SAND2012-10129P

Breaking Up the Cell Class

From Daxtoolkit

The current version of Cell in the execution environment is designed to encapsulate all pertinent information about a cell (much like the vtkCell classes in VTK). Unfortunately, we are finding that often we only want part of the information for the cell, and every piece of extra information we load, compute, or store adds to unnecessary overhead to an algorithm. This document proposes splitting the Cell class into multiple pieces each capturing a specific concept.

Contents

Current implementation and motivation

The current implementation of the cell classes is expected to encapsulate information relevant for the cell relative to the topology. All cell classes can return the point indices for each vertex. In addition, the CellVoxel class also stores the origin, spacing, and extent for the grid.

However, most times you only need one of these pieces of information. For example, if a specialized voxel gradient is using the spacing to compute the gradient, it does not need the point indices. Likewise, a threshold topology worklet passing vertices does not need information about origin, spacing, and extent.

Proposed solution

The proposed solution is based off the changes suggested byte the Cell Type Tags. The first step is to blow away all of the cell classes. These will then be placed with targeted template classes that address a specific function. Here are some suggested classes.

Cell Vertices

The basic descriptor of a cell is the list of indices that maps the vertices of the cell to the point index of

the mesh. This is used in topology generation to create new cells using points of the original mesh. This can also be used to create keys for topology resolution.

These indices can be represented in a templated class like the following. The obvious implementation is not listed here. (I've also left off all the DAX_EXEC_EXPORT macros.)

```
template<class CellTag>
class CellVertices
{
public:
  const static dax::Id NUM_POINTS = dax::CellTraits<CellTag>::NUM_POINTS;
  CellVertices(const dax::Tuple<dax::Id,NUM_POINTS> &pointIndices);
  dax::Id GetPointIndex(dax::Id vertexIndex) const;
  void SetPointIndex(dax::Id vertexIndex, dax::Id pointIndex);
  const dax::Tuple<dax::Id,NUM_POINTS> &GetPointIndices() const;
  void SetPointIndices(const dax::Tuple<dax::Id,NUM_POINTS> &pointIndices);
};
```

Although it is possible to simply represent CellVertices directly as a dax::Tuple (since it really is a thin wrapper around that class), I advocate having this separate class. First, it makes the intention more clear throughout the code. Second, it makes template parameters cleaner and more clear. For example, a declaration using CellVertices might look like this

```
dax::exec::CellVertices<CellTag> vertices;
```

whereas the tuple declaration would like like this

```
dax::Tuple<dax::Id,dax::CellTraits<CellTag>::NUM_POINTS> vertices;
```

Extent

For those grid types that use extent information (uniform, rectilinear, and curvilinear grids), it can be useful to get the extent information for the grid. This information can be used to create unique identifiers for faces and edges.

> Then again, maybe it is better to have a special binding that gives unique ids for these elements than pass around this specialized metadata. --16:23, 21 November 2012 (EST)

Axis Aligned Widths

There are some operations (such as derivatives and parametric coordinates) that can take advantage of the fact that elements like voxels are axis aligned. They do this by using the widths of the cell along each axis (basically the spacing attribute of a uniform grid) instead of using absolute point coordinates.

Binding

Because we have broken the Cell class into several pieces, it means that we have to be more clear about the binding of worklet arguments. To do this, we need to restore the BindCellPointIds functionality (see

SHA 8f4ad753bfbaed46f0ff4dc3ad914ea03cdf0708) and expand upon that other bindings for the other information listed above.

Because there is not default cell class, there should be a descriptive compile error if no specific binding is given for topology in the ExecutionSignature.

Acknowledgements

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 Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.