COMP 322: Fundamentals of Parallel Programming

Lecture 16: Summary of Barriers and Phasers

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

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

The world according to COMP 322 before Barriers and Phasers

- Most of the parallel constructs that we learned during Lectures 1-12 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
    - forall (like "finish forasync") creates and waits for a set of tasks specified by an iteration region
  - future get() waits for a specific task
  - async await waits for a set of DataDrivenFuture values before starting
- The only construct that we learned for coordination within tasks was atomic variables
  - Accesses to atomic variables are "undirected" and nondeterministic
- Motivation for barriers and phasers
  - Directed synchronization within tasks
  - Separate from synchronization associated with task creation and termination

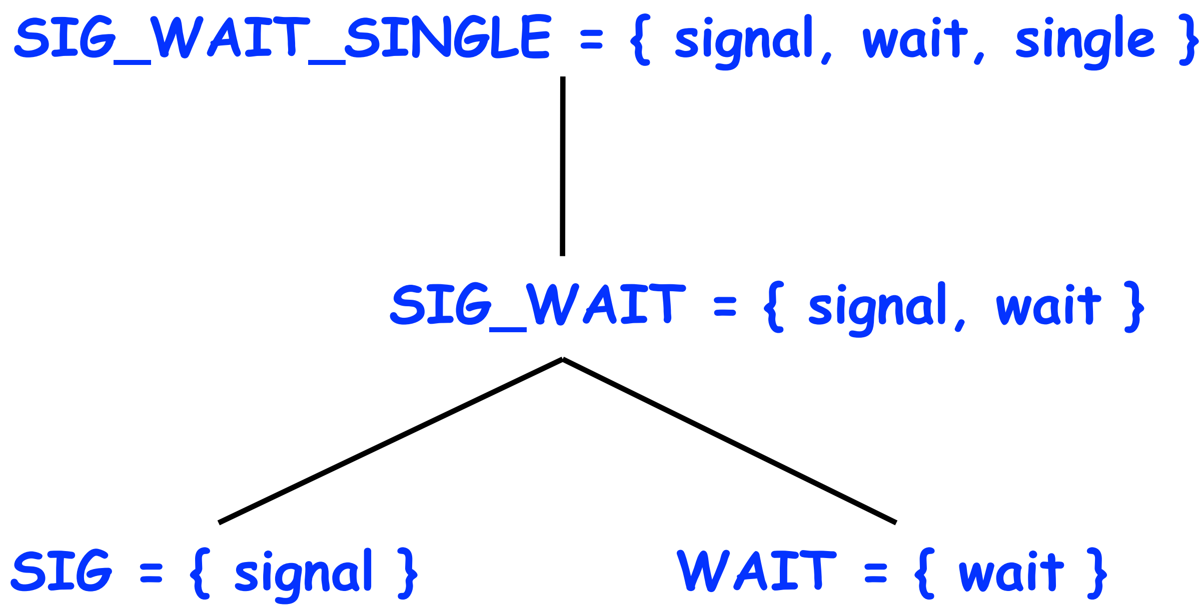
The world according to COMP 322 after Barriers and Phasers

- All directed synchronization can be expressed 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
    - next single -- augment barrier with "single" computations
  - 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_SIGNAL
    - signal statement can be used to support "fuzzy" barriers
    - phaser accumulators can perform per-phaser reduction
    - bounded phasers can limit how far ahead producer gets of consumers
    - phaser accumulators with bounded phasers can support bounded buffer streaming computations

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
  - NOTE: phaser WAIT has no relationship to Java wait/notify
- 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.

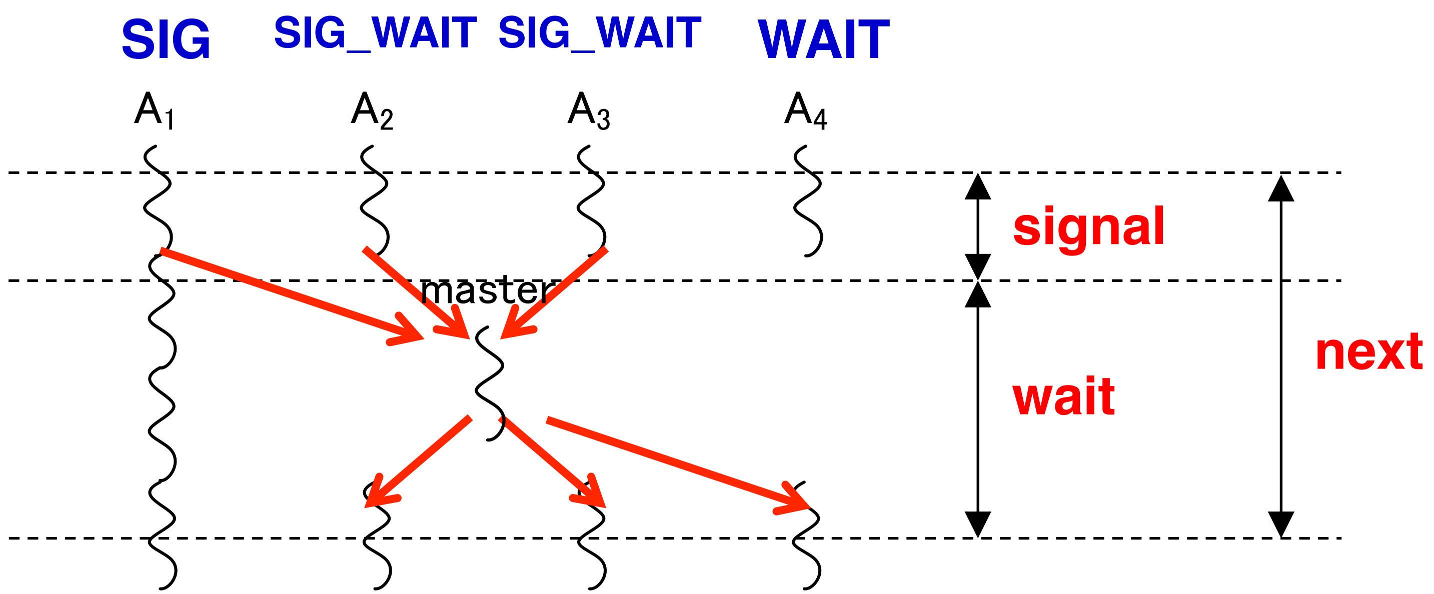
Simple Example with Four Async Tasks and One Phaser

```
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)**



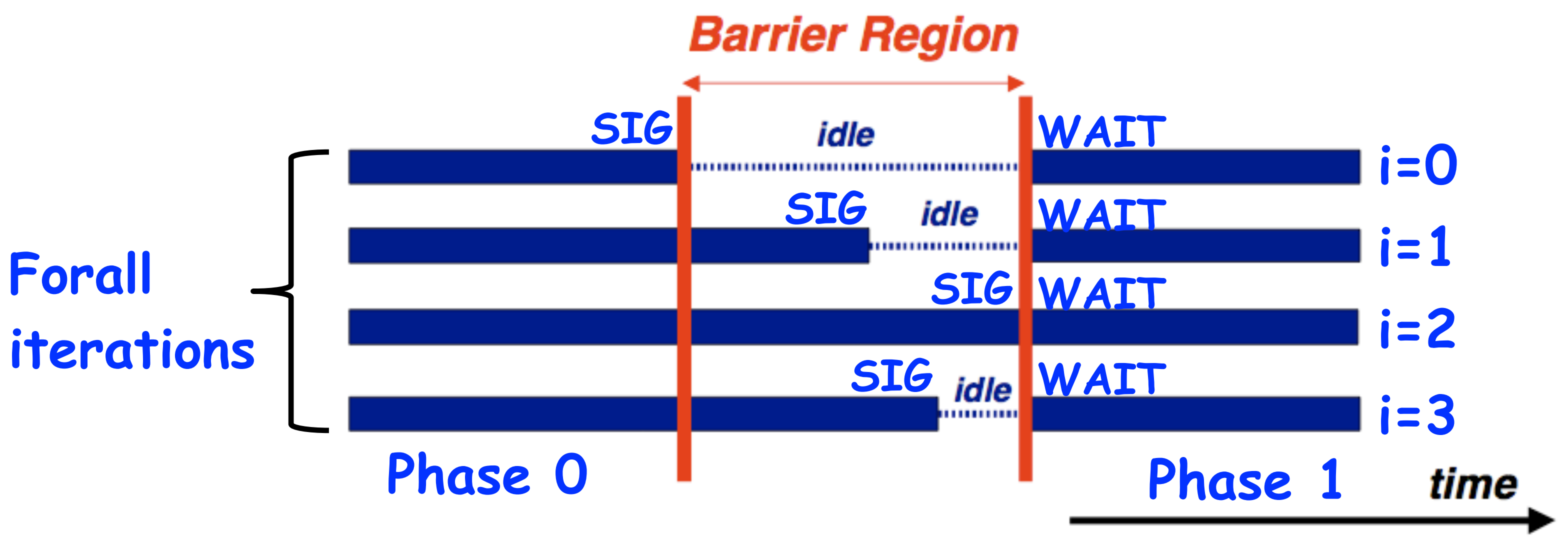
A master task receives all signals and broadcasts a barrier completion

HJ's forall statement = finish + forasync + barriers (next)

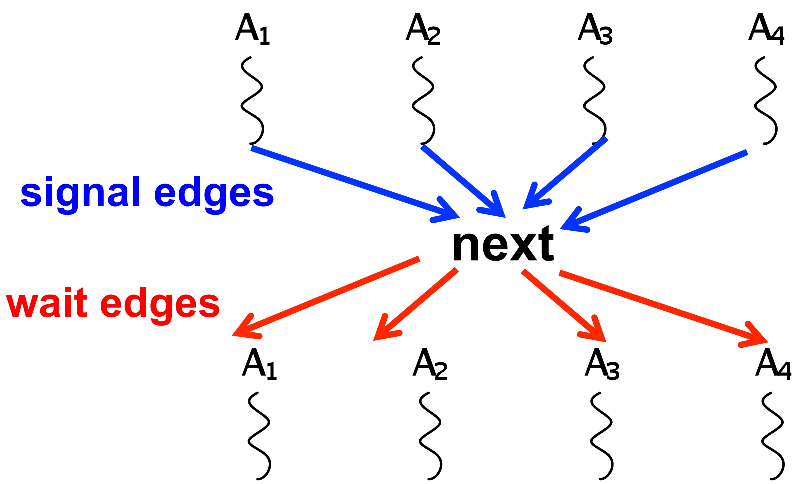
```
AtomicInteger rank = new AtomicInteger();
forall (point[i] : [0:m-1]) {
   int r = rank.getAndIncrement();                        } Phase 0
   System.out.println("Hello from task ranked " + r);
  next; // Acts as barrier between phases 0 and 1
  System.out.println("Goodbye from task ranked " + r);   } Phase 1
}
```

- *next* ➔ each forall iteration suspends at next until all iterations arrive (complete previous phase), after which the phase can be advanced
  - If a forall iteration terminates before executing "next", then the other iterations do not wait for it
  - Scope of synchronization is the closest enclosing forall statement
  - Special case of "phaser" construct (will be covered in following lectures)

Impact of barrier on scheduling forall iterations



Modeling a next operation in the computation graph



Recap of Observation 1 (Lecture 12): Scope of synchronization for “next” is closest enclosing forall statement

```
forall (point [i] : [0:m-1]) {
  System.out.println(“Starting forall iteration ” + i);
  next; // Acts as barrier for forall-i
  forall (point [j] : [0:n-1]) {
    System.out.println(“Hello from task (“ + i + “,”
                             + j + “)”);
    next; // Acts as barrier for forall-j
    System.out.println(“Goodbye from task (“ + i + “,”
                             + j + “)”);
  } // forall-j
  next; // Acts as barrier for forall-i
  System.out.println(“Ending forall iteration ” + i);
} // forall-i
```

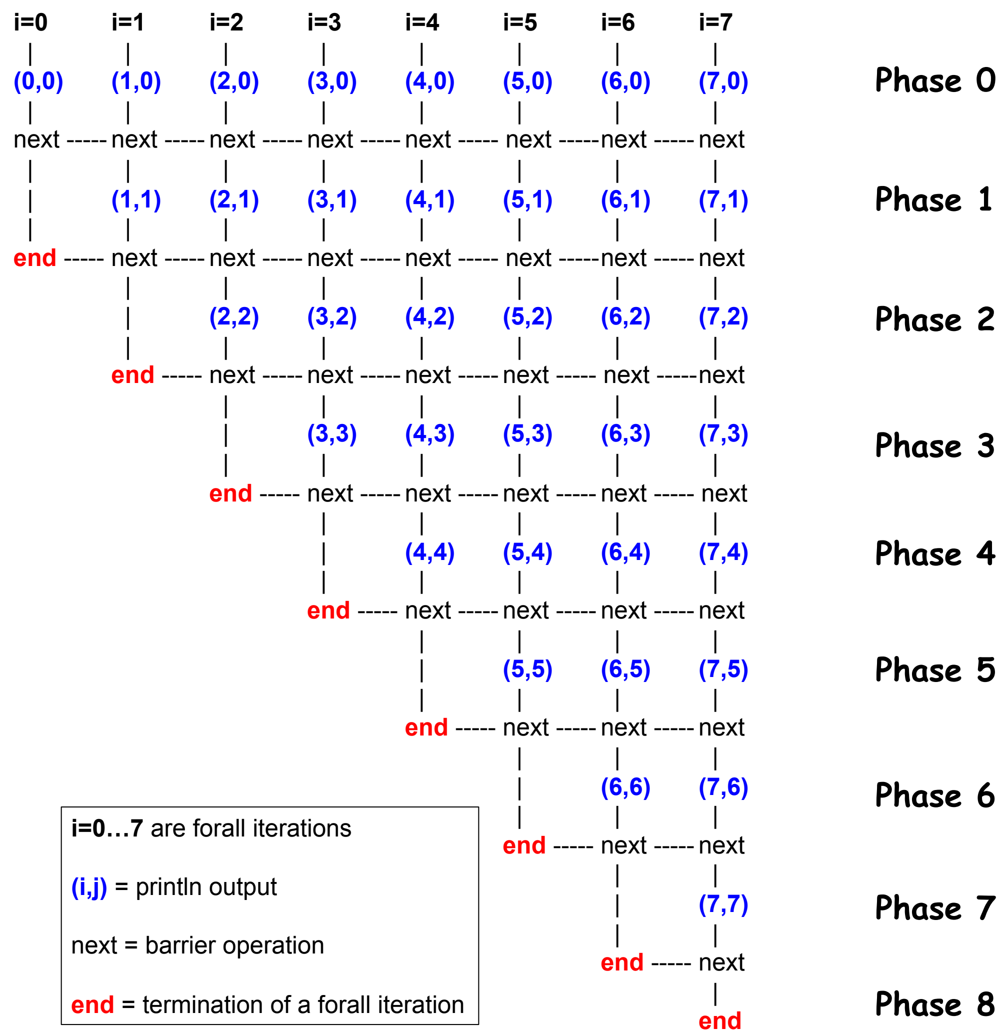
Recap of Observation 2 (Lecture 12): If a forall iteration terminates before “next”, then other iterations do not wait for it

```
1.  forall (point[i] : [0:m-1]) {
2.    for (point[j] : [0:i]) {
3.      // Forall iteration i is executing phase j
4.      System.out.println("(" + i + "," + j + ")");
5.      next;
6.    }
7.  }
```

- Outer forall-i loop has m iterations, 0…m-1
- Inner sequential j loop has i+1 iterations, 0…i
- Line 4 prints (task,phase) = (i, j) before performing a next operation.
- Iteration i = 0 of the forall-i loop prints (0, 0), performs a next, and then terminates. Iteration i = 1 of the forall-i loop prints (1,0), performs a next, prints (1,1), performs a next, and then terminates. And so on.

Illustration of Observation 2

- Iteration i=0 of the forall-i loop prints (0, 0) in Phase 0, performs a next, and then ends Phase 1 by terminating.
- Iteration i=1 of the forall-i loop prints (1,0) in Phase 0, performs a next, prints (1,1) in Phase 1, performs a next, and then ends Phase 2 by terminating.
- And so on until iteration i=8 ends an empty Phase 8 by terminating



Recap of Observation 3 (Lecture 12): Different forall iterations may perform "next" at different program points

```
1.  forall (point[i] : [0:m-1]) {
2.    if (i % 2 == 1) { // i is odd
3.      oddPhase0(i);
4.      next;
5.      oddPhase1(i);
6.    } else { // i is even
7.      evenPhase0(i);
8.      next;
9.      evenPhase1(i);
10.  } // if-else
11. } // forall
```

- Barrier operation synchronizes odd-numbered iterations at line 4 with even-numbered iterations in line 8
- next statement may even be in a method such as oddPhase1()

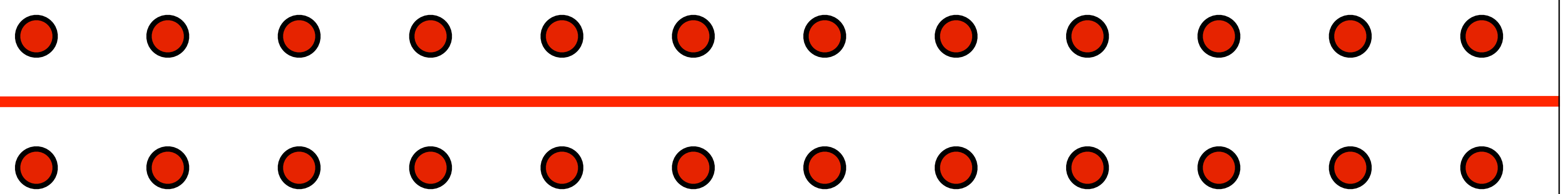
Use of next-with-single to print a log message between Hello and Goodbye phases

```
AtomicInteger rank = new AtomicInteger();
forall (point[i] : [0:m-1]) {
  // Start of Hello phase
  int r = rank.getAndIncrement();
  System.out.println("Hello from task ranked " + r);
  next single {
  System.out.println("LOG: Between Hello & Goodbye Phases");
  }
  // Start of Goodbye phase
  System.out.println("Goodbye from task ranked " + r);
} // forall
```

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

iter = i

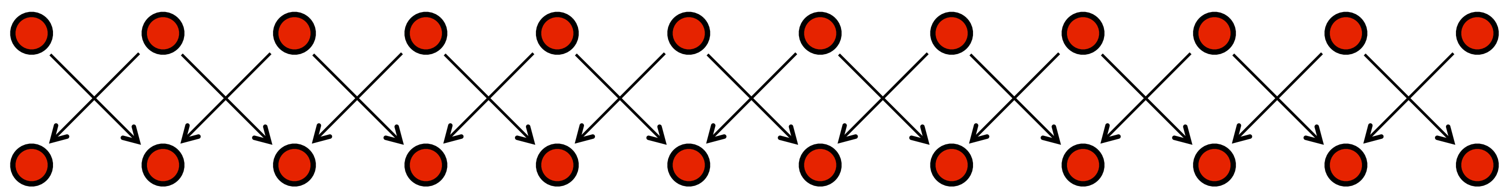
iter = i+1



Barrier synchronization

iter = i

iter = i+1

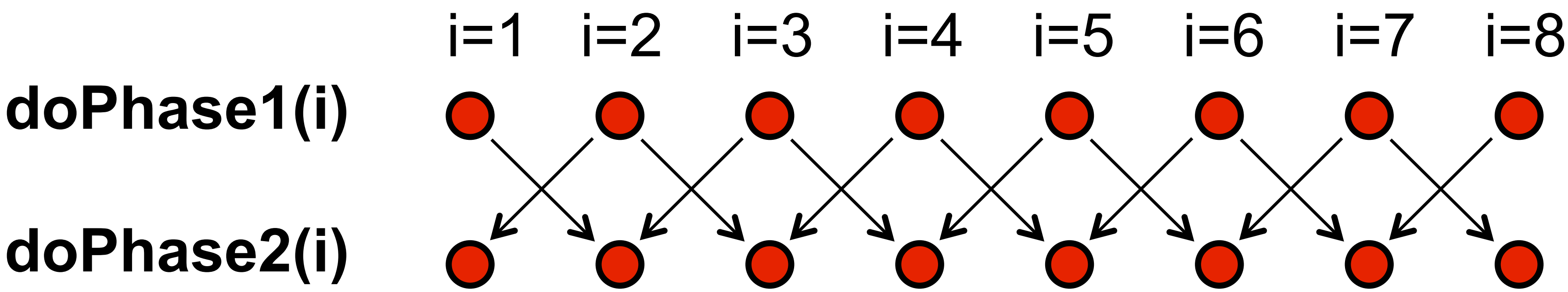


Point-to-point synchronization

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);
  }
}
```

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);
   }
}
```

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

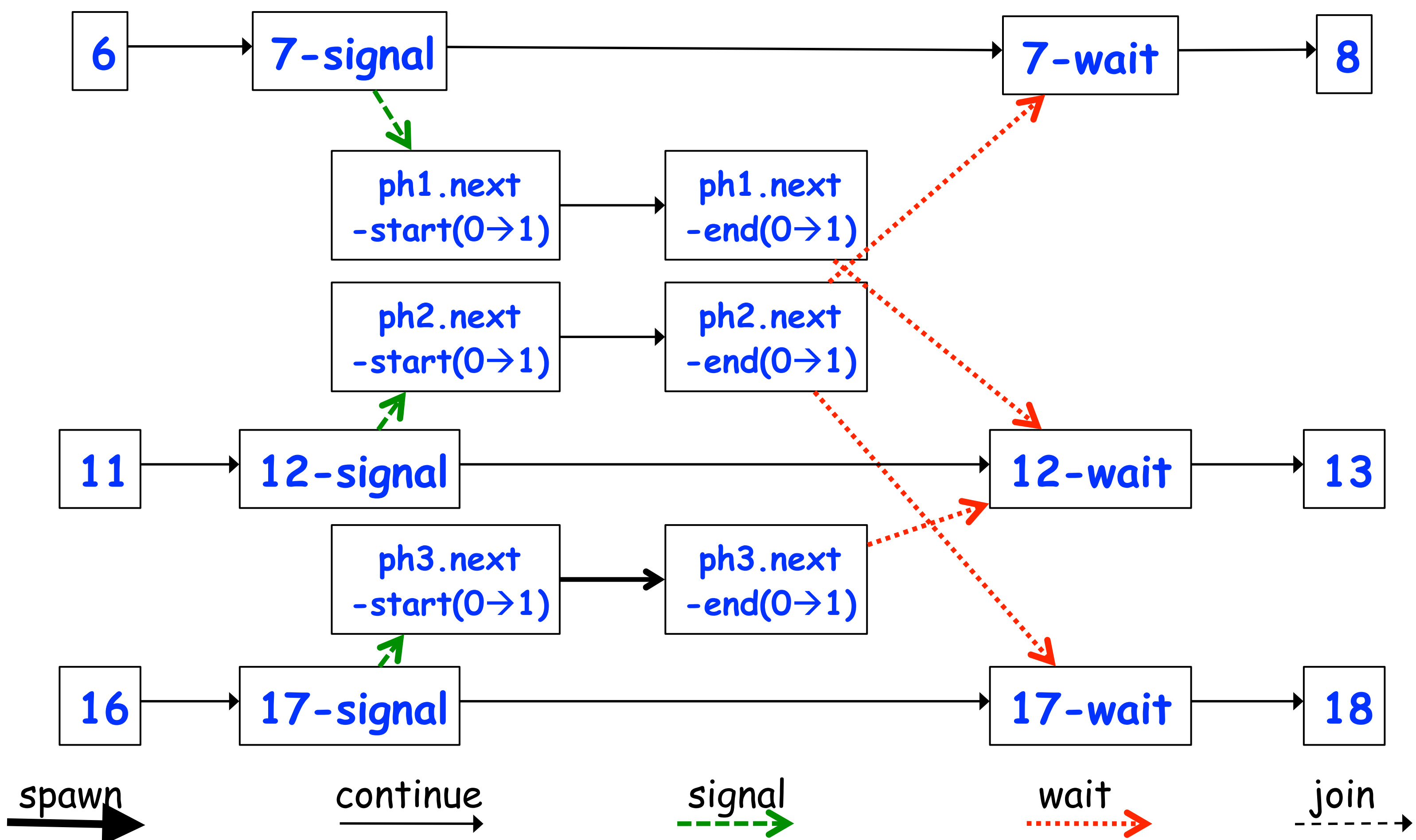
- ph.next-start(i→i+1) and ph.next-end(i→i+1)

Adding Phaser Operations to the Computation Graph (contd)

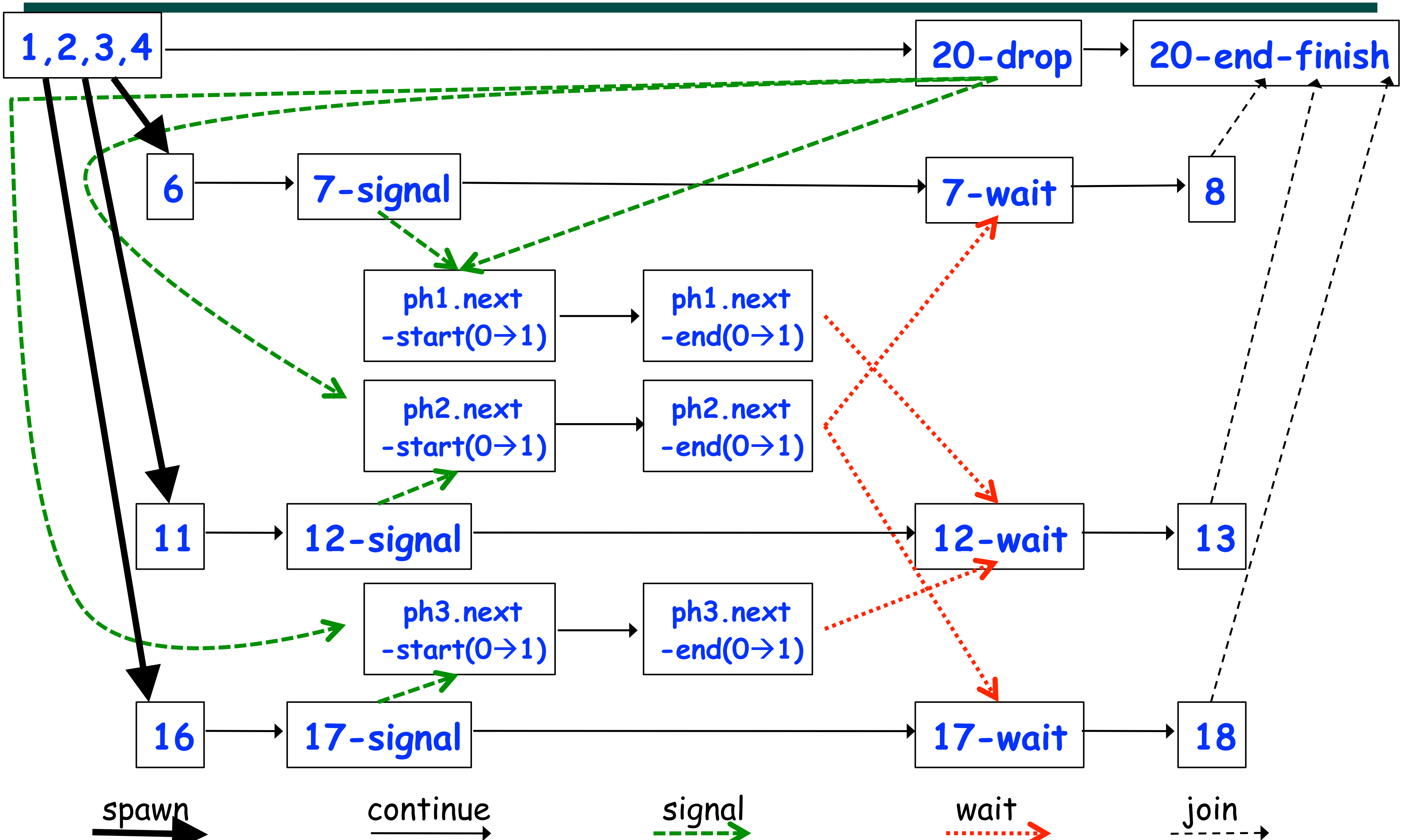
CG edges enforce ordering constraints among the nodes

- 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)

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



Full Computation Graph for m=3 example



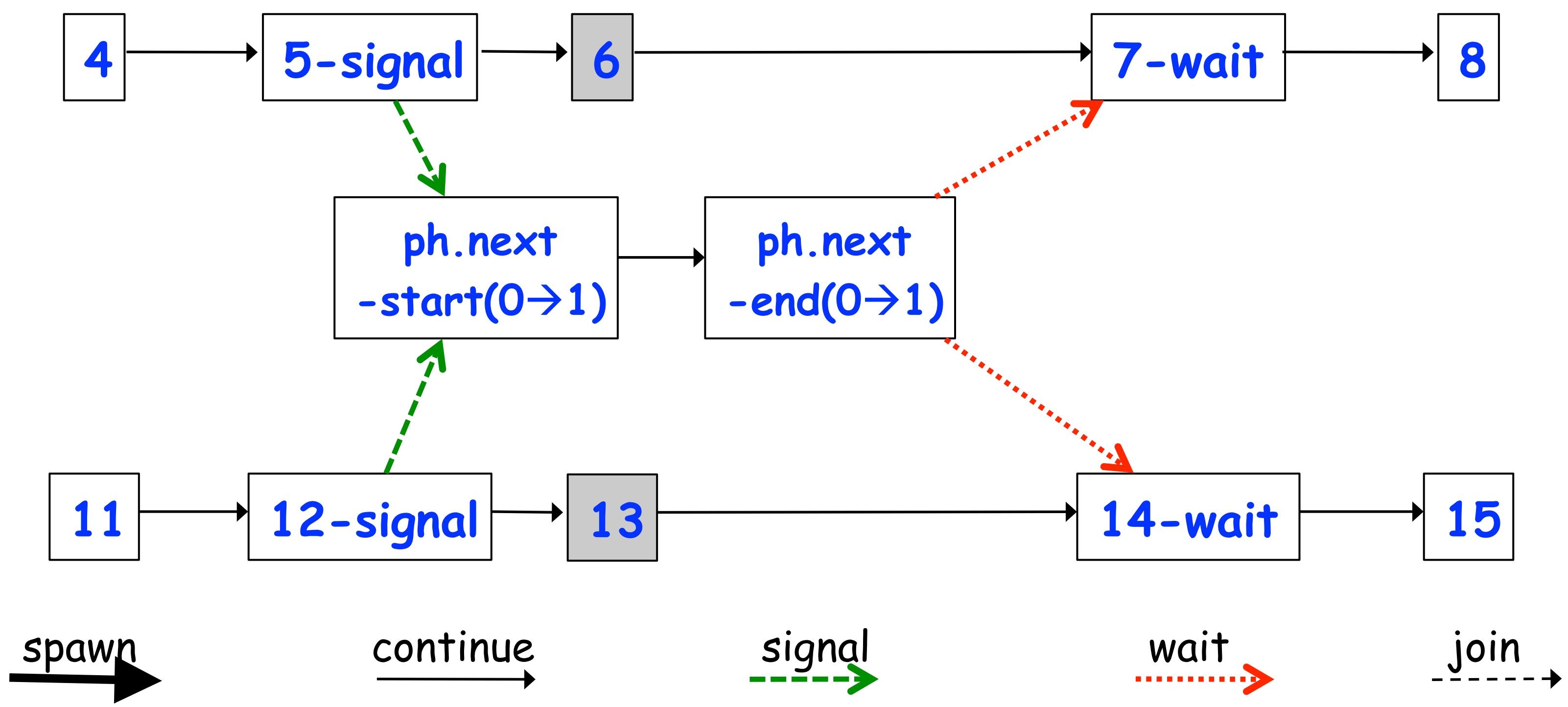
Signal statement

- 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"

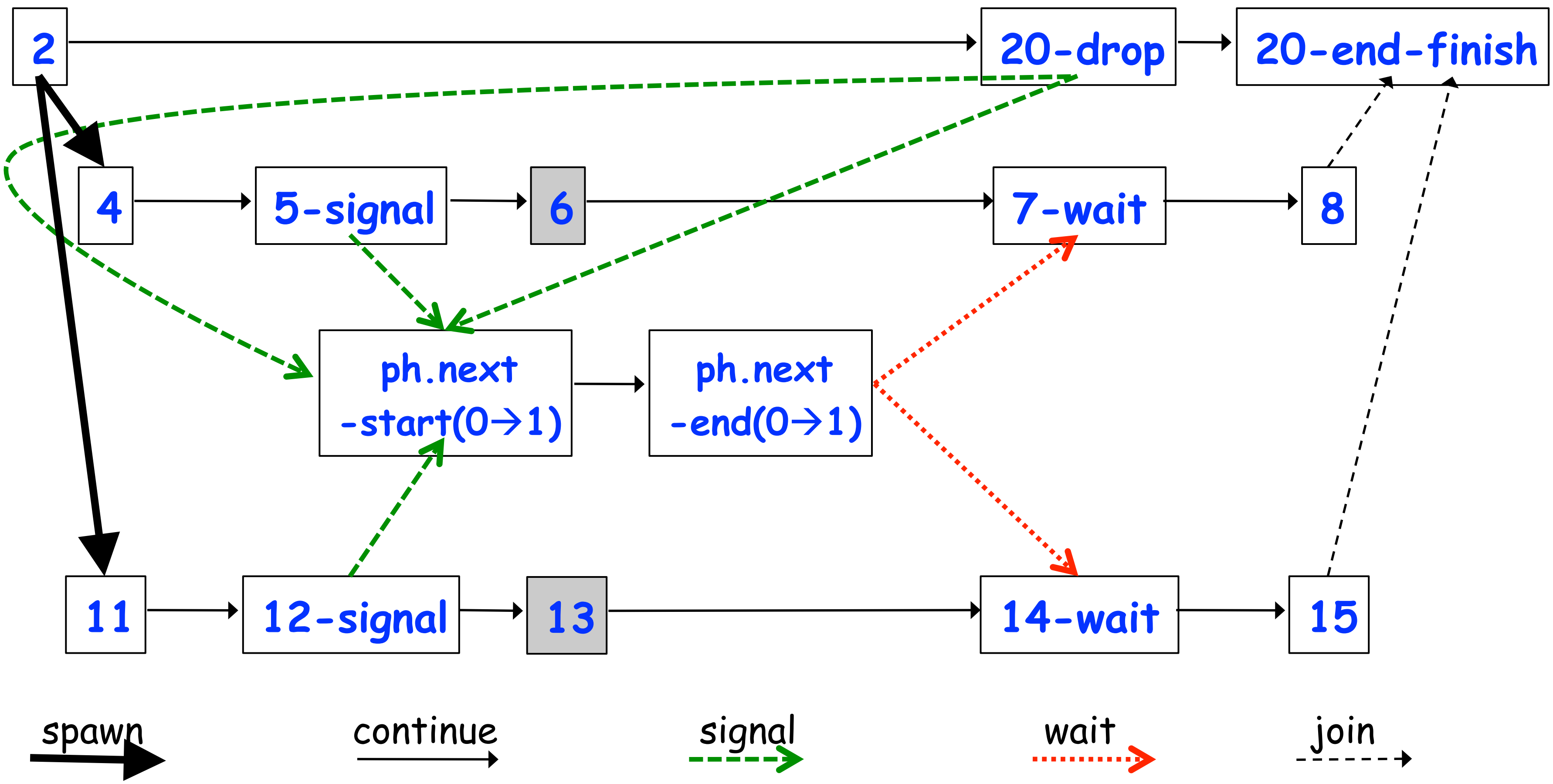
Example of Split-Phase Barrier

```
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



Operations on Phaser Accumulators

- **Creation**

```
accumulator ac = accumulator.factory.accumulator(op, type, phaser);
```

  - operator can be Operator.SUM, Operator.PROD, Operator.MIN, or Operator.MAX (as in finish accumulators)
  - type can be int.class or double.class (as in finish accumulators)
  - an extra "true" parameter results in lazy accumulation as in finish accumulators e.g., `accumulator.factory.accumulator(op, type, phaser, true)`

- **Accumulation**

```
ac.put(data);
```

  - data must be of type java.lang.Number, int, or double
  - Provides data for accumulation in <u>current</u> phase (can only be performed by a task registered on the phaser)

- **Retrieval**

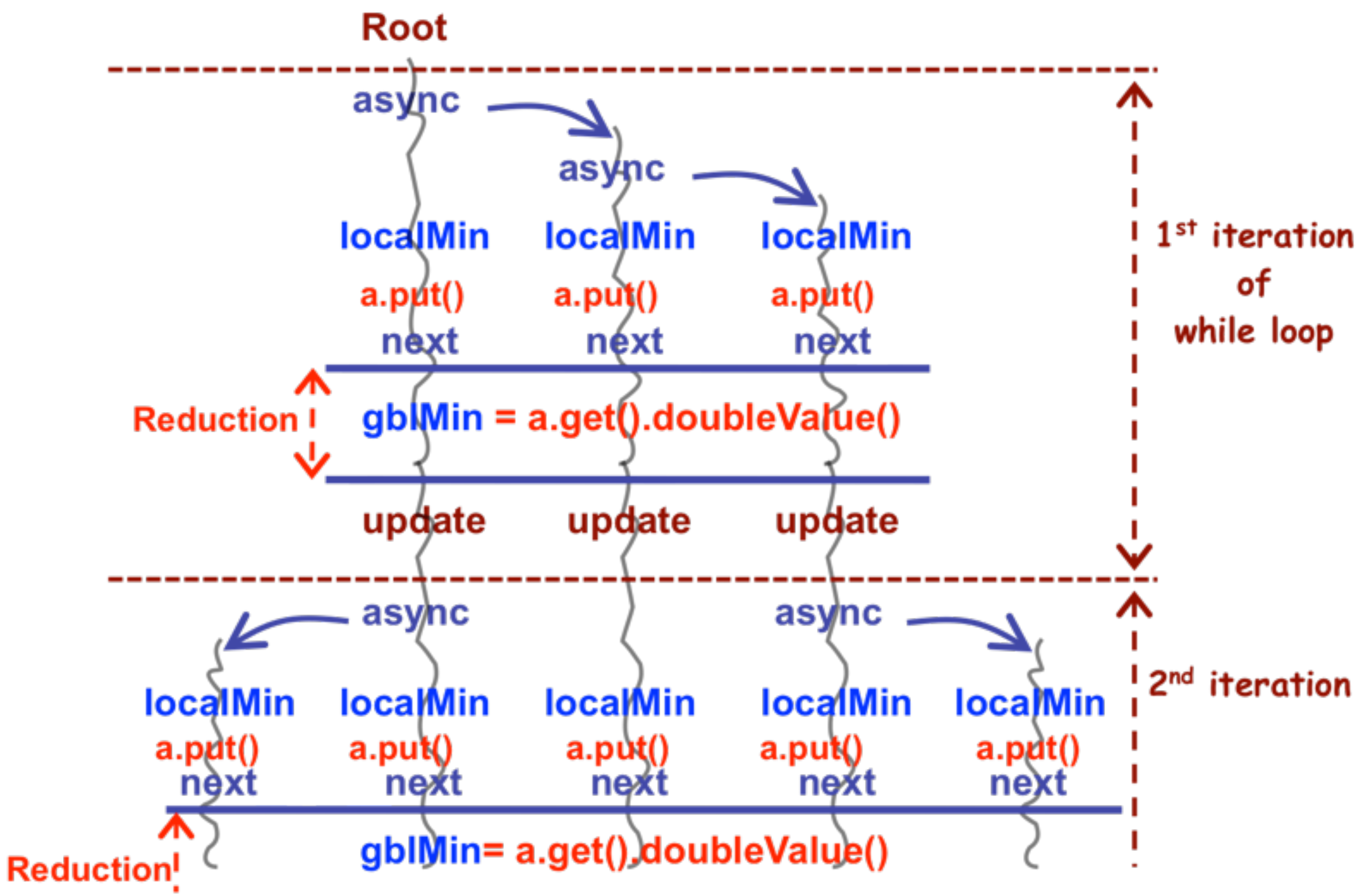
```
Number n = ac.get();
```

  - get() returns value from <u>previous</u> phase (can only be performed by a task registered on the phaser)
  - get() is non-blocking because the synchronization is handled by "next"
  - result from get() will be deterministic if HJ program does not use atomic or isolated constructs and is data-race-free (ignoring nondeterminism due to non-commutativity of arithmetic operations, e.g., underflow, overflow, rounding)

Example of Phaser Accumulators with Dynamic Parallelism: Search for Minimum Cost Solution

```
double gblMin = Double.MAX_VALUE;  double threshold = …;
SearchSpace gss = new SearchSpace(…); // Whole search space
finish {
  phaser ph = new phaser();
 accumulator a = accumulator.factory.accumulator(accumulator.MIN,
                                                      double.class, ph);
  calcMin(ph, gss, a);
}
. . .
void calcMin(phaser ph, SearchSpace mySs, accumulator a) {
  while (gblMin > threshold) {
    if (mySs.tooLarge()) {
      SearchSpace childSs = split(mySs);
       async phased { calcMin(ph, childSs, a); }
    }
   double localMin = findMin(mySs);
   a.put(localMin);
   next;
   gblMin = a.get().doubleValue();
   // update search spaces ...
  } // while
} // calcMin
```

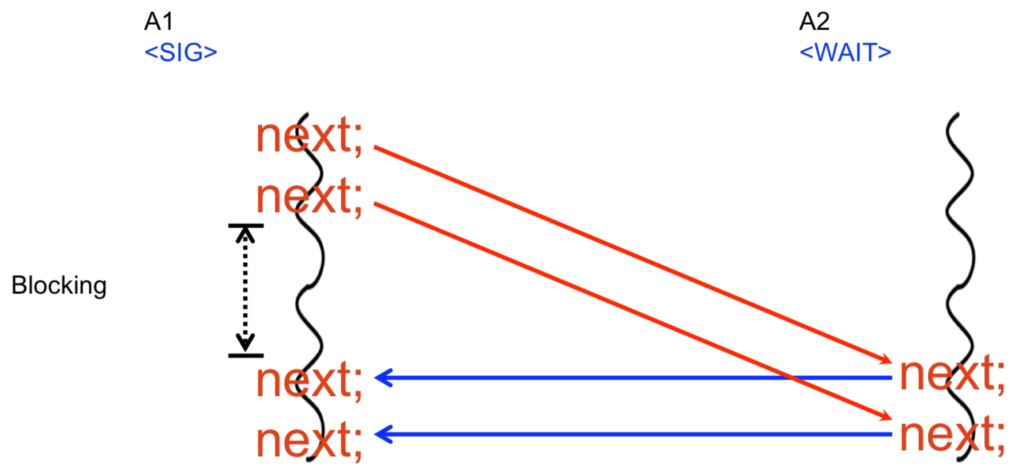
Execution of previous HJ program



Bound option in phasers

- Extra parameter in constructor
  - `new phaser(phaserMode m, int bound_size);`
- next operation
  - A task registered in SIG mode will block if it is >= bound_size phases past the current phase

```
...
phaser ph = new phaser(<SIG_WAIT>, 2 /*Bound size*/);
async phased (ph<SIG>) { next; next; ... /*A1*/ }
async phased (ph<WAIT>) { next; next; ... /*A2*/ }
...
```



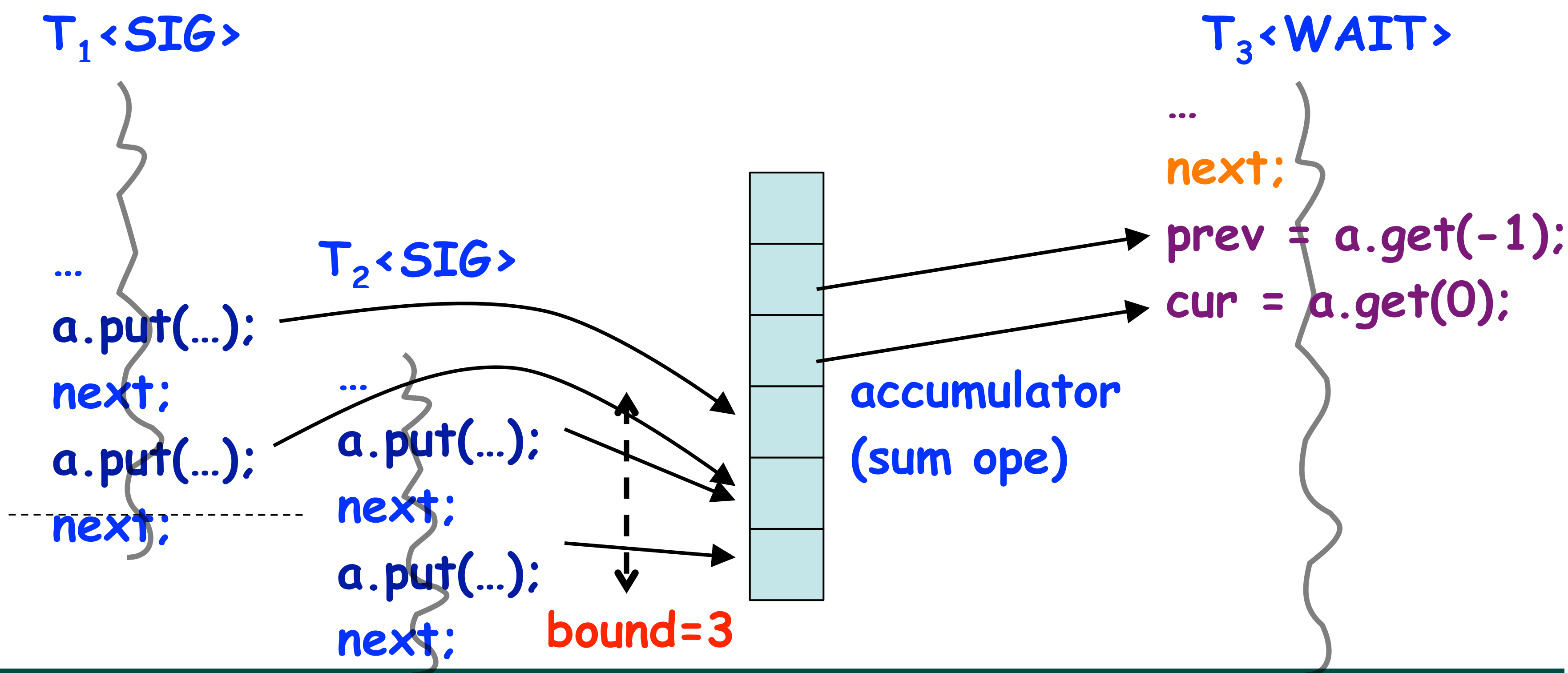
Single-Producer Single-Consumer Bounded Buffer

```
finish {
  phaser ph = new phaser(<SIG_WAIT>, bound_size);
  async phased (ph<SIG>)
    while (…) { insert(); next; } // producer
  async phased (ph<WAIT>)
   while (…) { next; remove(); } // consumer
}
```

Expanding Accumulators to support Bounded Buffers

```
phaser ph      = new phaser(SIG_WAIT, bound);
accumulator a = new accumulator(ph, SUM, double.class);
```

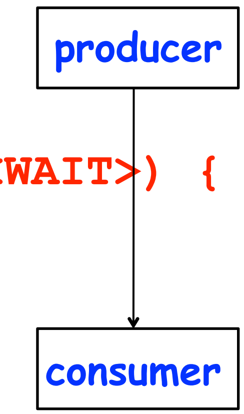
- Accumulator is now a bounded buffer
  - Stores results from bounded number of previous phase



Streaming Computations: Application of Bounded Buffer Computations

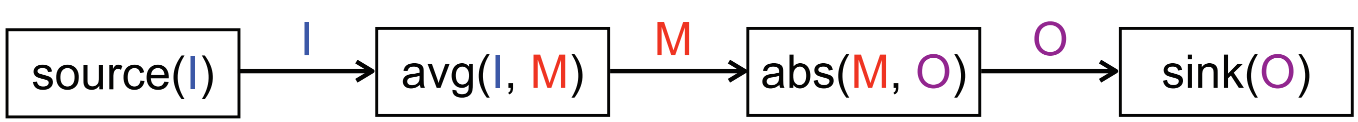
- Producer task (filter)
  - Insert data into stream
  - Can go ahead of consumers
  - Registered on phaser in SIG mode
- Consumer task (filter)
  - Consume data from stream
  - Must wait for producer
  - Registered on phaser in WAIT mode
- Streams
  - Manage communication among tasks
    - Retain data in bounded buffer
  - Accumulators can be expanded to implement bounded buffers
  - Need explicit phaser wait operation if a task needs to be both a producer and a consumer

```
phaser ph = new phaser();
async phased (ph<SIG>) {
  while(...) {
    wait;
    ...;
    ...
} }
async phased (ph<WAIT>) {
while(...) {
    ...
    next;
    ...
} }
```



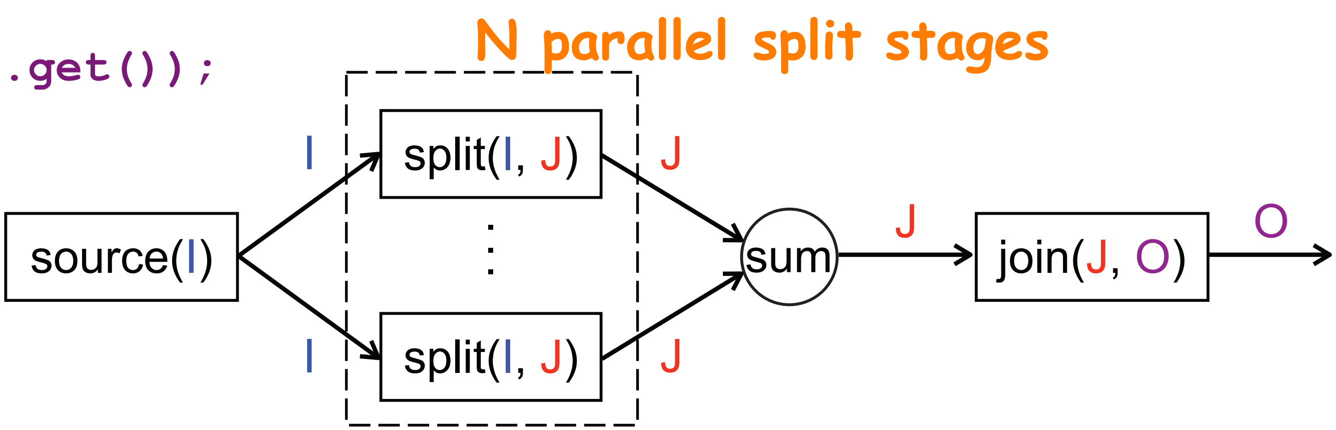
Streaming Computation: Pipeline

```
void Pipeline() {
    phaser phI      = new phaser(SIG_WAIT, bnd);
    accumulator I = new accumulator(phI, accumulator.ANY);
    phaser phM      = new phaser(SIG_WAIT, bnd);
    accumulator M = new accumulator(phM, accumulator.ANY);
    phaser phO      = new phaser(SIG_WAIT, bnd);
    accumulator O = new accumulator(phO, accumulator.ANY);
    async phased (phI<SIG>)                source(I);
    async phased (phI<WAIT>, phM<SIG>) avg(I,M);
    async phased (phM<WAIT>, phO<SIG>) abs(M,O);
    async phased (phO<WAIT>)               sink(O);
}
void avg(accumulator I, accumulator M) {
    while(...) {
        wait; wait;          // wait for two elements on I
        v1 = I.get(0);       // read first element
        v2 = I.get(-1);      // read second element (offset = -1)
        M.put((v1+v2)/2);    // put result on M
        signal;
} }
```



Streaming Patterns: Split-join

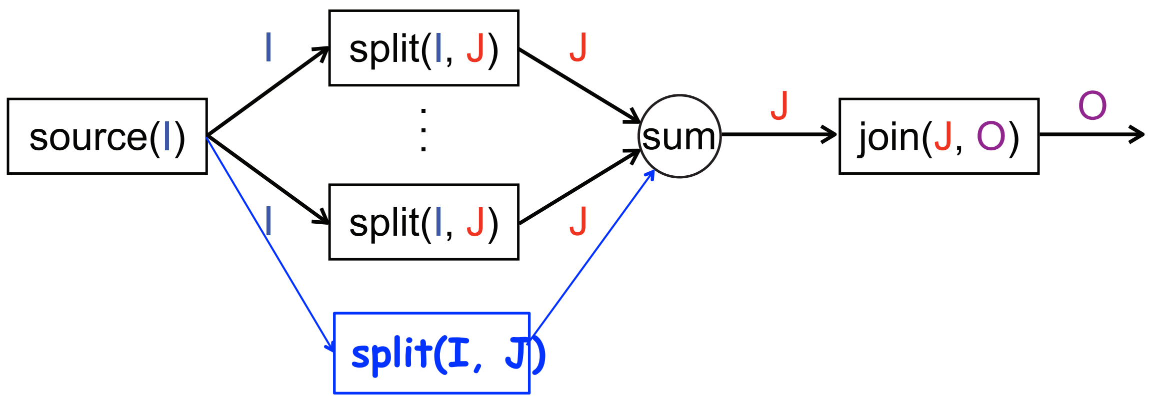
```
void Splitjoin() {
    phaser phI          = new phaser(SIG_WAIT, bnd);
    accumulator I       = new accumulator(phI, accumulator.ANY);
    phaser phJ          = new phaser(SIG_WAIT, bnd);
    accumulator J       = new accumulator(phJ, accumulator.SUM);

    async phased (phI<SIG>)                source(I);
    forasync (point [s] : [0:N-1])
        phased (phI<WAIT>, phJ<SIG>) split(I, J);
    async phased (phJ<WAIT>)               join(J);
}
split(I, J) {
    while(...) {
        wait;
        v = foo(I.get());
        J.put(v);
        signal;
} }
```



General Streaming Graphs with Dynamic Parallelism

- Dynamic split-join

```
dynamicSplit(I, J) {
  while(...) {
    if (spawnNewNode()) async phased dynamicSplit(I, J);
    if (terminate())    break;
    wait; ...
} }
```



Stages can be spawned/terminated dynamically

Announcements (REMINDER)

- 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
- No lab next week
  - Use the time for HW3 and to prepare for Exam 1
- 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