Published in final edited form as:
J Parallel Distrib Comput. 2009 January 1; 69(1): 39–53. doi:10.1016/j.jpdc.2008.07.006.

Scalable isosurface visualization of massive datasets on commodity off-the-shelf clusters

Xiaoyu Zhang[a] and **Chandrajit Bajaj**[b,*]

[a]Department of Computer Science, California State University San Marcos, San Marcos, CA 92096, United States

[b]Department of Computer Science, University of Texas at Austin, Austin, TX 78702, United States

Abstract

Tomographic imaging and computer simulations are increasingly yielding massive datasets. Interactive and exploratory visualizations have rapidly become indispensable tools to study large volumetric imaging and simulation data. Our scalable isosurface visualization framework on commodity off-the-shelf clusters is an end-to-end parallel and progressive platform, from initial data access to the final display. Interactive browsing of extracted isosurfaces is made possible by using parallel isosurface extraction, and rendering in conjunction with a new specialized piece of image compositing hardware called Metabuffer. In this paper, we focus on the back end scalability by introducing a fully parallel and out-of-core isosurface extraction algorithm. It achieves scalability by using both parallel and out-of-core processing and parallel disks. It statically partitions the volume data to parallel disks with a balanced workload spectrum, and builds I/O-optimal external interval trees to minimize the number of I/O operations of loading large data from disk. We also describe an isosurface compression scheme that is efficient for progress extraction, transmission and storage of isosurfaces.

Keywords

Parallel rendering; Metabuffer; Multi-resolution; Progressive mesh; Parallel and out-of-core isocontouring

1. Introduction

Tomographic imaging and computer simulations are increasingly yielding massive datasets. Interactive and exploratory visualizations have rapidly become indispensable tools to determine and browse regions of interest within volumetric imaging data, and verify and validate the results of computer simulations. One paradigm of exploratory visualization is to extract multiple 2-dimensional surfaces satisfying $w(\mathbf{x}) = const$ from a given scalar field $w(\mathbf{x})$, $\mathbf{x} \in \mathbf{R}^3$, and render it at interactive frame rate (~30 Hz). This interactive and exploratory visualization technique is popularly known as isocontour visualization.

Isocontour visualization for extremely large datasets poses challenging problems for both computation and rendering with guaranteed frame rates. First, large isosurfaces are to be extracted in a time-critical manner from those large datasets, whose sizes are from multi-

Correspondence to: Xiaoyu Zhang.

*Corresponding author. *E-mail addresses:* xiaoyu@csusm.edu (X. Zhang), bajaj@cs.utexas.edu (C. Bajaj)..

gigabytes to terabytes. As the size of the input data increases, isocontouring algorithms necessarily need to be executed out-of-core and/or on parallel machines for both efficiency and data accessibility. Second, the interactive aspect of the isocontour visualization demands that the scene is rendered quickly in order to provide responsive feedback to the user. In some cases, the detail allowed by a single high performance monitor may not be adequate for the resolution required. An even more common problem is that the dataset itself may be too large to store and render on a single machine. Third, the extracted isosurface may need to be transmitted from the computational servers to the rendering servers via the network, if the computational and rendering servers do not coexist on the same machines. It may also need to be saved on disk for future studies. Compact representation of the isosurfaces are thus used in order to meet the time limit of data transmission and save disk space.

We can think of the process of scalable time-critical isosurface visualization of massive data as a parallel and progressive stream from back to front as shown in Fig. 1 [61]. Due to the large size of the massive datasets, it is extremely time-consuming or even impossible to do isosurface extraction on a single processor. In order to scale to very large datasets, we use a computational back end consisting of both parallel processors and parallel disks. Large datasets are partitioned among the parallel processors in a load-balanced way and stored hierarchically on disks for efficient I/O access. Triangle streams of the isosurfaces are extracted at the back-end nodes progressively, which may be in compressed format, and sent to and rendered by the parallel rendering servers. All rendered images are then composited to display on a large multi-tiled screen.

Main results

Combining parallel and out-of-core computation, isosurface compression, and parallel rendering and compositing, we describe and implement a fully scalable system for interactive isosurface visualization across multiple isovalues and from different viewpoints. In this paper we highlight this framework, as shown in Fig. 2, and focus on the algorithms for back end parallel and out-of-core isosurface extraction and compression.

The rest of our paper is organized as follows: Section 2 discusses some background and related work; Section 3 describes the architecture of our framework for scalable isosurface visualization; Section 4.1 gives the details of our parallel and out-of-core isocontouring algorithm; Section 4.2 provides the details of our isosurface compression method and compares its results to that of another surface compression algorithm; Section 5 presents the performance of our parallel implementation on a COTS (Commodity Of The Shelf) cluster.

2. Related work

Parallel isocontouring

As the size of input data increases, isocontouring algorithms have to be extended to parallel machines for higher computational power and larger data addressability. Hansen and Hinker describe parallel methods for isosurface extraction on SIMD machines [25]. Ellsiepen describes a parallel isosurfacing method for FEM data by dynamically distributing working blocks to a number of connected workstations [20]. Shen, Hansen, Livnat and Johnson implement a parallel algorithm by partitioning load in the span space [51]. Parker et al. present a parallel isosurface rendering algorithm using ray tracing [44]. Walt et al. recently demonstrated interactive isosurface ray tracing on commodity desktop PCs by using advanced real-time ray tracing techniques to improve isosurface ray tracing performance [58]. Chiang and Silva give an implementation of out of-core isocontouring using the I/O optimal external interval tree on a single processor [11,12]. Bajaj et al. use range partition to reduce the size of

data that are loaded for given isocontour queries and balance the load within a range partition [5].

A very important issue of parallel computation is load balancing [17] that can be achieved mostly with two fundamental approaches: (i) static balancing, where the data is partitioned a priori with criteria that achieve load balancing at runtime [35], or (ii) dynamic balancing, where processors are given small chunks of data as they become available at run-time [20]. The partitions can take the shape of slices, shafts, or slabs [42]. A dynamic partitioning usually requires data redistribution at run time or data replication, which are expensive for massive datasets. The major concern about a static partitioning is that load balance may not always be good enough for the entire parameter space, i.e. the range of all possible isovalues in the case of isocontour visualization. Our results will show that balanced static data partitioning accrosss the entire parameter space is possible for large datasets.

Out-of-core isocontouring

Out-of-core computation becomes necessary as the data size exceeds the size of primary memory. The time of I/O operations may become the predominant factor in out-of-core isosurface extraction because of the large gap between memory and disk access speed. A simple sweep through all cells on a disk for each isocontour query is not a viable solution. We have to use external-memory search data structures that allow for loading only cells that intersect the isosurface. Finding all cell intervals whose ranges contain a query value ν in external memory is called the *stab query problem*, which is fundamental for out-of-core isocontouring. For a set of range intervals $S_I = \{I_i = [\min_i \max_i]: i = 1, 2 \ldots\}$, a stabbing query algorithm tries to find the subset $S_I(\nu) = \{I_j : I_j \in S_I, \min_j \leq \nu \leq \max_j\}$ in the least number of I/O operations and with linear storage space.

Several I/O optimal external data structures for the stabbing query problem have been studied recently. Kanellakis et al. [31] describe a data structure called meta-block tree that is an external-memory version of the priority search tree, where each interval [min, max] is mapped to a point (x, y) in the 2D range space with x = min and y = max. The meta-block tree has a height of $O(\log_B N)$, uses optimal $O\left(\frac{N}{B}\right)$ disk blocks, and performs a range query in an optimal number of $O\left(\log_B N + \frac{T}{B}\right)$ I/O operations, where N is the number of intervals and T is the size of output. Arge and Vitter [1] present an external interval tree data structure that is an external-memory version of the binary interval tree. Chiang and Silva [13] describe another I/O optimal external memory version of the interval tree, called binary-blocked I/O interval tree, for out-of-core isosurface extraction. Each internal node of a binary-blocked I/O interval tree has a branching factor of $B_f = \Theta(B)$ instead of $\Theta\left(\sqrt{B}\right)$ as in the external I/O interval tree [1]. Chiang et al. [12] manage to reduce the disk usage of a binary-blocked I/O interval tree, with a two-level indexing scheme that builds external search structure on clusters of cells, called meta-cells, instead of indexing directly on each cell. There is a tradeoff between the disk space and query time by varying the meta-cell size. They apply these external memory search structures for out-of-core isocontouring [11,12], with results indicating that isocontouring can be performed on data residing on secondary storage such that the I/O operations are not the limiting factor of the computation. While these external search structures are used for out-of-core isosurface extraction, all current methods have only considered the case of a single machine and a single disk. Single disk I/O would not be fast enough for tera-scale datasets even if an efficient external search structure is used. In order to achieve scalable performance for extracting isosurfaces from datasets of ever increasing size, we must combine the techniques of parallel computation and parallel I/O. While this paper focuses on extracting large isosurfaces from a single large volumetric data, Waters et al. recently presented an algorithm for out-of-core time-varying isosurface visualization [59]. This approach utilizes difference-

based compression techniques to minimize I/O bandwidth requirements for visualizing isosurfaces of data sets with many time steps.

Parallel graphics architecture

Since isosurfaces generated from large datasets are usually also very large, with hundreds of millions or even billions of triangles, it is beyond the rendering capability of a single graphics board. One important way to speed up the rendering process is to also use parallelism in polygon rendering. In general, most parallel rendering methods can be classified as sort-first, sort-middle, or sort-last, based on where data is sorted from object-space to image-space [37].

In the sort-first approach, the display space is broken into a number of non-overlapping display regions, which can vary in size and shape. Mueller [41] examines sort-first parallel rendering architecture and explores the issue of load balancing. Crocket [17] describes various considerations when building a parallel graphics library using a sort-first method. To load balance among rendering processors, small granularity partitions, such as interleaved scan-lines are used. However, many polygons are rendered multiple times for such small granularity partitions. Sort-first methods may suffer from load imbalance in both the geometric processing and rasterization, if polygons are not evenly distributed across the screen partitions, because polygons are assigned to the rendering process before geometric processing.

The sort-middle approach also partitions screen space among rendering processors, but it distributes transformed primitives instead of polygons to the rendering processors responsible for the portion of the screen into which these primitives fall. Since the distribution of polygons is not constrained by the partition of image-space, the rendering processors are usually well balanced in geometric processing. But it still suffers from load imbalance during the rasterization stage if primitives are not evenly distributed across the screen partitions. SGI Infinity reality graphics engine uses a sort-middle architecture internally in its graphics pipeline [39]. It uses 2-pixel wide stripes to ensure a good load balance of pixel work. However, it imposes a high cost in broadcasting transformed primitives among different pipes. Larger tiles have been used to reduce the broadcast overhead [29] at the cost of large reorder buffers.

The sort-last approach is also known as image composition. Each rendering server performs both geometric processing and rasterization, independent of other rendering processors. Local images rendered by the rendering server are blended together to form the final image. The sort-last method behaves particularly well with respect to load balancing, since polygons can be evenly distributed to any rendering process without constraint of screen partition. However, compositing hardware or a high bandwidth network is often needed to combine the output of the different processors into a single correct picture. Such approaches have been used since the 1960's in single-display systems [22,36], and more recent work includes [21,26]. Udeshi and Hansen [55] employ the multi-pipes of a SGI Onyx2 reality monster to render large isosurfaces in a sort-last fashion. They use a binary-swap method [34] to efficiently composite the images in $\log_2(P)$ steps, where P is the number of rendering processors. However, those custom-built parallel rendering machines are expensive, and do not easily expand as surfaces get larger.

Commodity parallel rendering

Many research groups recently studied the problem of using programmable PC graphics cards for parallel rendering [19,26,28,49,50]. Schneider [50] analyzes the suitability of PC workstations for parallel rendering by studying four different parallel polygon rendering scenarios: rendering single or multiple frames on a symmetric multiprocessors workstation, or a cluster of PCs. The Princeton University SHRIMP project [49] uses various load balancing schemes for a multi-projector rendering system driven by multiple PCs. Their method falls into

a sort-first category, since they try to break screen space into load-balanced pieces, and ship tiles of rendered pixels to the right display. Samanta et al. [48] later describe a hybrid sort-first and sort-last system in which sort-last pixel compositing overheads are significantly reduced by executing a dynamic sort-first primitive partition for each viewpoint. They further present a *k*-way replication approach [47] in which each 3D primitive of a large scene is replicated *k* times out of *n* PCs ($k \ll n$), in order to support geometric models larger than the main memory of any single PC.

Heirich and Moll [26] demonstrate how to build a scalable image composition system using off-the-shelf components, such as PC graphics cards, Field Programmable Gate Arrays (FPGAs), and gigabit network interfaces and switches, to composite rendered images from different machines by user-programmable associative and commutative combining operations. Lambeyda et al. [32] apply this architecture to interactive volumetric visualization of large rectilinear scalar fields. Moreland et al. [40] describe an inbox sort-last parallel rendering system on PC clusters to support a high-resolution display. They discuss techniques and algorithms to minimize the negative effects of large display resolution while maintaining good rendering performance on large datasets, by taking advantage of the spatial coherence in the data.

Our solution to the parallel image compositing problem is the Metabuffer, whose architecture is shown in Fig. 3. This is a sort-last multi-display image compositing system with several unique features, such as multi-resolution and antialiasing [8]. A similar project, though currently without stressing multi-resolution support, exists at Stanford University and is called Lightning-2 [27,52]. The Metabuffer is a custom hardware that supports a scalable number of PCs and an independently scalable number of displays—there is no *a priori* correspondence between the number of renderers and the number of displays to be used. It also allows any renderer to be responsible for any axis-aligned rectangular viewport within the global display space at each frame. Such viewports can be modified on a frame-by-frame basis, can overlap the boundaries of display tiles and each other arbitrarily, and can vary in size up to the size of the global display space. Thus each machine in the network is given equal access to all parts of the display space, and the overall screen is treated as a uniform display space, that is, as though it were driven via a single, large framebuffer, hence the name Metabuffer. More recently, Hewlett-Packard developed a parallel compositing library [43], a software API that facilitates the development of parallel rendering applications on commodity clusters.

Compression

Handling very large datasets benefits from, and sometimes even necessitates the use of compression techniques. We consider two aspects of compression in scientific visualization, working from compressed input data, and producing compressed output results. There are many benefits of applying compression techniques to input data: for instance, saving disk space, reducing data loading time, and satisfying main memory limit. Chen and Reif [10] describe doing computations directly on a compressed domain to speed up solving image processing problems, in particular for the splatting algorithm of volume rendering. Ihm et al. [30] give a wavelet-based 3D mesh compression scheme for volume rendering of visible human data. It loads the entire compressed volume in main memory and reconstructs voxel values at request during run-time. The same compression volume can also be used for isocontour visualization.

Isosurfaces extracted from volume datasets are rendered by rendering processors, which may differ from those computational processors that extract isosurfaces. In a client-server based data analysis system, it is important to transmit isosurfaces in compressed formats, in order to avoid the possible bottleneck of network communication. One possible approach to the isosurface compression problem is to first extract the isosurface into triangular meshes, and then apply to it one of the surface compression algorithms [7,14,18,24,53,54]. Although it is

conceptually simple, this method has several disadvantages. First the rendering servers have to wait until the computational servers finish both isosurface extraction and compression. The compression of the surface often takes a very long time, especially true for the very large surfaces extracted from large datasets, which contradicts the goal of using compression for real-time rendering. Furthermore, isosurfaces have the property that each vertex is an intersection point with one unique edge of the 3D volume. An algorithm designed specifically for an isosurface may get a better compression ratio. In this paper we will discuss a novel compression algorithm for isosurfaces extracted from 3D regular volumes that compresses the isosurface and allows streaming the compressed data to the rendering servers incrementally. The algorithm demonstrates a good compression ratio, can run incrementally and in parallel for large data sets, and may use compressed input volume data.

3. Framework

Fig. 2 illustrates the parallel end-to-end framework for scalable isosurface visualization on a commodity off-the-shelf cluster. It can be divided into a sequence of Pipelined Stages.

3.0.0.1. Pipelined stages

Triangle streams are generated by the back-end nodes by progressively extracting the isosurfaces. The triangle stream from the extraction node will be rendered by the middle parallel rendering servers. All rendered images will then be composited by the Metabuffer to display on a multi-tiled screen. The process of compositing images from multiple rendering servers to multiple displays using the Metabuffer is called parallel image composition. Given the required frame rate, the refresh time between two frames needs to be shared among these three stages: triangle extraction, rendering and image composition. While the image composition time taken by the Metabuffer is constant, how much external time left in the frame interval determines what resolution of triangles will be extracted and rendered.

The parallel back-side provides multi-resolution isosurfaces extracted from the volume datasets to satisfy the time limit. This stage is governed by the parallel and progressive triangle extraction algorithms. Producing multi-resolution representation of the data at the back-side is essential for the time-critical rendering of massive data. When the user changes viewing parameters frequently, coarser representations of the data are rendered in order to give the user responsive feedback. Only when the user chooses a certain viewing position and some interesting isovalue, are the details of the progressive mesh or isocontour streamed for rendering in order to produce higher resolution images. To reduce the time of data transmission over the network, the extracted mesh may be communicated to rendering servers in compressed format. Although the rendering servers might be on the same set of machines as the isocontour extraction processes, the parallel and progressive triangular mesh extraction process can in general scale independently of the number of parallel rendering servers. The triangle streams between the extraction and rendering servers can be transmitted progressively and in a compressed format.

3.0.0.2. The metabuffer

One novel feature of the framework is the parallel rendering and image composition system that is able to render the given scene in the least latency. The parallel front-side is built around the Metabuffer [8], which is custom hardware built from commodity PC components. The Metabuffer hardware provides several unique advantages to assist in rendering a large surface in parallel, such as arbitrarily located and overlapped viewports and multi-resolution. Each rendering server in Fig. 2 is mapped to a viewport on the screen space. A very important problem in parallel rendering is how to position those viewports and partition the mesh such that each rendering server has approximately an equal amount of work.

The Metabuffer allows the number of rendering servers to scale independently from the the number of display tiles. Since the Metabuffer allows the viewports to be located anywhere within the total display space and overlap each other, it is possible to achieve a much higher degree of load balancing. Since the viewports can vary in size, the system supports multi-resolution rendering, for instance allowing a single machine to render a background at low resolution, while other machines render foreground objects at much higher resolution.

Given the progressivity from the triangle extraction phase, to the final image composition phase in the framework, and the fact that each phase is fully parallelizable, we can expect to have a truly scalable isocontour visualization system for massive datasets.

4. Parallel and out-of-core algorithms

In this section we will discuss in more detail the parallel and out-of-core isocontouring algorithm and the isosurface compression algorithm mentioned in Section 3 that enables the framework for scalable time-critical visualization of massive datasets. First we discuss a distributed memory out-of-core/parallel isocontouring scheme that tries to take full advantage of the available hardware.

4.1. Scalable isosurface extraction

A scalable data analysis and visualization application must take data processing, I/O, network and rendering all into consideration. Specifically for the isocontour extraction of large volume data, it should have load balanced parallel computation for fast surface extraction, out-of-core computation to scale to datasets bigger than the size of the total main memory, and parallel I/O to avoid the bottleneck of accessing massive data on disk.

We can model such a system that combines parallel processing and parallel disk access with a model called the BSP-Disk model. The BSP-Disk model consists of P interconnected processors, each of which may have a local memory and disk. The BSP-Disk model combines the features of the BSP parallel computation model [56] and the PDM [57] parallel disk model. It is a distributed memory parallel computation model, while each processor can access its local disk in parallel. The BSP-Disk model very closely describes the characteristics of a PC cluster, where each node has its own CPU, local memory and local disk. The parameters of the BSP-Disk model are as follows:

- N: Total number of atomic units of the problem.
- M: Total number of atomic units that can fit in the one processor's main memory.
- P: Number of processors.
- D: Number of Disks.
- B: Number of atomic units that fits in one disk block.
- A: Time to read or write one disk block on the local disk.
- L: minimal synchronization time of the BSP model.
- g: gap parameter of the BSP model which characterizes the communication bandwidth.

In the case of large datasets, we have the design conditions $N > P \cdot M$ and $M \gg B$. In contrast to the PDM model where a single processor has equal access to all parallel disks, the disks in the BSP-Disk model are associated to different processors as their local disks. One very important example is that every processor has one associated local disk ($P = D$), which is often the case for a PC cluster. More generally $\frac{P}{D}$ processors can be assigned to every disk. Extra

communication time is required when one processor needs to access the data on a remote disk. In the BSP-Disk model one disk block can be loaded from every disk into memory in one parallel I/O, because each disk can be accessed independently. Thus up to D disk blocks can be read into main memory in one parallel I/O. In other words, the data access time is shared among the D disks.

The algorithms designed for the BSP-Disk model would consider all three parts of the time for a real parallel and out-of-core algorithm, local computation, local disk I/O, and communication. Therefore the time of the isosurface extraction can be written as

$$T=\max_P (T_w+T_{io}+T_c) \le \max_P (T_w)+\max_P (T_{io})+\max_P (T_c), \quad (1)$$

where T_w is the time for local computation, T_{io} is the time for local disk access and T_c is the time for communication. T_w and T_c are to be measured using the BSP model and T_{io} is measured as $A \times N_d$, where N_d is the number of parallel I/O operations. The objective of our isocontouring algorithm for large datasets is to speedup the computation by distributing the load to multiple processors, minimize the number of parallel I/Os, and minimize inter-processor communication (such as the remote disk accesses). These factors do not always play together for each other. We must make tradeoffs according to the real system parameters. To minimize the number of parallel I/O's, we need to load only those portions of dataset that contribute to the final result, and distribute data to the disks such that data is loaded evenly from the D local disks. Minimizing the local computation time requires good load balance among the processors. Finally minimizing communication means minimal data redistribution, and remote data access during computation.

An optimal algorithm for the PDM model requires optimal total disk storage space $O(N)$ and optimal number of parallel I/O operations $O\left(\frac{K}{D}\right)$, where K is the number of disk blocks that need to be loaded. A scalable algorithm for the BSP-Disk model should also have optimal disk storage space $O(N)$ because some datasets may have a size much larger than the size of a single disk. In other words, one cannot afford to replicate the entire datasets on each local disk. Instead, data assignment of large datasets should have a constant *replication factor*, which the maximum number of times that any atomic data element is duplicated on those disks. Our experimental results will demonstrate that for large datasets, even a distribution of replication factor 1 gives very good load balance and leaves little space for optimization with larger replication factors.

Isosurface extraction has the property that computation on each sub-volume of a dataset can progress independently. Thus dynamic load redistribution during the run time would require the transmission of data blocks from overloaded processors to idle processors, if one uses the minimum replication factor 1. Large amount of data then have to be communicated. On the other hand, a PC cluster has relatively low bandwidth and thus a large gap factor g, making communication the least scalable factor when data size becomes larger. These considerations lead us to choose the static data partition method for isocontour visualization in the BSP-Disk model. At the preprocessing stage, we partition the data among multiple computational nodes, and build the external interval tree as the index structure for the blocks on each disk. Here we have assumed each node has one processor and an associated local disk.

4.1.1. Data partition methods—In order to achieve good performance for a static workload allocation method for parallel computation, data must be partitioned carefully such that each processor has approximately the same amount of work. Furthermore data must be distributed among the D disks such that the number of parallel I/O is minimal. The work load of isocontouring computation on a processor is typically proportional to the size of its output.

In this section we discuss the preprocessing steps to prepare data for parallel and out-of-core isocontouring. Without loss of generality, we assume the dataset is originally stored on the *D* disks in such a way that each disk has an equal size slab of the volume. Because a 3D volume can be thought as a stack of 2D images, this is a simple and natural distribution. However, this distribution is far away from good load balancing. For an isocontour query, a big portion of the isosurface may fall into a single slab of the volume, which in turn causes one processor to have excessive computations and I/O operations, while others are idle. The data must be redistributed with replication factor 1 in such a way that each processor has approximately an equal amount of work for any given set of isocontour queries.

Cell is the minimum unit of the volume dataset. Function values are defined on the *vertices* of the cells, and usually tri-linearly interpolated inside cell. Ideally we can use the granularity of cell for the data partition and build the external interval tree for all the cells. However as shown in [12], it is very storage inefficient to build an external index data structure using the unit of cell, because of the high overhead of data duplication among cells. Instead, we use the atomic unit of block, which is usually a 3D rectangular slab of adjacent cells, as shown in Fig. 4. Although some extra cells maybe be loaded because of the larger granularity, block provides the possibility of tradeoff between disk space and I/O efficiency. In our implementation, we choose the size of block as the disk block size, such that one block can be loaded in one I/O operation. Since we use a block as the unit of data partition and accessing, it is possible to extract isosurface progressively at different resolution, to give user the basic shape of the surface with minimum delay.

The basic idea of balanced data partitioning is to consider a workload diagram as in Fig. 5(a), where the workload of an algorithm is plotted as a function of the parameter *p*. From such a diagram one can estimate the cost of isosurface extraction on the dataset *D* for different isovalues of *p*. The analysis can be performed at the level of single cells of *D*, so that it is possible to determine which cells are involved in the evaluation for a given parameter *p*. It is more convenient to conduct the analysis on the level of blocks, because blocks are the units of data distribution. The diagram of *D* is the sum of the sub-diagrams of its blocks. One key issue is how to construct a workload diagram. The ideal workload diagram would be the time taken to extract isosurfaces, which however is difficult to compute and depends highly on the environment. One can approximate the workload diagram with the contour spectrums [4,5], for instance the number of triangles in an isosurface or the area of an isosurface, etc. In the following discussion the workload histogram is approximated by the number of triangles in the isosurfaces. This is a piece-wise constant function that can be computed in constant time for each cell. It will be shown in later examples that the contour spectrum of triangle count, closely matches the diagram of actual isosurface extraction time. Fig. 5(b) shows the histogram for a real dataset, CT scan of the visible human male.

4.1.1.1. Optimal load balancing: The analysis of the isocontouring workload diagram allows immediate evaluation of the quality of a data partitioning scheme, in terms of load balancing during parallel computations. In fact the ideal load balancing for *P* processors would be achieved if the workload histogram of the data assigned to each processor is the same-scaled version ($\frac{1}{P}$ times) of the global histogram. Fig. 6 shows the ideal partitioning of a dataset *D* in the case of two processors where the units *b*, *c*, *d* are assigned to the first processor and the unit *a* is assigned to the second processor. In this ideal data partition, each processor does the same amount of work for every value of the parameter *p*.

One can evaluate the quality of any given data partitioning scheme by comparing its diagram with the ideal one. In the following sections we propose a data partition scheme that tries to stay close to the ideal case for a good load balance.

In order to minimize disk access at execution of out-of-core isocontouring, one needs to minimize the number of unused blocks that are loaded into main memory. Ideally, only the blocks that contribute to the final isosurface of the current value p are loaded from secondary memory into primary memory. Only the blocks in the shaded area of Fig. 7 intersect the isosurface of the isovalue p. Ideally, only those blocks should be loaded into main memory for extracting the isosurface.

Finding the optimized data distribution that minimizes the difference among the workload histograms of the P processors is a NP-hard problem. This is a stronger version of the optimized bin-packing problem [15], which is itself NP-hard. Therefore, one has to use an approximation algorithm for the data distribution of parallel and out-of-core isocontouring. Next we present both a deterministic greedy algorithm, and a randomized algorithm.

4.1.1.2. Greedy data decomposition algorithm: First we describe a greedy data decomposition algorithm that achieves reasonably good results in experiments. A triangular matrix in the range space as shown in Fig. 8 is constructed to help the data redistribution. The function value range $[V_{\min}, V_{\max}$ of a dataset D is divided into n segments $[V_0,V_1]$, $[V_1, V_2]$, …, $[V_{n-1}, V_n]$ with $V_0=V_{\min}, V_n = V_{\max}$. Therefore the triangular range space of D is divided into a triangular matrix of buckets. Every data block falls into one of the buckets according to its functional range. For instance, if a block has range $[x,y]$ and the bucket range interval size is a, the block would belong to the $\left[\lceil\frac{(x-\min)}{a}\rceil, \lceil\frac{(y-x)}{a}\rceil\right]$ matrix element.

Blocks falling into the same matrix element have a similar span in the range space. Thus blocks are categorized according to which triangular matrix element they fall into. Here we focus on distributing blocks in a matrix element e, and then we apply the same algorithm to blocks in any other element. Each processor initially has a collection of blocks for every matrix element before the data redistribution. The greedy algorithm is as follows:

1. Every processor sorts its collection of blocks in an element e by the decreasing order of range size.
2. Those blocks are then assigned to the P processors in a round-robin fashion. If processor p_i has a sorted list of blocks $(b_1, b_2, \ldots, b_n)$ in the matrix element e, b_1 is assigned to processor p_i, b_2 is assigned to processor p_{i+1} and b_j is assigned to processor $p_{(i+j-1) \bmod p}$.
3. Each processor collects blocks to be sent to other processors and transmits them to the destinations in rounds.

This greedy algorithm tries to ensure the each processor gets about $\frac{1}{P}$ of the blocks with similar range spans and sizes. Thus the number of blocks loaded by each processor is approximately the same. Experimental results in Section 5 verify this assumption.

4.1.1.3. Randomized data decomposition algorithm: For a large volume dataset, one has a large number of blocks to distribute among the P processors ($N \gg P$). Suppose each block b_i, $1 \le i \le N$ is just randomly assigned to one processor. For $i = 1, \ldots, N$, let x_i^j be a binary random variable which is 1 if the block b_i is assigned to the processor p_j, $1 \le j \le P$, and be 0 otherwise. Then $x_1^j, x_2^j, \ldots, x_N^j$ is a sequence of *independent Bernoulli trials* with $\mathbf{prod}\left(x_i^j=1\right)=1/P$. In average every processor p_j receives $\mathbf{E}\left(\sum_{i=1}^{N} x_i^j\right)=\frac{N}{P}$ blocks.

Let $\widehat{w}=\max_N(w_i)$, where w_i is the workload of the block b_i for a given isovalue ν. Then $W=\sum_{i=1}^{N} w_i$ is the total work of extracting the isosurface of isovalue ν. Let $a_i=w_i/\widehat{w};\quad a_i \in (0,1]$ be the normalized workload of the block b_i Then the weighted sum of Bernoulli trials $\Psi_j=\sum_{i=1}^{N} a_i x_i^j$ is the normalized workload assigned to the processor p_j. The expected workload assigned to the processor p_j is

$$\widehat{w}\times \mathbf{E}\left(\Psi_j\right)=\sum_{i=1}^{N}\frac{\widehat{w}a_i}{P}=\sum_{i=1}^{N}\frac{w_i}{P}=\frac{W}{P}. \tag{2}$$

The randomized algorithm is rather simple with the following steps:

1. Each processor determines the destination processors of its blocks randomly with an even probability 1/*P*.
2. Each processor collects blocks to be sent to other processors and transmits them to their destinations in rounds.

Let W_j be the workload assigned to the processor p_j for a given isovalue ν. It is possible to show that it is highly unlikely that W_j greatly exceeds the expected value *W/P*. We apply the following theorem by Raghavan and Spencer [45], which gives an inequality for the weighted sum of Bernoulli trials.

Theorem 1 ([45]). *Let* $a_1, \ldots, a_N$ *be reals in* (0, 1]. *Let* $x_1, \ldots, x_N$ *be independent Bernoulli trials with* $\mathbf{E}(x_i)=\rho_i$. *Let* $\Psi_\beta=\sum_{i=1}^{N} a_i x_i$. *If* $\mathbf{E}(\psi_\beta)>0$, *then for any* $\upsilon>0$

$$\mathbf{prob}\left(\Psi_\beta>(1+\nu)\,\mathbf{E}\left(\Psi_\beta\right)\right)<\left(\frac{e^{\nu}}{(1+\nu)^{(1+\nu)}}\right)^{\mathbf{E}(\Psi_\beta)}. \tag{3}$$

Using the above theorem, we can obtain following result.

Theorem 2. If the total workload W > $\alpha_{\widehat{W}}P$ log r for a given isovalue and blocks are randomly allocated among P processors, any processor has a workload $\leq \frac{2W}{P}$ with high probability $\geq (1-\frac{1}{r^c})$, where $\widehat{w}$ is the maximum workload of a block and $\alpha > \frac{c+1}{\log 4/e}\ (c \leq 0)$ is a constant.

Proof This proof is similar to the result in [23] on random emulation of *QSM* on BSP model. The following is based on a simple analysis in [46].

In the case of data partitioning for parallel isocontouring, assume the normalized workload $W'=\sum_{i=1}^{N} a_i > \alpha P \log r$, where $\alpha > \frac{c+1}{\log 4/e}\ (c>0)$ is a constant and $r \geq P$. Since $N \gg P$, this assumption usually holds except for isosurfaces that have very few triangles, which are of little interest, because they can be quickly extracted, regardless of how the dataset is partitioned. Then the probability of the processor p_j, having more than twice the average workload, is

$$\mathbf{prob}\left(\Psi_j>\frac{2W'}{P}\right)<\left(\frac{e}{4}\right)^{W'/P}.$$

Let *d* = 4/e > 1, then

$$\mathbf{prob}\left(\Psi_j > \frac{2W'}{P}\right) < d^{-W'/P} < d^{-\alpha \log r} = d^{\frac{-\alpha \log d \log r}{\log d}} = r^{-\alpha \log d} = r^{-c-1}.$$

The probability of any processor having more than twice the average workload is

$$\mathbf{prob}\left(\Psi > \frac{2W'}{P}\right) \leq P \times \mathbf{prob}\left(\Psi_j > \frac{2W'}{P}\right) < r^{-c}, \tag{4}$$

where $\psi = \max_P(\psi_j)$ is the maximum load of all processors. Therefore, we have shown that the randomized data partitioning for parallel and out-of-core isocontouring achieves good load balance for any isovalue with high probability.

After data is partitioned, an I/O-optimal interval tree [1] is built as index structure for the data on each disk in a way similar to [12]. The external memory interval tree has optimal $O(\log N + K)$ I/O operations for stabbing queries, where K is the number of active cells. Therefore the preprocessing steps can be summarized as follows.

1. **Break the data into blocks**. At the beginning of the data distribution, there is an initial partition of the volume data among the disks, which is however not load balanced for isosurface queries. For instance, one can just break the big volume into slabs, so each disk contains one slab of the data. First we divide the slabs into blocks, each of which is in the same order as the disk block size. With each block, we store all the information necessary for performing isocontour extraction on the block, including the function values on all its vertices and the geometric information such as the dimensions, origin, and span of the block. We also store the index and range interval of the block explicitly for later construction of external interval trees.
2. **Redistribution of data blocks**. For the data block redistribution, every processor runs the same algorithm to determine to which processors it needs to send blocks. Every processor p_i then in turn communicates to the other processors $p_j (j \neq i$ the number of blocks to be sent from p_i to p_j and the actual blocks. A deterministic or random method can be used to ensure each processor has approximately the same number of blocks and workload histogram.
3. Build external interval trees as the search data structures. During block redistribution, every block assigned to a processor is given a unique ID and written to a single file that contains all the blocks belonging to the processor. Every block is stored starting from a disk block boundary, and its location is easily decided from its ID. While a block is written to the block file, its associated range interval and ID are stored in an intermediate file, which will be processed later to build the external memory search data structure that points to the data blocks. Since even those index structures may not fit into main memory as the data size increases, we implement the external interval tree [1] for the full scalability of our algorithm. An external interval tree is built independently on each local disk with the intermediate file for blocks assigned to the disk. The implementation details of building an external interval tree are referred to [1,11].

4.1.2. Isocontour query processing—When an isovalue query comes in, each processor independently searches the external interval tree on its local disk to find all blocks whose ranges intersect the isovalue. Solving this stabbing query problem with an external interval tree requires optimal $O(T/B + \log_B N)$ I/O operations [1], where T is the number of active blocks.

If a block *b* is found to intersect the isosurface, the processor loads *b* from its local disk and extracts polygons from *b*. The extracted polygons are either written back to the local disk or rendered by graphics hardware. The pseudo-code for parallel and out-of-core isocontouring is shown in Fig. 9.

Isosurfaces can be extracted in compressed format, as we will describe in Section 4.2. To give the user quick responses, the extracted surface is streamed to the parallel rendering servers which may reside on the same machines. The rendered images from the parallel rendering engines are then composited by the Metabuffer. Since we use the block as the atomic unit for data distribution and isosurface extraction, isosurfaces can be extracted and rendered at different resolutions to meet the time limit. One can get a fast overview of the isosurface shape, before choosing to extract and render an interesting isosurface in more detail. Fig. 10 shows an isosurface (isovalue = 1200) of the visible human male MRI data that is extracted and rendered at three different resolutions.

This algorithm provides a load balanced and fully out-of-core method for isocontour extraction, which would be scalable to arbitrary large datasets. The extracted surfaces are rendered by the parallel rendering servers, and finally composited by the Metabuffer. Since the data partition gives approximately equal workload for every processor, each processor generates approximately an equal number of triangles and thus has a balanced rendering load as well. The surface extraction process and rending process can be run in parallel. Fig. 11 shows one isosurface extracted and rendered by eight processors, where data blocks are distributed randomly among the processors. Depending on the size of the main memory, one can allocate a portion of main memory as a block cache in order to reduce the number of I/O operations of loading blocks for different isovalues. When one processor needs to load a block to extract an isosurface, it first looks it up in the cache. If the block is in the cache, a pointer to the block is handed to the program and no disk I/O operation will be issued. If one uses the seed propagation method [3] for isosurface extraction in a block, the seed set and corresponding interval tree are also cached with the block. If the total cache size is large enough, the algorithm naturally transits to parallel and in-core computations. While the data size is much greater than the main memory size, our parallel and out-of-core algorithm is still very efficient with a minimum number of I/O operations.

4.2. Isosurface compression

In the visualization framework shown in Fig. 2, isosurfaces extracted on the computational nodes need to be rendered by the rendering servers and composed by the Metabuffer. The computational processes and rendering processes might not be present on the same machines. Therefore isosurfaces need to be transmitted from the back-end to rendering processes via the network. Compact representation of the isosurface should be used in the transmission in order to meet the time limit. Compact representations of the isosurfaces should be used in order to meet the time limit of display in a slow network. Furthermore, we often want to store those large isosurfaces on disk in order to examine them later from different viewing points. Compressed file format significantly reduces the required disk space for storing those isosurfaces.

In a real-time visualization environment, we want a compression method that can output a compressed stream of an isosurface to the rendering processes while the isosurface is being extracted. The parallelism between the rendering process and the computational process gives the user faster response time. An isosurface *S* extracted from regular 3D volume has a special property that differentiates it from an arbitrary triangular surface. It is embedded in the 3D mesh. Every vertex of *S* is an intersection point on an edge of the regular 3D volume. Instead of directly storing the (*x*, *y*, *z*) coordinates of a vertex, the coordinates can be easily reconstructed from the edge index and the function values on the two endpoints of the edge.

Isosurfaces are usually represented as triangular meshes of vertex and face indices. Such representation ignores the 3D embedding of these isosurfaces and is very inefficient. In this section we describe an isosurface compression scheme called *edge index encoding*, which is easy to compute and has several benefits over general-purpose surface compression algorithms.

Given an isovalue p and a regular 3D volume $\mathbf{V}$ in $\mathbf{R}^3$, the isosurface can be generated by the Marching Cubes method [33] that traverses every cell and computes its portion of the isosurface. According to the function values at the 8 vertices of a cell greater or less than the isovalue p, the topology of the isosurface inside the cell is determined as one of the $2^8 = 256$ possible configurations, whose number can be further reduced by rotational symmetry and vertex complement [33,38]. From the index of a cell and its configuration, we can easily derive what edges of the cell are intersected by the isosurface. A cell intersected by the isosurface is called a *valid cell*. The exact coordinates of the intersection point on an edge can be computed from the edge index, and the function values on the two endpoints of the intersected edge. A vertex that is one end point of an edge intersected by the isosurface is called a *relevant vertex*. Hence the representation of the isosurface can be reduced as encoding the configurations of valid cells and the function values on the relevant vertices. We further notice that a cell configuration can be determined, if we know which vertices of its 8 corners are relevant vertices. Therefore, to reconstruct the isosurface, it is only necessary to have the indices and function values of the relevant vertices and the indices of the valid cells.

Considering a 3D volume data of dimension $n_1 \times n_2 \times n_3$, one can think of it as a stack of n_2 slices, without loss of generality. The extraction of isosurface progresses along the layers between two adjacent slices. Only the data of two adjacent slices is needed in the main memory at any time, in order to compute the isosurface. Instead of computing the actual triangulation, we mark the relevant vertices on the slices and the valid cells in the layer. Fig. 12 shows an example of marked relevant vertices and valid cells for a 2D regular dataset. After a layer is finished, its first bounding slice is no longer needed, while its second slice becomes the first bounding slice of the next layer. One now can purge the first slice out of main memory, encode the relevant vertices of the slice and the valid cells of the layer, and send the encoded data to the rendering process. As soon as the rendering process receives the data of a layer and its bounding slices, it can then start to decode the triangulation of the isosurface within the layer and render them with the graphics pipe.

The steps of the encoding algorithm for isosurface extraction are:

1. Read the first slice into the main memory.
2. For each layer in the volume
    - Read the next slice into the main memory as the second slice.
    - March through all the cells in the layer. For every vertex on a slice, a bit r is set to 1 if it is a relevant vertex, $r = 0$ otherwise. Similarly for each cell in a layer between two adjacent slices, a bit i is set to 1 if it is a valid cell, $i = 0$ otherwise.
    - Encode the indices of the relevant vertices of the first slice using an entropy encoding method, such as adaptive run-length coding [9] or arithmetic coding [60].
    - Encode the quantized function values of the relevant vertices on the first slice with their second-order difference.
    - Encode the indices of the valid cells using entropy encoding.
    - Output the encoded data.

- Remove the first slice from the memory and set the second slice as the first.

3. Encode the last slice and output the encoded data.

The steps of the decoding process at the client side are the reverse of the encoding process. As soon as it receives the data for a layer and its bounding slices, the decoding process decodes the data and decides triangulations for the valid cells. The steps are as follows:

1. For each layer of the volume
    a. Receive the encoded indices of the valid cells in the layer and the encoded indices and function values of the relevant vertices on its bounding slices.
    b. Decode the data.
    c. For every valid cell in the layer
        - Find its configuration from the 8 vertices.
        - Compute the intersection points on its edges with linear interpolation.
        - Get the triangulation inside the cell from the Marching Cubes table.
    d. Render the triangles in the layer.

We tried the adaptive arithmetic coder [60] and adaptive Z-Coder [9] as the encoding method. In our implementation of isosurface compression, the Z-Coder runs faster than the arithmetic coder, and achieves similar compression ratios. There are many other entropy coders available. Any good coder should work as well for our purpose of encoding the indices and function values.

4.2.0.1. Index coding—Since we mark the relevant vertices on a slice with 1 and non-relevant vertices with 0, we can treat the slice as a bitmap of 0's and 1's. Similarly a layer can also be considered as a bitmap. For an ordinary isosurface, majority bits of the bitmaps are 0's. We encode the bitmap with the arithmetic coder or Z-Coder, to take advantage of the coherent pattern of relevant vertices and valid cells. There are many ways to convert a 2-dimensional bitmap into a bit sequence. However, our compression results suggest that the size variations among different conversion methods are insignificant. Therefore we just choose the simple scan-line conversion.

4.2.0.2. Function value coding—In addition to the indices of the relevant vertices and valid cells, we need to encode the function values on the relevant vertices. For the relevant vertices on each slice, we

- directly encode the value of the first relevant vertex.
- predictively encode the value differences of the rest relevant vertices.

This encoding process includes normalization, quantization, prediction, and entropy coding. Predictive coding is widely used in the geometry coding of the algorithms for surface compression [7,16]. A predictor combines the values of previously coded neighboring vertices, in order to construct a prediction of the current vertex value. The correction, or the difference between the actual value and its prediction, is usually less varied and more suitable for entropy encoding.

If the function values are floating point numbers, we first normalize them to the range of [-0.5, 0.5]. For a sequence of floating point values $x_0, \ldots, x_n$, the encoding scheme is as follows,

1. Encode the quantization of the first value x_0 and its sign relative to the isovalue p.

   $\tilde{x}_0$ = the quantized value of x_0.

2. For $i = 0, \ldots, n - 1$

   - Encode $\Delta_i = x_{i+1} - \tilde{x}_i$ using the given quantization bits and the sign of x_{i+1} relative to the isovalue p.
   - $\tilde{x}_{i+1} = \tilde{x}_i + \tilde{\Delta}_i$ where $\tilde{\Delta}_i$ is the quantized value of Δ_i.

Here we use the correction $\Delta_i = x_{i+1} - \tilde{x}$ instead of $x_i + 1 - x_i$ to prevent the accumulation of quantization errors [6]. We use an extra bit for every relevant vertex, to indicate the sign of its function value x relative to the isovalue p because the error introduced in the quantization may change x from $x < p$ to $\tilde{x} > p$ or from $x > p$ to $\tilde{x} < p$ if x is close enough to p. Such sign flip of x is very undesirable because it may cause errors in determining the configurations of the valid cells, thus changing the topology of the decoded isosurface. We pay the extra cost of one sign bit per value to avoid such topological inconsistency. When the decoder decompresses the vertex values, it checks if the sign of each value has been changed against the isovalue. If a value has been flipped, it shifts it by one quantization unit to the left or the right to recover its correct sign. Although the extra sign bit increases the size of compressed isosurfaces by a small fraction, it is small compared to the > 10 bits usually used for quantization. It has the advantage of numerical robustness even if fewer quantization bits are used, as shown in Fig. 13. While the surface becomes more bumpy with less quantization bits, the connectivity of the isosurface has been preserved correctly. For fixed-precision data types such as 16-bit short or 8-bit byte, the function value encoding can be lossless with a better compression ratio than the lossy general-purpose surface compression algorithms. Fewer quantization bits certainly can also be applied to the fix precision data types, if a smaller size surface file is required. Lossless compression is just one more benefit of the edge index encoding method that takes advantage of the embedding 3D volume.

4.2.0.3. Attribute coding—There is often more than one variable defined on the vertices of a 3D volume. A very important example is the 3D texture. Another example is simulations of multiple physical functions on a common 3D grid. It is very instructive to view the distribution of one function on the isosurfaces of another function [2]. The visualization of attributes on an isosurface is done by computing the attribute values on the vertices of the isosurface, and coloring the isosurface with the attribute values. The encoding of such attributes on isosurfaces is straightforward with our scheme. For the attribute variable, it is still true that only values on the relevant vertices are necessary to reconstruct the attribute values on the vertices of the isosurface. Thus we only need to apply the same predictive difference coding method to encode the attribute values, without adding any cost to encode the indices of intersected edges.

This method has yielded a very good compression ratio to isosurface of regular 3D mesh compared with the general purpose triangular surface compression algorithms as shown in Table 1, where the general algorithm is that of [7]. Furthermore, the edge index method has the advantage of incremental encoding and decoding, because two slices and one layer are necessary in the main memory at any time for the encoding and decoding processes. Thus both the compression and decompression of the isosurface using edge index encoding need only a small amount of main memory, and the reconstruction of the surface can start far before the whole surface transmission is finished. Furthermore the encoding of indices and function values is done during isosurface extraction, such that no expensive post-extraction compression process is necessary.

5. Implementation and results

In this section we present the implementation details, and some experimental results of our parallel and out-of-core isocontouring algorithm, and the scalable isocontour visualization framework. The primary interest in these tests is to show the scalability of our algorithm with the size of datasets and the number of machines.

Our implementation platform is a commodity off the-shelf PC cluster of Compaq SP750 workstations. Each node has an 800 MHz Pentium III processor with 256 MB of Rambus memory, a 9 GB system disk and a 18 GB data disk, and an nVdia Geforce II graphics card. These machines are all connected through a 100 Mb/s ethernet, while portions of the cluster are also connected by the Servernet II and Gigabit Ethernet. All our tests use the 100 Mb/s ethernet for inter-processor communication. All nodes run Debian Linux (kernel 2.2.18) as the operating system, and the disk block size is 4096 bytes. In the following tests, a cluster node renders its own portion of extracted isosurfaces, without redistribution of isosurfaces across the network.

We use several datasets of varied size, as shown in Table 2, and on a different number of machines to test the performance of our parallel and out-of-core isocontouring algorithm.

Those test datasets, whose sizes range from 134 MB to more than 16 GB, are from gas hydrodynamic simulations and the visible human project from the National Library of Medicine. Some of those datasets are too large for a single PC to handle in its main memory. Depending on the data size, the pre-processing step takes time from several minutes to several hours.

We first test the speedup of the extraction time for the visible human male MRI dataset from 1 processor to 96 processors. In this test we adopt the random block distribution method and implement it with a random hash function. The ability to run efficiently on a single processor demonstrates the out-of-core feature of our method. The near ideal speedup to 32 processors and very good speedup to even 96 processors show the scalability of the algorithm with the number of processors. Fig. 14 shows the speedup of isosurface extraction time for two typical isovalues 800 and 1200, corresponding to the skin and bone respectively, for the MRI dataset. The slopes of the two speedup curves are very similar to each other, and close to the ideal case, which demonstrates the parallel and out-of-core isocontour extraction algorithm scales equally well to different isovalues.

Fig. 15 shows the actual extraction and rendering time of each processor for an isosurface (isovalue = 800) from 1 to 32 processors. The surface extracted by one node is rendered by the same machine and the images are then composited by the Metabuffer with a small and constant delay, which is not shown in the figure. Every rendering server in this configuration has the viewport of the whole display space. In other words, each rendering server renders at the same resolution. The curves are fairly flat for different number of processors and once again demonstrate good load balance among the processors.

Next we compare the performance of the deterministic greedy and randomized data distribution methods. Histograms among 32 processors for the visible human male MRI data are shown in Fig. 16. The first row of figures show the number ofblocks loaded, number of triangles, and isosurface extraction time for the deterministic data distribution respectively; the second row shows the same curves for the randomized distribution. Each curve in the figures shows the result for an individual processor for the whole range of isovalues. Those figures demonstrate that both distributions have very good load balance for the whole range of isovalues from 0 to 1900. Furthermore, the curves of triangle count closely resemble the curves of actual extraction

and rendering time. It justifies our using the number of triangle to represent workload in data partitioning.

The deterministic method loads more balanced numbers of blocks, as expected from its implementation. The randomized method actually achieves better balance for real extraction time, especially for large isosurfaces. This verifies the Theorem 2 in Section 4.1. Similar results for the gas hydrodynamic simulation dataset are shown in Fig. 17.

We further test our implementation on the much larger visible human male cryosection and visible human female cryosection datasets. Fig. 18(a) shows the isosurface extraction time of the two datasets for isovalue 13,000. Fig. 18(b) shows the histogram of isosurface extraction time for the visible male cryosection dataset with 96 processors, where curves for the 96 processors are too close to be clearly distinguished. Similar curves exist for the visible female cryosection dataset. As the size of datasets gets larger, we achieve even better load balance among the processors as specified in Theorem 2. The time curves in Fig. 18(a) demonstrate very good speedup when the number of processors and disks increases. The sharp drop of computational time from 8 processors to 16 processors for the Male cryosection data and from 16 processors to 32 processors for the Female cryosection data is due to the better operating system disk performance for smaller partitions.

A very large isosurface (487,635,342 triangles) extracted from the Female cryosection data is shown in Fig. 19. While the surface is noisy because of the dataset, it demonstrates the scalability of our system to very large data and very large surfaces.

6. Conclusion

In this paper we propose a scalable time-critical rendering framework of massive datastreams, based around the Metabuffer image composition hardware. The time-critical rendering process can be thought of as a pipeline from parallel and progressive mesh generation, parallel rendering to parallel image composition. We describe in detail a scalable isocontouring algorithm by taking advantage of parallel processors and parallel disks. It statically partitions the volume data to balance the workload spectrum, and builds an I/O-optimal external interval tree to minimize the number of I/O operations of loading large data from disk. We present a deterministic and a randomized data partition algorithm that is proved to provide a load-balanced data partition for any isovalue with high probability. Experimental results show that both achieve very good load balance, while the randomized method outperforms the greedy algorithm in many cases. We also present an efficient isosurface compression algorithm that takes advantage of the embeding 3D mesh, compresses isosurface progressively, and can achieve better compression ratio than a general purpose compression algorithm.

While the algorithms were tested on a dated platform of PC clusters by today's standard, we were able to achieve a frame-rate of more than 3 frames/s for end-to-end extracting and visualizing the reasonably large “Male MRI” data on a cluster of 32 nodes. We also demonstrated the scalability of the algorithms up to the limit of our test platform, 96 nodes. More computational power is necessary for real interactivity for tera-byte datasets. There are also the remaining problems of introducing dynamic load balancing, by replicating some data on different disks, and accessing data from remote disks at runtime. It is an interesting problem to see how much improvement we can achieve by choosing larger replication factors and data distribution.

Acknowledgments

This research is supported in part by grants ACI-9982297, DMS-9873326 from the National Science Foundation, a DOE-ASCI grant BD4485-MOID from Sandia National Laboratory, Lawrence Livermore National Laboratory, from

grant UCSD 1018140 as part of the National Partnership for Advanced Computational Infrastructure, a grant BD 781 from the Texas Board of Higher Education, and a gift and cluster support from Compaq Computer Corporation.

Biographies

Xiaoyu Zhang is an associate professor of Computer Sciences at the California State University San Marcos. He received his Bachelor's and Master's degree in Physics from the University of Science and Technology of China (USTC) in 1993 and 1996 respectively. and his Ph.D. in Computer Sciences from the University of Texas at Austin in 2001. Dr. Zhang's research areas of interest include computer graphics and visualization, bio-informatics, parallel and distributed computing. He has served as reviewer for several conferences, and journals.

Chandrajit Bajaj is the Computational Applied Mathematics Chair in Visualization professor of computer sciences at the University of Texas at Austin as well as the director of the Center for Computational Visualization, in the Institute for Computational and Engineering Sciences (ICES). He graduated from the Indian Institute of Technology, Delhi with a Bachelor's Degree in Electrical Engineering, in 1980 and received his M.S. and Ph.D. degrees in Computer Sciences from Cornell University, in 1983, and 1984 respectively. Prior to the University of Texas, Bajaj was a professor of computer sciences at Purdue University and director of the Purdue center for image analysis and visualization. Dr. Bajaj's research areas of interest include Image Processing, Geometric Modeling, Computer Graphics, Visualization, and Computational Mathematics. Current research topics include de-noising, reconstruction and compression algorithms for volumetric and time-dependent imaging, as well as data structures that support multi-resolution finite element approximations of large geometries and multiple function fields.

References

[1]. Arge, L.; Vitter, JS. Optimal interval management in external memory. Proc. IEEE Foundations of Computer Science; 1996.

[2]. Bajaj C, Bernardini F, Xu G. Automatic reconstruction of surfaces and scalar fields from 3D scans, Proc. of the ACM Siggraph Computer Graphics. SIGGRAPH 95 1995;29(3):109–118.

[3]. Bajaj, C.; Pascucci, V.; Schikore, D. Proceedings: ACM Siggraph/IEEE Symposium on Volume Visualization. ACM Press; 1996. Fast isocontouring for improved interactivity.

[4]. Bajaj, C.; Pascucci, V.; Schikore, D. The contour spectrum. Proceedings of the 1997 IEEE Visualization Conference; 1997.

[5]. Bajaj C, Pascucci V, Thompson D, Zhang X. Parallel acclerated isocontouring for out-of-core visualization. Proceedings of IEEE Parallel Visualization and Graphics Symposium. 1999

[6]. Bajaj, C.; Pascucci, V.; Zhuang, G. Progressive compression and transmission of arbitrary triangular meshes. Proceedings of the 10th IEEE Visualization 1999 Conference; San Francisco, CA. 1999.

[7]. Bajaj, C.; Pascucci, V.; Zhuang, G. Single resolution compression of arbitrary triangular meshes with properties. IEEE Data Compression Conference; 1999.

[8]. Blanke, WJ.; Fussell, D.; Bajaj, C.; Zhang, X. Computer Science. University of Texas; Austin: Feb. 2000 The metabuffer: A scalable multiresolution multidisplay 3-d graphics system using commodity rendering engines, Tr2000-16.

[9]. Bottou, L.; Howard, P.; Bengio, Y. The z-coder adaptive binary coder. Proceedings of IEEE Data Compression Conference DCC'98; 1998.

[10]. Chen S, Reif J. Fast and compact volume rendering in the compressed transform domain. Data Compression. 1997

[11]. Chiang, Y.; Silva, CT. I/O optimal isosurface extraction. In: Yagel, R.; Hagen, H., editors. IEEE Visualization'97. IEEE; 1997.

[12]. Chiang, Y-J.; Silva, CT.; Schroeder, WJ. Interactive out-of-core isosurface extraction. Proceedings of the 9th Annual IEEE Conference on Visualization, VIS-98; ACM Press; 1998.

[13]. Chiang Y-J, Silva CT. External memory techniques for isosurface extraction in scientific visualization. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. 1999

[14]. Chow MM. Optimized geometry compression for real-time rendering. Proceedings of IEEE Visualization'97, Phoenix. 1997

[15]. Coffman, EG.; Garey, MR.; Johnson, DS. Approximation algorithms for bin packing: A survey. In: Hochbaum, D., editor. Approximation Algorithms. 1996.

[16]. Cohen-Or D, Levin D, Remez O. Progressive compression of arbitrary triangular meshes. Visualization'99. 1999

[17]. Crockett TW. Parallel rendering. Tech. Rep., ICASE. 1995

[18]. Deering M. Geometry compression. Computer Graphics (SIGGRAPH 95 Proceedings) 1995:13–20.

[19]. Eldridge M, Igehy H, Hanrahan P. Pomegranate: A fully scalable graphics architecture. Computer Graphics (SIGGRAPH 2000 Proceedings) 2000:443–454.

[20]. Ellsiepen P. Parallel isosurfacing in large unstructed datasets. Visualization in Scientific Computing, Springer-Verlag. 1995

[21]. Eyles J, Molnar S, Poulton J, Greer T, Lastra A, England N, Westover L. Pixelflow: The realization. Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware. 1997

[22]. Fussell DS, Rathi BD. A vlsi-oriented architecture for real-time raster display of shaded polygons. Graphics Interface'82. May;1982

[23]. Gibbons P, Matias Y, Ramachandran V. Can a shared memory model serve as a bridging model for parallel computation. Theory of Computing Systems 1999;32(3):327–359.(Special Issue on Papers from SPAA'97)

[24]. Gueziec, A.; Taubin, G.; Lazarus, F.; Horn, W. Tech. Rep. RC-20935. IBM T.J. Watson Research Center; 1997. Cutting and stitching: Efficient conversion of a non-manifold polygonal surface to a manifold.

[25]. Hansen C, Hinker P. Massively parallel isosurface extraction. Visualization '92. 1992

[26]. Heirich, A.; Moll, L. Scalable distributed visualization using off-the-shelf components. In: Ahrens, J.; Chalmers, A.; Shen, H-W., editors. Parallel Visualization and Graphics Symposium - 1999. San Francisco, California: 1999.

[27]. Humphreys G, Buck I, Eldridge M, Hanrahan P. Distributed rendering for scalable displays. Proceedings of Supercomputing 2000. 2000

[28]. Humphreys, G.; Hanrahan, P. A distributed graphics system for large tiled displays. Proceedings of IEEE Visualization Conference; 1999.

[29]. Igehy H, Stoll G, Hanrahan P. The design of a parallel graphics interface. Proc. SIGGRAPH 98. 1998

[30]. Ihm I, Park S. Wavelet-based 3d compression scheme for interactive visualization of very large volume data. Proceedings of Graphics Interface '98. 1998

[31]. Kanellakis P, Ramaswamy S, Vengroff D, Vitter JS. Indexing for data models with constraints and classes. Proc. ACM Symp. on Principles of Database Sys. 1993

[32]. Lombeyda S, Shand M, Moll L, Breen D, Heirich A. Scalable interactive volume rendering using off-the-shelf components. Proceedings of IEEE Parallel Visualization and Graphics Symposium. 2001

[33]. Lorensen W, Cline H. Marching cubes: A high resolution 3d surface construction algorithm. Computer Graphics 1987;21(4):163–169.

[34]. Ma K-L, Painter JS, Hansen CD, Krogh MF. Parallel volume rendering using binary-swap image composition. IEEE Computer Graphics and Applications 14(4)

[35]. Miguet S, Nicod J-M. A load-balanced parallel implementation of the marching-cubes algorithm, Tech. Rep. 95-24. Ecole Normale Supérieure de Lyon. 1995

[36]. Molnar, SE. Technical Report tr91-046. University of North Carolina; 1991. Image composition architectures for real-time image generation, Ph.d. dissertation.

[37]. Molnar S, Cox M, Ellsworth D, Fuchs H. A sorting classification of parallel rendering. IEEE Computer Graphics and Applications 14(4)

[38]. Montani C, Scateni R, Scopigno R. A modified look-up table for implicit disambiguation of marching cubes. The Visual Computer 1994;10(6):353–355.

[39]. Montrym J, Baum D, Dignam D, Migdal C. Infinitereality: A real-time graphics system. Proceedings of SIGGRAPH 97. 1997

[40]. Moreland K, Wylie B, Pavlakos C. Sort-last parallel rendering for viewing extremely large data sets on tile displays. Proceedings of IEEE Parallel Visualization and Graphics Symposium. 2001

[41]. Mueller C. The sort-first rendering architecture for high-performance graphics. Symposium on Interactive 3D Graphics. 1995

[42]. Neumann U. Communication costs for parallel volume-rendering applications. IEEE Computer Graphics & Applications 1994:49–58.

[43]. Paper HW. HP Scalable Visualization Array (HP SVA) Parallel Compositing Library. 2007

[44]. Parker S, Shirley P, Livnat Y, Hansen C, Sloan P. Interactive ray tracing for isosurface rendering. Visualization'98. 1998

[45]. Raghavan P. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. Journal of Computer and System Science 37 1988:130–143.

[46]. Ramachandran V. A general purpose shared-memory model for parallel computation. IMA Volumes in Mathematics and its Applications, Algorithms for Parallel Processing 1999;105:1–17.

[47]. Samanta R, Funkhouser T, Li K. Parallel rendering with *k*-way replication. Proceedings of IEEE Parallel Visualization and Graphics Symposium. 2001

[48]. Samanta R, Funkhouser T, Li K, Singh JP. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. Eurographics/Siggraph workshop on Graphics Hardware. 2000

[49]. Samanta R, Zheng J, Funkhouser T, Li K, Singh JP. Load balancing for multi-projector rendering systems. SIGGRAPH/Eurographics Workshop on Graphics Hardware. 1999

[50]. Schneider B-O. Parallel rendering on pc workstations. Parallel and Distributed Processing Techniques and Applications. 1998

[51]. Shen H, Hansen C, Livnat Y, Johnson C. Isosurfacing in span space with utmost efficiency (issue). Visualization'96. 1996

[52]. Stoll G, Eldridge M, Patterson D, Webb A, Berman S, Levoy R, Caywood C, Taveira M, Hunt S, Hanrahan P. Lightning-2: A high-performance display subsystem for pc clusters. Proceedings of SIGGRAPH 2001. 2001

[53]. Taubin G, Rossignac J. Geometric compression through topological surgery. ACM Transactions on Graphics 1996;17(2):84–115.

[54]. Touma, C.; Gotsman, C. Triangle mesh compression. In: Davis, W.; Booth, K.; Fourier, A., editors. Proceedings of the 24th Conference on Graphics Interface, GI-98; Morgan Kaufmann Publishers; 1998.

[55]. Udeshi T, Hansen C. Parallel multipipe rendering for very large isosurface visualization. Joint EUROGRAPHICS - IEEE TCCG Symposium on Visualization. 1999

[56]. Valiant LG. A bridging model for parallel computation. Communications of the ACM 1990;33:103–111.

[57]. Vitter JS. External memory algorithms and data structure. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. 1999

[58]. Wald I, Friedrich H, Marmitt G, Slusallek P, Seidel H-P. Faster isosurface ray tracing using implicit kd-trees. IEEE Transactions on Visualization and Computer Graphics 2005;11(5):562–572. [PubMed: 16144253]

[59]. Waters KW, Co CS, Joy M-KI. Using difference intervals for time-varying isosurface visualization. IEEE Transactions on Visualization and Computer Graphics 2006;12(5):1275–1282. [PubMed: 17080862]

[60]. Witten I, Neal R, Cleary J. Arithmetic coding for data compression. Communications of ACM 1987;30:520–540.

[61]. Zhang X, Bajaj C, Blanke W. Scalable isosurface visualization of massive datasets on cots clusters. Proceedings of IEEE Parallel Visualization and Graphics Symposium. 2001

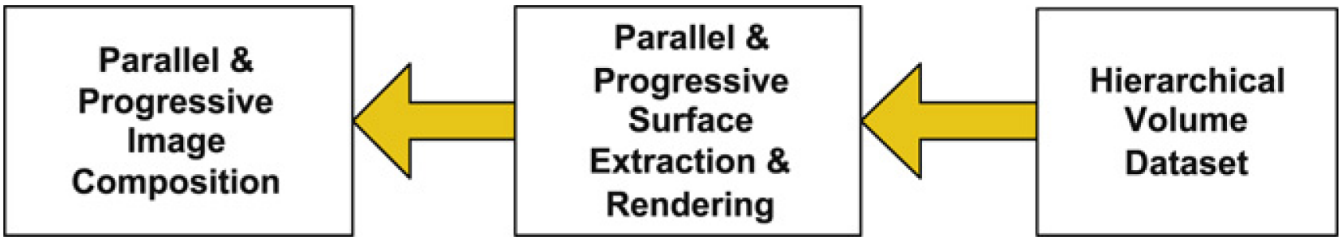

Fig. 1.
A schematic drawing showing the three stages of scalable isosurface visualization pipeline.

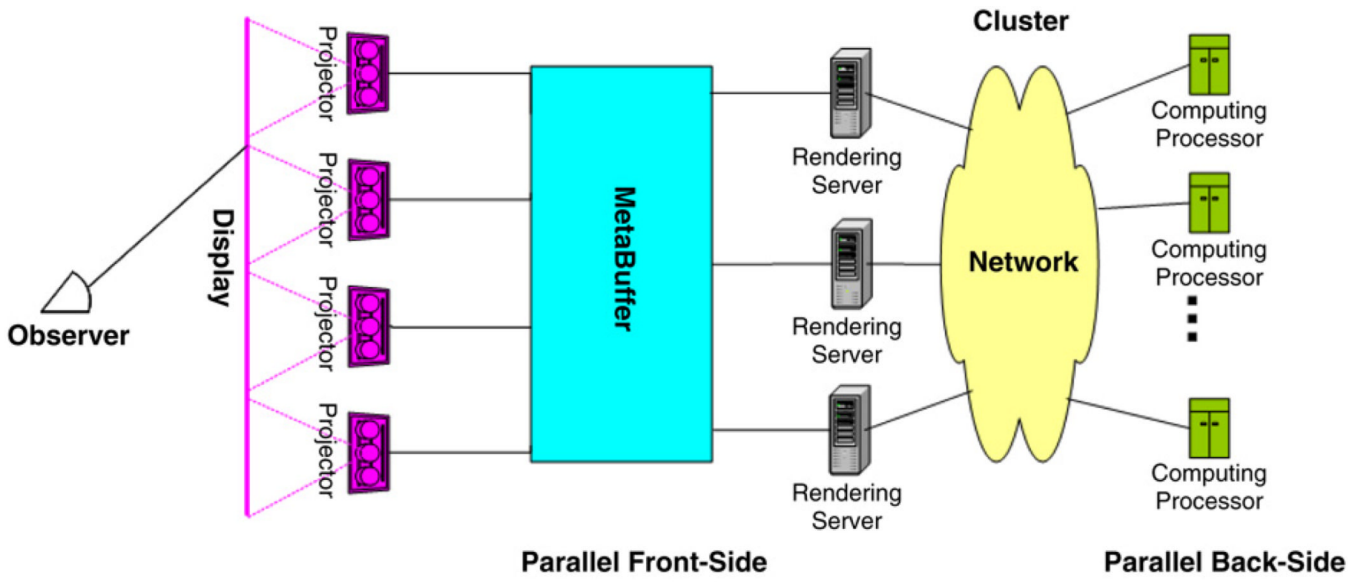

Fig. 2.

Our system architecture for scalable isosurface visualization. The parallel back-side accomplishes progressive surface extraction and rendering while the parallel front-side composites and displays progressively.

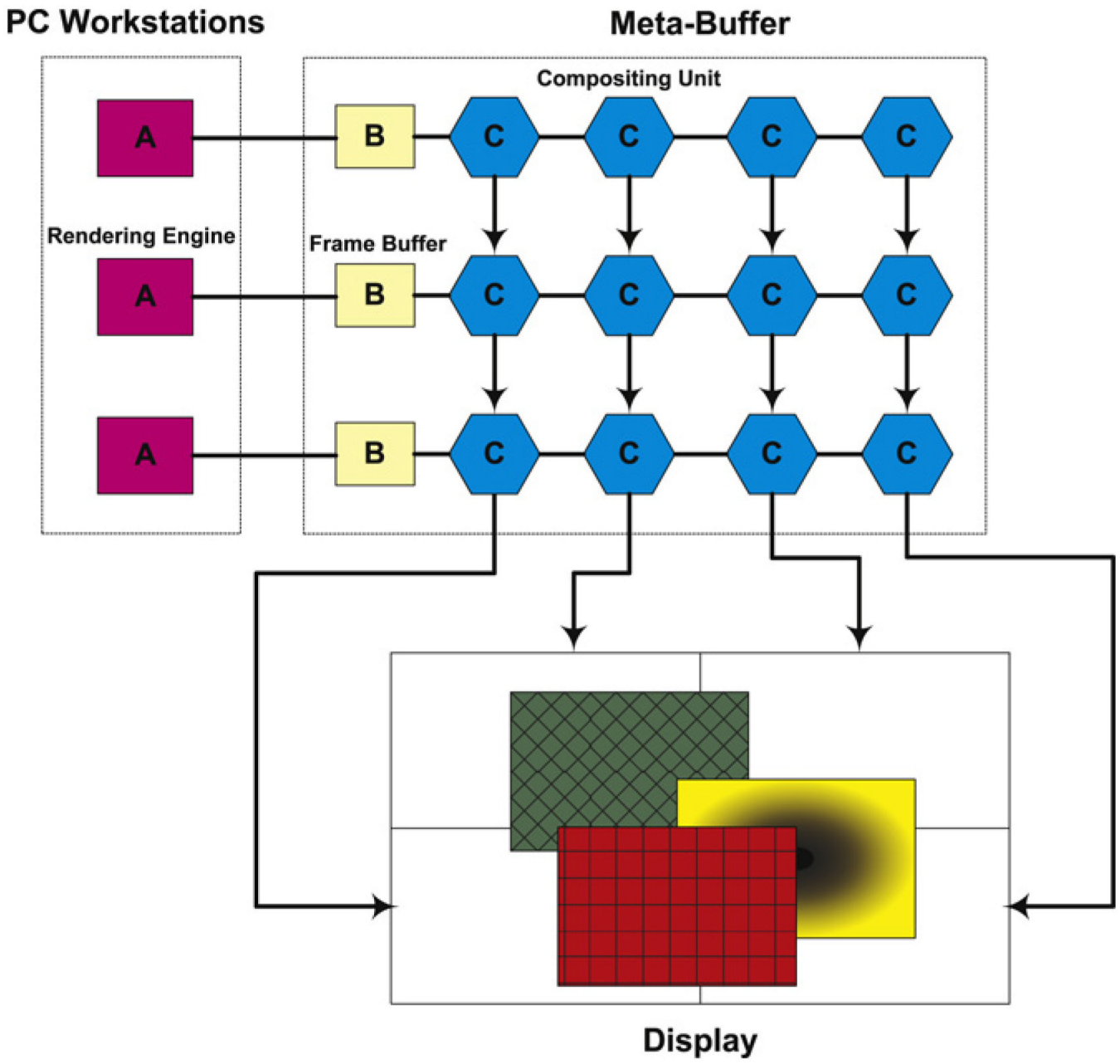

Fig. 3.
The Metabuffer architecture, where *A* represents a rendering engine, *B* is an on-board frame buffer, and *C* represents a composing unit.

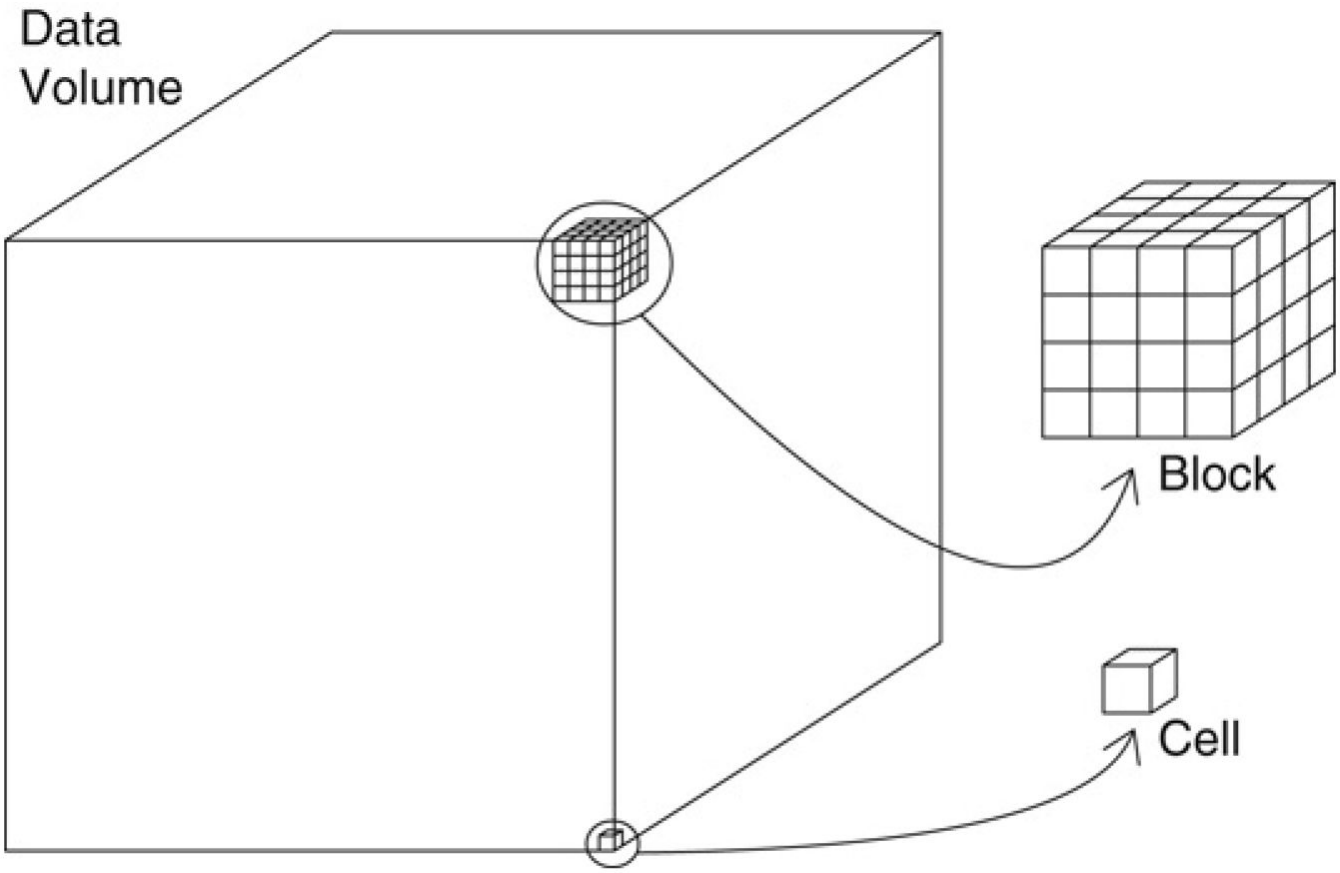

Fig. 4.
A 3D regular volumetric dataset, its cell, and a processing element of 4×4×4 block.

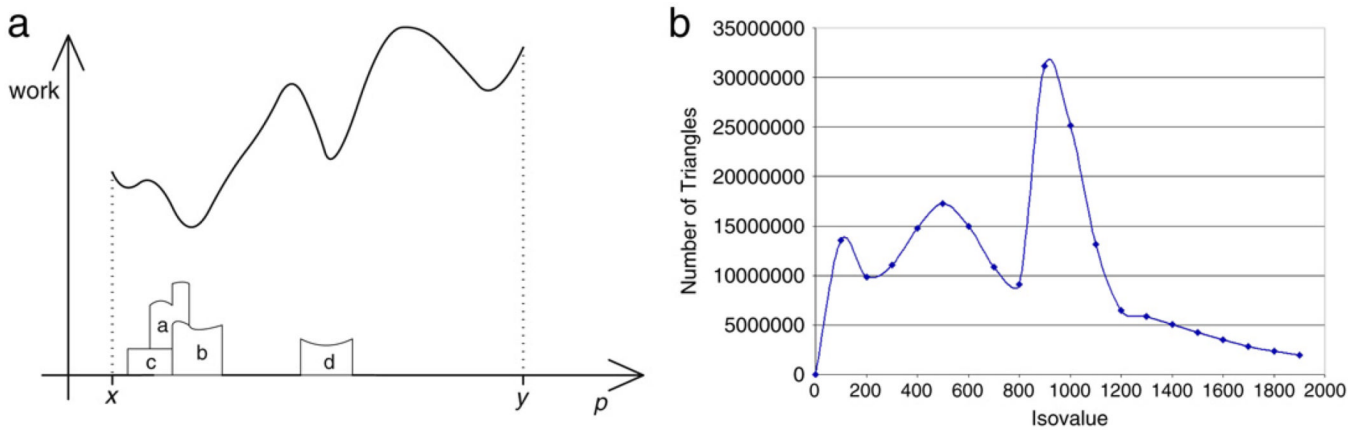

Fig. 5.

(a) Workload diagram of isocontouring on a fixed dataset *D* with respect to the isovalue *p* (horizontal axis). The overall diagram is the sum of the diagrams computed for each unit (*a*, *b*, *c*, *d*, …). (b) Histogram for a real dataset (CT scan of the visible human male).

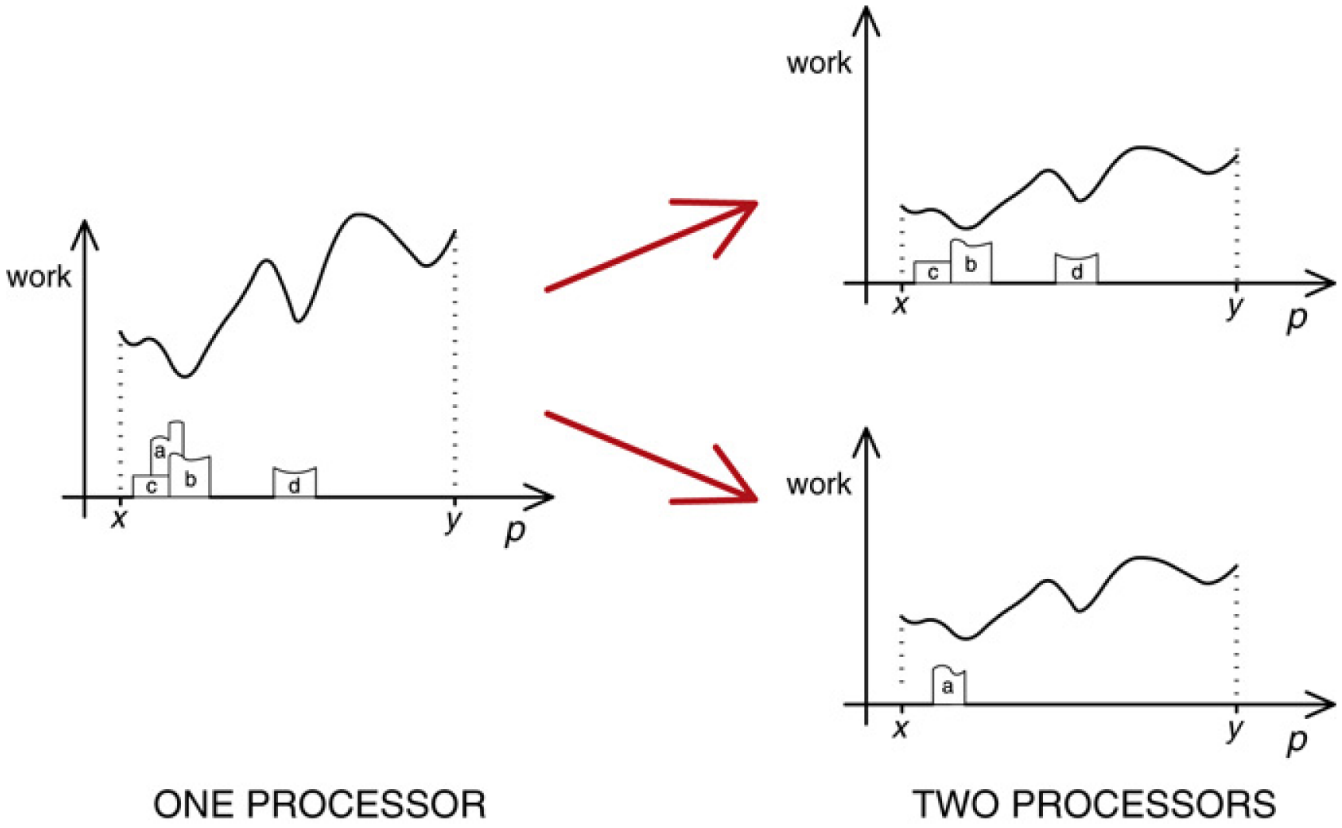

Fig. 6.
Optimal data partition for load-balanced parallel computations (two processors case).

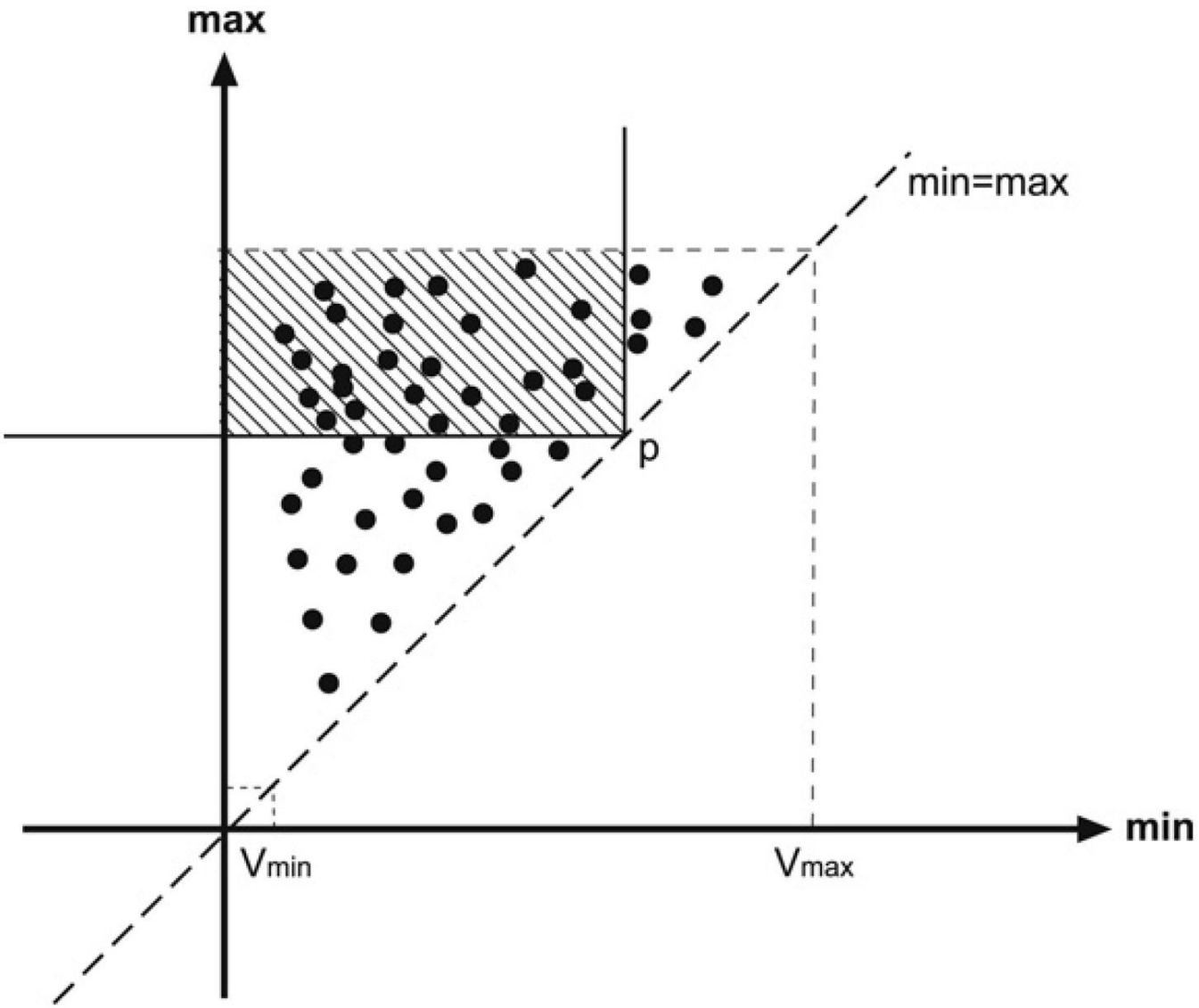

Fig. 7.
Blocks intersecting the isosurface of the isovalue *p* are shown in the range space (shaded area). V_{min} and V_{max} are the minimum and maximum values of the scalar field respectively.

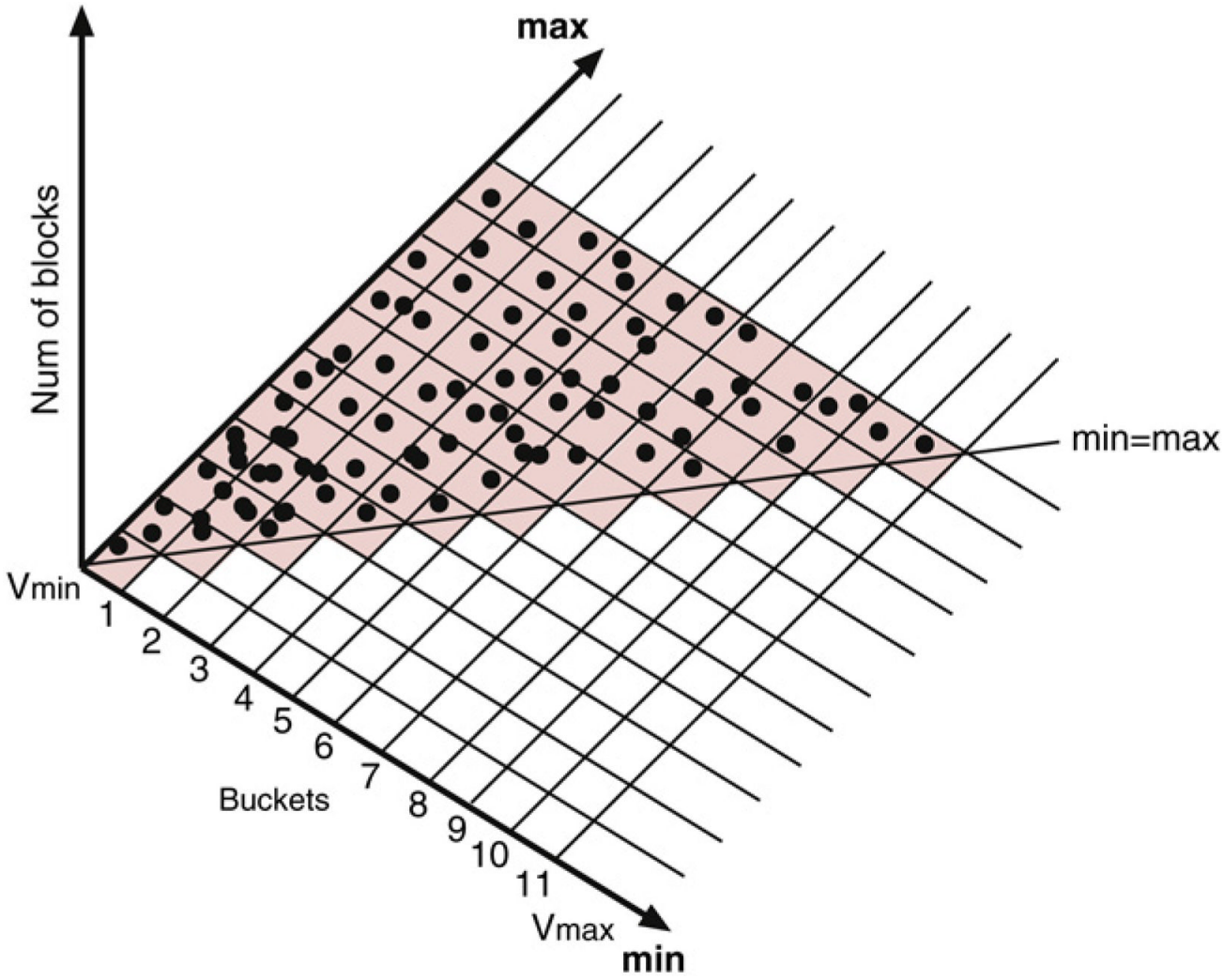

Fig. 8.
The triangular matrix for data decomposition. The range space of a dataset is divided into a triangular matrix (shaded area), where each matrix element spans a single bucket.

```
read_and_extract(Block b) {
    read the block b from the local disk;
    extract/render the polygons in b;
}
run in parallel {
    v = the query isovalue;
    n = root of the external interval tree;
    while (n is not a leaf) {
        s = the range segment of n that contains v;
        for (blocks b in s's left list with min ≤ v
                | in s's right list with max ≥ v
                | in the multi-lists that contain s)
            read_and_extract(b);
    n = n's child node corresponding to s;
    }
    for(blocks b in n with min ≤ v ≤ max)
        read_and_extract(b);
```

Fig. 9.
Pseudo-code for parallel and out-of-core isosurface extraction.

Fig. 10.
An isosurface for the Male MRI data (isovalue = 1200) is progressively extracted and rendered at different resolutions and from various viewpoints. The leftmost isosurface has 24,084 triangles; the center isosurface has 651,234 triangles, and the rightmost isosurface has 6442,810 triangles.

Fig. 11.
Individual portions of an isosurface (isovalue = 800) extracted from the visible male MRI data with eight computing processors and rendered by the same set of machines. The full resolution isosurface has 9,128,798 triangles.

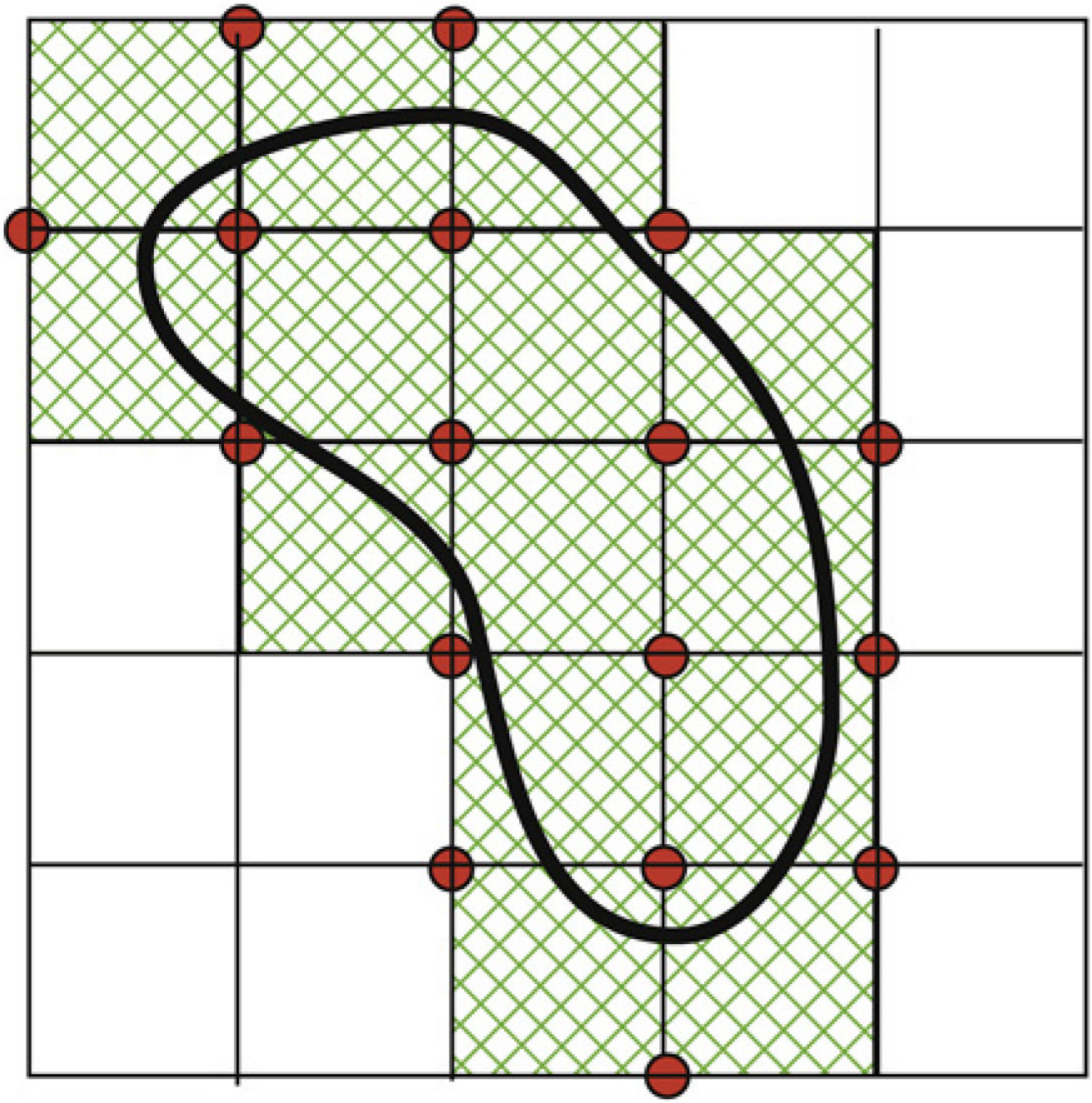

Fig. 12.
A 2D example of relevant vertices (big red points) and valid cells (shaded in green).

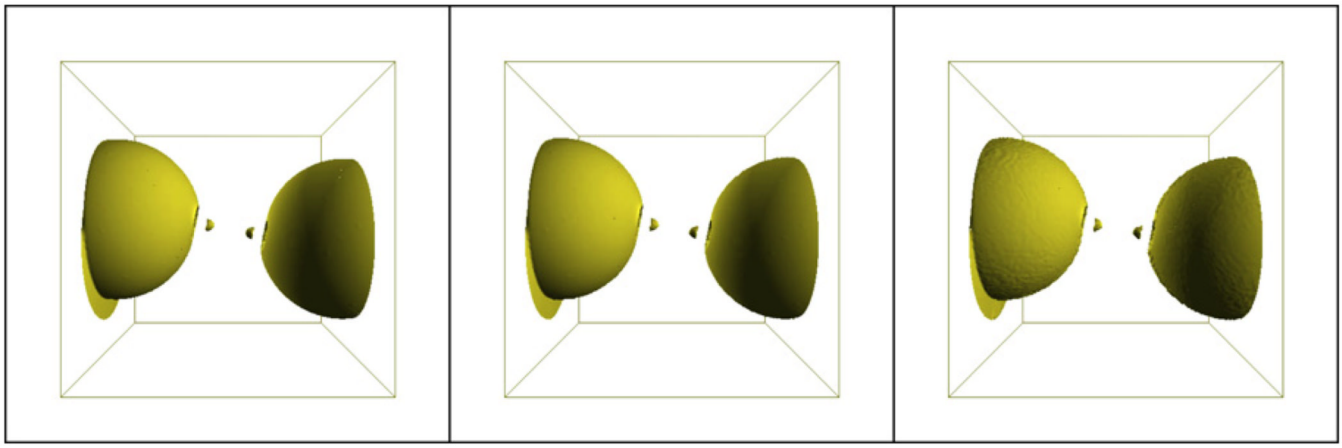

Fig. 13.
The edge index encoding of an isosurface keeps the right topology of the surface, even if few quantization bits are used. This figure shows an isosurface of a black hole simulation encoded respectively with (a) 14 bits, (b) 10 bits, and (c) 6 bits for the function value quantization.

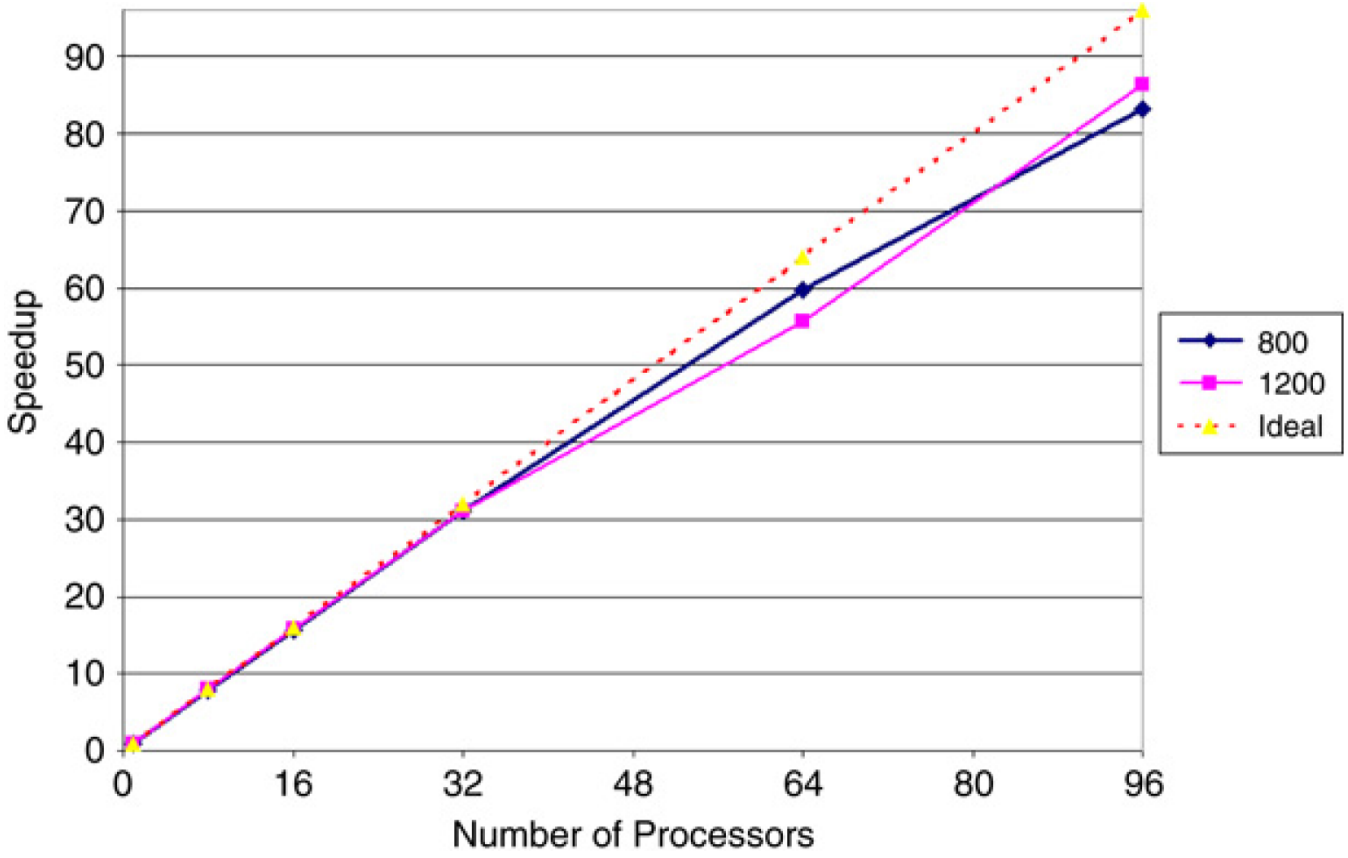

Fig. 14.
Speedup of isosurface extraction for two isovalues (800 and 1200) compared to the ideal case. The dataset used is the visible human male MRI data.

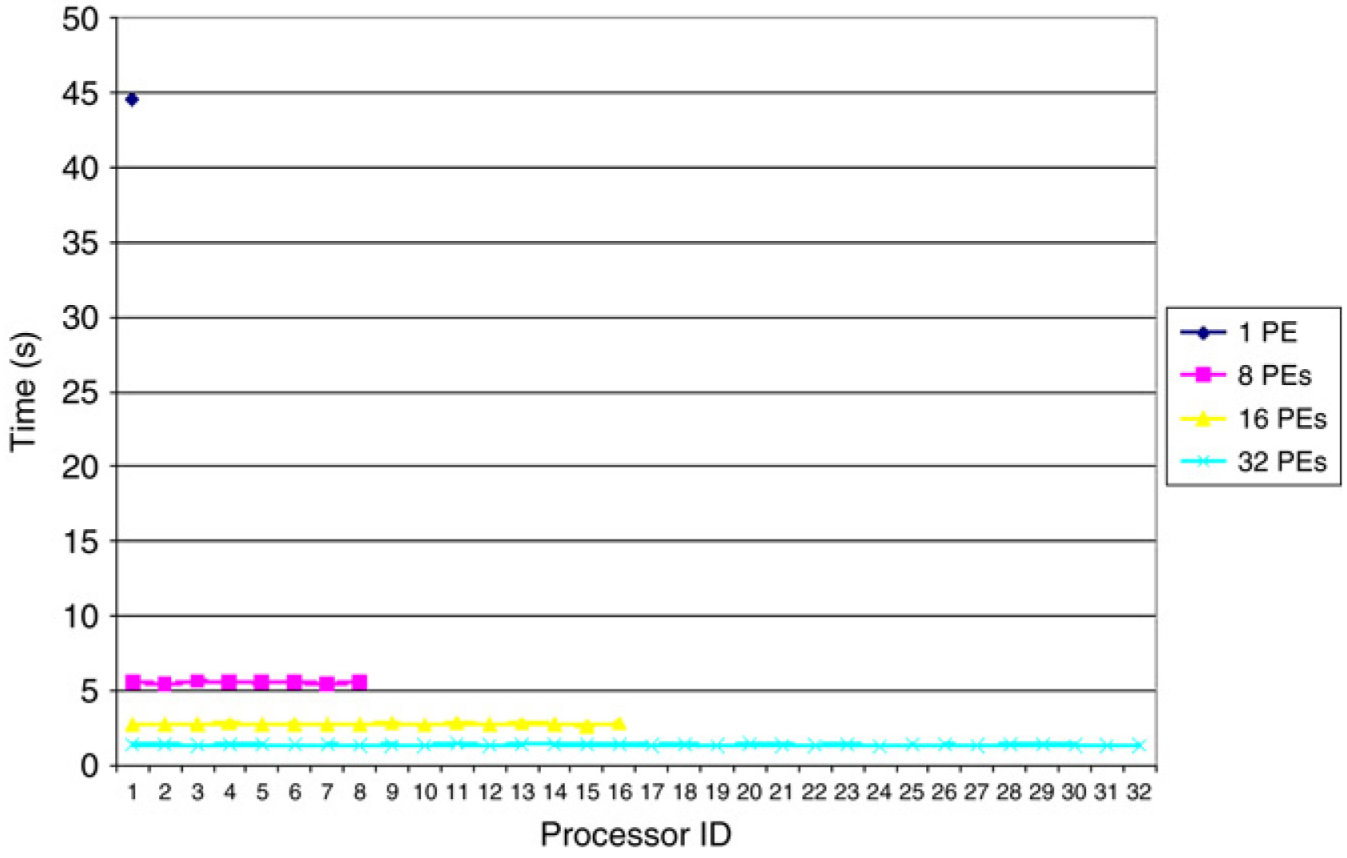

Fig. 15.

Time of individual processors to extract and render an isosurface (isovalue = 800) from the visible human male MRI dataset with 1, 8, 16, and 32 processors.

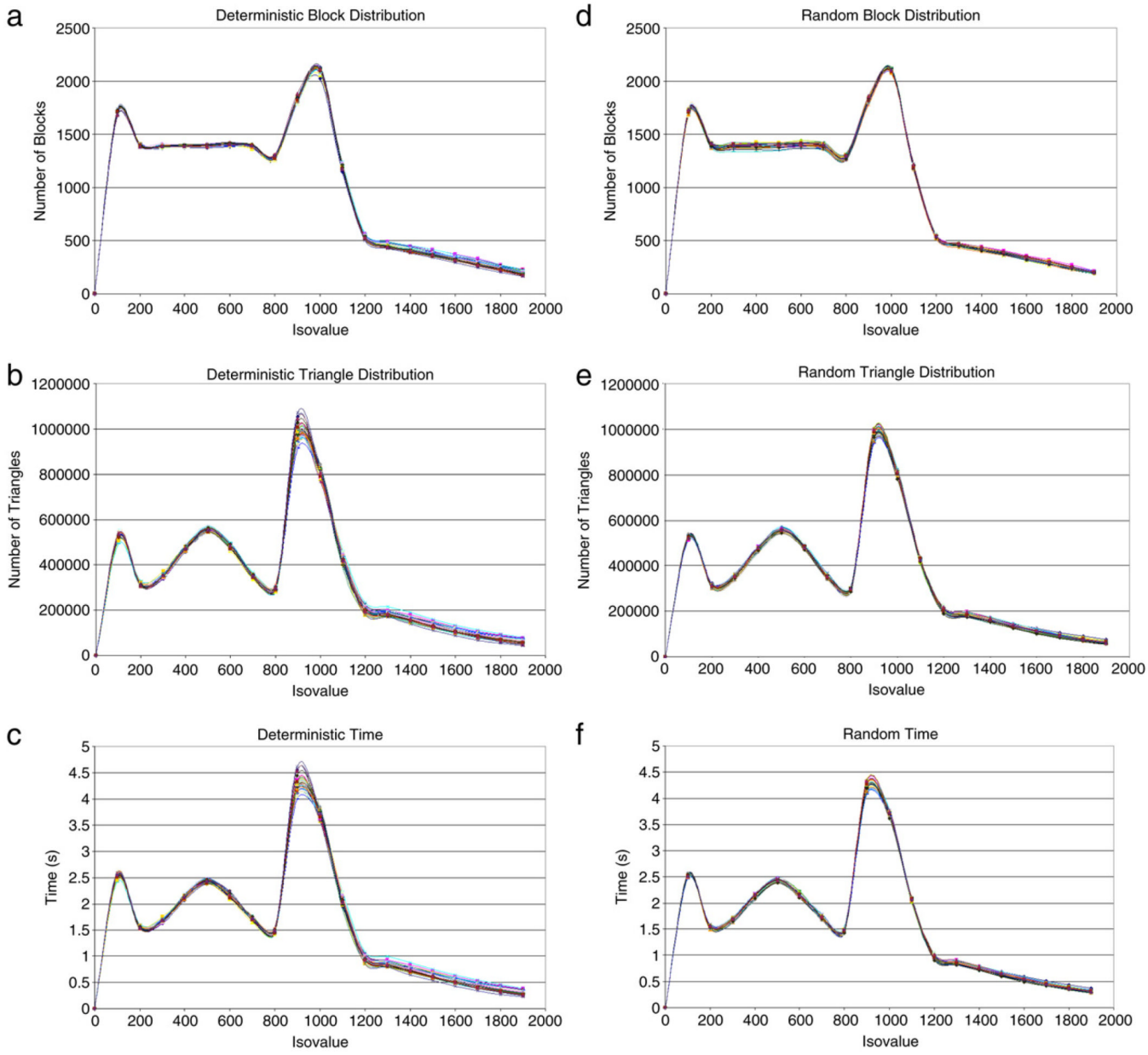

Fig. 16.

Histograms of block distribution, triangle distribution, and isosurface extraction and rendering time for the deterministic ((a), (b) and (c)) and random data partitioning method ((d), (e) and (f)) for the visible human male MRI data among 32 processors.

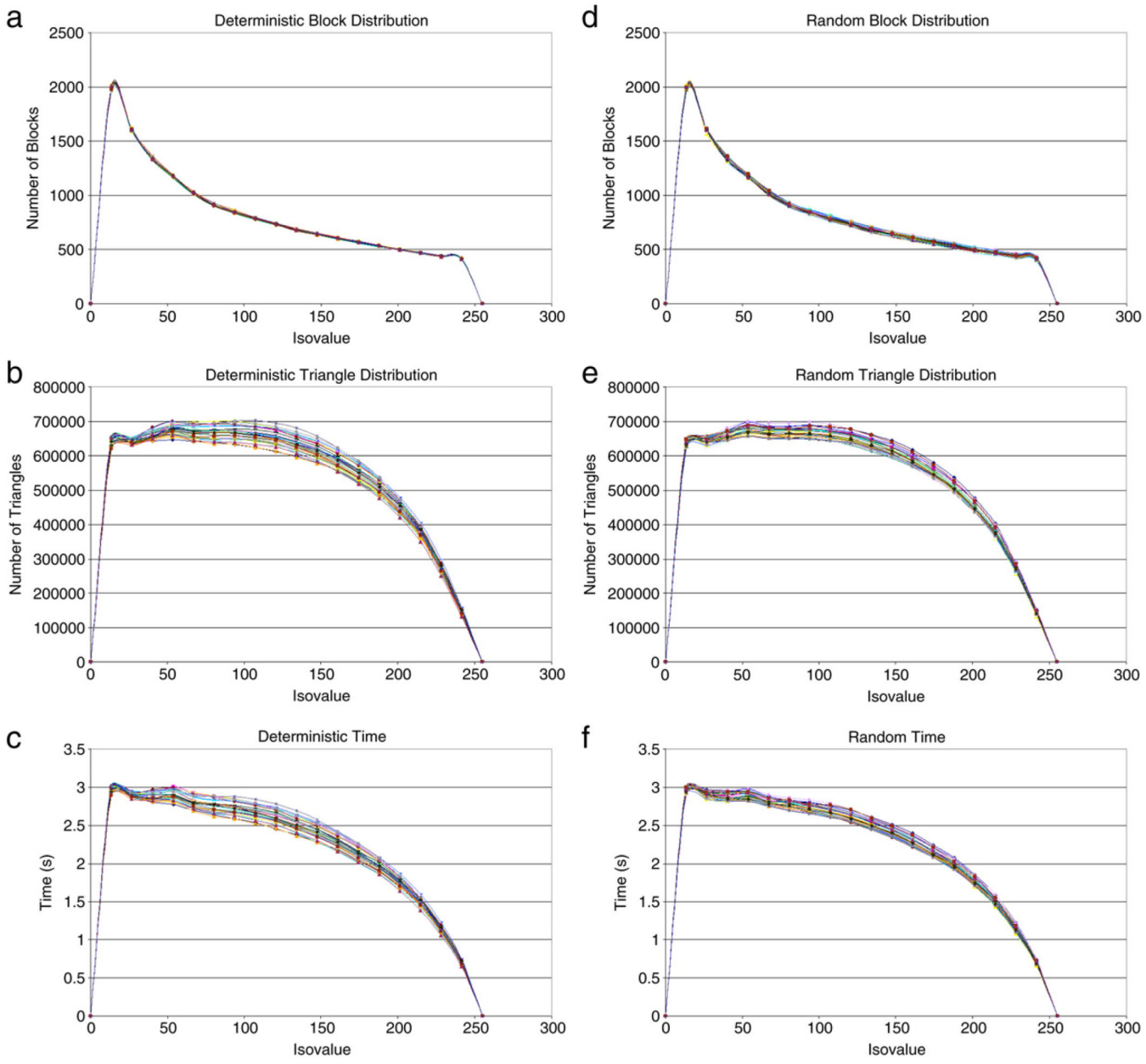

Fig. 17.

Histograms of block distribution, triangle distribution, and isosurface extraction and rendering time for the deterministic ((a), (b) and (c)) and random data partitioning ((d), (e) and (f)) for the gas hydrodynamic simulation dataset among 32 processors.

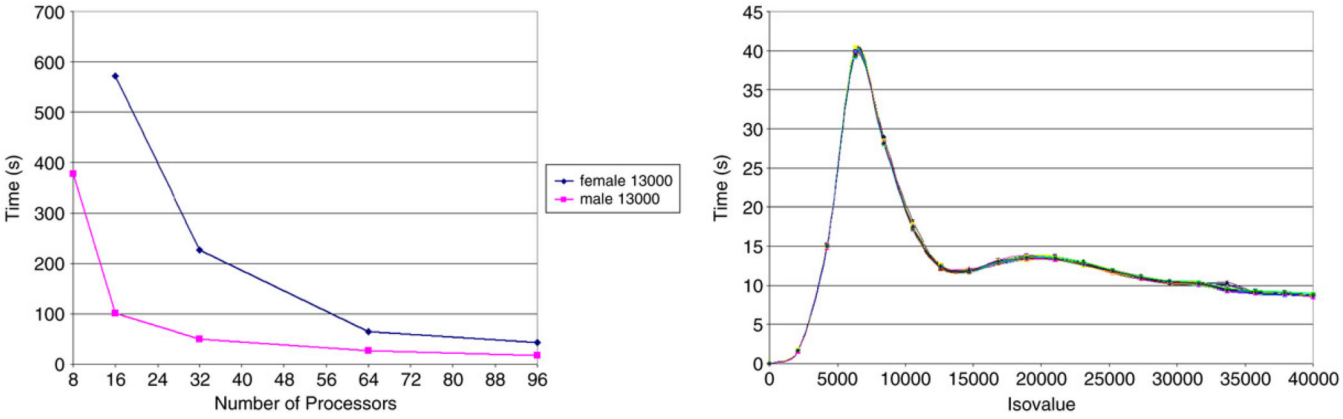

Fig. 18.

(a) Isosurface extraction time for the visible male cryosection and female cryosection data at isovalues 13,000. (b) Histogram of isosurface extraction time for the visible male cryosection data with 96 processors.

Fig. 19.
Two views of an isosurface (isovalue = 29,000) extracted and rendered from the full resolution Visible Female cryosection data. The isosurface contains 487,635,342 triangles.

Table 1

Comparison of compressed isosurface sizes in bytes using edge index coding with a general surface compression algorithm

Model	Value	Data type	Orig. size	Gen. alg.	Edge index
Hipip	0.0	float	1,465,862	129,068	88,804
Foot	600.	u_short	7,536,227	587,344	407,658
Foot	1500.	u_short	8,645,243	761,413	386,506
Engine	180.	u_char	3,601,040	224,893	121,504
Engine	89.	u_char	15,888,473	1,012,491	618,492

The general compression algorithm uses 8 bits for each coordinate per vertex. For unsigned short and char data type, we encode them directly using the predictive coder. Float values are normalized and quantized using 32 bits for the first point and 14 bits for difference.

Table 2

The sizes of our test datasets

Name	Dimension	Size
Gas density	512 × 512 × 512	134 MB
Male MRI	512 × 512 × 1252	656 MB
Male cryosection	1800 × 1000 × 1878	6.6 GB
Female cryosection	1600 × 1000 × 5186	16.5 GB