

SANDIA REPORT

SAND98-1224 • Revised

Unlimited Release

Printed March 1999

Supersedes SAND98-1224,
dated June 1998

Dynamic Isosurface Extraction and Level-of-Detail in Voxel Space

Peter B. Lamphere, John M. Linebarger and Arthurine R. Breckenridge

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Approved for public release; distribution is unlimited.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

Dynamic Isosurface Extraction and Level-of-Detail in Voxel Space

John M. Linebarger
Information Systems Applications Department
Sandia National Laboratories
P.O. Box 5800, MS 1137
Albuquerque, NM 87185-1137

Peter B. Lamphere
Arthurine R. Breckenridge
Computer Architectures Department
Sandia National Laboratories
P.O. Box 5800, MS 0318
Albuquerque, NM 87185-0318

Abstract

A new visualization representation is described, which dramatically improves interactivity for scientific visualizations of structured grid data sets by creating isosurfaces at interactive speeds and with dynamically changeable levels-of-detail (LOD). This representation enables greater interactivity by allowing an analyst to dynamically specify both the desired isosurface threshold and required level-of-detail to be used while rendering the image. A scientist can therefore view very large isosurfaces at interactive speeds (with a low level-of-detail), but has the full data set always available for analysis. The key idea is that various levels-of-detail are represented as differently sized hexahedral "virtual voxels," which are stored in a three-dimensional binary tree, or *kd-tree*; thus the level-of-detail representation is done in voxel space instead of the traditional approach which relies on surface or geometry space decimations. Utilizing the voxel space is an essential step to moving from a post-processing visualization paradigm to a quantitative, real-time paradigm. This algorithm has been implemented as an integral component of the EIGEN/VR project at Sandia National Laboratories, which provides a rich environment for scientists to interactively explore and visualize the results of very large-scale simulations performed on massively parallel supercomputers.

Index terms: Level-of-detail, isosurface extraction, voxel space, structured grids, *kd-trees*, multi-resolution.

1. Introduction

A terabyte of information has become easy to generate; indeed, data sets of this size are regularly being produced on the massively parallel Intel teraflop supercomputer at Sandia National Laboratories. More recently, clusters of personal computers have begun to reach this scale of computation. How can a scientist hope to comprehend this volume of data? How can groups of scientists on separate continents jointly work with data sets of this size? A single scientist cannot effectively examine this information using outdated text or two-dimensional display methods. Instead, scientists require a globally distributed, collaborative environment with interactive virtual worlds composed of computer generated visualizations. However, this is not easy; the revolution in computing enabled by teraflop-class machines surpasses the capabilities of traditional visualization and data analysis paradigms.

The Dynamic Isosurface Extraction and Level of Detail (referred to as DYNAISOLOD) research has extended Sandia's existing virtual environment tools by building a dynamic isosurface extraction and multi-resolution (level-of-detail, or LOD) system in voxel space for structured grid data sets that scales in immersion and in interactivity. By implementing a very fast isosurface creation algorithm, it gives scientists the freedom to explore their data interactively. This freedom is useless, however, if the scientist cannot render the isosurface at interactive speeds. The DYNAISOLOD tool also enables a scientist to select the detail of the resulting surface, and therefore reduce or increase the complexity (and therefore the rendering speed) interactively. This means that a scientist can both manipulate the entire geometry easily and analyze that geometry with great accuracy when desired, even with the largest data sets.

Isosurface extraction is an important technique for visualizing surfaces found in volumetric data (*i.e.*, data consisting of 3D scalar fields usually produced as the result of a computer simulation). The canonical algorithm for extracting isosurfaces is the "marching cubes" algorithm detailed by Lorensen and Cline [1], although more recently the use of particle systems has been pursued as an alternative approach [13]. Researchers have proposed numerous methods for creating levels-of-detail, including triangle decimation [2], the use of wavelet functions for surface approximation [3], and more recently, progressive meshes [4] and progressive simplicial complexes [5]. However, in all of these approaches, the sets of triangles generated by the marching cubes algorithm are dependent on the chosen isosurface value. For example, a user might request an isosurface showing the extent of the volume with a material fraction of at least 0.55, that is, a surface enclosing the regions which are more than 55% filled with material. In other cases the isosurface value may range from near zero up to large powers of ten, which can be encountered when analyzing simulations of shock, temperature, or pressure waves. With this type of data, interesting isovalues are often unknown in advance. Thus it is always highly desirable to allow the analyst to explore the problem space by dynamically changing the isosurface threshold. However, changing the isosurface value traditionally requires a complete recalculation of the new surface. Since the EIGEN/VR project seeks to allow arbitrary analysis of data immediately and interactively (*i.e.*, without such lengthy recalculation), a faster, dynamic isosurfacing technique provides the required solution.

The NOISE (Near Optimal IsoSurface Extraction) algorithm proposed by Y. Livnat, *et al.* [6] provided the baseline for the required solution. It allows rapid isosurface extraction from structured and unstructured data grids, such that the user can interactively change the isosurface threshold in a visualization environment and quickly view the resulting surface. A multi-dimensional search tree (called a *kd-tree*; see [7]) is used to store the minimum and maximum values of each voxel in the volumetric data set, and the resulting "span space" is searched for a given isosurface threshold. The output is a set of only those voxels that have potential intersections with the new isosurface, which are then triangulated using the marching cubes algorithm and finally rendered.

However, two needed enhancements were observed when we first implemented the NOISE algorithm at Sandia National Laboratories. The first was the sheer number of triangles generated from the ever-increasing growth of our simulation data volumes, which saturated the hardware rendering capabilities even of the largest graphics machines produced in 1997 (the Silicon Graphics, Inc. [SGI] Onyx2 Infinite Reality family). As might be expected, this compromised the interactive response promised by the NOISE algorithm. Clearly, a dynamic way of specifying *both* the isosurface threshold *and* the level-of-detail was needed to achieve an acceptable rendering frame rate when navigating through large simulation data volumes with many time steps. A particular benefit of such a multi-resolution approach is the ability to display a coarse approximation of the data at interactive speeds while the user navigates rapidly, but which also allows a finer approximation when she or he stops to study the details.

The second required enhancement was the solution of the well-known "ambiguous cases" triangulation problem that occurs when using the marching cubes algorithm. Resolution of such ambiguity either adds additional complexity or makes use of the data values in neighboring voxels (which can also have the side effect of improving the shading of the resulting triangles because of the gradient heuristic used; see [8] and [9]). Programs that perform marching cubes triangulation with ambiguity resolution logic are usually run in batch mode. Adding the additional logic to disambiguate the list of voxels returned by the NOISE algorithm during real-time visualization would either have made the frame rate unacceptably slow when an isosurface threshold was dynamically changed, or would have required the addition of additional disambiguation data to each voxel returned from the search (thus ballooning the file size). Fortunately, a straightforward approach that allows unambiguous triangulation of an arbitrary voxel without regard for its neighbors (at the price of occasional surface anomalies) has been published by C. Montani, *et al.* [10], who graciously provided us with a copy of the triangulation table.

The research algorithms for level-of-detail presentations (some of which were mentioned above) make adjustments to the geometry representations and, in our opinion, do not provide real solutions suitable for large-scale data sets. They operate in geometry space (the surfaces), not in voxel space that is most directly associated with the physical simulation code. In other words, each algorithm assumes a stable underlying surface geometry, which is then represented in varying levels of detail. Of course, the ability to dynamically specify the isosurface threshold at visualization time implies that in such an approach there *is* no stable underlying geometry, since the geometry generated depends entirely on the isosurface threshold value chosen. The analyst cannot work with the data in real time if significant delays are introduced while new geometry is created and another algorithm applied to reduce the data representation. Further, surface-based level-of-detail representations no longer work with the original data, only with the generated surfaces. To effectively visualize terabytes of information the visualization process will have to move closer to the computation that generated the data. Currently, most visualization is done by

post-processing the data after a simulation has written its results to a disk file, generally in a format convenient for the simulation code, but not for the visualization programs. This approach has numerous limitations and has been shown not to scale in real time. In particular, three dimensional volume data from a simulation should remain accessible as voxel data for the visualizations; the ability to recover the original data values that a surface represents is required in order to do effective quantitative analysis.

The approach we have taken with dynamic isosurfacing and level-of-detail representations is a powerful extension of the NOISE algorithm. Different levels-of-detail are represented as differently sized virtual voxels and the resolution value is stored as a third dimension of the *kd*-tree. Adjacent voxel cells in the volumetric data grid are joined to create larger "virtual voxels." For example, with structured rectilinear grids, the finest level-of-detail is represented by a $1 \times 1 \times 1$ voxel; coarser levels can be represented as $2 \times 2 \times 2$ voxels, $3 \times 3 \times 3$ voxels, etc. Thus the level-of-detail representation is done in voxel space, not geometry space. Spatial coherence of the resulting surface is largely preserved between virtual voxel resolutions because of the interpolative nature of the marching cubes triangulation algorithm.

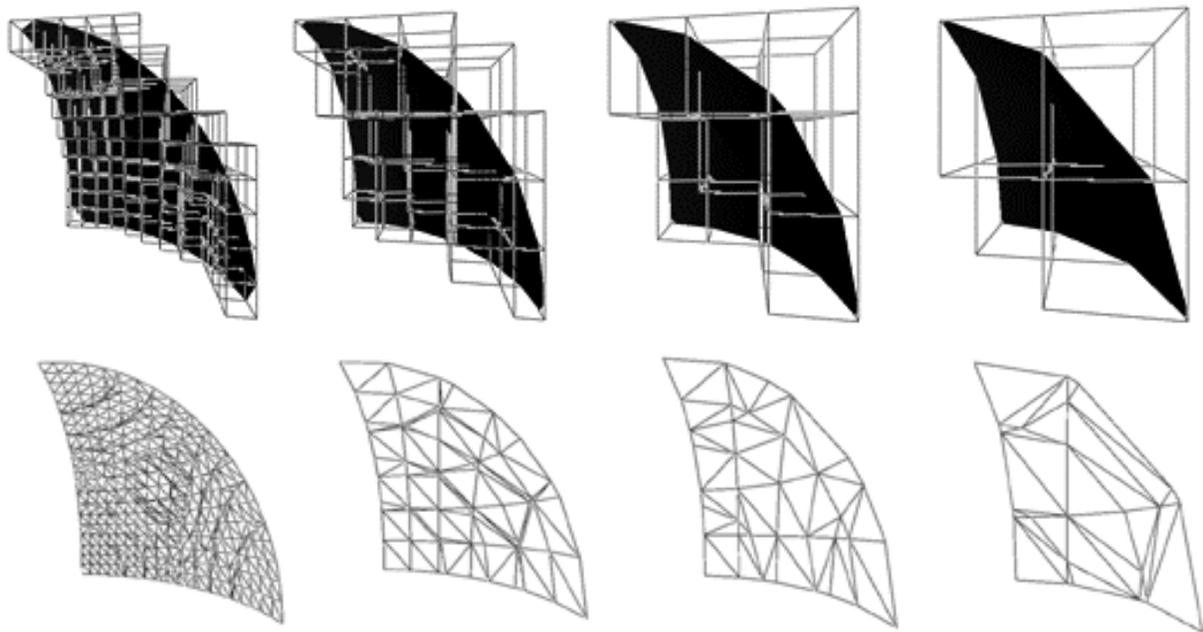


Figure 1: A simple isosurface with successively larger virtual voxels

2. Components of Dynamic Isosurface Extraction and Level of Detail (DYNAISOLOD)

The components of DYNAISOLOD can conveniently be divided into two halves: a parallel portion that uses the Message Passing Interface [MPI] standard to create a LODTree file (a 3D *kd*-tree with associated data), and a visualization-time part that includes search, geometry-creation, and rendering algorithms. Time and space optimizations have been added to each of these two components.

The DYNASOLOD algorithm is a component of Sandia's EIGEN/VR project for analyzing structured and unstructured grid simulation data sets produced by massively parallel supercomputers. The first application has been a proof of concept using structured rectilinear grid data sets produced by a Sandia-developed simulation code, CTH. An additional goal of this project is to compute and visualize data on the supercomputers and then deliver the image-based results to the desktop monitor, which is a complete paradigm shift from utilizing post-processing graphic engines. As a result, care has been taken at each stage of DYNASOLOD development to ensure that the resulting data file is readable from a Java program, with the eventual goal of providing a portable Java client which renders the geometry at the desired isosurface threshold and level-of-detail. The geometry could be created either on a data storage and geometry creation server (for large data volumes) or on the analyst's desktop (for small data volumes). Note that three visualization options exist depending on the capacity of the analyst's workstation: in descending order these options are visualization of geometry created on-the-fly directly from the simulation data volume, visualization of geometry created from the simulation data volume in a preprocessing step, and display of a two-dimensional image generated from the simulation data volume by a visualization server.

3. Parallel Algorithm for Parallel LODTree File Creation

Before describing the file creation algorithm, a brief review of the structure of a *kd*-tree is in order (see [7] for more details). In order to create this easily searchable, multidimensional (*i.e.*, multikey) data structure (such as the 3D data structure of an LODTree), a divide-and-conquer approach is used. First, the data set is partitioned into two halves, one of which contains nodes that have a higher key value than the root node, and the other containing nodes that have lower values. These two halves are again partitioned, but this time according to the second key. The four sub-branches are partitioned a third time according to the third key. Then the process repeats itself again for each of the resulting subtrees, starting again with the first key. This structure allows a very fast search, using any combination of criteria involving the three keys. In our case, this can be envisioned as a three-dimensional binary tree where each branch represents a new dimension. For DYNASOLOD, the three keys are the minimum value, the maximum value, and the level-of-detail of the virtual voxel. This whole partitioning process operates on a set of indexes into the original data set.

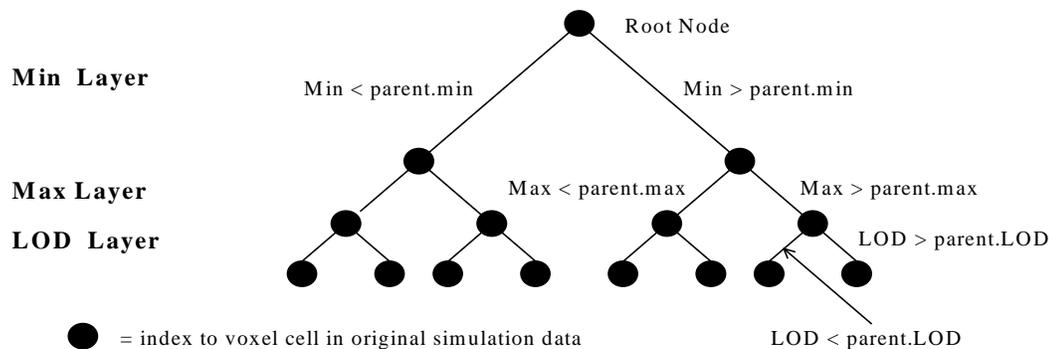


Figure 2: Three-dimensional LOD Search Tree

Like the NOISE algorithm, DYNAISOLOD uses a Quicksort-like partitioning scheme to find a median node and partition the subtrees. However, two significant extensions, were made to the NOISE algorithm. The first was the parallelization of the file creation algorithm, which was implemented using MPI, the Message Passing Interface standard. At each Quicksort-like recursive call, the top half of the node array is partitioned to another processor until the number of processors in the processor array is reached; at that point all additional recursive partitioning is done locally. (This approach to parallelization is frequently called “divide-and-conquer” with a gather at the end; see [11].) A heavily recursive subroutine was written to deterministically predict, based on the number of cells in the simulation data volume and the number of processors in the processor array, the size of the message sent to each processor upon such a recursive split. In addition, it also predicts the size of the resulting sorted message from each processor. Knowing these sizes in advance allows the other processors in the parallel computation to be prepared for work when they receive a message after a recursive array split. It also allows the root processor to determine the correct order in which to receive the partitioned message data from the child processors. Obviously, this strategy implies that the algorithm is most efficient when the number of processors in the processor array is a power of two, but it works correctly for an arbitrary number of processors.

The second extension was the extensive use of a bit-mapped representation of the data in each virtual voxel in order to significantly reduce the size of the search tree. To allow the file to be read by a Java program, compiler alignment of the in-memory data structure written out to the file was carefully considered, the maximum array index value (and thus the maximum array size) was limited to the contents of a 32-bit signed integer, and all numeric fields were defined as signed. (It may be possible to remove this limitation as Java matures.) An abbreviated version of the C++ header file for the structure follows.

```
typedef struct lod_file_header_struct
{
    char signature[ SIGNATURE_SIZE ]; // Version signature
    char source[ SOURCE_SIZE ]; // Source of the data
    int width, height, depth; // Dimensions of the data set
    int num_data_points; // Number of data points
    float min, max; // Min and Max for the tree
    int size; // Number of LOD nodes
    int resolutions; // The number of resolutions
} lod_file_header_t;

// A structure defining the whole in-memory LOD tree.
typedef struct lod_tree_struct
{
    lod_file_header_t header; // The file header information
    int* array_base_indices; // Array of indices into the data
                                // for the base cell vertex
    char* min_max_vertex_coords;
                                // Used to find the minimum/maximum data point in
                                // each cell (keys 1 and 2 for the search)
                                // The first 4 bits of this variable contain the
                                // canonical vertex coordinate of the cell vertex
                                // that has the minimum data value, and the last
                                // 4 bits contain the canonical vertex coordinate
                                // of the cell vertex that holds the max value.
                                //
```

```

// X111X111      <-- bitmap (X is unused bit)
// mxyzaxyz      mxyz = min cell vertex coordinate
//              axyz = max cell vertex coordinate
char* cell_dimensions;
// The size of the cell
// The first three bits of this are flags,
// indicating that the cell is not regular in a
// given dimension. The last 4 bytes contain the
// the regular size of the cell.
//
// X1111111      <-- bitmap (X is unused bit)
// xyzssss      x = is width irregular?
//              y = is height irregular?
//              z = is depth irregular?
//              s = regular cell size (2^4 max)
// Irregular cells occur only at the edges of
// the structured grid data set. Therefore,
// if x, y, or z is irregular, the extreme
// {<x, y, z>} coordinates in the cell are
// equal to the maximum possible width,
// height or depth.
float* data_values; // Array of scalar data values; the dimension
// is width * height * depth
float* data_coords[3]; // Arrays of coordinates associated with the
// data value array, for x, y, z.
} lod_tree_t;

```

As the file structure indicates, an LODTree consists of the *kd*-tree “index” into the structured grid data array as well as the data array itself (and the 3D coordinates that apply to the scalar values in the data array). The “index” is composed of three synchronized arrays (*array_base_indices*, *min_max_vertex_coords*, and *cell_dimensions*), each of which contains *header.size* number of elements. Note that the possibility of irregularly shaped virtual voxels at the edges of the grid is taken into account by this representation, and that the largest possible virtual voxel is 16x16x16. Also note that this compressed bitmapped representation works only for structured (rectilinear) grids.

A more detailed account of the flow of the LODTree file creation algorithm follows:

- A. The first processor initializes the MPI communications group, reads in the file, calculates the minimum and maximum data values for all the voxel cells in the file, and then begins its first Quicksort-like partitioning pass through the data. When the median cell is found, this processor schedules out the top half of the array to the next processor in the processor group and then proceeds to perform the same partitioning algorithm recursively on the bottom part. Each processor in turn, when it receives a message that contains its data, will perform the same partitioning algorithm on its message and farm out the top half to the next processor in the processor group, whose number is deterministically predicted by the highly recursive predictive routine mentioned earlier. It then continues to perform the partitioning algorithm on the bottom half of its array. This array subdivision and subcontracting to other processors continues until the limit of the processor array is reached, at which time the recursive partitioning algorithm is

- performed locally by each process.
- B. For efficiency, messages contain only the minimum amount of virtual voxel information necessary for the partitioning routine to function: the minimum and maximum data values, the level-of-detail, and the index into the full data array contained in processor 0 where all additional information is kept. The index is used by processor 0 to reconstitute the full data later when all of the correctly ordered virtual voxel stub messages have come back from all the processors.
 - C. Experimental analysis has shown that the most time-consuming and expensive part of the creation of the *kd*-tree is the Quicksort-like recursive partitioning (usually at least 90% of the serial run time). Thus the initial parallelization strategy chosen was simply to parallelize the recursive partitioning activities, not the initial calculation of the minimum and maximum data values for each voxel.
 - D. Once all the processors have finished their partitioning chores, processor 0 interrogates each process in turn, in the order that was also dynamically predicted by the recursive prediction routine, and retrieves the message stub back from each processor in correct *kd*-tree order. That message stub order is then used to write out the full *kd*-tree data; the result is a LODTree file, a 3D *kd*-tree. (Optionally, the integrity of the *kd*-tree can then be checked to ensure that the LOD-tree is a true *kd*-tree [*i.e.*, is structured correctly as a three-dimensional *kd*-tree]).

4. File Creation Options and Optimizations

Numerous run-time options and optimizations for the parallel file creation program are available. These options include:

- The maximum number of levels-of-detail desired
- The virtual voxel size increment, in each dimension, for each level-of-detail (defaults to 1)
- A post-processing check of the structural integrity of the LODTree file (*i.e.*, to verify that it is indeed a three-dimensional *kd*-tree)
- The "virtual voxel overlap factor" that specifies by how much an edge voxel can expand in order to prevent rows of small voxels at the edges of the grid.
- An optimization option for shared memory implementations of MPI, which synchronizes memory allocations and message passing associated with recursive splits between processors in order to reduce the overall memory footprint; this option essentially trades memory space for run time.

In addition, the MPI run-time environment provides a parameter that specifies the number of processors in the processor group.

5. Run-time Search and Geometry Creation Algorithm

At run-time we see how the effort spent creating the searchable tree results in much greater efficiency and flexibility for the user. The rendering component of the NOISE algorithm

starts with a search in span space, returning the set of cells which contain the isosurface value (*i.e.*, whose minimum value is less than the isovalue, and whose maximum value is greater). Our multi-resolution DYNASOLOD algorithm is essentially the NOISE algorithm with an extra parameter, the desired resolution (or level-of-detail) of the resulting isosurface.

After the LODTree file has been loaded, the user provides an isosurface threshold value (an isovalue) and a desired level-of-detail. The level-of-detail may be furnished indirectly, perhaps as some function of current navigation speed. The data set is then searched for cells that contain this isovalue and are of the appropriate size to create the correct level-of-detail. Recall that the cells in the data set have been sorted according to the three keys: minimum data value present in the cell, maximum data value present in the cell, and the size of the cell (resolution, or level-of-detail). The cells that we want to find are those with a minimum data value below the isovalue and a maximum data value above that threshold. These cells contain data points that intersect that isovalue, from which the geometric approximation of the isosurface can be generated.

Each node of the tree is interrogated in terms of the appropriate search key, as each level of the tree is partitioned on a different key. For example, the first level (root node) of the tree is searched by the minimum vertex value of the voxel (cell). If the minimum value is higher than the desired isovalue, the left child of the node is searched to find a lower isovalue, and therefore a cell that might contain the isovalue. On the other hand if the minimum value is lower than the desired isovalue, then both children are searched, because both higher and lower minimum values might still be larger than the isovalue.

At the next level, the cell maximum is queried. If the cell maximum is less than the isovalue, we know we need a higher maximum, and therefore must search all nodes along the right child. Otherwise, both children are searched. At the third level, the LOD key is searched. If the LOD of the current node is greater than the desired LOD, we search the left branch. If it is lesser, we follow the right branch. If the LOD happens to be what we want, then we search both branches, because both branches could contain nodes of the appropriate LOD.

This recursive search process continues for all subsequent levels of the tree: the minimum key, followed by maximum key, followed by LOD key and so on until the tree has been exhausted. If at any point during the search, the cell minimum is less than the isovalue, the cell maximum is greater than the isovalue, and the LOD of the cell is equal to the desired LOD, then that cell is added to the list of cells that contain the isosurface. Pseudocode of the search algorithm follows:

```
void search_minimum( branch ) {
    if( minimum value in cell > isovalue ) // isosurface does not
        search_maximum( left branch ); // intersect cell
    else {
        if( maximum value in cell > isovalue && LOD of cell ==
            desired LOD )
            add cell;
        search_maximum( right branch );
    }
}
```

```
void search_maximum( branch ) {
    if( maximum value in cell < isosurface ) // isosurface does not
        search_lod( right branch ); // intersect cell
    else {
        if( minimum value in cell < isosurface && LOD of cell ==
```

```

        desired LOD )
        add cell;
    search_lod( left branch );
}
}

void search_lod( branch ) {
    if( LOD of cell > desired LOD )
        search_minimum( left branch );
    else
        if( LOD of cell < desired LOD )
            search_mininum( right branch );
        else {
            // LOD == desired LOD
            if( minimum value in cell < isovalue &&
                maximum value in cell > isovalue )
                add cell;
            search_minimum( right branch );
            search_minimum( left branch );
        }
}
}

```

Various optimizations can improve the performance even of this efficient algorithm. For example, once we know that the minimum data value for the cell is less than the isovalue, we know that this will be true for all cells on the left branch of the tree (because all cells on the left branch have a smaller key than the parent node). Therefore, we no longer have to test the minimum key past that point. The same thing can be done for the maximum key. For the resolution (level-of-detail) key, it is possible to optimize the search only if we are aware that the level-of-detail desired is either the greatest or the least possible. We can stop testing the LOD keys if the search has passed on to the left (*i.e.*, smaller) subtree of an already minimum LOD node. Since the search is most time-consuming at the lowest level-of-detail, this is a worthwhile optimization.

Once the cells containing the isosurface are found DYNASOLOD generates triangles on a cell by cell basis, just as the Marching Cubes algorithm [1] does for each cell in the data set. First, it finds which corners of the cell are underneath or above the isovalue to determine which edges of the cell the isosurface intersects. From this we can decide how to triangulate the isosurface in the cell. Using the modified (*i.e.*, disambiguated) triangulation table mentioned above, we generate a set of triangles whose vertices are linearly interpolated along the cell edges.

This process results in redundant calculation of vertices, as triangles in each cell share vertices with triangles in neighboring cells. As we calculate each vertex we store it in a hash table, keyed to the edge that it lies on (since the isosurface approximation only intersects each cell edge at most once). When we encounter the same vertex again, we can simply copy the vertex coordinates from the hash table. After the coordinates of the triangles have been calculated, normals are computed by a cross-product, and then stored (via the same hash table) for later averaging with normals of other triangles that have the same vertex. The same hash table can be used to arrange adjacent triangles into triangle strips—a more compact representation that allows faster rendering, although it requires extra computation time to create. This offers a tradeoff between geometry creation and geometry time, which can vary by hardware platform.

6. Rendering Options and Optimizations

In keeping with DYNAISOLOD as a component of a scientific data analysis toolkit, several rendering options are provided and can be user-specified. The first option toggles the direction of the normals generated for the isosurface triangles. Some scientific data sets need normals pointed in the opposite direction, so the facility to choose the direction of the normals was provided. Second is the ability to render wire frame, flat shading, or smooth (Gouraud) shading versions of the generated triangles. The third involves the use of display lists and hash tables both to increase the speed of the rendering and to improve the shading of the resulting surface. The hash table is used to blend normals together for triangles that share vertices. A final option is to use the same hash table to generate triangle strips instead of individual triangles.

The goal is to provide user-specified tradeoffs between speed and accuracy. For ultimate speed, the user might choose the highest level-of-detail and wire frame rendering in order to navigate quickly through the data set; for greatest accuracy, the user might turn the hash table on and specify smooth shading in order to render the surface in the most precise way possible.

7. Results

The observed results are divided into the same categories as DYNAISOLOD itself: off-line parallel file creation and run-time tree search, geometry creation, and rendering.

Many metrics exist to measure the performance of a parallel algorithm (see [11]), but for the purposes of this paper we have chosen actual speedup

$$\frac{T(1)}{T(N)}$$

where the $T()$ is the time function and its argument is the number of processors used, versus expected speedup for a divide-and-conquer parallel algorithm, which is approximately

$$\frac{N}{\log_2 N}$$

where N is the number of processors used. The parallel algorithm contains approximately 25% serial code, which would make the upper bound of the expected speedup of the off-line (*i.e.*, pre-processed) parallel file creation

$$\frac{N}{\beta N + (1 - \beta)} = \frac{4}{(0.25) * 4 + 0.75} = 2.29,$$

where β is the fraction of the program that is serial, using Amhdal's Law. In contrast, the upper bound of the expected speedup is

$$N - (N - 1)\beta = 4 - (4 - 1) * 0.25 = 3.25$$

using the Gustafson-Barsis Law, and is

$$\frac{N}{\log_2 N} = \frac{4}{\log_2 4} = 2$$

using $N = 4$ and a formula for a first order approximation of the speedup of a divide-and-conquer parallel algorithm (see [11]). (Note that communication costs and operating system overhead are intentionally *not* considered by the expected speedup equation models above.)

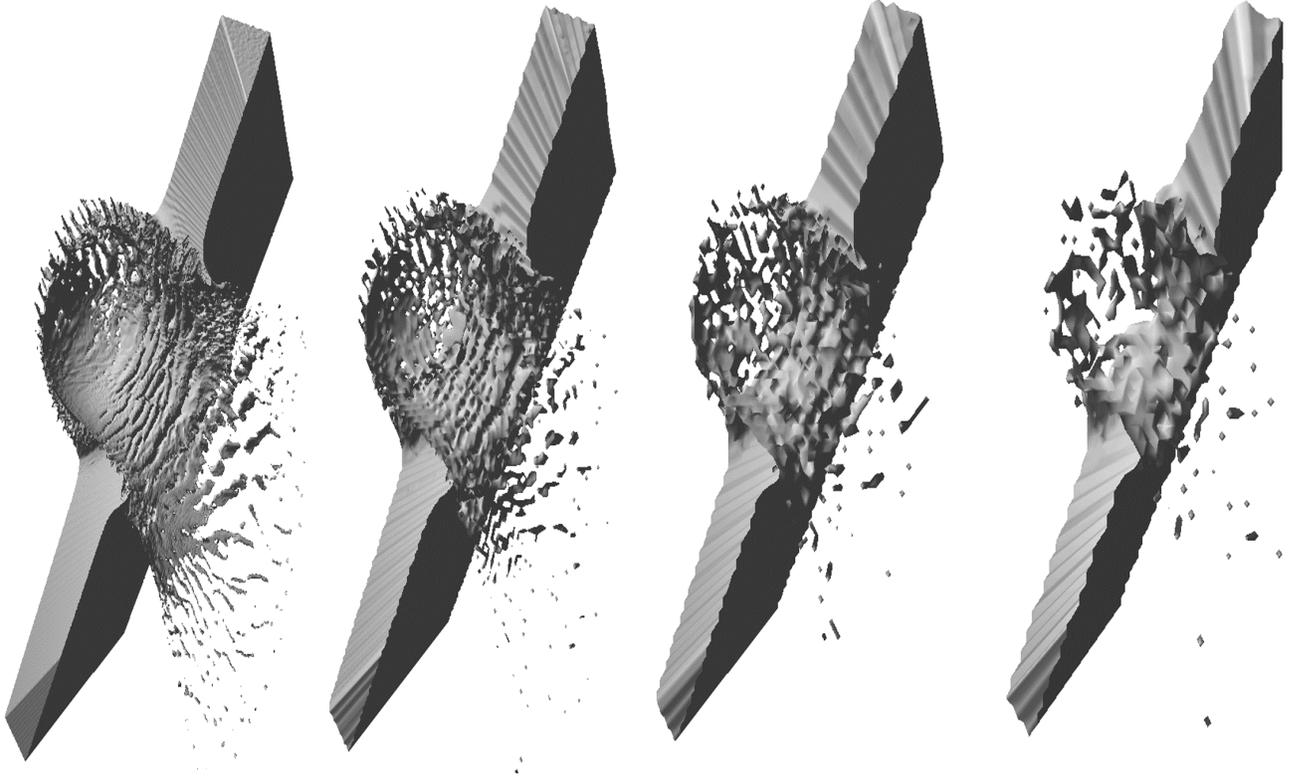


Figure 3: Progressively lower levels of detail applied to a 27 million cell shock physics calculation

Two sets of timing results follow. The first shows the creation of an LODTree file from a 28 million cell data set on an SGI Onyx Infinite Reality system with four R10000 processors and 4 GB of main memory; the second depicts the creation of an LODTree file from a 100 million cell data set on an SGI Onyx2 Infinite Reality machine with 16 R10000 processors and 32 GB of main memory. Each SGI machine utilized a shared memory implementation of MPI. Three timing graphs are shown, one for the creation time of both files, and an additional one for the parallelization speedup results for each file. Both actual and expected speedup for a divide-and-conquer algorithm are plotted on the 27 million cell graph, while only actual speedup is shown on the 100 million cell graph.

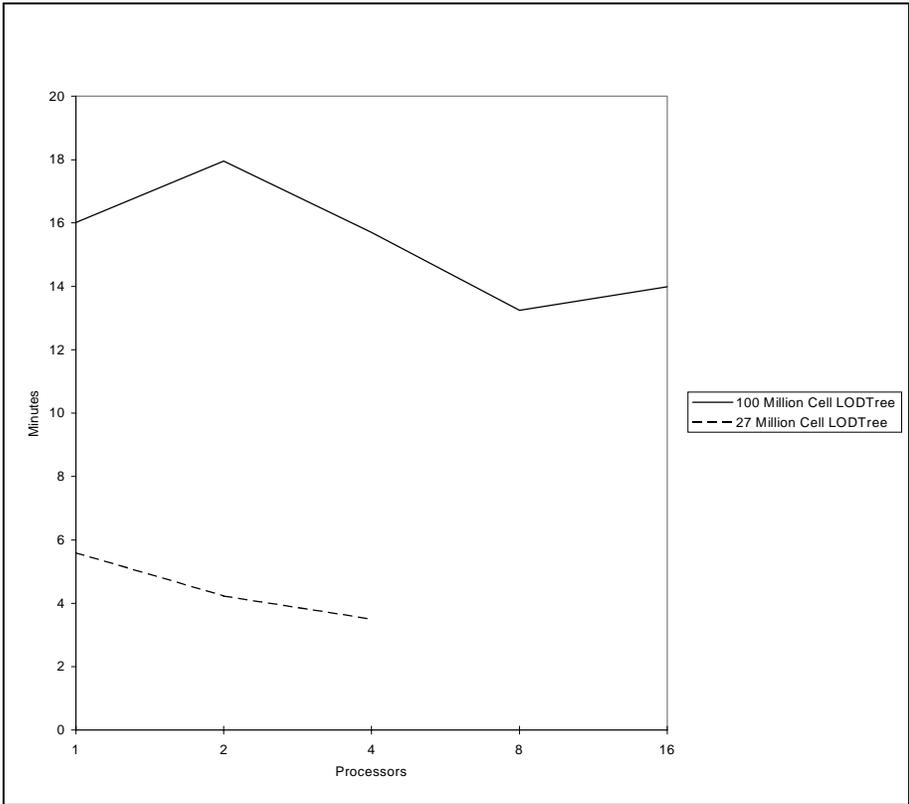


Figure 4: LODTree File Creation Timings

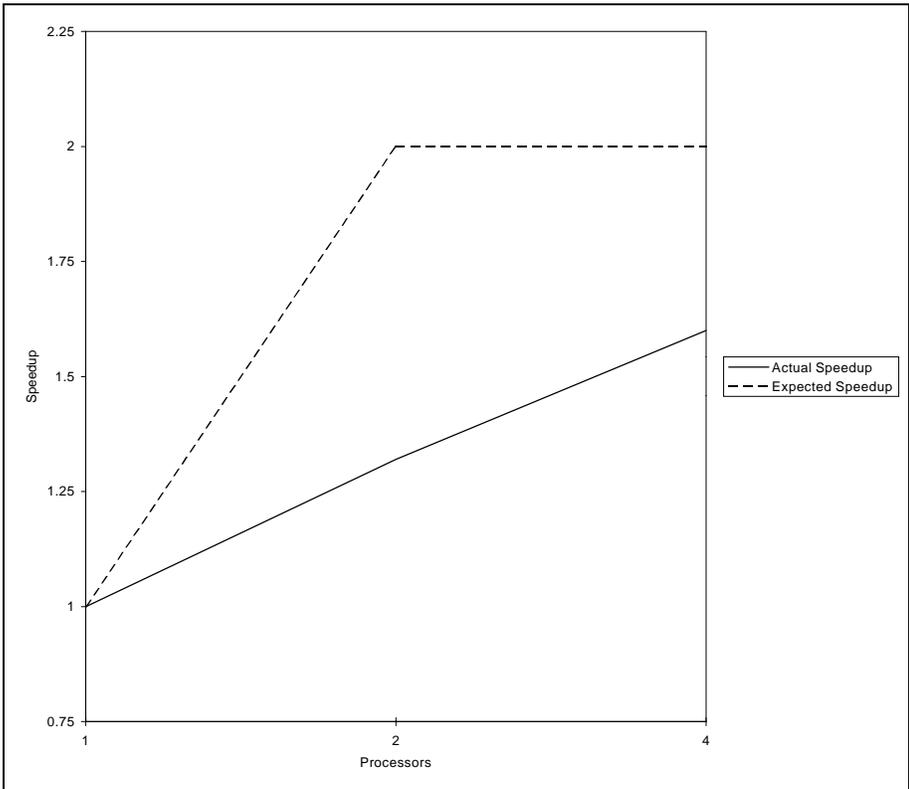


Figure 5: Actual vs. Expected Speedup for a 27 Million Cell LODTree File

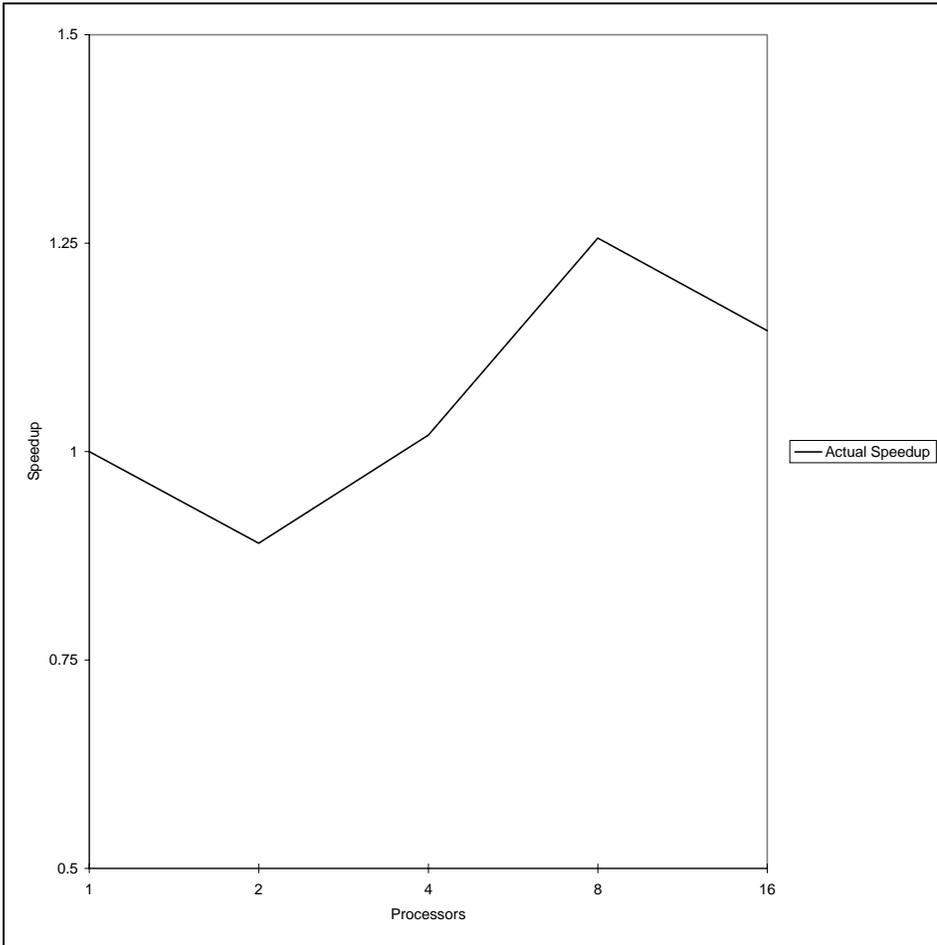


Figure 6: Actual Speedup for a 100 Million Cell LODTree File

Different results were exhibited by the two tests. Whereas the first test demonstrated a nice speedup as processors were added (and at least approached the expected speedup), the second test showed a slowdown as one processor was added, followed by a speedup as other processors were added, then another slowdown when all 16 processors were used. Our best explanation for this phenomenon is two-fold:

1. The large message communication and operating system overhead engendered by handling data sets of this massive size; and
2. The data dependencies of this algorithm. Since it is Quicksort-based, $O(n)$ performance is expected in the average case, but $O(n^2)$ performance can be encountered in the worst case (see [12]). Since this algorithm is only as fast as the slowest processor, its performance can be affected by the particular data being partitioned; the number of swaps required to find the median element is extremely data-dependent.

The storage overhead added to the base simulation volume data is also a concern, given the size of the data sets that we are working with. For each node in the tree, a four byte index is required, as well as two bytes for the min, max and LOD keys (see the description of the LODTree file above). Because of the way that virtual voxel expansion is done, each level of

detail creates additional cells equal to the original number of cells divided by the cube of the cell size. Therefore, there are a total of

$$k \sum_{i=1}^n \frac{1}{i^3}$$

nodes in the LODTree file (where n is the number of levels-of-detail and k is the number of voxels (cells) in the data file). For an infinite number of levels of detail,

$$\sum_{i=1}^{\infty} \frac{1}{i^3}$$

constitutes the Riemann zeta function

$$\zeta(3)$$

which converges to the value 1.20206. This factor provides an upper bound for the fractional number of LODTree cells required for each voxel (cell) in the simulation data file, and thus for the amount of storage required for the LODTree file. The conversion from cells into actual bytes of “index” (*i.e.*, LODTree) overhead required is straightforward. For example, in an LODTree that contains ten levels of detail,

$$\sum_{i=1}^{10} \frac{1}{i^3} = 1.12$$

which multiplied by the number of bytes occupied by each LODTree cell (six) yields 6.72 bytes of overhead per cell in the original structured grid data set. If we use four bytes as an approximation for the storage consumed by each voxel in the data file (the actual value is slightly higher to account for the data points that constitute the edges of the structured grid), we arrive at an approximate byte overhead factor of

$$\frac{6.72}{4} = 1.7.$$

Heavy use of a bit-mapped cell data representation was required in order to achieve this result (see description of the LODTree file above).

However, when comparing the overhead file size, we also need to compare the functionality obtained. Though geometry representations for an isosurface are much more compact, they require a separate representation for each threshold and level-of-detail. Thus, even if a geometry representation were a tenth the size of an LODtree file, the analyst would only be able view ten isosurface thresholds before the static geometry approach became less efficient than the dynamic technique.

The run-time algorithm was tested on one R10000 processor of an SGI Onyx with 4 GB of main memory using three different CTH calculations run on Sandia’s Intel teraflops supercomputer. We tested four different virtual voxel sizes, ranging from one, the original resolution of the data, to four, where each virtual voxel encloses 64 original data points. Also, we tested each data set using the triangle strip geometry representation to determine whether the computational cost of calculating the triangle strips resulted in a greater rendering time payoff.

The results are given in the following three tables and two figures:

LOD	Number of Triangles	Search time (sec)	Geometry Creation (sec)	Rendering Time (sec)
1 (High)	560,688	0.35	10.29	4.95
2	137,212	0.10	2.05	2.42
3	60,164	0.03	0.96	0.98
4 (Low)	33,760	0.02	0.42	0.54
LOD	Number of Triangle Strips	Search time (sec)	Geometry Creation (sec)	Rendering Time (sec)
1 (High)	67,264	0.32	16.29	2.66
2	13,538	0.09	3.28	1.89
3	7,754	0.03	1.48	0.83
4 (Low)	3,522	0.02	0.70	0.44

Table 1: LOD Rendering Performance Results using a 27 Million Cell Data Set

LOD	Number of Triangles	Search time (sec)	Geometry Creation (sec)	Rendering Time (sec)
1 (High)	465,856	0.25	6.16	3.01
2	110,528	0.08	1.32	1.65
3	48,158	0.03	0.56	0.68
4 (Low)	26,704	0.02	0.30	0.38
LOD	Number of Triangle Strips	Search time (sec)	Geometry Creation (sec)	Rendering Time (sec)
1 (High)	51,660	0.26	10.45	1.48
2	12,406	0.08	2.16	0.58
3	6,044	0.03	0.90	0.25
4 (Low)	3,371	0.02	0.48	0.13

Table 2: LOD Rendering Performance Results using a 54 Million Cell Data Set

LOD	Number of Triangles	Search time (sec)	Geometry Creation (sec)	Rendering Time (sec)
1 (High)	276,180	0.23	3.68	1.44
2	61,796	0.09	0.69	0.27
3	18,748	0.01	0.23	0.07
4 (Low)	8,460	0.01	0.08	0.04
LOD	Number of Triangle Strips	Search time (sec)	Geometry Creation (sec)	Rendering Time (sec)
1 (High)	60,584	0.22	5.81	1.05
2	14,268	0.08	1.03	0.16
3	4,686	0.01	0.32	0.05
4 (Low)	2,243	0.01	0.14	0.02

Table 3: LOD Rendering Performance Results using a 100 Million Cell Data Set

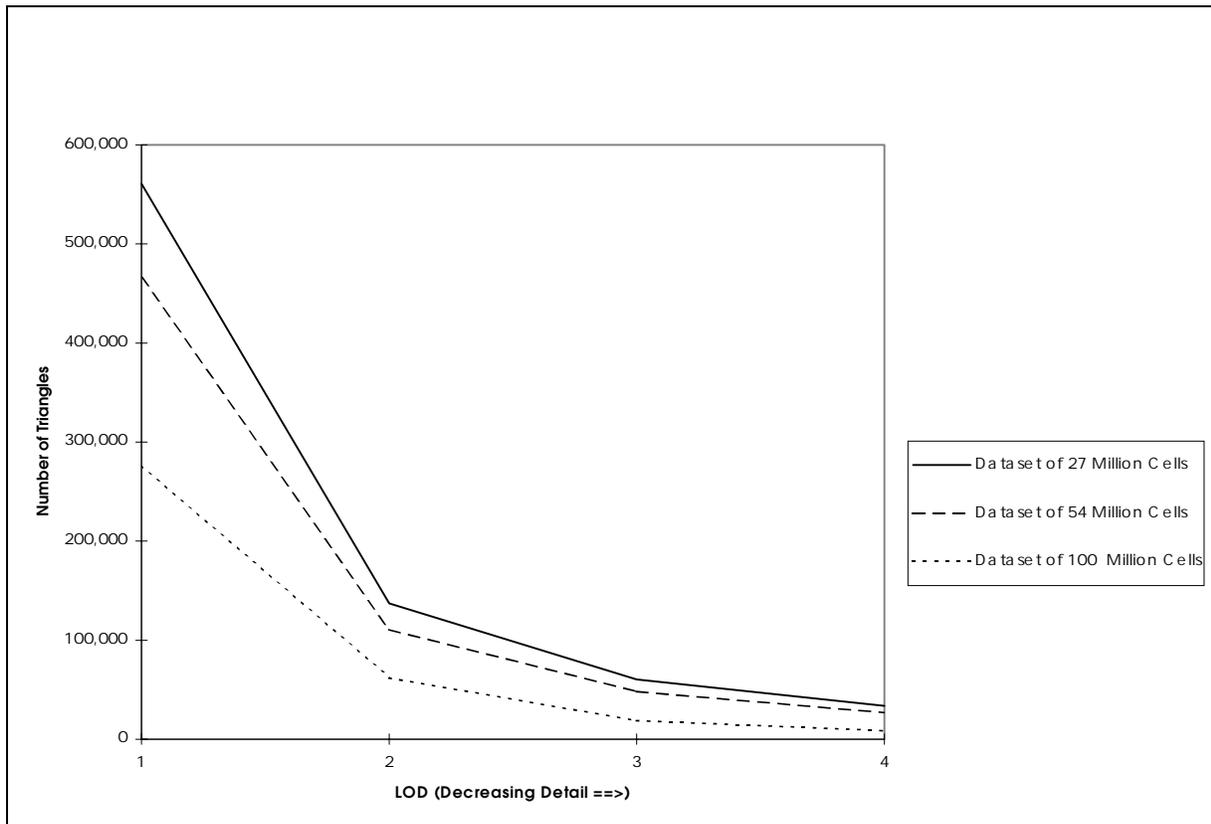


Figure 7: Triangle Reduction at Decreasing Levels of Detail

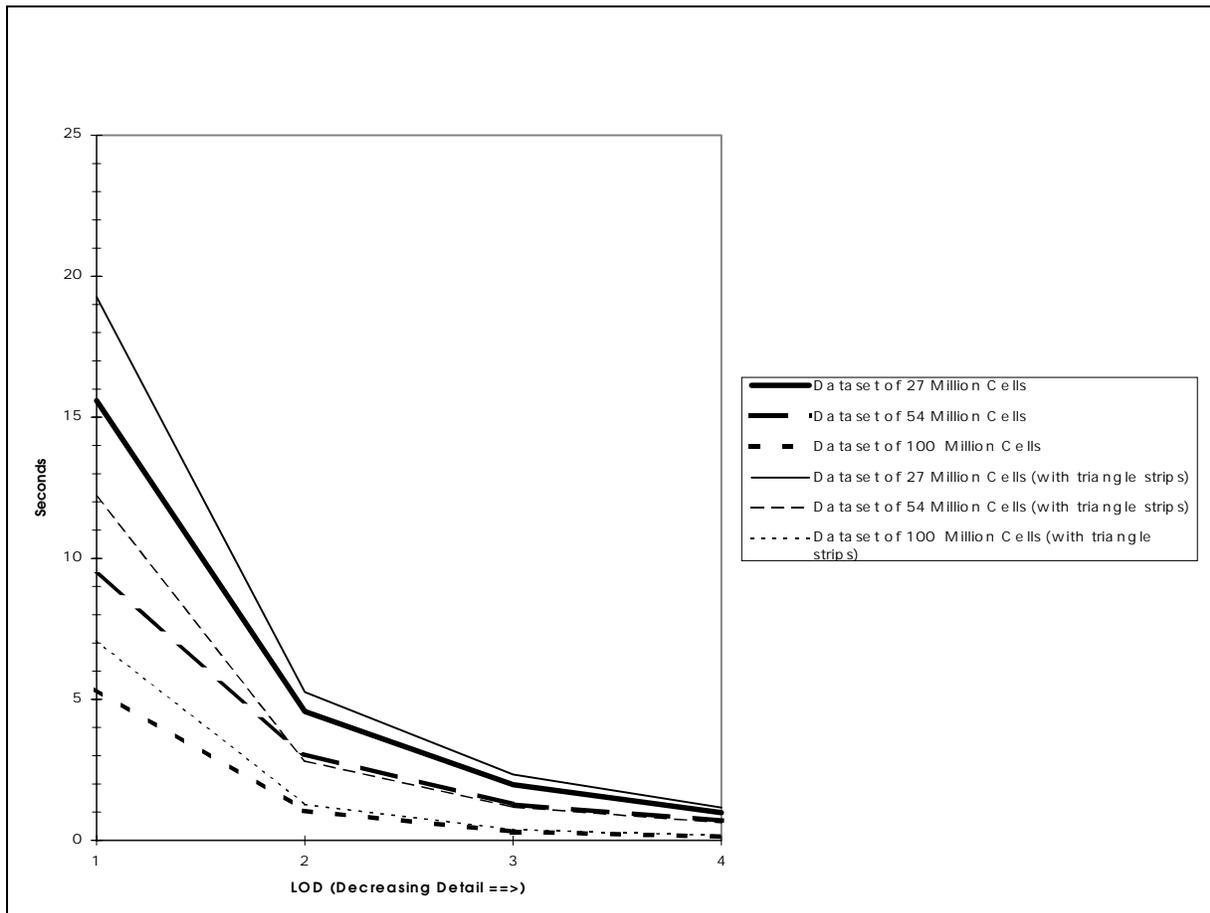


Figure 8: Total Isosurface Generation Time (Including Rendering)

The results clearly show that the search and creation algorithm scaled well to large computations. Like the 2D tree of the NOISE algorithm [6], the LOD tree is searchable in $O(\log N)$ time. This indicates that the total isosurface generation time is primarily dependent on the number of triangles in the isosurface, rather than the number of cells in the data set. Thus, on the smallest data set above, which happened to have a much larger generated isosurface in terms number of triangles, the algorithm took longer to generate the isosurface. This behavior is exactly the reverse of algorithms like marching cubes [1] which scale on the number of cells that they have to search. Also notice that in general, the triangle strip representation of the geometry did *not* decrease the total isosurface generation time.

However, the real success of the LOD algorithm is revealed as we progress to lower levels of detail—we see an exponential decrease in the number of triangles that are generated, and therefore also the total rendering time. Even though the largest isosurface is completely beyond acceptable interactive rendering frame rates at the highest level of detail, it can be displayed at a more acceptable speed (two to three frames a second) using a lower level of detail. Thus we allow the scientist or analyst to examine her or his data at interactive frame rates without losing the capability of looking at an arbitrary isosurface of the original simulation data.

We can conclude from the run-time performance at lower levels of detail (*i.e.*, larger virtual voxels) that despite increasingly larger problem sizes, dynamic isosurfacing and level-of-detail can scale to be interactive and almost immediate. By using a lower level-of-detail for

navigation, the rendering times are reduced by an order of magnitude, while still retaining the original data for exact analysis. This makes data sets that were previously too large for existing rendering hardware to now be readily accessible to the working scientist, which is a critical step in visualizing teraflop-class problems.

DYNAISOLOD also offers another advantage over static geometry methods in that it keeps the original data field in memory for scientists to analyze in ways other than isosurfacing. For example, if a scientist desired to query a data value at a specific point, display a colored cut plane through the data, or use some other dynamic analysis tool, she or he would be able to do so. In a case where only static geometry has been created and the original simulation data has been discarded, these kinds of quantitative capabilities are unavailable—all the scientist can look at is the previously generated isosurfaces.

Note that DYNAISOLOD is not meant to replace traditional marching cube methods, but instead should be used in conjunction with those methods. Statically generated geometry is quite adequate for data such as material volume fractions that do not need dynamic isosurfacing capability. Also, such static techniques have a very low impact on disk storage and main memory requirements. However, for simulation data objects with a much greater range of values, such as pressure fields and shock waves, the rich analysis capability provided by DYNAISOLOD has proven to be very helpful. A slider control for the dynamic isosurface threshold, provided by the visualization tool user interface, can range through any data value present in the simulation data providing a powerful quantitative analysis tool. Likewise, a dynamic level-of-detail slider control can range independently through all the levels-of-detail present. An extremely useful application is to tap the dynamic features of DYNAISOLOD early in the analysis process to discover appropriate isosurface threshold values to use for static geometry creation in subsequent stages of analysis.

8. Future Research Areas

Several future research areas have been identified. An obvious first is to reduce the serial component of the off-line parallel file creation algorithm, and to improve the performance of the triangle strip run-time creation algorithm. An additional goal is to move the file creation algorithm to a distributed memory parallel architecture like Sandia's Intel teraflop supercomputer or a network of personal computers. However, a current weakness in the DYNAISOLOD algorithm is that the first parallel process must be able to fit all of the simulation data in memory at one time in order to create the initial array partitioned around the median voxel. Thus the size of the file being processed is limited by the amount of memory available to the initial processor. For distributed memory machines (like the 128MB-per-node Intel teraflop supercomputer), which have a limited amount of memory available at each processor node, this poses a considerable porting challenge.

Another future research area is to link the DYNAISOLOD rendering engine to a number of different visualization application uses. For example, automatically generating triangles at a lower level-of-detail when the user is navigating through the data set in the visualization environment, sensing when the user stops navigating, and doing progressive refinement from there until the highest resolution in the file is reached, would be an extremely useful capability in exploring large, detailed data spaces.

Because this proof of concept was so successful, a future research area is the extension of this approach to unstructured data volumes. In order to keep the resulting file size as small as

possible, parallel compression and run-time decompression of voxel vertex data may need to be performed, which implies that the file representation of structured and unstructured grid data will probably be different. Another goal is intelligent level-of-detail, in which the voxel size at any given level-of-detail could vary depending on the rate of change of the data values in the cell. The goal is file size reduction, accomplished by allowing arbitrarily large virtual voxels at any given level-of-detail in areas of the grid where the data values were not changing very much, and smaller virtual voxels at the same level-of-detail in areas of the grid where the data was changing rapidly. Such an approach forms a tradeoff between detail and space; the canonical voxel size for that LOD is maintained when detail is necessary (*i.e.*, in voxels where the data values are changing rapidly) and can be expanded in voxels where the data was relatively static. A final future research area is to support tetrahedral data as well as hexahedral data.

9. Acknowledgements

This work was supported by Sandia National Laboratories, a multi-program laboratory operated by Sandia Corporation (a Lockheed Martin company) for the United States Department of Energy under contract DE-AC04-94AL85000. The assistance of other Sandia team members, specifically Tom Anderson and Patricia Crossno, in understanding the problem space and brainstorming possible solutions is also gratefully appreciated.

10. References

- [1] W.E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163-169, July 1987.
- [2] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen, "Decimation of Triangle Meshes," *Computer Graphics*, vol. 26, no. 2, pp. 65-70, July 1992.
- [3] M.H. Gross, R.Gatti, and O. Staadt, "Fast Multi-resolution Surface Meshing", *IEEE Visualization '95 Proceedings*, pp. 135-142, Nov. 1995.
- [4] H. Hoppe, "Progressive Meshes," *ACM SIGGRAPH '96 Proceedings*, pp. 99-108, Aug. 1996.
- [5] J. Popovic and H. Hoppe, "Progressive Simplicial Complexes," *ACM SIGGRAPH '97 Proceedings*, pp. 217-224, Aug. 1997.
- [6] Y. Livnat, H.-W. Shen, and C.R. Johnson, "A Near Optimal Isosurface Extraction Algorithm Using the Span Space," *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, no. 1, pp. 73-84, March 1996.
- [7] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, vol. 8, no. 9, pp. 509-517, September 1975.
- [8] G.M. Nielson and B. Hamann, "The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes," *IEEE Visualization '91 Proceedings*, pp. 83-91, October 1991.
- [9] A. Van Gelder and J. Wilhelms, "Topological Considerations in Isosurface Generation," *ACM Transactions on Graphics*, vol. 13, no. 4, pp. 337-375, October 1994.
- [10] C. Montani, R. Scateni, and R. Scopigno, "A modified look-up table for implicit disambiguation of Marching Cubes," *Visual Computer*, vol. 10, no. 6, pp. 353-355, 1994.
- [11] T. G. Lewis and H. El-Rewini with In-Kyn Kim. *Introduction to Parallel Computing*.

- Englewood Cliffs, New Jersey: Prentice-Hall, 1992.
- [12] R. Sedgewick. *Algorithms in C++*. Reading, Mass.: Addison-Wesley Publishing Co., 1992.
 - [13] Patricia J. Crossno and Edward S. Angel. "Isosurface Extraction from Volume Data using Particle Systems." Sandia National Laboratories Report SAND95-1131C, May 1995.

Distribution:

MS 1140	Jim Rice (6500)
MS 0977	Bill Cook (6524)
MS 1138	Larry Ellis (6531)
MS 1138	Bruce Malm (6532)
MS 1138	Sharon Chapa (6533)
MS 1137	Ken E. Washington (6534)
MS 1137	John M. Linebarger (6534)
MS 0318	George S. Davidson (9215)
MS 0318	Arthurine R. Breckenridge (9215)
MS 0318	Peter B. Lamphere (9215)
MS 9018	Central Technical Files (8940-2) [1]
MS 0899	Technical Library (4414) [5]
MS 0619	Review & Approval Desk (12690) [2] For DOE/OSTI