
COMP 322: Fundamentals of Parallel Programming

Lecture 16: Point-to-Point Synchronization with Phasers Review

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

<http://comp322.rice.edu/>

COMP 322

Lecture 16

17 February 2017



Worksheet 15a: Data Driven Futures (turn page over for worksheet 15b)

Name: _____ Netid: _____

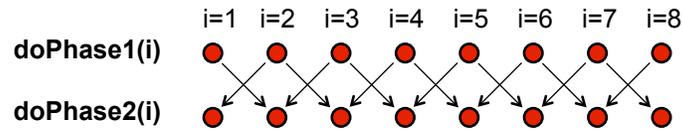
For the example below, will reordering the five async statements change the meaning of the program (assuming that the semantics of the reader/writer methods depends only on their parameters)? If so, show two orderings that exhibit different behaviors. If not, explain why not.

No, reordering the asyncs doesn't change the meaning of the problem. Regardless of the order, Task 3 will always wait on Task 1. Task 5 will always wait on Task 2. Task 4 will always wait on both Task 1 and 2.

```
1. DataDrivenFuture left = new DataDrivenFuture();
2. DataDrivenFuture right = new DataDrivenFuture();
3. finish {
4.   async await(left) leftReader(left); // Task3
5.   async await(right) rightReader(right); // Task5
6.   async await(left, right)
7.     bothReader(left, right); // Task4
8.   async left.put(leftWriter()); // Task1
9.   async right.put(rightWriter()); // Task2
10. }
```



Solution to Worksheet #15b: Left-Right Neighbor Synchronization using Phasers



Complete the phased clause below to implement the left-right neighbor synchronization shown above.

```
1. finish (() -> {
2.   final HjPhaser[] ph =
      new HjPhaser[m+2]; // array of phaser objects
3.   forseq(0, m+1, (i) -> { ph[i] = newPhaser(SIG_WAIT) });
4.   forseq(1, m, (i) -> {
5.     asyncPhased(
        ph[i-1].inMode(WAIT),
        ph[i].inMode(SIG),
        ph[i+1].inMode(WAIT), () -> {
6.       doPhase1(i);
7.       next();
8.       doPhase2(i); }); // asyncPhased
9.   }); // forseq
10.}); // finish
```

NOTE: Task-to-
phaser mappings can be
many-to-many in general. In
general, it is important to
understand the difference between
computation tasks (async's) and
synchronization objects
(phasers).



Recap: 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



Recap: 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 (ph1.inMode(<mode1>), ph2.inMode(<mode2>), ... () -> <stmt>)`
 - Spawned task is registered with `ph1` in `mode1`, `ph2` in `mode2`, ...
 - 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



Recap: Simple Example with Four Async Tasks and One Phaser

```
1. finish (() -> {
2.     ph = newPhaser(SIG_WAIT); // mode is SIG_WAIT
3.     asyncPhased(ph.inMode(SIG), () -> {
4.         // A1 (SIG mode)
5.         doA1Phase1(); next(); doA1Phase2(); });
6.     asyncPhased(ph.inMode(SIG_WAIT), () -> {
7.         // A2 (SIG_WAIT mode)
8.         doA2Phase1(); next(); doA2Phase2(); });
9.     asyncPhased(ph.inMode(HjPhaserMode.SIG_WAIT), () -> {
10.        // A3 (SIG_WAIT mode)
11.        doA3Phase1(); next(); doA3Phase2(); });
12.    asyncPhased(ph.inMode(HjPhaserMode.WAIT), () -> {
13.        // A4 (WAIT mode)
14.        doA4Phase1(); next(); doA4Phase2(); });
15. });
```



Recap: Computation Graph Schema Simple

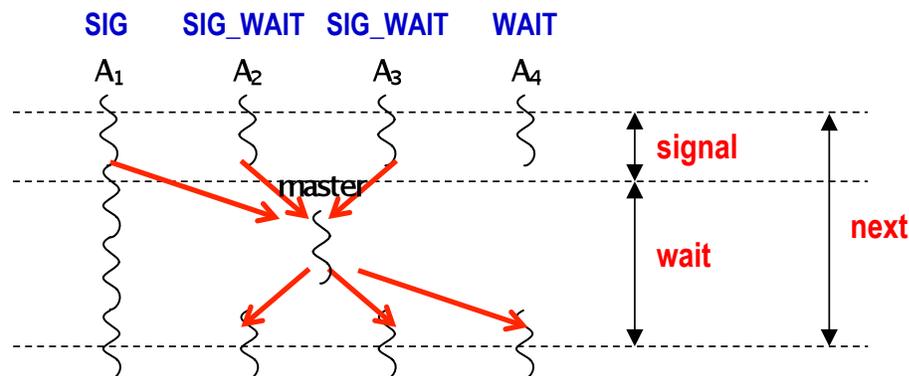
Example with Four Async Tasks and One Phaser

Semantics of **next** depends on registration mode

SIG_WAIT: **next** = signal + wait

SIG: **next** = signal

WAIT: **next** = wait

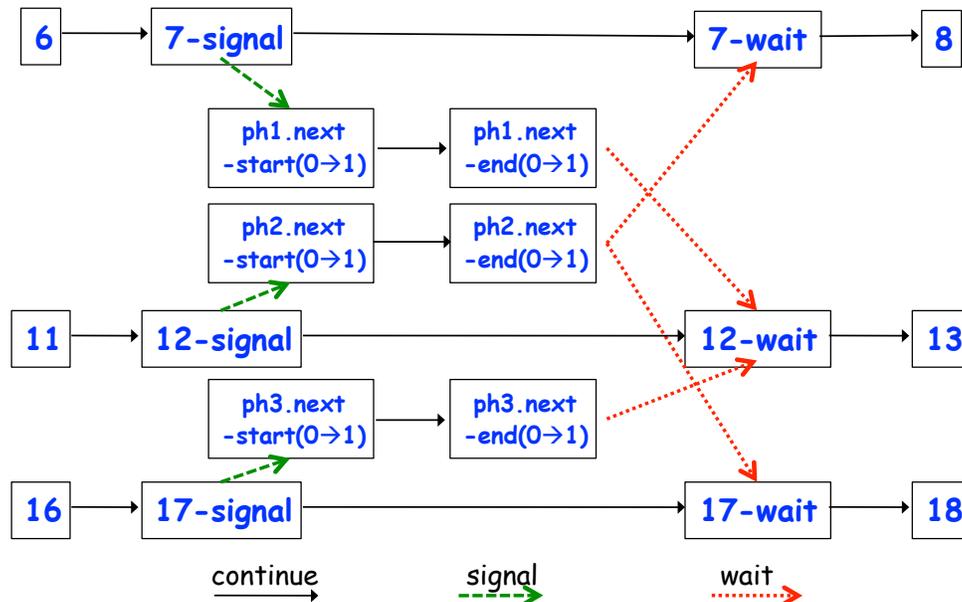


Recap: Left-Right Neighbor Synchronization (with m=3 tasks)

```
1. finish(() -> { // Task-0
2.   final HjPhaser ph1 = newPhaser(SIG_WAIT);
3.   final HjPhaser ph2 = newPhaser(SIG_WAIT);
4.   final HjPhaser ph3 = newPhaser(SIG_WAIT);
5.   asyncPhased(ph1.inMode(SIG), ph2.inMode(WAIT),
6.     () -> { doPhase1(1);
7.       next(); // signals ph1, waits on ph2
8.       doPhase2(1);
9.     }); // Task T1
10.  asyncPhased(ph2.inMode(SIG), ph1.inMode(WAIT), ph3.inMode(WAIT),
11.    () -> { doPhase1(2);
12.      next(); // signals ph2, waits on ph3
13.      doPhase2(2);
14.    }); // Task T2
15.  asyncPhased(ph3.inMode(SIG), ph2.inMode(WAIT),
16.    () -> { doPhase1(3);
17.      next(); // signals ph3, waits on ph2
18.      doPhase2(3);
19.    }); // Task T3
20.}); // finish
```



Recap: Computation Graph for m=3 example



9

COMP 322, Spring 2017 (V. Sarkar, M. Joyner)



Midterm exam (Exam 1)

- Midterm exam (Exam 1) will be held during COMP 322 lab time at 7pm on Wednesday, February 24, 2016
 - Closed-notes, closed-book, closed computer, written exam scheduled for 3 hours during 7pm — 10pm (but you can leave early if you're done early!)
 - Scope of exam is limited to Lectures 1 - 16 (all topics in Module 1 handout)
 - “Since this is a written exam and not a programming assignment, syntactic errors in program text will not be penalized (e.g., missing semicolons, incorrect spelling of keywords, etc) so long as the meaning of your solution is unambiguous.”
 - “If you believe there is any ambiguity or inconsistency in a question, you should state the ambiguity or inconsistency that you see, as well as any assumptions that you make to resolve it.”

10

COMP 322, Spring 2017 (V. Sarkar, M. Joyner)

