COMP 322: Fundamentals of Parallel Programming

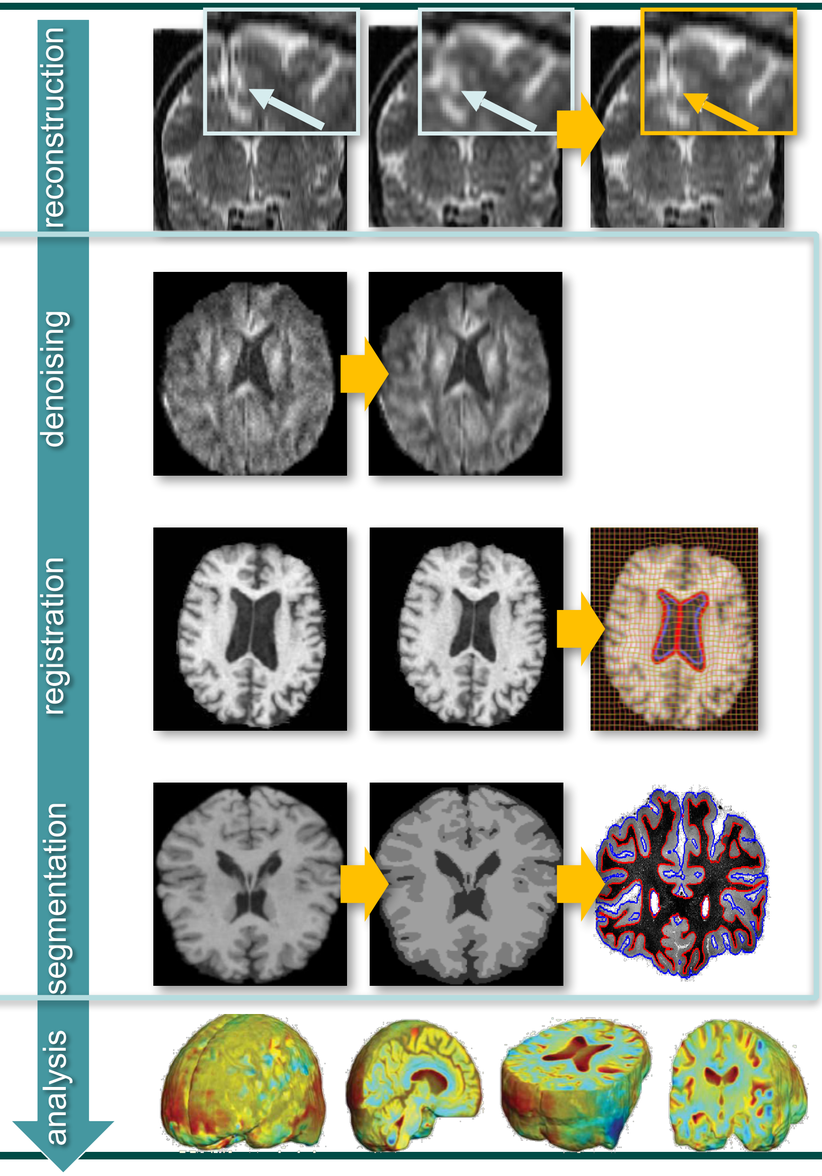
Lecture 17: Pipeline Parallelism, Signal Statement, Fuzzy Barriers

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Medical imaging pipeline

- New reconstruction methods
  - decrease radiation exposure (CT)
  - number of samples (MR)
- 3D/4D image analysis pipeline
  - Denoising
  - Registration
  - Segmentation
- Analysis
  - Real-time quantitative cancer assessment applications
- Potential:
  - order-of-magnitude performance improvement
  - power efficiency improvements
  - real-time clinical applications and simulations using patient imaging data

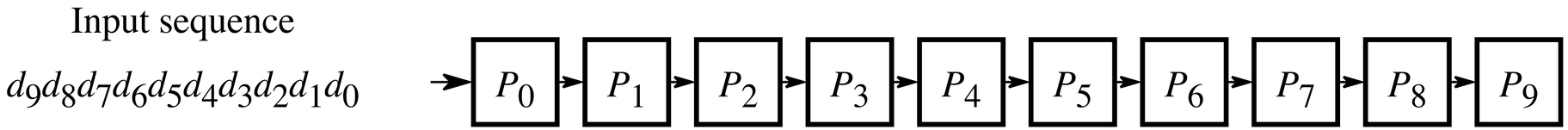


Slide credit: NSF Expeditions Center for Domain-Specific Computing (UCLA, Rice, OSU, UCSB)

Pipeline Parallelism: Another Example of Point-to-point Synchronization

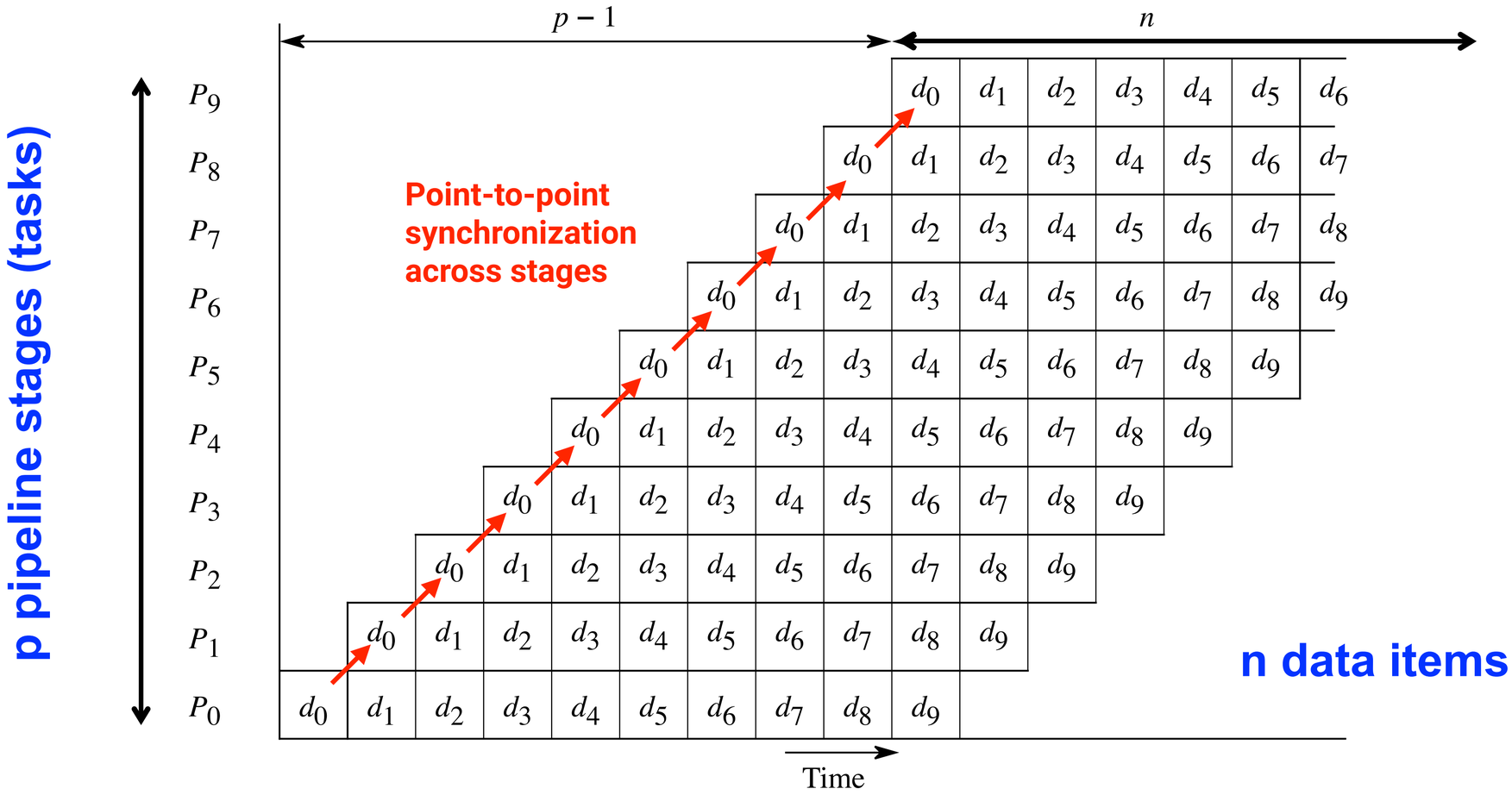
- Medical imaging pipeline with three stages
  1. Denoising stage generates a sequence of results, one per image.
  2. Registration stage's input is Denoising stage's output.
  3. Segmentation stage's input is Registration stage's output.
- Even though the processing is sequential for a single image, *pipeline parallelism* can be exploited via point-to-point synchronization between neighboring stages

General structure of a One-Dimensional Pipeline



- Assuming that the inputs d_0, d_1, . . . 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.

Complexity Analysis of One-Dimensional Pipeline

- Assume
  - n = number of items in input sequence
  - p = number of pipeline stages
  - each stage takes 1 unit of time to process a single data item
- WORK = n×p is the total work for all data items
- CPL = n + p − 1 is the critical path length of the pipeline
- Ideal parallelism, PAR = WORK/CPL = np/(n + p − 1)
- Boundary cases
  - p = 1 -> PAR = n/(n + 1 − 1) = 1
  - n = 1 -> PAR = p/(1 + p − 1) = 1
  - n = p -> PAR = p/(2 − 1/p) ≈ p/2
  - n ≫ p -> PAR ≈ p

Using a phaser to implement pipeline parallelism (unbounded buffer)

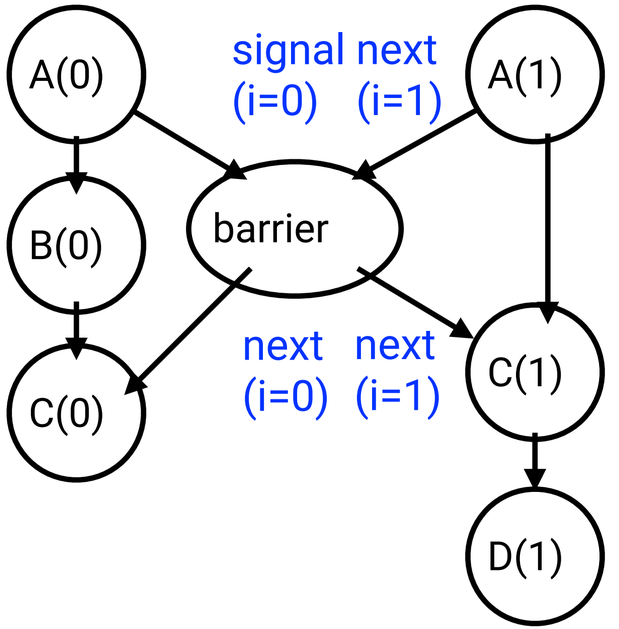
```
cPhased(ph.inMode(SIG), () -> {
 for (int i = 0; i < rounds; i++) {
     buffer.insert(…);
     // producer can go ahead as they are in SIG mode
     next();
 }


cPhased(ph.inMode(WAIT), () -> {
 for (int i = 0; i < rounds; i++) {
     next();
     buffer.remove(…);
 }
```


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")

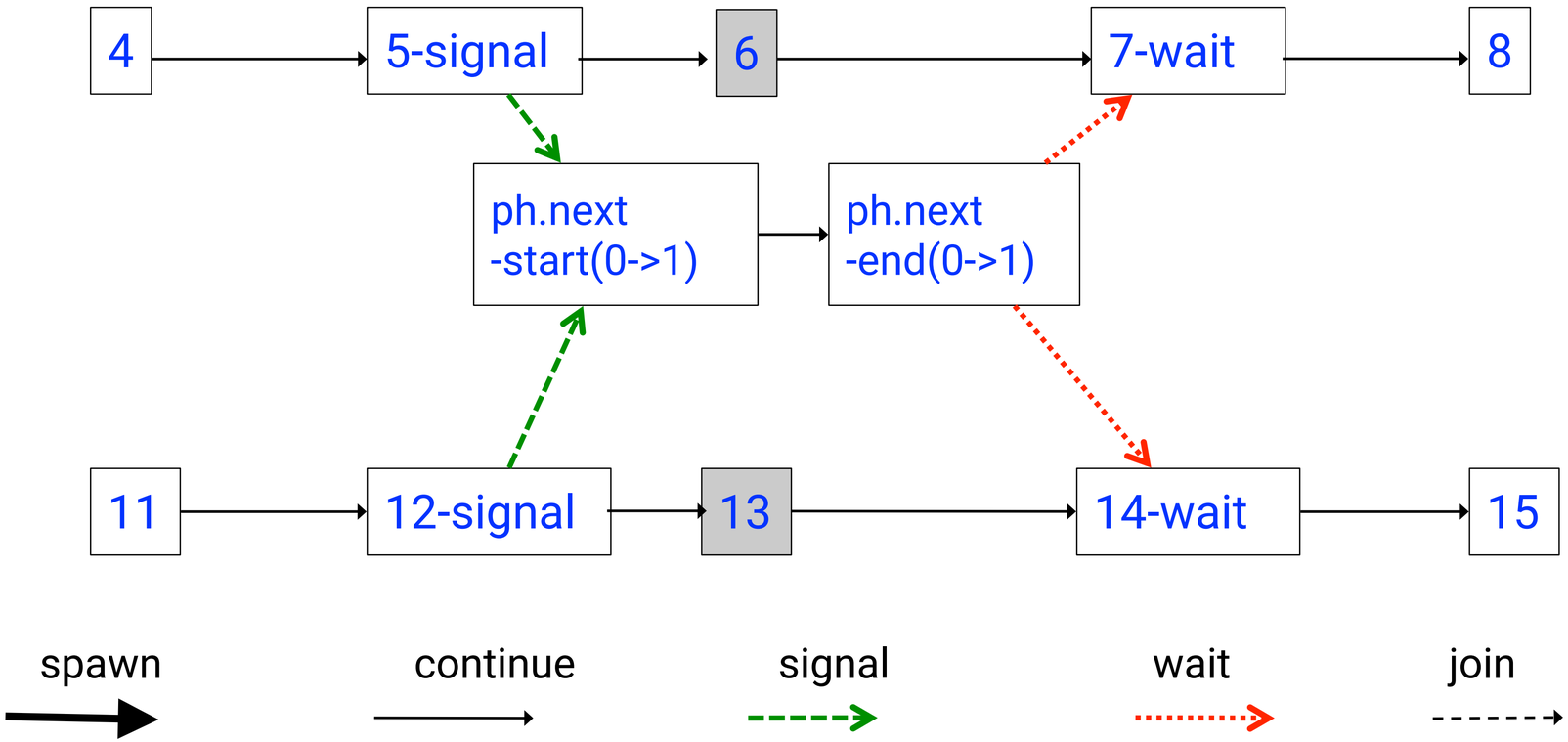
```
1. forall (point[i] : [0:1]) {
2.   A(i); // Phase 0
3.   if (i==0) { signal; B(i); }
4.   next; // Barrier
5.   C(i); // Phase 1
6.   if (i==1) { D(i); }
7. }
```



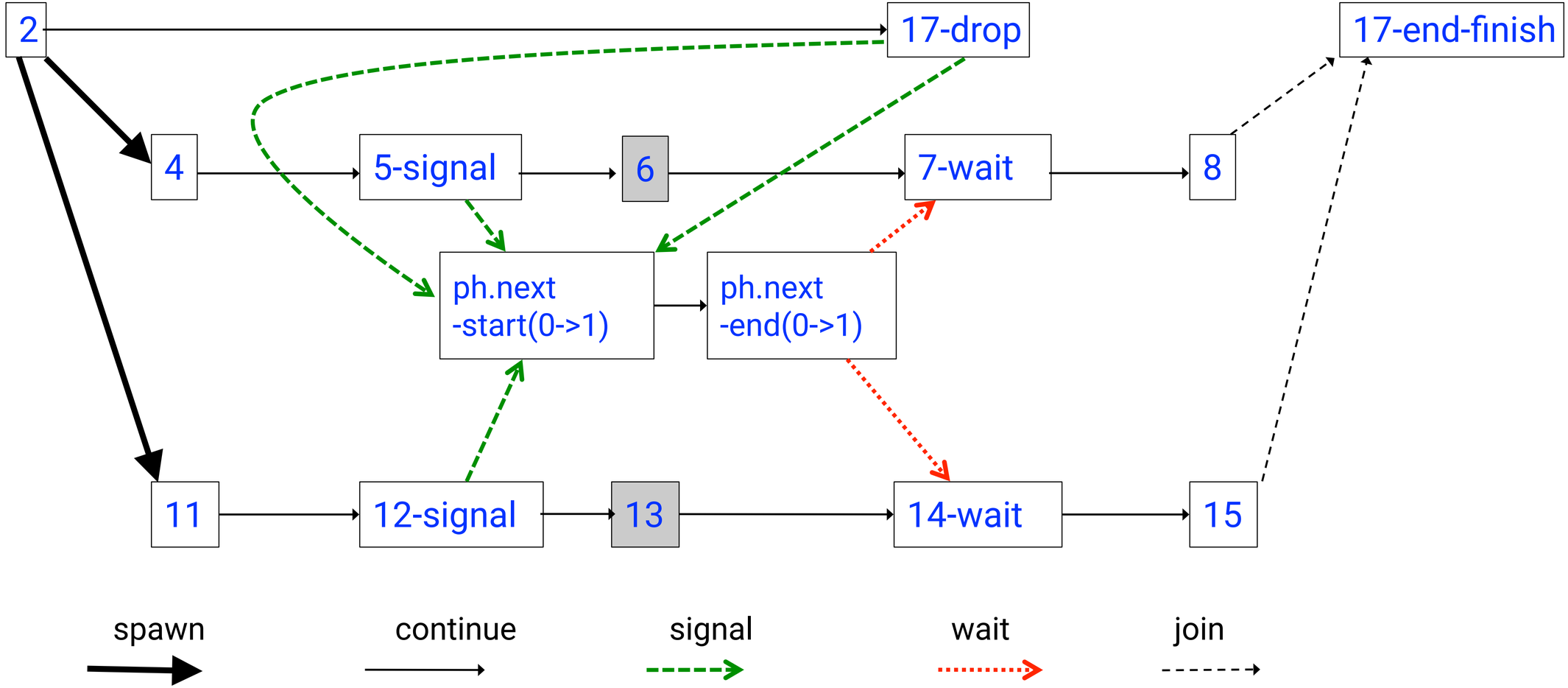
Another Example of a Split-Phase Barrier using the Signal Statement

```
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



Announcements & Reminders

- Lab 4 extension until Monday, Mar. 15th at 11:30am
- Quiz for Unit 3 (topics 3.1 - 3.7) due Monday, Mar. 15th by 11:59pm
- HW3 due Monday, April 5th by 11:59pm (includes written part)
  - Checkpoint 1 due Wednesday, March 24th by 11:59pm

Worksheet #17: Critical Path Length for Computation with Signal Statement

Compute the WORK and CPL values for the program shown below. How would they be different if the signal() statement was removed? (Hint: draw a computation graph as in slide 11)

```
1.finish(() -> {
2.  final HjPhaser ph = newPhaser(SIG_WAIT);
3.  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T1
4.    A(0); doWork(1);   // Shared work in phase 0
5.    signal();
6.    B(0); doWork(100); // Local work in phase 0
7.    next(); // Wait for T2 to complete shared work in phase 0
8.    C(0); doWork(1);
9.  });
10.  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T2
11.    A(1); doWork(1);   // Shared work in phase 0
12.    next(); // Wait for T1 to complete shared work in phase 0
13.    C(1); doWork(1);
14.    D(1); doWork(100); // Local work in phase 0
15.  });
16.}); // finish
```