Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2002)
D. Ebert, P. Brunet, I. Navazo (Editors)

Parallel and Out-of-core View-dependent Isocontour Visualization Using Random Data Distribution

Xiaoyu Zhang, Chandrajit Bajaj, Vijaya Ramachandran

Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712

Abstract

In this paper we describe a parallel and out-of-core view-dependent isocontour visualization algorithm that efficiently extracts and renders the visible portions of an isosurface from large datasets. The algorithm first creates an occlusion map using ray-casting and nearest neighbors. With the occlusion map constructed, the visible portion of the isosurface is extracted and rendered. All steps are in a single pass with minimal communication overhead. The overall workload is well balanced among parallel processors using random data distribution. Volumetric datasets are statically partitioned onto the local disks of each processor and loaded only when necessary. This out-of-core feature allows it to handle scalably large datasets. We additionally demonstrate significant speedup of the view-dependent isocontour visualization on a commodity off-the-shelf PC cluster.

1. Introduction

Today tomographic imaging and computer simulations are increasingly generating large datasets that are to be effectively visualized for better understanding of the underlying scientific information. Isocontour visualization is a popular interactive and exploratory visualization technique used to determine and browse regions of interest within volumetric imaging data, and validate the results of computer simulations. Interactive isocontour visualization extracts multiple 2-dimensional surfaces satisfying $F(\mathbf{x}) = const$ from a given scalar field $F(\mathbf{x})$, $\mathbf{x} \in \mathbf{R}^3$, and renders them at interactive frame rates ($30HZ$).

Related Work: 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. Hansen and Hinker[16] describe parallel methods for isosurface extraction on SIMD machines. Ellsiepen [11] describes a parallel isosurfacing method for FEM data by dynamically distributing working blocks to a number of connected workstations. Shen et al. [31] implement a parallel algorithm by partitioning load in the span space. Parker et al. [27] present a parallel isosurface rendering algorithm using ray-tracing. Chiang and Silva [6, 8] give an implementation of out-of-core isocontouring using the I/O optimal external interval tree on a single processor. Bajaj et al. [2] use range partition to reduce the size of data that are loaded for given isocontour queries and balance the load within a range partition. More recently Zhang et al. [35] propose a scalable isosurface visualization framework for massive datasets on commodity off-the-shelf clusters by combining the parallel and out-of-core isocontouring techniques. Chiang et al. [7] also try to combine parallel and out-of-core techniques for isosurface and volume rendering for unstructured grids.

An isocontour visualization is not complete without isosurfaces being rendered and displayed to the user. Rendering is a critical and indispensable part of visualization. In the case of isosurface rendering, the triangles of an isosurfaces are projected from the $3D$ object space to the $2D$ image space by graphics hardware. Efficient rendering is of special importance when the user wants to observe an isosurface from different viewpoints after the surface is extracted. Large isosurfaces extracted from large datasets often need to be rendered with parallel graphics pipes for interactivity.

Udeshi and Hansen [33] employ the multi-pipes of a SGI Onyx2 reality monster to render large isosurfaces in a sort-last fashion [25]. They use a binary-swap method [24] to efficiently composite the images in $log_2(P)$ steps, where P is the number of rendering processors. With the fast advances of PC graphics hardware, it is now possible for a single PC graphics card to render millions of triangles at the same or even better rate than some custom graphics boards. Combining the power of these commodity graphics hardware will

result in a very high performance rendering system at relatively low cost. Many research groups have recently studied the problem of applying the fast improving PC graphics hardware for parallel rendering [30, 17, 19, 4, 32].

However, rendering invisible polygons in a scene contributes nothing to the final image and slows down the rendering process. It is particularly true for isocontouring because those invisible polygons have to also be extracted at run time. It is often the case that many polygons of a sophisticated isosurface are not visible from certain viewpoints. As isosurfaces get larger, extracting and rendering those invisible polygons take a significant amount of time with no effect to the final image. Ideally those invisible polygons should not be extracted and rendered in order to allow for faster isocontour visualization.

A similar problem exists for rendering geometric models. In order to avoid rendering invisible polygons, many techniques of visibility culling have been developed [15, 14, 34, 21, 3, 18]. The Hierarchical Z-Buffer method of Greene et al. [15] uses an octree to manage the scene, which is then traversed and rendered from top to bottom, and a quadtree to store z-buffer, which allows for fast rejection of invisible geometries. Greene [14] further extends the method by substituting the z-buffer with a 3-state hierarchical tiling and traversing the octree in a front-to-back order. Zhang et al. [34] chose some polygons from a precomputed database as occluders, and render them to get a low resolution image of the occluders. A hierarchy of occlusion maps is then built upon the image and applied to occlude the rest of the polygons in the scene. Klosowski and Silva propose a conservative visibility culling technique [21] based on their Prioritized-Layered Projection algorithm [20]. They partition the scene using an octree and render the partitions in the order of probability of being visible. After an approximate image is generated, the rest partitions are then occluded against the current z-buffer.

Isocontouring has become a major source of large surfaces. Compared to visibility culling of geometric models, view-dependent isocontouring has an added emphasis on avoiding the extraction of invisible triangles. Parker et al. present a ray-based isocontouring algorithm without explicitly extracting isosurfaces to directly compute the color of each pixel[27]. A ray is cast from the viewpoint through every pixel on the to-be-rendered image. The first intersection point along a ray with the isosurface is computed by solving a cubic equation reduced from the trilinear interpolation of the data. The parallel version of this algorithm simply assigns the rays to different processors. Liu et al. [22] propose an isosurface extraction algorithm that uses ray casting as a way to identify active cells instead of directly computing the color of pixels as in [27], and then propagates from those active cells to form a view-dependent isosurface. There may be visible holes on the isosurface extracted by this algorithm if the number of rays is not large enough.

Livnat and Hansen present a view-dependent isosurface extraction algorithm [23] that applies an occlusion culling method similar to that in [14]. It decomposes the volume into an octree hierarchy and traverses it in a front-to-back order to determine the visible cells. This algorithm produces much fewer triangles with the extra overhead of visibility test. There is no parallel implementation of this method. Gao and Shen [12] give a parallel view-dependent isocontouring algorithm that employs a multi-pass occlusion culling that tries to load balance among multiple processors and produce fewer number of triangles. Isosurface extraction is done in parallel, and visibility culling and updating occlusion maps are performed in multiple rounds. This algorithm requires the entire dataset to be in every processor's main memory and may have a bottleneck of updating the visibility masks multiple times.

Those view-dependent isocontouring algorithms have improved the interactivity of isocontour extraction and can handle larger datasets. However, all of them have assumed the entire input dataset and its ancillary data structures are stored in main memory, which limits the scalability of those algorithms. As the size of datasets grows at a much faster pace than that of available memory of parallel computers, many datasets are becoming too large to be held in the main memory of today's common parallel machines.

Main Results: In this paper we present a parallel and out-of-core view-dependent isocontouring algorithm that is well load-balanced among the parallel processors. We use a random data distribution that has provable good load balance for both view-independent and view-dependent isocontour visualization. Since data blocks are stored on disks and indexed by an external interval tree [1], only data blocks that contribute to the view-dependent isosurface are loaded and processed.

Section 2 gives the randomized data distribution method that is load-balanced for both view-independent and view-dependent isocontour visualization. Section 3 describes the details of the parallel view-dependent isocontouring algorithm with the random data distribution. Section 4 then presents the implementation and results for the algorithm.

2. Load Balance with Random Data Distribution

A very important issue of parallel computation is load balancing [10], which is achieved mostly with two fundamental approaches: (i) static balancing, where the data is partitioned beforehand with criteria that guarantee load balancing at run time; or (ii) dynamic balancing, where processors are given small chunks of data as they become available. The units of data partitioning can take the shape of slices, shafts, slabs, or blocks.

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, which is the range of all possible

isovalues in the case of isocontour visualization. Isosurface extraction has the property that computation on each subvolume of a dataset can progress independently. Since communication is the least scalable factor, we choose a static data partition method. As demonstrated later, such a static data partition can achieve good load balance for the whole range of isovalues.

The parallel and out-of-core view-dependent isocontouring is based on the same framework for scalable isocontouring on commodity off-the-shelf workstations[35]. In this framework volume datasets are divided into blocks of the same order as disk blocks and statically partitioned onto multiple processors and their local disks without duplication. An external interval tree [1] is then built for the data partition on each local disk in order to load only relevant data blocks in out-of-core computations.

There are many possible ways to distribute blocks among the processors. The ideal data partitioning for parallel computation requires the workload histogram of each processor is a same scaled version ($\frac{1}{P}$ times) of the global histogram [2]. Since the whole workload histogram of a volume dataset is the sum of workloads for all blocks, data partition problem can be thought as assigning N weighted units to P bins such that the weight difference among the bins is minimized for a range of values.

2.1. Random Data Distribution

Obviously the optimal solution for this problem is NP-hard. It is a stronger version of the optimized bin-packing problem [9], which is itself NP-hard. We have to use an approximation algorithm for the data distribution of parallel and out-of-core isocontouring. While a deterministic greedy algorithm is given in [35], in this section we show that a randomized algorithm gives a well balanced distribution for large datasets.

For a large volume dataset, one has a large number (N) of blocks to distribute among the P processors ($N >> P$). 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 $\mathbf{prob}(x_i^j = 1) = 1/P$. In average every processor p_j receives $\mathbf{E}(\sum_{i=1}^N x_i^j) = \frac{N}{P}$ blocks.

Let $\hat{w} = \max_N(w_i)$, where w_i is the workload of the block b_i for a given isovalue v. Then $W = \sum_{i=1}^N w_i$ is the total work of extracting the isosurface of isovalue v. Let $a_i = w_i/\hat{w}$; $a_i \in (0,1]$ 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

$$\hat{w} \times \mathbf{E}(\Psi_j) = \sum_{i=1}^{N} \frac{\hat{w} a_i}{P} = \sum_{i=1}^{N} \frac{w_i}{P} = \frac{W}{P}. \qquad (1)$$

Let W_j be the workload assigned to the processor p_j for a given isovalue v. 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 [28], which gives an inequality for the weighted sum of Bernoulli trials.

Theorem 2.1 ([28]) 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 $\nu > 0$

$$\mathbf{prob}(\Psi_\beta > (1+\nu)\mathbf{E}(\Psi_\beta)) < \left(\frac{e^\nu}{(1+\nu)^{(1+\nu)}}\right)^{\mathbf{E}(\Psi_\beta)} \qquad (2)$$

Using the above theorem, we can obtain following result.

Theorem 2.2 For any $r \ge P$, if the total workload $W > \alpha \hat{w} P \log r$ for a given isovalue and blocks are randomly allocated among P processors, any processor has a workload $\le \frac{2W}{P}$ with probability $\ge \left(1 - \frac{1}{r^c}\right)$, where $\hat{w}$ is the maximum workload of a block, $\alpha > \frac{c+1}{\log 4/e}$ $(c > 0)$ is a constant.

Proof: This proof is similar to the result in [13] on randomized emulation of *QSM* on BSP model. The following is based on a simplified analysis in [29].

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 $r \ge P$. Since $N >> P$, 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

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

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

$$\mathbf{prob}\left(\Psi > \frac{2W'}{P}\right) \le P \times \mathbf{prob}\left(\Psi_j > \frac{2W'}{P}\right) < r^{-c}, \quad (3)$$

where $\Psi = \max_P(\Psi_j)$ is the maximum load of all processors. Therefore, we have the theorem for the randomized data partitioning for parallel and out-of-core isocontouring.

The same theorem 2.2 applies to data partitioning for the view-dependent isocontouring in section 3, where the weight w_i of the block b_i represents the contribution of b_i to the view-dependent isosurface. Therefore, one can expect the same random block distribution algorithm to provide good load balance for parallel view-dependent isocontouring as well.

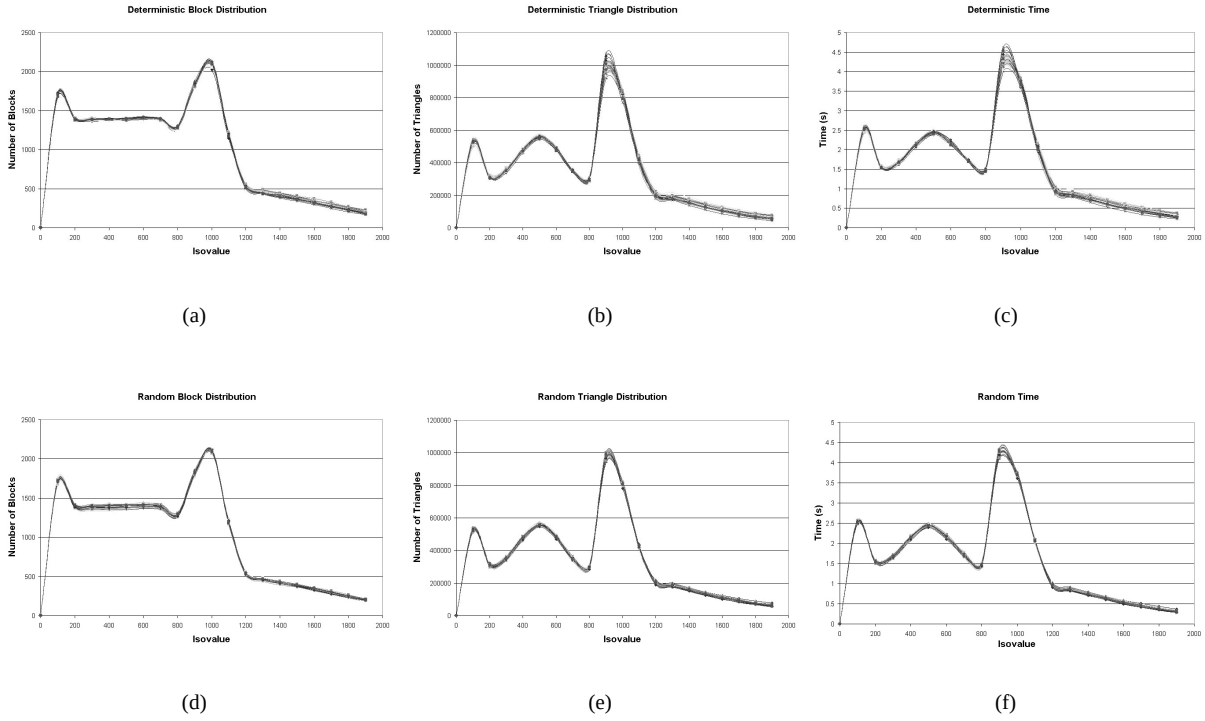


Figure 1: *Histograms of block, triangle distribution, and isosurface extraction and rendering time for both deterministic and random data partitionings of the visible human male MRI dataset.*

2.2. Comparison To a Deterministic Distribution

In this section we compare the performance of the randomized data distribution to the deterministic method in [35]. Workload histograms for the visible human male MRI data are shown in figure 1. The first row of figures show the number of blocks loaded, number of triangles, and isosurface extraction time for the deterministic 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.

While the deterministic method has better balance for the number of loaded blocks, the randomized method actually achieves better balance for real extraction time, especially for large isosurfaces. This verifies the theorem 2.2 in section 2.1.

3. Parallel and Out-of-core View-dependent Isocontouring

In this section we give a view-dependent isosurface extraction algorithm that integrates well with the framework for scalable parallel and out-of-core isocontouring in [35]. It uses the same data distribution and has the same nice features as good load balance, out-of-core, and minimum data replication. The view-dependent isocontouring algorithm has two major phases:

1. **Occluder Selection:** In the first phase of the algorithm, a number of rays are cast from the viewpoint to the volume in order to obtain an approximation of visible blocks. Those blocks are treated as *occluding blocks*. Isosurfaces inside those occluding blocks are extracted and then rendered to create an occlusion map for the second phase of the algorithm.
2. **Visibility Culling:** In the second phase, the remaining blocks are traversed and culled against the occlusion map. Only isosurfaces in the visible blocks are extracted and rendered.

Here are several good features of this view-dependent isocontouring algorithm:

- **Conservative:** For a given viewpoint, this view-dependent algorithm generates images of isosurfaces identical to those rendered without culling. There are no holes or other artifacts in the image because of visibility culling.
- **Single Pass:** The view-dependent isocontouring method requires traversing the blocks only once, and thus eliminates the large overhead of traversing the blocks and updating the occlusion map multiple times as in [12].
- **Full Parallelization:** Both the occluder selection and visibility culling phases are fully parallelizable. In our scalable isocontouring framework, each processor only extracts and renders isosurfaces of the blocks that reside on its local disk. The static data distribution methods offer good load balance for view-dependent isocontouring as well.
- **Out-of-core:** Since the view-dependent isocontouring algorithm is within the same framework that handles large volume datasets with external search data structures, it preserves the same property of loading only data blocks that contribute to the final view-dependent image.

Traversing through a volume dataset multiple times incurs significant overhead, especially when the dataset is very large. We choose a single-pass algorithm to avoid the communication overhead of multiple block traversals and occlusion map updates, which is especially expensive for a loosely connected PC cluster. In order to have an effective single pass visibility culling algorithm, it is essential to select a set of good occluders.

3.1. Occluder Selection

A subset of polygons are usually chosen as occluders for a polygonal scene. However, polygons of an isosurface do not exist before the isosurface is extracted. It is impossible to pre-compute a occluder database as in [34] for view-dependent isocontouring. In other words, the occluder selection for view-dependent isocontouring must be done at run time. In the framework of parallel and out-of-core isocontouring, datasets are statically partitioned into blocks and distributed among the processors. We must therefore consider choosing occluding blocks, which contain polygons to be rendered to create the occlusion map.

Let S_B be the set of selected occluding blocks. We use a ray-casting method described below, to choose the preliminary set of occluding blocks $S_B^1 \subset S_B$ by shooting a number of rays from the viewpoint to the volume. Along each ray, the blocks intersected by the ray are determined using Bresenham's line plotting algorithm [5], which is very fast to compute. As soon as a ray hits a block B with a range containing the isovalue v, the ray terminates and the block is added to the set S_B^1. Only the range of the block is utilized, and the scalar field of the block is not loaded. Any block B chosen this way must be a visible block since from the viewpoint to this block there exists at least one ray that is not blocked by any other polygons in front of it. We waste no time in extracting and rendering polygons in those blocks in S_B^1 for the view-dependent isosurface.

We use a combination of deterministic and randomly sampled rays to determine the blocks in S_B^1. Deterministic rays are cast from the viewpoint through the bounding box of the volume projection, such that every block is hit by at least one ray. In our implementation, we shoot a ray to the center of each boundary face of every boundary block. Besides the deterministic rays, some randomly sampled rays are also cast from the viewpoint in order to find more visible blocks. A 2D example is illustrated in figure 2, where occluding blocks chosen by the ray casting are lightly shaded.

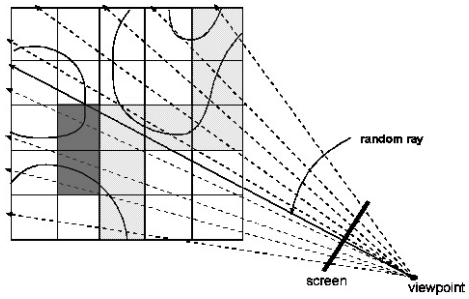


Figure 2: *Rays cast from a viewpoint into a 2D mesh. The lightly shaded blocks are the occluding blocks selected up by the rays.*

Although ray casting gives a good preliminary set of occluding blocks, some blocks that contains visible isosurfaces are not selected because other visible blocks in S_B^1 are in front of them and terminate the rays. Some examples are illustrated as the darkly shaded blocks in figure 2. Most common situations of missing blocks happen in the near neighborhood of the blocks in S_B^1 because of the continuity of isosurfaces. Those missed blocks may cause holes in the occlusion map that is constructed from the polygons in the occluding blocks. In order to reduce the number and size of those holes, we add the nearest neighboring blocks of those in S_B^1 to form the set of occluding blocks S_B. The pseudo-code for the occluder selection process is shown in figure 3.

Figure 8 (see color plates) shows some examples of occluder selection, where the pictures in the left column are images rendered only with polygons in the set of blocks S_B^1 from ray casting and the pictures in the middle column are based on the combination of S_B^1 and their nearest neighbors. We can observe that the images in the middle column are quite close to the final images shown in the right column. Therefore, those images in the middle column are good candidates for occlusion maps. The quality of occlusion maps is very important in a single-pass visibility culling method. We should note here that the cost of ray casting to compute the occluding blocks is only a very small fraction of the total isosurface extraction and rendering time.

Occluder Selection

```
Initialize the set of occluding blocks S_B = φ.
x = the query isovalue.

select_occluders(x) {
   S_B^1 = φ is the set of occluding blocks from ray casting.
   P_1 = a set of center points of all boundary faces.
   P_2 = a set of randomly chosen points on the image plane.

   for(all point p in P_1 ∪ P_2) {
      cast a ray r from the viewpoint v to p
      b = the first block intersected by the ray r.
      while (r is within the volume) {
         if(min(b) ≤ x and max(b) ≥ x) {
            S_B^1 = S_B^1 ∪ {b}
            break
         }
         b = the next block along the ray r
      }
   }
   for(each block b in S_B^1) {
      (i, j, k) = the index of b
      for(block b' of index (i ± 1, j ± 1, k ± 1)) {
         if(min(b') ≤ x and max(b') ≥ x & b' ∉ S_B^1)
            S_B = S_B ∪ {b'}
      }
   }
   S_B = S_B ∪ S_B^1
}
```

Figure 3: *Pseudo-code for occluder selection*

3.2. Visibility Culling

At the second stage of view-dependent isocontouring, we first extract isosurfaces from the occluding blocks in S_B and render them to create an occlusion map. After the occlusion map is constructed, the remaining blocks are traversed and culled against the occlusion map. As suggested in [14, 34, 3, 18], a hierarchical occlusion map may be constructed in order to decrease the time of visibility tests and occlusion map updates. Since the culling time is much less than the actual extraction time and no updates are done for the single-pass method, we can just use a single-level occlusion map for its simplicity. However, hierarchical maps are necessary for the faster demands of real-time view-dependent rendering.

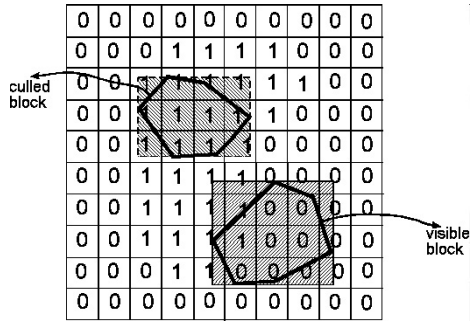


Figure 4: *Examples of visibility culling with the visibility masks shown.*

An occlusion map or mask is a 2*D* bitmap corresponding to the image rendered with polygons extracted from the blocks in S_B. An entry of the mask is set to 1 if the corresponding pixel in the rendered image is covered, otherwise it is set to 0 as shown in figure 4. The visibility tests are applied to only those blocks whose ranges contain the isovalue *v*. Eight vertices of such a block *b* are projected into the image plane. If there is any 0 entry of the occlusion map within the projected area, *b* is considered as visible. Rather than computing the exact projection area of *b*, we use the rectangular bounding box of *b*'s projection area to simplify the computation. This culling process also includes the frustum culling because blocks with projected bounding boxes outside the screen are automatically culled. The pseudo-code for visibility culling is shown in figure 5.

Visibility Culling

```
for(blocks b in S_B) {
   extract and render polygons from b
}
Construct an occlusion map M from the rendered image.

L = a list of all remaining blocks
b = the first element of L
while ( b ) {
   if(b ∉ S_B and min(b) ≤ v ≤ max(b)) {
      project the block b
      rect = the bounding box of b's projection area
      if(∃ a pixel (i, j) ∈ rect with M[i, j] == 0) {
         extract and render polygons from b
      }
   }
   b = the next block in L
}
```

Figure 5: *Pseudo-code for visibility culling*

We do not update the occlusion map during the culling process when polygons for a new blocks are rendered. It is arguable that updating the occlusion map would further cut the number of invisible polygons extracted. Although a few extra polygons might be generated using a fixed occlusion map, it pays off in avoiding the high overhead of updating the occlusion maps many times as in a multi-pass method [12]. Since we start from a good approximation of the final image, the number of extra polygons tends to be small. A fixed visibility mask allows any order of traversing blocks while updating an occlusion map requires a strict front-to-back order. Parallel implementations of a fixed mask method also generate the same number of triangles as the sequential case. Section 4 will compare the performance of the view-dependent isocontouring method to that without visibility culling. Data blocks and its isosurfaces are cached in main memory, such that a change of view parameters does not require recomputing isosurfaces in all blocks.

3.3. Parallelization

It is straightforward to parallelize the view-dependent isocontouring algorithm. The occluder selection stage can be parallelized by assigning rays to different processors. The

entire set of occluding blocks S_B is then the union of occluding blocks found by different processors.

Since data blocks are statically distributed among the processors, each processor has a subset of the occluding blocks in S_B. After a processor extracts and renders polygons in its subset of occluding blocks, it constructs a partial occlusion map. Those partial occlusion maps are then collected and merged into a single mask, which is the same as that in the sequential case. The merged occlusion map will be used to cull out invisible blocks in a view-dependent isocontour query. In the visibility culling phase, each processor checks only the blocks that reside on its local disk. The steps of the parallel view-dependent isocontouring algorithm are as follows:

1. Divide the rays to be cast equally among the P processors. Each processor fires $\frac{R}{P}$ rays to the volume, where R is the total number of rays needed to select S_B in the sequential algorithm. In this step, the range information of each block is replicated on all processors. The rays are assigned in a spatial coherent manner for better cache performance.
2. The entire set of occluding blocks S_B is constructed as the union of occluding blocks on all machines and broadcast to all processors.
3. Each processor p_i queries its external interval tree T to find a set of blocks S_v^i that resides on its local disk and intersects the isosurface. For each block $b \in S_v^i \cap S_B$, isosurfaces are extracted from b for a given isovalue v and rendered to the image plane. A partial visibility mask of bits 0 and 1 is constructed from the rendered image on each processor.
4. The occlusion maps from all processors are collected and merged into a common mask, and the merged mask is broadcast to all processors.
5. Each processor p_i traverses the remaining blocks $b \in S_v^i - S_B$ and projects b to the image plane. If b is visible, it extracts and renders the portion of isosurface from b.
6. The final image is constructed by compositing the image buffers from all the processors with the Metabuffer.

All the collecting and merging operations in the parallel algorithm can be performed using the binary swap method [24] and in run-length compressed format [26] for efficient utilization of the network. In the current implementation, we just apply the standard collective MPI functions such as MPI_Allreduce() for the communication. The extra information for parallel and out-of-core view-dependent isocontouring is only the range information of all blocks in the step of ray casting. The range information for a block is usually two floating point numbers and extremely small compared to the actual data of the block.

4. Implementation and Results

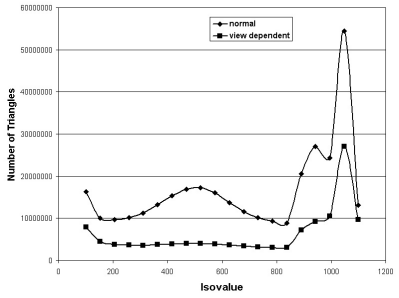
We use the same implementation platform as that in [35], because the view-dependent isocontouring shares the same framework and data distribution as the parallel and out-of-core isocontouring. Each computational node is a Compaq SP750 PC workstation, which consists of a 800MHZ Pentium III processor, 256MB main memory, a 9GB system disk and a 18GB data disk, and an nVdia Geforce II graphics card. These nodes are interconnected by a 100 Mb/s ethernet. These machines run Linux (kernel 2.2.18) as the operating system and each disk block size is 4,096 bytes.

Figure 9 (see color plates) shows the resulting images from applying the view-dependent isocontouring method to the visible human foot dataset. The view-dependent method renders the same image as the isocontouring algorithm with no culling, but it extracts only a fraction of triangles for a given viewpoint. Figure 9 (c) rotates the view-dependent isosurface to illustrate the portion that is not extracted.

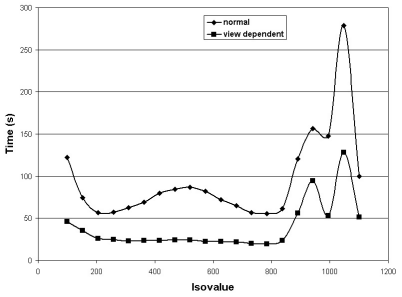
We further test the view-dependent isocontouring algorithm on the entire visible human male MRI dataset ($512 \times 512 \times 1252$), whose size 656MB is much greater than the main memory of a single machine, in order to demonstrate the out-of-core feature of our algorithm. In this example we choose the dimension of blocks as $10 \times 10 \times 20$, such that each block has approximately the same size as a disk block. Figure 6 compares the extraction time and the number of triangles extracted by the view-dependent isocontouring algorithm to the normal isosurface extraction method without visibility culling. Figure 6 (a) shows the actual number of triangles extracted by the two different methods, and figure 6 (b) compares the actual extraction and rendering time with varying isovalues for a fixed viewpoint on a single processor.

The view-dependent isocontouring algorithm is easily parallelized in the framework of parallel and out-of-core isocontouring. It is interesting to measure the speedup of the parallel implementation on the PC cluster. We apply the random distribution method in section 2 to distribute blocks among the processors. Following the similar analysis in theorem 2.2 by replacing the parameter a_i as the view-dependent load of the ith block, we expect good load balance among the processors for view-dependent isocontouring as well. We do not measure the image composition time because the final image shall be composited by the Metabuffer hardware with minimal constant delay.

We test the parallel implementation with an isovalue 800 for the visible male MRI dataset. Figure 7(a) shows the individual view-dependent isosurface extraction and rendering time for 2, 4, 8, and 16 processors respectively. The flatness of those curves demonstrates the good balance among the processors. The corresponding speedup curve is shown in figure 7(b). As the number of processors increases, the actual isosurface extraction and rendering time for each processor decreases proportionally in our implementation. The parallel view-dependent isocontouring algorithm requires two communication rounds to merge the set of occluding blocks and the occlusion map, which differs from the view-independent parallel isocontouring. The inter-processor communication

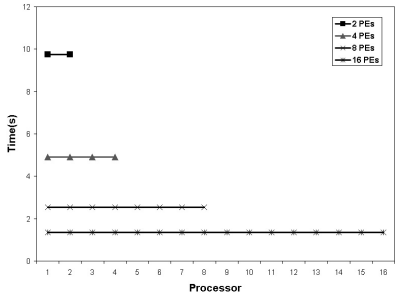


(a)

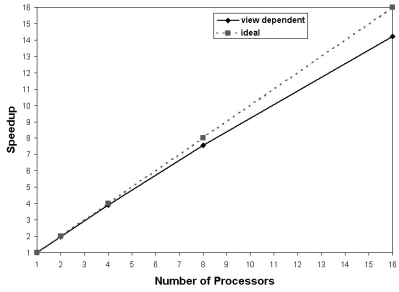


(b)

Figure 6: *Comparisons between normal and view-dependent isocontouring: (a) the number of extracted triangles as a function of isovalue, (b) the extraction plus rendering times of the two methods on a single processor. The upper and lower curves are the results for normal and view-dependent isocontouring respectively.*



(a)



(b)

Figure 7: *(a) Individual processor time for extracting and rendering a view-dependent isosurface (isovalue = 800) for the visible human male MRI dataset with 2, 4, 8, and 16 processors respectively. (b) Speedup chart of the same view-dependent isosurface extraction plus rendering times compared to the ideal case.*

cost may increase as the number of processor increases. We implement the merge of occluding blocks and occlusion masks with the *MPI_Allreduce*() function of the MPI library on the 100 Mb/s ethernet. A faster network and an implementation utilizing compression would reduce such communication cost and even improve the speedup.

5. Conclusion

In this paper we presented a randomized static data distribution for parallel isocontouring on commodity off-the-shelf clusters. We have proved that with high probability no processor would have much higher workload in this distribution if there are many blocks. Such a distribution is well balanced for both parallel view-independent and view-dependent isosurface extraction. Based on the random data distribution, we propose a parallel and out-of-core view-dependent isocontouring algorithm that is conservative and well load balanced. We demonstrate significant performance improvement over normal isosurface extraction and good speedup in a parallel implementation on PC clusters.

Acknowledgments: Dr. Chandrajit Bajaj and Dr. Xiaoyu Zhang's research was supported in part by grants ACI-9982297, CCR-9988357 from the National Science Foundation, a DOE-ASCI grant BD4485-MOID-1 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. Dr. Vijaya Ramachandran's research was supported by Texas Advanced Research Program Grant 00365-0029-1999 and NSF Grant CCR-9988160.

References

1. ARGE, L., AND VITTER, J. S. Optimal interval management in external memory. In *Proc. IEEE Foundations of Computer Science* (1996), pp. 560–569.

2. BAJAJ, C., PASCUCCI, V., THOMPSON, D., AND ZHANG, X. Parallel acclerated isocontouring for out-of-core visualization. In *Proceedings of IEEE Parallel Visualization and Graphics Symposium* (October 1999), pp. 97–104.

3. BARTZ, D., MEIβNER, M., AND HUTTNER, T. Opengl-assisted occlusion culling for large polygonal models. *Computer and Graphics 23*, 5 (1999).

4. BLANKE, W. J., FUSSELL, D., BAJAJ, C., AND ZHANG, X. The metabuffer: A scalable multiresolution multidisplay 3-d graphics system using commodity rendering engines. Tr2000-16, Computer Science, University of Texas at Austin, February 2000.

5. BRESENHAM, J. E. Algorithm for computer control of a digital plotter. *IBM Systems Journal 4*, 1 (1965), 25–30.

6. CHIANG, Y., AND SILVA, C. T. I/O optimal isosurface extraction. In *IEEE Visualization ́97* (Nov. 1997), R. Yagel and H. Hagen, Eds., IEEE, pp. 293–300.

7. CHIANG, Y.-J., FARIAS, R., SILVA, C. T., AND WEI, B. A unified infrastructure for parallel out-of-core isosurface and volume rendering of unstructured grids. In *Proceedings of IEEE Parallel Visualization and Graphics Symposium* (October 2001).

8. CHIANG, Y.-J., SILVA, C. T., AND SCHROEDER, W. J. Interactive out-of-core isosurface extraction. In *Proceedings of the 9th Annual IEEE Conference on Visualization (VIS-98)* (Oct. 18–23 1998), ACM Press, pp. 167–174.

9. COFFMAN, E. G., GAREY, M. R., AND JOHNSON, D. S. Approximation algorithms for bin packing: a survey. In *Approximation algorithms*, D. Hochbaum, Ed. 1996.

10. CROCKETT, T. W. Parallel rendering. Tech. rep., ICASE, 1995.

11. ELLSIEPEN, P. Parallel isosurfacing in large unstructed datasets. In *Visualization in Scientific Computing* (1995), Springer-Verlag, pp. 9–23.

12. GAO, J., AND SHEN, H.-W. Parallel view-dependent isosurface extraction using multi-pass occlusion culling. In *Proceedings of IEEE Parallel Visualization and Graphics Symposium* (October 2001).

13. GIBBONS, P., MATIAS, Y., AND RAMACHANDRAN, V. Can a shared memory model serve as a bridging model for parallel computation. *Theory of Computing Systems (Special Issue on Papers from SPAA '97) 32*, 3 (1999), 327–359.

14. GREENE, N. Hierarchical polygon tiling with coverage masks. *Computer Graphics 30*, Annual Conference Series (1996), 65–74.

15. GREENE, N., KASS, M., AND MILLER, G. Hierarchical Z-buffer visibility. *Computer Graphics 27*, Annual Conference Series (1993), 231–238.

16. HANSEN, C., AND HINKER, P. Massively parallel isosurface extraction. In *Visualization '92* (September 1992).

17. HEIRICH, A., AND MOLL, L. Scalable distributed visualization using off-the-shelf components. In *Parallel Visualization and Graphics Symposium – 1999* (San Francisco, California, October 1999), J. Ahrens, A. Chalmers, and H.-W. Shen, Eds.

18. HEY, H., TOBLER, R. F., AND PURGATHOFER, W. Real-time occlusion culling with a lazy occlusion grid. Tech. Rep. TR-186-2-01-02, Institute of Computer Graphics and Algorithms, Vienna University of Technology, January 2001.

19. HUMPHREYS, G., AND HANRAHAN, P. A distributed graphics system for large tiled displays. In *Proceedings of IEEE Visualization Conference* (1999), pp. 215–223.

20. KLOSOWSKI, J., AND SILVA, C. Rendering on a budget: A framework for time-critical rendering. In *IEEE Visualization 1999* (1999), pp. 115–122.

21. KLOSOWSKI, J. T., AND SILVA, C. T. Efficient conservative visibility culling using the prioritized-layered projection algorithm. In *Proceedings of the 2001 IEEE Visualization Conference* (2001).

22. LIU, Z., FINKELSTEIN, A., AND LI, K. Multi-resolution view dependent isosurface propagation, 2000.

23. LIVNAT, Y., AND HANSEN, C. View dependent isosurface extraction. In *IEEE Visualization '98* (1998), D. Ebert, H. Hagen, and H. Rushmeier, Eds., pp. 175–180.

24. MA, K.-L., PAINTER, J. S., HANSEN, C. D., AND KROGH, M. F. Parallel volume rendering using binary-swap image composition. *IEEE Computer Graphics and Applications 14*, 4 (July 1994).

25. MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (July 1994).

26. MORELAND, K., WYLIE, B., AND PAVLAKOS, C. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of IEEE Par-*

allel Visualization and Graphics Symposium (October 2001).

27. PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P. Interactive ray tracing for isosurface rendering. In *Visualization '98* (October 1998).

28. RAGHAVAN, P. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Science 37* (1988), 130–143.

29. RAMACHANDRAN, V. A general purpose shared-memory model for parallel computation. *Algorithms for Parallel Processing, Volume 105, IMA Volumes in Mathematics and its Applications* (1999), 1–17.

30. SAMANTA, R., ZHENG, J., FUNKHOUSER, T., LI, K., AND SINGH, J. P. Load balancing for multi-projector rendering systems. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (August 1999).

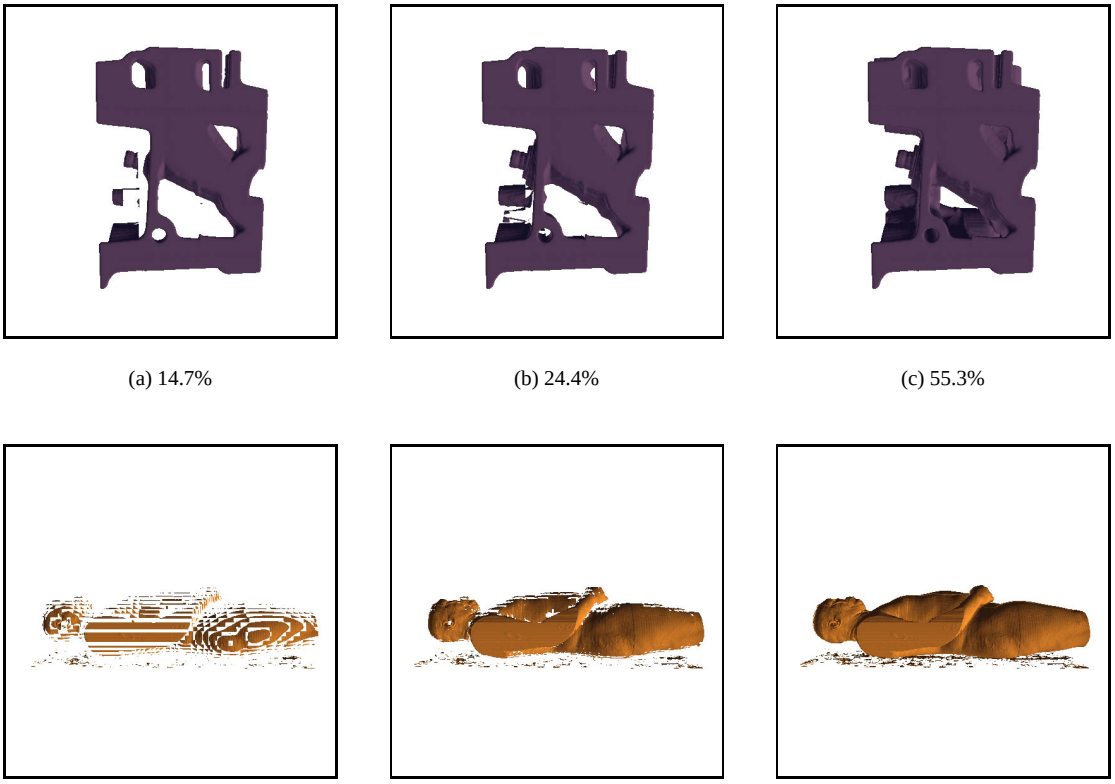
31. SHEN, H., HANSEN, C., LIVNAT, Y., AND JOHNSON, C. Isosurfacing in span space with utmost efficiency (issue). In *Visualization '96* (1996), pp. 287–294.

32. STOLL, G., ELDRIDGE, M., PATTERSON, D., WEBB, A., BERMAN, S., LEVOY, R., CAYWOOD, C., TAVEIRA, M., HUNT, S., AND HANRAHAN, P. Lightning-2: A high-performance display subsystem for pc clusters. In *Proceedings of SIGGRAPH 2001* (August 2001), pp. 141–149.

33. UDESHI, T., AND HANSEN, C. Parallel multipipe rendering for very large isosurface visualization. In *Joint EUROGRAPHICS - IEEE TCCG Symposium on Visualization* (1999).

34. ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF III, K. E. Visibility culling using hierarchical occlusion maps. *Computer Graphics 31*, Annual Conference Series (1997), 77–88.

35. ZHANG, X., BAJAJ, C., AND BLANKE, W. Scalable isosurface visualization of massive datasets on cots clusters. In *Proceedings of IEEE Parallel Visualization and Graphics Symposium* (October 2001).



(a) 14.7% (b) 24.4% (c) 55.3%

(d) 5.3% (e) 13.2% (f) 41.7%

Figure 8: *Some examples of view-dependent isosurface rendering, where the percentage of triangles to view-independent isosurfaces are shown. The pictures in the left column are occlusion maps generated by only ray casting; the middle pictures are occlusion maps from both blocks hit by ray casting and their nearest neighbors; the right pictures are final images rendered by the visibility culling method.*

(a) 320909 triangles (b) 199670 triangles (c) 199670 triangles

Figure 9: *A normal isosurface (a) looks exactly the same as a view-dependent isosurface (b) from a given view point (isovalue = 500). (c) shows the rotated surface of (b), where the red area is the portion of the isosurface that is not extracted in the view-dependent isocontour visualization.*