SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

CENTER FOR PARALLEL COMPUTING @ UTAH

SECSE, May 18, 2013

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)

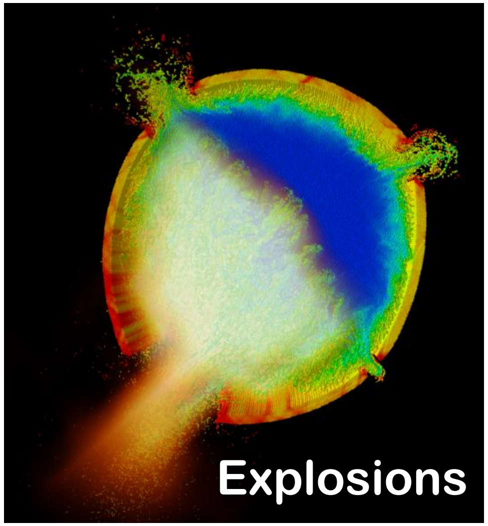
Uintah Runtime Verification (URV) Project

- 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

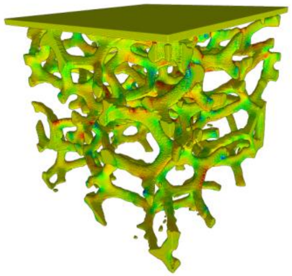
Uintah

Uintah Overview

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

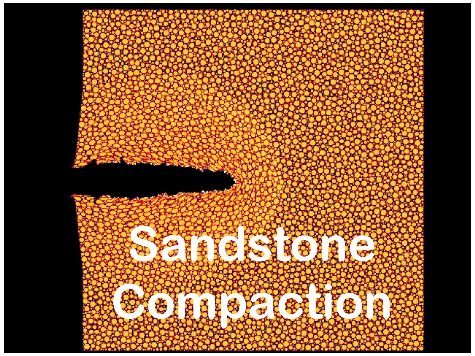


Explosions

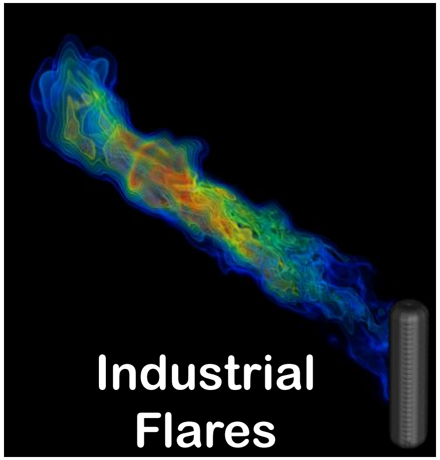


Foam Compaction

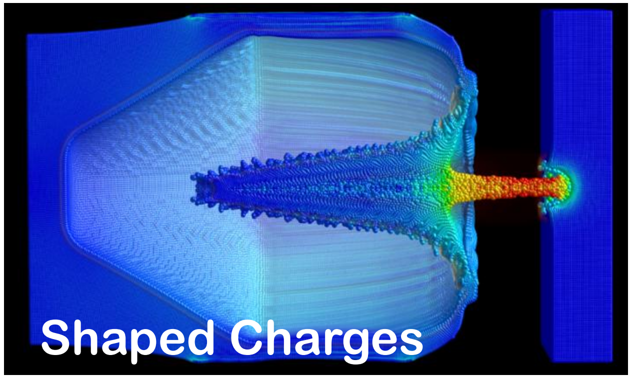
Carbon Capture Clean Coal Boiler



Sandstone Compaction



Industrial Flares



Shaped Charges

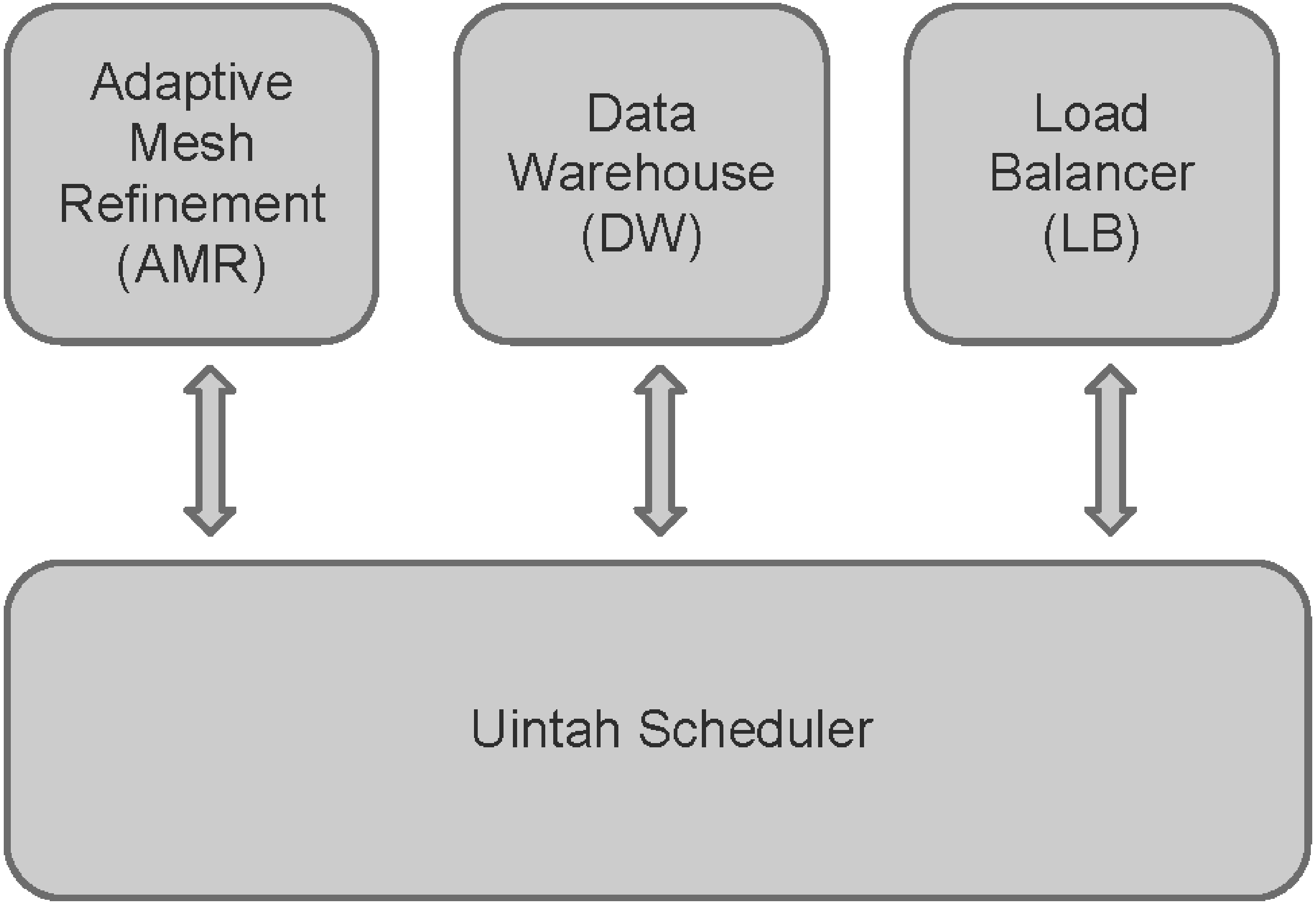
Plume Fires

Uintah Development

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

	Domain Expert (Engineering)	**Infrastructure Expert (Computer Science)**
Focus	Problem, methods	Performance, scalability
Responsibility	Simulation components	Infrastructure components
Contributions	Arches, ICE, MPM, MPM-ICE, etc.	Load balancing, AMR, task-graph scheduling, communication, checkpointing
View of Program	Serial code written for a patch	Parallel infrastructure, MPI, threads, GPU

Modular Architecture of Uintah



Benefits of Modular Architecture

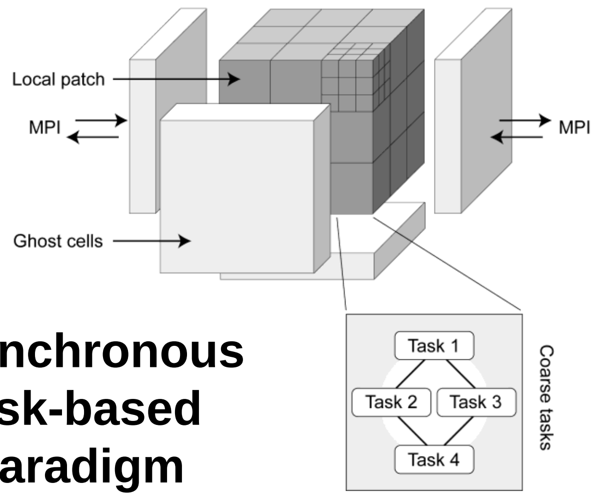
- All applications benefit from infrastructure improvements without change
- Allows infrastructure developers to make improvements without understanding the science of the domain expert
- Successfully scaled from 2K to 512K cores without any changes to applications code

Benefits of Modular Architecture cont.

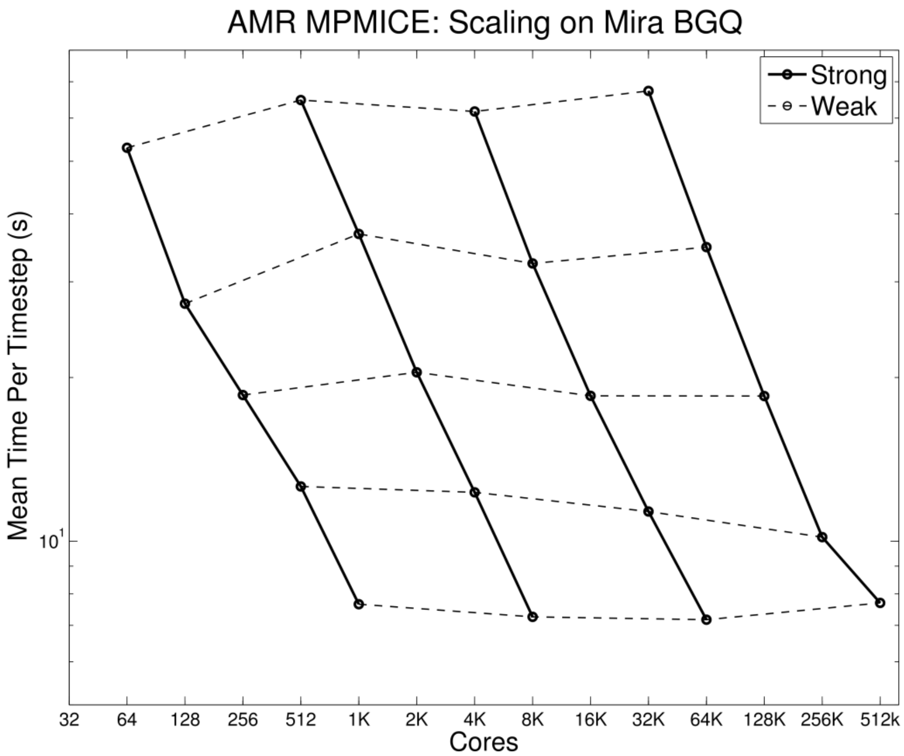
- Infrastructure components easily updated to follow the latest architectures
  - Multicore and GPU support, lock-free data warehouse…
- Adding formal methods is more feasible

Uintah Scalability

Patch-based domain decomposition



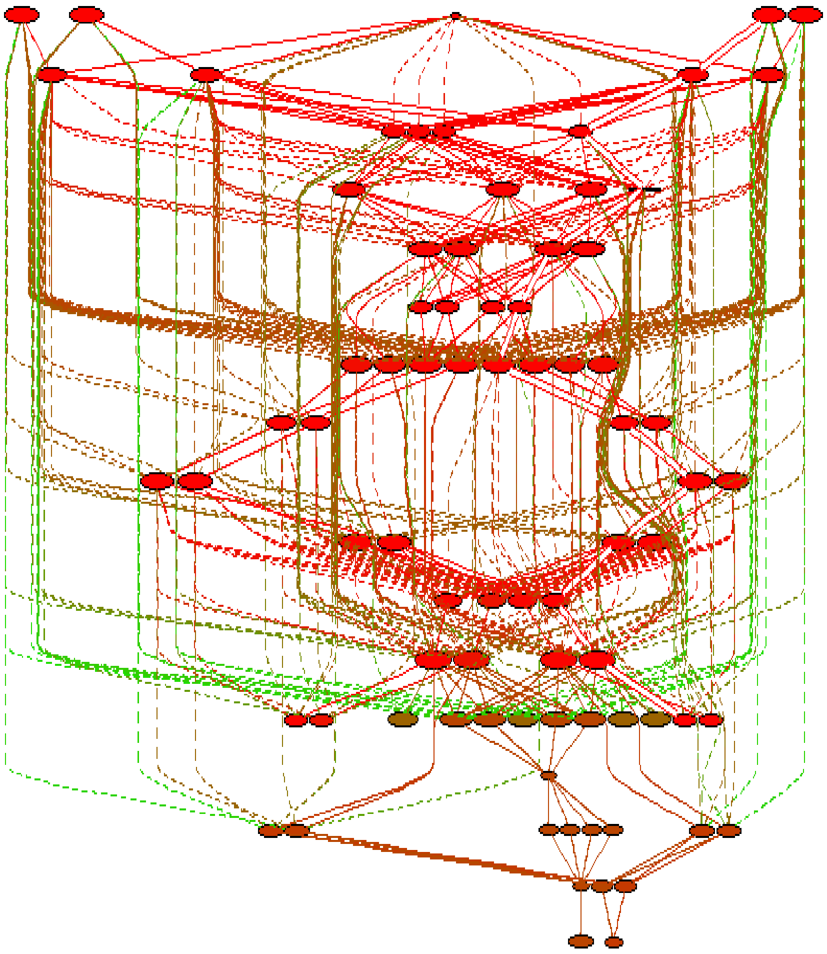
Asynchronous task-based paradigm



- 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



4 patch, single level ICE task graph

Support for Heterogeneous Systems

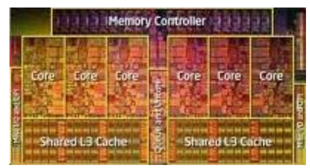
Xeon Phi

GPU

TACC Stampede
1000s of Xeon Phi Coprocessors

+

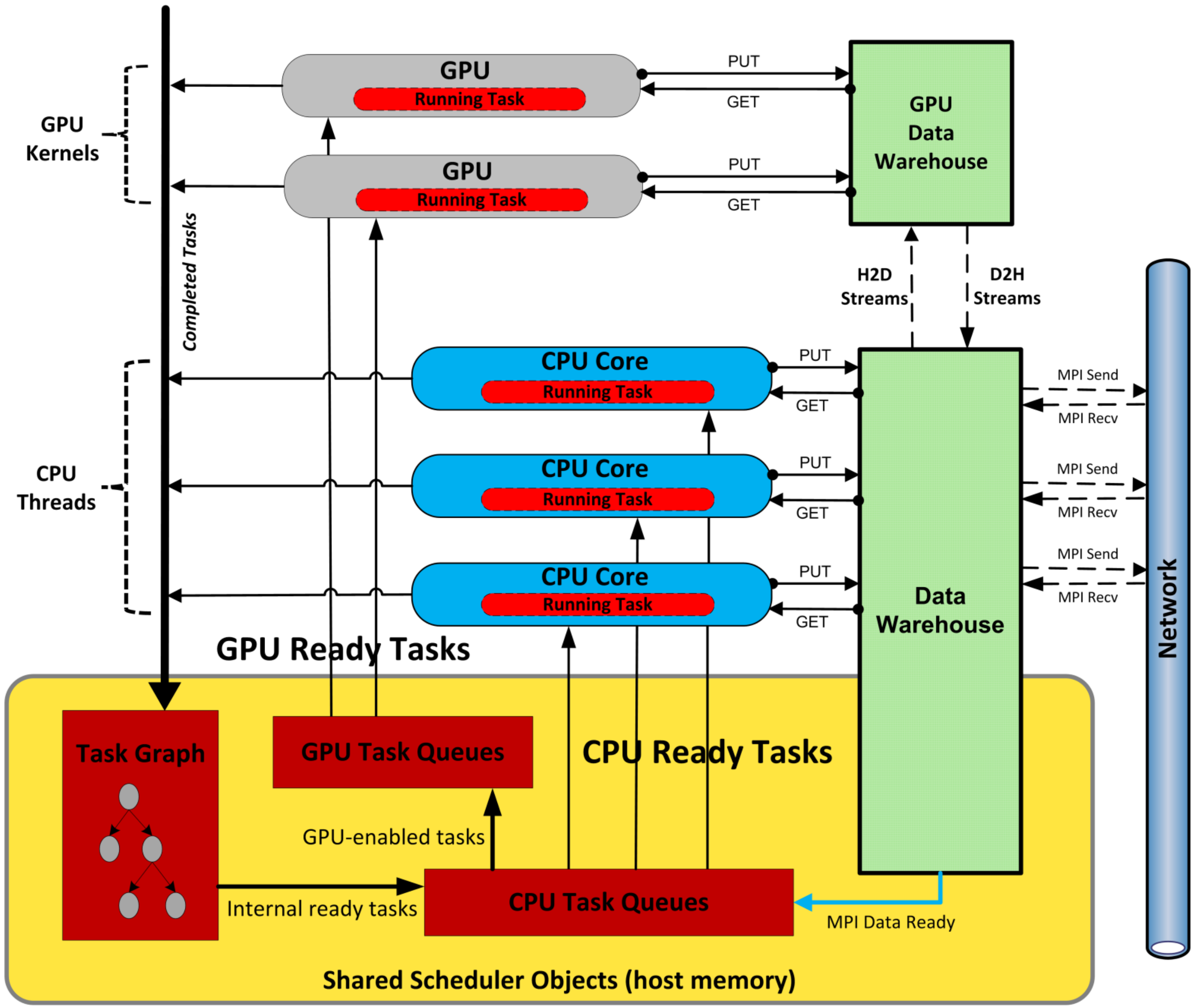
DOE Titan
1000s of GPUs



Multi-core CPU

- Utilize all on-node computational resources
- Uintah's asynchronous task-based approach well suited for coprocessor and accelerator designs
  - Introduce accelerator and coprocessor tasks

Heterogeneous Scheduler & Runtime



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

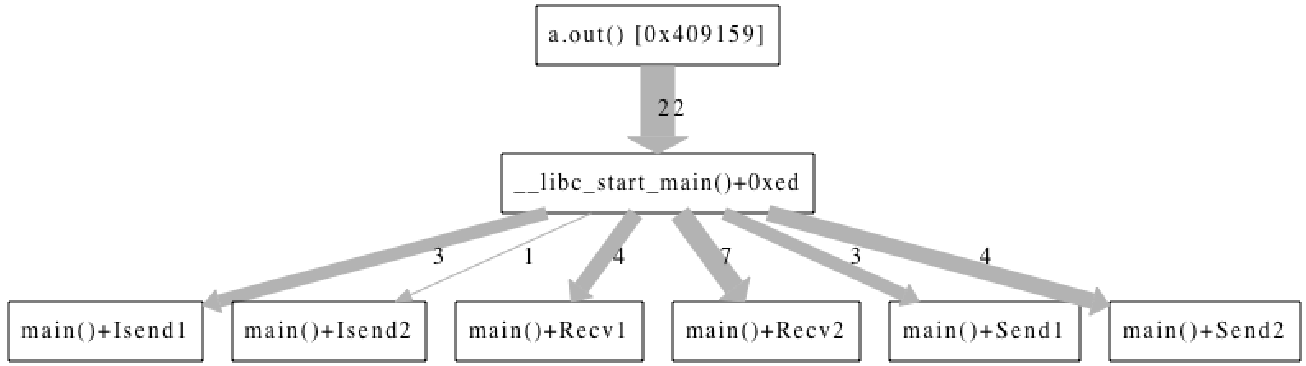
- Stack traces portray a story about the runtime execution of a program by showing
  - call paths leading to a particular function call
  - the number of times a particular path was taken
- Facilitate understanding and root cause analysis of complex bugs

Coalesced Stack Trace Graphs cont.

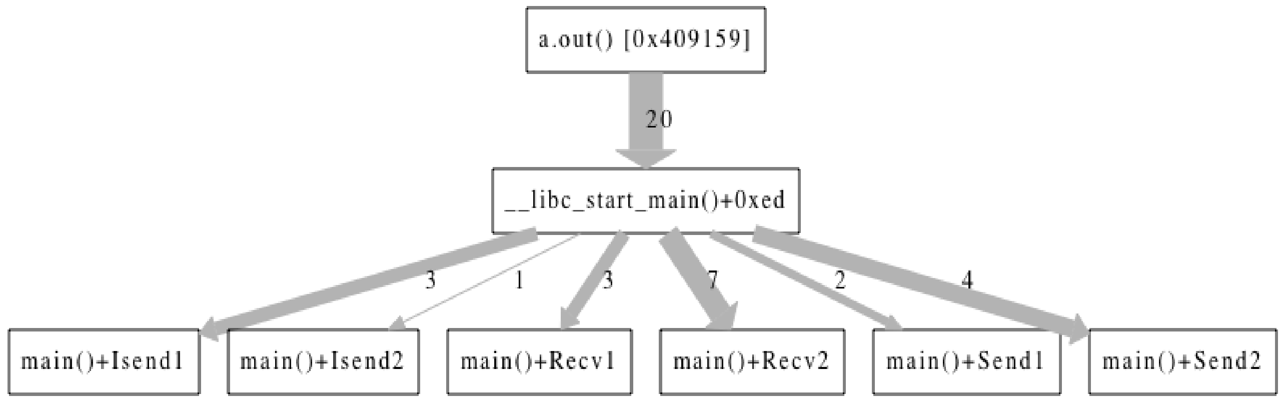
- The number of stack traces collected during execution gets very large
  - Coalesce millions of stack traces using adequate graph representations called Coalesced Stack Trace Graphs (CSTGs)
- Infrastructure developer controls where stack traces should be collected

Basic Idea: Diff CSTGs

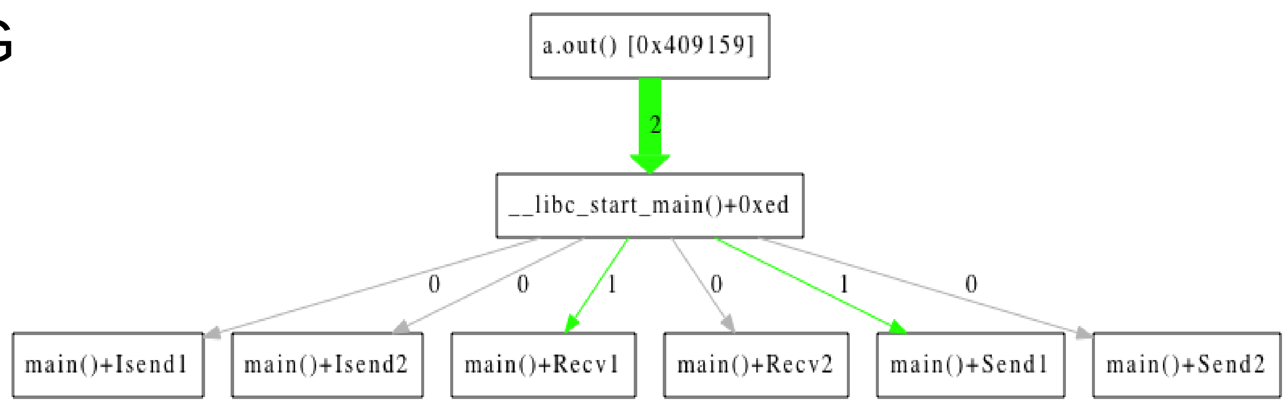
CSTG 1



CSTG 2



Diff CSTG



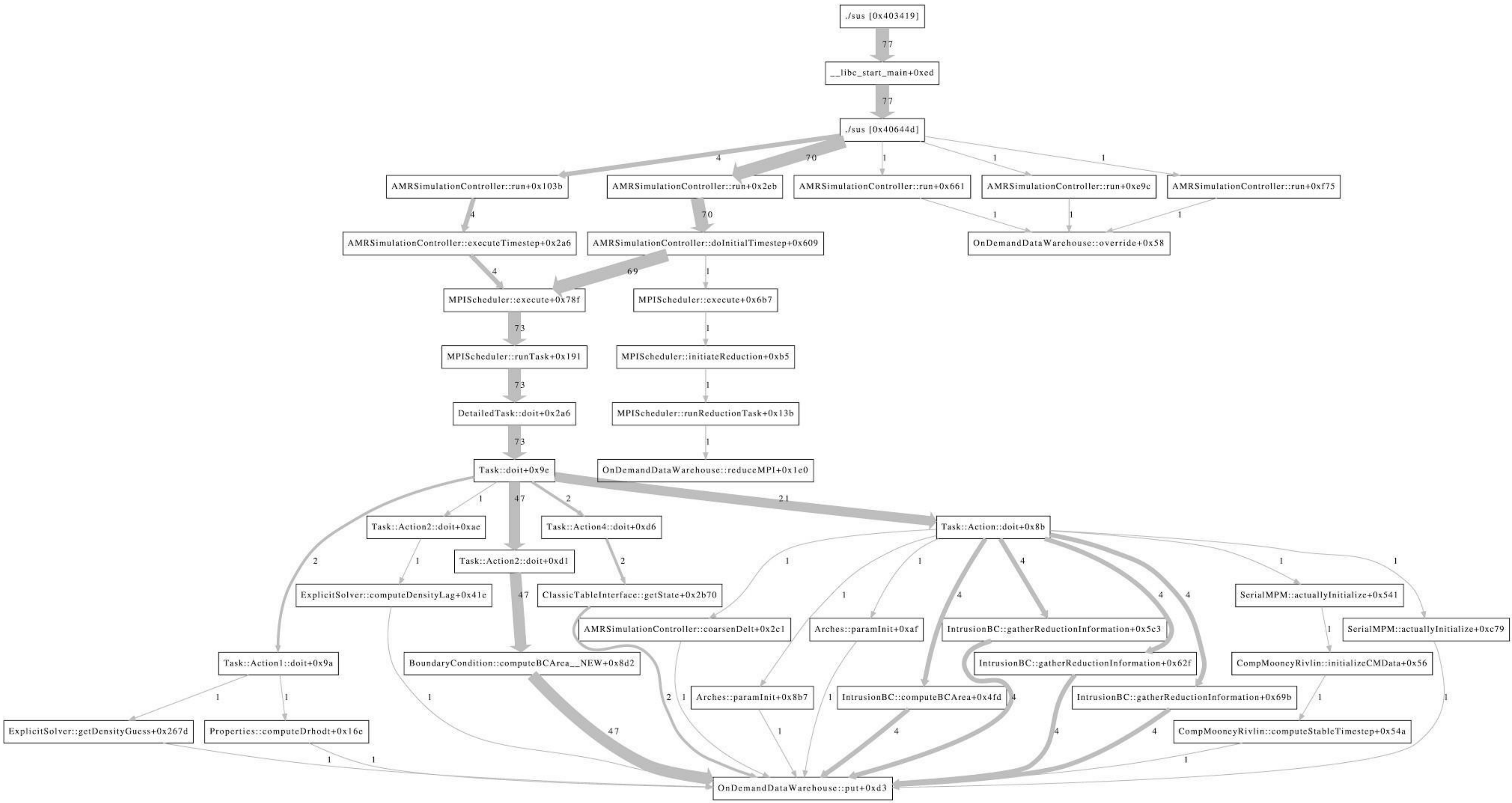
Two Case Studies using Real Bugs

- 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.

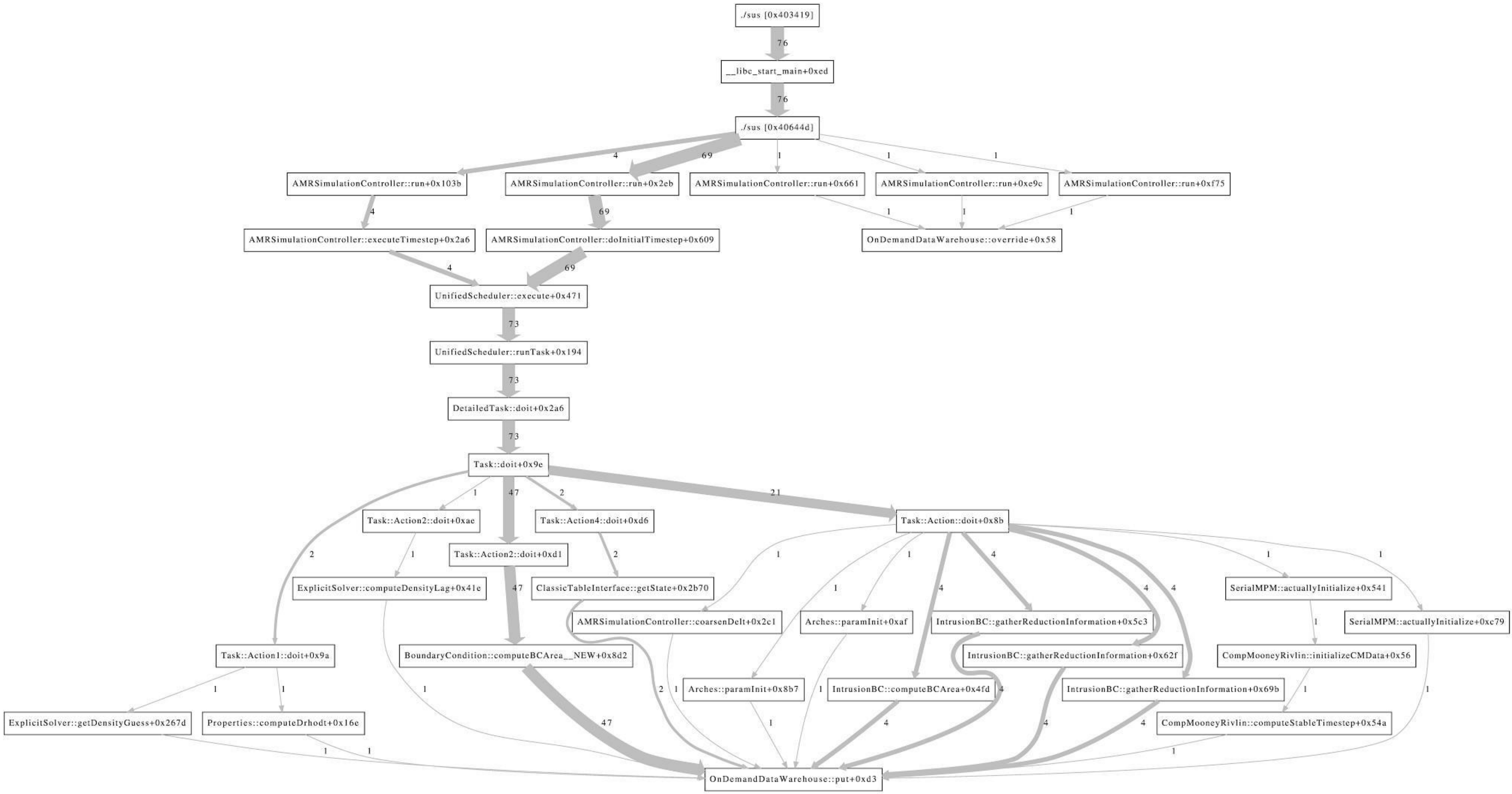
Bug Study 1: MiniBoiler

- 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

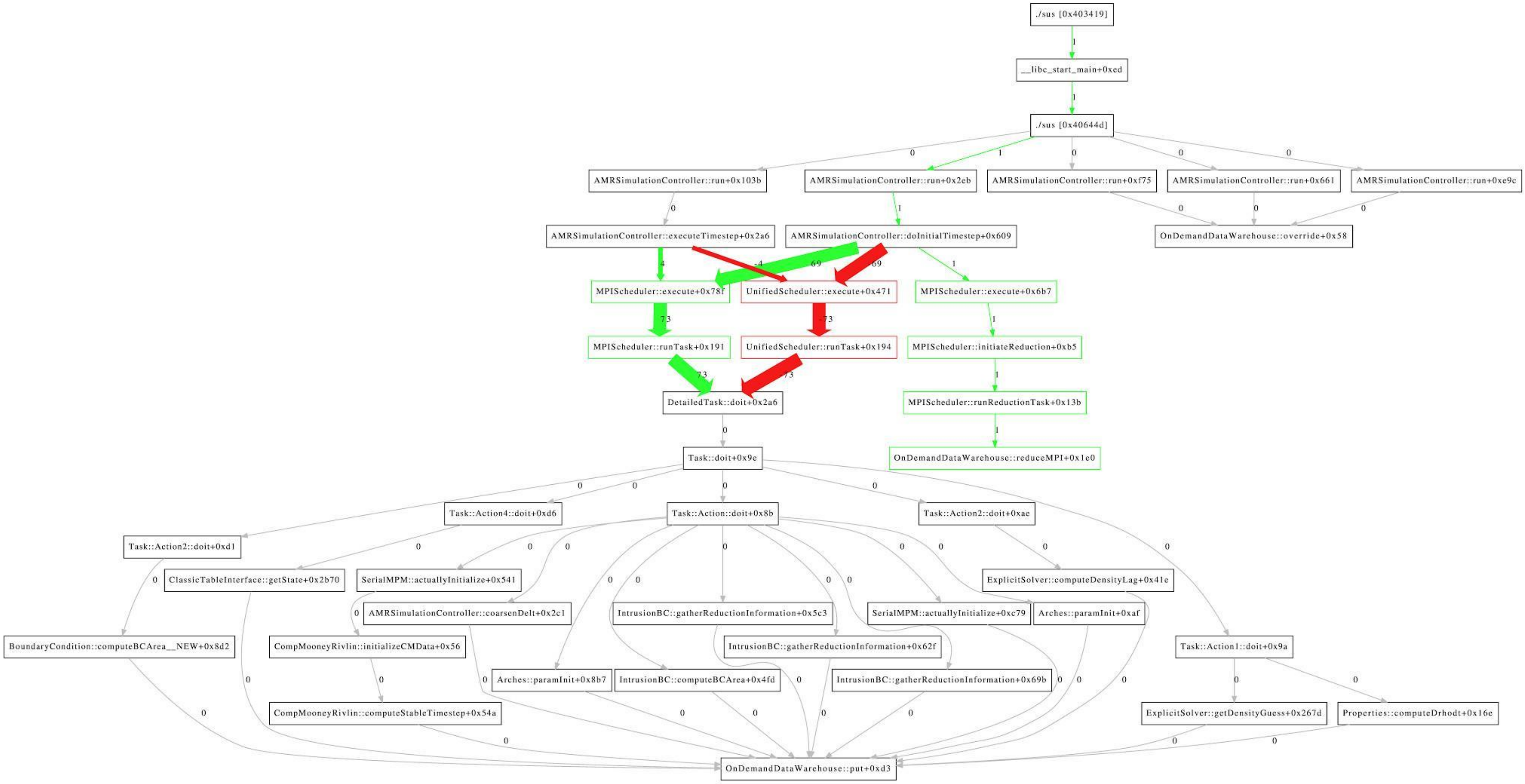
CSTG of MiniBoiler



CSTG of MiniBoiler Crash

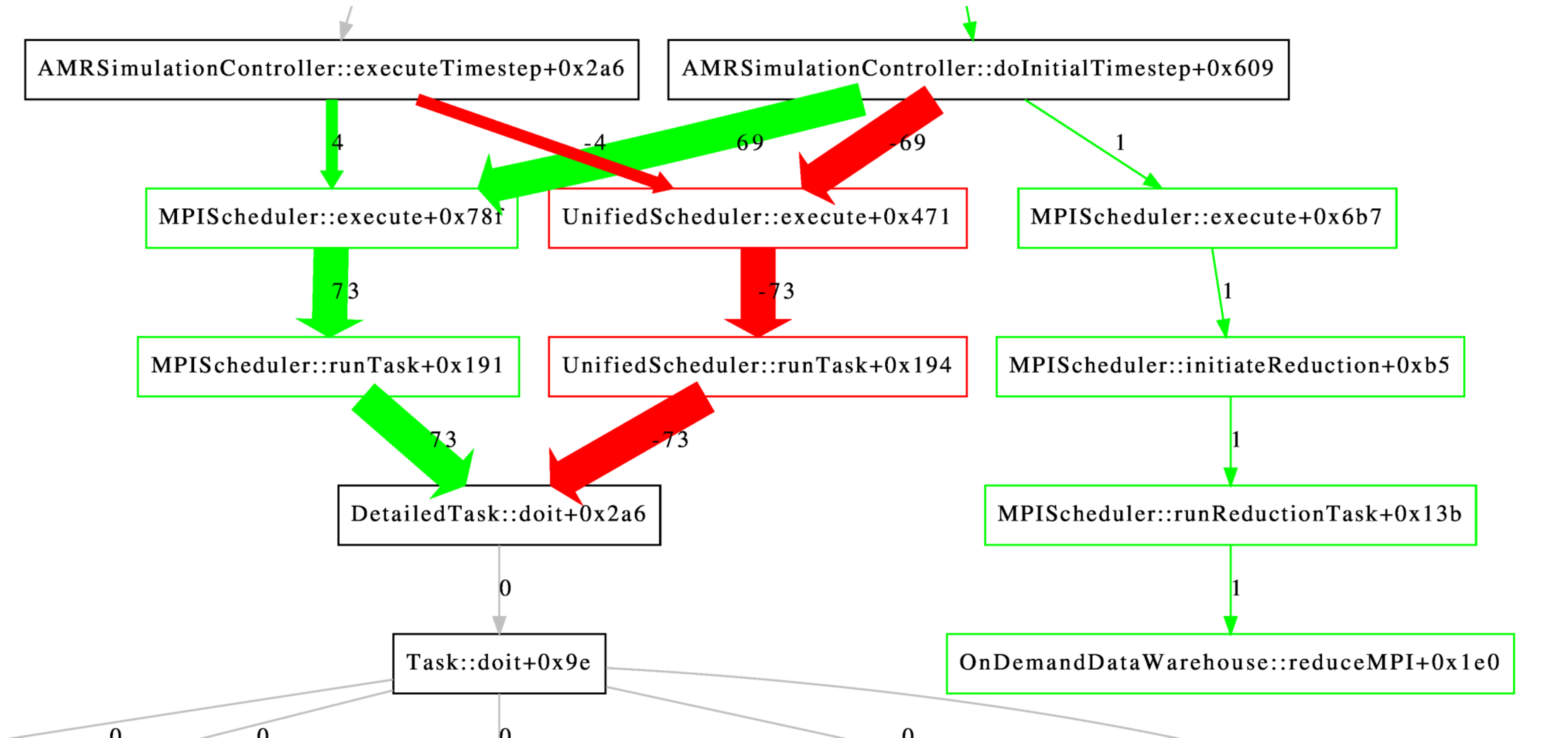


Diff of Good and Bad CSTG



Diff of Good and Bad CSTG cont.

- 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

```
while (...)
...
  if (task->getTask()->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

```
// 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;
}
```

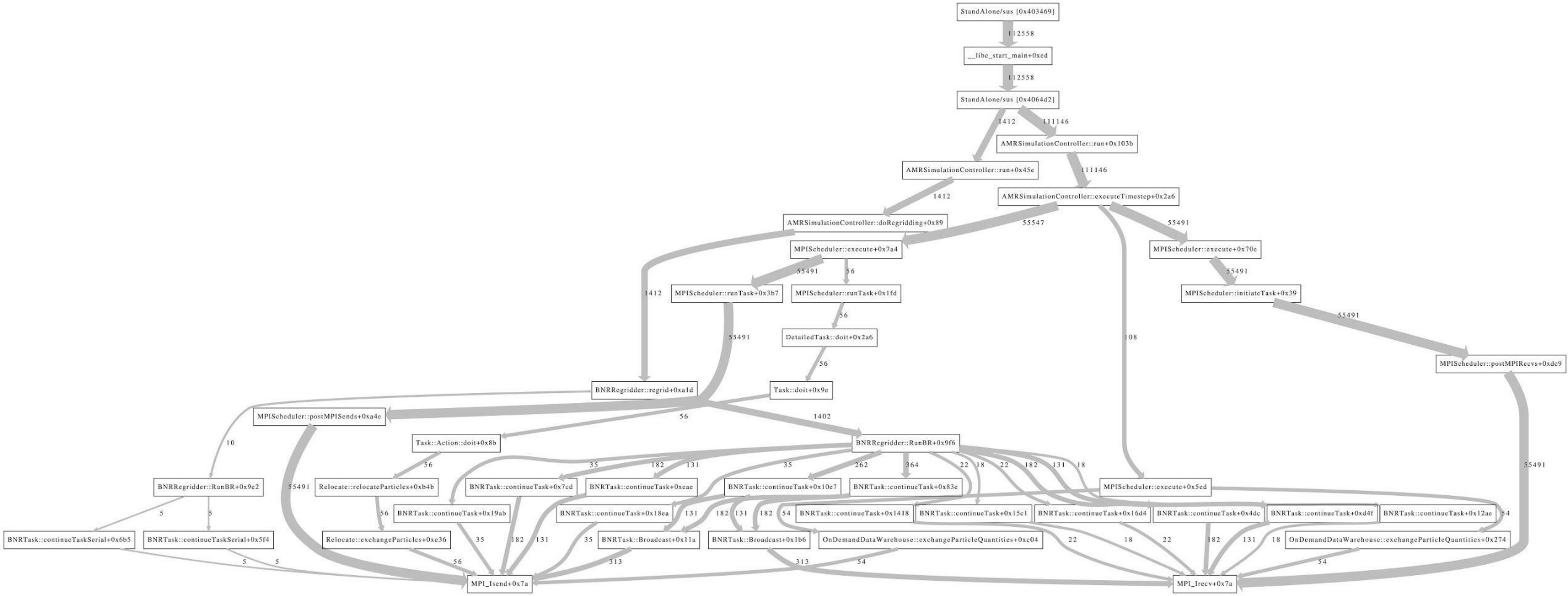
Understanding the Difference cont.

- 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

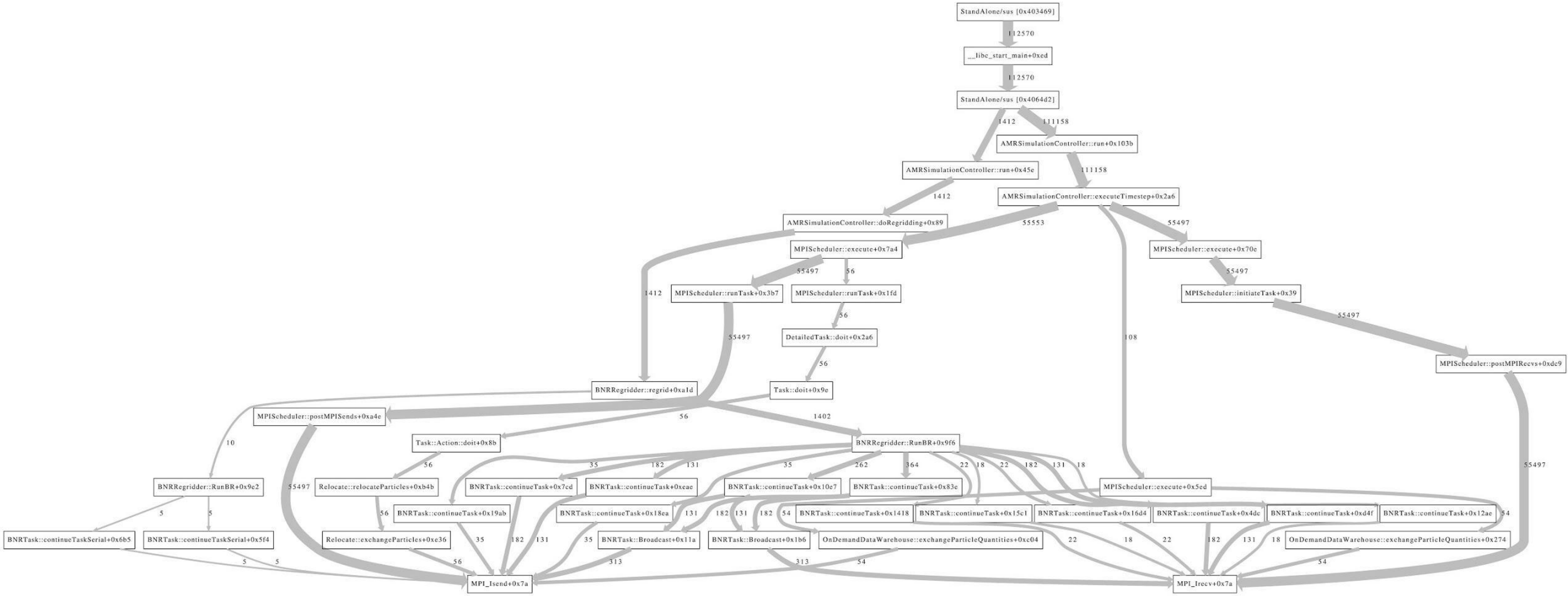
Bug Study 2: Explode2D_AMR

- 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

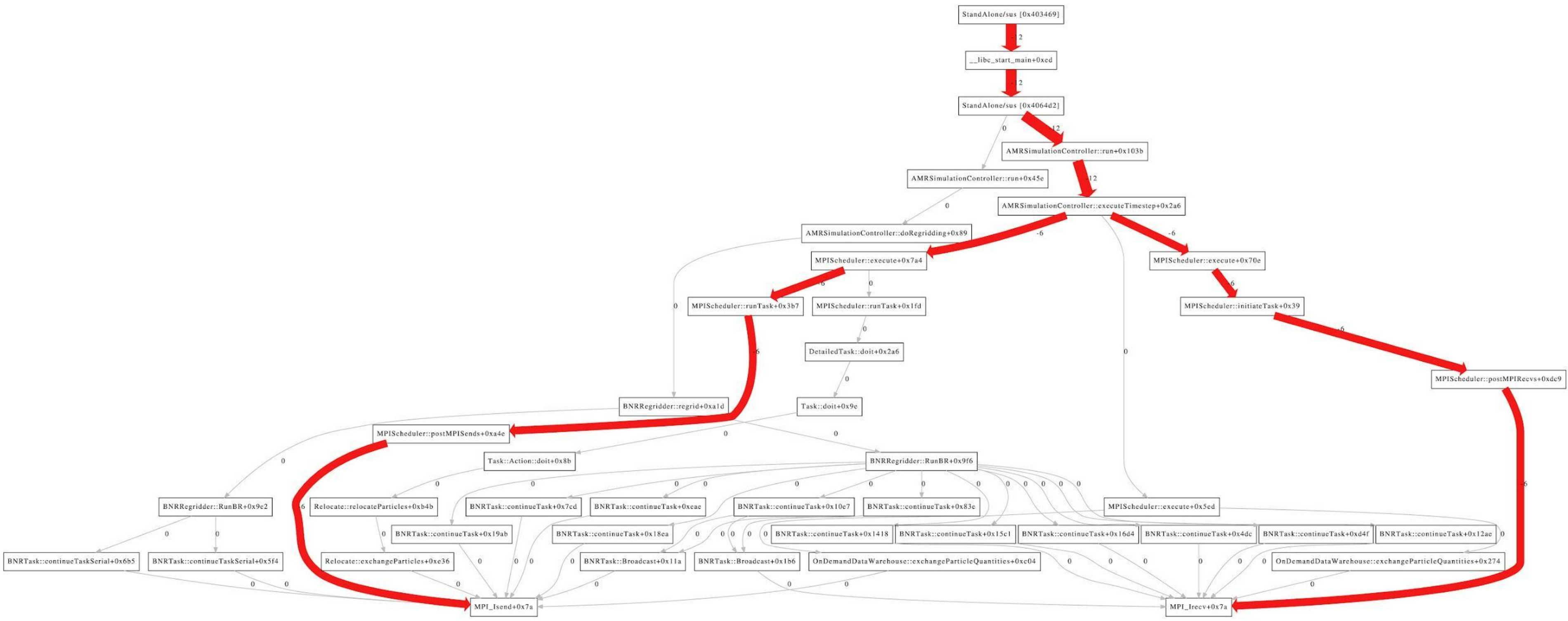

StandAlone/sus [0x403469]
__libc_start_main+0xed
StandAlone/sus [0x4064d2]
AMRSimulationController::run+0x103b
AMRSimulationController::run+0x45e
AMRSimulationController::executeTimestep+0x2a6
AMRSimulationController::doRegridding+0x89
MPIScheduler::execute+0x7a4
MPIScheduler::execute+0x70e
MPIScheduler::runTask+0x3b7
MPIScheduler::runTask+0x1fd
MPIScheduler::initiateTask+0x39
DetailedTask::doit+0x2a6
MPIScheduler::postMPIRecvs+0xdc9
BNRRegridder::regrid+0xa1d
Task::doit+0x9e
MPIScheduler::postMPISends+0xa4e
Task::Action::doit+0x8b
BNRRegridder::RunBR+0x9f6
BNRRegridder::RunBR+0x9e2
Relocate::relocateParticles+0xb4b
BNRTask::continueTask+0x7cd
BNRTask::continueTask+0xeae
BNRTask::continueTask+0x10e7
BNRTask::continueTask+0x83e
MPIScheduler::execute+0x5ed
BNRTask::continueTask+0x19ab
BNRTask::continueTask+0x18ea
BNRTask::continueTask+0x1418
BNRTask::continueTask+0x15c1
BNRTask::continueTask+0x16d4
BNRTask::continueTask+0x4dc
BNRTask::continueTask+0xd4f
BNRTask::continueTask+0x12ae
BNRTask::continueTaskSerial+0x6b5
BNRTask::continueTaskSerial+0x5f4
Relocate::exchangeParticles+0xe36
BNRTask::Broadcast+0x11a
BNRTask::Broadcast+0x1b6
OnDemandDataWarehouse::exchangeParticleQuantities+0xc04
OnDemandDataWarehouse::exchangeParticleQuantities+0x274
MPI_Isend+0x7a
MPI_Irecv+0x7a

Time Step N+1



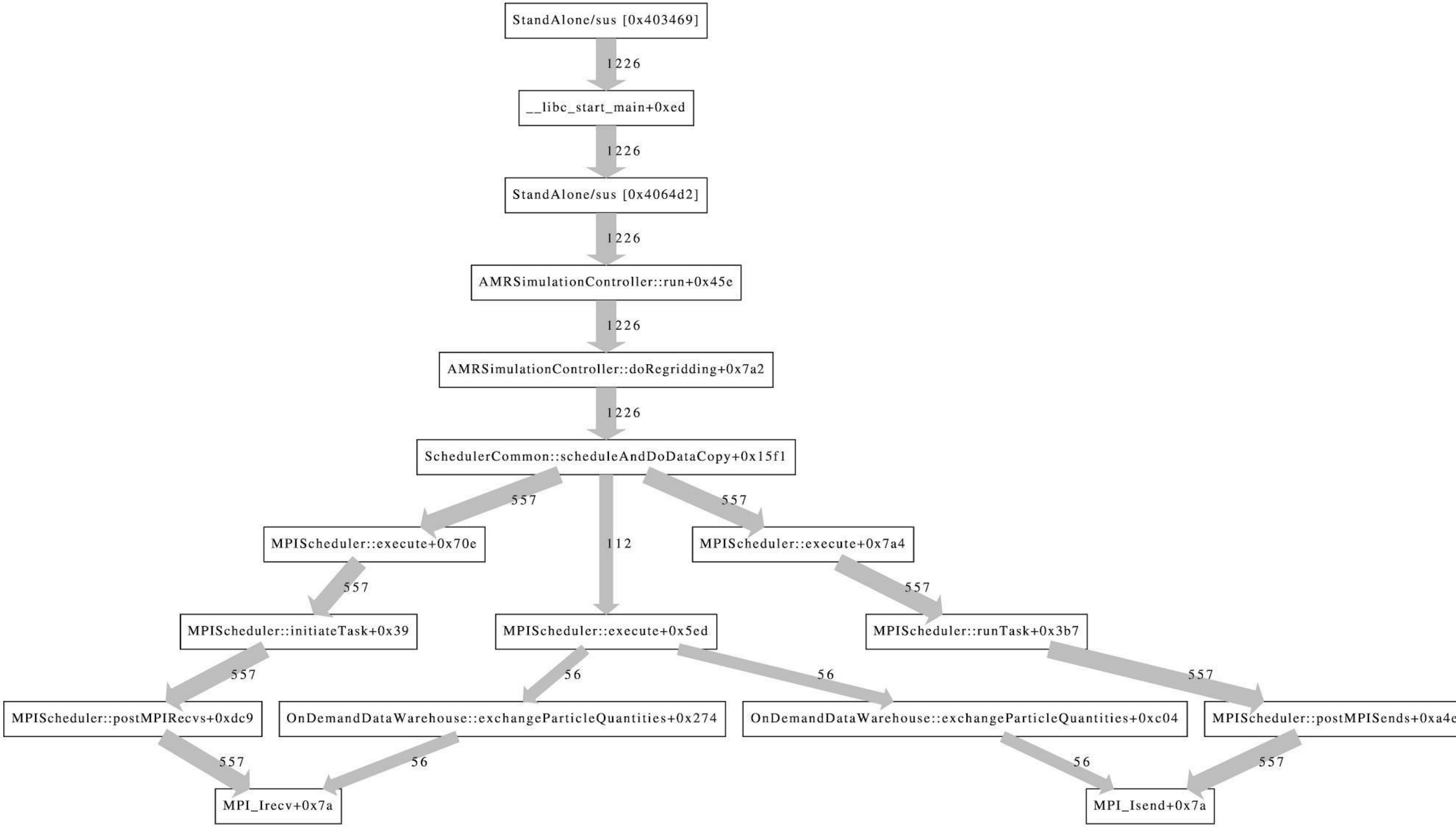
Comparison N/N+1

- Just fewer MPI sends and receives



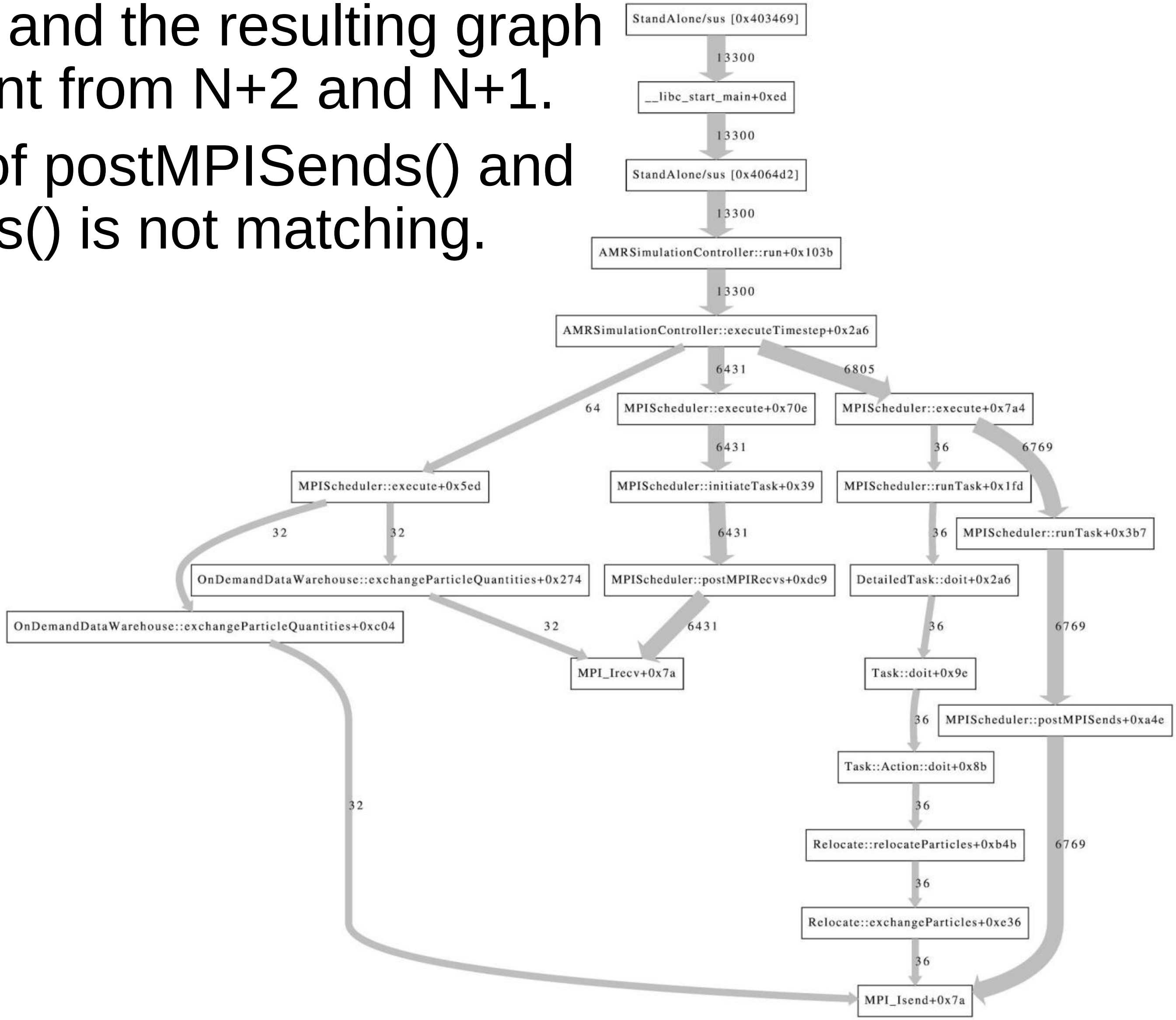
Time Step N+2

- Special event is happening



Time Step N+3

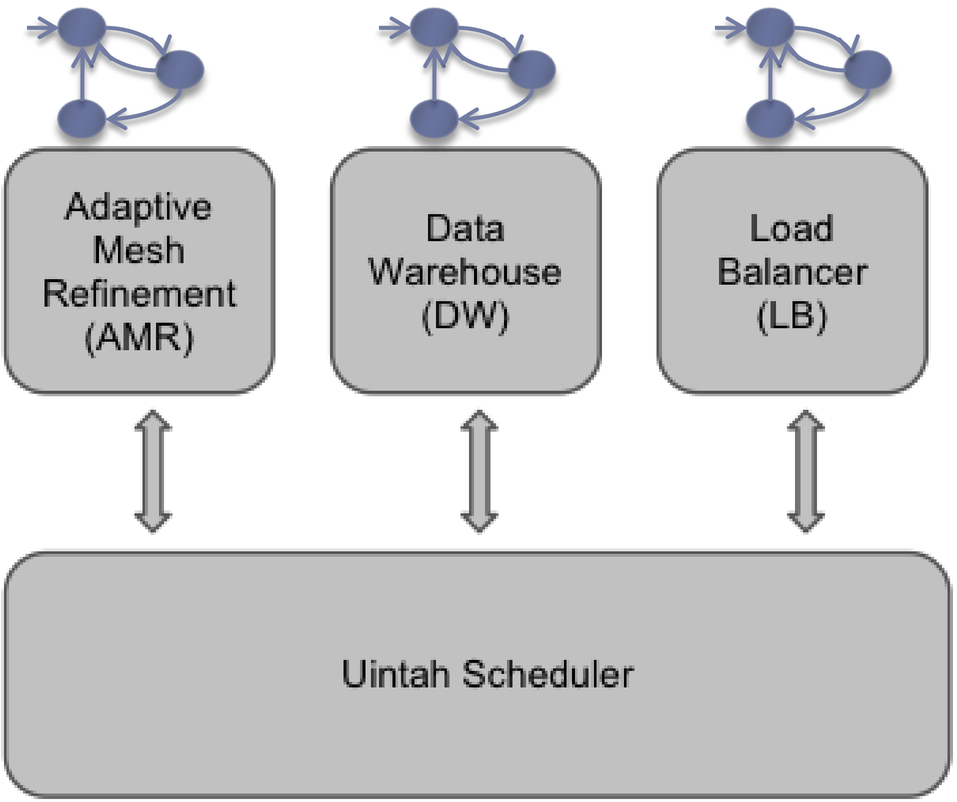
- Uintah hangs and the resulting graph is very different from N+2 and N+1.
- The number of postMPISends() and postMPIRecvs() is not matching.



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