4.1 Lecture Summary

---

4 Dataflow Synchronization and Pipelining

4.1 Split-phase Barriers with Java Phasers

Lecture Summary: In this lecture, we examined a variant of the *barrier* example that we studied earlier:

```
forall (i : [0:n-1]) {
  print HELLO, i;
  myId = lookup(i); // convert int to a string
  print BYE, myId;
}
```

We learned about Java's Phaser class, and that the operation ph.arriveAndAwaitAdvance(), can be used to implement a barrier through phaser object ph. We also observed that there are two possible positions for inserting a barrier between the two print statements above — before or after the call to lookup(i). However, upon closer examination, we can see that the call to lookup(i) is local to iteration i and that there is no specific need to either complete it before the barrier or to complete it after the barrier. In fact, the call to lookup(i) can be performed in parallel with the barrier. To facilitate this *split-phase barrier* (also known as a *fuzzy barrier*) we use two separate APIs from Java Phaser class — ph.arrive() and ph.awaitAdvance(). Together these two APIs form a barrier, but we now have the freedom to insert a computation such as lookup(i) between the two calls as follows:

```
// initialize phaser ph  for use by n tasks ("parties")
Phaser ph = new Phaser(n);
// Create forall loop with n iterations that operate on ph
forall (i : [0:n-1]) {
  print HELLO, i;
  int phase = ph.arrive();

  myId = lookup(i); // convert int to a string

  ph.awaitAdvance(phase);
  print BYE, myId;
```

}

Doing so enables the barrier processing to occur in parallel with the call to lookup(i), which was our desired outcome.

Optional Reading:

1. Documentation on Java's Phaser class.

4.2 Lecture Summary

---

4 Dataflow Synchronization and Pipelining

4.2 Point-to-Point Synchronization with Phasers

Lecture Summary: In this lecture, we looked at a parallel program example in which the span (critical path length) would be 6 units of time if we used a barrier, but is reduced to 5 units of time if we use individual phasers as shown in the following table:

Task 0	**Task 1**	**Task 2**
1a:X=A();//cost=1	1b:Y=B();//cost=2	1c:Z=C();//cost=3
2a:ph0.arrive();	*2b:ph1.arrive();*	*2c:ph2.arrive();*
3a:ph1.awaitAdvance(0);	*3b:ph0.awaitAdvance(0);*	*3c:ph1.awaitAdvance(0);*
4a:D(X,Y);//cost=3	*4b:ph2.awaitAdvance(0);*	4c:F(Y,Z);//cost=1
	5b:E(X,Y,Z);//cost=2	

Each column in the table represents execution of a separate task, and the calls to arrive() and awaitAdvance(0) represent synchronization across different tasks via phaser objects, ph0, ph1, and ph2, each of which is initialized with a party count of 1 (only one signalling task). (The parameter 0 in awaitAdvance(0) represents a transition from phase 0 to phase 1.)

Optional Reading:

1. Documentation on Java Phaser class.

4.3 Lecture Summary

---

4 Dataflow Synchronization and Pipelining

4.3 One-Dimensional Iterative Averaging with Phasers

Lecture Summary: In this lecture, we revisited the barrier-based Iterative Averaging example that we studied earlier, and observed that a full barrier is not necessary since *forall* iteration *i* only needs to wait for iterations $i - 1$ and $i + 1$ to complete their current phase before iteration *i* can move to its next phase. This idea can be captured by phasers, if we allocate an array of phasers as follows:

```
// Allocate array of phasers
Phaser[] ph = new Phaser[n+2]; //array of phasers
for (int i = 0; i < ph.length; i++) ph[i] = new Phaser(1);

// Main computation
forall ( i: [1:n-1]) {
  for (iter: [0:nsteps-1]) {
    newX[i] = (oldX[i-1] + oldX[i+1]) / 2;
    ph[i].arrive();

    if (index > 1) ph[i-1].awaitAdvance(iter);
    if (index < n-1) ph[i + 1].awaitAdvance(iter);
    swap pointers newX and oldX;
  }
}
```

As we learned earlier, grouping/chunking of parallel iterations in a *forall* can be an important consideration for performance (due to reduced overhead). The idea of grouping of parallel iterations can be extended to *forall* loops with phasers as follows:

```
// Allocate array of phasers proportional to number of chunked tasks
Phaser[] ph = new Phaser[tasks+2]; //array of phasers
for (int i = 0; i < ph.length; i++) ph[i] = new Phaser(1);

// Main computation
forall ( i: [0:tasks-1]) {
  for (iter: [0:nsteps-1]) {
    // Compute leftmost boundary element for group
    int left = i * (n / tasks) + 1;
    myNew[left] = (myVal[left - 1] + myVal[left + 1]) / 2.0;

    // Compute rightmost boundary element for group
```

```
    int right = (i + 1) * (n / tasks);
    myNew[right] = (myVal[right - 1] + myVal[right + 1]) / 2.0;

    // Signal arrival on phaser ph AND LEFT AND RIGHT ELEMENTS ARE AV
    int    index = i + 1;
    ph[index].arrive();

    // Compute interior elements in parallel with barrier
    for (int j = left + 1; j <= right - 1; j++)
      myNew[j] = (myVal[j - 1] + myVal[j + 1]) / 2.0;
    // Wait for previous phase to complete before advancing
    if (index > 1) ph[index - 1].awaitAdvance(iter);
    if (index < tasks) ph[index + 1].awaitAdvance(iter);
    swap pointers newX and oldX;
  }
}
```

Optional Reading:

1. Documentation on Java's Phaser class.

4.4 Lecture Summary

---

4 Dataflow Synchronization and Pipelining

4.4 Pipeline Parallelism

Lecture Summary: In this lecture, we studied how point-to-point synchronization can be used to build a one-dimensional *pipeline* with p tasks (stages), $T0, \ldots, Tp{-}1$. For example, three important stages in a *medical imaging* pipeline are *denoising*, *registration*, and *segmentation*.

We performed a simplified analysis of the *WORK* and *SPAN* for pipeline parallelism as follows. Let n be the number of input items and p the number of stages in the pipeline, $WORK = n \times p$ is the total work that must be done for all data items, and $CPL = n + p - 1$ is the *span*or *critical path length* for the pipeline. Thus, the ideal parallelism is $PAR = WORK / CPL = np / (n + p - 1)$. This formula can be validated by considering a few boundary cases. When $p = 1$, the ideal parallelism degenerates to $PAR = 1$, which confirms that the computation is sequential when only one stage is available. Likewise, when

$n = 1$, the ideal parallelism again degenerates to $PAR = 1$, which confirms that the computation is sequential when only one data item is available. When n is much larger than p ($n \gg p$), then the ideal parallelism approaches $PAR = p$ in the limit, which is the best possible case.

The synchronization required for pipeline parallelism can be implemented using phasers by allocating an array of phasers, such that phaser ph[i] is "signalled" in iteration i by a call to ph[i].arrive() as follows:

```
// Code for pipeline stage i
while ( there is an input to be processed ) {
  // wait for previous stage, if any
  if (i > 0) ph[i - 1].awaitAdvance();

  process input;

  // signal next stage
  ph[i].arrive();
}
```

Optional Reading:

1. Wikipedia article on Pipeline (computing).

4.5 Lecture Summary

---

4 Dataflow Synchronization and Pipelining

4.5 Data Flow Parallelism

Lecture Summary: Thus far, we have studied computation graphs as structures that are derived from parallel programs. In this lecture, we studied a dual approach advocated in the *data flow parallelism* model, which is to specify parallel programs as computation graphs. The simple data flow graph studied in the lecture consisted of five nodes and four edges: $A \rightarrow C$, $A \rightarrow D$, $B \rightarrow D$, $B \rightarrow E$. While futures can be used to generate such a computation graph, e.g., by including calls to A.get() and B.get() in task D, the computation graph edges are implicit in the get() calls when using futures. Instead, we introduced the asyncAwait notation to specify a task along with an explicit set of preconditions (events that the task must wait for before it can start execution). With this approach, the program can be generated directly from the computation graph as follows:

```
async( () -> {/* Task A */; A.put(); } ); // Complete task and trigger event A
```

```
async( () -> {/* Task B */; B.put(); } ); // Complete task and trigger event B
asyncAwait(A, () -> {/* Task C */} );              // Only execute task after event A is triggered
asyncAwait(A, B, () -> {/* Task D */} );       // Only execute task after events A, B are triggered
asyncAwait(B, () -> {/* Task E */} );              // Only execute task after event B is triggered
```

Interestingly, the order of the above statements is not significant. Just as a graph can be defined by enumerating its edges in any order, the above data flow program can be rewritten as follows, without changing its meaning:

```
asyncAwait(A, () -> {/* Task C */} );              // Only execute task after event A is triggered
asyncAwait(A, B, () -> {/* Task D */} );       // Only execute task after events A, B are triggered
asyncAwait(B, () -> {/* Task E */} );          // Only execute task after event B is triggered
async( () -> {/* Task A */; A.put(); } ); // Complete task and trigger event A
async( () -> {/* Task B */; B.put(); } ); // Complete task and trigger event B
```

Finally, we observed that the power and elegance of data flow parallel programming is accompanied by the possibility of a lack of progress that can be viewed as a form of “deadlock” if the program omits a put() call for signalling an event.

Optional Reading:

1. Wikipedia article on Data flow diagram

Mini Project 4: Using Phasers to Optimize Data-Parallel Applications

---

miniproject_4.zip

Project Goals & Outcomes

In lectures this week, you learned about phasers (not the Star Trek kind!) and how they can be used to express “fuzzy” (split-phase) barriers and point-to-point synchronization. You also saw how fuzzy barriers and point-to-point synchronization can be used to improve performance, relative to regular barriers. In particular, fuzzy barriers allow parallel programmers to expose more parallelism than a simple barrier (specified by the *arriveAndAwaitAdvance()* API) by overlapping the completion of a barrier with useful work. In a fuzzy barrier, the completion of the barrier is split into two steps. In the first step (the *arrive()* API), each thread signals to other threads that it has reached a point in its own execution where it is safe for any other thread to proceed past the barrier. In the second step (the *awaitAdvance()* API), each thread checks that all other threads have signaled that they have reached step one to be sure it can proceed past the barrier.

In this mini-project you will apply Java's phasers and Java threads to the One-Dimensional Iterative Averaging algorithm that was introduced in Week 3. Before getting started with this mini-project, it is recommended you review the first demo video in Week 4, in which Professor Sarkar walks through an example similar to this project.

Project Setup

Please refer to Mini-Project 0 for a description of the build and testing process used in this course.

Once you have downloaded and unzipped the provided project files using the gray button labeled miniproject_4.zip at the top of this description, you should see the project source code at:

miniproject_4/src/main/java/edu/coursera/parallel/OneDimAveragingPhaser.java

and the project tests at

miniproject_4/src/test/java/edu/coursera/parallel/OneDimAveragingPhaser.java.

Project Instructions

Your modifications should be done entirely inside of OneDimAveragingPhaser.java. You may not make any changes to the signatures of any public or protected methods inside of OneDimAveragingPhaser, or remove any of them. However, you are free to add any new methods you like. Any changes you make to OneDimAveragingPhaserTest.java will be ignored in the final grading process.

Your main goals for this assignment are listed below. OneDimAveragingPhaser.java also contains helpful TODOs.

1. Based on the provided reference sequential version in OneDimAveragingPhaser.runSequential and the reference parallel version in OneDimAveragingPhaser.runParallelBarrier, implement a parallel version of the one-dimensional iterative averaging algorithm that uses phasers to maximize the overlap between barrier completion and useful work being completed.

Project Evaluation

Your assignment submission should consist of only the OneDimAveragingPhaser.java file that you modified to implement this mini-project. As before, you can upload this file through the assignment page for this mini-project. After that, the Coursera autograder will take over and assess your submission, which includes building your code and running it on one or more tests. Your submission will be evaluated on Coursera's auto-grading system using 2 and 4 CPU cores. Note that the performance observed for tests on your local machine may differ from that on Coursera's auto-grading system, but that you will only be evaluated on the measured performance on Coursera. Also note that for all assignments in this course you are free to resubmit as many times as you like. See the Common Pitfalls page under Resources for more details. Please give it a few minutes to complete the grading. Once it has completed, you should see a score appear in the "Score" column of the "My submission" tab based on the following rubric:

- 100% – correctness and performance of the fuzzy barrier implementation of one-dimensional iterative averaging on an input of 4 * 1024 * 1024 elements using 4 cores