Systematic Debugging of Concurrent Systems Using Coalesced Stack Trace Graphs

Diego Caminha B. de Oliveira[1], Zvonimir Rakamarić[1], Ganesh Gopalakrishnan[1], Alan Humphrey[2], Qingyu Meng[2], and Martin Berzins[2]

[1] School of Computing
University of Utah, USA
{caminha, zvonimir, ganesh}@cs.utah.edu
[2] School of Computing and SCI Institute
University of Utah, USA
{ahumphre, qymeng, mb}@cs.utah.edu

Abstract. A central need during software development of large-scale parallel systems is tools that help help to identify the root causes of bugs quickly. Given the massive scale of these systems, tools that highlight changes—say introduced across software versions or their operating conditions (e.g., inputs, schedules)—can prove to be highly effective in practice. Conventional debuggers, while good at presenting details at the problem-site (e.g., crash), often omit contextual information to identify the root causes of the bug. We present a new approach to collect and coalesce stack traces, leading to an efficient summary display of salient system control flow differences in a graphical form called Coalesced Stack Trace Graphs (CSTG). CSTGs have helped us understand and debug situations within a computational framework called Uintah that has been deployed at large scale, and undergoes frequent version updates. In this paper, we detail CSTGs through case studies in the context of Uintah where unexpected behaviors caused by different versions of software or occurring across different time-steps of a system (e.g., due to non-determinism) are debugged. We show that CSTG also gives conventional debuggers a far more productive and guided role to play.

1 Introduction

There is widespread agreement that software engineering principles, including systematic debugging methods, must be brought to bear on high-performance computing (HPC) software development. HPC frameworks form the backbone of all science and engineering research, and any savings in the effort to locate and fix bugs in the short term, and maintain their integrity over the decades of their lifetime maximizes the "science per dollar" achieved.

Formal analysis methods such as model-checking, symbolic analysis, and dynamic formal analysis [12] are among the plethora of recent efforts addressing this need. With the growing scale and complexity of systems, most of these techniques are not applicable on real deployed software, and hence of no direct value

to people in the debugging trenches of advanced HPC software—especially those building computational frameworks.

Large-scale computational frameworks for are in a state of continuous development, in response to new user applications, larger problem scales, as well as new hardware platforms and software libraries. Tools that help expediently root-cause bugs are crucially important. Conventional HPC-oriented debuggers have made impressive strides in recent years (e.g., DDT [9] and RogueWave [19]).Unfortunately, debugging is never a linear story: the actual bug manifestation (e.g., a crash) may often have very little to do with the instructions present at the crash site. A designer, in general, needs far more contextual information before a bug is root-caused and corrected. Research on such error localization tools has made inadequate progress compared to the growing needs of this area.

This paper makes a contribution in this area by proposing a simple, yet versatile methodology for locating bugs with a useful amount of contextual information. Called *Coalesced Stack Trace Graphs (CSTG)*, our mechanism offers a succinct graphical display of a system execution focused on call paths to a set of target functions chosen by a user. Using CSTGs involves three-steps: (1) a user chooses target functions where stack trace collectors are automatically inserted, (2) our CSTG tool records the system behavior executing the inserted collectors in these functions, and (3) succinctly displays the differences between two such recordings over two scenarios. Typically, the target functions g_i are chosen based on the scenario/bug under investigation (e.g., the g_i could be an MPI messaging call or a hash-table insert call). Stack traces are then recorded over a user-chosen period of the system run. Each such stack trace is a nest of function calls $f_1, f_2, \ldots, g_k$ for some target function g_k. These call chains are merged whenever we have the situation of f_i calling f_{i+1} from the same calling context (i.e., program counter location).

Figs. 3 and 4 show CSTGs and their usages, which will be explained in much greater detail in the coming sections. In particular, we illustrate how CSTGs have helped during the development and analysis of Uintah [11], an open-source extensible software framework for solving complex multiscale multi-physics problems on cutting edge HPC systems.

We present the following real bug case studies: (1) a problem in which a hash-table lookup error was localized to a scheduler error, (2) an non-deterministic crash which revealed no issues at the crash site (3) another crash we explored using two different inputs applied to the same system version, and (4) an issue debugged to be caused by mismatching MPI sends and receives. These case studies show in detail how CSTGs have been used so far to better understand (and in many cases root-cause) these bugs.

2 Related Work

(A preliminary version of this paper for an HPC-oriented audience has appeared in [14]; this paper is an extended version.)

A stack trace is a sequence of function calls active at a certain point in the execution of a program. Stack traces are commonly used to observe crashes

and identify likely causes. There are empirical evidences that show that they help developers to fix bugs faster [21]. They are also often leveraged in parallel debugging. For instance, STAT [2] uses stack traces to present a summary view of the state of a distributed program at a point of interest (often around hangs). It 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.

Spectroscope [20] is another tool based on stack trace collection, where the emphasis is on performance anomaly detection. It works by comparing request flows (i.e., paths taken by the requests and time separations) across two executions. There are also efforts around the Windows Error Reporting system which helps analyze crash logs [3, 8, 13, 15]. More recent anomaly detection methods involve clustering and machine learning, and have located errors by identifying the *least progressed* of threads [7].

Synoptic [5] and Dynoptic [6] are tools that mine a model of the system from system execution logs. Synoptic mines a finite state machine (FSM) model representations of a sequential system from its logs capturing event orderings as invariants. Dynoptic mines a communicating FSM model from a distributed system. These systems have not been applied to large-scale code bases such as Uintah, and have not been demonstrated with respect to various modalities of differential exploration that we have investigated.

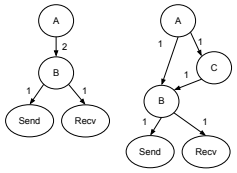

Fig. 1. CSTGs of a Simple Working (left) and Crashing (right) Program Runs

CSTGs are effectively used on anomaly detection based on the general approach of comparing two different executions or program versions, also known as *delta debugging* [24]. There are tools that spot behavioral differences such as RADAR [17]. However, they use different data and visualization methods. One novelty of our proposed method is using stack traces as the main source of information in the delta debugging process. Imagine a simple scenario where a program is run twice, the first time successfully, but the second time crashing just after making MPI `Send` and `Recv` calls. Fig. 1 summarizes these executions in terms of a CSTG. The differences between the call paths leading to the `Send` and `Recv` are highly likely to play a significant role in identifying the root-cause of the bug, as our case studies show later.

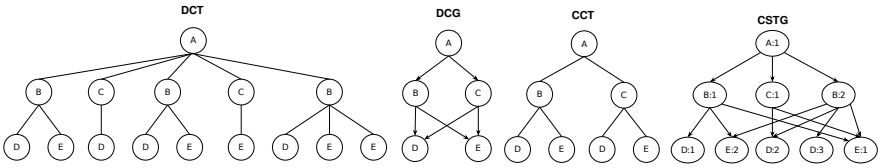

Fig. 2. Different Stack Trace Viewing Methods

2.1 Stack Trace Structures

One can roughly classify previous stack trace structures [1] into three classes as illustrated in Fig. 2. 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.

3 Coalesced Stack Trace Graphs (CSTG)

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 (see Fig. 2). 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 nodes (function call sites) encountered across the paths reaching the observation points of interest. In our experience, this size has been modest.

The feature of not recording every function activation is crucial to reduce the overhead and improve scalability. Many stack-trace-based methods, especially those used for performance measurement, employ sampling and record only a small percentage of all function activations. However, given that debugging is our end goal, doing sampling and not recording every call can potentially result in crucial information being missed. Hence, we choose to record every stack trace leading to the few user-chosen functions of interest. We also found such user input to be crucial in taming CSTG size since typically tracking only a small portion of program functions suffices to understand and root-cause a bug. Tracking all function invocations from a large program in a CSTG graph would significantly increase the effort necessary to understand the collected graphs, not to mention the difficulty of root-causing bugs.

There are many scenarios where CSTGs can be used, some of which are illustrated in this paper. Here we list some of these scenarios. Using CSTGs, one can make comparisons accross different:

- versions of a system (§5.1, Case Study 1);
- runs of the same system to understand the effects of nondeterminism (§5.1, Case Study 2);
- inputs (§5.1, Case Study 3);
- matching events such as allocate/free, open/close, lock/unlock, send/receive (§5.2, Scenario 1);
- time-steps, loop iterations, or cycling events;
- processes and threads.

3.1 Formal Definition

Consider an execution of an arbitrary sequential or concurrent program with a set of instructions $\{p_1, p_2, \ldots, p_N\}$ that collect stack traces. Executing an instruction p_j returns a snapshot s_k of the currently active call stack, i.e., it returns a stack trace. Each s_k is a stack of function names paired with their calling context[3]: $s_k = \langle f_1 : c_1\rangle, \langle f_2 : c_2\rangle, \ldots, \langle f_{Top-1} : c_{Top-1}\rangle, \langle f_{Top} : c_{Top}\rangle$. Here, for every $\langle f_i : c_i\rangle$ in s_k, where $i \neq Top$, c_i is the context (i.e., line number) within function f_i from which f_{i+1} is invoked. f_{Top} is the function within which an instruction p_j occurs and c_{Top} is context within function f_{Top} from which the instrumentation p_j is executed. Now, given the stack traces $s_1, s_2, \ldots, s_M$, we define a coalesced stack trace graph (CSTG) over these traces as follows:

- For each $\langle f_i : c_i\rangle$ present in some stack trace s_k, introduce the node $\langle f_i : c_i\rangle$. This node represents all the $\langle f_i : c_i\rangle$ instances present across the stack traces.
- Whenever a stack trace has two adjacent entries $\langle f_{i-1} : c_{i-1}\rangle$ and $\langle f_i : c_i\rangle$, introduce a directed edge from the latter to the former, weighted by the number of times such edge occurs across all stack traces.

As an illustration, in Fig. 3 there were 76 (left CSTG) and 77 (right CSTG) stack traces collected during the respective executions. The stack trace collection instructions were injected into functions `DW::put()`, `DW::reduceMPI()` and `DW::override()`. Each $\langle$*function* : *context*$\rangle$ pair only appears once in the CSTG (some contexts were omitted for simplicity); functions do appear more than once when they have different contexts (e.g., `AMRSim::run()`). Edge weights represent the number of times a consecutive pair of elements occurred across all stack traces.

4 Driving Example: Uintah HPC Framework

Uintah serves as a non-trivial test-bed for our work, given its complexity and continuing development both in terms of applications and in terms of parallel scalability. Uintah is an open-source, extensible software framework for solving complex multiscale multi-physics problems on cutting edge HPC systems [22]. The

[3] Say, the line number where the call to the next function in the stack is made or the instrumentation code is found.

class of problems solved by Uintah includes fluid, structure, and fluid-structure interaction problems, both with and without adaptive mesh refinement [4]. The framework has been in constant development and improvement over the past 10 years and now has about 1M lines of code and comments, and runs in a scalable manner on machines such as DOEs Titan at Oak Ridge National Laboratory and Mira at Argonne National Laboratory [16].

To promote reuse, easier maintenance, and extensibility, the developers of Uintah adopted component-based software engineering approach early on. A component-based design of Uintah enforces separation between large entities of software that can be swapped in and out, allowing them to be independently developed and tested within the entire framework. In addition, such modular software architecture promotes a clear separation of domain expert and infrastructure developer concerns.

Uintah employs a task-graph-based specification of the (sequential) user application code that is executed via a highly parallel task-based runtime system. Such an approach enables domain expert users to focus on what they know best, which is implementing sequential simulations components. At the same time, the runtime system can be independently improved without changing the user code as it relies only on the task abstraction and not on the details of what the tasks actually do. This distinction is important as it allows the parallel infrastructure components to be improved by computer scientists, who do not have to understand the simulation components in detail.

5 Case Studies

5.1 Root Cause Analysis of Uintah Bugs

The case studies we detail in this section focus on root-causing real bugs present in previous versions of Uintah. We leveraged traditional techniques, such as the use of `printf`s and a debugger (Allinea DDT [9]), in conjunction with CSTGs during the debugging process. All the debugging was carried out by a non-developer of the Uintah code-base who only had very limited knowledge of the overall Uintah code. The source code of Uintah and our CSTG-based tool, as well as the full graphs of our case studies are available online.[4]

Case Study 1: Mini Coal Boiler The *Mini Coal Boiler* problem is a real-world example modeling a smaller-scale version of the PSAAP [18] target problem that simulates coal combustion under oxy-coal conditions. This case study illustrates a typical scenario of a system under constant development in which a new component replacing an existing one causes a bug. We root-caused this bug using CSTGs to compare different versions of this Uintah simulation.

Uintah simulation variables are stored in a data warehouse. The data warehouse is a dictionary-based hash-map which maps a variable name and simulation

[4] `http://gdurl.com/pxPm/download`. For the ease of presentation, we simplify many of the function and variable names involved.

```
void DW::get(ReductionVariableBase& var,
             const VarLabel* label,
             const Level* level,
             int matlIndex /*= -1*/) {
  ...
  if(!d_levelDB.exists(label, matlIndex, level)) {
    THROW(UnknownVariable(label->getName(),
        getID(), level, matlIndex, "on reduction",
        __FILE__, __LINE__));
  }
  ...
}
```

Listing 1.1. Uintah Code Excerpt where the Mini Coal Boiler Exception is Thrown.

patch id to the memory address of a variable. When running Uintah on the *Mini Coal Boiler* problem, an exception is thrown in the data warehouse function `DW::get()`. After studying the code (see Listing 1.1), we discovered that the problem is caused by the triple `(label, matlIndex, level)` not being found in the hash table `d_levelDB`. However, the same error does not occur when using a different Uintah scheduler component.

At such a juncture, it is quite likely that an HPC developer equipped with a debugger such as DDT or RogueWave would derive no benefit from the power and sophistication of the debugger. They would likely have to fall back to using `printf`s. We show that CSTGs offer a better path.

One can think of two possible reasons why this element was not found in the data warehouse: either it was never inserted, or it was prematurely removed from it. With this line of investigation in mind, we proceed by inserting our CSTG stack trace collectors before every `put()` and `remove()` call of the hash table `d_levelDB`. Whenever one of these locations is reached during an execution, a stack trace is collected to create a CSTG. The leaves of the generated CSTGs are unique places where the collectors were added. We run Uintah twice, each time with a different version of the scheduler (i.e., buggy and correct), and collect stack traces visualized as CSTGs. Fig. 3 shows the CSTGs of the working and crashing executions.

It is not necessary to see all the details[5] in these CSTGs: it is apparent that there is a path to `reduceMPI()` in the working execution that does not appear in the crashing one. Fig. 4 focuses on that difference—the extra green path does not occur in the crashing execution. (The other difference is related to the different names of the schedulers.) By examining the path leading to `reduceMPI()`, we observe in the source code that the new, buggy scheduler never calls function `initiateReduction()` that would eventually add the missing data warehouse element causing the crash. Since the root cause of this bug is distant from the actual crash location, traditional debugging methods would not have been able to offer such useful contextual information that CSTGs provided.

[5] The zoomed out region of the CSTGs contains no information relevant for this study.

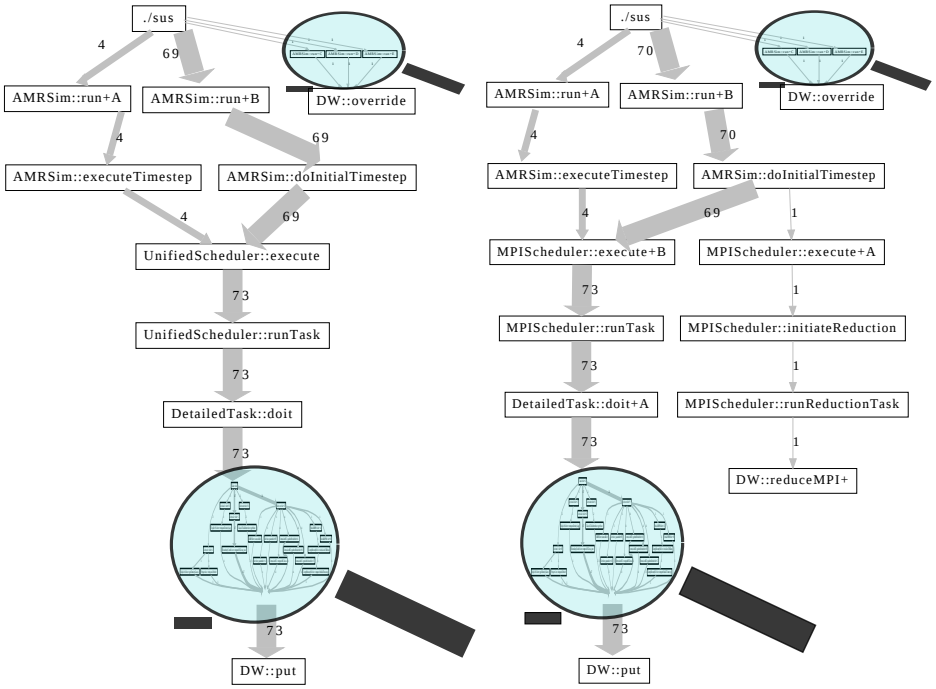

Fig. 3. Mini Coal Boiler Case Study CSTGs. Crashing execution is on the left and working execution is on the right. CSTGs contain all the paths leading to the instrumentations added before the `put()` and `remove()` calls of the hash table `d_levelDB`.

Case Study 2: Poisson2 The *Poisson2* problem is the second of four Uintah examples that solve Poisson's equation on a grid using Jacobi iteration, and each example exercises specific portions of the infrastructure. In this example, Poisson's equation is discretized and solved using an iterative method. The *Poisson2* problem employs the Uintah's sub-scheduler feature that enables finer iteration within a given simulation timestep of the top-level scheduler. It exercises a bug causing nondeterministic crashes during Uintah runs—a segmentation error occurs in the scheduler component of Uintah, more precisely in function `resetWaittime()`. In this scenario, we leveraged CSTGs to compare different, nondeterministic runs of the same version of the system.

Our root-cause exploration proceeded as follows. First, we investigated function `resetWaittime(double)`, where we noticed nothing out of the ordinary—there is only a simple assignment to the variable `d_waitstart`, as in Listing 1.2. Then, after running Uintah a few times on the same input, we noticed that the crash is nondeterministic: it typically occurs (in time-steps 1 or 2), but not always. Next, we decided to leverage our CSTGs to investigate this problem further. We added stack trace collectors to observe the execution history and paths leading to the crashing function `resetWaitTime()`. In this case study, we

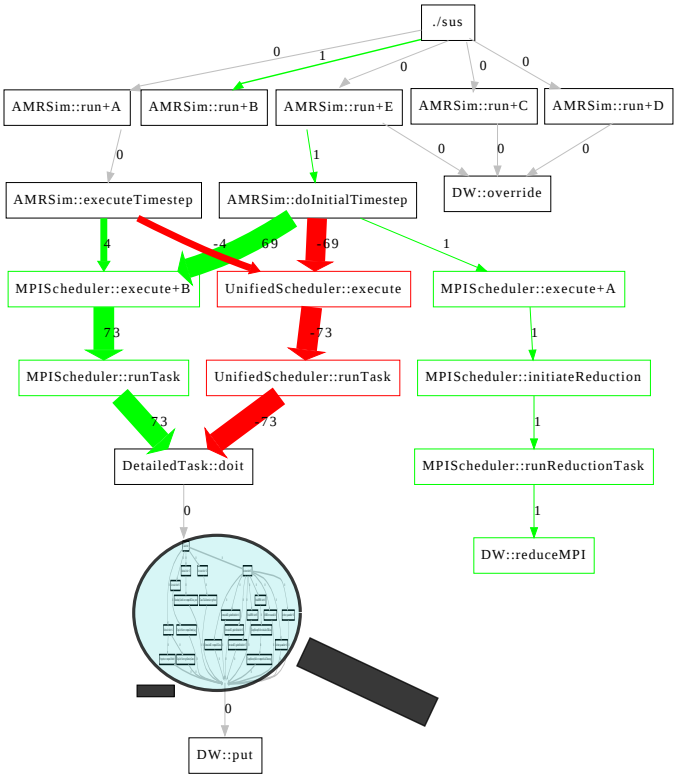

Fig. 4. Mini Coal Boiler Case Study CSTG Delta. It highlights the differences between the CSTGs in Fig. 3.

observe executions of the same system version where the crash does and does not happen.

Fig. 5 shows the CSTG delta resulting from these collections. We immediately learn that the crashing run contains the user-provided function `Poisson2::timeAdvance()` that is executed every time-step; however, only once does `timeAdvance()` invoke the observed function `resetWaittime()`. (It is worth remembering that functions are not recorded when invoked, but only when/if a collection point is reached.) Armed with much better understanding of when exactly the crash occurs, we switched to using a debugger to step through the code. We observed that the value of the variable `numThreads_` is abnormally high in the function `execute()` on the crashing path. There are two common reasons why this happens: either the variable is uninitialized or it suffers memory corruption.

Further exploration of the source code reveals that `numThreads_` is indeed never initialized before being used for the first time. While an initial value of an uninitialized variable is often zero, that is not guaranteed by the compiler—hence the nondeterministic behavior we observed. When the initial value of

```
void UnifiedSchedulerWorker::resetWaittime(double start) {
  d_waitstart = start; // crashing point
  d_waittime = 0.0;
}
```

Listing 1.2. Uintah Code Excerpt where the *Poisson2* Crash Occurs.

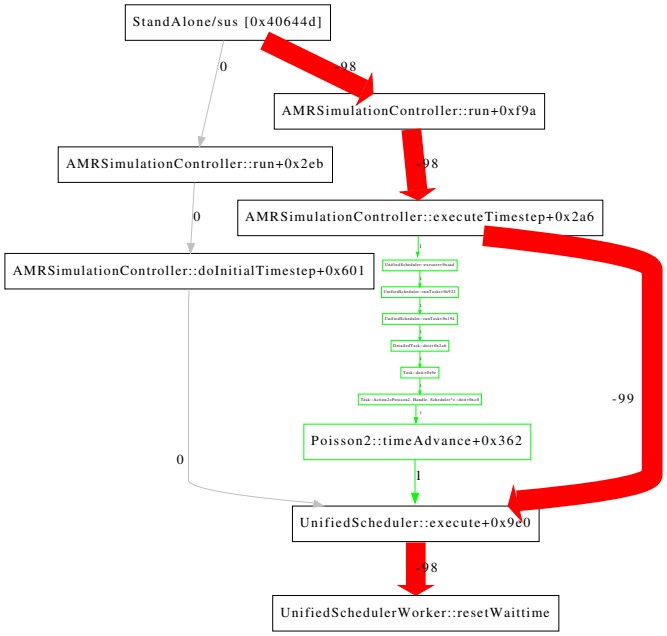

Fig. 5. Poisson2 Case Study CSTG Delta. There are 3 paths leading to the observed function in the crashing run, but only 2 paths in the non-crashing run. The extra path is highlighted in green.

`numThreads_` happens to be zero, `resetWaittime()` is not invoked. Occasionally, when the initial value is not zero, `resetWaittime()` is invoked and the crash happens soon thereafter. Note that `resetWaittime()` gets invoked several times from different parts of the code, but only once through the problematic path leading to the crash, as clearly shown in Fig. 5. As it turns out, the variable `d_waitstart` is only allocated and `numThreads_` initialized in the function `problemSetup()` that never gets invoked.

As an additional exercise, we used Valgrid [23] to expose this problematic usage of an uninitialized variable. And while Valgrind was successful in locating the problem, it took us several days to reach that point due to the large performance overhead of using Valgrind— each Uintah run took several hours to finish. Collecting CSTGs, on the other hand, incurs almost no performance overhead (see §7) and provides us with more contextual information than Valgrind.

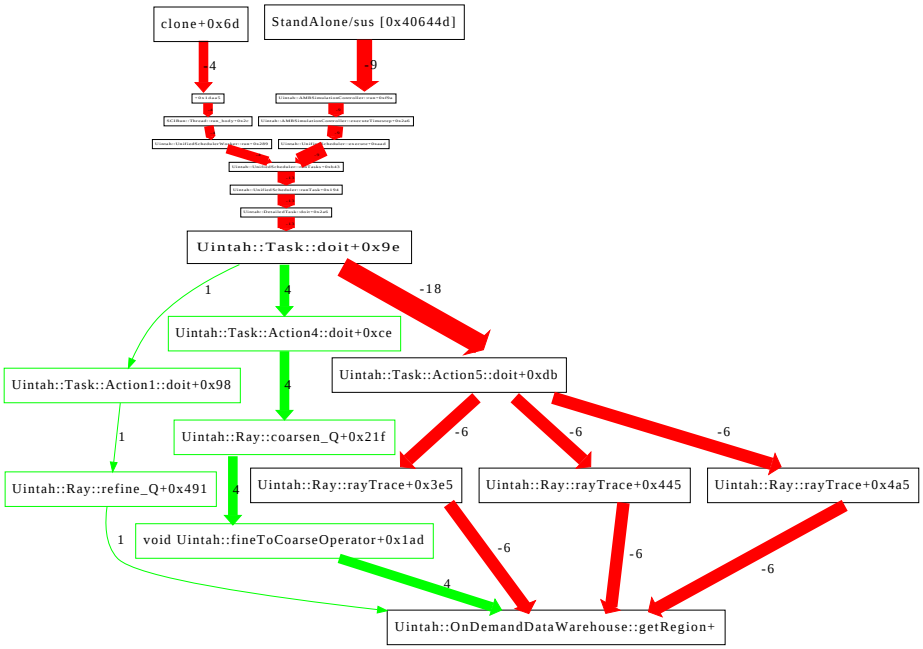

Fig. 6. Arches Case Study CSTG Delta Comparing Runs on Different Input Files.

To summarize, this is a user-introduced bug related to incorrect usage of the Uintah system. It is nondeterministic in nature, and hence could have stayed dormant for a long time. The use of CSTGs, and especially the ability to compare across two different executions, helped us to root-cause this bug. The synergistic role of a traditional debugger is also apparent, and such versatile solutions to root-cause bugs are paramount to improving the productivity of HPC developers.

Case Study 3: Arches Our third case study examines a radiative heat transfer benchmark simulation using the *Arches* component, which was designed for the simulation of turbulent reacting flows with participating media radiation. During this Uintah simulation, an exception is thrown in the data warehouse function `getRegion()`. After we performed the initial inspection of the source code, we observed that the problem is related to simulation patches and grid regions. However, the exact details were unclear. Here, we leveraged CSTGs to compare runs on slightly different inputs. In particular, we managed to only slightly modify the input to obtain a run that finishes without a crash.

A typical input file of Uintah contains many parameters defining grid values, variables, algorithms, and components used in setting up simulations. In this case study, we changed the problematic input file, modifying several parameters that we suspected were related to the crash. Varying these parameters, we were able to obtain an input that does not cause the crash and finishes normally.

We then generated CSTGs using the two different input file versions to compare the executions. Fig. 6 shows the CSTG delta. The paths traversing the green nodes only happen in the crashing run. Hence, our hypothesis was that there was something wrong in one of the functions belonging to these paths. Reporting back to the Uintah developers, we were able to confirm that the function `Uintah::Ray::refine_Q()` was calling `getRegion()` with wrong parameters, which resulted in this crash.

5.2 Other Usage Scenarios

We listed many envisioned CSTG usage scenarios in §3, some of which we covered in the previous section. Due to lack of space, we now only briefly present two additional CSTG usage scenarios.

Scenario 1: Matching Events In this scenario, we are leveraging CSTGs to observe call paths leading to `MPI_Isend` and `MPI_Irecv` in Uintah. In a typical message passage application, the number of message sends and receives should match. But in this case, we can notice from CSTGs that they do not match by looking at the incoming edges and their degrees (denoted with numbers on the edges). Similar CSTGs can be used as initial points of investigations of mismatches in the number of matching events.

Scenario 2: Same Inputs, Different Outputs In this scenario, we obtained two different versions of the GNU lexical analyzer *Flex* from the Software-artifact Infrastructure Repository [10]. For a particular input, these two versions were unexpectedly printing different outputs. We leveraged CSTGs to observe the call paths leading to the print character function in *Flex* called `outc()`. The CSTG delta clearly marks a path that occurs three times less in the execution of one version of *Flex* versus the other. We confirmed that this call path contains the modified code that generated the different output.

6 Implementation Details

In our current implementation of CSTGs in the context of Uintah running MPI on several nodes, we collect stack traces separately at every process. We achieve this by invoking the `backtrace()` function (from *execinfo.h*) each time a stack trace collection instruction is executed. An example of a stack trace collected is:

```
stack_trace:
MPIScheduler::postMPISends(DetailedTask*, int)+0xa15
MPIScheduler::runTask(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]
```

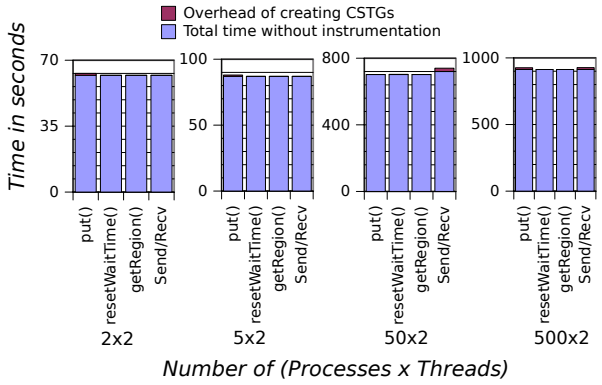

Fig. 7. Scaling Experiments Showing Overhead of CSTG Creation.

Each line in the stack trace is comprised of a complete function signature and a hexadecimal address indicating the calling context of the next called function.

We collect such stack traces to create a CSTG in memory at each process. After processing each stack trace, the current CSTG is updated with new nodes and/or edge-weights; the stack trace is then discarded. The cost of creating a CSTG is equal to maintaining a hash table, so the complexity is $\mathcal{O}(N)$ on average for N stack traces collected. Hence, CSTGs are memory efficient. At the end of an execution, per-process CSTGs are merged into one global graph. The merge operation itself is simple: if an edge already exists, its weight is incremented, and otherwise a new edge is added. Finally, we compare CSTGs by creating a graph diff, showing weight deficits as negative numbers (on red edges) and excesses as positive numbers (on green edges).

7 Scaling Experiments

In this section, we present preliminary scaling studies of the viability of using CSTGs at larger scale, approaching 1,000 nodes. In the scaling studies, we used the same stack trace collection points as before (namely `put()` as in *Mini Coal Boiler*, `resetWaitTime()` as in *Poisson2*, `getRegion()` as in *Arches*, and `Send/Recv` as in *Mathing Events*), albeit a different Uintah input file that was easy to scale. We compare pure Uintah runtimes with runtimes when the code was instrumented, stack traces collected, CSTGs created in memory, and graph files written to disk. The used input files do not produce crashes so that we can run the simulations to completion and build the full CSTGs. The experiments were performed on a cluster with 66 nodes, each node with 4 AMD Opteron Magny-Cours 6164HE 12-core 1.7GHz CPUs, 64GB RAM, 7200RPM SATA2 hard drives, and 10 Gigabit Ethernet.

Fig. 7 shows that the overhead of collecting stack traces and creating CSTGs in memory for various collection points. The overhead is very small (less than

1% in average) in all the scenarios tested, and clearly our solution scales well in practice. The time to collect a stack trace depends of the number of stack frames to be transversed. However, the time to collect the same stack trace is constant throughout the execution of the program. The complexity of creating a CSTG with N stack traces is $N \cdot \Theta(1)$, or $\mathcal{O}(N)$ on average, which is related to maintaining a hash table. In addition, there is no communication between processes involved (each process creates its own CSTG merged in the end), and the final files written are quite small (negligible I/O demand). Hence, the total overhead of creating CSTGs primarily depends on the number of stack traces collected, which changes depending on the simulated problem and inserted collection points. In our experiments, the number of stack traces collected per run ranges from 150 to 322,000. To conclude, our preliminary scaling studies show the feasibility of employing CSTGs without significant overhead in parallel HPC computational frameworks such as Uintah.

8 Conclusions

Finding the root-causes of bugs in the context of large-scale HPC projects is highly resource-intensive: it takes lead developers away from doing useful science, and engages them in a "fire-fighting" frenzy. Often, the actual bug manifestation (e.g., a crash site) has scanty information pertaining to its root-cause. In this paper, we propose a facility for expeditiously debugging sequential and parallel programs called Coalesced Stack Trace Graphs (CSTG). Our approach relies on finding salient differences in executions using CSTGs, and possible scenarios include comparing: different versions of a system, runs of the same system to understand the effects of nondeterminism, runs on different inputs, matching events such as message sends/receives, and different time-steps, loop iterations, or processes.

In the traditional debugging process, as a user gains knowledge about the problem at hand, he/she can use CSTGs to provide the necessary contextual information and greatly accelerate the identification of the root-cause. We demonstrated the applicability of CSTGs in several real bug case studies, primarily in the context of Uintah, an open-source software framework for solving complex multiscale multi-physics problems on cutting edge HPC systems. Our implementation of CSTGs is simple, has low overhead, and scales well in practice. It can be used in many scenarios to help in the debugging process of concurrent or sequential software, and despite most of our case studies were applied to Uintah, it does not depend on it as we illustrated with our *Flex* case study.

References

1. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, pages 85–96, 1997.
2. 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 *IPDPS*, pages 1–10, 2007.

3. K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle. Finding similar failures using callstack similarity. In *Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, 2008.
4. M. Berzins. Status of release of the Uintah computational framework. SCI Technical Report UUSCI-2012-001, 2012.
5. I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson. Mining temporal invariants from partially ordered logs. In *SLAML*, 2011.
6. I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *FSE*, pages 267–277, 2011.
7. G. Bronevetsky, I. Laguna, S. Bagchi, B. de Supinski, D. Ahn, and M. Schulz. AutomaDeD: Automata-based debugging for dissimilar parallel tasks. In *Dependable Systems and Networks (DSN)*, pages 231–240, 2010.
8. Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *ICSE*, pages 1084–1093, 2012.
9. Allinea DDT. `http://www.allinea.com/products/ddt`.
10. H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.
11. J. D. d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson. Uintah: A massively parallel problem solving environment. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 33–41, 2000.
12. 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 the ACM*, 54(12):82–91, Dec. 2011.
13. S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *International Conference on Software Engineering (ICSE)*, pages 145–155, 2012.
14. A. Humphrey, Q. Meng, M. Berzins, D. C. B. de Oliveira, Z. Rakamarić, and G. Gopalakrishnan. Systematic debugging methods for large scale HPC computational frameworks. *Computing in Science and Engineering*, 16(3):48 – 56, 2014.
15. S. Kim, T. Zimmermann, and N. Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 486–493, 2011.
16. Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. Investigating applications portability with the Uintah DAG-based runtime system on petascale supercomputers. Technical Report UUSCI-2013-003, SCI Institute, Utah, 2013.
17. F. Pastore, L. Mariani, and A. Goffi. Radar: A tool for debugging regression problems in C/C++ software. In *International Conference on Software Engineering (ICSE)*, pages 1335–1338, 2013.
18. Cleaner, Cheaper Energy is Goal of Supercomputer Research. `http://unews.utah.edu/news_releases/16m-for-coal-energy-research/`.
19. Rogue Wave Software. `http://www.totalviewtech.com`.
20. 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 *NSDI*, pages 4–4, 2011.
21. A. Schroter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Working Conference on Mining Software Repositories (MSR)*, pages 118–121, 2010.

22. Uintah. `http://www.uintah.utah.edu/`.
23. Valgrind. `http://valgrind.org`.
24. A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *FSE*, pages 253–267, 1999.