COMP 322: Fundamentals of Parallel Programming

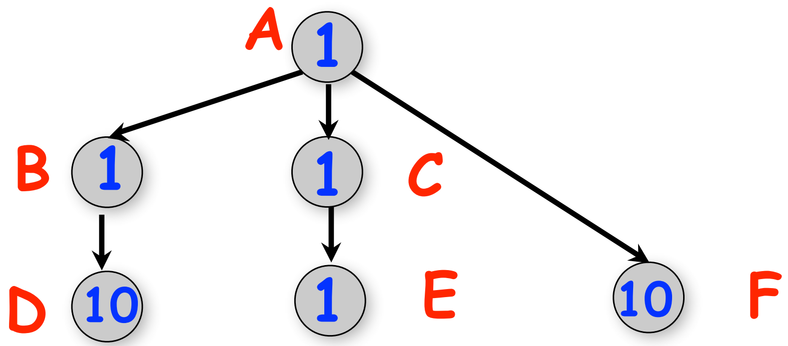
Lecture 16: Phasers, Point-to-Point Synchronization

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

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

Recap of Multiprocessor Scheduling of a Computation Graph (Lecture 3)



Schedule with execution time, $T_2 = 13$

Start time	Proc 1	Proc 2
0	A	
1	B	F
2	D	F
3	D	F
4	D	F
5	D	F
6	D	F
7	D	F
8	D	F
9	D	F
10	D	F
11	D	C
12		E
13		

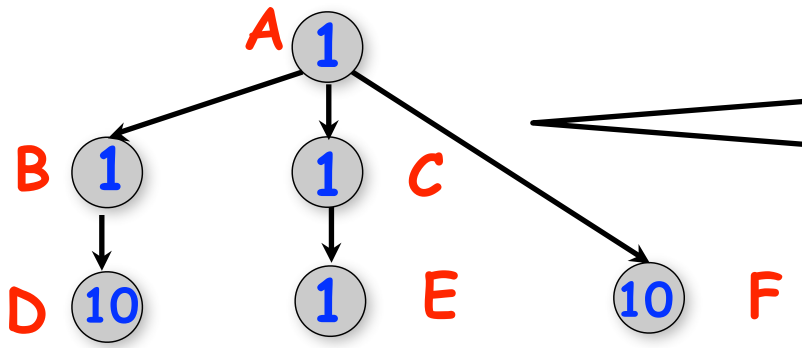
This schedule was obtained by mapping computation graph nodes to processor assuming:

1. Non-preemption (no context switch in the middle of a node)

2. Greedy schedule (a processor is never idle if work is available)

There may be multiple possible schedules with these assumptions

Two possible HJ programs for this Computation Graph (there can be others ...)



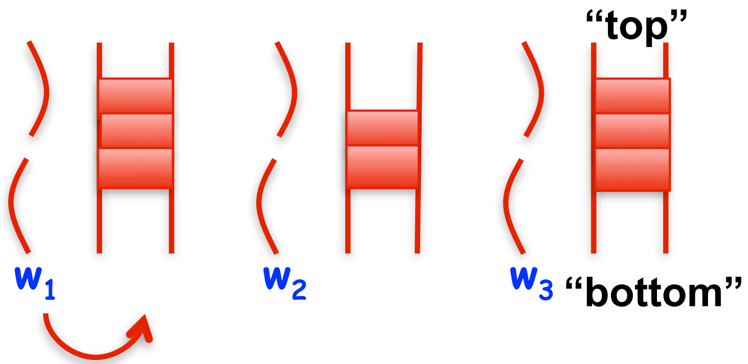
There is no significance to the left-to-right ordering of edges in a computation graph, which is why there can be multiple parallel programs for the same computation graph

```
// Program Q1
A;
finish {
  async { B; D; }
  async F;
  async { C; E; }
}
```

```
// Program Q2
A;
finish {
  async { C; E; }
  async F;
  async { B; D; }
}
```

Work-first vs. Help-first work-stealing policies (Lec 15)

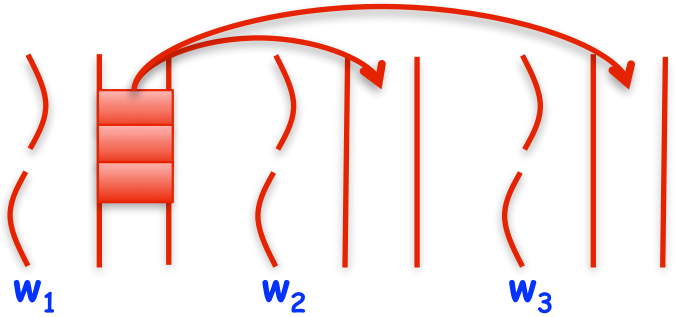
- When encountering an async
  - Help-first policy
    - Push async on “bottom” of local queue, and execute next statement
  - Work-first policy
    - Push continuation (remainder of task starting with next statement) on “bottom” of local queue, and execute async



Local push/pop by w_1

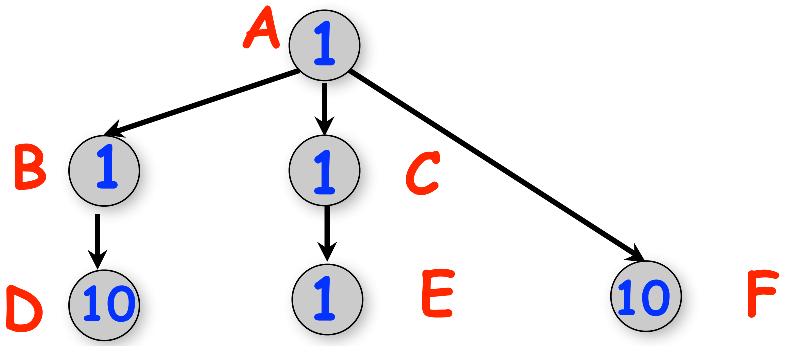
- When encountering the end of a finish scope
  - Help-first policy & Work-first policy
    - Store continuation for end-finish
      - Will be resumed by last async to complete in finish scope
    - Pop most recent item from “bottom” of local queue
    - If local queue is empty, steal from “top” of another worker’s queue

Stealing by w_2 and w_3



- Current HJ-lib runtime only supports help-first policy

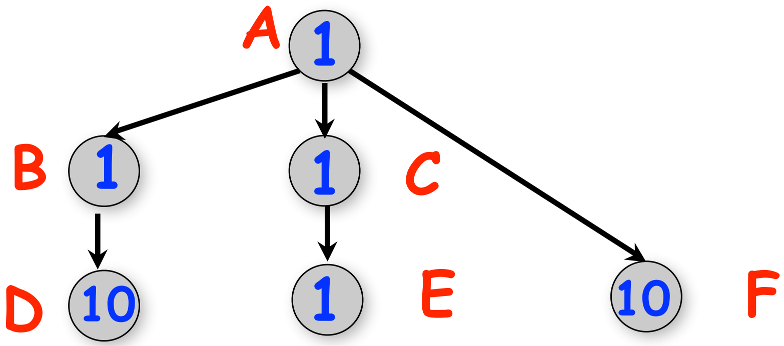
Scheduling Program Q1 using a Work-First Work-Stealing Scheduler



```
1. // Program Q1
2. A; // Executes on P1
3. finish {
4.   // P1 pushes continuation for 9,
5.   // and executes 6
6.   async { B; D; }
7.   // P2 pushes continuation for 11,
8.   // and executes 9
9.   async F;
10.   // P2 executes 11
11.  async { C; E; }
12. }
```

Start time	Proc 1	Proc 2
0	A	
1	B	F
2	D	F
3	D	F
4	D	F
5	D	F
6	D	F
7	D	F
8	D	F
9	D	F
10	D	F
11	D	C
12		E
13		

Scheduling Program Q1 using a Help-First Work-Stealing Scheduler

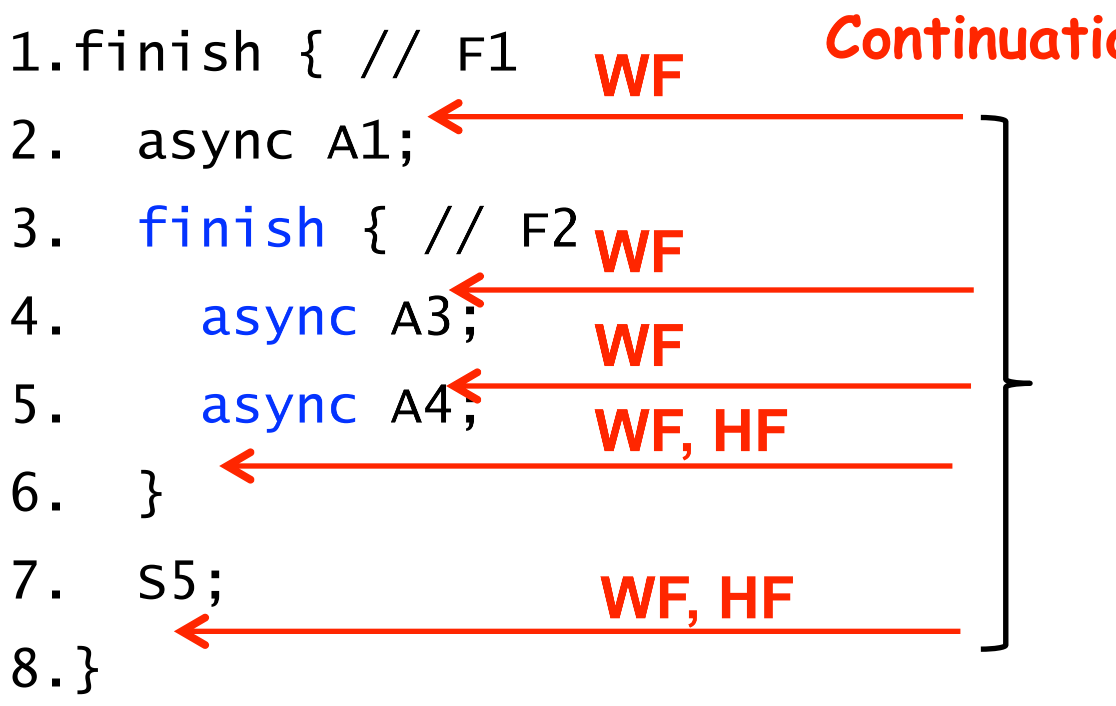


```
1. // Program Q1
2. A; // Executes on P1
3. finish {
4.   // P1 pushes 6, which is then
5.   // stolen by P2
6.   async { B; D; }
7.   // P1 pushes 8
8.   async F;
9.   // P1 pushes 10
10.  async { C; E; }
11. }
12. // P1 stores continuation and pops 10
13. // P1 pops 8
```

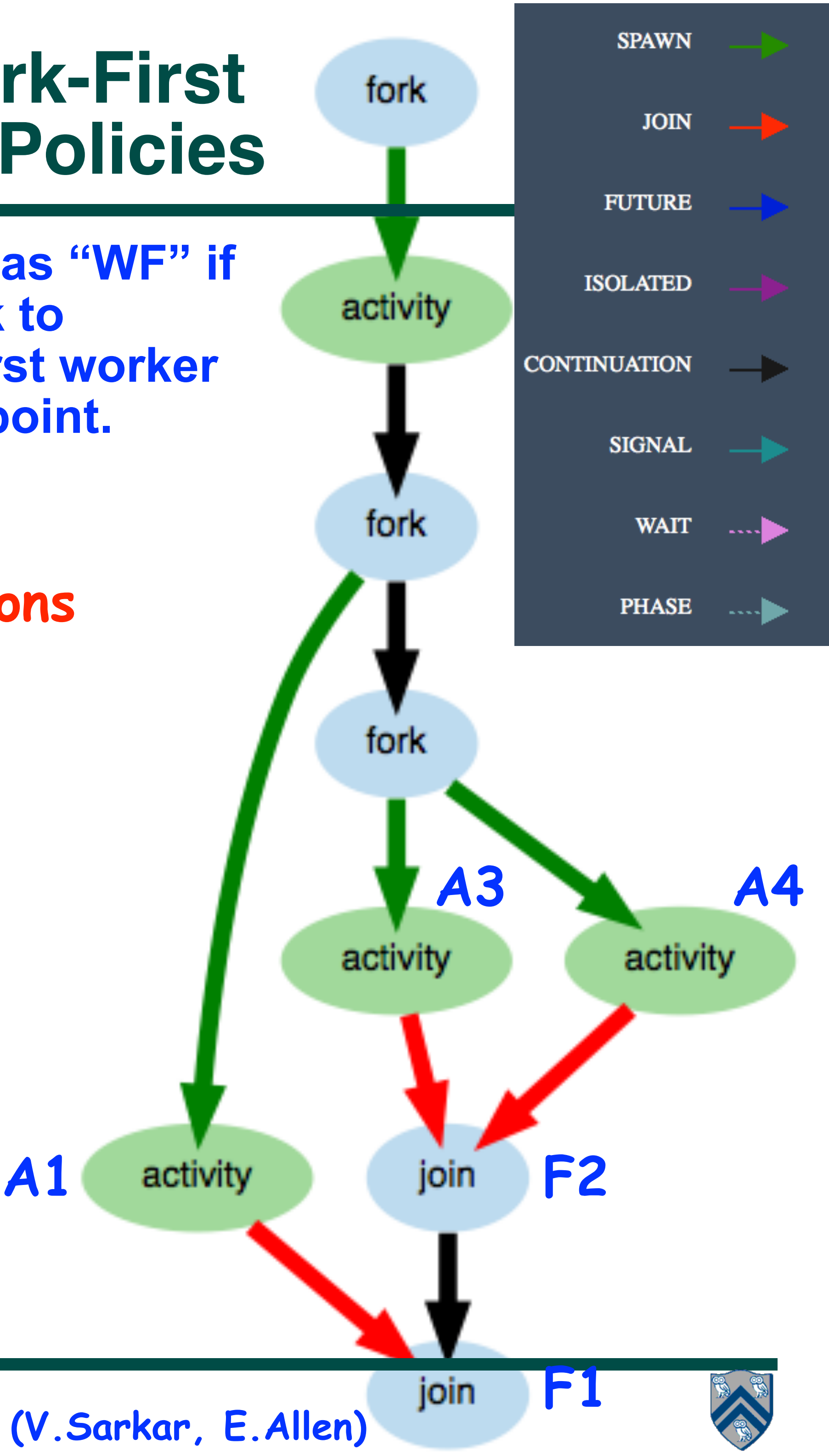
Start time	Proc 1	Proc 2
0	A	
1	C	B
2	E	D
3	F	D
4	F	D
5	F	D
6	F	D
7	F	D
8	F	D
9	F	D
10	F	D
11	F	D
12	F	
13		

Worksheet #15 solution: Work-First vs. Help-First Work-Stealing Policies

For each of the continuations below, label it as "WF" if a work-first worker can switch from one task to another at that point and as "HF" if a help-first worker can switch from one task to another at that point. Some continuations may have both labels.



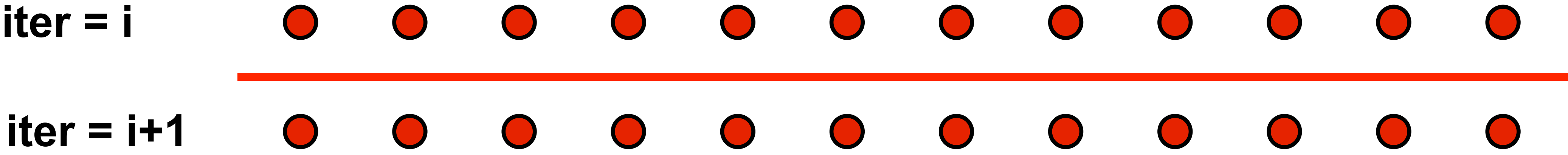
```
finish { // F1
  async A1;
  finish { // F2
    async A3;
    async A4;
  }
  S5;
}
```



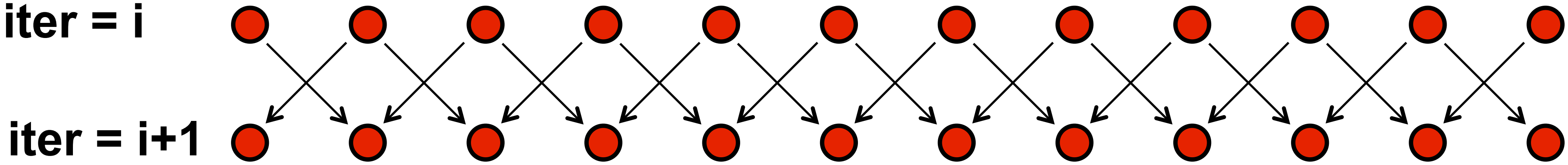
HJ code for One-Dimensional Iterative Averaging with forall-forseq structure and barriers (Recap from Lec 12)

```
double[] gVal=new double[n+2]; gVal[n+1] = 1;
double[] gNew=new double[n+2];
forallPhased(1, n, (j) -> { // Create n tasks
  // Initialize myVal and myNew as local pointers
  double[] myVal = gVal; double[] myNew = gNew;
  forseq(0, m-1, (iter) -> {
    // Compute MyNew as function of input array MyVal
    myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    next(); // Barrier before executing next iteration of iter loop
    // Swap local pointers, myVal and myNew
    double[] temp=myVal; myVal=myNew; myNew=temp;
    // myNew becomes input array for next iteration
  }); // forseq
}); // forall
```

Barrier vs Point-to-Point



Barrier synchronization



Point-to-point synchronization

Question: when 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

```
finish (() -> {
  ph = newPhaser(HjPhaserMode.SIG_WAIT); // mode is SIG_WAIT
  asyncPhased(ph.inMode(HjPhaserMode.SIG), () -> {
    // A1 (SIG mode)
    doA1Phase1(); next(); doA1Phase2(); });
  asyncPhased(ph.inMode(HjPhaserMode.DEFAULT_MODE), () -> {
    // A2 (default SIG_WAIT mode from parent)
    doA2Phase1(); next(); doA2Phase2(); });
  asyncPhased(ph.inMode(HjPhaserMode.DEFAULT_MODE), () -> {
    // A3 (default SIG_WAIT mode from parent)
    doA3Phase1(); next(); doA3Phase2(); });
  asyncPhased(ph.inMode(HjPhaserMode.WAIT), () -> {
    // A4 (WAIT mode)
    doA4Phase1(); next(); doA4Phase2(); });
});
```

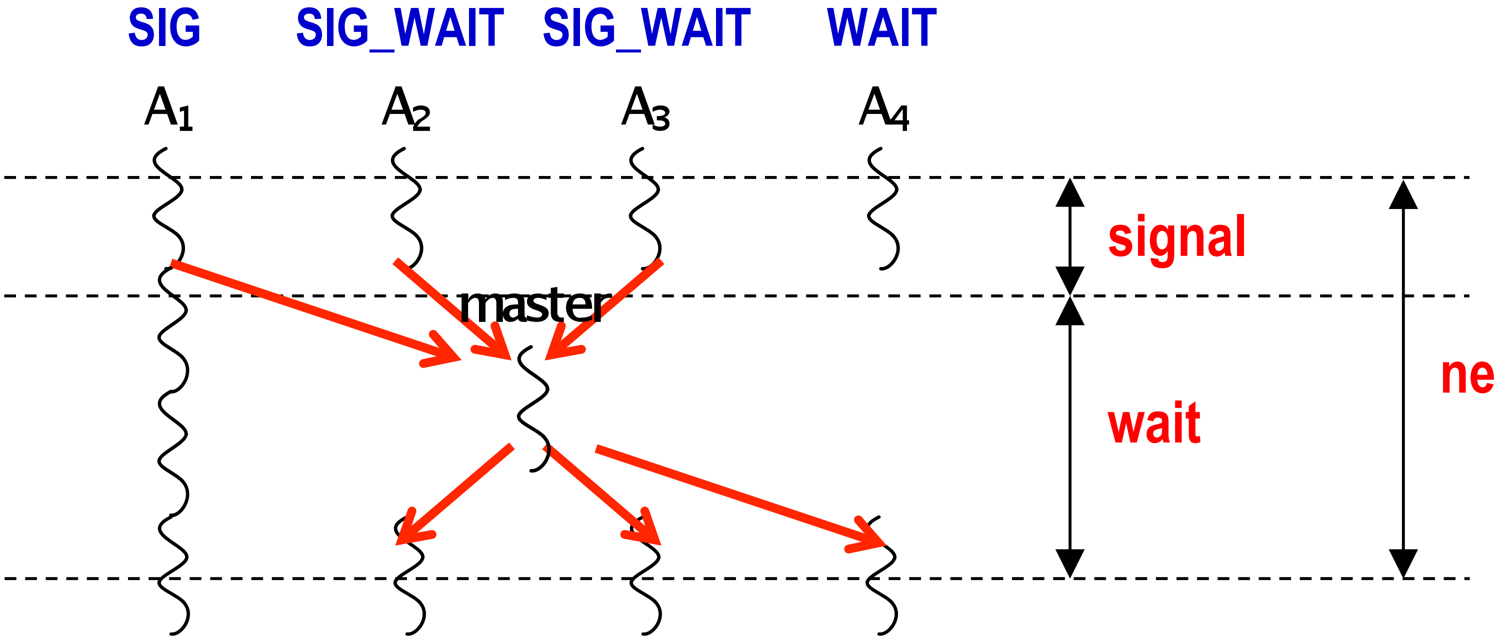
Simple Example with Four Async Tasks and One Phaser

Semantics of next depends on registration mode

SIG_WAIT: **next = signal + wait**

SIG: **next = signal**

WAIT: **next = wait**

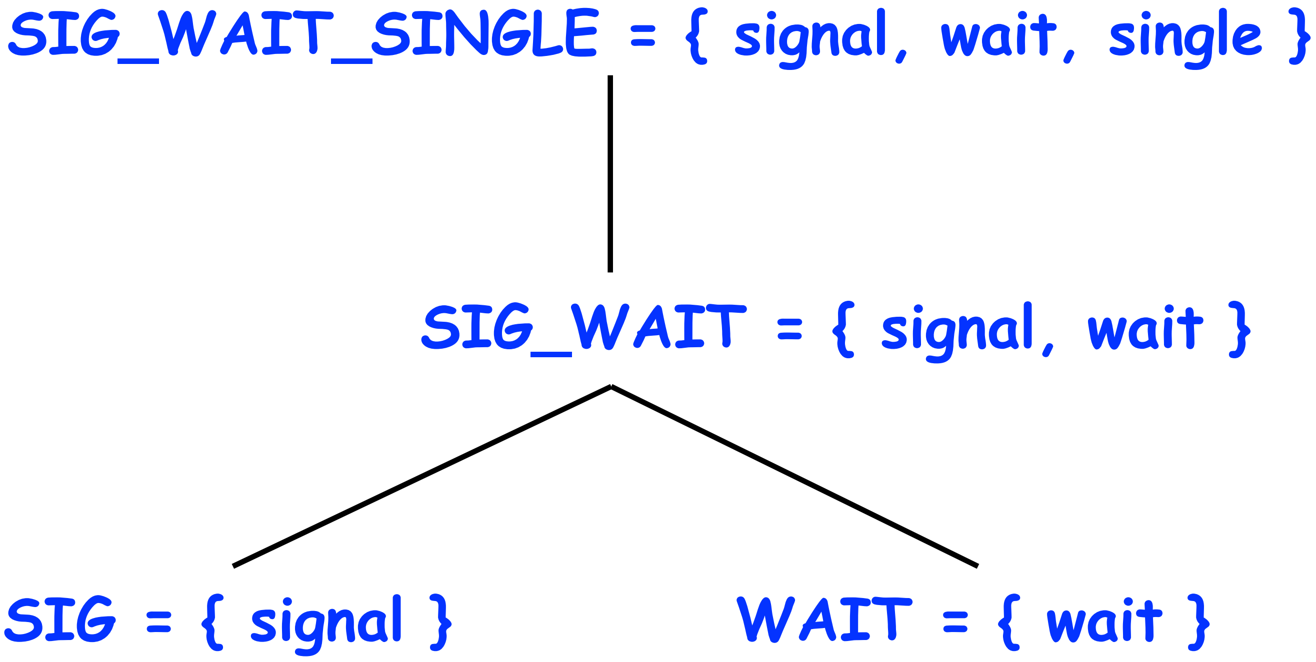


A master thread (worker) gathers all signals and broadcasts a barrier completion

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



SIG_WAIT_SINGLE = { signal, wait, single }

SIG_WAIT = { signal, wait }

SIG = { signal }

WAIT = { wait }

- 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.

forall barrier is just an implicit phaser!

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

is equivalent to

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

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