Flexible Analysis Software for Emerging Architectures

Kenneth Moreland*, Brad King†, Robert Maynard†, and Kwan-Liu Ma‡
*Sandia National Laboratories, Albuquerque, NM 87185-1326
†Kitware, Inc., Clifton Park, NY 12065
‡Computer Science Department, University of California at Davis, Davis, CA 95616-8562

***Abstract*—We are on the threshold of a transformative change in the basic architecture of high-performance computing. The use of accelerator processors, characterized by large core counts, shared but asymmetrical memory, and heavy thread loading, is quickly becoming the norm in high performance computing. These accelerators represent significant challenges in updating our existing base of software. An intrinsic problem with this transition is a fundamental programming shift from message passing processes to much more fine thread scheduling with memory sharing. Another problem is the lack of stability in accelerator implementation; processor and compiler technology is currently changing rapidly. In this paper we describe our approach to address these two immediate problems with respect to scientific analysis and visualization algorithms. Our approach to accelerator programming forms the basis of the Dax toolkit, a framework to build data analysis and visualization algorithms applicable to exascale computing.**

I. Introduction

Whereas supercomputers throughout the terascale era were almost unilaterally built from general purpose CPU processors on distributed memory nodes with a message passing interface, in petascale computing we are seeing the emerging use of accelerators to meet the execution and computation requirements of modern leadership-class facilities. This trend was kicked off when the Roadrunner supercomputer, first to achieve a petaFLOP, was built with Cell BE processors [11]. At the time, Roadrunner was an anomaly, but since then many high-performance computers followed this example. Today, over 12% of the Top 500 supercomputers, including the top performing system, Titan, incorporate accelerators, and that number is growing.[1]

These accelerators represent a significant departure from how we most often perform parallel processing. Computing on the previous generation of high performance computers involved partitioning data among distributed memory nodes and running independent processes that pass messages. However, accelerators do not work well with such an approach. Threads on an accelerator may be grouped in SIMD "warps," can have indeterminate scheduling, and may be incapable of direct message passing [17]. Even on processors with more complete and independent cores, taking advantage of shared memory threads can have its advantages [7], [12]. Ultimately, our algorithms must exhibit a more "pervasive parallelism" comprising a marked increase in concurrency and careful data management [1], [9].

[1]According to the 40$^{\text{th}}$ edition of the Top500 list of the world's supercomputers released November 12, 2012. Available from http://www.top500.org.

Another problem facing current research and development is the shifting landscape of the development environment. The Cell BE processors (and associated compiler environment) comprising Roadrunner is already discontinued. Instead, NVIDIA is aggressively pursing leadership in accelerator technology for scientific computation with Intel hot on its heels. Several compiler technologies such as OpenMP, CUDA, Intel Threading Building Blocks, and OpenACC also compete for multi-threaded programming.

Our team is creating the Dax toolkit [14], which seeks to provide a development framework for scientific data analysis and visualization algorithms for the next generation of high-performance computers and beyond. In this paper we document the following features.

- A general approach to data analysis and visualization algorithm development that provides a pervasive parallelism without the complexity of parallel programming.
- An adapter mechanism that encapsulates the change in behavior required to port the toolkit among devices and compilers.
- A concept-enabled mechanism to automatically build parallel scheduling code from signatures using C++ templates.

II. Previous Work

To implement algorithms that are configurable with respect to operations, data structures, and processor idiosyncrasies, Dax relies on well-established techniques of generic programming [15]. Generic programming uses C++ templates to direct the compiler to specialize a particular piece of code to alternate implementations.

To maximize the amount of code that has no parallel dependencies, Dax employs a functor-based execution mode [3]. The intention of this approach is to write a sequential section of code that operates on a small section of data as a functional object, and then schedule this function in parallel independently on large vector components. The technique can be thought of as a generalization of the map and reduce operations in a MapReduce [8] framework.

A toolkit with similar goals of simplifying many-core parallel programming and cross-device porting is Thrust [4]. Thrust is a more general template library that provides a number

of generic parallel algorithms. Thrust provides many of the desired attributes of Dax and is in fact used to implement many of them. What differentiates Dax is the simplification and specialization of its interface. We can provide generic algorithms and classes designed specifically for data analysis and visualization as well as better specialize the data management.

It should be noted that this paper does not cover message passing, distributed memory, or "hybrid" parallelism. Although this is clearly important in high-performance computing, the scope of this paper is only on the shared-memory, many-core parallelism part of this problem. The techniques discussed here can be coupled with existing distributed memory approaches [2], [16] to complete the hybrid parallelism required to run concurrently across an entire machine.

III. Algorithmic Approach

Our basic approach to building algorithms is to build kernels of execution as functors. These functors are designed to operate on a small element of data in a serial and stateless manner. Because this kernel does work on a small amount of data, we call it a *worklet*. Around this concept, we build a system to concurrently schedule these worklets across multiple elements of a vector.

This approach mirrors that of Baker et al. [3]. Both approaches use C++ templating to generically apply functors in parallel to vectors of data. Where our work significantly differs from that of Baker's is in that we are more focused on the computational geometry problems related to scientific visualization and data analysis.

Where Baker provides a simple mapping mechanism onto a vector, our system is designed to provide a variety of parallel scheduling operations. These result in worklet types that get scheduled in different ways. Each worklet type has a different set of capabilities. The current set of worklet types are

Field Map
: The Field Map is functionally equivalent to Baker's functional approach. It applies a worklet operation independently and in parallel to each entry in one or more field arrays.

Cell Map
: The Cell Map is similar to the Field Map in functionally except that it takes the topology of the mesh into consideration. The worklet is applied to each cell in the mesh and has access to any data, including point fields, on that cell. This map enables operations that must interpolate across the cell.

Topology Generator
: The Topology Generator works similarly to the Cell Map with the exception that instead of creating a new field, it creates a new topology (that is, new cells). One of the prerequisites of invoking a Topology Generator is a classification of how many new cells will be generated for each of the input cells. This could be a constant value (which would be typical for a tetrahedralization), or it might be different for each input cell (which would be typical for operations like threshold or contour) and captured in an array.

Point Reduction
: A Point Reduction operation collects values associated with a vertex in the mesh and performs an operation that reduces to a single value. Point Reduction has two primary uses. First, when topology generation creates new vertices, field and other information can be reduced to the new point. Second, a point reduction can gather information about all of its incident cells, which can be used to interpolate fields created with a Cell Map.

A common theme among all of the worklet types is their behavior of applying the same operation across many small elements. This approach is well proven to be an efficient mechanism to drive many compute cores simultaneously.

IV. Device Adapter

As multiple vendors vie to provide accelerator-type processors, a great variance in the computer architecture exists, and we expect to encounter further changes in the near future. Likewise, there exist multiple compiler environments and libraries for these devices. The most popular of these include OpenMP, CUDA, and OpenCL (although the latter does not yet support C++ classes and templates). These compiler technologies also vary from system to system.

Consequently, we require our Dax toolkit to easily port from one system to the next. At a minimum, we require a base language support, and the language we choose to support is C++. The majority of the code in Dax is constrained to the standard C++ language constructs to minimize the specialization from one system to the next.

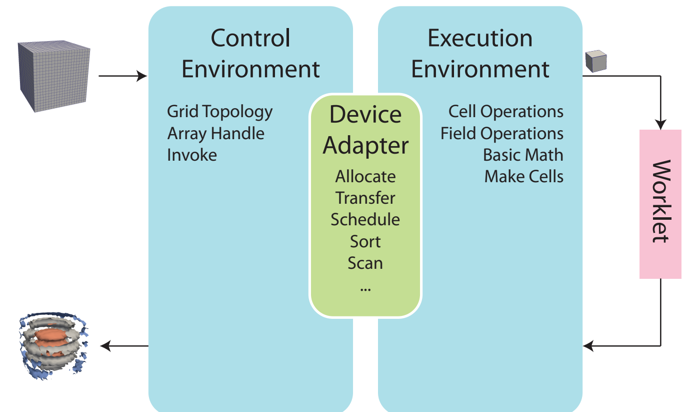


Fig. 1: Diagram of the Dax framework.

Figure 1 provides an overall diagram of the Dax framework. Dax is split into two environments, each with its own API. The *control environment* is used to describe data, interface with other libraries, and invoke parallel operations. The control environment is designed to run in a single thread within a process. Parallel algorithms are run in the *execution environment*. Worklets are built using the execution environment API, which constrains their operations to a safe region of data.

The control and execution environments are logically equivalent to the host and device environments, respectively, in CUDA. When compiling with CUDA, these environments mirror each other, but the same logical approach is taken when no such physical separation exists.

In between these two environments sits the *device adapter*. The device adapter encapsulates all the specialized code required for running on a particular device with a particular compiler technology. The functionality of the device adapter comprises two main parts: a collection of parallel algorithms and a module to transfer data between the control and execution environments.

Each device adapter is expected to implement a set algorithms containing parallel for, scan, lower bounds (parallel find), stream compact, and unique (remove duplicates). This list of operations is similar to those suggested by Blelloch [5] and also a subset of those provided by the Thrust library [4]. Thrust itself provides a convenient implementation for device adapters because it itself is portable among devices. However, the interface to the device adapter algorithms is independent of Thrust, and we have an example of a device adapter that can be built without Thrust.

A device adapter also provides a module to handle the transfer of data between the control and execution environments. Unlike other systems such as CUDA and Thrust, which explicitly define separate arrays and copy between them, the Dax device adapter allocates and copies data in one monolithic operation. The advantage of this approach is that a device adapter for a system that shares memory between the two environments (such as with OpenMP) can perform shallow copies to share the data.

With these basic device adapter facilities, we can build a support library for visualization algorithms. Because the interface for the device adapter is independent of the implementation for each device, this support library can be built in such a way to be portable across many devices.

V. Generic Array Handle

The data model for Dax is deliberately simple. The basic data container in Dax is an *array handle*. The array handle acts like a smart pointer to the data to manage its resource usage. Array handle objects maintain a reference count of how many instances point to the same array.

Array handle objects can also allocate and de-allocate data as necessary. For example, when an array handle is used to store the output of an algorithm, Dax will automatically allocate data in the array to store the appropriate amount of data.

An array handle object manages data in both the control and execution environment. When an array handle is used as input to an algorithm, the array handle automatically copies data to the execution environment. This is done using the data transfer module of the device adapter discussed in Section IV. As described previously, if the control and execution environments can share memory, then this data is not physically copied but rather shared. The array handle also maintains where data resides to avoid unnecessary copies. That is, if data is needed in the execution environment and is already available in the execution environment, no copy will be made. To help applications manage limited memory, the array handle allows applications to free memory either in the execution environment or in both environments.

In addition to adapting to various device memory spaces with the device adapter, the array handle can also adapt to memory layout in the control environment. This is an important technique when applying Dax algorithms to data defined in other library spaces. For example, some systems may define an array of coordinates as a single array with each entry containing 3 coordinates (an array of structures) whereas another might define the same data with three arrays, each containing a single coordinate (a structure of arrays). Rather than copy this data to some canonical structure, the array handle uses generic access to adapt to any layout.

This generic access is achieved through a *container* object. The container provides an encapsulated interface around the data so that any necessary strides or offsets may be handled internally.

One interesting consequence of using a generic container object to manage data within an array handle is that the container can be defined functionally rather than point to data stored in physical memory. Thus, implicit array handles are easily created by adapting to functional containers. For example, the point coordinates of a uniform rectilinear grid are implicit based on the topological position of the point. Thus, the point coordinates for uniform rectilinear grids can be implemented as an implicit array with the same interface as explicit arrays (where unstructured grid points would be stored).

VI. Schedule Metaprograms

Dax aims to relieve its users from the details of scheduling work in the execution environment. We ask the author of a worklet only to specify declarative meta-data describing its interfaces in the control and execution environments. A worklet is defined as a C++ class deriving from the worklet type and providing two meta-data typedefs and a function call operator implementing the functor.

```
1  struct Sine: public dax::exec::WorkletMapField {
2    typedef void ControlSignature(Field(In), Field(Out));
3    typedef _2 ExecutionSignature(_1);
4
5    template <class T>
6    DAX_EXEC_EXPORT T operator()(T v) const {
7      return dax::math::Sin(v);
8    }
9  };
```

Fig. 2: Example Worklet Definition

Figure 2 shows a sample Dax worklet definition with Dax-provided names shown as `Blue` text. The worklet type `WorkletMapField` (line 1) corresponds to the "Field Map" worklet type discussed in Section III.

The `ControlSignature` (line 2) has an entry for each argument one must provide to invoke the worklet from the control environment. Borrowing terminology from the C++ "concepts" proposal by Gregor et al. [10], each entry specifies a *concept* documenting requirements its argument must meet. Dax binds each invocation argument to its corresponding concept using a *concept map* defining how the concrete value type meets the requirements. Tags optionally specified in parentheses after the concept name in a control signature entry are provided to concepts maps to tell them more about how their argument will be used. Our example uses the `Field` concept to specify that arguments must provide an array of values indexable over some domain. The arguments have `In` and `Out` tags to indicate that they will be used as the input and output of the Field Map operation, respectively.

The `ExecutionSignature` entries (line 3) map one-to-one by position to the function call operator signature and specify what Dax should pass to invoke the worklet in the execution environment. The function call `operator()` (line 6) provides the implementation for the execution environment and must be marked with `DAX_EXEC_EXPORT` (like __device__ for CUDA). Each execution signature entry typically references a control signature argument position using a placeholder e.g. `_1` for the first argument and `_2` for the second argument. It also allows for signature entries that represent runtime information, such as current iteration index.

Dax automatically converts argument representations between the control and execution environments and extracts for each scheduled worklet execution argument values local to its operation.

```
1  std::vector<dax::Scalar> input(10);
2  for(std::size_t i=0; i < intput.size(); ++i)
3    { input[i]=1.0f+i; }
4
5  dax::cont::ArrayHandle<dax::Scalar> inputHandle =
6      dax::cont::make_ArrayHandle(input);
7  dax::cont::ArrayHandle<dax::Scalar> sineResult;
8
9  dax::cont::Schedule<> scheduler;
10 scheduler(Sine(), 1.0f, sineResult);
11 scheduler(Sine(), inputHandle, sineResult);
```

Fig. 3: Example Worklet Invocation

`Schedule` (line 9) is a variadic functor which is templated on the execution environment where worklets will be executed. The `operator()` of `Schedule` is the variadic function whose first argument is the worklet followed by the arguments that fulfill the worklet's `ControlSignature`.

When Dax binds each control invocation argument to its corresponding concept it determines the execution range for the argument given the worklet's domain. Combining these ranges determines how we schedule each worklet. Line 10 of Figure 3 shows the invocation of the Sine worklet for a length of one whereas line 11 shows the Sine worklet executing over the length of `inputHandle`.

VII. Results

A common problem introduced when adding layers of abstraction to a programming interface is the reduction in efficiency with the algorithms. To demonstrate that the added overhead of our generic programming is minimal, we perform timing measurements for threshold, a non-trivial algorithm that produces a new topology. Our implementation of threshold produces a compact mesh with unused vertices removed and connectivity information intact.

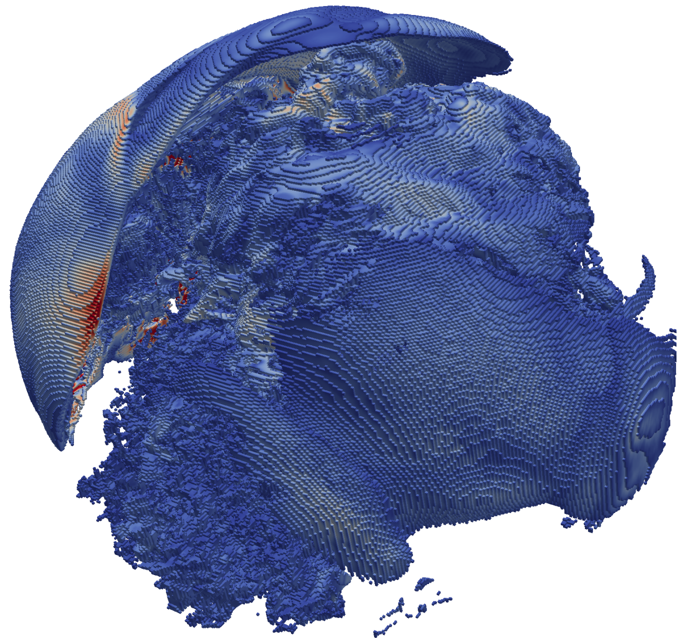


Fig. 4: Supernova dataset used in threshold timing experiments.

Our threshold algorithm in all instances is run on a large regular mesh from a supernova simulation made available by John Blondin at the North Carolina State University and Anthony Mezzacappa of Oak Ridge National Laboratory [6]. The data comprises a uniform grid of 432^3 points with a 32-bit floating point field value associated with each point. The result of our threshold algorithm, shown in Figure 4, contains 3,245,512 cells and 4,090,196 points. All runs are performed on a Mac desktop with dual Quad-Core Intel Xeon processors (8 cores total) and 32 GB of system memory. The system contains an NVIDIA Quadro 4000 with 2 GB of memory on which CUDA tests were run.

As described in Section IV, Dax's device adapter mechanism makes it easy to port the toolkit among different execution environments. We currently implement four device adapters: a serial execution that uses the C++ standard template library's algorithms, an OpenMP execution on multiple CPU cores that uses the Thrust library's algorithms, a CUDA execution on NVIDIA GPUs that also uses Thrust, and an Intel Threading Building Blocks (TBB) enabled execution on multiple CPU cores. We run our Dax threshold algorithm using

all three of these device adapters for comparison purposes. Because our threshold operation in Dax produces results that are isomorphic to those produced by VTK, we also run our algorithm using the VTK filter. The timing results of all these runs are summarized in Figure 5.

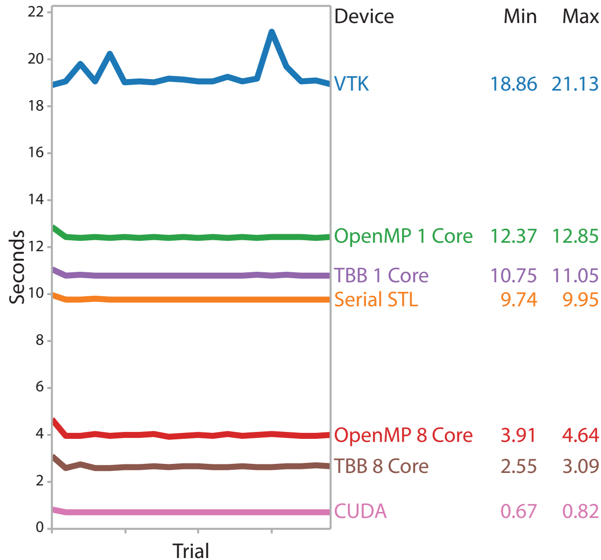


Fig. 5: Timing results for threshold operation.

First we note that even when Dax is running in serial it outperforms the VTK implementation. Because of the large differences between the implementations of these two algorithms as well as the toolkits they are in, we cannot draw too many direct conclusions from this comparison. However, these timings validate the general approach we are talking and indicate that our generic programming is a viable alternative to the direct memory access and polymorphic object behavior implemented in VTK.

Because the serial, OpenMP, and TBB device adapters use different algorithms, we ran the OpenMP and TBB implementations on a single core for comparison purposes. We note that the standard template library implementation is faster than the Thrust implementation. This probably indicates an additional overhead in the parallel algorithms. When the OpenMP and TBB implementations are run using all 8 cores of our system, we see a notable performance improvement. We also note that using CUDA to run the threshold algorithm on the GPU is much faster than any of the multi-core CPU versions. This time includes loading data from the CPU into the GPU and then pulling the results back to the CPU.

In addition to comparing the performance of the various device adapters available within Dax, we also present a comparison to the similar threshold implementation available in the PISTON library [13]. Like Dax, PISTON provides visualization algorithms that can be run on many threads and can be ported between architectures like multi-core CPU and GPU. Furthermore, both Dax and PISTON use similar algorithms to achieve this. However, whereas PISTON simply implements algorithms on top of the Thrust library, Dax implements the added layers of worklets, device adapters, array handles, and scheduling metaprograms. Thus, a comparison of these two implementations is a good indication of the overhead incurred from these added layers in Dax. Furthermore, Lo et al. [13] provide evidence that PISTON has comparable performance to tuned code in other libraries such as parallel VTK and NVIDIA SDK, which makes PISTON a good representation for the state of the art. The results for this experiment are summarized in Figure 6

Although our comparison tests for Dax and PISTON are run using the same data set and hardware as before, the details of the algorithm are slightly different to facilitate a better comparison. Whereas Dax provides an output isomorphic to VTK in the previous example, the PISTON implementation generates a somewhat different output. The first difference is that PISTON does not attempt to find duplicate or remove unused points in the output mesh. Dax has the ability to either perform this point resolution or skip it, so although point resolution is included in Figure 5, it is not included in Figure 6. Also, the PISTON threshold implementation is optimized to send its output directly to OpenGL for rendering. Thus, it adds an extra step of removing any interior cells that cannot be "seen" and providing arrays containing vertex and normal information for the quadrilateral faces of the cells. These vertex and normal arrays tend to be much larger than the list of hexahedra indices provided by Dax. So although we providing timing for the original algorithm implemented in PISTON, we also provide timing for a slightly modified PISTON algorithm that outputs the same hexahedra indices as Dax.

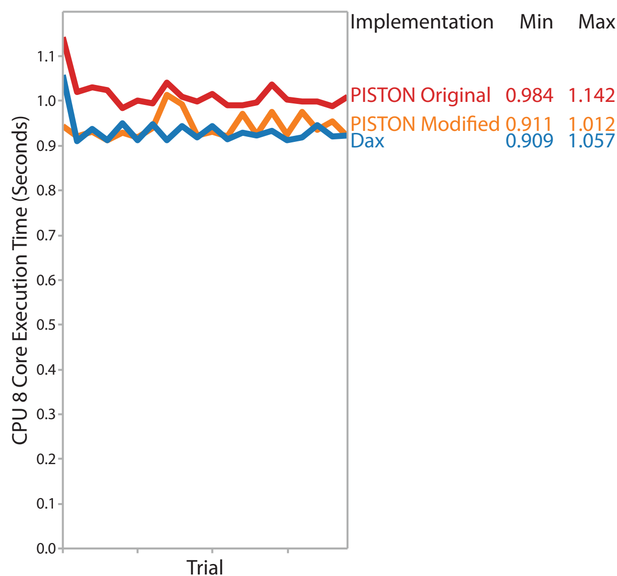
Examining the results in Figure 6, we note that the original PISTON is slower than the others by a small margin; however, this is principally from the fact that this version of threshold is generating larger arrays that are better suited for rendering. A more interesting comparison is between Dax and the modified version of PISTON. Our timings for OpenMP, shown in the left side of Figure 6, are roughly equivalent for the two algorithms. Our timings on a GPU, shown in the right side of Figure 6, show a marginal improvement with the modified PISTON, but otherwise very similar performance.

VIII. Conclusion

Recent advances in computer architecture represent many opportunities and challenges for scientific visualization as well as many other fields in high-performance computing. Accelerators represent a low-cost, low-power mechanism to achieve high computation rates.

Because of the diversity of accelerator architectures available, a project must do better than excel at any one specific system to be successful; it must adapt itself to a changing landscape of computer architectures.

One of the goals of our Dax project is to provide the flexibility to adjust to the idiosyncrasies of various processor technologies that might be available. We have demonstrated that it is possible through generic programming to adapt to a variety of programming environments with little overhead.



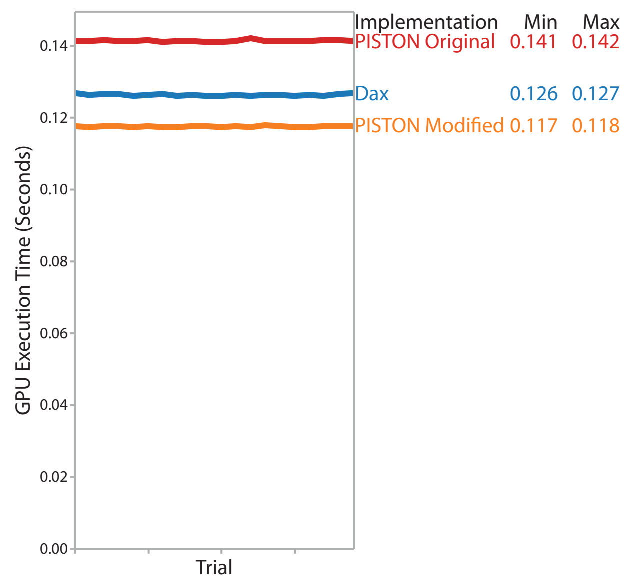


Fig. 6: Comparison of Dax performance to PISTON performance.

Acknowledgments

This work was supported in whole by the DOE Office of Science, Advanced Scientific Computing Research, under award number 10-014707, program manager Lucy Nowell.

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration.

SAND 2012-8450C

References

[1] S. Ahern, A. Shoshani, K.-L. Ma *et al.*, "Scientific discovery at the exascale," Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization, February 2011.

[2] J. Ahrens, C. Law, W. Schroeder, K. Martin, and M. Papka, "A parallel approach for efficiently visualizing extremely large, time-varying datasets," Los Alamos National Laboratory, Tech. Rep. #LAUR-00-1620, 2000.

[3] C. G. Baker, M. A. Heroux, H. C. Edwards, and A. B. Williams, "A light-weight API for portable multicore programming," in *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, February 2010, pp. 601 – 606, DOI 10.1109/PDP.2010.49. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5452412

[4] N. Bell and J. Hoberock, *GPU Computing Gems, Jade Edition*. Morgan Kaufmann, October 2011, ch. Thrust: A Productivity-Oriented Library for CUDA, pp. 359–371.

[5] G. E. Blelloch, *Vector Models for Data-Parallel Computing*. MIT Press, 1990, ISBN 0-262-02313-X.

[6] J. M. Blondin, A. Mezzacappa, and C. DeMarino, "Stability of standing accretion shocks, with an eye toward core-collapse supernovae," *The Astrophysical Journal*, vol. 584, no. 2, pp. 971–980, February 2003, DOI 10.1086/345812. [Online]. Available: http://iopscience.iop.org/0004-637X/584/2/971/

[7] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. Joy, "Streamline integration using MPI-hybrid parallelism on large multi-core architecture," *IEEE Transactions on Visualization and Computer Graphics*, December 2010, DOI 10.1109/TVCG.2010.259.

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, January 2008.

[9] J. Dongarra, P. Beechman *et al.*, "The international exascale software project roadmap," University of Tennessee, Tech. Rep. ut-cs-10-652, January 2010. [Online]. Available: http://www.cs.utk.edu/~library/TechReports/2010/ut-cs-10-652.pdf

[10] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine, "Concepts: linguistic support for generic programming in c++," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '06, vol. 41, no. 10, October 2006, pp. 291–310. [Online]. Available: http://doi.acm.org/10.1145/1167473.1167499

[11] P. Henning and A. B. White Jr., "Trailblazing with Roadrunner," *Computing in Science & Engineering*, pp. 91–95, July/August 2009.

[12] M. Howison, E. W. Bethel, and H. Childs, "Hybrid parallelism for volume rendering on large-, multi- and many-core systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 1, pp. 17–29, January 2011, DOI 10.1109/TVCG.2011.24.

[13] L.-T. Lo, C. Sewell, and J. Ahrens, "PISTON: A portable cross-platform framework for data-parallel visualization operators," Los Alamos National Laboratory, Tech. Rep. LA-UR-12-10227, 2012. [Online]. Available: http://viz.lanl.gov/projects/PISTON.html

[14] K. Moreland, U. Ayachit, B. Geveci, and K.-L. Ma, "Dax toolkit: A proposed framework for data analysis and visualization at extreme scale," in *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, October 2011, pp. 97–104, DOI 10.1109/LDAV.2011.6092323.

[15] D. R. Musser and A. Saini, *STL Tutorial and Reference Guide*. Addison Wesley, 1996, ISBN 0-201-63398-1.

[16] T. Peterka, R. Ross, W. Kendall, A. Gyulassy, V. Pascucci, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri, "Scalable parallel building blocks for custom data analysis," in *Proceedings of Large Data Analysis and Visualization Symposium LDAV'11*, October 2011, pp. 105–112, DOI 10.1109/LDAV.2011.6092324. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6092324

[17] J. Sanders and E. Kandrot, *CUDA by Example*. Addison Wesley, 2011, ISBN 978-0-13-138768-3.