Volume 67 Number 5 May 2007 ISSN 0743-7315
Journal of Parallel and Distributed Computing
EDITORS
Allan Gottlieb, New York University
Kai Hwang, University of Southern California
Sartaj Sahni, University of Florida
• Theory of Parallel/Distributed Computing
• Innovative Computer Architectures
• Supercomputers and Numerical Methods
• Software Tools and Environments
• Parallel Algorithms and Implementations
• Languages, Compilers, and Operating Systems
• Shared Memory Multiprocessors
• Multicomputers and Distributed Processing
• Fault-Tolerant Computing
• Artificial Intelligence Machines
• Applications and Performance Analysis
• Optical Computing
• Neural Networks
Available online at
ScienceDirect
www.sciencedirect.com
ELSEVIER

Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 67 (2007) 592–603
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

An efficient and scalable parallel algorithm for out-of-core isosurface extraction and rendering[☆]

Qin Wang[a,*], Joseph JaJa[a,1], Amitabh Varshney[b,1]

[a]*Institute for Advanced Computer Studies (UMIACS), Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742, USA*
[b]*Institute for Advanced Computer Studies (UMIACS), Department of Computer Science, University of Maryland, College Park, MD 20742, USA*

Received 30 May 2006; received in revised form 14 September 2006; accepted 19 December 2006
Available online 30 January 2007

Abstract

We consider the problem of isosurface extraction and rendering for large scale time-varying data. Such data sets have been appearing at an increasing rate especially from physics-based simulations, and can range in size from hundreds of gigabytes to tens of terabytes. Isosurface extraction and rendering is one of the most widely used visualization techniques to explore and analyze such data sets. A common strategy for isosurface extraction involves the determination of the so-called active cells followed by a triangulation of these cells based on linear interpolation, and ending with a rendering of the triangular mesh. We develop a new simple indexing scheme for out-of-core processing of large scale data sets, which enables the identification of the active cells extremely quickly, using more compact indexing structure and more effective bulk data movement than previous schemes. Moreover, our scheme leads to an efficient and scalable implementation on multiprocessor environments in which each processor has access to its own local disk. In particular, our parallel algorithm provably achieves load balancing across the processors independent of the isovalue, with almost no overhead in the total amount of work relative to the sequential algorithm. We conduct a large number of experimental tests on the University of Maryland Visualization Cluster using the Richtmyer–Meshkov instability data set, and obtain results that consistently validate the efficiency and the scalability of our algorithm.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Parallel isosurface extraction; Scientific visualization

1. Introduction

During the past few years, we have seen an increasing trend towards the generation and analysis of very large time-varying data sets in scientific simulation. Such data sets are characterized by their very large sizes ranging from hundreds of gigabytes to tens of terabytes with multiple superposed scalar and vector fields, demanding an imperative need for new interactive exploratory visualization capabilities. As an example of such a data set, consider the fundamental mixing process of the Richtmyer–Meshkov instability in inertial confinement fusion and supernovae from the ASCI team at the Lawrence Livermore National Labs.[2] This data set represents a simulation in which two gases, initially separated by a membrane, are pushed against a wire mesh. These are then perturbed with a superposition of long wavelength and short wavelength disturbances and a strong shock wave. This simulation took 9 days on 960 CPUs and produced about 2.1 TB of simulation data. The data show the characteristic development of bubbles and spikes and their subsequent merger and break-up over 270 time steps. Each time step, simulated over a $2048 \times 2048 \times 1920$ grid, has isosurfaces exceeding 500 million triangles with an average depth complexity of 50. Such high-resolution simulations allow elucidation of fine scale physics; in particular, when compared with coarser resolution cases, the data allow

[☆] A preliminary version of this work appeared in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2006.
[*] Corresponding author.
E-mail addresses: qinwang@umiacs.umd.edu (Q. Wang), joseph@umiacs.umd.edu (J. JaJa), varshney@cs.umd.edu (A. Varshney).
[1] This work supported by the NSF research infrastructure Grant CNS-04-03313.

[2] http://www.llnl.gov/CASC/asciturb/.

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2006.12.007

observations of a possible transition from a coherent to a turbulent state with increasing Reynolds number. Although there are a large set of visualization systems and techniques, they are usually targeted for several orders of magnitude smaller data sets, may just consider the issues of data representation and visualization in a fragmented manner, and do not scale to the terabyte-sized data sets. Visual interaction with large databases of dynamic simulation data sets requires the development of a high-performance graphics software infrastructure running on visualization platforms with access to large scale storage. In this paper, we develop provably scalable and efficient strategies for the parallel out-of-core isosurface extraction and rendering of time-varying scalar fields. Compared with other published algorithms, our approach has the following advantages:

- Our serial algorithm uses a smaller indexing structure and a more effective bulk data movement than the best known previous algorithms while achieving similar asymptotic bounds. In particular, the size of our indexing structure is shown to be orders of magnitude smaller than that of the interval tree for a number of well-known data sets.
- Our scheme can be implemented on a distributed storage multiprocessor environment such that the data distribution across the local disks of the different processors results in a *provably* balanced workload irrespective of the isovalue. Moreover, the total amount of work across the different processors is about the same as that required by our efficient serial algorithm.
- Our experimental results show that we can generate and render isosurfaces at the rate of $3.5 \sim 4.0$ million triangles per second on the Richtmyer–Meshkov data set using our algorithm on a single processor. On a 16-node cluster, we achieve scalable performance across widely different isovalues with a performance of up to 60 million triangles per second. The experimental results also show that our algorithm achieves excellent load balancing for a variety of data sets over a wide range of isovalues.

We make use of the University of Maryland visualization cluster in which each node consists of a two-way symmetric multiprocessor with 8 GB of main memory, a 60 GB local disk, and an NVIDIA GPU (Graphics Processing Unit). The nodes are interconnected via a 10 Gbps InfiniBand, with four nodes reserved for compositing the image frame buffer outputs of other processors and displaying the results on a wall-sized screen (multi-projector display).

2. Previous work

The Marching Cubes algorithm developed in [14] scans the entire cell set, one cell at a time, and determines whether a cell is cut by the isosurface, and in the affirmative generates a local triangulation of the isosurface intersection with the cell. The overall triangular mesh is then rendered on a screen. Many improvements to this initial algorithm have been reported in the literature. Some algorithms attempt to reduce the number of examined cells by using a spatial data structure such as the octree [21,22], while others partition the range of the scalar field values and construct an index on such a partition [19]. Note that the spatial data structures can only be used for structured grids (such as the Richtmyer–Meshkov data set), while the range partitioning schemes can be used for both structured and unstructured grids. A hybrid scheme that works only for structured grids determines a collection of "seed cells" and perform contour propagation starting from some seed cells that depend on the isovalue [3,13]. A theoretically optimal algorithm was described in [9], and involves the construction of an interval tree on the scalar field intervals defined by the cells [2]. Such a data structure enables the exploration of only the active cells (cells that intersect the isosurface) and hence it is output sensitive. This algorithm was later generalized into a theoretically optimal out-of-core isosurface extraction strategies in [7,8]. For multiprocessor environments, several parallel algorithms have been reported for the case when the data can fit in the main memory [10,11,15,17–19]. Of more interest to us are the parallel out-of-core algorithms such as those reported in [4,6,20,23–25]. We proceed to briefly discuss some of the recent algorithms and relate them to the work described in this paper.

We assume a multiprocessor environment in which each processor has its own local disk. The processors communicate and exchange data through an interconnection network using message passing. There is no shared storage pool. The initial input is assumed to be partitioned among the disks of the different processors. Therefore, a parallel out-of-core algorithm has to deal with (i) data layout among the parallel disks and the indexing structure to access the data; (ii) determining active cells and generating the corresponding triangles using the available processors; and (iii) rendering and displaying the output. Critical factors that influence the performance include the amount of work required to generate the index and organize the data on the different disks (preprocessing step); the relative computational loads of the different processors corresponding to an arbitrary isovalue; the performance of the rendering and rasterization into a single display; and the total amount of work relative to the (best) sequential algorithm. The preprocessing step described in [8,5,6] involves partitioning the data set into *metacells*, where each metacell is a cluster of neighboring cells and occupies about the same number of disk blocks (in particular, the metacell is trivially a subcube in the case of structured grids), and building a B-tree like interval tree, called binary-blocked I/O interval tree (BBIO tree). The computational cost of this step is asymptotically similar to an external sort, which is likely to be expensive in practice. The isosurface generation requires that a host traverses the BBIO tree to determine the indices of the active metacells, after which jobs are dispatched on demand to the available processors. In addition to the substantial preprocessing overhead, a significant bottleneck with this scheme is the host overhead in coordinating and dispatching jobs, and the unpredictability of the access pattern to the available disks. The algorithm described in [23] attempts to solve the load balancing problem by distributing the data based on a range space partition. The range of possible field values is partitioned into a number of intervals. Blocks are then assigned to triangular matrix entries depending on which intervals a block spans. An external interval tree (BBIO tree) is then built separately for the data on each processor. Again this strategy involves a very expensive preprocessing step but in addition there is no guarantee of load balancing among the processors. In fact, it is easy to construct a data set that will result in extremely unbalanced loads among the processors. Moreover, while the BBIO is "asymptotically optimal" for I/O performance (in terms of the number of active metacells brought to memory and not necessarily the active cells), it does incur a significant overhead in terms of size and performance. The extracted local surface is then streamed to parallel rendering servers, followed by compositing the outputs of the different frame buffers to a tiled-display. The preprocessing algorithm described in [25] is based on partitioning the range of scalar values into equal-sized subranges, creating afterwards a file of subcubes for each subrange. The blocks in each range file are then distributed across the different processors, based on a work estimate of each block. As in [23], the preprocessing is computationally expensive and there is no guarantee that the loads of the different processors will be about the same in general.

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

3. Computational model

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

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

4. Compact interval tree indexing scheme

Our algorithm can handle both structured and unstructured grids and makes use of the metacell notion introduced in [8]. In general, a metacell consists of a cluster of neighboring cells. All the metacells are about the same size, which is a small multiple of the disk block size. In particular, for the structured grid of the Richtmyer–Meshkov data set, our metacell consists of a subcube of size $9 \times 9 \times 9$, represented by a list of the scalar values appearing in a predefined order. Our indexing structure and isosurface query algorithm are designed upon the concept of metacell. With each metacell, we associate an interval $(v_{\min}, v_{\max})$ corresponding, respectively, to the minimum and maximum values of the scalar field over the metacell. Our compact interval tree structure makes use of the span space concept to organize the data layout. Before introducing this structure, we begin with a brief review of the standard binary interval tree.

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

Our compact interval tree is similar to the interval tree except that we do not store the two sorted lists of intervals at each node. Instead, we store the distinct values of the $v_{\max}$ endpoints of these intervals, sorted in decreasing order, and associate with each such value a pointer to a list of intervals sorted in increasing order of left endpoint value $v_{\min}$. We now explain the compact interval tree in the context of the isosurface problem and its relationship to the metacells generated from the input data. Consider the span space consisting of all possible combinations of the $(v_{\min}, v_{\max})$ values of the scalar field. With each such pair we associate a list containing the metacells whose minimum scalar field value is $v_{\min}$ and whose maximum scalar field value is $v_{\max}$. The essence of the scheme for our compact interval tree is illustrated through Fig. 1 representing the span space, and Fig. 2 representing the compact interval tree built upon the n distinct values of the endpoints of the intervals corresponding to the metacells.

Let v_{m0} be the median of all the endpoints. The root of the interval tree corresponds to all the intervals whose $v_{\min}$ values fall in the range $[v_0, \ldots, v_{m0}]$, and whose $v_{\max}$ values fall in the range $[v_{m0}, \ldots, v_n]$. Such intervals are represented as points in the square of Fig. 1 whose bottom right corner is located at (v_{m0}, v_{m0}). We group together all the metacells having the same $v_{\max}$ value in this square, and store them consecutively on disk from left to right in increasing order of their $v_{\min}$ values. We refer to this contiguous arrangement of all the metacells having the same $v_{\max}$ value within a square as a *brick*. The bricks within the square are in turn stored consecutively on disk in decreasing order of the $v_{\max}$ values. The root will contain the value v_{m0}, the number of non-empty bricks in the corresponding square (of the span space), and an index list of the corresponding bricks. This index list consists of at most $n/2$ entries corresponding to the non-empty bricks, each entry containing three fields: the $v_{\max}$ value of the brick, the *smallest* $v_{\min}$ value of the metacells in the brick, and a pointer that indicates the start position of the brick on the disk. Each brick contains contiguous metacells in increasing order of $v_{\min}$ values, and each metacell consists of its $v_{\min}$ value, its location information such as metacell ID, and a list of the scalar field values of the vertices (in a predefined order) within the metacell. We recursively repeat the process for the left and right children of the root. We will then obtain two smaller squares whose bottom right corners are located, respectively, at (v_{m10}, v_{m10}) and (v_{m11}, v_{m11}) in the span space, where v_{m10} and v_{m11} are the median values of the endpoints of the intervals associated, respectively, with the left and right subtrees of the root. In this case, each child will have at most $n/4$ non-empty index entries associated with its corresponding bricks on the disk. This recursive process is continued until all the intervals are exhausted. At this point we have captured all possible $(v_{\min}, v_{\max})$ pairs and their associated metacell lists.

Note that the size of the standard interval tree is typically much larger than the size of our indexing structure. We can upper bound the size of our compact interval tree as follows. There are at most $n/2$ index entries at each level of the compact interval tree and the height of the tree is no more than $O(\log_2 n)$. Hence our compact interval tree consists of $O(n \log n)$ index entries, each entry having three fields. Therefore the total size of our compact interval tree is $O(n \log n)$, while the size of the standard interval tree is $\Omega(N)$, where N is the total number of intervals and hence can be as large as $\Omega(n^2)$. In fact, the standard interval tree is always at least twice as large as our indexing structure regardless of the relative values of N and n since the interval tree stores each interval $(v_{\min}, v_{\max})$ twice while our indexing structure stores each such interval at most once (the extra space required for pointers to the corresponding data on disk is the same in both cases). In Table 1 we compare the sizes of the two indexing structures for some well-known data sets from LLNL, the Stanford Volume Data Archive and IEEE visualization. As can be seen from the table, our indexing structure is substantially smaller than the standard interval tree, even in the case when $N \approx n$.

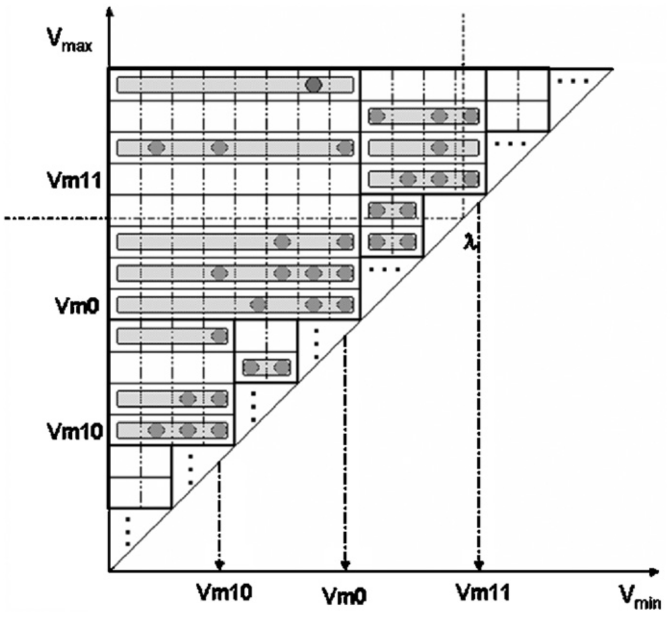

Fig. 1. Span space partitioning scheme for our indexing structure.

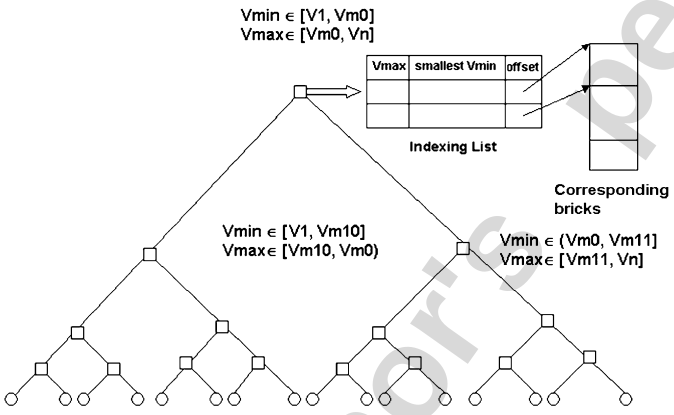

Fig. 2. Compact interval tree structure and the associated metacell lists.

Table 1
Size comparison between standard and compact interval trees

Data set name	Scalar field size	N	n	Size of interval tree	
				Standard	Compact
LLNL[a]	One bytes	18,970	227	222.3 KB	6.0 KB
MRBrain[b]	Two bytes	756,982	2894	11.6 MB	22.6 KB
CTHead[b]	Two bytes	817,642	3238	12.5 MB	25.3 KB
Pressure[c]	Four bytes	24,507,104	20,748,433	560.9 MB	237.4 MB
Velocity[c]	Four bytes	24,444,597	17,548,131	559.5 MB	200.8 MB

N denotes the number of distinct intervals and n represents the number of distinct $v_{\max}$ values.
[a]Richtmyer–Meshkov data set from http://www.llnl.gov/CASC/asciturb/.
[b]3D CT images from http://graphics.stanford.edu/data/voldata/.
[c]Simulation data of a Hurricane from http://vis.computer.org/vis2004contest/data.html.

5. Efficient and scalable isosurface extraction algorithm

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

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

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

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

5.1. Parallel processing

Assume that we have p processors, each with its own local disk, and the processors are interconnected with some type of a high-speed interconnection network. We will show how to partition the input data among the p local disks and apply our compact tree indexing structure to extract and render isosurfaces in a scalable and efficient way. The main challenge is to ensure load balancing among the processors for any isovalue while maintaining the same total amount of work as that of our sequential algorithm. We start by showing how to distribute the N input metacells among the local disks in such a way that the active metacells corresponding to any isovalue are spread almost evenly among the processors. We first sort N metacells in decreasing order of their $v_{\max}$ values. The N sorted metacells are then divided evenly into $\sqrt{N/p}$ sets, each set containing $\sqrt{pN}$ consecutive metacells in the sorted order. Then, we re-sort the metacells within each set by increasing order of $v_{\min}$ values. We now stripe all the N metacells as they appear after the two sorting steps across the p disks. That is, the first metacell is stored on the disk of the first processor, the second on the disk of the second processor, and so on wrapping around as necessary. We next prove the following property of our partitioning scheme.

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

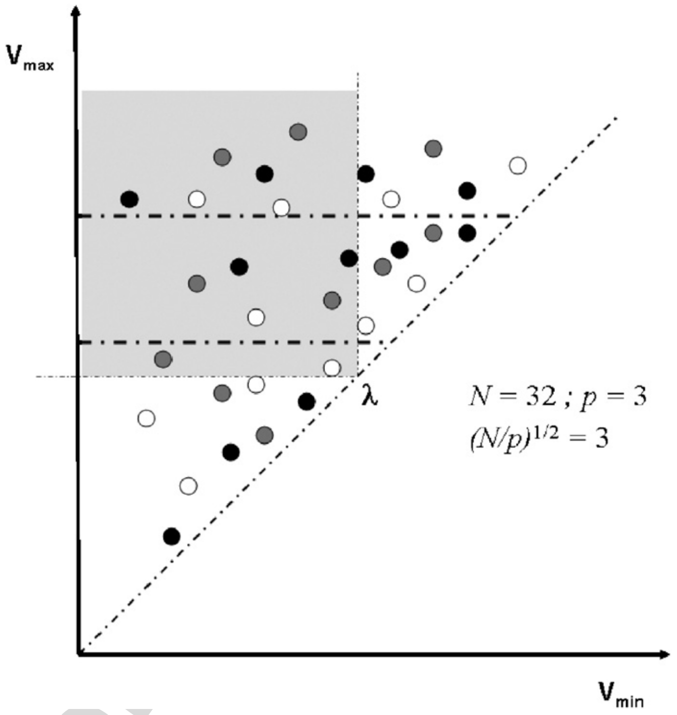

Fig. 3. Each pair $(v_{\min}, v_{\max})$ is illustrated as a point and assumed in this case to correspond to a single metacell. For $N = 32$ and $p = 3$, the metacells are partitioned into three sets separated by dashed lines. The colors black, white, and grey are used to denote the metacells belonging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

Proof. The first part of the lemma is obvious since the N metacells, after getting rearranged in the two sorting steps, are striped across the p disks. Given an isovalue λ, the corresponding active metacells are those represented in the shaded region shown in Fig. 3 of the span space. We now examine how these metacells are distributed among the p processors. After the first sorting step, our partitioning algorithm distributes the N metacells evenly into $\sqrt{N/p}$ sets, each of which contributes $\sqrt{pN}$ metacells. The second sorting step is performed within each set separately, after which all the metacells are striped across the p disks. Each set whose $v_{\max}$ values are larger than or equal to λ (the first two sets in Fig. 3) contributes an equal number of active metacells with a difference of at most one for each set since the metacells within each set were sorted in increasing order of $v_{\min}$. Since there are at most $\sqrt{N/p}$ such sets, the total difference of active metacells contributed by these sets is at most $\sqrt{N/p}$. We may have an additional set some of whose $v_{\max}$ values are larger than or equal to λ, and the rest are strictly smaller (the last set in Fig. 3). In this case, the maximum difference in the number of active metacells among all the p processors cannot possibly exceed $\sqrt{N/p}$, which is the number of metacells placed at each processor from this set. Therefore the maximum difference in the number of active metacells among all processors is bounded by $2\sqrt{N/p}$, and the proof of the lemma is complete. □

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

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

5.2. *Extension to time-varying data*

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

6. Experimental setup

Our platform consists of a 16-node visualization cluster, each node consists of a two-way SMP Dual-CPU running at 3.0 GHz, an 8 GB main memory, a 60 GB local disk that can achieve 70 MB/s I/O transfer rate, and one NVIDIA6800 GPU card with bi-directional 4 GBps data transfer rate to memory via PCI-Express (×16) Bus. The GPU communicates with CPU and RAM via MCH (memory controller hub). These 16 nodes are interconnected through 10 Gbps Topspin InfiniBand network. In addition, four nodes are connected to four projectors for a four-way tiled wall-sized display via their GPU card's DVI port. The architecture of a single node is illustrated in Fig. 4.

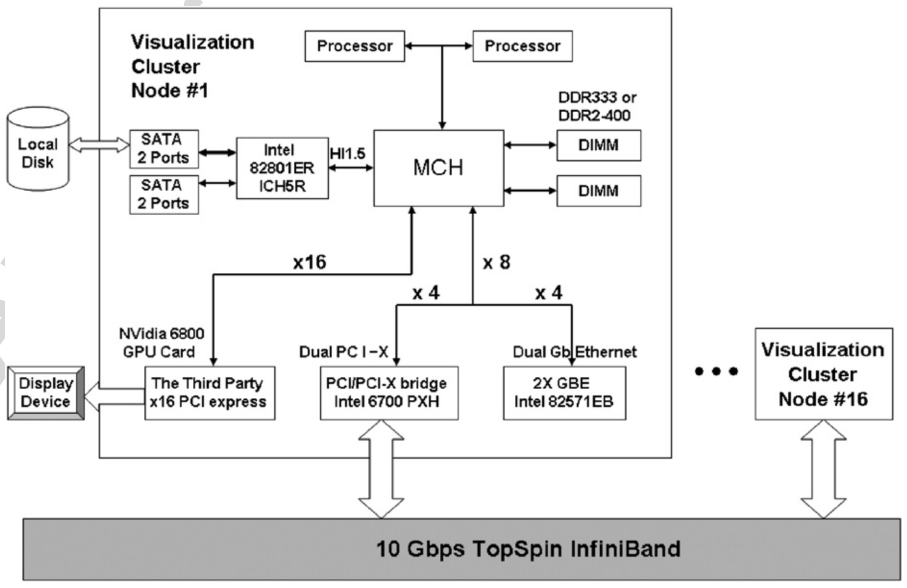

Fig. 4. The visualization cluster architectural diagram (Courtesy from Intel).

As a visualization cluster, each node of the system can run graphics programs and dispatch OpenGL commands to its GPU for rendering. The system software configuration includes Redhat Linux Enterprise 3.0, MPI, and the chromium package to enable the parallel rendering among multiple rendering nodes. Basically, the chromium is able to intercept OpenGL command calls from graphics applications and sends them to proper rendering servers according to the tiled-display layout[3] [12]. For the parallel rendering scheme, we use the *sort-last method* [16]. The essence of this method is to have each node render its triangles locally using the on-board GPU, after which the output is read back from the GPU's frame buffer and sorted according to the display server's tile layout. Different regions of the frame buffer including the z-buffer content are forwarded to the appropriate rendering servers, each of which will be responsible for displaying a specific region on the wall-sized display. At each rendering server, the components of the frame buffers from various processors are composited using their z-buffer contents and rendered to the display device connected to server's GPU. In our experiments, the time of sorting and shuffling the frame buffers among various nodes via 10 Gbps InfiniBand does not cause a noticeable overhead compared to time it takes to extract and render the triangles at each node as we will see later.

7. Experimental results

Since the performance of our algorithm depends critically on how the overall computational load is allocated to the different processors, we start by illustrating the load balancing achieved on a 16-processor cluster using seven different data sets, including the five data sets listed in Table 1. While the Richtmyer–Meshkov data use one-byte scalar fields, the MR-Brain and CTHead data sets consist of two-byte scalar fields, and the remaining four data sets (generated from a hurricane simulation) use floating point scalar fields. The average number of active metacells among all the processors as well as the maximum difference between the loads of any two processors are shown in Tables 2 and 3, over a wide range of isovalues. As the results show, the maximum difference of the number of active metacells among all the p processors is substantially smaller than the established upper bound of $2\sqrt{N/p}$, for all cases, and clearly illustrate the excellent load balancing achieved in practice by our scheme. Since the total work of our parallel scheme is about the same as the sequential algorithm, we expect the performance of our parallel algorithm to be linearly scalable as we show next in some detail for the Richtmyer–Meshkov data set.

7.1. Detailed performance results on the Richtmyer–Meshkov data set

We illustrate the performance of our parallel scheme on the Richtmyer–Meshkov data set using a wide range of isovalues. This data set consists of $2048 \times 2048 \times 1920$ one-byte scalar values for each time step and spans 270 time steps. The data amount to 7.5 GB for each time step for a total of 2.1 TB. Fig. 5 illustrates the isosurface generated for the isovalue 190 at time step 250 from a down-sampled version of the data set with $256 \times 256 \times 240$ one-byte scalar values. During the data preprocessing stage, we scan the data once and create the metacells, where each metacell consists of a four-byte ID indicating the location of the metacell, $9 \times 9 \times 9$ one-byte scalar values of the vertices, and the minimum values of the metacell vertices. At this point, the original data has been converted to $256 \times 256 \times 240$ metacells, each of size 734 bytes. Using our scheme, we stripe the metacells among the available disks and build each local indexing structure separately on each node. In particular, each node of the visualization cluster will hold a compact interval tree with pointers to the bricks stored on its local disk. For a single time step, this preprocessing takes about 30 minutes to complete on a single node of our cluster. We have done extensive testing of our algorithm using a wide range of isovalues as well as single and multiple nodes. A summary of our experimental results is given next.

7.1.1. Single time step case

We consider time step 250 of the Richtmyer–Meshkov data set and vary the isovalues from the value 10 up to the values 210, in steps of 20. For each of these isovalues, we ran the algorithm on one, two, four, eight, and sixteen nodes. We evaluate the performance of our isosurface extraction algorithm according to the following three metrics: (i) the I/O time it takes to retrieve the active metacells from the disk, referred to as Active MetaCell (AMC) retrieval time; (ii) the amount of CPU time required to go through the active metacells and generate the appropriate triangles, referred to as triangulation time; and (iii) finally the rendering time, which reflects the time it takes to render the triangles on the local GPU, after which the different frame buffers are composited to generate the final display. The time it takes to do the compositing of the very last step is included in the total time. The actual times obtained are summarized in Tables A1–A4 in Appendix A.

After preprocessing the data set for time step 250, we obtain 5, 592, 802 metacells that occupy a space of size 3.828 GB, which is nearly 50% smaller than the original 7.5 GB size since we eliminate the metacells for which all the vertices have the same scalar field value.

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

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

Table 2
Measured load imbalance in number of active metacells among sixteen nodes with varying isovalues for one-byte and two-byte data sets

LLNL data set			MRBrain data set			CTHead data set		
One-byte scalar $N = 5,629,653$; $p = 16$; $2(N/p)^{1/2} = 1186$			Two-byte scalar $N = 6,134,838$; $p = 16$; $2(N/p)^{1/2} = 1238$			Two-byte scalar $N = 5,799,589$; $p = 16$; $2(N/p)^{1/2} = 1204$		
Isovalue	Ave. $N^{\star}/p$	Max diff.	Isovalue	Ave. $N^{\star}/p$	Max diff.	Isovalue	Ave. $N^{\star}/p$	Max diff.
10	90,986	33	1253	24,657	108	0	12,196	36
30	129,505	28	1480	28,797	72	272	30,149	24
50	165,931	26	1707	36,185	61	544	18,255	42
70	177,806	18	1934	41,931	52	816	14,696	27
90	150,728	18	2161	29,164	38	1088	45,802	9
110	110,231	14	2388	17,132	44	1360	17,683	34
130	81,769	11	2615	7284	36	1632	15,536	27
150	64,839	5	2842	5425	42	1904	14,161	61
170	54,411	6	3069	3903	38	2176	12,258	22
190	47,232	7	3296	2052	18	2448	6055	44
210	41,658	12	3523	522	32	2720	1215	36

N denotes the number of intervals and p is the number of processors. Ave. $N^{\star}/p$ represents average number of active metacells per processor.

Table 3
Measured load imbalance in number of active metacells among sixteen nodes with varying isovalues for four-byte floating-point data sets corresponding to a hurricane simulation

Four-byte floating-point data sets: $N = 24,651,099$; $p = 16$; $2(N/p)^{1/2} = 2482$											
Pressure data set			Velocity data set			Vapor data set			Temp. data set		
Isovalue	Ave. $N^{\star}/p$	Max diff.	Isovalue	Ave. $N^{\star}/p$	Max diff.	Isovalue	Ave. $N^{\star}/p$	Max diff.	Isovalue	Ave. $N^{\star}/p$	Max diff.
−3272.59	83	9	−60.44	65	9	1.81	23,794	32	−68.85	31,832	51
−2763.30	214	16	−49.66	410	20	3.61	23,036	27	−59.81	17,375	17
−2254.02	410	30	−38.88	1315	38	5.41	22,190	31	−50.77	16,753	51
−1744.74	633	32	−28.10	2602	53	7.22	20,038	25	−41.73	16,508	54
−1235.46	1020	12	−17.32	6135	75	9.02	16,362	50	−32.69	16,726	39
−726.18	1678	34	−6.55	46,495	55	10.83	15,186	14	−23.65	16,810	55
−216.90	3095	39	4.23	49,948	70	12.63	14,103	71	−14.61	16,914	49
292.38	22,879	32	15.01	15,791	41	14.44	11,712	54	−5.56	17,100	50
801.66	19,730	65	25.79	5308	27	16.24	8861	27	3.48	21,832	75
1310.94	15,003	49	36.57	1087	39	18.05	5647	39	12.52	23,130	51
1820.22	9334	45	47.35	198	35	19.85	1224	29	21.56	17,562	68

N denotes the number of intervals and p is the number of processors. Ave. $N^{\star}/p$ represents average number of active metacells per processor.

Tables A1–A4 (in Appendix A) show the execution times of the major steps of our algorithm on two, four, eight, and sixteen processors over a wide range of isovalues. A careful examination of the experimental results in these tables clearly illustrate the scalability and the efficiency of our scheme. Note also that the compositing step (whose time is equal to the total time minus the sum of the times of the other steps) is extremely fast and takes at most a few hundreds of milliseconds even on sixteen processors.

The overall time spent on the extraction and rendering of isosurfaces for various isovalues is illustrated in Fig. 6. The corresponding speedups are highlighted in Fig. 7. As expected, our scheme achieves very good scalability relative to our extremely efficient serial algorithm, independent of the particular isovalue.

7.2. Time-varying case

We now consider the more general case of time-varying data sets that are to be explored by extracting and rendering isosurfaces corresponding to a time step and an isovalue. We can index the 270 time steps of the Richtmyer–Meshkov data set using our indexing scheme. The size of the resulting indexing structure is 3.2 MB, which easily fits into the main memory of a node. The layout of the data of each time step will be distributed across the processors as before. Extracting an iso-

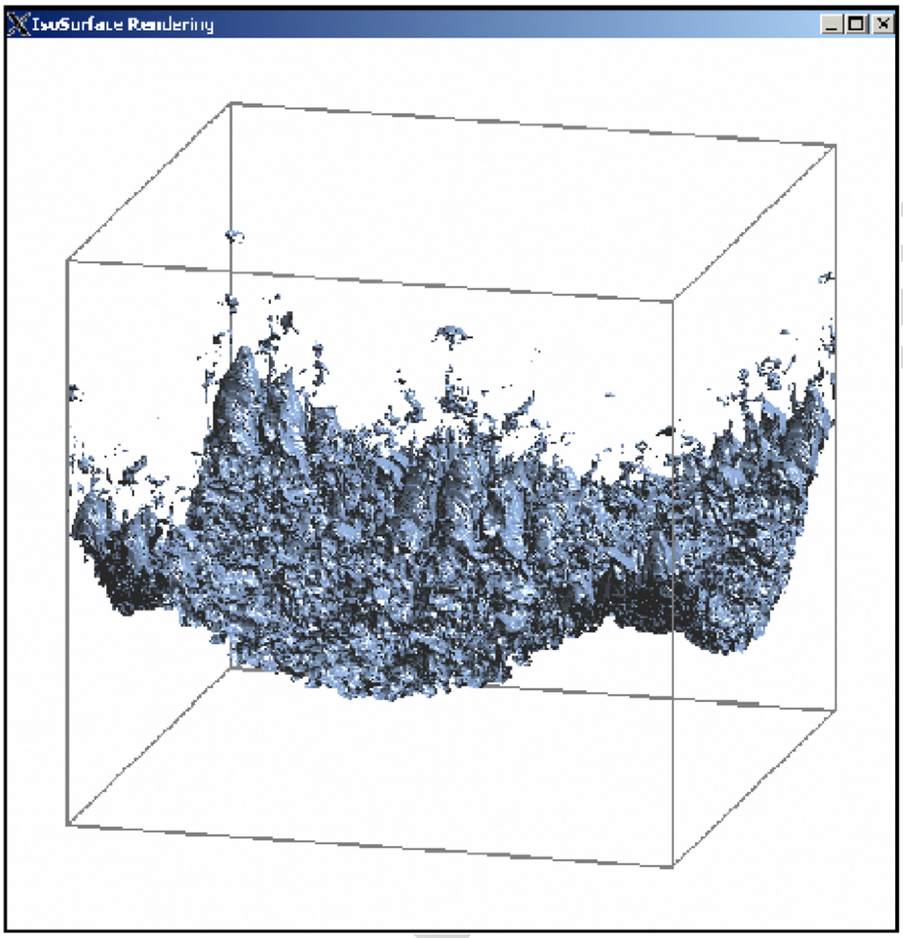

Fig. 5. Isosurface corresponding to the isovalue 190 at time step 250 from a down-sampled version of size $256 \times 256 \times 240$ of the Richtmyer–Meshkov data set.

Table 4
Performance summary of our algorithm

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

AMC refers to the active metacells. The last column shows that the performance is linear in the number of triangles and hence the algorithm is output sensitive.

surface of a time step amounts to determining the appropriate indexing structure for that time step, which can easily be performed since the whole indexing structure is in main memory. Table 5 shows the results for time steps 180 ∼ 195 for the isovalue of 70. Each row of the table lists the number of active metacells, the number of triangles generated, the execution time on a four-node configuration, and the overall rate of triangles rendered (millions per second).

8. Conclusion

In this paper, we have presented a new indexing structure and a new partitioning algorithm for out-of-core isosurface extraction and rendering of large scale data. The indexing scheme is based on a compact version of the interval tree that makes use of the span space concept whose size is $O(n \log n)$ compared to $\Omega(N)$ for the standard interval tree, where N is the number of

Table 5
Overall performance of four processors on isovalue 70 over 16 time steps

Time step	Number of active metacells	Number of triangles	Overall time (s)	Overall rate ($\times 10^6$/s)
180	2,249,247	499,936,480	35.971	13.898
181	2,259,741	502,356,768	32.934	15.253
182	2,269,996	504,717,568	32.717	15.427
183	2,280,249	507,140,192	33.349	15.207
184	2,290,438	509,504,102	33.145	15.372
185	2,300,808	511,877,068	33.628	15.222
186	2,310,642	514,222,240	34.121	15.071
187	2,320,869	516,675,168	34.524	14.966
188	2,330,322	519,026,094	33.732	15.387
189	2,340,363	521,496,928	34.300	15.204
190	2,295,699	516,444,992	31.405	16.445
191	2,360,385	526,339,520	35.163	14.969
192	2,370,458	528,728,448	34.523	15.315
193	2,380,148	531,149,920	35.256	15.066
194	2,389,433	533,600,192	35.172	15.171
195	2,399,116	536,050,312	35.891	14.936

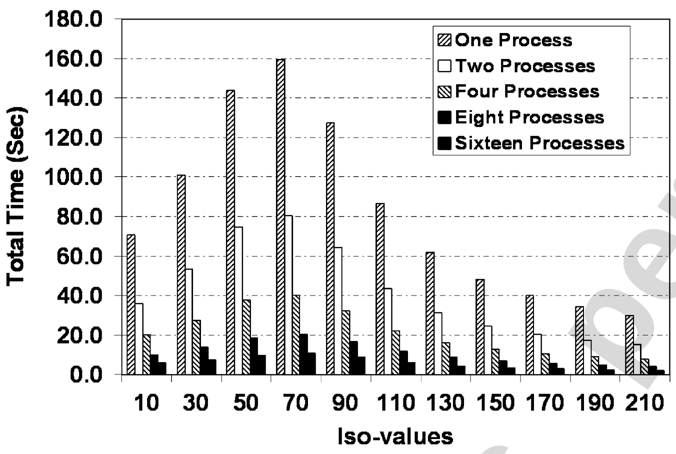

Fig. 6. Overall time of up to 16 processors over a range of isovalues.

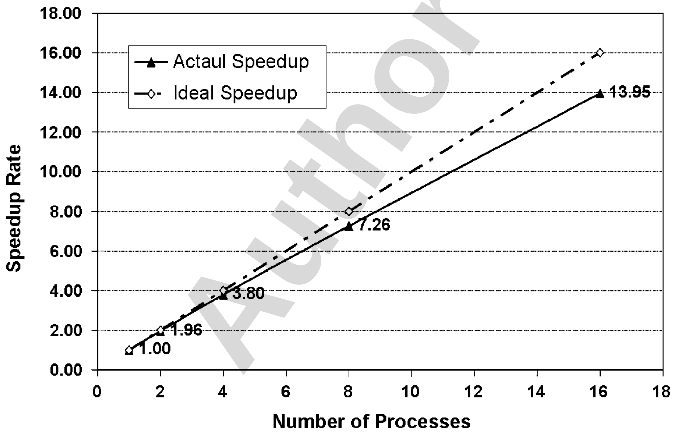

Fig. 7. Corresponding speedups of up to 16 processors over a range of isovalues.

all possible pairs of scalar field values appearing in metacells and n is the number of their distinct endpoints. The data are arranged in a compact layout on the disk, which enables very efficient I/O performance. We have shown that our new algorithm can easily be adapted to a multiprocessor environment, provably delivering efficient and scalable performance. The algorithm was tested extensively on a wide variety of data sets, and was shown in detail to achieve scalable performance on the Richtmyer–Meshkov data set over different processor configurations and different isovalues. However, there is an issue that needs further exploration, namely the compositing performed at the last step. As the number of processors increases, this step will require more communication time, which was in the order of hundreds of milliseconds in our experiments on the 16-node visualization cluster. However, for much larger clusters, we need to organize the compositing step more efficiently so that it does not become a bottleneck. We are currently in the process of developing and testing such a strategy in which images are merged in a multi-way hierarchical process. The details will be reported in a forthcoming paper.

Acknowledgments

We would like to thank Fritz McCall and Brad Erdman for setting up the visualization cluster, and Derek Juba for his help in configuring various software packages. We also would like to acknowledge Mark Duchaineau at the Lawrence Livermore National Lab for making the Richtmyer–Meshkov instability data set available to us and for guiding us through the initial stages of using it. This work was supported by the NSF research infrastructure Grant CNS-04-03313. A preliminary version of this paper appeared in the Proceedings of IPDPS'06 held in April 2006.

Appendix A. Performance for a varying number of processors

Execution time with two, four, eight and sixteen processors over varying isovalues are summarized in Tables A1–A4.

Table A1
Execution time with two processors over varying isovalues

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

Table A2
Execution time with four processors over varying isovalues

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

Table A3
Execution time with eight processors over varying isovalues

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

Table A4
Execution time with sixteen processors over varying isovalues

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

References

[1] A. Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Comm. ACM 31 (9) (1988) 1116–1127.
[2] L. Arge, J.S. Vitter, Optimal dynamic interval management in external memory (extended abstract), in: IEEE Symposium on Foundations of Computer Science, 1996, pp. 560–569.
[3] C.L. Bajaj, V. Pascucci, D.R. Schikore, Fast isocontouring for improved interactivity, in: 1996 Volume Visualization Symposium, October 1996, pp. 39–46.
[4] C.L. Bajaj, V. Pascucci, D. Thompson, X. Zhang, Parallel accelerated isocontouring for out-of-core visualization, in: Proceedings of 1999 IEEE Parallel Visualization and Graphics Symposium, 1999, pp. 97–104.
[5] Y. Chiang, C. Silva, External memory techniques for isosurface extraction in scientific visualization, External Memory Algorithms and Visualization, DIMACS Book Series, vol. 50, American Mathematical Society, Providence, RI, 1999, pp. 247–277.
[6] Y.-J. Chiang, R. Farias, C. Silva, B. Wei, A unified infrastructure for parallel out-of-core isosurface and volume rendering of unstructured grids, in: Proceedings of the IEEE Symposium on Parallel and Large-data Visualization and Graphics, 2001, pp. 59–66.
[7] Y.-J. Chiang, C.T. Silva, I/O optimal isosurface extraction, in: Proceedings of IEEE Visualization, 1997, pp. 293–300.
[8] Y.-J. Chiang, C.T. Silva, W.J. Schroeder, Interactive out-of-core isosurface extraction, in: Proceedings of IEEE Visualization, 1998, pp. 167–174.
[9] P. Cignoni, C. Montani, D. Darti, R. Scopigno, Optimal isosurface extraction from irregular volume data, in: Proceedings of the 1996 Symposium on Volume Visualization, San Francisco, USA, 1996, pp. 31–38.
[10] P. Ellsiepen, Parallel isosurfacing in large unstructured data sets, in: Visualization in Scientific Computing '95, Springer, Berlin, 1995, pp. 9–23.
[11] C. Hansen, P. Hinker, Massively parallel isosurface extraction, in: Proceedings of IEEE Visualization, 1992, pp. 77–83.
[12] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner, J. Klosowski, Chromium: a stream processing framework for interactive rendering on clusters, in: Proceedings of SIGGRAPH 2002, 2002, pp. 693–702.
[13] T. Itoh, K. Koyamada, Automatic isosurface propagation using an extreme graph and sorted boundary cell lists, IEEE Trans. Visualization Comput. Graphics 1 (4) (1995) 319–327.
[14] W.E. Lorensen, H.E. Cline, Marching cubes: a high resolution 3D surface construction algorithm, in: M.C. Stone (Ed.), Computer Graphics (SIGGRAPH '87 Proceedings), vol. 21, July 1987, pp. 161–169.
[15] S. Miguet, J.M. Nico, A load-balanced parallel implementation of marching-cubes algorithm, in: Proceedings of High Performance Computing Symposium '95, 1995, pp. 229–239.
[16] S. Molnar, M. Cox, D. Ellsworth, H. Fuchs, A sorting classification of parallel rendering, IEEE Comput. Graphics Appl. 14 (4) (1994) 23–32.
[17] T.S. Newman, N. Tang, Approaches that exploit vector-parallelism for three rendering and volume visualization techniques, Comput. Graphics 24 (5) (2000) 755–774.
[18] S. Parker, P. Shirley, Y. Livnat, C. Hansen, P.-P. Sloan, Interactive ray tracing for isosurface rendering, in: IEEE Visualization '98, October 1998, pp. 233–238.
[19] H.W. Shen, C.D. Hansen, Y. Livnat, C.R. Johnson, Isosurfacing in span space with utmost efficiency (ISSUE), in: IEEE Visualization'96, October 1996, pp. 281–294.
[20] C. Silva, Y. Chiang, J. El-Sana, P. Lindstrom, Out-of-core algorithms for scientific visualization and computer graphics, in: Visualization'02, Course Notes for Tutorial #4, 2002.
[21] P.M. Sutton, C.D. Hansen, Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON), in: IEEE Visualization '99, IEEE Computer Society Press, Silver Spring, MD, October 25–29 1999, pp. 147–154.
[22] J. Wilhelms, A. Van Gelder, Octrees for faster isosurface generation, Computer Graphics (San Diego Workshop on Volume Visualization), vol. 24, 1990, pp. 57–62.
[23] X. Zhang, C.L. Bajaj, W. Blanke, Scalable isosurface visualization of massive data sets on cots clusters, in: Proceedings of IEEE Symposium on Parallel and Large-data Visualization and Graphics, 2001, pp. 51–58.
[24] X. Zhang, C.L. Bajaj, V. Ramachandran, Parallel and out-of-core view-dependent isocontour visualization using random data distribution, in: Proceedings of Joint Eurographics—IEEE TCVG Symposium on Visualization and Graphics, 2002, pp. 9–18.
[25] H. Zhang, T.S. Newman, Efficient parallel out-of-core isosurface extraction, in: Proceedings of IEEE Symposium on Parallel and Large-data Visualization and Graphics (PVG) '03, October 2003, pp. 9–16.

Qin Wang is currently a 5th-year graduate student of the Department of Electrical and Computer Engineering in the University of Maryland at College Park. His Ph.D. research is in the area of Computer Engineering with focus on interactive visualization, large-scale data exploration and high-performance computing. Under Professor Joseph JaJa's advising, he has advanced to candidacy with papers published in the conferences of IPDPS and SSDBM, and expects to graduate by 2007. Qin Wang received a M.S. degree in Computer Science and the Award of outstanding graduate student from Georgia State University in 2002.

Joseph JaJa currently holds the position of Professor of Electrical and Computer Engineering and the Institute for Advanced Computer Studies at the University of Maryland, College Park. Dr. JaJa received his Ph.D. degree in Applied Mathematics from Harvard University. His current research interests are in parallel algorithms, digital preservation, and scientific visualization of large scale data. Dr. JaJa has received numerous awards including the IEEE Fellow Award in 1996, the 1997 R&D Award for the development software for tuning parallel programs, and the ACM Fellow Award in 2000. He served on several editorial boards, and is currently serving as a subject area editor for the Journal of Parallel and Distributed Computing and as an editor for the International Journal of Foundations of Computer Science.

Amitabh Varshney is a Professor of Computer Science at the University of Maryland. Varshney's research has addressed challenges in 3D interactive graphics and visualization for large graphics data sets by reconciling realism with interactivity through multiresolution techniques and high-performance computing. Varshney received the NSF CAREER award in 1995 and the first IEEE Visualization Technical Achievement Award in 2004.Varshney received a B.Tech. in Computer Science from the Indian Institute of Technology Delhi in 1989 and a M.S. and Ph.D. in Computer Science from the University of North Carolina at Chapel Hill in 1991 and 1994. Further details are available at http://www.cs.umd.edu/~varshney.