SANDIA REPORT

SAND2009-2014
Unlimited Release
Printed April 2009

Design Issues for Performing *In Situ* Analysis of Simulation Data

David Thompson, Nathan D. Fabian, Kenneth D. Moreland, Lisa G. Ice

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2009-2014
Unlimited Release
Printed April 2009

Design Issues for Performing *In Situ* Analysis of Simulation Data

David Thompson
Sandia National Laboratories
M.S. 9159, P.O. Box 969
Livermore, CA 94551, U.S.A.
dcthomp@sandia.gov

Nathan D. Fabian
Sandia National Laboratories
M.S. 1323, P. O. Box 5800
Albuquerque, NM 87185-1323, U.S.A
ndfabia@sandia.gov

Kenneth D. Moreland
Sandia National Laboratories
M.S. 1323, P. O. Box 5800
Albuquerque, NM 87185-1323, U.S.A
kmorel@sandia.gov

Lisa G. Ice
Sandia National Laboratories
M.S. 0822, P. O. Box 5800
Albuquerque, NM 87185-0822, U.S.A
lgice@sandia.gov

Abstract

In this report, we present issues associated with performing *in situ* analysis of simulation data. Because high-performance compute platforms are able to increase compute intensity much less expensively than disk bandwidth and because the reliability of components in the platform is relatively constant while the number of components is growing exponentially, it is becoming increasingly desirable to reduce the amount of data written to disk. This impacts the fidelity and usability of visualizations and other analysis produced *ex post facto* using large results files. Integrating analysis into each simulation that must run at scale on a case-by-case basis is not feasible. Instead, our goal is to create a general-purpose framework for *in situ* analyses that can be quickly adapted to simulation codes as required and this report outlines the first step of that process – defining issues that must be addressed by such a framework.

Acknowledgments

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94-AL85000.

Contents

This page intentionally left blank

1 Introduction

In this report, we present issues associated with performing *in situ* analysis (see Figure 1) of simulation data. Because high-performance compute platforms are able to increase compute intensity much less expensively than disk bandwidth and because the reliability of components in the platform is relatively constant while the number of components is growing exponentially, it is becoming increasingly desirable to reduce the frequency of writing data to disk. This impacts the fidelity and usability of visualizations and other analysis produced *ex post facto* using large results files. Integrating analysis into each simulation that must run at scale on a case-by-case basis is not feasible. Instead, our goal is to create a general-purpose framework for *in situ* analyses that can be quickly adapted to simulation codes as required and this report outlines the first step of that process – defining issues that must be addressed by such a framework.

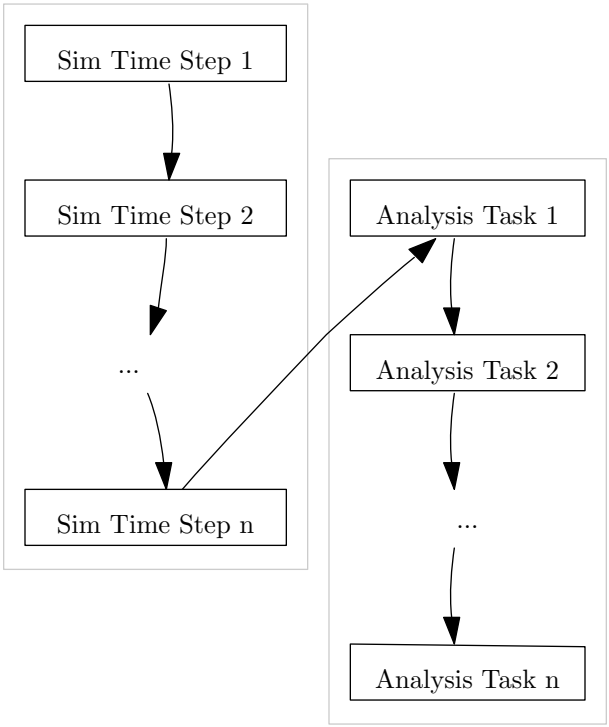
These issues may be broken down into 3 categories:

- hardware architecture,
- interfacing with a simulation, and
- interfacing with users.

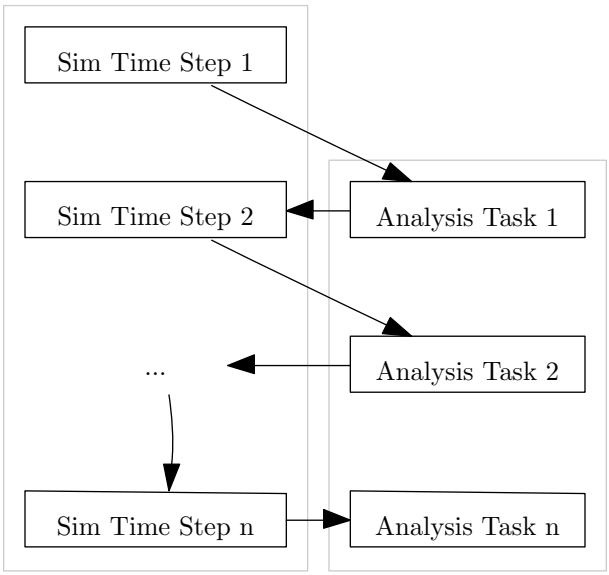
The remainder of the report covers each category in its own section. Each section presents general issues and also addresses particular simulation codes and platforms of interest to Sandia. Simulation codes such as CTH, S3D, and Presto are considered. They are targeted at machines such as Red Storm, Jaguar, Sequoia, and Roadrunner. Because all of these simulations address temporal phenomena by advancing state forward in time using a series of time steps, the rest of the report will discuss analysis as occurring between time steps. However, most of the issues we discuss will arise whatever the work units of a simulation or task being performed; the fact that some analysis must be executed on each work unit in order to avoid the cost of writing more to disk does not vary.

In this report, we'll make a distinction between *checkpoint* data – used for restarts when a job fails – and *results* data – used for analysis of the simulation. Some codes (e.g., S3D[1]) rely on checkpoint data for both post-processing and checkpointing while others (e.g., CTH, Presto) write both checkpoint and results data.

[1] Actually, S3D writes checkpoint data plus auxiliary results, but the auxiliary results are typically used in combination with the checkpoint data during post-processing.



(a) Flow of execution for a traditional code.



(b) Flow of execution for an *in situ* code.

Figure 1. Flow of execution for simulations.

2 Hardware Architecture

Many high-performance compute (HPC) platforms use lightweight kernels and system libraries that do not offer traditional Unix services. In particular, sockets and threads are not provided on catamount [KB05]. Lightweight kernels run on parallel computers place constraints on threads because they seek to minimize any discrepancies in the timing of message passing interface (MPI) collective operation calls. One school of thought posits that small discrepancies in timing can account for much of the degradation from linear speedups encountered as the number of processes grows into the thousands.

Both the absence of sockets and threads means that – whether it is wise to do so or not[2] – direct user interaction with a running simulation is hard to impossible on many HPC platforms. One way to circumvent some of these restrictions on catamount deployments is to use I/O or other service nodes. These nodes typically run a different operating system but have access to the high-speed interconnect used by MPI jobs running on the HPC nodes. If these nodes can take part in a job as the rank 0 node of a heterogeneous MPI run, they may open socket connections for desktop delivery and other user interaction. Otherwise, users must be content to examine files created by running jobs.

Using HPC nodes for analyses typically performed as post-processing presents other challenges. Some HPC platforms, such as Blue Gene/L [EGT04] provide a relatively small amount of memory per processor and no local disk for page swapping so if a simulation requires a large amount of memory and is unable or unwilling to release some of it in between time steps, other analyses must run with extremely limited memory. While modern `FORTRAN` codes allow dynamic array allocation, it is unrealistic to expect simulation developers to free and reallocate memory after each time step.

Finally, we know of no HPC platform that includes hardware designed for rasterization with each processor. This means that rendering images for later inspection must be done using software rasterization. Software rasterization can be many orders of magnitude slower than its hardware counterpart but if rendering hardware is available, it typically (1) is much smaller in capacity than the HPC platform it is coupled to and (2) requires data be transferred over a network from the HPC platform running the simulation. As HPC machines continue to increase in size, we do not expect hardware rendering to remain separate from the HPC; either hardware for rendering will become ubiquitous because this hardware also provides an architecture useful for general-purpose computing or software rendering will be used because it is not economically efficient to buy special-purpose rendering hardware and additional network resources in quantities large enough to scale with HPC platforms.

[2]The wisdom of interacting with running simulations is discussed in the next section.

3 Interfacing with a Simulation

The interface between a simulation and some set of analysis tasks to perform in between time steps is fraught with peril. Or at least mild apprehension. Just ask any simulation developer about it and watch his face crumple. In order to build an application that includes both simulation and analysis, you must be able to link object files created with different compilers (§3.1), share memory between the two (§3.2), provide information in at least one direction – from the simulation to the analysis (§3.3), and determine an appropriate balance between time spent on each (§3.4).

3.1 Linking

Several mentions of `FORTRAN` in the previous section have indicated one issue integrating simulation and analysis codes: linking. Many simulations are written in `FORTRAN` while analysis (especially visualization software) is typically written in C++. There is a standard for linking `FORTRAN` and C code in a cross-platform way, but it was not established until 2003 [Don06]. As a result, many legacy `FORTRAN` codes (not easily compiled using newer `FORTRAN` compilers or running on platforms where a newer compiler is unavailable) rely on vendor-specific conventions for passing arguments and return values. Inter-language linking is also hampered by the fact that C++ compilers typically mangle the names of functions to force type safety but no mangling scheme is in the specification – so compilers from different vendors (or even different versions from the same vendor!) will name functions differently. Interface functions must be placed inside an `extern "C"` { ... } block in C++.

Beyond argument and return-value conventions, linking `FORTRAN` and C++ can present other problems. Frequently, system libraries need not be explicitly specified when linking code compiled with either C++ or `FORTRAN` because these libraries are added by the language-specific compiler stage. However, when linking code that requires both types of system libraries, either the C++ or the `FORTRAN` compiler will be invoked and system libraries from whichever one is not invoked will be missing from the list of libraries sent to the linker.

3.2 Memory

As mentioned in §2, the amount of physical memory available to each process may be small. If a simulation can not or will not free any memory in between time steps to allow for analysis, it is up to the analysis to be as accommodating as possible. One way to accomplish this is to use memory during the analysis that is provided by the simulation. For example, S3D is a `FORTRAN` code that has a large amount of space used to store intermediate computations during a time step that is otherwise unused between time steps. The address of this storage is fixed for the entire length of the simulation. S3D provides the address of this space to post-processing code and fills part of it with variable values of interest after each time step. Any analysis may write data into this scratch space with the caveat that it will be trampled during the next time step calculation.

A library such as VTK could take advantage of this by overriding memory allocation for large arrays (such as the `vtkDataArrayTemplate` subclasses) to use storage in the scratch space for array values. See Figure 2 for an example. Clever allocation might even allow post-processing class instances to stay allocated for the entire length of a simulation as long as their persistent (but hopefully small) state was allocated from a separate pool of non-volatile memory. Note that care must be taken when dealing with multidimensional arrays that are passed between C and FORTRAN code as the storage order will be reversed between the two. Where the simulation provides data interleaved in a different order than the analysis requires (and it cannot be reordered for performance or code maintenance reasons), either a deep copy will have to be made or some special subclass of `vtkDataArray` that uses iterators will have to be implemented[3].

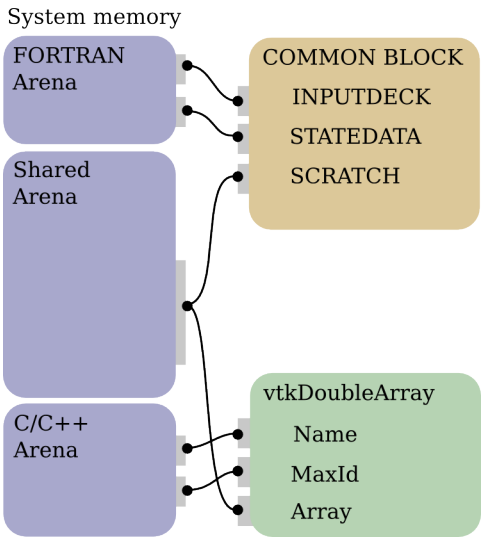


Figure 2. The system memory map is shown in blue to the left. To the right are examples of how FORTRAN variables (orange) and C++ instance members (green) might reference memory to share portions that may be volatile between time steps.

For the applications we are interested in, memory consumption is not yet an issue, but that will only continue to be true if relatively simple analysis is performed. On machines at Oak Ridge, S3D consumes approximately 30% of the 1 GiB per process that is available. The S3D executable is 15 MiB in size.

[3]This may not be possible without rewrites of VTK filters that retrieve pointers to array data and work with it directly. An example would be image data filters that use voxel, row, and plane strides obtained with `vtkImageData::GetIncrements()`.

3.3 Sharing information

Because analysis tasks are performed on data provided by the simulation, the simulation must be modified in some way to provide at least a minimal API to its data and (hopefully) metadata such as field names and the role data arrays play in a simulation. There are already some efforts underway to provide analyses access to simulation data, including the Interoperable Technologies for Advanced Petascale Simulations (ITAPS) mesh interface [CFD+07] and the Adaptable I/O System (ADIOS) [LKS+08]. In general, and assuming the simulation will be communicating with a reusable framework for analysis tasks as opposed to a one-off analysis code with pre-existing knowledge of the simulation, the simulation should:

Optional Pass control to the analysis tasks at the beginning and end of a job in order to allow for initialization and finalization code to be run. Any configuration information must either be provided here or after each and every time step.

Mandatory Provide pointers to information for further analysis, plus information about the size, type, and stride. If the address will not change over the course of the simulation, this may be done once at initialization.

Optional With the above, also provide information about the purpose or intent of the information. One example would be to indicate that a pointer references mesh connectivity, mesh coordinates, or field data.

Mandatory Pass control to the analysis task(s) after each time step (or other unit of work) is complete.

Optional With the above, also provide information about the current simulation time (or time step).

Optional Provide the analysis task(s) with pointers to data or functions for accepting the results of analysis.

Common configuration information passed during initialization includes (1) MPI state (such as a communicator that should be used), (2) any constraints on the amount of memory (or other system resources) that analysis tasks may consume, and (3) per-job configuration information from a simulation's input deck.

3.4 Balancing simulation and analysis workload

Once the simulation and analysis tasks have been integrated into a single executable, you must decide (or be able to control) the amount of time spent simulating vs. analyzing. This is not a simple decision to make, nor is it easy to enforce once you decide on a balance between the two. Let's call B the ratio of simulation time T_s to the total job time $T_j = T_s + T_a + T_{io,s} + T_{io,a}$ where T_a is the time required for all analysis tasks, $T_{io,s}$ is the time required by the simulation for I/O such

as checkpointing, and $T_{io,a}$ is the time required by the analysis tasks for I/O such as saving results. In general

$$B = \frac{T_s}{T_s + T_a + T_{io,s} + T_{io,a}}$$

if both checkpointing and analysis are performed in addition to simulation. In the case where no analysis is performed but both checkpoints and results are saved separately (perhaps at different rates) then T_a will be zero but both $T_{io,s}$ and $T_{io,a}$ will be positive.

Some checkpoint I/O strategies attempt to overlap $T_{io,s}$ and T_s by performing an in-memory copy and writing to disk from that so that the next time step can proceed while asynchronous I/O is occurring however it is not clear that this provides a significant advantage for simulations of the type Sandia runs [Pla93]. This strategy may need to be revisited for the case when analysis might be performed on an in-memory copy and the results of this analysis written to disk.

3.4.1 Determining a balance

In situations where checkpoints were previously used for post-processing (as opposed to dedicated results files), one way to approach the decision for B is to try to keep it constant as you transition from frequent checkpointing required to maintain the fidelity of any post-processing to sparse checkpointing to allow recovery on job failure. If you had to perform N_0 checkpoints before and N_1 checkpoints after, with each checkpoint consuming time T_c, then $T'_{io,s} = N_1 T_c < N_0 T_c$ and

$$T_a + T_{io,a} = (N_0 - N_1)\, T_c.$$

Of course, this only works if $N_0 > N_1$.

Depending on (1) how much more slowly disk bandwidth D grows than theoretical operating speed in the next years, (2) the size of a checkpoint s_c, and (3) whether the number of components in the system expected to fail per unit of time E_f stops increasing[4], it may not be possible to checkpoint a simulation at all. If $E_f \frac{s_c}{D} \geq 1$, then at least one failure is expected during the interval of time required by a single checkpoint. Given the trends of the last decade, s_c and E_f will continue to increase at a much faster rate than D due to slower growth in bandwidth and the prohibitive cost of increasing bandwidth by increasing the number of active disks. For instance, Schroeder and Gibson [SG07] see peak theoretical operating speeds continuing to grow at 100% per year while disk bandwidth grows at 20% per year.

Consider the case where no checkpointing may be performed. No state is written out at the conclusion of the job and only the results of the analysis are available for inspection. If a failure occurs during the job, the entire simulation must be recalculated. In this case $T_{io,s} = 0$. For no checkpointing to be a viable strategy, $E_f N (T_s + T_a + T_{io,a}) < 1$. Note that many simulations already operate in the regime where $E_f N T_s > 1$ – hence the need for checkpoints. These simulations simply cannot work without checkpointing of some sort and either D must be increased no matter how costly

[4]Note that the expectation of a particular *single* component failing per unit time $E_{f,c}$ is modeled as a constant but because the total number of components in HPC platforms is increasing rapidly, the effect is that E_f is increasing.

or some sort of robustness to component failure must be written in to the simulation and/or HPC kernel if it is to scale up. Note that increasing D generally involves adding many more components to the system and thus increases E_f.

Assuming D remains prohibitively expensive but that some technique such as job migration makes it possible to survive a component failure, then B should be chosen so that the time required to perform the analysis is less than the time required to checkpoint the state and obtain reasonable fidelity results by performing the analysis as a post-processing step. In this case, we know that $T_{io,s} \geq T_a + T_{io,a}$ so that

$$B = \frac{T_s}{T_s + T_a + T_{io,a}} \geq \frac{T_s}{T_s + T_{io,s}}$$

and the time spent checkpointing is simply replaced by the time spent on analysis plus any I/O required to save the analysis results for later inspection. However, this neglects the cost of job migration or any other strategy to accommodate component failure. When the cost of job migration is included, B may decrease as migration may differentially affect compute-intensive tasks (analysis) compared to I/O intensive tasks (checkpointing).

3.4.2 Achieving a balance

There are many strategies for achieving a given ratio B. Generally, these strategies are aimed at limiting either T_a or $T_{io,a}$:

T_a The fidelity of the analysis might be reduced by performing it less frequently than once per timestep.

T_a The fidelity of the analysis might be reduced by performing it on a reduced-resolution version of the simulation data[5].

T_a The analysis might only be performed on a subset of the simulation data specified by a person (either in a fixed manner – such as a spatial region of interest – or in a programmatic manner, when a certain condition is met in a region of simulation space-time).

T_a The analysis might be offloaded to service nodes with specialized hardware (such as video cards for rendering). Note that offloading generally only works if a subset of the data is offloaded since service nodes tend to be much less numerous than HPC nodes and network bandwidth will not accommodate transferring all the data in a reasonable amount of time.

$T_{io,a}$ The results of the analysis might be saved at a reduced resolution or subsetted.

$T_{io,a}$ The end product of the analysis might be put in a different form (e.g., $T_{io,a}$ could be reduced by writing a single rendered image of the simulation instead of writing a subset of simulation data for later rendering).

[5]This is simple for simulations on regular spatial grids where subsampling can be employed.

Of course, then the size of the analysis output is proportional to the size of the input, any of the approaches above that reduce T_a will also reduce $T_{io,a}$ as a side effect.

4 User Interface

First, it is important to note that users are stupid, greedy, lazy slobs. System administrators are thus forced to restrict the information and power available to them. As a consequence, they don't know when their job will start running on an HPC platform. They are too feckless to sit in front of a terminal while it runs. And they are too easily frustrated by latency to tolerate the framerates that remote visualization can currently provide. This means they are not around to poke at the screen and choose interesting simulation data to save or choose a type of analysis to perform.

Since we are these very creatures, it is clearly a moral imperative for us to do the minimum required by the job[6]. That generally means we will have little (if any) control over the simulation and analysis while they are running and leaves us with the what-would-I-do-if-I-were-Bob approach to *in situ* analysis: scripting (i.e., Bob decides in advance what he is willing to do) and machine learning (i.e., Bob thinks the machine is more likely to figure out what he would do). In practice, both scripting and machine learning will always be employed to some degree since almost any configuration information that would be used to prepare machine learning could be called a script and almost any script that performs analysis could be considered some sort of machine learning. However, it is convenient to discuss the two separately below.

4.1 Scripting

Scripting is most useful when the results of the simulation are relatively predictable and we know what we want to measure, as opposed to needing some way to explore the results to find new behavior.

One promising technique for generating scripts is to use an interactive tool such as ParaView [MRG+08] to perform some analysis task on a small dataset similar to the output of a large simulation to be run. The set of operations created by the user can then be saved to a script and quickly adapted by the user to work on an *in situ* analysis. In order for this approach to be effective,

- the output of visualization tools must be easily read by humans,
- the script created must be trivially portable to a batch environment on a different architecture, and
- must easily scale to the required job size.

One open issue with scripting is how analysis scripts for a particular run should be specified: they might be called out by filename in the input deck, or they might encompass the input deck. The former is much simpler to implement and allows existing tools that read and write input decks in legacy formats to continue working with minimal changes. The latter is clearly more flexible

[6] Anything else would fail to be lazy, and laziness is an approach that has worked well to date.

but does involve more work and might altogether prevent GUI tools from parsing input decks for presentation.

4.2 Machine Learning

Machine learning is most useful when the results of the simulation are unpredictable and we want to explore the results rather than compute some pre-determined quantity. The term machine learning can refer to a wide variety of tasks including feature recognition, feature tracking, classification, and hypothesis testing.

One expected use for machine learning is the automatic selection of subsets. As a simulation is scaled up to the point where saving the full results for later inspection is not possible, saving a very small subset might still be feasible. Static methods for subset selection (such as requesting certain elements or a particular set of grid extents) typically select much more data than is required, and this becomes increasingly unacceptable as the scale increases. Selecting blocks based on information such as gradient magnitudes might be more useful but not robust to situations where multiple scalar fields are interacting. Thus, feature detection (which might be based on parametric statistical modeling or parameter-free classification) could be an alternative that selects acceptably small subsets.

4.3 The Last Mile

Even assuming we are able to use machine learning and scripting to good effect, the results of a simulation analysis must be accessible to users. Because HPC platforms are expensive, there are few locations with direct (low-latency, high-bandwidth) access to the disks holding results. Even small subsets of of an exascale simulation may be too large to transmit across the internet for local inspection. On the other hand, the latency of the internet can leave image delivery techniques unusable. One solution is the use of lumigraph rendering [BBM+01]. In this approach, images rendered on the remote server are accumulated, along with their camera parameters, by the client (instead of being rendered once and then discarded). While waiting for more images to be delivered from the server in response to camera motion, the set of accumulated images is used to synthesize an image matching the new camera parameters.

5 Conclusion

We have presented a set of issues that must be addressed when developing visualization and other post-simulation analysis tools for HPC platforms over the next several years. They make a strong case for *in situ* analysis performed in lock step with a simulation, and given that human input cannot be required during simulation leaves us with a need for better scripting and feature recognition and analysis tools. Where humans do enter into the problem, greater physical distances, higher latencies, and lower bandwidths will be encountered. Running a large-scale simulation on a distant machine is not unlike directing an exploratory mission to the outer planets of the solar system from Earth; days or more may elapse before results are available, instructions must be pre-programmed, and results must be carefully selected before transmission back to humans who must be able to gain some insight from them.

References

[BBM+01] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured lumigraph rendering. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 425–432, New York, NY, USA, 2001. ACM.

[CFD+07] Kyle Chand, Brian Fix, Tamara Dahlgren, Lori Freitag Diachin, Xiaolin Li, Carl OllivierGooch, E. Seegyoung Seol, Mark S. Shephard, Tim Tautges, and Harold Trease. The ITAPS iMesh interface. Technical Report version 0.7, U. S. Department of Energy: Science Discovery through Advanced Computing (SciDAC), 2007.

[Don06] Aleksander Donev. Interoperability with C in FORTRAN 2003. *SIGPLAN FORTRAN Forum*, 25(1):8–12, 2006.

[EGT04] Toshikazu Ebisuzaki, Robert Germain, and Makoto Taiji. Petaflops computing. *Commun. ACM*, 47(11):42–45, 2004.

[KB05] Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, Catamount. Technical Report SAND2005-2780C, Sandia National Laboratories, 2005.

[LKS+08] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable io system (ADIOS). In *CLADE 2008 at HPDC*, Boston, Massachusetts, June 2008. ACM.

[MRG+08] Kenneth Moreland, David Rogers, John Greenfield, Berk Geveci, Patrick Marion, Alexander Neundorf, and Kent Eschenberg. Large scale visualization on the cray XT3 using ParaView. In *Cray User's Group*, May 5–8 2008.

[Pla93] James Steven Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Princeton, June 1993.

[SG07] Biancha Schroeder and Garth A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78, 2007. doi:10.1088/1742-6596/78/1/012022.

DISTRIBUTION:

1	MS 9159	David Thompson, 8963
1	MS 1323	Kenneth L. Moreland, 1424
1	MS 1323	Nathan D. Fabian, 1424
1	MS 0822	Lisa G. Ice, 9326
1	MS 9159	Robert L. Clay, 8963
1	MS 1323	David Rogers, 1424
1	MS 0823	Constantine J. Pavlakos, 9326
2	MS 9018	Central Technical Files, 8944
1	MS 0899	Technical Library, 9536

Sandia National Laboratories