COMP 322: Fundamentals of Parallel Programming

Lecture 15: Phasers, Point-to-point Synchronization

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu, shams.imam@twosigma.com

http://comp322.rice.edu/

Worksheet #14 solution: Data-Driven Tasks

For the example below, will reordering the five async statements change the meaning of the program (assuming that the semantics of the reader/writer methods depends only on their parameters)? If so, show two orderings that exhibit different behaviors. If not, explain why not. (You can use the space below this slide for your answer.)

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
  async await(left) leftReader(left); // Task3
  async await(right) rightReader(right); // Task5
  async await(left,right)
        bothReader(left,right); // Task4
  async left.put(leftWriter()); // Task1
  async right.put(rightWriter());// Task2
}
```

No, reordering consecutive async's will never change the meaning of the program, whether or not the async's have await clauses.

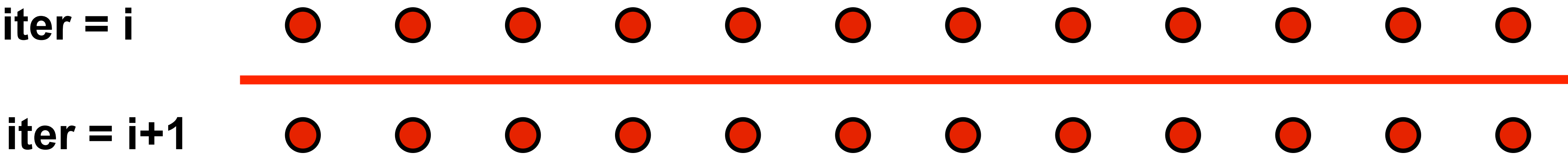
What is Deadlock?

- A parallel program execution contains a deadlock if some task's execution remains incomplete due to it being *blocked indefinitely* awaiting some condition
- Example of a program with a deadlocking execution

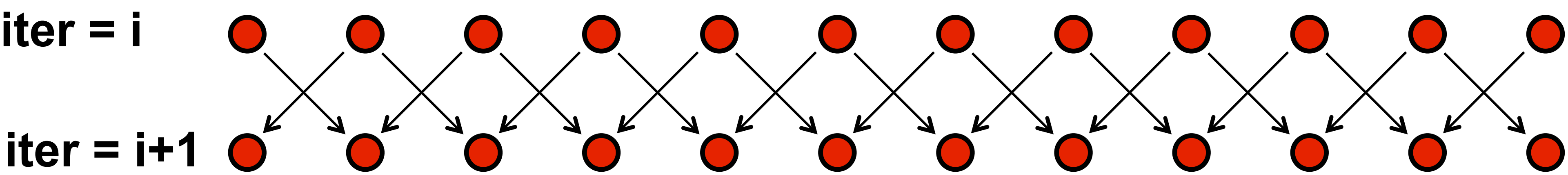
```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
 async await ( left ) right.put(rightBuilder()); // Task1
  async await ( right ) left.put(leftBuilder()); // Task2
}
```

- In this case, Task1 and Task2 are in a deadlock cycle.
- Can you create a deadlock example using any other blocking constructs that you've learned thus far in COMP 322?
  - e.g., end-finish, future-get, next (barrier) ?

Barrier vs Point-to-Point Synchronization for One-Dimensional Iterative Averaging Example



Barrier synchronization



Point-to-point synchronization

Question: when can the point-to-point computation graph result in a smaller CPL than the barrier computation graph?
Answer: when there is variability in the node execution times.

Phasers: a unified construct for barrier and point-to-point synchronization

- **HJ phasers unify barriers with point-to-point synchronization**
  - Inspiration for java.util.concurrent.Phaser
- **Previous example motivated the need for "point-to-point" synchronization**
  - With barriers, phase i of a task waits for *all* tasks associated with the same barrier to complete phase i-1
  - With phasers, phase i of a task can select a subset of tasks to wait for
- **Phaser properties**
  - Support for barrier and point-to-point synchronization
  - Support for dynamic parallelism --- the ability for tasks to drop phaser registrations on termination (end), and for new tasks to add phaser registrations (async phased)
  - A task may be registered on multiple phasers in different modes

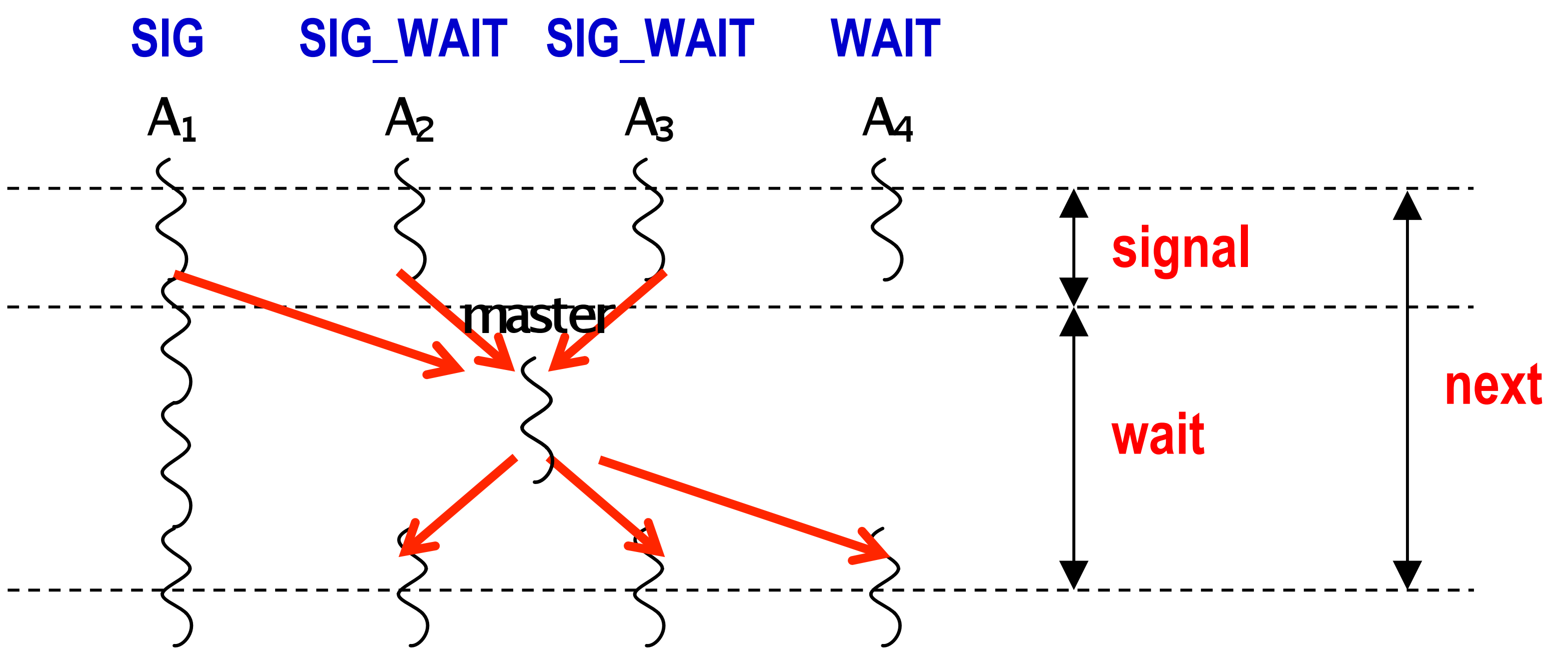
Simple Example with Four Async Tasks and One Phaser

```
finish (() -> {
  ph = newPhaser(SIG_WAIT); // mode is SIG_WAIT
  asyncPhased(ph.inMode(SIG), () -> {
    // A1 (SIG mode)
    doA1Phase1(); next(); doA1Phase2(); });
  asyncPhased(ph.inMode(SIG_WAIT), () -> {
    // A2 (SIG_WAIT mode)
    doA2Phase1(); next(); doA2Phase2(); });
  asyncPhased(ph.inMode(HjPhaserMode.SIG_WAIT), () -> {
    // A3 (SIG_WAIT mode)
    doA3Phase1(); next(); doA3Phase2(); });
  asyncPhased(ph.inMode(HjPhaserMode.WAIT), () -> {
    // A4 (WAIT mode)
    doA4Phase1(); next(); doA4Phase2(); });
});
```

Computation Graph Schema Simple Example with Four Async Tasks and One Phaser

Semantics of next depends on registration mode

- SIG_WAIT: **next = signal + wait**
- SIG: **next = signal**
- WAIT: **next = wait**



Summary of Phaser Construct

- **Phaser allocation**
  - **HjPhaser ph = newPhaser(mode);**
    - **Phaser ph is allocated with registration mode**
    - **Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)**
- **Registration Modes**
  - **HjPhaserMode.SIG, HjPhaserMode.WAIT, HjPhaserMode.SIG_WAIT, HjPhaserMode.SIG_WAIT_SINGLE**
    - **NOTE: phaser WAIT is unrelated to Java wait/notify (which we will study later)**
- **Phaser registration**
  - **asyncPhased (ph_1.inMode(<$mode_1$>), ph_2.inMode(<$mode_2$>), … () -> <stmt>)**
    - **Spawned task is registered with ph_1 in $mode_1$, ph_2 in $mode_2$, …**
    - **Child task's capabilities must be *subset* of parent's**
    - **asyncPhased <stmt> propagates all of parent's phaser registrations to child**
- **Synchronization**
  - **next();**
    - **Advance each phaser that current task is registered on to its next phase**
    - **Semantics depends on registration mode**
    - **Barrier is a special case of phaser, which is why *next* is used for both**

Capability Hierarchy

- A task can be registered in one of four modes with respect to a phaser: SIG_WAIT_SINGLE, SIG_WAIT, SIG, or WAIT. The mode defines the set of capabilities — signal, wait, single — that the task has with respect to the phaser. The subset relationship defines a natural hierarchy of the registration modes. A task can drop (but not add) capabilities after initialization.

SIG_WAIT_SINGLE = { signal, wait, single }

SIG_WAIT = { signal, wait }

SIG = { signal }

WAIT = { wait }

forall barrier is just an implicit phaser!

```
forallPhased(iLo, iHi, (i) -> {
  S1; next(); S2; next();{...}
});
```

is equivalent to

```
finish(() -> {
  // Implicit phaser for forall barrier
  final HjPhaser ph = newPhaser(SIG_WAIT);
  forseq(iLo, iHi, (i) -> {
    asyncPhased(ph.inMode(SIG_WAIT), () -> {
      S1; next(); S2; next();{...}
    }); // next statements in async refer to ph
});
```

Left-Right Neighbor Synchronization (with m=3 tasks)

```
finish(() -> { // Task-0
    final HjPhaser ph1 = newPhaser(SIG_WAIT);
    final HjPhaser ph2 = newPhaser(SIG_WAIT);
    final HjPhaser ph3 = newPhaser(SIG_WAIT);
    asyncPhased(ph1.inMode(SIG),ph2.inMode(WAIT),
      () -> { doPhase1(1);
        next(); // signals ph1, waits on ph2
        doPhase2(1);
    }); // Task T1
    asyncPhased(ph2.inMode(SIG),ph1.inMode(WAIT),ph3.inMode(WAIT),
      () -> { doPhase1(2);
        next(); // signals ph2, waits on ph3
        doPhase2(2);
    }); // Task T2
    asyncPhased(ph3.inMode(SIG),ph2.inMode(WAIT),
      () -> { doPhase1(3);
        next(); // signals ph3, waits on ph2
        doPhase2(3);
    }); // Task T3
}); // finish
```

Computation Graph for m=3 example
(without async-finish nodes and edges)