An Efficient and Scalable Parallel Algorithm for Out-of-Core Isosurface Extraction and Rendering

Qin Wang[1], Joseph JaJa[1], Amitabh Varshney[2]

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

[2]University of Maryland
Institute for Advanced Computer Studies
Dept. of Computer Science
College Park, MD 20742 USA
varshney@cs.umd.edu

Abstract

We consider the problem of isosurface extraction and rendering for large scale time varying data. Such datasets 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. We develop a new simple indexing scheme, which makes use of the concepts of the interval tree and the span space data structures. The new scheme enables isosurface extraction and rendering in I/O optimal time, using more compact indexing structure and more effective bulk data movement than the previous schemes. Moreover, our indexing scheme can be easily extended to a multiprocessor environment in which each processor has access to its own local disk. The resulting parallel algorithm is provably efficient and scalable. That is, it 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 dataset, and obtain results that consistently validate the efficiency and the scalability of our algorithm.

1. Introduction

During the past few years, we have seen an increasing trend towards the generation and analysis of very large time-varying datasets in scientific simulation. Such datasets 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 dataset that will be referred to extensively in this paper, 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 [1]. This dataset represents a simulation in which two gases, initially separated by a membrane, are pushed against a wire mesh. These are then perturbed with a superposition of long wavelength and short wavelength disturbances and a strong shock wave. This simulation took 9 days on 960 CPUs and produced about 2.1 terabytes of simulation data. The data shows the characteristic development of bubbles and spikes and their subsequent merger and break-up over 270 time steps. Each time step is 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 allows 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 datasets, consider the issues of data representation, and visualization in a fragmented manner, and do not scale to the terabyte-sized datasets. Visual interaction with large databases of dynamic simulation datasets requires the development of a high-performance graphics software infrastructure running on powerful 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 ren-

dering of time-varying scalar fields. Compared with other published algorithms, our approach has the following advantages:

- Our serial algorithm achieves the same asymptotic bounds as the optimal algorithms based on the external memory version of the interval tree, but with a much smaller indexing structure, a more effective bulk data movement, and without incurring the significant overhead of the external interval tree.

- Our scheme can be implemented on a 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. As a result, our parallel algorithm is linearly scalable with optimal I/O complexity and no communication is required except for the final phase of compositing the frame buffers from the different nodes to generate the final display. 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 $3.5 \sim 4.0M$ triangles per second on the Richtmyer-Meshkov dataset using our algorithm on a single processor. On an 8-node cluster, we achieve scalable performance across widely different isovalues with speed-ups up to 7.83 relative to the serial algorithm.

We make use of the University of Maryland visualization cluster in which each node consists of a 2-way symmetric multiprocessor with 8GB of main memory, an NVIDIA GPU (Graphics Processing Unit), coupled with a 60GB of local disk. The nodes are interconnected via a 10 Gbps InfiniBand, with four nodes reserved for compositing the buffer outputs of other processors and displaying the results on a tiled multi-projector wall-sized screen.

2. Previous Work

Many improvements to the initial Marching Cubes algorithm [2] have been reported in the literature. Some algorithms attempt to reduce the number of cells examined by using spatial data structures such as octree [3, 4], while others determine a collection of "seed cells" and perform contour propagation from these cells [5, 6], or partition the span space [7]. A theoretically optimal algorithm was described in [8], and involves building an interval tree that enables the exploration of only the active cells (cells that intersect the isosurface). This algorithm was later generalized to a theoretically optimal out-of-core isosurface extraction in [9, 10]. As for parallel algorithms for the case when the data fits within the main memories of the different processors, several algorithms have been reported in [7, 11, 12, 13, 14, 15]. Of more interest to us, are the out-of-core parallel algorithms such as those reported in [16, 17, 20, 22, 23]. We proceed to briefly discuss some of the most recent algorithms and relate them to the work described in this paper.

A parallel out-of-core algorithm has to deal with (i) data indexing and layout among the parallel disks available through the parallel system; (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 (preprocessing step); the relative computational loads of the different processors corresponding to an arbitrary isovalue; and the performance of the rendering and rasterization into a single display. The preprocessing step described in [10, 17, 18] involves partitioning the dataset into *metacells*, where each metacell is a cluster of neighboring cells and occupies about the same number of disk blocks, and building a B-tree like interval tree, called Binary-Blocked I/O interval tree (BBIO tree). The computational cost of this step is similar to an external sort, which is not insignificant. 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. A significant bottleneck with this scheme is the host overhead in coordinating and dispatching jobs, and the access pattern to the available disks is quite unpredictable. The algorithm described in [21] 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. It is easy to see that one can have a case in which the distribution of active cells among the processors for a given isovalue could be extremely unbalanced. The query processing described in that paper is somewhat similar to ours but we provably achieve load balancing for any isovalue using an indexing scheme that is in general more efficient than the external interval tree. The extracted local surface is streamed to parallel rendering servers,

followed by compositing the outputs of the different frame buffers to a tiled-display. The preprocessing algorithm described in [22] 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 [22], the preprocessing is computationally non-trivial and load-balancing is not guaranteed in general.

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 4KB or 8KB. We will use the standard model [24] to measure the I/O performance of our algorithms. We denote the input size by N, the disk block size by B, and the size of the main memory by M. In this work, we are assuming that N is much larger than M, which is in turn much larger than B. The 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.

An I/O optimal algorithm for extracting isosurfaces has been reported in [9, 10]. Its I/O complexity is $O(log_B(N/B) + T/B)$, where T is the total size of the cells satisfying the query relative to the given isovalue. The first term of the I/O complexity reflects the number of I/O accesses required to traverse the external version of the interval tree corresponding to the input data, and the second term captures the minimum number of I/O accesses required to read the active cells. We will next describe our indexing structure, called the *compact interval tree*, which is in general much smaller than the standard interval tree and can also be used to achieve asymptotically optimal I/O complexity.

4. Compact Interval Tree Indexing Scheme

Our algorithm can handle both structured and unstructured grids and makes use of the metacell notion introduced in [10]. 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 regular grid of the Richtmyer-Meshkov dataset, 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 metacells. 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 data structure to compact the data layout. Before introducing this structure, we begin with a brief review of the standard binary interval tree.

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

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

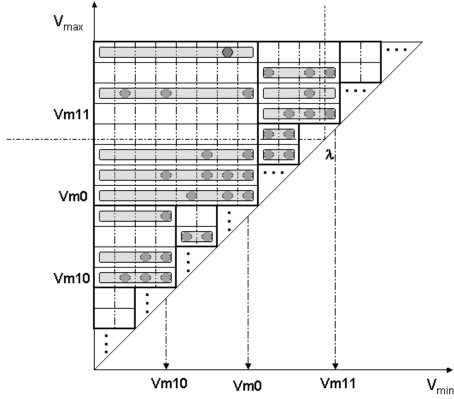

Figure 1. Span Space Partitioning Scheme for Our Indexing Structure.

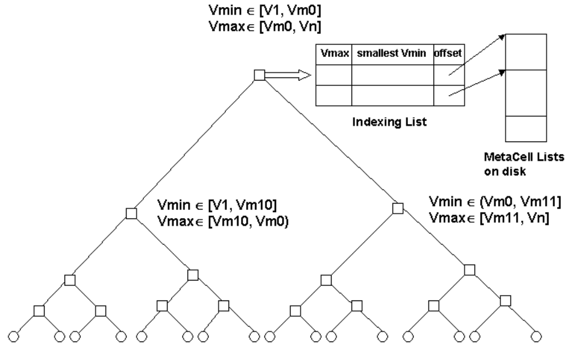

Figure 2. Binary Tree Structure and the Associated Metacell lists.

in increasing order of their $v_{\min}$ values. We will 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, 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 at least twice the size of our indexing structure, and is usually much larger. 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 $log_2 n$. 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 $\Omega(n^2)$. Note that the size of the preprocessed dataset will not increase relative to the original data size. In fact, it may be smaller as we do not include any of the metacells whose maximum and minimum scalar field values are equal. As we demonstrate later, this is indeed the case for the Richtmyer-Meshkov dataset, which results in a processed data set almost half the size of the original data set.

5. Efficient and Scalable Isosurface Extraction Algorithm

Let's first consider the case when the compact interval tree fits in main memory, i.e., $M = O(n \log n)$. Given a query isovalue λ, consider the unique path from the leaf node labeled with the largest value $\leq \lambda$ to the root. Each internal node on this path contains an index list with pointers to some bricks. For each such node, two cases can happen depending on whether 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 $v_{\min} \leq \lambda$, from each of the bricks on the index list of the node. These metacells can be retrieved from the disk starting from the first metacell on each brick until a metacell is encountered with a $v_{\min} > \lambda$. Note that since each entry of the index list contains the $v_{\min}$ of the corresponding brick, no I/O access will be performed if the brick contains no active metacells.

It is clear that the performance of our algorithm is I/O optimal under the assumption that the compact interval tree fits in main memory. This assumption is very likely to hold given the small size of our indexing structure and the ever increasing sizes of main memories. Note that for one, two, or three-byte scalar fields, the compact interval tree will fit in the main memory of any of today's processors regardless of the size of the collection and the variations in the scalar field values. This is not the case for the standard interval tree, even for two-byte scalar fields. In Table 1 below we compare the sizes of the two indexing structures for some well-known data sets, including the datasets Bunny, MRBrain, and CTHead from the Stanford Volume Data Archive [25]. As can be seen from the table, our indexing structure is substantially smaller than the standard interval tree, even in the case of $N \approx n$ such as Pressure and Velocity data sets.

In the unlikely case when the compact interval tree does not fit in main memory, we use the same strategy as in [10] and group each B nodes of the binary tree into one disk block thereby reducing the height of the tree to $O(\log_B n)$, which is then stored on the disk. As in [10], we can then retrieve the active metacells in asymptotically optimal I/O time but using a significantly more complex algorithm.

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

DataSet Name	Scalar Field Size	N	n	Size of Interval Tree	
				Standard	Compact
Bunny	2 bytes	2,502103	5,469	38.2MB	42.7KB
MRBrain	2 bytes	756,982	2,894	11.6MB	22.6KB
CTHead	2 bytes	817,642	3,238	12.5MB	25.3KB
Pressure	4 bytes	24,507,104	20,748,433	374.0MB	158.3 MB
Velocity	4 bytes	27,723,399	22,108,501	423.0MB	168.7MB

Table 1. Size Comparison between Standard and Compact Interval Trees. N denotes the number of distinct intervals and n represents the number of distinct v_{max} values. The dataset Bunny, MRBrain, CTHead are 3D CT images that were obtained from Stanford Volume Data Archive[25]. Those of Pressure and Velocity are simulation data of a Hurricane[26] and an earthquake[27] respectively.

5.1. Parallel Processing

Our indexing scheme can be easily adapted to a multiprocessor environment in which each node has access to its own local disk. Assume that we have p processors, each with its own local disk, and the processors are interconnected with a high-speed interconnection network. We now show how to distribute the metacells among the local disks in such a way that the active metacells corresponding to any isovalue are spread evenly among the processors. For each index in our structure, we stripe the metacells stored in a brick across the p disks, that is, the first metacell on each list is stored on the disk of the first processor, the second on the disk on the second processor, and so on wrapping around as necessary. For each processor, the indexing structure will be constructed as before except that each entry contains the $v_{\min}$ of the metacells in the corresponding local brick and a pointer to the local brick.

It is clear that, for any given isovalue, the active metacells are spread almost evenly among the local disks of the p processors. The isosurface query can be carried out simultaneously by all the p processors using their own local index lists. As a result roughly the same number of triangles is generated by each processor, which are then rendered locally. The p frame buffers will then be merged using their depth information to create the final output. Except for the very last step, we have provably split the work equally among the processors, without increasing the total work relative to the sequential algorithm. For large scale datasets such as the Richtmyer-Meshkov dataset, the last 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

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 dataset. For example, one-byte scalar data with hundreds of time steps will require an indexing structure of size at most 1MB, which can easily fit in the main memory of any of today's processors. Similarly for two-byte scalar data, the size of the indexing structure increases to hundreds of Megabytes, which is still reasonable and can easily fit in today's processors' main memory. In the case of Richtmyer-Meshkov data set, we have 270 time steps with 7.5GB per time step, which amounts to a total of about 2.1TB. However the total size of our indexing structure is only 1.6MB.

6. Experimental Setup

Our platform consists of a 16-Node visualization cluster, each node consists of a 2-way SMP Dual-CPU running at 3.0 GHz, an 8GB main memory, a 60GB local disk that can achieve 50 MB/sec I/O transfer rate, and one NVIDIA FX6800Ultra GPU card with bi-directional 4Gbps data transfer rate to memory via PCI-Express ($\times 16$) Bus. The GPU communicates with CPU and RAM via MCH(Memory Controller Hub). These 16 Nodes are inter-connected 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 Figure 3. 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. Chromium intercepts OpenGL command calls from the processors and sends them to proper rendering servers according to the tiled-display layout [28, 29]. For parallel rendering, we use the *sort-last method* [30]. 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 doesn't cause a noticeable overhead compared to time it takes to extract and render the triangles at each node.

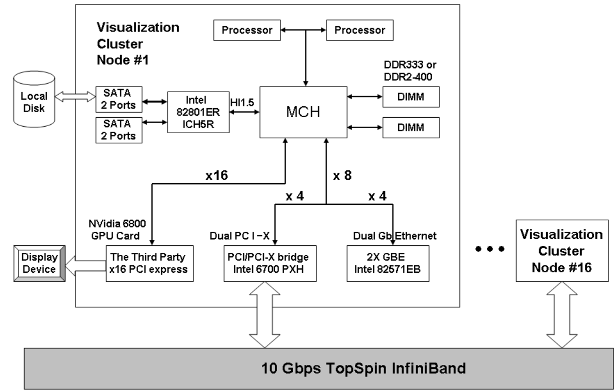

Figure 3. The Visualization Cluster Architectural diagram. (Courtesy from Intel)

7. Experimental Results

We have used the Richtmyer-Meshkov dataset to test our isosurface extraction algorithm based on the indexing scheme described earlier. This dataset consists of $2048 \times 2048 \times 1920$ one-byte scalar values for each time step and spans 270 time steps. The data amounts to 7.5GB for each time step for a total of 2.1TB. Figure 4 illustrates the isosurface generated for the isovalue 190 at time step 250 from a down-sampled version of the dataset 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 4-byte ID indicating the location of the metacell, $9 \times 9 \times 9$ one-byte scalar values of the vertices, and the minimum value of the metacell vertices. At this point, the original data has been converted to $256 \times 256 \times 240$ metacells, each of length 734 bytes. We remove all the metacells for which all the vertices have the same scalar value. For our dataset, this results in a dramatic saving of approximately 50% of disk space. Using our indexing scheme, we can create the indexing lists and corresponding bricks and stripe the corresponding metacells among the disks of the various processors as explained before. Each node of the visualization cluster holds an indexing structure 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.

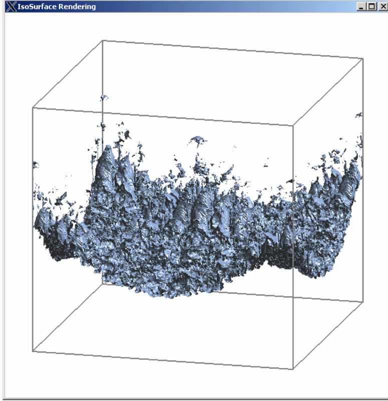

Figure 4. Isosurface corresponding to the isovalue 190 at time step 250 from a down-sampled version of size $256 \times 256 \times 240$ of the Richtmyer-Meshkov dataset.

Isovalue	Number of Triangles	Number of AMC	AMC Retrieve (sec)	Retrieve Rate (MB/sec)	Triangu lation (sec)	Render-ing (sec)	Total (sec)	Rate ($\times10^6$/s)
10	228,770,844	1,455,782	22.53	45.30	36.20	11.96	70.74	3.23
30	376,578,332	2,072,085	29.48	49.27	57.90	13.54	100.98	3.73
50	569,387,336	2,654,902	37.83	49.19	85.41	20.56	143.85	3.96
70	651,834,482	2,844,889	39.74	50.17	96.31	23.54	159.65	4.08
90	511,136,810	2,411,647	32.80	51.55	76.09	18.42	127.35	4.01
110	329,408,766	1,763,701	24.20	51.08	50.45	12.02	86.72	3.80
130	229,201,420	1,308,299	18.22	50.32	35.25	8.36	61.89	3.70
150	177,035,314	1,037,426	14.52	50.07	27.18	6.48	48.23	3.67
170	146,369,438	870,573	12.27	49.75	22.38	5.36	40.05	3.66
190	125,365,482	755,710	10.68	49.60	19.09	4.58	34.41	3.64
210	108,977,638	666,527	9.41	49.65	16.50	3.98	29.94	3.64

Table 2. Execution time on a single processor over varying Isovalues

Isovalue	AMC Retrieve (sec)	Triangu-lation (sec)	Render-ing (sec)	Total (sec)	Overall Rate ($\times10^6$/s)	Speedup
10	10.661	17.963	5.998	36.005	6.320	1.96
30	14.822	27.982	6.533	53.31	7.046	1.89
50	18.716	41.445	9.988	74.827	7.589	1.92
70	19.218	48.485	11.918	80.537	8.064	1.98
90	15.702	39.066	9.599	64.42	7.891	1.98
110	11.589	25.695	6.225	43.568	7.521	1.99
130	8.972	18.036	4.346	31.412	7.297	1.97
150	7.364	13.879	3.342	24.639	7.185	1.96
170	6.135	11.383	2.754	20.33	7.200	1.97
190	5.321	9.722	2.379	17.479	7.172	1.97
210	4.63	8.407	2.069	15.162	7.188	1.97

Table 3. Execution time with two processors over varying Isovalues

7.1. Single Time Step Case

We tested our scheme on a single time step of the Richtmyer-Meshkov dataset using isovalues ranging from 10 up to 210, in steps of 20. For each of these cases, we ran the algorithm on one, two, four and eight 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 actual times obtained are summarized in Tables 2 through 7.

After preprocessing the dataset for time step 250, we obtain $5,592,802$ metacells that occupy a space of size 3.828GB, which is nearly 50% smaller than the original 7.5GB size.

We first consider the performance of our algorithm on a single processor. From Table 2, we can see that the number of generated triangles varies from 100 million to 650 million over the range of isovalues from 10 to 210. Our indexing structure is of size 6KB, which is quite small compared to the size of the data. As shown in Table 2, we are able to achieve the I/O rate of about $50MB/s$ in retrieving the active metacells, with a linear relationship between the I/O time and the number of triangles generated. It is clear that the triangle generation stage is the bottleneck for the whole isosurface extraction as we need to go through each of the 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. Table 3 shows the performance when the algorithm runs on two nodes, while Tables 4 and 5 report the performance of each node on a four-node and eight-node combination for a wide range of isovalues respectively. Notice that the

Isovalue	AMC Retrieve (sec)	Triangu-lation (sec)	Render-ing (sec)	Total (sec)	Overall Rate ($x10^6$/s)	Speedup
10	6.343	8.907	2.953	19.979	11.450	3.54
30	8.137	13.596	3.121	27.604	13.642	3.66
50	9.799	20.161	4.782	37.810	15.059	3.80
70	9.698	23.986	5.84	40.172	16.226	3.97
90	7.887	19.601	4.784	32.331	15.810	3.94
110	5.978	13.055	3.139	22.230	14.818	3.90
130	4.806	9.152	2.196	16.213	14.137	3.82
150	3.938	7.042	1.693	12.730	13.907	3.79
170	3.289	5.803	1.400	10.549	13.875	3.80
190	2.894	4.955	1.203	9.110	13.761	3.78
210	2.415	4.276	1.056	7.806	13.961	3.84

Table 4. Execution time with four processors over varying Isovalues

Isovalue	AMC Retrieve (sec)	Triangu-lation (sec)	Render-ing (sec)	Total (sec)	Overall Rate ($x10^6$/s)	Speedup
10	3.541	4.462	1.460	9.972	23.055	7.09
30	4.246	6.776	1.633	13.989	27.012	7.22
50	4.752	10.023	2.449	18.558	30.761	7.75
70	4.630	12.119	3.042	20.387	32.049	7.83
90	3.812	10.045	2.542	16.648	30.784	7.65
110	3.056	6.775	1.688	11.867	27.860	7.31
130	2.624	4.800	1.184	8.771	26.256	7.06
150	2.193	3.714	0.922	6.947	25.629	6.94
170	1.840	3.071	0.774	5.797	25.416	6.91
190	1.527	2.616	0.671	4.928	25.630	6.98
210	1.269	2.262	0.578	4.220	26.041	7.09

Table 5. Execution time with eight processors over varying Isovalues

overall speedup ranges from 3.54 to 3.97 on four nodes, which is very close to the ideal linear speedup relative to our extremely efficient algorithm on a single node. Similarly the speedups on eight nodes range between 6.91 to 7.83.

To shed more light on the fact that we achieve a very good load balancing across the various nodes, the distributions of active metacells and the generated triangles across four nodes are shown respectively in Tables 6 and 7 for a wide range of isovalues. Both tables show that our scheme achieves a very good load balancing irrespective of the isovalue. The overall time spent on the extraction and rendering of isosurfaces for various isovalues is shown in Figure 5. The corresponding speedups are illustrated in Figure 6. As expected, our scheme achieves very good scalability relative to our extremely efficient serial algorithm, independent of the particular isovalue.

Isovalue	Node#0	Node#1	Node#2	Node#3
10	364,034	363,971	363,913	363,864
30	518,103	518,046	517,989	517,947
50	663,801	663,751	663,702	663,648
70	711,306	711,246	711,193	711,144
90	602,976	602,932	602,892	602,847
110	440,985	440,940	440,903	440,873
130	327,135	327,086	327,056	327,022
150	259,409	259,370	259,340	259,307
170	217,688	217,658	217,628	217,599
190	188,961	188,940	188,914	188,895
210	166,651	166,638	166,625	166,613

Table 6. Number of Active Meta Cells distributed over four nodes with varying Isovalues for Time-step # 250.

Isovalue	Node#0	Node#1	Node#2	Node#3
10	54,710,153	57,096,353	57,706,224	59,258,114
30	85,428,280	92,182,182	96,993,875	101,973,995
50	130,897,616	140,963,566	147,232,803	150,293,351
70	159,678,730	164,677,161	164,256,382	163,222,209
90	130,548,647	131,639,566	128,300,631	120,647,966
110	85,756,960	84,332,674	81,800,758	77,518,374
130	59,975,566	57,924,960	56,448,911	54,851,983
150	46,171,209	44,331,946	43,497,135	43,035,024
170	38,164,588	36,515,593	35,954,618	35,734,639
190	32,829,507	31,273,891	30,731,499	30,530,585
210	28,584,104	27,176,976	26,825,726	26,390,832

Table 7. Number of generated Triangles distributed over four nodes with varying Isovalues for Time-step # 250.

7.2. Time-varying Case

We now consider the more general case of time-varying datasets that are to be explored by extracting and rendering isosurfaces corresponding to a time step and an isovalue. We can index the 270 time steps of the Richtmyer-Meshkov dataset using our indexing scheme. The size of the resulting indexing structure is $1.6MB$, which easily fits into the main memory of a node. The layout of the data of each time step will be distributed across the processors as before. Extracting an isosurface of a time step amounts to determining the appropriate indexing structure for that time step, which can easily be performed since the whole indexing structure is in main memory. Table 8 shows the results for time steps 180 through 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).

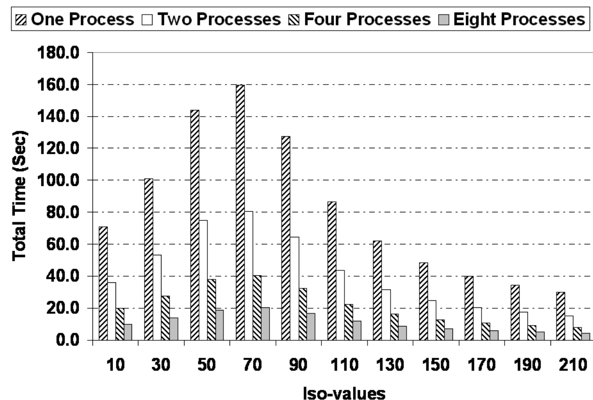

Figure 5. Overall Time of up to eight processors over a range of Isovalues.

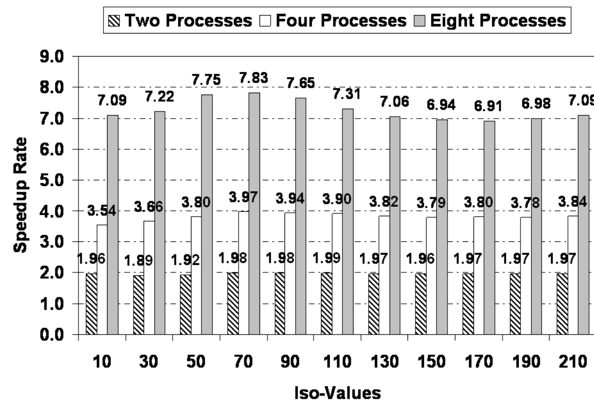

Figure 6. Corresponding Speedups of up to eight processors over a range of Isovalues.

8. Conclusion

In this paper, we have presented a new indexing scheme 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 data structure. The data is arranged in a compact layout on the disk, which enables optimal I/O performance. The size of our indexing structure is $O(n \log n)$ compared to $\Omega(N)$ for the standard interval tree, where N is the number of all possible pairs of scalar field values appearing in metacells and n is the number of their distinct endpoints. We have shown that our indexing structure can easily be adapted to a multiprocessor environment, provably delivering an efficient and scalable performance. The algorithm was tested extensively on the Richtmyer-Meshkov dataset, and its performance consistently agrees with the analysis reported in the paper.

Acknowledgements

We would like to thank Fritz McCall and Brad Erdman for setting up the visualization cluster, Derek Juba and Youngmin Kim for their 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 dataset available to us and for guiding us through the initial stages of using it. This work was supported by the NSF research infrastructure grant CNS-04-03313.

References

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

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

[3] J. Wilhelms and A.Van Gelder. Octrees for faster isosurface generation. In *Computer Graphics(San Diego Workshop on Volume Visualization)*, vol. 24, pp. 57–62, 1990.

[4] P. M. Sutton and 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, pp. 147–154, Oct. 25–29 1999.

[5] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for improved interactivity. In *1996 Volume Visualization Symposium*, pp. 39–46, Oct 1996.

[6] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extreme graph and sorted boundary cell lists. In *IEEE Transactions on Visualization and Computer Graphics*, 1(4): pp. 319–327, Dec 1995.

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

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

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

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

Time-Step	Num of Active Meta Cells	Number of Triangles	Overall Time (Sec)	Overall Rate ($x10^6$/s)
180	2,249,247	499,936,470	35.971	13.898
181	2,259,741	502,356,776	32.934	15.253
182	2,269,996	504,717,566	32.717	15.427
183	2,280,249	507,140,194	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	514,222,228	34.121	15.071
187	2,320,869	516,675,166	34.524	14.966
188	2,330,322	519,026,094	33.732	15.387
189	2,340,363	521,496,932	34.300	15.204
190	2,295,699	516,444,997	31.405	16.445
191	2,360,385	526,339,524	35.163	14.969
192	2,370,458	528,728,460	34.523	15.315
193	2,380,148	531,149,914	35.256	15.066
194	2,389,433	533,600,176	35.172	15.171
195	2,399,116	536,050,312	35.891	14.936

Table 8. Overall performance of four processors on Isovalue 70 over 16 time steps

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

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

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

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

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

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

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

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

[19] L. Arge and J. S. Vitter. Optimal Dynamic Interval Management in External Memory (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pp. 560–569, 1996.

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

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

[22] H. Zhang and T. S. Newman. Efficient Parallel Out-of-core Isosurface Extraction. In *Proc. IEEE Symposium on parallel and large-data visualization and graphics (PVG) '03*, pp. 9–16, Oct. 2003.

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

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

[25] http://graphics.stanford.edu/data/voldata/

[26] http://vis.computer.org/vis2004contest/data.html

[27] http://vis.computer.org/vis2006/

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

[29] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner and J. Klosowski. Chromium: A Stream Processing Framework for Interactive Rendering on Clusters. In *Proceedings of SIGGRAPH 2002*, pp. 693–702, 2002.

[30] S. Molnar, M. Cox, D. Ellsworth and H. Fuchs. A Sorting Classification of Parallel Rendering. In *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, pp. 23–32, July 1994.