COMP 322: Fundamentals of Parallel Programming

Lecture 18: Midterm Review

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

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

Worksheet #17: Critical Path Length for Computation with Signal Statement

Name: ______________________ Netid: ______________________

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

```
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
```

Async and Finish Constructs for Task Creation and Termination (Lecture 1)

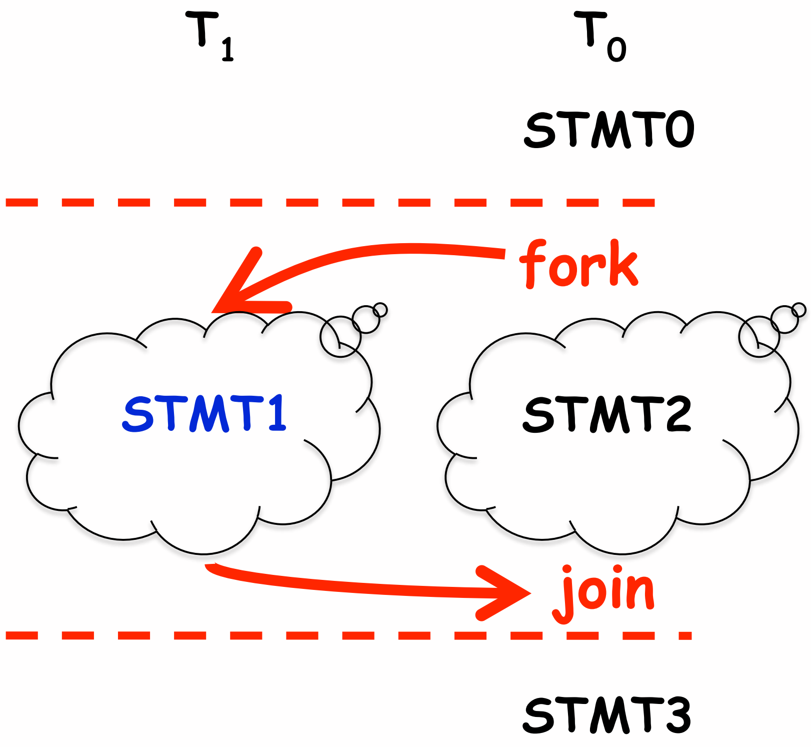
async S

- Creates a new child task that executes statement S

```
// T0(Parent task)
STMT0;
finish {    //Begin finish
   async {
      STMT1; //T1(Child task)
   }
   STMT2;    //Continue in T0
             //Wait for T1
}            //End finish
STMT3;       //Continue in T0
```

finish S

- Execute S, but wait until *all* asyncs in S's scope have terminated.



Acknowledgments: X10 and Habanero projects

Two-way Parallel Array Sum using HJ-Lib's finish & async API's

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
finish(() -> {
  async(() -> {
    // Child task computes sum of lower half of array
    for(int i=0; i < X.length/2; i++) sum1 += X[i];
  });
  // Parent task computes sum of upper half of array
  for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
});
// Parent task waits for child task to complete (join)
return sum1 + sum2;
```

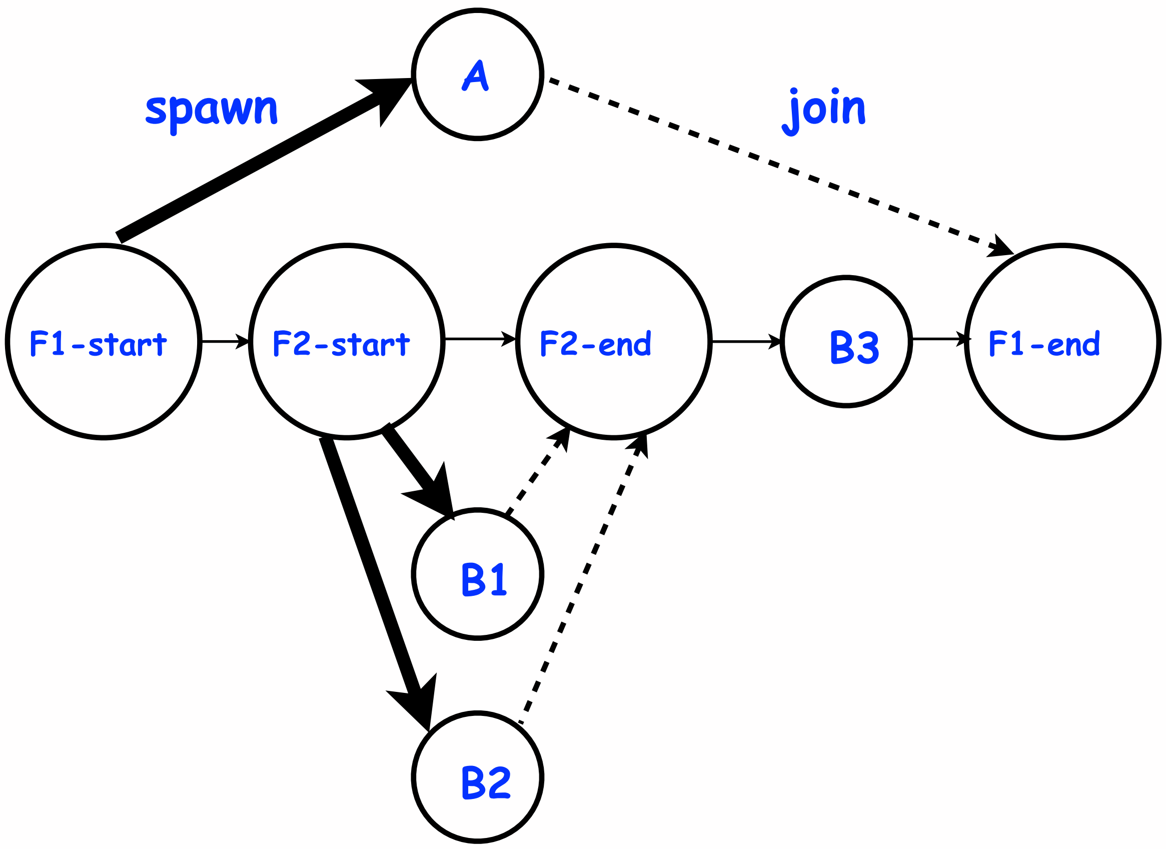
Computation Graphs (Lecture 2)

- A Computation Graph (CG) captures the dynamic execution of a parallel program, for a specific input
- CG nodes are "steps" in the program's execution
  - A step is a sequential subcomputation without any async, begin-finish and end-finish operations
- CG edges represent ordering constraints
  - "Continue" edges define sequencing of steps within a task
  - "Spawn" edges connect parent tasks to child async tasks
  - "Join" edges connect the end of each async task to its IEF's end-finish operations
- All computation graphs must be acyclic
  - It is not possible for a node to depend on itself
- Computation graphs are examples of "directed acyclic graphs" (dags)

Which statements can potentially be executed in parallel with each other?

```
1.  finish { // F1
2.    async A;
3.    finish { // F2
4.      async B1;
5.      async B2;
6.    } // F2
7.    B3;
8.  } // F1
```

Computation Graph



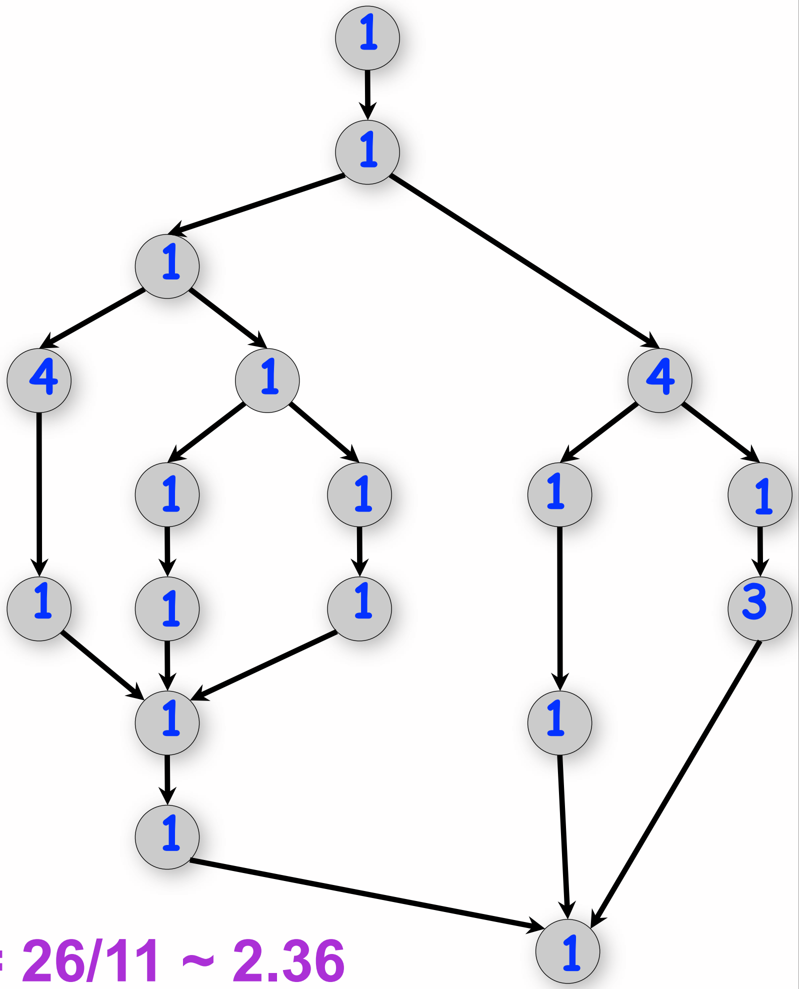
Complexity Measures for Computation Graphs

Define

- TIME(N) = execution time of node N
- WORK(G) = sum of TIME(N), for all nodes N in CG G
  - WORK(G) is the total work to be performed in G
- CPL(G) = length of a longest path in CG G, when adding up execution times of all nodes in the path
  - Such paths are called *critical paths*
  - CPL(G) is the length of these paths (critical path length)
  - CPL(G) is also the smallest possible execution time for the computation graph

Ideal Parallelism

- Define ideal parallelism of Computation G Graph as the ratio, WORK(G)/CPL(G)
- Ideal Parallelism is independent of the number of processors that the program executes on, and only depends on the computation graph



Example:

WORK(G) = 26

CPL(G) = 11

Ideal Parallelism = WORK(G)/CPL(G) = 26/11 ~ 2.36

Bounding the performance of Greedy Schedulers (Lecture 3)

Combine lower and upper bounds to get

$\max(WORK(G)/P, CPL(G)) \le T_P \le WORK(G)/P + CPL(G)$

Corollary 1: Any greedy scheduler achieves execution time T_P that is within a factor of 2 of the optimal time (since max(a,b) and (a+b) are within a factor of 2 of each other, for any $a \ge 0, b \ge 0$).

Corollary 2: Lower and upper bounds approach the same value whenever

- There's lots of parallelism, $WORK(G)/CPL(G) >> P$
- Or there's little parallelism, $WORK(G)/CPL(G) << P$

Amdahl's Law [1967] (Lecture 4)

- If $q \leq 1$ is the fraction of WORK in a parallel program that <u>must be executed sequentially</u> for a given input size S, then the best speedup that can be obtained for that program is $\text{Speedup}(S,P) \leq 1/q$.
- Observation follows directly from critical path length lower bound on parallel execution time
  - $\text{CPL} >= q * T(S,1)$
  - $T(S,P) >= q * T(S,1)$
  - $\text{Speedup}(S,P) = T(S,1)/T(S,P) <= 1/q$
- This upper bound on speedup simplistically assumes that work in program can be divided into sequential and parallel portions
  - Sequential portion of WORK = q
    - also denoted as f_S (fraction of sequential work)
  - Parallel portion of WORK = 1-q
    - also denoted as f_p (fraction of parallel work)
- Computation graph is more general and takes dependences into account

Extending Async Tasks with Return Values (Lecture 5)

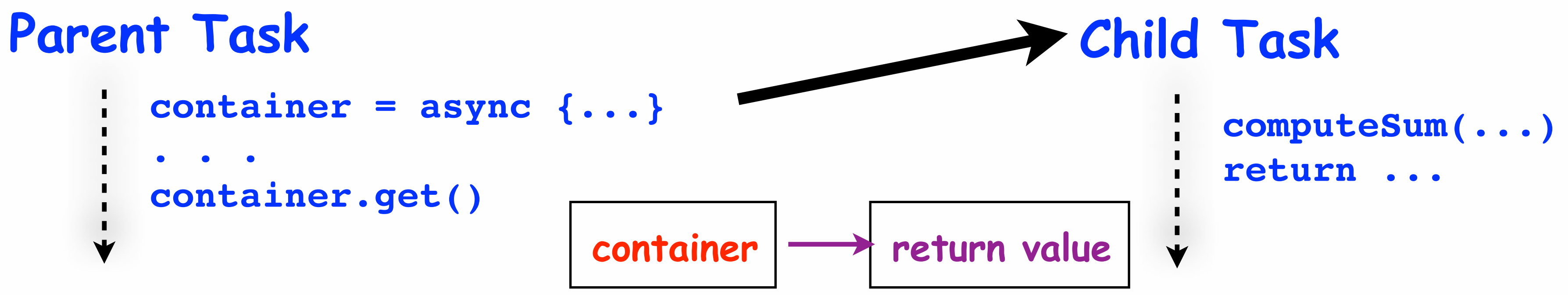
- Example Scenario in PseudoCode

```
// Parent task creates child async task
final future container =
  async { return computeSum(X, low, mid); };
. . .
// Later, parent examines the return value
int sum = container.get();
```

- Two issues to be addressed:

1) Distinction between container and value in container (box)

2) Synchronization to avoid race condition in container accesses



HJ Futures: Tasks with Return Values

`async { Stmt-Block }`

- Creates a new child task that executes Stmt-Block, which must terminate with a return statement and return value
- Async expression returns a reference to a container of type future

`Expr.get()`

- Evaluates Expr, and blocks if Expr's value is unavailable
- Unlike finish which waits for *all* tasks in the finish scope, a get() operation only waits for the specified async expression

Example: Two-way Parallel Array Sum using Future Tasks in HJ-Lib

```
// Parent Task T1 (main program)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final HjFuture sum1 = future (() -> { // Future Task T2
  int sum = 0;
  for(int i=0 ; i < X.length/2 ; i++) sum += X[i];
  return sum;
});
final HjFuture sum2 = future (() ->{ // Future Task T3
  int sum = 0;
  for(int i=X.length/2 ; i < X.length ; i++) sum += X[i];
  return sum;
});
//Task T1 waits for Tasks T2 and T3 to complete
int total = sum1.get() + sum2.get();
```

Use of Finish Accumulators to count solutions in Parallel NQueens (Lecture 6)

```
final FinishAccumulator ac =
                          newFinishAccumulator(Operator.SUM, int.class);
finish(ac) nqueens_kernel(new int[0], 0);
System.out.println(“No. of solutions = “ + ac.get().intValue())
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) ac.put(1);
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) async {
      /* allocate a temporary array and copy array a into it */
      int [] b = new int [depth+1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = i;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```

Functional vs. Structural Determinism (Lecture 7)

- 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
- Race-Free Determinism
  - If a parallel program is written using the constructs learned so far (finish, async, futures) and is known to be race-free, *then it must be both functionally deterministic and structurally deterministic*

A Classification of Parallel Programs

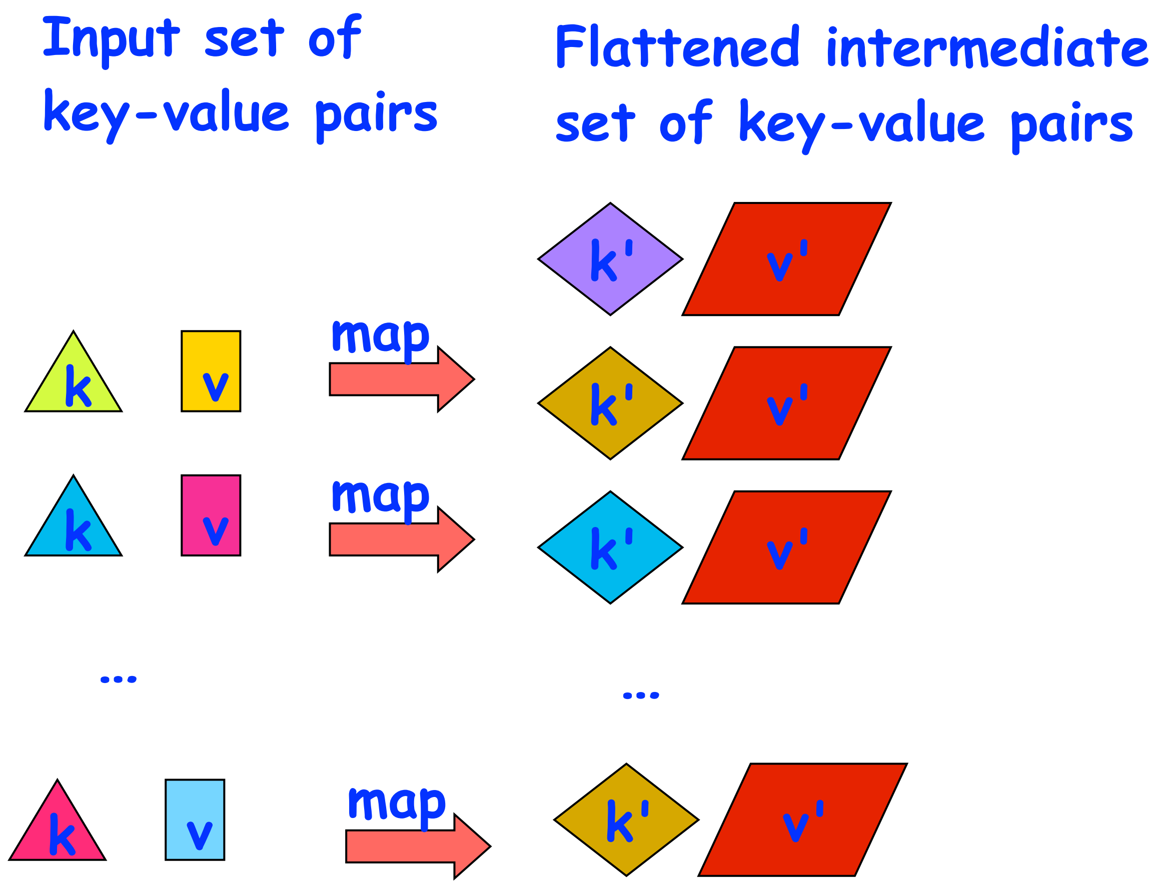
Data Race Free?	Functionally Deterministic?	Structurally Deterministic?	Example: String Search variation
Yes	Yes	Yes	Count of all occurrences
No	Yes	Yes	Existence of an occurrence
No	No	Yes	Index of any occurrence
No	Yes	No	"Eureka" extension for existence of an occurrence: do not create more async tasks after occurrence is found
No	No	No	"Eureka" extension for index of an occurrence: do not create more async tasks after occurrence is found

Data-Race-Free Determinism Property implies that it is not possible to write an HJ program with Yes in column 1, and No in column 2 or column 3 (when only using Module 1 constructs)

Map Reduce: Summary (Lecture 8)

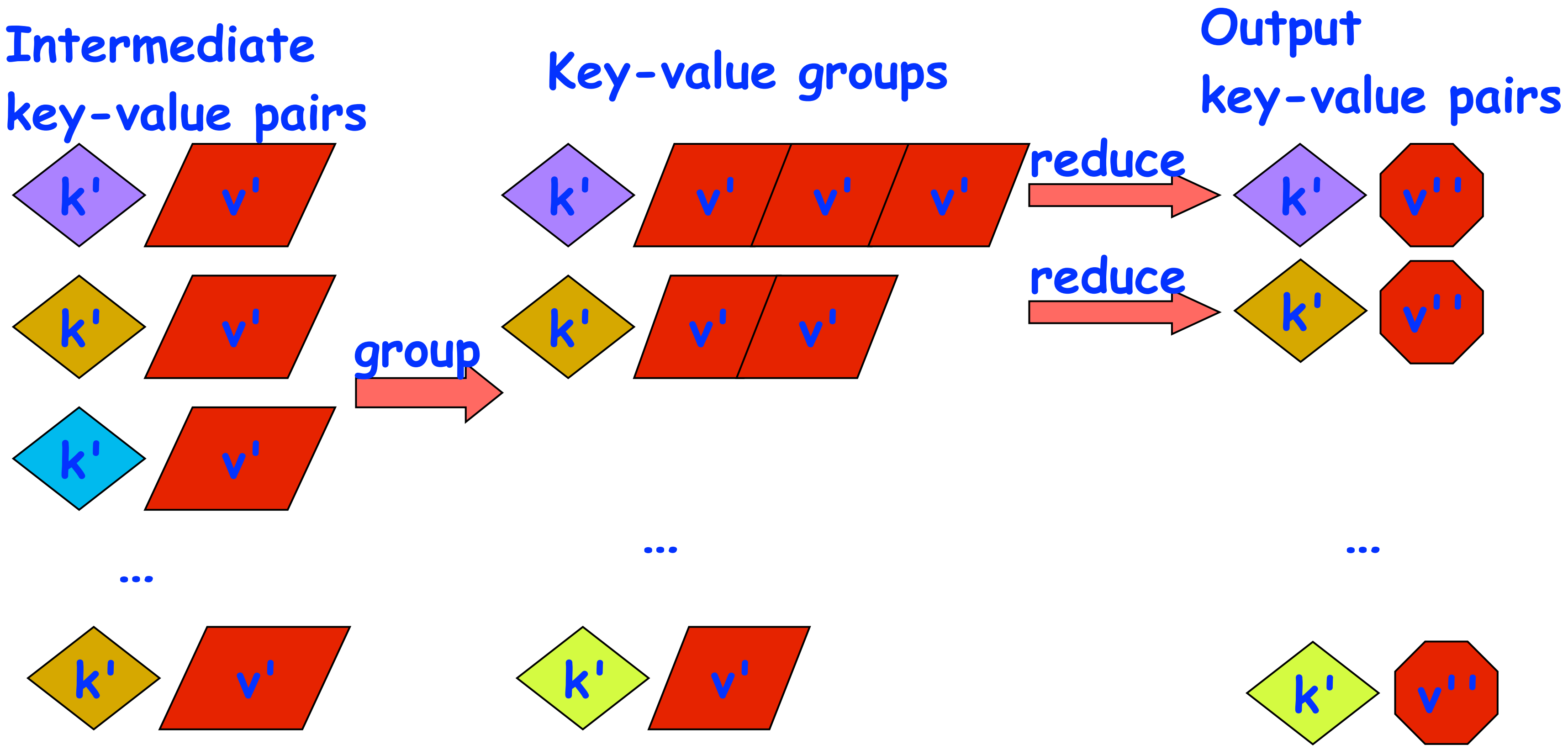
- Input set is of the form {(k1, v1), . . . (kn, vn)}, where (ki, vi) consists of a key, ki, and a value, vi.
  - Assume that the key and value objects are immutable, and that equality comparison is well defined on all key objects.
- Map function f generates sets of intermediate key-value pairs, f(ki,vi) = {(k1' ,v1'),...(km',vm')}. The kj' keys can be different from ki key in the input of the map function.
  - Assume that a flatten operation is performed as a post-pass after the map operations, so as to avoid dealing with a set of sets.
- Reduce operation groups together intermediate key-value pairs, {(k', vj')} with the same k', and generates a reduced key-value pair, (k',v''), for each such k', using reduce function g

MapReduce: The Map Step


Input set of key-value pairs
Flattened intermediate set of key-value pairs
k
v
map
k'
v'
k
v
map
k'
v'
k'
v'
k'
v'
...
...
k
v
map
k'
v'

Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

MapReduce: The Reduce Step



Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

seq clause for async statements (Lecture 10)

```
async seq(cond) <stmt> ≡ if (cond) <stmt> else async <stmt>
```

```
// Async task
async seq(size < thresholdSize) computeSum(X, lo, mid);

// Future example
final future<int> sum1 = future seq(size < thresholdSize)
                                 { return computeSum(X, lo, mid); };
```

- **“seq” clause specifies condition under which async should be executed sequentially**
  - **False ⇒ an async is created**
  - **True ⇒ the parent executes async body sequentially**
- **Avoids the need to duplicate code for both cases**

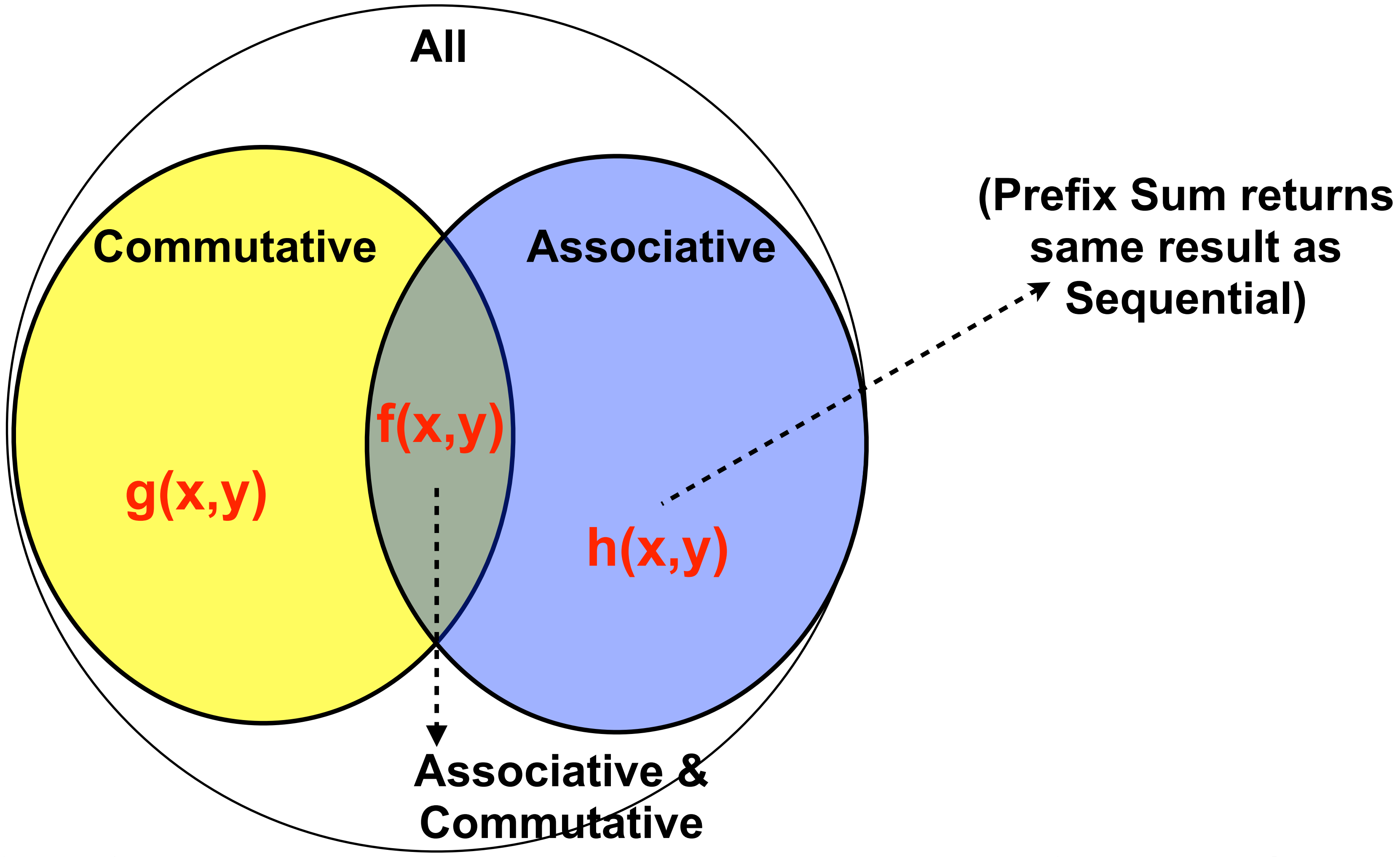
Use of asyncSeq API in HJlib (Quicksort example)

```
protected static void quicksort(
    final Comparable[] A, final int M, final int N) {
        if (M < N) {
            // A point in HJ is an integer tuple
            HjPoint p = partition(A, M, N);
            int I = p.get(0);
            int J = p.get(1);
            asyncSeq(I - M <= 5, () -> quicksort(A, M, I));
            asyncSeq(N - J <= 5, () -> quicksort(A, J, N));
         }
    }
```

Nqueens example with seq clause

```
void nqueensKernel(final int[] a, final int depth,
            final FinishAccumulator ac) {
  if (size == depth) {
    ac.put(1);
    return;
  }
  /* try each possible position for queen <depth> */
  for (int i = 0; i < size; i++) {
    final int ii = i;
    asyncSeq(depth >= cutoff_value, () -> {
      /* allocate a temporary array and copy <a> into it */
      final int[] b = new int[depth + 1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = ii;
      if (boardValid((depth + 1), b)) {
        nqueensKernel(b, depth + 1, ac);
      }
    });
  }
}
```

Venn diagram of binary functions


All
Commutative
Associative
f(x,y)
g(x,y)
h(x,y)
(Prefix Sum returns same result as Sequential)
Associative & Commutative

(Prefix Sum & Finish Accumulator return same result as Sequential)

Observations on finish-for-async version (Lecture 11)

- finish and async are general constructs, and are not specific to loops
  - Not easy to discern from a quick glance which loops are sequential vs. parallel
- Loops in sequential version of matrix multiplication are “perfectly nested”
  - e.g., no intervening statement between “for(i = ...)” and “for(j = ...)”
- The ordering of loops nested between finish and async is arbitrary
  - They are parallel loops and their iterations can be executed in any order

forall API's in HJlib
(http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/Module1.html)

- `static void forall(edu.rice.hj.api.HjRegion.HjRegion1D hjRegion, edu.rice.hj.api.HjProcedureInt1D body)`
- `static void forall(edu.rice.hj.api.HjRegion.HjRegion2D hjRegion, edu.rice.hj.api.HjProcedureInt2D body)`
- `static void forall(edu.rice.hj.api.HjRegion.HjRegion3D hjRegion, edu.rice.hj.api.HjProcedureInt3D body)`
- `static void forall(int s0, int e0, edu.rice.hj.api.HjProcedure<java.lang.Integer> body)`
- `static void forall(int s0, int e0, int s1, int e1, edu.rice.hj.api.HjProcedureInt2D body)`
- `static <T> void forall(java.lang.Iterable<T> iterable, edu.rice.hj.api.HjProcedure<T> body)`
- **NOTE: all forall API's include an implicit finish. forasync is like forall, but without the finish.**

forall examples: updates to a two-dimensional Java array

```
// Case 1: loops i,j can run in parallel
forall(0, m-1, 0, n-1, (i, j) -> { A[i][j] = F(A[i][j]);});

// Case 2: only loop i can run in parallel
forall(1, m-1, (i) -> {
  forseq(1, n-1, (j) -> { // Equivalent to “for (j=1;j<n;j++)”
     A[i][j] = F(A[i][j-1]) ;
}); });

// Case 3: only loop j can run in parallel
forseq(1, m-1, (i) -> { // Equivalent to “for (i=1;i<m;j++)”
  forall(1, n-1, (j) -> {
     A[i][j] = F(A[i-1][j]) ;
}); });
```

One-Dimensional Iterative Averaging Example

- **Initialize a one-dimensional array of (n+2) double's with boundary conditions, myVal[0] = 0 and myVal[n+1] = 1.**
- **In each iteration, each interior element myVal[i] in 1..n is replaced by the average of its left and right neighbors.**
  - **Two separate arrays are used in each iteration, one for old values and the other for the new values**
- **After a sufficient number of iterations, we expect each element of the array to converge to myVal[i] = i/(n+1)**
  - **In this case, myVal[i] = (myVal[i-1]+myVal[i+1])/2, for all i in 1..n**

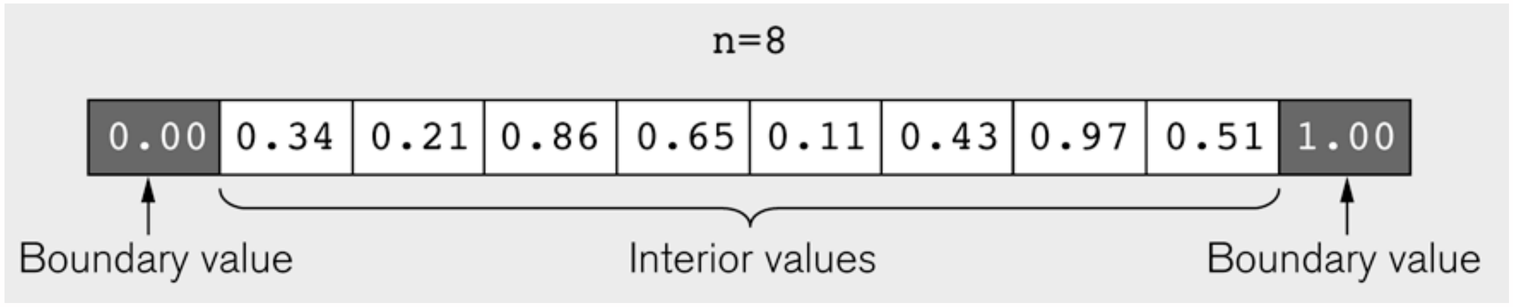


Illustration of an intermediate step for n = 8 (source: Figure 6.19 in Lin-Snyder book)

Example: HJ code for One-Dimensional Iterative Averaging with forseq-forall structure w/ chunking

```
int nc = numWorkerThreads();
forseq(0, m-1, (iter) -> {
  // Compute MyNew as function of input array MyVal
    forallChunked(1, n, n/nc, (j) -> { // Create n tasks
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    }); // forall
  temp=myVal; myVal=myNew; myNew=temp;// Swap myVal & myNew;
  // myNew becomes input array for next iteration
}); // for
```

Barriers (Lecture 12)

- Question: how can we transform this code so as to ensure that all tasks say hello before any tasks say goodbye?
- Approach 2: insert a "barrier" between the hello's and goodbye's
  - "next" statement in HJ's forall loops

```
// APPROACH 2
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);
});
```

Phase 0

Phase 1

- next ➔ each forall iteration suspends at next until all iterations arrive (complete previous phase), after which the phase can be advanced
  - If a forall iteration terminates before executing "next", then the other iterations do not wait for it
  - Scope of next is the closest enclosing forall statement
  - Special case of "phaser" construct (will be covered later in class)

Observation 1: Scope of synchronization for “next” is closest enclosing forall statement

```
forallPhased (0, m - 1, (i) -> {
 println(“Starting forall iteration ” + i);
 next(); // Acts as barrier for forall-i
 forallPhased (0, n - 1, (j) -> {
   println(“Hello from task (“ + i + “,” + j + “)”);
   next(); // Acts as barrier for forall-j
   println(“Goodbye from task (“ + i + “,” + j + “)”);
 } // forall-j
 next(); // Acts as barrier for forall-i
 println(“Ending forall iteration ” + i);
}); // forall-i
```

Observation 2: If a forall iteration terminates before "next", then other iterations do not wait for it

```
1.  forallPhased (0, m - 1, (i) -> {
2.    forseq (0, i, (j) -> {
3.      // forall iteration i is executing phase j
4.      System.out.println("(" + i + "," + j + ")");
5.      next();
6.    });
7.  });
```

- Outer forall-i loop has m iterations, 0…m-1
- Inner sequential j loop has i+1 iterations, 0…i
- Line 4 prints (task,phase) = (i, j) before performing a next operation.
- Iteration i = 0 of the forall-i loop prints (0, 0), performs a next, and then terminates. Iteration i = 1 of the forall-i loop prints (1,0), performs a next, prints (1,1), performs a next, and then terminates. And so on.

HJ code for One-Dimensional Iterative Averaging with grouped forall-forseq structure and barriers (Lecture 13)

```
double[] gVal=new double[n+2]; gVal[n+1] = 1;
double[] gNew=new double[n+2];
HjRegion1D iterSpace = newRectangularRegion1D(1,m);
int nc = numWorkerThreads();
forallPhased(1, nc, (jj) -> { // Create nc tasks
  // Initialize myVal and myNew as local pointers
  double[] myVal = gVal; double[] myNew = gNew;
  forseq(0, m-1, (iter) -> {
    forseq(myGroup(jj,iterSpace,nc), (j) -> {
      // Compute MyNew as function of input array MyVal
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    }); // forseq
    next(); // Barrier before executing next iteration of iter loop
    // Swap local pointers, myVal and myNew
    double[] temp=myVal; myVal=myNew; myNew=temp;
    // myNew becomes input array for next iter
  }); // forseq
}); // forall
```

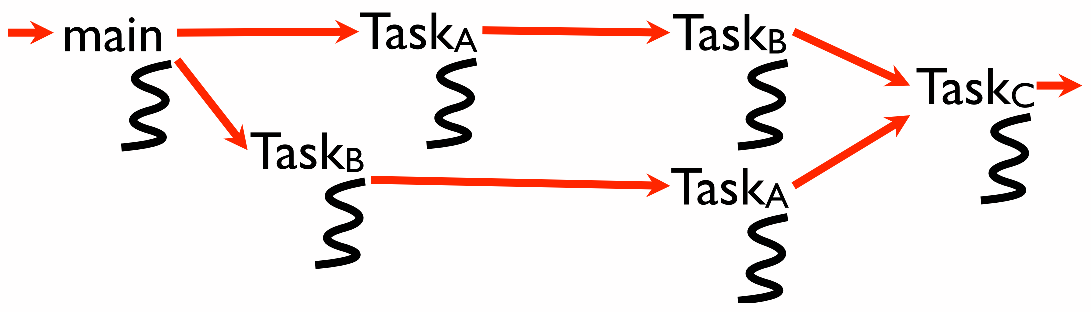
This program creates nc async tasks, and performs *m* barrier operations per task

Single Program Multiple Data (SPMD) Parallel Programming Model

Basic idea

- **Run the same code (program) on P workers**
- **Use the "rank" --- an ID ranging from 0 to (P-1) --- to determine what data is processed by which worker**
  - **Hence, "single-program" and "multiple-data"**
  - **Rank is equivalent to index in a top-level "forall (point[i] : [0:P-1])" loop**
- **Lower-level programming model than dynamic async/finish parallelism**
  - **Programmer's code is essentially at the worker level (each forall iteration is like a worker), and work distribution is managed by programmer by using barriers and other synchronization constructs**
  - **Harder to program but can be more efficient for restricted classes of applications (e.g. for OneDimAveraging, but not for nqueens)**
- **Convenient for hardware platforms that are not amenable to efficient dynamic task parallelism**
  - **General-Purpose Graphics Processing Unit (GPGPU) accelerators**
  - **Distributed-memory parallel machines**

Macro-Dataflow Programming (Lecture 14)



- "Macro-dataflow" = extension of dataflow model from instruction-level to task-level operations
- General idea: build an arbitrary task graph, but restrict all inter-task communications to single-assignment variables
    - Static dataflow ==> graph fixed when program execution starts
    - Dynamic dataflow ==> graph can grow dynamically
- Semantic guarantees: race-freedom, determinism
    - Deadlocks are possible due to unavailable inputs (but they are deterministic)

Extending HJ Futures for Macro-Dataflow: Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)

```
HjDataDrivenFuture<T1> ddfA = newDataDrivenFuture();
```

- Allocate an instance of a data-driven-future object (container)
- Object in container must be of type T1

```
asyncAwait(ddfA, ddfB, …, () -> Stmt);
```

- Create a new data-driven-task to start executing Stmt after all of ddfA, ddfB, … become available (i.e., after task becomes "enabled")

```
ddfA.put(V) ;
```

- Store object V (of type T1) in ddfA, thereby making ddfA available
- Single-assignment rule: at most one put is permitted on a given DDF

ddfA.get()

- Return value (of type T1) stored in ddfA
- Can only be performed by async's that contain ddfA in their await clause (hence no blocking is necessary for DDF gets)

Differences between Futures and DDFs/DDTs

- **Consumer task blocks on get() for each future that it reads, whereas async-await does not start execution till all DDFs are available**
- **Future tasks cannot deadlock, but it is possible for a DDT to block indefinitely ("deadlock") if one of its input DDFs never becomes available**
- **DDTs and DDFs are more general than futures**
  - **Producer task can only write to a single future object, where as a DDT can write to multiple DDF objects**
  - **The choice of which future object to write to is tied to a future task at creation time, where as the choice of output DDF can be deferred to any point with a DDT**
- **DDTs and DDFs can be more implemented more efficiently than futures**
  - **An "asyncAwait" statement does not block the worker, unlike a future.get()**
  - **You will never see the following message with "asyncAwait"**
    - "ERROR: Maximum number of hj threads per place reached"

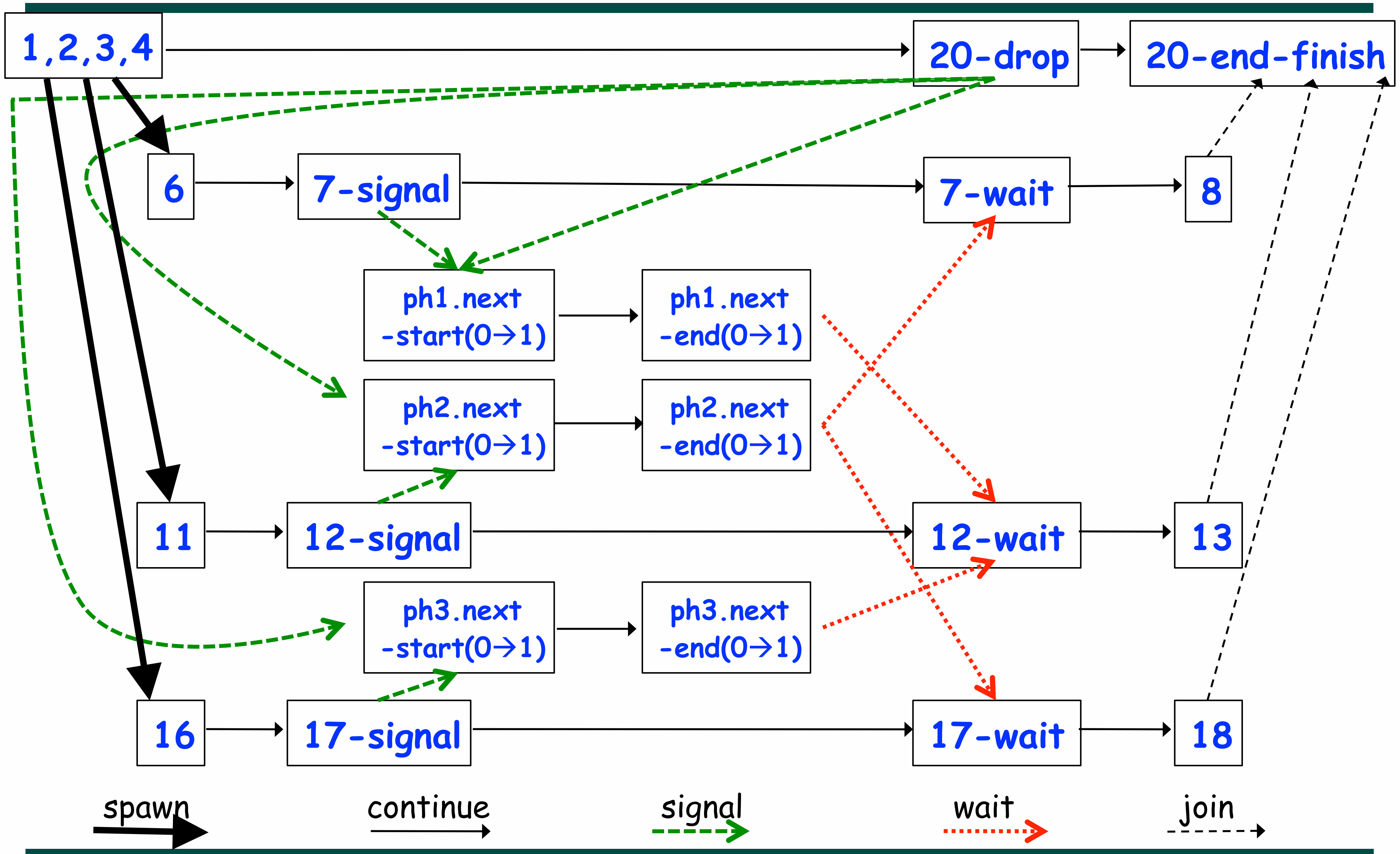
Summary of Phaser Construct (Lecture 16)

- Phaser allocation
  - HjPhaser ph = newPhaser(mode);
    - Phaser ph is allocated with registration mode
    - Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)
- Registration Modes
  - HjPhaserMode.SIG, HjPhaserMode.WAIT, HjPhaserMode.SIG_WAIT, HjPhaserMode.SIG_WAIT_SINGLE
    - NOTE: phaser WAIT is unrelated to Java wait/notify (which we will study later)
- Phaser registration
  - asyncPhased (ph_1.inMode(<$mode_1$>), ph_2.inMode(<$mode_2$>), … () -> <stmt>)
    - Spawned task is registered with ph_1 in $mode_1$, ph_2 in $mode_2$, …
    - Child task's capabilities must be subset of parent's
    - asyncPhased <stmt> propagates all of parent's phaser registrations to child
- Synchronization
  - next();
    - Advance each phaser that current task is registered on to its next phase
      All signals are performed, followed by all waits
    - Semantics depends on registration mode
    - Barrier is a special case of phaser, which is why *next* is used for both

Left-Right Neighbor Synchronization Example for m=3

```
finish(() -> { // Task-0
    final HjPhaser ph1 = newPhaser(SIG_WAIT);
    final HjPhaser ph2 = newPhaser(SIG_WAIT);
    final HjPhaser ph3 = newPhaser(SIG_WAIT);
    asyncPhased(ph1.inMode(SIG),ph2.inMode(WAIT),
      () -> { doPhase1(1);
        next(); // signals ph1, waits on ph2
        doPhase2(1);
    }); // Task T1
    asyncPhased(ph2.inMode(SIG),ph1.inMode(WAIT),ph3.inMode(WAIT),
      () -> { doPhase1(2);
         next(); // signals ph2, waits on ph3
         doPhase2(2);
     }); // Task T2
     asyncPhased(ph3.inMode(SIG),ph2.inMode(WAIT),
       () -> { doPhase1(3);
         next(); // signals ph3, waits on ph2
         doPhase2(3);
     }); // Task T3
}); // finish
```

Computation Graph for m=3 example



Announcements

- Take-home midterm exam (Exam 1) will be given after lecture on Wednesday, February 26, 2014
  - Closed-book, closed computer, written exam that can be taken in any 2-hour duration during that period
  - Will need to be returned to Penny Anderson (Duncan Hall 3080) by 4pm on Friday, February 28, 2014
    - Exam can also be picked up from Penny Anderson starting 2pm on Feb 26th if you're unable to attend lecture.
  - No lecture on Friday, Feb 28th
- Homework 3 is due by by 11:59pm on Wednesday, March 12, 2014
  - Programming assignment is more challenging than in previous homeworks --- start early!

Scope of Midterm Exam

- **Midterm exam will cover material from Lectures 1 - 17**
  - —Lecture 18 (Feb 26th) will be a Midterm review
- **Excerpts from midterm exam instructions**
  - —"closed-book, closed-notes, closed-computer"
  - —"Record start time when you open the exam, and end time when you finish. The total duration must be at most 2 hours. "
  - —"Since this is a written exam and not a programming assignment, syntactic errors in program text will not be penalized (e.g., missing semicolons, incorrect spelling of keywords, etc) so long as the meaning of your solution is unambiguous."
  - —"If you believe there is any ambiguity or inconsistency in a question, you should state the ambiguity or inconsistency that you see, as well as any assumptions that you make to resolve it."