The SDAV Software Frameworks for Visualization and Analysis on Next-Generation Multi-Core and Many-Core Architectures

Christopher Sewell*, Jeremy Meredith†, Kenneth Moreland‡, Tom Peterka§, Dave DeMarle¶,
Li-ta Lo*, James Ahrens*, Robert Maynard¶, Berk Geveci¶
*Los Alamos National Laboratory; Los Alamos, NM, USA; Email: csewell@lanl.gov
†Oak Ridge National Laboratory; Oak Ridge, TN, USA; Email: jsmeredith@ornl.gov
‡Sandia National Laboratory; Albuquerque, NM, USA; Email: kmorel@sandia.gov
§Argonne National Laboratory; Argonne, IL, USA; Email: tpeterka@mcs.anl.gov
¶Kitware; Clifton Park, NY, USA; Email: berk.geveci@kitware.com

***Abstract*—This paper surveys the four software frameworks being developed as part of the visualization pillar of the SDAV (Scalable Data Management, Analysis, and Visualization) Institute, one of the SciDAC (Scientific Discovery through Advanced Computing) Institutes established by the ASCR (Advanced Scientific Computing Research) Program of the U.S. Department of Energy. These frameworks include EAVL (Extreme-scale Analysis and Visualization Library), Dax (Data Analysis at Extreme), DIY (Do It Yourself), and PISTON. The objective of these frameworks is to facilitate the adaptation of visualization and analysis algorithms to take advantage of the available parallelism in emerging multi-core and many-core hardware architectures, in anticipation of the need for such algorithms to be run *in-situ* with LCF (leadership-class facilities) simulation codes on supercomputers.**

***Keywords*-SDAV; data-parallel; in-situ; visualization; mult-core and many-core architectures; VTK-m**

I. Introduction

Due to fundamental limitations of physics, hardware manufacturers are no longer able to significantly increase processor clock rates, and thus are instead increasing the computational capacity of their machines by increasing the number of cores that can operate in parallel. Furthermore, due to a number of constraints, most importantly power consumption, the storage capacity and speeds of supercomputers is not keeping up with the growth in computational speeds (flops). Therefore, in order to keep up with larger simulation runs enabled by the faster computers, visualization and analysis operations, which have traditionally been performed serially as a post-processing step, will need to be performed in-situ (or in-transit) with the simulation codes, running at the same time as the simulation (either in the same memory space or on separate processors) and making efficient use of the massive parallelism on these new architectures.

In anticipation of these in-situ use cases with LCF simulation codes, the SDAV Institute is developing software frameworks that will facilitate the adaptation of visualization and analysis algorithms to take advantage of the available parallelism in emerging mult-core and many-core hardware architectures. These frameworks will make it easier for domain scientists to take advantage of the parallelism available on a wide range of current and next-generation hardware architectures, and to incorporate them in-situ with their simulations. Collectively, these frameworks are known as the VTK-m Frameworks, as they attempt to enhance the traditional visualization and analysis infrastructure embodied by the popular VTK toolkit to better support multi and many-core architectures. In many cases, work has already begun to integrate these frameworks into VTK, in collaboration with Kitware, the developers of VTK.

II. The Frameworks

While all of the SDAV software frameworks have similar goals and have drawn on ideas from each other, they are currently pursued as independent research projects with different emphases. Broadly speaking, EAVL emphasizes the development of a new data model, Dax emphasizes the development of a new execution model, DIY provides a lightweight toolkit of commonly-used parallel functionality, and PISTON emphasizes portability and parallel algorithm development.

A. EAVL

EAVL [MAP*12] has three primary objectives: update the traditional data model to handle modern simulation codes and a wider range of data; investigate how an updated data and execution model can achieve the necessary computational, I/O, and memory efficiency; and explore methods for visualization algorithm developers to achieve these efficiency gains and better support next-generation architectures.

EAVL defines more flexible mesh structures which more efficiently support many non-traditional types of data, reducing memory usage and/or reducing the overhead of unnecessary copying of data. Examples of data that are inefficiently represented by traditional data models include non-physical data such as graphs, in which storing spatial coordinates for nodes is unnecessary; mixed data types, such

as molecular data in which atoms are represented by vertices and bonds by lines but atom-specific fields (e.g., atomic numbers) and bond-specific fields (e.g., bond strength) must be stored across both element types; very high order fields; and unique mesh topologies, such as unstructured adaptive mesh refinement or quad-trees.

Figure 1 shows a traditional data model along side the EAVL data model. EAVL allows a much more flexible compositing of the different primitive objects. It can easily support, for example, a set of coordinates which refer to either different components of the same field, or different fields. There can be multiple cell sets (for example, one for atoms and one for bonds), and fields may be associated with all or with only a subset of the cell sets (e.g., an atomic number field with only the atom cell set and a bond strength field with only the bond cell set).

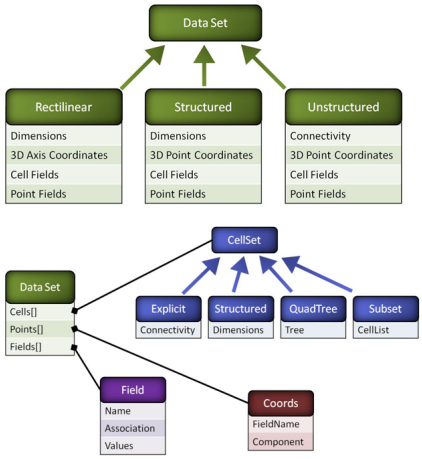

Figure 1. The traditional visualization data model (top) and the EAVL data model (bottom)

An example of a situation in which the EAVL data model can eliminate the need for unnecessary data copies is shown in Figure 2. A 2D structured grid, with its coordinates stored as alternating x and y values in a single field, is to be elevated, resulting in a 3D grid, using the computed z values in another field. Rather than having to allocate new memory for a 3D coordinate array and copying all the data to fit an alternating x, y, z format, a single pointer can be added to the coordinate object to point to the new field. Now, the first two components of the coordinates (x and y) refer to the zero and one components of the first field, and the third component of the coordinates (z) refers to the single component of the new field, with no data copying required.

Operators in EAVL are classified as either mutators, which modify existing data in-place, or filters, which create a new data set from an old one. Similar to other frameworks which will be described in more detail below, EAVL makes use of the functor concept to allow the user to write functions that can be executed on either the CPU or GPU. The EAVLLab

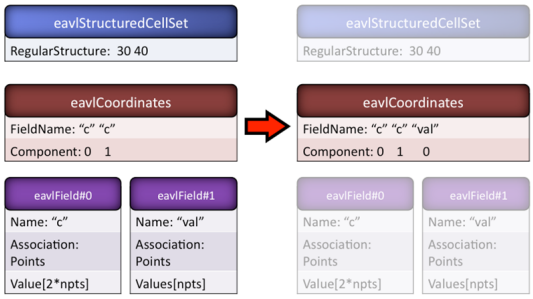

Figure 2. An example of elevating a structured grid using the EAVL data model

application provides a graphical user interface in which a user can load data sets, create new pipelines of EAVL mutators and filters, and visualize the results (Figure 6).

B. *Dax*

The Dax Toolkit supports the fine-grained concurrency for data analysis and visualization algorithms required to drive exascale computing. The basic computational unit of the Dax Toolkit is a worklet, a function that implements the algorithms behavior on an element of a mesh (that is, a point, edge, face, or cell) or a small local neighborhood [MAG*11]. The worklet is constrained to be serial and stateless; it can access only the element passed to and from the invocation. With this constraint, the serial worklet function can be concurrently scheduled on an unlimited number of threads without the complications of memory clashes or other race conditions.

The Dax Toolkit provides schedulers that apply worklets to all elements in an input mesh, the results of which are collected into a resulting mesh. Although worklets are not allowed communication, many visualization algorithms require operations such as variable array packing and coincident topology resolution that intrinsically require significant coordination among threads. Dax enables such algorithms by classifying and implementing the most common and versatile communicative operations, which, when used in conjunction with the appropriate worklets, complete the visualization algorithms.

The Dax Toolkit simplifies the development of parallel visualization algorithms. For example, Dax provides a generic mechanism to create a new topology based on input cells. The code in Listing 2 demonstrates all the code needed to build a new mesh by removing cells from an existing mesh. The ability to remove unneeded components and updating

```
1  struct PolyNormalFunctor
2  {
3          void operator()(float *x, float *y, float *z, float *n)
4          {
5          // get two adjacent edge vectors
6          float ax = x[1]-x[0],  ay = y[1]-y[0],  az = z[1]-z[0];
7          float bx = x[2]-x[1],  ay = y[2]-y[1],  az = z[2]-z[1];
8          // calculate their cross product
9          n[0] = ay*bz - az*by;  n[1] = az*bx - ax*bz;   n[2] = ax*by - ay*bx;
10         }
11 };
12
13 void FaceNormalFilter::Execute(...)
14 {
15         executor->AddOperation(new NodeToCellOp3(xcoord, ycoord, zcoord,
16                     outputnormals, inputcells, PolyNormalFunctor()));
17 }
```

Listing 1. Sample EAVL code for computing normals

fields is built into the system. An example of this code applied to a supernova dataset is shown in Figure 6.

As shown in Figure 3, Dax consists of a control environment, a serial environment (per process) used to set up data structures and dispatch parallel jobs, and an execution environment, the parallel environment in which data processing occurs. These two environments correspond with the typical configurations of commonly used accelerator programming models such as GPGPU and Intel Xeon Phi offloading, but can also work when both environments exist on the same hardware. Between these two environments, a device adapter handles data transfers and scheduling, allowing the execution environment to work on a variety of architectures.

The Dax control environment API provides basic topology structures used to represent meshes that are input and returned from the system. The control environment also contains generic array containers that allow Dax algorithms to adapt to memory structures of external applications [MKM*12]. The Dax execution environment API provides many utilities to facilitate writing functors, including basic types, vector operations, and small matrix operations. In addition, worklets can receive a cell as a parameter. Supported cell types include vertices, lines, triangles, quadrilaterals, voxels, tetrahedrons, wedges, and hexahedrons. Each cell type provides functions that implement commonly used operations that involve the interplay of constituent components such as interpolations and derivatives.

C. DIY

DIY [PET12] provides a lightweight, easily-maintainable library of functions commonly used in parallel visualization and analysis operators, including in-situ, coprocessing, and traditional post-processing use cases. It allows researchers

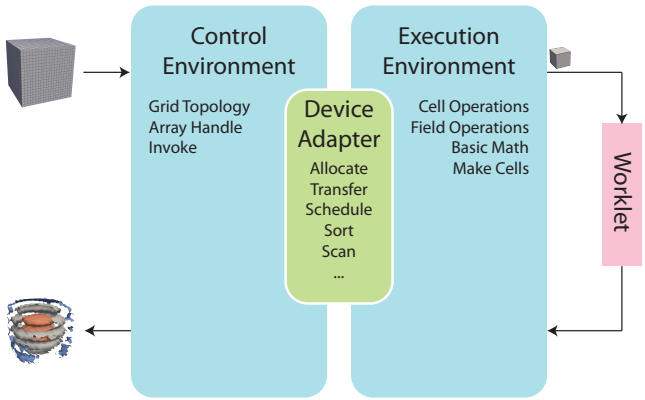

Figure 3. Diagram of the Dax control and execution environments

to focus on their own work rather than on assembling the parallel infrastructure from scratch. As shown in Figure 4, it consists of parallel I/O, domain decomposition, network communication, and utility functions. It supports global and neighborhood communication patterns, including nearest neighbor, merge-based reductions, and swap-based reductions. It has been used with a wide variety of large-scale simulations, including Morse-Smale complex of combustion, information entropy analysis of astrophysics, Voronoi tesselation of cosmology data, and particle tracing of thermal hydraulics flow (Figure 6).

D. PISTON

The primary goal of PISTON [LSA12] is to facilitate the development of visualization and analysis operators with highly portable performance. Due to the wide variety of current and next-generation parallel hardware architectures,

```
1
2  struct ThresholdTopology : dax::exec::WorkletGenerateTopology
3  {
4    typedef void ControlSignature(Topology, Topology(Out));
5    typedef void ExecutionSignature(Vertices(_1),Vertices(_2));
6
7    template<class InCellVerticesType, class OutCellVerticesType>
8    DAX_EXEC_EXPORT
9    void operator()(const InCellVerticesType &inVertices,
10                   OutCellVerticesType &outVertices) const
11   {
12     outVertices.SetFromTuple(inVertices.GetAsTuple());
13   }
14 };
15
16 template<class InGridType, class ClassifyArrayType,
17          class OutGridType, class ScalarArrayType>
18 DAX_CONT_EXPORT
19 void InvokeThreshold(const InGridType &inGrid,
20                      const ClassifyArrayType &classifyArray,
21                      OutGridType &outGrid,
22                      const ScalarArrayType &inScalars,
23                      ScalarArrayType &outScalars)
24 {
25   using namespace dax::cont;
26
27   // Create new geometry.
28   ScheduleGenerateTopology<ThresholdTopology> resolveTopology(classifyArray);
29   Scheduler<> scheduler;
30   scheduler.Invoke(resolveTopology, inGrid, outGrid);
31
32   // Convert input point scalar array to new grid.
33   resolveTopology.CompactPointField(inScalars, outScalars);
34 }
```

Listing 2. Sample Dax worklet and control code to threshold cells based on a given classification. This includes reconfiguring the connectivity to remove unused points and mapping a scalar field to the new topology.

developers are frequently forced to re-tune or even re-write their operators in order to take advantage of the available parallelism on new multi-core or many-core platforms. Some standards, such as OpenCL, may allow the same code to run on multiple supported platforms, but to actually run efficiently still requires architecture-specific optimizations in algorithm design and tuning. However, by restricting the programmer to using only a limited set of data-parallel primitives, portable performance can be obtained. Architecture-specific optimizations are thus only required for the small set of data-parallel primitives. Algorithms may still require clever design to run efficiently, but these efficiencies should be realized across all supported platforms.

Data parallelism [BLE90] is a programming model in which independent processors simultaneously perform the same task on different pieces of data. Data-parallel primitives operate on vectors, and may be customized through user-defined functors. For example, the transform operator simply applies a unary functor to each element of the input vector, or a binary functor to each pair of elements in two input vectors. The scan operator with a binary addition functor produces a vector of partial sums from the input vector. The reduction operator with a binary maximum functor returns the maximum element of the input vector. As the data sizes produced by simulations continue to increase, data parallelism is likely to be an effective approach to exploit available parallelism.

PISTON is built on top of NVIDIA's Thrust library [THR12]. Thrust is a C++ template library that provides implementations of data-parallel primitives in CUDA,

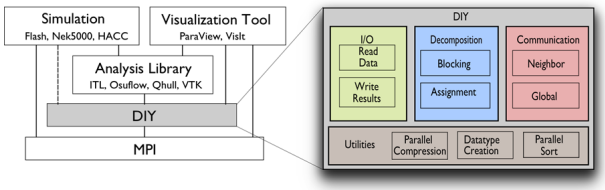

Figure 4. Diagram of the DIY library

OpenMP, and TBB. It is designed to be extensible to support additional backends to target other architectures. PISTON has enhanced the Thrust OpenMP backend, developed a new prototype backend for OpenCL, and, most significantly, designed and implemented data parallel algorithms for a suite of visualization and analysis operators using these data-parallel primitives.

One such algorithm developed in PISTON is the iso-surface operator for structured grids using marching cubes [LC87], [DZ07]. A naive data-parallel algorithm for this operator could be constructed by simply classifying each input cell as to whether it generates geometry using the transform primitive with a functor that examines the values at each vertex, stream compacting those cells that do generate geometry ("valid" cells), generating an equal number of output vertices for each valid cell (some of which are "phantom" vertices, generated in order to avoid branching in the kernels and to ensure each parallel thread outputs beginning at a predetermined offset in the output vector), and doing a final stream compaction to eliminate phantom geometry. However, this algorithm is inefficient due to the large amount of global memory movement and unnecessary generation and removal of phantom geometry. As illustrated in Figure 5, our optimization of this algorithm improves efficiency by generating a "reverse mapping" from output vertex index to input cell index (using transform, scan, and binary search data-parallel primitives), allowing it to "lazily" apply operations only to cells that will generate output. It allows the use of a gather operation rather than a stream compaction, and computes the correct offset into the final global output vertex vector, eliminating the need for phantom geometry and the extra stream compaction. The exact same operator code was compiled with the CUDA backend and run on an NVIDIA GPU, and compiled with the OpenMP backend and run on a multi-core Intel Xeon CPU, obtaining good parallel scaling on both. Using the same basic algorithmic design, cut surface and threshold operators have also been implemented.

PISTON has been integrated into VTK and into a ParaView plug-in. VTK filters have been implemented that encapsulate PISTON algorithms, and utility filters interconvert between the standard VTK data format (stored on the CPU) and the PISTON data format (Thrust device vectors, which may exist on the GPU). Results of PISTON algorithms may either be rendered directly on the GPU using the OpenGL/CUDA interop feature, or passed back to standard VTK renderers on the CPU using the PISTON utility filters. The VTK domain partitioning and image compositing infrastructure can be used to run in a distributed memory environment (with PISTON operators being applied locally on each node to its domain).

Ongoing research has been focused on expanding the domain of PISTON both in terms of application areas and hardware. The PISTON data model has been extended to support curvilinear coordinates. An extension of the marching cubes algorithm for structured grid isosurfaces supports the marching tetrahedra algorithm for an unstructured grid that has been decomposed into tetrahedra. Prototype rasterizing and ray-casting render operators allow the generation of images on systems without OpenGL. A wrapper is being developed for some Thrust operators to allow them to be called in a distributed memory environment. PISTON is being utilized in-situ with the VPIC (Vector Particle in Cell) simulation code, and a companion project, PINION, is developing portable data-parallel operators for physics computations (such as gradients, advection, and Lagrangians) in materials and hydrodynamics simulations. Various applications of PISTON are pictured in Figure 7. More details about these ongoing projects should appear in future publications.

III. Conclusion

While each of these frameworks has its own specific areas of focus, taken together, they are attempting to solve some of the fundamental challenges facing visualization and analysis in the era of massive parallelism and big data. The results of these research efforts, along with their depolyment in well-supported, commonly used open-source libraries such as VTK and ParaView, should better position the scientific community to be able to make use of the data produced by ever-larger-scale simulations in this new era.

Acknowledgment

The SciDAC Institute of Scalable Data Management, Analysis, and Visualization (SDAV) is funded by the DOE Office of Science through the Office of Advanced Scientific Computing Research (Contract No. 12-015215). Arie Shoshani is the Institute Director, and James Ahrens and Wes Bethel are the Visualization Project Chairs. Related

```
namespace piston {

template <typename VertexType>
class compute_normal
{
public:
  typedef typename thrust::counting_iterator<int> CountingIterator;
  typedef typename thrust::device_vector<float3> NormalsType;
  typedef typename NormalsType::iterator NormalsIterator;

  int nVerts;
  VertexType vertices;
  NormalsType normals;

  compute_normal(VertexType vertices, int nVerts) :
                vertices(vertices), nVerts(nVerts)  {};

  void operator()()
  {
    normals.resize(nVerts/3);
    thrust::transform(CountingIterator(0), CountingIterator(0)+nVerts/3,
                      normals.begin(), normal(vertices));
  }

  struct normal : public thrust::unary_function<int, float3>
  {
    VertexType vertices;

    normal(VertexType vertices) : vertices(vertices) {};

    __host__ __device__
    float3 operator() (int id) const
    {
      float3 v0 = vertices[id*3+0];
      float3 v1 = vertices[id*3+1];
      float3 v2 = vertices[id*3+2];

      float3 e0 = v1 - v0;
      float3 e1 = v2 - v0;

      return cross(e0, e1);
    }
  };

  NormalsIterator normals_begin() { return normals.begin(); }
  NormalsIterator normals_end()   { return normals.end();   }
};
}

compute_normal<thrust::device_vector<float3>::iterator>* cn = new
  compute_normal<thrust::device_vector<float3>::iterator>(v.begin(),v.size());
(*cn)();
normals.assign(cn->normals_begin(), cn->normals_end());
```

Listing 3. Sample PISTON code for an operator to compute normals

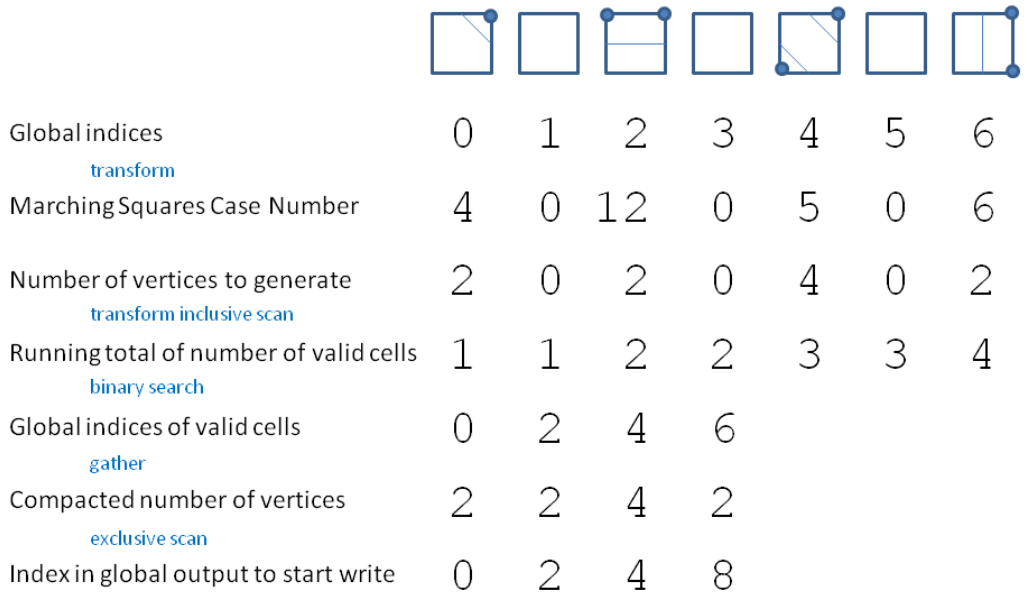

Figure 5. Illustration of the PISTON data-parallel isosurface algorithm

work has been funded by the Advanced Simulation and Computing Program and by Laboratory Directed Research and Development awards. Dax work was supported in whole by the DOE Office of Science, Advanced Scientific Computing Research, under award number 10-014707, program manager Lucy Nowell, and Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND 2012-10713C

REFERENCES

[MAP*12] J.S. Meredith, S. Ahern, D. Pugmire, R. Sisneros. "EAVL: The Extreme-scale Analysis and Visualization Library". *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, May 2012.

[MAG*11] Kenneth Moreland, Utkarsh Ayachit, Berk Geveci, and Kwan-Liu Ma. "Dax Toolkit: A Proposed Framework for Data Analysis and Visualization at Extreme Scale". *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, October 2011, pp. 97-104. DOI 10.1109/LDAV.2011.6092323.

[MKM*12] Kenneth Moreland, Brad King, Robert Maynard, and Kwan-Liu Ma. "Flexible Analysis Software for Emerging Architectures". *Petascale Data Analytics: Challenges and Opportunities* (PDAC-12), November 2012.

[PET12] Tom Peterka. "Do-It-Yourself Data Analysis: Selected Topics and Recent Adventures". Seminar at Los Alamos National Laboratory, February 28, 2012.

[LSA12] Li-ta Lo, Christopher Sewell, and James Ahrens. "PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators". *Eurographics Symposium on Parallel Graphics and Visualization*, May 2012.

[BLE90] Guy Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press. ISBN 0-262-02313-X. 1990.

[THR12] Thrust library. Project webpage: http://code.google.com/p/thrust/. Accessed 2012.

[LC87] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High-Resolution 3D Surface Construction Algorithm". *Computer Graphics*, Vol. 21, Num. 4, July 1987.

[DZ07] Christopher Dyken and Gernot Ziegler. "High-speed Marching Cubes using Histogram Pyramids". *Eurographics*, Vol. 26 Num. 3. 2007.

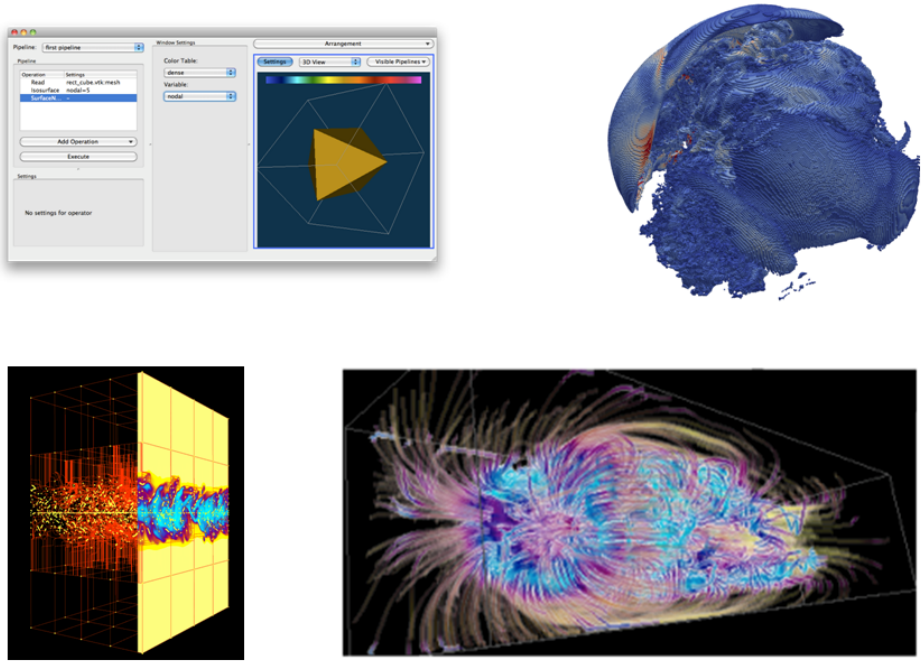

Figure 6. Clockwise, from top left: Isosurface generated in EAVL Lab; contour with subsequent vertex welding, coarsening, and subdivision produced using Dax; information entropy analysis of astrophysics data computed using DIY; Morse-Smale complex of combusion data computed using DIY

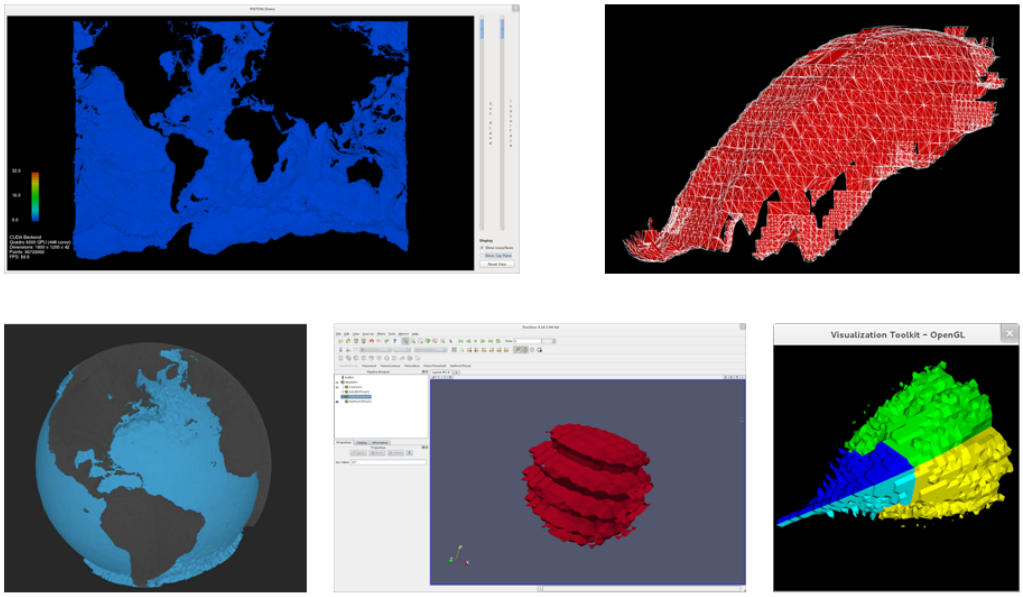

Figure 7. Clockwise, from top left: Isosurface of ocean temperature data set generated by PISTON; isosurface of an AMR data set from an asteroid simulation creating using PISTON's marching tetrahedra algorithm; isosurface generated on four nodes, each running PISTON, using VTK's domain decomposition and image composition; isosurface generated using the PISTON plug-in in ParaView; isotherms on a globe generated using PISTON's curvilinear coordinates