COMP 322: Fundamentals of Parallel Programming

Lecture 15: Point-to-Point Synchronization with Phasers

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Outline of Today’s Lecture

- **Point-to-Point Synchronization with Phasers**
- **Signal statement and split-phase barriers**

**Acknowledgments**
- COMP 322 Module 1 handout, Chapter 12
Phasers: a unified construct for barrier and point-to-point synchronization (Recap)

- HJ phasers unify barriers with point-to-point synchronization
- Examples in Lecture 14 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
  - Deadlock freedom --- a next operation will not lead to a situation where all active tasks are blocked indefinitely
  - Support for phaser accumulators --- reductions that can be performed with phasers
Simple Example with Four Async Tasks and One Phaser (Listing 43)

1. `finish`
   
2. `ph = new phaser();` // Default mode is SIG_WAIT
3. `async phased(ph<phaserMode.SIG>){` // A1 (SIG mode)
4.   `doA1Phase1(); next;`
5.   `doA1Phase2();` }
6. `async phased {` // A2 (default SIG_WAIT mode from parent)
7.   `doA2Phase1(); next;`
8.   `doA2Phase2();` }
9. `async phased {` // A3 (default SIG_WAIT mode from parent)
10. `doA3Phase1(); next;`
11. `doA3Phase2();` }
12. `async phased(ph<phaserMode.WAIT>){` // A4 (WAIT mode)
13.   `doA4Phase1(); next; doA4Phase2();` }
14. `}`
Simple Example with Four Async Tasks and One Phaser (Figure 48)

Semantics of \texttt{next} depends on registration mode

- SIG\_WAIT: \texttt{next} = \texttt{signal} + \texttt{wait}
- SIG: \texttt{next} = \texttt{signal}
- WAIT: \texttt{next} = \texttt{wait}

A master thread (worker) gathers all signals and broadcasts a barrier completion
Summary of Phaser Construct

- Phaser allocation
  - 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<sub>1</sub><mode<sub>1</sub>>, ph<sub>2</sub><mode<sub>2</sub>>, …) <stmt>
    - Spawned task is registered with ph<sub>1</sub> in mode<sub>1</sub>, ph<sub>2</sub> in mode<sub>2</sub>, …
    - 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
So, what is a phaser and how does it work?

- A phaser is a synchronization object --- you can allocate as many phasers as you choose, and also build arrays/collections of phasers.

- The task that allocates a phaser is automatically registered on the phaser in the mode specified in the constructor (SIG_WAIT is the default mode).

- A task can be registered on multiple phasers in different modes, specified in its “async phased” clause or due to its phaser allocations.

- A “next” operation performs all signal operations followed by all wait operations, according to the task’s phaser registrations.
  - Ordering of signal-wait avoids deadlock.
  - Degenerates gracefully when wait set or signal set is empty.

- A registration on phaser ph in mode m can only be included in “async phased” if the parent was also registered on ph with mode m (capability rule).

- Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF) for the allocation i.e., if phaser ph is allocated in finish scope F, then the task executing F must drop any registration that it has on ph when reaching the end-finish point for F.
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.

SIG_WAIT_SINGLE = \{ signal, wait, single \}

SIG_WAIT = \{ signal, wait \}

SIG = \{ signal \} \quad \text{WAIT} = \{ wait \}
Left-Right Neighbor Synchronization
Example for m=3 (Listing 46)

```java
finish { // Task T0
    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
Let's try another phaser example in Worksheet 15!
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\rightarrow i+1)`
- wait edges connect each phase transition node, `ph.next-end(i\rightarrow i+1)` to corresponding wait or next operations
- single edges connect each phase transition node, `ph.next-start(i\rightarrow 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\rightarrow i+1)`
Next-with-Single Statement (for SIG_WAIT_SINGLE registration mode)

next <single-stmt> is a barrier in which single-stmt is performed exactly once after all tasks have completed the previous phase and before any task begins its next phase.

Modeling next-with-single in the Computation Graph

A1 \rightarrow A2 \rightarrow A3 \rightarrow A4

signal edges

next-start

↓

single-statement

↓

next-end

wait edges

A1 \longrightarrow A2 \longrightarrow A3 \longrightarrow A4
One-Dimensional Iterative Averaging with Point-to-Point Synchronization (w/o chunking)

1. double[] gVal = new double[n+2]; double[] gNew = new double[n+2];
2. gVal[n+1] = 1; gNew[n+1] = 1;
3. phaser ph = new phaser[n+2];
4. finish { // phasers must be allocated in finish scope
5.   forall(point [i]:[0:n+1]) ph[i] = new phaser();
6.   forasync(point [j]:[1:n]) phased(ph[j]<phaserMode.SIG>,
7.     ph[j-1]<phaserMode.WAIT>,ph[j+1]<phaserMode.WAIT>){
8.     double[] myVal = gVal; double[] myNew = gNew; // Local pointers
9.     for (point [iter]: [0:numIters-1]) {
10.    myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
11.    next; // Point-to-point synchronization
12.    // Swap myVal and myNew
13.    double[] temp=myVal; myVal=myNew; myNew=temp;
14.    // myNew becomes input array for next iter
15.   } // for-iter
16. } // forasync-j
17. } // finish
forall barrier is just an implicit phaser

forall (point[i,j] : [iLo:iHi,jLo:jHi]) {
        S1; next; S2; next{...}
    }

is equivalent to

finish {
        // Implicit phaser for forall barrier
        phaser ph = new phaser(phaserMode.SIG_WAIT_SINGLE);
        forasync(point[i,j] : [iLo:iHi,jLo:jHi])
        phased(phaserMode.SIG_WAIT_SINGLE){
            S1; next; S2; next{...}
        } // next statements in async refer to ph
    }


Outline of Today’s Lecture

• Point-to-Point Synchronization with Phasers
• **Signal statement and split-phase barriers**

**Acknowledgments**

• COMP 322 Module 1 handout, Chapter 12
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 (Listing 50)

```java
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)

4 \rightarrow 5\text{-signal} \rightarrow 6 \rightarrow 7\text{-wait} \rightarrow 8

\text{spawn} \quad \text{continue} \quad \text{signal} \quad \text{wait} \quad \text{join
Full Computation Graph for Split-Phase Barrier Example (Figure 52)

2 \rightarrow 4 \rightarrow 5\text{-signal} \rightarrow 6 \rightarrow 7\text{-wait} \rightarrow 8

\text{ph.next} \rightarrow \text{start}(0 \rightarrow 1) \rightarrow 20\text{-drop} \rightarrow 20\text{-end-finish}

\text{ph.next} \rightarrow \text{end}(0 \rightarrow 1)

11 \rightarrow 12\text{-signal} \rightarrow 13 \rightarrow 14\text{-wait} \rightarrow 15

\text{spawn} \rightarrow \text{continue} \rightarrow \text{signal} \rightarrow \text{wait} \rightarrow \text{join}
Complete the phased clause below to implement the left-right neighbor synchronization shown above:

1. `finish` {
2.   `phaser[] ph = new phaser[m+2];` // array of phaser objects
3.   `for(point [i]:[0:m+1]) ph[i] = new phaser();`
4.   `for(point [i]:[1:m])`
5.   `async phased(ph[i-1]<...>, ph[i+1]<...>, ph[i]<...>) {
6.     doPhase1(i);
7.     `next;`
8.     doPhase2(i);
9.   }
10.}

Name 1: ___________________
Name 2: ___________________