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 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 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. 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 are orchestrated around time-stepped activities. Significant usage is made of infrastructural components (*e.g.*, schedulers), adaptive mesh refinement algorithms, as well as third-party libraries (*e.g.*, iterative solvers for large systems of linear equations). 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 for message passing parallel programs [1]. In further considering large software frameworks 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* [2] (or just Uintah), and the efforts we are putting into Uintah in order to debug its bugs quickly and effectively.

In particular, we summarize preliminary results [3] from an ongoing collaboration between a subset of its authors interested in building a high-end problem solving framework, and a subset interested in developing *formal* software testing approaches that can help eliminate code-level bugs, hence enhancing the value offered by the framework. Our observation is that *collaboration between HPC and core CS researchers is crucial in developing suitable rigorous software engineering approaches to modern computational frameworks*. In this spirit 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. Our work is aimed at bringing in

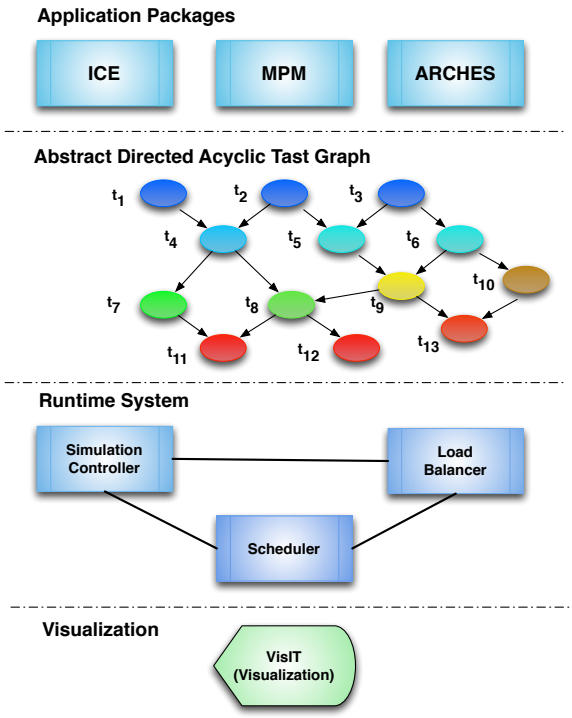


Fig. 1. Outline of Uintah Architecture

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. The key abstraction used in this approach is that of Coalesced Stack Trace Graphs (CSTGs). In the rest of this paper, we describe the form and use of CSTGs in connection with Uintah.

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 the Uintah Computational Framework[1] which originated in the University of Utah DOE Center for the Simulation of Accidental Fires and Explosions (C-SAFE) (9/97-3/08). 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. There may 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 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 [5].

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. Full details of both these machine and the scalability of Uintah are shown in our previous work [5].

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 multi-scale modeling of accidental explosions and detonations [4].

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

[1]For reasons of space, we cite here an earlier CiSE paper on the applications of Uintah [4]; we refer the reader to that paper for more information and references.

communication follow blocks of computation. The importance of this approach for exascale computing is expressed by recent studies [6]. Following this general direction, Uintah has evolved over the past decade to show promising results on problems as diverse as fluid-structure interaction and turbulent combustion at scales of about 500K CPU cores by incorporating shared memory (thread-based) schedulers as well as GPU-based schedulers [5]. The broad structure of Uintah is shown in Fig. 1, where the applications packages give rise to a directed task-graph which, in turn, is executed by a run-time 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 [5].

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. 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. 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., [1], [7]). 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. For example, Charm++ [8] has pioneered the task-graph approach and finds applications in high-end molecular dynamics simulations. 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 will transfer over to other similar computational frameworks—in existence or planned.

III. Uintah Runtime Verification (URV)

The current focus of the Uintah Runtime Verification project 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 light-weight 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 and can be found by such methods, as presented in this paper. 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 their 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 (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 to detect many serious bugs (examples are provided later in §III).

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 show 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 [9], [10] and HPCToolkit [11], their focus has not been on cross-version ("delta") debugging as we have implemented.

A. Coalesced Stack Trace Graphs (CSTG)

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.

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

void B() {
  A();
}

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

Listing 1. Illustrative Example of CSTGs

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 an aggregated view of stack traces recorded during an execution (see Fig. 4(a) or 4(b)). One may view CSTGs as a summary of control flow paths (represented as function call sequences) in an execution.

Spectroscope [12] collects stack traces to diagnose performance changes by comparing request flows. Alternatively, STAT [9] uses stack traces to present a summary view of the state of a distributed MPI message-passing 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 SPMD (single program, multiple data) 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 do not rely upon MPI-like SPMD behaviors. CSTGs detect anomalies based on the general approach of comparing two different executions.

B. Simple Example Illustrating CSTGs

To illustrate how CSTG instrumentation is done and how the collected stack traces are visualized as a graph, consider the simple example in Listing 1 that provides a mock-up of how we use CSTGs in the large. The `random()` call may be thought to be a complex piece of code that non-deterministically 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 Fig. 2. In the figure, we also show a third graph that highlights the difference between the first two CSTGs.

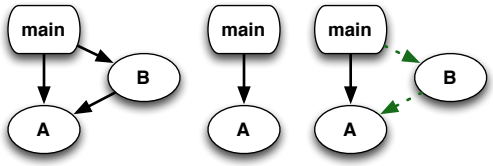


Fig. 2. CSTGs of the illustrative example. The CSTG on the left is obtained from the full execution of the example when `x > 0`. The middle CSTG is obtained when `x <= 0`. The difference graph on the right helps understand the execution differences.

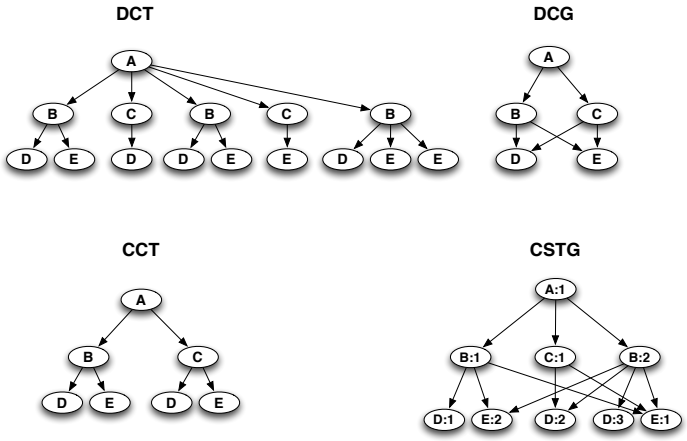


Fig. 3. Different Stack Trace Viewing Methods

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 under different scenarios (detailed in §III-C), produce CSTGs, and help 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.

C. Stack Trace Viewing Modalities

One can roughly classify previous stack trace viewing methods [13] into three equivalence classes as illustrated in Fig. 3. In Dynamic Call Trees (DCT), each node represents a single function activation. Edges represent calls between individual function activations. The size of a DCT is proportional to the number of calls in an execution. In Dynamic Call Graphs (DCG), each node represents all function activations. Edges represent calls between functions. The size of a DCG grows with the number of unique functions invoked in an execution.

In Calling Context Trees (CCT), each node represents a function activation in a unique call chain. Edges represent calls between function activations from different call chains. CCT is a projection of a dynamic call tree that discards redundant contextual information while preserving unique contexts.

Different from the previously described structures, CSTGs do not 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 (i.e., target functions) of interest; in our experience, this size has been modest.

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.

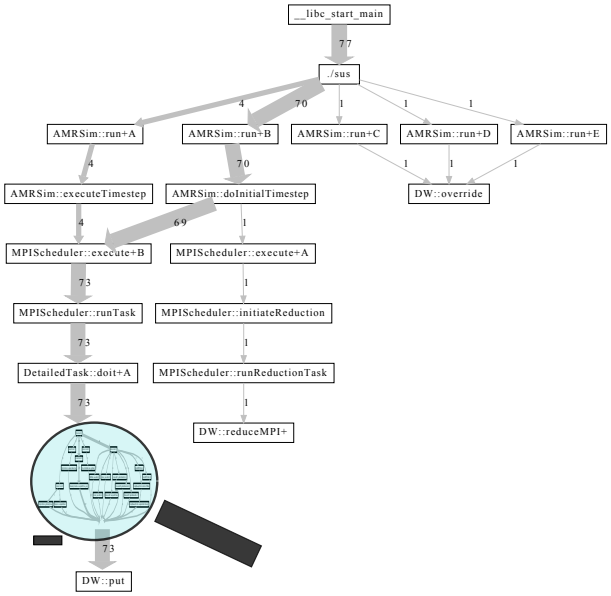
Different processes and threads. In many parallel programs, the same work is done in different threads or processes. CSTGs can be used to identify when a thread or process is not doing its assigned work properly by comparing it to other threads or processes, respectively.

Non-deterministic execution. We have performed case studies[2] that demonstrate the feasibility of using CSTGs to locate and help root-cause the onset of nondeterminism.

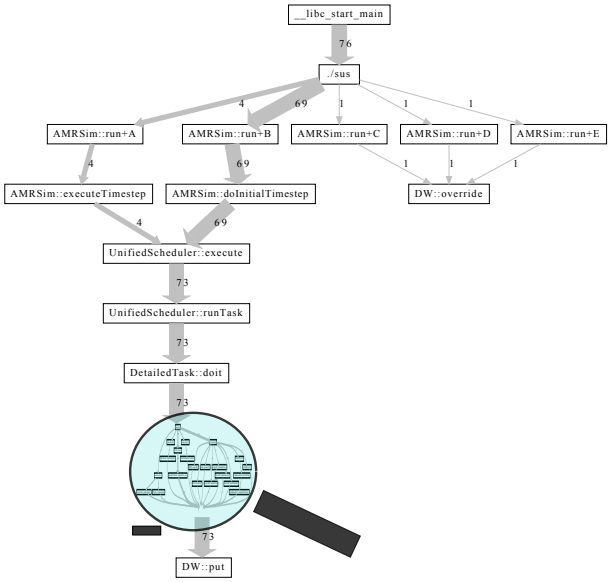
Different inputs. Sometimes changing the input of a program may cause a crash. Our studies show the success of CSTGs in this regard as well.

As we 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 (such as illustrated in Listing 1 as `cstg.addStackTrace()`, calls to which are recorded). Users may additionally exercise various conditional collection features we have provided in our CSTG package, as well as aggregate by different time periods, processes, or threads.

[2]Details at www.cs.utah.edu/fv/CSTG/. For the ease of presentation, we simplify many of the function and variable names involved, and zoom out irrelevant parts of the presented CSTGs.



(a) CSTG for the Working Version



(b) CSTG for the Crashing Version

Fig. 4. Using CSTGs to Understand a Bug. The concave (shrinking) lens abstracts away irrelevant portions of the CSTG

IV. 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

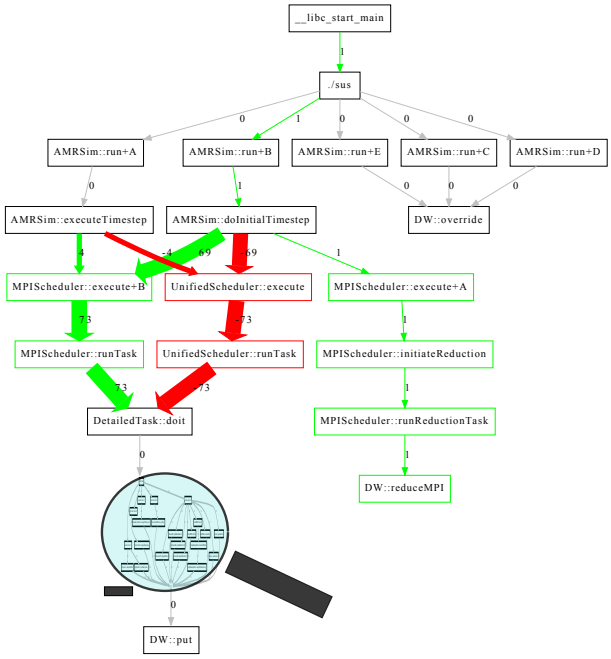


Fig. 5. Difference Graph. Highlights the differences from Figs. 4(a) and 4(b).

debugging sessions. All the debugging was carried out by a non-developer of the Uintah code-base who has only a very limited knowledge of the overall Uintah code. In this case study CSTGs were used to compare a working and non working version of Uintah. It is 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.

The Mini Coal Boiler problem is a real-world example and models a smaller scale version of the *PSAAP* target problem in which Uintah will use experimental data provided by an 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 which 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 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. 4(a) shows the CSTG of the working version, while Fig. 4(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. 5 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.

V. Implementation Details

In our current implementation of CSTGs 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).

VI. Scaling Studies

Figure 6 presents preliminary scaling studies of the viability of using CSTGs in the range of 100s of processes. The case study itself was the one presented in Section IV, with the same collection points. The experiments were performed in a cluster with 66 nodes, each node with a 4 AMD Opteron "Magny Cors" 6164HE 12-core 1.7GHz CPUs, 64 GB RAM, 7200RPM SATA2 Hard drives and 10 Gigabit Ethernet. The input file employed was one that does not produce a crash.

Figure 6 shows that the overhead of collecting stack traces is small (less than 5% on average), both when CSTGs are

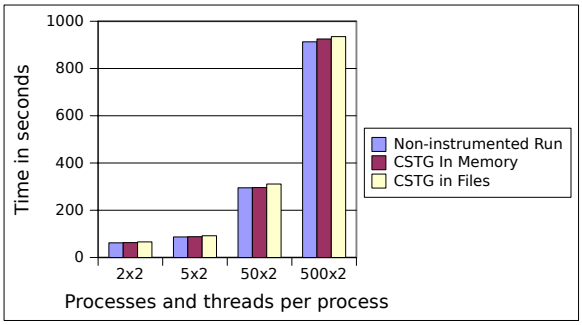


Fig. 6. Running time collecting stack traces to generate CSTGs.

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 are mostly counting edges). The run at the highest scale involved 127,000 stack traces, and the smaller runs 800, 7000 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 in the tool-kit of developers who may be able to easily downscale runs to 100s of processes so that they may comprehend salient execution differences. While more experience is needed, our studies in www.cs.utah.edu/fv/CSTG/ provide a growing body of evidence that CSTGs do work in practice.

VII. Concluding Remarks

In this paper, 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 is 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 are 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 root-cause. The CSTG approach described above is one way of improving the debugging of frameworks like Uintah.

Following the developments described above, 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.

Acknowledgements

The authors wish to thank the referees for their insightful comments. This work was supported by the National Science Foundation under grants OCI-0721659, the NSF OCI PetaApps program, through award OCI 0905068 and DOE NETL for funding under NET DE-EE0004449. This project used the University of Delaware's Chimera computer which was funded by the U.S. National Science Foundation Award CNS-0958512.

References

[1] 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.

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

[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] 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.

[5] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Investigating applications portability with the Uintah DAG-based runtime system on petascale supercomputers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013, pp. 96:1–96:12.

[6] 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.

[7] 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.

[8] http://charm.cs.uiuc.edu/.

[9] 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.

[10] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, and B. Rountree, "Large scale debugging of parallel tasks with AutomaDeD," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 50:1–50:10.

[11] 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.

[12] 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 USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011, pp. 4–4.

[13] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1997, pp. 85–96.