CSTG Case Studies

3 case studies with different uses of CSTGs are shown in this presentation:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Crashes/Hangs at</th>
<th>Comparison applied</th>
<th>Additional info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mini Coal Boiler</td>
<td>Data Warehouse</td>
<td>Different versions</td>
<td>An exception is thrown in function OnDemandDataWarehouse::get() when looking for an element that does not exist in a container.</td>
</tr>
<tr>
<td>MPMICE</td>
<td>Scheduler</td>
<td>Send/Receive match</td>
<td>Wrong calculation of neighbors mismatching sends and receives.</td>
</tr>
<tr>
<td>Poisson2</td>
<td>Scheduler</td>
<td>Different runs</td>
<td>A non deterministic error is causing Uintah to crash sometimes.</td>
</tr>
</tbody>
</table>
Bug Study 1: Mini Coal Boiler
Bug Study 1: Mini Coal Boiler

- An exception is thrown in function OnDemandDataWarehouse::get() when looking for an element that does not exist in a container.
- There are possible 2 reasons this element was not found:
  - it was never inserted or,
  - it was removed from the container.
- We insert stack trace collectors in before every 'put()' or 'remove()' of the container and visualize the result.
- We compare the same graph with a working execution of the same example.
Stack trace graph of working version
Stack trace graph of mini_coal
Crash
Comparison OK / Crashing

There is a path in the working version that leading to the `reduceMPI()` function that never happened in the crashing version.
Understanding the difference

- The two versions use different Schedulers. Reading the code, we can see that:
  - the UnifiedScheduler will never call `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;
  }
  ```
  - the MPIScheduler will sometimes call `initiateReduction()`
    ```cpp
    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);
    }
    ... }
  ```
Understanding the difference

- In `initiateReduction()` there is one condition that is true once and adds the element to the container that was never added in the crashing version.
Bug Study 2: MPMICE
Bug Study 2: MPMICE

- For this example we observe the stack traces separated by different time steps.
Stack trace graph of mini_coal
Crash
Time step N+1

-Almost as before, but nothing meaningful different.
Comparison N / N+1

- Only fewer MPI sends and receives.
Time step N+2

-Special event is happening
Time step N+3

- Problem hangs and the resulting graph is very different from N+2 and N+1.
- The number of `postMPI Sends()` and `PostMPI Recvs()` is not matching.
Bug Study 3: Poisson2
Case Study 3: Poisson2 - Introduction

- This problem calculates the Poisson distribution.
- Different from Poisson 1, Poisson 2 uses a sub-scheduler to do several iterations of the algorithm in one time-step of the 'simulation'.
Poisson 2 - The crash

- When executing Uintah with the poisson2 input, it gives a segmentation error outputting the following (partial) stack trace:

Thread "main"(pid 5512) caught signal SIGSEGV at address (nil) (segmentation violation)
Backtrace:
**1588** 1. /home/diego/workspace/urv_stack_tracer/dbg/lib/libCore_Thread.so(+0x1e57f) [0x572457f]
**1588** 2. /lib/x86_64-linux-gnu/libpthread.so.0(+0xfcb0) [0x7d00cb0]
**1588** 3. /home/diego/workspace/urv_stack_tracer/dbg/lib/libCCA_Components_Schedulers.so
  in Uintah::UnifiedSchedulerWorker::resetWaittime(double)
**1588** 4. /home/diego/workspace/urv_stack_tracer/dbg/lib/libCCA_Components_Schedulers.so
  in Uintah::UnifiedScheduler::execute(int, int)
**1588** 5. /home/diego/workspace/urv_stack_tracer/dbg/lib/libCCA_Components_Examples.so
  in Uintah::Poisson2::timeAdvance(Uintah::ProcessorGroup const*, Uintah::ComputeSubset<Uintah::Patch const*> const*, Uintah::ComputeSubset<int> const*, Uintah::DataWarehouse*, Uintah::DataWarehouse*, Uintah::Handle<Uintah::Level>, Uintah::Scheduler*)
Poisson 2 - The crashing function

• The last Uintah function before the crash is: `Uintah::UnifiedSchedulerWorker::resetWaittime(double)`
• Looking inside the function we see nothing exceptional:

```c
void UnifiedSchedulerWorker::resetWaittime(double start) {
    d_waitstart = start;
    d_waittime = 0.0;
}
```
Poisson 2 - Understanding the crash

- Running Uintah with the same input, it is possible to see that the crash is nondeterministic, it happens most of times in time-step 1 or 2, but occasionally may not even happen.
Poisson - Observing `resetWaittime()` execution with CSTG

- We added some instructions to observe the execution history and paths leading to the crashing function `resetWaitTime()`.
- For this problem, we want to observe executions where the crash happens and when it does not (see the next two slides).
Poisson2 - CSTG crashing at time-step 1

- There are 3 paths leading to the crashing function.
There are only 2 paths leading to the observed function. There were 100 time-steps in total.
Poisson2 - Understanding the difference

- The extra path that contains new (green) functions only occurs in the crashing run.
- It contains the user code `Poisson2::timeAdvance()` that is executed every time-step, but only one of these times the function will cause the call of the observed function `resetWaittime()`.
When debugging with Allinea DDT, at the moment of the crash we can observe in the `execute()` function that the value of the variable `numThreads_` is abnormally high. Two common problems are usually related to this:

- Uninitialized variable or Memory corruption.
Poisson2 - Uninitialized variable?

- *numThreads_* is a private variable of the UnifiedScheduler class. At each time step a new sub-scheduler of the type UnifiedScheduler is created in *Poisson2::timeAdvance()* function, and *numThreads_* needs to be initialized.

- If *numThreads_* is not being initialized it can have any value that is currently in its memory address.
Poisson2 - Yes, Uninitialized variable

- Reading the code from the sub-scheduler creation until the moment `numThreads_` is used for the first time, we see that it was never initialized.
- Very often when a private variable of a class is not initialized, its value is zero, but that is not guaranteed by the compiler. That explains why there is no path from `Poisson2::timeAdvance()` leading to the `resetWaittime()` but just when there is a crash.
  - The condition to `resetWaittime()` be executed coming from `Poisson2` is that `numThreads_` is greater than 0.
Poisson2 - Yes, Unitialized variable (confirmation)

- Other tools also help us to observe the uninitialized variable problem. This is a partial view from what we get from *Valgrind* just before the crash:

```plaintext
==5512== Use of uninitialised value of size 8
==5512==  at 0x74C2D99: Uintah::UnifiedSchedulerWorker::resetWaittime(double) (UnifiedScheduler.cc:2114)
==5512==  by 0x74BDBE7: Uintah::UnifiedScheduler::execute(int, int) (UnifiedScheduler.cc:477)
==5512==  by 0xC7E568F: Uintah::Poisson2::timeAdvance(...) (Poisson2.cc:193)
...
==5512== Invalid write of size 8
==5512==  at 0x74C2D99: Uintah::UnifiedSchedulerWorker::resetWaittime(double) (UnifiedScheduler.cc:2114)
==5512==  by 0x74BDBE7: Uintah::UnifiedScheduler::execute(int, int) (UnifiedScheduler.cc:477)
==5512==  by 0xC7E568F: Uintah::Poisson2::timeAdvance(...) (Poisson2.cc:193)
...
```
Poisson2 - Invalid write

- Invalid write given by Valgrind means that the program was trying to write to a heap memory that was not allocated. Looking again at resetWaittime() we see:

```c++
void UnifiedSchedulerWorker::resetWaittime(double start)
{
    d_waitstart = start; // -- Crashing point
    d_waittime = 0.0;
}
```

- The variable `d_waitstart` is only allocated at the function problemSetup() and the user never calls it.
  - problemSetup() is also where numThreads_ is initialized by the system.
Poisson2 - Conclusion

● This is a user problem related to incorrect use of the Uintah system.
● It is nondeterministic and may stay invisible for a while.
● The use of the CSTGs helped us to understand better the problem and faster observe abnormal paths leading to the crashing function.
● Combined with other traditional debugging tools, we can easily correct the problem.