COMP 322: Fundamentals of Parallel Programming

Lecture 16: Phaser Accumulators, Bounded Phasers

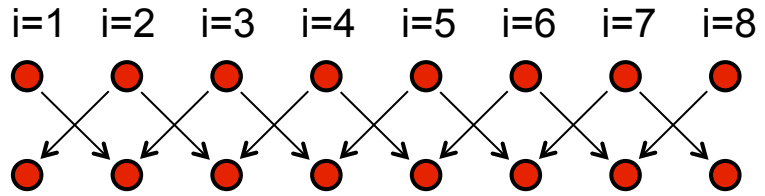
Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

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

Solution to Worksheet #15: Left-Right Neighbor Synchronization using Phasers

Name 1: _____

Name 2: _____



Complete the phased clause below to implement the left-right neighbor synchronization shown above

```
1. finish {
2.   phaser[] ph = new phaser[m+2]; // array of phaser objects
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;
8.      doPhase2(i);
9.    }
10.}
```


Outline of Today's Lecture

- **Phaser Accumulators**
- **Bounded Phasers**

Problem: Max reduction in One-Dimensional Iterative Averaging with Barrier Synchronization

```
1. finish {
2.   phaser ph = new phaser(phaserMode.SIG_WAIT); double maxDiff;
3.   forasync (point [jj]:[0:Cj-1]) phased(ph) { // forasync w/ explicit phaser
4.     double[] myVal = gVal; double[] myNew = gNew; // Local copies of pointers
5.     do { // use a do-while loop instead of a for loop
6.       // Compute MyNew as function of input array MyVal
7.       for (point [j]:getChunk([1:n],[Cj],[jj])) { // Iterate within chunk
8.         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.         // Compute normalized diff for element j; check for divide by zero
10.        double nDiff = myNew[j] == 0 ? 0 :
11.                       Math.abs(myNew[j]-myVal[j])/Math.abs(myNew[j]);
12.      }
13.      // QUESTION: how to compute max(nDiff) for all elements in this phase??
14.      maxDiff = ... ;
15.      next; // Barrier before executing next iteration of iter loop
16.      double[] temp=myVal; myVal=myNew; myNew=temp; // Swap myVal and myNew
17.    } while (maxDiff > ERROR_THRESHOLD);
18.  } // forasync
19.} // finish
```


Finish Accumulators provide max value over all phases, not per-phase max value

```
1.  accumulator m = accumulator.factory.accumulator(MAX, double.class);
2.  finish(m) {
3.    ph = new phaser(phaserMode.SIG_WAIT)
4.    forasync (point [jj]:[0:Cj-1]) phased(ph) { // Explicit chunked forall
5.      double[] myVal = gVal; double[] myNew = gNew; // Local copies of pointers
6.      while (m.get() > ERROR_THRESHOLD) {
7.        // Compute MyNew as function of input array MyVal
8.        for (point [j]:getChunk([1:n],Cj,jj)) { // Iterate within chunk
9.          myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10.         double nDiff = myNew[j] == 0 ? 0 :
11.                        Math.abs(myNew[j]-myVal[j])/Math.abs(myNew[j]);
12.         m.put(nDiff); // accumulate nDiff values into max function
13.       }
14.       next; // Barrier before executing next iteration of iter loop
15.       double[] temp=myVal; myVal=myNew; myNew=temp; // Swap myVal and myNew
16.     } while (m.get() > ERROR_THRESHOLD); // returns Double.MIN_VALUE (init value)
17.   } // forasync
18. } // finish
19. ... = m.get(); // max value overall phases (will come from first while iteration)
```


Phaser Accumulators

- **Phaser accumulators can accumulate values within a single phase e.g., between two "next" operations**

- Implementation notes:
  - HJ provides different implementations for the same accumulator semantics
    - Eager: Concurrent atomic accumulation by multiple tasks
    - Dynamic-lazy: Sequential accumulation at next
    - Fixed-Lazy: Optimization of dynamic-lazy when number of tasks registered on phaser accumulator is fixed in advance
  - Phasers and phaser accumulators are currently only supported by HJ's work-sharing runtime (w/ or w/o the fork-join variant, -fj), but not HJ's work-stealing runtime

Operations on Phaser Accumulators

- **Creation**

```
accumulator ac = accumulator.factory.accumulator(op,type,phaser);
```

  - *operator can be Operator.SUM, Operator.PROD, Operator.MIN, or Operator.MAX (as in finish accumulators)*
  - *type can be* int.class or double.class *(as in finish accumulators)*

- **Accumulation**

```
ac.put(data);
```

  - *data must be of type java.lang.Number, int, or double*
  - *Provides data for accumulation in current phase (can only be performed by a task registered on the phaser)*

- **Retrieval**

```
Number n = ac.get();
```

  - *get() returns value from previous phase (can only be performed by a task registered on the phaser)*
  - *get() is non-blocking because the synchronization is handled by "next"*
  - *result from get() will be deterministic if HJ program does not use atomic or isolated constructs and is data-race-free (ignoring nondeterminism due to non-commutativity of arithmetic operations, e.g., underflow, overflow, rounding)*

Example of Phaser Accumulators

```
1. finish {
2.   phaser ph = new phaser();
3.   accumulator a = accumulator.factory.accumulator(accumulator.SUM,
4.                                                    int.class, ph);
5.   accumulator b = accumulator.factory.accumulator(accumulator.MIN,
6.                                                    double.class, ph);
7.   for (int i = 0; i < n; i++) { // can use forasync instead
8.     async phased(ph<phaserMode.SIG_WAIT>) {
9.       int iv = 2*i + j;
10.      double dv = -1.5*i + j;
11.      a.put(iv);
12.      b.put(dv);
13.      next;
14.      int sum = a.get().intValue;
15.      double min = b.get().doubleValue();
16.      …
17.    } // async
18.  } // for
19.}
```

Allocation: Specify operator and type

put: Send a value to accumulator

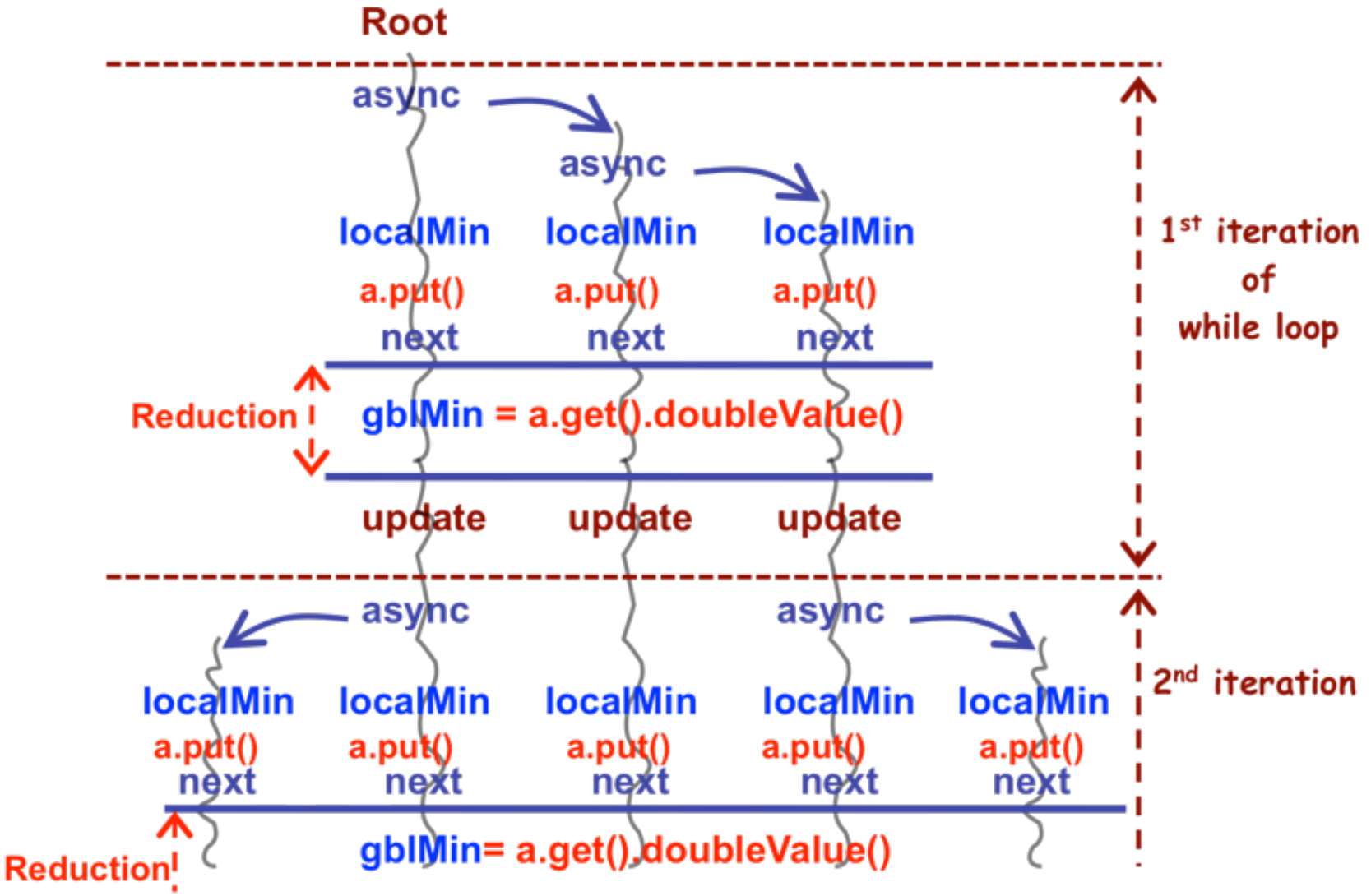
get: Return accumulator result from previous phase

Got the idea? Let's try another phaser example in Worksheet 16!

Example of Phaser Accumulators with Dynamic Parallelism: Search for Minimum Cost Solution

```
1. double gblMin = Double.MAX_VALUE;  double threshold = …;
2. SearchSpace gss = new SearchSpace(…); // Whole search space
3. finish {
4.   phaser ph = new phaser();
5.   accumulator a = accumulator.factory.accumulator(accumulator.MIN,
6.                                                   double.class, ph);
7.   calcMin(ph, gss, a);
8. }
9. . . .
10. void calcMin(phaser ph, SearchSpace mySs, accumulator a) {
11.  while (gblMin > threshold) {
12.   if (mySs.tooLarge()) {
13.      SearchSpace childSs = split(mySs);
14.       async phased { calcMin(ph, childSs, a); }
15.   }
16.   double localMin = findMin(mySs);
17.   a.put(localMin);
18.   next;
19.   gblMin = a.get().doubleValue();
20.   // update search spaces ...
21.  } // while
22.} // calcMin
```


Execution of previous HJ program



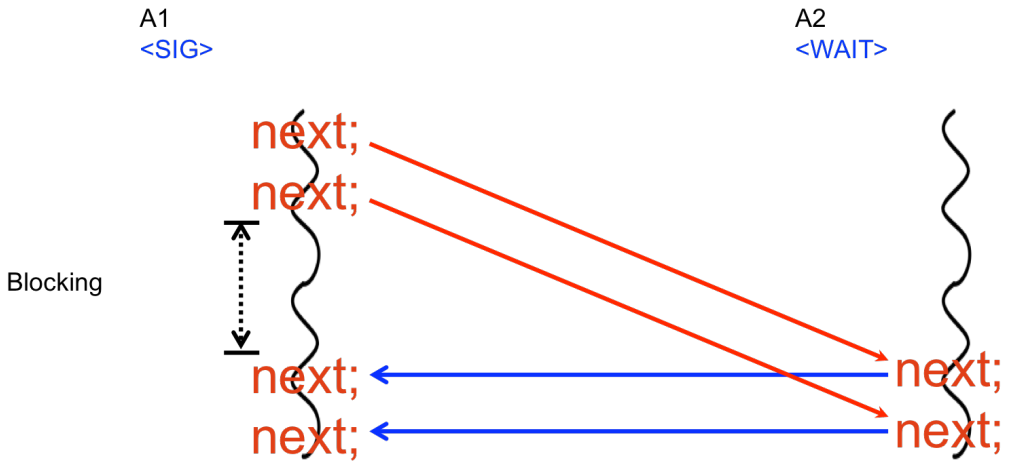
Outline of Today's Lecture

- Phaser Accumulators
- Bounded Phasers

Bound option in phasers

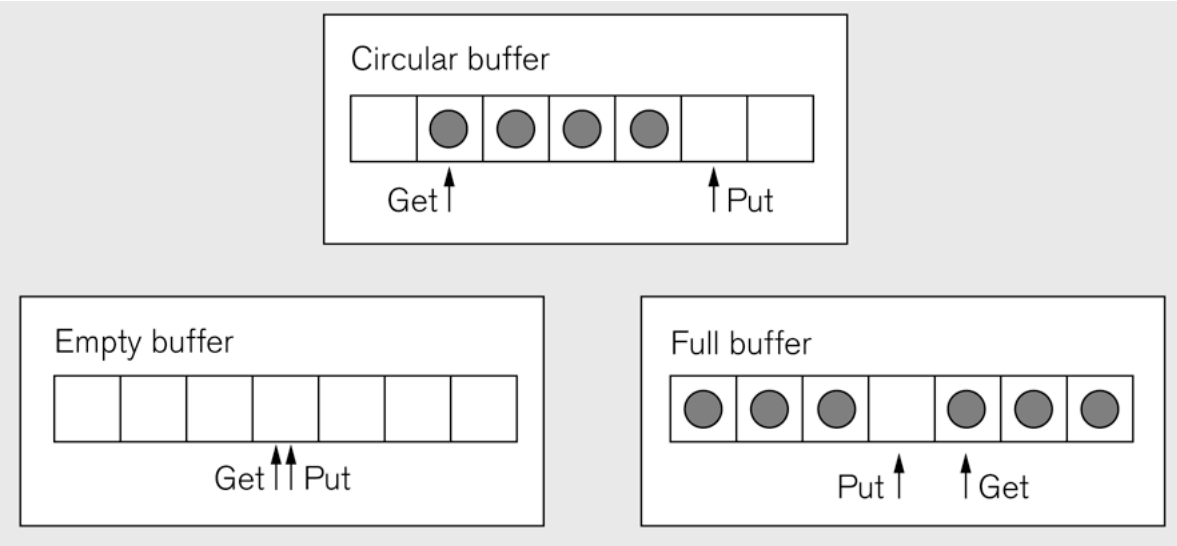
- Extra parameter in constructor
  - `new phaser(phaserMode m, int bound_size);`
- next operation
  - —A task registered in SIG mode will block if it is >= bound_size phases past the current phase

```
...
phaser ph = new phaser(<SIG_WAIT>, 2 /*Bound size*/);
async phased (ph<SIG>) { next; next; ... /*A1*/ }
async phased (ph<WAIT>) { next; next; ... /*A2*/ }
...
```



Single-Producer Single-Consumer Bounded Buffer Problem

A bounded buffer with a single producer and a single consumer. The Put and Get cursors indicate where the producer will insert the next item and where the consumer will remove its next item.



We will revisit this problem with multiple producers and consumers later in the course

- Requires nondeterministic merge in general

Single-Producer Single-Consumer Bounded Buffer

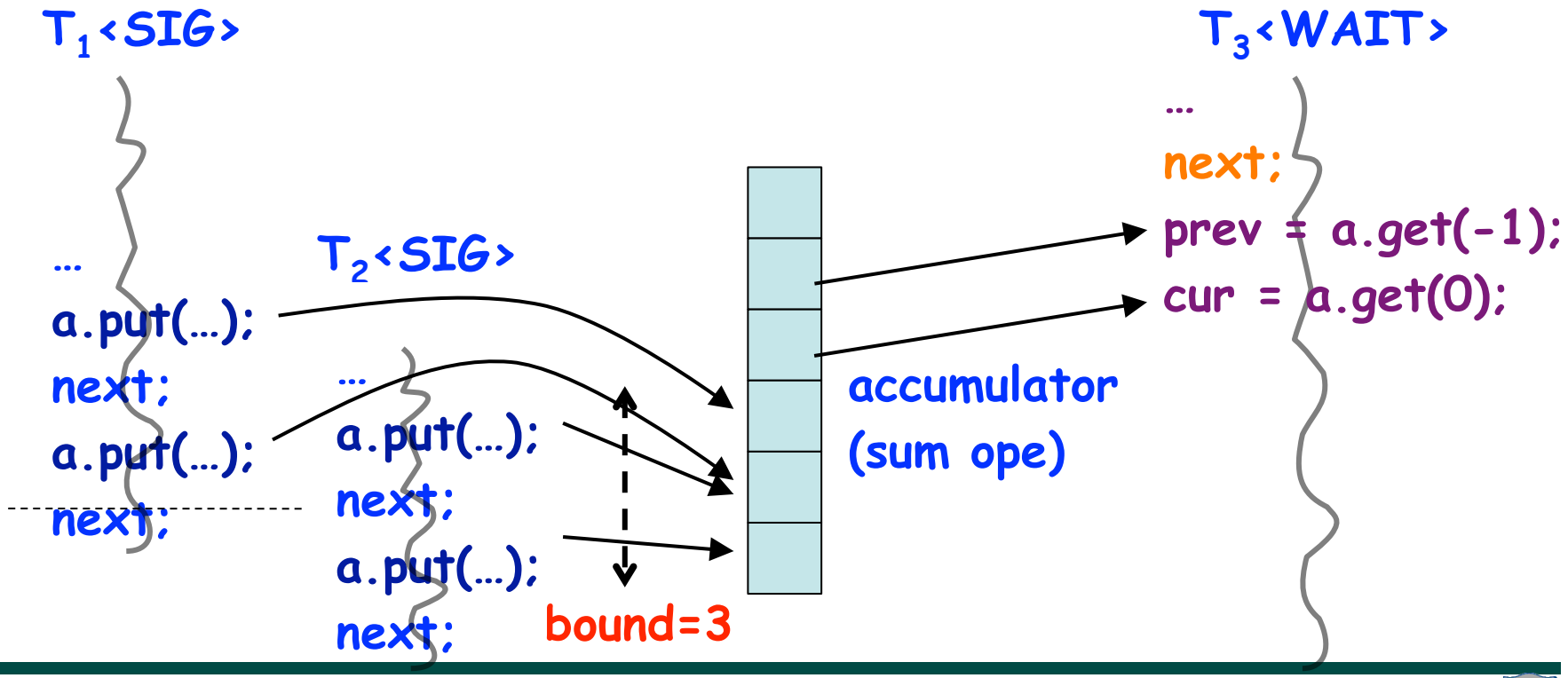
```
1. finish {
2.   phaser ph = new phaser(<SIG_WAIT>, bound_size);
3.   async phased (ph<SIG>)
4.     while (…) { insert(); next; } // producer
5.   async phased (ph<WAIT>)
6.    while (…) { next; remove(); } // consumer
7. }
```

- How would this code behave if there was no bound specified for the phaser?

Expanding Accumulators to support Bounded Buffers

```
phaser ph      = new phaser(SIG_WAIT, bound);
accumulator a = new accumulator(ph, SUM, double.class);
```

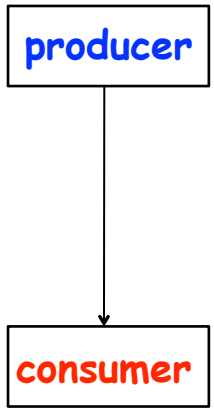
- Accumulator is now a bounded buffer
  - —Stores results from bounded number of previous phase



Streaming Computations: Application of Bounded Buffer Computations

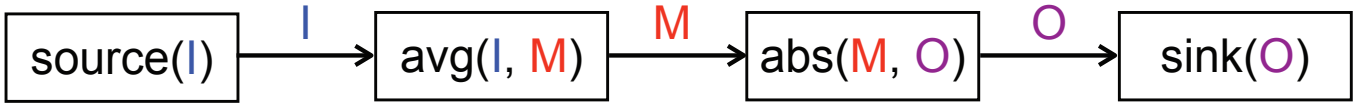
- Producer task (filter)
  - —Insert data into stream
  - —Can go ahead of consumers
  - —Registered on phaser in SIG mode
- Consumer task (filter)
  - —Consume data from stream
  - —Must wait for producer
  - —Registered on phaser in WAIT mode
- Streams
  - —Manage communication among tasks
    - Retain data in bounded buffer
  - —Accumulators can be expanded to implement bounded buffers
  - —*Need explicit phaser wait operation if a task needs to be both a producer and a consumer*

```
phaser ph = new phaser();
async phased (ph<SIG>) {
  while(...) {
    ...
    next;
} }

async phased (ph<WAIT>) {
while(...) {
    next;
    ...
} }
```



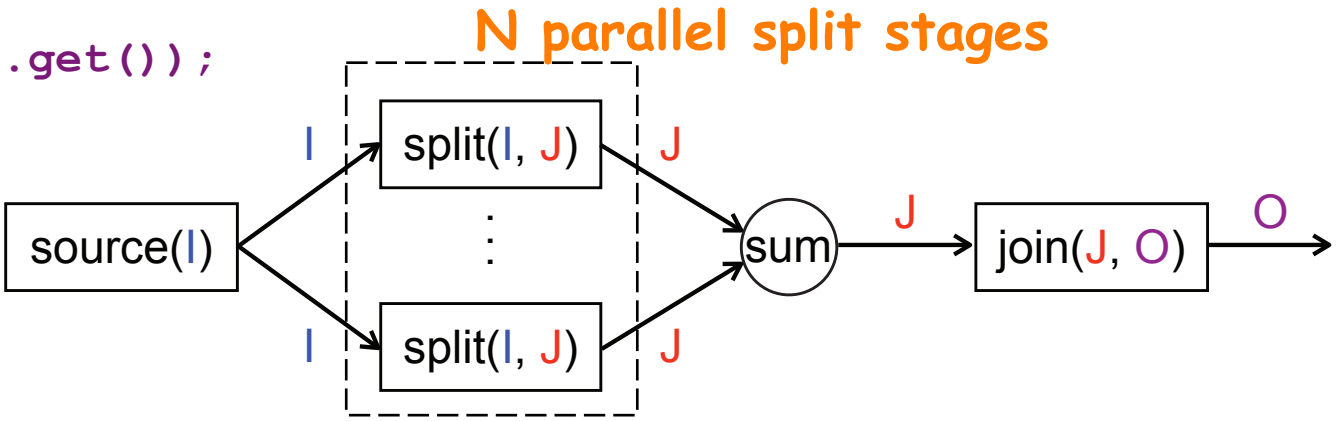
Streaming Computation: Pipeline

```
void Pipeline() {
    phaser phI    = new phaser(SIG_WAIT, bnd);
    accumulator I = new accumulator(phI, accumulator.ANY);
    phaser phM    = new phaser(SIG_WAIT, bnd);
    accumulator M = new accumulator(phM, accumulator.ANY);
    phaser phO    = new phaser(SIG_WAIT, bnd);
    accumulator O = new accumulator(phO, accumulator.ANY);
    async phased (phI<SIG>)                source(I);
    async phased (phI<WAIT>, phM<SIG>) avg(I,M);
    async phased (phM<WAIT>, phO<SIG>) abs(M,O);
    async phased (phO<WAIT>)               sink(O);
}
void avg(accumulator I, accumulator M) {
    while(...) {
        wait; wait;          // explicit wait for two I  elements
        v1 = I.get(0);       // read first element
        v2 = I.get(-1);      // read second element (offset = -1)
        M.put((v1+v2)/2); // put result on M
        signal;              // Explicit signal
} }
```



Streaming Patterns: Split-join

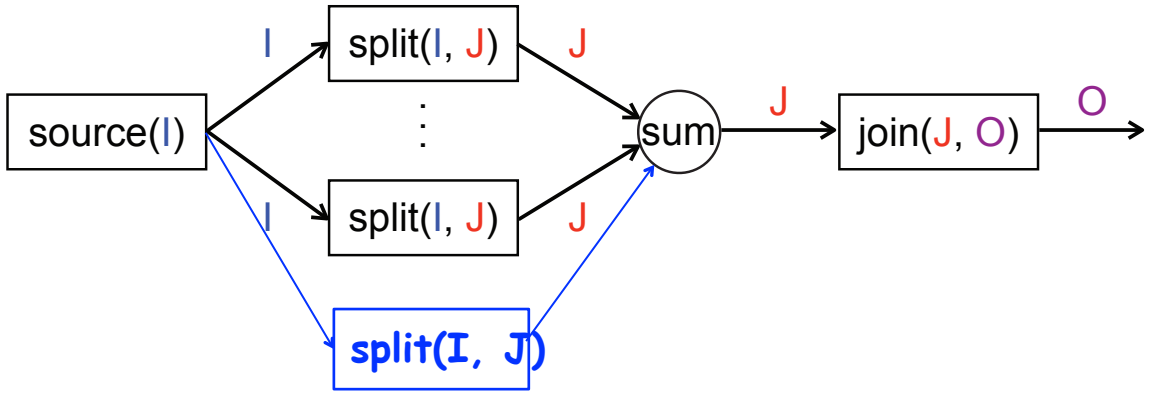
```
void Splitjoin() {
    phaser phI        = new phaser(SIG_WAIT, bnd);
    accumulator I     = new accumulator(phI, accumulator.ANY);
    phaser phJ        = new phaser(SIG_WAIT, bnd);
    accumulator J     = new accumulator(phJ, accumulator.SUM);

    async phased (phI<SIG>)                source(I);
    forasync (point [s] : [0:N-1])
        phased (phI<WAIT>, phJ<SIG>) split(I, J);
    async phased (phJ<WAIT>)               join(J);
}
split(I, J) {
    while(...) {
        wait;
        v = foo(I.get());
        J.put(v);
        signal;
} }
```



General Streaming Graphs with Dynamic Parallelism

- Dynamic split-join

```
dynamicSplit(I, J) {
  while(...) {
    if (spawnNewNode()) async phased dynamicSplit(I, J);
    if (terminate())    break;
    wait; ...
} }
```



Stages can be spawned/terminated dynamically

Summary of Barriers and Phasers

- Implicit phaser in a forall supports barriers as "next" statements
  - —Matching of next statements occurs dynamically during program execution
  - —Termination signals "dropping" of phaser registration
  - —next single -- augment barrier with "single" computations
- Explicit phasers
  - —Can be allocated and transmitted from parent to child tasks
  - —Phaser lifetime is restricted to its IEF (Immediately Enclosing FInish) scope of its creation
  - —Four registration modes -- SIG, WAIT, SIG_WAIT, SIG_WAIT_SINGLE
  - —signal statement can be used to support split-phase ("fuzzy") barriers
  - —phaser accumulators can perform per-phase reduction
  - —bounded phasers can limit how far ahead producer gets of consumers
  - —phaser accumulators with bounded phasers can support bounded buffer streaming computations

Worksheet #16: Left-Right Neighbor Synchronization using Phasers

Name 1: _____ Name 2: _____

Complete the three blank (_____) entries below for a correct use of phaser accumulators in the One-Dimensional Iterative Averaging Example

One Dimensional Iterative Averaging with Phaser Accumulators --- fill in the blanks

```
1. finish {
2.   ph = new phaser(phaserMode.SIG_WAIT)
3.   accumulator m = accumulator.factory.accumulator(MAX,double.class,_________);
4.   forasync (point [jj]:[0:Cj-1]) phased(ph) { // Explicit chunked forall
5.     double[] myVal = gVal; double[] myNew = gNew; // Local copies of pointers
6.     while (_________________ > ERROR_THRESHOLD) {
7.       // Compute MyNew as function of input array MyVal
8.       for (point [j]:getChunk([1:n],Cj,jj)) { // Iterate within chunk
9.         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10.       double nDiff = (myNew[j] == 0) ? 0 : // check for divide-by-zero
11.                      Math.abs(myNew[j]-myVal[j])/Math.abs(myNew[j]);
12.       _____________________; // accumulate nDiff values into max accumulator
13.      }
14.      next; // Barrier before executing next iteration of iter loop
15.      double[] temp=myVal; myVal=myNew; myNew=temp; // Swap myVal and myNew
16.    } // while
17.  } // forasync
18.} // finish
```