Isosurface Extraction in Time-varying Fields Using a Temporal Branch-on-Need Tree (T-BON)

Philip Sutton Charles D. Hansen
Department of Computer Science
University of Utah
Salt Lake City, UT 84112
[psutton | hansen] @cs.utah.edu

Abstract

The Temporal Branch-on-Need Tree (T-BON) extends the three-dimensional branch-on-need octree for time-varying isosurface extraction. At each time step, only those portions of the tree and data necessary to construct the current isosurface are read from disk. This algorithm can thus exploit the temporal locality of the isosurface and, as a geometric technique, spatial locality between cells in order to improve performance. Experimental results demonstrate the performance gained and memory overhead saved using this technique.

Keywords: isosurface, time-dependent scalar field visualization, multiresolution methods, octree

1 Introduction

Isosurface extraction is an important technique for visualizing volumetric data. By exposing contours of constant value, isosurfaces provide a mechanism for understanding the structure of a scalar field. This method has been used effectively in several disciplines, including medicine [12, 18], computational fluid dynamics (CFD) [6, 7], and molecular dynamics [10, 14]. Data sets from computational, as well as measurement sources, have continued to increase in size, often consisting of multiple time steps. Visualizing changes over time is crucial for understanding the dynamic behavior of the data. However, little attention has been focused on methods that directly address time dependent data.

The original Marching Cubes algorithm [13, 22] examined all cells in the data set to construct an isosurface. There has been tremendous research focused on reducing the number of cells visited when constructing an isosurface [1, 4, 11, 17]. While the search structures introduced by many of these methods increase the storage requirements, this overhead is offset by the acceleration of the isosurfacing technique. For time-varying data sets, data structures that index all cells at every time step can incur significant storage overhead. When the data and associated structures cannot all fit in memory, reading the required portions from disk presents an imposing bottleneck, potentially nullifying the performance gained from constructing the supplemental search structure.

We present an isosurface extraction algorithm for time-varying fields that minimizes the impact of the I/O bottleneck. By reading only those portions of the data and search tree necessary to construct the current isosurface, the *Temporal Branch-on-Need Tree (T-BON)* makes efficient use of I/O bandwidth while retaining the accelerated search characteristics of a hierarchical extraction technique. Since the tree is based on the geometry of the data, spatial coherence between cells can be exploited to increase performance.

In the following sections, we first discuss related work and then present our algorithm for extracting isosurfaces in time-varying fields. We then present experimental results demonstrating our algorithm's performance on three large-scale, time-varying data sets. Finally, we discuss our conclusions and suggest directions for future work.

2 Related Work

A number of different techniques have been introduced to increase the efficiency of isosurface extraction over the linear search proposed in the Marching Cubes algorithm [13, 22]. Wilhelms and van Gelder [21] describe the branch-on-need octree (BONO), a space-efficient variation on the traditional octree. Their data structure partitions the cells in the data based on their geometric positions. Extreme values (minimums and maximums) are propagated up the tree during construction such that only those nodes that span the isosurface, i.e. those with $min < isovalue < max$, are traversed during the extraction phase.

More recent methods have focused on partitioning the cells based on their extreme values. Livnat et al. [11] introduced the *span space*, where each cell is represented as a point in 2d space. The point's x-coordinate is defined by the cell's minimum value, and the y-coordinate by the maximum value. The NOISE algorithm described in [11] uses a Kd-tree to organize the points. Shen et al. [17] use a lattice subdivision of span space in their ISSUE algorithm. This simplifies and accelerates the search phase of the extraction, as only one element in the lattice requires a full min-max search of its cells. This acceleration comes at the cost of a less efficient memory footprint than the Kd-tree.

The Interval Tree technique introduced by Cignoni et al. [4] guarantees worst-case optimal efficiency. Cells, represented by the intervals defined by their extreme values, are grouped at the nodes of a balanced binary tree. For any isovalue query, at most one branch from a node is traversed.

An alternate technique is to propagate the isosurface from a set of seed cells. Itoh et al. [8, 9], Bajaj et al. [1], and van Kreveld et. al. [19] construct seed sets that contain at least one cell per connected component of each isosurface. The isosurface construction begins at a seed and is traced through neighboring cells using adjacency and intersection information.

An algorithm to improve I/O performance and allow efficient isosurface extraction on data sets larger than physical memory was described by Chiang et al. [2, 3]. An interval tree is built on disk using a two-level hierarchy. Cells are first grouped into meta-cells and a meta-interval defined. These meta-intervals are then composed into an interval tree, which is divided into disk block-sized groups to allow efficient transfer from disk.

Weigle and Banks [20] consider time-varying scalar data as a four-dimensional field. They construct an "isovolume" for each isovalue, representing the volume swept by the isosurface over time. Imposing a time constraint on the isovolume yields an instantaneous surface. This method elegantly captures temporal coherence, but is impractical for large data sets.

Shen [16] proposed the *Temporal Hierarchical Index Tree* to perform isosurface extraction on time-varying data sets. This method classifies the data cells by their extreme values over time. Temporal variation of cells is defined using lattice subdivision, extending the ISSUE algorithm. Nodes in the tree contain cells with differing temporal variation and are paged in from disk as needed to extract an isosurface at a particular time step. At every time step, an ISSUE search [17] is performed at each node. In order to accelerate the full min-max search, an Interval Tree is constructed in those lattice elements that may require such a search. The Temporal Hierarchical Index Tree shows significant improvement in storage requirements over construction of a span-space search structure which treats each time step as an independent data set. It achieves this while retaining an efficient search strategy for isosurface extraction.

Shen's work clearly accelerates the search for isosurfaces in time dependent data. However, at each time step the entire data domain (time step) is loaded into physical memory. The isosurface extraction process potentially needs to access all of the time steps in a time-varying data set. If all time steps do not simultaneously fit into physical memory, I/O can become a bottle neck. As noted by Wilhelms and Van Gelder [21], for a particular isovalue, large portions of the data not containing the isovalue need not be examined. Similarly these same large portions of the data need not be read from disk when constructing an isosurface. For time dependent data sets, this savings can be significant and has led us to develop a method aimed at exploiting this observation.

3 Temporal Branch-on-Need Tree (T-BON)

We use an algorithm that exploits locality in data by extending the branch-on-need octree, thus providing a mechanism that extracts isosurfaces efficiently from time-varying data. The branch-on-need octree resembles the even-subdivision octree, but partitions the cells more efficiently when the dimensions of the volume are not powers of two. Figure 1 compares the strategies in two dimensions. The even-subdivision strategy divides the volume in each direction at each level of the tree, while the branch-on-need strategy partitions the volume such that the "lower" subdivision in each direction covers the largest possible power of two cells. This results in fewer nodes, allowing the tree to be traversed more efficiently.

The branch-on-need octree provides an efficient search while introducing a low amount of overhead. In the worst case, when the volume contains $(n+1)^3$ cells (n is a power of two), $\frac{n^3}{7} + n^2$ nodes are produced [21]. Cells can be represented by an integer index into the data. Value-decomposition techniques such as ISSUE and the Interval Tree require two floating point numbers (minimum and maximum) and an integer index for each cell, in addition to the memory requirements of the search structure. The low storage cost makes the branch-on-need octree ideal for performing isosurface extraction on large, time dependent data sets.

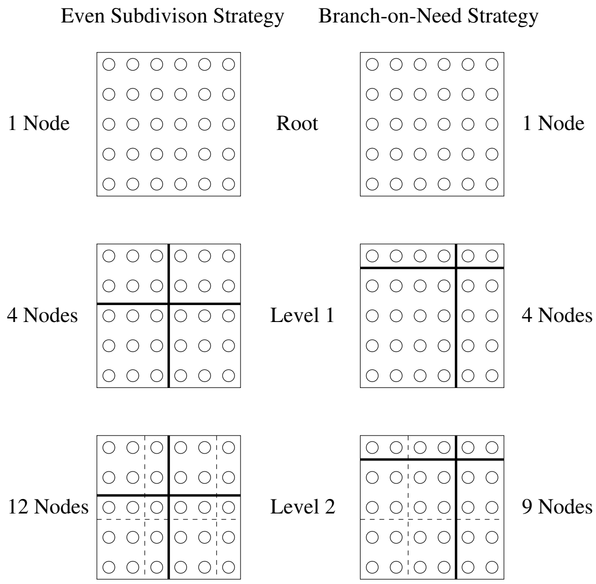

Figure 1: Two-dimensional example of the branch-on-need algorithm (from Wilhelms and van Gelder [21]). The branch-on-need strategy produces fewer nodes when the dimensions of the data are not powers of two.

3.1 Construction

In the preprocessing step, a branch-on-need octree is built for each time step in the data and stored to disk in two sections. The information common to all trees is saved only once. This includes the general infrastructure of the tree, such as branching factors and pointers to children or siblings. This information can be created knowing only the dimensions of the data. Extreme values for the nodes are computed and stored separately, as these values can vary at each time step. Our implementation represents the tree using a linear array, so all pointers are integer indices. The tree is packed in depth-first search order to facilitate retrieval in the search phase.

3.2 Search and Extraction

Before any isovalue queries, the tree infrastructure is read from disk and recreated in main memory. Queries are then accepted in the form $(timestep, isoval)$. Query resolution in the T-BON algorithm is based on demand-driven paging [5]. First the root extreme values of the branch-on-need octree corresponding to $timestep$ are read. If these values span $isoval$, the children of the root node are read. This process repeats recursively until the leaf nodes are reached, as shown in Figure 2. If a leaf node spans $isoval$, the disk block containing the data for the cells in that leaf is added to a list. Once the necessary portions of the tree have been brought into memory and traversed, all blocks in the list are read from disk. This block read causes some extraneous data to be read, but performs much better than randomly accessing the file to bring in strictly those data points required. The block size defaults to the size of a disk sector, but can be modified to tune the algorithm for different I/O subsystems. We show the results of modifying this parameter below. The tree is traversed a second time to construct the isosurface.

When more than one isovalue is queried for the same time step, this process can be modified to avoid rereading identical data. Two lists are maintained, one identifying nodes currently in memory and one representing the disk blocks read. By referencing these lists,

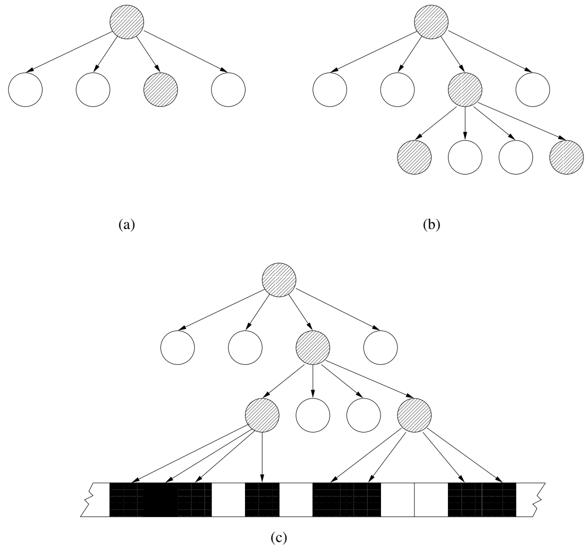

Figure 2: The children of a node that spans the isovalue (indicated by shading) are recursively brought into memory (a,b). At the leaf level, any data needed for the isosurface (black blocks) is fetched from disk, while unnecessary data (white blocks) is not read (c). A second pass through the tree constructs the isosurface using only the data in memory.

only differential nodes and data blocks must be brought from disk. The lists are purged when the time step changes. This additional processing and I/O are dominated by the triangle construction time, so performance in this case is usually comparable to performing a search with the tree and data already in memory.

4 Results

We tested our algorithm on three CFD data sets. Rage512 is a $512 \times 512 \times 512$ simulation of the classic Rayleigh-Taylor hydrodynamic instability, in which two fluids of differing densities mix. Figure 3 shows one isosurface from this data set. Rage256 is a downsampled version of Rage512 and contains $256 \times 256 \times 256$ vertices. The Jet256 data set describes a jet shockwave and also contains $256 \times 256 \times 256$ vertices. Figure 4 shows an isosurface from this data set. All data sets are represented by regular grids. Constructing a branch-on-need octree over an unstructured data set would be more difficult, but Parker et. al. [15] describe a method for constructing such a hierarchy.

4.1 T-BON Results

In the following tables, we describe the speedup obtained by using the T-BON tree instead of a pure BONO approach, which reads the entire tree and data from disk at each time step. We give the minimum, maximum, and average speedup values over multiple isovalues and multiple time steps. The tables show what percentage of the tree and data are read from disk for each of these cases, as well as the number of triangles created. The *blocksize* parameter defines the minimum amount of data fetched from disk in the block read described in Section 3.2. We show results for three values of this parameter. Table 1 shows results for the Rage512 data set. In this experiment, the time parameter was varied over the 21 time steps while holding the isovalue constant. This process was performed for five representative isovalues. Significant speedups were

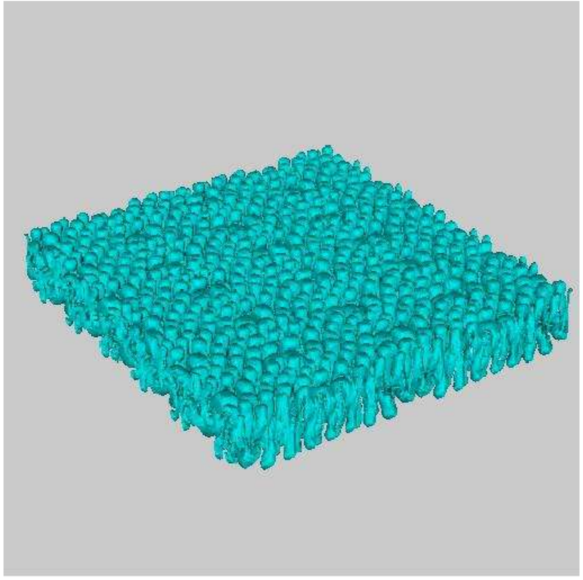

Figure 3: The isosurface with value 1.15 at time 16 in the RAGE data set, showing the bubbles formed by instability.

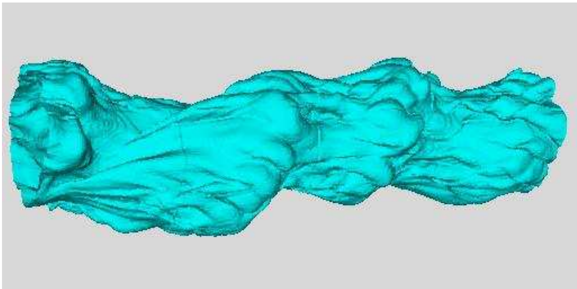

Figure 4: The isosurface with value 37 at time 60 in the Jet Shockwave data set.

Rage512			
# nodes	19,173,961		
data size	536.9 MB $\times$ 21 time steps		
memory footprint	919.6 MB		
Blocksize:	$\frac{1}{2}$Sectorsize		
	Min	Average	Max
speedup	1.26	2.44	4.21
# triangles ($\times 10^6$)	6.59	2.12	0.75
% nodes read	15.47	6.08	2.39
% data read	19.93	9.47	3.32
Blocksize:	Sectorsize		
	Min	Average	Max
speedup	0.94	1.99	4.19
# triangles ($\times 10^6$)	4.47	2.12	7.49
% nodes read	15.47	6.08	2.39
% data read	20.41	10.00	3.53
Blocksize:	4$\times$Sectorsize		
	Min	Average	Max
speedup	1.32	2.63	4.64
# triangles ($\times 10^6$)	6.59	2.12	7.49
% nodes read	15.47	6.08	2.39
% data read	20.57	10.23	3.72

Table 1: Performance results (varying time value) for the Rage512 data set.

obtained in all cases, with the largest *blocksize* showing the best improvement. The worst case performance of the T-BON tree is almost never worse than the pure branch-on-need octree approach, and can be over four times better. The average case shows substantial improvement, achieving approximately twice the performance of the BONO method. Notice that at each time step, we page in only about 20% of the data in the worst case. The low percentages of nodes and data read from disk offset the overhead of the T-BON algorithm and demonstrate the effectiveness of demand-driven fetching. Table 2 shows the results for the same experiment with the Rage256 data set. Table 3 summarizes the experiment on the Jet256 data set. In this case, the time parameter was varied over the 100 time steps, holding one of ten representative isovalues constant. These data sets also perform well under the T-BON structure. Even though the larger data sets show better improvements, the T-BON algorithm is over twice as fast on average for the smaller data sets.

Table 4 demonstrates the performance of the T-BON algorithm, using the Rage512 data set, when the user changes isovalues in the same time step. This query behavior favors the pure BONO approach, since the T-BON algorithm must read more of the tree and data at each new isovalue. The experiment summarized in Table 4 used ten representative isovalues at the initial time step. The T-BON method performs approximately 20% worse than the BONO in the average case, with the first isovalue performing much better (as expected from Table 1 above). Eventually, the entire tree and all of the data will be copied to memory and the performance of the T-BON algorithm will approach that of the pure BONO technique. The performance deficit in Table 4 can be explained by the layering effect in the Rage data set, as is evident in Figure 3. Each cell contains only a small range of values. When the isovalue changes, a new layer of cells must be read from disk, making little use of the previous layer. Table 5 shows similar results for the same experiment using the Rage256 data set. Table 6 summarizes the results for the Jet256 data set at time step 50. This data set allows better performance than the Rage data sets, extracting isosurfaces only 5% slower than the BONO in the average case. The low percentages of data and nodes read at each isovalue imply that values in the Jet256

Rage256			
# nodes	2,396,745		
data size	67.1 MB $\times$ 21 time steps		
memory footprint	114.8 MB		
Blocksize:	$\frac{1}{2}$Sectorsize		
	Min	Average	Max
speedup	1.39	2.35	4.38
# triangles ($\times 10^6$)	1.14	0.66	0.20
% nodes read	20.27	11.93	3.92
% data read	22.58	14.43	5.14
Blocksize:	Sectorsize		
	Min	Average	Max
speedup	1.42	2.37	4.10
# triangles ($\times 10^6$)	1.33	0.66	0.20
% nodes read	20.27	11.93	3.92
% data read	22.72	14.67	5.38
Blocksize:	4$\times$Sectorsize		
	Min	Average	Max
speedup	1.30	2.22	4.47
# triangles ($\times 10^6$)	1.33	0.66	0.20
% nodes read	20.27	11.93	3.92
% data read	23.22	15.28	5.86

Table 2: Performance results (varying time value) for the Rage256 data set.

Jet256			
# nodes	2,396,745		
data size	16.8 MB $\times$ 100 time steps		
memory footprint	50.1 MB		
Blocksize:	$\frac{1}{2}$Sectorsize		
	Min	Average	Max
speedup	1.11	2.52	2.99
# triangles ($\times 10^6$)	0.28	0.090	0.058
% nodes read	7.32	2.37	1.46
% data read	31.91	9.52	2.80
Blocksize:	Sectorsize		
	Min	Average	Max
speedup	1.25	2.83	3.98
# triangles ($\times 10^6$)	0.30	0.090	0.053
% nodes read	7.32	2.37	1.46
% data read	34.69	10.75	3.66
Blocksize:	4$\times$Sectorsize		
	Min	Average	Max
speedup	1.31	2.86	3.98
# triangles ($\times 10^6$)	0.31	0.090	0.053
% nodes read	7.32	2.37	1.46
% data read	51.56	17.67	11.13

Table 3: Performance results (varying time value) for the Jet256 data set.

Rage512			
Blocksize:	$\frac{1}{2}$Sectorsize		
	Min	Average	Max
speedup	0.46	0.82	1.56
% nodes read	7.44	1.46	0.17
% data read	13.54	2.57	0.00
Blocksize:	Sectorsize		
	Min	Average	Max
speedup	0.46	0.84	1.56
% nodes read	7.44	1.46	0.17
% data read	13.70	2.50	0.00
Blocksize:	4×Sectorsize		
	Min	Average	Max
speedup	0.51	0.96	1.83
% nodes read	7.44	1.46	0.17
% data read	14.02	2.66	0.00

Table 4: Performance results (constant time value) for the Rage512 data set.

Rage256			
Blocksize:	$\frac{1}{2}$Sectorsize		
	Min	Average	Max
speedup	0.47	0.80	1.30
% nodes read	9.29	2.00	0.067
% data read	17.95	2.30	0.00
Blocksize:	Sectorsize		
	Min	Average	Max
speedup	0.44	0.75	1.30
% nodes read	9.29	2.00	0.067
% data read	18.75	2.30	0.00
Blocksize:	4×Sectorsize		
	Min	Average	Max
speedup	0.46	0.79	1.28
% nodes read	9.29	2.00	0.067
% data read	20.09	2.44	0.00

Table 5: Performance results (constant time value) for the Rage256 data set.

Jet256			
Blocksize:	$\frac{1}{2}$Sectorsize		
	Min	Average	Max
speedup	0.47	0.95	2.89
% nodes read	2.02	0.32	0.035
% data read	7.40	0.93	0.00
Blocksize:	Sectorsize		
	Min	Average	Max
speedup	0.48	0.95	2.93
% nodes read	2.02	0.32	0.035
% data read	9.28	1.08	0.00
Blocksize:	4×Sectorsize		
	Min	Average	Max
speedup	0.48	0.95	2.90
% nodes read	2.02	0.32	0.035
% data read	16.60	1.66	0.00

Table 6: Performance results (constant time value) for the Jet256 data set.

Rage256			
Time Step 10			
isovalue	no locality	locality	savings
1.0	2.72	2.29	15.8%
1.1	8.35	6.29	24.7%
1.2	6.20	5.45	12.1%
1.3	6.05	3.95	34.7%
1.4	4.22	3.43	18.7%
1.5	1.51	1.31	13.2%

Table 7: Triangle construction time (in seconds) for the Rage256 data set, with and without local caching.

Jet256			
Time Step 50			
isovalue	no locality	locality	savings
5	0.72	0.60	16.7%
10	0.61	0.50	18.0%
15	0.69	0.56	18.8%
20	0.74	0.58	21.6%
200	0.76	0.59	22.4%
210	0.76	0.59	22.4%

Table 8: Triangle construction time (in seconds) for the Jet256 data set, with and without local caching.

data set are more evenly distributed than in the Rage data sets.

4.2 Implementation Details

Our implementation initially allocates enough memory for the data and tree structure of one time step and maintains this memory footprint throughout execution. Reads from disk simply overwrite any previous contents, while the two-pass traversal guarantees that no stale information is accessed. This organization optimizes for speed — a dynamic structure could use only that amount of memory needed for the current time step and isovalue, but would slow execution. A dynamic structure could easily be implemented for systems with limited memory to perform out-of-core isosurface extraction. The basic technique of reading the data and tree nodes on demand provides a convenient and intuitive basis for such an algorithm.

As a geometry-based technique, the temporal branch-on-need tree can leverage the spatial locality in the data to avoid performing duplicate calculations. Wilhelms and van Gelder use a hash table to exploit this property. Once the intersection point of the isosurface has been calculated along a cell edge, that edge is placed into a hash table, allowing cells that share the edge to retrieve the interpolated point without a calculation. When all cells that share an edge have accessed its hash entry, the edge is deleted from the table. As our implementation has no explicit representation of edges, the cost of constructing a hash key on the fly and accessing the table would not aid our algorithm's performance. Instead, we use a form of local caching similar to the chess-board approach proposed by Cignoni et al. [4]. Leaf nodes in the tree contain up to eight data cells. When a leaf spans the current isovalue, triangles are constructed in all these cells, using a local array to cache the interpolated points. Cells in the same leaf block access this array instead of interpolating a shared edge. Tables 7 and 8 show the reduction in triangle construction time using this technique on six representative isovalues. The times given are for constructing individual triangles in the Rage256 and Jet256 data sets. Methods for constructing triangle strips could take advantage of this local cache and improve both construction and rendering times.

5 Conclusions and Future Work

We have presented a low-latency isosurface extraction algorithm for time varying fields. By reading only those portions of the tree and data necessary for constructing the current isosurface, the temporal branch-on-need tree minimizes the I/O overhead. Additionally, this method requires low structural overhead and can exploit spatial locality between cells to increase performance. Experimental results demonstrate the advantages over reading the entire data set and search structure for each time step.

Several directions for improvements and extensions exist. We plan to apply this technique to unstructured data sets using the technique described by Parker et. al. [15] for constructing the hierarchy. Creating a dynamic structure for memory savings and out-of-core applications was mentioned in Section 4.1. The temporal branch-on-need tree can be parallelized to reduce the wall clock execution time. High level nodes in the tree could be distributed to different processing units, retaining the temporal and spatial locality inherent in the algorithm. We also plan to investigate bricking techniques. Rearranging the data points would allow for a more efficient first pass, requiring recursion to a lesser depth in the tree. This would also reduce the number of unnecessary data points transferred in the block read. Finally, methods of exploiting low temporal variations in the data could be added. For example, if the extreme values of a tree nodes changed only slowly over time, then a single node could be used for multiple time steps, as in [16]. However, the utility of this would depend on the behavior of the data and may not improve performance.

6 Acknowledgments

This work was supported in part by the C-SAFE DOE ASCI Alliance Center, the DOE Advanced Visualization Technology Center (AVTC), and a grant from LLNL. The authors would like to thank Yarden Livnat for his comments and suggestions. The Rage data set is courtesy of Robert Weaver (Los Alamos National Laboratory). The Jet data set was obtained from the Advanced Visualization Technology Center data repository at Argonne National Laboratory.

References

[1] Chandrajit L. Bajaj, Valerio Pascucci, and Daniel R. Schikore. Fast Isocontouring For Improved Interactivity. In *1996 Symposium on Volume Visualization*, pages 39–46. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1996.

[2] Yi-Jen Chiang and Cláudio T. Silva. I/O Optimal Isosurface Extraction. In Roni Yagel and Hans Hagen, editors, *Proceedings of Visualization 1997*, pages 293–300. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1997.

[3] Yi-Jen Chiang, Cláudio T. Silva, and William J. Schroeder. Interactive Out-Of-Core Isosurface Extraction. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings of Visualization 1998*, pages 167–174. IEEE Computer Society, ACM Press, New York, NY, October 1998.

[4] Paolo Cignoni, Paola Marino, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, April 1997.

[5] Michael Cox and David Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. In Roni Yagel and Hans Hagen, editors, *Proceedings of Visualization 1997*, pages 235–244. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1997.

[6] Jean M. Favre. Towards Efficient Visualization Support for Single-block and Multi-block Datasets. In Roni Yagel and Hans Hagen, editors, *Proceedings of Visualization 1997*, pages 423–428. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1997.

[7] Philip D. Heermann. Production Visualization for the ASCI One TeraFLOPS Machine. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings of Visualization 1998*, pages 459–462. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1998.

[8] Takayuki Itoh and Koji Koyamada. Automatic Isosurface Propagation Using an Extrema Graph and Sorted Boundary Cell Lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, December 1995.

[9] Takayuki Itoh, Yasushi Yamaguchi, and Koji Koyamada. Volume Thinning for Automatic Isosurface Propagation. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of Visualization 1996*, pages 303–310. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1996.

[10] Marco Lanzagorta, Milo V. Kral, J.Edward Swan II, George Spanos, Rob Rosenberg, and Eddy Kuo. Three-Dimensional Visualization of Microstructures. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings of Visualization 1998*, pages 487–490. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1998.

[11] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using The Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[12] William E. Lorensen. Marching Through the Visible Man. In Gregory M. Nielson and Deborah Silver, editors, *Proceedings of Visualization 1995*, pages 368–373. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1995.

[13] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In Maureen C. Stone, editor, *Computer Graphics*, volume 21, pages 163–169. ACM SIGGRAPH, ACM SIGGRAPH, July 1987.

[14] Colin R. F. Monks, Patricia J. Crossno, George Davidson, Constantine Pavlakos, Abraham Kupfer, Cláudio Silva, and Brian Wylie. Three Dimensional Visualization Of Proteins In Cellular Interactions. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of Visualization 1996*, pages 363–366. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1996.

[15] Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. Interactive Ray Tracing for Volume Visualization. *Submitted to IEEE Transactions on Visualization and Computer Graphics*, 1999.

[16] Han-Wei Shen. Isosurface Extraction in Time-varying Fields Using a Temporal Hierarchical Index Tree. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings of Visualization 1998*, pages 159–164. IEEE Computer Society, ACM Press, New York, NY, October 1998.

[17] Han-Wei Shen, Charles D. Hansen, Yarden Livnat, and Christopher R. Johnson. Isosurfacing in Span Space with Utmost Efficiency (ISSUE). In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of Visualization 1996*, pages 287–294. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1996.

[18] Ulf Tiede, Thomas Schiemann, and Karl Heinz Höhne. High Quality Rendering of Attributed Volume Data. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings of Visualization 1998*, pages 255–262. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1998.

[19] Marc van Kreveld, René van Oostrum, Chandrajit Bajaj, Valerio Pascucci, and Dan Schikore. Contour Trees and Small Seed Sets for Isosurface Traversal. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pages 212–220. ACM Special Interest Groups for Graphics and Algorithms and Computation Theory, ACM Press, New York, NY, June 1997.

[20] Chris Weigle and David C. Banks. Extracting Iso-valued Features in 4-dimensional Scalar Fields. In *1998 Symposium on Volume Visualization*, pages 103–110. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, October 1998.

[21] Jane Wilhelms and Allen van Gelder. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[22] Geoff Wyvill, Craig McPheeters, and Brain Wyvill. Data Structure for Soft Objects. *The Visual Computer*, 2(4):227–234, 1986.