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

Diego Caminha B. de Oliveira, Zvonimir Rakamarić, Ganesh Gopalakrishnan, Alan Humphrey, Qingyu Meng, Martin Berzins

Acknowledgements: NSF CCF 1241849 (EAGER: Formal Reliability Enhancement Methods for Million Core Computational Frameworks) and NSF ACI 1148127 SI2-SSE (Correctness Verification Tools for Extreme Scale Hybrid Concurrency)
Uintah Runtime Verification (URV) Project

- **Goal:** Analysis and checking of large high performance computing (HPC) problem solving environments
- **Credo:** Crash early, crash often, explain well.
- **Opportunity:** Formal methods and HPC teams sitting at the same table every two weeks since last summer
- **Focus:** Lightweight formal methods for the Uintah HPC problem solving environment
Uintah Overview

- Parallel, adaptive multi-physics framework
- Fluid-structure interaction problems
- Patch-based AMR using particles and mesh-based fluid-solve

- Explosions
- Foam Compaction
- Carbon Capture
- Clean Coal Boiler
- Industrial Flares
- Shaped Charges
- Sandstone Compaction
- Plume Fires
## Uintah Development

- Uintah is developed over a decade
  - DOE NETL, C-SAFE, ASC Center...
- Clear separation of application and infrastructure code from the start

<table>
<thead>
<tr>
<th></th>
<th><strong>Domain Expert (Engineering)</strong></th>
<th><strong>Infrastructure Expert (Computer Science)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Focus</strong></td>
<td>Problem, methods</td>
<td>Performance, scalability</td>
</tr>
<tr>
<td><strong>Responsibility</strong></td>
<td>Simulation components</td>
<td>Infrastructure components</td>
</tr>
<tr>
<td><strong>Contributions</strong></td>
<td>Arches, ICE, MPM, MPM-ICE, etc.</td>
<td>Load balancing, AMR, task-graph scheduling, communication, checkpointing</td>
</tr>
<tr>
<td><strong>View of Program</strong></td>
<td>Serial code written for a patch</td>
<td>Parallel infrastructure, MPI, threads, GPU</td>
</tr>
</tbody>
</table>
Modular Architecture of Uintah

- Adaptive Mesh Refinement (AMR)
- Data Warehouse (DW)
- Load Balancer (LB)

Uintah Scheduler
Benefits of Modular Architecture

- All applications benefit from infrastructure improvements without change
- Allows infrastructure developers to make improvements without understanding the science of the domain expert
- Successfully scaled from 2K to 512K cores without any changes to applications code
Benefits of Modular Architecture cont.

- Infrastructure components easily updated to follow the latest architectures
  - Multicore and GPU support, lock-free data warehouse...
- Adding formal methods is more feasible
Uintah Scalability

Patch-based domain decomposition

Asynchronous task-based paradigm

- 512K cores on ANL Mira (Blue Gene/Q)
- Multi-threaded MPI – shared memory model on-node
- Scalable, efficient, lock-free data structures
Uintah Task-Based Approach

- Task graph
  - Directed acyclic graph
- Asynchronous, out of order execution of tasks
  - Multi-stage work queue design
- Task – basic unit of work
  - Sequential C++ procedure with computation
- Allows Uintah to be generalized to support coprocessors and accelerators
  - No sweeping code changes
Support for Heterogeneous Systems

- Utilize all on-node computational resources
- Uintah’s asynchronous task-based approach well suited for coprocessor and accelerator designs
  - Introduce accelerator and coprocessor tasks
Lightweight Formal Methods
Lightweight Formal Methods for HPC

- Lightweight formal methods can help with
  - Exploring nondeterminism in a systematic way
  - Providing good measures of coverage
  - Explaining and root-causing errors
  - Runtime system monitoring
  - Hybrid concurrency
  - Memory models
  - Floating point precision

- This talk: Explaining and root-causing errors
Coalesced Stack Trace Graphs

- Stack traces portray a story about the runtime execution of a program by showing
  - call paths leading to a particular function call
  - the number of times a particular path was taken
- Facilitate understanding and root cause analysis of complex bugs
The number of stack traces collected during execution gets very large:
- Coalesce millions of stack traces using adequate graph representations called Coalesced Stack Trace Graphs (CSTGs).

Infrastructure developer controls where stack traces should be collected.
Basic Idea: Diff CSTGs

CSTG 1

CSTG 2

Diff CSTG
Two Case Studies using Real Bugs

- **MiniBoiler**
  - Simulation of oxy-combustion in large-scale clean coal boilers
  - An exception is thrown in the data warehouse function `get()` when looking for an element that does not exist in the data warehouse

- **Explode2D_AMR**
  - Simulation of explosion in Spanish Fork Canyon
  - Wrong calculation of neighbors causes a mismatch in the number of sends and receives causing Uintah to hang. This happens after the first regridding.
Bug Study 1: MiniBoiler

- An exception is thrown in the data warehouse function `get()` when looking for an element that does not exist in the warehouse.
- There are two possible reasons why this element was not found:
  - it was never inserted or,
  - it was inserted but then removed from the data warehouse.
- We insert stack trace collectors before data warehouse `put()` and `remove()` calls and visualize the result.
- We compare graphs of buggy and working executions.
CSTG of MiniBoiler
Diff of Good and Bad CSTG
There is a path in the good version leading to the `reduceMPI()` function that never happened in the crashing version.
Understanding the Difference

- The two versions use different schedulers
  - Good: MPIScheduler calls initiateReduction
  ```cpp
  while (...) {
      ...
      if (task->getType() == Task::Reduction) {
        if (!abort)
          initiateReduction(task);
      }
      else {
        initiateTask( task, abort, abort_point, iteration );
        processMPIRecvs(WAIT_ALL);
        ASSERT(recvs_.numRequests() == 0);
        runTask(task, iteration);
      }
      ...
  }
  ```
  - Bad: UnifiedScheduler never calls initiateReduction
  ```cpp
  // Do the work of the SingleProcessorScheduler and bail if not using MPI or GPU
  if (!Uintah::Parallel::usingMPI() || !Uintah::Parallel::usingGPU()) {
    for (int i = 0; i < ntasks; i++) {
      DetailedTask* dtask = dts->getTask(i);
      runTask(dtask, iteration, -1);
    }
    finalizeTimestep();
    return;
  }
  ```
initiateReduction adds an element into the data warehouse that never gets added in the crashing version

- The condition guarding this addition is evaluated to true only once
Bug Study 2: Explode2D_AMR

- Wrong calculation of neighbors causes a mismatch in sends and receives
- Happens after the first regridding
- Uintah hangs
- For this example we observe stack traces separated by different time steps
Time Step N
Time Step N+1
Comparison N/N+1

- Just fewer MPI sends and receives
Time Step N+2

- Special event is happening
Time Step N+3

- Uintah hangs and the resulting graph is very different from N+2 and N+1.
- The number of postMPISends() and postMPIRecvs() is not matching.
Summary

- CSTGs can be particularly useful to understand executions when comparing:
  - Working and non-working versions
  - Symmetric events such as Sends/Recvs, Lock/Unlock, New/Delete…
  - Repetitive sequences of events such as time steps
- Stack traces can be aggregated by different time periods, processes, threads…
Lightweight Formal Debugging Framework

- Learn specification automata from traces
- Generate runtime monitors
  - Run on idle cores
  - Schedule non-intrusively
- When monitor throws an exception
  - Start/stop stack trace collection
  - Display CSTGs