COMP 322: Fundamentals of Parallel Programming

Lecture 19: Pipeline Parallelism, Signal Statement, Fuzzy Barriers

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Worksheet #18: Cooperative vs Blocking Runtime scheduler

Assume that creating an async causes the task to be pushed into the work queue for execution by any available idle thread.

Fill the following table for the program shown on the right by adding the appropriate number of threads required to execute the program. For the minimum or maximum numbers, your answer must represent a schedule where at some point during the execution all threads are busy executing a task or blocked on some synchronization constraint.

	Minimum number of threads	Maximum number of threads
Cooperative Runtime	1	?
Blocking Runtime	?	?

```
10. finish {
11.    async { S1; }
12.    finish {
13.      async {
14.        finish {
15.          async { S2; }
16.          S3;
17.        }
18.        S4;
19.      }
20.      async {
21.        async { S5; }
22.        S6;
23.      }
24.      S7;
25.    }
26.    S8;
27. }
```
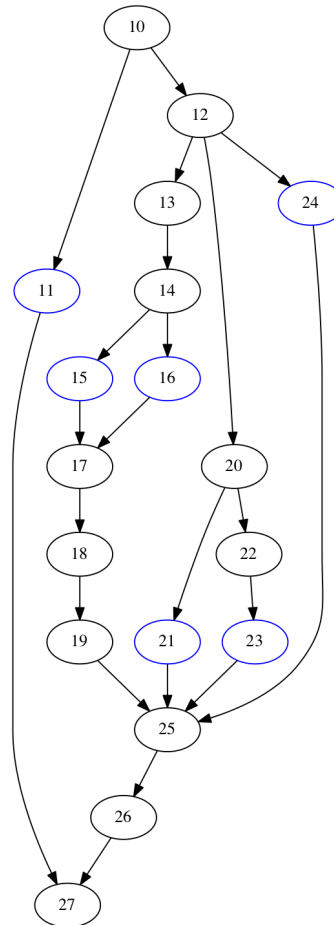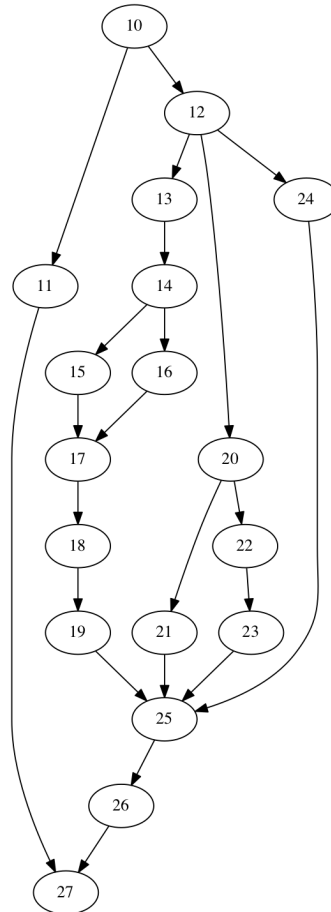
Worksheet #18: Cooperative vs Blocking Runtime scheduler

```
10. finish {
11.    async { S1; }
12.    finish {
13.       async {
14.          finish {
15.             async { S2; }
16.             S3;
17.          }
18.          S4;
19.       }
20.       async {
21.          async { S5; }
22.          S6;
23.       }
24.       S7;
25.    }
26.    S8;
27. }
```


	Maximum number of threads
Cooperative Runtime	6
Blocking Runtime	6

	Minimum number of threads
Cooperative Runtime	1
Blocking Runtime	?

Maximum threads: If we proceed through the graph in top-down manner incrementally, how many maximum leaf nodes can we have?

```
10. finish {
11.     async { S1; }
12.     finish {
13.         async {
14.             finish {
15.                 async { S2; }
16.                 S3;
17.             }
18.             S4;
19.         }
20.         async {
21.             async { S5; }
22.             S6;
23.         }
24.         S7;
25.     }
26.     S8;
27. }
```

work pool
11
13|20
21
21|15
21|15
15

Logical time	T1	T2	T3
0	10		
1	12	11	
2	24	13	20
3	25	14	22
4	25	16	23
5	25	17	21
6	25	17	15
7	25	17	
8	25	18	
9	25	19	
10	26		
11	27		
13			

	Minimum number of threads
Blocking Runtime	3

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


reconstruction

denoising

registration

segmentation

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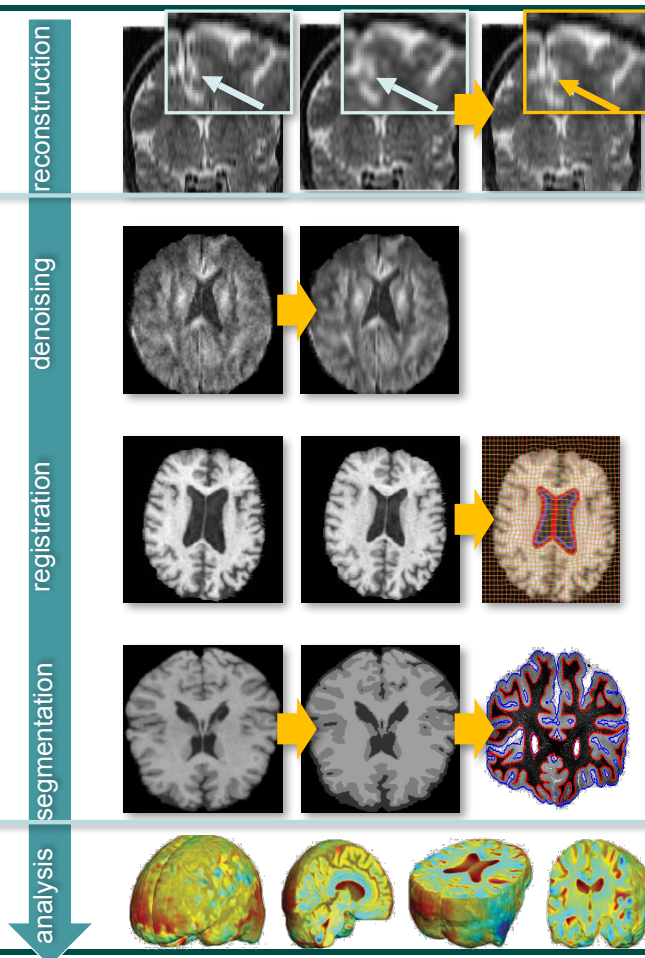
Pipeline Parallelism: Another Example of Point-to-point Synchronization

```
┌──────────┐      ┌──────────┐      ┌──────────┐
│ DENOISE  │ ───▶ │ REGISTER │ ───▶ │ SEGMENT  │
└──────────┘      └──────────┘      └──────────┘
```

- 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
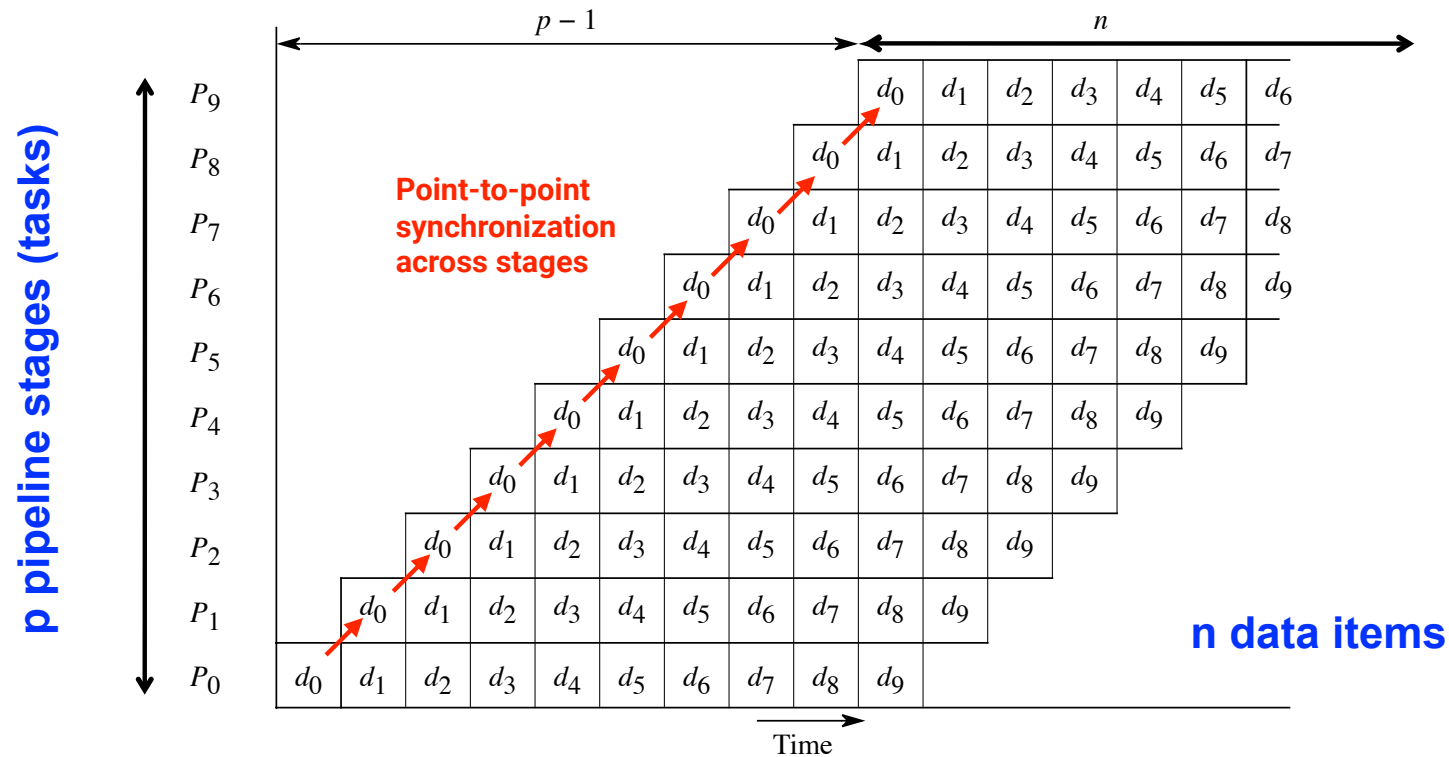
General structure of a One-Dimensional Pipeline

Input sequence

$d_9 d_8 d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0$ → P_0 ▸ P_1 ▸ P_2 ▸ P_3 ▸ P_4 ▸ P_5 ▸ P_6 ▸ P_7 ▸ P_8 ▸ P_9
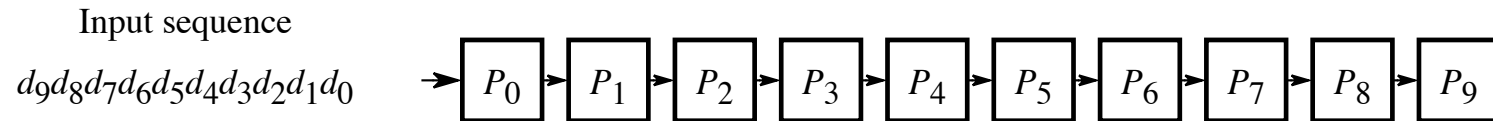
- Assuming that the inputs d_0, d_1, . . . arrive sequentially, pipeline parallelism can be exploited by enabling task (stage) P_i to work on item d_{k-i} when task (stage) P_0 is working on item d_k.

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)

```
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. });
```

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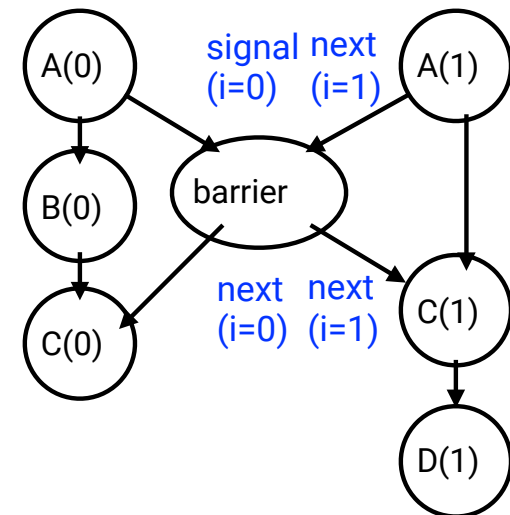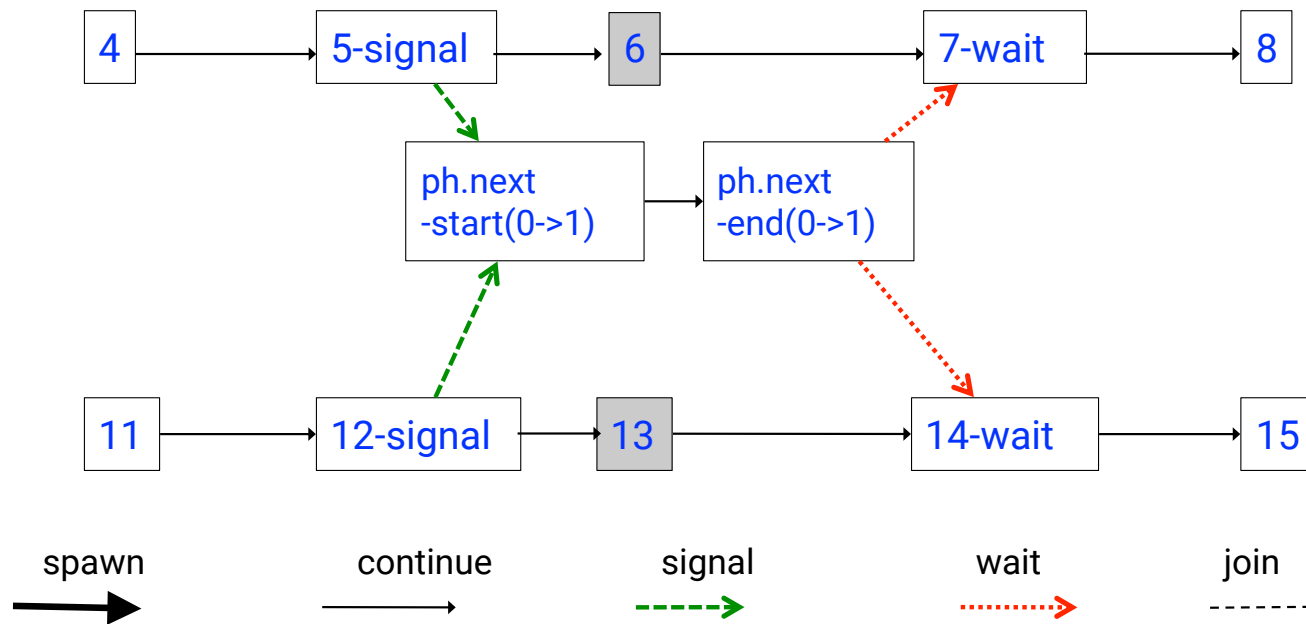
Another Example of a Split-Phase Barrier using the Signal Statement

```
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