Debugging PGAS Languages: Challenges and Approaches

Shivali Agarwal, IRL
R.K. Shyamasundar, TIFR

Classical Debugging

- Breakpoints (line, method, exception...)
- Stepping (typically works for sequential function calls)
- Examine call stack
- List variables
- Watchpoints
- Expression evaluation

Debugging in (A)PGAS Languages

```
L1 foo(){
L2   int i;
L3   async_task (place_1) {
L4       async_task (place_2) {....}
L5   }
L6   bar();
L7 }
```

```
L1 foo(){
L2   int i;
L3   for (j:1..n) {
L4       async_task (place_1) {....}
L5   }
L6   bar();
L7 }
```

- Are conventional breakpoints and stepping notions enough for debugging user level tasks?

Motivation

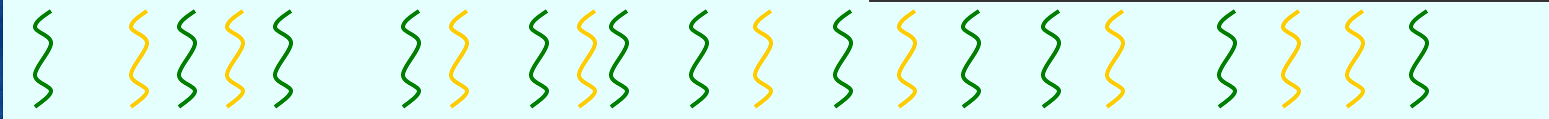
What does stepping into a parallel region mean?

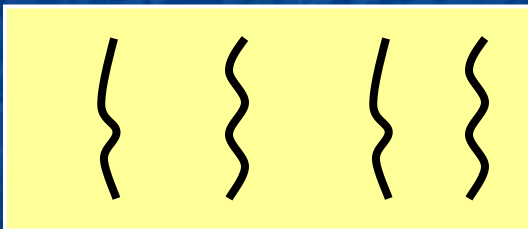
```
L1 foo(){
L2   int i;
L3   for (j:1..n) {
L4       async_task (place_1) {....}
L5   }
L6   bar();
L7 }
```

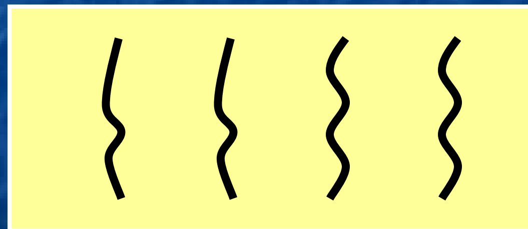
Should we bother about it?

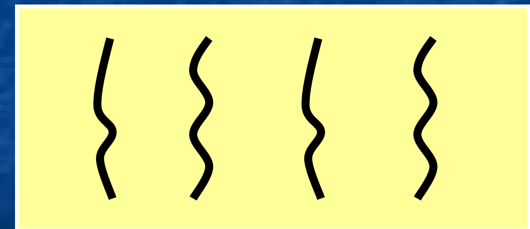
What role does scheduling policy play?

How can number of event be reduce to achieve scalability?









System thread   Scheduled task   Pending task

Challenges in Debugging PGAS Languages

- How to control and better visualize interleavings because of user level tasks?
- New constructs for control and synchronization require special support.
- Scalability is a big challenge because user level tasks can be huge in number resulting in large number of generated debug events.

Our Approach

- Debugger capabilities should be as close in nature to the programming paradigm as possible.
- We attack scalability problem by proposing ways to navigate a parallel program modeled on usage of language constructs and programmer’s view of flow control.
- This we believe will reduce the number of generated events because the user will have a readymade way of doing things which he would otherwise have done by listening on lot of events.
- The proposals we make are not tied to any particular view of processes and threads.

Main Contributions

- Interpreting stepping for atleast two different types of schedule policies. *Schedulepoints* for better control and visualization of interleavings

- New forms of stepping a program, namely, *step-finishpoint* and *phased-step* are proposed that correspond to specialized constructs present in PGAS languages.

- We show how the language runtime can be used to advantage to discover data races for variables that synchronize using atomic sections.

- Finally, a notion of *TraceArrays* is given that uses the standard tracing techniques to effectively debug distributed arrays.

Stepping : FIFO scheduling

1) STEP INTO: Unlike a traditional step into a function, step into an asynchronous task cannot guarantee the control of execution to transfer to task body

2) STEP OVER: A step over can guarantee task creation but not task scheduling.

Stepping : Work Stealing

1) STEP INTO: Step into an asynchronous task is analogous to stepping into a traditional function call with a difference that the step return is not well defined.

2) STEP OVER: A step over an async task shall mean the execution of the entire remaining depth until either a breakpoint is hit or some wait dependency like a sync point in Cilk occurs or task terminates and worker gets idle.

How to interpret stepping for asynchronous tasks?

FIFO: A ***step into*** should enable the user to get notified when the task gets scheduled and control in the parent thread should move to next instruction.

Work Stealing: A ***step into*** is like a function call and user should get notified when continuation gets scheduled.
A ***step over*** should bring the control to the continuation and the spawning async just continues as is.

STEP INTO: USER WANTS TO STEP INTO THE ASYNC TASKS WITH STEP RETURN BEING AS THE CONTINUATION.

STEP OVER: USER WANTS TO CONTINUE IN THE MAIN TASK AFTER CREATING THE ASYNC TASKS.

Stepping

step into

- Main task
- Task (suspended at L6)
- **Task at L3**

```
L1 foo(){
L2   int i;
L3   async_task (place_1)  ←
L4   {    async_task (place_2) {....}
L5   }
L6   bar();
L7 }
```

FIFO

step into

- Main task
- Task (suspended at L4)
- **Task continuation at L6**

```
L1 foo(){
L2   int i;
L3   async_task (place_1)  ←
L4   {    async_task (place_2)
L5          {....} }
L6   bar();
L7 }
```

Work Steal

SCHEDULEPOINTS

A ***schedulepoint*** is associated at points of asynchronous task creation or continuation.

- In case of FIFO, it tells the queued task to suspend as soon as it gets scheduled.
- In case of WS, it tells the continuation to suspend as soon as it gets scheduled.

1) We can get a finer level of control over asynchronous tasks in spite of not being in control of underlying system scheduling.

2) This can be implemented with the help of language runtime.

3) This capability shall enable users to simulate desired interleavings especially for work stealing scheduling.

4) In case of queue scheduling, this is very useful in case we want to chase down some task to see its interactions with descendants and study interference.

Debugging Synchronization constructs of PGAS

- **Step-finishpoint**

  -Step from a finish construct to the nested one or the next in sequence in the same task.

  -Returns to parent finish.

  -Note that it is difficult to achieve this using breakpoints.

- **Phased-step**

  One step corresponds to a phase, so all tasks executing in phase suspend after completing the phase computation.

- **Atomic Watchpoint**

  A watchpoint to indicate if the variable is being accessed in an atomic section. A variable can also be watched for being accessed in non-atomic way.

Step-finishpoint

```
foo(){
    finish {.. stmt1;..
            finish bar();}
}
bar(){
    for (i: 1..N)
       async_task {
           ... stmt2; ...}
}
```

```
foo(){
    finish {.. stmt1;..
               if (flag)
                  finish bar();
               else finish baz();}
}
```

Useful for:
- **debugging deadlock**
- **understanding program flow**

Phased-step

- Useful for clock implementation of barriers because it may be difficult to set breakpoints to achieve the same effect.

Useful for:
-inspecting variables after a phase
-understanding program flow
-debugging improper synchronization

Atomic Watchpoint

- User may not declare atomic sections in a consistent way. A normal watchpoint may be too tedious to find the issues that arise out of this.

Useful for:
-debugging data races
-Debugging atomic implementation and study fairness

Debugging Distributed Arrays

- TraceArrays :

  To trace changes to array elements within a code block. The output can be observed through data visualizers. Properties like percentage of remote accesses etc. can be analyzed.

  Trace is with respect to any task that executes that code block.

  Trace can be taken for specified group of tasks or places.

How to implement?

- Debug Engine should understand finishpoints
- Debug Engine should interact with language runtime for phased step.
- Tracing can be supported at various levels.
- Insert breakpoints at continuations in runtime to support schedulepoints.
- Atomic watchpoints can be done by using classical watchpoints through sophisticated debugger and runtime support.

Scalability Issues

- Efficiently start a large number of processes under the control of the debugger.
- Manage communication between debugger and processes in a scalable manner.
- Manage large numbers of debug events in order to prevent overwhelming the user interface.
- Cleanly terminate the debug engines and parallel program being debugged

Proposed Debugger capabilities-Conclusions

- More well defined stepping for asynchronous tasks.
- Better control over interleavings.
- Stepping that mirrors language constructs shall reduce number of events thus giving scalability.
- Tracing array elements for specified code sections should result in manageable trace files which can be used effectively to analyze data.

Thank You !!