COMP 322: Fundamentals of Parallel Programming

Lecture 15: Point-to-Point Synchronization with Phasers

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Worksheet #14 Solution: Iterative Averaging Revisited

Answer the questions in the table below for the versions of the Iterative Averaging code shown in slides 7, 8, 10, 11. Write in your answers as functions of m, n, and nc.

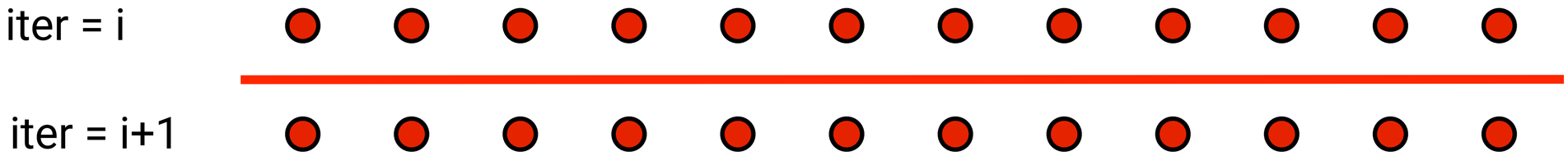
	Slide 7	Slide 8	Slide 10	Slide 11
How many tasks are created (excluding the main program task)?	m*n	n Incorrect: n * m	m*nc Incorrect: n * nc	nc Incorrect: n*m, m*nc
How many barrier operations (calls to next per task) are performed?	0 Incorrect: m	m Incorrect: m*n	0 Incorrect: m	m Incorrect: m*nc, nc

Which SPMD version is the most efficient?

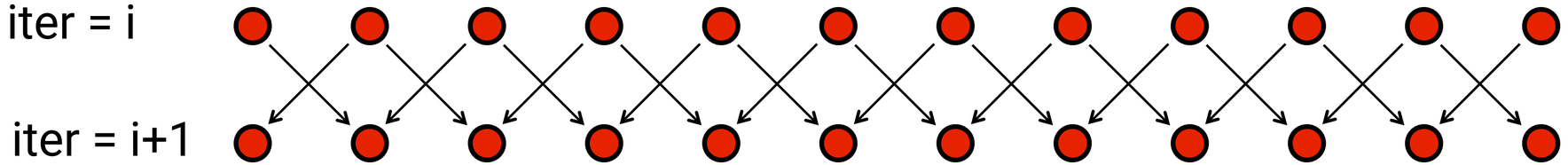
Announcements & Reminders

- No lab this week
- Midterm Exam on Thursday, Mar. 11th from 7pm - 9pm in Canvas
  - Open-book (class notes, slides, module)
  - No internet other than course site, Canvas.
- Quiz for Unit 3 (topics 3.1 - 3.7) due Monday, Mar. 15th by 11:59pm
- HW3 available today, due Monday, April 5th by 11:59pm
  - Checkpoint 1 due Wednesday, March 24th by 11:59pm

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



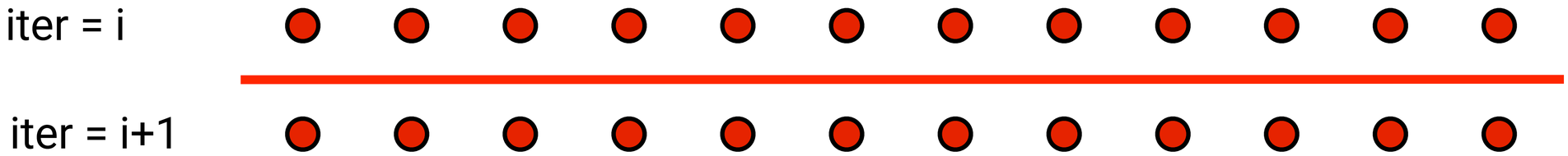
Barrier synchronization



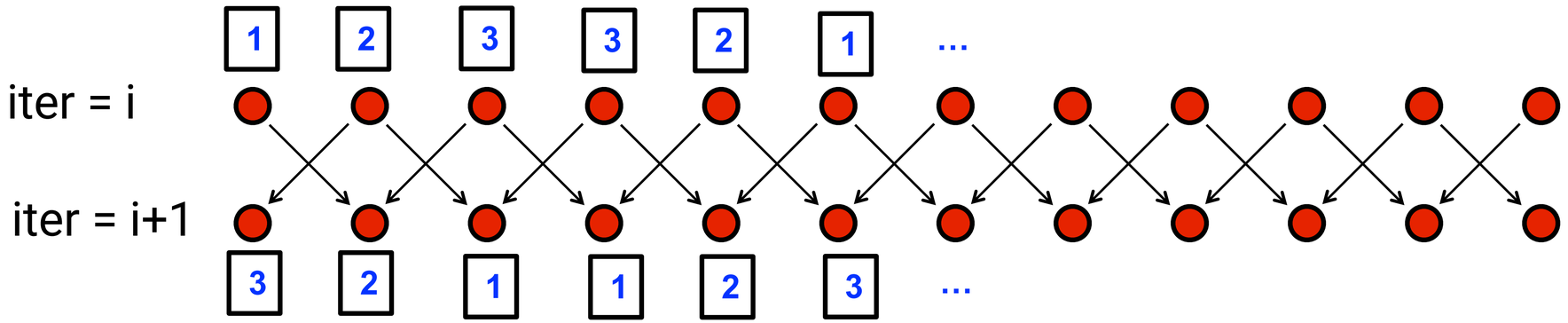
Point-to-point synchronization

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

```
1.  finish (() -> {
2.    ph = newPhaser(SIG_WAIT); // mode is SIG_WAIT
3.    asyncPhased(ph.inMode(SIG), () -> {
4.      // A1 (SIG mode)
5.      doA1Phase1(); next(); doA1Phase2(); });
6.    asyncPhased(ph.inMode(SIG_WAIT), () -> {
7.      // A2 (SIG_WAIT mode)
8.      doA2Phase1(); next(); doA2Phase2(); });
9.    asyncPhased(ph.inMode(HjPhaserMode.SIG_WAIT), () -> {
10.     // A3 (SIG_WAIT mode)
11.     doA3Phase1(); next(); doA3Phase2(); });
12.   asyncPhased(ph.inMode(HjPhaserMode.WAIT), () -> {
13.     // A4 (WAIT mode)
14.     doA4Phase1(); next(); doA4Phase2(); });
15.  });
```

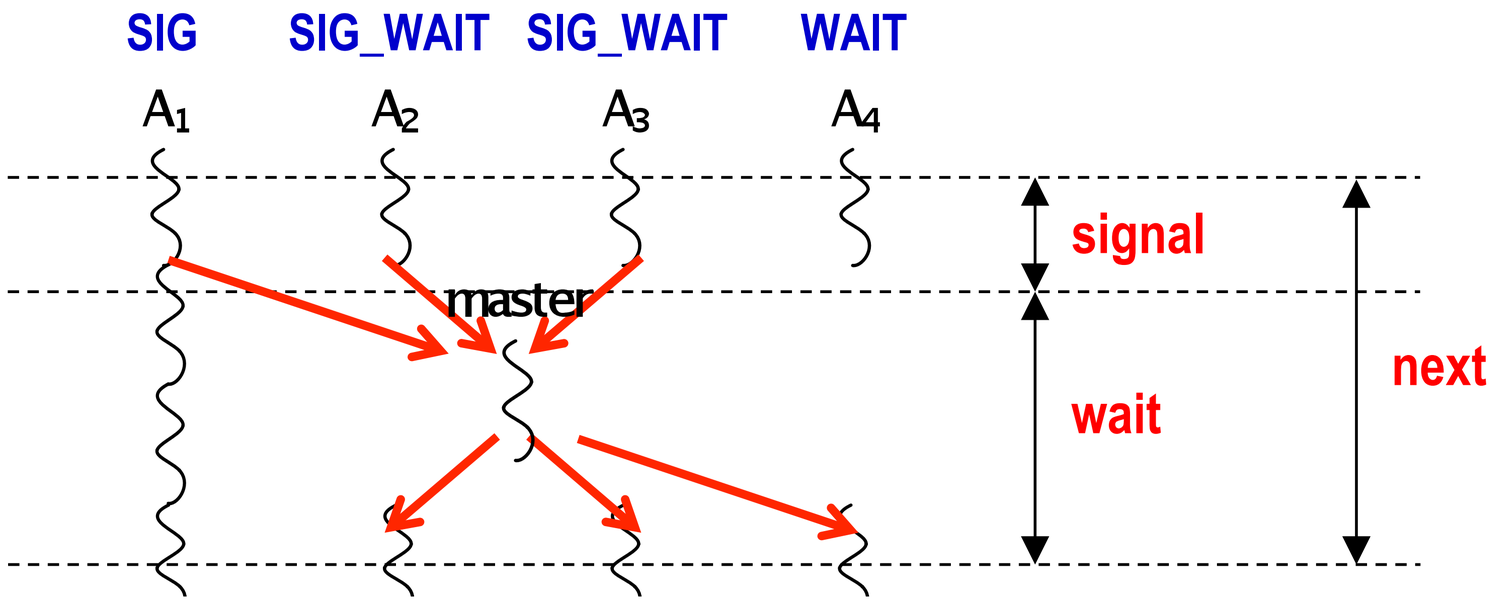

Computation Graph Schema Simple Example with Four Async Tasks and One Phaser

Semantics of next depends on registration mode

SIG_WAIT: **next = signal + wait**

SIG: **next = signal**

WAIT: **next = wait**

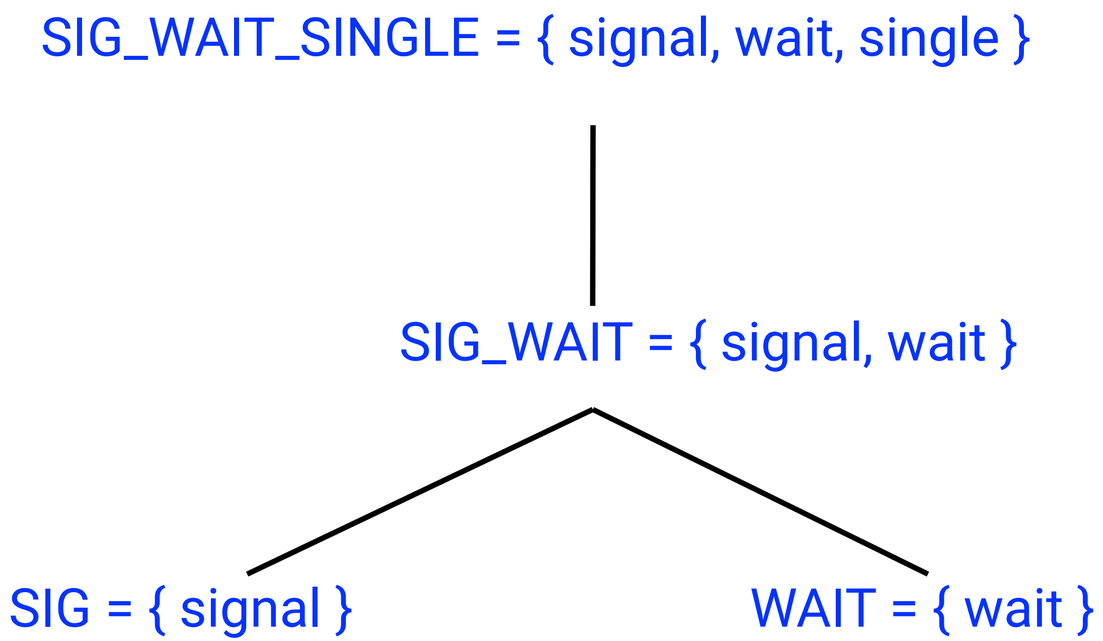


Summary of Phaser Construct

- 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 (which we will study later)
- 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

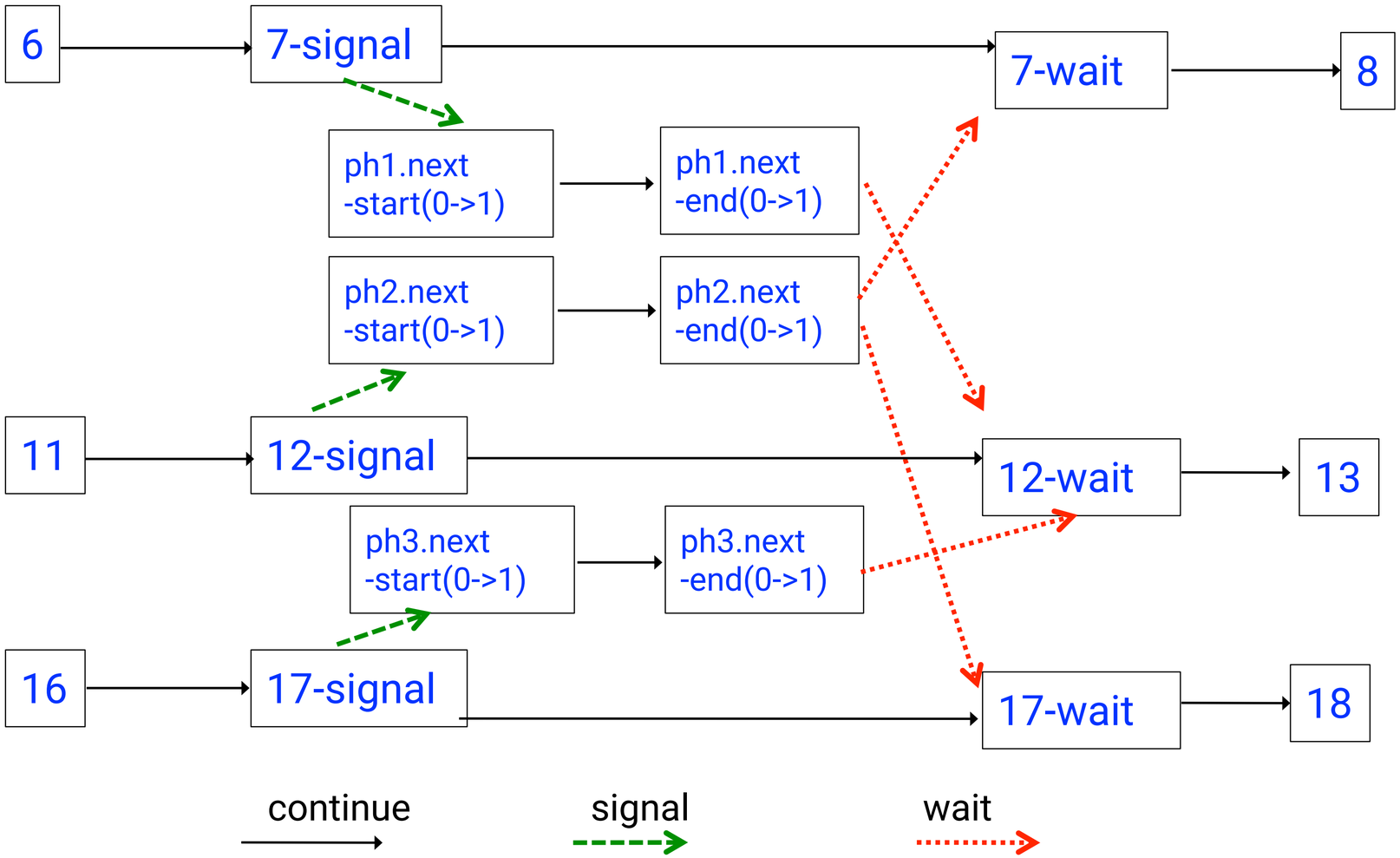
- A task can be registered in one of four modes with respect to a phaser: SIG_WAIT_SINGLE, SIG_WAIT, SIG, or WAIT. The mode defines the set of capabilities — signal, wait, single — that the task has with respect to the phaser. The subset relationship defines a natural hierarchy of the registration modes. A task can drop (but not add) capabilities after initialization.



Left-Right Neighbor Synchronization (with m=3 tasks)

```
1.finish(() -> { // Task-0
2.    final HjPhaser ph1 = newPhaser(SIG_WAIT);
3.    final HjPhaser ph2 = newPhaser(SIG_WAIT);
4.    final HjPhaser ph3 = newPhaser(SIG_WAIT);
5.    asyncPhased(ph1.inMode(SIG),ph2.inMode(WAIT),
6.      () -> { doPhase1(1);
7.        next(); // signals ph1, waits on ph2
8.        doPhase2(1);
9.    }); // Task T1
10.   asyncPhased(ph2.inMode(SIG),ph1.inMode(WAIT),ph3.inMode(WAIT),
11.     () -> { doPhase1(2);
12.       next(); // signals ph2, waits on ph3
13.       doPhase2(2);
14.    }); // Task T2
15.    asyncPhased(ph3.inMode(SIG),ph2.inMode(WAIT),
16.      () -> { doPhase1(3);
17.        next(); // signals ph3, waits on ph2
18.        doPhase2(3);
19.    }); // Task T3
20.}); // finish
```


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



forallPhased barrier is just an implicit phaser!

```
1. forallPhased(iLo, iHi, (i) -> {
2.   S1; next(); S2; next();{...}
3. });
```

is equivalent to

```
1. finish(() -> {
2.   // Implicit phaser for forall barrier
3.   final HjPhaser ph = newPhaser(SIG_WAIT);
4.   forseq(iLo, iHi, (i) -> {
5.     asyncPhased(ph.inMode(SIG_WAIT), () -> {
6.       S1; next(); S2; next();{...}
7.     }); // next statements in async refer to ph
8. });
```


Midterm exam

```
1. // Initially, assume all integers are prime.
2. final boolean[] isPrime = new boolean[N + 1];
3. for (int i = 2; i <= N; i++)
4.    isPrime[i] = true;
5. // mark non-primes <= N using Sieve of Eratosthenes
6. finish(() -> {
7.  for (int i = 2; i*i <= N; i++) {
8.    // If i is prime, then mark multiples of i as nonprime
9.    // It suffices to consider multiples i, i+1, ..., N/i
10.   if (isPrime[i])
11.      async(() -> {
12.       for (int j=i; i*j <= N; j++) {
13.         isPrime[i*j] = false;
14.         doWork(1);
15.       } // for
16.      }); // async
17. } // for
18.}); // finish
19.// Integer i is prime if and only if isPrime[i]=true
20.
```

- P1 = number of primes in the range 2..N
- P2 = number of primes in the range 2..square-root(N)

Worksheet #15: Reordered Asyncs with One Phaser

Task A4 has been moved up to line 6. Does this change the computation graph for slide 7? If so, draw the new computation graph. If not, explain why the computation graph is the same.

```
1. finish (() -> {
2.     ph = newPhaser(SIG_WAIT); // mode is SIG_WAIT
3.     asyncPhased(ph.inMode(SIG), () -> {
4.       // A1 (SIG mode)
5.       doA1Phase1(); next(); doA1Phase2(); });
6.     asyncPhased(ph.inMode(HjPhaserMode.WAIT), () -> {
7.      // A4 (WAIT mode)
8.      doA4Phase1(); next(); doA4Phase2(); });
9.     asyncPhased(ph.inMode(SIG_WAIT), () -> {
10.       // A2 (SIG_WAIT mode)
11.       doA2Phase1(); next(); doA2Phase2(); });
12.    asyncPhased(ph.inMode(HjPhaserMode.SIG_WAIT), () -> {
13.     // A3 (SIG_WAIT mode)
14.     doA3Phase1(); next(); doA3Phase2(); });
15.    });
```