Systematic Debugging Methods for Large-Scale HPC Computational Frameworks

Alan Humphrey, Qingyu Meng, Martin Berzins, Diego Caminha B. de Oliveira, Zvonimir Rakamaric, and Ganesh Gopalakrishnan | University of Utah

Parallel computational frameworks for high-performance computing are central to the advancement of simulation-based studies in science and engineering. Finding and fixing bugs in these frameworks can be time consuming. If left unchecked, these bugs diminish the amount of new science performed. A systematic study of the Uintah Computational Framework investigates debugging approaches, leveraging the framework's modular structure.

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 approaching a million cores in their current deployments and consisting of as much as a million lines of code.

The prevalence of software bugs in such large codes and the difficulty of debugging are well known. 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 code's parallel nature and for which testing might only occur through infrequently scheduled batch runs, possibly at large core counts.

This lengthy debugging process can arise even though computational framework creators contribute considerable effort and thought into carefully structuring these systems, while framework users write a nontrivial number of tests as well as assertions in their code. Part of the challenge in debugging HPC frameworks is that 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 orchestrated around time-stepped activities. Significant usage is made of infrastructural components (for example, schedulers), adaptive mesh refinement algorithms, as well as third-party libraries (for example, iterative solvers for large systems of linear equations). Compared to traditional software systems, researchers have paid relatively less attention to bugs occurring within HPC in general and computational frameworks in particular. However, this situation is rapidly changing. Recent work has provided a perspective on this issue for message passing parallel programs.[1] In further considering large software frameworks, we need steady progress in systematic testing methods that help trigger deeply hidden bugs, and systematic debugging methods that help observe these bugs and determine their root-cause.

This article presents our systematic study of the Uintah Computational Framework (see www.uintah.utah.edu) under development at the University of Utah, and the efforts we're putting into Uintah to debug it quickly and effectively. In particular, we summarize our preliminary results gained from an ongoing collaboration between this article's authors aimed at building a high-end problem-solving framework and developing formal software testing approaches that can help eliminate code-level bugs, hence enhancing the value offered by the framework.[2]

Our observation is that collaboration between HPC and computer science researchers is crucial in developing suitable rigorous software engineering approaches to modern computational frameworks. In this spirit, we're developing Uintah Runtime Verification (URV) techniques that can be deployed in field-debugging situations. We aim to make our results

 1521-9615/14/$31.00 © 2014 IEEE Copublished by the IEEE CS and the AIP

broadly applicable to other computational frameworks and HPC situations. While traditional debuggers (for example, Allinea DDT and Rogue Wave) 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. Our work is aimed at bringing in systematic (formal) techniques for both triggering bugs as well as debugging, which can be deployed in practice.

Uintah Computational Framework

A proven approach to solving large-scale multiphysics problems on large-scale parallel machines is to use computational frameworks such as the Uintah Computational Framework,[3] which originated at the University of Utah's Department of Energy (DoE) Center for the Simulation of Accidental Fires and Explosions. Researchers created Uintah to solve complex fluid-structure interaction problems on parallel computers. In particular, Uintah performs full physics simulations of fluid-structure interactions involving large deformations and phase changes. There might be 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 might include chemical or phase transformation between the solid and fluid phases. Uintah uses a full multimaterial approach in which each material is given a continuum description and is defined over the complete computational domain.

Uintah contains four main simulation components:

- the Implicit, Continuous-fluid Eulerian algorithm (ICE) code for both low and high-speed compressible flows;
- the multimaterial particle-based code Material Point Method (MPM) for structural mechanics;
- the combined fluid-structure interaction algorithm MPMICE; and
- the ARCHES turbulent reacting computational fluid dynamics (CFD) component 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 multiphysics simulation.

These Uintah components are C++ classes that follow a simple interface to establish connections with other system components. Uintah then uses 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 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 perform and the data dependencies between them.[4]

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 the message passing interface (MPI) or with parallelization in general. This is possible because an application-independent runtime system handles the parallel execution of the tasks. This division of labor between the application code and the runtime system lets developers of the underlying parallel infrastructure focus on scalability concerns such as load balancing; task scheduling; and communications, including accelerator or coprocessor interaction.

Uintah scales well on a variety of machines at small to medium scales (typically Intel or AMD processors with Infiniband interconnects) and on larger Cray machines such as Kraken and Titan. Uintah also runs on many other US National Science Foundation and DoE parallel computers (Stampede, Keeneland, Mira, and so on). Using its novel asynchronous task-based approach with fully automated load balancing Uintah demonstrates good weak and strong scalability up to 256,000 and 512,000 cores on DoE Titan and Mira, respectively. Full details of both these machines and Uintah's scalability are shown in our previous work.[4]

Uintah is used for a broad range of multiscale, multiphysics problems such as angiogenesis, tissue engineering, green urban modeling, blast-wave simulation, semiconductor design, and multiscale materials research. A recent example is the multiscale modeling of accidental explosions and detonations.[3]

One of the main approaches suggested for the move to multipetaflop 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.[5] Following this general direction, Uintah has

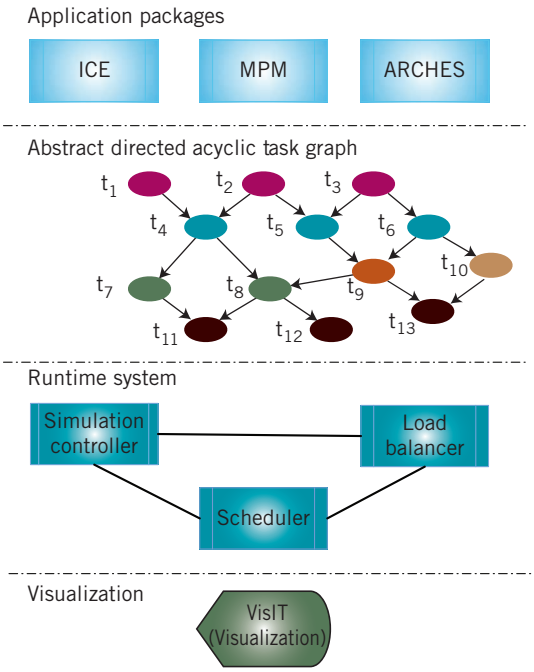

Figure 1. Outline of the Uintah architecture. The applications packages give rise to a directed task-graph, which, in turn, is executed by a runtime system.

evolved over the past decade, showing promising results on problems as diverse as fluid-structure interaction and turbulent combustion at scales of 500,000 CPU cores by incorporating shared memory (thread-based) schedulers as well as GPU-based schedulers.[4] Figure 1 shows the broad structure of Uintah, where the applications packages give rise to a directed task-graph, which, in turn, is executed by a runtime system. While this architecture has many advantages for scalability, its task-graph approach means that execution order varies from machine to machine, and with this the challenge of debugging increases.[4]

Frameworks such as Uintah are critically important components of our national HPC infrastructure, solving computationally challenging problems of great national consequence. Because these frameworks are based on sound and scalable organizational principles, they lend themselves to easy adaptation. 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 to stay at the leading edges of their subjects. There's always a shortage of CPU cycles, total memory capacity, network bandwidth, and advanced developer time. Structured software development and documentation are competing demands on expert designers' time, as much as is the demands to simulate new problems and achieve higher operating efficiencies by switching over to new machine architectures.

Previously, we've explored scalable, formal, debugging techniques for large-scale HPC and thread-based systems.[1,6] The URV project is different from these efforts because it attempts to integrate lightweight and scalable formal methods into a problem-solving environment that's undergoing rapid development and real usage at scale.

There are many active projects in which parallel computation is organized around task-graphs. For example, Charm++ (see http://charm.cs.uiuc.edu) has pioneered the task-graph approach and finds applications in high-end molecular dynamics simulations. Our interest in Uintah stems from two factors: Uintah has scaled by a factor of 1,000 in core-count over a decade and finds numerous real-world applications; and we're able to track its development and apply and evaluate formal methods in a judicious manner. We believe that our insights and results will transfer over to other existing and future computational frameworks.

Uintah Runtime Verification

The current focus of the URV project is enhancing the value of Uintah by eliminating showstopper code-level bugs as early as possible. In this connection, it's tempting to dismiss the use of lightweight and formal testing methods because many of these methods don't scale well, and many interesting field bugs occur only at scale. While this might be true in general, there are a number of bugs that are reproducible at lower scales and can be found by methods such as those presented in this article. This observation is supported by error logs from previous Uintah versions where many of the errors (for example, double-free of a lock and 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 large core counts with only batch access. Hence, they're clearly important and are the eventual goal of our future research.

In the URV project, we're motivated by one crucial observation: the ease with which we can downscale a system depends on how well it's structured. There are many poorly structured systems that allow only certain delicate combinations of their operating parameters; and sometimes, these parameters aren't 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 (related to problem sizes and the number of processes and threads) that must be respected. This gives us a fair amount of confidence that well-designed formal methods can be applied to Uintah at lower scales and that will detect many serious bugs.

Our main contribution in this article is our approach to debug large-scale parallel systems by highlighting the execution differences between the system's working and nonworking versions. 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 show that collecting CSTGs and diffing them is a practical approach by demonstrating how we've helped Uintah developers find the root-cause of 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 the Stack Trace Analysis Tool (STAT)[7,8] and HPCToolkit,[9] their focus hasn't been on cross-version (delta) debugging (as we've implemented).

Coalesced Stack Trace Graphs

A stack trace is a report of the active function calls at a certain point in time during program execution. Stack traces are commonly used to observe crashes and to learn where a program failed, being helpful in the debug phase of software development. They're also used in more advanced techniques to help find problems in parallel applications.

Collecting stack traces throughout program execution might 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 might be 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 CSTG, which is an aggregated view of stack traces recorded during an execution (see Figures 2a or 2b). We can view CSTGs as a summary of control flow paths (represented as function call sequences) in an execution.

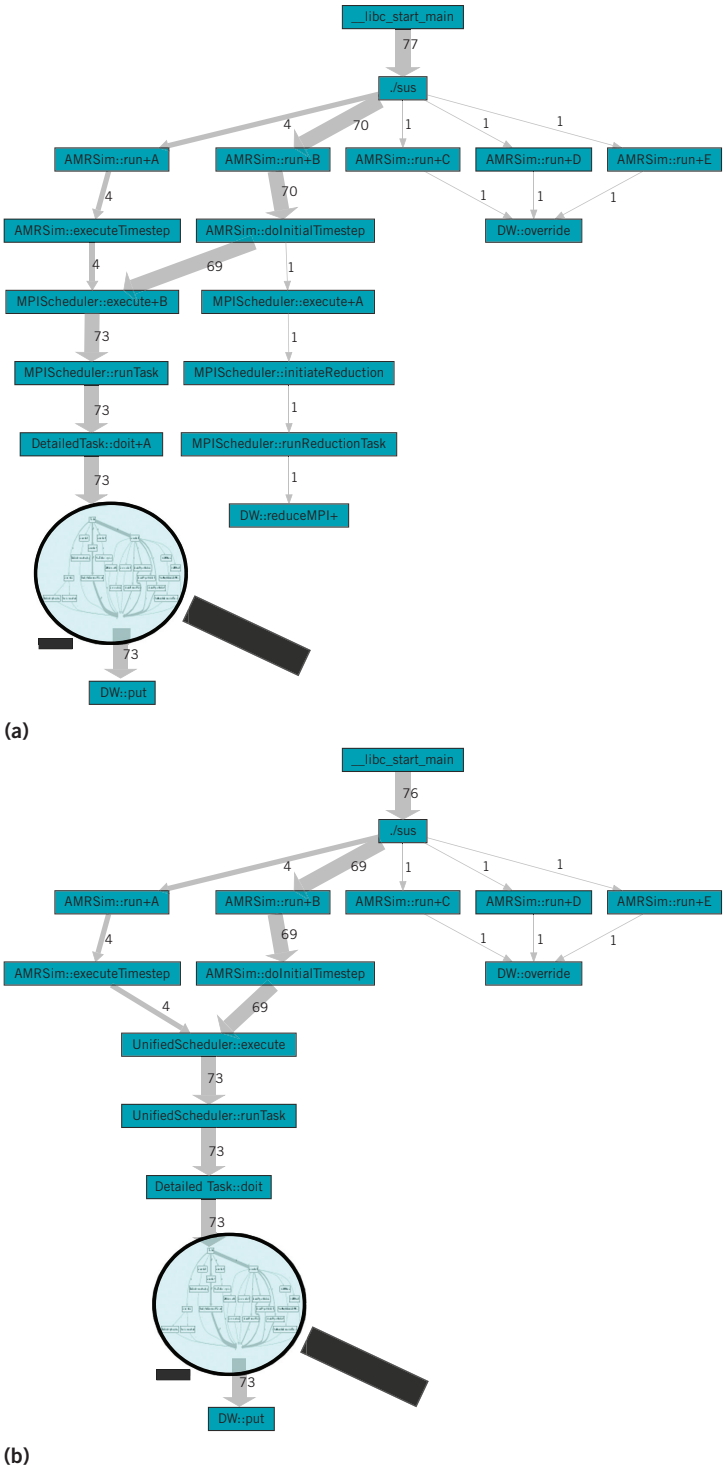

Figure 2. Using coalesced stack trace graphs (CSTGs) to understand a bug. The concave (shrinking) lens abstracts away irrelevant portions. CSTG for the (a) working and (b) crashing version.

Spectroscope collects stack traces to diagnose performance changes by comparing request flows.[10] Alternatively, STAT uses stack traces to present

```
void A()                {
       cstg.addStackTrace();
}
void B()                {

     A();
}
int   main() {
    int    x = random();
    if      (x > 0)  B();
    A();
}
```

Figure 3. Illustrative example of CSTGs. This example provides a mock-up of how we use CSTGs in the large scale.

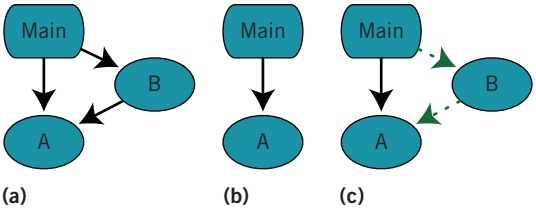

Figure 4. CSTGs of the illustrative example. (a) We obtain this CSTG from the full execution of the example when $x > 0$. (b) We obtain this CSTG when $x \leq 0$. (c) This graph helps understand the execution differences.

a summary view of the state of a distributed MPI program at a point of interest (often around hangs). STAT works by building equivalence classes of processes using stack traces, showing the split of these equivalence classes into divergent flows using a prefix tree. STAT corresponds well to the needs of MPI program debugging due to the single program, multiple data (SPMD) nature of MPI programs resulting in a prefix tree stem that remains coalesced for the most part. Debugging is accomplished by users noticing how process equivalence classes split off, and then understanding why some of the processes went with a different equivalence class. In our CSTG approach, we don't rely on MPI-like SPMD behaviors. CSTGs detect anomalies based on the general approach of comparing two different executions.

Simple Example Illustrating CSTGs

To illustrate how CSTG instrumentation occurs and how the collected stack traces are visualized as a graph, consider the simple example in Figure 3, which provides a mock-up of how we use CSTGs in the large scale. We can think of the `random()` call as complex piece of code that nondeterministically assigns `x`. Function `main()` conditionally calls `B()` if `x > 0`. Following this conditional, `main()` calls `A()`.

The collection of stack traces is done inside the function `A()`: every time `addStackTrace()` is executed, the nested stack of function calls leading to that point is recorded. After coalescing all the recorded stack traces together in a graph, there are two CSTGs that can be obtained from the full execution of this example, as shown in Figures 4a and 4b. Figure 4c shows a third graph that highlights the difference between the first two CSTGs.

In our actual debugging case studies using CSTGs, we expressed our knowledge of likely functions of interest in the Uintah code-base by inserting `cstg.addStackTrace()` calls into these functions. The CSTG tool does the rest automatically; it runs the example under test using different scenarios, produces CSTGs, and helps users see salient differences between the scenarios. The bug itself typically gets revealed and confirmed through the use of a traditional debugger, with the delta CSTGs providing significant focus and guidance in applying the debugger.

Stack Trace Viewing Modalities

We can roughly classify previous stack trace viewing methods[11] into three equivalence classes, as Figure 5 illustrates. In dynamic call trees, each node represents a single function activation. Edges represent calls between individual function activations. The size of a dynamic call tree is proportional to the number of calls in an execution. In dynamic call graphs, each node represents all function activations. Edges represent calls between functions. The size of a dynamic call graph grows with the number of unique functions invoked in an execution.

In calling context trees, each node represents a function activation in a unique call chain. Edges represent calls between function activations from different call chains. A calling context tree is a projection of a dynamic call tree that discards redundant contextual information while preserving unique contexts.

Different from the previously described structures, CSTGs don't record every function activation, but only the ones in stack traces leading to the user-chosen function(s) of interest. Each CSTG node represents all the activations of a particular function invocation. Hence, in addition to function names, CSTG nodes are also labeled with unique invocation IDs. Edges represent calls between functions. The size of a CSTG is determined by the number of different paths reaching the observations points (that is, target functions) of interest; in our experience, this size has been modest.

CSTG is a 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 compare different CSTGs. As examples we can cite the following:

- *Working and nonworking 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's supposed to do can be easier when comparing executions against the older working component.
- *Symmetric events (for example, 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 (for example, time steps).* It's common to find algorithms that behave the same (or 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 processes. CSTGs can identify when a thread or process isn't doing its assigned work properly by comparing it to other threads or processes, respectively.
- *Nondeterministic execution.* We've performed case studies (see www.cs.utah.edu/fv/CSTG) that demonstrate the feasibility of using CSTGs to locate and help find the root cause for the onset of nondeterminism.
- *Different inputs.* Sometimes changing the input of a program might cause a crash. Our studies show the success of CSTGs in this regard as well.

As you can see, CSTGs can be used in many different scenarios not limited to the previous list. Clearly, high-level user insights are important in governing where collection must occur. The collection itself is initiated by placing a special function call (as shown in Figure 3 as `cstg.addStackTrace()`, which is recorded). Users might additionally exercise various conditional collection features that we've provided in our CSTG package, as well as aggregating by different time periods, processes, or threads.

Understanding a Real Bug Using CSTGs

The case study we detail in this section investigates a real field bug that was present in an older version of Uintah. In conjunction with CSTGs, we also employed traditional techniques, such as the use of prints and a debugger (Allinea DDT)—albeit to a much reduced extent than in traditional debugging sessions. A person not involved with developing the Uintah code-base and who has only a limited knowledge of the overall Uintah code carried out all of the debugging. In this case study, we used CSTGs to compare a working and nonworking version of Uintah. It's a typical scenario of a system under constant development, in which a new component replacing an existing one causes a bug. Uintah source code, CSTG engine source code, presentations, and the full graphs of this and other case studies in different scenarios are available online at www.cs.utah.edu/fv/CSTG.

The mini coal boiler problem is a real-world example and models a smaller-scale version of the Predictive Science Academic Alliance Program (PSAAP) target problem, in which Uintah will use experimental data provided by industrial collaborator Alstom Power to simulate coal combustion under oxy-coal conditions.

Uintah simulation variables are stored in a data warehouse. The data warehouse is a dictionary-based hash-map that maps a variable name and patch ID to the memory address of a variable.

When running Uintah for solving the mini coal boiler problem, an exception is thrown in the function `DW::get()` when looking for an element that doesn't exist in the data warehouse. We can think of two possible reasons why this element

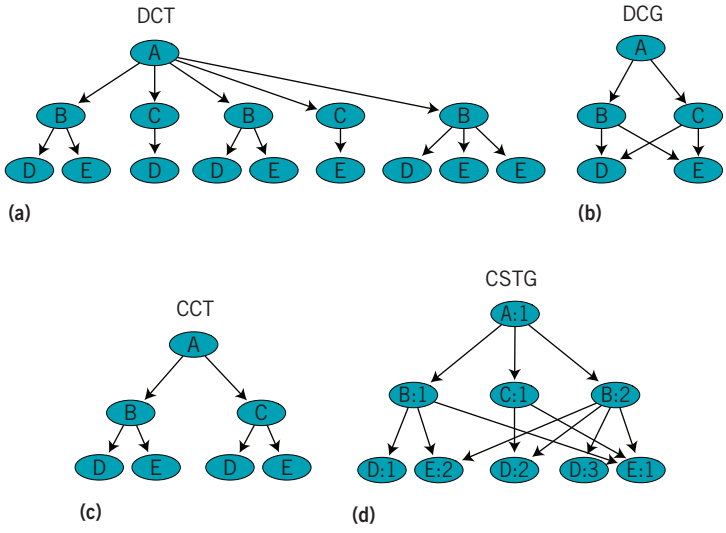

Figure 5. Different stack trace viewing methods: (a) dynamic call trees, (b) dynamic call graphs, (c) calling context trees, and (d) CSTG.

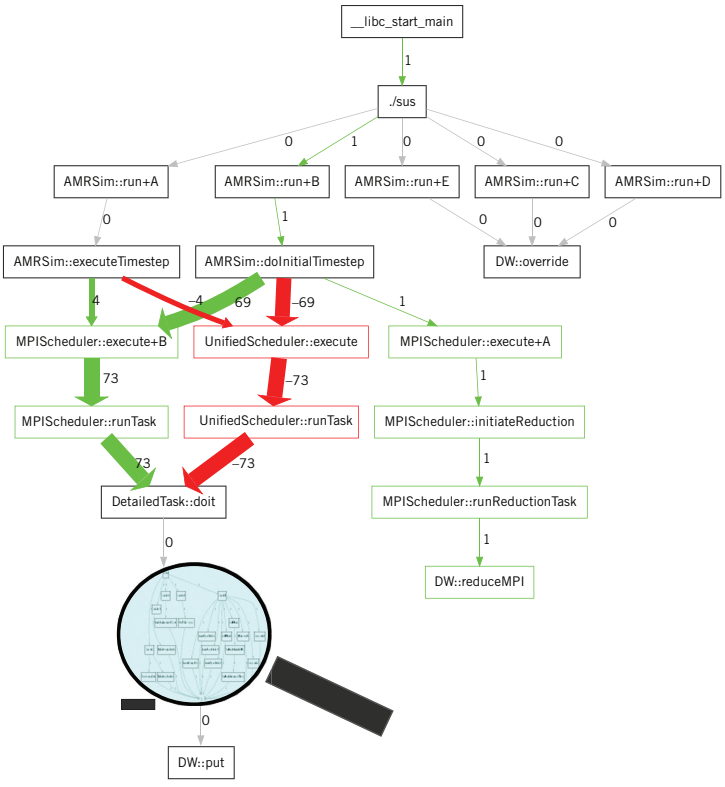

Figure 6. Difference graph. It's apparent that there's a path to `reduceMPI()` in the working version that doesn't appear in the crashing version, highlighting the differences from Figures 2a and 2b.

wasn't found: either it was never inserted, or it was prematurely removed from the data warehouse. Furthermore, the same error doesn't 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. Figure 2a shows the CSTG of the working version, while Figure 2b shows the CSTG of the crashing version.

It isn't necessary to see all the details in these CSTGs. However, it's apparent that there's a path to `reduceMPI()` in the working version that doesn't appear in the crashing version. Figure 6 shows precisely that difference—the extra green path doesn't 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 can 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. Because 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 than what we could achieve using only traditional debugging methods.

Implementation Details

In our current CSTG implementation in the context of Uintah running MPI on several nodes, we collect the stack traces separately at every processor (process) by invoking the `backtrace()` function (from C library *execinfo.h*) each time a stack trace collection instruction is executed. Stack traces can be written out to separate files and merged at the end of the execution for the generation of CSTGs offline. We have also recently added facilities to build CSTGs directly in memory. In this case, each stack trace is processed, added to a graph data structure and then discarded so memory overhead is minimal. An example of the current stack trace recorded is:

```
stack_trace:
MPIScheduler::postMPISends
  (Uintah::DetailedTask*, int)+0xa15
MPIScheduler::runTask
  (Uintah::DetailedTask*, int)+0x3b7
MPIScheduler::execute(int, int)+0x78f
AMRSimulationController::executeTimestep
  (double)+0x2a6
AMRSimulationController::run()+0x103b
StandAlone/sus() [0x4064d2]
__libc_start_main()+0xed
StandAlone/sus() [0x403469]
```

In this example, the first line starts with `stack_trace` that indicates where the stack trace starts. Each line in the stack trace is comprised of the complete function signature, plus a hexadecimal address indicating the calling context of the next called function. The graph is created using standard data structures and visualized using Graphviz. We compare CSTGs by creating a graph diff showing deficits as negative numbers (on red edges) and excesses as positive numbers (on green edges).

Scaling Studies

Figure 7 presents preliminary scaling studies of the viability of using CSTGs in the range of hundreds of processes. The case study itself was the one presented in the "Understanding a Real Bug Using CSTGs" section, with the same collection points. We performed the experiments in a cluster with 66 nodes, each node with four AMD Opteron (Magny-Cors) 6164HE 12-core 1.7-GHz CPUs,

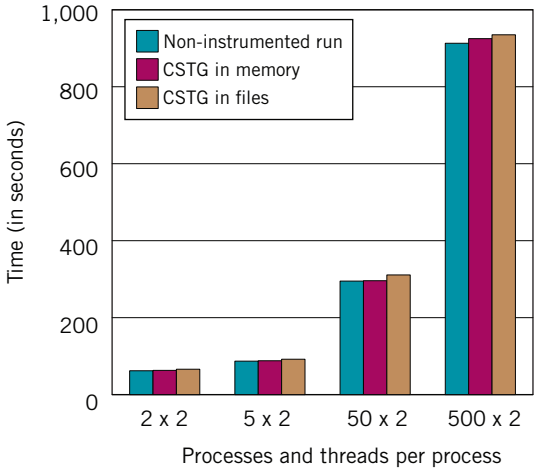

Figure 7. Running time collecting stack traces to generate CSTGs. The overhead of collecting stack traces is small (less than 5 percent on average), both when CSTGs are created in memory or when stack traces are recorded to files.

64 Gbytes of RAM, 7,200 RPM SATA2 hard drives, and 10-Gigabit Ethernet. The input file employed was one that doesn't produce a crash.

Figure 7 shows that the overhead of collecting stack traces is small (less than 5 percent on average), both when CSTGs are created in memory or when stack traces are recorded to files. (Clearly, in-memory collection eliminates interference with file I/O and the amount of memory used is minimal because we're mostly counting edges.) The run at the highest scale involved 127,000 stack traces, and the smaller runs 800; 7,000; and 35,000 stack traces. While the overhead will depend on the number of stack traces collected or where the instrumentation is placed, we believe that CSTGs do provide another tool for developers who might be able to easily downscale runs to hundreds of processes so that they can comprehend salient execution differences. While more experience is needed, our studies provide a growing body of evidence that CSTGs do work in practice (see www.cs.utah.edu/fv/CSTG).

In this article, we argue the need for a new approach to debugging large-scale software frameworks and demonstrate this approach in the context of the Uintah computational framework. Given the constant state of evolution of these frameworks in response to advances in software and hardware, it's essential to have the means to evolve the design and implementation of key components, and conduct differential verification across versions. A key need in the evolution of these frameworks is to have debugging tools that enhance the efficiency of the computational framework infrastructure developers when they're faced with tough debugging situations. 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 find the root cause. The CSTG approach described here is one way of improving the debugging of frameworks like Uintah.

Following the developments described in this article, 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're 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 might help identify concurrency errors pertaining to incorrect locking disciplines. We're 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 might 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 computer science's formal methods, and to computer science researchers who get to take part in concurrency verification problems of a fundamental nature that directly contribute to a nation's ability to conduct science and engineering research. ■

Acknowledgments

We thank the referees for their insightful comments. This work was supported by US National Science Foundation grant OCI-0721659 and NSF OCI PetaApps program grant OCI-0905068, as well as the US Department of Energy's National Energy Technology Laboratory under grant NET DE-EE0004449. This project used the University of Delaware's Chimera computer, which was funded by the US NSF Award CNS-0958512.

References

1. G. Gopalakrishnan et al., "Formal Analysis of MPI-Based Parallel Programs," *Comm. ACM*, vol. 54, no. 12, 2011, pp. 82–91.

2. D.C.B. de Oliveira et al., "Practical Formal Correctness Checking of Million-Core Problem Solving Environments for HPC," *Informal Proc. 5th Int'l Workshop Software Eng. for Computational Science and Eng.* (SE-CSE), 2013.
3. J. Beckvermit et al., "Multiscale Modeling of Accidental Explosions and Detonations," *Computing in Science & Eng.*, vol. 15, no. 4, pp. 76–86, 2013.
4. Q. Meng et al., "Investigating Applications Portability with the Uintah DAG-Based Runtime System on Petascale Supercomputers," *Proc. SC13: Int'l Conf. High Performance Computing, Networking, Storage and Analysis*, 2013; doi:10.1145/2503210.2503250.
5. 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.
6. M. Emmi, S. Qadeer, and Z. Rakamaric, "Delay-Bounded Scheduling," *Proc. ACM Sigplan-Sigact Symp. Principles of Programming Languages* (POPL), 2011, pp. 411–422.
7. D.C. Arnold et al., "Stack Trace Analysis for Large Scale Debugging," *Proc. Int'l Parallel and Distributed Processing Symp.*, 2007, pp. 1–10.
8. I. Laguna et al., "Large Scale Debugging of Parallel Tasks with AutomaDeD," *Proc. 2011 Int'l Conf. High Performance Computing, Networking, Storage and Analysis,* 2011, article no. 50.
9. L. Adhianto et al., "HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, 2010, pp. 685–701.
10. R.R. Sambasivan et al., "Diagnosing Performance Changes by Comparing Request Flows," *Proc. 8th Usenix Conf. Networked Systems Design and Implementation*, 2011, pp. 4–4.
11. G. Ammons, T. Ball, and J.R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *Proc. ACM Sigplan Conf. Programming Language Design and Implementation*, 1997, pp. 85–96.

Alan Humphrey is a software developer at the Scientific Computing and Imaging Institute and a PhD candidate in the School of Computing at the University of Utah. His research interests include high-performance computing, heterogeneous systems, and formal verification of concurrent systems, specifically the MPI. Humphrey has a BS in computer science from the University of Utah. Contact him at ahumphrey@sci.utah.edu.

Qingyu Meng is a research assistant in the Scientific Computing and Imaging Institute and a PhD candidate in the School of Computing at the University of Utah. His research interests include parallel computing and distributed runtime system. Meng has a BS in computer science from the University of Science and Technology of China. Contact him at qymeng@cs.utah.edu.

Martin Berzins is a professor of computer science in both the School of Computing and in the SCI Institute at the University of Utah. His research interests include mathematical software, numerical analysis, and parallel computing with application to challenging problems in science and engineering. Berzins has a PhD in mathematical software and numerical analysis from the University of Leeds. Contact him at mb@sci.utah.edu.

Diego Caminha B. de Oliveira is a postdoctoral researcher in the School of Computing at the University of Utah. His research interests include parallel computing, verification methods, and debugging techniques. de Oliveira has a PhD in computer science from the University of Nancy, France. Contact him at caminha@cs.utah.edu.

Zvonimir Rakamaric is an assistant professor in the School of Computing at the University of Utah. His research interests include improving the reliability and resilience of complex software systems by empowering developers with practical tools and techniques for analysis of their artifacts. Rakamaric has a PhD from the Department of Computer Science at the University of British Columbia, Canada. He's a recipient of the Microsoft Research Software Engineering Innovation Foundation Award, Microsoft Research Graduate Fellowship, and Silver Medal in the ACM Student Research Competition. Contact him at zvonimir@cs.utah.edu.

Ganesh Gopalakrishnan is a professor of computer science and is the director of the Center for Parallel Computing at the University of Utah. His research interests include verification methods and tool frameworks for parallel and concurrent systems, formal techniques to enhance system resilience, and combined static/dynamic correctness analysis for structured parallelism. Gopalakrishnan has a PhD in computer science from Stony Brook University. Contact him at ganesh@cs.utah.edu.