COMP 322: Fundamentals of Parallel Programming

Lecture 15: Phaser Accumulators, Bounded Phasers

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

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

Goals for Today's Lecture

- <u>Phaser Accumulators</u>
- Bounded Phasers

Problem: Max reduction in One-Dimensional Iterative Averaging with Barrier Synchronization (from Lecture 13)

```
1.  double[] gVal=new double[n+2]; double[] gNew=new double[n+2]; gVal[n+1] = 1; gNew[n+1] =
    2;
2.  int Cj = Runtime.getNumOfWorkers();
3.  finish {
4.    ph = new phaser(phaserMode.SIG_WAIT);
5.    forasync (point [jj]:[0:Cj-1]) phased(ph) { // Explicit chunked forall
6.      double[] myVal = gVal; double[] myNew = gNew; // Local copies of pointers
7.      for (int iter = 0; iter < numIters; iter++) {
8.        // Compute MyNew as function of input array MyVal
9.        for (point [j]:getChunk([1:n],[Cj],[jj])) { // Iterate within chunk
10.         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
11.         // Compute normalized diff of element j w.r.t. converged value, j/(n+1)
12.         double nDiff = Math.abs(myNew[j]-myVal[j])/((double)j/(double)(n+1));
13.       }
14.       // QUESTION: how to compute max(nDiff) across all elements in this phase??
15.       next; // Barrier before executing next iteration of iter loop
16.       double[] temp=myVal; myVal=myNew; myNew=temp; // Swap myVal and myNew
17.     } // for iter
18.   } // forasync
19. } // finish
```

Finish Accumulators (Lecture 12) provide overall max value, not per-phase max value

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

Phaser Accumulators

- Phaser accumulators can accumulate values within a single phase e.g., between two "next" operations
- HJ provides different implementations for the same accumulator semantics
  - Eager: Concurrent atomic accumulation by multiple tasks
    - Optional delay function to reduce bus congestion in atomic updates
  - Dynamic-lazy: Sequential accumulation at synchronization point
  - Fixed-Lazy: Lightweight implementation of dynamic-lazy (limited dynamic parallelism)
- NOTE: 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 system

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)
  - an extra "true" parameter results in lazy accumulation as in finish accumulators e.g., `accumulator.factory.accumulator(op, type, phaser, true)`

- 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 Usage of Phaser Accumulator API

```
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++) {
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.}
```

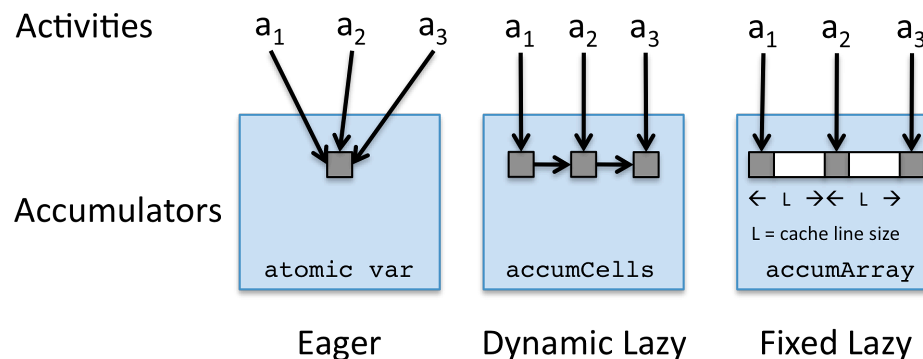
Allocation: Specify operator and type

put: Send a value to accumulator

get: Return accumulator result from previous phase

Different implementations for the accumulator API

- Eager
  - put: Update an atomic var in the accumulator
  - next: Store result from atomic var to read-only storage
- Dynamic-lazy
  - put: Put a value in accumCell
  - next: Perform reduction over accumCells
- Fixed-lazy
  - Like dynamic-lazy, but use accumArray instead of accumCells
  - Lightweight implementation due to primitive array access
  - For restricted case of bounded parallelism (up to array size)



Fixed-Lazy uses a fixed size array

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

Root

async

async

localMin localMin localMin

a.put() a.put() a.put()

next next next

Reduction

gblMin = a.get().doubleValue()

update update update

1st iteration of while loop

async async

localMin localMin localMin localMin localMin

a.put() a.put() a.put() a.put() a.put()

next next next next next

Reduction

gblMin= a.get().doubleValue()

2nd iteration

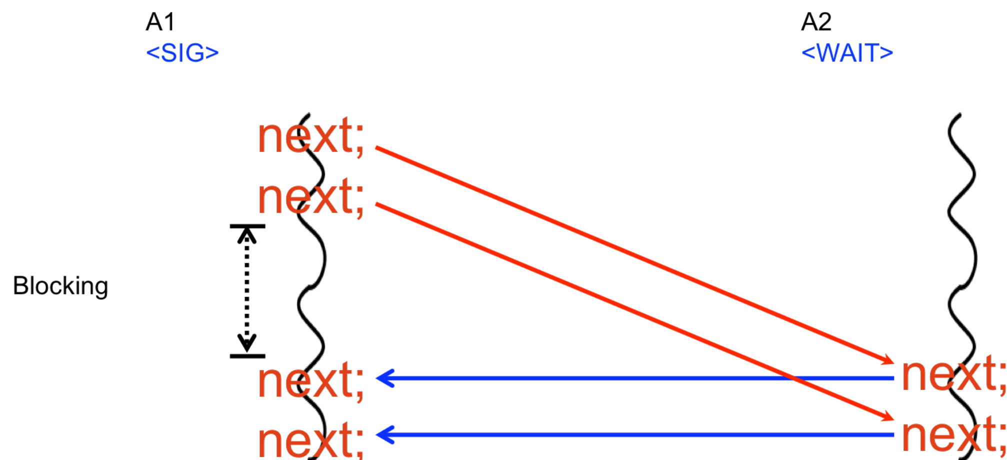
Goals for Today's Lecture

- Phaser Accumulators
- Bounded Phasers

Bound option in phasers

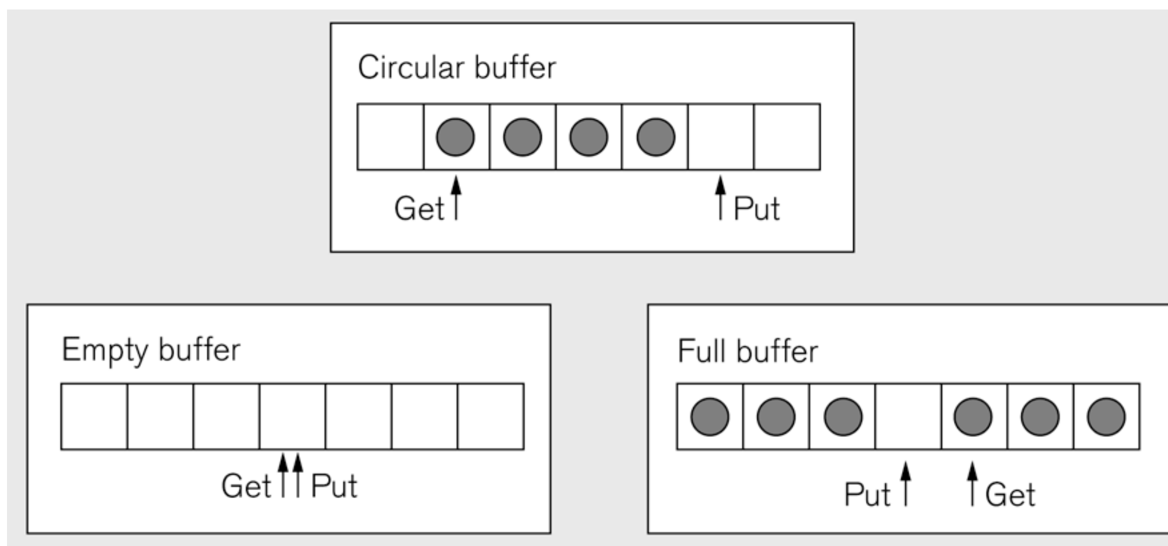
- Extra parameter in constructor
  - `new phaser(phaserMode m, int bound_size);`
- next operation
  - A task registered in SIG mode will block if it is >= bound_size phases past the current phase

```
...
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. }
```

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

T$_1$<SIG>

```
...
a.put(...);
next;
a.put(...);
next;
```

T$_2$<SIG>

```
...
a.put(...);
next;
a.put(...);
next;
```

bound=3

accumulator (sum ope)

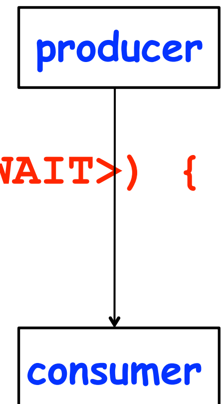
T$_3$<WAIT>

```
...
next;
prev = a.get(-1);
cur = a.get(0);
```

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

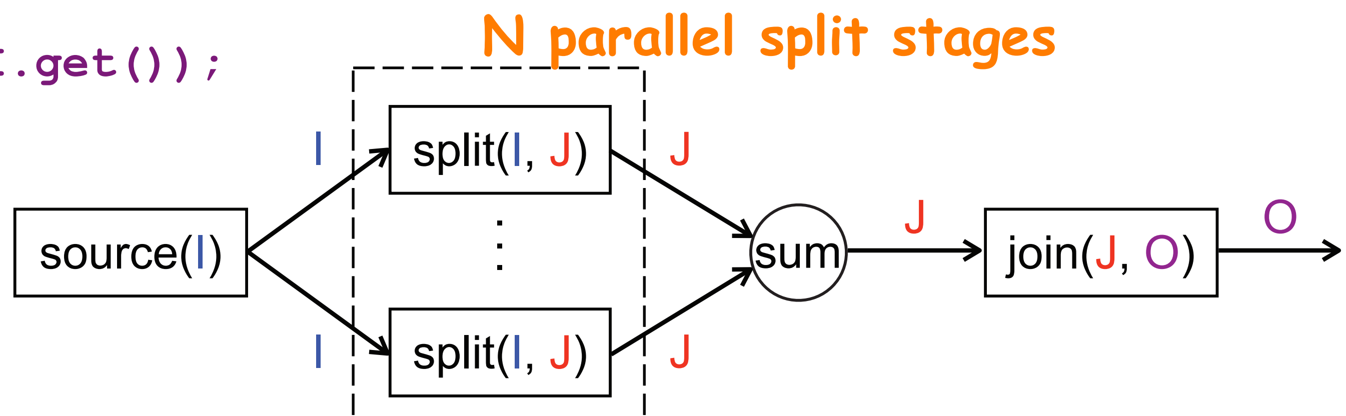


Streaming Computation: Pipeline

```
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;          // wait for two elements on I
        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;
} }
```

Streaming Patterns: Split-join

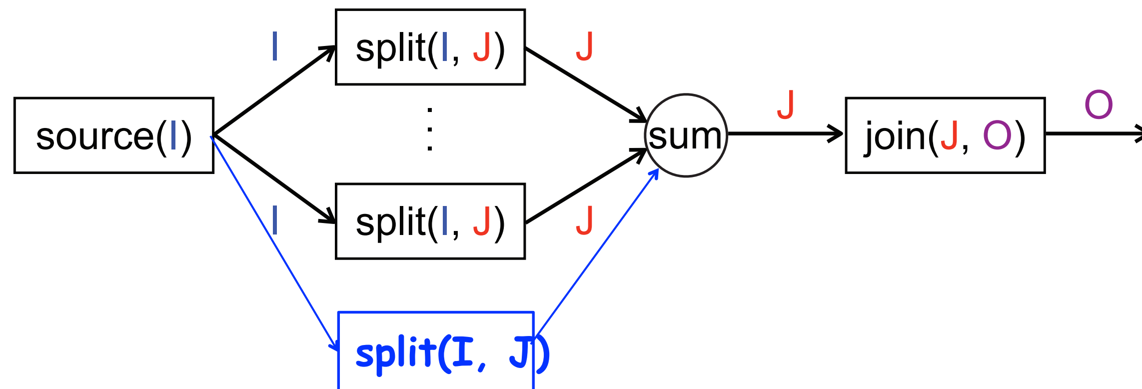
```
void Splitjoin() {
    phaser phI        = new phaser(SIG_WAIT, bnd);
    accumulator I     = new accumulator(phI, accumulator.ANY);
    phaser phJ        = new phaser(SIG_WAIT, bnd);
    accumulator J     = new accumulator(phJ, accumulator.SUM);

    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

Announcements (REMINDER)

- Homework 3 due on Wednesday, Feb 22nd
  - Performance results for parts 2 and 3 of assignment must be obtained on Sugar (see Section 4)
  - Start early --- you should complete the ideal parallel version this week
- No lab next week
  - Use the time for HW3 and to prepare for Exam 1
- Exam 1 will be held in the lecture on Friday, Feb 24th
  - Closed book 50-minute exam
  - Scope of exam includes lectures up to Monday, Feb 20th
  - Feb 22nd lecture will be a midterm review before exam
  - Contact me ASAP if you have an extenuating circumstance and need to take the midterm at an alternate time