Encoding Volumetric Grids For Streaming Isosurface Extraction

Ajith Mascarenhas[*] Martin Isenburg[*] Valerio Pascucci[†] Jack Snoeyink[*]

Abstract

Gridded volumetric data sets representing simulation or tomography output are commonly visualized by displaying a triangulated isosurface for a particular isovalue. When the grid is stored in a standard format, the entire volume must be loaded from disk, even though only a fraction of the grid cells may intersect the isosurface.

We propose a compressed on-disk representation for regular volume grids that allows streaming, I/O-efficient, out-of-core isosurface extraction. Unlike previous methods, we provide a guaranteed bound on the ratio between the number of cells loaded and number of cells intersecting the isosurface that applies for any isovalue. As grid cells are decompressed, we immediately extract vertices and triangles of the isosurface. Our output is a coherent streaming mesh, which facilitates subsequent processing, including on-the-fly simplification and compression.

1 Introduction

The increases in sizes of geometric data sets acquired by sampling devices and computed by scientific simulations has fostered demand for scalable processing techniques. In particular, stream processing is emerging as an I/O-efficient algorithmic framework that is sufficiently independent from the size of input and output data. Stream algorithms that seamlessly operate on large data sets using relative little memory have been developed for separate tasks, such as simplification, compression, and mesh generation [11]. But stream processing also creates the possibility of pipelining these algorithms together to work concurrently on the same dataset.

In order to stream-process geometric data it must be kept in a streamable representation. For polygon meshes, a streaming format provides concurrent access to coherently ordered vertices and triangles [11]. Such *streaming meshes* are represented as an interleaved sequence of vertices and triangles and include additional *finalization* information that specifies when vertices are no longer used. This information enables stream-processing algorithms to complete the computations involving finalized vertices and their triangles, optionally output the result, and then safely free up space for new mesh elements.

[*]Department of Computer Science, University of North Carolina at Chapel Hill, {ajith,isenburg,snoeyink}@cs.unc.edu

[†]Lawrence Livermore National Labs, pascucci@llnl.gov

In this paper, we develop a mesh representation that supports the extraction of isosurfaces from regular gridded volume data in two ways. First, for a chosen $\rho \geq 1$, it guarantees that the number of volume grid cells read from the disk is at most ρ times the number that intersect any isosurface. Second, and more importantly, it extracts directly to a streaming format, an important step for achieving end-to-end streaming pipelines in scientific visualization. This allows us, for example, to pipe the isosurface to a simplification process to get immediate visual feedback while it is being extracted. It also allows us to directly compress the isosurface as we write it to disk, or for transmission to a remote display site.

Related prior work. Isenburg and Lindstrom [11] present the underlying theory of creating and working with a streaming representation for polygonal meshes. They define coherent and compatible orderings of mesh vertices and triangles and present metrics and diagrams that characterize how streamable a particular mesh layout is. They also mention a straight-forward, but I/O-inefficient, method for creating streaming isosurface output. Their implementation for marching-cubes isosurface extraction [13] loads the volume grid layer by layer, outputs all vertices of one volume layer, followed by a set of triangles, and always finalizes the vertices from the previous layer before moving on to the next layer.

Such an approach is suitable in cases where the volume data changes often and only the isosurfaces for one or two isovalues are extracted, but not suitable for the common case, where volume grids are created by computationally intense simulations once, and different isosurfaces are extracted repeatedly to study the data. It is inefficient to scan the entire volume grid to extract an isosurface, as in general only a small percentage of grid cells will actually intersect a particular isosurface. Several external memory techniques have been proposed to avoid the I/O-overhead associated with a layer by layer scan of the volume. The common idea is to find an on-disk arrangement of the volume grid that allows selective loading of cells, and avoids loading those that clearly do not intersect the isosurface.

Chiang et al. [2] adapt the idea of treating isosurface extraction as an interval stabbing problem [3] to an out-of-core setting. They rearrange the on-disk layout of the volume grid into clusters of cells or *meta cells* that are referenced by an interval tree. This allows them to query only those meta cells that contain at least one cell intersecting the isosurface. One benefit of their approach is that all preprocessing can be done out-of-core using multiple external sorts. However, although the number of cells they need to load tends to be a within practical ratio to the number of intersecting cells, they cannot provide a tight bound on their number. Furthermore, this method cannot produce coherent streaming mesh output since the order in which the meta cells are processed does not follow the geometry of the extracted isosurface.

Static data decompositions that guarantee good external memory performance and parallel load balancing for large rectilinear grids were first introduced in [1] and successively improved with random data distribution [20] and view-dependent data selection [21]. The coarse-to-fine adaptive refinement used in [6] avoids loading fine resolution data for isosurfaces that are further from the viewpoint. Recent developments [19] deal with accurate estimate of the isosurface extraction cost to fine-tune the performance in external-memory. These, and all techniques that reorder or partition rectilinear grids, face the problem of increasing storage cost to maintain connectivity information that would otherwise be implicit.

The need for compact representations of geometry data has motivated research on geometric compression. Most works have focused on irregular triangulated surface meshes [15, 16, 8], with some generalizations to compress irregular volume meshes [7, 10]. Efficient encodings for regular volume grids has been investigated in two contexts. Some approaches compress the entire volume grid [5, 9], others compress only a single isosurface for a particular isovalue [14, 12] by encoding its occupancy grid plus information to refine the vertex placement. In either case the respective grid is compressed with a complete layer by layer traversal. In contrast, our scheme compresses a partial volume grid that encodes all isosurfaces within a range of isovalues. To support streaming, we do this with a traversal that follows the geometry of the isosurfaces and not by sweeping the grid.

Contributions. This paper provides a method for encoding and storing regularly gridded volumetric data on disk in a compressed format that supports streaming, out-of-core isosurface extraction while providing guarantees on the I/O-efficiency. For this, we partition the space of isovalues into ranges such that each range satisfies a specified upper bound on the ratio of the number of cells loaded from disk to the number of cells intersecting any isosurface within that range. The decoder loads and uncompresses only those parts of the volume grid that are relevant for the range containing the selected isosurface, which is extracted as a streaming mesh. We present results from experiments on a few sample datasets.

Our current implementation of the encoder works in-core and therefore requires substantial amounts of main memory when encoding large volume grids. While this is a limitation, in practice we find that really large datasets are usually created on high-end computing equipment with sufficient memory resources. The same computers could also be used to encode the volume data *before* distributing it for subsequent study on commodity PCs. Nevertheless, in the future we hope to also have a version of the encoder that operates out-of-core.

Outline. Section 2 reviews required background material and gives some definitions. Section 3 explains how to partition the gridded volume data into ranges of isovalues. Section 4 describes how to compress the resulting partial volume grids for compact storage on disk. Section 5 explains how to extract an isosurface from the compressed grids in a streaming fashion. Section 6 discusses results. We conclude and list future work in Section 7.

2 Preliminaries

We assume that the volumetric data is the sampling of a scalar function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$. The function value is extended to all points in the domain by creating a regular grid with scalar values at the grid points, and using a suitable interpolation function over each grid element, a unit cube, which we shall refer to as the *cell*.

For each cell c, the $\textsc{Span}(c)$ is the interval $[\textsc{minval}(c), \textsc{maxval}(c)]$, where $\textsc{minval}(c)$, and $\textsc{maxval}(c)$ return the minimum and maximum scalar value stored at the grid points of c. For cells u and v that share a face F_{uv}, $\textsc{Span}(F_{uv})$ is the interval of the scalar values stored at the grid points of the common face.

An *isosurface* (also called *level-set*) is the preimage of a constant value, $I_k = f^{-1}(k)$, for $k \in \mathbb{R}$. The set of cells $\mathcal{A}_k$ that intersect the isosurface I_k are called *active cells*; these are organized into face-connected components. Isosurfaces can be extracted by continuation [18]: starting from a *seed cell* in an isosurface component, the entire component can be extracted by breadth-first traversal through the cell adjacency graph. A *seed set* is a set of cells that contains at least one seed cell for each connected component of each possible isosurface.

We will use continuation over ranges, so we make a few extra definitions. A *range* R is an interval of $\mathbb{R}$. We define the *active cells for a range* R as $\mathcal{A}^R = \bigcup_{k \in R} \mathcal{A}_k$, which is the set of cells that are active for any isosurface with an isovalue in R. To define continuation, we consider the cells that are face-adjacent to a cell u. Given a range R and a cell u, a cell v is *R-connected* to u if there exists a path of cells, between u and v, such that, every consecutive pair of cells in the path are face-adjacent, and $R \cap \textsc{Span}(w) \neq \emptyset$ for every cell w in the path. The connected component for cell u, denoted $\mathcal{K}_u^R \subseteq \mathcal{A}^R$, is the set of cells that are R-connected to cell u.

3 Partitioning the Volume Grid

We begin with an overview of an algorithm that extracts a connected component of an isosurface, given a starting cell that intersects this component. During each iteration we process a cell that is face-adjacent to one or more previously processed cells. We extract isosurface vertices and triangles for the currently processed cell and enqueue any unprocessed face-adjacent neighbors that also intersect the isosurface. Thus, we process a stream of volume grid cells and produce a stream of isosurface vertices and triangles. We stop processing when the queue is empty.

To organize the volume data on disk, we encode the volume grid cells in the order of visitation for a particular isosurface into a file. The extraction procedure decodes the grid cells from disk visiting them in the same order as the encoder. Instead of accessing the entire volume dataset, only those cells that are relevant for extracting this particular isosurface are read. At any time only the cells in the queue and some bookkeeping information about vertices shared between cells needs to be kept in memory.

Instead of encoding the required grid cells for every possible isovalue we partition the space of isovalues into ranges, and encode for each range the cells that contain the corresponding union of isosurfaces. An obvious concern is I/O-efficiency; for any particular isovalue we would ideally process only those cells that intersect the isosurface. In practice we settle for the number of processed cells to be bound by a constant factor of the number of cells intersecting the isosurface.

We now sketch our algorithm that preprocesses a volume grid and stores it as several compressed files on disk. The input to the algorithm is a gridded volume data set representing a scalar function f, and a parameter ρ. The output is determined in two steps:

First, we partition the space of possible isovalues f into a set of non-overlapping ranges such that, for each range R, the number of active cells for the all isosurfaces in R is at most ρ times the number of active cells for any individual isosurface in R. In notation,

$$|\mathcal{A}^R| \leq \rho \min_{k \in R}\{|\mathcal{A}_k|\}.$$

Second, for each range R, we produce seed cells for each connected component of $\mathcal{A}^R$. For each seed, we propagate through the connected component and compress the values of f in propagation order to disk.

Range Partition. If, for each cell with non-empty span, we record the start and end of its interval, then range partitioning can be performed by a simple greedy process that will produce the smallest number of ranges satisfying the desired condition. Note that the number of intervals intersecting any value k is simply the number of interval starts minus the number of interval ends that are less than or equal to k. We simply scan the list of starts and ends in increasing order, keeping track of the current and total number of cells in the current range, and the minimum number intersected by any isosurface in the range. We cut off a new range whenever their ratio exceeds ρ.

Alternatively, one could use dynamic programming to determine the smallest parameter ρ to achieve a given number of ranges, or other desired properties.

Propagation Order. Given a cell c and an open range R_c, we compute the set of cells that are R_c-connected to c by performing a breadth-first search starting from c. We think of this procedure as a propagation of the range R_c and implement it as shown below:

```
PROPAGATERANGE(Cell $c$, Range $R_c$)
  ENQUEUE($Q, c$)
  Mark($c$)
  while($Q$ not empty) do
     $u$ = DEQUEUE($Q$)
     for each face-adjacent cell $v$ do
         if (($v$ not marked) AND
             ($R_c \cap$ SPAN($v$) $\neq \emptyset$) then
                 ENQUEUE($Q, v$)
                 Mark($v$)
         endif
     done
  done
```

We enqueue an unmarked face-adjacent cell only if its span intersects R_c in a non-empty interval. This excludes *flat cells*; cells with the same function value at all grid points, and non flat cells whose span intersects R_c in a point. We can ignore these cells as they do not contribute to any isosurface within the range R_c. Thus, the result of a call to PROPAGATERANGE is a set of covered cells in the range, bounded by cells that are hit, but not covered, namely the cells that poke above or below the range (or both), and the flat cells that lie within the range. We use the order that the range propagation induces on the grid cells to store the data associated with them on disk. In Section 4 we will show how this data can be compressed for efficient storage. As we will explain in the next paragraph, we use subsequent calls to PROPAGATERANGE to visit yet-uncovered portions of non-flat cells.

Seed set based covering. Since PROPAGATERANGE can produce the set of cells that are covered by a given seed in a given range, we greedily apply it until the entire data set is covered. We initially choose an arbitrary non-flat seed cell c, select a partition range $R = [a, b]$ that intersects SPAN(c), and call PROPAGATERANGE(c, R) to obtain the connected component $\mathcal{K}_c^R$, which is bounded above and below by portions of isosurfaces I_a and I_b.

For subsequent calls to PROPAGATERANGE, we propagate one seed cell and range per connected component of $\mathcal{A}_a \cap \mathcal{K}_c^R$ and $\mathcal{A}_b \cap \mathcal{K}_c^R$, which are the active cells of I_a and I_b. We maintain a collection of connected components of $\mathcal{A}_a \cap \mathcal{K}_c^R$ in a union-find (UF) data structure [4], and with each UF-root we maintain the seed cell for that component. (We process cells in $\mathcal{A}_b \cap \mathcal{K}_c^R$ in a similar fashion.) We give our initial seed cell c a new UF-component equipped with a pointer to c. We perform the following when propagating from cell u to its adjacent node v. If v is unmarked and has a residual lower range $[a', a]$ with $a' \leq a$, then $v \in \mathcal{A}_a \cap \mathcal{K}_c^R$ and we create a new UF-component equipped with a pointer to v. If $u \in \mathcal{A}_a \cap \mathcal{K}_c^R$, (i.e. it has a non-null UF-component), we union v's UF-component with u's. If $v \in \mathcal{A}_a \cap \mathcal{K}_c^R$ and is marked, we do not create a new UF-component, but perform the union of the UF-components of u and v as in the unmarked case. If v is a flat cell, it could be part of a *sea*; a connected component of flat-cells. We select a seed for such a sea as it could contain *islands*; connected components of non-flat cells. We create a new sea-component for v, and merge sea-components associated with adjacent non-flat cells.

When PROPAGATERANGE terminates, the root of each UF-component contains a pointer to a seed cell for a component of I_a, I_b, or a sea. For each component of I_a, we select from the partition the range with upper end-point equal to a, and for each component of I_b we select the range with lower end-point equal to b. We invoke PROPAGATERANGE on each of these seed cell/range pairs. We do not encode flat cells, but discover seeds for islands by propagation from sea seed cells. Since the domain is simply connected, isosurfaces nest, so to avoid generating a new seed for an already visited component, it is sufficient to remember a single component that started a propagation. We stop propagation when we exhaust all seeds.

4 Encoding Partial Volume Grids

For efficient storage to disk, we compress for each range of isovalues only those parts of the volume grid that are relevant for that particular range. These *partial volume grids* are composed of all grid cells that are active for the respective range. There are two types of information to encode: which are the active grid cells and what are the scalar values associated with their grid points. The information about which grid cells are active can – to a certain extent – be derived from propagating the range. The scalar values associated with the grid points of the seed cells and the grid points of the cells visited during range propagation are compressed using predictive coding.

Our approach is similar to the compression technique for hexahedral volume meshes by Isenburg and Alliez [10]. Each iteration of their encoding algorithm compresses a hexahedron that is face-adjacent to one or more previously compressed hexahedra. This involves specifying local connectivity around the hexahedron and the 3D position (plus optional scalar values) associated with those vertices of the hexahedron that are encountered for the first time. This vertex data is predicted from the data of previously compressed, neighboring vertices. This process maps naturally to our traversal of volume grid cells during the PROPAGATERANGE procedure.

Given a seed cell and a range, our procedure visits face-adjacent cells in a deterministic order that can often be derived from the scalar values of previously visited cells. Only occasionally we need explicit flag bits that tell the decoder if a cell is to be visited or not because it could not be decided from the previously decoded data alone.

A processed cell is in one of nine configurations face-adjacent to previously visited cells. These configurations are detailed in [10] and we provide them for convenience in Figure 1. The grid points whose scalar values are not yet encoded are called *free points*, and exist only in the *hut*, *step*, and *corner* configurations. We compress their scalar values when such a configuration occurs. Because a cell can in addition be point- or edge-adjacent to previously processed cells, some of its free points may already be encoded. To avoid re-encoding the values of such grid points we check for point- and edge-adjacent cells in the propagation queue.

The decoder reads the compressed data and performs an exact replay of the range propagating procedure. At this time any isosurface within the respective range can be extracted.

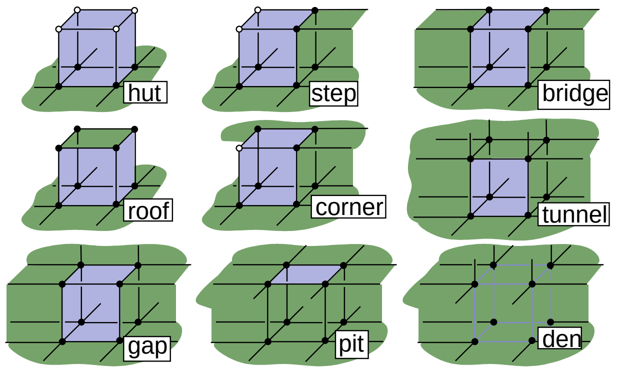
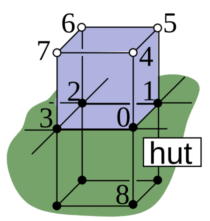


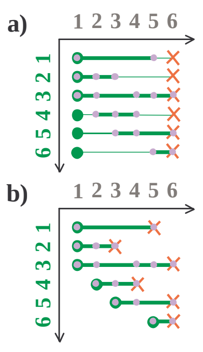
Figure 1: Nine different configurations in which a cell (blue) can be face-adjacent to processed cells (green). Free grid points are drawn as unfilled circles.

We use a prediction scheme similar to [10] to compress the scalar values at the free points. We predict the value at a grid point based on values from previously encoded grid points, compute the difference between the predicted and the actual value, and then store only this corrective value. Since these corrections tend to spread around zero they can be efficiently compressed with an arithmetic coder [17]. Whenever possible we use a single *parallelogram prediction* [16] within a face of the cell as as shown in Figure 2. Seed cells are special cases, since none of their scalar values have previously been encoded. The value at grid point p_0 is predicted as 0, the values at p_1 and p_2 cannot be parallelogram predicted and are instead predicted as a previously encoded value, say that at p_0. All subsequent values use the parallelogram rule, except for p_4 of the "hut" configuration which extends a ray from p_8 through p_0 if p_8 exists, or uses p_0 otherwise.



point	prediction rule
p_0	0
p_1	p_0
p_2	p_1
p_3	$p_0 - p_1 + p_2$
p_4	$2p_0 - p_8$ or p_0
p_5	$p_1 - p_0 + p_4$
p_6	$p_2 - p_1 + p_5$
p_7	$p_3 - p_2 + p_6$

Figure 2: Prediction rules for scalar values at grid points.



```
# standard .obj
#
v -0.1  1.1  0.2
v  0.4  0.2  0.5
v  0.6  0.9  0.4
v  0.7  0.6  0.6
v  0.9 -0.1  0.5
v  1.1  1.0  0.4
f  1  2  3
f  3  2  4
f  4  2  5
f  3  4  5
f  1  3  6
f  6  3  5
```

```
# streaming .sma
#
v -0.1  1.1  0.2
v  0.4  0.2  0.5
v  0.6  0.9  0.4
f  1  2  3
v  0.7  0.6  0.6
f  3  2  4
v  0.9 -0.1  0.5
f  4  2  5
x 2
f  3  4  5
x 4
v  1.1  1.0  0.4
f  1  3  6
x 1
f  6  3  5
x 6
x 3
x 5
```

Figure 3: (a) standard OBJ format: all vertices are introduced in the beginning and remain active until the end. (b) streaming mesh format: vertices are introduced when needed and finalized when no longer used—illustrated here with the 'x' command.

In PROPAGATERANGE an unmarked cell v that is face-adjacent to the current cell u is visited only if $R_c \cap \text{SPAN}(v) \neq \emptyset$, where R_c is the propagation range. To evaluate this condition the decoder would already need all scalar values of v, which may not yet all be decoded. Therefore the decoder tests first if $R_c \cap \text{SPAN}(F_{uv}) \neq \emptyset$ using the face F_{uv} of the already decoded cell u and, if true, adds v to the queue. If $R_c \cap \text{SPAN}(F_{uv}) = \emptyset$, the encoder outputs a flag bit, 1, if $R_c \cap \text{SPAN}(v) \neq \emptyset$, or 0 otherwise. Then the decoder will read this flag bit to decide if v is to be added to the queue or not.

We use four arithmetic contexts to encode the corrective values and one to encode the flag bits. We switch between the four contexts based on the prediction rule used: no prediction, delta, ray shooting, or parallelogram.

5 Streaming Isosurfaces

Our decoder simultaneously extracts the vertices and the triangles of the isosurface. While it does not know in advance how many vertices and triangles the final mesh will have, it does know when vertices will no longer be used for subsequent triangles. It comes therefore quite natural to output the extracted mesh using a streaming format [11]. A streaming mesh format can be as simple as the ASCII example in Figure 3. It interleaves vertices and the triangles that use them, and in addition specifies when vertices are referenced for the last time.

The fact that vertices and triangles can be written in an interleaved fashion enables our extraction process to immediately output the mesh data as it is produced. This can be done in a single pass—even if the the final vertex and triangle counts are not known a priori. Compare this to the possible alternative of using a standard indexed format where a block of vertices is followed by a block of triangles. A mesh generating application could then either accumulate all vertices and triangles in memory. That would prohibit the generation of meshes that are larger than the available main memory. It could also memory map a binary file and fill in the vertex and triangle arrays. But that would require to at least know the final number of mesh vertices in advance. Or it could write the produced vertices and triangles into two temporary files that are concatenated afterwards. But that would mean that the mesh is no longer produced in a single pass.

When producing large meshes, there is a big payoff in arranging vertices and triangles in a coherent manner. Isenburg and Lindstrom [11] describe metrics that measure, and diagrams that visualize the coherency in a *mesh layout* (e.g. in the ordering of vertices and triangles) and advocate to produce streaming meshes that are low in both *width* and *span*. The width of a streaming mesh corresponds to the maximal number of vertices that are *active* at the same time, whereas the span corresponds to the longest duration that any vertex remains active.

From our point of view, a vertex is active from the moment we output it until the moment we specify it as finalized. To avoid bloating the width, we need to output vertices just before the first triangle that references them and finalize them immediately after the last triangle that references them. The isosurface vertices lie on the edges of the volume grid and are incident to the cycle of triangles that is extracted from the cycle of volume grid cells sharing this edge. Hence, we output an isosurface vertex just before the first triangle of the first cell of such a cycle is written and we finalize it when the last triangle from the last cell of this cycle is written. For this, we maintain reference counters for each edge that intersects the isosurface. We initialize the count to the number of cells adjacent to the edge, decrement it as we process these cells, and finalize the corresponding vertex when the count reaches zero. This produces *compact* streaming meshes that have

dataset name	total points (T)	# encoded points (E)	compressed size (C)	bits per point E_{bpp}	T_{bpp}	encoding time
ENGINE	7,208,960	10,374,052	4.8 MB	3.88	5.58	(03:54)
JETDATA	16,777,216	4,086,503	2.6 MB	5.33	1.30	(02:25)
PPM256	16,777,216	12,731,615	7.7 MB	5.07	3.85	(04:58)
PPM512	134,217,728	50,340,331	30.5 MB	5.08	1.90	(25:55)

Table 1: Encoder performance for $\rho = 3$. The compressed size is the total of all files on disk. The bits per point for the encoded points is $E_{bpp} = 8C/E$, and the bits per point for the total points in the data set is $T_{bpp} = 8C/T$. The time is given in (mm:ss).

range	compressed size	# encoded cells	# encoded points
$[0, 3)$	0.00 MB	0	0
$[3, 4)$	0.02 MB	64,965	128,968
$[4, 52)$	1.90 MB	2,212,832	3,100,227
$[52, 122)$	2.54 MB	3,026,904	3,936,116
$[122, 193)$	1.70 MB	1,818,150	2,572,010
$[193, 228)$	1.00 MB	1,124,313	1,634,383
$[228, 229)$	0.46 MB	644,101	1,209,279
$[229, 230)$	0.04 MB	86,321	150,632
$[230, 255)$	0.00 MB	0	0

Table 2: The compressed size of the individual files for each of the nine range partitions of the PPM256 dataset at $\rho = 3$ and the number of cells and points in the corresponding partial grid.

ρ	# encoded points (E)	compressed size (C)	bits per point E_{bpp}	T_{bpp}
2	17,854,894	11.0 MB	5.16	5.50
3	12,731,615	7.70 MB	5.07	3.85
4	10,460,851	6.50 MB	5.20	3.25
5	9,854,635	6.02 MB	5.12	3.00
6	8,318,042	5.08 MB	5.12	2.54
7	8,020,754	4.84 MB	5.06	2.42

Table 3: Compression performance on PPM256 for varying ρ.

the lowest possible width for a given triangle ordering.

The triangle order of our output mesh is determined by the traversal of the volumetric cells, which is breadth-first. It was shown that a breadth-first traversal results in sufficiently low width for most practical applications [11], and furthermore assures that the "lifetime" of each vertex is proportional to the width, which translates into low span. It should be noted that although we traverse the volume grid component by component this does not imply that the isosurface is also traversed component-wise. Multiple isosurface components can be contained within a volume grid component and may be encountered in parallel. However, the number of isosurface components that are maximally traversed at the same time is limited to those contained within one component of the volume grid.

The coherence in the streaming mesh output we produce is illustrated in Figure 4 in the form of layout diagrams. These diagram display the coherency in reference between vertices, which are indexed along the vertical axis, and triangles, which are indexed along the horizontal axis – both are numbered in the order they are output. Triangles that share the same vertex are connected with horizontal line segments and vertices that are referenced by the same triangle are connected with vertical line segments. The closer these line segments group around the diagonal the more coherent is the layout.

The only other published alternative way of creating streaming isosurface output employs a marching-cubes implementation that streams the mesh layer by layer [11]. This approach outputs all vertices of one volume layer, followed by a set of triangles, and always finalizes the vertices from the previous layer before moving on to the next layer. Hence, both width and span of their streaming meshes are bound by the maximal number of vertices and triangles per layer. Note that this approach will traverse all isosurface components in parallel that simultaneously have an intersection with the sweeping plane.

6 Results

In this section we describe the performance of our encoder and decoder on a two processor 2.80GHz Intel Xeon PC with 4GB of RAM.

Datasets All datasets are regular gridded volumes with scalar values quantized to 8 bits.

ENGINE Dimensions: $256 \times 256 \times 110$. This dataset is a CT scan of two cylinders of an engine block.

JETDATA Data from the simulation of a supersonic fluid jet. Dimensions: $256 \times 256 \times 256$.

PPM256, PPM512 Data from the simulation study of the Richtmeyer-Meshkov instability, which occurs, for example, when a shock passes through an interface of two fluids of differing density. The dimensions of the original dataset are $2048 \times 2048 \times 1920$, we use a moderate $256 \times 256 \times 256$ sized and larger $512 \times 512 \times 512$ sized chunk of this dataset.

Encoder The performance of the encoder on our datasets with $\rho = 3$ is shown in Table 1. We compute two quan-

dataset name	decode time	# active cells	# cells decoded	cells/sec	decode + extract	# triangles extracted	tri/sec	max. queue
ENGINE	2.03 s	329,154	875,917	431K	3.76 s	659,416	175K	7,334
JETDATA	1.72 s	269,735	794,897	462K	3.12 s	541,892	173K	3,372
PPM256	4.06 s	980,009	1,818,150	447K	9.17 s	1,969,892	215K	5,298
PPM512	19.70 s	4,055,366	7,395,253	375K	42.70 s	8,149,282	190K	20,713

Table 4: Decoder performance measurements for data-sets encoded with $\rho = 3$. We set the isovalue to 127. We list the times for decoding the volume component containing the isosurface, for decoding and extracting the isosurface, the isosurface triangle count, and the maximum number of propagation queue elements used by the decoder. Each queue element uses 76 bytes.

tities to describe the achieved compression. First, bits per point for the grid points that were encoded E_{bpp}, and second, bits per point for the total number of grid points in the dataset T_{bpp}. Because we do not encode flat cells, and because most common dataset have a large number of flat cells, we expect the second quantity T_{bpp} to be smaller than the first quantity E_{bpp}. Notice, however, that for the ENGINE dataset the number of encoded points is larger than the total number of points as the duplication of points from cells that span multiple propagation ranges outweighs the gain from avoiding flat cells. The last column summarizes the encoding time, with the current implementation. Since encoding is done once for the entire dataset, as a preprocess for isosurface extraction later, we have not spent much effort on optimizing the code to make the encoding process run fastest and with the smallest memory footprint possible. This currently prevents us from encoding larger datasets, like the entire PPM dataset, and is our next target for improvement.

Table 2 shows the range partitioning for PPM256 with $\rho = 3$ along with the compressed file size. It also reports the number of encoded cells and points that are associated with the partial grids that cover each particular range. This is the amount of data that is read from disk and these are the number of cells and points that are decoded when an isosurface falling within that range is extracted.

Table 3 shows the compression performance on the PPM256 dataset for varying ρ. With increasing ρ, the number of ranges in the partition decrease, reducing duplicate encoding of points. We notice the number of encoded points, the compressed size, and T_{bpp} decrease, whereas E_{bpp} remains about the same for all values of ρ. This is, because the compressed data size varies more or less in proportion with the number of points encoded.

Decoder and streaming isosurfaces. In Table 4, we list running times and various counts for decoding and extracting isosurfaces from volume grids that were compressed with $\rho = 3$. We also report separate timings for only decoding the partial volume grids without extracting an isosurface. Figure 4 illustrates the stream characteristics of the extracted meshes using layout diagrams [11]. The diagrams show a thin diagonal line, which indicates good stream quality. The column for maximal queue size measures the maximum number of decoded cells that are in memory at the same time during the decode and extraction process. We observe that this quantity remains modest even for the relatively large 8 million triangle isosurface that is extracted from the PPM512 dataset. In fact, the memory footprint of the decoder remains below 7 MB for all datasets.

isovalue	decode + extract	# triangles extracted	tri/sec	max. queue
10	9.4 s	1,645,388	175K	10,064
50	11.1 s	2,272,748	204K	10,064
100	13.2 s	2,305,468	174K	10,246
220	5.2 s	1,022,658	196K	3,799

Table 5: Decoder performance for extracting isosurfaces corresponding to four different isovalues from PPM256 with $\rho = 3$.

dataset	Isenburg & Lindstrom [11]			our
name	x	y	z	method
ENGINE	8,955	8,262	19,903	5,302
JETDATA	3,222	6,738	6,342	2,578
PPM256	17,713	17,556	15,115	5,777
PPM512	37,154	37,482	75,504	22,744

Table 6: Stream-width (maximum number of active vertices) for isosurfaces at value=127. For Isenburg & Lindstrom's method we list the stream-width along each sweep direction.

Table 5 summarizes the performance of the decoder for extracting isosurfaces corresponding to four different isovalues from the PPM256 dataset. We get similar decode plus extraction times for similar sized isosurfaces. The maximal queue size for the first two isovalues is the same because they fall into the same range partition. This means that the same file (or the same partial grid) is decompressed during extraction of these isosurfaces.

Table 6 compares the stream-widths of isosurfaces obtained through our method with those obtained through the method of Isenburg and Lindstrom [11] for an isovalue of 127. Because Isenburg and Lindstrom process the volume layer by layer, the stream-width of their isosurfaces depends on the sweep direction.

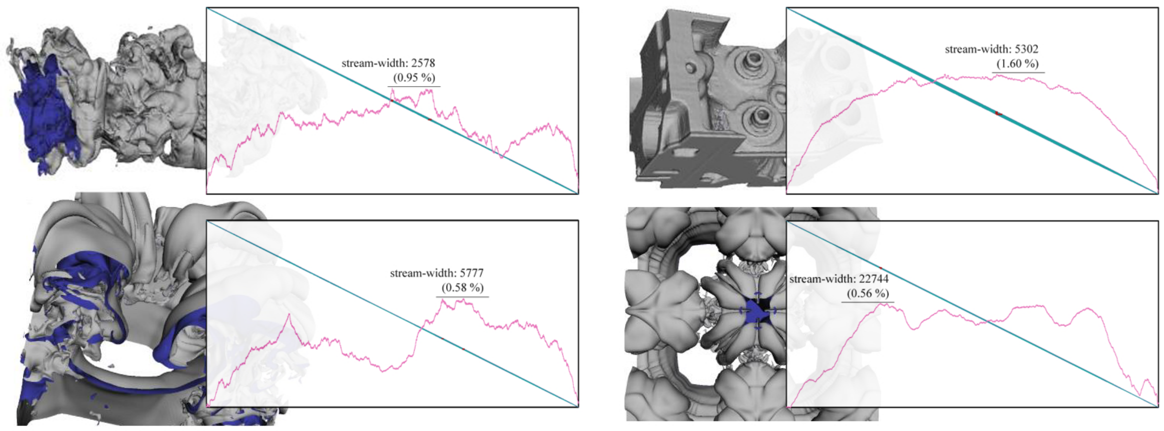


Figure 4: Isosurfaces at value=127 for JETDATA, ENGINE, PPM256, and PPM512 with their layout diagrams.

7 Conclusion and Future Work

We have presented an in-core encoder for compressing gridded volumetric data on disk, and an out-of-core decoder that performs I/O-efficient isosurface extraction into a streaming format. We ensure I/O-efficiency by specifying an upper bound, which applies to all isosurfaces, on the ratio of the number of cells loaded from disk to the number of cells intersecting the isosurface.

In the future, we would like to experiment with huge datasets like the entire PPM volume. Encoding such data will either require large memory resources or special out-of-core techniques. But whichever way the preprocessing is done, once encoded, our compressed volume grid representation allows efficient isosurface extraction on commodity PCs. While we have restricted our attention to regular gridded volumes, our techniques can also be extended to irregular volume meshes, although this will require explicit coding of the connectivity.

Acknowledgments JETDATA is from the Advanced Visualization Data Center's data repository courtesy of Andrea Malagoli and Milena Micono, LASR, University of Chicago. ENGINE is courtesy of General Electric Research and Development. PPM is courtesy of Lawrence Livermore National Laboratories. Funding was provided by NSF grant 0128426. Portions of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

[1] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proc. of Parallel Visualization and Graphics Symp.*, pages 97–104, 1999.

[2] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings of Visualization'98*, pages 167–174, 1998.

[3] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Trans. on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1994.

[5] J. Fowler and R. Yagel. Lossless compression of volume data. In *Proc. of the Symp. on Volume Visualization*, pages 43–53, 1994.

[6] B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci, and K. I. Joy. Interactive view-dependent rendering of large isosurfaces. In *Proceedings of Visualization'02*, pages 475–482, 2002.

[7] S. Gumhold, S. Guthe, and W. Straßer. Tetrahedral mesh compression with the cut-border machine. In *Proceedings of Visualization'99*, pages 51–58, 1999.

[8] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *SIGGRAPH'98 Conference Proceedings*, pages 133–140, 1998.

[9] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. In *Proceedings of Eurographics'03*, pages 343–348, 2003.

[10] M. Isenburg and P. Alliez. Compressing hexahedral volume meshes. In *Proc. of Pacific Graphics 2002*, pages 284–293, 2002.

[11] M. Isenburg and P. Lindstrom. Streaming meshes. *manuscript*, April 2004.

[12] H. Lee, M. Desbrun, and P. Schröder. Progressive encoding of complex isosurfaces. *ACM Trans. Graph.*, 22(3):471–476, 2003.

[13] W. E. Loresen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH'87 Conference Proceeding*, pages 163–169, 1987.

[14] G. Taubin. BLIC: bi-level isosurface compression. In *Proceedings of Visualization'02*, pages 451–458, 2002.

[15] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Trans. on Graphics*, 17(2):84–115, 1998.

[16] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface '98 Conference Proceedings*, pages 26–34, 1998.

[17] I. H. Witten, R. Neal, and J. G. Cleary. Arithmetic coding for data compression. In *Comm. of the ACM 30(6)*, pages 520–540, 1987.

[18] G. Wyvill, C. McPheeters, and B. Wyvill. Data Structure for Soft Objects. *The Visual Computer*, 2:227–234, 1986.

[19] H. Zhang and T. S. Newman. Efficient parallel out-of-core isosurface extraction. In *Proceedings of Parallel and Large-Data Visualization and Graphics Symposium'03*, pages 9–16, 2003.

[20] X. Zhang, C. Bajaj, and W. Blanke. Scalable isosurface visualization of massive datasets on COTS clusters. In *Proc. of Parallel and Large-Data Vis. and Graphics Symposium'01*, pages 51–58, 2001.

[21] X. Zhang, C. Bajaj, and V. Ramachandran. Parallel and out-of-core view-dependent isocontour visualization using random data distribution. In *Proceedings of Data Visualisation'02*, pages 9–18, 2002.