COMP 322: Fundamentals of Parallel Programming

Lecture 17: Midterm Review (Part 2)

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

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

Scope of Midterm Exam

- Midterm exam will cover material from Lectures 1 - 15
  - —Lecture 16 on Phaser Accumulators and Bounded Phasers is excluded
- Excerpts from midterm exam instructions
  - —"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."

Week 3 Lecture Quiz Solution: Question 1

Consider the following HJ class, FutureInteger, that contains a single field of type future<Integer>.

```
1.  class FutureInteger {
2.    private final future<Integer> f; // class FutureInteger is a box for a single field f of type f
uture<Integer>
3.    FutureInteger(future<Integer> input) {f = input;} // constructor
4.    public int value() {return f.get().intValue();}
5.    public static FutureInteger add(FutureInteger fi1, FutureInteger fi2) {
6.      return new FutureInteger(
7.                               // Create a future task and return a reference to its container
8.                               async<Integer>{perf.doWork(1); return fi1.value() + fi2.value();}
9.      );
10.  }
11.  public String toString() {return f.get().toString();}
12.  public static void main(String[] args)  {
13.    Random r = new Random();
14.    FutureInteger fi1 = new FutureInteger(async<Integer>{perf.doWork(1); return r.nextInt();});
15.    FutureInteger fi2 = new FutureInteger(async<Integer>{perf.doWork(1); return r.nextInt();});
16.    FutureInteger fi3 = add(fi1, fi2);
17.    FutureInteger fi4 = new FutureInteger(async<Integer>{perf.doWork(1); return r.nextInt();});
18.    FutureInteger fi5 = add(fi3,fi4);
19.    System.out.println("fi5 = " + fi5);
20.  }
21. }
```

The calls to perf.doWork(1) in lines 8, 14, 15, 17 indicate that each addition and each call to nextInt() is counted as 1 unit of work in the abstract performance metrics, and everything else is assumed to take zero time.

Which of the following statements are true?

Week 3 Lecture Quiz Solution: Question 1 (contd)

Your Answer		Score	Explanation
☑ In line 4, method value() can only return after the future task referenced by field f has terminated.	✔	1.00	The blocking semantics of f.get() ensures that method value() can only return after the future task referenced by field f has terminated.
☐ In lines 5-10, method add() can only return after both future tasks referenced by parameters fi1.f and fi2.f have terminated.	✔	1.00	Method add() creates a future task to perform the addition but does not wait for the addition to complete. Therefore, method add() can return before future tasks referenced by parameters fi1.f and fi2.f have terminated.
☐ In line 14, the call to "new FutureInteger()" must wait for the call to nextInt() to complete i.e., line 15 cannot start before time = 1 (in abstract metrics).	✔	1.00	The async future expression returns a reference to the future object right away, so the call to "new FutureInteger()" need not wait for the call to nextInt() to complete.
☑ The call to "new FutureInteger()" in line 17 (in abstract metrics) can start at time = 0.	✔	1.00	Statement 13 is assumed to take zero time, and statements 14-16 are async tasks, so statement 17 can start at time = 0.

Week 3 Lecture Quiz Solution: Question 2

The main program in Question 1 above results in WORK = 5 units since it includes three calls to nextInt() and two add operations.

What is its CPL (critical path length) value?

NOTE: Do not try this code in DrHJ to determine the answer. The current release will not give the correct answer for this program.

You entered:

3

Your Answer		Score	Explanation
3	✔	1.00	
Total		1.00 / 1.00	

Question Explanation

The critical path length is 3 because it includes one time unit for the first two calls to nextInt() to occur in parallel, one time unit for adding the values of fi1+f2, and the third time unit for adding the values of f3+f4.

Week 3 Lecture Quiz Solution: Questions 3 & 4

Consider the following HJ code for counting all occurrences of pattern in text, using finish accumulators:

```
1.  accumulator a = accumulator.factory.accumulator(SUM, int.class)
;
2.  finish (a) {
3.    for (int i = 0; i <= N - M; i++) async {
4.      int j;
5.      for (j = 0; j < M; j++)
6.        if (text[i+j] != pattern[j]) break;
7.      if (j == M) a.put(1); // found at offset i
8.    } // for-async
9.    int count1 = a.get().intValue();
10. } // finish
11. int count2 = a.get().intValue();
```

Answer = 0

Answer = number of occurrences of pattern

Prefix Sum Problem Statement (Lecture 8)

Given input array A, compute output array X as follows

$$X[i] = \sum_{0 \le j \le i} A[j]$$

- **The above is an inclusive prefix sum since X[i] includes A[i]**
- **For an exclusive prefix sum, perform the summation for 0 <=j <i**
- **Addition can be replaced by any *associative* operator, f**
- **It is easy to see that prefix sums can be computed sequentially in O(n) time**

```
// Copy input array A into output array X
X = new int[A.length]; System.arraycopy(A,0,X,0,A.length);
// Update array X with prefix sums (f can be +, for example)
for (int i=1 ; i < X.length ; i++ ) X[i] = f(X[i-1],X[i]);
```


Summary of Parallel Prefix Sum Algorithm

- Critical path length, CPL = O(log n)
- Total number of add operations, WORK = O(n)
- Optimal algorithm for P = O(n/log n) processors
  - —Adding more processors does not help
- Parallel Prefix Sum has several applications that go beyond computing the sum of array elements
- Parallel Prefix Sum can be used for any operation that is associative (need not be commutative)
  - —In contrast, finish accumulators require the operator to be both associative and commutative

How do associativity and commutativity make a difference?

Time for worksheet #8!

Worksheet #8 solution: Associativity and Commutativity

A Finish Accumulator (FA) can be used for any *associative and commutative* binary function.
Parallel Prefix (PP) algorithm can be used for any *associative* binary function (the same applies for parallel reductions in ArraySum1 and ArraySum2).

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).

For each of the following functions, indicate if it can be used in a finish accumulator or a parallel prefix sum algorithm or both or neither.

1) f(x,y) = x+y, for integers x, y, is associative and commutative
   ⇒ both FA and PP can be used

2) g(x,y) = (x+y)/2, for integers x, y, is commutative but not associative
   ⇒ neither FA nor PP can be used

3) h(s1,s2) = concat(s1, s2) for strings s1, s2 e.g., h("ab","cd") = "abcd" is associative but not commutative
   ⇒ PP can be used, but not FA

Week 4 Lecture Quiz Solution: Question 1

Consider the problem of partitioning an input array with respect to a pivot value, so as to obtain an output array with into two contiguous parts --- the left part with all elements < pivot and the right part with all elements ≥ pivot. This problem can be solved in parallel as follows:

1. Compute a 0/1 array, lt[*], such that lt[k] = 1 if in[k] < pivot, and 0 otherwise
2. In parallel with 1, compute a 0/1 array, ge[*], such that ge[k] = 1 if in[k] ≥ pivot, and 0 otherwise

} CPL=1

3. Compute ltPS, the (inclusive) prefix sum for lt, such that ltPS[i] = $\sum_{0 \le j \le i} lt[j]$, and ltCount = ltPS[size-1], the number of 1's in lt[*]
4. In parallel with 3, compute gePS, the (inclusive) prefix sum for ge, such that gePS[i] = $\sum_{0 \le j \le i} ge[j]$, and geCount = gePS[size-1], the number of 1's in ge[*]

} CPL = 10+10 = 20

5. For each k value such that lt[k] = 1, assign out[ltPS[k]-1] = in[k]
6. In parallel with 5, for each k value such that ge[k] = 1, assign out[ltCount + gePS[k]-1] = in[k]

Assume that the abstract metrics charge 1 unit of time for each < or ≥ comparison performed in steps 1 and 2, and 1 unit of time for each add performed in the prefix sums in steps 3 and 4. Everything else is assumed to have zero cost. What is the CPL value for this algorithm, assuming input and output array sizes of 1024 elements?

Total CPL = 21, but quiz answer said 11

Quiz grading will be updated to give zero weightage to this question!

Week 4 Lecture Quiz Solution: Question 2

Consider step 5 from Question 1:

5. For each k value such that lt[k] = 1, assign out[ltPS[k]-1] = in[k]

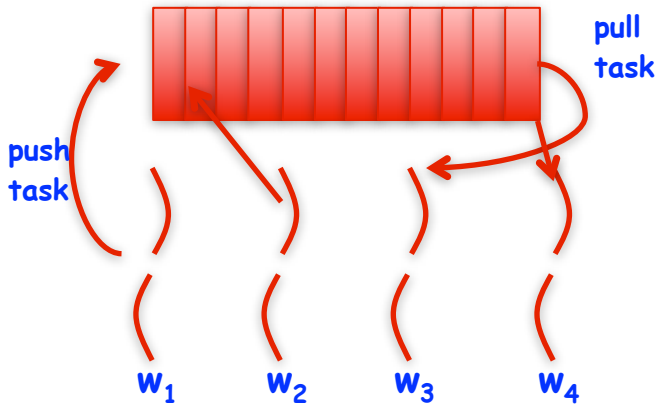
Which of the following statements is true?

Your Answer		Score	Explanation
☑ It is possible for ltPS[k] = n for some k, where n is the size of the in and out arrays	✔	1.00	Yes, this will happen if lt[k] = 1, for all k.
☑ It is possible for ltPS[k] < n for some k, where n is the size of the in and out arrays	✔	1.00	Yes, this will happen if lt[*] contains at least one zero, or if n > 1 since ltPS[0] can be at most 1.
☐ It is possible for ltPS[k] > n for some k, where n is the size of in and out array	✔	1.00	No, this is not possible.
☑ Step 5 results in packing all elements with lt[k] = 1 contiguously in the left part of the out[] array.	✔	1.00	Yes.

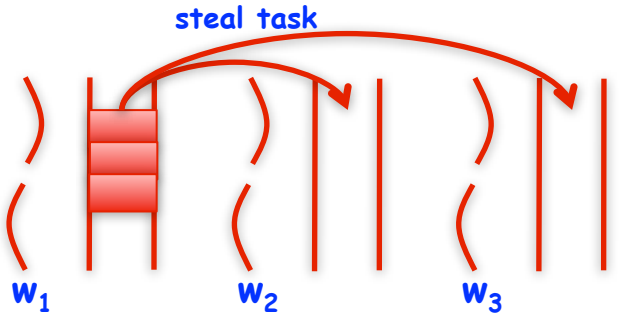
Confusion about first part: if pivot came from in[] array then the answer should be false. This part will also receive zero weightage.

Abstract vs. Real Performance --- Work-Sharing vs. Work-Stealing Scheduling (Lecture 9)

- **Work-Sharing**
  - —Busy worker eagerly distributes new work
  - —Easy implementation with global task pool
  - —Access to the global pool needs to be synchronized: scalability bottleneck
- **Work-Stealing**
  - —Busy worker incurs little overhead to create work
  - —Idle worker "steals" the tasks from busy workers
  - —Distributed task pools lead to improved scalability
  - —When task T_a spawns T_b, the worker can
    - –stay on T_a, making T_b available for execution by another processor (help-first policy), or
    - –start working on T_b first (work-first policy)



work-sharing



work-stealing runtime

Work-first vs. Help-first work-stealing policies on 2 processors (contd)

```
1. finish {
2.   // Start of Task T0 (main program)
3.   sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
4.   async { // Task T1 computes sum of upper half of array
5.     for(int i=X.length/2; i < X.length; i++)
6.       sum2 += X[i];
7.   }
8.   // T0 computes sum of lower half of array
9.   for(int i=0; i < X.length/2; i++) sum1 += X[i];
10. }
11. // Task T0 waits for Task T1 (join)
12. return sum1 + sum2;
13.} // finish
```

Help-First worker does not switch tasks
Work-first worker will switch tasks

Continuations

Help-First worker can switch tasks
Work-first worker can switch tasks

Let's try more of this in Worksheet #9 !

Worksheet #9 solution: Continuations and Work-First vs. Help-First Work-Stealing Policies

For each of the continuations below, label it as "WF" if a work-first worker can switch tasks at that point and as "HF" if a help-first worker can switch tasks at that point. Some continuations may have both labels.

```
1.finish { // F1
2.   async A1;
3.   finish { // F2
4.     async A3;
5.     async A4;
6.   }
7.   S5;
8.}
```

WF

WF

WF

WF, HF

WF, HF

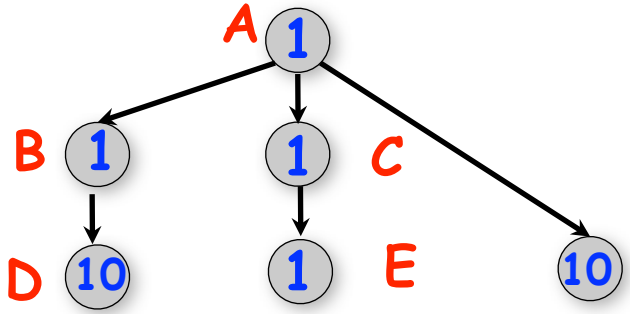
Continuations

Scheduling Policies Currently Available in HJ (Lecture 10)

DrHJ compiler option	Command-line option	Functional characteristics	Performance characteristics
work-sharing (Default option)	Compile: hjc -rt s (default) Runtime: hj (no option needed)	1) Supports full lang 2) Supports perf metrics	3) Creates additional worker threads when a task blocks
work-sharing (Fork-Join variant)	Compile: hjc -rt s (default) Runtime: hj -fj	1) + 2)	3) + 4) may perform better than work-sharing for recursive parallelism
work-stealing (Help-First policy)	Compile: hjc -rt h Runtime: hj (no option needed)	5) Only supports async, finish, forasync, isolated, atomic vars	6) Fixed number of worker threads 7) better for loop parallelism
work-stealing (Work-First policy)	Compile: hjc -rt w Runtime: hj (no option needed)	5) + 8) Supports data race detection	6) + 9) better for recursive parallelism
work-stealing (Adaptive policy)	Compile: hjc -rt h Runtime: hj (no option needed)	Same as 5)	10) automatically chooses between help-first and work-first policies on each async
cooperative *(under development)*	*Compile: hjc -rt c* *Runtime: hj* *(no option needed)*	*Currently supports 5) + Futures --- goal is to support everything!*	*Same as 6)*

Worksheet #10 solution: Scheduling Program Q2 using Work-First & Help-First Schedulers

Complete work-first and help-first schedules for the program shown below (using step times from the computation graph)

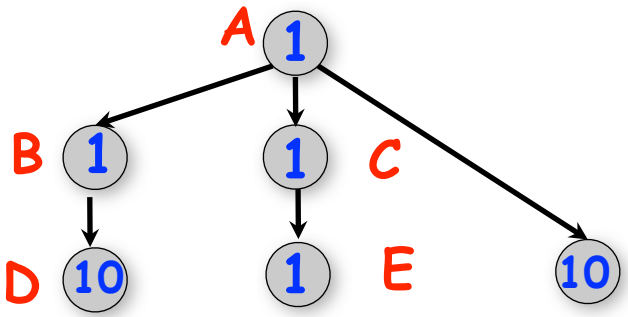



```
1. // Program Q2
2. A;
3. finish {
4.   async { C; E; }
5.   async F;
6.   async { B; D; }
7. }
```

Work-First Schedule

Start time	Proc 1	Proc 2
0	A	
1	C	F
2	E	F
3	B	F
4	D	F
5	D	F
6	D	F
7	D	F
8	D	F
9	D	F
10	D	F
11	D	
12	D	
13	D	

Worksheet #10 solution: Scheduling Program Q2 using Work-First & Help-First Schedulers (contd)




```
1. // Program Q2
2. A;
3. finish {
4.   async { C; E; }
5.   async F;
6.   async { B; D; }
7. }
```

Help-First Schedule

Start time	Proc 1	Proc 2
0	A	
1	B	C
2	D	E
3	D	F
4	D	F
5	D	F
6	D	F
7	D	F
8	D	F
9	D	F
10	D	F
11	D	F
12		F
13		

seq clause in HJ async statement

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

```
1. // Non-Future example
2. async seq(size < thresholdSize) computeSum(X, lo, mid);
3.
4. // Future example
5. final future<int> sum1 = async<int> seq(size < thresholdSize)
6.                                 { 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**
- **Also simplifies use of final variables (needed for futures)**

Parallel Solution to NQueens with Finish Accumulator and seq clause

```
1.  static accumulator count;
2.  . . .
3.  count = accumulator.factory.accumulator(SUM, int.class);
4.  finish(count) nqueens_kernel(new int[0], 0);
5.  System.out.println("No. of solutions = " + count.get().intValue());
6.  . . .
7.  void nqueens_kernel(int [] a, int depth) {
8.    if (size == depth) count.put(1);
9.    else
10.     /* try each possible position for queen at depth */
11.     for (int i =  0; i < size; i++) async seq(depth >= cutoff) {
12.       /* allocate a temporary array and copy array a into it */
13.       int [] b = new int [depth+1];
14.       System.arraycopy(a, 0, b, 0, depth);
15.       b[depth] = i;
16.       if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
17.     } // for-async
18. } // nqueens_kernel()
```


Week 4 Lecture Quiz Solution: Questions 3 and 4

Consider the following modified version of ArraySum2 with futures that also uses a seq clause:

```
  static final int THRESHOLD = 64;

  static int computeSum(int[] X, int lo, int hi) {
    if ( lo > hi ) {
      return 0;
    } else if ( lo == hi ) {
      return X[lo];
    } else {
      int mid = (lo+hi)/2;
      final future<int> sum1 = async<int> seq(hi-lo < THRESHOLD) {
 return computeSum(X, lo, mid); };
      final future<int> sum2 = async<int> seq(hi-lo < THRESHOLD){
return computeSum(X, mid+1, hi); };
      int local_sum = sum1.get() + sum2.get();
      perf.doWork(1); // Assume that add takes 1 unit of time, and
 that everything else is free
      return local_sum;
    }
  } // computeSum
```

Regardless of the THRESHOLD value, what is the total WORK done by the above program for an input array size of 128?

For input size = 128,

WORK = 127

and

CPL = 63+1 = 64 (because of seq clause)

Summary of HJ's forasync statement (Lecture 11)

```
forasync (point [i1] : [lo1:hi1]) <body>
forasync (point [i1,i2] : [lo1:hi1,lo2:hi2]) <body>
forasync (point [i1,i2,i3] : [lo1:hi1,lo2:hi2,lo3:hi3]) <body>
. . .
```

- forasync statement creates multiple async child tasks, one per iteration of the forasync
  - all child tasks can execute <body> in parallel
  - child tasks are distinguished by index "points" ([i1], [i1,i2], …)
- <body> can read local variables from parent (copy-in semantics like async)
- forasync needs a finish for termination, just like regular async tasks
  - Later, we will learn about replacing "finish forasync" by "forall"
- In addition to its convenient syntax, parallel loop constructs are easier to manage with "chunking", compared to for-for-async structures

Chunking a 1-dimensional forasync loop (General approach)

- Assume that the forasync loop originally iterates over region r

```
forasync(point[i] : r)
  BODY(i); // No. of tasks = r.size()
```

- Assume that we have a parameter, nc, for the desired number of chunks (tasks)
  - —A good choice is nc = Runtime.getNumOfWorkers(), as in Listing 31
- Assume that we have a helper method, getChunk(r, nc, ii) that returns the iteration range for chunk # ii as an HJ region
  - —e.g., getChunk([0:99], 4, 0) = [0:24] and getChunk([0:99], 4, 3) = [75:99]
  - —No requirement for nc to evenly divide r.size()
- The original forasync above can then be rewritten as

```
forasync(point[ii] : [0:nc-1])
  for(point[i] : getChunk(r,nc,ii))
    BODY(i); // No. of tasks = nc
```


Solution to Worksheet #11: One-dimensional Iterative Averaging Example

1) Assuming n=9 and the input array below, perform one iteration of the iterative averaging example by only filling in the blanks for odd values of j in the myNew[] array. 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, after myNew above in 1) becomes myVal[] in the next iteration?

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

Week 5 & 6 Lecture Quiz Solution: Question 1

Consider the original parallel version of the One-Dimensional Iterative Averaging algorithm from slide 18 of Lecture 11, extended with a call to doWork(1) in line 4 so that WORK = m*n:

```
0. myVal = new double[n+2]; myNew = double[n+2];
1. for (point [iter] : [0:m-1]) {
2.   // Compute MyNew as function of input array MyVal
3.   finish forasync (point [j] : [1:n]) {
4.     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0; perf.doWork(1);
5.   } // finish forasync
6.   temp=myVal; myVal=myNew; myNew=temp; // Swap myVal & myNew
7.   // myNew becomes input array for next iteration
8. } // for
```

Which of the following statements is true? (Recall that forasync creates one task for each iteration in its range.)

Your Answer		Score	Explanation
☐ The critical path length must be CPL = n	✔	1.00	No, CPL = m, the number of iterations of the for loop in line 1.
☑ The total number of async tasks created is m*n	✔	1.00	Yes, it's the product of the number of iterations of the for loop (line 1) and the forasync loop (line 3).
☑ If we replaced line 6 by the following, the semantics of the program will be unchanged i.e., it will produce the same output as before: `6. myVal=myNew; myNew=new double[n+2];`	✔	1.00	Yes. Instead of using only two arrays, we are now allocating a new array in each iteration of the for-iter loop (line 1). The semantics is unchanged, but this modified version will likely trigger garbage collection more frequently than the original version and run slower as a result.

Example: HJ code for One-Dimensional Iterative Averaging with chunked for-finish-forasync-for structure

```
1. int nc = Runtime.getNumOfWorkers();
2. for (point [iter] : [0:m-1]) {
3.   // Compute MyNew as function of input array MyVal
4.   finish forasync (point [jj] : [0:nc-1]) {
5.     for(point [j] : getChunk([1:n],nc,jj))
6.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
7.   } // finish forasync
8.   temp=myVal; myVal=myNew; myNew=temp;// Swap myVal & myNew;
9.   // myNew becomes input array for next iteration
10.} // for
```


Week 5 & 6 Lecture Quiz Solution: Question 2

Now consider the chunked version of the One-Dimensional Iterative Averaging algorithm introduced in slide 23 of Lecture 11:

```
0. myVal = new double[m+1]; myNew = double[m+1];
1. int nc = Runtime.getNumOfWorkers();
2. for (point [iter] : [0:m-1]) {
3.   // Compute MyNew as function of input array MyVal
4.   finish forasync (point [jj] : [0:nc-1]) {
5.     for(point [j] : getChunk([1:n],nc,jj)) {
6.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0; perf.doWork(1); }
7.   } // finish forasync
8.   temp=myVal; myVal=myNew; myNew=temp; // Swap myVal & myNew
9.   // myNew becomes input array for next iteration
10. } // for
```

Assume that m = 100, n = 1000, and nc =10, so that WORK = 100000. What is the CPL value for this version? (Enter your answer as an integer with no commas or decimal points.)

Question Explanation

The CPL is m*(n/nc) since each chunk in line 5 consists of n/nc iterations and there are m iterations in the outer for-iter loop (line 2). Thus, CPL = m*(n/nc) = 100*(1000/10) = 100*100 = 10000.

HJ's forall statement = finish + forasync + barriers (Lecture 12)

Goal 1 (minor): replace common finish-forasync idiom by forall
e.g., replace

```
finish forasync (point [I,J] : [0:N-1,0:N-1])
  for (point[K] : [0:N-1])
    C[I][J] += A[I][K] * B[K][J];
```

by

```
forall (point [I,J] : [0:N-1,0:N-1])
  for (point[K] : [0:N-1])
    C[I][J] += A[I][K] * B[K][J];
```

Goal 2 (major): Also support "barrier" synchronization

- Caveat: forall is only supported on the work-sharing runtime because of barrier synchronization

Hello-Goodbye Forall Example (contd)

- 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

```
1. // APPROACH 2
2. forall (point[i] : [0:m-1]) {
3.   int sq = i*i;
4.   System.out.println("Hello from task with square = " + sq);
5.   next; // Barrier
6.   System.out.println("Goodbye from task with square = " + sq);
7. }
```

Phase 0

Phase 1

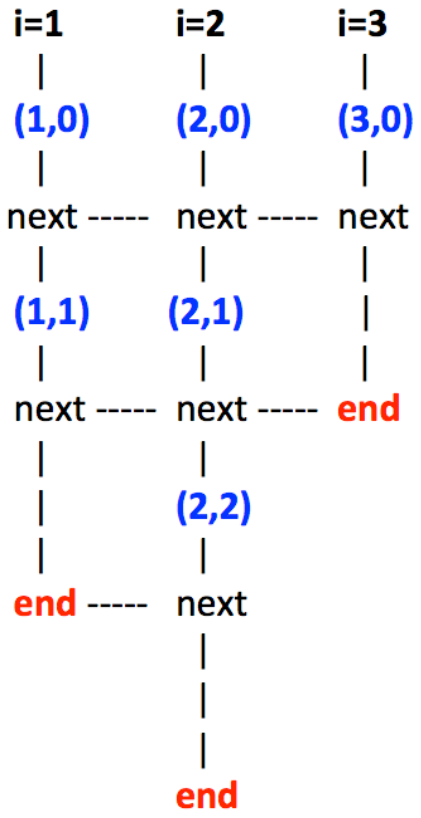
- next ➔ each forall iteration suspends at next until all iterations arrive (complete previous phase), after which the phase can be advanced
  - —If a forall iteration terminates before executing "next", then the other iterations do not wait for it
  - —Scope of next is the closest enclosing forall statement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

Worksheet #12: Forall Loops and Barriers

1) Draw a "barrier matching" figure similar to slide 14 for the code fragment below.

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

Solution



Single Program Multiple Data (SPMD) Parallel Programming Model (Lecture 13)

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

One-Dimensional Iterative Averaging: Barrier version with *chunked forall-for-for+next* structure is an SPMD program

```
1.  double[] gVal=new double[n+2]; double[] gNew=new double[n+2]; gVal[n+1] = 1;
2.  int nc = Runtime.getNumWorkers();
3.  forall (point [jj]:[0:nc-1]) { // Chunked forall is now the outermost loop
4.    double[] myVal = gVal; double[] myNew = gNew; // Copy of myVal/myNew pointers
5.    for (point [iter] : [0:m-1]) {
6.      // Compute MyNew as function of input array MyVal
7.      for (point [j]:getChunk([1:n],nc,jj)) // Iterate within chunk
8.         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.      next; // Barrier before executing next iteration of iter loop
10.     // Swap local pointers, myVal and myNew
11.     double[] temp=myVal; myVal=myNew; myNew=temp;
12.     // myNew becomes input array for next iter
13.   } // for
14. } // forall
```

Instead of async-finish, this SPMD version of OneDimAveraging creates one task per worker, uses getChunk() to distribute work, and use barriers to synchronize workers.

Week 5 & 6 Lecture Quiz Solution: Question 3 re. previous barrier version

Which of the following statements is true?

Your Answer		Score	Explanation
☐ The number of async tasks created for the chunked fork-join version earlier in Question 2 is nc.	✔	1.00	No, the number of async tasks created is m*nc.
☑ The number of async tasks created for the chunked barrier version above in Question 3 is nc.	✔	1.00	Yes, the forall loop loop in line 3 has exactly nc iterations.
☐ If nc evenly divides n, the CPL for the chunked barrier version is exactly n/nc	✔	1.00	No, the CPL is m*(n/nc) since the for-iter loop in line 5 has m iterations.

Use of next-with-single to print a log message between Hello and Goodbye phases

```
1.// Listing 37 in Module 1 handout
2. forall (point[i] : [0:m-1]) {
3.   int sq = i*i;
4.   System.out.println("Hello from task with square = " + sq);
5.   next { // next-with-single statement
6.     System.out.println("LOG: Between Hello & Goodbye phases");
7.   }
8.   System.out.println("Goodbye from task with square = " + sq);
9. }
```


Week 5 & 6 Lecture Quiz Solution: Question 4

Consider the following forall loop with a next-with-single statement as in slide 11 of Lecture 13:

```
1. forall (point[i] : [0:9]) {
2.    ...; perf.doWork(1);
3.    next { ...; perf.doWork(1); } // next-with-single statement
4.    ...; perf.doWork(1);
5. }
```

In this question, we are not concerned with the actual computation done in ..., but just the calls to perf.doWork().

Which of the following statements is true?

Your Answer		Score	Explanation
☐ The total value of WORK is 30	✔	0.50	No, since the next-with-single statement is done exactly once, it should contribute only once to WORK.
☑ The total value of WORK is 21	✔	0.50	Yes, 10 for phase 1, 1 for the next-with-single statement, and 10 for phase 2.
☐ The CPL value is 21	✔	0.50	No, the CPL is 3.
☑ The CPL value is 3	✔	0.50	Yes, 1 for phase 1, 1 for the next-with-single statement, and 1 for phase 2.

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

```
ddfA = new DataDrivenFuture<T1>();
```

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

```
async await(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()

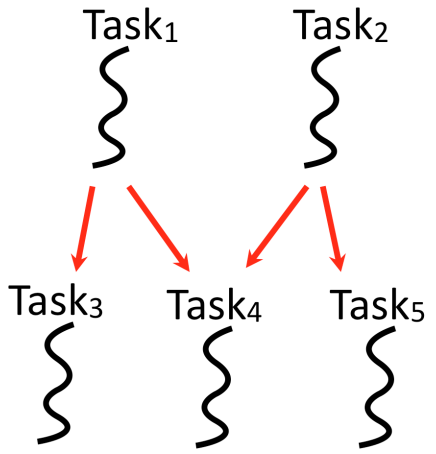
- Return value (of type T1) stored in ddfA
- Can only be performed by async's that contain ddfA in their await clause (hence no blocking is necessary for DDF gets)

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 "async await" statement does not block the worker, unlike a future.get()
  - —You will never see the following message with "async await"
    - "ERROR: Maximum number of hj threads per place reached"

Another Example with DDTs and DDFs

```
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. }
```



- await clauses capture data flow relationships

Interesting example. Let's discuss it further in Worksheet 13!

Worksheet #13: Data-Driven Tasks

For the example below, will reordering the five async statements change the meaning of the program? If so, show two orderings that exhibit different behaviors. If not, explain why not. (You can use the space below this slide for your answer.)

```
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. }
```

No, reordering consecutive async's will never change the meaning of the program, whether or not the async's have await clauses.

Week 5 & 6 Lecture Quiz Solution, Question 5: which of the following are true?

Your Answer		Score	Explanation
☐ It is possible to perform multiple calls to ddfA.put(), if ddfA contains a reference to a DataDrivenFuture object	✔	1.00	No, data-driven futures must obey the single assignment rule, so they only allow at most one put() operation.
☑ It is possible for a data-driven (async await) task to perform put() operations on multiple DataDrivenFuture objects e.g., ddfA.put() and ddfB.put()	✔	1.00	Yes, there is no restriction on how many distinct DDFs a DDT can perform put() operations on.
☐ It is impossible to create a deadlock using async await i.e., to create a situation where an async task can be blocked indefinitely	✔	1.00	No. If an async await task (T_A) specifies a DDF that no other task produces, then task T_A will be blocked indefinitely.

Recap of HJ constructs studied in Lectures 1-13

- Basic language summary can be found here:
  - —https://wiki.rice.edu/confluence/display/PARPROG/HJLanguageSummary
  - —Additional documentation in preparation
- Task creation constructs
  - — async *Stmt*
  - — async<T> { *Stmt ;* return ...; }
  - — forasync (point[i,j] : ...) *Stmt*
  - — forall (point[i,j] : ...) *Stmt*
- Loop constructs
  - — point
  - — region
  - — for (point[i,j] : ...) *Stmt*

Recap of HJ constructs studied in Lectures 1-13 (contd)

- Synchronization constructs
  - finish
  - f.get()
  - finish accumulators
  - next
  - async await
- Efficiency constructs
  - Converting async to async seq
  - Loop chunking with GetChunk()
  - Converting futures to data-driven futures
- Abstract metrics
  - perf.doWork(n)

Summary of Phaser Construct (Lecture 14)

- Phaser allocation
  - phaser ph = new phaser(mode);
    - Phaser ph is allocated with registration mode
    - Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)
- Registration Modes
  - phaserMode.SIG, phaserMode.WAIT, phaserMode.SIG_WAIT, phaserMode.SIG_WAIT_SINGLE
  - NOTE: phaser WAIT is unrelated to Java wait/notify (which we will study later)
- Phaser registration
  - async phased (ph_1<$mode_1$>, ph_2<$mode_2$>, …) <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
    - async phased <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

Simple Example with Four Async Tasks and One Phaser (Listing 43)

```
1. finish {
2.   ph = new phaser(); // Default mode is SIG_WAIT
3.   async phased(ph<phaserMode.SIG>){ //A1 (SIG mode)
4.     doA1Phase1(); next;
5.     doA1Phase2(); }
6.   async phased { //A2 (default SIG_WAIT mode from parent)
7.     doA2Phase1(); next;
8.     doA2Phase2(); }
9.   async phased { //A3 (default SIG_WAIT mode from parent)
10.    doA3Phase1(); next;
11.    doA3Phase2(); }
12.  async phased(ph<phaserMode.WAIT>){ //A4 (WAIT mode)
13.    doA4Phase1(); next; doA4Phase2(); }
14. }
```

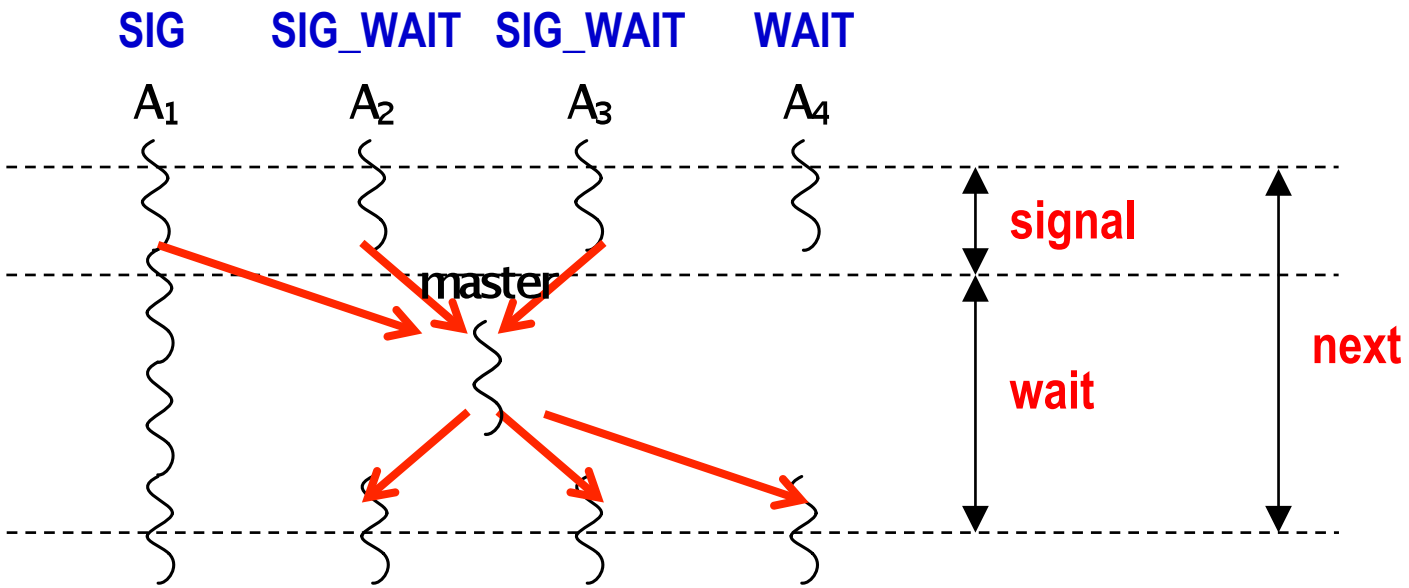

Simple Example with Four Async Tasks and One Phaser (Figure 48)

Semantics of next depends on registration mode

SIG_WAIT: **next = signal + wait**

SIG: **next = signal**

WAIT: **next = wait**



A master thread (worker) gathers all signals and broadcasts a barrier completion

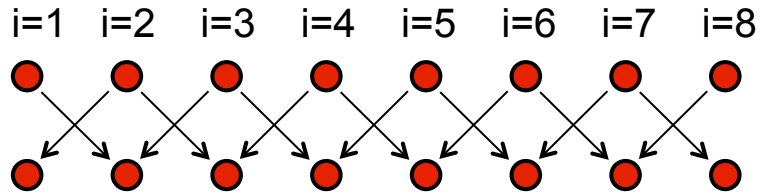
Week 5 & 6 Lecture Quiz Solution, Question 6: which of the following are true?

Your Answer		Score	Explanation
☑ Task A1 (created in line 3) can complete before the other tasks have started.	✔	1.00	Yes. Since task A1 is registered on phaser ph in signal mode, it need not wait for any other task.
☐ It is possible for the calls to doA2Phase1() in line 7 and doA3Phase2() in line 11 to execute in parallel.	✔	1.00	No. Since A2 and A3 are registered in SIG_WAIT mode on phaser ph, it is impossible for A2Phase1() in line 7 and doA3Phase2() in line 11 to execute in parallel due to the next operations on lines 7 and 10.
☐ Task A4 (created in line 12) can complete before the other tasks have started.	✔	1.00	No. Task A4 must wait for the other three tasks to perform their next operations before A4 can complete.

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

Name 1: _____

Name 2: _____

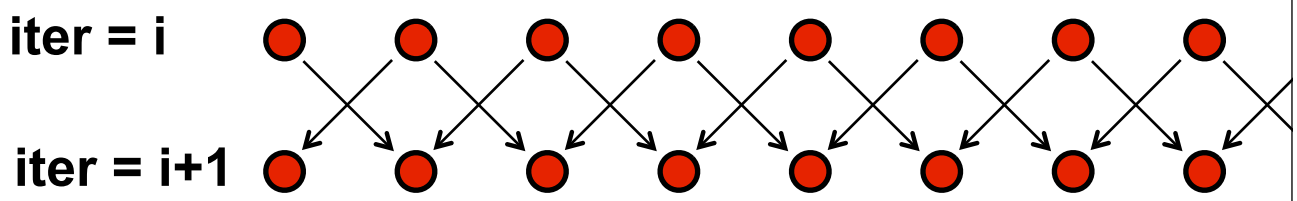


Complete the phased clause below to implement the left-right neighbor synchronization shown above

```
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.}
```


One-Dimensional Iterative Averaging with Point-to-Point Synchronization (w/o chunking)

```
1. double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
2. gVal[n+1] = 1; gNew[n+1] = 1;
3. phaser ph = new phaser[n+2];
4. finish { // phasers must be allocated in finish scope
5.   forall(point [i]:[0:n+1]) ph[i] = new phaser();
6.   forasync(point [j]:[1:n]) phased(ph[j]<phaserMode.SIG>,
7.                    ph[j-1]<phaserMode.WAIT>,ph[j+1]<phaserMode.WAIT>){
8.     double[] myVal = gVal; double[] myNew = gNew; // Local pointers
9.     for (point [iter] : [0:numIters-1]) {
10.      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
11.      next; // Point-to-point synchronization
12.      // Swap myVal and myNew
13.      double[] temp=myVal; myVal=myNew; myNew=temp;
14.      // myNew becomes input array for next iter
15.    } // for-iter
16.  } // forasync-j
17.} // finish
```



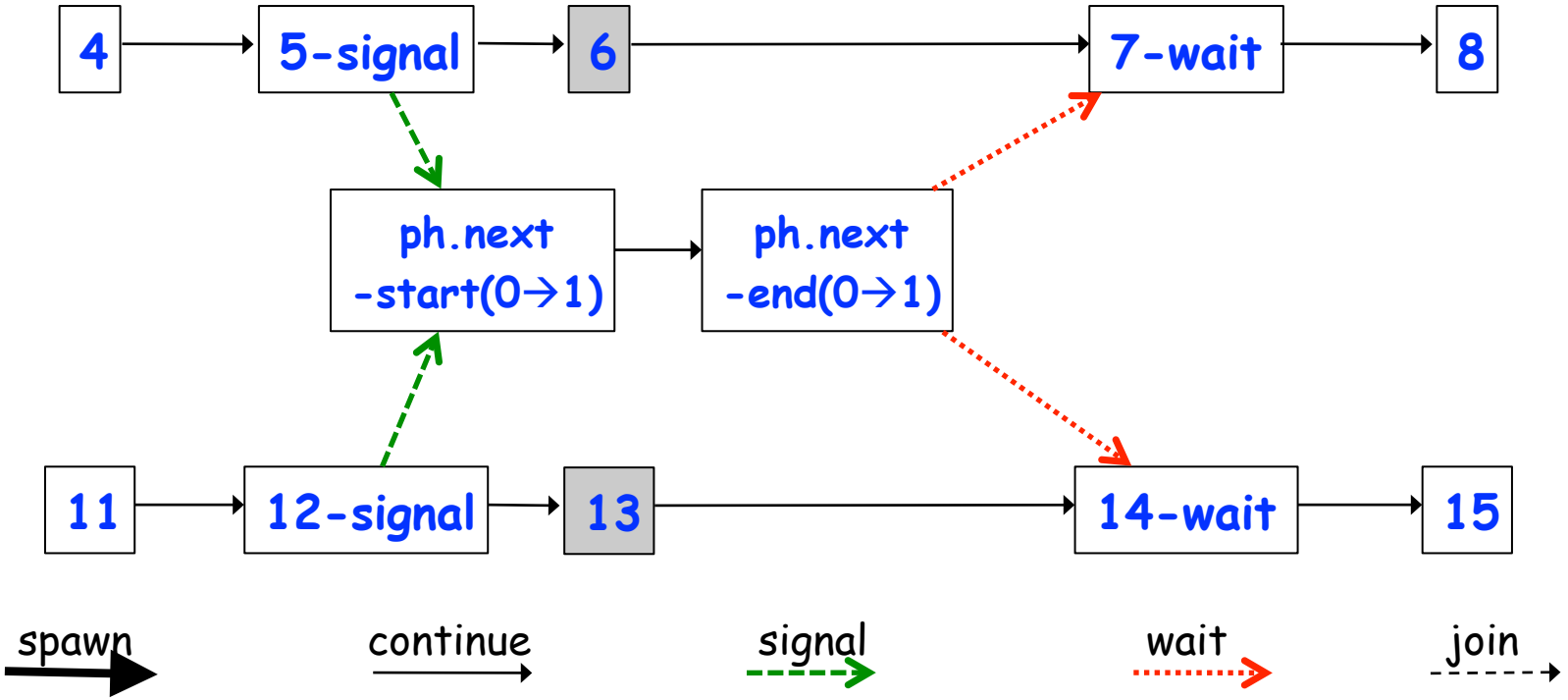
Signal statement

- When a task T performs a signal operation, it notifies all the phasers it is registered on that it has completed all the work expected by other tasks in the current phase ("shared" work).
  - —Since signal is a non-blocking operation, an early execution of signal cannot create a deadlock.
- Later, when T performs a next operation, the next degenerates to a wait since a signal has already been performed in the current phase.
- The execution of "local work" between signal and next is performed during phase transition
  - —Referred to as a "split-phase barrier" or "fuzzy barrier"

Example of Split-Phase Barrier (Listing 50)

```
1  finish {
2    phaser ph = new phaser(phaserMode.SIG_WAIT);
3    async phased { // Task T1
4      a = ... ;    // Shared work in phase 0
5      signal;      // Signal completion of a's computation
6      b = ... ;    // Local work in phase 0
7      next;        // Barrier — wait for T2 to compute x
8      b = f(b,x); // Use x computed by T2 in phase 0
9    }
10   async phased { // Task T2
11     x = ... ;    // Shared work in phase 0
12     signal;      // Signal completion of x's computation
13     y = ... ;    // Local work in phase 0
14     next;        // Barrier — wait for T1 to compute a
15     y = f(y,a); // Use a computed by T1 in phase 0
16   }
17 } // finish
```


Computation Graph for Split-Phase Barrier Example (without async and finish nodes and edges)



Week 5 & 6 Lecture Quiz Solution, Question 7: which of the following are true?

Your Answer		Score	Explanation
☐ Statement 4 can execute in parallel with statement 15	✔	1.00	No. The next statements in lines 7 and 14 prevent statement 4 from running in parallel with statement 15. (Both Tasks T1 and T2 are registered in SIG_WAIT mode on phaser ph.)
☑ Statement 6 can execute in parallel with statement 15	✔	1.00	Yes. Statement 6 appears after the signal statement in task T1. Thus, it is possible for task T2 to reach statement 15 before statement 6 has completed.
☐ The "next" operation in line 18 involves synchronization on phaser ph	✔	1.00	No. Since line 18 is outside the finish-scope in which phaser ph is allocated, that next statement only applies to any other phasers that may have been allocated outside the finish in lines 1-17. If there are none, then the next just becomes a no-op.

Announcements

- No lecture on Friday, Feb 22nd.
- No labs or lab quizzes this week
- No new lecture quiz this week. The lecture quiz for Weeks 5 & 6 was due on Tuesday night.
- Homework 3 is due by by 11:55pm on Friday, February 22, 2013
- Take-home midterm exam (Exam 1) will be given after lecture on Wednesday, February 20, 2013
  - Will need to be returned to Sherry Nassar (Duncan Hall 3137) by 4pm on Friday, February 22, 2013
  - Closed-book, closed computer written exam that can be taken in any 2-hour duration during that period