Out-of-Core Isosurface Extraction of Time-Varying Fields over Irregular Grids

Yi-Jen Chiang*

Department of Computer and Information Science, Polytechnic University

Abstract

In this paper, we propose a novel out-of-core isosurface extraction technique for large time-varying fields over irregular grids. We employ our *meta-cell* technique to explore the *spatial coherence* of the data, and our *time tree* algorithm to consider the *temporal coherence* as well. Our one-time preprocessing phase first partitions the dataset into *meta-cells* that cluster spatially neighboring cells together and are stored in disk. We then build a *time tree* to index the meta-cells for fast isosurface extraction. The time tree takes advantage of the temporal coherence among the scalar values at different time steps, and uses *BBIO trees* as secondary structures, which are stored in disk and support I/O-optimal interval searches. The time tree algorithm employs a novel *meta-interval collapsing scheme* and the *buffer technique*, to take care of the temporal coherence in an I/O-efficient way. We further make the time tree *cache-oblivious*, so that searching on it automatically performs optimal number of block transfers between any two consecutive levels of memory hierarchy (such as between cache and main memory and between main memory and disk) *simultaneously*. At run-time, we perform optimal cache-oblivious searches in the time tree, together with I/O-optimal searches in the BBIO trees, to read the *active* meta-cells from disk and generate the queried isosurface efficiently. The experiments demonstrate the effectiveness of our new technique. In particular, compared with the query-optimal main-memory algorithm [Cignoni et al. 1997] (extended for time-varying fields) when there is not enough main memory, our technique can speed up the isosurface queries from more than 18 hours to less than 4 minutes.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics Data Structures and Data Types I.3.m [Computer Graphics]: Miscellaneous—Scientific Visualization

Keywords: isosurface extraction, out-of-core techniques, time-varying fields, irregular grids

1 Introduction

In recent years, new challenges for scientific visualization emerged as the size of data generated from simulations grew exponentially [Bryson et al. 1997]. Such an exponential growth in data size is mainly due to the fact that scientists are now capable of performing simulations with finer temporal resolutions and a larger number of time steps. To understand the data and extract important dynamic features, it is very crucial for the scientists to explore the data back and forth in time, with various visualization parameters. However, the sheer size of the data often makes the task of interactive exploration impossible, as only a small portion of the data in the entire time series can fit into main memory, and the computation cost is often too high for an algorithm to run in real-time. Despite the importance and the big challenges posed by the time-varying datasets, most of the previous research has focused on the visualization of steady-state data (i.e., data with only a single time step), with only very few results reported on time-varying data visualization.

In this paper, we address the issues of limited main memory size and insufficient computing speed of the current graphics workstations for large time-varying data, by proposing a new out-of-core isosurface extraction technique. Our method focuses on the class of *irregular-grid* volume datasets, which is the most general class of volumetric data and has been proposed as an effective means of representing disparate field data that arises in a broad spectrum of applications including structural mechanics, computational fluid dynamics, partial differential equation solvers, and shock physics.

Isosurface extraction is one of the most important and widely used classes of visualization techniques for volume datasets. Specifically, for time-varying fields, performing an isosurface query (q,t) is to extract and display all the points (a surface) in the volume whose scalar values at time step t are the *isovalue* q. Although isosurface techniques have been developed to a high degree of sophistication, most of the algorithms require the entire dataset to be kept in main memory, which is a severe limitation on their applicability, especially for large scientific applications. Previously, we gave out-of-core isosurface techniques that are suitable for irregular grids [Chiang and Silva 1997; Chiang et al. 1998; Chiang et al. 2001] but do not work for time-varying data, and Sutton and Hansen [2000] gave an out-of-core isosurface technique for time-varying fields that is mainly focusing on regular grids and moreover does not make use of the *temporal coherence* of the data. The isosurface algorithm of Shen [1998] for time-varying fields takes advantage of the *temporal coherence*, but does not particularly focus on efficient out-of-core computation. Our new technique in this paper tries to fill in this gap.

Our algorithm makes use of the *spatial coherence* of the datasets via the *meta-cell* technique, as well as the *temporal coherence* via the *time tree* algorithm. There are two phases in our technique. In the one-time preprocessing phase, we first partition the dataset into *meta-cells* that are clusters of spatially neighboring cells and are stored in disk. The meta-cell technique was first proposed in [Chiang et al. 1998] that allows I/O-efficient partitioning and retrieving of the datasets for *irregular grids*. Here we make non-trivial extensions of the meta-cell technique so that it can efficiently handle time-varying fields as well. We then build a new *time tree* to index the meta-cells for fast isosurface extraction. The time tree takes advantage of the temporal coherence among the scalar values in different time steps, and uses our *BBIO trees* [Chiang et al.

*e-mail: yjc@poly.edu. Research supported in part by NSF CAREER Grant CCR-0093373, NSF Grant ACI-0118915, and NSF ITR Grant CCR-0081964.

1998] as secondary structures, which are stored in disk and support I/O-optimal interval searches. To make use of the temporal coherence, our novel *meta-interval collapsing scheme* employs a *Manhattan distance thresholding* in the *span space* and a *bottom-up meta-interval union* approach, which only needs local computations and avoids the global sorting step on all intervals (collected from all cells and all time steps) necessary in the *lattice partition* method used in [Shen 1998]. Moreover, the Manhattan distance thresholding may better capture the temporal coherence than the lattice partition method [Shen 1998]. The meta-interval collapsing scheme is additionally integrated with our new *buffer technique* to perform I/O-efficiently. We further make the time tree *cache-oblivious* by applying the technique of Bender et al. [2002], so that searching on it automatically performs optimal number of block transfers between any two consecutive levels of memory hierarchy (such as between cache and main memory and between main memory and disk) *at the same time*.

At run-time, we perform optimal cache-oblivious searches in the time tree, together with I/O-optimal searches in the BBIO trees, to read the *active* meta-cells from disk that contain all possible cells intersected by the isosurface. Finally, we generate the isosurface efficiently from these meta-cells retrieved.

The experiments demonstrate the effectiveness of our new technique. In particular, we can handle datasets of more than 10 million cells on a PC of only 55MB of RAM very efficiently, in both the preprocessing and the run-time phases. Compared with the query-optimal main-memory algorithm [Cignoni et al. 1997] (extended for time-varying fields) running on the same computer platform when there is not enough main memory, our algorithm can speed up the isosurface queries by a factor of about 9.9 times on a 512MB-RAM PC and a factor of about 281 on a 55MB-RAM PC.

2 Previous Work

In this section, we review the previous work on isosurface extraction, including out-of-core isosurface algorithms. For out-of-core techniques in graphics and scientific visualization problems other than isosurface extraction, we refer to the recent survey by Silva et al. [2002]. For theoretical results on out-of-core algorithms for graphs and for computational geometry problems, we refer to the survey by Vitter [2001]. Research on cache-oblivious algorithms and data structures is only at its early stage, in the theoretical algorithms community. We refer to the paper by Bender et al. [2002] and the references therein for the related work.

There is a very rich literature on isosurface extraction; we refer to [Livnat et al. 1996] for an excellent and thorough review. In Marching Cubes [Lorensen and Cline 1987], all cells in the volume dataset are searched for isosurface intersection. Techniques avoiding exhaustive scanning include using an octree [Wilhelms and Gelder 1990], identifying a collection of *seed cells* and performing contour propagation from the seed cells [Bajaj et al. 1996; Itoh and Koyamada 1995; van Kreveld et al. 1997], NOISE [Livnat et al. 1996], and other efficient methods [Shen and Johnson 1995; Shen et al. 1996]. Almost all these acceleration methods employ the following idea: producing for each cell c an interval $[\min, \max]$ consisting of the minimum and maximum scalar values of the vertices of c, the *active* cells intersected by the isosurface are exactly those cells whose intervals contain the isovalue q. This reduces the problem of finding active cells to that of *interval search*. The first *query-optimal* algorithm was given by Cignoni et al. [1997], by solving the interval search problem using the interval tree [Edelsbrunner 1983]. This gives the optimal query time in terms of main-memory computation. Concurrent to our work in this paper, Bordoloi and Shen [2003] proposed a technique to reduce the space overhead of the indexing structure, by compressing the interval information while maintaining an efficient search performance.

The first *out-of-core* isosurface technique was given by Chiang and Silva [1997]. They developed the *normalization technique* to efficiently access the data in disk, and used the I/O-optimal interval tree [Arge and Vitter 1996] to solve the interval search problem. Later, Chiang et al. [1998] further improved the disk space overhead and the preprocessing time of [Chiang and Silva 1997], at the cost of slightly increasing the isosurface query time, by developing a *two-level* indexing scheme, the *meta-cell* technique, and the *BBIO* tree which is used to index the meta-cells. These techniques are also extended to perform *parallel* out-of-core isosurface extraction and volume rendering by Chiang et al. [2001]. In addition, Bajaj et al. [1999] proposed a parallel and out-of-core isosurface approach based on contour propagation from seed cells.

The techniques mentioned so far are for steady-state datasets, and there are relatively few algorithms for time-varying fields. The *temporal branch-on-need octree* method was given by Sutton and Hansen [2000], and Shen [1998] gave a technique based on the *temporal hierarchical index tree* (the *THI tree* for short). As mentioned before, the technique of [Sutton and Hansen 2000] is an out-of-core approach most suitable for regular grids, and considers the *spatial coherence* rather than the *temporal coherence* of the data. On the other hand, the approach of Shen [1998] is a main-memory algorithm, taking advantage of the *temporal coherence* among the scalar values but not particularly focusing on out-of-core computation.

3 Our Approach

In this section we present our out-of-core isosurface extraction algorithm. We first give an overview, and then present each technical component in detail.

3.1 Overview

There are two major components in our algorithm: the *meta-cell* technique and the *time tree* algorithm. The meta-cell technique takes advantage of the spatial coherence of the dataset and partitions the data into *meta-cells* that cluster spatially neighboring cells together to support I/O-efficient accesses to the data. Here we need to extend the meta-cell technique of [Chiang et al. 1998] so that it can handle time-varying fields I/O-efficiently. Our novel time tree algorithm takes advantage of the temporal coherence of the data and indexes the meta-cells for fast isosurface extraction.

After constructing the meta-cells, we produce, for each meta-cell, the *meta-intervals* for each time step. The purpose of meta-intervals for a meta-cell is analogous to that of an interval for a cell. A meta-cell is *active* for query (q,t) if and only if some meta-interval at time step t contains the isovalue q. Intuitively, a meta-interval for t could be the $[\min, \max]$ interval by taking the minimum and maximum scalar values at t among all vertices in the meta-cell. However, such big range may contain *gaps*[1] in which no cell interval lies. Therefore, we define meta-intervals at time t as the *connected components* among the intervals at t of the cells in that meta-cell. Searching active meta-cells then amounts to performing interval searches on the meta-intervals. For each meta-interval, we store its meta-cell ID, which is the starting position of that meta-cell m in the meta-cell file in disk, to be used to retrieve m.

We now describe the basic data structure of the time tree. The purpose of the time tree is to reduce the number of meta-intervals to be stored in the indexing structures. Similar to the idea of the THI tree [Shen 1998], the time interval over the entire time steps is partitioned hierarchically into a fully balanced binary tree—the primary structure of the *time tree*, so that each tree node corresponds to a time interval that is the union of the time sub-intervals of its child

[1] Gaps only occur when disconnected components of cells belong to the same meta-cell.

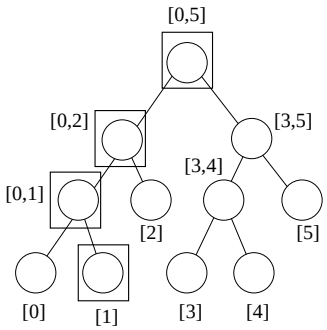

Figure 1: An example of the time tree for a time-varying field with time interval $[0,5]$. Each internal node labeled $[t_1,t_2]$ covers the time span $[t_1,t_2]$, and each leaf labeled $[t]$ corresponds to time step t. The nodes visited by an isosurface query (q,t) with $t = 1$ are indicated by squares.

nodes, where each leaf is for an individual time step (see Fig. 1). A meta-cell *m* is assigned to the highest nodes during whose time intervals the meta-intervals of *m* do not differ too much. Thus a meta-cell may be assigned to multiple nodes: if its meta-intervals over all time steps only change slightly, it may be assigned only to the root; in the other extreme, if its meta-intervals differ a lot at each time step, then it may be assigned to all leaves. When a meta-cell *m* is assigned to a node *u* covering the time span $[t_1,t_2]$, the meta-intervals of *m* at time steps $t_1,t_1+1,\cdots,t_2$ are *collapsed* to result in a fewer number of new meta-intervals; these collapsed meta-intervals are also assigned to *u*. The precise definition of the *closeness* between meta-intervals, as well as the *meta-interval collapsing scheme* are described in Section 3.2.1. It suffices to know here that collapsing meta-intervals only results in reporting a *superset* of the set of the actual active meta-cells for any isosurface query, so that *no* active cell will be missed. Finally, for each node *u* of the time tree, we build a BBIO tree [Chiang et al. 1998] as a secondary structure to store the collapsed meta-intervals assigned to *u*. Each such BBIO tree will be used to facilitate an I/O-optimal interval search on the meta-intervals stored in it.

For an isosurface query (q,t), we traverse along a simple root-to-leaf path in the time tree to visit all nodes whose time spans contain the time step *t* (see Fig. 1). For each such node, we query its BBIO tree for the isovalue *q*. This guarantees that the reported candidate meta-cells contain all active cells, and the isosurface can be generated by performing the Marching Cubes/Tetrahedra algorithm [Lorensen and Cline 1987] on these candidate meta-cells.

In the process of collapsing meta-intervals and assigning them to time-tree nodes, each node *u* may have many meta-intervals assigned to it so that they cannot fit in main memory. We give an efficient I/O technique to address this issue, called the *buffer technique*, described in Section 3.2.2.

The primary structure of the time tree has its size proportional to the number of time steps in the dataset. To make it scalable for a large number of time steps, we would like to make the structure I/O-efficient, applicable for the situations where the time tree cannot fit in main memory. However, standard out-of-core data structures are all stored in disk, requiring at least one disk read to access the tree, even when the entire tree can actually fit in main memory. Observe that the time tree is always needed for a query (as opposed to BBIO trees for which only some of them are visited in a query), so we would like to store the time tree in main memory whenever possible, taking advantage of the available main memory. We would even like it to be *cache efficient* for a high performance. We achieve all these goals by making the time tree *cache-oblivious*, applying the technique of Bender et al. [2002].

The beauty of a cache-oblivious data structure is that we can use *virtual memory* supported by OS, and still achieve an I/O-efficient performance (*I/O-optimal* in our case). Using virtual memory in a naive and straightforward way typically gives a very poor performance, as we may have to read the entire disk block just to access a single, small-size item and most of the disk reads are wasteful. A cache-oblivious data structure, on the other hand, is organized very cleverly so that the page faults generated by OS still give I/O-efficient performance. Moreover, the technique does not require the knowledge of the disk block size, and thus automatically works for *all* disk block sizes. As a result, it works for *any* two consecutive levels of memory hierarchy (such as between cache and main memory and between main memory and disk) *at the same time*. Making our time tree cache-oblivious thus achieves all our desired goals described above. We present the cache-oblivious technique in Section 3.2.3.

In summary, there are two phases in our overall algorithm: the preprocessing phase and the run-time phase. In the preprocessing phase, we perform the following steps.

1. Compute *meta-cells* and store in disk the meta-cell information for each meta-cell.
2. For each meta-cell, produce *meta-intervals* for each time step.
3. Build a time tree. For each meta-cell, use the *meta-cell collapsing scheme* to collapse all its meta-intervals appropriately and assign them to their destination nodes in the time tree. Use the *buffer technique* to hold the (collapsed) meta-intervals assigned to each node of the time tree.
4. For each time tree node *u*, build a BBIO tree as a secondary structure for all (collapsed) meta-intervals assigned to *u*. Each BBIO tree is stored in disk as part of the construction process.
5. Make the primary structure of the time tree *cache-oblivious*. Store the resulting structure in disk.

The entire preprocessing is I/O-efficient, and can be performed in time proportional to running external sorting a few times.

In the run-time phase, we start by reading the primary structure of the cache-oblivious time tree from disk to main memory. For a given query (q,t), we perform the following steps.

1. Traverse the time tree along a root-to-leaf path, visiting all nodes whose time spans contain the time step *t*.
2. For each such node *u* of the time tree visited, query the BBIO tree of *u* in disk to find the meta-intervals containing *q*.
3. For each meta-interval found, read the corresponding meta-cell from disk to main memory.
4. For each meta-cell read, perform the Marching Tetrahedra algorithm [Lorensen and Cline 1987] on its cells to generate isosurface triangles.

The main theme of our out-of-core technique is that the dataset is entirely kept in disk, and we only perform a small number of I/O operations to bring the small portion of the data needed to main memory. We remark that our query algorithm needs one page of disk block size in main memory to traverse the time tree (to hold the current portion paged in by a page fault), two pages to traverse the current BBIO tree, plus the space to hold one meta-cell, which is typically one to two pages (see Sections 3.3 and 4). Therefore we only need about 4–5 pages of main memory. This makes our query performance essentially *independent* of the main memory size available.

We now proceed to describe the technical details of the time tree algorithm and the meta-cell technique.

3.2 Time Tree Algorithm

In this section we describe the detailed algorithms for our time tree. This includes the *meta-interval collapsing scheme*, the *buffer technique*, and the *cache-oblivious technique*.

3.2.1 Meta-Interval Collapsing Scheme

Now we describe our meta-interval collapsing scheme. As mentioned in Section 3.1, for each meta-cell m, the task is to assign m to the highest nodes of the time tree during whose time intervals the meta-intervals of m do not differ too much; such low-variation meta-intervals are then *collapsed* appropriately to reduce the total number of meta-intervals.

First, we need to define how to measure the "closeness" between meta-intervals. We employ the following notion of *Manhattan distance thresholding* in the *span space*: Each meta-interval $[\min, \max]$ is transformed to a point $(\min, \max)$ in the 2-dimensional *span space*; two meta-intervals $[\min_1, \max_1]$ and $[\min_2, \max_2]$ are *close* if the *Manhattan distance* (also called the L_1 *distance*) of their span-space points $(\min_1, \max_1)$ and $(\min_2, \max_2)$ is within L, namely, if

$$|\min_1 - \min_2| + |\max_1 - \max_2| \leq L, \tag{1}$$

where the *Manhattan distance threshold* L is a user-specified parameter.

As a comparison, in the THI algorithm [Shen 1998], the "closeness" is defined by the following *lattice partition scheme*. First, the span space is partitioned into $T \times T$ non-uniformly spaced rectangles, called *lattice elements*, where T is a user-specified parameter. Two intervals are "close" if their span-space points fall into the *same* lattice element. Observe that two points that have a very small Manhattan distance may fall into different, adjacent lattice elements, one on each side of the lattice boundary and hence are considered "not close". On the other hand, two points that have a large Manhattan distance may still fall into the same lattice element (e.g., on the diagonal corners) and thus are considered "close". In that sense, our definition may better capture "closeness" and hence the temporal coherence of the data. Moreover, to define the lattice partition, it is necessary to globally sort all intervals from all cells and all time steps [Shen 1998]. In the out-of-core setting, this requires an expensive external sorting. On the contrary, our definition of "closeness" only needs a simple, local computation.

Now we describe our meta-interval collapsing scheme. We process the meta-cells one by one. For the current meta-cell m, we perform a *bottom-up* meta-interval collapsing scheme. Initially, the meta-intervals at each time t are temporarily deposited to the leaf for time t in the time tree. Recall from Section 3.1 that there may be several meta-intervals for a single time step. We have two types of operations: the *collapsibility test*, and the *union operation* to actually collapse meta-intervals. Starting from the leaves, for each pair of *sibling* nodes u and v, we test whether the meta-intervals deposited to u and those to v should be collapsed. If they should not, then we stop going up from u and v and assign the deposited meta-intervals to u and to v as their destination nodes. If they should, then we remove them from u and v, collapse them by the union operation, deposit the resulting meta-intervals to the parent of u and v, and repeat the process.

The collapsibility test is as follows. For the set I of the meta-intervals of m deposited at a time-tree node, we produce a single *super-interval* s whose min value is the minimum among the min values in I, and whose max value is the maximum among the max values in I. For two sibling nodes u and v, their deposited meta-intervals should be collapsed if their super-intervals s_u and s_v are *close*, according to our "closeness" definition given in (1).

To actually collapse the meta-intervals in sets I_u and I_v deposited to nodes u and v, we perform the *union* operation below. Let all the meta-interval endpoints in I_u be sorted in ascending order; similarly for those in I_v. We merge the two sorted endpoint lists so that the resulting list is also sorted. We now scan through the endpoints of the merged list, with a counter initialized to 0. A left endpoint encountered increases the counter by 1, and a right endpoint decreases the counter by 1. A "$0 \rightarrow 1$" transition starts a new meta-interval, and a "$1 \rightarrow 0$" transition ends the current meta-interval (see Fig. 2). In this way, the *gaps* are kept, so that in the future a query with isovalue q falling in one of the gaps will not cause a wasteful retrieval of the meta-cell from disk. Observe that the resulting meta-interval endpoints are also sorted, ready for the next round of the *union* operation. The list merging step and the counter scanning step can actually be combined into one pass, and the new super-interval obtained at the same time. We remark that initially the meta-interval endpoints in each time-tree leaf are already sorted when given, as originally the meta-intervals at time t are obtained by the *union* operation on the cell intervals at time t.

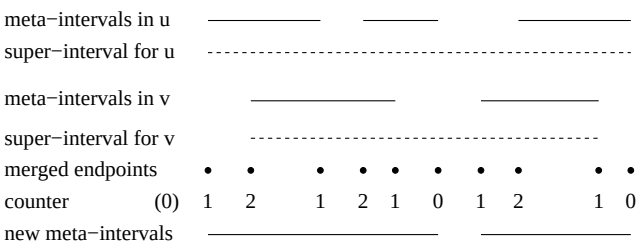

Figure 2: An example of the *union* operation on meta-intervals.

3.2.2 Buffer Technique

Now we describe our *buffer technique*, which is developed to make the *meta-interval collapsing scheme* (see Section 3.2.1) operate in an I/O-efficient way. As we process the meta-cells one by one and collapse and assign their meta-intervals to the appropriate destination nodes in the time tree, *each* time-tree node potentially can accumulate too many assigned meta-intervals to fit in main memory, or the assigned meta-intervals collectively from *all* time-tree nodes can exceed the amount to fit in main memory. We develop the following *buffer technique* to address this issue.

For each time-tree node u, we allocate one page of disk block size in main memory as the *buffer* for u. These buffers will be the only main-memory space needed to handle all assigned meta-intervals. Meta-intervals assigned to u are put to the buffer for u. When the buffer is full, we write the content of the buffer out to an appropriate place in disk, and the buffer is again free for use. Observe that we always write the meta-intervals to disk in units of a full block, and thus the number of I/O operations is optimal. The meta-intervals assigned to u, either in disk or in the buffer for u, will be accessed when it comes to build the BBIO tree for u. To allocate the "place holder" in disk, naively we would create one file per buffer. However, this is infeasible, because this would require us to create and keep open an *unbounded* number of files ($2t-1$ files for t time steps in data) during the procedure, but there is a hard limit (e.g. 256 in Unix) on the number of files a process can open. Our solution is to use a single file to hold all such buffer outputs, allocating a fixed-length "sub-file" of contiguous places for each buffer. The sub-file of a particular buffer might be full at some point, however. We use another file to collect all such "overflow" blocks from all sub-files. Each overflow block is stored in the next available place in that "overflow" file, with the position recorded. Note that different overflow blocks of the same buffer may not be stored contiguously, and hence accessing these blocks may be slower than accessing those in the sub-file of the buffer. This is why we want to

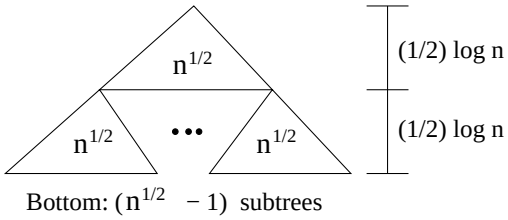

Figure 3: Intuition of the first-level recursion in making a tree cache-oblivious. An n-node tree is partitioned into $O(\sqrt{n})$ subtrees, shown as triangles, each with height $\frac{1}{2}\log n$ and $O(\sqrt{n})$ nodes.

use sub-file first and then the overflow file.

Another similar issue is that we have to implement one BBIO tree for each time-tree node. Each BBIO tree needs three files (files for tree nodes, for left lists, and for right lists). If we use three files for *each* BBIO tree, we would need to create an excessive number of files. Moreover, in the run-time phase, we would need to repeatedly perform wasteful open- and close-file operations just to keep the number of open files under the hard limit, which would greatly slow down the query performance. We address this issue by creating all BBIO trees in three global files. Each time a new BBIO tree needs to be created, we allocate the next available block from the tree-node file and record the position of the new root. The left/right list of a BBIO-tree node is also allocated from the next available space in the left-/right-list file. This provides a simple solution without changing the original method for constructing the BBIO tree. In the run time, by opening the three global files, all BBIO trees are open, ready for performing fast isosurface queries.

3.2.3 Cache-Oblivious Time Tree

We now describe the technique of Bender et al. [2002] for making a fully balanced binary tree *cache-oblivious*. We apply this technique to the primary structure of our time tree, which is a fully balanced binary tree.

The conversion approach is a recursive process. Suppose the tree has n nodes, with height $\log n$ (since it is fully balanced). At the first level of recursion, we partition the tree into $O(\sqrt{n})$ subtrees, each of height $\frac{1}{2}\log n$ and number of nodes $O(\sqrt{n})$. This is achieved by first taking the top subtree that contains the root and has height $\frac{1}{2}\log n$. It can be seen that this subtree has $O(\sqrt{n})$ nodes (since it is a complete binary tree with height $\frac{1}{2}\log n = \log\sqrt{n}$) and hence $O(\sqrt{n})$ leaves. The children of these leaves are the roots of the remaining subtrees, each of which again has height $\frac{1}{2}\log n$ and size $O(\sqrt{n})$. See Fig. 3.

After the partition, we organize all nodes in the same subtree into consecutive places in memory (such as consecutive entries in an array), viewing each subtree as a "block" at the current level of recursion. The array is just a "place holder" of the nodes of the original tree; the parent-child relationship of the original tree can be maintained by using pointers/indices to the array, updated for each re-arrangement of the nodes in the array. This finishes the first level of recursion. At the next level, we apply the same process recursively to each subtree, so that its nodes are re-arranged again *inside* its own "block." The recursive process continues until each subtree is just a *single* node. Obviously, the space requirement is optimal $O(n)$. To traverse along a root-to-leaf path, we proceed as usual following the child pointers.

It is interesting to see that traversing along a root-to-leaf path, though visiting $O(\log n)$ nodes, only goes across $O(\log_B n)$ disk block boundaries and hence causes $O(\log_B n)$ page faults or disk I/O's, where B is the disk block size, for *any value* of B. Note that $O(\log_B n)$ is the optimal number of I/O operations needed, matching the search I/O bound of a usual B-tree.

For the analysis purpose, suppose that the recursive process stops when the current subtree size S is no larger than B, for an underlying B. Notice that at this point each subtree can fit into a disk block, and the subsequent recursions only re-arrange the nodes inside a disk block and thus do not affect the I/O performance. Therefore we may as well ignore these subsequent recursions for the purpose of analysis. (Of course B is unknown and the actual process stops when a subtree is just a single node. These facts should not be confused with the analysis-only description here.)

Each subtree, with size S, has height $O(\log S)$ and fits in one disk block. Going along a root-to-leaf path visits $O(\log n)$ nodes, in which we go across a disk block boundary (i.e., a subtree boundary) for every $O(\log S)$ nodes. This shows that we perform $O(\frac{\log n}{\log S}) = O(\log_S n)$ I/O's. Observe that we stop the partition when $S \le B$. Also, since each recursion reduces the subtree size from n to $\sqrt{n}$, we have $S^2 > B$ (otherwise if $S^2 \le B$ we would have stopped the recursion earlier). This means that $S > \sqrt{B}$, and hence the I/O bound is $O(\log_S n) = O(\log_{\sqrt{B}} n) = O(\log_B n)$, as desired. Since the algorithm does not know the specific value of B and it works for *any* value of B, the same "I/O" bound automatically applies to the number of block transfers between *any* pair of consecutive levels of the memory hierarchy, such as between cache and main memory and between main memory and disk, *at the same time*.

3.3 Meta-cell Technique for Time-Varying Fields

In this section we describe our new *meta-cell technique*, which is developed by making important extensions from the original meta-cell technique [Chiang et al. 1998] to handle time-varying fields.

We first review the original meta-cell technique [Chiang et al. 1998] that only considers the case of a single time step for each vertex. Typical input of an unstructured-grid dataset has a vertex list and a cell list, where each vertex appears only once in the vertex list, and each tetrahedral cell has four vertices represented by four pointers (i.e., indices) to the corresponding entries in the vertex list. While this is a very compact representation, it is not suitable for out-of-core access, as random accesses in disk by following pointers to vertex list are very inefficient.

In the meta-cell technique [Chiang et al. 1998], we try to optimize both the disk-access cost and the disk-space requirement. We utilize the *spatial coherence* of the data by clustering spatially neighboring cells together to form a meta-cell. Each meta-cell has self-contained information and is always read as a whole from disk to main memory. This enables us to use a compact representation for each meta-cell, namely a local vertex list and a local cell list, where each cell has four pointers (indices) to the local vertex list.

The meta-cells are constructed as follows. First, we use an external sorting to sort all vertices by their x-values, and partition them evenly into k chunks, where k is a parameter that can be adjusted. Then, for each of the k chunks, we externally sort the vertices by the y-values and again partition them evenly into k chunks. Finally, we repeat for the z-values. We now have k^3 chunks, each having about the same number of vertices. Each final chunk corresponds to a meta-cell, whose vertices are the vertices of the chunk. A cell with all vertices in the same meta-cell is assigned to that meta-cell; if the vertices belong to different meta-cells, then a voting scheme is used, and the missing vertices are duplicated into the meta-cell that owns this cell. We then construct the local vertex list and the local cell list for each meta-cell. The meta-cells may differ dramatically in volume, but have essentially the same storage size.

To extend the meta-cell technique for time-varying fields, an intuitive approach would be to extend each vertex entry from a record

of x-, y-, z-coordinates and a scalar value f to a record of the same coordinates plus scalar values $f_1, \cdots, f_r$ for all time steps at that vertex. While this works in essentially the same way, it is not the most efficient way. Consider an isosurface query (q,t). For each active meta-cell, we only need to access its local cell and vertex *coordinate* information, plus the scalar values of all vertices *at time step t only*. We achieve this by organizing a meta-cell as a local cell list, a local vertex list containing *only coordinates* for each vertex, and a scalar-value list organized as scalar values of all the local vertices at time step 0, appearing in the same order as the corresponding vertices in the vertex list, then the scalar values of all local vertices at time step 1 in the same order, and so on. In this way, we no longer need to access the entire meta-cell, and the query time as well as the main-memory requirement at run time are both *independent* of the number of time steps in the data. This provides an efficient access approach for time-varying meta-cells.

As for meta-cell construction, observe that various construction steps involve the interplay between the cells and the vertices, and require sortings for the cell and vertex entries. If each vertex entry carries all its scalar values, then we have to sort long records in the sorting process, which is very time consuming. The idea is to decouple the vertex scalar values from each vertex record, tagging a vertex ID v_{id} to the scalar-value record $(f_1, \cdots, f_r)$ for each vertex, and sort each vertex entry containing only its coordinates. At the end, after assigning/duplicating vertices to meta-cells, we produce a file consisting of the tuples (m_{id}, v_{id}), meaning that vertex v_{id} is assigned or duplicated to the meta-cell m_{id}. (If vertex v_{id} is duplicated, there are multiple entries of (m_{id}, v_{id}) with the same v_{id} but different m_{id}, each for a different meta-cell the vertex is put into.) We then replace each v_{id} in (m_{id}, v_{id}) with the corresponding scalar-value record $(v_{id}, f_1, \cdots, f_r)$. While this replacement step is easy in main memory by pointer de-referencing, it is non-trivial in out-of-core computation as we need to avoid random accesses by following pointers. We carry out this step as follows. The scalar-value records $(v_{id}, f_1, \cdots, f_r)$ are already in sorted order by increasing v_{id}. We externally sort the tuples (m_{id}, v_{id}) by increasing v_{id}. Note that the records in the two files now appear in the *same* increasing order of v_{id}. We then linearly scan the two files simultaneously to carry out the replacement step easily. Finally, we perform a global sorting on the resulting file of tuples $(m_{id}, v_{id}, f_1, \cdots, f_r)$, using m_{id} as the first key and v_{id} as the second key. This will put all scalar-value records of the same meta-cell together, ordered by the vertex ID within each meta-cell. In this way, the meta-cell computation process has a minimum dependency on the number of time steps in the data, and can be performed much more efficiently.

4 Results

We have implemented our technique in C/C++ and ran our experiments on an HP Visualize XL PC with dual 1GHz Pentium III CPUs, 2GB RAM, and an fx10 graphics card, running under RedHat Linux 6.2. An interesting feature of the Linux operation system is that we can change the RAM size at the system boot time. For example, the command "linux = 512M" makes the RAM size as if there were only 512MB, and the virtual memory feature makes the system to swap when the main memory usage exceeds 512MB, even though the physical RAM size is 2GB. This feature is used to test the scalability of our technique with respect to different main memory sizes available.

The datasets we tested are listed in Table 1; they are given as tetrahedral meshes. The Tpost and TL datasets have the same vertices and cells, with different numbers of time steps; similarly for the Vorts5 and Vorts9 datasets. Note that Vorts5 and Vorts9 have more than 10 million cells. Representative isosurfaces generated by our program are shown in Figure 4.

For the purpose of comparisons, we have also implemented

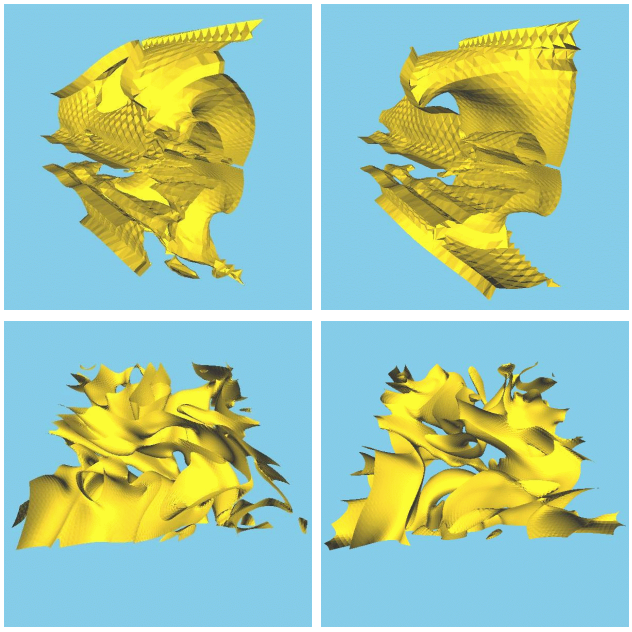

Figure 4: Representative isosurfaces. Top row: isosurfaces from the Tpost dataset with isovalue $q = 1.0$ at two different time steps. Bottom row: isosurfaces from the Vorts9 dataset with isovalue $q = 2.1632$ at two different time steps.

a main-memory isosurface extraction approach for time-varying datasets. After reading and storing the input data into cell and vertex tables in main memory, this method proceeds to create a main-memory interval tree for each time step; this is a direct extension of the query-optimal main-memory isosurface algorithm [Cignoni et al. 1997] to handle time-varying data. We refer to this program as MMint, and the program of our new out-of-core approach as OOC. When there is enough main memory to hold both input data and the interval trees, MMint should give the fastest query performance, even faster than the THI tree algorithm [Shen 1998], since MMint finds the exact set S of active cells while the THI tree method finds a *superset* of S. Of course, the THI tree algorithm requires much less main memory space, but it is still a main-memory algorithm, and we would expect it to exhibit a similar behavior to that of MMint when the main memory limitation is hit.

In Table 1, we show the statistical results of OOC and MMint. We observe that different values of the Manhattan distance threshold L give different reduction rates in the process of meta-interval collapsing; however the effect also depends on different temporal coherences in different datasets. For example, while setting L to 0.001 results in a reduction rate of about 90% in Tpost, setting it to 0.01 gives only very small reduction rates (0.03%-0.04%) in Vorts5 and Vorts9. With $L = 1.2$ in TL, the reduction rate is about 99%.

We see in Table 1 that the meta-cell file is much larger than the files of all BBIO trees and of the time tree. As expected, the meta-cell file essentially is the data file, while the BBIO trees and the time tree serve as indexing structures. The size of the time tree is particularly small, as we only have no more than 100 time steps. For large simulation applications where there are hundreds of thousands of time steps, we could expect the size of the time tree to become much larger, and our technique using the cache-oblivious approach is I/O-optimal and scalable for any number of time steps.

We note that the size increase in TL (208%) is much larger than the size increase in Tpost (103%), even though they have the same number of meta-cells (obtained by using the same parameter k for

Data	Tpost	TL	Vorts5	Vorts9
# cells	615195	615195	10241915	10241915
# vertices	131072	131072	2097152	2097152
# time steps	10	100	5	9
original size	16.7MB	64MB	231MB	265MB
# meta-cells	8419	8419	32768	32768
max # cells	98	98	320	320
max # vert.	100	100	126	126
L	0.001	1.2	0.01	0.01
org # m-intvl	92436	924940	163840	294912
new # m-intvl	9239	8816	163783	294786
meta-cells	33.4MB	196MB	294MB	358MB
BBIO trees	0.45MB	0.4MB	4.4MB	7.8MB
time tree	0.3KB	3.2KB	0.14KB	0.27KB
total	33.9MB	197MB	298MB	366MB
increase	103%	208%	29%	38%
MM (query)	4 bk	4 bk	5 bk	5 bk
MMint tables	16.7MB	64MB	231MB	265MB
MMint trees	148MB	1.48GB	1.23GB	2.21GB
MMint total	165MB	1.54GB	1.46GB	2.48GB

Table 1: Experimental results of the statistics of the algorithms. We list the number of cells, vertices, and time steps of the datasets, as well as the size of the original binary files. We then show the resulting number of meta-cells, maximum numbers of cells and of vertices in a meta-cell, followed by the Manhattan distance threshold L used in meta-interval collapsing, and the total number of meta-intervals *before* ("org") and *after* ("new") the collapsing. Next, we show the sizes of the overall meta-cells, of the overall BBIO trees, and of the time tree, followed by the total size of these data structures stored in disk, and the percentage of size increase compared to the original data. We also list the main memory requirement of OOC to perform isosurface queries in number of blocks (4KB each), excluding the space to hold the isosurface triangles. Finally, we show the main memory requirement of MMint for the cell plus the vertex tables, for all the interval trees, and their total size, which does not include the space to hold the isosurface triangles.

the number of partitions; see Section 3.3). Since these two datasets have the same vertices and cells and only differ in the scalar values (10 v.s. 100 time steps), the resulting meta-cell structures are the same. Recall that during the meta-cell computation, some cells lying on the boundary of different meta-cells cause the cell vertices to be duplicated (in order to make each meta-cell self-contained). Although exactly the same vertices are duplicated, each such vertex in Tpost causes its 10 time-step scalar values to be duplicated while in TL 100 time-step scalar values are duplicated, resulting in a much larger size-increase factor in TL. This shows that the vertex duplication has a bigger impact to the size increase for time-varying data, especially for large number of time steps. The size increases in Vorts5 and Vorts9, on the other hand, are much smaller (29% and 38%), since there are only 5 and 9 time steps but the geometry is much more complicated (more than 10 million cells), and thus the scalar values account for only a small portion of the data.

In the last part of Table 1, we show the main memory requirement of OOC and MMint in finding active cells during isosurface queries. As mentioned in Section 3.1, OOC only needs 3 pages of disk block size to traverse both the time tree and the current BBIO tree, plus the space to hold the maximum-size meta-cell. As described in Section 3.3, we only need to access and accommodate the scalar values of *one* time step for each meta-cell searched. For Tpost and TL one block (of size 4KB) is enough for holding a meta-cell, and for Vorts5 and Vorts9 two blocks are enough. Therefore an overall of 4–5 blocks in main memory are enough for OOC. On the other hand, we see that MMint needs a huge amount of main memory, ranging from 165MB to 2.48GB. Notice however that each interval tree for a single time step needs about the same amount of space as the input data, which, as a main-memory algorithm, is reasonable for a steady-state data.

Next we show in Table 2 the preprocessing times of OOC and MMint, running under the RAM-size settings of 2GB, 512MB, and 55MB. As can be seen, the preprocessing time of OOC was mainly spent on the construction of the meta-cells. Therefore, for Vorts5 and Vorts9 where the geometry is very complicated, OOC was impacted more. On the other hand, MMint mainly spent its preprocessing time on building the interval trees, and hence a large number of time steps such as that in TL also had a considerable impact on MMint. This explains why for TL under 2GB (where everything can fit in main memory) MMint still ran slower than OOC (498.2 seconds vs. 320.9 seconds). Also, for Vorts5 under 512MB, although swappings occurred for MMint, it still ran faster than OOC since there is only 5 time steps but the geometry is very complicated. Under 512MB, OOC already exhibits a clear advantage over MMint for "larger" datasets such as TL and Vorts9. As we reduce the RAM size to 55MB, while MMint suffers from thrashing (e.g., 4456.6 seconds for Vorts9), the OOC preprocessing time is essentially unchanged (e.g., around 1326 seconds for Vorts9 under all three RAM sizes), showing a nice property of running times independent of the main memory size available, as desired.

Data	Tpost	TL	Vorts5	Vorts9
meta-cells 2G	78.8s	316.9s	1129.3s	1322.7s
time+BBIO 2G	0.4s	3.97s	2.29s	4.02s
total OOC 2G	79.2s	320.9s	1131.6s	1326.7s
MMint 2G	49.9s	498.2s	556s	1000.7s
meta-cells 512M	78.8s	319.3s	1129.8s	1325.7s
time+BBIO 512M	0.43s	3.83s	2.2s	3.97s
total OOC 512M	79.2s	323.1s	1132.0s	1329.7s
MMint 512M	50.2s	529.8s	820.8s	1505.6s
meta-cells 55M	78.7s	317.0s	1129.1s	1321.3s
time+BBIO 55M	0.47s	3.88s	2.35s	4.19s
total OOC 55M	79.2s	320.9s	1131.5s	1325.5s
MMint 55M	79.9s	1080.1s	1627.3s	4456.6s

Table 2: Experimental results of the preprocessing phase. We list the preprocessing times (in seconds) of OOC and of MMint for running under various RAM-size settings. Each OOC preprocessing time is also broken into the times for constructing the meta-cells and for constructing the time tree plus the BBIO trees.

In Table 3, we show the total running times of performing 20 isosurface queries on each dataset, under 2GB, 512MB, and 55MB of RAM. Under 2GB, MMint performs faster than OOC but OOC is still reasonably fast. As we reduce the RAM size, OOC again shows a clear advantage over MMint: under 512MB, OOC shows a speed-up factor of about 9.9 times for Vorts9 (87.47 seconds vs. 868.96 seconds), and under 55MB, with a speed-up factor of about 281, OOC improves the running time for Vorts5 from 65736.3 seconds (about 18.26 hours) to 233.77 seconds (about 3.9 minutes)!

It is interesting to see that although MMint requires a smaller amount of main memory for Vorts5 than for TL (1.46GB vs. 1.54GB), the thrashing for Vorts5 is much worse. This is because the resulting isosurfaces in Vorts5 typically have about 30 times as many triangles as those in TL. This indicates that for large datasets and isosurfaces, in order to achieve efficient isosurface extraction, it is not enough to just reduce the size of the indexing structure and still keep the data in main memory, since the major source of thrashing comes from randomly accessing the active cells in the data tables. Thus, we need to have an out-of-core technique to completely

Data	Tpost	TL	Vorts5	Vorts9
max # iso tris	40201	40201	1276348	1276348
ave # iso tris	22428.9	22428.9	605965.8	605965.8
OOC 2G	2.86s	3.04s	86.14s	86.66s
MMint 2G	0.92s	0.97s	26.51s	29.11s
OOC 512M	2.81s	2.99s	84.74s	87.47s
MMint 512M	0.95s	2.93s	321.09s	868.96s
OOC 55M	3.1s	30.38s	233.77s	258.22s
MMint 55M	8.65s	185.35s	65736.3s	N/A

Table 3: Experimental results of the query phase. For each dataset, we show the total running times (in seconds) of performing 20 isosurface queries under various RAM-size settings. Each pair of datasets (Tpost and TL, and Vorts5 and Vorts9) have the same set of resulting isosurfaces. The running times do not include the isosurface rendering time. We also list the maximum and average numbers of triangles in the resulting isosurfaces.

avoid storing the data and the indexing structures in main memory. We remark that for Vorts5 and Vorts9 with large numbers (as large as 1276348) of isosurface triangles, the 55MB setting seems to have some impact on OOC, not on the search process but rather on the ability to hold all isosurface triangles for the rendering purpose. Still, by not holding the dataset in main memory and only fetching the small necessary portions of the data from disk, OOC can perform quite efficiently for the Vorts5 and Vorts9 datasets of over 10 million cells on a PC with only 55MB of main memory.

5 Conclusions

We have presented a novel out-of-core isosurface extraction algorithm for time-varying fields over irregular grids. Our algorithm integrates several interesting ideas such as the *time tree* data structure, the *meta-interval collapsing scheme*, the *buffer technique*, the *cache-oblivious technique*, and the *meta-cell technique* for time-varying fields.

Our experiments show that for large datasets, main-memory algorithms should be avoided, even for the query-optimal ones such as MMint extended from [Cignoni et al. 1997]. First, the indexing structures themselves can be the major main-memory-space overhead, especially for large time steps. More importantly, for large datasets and isosurfaces, it is not enough to just reduce the size of the indexing structures, as the major source of thrashing can come from randomly accessing the active cells of the input data stored in main memory. Our technique, on the other hand, by exploring the spatial- and temporal-coherences of the data as well as putting all data and indexing structures in disk, completely avoids the main-memory limitation, and achieves a scalable performance in both preprocessing and run-time phases, as well as a huge speed-up in isosurface queries.

In conclusion, our work of developing out-of-core techniques for time-varying datasets de-couples the size of a visualization task from the amount of computational resources available, and indicates a promising direction towards resolving the big challenges posed by large-scale time-varying data visualization problems.

Acknowledgements

We thank Han-Wei Shen for providing his Marching Tetrahedra code and the test datasets used in this work.

References

ARGE, L., AND VITTER, J. S. 1996. Optimal interval management in external memory. In *Proc. IEEE Foundations of Comp. Sci.*, 560–569.

BAJAJ, C. L., PASCUCCI, V., AND SCHIKORE, D. R. 1996. Fast isocontouring for improved interactivity. In *Proc. Volume Visualization Sympos.*, 39–46.

BAJAJ, C., PASCUCCI, V., D.THOMPSON, AND ZHANG, X. 1999. Parallel accelerated isocontouring for out-of-core visualization. In *Proc. IEEE Parallel Visualization and Graphics Sympos.*, 97–104.

BENDER, M. A., DUAN, Z., IACONO, J., AND WU, J. 2002. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. ACM-SIAM Sympos. on Discrete Algorithms*, 29–38.

BORDOLOI, U., AND SHEN, H.-W. 2003. Space efficient fast isosurface extraction for large datasets. In *Proc. IEEE Visualization*.

BRYSON, S., KENWRIGHT, D., AND COX, M. 1997. *Exploring gigabyte datasets in real time: algorithms, data Management, and time-critical Design.* ACM SIGGRAPH course note.

CHIANG, Y.-J., AND SILVA, C. T. 1997. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, 293–300.

CHIANG, Y.-J., SILVA, C. T., AND SCHROEDER, W. J. 1998. Interactive out-of-core isosurface extraction. In *Proc. IEEE Visualization*, 167–174.

CHIANG, Y.-J., FARIAS, R., SILVA, C., AND WEI, B. 2001. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *Proc. IEEE Sympos. on Parallel and Large-Data Visualization and Graphics*, 59–66.

CIGNONI, P., MARINO, P., MONTANI, C., PUPPO, E., AND SCOPIGNO, R. 1997. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (April - June), 158–170.

EDELSBRUNNER, H. 1983. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math. 13*, 209–219.

ITOH, T., AND KOYAMADA, K. 1995. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics 1*, 4 (Dec.), 319–327.

LIVNAT, Y., SHEN, H.-W., AND JOHNSON, C. 1996. A near optimal isosurface extraction algorithm using span space. *IEEE Transactions on Visualization and Computer Graphics 2*, 1 (Mar.), 73–84.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, M. C. Stone, Ed., vol. 21, 163–169.

SHEN, H.-W., AND JOHNSON, C. 1995. Sweeping simplices: A fast iso-surface extraction algorithm for unstructured grids. In *Proc. IEEE Visualization*, 143–150.

SHEN, H., HANSEN, C. D., LIVNAT, Y., AND JOHNSON, C. R. 1996. Isosurfacing in span space with utmost efficiency (ISSUE). In *Proc. IEEE Visualization*.

SHEN, H.-W. 1998. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In *Proc. IEEE Visualization*, 159–166.

SILVA, C., CHIANG, Y.-J., EL-SANA, J., AND LINDSTROM, P., 2002. Out-of-core algorithms for scientific visualization and computer graphics. Tutorial Course Notes, IEEE Visualization. `http://cis.poly.edu/chiang/Vis02-tutorial4.pdf`.

SUTTON, P. M., AND HANSEN, C. D. 2000. Accelerated isosurface extraction in time-varying fields. *IEEE Transactions on Visualization and Computer Graphics 6*, 2 (April - June), 98–107.

VAN KREVELD, M., VAN OOSTRUM, R., BAJAJ, C. L., PASCUCCI, V., AND SCHIKORE, D. R. 1997. Contour trees and small seed sets for isosurface traversal. In *Proc. ACM Symp. on Comput. Geom.*, 212–220.

VITTER, J. S. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys 33*, 2, 209–271.

WILHELMS, J., AND GELDER, A. V. 1990. Octrees for faster isosurface generation. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, vol. 24, 57–62.