COMP 322: Fundamentals of Parallel Programming

Lecture 16: Midterm Review

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

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

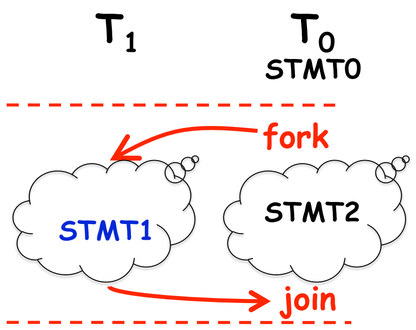
async S

- Creates a new child task that executes statement S

finish S

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

```
// 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
```



One possible solution to Worksheet 1 (without statement reordering)

```
1. finish {
2.   async { Watch COMP 322 video for topic 1.2 by 1pm on Wednesday
3.           Watch COMP 322 video for topic 1.3 by 1pm on Wednesday
4.          }
5.   async Make your bed
6.   async { Clean out your fridge
7.           Buy food supplies and store them in fridge }
8.   finish { async Run load 1 in washer
9.            Run load 2 in washer }
10.   async Run load 1 in dryer
11.   async Run load 2 in dryer
12.   async Call your family
13. }
14. Post on Facebook that you're done with all your tasks!
```

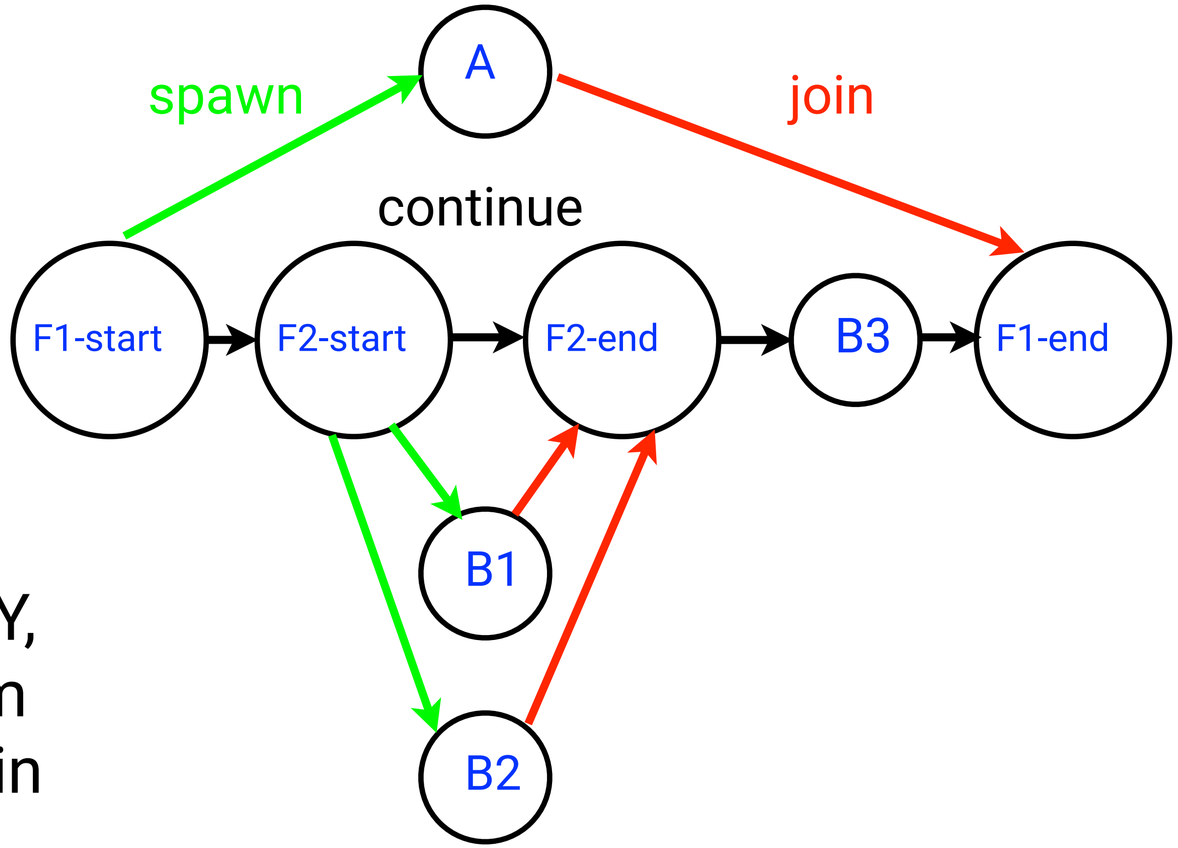
Computation Graphs (Lecture 2)

- 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 or 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



<u>Key idea</u>: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.

Complexity Measures for Computation Graphs

Define

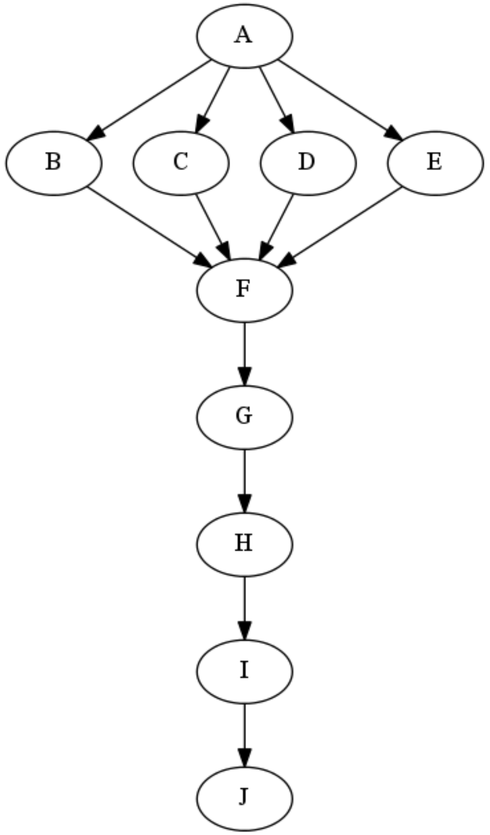
- TIME(N) = execution time of node N
- WORK(G) = sum of TIME(N), for all nodes N in CG G
  - WORK(G) is the total work to be performed in G
- CPL(G) = length of a longest path in CG G, when adding up execution times of all nodes in the path
  - Such paths are called *critical paths*
  - CPL(G) is the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

Which Computation Graph has more ideal parallelism?

Assume that all nodes have TIME = 1, so WORK = 10 for both graphs.

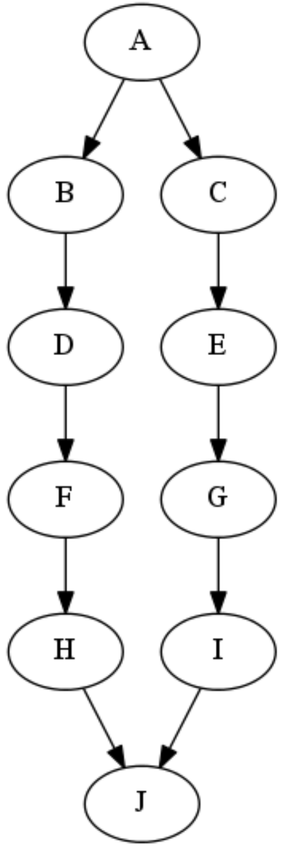
Computation Graph 1

CPL = 7



Computation Graph 2

CPL = 6



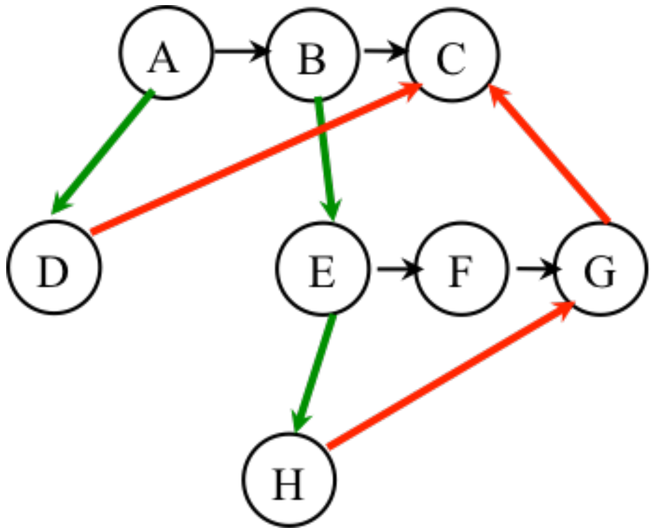
Data Races

A data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) S1 and S2 in CG such that:

1. S1 does not depend on S2 and S2 does not depend on S1, i.e., S1 and S2 can potentially execute in parallel, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write.

- A data-race is usually considered an error. The result of a read operation in a data race is undefined. The result of a write operation is undefined if there are two or more writes to the same location.
  - Note that our definition of data race includes the case that both S1 and S2 write the same value in location L, even if that may not be considered an error.
- Above definition includes all "potential" data races i.e., we consider it to be a data race even if S1 and S2 end up executing on the same processor.

One Possible Solution to Worksheet 2 (Reverse Engineering a Computation Graph)




```
1.A();
2.finish { // F1
3.   async D();
4.   B();
5.   E();
6.   finish { // F2
7.     async H();
8.     F();
9.   } // F2
10. G();
11.} // F1
12.C();
```

Observations:

- Any node with out-degree > 1 must be an async (must have an outgoing spawn edge)
- Any node with in-degree > 1 must be an end-finish (must have an incoming join edge
- Adding or removing transitive edges does not impact ordering constraints

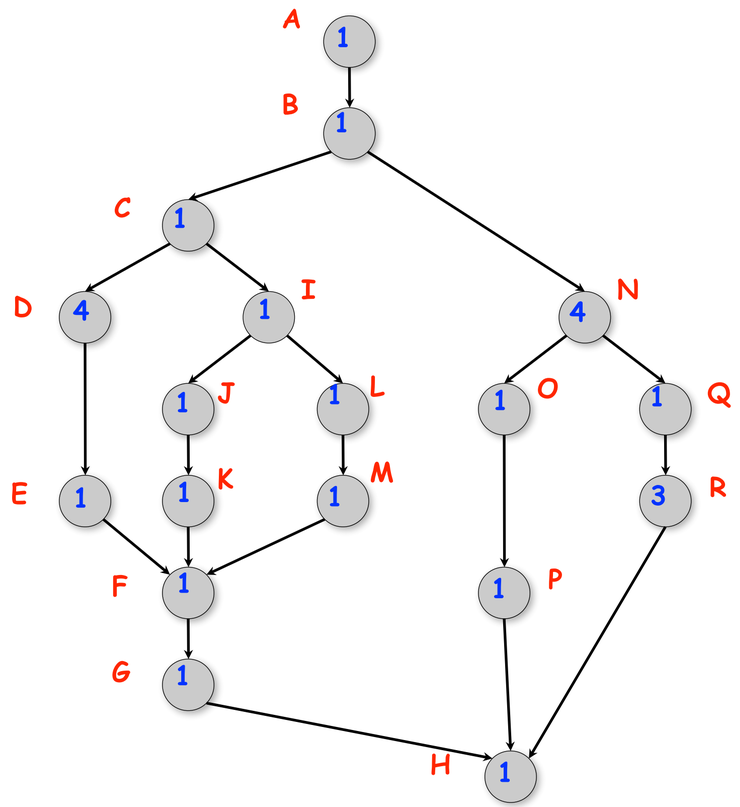
Bounding the Performance of Greedy Schedulers (Lecture 3)

Combine lower and upper bounds to get

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

Corollary: 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 $a \geq 0, b \geq 0$).

One Possible Solution to Worksheet 3 (Multiprocessor Scheduling)



Start time	Proc 1	Proc 2
0	A	
1	B	
2	C	N
3	D	N
4	D	N
5	D	N
6	D	O
7	I	Q
8	J	R
9	L	R
10	K	R
11	M	E
12	F	P
13	G	
14	H	
15		

- As before, WORK = 26 and CPL = 11 for this graph
- T_2 = 15, for the 2-processor schedule on the right
- We can also see that max(CPL,WORK/2) <= T_2 < CPL + WORK/2
- There are 4 idle slots in this schedule — can we do better than T_2 = 15 ?

Solution to Worksheet 4

- Estimate T(S,P) ~ WORK(G,S)/P + CPL(G,S) = (S-1)/P + log2(S) for the parallel array sum computation shown in slide 4.
- Assume S = 1024 ==> log2(S) = 10
- Compute for 10, 100, 1000 processors
  - T(P) = 1023/P + 10, for P > 1
  - Speedup(10) = T(1)/T(10) = 1023/112.3 ~ 9.2
  - Speedup(100) = T(1)/T(100) = 1023/20.2 ~ 51.1
  - Speedup(1000) = T(1)/T(1000) = 1023/11.0 ~ 93.7
- Why does the speedup not increase linearly in proportion to the number of processors?
  - Because of the critical path length, log2(S), is a bottleneck

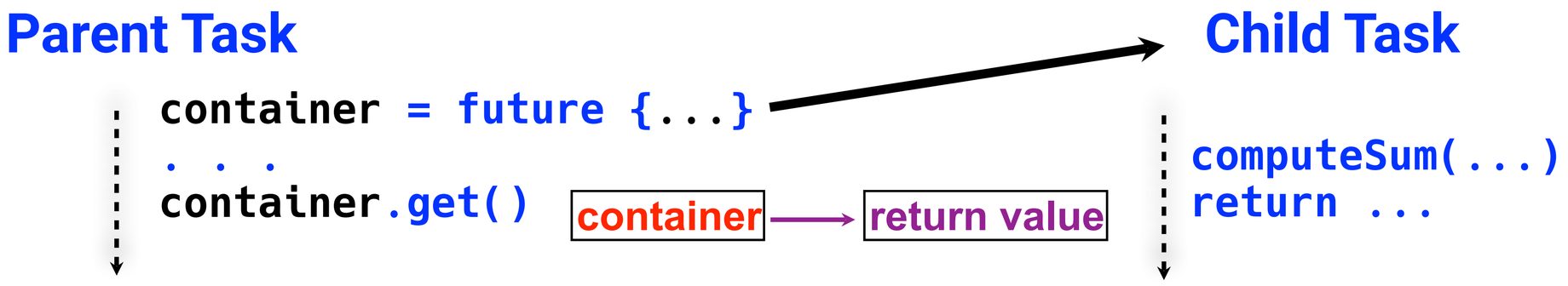
Functional Parallelism: Adding Return Values to Async Tasks (Lecture 5)

Example Scenario (PseudoCode)

```
// Parent task creates child async task
future<Integer> container = future { return computeSum(X,low,mid); };
. . .
// Later, parent examines the return value
Integer 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



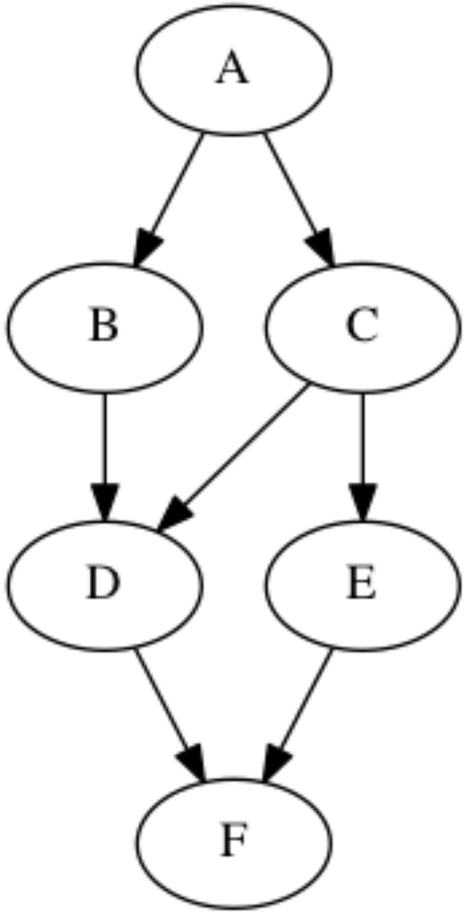
Worksheet #5: Computation Graphs for Async-Finish and Future Constructs

1) Can you write pseudocode with <u>async-finish</u> constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.

No. Finish cannot be used to ensure that D only waits for B and C, while E waits only for C.

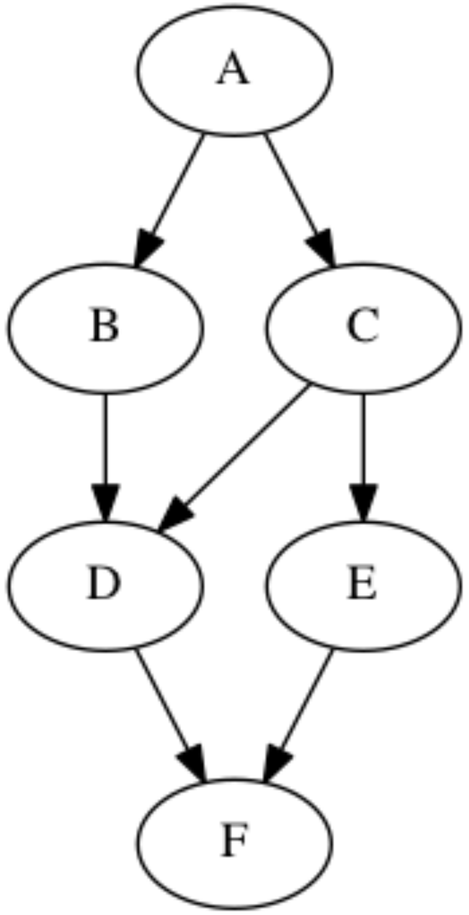
2) Can you write pseudocode with <u>future async-get</u> constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.

Yes, see program sketch with dummy return values.



Worksheet #5 solution (contd)

```
1.  HjFuture<String> A = future(() -> {
2.      return "A"; });
3.  HjFuture<String> B = future(() -> {
4.      A.get(); return "B"; });
5.  HjFuture<String> C = future(() -> {
6.      A.get(); return "C"; });
7.  HjFuture<String> D = future(() -> {
8.      // Order of B.get() & C.get() doesn't matter
9.      B.get(); C.get(); return "D"; });
10. HjFuture<String> E = future(() -> {
11.     C.get(); return "E"; });
12. HjFuture<String> F = future(() -> {
13.     D.get(); E.get(); return "F"; });
14. F.get();
```



Extending Finish Construct with "Finish Accumulators" (Lecture 6)

- Creation

```
accumulator ac = newFinishAccumulator(operator, type);
```

- *Operator must be <u>associative</u> and <u>commutative</u> (creating task "owns" accumulator)*
- Registration

```
finish (ac1, ac2, ...) { ... }
```

- *Accumulators ac1, ac2, ... are registered with the finish scope*
- Accumulation

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

- *Can be performed in parallel by any statement in finish scope that registers ac. Note that a put contributes to the accumulator, but does not overwrite it.*
- Retrieval

```
ac.get();
```

- *Returns initial value if called before end-finish, or final value after end-finish*
- `get()` *is nonblocking because no synchronization is needed (finish provides the necessary synchronization)*

Error Conditions with Finish Accumulators

1. Non-owner task cannot access accumulator outside registered finish

```
// T1 allocates accumulator a
accumulator a = newFinishAccumulator(…);
a.put(1); // T1 can access a
async { // T2 cannot access a
  a.put(1); Number v1 = a.get();
}
```

2. Non-owner task cannot register accumulator with a finish

```
// T1 allocates accumulator a
accumulator a = newFinishAccumulator(...);
async {
  // T2 cannot register a with finish
  finish (a) { async a.put(1);  }
}
```

Worksheet #6: Associativity and Commutativity

Recap:
A binary function f is *associative* if f(f(x,y),z) = f(x,f(y,z)).
A binary function f is *commutative* if f(x,y) = f(y,x).

Worksheet problems:
1) Claim: a Finish Accumulator (FA) can only be used with operators that are *associative and commutative.*
Why? What can go wrong with accumulators if the operator is non-associative or non-commutative?
You may get different answers in different executions if the operator is non-associative or non-commutative
e.g., an accumulator can be implemented using one "partial accumulator" per processor core.

2) For each of the following functions, indicate if it is associative and/or commutative.
a) f(x,y) = x+y, for integers x, y, is associative and commutative
b) g(x,y) = (x+y)/2, for integers x, y, is commutative but not associative
c) h(s1,s2) = concat(s1, s2) for strings s1, s2, e.g., h("ab","cd") = "abcd", is associative but not commutative

Map Reduce: Summary (Lecture 7)

- Input set is of the form {(k1, v1), . . . (kn, vn)}, where (ki, vi) consists of a key, ki, and a value, vi.
  - Assume key and value objects are immutable

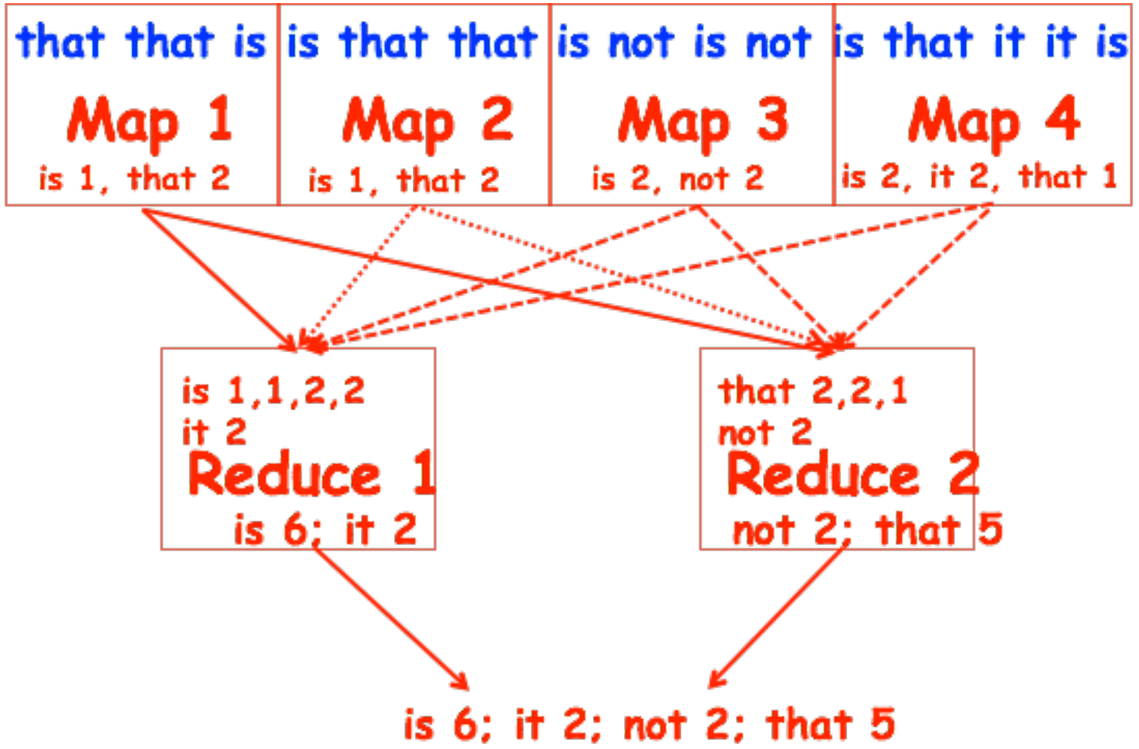
- Map function f generates sets of intermediate key-value pairs, f(ki,vi) = {(k1' ,v1'),...(km',vm')}. The km' keys can be different from ki key in the map function.
  - Assume that a flatten operation is performed as a post-pass after the map operations, so as to avoid dealing with a set of sets.

- 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

Worksheet #7: Analysis of Map Reduce Example

Analyze the total WORK and CPL for the Map reduce example:

- Assume that each Map step has WORK = number of input words, and CPL=1
- Assume that each Reduce step has WORK = number of input word-count pairs, and CPL = log2(# occurrences for input word with largest # pairs)



WORK/CPL for all Map steps:

- WORK = 15
- CPL = 1 (ignore impact of local sums on CPL)

WORK/CPL for Reduce 1 step:

- WORK = 5
- CPL = ceiling(log2(4)) = 2

WORK/CPL for Reduce 2 step:

- WORK = 4
- CPL = ceiling(log2(3)) = 2

Total WORK and CPL

- WORK = 15+5+4 = 24
- CPL = 1 + 2 = 3

Functional vs. Structural Determinism

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

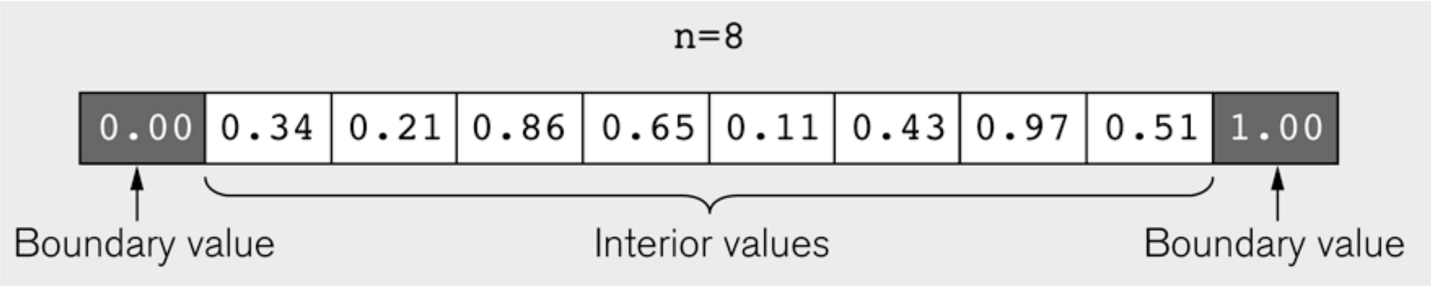
Worksheet #8: Classifying different versions of parallel search algorithms

Enter “YES” or “NO”, as appropriate, in each box below

Example: String Search variation	Data Race Free?	Functionally Deterministic?	Structurally Deterministic?
V1: Count of all occurrences	YES	YES	YES
V2: Existence of an occurrence	NO	YES	YES
V3: Index of any occurrence	NO	NO	YES
V4: Optimized existence of an occurrence: do not create more async tasks after occurrence is found	NO	YES	NO
V5: Optimized index of any occurrence: do not create more async tasks after occurrence is found	NO	NO	NO

One-Dimensional Iterative Averaging Example (Lecture 10)

- 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] = (myVal[i-1]+myVal[i+1])/2, for all i in 1..n



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

HJ code for One-Dimensional Iterative Averaging

```
1.// Intialize m, n, myVal, newVal
2.m = … ; n = … ;
3.float[] myVal = new float[n+2];
4.float[] myNew = new float[n+2];
5.forseq(0, m-1, (iter) -> {
6.  // Compute MyNew as function of input array MyVal
7.    forall(1, n, (j) -> { // Create n tasks
8.        myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.    }); // forall
10.  // What is the purpose of line 11 below?
11.  float[] temp=myVal; myVal=myNew; myNew=temp;
12.}); // forseq
```

Solution to Worksheet #10: One-dimensional Iterative Averaging Example

1) Assuming n=9 and the input array below, perform a "half-iteration" of the iterative averaging example by only filling in the blanks for odd values of j in the myNew[] array (different from the real algorithm). Recall that the computation is "myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;"

index, j	0	1	2	3	4	5	6	7	8	9	10
myVal	0	0	0.2	0	0.4	0	0.6	0	0.8	0	1
myNew	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1

2) Will the contents of myVal[] and myNew[] change in further iterations?

No, this represents the converged value (equilibrium/fixpoint).

3) Write the formula for the final value of myNew[i] as a function of i and n. In general, this is the value that we will get if m (= #iterations in sequential for-iter loop) is large enough.

After a sufficiently large number of iterations, the iterated averaging code will converge with myNew[i] = myVal[i] = i / (n+1)

Barriers (Lecture 11)

- Question: how can we transform this code so as to ensure that all tasks say hello before any tasks say goodbye, without having to change the local variable?
- Approach 2: insert a "barrier" ("next" statement) between the hello's and goodbye's

```
// 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

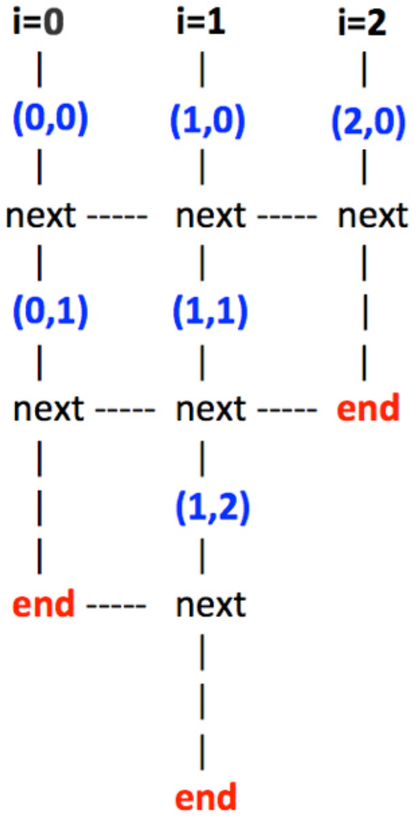
- next -> each forallPhased iteration waits at barrier until all iterations arrive (previous phase is completed), after which the next phase can start
  - Scope of next is the closest enclosing forallPhased statement
  - If a forallPhased iteration terminates before executing "next", then the other iterations don't wait for it

Worksheet #11: Forall Loops and Barriers

Draw a “barrier matching” figure similar to lecture 12 slide 11 for the code fragment below.

```
1. String[] a = { “ab”, “cde”, “f” };
2. . . . int m = a.length; . . .
3. forallPhased (0, m-1, (i) -> {
4.   for (int j = 0; j < a[i].length(); j++) {
5.     // forall iteration i is executing phase j
6.     System.out.println("(" + i + "," + j + ")");
7.     next();
8.   }
9. });
```

Solution



Extending HJ Futures for Macro-Dataflow: Data-Driven Futures (Lecture 12)

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

- Allocate an instance of a data-driven-future object (container)
- Object in container must be of type T1, and can only be assigned once via put() operations
- HjDataDrivenFuture extends the HjFuture interface

```
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

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

```
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")
- Await clause can be used to implement "nodes" and "edges" in a computation graph

```
ddfA.get()
```

- Return value (of type T1) stored in ddfA
- Throws an exception if put() has not been performed
  - Should be performed by async's that contain ddfA in their await clause, or if there's some other synchronization to guarantee that the put() was performed

Worksheet #12: Data Driven Futures

For the example below, will reordering the five async statements change the meaning of the program (assuming that the semantics of the reader/writer methods depends only on their parameters)? If so, show two orderings that exhibit different behaviors. If not, explain why not.

No, reordering the asyncs doesn't change the meaning of the program. Regardless of the order, Task 3 will always wait on Task 1. Task 5 will always wait on Task 2. Task 4 will always wait on both Task 1 and 2.

```
1. DataDrivenFuture left = new DataDrivenFuture();
2. DataDrivenFuture right = new DataDrivenFuture();
3. finish {
4.   async await(left) leftReader(left); // Task3
5.   async await(right) rightReader(right); // Task5
6.   async await(left,right)
7.         bothReader(left,right); // Task4
8.   async left.put(leftWriter()); // Task1
9.   async right.put(rightWriter());// Task2
10. }
```

Converting `forseq-forall` version into a `forall-forseq` version with barriers (Lecture 14)

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

Worksheet #14 Solution: Iterative Averaging Revisited

Answer the questions in the table below for the versions of the Iterative Averaging code shown in slides 7, 8, 10, 12. Write in your answers as functions of m, n, and nc.

	Slide 7	Slide 8	Slide 10	Slide 12
How many tasks are created (excluding the main program task)?	m*n	n Incorrect: n * m	m*nc Incorrect: n * nc	nc Incorrect: n*m, m*nc
How many barrier operations (calls to next per task) are performed?	0 Incorrect: m	m Incorrect: m*n	0 Incorrect: m	m Incorrect: m*nc, nc

Which SPMD version is the most efficient?

Summary of Phaser Construct (Lecture 15)

- 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
    - Semantics depends on registration mode
    - Barrier is a special case of phaser, which is why next is used for both

Worksheet #15: Reordered Asyncs with One Phaser

Task A4 has been moved up to line 6. Does this change the computation graph in slide 9? If so, draw the new computation graph. If not, explain why the computation graph is the same.

No, A4 still needs to wait on A2 and A3 to signal before it can start doA4Phase2().

```
1. finish (() -> {
2.     ph = newPhaser(SIG_WAIT); // mode is SIG_WAIT
3.     asyncPhased(ph.inMode(SIG), () -> {
4.       // A1 (SIG mode)
5.       doA1Phase1(); next(); doA1Phase2(); });
6.     asyncPhased(ph.inMode(HjPhaserMode.WAIT), () -> {
7.      // A4 (WAIT mode)
8.      doA4Phase1(); next(); doA4Phase2(); });
9.     asyncPhased(ph.inMode(SIG_WAIT), () -> {
10.      // A2 (SIG_WAIT mode)
11.      doA2Phase1(); next(); doA2Phase2(); });
12.    asyncPhased(ph.inMode(HjPhaserMode.SIG_WAIT), () -> {
13.     // A3 (SIG_WAIT mode)
14.     doA3Phase1(); next(); doA3Phase2(); });
15.    });
```

Summary of Parallel Programming Constructs you've learned so far

- **Task Parallelism (Unit 1)**
  - —Async (task creation)
  - —Finish (structured task termination)
- **Functional Parallelism (Unit 2)**
  - —Future (task creation)
  - —Future get() (task termination with return value)
  - —Accumulators (functional reduction)
  - —Map-Reduce (functional parallelism & reduction on key-value pairs)
- **Loop Parallelism (Unit 3)**
  - —Forall (parallel loops)
  - —Barriers (all-to-all synchronization)
- **Dataflow Parallelism (Unit 4)**
  - —Data-Driven Tasks (dataflow parallelism)
  - —Phasers (point-to-point synchronization)
  - —Phaser-specific next operations

Midterm Exam

- Midterm exam (Exam 1) will be held at 7pm on Thursday, March 11, 2021 in Canvas
  - Open-notes, open-book exam scheduled for 2 hours during 7pm — 9pm
  - You are allowed to use your laptop ONLY to enter your answers in Canvas, nothing else
  - Scope of exam is limited to Lectures 1 - 15 (all topics except section 4.1 and 4.4 in Module 1 handout)
  - 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.