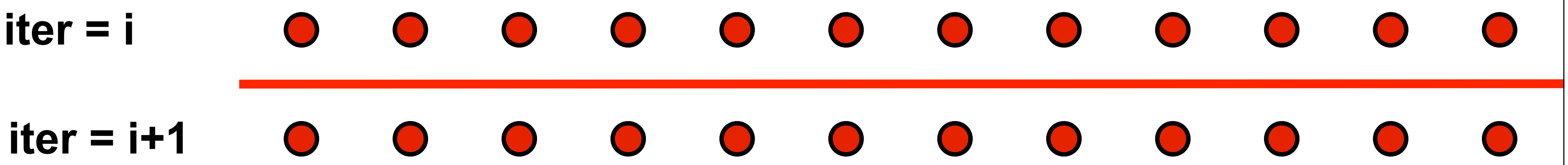
COMP 322: Fundamentals of Parallel Programming

Lecture 14: Unification of Barrier and Point-to-point Synchronization with Phasers

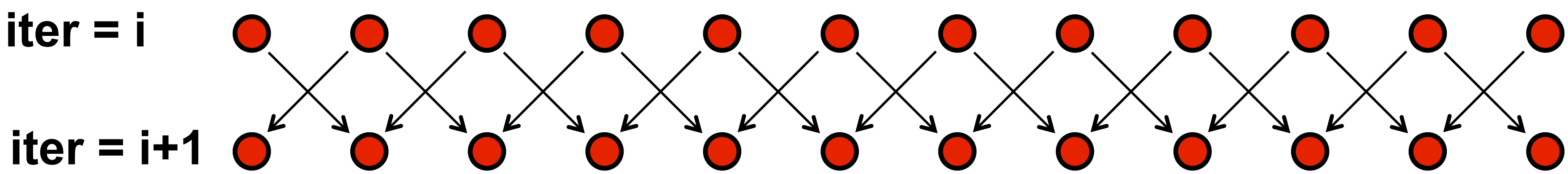
Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

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

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



Barrier synchronization

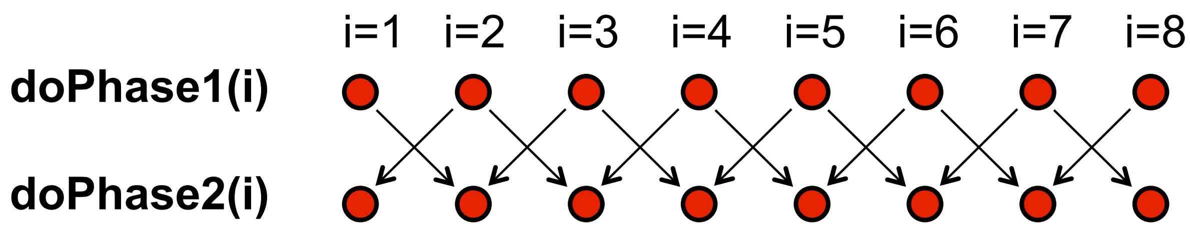


Point-to-point synchronization

Left-Right Neighbor Synchronization Example

```
finish { // Expanded finish-for-async version of forall
  for (point[i] : [1:m])
    async {
      doPhase1(i);
 // Iteration i waits for i-1 and i+1 to complete Phase 1
      doPhase2(i);
    } // async
} // finish
```

- 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)



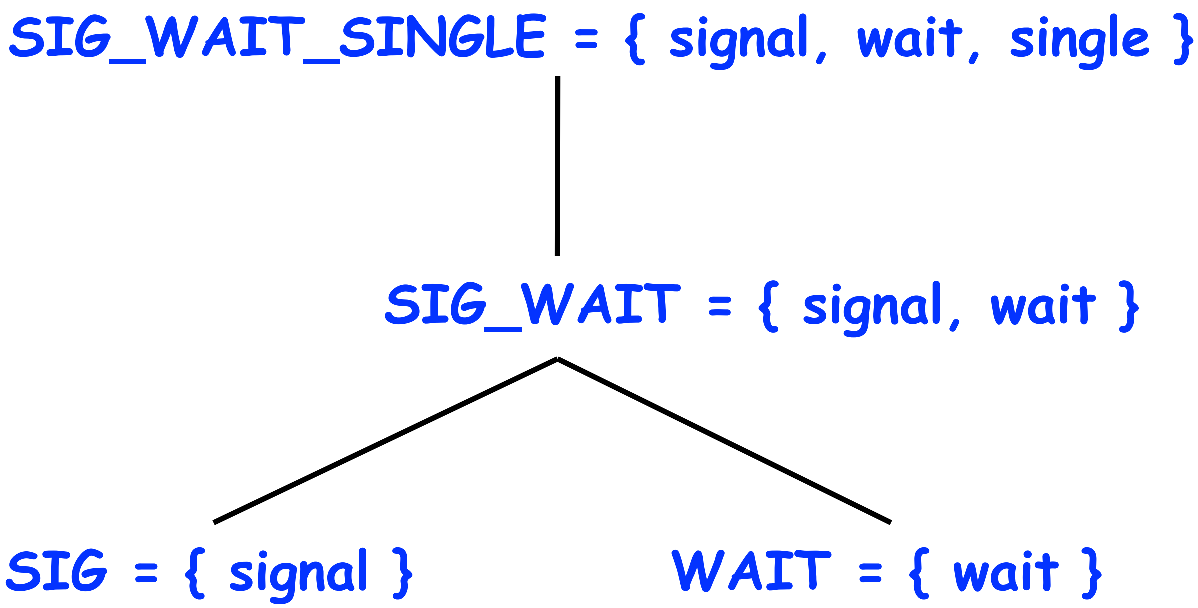
Phasers: a unified construct for barrier and point-to-point synchronization

- Previous examples motivated the need for point-to-point synchronization
- HJ phasers unify barriers with point-to-point synchronization
- A limited version of phasers was also added to the Java 7 java.util.concurrent.Phaser library (with acknowledgment to Rice)
- Phaser properties
  - Barrier and point-to-point synchronization
  - Supports dynamic parallelism i.e., the ability for tasks to drop phaser registrations on termination, and for new tasks to add new phaser registrations.
  - Deadlock freedom
  - Support for phaser accumulators (reductions that can be performed with phasers)

Summary of Phaser Construct

- Phaser allocation
  - phaser ph = new phaser(mode);
    - Phaser ph is allocated with registration mode
    - Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)
- Registration Modes
  - phaserMode.SIG, phaserMode.WAIT, phaserMode.SIG_WAIT, phaserMode.SIG_WAIT_SINGLE
- Phaser registration
  - async phased (ph_1<$mode_1$>, ph_2<$mode_2$>, …) <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
    - async phased <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

Capability Hierarchy



- At any point in time, 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.

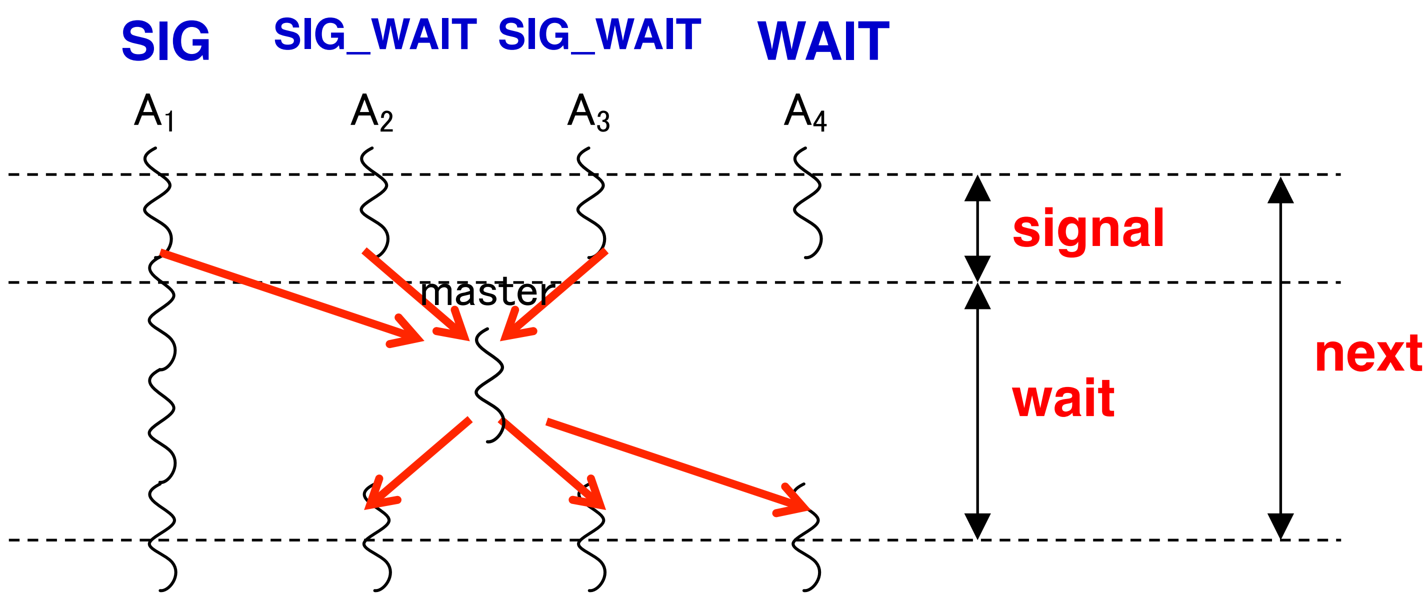
Simple Example with Four Async Tasks and One Phaser

```
finish {
  ph = new phaser(); // Default mode is SIG_WAIT
  async phased(ph<phaserMode.SIG>){ //A1 (SIG mode)
    doA1Phase1(); next;
    doA1Phase2(); }
  async phased { //A2 (default SIG_WAIT mode from parent)
    doA2Phase1(); next;
    doA2Phase2(); }
  async phased { //A3 (default SIG_WAIT mode from parent)
    doA3Phase1(); next;
    doA3Phase2(); }
  async phased(ph<phaserMode.WAIT>){ //A4 (WAIT mode)
    doA4Phase1(); next; doA4Phase2(); }
}
```

Simple Example with Four Async Tasks and One Phaser (contd)

Semantics of next depends on registration mode

- **SIG_WAIT: next = signal + wait**
- **SIG: next = signal (Don't wait for any task)**
- **WAIT: next = wait (Don't disturb any task)**



A master task receives all signals and broadcasts a barrier completion

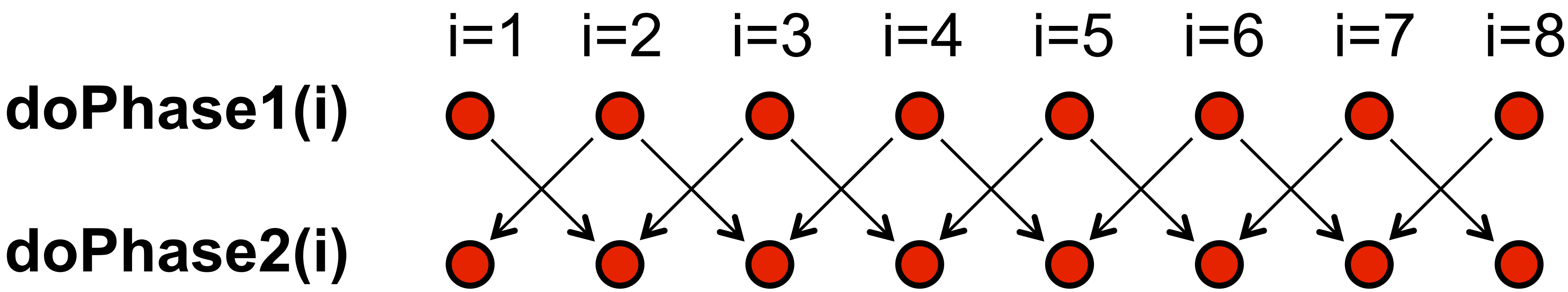
forall barrier is just an implicit phaser

```
forall (point[i,j] : [iLo:iHi,jLo:jHi])
  <body>
```

is equivalent to

```
finish {
  // Implicit phaser
  phaser ph = new phaser(phaserMode.SIG_WAIT);
  for(point[i,j] : [iLo:iHi,jLo:jHi])
    async phased(phaserMode.SIG_WAIT)
      <body> // next statements refer to ph
}
```

Left-Right Neighbor Synchronization Example

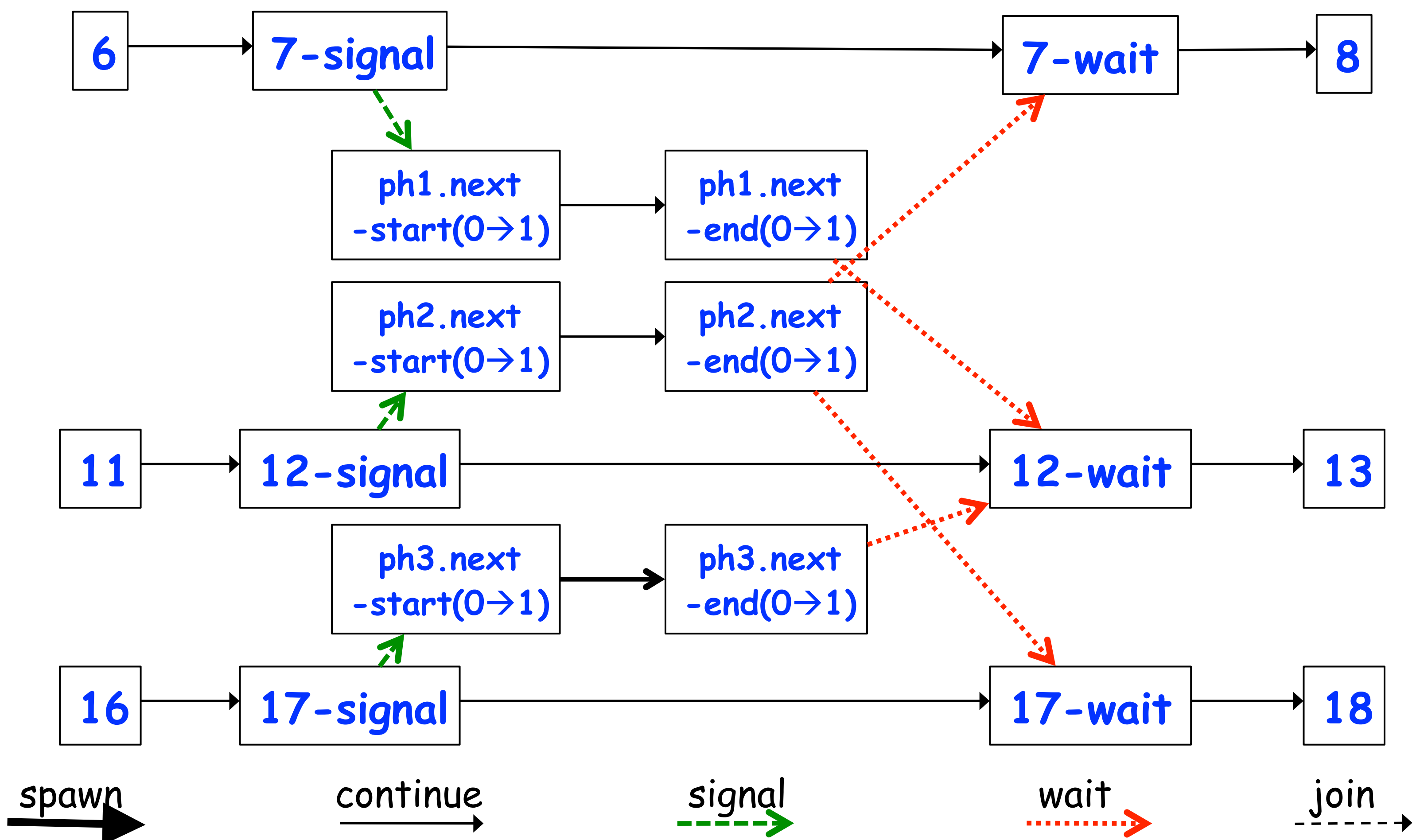



```
finish {
  phaser[] ph = new phaser[m+2];
  for(point [i]:[0:m+1]) ph[i] = new phaser();
  for(point [i] : [1:m])
   async phased(ph[i]<SIG>, ph[i-1]<WAIT>, ph[i+1]<WAIT>) {
     doPhase1(i);
     next; // Signal ph[i] & wait on ph[i-1], ph[i+1]
     doPhase2(i);
   }
}
```

Left-Right Neighbor Synchronization Example for m=3

```
finish {
  phaser ph1 = new phaser(); // Default mode is SIG_WAIT
  phaser ph2 = new phaser(); // Default mode is SIG_WAIT
  phaser ph3 = new phaser(); // Default mode is SIG_WAIT
  async phased(ph1<SIG>, ph2<WAIT>) { // i = 1
    doPhase1(1);
    next; // Signals ph1, and waits on ph2
    doPhase2(1);
  }
  async phased(ph2<SIG>, ph1<WAIT>, ph3<WAIT>) { // i = 2
    doPhase1(2);
    next; // Signals ph2, and waits on ph1 and ph3
    doPhase2(2);
  }
  async phased(ph3<SIG>, ph2<WAIT>) { // i = 3
    doPhase1(3);
    next; // Signals ph3, and waits on ph2
    doPhase2(3);
  }
}
```

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



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

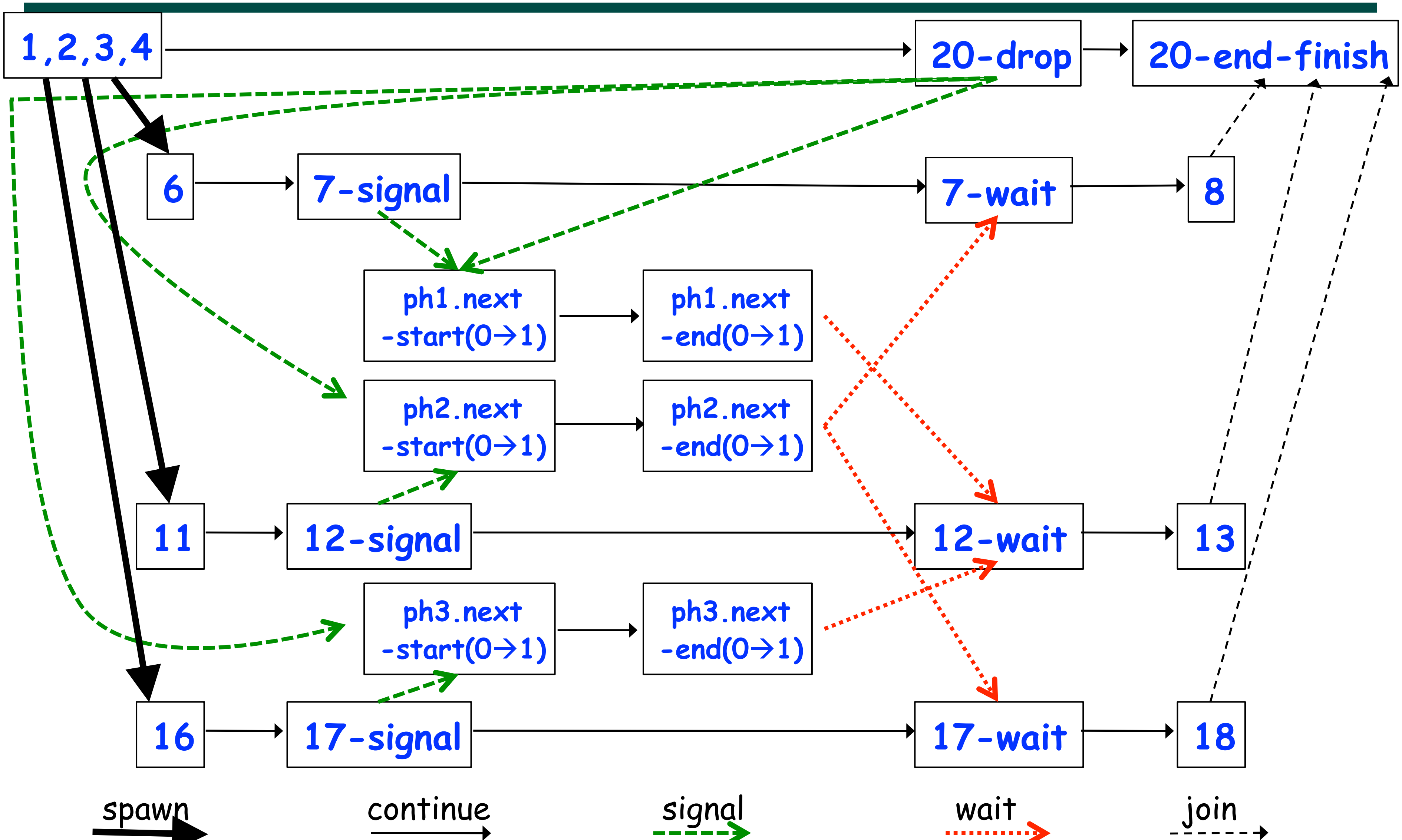
- ph.next-start(i→i+1) and ph.next-end(i→i+1)

Adding Phaser Operations to the Computation Graph (contd)

CG edges enforce ordering constraints among the nodes

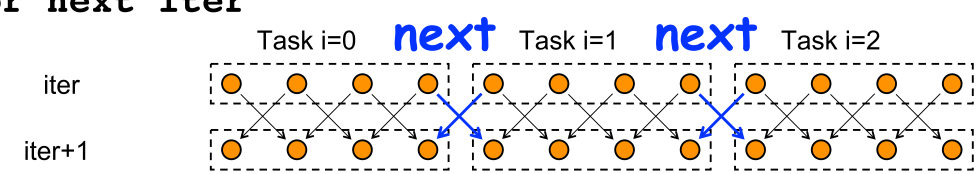
- continue edges capture sequencing of steps within a task
- spawn edges connect parent tasks to child async tasks
- join edges connect descendant tasks to their Immediately Enclosing Finish (IEF) operations and to get() operations for future tasks
- signal edges connect each signal or drop operation 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)

Full Computation Graph for m=3 example



One-Dimensional Iterative Averaging with Point-to-Point Synchronization (compare with slide 9, Lecture 13)

```
double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
gVal[n+1] = 1; gNew[n+1] = 1;
int Cj = Runtime.getNumOfWorkers();
finish {
  phaser ph = new phaser[Cj+2];
  for(point [i]:[0:Cj+1]) ph[i] = new phaser();
  for(point [jj]:[0:Cj-1])
  async phased(ph[jj+1]<SIG>,ph[jj]<WAIT>, ph[jj+2]<WAIT>) {
    double[] myVal = gVal; double[] myNew = gNew; // Local copy of pointers
    for (point [iter] : [0:numIters-1]) {
      for (point [j]:getChunk([1:n],[Cj],[jj])) // Iterate within chunk
         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
      next; // Point-to-point synchronization
      // Swap myVal and myNew
      double[] temp=myVal; myVal=myNew; myNew=temp;
      // myNew becomes input array for next iter
    } // for
} // finish
```



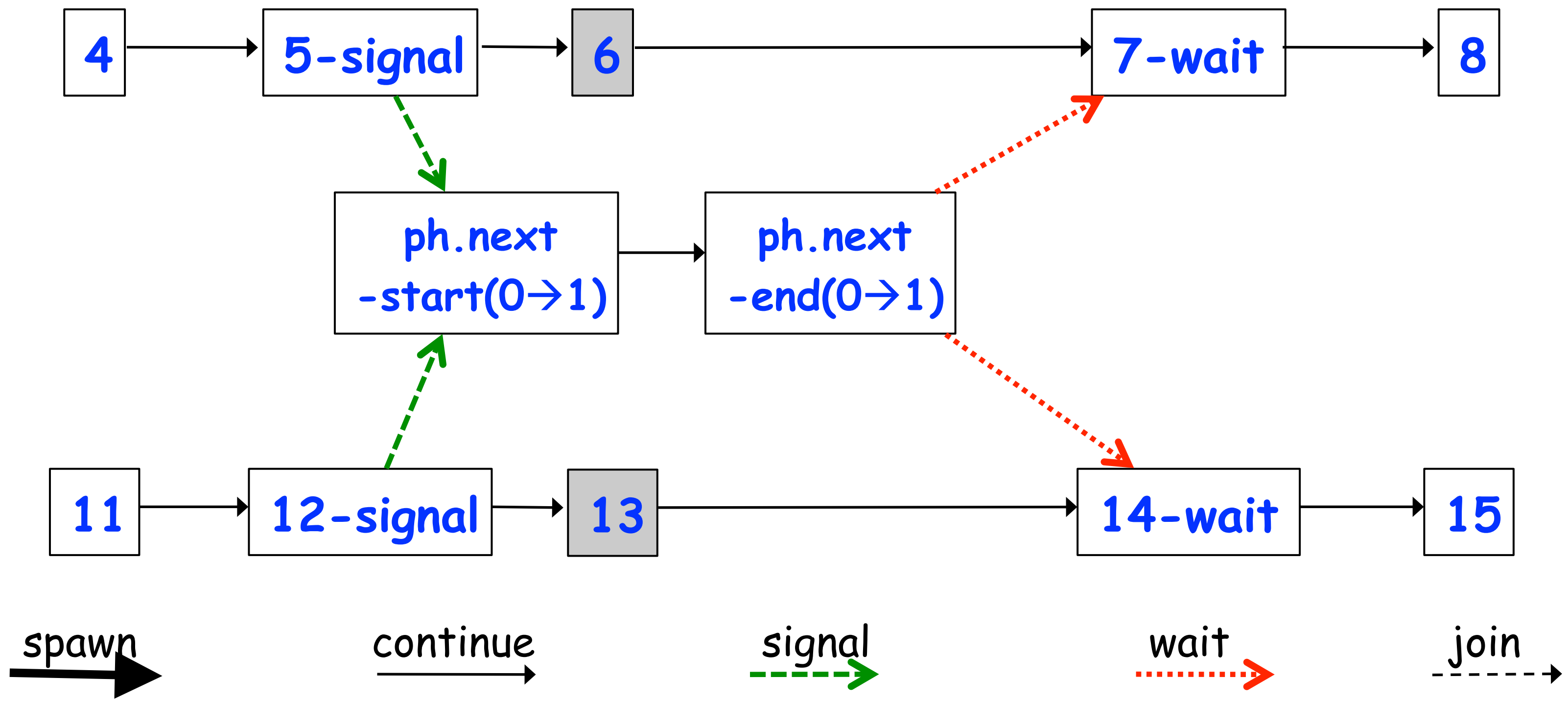
Signal statement

- When a task T performs a signal operation, it notifies all the phasers it is registered on that it has completed all the work expected by other tasks in the current phase ("shared" work).
  - Since signal is a non-blocking operation, an early execution of signal cannot create a deadlock.
- Later, when T performs a next operation, the next degenerates to a wait since a signal has already been performed in the current phase.
- The execution of "local work" between signal and next is performed during phase transition
  - Referred to as a "split-phase barrier" or "fuzzy barrier"

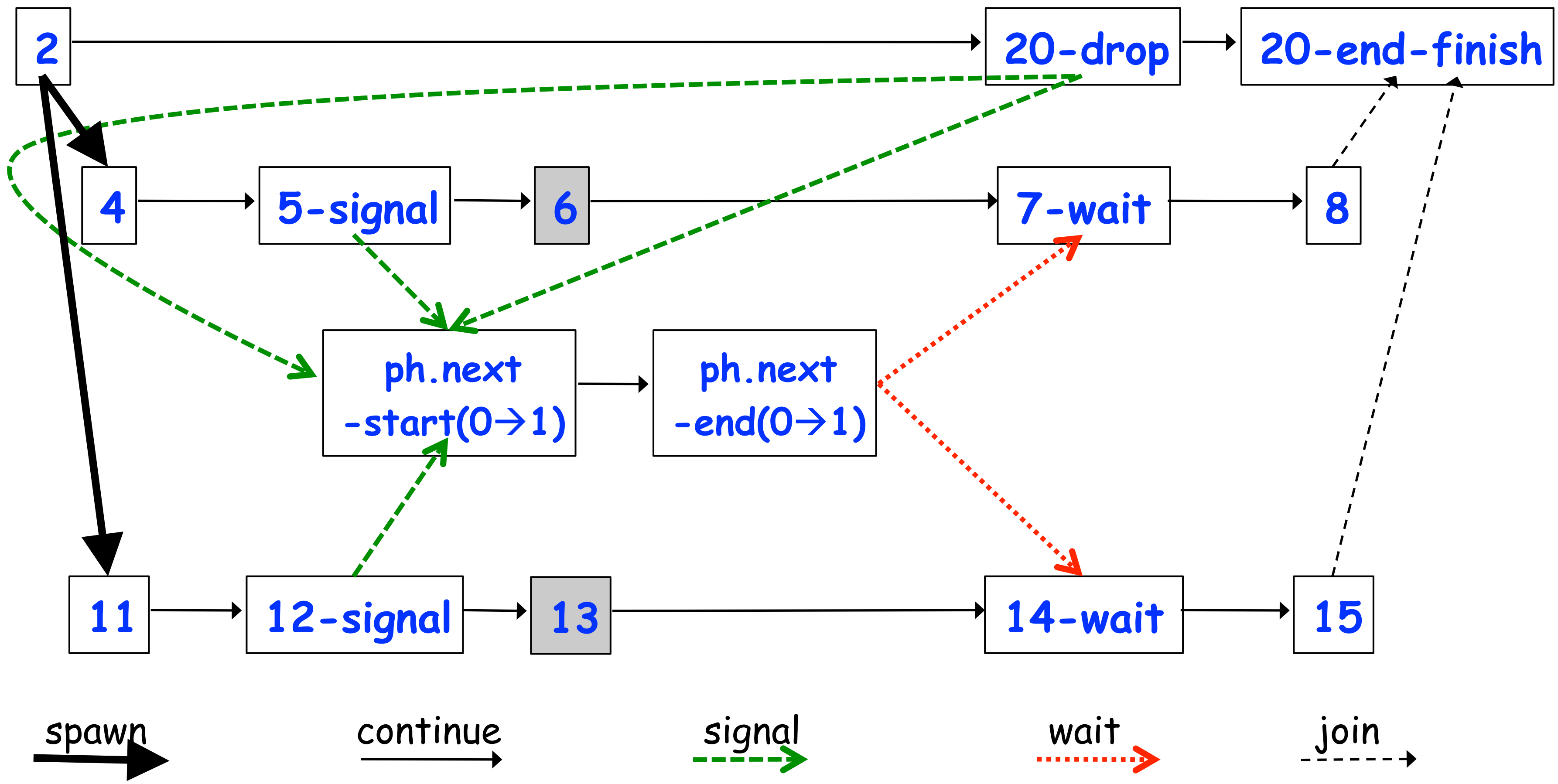
Example of Split-Phase Barrier

```
finish {
  phaser ph = new phaser(phaserMode.SIG_WAIT);
  async phased { // Task T1
    a = ... ;     // Shared work in phase 0
    signal;       // Signal completion of a's computation
    b = ... ;     // Local work in phase 0
    next;         // Barrier -- wait for T2 to compute x
    b = f(b,x);   // Use x computed by T2 in phase 0
  }
  async phased { // Task T2
    x = ... ;     // Shared work in phase 0
    signal;       // Signal completion of x's computation
    y = ... ;     // Local work in phase 0
    next;         // Barrier -- wait for T1 to compute a
    y = f(y,a);   // Use a computed by T1 in phase 0
  }
} // finish
```

Computation Graph for Split-Phase Barrier Example (without async and finish nodes and edges)



Full Computation Graph for Split-Phase Barrier Example



Optimized One-Dimensional Iterative Averaging with Split-Phase Point-to-Point Synchronization

```
double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
gVal[n+1] = 1; gNew[n+1] = 1;
int Cj = Runtime.getNumOfWorkers();
finish {
  phaser ph = new phaser[Cj+2];
  for(point [i]:[0:Cj+1]) ph[i] = new phaser();
  for(point [jj]:[0:Cj-1])
  async phased(ph[jj+1]<SIG>,ph[jj]<WAIT>, ph[jj+2]<WAIT>) {
    double[] myVal = gVal; double[] myNew = gNew; // Local copy of pointers
    for (point [iter] : [0:numIters-1]) {
      region r = getChunk([1:n],[Cj],[jj]); int lo = r.rank(0).low(); int hi = r.rank(0).high();
      myNew[lo] = (myVal[lo-1] + myVal[lo+1])/2.0; myNew[hi] = (myVal[hi-1] + myVal[hi+1])/2.0;
      signal; // signal ph[jj+1]
      for (point [j]: [lo+1:hi-1]) // Iterate within chunk
         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
      next; // wait on ph[jj] and ph[jj+2]
      // Swap myVal and myNew
      double[] temp=myVal; myVal=myNew; myNew=temp;
      // myNew becomes input array for next iter
    } // for
} // finish
```

Announcements

- Homework 3 due on Wednesday, Feb 22nd
  - Performance results for parts 2 and 3 of assignment must be obtained on Sugar (see Section 4)
  - Start early --- you should complete the ideal parallel version this week
- Exam 1 will be held in the lecture on Friday, Feb 24th
  - Closed book 50-minute exam
  - Scope of exam includes lectures up to Monday, Feb 20th
  - Feb 22nd lecture will be a midterm review before exam
  - Contact me ASAP if you have an extenuating circumstance and need to take the midterm at an alternate time