

COMP 322: Fundamentals of Parallel Programming

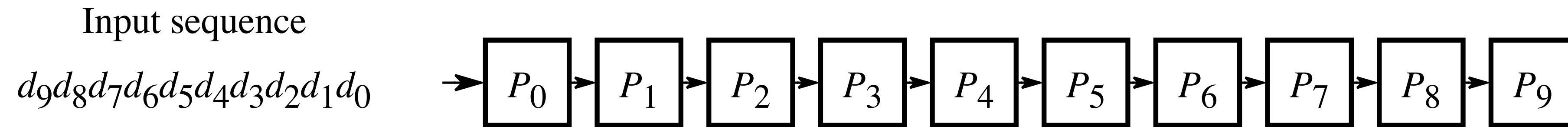
Lecture 23: Pipeline Parallelism, Signal Statement, Fuzzy Barriers

Mack Joyner
mjoyner@rice.edu

<http://comp322.rice.edu>



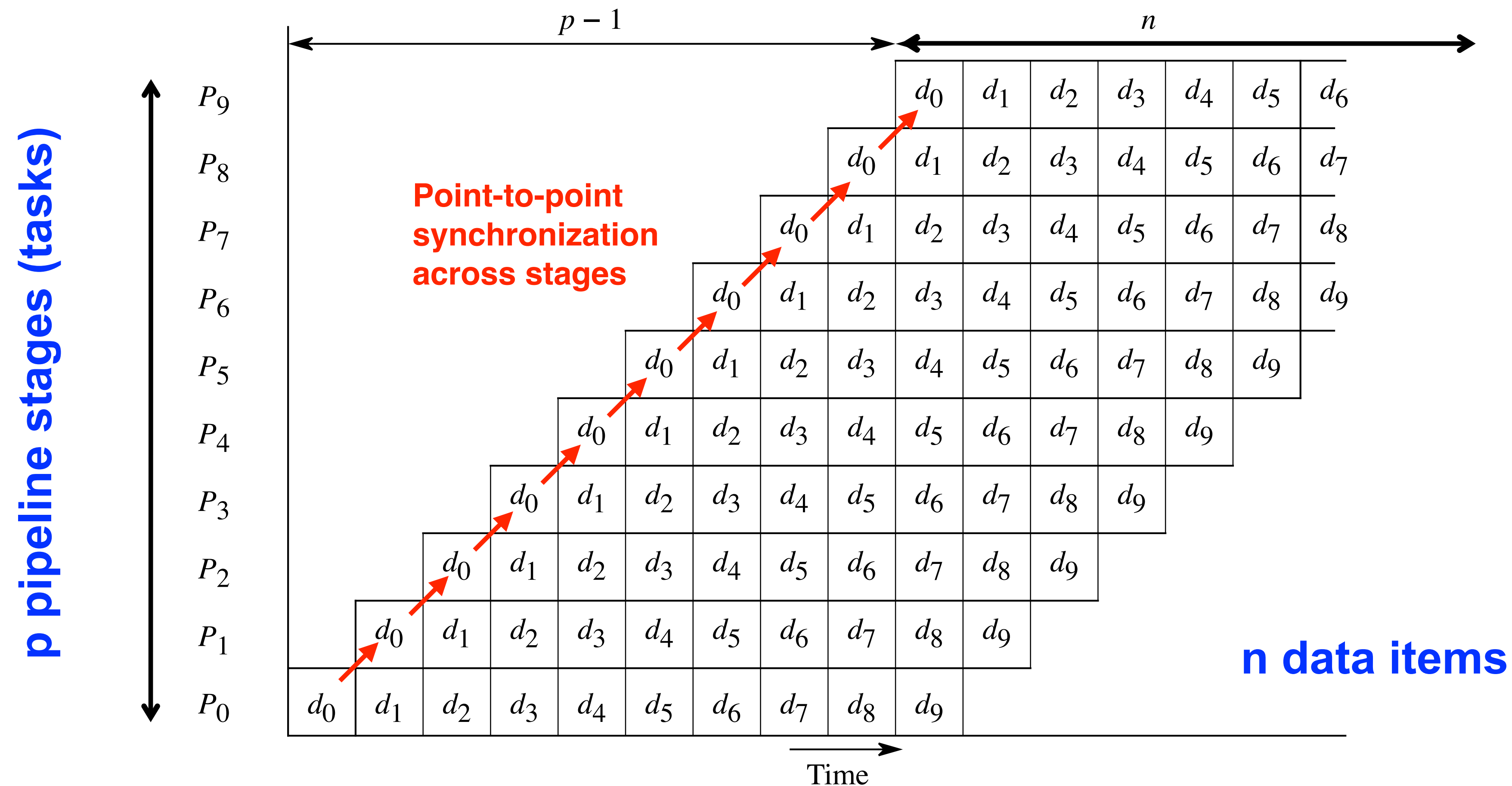
General structure of a One-Dimensional Pipeline



- Assuming that the inputs d_0, d_1, \dots arrive sequentially, pipeline parallelism can be exploited by enabling task (stage) P_i to work on item d_{k-i} when task (stage) P_0 is working on item d_k .



Timing Diagram for One-Dimensional Pipeline



- Horizontal axis shows progress of time from left to right, and vertical axis shows which data item is being processed by which pipeline stage at a given time.



Using a phaser to implement pipeline parallelism (unbounded buffer)

```
1. asyncPhased(ph.inMode(...), () -> {
2.     for (int i = 0; i < rounds; i++) {
3.         ...
4.         buffer.insert(...);
5.         ...
6.     }
7. });
8.
9. asyncPhased(ph.inMode(...), () -> {
10.    for (int i = 0; i < rounds; i++) {
11.        ...
12.        buffer.remove(...)
13.        ...
14.    }
15. });
```



Using a phaser to implement pipeline parallelism (unbounded buffer)

```
1. asyncPhased(ph.inMode(SIG), () -> {
2.     for (int i = 0; i < rounds; i++) {
3.         buffer.insert(...);
4.         // producer can go ahead as they are in SIG mode
5.         next();
6.     }
7. });
8.
9. asyncPhased(ph.inMode(WAIT), () -> {
10.    for (int i = 0; i < rounds; i++) {
11.        next();
12.        buffer.remove(...);
13.    }
14. });
```



Using a phaser to implement pipeline parallelism (bounded buffer with size = 2)

```
1. asyncPhased(ph.inMode(...), () -> {
2.     for (int i = 0; i < rounds; i++) {
3.         ...
4.         buffer.insert(...);
5.         ...
6.     }
7. });
8.
9. asyncPhased(ph.inMode(...), () -> {
10.    for (int i = 0; i < rounds; i++) {
11.        ...
12.        buffer.remove(...)
13.        ...
14.    }
15. });
```



Using a phaser to implement pipeline parallelism (bounded buffer with size = 2)

```
1. asyncPhased(ph.inMode(SIG_WAIT), () -> {
2.     for (int i = 0; i < rounds; i++) {
3.         buffer.insert(...);
4.         // producer can go ahead as they are in SIG mode
5.         next();
6.     }
7. });
8.
9. asyncPhased(ph.inMode(SIG_WAIT), () -> {
10.    for (int i = 0; i < rounds; i++) {
11.        next();
12.        buffer.remove(...);
13.    }
14. });
```



Using a phaser to implement pipeline parallelism? (bounded buffer with size = 1)

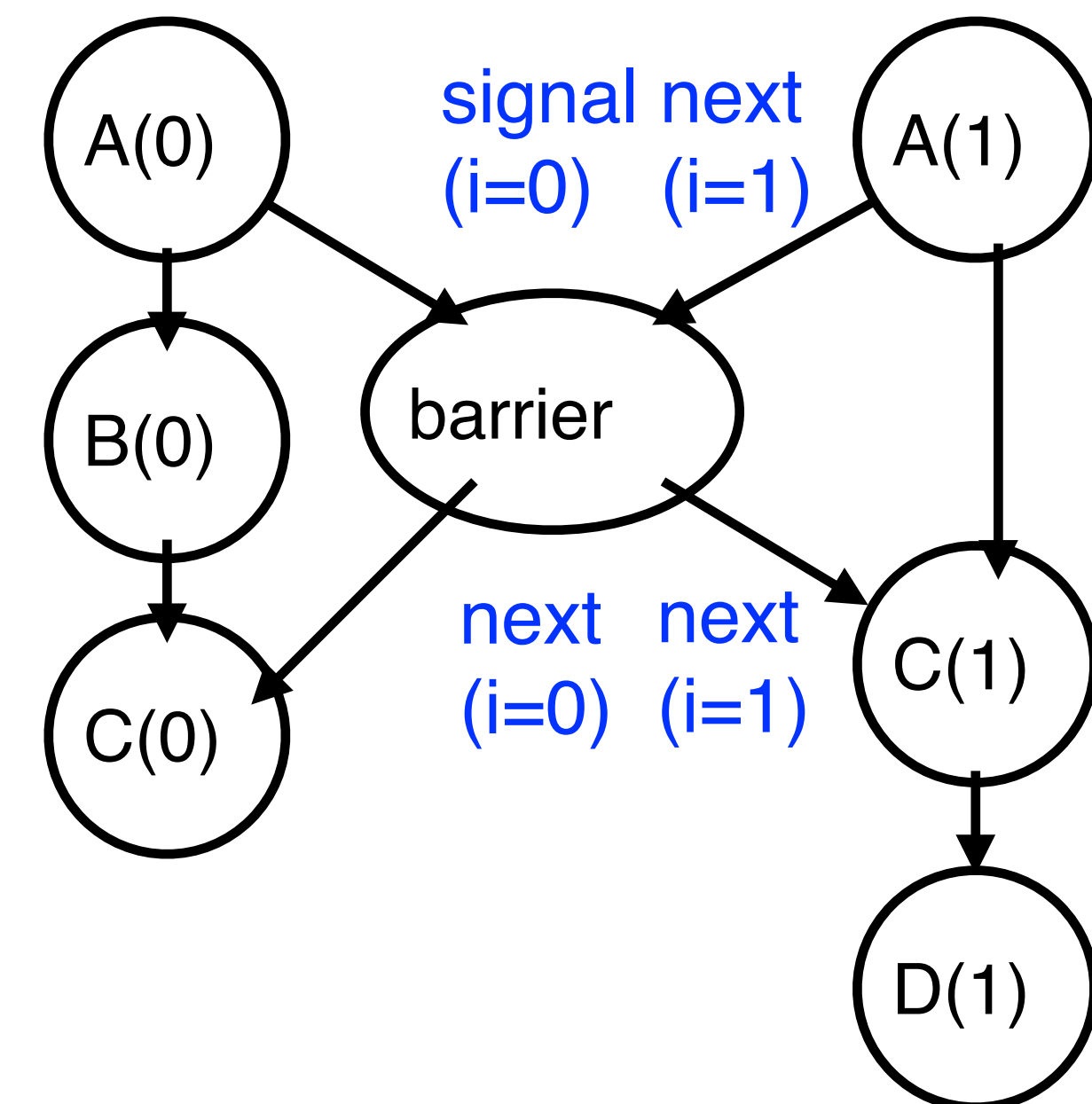
```
1. asyncPhased(ph.inMode(...), () -> {
2.     for (int i = 0; i < rounds; i++) {
3.         ...
4.         buffer.insert(...);
5.         ...
6.     }
7. });
8.
9. asyncPhased(ph.inMode(...), () -> {
10.    for (int i = 0; i < rounds; i++) {
11.        ...
12.        buffer.remove(...)
13.        ...
14.    }
15. });
```



Signal statement & Fuzzy barriers

- 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 (“shared” work) in the current phase.
- 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 overlapped with the phase transition (referred to as a “split-phase barrier” or “fuzzy barrier”)

```
1. forallPhased(point[i] : [0:1]) {  
2.   A(i); // Phase 0  
3.   if (i==0) { signal; B(i); }  
4.   next; // Barrier  
5.   C(i); // Phase 1  
6.   if (i==1) { D(i); }  
7. }
```

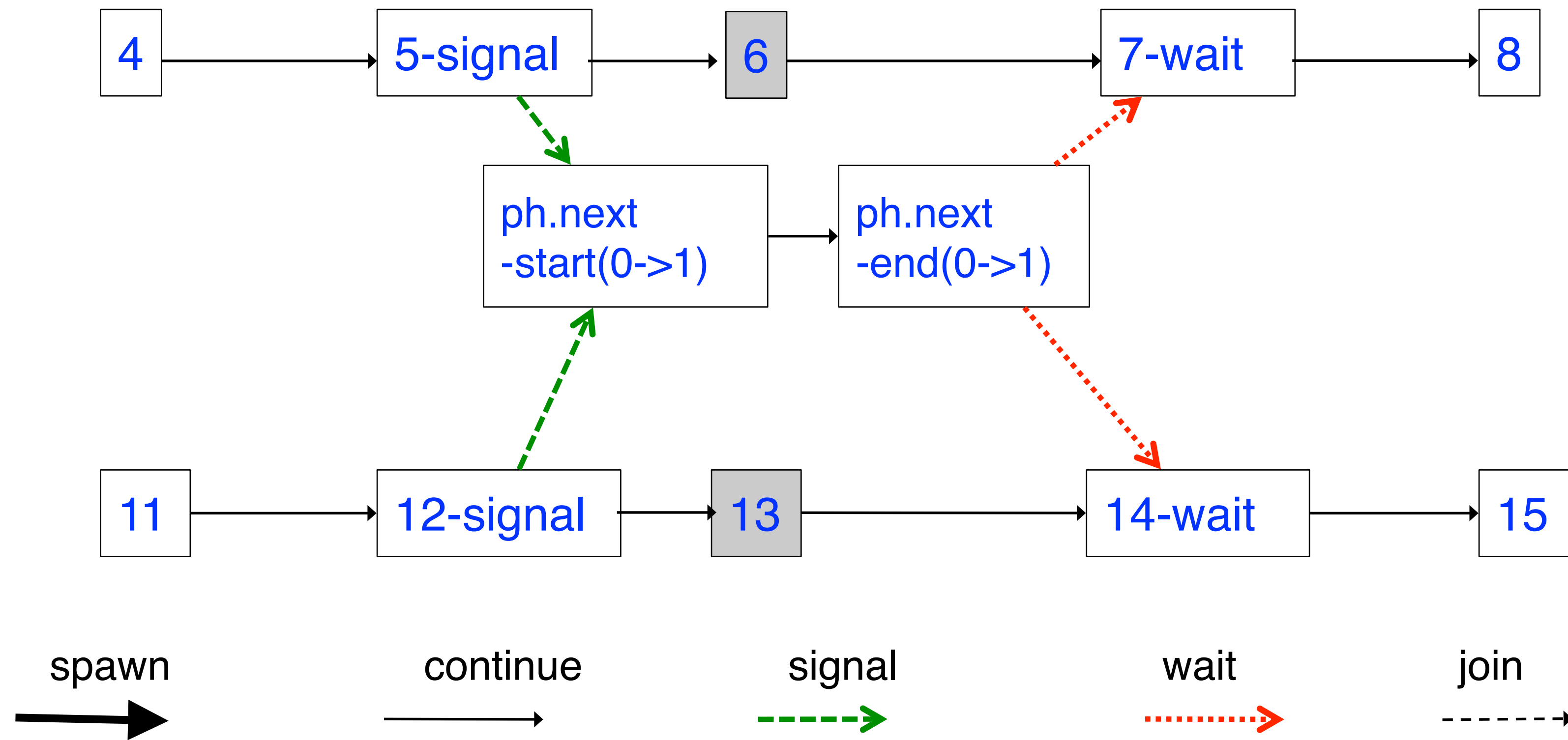


Another Example of a Split-Phase Barrier using the Signal Statement

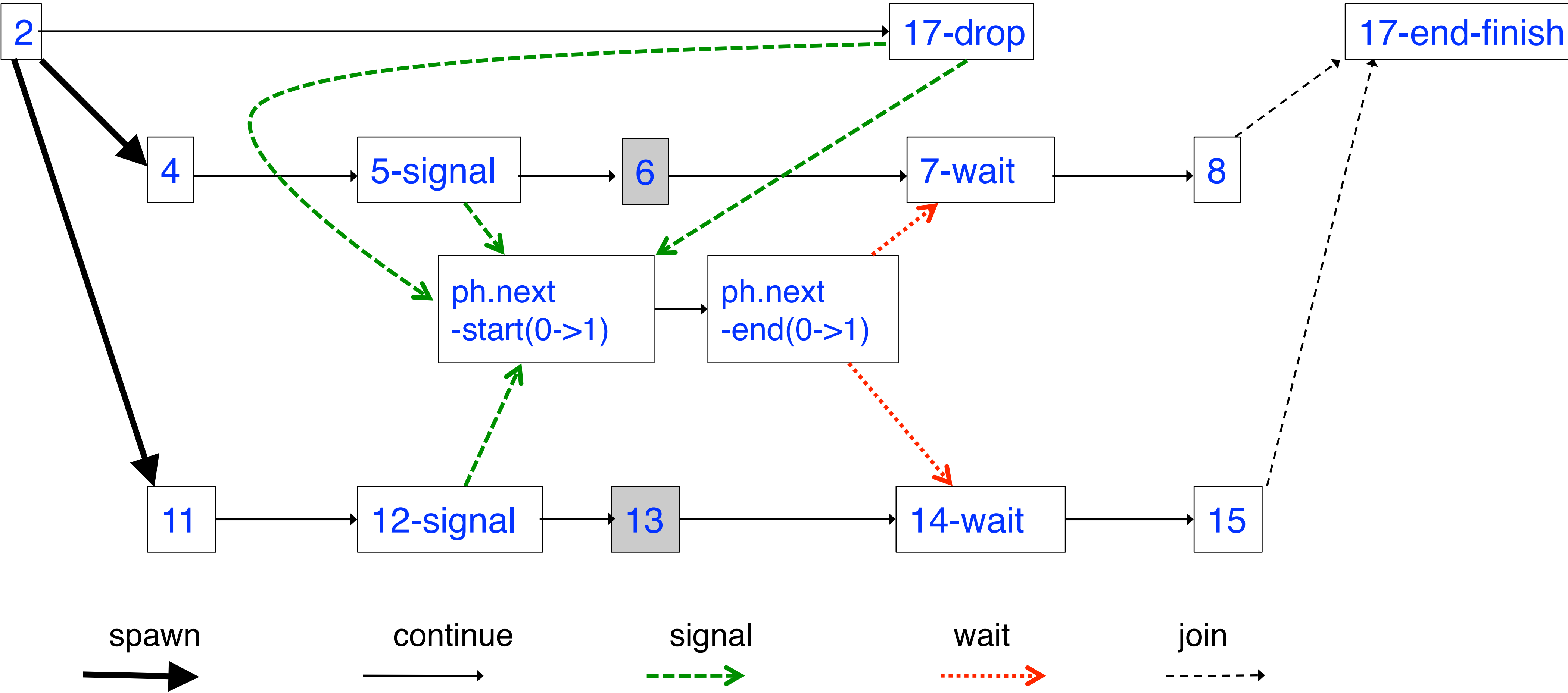
```
1. finish() -> {
2.   final HjPhaser ph = newPhaser(SIG_WAIT);
3.   asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T1
4.     a = ... ; // Shared work in phase 0
5.     signal(); // Signal completion of a's computation
6.     b = ... ; // Local work in phase 0
7.     next(); // Barrier -- wait for T2 to compute x
8.     b = f(b,x); // Use x computed by T2 in phase 0
9.   });
10.  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T2
11.    x = ... ; // Shared work in phase 0
12.    signal(); // Signal completion of x's computation
13.    y = ... ; // Local work in phase 0
14.    next(); // Barrier -- wait for T1 to compute a
15.    y = f(y,a); // Use a computed by T1 in phase 0
16.  });
17.}); // finish
```



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



Full Computation Graph for Split-Phase Barrier Example



The world according to COMP 322 before Barriers and Phasers

- All the other parallel constructs that we learned 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

