The Visual Computer manuscript No.
(will be inserted by the editor)

Chuan-kai Yang · Tzi-cker Chiueh

Integration of Volume Decompression and Out-of-Core Iso-Surface Extraction from Irregular Volume Data

Abstract Volume datasets tend to grow larger and larger as modern technology advances, thus imposing a storage constraint on most systems. One general solution to alleviate this problem is to apply volume compression on volume datasets. However, as volume rendering is often the most important purpose why a volume dataset was generated in the first place, we must take into account how a volume dataset could be efficiently rendered when it is stored in a compressed form. Our previous work [21] has shown that it is possible to perform an *on-the-fly direct volume rendering from irregular volume data*. In this paper, we further extend that work to demonstrate that a similar integration can also be achieved on *iso-surface extraction* and volume decompression for irregular volume data. In particular, our work involves a dataset decomposition process, where instead of a *coordinate-based* decomposition used by conventional out-of-core iso-surface extraction algorithms, we choose to use a *layer-based* structure. Each such layer contains a collection of tetrahedra whose associated scalar values fall within a specific range, and could be compressed independently to reduce its storage requirement. The layer structure is particularly suitable for *out-of-core iso-surface extraction*, where the required memory exceeds the physical memory capacity of the machine that the process is running on. Furthermore, with this work, we can perform *on-the-fly iso-surface extraction during decompression*, and the computation only involves the layer that contains the query value, rather than the entire dataset. Experiments show that our approach could improve the performance up to ten times when compared with the results based on the traditional coordinate-based approaches.

Chuan-kai Yang
Department of Information Management
National Taiwan University of Science and Technology
43 Keelung Road, Section 4, Taipei, 106, Taiwan
E-mail: ckyang@cs.ntust.edu.tw

Tzi-cker Chiueh
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794, USA
E-mail: chiueh@cs.sunysb.edu

Keywords Irregular Grids, Tetrahedral Mesh Compression, Iso-Surface Extraction, Volume Rendering, Out-of-Core

1 Introduction

Modern sensors/simulators can offer more and more accurate sampling in both spatial and temporal dimensions, thus leading to volume datasets with enormous sizes. To reduce the storage requirement for volume datasets, volume compression techniques are often used. However, as volume rendering would eventually be performed on a volume dataset, how to efficiently render a volume dataset which is stored on disks in a compressed form becomes an important issue. A naive approach would be to defer the entire rendering task until the decompression process is complete. Although simple and flexible, this approach introduces an undesired long latency for the renderer and the required memory for execution cannot take advantage of the volume compression at all. Previously we have done similar research on combining direct volume rendering with volume decompression [21]. In this paper, we further extend our work to *indirect volume rendering* (i.e. *iso-surface extraction*) and propose a scheme to integrate the volume decompression and iso-surface extraction into a single pipeline, thus minimizing both the rendering latency and the required memory.

As volume datasets grow larger and larger, memory usage may also become a serious issue, that is, the iso-surface extraction process may become *out-of-core*, i.e., the required memory for execution exceeds a system's physical memory capacity. To deal with this, one of the common conventional approaches is to decompose a dataset into partitions based on its coordinates, and at run time each partition can be loaded into memory for iso-surface extraction, while the final result is the union of all the iso-surfaces extracted from each partition. Rather than applying conventional approaches, this work employs a different way of decomposing a volume dataset into *layers*, where each layer corresponds to a collection of tetrahedra whose associated scalar values fall within a specific range. We argue that for some datasets, this decomposition is more "friendly" to the iso-surface ex-

traction process than the conventional ones, because at run time, for any query, *at most one layer is needed* for extracting the corresponding iso-surface(s). Note that in the ensuing sections, *range-based*, *value-based* and *layer-based* partitionings all represent the same scheme where the decomposition is according to the scalar values associated with grid cells, while *space-based* and *coordinate-based* partitionings represent another scheme where the decomposition is according to the coordinates of the grid cells. The main contribution of this paper is not only on proposing a layer-based partitioning framework, but also on how a balanced partitioning, in terms of number of tetrahedra, can be achieved. The latter may seem simple for now, but will be shown to be a surprisingly non-trivial task in section 3.

The rest of the paper is organized as the following. Section 2 reviews the literature related to this work. Section 3 details the main ideas and procedures on how to convert a manifold tetrahedral mesh into a desired number of manifold sub-meshes. Section 4 evaluates our approaches and compares them against some traditional approaches, while section 5 concludes this work and points out some potential future directions.

2 Related Work

An iso-surface extraction algorithm extracts surfaces from a volumetric dataset when given an iso-value or iso-density of interest. The given datasets can be either regular or irregular. Many different iso-surface extraction algorithms have been proposed. Lorensen et al. [14] pioneered the iso-surface extraction research with the proposal of *Marching Cubes* algorithm on regular datasets. *Marching Tetrahedra* algorithm is employing a similar idea to marching cubes, but instead of operating on cubes, it operates on tetrahedra.

There are essentially two phases in iso-surface extraction. The first phase is the search for intersecting cells. The second phase is the generation of triangles. While the second phase can be sped up by using a table-lookup and thus leaves little room for further improvement, the first phase can be accelerated significantly by using different data structures and/or searching algorithms. There are basically three searching approaches: space-based, range-based and surface-based. The first approach partitions the input dataset based on cells' spatial coordinates, such as the octree decomposition scheme proposed by Wilhelms et al. [20]. The second approach also employs data partitioning but is based on the scalar values associated with cells. Gallagher's *span-filtering* is one such method [7]. Livnat et al.'s approach, though also a range-based approach, performs its partitioning in a rather different *min-max* domain, to achieve better efficiency [13]. The last approach tries to grow iso-surface(s) using the adjacency from a much smaller *seed-set*, which is identified in the preprocessing time, and methods by Itoh et al. [11,12], Bajaj et al. [1,2] and Hung et al. [10] are all falling into this category.

Using a range-based approach, Cignoni et al. [5] demonstrated how to use the concept of *interval trees* to answer an iso-surface query, which is essentially a *stabbing query*, and the original idea came from Edelsbrunner [6] and McCreight [16]. This method could achieve an optimal time complexity for searching cells intersecting with the desired iso-surface(s). Chiang et al. [3] applied this idea but changed the *branching factor* to make the size of each node in the interval tree no larger than a disk block to significantly reduce the number of disk I/Os and the memory required for query processing. Later Chiang et al. [4] extended their work to perform *out-of-core* iso-surface extraction as well by partitioning the original grids into *meta-cells* and used a KD-tree decomposition to make sure each meta-cell fit into the memory. Ma et al. [15] proposed a decomposition scheme where each partition though also called *layer*, is derived based on the spatial information.

3 Layered Structures for Out-of-Core Iso-Surface Extraction

In this section we describe how to convert a manifold tetrahedral mesh into a desired number of manifold sub-meshes of approximately the same size, and each sub-mesh could be further compressed to save storage. We first concentrate on how the initial equal-sized sub-meshes are derived from the original mesh, then we explain how these potentially non-manifold sub-meshes could be patched into manifold ones, and finally we show how these manifold sub-meshes are compressed. All these procedures are done at the preprocessing time, while at the run time, only one sub-mesh would be required for answering a given iso-value query. Figure 1 summarizes these main procedures and their relationships in this system.

Unlike the works by Chiang et al. [3,4] and Ma et al. [15], which decompose a dataset "spatially" according to its coordinates, we argue that a decomposition of a dataset based on its scalar values may be more efficient, as iso-surface extraction is to extract surfaces for a particular *iso-value*. As will be shown later in the performance results section, an iso-surface extraction process could involve numerous partitions when employing a coordinate-based partitioning scheme, while only at most one partition is needed if a different partitioning scheme is selected.

We propose a *value-based* decomposition approach that works as follows. First, we compute the range that encompasses all scalar values. Second, this range is partitioned into sub-ranges. There are many ways to partition a range into sub-ranges. A naive way is to assume a uniform distribution of query iso-values, therefore each sub-range is of the same size. We may also assume a *normal distribution* therefore forming larger sub-ranges around the mean and smaller ones towards boundary values. Another idea is to make use of the histogram and ensure that each sub-range catches more or less the same number of tetrahedra. The last method seems most attractive because the number of sub-ranges, say N,

may guarantee that only $1/N$ of the total tetrahedra need to be touched to answer a query, assuming the number of tetrahedra that span multiple sub-ranges is relatively small. However, the implementation of this idea is amazingly difficult, because the overlapping among layers is not negligible. We have developed several heuristics to get fairly good approximated results, and these methods will be shown later.

After the boundaries of sub-ranges are determined, the third phase is the distribution phase. Each tetrahedron is distributed to the sub-range with which its data value range overlaps. As mentioned, a tetrahedron may intersect with multiple sub-ranges, but in general such cases should mostly happen at the boundary of each sub-range. Notice that the resulting sub-mesh, corresponding to each sub-range, need not be connected, even though the original mesh is. Worse yet, the sub-mesh may not be a *manifold*, even though the original mesh is. As our previous tetrahedral mesh compression algorithm([21]) cannot compress a non-manifold mesh, this requires modification to the compression and decomposition algorithms. We will discuss some possible solutions for this issue. Fourth, we build an interval tree from all sub-ranges as if we treat each sub-range as a self-contained tetrahedron. Given an iso-value, this interval tree can be used to quickly locate which sub-mesh to further explore at run time. Finally each sub-mesh could be compressed separately to further reduce the storage requirement. In the compression algorithm, the traversal order may need to be modified to favor the tetrahedra with closer scalar values, rather than our original *breadth-first* traversal strategy; however, we did not further explore this variant in this work.

Our scheme combines compression with interval tree and could proportionally restrict the search scope to a fixed proportion of the original mesh. Furthermore, when decompression and iso-surface extraction are put together, the target iso-surface(s) can be generated in the form of triangle strips because decompression algorithm itself always enumerates triangles of the sub-mesh in a "spiral" and adjacent way. The original *coordinate-based* approach, on the other hand, may need to spend extra effort stitching together triangular meshes from multiple partitions in order to obtain longer triangle strips for better rendering performance.

3.1 Layer Partitioning Algorithms

In this sub-section we describe how we partition a tetrahedral mesh into layers with each layer containing roughly the same number of tetrahedra. As stated before, each such layer corresponds to a specific range of scalar values, therefore the problem of layer partitioning is reduced to the problem of range partitioning. Here we formulate the problem as follows: given a scalar value range, say an interval $[a, b]$, and the number of desired partitions, say M, we want to find a way to partition $[a, b]$ into M sub-ranges such that each sub-range captures approximately the same number of tetrahedra, and adjacent ranges are disjoint except at the partitioning points. At first glance this may look easy but it turns out to be a difficult problem. To ease the discussion, let us define a *cut* to be a partition point where two sub-ranges meet. For example, if we are to partition a range into two sub-ranges, only one cut is needed to divide the range into two partitions. Therefore the stated problem becomes finding out the *balanced cuts* that bring roughly the same number of tetrahedra to each partition. Also for the sake of convenience, we say the range of a tetrahedron is the interval defined by the minimal and maximal scalar values associated with its four vertices.

The main difficulty of this partition problem comes from the fact that each cut has a "side effect", which makes the finding of *balanced cuts* very difficult. After a cut is made, two associated partitions or sub-ranges will be formed, and each of them will define a set of tetrahedra that intersect with its associated sub-range. However, as these two sub-ranges intersect with each other at the cutting point, all the tetrahedra whose range contain the cutting point will belong to both partitions. The number of these "overlapped" tetrahedra is not known in advance and in practice its magnitude is not negligible. This makes the estimation of the number of tetrahedra in each partition difficult, thus complicating the process of seeking for the balanced cuts. Several algorithms have been experimented, and each of them will be discussed in details.

There is a naive approach for solving this problem. Although we will not employ such a brute force method, its description will set a better stage for our ensuing discussion. Assume there are N tetrahedra, which defines N intervals or $2N$ end points, and we want to make M balanced partitions. Further assume there are N' distinct end points out of these $2N$ end points. Statistics show that usually N' is about one sixth of N. These N' distinct end points defines $N' - 1$ intervals. It is evident that within each of these $N' - 1$ intervals, excluding the end points, it does not matter where we place a cut, because the resulting partitions defined by this cut, in terms of included tetrahedra, will be the same. In addition, we could also place our cuts at the exact locations of these N' distinct end points, thus making the total number of combinations to be $C_{M-1}^{N'-1+N'}$. Apparently this number grows exponentially as the value of N increases. And it is indeed the case for actual volume datasets, where the number of tetrahedra could be more than a million. Due to this high complexity, we therefore resort to different methods to solve this problem. Several algorithms have been experimented, and each of them will be discussed next.

3.1.1 Recursive Binary Search

The most simple solution is probably to use the *divide and conquer* strategy. Although this technique does not work very well, it can be used to explain, in a more concrete way, why it is difficult to come up with a deterministic partitioning algorithm. In addition, it also serves as a foundation for all the ensuing algorithms. For simplicity, we first assume that M, the number of partitions, is of the form 2^p, where p is a positive integer. Later we will show how to remove this constraint.

Let us start with $M = 2$, the simplest form of partitioning. The goal is thus to find one cut that splits the total range into two sub-ranges, and it can be done by using a *binary search* as follows. Initially we set this interval to be the whole range to be partitioned, and we make a cut in the middle of this interval. If the numbers of intersecting tetrahedra in both partitions are the same, then we are done; otherwise if the number of tetrahedra in the right partition is smaller or larger, then we reset the new interval's right or left end point to the current middle point to find a new cut and compare the resulting two partitions after the new cut is made, and so on, until either one of the following two conditions holds: both partitions carry the same number of tetrahedra or the middle point has converged to a point. The second case could arise when there does not exist any cut that can make both partitions have exactly the same number of tetrahedra, and we are forced to make a cut that will leave the difference between the left and right partitions minimal. This indicates that there is a lower bound on the difference, in terms of number of tetrahedra, between any two possible partitions, as shown in Figure 2. In this figure, the interval to be partitioned is marked by *start* and *end*, and all other shorter horizontal lines represent the ranges of tetrahedra that are contained within this interval. As one can see here the minimal difference we can get is to place a cut (the vertical dotted line) in between of the two groups of tetrahedra. And the minimal difference is one.

For $M = 2$, after locating the best cut via a binary search, the partitioning process is complete. For $M = 2^p$, where p is greater than 1, we could *recursively* sub-divide both partitions using exactly the same algorithm until the number of partitions has been reached, and that is why will refer to this algorithm as a *recursive binary search* method. However, these recursions may destroy the balanced partitions we have maintained previously. Let us take $M = 4$ as an example, as shown in Figure 3a, at the first level we could divide the whole range into two sub-ranges with each sub-range having four tetrahedra each by making the cut marked by 1. Now at the second level, the left half can find its best cut, marked by 2, and so can the right half, marked by 3. However, since the interval distribution in the left half may be very different from that in the right half, the cuts made in both halves may intersect with different numbers of tetrahedra. As can be seen in this figure, the cut marked by 2 indeed divides two partitions marked by A and B balancedly, and so so does the cut marked by 3. But the number of tetrahedra returned to the first level from the left partitions would be 4, which comes from the sum of number of tetrahedra intersecting with partition A (two tetrahedra) and partition B (two tetrahedra). For the right partition at the first level, the corresponding number would be 6 (three tetrahedra in partition C and three in partition D), thus making the partitions un-balanced at the first level. This explains how and why a cut made locally could affect the global balanced state that we have tried to maintain so far. A better solution does exist and is shown in Figure 3b. Nevertheless this scheme still provides a way to make balanced partitions *in a very rough sense*. And all the other methods to be presented later are in fact trying to adjust these rough partitions to be more balanced. The pseudo code of this *recursive binary search* partition algorithm is outlined in Figure 4. In this figure, *starting point* and *ending point* denote the end points of a range to be partitioned into *part_number* partitions, and the function *number_of_tetra_in(x, y)* returns the number of tetrahedra intersecting with a given interval $[x, y]$. We let *part_num_left* and *part_num_right* be the corresponding partition numbers to be passed onto the left partition and right partition at the next level, while *count_left* and *count_right* the numbers of included tetrahedra in the left and right partitions.

There is one implementation issue regarding the performance. During the binary search, we need to perform the following operation repeatedly: find the number of tetrahedra intersecting with a given interval using the function *number_of_tetra_in(x, y)* in Figure 4. A linear scanning of all the tetrahedra each time to answer such a query is too expensive; instead, we first build an interval tree from all the tetrahedra's ranges, and each time for such a query we just search the interval tree to get the answer with an $\mathcal{O}(\log N)$ time complexity, where N is the number of tetrahedra. Notice the algorithm of searching for the tetrahedra intersecting with *an interval* is slightly different from the traditional one, i.e., the one searching for the tetrahedra intersecting with a *query value*, but it can be shown that their asymptotic time complexities are still the same. Overall it can be shown that by employing an interval tree during the search, the total time complexity for this *recursive binary search* approach is $\mathcal{O}(M \log N^{\log L})$, where L is the size of the entire range of scalar values to be partitioned.

For M not in the form of 2^p, we just need to change the line marked with $(\#\#)$ in Figure 4 into the following:

```
diff = count_left * part_num_right -
    count_right * part_num_left
```

and the line marked with $(**)$ into the following:

```
if count_left * part_num_right <
    count_right * part_num_left then
```

without changing the rest of the original algorithm. The modified algorithm take into account the weighting factors when two halves to be further partitioned have different partition numbers. Notice that this approach only makes *roughly balanced partitions*.

3.1.2 Simultaneous Binary Search

Since the *recursive binary search* method can only provide a very coarse solution, we have experimented with a *simultaneous binary search* approach to exhaustively probe for all plausible solutions within smaller ranges. This partitioning algorithm is remarkably similar to the original version with just a slight modification on how the recursion performs. The pseudo code is listed in Figure 5. The idea can be best explained with an $M = 4$ example, where this *simultaneous binary search* approach essentially performs binary searches

simultaneously at two levels. At the first level, instead of locating the best cut to balance the top-level two partitions before proceeding to the second level, the midpoint of its initial interval is selected as the first cut, and this cut is forwarded to the partitioning at the second level. At the second level, since this is the bottommost level, a usual binary search is performed to locate the best cut at this level, and the total number of tetrahedra, including those extra ones introduced by its internal cut, is returned to the first level. According to these return values from both halves, the top level proceeds with just *one step further in its binary search*, that is, shrinking its current interval size by half, and selecting a new cut. This newer cut is again passed to the second level and tested to see if such a cut could lead to more balanced partitions even when both partitions have their own internal partitions. If so, then we are done, otherwise one more step of binary search at the top level is performed, and so on, until the binary search process finally converges at the top level. The scenario can be easily generalized to multiple levels where we perform multiple binary search simultaneously at all levels. Notice that a convergence is guaranteed as at each level the interval size for picking its cut continues to shrink by half. By making the same change of the lines marked by (##) and (**) in Figure 5 to the following:

```
diff = count_left * part_num_right -
    count_right * part_num_left
if count_left * part_num_right <
    count_right * part_num_left then
```

then this algorithm can also handle the case where M is not in the form of 2^p.

In general, this scheme can find a very good result, and it is evident that it is tremendously faster than using the brute-force combinatorial approach, i.e., to find the best solution from all $C_{M-1}^{N'-1+N'}$ cases, assuming there are N' distinct end points. However, it is still quite slow in practice. By a simple analysis, it can be shown that the time complexity is $\mathcal{O}((M \log N^{\log L})^{\log L})$, and as N can be huge for real datasets, this *simultaneous binary search* approach may not be feasible in practice.

3.1.3 Heuristics for Balancing Partitions

As the *simultaneous binary search* approach is too slow, and recall that the *recursive binary search* approach, on the other hand, has a smaller complexity of $\mathcal{O}(M \log N^{\log L})$, our next attempt is to apply some heuristics to iteratively *post-balance* the partitions generated by the *recursive binary search* approach to obtain a better solution. The pseudo code for this *post-balancing* process is given in Figure 6, where the post-balancing task is performed within the function *balance_with_heuristics()*. The goal of this function is to minimize the *variance* of these partitions. We define *variance* to be:

$$\sum_{i=1}^{M} (p_i - \overline{p})^2 \tag{1}$$

where p_i denotes the number of tetrahedra in partition i, and $\overline{p}$ is the average of all such p_is. For convenience, among the total M partitions initially generated by the *recursive binary search* approach, we refer to *maximal partition* and *minimal partition* as the partitions that have the most and least number of tetrahedra, respectively.

Function *balance_with_heuristics()* is performed by iterations. Each time it finds the indices of the minimal and maximal partitions. Assume M_1 and M_2 denote the indices of these partitions and further assume $M_1 < M_2$. The next step is to balance only the partitions whose indices are in the range from M_1 to M_2, as these are currently the most imbalanced partitions. Function *balance_partitions_between(M_1, M_2)* does this job by performing iterations to repeatedly identify which two adjacent partitions within this range have the most difference, and apply *balance_two_partitions()* on these two partitions. This second iteration only stops when we cannot find smaller difference among all the adjacent partitions. In function *balance_two_partitions()*, we could use a binary search, as we did in the *recursive binary search* approach, to find the best cut to balance the identified two adjacent partitions. The first iteration, the iteration in function *balance_with_heuristics()*, stops only when either all the partitions have the same number of tetrahedra or the difference between the maximal and minimal partitions cannot be further minimized.

This algorithm converges much faster than the *simultaneous binary search* approach and produces good results for most datasets. However, as the number of partitions increases, and the difference between partitions decreases, this algorithm may fail to balance partitions effectively, as shown by an example in Figure 7. In this example, each partition has the same small difference with its neighbors. The difference is so small that even by calling the function *balance_two_partitions()* on the selected two partitions still cannot help to bring down the difference, as sometimes even the best cut within these two partitions cannot reduce their difference to zero, as shown previously in Figure 2. As a result, even though the overall resulting partitions are far from balanced, this algorithm still stops earlier than it should. On the other hand, this algorithm stops only when the difference between the maximal and minimal partitions cannot be further reduced, and therefore in some cases the iteration may take a long time to converge. To have a better result, this example illustrates that sometimes we need to have some *global coordination* among all partitions, or somehow it doesn't seem necessary that we seek for the locally best answers each time.

3.1.4 Simulated Annealing for Balancing Partitions

The previous observation inspires us to use a well-known optimization technique called *simulated annealing*, which can efficiently solve the optimization problems where it is very easy to get "trapped" into a *local minimum* or *local maximum* without reaching the *global minimum* or *global maximum* that an application desires. The basic idea of *sim-*

ulated annealing can be best explained from examples in thermodynamics, such as how liquids freeze and crystallize, or metals cool and anneal. For materials that are very hot and need to be cooled down slowly without intervention, nature can amazingly find a way to reach the minimal energy state, i.e., to achieve the *global minimum*. On the other hand, if we try the cool the materials down rapidly, oftentimes we will be trapped in a *local minimum* of the energy state. Imagine we have an energy function, and our goal is find the *global minimum* in this function. Many of the optimization techniques, such as the previous heuristics that we used, always try to go downhill along the energy function as far as possible, and as a result, they may end up in some valleys not containing the *global minimum*. The first algorithm that applied the idea of *simulated annealing* was given by Metropolis et al. [17] in 1953, and it used a so called *Boltzmann* probability distribution:

$$Prob(E) \sim exp(-E/\kappa T) \quad (2)$$

which allows us, even with a small probability, to move from a low energy state to a high energy state, thus making it possible to get out of a local valley and probing towards the *global minimum*. Here T is the temperature, E denotes the energy, and κ (Boltzmann's constant) is a constant converting temperature to energy. For example, the probability to change from E_1 to E_2 is: $exp[-(E_2 - E_1)/\kappa T]$. Notice that the probability can be greater than 1 when $E_2 < E_1$, which simply means the change from a higher energy state to a lower energy state is always granted. On the other hand, even when $E_2 > E_1$ it is still with a probability of $exp[-(E_2 - E_1)/\kappa T]$ that we can make a move from a lower energy state E_1 to a higher energy state E_2. As stated before, this allows the system to get out of a *local minimum* and probe for the *global minimum*. Also notice that, as T decreases over time, the associated probability of going from a lower energy state to a higher one also decreases, just as what will happen in thermodynamics. We will also use this *Boltzmann* probability distribution as our *oracle* in determining if a change should be carried out or not.

We adopted a program structure that is similar to the one in *Numerical Recipes in C* [18] to perform *post-balancing* on the partitions generated by the *recursive binary search* approach. The pseudo code is shown in Figure 8. We define E to be the variance of numbers of tetrahedra in these partitions, as shown in Equation 1. The algorithm consists of two loops. The outer one iterates 100 times while the inner one loops "*nover*" times to randomly select a cut (from the total $M - 1$ cuts) and its moving direction: left or right. The cost of moving this cut towards its selected direction, defined as $\triangle E$, is calculated and forwarded to the *simulated annealing oracle*. If the oracle says "yes", then this movement is carried out, i.e., we move the cut, leading to the updates of associated partitions, and we also increase "*nsucc*", the number of successful movement, by one. As *nsucc* accumulates to the amount of "*nlimit*", then we are done. In our implementation, the value of *nlimit* is set to be $100 * numP$, with *numP* being the number of distinct end points. After numerous trials, the initial value of T (in our code, we use t) is set to be 10^7 so that it is considerably larger than the largest $\triangle E$ usually encountered, and its value is decreased by multiplying with "*TFACTOR*", which is set to be 0.9, at the end of each of the *nover* iterations. The value of *nover* is set to be $100 * numP$.

When moving a cut leftward or rightward, what we want is to move this cut by the least amount but still affecting the associated two partitions. Recall there are N' distinct end points and $N' - 1$ intervals. If we view these $N' - 1$ intervals as open intervals, i.e., not including their ending points, then the original range to be partitioned can be viewed as a composition of all these $N' + N' - 1$ components. To move a cut by the least effective amount, it simply means this cut should move its position from its original component to the next component on its left or right. The next step is to calculate the associated cost of this movement. This could be done efficiently by associating each distinct end point with two numbers: the number of tetrahedra having this end point as their left end points, and the number of tetrahedra having this end point as their right end points. From this representation, the number of tetrahedra in the two partitions after moving the cut could be easily calculated. Recall our goal is to minimize E, the variance of the number of tetrahedra in these partitions. Let $\triangle left$ be $left_{new} - left_{old}$, and $\triangle right$ be $right_{new} - right_{old}$, where $left_{new}$ and $right_{new}$ are the new number of tetrahedra in the left and right partitions after moving the cut within, while $left_{old}$ and $right_{old}$ are their current number of tetrahedra, respectively. Then it can be shown that $\triangle E$ can be calculated by:

$$\triangle E = left^2_{new} - left^2_{old} + right^2_{new} - right^2_{old} - \\ (part_sum + \triangle left + \triangle right)^2 - part_sum^2/M$$

where $part_{sum}$ is the sum of all the number of tetrahedra in all partitions, while M is the desired number of partitions. Notice $part_{sum}$ is not equal to the total number of tetrahedra as adjacent partitions could include tetrahedra that belong to both partitions. By maintaining the value of $part_{sum}$ throughout the balancing process, each such cost of $\triangle E$ can be calculated with a *constant time complexity*.

This *simulated annealing* approach, although occasionally is slower than the heuristics-based solution, can effectively balance all the partitions. We will demonstrate its results in the performance results section.

3.2 Modification to the Mesh Compression Algorithm

After determining each sub-range, we could distribute each tetrahedron to the sub-range it intersects with. As long as a tetrahedron intersects with a sub-range, the corresponding sub-mesh, which we will also call the *layer* interchangeably in the ensuing context, should include this tetrahedron; therefore a tetrahedron may be included in multiple sub-ranges. The next step is to compress each layer. However, our tetrahedral mesh compression algorithm in [21] may not work on compressing and decompressing these layers. It is due to the following reasons. First of all, each sub-mesh may

not be *connected*. A connected mesh means for every two vertices in this mesh, there exists a *continuous path* contained within the mesh, to connect them. Second, each sub-mesh may not be a *manifold*. A *manifold* mesh requires that there exists no *singular edges* and no *singular vertices*. To understand the definition of a *singular edge* and a *singular vertex*, we need to first define what a *boundary face* is. A face is called a *boundary face* if there is only one tetrahedron using it, while an edge and a vertex on a boundary face is called a *boundary edge* and *boundary vertex*, respectively. A *singular edge* is thus defined to be a boundary edge that has more than two boundary faces using it. An example of a *singular edge* is shown in Figure 9a. Similarly, a *singular vertex* is a boundary vertex that does not have a *local neighborhood*. To be more concrete, for a boundary vertex that is *non-singular*, if one starts with any boundary face that is adjacent to this non-singular vertex, and follows the next edge-adjacent boundary face without touching any of the adjacent boundary faces twice, the traversal should eventually lead back to the original starting face, and the traversed faces will include all the boundary faces adjacent to this vertex. An example of a singular vertex is shown in Figure 9b.

It is possible that a sub-mesh could be dis-connected or a non-manifold, even though the original mesh is connected and a manifold. We demonstrate this possibility in Figure 10, where vertices are marked with their associated scalar values. The 2D example as shown in Figure 10a could demonstrate how a dis-connect sub-mesh could be formed from a originally connected mesh. Here tetrahedra are represented by triangles, and if there exists a sub-range of $[1, 2]$, then the corresponding sub-mesh is the union of tetrahedra A and E, thus making it a dis-connected sub-mesh. In the 3D example given in Figure 10b, tetrahedra A, B and C together originally form a manifold tetrahedral mesh. If there exists a sub-range of $[1, 2]$, then apparently only tetrahedra A and B will be included in the corresponding sub-mesh, which makes the edge defined by the vertices with scalar values 4 and 5 become a *singular edge*, and thus the sub-mesh is a non-manifold.

3.2.1 Outward Tetrahedral Compression

Since there is no need to represent the boundary surface separately, we choose to use the outward variant of our *lossless tetrahedral mesh compression algorithm* [21], which could also demonstrate that our compression algorithm works both outward as well as inward. The outward compression algorithm starts with an arbitrarily seed tetrahedron, and uses its four faces to form the initial surface. From the faces on this surface, more tetrahedra are enumerated and a new surface is again formed to enumerate more tetrahedra, and so on, until all the tetrahedra are visited. Several cases are distinguished when enumerating tetrahedra, as is described in details in [21].

3.2.2 Dealing with Dis-Connected Components

Notice that in the outward tetrahedral compression algorithm, as we grow from a surface to another surface for enumerating more tetrahedra, all these tetrahedra must be connected to each other, as a tetrahedron being enumerated must be adjacent to the current enumerating surface. This implies that such an algorithm cannot deal with a dis-connected mesh, as the enumeration of one connected component cannot reach a different connected component in the same mesh. To deal with a dis-connected mesh, we just need to associate a label with each tetrahedron and mark it once it is visited. As the compression for a connected component is done, through exactly the same algorithm, we just check to see if there exist tetrahedra not being visited yet. If so, one of such unvisited tetrahedra is taken as a new seed tetrahedron so that we could enumerate its associated connected component. Otherwise it means we have visited all the connected components of a dis-connected mesh.

3.2.3 Dealing with Non-Manifold Meshes

The original tetrahedral mesh compression algorithm that we proposed in [21] assumes the input mesh to be a manifold. Therefore if a generated sub-mesh is not a manifold due to range partitioning, we have to convert it into a manifold sub-mesh so that it could be compressed later. However, this conversion is non-trivial, and it usually requires a *local fix* at all the locations where *singular edges* and *singular vertices* occur.

There are two approaches to perform a local fix. The first approach, by Guéziec et al. [8] and by Rossignac et al. [19], tries to identify the manifold sub-components as large as possible from a non-manifold mesh. After that, the vertices that fall within the intersection set of all these sub-components are duplicated so that they can be treated as dis-connected manifold sub-meshes, which can be easily taken care of by the method mentioned just previously. However, we found this vertex duplication process too complex as the structure of a non-manifold mesh could get very complicated. Another approach is to patch new tetrahedra until the non-manifold region becomes a manifold region. We apply the second approach as it provides a much simpler approach compared with the first approach, especially when all non-manifold sub-meshes were originally extracted from a manifold mesh.

There are two phases in the patching process: the first phase tries to fix all the singular edges, and the second phase tries to fixes all the singular vertices. We discuss these two phases one by one. Notice that an input query sent to a particular layer of the original mesh and to the original mesh itself should obtain the same results. This implies that patching more tetrahedra to fix a singular edge should be carefully done so that we do not erroneously include tetrahedra that are not present in the original mesh. Based on this principle, the patching procedure works as follows. Recall a *singular edge* has more than two boundary faces passing through it.

Since the original mesh is a manifold, its sub-meshes formed by selecting a subset of the original set of tetrahedra must have the property that each singular edge should have an even number of boundary faces passing through it. In all our testing cases, there are always four faces passing through a *singular edge*, and the method we describe here can be easily generalized to deal with other cases where there are more than four faces passing through a singular edge. From these four faces, we first divide them into two groups. The basic idea is to find the tetrahedra in the original mesh that use these four faces. To patch as few tetrahedra as possible, we divide these four faces into pairs with each pair in one manifold component. For example, in Figure 9a, boundary faces $\triangle xys$ and $\triangle xyt$ form one pair, while boundary faces $\triangle xyu$ and $\triangle xyv$ form the other pair. Notice that in this figure if the tetrahedron defined by the four vertices x, y, t and v is present in the original mesh, we would give the inclusion of such a tetrahedron a higher priority over the inclusion of other possible tetrahedra, because patching one such tetrahedron could immediately make this edge *non-singular*. This pairing can be achieved by testing if two boundary faces are connected in a *face-adjacent* way; that is, if all the tetrahedra using these two boundary faces are face-adjacent to each other while all of them are also adjacent to this singular edge.

Note that in Figure 9a it is not necessary that faces $\triangle xys$ and $\triangle xyt$ belong to the same tetrahedron, i.e., there could be many tetrahedra in between of these two boundary faces, and all of them are face-adjacent to each other. After forming the pairs, there are four possibilities to form a tetrahedron out of these four boundary faces. As soon as one of the tetrahedra is found, the patching process could stop. However, it is possible that none of the four combinations could lead to a tetrahedron in the old mesh. If such a case happens, we will then try to process each of the boundary faces that was not a boundary face in the original mesh, so that we could locate the other tetrahedron that uses a given boundary face. To reduce the size of a patched sub-mesh, when searching the tetrahedra using these boundary faces, those tetrahedra that do not include new vertices other than the existing ones already in the sub-mesh are preferred. Only when such tetrahedra cannot be found through all the boundary faces will the tetrahedra including extra vertices be considered. Notice that the inclusion of new vertex is acceptable as it will not include any tetrahedra that were not in the original mesh. In the case when each boundary face is tested for including tetrahedra, the patching of tetrahedra does not stop until the current edge becomes non-singular. Whenever a tetrahedron from the old mesh is included in a sub-mesh, we should check if it will cause some boundary edges originally being non-singular to become singular. This could be done efficiently by only checking if all the six edges of the newly added tetrahedra are singular edges or not; and if not, fix them recursively. Notice that it is also possible the addition of a new tetrahedron may turn a non-singular vertex into a singular one. However, this will not incur any problem as we only fix all the singular vertices after we fix all the singular edges.

To patch tetrahedra for a singular vertex, we apply a very similar procedure but without the initial pairing up phase. Each boundary face is also used to find its second tetrahedron from the old mesh. If the other tetrahedron is found, then it is included in the sub-mesh. This patching process will not stop until the current vertex becomes non-singular. Tetrahedra that do not include new vertices to a sub-mesh are preferred. Whenever a new tetrahedron is included, each of its six edges is tested for being a singular edge or not, and is patched recursively if necessary. Notice that, a new tetrahedron may also cause some non-singular vertices to become singular. Therefore in addition to the inner loop to fix all the vertices currently known to be singular, an outer loop is also added to keep looking for those vertices that are just made singular. The outer loop only exits when there exist no singular vertices.

4 Performance Results

In this section we demonstrate the performance of our system. A Pentium 4 1.5GHz machine, with 512MB memory and running on Linux Redhat 7.2, is used to conduct our experiments. Note that to simulate the *out-of-core* scenario, we may restrict the usage of memory; that is, it is not the case that all of 512MB memory is available to our testing cases at all times. This could be done easily as in Linux Lilo BootLoader; for instance, we could use the following parameter setting *append="128M"* to make only 128MB memory available for the Linux operating system. Most of the testing datasets used in this study could be downloaded from NASA's website, and the rendered images from these datasets could be found in [9,21].

4.1 Comparison of Algorithms

To briefly compare the effectiveness of these four methods, we try each of the method on the following testing cases: dividing the *fighter* and *liquid oxygen post* datasets into 16 layers. In Figure 11 we compare the resulting partition sizes in terms of number of tetrahedra by using these four methods. Here the y axis represents the size of each partition, in terms of percentage with respect to the total number of tetrahedra in the original mesh. The x axis represents the partition index, i.e., the $i - th$ partition, where i ranges from 1 to the number of partitions involved. For example, in this figure, the *simulated annealing* approach generates 16 partitions for the *fighter* dataset and each of them contains about 21.5% of the total number of tetrahedra in original mesh. As shown in this figure, the *simulated annealing* approach produces as good results as the *simultaneous binary search* approach for both datasets. However, as shown in Table 1, where we compare the running time of these four methods for making this partitioning, the *simultaneous binary search* approach takes a significantly longer time to converge. In fact, for both datasets, the *simulated annealing* approach even beats the

one using heuristics in terms of running time. Since the *simulated annealing* approach can produce results as good as the *simultaneous binary search* approach and is much faster, we use it for all the following measurements.

4.2 Partitioning Overhead in Storage Cost

There are several ways to determine how many layers we should make for a given dataset. One way is to make sure only a fixed and desired percentage of tetrahedra is included in each layer. Another way is to require that the total number of tetrahedra from all partitions does not exceed a prespecified limit. We adopt the second idea and require that the total storage requirement, after applying compression on each layer, should not exceed the original dataset size in its *binary representation*. Since partitions could overlap with each other at their boundary points, and these overlappings are unknown before making the partitions, it is difficult to determine the total number of tetrahedra from all the partitions. Furthermore, the compression efficiency for each sub-mesh is also unknown beforehand, thus complicating the determination of number of partitions that we want to make. Due to this difficulty, and assuming that more partitions should produce overall more tetrahedra from all partitions, we probe for the desired number of partitions as follows. We start with the number of partitions at 2, and keep on doubling this number until the total storage requirement of all the compressed, patched sub-meshes exceeds the original mesh size in its binary representation. Assume p is the largest value such that making 2^p partitions can still keep the total storage requirement below the original dataset file size. The next step is to simply perform a binary search in the range $[2^p, 2^{p+1}-1]$ until the ideal partition number is located. In fact, this probing idea can also be applied to probe for the optimal number of partitions where we want each partition to capture at most some fixed percentage of tetrahedra.

In addition to the partitioning phase, each sub-mesh also needs to go through the patching phase, which makes sure that each sub-mesh is a manifold before being sent to the final compression phase. According to the constraint on the original file size, the desired number of partitions for each dataset is listed in Table 2. The maximal percentage of tetrahedra that need to be patched among all the layers for each dataset, is also listed in this table. Here the percentage is with respect to the initial sub-mesh size as generated by the *simulated annealing* approach. As can be seen here, the number of tetrahedra required for patching is usually very insignificant. The average compression cost, in terms of bits per tetrahedron, for each dataset, is also listed in this table. Compared with the results shown in [21], these numbers are larger. This is because the sub-meshes are smaller and fragmented, thus affecting the overall compression efficiency. We also listed the storage overhead of all layers before the final compression step is applied, which excludes the original input mesh size. For example, all the 17 layers of the *fighter* dataset will take $100\% + 263.14\% = 363.14\%$ of the size of *fighter* dataset, in its binary representation. Although these overheads seem to be high, by applying the compression on top of each layer, the total storage size could be reduced so that it is no more than the original input mesh size. In this table, the number of partitions for *combustion chamber* seems to be smaller than it should be, when compared with other datasets. This is mainly because the scalar values of this dataset *have a very random distribution*, as also can be seen from the corresponding rendered image in [21]. This leads to a significant overlapping effect, where a small number of partitions would already make their total storage size sufficiently close to the original data size. This also reflects the limitation of our system: when the volume datasets contain high variations, which cause most of the iso-surfaces to span most regions of the datasets, our approach may not provide much improvement.

4.3 Partitioning Overhead in Processing Time

Figure 12 gives a timing breakdown for generating all the layers of a dataset. As can be seen in this figure, most of the layer generation time is spent on the *simulated annealing* and tetrahedra patching steps, with compression accounting for just a minor proportion.

4.4 Tetrahedra Generation time of Out-of-Core Iso-Surface Extraction

In this sub-section, we compare the results among the following four approaches. First, a query is directly performed on the original mesh. Second, we partition each of the original six meshes into $4 \times 4 \times 4 = 64$ spatial partitions, i.e., making 4 partitions along each primary axis. A query is sent to each partition whose aggregated range of scalar values contains the query value. Notice that such a spatial partition is common when performing *out-of-core* iso-surface extraction, so that each partition can be processed within the memory. The choice of $4 \times 4 \times 4 = 64$ is rather arbitrary as their associated storage overhead is relatively small. Here the overhead is defined as the total size, in terms of number of tetrahedra, of all partitions, excluding the original dataset size, and is listed in Table 3. For example, the first entry 77.23% comes from the fact that the total number of tetrahedra from all 64 spatial partitions is equal to 177.23% of the number of tetrahedra in the original mesh. Notice that, compared with Table 2, the overhead from the space-based partitioning approach is smaller than that from the layer-based approach. This is mainly because to overlap with a spatial partition, a tetrahedron must have its coordinates intersect with the spatial partition in all x, y, and z components, thus reducing the probability of intersections. This in general leads to a smaller overhead. Third, we partition each of the six meshes into layers, and the number of layers for each dataset is listed in Table 2. Each layer is patched to a

manifold and compressed individually. At run time, only one layer (there may be more than one layers qualify) is retrieved and an *on-the-fly iso-surface extraction during decompression* step is performed exclusively on that layer. The fourth approach is the same as the third one except that we do not compress each layer but store it as a binary file on the disk. At run time, only one layer will be retrieved and searched for intersecting tetrahedra, but no decompression will be involved.

As mentioned before, iso-surface extraction includes two phases, searching for intersecting tetrahedra, and generation of triangles from those intersecting tetrahedra. The results shown in Figure 13 only include numbers from the first phase, as the generation of triangles from the intersecting tetrahedra should take roughly the same time when the resulting tetrahedra are the same for all the four approaches. Our goal is to be able to perform *out-of-core iso-surface extraction*, where an input dataset could not be completely memory-resident. For such datasets, many techniques proposed to speed up the search phase, become either useless or difficult to use. For example, the *interval tree* requires the same order of space complexity as the dataset size itself, in terms of required memory. Either building an interval tree before searching, which takes more than linear time of processing, or loading an interval tree that was pre-built in the preprocessing time, requires at least a linear time of processing. This makes the use of an interval tree to speed up the searching phase pointless. We therefore just used a linear search for all the four approaches in the searching phase.

As can be seen from this figure, our layer-based approach, or the third approach, wins in most cases. In the cases where the layer-based loses, it is mainly because of the following three reasons. First, the query values may happen to intersect with only very few number of spatial partitions. For example, in the *blunt-fin* dataset, some large scalar values occur to aggregate around a region, which happens to be contained in one spatial partition. As shown in Figure 14, the lower right spatial partition may capture a region of interest. And it is evident here that the size of a spatial partition could be as small as zero (empty partition). Therefore a query value fully contained within such a small spatial partition may get a quicker response than that from a layer-based partitioning scheme, if the number of layers is not large enough, which is also the second reason. As mentioned before, the number of layers for the *combustion chamber* dataset is just 14, which is substantially less than the number of spatial partitions, 64. Third, the extra decompression time incurs a significant overhead. As shown in Figure 13, the fourth approach not only wins in all cases, but also wins significantly in most cases. A further timing break down of the third approach is given in Figure 15, where the decompression time indeed takes a major portion of the total time. Notice that these performance numbers are measured without the memory being stressed. Since we do not have datasets that are big enough, we instead try to *limit the memory available to the system* so that some out-of-core iso-surface extractions could take place. Figure 16 demonstrates that under the environment with different memory capacity, our layer-based approaches, with or without the compression, could be more than one order of magnitude faster than the case where either no partitioning or only spatial partitioning is used.

One of the benefits of our system is the uniform size of each partition and each time *only one layer* needs to be touched for extracting the target iso-surface. On the contrary, a spatial-decomposition may fail to provide such an advantage. For example, if the *blunt-fin* dataset is to be spatially decomposed into $3 \times 3 \times 3$ partitions, then for a query value of 3, only 11.11% of the partitions need to be touched. 33.3% of the partitions need to be touched if a query value of 2 is given. However, all partitions are needed if the input query value is 1. Figure 17 further compares the four approaches in terms of the number of I/Os (just reads, no writes) performed, and the number of bytes read in for each of the four approaches. As can be seen in this table, the third approach, i.e., the layer-based approach with compression, requires the least number of I/Os and the least number of bytes to be read. However, due to its decompression overhead, it is still slower than the fourth approach. On the other hand, the fourth approach only requires more number of I/Os and more bytes to be read than the third approach. Without the decompression overhead, it is much faster than all other approaches. Therefore, in the case where storage is not a major issue and memory capacity is of a greater concern, such as the case for out-of-core iso-surface extraction, the layer-based approach without compression could provide an even better solution. Even when storage and memory do not pose as problems, the fourth approach could still give us up to 10 times performance improvement, as shown in Figure 13. If both storage and memory are concerns, then the third approach, i.e., the layer-based approach with compression, may provide a better solution.

For triangle generation, we resort to the Visualization Toolkit (VTK). The resulting tetrahedra generated in the first phase are written to a binary file in the VTK's *unstructured grid* format, and VTK could generate the interpolated triangles for the desired iso-surface(s). With a spatial decomposition, a tetrahedron may be included in more than one partitions, therefore in general the number of intersecting tetrahedra is larger than it should be. However, the polygonal renderer in VTK could take care of duplication, as triangles could be treated as *opaque* objects and duplicated tetrahedra should not generate different results. Figure 18 shows two images rendered from these different ways of decompositions. Notice that as the generated surface in general is *opaque*, it is difficult to tell if these two surfaces are really the same. To further confirm this, we write a tetrahedral mesh comparison program as follows. Each tetrahedral mesh is first converted to its *normal form*, then two *normalized* tetrahedral meshes can be compared easily, as each tetrahedral mesh can only have a unique *normal form*. A normal form of a mesh is defined as follows. All the vertices are sorted according to their coordinates, and the comparison is done by first comparing their z components, and y components should there is a tie, and x component if both

z and y components cannot be used to distinguish the two vertices. Assume there are no duplicated vertices, then the next step is to update the indices of tetrahedra with respect to this new order. The indices of each tetrahedron are also sorted. All the tetrahedra, with their vertex indices sorted, are again sorted and compared with each other by their vertex index values, starting from their smallest indices, and eventually larger ones should all the previous vertex indices are the same. For a mesh with duplicated vertices or tetrahedra, as is the case for an iso-surface extraction scheme using a spatial decomposition, we should retain only one copy of all duplicated vertices and tetrahedra. This could be done by sorting and eliminating all duplicated vertices. Duplicated tetrahedra can be easily discovered and discarded by the index sorting step.

5 Conclusion and Future Work

We have successfully integrated iso-surface extraction with volume compression. Our system first decomposes an input tetrahedral mesh into layers. Each such layer consists of tetrahedra whose scalar values falling into a particular sub-range of the scalar value range of the original mesh. For an iso-surface query, at most one such layer is required to produce the desired iso-surface results. We individually compress each of these layers to save their storage requirements. At run time, we first locate a layer that contains the query value, then perform an *on-the-fly iso-surface extraction during decompression* exclusively on that layer. This layer-based decomposition is especially useful for the purpose of *out-of-core* iso-surface extraction, as we could make a layer as small as we desire so that it can be fit into the main memory, and no other layers will be needed for outputting the correct results. In contrast, the traditional approach to handle out-of-core iso-surface extraction is to decompose a dataset spatially. Although each such partition can be made small enough to fit into the main memory, there is no guarantee that other partitions won't be touched. In the case where disk storage is not a concern but memory is, the use of layer-based partition should be even more attractive. Our system could improve the cell searching time in the iso-surface extraction process to be *up to 10 times faster*, without increasing the disk storage requirement of the input mesh. However, we must point out that for datasets containing high variations where most of the iso-surfaces span most regions of the entire datasets, our approach may not provide much improvement over the space-based approaches for obvious reasons. Therefore the dealing with general datasets would be one of our future directions. One possible solution would be to first design a *dataset variation checking system*, and once a dataset is detected to have high variation, we could resort to traditional coordinate-based partitioning scheme. Here the term *variation* could be defined the average number of tetrahedra intersecting with a given scalar value. An even better framework could be to hierarchically partition into regions with small variation, and apply this paper's approach on those partitions intersecting with a given iso-value. Though such a hierarchical method looks more complicated than the original coordinate-based approaches, the proportion of datasets gets touched will be significantly reduced, as within each intersecting partition, at most one layer is needed. And at the same time, the partitioning overheads due to overlappings could be minimized as well. There is one more concern, raised by one of the reviewers, pertaining to the use of this framework: could this work be applied to regular volume data as well? There are at least two possible solutions. The first one is to convert a regular volume dataset into an irregular one. Though simple, this approach would incur more storage overhead and slow down the rendering process. The second one is to convert the operation unit used in this work from a tetrahedron to a cubic cell (formed by 8 voxels). However, due to the sparseness of the resulting partitions, the implicitness or regularity may get destroyed, thus requiring some special data-structures. Therefore an efficiency tradeoff between storage and accessibility may exist. Nevertheless, in the future, it is still interesting to see how the data reduction due to layer partitioning could impact the iso-surface extraction process for regular volume data.

6 Acknowledgment

This work was supported in part by the National Science Council under the grant NSC 92-2213-E-011-082.

References

1. C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast Isocontouring for Improved Interactivity. In *Proceedings on 1996 Symposium on Volume Visualization*, pages 39–46, October 1996.
2. C. L. Bajaj, V. pascucci, and D. R. Schikore. Fast Isocontouring for Structured and Unstructured Meshes in Any Dimension. In *IEEE Visualization '97 Late Breaking Hot Topics*, 1997.
3. Y. Chiang and C.T. Silva. I/O Optimal Isosurface Extraction. In *IEEE Visualization '97*, pages 293–300, October 1997.
4. Y. Chiang, C.T. Silva, and W.J. Schroeder. Interactive Out-of-Core Isosurface Extraction. In *IEEE Visualization '98*, pages 167–174, October 1998.
5. P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, April 1997.
6. H. Edelsbrunner. Dynamic Data Structures for Orthogonal Intersection Queries. Technical Report Report F59, Inst. Informationsverarb.,Tech. University Graz, 1980.
7. R. S. Gallagher. Span Filtering: An Optimization Scheme for Volume Visualization of Large Finite Element Models. In *IEEE Visualization '91*, pages 68–75, October 1991.
8. A. Guéziec, F. Bossen, G. Taubin, and C. T. Silva. Effi cient Compression of Non-Manifold Polygonal Meshes. In *IEEE Visualization '99*, pages 73–80, 1999.
9. W. Hong and A. E. Kaufman. Feature Preserved Volume Simplifi cation. In *Symposium on Solid Modeling and Applications 2003*, pages 334–339, 2003.
10. C. Hung and C. Yang. A Simple and Novel Seed-Set Finding Approach for Iso-Surface Extraction. In *EuroVis 2005*, pages 125–132, June 2005.

11. 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.
12. T. Itoh, Y. Yamaguchi, and K. Koyamada. Volume Thinning for Automatic Isosurface Propagation. In *IEEE Visualization '96*, pages 303–310, October 1996.
13. Y. Livnat, H. Shen, and C. R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, March 1996.
14. W. E. Lorensen and H. E. Cline. Marching Cube: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163–169, July 1987.
15. K. Ma, G. Abla, and E. Lum. Layer Data Organization for Visualizing Unstructured-Grid Data. In *Proceedings of SPIE, Visual Data Exploration and Analysis VIII*, pages 111–120, 2001.
16. E. M. McCreight. Effi cient Algorithms for Enumerating Intersecting Invervals and Rectangles. Technical Report Report CSL-80-9, Xerox Palo Alto Res. Center, 1980.
17. N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.
18. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1997.
19. J. Rossignac and D. Cardoze. Matchmaker: Manifold BReps for Non-Manifold R-Sets. In *Proceedings of the ACM Symposium on Solid Modeling*, pages 31–41, 1999.
20. J. Wilhelms and A. V. Gelder. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.
21. C. Yang, T. Mitra, and T. Chiueh. On-the-Fly Rendering of Losslessly Compressed Irregular Volume Data. In *Proceedings on Visualization '2000*, pages 101–108, October 2000.

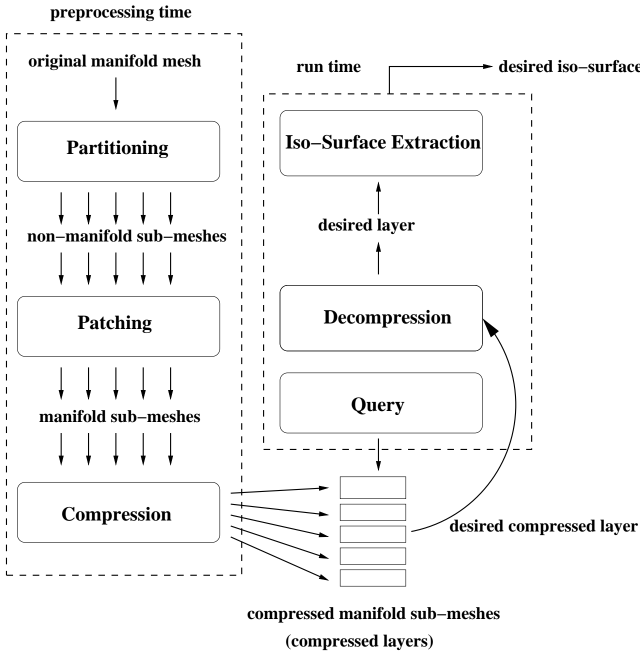


Fig. 1 *The main procedures and their relationships in our system.*

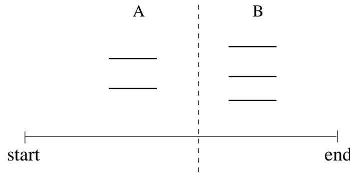


Fig. 2 *An example where no cuts could make two completely balanced partitions.*

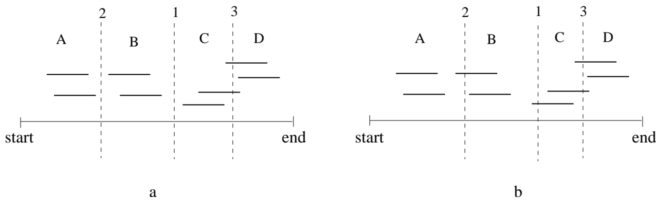


Fig. 3 *A problematic example for using the recursive binary search.*

Methods	RBS	SBS	H	SA
fighter	0.625	1225.95	61.34	17.25
post	1.34	2448.72	420.98	100.51

Table 1 *Running time (in seconds) comparison on making 16 partitions from the fi ghter and liquid oxygen post datasets, using the following four algorithms: Recursive Binary Search (RBS), Simultaneous Binary Search (SBS), Heuristics (H), and Simulated Annealing (SA).*

spx	fighter	blunt	comb	post	delta
77.23.%	35.1%	27.56%	32.2%	20.52%	9.27%

Table 3 *The partition overhead for using a $4\times4\times4$ spatial partitioning for each dataset.*

```
int mesh_partition(starting_point, ending_point, part_number)
{
    if part_number = 1 then
      return number_of_tetra_in(starting_point, ending_point)

    part_number_left = part_num / 2
    part_number_right = part_num - part_num_left

    a = starting_point
    b = ending_point
    old_cut = a

    while (TRUE) {
        cut = (a + b)/2
        count_left = number_of_tetra_in(starting_point, cut)
        count_right = number_of_tetra_in(cut, ending_point)
(##)    diff = count_left - count_right
        if diff = 0, or old_cut is sufficiently close to cut then
          break
        old_cut = cut
(**)    if count_left > count_right then
          b = cut
        else then
          a = cut
    }
    count_left = mesh_partition(starting_point, cut, part_num_left)
    count_right = mesh_partition(cut, ending_point, part_num_right)
    return count_left + count_right
}
```

Fig. 4 *The pseudo code for the partitioning algorithm by a* recursive binary search *approach.*

```
int mesh_partition(starting_point, ending_point, part_number)
{
    if part_number = 1 then
      return number_of_tetra_in(starting_point, ending_point)

    part_number_left = part_num / 2
    part_number_right = part_num - part_num_left

    a = starting_point
    b = ending_point
    old_cut = a

    while (TRUE) {
        cut = (a + b)/2
        count_left = mesh_partition(starting_point, cut, part_num_left)
        count_right = mesh_partition (cut, ending_point, part_num_right)
(##)    diff = count_left - count_right
        if diff = 0, or old_cut is sufficiently close to cut then
          break
        old_cut = cut
(**)    if count_left > count_right then
          b = cut
        else then
          a = cut
    }
    return count_left + count_right
}
```

Fig. 5 *The pseudo code for the partitioning algorithm by a* simultaneous binary search *approach.*

	spx	fighter	blunt	comb	post	delta
# of layers	8	17	27	14	50	24
max patching	1.93%	3.5%	1.53%	0.18%	0.24%	0.38%
avg. tetra. cost	3.29	3.38	3.44	3.49	3.41	2.34
overhead	360.06%	263.14%	237.22%	212.84%	228.81%	276.56%

Table 2 *The number of partitions that results in the same dataset size as the input mesh, the maximal percentage of patching tetrahedra encountered, the average compression efficiency (in terms of number of bits per tetrahedron) of all layers, and the overhead of storage size of all the layers before compression, for each dataset.*

```
balance_partitions_between(starting_part_index, ending_part_index)
{
    old_diff = 0
    while (TRUE) {
        max_part_index = the index of partition whose difference
            with its next partition is maximal
        diff = the difference between partition max_part_index and
            partition max_part_index+1
        if diff = old_diff then
            break
        old_diff = diff
        balance_two_partitions(max_part_index, max_part_index + 1)
    }
}

balance_with_heuristics()
{
    old_diff = 0
    while(TRUE) {
        min_part_index = the index of partition with least tetrahedra
        max_part_index = the index of partition with most tetrahedra
        if min_part_index = max_part_index then
            break
        diff = difference in number of tetrahedra between
            partition min_part_index and partition max_part_index
        if diff = old_diff then
            break
        old_diff = diff
        if min_part_index < max_part_index then
            balance_partitions_between(min_part_index, max_part_index)
        else then
            balance_partitions_between(max_part_index, min_part_index)
    }
}
```

Fig. 6 *The pseudo code for balancing partitions by heuristics.*

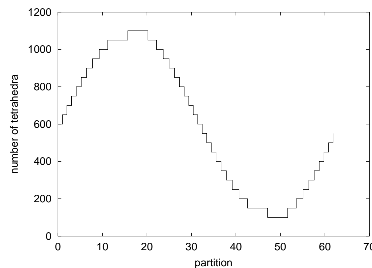


Fig. 7 *A problematic example for using our heuristics to balance partitions.*

```
balance_with_simulated_annealing()
{
    for j from 1 to 100 {
        nsucc = 0
        for k from 1 to nover {
            cut = pick a randomly selected cut
            direction = pick a random direction: left or right
            if the direction is left then
                cost = cost to move the cut leftward
                if oracle(cost, t) says yes then
                    move the cut leftward
                    nsucc = nsucc + 1
            else then
                cost = cost to move the cut rightward
                if oracle(cost, t) says yes then
                    move the cut rightward
                    nsucc = nsucc + 1
            if nsucc >= nlimit then
                break
        }
        t = t * TFACTOR
        if nsucc = 0 then
            break
    }
}
```

Fig. 8 *The pseudo code of using a* simulated annealing *approach to balance partitions.*

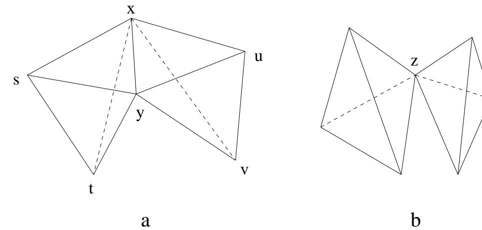


Fig. 9 *Examples of non-manifold meshes. In* **a**, *the edge marked by* x *and* y *is a singular edge. In* **b**, *the vertex marked by* z *is a singular vertex.*

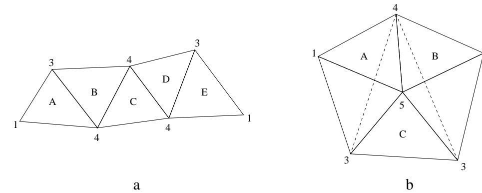


Fig. 10 *Reasons of why a dis-connect or a non-manifold sub-mesh could be formed. Vertices are marked with their associated scalar values. In* **a**, *a 2D example is given, where each triangle represents a tetrahedron. A sub-range of* $[1, 2]$ *will create a dis-connected sub-mesh that includes only tetrahedra* A *and* E. *In* **b**, *a sub-range of* $[1, 2]$ *will not include tetrahedra* C, *thus leading to a non-manifold sub-mesh.*

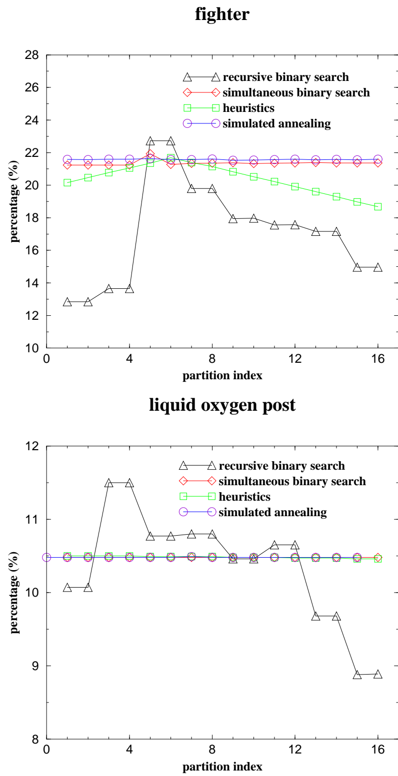


Fig. 11 *Comparison of the effectiveness by making 16 partitions from the fi ghter and the liquid oxygen post datasets using all four algorithms.*

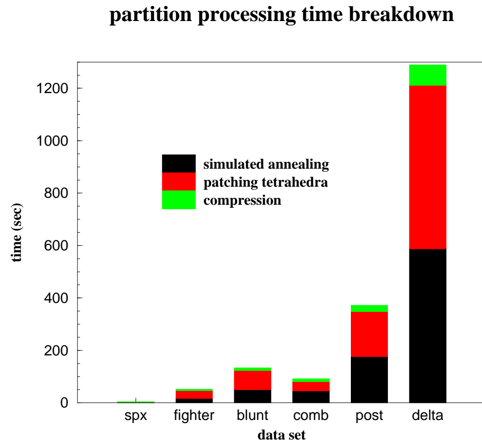


Fig. 12 *The timing breakdown on the generation of layers for each dataset.*

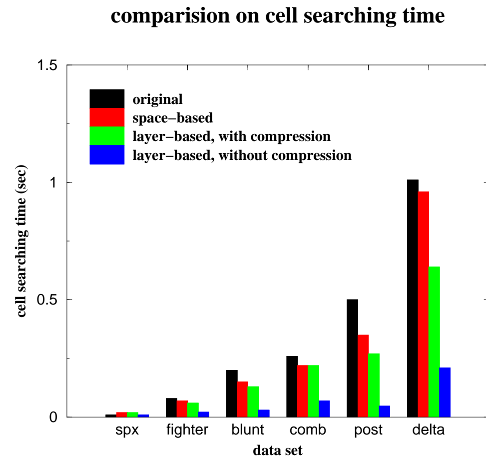


Fig. 13 *Comparison of the cell searching time among four approaches to perform iso-surface extraction for each dataset. We uniformly select 100 sampled scalar values from the scalar value range of each dataset to be used as the query values.*

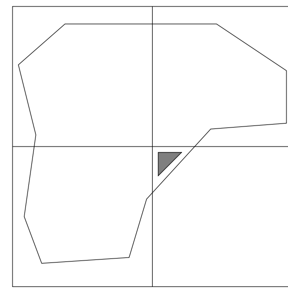


Fig. 14 *A 2D example of a space-based partitioning.*

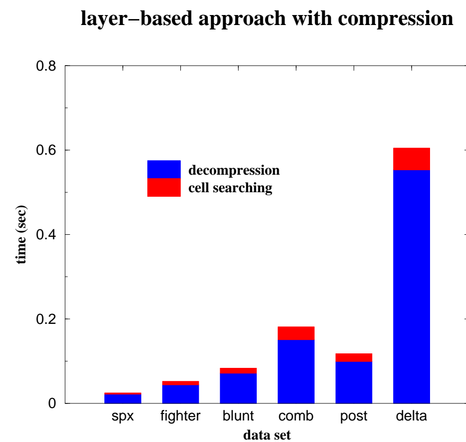


Fig. 15 *Breakdown of the cell searching time for iso-surface extraction using the layer-based approach with compression.*

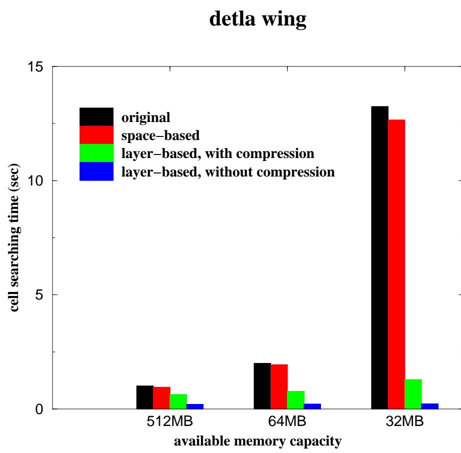
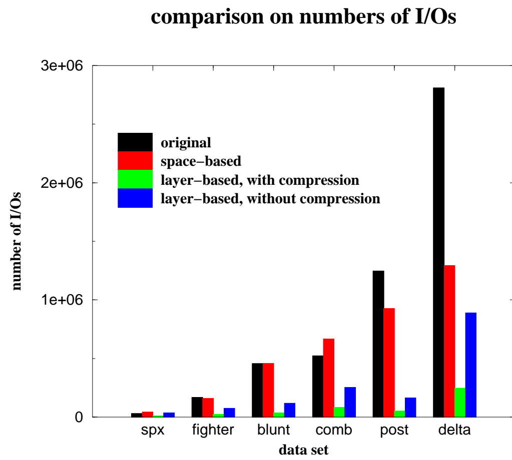


Fig. 16 *The cell searching time for iso-surface extraction on the* delta wing *dataset under different memory capacity.*



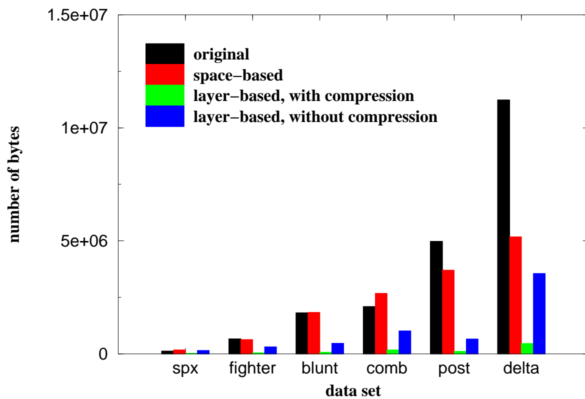


Fig. 17 *Comparisons on the number of I/Os and number of bytes read from the disk, for all the four approaches mentioned in the text.*

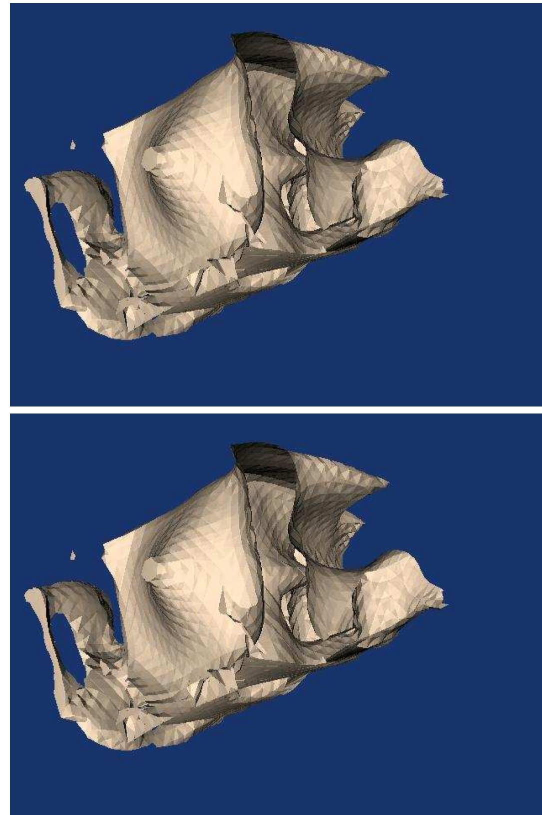


Fig. 18 *Rendered iso-surfaces using both the coordinate-based (top image) and layer-based (bottom image) decompositions.*

Chuan-Kai Yang received his Ph.D. degree in computer science from Stony Brook University, USA, in 2002, and his M.S. and B.S. degree in computer science and in mathematics from National Taiwan University in 1993 and 1991, respectively. He has been an Assistant Processor of the information management department, National Taiwan University of Science and Technology since 2002. His research interests include computer graphics, scientifi c visualization, multimedia systems, and computational geometry.

Tzi-Cker Chiueh is currently a Professor in Computer Science Department of Stony Brook University. He received his B.S. in EE from National Taiwan University, M.S. in CS from Stanford University, and Ph.D. in CS from University of California at Berkeley in 1984, 1988, and 1992, respectively. He received an NSF CAREER award in 1995. Dr. Chiueh's research interest is on computer security, network/storage QoS, and wireless networking.