Dax: Data Analysis at Extreme

Kenneth Moreland
Sandia National Laboratories

Utkarsh Ayachit
Kitware, Inc.

Berk Geveci
Kitware, Inc.

Kwan-Liu Ma
University of California at Davis

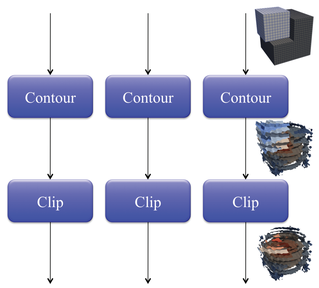
Abstract

Experts agree that the exascale machine will comprise processors that contain many cores. Furthermore, physical limitations will prevent data movement in and out of the chip (that is, between main memory and the processing cores) from keeping pace with improvements in overall compute performance. To use these processors to their fullest capability, it is essential to carefully consider fine grained concurrency and memory access.

This project investigates a new type of visualization framework that exhibits a pervasive parallelism necessary to run on exascale machines. Our framework achieves this by defining algorithms in terms of localized stateless functions. These functions can be connected in much the same way as filters in the visualization pipeline. But our framework's design allows functions to be concurrently running on massive amounts of lightweight threads. Only with such fine-grained concurrency can we hope to fill the billions of threads we expect will be necessary for efficient computation on an exascale computer.

Current Visualization Pipeline

Most of today's visualization libraries and applications are based off what is known as the visualization pipeline. In the visualization pipeline model, algorithms are encapsulated as filter components with inputs and outputs. These filters can be combined by connecting the outputs of one filter to the inputs of another.



The most successful way of achieving concurrency is to use a data-parallel approach. The input mesh is partitioned, and the partitions are divided amongst distributed memory nodes. The pipeline is replicated on all nodes, and the same operations are done on each partition.

Scaling to Extreme

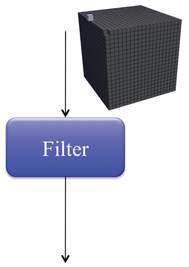
According to the International Exascale Software Project Roadmap, we expect an exascale computer to require 50,000 times more concurrent threads and provide only 500 times the memory.

	Jaguar – XT5	Exascale	Increase
Cores	224,256	100 M – 1 B	~1,000×
Concurrency	224,256 way	10 billion way	~50,000×
Memory	300 TB	128 PB	~500×

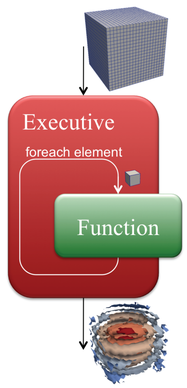
Will current visualization scale? Implementations of tools like ParaView and VisIt rely on MPI for most of their concurrency. Using MPI to generate the requisite threads will require more memory than available on the system. Even ignoring this problem, an extreme-scale computer will require the data to be partitioned on too fine a level to effectively run a visualization pipeline.

Revisiting the Pipeline

We need a visualization framework that performs on the finest level of concurrency possible. Consider a filter-type object that operates on a single element of a mesh. In order for this unit to be executed concurrently over all elements of the mesh, it must be completely stateless and have memory access limited to the "safe" locations given to it. The solution is remarkably basic: a function. This stateless serial function is the basic building block in the Dax Toolkit and the unit the visualization algorithm developer creates.



Dax Algorithm Execution



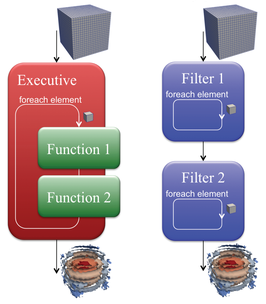
The Dax Toolkit provides a unit called an executive that accepts a mesh, iterates over all elements in the mesh, invokes one or more of these stateless functions on each element, and collects the resulting values for each element. Conceptually we can think of this iteration as a serial operation, but of course in practice the executive will schedule the operation on multiple threads. Because the function is constrained to be stateless and operate on the single element it is given, the execution can be scheduled concurrently without danger of pitfalls such as race conditions and deadlock.

Dax Toolkit Features

Algorithms are expressed as serial functions. Thus, a developer becomes more efficient with the Dax Toolkit by focusing on the details of the algorithm rather than the intricacies of the parallel system.

By applying different Dax executives, a single algorithm implementation can be adapted to multiple execution environments such as a serial loop (for debugging purposes), on multiple CPU cores, or a GPU-type accelerator.

An executive of the Dax Toolkit can chain multiple functions together within a single iteration of the data. Consequently, an entire chain of operations can be performed for a single memory read/write. Such execution behavior maximizes the instruction-to-memory-fetch ratio. In comparison, each filter in a traditional visualization pipeline must independently iterate over an entire data set.



Using the Dax Toolkit

The Dax Toolkit provides a rich API that a developer can use when writing functions. This C-based API makes it possible to port to different devices including GPUs. The C-API provides accessors to mesh geometry and topology. By providing abstractions, the Dax API keeps the user code isolated of platform related dependencies.

```
// A functor that processes attribute arrays e.g. array calculator.
__functor__ void UnaryCalculator(
  const daxWork work, const daxArray* input, daxArray* output)
{
  float in_value = daxGetArrayValue(work, input);
  float result = <operation>(in_value);
  daxSetArrayValue(work, output);
}


// Functor that computes cell-scalars based on point-scalars.
__functor__ void CellAverage(
  const daxWork work,
  const daxArray* __positions__ in_positions,
  const daxArray* __and__(__connections__, __ref__(in_positions)) in_connections,
  const daxArray* __dep__(in_positions) inputArray,
  daxArray* __dep__(in_connections) outputArray)
{
  // Get the connected-components using the connections array.
  daxConnectedComponent cell;
  daxGetConnectedComponent(work, in_connections, &cell);

  float sum_value = 0.0;
  for (int cc=0; cc < daxGetNumberOfElements(&cell); cc++)
    {
    // Generate a "work" for the point of interest.
    daxWork point_work;
    daxGetWorkForElement(&cell, cc, &point_work);

    sum_value += daxGetArrayValue(point_work, inputArray);
    }
  sum_value /= daxGetNumberOfElements(&cell);
  daxSetArrayValue(work, outputArray, sum_value);
}
```

Algorithm functions are built into modules. Modules can be connected together to form pipelines. Once the pipeline is set up, one can schedule an execution by using the executive.

```
daxElevationModule elevation;
daxCellAverageModule cellAverage;

daxExecutive executive;
executive.Connect(elevation, elevation->GetOutputPort("output"),
   cellAverage, cellAverage->GetInputPort("input_array"));
...
executive.Execute();
```


Acknowledgements

For more information, please visit us at http://daxtoolkit.org.

The Dax Toolkit is funded by a DOE ASCR grant for Scientific Data Management and Analysis at Extreme Scale.

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.