COMP 322: Fundamentals of Parallel Programming

Lecture 22: Point to Point Synchronization with Phasers

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Parallelizing loops in Matrix Multiplication example using forall

```
// Parallel version using forall
forall(0, n-1, 0, n-1, (i, j) -> {
    c[i][j] = 0;
 });
forall(0, n-1, 0, n-1, (i, j) -> {
   forseq(0, n-1, (k) -> {
      c[i][j] += a[i][k] * b[k][j];
   });
});
// Print first element of output matrix
println(c[0][0]);
```

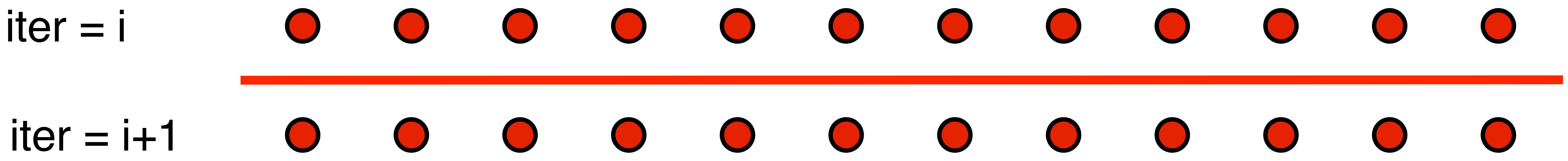
$$c[i,j] = \sum_{0 \le k < n} a[i,k] * b[k,j]$$

Parallelizing loops in Matrix Multiplication example using forall

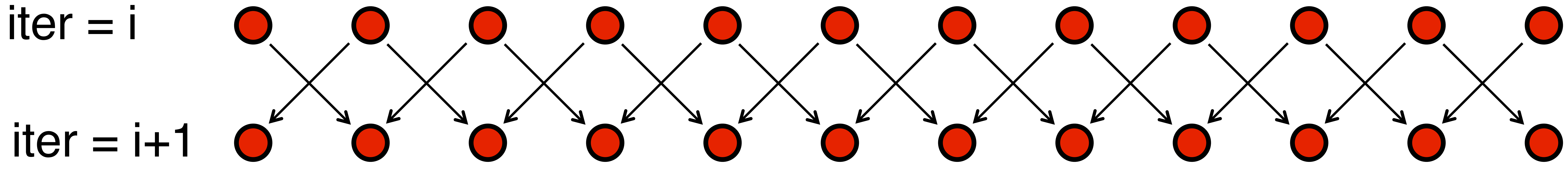
```
// Parallel version using forall
 forallPhased(0, n-1, 0, n-1, (i, j) -> {
     c[i][j] = 0;
     next();
     forseq(0, n-1, (k) -> {
       c[i][j] += a[i][k] * b[k][j];
     });
 });
// Print first element of output matrix
println(c[0][0]);
```

$$c[i,j] = \sum_{0 \le k < n} a[i,k] * b[k,j]$$

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



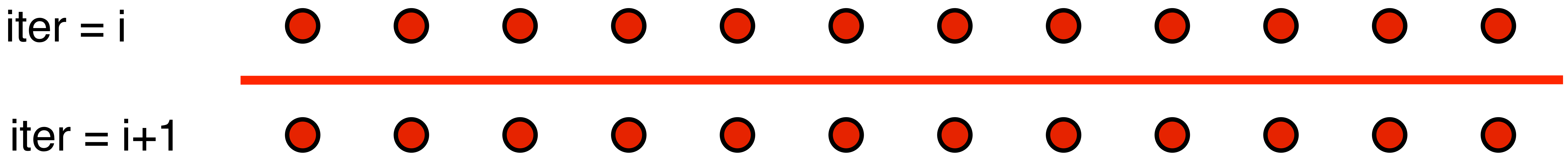
Barrier synchronization



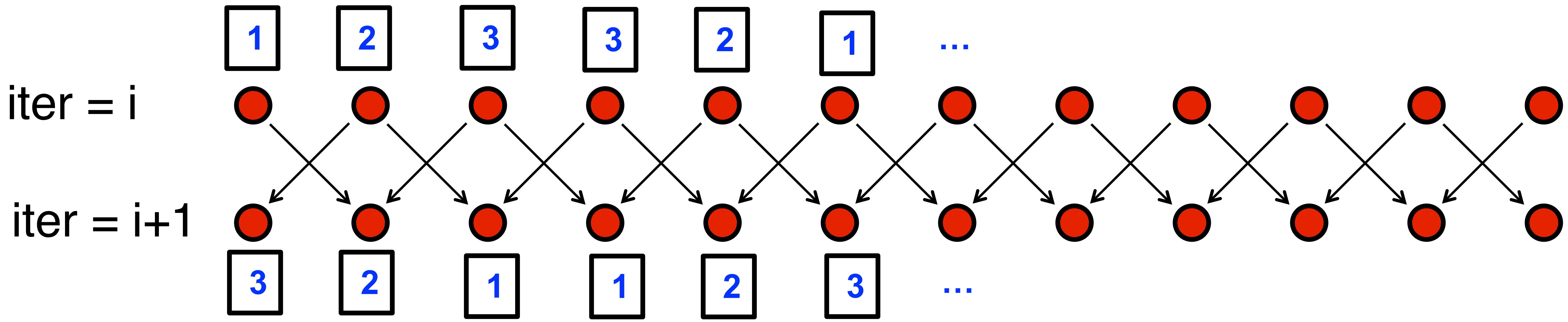
Point-to-point synchronization

Question: Can the point-to-point computation graph result in a smaller CPL than the barrier computation graph?

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



Barrier synchronization



Point-to-point synchronization

Question: Can the point-to-point computation graph result in a smaller CPL than the barrier computation graph?

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(); });
});
```

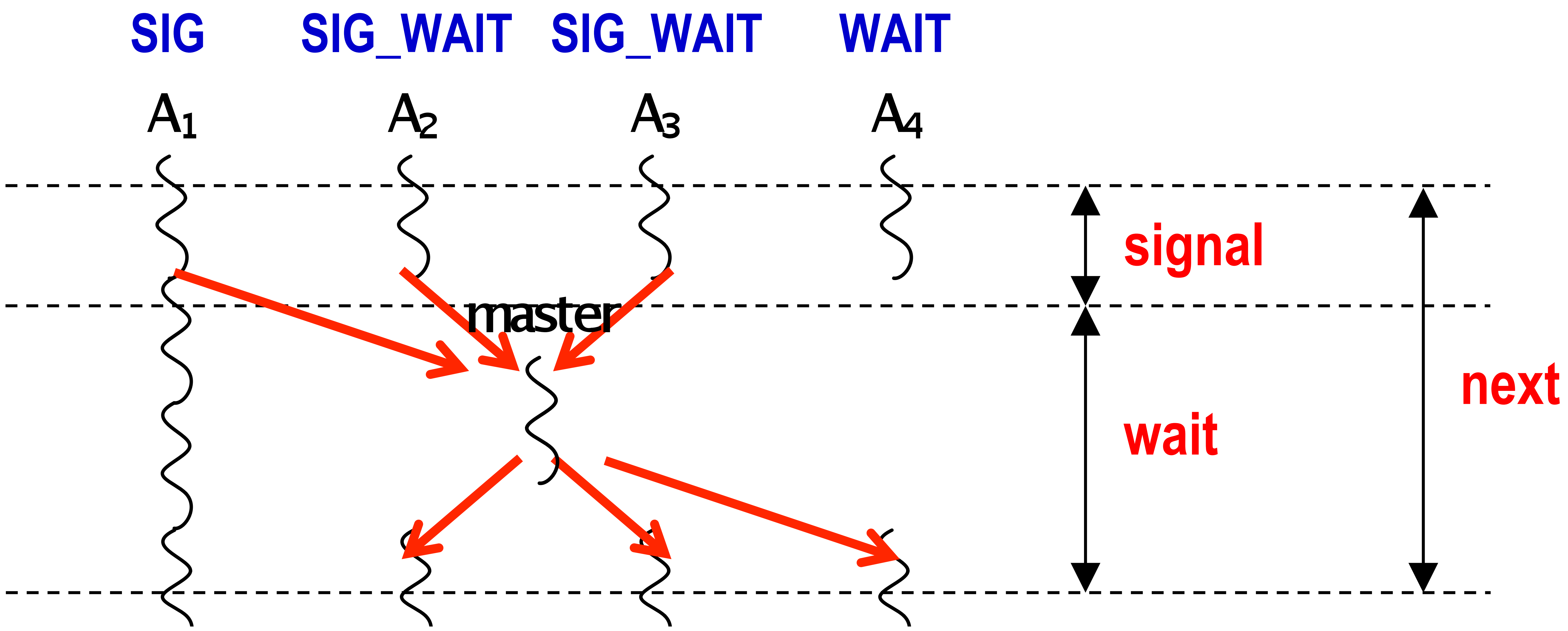
Computation Graph Schema Simple Example with Four Async Tasks and One Phaser

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
- 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 }

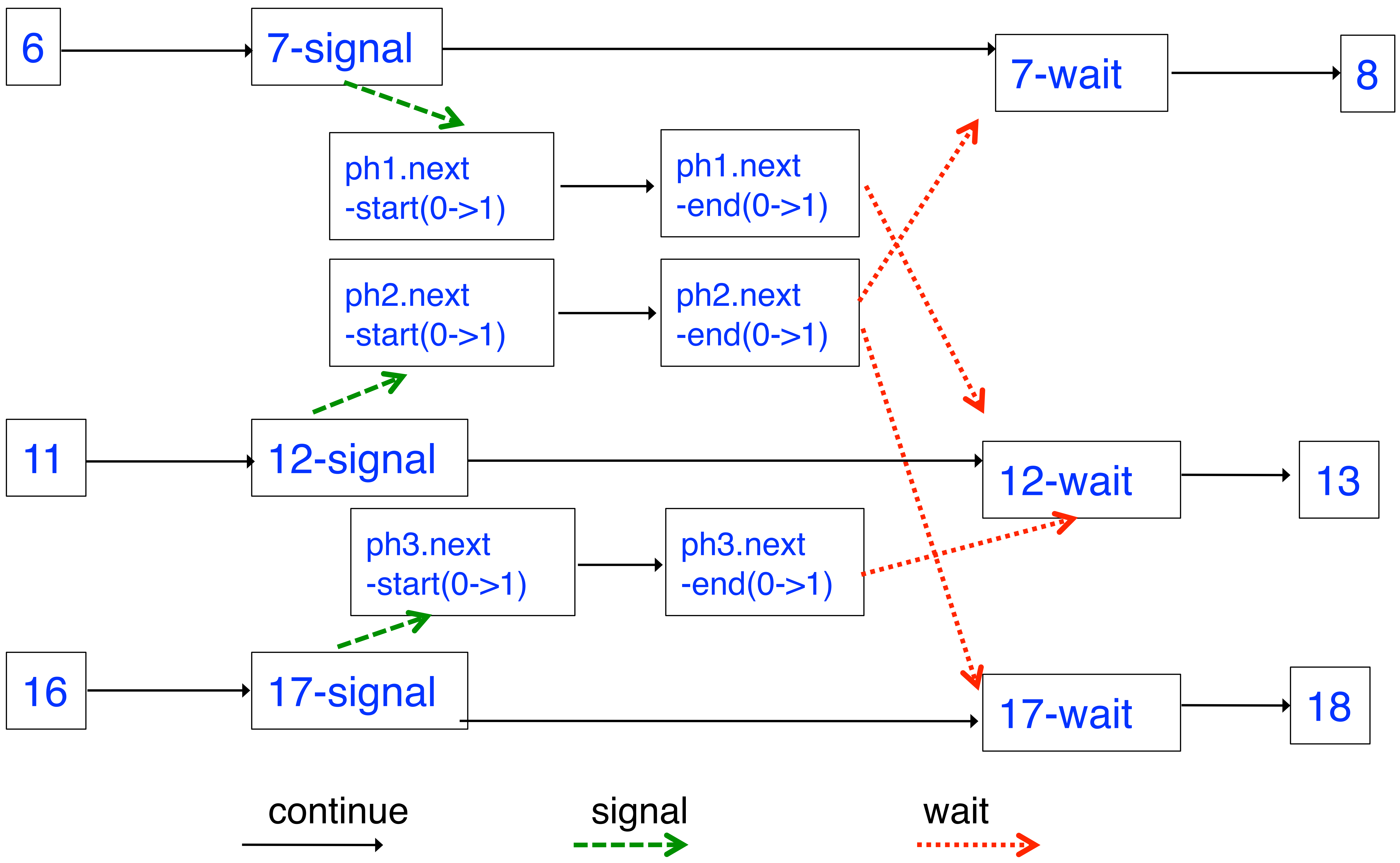
SIG = { signal }

WAIT = { wait }

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 ph1, 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)



forallPhased 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
});
```