Crash Early, Crash Often, Explain Well

Practical Formal Correctness Checking of Million-core Problem Solving Environments for HPC

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

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

***Abstract*—While formal correctness checking methods have been deployed at scale in a number of important practical domains, we believe that such an experiment has yet to occur in the domain of high performance computing at the scale of a million CPU cores. This paper presents preliminary results from the Uintah Runtime Verification (URV) project that has been launched with this objective. Uintah is an asynchronous task-graph based problem-solving environment that has shown promising results on problems as diverse as fluid-structure interaction and turbulent combustion at well over 200K cores to date. Uintah has been tested on leading platforms such as Kraken, Keenland, and Titan consisting of multicore CPUs and GPUs, incorporates several innovative design features, and is following a roadmap for development well into the million core regime. The main results from the URV project to date are crystallized in two observations: (1) A diverse array of well-known ideas from light-weight formal methods and testing/observing HPC systems at scale have an excellent chance of succeeding. The real challenges are in finding out exactly which combinations of ideas to deploy, and where. (2) Large-scale problem solving environments for HPC must be designed such that they can be "crashed early" (at smaller scales of deployment) and "crashed often" (have effective ways of input generation and schedule perturbation that cause vulnerabilities to be attacked with higher probability). Furthermore, following each crash, one must "explain well" (given the extremely obscure ways in which an error finally manifests itself, we must develop ways to record information leading up to the crash in informative ways, to minimize off-site debugging burden). Our plans to achieve these goals and to measure our success are described. We also highlight some of the broadly applicable concepts and approaches.**

I. Introduction

Any software system that retains its initial conceptual blueprint and achieves substantial scale over a relatively short period of time must have good underlying organizational principles and software architecture. In this sense, there are many well-engineered high performance computing (HPC) systems in existence. However, parallel computing for HPC is almost perpetually at the cutting edge of available processor types and is incorporating heterogeneous concurrency models, hence maintaining the integrity of the code is a constant challenge. It is clear that code-level bugs can take away from the amount of useful science that can be conducted on today's scarce high-end HPC platforms.

This paper[1] summarizes preliminary results from an ongoing collaboration between a subset of its authors interested in building a high-end problem solving environment called Uintah, and another subset interested in developing *formal* software testing approaches that can help eliminate code-level bugs, and hence enhance the value offered by Uintah. While this collaboration is in its early stages, our ongoing research project dubbed *Uintah Runtime Verification* (URV) already offers a number of valuable insights.

This paper is largely focused on scalable formal testing techniques that we plan to develop and experiment with during the URV project. Long-term, we anticipate the role of formal methods to go beyond code-level correctness checks, informing future software architectural evolutions in response to the ever-growing demand for computational power. We discuss some of these long-term plans as well.

Growth Phases of Uintah. One of the main approaches suggested for the move to multi-petaflop architectures (and eventually exascale) is to use a graph representation of the computation to schedule work, as opposed to a bulk-synchronous approach in which blocks of communication follow blocks of computation [2]. The importance of this approach for exascale computing is expressed by recent studies [3]. Following this general direction, Uintah has evolved over the past decade over three significant phases:

- 1998-2005 [4]: having a static task-graph structure and running at about 1000 cores;
- 2005-2010 [5], [6]: incorporating many dynamic techniques, including new adaptive mesh refinement methods, and scaling to about 100K cores;
- 2010-2013 [7], [8]: Uintah has shown promising results on problems as diverse as fluid-structure interaction and turbulent combustion at scales well over 200K CPU cores. It presently incorporates shared memory (thread-based) schedulers as well as GPU-based schedulers.

Problem solving environments 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

[1]Title modeled after C.A.R. Hoare's talk "Assert Early, Assert Often" [1].

consequence. Being based on sound and scalable organizational principles, they lend themselves to easy adaptation as witnessed by the Uintah phases mentioned above. For example, GPU schedulers were incorporated into Uintah in a matter of weeks. This fundamentally leads to systems such as Uintah being in a state of perpetual development. End-users have no apparent limits on the scale of problems they want to study or the levels of resolution they want to achieve. There is always a shortage of CPU cycles, total memory capacity, network bandwidth, and advanced developer time. In addition, there is constant pressure to achieve useful science under tight budgets. Structured software development and documentation compete for expert designer time as much as the demands to simulate new problems and to achieve higher operating efficiencies by switching over to new machine architectures.

Previously, the formal methods authors of this paper have explored various scalable formal debugging techniques for large-scale HPC systems and thread-based systems [9]–[13]. A few other researchers also have investigated the use of formal methods for enhancing the correctness of HPC systems [14]–[16]. As far as we know, the URV project is different from these efforts in that it is an attempt to integrate lightweight and scalable formal methods into a problem-solving environment that is undergoing rapid development and real usage at scale.

There are many active projects in which parallel computation is organized around task-graphs. Charm++ [17] has pioneered the task-graph approach and finds applications in high-end molecular dynamics simulations. The DAGuE framework [18] is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. Our interest in Uintah stems from two factors: (1) Uintah has scaled by a factor of 100 in core-count over a decade, and finds numerous real-world applications, (2) we are able to track its development and apply and evaluate formal methods in a judicious manner. We believe that our insights and results would transfer over to other systems, eventually. We now expand on the three themes set forth in the title. A roadmap for the rest of this paper then follows.

Crash Early. As mentioned earlier, the current focus of URV is to help enhance the value of Uintah by eliminating show-stopper code-level bugs as early as possible. In this connection, it is too tempting to dismiss the use of formal testing methods on account of the fact that many of these methods do not scale well, and that many interesting field bugs (and the associated crashes) occur only at scale. While this may be true in general, there are a number of bugs which are reproducible at lower scales—provided of course we know how to force the system under test to follow different executions. 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. However, not every system can be easily tested at lower scales such as lower number of threads, processes, address ranges, etc. The ease with which a system can be downscaled depends on how *well structured* it is. There are many poorly structured systems that allow only certain delicate combinations of such operating parameters; sometimes, these parameters are not well documented.

Uintah, on the other hand, follows a fairly modular design, allowing many problems to be run across a wide range of operating scales—from two to thousands of CPU cores in many cases. There are only relatively simple and well-documented parameter dependencies that must be respected (relating to problem sizes and the number of processes and threads). This gives us a fair amount of confidence that well-designed formal methods can be applied to Uintah at lower scales. This will avoid the nightmarish problem of obtaining large-scale CPU/thread allocations, and then facing the extreme difficulty of discerning errors when they occur.

In our research, we do not directly target bugs that depend on system scale, as they are dependent on factors such as resource exhaustion, faulty components, and misbehaving libraries (*e.g.*, MPI). Uintah's modular checkpoint and restart facilities can ameliorate the severity of these errors.

Crash Often. The second testing problem is to discover effective ways of perturbing a system under test. A system may appear to be working correctly—lulling its users into a false sense of confidence—but suddenly crash when the CPU/thread scheduling policies are changed or various subsystems interact at different rates. A system's testing efficacy depends on the number of *transient behaviors* that the testing method induces. This is the view espoused by the proponents of *model-checking*, which can be viewed as the extreme limit of all such schedule perturbation methods. Unfortunately, not all transient behaviors are relevant: applying schedule perturbations blindly can simply waste testing cycles without hitting bugs. In our preliminary experiments with Uintah, we introduced such schedule perturbations within Uintah's mainline scheduler, without any discernible benefits. We suspect that all we accomplished was to change the specific thread assigned to carry out a particular Uintah task—and not the nature of the task itself.

Our ongoing research is aimed at calibrating execution perturbation methods in terms of their effect on system behavior. The approach is hinged on developing a practical technique to (indirectly) observe the internal state of a system. Even though the internal state of a large-scale system is impractical to directly observe, we believe that it can be characterized by *sequences of key events that threads and processes engage in*. This intuition is the basis of many HPC debugging tools and approaches such as STAT [19], HPCToolkit [20], and AutomaDed [21]. Large-scale systems such as Uintah have a correspondingly large number of threads and processes that engage in "similar" activities over time. Every now and then, interesting outlier events occurs, and it is these outliers that we hope to observe as a function of execution perturbation.

In ongoing research, we are collecting such instrumented event traces in the form of key function call sequences. We are building a framework to collect and equivalence-class the traces in a manner that filters out spatial (*e.g.*, process/thread

ID) and temporal (*e.g.*, time-recurrent behavioral) symmetry. We will employ this framework to discern outlier behaviors. Armed with this framework, we will experiment with various execution perturbation methods, and rank them in their importance. Higher ranked schedule perturbation methods will be those that increase the variety of transient behaviors manifested.

In terms of execution perturbation methods, one approach is clearly that of schedule perturbations—changing the order in which interacting (and presumably commuting) events occur. Another important way to cause execution perturbation is likely to be through the *problem description file*, which specifies the exact simulation problem and its parameters. Our plans to explore these execution perturbation methods are detailed in §IV.

Explain Well. While tripping an error requires the use of well-designed execution inputs as well as schedule selection, it is also equally important to develop effective techniques for explaining and root-causing errors.

- Large-scale HPC systems often have impoverished execution environments. For instance, many do not have the option to generate a `core` file following a crash. This fact implies that explaining field failures can be next to impossible. A few options suggest themselves quite naturally:
  - Develop better (more incisive) pre-deployment debugging methods, to minimize the need for post-deployment error checking.
  - Develop better error monitors. For instance, instead of relying on accidental architectural or OS feature being triggered at the end of a cascading failure sequence (*e.g.*, illegal memory reference, segfault), it would be far more informative to have a simple finite-state machines look for an erroneous execution sequence (*e.g.*, double-free of a lock) and report that.
  - Develop parsimonious representations of the execution state—akin to a well-designed air-craft *black box*—instead of traditional `core` files. Ideally, 'black-boxes' generated from systems such as Uintah must contain the most recent history of events prior to the crash. This history should be maintained in a temporally-refined manner: include major system events well before the crash, and finer details of events approaching the crash.
- The difficulty of explaining errors was described recently by the authors of the popular Photoshop tool [22]:

  > ...the single longest-lived bug I know of in Photoshop ended up being nestled in that code. It hid out in there for about 10 years. We would turn on the asynchronous I/O and end up hitting that bug. We would search for it for weeks, but then just have to give up and ship the app without the asynchronous I/O being turned on. Every couple of versions we would turn it back on so we could set off looking for the bug again.
  >
  > It turned out to be a very simple problem. Like so many other aspects of Photoshop, it had to do with the fact that the app was written first for the Macintosh and then moved over to Windows. On the Macintosh, the set file position call is atomic—a single call—whereas on Windows, it's a pair of calls. The person who put that in there didn't think about the fact that the pair of calls has to be made atomic whenever you're sharing the file position across threads.

  With the increasing scale of HPC systems, it appears that errors that were once elusive can be made to occur on a more predictable basis ("law of large numbers"). Even so, the root-causing of bugs remains extremely difficult—especially if the system scale is large.

In §V, we provide details of how we plan to work towards synthesizing better run-time error explanations.

Roadmap. We first describe the Uintah framework (§II). Ongoing work on schedule instrumentation (§III) and state equivalencing (§IV) are then described. Our thoughts on building a dynamic verifier for Uintah are described in §V. Concluding remarks then follow (§VI).

II. The Uintah Framework

The aim of Uintah is to be able to solve complex multiscale multi-physics problems, such as the benchmark C-SAFE problem. This is a multi-physics, large deformation, fluid-structure problem consisting of a small cylindrical steel container filled with a plastic bonded explosive (PBX9501) subjected to convective and radiative heat fluxes from a fire [23]. In order to solve many such complex multiscale multi-physics problems (e.g., [24]–[27]), Uintah makes use of a component design that enforces separation between large entities of software and can be swapped in and out, allowing them to be independently developed and tested within the entire framework (see Fig. 1).

Uintah is novel in its use of a task-based paradigm, with complete isolation of the user from parallelism. The individual tasks are viewed as part of a directed acyclic graph (DAG) and are executed adaptively, asynchronously, and now also often out of order [7]. This is done by utilizing an abstract task-graph representation of parallel computation and communication to express data dependencies between components. The task-graph is a directed acyclic graph of tasks (see Fig. 2). Each task consumes some input and produces some output, which is in turn the input of some future task. These inputs and outputs are specified for each patch in a structured Adaptive Mesh Refinement (AMR) grid. Adaptive structured meshes consist of hexahedral patches, often of 16^3 elements, that hold the problem data [4]. Associated with each task is a C++ method which is used to perform the actual computation. Each component specifies a list of tasks to be performed and the data dependencies between them. The task-graph approach of Uintah shares many features with the migratable object philosophy of Charm++ [28]. In order to increase efficiency, the task graph is created and stored locally [5]. We now describe the main *distributed* subsystems of Uintah shown in Fig. 1.

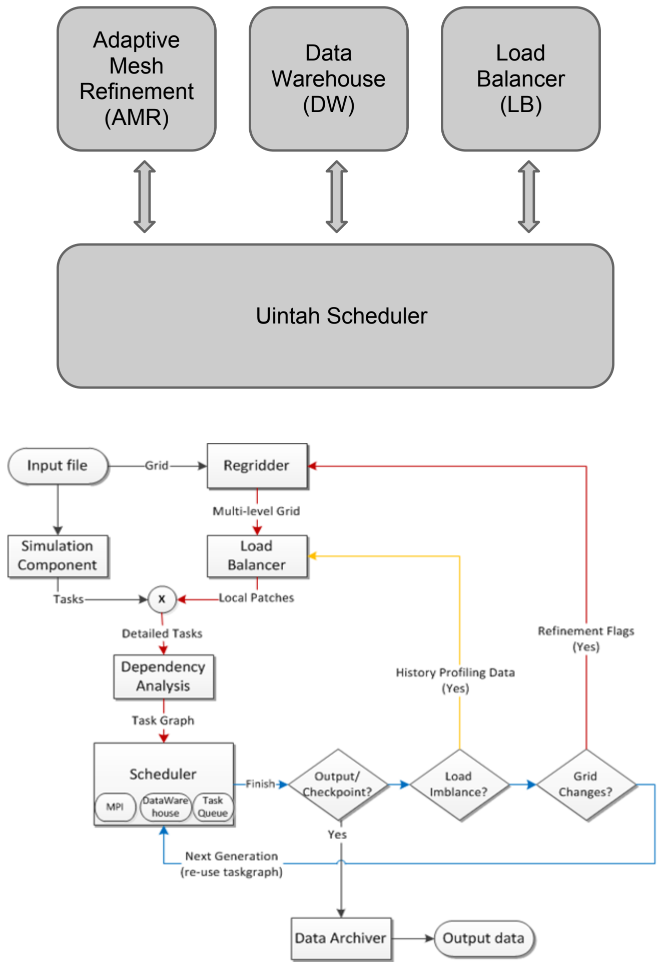

Fig. 1. Overall Uintah System Organization and Operation.

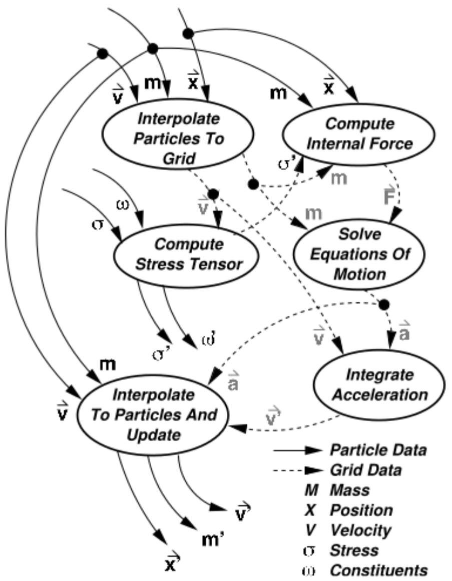

Fig. 2. Example of Uintah Task Graph (from [7]).

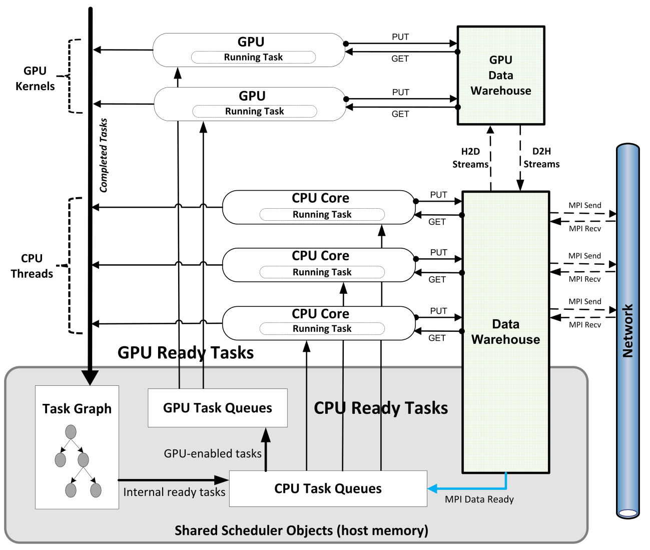

Fig. 3. Architecture of Uintah Hybrid Task Scheduling System (from [29]).

Task Scheduler. Uintah's task scheduler, given in Fig. 3, is responsible for computing the dependencies of tasks, determining the order of execution, and ensuring that the correct inter-process communication is performed [5]. In this mapping communication is automatically overlapped with computation. Originally, Uintah used a static scheduler in which tasks were executed in a predetermined order. This caused delays when a single task was waiting for a message. The new Uintah dynamic scheduler changes the task order during the execution to ensure that processors do not have to wait [7]. As of recently, the dynamic scheduler has been extended with support for GPU tasks [8], [29]. This *hybrid* scheduler required a large amount of development to support the out-of-order execution, which produced a significant performance benefit in lowering both the MPI wait time and the overall runtime. The hybrid scheduler utilizes four task queues. For CPU tasks, it has an internal ready queue and an external ready queue. For GPU tasks, it has an initially ready GPU queue, containing tasks that have requisite simulation variable data copies from host-to-device pending, and a second queue for the corresponding device-to-host data copies pending completion. First, if a task's internal dependencies are satisfied, it will be put in the CPU internal ready queue, where it will wait until all required MPI communication has finished. Then, the task moves to the external ready queue. In this same step, if the task is GPU-enabled, it is put into the host-to-device copy queue for advancement toward execution. Ultimately, the task goes to the pending device-to-host copies queue. As long as the external queue (resp., GPU task queue) is not empty, the processor (resp., GPU) always has tasks to run. This can help to overlap the MPI communication time with task execution, and the approach significantly reduces MPI wait times [5], [7], [29].

Data Warehouse. The core scheduler component stores simulation variables in a data warehouse. The data warehouse is a dictionary-based hash-map which maps a variable name and patch id to the memory address of a variable. Each task can get its read and write variable memory by querying the data warehouse with a variable name and a patch id. The task dependencies of the task graph guarantee that there are no memory conflicts on local variables access, while variable versioning guarantees that there are no memory conflicts on foreign variables access. These mechanisms have been implemented for supporting out-of-order task execution in our previous work using a dynamic MPI scheduler [7]. This means that a task's variable memory has already been isolated. Hence, no locks are needed for reads and writes on a task's variables memory. However, the dictionary data itself still needs to be protected when a new variable is created or an old variable is no longer needed by other tasks. As dictionary data must be consistent across the worker threads, the data warehouse has to be modified to be thread-safe by the addition of read-only and read-write locks. When a task needs to query the memory position of a variable, a read-only lock must be acquired before this operation is done. When a task needs the data warehouse to allocate a new variable, or to cleanup an old variable, a read-write lock must be acquired before this operation is done. While this increases the overhead of multi-threaded scheduling, locking on dictionary data is still more efficient than locking all the variables.

Adaptive Mesh Refinement (AMR). Dynamically refining the mesh allows the simulation to reduce error in regions where it is significant, while maintaining a coarse mesh in other regions for performance reasons. This is particularly important for simulations with moving features. The simulation components maintain a set of refinement flags, which specify the regions that need to be refined. As the simulation evolves, the refinement flags are used to create new grids with the necessary refinement.

Whenever the grid changes, a series of steps must be taken prior to continuing the simulation. First, the patches must be redistributed evenly across processors through load balancing. Second, all patches must be populated with data either by copying from the same region of the previous grid or by interpolating from a coarser region of the previous grid. Finally, the task graph must be recompiled.

Load Balancer. A load balancer component is responsible for assigning patches to processors. It attempts to distribute work evenly while minimizing communication by placing patches that communicate with each other on the same processor. A large load imbalance will cause processors with less work to wait for others to finish their computation, leading to poor utilization of system resources. In addition, too much communication also causes performance issues. By clustering neighboring patches together, the framework can greatly reduce the necessary communication overhead, and hence also the overall runtime. Finally, both AMR and particle methods require constant rebalancing of workload: with AMR methods, the workload changes as the mesh changes, while with particle methods, the workload can change on each time step as particles move throughout the domain. If a slow load balancing algorithm is used and load balancing occurs often, the time to load balance can dominate the overall runtime. In this case, it may be preferable to use a faster load balancing algorithm at the cost of more load imbalance. In addition, the time to migrate data between processors may also become significant when load balancing often.

```
waitPermission(Condition c){
  IsendRequestMsg(c, masterRank);
  waitPermissionMsg(masterRank);
}
```

Fig. 4. Pseudocode of `waitPermission`.

III. Schedule Instrumentation

Early on in the URV project, it became apparent to us that we need a flexible scheme to observe the Uintah system under operation and to control its internal schedules. We preferred our own home-grown solution implemented using MPI (a later implementation will perhaps use PnMPI [30] or MPE [31]). This implementation is based around a *master controller*, and is now briefly described and illustrated in Fig. 6.

The master controller is an extra MPI process dynamically created during the execution of a MPI program. It is used to take control of the execution of threads (or MPI processes) and return the control back when certain conditions are satisfied. The user of the master controller inserts conditional control points in the main application. When a thread reaches a control point, it sends a message to the master controller asking to continue the execution. The master controller is responsible for managing these requests and sending back a message giving the permission to the original thread when conditions are satisfied.

There are two important functions in the master controller:

- `waitPermission(Condition c)` and
- `processRequests()`.

Function `waitPermission(Condition c)` is called by any thread and will block it until condition `c` is satisfied (see Fig. 4). A condition can be, for example, a time to wait or an event to happen. Function `processRequests()` receives all the requests from any process in the main application, and sends back messages to unblock the threads when the conditions of the requests are satisfied (see Fig. 5). A thread that calls `waitPermission` will send an asynchronous message to the master controller, and then get blocked while waiting for the permission message to arrive.

The master controller will spend most of its runtime in the main loop of `processRequests`, until a request to finish it is fired. In the first part of each iteration, the master controller calls `Iprobe` that asynchronously checks if a new message needs to be received from any of the other MPI processes. If that is the case, the new message is received and saved. In the second part of each iteration, the master controller goes through all the requests that are not yet satisfied, and for each it checks if now they are. When a request becomes satisfied,

```
processRequests(){
  while (!FINALIZE) {
    Iprobe(ANY_SOURCE, newMsg);
    if (newMsg){
      receiveRequestMsg(request);
      saveRequest(request);
    }
    foreach unsatisfied request do {
      if (request.satisfied()) {
        sendPermissionMsg(request.sourceRank);
        request.satisfy();
      }
    }
  }
}
```

Fig. 5. Pseudocode of `processRequests`.

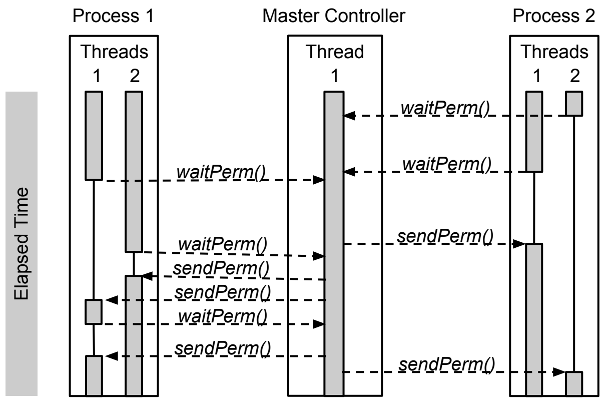

Fig. 6. Interaction between Two Processes and the Master Controller.

a message is sent to the original thread giving it permission to continue its execution.

In the simple example of Fig. 6, we see how two processes with two threads each interact with the master controller. When a control point is reached during the execution of the main application, a message is sent to the master controller and the thread gets blocked waiting for a message back with the permission to continue. A permission will be granted when the sent condition is satisfied.

Our instrumentation scheme using the master controller has allowed us to understand salient aspects of Uintah's internal operation. It will be the main infrastructure for our initial set of schedule perturbation and event observation experiments.

IV. State Equivalencing and Visualization

Efficient techniques for summarizing the internal state of large-scale *concurrent* systems are of great interest. Intuitively, the state of any system is described by (equivalence classes of) its executions, where the equivalencing is done based on observational criteria of interest (*e.g.*, modulo some specific output assertions or invariants). Function call sequence tracing is, for example, one convenient way to summarize its states, as demonstrated in debugging tools such as STAT [19], HPC-Toolkit [20], and AutomaDed [21]). How do these ideas apply to very large-scale distributed system such as Uintah? Clearly, exact state estimation is a very hard problem. However, in the URV project, our interest is not to answer this question exactly. Rather, we are primarily interested in discovering what schedule perturbations or problem-specification files make new behaviors happen. For this purpose, we have come up with the concept of (what we call) *relevant concurrent events* (RCE). For each concurrent thread τ, an RCE is a finite sequence $f_1, f_2, \ldots, f_N, g$ where $f_1, \ldots, f_N$ are function calls made by τ and g is a global event such as lock acquire/release, MPI message activity—and anything that is globally visible to other threads. Clearly, each thread of a system such as Uintah is associated with thousands of RCEs. However, intuition suggests that in systems that have many identical components and threads, *many RCEs tend to recur during the course of a large-scale simulation.*

At present, we have instrumented Uintah using our master-controller based architecture explained in §III. We are in the process of recording various RCEs emitted by the Uintah threads. Currently, we are generating extremely simple RCEs where the sequence $f_1, \ldots, f_N$ is very short, or in some cases even empty (thus going purely by the global events). Our intuition is that the longer the function call sequence of an RCE, the more context information we are associating with the global action g; however, the cost of logging and studying the RCEs also correspondingly goes up.

We are in the process of experimenting with different equivalencing criteria for RCEs. Our hope is that if we trace the sequence of RCEs that a system transitions through (after we suitably equivalence the RCEs), we will be able to tell better which sequences of events are *outliers* and which are merely repetitions of prior actions. In a system such as Uintah, there is a high degree of spatial symmetry (identical or nearly identical components) as well as temporal symmetry (behaviors that tend to recur over simulation time steps). Thus, we are considering both spatial equivalencing (*e.g.*, forget the distinction between two identical RCEs performed by distinct threads during one simulation step) and temporal equivalencing (one RCE performed by two threads across distinct simulation steps) approaches.

Following the equivalencing, we will produce visualizations of RCE transitions produced by typical Uintah simulations, portraying the recurrent nature of RCEs, and of course highlighting outlier behaviors. We will then subject Uintah to different perturbations and hope to rank the efficacy of these perturbations based on the degree of salient changes they induce across RCE transition diagrams. We believe that this study will lead to a better understanding of which active testing approaches [32] are more effective for large-scale systems.

The purpose of the aforesaid visualization studies is to design better perturbation methods. However, in the final deployment of Uintah, we will still log key system events and maintain them in a 'black box' in the temporally-refined manner mentioned earlier. In fact, key system invariants and contracts (*e.g.*, no unlock before a prior lock) will be checked at runtime. The "law of large numbers" observation suggests that we do not need to perform these contract checks within each and every one of the millions of threads; instead, we might be able to take a sampling approach (spatial or temporal). In fact, checking invariants at each and every thread can detrimentally impact performance and energy consumption, and so we strongly believe that suitable sampling-based approaches are essential. We anticipate our studies pertaining to RCEs and the generation of 'black boxes' to be finished over the coming year.

V. Dynamic and Runtime Verification

Bugs encountered during the development of Uintah can be roughly classified into two equivalence classes:

- *Shallow bugs* that are easy to trigger and reproduce using existing regressions and configurations at small scale, and therefore much easier to fix. Such bugs often appear during, or right after the implementation of a new feature.
- *Deep-seated bugs* are triggered only by very specific configuration and runtime parameters, and often are not easy to reproduce. Such bugs can be dormant for months, and typically manifest when the code of Uintah is ported to a new platform. These bugs sap designer productivity, often requiring weeks of their undivided attention.

The primary goal of the runtime verification approaches being developed during the URV project is to help find and explain deep-seated bugs.

Our runtime testing approaches fall under the umbrella of *active testing* [32]. They aim to perturb execution schedules while not violating the causal order of task executions implied by the Uintah task graphs. Such perturbations can be introduced into any of the Uintah components (scheduler, data warehouse, AMR, or load balancer). More specifically, we are investigating two complementary perturbation techniques: *schedule perturbation* and *configuration fuzzing*.

The goal of schedule perturbation is to alter the order in which interacting events (e.g., thread locking, MPI messages) occur during the execution of Uintah. Configuration fuzzing, on the other hand, tries to trigger perturbations in executions by altering Uintah's input configuration file, which specifies the Uintah simulation problem and its parameters.

Schedule Perturbation. Schedule perturbation has already proved to be effective for finding bugs in the context of multi-threaded shared-memory programs (*e.g.*, [10], [33]) and small-scale message-passing applications (*e.g.*, [9]). However, to the best of our knowledge, efficient schedule perturbation for a large computational science problem solving environment such as Uintah has not been attempted before. The number of events that could be perturbed in a system such as Uintah is simply too large for the conventional approaches, which pick events to perturb either blindly or using simple heuristics without any domain knowledge. As said before, blindly perturbing schedules of worker threads in the Uintah's task scheduler might only alter which thread will execute which Uintah task.

The URV project aims to develop methods that leverage knowledge of the system architecture and code to improve the schedule perturbation methods. For example, we can exploit the fact that Uintah's execution advances along the lines of

```
<LoadBalancer type="DLB">
  <costAlgorithm>ModelLS</costAlgorithm>
  <hasParticles>true</hasParticles>
  <timestepInterval>25</timestepInterval>
  <gainThreshold>0.15</gainThreshold>
  <outputNthProc>1</outputNthProc>
  <doSpaceCurve>true</doSpaceCurve>
</LoadBalancer>
```

Fig. 7. Excerpt from a Uintah Problem Specification (UPS) File. The excerpt provides parameters for a load balancer.

simulation time steps that are fairly symmetric (across time). This may give us the opportunity to pursue novel schedule perturbation methods that do not apply in a general active-testing framework. As an example, if the Uintah active tester determines in simulation timestep t that a specific schedule perturbation is productive to explore, *it need not re-execute the Uintah simulation beginning timestep* t *with this new schedule perturbation*, similar to what a model checker [34] might have done. Instead, we believe that we can exercise the same schedule perturbation during the subsequent timestep $t+1$. Of course, we plan to measure the efficacy of such heuristics in our future work.

We are currently working on improving the effectiveness of schedule perturbation by (1) incorporating domain-specific knowledge about Uintah into our schedule perturbation engine, and (2) relying on state equivalencing and visualization techniques described in §IV to guide the perturbation algorithm and provide user feedback. Therefore, developers of the system will have the ability to customize our perturbation engine based on their knowledge of the system and their intuitions about where bugs are likely hidden. They will also be guided by the visualizations of RCE sequences, as described in §IV. We will allow developers to specify the events they are interested in causing schedule perturbations around. Furthermore, we will develop a simple interface for modularly plugging in different perturbation strategies. While the perturbation engine will come with several default perturbation strategy plugins, it will also allow developers to modify them based on their domain knowledge. The ultimate goal is to empower the developers of the system with different ways in which they could focus the strength of our tools on the parts of the system they currently find interesting or challenging. We believe that these mechanisms are essential to achieve the desired degree of scalability and practical usability of formal ideas in the context of large-scale computational science problem solving environments.

Configuration Fuzzing. Configuration fuzzing aims to push Uintah into different interesting executions by *fuzzing* [35] its problem description input files. Fuzzing is a well-known testing technique that systematically introduces random data into structured input files while trying to keep them well-formed. Combined with formal techniques, fuzzing has shown great success in finding security vulnerabilities in software (*e.g.*, [36], [37]). In the context of Uintah, we use fuzzing on problem description input files, called Uintah Problem Specification (UPS) files, in order to perturb executions of the system. UPS files use a well-specified XML format in which the exact simulation problem and its parameters, as well as the different parameters of the Uintah framework itself, are defined. Fig. 7 gives a UPS file excerpt providing parameters for the load balancer component of Uintah introduced in §II. First, a type of the used load balancer has to be specified, which is dynamic load balancer (`DLB`) in this case [38]. Then, various parameters used to control the load balancer are set. For example, `costAlgorithm` defines which cost estimation algorithm should be used, while `timestepInterval` specifies how many simulation steps have to pass before load balancing gets reevaluated. Our fuzzing approach will alter values of relevant input parameters in a meaningful way guided by domain-specific knowledge about the system. The goal is to push Uintah into potentially disruptive and unusual execution paths, for example by setting `timestepInterval` to 1 and therefore stressing the load balancer and its interaction with the rest of the system.

VI. Concluding Remarks

Status. While much of this paper has portrayed the URV project and its goals, the master-controller-based scheduler is in operation, and we have begun developing visualizations of RCE transition diagrams. Here are a few concrete data sets from our preliminary studies:

- Events for the visualization tool:
  - File name: hotBlob_AMR
  - Number of processes: 4
  - Number of threads in each process: 4
  - Number of patches in total: 16
  - Number of events been monitored: 9 (5 in the scheduler, 3 in MPI, 1 in AMR).
  - Running time: around 1 min.
  - Number of events recorded: 255k (50k MPI Isend, 50k MPI Irecev, 150k tasks total, 372 time steps, 360 regridding steps).
- Lock study, capturing every lock call:
  - Running time: about 25 minutes.
  - Number of lock calls: 900k

We have casually observed a very high degree of symmetry in these events, as expected. Of course, a more exact and systematic portrayal of symmetry modulo various equivalencing criteria is a much awaited future result. Clearly, all the scalability directions suggested in this paper are essential in order to observe the system state transitions, check for invariant violations online, and then to log a temporally refined sequence of events for the ultimate *Uintah Blackbox*.

Some Tangible Metrics of Success. As we are unaware of a previous study of applying formal methods to a problem solving environment for HPC, we would also like to develop a set of metrics to assess our own success. Some plausible metrics are now listed:

- How many prior bugs (gleaned from Uintah regressions and `svn` logs) can be reproduced using our new methods?
- What scale reduction was possible in configuring Uintah such that the bug was still reproduced? (This sheds light on whether some of the bugs encountered are problem-size dependent.)
- How easy has bug root-causing and explanation become?
- What are the overheads of keeping error-monitors continuously in operation? How much do they slow down and/or increase power consumption?
- What sampling approaches, scheduler perturbation, and fuzzing approaches are effective for what kinds of problems and scales?
- What are measured bug omission rates (if this number can be obtained) and how can they be lowered?
- Are there any possible "probe effects", *i.e.*, does the deployment of our runtime techniques itself mask bugs?

Our ongoing work is aimed at answering these questions and developing solutions that can be used at various stages of software evolution: from design validation to runtime monitors that can continually monitor correctness.

Broadly Applicable Concepts and Approaches. We are interested in developing concepts and methods that will apply to other high performance computing software installations besides Uintah. The obvious difficulty is of course that each HPC system differs from the other in terms of overall organizational principles, the scale at which it is meant to operate, and the life-cycle of the software itself (how many developers are involved and over what period of time the code has evolved). Even if focusing just on HPC problem solving environments based on task-graph reduction, these details vary widely. However, we strongly believe that some of the instrumentation, monitoring, and schedule perturbation techniques we develop will have broader appeal in similar systems. Most importantly, the HPC community and the computer science formal methods community have very rarely overlapped in the past—a point also made recently by Gopalakrishnan et al. [9]. One of the most tangible high-level outcomes of the URV project may be to lend credence to our strong belief that collaborations such as ours are possible, and are beneficial to both sides: to HPC researchers who gain an appreciation of CS formal methods; and to CS researchers who get a chance to involve in concurrency verification problems of a more fundamental nature that directly contributes to a nation's ability to conduct science and engineering research.

References

[1] C. Hoare, "Assert early, assert often," http://research.microsoft.com/en-us/people/thoare/.

[2] M. Berzins, J. Schmidt, Q. Meng, and A. Humphrey, "Past, present, and future scalability of the Uintah software," in *Proceedings of the Blue Waters Workshop*, 2013, to appear.

[3] 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.

[4] J. D. d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, “Uintah: A massively parallel problem solving environment,” in *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2000, pp. 33–41.

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

[6] J. Luitjens and M. Berzins, “Improving the performance of Uintah: A large-scale adaptive meshing computational framework,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010, pp. 1–10.

[7] Q. Meng, J. Luitjens, and M. Berzins, “Dynamic task scheduling for the Uintah framework,” in *Proceedings of the IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*, 2010, pp. 1–10.

[8] A. Humphrey, Q. Meng, M. Berzins, and T. Harman, “Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system,” in *Proceedings of the Conference of the Extreme Science and Engineering Discovery Environment (XSEDE): Bridging from the eXtreme to the campus and beyond*, 2012, pp. 4:1–4:8.

[9] 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.

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

[11] S. F. Siegel and G. Gopalakrishnan, “Formal analysis of message passing,” in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2007, invited tutorial.

[12] A. Vo, G. Gopalakrishnan, R. M. Kirby, B. R. de Supinski, M. Schulz, and G. Bronevetsky, “Large scale verification of MPI programs using Lamport clocks with lazy update,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011, pp. 330–339.

[13] M. Müller, B. de Supinski, G. Gopalakrishnan, T. Hilbrich, and D. Lecomber, “Dealing with MPI bugs at scale: Best practices, automatic detection, debugging, and formal verification,” http://www.cs.utah.edu/fv/publications/sc11_with_handson.pptx, Nov. 2011, full-day tutorial at Supercomputing.

[14] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu, “Efficient data race detection for distributed memory parallel programs,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 51:1–51:12.

[15] S. F. Siegel, “Model checking nonblocking MPI programs,” in *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2007, pp. 44–58.

[16] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, “Combining symbolic execution with model checking to verify parallel numerical programs,” *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 2, pp. 1–34, 2008.

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

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

[19] 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.

[20] 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.

[21] G. Bronevetsky, I. Laguna, S. Bagchi, B. de Supinski, D. Ahn, and M. Schulz, “AutomaDeD: Automata-based debugging for dissimilar parallel tasks,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 231–240.

[22] C. Cole and R. Williams, “Photoshop scalability : Keeping it simple,” in *ACM Queue*, Sep. 2010, http://queue.acm.org/detail.cfm?id=1858330.

[23] J. E. Guilkey, T. B. Harman, and B. Banerjee, “An Eulerian-Lagrangian approach for simulating explosions of energetic devices,” *Comput. Struct.*, vol. 85, no. 11-14, pp. 660–674, 2007.

[24] I. Ionescu, J. E. Guilkey, M. Berzins, R. M. Kirby, and J. A. Weiss, “Simulation of soft tissue failure using the material point method,” *Journal of Biomechanical Engineering*, vol. 128, pp. 917–924, 2006.

[25] J. E. Guilkey, J. B. Hoying, and J. A. Weiss, “Computational modeling of multicellular constructs with the material point method,” *Journal of Biomechanics*, vol. 39, no. 11, pp. 2074–2086, 2006.

[26] G. Krishnamoorthy, S. Borodai, R. Rawat, J. Spinti, and P. J. Smith, “Numerical modeling of radiative heat transfer in pool fire simulations,” in *Proceedings of the ASME International Mechanical Engineering Congress and Exposition (IMECE)*, 2005, pp. 327–337.

[27] A. Brydon, S. Bardenhagen, E. Miller, and G. Seidler, “Simulation of the densification of real open-celled foam microstructures,” *Journal of the Mechanics and Physics of Solids*, vol. 53, no. 12, pp. 2638–2660, 2005.

[28] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng, “Programming petascale applications with Charm++ and AMPI,” in *Petascale Computing: Algorithms and Applications*, 2008, pp. 421–441.

[29] Q. Meng, A. Humphrey, and M. Berzins, “The Uintah framework: A unified heterogeneous task scheduling and runtime system,” in *Proceedings of the International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, 2012.

[30] M. Schulz and B. R. de Supinski, “PNMPI tools: A whole lot greater than the sum of their parts,” in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2007, pp. 30:1–30:10.

[31] “MPI: Performance visualization,” http://www.mcs.anl.gov/research/projects/perfvis/software/MPE/index.htm.

[32] P. Joshi, M. Naik, C.-S. Park, and K. Sen, “CalFuzzer: An extensible active testing framework for concurrent programs,” in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2009, pp. 675–681.

[33] M. Musuvathi and S. Qadeer, “Iterative context bounding for systematic testing of multithreaded programs,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 446–455.

[34] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[35] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[36] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based whitebox fuzzing,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008, pp. 206–215.

[37] V. Ganesh, T. Leek, and M. Rinard, “Taint-based directed whitebox fuzzing,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009, pp. 474–484.

[38] Q. Meng, J. Luitjens, and M. Berzins, “A comparison of load balancing algorithms for AMR in Uintah,” Scientific Computing and Imaging Institute, University of Utah, Tech. Rep. UUSCI-2008-006, October 2008.