COMP 322: Fundamentals of Parallel Programming

Lecture 19: Midterm Review

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

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

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

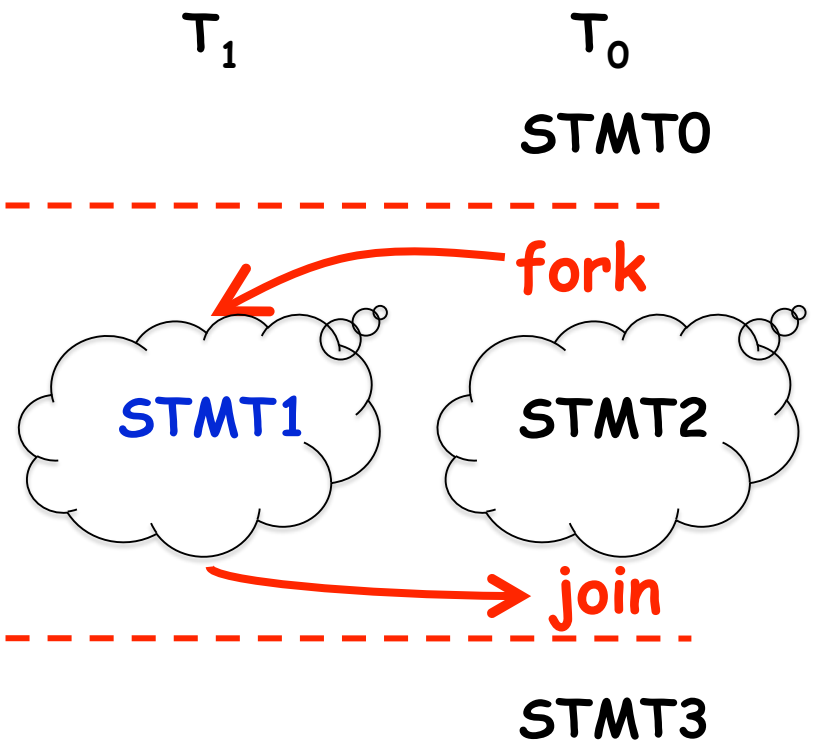
async S

- 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
```



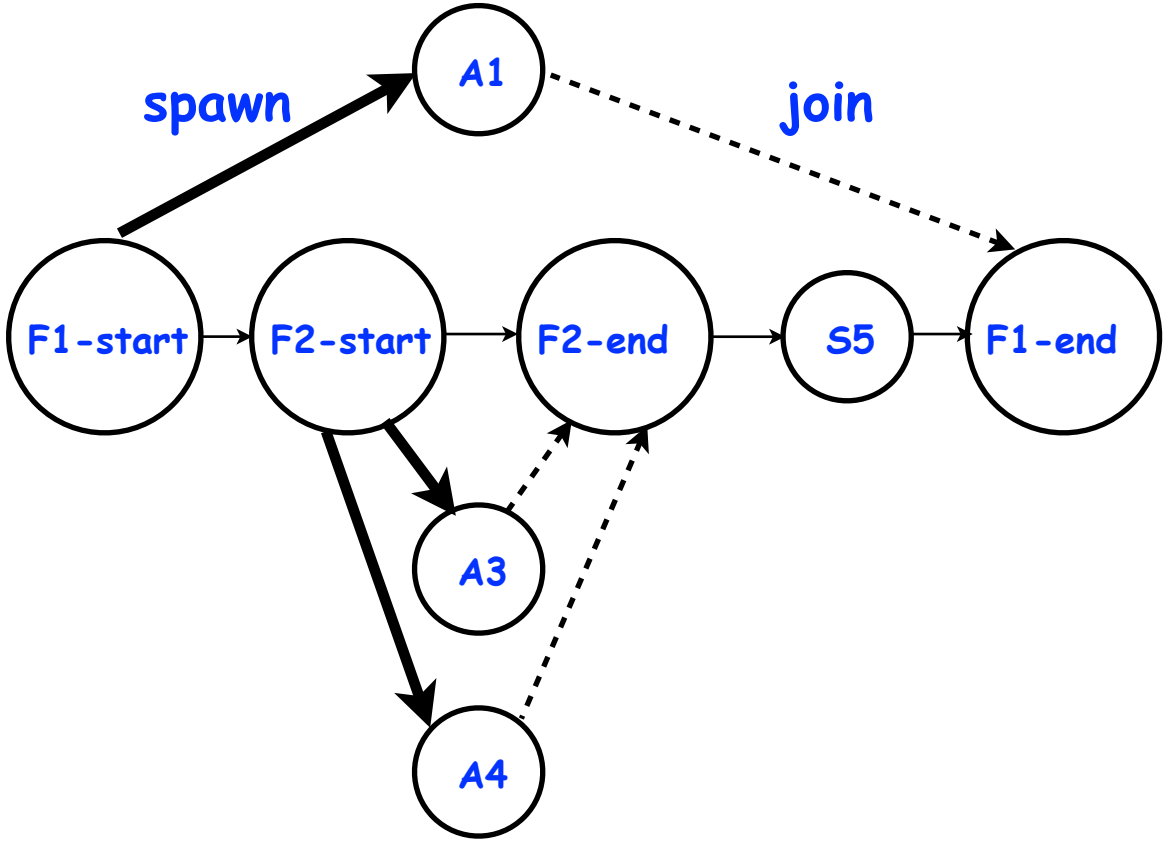
Computation Graphs for HJ Programs (Lecture 2)

- A Computation Graph (CG) captures the dynamic execution of an HJ 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

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

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

Computation Graph



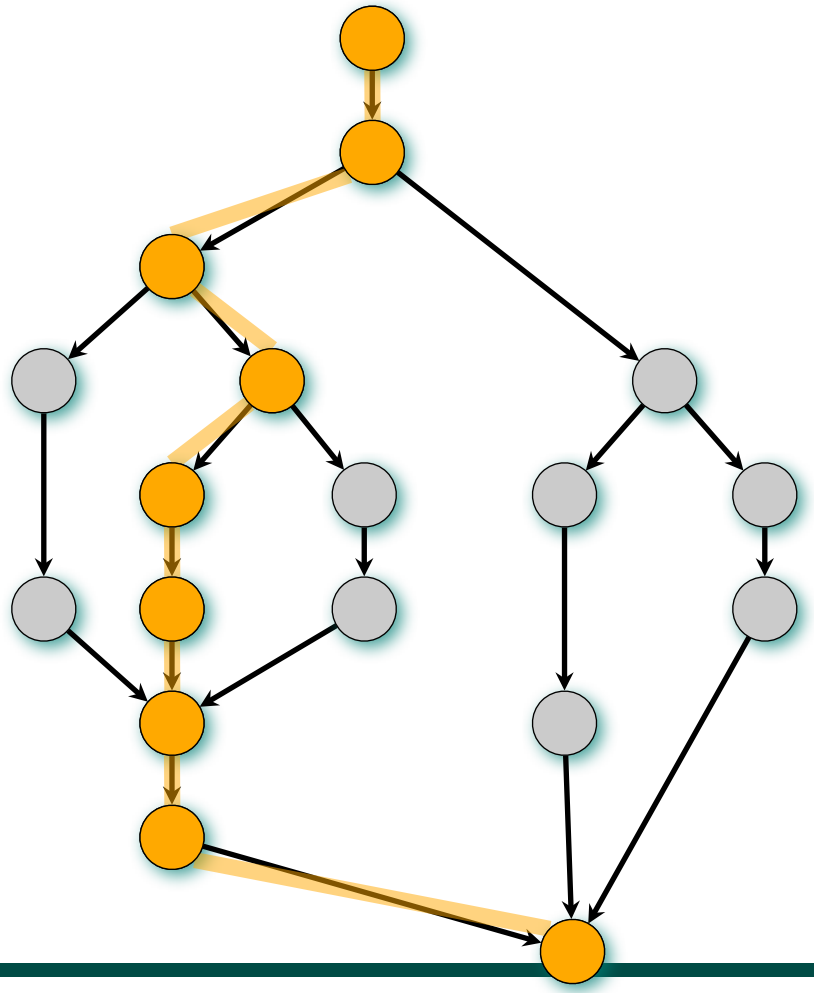
Complexity Measures for Computation Graphs

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)

Example (contd)

- Assume time(N) = 1 for all nodes in this graph



CPL(G) = 9

Ideal speedup
= WORK(G)/CPL(G)
= 2

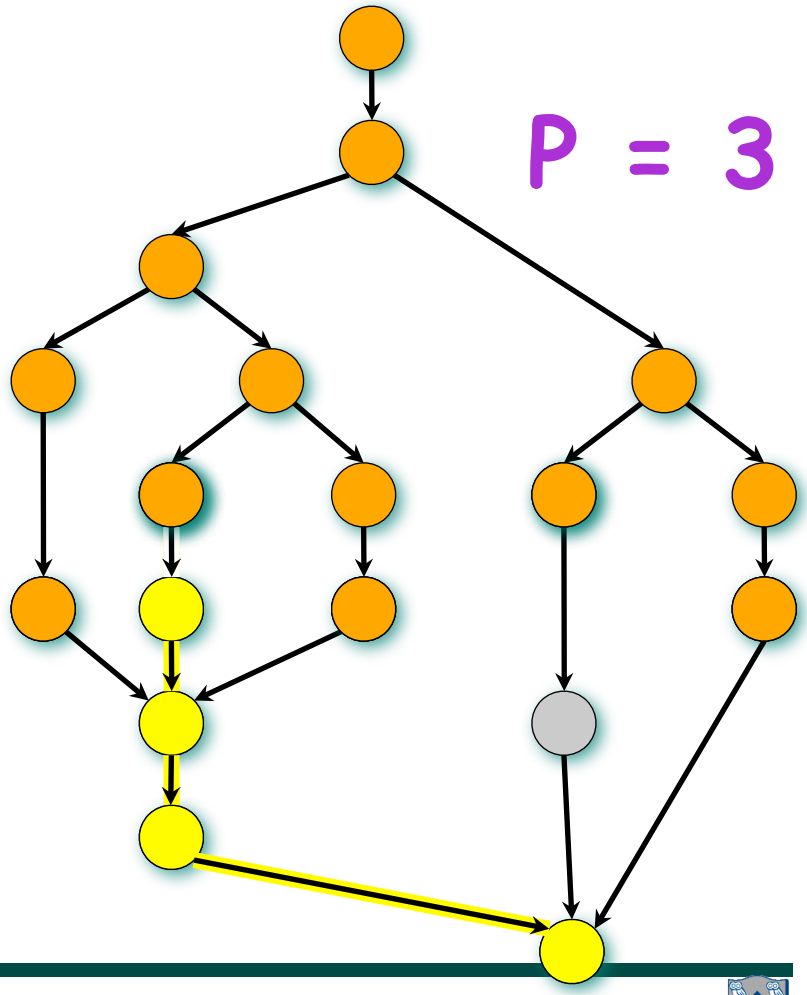
Lower Bounds on Execution Time (Lecture 3)

- Let T_P = execution time of computation graph on P processors
  - Assume an idealized machine where node N takes TIME(N) regardless of which processor it executes on, and that there is no overhead for creating parallel tasks
- Observations
  - T_1 = WORK(G)
  - T_∞ = CPL(G)
- Lower bounds
  - Capacity bound: $T_P \geq$ WORK(G)/P
  - Critical path bound: $T_P \geq$ CPL(G)
- Putting them together
  - $T_P \geq$ max(WORK(G)/P, CPL(G))

Upper Bound on Execution Time: Greedy-Scheduling Theorem

Theorem [Graham '66]. *Any greedy scheduler achieves*

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



Proof sketch:

- Define a time step to be complete if ≥ P nodes are ready at that time, or incomplete otherwise

complete time steps ≤ WORK(G)/P, since each complete step performs P work.

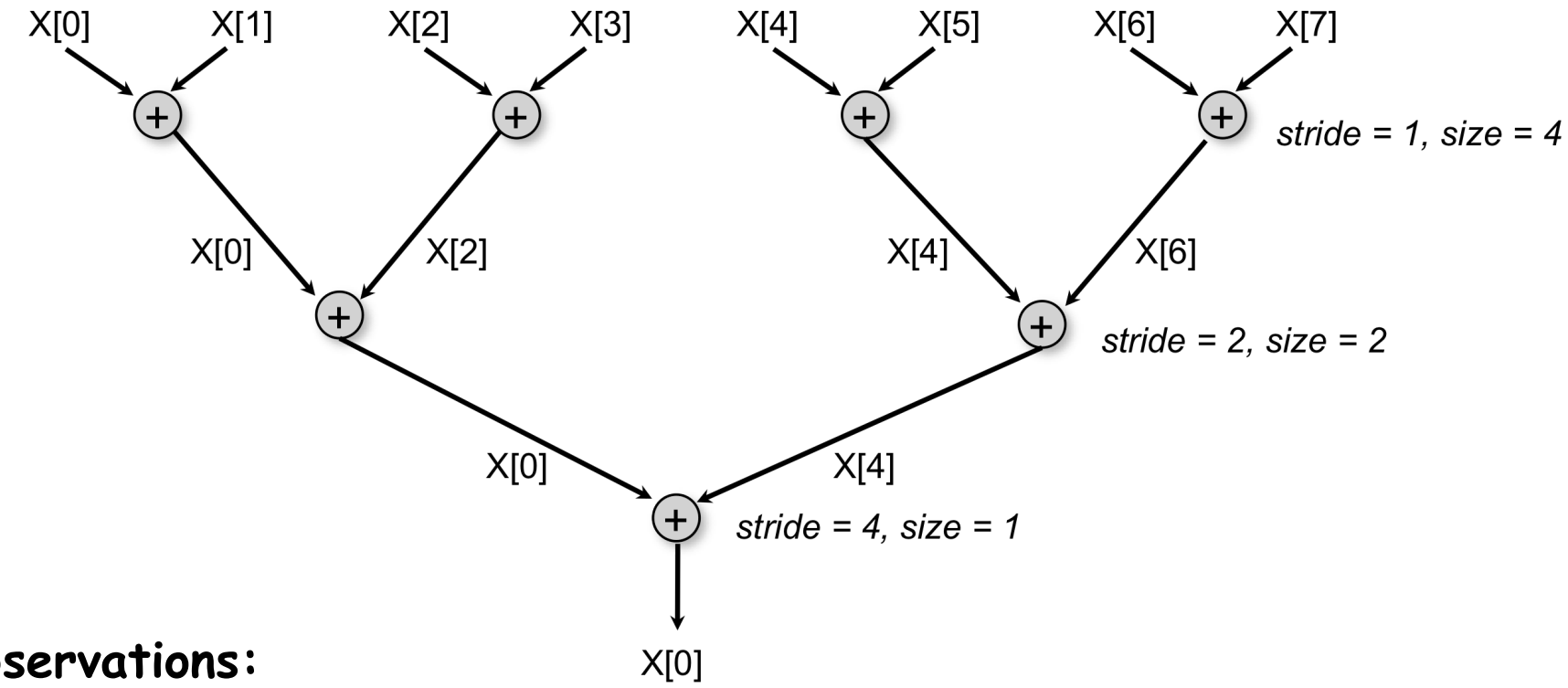
incomplete time steps ≤ CPL(G), since each incomplete step reduces the span of the unexecuted dag by 1.

ArraySum1: computing the sum of arbitrary sized arrays

```
for ( int stride = 1; stride < X.length ; stride *= 2 ) {
  // Compute size = number of additions to be performed in stride
  int size=ceilDiv(X.length,2*stride);
  finish for(int i = 0; i < size; i++)
    async {
      if ( (2*i+1)*stride < X.length )
        X[2*i*stride]+=X[(2*i+1)*stride];
    } // finish-for-async
} // for

// Divide x by y, round up to next largest int, and return result
static int ceilDiv(int x, int y) { return (x+y-1) / y; }
```


Reduction Tree Schema for computing Array Sum in parallel



Observations:

- This algorithm overwrites X (make a copy if X is needed later)
- stride = distance between array subscript inputs for each addition
- size = number of additions that can be executed in parallel in each level (stage)

ArraySum1 pre-pass when P < array length (Lecture 4)

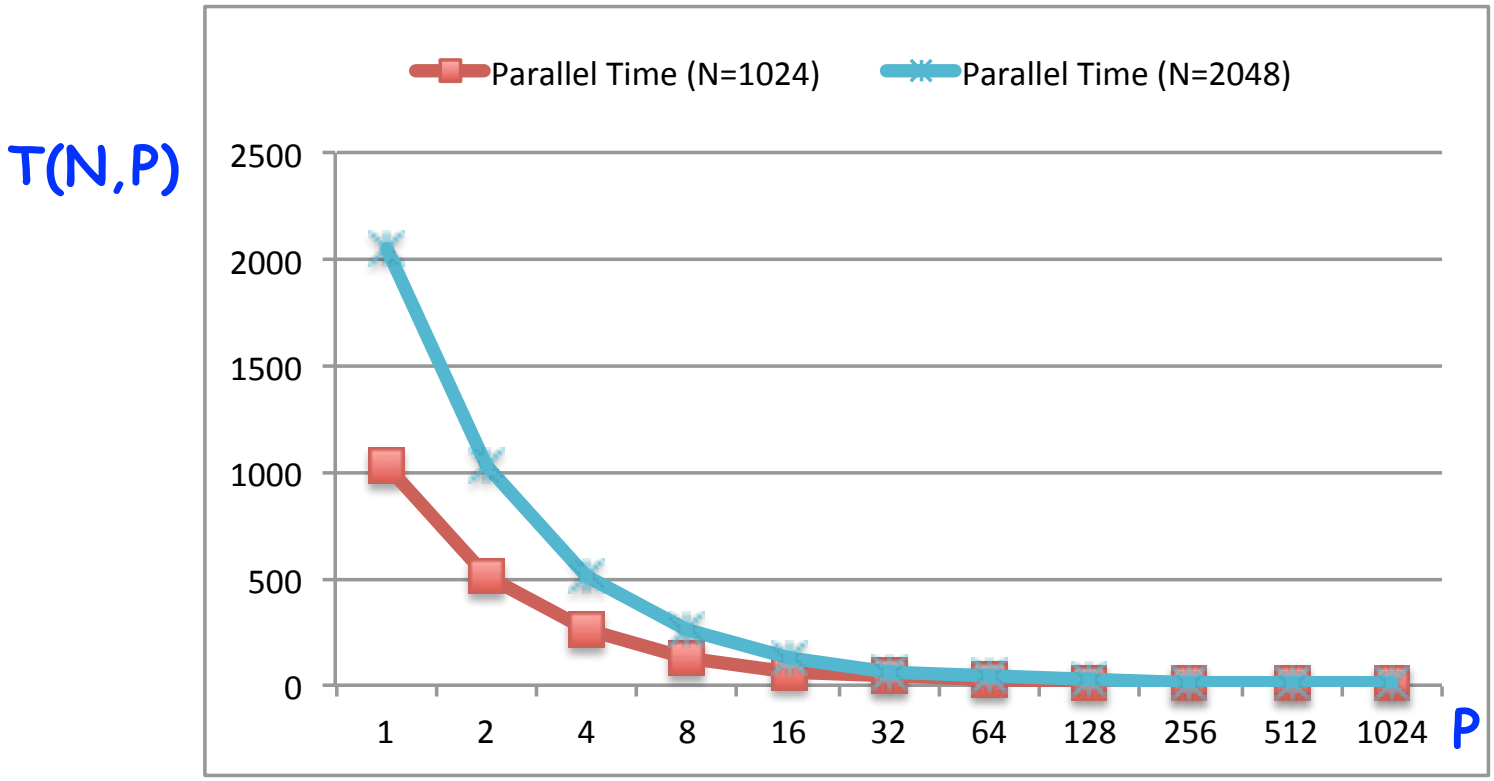
```
1. // Start of pre-pass: compute P partial sums in parallel
2. finish for(int j = 0; j < P; j++) // Create P tasks
3.     async {
4.       // Compute sum of A[j],A[j+P],... in task (processor) j
5.       // Any other decomposition into P partial sums is fine too
6.       for(int i = j; i < A.length; i += P) X[j] += A[i];
7.     } // finish-for-async
8. // End of pre-pass: now X[0..P-1] has P partial sums of array A
9. // Use ArraySum1 algorithm (slide 5) to obtain total sum
```

Complexity analysis

- Parallel time for pre-pass in lines 1-7 = O(N/P), where N = A.length
- Parallel time for ArraySum1 algorithm = O(log P)
- Total parallel time, T(N,P) = O(N/P + log P)

ArraySum: Ideal Parallel Time as function of P

- Total parallel time, $T(N,P) = N/P + \log_2(\min(P,N))$, depends on
  - Input size, N
  - Number of processors, P



Async-Finish Exception Semantics (Lecture 5)

- Exceptions thrown by multiple async's are accumulated into a "MultipleExceptions" collection at their Immediately Enclosing Finish

```
1.  try {
2.    finish for (int i = 0; i < size; i++)
3.      async {
4.        // Add explicit ArrayIndexOutOfBoundsException with X[-1]
5.        X[2*i*step] += X[(2*i+1)*step] + X[-1];
6.      } // finish-for-async
7.    } // try
8.  catch (Throwable t) {
9.     if (t instanceof hj.lang.MultipleExceptions)
10.      ... // Process the collection, t.exceptions
11.    else // single exception
12.      ... // Process t
13. }
```


Formal Definition of Data Races

Formally, 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., there is no path of dependence edges from S1 to S2 or from S2 to S1 in *CG*, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write.

Data races are challenging because of

- Nondeterminism: different executions of the parallel program with the same input may result in different outputs.
- Debugging and Testing: it is usually impossible to guarantee that all possible orderings of the accesses to a location will be encountered during program debugging and testing.

Data Race Example

```
// Incorrect parallel version
for ( p = first; p != null; p = p.next)
    async p.x = p.y + p.z;
```

Image source: http://www.motorcycle.com/images content/Review/6crf1022.jpg

```
for ( p = first; p != null; p = p.next)
    sum += p.x;
```

Image source: http://users.rcn.com/hwbingham/lexbike/bike.gif

- Race between Honda motorcycle (writing p.x) and Minuteman bicycle (reading p.x)
  - Who will get there first?

java.util.concurrent.atomic.AtomicInteger (Lecture 6)

- Constructors
  - `new AtomicInteger()`
    - Creates a new AtomicInteger with initial value 0
  - `new AtomicInteger(int initialValue)`
    - Creates a new AtomicInteger with the given initial value
- Selected methods
  - `int addAndGet(int delta)`
    - Atomically adds delta to the current value of the atomic variable, and returns the new value
  - `int getAndAdd(int delta)`
    - Atomically returns the current value of the atomic variable, and adds delta to the current value
- Similar interfaces available for `LongInteger`
  - No worry about lower/upper half issues when using a LongInteger atomic variable

Work-Sharing Pattern using AtomicInteger

```
1.  import java.util.concurrent.atomic.AtomicInteger;
2.  . . .
3.  String[] X = ... ; int numTasks = ...;
4.  AtomicInteger a = new AtomicInteger();
5.  . . .
6.  finish for (int i=0; i<numTasks; i++ )
7.    async {
8.      do {
9.         int j = a.getAndAdd(1);
10.        // can also use a.getAndIncrement()
11.        if (j >= X.length) break;
12.        . . . // Process X[j]
13.     } while (true);
14.   } // finish-for-async
```


Solution Counting Pattern using AtomicInteger

```
1.  import java.util.concurrent.atomic.AtomicInteger;
2.  . . .
3.  AtomicInteger count = new AtomicInteger();
4.  finish nqueens_kernel(new int[0], 0);
5.  . . .
6.  void nqueens_kernel(int [] a, int depth) {
7.    if (size == depth) count.addAndGet(1);
8.    else
9.      /* try each possible position for queen at depth */
10.     for (int i =  0; i < size; i++) async {
11.       /* allocate a temporary array and copy array a into it */
12.       int [] b = new int [depth+1];
13.       System.arraycopy(a, 0, b, 0, depth);
14.       b[depth] = i;
15.       if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
16.     } // for-async
17. } // nqueens_kernel()
```


HJ Futures: Tasks with Return Values (Lecture 7)

```
async<T> { <Stmt-Block> }
```

- Creates a new child task that executes Stmt-Block, which must terminate with a return statement returning a value of type T
- Async expression returns a reference to a container of type future<T>
- Values of type future<T> can only be assigned to final variables

```
Expr.get()
```

- Evaluates Expr, and blocks if Expr's value is unavailable
- Expr must be of type future<T>
- Return value from Expr.get() will then be T
- 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

```
1.  // Parent Task T1 (main program)
2.  // Compute sum1 (lower half) and sum2 (upper half) in parallel
3.  final future<int> sum1 = async<int> { // Future Task T2
4.    int sum = 0;
5.    for(int i=0 ; i < X.length/2 ; i++) sum += X[i];
6.    return sum;
7.  }; //NOTE: semicolon needed to terminate assignment to sum1
8.  final future<int> sum2 = async<int> { // Future Task T3
9.    int sum = 0;
10.   for(int i=X.length/2 ; i < X.length ; i++) sum += X[i];
11.   return sum;
12. }; //NOTE: semicolon needed to terminate assignment to sum2
13. //Task T1 waits for Tasks T2 and T3 to complete
14. int total = sum1.get() + sum2.get();
```

Why are these semicolons needed?

Comparison of Future Task and Regular Async Versions of Two-Way Array Sum

- Future task version initializes two references to future objects, sum1 and sum2, and both are declared as final
- No finish construct needed in this example
  - Instead parent task waits for child tasks by performing sum1.get() and sum2.get()
- Guaranteed absence of race conditions in Future Task example
  - No race on sum because it is a local variable in tasks T2 and T3
  - No race on future variables, sum1 and sum2, because of blocking-read semantics

Extending HJ Futures for Macro-Dataflow (Lecture 8): Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)

```
ddfA = new DataDrivenFuture();
```

- Allocate an instance of a data-driven-future object (container)

```
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 in ddfA, thereby making ddfA available
- Single-assignment rule: at most one put is permitted on a given DDF

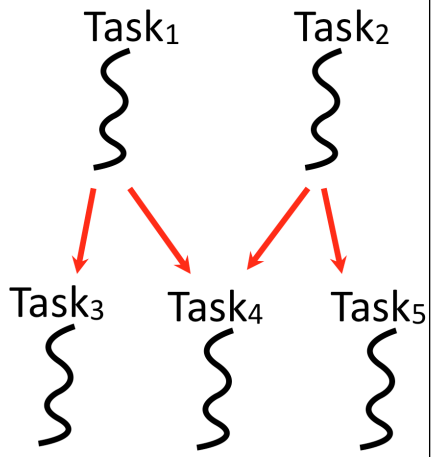
```
ddfA.get()
```

- Return value 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)

Example Habanero Java code fragment with Data-Driven Futures

```
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



Differences between Futures and DDFs/DDTs

- Consumer blocks on get() for each future that it reads, whereas async-await does not start execution till all DDFs are available
- Producer task can only write to a single future object, where as a DDF task 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 DDF task
- Future tasks cannot deadlock, but it is possible for a DDF task to never be enabled, if one of its input DDFs never becomes available. This can be viewed as a special case of deadlock.
  - This deadlock case can be resolved by ensuring that each finish construct moves past the end-finish when all enabled async tasks in its scope have terminated, thereby ignoring any remaining non-enabled async tasks.

seq clause in HJ async statement (Lecture 9)

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

- seq clause specifies condition under which async should be executed sequentially

```
1. void fib (int n) {
2.     if (n<2) {
3.         . . .
4.     } else {
5.         finish {
6.             async seq(n <= THRESHOLD) fib(n-1);
7.             async seq(n <= THRESHOLD) fib(n-2);
8.         }
9.      } // if-else
10.} // fib()
```


hj.lang.point, an index type for multi-dimensional loops

- A point is an element of an n-dimensional Cartesian space (n>=1) with integer-valued coordinates e.g., [5], [1, 2], …
  - Dimensions of a point are numbered from 0 to n-1
  - n is also referred to as the rank of the point
- A point variable can hold values of different ranks e.g.,
  - point p; p = [1]; … p = [2,3]; …
- The following operations are defined on point-valued expression p1
  - p1.rank --- returns rank of point p1
  - p1.get(i) --- returns element i of point p1
    - Returns element (i mod p1.rank) if i < 0 or i >= p1.rank
  - p1.lt(p2), p1.le(p2), p1.gt(p2), p1.ge(p2)
    - Returns true iff p1 is lexicographically <, <=, >, or >= p2
    - Only defined when p1.rank and p1.rank are equal

hj.lang.region, a rectangular iteration space for multi-dimensional loops

A region is the set of *points* contained in a rectangular subspace

A region variable can hold values of different ranks e.g.,

- region R; R = [0:10]; … R = [-100:100, -100:100]; … R = [0:-1]; …

Operations

- R.rank ::= # dimensions in region;
- R.size() ::= # points in region
- R.contains(P) ::= predicate if region R contains point P
- R.contains(S) ::= predicate if region R contains region S
- R.equal(S) ::= true if region R equals region S
- R.rank(i) ::= projection of region R on dimension i (a one-dimensional region)
- R.rank(i).low() ::= lower bound of i^{th} dimension of region R
- R.rank(i).high() ::= upper bound of i^{th} dimension of region R
- R.ordinal(P) ::= ordinal value of point P in region R
- R.coord(N) ::= point in region R with ordinal value = N

Summary of forasync statement

```
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"

Pointwise sequential for loop

- HJ extends Java's for loop to support sequential iteration over points in region R in canonical lexicographic order
  - `for ( point p : R ) . . .`
- Standard point operations can be used to extract individual index values from point p
  - `for ( point p : R ) { int i = p.get(0); int j = p.get(1); . . . }`
- Or an "exploded" syntax is commonly used instead of explicitly declaring a point variable
  - `for ( point [i,j] : R ) { . . . }`
- The exploded syntax declares the constituent variables (i, j, …) as local int variables in the scope of the for loop body

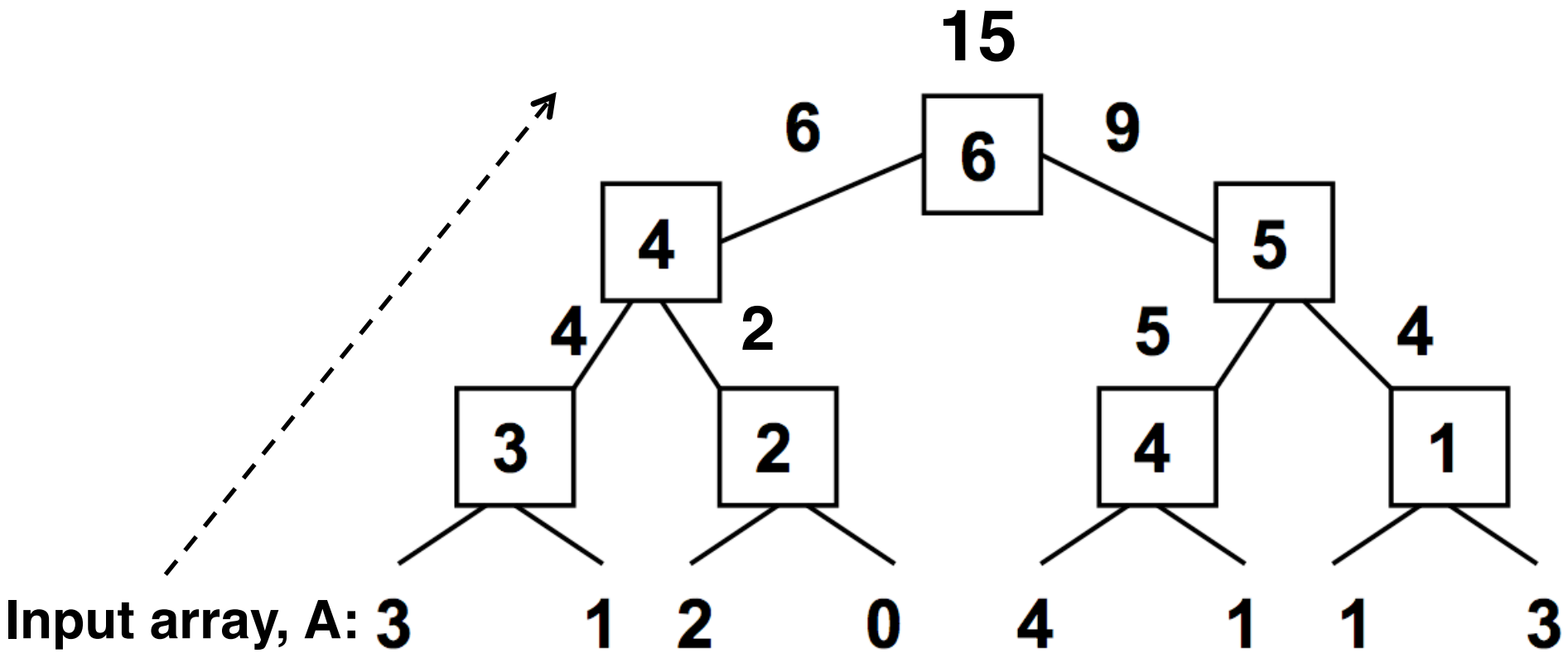
Example: HJ code for One-Dimensional Iterative Averaging with chunked for-finish-forasync-for (Lecture 10)

```
1. for (point [iter] : [0:iterations-1]) {
2.   // Compute MyNew as function of input array MyVal
3.   int Cj = ...; // Set to desired number of chunks
4.   finish forasync (point [jj]:[0:Cj-1]) {
5.     for (point [j]:getChunk([1:n],[Cj],[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
```

- How many tasks does this chunked version create?

Parallel Prefix Sum: Upward Sweep

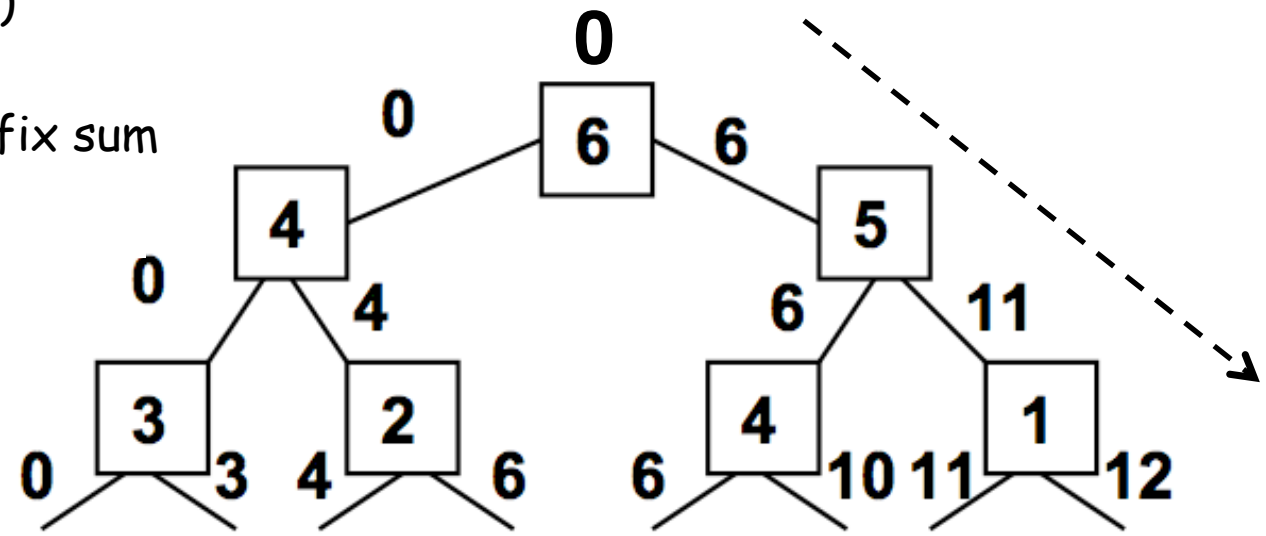
1. Receive values from children
2. Store left value in box (will contribute to prefix sum for right subtree in downward sweep)
3. Send left+right value to parent



Parallel Prefix Sum: Downward Sweep

1. Receive value from parent (root receives 0)
2. Send parent's value to left child (prefix sum for elements to left of left child's subtree)
3. Send parent+box value to right child (prefix sum for elements to left of right child's subtree)

Add A[i] to get final prefix sum

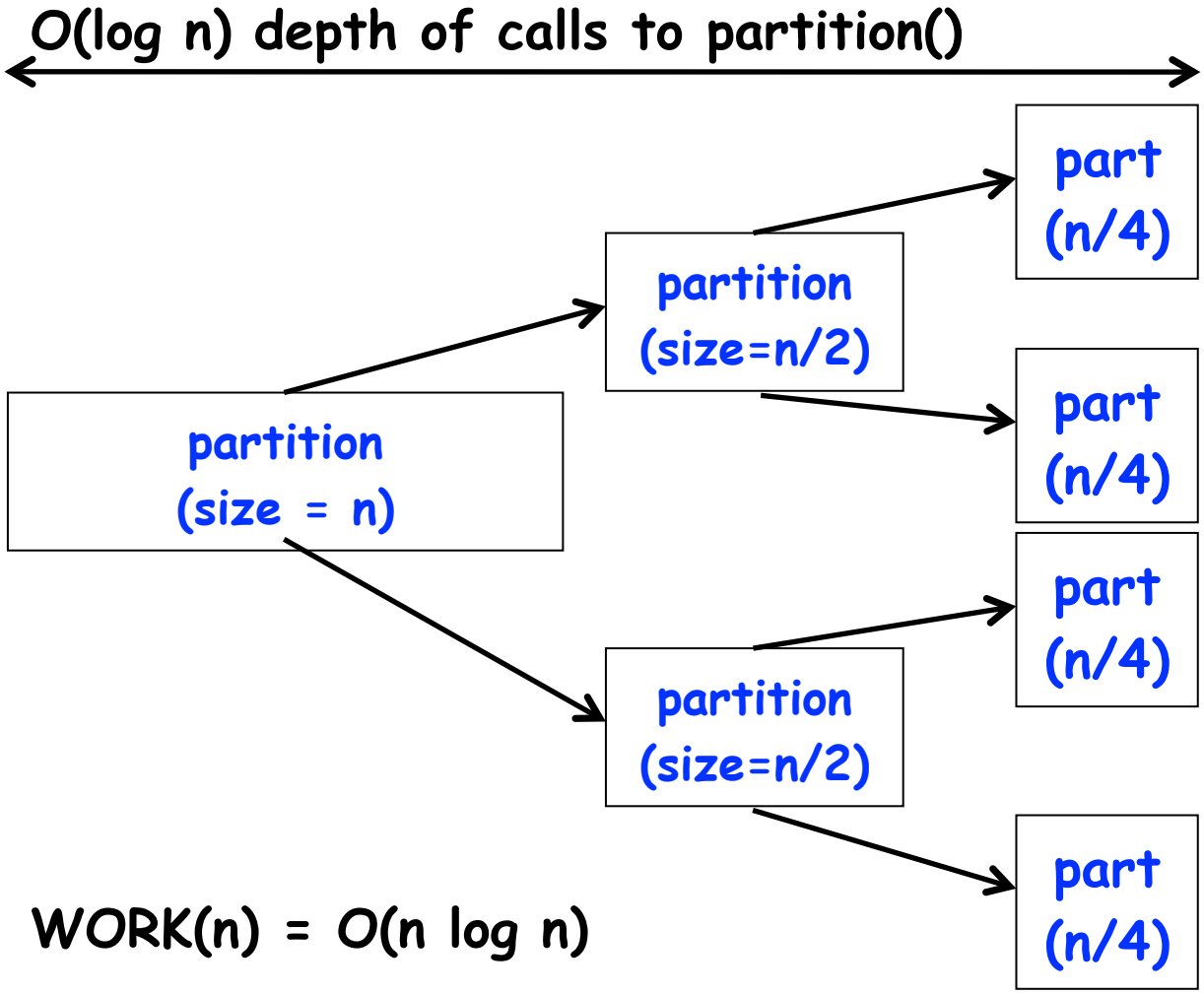


Prefix sum excluding A[i]	0	3	4	6	6	10	11	12
+ A[i]	3	1	2	0	4	1	1	3
= Prefix sum including A[i]	3	4	6	6	10	11	12	15

Two Opportunities in Parallelizing Quicksort (Lecture 11)

```
procedure Quicksort(S) {
    if S contains at most one element then return S
    else {
        choose an element a randomly from S;
        // Opportunity: Parallelize partitioning
        let S1, S2 and S3 be the sequences of elements in S less
        than, equal to, and greater than a, respectively;
        // Opportunity: Parallelize recursive calls
        return (Quicksort(S1) followed by S2 followed by
                Quicksort(S3))
     } // else
} // procedure
```

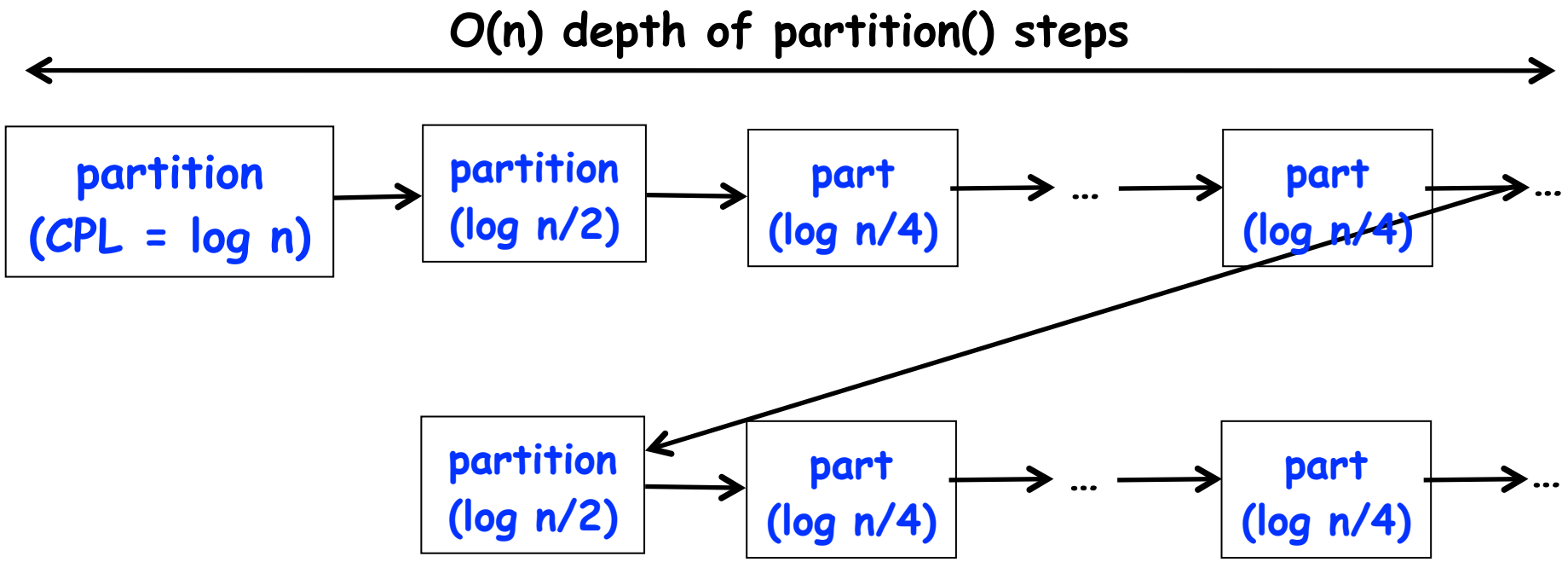

Approach 1: sequential partition, parallel calls



WORK(n) = O(n log n)

CPL(n) = O(n) + O(n/2) + O(n/4) + … = O(n)

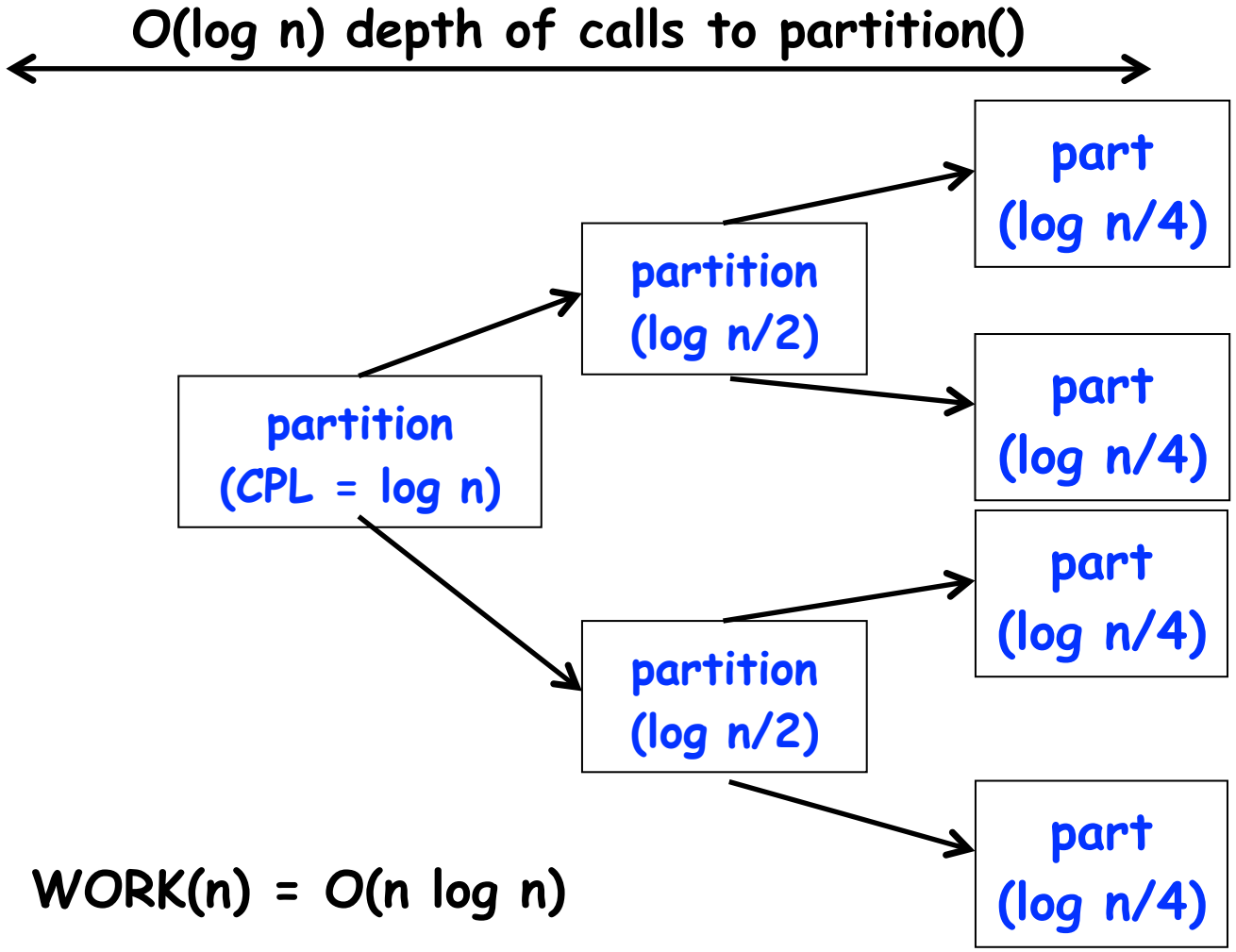
Approach 2: Parallel partition, sequential calls



WORK(n) = O(n log n)

CPL(n) = log(n) + 2 log(n/2) + 4 log(n/4) + … = O(n)

Approach 3: parallel partition, parallel calls



WORK(n) = O(n log n)

CPL(n) = O(log n) + O(log n/2) + O(log n/4) + … = $O(\log^2 n)$

Finish Accumulators in HJ (Lecture 12)

- Creation

```
accumulator ac = accumulator.factory.accumulator(operator, type);
```

  - operator can be Operator.SUM, Operator.PROD, Operator.MIN, or Operator.MAX
  - type can be int.class or double.class
  - extensions to support generic types, and user-defined operators and types are in progress

- Accumulation

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

  - data must be of type java.lang.Number, int, or double

- Retrieval

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

  - get() can only be performed outside finish scope that ac is registered with
  - get() is nonblocking because finish provides the necessary synchronization
  - result from get() will be deterministic if HJ program does not use atomic or isolated constructs and is data-race-free

Solution Counting Pattern using Finish Accumulators (NQueens revisited)

```
1.  static accumulator a;
2.  . . .
3.  a = accumulator.factory.accumulator(SUM, int.class);
4.  finish(a) nqueens_kernel(new int[0], 0);
5.  System.out.println("No. of solutions = " + a.get().intValue())
6.  . . .
7.  void nqueens_kernel(int [] a, int depth) {
8.    if (size == depth) a.put(1);
9.    else
10.     /* try each possible position for queen at depth */
11.     for (int i =  0; i < size; i++) async {
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()
```


Atomic Variables vs. Accumulators

Atomic variables

- **Pros:**
  - simple construct that can be used anywhere in HJ code
  - supports nondeterminism e.g., work-sharing example in Lecture 6
- **Cons:**
  - can be a sequential bottleneck with large number of simultaneous parallel accesses
  - supports nondeterminism

Finish accumulators

- **Pros:**
  - integration with finish structure guarantees determinism and reduces errors
  - supports more reduction operations (max, min, product) than AtomicInteger
  - lazy implementation with work-stealing schedulers is more scalable than AtomicInteger operations
- **Con:**
  - does not support nondeterminism

HJ's forall statement = finish + forasync + next (Lecture 16: Summary of Barriers and Phasers)

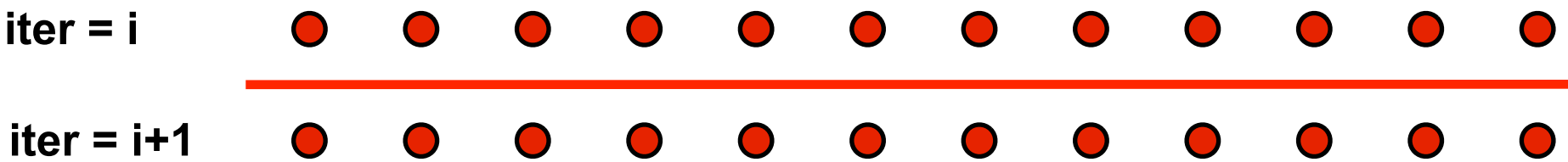
```
AtomicInteger rank = new AtomicInteger();
forall (point[i] : [0:m-1]) {
  int r = rank.getAndIncrement();
  System.out.println("Hello from task ranked " + r);
  next; // Acts as barrier between phases 0 and 1
  System.out.println("Goodbye from task ranked " + r);
}
```

Phase 0

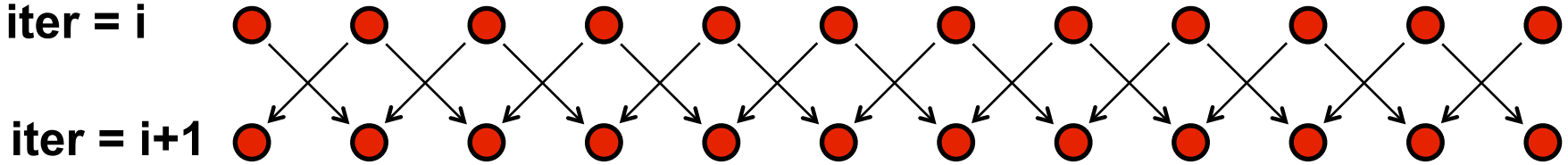
Phase 1

- 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 synchronization is the closest enclosing forall statement
  - Special case of "phaser" construct

Barrier vs Point-to-Point Synchronization for One-Dimensional Iterative Averaging Example



Barrier synchronization

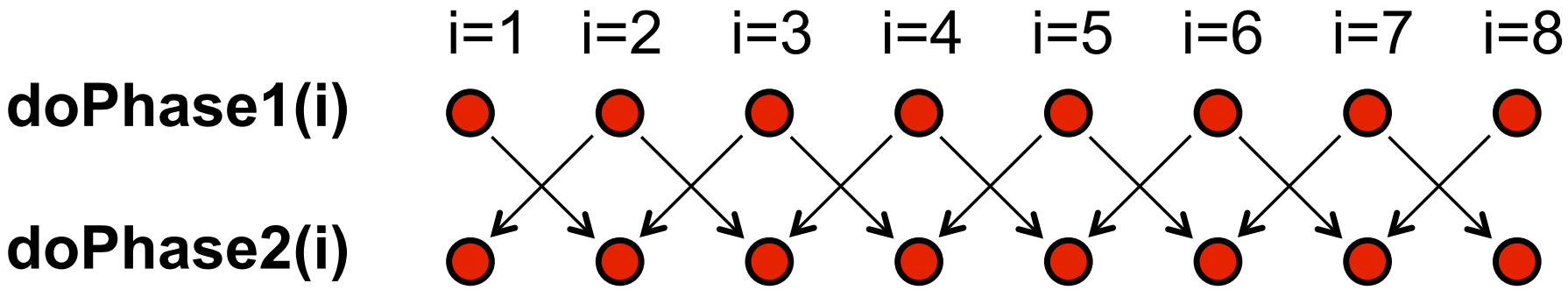


Point-to-point synchronization

Summary of Phaser Construct

- 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 has no relationship to Java wait/notify
- 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

Left-Right Neighbor Synchronization Example




```
1. finish {
2.   phaser[] ph = new phaser[m+2];
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; // Signal ph[i] & wait on ph[i-1], ph[i+1]
8.      doPhase2(i);
9.    }
10.}
```


Announcements

- Homework 3 due by 11:55pm on Friday, Feb 24th
  - Performance results for parts 2 and 3 of assignment must be obtained on Sugar (see Section 4)
- Exam 1 is a take-home exam
  - Maximum duration = 2 hours
  - Closed-book, closed-notes, closed-computer
  - Pick up exam from Amanda Nokleby's office (Duncan Hall 3137) any time starting 9am on Thursday, Feb 23rd
  - Return exam to Amanda's office by 4pm on Friday, Feb 24th
  - Written exam --- no penalty for minor syntactic errors in program text, so long as the meaning of the program is unambiguous.
  - If you believe there is any ambiguity or inconsistency in a question, you should state the ambiguity or inconsistency that you see, and any assumptions that you make to resolve it.
  - Scope of exam includes Lectures 1-16
    - Lectures 17 & 18 (Places) will be in scope for Exam 2