COMP 322: Fundamentals of Parallel Programming

Lecture 14: Recap of HJ constructs, Point-to-Point Synchronization, Pipeline Parallelism, Introduction to Phasers

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

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

Announcements

- Combined lecture quiz for this week (Week 5) and next week (Week 6) will be assigned on Thursday, Feb 14, and due by Sunday, Feb 17
- No labs or lab quizzes next week (Week 6)
- Homework 3 is due by by 11:55pm on Friday, February 22, 2013
- Take-home midterm (Exam 1) will be given after lecture on Wednesday, February 20, 2013
  - Closed-book, closed computer
  - 2-hour duration
  - Will need to be returned to Sherry Nassar (Duncan Hall 3137) by 4pm on Friday, February 22, 2013

Worksheet #13: Forall Loops and Barriers

For the example below, will reordering the five async statements change the meaning of the program? If so, show two orderings that exhibit different behaviors. If not, explain why not. (You can use the space below this slide for your answer.)

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
  async await(left) leftReader(left); // Task3
  async await(right) rightReader(right); // Task5
  async await(left,right)
        bothReader(left,right); // Task4
  async left.put(leftWriter()); // Task1
  async right.put(rightWriter());// Task2
}
```

No, reordering consecutive async's will never change the meaning of the program, whether or not the async's have await clauses.

Outline of Today's Lecture

- **<u>Recap of HJ constructs studied so far</u>**
- **Point-to-Point Synchronization, Pipeline Parallelism**
- **Introduction to Phasers**

Acknowledgments

- COMP 322 Module 1 handout, Sections 12.1, 12.2
- Knowing When to Parallelize: Rules-of-Thumb based on User Experiences. Cherri Pancake.
  - —Source for seismic imaging example
- Barry Wilkinson and Michael Allen. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition). Prentice-Hall, 2004.
  - —Source for figures related to pipeline parallelism

Recap of HJ constructs studied so far

- Basic language summary can be found here:
  - —https://wiki.rice.edu/confluence/display/PARPROG/HJLanguageSummary
  - —Additional documentation in preparation
- Task creation constructs
  - — async *Stmt*
  - — async<T> { *Stmt ;* return ...; }
  - — forasync (point[i,j] : ...) *Stmt*
  - — forall (point[i,j] : ...) *Stmt*
- Loop constructs
  - — point
  - — region
  - — for (point[i,j] : ...) *Stmt*

Recap of HJ constructs studied so far (contd)

- Synchronization constructs
  - finish
  - f.get()
  - finish accumulators
  - next
  - async await
- Efficiency constructs
  - Converting async to async seq
  - Loop chunking with GetChunk()
  - Converting futures to data-driven futures
- Abstract metrics
  - perf.doWork(n)

Outline of Today's Lecture

- **Recap of HJ constructs studied so far**
- **<u>Point-to-Point Synchronization, Pipeline Parallelism</u>**
- **Introduction to Phasers**

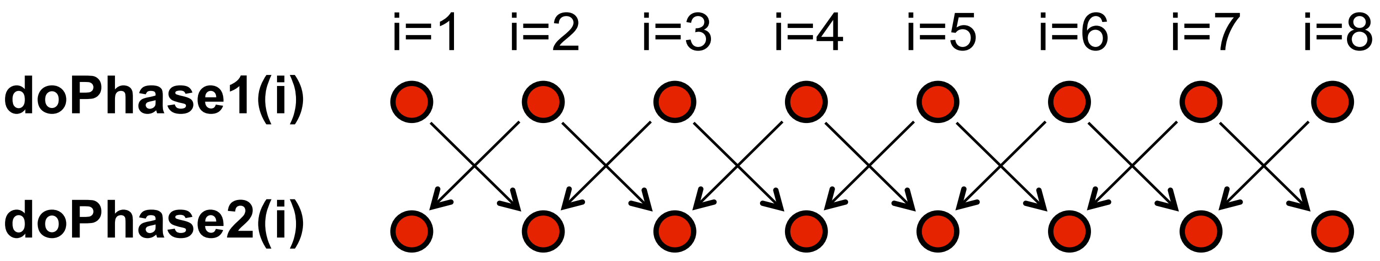
Acknowledgments

- COMP 322 Module 1 handout, Sections 12.1, 12.2
- Knowing When to Parallelize: Rules-of-Thumb based on User Experiences. Cherri Pancake.
  - —Source for seismic imaging example
- Barry Wilkinson and Michael Allen. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition). Prentice-Hall, 2004.
  - —Source for figures related to pipeline parallelism

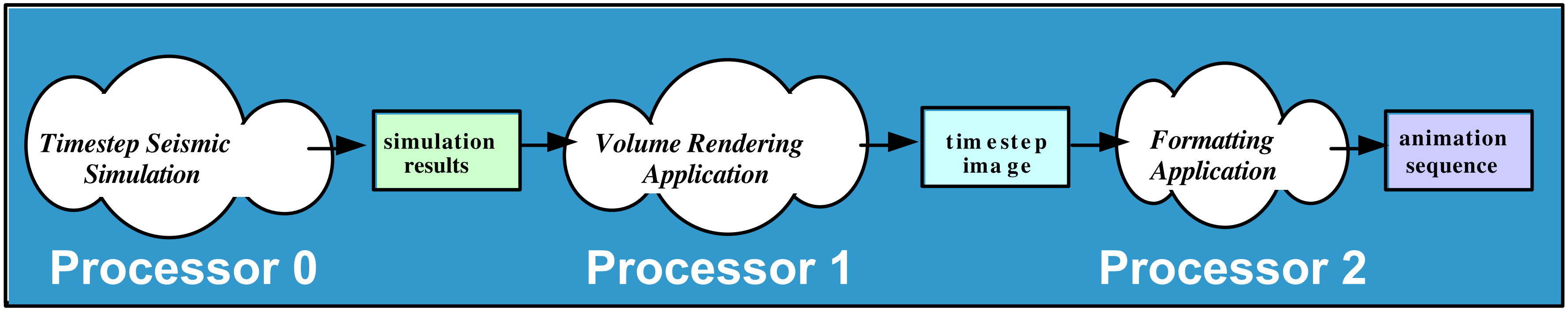
Point-to-Point Synchronization: Example 1

```
finish { // Expanded finish-forasync version of forall
 forasync (point[i] : [1:m]) {
  doPhase1(i);
  // Iteration i waits for i-1 and i+1 to complete Phase 1
  doPhase2(i);
 }
}
```

- **Need synchronization where iteration i only waits for iterations i−1 and i+1 to complete their work in doPhase1() before it starts doPhase2(i)**
  - **Less constrained than a barrier --- only waits for two preceding iterations**
  - **More general than async await --- waiting occurs in middle of task**

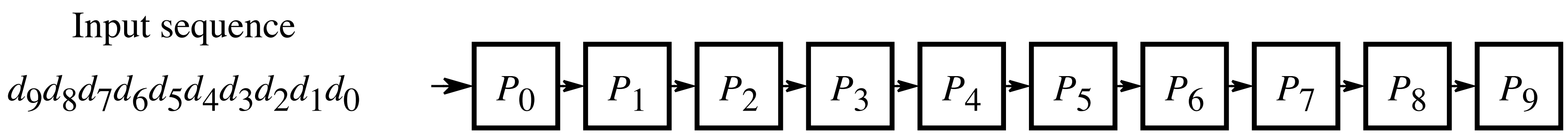


Point-to-point Synchronization: Example 2
Pipeline Parallelism



- Seismic imaging pipeline example with three stages
  1. Simulation generates a sequence of results, one per time step.
  2. Rendering takes simulation results for one time step as input, and generates an image for that time step.
  3. Formatting image as input and outputs it into an animation sequence.
- Even though the processing is sequential for a single time step, pipeline parallelism can be exploited via point-to-point synchronization between neighboring stages

General structure of a One-Dimensional Pipeline



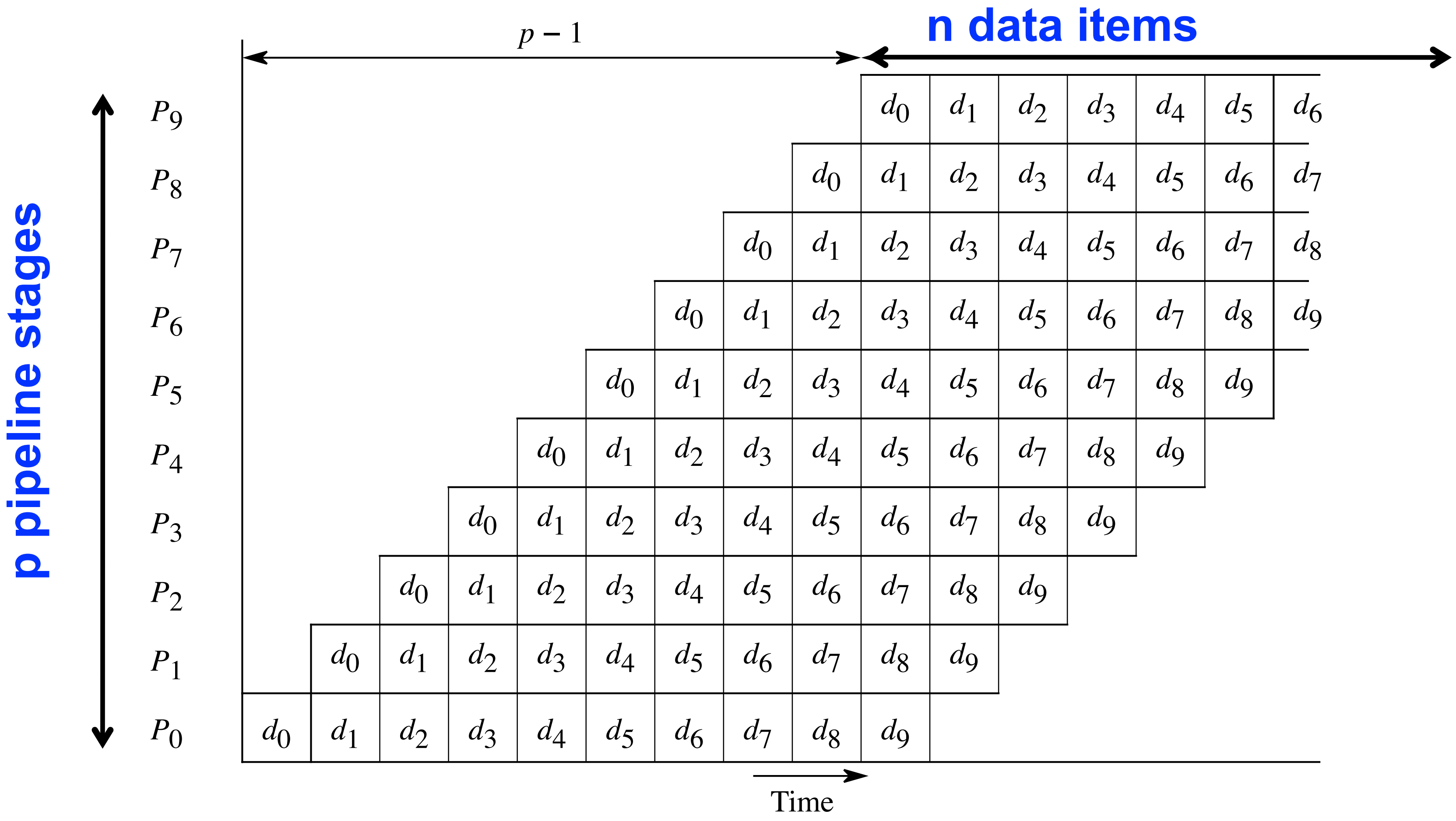
Input sequence

$d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0$

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



P_9										d_0	d_1	d_2	d_3	d_4	d_5	d_6
P_8									d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
P_7								d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
P_6							d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9
P_5						d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	
P_4					d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9		
P_3				d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9			
P_2			d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9				
P_1		d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9					
P_0	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9						

- 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 = $n \times p$ 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 = $p/(2 - 1/p) \approx p/2$
  - —$n \gg p$ ➔ PAR $\approx p$

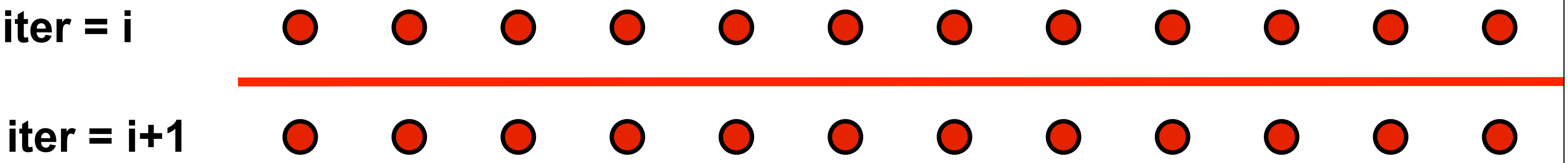
Outline of Today's Lecture

- **Recap of HJ constructs studied so far**
- **Point-to-Point Synchronization, Pipeline Parallelism**
- **<u>Introduction to Phasers</u>**

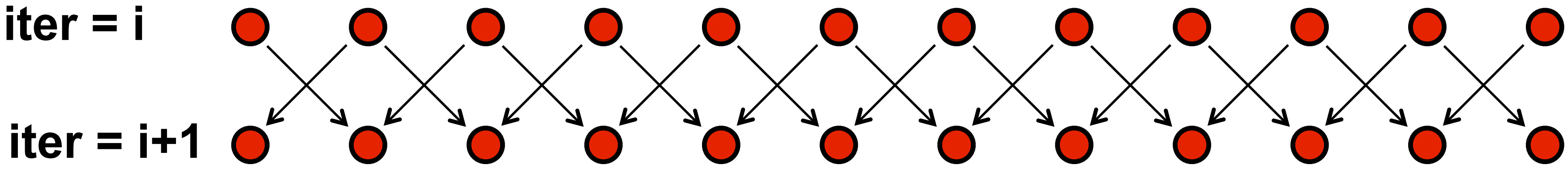
Acknowledgments

- COMP 322 Module 1 handout, Sections 12.1, 12.2
- Knowing When to Parallelize: Rules-of-Thumb based on User Experiences. Cherri Pancake.
  - —Source for seismic imaging example
- Barry Wilkinson and Michael Allen. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition). Prentice-Hall, 2004.
  - —Source for figures related to pipeline parallelism

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



Barrier synchronization



Point-to-point synchronization

(Left-right neighbor synchronization)

Phasers: a unified construct for barrier and point-to-point synchronization

- Previous examples motivated the need for point-to-point synchronization
- HJ phasers unify barriers with point-to-point synchronization
- A limited version of phasers was also added to the Java 7 java.util.concurrent.Phaser library (with acknowledgment to Rice)
- Phaser properties
  - —Barrier and point-to-point synchronization
  - —Supports dynamic parallelism i.e., the ability for tasks to drop phaser registrations on termination, and for new tasks to add new phaser registrations.
  - —Deadlock freedom
  - —Support for phaser accumulators (reductions that can be performed with phasers)

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 is unrelated to Java wait/notify (which we will study later)
- 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
    - Barrier is a special case of phaser, which is why next is used for both

Simple Example with Four Async Tasks and One Phaser (Listing 43)

```
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(); }
}
```

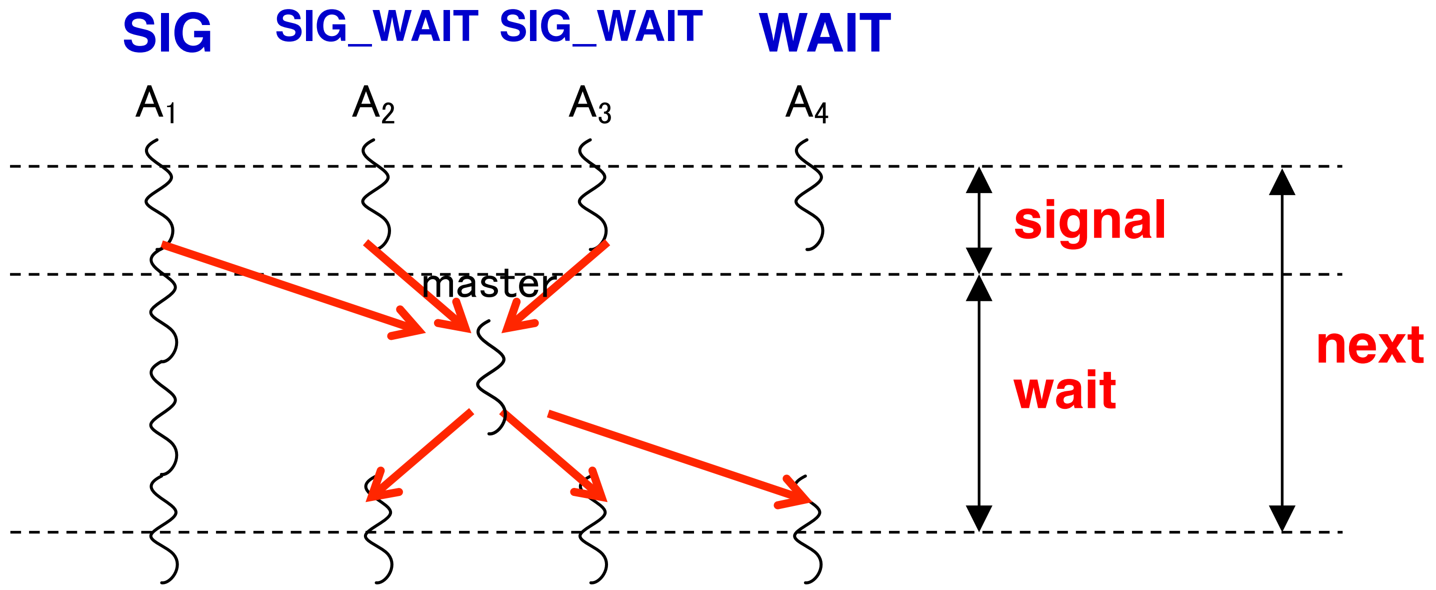
Simple Example with Four Async Tasks and One Phaser (contd)

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)


SIG

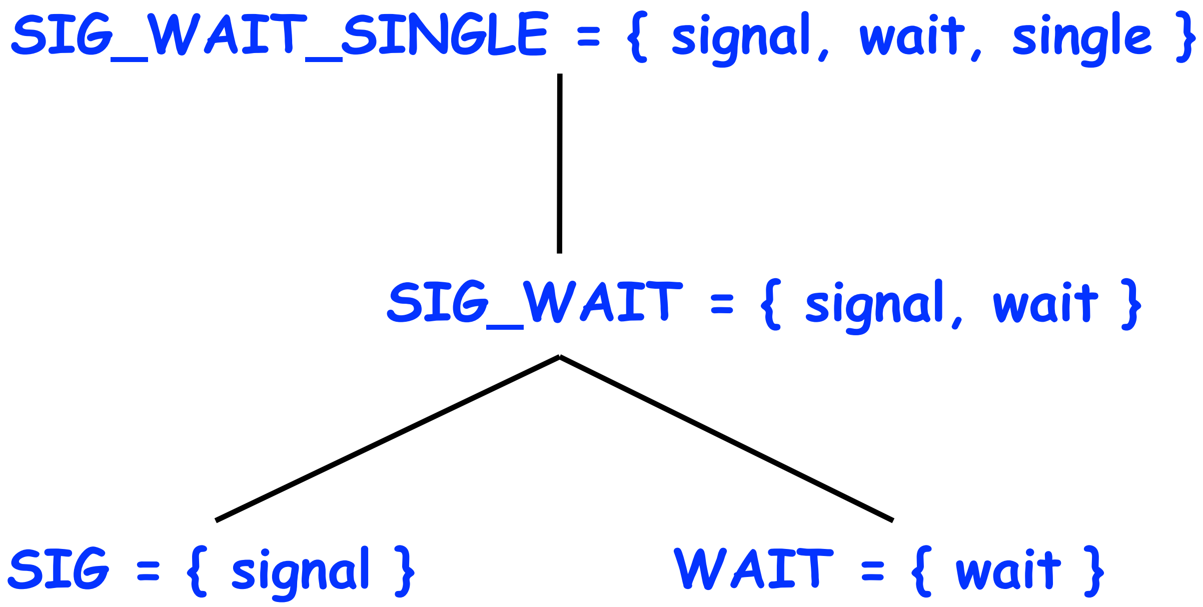
SIG_WAIT

SIG_WAIT

WAIT

A master task receives all signals and broadcasts a barrier completion

Capability Hierarchy



SIG_WAIT_SINGLE = { signal, wait, single }

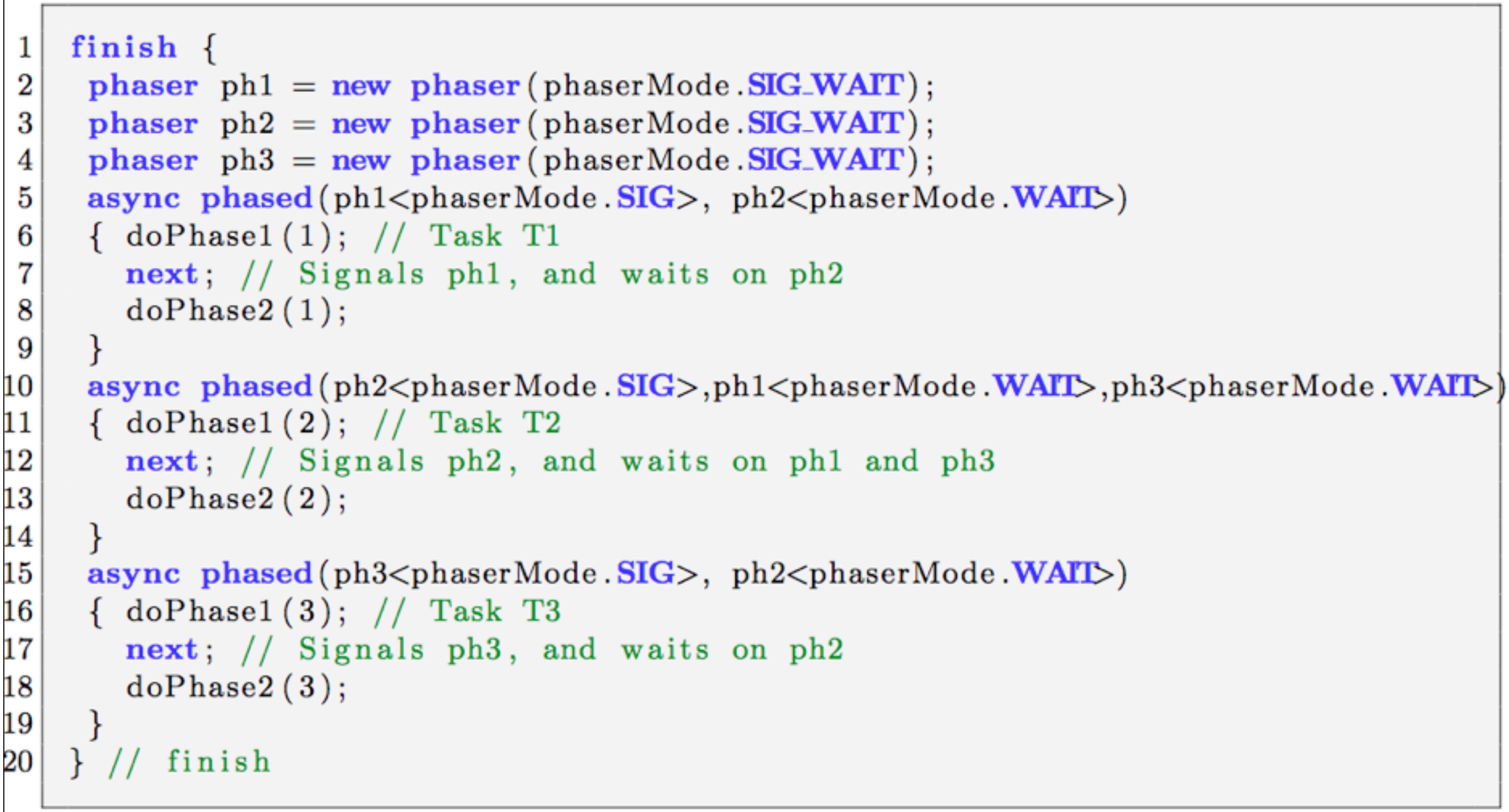
SIG_WAIT = { signal, wait }

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.**

Left-Right Neighbor Synchronization Example for m=3 (Listing 46)



```
finish {
 phaser ph1 = new phaser(phaserMode.SIG_WAIT);
 phaser ph2 = new phaser(phaserMode.SIG_WAIT);
 phaser ph3 = new phaser(phaserMode.SIG_WAIT);
 async phased(ph1<phaserMode.SIG>, ph2<phaserMode.WAIT>)
 { doPhase1(1); // Task T1
   next; // Signals ph1, and waits on ph2
   doPhase2(1);
 }
 async phased(ph2<phaserMode.SIG>,ph1<phaserMode.WAIT>,ph3<phaserMode.WAIT>)
 { doPhase1(2); // Task T2
   next; // Signals ph2, and waits on ph1 and ph3
   doPhase2(2);
 }
 async phased(ph3<phaserMode.SIG>, ph2<phaserMode.WAIT>)
 { doPhase1(3); // Task T3
   next; // Signals ph3, and waits on ph2
   doPhase2(3);
 }
} // finish
```

Listing 46: Example of left-right neighbor synchronization for $m = 3$ case