COMP 322: Fundamentals of Parallel Programming

Lecture 18: Phaser-specific Next Operations, Classification of Parallel Programs

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

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

Solution to Worksheet #17: Critical Path Length for Computation with

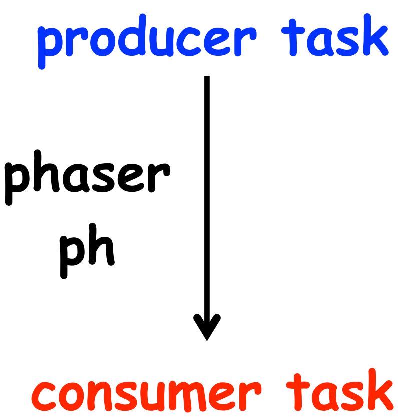
Compute the WORK and CPL values for the program shown below. (WORK = 204, CPL = 102). How would they be different if the signal() statement was removed? (CPL would increase to 202.)

```
finish(() -> {
  final HjPhaser ph = newPhaser(SIG_WAIT);
  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T1
    A(0); doWork(1);   // Shared work in phase 0
    signal();
    B(0); doWork(100); // Local work in phase 0
    next(); // Wait for T2 to complete shared work in phase 0
    C(0); doWork(1);
  });
  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T2
    A(1); doWork(1);   // Shared work in phase 0
    next(); // Wait for T1 to complete shared work in phase 0
    C(1); doWork(1);
    D(1); doWork(100); // Local work in phase 0
  });
}); // finish
```

Recap of Producer-Consumer pattern with phasers (used for implementing pipeline parallelism)

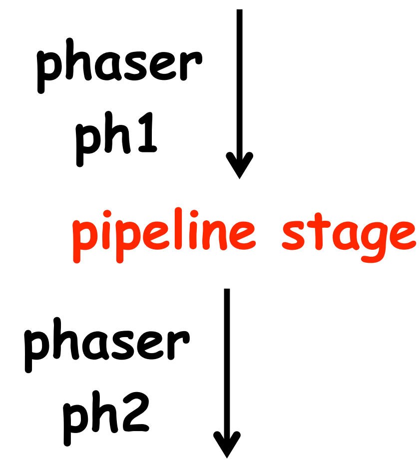
```
asyncPhased(ph.inMode(SIG), () -> {
    for (int i = 0; i < rounds; i++) {
        buffer.insert(…);
        // producer can go ahead as they are in SIG mode
        next();
    }
});

asyncPhased(ph.inMode(WAIT), () -> {
    for (int i = 0; i < rounds; i++) {
        next();
        buffer.remove(…);
    }
});
```



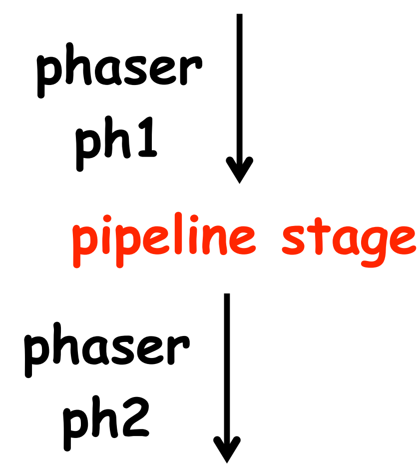
How to implement a pipeline stage that waits on one phaser and signals another?

```
asyncPhased(ph1.inMode(WAIT),
            ph2.inMode(SIG), () -> {
  for (int i = 0; i < rounds; i++) {
       // If we add next() here to wait on
       // ph1, it will also signal ph2.
       // Need an phaser-specific next!
       x = buffer1.remove();
       y = f(x);
      buffer2.insert();
  }
});
```



Implementing a pipeline stage with phaser-specific doNext() operations

```
asyncPhased(ph1.inMode(WAIT),
                ph2.inMode(SIG), () -> {
  for (int i = 0; i < rounds; i++) {
        // Wait-only operation on ph1
        ph1.doNext();
        x = buffer1.remove();
        y = f(x);
        buffer2.insert();
        // Signal-only operation on ph2
        ph2.doNext();
  }
});
```



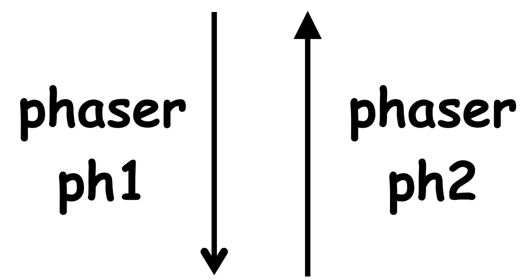
next() vs. phaser-specific doNext()

- next()
  - General operation that first signals all phasers that current task is registered on in signal or signal-wait mode, and then waits on all phasers that current task is registered on in wait or signal-wait mode
- ph.doNext()
  - Performs a next operation restricted to a specific phaser
  - First signals ph, if current task is registered on ph in signal or signal-wait mode
  - Then waits on ph, if current task is registered on ph in wait or signal-wait mode

Phaser-specific next operations can lead to deadlock, if used incorrectly

```
asyncPhased(ph1.inMode(SIG), ph2.inMode(WAIT),() -> {
      for (int i = 0; i < rounds; i++) {
          ph2.doNext(); // waits on ph2
           . . .
           ph1.doNext(); // signals ph1
      }
 });

asyncPhased(ph2.inMode(SIG), ph1.inMode(WAIT), () -> {
      for (int i = 0; i < rounds; i++) {
           ph1.doNext(); // waits on ph1
           . . .
           ph2.doNext(); // signals ph2
      }
 });
```



Summary of Parallel Programming Constructs you've learned so far

- Task Parallelism (Unit 1)
  - —Async (task creation)
  - —Finish (structured task termination)
- Functional Parallelism (Unit 2)
  - —Future (task creation)
  - —Future get() (task termination with return value)
  - —Accumulators (functional reduction)
  - —Map-Reduce (functional parallelism & reduction on key-value pairs)
- Loop Parallelism (Unit 3)
  - —Forall (parallel loops)
  - —Barriers (all-to-all synchronization)
- Dataflow Parallelism (Unit 4)
  - —Data-Driven Tasks (dataflow parallelism)
  - —Phasers (point-to-point synchronization)
  - —Phaser-specific next operations

Semantic Property #1: Serializability

- **Also referred to as "serial elision" property**
  - A parallel program, P, satisfies the serial elision property if removing all parallel constructs results in a serial program, S, that represents a legal execution of program P
- **Constructs that satisfy the serial elision property**
  - Async (task creation)
  - Finish (structured task termination)
  - Future (task creation)
  - Future get() (task termination with return value)
  - Accumulators (functional reduction)
  - Map-Reduce (functional parallelism & reduction on key-value pairs)
  - Forall without barriers (parallel loops)

Example of a parallel program that satisfies the serial elision property

```
finish { // F1
  async A; // Boil pasta (20)
  finish { // F2
    async B1; // Chop veggies (5)
    async B2; // Brown meat (10)
  } // F2
  B3; // Make pasta sauce (10)
} // F1
```

Step B1

Step B2

Step B3

Step A

Example of a parallel program that does not satisfy the serial elision property

```
forallPhased (0, m - 1, (i) -> {
  int sq = i*i;
  System.out.println("Hello from task with square = " + sq);
  next(); // Barrier
  System.out.println("Goodbye from task with square = " + sq);
});
```

Why does this program violate the serial elision property?

Semantic Property #2: Deadlock Freedom

- A parallel program, P, satisfies the deadlock freedom property if no execution of the program can reach a state in which one or more tasks are permanently blocked/suspended
- Constructs that satisfy the deadlock freedom property
  - —Async (task creation)
  - —Finish (structured task termination)
  - —Future (task creation)
  - —Future get() (task termination with return value)
  - —Accumulators (functional reduction)
  - —Map-Reduce (functional parallelism & reduction on key-value pairs)
  - —Forall (parallel loops)
  - —Barriers (all-to-all synchronization)
  - —Phasers without phaser-specific next operations

Example of a parallel program that does not satisfy the deadlock-freedom property

```
HjDataDrivenFuture left = newDataDrivenFuture();
HjDataDrivenFuture right = newDataDrivenFuture();
finish(() -> {
  asyncAwait(left, () -> {
      right.put(rightWriter()); });
  asyncAwait(right, () -> {
      left.put(leftWriter()); });
});
```

Why does this program violate the deadlock-freedom property?

Semantic Property #3: Data Race Freedom

- A parallel program, P, satisfies the data race freedom property if no execution of the program can exhibit a data race
- In general, can only be guaranteed in very special cases e.g.,
  - —Shared data that is allocated in futures or data-driven futures
  - —Shared data this is immutable e.g., like Java strings
  - —Shared data for which all steps that read or write it are totally ordered in the computation graph (includes case of “ownership” transfer from one task to another)

Example of a Data Race

```
1.  // Start of Task T0 (main program)
2.  sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
3.  async { // Task T0 computes sum of lower half of array
4.    for(int i=0; i < X.length/2; i++)
5.      sum1 += X[i];
6.  }
7.  async { // Task T1 computes sum of upper half of array
8.    for(int i=X.length/2; i < X.length; i++)
9.      sum2 += X[i];
10. }
11. // Task T0 waits for Task T1 (join)
12. return sum1 + sum2;
```

Data race between accesses of sum1 in lines 5 and 12, and accesses of sum2 in lines 9 and 12

Semantic Property #4: Functional and Structural Determinism

- A parallel program is said to be *functionally deterministic* if it always computes the same answer when given the same input
- A parallel program is said to be *structurally deterministic* if it always produces the same computation graph when given the same input
- In general, functional and structural determinism can only be guaranteed in very special cases, because of potential for data races

Example with Functional Determinism and Structural Nondeterminism

```
static boolean found = false; //static field
. . .
finish for (int i = 0; i <= N - M; i++) {
  if (found) break; // Eureka!
  async {
    for (j = 0; j < M; j++)
      if (text[i+j] != pattern[j]) break;
    if (j == M) found = true;
   } // async
} // finish-for
```

Example with Structural Determinism and Functional Nondeterminism

```
// Index of an occurrence
static int index = -1; // static field
. . .
finish
 for (int i = 0; i <= N - M; i++)
  async {
   for (j = 0; j < M; j++)
    if (text[i+j] != pattern[j]) break;
   if (j == M) index = i; // found at i
  }
```

Semantic Property #5: Data-Race-Free Determinism

- If a parallel program is known to be data-race-free, then it must be both functionally deterministic and structurally deterministic
- All HJlib constructs that you have learned thus far satisfy this property!
  - Does not apply to parallel Java constructs in general

- Task Parallelism (Unit 1)
  - —Async (task creation)
  - —Finish (structured task termination)
- Functional Parallelism (Unit 2)
  - —Future (task creation)
  - —Future get() (task termination with return value)
  - —Accumulators (functional reduction)
  - —Map-Reduce (functional parallelism & reduction on key-value pairs)
- Loop Parallelism (Unit 3)
  - —Forall (parallel loops)
  - —Barriers (all-to-all synchronization)
- Dataflow Parallelism (Unit 4)
  - —Data-Driven Tasks (dataflow parallelism)
  - —Phasers (point-to-point synchronization)
  - —Phaser-specific next operations

Semantic Properties of Parallel Programs

- *Serializable* programs =

  { async, finish, future }
- *Deadlock-free* programs =

  Serializable U { barriers, phasers }
- Programs for which data-race-freedom implies both structural and functional determinism =

  Deadlock-free U { per-phaser next,
  async await }