Systematic Debugging Methods for Large Scale HPC Computational Frameworks

Alan Humphrey,
Qingyu Meng,
Martin Berzins
School of Computing and SCI Institute
University of Utah, USA
{ahumphre,qymeng,mb}@cs.utah.edu

Diego Caminha B. de Oliveira,
Zvonimir Rakamarić,
Ganesh Gopalakrishnan
School of Computing
University of Utah, USA
{caminha,zvonimir,ganesh}@cs.utah.edu

***Abstract*—Parallel computational frameworks for high performance computing (HPC) are central to the advancement of simulation based studies in science and engineering. Unfortunately, finding and fixing bugs in these frameworks can be extremely time consuming. Left unchecked, these bugs can drastically diminish the amount of new science that can be performed. This paper presents our systematic study of the Uintah Computational Framework, and our approaches to debug it more incisively. Our key insight is to leverage the modular structure of Uintah which lends itself to systematic debugging. In particular, we have developed a new approach based on Coalesced Stack Trace Graphs (CSTGs) that summarize the system behavior in terms of key control flows manifested through function invocation chains. We illustrate several scenarios how CSTGs could help efficiently localize bugs, and present a case study of how we found and fixed a real Uintah bug using CSTGs.**

***Index Terms*—Computational Modeling and Frameworks, Parallel Programming, Reliability, Debugging Aids.**

I. Introduction

Computational frameworks for high performance computing (HPC) are central to the advancement of simulation based studies in science and engineering. With the growing scale of problems and the growing need to simulate problems at higher resolutions, modern computational frameworks continue to escalate in scale, now exceeding a million cores in their current deployments.

As with many such codes, there are bugs present in the code implementing computational frameworks. In the case of large parallel frameworks, finding and fixing bugs can be an order of magnitude more time consuming, particularly for those bugs that arise from the parallel nature of the code and for which testing may only be done through infrequently scheduled batch runs, possibly at large core counts.

This lengthy debugging process can arise even though the creators of computational frameworks put in considerable effort and thought into carefully structuring them, while users of these frameworks also write a non-trivial number of tests as well as assertions in their code. Clearly, we need steady progress to be made in systematic testing methods that help trigger deeply hidden bugs, and also systematic debugging methods that help observe these bugs as well as root-cause them. This paper presents our systematic study of a computational framework under development at the University of Utah called the *Uintah Computational Framework* [1] (or just Uintah), and the efforts we are putting into Uintah in order to debug its bugs quickly and effectively.

The prevalence of software bugs and the difficulty of debugging are well known. As one recent example, the authors of the popular Photoshop tool mention: *...the single longest-lived bug I know of in Photoshop ended up being nestled in that code. It hid out in there for about 10 years. ... the person who put that in there didn't think about the fact that ... [a pair of calls] had to be made atomic whenever you're sharing the file position across threads.* This reinforces the fact that bugs that are rare to trigger and hard to root-cause (debug) can be serious impediments to the design of large-scale and reliable software. Our work is aimed at bringing in systematic (*formal*) techniques for both triggering bugs as well as debugging, which can be deployed in practice. Our main contribution is a light-weight technique for comparing two executions of a system—one typically the working ("golden") version and the other the new version being tested—based on their execution profiles.

Compared to "traditional software systems," there has, historically, been relatively less attention paid to bugs occurring within HPC in general and computational frameworks in particular. However, this situation is rapidly changing. Recently, the authors provided a perspective on this issue, and covered facts specific to the academia and national laboratories [2]. Two take-away points from this article are now elaborated. First, HPC and the continuous mathematics underlying the classes of problems solved by HPC codes is not taught in mainstream Computer Science curricula, and there is insufficient interest created amongst students and faculty interested in rigorous software engineering methods to apply their techniques to HPC. Second, the styles of concurrency present in HPC qualitatively differ from well-studied situations in rigorous software engineering. For instance, in rigorous software engineering, considerable attention has been paid to device drivers, operating systems, and transactional systems. In contrast, in HPC, typical computations are based upon large coupled systems of partial differential equations, run for days (if not months), and are orchestrated around time-stepped activities. Significant usage is made of infrastructural

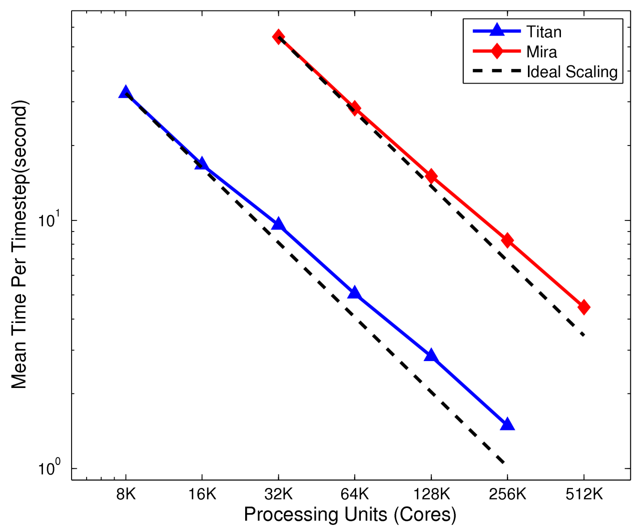


Fig. 1. Strong Scaling of Uintah Benchmark Problem Using AMR

components (*e.g.*, schedulers), adaptive mesh refinement algorithms, as well as third-party libraries (*e.g.*, iterative solvers for large systems of linear equations). A typical computer-science-trained software engineering researcher does not have the background to understand all the components and their interactions. *Collaboration between HPC and core CS researchers is crucial in developing suitable rigorous software engineering approaches to modern computational frameworks*.

This paper summarizes preliminary results [3] from an ongoing collaboration between a subset of its authors interested in building a high-end problem solving environment called Uintah, and a subset interested in developing *formal* software testing approaches that can help eliminate code-level bugs, hence enhancing the value offered by Uintah. We are developing *Uintah system Runtime Verification* (URV) techniques that can be deployed in field-debugging situations. We aim to make our results broadly applicable to other computational frameworks and HPC situations. While traditional debuggers (*e.g.*, Allinea DDT and Roguewave) are the mainstay of today's debugging methods, typically these tools are good at explaining the execution steps close to the error site itself—and not at providing high level explanations of cross-version changes.

II. The Uintah Computational Framework

A proven approach to solving large-scale multi-physics problems on large-scale parallel machines is to use computational frameworks such as Flash [4] and Chombo [5].

The Uintah Computational Framework[1] originated in the University of Utah DOE Center for the Simulation of Accidental Fires and Explosions (C-SAFE) (9/97-3/08) which focused on providing software for the numerical modeling and simulation of accidental fires and explosions. Uintah was intended to make it possible to solve complex fluid-structure interaction problems on parallel computers. In particular Uintah is designed for full physics simulations of fluid-structure interactions involving large deformations and phase change. The term full physics refers to problems involving strong coupling between the fluid and solid phases with a full Navier-Stokes representation of fluid phase materials and the transient, nonlinear response of solid phase materials which may include chemical or phase transformation between the solid and fluid phases. Uintah uses a full multi-material approach in which each material is given a continuum description and is defined over the complete computational domain.

Uintah contains four main simulation components: 1) the ICE code for both low and high-speed compressible flows; 2) the multi-material particle-based code MPM for structural mechanics; 3) the combined fluid-structure interaction (FSI) algorithm MPM-ICE; and 4) the ARCHES turbulent reacting CFD component that was designed for simulation of turbulent reacting flows with participating media radiation. Uintah makes it possible to integrate multiple simulation components, analyze the dependencies and communication patterns between these components, and efficiently execute the resulting multi-physics simulation.

These Uintah components are C++ classes that follow a simple interface to establish connections with other components in the system. Uintah then utilizes a task-graph of parallel computation and communication to express data dependencies between multiple application components. The task-graph is a directed acyclic graph (DAG) in which each task reads inputs from the preceding task and produces outputs for the subsequent tasks. The task's inputs and outputs are specified for a generic patch in a structured adaptive mesh refinement (SAMR) grid, thus a DAG will be created with tasks of only local patches. Each task has a C++ method for the actual computation and each component specifies a list of tasks to be performed and the data dependencies between them [7].

This design allows the application developer to only be concerned with solving the partial differential equations on a local set of block-structured adaptive meshes, without worrying about explicit message passing calls in MPI, or indeed parallelization in general. This is possible as the parallel execution of the tasks is handled by a runtime system that is application-independent. This division of labor between the application code and the runtime system allows the developers of the underlying parallel infrastructure to focus on scalability concerns such as load balancing, task scheduling, communications, including accelerator or co-processor interaction.

Uintah scales well on a variety of machines at small to medium scale (typically Intel or AMD processors with Infiniband interconnects) and on larger Cray machines such as Kraken and Titan. Uintah also runs on many other NSF and DOE parallel computers (Stampede, Keeneland, Mira, etc). Using its novel asynchronous task-based approach with fully automated load balancing Uintah demonstrates good weak and strong scalability up to 256k and 512k cores on DOE Titan and Mira respectively as shown in Fig. 1.

[1]For reasons of space, we cite here an earlier CiSE paper on the applications of Uintah [6]; we refer the reader to that paper for more information and references.

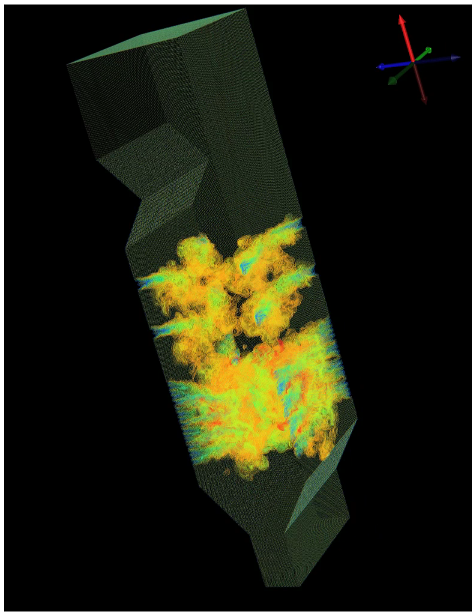


Fig. 2. Volume Rendering of O_2 Concentrations in a Clean Coal Boiler

Uintah is used for a broad range of multi-scale multi-physics problems such as angiogenesis, tissue engineering, green urban modeling, blast-wave simulation, semi-conductor design and multi-scale materials research. A recent example is the multiscale modeling of accidental explosions and detonations [6]. As part of the National Nuclear Security Administrations (NNSA) Predictive Science Academic Alliance Program II (PSAAP II), the University of Utah will serve as a Multidisciplinary Simulation Center and will also use Uintah on Department of Energy supercomputers to develop a prototype low-cost, low-emissions coal power plant that could electrify a mid-sized city. The goal of this research is to help power poor nations while reducing greenhouse emissions in developed ones. Uintah will be used to simulate and predict performance for a proposed 350-megawatt boiler system that burns pulverized coal boiler with pure oxygen rather than air. These simulations (Fig. 2) will help guide the design of such a boiler to ultimately provide economical power to developing nations, and help industrialized nations meet increasingly stringent emissions standards. Through a Collaborative Research Alliance (CRA) funded by the U.S. Army Research Laboratory, Uintah will also play a key role in serving as the framework for multi-scale modeling tools developed within this research alliance to ultimately deliver simulation-based design of visionary electronic materials, devices and systems. The objective of this Alliance is to conduct fundamental research necessary to enable the quantitative understanding of electronic materials from the smallest to the largest relevant scales.

An important goal in this Department of Defense project is for Uintah to become the underlying scaffolding for simulation tools developed by the CRA, and ultimately that Uintah supports multiscale, multi-component coupling between arbitrary component subsystems. Initial work in this area has been focused on the development of generic, particle based simulation capabilities within Uintah based on functionality within the Lucretius molecular simulation package [8] (developed at the University of Utah). Once fully implemented, molecular dynamics (MD) simulations with polarizable force fields and arbitrary imposed external potentials may be efficiently parallelized by Uintahs infrastructure to achieve handling of much larger systems.

A. *Growth Phases of Uintah*

One of the main approaches suggested for the move to multi-petaflop architectures (and eventually exascale) is to use a graph representation of the computation to schedule work, as opposed to a bulk-synchronous approach in which blocks of communication follow blocks of computation. The importance of this approach for exascale computing is expressed by recent studies [9]. Following this general direction, Uintah has evolved over the past decade over three significant phases:

- 1998–2005: having a static task-graph structure and running at about 1000 cores;
- 2005–2010: incorporating many dynamic techniques, including new adaptive mesh refinement methods, and scaling to about 100K cores;
- 2010–2013: Uintah has shown promising results on problems as diverse as fluid-structure interaction and turbulent combustion at scales well over 500K CPU cores. It presently incorporates shared memory (thread-based) schedulers as well as GPU-based schedulers.

Frameworks such as Uintah aspire to be critically important components of our national high performance computing infrastructure, contributing to the solution of computationally challenging problems of great national consequence. Being based on sound and scalable organizational principles, they lend themselves to easy adaptation as witnessed by the Uintah phases mentioned above. For example, GPU schedulers were incorporated into Uintah in a matter of weeks. This fundamentally leads to systems such as Uintah being in a state of perpetual development. Furthermore end-users are always trying to solve larger and more challenging problems as they stay at leading edges of their subjects. There is always a shortage of CPU cycles, total memory capacity, network bandwidth, and advanced developer time. In addition, there is constant pressure to achieve useful science under tight budgets. Structured software development and documentation compete for expert designer time as much as the demands to simulate new problems and to achieve higher operating efficiencies by switching over to new machine architectures.

Previously, the formal methods authors of this paper have explored various scalable formal debugging techniques for large-scale HPC and thread-based systems (e.g., [2], [10]). The URV project is different from these efforts since it is an attempt to integrate light-weight and scalable formal methods into a problem-solving environment that is undergoing rapid development and real usage at scale.

There are many active projects in which parallel computation is organized around task-graphs. Charm++ [11] has pioneered the task-graph approach and finds applications in high-end molecular dynamics simulations. The DAGuE framework [12] is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. Our interest in Uintah stems from two factors: (1) Uintah has scaled by a factor of 1000 in core-count over a decade and finds numerous real-world applications, (2) we are able to track its development and apply and evaluate formal methods in a judicious manner. We believe that our insights and results would transfer over to other similar computational frameworks—in existence or planned.

III. Uintah Runtime Verification

The current focus of URV is to help enhance the value of Uintah by eliminating show-stopper code-level bugs as early as possible. In this connection, it is too tempting to dismiss the use of formal testing methods on account of the fact that many of these methods do not scale well, and that many interesting field bugs occur only at scale. While this may be true in general, there are a number of bugs which are reproducible at lower scales. This observation is supported by error logs from previous Uintah versions where many of the errors (*e.g.*, double-free of a lock, mismatched MPI send and receive addresses) were unrelated to problem scale. Of course, scale-dependent bugs do exist. According to our experience, such bugs are due to subtle combinations of code and message passing, and are sometimes exceptionally challenging to find at very large core counts with only batch access. Hence, they are clearly important and are the eventual goal of our future research.

In the URV project, we are motivated by one crucial observation: *the ease with which a system can be downscaled depends on how well structured it is.* There are many poorly structured systems that allow only certain delicate combinations of such operating parameters; sometimes, these parameters are not well documented. Uintah, on the other hand, follows a fairly modular design, allowing many problems to be run across a wide range of operating scales—from two to thousands of CPU cores in many cases. There are only relatively simple and well-documented parameter dependencies that must be respected (relating to problem sizes and the number of processes and threads). This gives us a fair amount of confidence that well-designed formal methods can be applied to Uintah at lower scales to detect many serious bugs (examples are provided later in §III).

Scalable and Intuitive Debugging Methods. Our main contribution in this paper is our approach to debug large-scale parallel systems by highlighting the execution differences between working and non-working versions of the system. A straightforward "diff" of these systems (say by comparing actual temporal traces) has an extremely low likelihood of root-causing problems. This is because the actual parallel program schedules of various threads and processes are likely to differ from run to run—*even for just one version of a system.* Our method relies on obtaining *Coalesced Stack Trace Graphs* (CSTGs) that tend to forget schedule variations and highlight the flow of function calls during execution. We demonstrate that collecting CSTGs and diffing them is a practical approach by demonstrating how we have helped Uintah developers root-cause a bug caused by switching to a different Uintah scheduler. While stack trace collection and analysis has been previously studied in the context of tools and approaches such as STAT [13] and HPCToolkit [14], their focus has not been on cross-version ("delta") debugging as we have implemented.

A. Coalesced Stack Trace Graphs

A stack trace is a report of the active function calls at a certain point in time during the execution of a program. Stack traces are commonly used to observe crashes and to learn where a program failed, being very helpful in the debug phase of software development. They are also being used in more advanced techniques to help find problems in parallel applications. For instance, STAT [13] uses stack traces to compare the state of different processes in a given time, making it easier to identify when a process is behaving differently. Spectroscope [15] collect stack traces to diagnose performance changes by comparing request flows.

Collecting stack traces throughout the execution of a program may reveal interesting facts about its behavior. For instance, it can show the number of times a function was called and the different call paths leading to a function call. However, the number of stack traces that can be obtained from an execution may be very large. Therefore, for better understanding of this data, we use graphs that can compact several millions of stack traces in one manageable figure. We call such a graph Coalesced Stack Trace Graph (CSTG), which is a non-chronological view of everything recorded during an execution (see Fig. 3(a) or 3(b)).

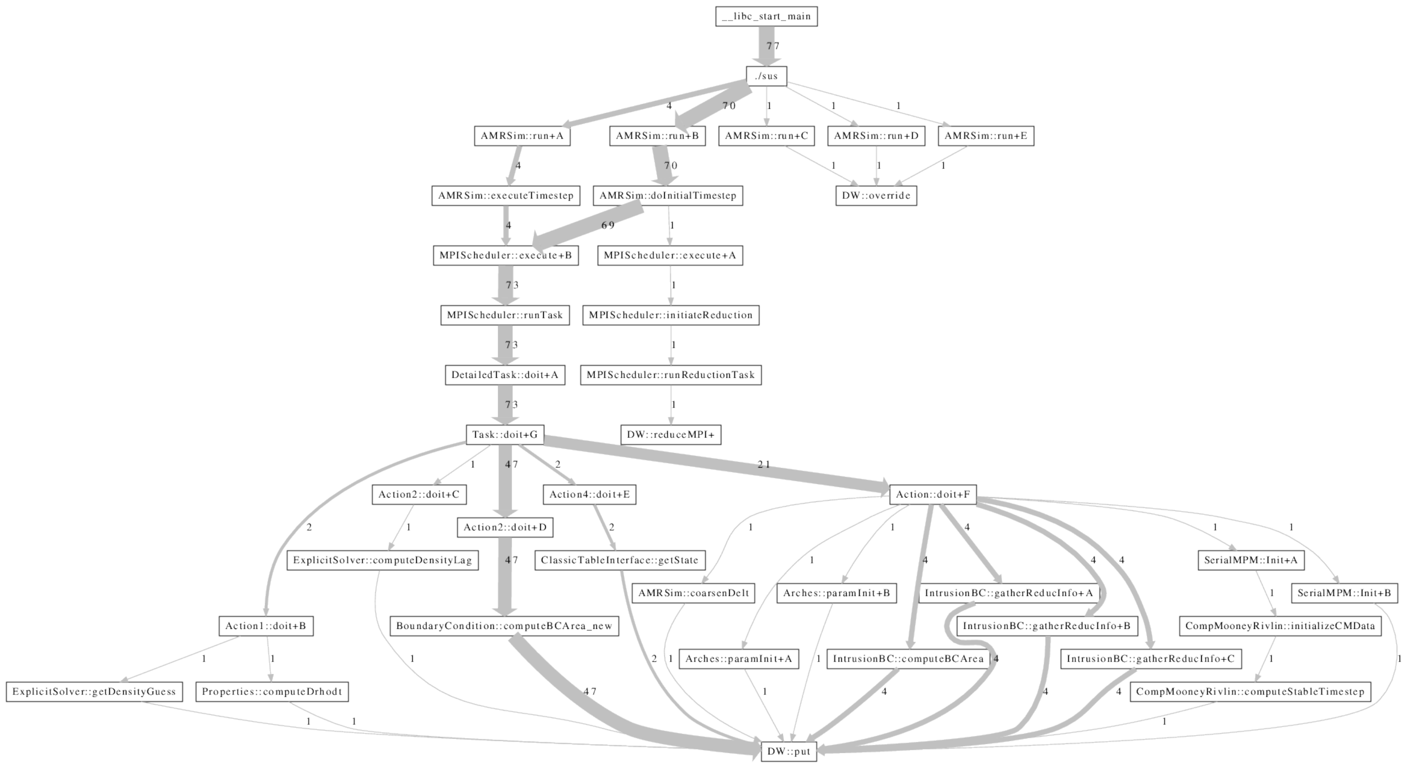
CSTG is a very compact and useful way to better understand a program execution. More importantly, CSTGs have proven helpful in many realistic bug-hunting scenarios, especially when we are able to compare different CSTGs. As examples we can cite:

Working and non working versions. Software projects are often constantly evolving. New components are developed to replace the old ones, and sometimes they carry new bugs. Understanding why a new component is not doing what it is suppose to do can be easier when comparing executions against the older working component.

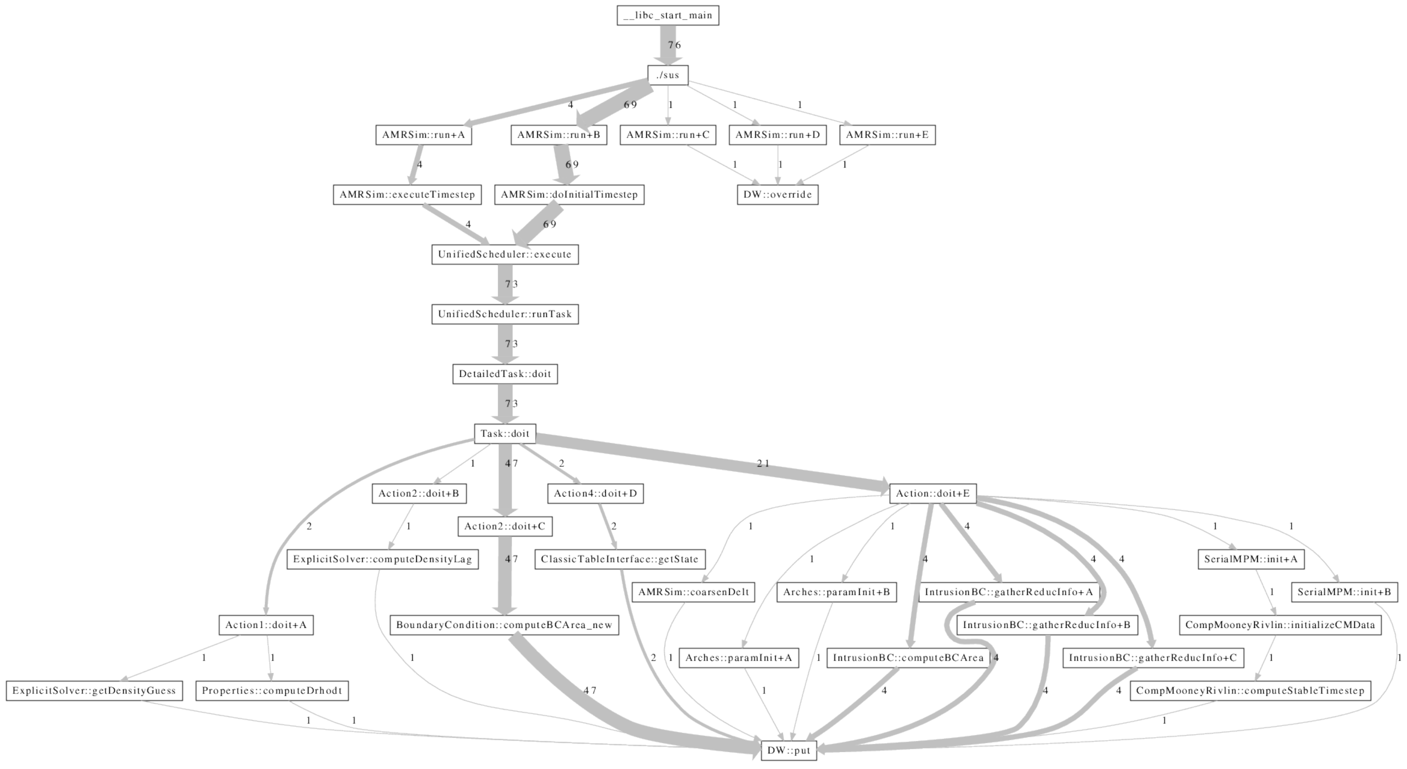
Symmetric events (*e.g.*, sends/recvs, lock/unlock, new/delete). Matching events are common in any program. Having a simple visual representation of such events allows for a quick identification of potential problems.

Repetitive sets of events (*e.g.*, time-steps). It is common to find algorithms that behave the same (or very similarly) through a sequence of steps, such as in simulations and loop iterations. Noticing that something unusual is happening at some execution step is often easier when using CSTGs.

Different processes and threads. In many parallel programs, the same work is done in different threads or



(a) CSTG for the Working Version.



(b) CSTG for the Crashing Version.

Fig. 3. Using CSTGs to Understand a Bug.

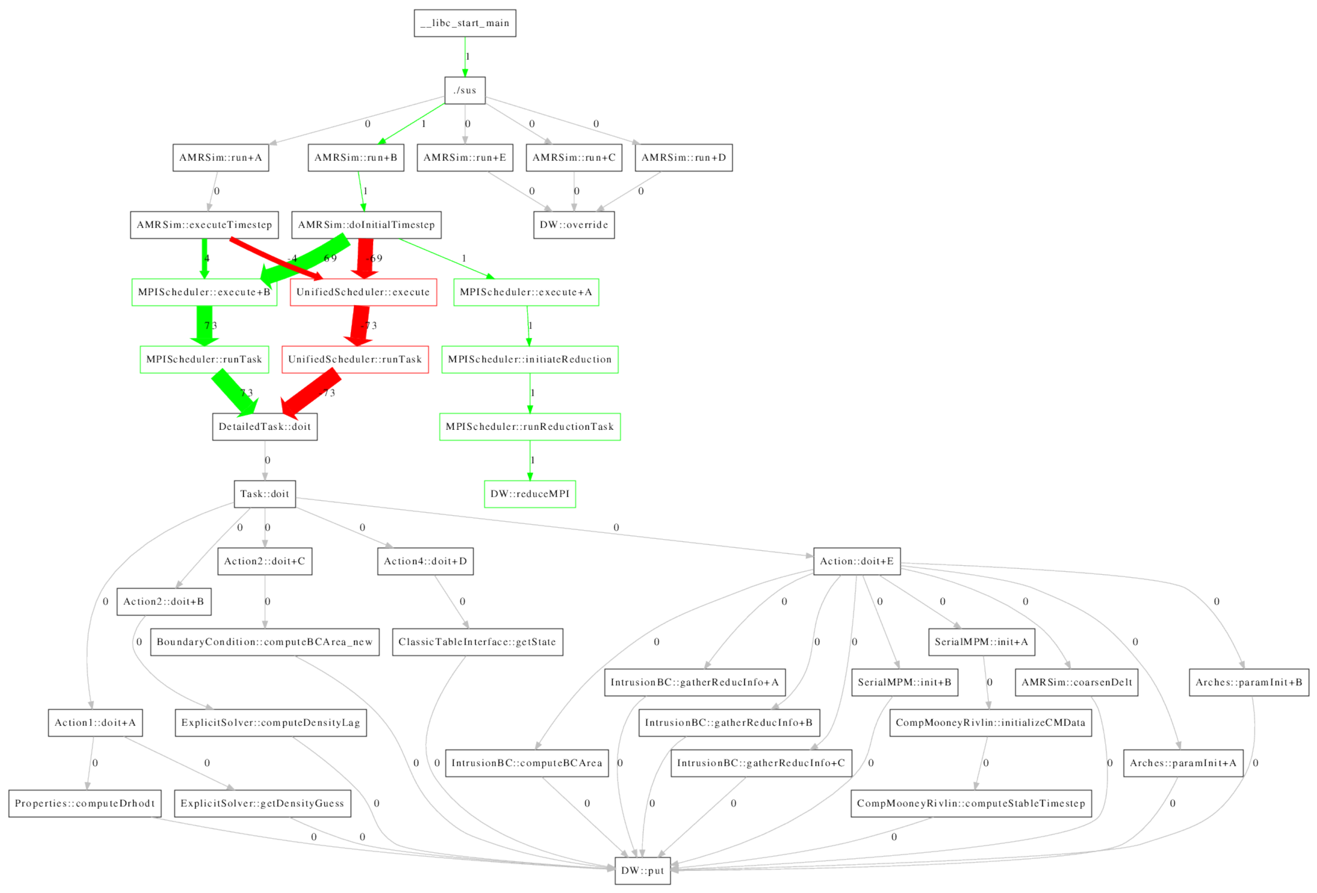

__libc_start_main
./sus
AMRSim::run+A
AMRSim::run+B
AMRSim::run+E
AMRSim::run+C
AMRSim::run+D
AMRSim::executeTimestep
AMRSim::doInitialTimestep
DW::override
MPIScheduler::execute+B
UnifiedScheduler::execute
MPIScheduler::execute+A
MPIScheduler::runTask
UnifiedScheduler::runTask
MPIScheduler::initiateReduction
DetailedTask::doit
MPIScheduler::runReductionTask
Task::doit
DW::reduceMPI
Action2::doit+C
Action4::doit+D
Action::doit+E
Action2::doit+B
BoundaryCondition::computeBCArea_new
ClassicTableInterface::getState
SerialMPM::init+A
IntrusionBC::gatherReducInfo+A
SerialMPM::init+B
AMRSim::coarsenDelt
Arches::paramInit+B
Action1::doit+A
ExplicitSolver::computeDensityLag
IntrusionBC::gatherReducInfo+B
CompMooneyRivlin::initializeCMData
IntrusionBC::computeBCArea
IntrusionBC::gatherReducInfo+C
Arches::paramInit+A
Properties::computeDrhodt
ExplicitSolver::getDensityGuess
CompMooneyRivlin::computeStableTimestep
DW::put

Fig. 4. Difference Graph. Highlights the differences from Fig. 3(a) and Fig. 3(b).

processes. CSTGs can be used to identify when a thread or process is not doing its assigned work properly by comparing it to other threads or processes, respectively.

As we can see, CSTGs can be used in many different scenarios not limited to the previous list. It is up to the user to identify interesting collection points according to what he or she wants to observe. The collection is made using a simple function call. Every time this function is called, a stack trace is collected and written to a file. The user can create variables and conditions to control when to start, pause, and stop the collection. Stack traces can be easily aggregated by different time periods, processes, or threads by creating one file per group of interest.

B. Understanding a Real Bug using CSTGs

Uintah simulation variables are stored in a data warehouse. The data warehouse is a dictionary-based hash-map which maps a variable name and patch id to the memory address of a variable. Each task can get its read and write variable memory by querying the data warehouse with a variable name and a patch id. The task dependencies of the task graph guarantee that there are no memory conflicts on local variables access, while variable versioning guarantees that there are no memory conflicts on foreign variables access. This means that a task's variable memory has already been isolated. Hence, no locks are needed for reads and writes on a task's variables memory. However, the dictionary data itself still needs to be protected when a new variable is created or an old variable is no longer needed by other tasks. While this increases the overhead of multi-threaded scheduling, locking on dictionary data is still more efficient than locking all the variables.

We performed an initial case study to assess the usability of CSTGs by using them to gain understanding of a real Uintah bug. When running Uintah on a particular input (mini_boiler.ups, available with the source code), an exception is thrown in the function *DW::get()* when looking for an element that does not exist in the data warehouse. One can think of two possible reasons why this element was not found: either it was never inserted, or it was prematurely removed from the data warehouse. Furthermore, the same error does not appear when using a different Uintah scheduler component.

We proceed by inserting stack trace collectors before every *put()* and *remove()* function of the data warehouse. Then, we run Uintah in turn with both versions of the scheduler, and collect stack traces visualized as CSTGs. Fig. 3(a) shows the CSTG of the working version, while Fig. 3(b) shows the CSTG of the crashing version.

It is not necessary to see all the details in these CSTGs.

However, it is apparent that there is a path to *reduceMPI()* in the working version that does not appear in the crashing version. Fig. 4 shows precisely that difference—the extra green path does not occur in the crashing version. (The other difference is related to the different names of the schedulers.) By examining the path leading to *reduceMPI()*, we are able to observe in the source code that the new scheduler never calls function *initiateReduction()* that would eventually add the missing data warehouse element that caused the crash. Since the root cause of this bug is quite distant from the actual crash location, relying on CSTGs enabled us to gain understanding of this bug faster then what we would have been able to achieve using only traditional debugging methods.

IV. Concluding Remarks

In this paper, we describe the Uintah Computational Framework's evolution, and its current capabilities. We argue the need for a rigorous approach to the software engineering of computational frameworks which have a long life-span of decades. Given the constant state of evolution of these frameworks in response to advances in software and hardware, it is essential to have the means to evolve the design and implementation of key components, and conduct differential verification across versions. Formal documentation, as an addition to existing guides, is another vital need, given the personnel changes that can occur during the lifetime of these frameworks. Without adequate tools for efficient debugging, HPC projects can become crippled, with their lead developers saddled with bugs that can take days or weeks to root-cause.

Following our recent successes, the collection and analysis of CSTGs will be the imminent focus of the URV project. In addition to straightforward approaches to compute differences between CSTGs, we are beginning to investigate other means of compressing the information contained in CSTGs and make the difference computation more insightful. For example, decorating CSTGs with information pertaining to locks may help identify concurrency errors pertaining to incorrect locking disciplines. We are also directing CSTG collection and analysis to target centrally important Uintah components, including the Data Warehouse.

One of the most tangible high-level outcomes of the URV project may be to lend credence to our strong belief that collaborations such as ours are possible, and are beneficial to both sides: to HPC researchers who gain an appreciation of CS formal methods; and to CS researchers who get a chance to involve in concurrency verification problems of a more fundamental nature that directly contributes to a nation's ability to conduct science and engineering research.

References

[1] "Uintah computational framework," http://www.uintah.utah.edu.

[2] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. de Supinski, M. Schulz, and G. Bronevetsky, "Formal analysis of MPI-based parallel programs," *Communications of ACM*, vol. 54, no. 12, pp. 82–91, Dec. 2011.

[3] D. C. B. de Oliveira, Z. Rakamarić, G. Gopalakrishnan, A. Humphrey, Q. Meng, and M. Berzins, "Practical formal correctness checking of million-core problem solving environments for HPC," in *Informal Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)*, 2013.

[4] A. Dubey, K. Antypas, M. K. Ganapathy, L. B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide, "Extensible component-based architecture for flash, a massively parallel, multiphysics simulation code," *Parallel Comput.*, vol. 35, no. 10-11, pp. 512–522, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2009.08.001

[5] P. Colella, D. Graves, N. Keen, T. Ligocki, D. Martin, P. McCorquodale, D. Modiano, P. Schwartz, T. Sternberg, and B. V. Straalen, "Chombo software package for AMR applications design document. Technical report," Lawrence Berkely National Laboratory, Applied Numerical Algorithms Group, Computational Research Division, Tech. Rep., 2009.

[6] J. Beckvermit, J. Peterson, T. Harman, S. Bardenhagen, C. Wight, Q. Meng, and M. Berzins, "Multiscale modeling of accidental explosions and detonations," *Computing in Science and Engineering*, vol. 15, no. 4, pp. 76–86, 2013. [Online]. Available: http://link.aip.org/link/?CSX/15/76/1

[7] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. A. Wight, and J. R. Peterson, "Uintah: A scalable framework for hazard analysis," in *Proceedings of the TeraGrid Conference*, 2010, pp. 3:1–3:8.

[8] D. Bedrov, J. B. Hooper, G. D. Smith, and T. D. Sewell, "Shock-induced transformations in crystalline rdx: A uniaxial constant-stress hugoniostat molecular dynamics simulation study," *The Journal of Chemical Physics*, vol. 131, no. 3, p. 034712, 2009. [Online]. Available: http://link.aip.org/link/?JCP/131/034712/1

[9] D. L. Brown and P. Messina, "Scientific grand challenges, crosscutting technologies for computing at the exascale," 2010, http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/crosscutting_grand_challenges.pdf.

[10] M. Emmi, S. Qadeer, and Z. Rakamarić, "Delay-bounded scheduling," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011, pp. 411–422.

[11] L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++*. MIT Press, 1996, pp. 175–213.

[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[13] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2007, pp. 1–10.

[14] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[15] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972463