Efficient Parallel Extraction of Crack-Free Isosurfaces from Adaptive Mesh Refinement (AMR) Data

Gunther H. Weber*
Lawrence Berkeley National Laboratory
University of California, Davis

Hank Childs†
Lawrence Berkeley National Laboratory
University of California, Davis

Jeremy S. Meredith‡
Oak Ridge National Laboratory

ABSTRACT

We present a novel extraction scheme for crack-free isosurfaces from adaptive mesh refinement (AMR) data that builds on prior work utilizing dual grids and filling resulting gaps with stitch cells. We use a case-table-based approach to simplify the implementation of stitch cell generation. The most significant benefit of our new approach is that it uses ghost data to handle parallel isosurface extraction efficiently. We further present the results of applying this method to large scale data sets and analyze its computation time on parallel high-performance computing (HPC) platforms.

Index Terms: D.1.3 [Software]: Programming Techniques—Concurrent Programming; I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1 INTRODUCTION

Many physical phenomena, such as star formation, span large spatiotemporal scales. They comprise both vast empty areas and regions characterized by rapid changes in behavior. To represent the domain in an efficient way, simulation techniques must vary the resolution to adapt to local features. The block-structured adaptive mesh refinement (AMR) approach [2] addresses this challenge by creating a hierarchy of axis-aligned rectilinear grids. This representation requires less storage overhead than unstructured grids—it is only necessary to store the layout of all grids with respect to each other since connectivity within each rectilinear grid is implicit—and makes it possible to represent different parts of the domain at varying resolutions. Due to its effectiveness, an increasing number of application domains makes use of this simulation technique.

The hierarchical representation of AMR data makes data analysis particularly challenging. It is necessary to take into account that a finer grid may invalidate and replace the value of a given grid cell. A greater challenge is in handling transitions between hierarchy levels such that no discontinuities at boundaries appear between refinement levels. Like in many hierarchical data representations, cracks in an extracted isosurface arise due to T-junctions between levels. These cracks distract from a visualization's exploration- or communication-oriented objective and introduce questions of correctness. Furthermore, they affect the accuracy of quantities—such as surface area—derived from an isosurface.

T-junctions originate from re-sampling the commonly cell-centered AMR grids to a vertex centered representation for use with the standard marching cubes (MC) isosurface extraction algorithm. It is possible to avoid these T-junctions by using dual grids—the grids defined by the cell centers—instead of re-sampling [15]. However, previous work defined stitch cell generation procedurally, using a large number of special cases. More importantly, it generated stitch cells serially and required the entire data set to reside in the memory of a single processor.

Although the basic premise of AMR simulations is to perform simulations using less computation, commonly AMR simulations are massively parallel and generate very large data sets. These very large data sets mandate advanced processing techniques when performing visualization and analysis, which typically comes through parallelism. Thus, there is a need for efficient algorithms for parallel crack-free isosurface extraction.

Our goal was to design an algorithm that would work in a distributed-memory setting. Previous approaches were limited in the data sets they could process by the system memory of a single node. For example, a performance comparison to previous approaches would be impossible for the larger data set we considered, as it would not fit in memory, and previous algorithms would thus fail. We view the ability to run analysis on the system that generates the data as a key requirement. Although some sufficiently high-memory systems do exist to handle large data sets, the transfer time from the distributed-memory supercomputer to this remote shared-memory system is generally prohibitive.

Our new approach extends from prior work using dual grids and stitch cells to define continuous interpolation and isosurface extraction simplifying its implementation by using a case table. To facilitate parallelization, we utilize *ghost cells*, a concept originating from simulation. By extending grids with a one-cell wide layer of cells and filling these cells with values from adjacent grids or the next coarser level, all data required to construct stitch cells are available locally, and it becomes possible to process AMR data on a per-grid basis.

While we designed our algorithms for the same type of computational resources as those used by the simulation, we perform analysis and visualization as a post-processing step. To make this post-processing step applicable to results of the wide range of block-structured AMR simulation codes in use today, our method does not rely on data structures used by the simulation, but constructs all necessary information in a self-contained implementation. We also describe a parallel algorithm that initializes ghost cells with data appropriate for our scheme and measure performance of parallel isosurface extraction from AMR data.

In summary, our contributions are:

- A simplified stitch cell generation approach using case tables.
- Utilizing ghost cells to parallelize stitch cell generation.
- A parallel algorithm for initializing ghost cells appropriately.
- Performance measurements on current HPC platforms.

2 BACKGROUND AND RELATED WORK

To provide the background for our method, Section 2.1 describes the data format produced by block-structured AMR simulations, which serves as our input. Section 2.2 reviews existing work on isosurface extraction from hierarchical data representations. We particularly emphasize prior work on isosurface extraction from AMR data. Our parallelization approach uses ghost data to enable stitch cell generation on a per-grid basis in a data parallel implementation. Section 2.3 reviews both the data parallel approach and the ghost data concept, which originated with simulation codes and is used in visualization as well.

*Email: GHWeber@lbl.gov
†Email: HRChilds@lbl.gov
‡Email: JSMeredith@ornl.gov

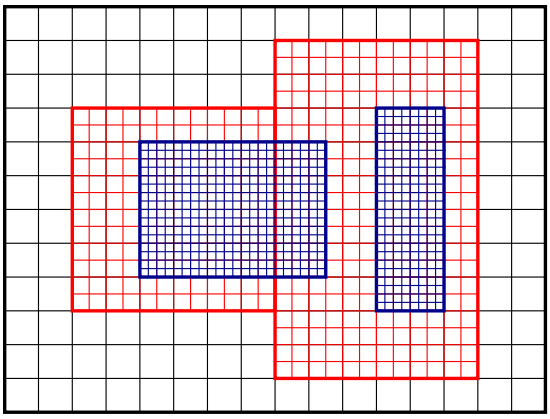


Figure 1. Two-dimensional (2D) AMR hierarchy consisting of five boxes in three levels. The coarsest level (colored black) has just one box. The middle level (colored red) has two boxes which abut. The finest level (colored blue) has two disjoint boxes.

2.1 Block-structured Adaptive Mesh Refinement

Block-structured AMR [2] uses a hierarchy of axis-aligned rectilinear grids called *boxes* (BoxLib[4] and Chombo[1]), *patches* (SAMRAI [17]), or *subgrids* (Enzo [3]) as building blocks to represent the domain. These grids are ordered in a hierarchy of levels having increasing resolution, where data in finer levels replaces those in coarser levels. The result is a representation that supports adapting to resolution changes, while requiring little storage overhead for grid structure.

Figure 1 shows an example of a two-dimensional (2D) AMR hierarchy. The *root level* (drawn black in the figure) is the coarsest level and covers the entire simulation domain with one or more axis-aligned, rectilinear grids. All grids in a given level have the same resolution, i.e., the same cell spacing.

If a level does not represent the domain at a sufficient resolution, it can serve as the parent of a refining child level. This child level covers a subregion of the parent level. A single box of the child level is not necessarily contained in any single box of the parent level—such as the left second level box in the figure—but the parent level always contains the child level completely. Levels other than the root level are not necessarily connected. For example, in the figure the two boxes comprising the second level are not adjacent.

The extents of grids in a refining level always coincide with grid cell boundaries in its parent level. As a consequence, any cell in the coarse level is either refined completely or not refined at all. Cells covered by finer cells become invalid. Most AMR codes still assign meaningful values to them by interpolation, but one can think of fine cells overriding information in coarse cells. Grid spacing in the child level is always an integer fraction of the spacing in the parent level. The *refinement ratio* for a level specifies how many child level cells a parent level cell contains along each axis. For example, all levels in Figure 1 have a refinement ratio of two.

Locations and extents of individual boxes are given as integer indices of cells of a single rectilinear grid that covers the bounding box of the domain and which has the same cell spacing as all grids of the level. In Figure 1, the two boxes of the first level (red) have the extents of $(2,3)$–$(7,8)$ and $(8,1)$–$(13,10)$. Each level uses a single grid with the same grid spacing as the level covering the entire domain—not just the bounding box of the level—as reference frame. To translate global coordinates from a parent to a child level, one multiplies them by the refinement ratio. The boxes in the second level have extents of $(8,8)$–$(18,15)$ and $(22,6)$–$(25,17)$.

2.2 Isosurface Extraction

Isosurfaces are a common building block for the visualization and data analysis of three-dimensional (3D) scalar fields. Most visualization tools use the MC [10] method to extract isosurfaces from scalar data. A correct implementation of this method [13] produces watertight, closed isosurfaces when applied to single grids, but it often exhibits cracks in the resulting isosurface when applied to hierarchical data representations. These cracks are due to T-junctions in the data representation.

This problem was first observed and fixed for octree-based multiresolution data representations [14, 16]. Most AMR simulations produce cell-centered data, and only re-sampling to a vertex-centered representation leads to T-junctions. An alternate approach changes the grids to the dual grids formed by the cell centers of AMR boxes, thus making it possible to utilize original values from the simulation and to avoid T-junctions. By using a procedural scheme, it is possible to fill gaps arising from the use of dual grids [15]. This stitch cell generation scheme imposes an additional condition on AMR data sets. There must be at least one layer of grid cells between the child level and the boundary of the parent level. BoxLib and Chombo simulations satisfy this requirement, which ensures there are no transitions between arbitrary levels. However, this approach still has limitations restricting its wide spread application: (i) it defines stitch cell generation procedurally, requiring a large number of special cases; (ii) algorithms operating on dual grids and stitch cells need to work on a per-cell basis—processing a cell immediately after generation—to avoid memory bottlenecks; and most importantly, (iii) it generates stitch cells serially and requires the entire data set to reside in the memory of a single processor. Recent work [11] generalized stitch cell generation by removing the restriction requiring a distance of a least one cell between the child level and parent level boundary, making crack-free isosurfaces available to a wider range of AMR simulations, such as Enzo [3] astrophysics simulations. However, this approach also generates stitch cells serially and needs access to an entire data set on a single processor.

Fang et al. [7] developed a crack-free isosurface extraction approach that preserves the original grids, which is useful for debugging purposes. To avoid cracks, they create transition regions comprised of pyramids between grids of different resolutions. The construction of these transition regions is also difficult to parallelize.

2.3 Parallelism and Ghost Cells

Parallelism—specifically data parallelism—is the most common approach for visualizing large data sets, and it is employed by popular tools such as EnSight [6], ParaView [9], and VisIt [5]. The data set is divided into pieces and the pieces are distributed over the processing elements. The processing elements run identical programs and only differ in which pieces they operate on. Many visualization algorithms thrive in a data parallel setting because they are embarrassingly parallel; they are able to process each cell independently of the others. Other algorithms, however, require data from the surrounding cells. For these algorithms, artifacts may occur around the boundaries of each piece, since some of the surrounding data will be located on other processing elements.

The typical approach for dealing with these artifacts is "ghost cells": one or more extra layers of cells placed around the boundary. These ghost cells complement the regular cells; they ensure that the data from surrounding cells are always available. Ghost cells are processed like regular cells, but the results that come from ghost cells are discarded before rendering. This approach works well because the ghost cells duplicate regular cells and the results from just the regular cells are sufficient.

There are multiple sources for ghost cells. Simulation codes regularly utilize ghost cells for their own calculations. In some cases, the simulation writes out its ghost cells alongside the regular ones,

meaning that ghost cells are readily available with no additional processing. When the simulation removes ghost cells from its output, though, it is up to the visualization tool to generate this data. Sometimes the simulation code describes how its pieces abut, easing the process of identifying which cells should be duplicated as ghost. The module described in this paper follows this approach; it uses abutment information from the simulation code to re-create ghost data. Further, this sort of information is regularly available for the AMR data sets considered in this paper. This is not the only approach, however. The D3 module [12] redistributes the entire data set to optimize rendering and creates ghost data in the process. This approach works for all data sets, but incurs significant costs as so much data must be moved. Finally, the approach described in this paper and in D3 require collective communication and can not work in an out-of-core setting. Isenburg et al. [8] devised a scheme for special configurations—specifically block-decomposed rectilinear grids—that can perform in an out-of-core setting. Of course, the algorithm described in this paper only specifies what type of ghost cells are required and is agnostic to the method of ghost generation. If the technique described by Isenburg et al. [8] was adapted to work with AMR grids, then it would also be suitable for our algorithm.

3 Algorithm

We base our crack-free isosurface extraction method on previous work using dual grids and procedurally generated stitch cells [15]. Our main design goals for improving this approach were: (i) enabling data parallel stitch cell generation and isosurface extraction that operate on individual AMR boxes separately, and (ii) maintaining a rectilinear grid representation as long as possible. As discussed in the previous section, we achieve the first goal—the ability to handle boxes individually—using ghost cells.

Like in the previous approach [15], we use stitch cells corresponding to linear VTK cells (hexahedron, pyramid, wedge and tetrahedron) with values specified at the defining vertices. Stitch cells connect a box to its neighboring boxes in the same level, or containing/adjacent boxes in the coarser parent level. To construct stitch cells and assign values to their vertices, we require access to adjacent samples in these boxes. Consequently, for our approach, a single layer of ghost cells around a box is always sufficient to determine appropriate stitch cell vertex values. To ensure that stitch cell vertex values are consistent, the values in the ghost cells must satisfy the following conditions:

- Stitch cell generation requires one layer of ghost data around the entire mesh to create stitch cells to all surrounding grids. The only exceptions are boundaries that coincide with the boundary of the domain. Here, there is no need to connect to other grids and hence no need for ghost data.

- If the ghost cell coincides with the cell of an adjacent box, stitch cell generation needs access the value of that adjacent cell; the ghost cell must be filled with the value from the adjacent box.

- If no neighboring box overlapping the ghost cell exists in the same level, the stitch cell must be filled with the value from the cell it refines in the parent level. (Since a level is always completely contained in its parent level, there will always be such a cell.) That way, several cells in the fine box (the number of cells corresponds to the refinement ratio) are filled with the value of the coarse grid cell, which simplifies look-up for connecting the coarse grid value in stitch cell generation.

- When generating ghost data for a box in level i, only data in the same level i and the parent level $i-1$ are considered. Data from finer levels can be ignored since we generate stitch cells connecting a level to its parent level. Data values in a finer level (greater than i) are handled when generating stitch cells for boxes in that level.

Section 3.1 describes generating this ghost information. This layer of ghost cells around the current grid, along with access to the bounding box hierarchy of AMR data, fully supports pure local stitch cell generation on a per grid basis as discussed in Section 3.2.

In simulations, ghost data is normally used at the boundaries to support, e.g., the computation of gradients using only data locally available. For visualization purposes, these cells are typically blanked out, or any generated geometry corresponding to them is removed. It is possible to use a similar concept to handle cells in a box that are invalidated by a finer resolution box. Instead of removing individual cells from a box—either converting boxes to unstructured meshes or a set of boxes that leaves out refined regions—it is more computationally and storage efficient to keep values in these cells and mark them as "ghost." This gives rise to a new type of "ghost cell," i.e., one ghost cell marked as invalid because there is a more accurate data representation available. We handle ghost data at the boundaries as well as ghost data due to invalidation by finer grids, by using an array of flags that specify for each grid cell whether it is a ghost cell or not. If a cell is flagged as ghost, the flag also specifies its type. All visualization algorithms operate on the original rectilinear grids and use the ghost array information to blank out ghost cells. This is the standard implementation in VisIt. In the dual grid, a cell is marked as ghost if any of the cells in the original grid corresponding to its vertices is labeled as a ghost cell, i.e., if it connects any samples that are flagged as invalid.

3.1 Ghost Cell Generation

The ghost cell generation happens immediately after the boxes are partitioned over the processing elements and loaded. When the data

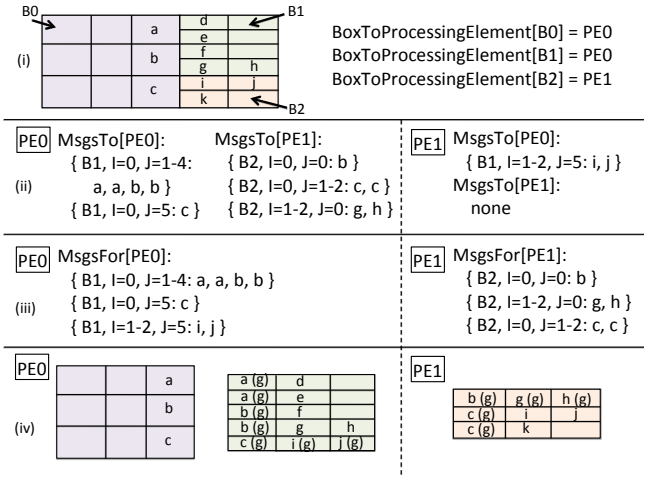


Figure 2. An example of ghost cell generation. (i) shows three boxes (B0, B1, and B2), and their relationship. Only the values that need to be exchanged are labeled. (i) also shows the result of the **Initialize** phase, with BoxToProcessingElement stating that B0 and B1 reside on Processing Element #0 (PE0) and P2 on Processing Element #1 (PE1). (ii) shows the result of the **Pack** phase. Each processing element constructs messages to the other processing elements. The messages contain the destination box, the data, and the location of that data on the destination patch. (iii) shows the result of the **Exchange** phase. Each processing element now has all of the messages for its own patches. (iv) shows the result of the **Unpack** phase. The boxes now overlap spatially with their ghost data, although they are displayed apart for clarity. Cells that are marked as ghost have a "(g)" in their label.

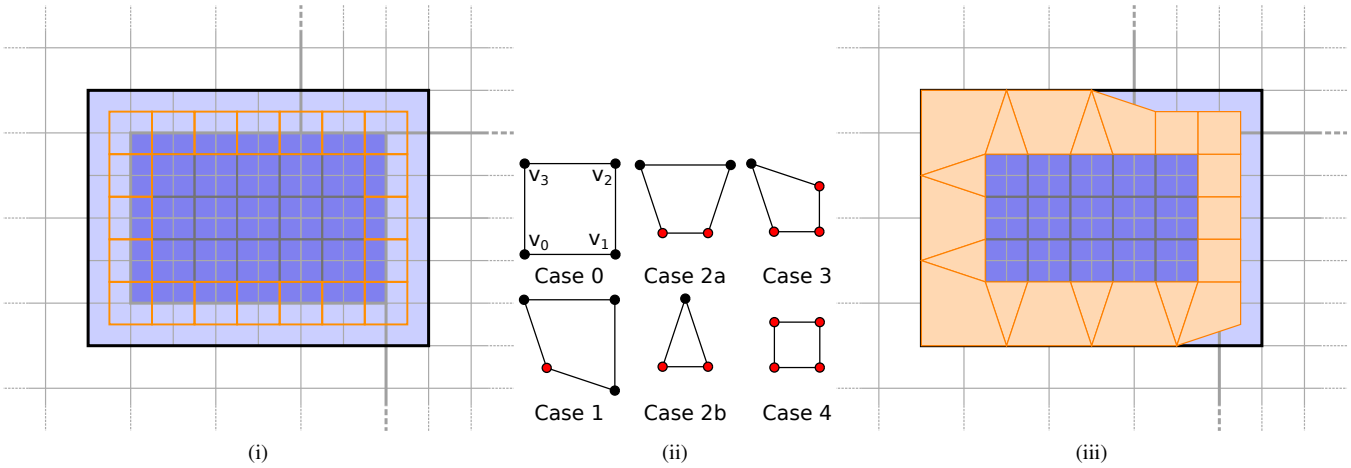


Figure 3. Stitch cell generation. We map dual grid cells (i) that contain at least one ghost cell (orange) to stitch cells using a case table (ii) and obtain stitch cells (iii).

is loaded, information about the boxes is also loaded and exists on every processing element. This information is called *BoxAbutment* and, for some box B, $BoxAbutment[B]$ lists the boxes, B_0, B_1, ... B_{n-1} that B abuts with and the locations of those abutments. This information includes abutment for boxes at the same refinement level and the nesting for boxes at different refinement levels. The ghost cell generation process requires collective communication; all processing elements enter and exit the routine at the same time.

The ghost cells are created via four phases:

- **Initialize:** Each processing element identifies which boxes it has loaded. The processing elements then exchange information to create a map of which processing element each box resides on. This map is called $BoxToProcessingElement$.
- **Pack:** Every processing element iterates over the boxes it owns. For each box B, its abutting patches B_i are determined using $BoxAbutment[B]$. For each of these B_i, one layer of cells in the abutment region is converted to a message and this message is appended into the overall message for the processing element designated by $BoxToProcessElement[B_i]$.
- **Exchange:** Each processing element sends its messages to the others. This is implemented with an `MPI_Alltoallv` command, allowing the MPI implementation to choose a communication paradigm best suited for making effective use of the interconnect.
- **Unpack:** Every processing element extends each of its boxes with ghost cells. The necessary data for the ghost cells is available from the incoming messages received during the Exchange phase.

An example of this process is described in Figure 2.

3.2 Stitch Cell Generation

In addition to using ghost cells to parallelize stitch cell generation, we also wanted to simplify the implementation of stitch cell generation. The original stitch cell generation approach [15] considered many special cases, requiring a substantial implementation effort. To reduce this effort, we consider the dual grid of a box that also includes all of its ghost cells. Using this "complete" dual grid, it is possible to view stitch cell generation as "mapping" dual grid cells that have one or more vertices corresponding to ghost zone cells of the original box to stitch cells via a case table.

Figure 3(i) illustrates this stitch cell generation approach for a 2D box, marked as a bold, black rectangle in the figure. Within the box, a blue background indicates regular cells, and a light blue background marks ghost cells. Bold grey lines indicate two adjacent boxes in the same level that need to be considered when generating stitch cells. The figure shows the complete dual grid as orange and grey rectangles. Grey rectangles make up the dual grid of the actual box and connect only vertices corresponding regular cells of the box. Orange rectangles contain at least one vertex corresponding to a ghost cell and consequently connect the box to its surroundings. Using a case table, it is possible to convert these dual grid cells into stitch cells.

Figure 3(iii) shows the stitch cells resulting from this conversion. To construct a case table for this mapping, we observe that the type of stitch cell depends on two criteria. To convert a given dual grid cell into a stitch cell, the first step is to determine which of its vertices belong to the parent level and which of its vertices belong to the child level. This configuration determines the major case. Figure 3(ii) shows that in 2D, five base cases exist that correspond to zero, one, two, three or four vertices belonging to the child level. It is possible to derive all major cases from these five base cases.

Some of the major cases can have sub-cases, depending on whether any of the vertices belonging to the parent level correspond to the same parent level vertex. Case 2a connects two child level vertices to two parent level vertices. Since the dual grid used to generate stitch cells is at the resolution of the child level, and since each parent level cell contains multiple refined cells (corresponding to the refinement ratio), it is possible that the two vertices belonging to the parent level are actually a single parent level vertex. This case occurs, e.g., for the second orange dual grid cell in the top row of Figure 3(i). In that case, it is necessary to generate a stitch cell connecting to that single vertex. We note, in two dimensions Case 2 is the only case with sub-cases. Case 1 is the only other case with more than one parent level vertex where such a configuration may be possible. However, along both axes, the parent level vertex is connected to a child level vertex. This means that along both axes there is a crossing from a refined to an unrefined region. Since a parent cell is either refined or unrefined, this connection implies that along each axis there is a crossing of parent cells and no two vertices can map to the same parent level grid cell.

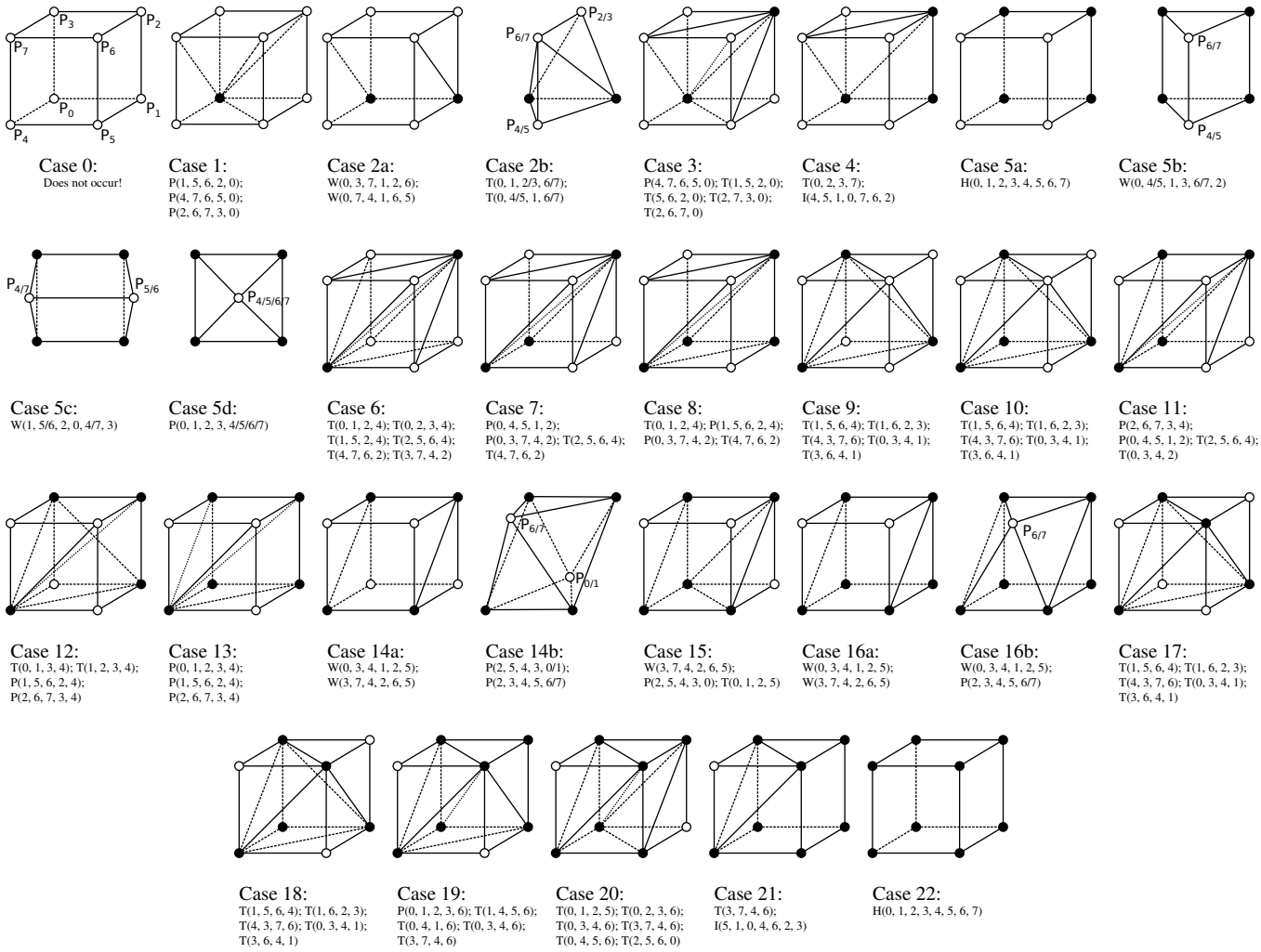
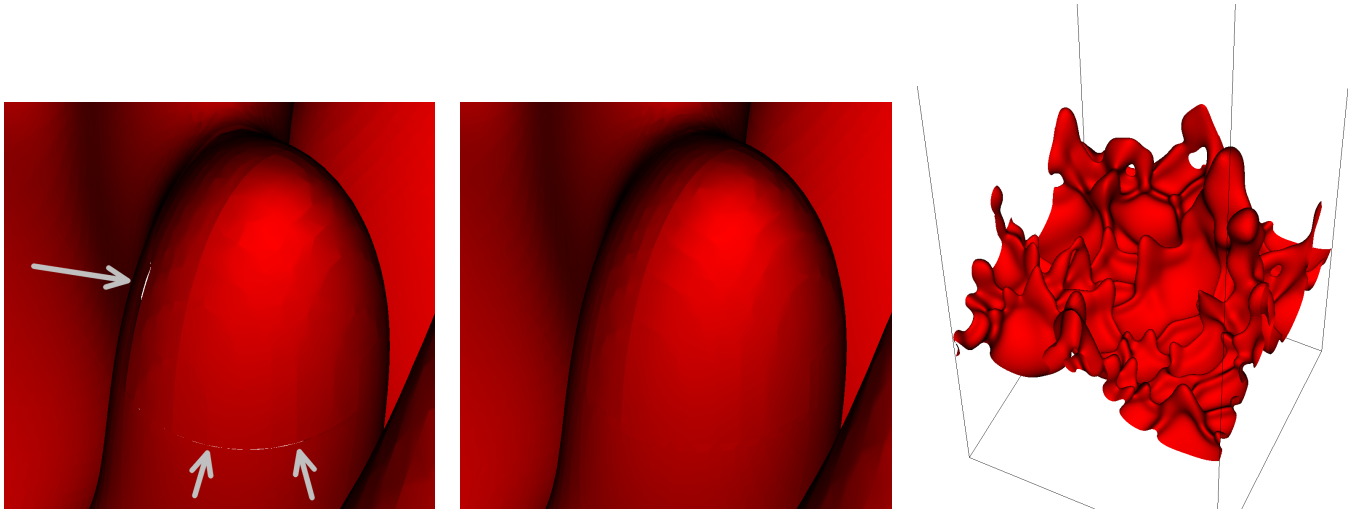


Figure 4. 22 base cases for 3D stitch cell generation from dual grid cells. The tesselation into cells is the same as that produced by the orignal approach [15]. Case 0 does not occur since at least one vertex is refined. We use it to show the numbering of vertices. Tesselation subdivides into pyramids (P), wedge (W), tetrahedra (T), hexahedral cells (H) and the irregular cell type (I) described by Weber et al. [15]. Vertices are specified in the order expected by VTK.

For the purpose of determining whether the two parent level vertices are identical, we use the index of grid cells of the box. If we start counting indices at -1 along each axis, the cell with index 0 will be the first non-ghost cell of a patch. For a fixed refinement ratio between levels, we look at the index of the cell along an axis and compute the remainder modulo the refinement ratio. Consider the dual stitch cell having box cell (i, j) as origin. This dual cell is defined by the box cells (i, j), $(i+1, j)$, $(i+1, j+1)$ and $(i, j+1)$ as vertices. If the remainder of $i+1$ modulo the refinement ratio is zero, this original cell lies in a different parent level cell than the cell with index i, and the two parent level vertices in the i direction are different. If the remainder of $i+1$ module the refinement ratio is non-zero, the two parent level vertices are identical in the i direction. The same method can be used in the j direction. The fact that we use the same coarse cell value to fill all ghost cells overlapping it simplifies data access, as we can read from any ghost cell the correct value for the parent level cell.

In 2D, our approach works as follows. We first use the bounding box AMR hierarchy information to compute an array that specifies an integer identifier of the neighboring box overlapping this ghost cell, or -1 if there are none and the cell corresponds to the parent level. We then iterate over all dual cells that contain at least one ghost cell or the original box. We compute a case number analogous to marching cubes by setting those bits in a 4-bit integer corresponding to refined vertices. Whether a vertex is refined or not can be determined using the previously computed neighboring box information. In the case table, we also store whether we need to check along the i or j direction to determine the sub-case for Case 2. We then use the method outlined in the previous paragraph to compute the sub-case. After, we look up appropriate values for child and parent level vertices according to the global index. To avoid creating the same stitch cell for multiple grids, we follow the convention that a stitch cell belongs to the grid with the lowest integer identifier. Using the neighborhood information, we discard any stitch cells that contain a vertex of a same-level patch with a lower integer identifier.

The 3D case works analogous. Similar to marching cubes without inversion there are 22 base cases [13], which we show in Fig-



(i) (ii) (iii)

Figure 5. Isosurface (temperature of 1225K) for the hydrogen flame data set. (i) Close-up view of the isosurface extracted via re-sampling to a vertex-centered format. Cracks are easily visible. (ii) Close-up view of the same region extracted using our new method. There are no discontinuities in the isosurface. (iii) View of the entire crack-free isosurface extracted using our new approach. The isosurface consists of approximately 2.2 milion triangles.

ure 4. We use these base cases to generate tessellations for all refinement configuration. Unlike in the 2D cases, it is now possible to have either two (e.g., Case 2b, Case 5b/c) or four (only Case 5d) coarse vertices corresponding to a single parent level vertex. Thus, we have many cases without sub-cases, a few cases (Case 2, 14 and 16) with two sub-cases and one case (Case 5) with four sub-cases. We store information in the case table: how many sub-cases there are for each case and along which axes we need to check for identical parent level vertices to compute a sub-case. We then read the stitch cell from the case table.

4 Results

To test our algorithm we examined results from two data sets on the "Hopper" system at the National Energy Research Scientific Computing Center (NERSC). "Hopper" is a Cray X6 with 6,384 nodes comprising 153,216 processor cores connected by a proprietary "Cray Gemini Interconnect" that share 212TB of memory and achieve a peak performance of 1.28 Petaflops/s. Each node contains 24 cores—provided by two twelve-core AMD "MagnyCours" 2.1GHz processors—with 32GB (6,000 nodes) or 64GB (384 nodes) of shared memory. Each core has its own L1 and L2 caches, with 64KB and 512KB respectively, and six cores on the "MagnyCours" processor share one 6MB L3 cache. "Hopper" provides two 1PB Lustre file systems as local scratch space. Each scratch space file system has peak data transfer performance of 35GB/s. Hopper also has access to the NERSC global file system, which is mounted on all NERSC systems and has a peak data transfer performance of 15 GB/s.

The first data set is a relatively small 3D BoxLib AMR simulation of a hydrogen flame. It consists of 2,581 boxes in three hierarchy levels containing a total of 48,531,968 grid cells. The simulation data size for all 22 scalar variables is 8.1GB, which is approximately 377MB per scalar variable. Figure 5 highlights the difference between an isosurface extracted from a re-sampled data set and the continuous isosurface provided by our new approach. We stored this data set on the local scratch file system.

The second data set is a larger 3D BoxLib AMR simulation of a methane flame. This data set consists of 7,747 boxes in 3 levels containing a total of 1,525,420,032 grid cells. The data set has a total size of 592GB for 52 scalar variables, which corresponds to a size of approximately 11.4GB per scalar variable. We stored this data set on the global NERSC file system. Figure 6 shows an isosurface extracted using our approach. Both data sets were provided by the Center for Computational Sciences and Engineering (CCSE) at the Lawrence Berkeley National Laboratory (LBNL).

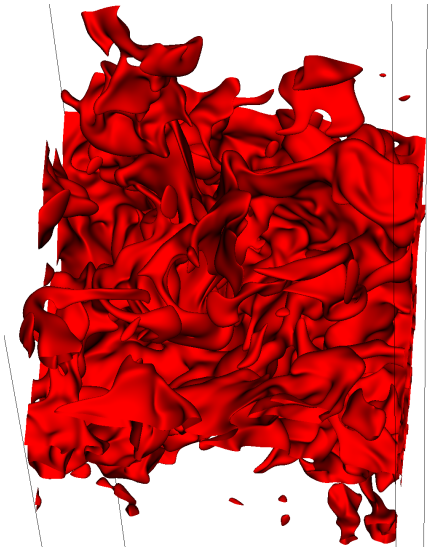


Figure 6. Isosurface (temperature of 1225K) for methane flame data set extracted using our algorithm. The isosurface consists of approximately 29.7 milion triangles.

To test performance and examine any scaling limitations, we executed a full visualization pipeline on these test data sets that included disk I/O, ghost data generation, stitch cell generation, and contouring (of the entire data consisting of dual grids and stitch

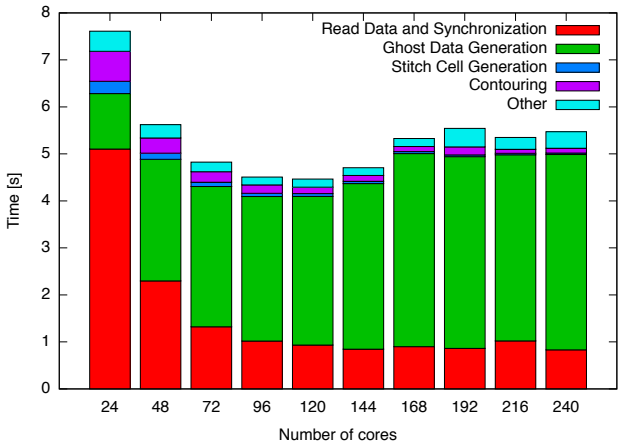


Figure 7. Total runtime of a full visualization pipeline execution on the hydrogen flame data set under strong scaling. Bars are colored by each phase of the pipeline.

cells). We ran this operation on a variety of core counts on the "Hopper" system in a strong scaling mode.

Figure 7 shows the runtime of this visualization operation on the smaller hydrogen flame data set. (Note that the "Other" time includes some final synchronization, e.g., due to a nonuniform distribution of geometry generated from the contour operation on each task.) In this figure, we see excellent scaling of the stitch cell algorithm, as well as of the following contouring operation, even on this small data set. Unfortunately, we reach the best overall performance with this visualization pipeline at only 120 cores.

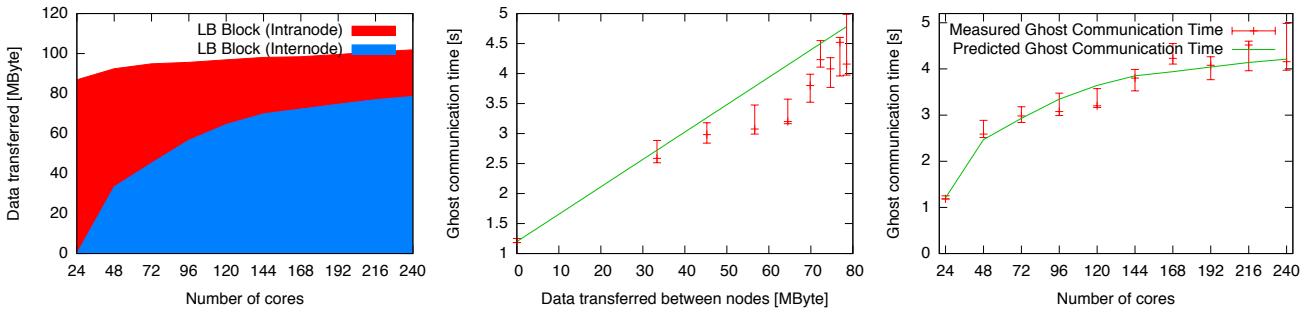
Although disk I/O is one factor limiting the scaling in this example, the main bottleneck is the ghost data generation phase, as we see its runtime increase with core count. One hypothesis for this effect is that as the number of tasks increases, the number of *adjacent* domains each task can possess decreases, and thus the number of ghost cells which must be sent to another MPI task will increase. To explore this effect further, we examined the total amount of data communicated among the tasks during the ghost cell generation phase. The results are shown in Figure 8(i). We see that between 24 and 240 cores, the amount of data communicated among MPI tasks does increase. However, this increase is only approximately 15%, and so the increased quantity of data alone is insufficient to explain the greater runtime of almost 4× in ghost cell generation across this same range of core counts. Instead, the increase in runtime is mostly due to inter-node communication. For example, each node on "Hopper" contains 24 cores, and so when we go from 24 to 48 cores, we now must traverse the system interconnect during the ghost cell communication phase, and as we add more nodes, a greater percentage of communication traffic must cross compute node boundaries. Figure 8(ii) shows a plot of of ghost communication time as a function of data transferred between the nodes. The error bars show the timings for three runs—the maximum and minimum time as boundaries of the bar and the third time as an additional mark between them. We note, the ghost zone creation time increases close to linearly with the data transferred between different nodes. Figure 8(iii) illustrates that a very simple communication model can predict communication times fairly accurately. For predicting ghost zone generation times, we assume two data transfer rates: one for transfers within a node and another rate for transfers between nodes.

We also examined performance of this same visualization pipeline on the larger methane flame data set. The results, shown in Figure 9, show a better scaling performance for total runtime on this larger data set than on the smaller hydrogen flame data set. Note that these results no longer show an increased runtime in the ghost cell generation for increased core counts. There are two reasons for this. First, on a larger data set, the ghost cells are a smaller percentage of the total data set size. Second, as this problem was too large to run on fewer than three compute nodes, runs on even the smallest number of cores already involved a substantial portion of inter-node communication.

The individual contributions of each phase of the computation towards these runtimes are shown in Figure 10. Here, we see clearly the strong scaling of the stitch cell generation; on this and on the smaller hydrogen flame problem, it scales consistently up to the largest tested number of tasks and composes the smallest runtime among the pipeline phases. We also see that the runtime of ghost data generation is approximately flat up to 312 cores for a problem of this size; the greatest source of scaling inconsistency on this problem is due to the I/O phase.

5 Conclusions and Future Work

We have presented a novel algorithm for artifact-free isosurface generation for AMR data in a parallel setting and demonstrated its performance. The algorithm is ideal for the parallel data flow network environments that are currently popular and used in tools



(i) (ii) (iii)

Figure 8. (i) Total amount of data transferred globally during the ghost cell communication phase of a visualization pipeline execution on the hydrogen flame data set. (ii) Ghost communication time as a function of inter-node data communication. (iii) Comparison of actual ghost zone communication time to a simple model that assumes two constant data transfer rates for intra- and inter-node communication.

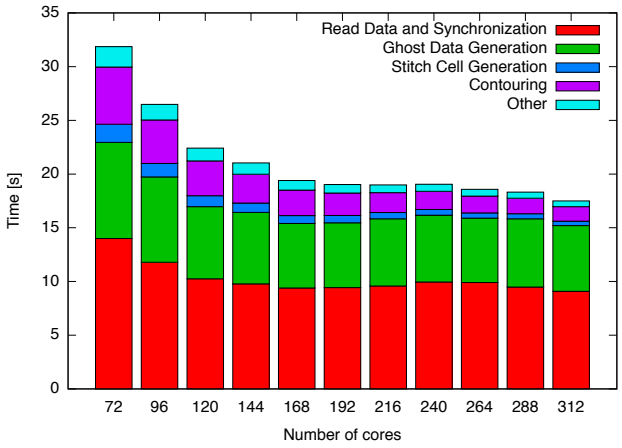


Figure 9. Total runtime of a full visualization pipeline execution on the methane flame data set under strong scaling.

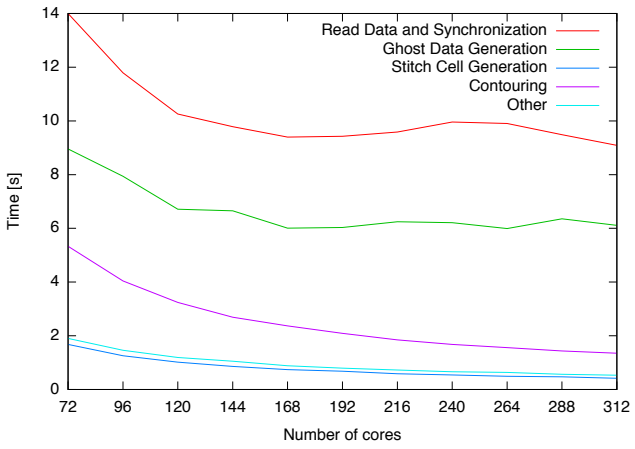


Figure 10. Runtime of each phase of a full visualization pipeline execution on the methane flame data set under strong scaling.

like VisIt and ParaView: it can be implemented as a filter, works with very large AMR data sets, and it makes no assumptions about which boxes are present on each processing element. Further, the stitch cell generation portion is fast and scales well.

Our algorithm depends on the presence of ghost data. Sometimes this ghost data is readily available from the simulation code, especially in an *in situ* setting. But, to demonstrate that our algorithm can work in all circumstances, we also described a method for dynamically generating ghost data for any block-structured AMR data set. This method can be inefficient, particularly for smaller data sets where ghost cells are a large proportion of the total number of cells; writing a faster ghost data generation module would be excellent future work. Of course, the primary contribution we describe is the stitch cell generation algorithm itself, which works in a distributed memory parallel setting, and our algorithm would be compatible with any future ghost data generation modules. It should be possible to use the case table approach for AMR data sets where a child level is not surrounded by a layer of cells, such as Enzo AMR simulations and handle the same range of data sets as the approach by Moran et al. [11]. The main modifications necessary to support such data sets are implementing ghost cell communication between arbitrary levels and changing the criterion to determine whether two parent level vertices in a ghost cell are identical.

Acknowledgments

This work was supported by the Director, Office of Science, Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 and DE-AC05-00OR22725 and the use of resources at the National Energy Research Scientific Computing (NERSC) Center. We thank Terry Ligocki from the LBNL Applied Numerical Algorithms Group (ANAG) for generating data sets used to debug and test our method. We also thank the members of the LBNL CCSE for providing the data sets used to benchmark our method.

Disclaimer

References

[1] Applied Numerical Algorithms Group. Chombo. `https://commons.lbl.gov/display/chombo/`.

[2] M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, 1989.

[3] G. L. Bryan, T. Abel, and M. L. Norman. Achieving extreme resolution in numerical cosmology using adaptive mesh refinement: resolving primordial star formation. In *Proc. ACM/IEEE Supercomputing Conference*, 2001. doi: 10.1145/582034.58204.

[4] Center for Computational Sciences and Engineering. Boxlib. `https://ccse.lbl.gov/BoxLib/`.

[5] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, K. Bonnell, M. Miller, G. H. Weber, C. Harrison, D. Pugmire, T. Fogal, C. Garth, A. Sanderson, E. W. Bethel, M. Durant, D. Camp, J. M. Favre, O. Rübel, P. Navrátil, M. Wheeler, P. Selby, and F. Vivodtzev. VisIt: An end-user tool for visualizing and analyzing very large data. In *Proc. SciDAC 2011*. `http://www.mcs.anl.gov/uploads/cels/papers/scidac11/final/childs_hank.pdf`.

[6] Computational Engineering International Inc. EnSight User's Manual.

[7] D. C. Fang, G. H. Weber, H. Childs, E. Brugger, B. Hamann, and K. Joy. Extracting geometrically continuous isosurfaces from adaptive mesh refinement data. In *Proc. 2004 Hawaii International Conference on Computer Sciences*, pages 216–224, 2004.

[8] M. Isenburg, P. Lindstrom, and H. Childs. Parallel and Streaming Generation of Ghost Data for Structured Grids. *IEEE Computer Graphics and Applications*, 30(3):32–44, 2010.

[9] C. C. Law, A. Henderson, and J. Ahrens. An Application Architecture for Large Data Visualization. In *Proc. IEEE Symposium on Parallel and Large-data Visualization (PVG)*, pages 125–128, 2001.

[10] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (Proc. ACM SIGGRAPH 87)*, 21(4):163–169, 1987.

[11] P. Moran and D. Ellsworth. Visualization of AMR data with multilevel dual-mesh interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1862–1871, 2011.

[12] K. Moreland, L. Avila, and L. A. Fisk. Parallel unstructured volume rendering in ParaView. In *Visualization and Data Analysis 2007, Proc. SPIE-IS&T Electronic Imaging*, volume 6495, page 64950F, 2007. doi: 10.1117/12.704533.

[13] G. M. Nielson. On marching cubes. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):341–351, 2003.

[14] R. Shu, C. Zhou, and M. S. Kankanhalli. Adaptive marching cubes. *The Visual Computer*, 11(4):202–217, 1995.

[15] G. H. Weber, O. Kreylos, T. J. Ligocki, J. M. Shalf, H. Hagen, B. Hamann, and K. I. Joy. Extraction of crack-free isosurfaces from adaptive mesh refinement data. In *Hierarchical and Geometrical Methods in Scientific Visualization*, pages 19–40. Springer Verlag, 2003.

[16] R. Westermann, L. Kobbelt, and T. Ertl. Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces. *The Visual Computer*, 15(2):100–111, 1999.

[17] A. Wissink, R. Hornung, S. Kohn, S. Smith, and N. Elliott. Large scale parallel structured AMR calculations using the SAMRAI framework. In *Proc. ACM/IEEE Supercomputing Conference*, 2001. doi: 10.1109/SC.2001.10029.