COMP 322: Fundamentals of Parallel Programming

Lecture 2: Async-Finish Parallel Programming and Computation Graphs

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

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

Outline of Today's Lecture

- Async-Finish Parallel Programming (contd)
- Computation Graphs

- Acknowledgments
  - Cilk lectures, http://supertech.csail.mit.edu/cilk/
  - COMP 322 Module 1 handout, Sections 1.1, 1.2, 2.1, 2.2
    - https://svn.rice.edu/r/comp322/course/module1-2013-01-06.pdf

Async and Finish Statements for Task Creation and Termination

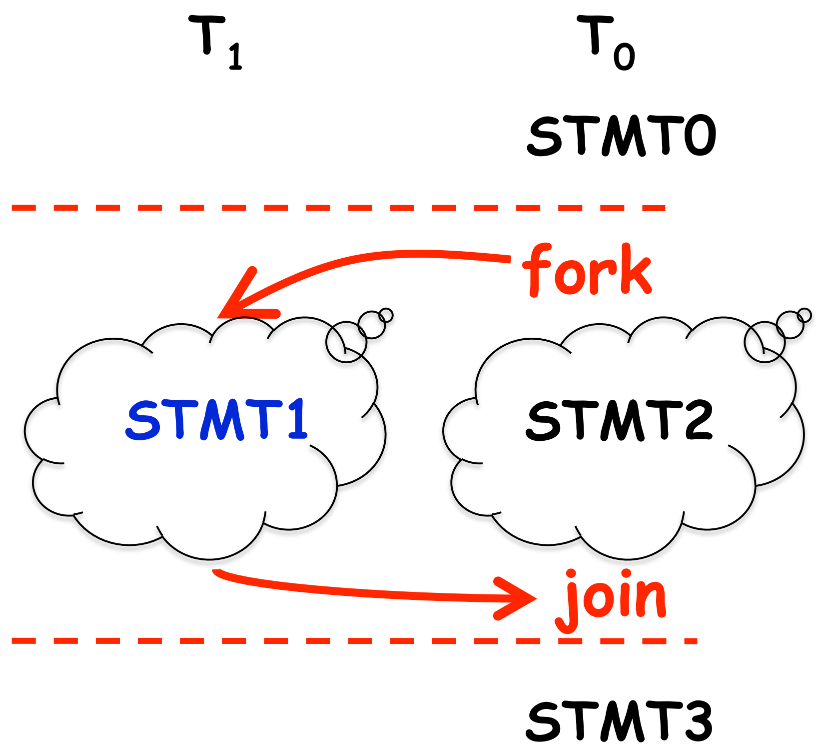
async S

- 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
```



Some Properties of Async & Finish constructs

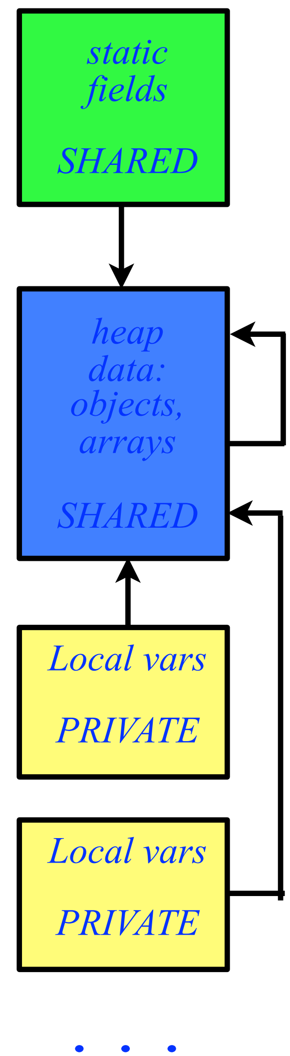
1. Scope of async/finish can be any arbitrary statement
   – async/finish constructs can be arbitrarily nested e.g.,
   – `finish { async S1; finish { async S2; S3; } S4; } S5;`
2. A method may return before all its async's have terminated
   – Enclose method body in a finish if you don't want this to happen
   – main() method is enclosed in an implicit finish e.g.,
   – `main(){ foo();}  void foo() {async S1; S2; return;}`
3. Each dynamic async task will have a unique Immediately Enclosing Finish (IEF) at runtime
4. Async/finish constructs cannot "deadlock"
   – Cannot have a situation where both task A waits for task B to finish, and task B waits for task A to finish
5. Async tasks can read/write shared data via objects and arrays
   – Local variables have special restrictions

Shared and Private data in Java's Storage Model

Java's storage model contains three memory regions:

1. Static Data: region of memory reserved for variables that are not allocated or destroyed during a class' lifetime, such as static fields.
   - Static fields can be shared among threads/tasks
2. Heap Data: region of memory for dynamically allocated objects and arrays (created by "new").
   - Heap data can be shared among threads/tasks
3. Stack Data: Each time you call a method, Java allocates a new block of memory called a stack frame to hold its local variables
   - Local variables are private to a given thread/task

All references (pointers) must point to heap data --- no references can point to static or stack data



Local Variables

Three rules for accessing local variables across tasks in HJ:

1) *An async may read the value of any final outer local var*

```
final int i1 = 1; async { ... = i1; /* i1=1 */ }
```

2) *An async may read the value of any non-final outer local var (copied on entry to async like method parameters)*

```
int i2 = 2; // i2=2 is copied on entry to the async
async { ... = i2; /* i2=2*/}
i2 = 3; // This assignment is not seen by the above async
```

3) *An async is not permitted to modify an outer local var*

```
int[] A; async { A = ...; /*ERROR*/  A[i] = ...; /*OK*/ }
```

Converting sequential Java programs to parallel Async-Finish HJ programs

One possible approach:

1. Create "ideal" parallel version
   - Insert async's at all points where parallelism can logically be exploited
   - Insert finish's to ensure that the parallel version produces the same results as the sequential version

2. Transform ideal parallelism to useful parallelism
   - Merge or remove async's to amortize overhead
   - Replace finish by more efficient synchronization constructs
   - (to be covered later in course)

Example usages of async for ideal parallelism (Listing 1, Module 1, page 9)

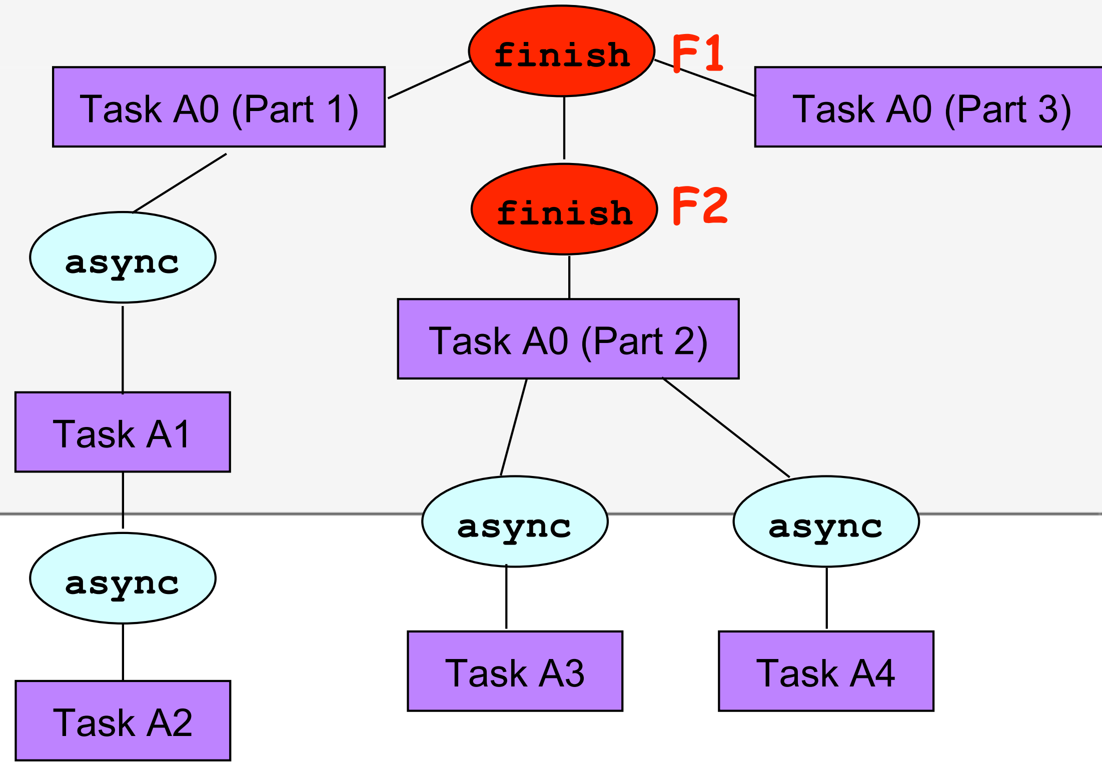
```
// Example 1: execute iterations of a counted for loop in parallel
// (we will later see forall loops as a shorthand for this common case)
for (int i = 0; i < A.length; i++)
  async { A[i] = B[i] + C[i]; } // value of i is copied on entry to async

// Example 2: execute iterations of a while loop in parallel
p = first;
while ( p != null ) {
  async { p.x = p.y + p.z; } // value of p is copied on entry to async
  p = p.next;
}

// Example 3: Example 2 rewritten as a recursive method
static void process(T p) {
  if ( p != null ) {
    async { p.x = p.y + p.z; } // value of p is copied on entry to async
    process(p.next);
  }
}

// Example 4: execute method calls in parallel
async left_s = quickSort(left);
async right_s = quickSort(right);
```

Insertion of finish for correct ideal parallelism (Listing 5, Module 1, page 12)

```
// Example 1: Sequential version
for (int i = 0; i < A.length; i++) A[i] = B[i] + C[i];
System.out.println(A[0]);

// Example 1: Incorrect parallel version
for (int i = 0; i < A.length; i++) async A[i] = B[i] + C[i];
System.out.println(A[0]);

// Example 1: Correct parallel version
finish for (int i = 0; i < A.length; i++) async A[i] = B[i] + C[i];
System.out.println(A[0]);

// Example 2: Sequential version
p = first;
while ( p != null ) {
  p.x = p.y + p.z; p = p.next;
}
System.out.println(first.x);

// Example 2: Incorrect parallel version
p = first;
while ( p != null ) {
  async { p.x = p.y + p.z; }
  p = p.next;
}
System.out.println(first.x);

// Example 2: Correct parallel version
p = first;
finish while ( p != null ) {
  async { p.x = p.y + p.z; }
  p = p.next;
}
System.out.println(first.x);
```

Dynamic Finish-Async nesting structure and Immediately Enclosing Finish (IEF)

```
finish { // F1
  // Part 1 of Task A0
  async {A1; async A2;}
  finish { // F2
    // Part 2 of Task A0
    async A3;
    async A4;
  }
  // Part 3 of Task A0
}
```



- IEF(A3) = IEF(A4) = F2
- IEF(A1) = IEF(A2) = F1
- Module 1: Listing 6 & Figure 7

How can an Async Task interact with its Parent Task?

- **Data flow**
  - Async task can read from static fields, objects, arrays, and local variables written by parent task
    - Same rule as method calls, except that parent's local variables are passed as <u>implicit</u> parameters
  - Async task can write to static fields, objects, arrays (but not parent's local variables) to be read by parent task after end-finish
    - Same rule as method calls, except that method calls also have return values
    - We will learn soon about an extension to asyncs with return values (futures)
- **Control flow**
  - Async task can execute a return statement (different from method return)
  - Async task can throw an exception
  - NOTE: break/continue cannot cross async boundaries

Data Flow: Use of Static Fields to Communicate Return Value from an Async Task

```
static int sum1 = 0, sum2 = 0;
public static void main(String[] argv) { // caller
  int[] X = new int[...];
  ... // Initialize X
  int sum;
  finish { // Async's have same access rules as methods
    async for(int i=X.length/2; i < X.length; i++)
            sum2 += X[i];
    async for(int i=0; i < X.length/2; i++)
            sum1 += X[i];
   }
   sum = sum1 + sum2;
   ....
 }
```

Data Flow: Use of an Object to Communicate Return Values from Async Tasks (Better Approach)

```
public class TwoIntegers {int sum1; int sum2;}
. . .
 public static void main(String[] argv) { // caller
 int[] X = new int[...]; ... // Initialize X
 int sum;
 TwoIntegers r = new TwoIntegers();
 finish { // Async's have same access rules as methods
   async for(int i=X.length/2; i < X.length; i++)
           r.sum2 += X[i];
   async for(int i=0; i < X.length/2; i++)
           r.sum1 += X[i];
  }
  sum = r.sum1 + r.sum2;
  ....
}
```

Control Flow: Semantics of HJ return statement

- Java semantics for return
  - Return from enclosing method
- HJ semantics for return statement
  - Return from immediately enclosing async or method

```
void foo() {
  if (...) return; // Returns from method foo()
  async { ... return; ... } // Returns from async
  . . .
}
```

Control Flow: Semantics of HJ break and continue statements

- Java semantics for break/continue
  - Perform appropriate action for innermost enclosing loop (or labeled loop)
  - It's an error to execute a break/continue statement without an enclosing loop
- HJ semantics for break/continue
  - It's also an error to execute a break/continue statement in an async without an enclosing loop in the same async
  - Results in cryptic error messages from HJ compiler
    - "Target of branch statement not found"
    - "Unreachable statement"

```
1. void foo() {
2.   while (...) {
3.     async {
4.       while (...) { ... break; ... } // Okay
5.       break; // Error --- does not relate to while loop in line 2
6.     } } }
```

Examples of Common Errors made by beginner HJ Programmers

```
finish for (int i = 0; i <= N – M
  int j;
  async {
    for (j = 0; j < M; j++) {
      async {
        if (text[i+j] != pattern[j]) break;
      }
      if (j == M) return i;// found at off
    }
}
```

Async cannot modify local variable in parent's scope

No loop enclosing break in async

Return statement in basic async task cannot take a value

Habanero-Java (HJ) Compilation and Execution Environment

DrHJ IDE (optional)

Foo.hj

HJ source program --- must contain a class named Foo with a public static void main(String[] args) method

hjc Foo.hj

HJ compiler

HJ compiler translates Foo.hj to Foo.class, and inserts calls to HJ runtime as needed

Foo.class

hj –places m:n Foo

HJ Runtime Environment =
JRE + HJ libraries +
HJ Multithreaded Runtime

HJ runtime allocates m*n worker threads across m "places" (default values: m = 1 place, n = # hardware cores/threads)

HJ Program Output

Data Race Detection Output, HJ Computation Graph, HJ Abstract Performance Metrics (all enabled by appropriate options)

Outline of Today's Lecture

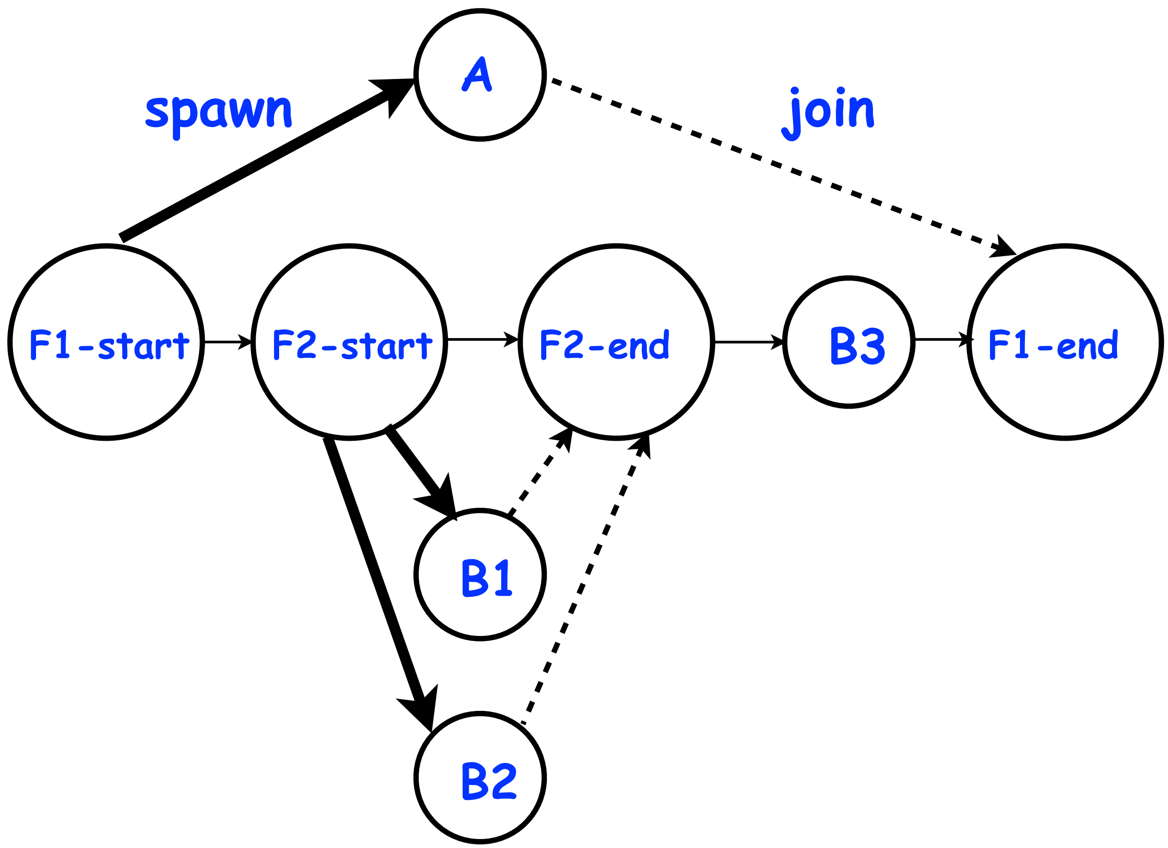
- Async-Finish Parallel Programming (contd)
- Computation Graphs

- Acknowledgments
  - Cilk lectures, http://supertech.csail.mit.edu/cilk/
  - COMP 322 Module 1 handout, Sections 1.3, 2.3, 2.4, 3.1
    - https://svn.rice.edu/r/comp322/course/module1-2013-01-06.pdf

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

```
finish { // F1
  async A;
  finish { // F2
    async B1;
    async B2;
  } // F2
  B3;
} // F1
```

Computation Graph



Computation Graphs for HJ Programs

- 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
- 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)

Complexity Measures for Computation Graphs

Define

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

What is the critical path length of this parallel computation?

```
finish { // F1
  async A; // Boil pasta
  finish { // F2
    async B1; // Chop veggies
    async B2; // Brown meat
  } // F2
  B3; // Make pasta sauce
} // F1
```

Step B1

Step B2

Step B3

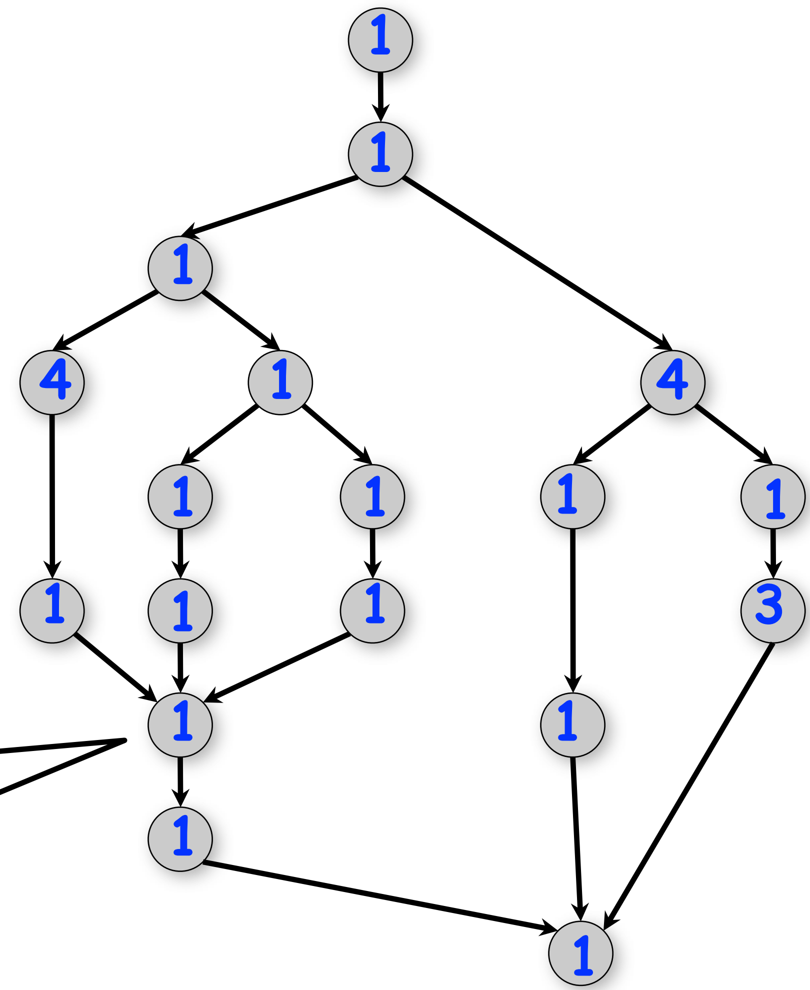
Step A

Ideal Parallelism

Define **ideal parallelism** of Computation G Graph as the ratio, WORK(G)/CPL(G)

Ideal Parallelism is independent of the number of processors that the program executes on, and only depends on the computation graph

What is the ideal parallelism of this graph?
Time for worksheet #2!

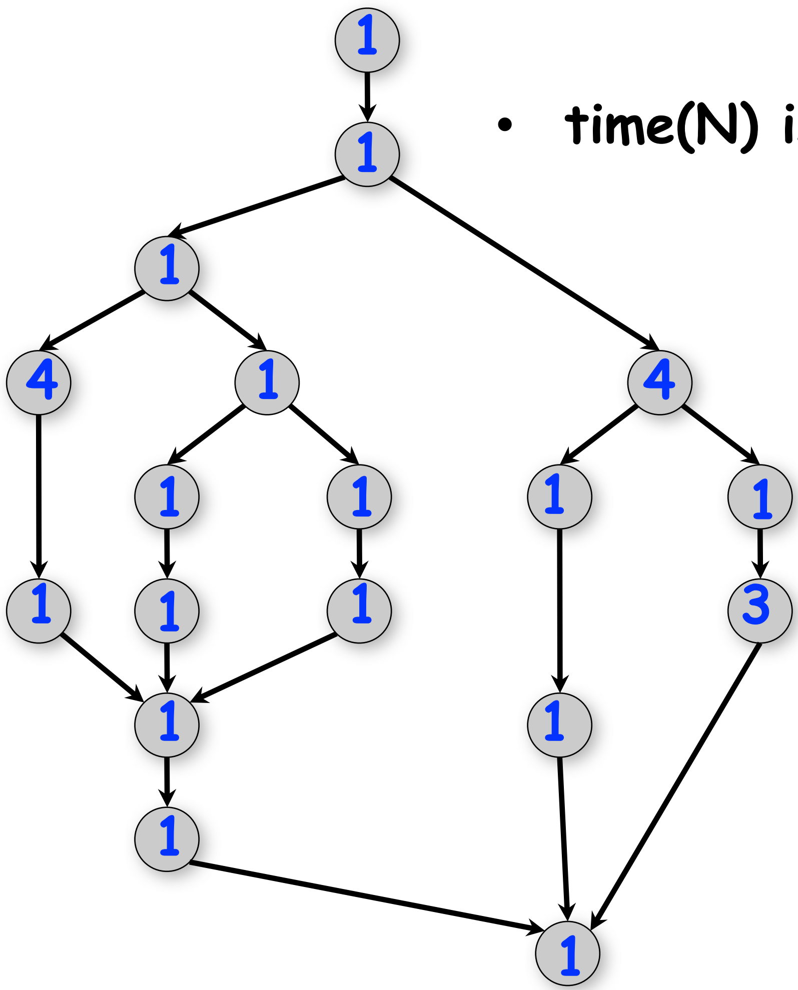


Course Announcements

- Homework 1 has been posted
  - Contains written and programming components
  - Due by 5pm on Wednesday, Jan 23rd
  - Must be submitted using "turnin" script introduced in Lab 1
    - In case of problems, email a zip file to comp322-staff at mailman.rice.edu before the deadline
  - See course web site for penalties for late submissions
- Instructor's office hours are during 2pm - 3pm on MWF
  - Please stop by if you have problems with any of the following
    - Accessing the Module 1 handout
    - Using the turnin script
    - You did not receive the welcome email sent to comp322-all on Sunday night

Worksheet #2: what is the critical path length and ideal parallelism of this graph?

Name 1: ______________________ Name 2: ______________________



- time(N) is labeled for all nodes N in the graph

WORK(G) = 26

CPL(G) =

Ideal Parallelism
= WORK(G)/CPL(G)
=