Oh, $#*@! Exascale! The Effect of Emerging Architectures on Scientific Discovery

Kenneth Moreland*
*Sandia National Laboratories, Albuquerque, NM 87185-1326

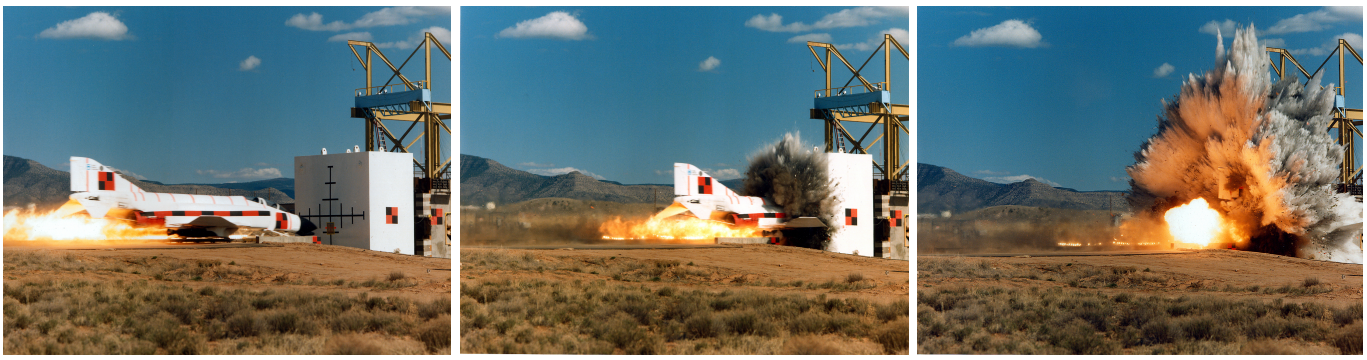

Fig. 1: This 1988 rocket-sled test has nothing to do with exascale computing per se, but it makes for an effective metaphor for the "brick wall" we anticipate our high-performance computing code to collide with.

***Abstract*—The predictions for exascale computing are dire. Although we have benefited from a consistent supercomputer architecture design, even across manufacturers, for well over a decade, recent trends indicate that future high-performance computers will have different hardware structure and programming models to which software must adapt. This paper provides an informal discussion on the ways in which changes in high-performance computing architecture will profoundly affect the scalability of our current generation of scientific visualization and analysis codes and how we must adapt our applications, workflows, and attitudes to continue our success at exascale computing.**

I. Introduction

Thanks to recent advances in parallel visualization algorithms and frameworks, we currently benefit from production large-scale scientific-visualization applications such as ParaView [4] and VisIt [18], which are shown to be very scalable up to our current generation of petascale computers [11]. Although the development of these tools comes from the successful scaling of techniques originally pioneered over ten years ago [2], [36], recent trends indicate that future high-performance computers will have different hardware structure and programming models to which software must adapt.

These grave predictions come from multiple workshops [12], [29], [33] where experts convened to discuss the roadmap to building exascale computers (that is, computers capable of performing 10^{18} floating point operations per second). Table I gives a summary comparison between an existing petaFLOP computer and the expected system performance of a future computer capable of an exaFLOP. Note that there is uncertainty in what type of computer architecture will be used to achieve an exaFLOP (and it is possible different architectures may be used in different instances). To compensate for this uncertainty, experts use the term design "swim lanes" that capture these different approaches. Two likely swim lanes are captured in Table I.

If all the components of supercomputers were to be scaled uniformly, then the "Factor Change" column in Table I would have uniform values. However, the factor of change varies wildly from very small changes to five orders of magnitude. The main cause for most of this variability comes from compromises made to achieve the desired system peak (the first column of Table I) with the constraints of a limited power budget (the second column of Table I). DOE has established a power budget for the exascale machine at 20 MW [33]. The reason for this budget is very pragmatic. The operating cost of a supercomputer is roughly $1 million per megawatt per year. As such, DOE determines that it cannot afford more than $20 million per year to operate a single supercomputer. Thus, the challenge of exascale computing is getting about three orders of magnitude improvement in computation rate using not much more power than we do today.

Using the predictions in Table I, this paper provides an overview of the changes required to our scalable code and how we use our applications. In particular, we will note the contrast between the factor change of related system parameters and discuss the implications. This paper provides an informal discussion of these issues. For a more formal discussion, consult the report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization [1].

II. Concurrency

Let us first compare the amount of memory expected on our exascale machine with the amount of concurrency programs

TABLE I: Comparison of a petascale supercomputer to an expected exascale supercomputer [1].

System Parameter	Petascale	Exascale (Prediction) Swim Lane 1	Swim Lane 2	Factor Change
System Peak	2 Pf/s	1 Ef/s		500
Power	6 MW	≤20 MW		3
System Memory	0.3 PB	32–64 PB		100–200
Total Concurrency	225K	1B×10	1B×100	40,000–400,000
Node Performance	125 GF	1 TF	10 TF	8–80
Node Core Count	12	1,000	10,000	83–830
Network BW	1.5 GB/s	100 GB/s	1000 GB/s	66–660
System Size (nodes)	18700	1,000,000	100,000	50–500
I/O Capacity	15 PB	300–1000 PB		20–67
I/O BW	0.2 TB/s	20–60 TB/s		10–30

will have to exhibit at scale. The system memory of the machine will grow by a factor of about a hundred, which is an order of magnitude lower than than the growth of the computational power.

The reason for this low growth of memory is that memory tends to be power hungry. Thus, we simply cannot afford to grow the amount of memory in the system at a rate equal to the computation. Furthermore, additional memory does not directly contribute to the computational rate of the system. Of course, by that logic we could create an even cheaper exascale machine by not adding or perhaps even removing some system memory. However, doing so would render the machine useless, and neither DOE nor any other organization is willing to spend millions of dollars on a useless machine. So, the amount of memory to be added to an exascale machine is an engineering decision to maximize the utility of the system while keeping everything in budget.

In contrast, the total concurrency required by the system will grow by up to five orders of magnitude. The reason for this staggering increase in concurrency is twofold. First, although Moore's law still holds for the scaling of the number of transistors (so far), this scaling is no longer contributing to the faster operation of processing cores [12]. Instead, increased computing power is achieved by adding more cores to a processor. Given this fixed calculation rate per core, we can predict that we will need roughly a billion cores to sustain a rate of 10^{18} floating point operations per second.

Of course, the change in Moore's law only accounts for an increase of three orders of magnitude. The second factor that is increasing total concurrency is the interface between the processor and the memory holding the data it operates on. Because the latency to off-chip memory is not expected to improve substantially, practical applications will likely have to run 10 to 100 threads per core (depending on the type of processor) to hide this latency by swapping threads during memory fetches [33]. Consequently, a program could require up to *100 billion concurrent threads* to maintain an exaFLOP.

A. Scalability of Current Applications

This combination of low memory growth and high concurrency growth represents serious scaling issues for our parallel high-performance computation code. Our current production scientific-visualization applications use a distributed memory model with a message passing interface, embodied in the use of MPI [32], to implement concurrent processing and communication. Although this simple but effective model has served well to the petascale era of supercomputing, it fails to capture important emerging features in high-performance computing, and it is thus doomed to break down unless major restructuring efforts are enacted.

To understand why our parallel production applications will fail to scale, let us consider a simple and artificial but realistic and representative example of performing scientific visualization on a regular grid of cells. For a capability run on the petascale machine represented in Table I we could expect a grid of 1 trillion cells [11]. On the exascale machine, having about 100 times more memory, we could expect a grid of 100 trillion cells.

Our current production scientific-visualization applications use MPI to represent all concurrency, and MPI uses operating system processes to represent concurrent execution. Thus, to run on the entirety of a petascale machine, an application would need about 200 thousand processes whereas an exascale machine could require up to 100 billion processes. One problem with an operating system process is that it is a heavy weight object. Associated with each one is an entire program state including a copy of the machine instructions to be executed. A library of general-purpose scientific visualization algorithms could be expected to run at least 20 MB, and these 20 MB would have to be replicated on each process. Replicating this data 200 thousand times on the petascale machine yields 4 TB, which is reasonable at less than 2% of the overall memory in the system. However, replicating this data 100 billion times on the exascale machine yields 2 EB, well over the amount of memory available on the entire machine (and, incidentally, all of its storage). Thus, we cannot even start our application on the exascale machine, and that is before we even load any data.

But let us say we get around that problem, which is an active area of research [6]. Another problem that we run into is the breakdown of Gustafson's law. Scientific visualization code, as well as most modern high-performance parallel code, gets around the limits of parallel efficiency proposed by Amdahl [3] by scaling up the problem size along with the concurrency [15]. By considering a scaled speedup, we can

allow the problem size to be an increasing function of the number of processes [27]. However, limiting the growth of memory limits the scale to which we can grow problems.

Back to our example, assuming that our visualization algorithm uses data parallelism, which most modern visualization algorithms do [22], our 1 trillion cell petascale problem will be broken into partitions of 5 million cells for each thread. Experience shows 5 million cells per thread to be an efficient amount to drive a visualization pipeline thread [21]. In contrast, the equivalent exascale mesh of 100 trillion cells would be divided into partitions of 1000 cells, a ridiculously small and inefficient amount to overcome the overhead of a parallel program.

But even ignoring this problem, others abound. Consider the issue of adding ghost cells (or sometimes called halo cells) to our partitions. Ghost cells are critical to the operation of our current parallel scientific visualization algorithms; they limit the communication required in the algorithms to make the parallel overhead manageable. Assuming our petascale problem is broken into 5 million cell partitions that are roughly 171^3 blocks, each block would require 6×171^2 or about 175 thousand ghost cells. All total, this is 35 billion ghost cells, which amounts to about 3.5% of the original data size. In contrast, our exascale mesh would be broken into 1000 cell partitions of 10^3 blocks, and each block would require $6 \times 10^2 = 600$ ghost cells. All total, this is 60 trillion ghost cells, which amounts to about 60% of the original data size. Growing the memory overhead by 60% is simply not feasible in most serious applications.

B. Exascale Programming Challenges

These problems and many others plague our efforts to achieve scientific discovery at exascale. Simply put, at some point our current approach of domain decomposition fails. Eventually we require too many ghost cells, too much communication, or simply have too fine of partitions to be efficient. What this means is that a significant portion of our code needs to be redesigned and reimplemented.

In addition to revisiting and reengineering our scientific visualization code, we must also cope with new, emerging, and conflicting architectures and programming models. The most blatant challenge is the proliferation of compiler and libraries used to program these new multithreaded processors. Currently popular technologies include OpenMP, CUDA, OpenCL, Intel Threading Building Blocks, and OpenACC with many others also proposed and available. None of these solutions is universally available for all hardware and all development environments, nor are any likely to become a universal solution. Thus, if a developer wants broad access to his or her application, the code must be ported across significantly different programming environments.

Furthermore, different hardware environments have their own idiosyncrasies that must be taken into account above any basic compiler porting issues. For example, a typical multicore CPU processor has a cache structure optimized to provide independent memory access to each core. This means that memory must be divided along cache lines among cores to prevent inefficiencies associated with issues such as false sharing [28]. In contrast, a GPU accelerator typically has cores grouped together in blocks in which threads group memory fetches and share cache [30]. This differing organization of threads requires equally different organization of scheduling and memory access within a program.

Eventually, scientific-visualization code must be reengineered using to use an efficient shared-memory execution model (or more specifically, allow a hybrid shared-distributed execution [10]). Although we have been successful at implementing efficient, scalable distributed-memory algorithms, writing algorithms in this threaded shared-memory environment poses many challenges. It is a common misconception that threaded shared-memory programming is easier because it does not require the data partitioning and message passing typically required for distributed-memory programming. This, however, is untrue because although distributed-memory programming has a larger up-front cost of determining memory management and communication, this design process inherently leads us to efficiently structured code. Proper partitioning is as important in shared memory as in distributed memory, but threaded programming generally does not provide the programmatic constraints to simplify and control the partitioning. Furthermore, distributed memory models' isolation of memory spaces helps prevent read-write collisions and deadlocks whereas any errant memory access anywhere in a threaded program can lead to incorrect code that is difficult to debug.

Research is already underway to advance the state of the art in scientific visualization to finely threaded algorithms. The most successful approaches in terms of ease of programming, portability, and maintainability isolate the parts of code responsible for parallel scheduling, communication, and critical sections using techniques like functors [5] and using basic parallel algorithms as building blocks [9].

C. Emerging Frameworks for Scientific Discovery at Exascale

DOE is currently addressing many of the programming challenges for exascale through funding of various projects addressing different aspects of the problem. The first project, PISTON [19], facilitates the development of visualization and analysis operators with highly portable performance. As noted previously, there is a wide variety in the hardware and development environments for multithreaded environments, and PISTON provides mechanisms both to compile across these development environments and to run efficiently on many different devices. It does this using a data parallelism model and basic parallel operations much like that proposed by Blelloch [9].

The second project, Dax [23], simplifies the development of parallel visualization algorithms. It does so by allowing developers to build algorithms out of worklets, which are functions that implement an algorithm's behavior on an element of a mesh or a small local neighborhood. Worklets are designed in serial but can be concurrently scheduled on an unlimited num-

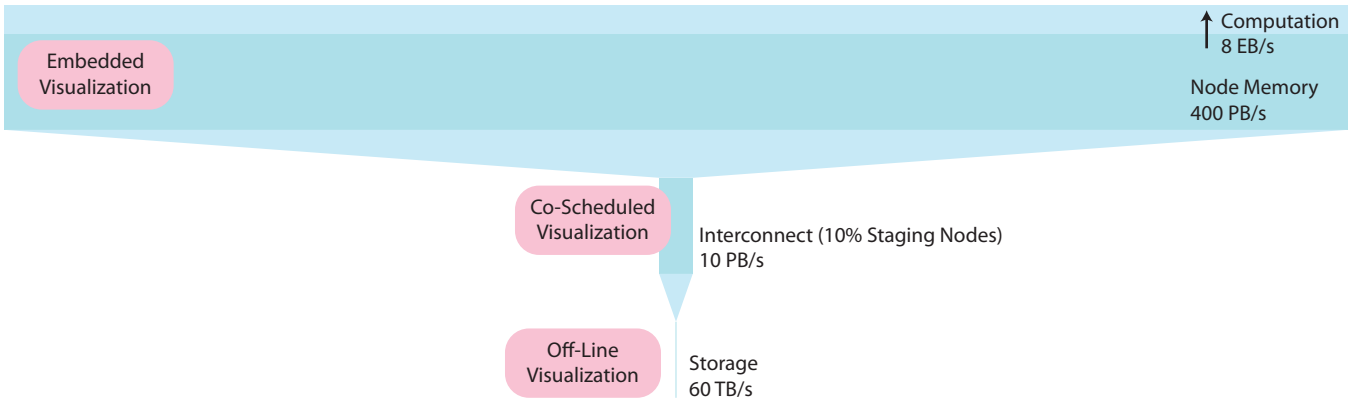

Fig. 2: Visual depiction of the relative bandwidth of exascale system components [1].

ber of threads without the complications of memory clashes or other race conditions. Dax also provides many of the basic topology structures and common operations on them to better facilitate algorithm development. It also allows algorithms to adapt to memory structures of external applications without having to copy to a different data structure [24].

The third project, EAVL [20], updates the traditional data model for modern simulation codes and investigates how the updated model can achieve computational and memory efficiency. EAVL defines more flexible mesh structures, which more efficiently support many non-traditional types of data such as for graphs, mixed data types, high-order fields, and adaptive meshes.

See Sewell et al. [31] for a more in depth description of these and other exascale-related projects. The good news from these projects is that they provide good introductory steps to implementing high-performance visualization at exascale. If successful, these changes should produce few negative or even noticeable changes to the abilities of production visualization software.

III. Recording Results

Let us consider a different aspect of the exascale machine. Consider the change in the bandwidth to and from the storage system (bottom row of Table I). Although the peak performance of the computer should rise by three orders of magnitude, the bandwidth to the storage only increases by one order of magnitude. The consequence is that a smaller fraction of results can be recorded when a simulation is run.

This is not a new trend in supercomputing. For the last 20 years one could expect each generation of supercomputer to have more floating point operations per second and more concurrency, but comparatively less storage and I/O bandwidth. That is not to say that the storage systems have not been improving; they have just been improving at a much slower rate than the rest of the system. As an example, two generations ago ASCI Purple, a 100 teraFLOP machine, had a peak bandwidth of 140 GB/s to its parallel storage. One generation ago, JaguarPF, a petaFLOP machine, had a peak bandwidth of 200 GB/s to its parallel storage, which is not a dramatic improvement. To exacerbate the issue, few real applications are capable of achieving anywhere near peak performance, and all applications must sometimes deal with file system contention with jobs both on and off the computer.

Thus, as simulations are scaled up on larger machines and perform calculations at faster rates, there comes a time when it is impractical to write results at a fast enough rate to do a proper analysis. When this time comes, we must become smarter about what gets written out and we must move analysis closer to the data because there is no point in running a simulation in the first place if we do not get the proper analysis out of it.

A. Relative Data Bandwidth

Figure 2 provides a Minard plot depicting the bandwidths of different I/O components by the proportional width of their respective blue bars. (See Tufte [34] for a longer description of Minard plots and their merits.) Not shown on this plot is the rate at which data can be computed. At 10^{18} double precision floating point operations per second, the exascale computer on aggregate will produce 8 EB per second. On this plot that would be almost 4 yards across. The aggregate bandwidth of the local memory — that is, the speed at which data can be pulled from memory to a local process summed over the entire machine — is 400 PB/s. This is as fast as we can reasonably expect to access data on the system, but it is only available in the same job space as the running simulation. If we were to offload that data to another job, say a staging job running on a generous allocation of 10% of the overall nodes, then we could stream the data to this job at 10 PB/s, which is pretty fast but only 2.5% the local access rate. Furthermore, one must worry about the limited memory available on the staging nodes. To move the data entirely to disk storage, the bandwidth drops way down to 60 TB/s. This is only 0.015% the rate at which data can be written to local memory and only 0.00075% the rate at which data can be computed. So, only an extremely small fraction of data can ever hope to be captured on disk.

The problem, of course, is that the majority of visualization and analysis done today is performed "off-line" after the data has been written to disk. Off-line visualization offers

TABLE II: List of visualization solutions with respect to coupling with a running simulation and their respective properties.

	Capability	Coupling	Footprint	Transfer	Interactive
Tightly Integrated	Low	Tight	Low	None	No
Embedded	High	Tight	High	Possible memcopy	No
Hybrid	High	Tight	Medium	Subset High-Speed Transfer	Yes
Co-Scheduled	High	Loose	Extra Nodes	High-Speed Transfer	Yes
Off-Line	High	Loose	None	Slow Persistent Storage Cost	Yes

many advantages that make it the easiest way to perform visualization. First, it makes the interface between simulation and visualization simple. The visualization needs only to understand the format used when the simulation writes the data to disk. Second, it makes scheduling and performing the visualization, particularly when done by a human, much easier. The visualization job can be scheduled completely independently of the simulation and at a time most convenient to the user. Third, the results data are placed in persistent storage, so any analysis not performed immediately can be done at a later time if they are later deemed necessary.

None of these advantages mean anything, however, if it is not possible to get the necessary data to the storage system. In the case where there is not sufficient bandwidth to store raw results from the simulation, then the visualization must be moved "closer" to the visualization (upwards in the Minard plot of Figure 2). You could co-schedule a visualization job at the same time as the simulation job and set up a transfer mechanism over a high speed network. If that still is not enough bandwidth, you could embed the visualization directly in the simulation so that the visualization has direct access to the data while it is still in local memory.

B. Gamut of Coinciding Visualization Solutions

Given the fact that the bandwidth to results data increases as we move closer to the visualization, it is reasonable to ask why we would not always choose to embed directly with the simulation. Although this would certainly optimize the bandwidth to the data, moving visualization closer to simulation comes at a cost.

The first cost is that of complexity in development. Creating a visualization that can be co-scheduled requires a complex connection between the simulation and visualization and generally requires a fully featured I/O system to manage it. The co-scheduled visualization also requires added complexity in the job scheduler of the parallel computer. Embedding a visualization in a simulation requires a complete integration of simulation and visualization codes. In practical terms, it means that both development teams must work together, which tends to involve crossing several cultural barriers.

The second cost is that of loss in functionality. Once we move our visualization away from off-line, the data we work with becomes transient. A co-scheduled visualization has access only to data that is locally available, which may be no more than one snapshot in time. Once the visualization moves on to another snapshot in time, or if the simulation passes a snapshot in time before the visualization is ready, that data is lost forever. An embedded visualization has the further constraint that whatever operations are to be performed, they have to already be declared when the simulation is ready to invoke them.

Consequently, there is no simple single solution that can address all of our current or projected visualization coupling needs. Our group has experimented with an entire spectrum of solutions and found each one to have its own advantages and disadvantages as summarized in Table II.

On one end of the spectrum we can have a very tightly integrated visualization component built from the ground up for a specific need in a simulation. Such a targeted solution can have a very small footprint as it can make specialized optimizations for the particular data types and representations used. Such a solution can be perfect for an analysis team that wants a specific, targeted, and fairly simple functionality. However, in our experience a successful integration usually begets further requests for new features, which would be readily available in a more general-purpose library. Hence, the tightly integrated approach often spirals into a large duplication of efforts between projects, and we try to avoid it.

A preferred method, from the visualization software developer's standpoint, is to embed a general-purpose visualization library into a simulation [14], [35]. Once integrated, the simulation now has access to all the visualization capabilities of the visualization library as well as possible integration with already familiar visualization tools. The disadvantage of such an approach is that the footprint of such a library tends to be high, especially when dynamic selection of visualization capabilities is supported. Also, it means that any visualization algorithm used must scale to the same job size as the simulation it is embedded in, which can be challenging [13] although also useful outside the simulation.

Co-scheduled visualization (also often called *in transit* visualization) alleviates some of the problems of embedded *in situ* [8], [17], [25]. The coupling between visualization and simulation can be simplified by connecting them indirectly through an I/O layer although general-purpose I/O layers are still in development. Also, the co-scheduled visualization job can provide an entire suite of visualization tools without directly adding to the footprint of the simulation although it does require a sufficiently large separate allocation of nodes on the same computer or nearby. A disadvantage of the co-scheduled method is that an overhead is incurred for transferring the data from simulation job to visualization job, but once that is done the two processes can execute asynchronously, further

eliminating some of the overhead in the simulation.

In some circumstances we have found a hybrid method between embedded and co-scheduled visualization is useful. In this method, a visualization component is directly embedded in the simulation, but the visualization can also connect and transfer data to a separate and smaller visualization job [25]. The intention here is to provide a tightly coupled and highly scalable algorithm directly in the simulation code that can quickly extract salient information and pass this reduced data to another process. This second process can then be used to serve an interactive visualization to a user without blocking the large simulation.

IV. Other Exascale Challenges

So far we have considered some of the most urgent problems facing us for exascale computing and the ones we are directly working on at Sandia National Laboratories. There are, however, many more challenges that face us as we approach exascale, and this section gives a brief overview of some other important issues.

A. Resilience

Although our production scientific-visualization tools are designed to provide reliable operation, resilience has not been the primary concern for their development. Should a problem occur during visualization, such as a hardware failure, that causes a catastrophic interruption, it is of relatively little consequence to simply restart the visualization and reload the data when running in off-line mode. However, things can change dramatically for exascale. When running *in situ* at the exascale, it is vital that the visualization components be robust. It is not looked upon favorably when visualization brings down a large simulation job.

At exascale we expect failures to occur more frequently as well. The exascale computer comprises significantly more hardware components. More components means a greater chance that one will fail, which means that the mean time to failure goes down. When the mean time to failure drops low enough, even independent visualization jobs need to consider resiliency.

In addition to more likely catastrophic failures, high-performance computing may need to be adept at handling soft errors, which are errors such as small data corruption that are never detected. Soft errors are more likely at the exascale not only due to the greater number of components but also because some error detecting hardware may be removed in an effort to reduce the power consumed.

Ultimately, the operation of an exascale computer may be less deterministic than what we currently expect of a petascale computer. Not only must our code be able to operate under uncertainty, but it must also be able to characterize the uncertainty in the data that it produces.

B. Compression and Extraction

As described in Section III, less results data can be captured from a simulation. Ultimately, this means that we must make better use of the bandwidth that we do have available by providing more information with less data. One straightforward way to do this is to compress the data. Data compression has a long and fruitful history, but scientific analysis and visualization provides several new challenges. Scientific data tends to be of floating point numbers with well defined but possibly complicated and irregular structure, which is different from most other uses of compression. Furthermore, we could benefit from both lossless and lossy compression, but for lossy compression it is very important to be able to both constrain and characterize the inaccuracy introduced by lossy compression.

Another approach to reducing the amount of data returned is by extracting exactly the information desired. One direct method of extraction is to perform a feature characterization and write out properties of features. The challenge of this approach is that feature characterization is very specific to a particular scientific domain and analysis. Furthermore, features are often difficult to specify let alone extract, and often feature identification is an expensive operation.

C. Provenance

Provenance provides a record of what operations and parameters produced a given set of data. Recent works of provenance in scientific visualization systems record the set of parameters used to build a visualization and the history of exploration to get there [7], [16].

With the added focus of *in situ* visualization at the exascale, provenance takes on a new and important role. Because data is transient, it becomes impossible to revisit old data and verify old visualization results. To maintain confidence in the data analysis without the original data, it is important to capture what methods were used to draw any set of conclusions.

D. Uncertainty Quantification

As the computational power of supercomputers increases, we expect greater efforts to be made in tracking uncertainty and quantifying the uncertainty in simulations. However, this uncertainty quantification is of little use if the analysis can provide no insight into the envelope of possible values. Visualization systems need to incorporate uncertainty in their visual representation of data [26].

V. Final Remarks

The transition from petascale to exascale computing is an exciting time for all aspects of high-performance computing. A fundamental shift in the basic computer architecture of our machines presents a great many challenges for scientific visualization as well as all other large-scale applications, but it also presents us with numerous opportunities as well. Exascale promises major advances in computing ability, which, if properly harnessed, can provide major advances in science.

One interesting upshot of the exascale challenge is that it brings researchers from different disciplines together. Because the engineering decisions for the design of every part of the

system from the hardware to the application are so interconnected, the study of exascale has lead to several codesign centers that share system problems and solutions to mutually design the entire system. More pervasively, we find that moving to exascale requires us to transition from a technology-driven science, where one focuses on the computational tools, to a discovery-driven science. Discovery-driven science dictates that the primary focus of a scientific endeavor be the end goal and the scientific discoveries we need to make. With it, we holistically consider the entire computational experiment design rather than a series of mutually exclusive steps.

ACKNOWLEDGMENTS

This report is a summary of ongoing work by a great number collaborators over many institutions. In particular, I would like to thank the following: Nathan Fabian and Ron Oldfield from Sandia National Laboratories; Kwan-Liu Ma, Robert Miller, and Yecong Ye from the University of California at Davis; Berk Geveci, Utkarsh Ayachit, Robert Maynard, Brad King, Andrew C. Bauer, Pat Marion, Sebastien Jourdain, David DeMarle, and David Thompson at Kitware, Inc.; James Ahrens and Jonathan Woodring at Los Alamos National Laboratory; Scott Klasky and Norbert Podhorszki at Oak Ridge National Laboratory; Venkatram Vishwanath, Mark Hereld, and Michael E. Papka at Argonne National Laboratory; Michel Rasquin and Kenneth E. Jansen at the University of Colorado at Boulder; Ciprian Docan and Manish Parashar at Rutgers University.

This work was supported in part by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. 12-015215, through the Scientific Discovery through Advanced Computing (SciDAC) Institute of Scalable Data Management, Analysis and Visualization.

This work was supported in part 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 2013-0180C

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] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the Spring Joint Computer Conference (AFIPS '67)*, April 1967, pp. 483–485, DOI 10.1145/1465482.1465560.

[4] U. Ayachit *et al.*, *The ParaView Guide: A Parallel Visualization Application*, 4th ed. Kitware Inc., 2012, ISBN 978-1-930934-24-5.

[5] 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.

[6] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "Mpi on millions of cores," *Parallel Processing Letters*, vol. 21, no. 1, pp. 45–60, March 2011.

[7] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: Enabling interactive multiple-view visualizations," in *Proceedings of IEEE Visualization*, October 2005, pp. 135–142.

[8] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali, "Parallel computational steering for HPC applications using HDF5 files in distributed shared memory," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 6, pp. 852–864, June 2012, DOI 10.1109/TVCG.2012.63.

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

[10] L. Chen and I. Fujishiro, "Optimization strategies using hybrid MPI+OpenMP parallelization for large-scale data visualization on earth simulator," in *A Practical Programming Model for the Multi-Core Era*. Springer, 2008, vol. 4935, pp. 112–124, DOI 10.1007/978-3-540-69303-1_10.

[11] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. H. Weber, and E. W. Bethel, "Extreme scaling of production visualization software on diverse architectures," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp. 22–31, May/June 2010, DOI 10.1109/MCG.2010.51.

[12] J. Dongarra, P. Beechman *et al.*, "The international exascale software project roadmap," University of Tennessee, Tech. Rep. ut-cs-10-652, January 2010.

[13] N. Fabian, "*In situ* fragment detection at scale," in *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, October 2012, pp. 105–108, DOI 10.1109/LDAV.2012.6378983.

[14] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. E. Jansen, "The ParaView coprocessing library: A scalable, general purpose in situ visualization library," in *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, October 2011, pp. 89–96, DOI 10.1109/LDAV.2011.6092322.

[15] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, May 1988, DOI 10.1145/42411.42415.

[16] T. Jankun-Kelly, K.-L. Ma, and M. Gertz, "A model for the visualization exploration process," in *Proceedings of IEEE Visualization 2002*, October 2002, pp. 323–330.

[17] S. Klasky *et al.*, "In situ data processing for extreme scale computing," in *Proceedings of SciDAC 2011*, July 2011.

[18] *VisIt User's Manual*, Lawrence Livermore National Laboratory, October 2005, technical Report UCRL-SM-220449.

[19] 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.

[20] J. S. Meredith, R. Sisneros, D. Pugmire, and S. Ahern, "A distributed data-parallel framework for analysis and visualization algorithm development," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*, March 2012, pp. 11–19, DOI 10.1145/2159430.2159432.

[21] K. Moreland, "The ParaView tutorial, version 3.12," Sandia National Laboratories, Tech. Rep. SAND 2011-8896P, 2011.

[22] ——, "A survey of visualization pipelines," *IEEE Transactions on Visualization and Computer Graphics*, 2012, (preprint).

[23] 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.

[24] K. Moreland, B. King, R. Maynard, and K.-L. Ma, "Flexible analysis software for emerging architectures," in *Petascale Data Analytics: Challenges and Opportunities (PDAC-12)*, November 2012.

[25] K. Moreland, R. Oldfield, P. Marion, S. Jourdain, N. Podhorszki, V. Vishwanath, N. Fabian, C. Docan, M. Parashar, M. Hereld, M. E. Papka, and S. Klasky, "Examples of *in transit* visualization," in *Petascale*

Data Analytics: Challenges and Opportunities (PDAC-11), November 2011.

[26] K. Potter, P. Rosen, and C. R. Johnson, "From quantification to visualization: A taxonomy of uncertainty visualization approaches," in *Uncertainty Quantification in Scientific Computing*, ser. IFIP Advances in Information and Communication Technology Series, vol. 377, 2012, pp. 226–249.

[27] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004, ISBN 978-0-07-282256-4.

[28] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007, ISBN 978-0-596-51480-8.

[29] M. Richards *et al.*, "Exascale software study: Software challenges in extreme scale systems," DARPA Information Processing Techniques Office (IPTO), Tech. Rep., September 2009.

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

[31] C. Sewell, J. Meredith, K. Moreland, T. Peterka, D. DeMarle, L. ta Lo, J. Ahrens, R. Maynard, and B. Geveci, "The SDAV software frameworks for visualization and analysis on next-generation multi-core and many-core architectures," in *Proceedings of the Ultrascale Visualization Workshop*, November 2012.

[32] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, 2nd ed. MIT Press, 1998, vol. 1, The MPI Core, ISBN 0-262-69215-5.

[33] R. Stevens, A. White *et al.*, "Architectures and technology for extreme scale computing," ASCR Scientific Grand Challenges Workshop Series, Tech. Rep., December 2009.

[34] E. R. Tufte, *The Visual Display of Quantitative Information*, 2nd ed. Graphics Press, 2001, ISBN 0-9613921-4-2.

[35] B. Whitlock, "Getting data into VisIt," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-SM-446033, July 2010.

[36] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland, "Scalable rendering on PC clusters," *IEEE Computer Graphics and Applications*, vol. 21, no. 4, pp. 62–70, July/August 2001.