COMP 322: Fundamentals of Parallel Programming

Lecture 14: Unification of Barrier and 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
Barrier vs Point-to-Point Synchronization for One-Dimensional Iterative Averaging Example

Barrier synchronization

Point-to-point synchronization
Left-Right Neighbor Synchronization Example

1. `finish` { // Expanded `finish-for-async` version of `forall`

2. `for` (point[i] : [1:m])

3. `async` {

4. `doPhase1(i);`

5. `// Iteration i waits for i-1 and i+1 to complete Phase 1

5. `doPhase2(i);`

6. } // async

7. } // finish

• 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)
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.Phaseer` 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 has no relationship to Java wait/notify

- 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
Capability Hierarchy

SIG_WAIT_SINGLE = \{ \text{signal, wait, single} \}

SIG_WAIT = \{ \text{signal, wait} \}

SIG = \{ \text{signal} \} \quad \text{WAIT} = \{ \text{wait} \}

- 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. `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 (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
forall barrier is just an implicit phaser

1. forall (point[i,j] : [iLo:iHi,jLo:jHi])
2. <body>

is equivalent to

3. finish {
4.   // Implicit phaser
5.   phaser ph = new phaser(phaserMode.SIG_WAIT);
6.   for((point[i,j] : [iLo:iHi,jLo:jHi])
7.       async phased(phaserMode.SIG_WAIT)
8.   <body> // next statements refer to ph
9. }
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.}

```
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.}
```
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);
    }
}
```
Computation Graph for $m=3$ example (without async/finish nodes and edges)

```
6  7-signal  7-wait  8
  |                  |
  | ph1.next         | ph1.next         |
  | -start(0→1)     | -end(0→1)       |
  +-----------------+-----------------+
11  12-signal  12-wait  13
  |                  |
  | ph2.next         | ph2.next         |
  | -start(0→1)     | -end(0→1)       |
  +-----------------+-----------------+
16  17-signal  17-wait  18
  |                  |
  | ph3.next         | ph3.next         |
  | -start(0→1)     | -end(0→1)       |
  +-----------------+-----------------+
```

\[ \text{spawn} \quad \text{continue} \quad \text{signal} \quad \text{wait} \quad \text{join} \]
Adding Phaser Operations to the Computation Graph

\(CG\) node = step

Step boundaries are induced by continuation points

- \textit{async}: source of a spawn edge
- \textit{end-finish}: destination of join edges
- \textit{future.get()}: destination of a join edge
- \textit{signal, drop}: source of signal edges
- \textit{wait}: destination of wait edges
- \textit{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

- \(ph.next\text{-start}(i \rightarrow i+1)\) and \(ph.next\text{-end}(i \rightarrow i+1)\)
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)`
Full Computation Graph for m=3 example

1,2,3,4

6

7-signal

ph1.next
-start(0→1)

ph1.next
-end(0→1)

ph2.next
-start(0→1)

ph2.next
-end(0→1)

11

12-signal

ph3.next
-start(0→1)

ph3.next
-end(0→1)

16

17-signal

12-drop

20-end-finish

8

13

18

spawn

continue

signal

wait

join
One-Dimensional Iterative Averaging with Point-to-Point Synchronization (compare with slide 9, Lecture 13)

1. double[] gVal = new double[n+2]; double[] gNew = new double[n+2];
2. gVal[n+1] = 1; gNew[n+1] = 1;
3. int Cj = Runtime.getRuntime().getNumOfWorkers();
4. finish {
5.     phaser ph = new phaser[Cj+2];
6.     for(point [i]:[0:Cj+1]) ph[i] = new phaser();
7.     for(point [jj]:[0:Cj-1])
9.             double[] myVal = gVal; double[] myNew = gNew; // Local copy of pointers
10.            for (point [iter] : [0:numIters-1]) {
11.                for (point [j]:getChunk([1:n],[Cj],[jj])) // Iterate within chunk
12.                    myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
13.                next; // Point-to-point synchronization
14.            } // for
15.        } // async
16.    } // finish
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 \rightarrow 5\text{-signal} \rightarrow 6 \rightarrow 7\text{-wait} \rightarrow 8

\begin{align*}
\text{ph.next} & \quad \text{ph.next} \\
\text{-start}(0 \rightarrow 1) & \quad \text{-end}(0 \rightarrow 1)
\end{align*}

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

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

2 \rightarrow 20\text{-drop} \rightarrow 20\text{-end-finish}

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

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

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

spawn \quad \text{continue} \quad \text{signal} \quad \text{wait} \quad \text{join}
1. double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
2. gVal[n+1] = 1; gNew[n+1] = 1;
3. int Cj = Runtime.getRuntime().getAvailableProcessors();
4. finish {
5.     phaser ph = new Phaser[Cj+2];
6.     for(point [i]:[0:Cj+1]) ph[i] = new Phaser();
7.     for(point [jj]:[0:Cj-1])
9.             double[] myVal = gVal; double[] myNew = gNew; // Local copy of pointers
10.            for (point [iter] : [0:numIters-1]) {
11.                region r = getChunk([1:n],[Cj],[jj]); int lo = r.rank(0).low(); int hi = r.rank(0).high();
12.                myNew[lo] = (myVal[lo-1] + myVal[lo+1])/2.0; myNew[hi] = (myVal[hi-1] + myVal[hi+1])/2.0;
13.                signal; // signal ph[jj+1]
14.                for (point [j]: [lo+1:hi-1]) // Iterate within chunk
15.                    myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
16.                next; // wait on ph[jj] and ph[jj+2]
17.            } // for
18.        } // finish

Announcements

• 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

• 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