# ABSTRACT

Title of dissertation: NEW ALGORITHMIC TECHNIQUES FOR LARGE SCALE VOLUMETRIC DATA VISUALIZATION ON PARALLEL ARCHITECTURES

QIN WANG, Doctor of Philosophy, 2008

Dissertation directed by: Professor Joseph JaJa
Department of Electrical and
Computer Engineering

Volume visualization is widely used as an effective approach for the visual exploration, computational analysis, and manipulation of volumetric datasets. Due to the dramatic advances in imaging instruments and computing technologies, such datasets are now appearing at a very fast rate with increasingly larger sizes in many engineering, science and medical applications. Isosurface and direct volume rendering(DVR) are two of the most widely used techniques to render such datasets. This dissertation introduces novel techniques for rendering isosurfaces and volumes, and extends these techniques to multiprocessor architectures. We first focus on cluster-based techniques for isosurface extraction and rendering using polygonal approximation. We present a new simple indexing scheme and data layout approach, which enable scalable and efficient isosurface generation. This algorithm is the first known parallel algorithm to achieve provable load balancing on multiprocessor systems. We also develop an algorithm to generate isosurfaces using ray-casting on multi-core processors. Our method is based on a hybrid strategy that begins with an object order traversal of the data followed by ray-casting on ordered sets of an adaptive number of subcubes, one set for each small group of pixels on the image. We develop a multithreaded implementation, which uses new dynamic load balancing techniques that start with an image partitioning for the initial stage and then perform dynamic allocation of groups of ray-casting tasks among the different threads. The strategy ensures almost equal loads among the cores while maintaining spatial data locality. This scheme is extended to perform direct volume rendering and is shown to achieve similar improvements in terms of overall performance, load balancing, and scalability. We conduct a large number of tests for all our algorithms on the University of Maryland Visualization Cluster and on the 8-core Clovertown platform using a wide variety of datasets such as Richtmyer-Meshkov Instability dataset (7.5GB for each time step) and Visible Human dataset ($\sim$1GB). We obtain results that consistently validate the efficiency and the scalability of our algorithms. In particular, the overall performance of our hybrid ray-casting scheme achieves an interactive rendering rate on high resolution ($1024^2$) screens for all the datasets tested.

# NEW ALGORITHMIC TECHNIQUES FOR LARGE SCALE VOLUMETRIC DATA VISUALIZATION ON PARALLEL ARCHITECTURES

by

Qin Wang

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

## Acknowledgments

I would like to thank my advisor, Professor Joseph JaJa for bringing me an invaluable opportunity to work on challenging and interesting projects to fulfill my Ph.D. study. His expertise and research attitude inspire me throughout the course. Without his support and advising, I could not complete this program and learn so much.

I would also like to thank Professor Amitabh Varshney and his group in Compute Science department for the valuable research collaboration. Professor Varshney always made himself available for the discussion and his thoughtful opinion has been very helpful on the projects.

My gratitude goes to Fritz McCall and UMIACS help desk staff for their prompt technical assistance in the system setup and configuration. I am also grateful to my committee members for their serving on the board and their expertise to examine my research work.

I am indebted to my family for their endless supports and patience of many years.

# Table of Contents

# List of Tables

# List of Figures

# List of Abbreviations

| | |
|---|---|
| AMC | Active MetaCell |
| BBIO | Binary-Blocked I/O |
| BONO | Branch on Need Octree |
| CPU | Central Processing Unit |
| CT | Computer Tomography |
| DVR | Direct Volume Rendering |
| ESL | Empty Space Leap |
| GPU | Graphics Processing Unit |
| LLNL | Lawrence Livermore National Laboratory |
| LOD | Level of Detail |
| MIMD | Multiple Instruction Multiple Data |
| MRI | Magnetic Resonance Imaging |
| NIH | National Institutes of Health |
| NUMA | Non-Uniform Memory Access |
| RMI | Richtmyer-Meshkov Instability |
| SIMD | Single Instruction Multiple Data |
| TB | Tera-Byte |

# Chapter 1

# Introduction

Visualization is a process that creates images to convey salient information about the underlying data. Because of the extraordinary capability of the human visual system to interpret images quickly and effectively, visualization is a very important method for exploring and analyzing the features, patterns and relations embedded in the data. Nowadays, with the fast evolving computing technology, visualization is playing an ever-expanding role in applications in science, engineering, medicine, multimedia, finance, etc. In particular, scientific visualization is now an integral component of scientific computation and simulation, which aims to create a visual representation of complex scientific data to provide insights underlying the physical phenomena. Such visual abstraction and reduction of underlying data into images is in general a very computationally demanding process.

## 1.1 Challenges

During the past three decades, there has been unprecedented growth in computational and acquisition technologies, which resulted in the increasing capability of investigating the physical world in detail with high precision as well as modeling and simulating very complex phenomena. It is not uncommon that hours, or even days of simulation running on high performance computers produce vast amount of data ranging from hundreds of giga-bytes to tens of tera-bytes. As an example of such a dataset, consider the process of the Richtmyer-Meshkov instability in inertial confinement fusion and supernovae from the ASCI team at the Lawrence Livermore National Labs(LLNL). This dataset represents a simulation in which two gases, initially separated by a membrane, are pushed against a wire mesh. These are then perturbed with a superposition of long wavelength and short wavelength disturbances and a strong shock wave. The simulation took 9 days on 960 CPUs and produced about 2.1 TB of simulation data. The data shows the characteristic development of bubbles and spikes and their subsequent merger and break-up over 272 time steps, each consisting of $2048^2 \times 1920$ volume of one-byte scalar field. Another example is the popular medical dataset coming from the Visible Human Project established by the National Institute of Health(NIH). The generated dataset contains $1,871$ cross-section CT and anatomical images in the size of 15 GB for the male subject while $5,189$ images of 39 GB for the female. In the past decade, a great number of efforts were made to visually explore and analyze such large scale datasets interactively with mixed results.

Since the visualization of large scale data demands a great deal of computation and irregular communication, previously only supercomputers equipped with proprietary graphic system, such as SGI Reality Monster with 128 processors, were able to support the visual exploration on large datasets. However, such super computers can only be afforded by a limited number of researchers. Since early 1990's, driven by the availability of high speed networks and cheap prices of PCs, clustering has become a cost-effective platform to provide massive computing power. In a cost-effective way, visualization systems built upon a cluster consisting of off-shelf computers with GPU cards have been since then exploited to visualize 3D datasets. More recently, the newly emerging multi-core processor technology which includes multiple CPUs on a single chip is attracting considerable attention from researchers to conduct visualization on this new platform. Although there are a number of visualization systems and techniques developed using these parallel platforms, they are usually targeted for several orders of magnitude smaller datasets, and do not scale to gigabyte or terabyte-sized datasets. Hence, visual exploration and interaction with large scale datasets requires the development of a high-performance visualization software infrastructure running on affordable parallel platforms that can essentially handle the growing size of datasets, extract the features of interest efficiently and render them interactively on high resolution screen.

## 1.2 Contributions

This dissertation considers the problem of visualizing large scale datasets using isosurfaces and Direct Volume Rendering (DVR) technologies on two different parallel architectures, namely multiprocessor clusters and multi-core processors. We have developed three new algorithms that offer significant improvements over prior schemes. Part of the work in this dissertation appears in [24] and [69].

Our first major contribution concerns the extraction of a polygonal approximation of isosurfaces. A new compact indexing structure and a new data partitioning scheme are developed for out-of-core isosurface extraction and rendering of large scale data on a multiprocessor environment. This scheme results in the following contributions:

- The algorithm uses a smaller indexing structure and a more effective bulk data movement than the best known previous algorithms while achieving similar asymptotic bounds. In particular, the size of our indexing structure is shown to be orders of magnitude smaller than that of the interval tree for a number of well-known datasets.

- Our scheme can be implemented on a distributed storage multiprocessor environment such that the data distribution across the local disks of the different processors results in a *provably* balanced workload irrespective of the isovalue. Moreover, the total amount of work across the different processors is about the same as that required by the serial version of the algorithm.

- The experimental results show that isosurfaces can be generated and rendered at the rate of $3.5 \sim 4.0$ million triangles per second on the Richtmyer-Meshkov dataset using our algorithm on a single processor. On a 16-node cluster, we achieve scalable performance across widely different isovalues with a performance of up to 60 million triangles per second. The experimental results also show that our algorithm achieves excellent load balancing for a variety of datasets over a wide range of isovalues.

Our second major contribution is the development of a novel hybrid strategy for rendering isosurfaces using ray-casting on a multi-core processor. The proposed method is based on a hybrid strategy that begins with an object order traversal of the data followed by ray-casting on ordered sets of an adaptive number of subcubes. Compared with existing ray-casting approaches, our scheme achieves the following:

- A short list of possible candidate data blocks is generated for each small set of contiguous pixels through the traversal of a *BONO tree* (Branch On Need Octree), built upon a coarse version of the volumetric data. Such a process can be performed extremely fast because the BONO tree is very compact and the imposed upper bound of the data block list restricts the traversal significantly. This enables us to identify almost all the pixels with rays not intersecting the isosurfaces through the BONO tree travel, and the percentage of the non-intersecting rays cast is extremely small.

- We cast rays that traverse through a limited number of blocks in a front to back order, and skip 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 spatial locality of pixels can be exploited 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.

- The multi-threaded implementation on multi-core processors, using new dynamic load balancing techniques, ensures almost equal loads among the cores while maintaining spatial data locality.

- The extensive testing of our algorithm on a variety of datasets, of widely different complexities, indicates that the scheme can easily achieve interactive high resolution rendering of isosurfaces of large scale volumetric scalar data on emerging multi-core processors.

Our third major contribution is to extend the hybrid ray-casting scheme to accelerate direct volume rendering on large scale datasets as well. Our extended scheme achieves:

- The BONO tree is augmented with extra dimensional information to accommodate the specification and application of interactive 2D transfer functions at runtime.

- A novel block discrimination method is proposed to classify the blocks into various types and directly support empty space leaping in an effective way in the case of direct volume rendering.

- The multi-threaded implementation demonstrates a high degree of scalability, excellent load balancing, and effective memory management on multi-core processors.

- Extensive experimental tests show significantly superior performance over previously published algorithms for direct volume rendering by ray-casting, and result in interactive rates for very large datasets such as the Lawrence-Livermore instability dataset on high-resolution screens.

## 1.3 Outline

The rest of this dissertation is organized as follows. Chapter 2 introduces existing visualization technologies as well as some related acceleration techniques. Chapter 3 reports on the previous algorithms and methods used to render isosurfaces and DVR, but mainly focusing on multiprocessor environment and multi-core processors. Chapter 4 presents a new simple indexing scheme and data layout approach, which enables scalable and efficient isosurface generation resulting in the first known parallel algorithm to achieve provable load balancing on multiprocessor cluster systems. The novel hybrid scheme that improves isosurface ray-casting is presented in Chapter 5 and its extension to direct volume rendering is described in Chapter 6. We conclude in Chapter 7.

# Chapter 2

# Volume Visualization

Volume visualization is used to create images from scalar and vector datasets defined typically on 3D or 4D space. It involves a process of transforming a multidimensional dataset onto a 2D image plane to gain insights of the salient features and the structure contained within the data. The transformation takes place by sampling the 3D data, interpolating the data value, classifying the optical properties of the material, and compositing the visual effects to come up with 2D images. There are two widely used methods for volume visualization, *isosurfaces* or *Direct Volume Rendering(DVR)*. The isosurface method computes and renders surfaces with constant density value while the DVR method can more generally reveal the underlying translucent structure and the features of the data. Before discussing various techniques regarding these two methods, we first introduce the data representation of the underlying datasets.

## 2.1 Volumetric Data

A volumetric dataset consists of a collection of values obtained by sampling a function at various locations in a 3D space, which can be in structured or unstructured format. A structured dataset is associated with certain form of regularity in sampling positions, while positions can be arbitrary in an unstructured dataset.

This thesis primarily studies methods that are applicable to structured datasets dominated in many scientific domains. Typically, instrument measurements and computer simulation are the two main sources for such datasets. For example, in the Visible Human Project, Computer Tomography(CT), Magnetic Resonance Imaging(MRI), and Anatomical images are obtained via the experimental measurement of male and female cadavers. These 2D images are stacked together to form a 3D data grid with each pixel representing a grid point. On the other hand, the RMI dataset from LLNL is generated directly by computer simulation, with one scalar density value at each grid point. An illustrative 3D grid is shown in Figure 2.1, where each point on the grid is called a *voxel* (as analogous to *pixel* in 2D image), and is associated with a scalar value. Alternatively, this grid can be viewed as a collection of cubic *cells*, each of which contains eight neighboring grid points. The $V_{min}$ and $V_{max}$ of a cell are defined respectively as the minimum and maximum of eight scalar values belonging to the cell. Based upon such volumetric data representation, various techniques have been developed to visualize the dataset using isosurfaces and direct volume rendering.

## 2.2 Basic Visualization Techniques

### 2.2.1 Isosurface

Given a scalar density function $F(x)$ over a 3D geometric space $G$, $x \in G, F \in V$ ($V$ specifies a scalar value space), an isosurface $S$ consists of the 3D point $x$ such that $F(x) = \lambda$, for some constant isovalue $\lambda$.

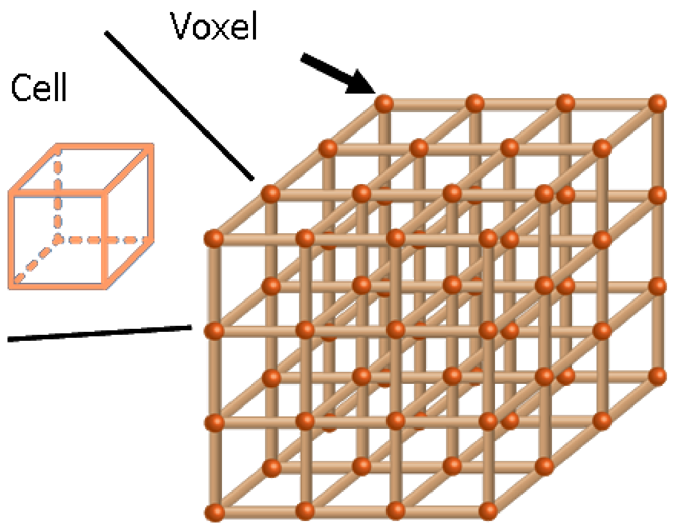

Figure 2.1: Illustrative 3D grid, each grid point contains a scalar value measured at a discrete sampling location.

As the density function $F(x)$ is only known through a set of sample values $f_i$ at discrete sampling locations over the 3D space, i.e., $F = \{f_i\}, f_i \in V$, in practice an isosurfaces can be approximated through interpolated values over the set $\{f_i\}$. For the volumetric data in the 3D grid representation, eight scalar values of a cell are referenced to interpolate points on the isosurface located inside each cell. Two different approaches are adopted to carry out the interpolation process, one by polygonal approximation, the other by ray-casting.

### 2.2.2 Polygonal Approximation

A popular method for computing an isosurface is by approximating the surface with a polygonal mesh followed by rendering the mesh using the usual graphics pipeline. The isosurface inside each cubic cell of a volumetric dataset can be approx-

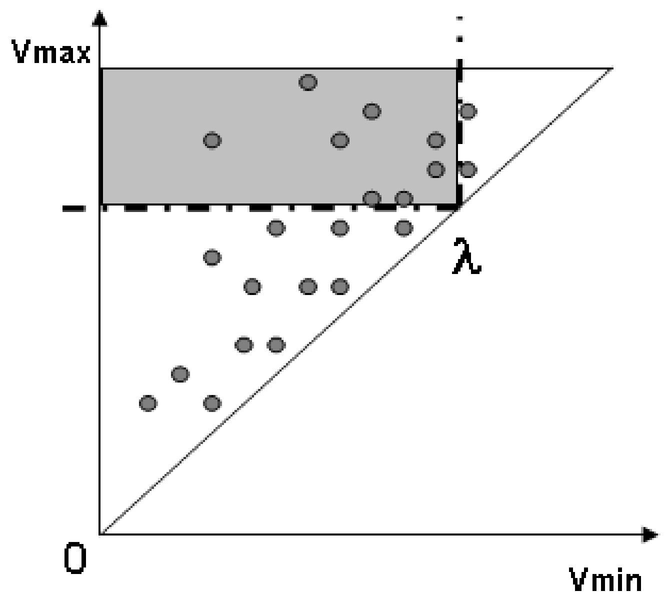

Figure 2.2: Span space for active cells.

imated by a set of polygonal geometries such as triangles as follows. The process begins by examining the isovalue $\lambda$ with $V_{min}$ and $V_{max}$ of the cell. If the condition $V_{min} \leq \lambda < V_{max}$ is satisfied (such as those cells located in the rectangular shaded region in span space of Figure 2.2), the cell intersects isosurface and will be called active. An intersecting position is determined along each edge by linear interpolation based upon scalar values of two end voxeles and isovalue $\lambda$. After these intersecting positions are obtained from edges, a triangle mesh can be composed to approximate the isosurface inside of the cell. There are in total of at most 256 intersecting scenarios. In reality, 15 cases are sufficient since the other cases can be generated by symmetry. These are represented in Figure 2.3.

This type of geometric approximation is also called *triangulation*, and was first proposed by Lorensen in [14]. In this scheme, the isosurface is approximated from

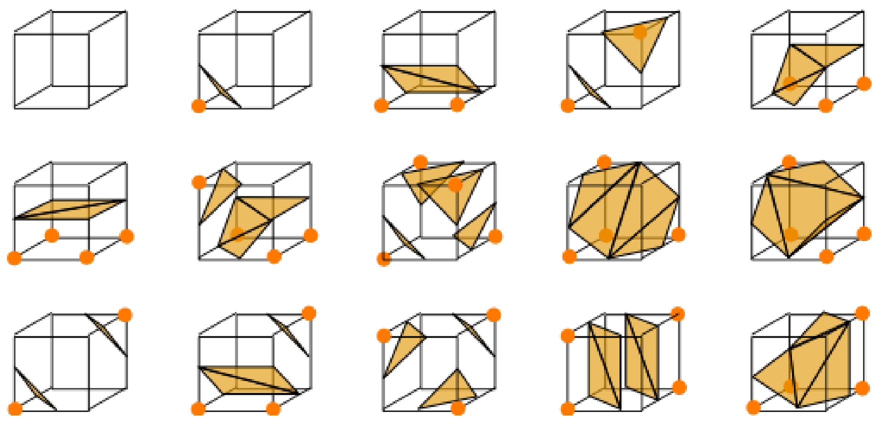

Figure 2.3: 15 unique triangulation cases among cubic cell.

each active cell one after another in a brute force manner. As a result, a great number of triangles are generated for large scale dataset, which could amount to massive computations. There have been many acceleration techniques proposed to efficiently locate the active cells, some of which will be introduced in Section 2.3.

### 2.2.3 Pixel-based Ray-casting

The triangulation process generates a geometric approximation of isosurfaces, whose rendering may produce visual artifacts. Moreover, this process doesn't take into consideration the position of the viewpoint and hence may waste a great deal of computation on triangles that may not be visible from the viewpoints of interest.

Another technique is to use ray-casting. For each pixel on the screen, a ray is cast along the viewing direction across the data grid. When the ray encounters an active cell, a trilinear interpolation process takes place within the cell to locate the

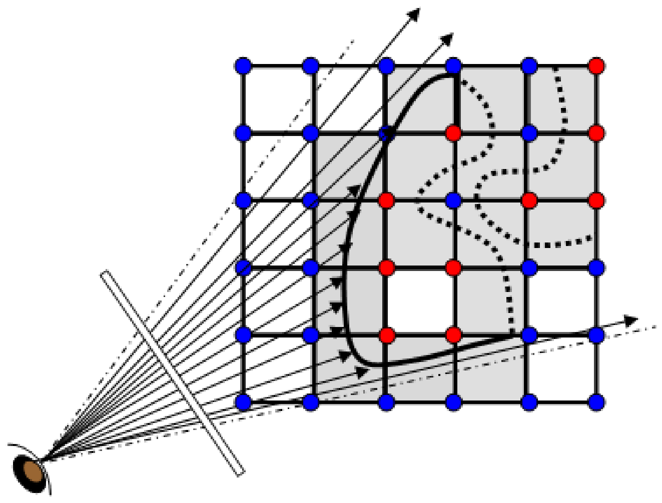

Figure 2.4: Illustrative recursive ray traversal in 2D data space.

intersecting position where its density value is equal to $\lambda$ [36]. The ray goes through the cells following the forward viewing direction until such intersecting position is found. In such case, a shading is performed for the corresponding pixel on the screen relative to the lighting and selected shade model. The ray-casting process is inherently a view-dependent approach for isosurface rendering since it only locates and shades the first intersection voxel and skips all the data behind. Also, each intersecting position along the ray is accurately computed such that the visual quality of ray-casting isosurface can be very high. Unlike the polygonal approximation in which the performance is directly related to the number of extracted triangles, the pixel-by-pixel approach of ray-casting makes the rendering speed more sensitive to the screen size other than the size of the dataset and the complexity of the isosurface. An illustrative ray-casting diagram in 2D space is shown in Figure 2.4 with shaded squares specifying the active cells and dashed lines indicating the part of isosurfaces invisible from the viewpoint.

### 2.2.4 Direct Volume Rendering

Although isosurfaces play a very important role in volume visualization, they have several drawbacks. First, triangle meshes are only an approximated representation of a surface; second, much of the information gets lost by extracting only surfaces from the original 3D dataset since the surface is just a 2D geometric representation; third, amorphous phenomena, such as clouds, gas, soft tissue, etc., cannot be adequately represented by means of surfaces. Therefore, it is desirable to generate a volumetric representation that extracts and renders the information directly from the dataset without going through an intermediate geometric approximation.

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 [60], the color on the final image can be computed by the following integration:

$$C = \int_0^L C_s \tau_s e^{-\int_0^s \tau_t dt} ds \tag{2.1}$$

$C_s$ and $\tau_s$ are the material color and opacity per unit length respectively at the location $s$. In most cases, this integration has no analytic solution but can be composited approximately in front-to-back order using a recursive Riemann sum [49] as in the formula below:

$$\alpha_{i+1} = \alpha_i + (1 - \alpha_i)\alpha_s \tag{2.2}$$

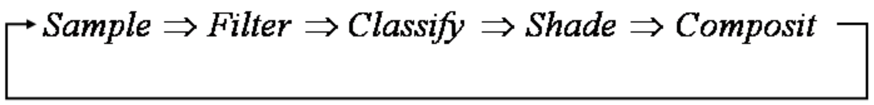

Figure 2.5: DVR pipeline processing.

$$C_{i+1} = C_i + (1 - \alpha_i)\alpha_s C_s \tag{2.3}$$

where $\alpha_s = \int_{s_i}^{s_{i+1}} \tau_t dt$, $\alpha_s$ and $C_s$ are usually mapped from the properties of the voxels (such as scalar field value and/or gradient value) by a user-specified *transfer function*. The stages in the pipeline process of direct volume rendering are illustrated in Figure 2.5. First, a series of discrete locations are sampled uniformly along each ray one at a time; next, the scalar value at each location is obtained by applying certain filters such as trilinear interpolator; third, the optical properties of the material, captured by opacity and color, are obtained by classification through a transfer function; There can be an optional shading process to factor in the contribution from the lighting effect; At last, the composition step accumulates the opacity and colors over the ray in a front-to-back order following the recursive composition equations 2.2 and 2.3. In general, the DVR process is much more computationally intensive than isosurface computation in that the final pixel color is determined by the composition of visual contributions from the data throughout the entire space.

### 2.2.5 Transfer Function

The transfer function, which maps the material properties of a voxel onto opacity and color, is critical in capturing and rendering the features and patterns of interest from the underlying dataset. Transfer functions can be one-dimensional, i.e. mapping from scalar density value to optical properties, or multi-dimensional using the additional dimensions for gradient and/or curvature, which can improve the capability of distinguishing the features based upon their higher order derivatives. The classification via transfer function is achieved through a table mapping, which is straightforward and efficient for one or two dimensions but much more complex to manipulate for higher dimensions. A transfer function could introduce extra artifacts when the mapping from density value to opacity/color raises the Nyquist frequency of the underlying data. However, increasing the sampling rate would produce significantly more computational work load and compromise the rendering speed. One way to suppress such artifacts without increasing the sampling rate is to adopt pre-integral transfer functions. The method approximates $\alpha_s$ through the integration over the density field instead of the spatial space as follows:

$$\alpha_s = \int_{s_i}^{s_{i+1}} \tau(t)dt = \int_{f_i}^{f_{i+1}} \tau(f)\frac{dt}{df}df \approx \frac{\Delta s_i}{\Delta f_i}(T(f_{i+1}) - T(f_i)) \tag{2.4}$$

Where $T(f) = \int_0^f \tau(f)df$, called pre-integral transfer function, which prevents the piecewise mapping from the field value to optical properties.

## 2.3 Acceleration Techniques

Volume visualization is a computationally demanding task especially for large scale datasets and high resolution screens. There are various techniques that have been developed over the past 15 years or so to accelerate the process of isosurface generation and direct volume rendering. The most widely adopted techniques are described next.

### 2.3.1 Metacell

Metacell is defined as a group of neighboring cells within a spatial bounding box. In particular, the metacell is trivially a subcube in the case of structured grids. The initial concept of metacell is driven by clustering the cells into the size of one or several contiguous disk blocks to facilitate I/O access. Using metacells can also reduce the data duplication on the boundary voxels, otherwise almost eight times size of data would be expected in the representation of a collection of individual cells. Obviously, data locality is improved by utilizing metacell as well. In addition, another advantage of metacell is that a hierarchical spatial indexing structure can be constructed upon metacells, the size of this hierarchical structure will then be compact, and hence will improve the memory access performance for large datasets.

### 2.3.2 Hierarchical Structures

Hierarchical structures have been widely used to avoid processing the parts of the data which do not contribute visually to the final image. Two widely used structures are the octree [23] and the kd-tree [13]. A node of such hierarchical structures represents a spatially bounded region. An octree recursively divides the spatial space into eight sub-regions while a kd-tree splits the space recursively into two halves alternatively along each spatial dimension. In extracting isosurfaces by polygonal mesh approximation, each node of the octree or kd-tree is augmented with the minimum and the maximum scalar values ($V_{min}$ and $V_{max}$) of the subtree rooted at this node. If the stabbing condition is not satisfied at the node (i.e. the isovalue $\lambda$ is not between $V_{min}$ and $V_{max}$), the whole subtree is removed from further processing. Another important hierarchical structure is the interval tree [2, 28] which enables the extraction of the active cells in an optimal fashion. Basically, each node of an interval tree contains a set of intervals denoted by the values of $V_{min}$ and $V_{max}$ which sandwitch a median value associated with the node. The intervals with both $V_{min}$ and $V_{max}$ smaller (or larger) than the median value belong to the left (or right) subtree respectively. Based upon their individual median values, the nodes constitute a binary tree which allow us to determine the intervals containing a given isovalue through a binary search process. More discussions will be given on this structure along with the description of our new algorithm in Section 4.2.

Regarding the ray-casting technology, each ray advances forward along the viewing direction in a front-to-back order using the hierarchical structures. An illustrative 2D case of recursive ray traversal is shown in Figure 2.6. Starting from the root node, each ray intersecting the bounding box of the current node is tested for the intersection with the middle planes perpendicular to the three axes. These intersecting positions located within the bounding box are then sorted according

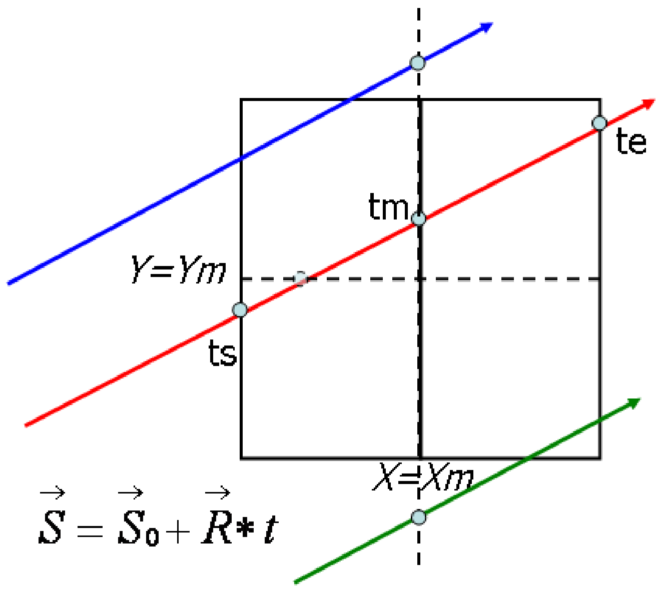

Figure 2.6: Illustrative recursive ray traversal.

to their distance to the ray source position. Based upon their ordering, the front-to-back visiting order to the current node can be easily established as well as the recursive traversal of the ray. A hierarchical indexing structure provides a critical support for other acceleration techniques introduced below.

### 2.3.3 Empty Space Leaping

Important features in volumetric datasets are partially captured by isovalues or transfer functions. Parts of a dataset may not produce any visual effect on the final image. In the method of ray-casting, the ray can skip the empty (or transparent) region when traversing across the dataset to save computation and memory access. Using the octree or kd-tree, the visibility of each block corresponding to a node can be quickly determined by examining the isosurface stabbing condition or testing the difference of the pre-integral transfer function value between $V_{min}$ and $V_{max}$ of the node. If there is no visual contribution from the region represented by this node, the ray is advanced forward to skip the traversal within the subtree under this node. The empty space leaping is particularly effective when the features of interest come from a limited part of the whole dataset.

### 2.3.4 Early Ray Termination

Another important acceleration technology for ray-casting is early ray termination. As to the isosurface rendering, the early ray termination is an inherent part of the technique as the ray stops after intersecting the isosurface. When rays traverse across a volume grid in front-to-back order for direct volume rendering, many of the rays may quickly accumulate an opacity close to 1 by Equation 2.2. This means that features of the dataset located in the back are occluded and need not to be processed to produce the final image. Those rays can be terminated before they pass through the entire grid.

### 2.3.5 Packet-of-Rays

The packet-of-rays exploits the CPU SIMD extensions that exist in many current architectures to apply a single operation on four operands at once. In such way a packet of four rays can be traversed as one single unit throughout the volumetric dataset. An extended version of this technology can traverse a beam of rays (not limited to four rays) across the data grid in the hope of achieving significant speedups. But, both techniques suffer from the overhead caused by merging and splitting of rays or the frustum that encloses the ray beam during the traversal process. The relatively high speedups reported are only observed in coherent scenes.

# Chapter 3

# Previous Work

This chapter will present an overview of a number of algorithms and techniques directly related to this dissertation. These methods are divided into three categories: isosurface extraction and rendering, isosurface ray-casting, and direct volume rendering. The review begins by describing the sequential algorithms but the focus will be more on parallel schemes developed for multiprocessor systems. The contributions made by these schemes are summarized along with their limitations especially when dealing with the challenges imposed by large scale datasets. The discussion aims to introduce the great efforts many researchers have devoted to this field and at the same time to provide sufficient context to understand the contributions of our work.

## 3.1 Isosurface Extraction and Rendering

The Marching Cubes(MC) algorithm proposed in [14] scans the entire cell set, one cell at a time, and determines whether a cell is cut by the isosurface, and in the affirmative generates a local triangulation of the isosurface intersection with the cell. The overall triangular mesh is then rendered. Many improvements to this initial algorithm have been reported in the literature. Some algorithms attempt to reduce the number of examined cells by using a spatial data structure such as the octree [22, 23], while others partition the range of the scalar field values and construct an index on the span space [20]. A theoretically optimal algorithm was described in [8], and involves the construction of an interval tree on the scalar field intervals defined by the cells. Such a data structure enables the exploration of only the active cells (cells that intersect the isosurface) and hence it is output sensitive. This algorithm was later generalized into a theoretically optimal out-of-core isosurface extraction strategies in [4, 5]. For multiprocessor environments, several parallel algorithms have been reported for the case when the data can fit in the main memory [9, 10, 16, 18, 19, 20]. As to larger scale datasets, parallel out-of-core algorithms such as those in [3, 7, 21, 25, 26, 27] have been reported.

For a multiprocessor cluster environment in which each processor has its own main memory and local disk, the processors communicate and exchange data through an interconnection network using message passing. Critical factors that influence the performance include the amount of work required to generate the index and organize the data on the different disks (preprocessing step); the relative computational loads of the different processors corresponding to an arbitrary isovalue; the performance of the rendering and rasterization into a single display; and the total amount of work relative to the (best) sequential algorithm. The preprocessing step described in [5, 6, 7] involves partitioning the dataset into metacells and building a B-tree like interval tree, called Binary-Blocked I/O interval tree (BBIO tree). The computational cost of this step is asymptotically similar to an external sort, which is likely to be too expensive in practice. The isosurface generation requires that a host traverses the BBIO tree to determine the indices of the active metacells, after which jobs are dispatched on demand to the available processors. In addition to the substantial preprocessing overhead, a significant bottleneck with this scheme is the host overhead in coordinating and dispatching jobs, and the unpredictability of the access pattern to the available disks. The algorithm described in [25] attempts to solve the load balancing problem by distributing the data based on a range space partition. The range of possible field values is partitioned into a number of intervals. Blocks are then assigned to triangular matrix entries depending on which intervals a block spans. An external interval tree (BBIO tree) is then built separately for the data on each processor. Again this strategy involves a very expensive preprocessing step but in addition there is no guarantee of load balancing among the processors. In fact, it is easy to construct a dataset that will result in extremely unbalanced loads among the processors. Moreover, while the BBIO is "asymptotically optimal" for I/O performance (in terms of the number of active metacells brought to memory and not necessarily the active cells), it does incur a significant overhead in terms of size and performance. The extracted local surface is then streamed to parallel rendering servers, followed by compositing the outputs of the different frame buffers to a tiled-display. The preprocessing algorithm described in [27] is based on partitioning the range of scalar values into equal-sized subranges, creating afterwards a file of subcubes for each subrange. The blocks in each range file are then distributed across the different processors, based on a work estimate of each block. As in [25], the preprocessing is computationally expensive and there is no guarantee that the loads of the different processors will be about the same in general.

In summary, existing parallel out-of-core algorithms either require unpredictable data accesses to the disks, which can cause a considerable overhead, or use complex indexing schemes with no guaranteed load balancing. The parallel out-of-core algorithm proposed in this thesis work uses a simple indexing scheme with provable load balancing among the processors (regardless of the datasets or the isovalue) and incurs almost no overhead relative to the efficient sequential version of the algorithm.

## 3.2 Isosurface Ray-Casting

Isosurface ray-casting directly generates the isosurface by shooting rays from the viewpoint through the pixels and computing the intersections of these rays with the isosurface. As a special case of ray tracing technique, ray-casting only considers and traces the primary ray. Nevertheless, ray-casting delivers most effectively the visual information of isosurfaces and is the most time-consuming component of ray tracing. 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. [19] 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. [42] 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 [23, 33], 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 [30] 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 [42].

The multi-layer ray tracing algorithm (MLRTA) reported in [39] exploits spatial coherence in image space to speedup the rendering of geometric objects of a few million triangles. The speedups achieved by the MLRTA method basically rely on coherent scenes containing geometric objects of large 3D extension to ensure that a wide range of ray beams treated as a single traversal entity can quickly detect deep entry points. However, in order to handle small objects, a repeated tile splitting process is required to subdivide the ray beams gradually, thereby creating a significant overhead and resulting in a traversal that is similar to the octree traversal. Hence, it is questionable whether the method will perform well on complex (and possibly incoherent) isosurfaces in large scale datasets such as LLNL where the concept of large object extension is not applicable. 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 [33, 34, 30]. 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 [33] 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 [34], resulting in faster rendering of LOD data in some cases.

The ray-casting implemented on GPUs also appeared in the past few years with the advent of improved GPU programmability. The GPU-based ray-casting implementation is reported in [43, 32]. However, the GPU-based approach is constrained by its on-board memory size (usually 512MB) and its fairly strict SIMD programming model.

## 3.3 Direct Volume Rendering

Software techniques developed for direct volume rendering fall primarily into three main categories: Ray-casting [55, 56], Splatting [72] and hybrid strategies such as the Shear-Warp algorithm [53]. Although the GPU-based technology currently delivers the fastest rendering of volumetric data due to its massive pipeline stream processing, much improved GPU programmability, and 3-D texture mapping [47, 52, 65, 48, 11], the high rendering frame rate is observed when the processed data is able to fit in the GPU on-board memory. Currently, the limited size (i.e. 512MB $\sim$ 1024MB) of on-board memory prevents GPU from achieving interactive volume rendering on large size datasets. Researchers have considered a number of parallel volume rendering schemes for ray-casting [63], splatting [62] and shear-warp [54, 66] 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 [62]. Shear-warp generally produces low quality images due to its bilinear interpolation with severe artifacts showing up when changing the viewing direction around $45^o$ [53, 54]. Consequently, ray-casting seems to be the preferred technique in terms of its technical simplicity and ability of producing 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 size), which makes it hard to achieve interactive rendering for moderate size datasets. Refer to [61, 49] for a good overview and a comparison of the various volume rendering techniques.

Since the direct volume rendering method by ray-casting was first published by Levoy in [55], many alternate versions, such as those reported in [56, 68, 64, 70, 71], have been introduced. However, empty space leaping requires either a hierarchical structure (i.e. multilevel-grid [64], octree [56], kd-tree [68], etc) which incurs a significant overhead, or a space-consuming per-vertex register structure such as DFB-jumping described in [70] which however can not accommodate interactively changing the transfer function. To improve the efficiency of ray-casting for direct volume rendering, a two-step method is proposed in [41]. The first step consists of projecting the boundary cells onto the image plane using graphics hardware, and the second step applies the octree ray tracing but now slightly more constrained. Some other schemes use quantized voxels to accelerate volume rendering [37], or cache-efficient layouts of bounding volume hierarchies [44] to improve kd-tree access during ray casting for some medium size of datasets. 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 [67] or cell-template [71]. 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 [38], 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 [63, 46, 57, 58, 64, 45]. 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 serial ray-casting algorithm [56] on shared-memory MIMD system in [63] and achieved the scalability around $70 \sim 80\%$ by utilizing dynamic task stealing from other processors at runtime. Other similar implementations [46, 64] 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 [57, 58, 59]. But, these algorithms require additional merging and re-sorting of 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 [54, 66] 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 [45] divides full screen into small image tiles and the dataset hierarchically into bricks. The algorithm obtains data bricks that intersect with viewing frustum by repeated octree traversal, which incurs a significant overhead, in order to achieve load balancing with dynamically assigned both image tiles and data bricks among processors.

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. Initially, the transfer function was applied at the preprocessing step as in [56], 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 [50, 51]. Furthermore, pre-integral transfer function was introduced in [47] 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.

Chapter 4

# Compact Interval Tree for Isosurface Extraction and Rendering

In this chapter, we introduce novel techniques to enable efficient isosurfaces extraction for large scale datasets that do not fit in main memory. Specifically, we develop a simple indexing scheme, called *compact interval tree*, which is orders of magnitude smaller than existing indexing structures, and meanwhile provides more efficient bulk data movement. The parallel implementation of this algorithm on a distributed multiprocessor environment introduces a new scheme to distribute the data, which results in a provable load balanced workload among the processors regardless of the datasets or the isovalue, and incurs almost no overhead relative to the efficient sequential version of the algorithm. We make use of the University of Maryland visualization cluster in which each node consists of a 2-way symmetric multiprocessor with 8GB of main memory, a 60GB local disk, and an NVIDIA GPU (Graphics Processing Unit). The nodes are interconnected via a 10 Gbps InfiniBand, with four nodes reserved for compositing the image frame buffer outputs from the processors and displaying the results on a wall-sized screen (multi-projector display). We start by defining the out-of-core computational model used to evaluate our methods for isosurface extraction.

## 4.1 Isosurface Extraction Computational Model

Due to the electromechanical components, disks have two to three orders of magnitude longer access time than random-access main memory. In order to amortize the access time over a large amount of data, a single disk access reads or writes a block of contiguous data at once, typically of size 4KB or 8KB. We will use the standard model [1] to measure the I/O performance of our algorithms. We denote the input size by $N$, the disk block size by $B$, and the size of the main memory by $M$. In this work, we are assuming that $N$ is much larger than $M$, which in turn is much larger than $B$. The I/O performance of an external memory algorithm is measured by the number of I/O operations, each such operation involving the reading or writing of a single disk block. As a result, scanning contiguously the input data requires $O(N/B)$ I/O operations.

Our parallel computation model consists of a number of processors, each with its own local main memory and disk, interconnected through an interconnection network. The processors communicate and exchange data through message passing using the interconnection network. Since we are interested in large scale data, we assume that the input data resides on the disks of the available processors. A preprocessing step involves rearranging the data among the disks with the goal to optimizing the access patterns to the data, and to distributing the computational load equally among the processors.

## 4.2 Compact Interval Tree Indexing Scheme

Our algorithm can handle both structured and unstructured grids and makes use of the metacell notion introduced in Section 2.3.1. In general, a metacell consists of a cluster of neighboring cells. All the metacells are about the same size, which is a small multiple of the disk block size. In particular, for the structured grid of the Richtmyer-Meshkov dataset, our metacell consists of a subcube of $8 \times 8 \times 8$ neighboring cells, represented by a list of the scalar values appearing in a predefined order. Our indexing structure and isosurface query algorithm are designed upon the concept of metacell. With each metacell, we associate an interval $(v_{\min}, v_{\max})$ corresponding respectively to the minimum and maximum values of the scalar field over the metacell. Our compact interval tree structure makes use of the span space concept to organize the data layout. Before introducing this structure, we begin with a brief review of the standard binary interval tree.

### 4.2.1 Standard Interval Tree

Given a set of intervals, to build the binary interval tree, we store the median of the endpoints of the intervals at the root and assign all the intervals containing that value to the root. We then recursively build the left and right subtrees corresponding respectively to the intervals completely to the left and the right of the value stored at the root. More specifically, each node of the tree holds a splitting value $v_m$ and two secondary lists of the intervals $(v_{\min}, v_{\max})$ satisfying the condition $v_{\min} \leq v_m \leq v_{\max}$, one list in increasing order of $v_{\min}$ values and the second in decreasing $v_{\max}$ values.

The remaining intervals with $v_{\max} < v_m$ are assigned to the left subtree while the intervals with $v_m < v_{\min}$ are assigned to the right subtree. The tree is completed when all the intervals are captured and assigned to a node. Because two copies of a list of intervals are stored at each node, the size of the interval tree is at least twice as large as the size of original set of intervals.

### 4.2.2 Our Indexing Scheme

Our compact interval tree is similar to the interval tree except that we don't store the two sorted lists of intervals at each node. Instead, we store the distinct values of the $v_{\max}$ endpoints of these intervals, sorted in decreasing order, and associate with each such value a pointer to a list of intervals sorted in increasing order of left endpoint value $v_{\min}$. We now explain the compact interval tree in the context of the isosurface problem and its relationship to the metacells generated from the input data. Consider the span space consisting of all possible combinations of the $(v_{\min}, v_{\max})$ values of the scalar field. With each such pair we associate a list containing the metacells whose minimum scalar field value is $v_{\min}$ and whose maximum scalar field value is $v_{\max}$. The essence of the scheme for our compact interval tree is illustrated through Figure 4.1 representing the span space, and Figure 4.2 representing the compact interval tree built upon the $n$ distinct values of the endpoints of the intervals corresponding to the metacells.

Let $v_{m0}$ be the median of all the endpoints ranging from $v_0$ to $v_n$. The root of the interval tree corresponds to all the intervals whose $v_{\min}$ values fall in the range

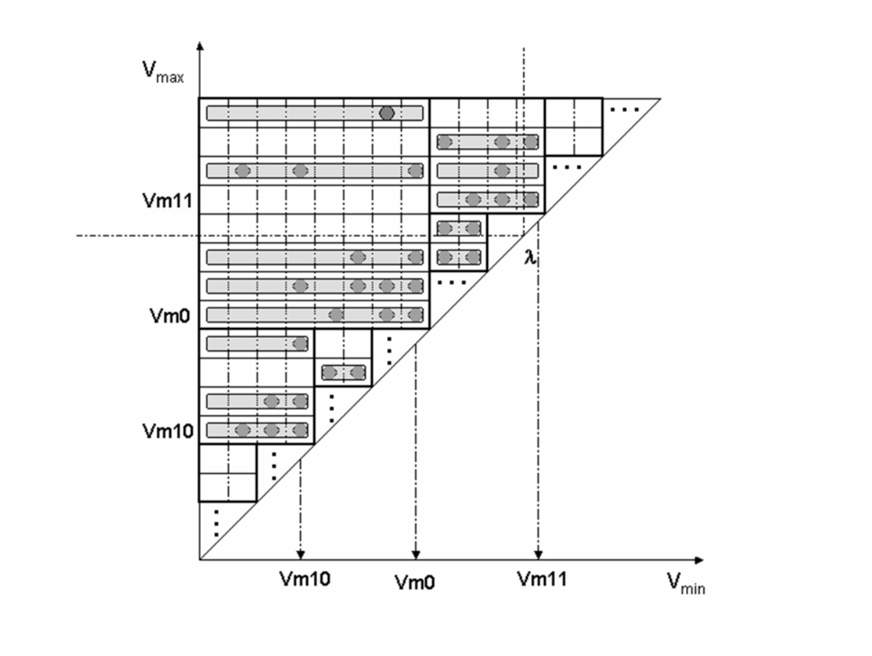

Figure 4.1: Span space partitioning scheme for our indexing structure.

$[v_0, \ldots, v_{m0}]$, and whose $v_{\max}$ values fall in the range $[v_{m0}, \ldots, v_n]$. Such intervals are represented as points in the square of Figure 4.1 whose bottom right corner is located at $(v_{m0}, v_{m0})$. We group together all the metacells having the same $v_{\max}$ value in this square, and store them consecutively on disk from left to right in increasing order of their $v_{\min}$ values. We refer to this contiguous arrangement of all the metacells having the same $v_{\max}$ value within a square as a *brick*. The bricks within the square are in turn stored consecutively on disk in decreasing order of the $v_{\max}$ values. The root will contain the value $v_{m0}$, the number of non-empty bricks in the corresponding square (of the span space), and an index list of the corresponding bricks. This index list consists of at most $n/2$ entries corresponding to the non-empty bricks, each entry containing three fields: the $v_{\max}$ value of the brick, the *smallest* $v_{\min}$ value of the metacells in the brick, and a pointer that indicates the start position of the brick

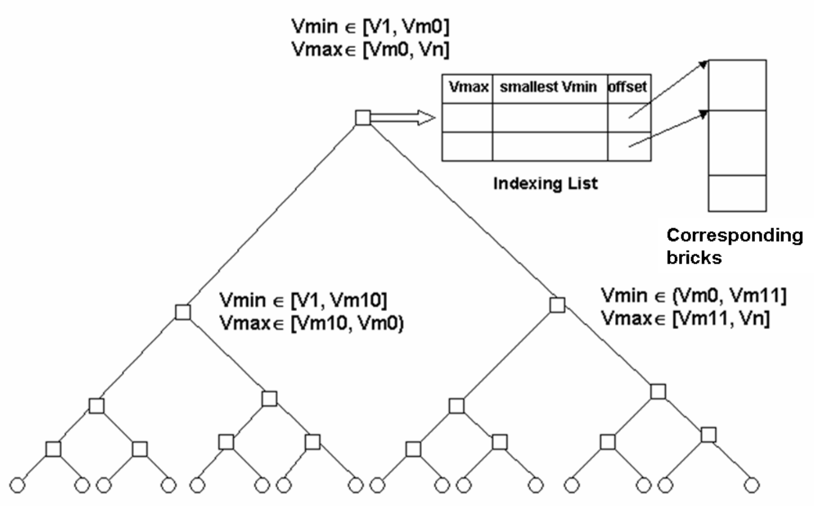

Figure 4.2: Compact interval tree structure and the associated metacell lists.

on the disk. Each brick contains contiguous metacells in increasing order of $v_{\min}$ values, and each metacell consists of its $v_{\min}$ value, its location information such as metacell ID, and a list of the scalar field values of the vertices (in a predefined order) within the metacell. We recursively repeat the process for the left and right children of the root. We will then obtain two smaller squares whose bottom right corners are located respectively at $(v_{m10}, v_{m10})$ and $(v_{m11}, v_{m11})$ in the span space, where $v_{m10}$ and $v_{m11}$ are the median values of the endpoints of the intervals associated respectively with the left and right subtrees of the root. In this case, each child will have at most $n/4$ non-empty index entries associated with its corresponding bricks on the disk. This recursive process is continued until all the intervals are exhausted. At this point we have captured all possible $(v_{\min}, v_{\max})$ pairs and their associated metacell lists.

### 4.2.3 Space Consumption Analysis

Note that the size of the standard interval tree is typically much larger than the size of our indexing structure. We can upper bound the size of our compact interval tree as follows. There are at most $n/2$ index entries at each level of the compact interval tree and the height of the tree is no more than $O(\log n)$. Hence our compact interval tree consists of $O(n \log n)$ index entries, each entry having three fields. Therefore the total size of our compact interval tree is $O(n \log n)$, while the size of the standard interval tree is $\Omega(N)$, where $N$ is the total number of intervals and hence can be as large as $\Omega(n^2)$. In fact, the standard interval tree is always at least twice as large as our indexing structure regardless of the relative values of $N$ and $n$ since the interval tree stores each interval $(v_{\min}, v_{\max})$ twice while our indexing structure stores each such interval at most once (the extra space required for pointers to the corresponding data on disk is the same in both cases).

In Table 4.1 below we compare the sizes of the two indexing structures for some well-known data sets from LLNL, the Stanford Volume Data Archive and IEEE visualization. While the Richtmyer-Meshkov data uses one-byte scalar fields, the MRBrain and CTHead datasets consist of two-byte scalar fields, and the remaining four data sets (generated from a hurricane simulation) use floating point scalar fields. As can be seen from the table, our indexing structure is substantially smaller than the standard interval tree, even in the case when $N \approx n$.

Table 4.1: Size comparison between standard and compact interval trees. $N$ denotes the number of distinct intervals and $n$ represents the number of distinct $v_{\max}$ values.

| DataSet Name | Scalar Field Size | $N$ | $n$ | Size of Interval Tree: Standard | Size of Interval Tree: Compact |
|---|---|---|---|---|---|
| RMI [a] | one bytes | 18,970 | 227 | 222.3 KB | 6.0 KB |
| MRBrain [b] | two bytes | 756,982 | 2,894 | 11.6 MB | 22.6 KB |
| CTHead [b] | two bytes | 817,642 | 3,238 | 12.5 MB | 25.3 KB |
| Pressure [c] | four bytes | 24,507,104 | 20,748,433 | 560.9 MB | 237.4 MB |
| Velocity [c] | four bytes | 24,444,597 | 17,548,131 | 559.5 MB | 200.8 MB |

[a]Richtmyer-Meshkov Instability dataset from http://www.llnl.gov/CASC/asciturb/
[b]3D CT images from http://graphics.stanford.edu/data/voldata/
[c]Simulation data of a Hurricane from http://vis.computer.org/vis2004contest/data.html

## 4.3 Efficient Isosurface Extraction Method

Given a query isovalue $\lambda$, consider the unique path from the leaf node labeled with the largest value $\leq \lambda$ to the root. Each internal node on this path contains an index list with pointers to some bricks. For each such node, two cases can happen depending on whether $\lambda$ belongs to the right or left subtree of the node.

*Case 1*: $\lambda$ falls within the range covered by the node's right subtree. In this case, the active metacells associated with this node can be retrieved from the disk sequentially starting from the first brick until we reach the brick with the smallest value $v_{\max}$ larger than $\lambda$.

*Case 2*: $\lambda$ falls within the range covered by the node's left subtree. The active metacells are those whose $v_{\min}$ values satisfy $v_{\min} \leq \lambda$, from each of the bricks on the index list of the node. These metacells can be retrieved from the disk starting from the first metacell on each brick until a metacell is encountered with a $v_{\min} > \lambda$. Note that since each entry of the index list contains the smallest $v_{\min}$ of the corresponding brick, no I/O access will be performed if the brick contains no active metacells.

Once an active metacell is in memory, any of the several variations of the Marching Cubes algorithm can be used to precisely determine the active cells within the metacell and generate the appropriate triangles defining the isosurface mesh.

## 4.4 Parallel Processing Scheme

Assume that we have $p$ processors, each with its own local disk, and the processors are interconnected with some type of a high-speed interconnection network. We will show how to partition the input data among the $p$ local disks and apply our compact tree indexing structure to extract and render isosurfaces in a scalable and efficient way. The main challenge is to ensure load balancing among the processors for any isovalue while maintaining the same total amount of work as that of our sequential algorithm.

### 4.4.1 Span Space Partitioning

We start by showing how to distribute the $N$ input metacells among the local disks in such a way that the active metacells corresponding to any isovalue are spread almost evenly among the processors. We first sort the $N$ metacells in decreasing order of their $v_{max}$ values. The $N$ sorted metacells are then divided evenly into $\sqrt{N/p}$ sets, each set containing $\sqrt{pN}$ consecutive metacells in the sorted order. Then, we re-sort the metacells within each set by increasing order of $v_{min}$ values.

We now stripe all the $N$ metacells as they appear after the two sorting steps across the $p$ disks. That is, the first metacell is stored on the disk of the first processor, the second on the disk of the second processor, and so on wrapping around as necessary. We next prove the following property of our partitioning scheme.

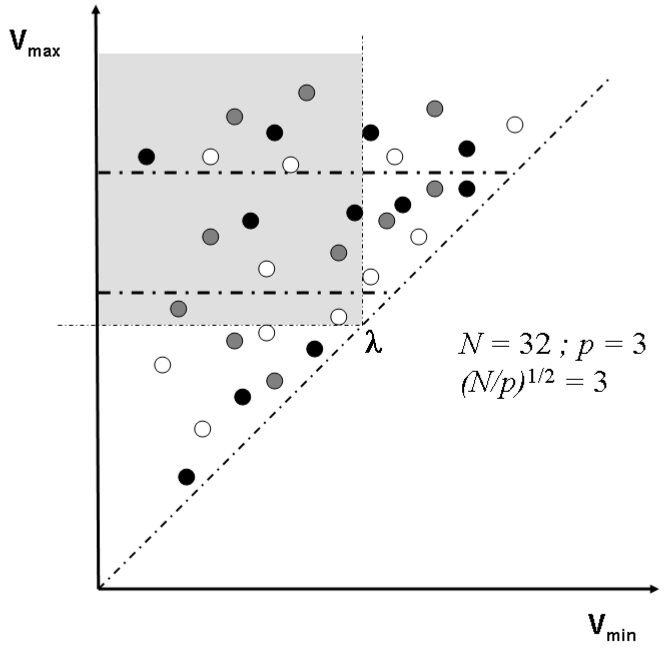

Figure 4.3: Each pair $(v_{\min}, v_{\max})$ is illustrated as a point and assumed in this case to correspond to a single metacell. For $N = 32$ and $p = 3$, the metacells are partitioned into 3 sets separated by dashed lines. The colors black, white, and grey are used to denote the metacells belonging to the first, second, and third processor respectively.

## 4.4.2 Provable Load Balancing

**Lemma**: Our metacell partitioning algorithm distributes the $N$ metacells of the initial dataset onto $p$ processors such that each processor receives $N/p + O(1)$ metacells. For any isovalue $\lambda$, each processor will hold at most $N^*/p+2\sqrt{N/p}$ active metacells, where $N^*$ is the total number of active metacells corresponding to the isovalue $\lambda$.

**Proof**: The first part of the lemma is obvious since the $N$ metacells, after getting rearranged in the two sorting steps, are striped across the $p$ disks. Given an isovalue $\lambda$, the corresponding active metacells are those represented in the shaded region shown in Figure 4.3 of the span space. We now examine how these metacells are distributed among the $p$ processors. After the first sorting step, our partitioning algorithm distributes the $N$ metacells evenly into $\sqrt{N/p}$ sets, each of which contributes $\sqrt{pN}$ metacells. The second sorting step is performed within each set separately, after which all the metacells are striped across the $p$ disks. Each set whose $v_{\max}$ values are larger than or equal to $\lambda$ (the first two sets in Figure 4.3) contributes an equal number of active metacells with a difference of at most one for each set since the metacells within each set were sorted in increasing order of $v_{\min}$. Since there are at most $\sqrt{N/p}$ such sets, the total difference of active metacells contributed by these sets is at most $\sqrt{N/p}$. We may have an additional set some of whose $v_{\max}$ values are larger than or equal to $\lambda$, and the rest are strictly smaller (the last set in Figure 4.3). In this case, the maximum difference in the number of active metacells among all the $p$ processors can not possibly exceed $\sqrt{N/p}$, which is the number of metacells placed at each processor from this set. Therefore the maximum difference in the number of active metacells among all processors is bounded by $2\sqrt{N/p}$, and the proof of the lemma is complete.

A few observations about our partitioning algorithm are in order. The first is that the above upper bound on the number of active metacells per processor is extremely conservative. As we will illustrate later, the bounds achieved in practice across widely different data sets are substantially better than this upper bound. The second observation is that in general we expect $N^*$ to be of order $N^{2/3}$ for interesting isovalues in which case $2\sqrt{N/p}$ is asymptotically much smaller than $N^*/p$. Third, our scheme is the only scheme, as far as the authors know, for which asymptotic load balancing bounds can actually be established.

### 4.4.3 Sort-last Composition

Once the metacells are distributed among the $p$ processors, we create a compact interval tree that corresponds to the local data set. The isosurface query can now be carried out simultaneously by all the $p$ processors using their own local indexing lists. As a result, roughly the same number of triangles are generated by each processor, which are then rendered locally. The $p$ frame buffers will then be composited in a binary-swap way [15] using their depth information to create the final output. Except for the very last step, we have provably split the work asymptotically equally among the processors, without increasing the total work relative to the sequential algorithm and without requiring communication between the processors (such a strategy is referred to as *sort-last* strategy in the literature). For large scale datasets such as the Richtmyer-Meshkov dataset whose isosurfaces consist of hundreds of millions of triangles, the compositing step involves the movement of data that is orders of magnitude smaller than the total size of the triangles, and hence can be done extremely quickly given a high-speed interconnection network as will be illustrated later.

## 4.5 Extension to Time Varying Data

Given the extremely compact size of our indexing structure, this scheme can be easily extended to deal with large scale time-varying data as follows. We have shown that the size of our indexing structure is $O(n \log n)$ for a single time step during which there are $n$ distinct values of the endpoints of the intervals corresponding to the metacells. To index time-varying data of $m$ time steps, we can use the same indexing scheme for each time step separately resulting in an indexing structure of size $O(mn \log n)$. Note that the size of the indexing structure depends only on the number of time steps, which is typically small, say in the order of hundreds and rarely in the thousands, but independent of the total number of cells of the given dataset. For example, one-byte scalar data with hundreds of time steps will require an indexing structure of size at most a few megabytes, regardless of the size of each grid since the number $n$ of possible distinct values is $2^8 = 256$. Similarly for two-byte scalar data, the size of the indexing structure increases to hundreds of Megabytes, which is still reasonable and can easily fit in today's processors' main memory. In the case of Richtmyer-Meshkov data set, we have 270 time steps with 7.5GB per time step, which amounts to a total of about 2.1TB. However the size of our indexing structure for the whole data set is only 3.2MB.

## 4.6 UM Visualization Cluster

The testing platform consists of a 16-node visualization cluster, each node consists of a 2-way SMP Dual-CPU running at 3.0 GHz, an 8GB main memory, a

60GB local disk that can achieve 70 MB/sec I/O transfer rate, and one NVIDIA6800 GPU card with bi-directional 4GBps data transfer rate to memory via PCI-Express (×16) Bus. The GPU communicates with CPU and RAM via the MCH(Memory Controller Hub). These 16 Nodes are inter-connected through 10Gbps Topspin InfiniBand network. In addition, four nodes are connected to four projectors for a four-way tiled wall-sized display via their GPU card's DVI port. The architecture of a single node is illustrated in Figure 4.4.

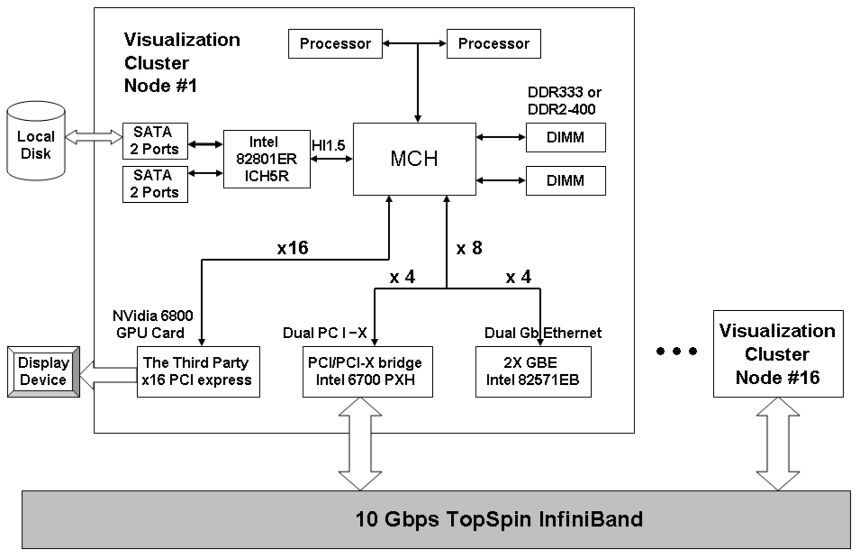

Figure 4.4: The visualization cluster architectural diagram.

As a visualization cluster, each node of the system can run graphics programs and dispatch OpenGL commands to its GPU for rendering. The system software configuration includes Redhat Linux Enterprise 3.0, MPI, and the Chromium package to enable the parallel rendering among multiple rendering nodes. Basically, the Chromium is able to intercept OpenGL command calls from graphics applications and sends them to proper rendering servers according to the tiled-display layout[3] [12]. For the parallel rendering scheme, we use the *sort-last method* [17]. The essence of this method is to have each node render its triangles locally using the on-board GPU, after which the output is read back from the GPU's frame buffer and sorted according to the display server's tile layout. Different regions of the frame buffer including the z-buffer content are forwarded to the appropriate rendering servers, each of which will be responsible for displaying a specific region on the wall-sized display. At each rendering server, the components of the frame buffers from various processors are composited using their z-buffer contents and rendered to the display device connected to server's GPU. In our experiments, the time of sorting and shuffling the frame buffers among various nodes via 10Gbps InfiniBand doesn't cause a noticeable overhead compared to time it takes to extract and render the triangles at each node as we will see later.

## 4.7 Experiments and Performance Analysis

We tested the performance of our parallel scheme intensively on the Richtmyer-Meshkov dataset because this is one of the most challenging datasets in terms of both size scale and data complexity. This dataset consists of $2048 \times 2048 \times 1920$ one-byte scalar values with very high complexity for each time step and spans 270 time steps. The data amounts to 7.5GB for each time step for a total of 2.1TB.

[3]http://chromium.sourceforge.net/

### 4.7.1 Preprocessing

During the data preprocessing stage, we scan the data once and create the metacells, where each metacell consists of a four-byte ID indicating the location of the metacell, $9 \times 9 \times 9$ one-byte scalar values of the vertices, for a subcube of $8 \times 8 \times 8$ cells and the minimum values of the metacell vertices. At this point, the original data has been converted to $256 \times 256 \times 240$ metacells, each of length 734 bytes. Using our scheme, we stripe the metacells among the available disks and build each local indexing structure separately on each node. In particular, each node of the visualization cluster will hold a compact interval tree with pointers to the bricks stored on its local disk. For a single time step, this preprocessing takes about 30 minutes to complete on a single node of our cluster.

### 4.7.2 Load Balancing

Since the performance of our algorithm depends critically on how the overall computational load is allocated to the different processors, we start by illustrating the load balancing achieved on a 16-processor cluster using seven different datasets, including the five datasets listed in Table 4.1. The average number of active metacells among all the processors as well as the maximum difference between the loads of any two processors are shown in Tables 4.2 and 4.3, over a wide range of isovalues. As the results show, the maximum difference of the number of active metacells among all the $p$ processors is substantially smaller than the established upper bound of $2\sqrt{N/p}$, for all cases, and clearly illustrate the excellent load balancing achieved

Table 4.2: Measured load imbalance in number of active metacells among sixteen nodes with varying isovalues for one-byte and two-byte datasets. $N$ denotes the number of intervals and $p$ is the number of processors, which is equal to 16. Ave. $N^\star/p$ represents average number of active metacells per processor.

| LLNL Dataset | | | MRBrain Dataset | | | CTHead Dataset | | |
|---|---|---|---|---|---|---|---|---|
| One-byte scalar $N = 5,629,653$ $2(N/p)^{1/2} = 1,186$ | | | Two-byte scalar $N = 6,134,838$ $2(N/p)^{1/2} = 1,238$ | | | Two-byte scalar $N = 5,799,589$ $2(N/p)^{1/2} = 1,204$ | | |
| Iso-Value | Ave. $N^\star/p$ | Max Diff | Iso-Value | Ave. $N^\star/p$ | Max Diff | Iso-Value | Ave. $N^\star/p$ | Max Diff |
| 10 | 90,986 | 33 | 1,253 | 24,657 | 108 | 0 | 12,196 | 36 |
| 30 | 129,505 | 28 | 1,480 | 28,797 | 72 | 272 | 30,149 | 24 |
| 50 | 165,931 | 26 | 1,707 | 36,185 | 61 | 544 | 18,255 | 42 |
| 70 | 177,806 | 18 | 1,934 | 41,931 | 52 | 816 | 14,696 | 27 |
| 90 | 150,728 | 18 | 2,161 | 29,164 | 38 | 1,088 | 45,802 | 9 |
| 110 | 110,231 | 14 | 2,388 | 17,132 | 44 | 1,360 | 17,683 | 34 |
| 130 | 81,769 | 11 | 2,615 | 7,284 | 36 | 1,632 | 15,536 | 27 |
| 150 | 64,839 | 5 | 2,842 | 5,425 | 42 | 1,904 | 14,161 | 61 |
| 170 | 54,411 | 6 | 3,069 | 3,903 | 38 | 2,176 | 12,258 | 22 |
| 190 | 47,232 | 7 | 3,296 | 2,052 | 18 | 2,448 | 6,055 | 44 |
| 210 | 41,658 | 12 | 3,523 | 522 | 32 | 2,720 | 1,215 | 36 |

in practice by our scheme. Since the total work of our parallel scheme is about the same as the sequential algorithm, we expect the performance of our parallel algorithm to be linearly scalable as we show next in some detail for the Richtmyer-Meshkov dataset.

### 4.7.3 Overall Performance and Scalability

We consider time step 250 of the Richtmyer-Meshkov dataset. After preprocessing the dataset, we obtain $5,592,802$ metacells that occupy a space of size

Table 4.3: Measured load imbalance in number of active metacells among sixteen nodes with varying isovalues for four-byte floating-point datasets corresponding to a hurricane simulation. $N$ denotes the number of intervals and $p$ is the number of processors, which is equal to 16. Ave. $N^{\star}/p$ represents average number of active metacells per processor.

| Four-byte floating-point datasets: $N = 24,651,099$; $2(N/p)^{1/2} = 2,482$ | | | | | |
|---|---|---|---|---|---|
| Pressure Dataset | | | Velocity Dataset | | |
| IsoValue | Ave. $N^{\star}/p$ | Max Diff | IsoValue | Ave. $N^{\star}/p$ | Max Diff |
| -3272.59 | 83 | 9 | -60.44 | 65 | 9 |
| -2763.30 | 214 | 16 | -49.66 | 410 | 20 |
| -2254.02 | 410 | 30 | -38.88 | 1,315 | 38 |
| -1744.74 | 633 | 32 | -28.10 | 2,602 | 53 |
| -1235.46 | 1,020 | 12 | -17.32 | 6,135 | 75 |
| -726.18 | 1,678 | 34 | -6.55 | 46,495 | 55 |
| -216.90 | 3,095 | 39 | 4.23 | 49,948 | 70 |
| 292.38 | 22,879 | 32 | 15.01 | 15,791 | 41 |
| 801.66 | 19,730 | 65 | 25.79 | 5,308 | 27 |
| 1310.94 | 15,003 | 49 | 36.57 | 1,087 | 39 |
| 1820.22 | 9,334 | 45 | 47.35 | 198 | 35 |

3.828GB, which is nearly 50% smaller than the original 7.5GB size since we eliminate the metacells for which all the vertices have the same scalar field value. We vary the isovalues from 10 up to 210, in steps of 20. For each of these isovalues, we ran the algorithm on one, two, four, eight, and sixteen nodes. We evaluate the performance of our isosurface extraction algorithm according to the following three metrics:

(i) the I/O time it takes to retrieve the active metacells from the disk, referred to as Active MetaCell(AMC) Retrieval time;

(ii) the amount of CPU time required to go through the active metacells and generate the appropriate triangles, referred to as Triangulation time;

(iii) finally the rendering time, which reflects the time it takes to render the triangles on the local GPU, after which the different frame buffers are composited to generate the final display. The time it takes to do the compositing of the very last step is included in the total time.

Table 4.4: Performance summary of our algorithm on a single node. AMC refers to the active metacells. The last column shows that the performance is linear in the number of triangles and hence the algorithm is output sensitive.

| Iso-value | Number of Triangles | Number of AMC | AMC Retrieval (sec) | Triang-ulation (sec) | Rend-ering (sec) | Total Time (sec) | Overall Rate (sec) |
|---|---|---|---|---|---|---|---|
| 10 | 228,770,844 | 1,455,782 | 15.49 | 36.2 | 19.00 | 70.74 | 3.23 |
| 30 | 376,578,332 | 2,072,085 | 21.52 | 57.9 | 21.50 | 100.98 | 3.73 |
| 50 | 569,387,336 | 2,654,902 | 25.74 | 85.41 | 32.65 | 143.85 | 3.96 |
| 70 | 651,834,482 | 2,844,889 | 25.89 | 96.31 | 37.39 | 159.65 | 4.08 |
| 90 | 511,136,810 | 2,411,647 | 21.96 | 76.09 | 29.26 | 127.35 | 4.01 |
| 110 | 329,408,766 | 1,763,701 | 17.13 | 50.45 | 19.09 | 86.72 | 3.8 |
| 130 | 229,201,420 | 1,308,299 | 13.30 | 35.25 | 13.28 | 61.89 | 3.7 |
| 150 | 177,035,314 | 1,037,426 | 10.71 | 27.18 | 10.29 | 48.23 | 3.67 |
| 170 | 146,369,438 | 870,573 | 9.12 | 22.38 | 8.51 | 40.05 | 3.66 |
| 190 | 125,365,482 | 755,710 | 7.99 | 19.09 | 7.27 | 34.41 | 3.64 |
| 210 | 108,977,638 | 666,527 | 7.07 | 16.5 | 6.32 | 29.94 | 3.64 |

We first consider the performance of our algorithm on a single processor. From Table 4.4, we can see that the number of generated triangles varies from 100 million to 650 million over the range of isovalues from 10 to 210. Our indexing structure is of size 6KB, which is quite small compared to the size of the data. As shown in Table 4, we are able to achieve the I/O rate of about 70MB/s in retrieving the active metacells, with a linear relationship between the total time and the number of triangles generated. It is also shown that the triangle generation stage is the bottleneck for the whole isosurface extraction as we need to go through each of the active unit cells within an active metacell to generate the triangles as necessary. Once the triangles are generated, they are rendered on the GPU very quickly. As a result we were able to extract and render isosurfaces at the rate of almost 4 million triangles per second.

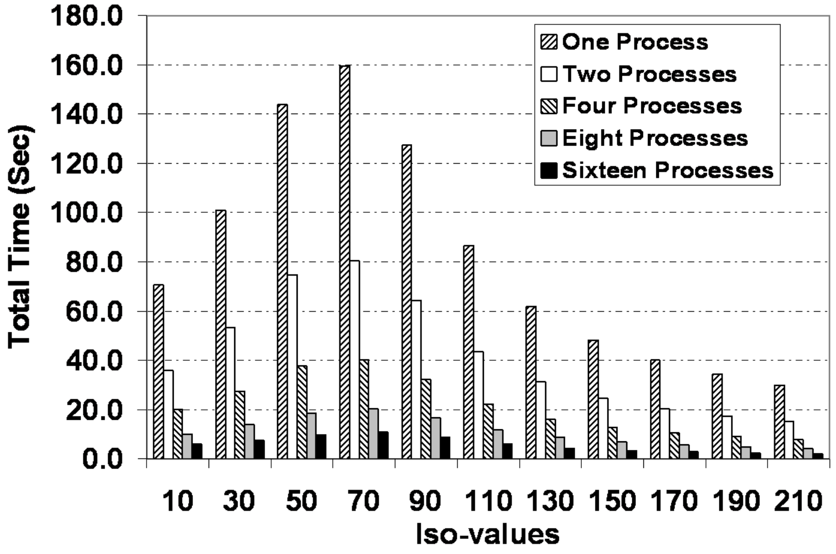

Figure 4.5: Overall execution time of up to sixteen processors over a range of iso-values.

Table A.1 through Table A.4 (in the Appendix A) show the execution times of the major steps of our algorithm on 2, 4, 8, and 16 processors over a wide range of isovalues. A careful examination of the experimental results in these tables clearly illustrate the scalability and the efficiency of our scheme. Note also that the compositing step (whose time is equal to the total time minus the sum of the times of the other steps) is extremely fast and takes at most a few hundreds of milliseconds even on 16 processors.

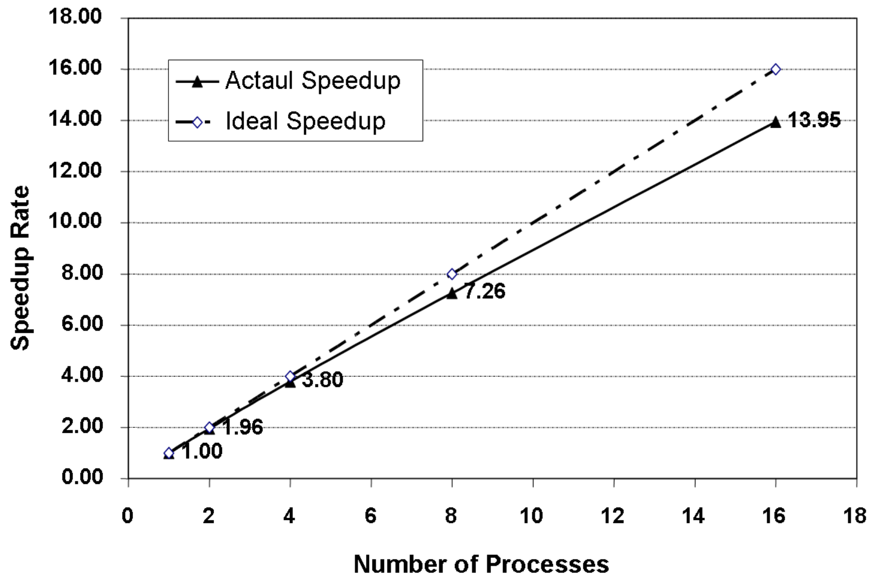

Figure 4.6: Corresponding speedups of up to sixteen processors over a range of isovalues.

The overall time spent on the extraction and rendering of isosurfaces for various isovalues is illustrated in Figure 4.5. The corresponding speedups are highlighted in Figure 4.6. As expected, our scheme achieves very good scalability relative to our extremely efficient serial algorithm, independent of the particular isovalue.

We now consider the more general case of time-varying datasets that are to be explored by extracting and rendering isosurfaces corresponding to a time step and an isovalue. We can index the 270 time steps of the Richtmyer-Meshkov dataset using our indexing scheme. The size of the resulting indexing structure is 3.2MB, which easily fits into the main memory of a node. The layout of the data of each time step will be distributed across the processors as before. Extracting an isosurface of a time step amounts to determining the appropriate indexing structure for that time step, which can easily be performed since the whole indexing structure is in main memory. Table 4.5 shows the results for time steps 180 through 195 for the isovalue of 70 on four processors. Each row of the table lists the number of active metacells, the number of triangles generated, the execution time on a four-node configuration, and the overall rate of triangles rendered (millions per second).

Table 4.5: Overall performance of four processors on Isovalue 70 over 16 time steps

| Time-Step | Num of Active Meta Cells | Number of Triangles | Overall Time (Sec) | Overall Rate ($\times 10^6/s$) |
|---|---|---|---|---|
| 180 | 2,249,247 | 499,936,480 | 35.971 | 13.898 |
| 181 | 2,259,741 | 502,356,768 | 32.934 | 15.253 |
| 182 | 2,269,996 | 504,717,568 | 32.717 | 15.427 |
| 183 | 2,280,249 | 507,140,192 | 33.349 | 15.207 |
| 184 | 2,290,438 | 509,504,96 | 33.145 | 15.372 |
| 185 | 2,300,808 | 511,877,56 | 33.628 | 15.222 |
| 186 | 2,310,642 | 514,222,240 | 34.121 | 15.071 |
| 187 | 2,320,869 | 516,675,168 | 34.524 | 14.966 |
| 188 | 2,330,322 | 519,26,80 | 33.732 | 15.387 |
| 189 | 2,340,363 | 521,496,928 | 34.3 | 15.204 |
| 190 | 2,295,699 | 516,444,992 | 31.405 | 16.445 |
| 191 | 2,360,385 | 526,339,520 | 35.163 | 14.969 |
| 192 | 2,370,458 | 528,728,448 | 34.523 | 15.315 |
| 193 | 2,380,148 | 531,149,920 | 35.256 | 15.066 |
| 194 | 2,389,433 | 533,600,192 | 35.172 | 15.171 |
| 195 | 2,399,116 | 536,50,304 | 35.891 | 14.936 |

## 4.8 Summary

In this chapter, a new indexing structure and a new partitioning algorithm are presented for out-of-core isosurface extraction and rendering of large scale data. The indexing scheme is based on a compact version of the interval tree that makes use of the span space concept whose size is $O(n \log n)$ compared to $\Omega(N)$ for the standard interval tree, where $N$ is the number of all possible pairs of scalar field values appearing in metacells and $n$ is the number of their distinct endpoints. The data is arranged in a compact layout on the disk, which enables very efficient I/O performance. We have shown that the new algorithm can easily be adapted to a multiprocessor environment, provably delivering efficient and scalable performance. The algorithm was tested extensively on a wide variety of datasets, and was shown in detail to achieve scalable performance on the Richtmyer-Meshkov dataset over different processor configurations and different isovalues.

# Chapter 5

## Hybrid Ray-casting for Isosurface Rendering

A major drawback of polygonal-based isosurface generation is the need to extract a potentially very large triangular mesh for each specific isovalue, a large part of which, however, 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, as shown in the previous chapter, it consists of hundreds of millions of triangles in the Richtmyer-Meshkov instability dataset from LLNL. One approach 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 [35], 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 achieve interactive rendering, researchers have resorted to parallel algorithms such as those that appeared in [31, 26]. A recent method based on an elaborate data structure for a persistent octree was described in [40]. On the other hand, we should note that the quality of the generated isosurfaces using triangular meshes may not be of high quality especially for complex regions as the triangular mesh is just a polygonal approximation of the surface.

## 5.1 Features of Existing Ray-casting Algorithms

An alternative approach for generating isosurfaces is using ray-casting methodology. Such an approach was first proposed in [19], using a brute-force ray-casting on the SGI Reality Monster based upon a shared memory multiprocessor. A distributed memory version was described in [29], and schemes based on kd-trees and octree are described in [30, 42, 33]. In general, there are several main features that have been exploited in the literature to speed up the computation of isosurfaces through ray-casting listed as follow:

- 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-casting 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 $2 \times 2$ adjacent pixels) as the unit traversal through the volumetric dataset.

Except for applying the above techniques in different ways, all the known isosurface ray-casting algorithms follow more or less the same basic strategy. In this part of work, we introduce a new strategy to greatly improve ray-casting for isosurface rendering. This method is a combination of object-order projection of a coarse version of the data and a very efficient ray-casting restricted to a few data blocks for each packet of rays (corresponding to adjacent pixels). For clarity, we start by presenting the basic version for single ray, which will be extended to exploit the idea of packets of rays afterwards.

## 5.2 Basic Strategy

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 is found. This is done through a trilinear interpolation process and followed by the shading of the corresponding pixel under the diffuse shading model (which is widely used in scientific visualization).

### 5.2.1 Compact BONO-Tree Construction

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) [23] 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.

### 5.2.2 Phase I: Bounded Block List Generation

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 the value $k$ is rather small for 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.

### 5.2.3 Phase II: Spatial-Aware Ray-Casting

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, the ray goes through active cells one by one within a block via the front-to-back octree traversal. When the ray encounters an active cell, we perform a trilinear interpolation upon the active cell for the isosurface intersection, which is followed by (diffuse) shading if such intersection exists. The intersection interpolation is computed accurately using the method described in [36] while the normals are computed using forward difference. If the ray reaches the end of the list without finding such intersection (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, it 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.

## 5.3 Improvements on Basic Strategy

We now consider some acceleration techniques for the basic strategy, which exploit the coherency between nearby pixels and their associated block lists.

### 5.3.1 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 $2 \times 2$ as used in our experimental results) into a single list. This is somewhat similar in spirit to the use of packet of rays in [42, 34]. 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 [33, 34]. 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.

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 $2 \times 2$) 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

Table 5.1: List generation time on single-core for Far and Close views with various upper bound $k$ using single pixels (resulting in $1024 \times 1024$ lists) and groups of $2 \times 2$ adjacent pixels (resulting in $512 \times 512$ 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.

| | List upper bound $k$ | | Time (msec) | | Ratio |
|---|---|---|---|---|---|
| # of lists | $1024^2$ | $512^2$ | $1024^2$ | $512^2$ | |
| 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 |

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 5.1, 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 $2 \times 2$ 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 22, compared to 12 in the single pixel case, and therefore the overhead will be minimal.

### 5.3.2 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 RMI dataset to handle the rendering of far views, which results in $6 \times 6$ 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 $2 \times 2$ 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. $12 \times 12$ on average when we use $4 \times 4 \times 4$ blocks for a $16:1$ zoom-in close view). Figure 5.1 illustrates the performance of each of Phase I and Phase II on the RMI dataset as a function of the block size and the viewpoint.

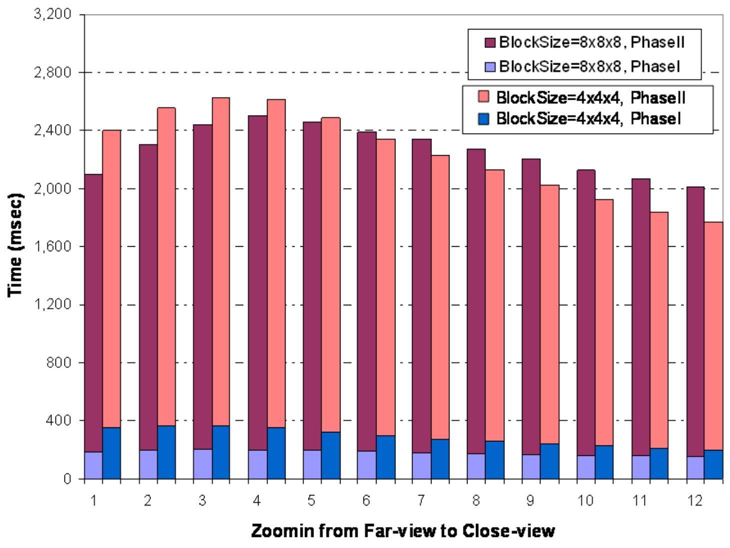

Figure 5.1: Execution time of list generation(PhaseI) and ray-casting(PhaseII) on a single-core as we vary the zoom-in from far-view to close-view for block sizes $8^3$ and $4^3$.

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.

With the adaptation of the block size from $8 \times 8 \times 8$ to $4 \times 4 \times 4$, our tests show an overall speedup of around $10 \sim 20\%$.

## 5.4 Parallel Processing Scheme

As 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. 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. Under this perspective, 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.

The multi-core architecture presents opportunities for speeding up demanding computations if the available resources can be effectively used. Typically a multi-threaded program is used to specify the tasks executed on the cores and the coordination involved among these tasks. In general, assume there are $p$ cores on a multi-core processor, with some local (possibly shared) cache or memory available for each core. Using $p$ threads, each phase in the serial version of our algorithm is parallelized as follows.

### 5.4.1 Block List Generation

To handle Phase I, the screen is divided into almost equal contiguous regions. Each thread traverses the BONO tree and creates the lists of blocks corresponding to its groups of pixels within the region. 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 $k$.

### 5.4.2 Z-Order Screen Partitioning

We then partition the screen and group the ray-casting tasks through all the lists as follows.

i. Partition the screen into small image-size tiles (for example $8 \times 8$ or $16 \times 16$) and order these tiles using a Z-order (or a space-filling curve such as Hilbert space filling curve). Such an 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.

ii. After the lists are generated, 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.

iii. 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 divided 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 divided 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 a 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.

### 5.4.3 Dynamic Task Scheduling of Ray-Casting

We perform ray-casting in Phase II dynamically as follows. Initially, each thread will grab a group of image tiles from the ordered list $L_I$ created through screen partitioning. A thread will then process its group by shooting rays through the pixels in the group using the generated data block lists. 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.

## 5.5 The Clovertown Multi-Core Processors

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 4 MB, and hence the total L2 cache available is 8 MB. Its architectural diagram is shown in Figure 5.2. Our Clovertown platform has 8 GB of main memory, which constitutes an upper bound on the size of the datasets used in our experiments.

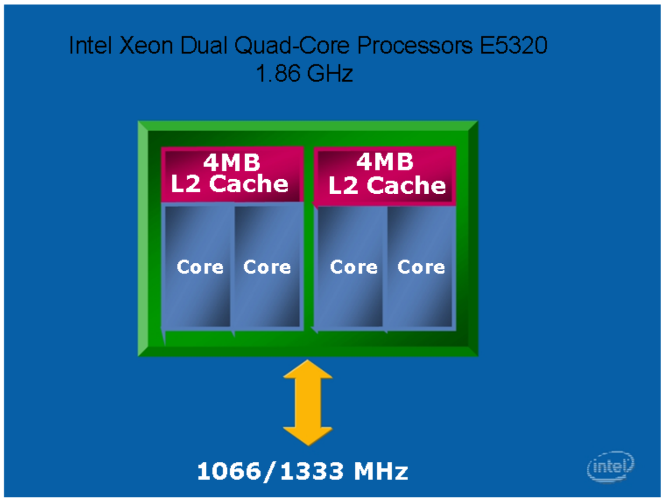

Figure 5.2: Intel Quad-Core Processor Architectural Diagram.

## 5.6 Experiments and Performance Analysis

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 RMI dataset from LLNL. 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 tri-linear interpolation equation as in [36], 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 in the 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 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 octree traversal algorithm is based on a dynamic allocation of static small screen tiles ($8 \times 8$ 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 octree traversal algorithm is rather similar to that achieved by the octree algorithm reported in [33].

### 5.6.1 Datasets Used and Space Consumption

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 6.2). 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 6.1 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

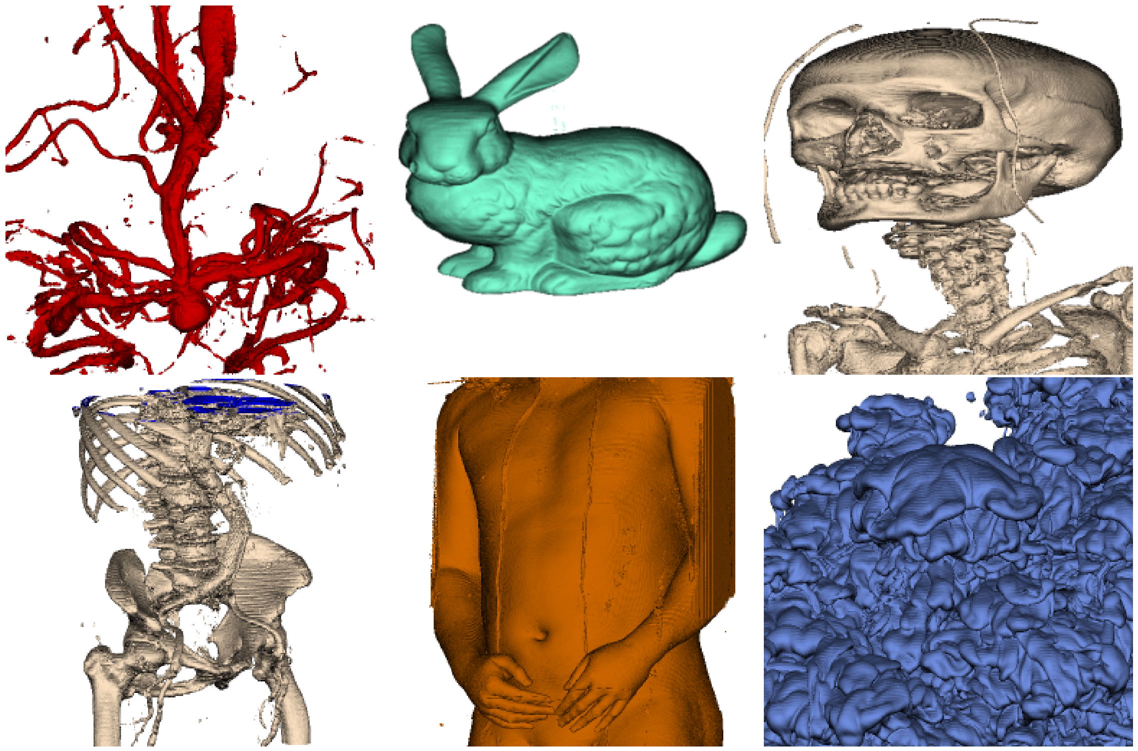

Figure 5.3: The six datasets used in our experiments. From left to right, top to down, the datasets are: Aneurism, Bunny, Skull, Abdominal, VisMale and RMI respectively.

is around 46MB for the RMI 8GB dataset. On the other hand, the accumulated size of the finer indexing structures (that is, octrees) for all the data blocks is just a fraction of original dataset, no more than $\frac{1}{4}$ as shown in Table 5.3. Note that the total sizes of our indexing structures are substantially smaller than those used by the kd-tree algorithms (such as [42, 30]).

Due to the simplicity of our indexing structure, the preprocessing times are also much better than any of the published preprocessing times even when the previous algorithms are run on faster processors. We note that the compressed octree reported in [33] consolidates regions with uniform values into a single octree node, and hence can result in significant space savings especially for the case when

Table 5.2: Parameters of various datasets used

| 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 |
| RMI | 1 bytes | $2048^2 \times 1920$ | 7.5 GB | $8^3$ | 5,655 K |

Table 5.3: Size of our indexing structure for Blocks and BONO tree along with their preprocessing time

| 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 % |
| RMI | 809.08 | 46.79 | 520.0 | 1.950 | 10.53 % |

the spatial variability is relatively small. In fact a compression of up to 25% is reported for the RMI dataset. Since in our case the indexing structure is separate than the data itself, we can make use of this technique to compress the volumetric data as well.

### 5.6.2 Performance Implication on List Upper Bound

In addition to our new strategy that combines object order traversal followed by ray-casting, 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;

(iv) the time needed for ray-casting of the unfinished pixels (that is, those pixels whose lists were of size $k$ with no intersection 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 RMI 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 octree ray-casting is the same as the case when $k = 0$). The corresponding results are illustrated in Figures 5.4.

From these results, we can make the following observations. First, the octree 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 RMI data is high ($\sim 50$) 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) takes an increasingly larger

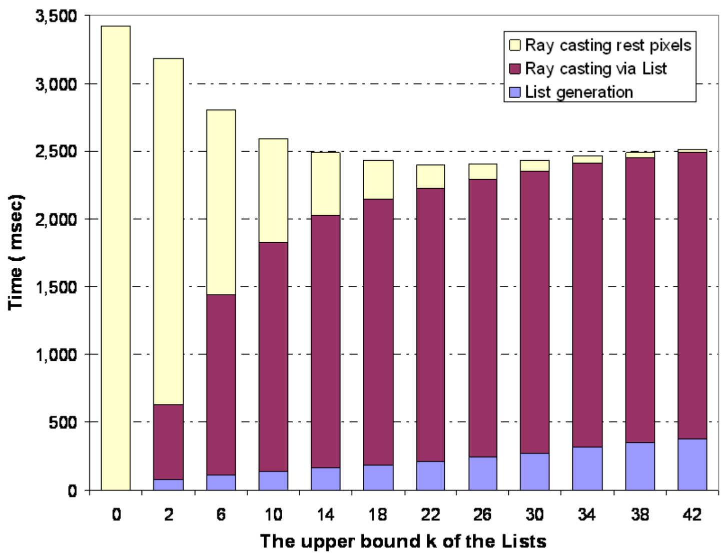

Figure 5.4: Execution times of the different stages of our algorithm on a single core vs. the value of upper bound $k$. The results are for the Far-view of the RMI dataset of time step 250 using $1024^2$ screen resolution.

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 can be verified more clearly by examining the curve shown in Figure 5.5. 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 RMI 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 existing strategy. The traversal of the BONO tree effectively identifies the area on the screen

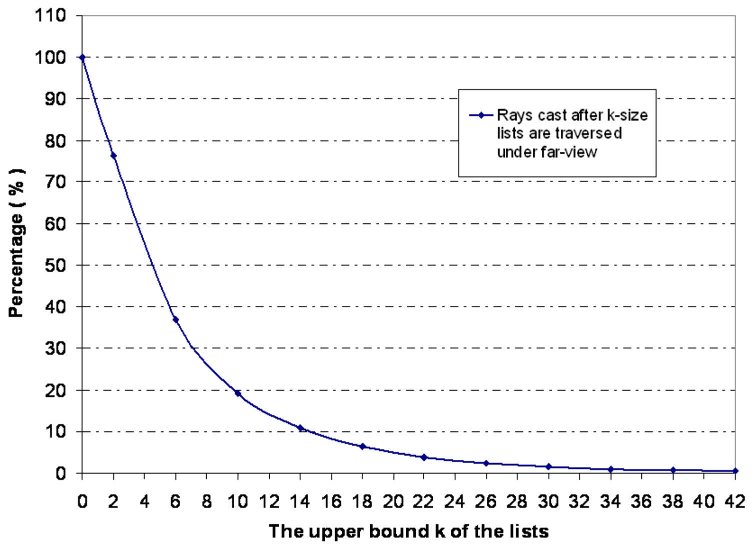

Figure 5.5: Percentage of pixels left to shade after going through $k$ blocks from the lists for the $2048^2 \times 1920$ RMI dataset at time step 250. Screen size is $1024^2$.

where the isosurface is mapped, passing this information for ray-casting through the block lists. Figure 5.6 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 5.6, 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 octree 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

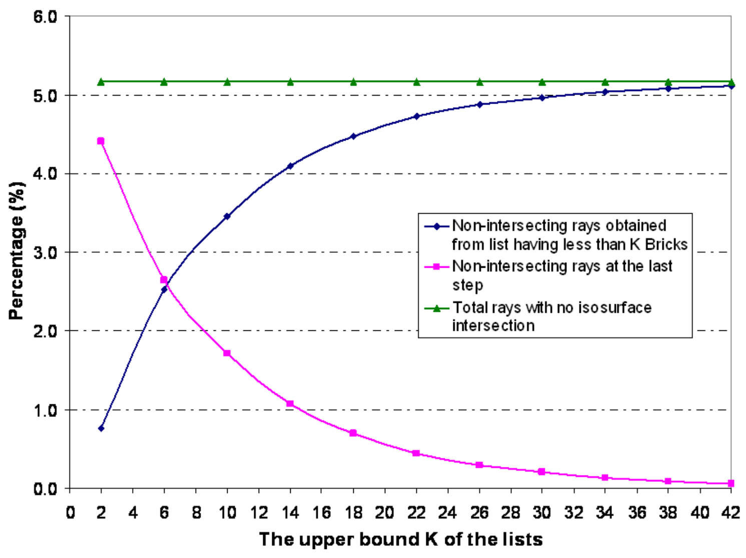

Figure 5.6: Analysis of the percentage of rays having no intersection with the iso-surface in our scheme using the $2048^2 \times 1920$ RMI dataset at time step 250 under Far-view. Screen size is $1024^2$.

uncommon in Far-view, the percentage of non-intersecting rays over total number of rays cast could be large (such as in Aneurism and Abdominal datasets). For the same RMI dataset and the same screen resolution, octree ray-casting 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 the testing that an optimal value of $k$ has been selected and report the performance corresponding to this value.

### 5.6.3 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 ($\phi$ ,

$\theta$), 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 are described later in Section 5.6.6, 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 RMI dataset (time step 250 and the isovalue is equal to 70), is listed in Table 5.4 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.

Table 5.4: Performance of our algorithm on the Clovertown in $fps$ for the RMI dataset with screen resolution $1024^2$ and isovalue 70 under Far and Close views.

| $1024^2$ 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 |

As already noted in previous research [19, 42], 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 octree 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 5.5 the comparison of number of ray traversal steps in our algorithm and octree ray-casting for various datasets clearly elucidates this aspect.

Table 5.5: Number of ray traversal steps undertaken during ray-casting in octree ray-casting and our algorithm for a screen size of $1024^2$ screen using all the datasets considered in this paper.

| $1024^2$ Screen Dataset | octree 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 |
| RMI (far) | 42,228 | 12,944 | 3.26 |
| RMI (close) | 42,868 | 6,129 | 6.99 |

Table 5.6: Measured performance on 8-core Clovertown in $fps$ for our scheme and the octree ray-casting algorithm under Far-view setting for various datasets

| Screen size | $512^2$ | | | $1024^2$ | | |
|---|---|---|---|---|---|---|
| Dataset | octree | ours | ratio | octree | 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 |

We now report a summary of our performance results on the other datasets illustrated in Fig. B.4. 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 5.6. Since these datasets have lower depth complexity than the RMI 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 octree ray-casting algorithm. Note also the significant performance achieved for the lower resolution screen of size $512^2$.

### 5.6.4 Performance Comparison

We compare our algorithm to the algorithm reported in [33], which uses the 16-core NUMA 2.4 GHz Opteron workstation. As far as we know, the performance numbers published in [33] are the best known for the general isosurface ray-casting problem. Since our platform is different than theirs, we need to calibrate the two processors. Comparing the SPEC benchmark[1] 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 5.7, we note that the Opteron runs slightly faster and has significantly better throughput than the Intel Xeon.

Table 5.7: Performance comparison between an 8-core Opteron 8218 and an 8-core Xeon E5320 using the SPEC benchmark.

| CPU Model | | AMD Opteron 8218 | Intel Xeon E5320 |
|---|---|---|---|
| CPU Clock<br>Multi-Core | | 2.6 GHz 4 processors<br>2-core per die | 1.86 GHz 2 processors<br>4-core per die |
| L1 Cache per core<br>L2 Cache per die<br>Main memory | | 64 KB I + 64 KB D<br>2 MB I+D<br>32 GB DDR2-5300 | 32 KB I + 32 KB D<br>8 MB I+D<br>16 GB DDR2-5300 |
| Speed | Cint<br>Cfp | 11.3<br>11.9 | 11.1<br>9.57 |
| Through-put | Cint_rate<br>Cfp_rate | 85.3<br>83.2 | 58.5<br>41.3 |

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

Table 5.8: Performance comparison in $fps$ for RMI 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 RMI datasets and testing views correspond to their settings with isovalue = 20.

| 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 |

Listed in Table 5.8 are the performance numbers reported in [33] 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 much better for close views and only worse by no more than 25% 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 [33].

### 5.6.5 Cache Performance

A critical factor affecting the performance of any ray-casting 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

Table 5.9: Comparison of cache misses of our algorithm and the octree ray-casting algorithm on the RMI dataset with isovalue 70 and screen resolution $1024^2$. Data load request refers to the number of requests issued for min/max and voxel values during the ray-casting; Cache miss is the number of requests that fail to find the requested data inside the cache after the initial load; Miss rate is calculated as cache miss divided by data load request.

| Number of Cores | | | Single-core | | 8-core | |
|---|---|---|---|---|---|---|
| View | Method | Data load request | Cache miss | Miss rate | Cache miss | Miss rate |
| Far | Octree | 122,341K | 2,153K | 1.76% | 3,340K | 2.73% |
| | Ours | 66,336K | 813K | 1.25% | 836K | 1.26% |
| | Ratio | 1.84 | 2.65 | – | 4.00 | – |
| Close | Octree | 123,881K | 2,384K | 1.92% | 5,305K | 4.28% |
| | Ours | 52,660K | 723K | 1.37% | 764K | 1.45% |
| | Ratio | 2.35 | 3.25 | – | 6.94 | – |

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, $8 \times 8$) 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 5.9 shows the cache miss rates achieved by our scheme and the octree ray-casting, on both a single core and an 8-core Clovertown for far and close views of the RMI dataset respectively. Here we have excluded the initial misses caused by the first time access to the data. As can be seen from Table 5.9, the number of data load requests and cache misses in our method are significantly lower than those for the octree method, and a thread in our method will rarely need to access the main memory after the first time the data was loaded.

### 5.6.6 Scalability

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 group of pixels 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 5.10, which shows the average frame rate over six views for the two different settings of the viewpoint on the RMI dataset using a varying number of cores. The results are for $512^2$ and $1024^2$ screen resolution respectively.

Table 5.10: Average frame rate of our algorithm on Clovertown for the RMI dataset at time step 250 under a varying number of CPU cores using $512^2$ and $1024^2$ screen resolution.

| Screen Size | $512^2$ | | $1024^2$ | |
|---|---|---|---|---|
| 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 | 92.4% | 92.2% | 97.5% | 96.2% |

In fact, an examination of Table 5.10 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, where the scalability at 8-core is computed as the percentage of the performance in $fps$ achieved on 8-core relative to the performance on a single core multiplied by 8 (best possible speedup). 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.

Table 5.11: 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 RMI dataset for both far and close views. The work load of Phase II is measured by the number of rays cast, the number of ray traversal steps and intersections. Total includes the synchronization time and writing time of the frame buffer.

| 8-core / $1024^2$ | | Number of ($\times 10^3$) | | | Time (msec) | | |
|---|---|---|---|---|---|---|---|
| View | Proc No. | Rays | Traversal | Intersect | I | II | Total |
| Far | 0 | 98 | 1,380 | 277 | 19 | 223 | 250 |
| | 1 | 102 | 1,353 | 274 | 22 | 223 | 250 |
| | 2 | 99 | 1,365 | 276 | 22 | 223 | 250 |
| | 3 | 97 | 1,393 | 274 | 18 | 224 | 251 |
| | 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 |
| $\frac{\sigma}{Avg} \times 100\%$ | | 2.14 | 0.81 | 0.48 | 10.8 | 0.21 | 0.19 |
| Close | 0 | 147 | 850 | 382 | 21 | 202 | 231 |
| | 1 | 155 | 844 | 378 | 21 | 203 | 232 |
| | 2 | 140 | 897 | 360 | 21 | 202 | 231 |
| | 3 | 127 | 915 | 381 | 20 | 202 | 231 |
| | 4 | 149 | 857 | 379 | 21 | 203 | 232 |
| | 5 | 157 | 880 | 371 | 22 | 202 | 231 |
| | 6 | 149 | 852 | 377 | 20 | 200 | 230 |
| | 7 | 151 | 844 | 382 | 21 | 203 | 232 |
| $\frac{\sigma}{Avg} \times 100\%$ | | 4.39 | 2.61 | 1.39 | 2.32 | 0.31 | 0.27 |

Another way to illustrate the scalability of our scheme is through Table 6.6 that shows the loads on the different threads for the RMI 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 intersection are almost evenly distributed among the threads regardless of the viewpoint. Therefore the loads are extremely well-balanced among the different threads.

## 5.7 Summary

In this chapter we presented a novel hybrid strategy for rendering isosurfaces by ray-casting. 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-casting 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.

# Chapter 6

# Extension of Hybrid Scheme to Direct Volume Rendering(DVR)

As mentioned before, the surface representation is not able to account for the internal translucent structures of the datasets which appear in most applications. Direct volume rendering creates images from the 3D datasets by sampling the optical properties and compositing their visual effects throughout the data volume. While there are several different approaches to perform direct volume rendering, we focus here on the ray-casting method since this seems to produce the highest visualization quality for volumetric datasets among software-based technology (as outlined in Section 3.3). In general, the work involved in direct volume rendering using ray-casting can be divided into two parts: the first part comes from the traversal of the rays across the data volume, and the second is the opacity/color composition along the ray, which makes use of Riemann sums and a transfer function. Hence, this process demands much more computation than in the isosurface case. In this chapter, we extend the method presented in the previous chapter to develop an extremely effective algorithm for DVR.

Before presenting our method, we introduce some terminology for describing our algorithm. We denote a set of spatially adjacent cells (e.g., $4 \times 4 \times 4$ or $8 \times 8 \times 8$ cells) as a *data block* or simply *block*. Given a transfer function and a preset small positive value $\epsilon$, we can characterize data blocks as follows: *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 $\geq \epsilon$.

## 6.1 Extended Hybrid Ray-casting Scheme

Similar to the hybrid ray-casting scheme for isosurfaces, our strategy for DVR consists of two main phases.

Phase I involves an object order traversal of the data using a spatial hierarchical BONO tree representation whose leaves represent data blocks. The main purpose of this phase is to determine for each group of pixels (containing $2 \times 2$ pixels in our implementation) a list of non-transparent data blocks (i.e. semi-transparent and diffuse blocks) encountered by the packet of rays in a front-to-back order. As before, we place an upper bound $k$ on the number of such data blocks for all the lists. This will substantially reduce the traversal time without affecting the rendering quality.

Phase II consists of shooting rays through the pixels using the lists of data blocks generated during Phase I. The blocks are processed in a front-to-back order and the early termination technique 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.

We next provide more details about our scheme.

### 6.1.1 Pre-processing

The preprocessing step starts by constructing the BONO-tree as described in Section 5.2.1 except that a region with constant voxel value can not now be ignored. We organize our volumetric data into a coarse grid of data blocks with fixed size (each block contains $8 \times 8 \times 8$ or $4 \times 4 \times 4$ cells), 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. To accommodate 2D transfer functions, we compute the normal magnitude at each vertex using the method of center difference. We use an octree to index the data within a data block and store the minimum and maximum density values and normal magnitude pairs, $(d_{min}, h_{min})$ and $(d_{max}, h_{max})$, at each node with the leaves corresponding to $2 \times 2 \times 2$ cells, where $d_{min}$ and $h_{min}$ are respectively the minimum density value and the minimum normal magnitude of all the vertices contained within the subtree. The parameters $d_{max}$ and $h_{max}$ correspond to the maximum values. In addition, we build a BONO (Branch-On-Need Octree) [23] tree on the coarse grid consisting of the data blocks, augmented as usual by the appropriate min/max pairs of density values and magnitude of normals. 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 data. We will later show that our indexing structure is compact for all our datasets and the preprocessing step can be performed very quickly even for very large datasets.

### 6.1.2 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 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 $\epsilon$ and user-specified transfer function $opacity(v, g)$, the value of $D(v_i, g_j)$ is set as 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 which corresponds to a subvolume (See Figure 6.1). 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 $D(v_i, g_j)$ we can compute the value $\Delta$, the number of pairs $(v_k, g_l)$ such that $v_{min} \leq v_k \leq v_{max}$, $g_{min} \leq g_l \leq g_{max}$, and $opacitiy(v_k, g_l) > \epsilon$ by Equation 6.1

$$\Delta = D(v_{max}, g_{max}) - D(v_{min-1}, g_{max}) - D(v_{max}, g_{min-1}) + D(v_{min-1}, g_{min-1}) \tag{6.1}$$

Then, the $\Delta$ value can be used to classify the blocks as follows:

- If $\Delta = 0$, the block is transparent;

- If $\Delta$ is equal to the number of pairs in the box bounded 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 this case the block could be *transparent* in which case we will process it unnecessarily but this will not affect in any way the correctness of the rendering process.

Since $\Delta$ represents the number of discrete values of the pairs $(v_i, g_j)$ which are mapped to an opacity larger than $\epsilon$, 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 extended scheme.

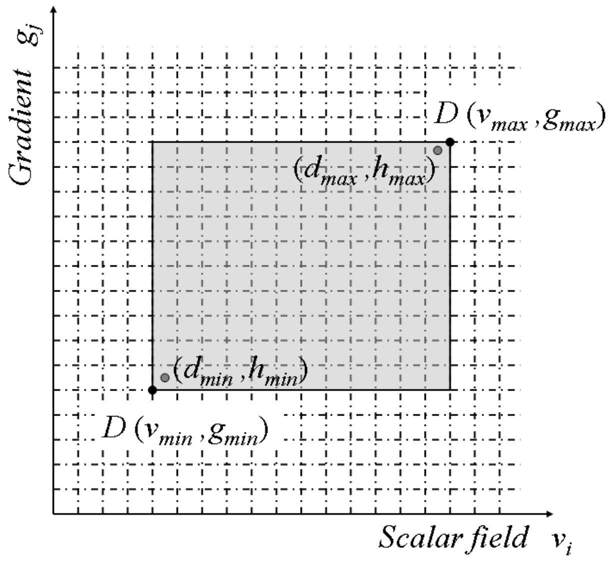

Figure 6.1: An illustrative diagram for employing 2D-dimensional discriminate function $D(v_i, g_j)$, given $(d_{min},\ h_{min})$ and $(d_{max},\ h_{max})$. The shaded area is bounded by discrete pair values $(v_{min},\ g_{min})$ and $(v_{max},\ g_{max})$. $v_{min} = \lfloor d_{min} \rfloor$, $g_{min} = \lfloor h_{min} \rfloor$, $v_{max} = \lceil d_{max} \rceil$, $g_{max} = \lceil h_{max} \rceil$ when $v_i$ and $g_j$ takes consecutive integer values.

### 6.1.3 Sequential Algorithm

After the preprocessing step, Phase I of our strategy is implemented as follows. The BONO tree is traversed starting from the root. Assume we reach a node $v$. The block discrimination process is applied to $v$ with min/max range values retrieved from the node. We skip the node if it is classified as transparent; otherwise, we process the node as follows. We project the minimum bounding box of $v$ onto the screen, and consider all the sizes of the lists corresponding to all the groups of pixels intersecting the projection. Should any of the lists be of size smaller than the upper bound $k$, we proceed and traverse the children of node $v$ if $v$ is semi-transparent in a front-to-back order relative to the viewpoint. Once a block of certain size (such as $8 \times 8 \times 8$ for far view or $4 \times 4 \times 4$ for close view) is reached, we augment the lists corresponding to the groups of pixels, which overlap the projection, with a pointer to the block. If $v$ is classified as diffuse, we project the minimum bounding box of $v$ onto the screen and augment the corresponding block lists by pointing to node $v$ whenever the size of the list is less than $k$.

Since the nodes are traversed and projected along the viewing order, a falsely classified semi-transparent node will not produce an error and is 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 group of pixels, such that the blocks are arranged in front-to-back order relative to the viewpoint.

During Phase II, we shoot rays through the pixels of each group using the corresponding block lists. Each ray proceeds using grid traversal for diffuse blocks or octree traversal for semi-transparent blocks along the viewing direction while compositing the colors and updating the opacity values using the Riemann sum (Equations 2.2 and 2.3) with the specified transfer function, while 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 volume boundary. We will also show that the percentage of such continued ray traversals is minimal given any sufficiently large value of $k$.

### 6.1.4 Parallel Processing

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 work; (ii) ensure as little communication and coordination among the threads as possible; and (iii) make effective use of the memory hierarchy present in multi-core 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.

The parallel scheme of extended hybrid ray-casting is very similar to the one for isosurface ray-casting presented before, and involves a progressively finer partition of the tasks, each of which corresponding to shooting rays through pixels, coupled with dynamic allocation of these tasks to the cores as they become available. In fact, the multithreaded scheme for hybrid ray-casting described in the previous chapter can be easily adapt to direct volume rendering and achieve the three goals above by modifying the computation of weight $W$ in the step of Z-Order Screen Partitioning (in Section 5.4.2). That is, in our parallel direct volume rendering, the weight $W$ is computed as the summation of the sizes of all the blocks on the lists rather than the number of non-empty lists to more accurately estimate the workload during the following ray-casting task.

In the next section, we present experimental results over a wide variety of datasets, which illustrate the scalability of our algorithm and the excellent load balance achieved among the different cores.

## 6.2 Experiments and Performance Analysis

We have conducted extensive tests of our algorithm on six volumetric 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.

Since the work involved in direct volume rendering using ray-casting can be divided into the ray traversal across space, and the color composition along the rays through non-transparent region, we report on the number of ray traversal steps (excluding those steps that contribute to 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 using a varying number of cores on Clovertown platform. Note that excluding Phase I of our scheme (i.e., block-list length $k = 0$) converts our algorithm into a variant of the well-known image-based DVR approaches, such as those proposed by [56] and [64]. 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.

### 6.2.1 Datasets Used and Space Consumption

The six datasets used in our tests (shown in Figure 6.2 as rendered by our scheme) belong to the following three major domains: geometric objects (Bonsai, XTree), medical CT scan (Cadaver, Prone, VF), and scientific simulation (RMI). In our scheme, data blocks consisting of background voxels do not need to be indexed, resulting in a very compact representation in general. As shown in Table 6.1 for the six datasets, the corresponding BONO trees are quite compact and the space required by all the octrees representing the data blocks 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 6.3 to enable us to construct the 2D transfer function for volume rendering.

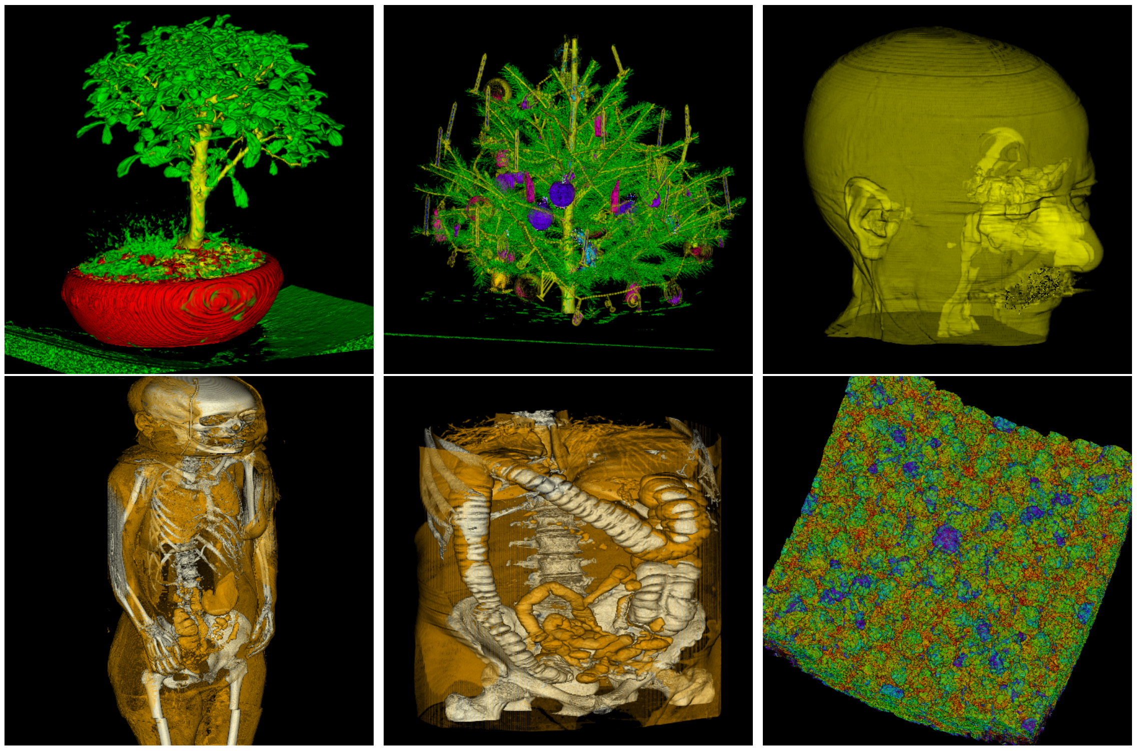

Figure 6.2: The six datasets used in our tests. From left to right, top to down, the datasets are: Bonsai, XmasTree(XTree), Cadaver, VisFemale(VF), Prone and Richtmyer-Meshkov Instability (RMI) respectively.

### 6.2.2 Performance Implication on List Upper Bound

As in the case of isosurface ray-casting, the upper bound $k$ imposed on the block-list plays a critical role in accelerating the ray-casting with a very small overhead incurred during Phase I of our algorithm. To quantify the performance gain due to the value of $k$, we analyze our performance as follows.

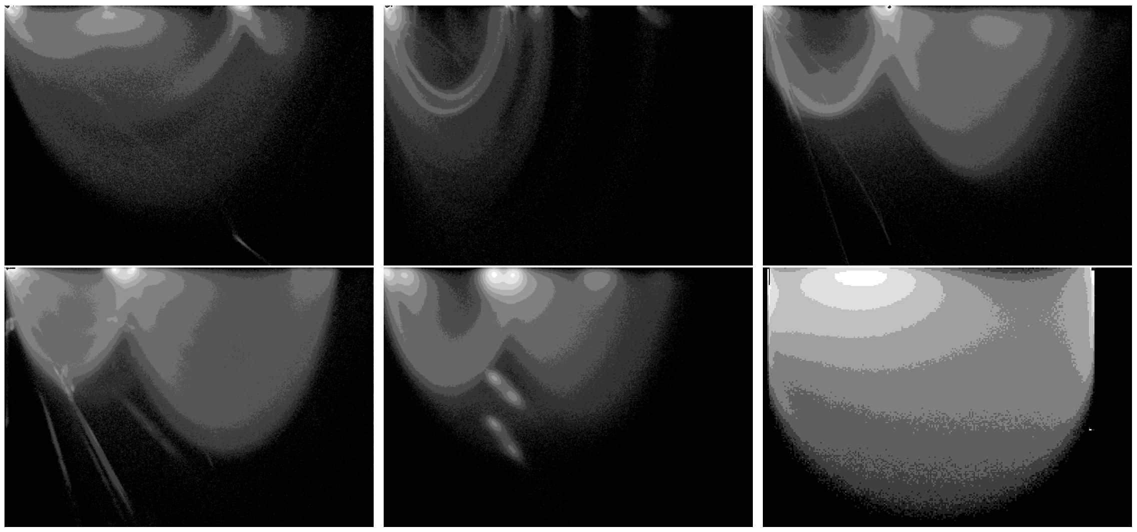

Figure 6.3: The 2D histograms (i.e. density value and magnitude of the gradient) of the six datasets used in our tests. From left to right, top to down, the datasets are: Bonsai, XmasTree, Cadaver, VisFemale, Prone and RMI respectively.

The total execution time of our algorithm consists of four main components:

(i) the time of Phase I for generating the lists of blocks using the BONO tree traversal;

(ii) the time of grouping the small-sized image tiles into larger tasks for dynamic allocation among the processor cores, which is typically about $2 \sim 3$ msec;

(iii) the time of ray-casting through the data blocks on the lists to composite

Table 6.1: Sizes of grid and indexing structure for various datasets used along with their preprocessing time. The dimension of data block in pre-processing is of $4 \times 4 \times 4$ cells for all the tested datasets except that RMI dataset has $8 \times 8 \times 8$ cells for each block.

| Dataset | Dimension | Grid | BONO | Block | Time(sec) |
|---|---|---|---|---|---|
| Bonsai | $512^2 \times 308 \times 2$B | 154MB | 8.5MB | 44.2 (28.7 %) | 29.3 |
| Cavader | $512^2 \times 424 \times 2$B | 212MB | 8.2MB | 45.1 (21.3 %) | 31.2 |
| Prone | $512^2 \times 462 \times 2$B | 231MB | 17.1MB | 94.3 (36.8 %) | 33.9 |
| XTree | $512^2 \times 512 \times 2$B | 256MB | 17.4MB | 94.0 (36.7 %) | 35.5 |
| VF | $512^2 \times 1734 \times 2$B | 867MB | 45.0MB | 159.5 (18.4 %) | 125.1 |
| RMI | $2048^2 \times 1920 \times 1$B | 7,680MB | 51.27MB | 1,596 (20.8 %) | 736.1 |

the colors and opacities;

(iv) the time needed for ray-casting of the unfinished pixels (i.e., those pixels whose lists were of size $k$ but with accumulated opacity short of threshold).

We divide the work in Phase II into color composition steps over non-transparent regions and ray traversal steps for empty space leaping via the hierarchical structure. We conduct a series of tests on the LLNL RMI dataset at the time step 250 to capture the trade-off among the execution times of the various steps with varying $k$ values, using a single core on our Clovertown platform with a $1024^2$ screen.

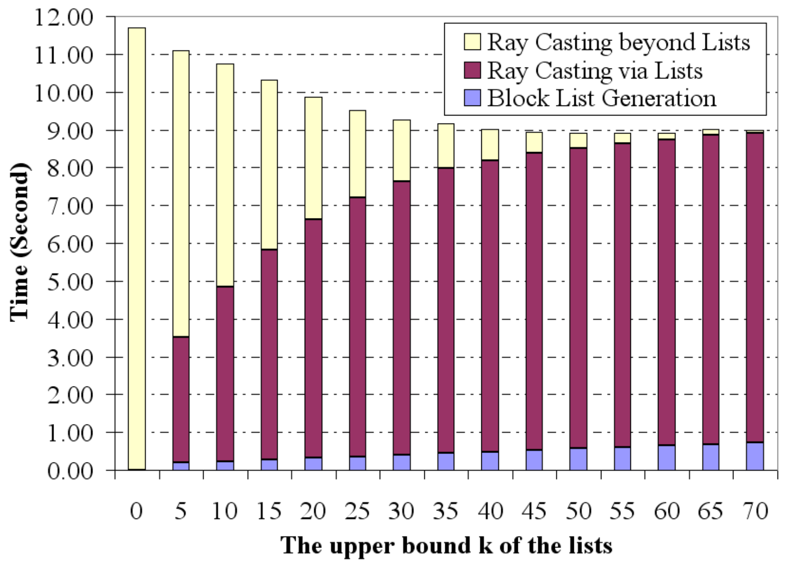

Figure 6.4: Execution times of the different stages of our algorithm on a single core vs. the value of the upper bound $k$. The results are for the Far-view of the LLNL RMI dataset of time step 250 using $1024^2$ screen resolution.

From the results in Figure 6.4, we can make the following observations. First, the pure ray-casting without the generation of block-lists (i.e., $k = 0$) has the longest execution time by a factor of approximately 40% relative to the best value of $k$. Second, the time it takes to generate the block lists in Phase I (indicated in

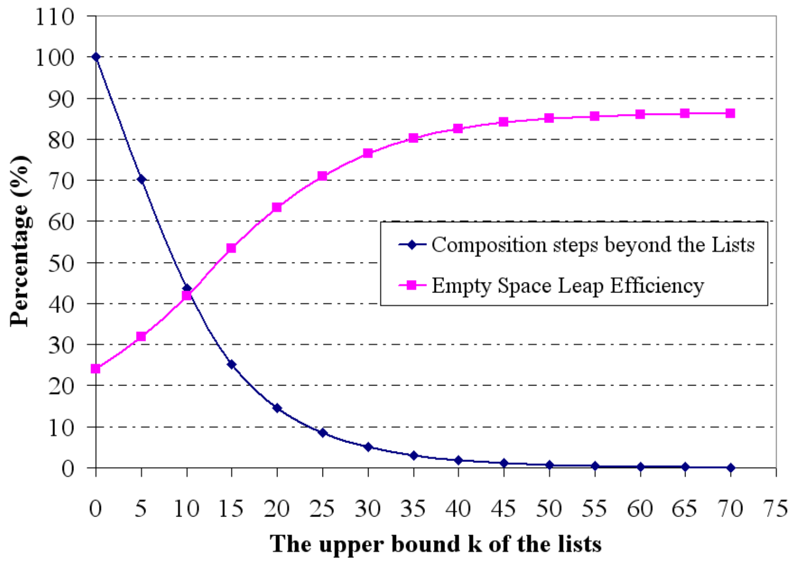

Figure 6.5: Percentage of composition steps left after going through $k$ blocks from the lists and Percentage of composition steps over total steps (including composition and traversal steps), i.e. Empty Space Leap Efficiency, for the Far-view of RMI dataset at time step 250. Screen size is $1024^2$.

blue) increases with the value $k$ almost linearly but its contribution to the total time is less than 10% for all the tested values of $k$. Third, the time that Phase II takes for compositing beyond the $k$ blocks (indicated in yellow) goes down very quickly initially as $k$ increases and then somewhat levels off, which makes an optimal value $k$ at around 40.

During the ray traversal in Phase II, traversal steps lead the rays to skip the empty space while compositing steps compute the optical interaction with the material and composite the visual effects along the ray transport. We measure the percentage of the number of compositing steps over the total number of steps (which include compositing and traversal steps), called *Empty Space Leap(ESL) Efficiency*, as well as the percentage of the number of compositing steps beyond the $k$ blocks over the total number of the steps. As illustrated in Figure 6.5, the increasing *ESL*

Efficiency with the list upper bound $k$ (red curve) indicates that utilizing block lists in Phase II significantly reduces the number of ray traversal steps because the number of compositing steps stays the same over the value $k$. However, the increment becomes somewhat flat when $k$ reaches a certain value because much fewer composition steps are left after an increasing number of $k$ blocks are processed by ray-casting (indicated by the blue curve in Figure 6.5).

Table 6.2: Measured execution time of our DVR algorithm for Phase I and II as well as the number of composition(CP) steps within optimal $k$ 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 CP steps ($\times 10^3$) | | |
|---|---|---|---|---|---|---|
| Datasets | $k$ | I | II | $k$ blocks | beyond | (%) |
| Bonsai | 70 | 0.32 | 3.87 | 10,751 | 164 | 1.50 |
| Cadaver | 75 | 0.24 | 4.52 | 11,926 | 350 | 2.85 |
| Prone | 70 | 0.67 | 8.21 | 19,966 | 387 | 1.90 |
| XTree | 70 | 0.38 | 5.24 | 16,243 | 284 | 1.72 |
| VF(far) | 70 | 0.34 | 5.26 | 13,536 | 186 | 1.36 |
| VF(close) | 70 | 0.56 | 8.20 | 29,158 | 334 | 1.13 |
| RMI(far) | 40 | 0.34 | 5.98 | 15,901 | 289 | 1.79 |
| RMI(close) | 60 | 0.67 | 7.44 | 22,862 | 346 | 1.50 |

The tests conducted on the other datasets yield similar results under their individual optimal values of $k$ to those obtained for the LLNL RMI dataset. As can be seen from these results listed in Table 6.2, the time of Phase I is well below 10% of the Phase II time while the number of compositing steps beyond $k$ blocks are a small percentage of the total number of steps. The numbers of compositing and ray traversal steps undertaken in Phase II for various datasets and view types under corresponding optimal $k$ values in Table 6.3 confirm that the number of traversal steps are dramatically reduced by using Phase I, which in turn results in an increase

Table 6.3: 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 value $k^*$ 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 |

on ESL efficiency by a factor of up to 4.

### 6.2.3 Overall Performance

We report on the overall performance 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. The comparison is made with the well-known DVR ray-casting approach based on the use of the octree, which is same as our implementation when the block list length $k$ is set to zero. However we use our multi-threaded scheme in all cases, which gives a substantial boost to the previous DVR ray-casting approach in terms of scalability on our multi-core processor. In fact, the performance numbers reported for this scheme on the 8-core platform are significantly better than the published ray-casting results for similar size datasets and screens with software-based acceleration techniques. Table 6.4 summarizes the results, and shows that the frame rate of our algorithm can reach $2 \sim 6 fps$ for

$512^2$ screen and $1 \sim 2fps$ for a screen of resolution $1024^2$. For the $1024^2$ screen, the performance improvement over the well-known octree algorithm (with our new multi-threaded implementation) can range from 40% to over 100%, while it becomes a bit lower on a $512^2$ screen because Phase I is less effective when the projection of BONO tree node covers fewer pixels under lower resolution.

Table 6.4: Measured DVR performance on 8-core Clovertown in $fps$ for our scheme and the octree ray-casting algorithm on $512^2$ and $1024^2$ screen for various datasets

| Screen size | $512^2$ | | | $1024^2$ | | |
|---|---|---|---|---|---|---|
| Dataset | octree | ours | ratio | octree | ours | ratio |
| Bonsai | 4.67 | 6.58 | 1.41 | 1.26 | 1.84 | 1.46 |
| Cadaver | 3.50 | 5.46 | 1.56 | 0.93 | 1.60 | 1.72 |
| Prone | 2.00 | 2.99 | 1.49 | 0.53 | 0.89 | 1.68 |
| XTree | 2.92 | 4.42 | 1.52 | 0.78 | 1.36 | 1.74 |
| VF(far) | 2.89 | 3.92 | 1.36 | 0.81 | 1.31 | 1.61 |
| VF(close) | 1.57 | 2.54 | 1.62 | 0.40 | 0.89 | 2.19 |
| RMI(far) | 3.01 | 3.76 | 1.25 | 0.87 | 1.26 | 1.45 |
| RMI(close) | 2.62 | 3.85 | 1.47 | 0.69 | 0.98 | 1.42 |

Table 6.5: Average frame rate of our DVR algorithm on Clovertown for the 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% |

### 6.2.4 Scalability

We turn now to examining 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 6.5 shows the average frame rate over six views for the two different settings of the viewpoint on the 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 we increase the number of cores to the maximum of eight available on our platform.

Table 6.6 illustrates the loads on the different threads for the 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 cost. Therefore the loads are extremely well-balanced among the different threads.

## 6.3 Summary

In this chapter, we extended the hybrid ray-casting strategy for direct volume rendering. The resulting algorithm uses an object order traversal coupled with a novel block discrimination method to generate a list of blocks in viewing order for each group of pixels and achieve efficient empty space skipping in the following ray-casting stage. We have shown that the total size of our indexing structure is very compact and that our performance is significantly superior along with high scalability and excellent load balancing relative to the published software-based

Table 6.6: The work from the two phases distributed among eight threads running on 8-core for $1024^2$ screen along with their corresponding individual execution time. The tests are done on 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 steps($\times 10^3$) | | Time (msec) | | |
|---|---|---|---|---|---|---|
| View | Proc No. | Composition | Traversal | I | 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 |
| $\frac{\sigma}{Avg} \times 100\%$ | | 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 |
| $\frac{\sigma}{Avg} \times 100\%$ | | 6.56 | 0.73 | 1.36 | 0.10 | 0.10 |

schemes. We presented the results of some of our extensive tests, showing interactive rendering rates for a variety of datasets of widely different sizes. All these results indicate that our scheme can deliver interactive rendering of large scale volumetric data on emerging multi-core processors.

# Chapter 7

# Conclusion

Volume Rendering is widely used as an effective approach for the visual exploration, computational analysis, and manipulation of volumetric datasets. Due to the dramatic advances in imaging instruments and computing technologies, such datasets are now appearing in many engineering, science and medical applications at a very fast rate with increasingly larger sizes from giga-bytes to tera-bytes. The visualization of such datasets is critical to enable the discovery of features and patterns of interest. However, visualizing such large scale of datasets interactively is a computationally demanding task and poses significant challenges to carry it out on current computing platforms. Currently clusters of processors and multi-core processors are the two most important computing platforms. A significant amount of research has concentrated on exploiting their computational capability and potential use for volume visualization of large scale datasets.

## 7.1 Summary

In this dissertation, we have developed parallel schemes for visualizing volumetric datasets by rendering isosurfaces or by direct volume rendering on two parallel architectures, multiprocessor clusters and multi-core processors. Our first algorithm for isosurface computation contains a new simple indexing structure called compact interval tree for out-of-core isosurface extraction and rendering of large scale data sets and an efficient and scalable implementation on multiprocessor environments in which each processor has access to its own local disk. The second scheme proposes a new hybrid method for the interactive rendering of isosurfaces using ray-casting 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. We have extended the hybrid ray-casting scheme to parallel direct volume rendering with a new block discrimination method in support of interactive 2D transfer function and efficient empty space leaping.

The key components in our new schemes include compact interval tree indexing structure, span space partitioning scheme, upper bounded block list generation for groups of pixels, Z-order screen partitioning and dynamic task dispatching, as well as block discrimination method under 2D transfer function. The major achievements are summarized below:

- Part I: The compact interval tree enables the identification of the active cells extremely quickly, using more compact indexing structure and more effective bulk data movement than previous schemes. Moreover, the span space partitioning leads to a parallel implementation of the algorithm that provably achieves load balancing across the processors independent of the isovalue, with almost no overhead in the total amount of work relative to the sequential algorithm. We have conducted a large number of experimental tests on the University of Maryland Visualization Cluster using the RichtmyerMeshkov instability data set, and have obtained results that consistently validate the efficiency and the scalability of our algorithm.

- Part II: The generation of the lists of data blocks enables the identification of the possible regions which could produce the isosurfaces for each pixel on the screen and significantly reduce the computation by efficiently skipping the empty space and irrelevant pixels. The grouping of nearby pixels exploits the data locality among the neighboring block lists and helps achieve higher cache performance. The upper bound imposed on the block list makes the list generation process computationally and algorithmically efficient. Based upon the block lists, the Z-order screen partitioning scheme allows 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 have tested our algorithm on a dual-processor Clovertown platform, each consisting of a Quad-Core 1.86 GHz Intel Xeon Processor, showing that our method is efficient and scalable, and achieves high cache performance and excellent load balancing, resulting in an 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.

- Part III: The hybrid ray-casting scheme was extended to direct volume rendering with the same high degree of scalability, excellent load balancing, and efficient memory management as in the case of isosurface. The block discrimination method allows interactively specifying a 2D transfer function and produce the block lists to achieve efficient empty space leaping in direct volume rendering. Our extensive experimental tests on the same Clovertown platform over a wide variety of well-known volumetric datasets show significantly superior performance for direct volume rendering, and result in interactive rates for very large datasets such as the RichtmyerMeshkov instability dataset on high-resolution screens.

## 7.2 Further Discussion

Our hybrid ray-casting scheme can achieve interactive isosurface rendering and direct volume rendering on large scale volumetric datasets upon high resolution screen. This technology can also be applied to other areas as well. Some of the possible extensions include:

(i) Geometric object visualization — physical objects are usually represented by bounding polygonal meshes. The BONO tree built upon the bounding box of nearby polygons can produce the list of 3D blocks which is used to identify potential intersection with rays and skip the empty space up front. Similar to the case of isosurface, the intersecting location between the viewing ray and the object can be computed and the shading can be performed for the corresponding pixel. However, since the size of polygons may differ dramatically from each other, the size of 3D blocks can be adaptive as well to accommodate the difference and obtain the higher acceleration based upon our hybrid scheme.

(ii) Multivariate data visualization — visualizing multivariate datasets is an important and effective method to discover and reveal the relations among the underlying data components and has received more research attention in the literature. For the single field datasets, one dimensional transfer function can be the choice. While, 2D or higher dimensional transfer functions are used to work on multivariate datasets with each dimension corresponding to one individual data component. Our extended hybrid ray-casting scheme for DVR can directly adapt to the two-scalar field datasets by associating the secondary dimension with the values on the other scalar field. For the multivariate datasets with more scalar fields, we can build the BONO tree with more augmented min/max ranges and examine the relation between any two scalar fields swiftly by the existing handling method for 2D transfer function. However, to quickly explore the relation among three or more scalar fields demands a new way of handling transfer function, which is one of current research topics in the field of volume visualization and beyond the scope of this study.

Our ray-casting techniques focus on speeding up primary ray traversal but does not seem to be applicable to secondary ray traversals for ray-tracing in a straightforward way. Nevertheless, the primary ray traversal is the most time consuming part in ray-tracing and using ray-casting can sufficiently produce high quality isosurfaces and direct volume rendering for most applications in scientific visualization. Given the ever increasing sizes of the volumetric data, an interesting issue is whether our ray-casting techniques can be extended to handle out-of-core data. We believe that this is the case and the coherency among the neighboring blocks and between two consecutive views can be exploited to accelerate ray-casting upon out-of-core datasets. It is expected that the details of such extensions will come up in future research.

# Appendix A

## Performance of Our Isosurface Extraction Algorithm on the Cluster

Table A.1: Execution time of our isosurface extraction algorithm with two processors over varying isovalues.

| Iso-value | Retrieval (sec) | Triangulation(sec) | Rendering(sec) | Total Time(sec) | Rate $(10^6/s)$ | Speedup |
|---|---|---|---|---|---|---|
| 10 | 7.774 | 18.391 | 9.625 | 36.005 | 6.32 | 1.96 |
| 30 | 11.015 | 29.338 | 10.948 | 53.31 | 7.046 | 1.89 |
| 50 | 12.871 | 43.044 | 16.549 | 74.827 | 7.589 | 1.92 |
| 70 | 12.971 | 48.655 | 17.532 | 80.537 | 8.064 | 1.98 |
| 90 | 11.03 | 38.386 | 14.842 | 64.42 | 7.891 | 1.98 |
| 110 | 8.615 | 25.165 | 9.596 | 43.568 | 7.521 | 1.99 |
| 130 | 6.732 | 17.672 | 6.711 | 31.412 | 7.297 | 1.97 |
| 150 | 5.497 | 13.607 | 5.181 | 24.639 | 7.185 | 1.96 |
| 170 | 4.631 | 11.185 | 4.28 | 20.33 | 7.2 | 1.97 |
| 190 | 4.012 | 9.548 | 3.674 | 17.479 | 7.172 | 1.97 |
| 210 | 3.585 | 8.267 | 3.195 | 15.162 | 7.188 | 1.97 |

Table A.2: Execution time of our isosurface extraction algorithm with four processors over varying isovalues.

| Iso-value | Retrieval (sec) | Triangulation(sec) | Rendering(sec) | Total Time(sec) | Rate $(10^6/s)$ | Speedup |
|---|---|---|---|---|---|---|
| 10 | 4.583 | 9.299 | 4.867 | 19.979 | 11.45 | 3.54 |
| 30 | 6.009 | 14.705 | 5.455 | 27.604 | 13.642 | 3.66 |
| 50 | 6.601 | 21.58 | 8.251 | 37.81 | 15.059 | 3.8 |
| 70 | 6.243 | 24.186 | 9.475 | 40.172 | 16.226 | 3.97 |
| 90 | 5.228 | 19.109 | 7.436 | 32.331 | 15.81 | 3.94 |
| 110 | 4.38 | 12.645 | 4.783 | 22.23 | 14.818 | 3.9 |
| 130 | 3.606 | 8.827 | 3.323 | 16.213 | 14.137 | 3.82 |
| 150 | 2.956 | 6.8 | 2.571 | 12.73 | 13.907 | 3.79 |
| 170 | 2.482 | 5.596 | 2.135 | 10.549 | 13.875 | 3.8 |
| 190 | 2.136 | 4.779 | 1.825 | 9.11 | 13.761 | 3.78 |
| 210 | 1.799 | 4.119 | 1.587 | 7.806 | 13.961 | 3.84 |

Table A.3: Execution time of our isosurface extraction algorithm with eight processors over varying isovalues.

| Iso-value | Retrieval (sec) | Triangulation(sec) | Rendering(sec) | Total Time(sec) | Rate $(10^6/s)$ | Speedup |
|---|---|---|---|---|---|---|
| 10 | 2.332 | 4.72 | 2.511 | 9.972 | 23.055 | 7.09 |
| 30 | 3.004 | 7.458 | 2.778 | 13.989 | 27.012 | 7.22 |
| 50 | 2.984 | 10.875 | 4.192 | 18.558 | 30.761 | 7.75 |
| 70 | 3.005 | 12.263 | 4.792 | 20.387 | 32.049 | 7.83 |
| 90 | 2.733 | 9.724 | 3.761 | 16.648 | 30.784 | 7.65 |
| 110 | 2.269 | 6.382 | 2.422 | 11.867 | 27.86 | 7.31 |
| 130 | 1.903 | 4.456 | 1.691 | 8.771 | 26.256 | 7.06 |
| 150 | 1.561 | 3.433 | 1.315 | 6.947 | 25.629 | 6.94 |
| 170 | 1.293 | 2.827 | 1.09 | 5.797 | 25.416 | 6.91 |
| 190 | 1.08 | 2.408 | 0.934 | 4.928 | 25.63 | 6.98 |
| 210 | 0.923 | 2.088 | 0.81 | 4.22 | 26.041 | 7.09 |

Table A.4: Execution time of our isosurface extraction algorithm with sixteen processors over varying isovalues.

| Iso-value | Retrieval (sec) | Triangulation(sec) | Rendering(sec) | Total Time(sec) | Rate ($10^6/s$) | Speedup |
|---|---|---|---|---|---|---|
| 10 | 1.396 | 2.372 | 1.251 | 5.994 | 38.167 | 11.8 |
| 30 | 1.461 | 3.759 | 1.372 | 7.584 | 49.654 | 13.31 |
| 50 | 1.776 | 5.493 | 2.044 | 9.911 | 57.45 | 14.51 |
| 70 | 1.803 | 6.16 | 2.32 | 10.936 | 59.604 | 14.6 |
| 90 | 1.538 | 4.85 | 1.813 | 8.769 | 58.289 | 14.52 |
| 110 | 1.222 | 3.186 | 1.182 | 6.019 | 54.728 | 14.41 |
| 130 | 0.934 | 2.229 | 0.828 | 4.379 | 52.341 | 14.13 |
| 150 | 0.749 | 1.719 | 0.642 | 3.472 | 50.989 | 13.89 |
| 170 | 0.645 | 1.418 | 0.528 | 2.868 | 51.035 | 13.96 |
| 190 | 0.572 | 1.206 | 0.449 | 2.463 | 50.9 | 13.97 |
| 210 | 0.515 | 1.046 | 0.388 | 2.09 | 52.142 | 14.33 |

# Appendix B

## Rendered Isosurface and DVR Images by Ray-casting for RMI Datasets

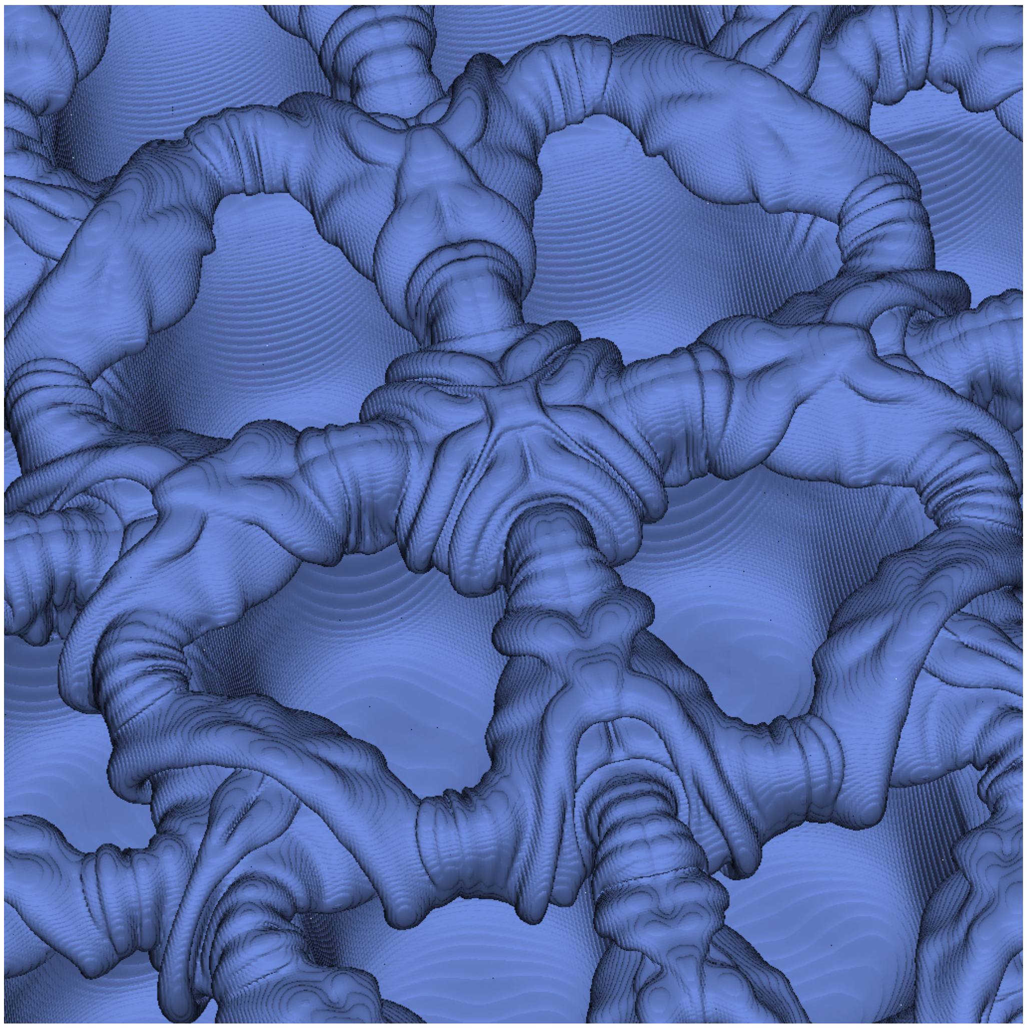

Figure B.1: Isosurface image rendered by ray-casting under close-view for RMI dataset at time step 50 with isovalue 70.

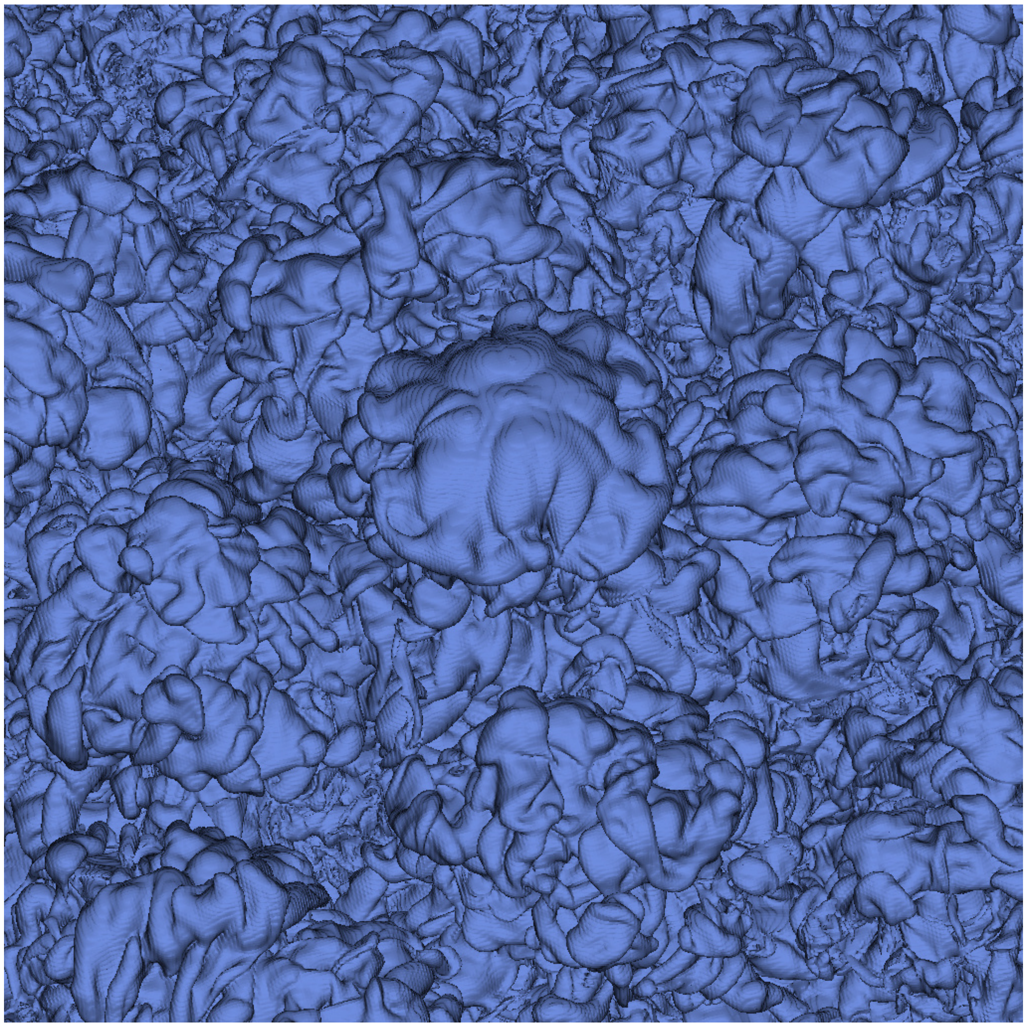

Figure B.2: Isosurface image rendered by ray-casting under close-view for RMI dataset at time step 250 with isovalue 70.

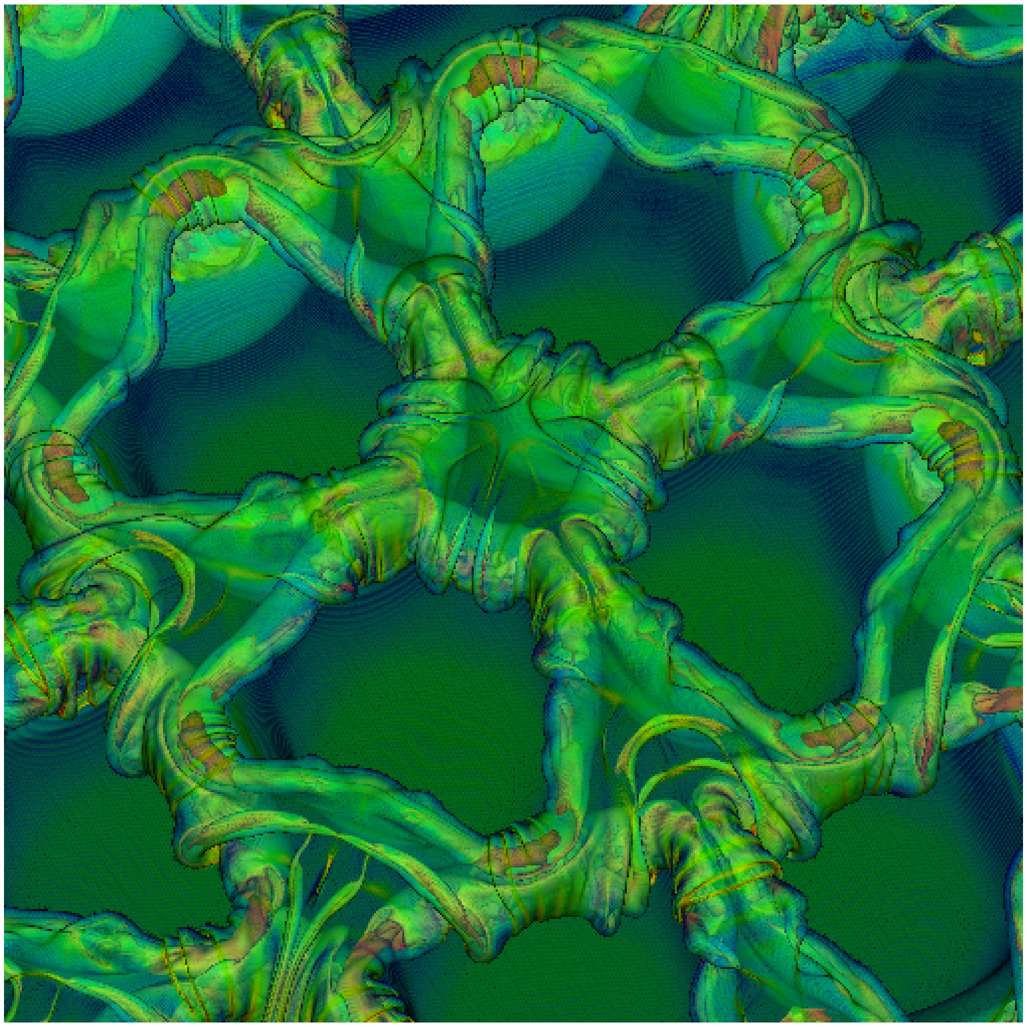

Figure B.3: DVR image rendered by ray-casting under close-view for RMI dataset at time step 50.

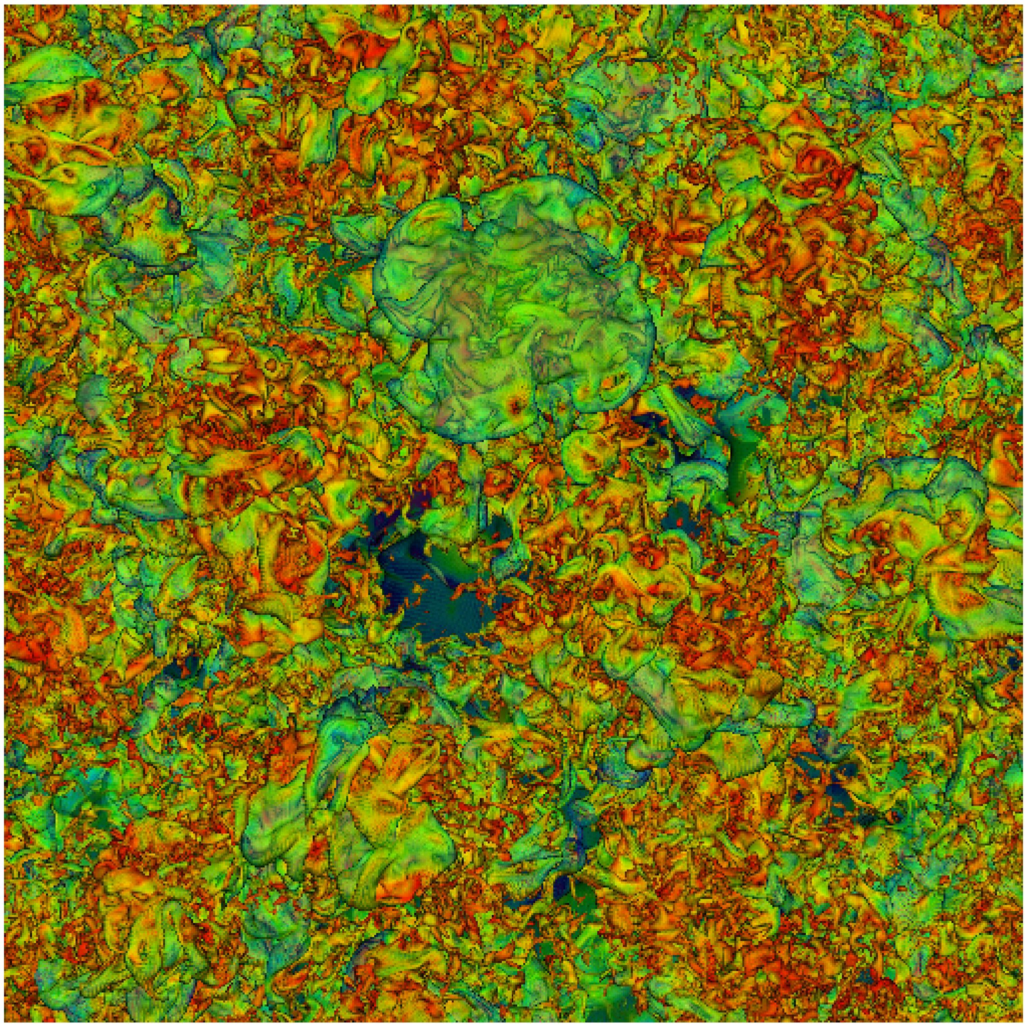

Figure B.4: DVR image rendered by ray-casting under close-view for RMI dataset at time step 250.

# Bibliography

[1] A. Aggarwal and J. S. Vitter, *The input/output complexity of sorting and related problems*, Communications of the ACM, 31(9), pp.1116–1127, 1988.

[2] L. Arge and J. S. Vitter, *Optimal dynamic interval management in external memory (extended abstract)*, IEEE Symposium on Foundations of Computer Science, pp. 560–569, 1996.

[3] C. L. Bajaj, V. Pascucci, D. Thompson and X. Zhang, *Parallel accelerated isocontouring for out-of-core visualization*, Proceedings of 1999 IEEE Parallel Vis. and Graphics Symp., pp. 97–104, 1999

[4] Y–J. Chiang and C. T. Silva, *I/O optimal isosurface extraction*, Proceedings IEEE Visualization, pp. 293–300, 1997.

[5] Y–J. Chiang, C. T. Silva and W. J. Schroeder, *Interactive out-of-core isosurface extraction*, Proceedings IEEE Visualization, pp. 167–174, 1998.

[6] Y. Chiang and C. Silva, *External memory techniques for isosurface extraction in scientific visualization*, External Memory Algorithms and Visualization, Vol. 50, pp. 247–277, DIMACS Book Series, American Mathematical Society, 1999

[7] Y–J. Chiang, R. Farias, C. Silva and B. Wei, *A unified infrastructure for parallel out-of-core isosurface and volume rendering of unstructured grids*, Proc. IEEE Symp. on parallel and large-data visualization and graphics, pp. 59–66, 2001

[8] P. Cignoni, C. Montani, D. Darti and R. Scopigno, *Optimal isosurface extraction from irregular volume data*, Proceedings of the 1996 symposium on Volume visualization, pp. 31–38 San Francisco, USA 1996.

[9] P. Ellsiepen, *Parallel isosurfacing in large unstructured datasets*, Visualization in scientific computing ' 95, pp. 9–23, Springer Verlag, 1995.

[10] C. Hansen and P. Hinker, *Massively parallel isosurface extraction*, Proc. IEEE Visualization, pp. 77–83, 1992.

[11] W. Hong, F. Qiu and A. Kaufman, *GPU-based object-order raycasting for large datasets*, EurographicsIEEE VGTC Workshop on Volume Graphics, pp. 177-186, 2005.

[12] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner and J. Klosowski, *Chromium: A stream processing framework for interactive rendering on clusters*, Proceedings of SIGGRAPH 2002, pp. 693–702, 2002.

[13] Y. Livnat, H. Shen, and C.R. Johnson, *A Near Optimal Isosurface Extraction Algorithm Using the Span Space*, IEEE Trans. Visualization and Computer Graphics, vol. 2, no. 1, pp. 73–84, March 1996.

[14] 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.

[15] K. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, *Parallel volume rendering using binary-swap image composition*, Computer Graphics and Application, vol. 14(4), pp. 59-68, 1994.

[16] S. Miguet and J. M. Nico, *A load-balanced parallel implementation of marching-cubes algorithm*, Proceedings of High performance computing Symp. '95, pp. 229–239, 1995.

[17] S. Molnar, M. Cox, D. Ellsworth and H. Fuchs, *A sorting classification of parallel rendering*, IEEE Computer Graphics and Applications, Vol. 14, No. 4, pp. 23–32, July 1994.

[18] 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.

[19] 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.

[20] H. W. Shen, C. D. Hansen, Y. Livnat and C. R. Johnson, *Isosurfacing in span space with utmost efficiency (ISSUE)*, IEEE Visualization'96, pp. 281–294, Oct 1996.

[21] C. Silva, Y. Chiang, J. El-Sana and P. Lindstrom, *Out-of-core algorithms for scientific visualization and computer graphics*, Visualization'02, Course Notes for Tutorial #4, 2002.

[22] P. M. Sutton and C. D. Hansen, *Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON)*, IEEE Visualization '99, IEEE Computer Society Press, pp. 147–154, Oct. 25–29 1999.

[23] 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.

[24] Q. Wang, J. JaJa and A. Varshney, *An Efficient and Scalable Parallel Algorithm for Out-of-core Isosurface Extraction and Rendering*, Journal of Parallel and Distributed Computing (JPDC), vol.6, no.5, pp. 592–603, May 2007.

[25] X. Zhang, C. L. Bajaj and W. Blanke, *Scalable isosurface visualization of massive datasets on cots clusters*, Proc. IEEE Symposuim on parallel and large-data visualization and graphics, pp. 51–58, 2001.

[26] 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.

[27] H. Zhang and T. S. Newman, *Efficient parallel out-of-core isosurface extraction*, Proc. IEEE Symposium on parallel and large-data visualization and graphics (PVG) '03, pp. 9–16, Oct. 2003.

[28] P. Cignoni, P. Marino, C. Montani, E. Puppo and R. Scopigno, *Speeding up isosurface extraction using interval trees*, IEEE Transactions on Visualization and Computer Graphics vol. 3, no. 2, pp. 158–170, April–June, 1997.

[29] D. E. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen, *Distributed Interactive Ray Tracing for Large Volume Visualization*, Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG), pp. 87–94, 2003.

[30] 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*, Proceedings of Computer Graphics and Imaging (CGIM), pp. 67–74, 2007.

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

[32] 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.

[33] 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.

[34] 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.

[35] Y. Livnat and C. Hansen, *View Dependent Isosurface Extraction*, Proceedings of the conference on IEEE Visualization 1998, pp. 175–180, 1998.

[36] 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.

[37] B. Mora, J. P. Jessel and R. Caubet, *Accelerating volume rendering with quantized voxels*, Proceedings of the 2000 IEEE symposium on Volume visualization, pp. 63–70, October, 2000.

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

[39] A. Reshetov, A. Soupikov and J. Hurley, *Multi-level ray tracing algorithm*, ACM Transactions on Graphics, Proceedings of ACM SIGGRAPH 2005, pp. 1176–1185, 2005.

[40] Q. M. Shi, J. JaJa, *Isosurface extraction and spatial filtering using persistent octree (POT)*, IEEE Transactions on Visualization and Computer Graphics vol. 12, no. 5, pp. 1283–1290, September, 2006.

[41] L. Sobierarjski and R. Avila, *A Hardware Acceleration Method for Volume Ray Tracing*, Proceedings of the 6th conference on IEEE Visualization 1995, pp. 27–34, 1995.

[42] 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. 562–572, 2005.

[43] R. Westermann and B. Sevenich, *Accelerated volume ray-casting using texture mapping*, Proceedings of the conference on IEEE Visualization 2001, pp. 271–278, 2001.

[44] S. E. Yoon and D. Manocha, *Cache-efficient layouts of bounding volume hierarchies*, EUROGRAPHICS 2006, vol. 25, no. 3, pp. 507–516, 2006.

[45] 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.

[46] J. Challinger, *Scalable Parallel Volume Raycasting for Nonrectilinear Computational Grids*, Proc. of Parallel Rendering Symposium93, pp. 81–88, 1993.

[47] 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.

[48] T. Foley and J. Sugerman, *KD-Tree Acceleration Structures for a GPU Raytracer*, In Proceedings of Graphics Hardware 2005

[49] A. Kaufman and K. Mueller, *Overview of Volume Rendering*, Chapter of Visualization Handbook, 2005

[50] G. Kindlmann, and J. Durkin, *Semi-automatic generation of transfer functions for direct volume rendering*, Symp. Volume Visualization98, pp. 79–86, 1998.

[51] 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.

[52] J. Kruger and R. Westermann, *Acceleration Techniques for GPU-based Volume Rendering*, IEEE Visualization 2003.

[53] P. Lacroute and M. Levoy, *Fast volume rendering using a shear-warp factorization of the viewing transformation*, Proc. SIGGRAPH 94, pp. 451–458, 1994.

[54] P. Lacroute, *Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization*, IEEE Parallel Rendering Symposium 95 Proceedings, pp. 15-22, 1995.

[55] M. Levoy, *Display of surfaces from volume data*, IEEE Comp. Graph. and Appl., vol. 8, no. 5, pp. 29–37, 1988.

[56] Marc Levoy, *Efficient ray tracing of volume data*, ACM Transactions on Graphics, 9(3): pp. 245–261, July 1990.

[57] K. Ma, *Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures*, Proc. of Parallel Rendering Symposium95, pp. 23–30, 1995.

[58] 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.

[59] 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

[60] N. Max, *Optical models for direct volume rendering*, IEEE Trans. Vis. and Comp. Graph., vol. 1, no. 2, pp. 99–108, 1995.

[61] 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.

[62] K. Mueller, N. Shareef, J. Huang, and R. Crawfis, *High quality 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.

[63] J. Nieh, and M. Levoy, *Volume Rendering on Scalable Shared- Memory MIMD Architectures*, Proc. of Volume Visualization Symposium, pp. 17–24, 1992.

[64] 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.

[65] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl and W. Strasser, *Smart Hardware-Accelerated Volume Rendering*, Proceedings of the symposium on Data visualisation 2003, pp. 231–238, 2003

[66] 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

[67] L. Sobierarjski and R. Avila, *A Hardware Acceleration Method for Volume Ray Tracing*, Proceedings of the 6th conference on IEEE Visualization 1995, pp. 27-34, 1995.

[68] K. Subramaniam and D. Fussel *Applying Space Subdivision Techniques to Volume Rendering*, In Proc. IEEE Visualization, pp. 150–158, Oct. 1990.

[69] Q. Wang and J. JaJa, *Intreactive High Resolution Isosurface Ray Casting on Multi-core Processors*, IEEE Visualization and Computer Graphics, Preprint, Dec 2007.

[70] 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. 397-400, 1999.

[71] M. Wan, A. Sadiq, and A. Kaufman, *Fast and Reliable Space Leaping for Interactive Volume Rendering*, In Proc. IEEE Visualization, pp. 195-202, Oct. 2002.

[72] L. Westover, *Footprint evaluation for volume rendering*, SIGGRAPH 90, pp. 367–376, 1990.