Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs, T. Kuhlen, and F. Marton (Editors)

EAVL: The Extreme-scale Analysis and Visualization Library

J. S. Meredith[1] S. Ahern[1] D. Pugmire[1] and R. Sisneros[2]

[1]Oak Ridge National Laboratory, TN, USA
[2]National Center for Supercomputing Applications, IL, USA

Abstract

Analysis and visualization of the data generated by scientific simulation codes is a key step in enabling science from computation. However, a number of challenges lie along the current hardware and software paths to scientific discovery. First, only advanced parallelism techniques can take full advantage of the unprecedented scale of coming machines. In addition, as computational improvements outpace those of I/O, more data will be discarded and I/O-heavy analysis will suffer. Furthermore, the limited memory environment, particularly in the context of in situ analysis which can sidestep some I/O limitations, will require efficiency of both algorithms and infrastructure. Finally, advanced simulation codes with complex data models require commensurate data models in analysis tools. However, community visualization and analysis tools designed for parallelism and large data fall short in a number of these areas. In this paper, we describe EAVL, a new library with infrastructure and algorithms designed to address these critical needs for current and future generations of scientific software and hardware. We show results from EAVL demonstrating the strengths of its robust data model, advanced parallelism, and efficiency.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems—C.1.3 [Computer Systems Organization]: Processor Architectures—Heterogeneous Systems

1. Introduction

Extracting scientific results from computational simulations is growing increasingly difficult due to the changing hardware and software supercomputing environments. As highlighted in the the 2011 U.S. Department of Energy report "Scientific Discovery at the Exascale", future architectures are expected to show consistent trends: required concurrency will increase tremendously, per-core memory will be reduced, and I/O will be slower relative to both computation and memory speeds [ASM*11]. In response to each of these factors, the software ecosystem must change, further impacting visualization and analysis tool design for these systems. We consider each of these factors in turn.

- **Concurrency:** The Exascale DOE report predicts concurrency will rise by a factor of 40,000 to 400,000 in this decade due to increases in both shared- and distributed-memory parallelism. As such, advanced techniques will be required to take advantage of both types of parallelism. For example, the Message Passing Interface (MPI) [GLS99] can successfully accommodate large distributed parallelism, but relying on it within a shared-memory node is unlikely to result in high efficiencies. Graphics processing units (GPUs) are a current-day example of the types of parallelism which may be required to utilize these future compute nodes.
- **Memory:** The report also predicts available memory will rise by only a factor of 100. Relative to the increase in concurrency, this represents a drastic reduction in per-core memory. Utilizing more efficient representations for data models and developing algorithms with lower requirements for temporary storage will both be helpful in mitigating the effects of the memory reduction.
- **I/O:** Furthermore, the report predicts that the I/O subsystem will be smaller and slower relative to both the computational and memory subsystems. This is, again, a continuation of an existing trend; the amount of computed data it is possible to write to disk is shrinking, and computational simulations often discard many time steps between saving snapshots. Processing data in situ as it is generated [FMT*, WFM11, YWG*10] is one practical solution for analysis methods that work on short windows of simulation time, but sharing computational nodes with the simulations will further constrain memory usage. As post-processing is largely constrained by I/O, where in situ is not possible, analysis software must have strong techniques for minimizing time spent reading from disk.

DOI: 10.2312/EGPGV/EGPGV12/021-030

- **Data:** As simulation codes evolve, new and updated mesh and data models appear. MADNESS (http://code.google.com/p/m-a-d-n-e-s-s/), for example, refines its grid on a per-cell basis, and its variables can be potentially high order (e.g., $K = 20$) Legendre polynomial series, resulting in over 8,000 coefficients for each variable in a single cell. GenASiS (http://astro.phys.utk.edu/activities:genasis) supports complex refinement schemes on high-dimensional grids. Non-physical data are becoming more common, and these are a poor match for visualization tools designed for three-dimensional physical simulations. Molecular data requires mixed-dimensionality fields on a single mesh (0D for atomic numbers and 1D for bond strengths), and many engineering codes require 2D or 1D subsets of a 3-dimensional grid (known as *side sets* and *node sets*, respectively). For visualization tools to correctly analyze the data generated by simulation codes, they must contain a superset of simulation data models.

Taken together, these known software and hardware challenges inform requirements for future general-purpose visualization and analysis tools: they must have highly efficient algorithms, support advanced parallelism including heterogeneous systems, and contain robust data models.

2. Related Work

A common model for production software development is to provide maximal end-user functionality in minimal time. Unfortunately, in the context of visualization and analysis, this approach has largely overlooked needs for the exascale [ASM*11,ARS11], resulting in large risks for achieving scientific discovery in the coming years.

Numerous software toolkits do exist for scientific visualization and analysis, but though many are effective, each has disadvantages. Development of OpenDX (http://opendx.org), open-sourced from IBM's Visualization Data Explorer, has long ceased. AVS/Express (http://www.avs.com) and EnSight (http://ensight.com) both have some parallel support, though both are closed-source, making community expansion and integration challenging. The Visualization Toolkit [SML04] is a de facto standard in open source visualization libraries; two scalable open source visualization and analysis tools, VisIt [CBB*05] (http://visitusers.org) and ParaView (http://paraview.org), both popular in the U.S. Department of Energy high-performance computing community, rely on VTK for their underlying data structures and many staple algorithms.

Unfortunately, the commonly used scientific data model in VTK (`vtkDataSet`) has some shortcomings in terms of parallelism, efficiency, and data model expressiveness. It does not support data parallelism or general acceleration via graphics processors, and leading examples of distributed scaling were accomplished by layering a spatial decomposition on the serial `vtkDataSet` model [CPA*10]. It supports a small number of fixed mesh types, and even scientific data which it can represent correctly is often forced into an inefficient data structure. Furthermore, the data model is limited compared to the demands of contemporary scientific simulation codes, missing support for necessary features mentioned in Section 1, such as mixed-dimensionality elements in a single data set, general high order polynomial elements, non-Cartesian space or dimensionalities other than three, multi-dimensional state spaces, and quadtree meshes. (Though the VTK library includes other, more flexible data types, including `vtkTable` and `vtkGraph`, those cannot be used for general scientific visualization.)

In the late 1990's and early 2000's, several scientific data model libraries were developed, including Field Model [Mor01], CDMLib [ABM*99], Data Models and Formats [Sch00] and Sets And Fields [MRM*01]. These efforts all intended to improve the sharability of scientific data by formalization of the underlying data model. While each drew from the mathematical underpinnings of spatial discretization and vector spaces, as does EAVL, none of the implementations were designed for the restricted memory spaces of future architectures nor the extreme concurrencies of many-core processors.

The move to the next generation of architectures necessitates a new approach to algorithm design. One possibility is a domain-specific language, which may provide for productive and efficient algorithm development [DJP*11, CSB*11]. Another is to make the change from common message passing techniques like MPI [GLS99] to accommodate extreme levels of concurrency. The nearest current measure of future node architectures is the graphics processing unit (GPU), often programmed using explicit data-parallel languages like CUDA (http://developer.nvidia.com/cuda) and OpenCL (http://www.khronos.org/opencl/) or compiler directives [PGI10, DBB07, LE10]. This is the principle behind the Dax toolkit [MAGM11], which provides a framework for high node-level concurrency, implementing a subset of visualization algorithms, though to our knowledge, Dax does not currently address considerations of analysis at the exascale beyond concurrency. PISTON [LSA] is set of analysis operators written on top of Thrust [BH11], which is capable of creating code for GPUs using CUDA and OpenMP, and similarly focuses on node-level concurrency. These works each embody important considerations for general-purpose analysis libraries. However, we propose herein a framework with considerations not just for concurrency and programmability, but also for the memory and I/O constraints of future architectures and for the scientific simulations designed to use them.

3. Details

The Extreme-scale Analysis and Visualization Library (EAVL) project comprises three aims: a new data model, increased efficiency, and new avenues for scalability. EAVL re-

visits many of the assumptions endemic to current large-data visualization and analysis tools. For example, many assume scientific data is always in three dimensional space and falls into a few narrowly defined mesh structures. Such simplistic assumptions have two drawbacks: many types of scientific data simply do not fit into these structures, while the data that do fit are often forced into a less efficient structure. Improvements in the data model can thus lead to improved memory efficiency and improved algorithmic efficiency. EAVL also provides other controls for memory footprint and algorithmic improvements, supports both distributed and data parallelism, and can transparently target heterogeneous systems. Below, we visit each of these improvements in detail, and we measure the contributions of EAVL against the current gold standard for visualization libraries, VTK.

3.1. Data Model

The data model is the foundation for the internal storage and operational aspects of a visualization and analysis library. For example, a traditional data model like that of the VTK data set might be described as a few choices of grid types — such as rectilinear, structured, and unstructured — where each mesh has a set of three-dimensional point locations and a set of cells referencing those points, and fields live on either the points or the cells. In this section, we describe our goals for the EAVL data model and give a high-level overview of its design, discuss the features our design provides, and show detailed examples of its application to new and existing types of scientific data.

3.1.1. Data Model Design Overview

Developers of new data models risk becoming mired in topological mathematics and other quandaries and can become victims of their own ambition. With EAVL, our approach is to make only a few substantive changes to the traditional data model in order to address its main deficiencies. We had a number of high-level goals in mind when designing EAVL's data model, such as allowing more flexible point and cell arrangements, reducing memory usage and memory copies, better supporting non-physical data, efficiently supporting subset topologies, and enabling fine-grain parallelism and support for future system architectures.

For example, in EAVL the data set class is not chosen from a short list of predefined types. Instead, it is more flexible, simply containing zero or more sets of coordinates objects and zero or more sets of cell objects. It also contains a separate logical structure object which describes the basis of arrangement for the points or cells, and it contains a set of fields which are each associated with some part of the mesh.

EAVL provides a variety of concrete types of cell sets. For example, one is for regular arrangements, another is for explicit connectivity, and others are for implicit or explicit subsets of the mesh. EAVL also provides a variety of concrete types of coordinate arrays: all of them refer to one or more of the fields on the mesh, and all have a method to translate their raw point locations into Cartesian space. And finally, the fundamental array class used throughout EAVL is heterogeneous memory space-aware.

3.1.2. Data Model Features

More detail on the data model in EAVL can be found at `http://ft.ornl.gov/eavl`. Here, we expand here on the implications of this overall design:

- **Meshes can have any number of coordinate systems.** This includes the possibility of zero coordinate systems, useful for non-physical data.
- **Each coordinate system has an arbitrary spatial dimension.** This can be less than three, or arbitrarily high. Two coordinate systems on the same mesh may have a different spatial dimension.
- **Coordinate arrays can be interleaved or separated.** For example, explicit coordinates for a VTK unstructured grid must be interleaved into a single three-component array, whereas EAVL allows any mixture of single- or multi-component coordinate arrays on a single mesh.
- **The point structure is independent of the cell structure.** Regular geometry can refer to explicit points, and explicit geometry can refer to regular point arrangements.
- **Meshes have an explicit logical structure.** Separate from the cells and points, it might represent a multi-dimensional regular basis or add a refinement dimension.
- **Fields are associated with one of the mesh structures.** Traditional "cell" and "point" fields are supported, as are fields on the whole mesh, as are fields associated with a logical dimension.
- **Coordinate arrays are fields on the mesh.** In a curvilinear or unstructured mesh, for example, coordinate fields are associated with the mesh points. In a rectilinear mesh, each coordinate array is instead associated with one of the logical dimensions.
- **Each coordinate array can be defined on a different mesh structure.** The X and Y coordinates might be defined on the i and j logical dimensions, for example, with the Z coordinate defined explicitly for every point.
- **Each mesh can have an arbitrary number of *sets* of cells.** Pure "point meshes" need no cells, and might define no cell sets. An unstructured mesh might define multiple groupings of cells on which to associate different fields.
- **Cell sets can be defined in terms of other cell sets.** A subset relationship can be as simple as *"this-set-of-cells-from-"* a different cell set.
- **Faces and edges are simply cell sets.** They use an *"all-faces-of-"* relationship with another cell set, for example.
- **Arrays support handles to accelerator device memory.** They can transparently copy data between host and device memory as needed.

In combination, these (sometimes simple) additions to the traditional data model result in numerous strengths. Below, we explore examples.

3.1.3. Improving Support for Traditional Data

Even simple extensions can have obvious benefit. For example, by allowing fewer than three spatial dimensions, EAVL saves memory which would be wasted on second or third coordinate values for 1D/2D data. This same example also improves performance, as algorithms must only operate on 1/3 or 2/3 of the data. These and other examples of how the data model improves memory and performance efficiency are detailed below in Section 3.2.

By supporting both interleaved and separated coordinate arrays, we can achieve higher device bandwidth on GPUs where one variant may result in a more optimal memory access pattern. By encapsulating both host and device memory handles, arrays contain features necessary to support heterogeneous node architectures. These and other considerations for data parallelism are detailed in Section 3.3.

Other extensions make working with scientific data more convenient. For example, a geospatial data set on the surface of the Earth is simultaneously defined in two coordinate systems: two-dimensional latitude/longitude and three-dimensional X/Y/Z. As these coordinate arrays are simply mesh fields, they can be analyzed as easily as other field data.

These coordinate arrays may also be used to address common data modeling difficulties. For example, after slicing a curvilinear 3D grid (i.e., with explicit point coordinates) by the plane $Z = 6$, storing this "6.0" value only once for the entire grid is not just more memory efficient and generally higher performance, it is more elegant.

Other extensions make working with scientific data more correct. Molecular data is one example: in a data model supporting only a single set of cells on a mesh, these cells must include both atoms and bonds. Within this environment, a field on the bonds (like covalent bonding strength) must include dummy values for the atoms, as a field on the cells must have values for every mesh cell. Any existing analysis algorithm applied to this field would have incorrect results, as it would be including these dummy values. By allowing the atoms and bonds to be separate sets of cells, fields can be defined on only one of the two, and analysis algorithms will automatically (and correctly) use only relevant values.

3.1.4. Supporting Non-traditional Data

Another goal of creating a new data model is to ensure we can support the complex structures of modern simulation applications. Just as rethinking the taxonomy of mesh classification improves support for data that fit into the traditional model like that of the VTK data set, it also allows support for types of data that do not.

Making the choice of cell and point structures more independent is one way of enabling new types of data sets, as data is no longer constrained to arbitrary combinations of these structures, like "unstructured" and "curvilinear." Adding a refinement dimension in a logical structure allows a basis for block-structured AMR (advanced mesh refinement) grids, or even per-cell refinement grids like quadtrees.

By allowing multiple cell sets on a single mesh, we enable concepts like "side sets" where faces of a subset of volumetric cells are collected into an explicit grouping. Side sets share the points with the volumetric grid, but only some variables exist on both the volumetric and surface elements. Without this feature, one must represent the volumetric and surface elements as different meshes, and mapping the relationship between the two would be extremely expensive as the meshes cannot share a common point indexing.

Similarly, a volumetric subset of volumetric cells gives an efficient representation of the result from a "threshold" operation. This also provides a native solution for node sets and flux surfaces, and for face and edge data, all of which have traditionally been a challenge in analysis tools.

The additional self-descriptiveness of fields allows more informed interpretation. For example, visualization tools often interpolate point-based fields linearly along adjoining cells. However, fields like "atomic number" on a molecular mesh cannot be interpolated, and EAVL allows us to tag the field as piecewise constant. This descriptiveness also clarifies the interpretation of a three-component cell array as a high-order polynomial on the cells instead of a 3D vector.

And one of the simplest examples is native support for data with unusual spatial dimensions. These might be pure parameter studies (like a 4D data cube whose values are the reaction rates under varying concentrations of four chemical species) or a simple graph of vertices and edges with no spatial coordinates whatsoever. Concepts like *time* can also be treated efficiently as coordinate axes here.

3.2. Efficiency

3.2.1. Memory Efficiency

One of the most obvious needs for improvement in future analysis software is in memory efficiency; as mentioned, growth in system memory capacity is slow, and in situ scenarios place additional burdens on memory usage. Fortunately, a more descriptive data model, as described above, has positive implications for efficiency:

- A 2D curvilinear grid can cut its memory usage by one third when we remove the requirement for the Z coordinate array, generally all zeros. Similarly, one dimensional or entirely non-spatial data can realize substantial savings.
- Due to the separation of point and cell structures, instead of creating an explicit unstructured grid to represent an irregular subset of a regular grid, we can retain the more efficient regular point arrangement and make only the cell connectivity explicit.
- Better yet, this irregular subset can be represented even more compactly as two cell sets on the same mesh: the first contains the original regular cells, and the second is a cell set referencing an explicit subset of the first.

- As each coordinate axis can be independently implicit or explicit in EAVL, a 2D terrain image elevated by a height field into 3D could have its X and Y axes remain a space-efficient implicit outer product, with the Z axis containing the only explicitly represented values.
- A skewed regular grid, common in molecular data, could compactly specify its coordinate axes as a 4×4 matrix transformation relative to Cartesian X/Y/Z axes.

EAVL includes improved controls for memory usage as well. For example, some algorithms support operation in-place; a trivial example would be a coordinate displacement operation which simply overwrites each point, providing nearly zero memory growth. We have also explored adding tiling to our basic array structure: arrays appear allocated at initialization, but may actually be allocated (or freed) in blocks as necessary, allowing algorithms to free an input mesh as the output mesh is generated.

3.2.2. Algorithmic Efficiency

Memory efficiency can translate directly into algorithmic efficiency. As data movement becomes a driving factor in overall computational performance more than raw arithmetic prowess, moving fewer bytes will result in not just reduced memory usage but also reduced run times for many algorithms. For example, as EAVL requires just two coordinate values for two-dimensional data, it cuts not just memory usage but also memory accesses by one third. In fact, all memory savings suggested in Section 3.2.1 can contribute to performance improvements.

This more descriptive data model allows improved performance in other ways as well. For example, by allowing implicit coordinates to remain implicit, we save the step of conversion to explicit coordinates. By allowing a native subset-of-cells construct, we skip the step of creating an entirely new data set during a threshold operation. By allowing rectilinear grids, even when in transformed space, to stay rectilinear, the many algorithms which operate more efficiently on rectilinear grids can generally continue to do so, with only a small extension.

3.3. Scalability

3.3.1. Node-level Parallelism

To achieve data parallelism in EAVL algorithms, we adopt a functor/iterator model: a function object, or functor, is an object which encodes an operation, and an iterator defines the execution pattern. By separating these two pieces, EAVL allows more ways to combine them, increasing reuse and flexibility. This is much like the high productivity computational model of Thrust [BH11]. Here, however, we provide a set of execution patterns based on mesh topological constructs (like a node-to-cell pattern, a cell-to-face pattern, and so on), and these support functors with inputs and outputs common to visualization algorithms such as scalar fields and point coordinates. Furthermore, we provide both OpenMP and CUDA implementations of these execution patterns, and the same functor code may be passed to either. This enables algorithms to be be written once and execute efficiently on several data-parallel architectures. Within a node, algorithm developers may always revert to serial code or manual CUDA/OpenMP kernels; by supporting heterogeneous memory spaces automatically, the built-in EAVL array type simplifies this process.

Listing 1 shows an example of CUDA code implementing the node-to-cell operation in EAVL. The actual code in EAVL contains other optimizations, and it is slightly more complex — for example, it handles both interleaved and separated coordinates via a multiplier and offset for each input array, and a variant for structured cells needs only the mesh dimensions instead of explicit connectivity. The CPU execution path simply uses a *for* loop over the output cells, with a pragma to enable OpenMP parallelism. Note that `ExplicitConnectivity` contains dynamic data but transparently handles the transfer to device memory. Also, the result is transfered back to the host later, but only when necessary, allowing data to remain on the device as much as possible.

Listing 2 shows this pattern being instantiated to calculate the surface normal using a cross-product functor. It also highlights how a developer can pass values to functors during construction. Functors have other benefits as well: unlike function pointers, they are available even in languages (like older CUDA compute capabilities) where function pointers are not, and they can be effectively inlined by compilers.

Listing 1: *CUDA code in EAVL for a node-to-cell execution pattern for explicit cell connectivity and three inputs.*

```
template <class F>
__global__ void NodeToCellKernel3(float *array0,
                                  float *array1,
                                  float *array2,
                                  float *out,
                                  ExplicitConnectivity conn,
                                  F functor)
{
  const int index = blockIdx.x * blockDim.x + threadIdx.x;
  int nNodes, nodeIds[8];
  float nodeValues[3][8];

  conn.GetCellNodes(index, nNodes, nodeIds);
  for (int i=0; i<nNodes; i++)
  {
    nodeValues[0][i] = array0[nodeIds[i]];
    nodeValues[1][i] = array1[nodeIds[i]];
    nodeValues[2][i] = array2[nodeIds[i]];
  }
  functor(nodeValues[0],
          nodeValues[1],
          nodeValues[2],
          &out[index * 3]);
}

void NodeToCellOp3::ExecuteCUDA()
{
  float *d_arr0 = (float*)array0->GetCUDAArray();
  float *d_arr1 = (float*)array1->GetCUDAArray();
  float *d_arr2 = (float*)array2->GetCUDAArray();
  float *d_out  = (float*)output->GetCUDAArray();
  // calculate CUDA thread grid nb/nt
  nodeToCellKernel3<<<nb,nt>>>(d_arr0, d_arr1, d_arr2,
                               d_out, conn, functor);
}
```

Listing 2: *To calculate face surface normals, a developer instantiates the node-to-cell pattern using mesh coordinates as the input arrays and a cross product functor.*

```
struct PolyNormalFunctor
{
  bool normalize;
  PolyNormalFunctor(bool n) : normalize(n) { }
  void operator()(float *x, float *y, float *z, float *n)
  {
    float ax = x[1]-x[0], ay = y[1]-y[0], az = z[1]-z[0];
    float bx = x[2]-x[1], ay = y[2]-y[1], az = z[2]-z[1];
    n[0] = ay*bz - az*by;
    n[1] = az*bx - ax*bz;
    n[2] = ax*by - ay*bx;
    if (normalize)
    {
      float len = sqrt(n[0]*n[0]+n[1]*n[1]+n[2]*n[2]);
      n[0]/=len;  n[1]/=len;  n[2]/=len;
    }
  }
};


void CalculateFaceNormals(...)
{
  // ...
  executor->AddOperation(
    new NodeToCellOp3(xcoord, ycoord, zcoord,
                      outputnormals,
                      inputcells,
                      PolyNormalFunctor(false)));
}
```

Developers can combine individual data-parallel patterns into complex sequences. For example, in an *isosurface* operation, a node-to-node pattern evaluates a nodal field against a target value, and a node-to-cell pattern evaluates these booleans as an integer bit pattern, resulting in a lookup case (via a gather pattern) for the isosurface tables. As most cells do not contribute to the resulting isosurface, it is more efficient to parallelize over output geometry. To accomplish this, we use a reduction pattern to calculate the number of output pieces, a prefix sum pattern to find each cell's starting index into the output geometry, and a scatter pattern to map the isosurface cases into the output arrays. A node-to-edge pattern then calculates the final coordinate locations for the nodes of the isosurface mesh.

3.3.2. Distributed Parallelism

In [CPA*10] we see that a pure spatial decomposition is effective at scaling to current node counts and very large data sets. In EAVL, we support this same general decomposition strategy via MPI, allowing ghost cells for operations where only a local neighborhood of information for each cell is necessary to minimize parallel communication. Allowing MPI at the functor level is impossible (as these must support execution on GPUs), but EAVL allows MPI calls at the filter level for more explicit communication when necessary.

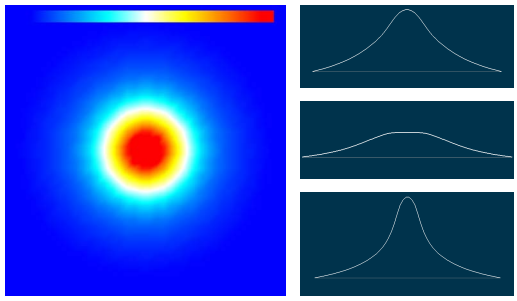
In addition to single-block file readers (e.g., a legacy VTK importer) and simulation-specific readers (e.g., MADNESS and CHIMERA), EAVL contains file format readers which support parallelism, such as Silo and BOV, and a NetCDF reader supporting automatic domain decomposition and parallel hyperslab I/O. EAVL also supports a file-list importer which interprets a series of files as a time sequence. EAVL's native understanding of discrete dimensions also provides another dimension along which to parallelize; section 4.3.3 shows the advantages of spatiotemporal parallelism support.

4. Results

4.1. Data Model

In this section we present results highlighting our design decisions in EAVL to support the complexities of current and future simulation codes which are difficult to represent correctly using traditional data models.

4.1.1. High-dimensionality Grids in CHIMERA



(a) collapse to 2D (b) collapse to 1D

Figure 1: *Dimensional collapse of a 5D CHIMERA data set. (a) 2D collapse to X/Y/Z, then a slice. (b) 4D collapse to X for the electron, anti-electron, and tau neutrino flavors.*

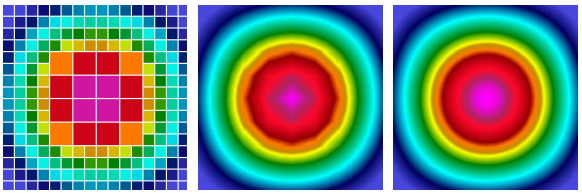
CHIMERA [BMH*10] is a tightly coupled multi-physics code for simulating core-collapse supernovae. CHIMERA calculates stellar gas hydrodynamics and nuclear kinetics, represented on a three dimensional spatial grid, and ray-by-ray neutrino transport, represented on a five dimensional grid consiting of three spatial dimensions, neutrino flavor (electron, anti-electron, tau and anti-tau) and energy group (currently 20 levels). The flexibility of the data model in EAVL allows an accurate and consise representation of CHIMERA data.

Since many visualization algorithms only operate in two or three dimensions, dimensionality reduction filters have been implemented. Figure 1 shows the ψ_0 variable in a 5D CHIMERA data set after dimensional collapse operations. In Figure 1(a), the data set was reduced by averaging the neutrino flavor and energy group dimensions, followed by a center slice of the resulting spatially-3D data set. In Figure 1(b), the data set was collapsed by averaging the Y and Z spatial dimensions as well as the energy group dimension, and the result plotted for three neutrino flavors along the X dimension.

4.1.2. High Order Quadtrees in MADNESS

Quadtrees and octrees are tree based data structures which can be used for partitioning a two- or three-dimensional

space, respectively. They have gained popularity in the visualization community as an efficient acceleration structure. However, their use as a mesh structure is not common, and so visualization tool support is generally unavailable. MADNESS (http://code.google.com/p/m-a-d-n-e-s-s/) is a multi-resolution, adaptive framework for simulations that uses quadtrees and octrees as a mesh structure. The variable values in the quadtree are high-order, represented by coefficients for Legendre polynomials of up to order 20.



(a) constant (b) bi-linear (c) bi-quadratic

Figure 2: *Rendering a high-order MADNESS quadtree mesh with different methods of interpolation.*

Figure 2 shows the quadtree mesh structure of a MADNESS data set with a standard per-cell refinement. For this data set, each cell has nine values: coefficients for a bi-quadratic function. In Figure 2(a) EAVL renders the data set by coloring each cell using the value of this function at its center. In 2(b), we see the result when values are assigned to the nodes of the mesh, rendering with standard bilinear interpolation. Note the discontinuities where the refinement level changes between adjacent cells. In Figure 2(c), we see the result when these high-order coefficients are passed down to the rendering infrastructure where tessellation occurs the fly — and as such, requires no extra memory. EAVL also supports tesselation within a visualization pipeline, so high order fields can be approximated using finer-grain linear fields that are more widely supported in common operations such as isosurfacing.

4.2. Efficiency

Another of the primary design considerations in EAVL was memory efficiency. In this section we explore several examples that highlight the benefits of these design decisions.

4.2.1. 2D to 3D Elevation

	rectilinear		*structured*		*unstructured*	
	before	after	before	after	before	after
EAVL	11 kB	11 kB	21 kB	21 kB	17 kB	17 kB
VTK	11 kB	21 kB	26 kB	21 kB	19 kB	17 kB

Table 1: *Memory required before and after elevating three different data sets in both VTK and EAVL.*

Table 1 shows a comparision of the memory usage of an elevation operation on three small two-dimensional data sets containing two scalar fields, u and v. In each case, the mesh was elevated into three dimensions by a height field defined by v, and then colored by u. Note that we measure VTK memory usage here through the number of bytes each data set occupies as a binary file on disk, as this avoids counting temporary internal memory usage.

VTK does not support the combination of coordinate arrays on logical dimensions with ones associated with the points, so the rectilinear data set must be converted to the more explicit curvilinear grid. For the curvilinear and unstructured inputs, the operation copies the v array elements over the third (dummy) coordinate value already existing in the explicit coordinate arrays, and then discards the v array. Thus, starting memory usage is higher, and after the linear-time copy operation shrinks to the same usage as EAVL.

In the case of EAVL, for each input only a constant-time modification of the meta-data is required to set the third coordinate axis to point to the existing scalar field v; memory usage is also nearly constant.

4.2.2. Rectilinear Threshold

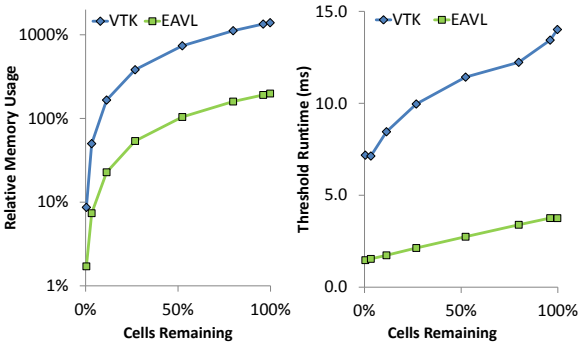


Figure 3: *Memory and runtimes for various threshold values subsetting a rectilinear grid.*

In Section 3.1, we described the ability for a single mesh to mix and match point and cell structures, and for one cell set to refer to subsets of other cell sets. A threshold operation on a regular grid provides a concrete example of these benefits. In Figure 3(a), we show the memory usage of the resulting data set after thresholding the rectilinear *noise* data set (included with VisIt) by a variety of cutoff values. In this figure, we see that the threshold operation resulting in a fully explicit VTK unstructured grid resulted in a memory increase of up to 14× in the worst case. In EAVL, this same scenario was a seven-fold improvement compared to VTK. In Figure 3(b), we see the improvement in runtime of the same operation; in EAVL, this operation requires not just less storage, but also less computational work due to the more descriptive data model.

4.2.3. Face Data

A native representation of face and edge data is another strength of the EAVL data model. Without this ability, users must find workarounds for storing face-centered field data.

For example, a 100^3 regular grid has $99 \times 99 \times 100 \times 3 = 2.9M$ faces. In a data model which does not support face data on regular grids, one might store these faces as a fully explicit polygonal data set, or as three hundred regular 2D grids (100 along each of the three coordinate axes). As Table 2 shows, the latter option comes closer in memory usage to the native face data avenue as present in EAVL, with the added expense of managing 300 meshes which were intended to be one. And with either workaround, the face data must be in a separate mesh from the volumetric cells, or one cannot manage fields separately on each — and unfortunately, in a separate mesh one loses the intrinsic information mapping the faces to the cells and points of the original mesh. With EAVL, however, the single regular grid retains the mapping between faces, volumetric cells, and the points.

Single VTK polygonal data set	74.8 MB
300 VTK regular grids	12.3 MB
Single EAVL regular grid	11.8 MB

Table 2: *Memory required for storage of the mesh and a single face-centered scalar field on a* 100^3 *regular grid.*

4.2.4. In-Place Algorithms

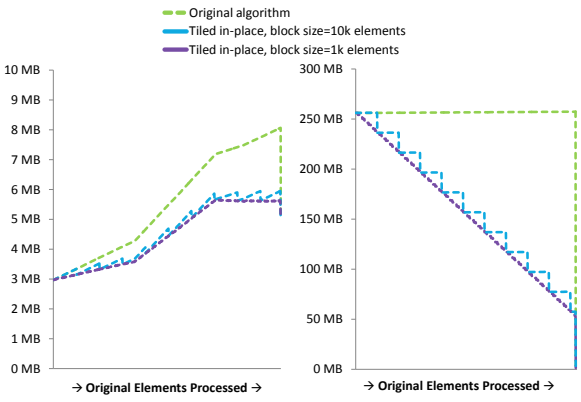


Figure 4: *Memory usage over time, with and without tiling. Left: external facelist filter on a rectilinear data set. Right: isosurface on the* vortex *data set.*

As mentioned in Section 3.2.1, for data-parallel algorithms, it is possible to free data as it is consumed. To accomplish this wile minimizing overheads and retaining sufficient data level parallelism, we added experimental support for tiling arrays in EAVL into chunks of a configurable size. To test this capability, we added code to the isosurface and external facelist algorithms to tag elements as no longer needed, allowing EAVL to free their memory in chunks. This allowed these algorithms to operate in an in-place manner with little developer effort. Figure 4 shows memory usage as these algorithms proceed on two data sets. Of course, final memory usage is identical with or without chunking when the algorithm completes, as one can then free the input data entirely. Compared to an unmodified algorithm, though, both peak and average memory usage can be significantly lower operating in this mode.

4.3. Parallelism

In this section, we explore examples of EAVL's multiple levels of parallelism designed to address the concurrency constraints of coming generations of supercomputing architectures.

4.3.1. Scaling

To evaluate the capability of EAVL to support distributed scaling, we executed parallel file reader I/O, isosurface, and then surface normal operation on a large, 100-billion cell, 1000-computational domain data set. The source data set was a core-collapse supernova simulation on a curvilinear mesh using the CHIMERA code. We up-sampled the entropy variable onto a 4640^3 rectilinear grid, reading it in a strong scaling mode using 125 to 500 processors of the Lens cluster at Oak Ridge National Laboratory, calculating a sequence of isosurfaces and their surface normals. The results, in Table 3, show high scaling efficiencies.

Number of Processors	125	250	500
Total Runtime	1428 sec	707 sec	353 sec

Table 3: *Runtimes to load and calculate a series of isosurfaces, and their surface normals, on 100-billion cell data set.*

4.3.2. Data Parallelism

To explore data parallelism in EAVL, we examine how the face normal operation from Section 3.3.1 performs in our data-parallel framework. This operation is implemented as a cross-product functor passed to a node-to-cell iteration pattern.

As previously noted, EAVL currently provides both CUDA and OpenMP back ends for its iteration patterns, and both use the same functor code provided by the algorithm developer. Figure 5 illustrates the timings of this surface normal filter when run on the *noise* data set. These results show significant speedups using both OpenMP and the GPU. Note that timings include all kernel launch overheads, but not PCI-Express data transfer times on the GPU, as we expect a significant proportion of chained filters to operate on the same device, thus amortizing or eliminating any transfer costs.

In addition to running on GPUs and multi-core CPUs, we also ported EAVL to the Intel® Many Integrated Core

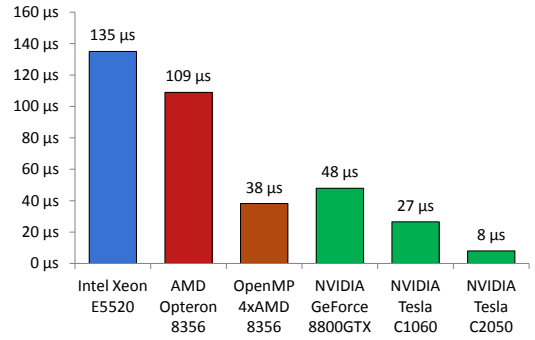


Figure 5: *Timings for serial and parallel runs to calculate the face-centered surface normal on a noise data set.*

(MIC) Architecture Software Development Platform. Using the Intel compiler to generate native code for the accelerator (codenamed Knights Ferry) using OpenMP for data-level parallelism, we ran the surface normal calculation on the same data set. Although absolute performance numbers from this pre-production development platform would not be representative of a final product, we could investigate scaling of the architecture. Table 4 shows the parallel efficiency across a range of threads. As before, we do not measure PCIe transfer times, but we do include overheads from OpenMP thread launches. We see an initial efficiency penalty from enabling OpenMP threading, but efficiency quickly improves, peaking around 16 threads for this problem. Runtimes improved consistently with each increase in thread count; the fastest runtime was achieved with 120 threads. As these results show a strong combination of high efficiency and ease of programming, we believe our data-parallel strategy is effective for not CPUs and GPUs but the Intel MIC architecture as well, and we plan to continue our support for it.

Threads	2	3	4	8	12	16	20	30	60	90	120
Efficiency (%)	63	73	74	80	81	85	81	70	62	50	40

Table 4: *Scaling efficiency by thread count (relative to single-threaded performance) on the Intel MIC SDP for the surface normal calculation on a noise data set.*

4.3.3. Distributed Parallelism

Figure 6 shows a scaling experiment we performed on the Lens cluster at ORNL to examine the benefits of temporal parallelism calculating the maximum-over-all-time on a 50-year climate simulation from the Community Climate System Model (CCSM). Note that the number of nodes (and available I/O bandwidth) increased when using up to 16 processors. To determine the performance baseline of spatial parallelism, we compared against VisIt, known to scale well with a spatial decomposition. While VisIt performed almost identically to EAVL with a single processor, it was clear that the spatial parallelism in VisIt reached scaling limits before an equivalent EAVL analysis using temporal parallelism. To

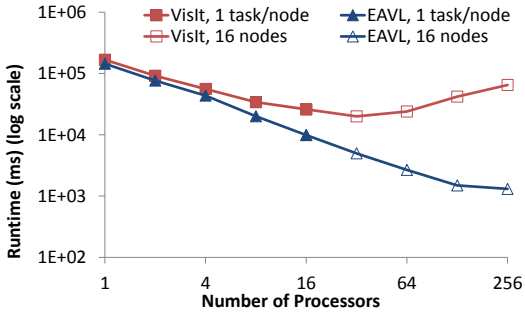


Figure 6: *Strong scaling runtimes to calculate the maximum value over all time from a 50-year CCSM NetCDF data set. EAVL runs used only temporal parallelism, and VisIt runs used only spatial parallelism. Scaling up to 16 processors used 1 task/node, and up to 256 processors by adding tasks on 16 nodes.*

further explore this effect, we ran this same test using various combinations of spatial and temporal parallelism. The results are shown in Table 5. Although temporal parallelism generally results in higher efficiency, it also increases memory usage; the hybrid parallelism enabled by EAVL thus allows for more fine-grained trade-offs between maximizing I/O efficiency and minimizing memory usage.

Temporal Parallelism	Spatial Parallelism			
	1-way	2-way	4-way	8-way
16-way	-	87%	79%	36%
32-way	105%	92%	50%	20%
64-way	98%	78%	42%	
128-way	87%	59%		
256-way	69%			

Table 5: *Scaling efficiency of temporal versus spatial parallelism on a 50-year CCSM NetCDF data set. Efficiency is calculated relative to the runtime with 16-way temporal parallelism and no spatial parallelism.*

5. Conclusion and Future Work

We have implemented an analysis and visualization library with a combination of features to support current and future scientific software and hardware platforms. With a strong data model, EAVL supports complex structures like high dimensionality, arbitrary refinement and mixed topology meshes that are becoming more common in simulation applications but are poorly supported in traditional visualization libraries. This data model also forms a basis for robust memory and algorithmic efficiency. Parallelism is implemented at multiple levels, supporting heterogeneous memory spaces and data-parallel architectures like multi-core CPUs, many-core accelerators like GPUs, and hybrid distributed parallelism for improved scaling at larger node counts.

We plan to continue development of EAVL towards production-ready status; more details will be made available at http://ft.ornl.gov/eavl. Our plan includes continued improvements to the data model, such as optimizations of data layouts, additional support for arbitrary refinement grid topologies, data-based partitions, and additional predefined data-parallel functors and iteration patterns. We also plan to increase the number of built-in algorithms, develop an execution framework for data flow pipelines, and provide an easy to use application programming interface. Finally, we will explore deployment avenues such as integration with existing visualization tools and integration in situ with simulation codes.

6. Acknowledgments

The authors would especially like to thank the anonymous reviewers for their insightful feedback. Research sponsored by the Laboratory Directed Research and Development (LDRD) program of Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. As such, the U.S. Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or allow others to do so, for Government purposes only.

References

[ABM*99] AMBROSIANO J., BUTLER D., MATARAZZO C., MILLER M., SCHOOF L.: *Development of a common data model for scientific simulations*. Tech. Rep. LA-UR–99-722; CONF-990707, Los Alamos National Lab., NM, USA, 1999. 2

[ARS11] AHRENS J., ROGERS D., SPRINGMEYER B.: Visualization and data analysis at the exascale. National Nuclear Security Administration (NNSA) Accelerated Strategic Computing (ASC) Exascale Environment Planning Process. 2

[ASM*11] AHERN S., SHOSHANI A., MA K.-L., CHOUDHARY A., CRITCHLOW T., KLASKY S., PASCUCCI V., AHRENS J., BETHEL E. W., CHILDS H., HUANG J., JOY K., KOZIOL Q., LOFSTEAD G., MEREDITH J., MORELAND K., OSTROUCHOV G., PAPKA M., VISHWANATH V., WOLF M., WRIGHT N., WU K.: *Scientific Discovery at the Exascale, a Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization*. 2011. 1, 2

[BH11] BELL N., HOBEROCK J.: Thrust: A productivity-oriented library for CUDA. In *GPU Computing Gems*, Hwu W.-M., (Ed.). Elsevier/Morgan Kaufmann, 2011, pp. 359–371. 2, 5

[BMH*10] BRUENN S. W., MEZZACAPPA A., HIX W. R., BLONDIN J. M., MARRONETTI P., MESSER O. E. B., DIRK C. J., YOSHIDA S.: 2d and 3d core-collapse supernovae simulation results obtained with the chimera code. *USA. AIP Conference Proceedings 180*, arXiv:1002.4914 (Mar 2010). 6

[CBB*05] CHILDS H., BRUGGER E. S., BONNELL K. S., MEREDITH J. S., MILLER M., WHITLOCK B. J., MAX N.: A contract-based system for large data visualization. In *Proceedings of IEEE Visualization 2005* (2005), pp. 190–198. 2

[CPA*10] CHILDS H., PUGMIRE D., AHERN S., WHITLOCK B., HOWISON M., PRABHAT, WEBER G. H., BETHEL E. W.: Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications 30*, 3 (2010), 22–31. 2, 6

[CSB*11] CHAFI H., SUJEETH A. K., BROWN K. J., LEE H., ATREYA A. R., OLUKOTUN K.: A domain-specific approach to heterogeneous parallelism. In *16th Annual Symposium on Principles and Practice of Parallel Programming* (2011). 2

[DBB07] DOLBEAU R., BIHAN S., BODIN F.: HMPP: a hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)* (2007). 2

[DJP*11] DEVITO Z., JOUBERT N., PALACIOS F., OAKLEY S., MEDINA M., BARRIENTOS M., ELSEN E., HAM F., AIKEN A., DURAISAMY K., DARVE E., ALONSO J., HANRAHAN P.: Liszt: a domain specific language for building portable mesh-based pde solvers. In *Conference on High Performance Computing Networking, Storage and Analysis* (2011), p. 9. 2

[FMT*] FABIAN N., MORELAND K., THOMPSON D., BAUER A. C., MARION P., GEVECI B., RASQUIN M., JANSEN K. E.: The ParaView coprocessing library: A scalable, general purpose in situ visualization library. 1

[GLS99] GROPP W., LUSK E., SKJELLUM A.: *Using MPI: portable parallel programming with the message-passing interface*, 2nd ed. MIT Press, Cambridge, MA, 1999. 1, 2

[LE10] LEE S., EIGENMANN R.: OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11. 2

[LSA] LO L.-T., SEWELL C., AHRENS J.: *PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators*. Los Alamos National Laboratory. 2

[MAGM11] MORELAND K., AYACHIT U., GEVECI B., MA K.-L.: Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *Proceedings of the 2011 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)* (2011), pp. 97–104. 2

[Mor01] MORAN P. J.: *Field Model: An Object-Oriented Data Model for Fields*. Tech. Rep. NAS-01-005, NASA Ames Research Center, 2001. 2

[MRM*01] MILLER M. C., REUS J. F., MATZKE R. P., ARRIGHI W. J., SCHOOF L. A., HITT R. T., ESPEN P. K.: Enabling interoperation of high performance, scientific computing applications: Modeling scientific data with the sets & fields (saf) modeling system. In *Proceedings of the International Conference on Computational Science-Part II* (London, UK, UK, 2001), ICCS '01, Springer-Verlag, pp. 158–170. 2

[PGI10] THE PORTLAND GROUP: *PGI Accelerator programming model for Fortran & C*, Nov 2010. 2

[Sch00] SCHOOF L.: The ASCI data models and formats (DMF) effort: A comprehensive approach to interoperable scientific data management and analysis. *In 4th Symposium on Multidisciplinary Applications and Interoperable Computing* (2000). 2

[SML04] SCHROEDER W., MARTIN K., LORENSEN B.: *The Visualization Toolkit, Third Edition*. Kitware Inc., 2004. 2

[WFM11] WHITLOCK B., FAVRE J. M., MEREDITH J. S.: Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (2011), Kuhlen T., Pajarola R., Zhou K., (Eds.), pp. 101–109. 1

[YWG*10] YU H., WANG C., GROUT R. W., CHEN J. H., MA K.-L.: In-situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications 30*, 3 (May/June 2010), 45–57. 1