COMP 322: Fundamentals of Parallel Programming

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

Lecture 18: Midterm Review

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

Announcements

- No COMP 322 labs this week
- No lecture on Friday, Feb 25th
- Class survey to be conducted by undergraduate TAs, Max Grossman and Christopher Nunu, during spring break
  - Please make your best effort to participate. Your feedback will impact how COMP 322 is taught in the second half of the semester.
- Midterm exam to be handed out after today's lecture
  - 2-hour take-home written exam
    - Closed-book, closed-notes, closed-computer
  - Must be handed in to Amanda Nokleby in Duncan Hall Room 3137 by 5pm on Friday, Feb 25th
    - You can slide it under her door id she's not in
  - Scope of midterm exam will be Lectures 1-15 and Lecture 17
    - Lecture 16 (Bitonic Sort) will not be included in midterm exam

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

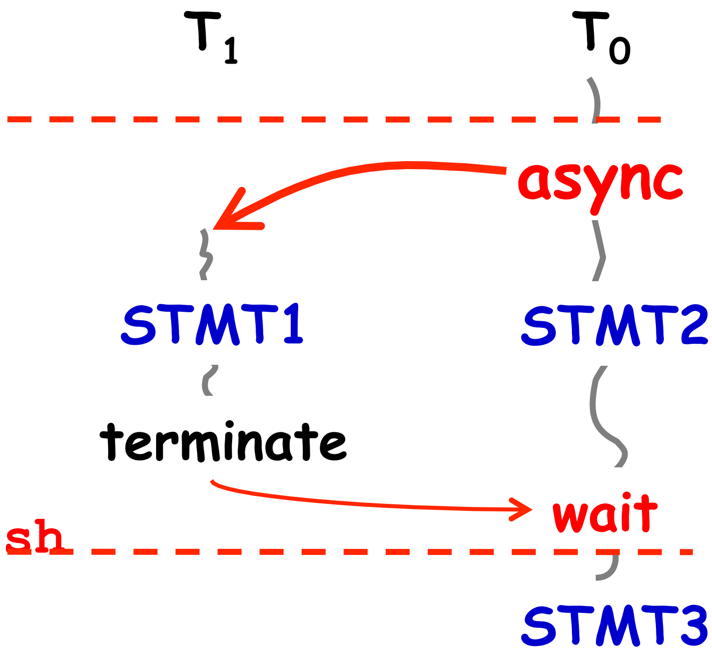
async S

- Creates a new child task that executes statement S
- Parent task immediately continues to statement following the async

finish S

- Execute S, but wait until *all* (transitively) spawned asyncs in S's scope have terminated.
- Implicit finish between start and end of main program

```
//Task T0(Parent)
finish {    //Begin finish
  async
    STMT1; //T1(Child)
  //Continuation
  STMT2;    //T0
} //Continuation //End finish
STMT3;     //T0
```

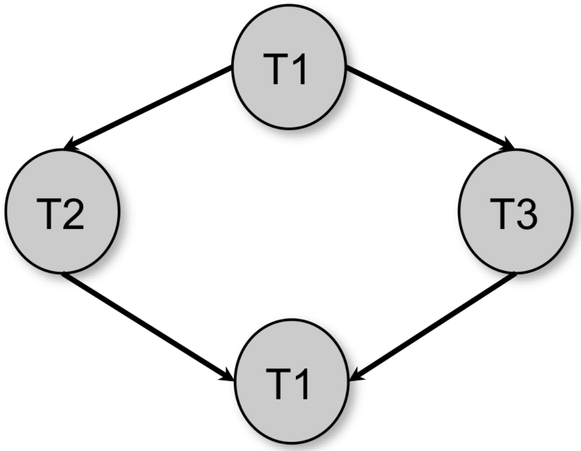


Example of a Parallel Program: Array Sum with two tasks (Lecture 1)

```
// Start of Task T1 (main program)
sum1 = 0; sum2 = 0;
// Assume that sum1 & sum2 are fields
finish {
  // Compute sum1 (lower half) and sum2
  // (upper half) in parallel
  async for (int i=0; i < X.length/2; i++)
      sum1 += X[i]; // Task T2
  async for (int i=X.length/2; i < X.length; i++)
      sum2 += X[i]; // Task T3
}
//Task T1 waits for Tasks T2 and T3
int sum = sum1 + sum2; // Continuation of Task T1
```

Computation Graph

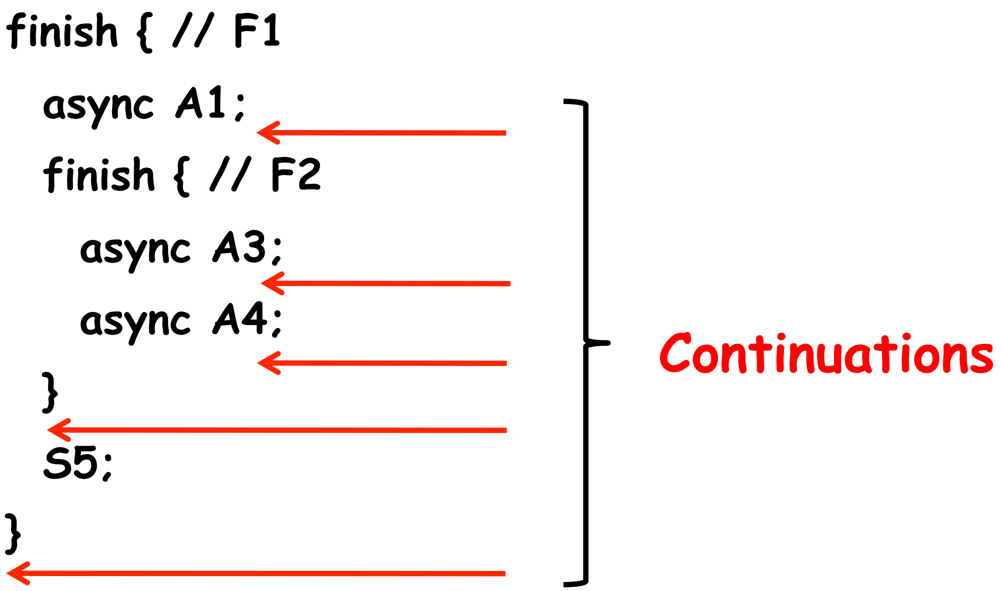
// Start of Task T1 (main program)



// Continuation of Task T1

Continuations (Lecture 2)

- A continuation is one of two kinds of program points
  - The point in the parent task immediately following an async
  - The point immediately following an end-finish
- Continuations are also referred to as task-switching points
  - Program points at which a worker may switch execution between different tasks



```
finish { // F1
  async A1;
  finish { // F2
    async A3;
    async A4;
  }
  S5;
}
```

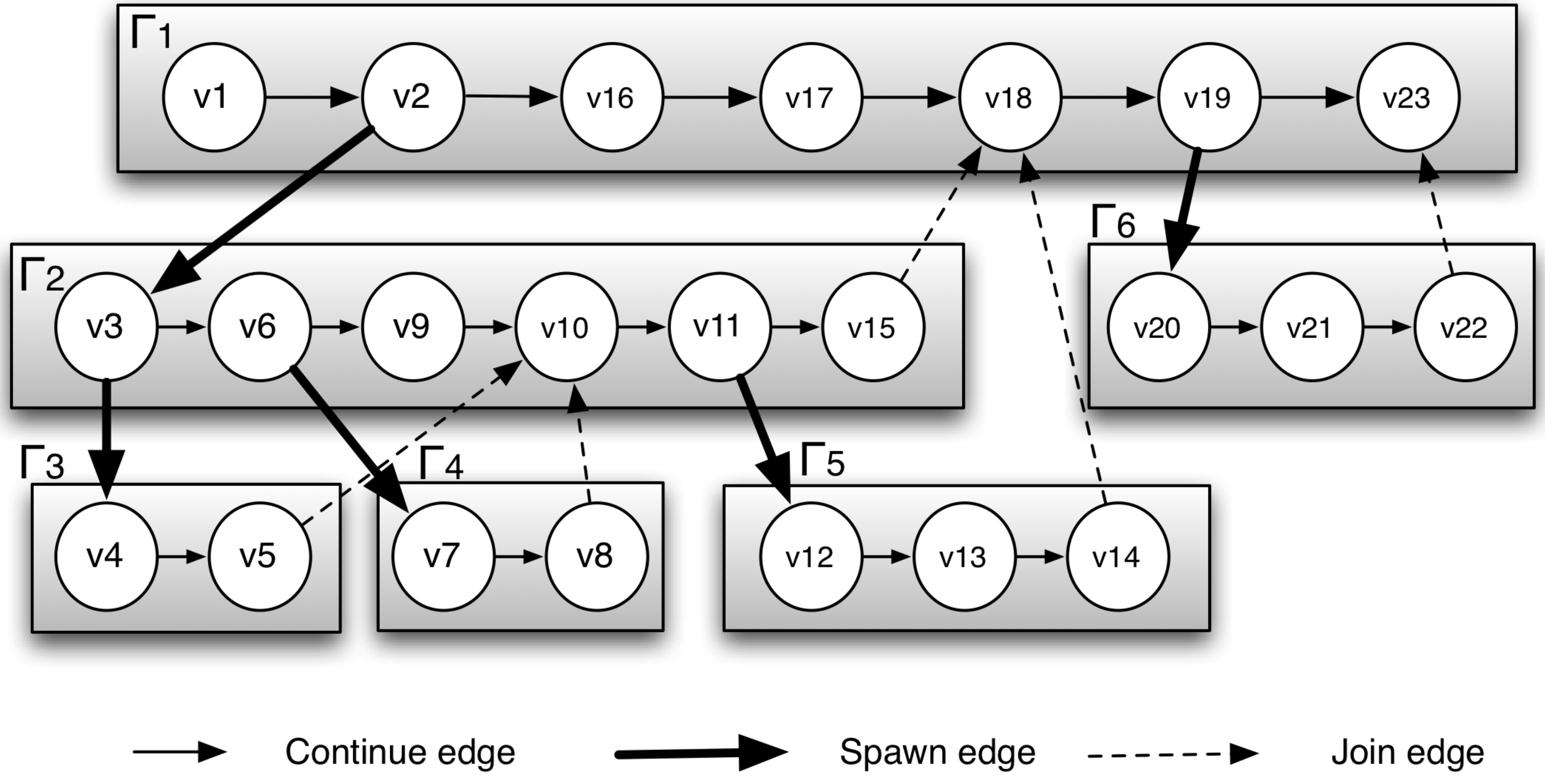
Computation Graphs for HJ Programs (Lecture 3)

- A Computation Graph (CG) is an abstract data structure that captures the dynamic execution of an HJ program
- The nodes in the CG are *steps* in the program's execution
  - A step is a sequential subcomputation of a task that contains no continuation points
  - When a worker starts executing a step, it can execute the entire step without interruption
  - Steps need not be maximal i.e., it is acceptable to split a step into smaller steps if so desired

Computation Graph Edges (Lecture 3)

- CG edges represent ordering constraints
- There are three kinds of CG edges of interest in an HJ program with finish &async operations
    1. *Continue* edges define sequencing of steps within a task
    2. *Spawn* edges connect parent tasks to child async tasks
    3. *Join* edges connect async tasks to their Immediately Enclosing Finish (IEF) operations

Computation Graph for previous HJ Example (Lecture 3)



Observation: Step v16 can potentially execute in parallel with steps v3 … v15

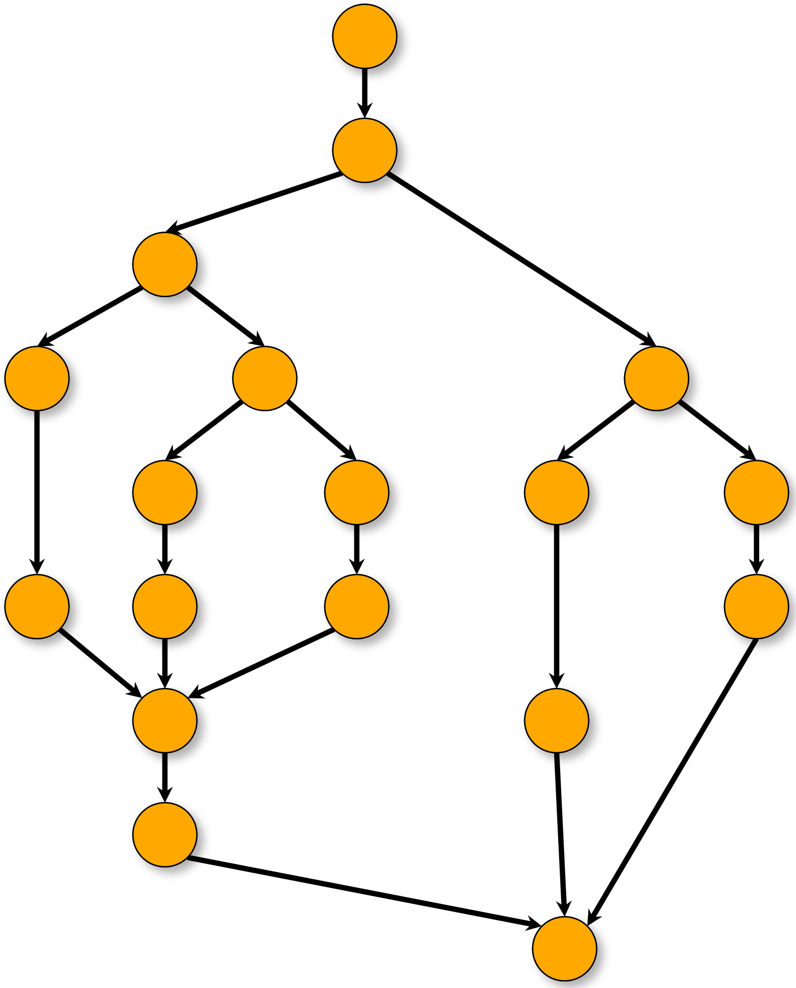
Complexity Measures for Computation Graphs (Lecture 3)

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 amount of work to be performed in G
- CPL(G) = length of a longest path in CG G, when adding up the 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 (Lecture 3)

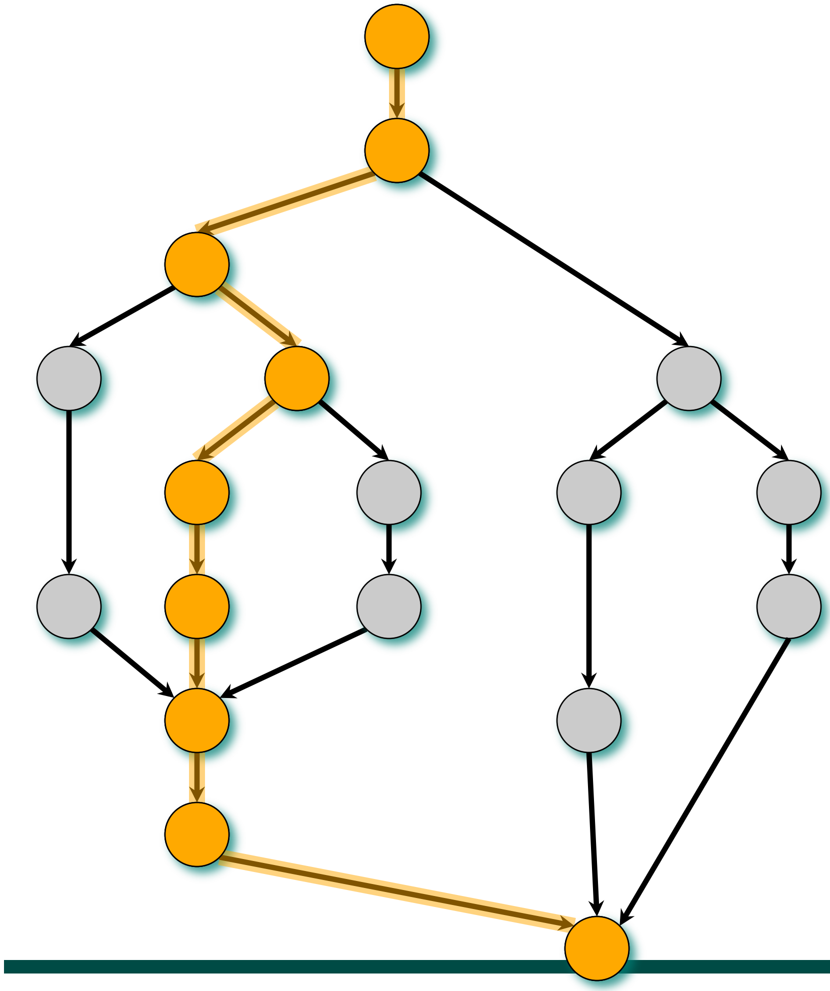
- Assume time(N) = 1 for all nodes in this graph



WORK(G) = 18

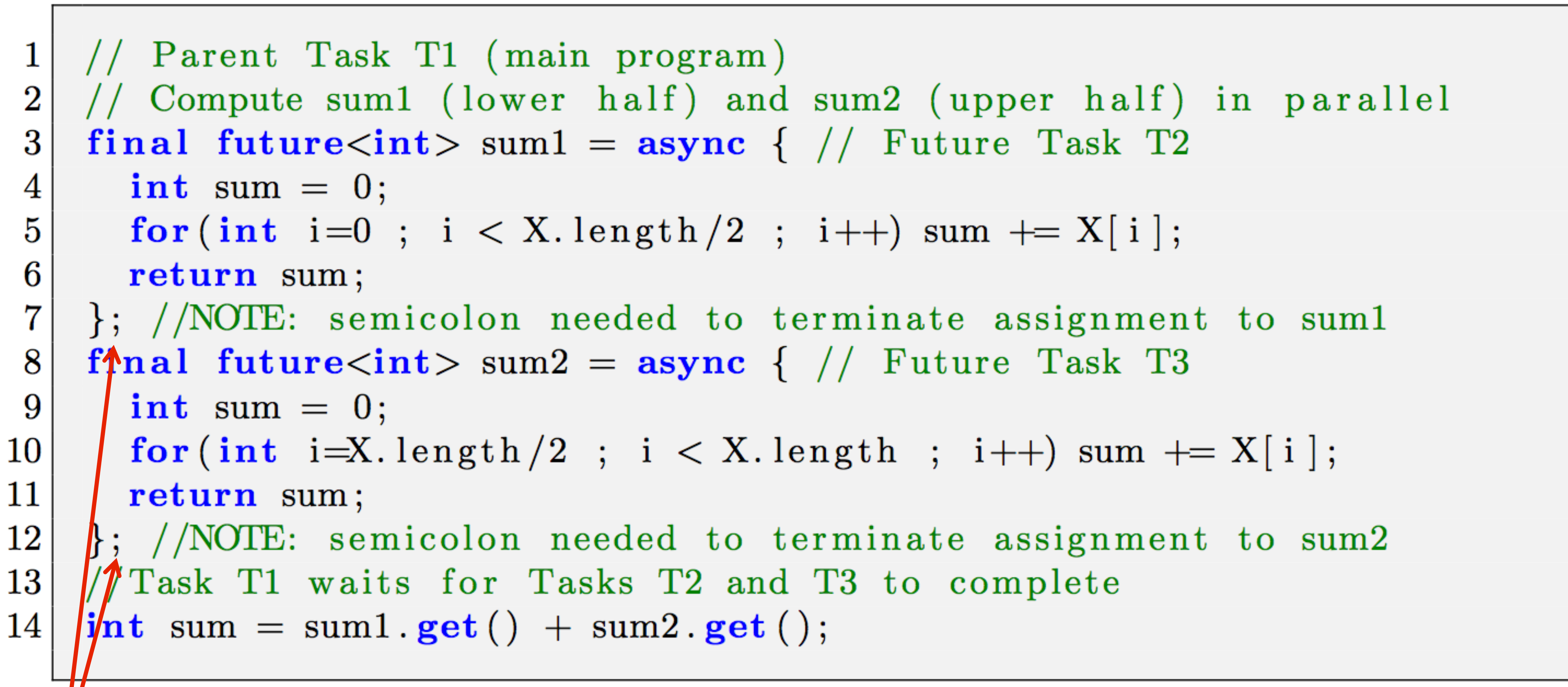
Example (contd, Lecture 3)

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



CPL(G) = 9

Example: Two-way Parallel Array Sum using Future Tasks (Lecture 4)



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

Listing 1: Two-way Parallel ArraySum using Future Tasks

Why are these semicolons needed?

Summing an arbitrary sized array using Iterative method (Lecture 5)

```
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; }
```

Summing an arbitrary sized array using a Recursive method and Future Tasks (Lecture 5)

```
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> {return computeSum(X, lo, mid);};
    final future<int> sum2 =
      async<int> {return computeSum(X, mid+1, hi);};
    return sum1.get() + sum2.get();
  }
} // computeSum
int sum = computeSum(X, 0, X.length-1); // main program code
```

Can be replaced by finish-async, but future tasks are more natural

Formal Definition of Data Races (Lecture 6)

Formally, a data race occurs on location L in a program execution with computation graph CG if there exist steps S1 and S2 in computation graph 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 it is usually impossible to guarantee that all possible orderings of the accesses to a location will be encountered during program testing.

Thus, no amount of testing may be able to detect errors that might only become manifest in production use.

Example of Incorrect Parallelization from Homework 1 (Lecture 6)

```
1.  // Sequential version
2.  for ( p = first; p != null; p = p.next) p.x = p.y + p.z;
3.  for ( p = first; p != null; p = p.next) sum += p.x;
4.
5.  // Incorrect parallel version
6.  for ( p = first; p != null; p = p.next)
7.        async p.x = p.y + p.z;
8.  for ( p = first; p != null; p = p.next)
9.        sum += p.x;
```

Why was this version incorrect?

What does its computation graph say about writes to p.x in line 7 and reads of p.x in line 9?

Summary of forall statement (Lecture 7)

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

- forall statement creates multiple async child tasks, one per iteration of the forall
  - all child tasks can execute <body> in parallel
  - child tasks are distinguished by index "points" ([i1], [i1,i2], …)
- forall statement completes and parent task proceeds to the following statement when all child tasks have completed (implicit finish)
- <body> can read local variables from parent (copy-in semantics like async)

Amdahl's Law (Lecture 9)

- If $q \leq 1$ is the fraction of WORK in a parallel program that must be executed sequentially, then the best speedup that can be obtained for that program is Speedup $\leq 1/q$.
- Observation follows directly from critical path length lower bound on parallel execution time, $t_P \geq CPL(G)$
- If fraction q of WORK is sequential then $CPL(G) \geq q\text{WORK}$
- Therefore, Speedup = t_1/t_P must be $\leq \text{WORK}/(q\text{WORK}) = 1/q$

- Sequential portion of WORK = q (also denoted as f_S sometimes)
- Parallel portion of WORK = 1-q (also denoted as f_p sometimes)

HJ isolated statement (Lecture 10)

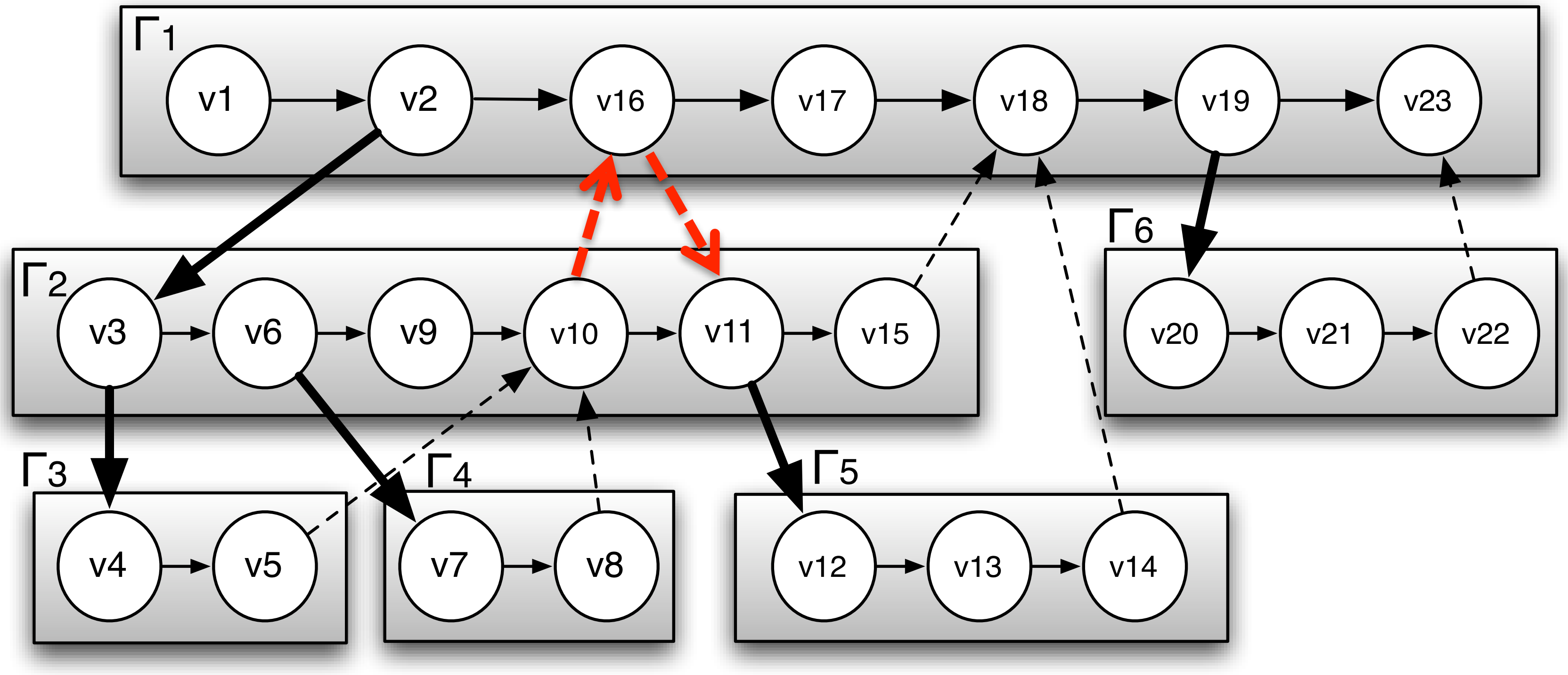
isolated <body>

- Two tasks executing isolated statements with interfering accesses must perform the isolated statement in mutual exclusion
  - — Two instances of isolated statements, ⟨stmt1⟩ and ⟨stmt2⟩, are said to interfere with each other if both access a shared location, such that at least one of the accesses is a write.
  - ➔ Weak isolation guarantee: no mutual exclusion applies to non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances
- Isolated statements may be nested (redundant)
- Isolated statements must not contain any other parallel statement: async, finish, get, forall
- In case of exception, all updates performed by <body> before throwing the exception will be observable after exiting <body>

Serialized Computation Graph for Isolated Statements (Lecture 10)

- Model each instance of an isolated statement as a distinct step (node) in the CG.
- Need to reason about the order in which interfering isolated statements are executed
  - complicated because the order may vary from execution to execution
- Introduce Serialized Computation Graph (SCG) that includes a specific ordering of all interfering isolated statements.
  - SCG consists of a CG with additional serialization edges.
  - Each time an isolated step, S', is executed, we add a serialization edge from S to S' for each isolated step, S, that has already executed such that S and S' have interfering accesses.
  - An SCG represents a set of executions in which all interfering isolated statements execute in the same order.

Example of Serialized Computation Graph with Serialization Edges (Lecture 10)



→ Continue edge ➞ Spawn edge ⇢ Join edge

⇢ **Serialization edge**

v10: isolated { x ++; y = 10; }
v11: isolated { x++; y = 11; }
v16: isolated { x++; y = 16; }

Barrier Synchronization: HJ’s “next” statement in forall construct (Lecture 12)

```
rank.count = 0; // rank object contains an int field, count
forall (point[i] : [0:m-1]) {
  int r;
  isolated {r = rank.count++;}
  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

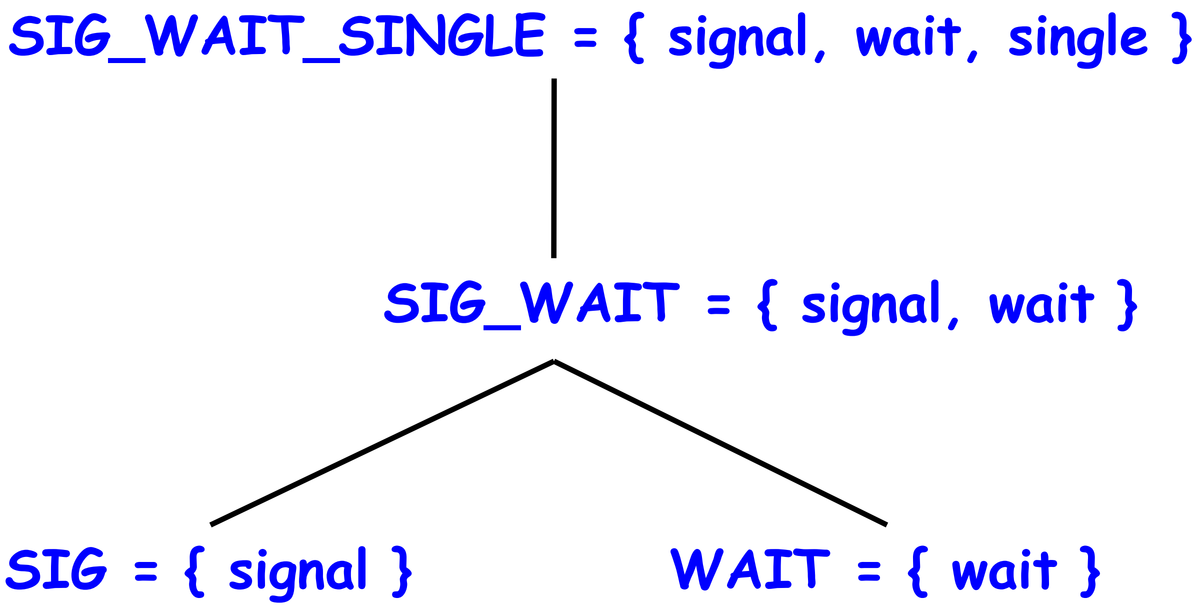
Phase 1

- **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 (will be covered in following lectures)

Summary of Phaser Construct (Lecture 15)

- 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*
- 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

Capability Hierarchy



SIG_WAIT_SINGLE = { signal, wait, single }

SIG_WAIT = { signal, wait }

SIG = { signal }

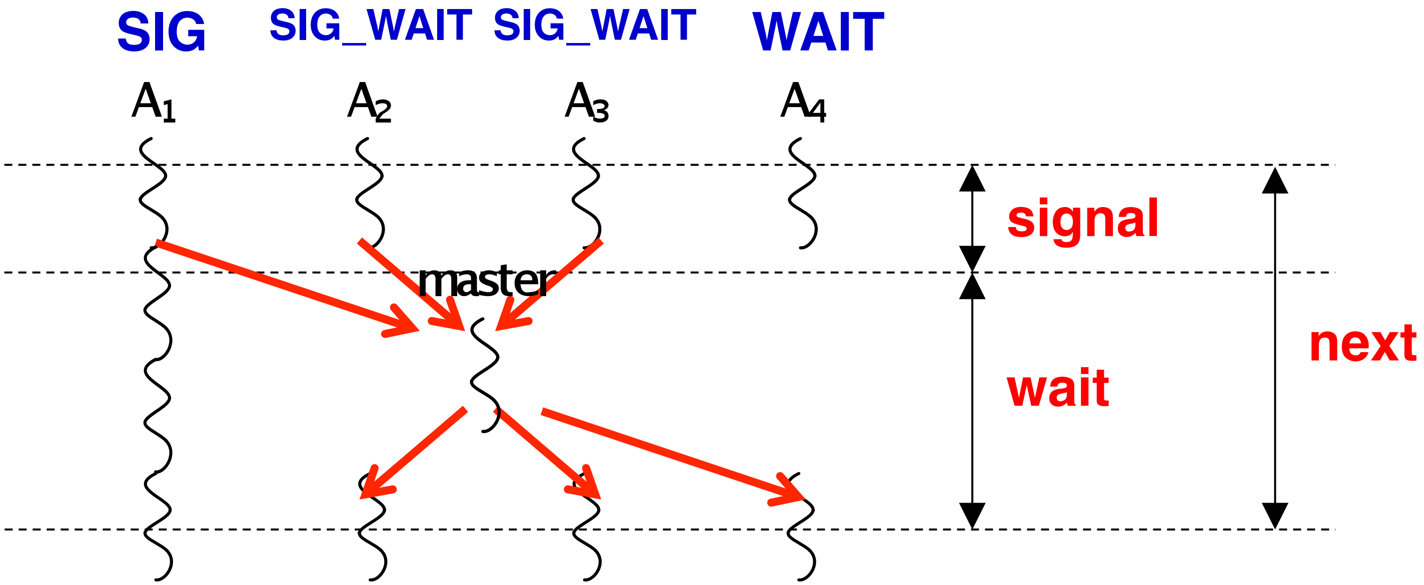
WAIT = { wait }

- At any point in time, a task can be registered in one of four modes with respect to a phaser: SIG_WAIT_SINGLE, SIG_WAIT, SIG, or WAIT. The mode defines the set of capabilities — signal, wait, single — that the task has with respect to the phaser. The subset relationship defines a natural hierarchy of the registration modes.

next operation
(Lecture 15)

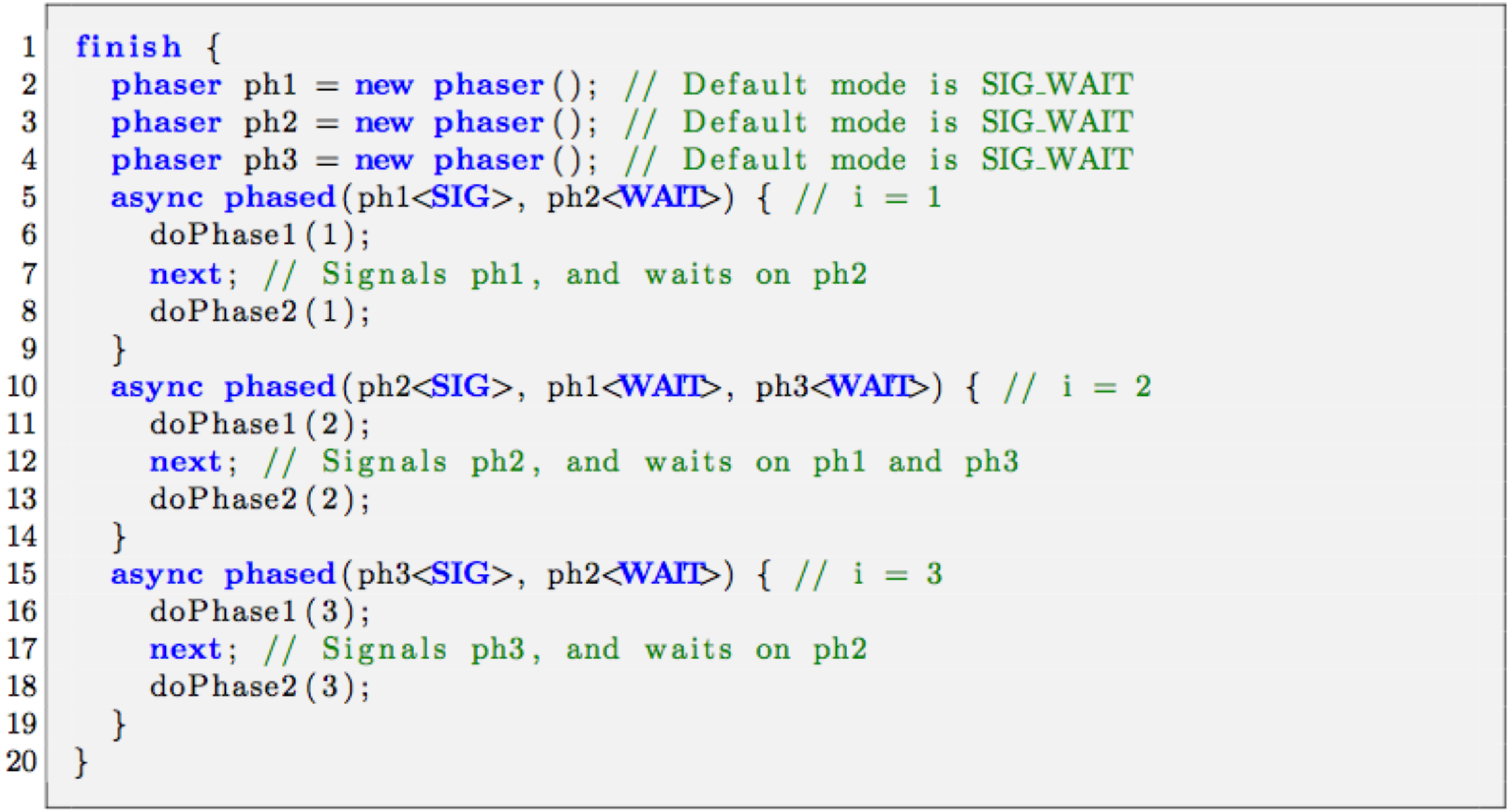
Semantics of next depends on registration mode

- **SIG_WAIT: next = signal + wait**
- **SIG: next = signal**
- **WAIT: next = wait**



A master task receives all signals and broadcasts a barrier completion

Left-Right Neighbor Synchronization Example for m=3 using Phasers (Lecture 15)



```
finish {
  phaser ph1 = new phaser(); // Default mode is SIG_WAIT
  phaser ph2 = new phaser(); // Default mode is SIG_WAIT
  phaser ph3 = new phaser(); // Default mode is SIG_WAIT
  async phased(ph1<SIG>, ph2<WAIT>) { // i = 1
    doPhase1(1);
    next; // Signals ph1, and waits on ph2
    doPhase2(1);
  }
  async phased(ph2<SIG>, ph1<WAIT>, ph3<WAIT>) { // i = 2
    doPhase1(2);
    next; // Signals ph2, and waits on ph1 and ph3
    doPhase2(2);
  }
  async phased(ph3<SIG>, ph2<WAIT>) { // i = 3
    doPhase1(3);
    next; // Signals ph3, and waits on ph2
    doPhase2(3);
  }
}
```

Listing 3: Extension of example in Listing 1 with three phasers for $m = 3$