CS 314 Principles of Programming Languages

Lecture 18: Parallelism and Dependence Analysis

Prof. Zheng Zhang

Rutgers University

November 9, 2018

Class Information

- Project 2 is released.
- Homework 7 will be released this weekend.

Programming with Concurrency

- A PROCESS or THREAD is a potentially-active execution context
- Classic *von Neumann* model of computing has single thread of control, however parallel programs have more than one
- A process or thread can be thought of as

  An abstraction of a physical PROCESSOR
- Processes/Threads can come from
  - Multiple CPUs
  - Kernel-level multiplexing of single physical machine
  - Language or library level multiplexing of kernel-level abstraction
- They can run
  - In true **parallel**
  - Unpredictably interleaved
  - Run-until-block

Dependence and Parallelization

Dependence analysis is fundamental to parallelization analysis

Dependence relation: all *task–to–task* execution orderings that must be preserved if the meaning of the program is to remain the same.

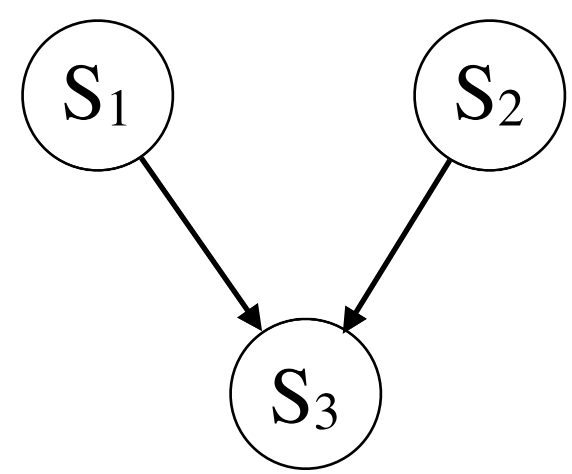
The dependence relation can be modeled as a directed graph such that if $A \rightarrow B$, the result of task A is required for the processing of task B

Example:

S_1: pi = 3.14

S_2: R = 5

S_3: Area = pi * R^2

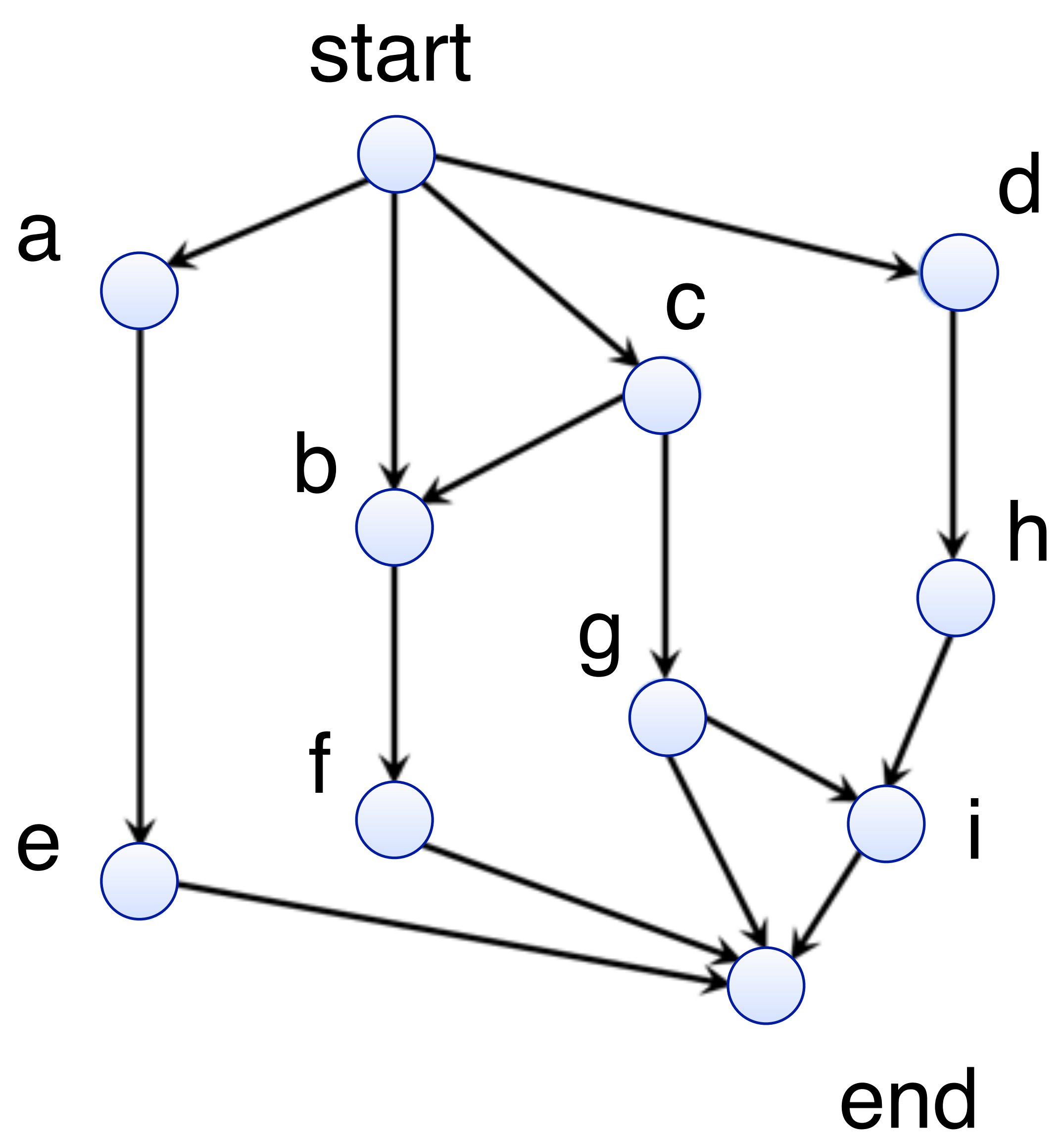


Statement-level dependence graph

Dependence Graph

- Directed acyclic graph (DAG)
- A node represents a task
- A directed edge represents precedence constraint

DAG example 1:

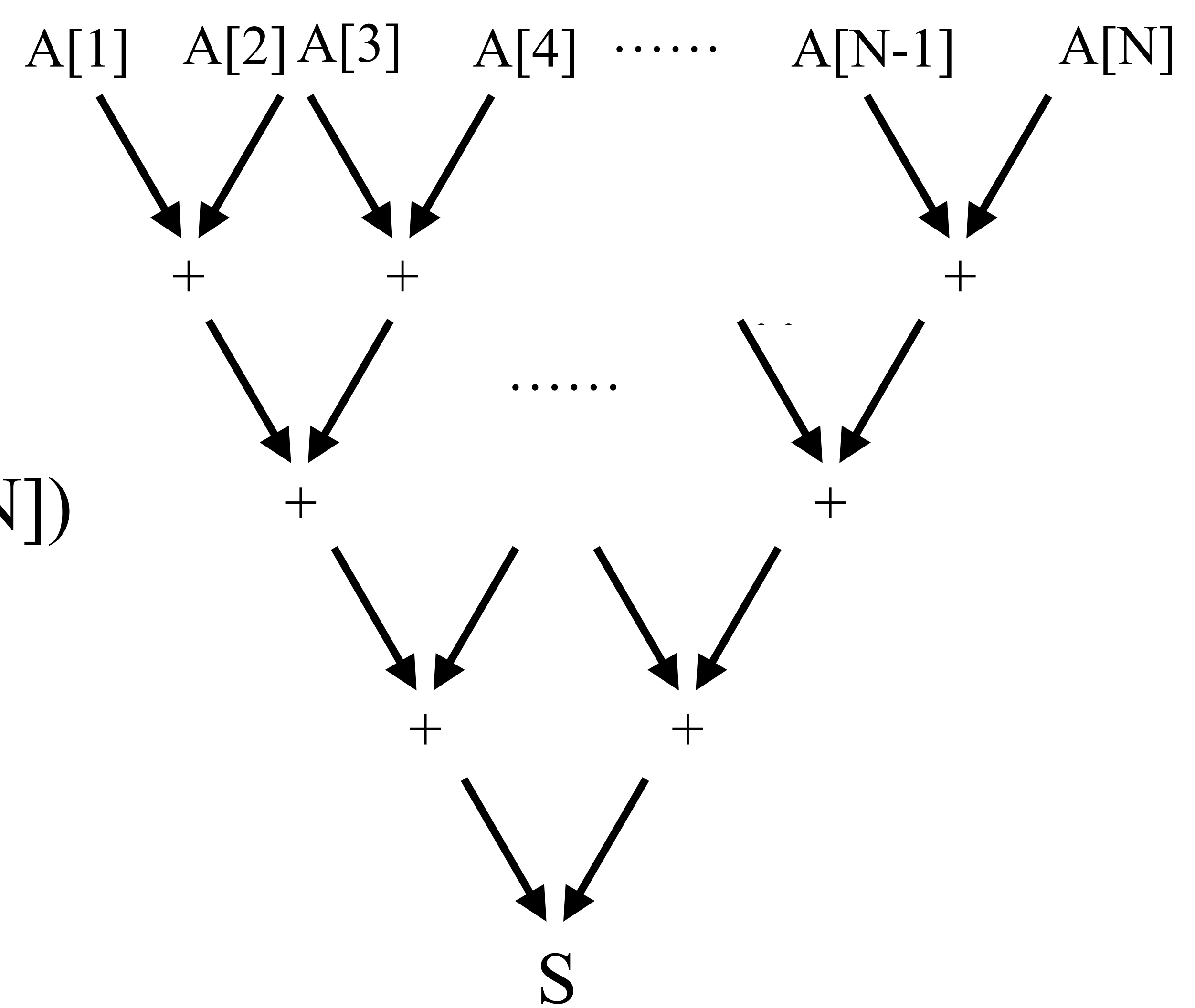


Dependence Graph

- Directed acyclic graph (DAG)
- A node represents a task
- A directed edge represents precedence constraint

DAG example 2:

S = sum(A[1], A[2], …, A[N])



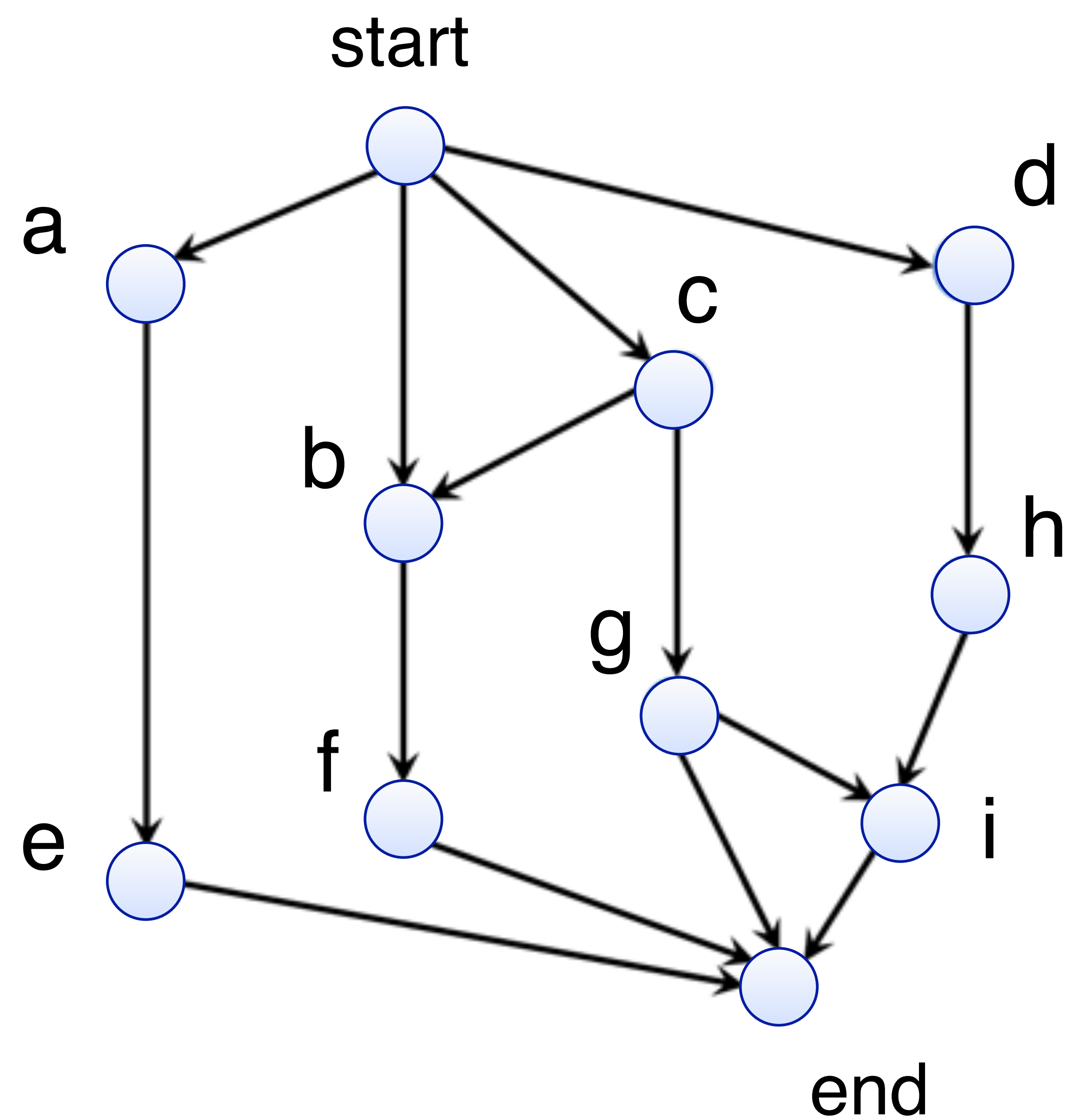
Scheduling a DAG

T_p: time to perform computation with *p* processors

- T_1 : **work** (total # operations)
- T_∞ : **critical path** or **span**

$$T_p \geq \frac{T_1}{p}, \qquad T_p \geq T_\infty$$

Assuming a task takes 1 unit time



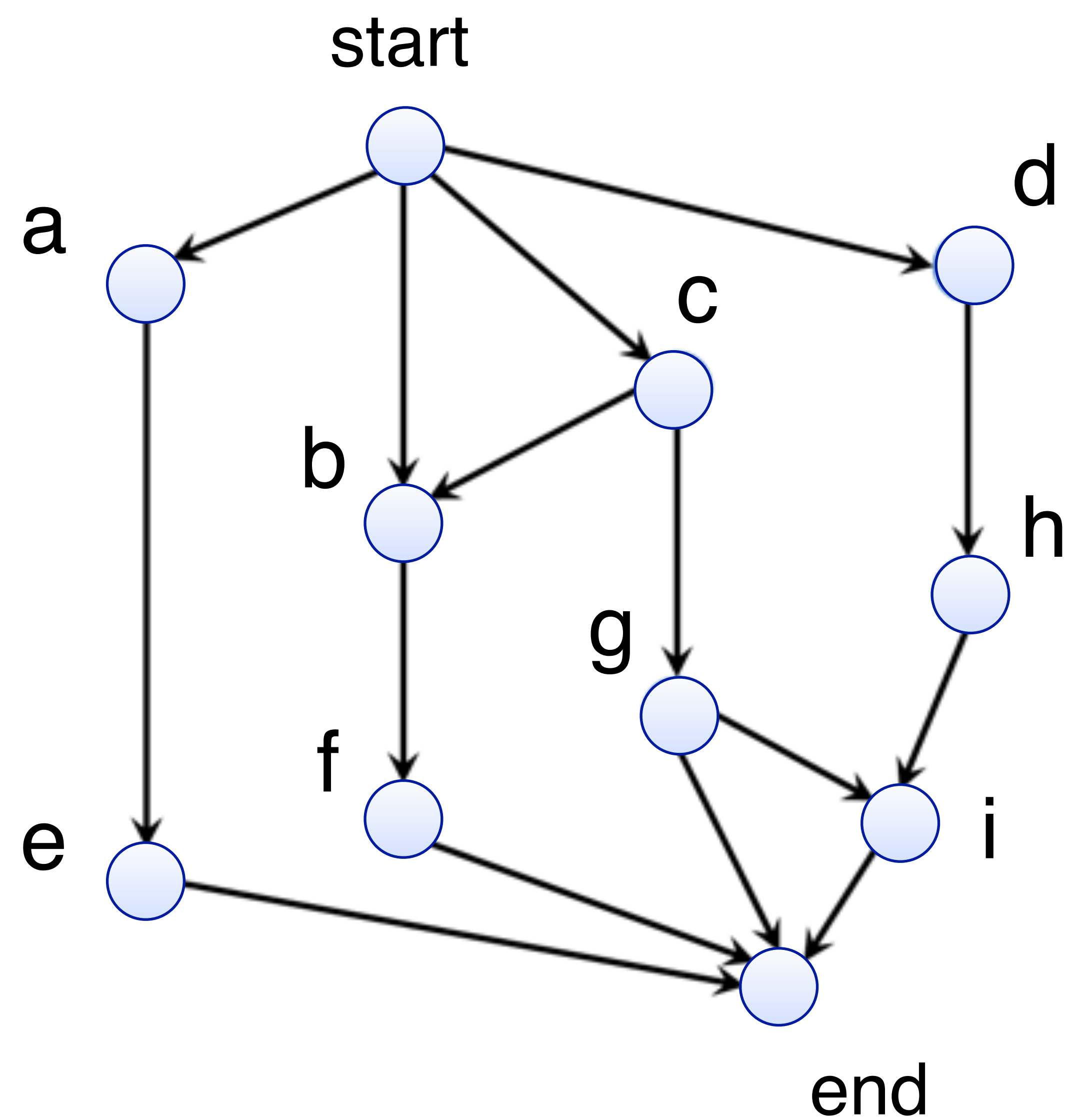
Scheduling a DAG

T_p: time to perform computation with *p* processors

- T_1 : **work** (total # operations)
- T_∞ : **critical path** or **span**

$$T_p \geq \frac{T_1}{p}, \qquad T_p \geq T_\infty$$

Assuming a task takes 1 unit time



$T_1 = ?$

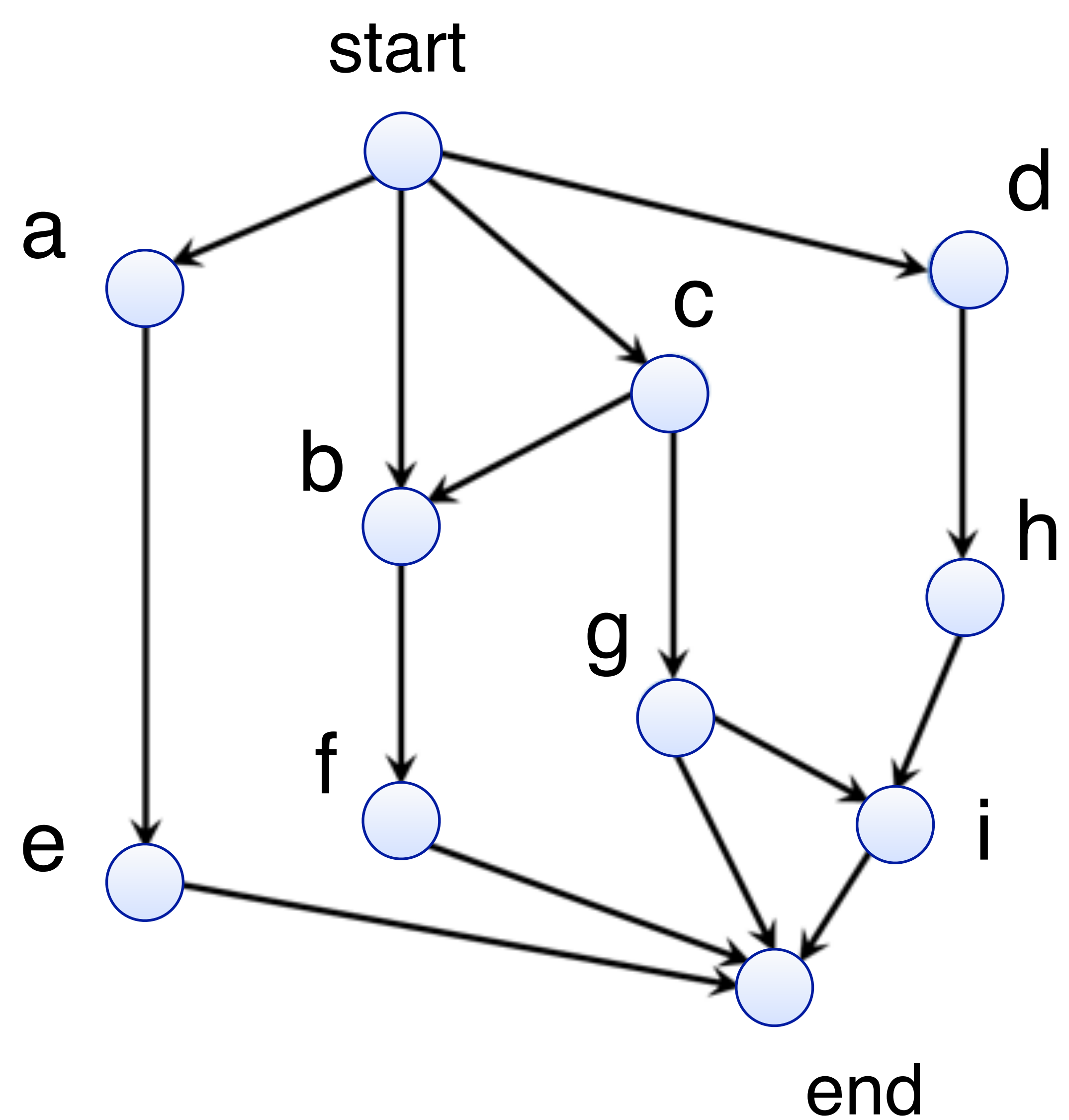
Scheduling a DAG

T_p: time to perform computation with *p* processors

- T_1 : **work** (total # operations)
- T_∞ : **critical path** or **span**

$$T_p \geq \frac{T_1}{p}, \qquad T_p \geq T_\infty$$

Assuming a task takes 1 unit time



$T_\infty = ?$

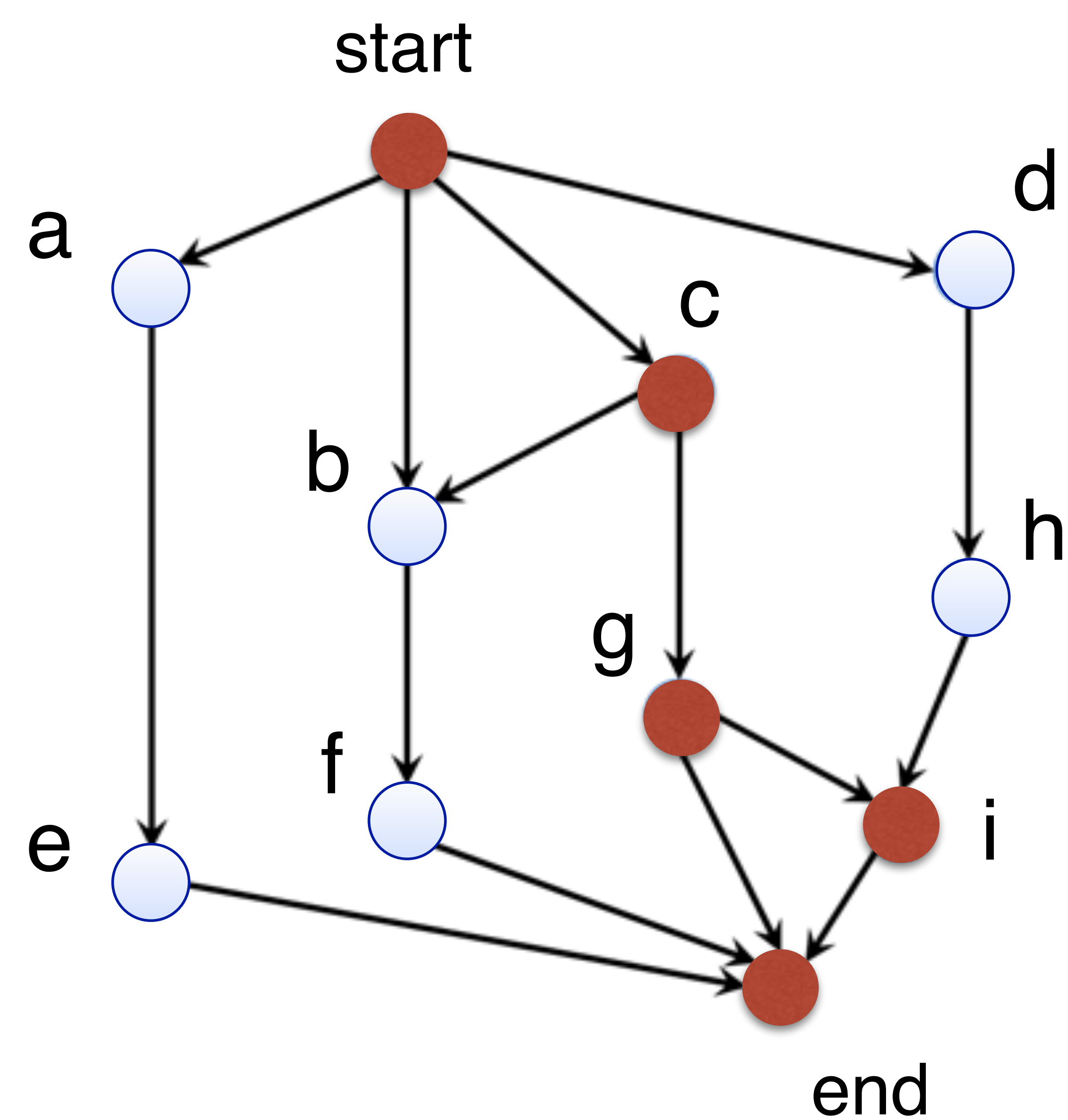
Scheduling a DAG

T_p: time to perform computation with *p* processors

- T_1 : **work** (total # operations)
- T_∞ : **critical path** or **span**

$$T_p \geq \frac{T_1}{p}, \qquad T_p \geq T_\infty$$

Assuming a task takes 1 unit time



$T_\infty = ?$

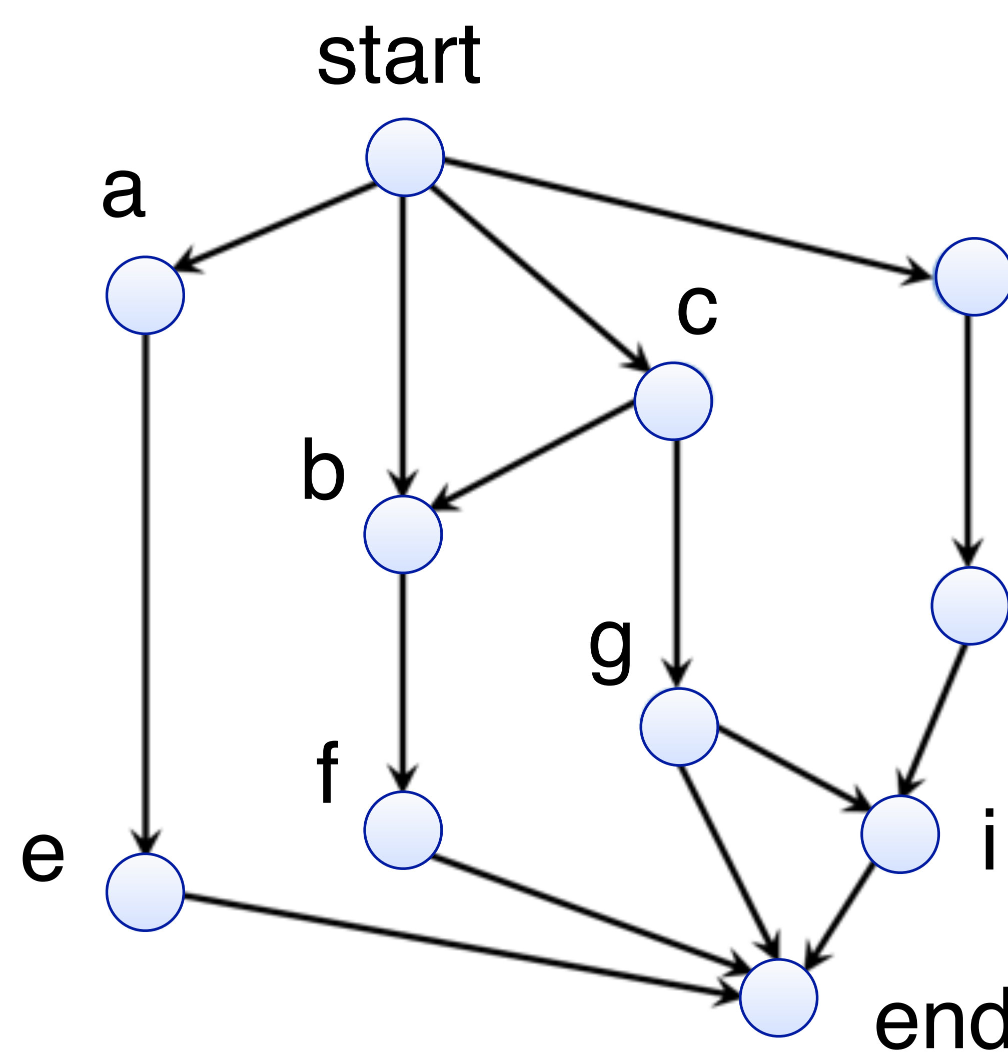
Computing Critical Path

Compute the earliest start time of each node

- Keep a value called S(n) associated with each node n
- For each node n

  S(n) is the maximum of { S(p) + 1 }, for all p ∈ pred(n)

Assuming a task takes 1 unit time



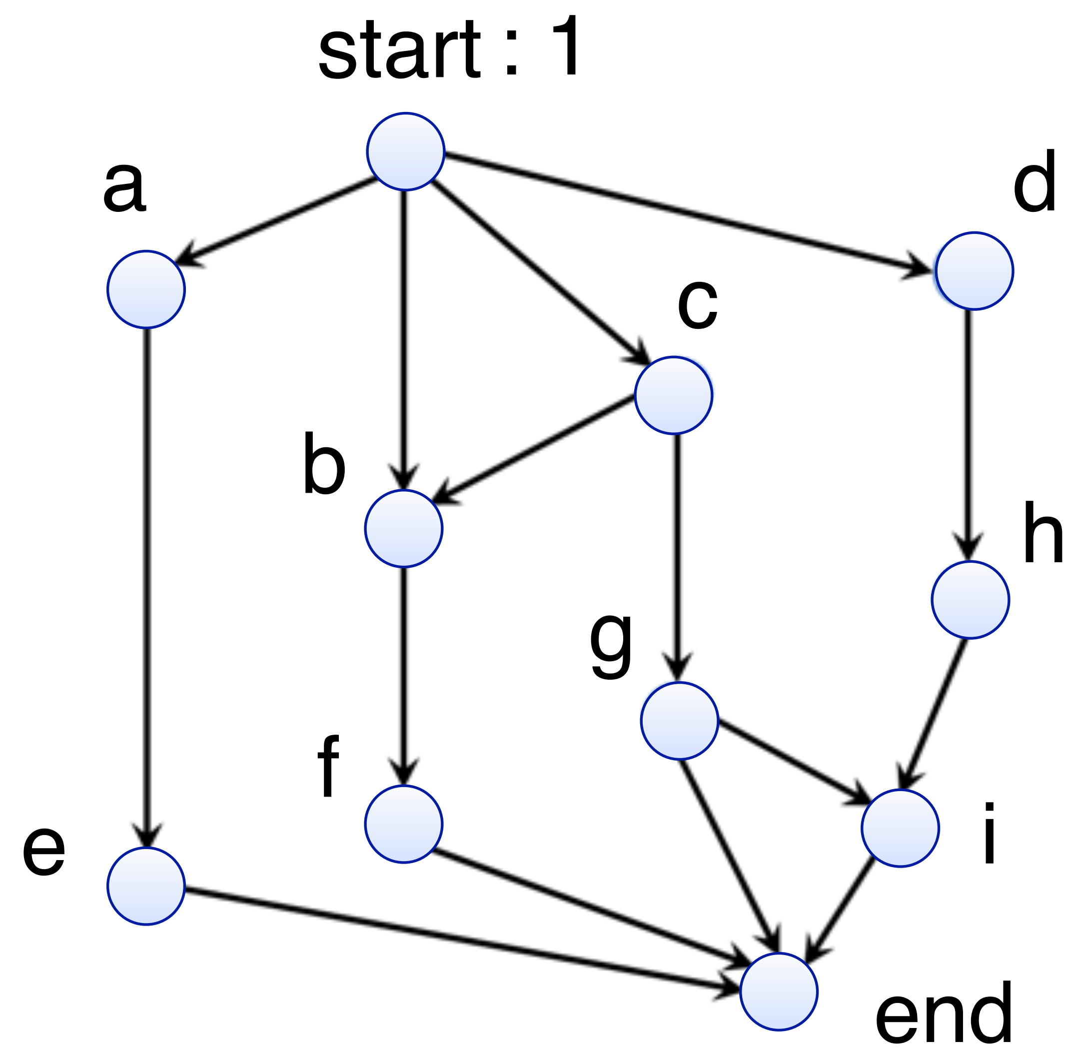
Computing Critical Path

Compute the earliest start time of each node

- Keep a value called S(n) associated with each node n
- For each node n

  S(n) is the maximum of { S(p) + 1 }, for all $p \in$ pred(n)

Assuming a task takes 1 unit time



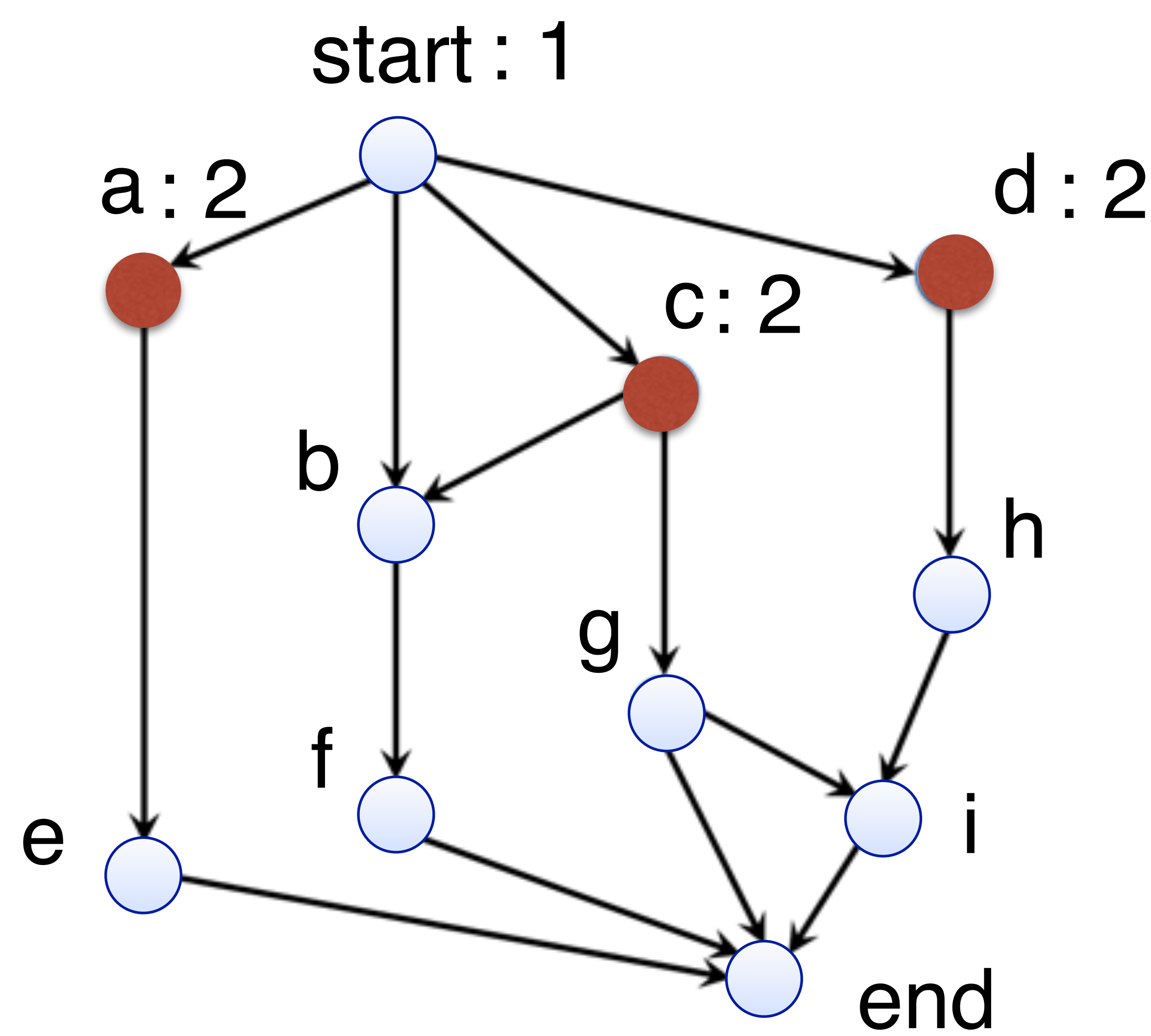
Computing Critical Path

Compute the earliest start time of each node

- Keep a value called S(n) associated with each node n
- For each node n

  S(n) is the maximum of { S(p) + 1 }, for all p $\in$ pred(n)

Assuming a task takes 1 unit time



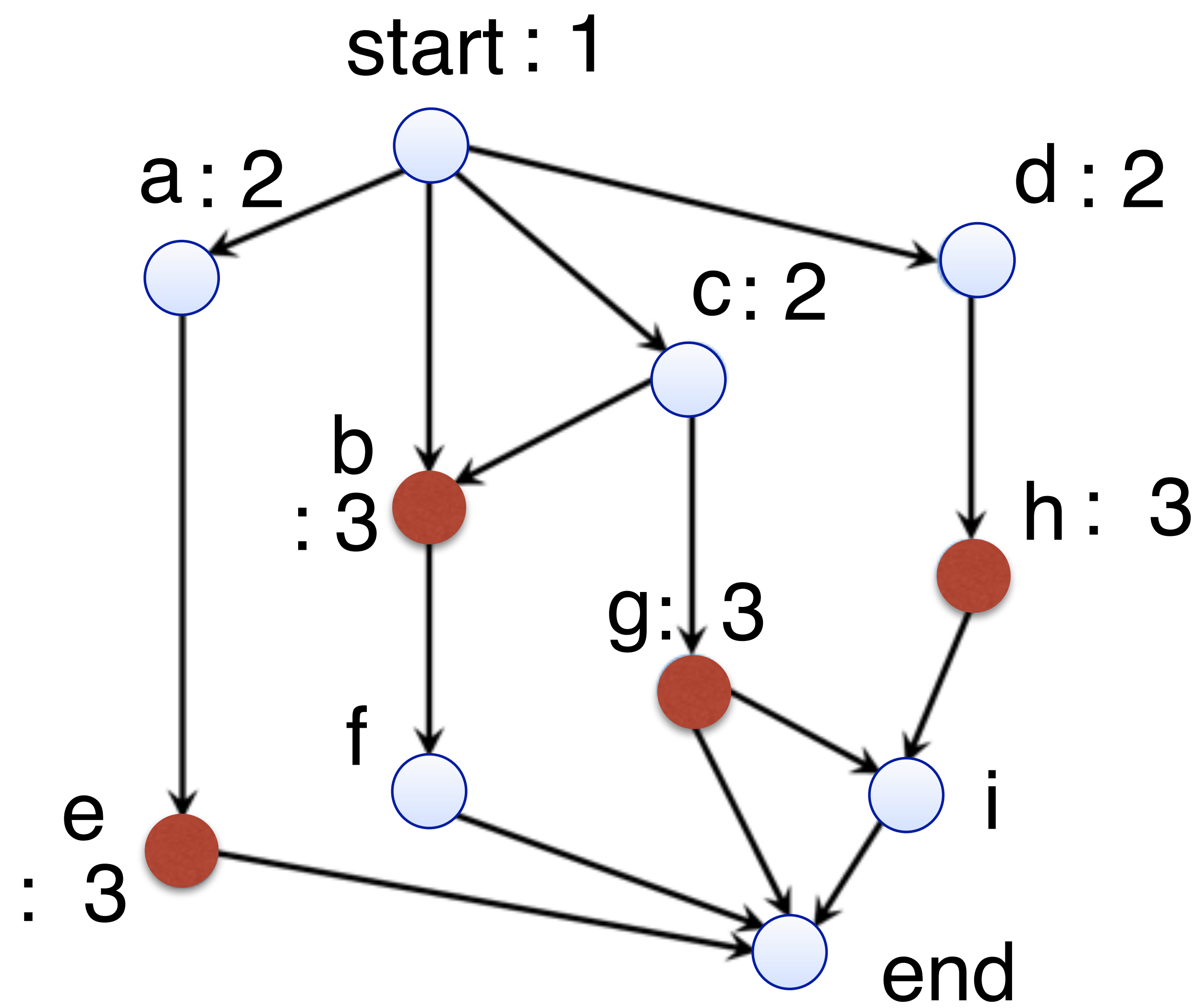
Computing Critical Path

Compute the earliest start time of each node

- Keep a value called S(n) associated with each node n
- For each node n

  S(n) is the maximum of { S(p) + 1 }, for all p $\in$ pred(n)

Assuming a task takes 1 unit time



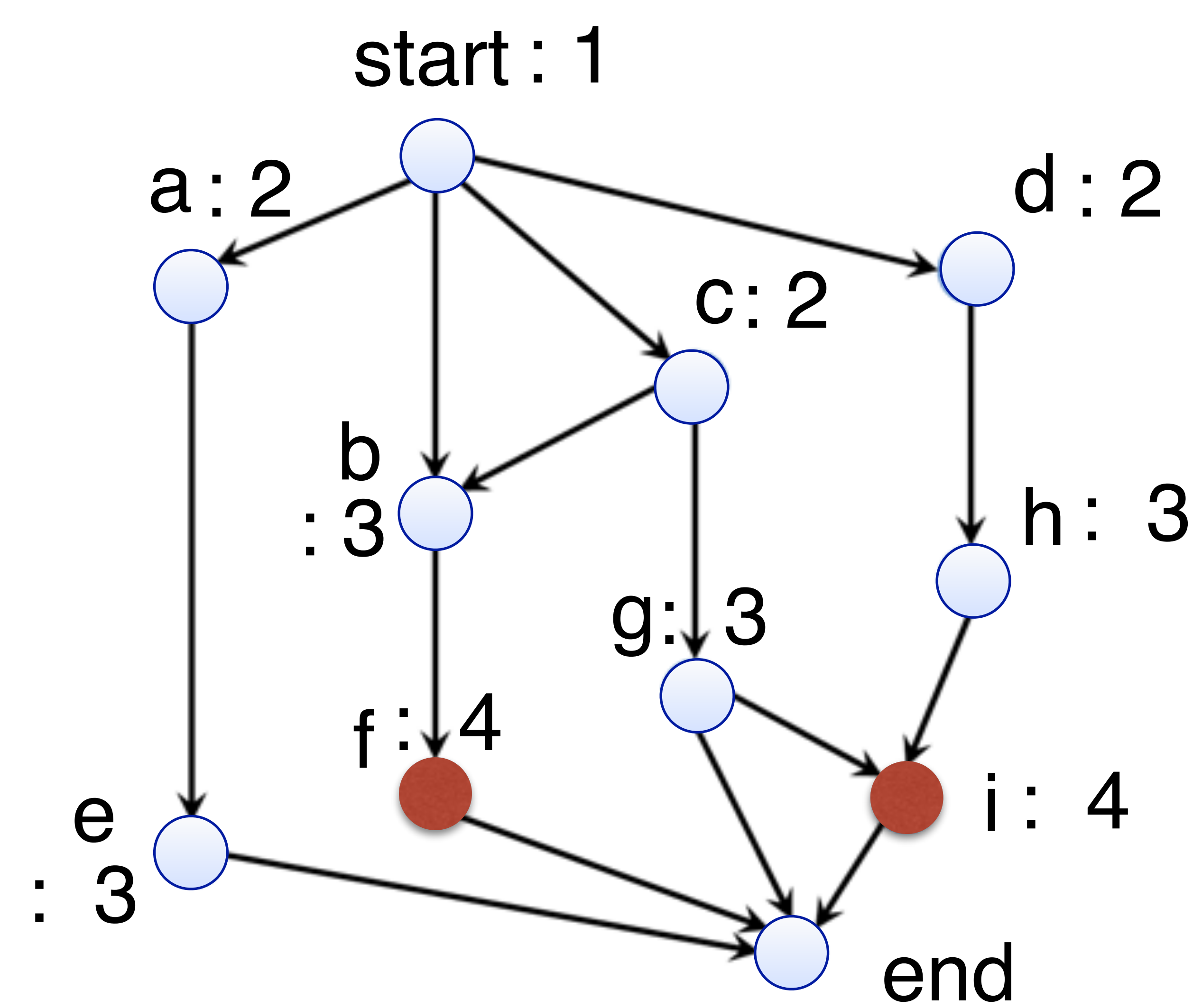
Computing Critical Path

Compute the earliest start time of each node

- Keep a value called S(n) associated with each node n
- For each node n

  S(n) is the maximum of { S(p) + 1 }, for all p ∈ pred(n)

Assuming a task takes 1 unit time



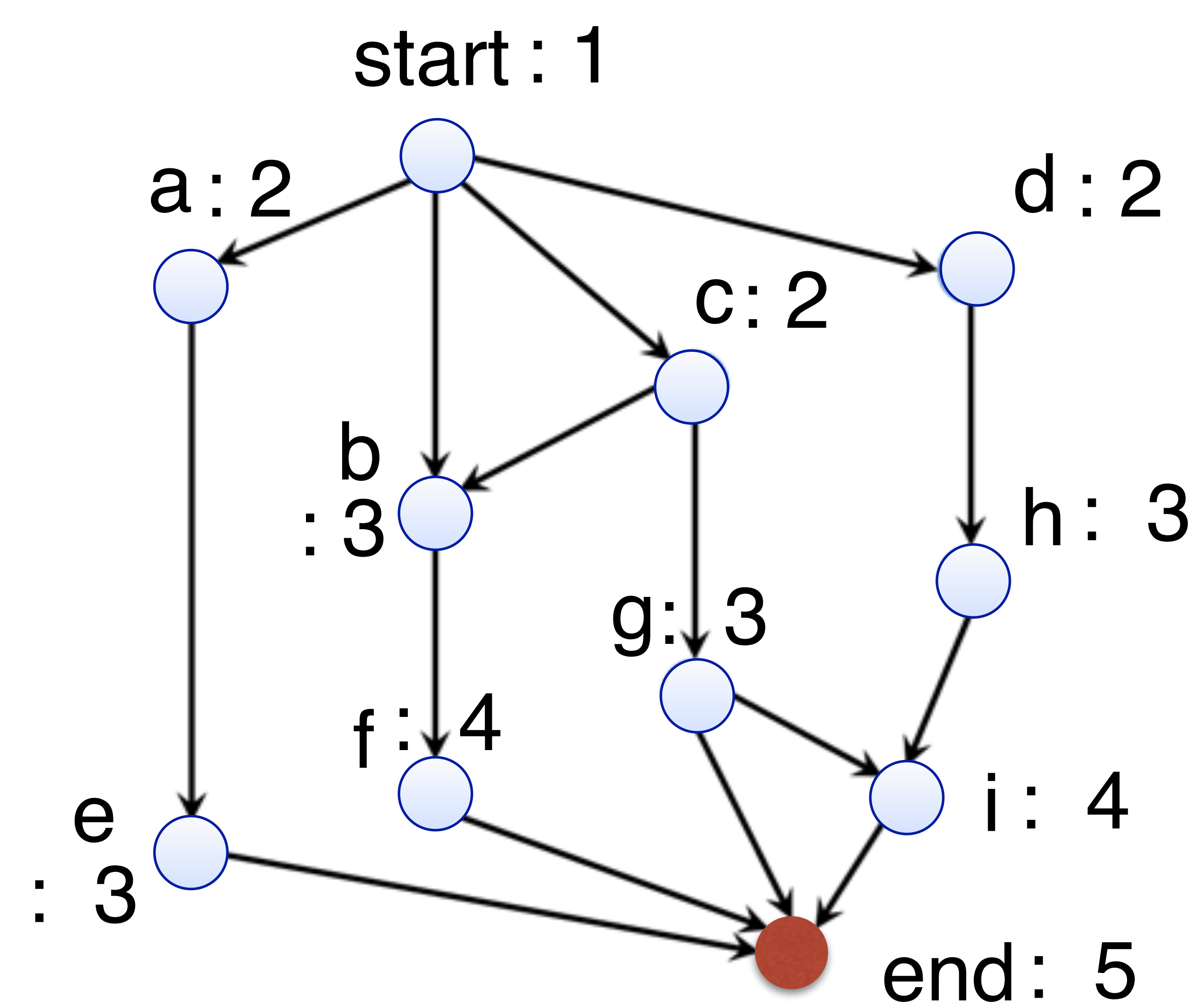
Computing Critical Path

Compute the earliest start time of each node

- Keep a value called S(n) associated with each node n
- For each node n

  S(n) is the maximum of { S(p) + 1 }, for all p ∈ pred(n)

Assuming a task takes 1 unit time



List Scheduling

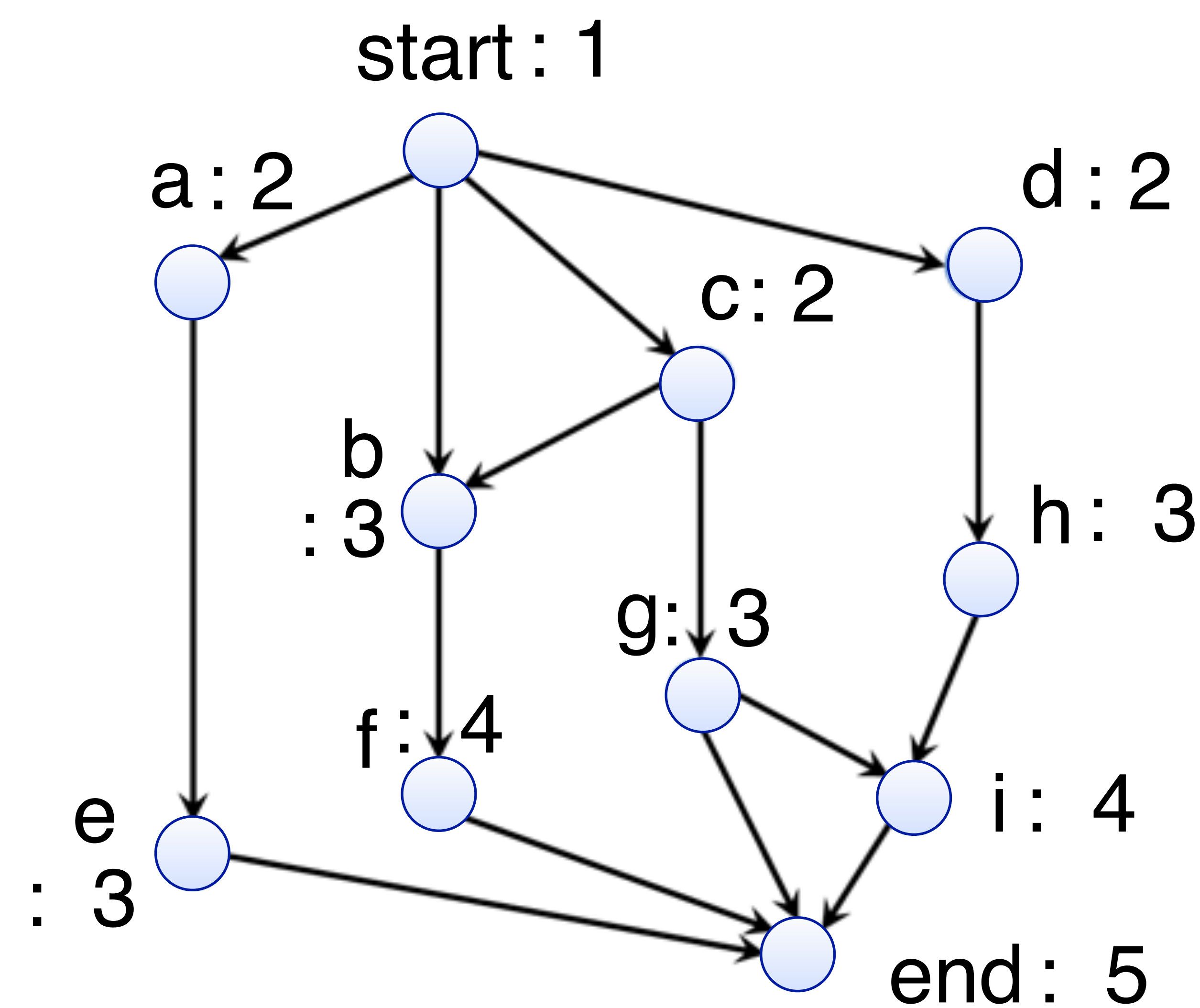
Based on if the dependence constraints have been resolved

- Schedule the nodes that are ready at every time tick
- A completed operation at the end of one time step can lead to more ready operations at next time tick

Four threads T1, T2, T3, T4

	1	2	3	4	5
T1	start	a	b	f	end
T2		c	e	i	
T3		d	g		
T4			h		

Assuming a task takes 1 unit time



Automatic Parallelization

We will use **loop analysis** as an example to describe automatic dependence analysis and parallelization.

Assumptions:

1. We only have scalar and subscripted variables (no pointers and no control dependence) for loop dependence analysis.
2. We focus on ***affine loops***: both loop bounds and memory references are affine functions of loop induction variables.

A function $f(x_1, x_2, ..., x_n)$ is **affine** if it is in such a form:

$\mathbf{f} = c_0 + c_1 * \boldsymbol{x_1} + c_2 * \boldsymbol{x_2} + \ldots + c_n * \boldsymbol{x_n}$, where c_i are all constants

Affine Loops

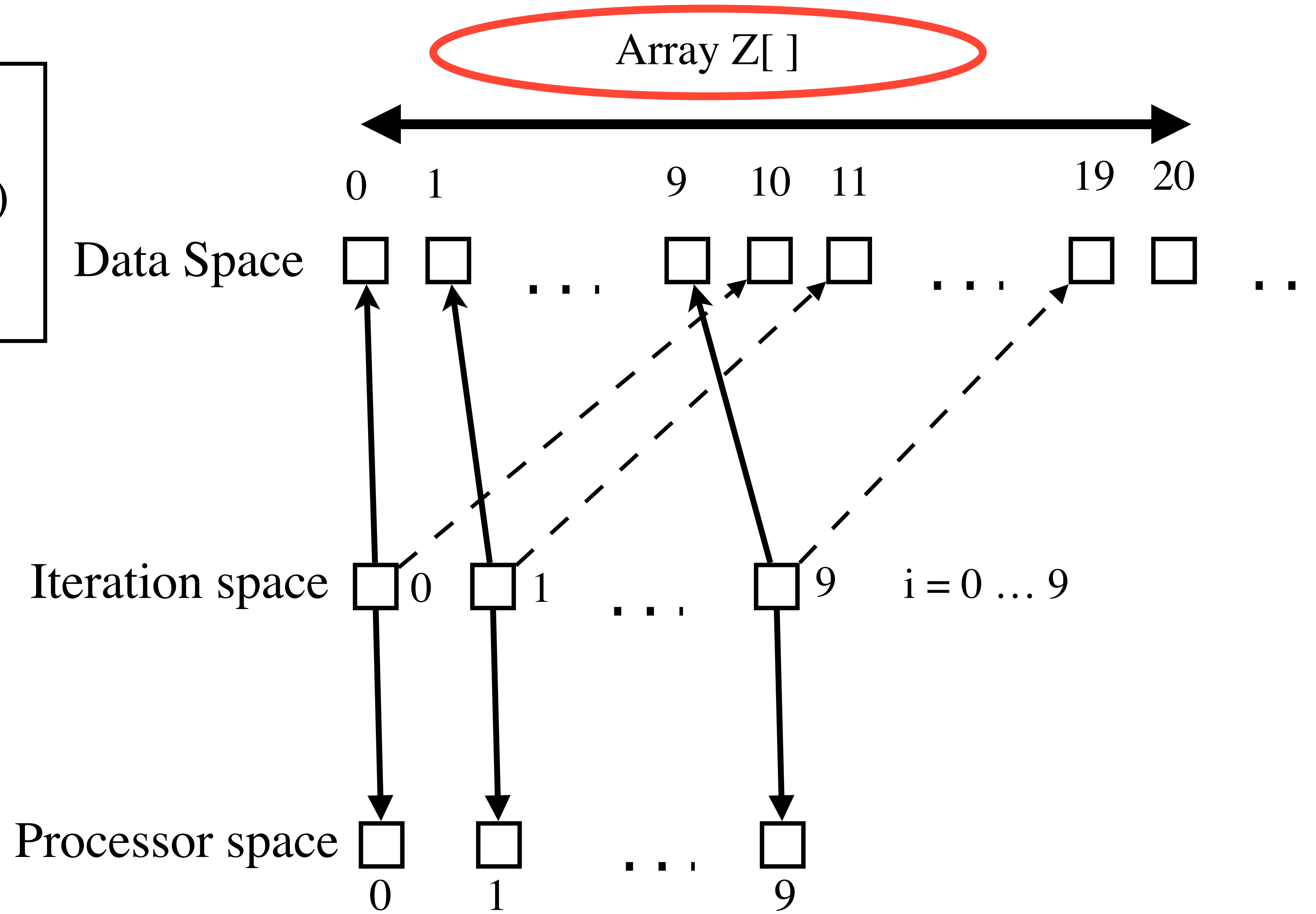
Three spaces

- Iteration space
  - The set of dynamic execution instances
  - i.e. the set of value vectors taken by loop indices
  - A k-dimensional space for a k-level loop nest
- Data space
  - The set of array elements accessed
  - An n-dimensional space for an n-dimensional array
- Processor space
  - The set of processors in the system
  - In analysis, we may pretend there are unbounded # of virtual processors

Three Spaces

- **Iteration space, data space, and processor space**

```
float Z[100];
for (i=0; i<10; i++)
  Z[i+10] = Z[i];
```



Assuming one task is one loop iteration, what is the maximum parallelism?

Maximum parallelism: T_1 / T_∞

Dependence Definition

Bernstein's Condition: — There is a data dependence from statement (instance) S_1 to statement S_2 (instance) if

- Both statements (instances) access the same memory locations
- One of them is a write
- There is a run-time execution path from S_1 to S_2

```
float Z[100];
for (i=0; i<10; i++)
  Z[i+10] = Z[i];
```

No dependence across any two loop iterations!

Data Dependence Classifications

"S_2 depends on S_1" — ($S_1\ \delta\ S_2$)

True (flow) dependence

occurs when S1 writes a memory location that S2 later reads (RAW).

Anti dependence

occurs when S1 reads a memory location that S2 later writes (WAR).

Output dependence

occurs when S1 writes a memory location that S2 later writes (WAW).

Input dependence

occurs when S1 reads a memory location that S2 later reads (RAR).

Simple Dependence Testing

- **Examples:**

```
for (i = 1; i <= 100; i++) {
  S1:  A[i] = ...
  S2:  ...= A[i - 1]
}
```

```
float Z[100];
for (i =0; i < 12; i++) {
   S: Z[ i+10 ] = Z[i];
}
```

1. Is there dependence?
2. If so, what type of dependence?
3. From which statement (instance) to which statement (instance)?

Dependence Testing

Single Induction Variable (SIV) Test

- Single loop nest with constant lower (LB) and upper (UB) bound, and step 1.

```
    for i = LB, UB, 1
          ...
    endfor
```

- Two array references as affine function of loop induction variable

```
    for i = LB, UB, 1
R1:     X(a*i + c1) = ...        \\ write
R2:       ... = X(a*i + c2) ...  \\ read
    endfor
```

Question: Is there a true dependence between R1 and R2?

Dependence Testing

```
    for i = LB, UB, 1
R1:     X(a*i + c1) = ...        \\ write
R2:       ... = X(a*i + c2) ...  \\ read
    endfor
```

There is a dependence between R1 and R2 **iff**

$$\exists\, i, i': \; LB \leq i \leq i' \leq UB \text{ and } (a * i + c_1) = (a * i' + c_2)$$

where i and i' represent two iterations in the iteration space. This means that in both iterations, the same element of array X is accessed.

So let's just solve the equation:

$$(a * i + c_1) = (a * i' + c_2) \quad \Rightarrow \quad (c_1 - c_2)/a = i' - i = \Delta d$$

There is a dependence iff

- Δd is an integer value
- $UB - LB \geq \Delta d \geq 0$

Simple Dependence Testing

- **Examples:**

```
for (i = 1; i <= 100; i++) {
   S1:  A[i] = ...
   S2:  ...= A[i - 1]
}
```

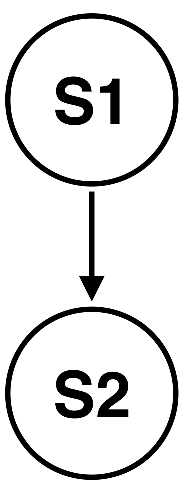
```
float Z[100];
for (i =0; i < 12; i++) {
    S: Z[ i+10 ] = Z[i];
}
```

1. Is there dependence?
2. If so, what type of dependence?
3. From which statement (instance) to which statement (instance)?

Simple Dependence Testing

- **Examples:**

```
for (i = 1; i <= 100; i++) {
   S1:  A[i] = ...
   S2:  ...= A[i - 1]
}
```

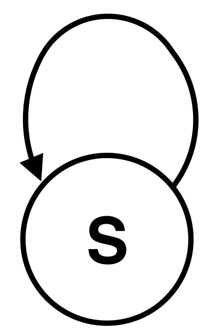


True Dependence
(read after write):
Wt: A[i] in S1 →
Rd: A[i’-1] in S2

i’ = **i** +1

Δd = 1

```
float Z[100];
for (i =0; i < 12; i++) {
   S: Z[ i+10 ] = Z[i];
}
```



True Dependence
(read after write):
Wt: Z[i+10] in S →
Rd: Z[i’] in S

i’ = **i** +10

Δd = 10

Simple Dependence Testing

- **More Examples:**

```
for (i = 1; i <= 100; i++) {
    R1:  X(i) = ...
    R2:  ... = X(i + 2)
}
```

```
for (i = 3; i <= 15, i++) {
    S1:  X(2 * i) = ...
    S2:  ... = X(2 * i - 1)
}
```

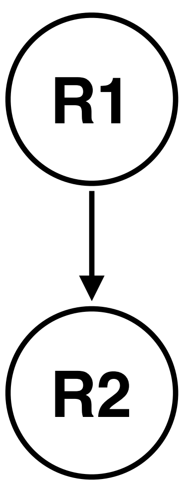
1. Is there dependence?
2. If so, what type of dependence?
3. From which statement (instance) to which statement (instance)?

Simple Dependence Testing

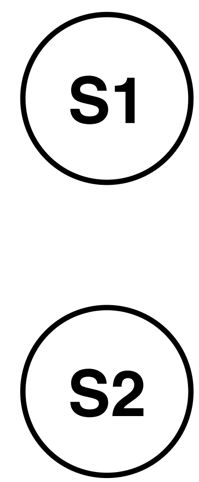
- **More Examples:**

```
for (i = 1; i <= 100; i++) {
    R1:  X[i] = ...
    R2:  ... = X[i + 2]
}
```

```
for (i = 3; i <= 15, i++) {
    S1:  X[2 * i] = ...
    S2:  ... = X[2 * i - 1]
}
```



Anti Dependence
(write after read):
Rd: X[i+2] in R2 →
Wt: X[i’] in R1



No dependence!

Next Class

Reading:

- ALSU, Chapter 11.1 - 11.3