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
  - Deterministic 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_SINGLE
    - signal statement can be used to support “fuzzy” barriers
    - phaser accumulators can perform per-phasor 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
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

1. `finish {}`
2. `ph = new phaser(); // Default mode is SIG_WAIT`
3. `async phased(ph<phaserMode.SIG>){ //A1 (SIG mode)`
4. 
5. `doA1Phase1(); next;`
6. `doA1Phase2(); }`
7. `async phased { //A2 (default SIG_WAIT mode from parent)`
8. 
9. `doA2Phase1(); next;`
10. `doA2Phase2(); }`
11. `async phased { //A3 (default SIG_WAIT mode from parent)`
12. 
13. `doA3Phase1(); next;`
14. `doA3Phase2(); }`
15. `async phased(ph<phaserMode.WAIT>){ //A4 (WAIT mode)`
16. 
17. `doA4Phase1(); next; doA4Phase2(); }`
18. `}`
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
AtomicInteger rank = new AtomicInteger();

forall (point[i] : [0:m-1]) {
    int r = rank.getAndIncrement();
    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);
}

• 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

```java
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.

$i=0$...$7$ are forall iterations

$(i,j) =$ println output

next = barrier operation

end = termination of a forall iteration
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

1. AtomicInteger rank = new AtomicInteger();
2. forall (point[i] : [0:m-1]) {
3.   // Start of Hello phase
4.   int r = rank.getAndIncrement();
5.   System.out.println("Hello from task ranked " + r);
6.   next single {
7.     System.out.println("LOG: Between Hello & Goodbye Phases");
8.   }
9.   // Start of Goodbye phase
10.  System.out.println("Goodbye from task ranked " + r);
11. } // 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

```java
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

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

\[ 6 \rightarrow 7\text{-signal} \rightarrow 7\text{-wait} \rightarrow 8 \]

\[ 11 \rightarrow 12\text{-signal} \rightarrow 12\text{-wait} \rightarrow 13 \]

\[ 16 \rightarrow 17\text{-signal} \rightarrow 17\text{-wait} \rightarrow 18 \]

- **ph1.next**
  - start\((0\rightarrow1)\)
  - end\((0\rightarrow1)\)

- **ph2.next**
  - start\((0\rightarrow1)\)
  - end\((0\rightarrow1)\)

- **ph3.next**
  - start\((0\rightarrow1)\)
  - end\((0\rightarrow1)\)
Full Computation Graph for m=3 example

1, 2, 3, 4 → 20-drop → 20-end-finish

6 → 7-signal

ph1.next
-start(0→1)

ph1.next
-end(0→1)

11 → 12-signal

ph2.next
-start(0→1)

ph2.next
-end(0→1)

16 → 17-signal

ph3.next
-start(0→1)

ph3.next
-end(0→1)

7-signal → 7-wait → 8

12-signal → 12-wait → 13

17-signal → 17-wait → 18

spawn
continue
signal
wait
join
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

```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 → 5-signal → 6 → 7-wait → 8

11 → 12-signal → 13 → 14-wait → 15

spawn → continue → signal → wait → join
Full Computation Graph for Split-Phase Barrier Example

Diagram showing a computation graph with nodes labeled 2, 4, 5, 6, 7, 8, 11, 12, 13, 14, and 15, connected by edges labeled with actions such as "spawn", "continue", "signal", "wait", and "join".
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