I/O Optimal Isosurface Extraction

(Extended Abstract)

Yi-Jen Chiang* Cláudio T. Silva†

State University of New York at Stony Brook

Abstract

In this paper we give I/O-optimal techniques for the extraction of isosurfaces from volumetric data, by a novel application of the I/O-optimal interval tree of Arge and Vitter. The main idea is to preprocess the dataset *once and for all* to build an efficient search structure in *disk*, and then each time we want to extract an isosurface, we perform an *output-sensitive* query on the search structure to retrieve only those *active* cells that are intersected by the isosurface. During the query operation, only two blocks of main memory space are needed, and only those active cells are brought into the main memory, plus some negligible overhead of disk accesses. This implies that we can efficiently visualize very large datasets on workstations with just enough main memory to hold the *isosurfaces themselves*. The implementation is delicate but not complicated. We give the first implementation of the I/O-optimal interval tree, and also implement our methods as an *I/O filter* for Vtk's isosurface extraction for the case of unstructured grids. We show that, in practice, our algorithms improve the performance of isosurface extraction by speeding up the active-cell searching process so that it is no longer a bottleneck. Moreover, this search time is independent of the main memory available. The practical efficiency of our techniques reflects their theoretical optimality.

1 Introduction

Isosurface extraction represents one of the most effective and powerful techniques for the investigation of volume 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, the extraction of the isosurface of q is to compute and display isosurface $C(q) = \{\mathbf{x} | \mathcal{F}(\mathbf{x}) = q\}$. The computational process of isosurface extraction can be viewed as consisting of two phases. First, in the *search phase*, one finds all cells of the dataset that are intersected by the isosurface; such cells are called *active cells*. 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. Let 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 $O(N^{2/3})$ [9], therefore an exhaustive scanning of all cells in the search phase is found to be inefficient, spending a large portion of time traversing cells that are not active. A lot of research efforts have thus focused on developing *output-sensitive* algorithms to speed up the search phase by avoiding such exhaustive scanning.

Most algorithms developed so far (except for the inefficient exhaustive scanning), however, require the time and main memory space to read and keep the entire dataset in the main memory, plus some additional preprocessing time and main memory space if some structures are built to speed up the search phase. Unfortunately, for really 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. On the other hand, when visualizing isosurfaces, as opposed to volume rendering, only a small portion ($K = O(N^{2/3})$ active cells) of the dataset is ever needed. This seems to indicate that it is not necessary to use time and main memory to load and store the whole volume. If users never had to load a dataset completely but rather only had to store the triangles that defined the isosurface, effective visualization could be performed on low to middle range workstations, or even on PCs in some cases.

In this paper, we present I/O-efficient techniques to resolve the memory issue and to speed up the search phase of isosurface extraction, by a novel use of the I/O-optimal interval tree of [1]. After preprocessing, we can query for active cells using only two blocks of main memory space, and only $O(\log_B N + K/B)$ disk reads, where B is the number of cells that can fit into a disk block (and thus $\lceil K/B \rceil$ disk reads are necessary to report all K active cells). The technique hence acts as an *I/O-filter*, performing only those disk accesses to the dataset that are needed, plus a negligible overhead. The search structure is in fact *I/O-optimal* in space, query, and preprocessing.

Previous Related Work

As mentioned before, isosurface extraction has been the focus of much research. Here we briefly review the results that focus on speeding up the search phase of isosurface extraction by acting as a *filter*, avoiding traversals of cells that are not active. (Please see [11] for an excellent and thorough review.)

In Marching Cubes [12], all cells in the volume dataset are searched for isosurface intersection. Essentially, each time a user runs Marching Cubes, $O(N)$ time is needed. Concerning the main memory issue, this technique does not require the entire dataset to fit into the main memory, but $\lceil N/B \rceil$ disk reads are necessary. Wilhems and Van Gelder [20] propose a method of using an octree to optimize isosurface extraction. This algorithm has worst-case time of $O(K + K \log(N/K))$ (this analysis is presented by Livnat *et al.* [11]) for isosurface queries, once the octree has been built.

Itoh and Kayamada [9] propose a method based on identifying a collection of *seed cells* from which isosurfaces can be propagated by performing local search. Basically, once the seed cells have been identified, they claim to have a nearly $O(N^{2/3})$ expected performance. (Livnat *et al.* [11] estimate the worst-case running time to be $O(N)$, and the memory overhead to be quite high.) More re-

*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.

cently, Bajaj *et al.* [2] propose another contour propagation scheme, with expected performance of $O(K)$. Livnat *et al.* [11] propose NOISE, an $O(\sqrt{N} + K)$-time algorithm. Shen *et al.* [17, 16] also propose nearly optimal isosurface extraction methods.

The first *optimal* isosurface extraction algorithm was given by Cignoni *et al.* [6], based on the use of an interval tree to store the intervals induced by the dataset cells. After an $O(N \log N)$-time preprocessing, queries can be performed in optimal $O(\log N + K)$ time. This achieves tight theoretical bounds.

All the techniques mentioned above are main-memory algorithms, requiring the entire dataset to fit in the main memory (except for Marching Cubes). There is also a class of *external-memory* algorithms (not particularly for isosurface extraction; see [5] for references). For large-scale applications in which the problem is too large to fit in the main memory, Input/Output (I/O) communication between fast main memory and slower external memory (disk) becomes a major bottleneck. Algorithms specifically designed to reduce the I/O bottleneck are called *external-memory* algorithms. Although most of the results in this area of research are theoretical, the experiments of Chiang [3] and of Vengroff and Vitter [19] on some of these techniques show that they result in significant improvements over traditional algorithms in practice. Also, Teller *et al.* [18] describe a system to compute radiosity solutions for polygonal environments larger than main memory, and Funkhouser *et al.* [7] present prefetching techniques for interactive walk-throughs in architectural virtual environments whose models are larger than main memory.

Our Results

We give I/O-optimal techniques for isosurface extraction, by a novel use of the external-memory interval tree of Arge and Vitter [1].

Following the ideas of Cignoni *et al.* [6], we produce for each cell an interval $[\min, \max]$, where $\min$ and $\max$ are the minimum and maximum values among the scalar values of the vertices of the cell. Then given an isovalue q, the active cells are exactly those cells whose intervals *contain* q (i.e., $\min \leq q \leq \max$). This reduces the searching of active cells to the following problem called *stabbing queries*: given a set of intervals and a query point q in 1D, find all intervals containing q. We then use the external-memory interval tree of [1] to solve the stabbing queries in an I/O-optimal way. We propose a new preprocessing algorithm for the interval tree which is I/O-optimal. We give the first implementation of the I/O-optimal interval tree, and also implement our methods as an *I/O filter* for Vtk's isosurface extraction routine [14] for the case of unstructured grids. Our experiments show that the search phase, originally the bottleneck of isosurface extraction, now needs *less time* than the generation phase, i.e., the search phase is not a bottleneck any more!

The advantages of our techniques can be summarized as follows.

- Our query algorithm is output-sensitive and improves the performance of isosurface extraction by speeding up the search phase so that it is no longer a bottleneck, and yet by building the search structure in disk, our preprocessing is performed *once and for all*, as opposed to other output-sensitive techniques which require preprocessing each time the process starts to run. Our search structure can be duplicated by just copying files, without ever running the preprocessing again.

- Our query algorithm only needs two blocks of main memory, and only brings to main memory those K active cells, where K is usually $O(N^{2/3})$. Other techniques either have to visit the whole dataset during queries, or have to use a large amount of main memory to keep the entire dataset plus some additional search structure.

- Our preprocessing algorithm needs only a fixed amount of main memory space, which can be parameterized. For datasets much larger than can fit into main memory, the running time is the same as performing a few external sortings, and is linear in the size of the datasets. Thus our preprocessing method is both efficient and predictable.

Our techniques have a wide range of applications. In addition to improving the performance of isosurface extraction, there are some other implications:

- Datasets can be visualized very efficiently on workstations with just enough main memory to hold the *isosurfaces themselves*.

- Datasets can be kept in remote file servers. Only the necessary parts are fetched during visualization, thus even at ethernet speeds, interactive isosurface extraction can be achieved.

Organization of the Paper

The rest of the paper is organized as follows. In Section 2, we present the I/O-optimal interval tree of [1], and our novel preprocessing algorithm. Next we present in Section 3 the implementation details of our *I/O filter* for Vtk's isosurface extraction routine. In Section 4, we present the overall experimental performance of our methods, followed by conclusions in Section 5.

Because of limited space, our presentation is sometimes sketchy. We refer to the full paper [5] for full details and more complete references.

2 I/O Optimal Interval Tree

As described in the previous section, we produce an interval $I = [\min, \max]$ for each cell of the dataset, and use the I/O-optimal interval tree of [1] to find the active cells of an isosurface by performing stabbing queries.

Since pointer references are very inefficient for disk accesses, we store the *direct cell information* together with the corresponding interval whenever that interval has to be stored in the interval tree. Thus each record of an interval includes the cell ID, the 3D coordinates and the scalar values of the vertices of the cell, and the left and right endpoints of the interval.

If the input dataset is given in the format providing direct cell information, then we can build the interval tree directly. Unfortunately, the datasets are often given in the format that contains indices to vertices. Thus we have to de-reference the indices before actually building the interval tree. We call this de-referencing process *normalization*. Using the technique of [4], we can efficiently perform normalization as follows. We make one file (the *vertex file*) containing the direct information of the vertices (3D coordinates and scalar values), and another file (the *cell file*) of cell records with vertex indices. In the first pass, we externally sort the cell file by the indices (pointers) to the first vertex, so that the first group in the file contains cells whose first vertices are vertex 1, the second group contains cells whose first vertices are vertex 2, and so on. Then by scanning through the vertex file and the cell file simultaneously, we fill in the direct information of the first vertex of each cell in the cell file. In the next pass, we sort the cell file by the indices to the second vertices, and fill in the direct information of the second vertex of each cell in the same way. By repeating the process for each vertex of the cells, we obtain the direct information for each cell; this completes the normalization process.

2.1 Data Structure

We first review the data structure of the I/O-optimal interval tree [1]. Each node of the tree (for readers not acquainted with interval trees, see [13, pages 360–361]) is one block in disk, capable of holding $O(B)$ items. The *branching factor*, b, is the maximum number of children an internal node can have. We let $b = O(\sqrt{B})$; the reason will be clear later. Let S be the set of all N intervals, and E be the set of $2N$ endpoints of the intervals in S. We denote by $n = \lceil |E|/B \rceil$ the number of *blocks* in E. First, we sort E from left to right in the order of increasing values, assuming that all endpoints have distinct values (we use cell ID to break ties). Set E is now *fixed* and will be used to define *slab boundaries* for each internal node of the tree. The interval tree on E and S is defined recursively as follows. The root u is associated with the entire range of E and with all intervals S. If S has no more than B intervals, then node u is a leaf storing all intervals of S. Otherwise u is an internal node. We then evenly divide E into b *slabs* $E_0, E_1, \cdots, E_{b-1}$, each containing the same number $\lceil n/b \rceil$ *blocks* of endpoints in E. The $b-1$ *slab boundaries* are the first endpoints of slabs $E_1, \cdots, E_{b-1}$. We store the endpoint values of the slab boundaries in node u as keys. We use these keys to consider each interval $I \in S$ (see Fig. 1). If both endpoints of I lie inside the same slab, say the i-th slab E_i, then I belongs to the i-th child u_i of node u, and is put into the interval set $S_i \subseteq S$ (e.g., in Fig. 1, interval I_0 is put in S_0). Otherwise (the two endpoints of I belong to different slabs, i.e., I crosses one or more slab boundaries), I belongs to node u. The intervals belonging to node u will be stored in the secondary lists of u pointed by pointers in u. We adopt the convention that if an endpoint of I is exactly the slab boundary separating slabs E_{i-1} and E_i, that endpoint is considered as lying in slab E_i; this is consistent with our choice of slab boundaries. We associate each child u_i of node u with the range of slab E_i and with the interval set S_i, and define the subtree rooted at u_i recursively as the interval tree on range E_i and intervals S_i. Notice that slab E_i is *pre-defined* when E is given ($\lceil n/b \rceil$ blocks of endpoints in the first level, $\lceil n/b^2 \rceil$ blocks of endpoints in the next level, and so on), but set S_i has to be decided by scanning through the intervals in S and distribute them appropriately according to the slab boundaries. Observe that S_i may be empty, in which case child u_i of u is null (it is also possible that all children of u are null).

For each internal node u, we use three kinds of secondary structures to store the intervals belonging to u: the *left, right* and *multi* lists, described as follows.

- There are b *left* lists, each corresponding to a *slab* of u. For each i, the i-th *left* list stores the intervals belonging to u whose left endpoints lie in the i-th slab E_i. Each list is sorted by *increasing left endpoint values* of the intervals (see Fig. 1).

- There are also b *right* lists, each corresponding to a *slab* of u. For each i, the i-th *right* list stores the intervals belonging to u whose right endpoints lie in the i-th slab E_i. Each list is sorted by *decreasing right endpoint values* of the intervals (see Fig. 1).

- There are $(b-1)(b-2)/2$ *multi* lists, each corresponding to a *multi-slab* of u. A *multi-slab* $[i, j]$, $0 \le i \le j \le b-1$, is defined to be the union of slabs $E_i, \cdots, E_j$. The *multi* list for multi-slab $[i, j]$ stores all intervals of u that *completely span* $E_i \cup \cdots \cup E_j$, i.e., all intervals of u whose left endpoints lie in slab E_{i-1} and whose right endpoints lie in slab E_{j+1}. Since the *multi* lists $[0, k]$ for any k and the *multi* lists $[\ell, b-1]$ for any l are always empty by the definition, we only care about multi-slabs $[1,1], \cdots, [1, b-2], [2,2], \cdots, [2, b-2], \cdots, [i,i], \cdots, [i, b-2], \cdots, [b-2, b-2]$. Thus there are $(b-1)(b-2)/2$ such multi-slabs and the associated *multi* lists (see Fig. 1).

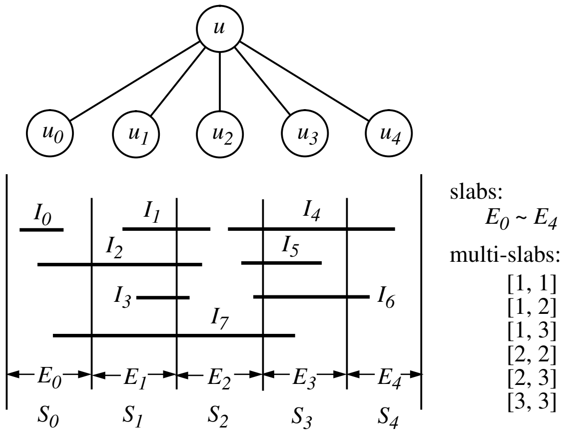

Figure 1: A schematic example of the I/O-optimal interval tree for branching factor $b = 5$. Note that this is not a complete example and some intervals are not shown (in a complete example, each slab boundary is an interval endpoint, and each slab E_i has same blocks of endpoints). Consider only the intervals shown here and node u. The interval sets for its children are: $S_0 = \{I_0\}$, and $S_1 = S_2 = S_3 = S_4 = \emptyset$. Its *left* lists are: $left(0) = \{I_2, I_7\}$, $left(1) = \{I_1, I_3\}$, $left(2) = \{I_4, I_5, I_6\}$, and $left(3) = left(4) = \emptyset$ (the intervals in each list are sorted in the order as they appear). Its *right* lists are: $right(0) = right(1) = \emptyset$, $right(2) = \{I_1, I_2, I_3\}$, $right(3) = \{I_5, I_7\}$, and $right(4) = \{I_4, I_6\}$ (again the intervals in each list are sorted in the order as they appear). Its *multi* lists are: $multi([1,1]) = \{I_2\}$, $multi([1,2]) = \{I_7\}$, $multi([1,3]) = multi([2,2]) = multi([2,3]) = \emptyset$, and $multi([3,3]) = \{I_4, I_6\}$.

For each *left*, *right*, or *multi* list, we store the entire list in consecutive blocks in disk, and in node u we store a pointer to the starting position of the list in disk. Observe that there are b *left* and b *right* lists, and $O(b^2) = O(B)$ *multi* lists. Thus we need to keep $O(B)$ items of information in node u, which is one disk block capable of holding $O(B)$ items. This explains why the branching factor b is taken as $O(\sqrt{B})$.

Now let us analyze some properties of the interval tree. First, the height of the tree is $O(\log_b N) = O(\log_B N)$, because each time we go down one level of the tree, the range of slab E associated with a node is reduced by a factor of b. Secondly, each interval belongs to exactly one node, and is stored at most three times: if it belongs to a leaf node, then it is stored only once; if it belongs to an internal node, then it is stored once in some *left* list, once in some *right* list, and possibly one more time in some *multi* list. Therefore we need *roughly* $O(N/B)$ disk blocks to store the entire data structure. In theory, more blocks may be needed, and the optimal space of $O(N/B)$ blocks is achieved by using the corner structure [10]; see [1, 5] for more details.

2.2 Query Algorithm

We now review the query algorithm given in [1]. Given query value q, we perform the following recursive process starting from the root of the interval tree. For the current node u that we want to visit, we read it from disk (recall that u is a disk block). If u is a leaf, we just check the $O(B)$ intervals stored in u, report those intervals containing q and stop. If u is an internal node, we perform a binary search for q on the keys (slab boundaries) stored in u to identify the slab E_i containing q. Now we want to report all intervals belonging to u that contain q. We check the i-th *left* list $left(i)$ of u, scan through the list from beginning and report intervals, until we reach some interval whose *left* endpoint value is *larger* than q.

Recall that each *left* list is sorted by increasing left endpoint values of the intervals. Similarly, we check the *right* list $right(i)$ of u, scan through the list from beginning and report intervals, until we reach some interval whose *right* endpoint value is *smaller* than q. Again, recall that each *right* list is sorted by decreasing right endpoint values of the intervals. Also, all intervals that *span* slab E_i must contain q and thus must be reported. We carry out this task by reporting all intervals in *multi* lists $multi([\ell, r])$, where $1 \leq \ell \leq i$ and $i \leq r \leq b-2$. Finally, we visit the i-th child u_i of node u, and recursively perform the same process on u_i (if child u_i is null then we stop).

2.3 Preprocessing Algorithm

We give a new preprocessing algorithm achieving nearly optimal $O(\frac{N}{B} \log_B N)$ I/O's. It is based on a paradigm we call *scan and distribute*, inspired by the *distribution sweep* I/O technique [3, 8]. The algorithm can also be made I/O-optimal [5].

The algorithm follows the definition of the interval tree given in Section 2.1. We start by duplicating each interval record, one with the left endpoint as the key and the other with the right endpoint as the key. We then sort (using external sorting) all these endpoints by the keys from left to right in increasing order (breaking ties by cell ID's). This gives the sorted set of endpoints E. This set is used to decide the range and the slab boundaries of the current node throughout the process. Again, we use n to denote the number of blocks in E. The set S of intervals sorted by increasing left endpoint values is initially created by copying and filtering out the right endpoints from E. Now we use a recursive procedure to build the tree, as follows. The root u of the tree is associated with the entire range of E and the entire interval set S. If S has no more than B intervals, then we make node u a leaf, store those intervals in u, and stop. Otherwise, we make u an internal node. We extract the $b-1$ endpoints from E as slab boundaries which evenly divide E into b slabs $E_0, \cdots, E_{b-1}$ of $\lceil n/b \rceil$ blocks each. We then scan through the set S; for each interval I being examined, we perform a binary search for each of the two endpoints on the slab boundaries, and decide whether I lies entirely inside some slab E_i (in which case we have to put I into set S_i), or crosses some slab boundary (in which case I belongs to node u and we want to put I into appropriate *left, right* and/or *multi* lists). We use a temporary list for each S_i, and similarly for each list $left(i)$, $right(i)$, and $multi([\ell, r])$ of node u. Each temporary list is kept in consecutive blocks in disk, and is also associated with one *buffer*, which is of one disk block size, in the main memory. Notice that we only need $O(B)$ blocks in the main memory for the buffers. In our actual implementation, we only need $4b$ buffers instead; see Section 3. Each time we want to put the current interval I into some set S_i or some *left/right/multi* list, we just insert I into the buffer of the corresponding temporary list. When the buffer is full, we write the buffer out to the corresponding temporary list in disk, and the buffer is again available for use. After scanning through the entire interval set S, each interval in S is distributed to its appropriate temporary lists. Observe that originally S is sorted by increasing left endpoint values of the intervals, thus after this *scan and distribute* process is done, each temporary list is automatically sorted by increasing left endpoint values as well. We then sort (using external sorting) each *right* list by decreasing right endpoint values of the intervals, so that all temporary lists are in the desired sorted orders. After this, we copy each temporary list back to its corresponding list in disk, set up appropriate information (e.g., the number of intervals and a pointer to the starting position in disk of each list, etc) in node u. Finally, we write node u to the disk, and recursively perform the same procedure on each child u_i (of node u) with the range of slab E_i and interval set S_i (if S_i is empty then node u_i is null).

3 Implementation

We implemented our methods as an *I/O filter* for Vtk's isosurface extraction for the case of unstructured grids. First, `ioBuild` is used to preprocess the dataset to build the interval tree, and then `ioQuery` is used to report the active cells of a given isosurface. We first describe how to implement the interval tree, and then describe the interface with Vtk.

3.1 External Memory Interval Tree

We describe the organization of the data structure. We use files `dataset.intTree`, `dataset.left`, `dataset.right`, and `dataset.multi` to hold the interval tree nodes, all *left* lists, all *right* lists, and all *multi* lists, respectively. Every time we create a new tree node, we allocate the next available block from file `dataset.intTree` and store the node there. (The root of the tree always starts from position 0 of file `dataset.intTree`.) Similarly, every time we create a new *left* (resp. *right/multi*) list, we allocate the next available consecutive blocks just enough to hold the list, from file `dataset.left` (resp. `dataset.right/dataset.multi`) and store the list there. Notice that we always allocate disk space of size an *integral* number of blocks. Each block in file `dataset.left`, `dataset.right` and `dataset.multi` stores up to B intervals. Each interval contains a cell ID, four vertices of the cell (x-, y-, z- values and the scalar value of each vertex), and the left and right endpoints of the interval associated with the cell, which are the min and max values of the four scalar values. In our case with one disk block being 4,096 bytes, a block can store up to 53 intervals, i.e., $B = 53$.

Now we describe the layout of the interval tree nodes, each of size one disk block. It has a flag to indicate whether it is a leaf or an internal node. If it is a leaf, the rest of the node contains the following: (1) number ($\leq B$) of intervals stored in the node, and (2) actual intervals stored. If it is an internal node, the rest of the node contains the following: (1) number ($\leq b-1$) of keys (i.e., slab boundaries) stored, (2) the actual slab boundaries stored, (3) b pointers to the starting positions of the children nodes in file `dataset.intTree` (-1 if that child is null), (4) information about its b *left* lists, (5) information about its b *right* lists, and (6) information about its $(b-1)(b-2)/2$ *multi* lists. The information about each *left* list include: (a) a pointer to the starting position of the list in file `dataset.left`, (b) number of intervals in the list, and (c) the minimum left endpoint value among all intervals of the list (which is just the left endpoint value of the first interval in the list, according to the sorted order of the list). Item (c) is used to speed up the query: when we need to check the query point q against this *left* list, if q is smaller than the value stored here, then no interval in the list will contain q and thus we can avoid reading any block of the list. The information about each *right* list are similar. The information about each *multi* list are also similar, but do not contain item (c) since during queries a *multi* list is reported as a whole with no checking necessary. From the size of each data type of each field and the fact that a node holds at most 4,096 bytes, we can compute the branching factor b of the tree, which in our case is 29 (observe that $b > \sqrt{B}$ here). For the simplicity of the coding, we currently do not implement the corner structure.

There are some interesting issues involved in the implementation of the preprocessing algorithm. Recall from Section 2.3 that during the *scan and distribute* process for the current node u (associated with endpoint set E and interval set S), we use a temporary list for each of the b *left* lists, b *right* lists, $(b-1)(b-2)/2$ *multi* lists, and b interval sets S_i. A first attempt would be to use a file for each temporary list. This would require us to open $3b+(b-1)(b-2)/2$ files *at the same time*, since no temporary list is completed until one pass of the *scan and distribute* process is done. 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).

Our solution to this problem is to use a scratch file as a collection of the temporary lists of the same type. For example, we use file `dataset.left_temp` to collect all temporary lists for the b *left* lists. Observe that all intervals belonging to $left(i)$ (and thus belonging to the temporary list of $left(i)$) must have their left endpoints lying in slab E_i, but there are no more than $\lceil n/b \rceil$ blocks of endpoints in slab E_i, where n is the number of blocks in E. Therefore, each temporary list has at most $\lceil n/b \rceil$ blocks of intervals, and thus we let the i-th temporary list start from block $i \cdot \lceil n/b \rceil$ of file `dataset.left_temp`, for $i = 0, \cdots, b-1$. Notice that the size of file `dataset.left_temp` is no more than the size of E. After the construction of all temporary left lists is complete, we copy them to the file `dataset.left`, and the scratch file `dataset.left_temp` is again available for use in the next recursion. We handle the temporary lists for the *right* lists in the same way, except that before copying to file `dataset.right`, each temporary right list has to be sorted first (in the order of decreasing right endpoint values). In the same way, each interval set S_i for child u_i of node u has at most $\lceil n/b \rceil$ blocks. We scan the entire file for S (file `dataset.intvls`) and distribute the intervals to the appropriate temporary lists for each S_i (i.e., appropriate portions of the scratch file holding a collection of all temporary lists for each S_i), and then copy the temporary lists back to the corresponding portions of file `dataset.intvls`. Now each set S_i is just the portion of file `dataset.intvls` starting from block $i \cdot \lceil n/b \rceil$ with no more than $\lceil n/b \rceil$ blocks. Each set S_i, together with slab E_i, are then used as input for the next level of recursion. Finally, consider the construction of the *multi* lists for the current node u. By the same argument, each list has at most $\lceil n/b \rceil$ blocks. Unfortunately, there are $(b-1)(b-2)/2$ such lists. If we collected all temporary lists into a single scratch file by the above method, then this scratch file would have size $\Theta(nb)$ blocks, which is definitely undesirable. To solve the problem, observe that $multi([i, j])$ consists of all intervals with left endpoints in the same slab E_{i-1}. Therefore, we construct all *multi* lists $multi([i, j])$ for a fixed i from the *left* list $left(i-1)$ (again by a *scan and distribute* process), and repeat the process $b-2$ times for all possible values of i. Then during each iteration, there are at most b *multi* lists being constructed, and thus our scratch file only needs n blocks of space. The number of buffers in main memory needed for constructing the *multi* lists is also reduced from $(b-1)(b-2)/2$ to b. In summary, to construct the *left, right*, and *multi* lists and the interval sets S_i for children, we use four scratch files, each with size n blocks, and also $4b$ blocks of main memory as buffers.

Handling Degeneracies

The issue of degenerate cases arises when the endpoint values of the intervals are not distinct. We use cell ID's to break ties. We also adopt the convention that if an endpoint is exactly the slab boundary separating slabs E_{i-1} and E_i, then this endpoint is considered as lying in slab E_i; this is consistent with our choice of slab boundaries (see Section 2.1). In the internal node of the interval tree, we only store the endpoint values as slab boundaries (keys), without storing the corresponding cell ID's. During query operations, if the query value q has the same value as some slab boundary, we consider all slabs that can possibly contain the value of q, and perform the query operation on all such slabs accordingly. This ensures that all answers are correctly reported. Notice that if several slab boundaries have the same value as that of q, we only need to go to the two children respectively to the left of the leftmost such slab boundary and to the right of the rightmost such boundary. This is because all other children in between can only contain intervals whose two endpoint values are the same (same as q); the corresponding cells of such intervals are thus lying in the *interior* of the isosurface and therefore are not interesting cells to be reported.

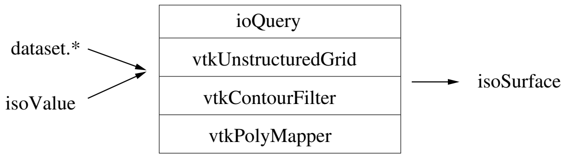

Figure 2: Isosurface extraction phase. Given the four files of the interval tree and an isovalue, `ioQuery` filters the dataset and passes to Vtk only those active cells of the isosurface. Several Vtk methods are used to generate the isosurface, in particular, vtkUnstructuredGrid, vtkContourFilter, and vtkPolyMapper.

3.2 Interfacing with Vtk

Extracting isosurfaces with `ioBuild` and `ioQuery` is relatively simple. The input to `ioBuild` is a Toff file*, which unfortunately contains indices to vertices. Therefore, the first part of `ioBuild` is to normalize the Toff file, de-referencing these indices as described in the beginning of Section 2, before the actual construction of the interval tree can begin. Since there are four vertices in each cell (tetrahedron), four passes over the input are necessary. If the files are already given in de-referenced form, the first part of `ioBuild` would not be necessary.

A full isosurface extraction pipeline should include several steps in addition to finding active cells. In particular, (1) intersection points and triangles have to be computed; (2) triangles need to be decimated [15]; and (3) triangle strips have to be generated. Steps (1)–(3) can be carried out by the existing code in Vtk [14], which makes it a perfect match for a proof-of-concept implementation of our I/O techniques. Our current code only implements the actual triangle generation. Using Vtk's simple pipeline scheme, it is a simple programming exercise to further process the triangulation, decimate it and create the strips.

The `ioQuery` code is implemented by linking our I/O querying code with Vtk's isosurface generation, as shown in Fig. 2. 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 `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 [14]).

4 Experimental Results

In this section we present experimental results of actively using our I/O filtering techniques on real datasets. We have run our experiments on four different datasets shown in Table 2. All of these datasets are tetrahedralized versions of well-known datasets. Our primary interest in this initial implementation was to quantify the

*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.

I/O overhead, if any, both in terms of memory and time, and to compare with Vtk's native isosurface implementation.

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 (kernel 2.0.27) 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).

Preprocessing with `ioBuild`

Using `ioBuild` is a fully automatic process. The only argument is the name of the input file containing tetrahedral cells. During its execution, `ioBuild` creates and writes multiple files, but in the end only the four files are kept. In analyzing the behavior of external memory algorithms, it is very important to take into account the amount of main memory used by the algorithm. In general, the more memory it is given, the less I/O operations it needs. For instance, when sorting a file that can be kept entirely in main memory, we just need to touch the file twice, once for reading, and once for writing. As the available main memory is smaller, we need to perform more I/O operations. The scalability of an external memory algorithm is best seen when the main memory is smaller than the dataset. In order to simulate this, we only allow `ioBuild` to allocate a 1024K blocks (4 Mb) of RAM. This is a parameter in the program.

The first time we ran `ioBuild`, it just seemed to be *too fast* for the number of I/O reads and writes `ioBuild` was issuing. It turned out that the operating system was able to optimize (by caching) a lot of those I/O requests, and the CPU was running at nearly 95% of usage. In order to avoid these side effects of the operating system, 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, and during the time of our benchmarks, our system was fully functional (i.e., it was not in single user mode; all system-related daemons were active). That is, we ran our experiments in a "normal" environment on the datasets with size equal to or larger than the main memory. (Remember that only 4M of RAM were actually used by `ioBuild`.) The actual CPU usage percentage during this preprocessing was in the range of low teens.

In Table 1 we show all relevant experimental data obtained from running `ioBuild` with a 16M/4M configuration. Recall from Section 3.2 that the first part of `ioBuild` is to normalize (i.e., dereference) the input Toff file (which in turn amounts to several external sorting operations), and the second part is to actually construct the interval tree. The basic operations of `ioBuild` are hence external sorting and scanning (the *scan and distribute* process described in Section 2.3). It should be clear from Table 1 that the overall running time of `ioBuild` is the same as performing external sorting several times, and is *linear* in the size of the datasets. Therefore `ioBuild` is both efficient and predictable — for a given system configuration, one can estimate (or extrapolate) the overall running time based on a linear behavior. This preprocessing can be made much faster by using faster disks, and optimized to use more memory. For instance, in an SGI Power Challenge, it only takes 10 minutes to preprocess the Delta Wing dataset.

The interval tree data structure requires more disk space than the normalized tetrahedron file, which is also larger than the original Toff file. The average increase in disk space is about 7.7 times, and the maximum increase is 8.5 times (see Table 1). During the preprocessing time, scratch files used for computation are necessary. In general, one needs about 16 times the amount of disk space of the original Toff file to generate the interval tree. Notice, however, that this extra disk space for scratch files is needed *once and for all*, since after running `ioBuild` once to build the interval tree files, these tree files can be duplicated by just copying, without ever running `ioBuild` again.

	Blunt	Chamber	Post	Delta
intTree	303K	147K	238K	750K
left	15.6M	17.3M	41.0M	80.5M
right	15.6M	17.3M	41.0M	80.5M
multi	17.6M	20.0M	34.6M	80.6M
Total Size	49.2M	54.7M	116.8M	242.3M
Original Size	5.8M	6.8M	16.4M	33.8M
Ratio of 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 1: Statistics for the running of `ioBuild` in a machine with 16M of main memory and 4M of buffer memory, for the datasets in Table 2. The first four values are the sizes of the four files kept after the preprocessing. "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 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.

Because it might be necessary to use as much as 16 times the original disk space for preprocessing a given dataset, we believe that in large production environments, large scratch areas should be available for preprocessing. We see this as a minor cost for the overall savings in both time and space later on. One should also note, that disk prices are on the order of 35-40 times lower than main memory prices. So 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 costs. (In [17], a twofold main memory overhead is reported for improved, although not optimal, isosurface generation times.) Again, we would like to stress the overall speed advantage in isosurface extraction time our technique provides, since it only performs the preprocessing once.

Isosurface Extraction with `ioQuery`

The true performance of an isosurface extraction technique is the actual time it takes to generate a given isosurface. As explained before, our code is coupled with Vtk (see Fig. 2). Basically, during isosurface extraction, we find the active cells, and use Vtk's isosurface capabilities to actually generate the isosurface. In the following, we use `ioQuery` to denote the entire isosurface extraction code.

We ran three batteries of tests, each with different amount of core memory (128M, 64M, and 32M). Each test consists of calculating 10 isosurfaces with isovalues in the range of the scalar values of each dataset, by using our code and also the Vtk-only isosurface code (denoted by `vtkiso`). We did not run X-windows during the isosurface extraction time, and the output of vtkPolyMapper was saved in a file instead.

We found that our approach has several advantages. Initially we thought that our method would only be useful for really large

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 2: A list of the datasets used for testing. After normalization, each cell is 64 bytes long (data is represented as floats).

datasets, but our experiments show that even for smaller datasets the I/O querying time is negligible when compared to the overall isosurface extraction time.

Another advantage is that, for large isosurfaces, the memory requirements of the triangle mesh is very large. If the entire volume dataset is kept in the main memory, then even if the dataset itself can fit, adding the triangle mesh might make the main memory not large enough and cause system swap. The fact that `ioQuery` only uses two blocks of main memory during active-cell searching makes our approach very favorable: the main memory space can be saved for the triangle mesh of the isosurface, so that system swap is much less likely to occur.

We summarize in Table 3 the *total* running times for the extraction of the 10 isosurfaces using `ioQuery` and `vtkiso` with different amount of main memory. It should be clear that for larger datasets (e.g., Post and Delta) `ioQuery` is much faster, and the margins increase significantly as the main memory size goes down.

Figures 3 to 5 summarize detailed benchmarks, with top rows presenting the performance of running `ioQuery`, and the bottom rows for the performance of `vtkiso`. For each isosurface calculated using `ioQuery`, we break the time into four categories:

(1) I/O time (shown in red) – This is the time to identify and bring in from disk the active cells of the isosurface in question.

(2) Copy Time (shown in yellow) – In order to use Vtk's isosurface capabilities, we need to generate a vtkUnstructuredGrid object that contains the active cells we just obtained. We refer to the time for this process as "Copy Time".

(3) Isosurface (shown in blue) – This is the time for Vtk's isosurface code to actually generate the isosurface from the active cells.

(4) File Output (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. 3–5, and there are two additional costs *not shown*: the reading of the dataset, and the generation of the vtkUnstructuredGrid. These two operations are performed only once when `vtkiso` starts to run (i.e., at the beginning of each batch of the 10 isosurface extractions), and thus it is not possible to spread the costs over each individual isosurface extraction. Therefore we do not show them in Figs. 3–5, but only 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 64M and 32M, and for Delta in all three memory sizes, are all much larger than the corresponding entries of `ioQuery`. This means that before the time `vtkiso` finishes reading the dataset and generating a vtkUnstructuredGrid, `ioQuery` already finished calculating all 10 isosurfaces!

Regarding Figs. 3–5, we make the following observations:

- For `ioQuery`, in general the I/O time is only a small fraction of the overall isosurface extraction time (see the top rows of Figs. 3–5). In particular, for most of the time (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. One can see the output-sensitive behavior by noting that when small isosurfaces (or no isosurfaces) are generated, `ioQuery` takes negligible time.

- For `ioQuery`, the I/O times almost do not change with the amount of main memory (see the top rows of Figs. 3–5). This shows that our technique for finding active cells only needs a very small amount of main memory, and the performance is independent of the size of the main memory available.

- For `ioQuery`, the overall running times almost do not change with the amount of main memory (see the top rows of Figs. 3–5). We expect that the overall running times will increase only when Vtk needs more main memory than available to store the actual polygons generated (and thus causes page faults). The fact that our technique only needs a very small amount of main memory to find active cells enables us to visualize much larger isosurfaces very efficiently.

- Even without taking into account the two costs of `vtkiso` (reading the dataset and generating a vtkUnstructuredGrid), comparing the times of `vtkiso` and of `ioQuery` as shown in Figs. 3–5 (which is thus not fair for `ioQuery`), `ioQuery` is already much faster for large datasets. For example, for the 8th isosurface of Delta Wing using 128M of RAM, it takes `ioQuery` 11 seconds, while `vtkiso` takes about 34 seconds. As for small datasets, the overhead of `ioQuery` is negligible. But the advantages really start to show as the main memory size goes down. For instance, with 64M, for the 8th isosurface of Delta Wing, it takes `ioQuery` the same 11 seconds, while `vtkiso` takes 300 seconds!

	Blunt	Chamber	Post	Delta
`ioQuery` – 128M	7s	16s	18s	31s
`vtkiso` – 128M	15s	22s	44s	182s
`vtkiso` I/O – 128M	3s	2s	12s	40s
`ioQuery` – 64M	9s	17s	21s	31s
`vtkiso` – 64M	18s	27s	112s	3122s
`vtkiso` I/O – 64M	5s	6s	67s	224s
`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 `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.

5 Conclusions

We have presented an I/O-optimal technique for isosurface extraction from volumetric data. Our method is to preprocess the dataset to build an I/O-optimal interval tree in disk, and then to extract isosurfaces by querying the interval tree for active cells and generating the isosurfaces from those cells. We discussed the theoretical aspects of the method. It is I/O-optimal in space, query, and preprocessing. We also established its practical efficiency through experimental results, based on our implementation of an *I/O filter* for Vtk's isosurface extraction for the case of unstructured grids. We show that the time for searching active cells is less than the time for generating the isosurfaces from active cells, and this search time is independent of the main memory available. Also, with the interval tree built in disk, there is no need to load the entire dataset into main memory. In addition, the preprocessing is performed once and for all; its running time is the same as running external sorting a few times, and is linear in the size of the datasets. All these features make our technique especially suitable for use with very large datasets, or when there is only a small amount of main memory. In fact, even for smaller datasets with enough main memory, our method introduces only a negligible overhead.

Acknowledgements

We thank Pat Crossno, Lichan Hong, Joseph Mitchell and Dino Pavlakos for useful criticism and help in this work. We thank Lars Arge and Jeff Vitter for useful comments about the presentation of their I/O-optimal interval tree, and Lars Arge for useful discussions on the preprocessing of the tree. The first author would also like to thank Roberto Tamassia for his constant support and encouragement. 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[14]. 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] L. Arge and J. S. Vitter. Optimal interval management in external memory. In *Proc. IEEE Foundations of Comp. Sci.*, pages 560–569, 1996.

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

[3] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures*, pages 346–357, 1995.

[4] 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.

[5] Y.-J. Chiang and C. T. Silva. I/O Optimal Isosurface Extraction. 1997. Full paper available at `http://www.ams.sunysb.edu/~yjc/iso.html`.

[6] 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.

[7] T. A. Funkhouser, S. Teller, C. H. Séquin, and D. Khorramabadi. Database management for models larger than main memory. In *Interactive Walkthrough of Large Geometric Databases, Course Notes 32, SIGGRAPH '95*, pages E15–E60, 1995.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, July 1987.

[13] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

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

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

[16] 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.

[17] 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.

[18] S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan. Partitioning and ordering large radiosity computations. Computer Graphics Proceedings, Annual Conference Series, pages 443–450. ACM SIGGRAPH, ACM Press, July 1994.

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

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

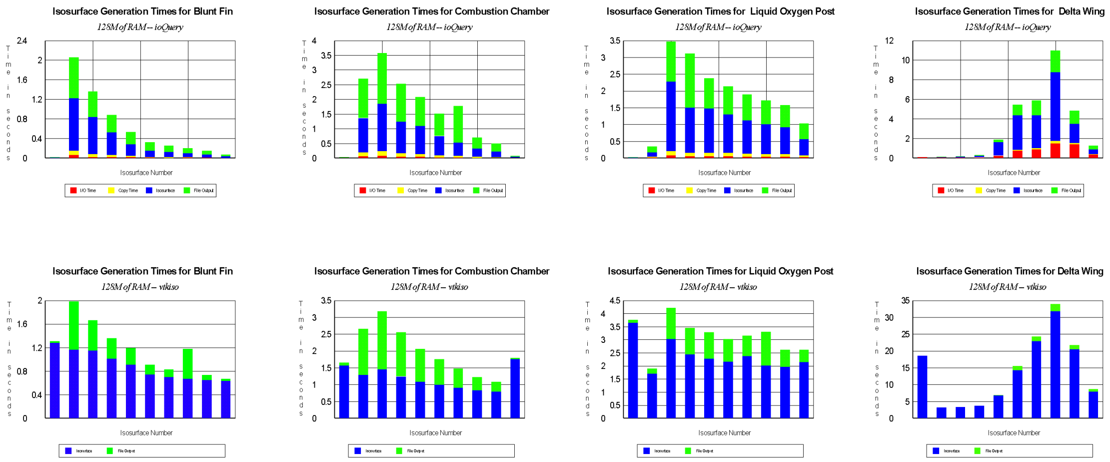

Figure 3: Running times for extracting isosurfaces using `ioQuery` (top row) and `vtkiso` (bottom row) with 128M of main memory.

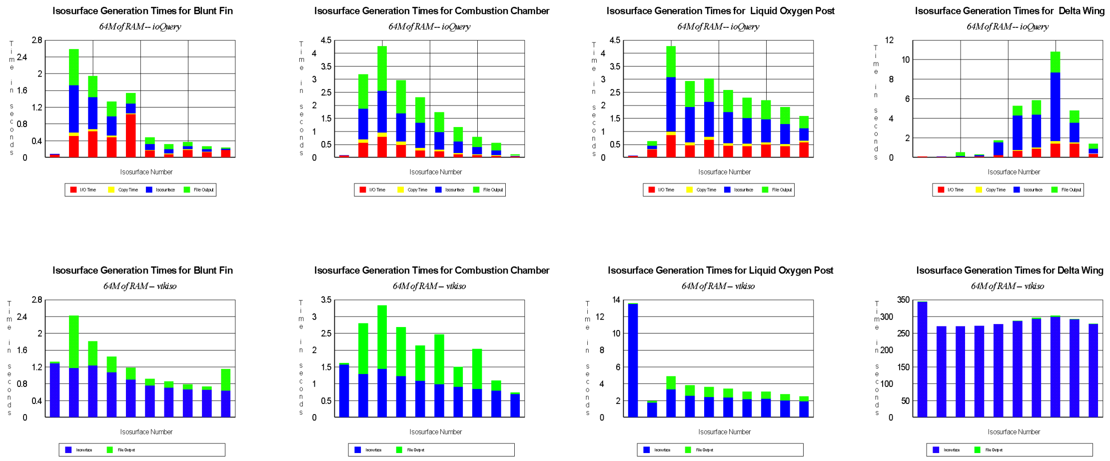

Figure 4: Running times for extracting isosurfaces using `ioQuery` (top row) and `vtkiso` (bottom row) with 64M of main memory.

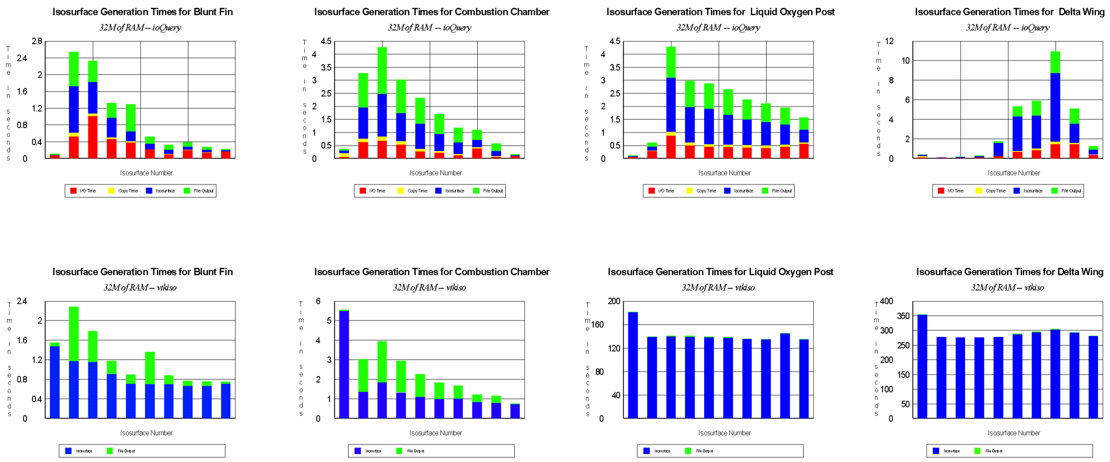

Figure 5: Running times for extracting isosurfaces using `ioQuery` (top row) and `vtkiso` (bottom row) with 32M of main memory.