COMP 322: Fundamentals of Parallel Programming

Lecture 16: Point-to-point Synchronization with Phasers

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Worksheet #14: Data-Driven Tasks

For the example below, will reordering the five async statements change the meaning of the program? 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.)

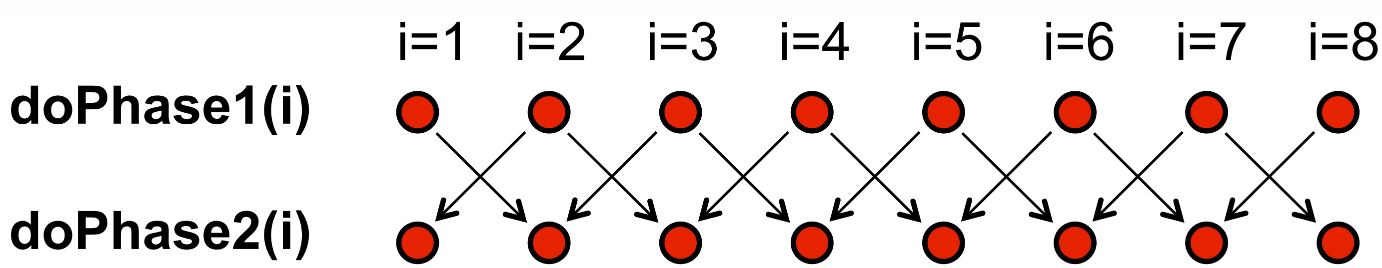
```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
  async await(left) leftReader(left); // Task3
  async await(right) rightReader(right); // Task5
  async await(left,right)
        bothReader(left,right); // Task4
  async left.put(leftWriter());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

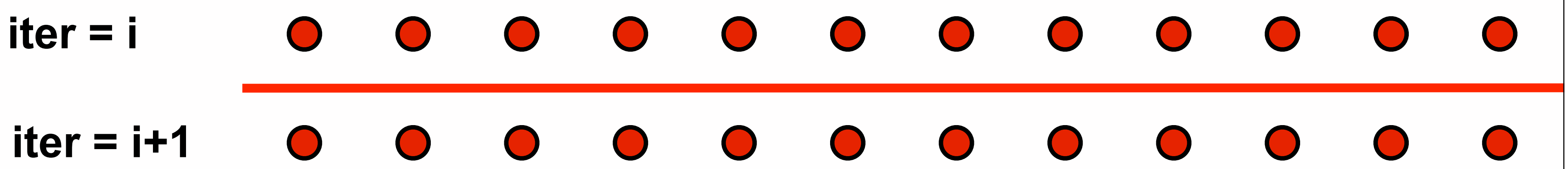
Motivation for Point-to-Point Synchronization

```
finish(() -> { // Expanded finish-forasync version of forall
  forasyncPhased(1, m, (i) -> {
    doPhase1(i);
    // Iteration i waits for i-1 and i+1 to complete Phase 1
    doPhase2(i);
  });
});
```

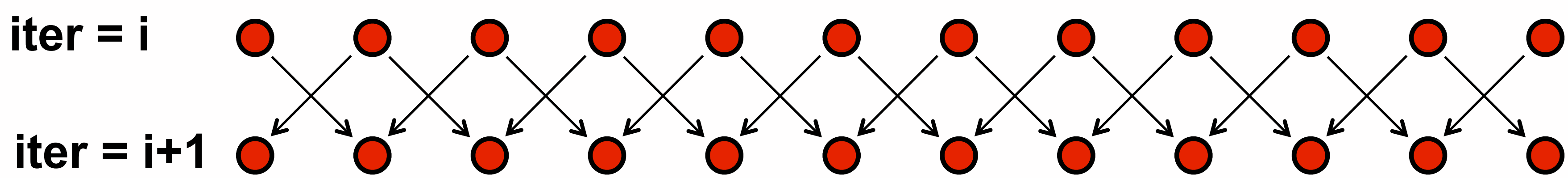
- **Need synchronization where iteration i only waits for iterations i−1 and i+1 to complete their work in doPhase1() before it starts doPhase2(i)**
  - **Less constrained than a barrier --- only waits for two preceding iterations**
  - **More general than async await --- waiting occurs in middle of task**



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



Barrier synchronization



Point-to-point synchronization

(Left-right neighbor synchronization)

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
  - Deadlock freedom --- a next operation will not lead to a situation where all active tasks are blocked indefinitely
    - but use of explicit doWait() can lead to deadlock

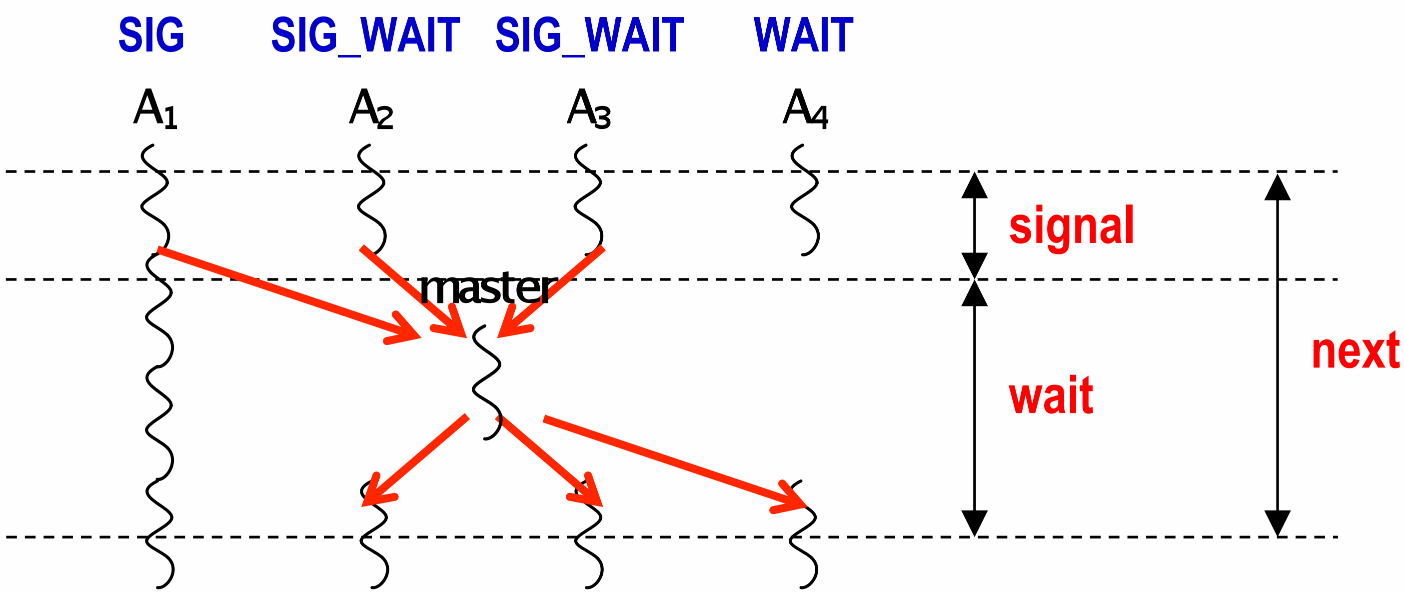
Simple Example with Four Async Tasks and One Phaser

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

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**



A master thread (worker) gathers all signals and broadcasts a barrier completion

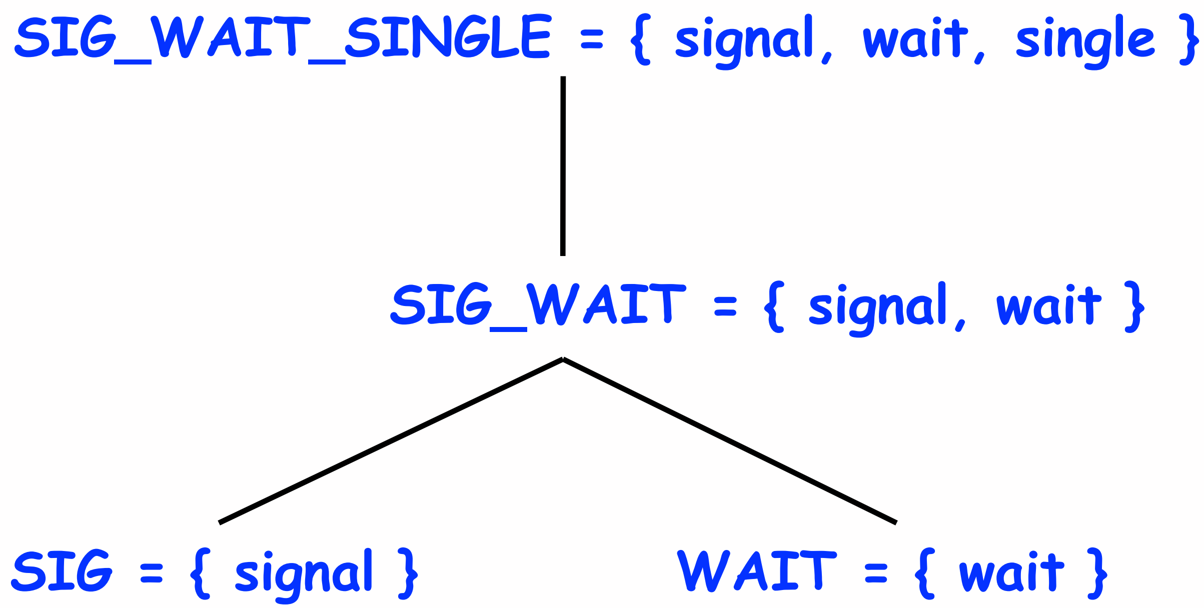
Summary of Phaser Construct

- 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

So, what is a phaser and how does it work?

- **A phaser is a synchronization *object* --- you can allocate as many phasers as you choose, and also build arrays/collections of phasers**
- **The task that allocates a phaser is automatically *registered* on the phaser in the mode specified in the constructor (SIG_WAIT is the default mode)**
- **A task can be registered on *multiple phasers in different modes*, specified in its "async phased" clause or due to its phaser allocations**
- **A "next" operation performs *all signal operations followed by all wait operations*, according to the task's phaser registrations**
  - **Ordering of signal-wait avoids deadlock**
  - **Degenerates gracefully when wait set or signal set is empty**
- **A registration on phaser ph in mode m can only be included in "async phased" if the parent was also registered on ph with mode m (*capability rule*)**
- **Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF) for the allocation i.e., if phaser ph is allocated in finish scope F, then the task executing F *drops any registration that it has on ph when reaching the end-finish point for F***

Capability Hierarchy



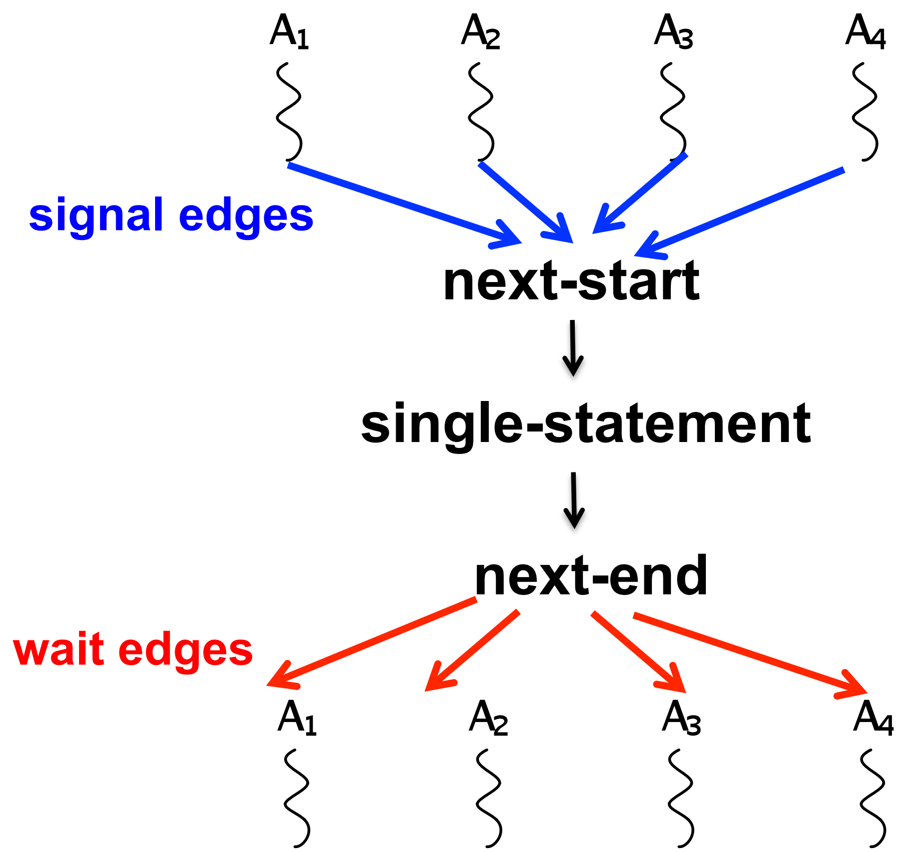
- A task can be registered in one of four modes with respect to a phaser: SIG_WAIT_SINGLE, SIG_WAIT, SIG, or WAIT. The mode defines the set of capabilities — signal, wait, single — that the task has with respect to the phaser. The subset relationship defines a natural hierarchy of the registration modes.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

Next-with-Single Statement (for SIG_WAIT_SINGLE registration mode)

next <single-stmt> is a barrier in which single-stmt is performed exactly once after all tasks have completed the previous phase and before any task begins its next phase.

NOTE: single statement are not currently implemented in HJ-lib

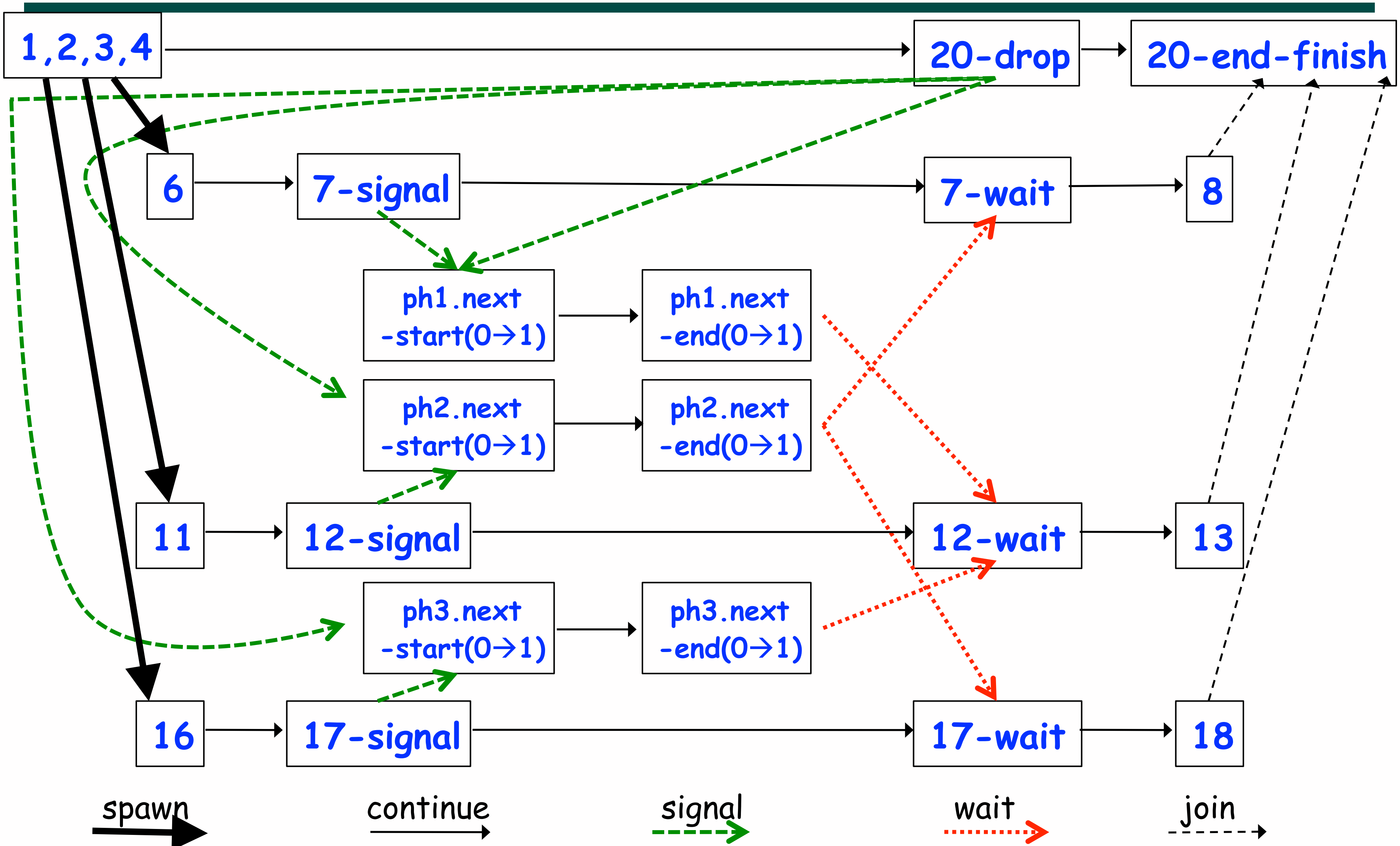
Modeling next-with-single in the Computation Graph



Left-Right Neighbor Synchronization Example for m=3

```
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), () -> { // Task-1
        doPhase1(1);
        next(); // signals ph1, waits on ph2
        doPhase2(1);
    });
    asyncPhased(ph2.inMode(SIG), ph3.inMode(WAIT), () -> { // Task-2
        doPhase1(2);
        next(); // signals ph2, waits on ph3
        doPhase2(2);
    });
    asyncPhased(ph3.inMode(SIG), ph2.inMode(WAIT), () -> { // Task-3
        doPhase1(3);
        next(); // signals ph3, waits on ph2
        doPhase2(3);
    });
}); // finish
```

Computation Graph for m=3 example



Adding Phaser Operations to the Computation Graph

CG node = step

Step boundaries are induced by continuation points

- async: source of a spawn edge
- end-finish: destination of join edges
- future.get(): destination of a join edge
- signal, drop: source of signal edges
- wait: destination of wait edges
- next: modeled as signal + wait

CG also includes an unbounded set of pairs of phase transition nodes for each phaser ph allocated during program execution

- ph.next-start(i→i+1) and ph.next-end(i→i+1)

Adding Phaser Operations to the Computation Graph (contd)

CG edges enforce ordering constraints among the nodes

- continue edges capture sequencing of steps within a task
- spawn edges connect parent tasks to child async tasks
- join edges connect descendant tasks to their Immediately Enclosing Finish (IEF) operations and to get() operations for future tasks
- signal edges connect each signal or drop operation to the corresponding phase transition node, ph.next-start(i→i+1)
- wait edges connect each phase transition node, ph.next-end(i→i+1) to corresponding wait or next operations
- single edges connect each phase transition node, ph.next-start(i→i+1) to the start of a single statement instance, and from the end of that single statement to the phase transition node, ph.next-end(i→i+1)

forall barrier is just an implicit phaser

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

is equivalent to

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

The world according to COMP 322 before Barriers and Phasers

- Most of the parallel constructs that we learned during Lectures 1-11 focused on task creation and termination
  - async creates a task
    - forasync creates a set of tasks specified by an iteration region
  - finish waits for a set of tasks to terminate
    - forall (like “finish forasync”) creates and waits for a set of tasks specified by an iteration region
  - future get() waits for a specific task to terminate
  - asyncAwait() waits for a set of DataDrivenFuture values before starting
- Motivation for barriers and phasers
  - Deterministic directed synchronization within tasks
  - Separate from synchronization associated with task creation and termination

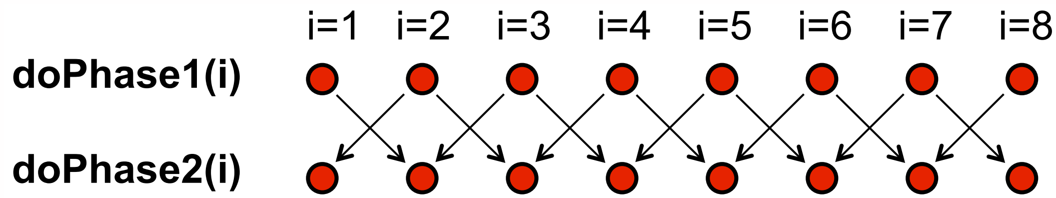
The world according to COMP 322 after Barriers and Phasers

- SPMD model: express iterative synchronization using phasers
  - Implicit phaser in a forall supports barriers as “next” statements
    - Matching of next statements occurs dynamically during program execution
    - Termination signals “dropping” of phaser registration
  - Explicit phasers
    - Can be allocated and transmitted from parent to child tasks
    - Phaser lifetime is restricted to its IEF (Immediately Enclosing FInish) scope of its creation
    - Four registration modes -- SIG, WAIT, SIG_WAIT, SIG_WAIT_SINGLE
    - signal statement can be used to support “fuzzy” barriers
    - bounded phasers can limit how far ahead producer gets of consumers
- Difference between phasers and data-driven tasks (DDTs)
  - DDTs enforce a single point-to-point synchronization at the start of a task
  - Phasers enforce multiple point-to-point synchronizations within a task

Worksheet #16: Left-Right Neighbor Synchronization using Phasers

Name: ______________________

Netid: ______________________



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

```
finish (() -> {
  final HjPhaser[] ph =
      new HjPhaser[m+2]; // array of phaser objects
  forseq(0, m+1, (i) -> { ph[i] = newPhaser(SIG_WAIT) });
  forseq(1, m, (i) -> {
    asyncPhased(
       ph[i-1].inMode(......),
       ph[i].inMode(......),
       ph[i+1].inMode(......), () {
     doPhase1(i);
     next();
     doPhase2(i); }); // asyncPhased
  }); // forseq
}); // finish
```

COMP 322: Fundamentals of Parallel Programming (Spring 2014)
Instructor: Vivek Sarkar
Worksheet 15: due by start of next class

Name: ___________________________ Netid: _______________

Honor Code Policy: You are free to discuss all aspects of in-class worksheets with your other classmates, the teaching assistants and the professor during the class. You can work in a group in a class, and write down the solution that you obtained as a group. If you work on the worksheet outside of class, then it must be an individual effort. If you use any material from external sources, you must provide proper attribution.

HJ-Lib Module 1 APIs

Consider the following HJ-Lib APIs:

1. **async**(**HjRunnable** runnable)
2. **asyncAwait**(**HjFuture**<? extends **Object**> f1, **HjRunnable** runnable)
3. **asyncSeq**(boolean sequentialize, **HjRunnable** runnable)
4. **doWork**(long n)
5. **finish**(**FinishAccumulator** f1, **HjRunnable** runnable)
6. **forall**(**Iterable**<T> iterable, **HjProcedure**<T> body)
7. **forasyncChunked**(int startInc, int endInc, **HjProcedure**<**Integer**> body)
8. **future**(**HjCallable**<V> callable)
9. **futureAwait**(**HjFuture**<? extends **Object**> f1, **HjCallable**<V> callable)
10. **next**()

For each of the following functionalities, enter the number of the API above that matches the functionality:

Functionality	**API number**