Interactive Out-Of-Core Isosurface Extraction

Yi-Jen Chiang[*] Polytechnic University

Cláudio T. Silva[†] IBM T. J. Watson Research Center

William J. Schroeder[‡] Kitware

Abstract

In this paper, we present a novel *out-of-core* technique for the interactive computation of isosurfaces from volume data. Our algorithm minimizes the main memory and disk space requirements on the visualization workstation, while speeding up isosurface extraction queries. Our overall approach is a *two-level indexing* scheme. First, by our *meta-cell* technique, we partition the original dataset into clusters of cells, called *meta-cells*. Secondly, we produce *meta-intervals* associated with the meta-cells, and build an indexing data structure on the meta-intervals. We *separate* the cell information, kept only in meta-cells in disk, from the indexing structure, which is also in disk and only contains pointers to meta-cells. Our meta-cell technique is an I/O-efficient approach for computing a k-d-tree-like partition of the dataset. Our indexing data structure, the *binary-blocked I/O interval tree*, is a new I/O-optimal data structure to perform *stabbing queries* that report from a set of meta-intervals (or intervals) those containing a query value q. Our tree is simpler to implement, and is also more space-efficient in practice than the existing structures. To perform an isosurface query, we first query the indexing structure, and then use the reported meta-cell pointers to read from disk the *active* meta-cells intersected by the isosurface. The isosurface itself can then be generated from active meta-cells. Rather than being a single-cost indexing approach, our technique exhibits a *smooth trade-off* between query time and disk space.

Keywords: Isosurface Extraction, Marching Cubes, Out-Of-Core Computation, Interval Tree, Scientific Visualization.

1 Introduction

Isosurface extraction represents one of the most effective and widely used techniques for the visualization of volume datasets. Formally, 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 q, extracting the isosurface of q is to compute the isosurface $C(q) = \{\mathbf{x}|\mathcal{F}(\mathbf{x}) = q\}$. The computation process can be divided into two phases: First, one finds the *active* cells that are intersected by the isosurface (the *search phase*), and then, one can compute the isosurface from the active cells (the *generation phase*). Most of the isosurface algorithms require the entire dataset to be kept in main memory, which is a severe limitation on their applicability, especially for large scientific applications.

In this paper, we present an isosurface technique whose main memory and disk space requirements on the visualization workstation are minimized, while speeding up the isosurface extraction procedure. In the same flavor as the methods of [10, 11], we index the dataset cells to achieve output-sensitive searches. Also, as in [10, 11], we keep both the indices (*i.e.*, intervals obtained from the cells) and the original dataset in *disk*, rather than in main memory. Moreover, during isosurface queries only a small portion of the dataset is touched and brought to main memory, by performing (using an indexing data structure) *stabbing queries* that report from a set of intervals those containing the query value q.

In [10, 11], to avoid inefficient *pointer references* in disk, the *direct cell information* is stored with its interval, in the indexing data structure. This is very inefficient in disk space, since the vertex information is duplicated many times, once for each cell sharing the vertex. Moreover, in the indexing structures [3, 18] used, each interval is stored three times in practice, increasing the duplications of vertex information by another factor of three. To eliminate this inefficiency, our indexing scheme uses a *two-level* structure. First, we partition the original dataset into clusters of cells, called *meta-cells*. Secondly, we produce *meta-intervals* associated with the meta-cells, and build our indexing data structure on the meta-intervals. We *separate* the cell information, kept only in meta-cells in disk, from the indexing structure, which is also in disk and only contains pointers to meta-cells. Isosurface queries are performed by first querying the structure, then using the reported meta-cell pointers to read from disk the *active* meta-cells intersected by the isosurface, which can then be generated from the active meta-cells.

While we need to perform *pointer references* in disk from the indexing structure to meta-cells, the *spatial coherences* of isosurfaces and of our meta-cells ensure that each meta-cell being read contains *many* active cells, so such pointer references are efficient. Also, a meta-cell is always read as a whole, hence we can use pointers *within* a meta-cell to store each meta-cell compactly. In this way, we obtain efficiencies in *both* query time and disk space. Two new techniques lie at the heart of this paper. One is the *meta-cell* technique that computes the spatially coherent meta-cells. The other is the *binary-blocked I/O interval tree*, a new I/O-optimal stabbing-query data structure that is simpler to implement and more space-efficient in practice than those in [3, 18]. We believe both techniques will find applications other than efficient out-of-core isosurface extraction.

We summarize the contributions of this work as follows.

- We present a novel out-of-core isosurface technique that improves [10, 11]. While keeping the querying time and main memory requirement small, the disk space overhead is reduced by more than one order of magnitude.
- We give a new *meta-cell* technique that partitions a volume dataset into spatially coherent meta-cells. This can be viewed as an out-of-core k-d-tree-like partition, and is efficiently carried out by performing external sorting a few times.
- We propose the *binary-blocked I/O interval tree*, a new I/O-optimal stabbing-query data structure. Previous such structures [3, 18] both have three types of secondary lists, but our tree has only two types of lists (as in the original main memory interval tree of [14]), so it has the tree size reduced by a factor of 2/3 in practice, and is also simpler to implement.

Previous Related Work

We first briefly review the work on out-of-core, or *I/O* techniques. In addition to early work on sorting and scientific computing, recently there have been I/O algorithms for graphs and for computational geometry; see [10, 11] for the references. Although most

[*] yjc@photon.poly.edu

[†] csilva@watson.ibm.com

[‡] william.schroeder@kitware.com

of the results are theoretical, the experiments of Chiang [8], Vengroff and Vitter [27], and Arge *et al.* [2] on some of these techniques show that they result in significant improvements over traditional algorithms in practice. Teller *et al.* [24] describe a system to compute radiosity solutions for polygonal environments larger than main memory, and Funkhouser *et al.* [15] present prefetching techniques for interactive walk-throughs in large architectural virtual environments. Very recently, Pharr *et al.* [21] give memory-coherent ray-tracing algorithms, Cox and Ellsworth [13] present application-controlled demand paging methods, and Ueng *el al.* [25] propose out-of-core streamline techniques.

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 let N denote the number of cells in the dataset, and K the number of active cells. In Marching Cubes [20], all cells are searched for isosurface intersection, and thus $O(N)$ time is needed. Techniques avoiding exhaustive scanning include using an octree [28], identifying a collection of *seed cells* and performing contour propagation from the seed cells [4, 17, 26], NOISE [19], and other nearly optimal isosurface extraction methods [23]. The first *optimal* isosurface extraction algorithm was given by Cignoni *et al.* [12], based on the following two ideas. First, for each cell, they produce an interval $I = [\min, \max]$ where $\min$ and $\max$ are the minimum and maximum of the scalar values in the cell vertices. Then the active cells are exactly those cells whose intervals contain q. Searching active cells then amounts to performing stabbing queries. Secondly, the stabbing queries are solved by using an internal-memory interval tree [14]. After an $O(N \log N)$-time preprocessing, active cells can be found in optimal $O(\log N + K)$ time.

The first *out-of-core* isosurface technique was given by Chiang and Silva [10]. They follow the ideas of Cignoni *et al.* [12], but use the I/O-optimal interval tree of [3] to solve the stabbing queries. In their follow-up paper [11], they replaced the I/O interval tree of [3] with the metablock tree [18]. With their techniques, datasets much larger than main memory can be visualized very efficiently. The major drawback is the large overhead in disk space to hold the search structure, and the disk scratch space needed to build the structure. Another out-of-core isosurface technique, based on contour propagation from seed cells, is recently proposed in [5] (where no out-of-core implementation is reported).

2 Main Techniques

In this section we present our isosurface algorithm. There are two major techniques: the *meta-cell* technique, which is used to construct *meta-cells* from dataset cells, and the *binary-blocked I/O interval tree*, which is a new I/O-optimal stabbing-query data structure, used to serve as an *indexing* structure for the meta-cells. We show the preprocessing pipeline of our overall algorithm in Fig. 1. The main tasks are as follows:

(1) Group spatially neighboring cells into *meta-cells*. The total number of vertices in each meta-cell is roughly the same, so that during queries each meta-cell can be retrieved from disk with approximately the same I/O cost. Each cell is assigned to exactly one meta-cell.

(2) Compute and store in disk the meta-cell information for each meta-cell.

(3) Compute *meta-intervals* associated with each meta-cell. Each meta-interval is an interval $[\min, \max]$, to be defined later.

(4) Build in disk a binary-blocked I/O interval tree on meta-intervals. For each meta-interval, only its $\min$ and $\max$ values and the meta-cell ID are stored in the tree, where meta-cell ID is a pointer to the corresponding meta-cell record in disk.

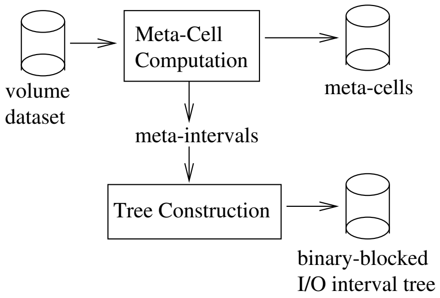


Figure 1: The preprocessing pipeline of our isosurface technique.

We describe the representation of meta-cells. Each meta-cell has a list of vertices, where each vertex entry contains its x-, y-, z- and scalar values, and a list of cells, where each cell entry contains pointers to its vertices in the vertex list. In this way, a vertex shared by many cells in the same meta-cell is stored just *once* in that meta-cell. The only duplications of vertex information occur when a vertex belongs to two cells in *different* meta-cells; in this case we let both meta-cells include that vertex in their vertex lists, so that each meta-cell has *self-contained* vertex and cell lists. We store the meta-cells, one after another, in disk.

The purpose of meta-intervals for a meta-cell is analogous to that of interval for a cell: a meta-cell is *active*, *i.e.*, intersected by the isosurface of q, if and only if one of its meta-intervals contains q. Intuitively, we could just take the minimum and maximum scalar values among the vertices to define the meta-interval (as cell intervals), but such big range would contain *gaps** in which no cell interval lies. Therefore, we break such big range into pieces, each a meta-interval, by the gaps. Formally, we define the *meta-intervals* of a meta-cell as the *connected components* among the intervals of the cells in that meta-cell. With this definition, searching active meta-cells amounts to performing stabbing queries on the meta-intervals. The query pipeline of our overall algorithm is shown in Fig. 2. We have the following steps:

(1) Find all meta-intervals (and the corresponding meta-cell ID's) containing q, by querying the binary-blocked I/O interval tree in disk.

(2) (Internally) sort the reported meta-cell ID's. This makes the subsequent disk reads for active meta-cells *sequential* (except for skipping inactive meta-cells), and minimizes the disk-head movements.

(3) For each active meta-cell, read it from disk to main memory, identify active cells and compute isosurface triangles, throw away the current meta-cell from main memory and repeat the process for the next active meta-cell. At the end, patch the generated triangles and perform the remaining operations in the generation phase to generate and display the isosurface.

Now we argue that in step (3) the pointer references in disk to read meta-cells are efficient, *i.e.*, there are many active cells in an active meta-cell. Intuitively, by the way we construct the meta-cells, we can think of each meta-cell as a cube, with roughly the same number of cells in each dimension. Also, by the *spatial coherence* of an isosurface, usually there are not many meta-cells that

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

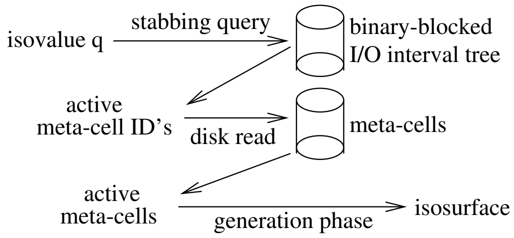


Figure 2: The query pipeline of our isosurface technique.

are cut *only through corners* by the isosurface. Thus by a dimension argument, if an active meta-cell has C cells, for most times the isosurface cuts through $C^{2/3}$ cells. This is similar to the argument that usually there are $\Theta(N^{2/3})$ active cells in an N-cell volume dataset. Then this means that we read C cells (a whole meta-cell) for every $C^{2/3}$ active cells, *i.e.*, we traverse a *thickness* of $C^{1/3}$ layers of cells, for one layer of isosurface. Therefore we read $C^{1/3} \cdot (K/B)$ disk blocks for K active cells, which is a factor of $C^{1/3}$ from optimal (B is the number of cells fitting in one disk block). Notice that when the size of meta-cells is increased, the number of duplicated vertices is decreased (less vertices in meta-cell boundaries), and the number of meta-intervals is also decreased (less meta-cells), while the number C is increased. Hence we have a *trade-off* between space and query time, by varying the meta-cell size. Since the major cost in disk reads is in *disk-head movements* (*e.g.*, reading two disk blocks takes approximately the same time as reading one block, after moving the disk head), we can increase meta-cell sizes while keeping the effect of the factor $C^{1/3}$ negligible. (We shall see the actual trade-off between disk space and query time when we present the experimental results in Section 3.)

2.1 Meta-Cell Computation

The efficient subdivision of the dataset into meta-cells lies at the heart of our overall isosurface algorithm. The computation is similar to the partition induced by a k-d-tree [6], but we do not need to compute the multiple levels. Since direct random access to vertices is very inefficient in disk, we develop a new technique that is I/O-efficient, by essentially performing external sorting a few times. We assume that the input dataset is in a general "index cell set" (ICS) format, *i.e.*, there is a list of vertices, each containing its x-, y-, z- and scalar values, and a list of cells, each containing pointers to its vertices in the vertex list. We want to partition the dataset into H^3 meta-cells, where H is a parameter we can adjust to vary the meta-cell sizes, usually several disk blocks. The final output of meta-cell computation is a single file that contains all meta-cells, one after another, each an *independent* ICS file (*i.e.*, the pointer references from cells of a meta-cell are *within* the meta-cell). We also produce meta-intervals for each meta-cell.

For simplicity, we assume that the input cell list contains cells of the same type (*e.g.*, tetrahedral cells). If this is not the case, we can first scan the cell list and put different types of cells into different cell lists. In the following, we refer to meta-cell ID's as numbers $0, 1, \cdots$ to number the meta-cells; we refer to them as *pointers* to the meta-cell positions in disk, as we previously do, only after the meta-cell computation is complete. Our meta-cell computation consists of the following steps.

1. Partition vertices into clusters of equal size. This is the *key* step in constructing meta-cells. We use each resulting cluster to define a meta-cell, whose vertices are those in the cluster, plus some *duplicated* vertices to be constructed later. Observe that meta-cells may differ dramatically in their volumes, but their numbers of vertices are roughly the same. The partitioning method is very simple. We first externally sort all vertices by the x-values, and partition them into H consecutive chunks. Then, for each such chunk, we externally sort its vertices by the y-values, and partition them into H chunks. Finally, we repeat the process for each refined chunk, except that we externally sort the vertices by the z-values. We take the final chunks as clusters. Clearly, each cluster has spatially neighboring vertices. The computing cost is bounded by three passes of external sorting. This step actually *assigns* vertices to meta-cells. We produce a *vertex-assignment* list with entries (v_{id}, m_{id}), indicating that vertex v_{id} is assigned to meta-cell m_{id}.

2. Assign cells to meta-cells and duplicate vertices. Our assignment of cells to meta-cells attempts to minimize the wasted space. The basic coverage criterion is to see how a cell's vertices have been mapped to meta-cells. A cell whose vertices all belong to the same meta-cell is assigned to that meta-cell. Otherwise, the cell is in the boundary, and a simple voting scheme is used: the meta-cell that contains the *most* vertices owns that cell, and the *missing* vertices of the cell have to be duplicated and inserted to this meta-cell. We break ties arbitrarily. In order to determine this assignment, we need to obtain for each cell, the destination meta-cells of its vertices. For in-core computation, this is easily computed by a pointer de-reference. But the out-of-core counterpart of this computation is not so simple. Our basic operation is the *join* operation (commonly used in database), using the vertex ID as the *key*, in both the cell list and the vertex-assignment list. The join operation can be performed I/O-efficiently, by externally sorting both lists by the key, and scanning through both lists to fill in the information needed [7, 9]. For example, to fill in the destination meta-cell ID of the *first* vertex in each cell, we sort the cell records in the cell list by the vertex ID's of their *first* vertices, so that the first group contains the cells whose first vertices are vertex 1, the second group contains the cells whose first vertices are vertex 2, and so on. We also sort the vertex-assignment list by vertex ID, so that we know the destination meta-cell ID's of vertex 1, of vertex 2, etc., in that sequential order. We then scan through both lists and fill in the destination meta-cell ID of the first vertex, for each cell in the cell list. We need to perform as many join operations as the degree of the cell (*i.e.*, for tetrahedra we need to perform four joins). Once all the vertex-to-meta-cell assignments have been propagated to the cell list, a single scan is enough not only to assign cells to meta-cells, but also to decide which vertices to duplicate and insert to which meta-cells. For the latter, we produce a *vertex-duplication* list with entries (v_{id}, m_{id}), indicating that vertex v_{id} has to be duplicated and inserted to meta-cell m_{id}.

3. Compute the vertex and cell lists for each meta-cell. To actually duplicate vertices and insert them to appropriate meta-cells, we first need to de-reference the vertex ID's (to obtain the *complete* vertex information) from the vertex-duplication list. We can do this by using one join operation, using vertex ID as the key, on the original input vertex list and the vertex-duplication list. Now the vertex-duplication list contains for each entry the complete vertex information, together with the ID of the meta-cell to which the vertex must be inserted. We also have a list for assigning cells to meta-cells. To finish the generation of meta-cells, we use a main join operation on these lists, using meta-cell ID as the main key. To avoid possible replications of the same vertex inside a meta-cell, we use vertex ID's as the secondary key during the sorting for the join operation. Finally, we update the vertex pointers for the cells *within* each meta-cell. This can be easily done since each meta-cell can be kept in the main memory.

4. Compute meta-intervals for each meta-cell. Since each meta-cell can fit in main memory, this step only involves in-core computation. First, we compute the interval for each cell in the meta-cell. Then we sort all interval endpoints. We scan through the endpoints, with a counter initialized to 0. A left endpoint encountered increases the counter by 1, and a right endpoint decreases the counter by 1. A "$0 \to 1$" transition gives the beginning of a new meta-interval, and a "$1 \to 0$" transition gives the end of the current meta-interval. We can easily see that the computation is correct, and the computing time is bounded by that of internal sorting.

2.2 Binary-Blocked I/O Interval Tree

Now we present our *binary-blocked I/O interval tree*. Since it is a general stabbing-query data structure, we use the general term *interval* to refer to the underlying intervals or meta-intervals being manipulated. We use *unique cell ID's* to break a tie between endpoint values. In the case of meta-intervals and meta-cells, it is easy to see that each entry of (endpoint value, meta-cell ID) is distinct. We use N to denote the total number of intervals considered, and M and B the numbers of intervals fitting in main memory and in one disk block, respectively. One I/O operation reads or writes one disk block.

Our interval tree is I/O-optimal in space, query, and preprocessing, and is an extension of the original (main memory, binary) interval tree of [14]. Our *branching factor Bf* (*i.e.*, the maximum number of children of an internal node) is increased from 2 to $\Theta(B)$, to reduce the tree height from $O(\log_2 N)$ to $O(\log_B N)$, like B-trees. We remark that the previous I/O-optimal interval tree of [3] also increases *Bf* (to $\Theta(\sqrt{B})$) to make tree height $O(\log_B N)$, but an additional type of secondary lists is introduced, which potentially increases the space by a factor of 3/2 (originally the binary interval tree has two types of secondary lists). Our tree does not introduce any new type of lists, so is simpler to implement and also is more space-efficient in practice.

2.2.1 Data Structure

Before describing our binary-blocked I/O interval tree, we first review the original (main memory) interval tree of [14]. Given a set of N intervals, such interval tree T is defined recursively as follows. If there is only one interval, then the current node r is a leaf containing that interval. Otherwise, node r stores as a key the median value m that partitions the interval endpoints into two slabs, each having the same number of endpoints that are smaller (resp. larger) than m. The intervals that contain m are assigned to node r. The intervals with both endpoints smaller than m are assigned to the left slab; similarly, the intervals with both endpoints larger than m are assigned to the right slab. The left and right subtrees of r are recursively defined as the interval trees on the intervals in the left and right slabs, respectively. In addition, each internal node u of T has two secondary lists: the *left list*, which stores the intervals assigned to u, sorted in *increasing left endpoint values*, and the *right list*, which stores the same set of intervals, sorted in *decreasing right endpoint values*. It is easy to see that the tree height is $O(\log_2 N)$. Also, each interval is assigned to exactly one node, and is stored either twice (when assigned to an internal node) or once (when assigned to a leaf), and thus the overall space is $O(N)$.

In our binary-blocked I/O interval tree, $\mathcal{T}$, each node is one disk block, capable of holding B items. We want to increase the branching factor *Bf* so that the tree height is $O(\log_B N)$. The intuition of our method is extremely simple: we *block* a subtree of the binary interval tree T into one node of $\mathcal{T}$ (see Fig. 3). In the following, we refer to the nodes of T as *small nodes*. We take the branching factor *Bf* to be $\Theta(B)$. Then in an internal node of $\mathcal{T}$, there are $Bf-1$ small nodes, each having a key, a pointer to its left list and a pointer to its right list, where all left and right lists are stored in disk.

Now we give a more formal definition of tree $\mathcal{T}$. First, we sort all *left* endpoints of the N intervals in increasing order from left to right, into set E. We use (meta-)cell ID's to break ties. Set E is used to define the keys in small nodes. Then tree $\mathcal{T}$ is recursively defined as follows. If there are no more than B intervals, then the current node u is a leaf node storing all intervals. Otherwise, u is an internal node. We take $Bf - 1$ median values from E, which partition E into *Bf* slabs, each with the same number of endpoints. We store sorted, in non-decreasing order, these $Bf-1$ median values in node u, which serve as the keys of the $Bf - 1$ small nodes in u. We *implicitly* build a subtree of T on these $Bf - 1$ small nodes, by a *binary-search scheme*: the root key is the median of the $Bf - 1$ sorted keys, the key of the left child of the root is the median of the lower half keys, and the right-child key is the median of the upper half keys, and so on. Now consider the intervals. The intervals that contain one or more keys of u are assigned to u. In fact, each such interval I is assigned to the *highest* small node (in the subtree in u) whose key is contained in I; we store I in the corresponding left and right lists of that small node. For the remaining intervals, each has both endpoints in the same slab and is assigned to that slab. We recursively define the *Bf* subtrees of node u as the binary-blocked I/O interval trees on the intervals in the *Bf* slabs.

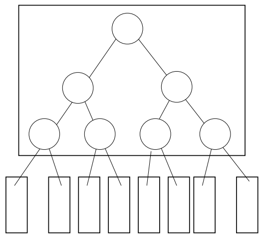


Figure 3: Intuition of binary-blocked I/O interval tree $\mathcal{T}$: each circle is a node in the binary interval tree T, and each rectangle, which blocks a subtree of T, is a node of $\mathcal{T}$.

Notice that with the above binary-search scheme for implicitly building a (sub)tree on the keys stored in an internal node u, *Bf* does not need to be a power of 2 — we can make *Bf* as large as possible, as long as the $Bf - 1$ keys, the $2(Bf - 1)$ pointers to the left and right lists, and the *Bf* pointers to the children, etc., can all fit into one disk block. As a comparison, in the I/O interval tree of [3], each internal node has $\Theta(Bf)$ left lists, $\Theta(Bf)$ right lists, and additional $\Theta(Bf^2)$ *multi* lists, and thus *Bf* is taken as $\Theta(\sqrt{B})$. Also, an interval can be stored up to three times. It is easy to see that our tree $\mathcal{T}$ has height $O(\log_B N)$, and the overall space complexity is optimal $O(N/B)$ disk blocks.

2.2.2 Query Algorithm

Our query algorithm for the binary-blocked I/O interval tree $\mathcal{T}$ is very simple and mimics the query algorithm for the binary interval tree T. Given a query point q, we perform the following recursive process starting from the root of $\mathcal{T}$. For the current node u, we read u from disk. Now consider the subtree T_u implicitly built on the small nodes in u by the binary-search scheme. Using the same binary-search scheme, we follow a root-to-leaf path in T_u. Let r be the current small node of T_u being visited, with key value m. If $q = m$, then we report all intervals in the left (or equivalently, right) list of r and stop. If $q < m$, we scan and report the intervals in the left list of r, until the first interval with left endpoint larger than q is encountered. Recall that the left lists are sorted by increasing left endpoint values. After that, we proceed to the left child of r in T_u. Similarly, if $q > m$, we scan and report the intervals in the right list of r, until the first interval with right endpoint smaller than q is encountered. Then we proceed to the right child of r in T_u. At the end, if q is not equal to any key in T_u, the binary search on the $Bf - 1$ keys locates q in one of the *Bf* slabs. We then visit the child node of u in $\mathcal{T}$ which corresponds to that slab, and apply the same process recursively. Finally, when we reach a leaf node of $\mathcal{T}$, we check the $O(B)$ intervals stored to report those that contain q, and stop. Although the tree height is $O(\log_B N)$, in the worstcase we might need to perform a total of $O(\log_2(N/B) + K/B)$ I/O operations for a query. We can improve this bound to optimal $O(\log_B N + K/B)$ I/O's by using the *corner structures* [18]; we omit the details here in order to stay within the page limitations.

2.2.3 Preprocessing Algorithm

We describe our preprocessing algorithm for building the tree $\mathcal{T}$. It is based on the *scan and distribute* paradigm originated from the *distribution sweep* I/O technique [8, 16]. Our algorithm follows the definition of $\mathcal{T}$ given in Section 2.2.1. In the first phase, we sort (using external sorting) all N input intervals in increasing *left* endpoint values from left to right, into a set S. We use (meta-)cell ID's to break a tie. We also copy the *left* endpoints, in the same sorted order, from S to another set E. The set E is used to define median values to partition E into slabs throughout the process.

The second phase is a recursive process. If there are no more than B intervals, then we make the current node u a leaf, store all intervals in u and stop. Otherwise, node u is an internal node. We first take the $Bf - 1$ median values from E that partition E into Bf slabs, each containing the same number of endpoints. We store sorted in u, in non-decreasing order from left to right, these median values as the keys in the small nodes of u. We now scan all intervals (from S) to distribute them to node u or to one of the Bf slabs. We maintain a temporary list for node u, and also a temporary list for each of the Bf slabs. For each temporary list, we keep one block in the main memory as a *buffer*, and keep the remaining blocks in disk. Each time an interval is distribute to node u or to a slab, we put that interval to the corresponding buffer; when a buffer is full, it is written to the corresponding list in disk. The distribution of each interval I is carried out by the *binary-search scheme* described in Section 2.2.1, which implicitly defines a balanced binary tree T_u on the $Bf - 1$ keys and the corresponding small nodes in u. We perform this binary search on these keys to find the highest small node r whose key is contained in I, in which case we assign I to small node r (and also to the current node u), by appending the small node ID of r to I and putting it to the temporary list for node u, or to find that no such small node exists and both endpoints of I lie in the same slab, in which case we distribute I to that slab by putting I to the corresponding temporary list. When all intervals in S are scanned and distributed, each temporary list has all its intervals, automatically sorted in increasing left-endpoint values. Now we sort the intervals belonging to node u by small node ID as the first key and the left-endpoint value as the second key, in increasing order, so that intervals assigned to the same small node are put together, sorted in increasing left-endpoint values. We read these intervals to set up the left lists of all small nodes in u. Then we copy each such left list to its corresponding right list, and sort the right list by decreasing right-endpoint values. The corner structure for node u, if we want to construct, can be built at this point. This completes the construction of node u. Finally, we perform the process recursively on each of the Bf slabs, using the intervals in the corresponding temporary list as input, to build each subtree of node u.

We remark that in the above *scan and distribute* process, instead of keeping all intervals assigned to the current node u in *one* temporary list, we could maintain $Bf - 1$ temporary lists for the $Bf - 1$ small nodes of u. This would eliminate the subsequent sorting by small node ID's (which is used to *re-distribute* the intervals of u into individual small nodes). But for the actual implementation, our method is used to address the system issue that a process cannot open too many files simultaneously, while avoiding a blow-up in disk scratch space. It can be shown that the overall preprocessing takes nearly optimal $O(\frac{N}{B}\log_B N)$ I/O's. We can also make the bound optimal ($O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$, as the external sorting bound [1], where M is the number of intervals fitting in main memory) by the tree-height conversion method in [10].

3 Experimental Results and Analysis

In this section, we attempt to experimentally assess the advantages and shortcomings of our new technique, in particularly as compared to our previous work [10, 11]. We consider five datasets in our study. Four of them were used in our previous papers [10, 11], and a new, larger dataset, Cyl3 with about 5.8M cells has been added to our test set. Table 1 summarize their properties.

Our experimental set-up is similar to the one we used in [10, 11]. Our benchmark machine is an off-the-shelf PC: a Pentium Pro, 200MHz with 128M of RAM, and 768M of swap space. Using Linux, we booted the machine in two different configurations, with 64M and 128M of main memory. For preprocessing, we used the machine with only 64M of main memory, and for computing the isosurfaces we varied the amount of main memory. Because of the usage of the operating system and X-windows, we estimate that only half to two thirds of main memory was actually available for computations.

Meta-cell Generation

Computing the meta-cells is a core operation of our technique, and one of the main differences between our new method and [10, 11]. Meta-cell generation is basically divided into five parts: (1) normalizing the original file, which involves separating the vertices and each type of cells into their own files, (2) mapping the vertices into meta-cells, (3) mapping the cells into meta-cells, (4) completing the meta-cell information and writing to the meta-cell file, and (5) computing the meta-intervals used for indexing. As can be seen in Tables 2 and 3, meta-cell generation can be expensive, in particular for large datasets, such as Cyl3. The main reason for this is that we do not assume any kind of pre-determined spatial coherence in our input, forcing us to perform several *external sorts* on different *keys*, over very large files.

There are several ways to make this faster. The most obvious would be to use a larger machine with enough main memory for the computation. In this case, the geometric hashing we are using becomes trivial, and clearly can be performed very efficiently. A less obvious observation is that due to the fact that we are essentially performing a global geometric hashing operation, given information about the relative positions of the vertices (basically, rough bounding boxes), the computation can be performed more efficiently. For instance, if we already have some meta-cell subdivision, we do not need to recompute another one from scratch, instead it is possible to either refine a coarser subdivision, or join multiple fine subdivisions into coarser ones. We conjecture (though have not tried yet) that we should be able to manage multi-gigabyte scientific datasets computed in distributed memory parallel machines, by running our meta-cell generation on each piece individually, since, in general, they are organized in mostly disjoint chunks of spatially coherent data.

Tables 2 and 3 give some important performance statistics. In Table 2, a global view of the performance of our technique can be seen on four different datasets. It is interesting to note that by varying the number of meta-cells, we can effectively control the disk space overhead. In general, the smaller number of cells in a meta-cell, the faster the querying and fetching, and also the more accurate the isosurface search. In Table 3 we vary the number of meta-cells used for the Delta dataset. This table shows that our algorithm scales well with increasing meta-cell sizes. The most important feature is the linear dependency of the querying accuracy versus the disk space overhead. For example, using a total of 146 meta-cells (at 7% disk overhead), for a given isosurface, we need 3.34s to find the active cells. When using 30,628 meta-cells (at 63% disk overhead), we only need 1.18s to find the correct cells. Basically, the more meta-cells, the more accurate our active-cell searchers, and the less amount of data we need to fetch from disk.

Name	# of Cells	Original Size	Binary Size
Blunt Fin	187K	5.8M	3.7M
Comb. Chamber	215K	6.8M	4.2M
Liquid Oxygen Post	513K	16.4M	10M
Delta Wing	1M	33.8M	19.4M
Cyl3	5.8M	337M	152M

Table 1: A list of the datasets used for testing. Original size is the file size as an ASCII ".scalar" or ".vtk" file.

	Blunt	Chamber	Post	Cyl3
# of meta-cells	737	1009	1870	27896
Normalization	3.1s	3.5s	8.8s	158s
Vertex Map	2.8s	3.6s	8.3s	382s
Cell Map	19s	24.1s	58.1s	783s
Meta-Cell Info	20.8s	24s	67.8s	1179s
Meta-Intervals	4.2s	4.8s	11.7s	147s
Total	50s	60s	154.8s	3652s
Original Size	3.65M	4.19M	10M	152M
Meta-Cell Size	4.39M	5M	12.2M	271M
Avg Vertex	118.1	102.1	133.2	399
Avg Cell	254.2	213.1	274.5	208
Increase	20%	21%	22%	78%
BBIO_Tree (size)	29K	28K	84K	1.7M
BBIO_Tree (time)	0.35s	0.67s	1.23s	43s

Table 2: Statistics for preprocessing isosurfaces on different datasets. First, we show the number of meta-cells used for partitioning the dataset, followed by the times for each step of the meta-cell computation and its total time. Secondly, the original dataset size and the size of the meta-cell file are shown. We also show the average numbers of vertices and of cells per meta-cell, and the overall increase in storage. Finally, we show the size (in bytes) of the BBIO tree and its construction time.

An interesting point is that the more data fetched, the more work (and main memory usage) for the isosurface generation engine. By paying the 63% disk overhead, we only need to fetch 16% of the dataset into main memory, which is clearly a substantial saving.

Figs. 4a and 5a show the bounding boxes of two meta-cell decompositions on the same dataset. The dataset used was a low resolution version of the dataset Cyl3 used in Tables 2 and 4 to avoid cluttering. One can see from the two figures that our algorithm samples the higher-resolution areas with more meta-cells, while using lower numbers of meta-cells in areas with less details.

Meta-cell Indexing

The number of meta-intervals generated is directly proportional to the number of meta-cells. The size of the interval tree (denoted by BBIO tree) increases when the dataset gets larger (*e.g.*, for the Cyl3 dataset shown in Table 2 is 1.7M), and may be well beyond the main memory size for larger dataset. This is the major reason why we need the BBIO tree, to ensure the scalability for a large number of meta-intervals being indexed. In addition, as opposed to in-core indexing structures, we need not spend the time to build/load the tree in main memory every time the process starts to run. Tables 2 and 3 also contain information related to the construction of the trees, and their respective sizes. Having the indexing data structure separated from the meta-cells is important, since in several applications multiple indexing structures can point to the same set of meta-cells. For instance, in handling time-varying datasets, one can keep a single copy of the geometric data (in the meta-cells), and have multiple BBIO trees for indexing different time steps.

Isosurface Extraction Queries

Table 3 already presents some limited querying information that demonstrates the effectiveness of the meta-cell blocking as a function of the disk space overhead. Particularly interesting are the data given in Table 3, which shows how the isosurface extraction cost changes with meta-cell sizes. As the number of meta-cells increases (and the disk space overhead also increases due to more vertex replications), the query time decreases. This shows that our technique provides a smooth trade-off between disk space overhead and querying performance. A visual representation of this effect can be seen from Figs. 4b and 5b, which show the bounding boxes of the fetched (*i.e.*, active) meta-cells during the query of the isosurface with value 0.0623775 in the Cyl dataset. Figs. 4c and 5c show the actual isosurfaces superimposed to the active meta-cells. Even for this down-sampled dataset and the coarse meta-cells, one can see the effect of more meta-cells in culling away larger portions of the dataset not containing the isosurface. Note the difference between Figs. 4b and 5b in the middle of the dataset where the cells do not get touched. As the number of meta-cells increases, the active meta-cells are refined and resemble the isosurface.

It is important to study the overall performance of the isosurface extraction query pipeline. Ideally, we would like to compare four different techniques: (1) the plain Vtk [22] pipeline; (2) an output-sensitive in-core isosurface algorithm (such as the one presented in [12]); (3) our previous work [10, 11]; (4) our new algorithm. Unfortunately, we do not have (2)†. With respect to the comparisons with (3) [10], we will not be able to compare for the Cyl3 dataset, since we would need over 2.4GB of disk to perform the preprocessing (and several hours).

Table 4 summarizes our benchmarks. Points worth noting:

- Our previous technique, `ioQuery` [10], performs better than both `mcQuery` (our new code) and `vtkIso` (the pure Vtk code) in all cases. This is not really a surprise, since `ioQuery` performs an exact search, only bringing active cells into main memory. Thus, it does not waste either disk bandwidth or main memory space. Unfortunately, as we pointed out before, `ioQuery` is not practical, since it uses about 8 times as much disk space as the original dataset to keep the search structure, and it needs 16 times as much disk scratch space for preprocessing.

- Our new querying code, `mcQuery`, performs better than `vtkIso` for most examples. In particular, for Cyl3, it is over 20 times faster than pure Vtk, and even in cases where there is enough main memory such as for the Delta dataset, with only 63% disk overhead, it is about five times faster than Vtk. In fact, in some cases (such as for Post and Delta), we are able to finish querying while Vtk is still reading the dataset.

One last note about the implementation. Some might be wondering how come Vtk needs so much main memory to compute isosurfaces. In fact, it might require two to three times as much main memory as the original dataset. Without further study, we can only speculate. There are several main memory overheads for isosurface calculation, besides the isosurface itself. For instance, one

†We believe techniques such as [12] have active cell search times at least comparable to the ones we have, but in general, these other techniques need the whole dataset to be loaded into main memory, and the preprocessing has to be done each time the dataset is loaded. Also, the indexing data structures increase the amount of main memory needed (if only by a small amount), thus making these methods less likely to be used for very large datasets.

# of meta-cells	146	361	1100	2364	3600	8400	30628
Total Time	618s	427.4s	346s	331s	331s	347s	376s
Meta-Cell Size	20.8M	21.5M	22.6M	23.7M	24.6M	26.7M	31.7M
Avg Vertex	2032.8	940.1	370.4	202.8	148.2	79.3	31.4
Avg Cell	6888.1	2785.8	914.2	425.4	279.35	119.7	32.8
Increase	7%	10%	16%	22%	26.9%	37.9%	63%
BBIO_Tree (size)	4K	16K	48K	112K	168K	640K	1.7M
BBIO_Tree (time)	0.42s	0.61s	1.51s	1.94s	3.78s	13.1s	31.9s
Query (act)	49.3K	49.3K	49.3K	49.3K	49.3K	49.3K	49.3K
Query (fetch)	704K	560K	418K	345K	320K	247K	167K
Query (mc)	87	189	414	754	1094	1996	4923
Perc. (mc)	59%	52%	37%	31%	30%	23%	16%
Query Time	3.34s	2.76s	2.09s	1.82s	1.73s	1.5s	1.18s

Table 3: Statistics for preprocessing and querying isosurfaces on the Delta dataset (original binary file size 19.4M). The entries for preprocessing are as defined in Table 2. We also show the performance of a representative isosurface query with 64M of RAM: number of active cells ("act"), number of cells fetched ("fetch"), number of fetched (*i.e.*, active) meta-cells ("mc"), the ratio between the numbers of active and overall meta-cells ("Perc. (mc)"), and finally the time for finding the active cells (the time for actual isosurface generation is not included).

is the Vtk "locator" class, which is used to avoid outputting multiple vertices for the same spatial location.

4 Conclusions

In this paper we present a new out-of-core algorithm for output-sensitive isosurface extraction. In our tests, our algorithm has shown to be both robust and effective in optimizing isosurface queries. Regardless of the size of the dataset, our techniques provide a cost-effective method to speed up isosurface extraction from volume data. The actual code can be made much faster by fine tuning the disk I/O. This is an interesting but hard and time-consuming task, and might often be non-portable across platforms, since the interplay among the operating system, the algorithms, and the disk is non-trivial to optimize. We believe that a substantial speed-up can be achieved by optimizing the external sorting and the file copying primitives.

In the process, we developed two new techniques of independent interest. First, our binary-blocked I/O interval is easier to implement, and uses less disk space than the existing external-memory stabbing-query data structures. Secondly, the technique we use to compute the meta-cells has a wider applicability in the preprocessing of general cell structures larger than main memory. For example, one could use our technique to break polyhedral surfaces larger than main memory into spatially coherent sections for simplification, or to break large volumetric grids into smaller ones for rendering purposes.

We believe this work brings efficient out-of-core isosurface techniques closer to practicality. One remaining challenge is to improve the preprocessing times for large datasets, which, even though is much lower than the ones presented in [10, 11], is still fairly costly.

Acknowledgments

Yi-Jen Chiang was supported in part by NSF Grant DMS-9312098. The work of Cláudio T. Silva was partially supported by Sandia National Labs and the Dept. of Energy Mathematics, Information, and Computer Science Office, and by NSF Grant CDA-9626370.

References

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

[2] 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.

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

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

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

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[7] Y.-J. Chiang. Dynamic and I/O-efficient algorithms for computational geometry and graph problems: theoretical and experimental results. Ph.D. Thesis, Technical Report CS-95-27, Dept. Computer Science, Brown University, 1995.

[8] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Computational Geometry: Theory and Applications*, 9(4):211–236, 1998.

[9] 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.

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

[11] Y.-J. Chiang and C. T. Silva. Isosurface extraction in large scientific visualization applications using the I/O-filter technique. Technical Report, University at Stony Brook, 1997.

[12] 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.

[13] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proc. IEEE Visualization*, pages 235–244, 1997.

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

[15] T. A. Funkhouser, S. Teller, C. H. Séquin, and D. Khorramabadi. Database management for models larger than main memory. *Presence: Teleoperators and Virtual Environments*, Vol.5, No.1, 1996.

	Blunt	Chamber	Post	Delta	Cyl3
# of meta-cells	737	1009	1870	30628	27896
mc disk overhead	20%	21%	22%	63%	78%
Iso value	0.67	0.30	0.10	0.21	0.062
Act. cells	20K	37K	16.4K	49K	100K
% fetched	50%	74%	38%	17%	19%
Fetched cells	93K	160K	193K	167K	1.1M
`mcQuery - 64MB`	**4.8s**	**6.97s**	**6.7s**	**16.4s**	**60s**
BBIO	0.1s	0.01s	0.1s	0.1s	0.3s
Disk I/O	2.1s	0.74s	0.9s	1.6s	11.4s
Iso comp.	2.68s	6.2s	5.8s	14.7s	48s
`mcQuery - 128MB`	**3.29s**	**6.8s**	**6.6s**	**15.5s**	**38s**
BBIO	0.1s	0.1s	0.05s	0.1s	0.1s
Disk I/O	0.53s	0.92s	0.82s	1s	11.3s
Iso comp.	2.75s	5.86s	5.78s	14.4s	26.8s
`vtkIso - 64MB`	**5.7s**	**8.1s**	**124s**	**432s**	**2032s**
Disk I/O	4.02s	4.39s	10.8s	23s	428s
Iso comp.	1.69s	3.69s	113s	409s	1604s
`vtkIso - 128MB`	**5.7s**	**8.13s**	**123s**	**425s**	**1337s**
Disk I/O	4.04s	4.41s	11.3s	21.6s	255s
Iso comp.	1.73s	3.72s	112s	403s	1082s
`ioQuery - 64MB`	**1.7s**	**2s**	**1.7s**	**11.7s**	**NA**
Disk I/O	0.5s	0.7s	0.5s	1.7s	NA
Iso comp.	1.2s	1.3s	1.2s	10s	NA
`ioQuery - 128MB`	**1.5s**	**1.5s**	**1.4s**	**11.8s**	**NA**
Disk I/O	0.3s	0.2s	0.2s	1.7s	NA
Iso comp.	1.2s	1.3s	1.2s	10s	NA

Table 4: Statistics for querying isosurfaces on different datasets using 3 different codes: `mcQuery`, `vtkIso`, and `ioQuery`, under two different main memory configurations (64M and 128M). On the top, we specify the datasets, the total number of meta-cells, the meta-cell disk space overhead, the isosurface value being queried, the number of active cells for the particular isovalue, the percentage of the dataset that was fetched during querying, and the actual number of fetched cells. We highlight in bold the overall isosurface generation time for each run. Below we break the times up into its major components. "Iso comp." is always the time to actually compute the isosurface (depending on the method, the number of cells being used for the computation varies). For `mcQuery`, BBIO is the time it takes to query the BBIO tree; Disk I/O is the time to bring the active meta-cells into main memory. For `vtkIso`, Disk I/O is the time to read the dataset from disk. For `ioQuery`, Disk I/O is the time to search (and fetch at the same time) the active cells from disk. Note that `mcQuery` reduces the disk space overhead of `ioQuery` by more than one order of magnitude.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Proceedings of SIGGRAPH '87*, pages 163–169, July 1987.

[21] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Proceedings of SIGGRAPH '97*, pages 101–108, August 1997.

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

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

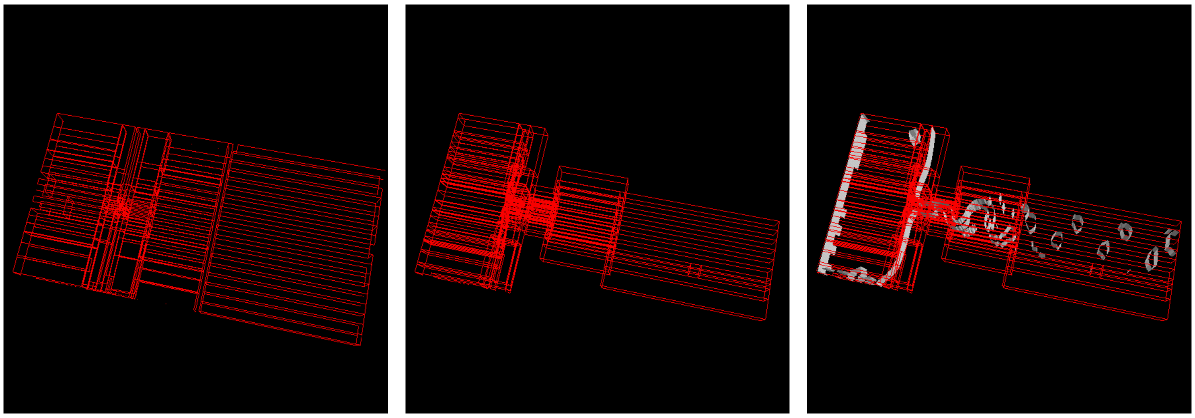
[24] S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan. Partitioning and ordering large radiosity computations. *Proceedings of SIGGRAPH '94*, pages 443–450. July 1994.

[25] S. K. Ueng, K. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, 1997.

[26] 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.

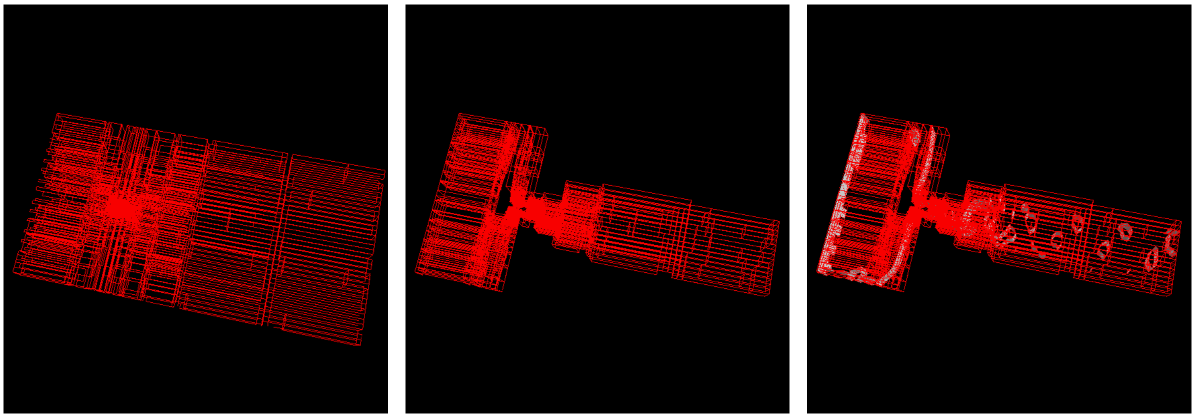
[27] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995.

[28] 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.



(a) (b) (c)

Figure 4: Illustration for the distribution of 6^3 meta-cells: (a) the bounding boxes of the meta-cells; (b) the bounding boxes of the fetched meta-cells during a query; (c) the fetched meta-cells superimposed with the isosurface.



(a) (b) (c)

Figure 5: Illustration for the distribution of 10^3 meta-cells in the same dataset as in Fig. 4.