COMP 322: Fundamentals of Parallel Programming

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

Lecture 17: Advanced Phaser Topics

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

• Feb 23rd lecture will be a Midterm Review
• No COMP 322 labs this week
• No lecture on Friday, Feb 25th since midterm exam is due that day
  — Midterm will be a 2-hour take-home written exam
    - Closed-book, closed-notes, closed-computer
  — Will be given out at lecture on Wed, Feb 23rd
  — Must be handed in by 5pm on Friday, Feb 25th
  — Scope of midterm exam will be Lectures 1-15 and Lecture 17
    - Lecture 16 (Bitonic Sort) will not be included in midterm exam
Acknowledgments for Today’s Lecture

• Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. Jun Shirako et al. ICS ’08


• Handout for Lectures 17
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
• isolated-start: destination of serialization edges
• isolated-end: source of serialization edges
• 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)
Left-Right Neighbor Synchronization
Example for m=3 (Listing 1)

```java
finish {
    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 1: Example of left-right neighbor synchronization for $m = 3$ case
Computation Graph for m=3 example

1, 2, 3 → 20-drop → 20-end-finish

6 → 7-signal → 7-wait → 8

11 → 12-signal → 12-wait → 13

16 → 17-signal → 17-wait → 18

spawn continue signal wait join

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

1. **6** → **7-signal** → **7-wait** → **8**
2. **11** → **12-signal** → **12-wait** → **13**
3. **16** → **17-signal** → **17-wait** → **18**

Nodes:
- **ph1.next**
  - start(0→1)
  - end(0→1)

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

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

Edges:
- **spawn**
- **continue**
- **signal**
- **wait**
- **join**
Translation of Barrier to Phaser Version

```
rank.count = 0;  // rank object contains an int field, count
forall (point[i] : [0:m−1]) {
  // Start of phase 0
  int r;
  isolated {r = rank.count++;}
  System.out.println(“Hello from task ranked” + r);
  next;  // Acts as barrier between phases 0 and 1
  // Start of phase 1
  System.out.println(“Goodbye from task ranked” + r);
}
```

Listing 2: Hello-Goodbye forall loop with barrier (next) statement

```
rank.count = 0;  // rank object contains an int field, count
finish {
  phaser ph = new phaser(phaserMode.SIG_WAIT);
  for (point[i] : [0:m−1]) async phased {
    // Start of phase 0
    int r;
    isolated {r = rank.count++;}
    System.out.println(“Hello from task ranked” + r);
    next;  // Acts as barrier between phases 0 and 1
    // Start of phase 1
    System.out.println(“Goodbye from task ranked” + r);
  }  // for async phased
}
```

Listing 3: Translation of Listing 2 to a finish-for-async-phased code structure (phaser version)
Optimized One-Dimensional Iterative Averaging with Barrier Synchronization

```java
double[] val1 = new double[n]; val[0] = 0; val[n+1] = 1;
double[] val2 = new double[n];
int batch_size = CeilDiv(n, t); // Number of elements per task
forall (point [i] : [0:t-1]) { // Create t tasks
double[] myVal = val1; double myNew = val2; double[] temp = null;
int start = i*batchSize+1; int end = Math.min(start+batchSize-1, n);
for (point [iter] : [0:iterations-1]) {
    for (point [j] : [start:end])
        myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    next; // barrier
    temp = myNew; myNew = myVal; myVal = temp; // swap(myNew, myVal)
} // for
} // forall
```

Listing 4: Optimized One-Dimensional Iterative Averaging Example using forall-for-next computation structure with \( t \) parallel tasks working on an array with \( n + 2 \) elements (each task processes a batch of array elements)
Optimized One-Dimensional Iterative Averaging with Point-to-Point Synchronization

```java
double[] val1 = new double[n]; val[0] = 0; val[n+1] = 1;
double[] val2 = new double[n];
int batchSize = CeilDiv(n,t); // Number of elements per task

finish {
    phaser ph = new phaser[t+2];
    forall(point [i]:[0:t+1]) ph[i]=new phaser(phaserMode.SIG_WAIT);
    for (point [i] : [1:t])
        async phased(ph[i]<SIG>, ph[i−1]<WAIT>, ph[i+1]<WAIT>) {
            double[] myVal = val1; double myNew = val2; double[] temp = null;
            int start = (i−1)*batchSize + 1; int end = Math.min(start+batchSize−1,n);
            for (point [iter] : [0:iterations−1]) {
                for (point [j] : [start:end])
                    myNew[j] = (myVal[j−1] + myVal[j+1])/2.0;
                next; // signal ph[i] and wait on ph[i−1] and ph[i+1]
                temp = myNew; myNew = myVal; myVal = temp; // swap(myNew, myVal)
            } // for
        } // for-async
} // finish
```

Listing 5: Optimized One-Dimensional Iterative Averaging Example using point-to-point synchronization, instead of barrier synchronization as in Listing 4
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
```

Listing 6: Example of split-phase barrier
Computation Graph for Split-Phase Barrier Example

2 \rightarrow 4 \rightarrow 5 - signal \rightarrow 6 \rightarrow 7 - wait \rightarrow 8

20 - drop \rightarrow 20 - end - finish

11 \rightarrow 12 - signal \rightarrow 13 \rightarrow 14 - wait \rightarrow 15

spawn \rightarrow continue \rightarrow signal \rightarrow wait \rightarrow join

ph.next - start(0 \rightarrow 1) \rightarrow ph.next - end(0 \rightarrow 1)
Computation Graph for Split-Phase Barrier Example (without async and finish nodes and edges)

```
4 -> 5-signal -> 6 -> 7-wait -> 8

ph.next
    -start(0→1)

ph.next
    -end(0→1)

11 -> 12-signal -> 13 -> 14-wait -> 15

spawn
continue
signal
wait
join
```
Optimized One-Dimensional Iterative Averaging with Split-Phase Point-to-Point Synchronization

```java
    double[] val1 = new double[n]; val[0] = 0; val[n+1] = 1;
    double[] val2 = new double[n];
    int batchSize = CeilDiv(n,t); // Number of elements per task
    finish {
        phaser ph = new phaser[t+2];
        forall (point [i]:[0:t+1]) ph[i]=new phaser(phaserMode.SIG_WAIT);
        for (point [i] : [1:t])
            async phased(ph[i]<SIG>, ph[i-1]<WAIT>, ph[i+1]<WAIT>) {
                double[] myVal = val1; double myNew = val2; double[] temp = null;
                int start=(i-1)*batchSize+1; int end=Math.min(start+batchSize-1,n);
                for (point [iter] : [0:iterations-1]) {
                    myNew[start] = (myVal[start-1] + myVal[start+1])/2.0;
                    myNew[end] = (myVal[end-1] + myVal[end+1])/2.0;
                    signal; // signal ph[i]
                for (point [j] : [start+1:end-1])
                    myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
                    next; // wait on ph[i-1] and ph[i+1]
                    temp = myNew; myNew = myVal; myVal = temp; // swap(myNew, myVal)
                } // for
            } // async
    } // finish
```

Listing 7: Optimized One-Dimensional Iterative Averaging Example using signal statements for split-phase point-to-point synchronization