

Case Study: Visual Debugging Of Finite Element Codes

Patricia Crossno¹

David H. Rogers²

Christopher J. Garasi³

Sandia National Laboratories⁴

Abstract

We present an innovative application developed at Sandia National Laboratories for visual debugging of unstructured finite element physics codes. Our tool automatically locates anomalous regions, such as inverted elements or nodes whose variable values lie outside a prescribed range, then extracts mesh subsets around these features for detailed examination. The subsets are viewed using color coding of variable values superimposed on the mesh structure. This allows the values and their relative spatial locations within the mesh to be correlated at a glance. Both topological irregularities and hot spots within the data stand out visually, allowing the user to explore the exact numeric values of the grid at surrounding points over time. We demonstrate the utility of this approach by debugging a cell inversion in a simulation of an exploding wire.

CR Categories and Subject Descriptors: I.3.8 [Computer Graphics]: Applications; C.4 [Performance of Systems] Modeling techniques – Performance attributes.

Additional Keywords: visual debugging, parallel finite element codes and simulations.

1. INTRODUCTION

We use visualization to help us understand the results of complex, scientific simulations. Transforming large quantities of numbers into three-dimensional shapes and images allows us to use our innate visual processing skills to interpret the data and identify patterns. We use these same skills to assist us in understanding the complexities of software systems, such as massively parallel physics simulations.

Typical scientific visualization algorithms seek to hide the underlying data structures from the viewer. For instance, if the visualization is of an isosurface through a volumetric data set, the goal is to present a smooth surface that does not provide any indication of the underlying grid resolution or processor decomposition. However, while debugging a simulation, if an error in the output is identified while visualizing the results, it is difficult to correlate the error with the underlying data structures on the appropriate processor. In this situation, it would be better to explicitly represent data structures, processor assignment, and other program elements to assist the software developer in finding the error that led to the anomalous results.

Given that software execution is a dynamic process, another component of understanding the complexities of simulation codes is to literally *see* how variables change over time. By abstracting the variable values to color or shape, mapping them into a three-

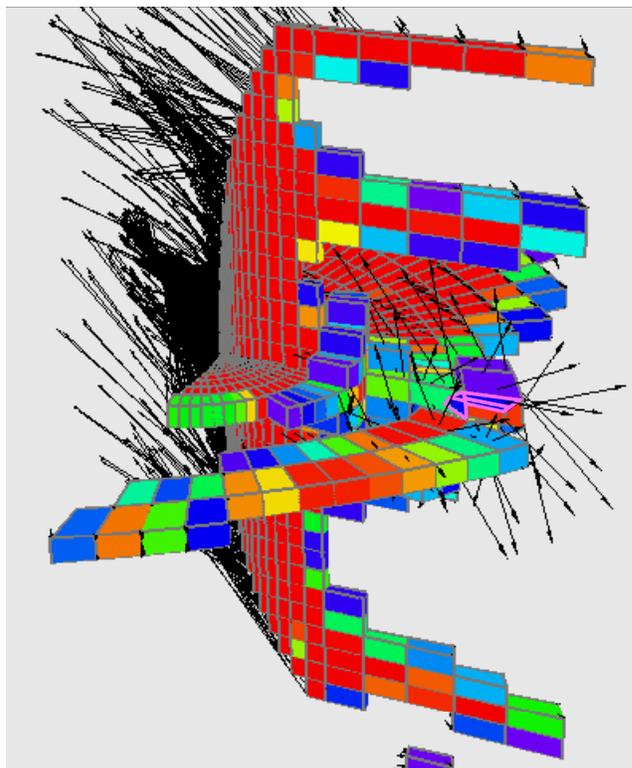


Figure 1: The simulation is of an exploding wire. The volume fraction of the wire material in each element is shown through color, with low amounts as blue and high amounts in red. Elements with levels below a threshold are culled. An inverted element is highlighted in pink on the right. The elements are subsets of extracted planes taken through the inverted element. Black arrows show the derived vectors for magnetic force, $J \times B$.

dimensional domain so that proximity relationships can be correlated with the values, and then showing this over time, the software developer can gain insights into complex, dynamic behaviors within the code. Without this sort of tool, the developer must try to mentally generate a model of variable values in space changing through time. For most people this is very difficult to do, so the dynamics of program behavior must be gleaned through traditional animations of the simulation.

In response to these issues, we have developed a suite of tools for visually debugging parallel physics simulations that make the underlying finite element structures explicit in the visualization. Visual complexity is reduced by using feature detection to find and extract regions of interest from within the mesh prior to detailed viewing. Context within the larger mesh is provided by also extracting external element faces for either the entire volume or blocks of elements within the volume representing significant model components or parts.

¹pjcross@sandia.gov, ²dhroger@sandia.gov, ³cjgaras@sandia.gov

⁴P.O. Box 5800, Albuquerque NM 87185-0822

We have used this tool to help debug a three-dimensional simulation of an exploding wire in which an element inverted unexpectedly. Using our tools, we were able to rapidly locate the inverted element and extract a subset of elements surrounding it. Examining this mesh subset in our viewer as shown in Figure 1, we were able to *see* the topology that lead to the failure.

2. RELATED WORK

This work builds upon our previous work in debugging particle systems [3] and cluster hardware [4][5]. Although we have not found any previous work that uses visualization to debug finite element codes in particular, using visualization to debug or do performance analysis of parallel programs is relevant. The *Rivet* system performs analysis and visualization of parallel applications in a shared memory environment [1]. Browne et al created a directed graph representation for parallel programs to simplify parallel programming [2]. *The Interactive Visualization Debugger* integrates debugging, performance analysis, and data visualization for message passing parallel applications [6]. *Devise* is a generic integrated performance analysis and visualization system that has been coupled with the *Paradyn Parallel Performance Tool* [7]. The *MAD* environment is a tool set for parallel program debugging [8]. Zhang, Hintz, and Ma use graph formalisms and notation to visualize parallel programs and their execution [9].

3. IMPLEMENTATION

Finite element models use elements and groups of elements, called element blocks, as central conceptual components. In the simulation examined in this paper, each element is a hexahedron and is formed by eight nodes designating the corners of the element. Variable values are associated with both elements and nodes and change over time as the simulation progresses. These values are output at regular time intervals and represent changes in the program's state.

Our implementation consists of three parts. First we have a parallel query tool, which divides up the simulation model across multiple processors, scans each of the variables at each time step, and outputs value range information and histograms. Next, we use a parallel feature extraction tool to search the data for either topologically incorrect elements or variable values outside a user-specified range, extracting a neighborhood of elements around any features that are found. Lastly, we have a viewer that draws mesh subsets using color-coding of variables combined with explicit representations of the underlying grids, which allows the user to see anomalies in the data visually and to drill down to get detailed numeric information from specific elements or nodes.

The tool is implemented using a combination of C and C++ using OpenGL as the graphics API. The user interface is implemented using Tcl/Tk. The tool is platform independent and runs on IRIX, LINUX and NT systems.

3.1 Query

The first time a user interacts with a data set, we generate a meta-data file describing the names and types of variables, their minimum and maximum values at each time step, and the distribution of values per variable per time step. This meta-data is then displayed both numerically and visually to provide information to the user in selecting parameters for feature detection and mesh subset extraction.

Typically, data sets are generated in parallel on a cluster. Each processor writes a separate output file that describes a

section of the global grid over a series of time steps in the simulation. The query tool also runs in parallel, though it need not run on the same number of processors as the original simulation. Each processor scans its portion of the data and the results are gathered onto a single processor and written out into an XML file.

3.2 Feature Extraction

Mesh data can be searched for both inverted elements and variable values that fall outside a user-specified range. Inverted elements are detected by searching for elements in the final time step that have a negative Jacobian. Value scanning can be done on element and node variables over a user-specified range of time steps. The variables are selected and their limits are set using a graphical user interface (GUI). The extraction tool then searches for values outside that range.

Once features are found within the mesh, the extraction tool marks those elements for output. If the feature is centered on a node, all of the elements using that node are marked. A neighborhood distance parameter defines the size of the feature-centered sub-mesh to output. This distance is a topological radius that can be thought of as the number of layers of elements around the feature, where the elements are maximally connected using face, edge, and node connectivity. A small neighborhood of elements is shown in Figure 2. Multiple features can be extracted at once, even if they form disconnected components.

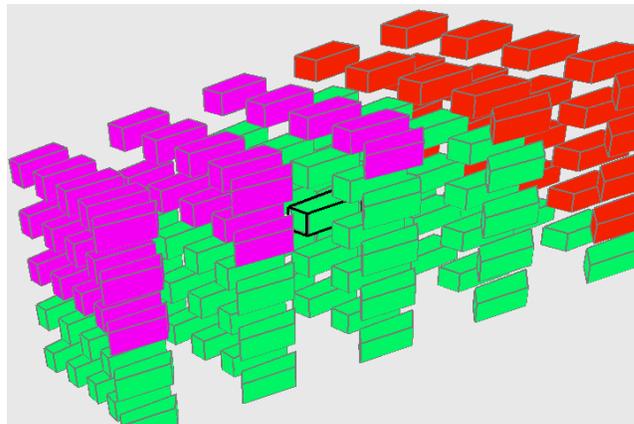


Figure 2: The inverted element, which is highlighted in black, is surrounded by a neighborhood of elements. Elements are color-coded by processor id.

3.3 Mesh Subset Viewer

The mesh subset viewer provides a unique paradigm for debugging and is the core of the project. To make the images easier to understand, the viewer is entirely application specific and only deals with finite element meshes. The challenge in the viewer is to deal with the problem of scale in dealing with very large meshes, while still providing both context and focus.

3.3.1 Context

As the number of elements within a simulation increases, visualizing all of the elements fails when the number of elements exceeds the number of pixels in the display. In fact, even moderate-sized meshes present a viewing problem because the meshes are three-dimensional and interior elements are obscured. Subset meshes require context with respect to the original mesh in order to be useful. The viewer provides context in three ways.

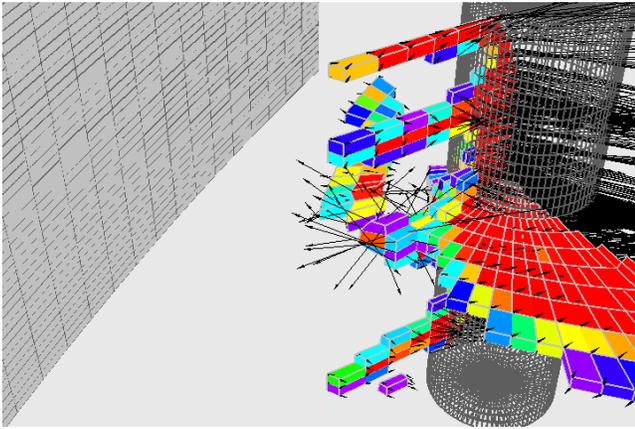


Figure 3: External faces showing selected element blocks provide context for the mesh subset shown in Figure 1. The element block on the left is drawn as a solid with the element boundaries outlined. The cylindrical element block representing the wire is drawn in wire-frame mode to permit elements interior to the wire to be visible. The element block encompassing the inverted element and the elements in the material interface has been turned off.

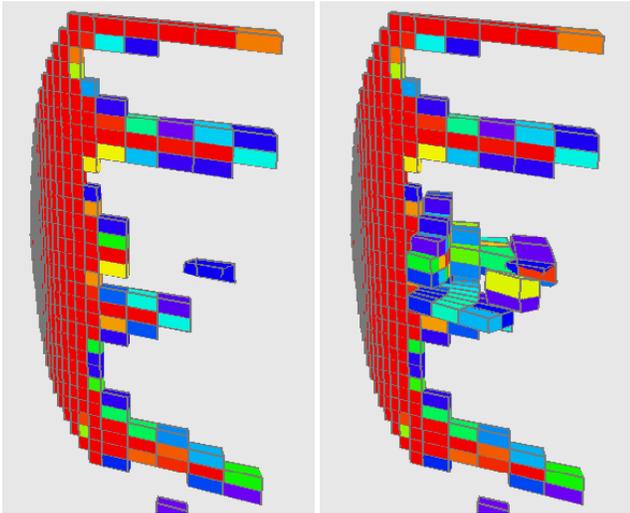


Figure 4: The material interface between the wire and the vacuum is shown in these two images. The elements are colored by the amount of the wire material in each element. As the wire is exploding, fingers of the material are coming off the wire, which lies to the left as the solid red region. Although in the image on the left the inverted element appears isolated from the material, by adding additional elements from the neighborhood around the inverted element, we can see it is tenuously connected.

First, the user needs to know where the subset lies relative to the boundaries and components of the original model. During the extraction process, the user can choose to output external element faces from significant structures within the model, which are then loaded into the viewer along with the mesh subset. There are three options for defining the external faces: the boundary elements for the entire volume, the boundary elements for each piece of the model or element block, or a user specified list of element block groups. In the viewer, these faces can be viewed as wire frames, solid surfaces, and groups of faces can be turned on and off. These viewing options are shown in Figure 3.

The second type of context is to know where the feature element lies relative to various material boundaries that change dynamically during the simulation. Typically, the neighborhood of elements extracted surrounding the feature element is not large enough to see the salient aspects of the material interface, so we also allow the user to optionally extract topologically-orthogonal “planes” of elements centered on a feature element. An element plane is found in a manner similar to that used in extracting a neighborhood of elements, but instead of stepping out in all directions using maximum connectivity, a subset of faces is followed in an ever-expanding ring from the center element. This is repeated for each orthogonal direction. As elements are extracted as either features or planes, they are tagged with a USAGE variable so that the viewer can quickly show or hide these groups. Examples of topological planes displaying material interfaces are given in both Figure 1 and Figure 4.

Time is the third contextual frame of reference. Not only do the contents of variables change over time, but the topology of the mesh can also warp as node positions shift. Additionally, animation provides insights into dynamic behaviors, such as watching the progression of the material interface or the propagation of temperature over time. We provide a VCR-like interface that allows both animation and single stepping. Random access to individual time steps is also available.

3.3.2 Focus

The two central issues in the visualization of the focus region are how to maximize information content, while simultaneously minimizing visual clutter. We use color, shape, mesh topology, and spatial juxtaposition to represent model information. To reduce the number of objects in the display to just those which address a specific question, we provide a variety of filtering mechanisms that give the user fine control of the features desired in the culled subset of focus elements.

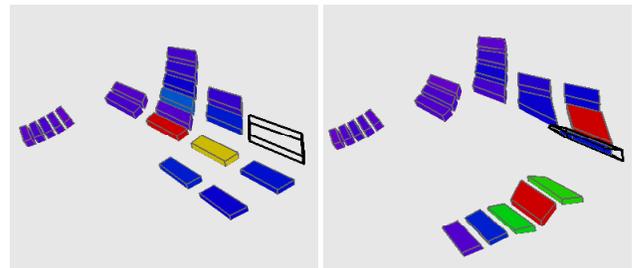


Figure 5: The image on the left shows time dump 16. On the right is time dump 17. The inverted element is highlighted in black. Elements are colored by temperature and culled to show only higher temperatures.

We use a spectrum scale for our color-encoding scheme because it is familiar to our users and they requested it. In this scale, the color order from high values to low values is red, orange, yellow, green, cyan, blue, indigo. Violet is the bottom of the scale and elements with this color are generally culled.

We render elements as colored boxes with their edges highlighted in black to reinforce the three-dimensionality of the model. We can either draw the elements at full size as shown in Figure 4, or we can draw the elements at a reduced scale as shown in Figure 5 and Figure 2. One advantage of reducing the scale is that it enables the user to see interior elements, or the sides of elements lying in a group. In Figure 5, the color-coding combined with the topological information tells us that the inverted element

had a lower temperature while the adjacent red element had an unusually high temperature compared to nearby elements.

Although the nodes are implicitly rendered as the corners of the elements, the nodes can be explicitly represented in one of two ways. To view scalar-valued nodal variables, or a single component of a vector variable, the user can enable small colored spheres to be drawn at element corners. For vector variables, scalable arrow glyphs can be either combined with the spheres or drawn independently as shown in Figure 1. By rendering a combination of colored element boxes, colored nodal spheres, and nodal vector arrows, three different variable values can be viewed simultaneously. These variables can then be viewed changing over time through the animation capability. In addition, derived quantities can also be generated and viewed, such as the vector cross product shown in Figure 1.

Elements can be selected using picking. Numeric values for the picked element's variables appear in a display window. These values are updated with each time increment during animation or when single stepping. The highlight designating the picked element provides feedback and element tracking as the element color or shape changes through time.

We reduce visual clutter by providing many different ways to cull sections of the model. The coarsest level of culling is done during the extraction step with the selection of a neighborhood of elements and topological planes around a feature. Once in the viewer, the neighborhood and each of the planes is treated as a separate object whose display is individually controllable. Elements within each of these objects are also automatically partitioned into processor groups based on the element's processor id during the simulation. These groups can also be individually controlled, so a display can be constructed from a combination of processor groups and extraction groups.

4. RESULTS

We used our tools to debug a simulation of an exploding wire in which an element inverted after the 17th time dump. The simulation model was of moderate size and consisted of 241,600 elements distributed over 40 processors. Prior to our tool's development, finding the inverted element would have required manually searching for the inverted element using a conventional scientific visualization package to whittle away elements from the model with slicing planes until the inverted element could be seen. Our extraction tool automatically located the inverted element and extracted a neighborhood of elements around it, along with the topological planes running through the element.

Using the viewer, we were able to determine that the inverted element existed at the outer radial edge of the exploded material as seen in Figure 4. These boundary elements are mixtures of material and vacuum. By culling the boundary elements based on the percentage of material, we were able to answer the question: is the inverted element an isolated fragment, tenuously attached, or a portion of the main body? This illustrates a potential algorithmic issue with the computation of magnetic forces in a mixed element. This hypothesis was verified using the derived magnetic force vector, $\mathbf{J} \times \mathbf{B}$, which in Figure 1 shows anomalous values at the inverted element. Additionally, the time history of neighboring elements indicates a precursor phenomenon involving temperature, as shown in the time dumps in Figure 5.

5. CONCLUSIONS AND FUTURE WORK

Automatic feature location and extraction combined with visualization of color-coded variables mapped into the three-

dimensional topology and components of the finite element mesh provide a powerful tool for visually debugging simulation codes. By showing the user multiple variable values in their spatial context, how these values change over time, and how a combination of element and node variables interact, our tools allow the user to examine the simulation at a higher level of abstraction than a conventional textual debugger can provide.

In the future, we intend to generalize our derived variable capability to allow users to enter expressions using variables generated by the simulation to produce arbitrary derived element or node variables for display in the viewer. Additionally, we want to tie histograms and statistical information in the query tool back into the viewer, so that the user can filter the display of elements by selecting sections of the histogram.

6. ACKNOWLEDGMENTS

We thank Daniel Carroll for providing the element inversion code and for test data sets. The DOE Mathematics, Information, and Computer Science Office funded part of this research. The work was performed at Sandia National Laboratories. Sandia is a multi-program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

References

- [1] Robert Bosch, Chris Stolte, Gordon Stoll, Mendel Rosenblum, and Pat Hanrahan. Performance Analysis and Visualization of Parallel Systems Using SimOS and Rivet: A Case Study. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 360-371, January 2000.
- [2] James C. Browne, Syed I. Hyder, Jack Dongarra, Keith Moore, and Peter Newton. Visual Programming and Debugging for Parallel Computing. *IEEE Parallel & Distributed Technology: Systems & Applications*, Vol. 3, Issue 1, pages 75-83, 1995.
- [3] Patricia Crossno and Edward Angel. Visual Debugging of Visualization Software: A Case Study for Particle Systems. In *Proceedings of Visualization '99*, pages 417-420, October 1999.
- [4] Patricia Crossno and Rena Haynes. Case Study: Visual Debugging of Cluster Hardware. In *Proceedings of Visualization 2001*, pages 429-432, October 2001.
- [5] Rena Haynes, Patricia Crossno, and Eric Russell. A Visualization Tool for Analyzing Cluster Performance Data. In *Proceedings IEEE International Conference on Cluster Computing 2001*, pages 295-302, October 2001.
- [6] Ming C. Hao, Alan H. Karp, Milon Mackey, Vineet Singh, and Jane Chien. On-the-Fly Visualization and Debugging of Parallel Programs. In *Proceedings International Workshop on Modeling, Analysis, and Simulation of Computer Telecommunication Systems*, pages 386-391, 1994.
- [7] Karen L. Karavanic, Jussi Myllymaki, Miron Livny, and Barton P. Miller. Integrated Visualization of Parallel Performance Data. *Parallel Computing*, Vol. 23, pages 181-198, 1997.
- [8] D. Kranzlmüller, S. Grabner, and J. Volkert. Debugging with the MAD Environment. *Journal of Parallel Computing*, Vol. 23, No. 1-2, pages 199-217, April 1997.
- [9] Kang Zhang, Tom Hintz, and Xianwu Ma. The Role of Graphics in Parallel Program Development. *Journal of Visual Languages and Computing*, Vol. 10, No. 3, pages 215-243, June 1999.