No, reordering consecutive async’s will never change the meaning of the program, whether or not the async’s have await clauses.
What is Deadlock?

- A parallel program execution contains a deadlock if some task’s execution remains incomplete due to it being *blocked indefinitely* awaiting some condition.

- Example of a program with a deadlocking execution:

  ```java
  DataDrivenFuture left = new DataDrivenFuture();
  DataDrivenFuture right = new DataDrivenFuture();
  finish {
    async await ( left ) right.put(rightBuilder()); // Task1
    async await ( right ) left.put(leftBuilder()); // Task2
  }
  
  In this case, Task1 and Task2 are in a deadlock cycle.

- Can you create a deadlock example using any other blocking constructs that you’ve learned thus far in COMP 322?
  - e.g., end-finish, future-get, next (barrier)?
Question: when can the point-to-point computation graph result in a smaller CPL than the barrier computation graph?
Answer: when there is variability in the node execution times.
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.Phase*

- 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

1. `finish () -> {
2. ph = newPhaser(SIG_WAIT); // mode is SIG_WAIT
3. asyncPhased(ph.inMode(SIG), () -> {
4.     // A1 (SIG mode)
5.         doA1Phase1(); next(); doA1Phase2(); });
6. asyncPhased(ph.inMode(SIG_WAIT), () -> {
7.     // A2 (SIG_WAIT mode)
8.         doA2Phase1(); next(); doA2Phase2(); });
9. asyncPhased(ph.inMode(HjPhaserMode.SIG_WAIT), () -> {
10.    // A3 (SIG_WAIT mode)
11.        doA3Phase1(); next(); doA3Phase2(); });
12. asyncPhased(ph.inMode(HjPhaserMode.WAIT), () -> {
13.    // A4 (WAIT mode)
14.        doA4Phase1(); next(); doA4Phase2(); });
15. });`
Semantics of \texttt{next} depends on registration mode

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

\texttt{SIG}: \texttt{next} = \texttt{signal}

\texttt{WAIT}: \texttt{next} = \texttt{wait}
Summary of Phaser Construct

- Phaser allocation
  - \texttt{HjPhaser ph = newPhaser(mode);}
    - Phaser \texttt{ph} is allocated with registration mode
    - Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)

- Registration Modes
  - \texttt{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
  - \texttt{asyncPhased (ph\textsubscript{1}.inMode(<mode\textsubscript{1}>), ph\textsubscript{2}.inMode(<mode\textsubscript{2}>), \ldots () -> <stmt> )}
    - Spawned task is registered with \texttt{ph\textsubscript{1}} in \texttt{mode\textsubscript{1}}, \texttt{ph\textsubscript{2}} in \texttt{mode\textsubscript{2}}, \ldots
    - Child task’s capabilities must be \textit{subset} of parent’s
    - \texttt{asyncPhased <stmt>} propagates all of parent’s phaser registrations to child

- Synchronization
  - \texttt{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 \texttt{next} is used for both
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.

\[
\text{SIG\_WAIT\_SINGLE} = \{ \text{signal, wait, single} \} \\
\text{SIG\_WAIT} = \{ \text{signal, wait} \} \\
\text{SIG} = \{ \text{signal} \} \\
\text{WAIT} = \{ \text{wait} \}
\]
forall barrier is just an implicit phaser!

1. `forallPhased(iLo, iHi, (i) -> {`
2. `s1; next(); s2; next();{...}
3. `});

is equivalent to

1. `finish(() -> {
2. `// Implicit phaser for forall barrier
3. `final HjPhaser ph = newPhaser(SIG_WAIT);
4. `forseq(iLo, iHi, (i) -> {
5. `asyncPhased(ph.inMode(SIG_WAIT), () -> {
6. `s1; next(); s2; next();{...}
7. `}); // next statements in async refer to ph
8. `});
Left-Right Neighbor Synchronization (with m=3 tasks)

1. `finish()` -> { // Task-0
2.     final HjPhaser ph1 = newPhaser(SIG_WAIT);
3.     final HjPhaser ph2 = newPhaser(SIG_WAIT);
4.     final HjPhaser ph3 = newPhaser(SIG_WAIT);
5.     asyncPhased(ph1.inMode(SIG),ph2.inMode(WAIT),
6.             () -> { doPhase1(1);
7.             next(); // signals ph1, waits on ph2
8.             doPhase2(1);
9.         }); // Task T1
10.    asyncPhased(ph2.inMode(SIG),ph1.inMode(WAIT),ph3.inMode(WAIT),
11.        () -> { doPhase1(2);
12.        next(); // signals ph2, waits on ph3
13.        doPhase2(2);
14.    }); // Task T2
15.    asyncPhased(ph3.inMode(SIG),ph2.inMode(WAIT),
16.        () -> { doPhase1(3);
17.        next(); // signals ph3, waits on ph2
18.        doPhase2(3);
19.    }); // Task T3
20.}); // finish
Computation Graph for m=3 example (without async-finish nodes and edges)

- 6 → 7-signal → 7-wait → 8
- 11 → 12-signal → 12-wait → 13
- 16 → 17-signal → 17-wait → 18

- ph1.next
  - start(0→1) → ph1.next
  - end(0→1)

- ph2.next
  - start(0→1) → ph2.next
  - end(0→1)

- ph3.next
  - start(0→1) → ph3.next
  - end(0→1)

- continue
- signal
- wait