Isosurface Extraction in Large Scientific Visualization Applications Using the I/O-filter Technique

(Extended Abstract)

Yi-Jen Chiang* Cláudio T. Silva†

November 19, 1997

Abstract

For large scientific visualization applications, it is hopeless to hold the entire datasets in main memory. Previously, we proposed the *I/O-filter* technique, which is the first I/O-optimal method for the problem of *isosurface extraction* in scientific visualization. *I/O-filter* works by indexing and reorganizing the datasets in disk, so that isosurface can be extracted with a very small amount of disk I/O's. The main advantage of this approach is that datasets much larger than main memory can be visualized very efficiently, possibly even on low-end machines. The original *I/O-filter* technique uses the I/O-optimal interval tree of Arge and Vitter as the indexing data structure, together with the isosurface engine from Vtk (one of the currently best visualization packages). The main shortcoming of this approach was the overheads of the disk scratch space and the preprocessing time necessary to build the data structure, and of the disk space needed to hold the data structure.

In this paper, we improve the first version of *I/O-filter* by reducing its data structure size, the disk scratch space and the preprocessing time, while keeping the isosurface query time the same. We achieve the improvements through a new implementation of *I/O-filter*, by replacing the interval tree with the *metablock tree* of Kanellakis *et al.*, which is also I/O-optimal for query and space. In the process, we propose two simple preprocessing algorithms for static metablock tree that are I/O-optimal. We give the first implementation of metablock tree, and compare the practical performance between metablock tree and interval tree under the same framework of *I/O-filter* with real-world test data. Our experiments provide detailed quantitative evaluation of the implementations of the two trees, and lead to new insights to their properties. As mentioned, the metablock-tree implementation results in performance improvements of the *I/O-filter* method. Moreover, for our test datasets larger than main memory, the isosurface queries in both implementations of *I/O-filter* are about two orders of magnitude faster than those in Vtk's original implementation, showing that *I/O-filter* is an excellent isosurface technique for large scientific visualization applications.

Keywords: Isosurface Extraction, Marching Cubes, I/O-Efficient Computation, Metablock Tree, Interval Tree, Experimentation, Scientific Visualization, Computer Graphics.

*Department of Applied Mathematics and Statistics, SUNY Stony Brook, NY 11794-3600; yjc@ams.sunysb.edu. Supported in part by NSF Grant DMS-9312098.

†Department of Applied Mathematics and Statistics, SUNY Stony Brook, NY 11794-3600; csilva@ams.sunysb.edu. Partially supported by Sandia National Labs and the Dept. of Energy Mathematics, Information, and Computer Science Office, and by the National Science Foundation (NSF), grant CDA-9626370.

1 Introduction

The field of computer graphics can be roughly classified into two subfields: *surface graphics*, in which objects are defined by surfaces, and *volume graphics* [21, 22], in which objects are given by datasets consisting of 3D sample points over their volume. In volume graphics, objects are usually modeled as *fuzzy* entities. This representation leads to greater freedom, and also makes it possible to visualize the *interior* of an object. Notice that this is harder for traditional surface-graphics objects. Since the dataset consists of points sampling the entire volume rather than just vertices defining the surfaces, typical volume datasets are huge. This makes volume visualization an ideal application domain for I/O techniques.

Input/Output (I/O) communication between fast internal memory and slower external memory is the major bottleneck in many large-scale applications. Algorithms specifically designed to reduce the I/O bottleneck are called *external-memory* algorithms. The issue of this I/O bottleneck is becoming more and more important, since problem sizes of applications are getting larger and larger, and technological advances are increasing CPU speeds at an annual rate of 40–60% while disk transfer rates are only increasing by 7–10% annually [29]. Due to this increasing importance, considerable attention has been given to the development of external-memory algorithms and data structures in recent years. Most of the developed techniques, however, are shown to be efficient only *in theory*, and their performance *in practice* is yet to be evaluated. Also, system issues, usually very important for the actual performance of the algorithms, are often left untouched. In particular, the question of how much impact these I/O techniques can make on *real-world applications* is yet to be answered.

In this paper, we consider an I/O technique for one of the most important problems in volume graphics: *isosurface extraction* in scientific visualization. Isosurface extraction represents one of the most effective and powerful techniques for the investigation of volume datasets. It has been used extensively, particularly in visualization [24, 26], simplification [18], and implicit modeling [30]. Isosurfaces also play an important role in other areas of science such as biology, medicine, chemistry, computational fluid dynamics, etc., since they can be used to study and perform detailed measurements of properties of the datasets. In fact, nearly all visualization packages include an isosurface extraction component. Its widespread use makes efficient isosurface extraction a very important problem.

The problem of isosurface extraction can be stated as follows. A *scalar volume dataset* consists of tuples $(\mathbf{x}, \mathcal{F}(\mathbf{x}))$, where $\mathbf{x}$ is a 3D point and $\mathcal{F}$ is a scalar function defined over 3D points. Given an isovalue (a scalar value) q, to extract the isosurface of q is to compute and display isosurface $C(q) = \{\mathbf{x} | \mathcal{F}(\mathbf{x}) = q\}$. Typical isosurfaces (generated by our code) are shown in Fig. 6, where the Blunt Fin dataset shows an airflow through a flat plate with a blunt fin, and the Combustion Chamber dataset comes from a combustion simulation.

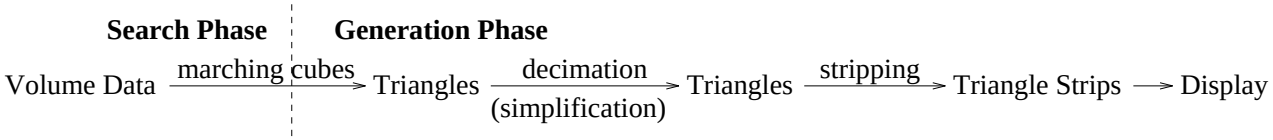

Figure 1: A pipeline of the isosurface extraction process.

The computational process of isosurface extraction can be viewed as consisting of two phases (see Fig. 1). First, in the *search phase*, one finds all *active* cells of the dataset that are intersected by the isosurface. Next, in the *generation phase*, depending on the type of cells, one can apply an algorithm to actually generate the isosurface from those active cells (Marching Cubes [24] is one such algorithm for hexahedral cells). Notice that the search phase is the bottleneck of the entire process, since it searches the 3D dataset and produces 2D data. In fact, letting N be the total number of cells in the

dataset and K the number of active cells, it is estimated that the average value of K is $O(N^{2/3})$ [19]. Therefore an exhaustive scanning of all cells in the search phase is inefficient, and a lot of research efforts have thus focused on developing *output-sensitive* algorithms to speed up the search phase. In the following we use M and B to respectively denote the numbers of cells fitting in the main memory and in a disk block. Each I/O operation reads or writes one disk block.

Previous Related Work

We first briefly review the work on I/O techniques. In addition to early work on sorting and scientific computing [2, 27, 42], recently various researchers have been investigating external-memory algorithms for graphs [1, 12] and for computational geometry [1, 3, 5, 6, 7, 10, 17, 20, 28, 37, 41]. As mentioned before, most of the results are theoretical, and yet the experiments of Chiang [11], Vengroff and Vitter [40], and Arge *et al.* [5] on some of these techniques show that they result in significant improvements over traditional algorithms in practice.

As for isosurface extraction, there is a very rich literature. Here we only briefly review the results that focus on speeding up the search phase. We refer to [23] for an excellent and thorough review. In Marching Cubes [24], all cells in the volume dataset are searched for isosurface intersection, and thus $O(N)$ time is needed. This technique does not require the entire dataset to fit into the main memory, but $\lceil N/B \rceil$ disk reads are necessary. Techniques avoiding exhaustive scanning include using an octree [43], identifying a collection of *seed cells* and performing contour propagation from the seed cells [8, 19, 39], NOISE [23], and other nearly optimal isosurface extraction methods [34, 35]. The first *optimal* isosurface extraction algorithm was given by Cignoni *et al.* [14], based on the following two ideas. First, by producing for each cell an interval, the active-cell searching process is reduced to the following problem of *stabbing queries*: given a set of intervals and a query point q in 1D, report all intervals (and the associated cells) containing q. Secondly, the stabbing queries are solved by the use of an internal-memory interval tree [16]. After an $O(N \log N)$-time preprocessing, active cells for a query q can be found in optimal $O(\log N + K)$ time. This achieves tight theoretical bounds in terms of internal computation.

All the isosurface techniques mentioned above are main-memory algorithms. Except for the exhaustive scanning method of Marching Cubes, all of them require the time and main memory space to read and keep the entire dataset in the main memory, plus additional preprocessing time and main memory space to build the search structure. Unfortunately, for (usually) very large volume datasets, these methods often suffer the problem of not having enough main memory, which can cause a major slow-down of the algorithms due to a large number of page faults.

In [13] we give *I/O-filter*, the first I/O-optimal technique for isosurface extraction. We follow the ideas of Cignoni *et al.* [14], but use I/O-optimal interval tree [7] to solve the stabbing queries. We give the first implementation of the I/O interval tree, and also implement our method as an *I/O filter* for the isosurface extraction routine of Vtk [31, 32] (which is one of the currently best visualization packages) for the case of irregular grids. With this *I/O-filter* technique, only the K active cells are brought into the main memory (via I/O-optimal stabbing queries in disk, using $O(\log_B N + K/B)$ I/O's); this much smaller set of K cells is then treated as an input to Vtk. Thus we *filter out* a large number of unnecessary cells from the original input, without touching the entire dataset in disk. Our experiments show that datasets much larger than main memory can be visualized very efficiently, and in fact the search phase is no longer a bottleneck. Another out-of-core isosurface technique, based on contour propagation from seed cells, is recently given in [9].

Very recently Agarwal *et al.* [1] developed I/O techniques to solve a similar *contour-line (i.e., isoline) extraction* problem in GIS (which was originally solved in [38] by reducing the problem to stabbing

queries and then solving stabbing queries with internal-memory interval tree [16]). In addition to just reporting "active regions", their techniques can also report the contour-line segments *in sorted order* within the same optimal I/O bound, assuming there are too many active regions to fit into the main memory. Notice that under this assumption, our *I/O-filter* method needs additional $O(K)$ I/O's if no further I/O technique is used, which is far from optimal. However, for our particular problem of isosurface extraction, we just assume that all active cells can fit into the main memory and do not explore any I/O techniques beyond the search phase (see Fig. 1). The reason for this is twofold: (1) since there are only $K = O(N^{2/3})$ active cells, in practice K is fairly small compared to N (the largest isosurface in our experiments has only 55205 active cells, occupying 4.2Mb); (2) for visualization purposes, we need to render the isosurface in real time. For this to be possible, we need to use a machine with enough main memory to hold the isosurface. Therefore in practice the active cells have to all fit into the main memory.

Our Results

Our main goal in this paper is to address the deficiencies of the first version of *I/O-filter*, namely the large disk scratch space and the preprocessing time to build the data structure, and the disk space to hold the data structure [13]. We improve the first version of *I/O-filter* by reducing all these overheads, while keeping the isosurface query time the same.

We achieve the performance improvements through a new implementation of *I/O-filter*, by replacing the I/O interval tree with the *metablock tree* of Kanellakis *et al.* [20], which is an external-memory version of priority search tree [25] and is also I/O-optimal for query ($O(\log_B N + \frac{K}{B})$ I/O's) and space ($O(N/B)$ disk blocks). In [20] the main focus is on the dynamic version (it is *insertion-only*) and no preprocessing algorithm for static version is given. We propose two simple preprocessing algorithms achieving optimal $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/O's. We give the first implementation of metablock tree, and compare the practical performance between the implementations of the two trees under the same framework of *I/O-filter* for isosurface extraction. We also compare the performance with that of Vtk's original implementation. We summarize our extensive experimental results in 24 charts and three tables, which provide detailed quantitative evaluation of the methods and also lead to new insights to their properties. For example, we find that for queries the implementation of interval tree in general needs less block reads, but the implementation of metablock tree inherently has a better locality for disk accesses and thus needs less *disk-head movements*; their query performance is therefore about the same. Also, metablock tree reduces the tree construction time by a factor of about 1/2, and the average data structure size in disk reduces from 7.7 times the original dataset size to 7.2 times. In addition, the disk scratch space needed for preprocessing reduces from 16 times the original dataset size to 10 times. All these features improve the practical performance of *I/O-filter*, and the observed behavior is consistent with our analysis on the properties of the trees. The experiments also show that both implementations of *I/O-filter* are about two orders of magnitude faster than Vtk's original implementation when performing isosurface queries on our test datasets larger than main memory. This demonstrates that *I/O-filter* is an excellent isosurface technique for large scientific visualization applications.

We remark that we do not implement the more complicated *corner structure* in either interval tree or metablock tree. The corner structure is needed to achieve the theoretically I/O-optimal queries and the optimal disk space in the interval tree [7], so implementing the corner structure could potentially improve the query and space performance of the interval tree, at the expense of potentially increasing the preprocessing time. The corner structure is not implemented in [13] for simplicity of coding and a potentially faster preprocessing. The corner structure is also needed to achieve the theoretically I/O-optimal queries in the metablock tree, at the expense of increasing both the disk space and the

preprocessing time [20]. Since currently the I/O query time for each tree is already less than the CPU time for generating the isosurface, this I/O query time can be hidden by overlapping it with the CPU time, and thus there is no need to speed up the I/O queries any further. Also, since our main interest is to reduce the preprocessing time and the disk space of *I/O-filter* via the implementation of metablock tree, we deliberately choose not to implement the corner structure because its implementation will increase these overheads in metablock tree. However, from the view point of data-structure experimentation, it is an interesting open question to investigate the effects of the corner structure to the practical performance measures of the two trees.

We summarize the contribution of this work as follows:

- We give the first implementation of metablock tree. The implementation is robust and handles all degenerate cases. In the process, we also propose two simple preprocessing algorithms for static metablock tree that are I/O-optimal.
- We perform experimental studies on the practical performance between the implementations of metablock tree and I/O interval tree, under the same framework of *I/O-filter* for isosurface extraction. Both trees are implemented on top of a collection of our general-purpose I/O functions (which support basic I/O operations and main memory buffer management), so that we can make fair comparisons between them. In addition, the test data are drawn from real-world applications.
- The metablock-tree implementation results in performance improvement of *I/O-filter* for the isosurface extraction problem. While keeping the isosurface query time the same (which outperforms Vtk by orders of magnitude), the preprocessing time is greatly improved, and the data structure size and the disk scratch space needed for preprocessing are both reduced.

2 Metablock Tree

2.1 Data Structure

We first briefly review the metablock tree data structure [20], which is an external-memory version of priority search tree [25]. We use Bf to denote the branching factor of the tree, and recall that B is the number of cells (intervals) that can fit in one disk block. The stabbing query problem is solved in the *dual space*, where each interval $[left, right]$ is mapped to a dual point (x, y) with $x = left$ and $y = right$. Then the query "find intervals $[x, y]$ with $x \leq q \leq y$" amounts to the following *two-sided orthogonal range query* in the dual space: report all dual points (x, y) lying in the intersection of the half planes $x \leq q$ and $y \geq q$. Observe that all intervals $[left, right]$ have $left \leq right$, and thus all dual points lie in the half plane $x \leq y$. Also, the "corner" induced by the two sides of the query is the dual point (q, q), so all query corners lie on the line $x = y$.

Metablock tree stores dual points in the same spirit as priority search tree, but increases the branching factor Bf from 2 to $\Theta(B)$, and also stores $Bf \cdot B$ points in each tree node. The main structure of metablock tree is defined recursively as follows (see Fig. 2(a)): if there are no more than $Bf \cdot B$ points, then all of them are assigned to the current node, which is a leaf; otherwise, the topmost $Bf \cdot B$ points are assigned to the current node, and the remaining points are distributed by their x-coordinates into Bf vertical slabs, each containing the same number of points. Now the Bf subtrees of the current node are just the metablock trees defined on the Bf vertical slabs. The $Bf - 1$ slab boundaries are stored in the current node as keys for deciding which child to go during search. Notice that each internal node has no more than Bf children, and there are Bf blocks of points assigned to it. For each node, the

points assigned to it are stored twice, respectively in two lists in disk of the same size: the *horizontal* list, where the points are horizontally blocked and stored sorted by decreasing y-coordinates, and the *vertical* list, where the points are vertically blocked and stored sorted by increasing x-coordinates. We use unique dataset cell ID's to break a tie. Each node has two pointers to its *horizontal* and *vertical* lists. Also, the "bottom"(i.e., the y-value of the bottommost point) of the *horizontal* list is stored in the node.

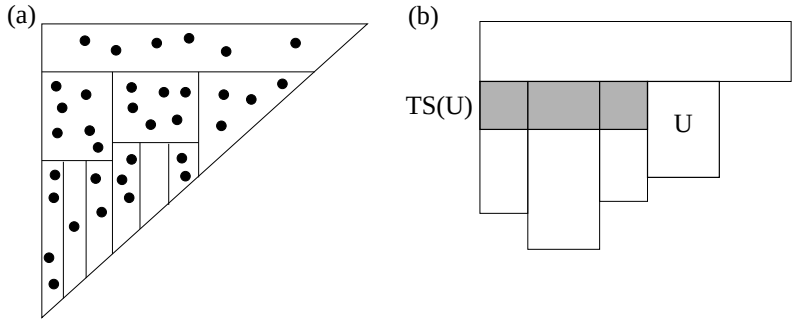

Figure 2: A schematic example of metablock tree: (a) the main structure; (b) the TS list. In (a), $Bf = 3$ and $B = 2$, so each node has up to 6 points assigned to it. We relax the requirement that each vertical slab have the same number of points.

The second piece of organization is the TS list maintained in disk for each node U (see Fig. 2(b)): the list $TS(U)$ has at most Bf blocks, storing the topmost Bf blocks of points from *all left siblings* of U (if there are less than $Bf \cdot B$ points then all of them are stored in $TS(U)$). The points in TS list are horizontally blocked, stored sorted by decreasing y-coordinates. Again each node has a pointer to its TS list, and also stores the "bottom" of the TS list.

The final piece of organization is the *corner structure*. A corner structure can store $t \leq Bf \cdot B$ points in optimal $O(t/B)$ disk blocks, so that a two-sided orthogonal range query can be answered in optimal $O(k/B + 1)$ I/O's, where k is the number of points reported. Assuming all t points can fit in the main memory, a corner structure can be built in optimal $O(t/B)$ I/O's. We refer to [20] for more details. In a metablock tree, for each node U where a query corner can possibly lie, a corner structure is built for the ($\leq Bf \cdot B$) points assigned to U. Since any query corner must lie on line $x = y$, those nodes U are (1) all leaves, and (2) all nodes in the rightmost root-to-leaf path, including the root (see Fig. 2(a)). It is easy to see that the entire metablock tree has height $O(\log_B N)$ (recall that $Bf = \Theta(B)$) and uses optimal $O(N/B)$ blocks of disk space [20]. Also, it can be seen that the corner structures are *additional* structures to metablock tree; we can save some storage space by not implementing the corner structures (at the cost of increasing the worst-case query bound; see Section 2.2).

As we shall see in Section 2.3, we will slightly modify the definition of metablock tree to ease the task of preprocessing, while keeping the bounds of tree height and tree storage space the same.

2.2 Query Algorithm

Now we review the query algorithm given in [20]. Given query value q, we perform the following recursive procedure starting with *meta-query* (q, the root of the metablock tree). Recall that we want to report all dual points lying in $x \leq q$ and $y \geq q$. We maintain the invariant that the current node U being visited always has its x-range containing the vertical line $x = q$.

Procedure *meta-query* (query q, node U)

1. If U contains the corner of q, i.e., the bottom of the *horizontal* list of U is lower than the horizontal line $y = q$, then use the corner structure of U to answer the query and stop.

2. Otherwise ($y(bottom(U) \geq q$), all points of U are above or on the horizontal line $y = q$. Report all points of U that are on or to the left of vertical line $x = q$, using the *vertical* list of U.

3. Find the child U_c (of U) whose x-range contains the vertical line $x = q$. Node U_c will be the next node to be recursively visited by *meta-query*.

4. Before recursively visiting U_c, take care of the left-sibling subtrees of U_c first (points in all these subtrees are on or to the left of vertical line $x = q$, and thus it suffices to just check their heights):
   (a) If the bottom of $TS(U_c)$ is lower than horizontal line $y = q$, then report the points in $TS(U_c)$ that lie inside the query range. Go to step 5.
   (b) Else, for each left sibling W of U_c, repeatedly call procedure *H-report* (query q, node W). (*H-report* is another recursive procedure given below.)

5. Recursively call *meta-query* (query q, node U_c).

H-report is another recursive procedure for which we maintain the invariant that the current node W being visited have all its points lying on or to the left of vertical line $x = q$, and thus we only need to consider the condition $y \geq q$.

Procedure *H-report* (query q, node W)

1. Use the *horizontal* list of W to report all points of W lying on or above horizontal line $y = q$.

2. If the bottom of W is lower than line $y = q$ then stop.
   Otherwise, for each child V of W, repeatedly call *H-report* (query q, node V) recursively.

It can be shown that the queries are performed in optimal $O(\log_B N + \frac{K}{B})$ I/O's [20]. We remark that only one node in the search path would possibly use its corner structure to report its points lying in the query range since there is at most one node containing the query corner (q, q). If we do not implement the corner structure, then step 1 of Procedure *meta-query* can still be performed by checking the *vertical* list of U up to the point where the current point lies to the right of vertical line $x = q$ and reporting all points thus checked with $y \geq q$. This might perform extra Bf I/O's to examine the entire *vertical* list without reporting any point, and hence is not optimal. However, if $K \geq \alpha \cdot (Bf \cdot B)$ for some constant $\alpha < 1$ then this is still worst-case I/O-optimal since we need to perform $\Omega(Bf)$ I/O's to just report the answer.

2.3 Preprocessing Algorithms

Now we present our two simple preprocessing algorithms achieving optimal $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/O's. The first one is based on a paradigm we call *scan and distribute* inspired by the distribution sweep I/O technique [11, 17]. The second one applies the buffer technique, making use of a buffer tree [3]. Both methods relies on a slight modification of the definition of metablock tree.

In the original definition of metablock tree, the vertical slabs for the subtrees of the current node are defined by dividing the *remaining* points not assigned to the current node into Bf groups. This makes the distribution of the points into the slabs more difficult, since in order to assign the topmost

Bf blocks to the current node we have to sort the points by y-values, and yet the slab boundaries (x-values) from the remaining points cannot be directly decided. There is a simple way around it: we first sort all N points by increasing x-values into a fixed set X. Now set X is used to decide the slab boundaries: the root corresponds to the entire x-range of X, and each child of the root corresponds to an x-range spanned by consecutive $|X|/Bf$ points in X, and so on. In this way, the slab boundaries of the entire metablock tree is *pre-fixed*, and the tree height is still $O(\log_B N)$.

Our first method is based on scan and distribute. In the first phase, we sort all points into set X as above and also sort all points by decreasing y-values into set Y. Now the second phase is a recursive procedure. We assign the first *Bf* blocks in set Y to the root (and build its *horizontal* and *vertical* lists), and scan the remaining points to distribute them to the vertical slabs of the root. For each vertical slab we maintain a temporary list, which keeps one block in the main memory as a buffer and the remaining blocks in disk. Each time a point is distribute to a slab, we put that point to the corresponding buffer; when the buffer is full, it is written to the corresponding list in disk. When all points are scanned and distributed, each temporary list has all its points, automatically sorted by decreasing y. Now we build the TS lists for children nodes U_0, $U_1, \cdots$ numbered left to right. Starting from U_1, $TS(U_i)$ is computed by merging two sorted lists in decreasing y and taking the first *Bf* blocks, where the two lists are $TS(U_{i-1})$ and the temporary list for slab $i-1$, both sorted in decreasing y. Note that for the initial condition $TS(U_0) = \emptyset$. (It suffices to consider $TS(U_{i-1})$ to take care of all points in slabs $0, 1, \cdots, i-2$ that can possibly enter $TS(U_i)$, since each TS list contains up to *Bf* blocks of points.) After this, we apply the procedure recursively to each slab. When the current slab contains no more than *Bf* blocks of points, the current node is a leaf and we stop. The corner structures are built for appropriate nodes as the recursive procedure goes. It is easy to see that at each level of the tree, the total number of I/O's for that level is $O(N/B)$, and thus the entire process uses $O(\frac{N}{B}\log_B N)$ I/O's. Using the same technique that turns the nearly-optimal $O(\frac{N}{B}\log_B N)$ bound to optimal in building the static I/O interval tree [4], we can turn this nearly-optimal bound to optimal $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$.

The second method is inspired by the buffer technique used for building static I/O interval tree [4]. We first perform two sortings to obtain sets X and Y as before, and then build a buffer tree whose primary structure is a complete *Bf*-ary tree defined on set X. Notice that this primary structure is basically the same as the main structure of metablock tree. We then use the buffer tree to assign the points belonging to each node: we sequentially "feed" the points from set Y, and perform the usual buffer-tree insertion operation without rebalancing the tree. The main difference is that each node U of the buffer tree now has a "capacity" of *Bf* blocks, i.e., the first *Bf* blocks of points *inserted* to U are assigned to U and are stored into the *horizontal* and *vertical* lists of U; after the first *Bf* blocks, the subsequent points inserted to U are then put into the buffer of U as in the usual buffer tree. After all points of set Y are inserted, we empty all buffers to push down all points to their appropriate nodes. Now we remove all subtrees that contain no point (replacing such subtree rooted at node U with an empty leaf if the parent of U is not already a leaf), and also free all buffers. This buffer tree has now been turned into a metablock tree; we now traverse the tree once again to build the TS lists and also the corner structures. Notice that while corner structures can be built along with the insertions, the TS list of node U cannot be built until all left siblings of U know the points belonging to them; this is why we have to perform additional traversal of the tree after the insertions are complete. Since each insertion has amortized $O(\frac{1}{B}h)$ I/O cost where h is the tree height [3], the entire process takes $O(\frac{N}{B}\log_B N)$ I/O's. Again using the same technique [4], we can turn this nearly-optimal bound to optimal $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$.

3 Implementation

Our code `metaBuild` is used to first *normalize* the input dataset and then build a metablock tree, and code `metaQuery` is used to perform isosurface queries by searching the metablock tree. Notice that the input is a Toff file[1], which has to be *normalized* to de-reference the pointers so that inefficient pointer references in disk can be avoided [13]. After normalization, we obtain *direct cell information* for each cell: the x-, y-, z-values and the scalar values of each of the four vertices of the cell. Similarly, as in [13], `ioBuild` normalizes the input dataset and builds an I/O interval tree, and `ioQuery` uses the interval tree for isosurface queries. Both `metaQuery` and `ioQuery` are linked with Vtk's isosurface code.

In `metaBuild`, we use the *basic* scan and distribute algorithm (i.e., without turning the nearly-optimal bound to optimal) in Section 2.3 to build the metablock tree, for the following reasons. (1) The nearly-optimal bound is almost the same as the optimal one in practice; in particular, the log terms correspond to the tree height, and in all our experiments the height of metablock tree (i.e., the larger log term) is at most 3. (2) As for choosing between the scan and distribute algorithm and the buffer technique, we observe that (a) the two-phase buffer-emptying operation [3] is more complicated than the operations in scan and distribute, and (b) the buffer technique needs an extra pass of traversing the tree to build the TS lists, and an extra work of removing the empty subtrees. Therefore we choose the basic scan and distribute algorithm. For similar reasons, `ioBuild` uses an algorithm based on *scan and distribute* that builds the I/O interval tree in the same I/O bound $O(\frac{N}{B}\log_B N)$ [13]. Also, for reasons described in Section 1, we decide not to implement the corner structure.

Each node of metablock/interval tree is one disk block, whose size decides the value of branching factor Bf and also the value of B (number of cells fitting in one disk block). In our case each disk block is of size 4,096 bytes, thus $Bf = 505$ for metablock tree and $B = 53$. For comparison, in the I/O interval tree Bf is 29 [13]. We remark that in our implementation, only *four* blocks of main memory are needed in searching active cells with the metablock tree (and only *two* blocks are needed for searching with the interval tree).

3.1 Implementing Metablock Tree

Data Structure

We describe the organization of the data structure. We use files `dataset.metaTree`, `dataset.horizontal`, `dataset.vertical`, and `dataset.TS` to hold the metablock tree nodes, all *horizontal* lists, all *vertical* lists, and all TS lists, respectively. Every time we create a new tree node, we allocate the next available block from file `dataset.metaTree` and store the node there. (The root of the tree always starts from position 0 of the file.) Similarly, every time we create a new *horizontal* (resp. *vertical*/TS) list, we allocate the next available consecutive blocks just enough to hold the list, from file `dataset.horizontal` (resp. `dataset.vertical`/`dataset.TS`) and store the list there. We always allocate disk space of size an *integral* number of blocks. Each block in file `dataset.horizontal`, `dataset.vertical` and `dataset.TS` stores up to B dual points (intervals). Each dual point contains a cell ID, four vertices of the cell (x-, y-, z- values and the scalar value of each vertex), and the x- and y-values of the dual point (i.e., the left and right endpoints of the interval associated with the cell, which are the min and max values of the four scalar values).

Now we give the layout of the metablock tree node. Each node contains the following: (1) a flag

[1] A Toff file is analogous to the Geomview "off" file. It has the number of vertices and tetrahedra, followed by a list of the vertices and a list of the tetrahedra, each of which is specified using the vertex locations in the file as an index. See [36].

indicating whether the node is a leaf or an internal node, (2) the number ($\leq Bf \cdot B$) of dual points assigned to the node (which is the number of dual points stored in each of the *horizontal* and *vertical* lists of the node), (3) information about its *horizontal* list, (4) information about its *vertical* list, (5) information about its TS list, (6) the number ($\leq Bf-1$) of keys (slab boundaries) stored in the node, (7) the actual slab boundaries stored, (8) Bf pointers to the starting positions of the children nodes in file `dataset.metaTree`. Notice that items (6)–(8) are maintained only if the node is an internal node. A child pointer (item (8)) is usually a positive integer P, but we sometimes make it negative, $-P$, indicating that the child starts at position P and that the child has 0 points assigned to it. During queries, this child will be needed only for its TS list to take care of its left siblings; the recursive call of Procedure *meta-query* on this child (step 5) can be avoided.

The information about a *horizontal* list include the following: (a) a pointer to the starting position of the list in file `dataset.horizontal`, (b) y-value of the topmost (first) point in the list, (c) y-value of the bottom (last) point in the list. Item (b) is used to speed up the query: if the topmost point is already below horizontal line $y = q$, then no point in the list will be inside the query range and we can avoid any block read of the list. The information about the *vertical* list are similar. The information about the TS list are also similar, with an additional item, i.e., the number of points stored in that TS list.

Preprocessing

There is an interesting issue associated with the implementation of the scan and distribute preprocessing algorithm. Recall from Section 2.3 that for each vertical slab of the current node U we maintain a temporary list to keep all points that are distributed to this slab. Notice that no temporary list is completed until one pass of the scan and distribute process is done. If we use one file for each temporary list, then in the process of scan and distribute all these $Bf = 505$ files have to be open at the same time. Unfortunately, there is a hard limit imposed by the operating system on the number of files a process can open simultaneously (given by the system parameter `OPEN_MAX`; older version of Unix allowed up to 20 open files and this was increased to 64 by many systems).

We solve this problem by using a scratch file `dataset.Y_temp` as a collection of all temporary lists. Let `dataset.Y` be the file for set Y. Observe that by the way we define the slabs, each slab contains at most $\lceil n/Bf \rceil$ blocks of points, where n is the number of blocks in set Y (set X). Therefore the size of file `dataset.Y_temp` is the same as the size of the file for set Y (X). We let the i-th temporary list start from block $i \cdot \lceil n/Bf \rceil$ of file `dataset.Y_temp`, for $i = 0, \cdots, Bf-1$. After the construction of all temporary lists is complete, we copy them to the corresponding locations in file `dataset.Y`, and the scratch file `dataset.Y_temp` is again available for use. Now to perform a recursion on each slab i, we use the portion of file `dataset.Y` staring from block $i \cdot \lceil n/Bf \rceil$ with no more than $\lceil n/Bf \rceil$ blocks as the new Y set to the subproblem.

Handling Degeneracies

Degenerate cases arise when dual points have non-distinct x- or y-values. We use cell ID's to break ties. During tree construction, the key of each slab boundary used to distribute the points includes both the x-value (primary key) and the cell ID (secondary key). In addition, if the (x-value, cell ID) of a point is exactly the slab boundary separating slabs $i-1$ and i, then this point is considered as lying in slab i. In the nodes of metablock tree, we only store the x-values as slab boundaries (keys), without storing the corresponding cell ID's. During query operations, to find the child U_c whose x-range contains q, we find the *rightmost* such child — since we want to report points with $x \leq q$ and $y \geq q$, this ensures that no point satisfying the x-condition of the query is missed. Similarly, to

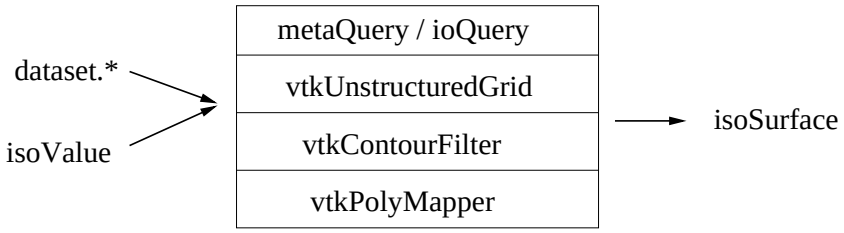

Figure 3: Isosurface extraction phase. Given the four data structure files of the metablock/interval tree and an isovalue, `metaQuery/ioQuery` filters the dataset and passes to Vtk only those active cells of the isosurface. Several Vtk methods are used to generate the isosurface, in particular, vtkUnstructuredGrid, vtkContourFilter, and vtkPolyMapper.

handle the y-condition properly, whenever we consider a *horizontal* list or a TS list, we do not stop if the bottom of the list has the same y-value as q, but instead, we stop only when the bottom is truly lower than horizontal line $y = q$.

3.2 Interfacing with Vtk

A full isosurface extraction pipeline should include several steps in addition to finding active cells (see Fig. 1). In particular, (1) intersection points and triangles have to be computed; (2) triangles need to be decimated [33]; and (3) triangle strips have to be generated. Steps (1)–(3) can be carried out by the existing code in Vtk [32]. Our two pieces of isosurface querying code, `metaQuery` and `ioQuery`, are implemented by linking the respective I/O querying code with Vtk's isosurface generation code, as shown in Fig. 3. Given an isovalue, (1) all the active cells are collected from disk; (2) a vtkUnstructuredGrid is generated; (3) the isosurface is extracted with vtkContourFilter; and (4) the isosurface is saved in a file with vtkPolyMapper. At this point, memory is deallocated. If multiple isosurfaces are needed, this process is repeated. Note that this approach requires double buffering of the active cells during the creation of the vtkUnstructuredGrid data structure. A more sophisticated implementation would be to incorporate the functionality of `metaQuery` (resp. `ioQuery`) inside the Vtk data structures and make the methods I/O aware. This should be possible due to Vtk's pipeline evaluation scheme (see Chapter 4 of [32]).

4 Experimental Results and Analysis

In this section we present experimental results of running the two implementations of *I/O-filter* and also Vtk's native isosurface implementation on real datasets. We have run our experiments on four different datasets shown in Table 1. All of these datasets are tetrahedralized versions of well-known datasets. Our benchmark machine was an off-the-shelf PC: a Pentium Pro, 200MHz with 128M of RAM, and two EIDE Western Digital 2.5Gb hard disk (5400 RPM, 128Kb cache, 12ms seek time). Each disk block size is 4,096 bytes. We ran Linux (kernels 2.0.27, and 2.0.30) on this machine. One interesting property of Linux is that it allows during booting the specification of the exact amount of main memory to use. This allows us to *fake* for the isosurface code a given amount of main memory to use (after this memory is completely used, the system will start to use disk swap space and have page faults).

Name	# of Cells	Original Size	Size after Normalization
Blunt Fin	187K	5.8M	12M
Combustion Chamber	215K	6.8M	13M
Liquid Oxygen Post	513K	16.4M	33M
Delta Wing	1,005K	33.8M	64M

Table 1: A list of the datasets used for testing. After normalization, each cell is 64 bytes long (3D coordinates and scalar fields are represented as floats).

Preprocessing with `metaBuild` and `ioBuild`

Both `metaBuild` and `ioBuild` take the input file, normalize it, and then respectively build a metablock tree and an interval tree. During tree construction, multiple files are created as scratch files for computation, but at the end only four files are left as the final data structure files. See Table 2 for the names of the files for each program.

As noted in [13], the scalability of I/O algorithms is best tested when the dataset size exceeds the main memory size. When we first ran `ioBuild` with 128M of RAM on the datasets, it seemed *too fast* regarding the numbers of I/O reads and writes it was issuing. It turned out that the OS was able to optimize (by *caching*) a lot of those I/O requests, and the CPU was running at nearly 95% of usage. To avoid these side effects of the OS, we lowered the amount of main memory of our system by starting Linux with a "linux mem=16M" command line at kernel boot time. Basically, about 14M of main memory can actually be used by applications (after normalization the input sizes range from 12M to 64M; see Table 1). Also, we allow the programs to allocate only 1024K blocks (4 Mb) of RAM. This is a compile-time parameter in the programs. With this 16M/4M configuration, the actual CPU usage percentage during the execution of `metaBuild` and `ioBuild` was in the range of low teens.

In Table 2 we show all relevant experimental data obtained from running `metaBuild` and `ioBuild` with the 16M/4M configuration. We make the following observations and analysis.

- The numbers of tree nodes are fairly small: 9–39 for metablock tree and 36–183 for interval tree. Interval tree has more nodes because its branching factor is smaller (asymptotically its Bf is taken as $\Theta(\sqrt{B})$ rather than $\Theta(B)$ in metablock tree). Considering the actual values of Bf (505 for metablock tree and 29 for interval tree), we know that the tree height for metablock tree is at most 3, and at most 4 for interval tree. This shows that reducing the branching factor from $\Theta(B)$ to $\Theta(\sqrt{B})$ in interval tree almost dose not increase the tree height. Also, the nearly-optimal $O(\frac{N}{B}\log_B N)$ bound for preprocessing is actually the same as the optimal bound $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ in practice.

- For data structure size, metablock tree (average size increase: 7.2 times the original input size) is somewhat smaller than interval tree (average 7.7 times). This can be seen from the structures of the two trees. While the *left, right* and *multi* lists of any node in interval tree can each have a nonful final block, the *horizontal, vertical*, and TS lists for internal nodes of metablock tree are always of fully Bf blocks (except for the TS list of node U where *all left siblings of* U *are leaves*). Therefore metablock tree potentially has a better disk block usage than interval tree.

- For tree construction time, metablock tree takes only about half the time for interval tree (metablock tree also uses much less numbers of I/O's than interval tree). We analyze this as follows. (1) Although metablock tree performs two sorting (sets X and Y) while interval tree

performs only one sorting (the interval endpoint set sorted in increasing endpoint values), the endpoint set is twice as large as set X or Y, since each interval (i.e., a dual point in set X (or set Y)) is duplicated for its two endpoints. Thus the sorting costs are the same. (2) During the scan and distribute process, metablock tree needs only one temporary file to maintain the temporary lists for vertical slabs, while interval tree needs four files respectively for the temporary lists for children slabs, *left, right*, and *multi* lists [13]. The overhead of maintaining three extra temporary-list files slows down the construction of interval tree. (3) In addition, to avoid blowing up the scratch space by a factor of Bf (= 29) in interval tree, the *multi* lists of a node U are constructed by performing scan and distribute for each of the Bf *left* lists of U [13]. This may be another source of inefficiency. (4) While both the *vertical* list in metablock tree and the *right* list in interval tree need an extra sorting to put them in desired ordering, the *vertical* list is no more than $Bf = 505$ blocks and can always be sorted *internally*, but the *right* list can be arbitrarily long and may need *external* sorting. This again may potentially slow down the interval tree construction.

- For the disk scratch space used during construction, metablock tree needs about 10 times the original size of input dataset, improving over interval tree (about 16 times). As mentioned above, this is due to the three extra temporary-list files needed for interval tree.

In summary, `metaBuild` outperforms `ioBuild` in every aspect, most notably the tree construction time. This can be considerable for really large (*i.e.*, gigabyte- or terabyte-size) datasets. We remark that in large production environments where large disk scratch areas are available, the 10–16 times the original input size of disk scratch space is a minor cost, since the preprocessing is done *once and for all*; after the trees are built in disk, the tree files can be duplicated by just copying. However, the improvement from 16 times to 10 times still increases the applicability of our *I/O-filter* technique. Also, given that disk prices are on the order of 35-40 times lower than main memory prices, the overall cost of a four to eight factor increase in disk space overhead is negligible when compared to a twofold increase in main memory.

Isosurface Extraction with `metaQuery` and `ioQuery`

In the following we use `metaQuery` and `ioQuery` to denote the entire isosurface extraction codes shown in Fig. 3, and `vtkiso` to denote the Vtk-only isosurface code. We ran two batteries of tests, each with different amount of core memory (128M and 32M). Each test consists of calculating 10 isosurfaces with isovalues in the range of the scalar values of each dataset, by using `metaQuery`, `ioQuery`, and `vtkiso`. We did not run X-windows during the isosurface extraction time, and the output of vtkPolyMapper was saved in a file instead. Some representative isosurfaces are shown in Fig. 6. We summarize in Table 3 the *total* running times for the extraction of the 10 isosurfaces using `metaQuery`, `ioQuery`, and `vtkiso` with different amount of main memory. Observe that both `metaQuery` and `ioQuery` have significant advantages over `vtkiso`, especially for large dataset and/or small main memory.

Figures 4 and 5 summarize detailed benchmarks. For each isosurface calculated using `metaQuery` and `ioQuery`, we break the time into four categories: **(1)** I/O time (the bottommost part, shown in red) – This is the time to identify and bring in from disk the active cells of the isosurface. **(2)** Copy Time (the second part from bottom, shown in yellow) – In order to use Vtk's isosurface capabilities, we need to generate a vtkUnstructuredGrid object that contains the active cells just obtained. We refer to the time for this process as "Copy Time". **(3)** Isosurface (the third part from bottom, shown in blue) – This is the time for Vtk's isosurface code to actually generate the isosurface from the active cells. **(4)** File Output (the topmost part, shown in green) – In the end we write to disk a file containing the actual isosurface in Vtk format. As for the performance of `vtkiso`, only items (3)

and (4) are shown in Figs. 4 and 5. Two additional costs are *not shown*: the reading of the dataset, and the generation of the vtkUnstructuredGrid. These two operations are performed only once at the beginning of each batch of 10 isosurface extractions. We show the sum of the two costs as `vtkiso` I/O in Table 3. It is very interesting to see (from Table 3) that the `vtkiso` I/O entries for Post in 32M, and for Delta in both memory sizes, are all much larger than the corresponding entries of `metaQuery` and `ioQuery`. This means that before the time `vtkiso` finishes reading the dataset and generating a vtkUnstructuredGrid, `metaQuery` and `ioQuery` already finished computing all 10 isosurfaces!

Regarding Figs. 4 and 5, we observe the following:
(A) For both `ioQuery` and `metaQuery`, in most cases (1) is smaller than (3), especially as the datasets get larger. This means that the active-cell searching process is not a bottleneck any more; the effect is even more significant for larger datasets. By overlapping the I/O time of (1) with the CPU time of (3), we can further improve the performance so that I/O has no cost at all! Thus there is no need to improve the I/O query time of metablock/interval tree any further. Also, one can see the output-sensitive behavior by noting that when small isosurfaces (or no isosurfaces) are generated, our *I/O-filter* techniques take negligible time.
(B) For both `metaQuery` and `ioQuery`, the I/O times almost do not change with the amount of main memory. As mentioned before, metablock tree only needs four blocks of main memory in searching active cells, and interval tree only needs two blocks. Thus the performance is independent of the size of the main memory available. Similarly, the overall running times of `metaQuery` and `ioQuery` almost do not change with the amount of main memory.

In Table 4, we show detailed performance of querying the metablock tree and interval tree. It is interesting to see that metablock tree usually has more disk reads, and yet the query times are comparable or sometimes even faster. This can be explained by a better locality of disk accesses of metablock tree from the structures of the two trees. In metablock tree, the *horizontal*, *vertical*, and TS lists are always read sequentially during queries, but in interval tree, although the *left* and *right* lists are always read sequentially, the *multi* lists reported inherently cause scattered disk accesses: for query q lying in slab i, all *multi* lists of the multi-slabs spanning slab i are reported; these include multi-slabs $[1,i], [1,i+1], \cdots, [1,Bf-2]$, $[2,i], [2,i+1], \cdots, [2,Bf-2], \cdots, [i,i], [i,i+1], \cdots, [i,Bf-2]$ (see [13]). While $[\ell,\cdot]$'s are in consecutive places of a file and can be sequentially accessed, changing from $[\ell,Bf-2]$ to $[\ell+1,i]$ causes non-sequential disk reads (since $[\ell+1,\ell+1], \cdots, [\ell+1,i-1]$ are skipped). This also leads us to believe that in order to correctly model I/O algorithms, some cost should be associated with *disk-head movements*, since this is one of the major costs involved. This is similar to the parallel computation models (see, for instance, the LogP model of Culler *et al.* [15]), where there is also a charge on latency (or per message overhead), as well as a cost associated with the overall bandwidth used by the algorithm.

5 Conclusions

We give a new implementation of our *I/O-filter* technique for the problem of isosurface extraction, by replacing the I/O interval tree with the first implementation of metablock tree. We provide experimental comparisons of the practical performance between the implementations of the two trees with real-world test data. The metablock-tree implementation also results in performance improvements of *I/O-filter*. In addition, both implementations of *I/O-filter* have significant advantages over Vtk's original implementation. We conclude that *I/O-filter* is an excellent isosurface technique for large scientific visualization applications.

Acknowledgements

We thank the Computational Geometry Laboratory (J. Mitchell, Director) and the Center for Visual Computing (A. Kaufman, Director), for use of the computing resources. The Blunt Fin, the Liquid Oxygen Post, and the Delta Wing datasets are courtesy of NASA. The Combustion Chamber dataset is from Vtk[32]. We thank W. Schroeder, K. Martin, and B. Lorensen for Vtk; the Geometry Center of the University of Minnesota for Geomview; and Linus Torvals, and the Linux community for Linux. Without these powerful tools, it would have been much harder to perform this work.

References

[1] P. K. Agarwal, L. Arge, T. M. Murali, K. R. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 1998 (to appear).

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

[3] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures*, pages 334–345, 1995.

[4] L. Arge. Personal communication, April, 1997.

[5] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 1998 (to appear).

[6] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. European Symp. Algorithms*, pages 295–310, 1995.

[7] L. Arge and J. S. Vitter. Optimal interval management in external memory. In *Proc. IEEE Foundations of Comp. Sci.*, pages 560–569, 1996.

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

[9] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for structured and unstructured meshes in any dimention. In *Proc. Late Breaking Hop Topics*, pages 25–28, 1997.

[10] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. Workshop on Algorithms and Data Structures*, pages 381–392, 1995.

[11] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Comput. Geom.: Theory and Appl.*, (to appear).

[12] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[13] Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, pages 293–300, 1997.

[14] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *1996 Volume Visualization Symposium*, pages 31–38, October 1996.

[15] D. E. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[16] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.

[17] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Comp. Sci.*, pages 714–723, 1993.

[18] T. He, L. Hong, A. Varshney, and S. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–184, June 1996.

[19] T. Itoh and K. 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.

[20] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM Symp. on Principles of Database Sys.*, pages 233–243, 1993.

[21] A. Kaufman. Volume visualization. In J. G. Webster, editor, *Encyclopedia of Electrical and Electronics Engineering*, pages 163–169. Wiley Publishing, 1997.

[22] A. Kaufman, D. Cohen, and R. Yagel. Volume graphics. *IEEE Computer*, 26:51–64, July 1993.

[23] Y. Livnat, H.-W. Shen, and C.R. Johnson. A near optimal isosurface extraction algorithm using span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, March 1996.

[24] 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)*, volume 21, pages 163–169, July 1987.

[25] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14:257–276, 1985.

[26] G. M. Nielson and B. Hamann. The asymptotic decider: Removing the ambiguity in marching cubes. In *Visualization '91*, pages 83–91, 1991.

[27] M. H. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. of the 26th Hawaii Int. Conf. on Systems Sciences*, January 1993.

[28] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. on Principles of Database Sys.*, pages 25–35, 1994.

[29] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[30] W. Schroeder, W. Lorensen, and C. Linthicum. Implicit modeling of swept surfaces and volumes. In *IEEE Visualization '94*, pages 40–45, October 1994.

[31] W. Schroeder, K. Martin, and W. Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *IEEE Visualization '96*, October 1996.

[32] W. Schroeder, K. Martin, and W. Lorensen. *The Visualization Toolkit.* Prentice-Hall, 1996.

[33] W. Schroeder, J. Zarge, and W. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.

[34] H.-W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96*, October 1996.

[35] H.-W. Shen and C.R. Johnson. Sweeping simplices: A fast iso-surface extraction algorithm for unstructured grids. In *IEEE Visualization '95*, pages 143–150, October 1995.

[36] C. T. Silva, J. S. B. Mitchell, and A. E. Kaufman. Fast rendering of irregular grids. In *1996 Volume Visualization Symposium*, pages 15–22, October 1996.

[37] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995.

[38] M. van Kreveld. Efficient methods for isoline extraction from a digital elevation model based on triangulated irregular networks. In *Proc. Intl. Symp. on Spatial Data Handling*, pages 835–847, 1994.

[39] M. van Kreveld, R. van Oostrum, C. L. Bajaj, V. Pascucci, and D. R. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. ACM Symp. on Comput. Geom.*, pages 212–220, 1997.

[40] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995.

[41] D. E. Vengroff and J. S. Vitter. Efficient 3-D range searching in external memory. In *Proc. Annu. ACM Sympos. Theory. Comput.*, pages 192–201, 1996.

[42] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

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

Metablock Tree: `metaBuild`				
	Blunt	Chamber	Post	Delta
metaTree [size (# nodes)]	37K (9)	41K (10)	86K (21)	160K (39)
horizontal [size (# blocks)]	14.5M (3539)	16.6M (4062)	39.7M (9693)	77.8M (18988)
vertical [size (# blocks)]	14.5M (3539)	16.6M (4062)	39.7M (9693)	77.8M (18988)
TS [size (# blocks)]	14.5M (3535)	16.5M (4040)	39.3M (9593)	76.5M (18685)
Total Size	43.5M	49.9M	118.8M	232.2M
Original Size	5.8M	6.8M	16.4M	33.8M
Ratio of Size Increase	7.5	7.3	7.2	6.9
Normalization	348s	465s	920s	1798s
Tree Construction	180s	219s	533s	1071s
Total Time	528s	684s	1453s	2869s
Page Ins	92K	106K	253K	497K
Page Outs	95K	109K	262K	513K
Interval Tree: `ioBuild`				
	Blunt	Chamber	Post	Delta
intTree [size (# nodes)]	303K (74)	147K (36)	238K (58)	750K (183)
left [size (# blocks)]	15.6M (3816)	17.3M (4222)	41.0M (10000)	80.5M (19650)
right [size (# blocks)]	15.6M (3821)	17.3M (4229)	41.0M (10008)	80.5M (19653)
multi [size (# blocks)]	17.6M (4305)	20.0M (4874)	34.6M (8446)	80.6M (19682)
Total Size	49.2M	54.7M	116.8M	242.3M
Original Size	5.8M	6.8M	16.4M	33.8M
Ratio of Size Increase	8.5	8.0	7.1	7.2
Normalization	348s	465s	920s	1798s
Tree Construction	361s	391s	928s	1982s
Total Time	709s	856s	1848s	3780s
Page Ins	103K	115K	270K	570K
Page Outs	109K	123K	286K	605K

Table 2: Statistics for running `metaBuild` (upper half) and `ioBuild` (lower half) in a machine with 16M of main memory and 4M of buffer memory, for the datasets in Table 1. The first four values in each half of the table are the sizes of the four files kept after the preprocessing; the number of nodes of each tree and each file size expressed in terms of number of disk blocks are also shown. "Total size" is the total amount of disk space used after preprocessing. "Normalization" is the time used to convert the input Toff file to a normalized (i.e., de-referenced) file. "Tree Construction" is the actual time used to create the metablock/interval tree data files from the normalized file. "Total Time" is the overall running time of the whole preprocessing. "Page Ins" and "Outs" are the numbers of disk block reads and writes requested to the operating system.

	Blunt	Chamber	Post	Delta
`metaQuery` – 128M	9s	17s	19s	26s
`ioQuery` – 128M	7s	16s	18s	31s
`vtkiso` – 128M	15s	22s	44s	182s
`vtkiso` I/O – 128M	3s	2s	12s	40s
`metaQuery` – 32M	9s	19s	21s	31s
`ioQuery` – 32M	10s	19s	22s	32s
`vtkiso` – 32M	21s	54s	1563s	3188s
`vtkiso` I/O – 32M	8s	28s	123s	249s

Table 3: Overall running times for the extraction of the 10 isosurfaces using `metaQuery`, `ioQuery`, and `vtkiso` with different amount of main memory. These include all the time to read the datasets and write the isosurfaces to files. `vtkiso` I/O is the fractional amount of time of `vtkiso` for reading the dataset and generating a vtkUnstructuredGrid object.

Blunt (187K cells)											
Isosurface ID		1	2	3	4	5	6	7	8	9	10
Active Cells		20	20981	13679	9383	6490	4567	3547	2665	2047	950
`metaQuery`	Page Ins	3	739	635	222	439	296	222	155	87	25
	Time (sec)	0.14	0.59	1.27	0.08	0.31	0.01	0.01	0	0	0
`ioQuery`	Page Ins	5	509	317	243	256	168	91	66	60	44
	Time (sec)	0.07	0.52	1.01	0.46	0.38	0.21	0.09	0.19	0.13	0.16
Chamber (215K cells)											
Isosurface ID		1	2	3	4	5	6	7	8	9	10
Active Cells		32	30733	38385	28552	21745	16608	12457	8698	6553	793
`metaQuery`	Page Ins	3	1003	1131	579	414	387	268	176	129	16
	Time (sec)	0.12	0.78	1.06	0.41	0.18	0.25	0.02	0.01	0.01	0
`ioQuery`	Page Ins	4	632	879	632	489	342	287	205	159	18
	Time (sec)	0.06	0.62	0.67	0.52	0.27	0.21	0.12	0.38	0.05	0.08
Post (513K cells)											
Isosurface ID		1	2	3	4	5	6	7	8	9	10
Active Cells		20	4606	30223	26192	23961	22193	20602	19352	18007	13012
`metaQuery`	Page Ins	3	432	588	497	770	422	422	552	604	307
	Time (sec)	0.05	0.29	0.68	0.72	0.53	0.34	0.21	0.41	0.43	0.43
`ioQuery`	Page Ins	4	179	710	579	614	571	545	499	473	377
	Time (sec)	0.06	0.28	0.87	0.49	0.45	0.42	0.41	0.4	0.44	0.56
Delta (1,005K cells)											
Isosurface ID		1	2	3	4	5	6	7	8	9	10
Active Cells		32	296	1150	1932	5238	24788	36738	55205	32677	8902
`metaQuery`	Page Ins	3	8	506	503	471	617	705	1270	1088	440
	Time (sec)	0.05	0	0.31	0.02	0.02	0.89	0.59	1.24	1.88	0.29
`ioQuery`	Page Ins	6	31	35	46	158	578	888	1171	765	271
	Time (sec)	0.1	0.05	0.05	0.09	0.2	0.67	0.85	1.44	1.43	0.35

Table 4: Searching active cells on metablock tree (using `metaQuery`) and on interval tree (using `ioQuery`) in a machine with 32M of main memory. This shows the performance of the query operations of the two trees. (A "0" entry means "less than 0.01 before rounding".)

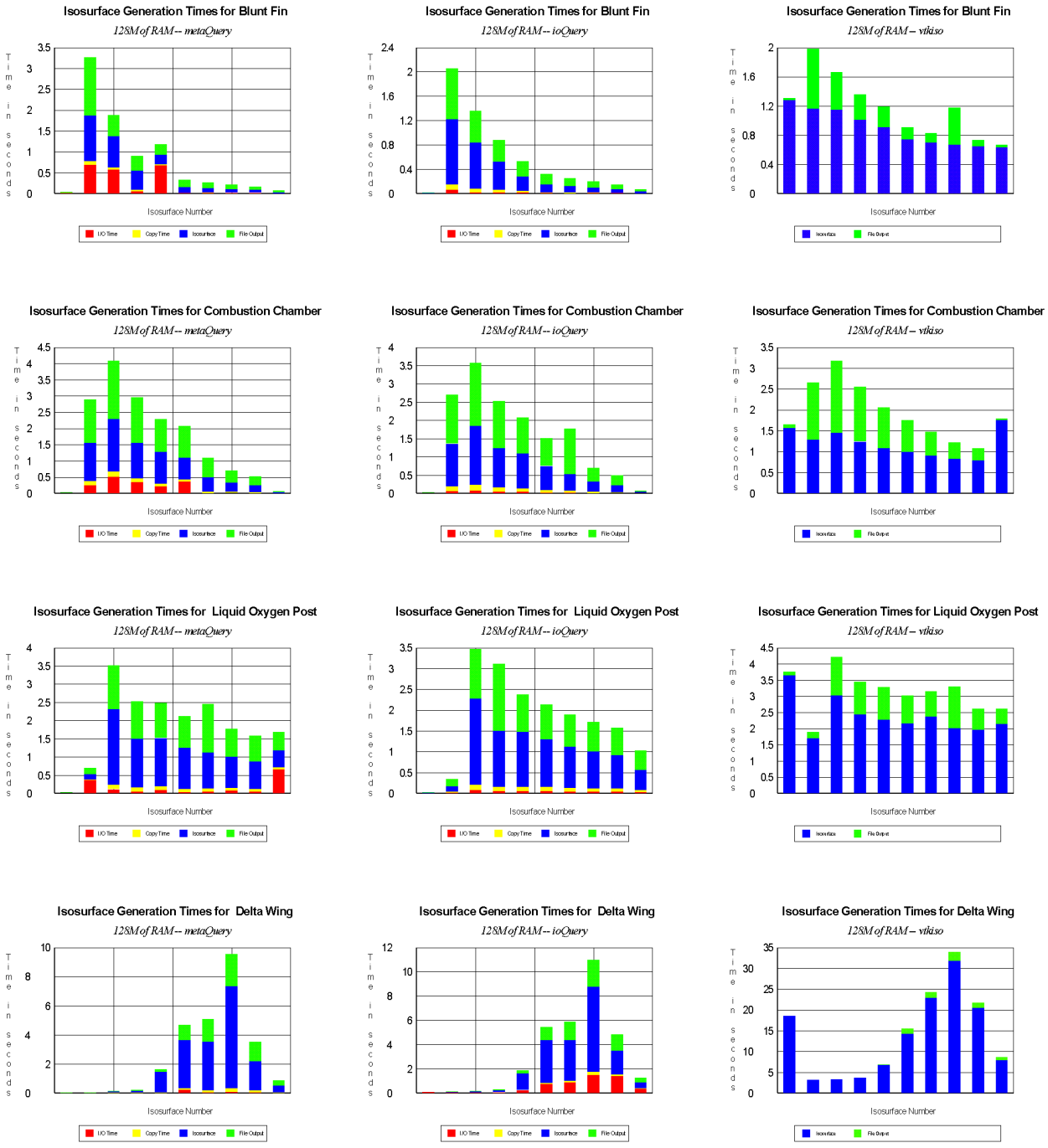

Figure 4: Running times for extracting isosurfaces using `metaQuery` (left column), `ioQuery` (middle column), and `vtkiso` (right column) with 128M of main memory. Note that two costs of `vtkiso` are not shown.

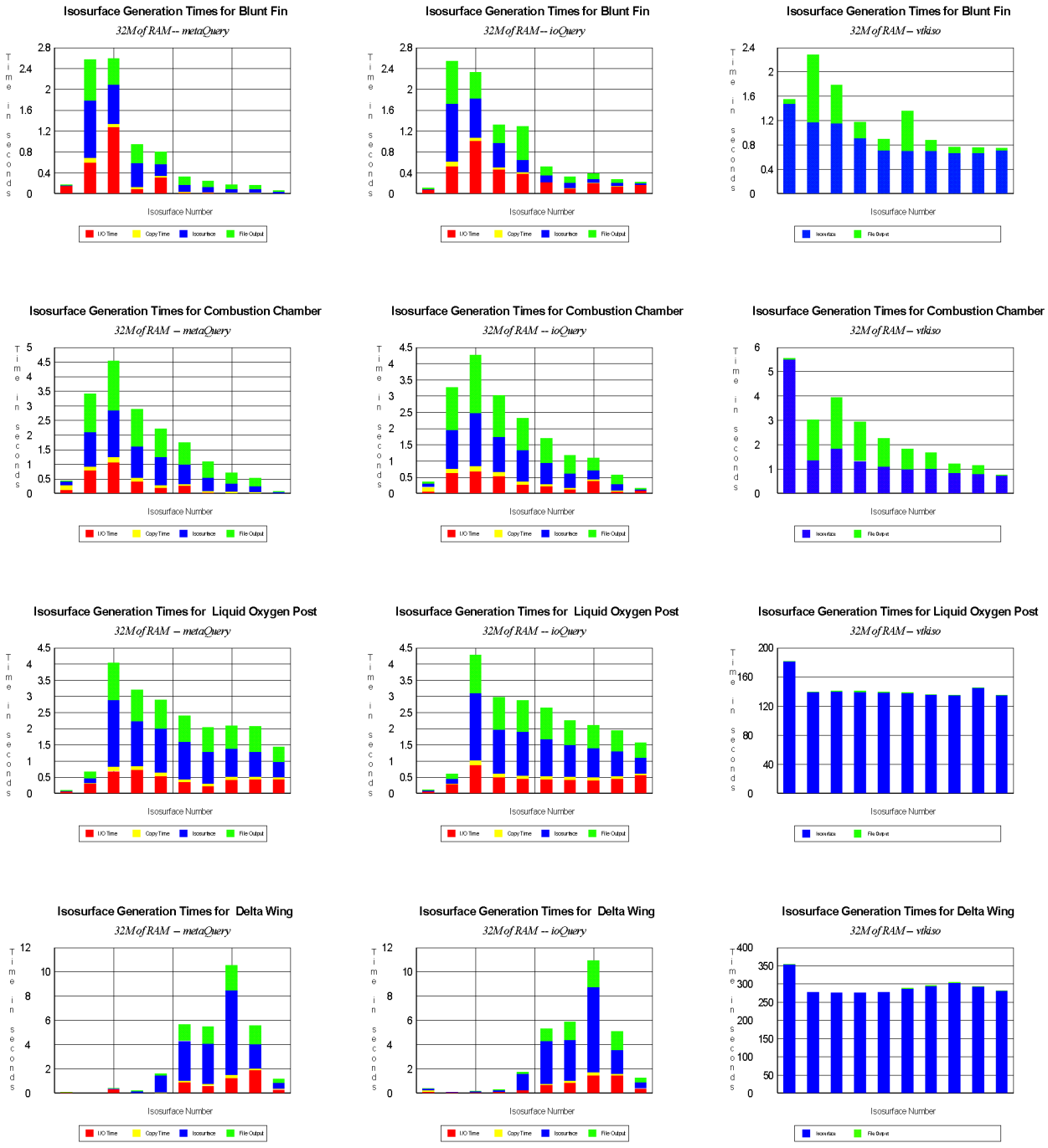

Figure 5: Running times for extracting isosurfaces using `metaQuery` (left column), `ioQuery` (middle column), and `vtkiso` (right column) with 32M of main memory. Note that two costs of `vtkiso` are not shown.

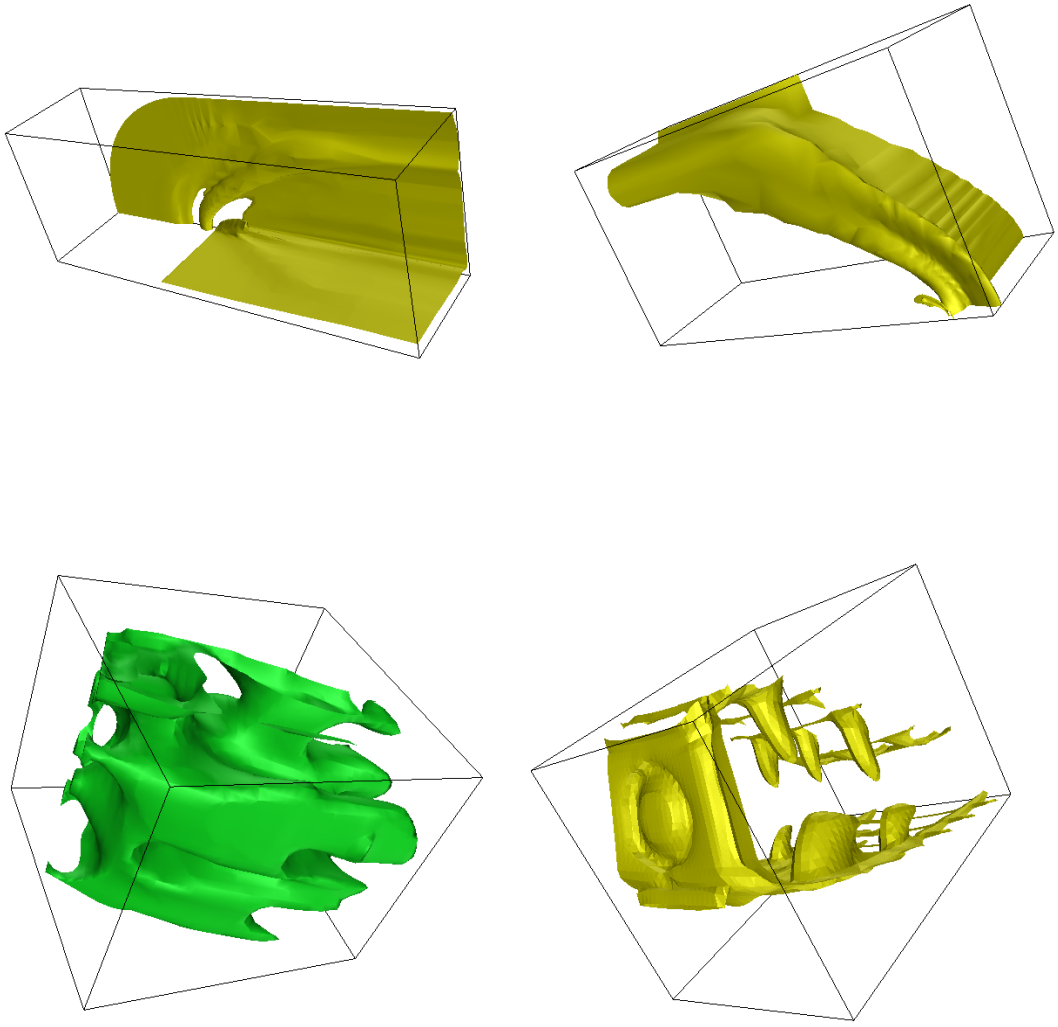

Figure 6: Typical isosurfaces: The upper two are for the Blunt Fin dataset, and those in the bottom are for the Combustion Chamber dataset.