Interactive High Resolution Isosurface Ray Tracing on Multi-Core Processors ⋆

Qin Wang [a] Joseph JaJa [a,1]

[a] *Institute for Advanced Computer Studies, Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742 USA.*
{*qinwang, joseph*}*@umiacs.umd.edu*

Abstract

We present a new method for the interactive rendering of isosurfaces using ray tracing on multi-core processors. This method consists of a combination of an object-order traversal that coarsely identifies possible candidate $3D$ data blocks for each small set of contiguous pixels, and an isosurface ray casting strategy tailored for the resulting limited-size lists of candidate $3D$ data blocks. Our implementation scheme results in a compact indexing structure and makes careful use of multi-threading and memory management environments commonly present in multi-core processors. While static screen partitioning is widely used in the literature, our scheme starts with an image partitioning for the initial stage and then performs dynamic allocation of groups of ray casting tasks among the different threads to ensure almost equal loads among the different cores while maintaining spatial locality. We also pay a particular attention to the overhead incurred by moving the data across the different levels of the memory hierarchy. We test our system on a two-processor Clovertown platform, each consisting of a Quad-Core 1.86 GHz Intel Xeon Processor, and present detailed experimental results for a number of widely different benchmarks. We show that our system is efficient and scalable, and achieves high cache performance and excellent load balancing, resulting in an overall performance that is superior to any of the previous algorithms. In fact, we achieve interactive isosurface rendering on a screen with 1024^2 resolution for all the datasets tested up to the maximum size that can fit in the main memory of our platform.

⋆ The version of this work has been submitted to IEEE Transaction on Visualization and Computer Graphics (TVCG), 2007.

[1] This work is supported by the NSF research infrastructure grant CNS-04-03313.

24 August 2007

1 Introduction

Rendering isosurfaces is widely recognized as an effective approach for the visual exploration, computational analysis, and manipulation of volumetric datasets. Such datasets are appearing at a very fast rate with increasingly larger sizes due to the dramatic advances in imaging instruments and computing technologies. In particular, as the speed of processors continues to improve, researchers are performing large scientific simulations to study very complex phenomena at increasingly finer resolution scales. Such simulations end up generating very large datasets that need to be examined at a relatively fine scale. One such set is the Richtmyer-Meshkov instability dataset produced by the ASCI team at Lawrence Livermore National Labs (LLNL), consisting of 270 time steps, each consisting of $2048^2 \times 1920$ volume of one-byte scalar field [2]. The most commonly used method for visualizing isosurfaces is to compute a triangular mesh approximation of the isosurface followed by rendering the triangles through a graphics hardware. This method was popularized by the introduction of the Marching Cubes algorithm in [9], and has since been improved using a wide number of different techniques (e.g., [1,16]). The resulting efficient schemes create either spatial or range-based indexing structures through a preprocessing step in such a way as to enable the extraction and rendering of the isosurface in time that primarily depends on the size of the triangular mesh of the isosurface rather than the size of the input dataset.

A major drawback of the above scheme is the extraction of a possibly very large triangular mesh for each specific isovalue, a large part of which may not be visible from any specific viewpoint. The size of this view-independent triangular mesh tends to increase significantly as the size of the dataset grows or as the structure of the isosurface becomes more complex. In fact, it is not uncommon to encounter isosurfaces whose triangular meshes, as generated by any variant of the Marching Cubes algorithm, consist of hundreds of millions of triangles such as the Richtmyer-Meshkov instability dataset from LLNL or some of the visible human datasets from National Library of Medicine [3]. One way to address this drawback is to only extract and render the triangles that cover the portions of the isosurface, which are visible from the viewpoints of interest. A scheme for view-dependent rendering was introduced in [8], in which they showed that the view dependent visualization can significantly reduce the complexity of the rendered surfaces. Unfortunately, view dependent isosurface generation algorithms tend to be relatively slow in extracting triangles as they have to deal with two different types of constraints. The first involves a range search relative to the given isovalue, while the second constraint amounts to a spatial filtering to identify the visible portion of the isosurface. In order to

[2] http://www.llnl.gov/CASC/asciturb/
[3] http://www.nlm.nih.gov/research/visible/

achieve interactive rendering, researchers have resorted to parallel algorithms such as those that appeared in [4,24]. A recent method based on an elaborate data structure for a persistent octree was described in [17]. We should note that the quality of the generated isosurfaces using triangular meshes may be poor especially for complex regions as the triangular mesh is just a polygonal approximation of the surface.

In this paper, we consider the alternative approach of generating isosurfaces by using ray tracing. Such an approach was first proposed in [14], using a brute-force ray tracing on the SGI Reality Monster, which is a shared memory multiprocessor. A distributed memory version was described in [2], and schemes based on *kd*-trees and octree are described in [3,19,6]. In the next section, we will provide an overview of the isosurface ray tracing algorithms since they are directly related to the algorithm described in this paper, and then proceed to describe our algorithm.

2 Previous Work

Isosurface ray tracing directly computes the isosurface by shooting rays from the viewpoint through the pixels and computing the intersections of these rays with the isosurface. This method, coupled with various techniques for improving visual effects such as shading, reflection, and global illumination, can generate extremely high quality visualization of isosurfaces. However, the method is computationally demanding, especially for high resolution screens, since it is pixel-by-pixel approach and hence its complexity depends on the number of rays and the size of the dataset as well as on the scheme used for ray traversal and for computing and shading the intersection voxels of the rays with the isosurface. Note that in general the memory access is relatively expensive as the voxels are not processed in the same order as the data layout, and cache performance can be poor since the cells intersected with cast rays are not easily predictable.

Given the high computational requirements of isosurface ray tracing, Parker et. al. [14] describe an implementation on the SGI Reality Monster, which is a shared-memory multiprocessor with up to 128 processors, and show interactive isosurface rendering of the $1GB$ Visible Woman dataset. The screen size used is 512^2. Their algorithm uses a simple multi-level spatial hierarchy with a $3D$ tiling of the input data to improve cache performance. More recently, Wald et. al. [19] describe an implementation using a combination of a *kd*-tree and coherent ray tracing that exploits the SIMD extensions that are available on many of the current multi-core processors. A *kd*-tree is a binary tree that represents a spatial partitioning of the volume data. Each node, except for the leaves, represents a splitting plane that is closest to the center of the largest

dimension. Each node contains the minimum and the maximum of the densities contained within the subtree. This structure is very similar to the octree as described in [21,6], except that the authors claim that the *kd*-tree more easily enables a simple handling of packets of rays used in coherent ray tracing. Coherent ray tracing traverses packet of rays through the *kd*-tree in order to make effective use of the SIMD extensions. However this comes at the cost of creating an indexing structure that is at least twice as large as the input data size. For example, their most compact *kd*-tree representation of the $8GB$ LLNL dataset is of size $18GB$. The authors illustrate the performance of their scheme on a single and on a 5-node cluster of dual-1.8 GHz AMD Opteron, with a default screen size of 512^2. A more compact *kd*-tree is introduced in [3] and used for isosurface ray tracing on two-processor platform, each is a dual-core 2.6 GHz AMD Opteron with $16GB$ RAM. They show interactive rate isosurface rendering for a variety of datasets of sizes up to $8GB$ but using half the memory used in [19]. The multi-layer ray tracing method using frustum traversal proposed in [15] utilizes spatial coherence in image space to speedup the rendering of geometric objects of a few million triangles (basically involving coherent scenes) but it is unclear whether the method will perform well on complex (and possibly incoherent) isosurfaces in large scale datasets such as LLNL. In fact, it has been noticed that the packet ray traversal technique (including the exploitation of SIMD instructions) may perform poorly on incoherent complex scenarios where frequent ray splitting and merging could lead to a worse performance than just using a single-ray [6,7,3]. As the most recent implementation of the single-ray scheme, an octree representation that contains the scalar data as well as the range information is used in [6] to generate competitive performance on multi-core processors. Its multi-resolution level of detail (LOD) version that incorporates coherent ray tracing to work around the problem of incoherent scenes appears in [7], resulting in faster rendering of LOD data in some cases.

Additional work that is somewhat related to our work but addressing the general direct volume rendering problem appears in [18,20]. To improve the efficiency of ray tracing for direct volume rendering, a two-step method is proposed in [18]. The first step consists of projecting the boundary cells onto the image plane using graphics hardware, and the second step applies the standard ray tracing but now slightly more constrained. In [20], $3D$-textures are used to estimate the start of ray traversal. Some other schemes use quantized voxels to accelerate volume rendering [11], or object-order projection to speed up ray casting for parallel view port volume rendering [12], as well as cache-efficient layouts of bounding volume hierarchies [23] to improve *kd*-tree access during ray casting for some medium size of datasets. The direct volume rendering and ray casting implemented on GPUs also appeared in the past few years with the advent of improved GPU programmability. Some recent related advances are reported in [5]. However, the GPU-based approach is constrained by its on-board memory size (usually $512MB$) and its fairly strict SIMD programming

model.

3 A Novel Hybrid Strategy

In general, there are several main features that have been exploited in the literature to speed up the computation of isosurfaces through ray tracing. These are:

- The use of spatial decomposition indexing structures augmented by the range of the densities at each node. Such a structure enables the culling away of large parts of the data, which are not part of the visible portion of the isosurface. Also, the input data can be incorporated into the structure to generate a multiresolution representation of the volumetric data.
- The mapping of the inherent parallelism of ray tracing into pipelined and parallel architectures since the ray traversal corresponding to any pixel can be performed independently of any other ray traversal.
- An attempt to optimize cache performance by processing chunks of the input data of suitable sizes.
- The exploitation of the SIMD extensions on some of the newer multi-core processors, which led to the idea of shooting a packet of rays (typically, 4 rays corresponding to 2×2 adjacent pixels) as the unit traversal through the volumetric dataset.

Except for applying the above techniques in different ways, all the known isosurface ray tracing algorithms follow more or less the same basic strategy. We restrict ourselves here to primary ray tracing as it delivers most effectively the visual information of isosurfaces and is the most time-consuming component of ray tracing. In this paper, we introduce a new strategy to greatly improve primary ray tracing to generate isosurfaces. Our method is a combination of object-order projection of a coarse version of the data and a very efficient ray tracing restricted to a few data blocks for each packet of rays (corresponding to adjacent pixels). For clarity, we start by presenting the single ray version, which will be extended to packets of rays in the next section. At a high level, our scheme consists of the following two phases.

Phase I: We perform a traversal of a $3D$-tiled version of our volumetric dataset, using a very compact data structure, to identify, through projection from object space onto the image space, the visible and isosurface-intersecting $3D$-tiles corresponding to each pixel. From now on, we refer to a $3D$-tile of the input data as a **data block** or simply a **block**. At the end of this phase, we will have, for each pixel, a list of data blocks that are visible from the ray through this pixel and that intersect the isosurface, organized in a front to back order relative to the viewpoint.

Phase II: We now shoot a ray from each pixel through its ordered list of data blocks constructed during Phase I (assuming the list is non-empty; otherwise there is no work to be done), checking whether an intersection voxel with the isosurface lies within a block from the list . There is no need to proceed further once an intersection voxel is found. This will be followed with a trilinear interpolation and shading of the corresponding voxel.

We now proceed to provide more details about each phase and to explain how such a strategy can be optimized on multi-core processor architectures and how it can make effective use of their memory hierarchies.

We organize our volumetric data into a coarse grid of equal-sized blocks, where the scalar field values within each block are stored contiguously in a pre-defined order and the block is identified by the coordinates of a pre-specified corner. We use an octree to index the data within a block such that the leaves correspond to $2 \times 2 \times 2$ cells. That is, each leaf will contain a pointer to such a cell. As usual, each node of the octree will contain the minimum and maximum of the values of the voxels lying within the region represented by the node. In addition, we build a BONO (Branch-On-Need Octree) [21] tree for the coarse grid, augmented as usual by the appropriate value ranges. The BONO structure is very similar to the octree except that, for data resolutions other than powers of two, BONO avoids allocating nodes of empty subtrees, and hence it is more space-efficient than the original octree. Note that the blocks are always chosen so that each dimension is a power of two, and hence the use of octrees to index their scalar data.

Phase I is implemented as follows. For efficiency reasons, we limit the size of the list of blocks associated with each pixel to a fixed constant k. We later show that $k \leq 20$ seems to give the most efficient implementation. We note that we will always obtain the correct visible isosurface regardless of the value of k. The BONO tree representing the coarse grid is traversed starting from the root. Assume we reach a node v of the tree. If the range stored in v contains the isovalue, we project the minimum axis-aligned bounding box (AABB) of v onto the screen. Such a $3D$ AABB is computed by using the coordinates of a pre-specified corner, whose x, y, and z extensions can be deduced from the level of v. We consider all the pixels falling within the projected area. If the size of the list of any such pixel is less than k, we traverse the children of v in a front to back order relative to the view point. Otherwise, we skip the subtree rooted at v. Once a leaf is reached, the list of each pixel falling within the projection of the minimum bounding box of the corresponding block is augmented with a pointer to this block unless the list already has k blocks. Notice that at the end of this phase, we have a list of size at most k blocks associated with each pixel, and organized in a front to back order since this is how the BONO tree was traversed. The limit imposed by the value k makes this phase quite efficient.

Phase II is implemented as follows. For each pixel with non-empty list , we shoot a ray from this pixel through the list of its blocks, one block at a time in the order they appear on the list. If the list is empty, the ray does not intersect the isosurface. Otherwise, if the ray intersects a voxel on the isosurface for a block on the list, we perform a trilinear interpolation using the unit cell containing the voxel followed by (diffuse) shading. The ray voxel intersection is computed using the method described in [10] while the normals are computed using forward difference. If the ray reaches the end of the list without finding such a voxel (and hence the list is of size k), we revert to the traditional approach by resuming the traversal of the BONO tree from where we stopped during the first phase, which is indicated at the end of each k-sized list. Lastly, if the list is of size less than k with no intersection found at this stage, which implies that this ray does not intersect the isosurface. Clearly we will always end up with the correct intersection points of all the rays with the isosurface regardless of the value of k. However, we will later show that the case when we have to resume the traversal of the BONO tree (as in the traditional approach) occurs rarely if k is chosen appropriately. Among the advantages of our scheme are:

- The traversal of the BONO tree of the coarse version of the volumetric data can be performed extremely fast since its size is very compact and each projection enables us to increase the sizes of the lists of many nearby pixels simultaneously. Also, the upper bound imposed by the value of k restricts the traversal significantly.
- Almost all the pixels with rays not intersecting the isosurfaces will be identified through the first phase of our algorithm, and we only shoot very few non-intersecting rays during the second phase. We will later show that the percentage of the non-intersecting rays cast is extremely small.
- The traversal of a ray is now conducted through visible blocks in a front to back order, and hence we are skipping in general a substantial fraction of irrelevant portions of the volumetric data up front.
- Nearby pixels will likely have a number of common blocks on their lists and hence we can use spatial locality of pixels to achieve high performance caching. That is, processing nearby pixels can make effective use of caching since their corresponding lists are short and are likely to share blocks. We will show how to exploit this feature to significantly improve performance.

We next consider a couple of optimization techniques to this basic scheme.

4 Improvements on Basic Scheme

4.1 Extension to Packets of Rays

Our scheme builds for each pixel a small size list of data blocks that are visible from the ray through this pixel and that intersect the isosurface. In general, we expect the lists of adjacent pixels to significantly overlap, especially for close views. We exploit this feature by combining the lists of each group of adjacent pixels (say 2×2 as used in our experimental results) into a single list. This is somewhat similar in spirit to the use of packet of rays in [19,7]. However in this work we do not make use of SIMD instructions to process the packet of rays for ray casting since its success depends upon scene coherency and the use of such instructions may lead to poorer rendering performance on complex incoherent scenarios as pointed out in [6,7]. Instead, our emphasis here is on high level algorithmic techniques that are applicable to all scenes. However, we intend to explore in the future the additional benefits of our algorithm when SIMD instructions are exploited to process each grouped list of data blocks for coherent scenes.

View	List upper bound k		Time (msec)		
Type	1024^2 lists	512^2 lists	1024^2 lists	512^2 lists	Ratio
Far	3	6	273	110	2.48
	7	14	362	142	2.55
	11	22	434	169	2.57
Close	3	6	228	68	3.35
	7	14	334	92	3.63
	11	22	449	119	3.77

Table 1. List generation time on single-core for Far and Close views with various upper bound k using single pixels (resulting in 1024×1024 lists) and groups of 2×2 of adjacent pixels (resulting in 512×512 lists) on a 1024^2 screen. To make the comparison fair, the list upper bound k is adjusted so that two cases generate relatively the same number of shaded pixels after rays are cast through the data blocks on the lists.

During the Phase I creation of the lists, we traverse the BONO tree as before. However we create lists for each packet of rays (corresponding to an adjacent group of pixels typically 2×2) rather than a separate list for each pixel. Whenever such a group of pixels overlaps with the projection of the current node being traversed, the group's list is processed as before. Since we are now creating fewer lists, the performance of Phase I improves substantially as illustrated in Table 1, which shows the execution times corresponding to

different values of k to generate respectively 1024^2 lists (one list per pixel) and 512^2 lists (one list for each 2×2 adjacent pixels).

During Phase II, a slight overhead will be incurred as the upper bound k on the size of the list needs to be increased for the grouped list. However we will show later that we achieve the best performance when k is around 20, compared to 12 in the single pixel case, and therefore the overhead will be minimal.

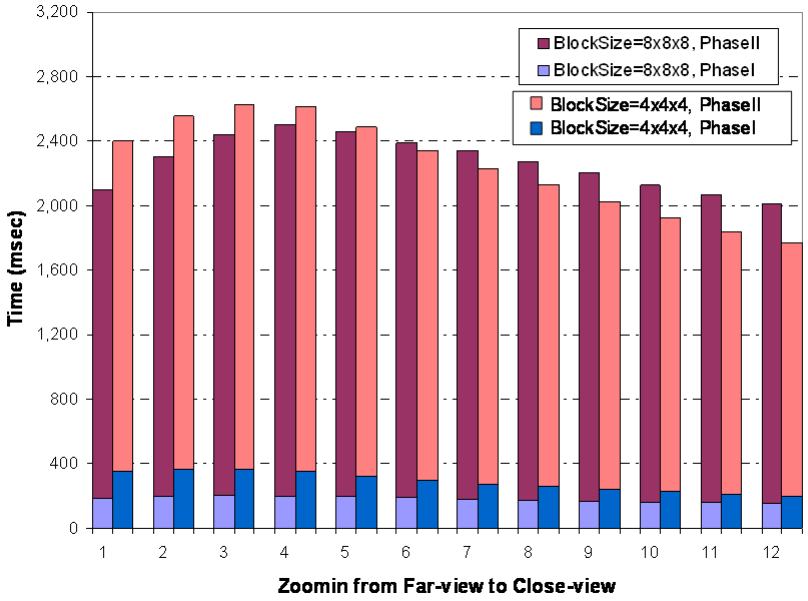

Fig. 1. Execution time of List generation(PhaseI) and ray casting(PhaseII) on single-core as we vary the zoom-in from far-view to close-view for block sizes 8^3 and 4^3.

4.2 Adaptive Block Size

Another improvement to our basic scheme is to make the size of the data block adaptive. For far views, we can use relatively large size blocks especially when processing large volumetric data. For example, we use $8 \times 8 \times 8$ blocks for the LLNL dataset to handle the rendering of far views, which results in 6×6 pixels on average being covered by the projection of a data block, and this seems to achieve the best performance when generating the lists for groups of 2×2 adjacent pixels. However when we zoom in for close views, the smaller size blocks are more effective especially that the number of BONO nodes visited and the number of projected blocks are much smaller but the projection of a block covers more pixels (e.g. 12×12 on average when we use $4 \times 4 \times 4$ blocks for a $16 : 1$ zoom-in close view). Figure 1 illustrates the performance of each of Phase I and Phase II on the LLNL dataset as a function of the block size and the viewpoint.

We make our scheme adaptive as follows. We visit the BONO nodes as before, except that, for close views, at the end of the traversal of the BONO tree, we proceed with the octree traversal of the BONO tree leaves until we reach the desired block size. After reaching the desired size, we proceed using the current blocks to compute the minimum bounding box and construct the lists for the various packets of rays.

5 Multithreaded Implementation

As the multi-core processors begin to dominate the computing market, new programming paradigms are needed to fully exploit the performance opportunities offered by these processors. In general, parallel programming remains a difficult task in spite of the considerable related research efforts undertaken during the past several decades. Unfortunately, this task becomes even more difficult for multi-core processors given the limited on-chip memory, and the typical complex memory hierarchies present in such architectures. Moreover, there are currently no widely adopted mechanisms for handling communication or memory accesses on such processors. Compare this for example with distributed memory multiprocessors for which the message passing MPI communication libraries have been quite successful in supporting many applications. On the other hand, multi-core processors present an opportunity for speeding up the computation by partitioning the load among the cores, but a careful management of the memory hierarchy (including whatever caches are available) is critical to the overall performance, in addition to the usual problem of trying to ensure balanced loads among the cores with as little communication as possible. In this paper, we will focus on programming a single multi-core processor rather than a cluster of these processors since we believe this is where the main challenge is, and moreover a multi-core processor will soon be the common platform for most people. Programming clusters of such processors will probably be a relatively easy extension of that of the single multi-core processor since we can make use of the many cluster programming techniques that have been developed over the past twenty years or so. We use the Clovertown platform, consisting of two Quad-Core Intel 1.86 GHz Xeon Processors 5320. Each dual-core on a Quad-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 experiments.

In general, assume we have p cores on a multi-core processor, with some local (possibly shared) cache or memory available for each core. Using p threads, our scheme is implemented as follows.

Step1. To handle Phase I, the screen is divided into almost equal contiguous

regions, with each thread responsible for creating the lists of $3D$ data blocks corresponding to the pixels in its region. Each thread traverses the BONO tree and creates the lists of blocks corresponding to its groups of pixels. Hence a traversal of a node is followed by traversing the children nodes in a front to back order only if the projection of the minimum bounding box of the node intersects with the thread's screen region and there is at least one list associated with the region which is not full (i.e., its size is less than k). Note that our BONO tree is small and only a fraction (no more than 10%) of the total BONO tree nodes are actually accessed during Phase I due to the imposed list upper bound k.

Step2. To handle Phase II, we start by partitioning the ray casting tasks through all the lists as follows.

2.1 Partition the screen into small image-size tiles (for example 8×8 or 16×16) and order these tiles using a Z-order (or a space-filling curve such Hilbert space filling curve). Such ordering will ensure a high degree of spatial locality of nearby tiles and will result in high cache performance as we will show later. This step is performed during the preprocessing stage and takes a few milliseconds.

2.2 After the lists are generated in **Step1**, assign a weight to each small-size image tile, which is equal to the number of non-empty lists within the tile, and compute the total weight W of all the tiles.

2.3 Following the Z-order of the image tiles, group the tiles as follows. The first set of tiles whose total weight is $\frac{W}{2}$ are grouped into p equal groups, each group consisting of a contiguous set of tiles following the Z-order. A group is identified by a pair of indices indicating the first and the last image tile in the group. The second set of remaining tiles whose total weight is $\frac{W}{4}$ is grouped equally as before into p groups. This process is repeated until each image tile is associated with a group, and hence we need at most logarithmic number of iterations in screen size, each iteration creating p groups. The result is a list L_I of pairs of indices, each pair delineating a group of image tiles.

Step3. We perform ray casting dynamically as follows. Initially, each thread will grab a group of image tiles from the ordered list L_I created in **Step2**. A thread will then process its group by shooting rays through the pixels in the group using the data block lists generated in **Step1**. Once a thread completes the processing of its group, it grabs the first available group of tiles from the list L_I, and start processing the corresponding group. The process continues until all the image tiles are processed.

Our dynamic allocation of the ray casting tasks attempts to achieve an optimal trade-off between two conflicting requirements. The first is the desire to have fine-grain tasks to be assigned dynamically with the goal of achieving

tight load balancing. The second requirement is to make the number of jobs as small as possible with the goal of minimizing the amount of coordination and synchronization among the threads. In our list L_I, we start with jobs (corresponding to groups of image tiles) that are relatively large, and decrease the sizes until we reach fine-grain jobs at the last p positions of L_I. Therefore our strategy seems to strike an optimal balance between the two requirements. In the next section, we will illustrate the performance of each step, and show in particular that we are able to achieve a very tight load balancing among the different threads as well as very high cache performance.

6 Experimental Results

We have conducted extensive testing of our algorithm on six datasets whose sizes range from about $100MB$ to $8GB$, which is the largest dataset that can fit into the main memory of our Clovertown platform. Although the isosurface can be generated from an arbitrary viewing point, we report our test results for two typical views: Far-view that enables the viewing of the complete isosurface on the screen; and Close-view that consists of a zooming by a ratio of $16:1$ to view details of regions of interest. These two view settings will typically involve significantly different numbers of voxels intersecting the isosurface, which directly influence the performance of any isosurface rendering algorithm. Hence, we measure the corresponding performance separately to shed more light into the robustness of our scheme. In addition, we take six different viewing angles for both Far-view and Close-view, specified by zenith angle $\phi = \{15^o, 45^o, 75^o\}$ and azimuth angle $\theta = \{22.5^o, 45^o\}$ in spherical coordinates. Due to the high topological complexity of most generated isosurfaces, the screen resolution for our testing is typically set at 1024^2, which for example enables the highlighting of the fine details of the complex LLNL dataset. As described before, our scheme consists of an initial phase that generates a list of data blocks for each packet of rays, followed by a dynamic allocation of groups of Z-ordered image tiles among the processor cores, and ending with ray casting through the lists associated with groups of adjacent pixels. If the ray intersects the isosurface, the intersection position is calculated by solving a trilinear interpolation equation as in [10], then the pixel is shaded by computing the forward difference gradient as the normal at the intersection position and applying the diffuse shading model. We measure the execution time of each phase as well as the overall rendering frame rate of the corresponding isosurfaces. We will show scalability both in data output size and number of cores used. In particular, we run our tests on $1, 2, 4, 8$ CPU cores of our Clovertown platform and measure the performance for each case separately.

At this point, we note that comparing our experimental results with those of previous algorithms is not straightforward (except when comparing the

sizes of the indexing structures) since prior work did not provide sufficient details about their testing scenarios and they used different processors (which sometimes were faster in CPU clock speed than our 1.86 GHz Quad-core processors and had more main memory). However we will see later that our performance numbers suggest significantly better performance than any of the published algorithms. To illustrate the relative increased performance achieved by our techniques in a concrete way, we implemented a standard ray casting algorithm using the octree indexing structure, while trying to make as effective use of the memory hierarchy and multithreading as much as possible. All the detailed steps for ray traversal, computing the intersection points, and shading are the same as in our algorithm. In particular, our multi-threaded implementation of the standard algorithm is based on a dynamic allocation of static small screen tiles (16×16 pixels) to the different processor cores. Therefore the comparison between the two algorithms running on the same machine with identical datasets, viewpoints, and screen sizes will highlight the differences in the strategies used by both algorithms rather than the small implementation details. Moreover, it appears that the performance of this standard octree algorithm is rather very similar to that achieved by the octree algorithm reported in [6].

The rest of this section is organized as follows. We first present the attributes of our indexing structures for the datasets used, illustrating their substantially smaller sizes than the *kd*-trees used in previous work. We then demonstrate the critical importance of the size limit on the lists of data blocks by focusing on the rendering of the complex LLNL isosurfaces on high resolution screens. We end with an illustration of the overall performance of our algorithm for all the six datasets, and demonstrate adaptability to the complexity of the rendered scenes, high cache performance, and scalability in number of cores.

6.1 Datasets Used

We selected six datasets for our tests, which can generate spatially sparse or dense, topologically smooth or complex isosurfaces, and which represent most types of isosurfaces encountered in various applications (Figure 2). The sizes of these datasets vary from $87MB$ to $8GB$, which is the largest that can fit into our main memory. These datasets illustrate our scheme's adaptivity to various types of isosurfaces and data sizes. Table 2 illustrates the block size used for each dataset and the corresponding number of blocks for each case. In all cases, the number of blocks is relatively small and does not exceed a few millions, and hence the corresponding BONO tree is very compact and can be constructed extremely quickly. In fact, our largest BONO tree is around $46MB$ for the LLNL $8GB$ dataset. On the other hand, the accumulated size of the finer indexing structures (that is, octrees) for all the data blocks is just

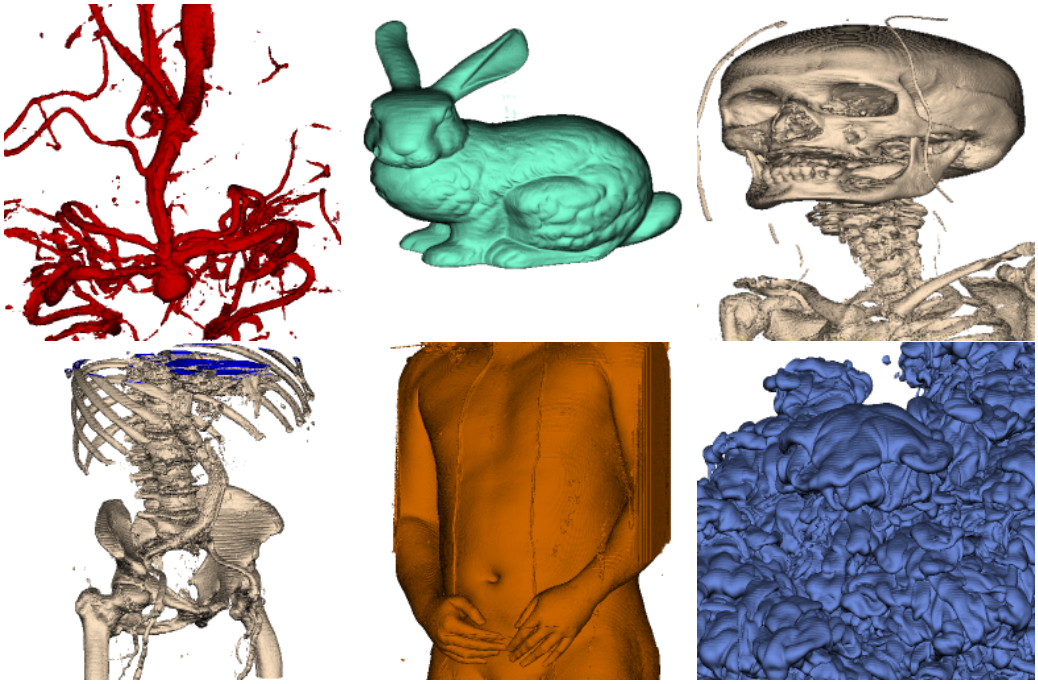

Fig. 2. The six datasets used in our experiments. From left to right, top to down, the datasets are: Aneurism, Bunny, Skull, Abdominal, VisMale and LLNL respectively.

a fraction of original dataset, no more than $\frac{1}{4}$ as shown in Table 3. Note that the total sizes of our indexing structures are substantially smaller than those used by the *kd*-tree algorithms (such as [19,3]). The preprocessing times are also much better than any of the published preprocessing times even when the previous algorithms are run on faster clocked processors.

Data Sets	Field Size	Grid Size	Data Size	Block Size	# of Blocks
Abdominal	2 bytes	$512^2 \times 174$	87 MB	4^3	302 K
Bunny	2 bytes	$512^2 \times 360$	180 MB	4^3	1,181 K
Aneurism	2 bytes	$512^2 \times 512$	256 MB	4^3	1,620 K
Skull	2 bytes	$512^2 \times 512$	256 MB	4^3	1,680 K
VisMale	2 bytes	$512^2 \times 1882$	941 MB	8^3	663 K
LLNL	1 bytes	$2048^2 \times 1920$	7.5 GB	8^3	5,655 K

Table 2. Parameters of various datasets used

Data	Indexing Size (MB)		Preprocess Time (Sec)		Space
	Blocks	BONO	Blocks	BONO	Overhead
Abdominal	11.52	2.97	7.1	0.172	13.24 %
Bunny	45.07	11.43	15.2	0.332	25.04 %
Aneurism	61.80	15.68	23.4	0.407	24.14 %
Skull	64.12	16.26	25.6	0.427	25.05 %
VisMale	187.37	6.43	88.3	0.221	19.91 %
LLNL	809.08	46.79	520.0	1.950	10.53 %

Table 3. Size of our indexing structure for Blocks and BONO tree along with their preprocessing time

6.2 Performance Implication of the Upper Bound on the Lists of Tiles

In addition to our new strategy that combines object order traversal followed by ray tracing, we make use of a novel trick by putting a limit k on the number of blocks computed for each group of adjacent pixels (corresponding to a packet of rays). We examine here the critical importance of such an upper bound. The total execution time of our algorithm consists of four main components: (i) the time it takes to traverse the BONO tree and to generate the lists of blocks; (ii) the time it takes to group the small-sized image tiles into groups for dynamic allocation among the processor cores; (iii) the time to perform ray casting through the data block lists; and (iv) the time needed for ray casting of the unfinished pixels (that is, those pixels whose lists were of size k with no intersecting voxels found in step (iii)). The amount of work involved in grouping the image tiles is small (in the order of $2 \sim 3$ milliseconds). The bulk of the time is spent on steps (i), (iii), and (iv). In order to illustrate the trade-off involved relative to the upper bound k and the various stages of the algorithm, we ran a number of experiments on the LLNL dataset of time step 250 using the isovalue of 70 and screen resolution 1024^2 for Far-view settings on our Clovertown platform. We measured the execution time on a single core for different values of k, ranging from 0 to 42 (note that standard ray casting is the same as the case when $k = 0$). The corresponding results are illustrated in Figures 3.

From these results, we can make the following observations. First, the standard ray casting corresponding to the case when $k = 0$ has the longest execution time by a factor of approximately 40% relative to our algorithm for the best value of k. Second, the time it takes to generate the block lists (indicated in blue) increases with the value k almost linearly because the depth complexity of LLNL data is high (~ 50) but its contribution to the total time is less than 10% for $k \leq 30$. Third, the ray casting on the block lists (indicated in red)

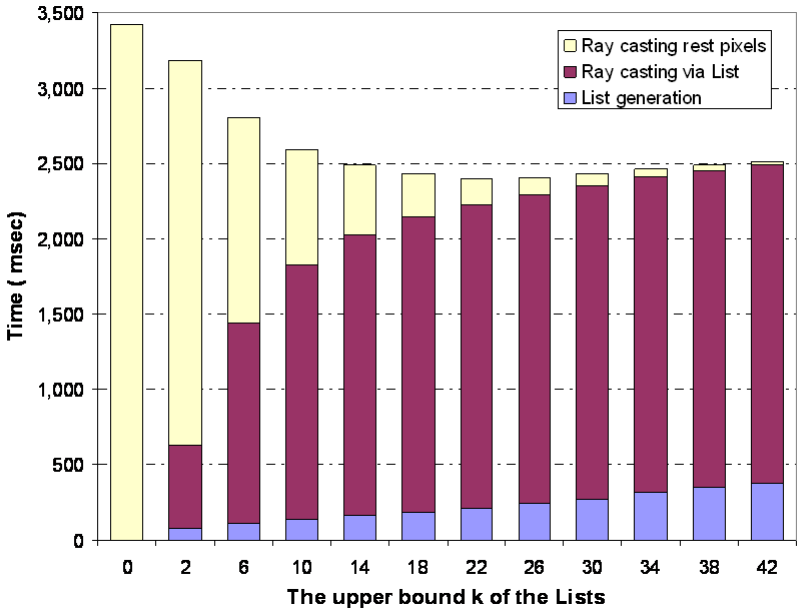

Fig. 3. Execution times of the different stages of our algorithm on a single core vs. the value of upper bound k. The results are for the Far-view of the LLNL dataset of time step 250 using 1024^2 screen resolution.

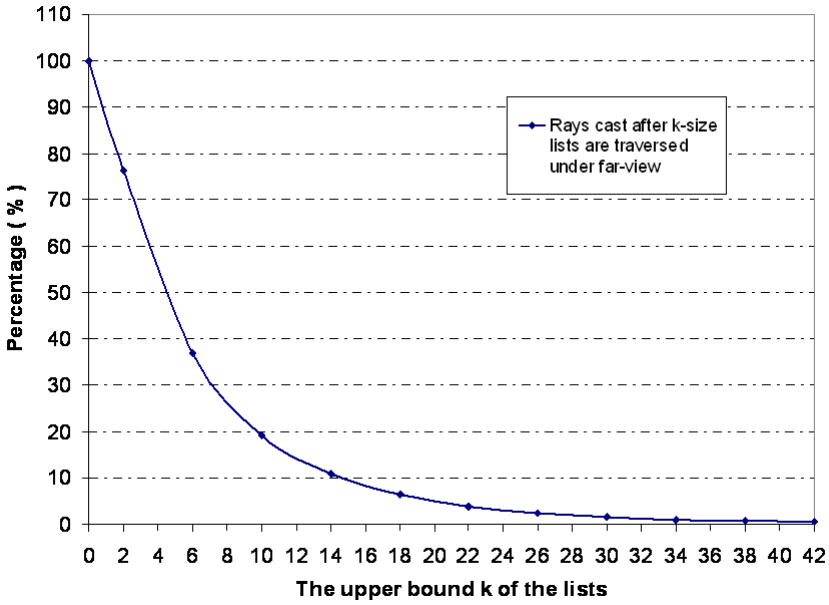

Fig. 4. Percentage of pixels left to shade after going through k blocks from the lists for the $2048^2 \times 1920$ LLNL dataset at time step 250. Screen size is 1024^2.

takes an increasingly larger fraction of the total execution time as k increases, and is significantly larger than the time it takes to generate the lists. Fourth, and perhaps most importantly, the number of rays that have no intersection with the isosurface after going through exactly k blocks (indicated in yellow) drops very quickly initially as k increases and then somewhat levels off, which

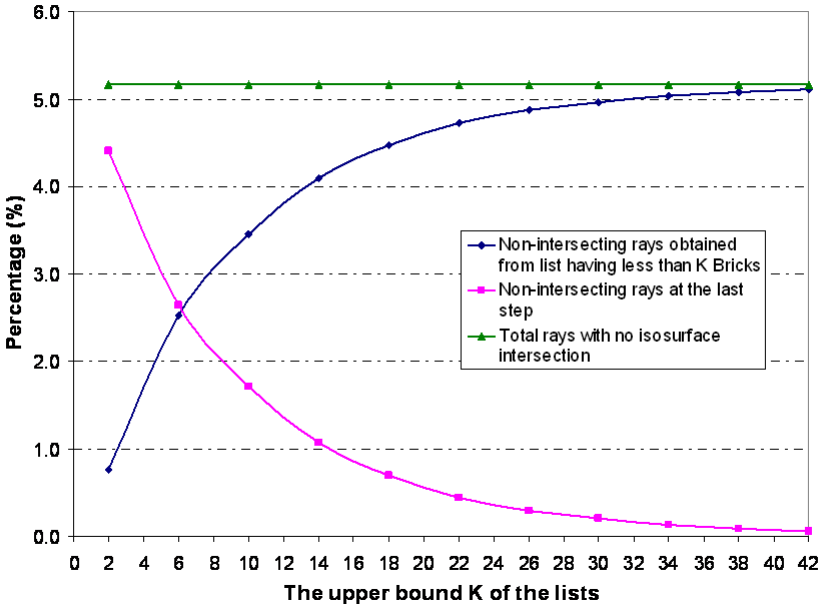

Fig. 5. Analysis of the percentage of rays having no intersection with the isosurface in our scheme using the $2048^2 \times 1920$ LLNL dataset at time step 250 under Far-view. Screen size is 1024^2.

can be verified more clearly by examining the curve shown in Figure 4. The combined effect of these properties lead to an optimal value for k that in our experiments has been in the range $16 \sim 22$. For example, the optimal value of k is around $20 \sim 22$ for the LLNL dataset, while for the VisMale dataset, the optimal value of k is around $16 \sim 18$ because of a smaller complexity depth.

Another important benefit of our scheme is the significant decrease in the number of rays cast which do not intersect the isosurface relative to the standard strategy. The traversal of the BONO tree effectively identifies the area on the screen where the isosurface is mapped, passing this information for ray casting through the block lists. Figure 5 illustrates the dependence of the non-intersecting rays cast upon the value of k. The blue curve represents the number of non-intersecting rays determined when going through the lists containing less than k blocks, while the red curve represents the number of non-intersecting rays determined at the very last step of the algorithm after their k-size lists were completed. Obviously, the total number of non-intersecting rays cast is the sum of these two numbers, and does not depend upon the value $k > 0$. The percentage is out of the total number of rays cast by our algorithm. As shown in Figure 5, the value of k directly impacts these two numbers, while the total number of non-intersecting rays cast by our algorithm (for $k > 0$) is about 5.2% of the total number of rays cast. On the other hand, the standard ray casting doesn't filter out any ray initially and simply shoots a ray through each pixel. When the isosurface doesn't occupy most of the screen, which is not uncommon in Far-view, the percentage of non-intersecting rays over total number of rays cast could be large (such as in Aneurism and Ab-

dominal datasets). For the same LLNL dataset and the same screen resolution, standard ray tracing ends up with around 45% non-intersecting rays on average over the six tested viewpoints under Far-view. This clearly illustrates the power of our hybrid strategy that manages to almost eliminate the casting of non-intersecting rays.

We will assume for the rest of this paper that an optimal value of k has been selected and report the performance corresponding to this value.

6.3 Overall Performance

In this section, we give an overview on the overall performance of our algorithm on different datasets using a range of viewpoints. The tests conducted are for both the Far-view and the Close-view, each from six viewing angles specified by (ϕ , θ), using a 1024^2 screen resolution. A variable number of cores, up to 8, are used by running the multi-threaded version of our algorithm. While the scalability of our algorithm and a detailed analysis of the load balance achieved are described in Section 6.6, we report here on the overall performance and compare it with the best published results. The performance, expressed in terms of fps to render the LLNL dataset (time step 250 and the isovalue is equal to 70), is listed in Table 4 for the Far-view and the Close-view at the six different viewing angles. As can be seen, we achieve interactive rates regardless of the viewpoint or the viewing angle for a very complex isosurface on a high resolution screen. These results illustrate the robustness of our scheme regardless of the complexity of the scene. Note that the number of cores is supposed to steadily increase in the future (perhaps doubling every 18 $\sim$ 24 months), and hence our scheme will easily achieve interactive rates on future desktop or laptop processors.

1024^2 Screen		$\phi - \theta$					
View	Core	15-22	15-45	45-22	45-45	75-22	75-45
Far	2-core	0.87	0.87	0.84	0.81	1.36	1.35
	8-core	3.41	3.44	3.28	3.20	5.29	5.24
Close	2-core	1.32	1.27	1.08	1.00	1.04	0.98
	8-core	5.08	4.85	4.15	3.85	4.12	3.83

Table 4. Performance of our algorithm on the Clovertown in fps for the LLNL dataset with screen resolution 1024^2 and isovalue 70 under Far and Close views.

As already noted in previous research [14,19], the ray traversal across the spatial acceleration structure, such as kd-trees or octrees, constitutes the major portion of the total execution time (usually around 65% $\sim$ 70%) in standard

ray casting. Yet, Phase I of our algorithm uses efficient object-order projection of blocks to considerably reduce the number of ray traversal steps in Phase II, which in large part leads to our superior performance. In Table 5 the comparison of number of ray traversal steps in our algorithm and standard ray casting for various datasets clearly elucidates this aspect.

1024^2 Screen Dataset	standard ray casting ($\times 10^3$)	ours ($\times 10^3$)	ratio
Abdominal	39,992	5,865	6.82
Bunny	22,547	2,585	8.72
Aneurism	45,237	3,291	13.8
Skull	37,648	5,463	6.89
VisMale	21,421	3,174	6.75
LLNL (far)	42,228	12,944	3.26
LLNL (close)	42,868	6,129	6.99

Table 5. Number of ray traversal steps undertaken during ray casting in standard ray casting and our algorithm for a screen size of 1024^2 screen using all the datasets considered in this paper.

Screen size	512^2			1024^2		
Dataset	standard	ours	ratio	standard	ours	ratio
Abdominal	12.99	24.65	1.90	3.80	7.87	2.07
Bunny	22.22	38.56	1.74	6.49	12.66	1.95
Aneurism	13.89	39.33	2.83	3.77	12.35	3.27
Skull	13.70	25.02	1.83	3.76	7.19	1.91
visMale	18.52	29.68	1.60	5.52	9.26	1.68

Table 6. Measured performance on 8-core Clovertown in fps for our scheme and the standard octree ray tracing algorithm under Far-view setting for various datasets

We now report a summary of our performance results on the other datasets illustrated in Fig. 2. These results, expressed in terms of fps under the Far-view setting and taking the average over the different viewing angles, are shown in Table 6. Since these datasets have lower depth complexity than the LLNL dataset, combined with the fact that their isosurfaces cover the screen unevenly, our algorithm delivers a faster interactive rendering rate and achieves further performance improvements over the standard ray casting algorithm. Note also the significant performance achieved for the lower resolution screen of size 512^2.

Finally, we compare our algorithm to the algorithm reported in [6], which uses the 16-core NUMA 2.4 GHz Opteron workstation. As far as the authors know, the performance numbers published in [6] are the best known for the general isosurface ray tracing problem. Since our platform is different than theirs, we need to calibrate the two processors. Comparing the SPEC benchmark [4] performance on the AMD Opteron 2.6 GHz and the Intel Xeon 1.86 GHz with the same number of cores (8 in each case) as shown in Table 7, we note that the Opteron runs slightly faster and has significantly better throughput than the Intel Xeon. Listed in Table 8 are the performance numbers reported in [6] on their 16-core NUMA and the performance numbers of our algorithm on the Clovertown 8-core using the same dataset, the same viewpoint, and the same screen size. While the number of cores on their platform is twice the number of cores on our platform and they have access to $64GB$ of memory compared to $8GB$ on our platform, our performance is significantly better for close views and only slightly worse for far views. As we show later, our algorithm is highly scalable and hence we expect our performance to almost double on a 16-core Clovertown, and hence the resulting performance will be significantly better than that of the algorithm in [6].

		AMD	Intel
CPU Model		Opteron 8218	Xeon E5320
CPU Clock		2.6 GHz	1.86 GHz
Multi-Core		4 processors	2 processors
		2-core per die	4-core per die
L1 Cache per core		64 KB I + 64 KB D	32 KB I + 32 KB D
L2 Cache per die		2 MB I+D	4 MB I+D
Main memory		32 GB	16 GB
Speed	Cint	11.3	11.1
	Cfp	11.9	9.57
Through-put	Cint_rate	85.3	58.5
	Cfp_rate	83.2	41.3

Table 7. Performance comparison between an 8-core Opteron 8218 and an 8-core Xeon E5320 using the SPEC benchmark.

[4] http://www.spec.org/benchmarks.html

Screen 1024^2		NUMA 16-Core	Clovertown 8-Core
View	Time step	Knoll et. al.	ours
Far	50	7.4	5.88
	150	5.7	4.90
	270	4.7	4.15
Close	50	4.3	7.04
	150	3.6	5.81
	270	3.5	5.58

Table 8. Performance Comparison in fps for LLNL datasets on 1024^2 screen resolution. NUMA is Knoll's platform consisting of AMD 2.4GHz 16-core Opteron with 64GB memory, Clovertown is our platform consisting of Intel 1.86GHz 8-Core with 8GB memory. The LLNL datasets and testing views correspond to their settings with isovalue = 20.

6.4 Adaptivity Upon Data Complexity

An overall critical issue regarding the performance of isosurface rendering algorithms is the way they depend on the input data size. The original MC algorithm had to traverse all the unit cells of the volumetric data and hence it was soon discovered that the algorithm is too slow for large datasets. Efforts were then directed toward reducing the rendering algorithm execution time so that it primarily depends on the size of the intermediate triangular mesh generated by the MC strategy rather than the whole input data. Several such variations of the MC algorithm already exist [1,16]. However the triangular mesh approximation is typically much larger than what is needed to render the isosurface from a particular viewpoint, and hence come up the efforts for efficient view-dependent algorithms. We argue that our algorithm in general adapts extremely well to the size of the *visible portions* of the isosurface rather than to the size of the input dataset. In fact, *any* algorithm has to examine the visible portions of the isosurface in order to render it, and hence the running time has to be at least proportional to the size of the visible portion of the isosurface (that is, proportional to the number of visible voxels on the isosurface). For an optimal value of k, our algorithm spends a small fraction of the overall time (less than 10%) to determine the lists of blocks for all the pixels through the BONO tree traversal, while almost all the remaining time is spent on determining and shading the voxels that intersect the isosurface. Note that after the first phase we don't shoot rays for the pixels with empty lists, skip a large number of ray traversal steps that are otherwise required in standard ray casting algorithm, and just spend a negligible amount of time on a small number of rays that at the very last step don't end up intersecting the isosurface. This is illustrated in Table 5 and Figure 6, which highlight in

black and yellow the areas explored by our algorithms but they are not part of the isosurface.

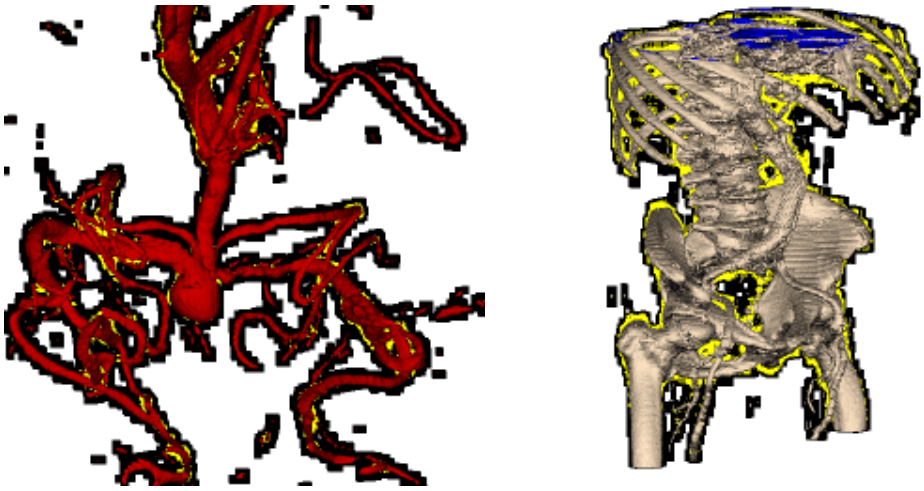

Fig. 6. The illustration of the adaptivity of our scheme using isosurfaces from Aneurism and Abdominal datasets. The pixels colored black or yellow are those from which non-intersecting rays are cast by our algorithm (and no other non-intersecting rays are cast). Color black specifies the area where rays do not intersect isosurface after processing the tile lists having less than k blocks; Color yellow represents the area for which our algorithm reverts to the standard ray casting algorithm but ends up with non-intersecting rays.

6.5 Cache Performance

A critical factor affecting the performance of any ray tracing algorithm is the irregular data access, which makes it difficult to exploit caches. This issue is even more critical on multi-core processors as the overhead of memory accesses becomes relatively more significant. During Phase I of our scheme, the data access is relatively regular as we process the data in object order and generate block lists. During Phase II, Our scheme sorts the small-size image tiles (typically, 8×8) into a Z-order, and group the tiles into decreasing size groups that depend on the weight of each tile, followed by dynamic allocation of these groups to the different threads. We now illustrate the resulting cache performance. Table 9 shows the cache miss rates achieved by our scheme during Phase II both on a single core and on an 8-core Clovertown for far and close views of the LLNL dataset. Here we have excluded the initial misses caused by the first time access to the data. These results clearly show that a thread will rarely need to access the main memory after the first time the data was loaded.

Number of Cores		Single-core		8-core	
View Type	# of data load requests	# of L2 cache miss	Miss Rate	# of L2 cache miss	Miss Rate
Far	66,712K	768K	1.15%	770K	1.15%
Close	53,380K	750K	1.41%	788K	1.48%

Table 9. Cache profiling of data request during Phase II for LLNL dataset with isovalue 70 and screen resolution 1024^2. Data load request is the number of requests issued for min/max and voxel values during the ray casting; L2 cache miss is the number of requests that fail to find the requested data inside the cache after the initial load.

6.6 Scalability of Our Algorithm

Our scheme achieves a very good scalability in terms of the number of cores used. The first phase divides the image equally among the core processors, and hence the work load is distributed almost equally among them. Before performing the ray casting phase, we create an ordered list of groups of small-size image tiles, which are then dynamically allocated to the threads as they become available. While the lists associated with each packet of rays are of different sizes, they are upper bounded by the value of k, which is typically less than or equal to 22. Given the dynamic allocation, we expect the loads on the different threads to be almost equally distributed, resulting in scalable performance. This is indeed the case as illustrated in Table 10, which shows the average frame rate over six views for the two different settings of the viewpoint on the LLNL dataset using a varying number of cores. The results are for 512^2 and 1024^2 screen resolution respectively.

Screen Size	512^2		1024^2	
Cores	Far-view	Close-view	Far-view	Close-view
1	1.77	1.97	0.51	0.56
2	3.53	3.82	1.02	1.12
4	7.03	7.64	2.04	2.23
8	13.08	14.53	3.98	4.31
Scalability over 8-core	92.4%	92.2%	97.5%	96.2%

Table 10. Average frame rate of our algorithm on Clovertown for the LLNL dataset at time step 250 under a varying number of CPU cores using 512^2 and 1024^2 screen resolution.

8-core / 1024^2		Number of ($\times 10^3$)			Time (msec)		
View	Proc.	Rays	Traversal	Intersect	Phase		Total
Type	No.	Cast	Steps	Voxels	I	II	
	0	98	1,380	277	19	223	250
	1	102	1,353	274	22	223	250
F	2	99	1,365	276	22	223	250
a	3	97	1,393	274	18	224	251
r	4	96	1,355	278	16	223	250
	5	101	1,344	275	22	223	250
	6	100	1,365	273	21	223	250
	7	94	1,361	275	18	223	250
$\frac{\sigma}{Ave} \times 100\%$		2.14	0.81	0.48	10.8	0.21	0.19
	0	147	850	382	21	202	231
	1	155	844	378	21	203	232
C	2	140	897	360	21	202	231
l	3	127	915	381	20	202	231
o	4	149	857	379	21	203	232
s	5	157	880	371	22	202	231
e	6	149	852	377	20	200	230
	7	151	844	382	21	203	232
$\frac{\sigma}{Ave} \times 100\%$		4.39	2.61	1.39	2.32	0.31	0.27

Table 11. The work from two Phases distributed among eight threads running among 8-core for 1024^2 screen and isovalue 70 along with their corresponding individual execution time. The tests are done on LLNL dataset for both far and close views. The work load is measured by the number of projected blocks and the number of ray traversal steps and voxel intersections respectively for Phase I and Phase II. Total includes the synchronization time and writing time of the frame buffer.

In fact, an examination of Table 10 reveals that the scalability of our algorithm is above 90% for both views for up to the maximum number of cores available on our Clovertown platform. Clearly, the advantage of ray-redistribution in our scheme is more useful for the sparse isosurfaces such as those generated by the Abdominal, Aneurism, and Skull datasets since many of the block lists will be empty.

Another way to illustrate the scalability of our scheme is through Table 11

that shows the loads on the different threads for the LLNL dataset for both the far and close views. We provide more details for Phase II since it constitutes approximately 90% of the total computational load. Note that the numbers of ray cast, octree traversal steps, and intersecting voxels are almost evenly distributed among the threads regardless of the viewpoint. Therefore the loads are extremely well-balanced among the different threads.

7 Conclusion

In this paper we presented a novel hybrid strategy for rendering isosurfaces by ray tracing. The resulting algorithm starts with an object order traversal that eliminates almost all the pixels with non-intersecting rays and creates short lists of ordered small data blocks for the remaining pixels, then apply ray casting for relevant pixels on these lists. We have shown that the total size of our indexing structure is very compact and that our performance is significantly superior relative to the published isosurface ray tracing algorithms. We have also shown that our algorithm can effectively exploit the memory hierarchies and its multithreaded implementation can efficiently utilize the multicore platform, which is available on almost all new processors. We presented the results of some of our extensive tests, showing interactive rendering rates for a variety of datasets, of widely different complexities, of size up to that of our main memory on a high resolution 1024^2 screen. All these results indicate that our scheme can easily achieve interactive rendering of isosurfaces of large scale volumetric scalar data on emerging multi-core processors.

Acknowledgment

We would like to thank Amitabh Varshney for his help and advice on this work. We would also like to acknowledge Mark Duchaineau at the Lawrence Livermore National Lab for making the Richtmyer-Meshkov instability dataset available to us and for guiding us through the initial stages of using it. This work was supported by the NSF research infrastructure grant CNS-04-03313.

Appendices: Isosurface Images from LLNL dataset

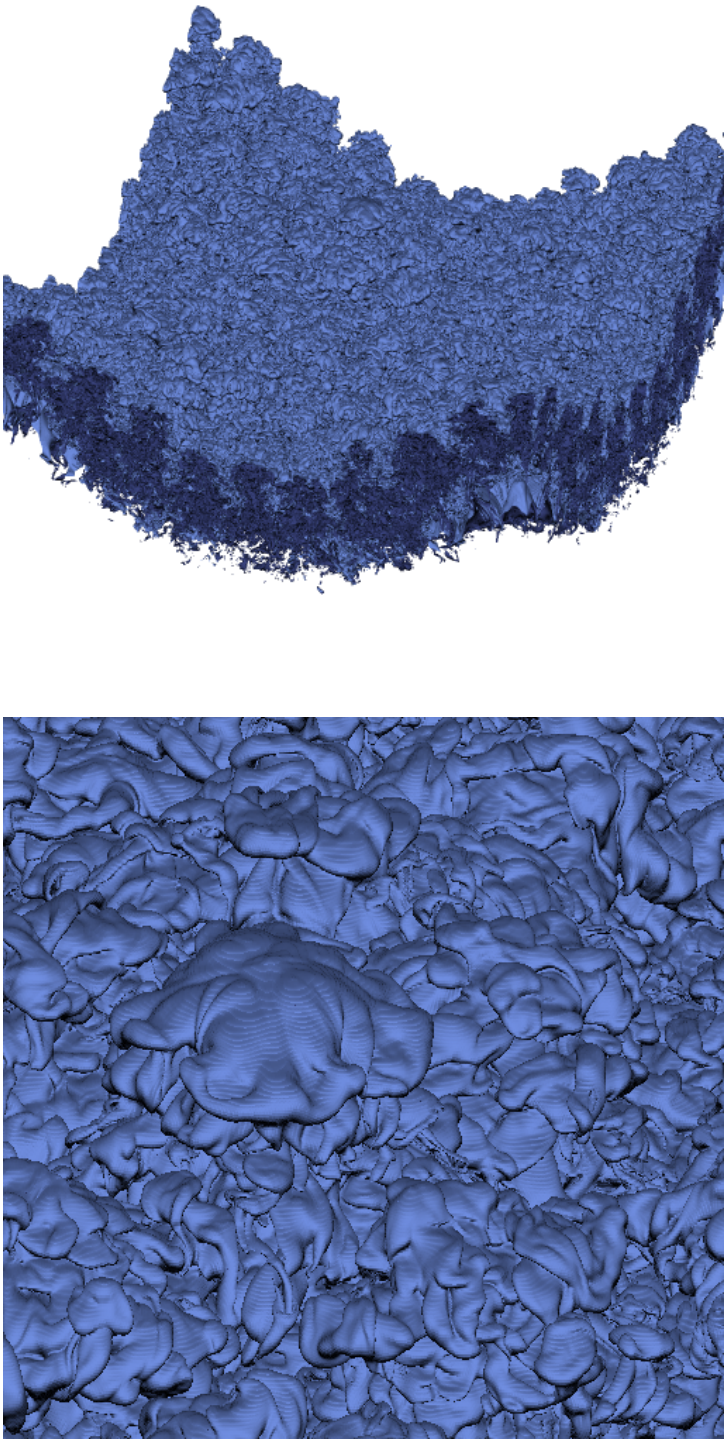

Fig. 7. Isosurface images of value 70 from Far-view and Close-view of LLNL dataset at time step 250.

References

[1] P. Cignoni, P. Marino, C. Montoni, E. Pupp and R. Scopigno, *Speeding up isosurface extraction using interval trees*, IEEE Transactions on Visualization and Computer Graphics Vol. 3, no. 2 April-June, pp. 158-170, 1997.

[2] D.E. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen, *Distributed Interactive Ray Tracing for Large Volume Visualization*, Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG), pp. 87-94, 2003.

[3] M. Gross, C. Lojewski, M. Bertram and H. Hagen, *Fast implicit kd-trees: accelerated isosurface ray tracing and maximum intensity projection for large scalar fields*, to appear in Proceedings of Computer Graphics and Imaging (CGIM), 2007

[4] J. Gao and H.-W. Shen, *Parallel View-Dependent Isosurface Extraction Using Multi-Pass Occlusion Culling*, Proc. of 2001 IEEE Symposium in Parallel and Large Data Visualization and Graphics. pp. 67-74, 2001.

[5] M, Hadwiger, C. Sigg, H. Scharsach, K. Bhler and M. Gross, *Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces, Proceedings of Eurographics 2005*, pp. 303-312, 2005

[6] 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. 115-124, 2006

[7] A. Knoll, C. D. Hansen and I. Wald, *Coherent Multiresolution Isosurface Ray Tracing*, Scientific Computing and Imaging Institute, University of Utah, Techinal Report No. UUSCI-2007-001, 2007.

[8] Y. Livnat and C. Hansen, *View Dependent Isosurface Extraction*, Proc. of Visualization '98. pp. 175-180, 1998.

[9] W. E. Lorensen and H. E. Cline, *Marching Cubes: A high resolution 3D surface construction algorithm, Maureen C. Stone, editor.* Computer Graphics (SIGGRAPH '87 Proceedings), vol. 21, pp. 161-169, July 1987.

[10] G. Marmitt, H. Friedrich, A. Kleer and S. Parker, *Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing*, Proceedings of Vision, Modeling and Visualization(VMV), pp. 429-435, 2004.

[11] B. Mora, J. P. Jessel and R. Caubet, *Accelerating volume rendering with quantized voxels*, IEEE/ACM SIGGRAPH Volume visualization and graphics symposium, pp. 63-70, Oct 2000

[12] B. Mora, J. P. Jessel and R. Caubet, *A new object-order ray-casting algorithm*, Proceedings of the conference on Visualization 2002, pp. 203-210, Oct 2002

[13] T. S. Newman and N. Tang, *Approaches that exploit vector-parallelism for three rendering and volume visualization techniques*, Computer and Graphics, Vol. 24, no. 5 pp. 755-774, 2000.

[14] S. Parker, P. Shirley, Y. Livnat, C. Hansen and P. P. Sloan, *Interactive ray tracing for isosurface rendering*, IEEE Visualization '98, pp. 233-238, Oct. 1998.

[15] A. Reshetov, A. Soupikov and J. Hurley, *Multi-level ray tracing algorithm*, ACM Transaction of Graphics, Proceedings of ACM SIGGRAPH 2005, 24(3), pp. 1176-1185, 2005.

[16] P. Sutton, C. Hansen, H. W. Shen and D. Schikore, *A case study of isosurface extraction algorithm performance*, 2nd Joint Eurographics-IEEE TCCG Symposium on Visualization, May 2000.

[17] Q. M. Shi, J. JaJa, *Isosurface extraction and spatial filtering using persistent octree (POT)*, IEEE Visualization and Computer Graphics 12(5), pp. 1283-1290, Oct. 2006

[18] L. Sobierarjski and R. Avila, *A Hardware Acceleration Method for Volume Ray Tracing*, IEEE Visualization, pp. 27-34, 1995.

[19] 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, 11 (5), pp. 562-572, 2005

[20] R. Westermann and B. Sevenich, *Accelerated volume ray-casting using texture mapping*, Proc. IEEE Visualization, 2001

[21] 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.

[22] J. Wilhelms and A.Van Gelder, *A coherent projection approach for direct volume rendering*, SIGGRAPH, pp. 275-284, 1991

[23] S. E. Yoon and D. Manocha, *Cache-efficient layouts of bounding volume hierarchies*, Computer Graphics Forum, Volume 25, Issue 3, 2006

[24] X. Zhang, C. L. Bajaj and V. Ramachandran, *Parallel and out-of-core view-dependent isocontour visualization using random data distribution*, Proc. Joint Eurographics-IEEE TCVG Symp. on visualization and graphics, pp. 9-18, 2002.