm.

Fig. 11 shows the comparisons of I/O performance between the two file layouts with the 1024 bytes PVFS strip size. Fig. 11(a), 11(b), 11(c), 11(d) clearly illustrate the improvements achieved by the visibility-based file layout. For each view direction the same number of data blocks were read in both file layouts. PVFS's noncontiguous optimizations performed more efficiently when using the visibility-based layout. Results show that there were up to 50% of I/O time reduction using the visibility-based file layout under different view directions. The same performance improvements were observed using different transfer functions. Moreover, the variances of I/O time under different view directions with the visibility-based layout were far less than those with the original layout. This is because in the visibility-based layout, there existed fewer numbers of non-contiguous I/O requests compared with the original file layout. Therefore, the I/O performance was much more stable. Fig. 12 and Fig. 13 show the comparisons of I/O time between the two file layouts with 4096 and 8192 bytes PVFS strip sizes. With a fixed number of I/O servers, we increased the strip sizes and performed the same tests as in Fig. 11. We observed that with a larger strip size the time needed to read the same number of data blocks decreased. The visibility-based file layout still outperformed the original layout by about 50%. Since the size of a data block is $16 \times 16 \times 16 \times 2 = 8192$ voxels, it is unnecessary to use a larger strip size than the size of a block. Results in Fig. 11, 12, 13 demonstrate that the Perview Sampled Histogram can effectively predicted the I/O patterns and optimize the I/O speed when rendering large data sets.



(a) I/O time for TF1 (b) I/O time for TF2

(c) I/O time for TF3 (d) I/O time for TF4

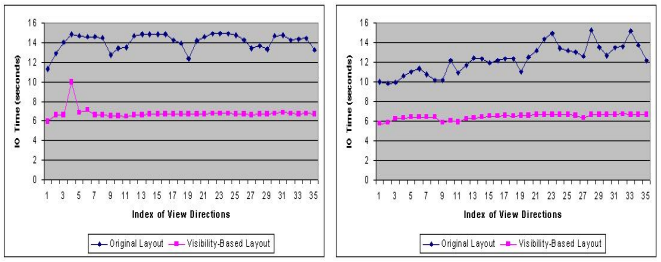
Figure 11: Comparisons of I/O time between the original file layout and the visibility-based file layout using the Viswoman($512 \times 512 \times 1728$) data set with a data block size of $16 \times 16 \times 16$ voxels. Four transfer functions (TF1-TF4) were used. The PVFS strip size was 1024 bytes and 32 I/O servers were used.

5 Conclusions and Future Work

This paper introduces the concept of Perview Sampled Histogram to represent the visibility information for each data block in a parallel volume rendering algorithm. Such histograms are pre-computed for each block from a set of sample views. Perview Sampled Histogram can be used in parallel volume rendering algorithms to assist culling invisible volume blocks. The major advantage of the Perview Sampled Histogram is its independency of any transfer function. Since it is completely data-centric, visibility culling using Perview Sampled Histograms is more flexible and robust. Experimental results of the I/O performance show that the Perview Sampled Histograms are effective in predicting the pattern of block data I/O. Theoretically, Perview Sampled Histograms can work well beyond the 1D transfer function. But with higher dimensional transfer functions, the amount of information needed to be stored will grow. In the future, we plan to design more compact representation for the histograms to reduce the storage requirements. The concept of Perview Sampled Histograms can be used in time-varying volume rendering and we will explore this in the future.

References

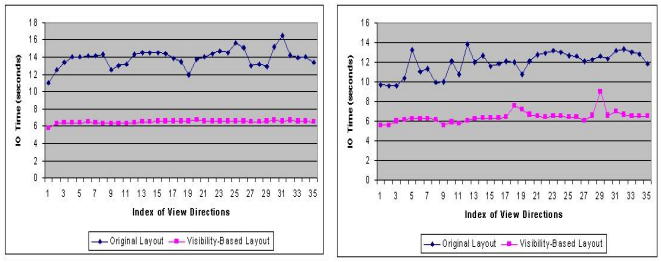
[1] W.-K. L. R. R. A. Ching, A. Choudhary and W. Gropp. Noncontiguous i/o through pvfs. In *IEEE International Conference on Cluster Computing '02*, 2002.



(a) I/O time for TF1 (b) I/O time for TF2

(c) I/O time for TF3 (d) I/O time for TF4

Figure 12: Comparisons of I/O time between the original file layout and the visibility-based file layout using the Viswoman($512 \times 512 \times 1728$) data set with a data block size of $16 \times 16 \times 16$ voxels. Four transfer functions (TF1-TF4) were used. The PVFS strip size was 4096 bytes and 32 I/O servers were used.



(a) I/O time for TF1 (b) I/O time for TF2

(c) I/O time for TF3 (d) I/O time for TF4

Figure 13: Comparisons of I/O time between the original file layout and the visibility-based file layout using the Viswoman($512 \times 512 \times 1728$) data set with a data block size of $16 \times 16 \times 16$ voxels. Four transfer functions (TF1-TF4) were used. The PVFS strip size was 8192 bytes and 32 I/O servers were used.

[2] D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5), May 1995.

[3] V. O. G. S. W. T. F. Isaila, G. Malpohl. Integrating collective i/o and cooperative caching into the clusterfile parallel file system. In *the 18th annual international conference on Supercomputing*, Oct. 2004.

[4] J. Gao and H.-W. Shen. Parallel view-dependent isosurface extraction using multi-pass occlusion culling. Oct. 2001.

[5] J. Gao and H.-W. Shen. Hardware-assisted view-dependent isosurface extraction using spherical partition. In *Joint EUROGRAPHICS-IEEE TCVG Symposium on Visualization*, pages 67–75, Oct. 2003.

[6] J. Gao and H.-W. Shen. Visibility culling using plenoptic opacity functions for large volume visualization. In *Visualization '2003*, pages 341–348, Oct. 2003.

[7] J. Gao and H.-W. Shen. Visibility culling for time-varying volume rendernig using temporal occlusion coherence. In *Visualization '2004*, pages 147–154, Oct. 2004.

[8] S. Guthe and W. Strasser. Advanced techniques for high-quality multi-resolution volume rendering. *Computers & Graphics*, 28(1):51–58, June 2004.

[9] K.-L. M. H-F. Yu and J. Welling. I/o strategies for parallel rendering of large time-varying volume data. In *Eurographics Symposium on Parallel Graphics and Visualization '04*, 2004.

[10] F. Hillier and G. Lieberman. *Introduction to Mathematical Programming*. McGraw-Hill, second edition, 1995.

[11] S. N.-C. R. S. P. HUANG, J. and K. MUELLER. A parallel splatting algorithm with occlusion culling. In *3rd Eurographics Workshop on Parallel Graphics and Visualization, Girona, Spain*, pages 125–132, 2000.

[12] F. Isaila and W. F. Tichy. Clusterfile: a flexible physical layout parallel file system. *CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE*, 15:653–679, 2003.

[13] B. J. R. H. J.-P. Prost, R. Treumann 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.

[14] J. L. T. J. Worringen and H. Ritzdorf. Fast parallel non-contiguous file access. In *ACM/IEEE SuperComputing '03*, Nov. 2003.

[15] J. L. T. J. Worringen and H. Ritzdorf. Improving generic non-contiguous file access for mpi-io. *Recent Advances in Parallel Virtual Machine and Message Passing Interface.*, 2840 of Lecture Notes in Computer Science:309–318, 2003.

[16] C. R. J. J.Z. Gao, J. Huang and S. Atchley. Distributed data management for large volume visualization. In *Visualization '2005*, Oct. 2005.

[17] M. Levoy. Display of surfaces from volume data. In *IEEE Computer Graphics and Applications '88*, volume 8, May 1988.

[18] M. Levoy. *Display of Surfaces from Volume Data*. PhD thesis, University of North Carolina at Chapel Hill, 1989.

[19] 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.

[20] P. J. S. H. C. D. MA, K.-L. and M. F. KROGH. Parallel volume rendering using binary-swap compositing. In *IEEE Computer Graphics and Applications*, volume 14, pages 59–68, 1994.

[21] N. MAX. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2), 1995.

[22] S. P. L. Y. H. C. PARKER, S. and P.-P. SLOAN. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, pages 233–238, 1998.

[23] W. G. R. Thakur and E. Lusk. Data sieving and collective i/o in romio. In *the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, 1999.

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

[25] J. G. S. Guthe, M. Wand and W. StraBer. Interactive rendering of large volume data sets. In *IEEE Visualization '02*, Oct. 2002.

[26] K. C. W-K. Liao and C. A. Collective caching: Application-aware client-side file caching. In *Symposium on High Performance Distributed Computing '05*, 2005.

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

[28] C. T. Y. Rubner and L. J. Guibas. A metric for distributions with applications to image databases. In *IEEE International Conference on Computer Vision,98*, Jan. 1998.

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

[30] C.-K. Yang and T.-C. Chiueh. I/o conscious volume rendering. In *the Joint Eurographics/IEEE TCVG Symposium on Visualizatation '01*, pages 263–272, May 2001.

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