CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **00**:1–17 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Interactive Direct Volume Rendering on Desktop Multicore Processors

Qin Wang† and Joseph JaJa

Institute for Advanced Computer Studies and Department of Electrical and Computer Engineering, University of Maryland, College Park, USA.

SUMMARY

We present a new multithreaded implementation for the computationally demanding direct volume rendering (DVR) of volumetric datasets on desktop multicore processors using ray casting. The new implementation achieves interactive rendering of very large volumes, even on high resolution screens. Our implementation is based on a new algorithm that combines an object order traversal of the volumetric data followed by a focused ray casting. Using a very compact data structure, our method starts with a quick association of data subcubes with fine-grain screen tiles appearing along the viewing direction in front to back order. The next stage uses very limited ray casting on the generated sets of subcubes while skipping empty or transparent space and applying early ray termination in an effective way. Our multithreaded implementation makes use of new dynamic techniques to ensure effective memory management and load balancing. Our software enables a user to interactively explore large data sets through direct volume rendering while arbitrarily specifying a 2D transfer function. We test our system on a wide variety of well-known volumetric datasets on a two-processor Clovertown platform, each consisting of a Quad-Core 1.86 GHz Intel Xeon Processor. Our experimental tests demonstrate direct volume rendering at interactive rates for the largest datasets that can fit in the main memory on our platform. These tests also indicate a high degree of scalability, excellent load balancing, and efficient memory management across the datasets used.

KEY WORDS: *Direct Volume Rendering, Volume Visualization, Multicore Processors, Multithreaded Algorithms, Parallel Algorithms*

*Correspondence to: Joseph JaJa, Department of Electrical and Computer Engineering, University of Maryland, College Park, MD. 20742. Email: joseph@umiacs.umd.edu
†Current address: CDA Group, The MathWorks,Inc., Natick, MA. Email: qwang@mathworks.com
Contract/grant sponsor: This work was partially supported by an NSF Research Infrastructure Grant; contract/grant number: CNS-04-03313

Received 1 December 2008
Revised 20 March 2009

1. INTRODUCTION

Due to the dramatic advances in imaging devices and computing technologies, volumetric datasets are now appearing in many engineering, science and medical applications at a very fast rate with increasingly larger sizes. The effective visualization of such datasets is critical to understand and explain the information contained in the datasets and to discover features, patterns, and trends of interest. Consider for example the data set produced by the ASCI team at Lawrence Livermore National Labs to simulate the fundamental mixing process of the Richtmyer-Meshkov Instability (RMI) in inertial confinement fusion and supernovae. This dataset consists of 270 time steps, each consisting of $2048^2 \times 1920$ volume of one-byte scalar field [†]. The data shows the characteristic development of bubbles and spikes and their subsequent merger and break-up over the 270 time steps. High resolution rendering allows elucidation of fine scale physics, and in particular enable observations of a possible transition from a coherent to a turbulent state with increasing Reynolds number. Existing visualization techniques are usually targeted for several orders of magnitude smaller data sets and do not scale well for terabytes scale data sets.

In this paper, we focus our attention on direct volume rendering (DVR) algorithms and their implementations on multicore processors. Unlike volume rendering through isosurfaces, DVR considers the volume as medium in which light can be absorbed, scattered, or emitted, as it passes through the volume. Optical properties of the volume elements such as absorption, scattering, and emission, are captured by the so called *transfer function*. Therefore, DVR has the potential of capturing much finer details present in diffuse, amorphous, or gelatin-like regions than is possible by simply using isosurfaces. Software techniques developed for DVR fall primarily into three main categories: Ray casting [14, 15], Splatting [34] and hybrid strategies such as the Shear-Warp algorithm [12]. Although the GPU-based technology currently delivers the fastest rendering of volumetric data, GPUs have primarily used specialized texture mapping for direct volume rendering [5, 11, 6], which severely limits the rendering flexibility and constrains the visualization quality. We concentrate here on software implementations of DVR algorithms on general purpose desktop multicore processors since they offer a very general programming environment with increasingly high performance capabilities.

Researchers have considered a number of parallel volume rendering schemes for ray casting [24], splatting [23] and shear-warp [13, 26] using either PC clusters or tightly coupled MIMD supercomputer systems. Although the splatting method avoids sampling over empty regions by projection of kernel functions, its parallel version seems to be inefficient and complicated in the way it utilizes the early termination technique [23]. Shear-warp generally produces low quality images due to its bilinear interpolation with severe artifacts showing up when changing the viewing direction around 45^o [12, 13]. Consequently, ray casting seems to be the preferred technique in terms of its technical simplicity and its ability to produce higher quality images. Nevertheless, the ray casting approach usually involves significantly more computation due to its pixel-by-pixel processing (and hence heavily depends on the screen

[†]http://www.llnl.gov/CASC/asciturb/

size), which makes it hard to achieve interactive rendering even for moderate size datasets. For a good overview and a comparison of the various volume rendering techniques, we refer the reader to [7, 16, 21].

In this paper, we present a new method to perform direct volume rendering using ray casting which is particularly well-suited for the emerging multi-core processors. The resulting software achieves better performance over previously published schemes and ensures excellent load balancing and extremely low overhead in managing the memory hierarchy typically present in multicore processors.

2. PREVIOUS WORK

Direct volume rendering by ray casting involves shooting a ray through each pixel on the screen along the viewing direction and directly computing the opacity and color of the pixel using sampled data at equal distance intervals along the ray. Given the low-albedo optical model described in [20], the final opacity and color of each pixel can be composited approximately in front-to-back order using a recursive Riemann sum [7] as in the formula below:

$$\alpha_{i+1} = \alpha_i + (1 - \alpha_i)\alpha_s; \quad C_{i+1} = C_i + (1 - \alpha_i)\alpha_s C_s \tag{1}$$

where C_i and α_i represent respectively the color and opacity accumulated through the first i ray segments, and α_s and C_s are usually mapped from the properties of the vertex s (such as scalar field value and/or gradient value) by a user-specified transfer function.

Since the direct volume rendering method by ray casting was first published by Levoy in [14], many alternate versions, such as those reported in in [15, 28, 25, 31, 33], have been introduced. Some acceleration techniques turned out to be effective. One is the early ray termination which stops further data sampling along the ray when the accumulated opacity gets close to one. Another is empty space leaping in that the ray skips sampling the cells where the opacity value is very small. However, space leaping requires either a hierarchical structure (i.e. multilevel-grid [25], octree [15], kd-tree [28], etc) which incurs a significant overhead, or a space-consuming per-vertex register structure such as DFB-jumping described in [31] which however can not accommodate interactively changing the transfer function. On the other hand, the idea of projecting non-empty cells onto screen was introduced to determine the first and last cells hit by each ray using either GPU hardware [27] or cell-template [32, 33]. But the schemes are inefficient in skipping empty regions between the bounded range. To skip empty space more effectively and reduce the overhead of repeated hierarchical traversal, an object-oriented ray casting method is proposed in [22], which performs ray casting right after each cell projection and processes one cell after another in front-to-back octree traversal order. The limitation of this approach is that there is no straightforward way to parallelize this algorithm which involves in-order traversal of octree.

The parallelization of the general ray casting method for direct volume rendering has been described by many researchers [24, 4, 17, 18, 25, 2]. The approaches used can be classified according to three categories: image-space, object-space and hybrid. The image-space parallel scheme is easier to implement due to the independent pixel-by-pixel approach of ray casting but makes it quite hard to achieve good load balancing and cache coherence due to the

irregular data access pattern and the unpredictable image complexity on the screen with varying transfer function and viewing direction. Nieh presented a parallel scheme for the ray casting algorithm on shared-memory MIMD system in [24] and achieved the scalability around $70 \sim 80\%$ by utilizing dynamic task stealing from other processors at runtime. Other similar implementations [4, 25] yield relatively the same scalability performance. On the other hand, the object-space partition scheme is relatively popular on distributed systems. The basic idea relies on dividing the dataset evenly in a certain way or randomly among the processors in the hope of achieving good load balancing and regular memory access pattern [17, 18, 19]. But, these algorithms require additional merging and re-sorting of all the ray segments to produce correct rendering of images with significant synchronization time for the image composition stage. Moreover, the early ray termination condition can not be easily applied under object-space partitioning schemes. Using a completely different strategy, hybrid parallel shear-warp algorithms [13, 26] require additional 2D image warping process and the image quality does not in general match those produced by the other algorithms. Another hybrid parallel scheme [2] divides full screen into small image tiles and the dataset hierarchically into bricks, but incurs a significant overhead in order to achieve load balancing with dynamically assigned both image tiles and data bricks among processors. We also note that an implementation of ray tracing, which is more general than ray casting, was reported in [3] on the Cell Broadband Engine Architecture, a specialized multi-core processor with peak performance up to 200GFlops.

Finally we note that the transfer function, which maps the material properties of the data into opacity and colors, is critical in capturing and rendering the features and patterns of interest. We pay a particular attention to how the transfer function is constructed. Initially, the transfer function was applied at the preprocessing step as in [15], which does not allow interactive modification and rendering. More recently, researchers have allowed users to specify and modify the transfer function at run time using a friendly interface, and introduced multi-dimensional transfer functions that can be much more effective in capturing complex patterns and features [8, 9]. Furthermore, the pre-integral transfer function technique was introduced in [5] to suppress the artifacts stemming from discrete sampling along the ray. Our scheme supports the interactive construction of multi-dimensional transfer functions while using the pre-integral technique.

3. A NEW MULTITHREADED STRATEGY

We start by introducing some terminology that will be used to describe our algorithm. A *data cell* refers to the $3D$ rectilinear region enclosed by 8 neighboring vertices. A *subcube* is simply a pre-defined fixed number of spatially adjacent data cells (e.g., $4 \times 4 \times 4$ or $8 \times 8 \times 8$), and a *data block* or simply *block* refers to a varying number of neighboring data cells forming a $3D$ rectilinear region. Note that a data cell and subcube are special cases of a data block. Given a transfer function and a preset small positive value ϵ, we characterize data blocks into one of the following types: *transparent block*, each vertex of which has opacity $< \epsilon$; *semi-transparent block*, some vertices of which have opacity $< \epsilon$ while other vertices have opacity $\geq \epsilon$; and *diffuse block*, all the vertices of which have opacity $\geq \epsilon$.

Briefly, our strategy is an extension of our isosurface ray casting algorithm presented in [30]. Our DVR algorithm proceeds in two main phases. Phase I involves a coarse traversal of the data using an octree representation whose leaves represent subcubes. The main purpose of this phase is to determine for each small screen tile (2×2 in our implementation) a list of non-transparent data blocks (that is, semi-transparent and diffuse blocks) encountered by the packet of rays through the image tile in a front-to-back order. We place an upper bound k on the number of such data blocks for all the image tiles. This will substantially reduce the traversal time without affecting the rendering quality. A simple static allocation of the workload among the processor cores will guarantee excellent performance for this phase. Note that this phase is significantly different than the corresponding phase of the algorithm in [30] since we now have to deal with diffuse and semi-transparent blocks, whose sizes can be arbitrary and are determined at run time.

Phase II consists of shooting rays through fine-grain image tiles using the ordered sets of data blocks generated during Phase I. The blocks are processed in a front-to-back order and early termination is applied on each ray when possible. In the case when k blocks are processed with a resulting opacity short of the threshold, we extend the ray traversal further across the dataset to continue compositing along the view direction. This phase, which is much more computationally demanding than Phase I, requires complex and dynamic allocation of tasks and memory management schemes to guarantee efficiency and scalability, which will be explained later. In particular, a special weight function is used during the dynamic allocation of tasks to the different threads.

We next provide more details about our scheme.

3.1. Preprocessing

We organize our volumetric data into a coarse grid of equal-sized subcubes, where the scalar field values within each subcube are stored contiguously in a pre-defined order and the subcube is identified by the coordinates of a pre-specified corner. To work with 2D transfer functions, we compute the normal magnitude for each vertex by the method of center difference. We use an octree to index the data within a subcube and store the minimum and maximum density values and normal magnitudes in each node with the leaves corresponding to $2 \times 2 \times 2$ data cells. In addition, we build a BONO (Branch-On-Need Octree) [35] tree on the coarse grid consisting of the subcubes, augmented as usual by the appropriate min/max value ranges of density and magnitude of normals. Note that the subcubes are always chosen so that each dimension is a power of two, and hence the use of octrees to index their scalar data. We will later show that our indexing structure is quite compact for all our datasets and the preprocessing step can be performed very quickly even for very large datasets.

3.2. Block Discrimination Method

We devise a new method that is able to quickly classify data blocks as *transparent*, *semi-transparent*, or *diffuse* to allow for efficient empty space leaping based on any 2D transfer function selected by the user. The method creates a discrimination function implemented as a table generated quickly once the 2D transfer function is specified. The construction procedure

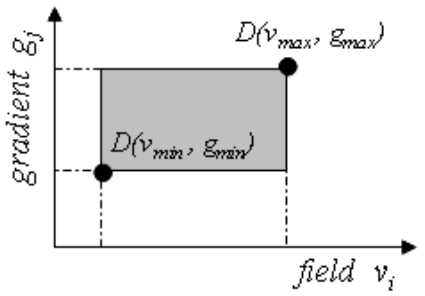

Figure 1. An illustrative diagram for 2D-dimensional discriminate function $D(v_i, g_j)$. The shaded area is bounded by discrete pair values (v_{min}, g_{min}) and (v_{max}, g_{max}).

of such a table $D(v_i, g_j)$ with discrete values v_i for the scalar field and g_j for the gradient magnitude amounts to the following computation. Given a small preset value ϵ, the value of $D(v_i, g_j)$ is set equal to the number of pairs (v_k, g_l) such that $v_k \leq v_i$, $g_l \leq g_j$, and $opacitiy(v_k, g_l) > \epsilon$.

During the BONO tree traversal step in Phase I, the minimum and maximum density and gradient pairs, (d_{min}, h_{min}) and (d_{max}, h_{max}) are retrieved from a BONO tree node. The discrimination method finds the largest discrete value v_{min} smaller than d_{min}, and the smallest discrete value v_{max} which is larger than or equal to d_{max}, and similarly for g_{min} and g_{max} with respect to the gradient magnitudes h_{min} and h_{max}. Using the table D we can compute the non-negative difference value δ of the D values over the box area bounded by the corresponding pairs (v_{min}, g_{min}) and (v_{max}, g_{max}) as shown in Figure 3.2, i.e, $\delta = D(v_{max}, g_{max}) - D(v_{min-1}, g_{max}) - D(v_{max}, g_{min-1}) + D(v_{min-1}, g_{min-1})$. Then, the δ value can be used to classify the blocks as follows:

If $\delta = 0$, the block is transparent; If δ is equal to the number of pairs in the box determined by (v_{min}, g_{min}) and (v_{max}, g_{max}), the block is classified as diffuse. Otherwise, the block is classified as semi-transparent. Note that in the latter case the block could be *transparent*, which means we will process it unnecessarily but clearly this will not affect in any way the correctness of the rendering process.

Since δ represents the number of discrete values of the pairs (v_i, g_j) which are mapped to an opacity larger than ϵ, it is easy to observe that our designation of transparent and diffuse blocks is always accurate. Note that our block discrimination method involves only the transfer function and the min/max values of the scalar field and gradient magnitude of the data blocks, and hence can be processed very quickly via table look-up and incorporated into Phase I of our scheme.

3.3. Algorithm Overview

After the preprocessing step, Phase I of our strategy can be implemented as follows. The BONO tree is traversed starting from the root. Assume we reach a node v of the tree. The block discrimination process is applied with min/max range values retrieved from the node. Using the transfer function table, if the node is classified as semi-transparent, we project the minimum bounding box of v onto the screen, and consider all the sizes of the lists corresponding to all the image tiles within the projection. Should any of the lists be of size smaller than the preset value of list upper bound k, we proceed and traverse the children of v in a front-to-back order relative to the viewpoint. Otherwise we skip the subtree rooted at node v. Once a subcube is reached, we augment the lists corresponding to screen tiles, which overlap the projection on the screen, with a pointer to the octree representing the subcube. Should the node be classified as diffuse, we follow the same process but traverse the corresponding block as a grid structure. Since the nodes are traversed and projected along the viewing order, a falsely classified semi-transparent node will not produce an error and is more likely to get corrected once we traverse down its subtree. Therefore we end up with lists, each with no more than k blocks for each small image tile, such that the blocks are arranged in front-to-back order relative to the viewpoint.

During Phase II, we shoot rays through the pixels using the corresponding block lists. Each ray proceeds using grid traversal [1] across these blocks along the viewing direction while compositing the colors and updating the opacity values using Equations (1), as well as applying early ray termination whenever possible. If the accumulated opacity is lower than the threshold (e.g., 0.99 in our setting) after the ray goes through up to k blocks on the list, the ray continues traversal further across the dataset via the BONO tree structure until either reaching the threshold or going beyond the date space boundary. We will later show that the percentage of such continued ray traversals is minimized given a good value for k.

3.4. Multithreaded Implementation on Multicore Processors

The main trend in high performance computing systems - general purpose, embedded, DSPs, or GPUs - is clearly to include more cores on a single chip. It is widely expected that the number of cores per chip will double every 18-24 months, and this trend is likely to continue at least for the next 10 years. The architectures of the current and emerging multicore processors vary significantly, all including several levels of memory hierarchy, SIMD or vector type operations, and multithreaded cores. These systems offer unprecedented opportunities for speeding up demanding computations on a single processor if the available resources can be effectively used. Typically a multi-threaded program is used to specify the tasks executed on the cores, which also includes coordination and synchronization tasks to ensure correct execution.

The objective of a multi-threaded implementation of our algorithm is to allocate the computation among the different cores in such a way as to (i) achieve an almost equal distribution of the workload regardless of the input and the viewing position or direction; (ii) ensure as little communication and coordination among the threads as possible; and (iii) make effective use of the memory hierarchy typically present in multicore architectures. This last requirement is especially important when dealing with irregular data movement for large

scale datasets as is the case for our volume rendering problem. As we have seen, many previous parallel implementations of ray casting essentially amount to decomposing the screen equally among the processors followed by each processor shooting rays through the pixels of its screen tile. In general, such an implementation results in poor load balancing and inefficient use of the memory hierarchy. We use a novel strategy to address these bottlenecks.

Given a p-core processor, we describe a multithreaded implementation that achieves fine-grain load balancing while ensuring effective utilization of the memory hierarchy with little coordination or synchronization overhead. As described in the previous section, we preprocess the data to create a BONO tree such that each leaf corresponds to a subcube and each tree node contains the minimum and the maximum value ranges of density and magnitude of normals. For the multithreaded version, we augment the preprocessing step by a compact representation of fine-grain partition of the screens (each corresponding tile is of size 4×4 or 8×8) arranged in a Z-order. This will ensure a high degree of spatial locality, which will make the data traversal during ray casting quite effective.

Phase I, which takes roughly 10% of the total time of the sequential algorithm, is implemented using the traditional work partitioning technique. That is, the screen is divided equally into almost equal-size and contiguous tiles, and each thread performs a BONO-tree traversal restricted to its own tile. Hence a traversal of a node v is followed by traversing the children in a front to back order only if the projection of the minimum bounding box of v intersects the thread's tile and there is at least one list associated 2×2 adjacent pixels in the tile which is not complete. In our case, the BONO tree is very small compared to the size of the data, and a typical thread needs to access a small fraction of the total BONO nodes.

Phase II requires fine-grain dynamic allocation of ray casting operations as follows. We first perform an estimate of the workload on each of the fine-grain tiles, arranged in Z-order, which were generated during the preprocessing step. For each such tile, we sum the sizes of the blocks on all the tile's lists generated during Phase I. Note that these blocks could be of different sizes, and are either semi-transparent or diffuse since transparent blocks are skipped. We let W be the total of the weights of all the fine-grain tiles, and create an ordered sequence of tasks as follows. The first p tasks consist of the initial set of tiles (in Z-order) whose total weight is $\frac{W}{2}$ arranged into p equal-weight groups. Note that each such task consists of a pair of indices indicating the first and the last tile (in Z-order) on that task. We continue through the Z-ordered set of fine tiles, but now picking the sequence of tiles of total weight approximately $\frac{W}{4}$, and continue in this fashion until all the fine-grain tiles are exhausted. The tasks are arranged in the same order they were created. As we will show later, this part of Phase II can be executed very quickly even for high-resolution screens.

The ray casting operations are allocated dynamically as follows. Initially, each thread grabs a task from the ordered sequence of tasks, and starts executing the ray casting operations (using SIMD operations if possible) corresponding to each task. Once a thread completes the processing of its task, it grabs the first available task remaining in the sequence of tasks and immediately starts ray casting through the pixels on this task. The process continues until all the fine-grain image tiles are processed.

The dynamic allocation of ray casting achieves an optimal trade-off between two conflicting requirements. The first is to ensure load balancing through the execution of fine-grain tasks. The second is to minimize the amount of coordination and synchronization by making

the number of tasks as small as possible, thereby implying coarse-grain tasks. In our implementation we start with fairly large tasks, and decrease geometrically the sizes of these tasks until we reach fine-grain tasks during the processing of the last p tasks.

4. EXPERIMENTAL RESULTS

We have conducted extensive tests of our new multithreaded algorithm on six datasets with sizes varying from 150MB to 7.5GB. We select six viewpoints with zenith angles $\phi = \{15^o, 45^o, 75^o\}$ and azimuth angles $\theta = \{12^o, 102^o\}$ in spherical coordinates and report their average frame rates. The tests were conducted on the Clovertown platform, consisting of two Quad-Core Intel 1.80 GHz Xeon Processors 5320. Each dual-core shares an L2 cache of size 4MB, and hence the total L2 cache available is 8MB. Our Clovertown platform has 8GB of main memory, which constitutes an upper bound on the size of the datasets used in our tests. We generate visualizations with screen resolutions 512^2 and 1024^2, and report corresponding performance for Far-view and Close-view settings.

In general, the work involved in direct volume rendering (DVR) using ray casting can be divided into two parts: the first is ray traversal across space, and the second is interpolation and color composition along the rays through non-transparent region. Hence, we report on the number of ray traversal steps (excluding those steps that contribute to interpolation and color composition) and the number of color composition steps separately. Also measured are the execution times during Phase I and Phase II of our multithreaded scheme with a varying number of cores of our Clovertown platform. Note that excluding Phase I of our scheme (i.e., setting block-list length to $k = 0$) converts our algorithm into a variant of the well-known image-based DVR approaches, such as those proposed by [15] and [25], using our new multithreaded implementation. We observe considerable performance improvement when Phase I of our scheme is applied to accelerate the ray casting process using suitable values of k. The detailed performance numbers will be presented after introducing the datasets used.

4.1. Datasets Used

The six datasets used in our tests (shown in Figure 2 as rendered by our scheme) belong to the following three major domains: geometric objects (Bonsai$-512^2\times308\times$2B, XTree$-512^2\times512\times$2B), medical CT scan (Cadaver$-512^2\times424\times$2B, Prone$-512^2\times462\times$2B, VF$-512^2\times1734\times$2B), and scientific simulation (RMI$-2048^2\times1920\times$1B). In our scheme, the subcubes consisting of background voxels do not need to be indexed, resulting in a very compact representation in general. As shown in Table I for the six datasets, the corresponding BONO trees are quite compact and the space required by all the octrees representing the subcubes is in general below 40% of original data size despite the fact that we augment the data with the minimum and maximum gradient values within each octree node to make use of the $2D$ transfer function. During preprocessing, we also generate the $2D$ histograms in terms of density and gradient values as shown in Figure 3 to enable us to construct the $2D$ transfer function for volume rendering.

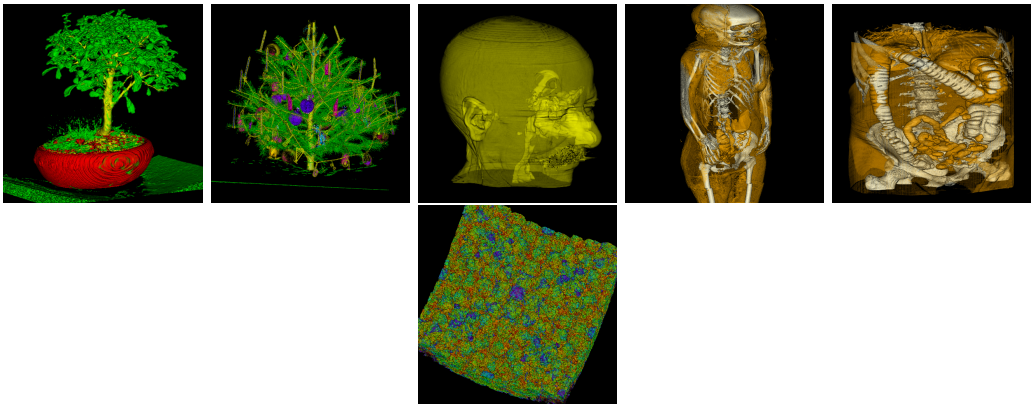

Figure 2. The six datasets used in our tests. From left to right, the datasets are: Bonsai, XmasTree(XTree), Cadaver, VisFemale(VF), Prone and LLNL Richtmyer-Meshkov Instability (RMI) respectively.

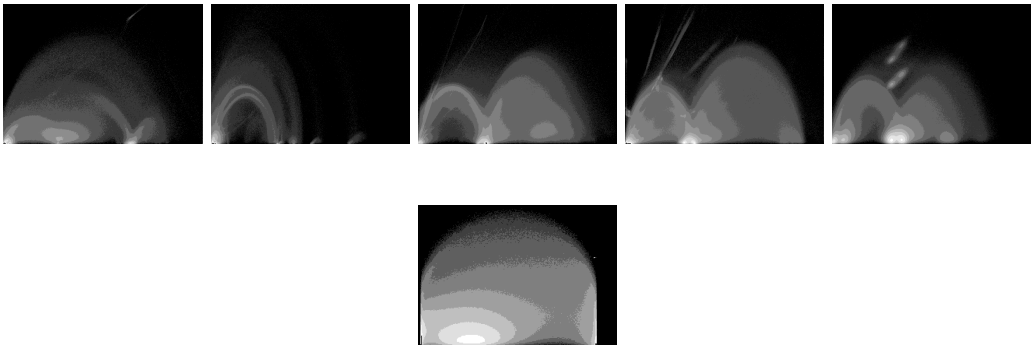

Figure 3. The 2D histograms representing density values versus the magnitudes of the gradient with 0 at the lower left corner of the six datasets used in our tests. From left to right, the datasets are: Bonsai, XmasTree, Cadaver, VisFemale, Prone and LLNL respectively.

Table I. Sizes of grid and indexing structure for various datasets used along with their preprocessing time. The dimension of subcube is $4 \times 4 \times 4$ for all the tested datsets except for the LLNL RMI dataset that has $8 \times 8 \times 8$-subcubes.

	Size (MB)			Time
Dataset	Grid	BONO	Subcubes	(sec)
Bonsai	154	8.5	44.2 (28.7 %)	17.1
Cavader	212	8.2	45.1 (21.3 %)	21.2
Prone	231	17.1	94.3 (36.8 %)	23.4
XTree	256	17.4	94.0 (36.7 %)	25.6
VF	867	45.0	159.5 (18.4 %)	88.3
RMI	7,680	51.27	1,596 (20.8 %)	520.0

4.2. Overall Performance

Before presenting our performance results in terms of execution time, scalability, and load balancing, we make a few observations regarding the value of the upper bound k based on the extensive tests conducted on our datasets. The first observation is that the best value of k ranges between 40 and 75 depending on the dataset used, and that the time it takes to generate the block lists in Phase I is less than 10% of the total time for all the tested k values. The second observation is that the time that Phase II takes for compositing beyond the k blocks goes down very quickly initially as k increases and then somewhat levels off when an optimal value of k is reached.

We also measure the percentage of number of color compositing steps over the total number of steps (including compositing and traversal steps), called *Empty Space Leap(ESL) Efficiency*, involved in Phase II as well as the percentage of the number of compositing steps beyond the k blocks. Some of the experimental results are illustrated in Table II. It is clear that the time of Phase I is well below 10% of Phase II time while the number of compositing steps beyond k blocks is a small percentage of the total steps. The numbers of compositing and ray traversal steps undertaken in Phase II for various datasets and view types under corresponding optimal k values as shown in Table III confirm that the number of traversal steps is dramatically reduced by using Phase I, which in turn results in an increase on ESL efficiency by a factor of up to 4.

We now turn to the overall performance results of our algorithm in terms of *fps* on visualizing the six datasets using up to 8 cores on our Clovertown system under 512^2 and 1024^2 screen resolutions respectively. Table IV summarizes the results, and shows that the frame rate of our algorithm can reach $2 \sim 6fps$ for 512^2 screen and $1 \sim 2fps$ for a screen of resolution 1024^2, which is substantially better than any of the previously published results on DVR algorithms running on general purpose processors. For example, on 1024^2 screen, the performance improvement over the well-known octree algorithm (even using our new multi-threaded implementation) can range from 40% to over 100% just because of our hybrid scheme.

We note that the performance of our algorithm depends on the chosen transfer function as does any variant of the ray casting algorithm. However, even in the worst case scenario, our

Table II. Measured execution time of our DVR algorithm for Phase I and II as well as the number of compositing steps within k blocks on the lists and beyond these blocks on single-core of Clovertown for a screen of resolution 1024^2 and different view type settings

	Time (sec)		# of Compositions ($\times 10^3$)			
Datasets	I	II	k	k-list	beyond	(%)
Bonsai	0.32	3.87	70	10,751	164	1.50
Cadaver	0.24	4.52	75	11,926	350	2.85
Prone	0.67	8.21	70	19,966	387	1.90
XTree	0.38	5.24	70	16,243	284	1.72
VF(far)	0.34	5.26	70	13,536	186	1.36
VF(close)	0.56	8.20	70	29,158	334	1.13
RMI(far)	0.34	5.98	40	15,901	289	1.79
RMI(close)	0.67	7.44	60	22,862	346	1.50

Table III. Number of composition(CP) steps and number of ray traversal steps undertaken to skip empty space during DVR ray casting with $k = 0$ and $k =$ optimal k^* in our algorithm under screen size of 1024^2 for various datasets and view types. ESL Efficiency(ESL Eff.) is defined as the percentage of number of compositions over number of total steps including composition and traversal steps.

# of steps	CP	Traversal		ESL Eff. (%)		
($\times 10^6$)	both	$k = 0$	k^*	$k = 0$	k^*	ratio
Bonsai	10.9	50.1	4.5	17.9	70.9	4.0
Cadaver	12.3	74.7	8.3	14.1	59.7	4.2
Prone	20.4	122.1	14.9	14.3	57.8	4.0
XTree	16.5	83.8	7.5	16.5	68.8	4.2
VF(far)	13.7	72.1	12.6	16.0	52.1	3.3
VF(close)	29.5	146.7	12.9	16.7	69.7	4.2
RMI(far)	16.2	51.0	3.4	24.1	82.5	3.4
RMI(close)	23.2	69.7	4.6	25.0	83.4	3.3

algorithm will achieve improved performance than previously published algorithms due to the processing and to Step 1. In all our tests, we selected transfer functions that highlight all the significant structures in the volumes, and in fact our renderings seem to reveal more details than previous renderings of the same datasets. We believe that our frame rates represent realistic expected performance of our algorithm for most scientific applications.

We now examine the scalability of our algorithm in terms of the number of cores and the corresponding loads on the cores induced by our multi-threaded dynamic allocation scheme. Table V shows the average frame rate over six views for the two different settings of the viewpoint on the LLNL RMI dataset using a varying number of cores. The results are for 512^2 and 1024^2 screen resolution respectively, both showing a scalability well above 90% as

Table IV. Measured DVR performance on 8-core Clovertown in fps for our scheme on 512^2 and 1024^2 screen for various datasets

Screen size	512^2	1024^2
Dataset	FPS	FPS
Bonsai	6.58	1.84
Cadaver	5.46	1.60
Prone	2.99	0.89
XTree	4.42	1.36
VF(far)	3.92	1.31
VF(close)	2.54	0.89
RMI(far)	3.76	1.26
RMI(close)	3.85	0.98

Table V. Average frame rate of our DVR algorithm on Clovertown for the LLNL RMI dataset at time step 250 under a varying number of CPU cores using 512^2 and 1024^2 screen resolution.

Screen Size	512^2		1024^2	
# of Cores	Far	Close	Far	Close
1	0.48	0.51	0.16	0.13
2	0.96	1.02	0.31	0.25
4	1.93	2.03	0.64	0.50
8	3.76	3.85	1.26	0.98
Scalability	97.0%	94.2%	98.7%	95.1%

we increase the number of cores to the maximum of eight available on our platform. The scalability results for the other data sets are very similar.

Table VI illustrates the loads on the different threads for the LLNL RMI dataset for both the far and close views. Note that the numbers of compositing and ray traversal steps are almost evenly distributed among the threads regardless of the viewpoint for Phase II which takes up more than 90% of the total computational load. Therefore the loads are extremely well-balanced among the different threads.

5. CONCLUSION

In this paper we presented a new hybrid strategy for direct volume rendering by ray casting. The resulting algorithm uses an object order traversal coupled with a novel block discrimination method to generate a list of blocks along the viewing direction for each small image tile, and achieve efficient empty space skipping and early ray termination when shooting rays through the pixels. We have shown that the total size of our indexing structure is very compact and that our multithreaded implementation achieves interactive rates for the largest datasets that can

Table VI. The work from two Phases distributed among eight threads running among 8-core for 1024^2 screen along with their corresponding individual execution time. The tests are done on LLNL RMI dataset for both far and close views. The work load of Phase II is measured by the number of composition steps and the number of ray traversal steps. Total time includes the synchronization time and writing time of the frame buffer.

		Number of ($\times 10^3$)		Time (msec)		
View & Proc No.		Compo-sitions	Traversal Steps	Phase I	Phase II	total
Far	0	421	2,036	40	721	781
	1	387	2,033	50	721	781
	2	392	2,035	58	720	780
	3	447	1,994	45	720	780
	4	392	2,022	46	720	780
	5	470	2,028	57	721	781
	6	391	2,033	47	721	781
	7	427	2,020	40	722	782
$\frac{\sigma}{Ave} \times 100\%$		6.93	0.64	13.35	0.09	0.08
Close	0	670	2,917	72	932	1,007
	1	758	2,897	70	932	1,007
	2	689	2,937	73	933	1,008
	3	767	2,915	71	932	1,007
	4	771	2,900	71	933	1,008
	5	657	2,969	72	932	1,007
	6	793	2,921	71	933	1,008
	7	739	2,920	70	935	1,010
$\frac{\sigma}{Ave} \times 100\%$		6.56	0.73	1.36	0.10	0.10

fit in the main memory of our platform. Moreover we have shown a high degree of scalability, excellent load balancing, and effective memory management on multicore processors. Our system also allows the user to easily and interactively specify any 2D transfer function to generate the desired visualization.

REFERENCES

1. J. Amanatides and A. Woo, A Fast Voxel Traversal Algorithm for Ray Tracing, Eurographics '87, pp. 3-10, 1987.
2. C. Bajaj, I. Ihm, G. Koo, and S. Park, Parallel Ray Casting of Visible Human on Distributed Memory Architectures, In Data Visualization, Eurographics, pp. 269-276, May 1999.
3. C. Benthin, I. Wald, M. Scherbaum and H. Friedrich, Ray Tracing on the Cell Processor, Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pp. 15-23, 2006.
4. J. Challinger, Scalable Parallel Volume Raycasting for Nonrectilinear Computational Grids, Proc. of Parallel Rendering Symposium93, pp. 81-88, 1993.
5. K. Engel, M. Kraus, and T. Ertl, High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading, Proc. of SIGGRAPH Graphics Hardware Workshop01, pp. 9-16, 2001.
6. T. Foley and J. Sugerman, KD-Tree Acceleration Structures for a GPU Raytracer, In Proceedings of Graphics Hardware 2005.
7. A. Kaufman and K. Mueller, Overview of Volume Rendering, Chapter of Visualization Handbook, 2005
8. G. Kindlmann, and J. Durkin, Semi-automatic generation of transfer functions for direct volume rendering, Symp. Volume Visualization98, pp. 79-86, 1998.
9. J. Kniss, G. Kindlmann, and C. Hansen, Multidimensional Transfer Functions for Interactive Volume Rendering, IEEE Transactions on Visualization and Computer Graphics, vol. 8, no. 3, pp. 270-285, 2002.
10. A. Knoll, S. G. Parker and C. D. Hansen, Interactive Isosurface Ray Tracing of Large Octree Volumes, Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pp. 115124, 2006.
11. J. Kruger and R. Westermann, Acceleration Techniques for GPU-based Volume Rendering, IEEE Visualization 2003.
12. P. Lacroute and M. Levoy, Fast volume rendering using a shear-warp factorization of the viewing transformation, Proc. SIGGRAPH 94, pp. 451-458, 1994.
13. P. Lacroute. Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization. IEEE Parallel Rendering Symposium 95 Proceedings, pp. 1522, 1995.
14. M. Levoy, Display of surfaces from volume data, IEEE Comp. Graph. and Appl., vol. 8, no. 5, pp. 29-37, 1988.
15. Marc Levoy, Efficient ray tracing of volume data. ACM Transactions on Graphics, 9(3): pp. 245-261, July 1990.
16. G. Marmitt, H. Friedrich, and P. Slusallek, Interactive Volume Rendering with Ray Tracing, Eurographics State of the Art Reports, 2006.
17. K. Ma., Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures, Proc. of Parallel Rendering Symposium95, pp. 23-30, 1995.
18. K. Ma, and T. Crockett, A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data, Proc. of Parallel Rendering Symposium 97, 1997.
19. M. Matsui, F. Ino and K. Hagihara, Parallel Volume Rendering with Early Ray Termination for Visualizing Large-Scale Datasets, ISPA 2004, pp. 245-256, 2004
20. N. Max, Optical models for direct volume rendering, IEEE Trans. Vis. and Comp. Graph., vol. 1, no. 2, pp. 99-108, 1995.
21. M. Meiner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis, A practical comparison of popular volume rendering algorithms, Symposium on Volume Visualization and Graphics 2000, pp. 81-90, 2000.
22. B. Mora, J. P. Jessel and R. Caubet, A new object-order ray-casting algorithm, Proceedings of the conference on IEEE Visualization 2002, pp. 203210, October, 2002.
23. K. Mueller, N. Shareef, J. Huang, and R. Crawfis, Highquality splatting on rectilinear grids with efficient culling of occluded voxels, IEEE Transactions on Visualization and Computer Graphics, vol. 5, no. 2, pp. 116-134, 1999.
24. J. Nieh, and M. Levoy, Volume Rendering on Scalable Shared- Memory MIMD Architectures, Proc. of Volume Visualization Symposium, pp. 17-24, 1992.
25. S. Parker, M. Parker. Y. Livnat, P. Sloan, C. Hansen, and P. Shirley, Interactive Ray Tracing for Volume Visualization, IEEE Transactions on Visualization and Computer Graphics, vol. 5, no. 3, pp. 238-250, 1999. Smart Hardware-Accelerated visualisation 2003, pp. 231-238, 2003
26. J. P. Schulze, U. Lang, The Parallelization of the Perspective Shear-Warp Volume Rendering Algorithm, Proceedings of the 4th Eurographics Workshop on Parallel Graphics and Visualization, pp. 61-69, 2002
27. L. Sobierarjski and R. Avila, A Hardware Acceleration Method for Volume Ray Tracing, Proceedings of the 6th conference on IEEE Visualization 1995, pp. 2734, 1995.

28. K. Subramaniam and D. Fussel. Applying Space Subdivision Techniques to Volume Rendering. In Proc. IEEE Visualization, pp. 150-158, Oct. 1990.
29. I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H. P. Seidel, Faster Isosurface Ray Tracing using Implicit KD-Trees, IEEE Transactions on Computer Graphics and Visualization, vol. 11, no. 5, pp. 562572, 2005.
30. Q. Wang and J. JaJa, Intreactive High Resolution Isosurface Ray Casting on Multi-core Processors, IEEE Visualization and Computer Graphics 2007, Dec 2007
31. M. Wan, Q. Tang, A. Kaufman, Z. Liang, and M. Wax, Volume Rendering Based Interactive Navigation within the Human Colon, Proc. of IEEE Visualization99, pp.397-400, 1999.
32. M. Wan, Q. Tang, A. Kaufman, Z. Liang, and M. Wax. Volume Rendering Based Interactive Navigation within the Human Colon. In Proc. IEEE Visualization, pp. 397400, 1999.
33. M.Wan, A. Sadiq, and A. Kaufman. Fast and Reliable Space Leaping for Interactive Volume Rendering. In Proc. IEEE Visualization, pp. 195202, Oct. 2002. projection,
34. L. Westover, Footprint evaluation for volume rendering, SIGGRAPH 90, pp. 367-376, 1990.
35. J. Wilhelms and A.Van Gelder, *Octrees for faster isosurface generation*, Computer Graphics(San Diego Workshop on Volume Visualization), vol. 24, pp. 57–62, 1990.