Parallelism 2

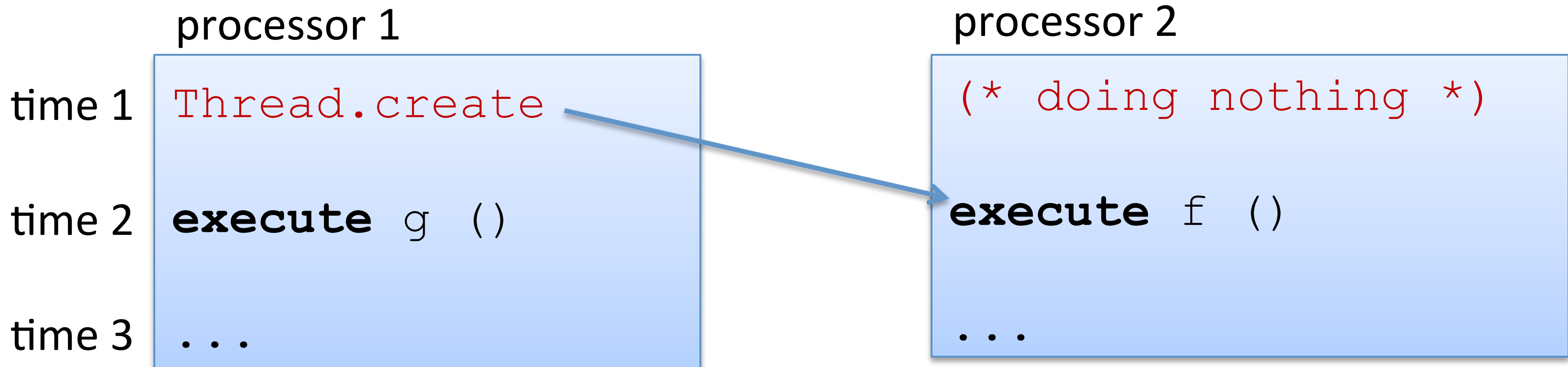
COS 326
David Walker
Princeton University

slides copyright 2017 David Walker
permission granted to reuse these slides for non-commercial educational purposes

Last Time: Threads!

A *thread*: an abstraction of a processor

```
let t = Thread.create f () in
let y = g () in
 ...
```



THREADS & COORDINATION

Coordination

```
Thread.create : ('a -> 'b) -> 'a -> Thread.t

let t = Thread.create f () in
let y = g () in
 ...
```

How do we get back the result that `t` is computing?

First Attempt

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
  match !r with
    | Some v -> (* compute with v and y *)
    | None -> ???
```

What’s wrong with this?

Second Attempt

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

Two Problems

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

First, we are *busy-waiting*.

- consuming cpu without doing something useful.
- the processor could be either running a useful thread/program or power down.

Two Problems

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

Second, an operation like r := Some v may not be *atomic.*

- r := Some v requires us to copy the bytes of Some v into the ref r
- we might see part of the bytes (corresponding to Some) before we've written in the other parts (e.g., v).
- So the waiter might see the wrong value.

An Aside: Atomicity

Consider the following:

```
let inc(r:int ref) = r := (!r) + 1
```

and suppose two threads are incrementing the same ref r:

Thread 1	Thread 2
`inc(r);`	`inc(r);`
`!r`	`!r`

If r initially holds 0, then what will Thread 1 see when it reads r?

Atomicity

The problem is that we can't see exactly what instructions the compiler might produce to execute the code.

It might look like this:

Thread 1	Thread 2
`R1 := load(p);`	`R1 := load(p);`
`R1 := R1 + 1;`	`R1 := R1 + 1;`
`store R1 into r`	`store R1 into p`
`R1 := load(p)`	`R1 := load(p)`

Atomicity

But a clever compiler might optimize this to:

<table>
<tr><th><u>Thread 1</u></th><th><u>Thread 2</u></th></tr>
<tr><td><code>R1 := load(p);</code></td><td><code>R1 := load(p);</code></td></tr>
<tr><td><code>R1 := R1 + 1;</code></td><td><code>R1 := R1 + 1;</code></td></tr>
<tr><td><code>store R1 into p</code></td><td><code>store R1 into p</code></td></tr>
<tr><td><s><code>R1 := load(r)</code></s></td><td><s><code>R1 := load(r)</code></s></td></tr>
</table>

Atomicity

Furthermore, we don't know when the OS might interrupt one thread and run the other.

<u>Thread 1</u>

```
R1 := load(p);
R1 := R1 + 1;
store R1 into p
R1 := load(r)
```

<u>Thread 2</u>

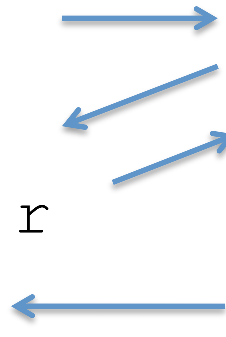
```
R1 := load(p);
R1 := R1 + 1;
store R1 into p
R1 := load(p)
```

(The situation is similar, but not quite the same on multi-processor systems.)

Atomicity

One possible interleaving of the instructions:

Thread 1	Thread 2
`R1 := load(p);`	`R1 := load(p);`
`R1 := R1 + 1;`	`R1 := R1 + 1;`
`store R1 into r`	`store R1 into p`
`R1 := load(p)`	`R1 := load(p)`



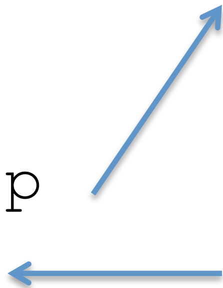
What answer do we get?

Atomicity

Another possible interleaving:

<u>Thread 1</u>

```
R1 := load(p);
R1 := R1 + 1;
store R1 into p
R1 := load(p)
```



<u>Thread 2</u>

```
R1 := load(p);
R1 := R1 + 1;
store R1 into p
R1 := load(p)
```

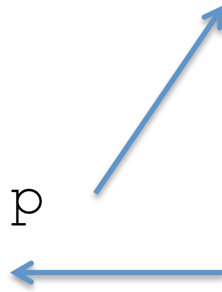
What answer do we get this time?

Atomicity

Another possible interleaving:

<u>Thread 1</u>

```
R1 := load(p);
R1 := R1 + 1;
store R1 into p
R1 := load(r)
```



<u>Thread 2</u>

```
R1 := load(p);
R1 := R1 + 1;
store R1 into p
R1 := load(p)
```

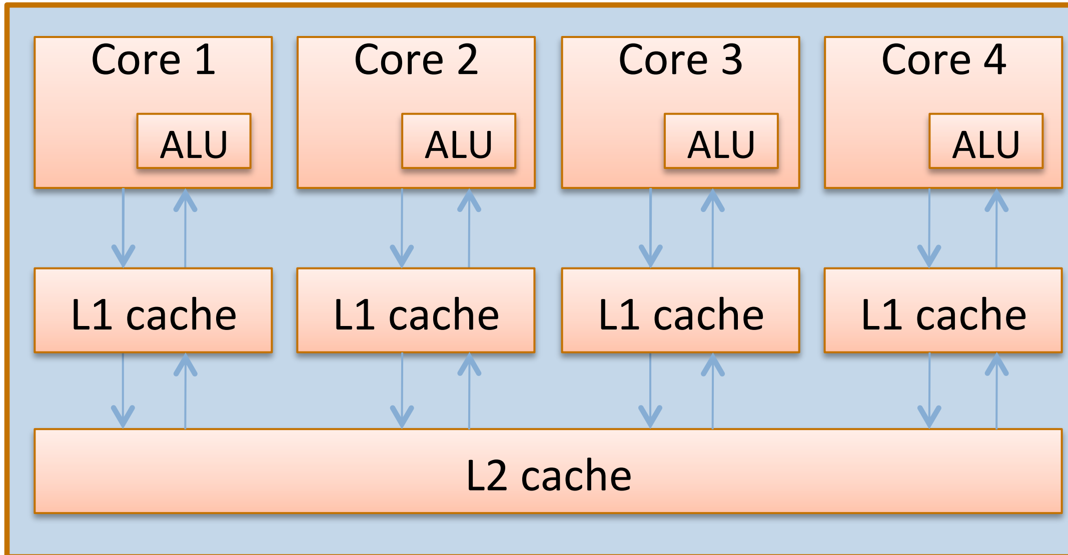
What answer do we get this time?

Moral: The system is responsible for *scheduling* execution of instructions.

Moral: This can lead to an enormous degree of *nondeterminism*.

Even Worse ...

In fact, today's multicore processors don't treat memory in a *sequentially consistent* fashion. That means that *we can't even assume that what we will see corresponds to some interleaving of the threads' instructions!*



When Core1 stores to "memory", it *lazily* propagates to Core2's L1 cache. The load at Core2 might not see it, unless there is an explicit synchronization.

Beyond the scope of this class! But the take-away is this: It's not a good idea to use ordinary loads/stores to synchronize threads; you should use explicit synchronization primitives so the hardware and optimizing compiler don't optimize them away.

Even Worse ...

In fact, today's multicore processors don't treat memory in a *sequentially consistent* fashion. That means that *we can't even assume that what we will see corresponds to some interleaving of the threads' instructions!*

Thread 1

```
R1  := load(r);
R1  := R1 + 1;
store R1 into r
R1  := load(r)
```

Thread 2

```
R1  := load(r);
R1  := R1 + 1;
store R1 into r
R1  := load(r)
```

Beyond the scope of this class! But the take-away is this: It's not a good idea to use ordinary loads/stores to synchronize threads; you should use explicit synchronization primitives so the hardware and optimizing compiler don't optimize them away.

The Happens Before Relation

We assume OCaml obeys a particular *Happens Before* relation:

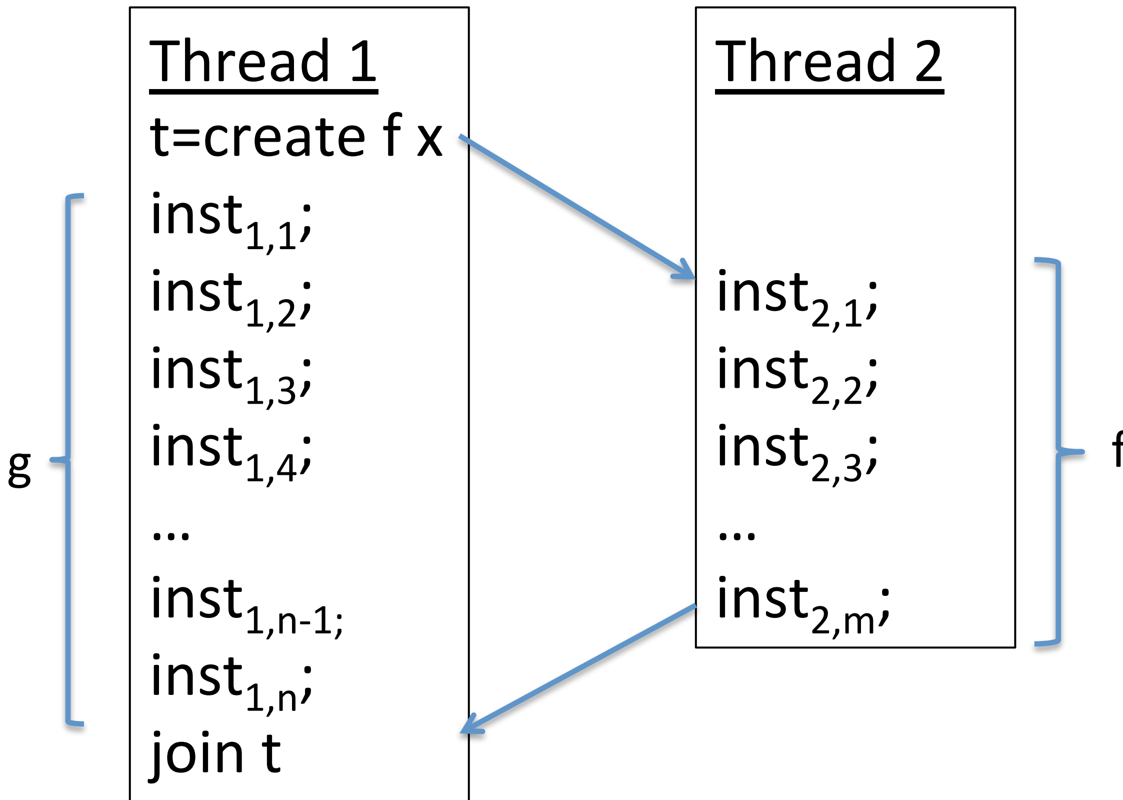
Rule 1: Given two expressions (or instructions) in sequence:
e1; e2 we know that *e1 happens before e2*.

Rule 2: Given a program:

```
let t = Thread.create f x in
   ....
Thread.join t;
e
```

we know that *(f x) happens before e*.

In Pictures



We know that for each thread the previous instructions must happen before the later instructions.

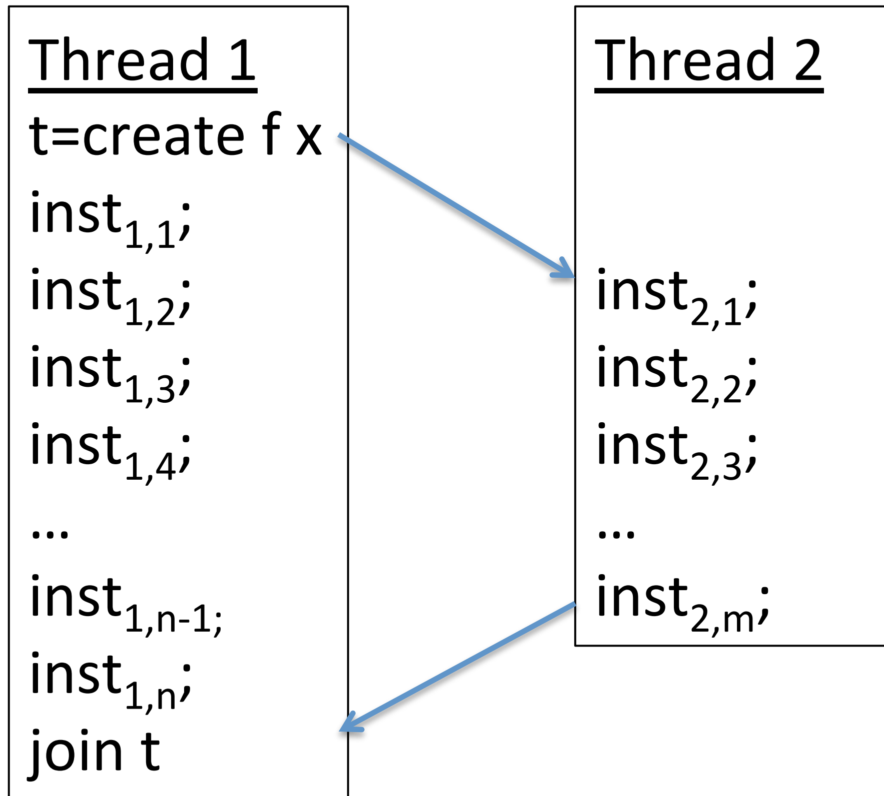
So for instance, $inst_{1,1}$ must happen before $inst_{1,2}$.

In Pictures

Thread 1	Thread 2
t=create f x	
$inst_{1,1}$;	
$inst_{1,2}$;	$inst_{2,1}$;
$inst_{1,3}$;	$inst_{2,2}$;
$inst_{1,4}$;	$inst_{2,3}$;
…	…
$inst_{1,n-1;}$	$inst_{2,m}$;
$inst_{1,n}$;	
join t	

We also know that the fork must happen before the first instruction of the second thread.

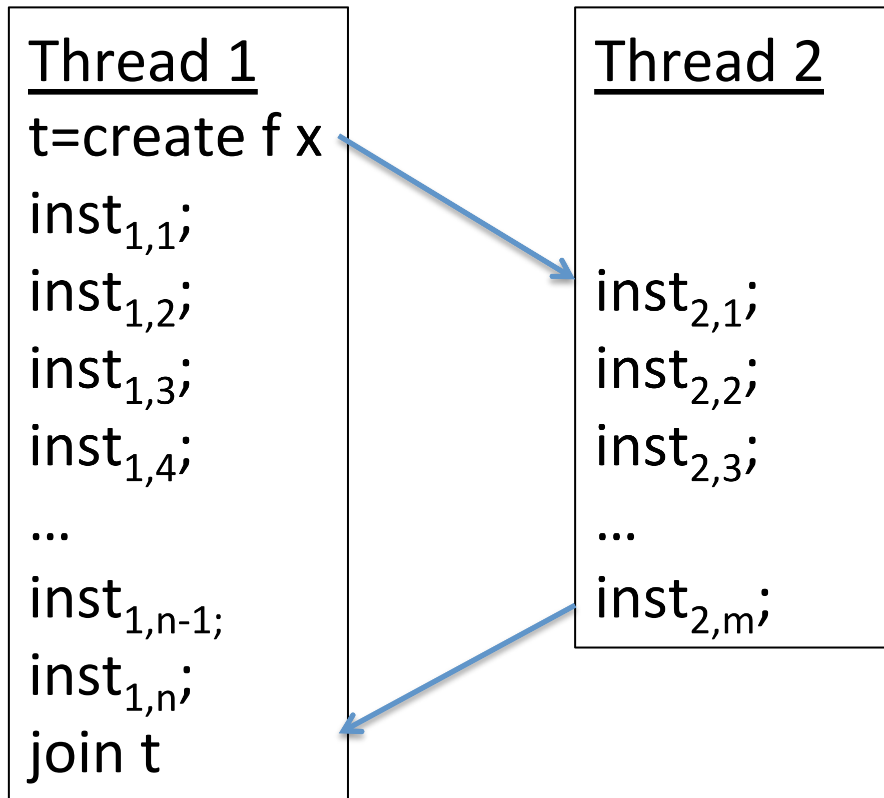
In Pictures



We also know that the fork must happen before the first instruction of the second thread.

And thanks to the join, we know that all of the instructions of the second thread must be completed before the join finishes.

In Pictures



However, in general, we do not know whether $inst_{1,i}$ executes before or after $inst_{2,j}$.

In general, *synchronization instructions like fork and join reduce the number of possible interleavings.*

Synchronization cuts down nondeterminism.

In the absence of synchronization we don't know anything…

Another approach to the coordination Problem

```
Thread.create : ('a -> 'b) -> 'a -> Thread.t

let t = Thread.create f () in
let y = g () in
  ...
```

How do we get back the result that `t` is computing?

One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
    Thread.join t ;
    match !r with
    | Some v -> (* compute with v and y *)
    | None -> failwith "impossible"
```

One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
    Thread.join t ;
    match !r with
    | Some v -> (* comput   ith v and y *)
    | None -> failwith "impos     e"
```

`Thread.join t` causes the current thread to *wait* until the thread `t` terminates.

One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
    Thread.join t ;
    match !r with
    | Some v -> (* compute with v and y *)
    | None -> failwith "impossible"
```

Synchronization

So after the join, we know that any of the operations of t have *completed*.

FUTURES: A PARALLEL PROGRAMMING ABSTRACTION

Futures

The fork-join pattern we just saw is so common, we'll create an abstraction for it:

```
module type FUTURE =
sig
  type 'a future

  (* future f x forks a thread to run f(x)
     and stores the result in a future when complete *)
  val future : ('a->'b) -> 'a -> 'b future

  (* force f causes us to wait until the
     thread computing the future value is done
     and then returns its value. *)
  val force : 'a future -> 'a
end
```

Does that interface looks familiar .... ?

Future Implementation

```
module Future : FUTURE =
struct
  type ‘a future = {tid   : Thread.t       ;
                    value : ’a option ref }

end
```

Future Implementation

```
module Future : FUTURE =
struct
  type 'a future = {tid    : Thread.t      ;
                    value : 'a option ref }

  let future(f:'a->'b)(x:'a) : 'b future =
    let r = ref None in
    let t = Thread.create (fun () -> r := Some(f x)) ()
    in
    {tid=t ; value=r}

end
```

Future Implementation

```
module Future : FUTURE =
struct
  type 'a future = {tid   : Thread.t      ;
                    value : 'a option ref }

  let future(f:'a->'b)(x:'a) : 'b future =
    let r = ref None in
    let t = Thread.create (fun () -> r := Some(f x)) ()
    in
    {tid=t ; value=r}

  let force (f:'a future) : 'a =
    Thread.join f.tid ;
    match !(f.value) with
    | Some v -> v
    | None -> failwith "impossible!"
end
```

Now using Futures

```
let x = future f () in
let y = g () in
let v = force x in
(* compute with v and y *)
```

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + v
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + v
```

without futures library:

```
let r = ref None
let t = Thread.create
          (fun _ -> r := Some(f ()))
          ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

what happens if we delete these lines?

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
          (fun _ -> r := Some(f ()))
          ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

what happens if we use x and forget to force?

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let y = g () in
let v = force x in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
          (fun _ -> r := Some(f ()))
          ()
in
let y = g() in
Thread.join t ;
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

Moral: Futures + typing ensure entire categories of errors can't happen -- you protect yourself from your own stupidity

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let v = force x in
let y = g () in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
          (fun _ -> r := Some(f ()))
          ()
in
Thread.join t ;
let y = g() in
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

what happens if you relocate force, join?

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force :'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let v = force x in
let y = g () in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
           (fun _ -> r := Some(f ()))
           ()
in
Thread.join t ;
let y = g() in
match !r with
    Some v -> y + v
  | None -> failwith "impossible"
```

Moral: Futures are not a universal savior

An Example: Mergesort on Arrays

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) : 'a array =
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1 = msort start half in
             let a2 = msort (start + half)
                            (len - half) in
               merge a1 a2

  and merge (a1:'a array) (a2:'a array) : 'a array =
      ...
```

An Example: Mergesort on Arrays

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) :
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1 = msort start half in
             let a2 = msort (start + half)
                            (len - half) in
               merge a1 a2

  and merge (a1:'a array) (a2:'a array) : 'a array =
      ...
```

Opportunity for parallelization

Making Mergesort Parallel

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) : 'a array =
    match len with
      | 0 -> Array.of_list []
      | 1 -> Array.make 1 arr.(start)
      | _ -> let half = len / 2 in
             let a1_f =
               Future.future (msort start) half in
             let a2 =
               msort (start + half)(len - half) in
             merge (Future.force a1_f) a2

  and merge (a1:'a array) (a2:'a array) : 'a array =
```

Divide-and-Conquer

This is an instance of a basic *divide-and-conquer* pattern in parallel programming

- take the problem to be solved and divide it in half
- fork a thread to solve the first half
- simultaneously solve the second half
- synchronize with the thread we forked to get its results
- combine the two solution halves into a solution for the whole problem.

Warning: the fact that we only had to rewrite 2 lines of code for mergesort made the parallelization transformation look deceptively easy

- we also had to verify that any two threads did not touch overlapping portions of the array -- if they did we would have to again worry about scheduling nondeterminism

Caveats

There is some overhead for creating a thread.

- On uniprocessor, parallel code *slower* than sequential code.

Even on a multiprocessor, we do *not always* want to fork.

- when the subarray is small, faster to sort it sequentially than to fork
  - similar to using insertion sort when arrays are small vs. quicksort
- this is known as a *granularity problem*
  - more parallelism than we can effectively take advantage of.

Caveats

In a good implementation of futures, a compiler and run-time system might look to see whether the cost of doing the fork is justified by the amount of work that will be done. Today, it's up to you to figure this out… ☹

- typically, use parallel divide-and-conquer until:
  - (a) we have generated *at least* as many threads as there are processors
    - often *more threads* than processors because different jobs take different amounts of time to complete and we would like to keep all processors busy
  - (b) the sub-arrays have gotten small enough that it's not worth forking.

We're not going to worry about these performance-tuning details too much but rather focus on the distinctions between *parallel* and *sequential algorithms*.

Another Example

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {left  : 'a tree ;
               value : 'a      ;
               right : 'a tree }

let rec fold (f:'a -> 'b -> 'b -> 'b) (u:'b)
             (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
     f n.value (fold f u n.left) (fold f u n.right)

let sum (t:int tree) = fold (+) 0 t
```

Another Example

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {left  : 'a tree ;
               value : 'a      ;
               right : 'a tree }

let rec pfold (f:'a -> 'b -> 'b -> 'b) (u:'b)
              (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
     let l_f = Future.future (pfold f u) n.left in
     let r = pfold f u n.right in
     f n.value (Future.force l_f) r

let sum (t:int tree) = pfold (+) 0 t
```

Note

If the tree is unbalanced, then we're not going to get the same speedup as if it's balanced.

Consider the degenerate case of a list.

- The forked child will terminate without doing any useful work.
- So the parent is going to have to do all that work.
- Pure overhead... ☹

In general, lists are a horrible data structure for parallelism.

- *we can't cut the list in half in constant time*
- for arrays and trees, we can do that (assuming the tree is balanced.)

Side Effects?

```
type 'a tree = Leaf | Node of 'a node
and 'a node = { left  : 'a tree ;
                value : 'a      ;
                right : 'a tree }

let rec pfold (f:'a -> 'b -> 'b -> 'b) (u:'b)
              (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
     let l_f = Future.future (pfold f u) n.left in
     let r = pfold f u n.right in
     f n.value (Future.force l_f) r

let print (t:int tree) =
  pfold (fun n _ _ -> Printf.print "%d\n" n) ()
```

Huge Point

If code is purely functional, then it never matters in what order it is run.

If f () and g () are pure then all of the following are equivalent:

```
let x = f() in
let y = g() in
e
```

```
let x_f = future f () in
let y   = g ()        in
let x   = force x_f   in
e
```

```
let y = g () in
let x = f () in
e
```

```
let y_g = future g () in
let x   = f ()        in
let y   = force y_g   in
e
```

Huge Point

If code is purely functional, then it never matters in what order it is run.

If f () and g () are pure then all of the following are equivalent:

```
let x = f() in
let y = g() in
e
```

```
let x_f = future f () in
let y   = g ()        in
let x   = force x_f   in
e
```

```
let y = g () in
let x = f () in
e
```

```
let y_g = future g () in
let x   = f ()        in
let y   = force y_g   in
e
```

As soon as we introduce *side-effects*, the order starts to matter.

- This is why, IMHO, *imperative* languages where even the simplest of program phrases involves a side effect, are doomed.
- Of course, we've been saying this for 30 years!
- See J. Backus's Turing Award lecture, *"Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs."*

  http://www.cs.cmu.edu/~crary/819-f09/Backus78.pdf

Future Reasoning in a Nutshell

```
f ()  ==  force (future f ())
```

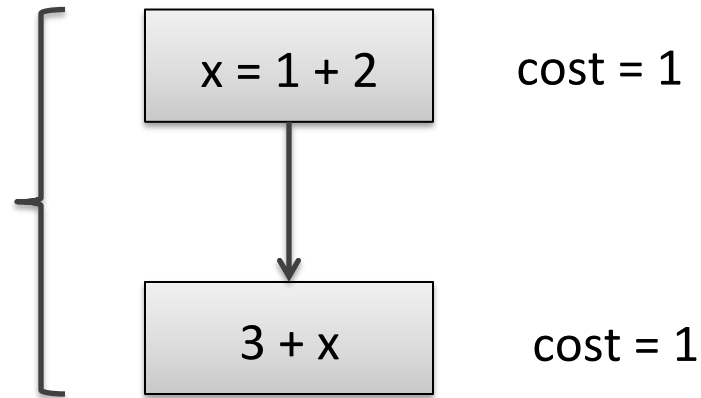
Scheduling
Parallel Computations

Visualizing Computational Costs

```
let x = 1 + 2 in
3 + x
```

Visualizing Computational Costs

let x = 1 + 2 in
3 + x

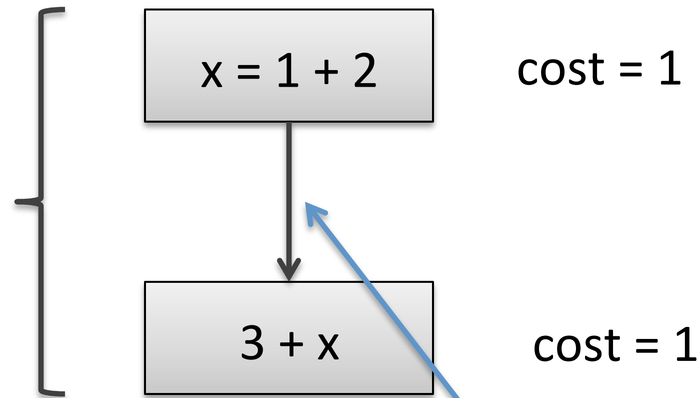


cost = 1

cost = 1

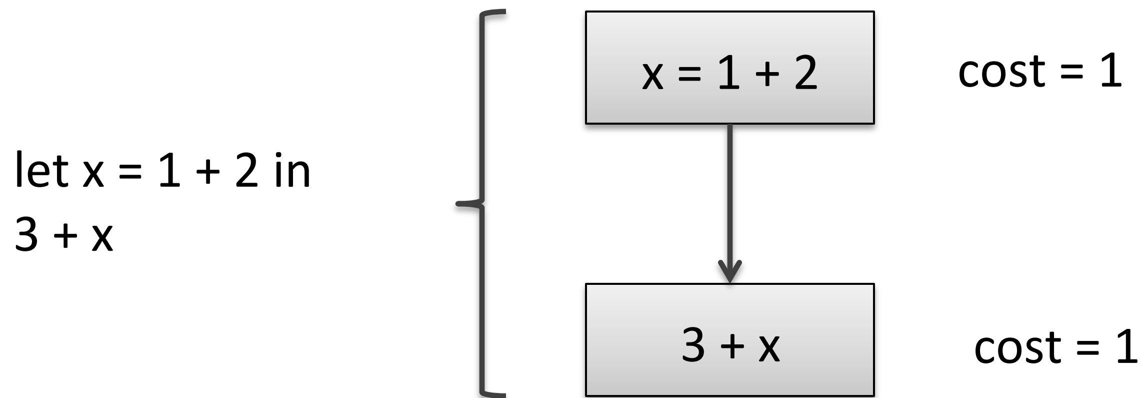
Visualizing Computational Costs

let x = 1 + 2 in
3 + x



dependence:
x = 1 + 2 *happens before* 3 + x

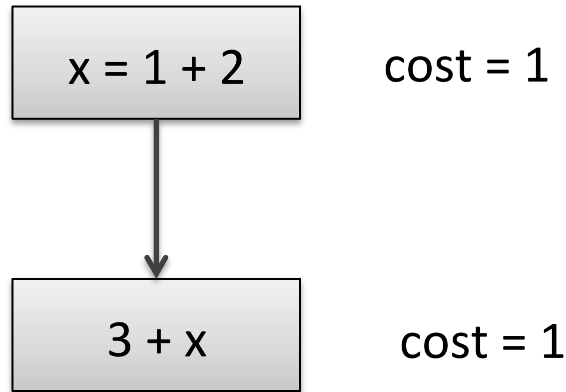
Visualizing Computational Costs



Execution of dependency diagrams: A processor can only begin executing the computation associated with a block when the computations of all of its predecessor blocks have been completed.

Visualizing Computational Costs

step 1:
execute first block



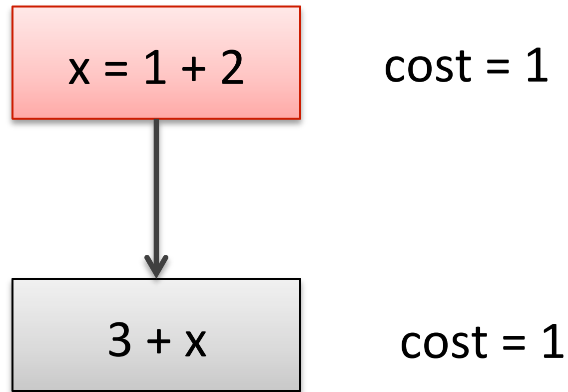
cost = 1

cost = 1

Cost so far: 0

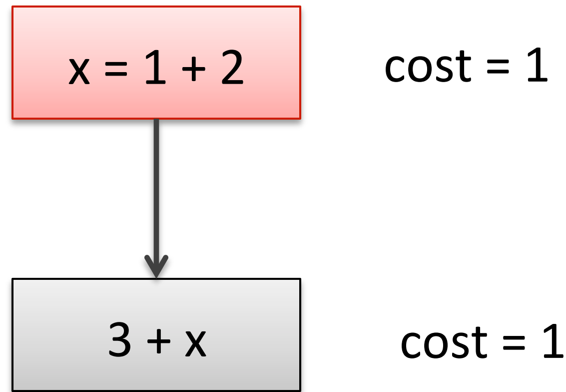
Visualizing Computational Costs

step 1:
execute first block



Cost so far: 1

Visualizing Computational Costs



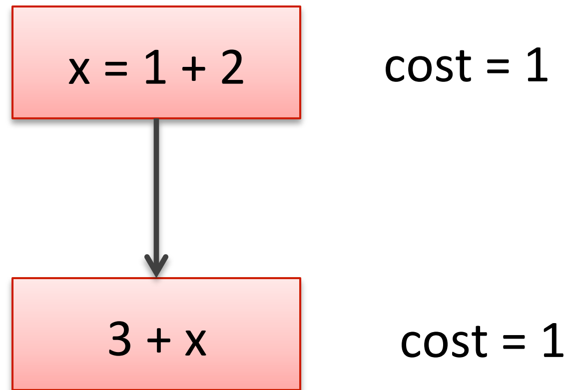
cost = 1

cost = 1

step 2:
execute second block because all of its predecessors have been completed

Cost so far: 1

Visualizing Computational Costs

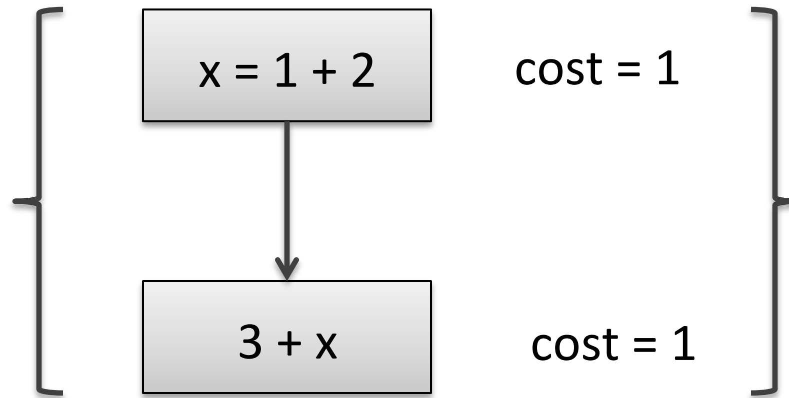


step 2:
execute second block because all of its predecessors have been completed

Cost so far: 1 + 1

Visualizing Computational Costs

let x = 1 + 2 in
3 + x

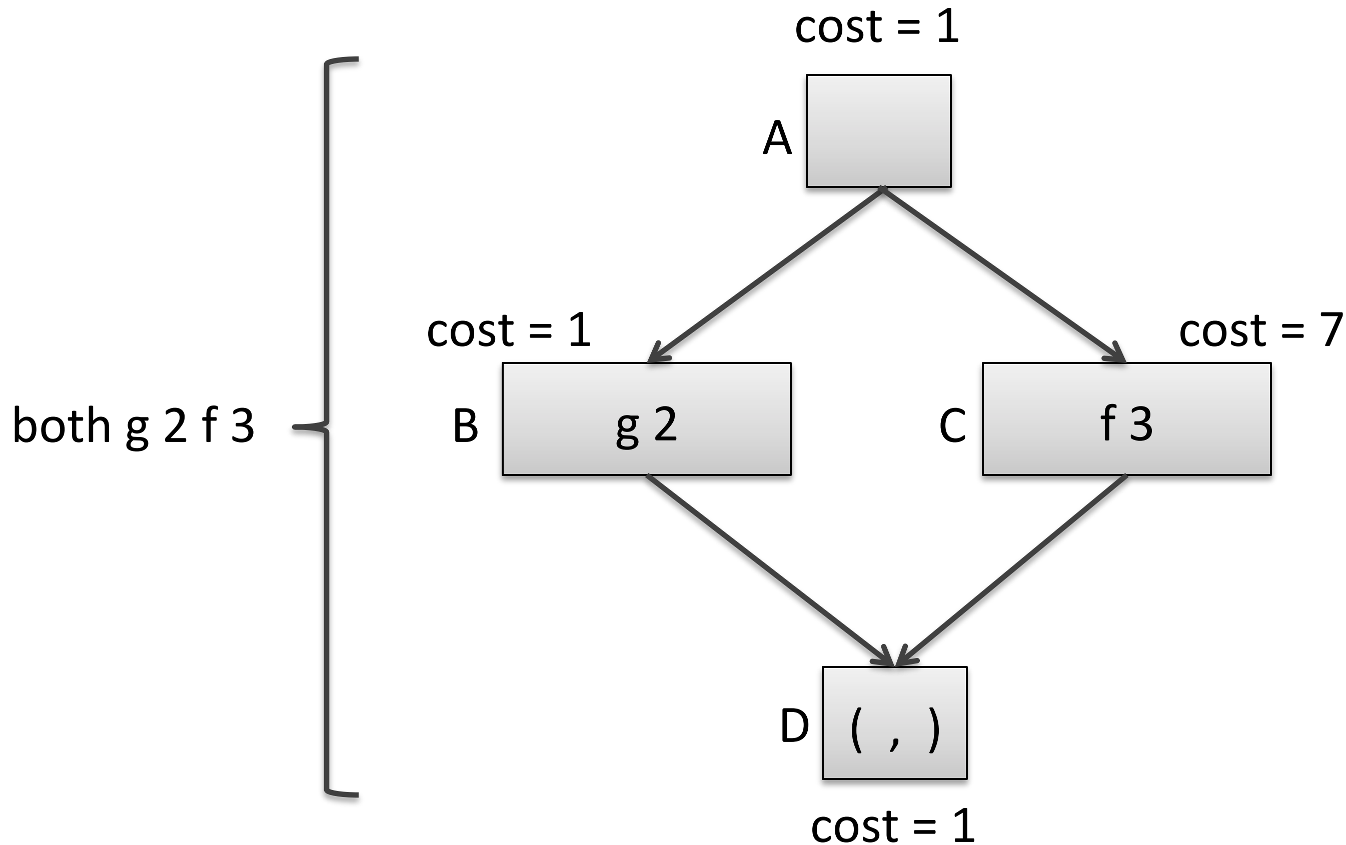


total cost
= 1 + 1
= 2

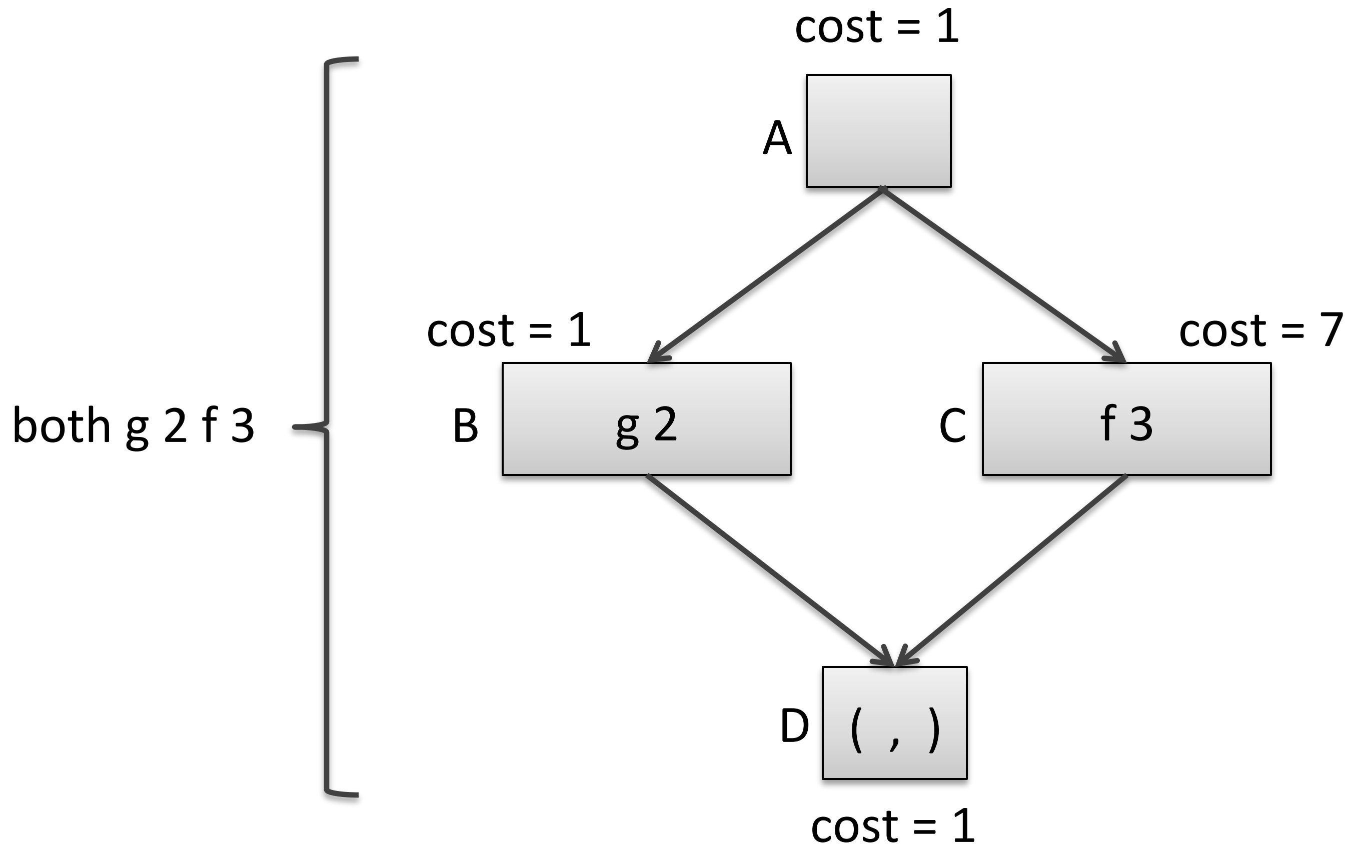
A Handy Abbreviation

```
let both f x g y =
  let r1 = future f x in
  let r2 = future g y in
  (force r1, force r2)
```

Visualizing Computational Costs

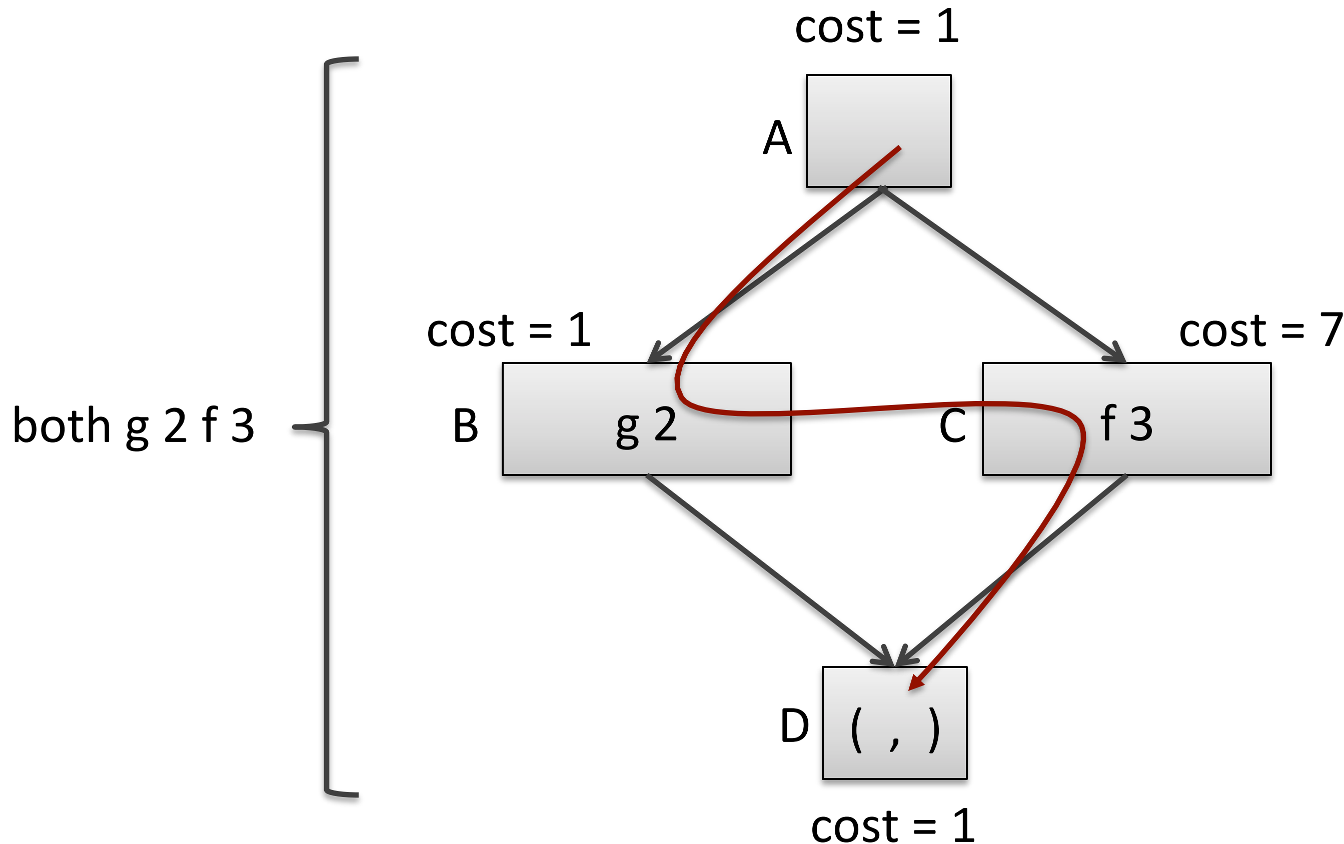


Visualizing Computational Costs



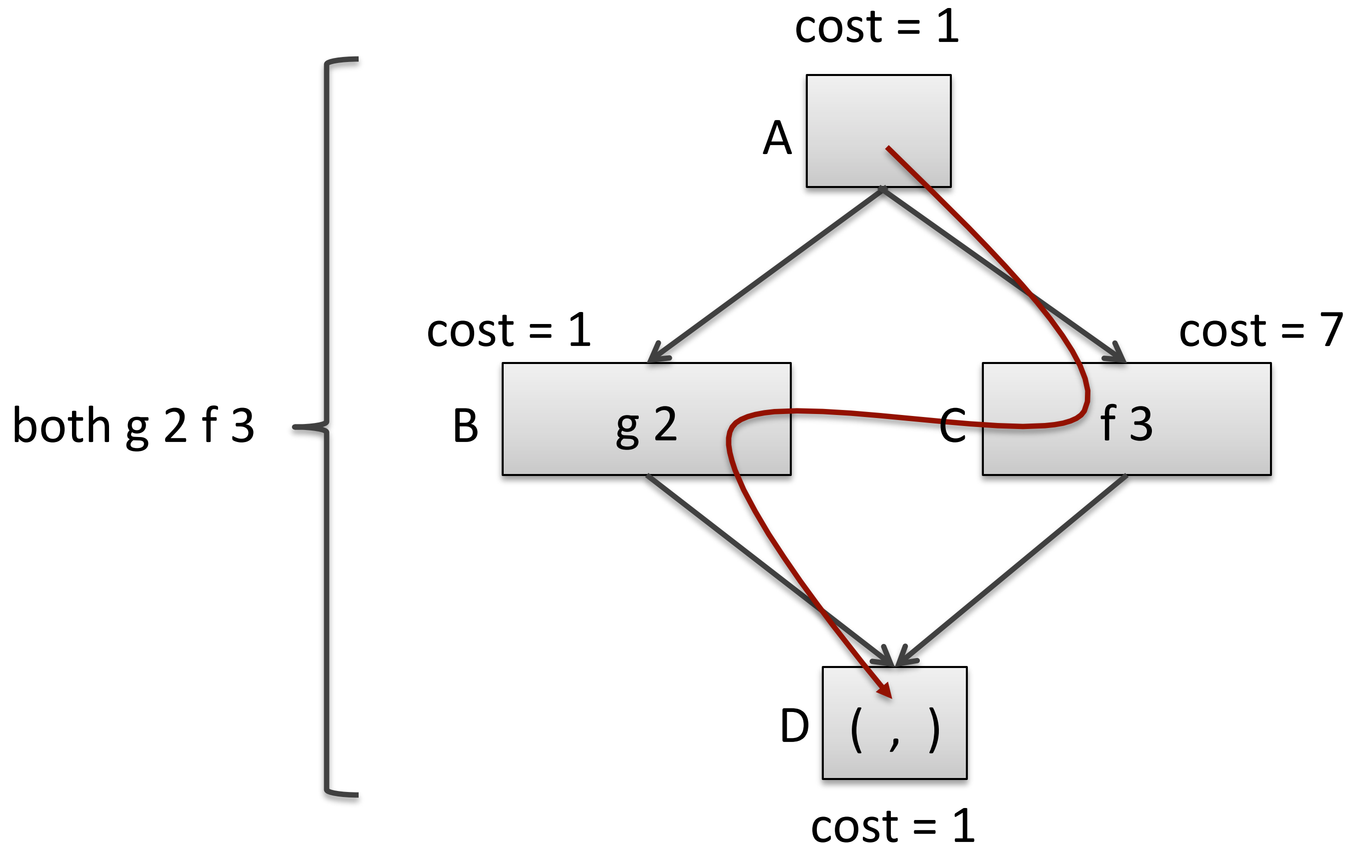
Suppose we have 1 processor. How much time does this computation take?

Visualizing Computational Costs



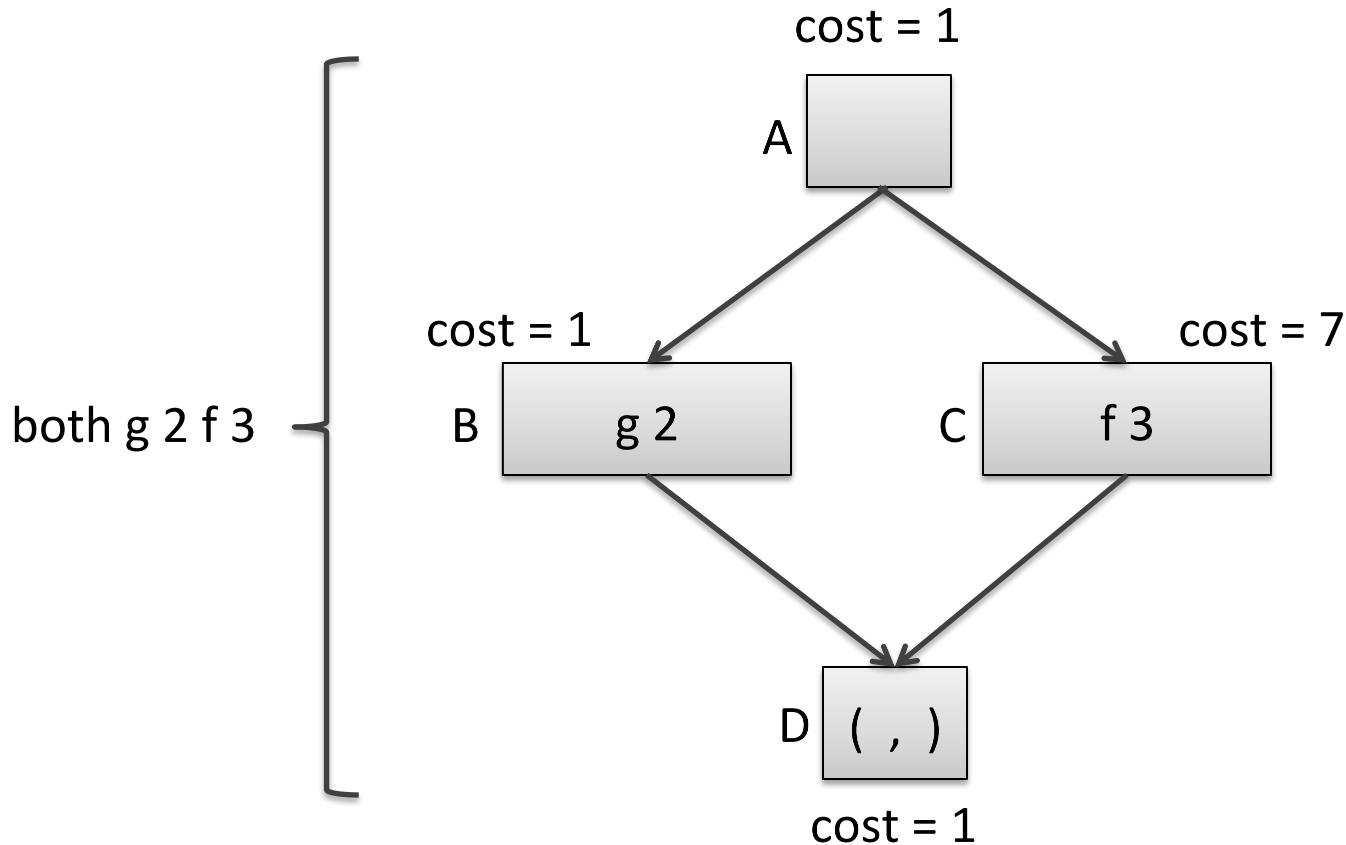
Suppose we have 1 processor. How much time does this computation take?
Scheduld A-B-C-D: 1 + 1 + 7 + 1

Visualizing Computational Costs



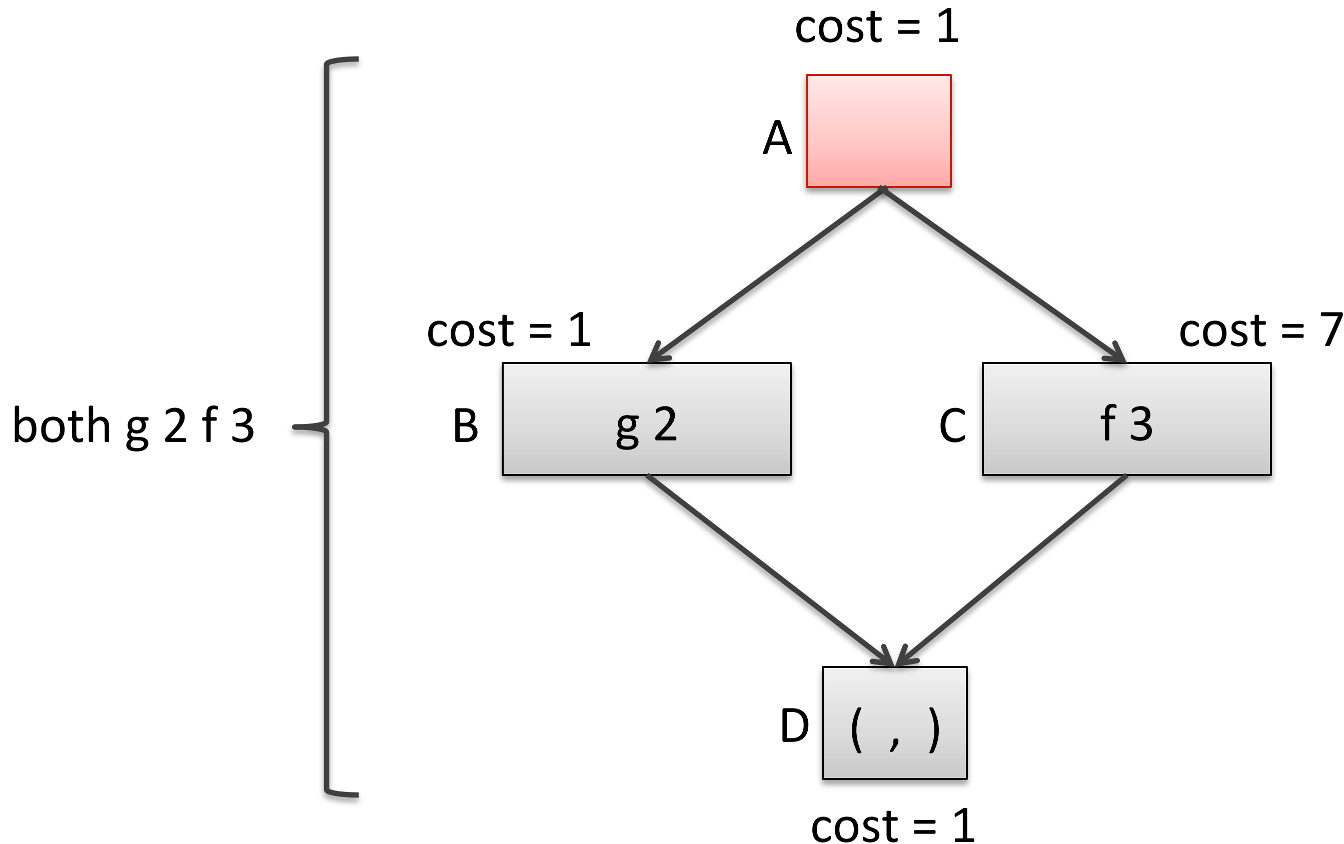
Suppose we have 1 processor. How much time does this computation take?
Schedule A-C-B-D: 1 + 1 + 7 + 1

Visualizing Computational Costs



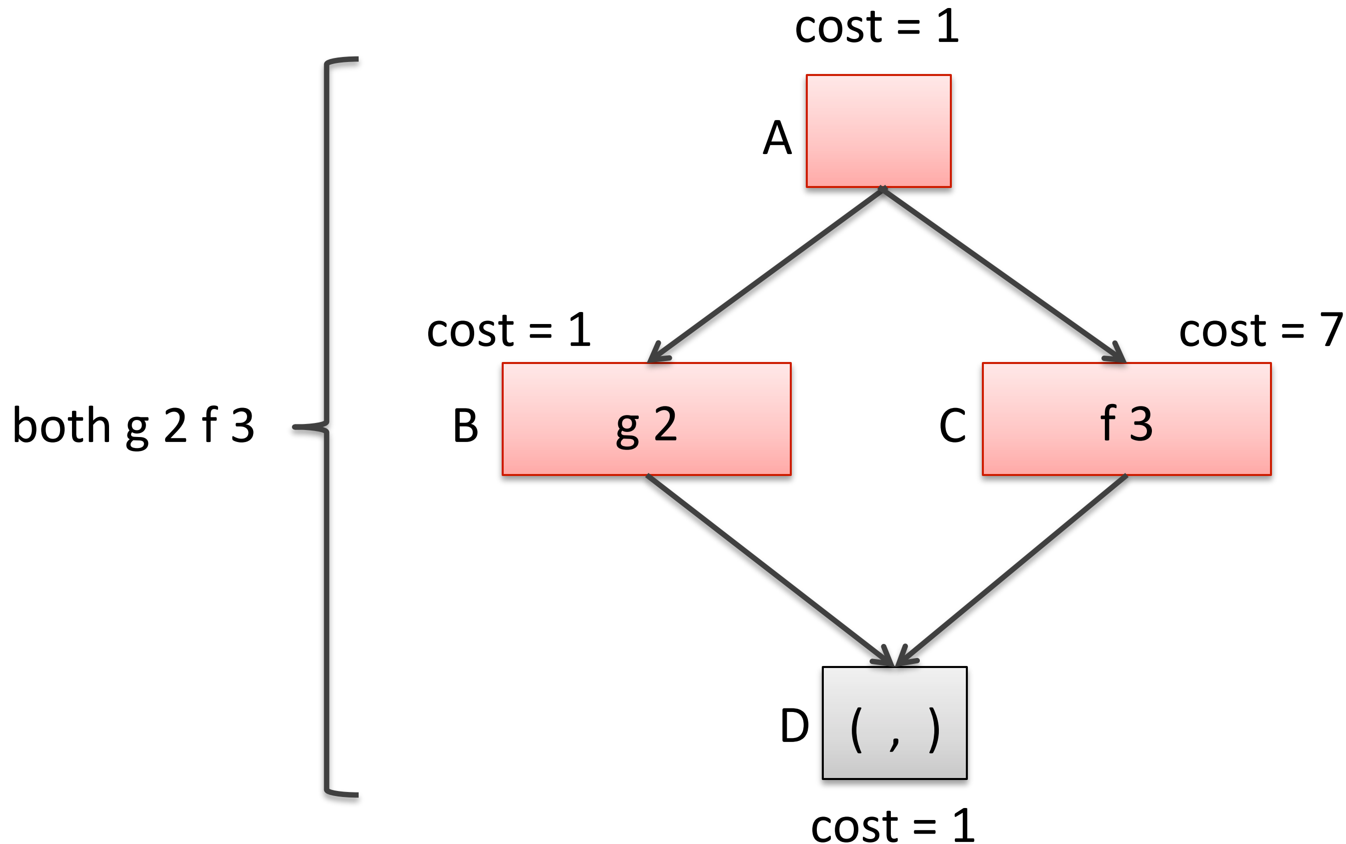
Suppose we have 2 processors. How much time does this computation take?

Visualizing Computational Costs



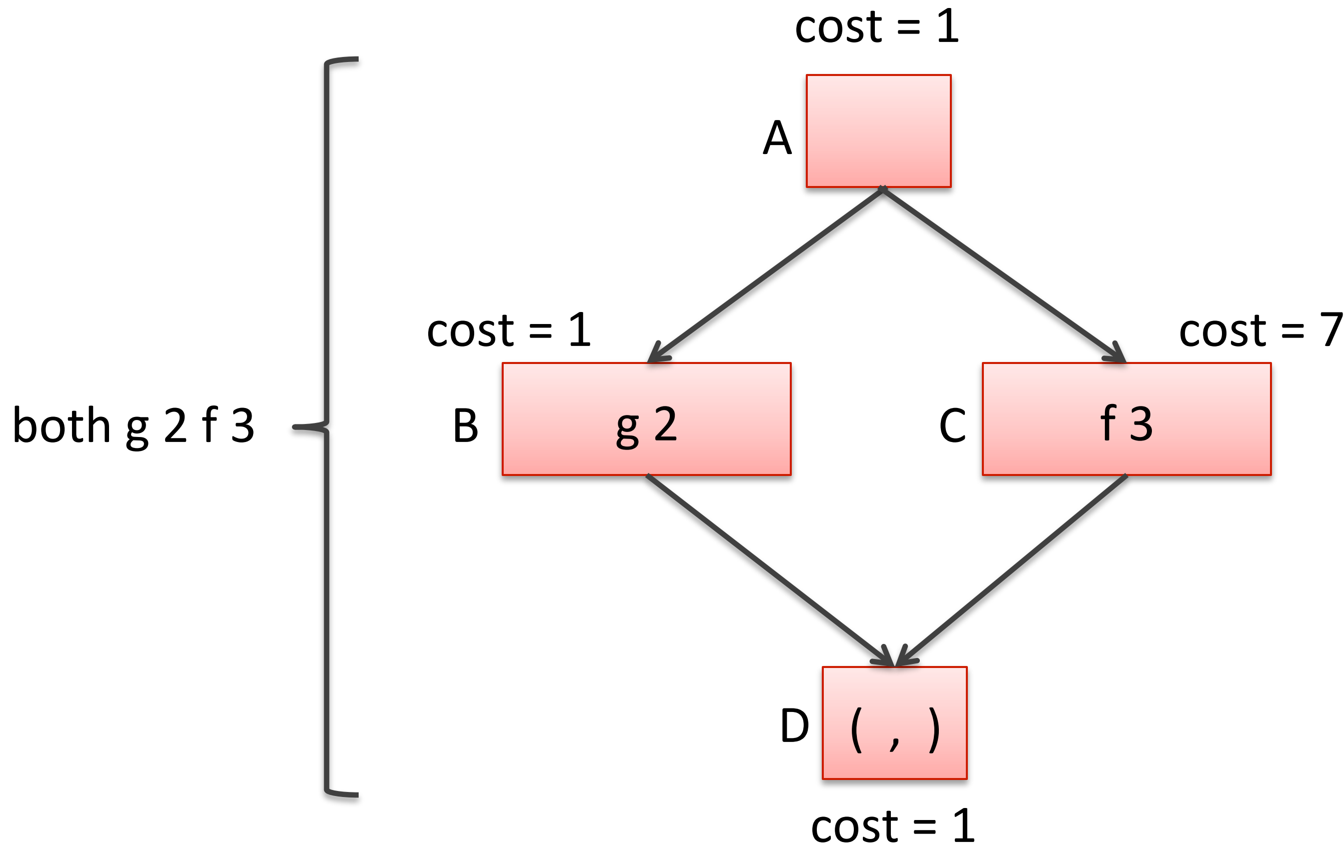
Suppose we have 2 processors. How much time does this computation take?
Cost so far: 1

Visualizing Computational Costs



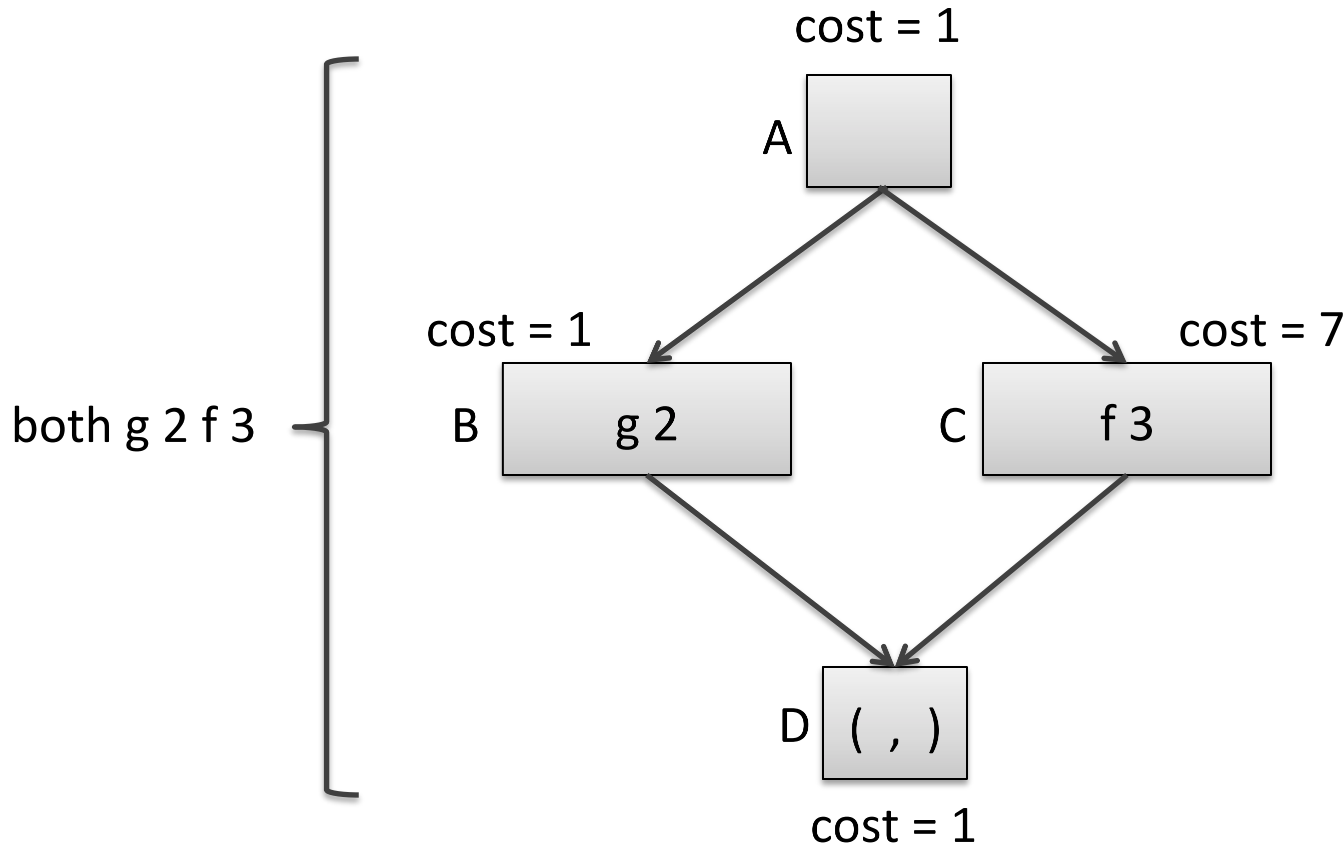
Suppose we have 2 processors. How much time does this computation take?
Cost so far: 1 + max(1,7)

Visualizing Computational Costs



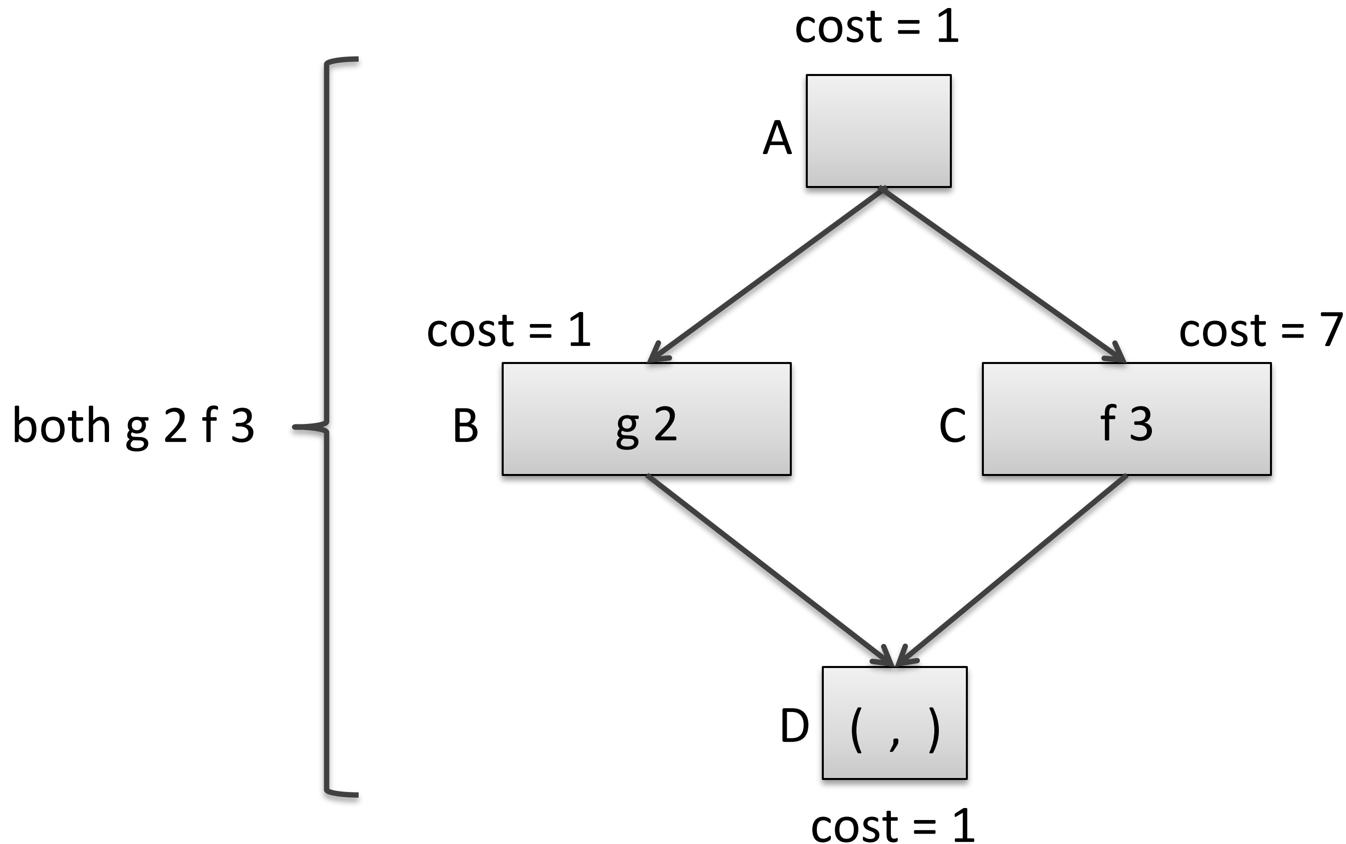
Suppose we have 2 processors. How much time does this computation take?
Cost so far: 1 + max(1,7) + 1

Visualizing Computational Costs



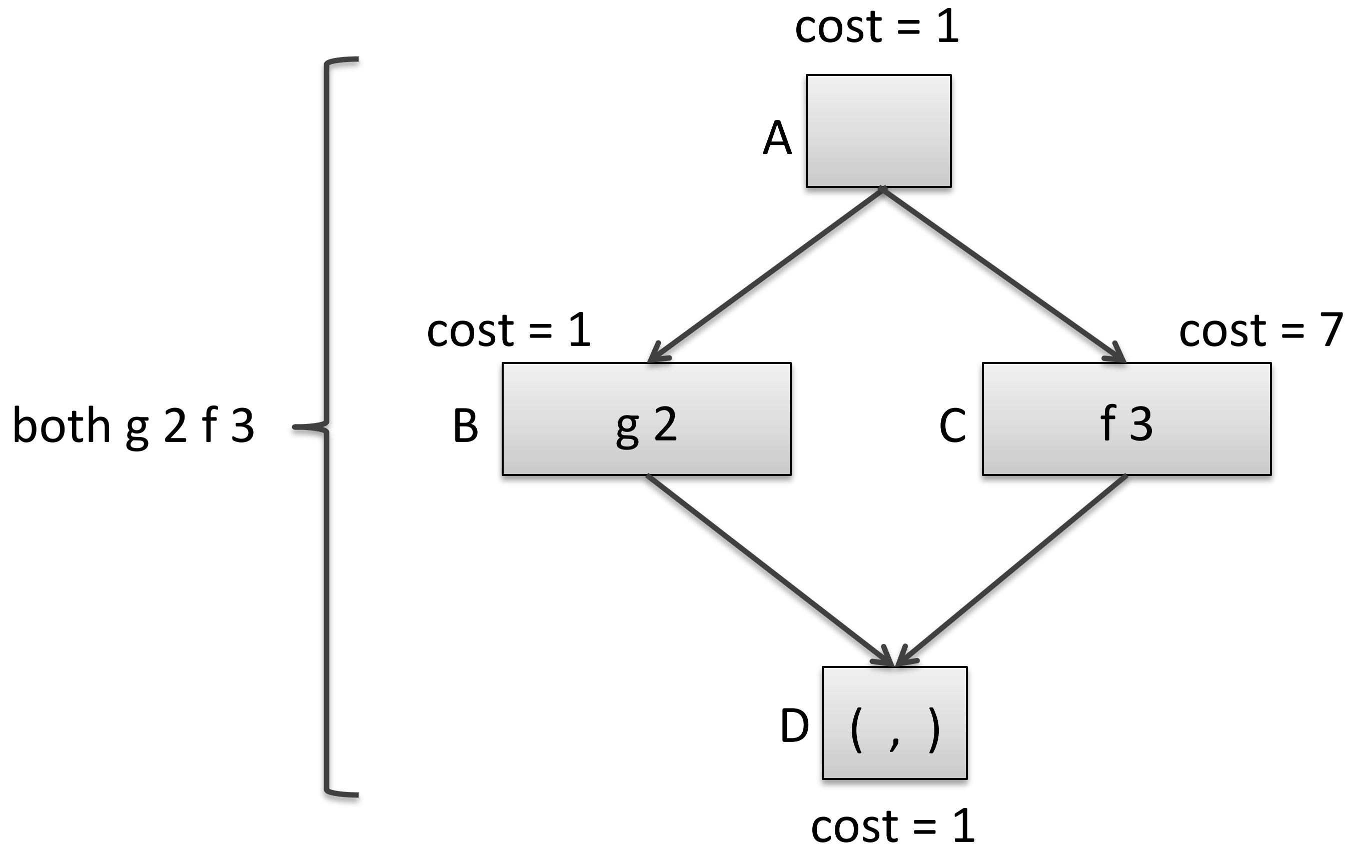
Suppose we have 2 processors. How much time does this computation take?
Total cost: 1 + max(1,7) + 1. We say the *schedule* we used was: A-CB-D

Visualizing Computational Costs



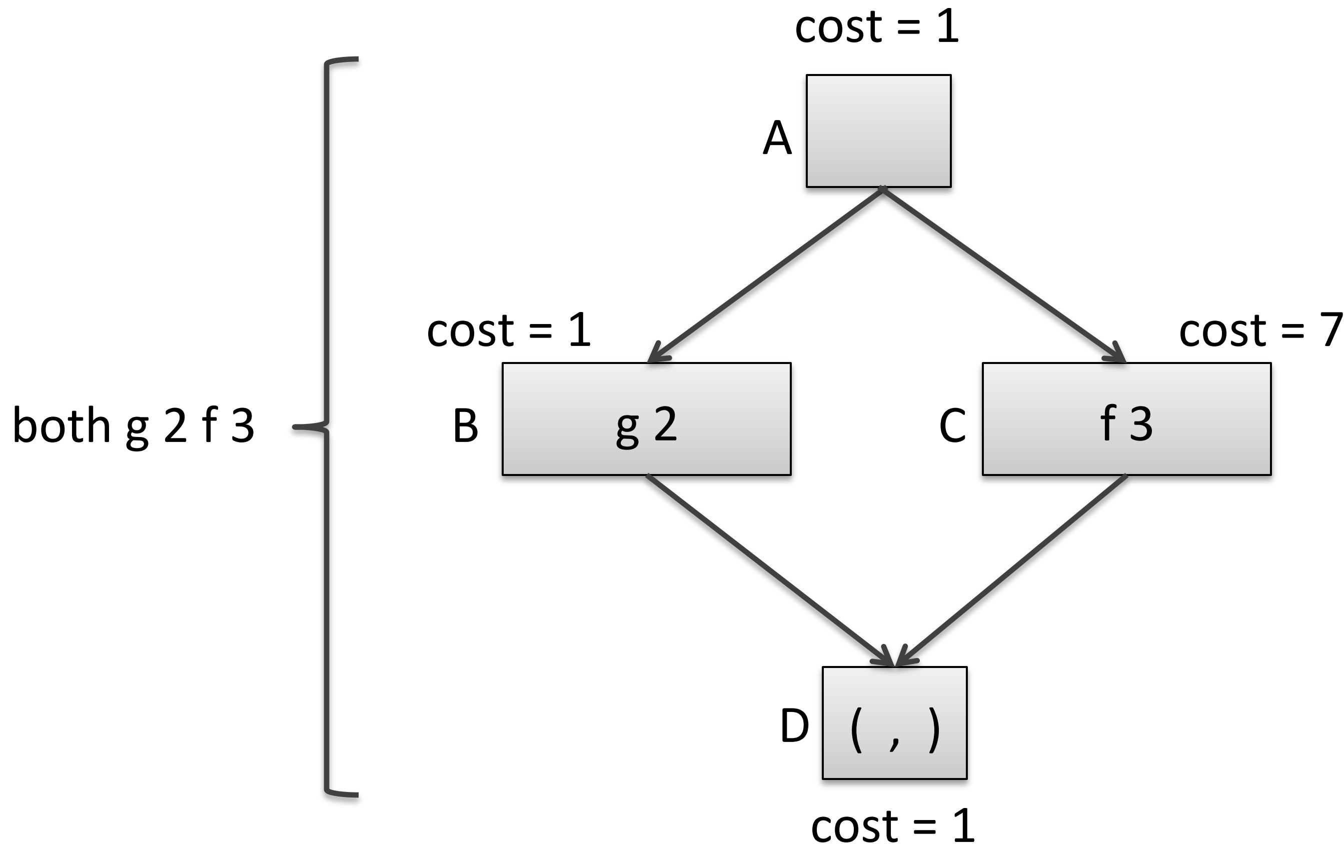
Suppose we have 3 processors. How much time does this computation take?

Visualizing Computational Costs



Suppose we have 3 processors. How much time does this computation take?
Schedule A-BC-D: 1 + max(1,7) + 1 = 9

Visualizing Computational Costs



Suppose we have infinite processors. How much time does this computation take?
Schedule A-BC-D: 1 + max(1,7) + 1 = 9

Work and Span

- Understanding the complexity of a parallel program is a little more complex than a sequential program
  - the number of processors has a significant effect

- One way to *approximate* the cost is to consider a parallel algorithm independently of the machine it runs on is to consider *two* metrics:
  - Work: The cost of executing a program with just 1 processor.
  - Span: The cost of executing a program with an infinite number of processors

- Always good to minimize work
  - Every instruction executed consumes energy
  - Minimize span as a second consideration
  - Communication costs are also crucial (we are ignoring them)

Parallelism

The parallelism of an algorithm is an estimate of the maximum number of processors an algorithm can profit from.

- parallelism = work / span

If work = span then parallelism = 1.

- We can only use 1 processor
- It's a sequential algorithm

If span = ½ work then parallelism = 2

- We can use up to 2 processors

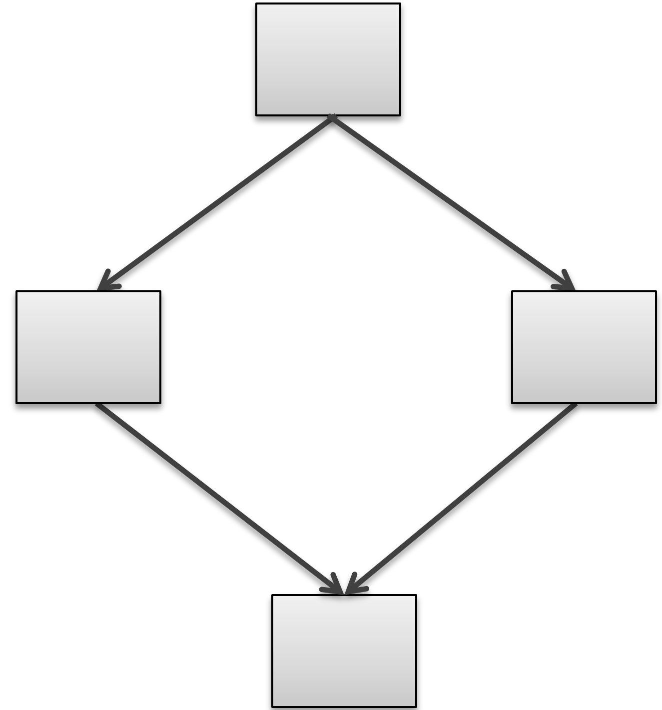
If work = 100, span = 1

- All operations are independent & can be executed in parallel
- We can use up to 100 processors

Series-Parallel Graphs

one operation

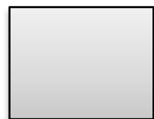
two operations
in sequence:
e1; e2



two operations
in parallel:
both (fun _-> e1)
(fun _ -> e2)

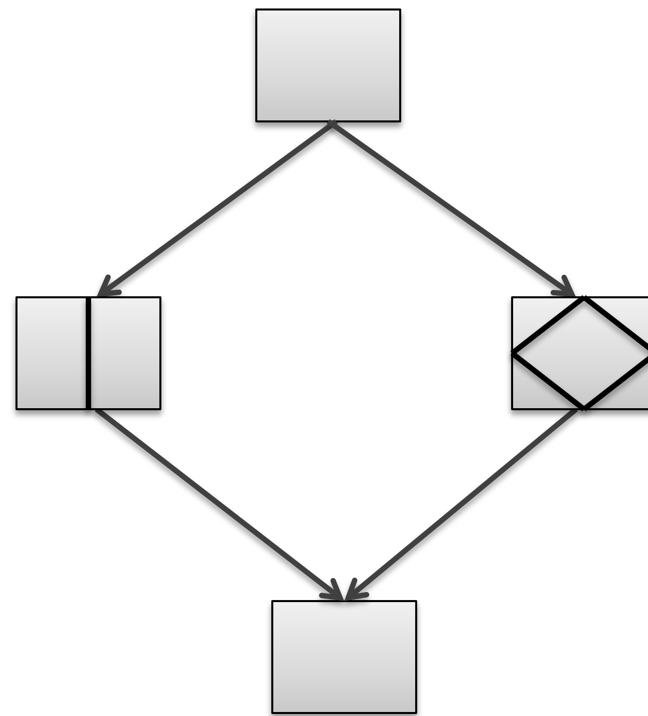
Series-parallel graphs arise from execution of functional programs with parallel pairs. Also known as well-structured, nested parallelism.

Series-Parallel Graphs Compose



one operation

two graphs
in sequence

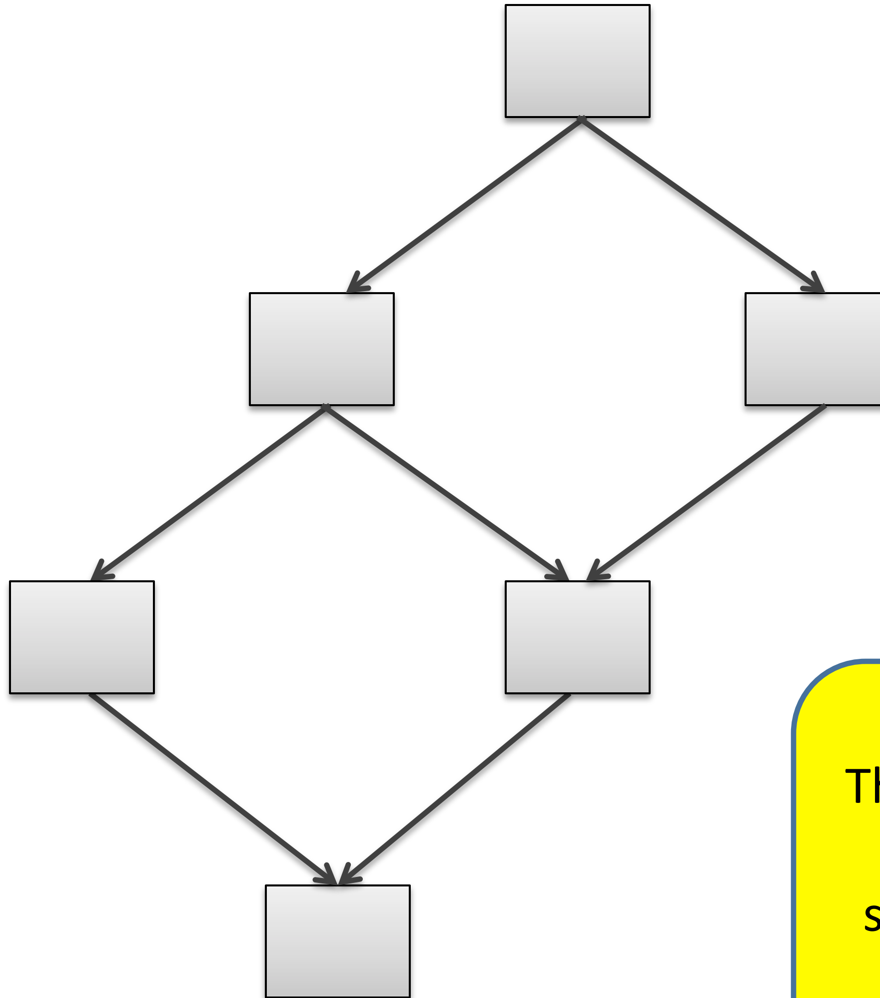


two graphs
in parallel

In general, a series-parallel graph has a source and a sink and is:

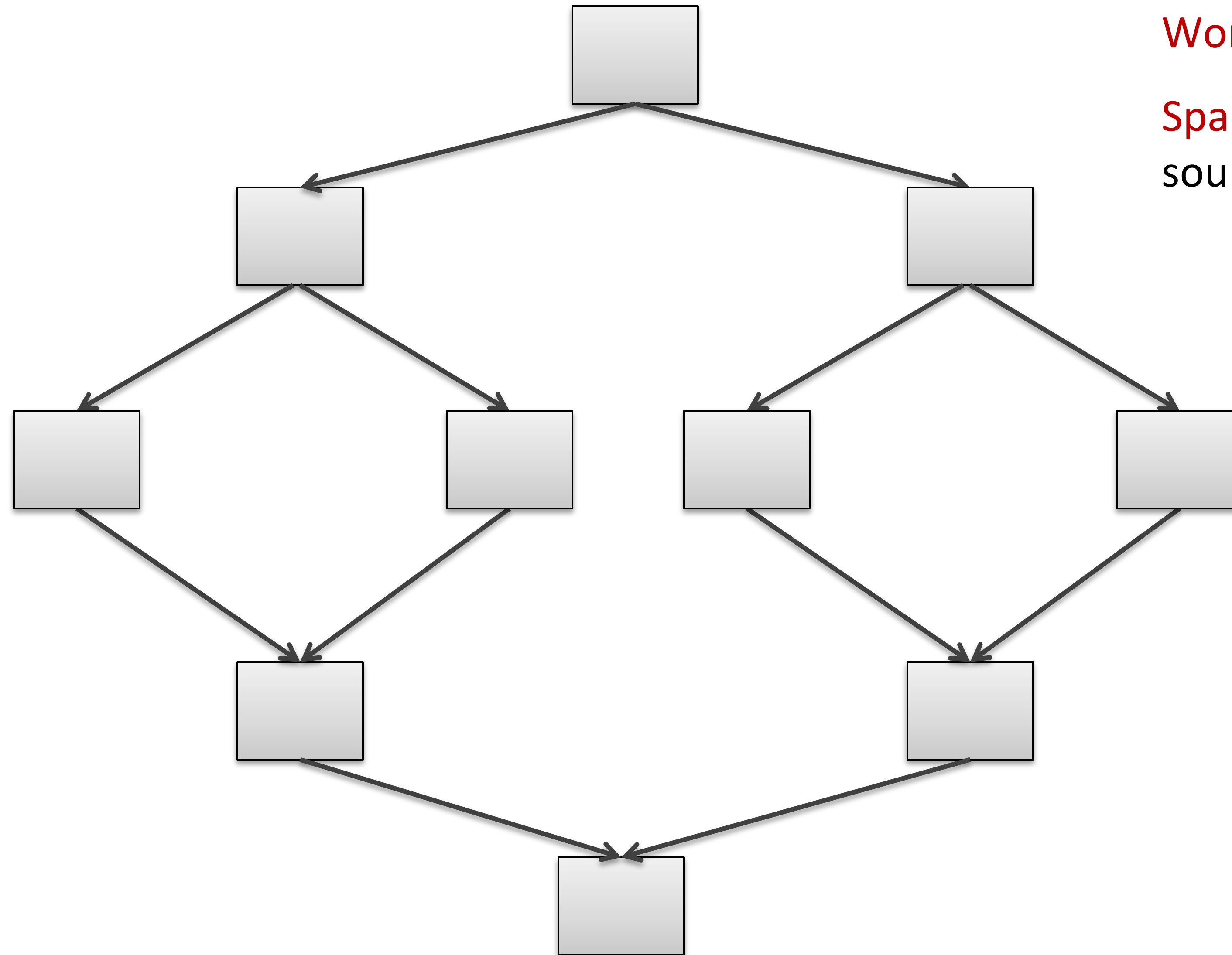
- a single node, or
- two series-parallel graphs in sequence, or
- two series-parallel graphs in parallel

Not a Series-Parallel Graph



However:
The results about greedy schedulers (next few slides) do apply to DAG schedules as well as series-parallel schedules!

Work and Span of Acyclic Graphs

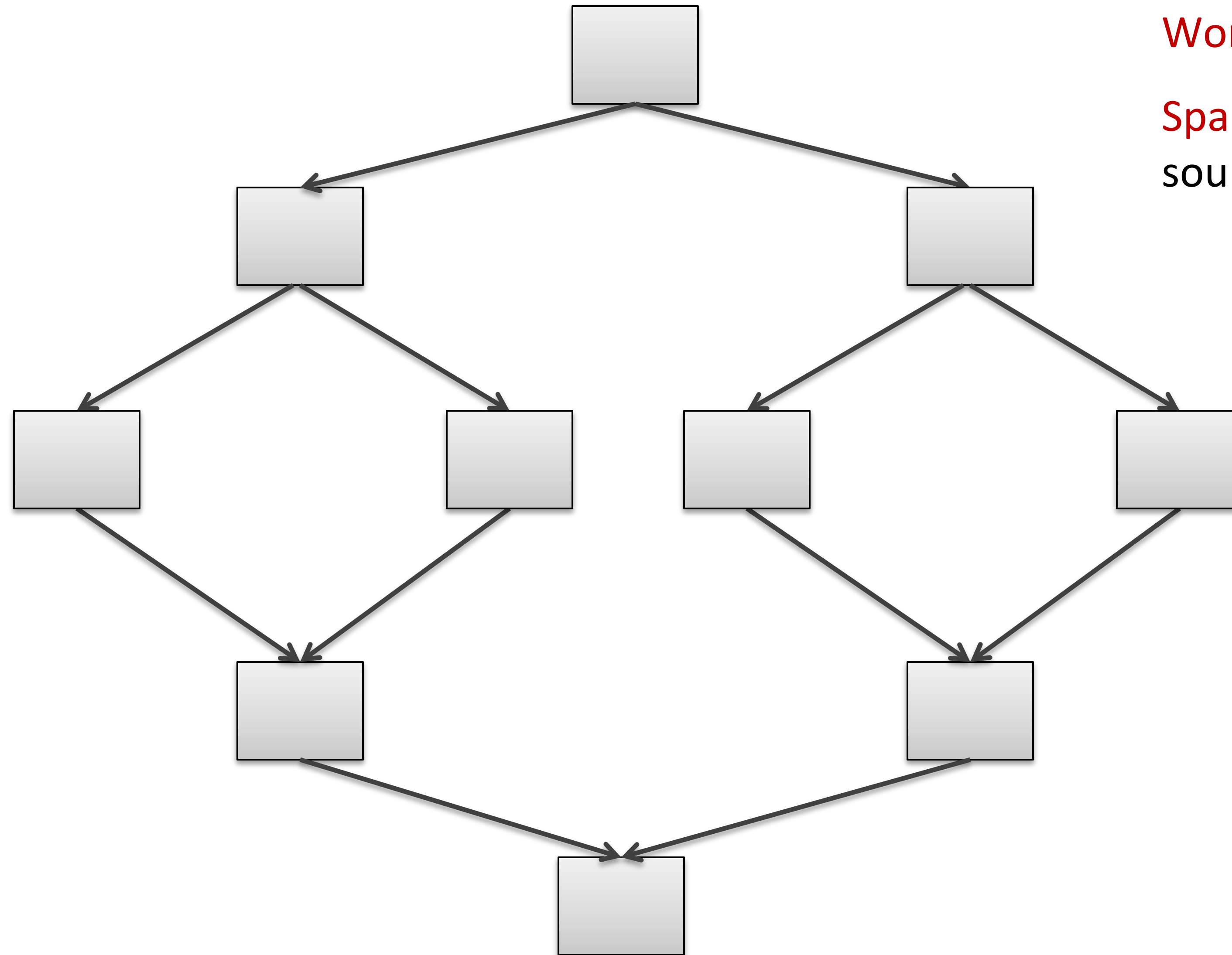


Let's assume each node costs 1.

Work: sum the nodes.

Span: longest path from source to sink.

Work and Span of Acyclic Graphs



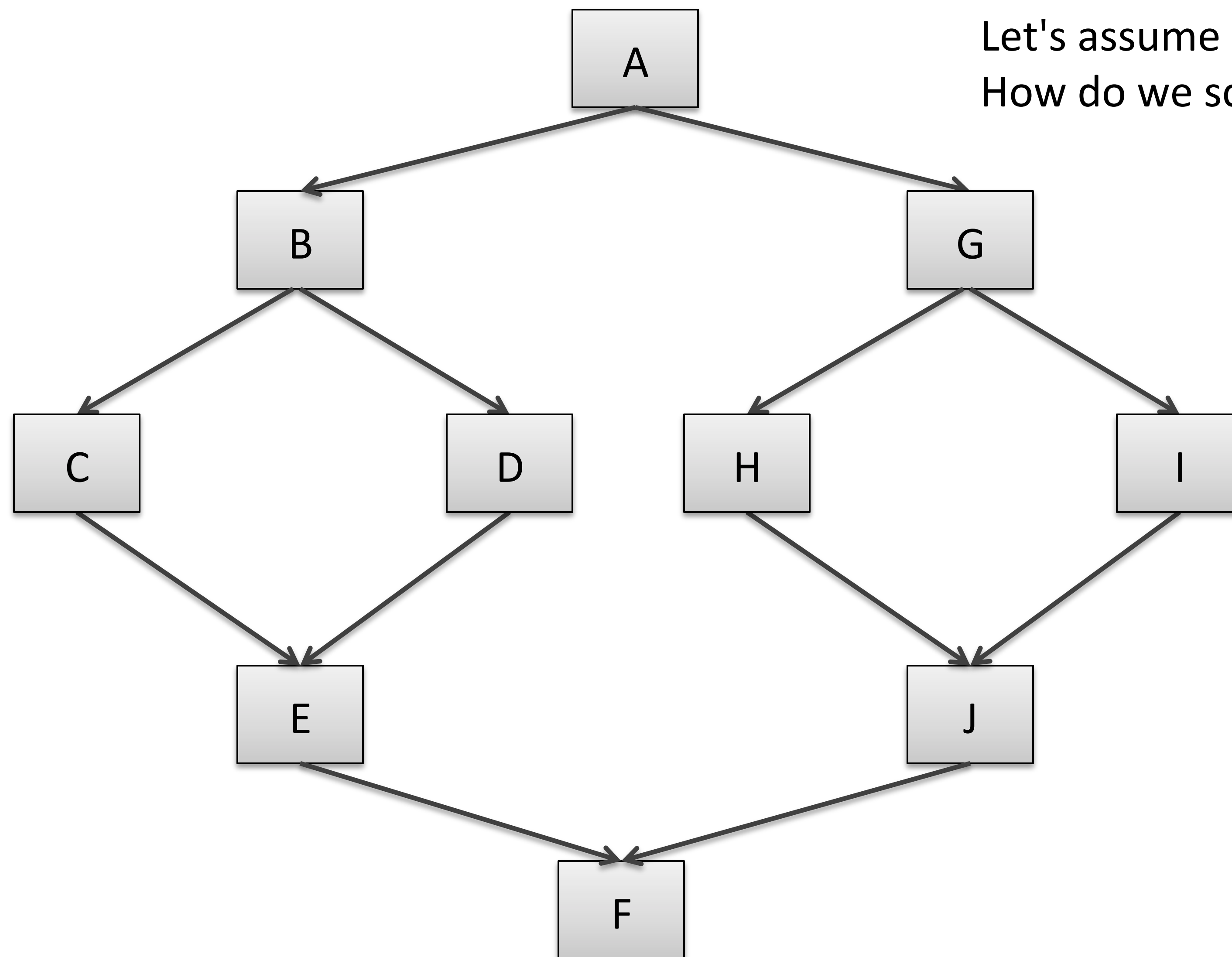
Let's assume each node costs 1.

Work: sum the nodes.

Span: longest path from source to sink.

work = 10
span = 5

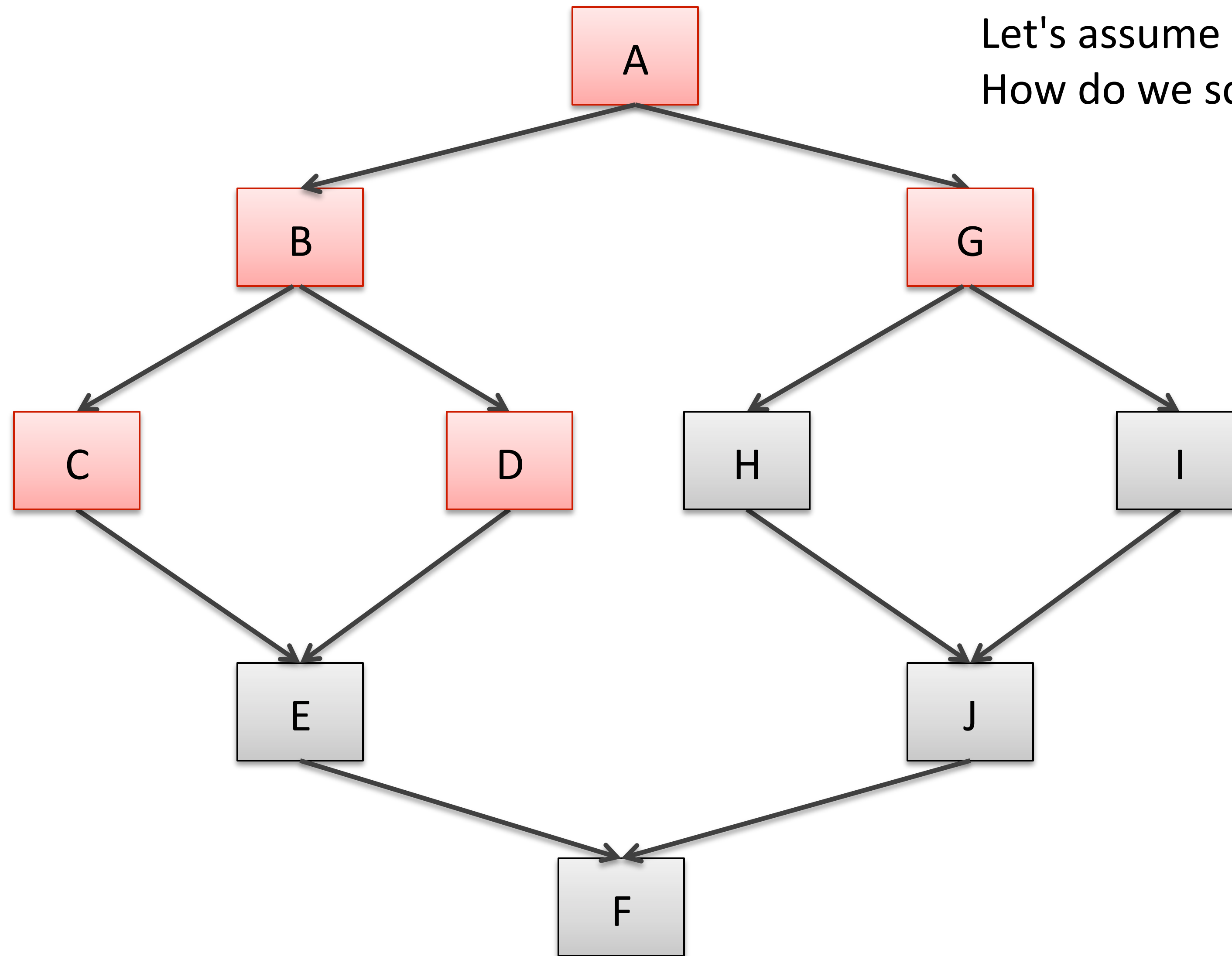
Scheduling



Let's assume each node costs 1.

Let's assume we have 2 processors.
How do we schedule computation?

Scheduling

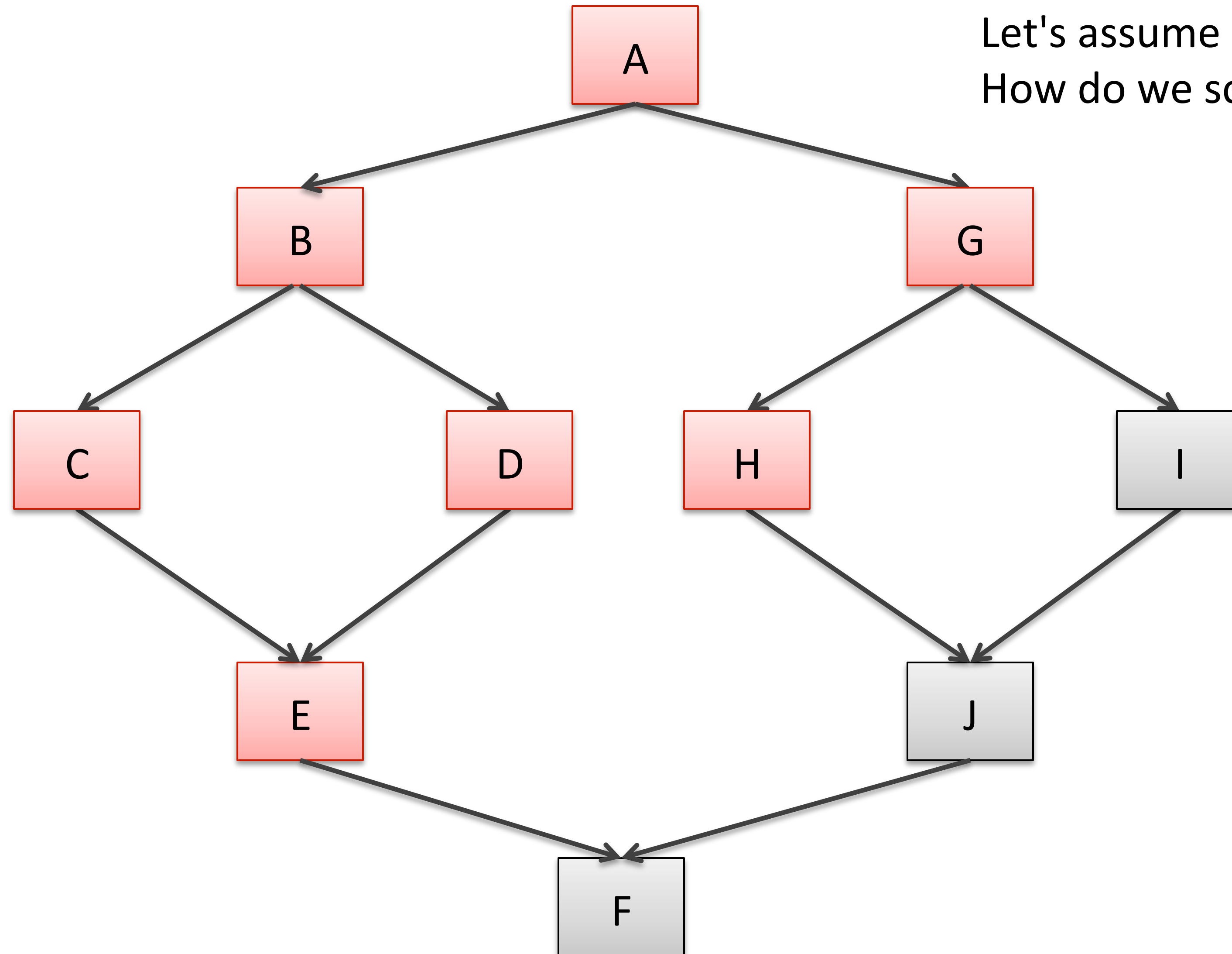


Let's assume each node costs 1.

Let's assume we have 2 processors.
How do we schedule computation?

Option 1:
A
B G
C D

Scheduling

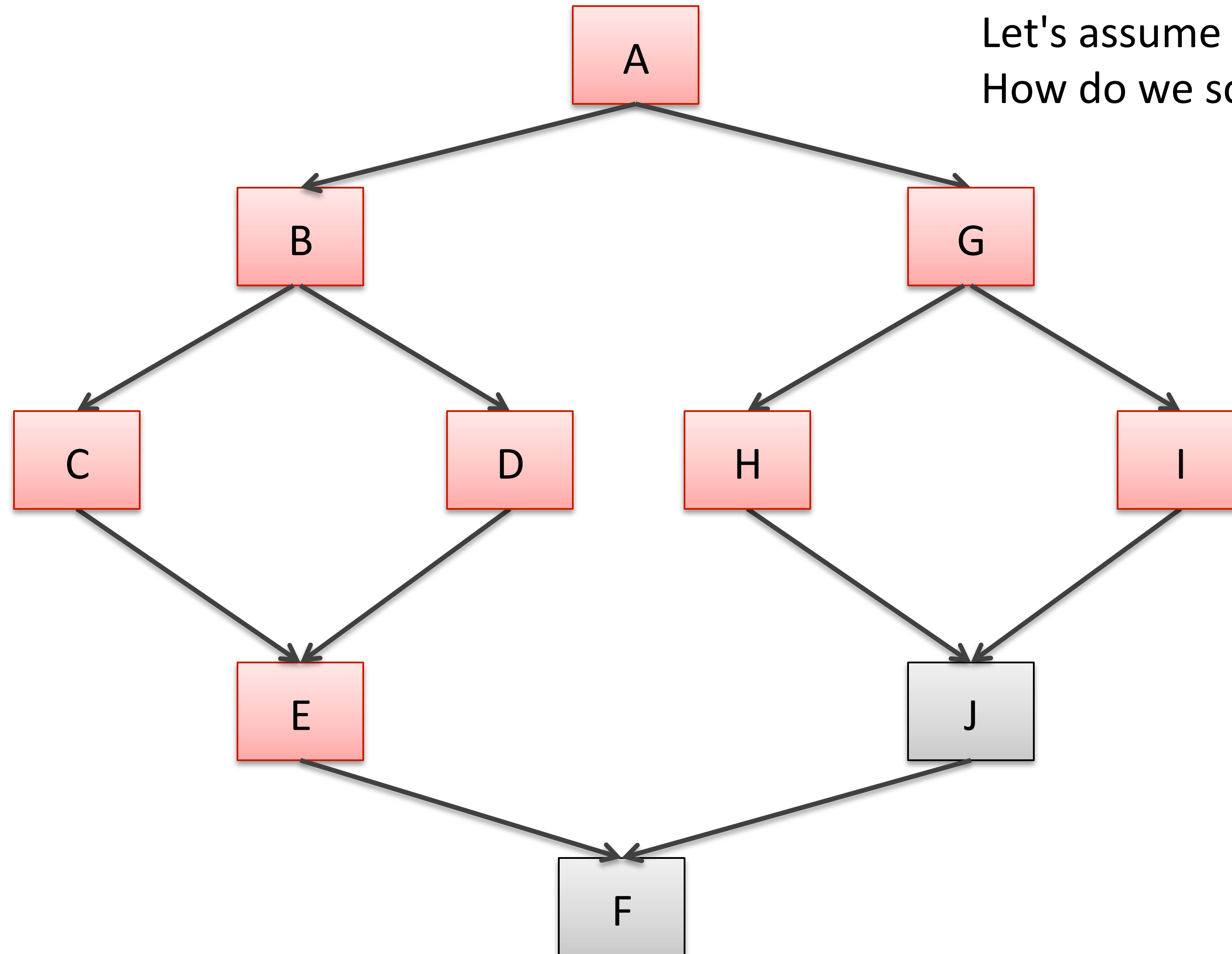


Let's assume each node costs 1.

Let's assume we have 2 processors.
How do we schedule computation?

Option 1:
A
B G
C D
E H

Scheduling

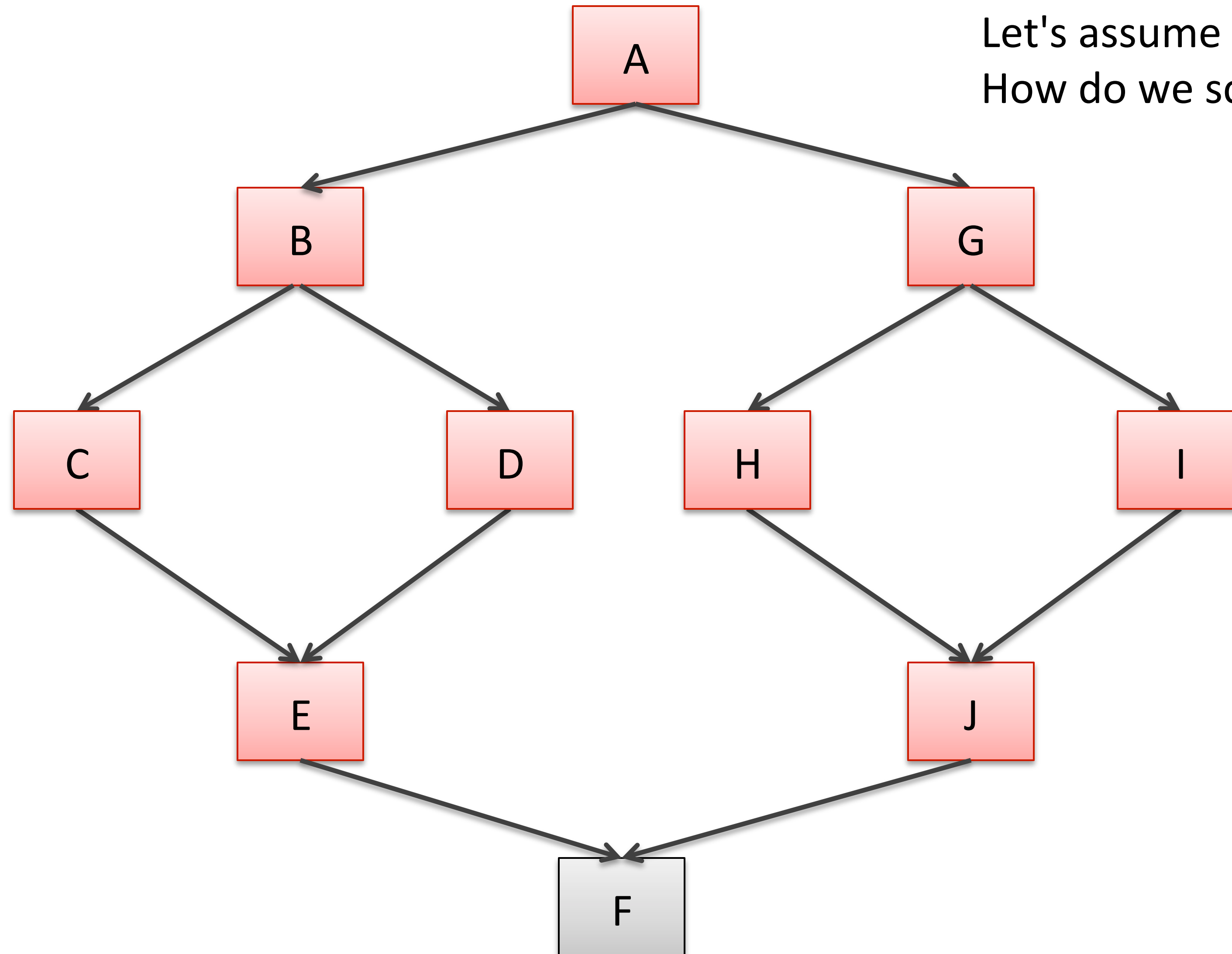


Let's assume each node costs 1.

Let's assume we have 2 processors.
How do we schedule computation?

Option 1:
A
B G
C D
E H
I

Scheduling

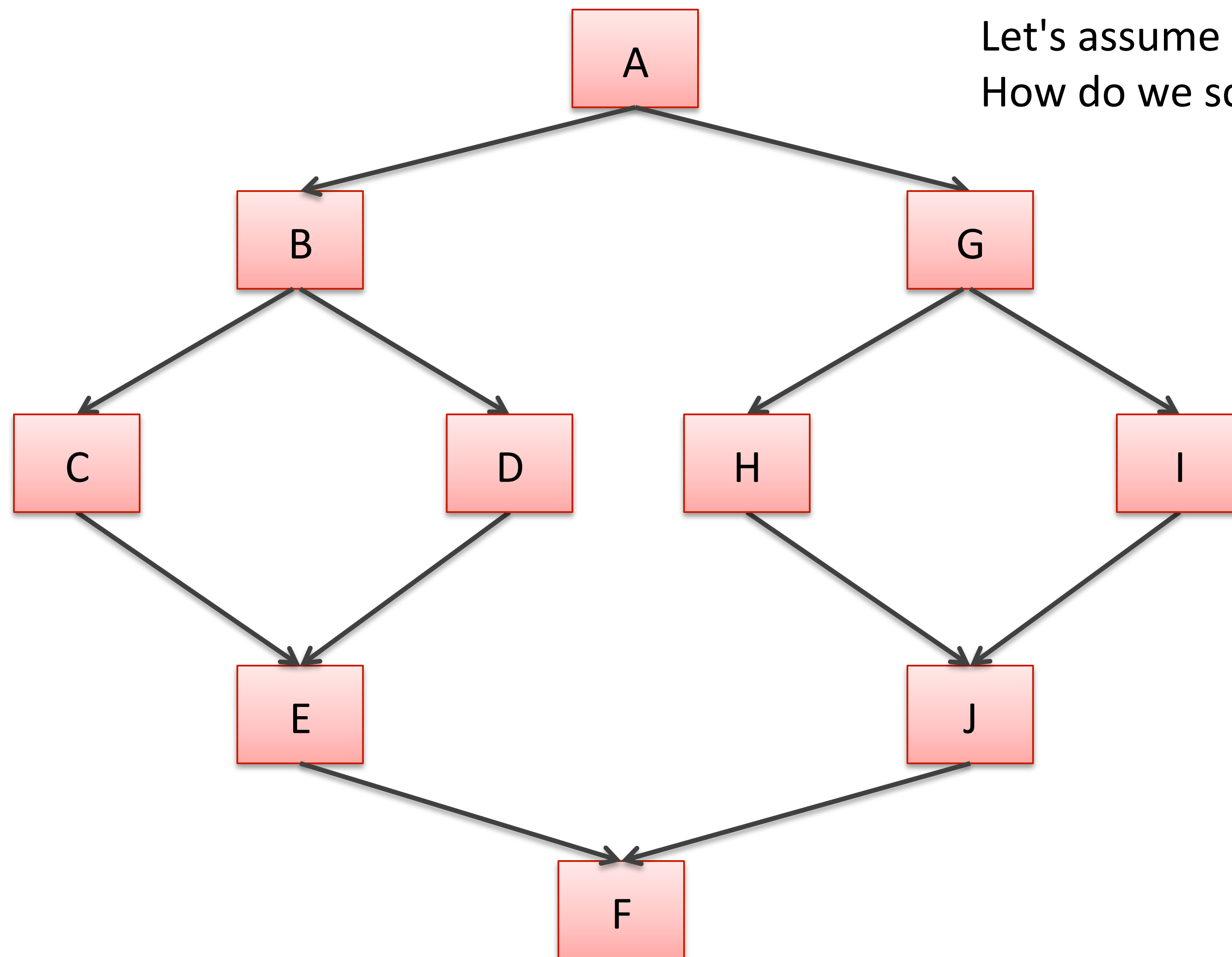


Let's assume each node costs 1.

Let's assume we have 2 processors.
How do we schedule computation?

Option 1:
A
B G
C D
E H
I
J

Scheduling

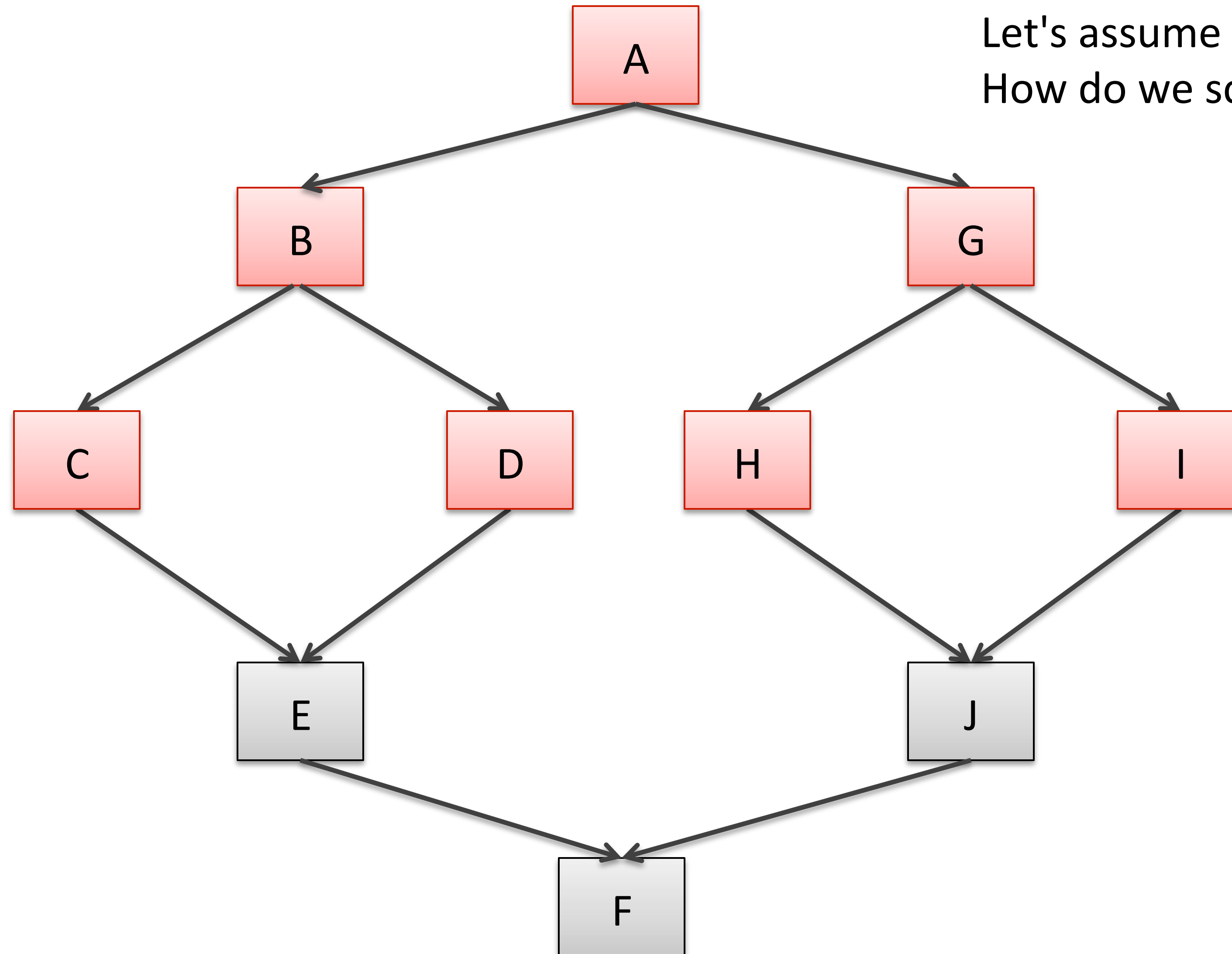


Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?

Option 1:
A
B G
C D
E H
I
J
F

Scheduling

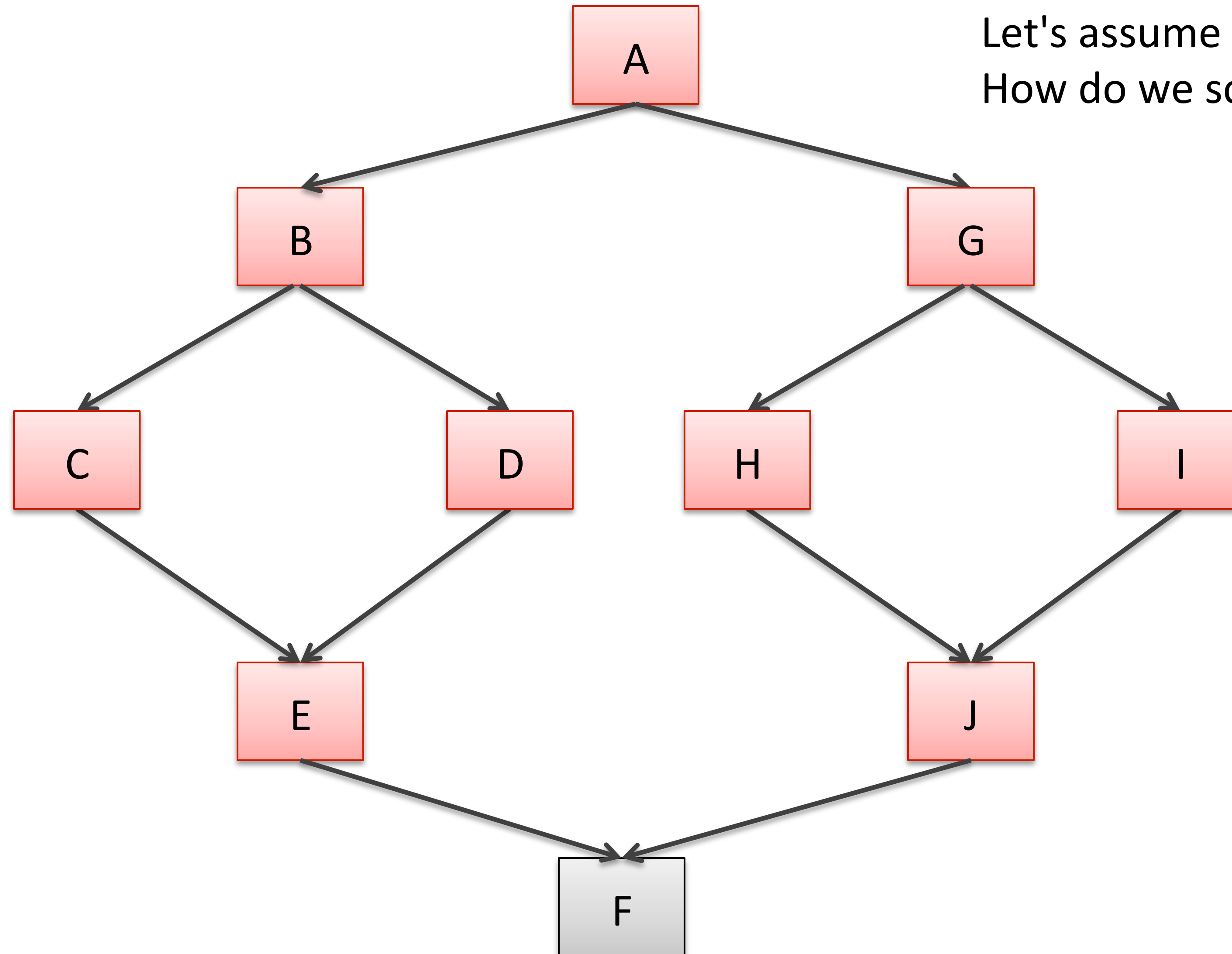


Let's assume each node costs 1.

Let's assume we have 2 processors.
How do we schedule computation?

Option 1:
A
B G
C D
~~E H~~ H I
~~I~~
~~J~~
~~F~~

Scheduling

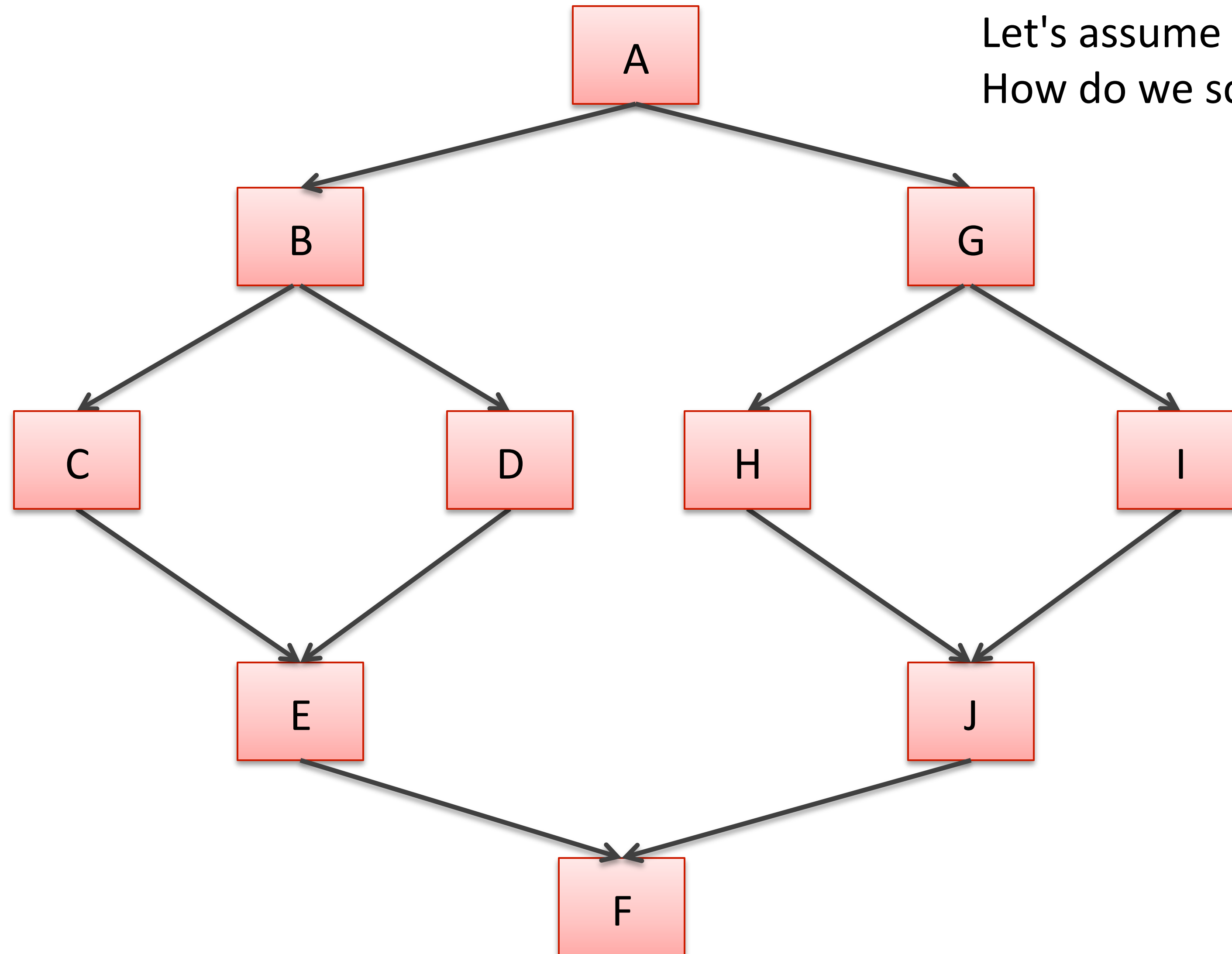


Let's assume each node costs 1.

Let's assume we have 2 processors.
How do we schedule computation?

Option 1:
A
B G
C D
~~E H~~ H I
~~I~~ E J
~~J~~
~~F~~

Scheduling

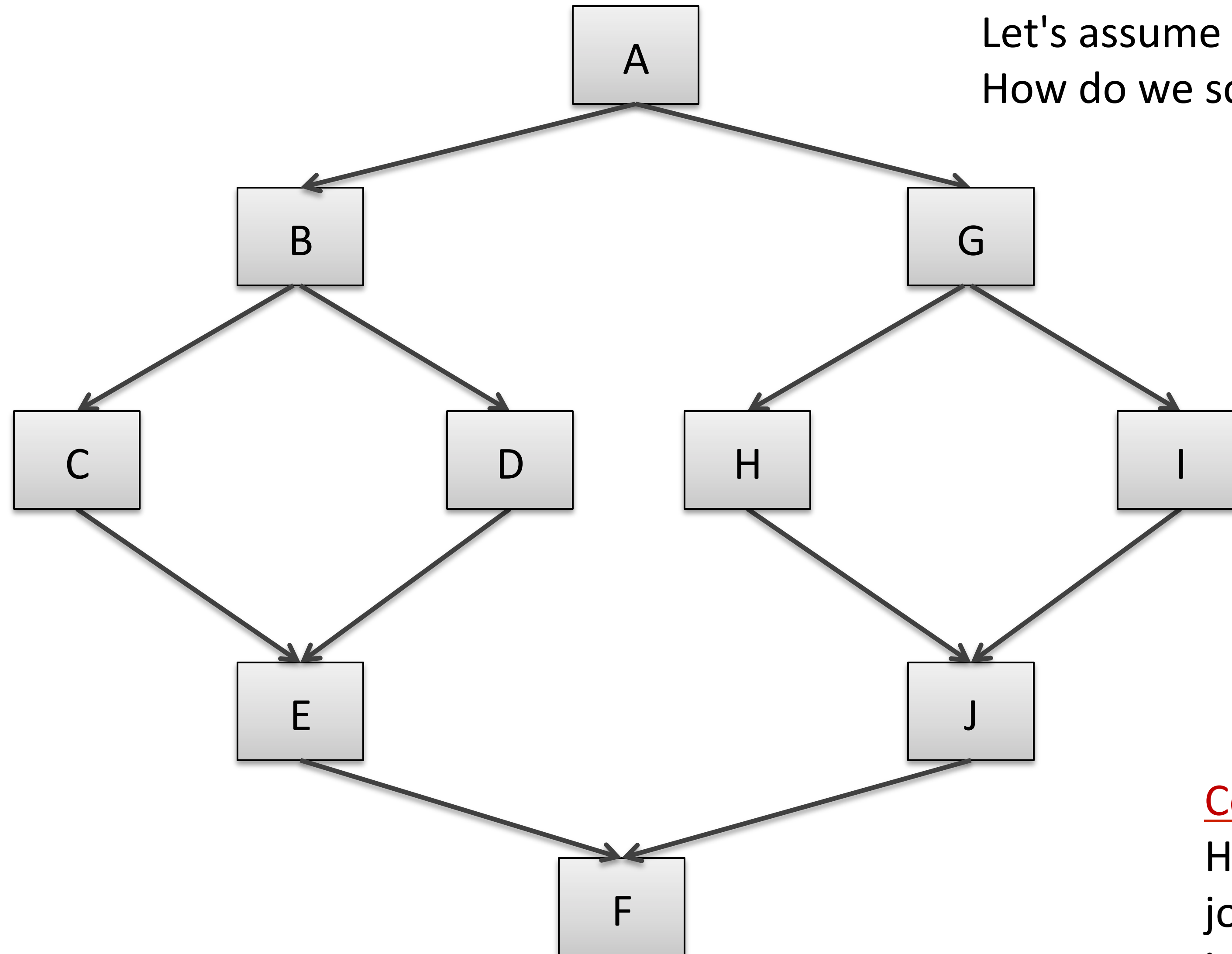


Let's assume each node costs 1.

Let's assume we have 2 processors.
How do we schedule computation?

Option 1:
A
B G
C D
~~E H~~ H I
~~I~~ E J
~~J~~ F
~~F~~

Scheduling



Let's assume each node costs 1.

Let's assume we have 2 processors.
How do we schedule computation?

Option 1:

A
B G
C D
~~E H~~ H I
~~I~~ E J
~~J~~ F
~~F~~

Conclusion:

How you schedule jobs can have an impact on performance

Greedy Schedulers

- Greedy schedulers will schedule some task to a processor as soon as that processor is free.
  - Doesn't sound so smart!

- Properties (for p processors):
  - $T(p) < work/p + span$
    - won't be worse than dividing up the data perfectly between processors, except for the last little bit, which causes you to add the span on top of the perfect division
  - $T(p) >= max(work/p, span)$
    - can't do better than perfect division between processors (work/p)
    - can't be faster than span

Greedy Schedulers

Properties (for p processors):

max(work/p, span) <= T(p) < work/p + span

Consequences:

- as span gets small relative to work/p
  - work/p + span ==> work/p
  - max(work/p, span) ==> work/p
  - so T(p) ==> work/p -- greedy schedulers converge to the optimum!
- if span approaches the work
  - work/p + span ==> span
  - max(work/p, span) ==> span
  - so T(p) ==> span – greedy schedulers converge to the optimum!

SUMMARY

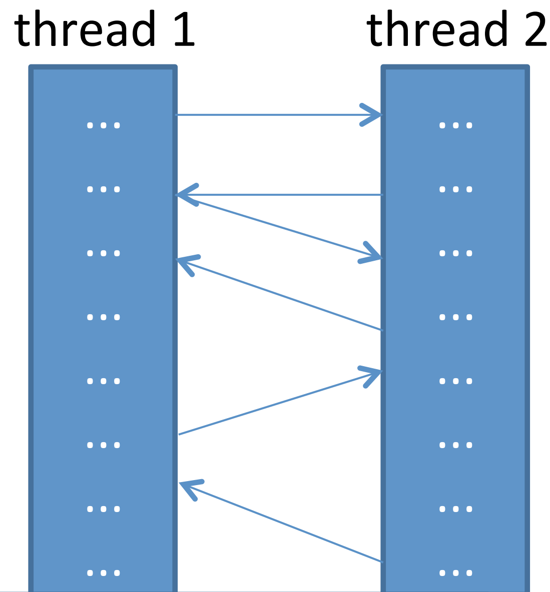
Programming with mutation, threads and locks

Reasoning about the correctness of *pure* parallel programs that include futures is easy -- no harder than ordinary, sequential programs. (Reasoning about their performance may be harder.)

Reasoning about shared variables and synchronization is *hard* in general, but *futures* are a *discipline* for getting it right.

Much of programming-language design is the art of finding good disciplines in which it's harder* to write bad programs.

Even aside from PL design, the same is true of software engineering with Abstract Data Types: if you engineer *disciplines* into your interfaces, it is harder for the user to get it wrong.



*but somebody will always find a way…