

Parallel Sequences

COS 326

Andrew Appel

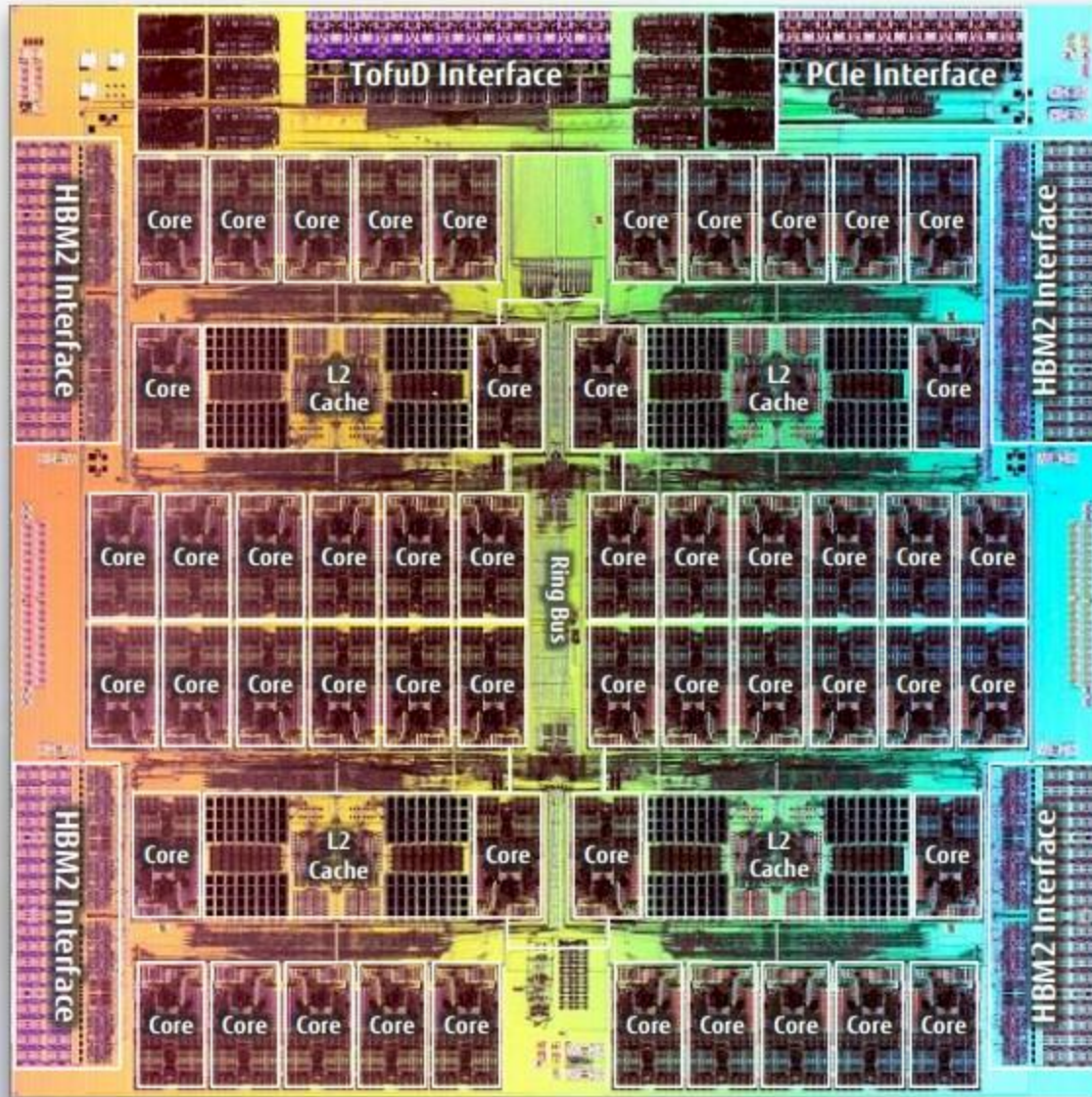
Princeton University

Credits:

Dan Grossman, U.Wash.

Guy Blelloch, Bob Harper (CMU), Dan Licata (Wesleyan)

Parallel Programming



Fujitsu A64FX (48 ARM cores)

Programming with shared mutable data is very hard!

How can we leverage

- pure functions
- immutable data
- function composition

to write large-scale parallel programs?

What if you had a really big job to do?

Example: Create an index of every web page on the planet.

- Google does that regularly!
- There are billions of them!

Example: Search facebook for a friend or twitter for a tweet

To get big jobs done, we typically need 1000s of computers, but:

- how do we distribute work across all those computers?
- you definitely can't use shared-memory parallelism because the computers don't share memory!
- when you use 1 computer, you just hope it doesn't fail. If it does, you go to the store, buy a new one and restart the job.
- when you use 1000s of computers at a time, failures become the norm. what to do when 1 of 1000 computers fail? Start over?

Big Jobs ---> Better Abstractions

Need high-level interfaces to shield application programmers from the complex details. Complex implementations solve the problems of distribution, fault tolerance and performance.

Common abstraction: Parallel collections

Example collections: sets, tables, dictionaries, sequences

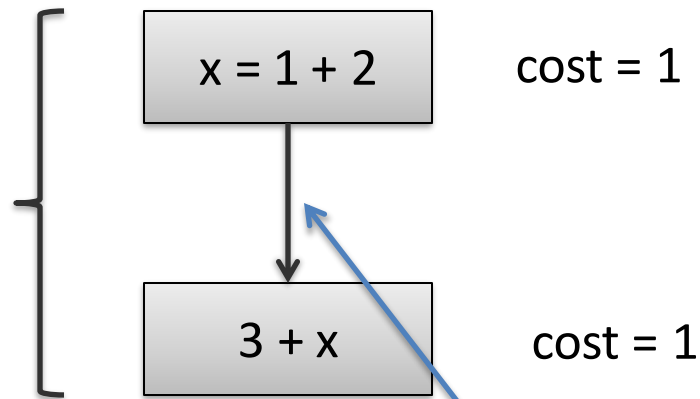
Example bulk operations: create, map, reduce, join, filter



COMPLEXITY OF PARALLEL ALGORITHMS

Visualizing Computational Costs

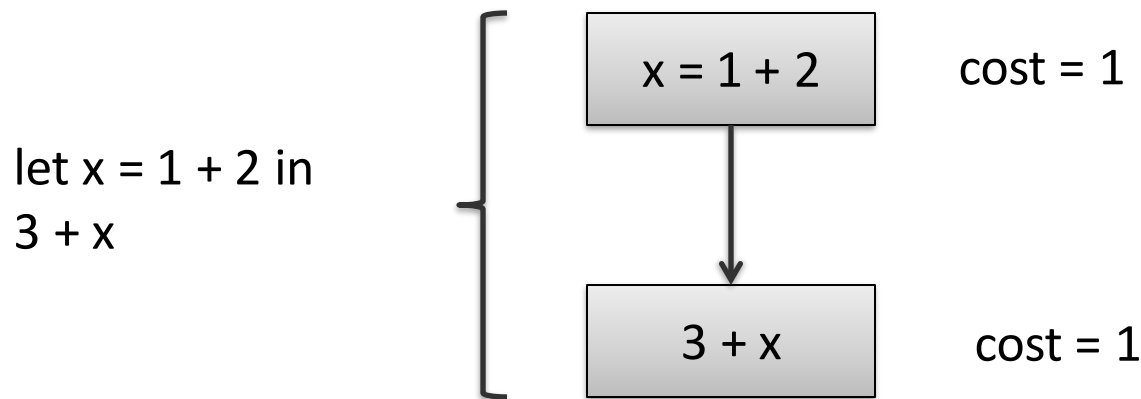
let $x = 1 + 2$ in
 $3 + x$



dependence:

$x = 1 + 2$ *happens before* $3 + x$

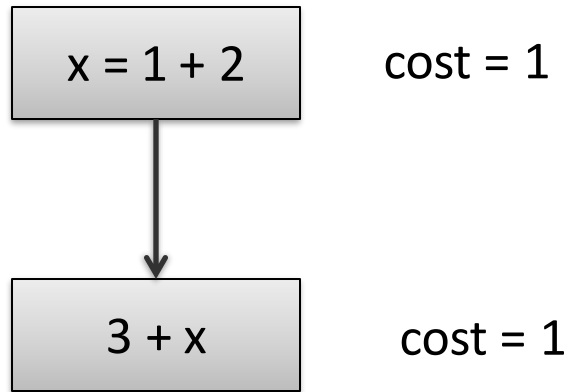
Visualizing Computational Costs



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

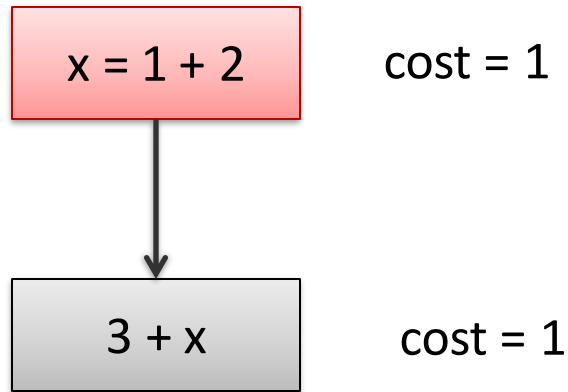
step 1:
execute first block



Cost so far: 0

Visualizing Computational Costs

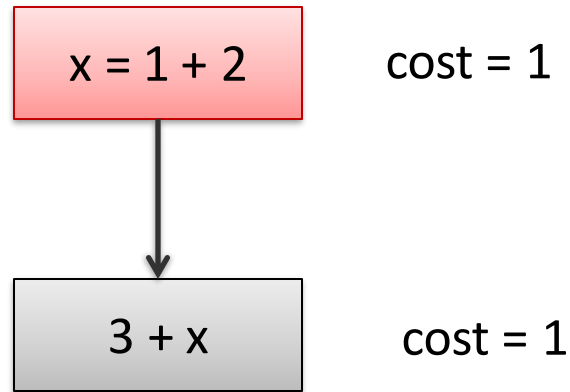
step 1:
execute first block



Cost so far: 1

Visualizing Computational Costs

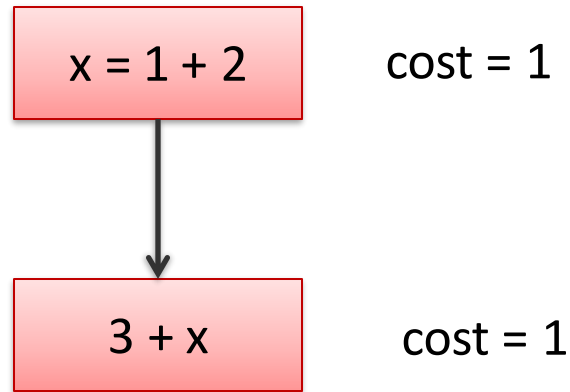
step 2:
execute second block
because all of its
predecessors have
been completed



Cost so far: 1

Visualizing Computational Costs

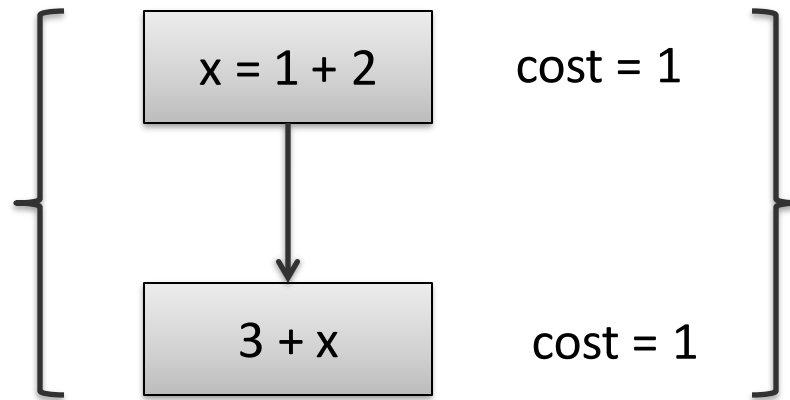
step 2:
execute second block
because all of its
predecessors have
been completed



Cost so far: 1 + 1

Visualizing Computational Costs

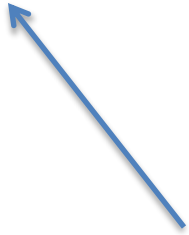
let $x = 1 + 2$ in
 $3 + x$



total cost
 $= 1 + 1$
 $= 2$

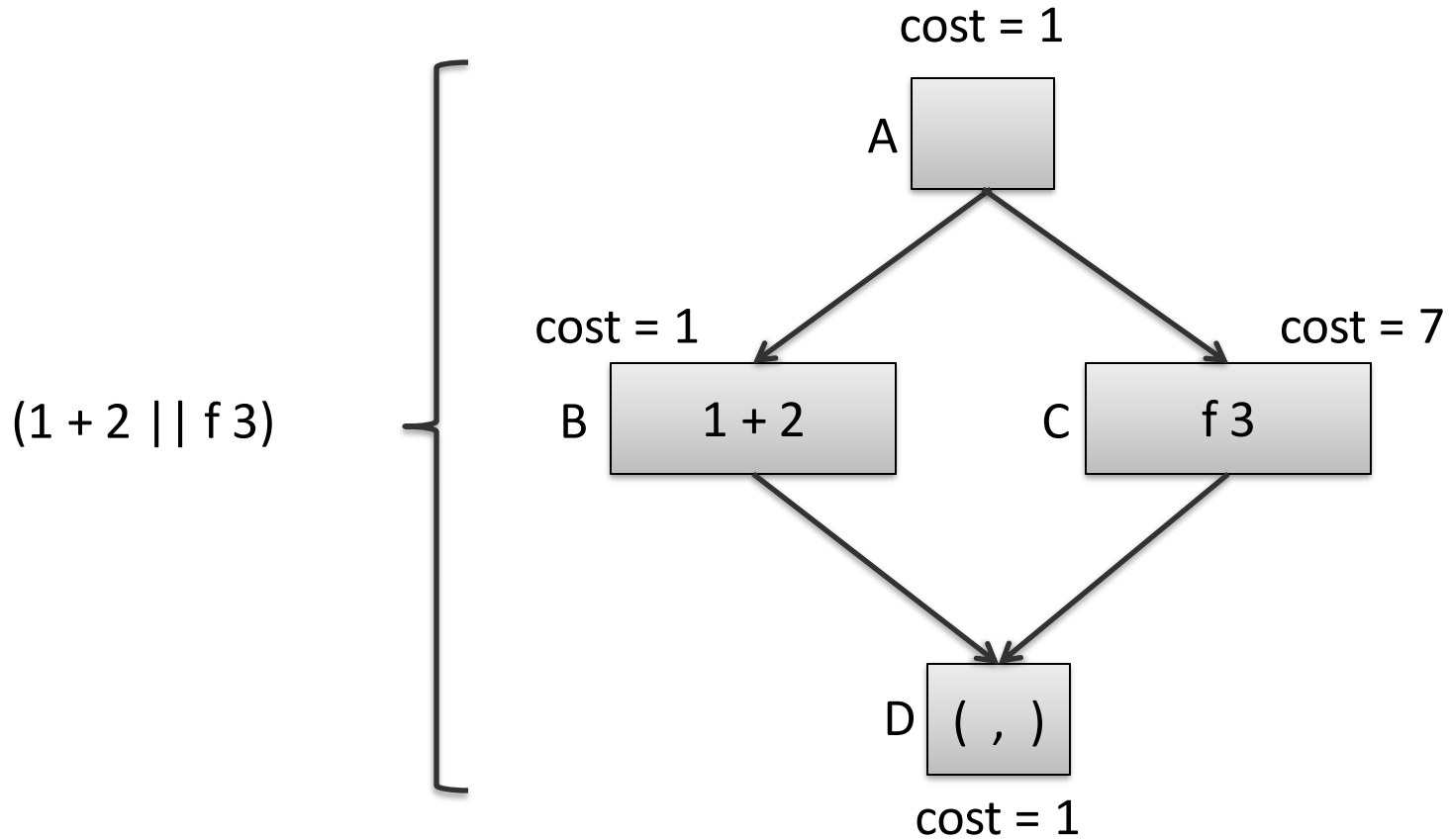
Visualizing Computational Costs

`(1 + 2 || f 3)`

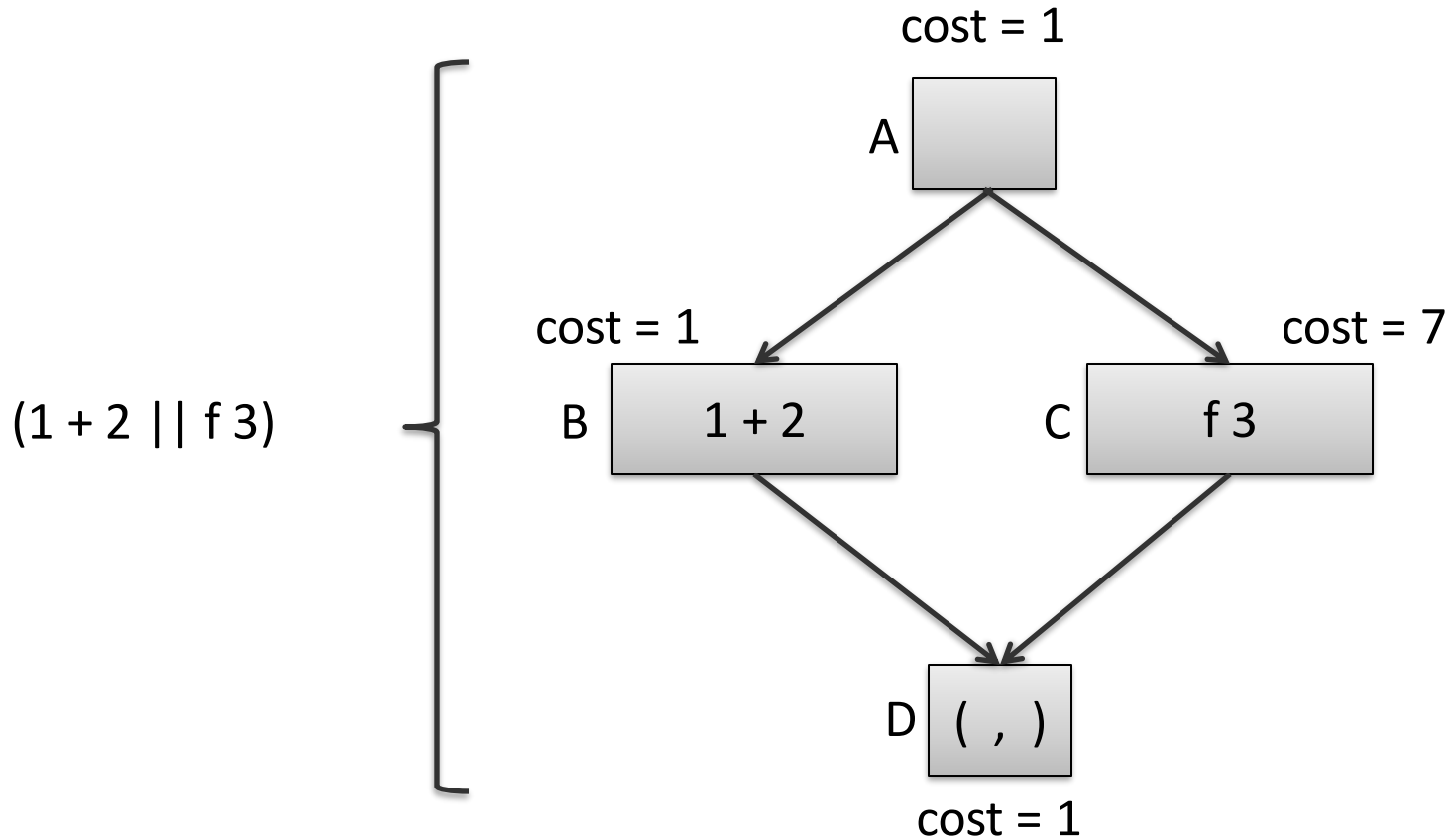


parallel pair:
compute both left and right-hand sides independently
return pair of values
(easy to implement using futures)

Visualizing Computational Costs

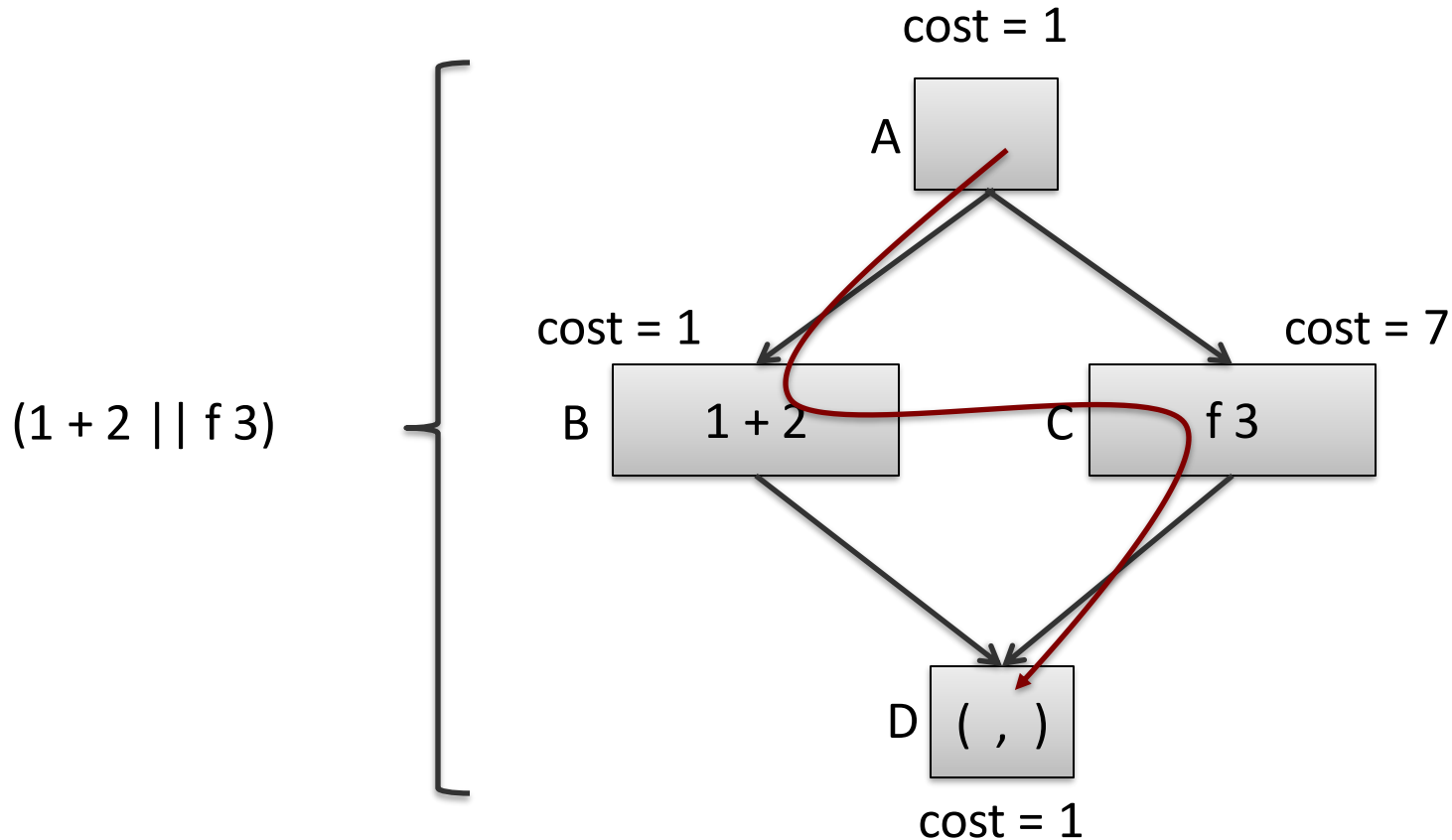


Visualizing Computational Costs



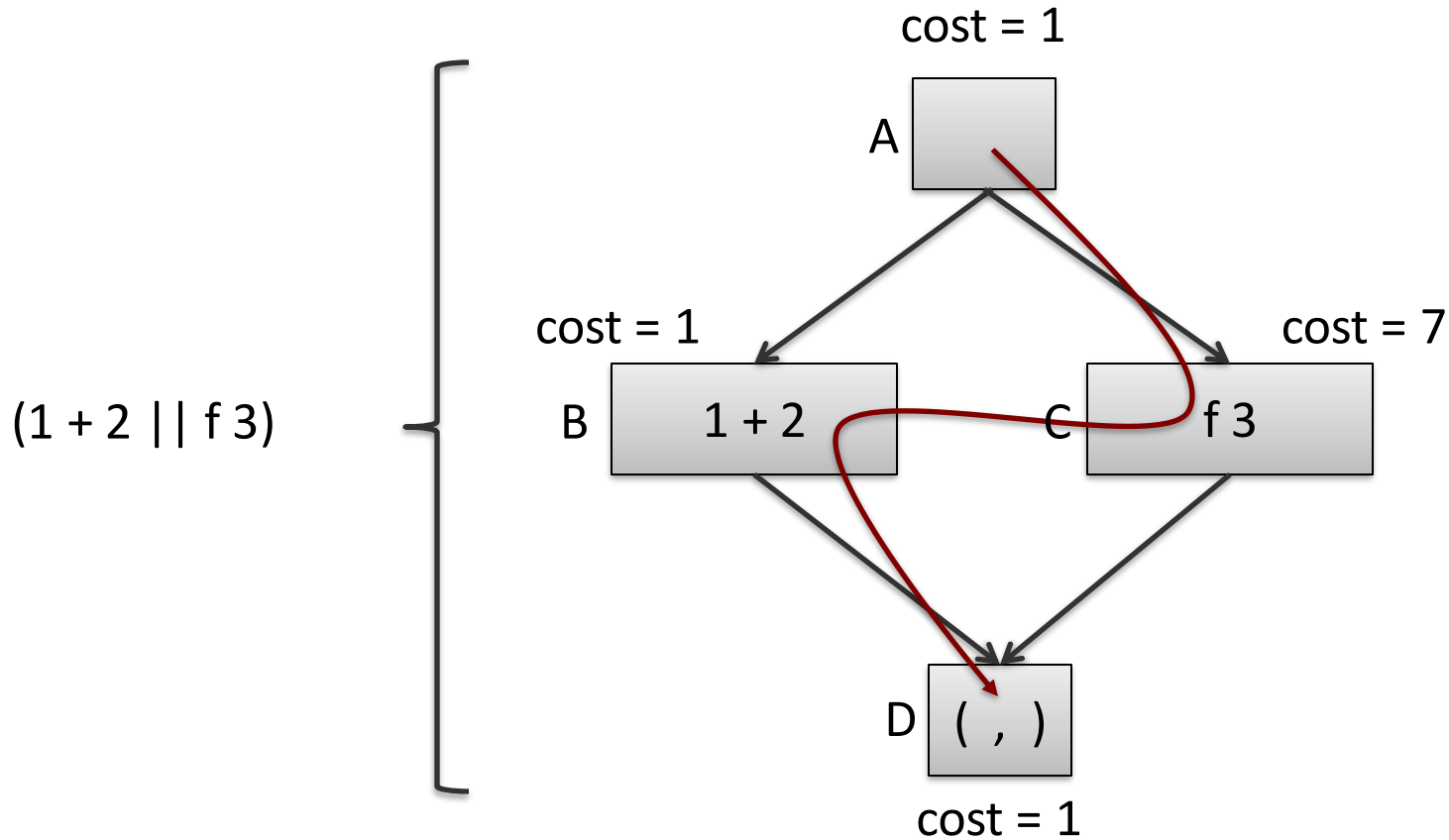
Suppose we have 1 processor. How much time does this computation take?

Visualizing Computational Costs



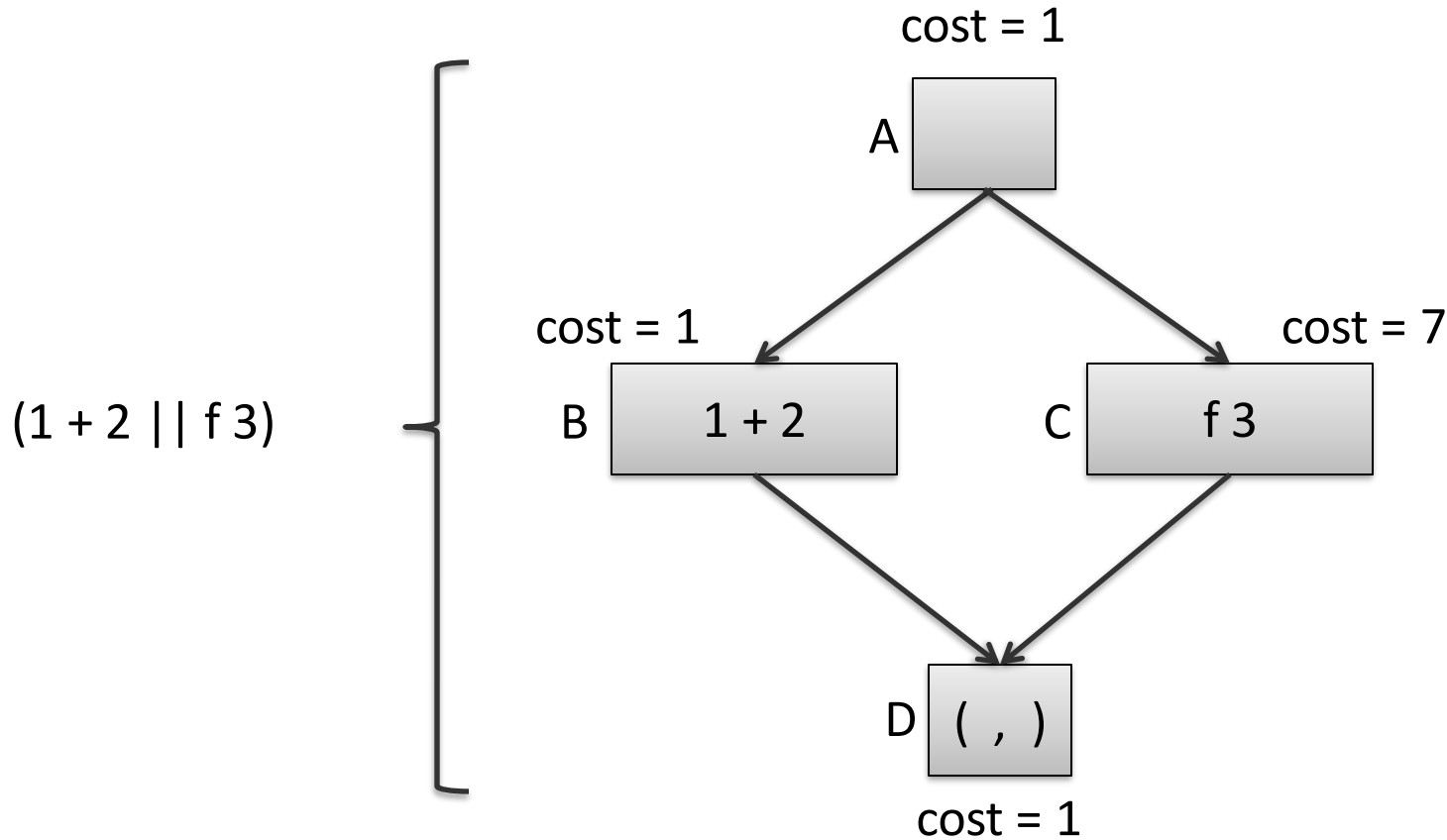
Suppose we have 1 processor. How much time does this computation take?
Schedule A-B-C-D: $1 + 1 + 7 + 1$

Visualizing Computational Costs



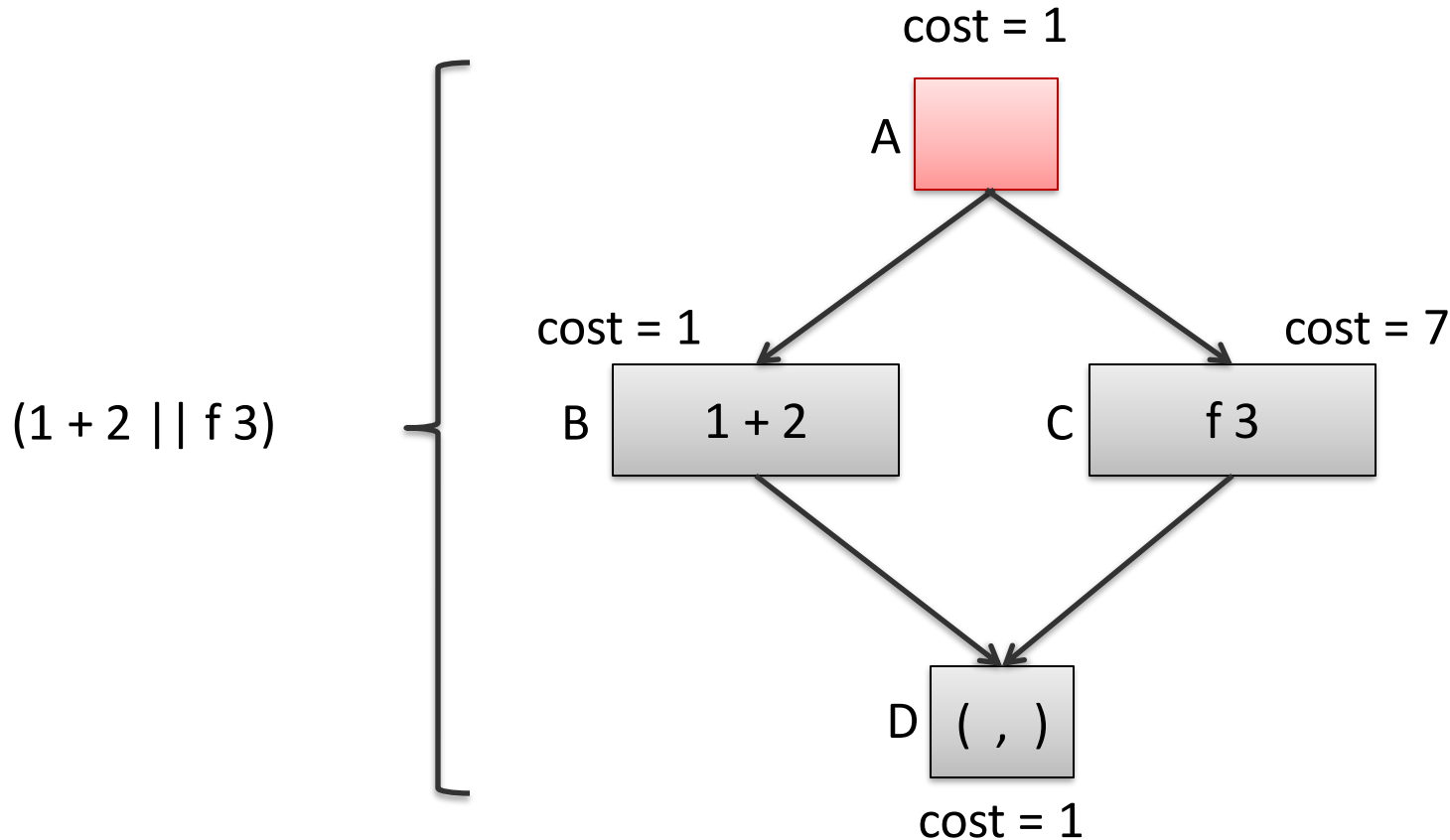
Suppose we have 1 processor. How much time does this computation take?
Schedule A-C-B-D: 1 + 1 + 7 + 1

Visualizing Computational Costs



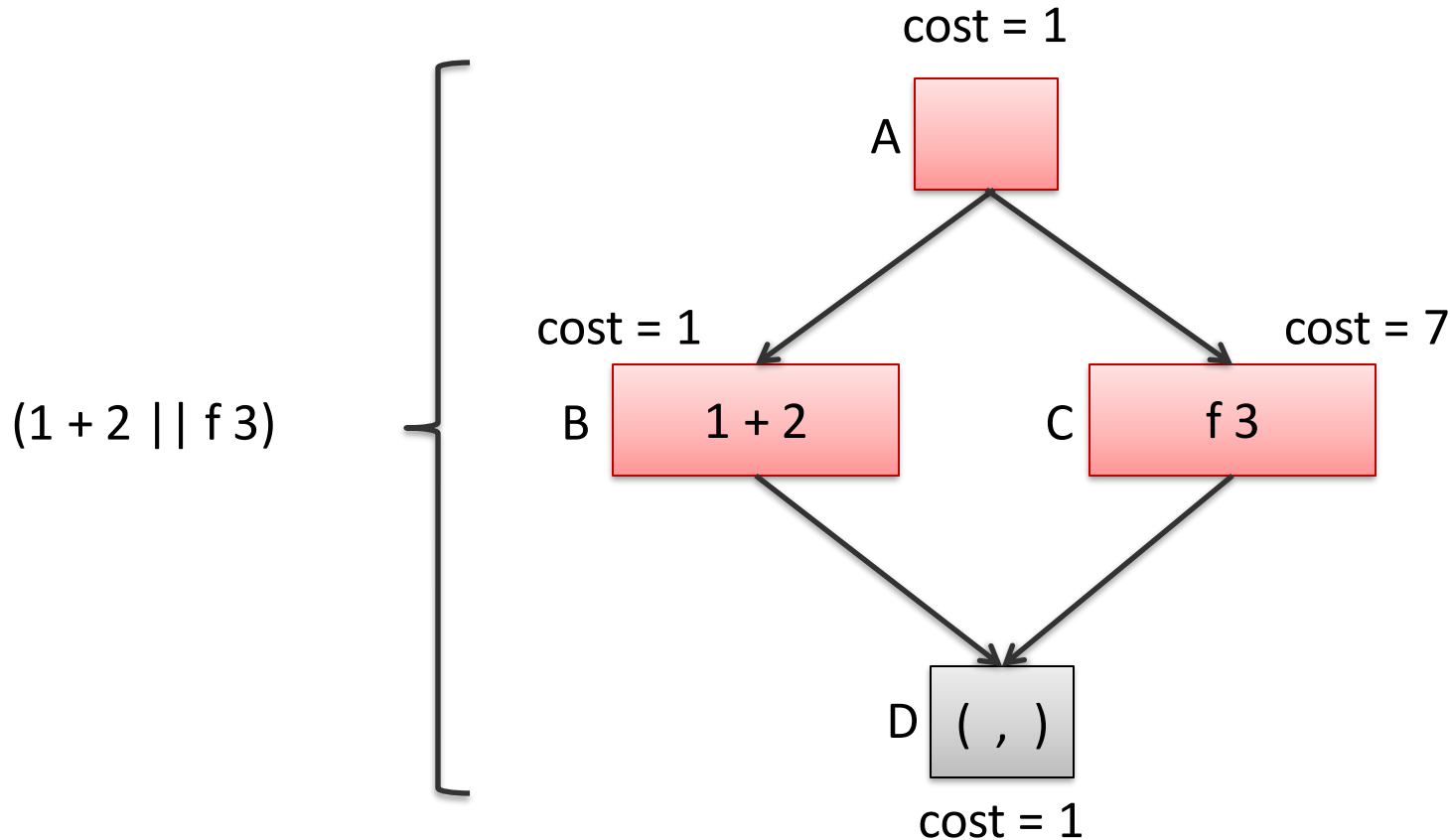
Suppose we have **2 processors**. How much time does this computation take?

Visualizing Computational Costs



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

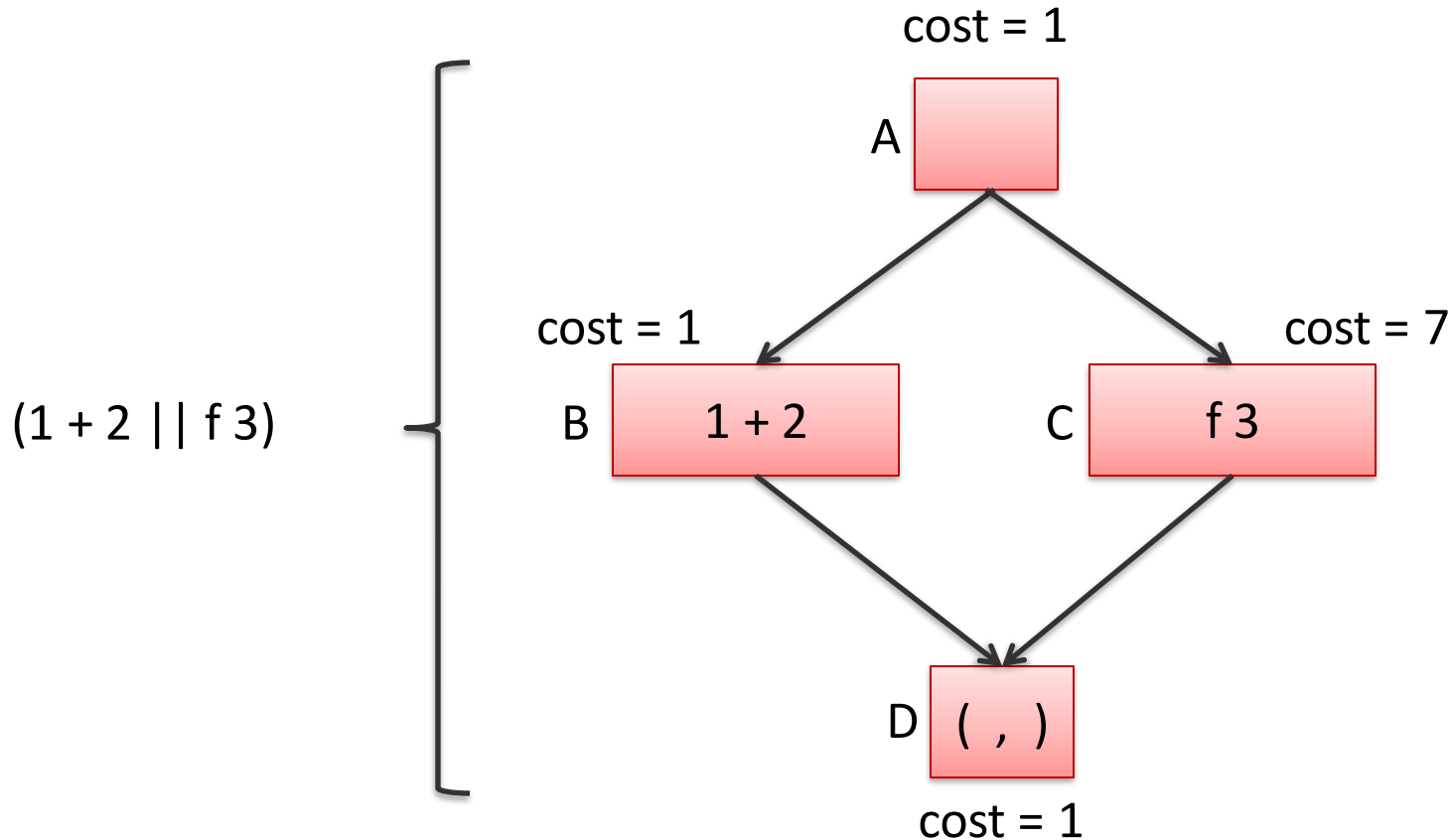
Visualizing Computational Costs



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

Cost so far: $1 + \max(1, 7)$

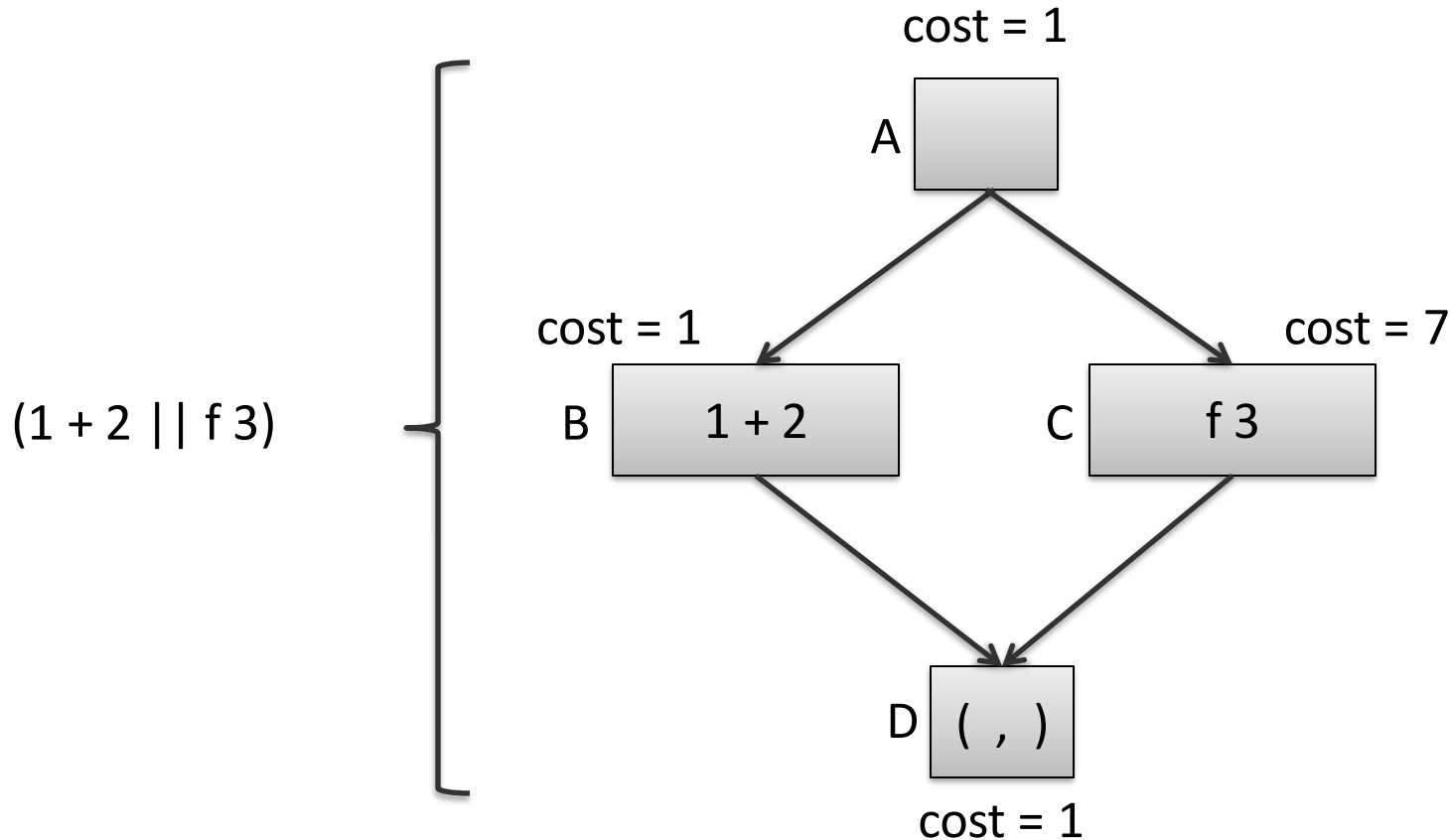
Visualizing Computational Costs



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

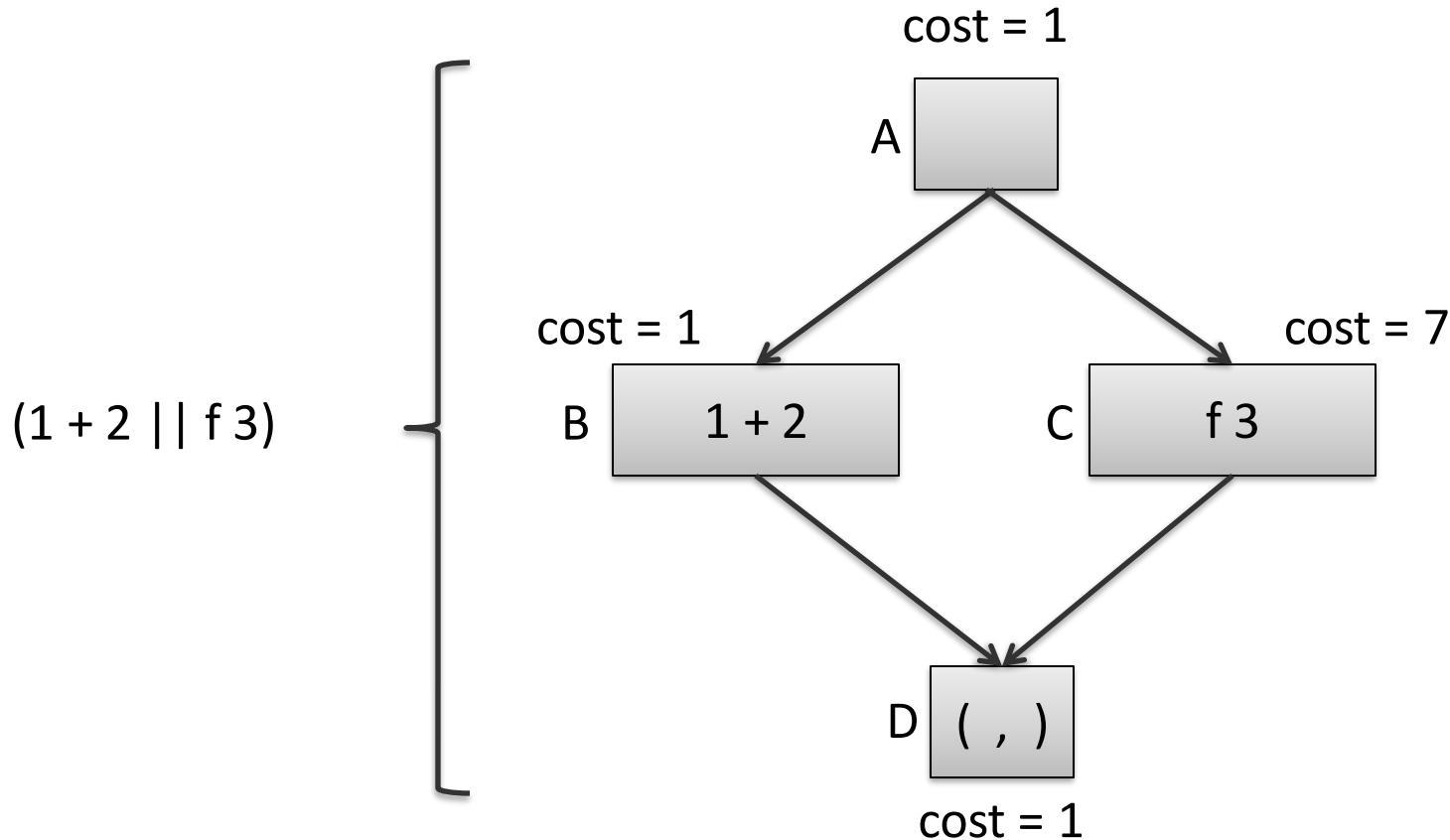
Cost so far: $1 + \max(1,7) + 1$

Visualizing Computational Costs



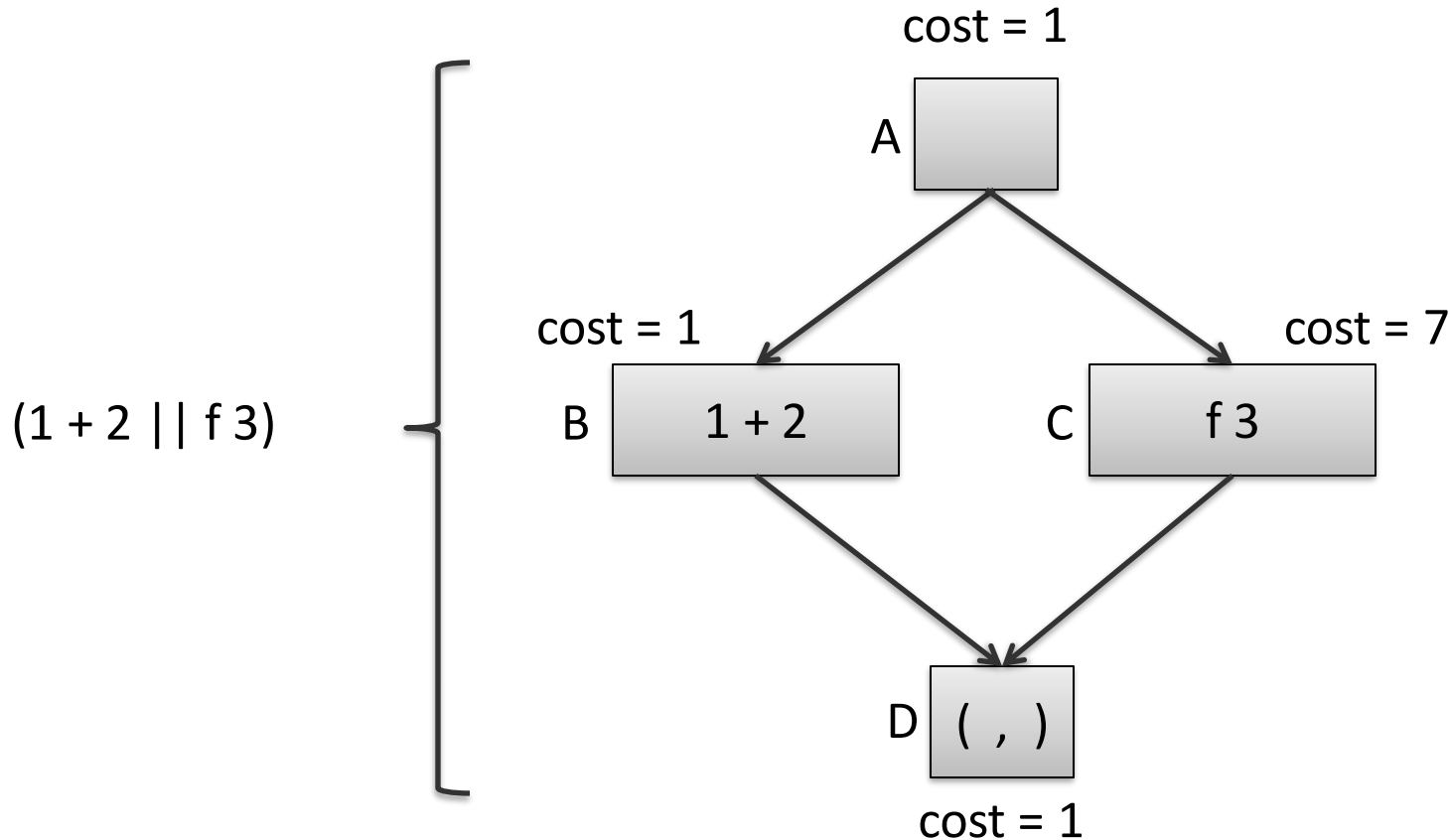
Suppose we have **2 processors**. How much time does this computation take?
Total cost: $1 + \max(1,7) + 1$. We say the *schedule* we used was: A-CB-D

Visualizing Computational Costs



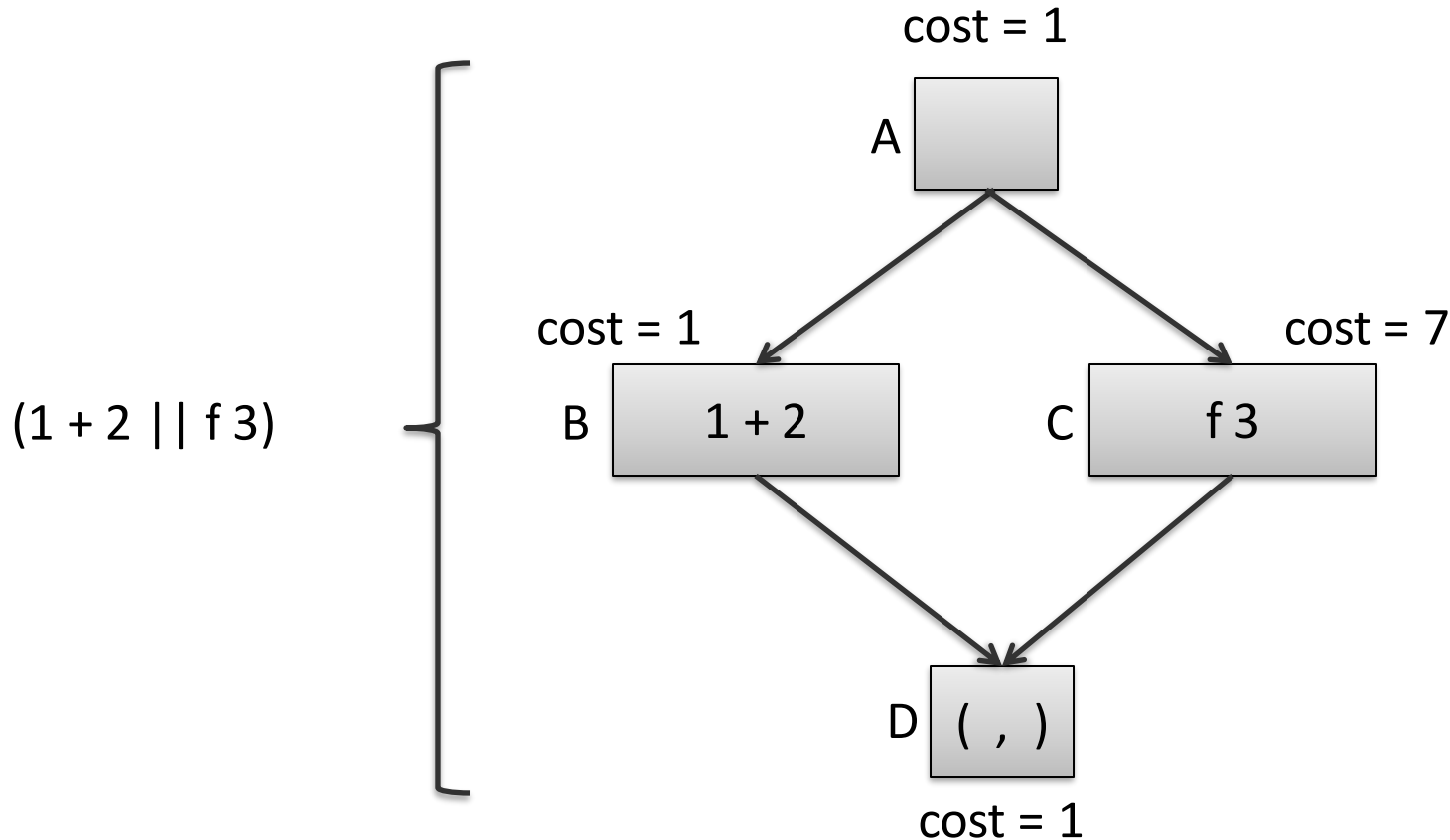
Suppose we have **3 processors**. How much time does this computation take?

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.

- $\text{parallelism} = \text{work} / \text{span}$

If $\text{work} = \text{span}$ then $\text{parallelism} = 1$.

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

If $\text{span} = \frac{1}{2} \text{work}$ then $\text{parallelism} = 2$

- We can use up to 2 processors

If $\text{work} = 100$, $\text{span} = 1$

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

Related concept:

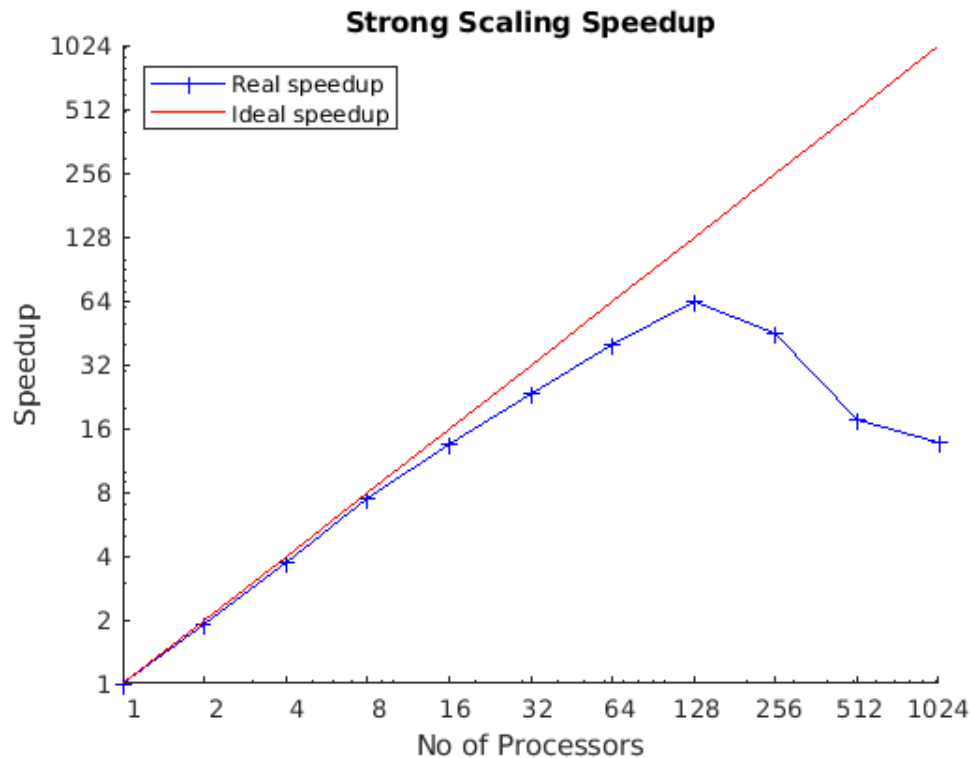
“speedup”

How much faster is
the n-processor
version

*in practice, not just
in theory*

$$\text{speedup} = \frac{\text{time taken by sequential algorithm}}{\text{time taken by parallel algorithm}}$$

“speedup of 45 on 256 processors”



What's wrong with this graph?

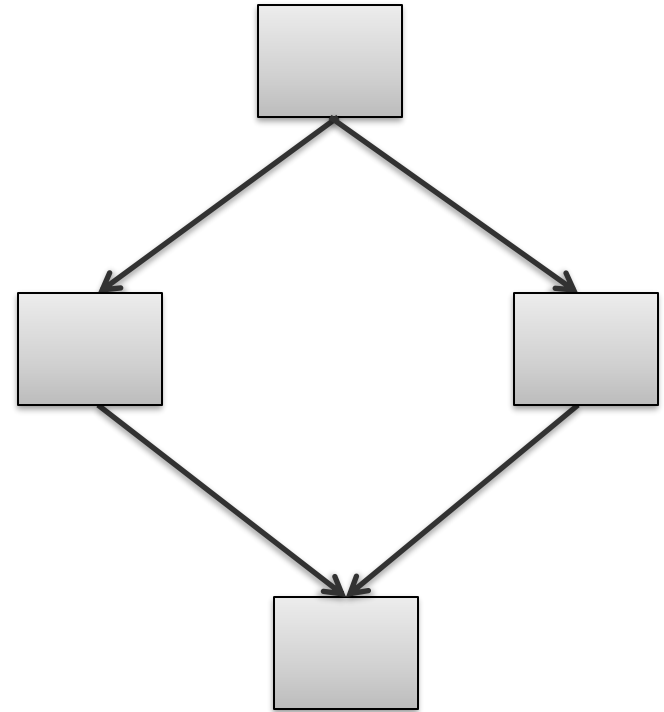
Series-Parallel Graphs



one operation



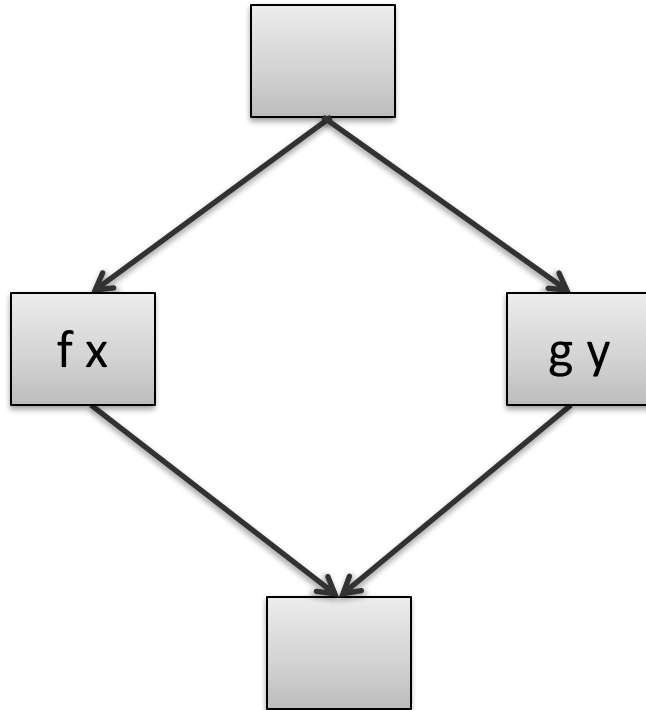
two operations
in sequence
 $e1; e2$



two operations
in parallel
 $(e1 \parallel e2)$

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

Parallel Pairs



```
let both f x g y =  
  let ff = future f x in  
  let gv = g y in  
  (force ff, gv)
```

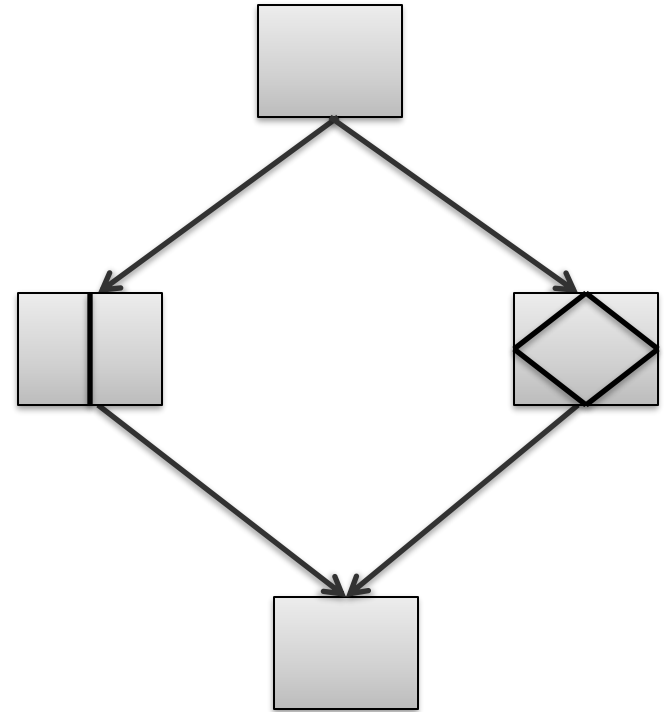
Series-Parallel Graphs Compose



one operation



two graphs
in sequence

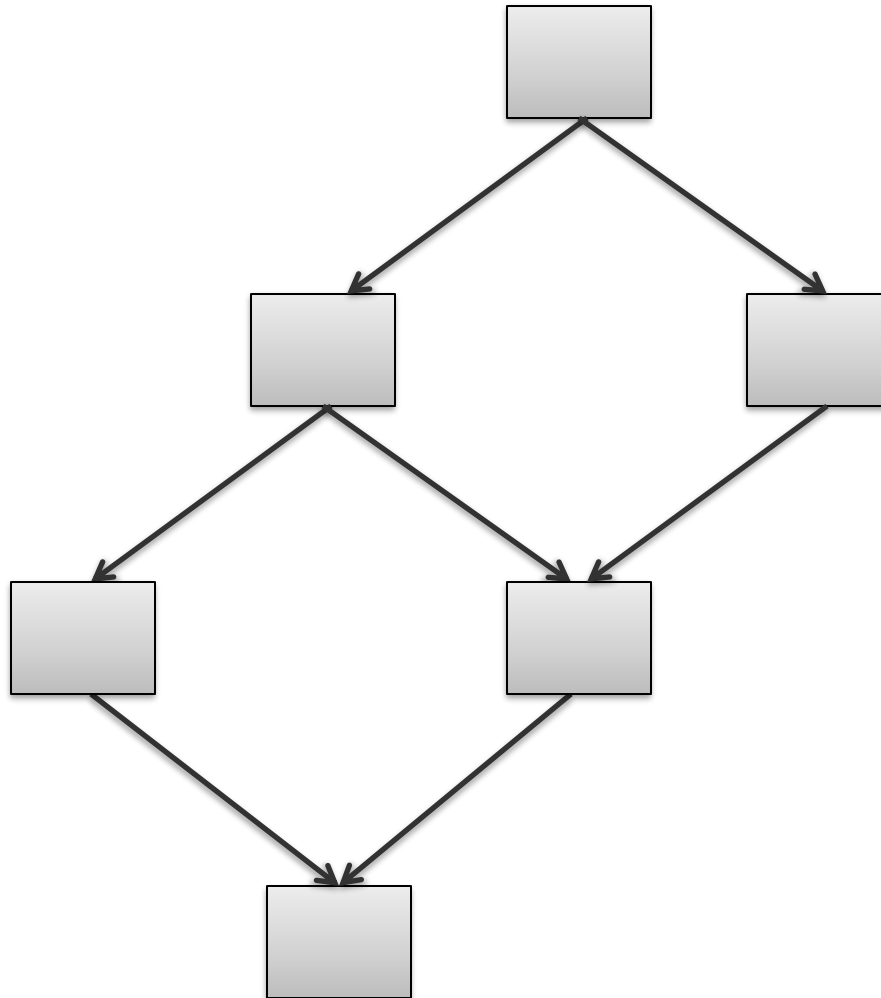


two graphs
in parallel

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



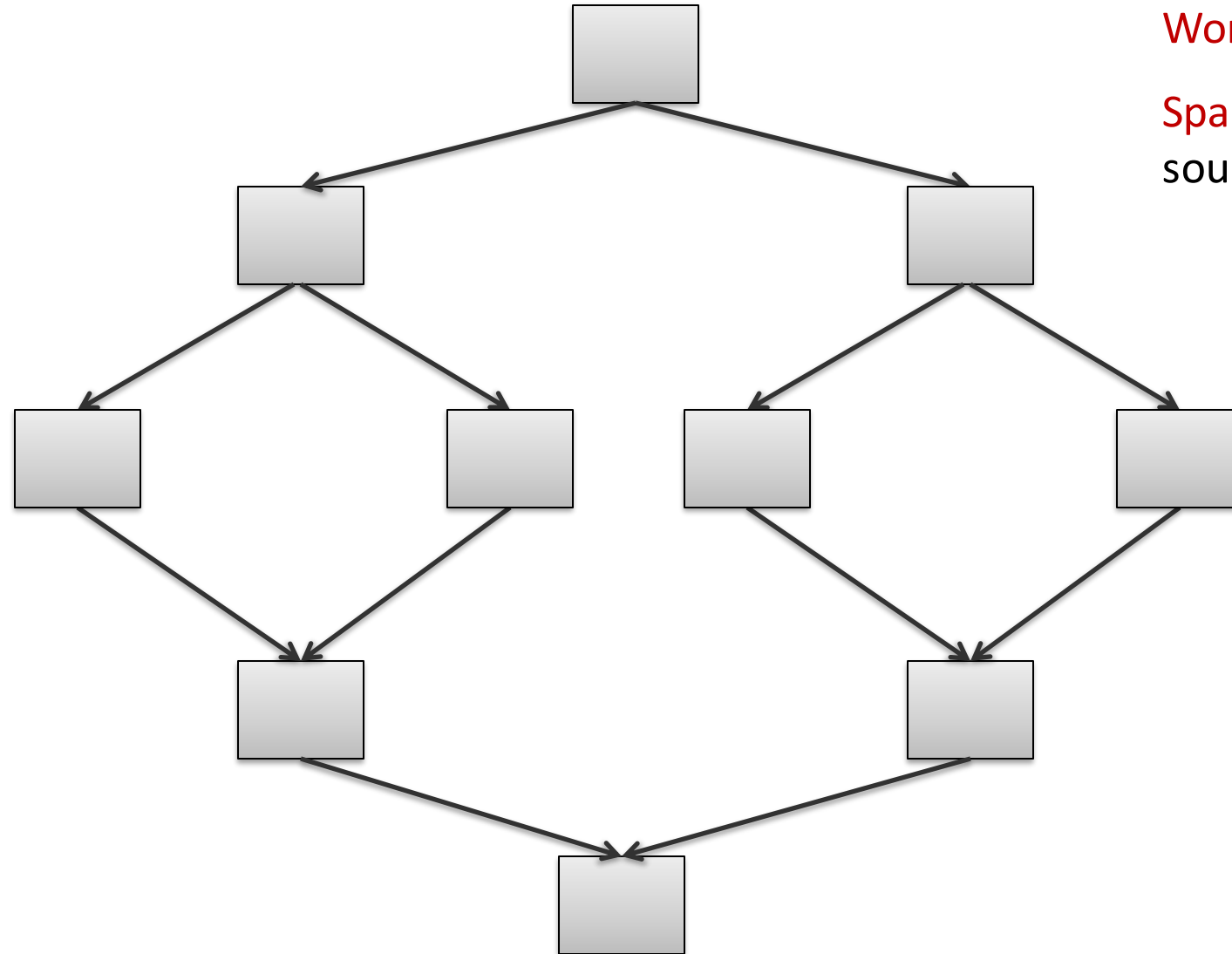
However:
The results about greedy schedulers (next few slides) do apply to DAG schedules as well as series-parallel schedules!

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

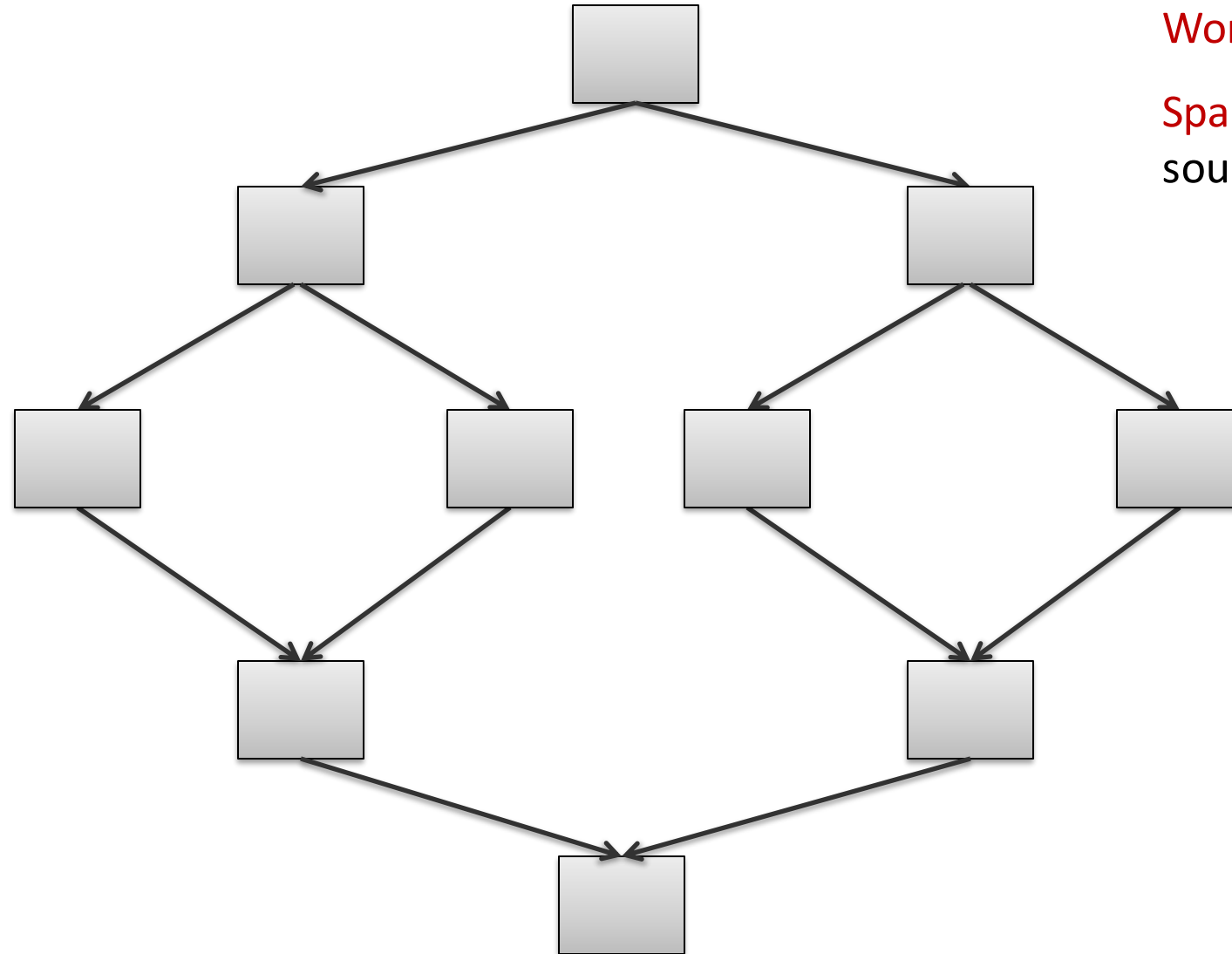
Let's assume each node costs 1.

Work: sum the nodes.

Span: longest path from source to sink.

work = 10

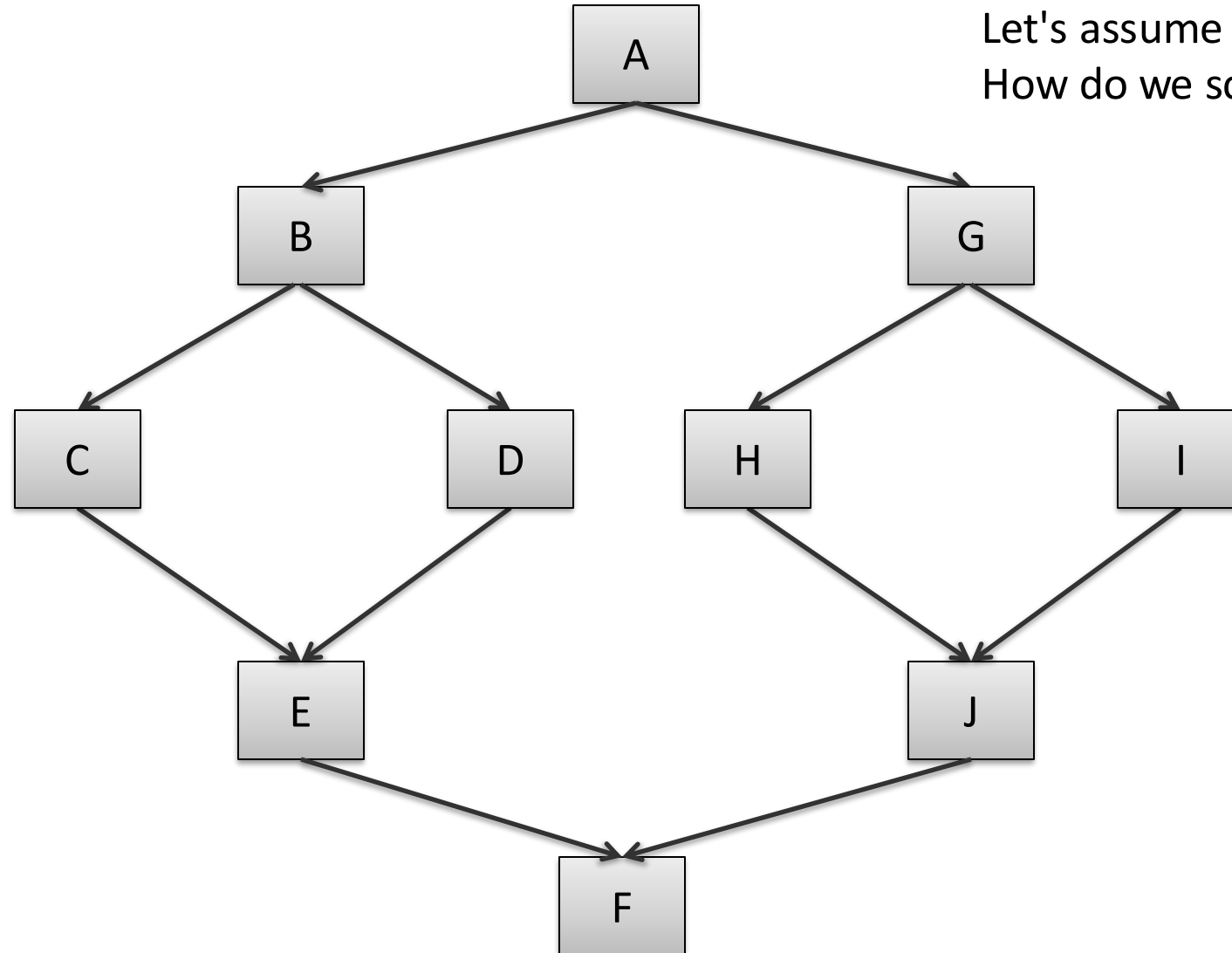
span = 5



Scheduling

Let's assume each node costs 1.

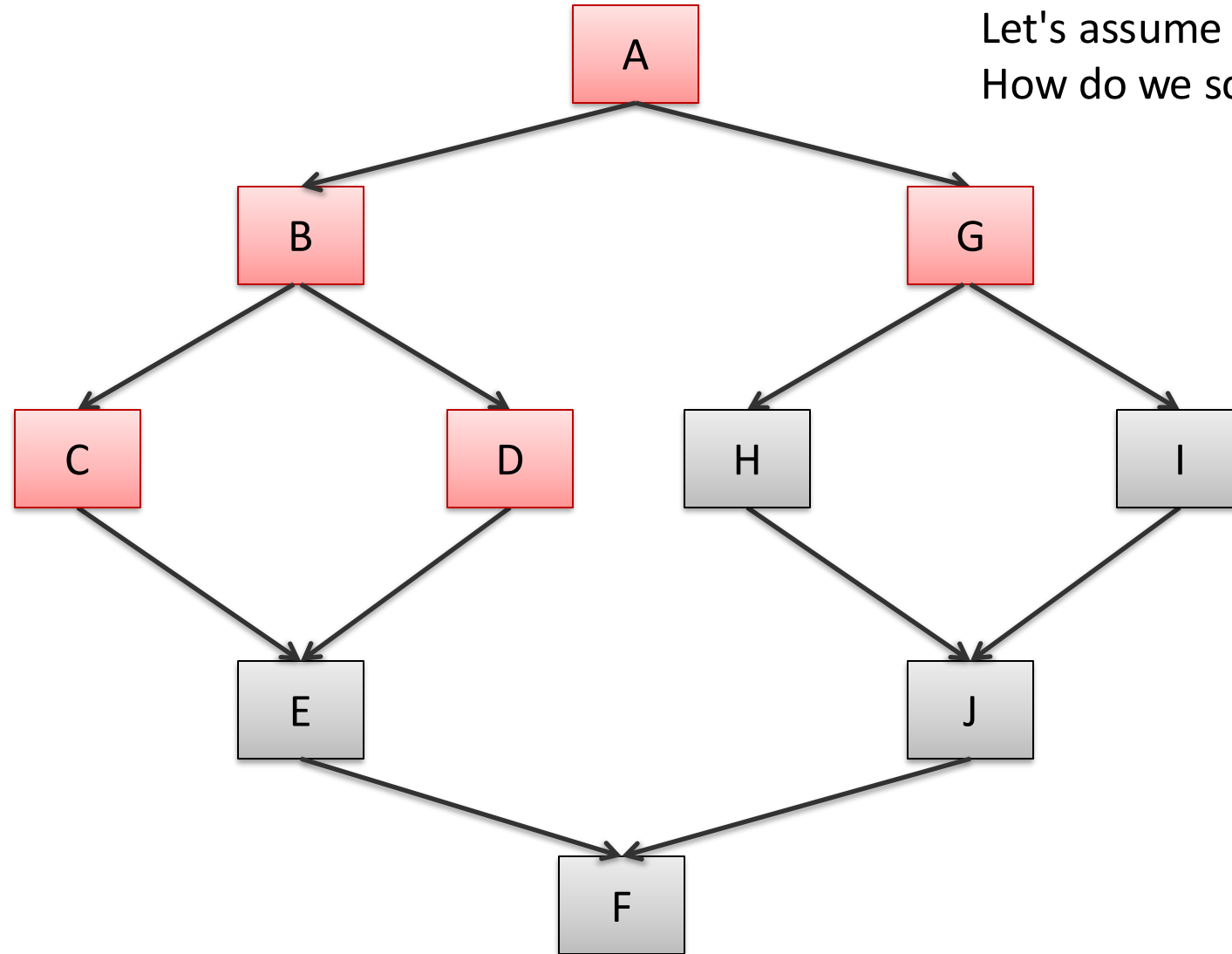
Let's assume we have 2 processors.
How do we schedule computation?



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

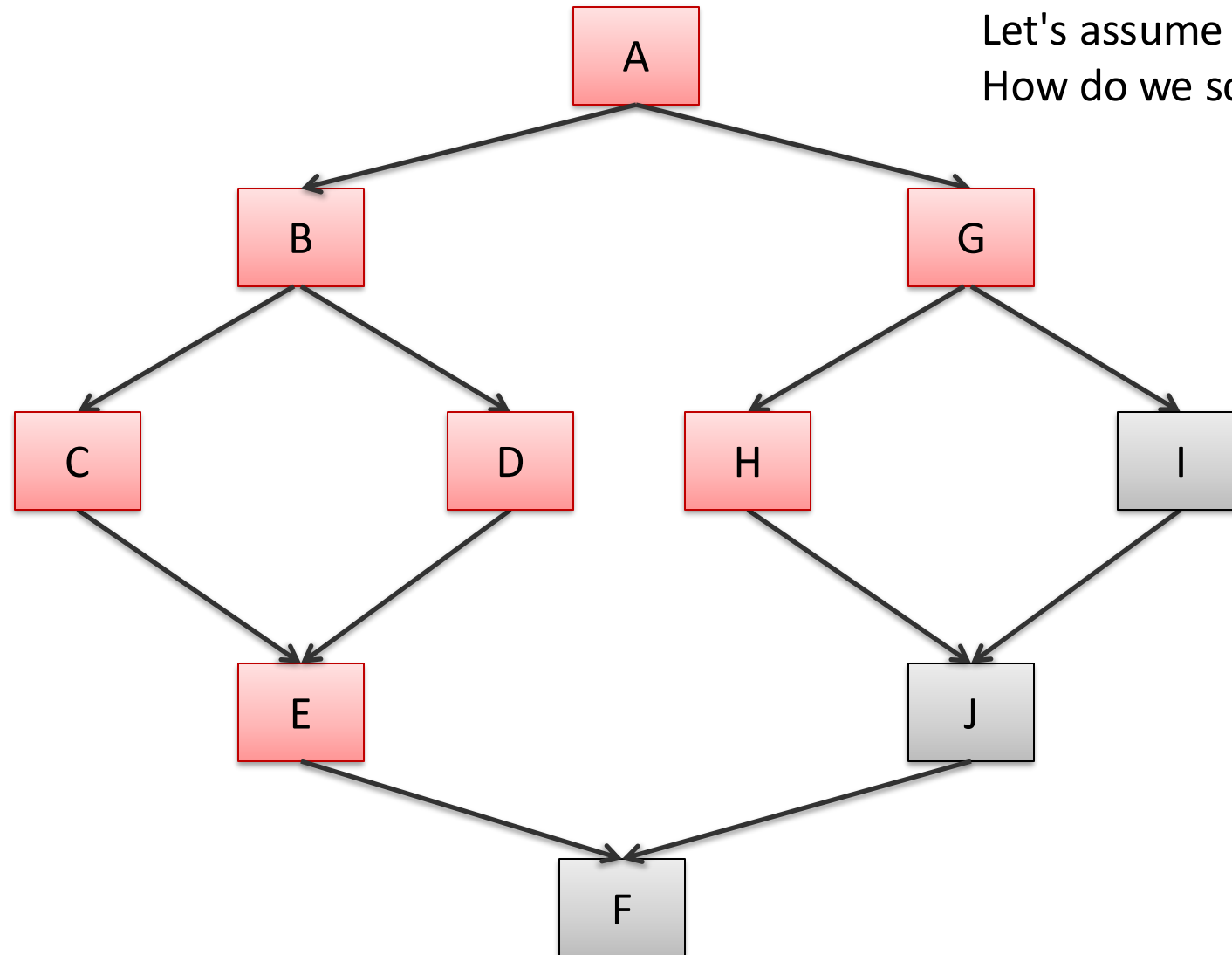
Scheduling

Let's assume each node costs 1.

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

Option 1:

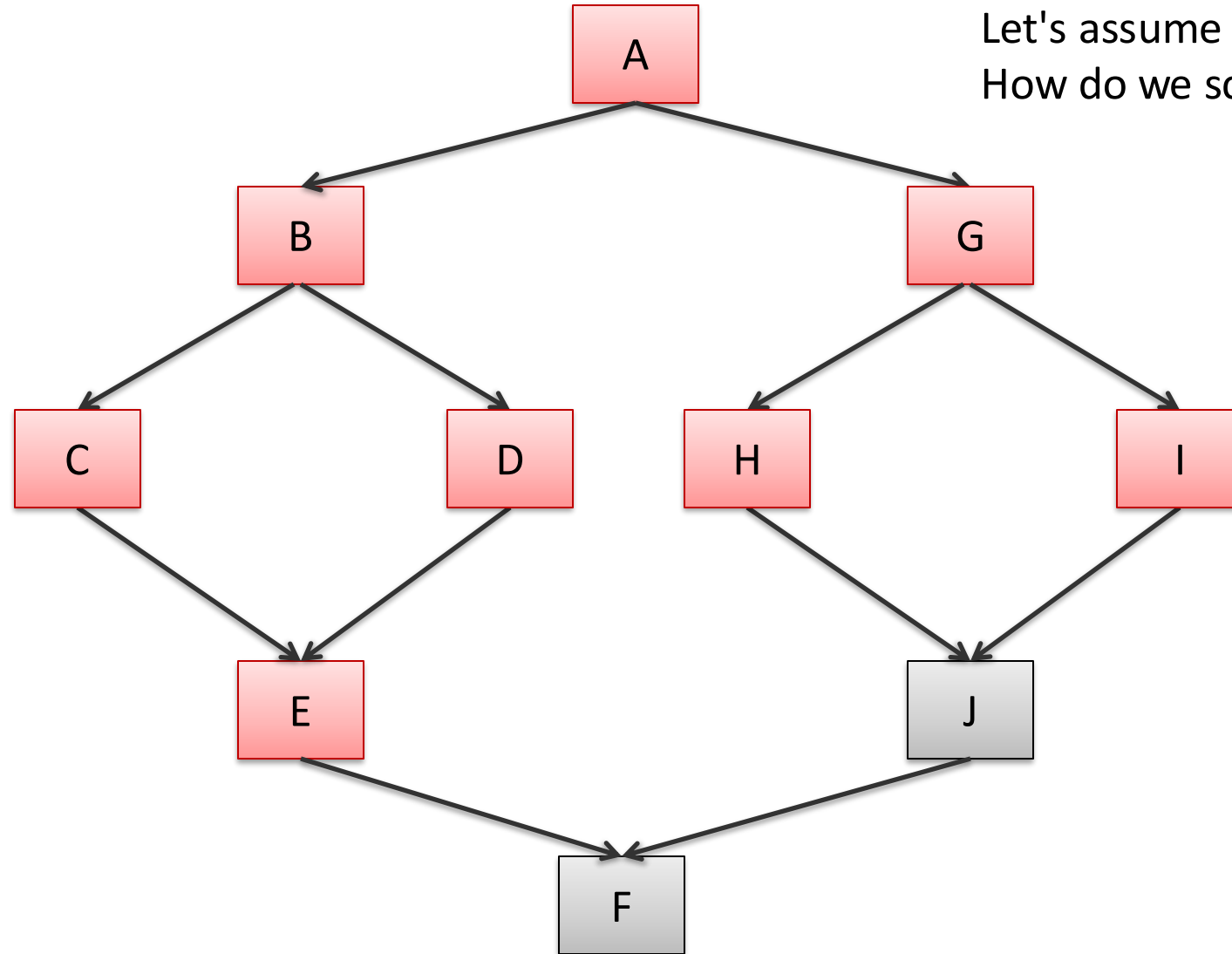
A
B G
C D
E H



Scheduling

Let's assume each node costs 1.

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



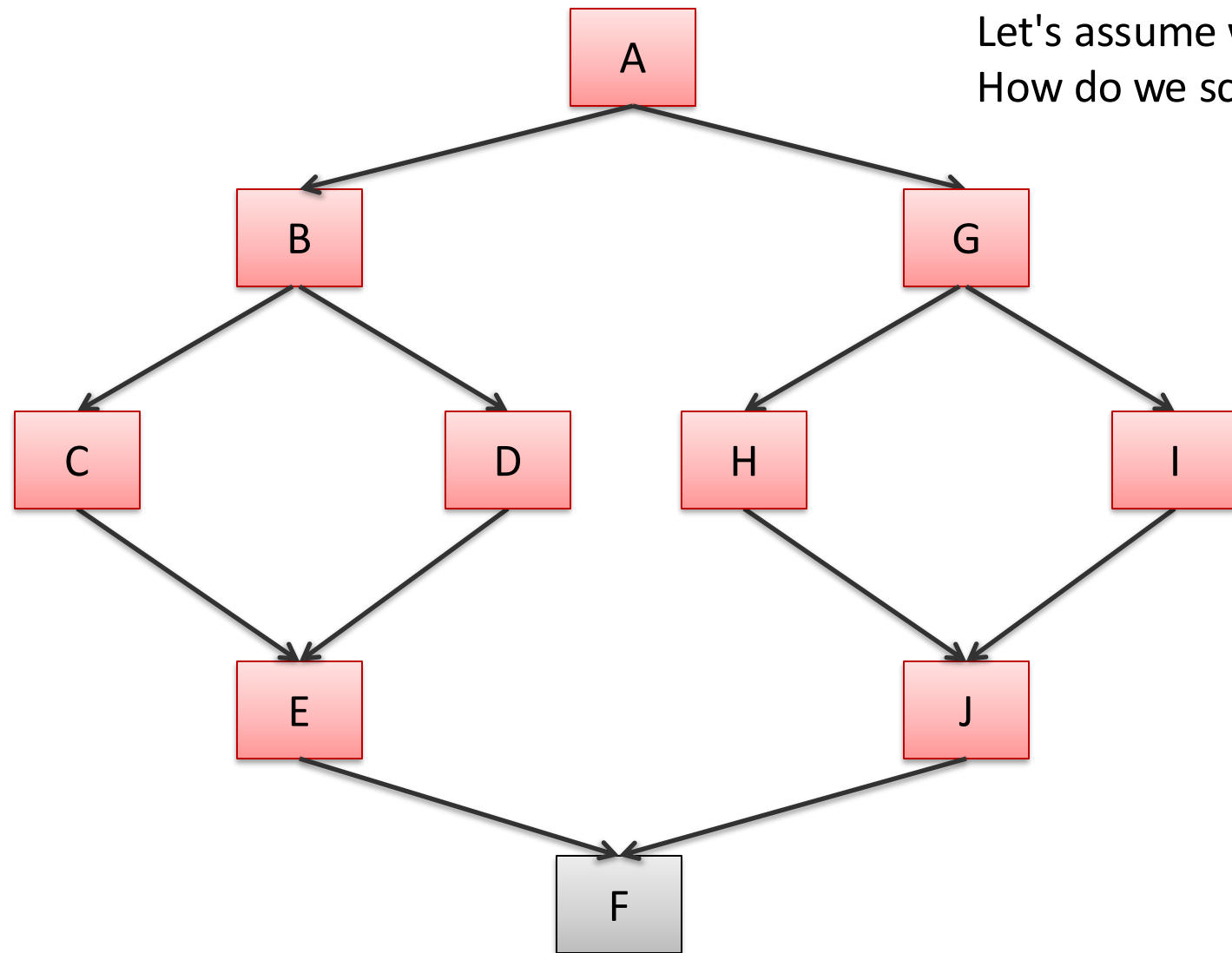
Option 1:

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?



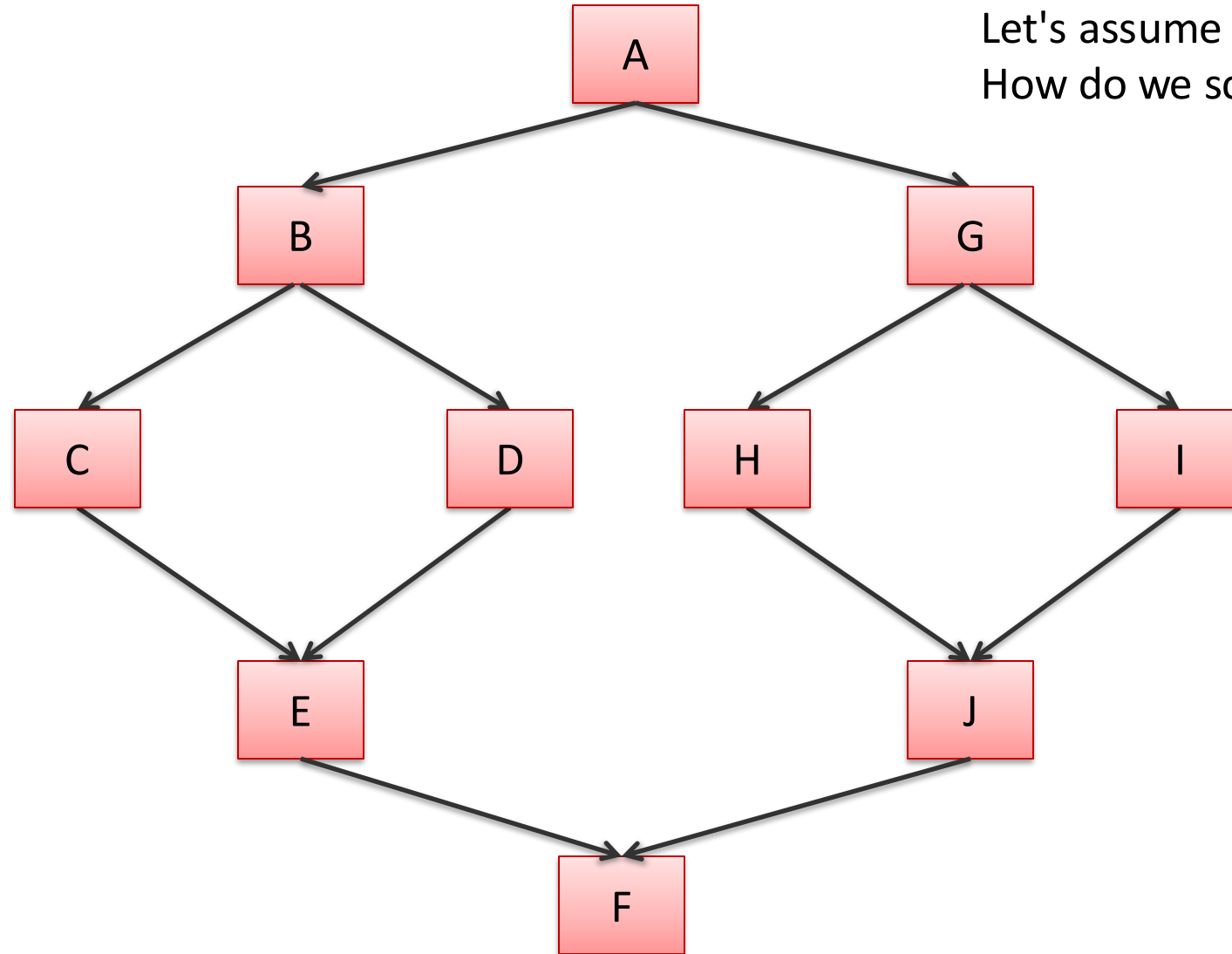
Option 1:

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?



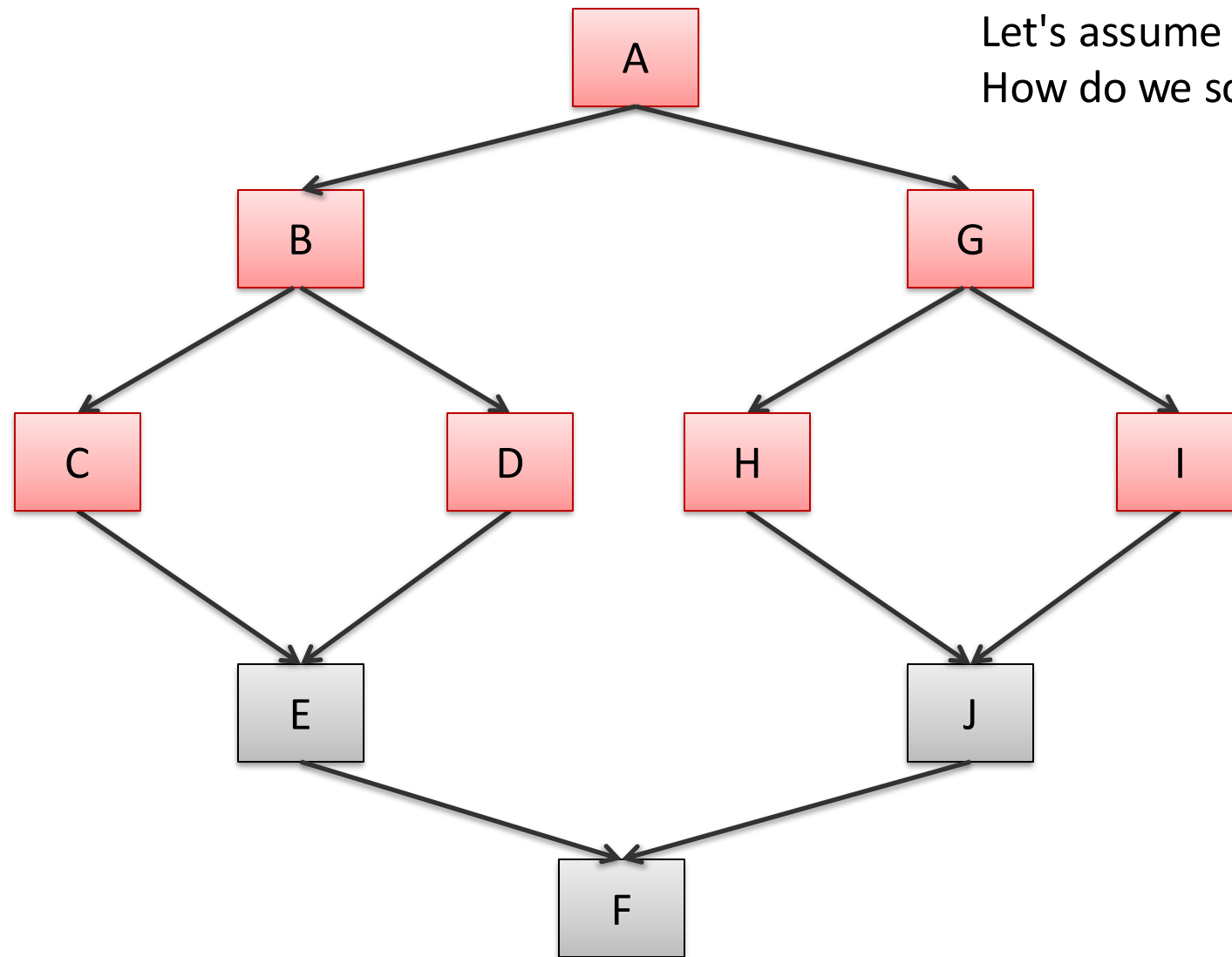
Option 1:

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

↓

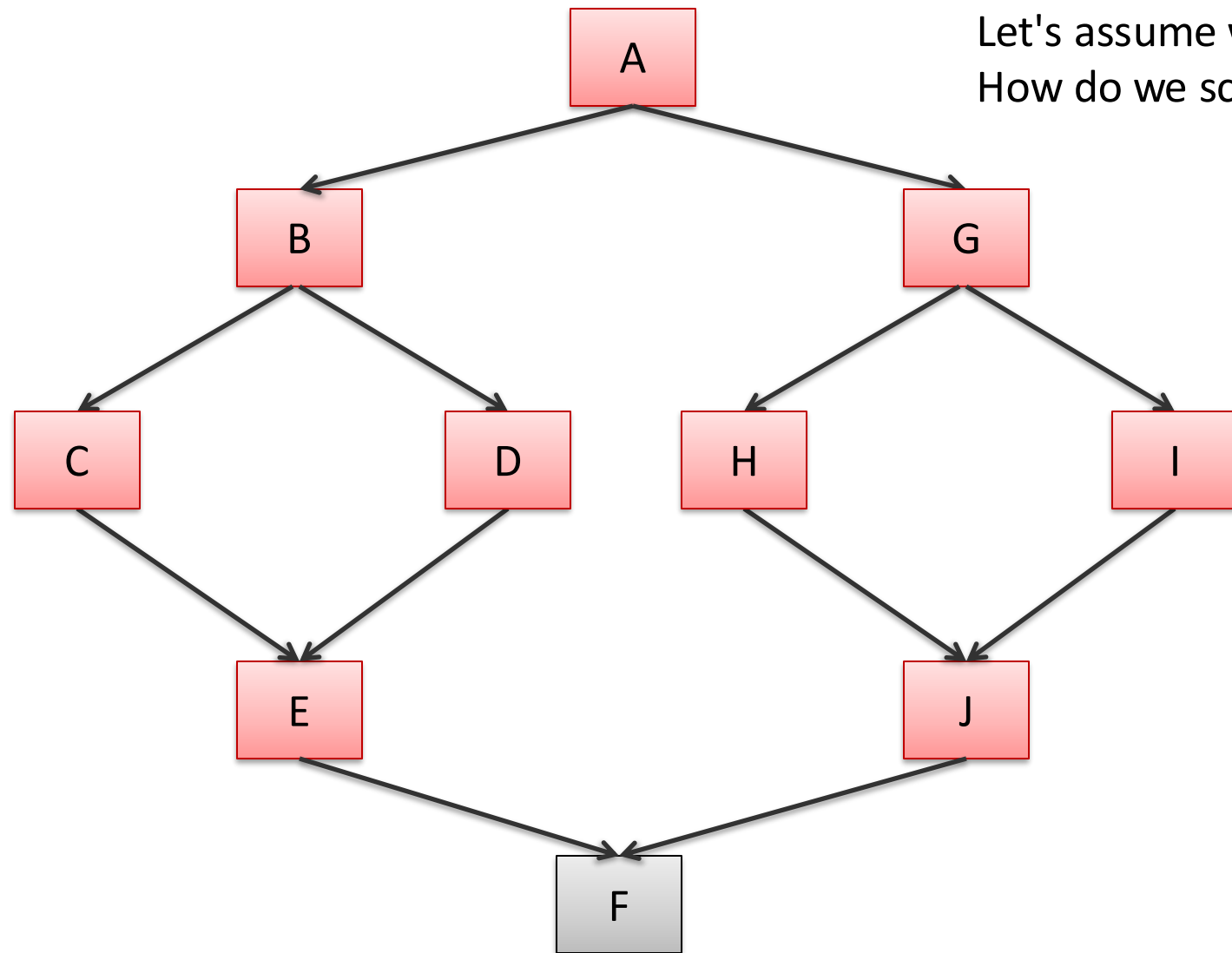
↓

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~~

↓

↓

F

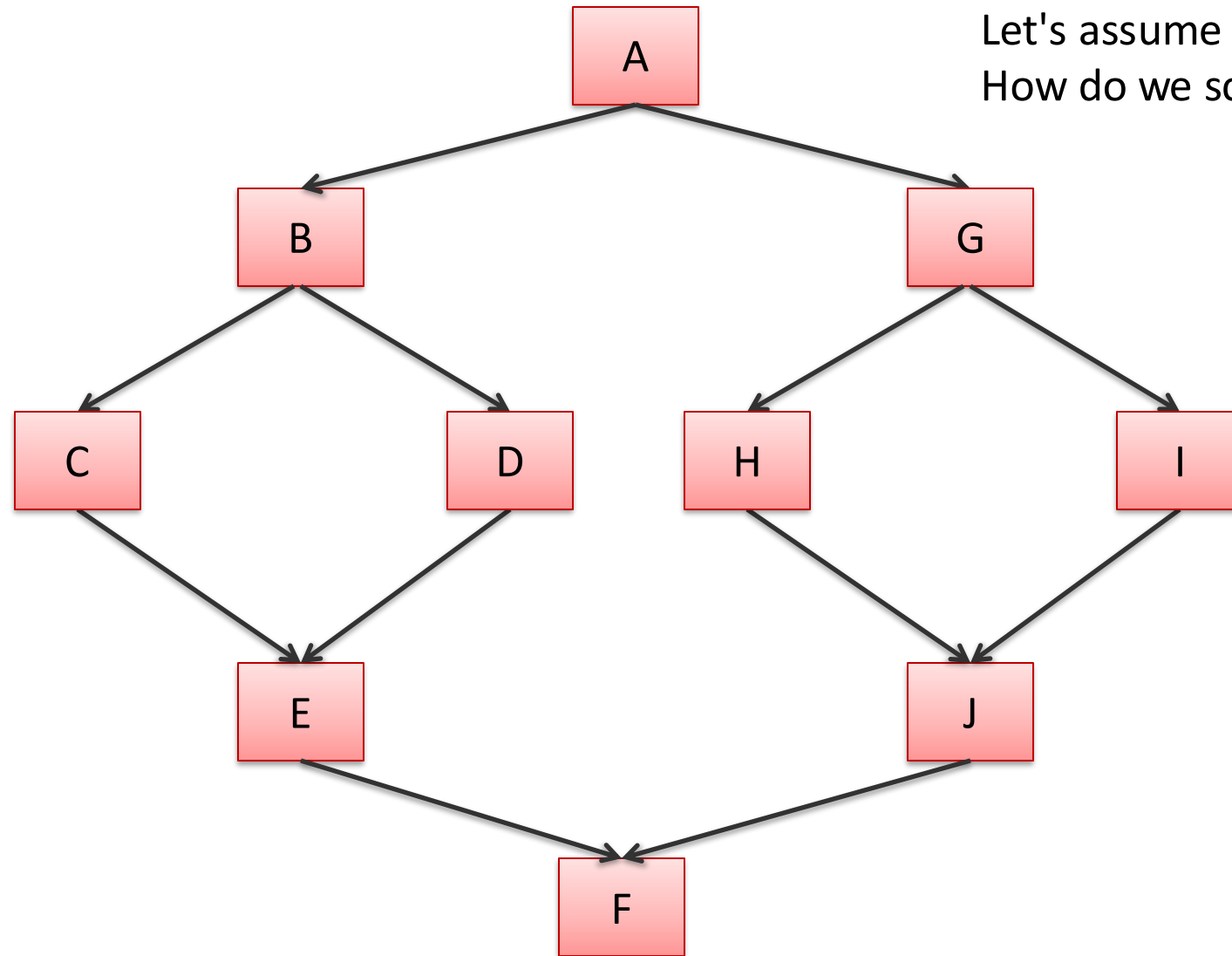
H I

E 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~~

+

J

F

H I

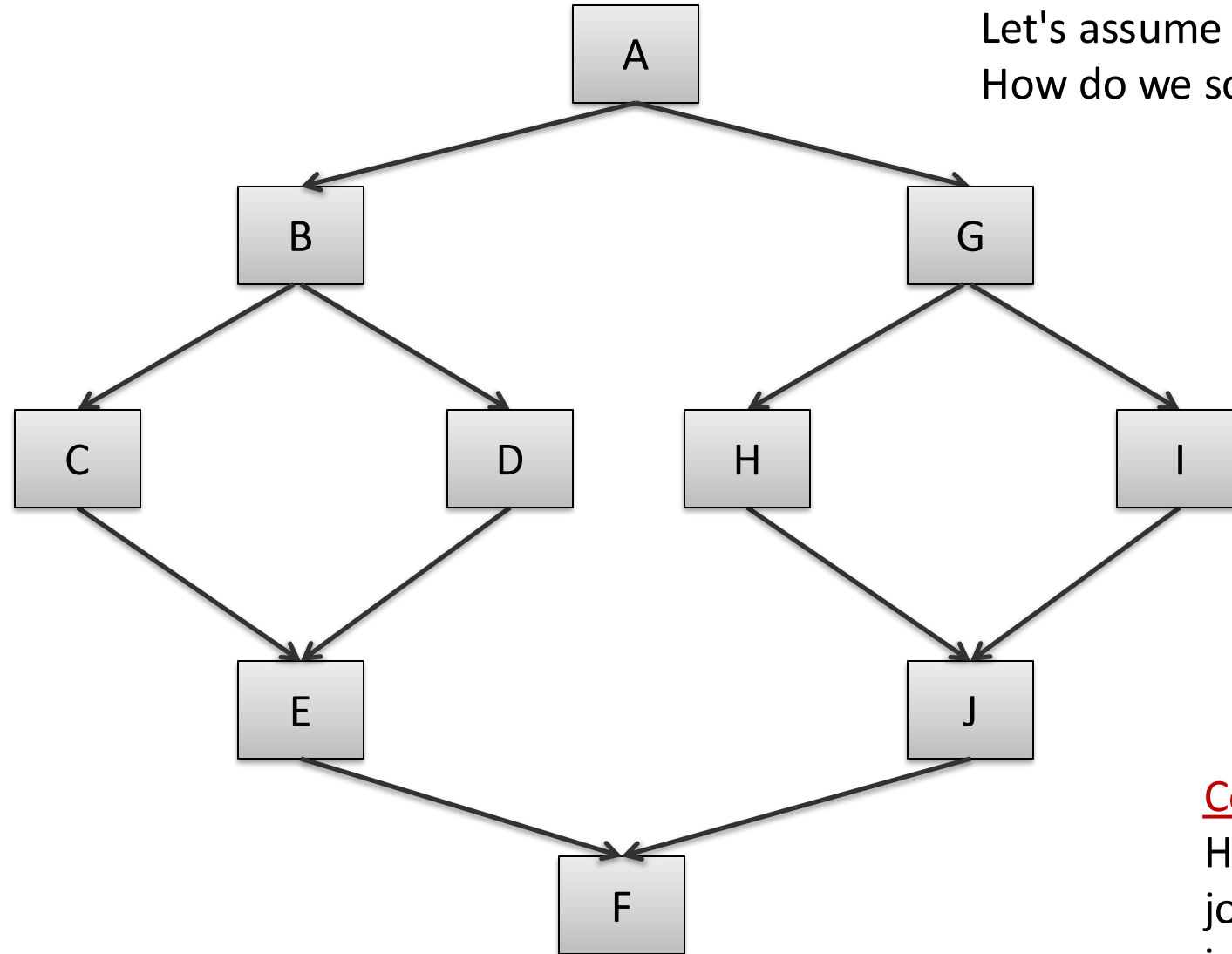
E J

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~~

↓

↓

F

H I

E J

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!

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):

- $T(p) < \text{work}/p + \text{span}$
 - 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
- $T(p) \geq \max(\text{work}/p, \text{span})$
 - can't do better than perfect division between processors (work/p)
 - can't be faster than span

Greedy Schedulers

Properties (for p processors):

$$\max(\text{work}/p, \text{span}) \leq T(p) < \text{work}/p + \text{span}$$

Consequences:

- as span gets small relative to work/p
 - $\text{work}/p + \text{span} \implies \text{work}/p$
 - $\max(\text{work}/p, \text{span}) \implies \text{work}/p$
 - so $T(p) \implies \text{work}/p$ greedy schedulers converge to the optimum!
- if span approaches the work
 - $\text{work}/p + \text{span} \implies \text{span}$
 - $\max(\text{work}/p, \text{span}) \implies \text{span}$
 - so $T(p) \implies \text{span}$ greedy schedulers converge to the optimum!

And therefore

Even though greedy schedulers are simple to implement,
they can be effective in building a parallel programming system.

and

This *supports* the idea that **work and span** are useful ways to reason about the cost of parallel programs.

PARALLEL SEQUENCES

Parallel Sequences

Parallel sequences

$$\langle e_0, e_1, e_2, \dots, e_{n-1} \rangle$$

Operations:

- creation (called **tabulate**)
- indexing an element in constant span
- map
- scan -- like a fold: $\langle u, u + e_0, u + e_0 + e_1, \dots \rangle$ log n span!

Languages:

- Nesl [Blelloch]
- Data-parallel Haskell

Parallel Sequences: Selected Operations

```
tabulate : (int -> 'a) -> int -> 'a seq  
  
tabulate f n == <f 0, f 1, ..., f (n-1)>  
work = O(n)      span = O(1)
```

Parallel Sequences: Selected Operations

```
tabulate : (int -> 'a) -> int -> 'a seq
```

```
tabulate f n == <f 0, f 1, ..., f (n-1)>
```

```
work = O(n)          span = O(1)
```

```
nth : 'a seq -> int -> 'a
```

```
nth <e0, e1, ..., e(n-1)> i == ei
```

```
work = O(1)          span = O(1)
```

Parallel Sequences: Selected Operations

```
tabulate : (int -> 'a) -> int -> 'a seq
```

```
tabulate f n == <f 0, f 1, ..., f (n-1)>  
work = O(n)      span = O(1)
```

```
nth : 'a seq -> int -> 'a
```

```
nth <e0, e1, ..., e(n-1)> i == ei  
work = O(1)      span = O(1)
```

```
length : 'a seq -> int
```

```
length <e0, e1, ..., e(n-1)> == n  
work = O(1)      span = O(1)
```

Example Problems

Write a function that creates the sequence $\langle 0, \dots, n-1 \rangle$
with $\text{Span} = O(1)$ and $\text{Work} = O(n)$.

Operations:

		Work	Span
tabulate	$f\ n$	n	1
nth	$i\ s$	1	1
length	s	1	1

Example Problems

Write a function that creates the sequence $\langle 0, \dots, n-1 \rangle$
with $\text{Span} = O(1)$ and $\text{Work} = O(n)$.

```
(* create n == <0, 1, ..., n-1> *)  
let create n =
```

Operations:

	Work	Span
tabulate f n	n	1
nth i s	1	1
length s	1	1

Example Problems

Write a function that creates the sequence $\langle 0, \dots, n-1 \rangle$
with $\text{Span} = O(1)$ and $\text{Work} = O(n)$.

```
(* create n == <0, 1, ..., n-1> *)  
let create n =  
  tabulate (fun i -> i) n
```

Operations:

	Work	Span
tabulate f n	n	1
nth i s	1	1
length s	1	1

Example Problems

Write a function such that given a sequence $\langle v_0, \dots, v_{n-1} \rangle$, maps f over each element of the sequence with $\text{Span} = O(1)$ and $\text{Work} = O(n)$, returning the new sequence (if f is constant work)

Operations:

	Work	Span
<code>tabulate f n</code>	<code>n</code>	<code>1</code>
<code>nth i s</code>	<code>1</code>	<code>1</code>
<code>length s</code>	<code>1</code>	<code>1</code>

Example Problems

Write a function such that given a sequence $\langle v_0, \dots, v_{n-1} \rangle$, maps f over each element of the sequence with $\text{Span} = O(1)$ and $\text{Work} = O(n)$, returning the new sequence (if f is constant work)

```
(* map f <v0, ..., vn-1> == <f v0, ..., f vn-1> *)  
let map f s =
```

Operations:

	Work	Span
<code>tabulate f n</code>	<code>n</code>	<code>1</code>
<code>nth i s</code>	<code>1</code>	<code>1</code>
<code>length s</code>	<code>1</code>	<code>1</code>

Example Problems

Write a function such that given a sequence $\langle v_0, \dots, v_{n-1} \rangle$, maps f over each element of the sequence with $\text{Span} = O(1)$ and $\text{Work} = O(n)$, returning the new sequence (if f is constant work)

```
(* map f <v0, ..., vn-1> == <f v0, ..., f vn-1> *)  
let map f s =  
  tabulate (fun i -> f (nth s i)) (length s)
```

Operations:

	Work	Span
tabulate f n	n	1
nth i s	1	1
length s	1	1

Example Problems

Write a function such that given a sequence $\langle v_0, \dots, v_{n-1} \rangle$, reverses the sequence. with $\text{Span} = O(1)$ and $\text{Work} = O(n)$

Operations:

	Work	Span
<code>tabulate f n</code>	<code>n</code>	<code>1</code>
<code>nth i s</code>	<code>1</code>	<code>1</code>
<code>length s</code>	<code>1</code>	<code>1</code>

Example Problems

Write a function such that given a sequence $\langle v_0, \dots, v_{n-1} \rangle$, reverses the sequence. with $\text{Span} = O(1)$ and $\text{Work} = O(n)$

```
(* reverse  $\langle v_0, \dots, v_{n-1} \rangle == \langle v_{n-1}, \dots, v_0 \rangle$  *)  
let reverse s =
```

Operations:

	Work	Span
tabulate f n	n	1
nth i s	1	1
length s	1	1

Example Problems

Write a function such that given a sequence $\langle v_0, \dots, v_{n-1} \rangle$, reverses the sequence. with Span = $O(1)$ and Work = $O(n)$

```
(* reverse  $\langle v_0, \dots, v_{n-1} \rangle == \langle v_{n-1}, \dots, v_0 \rangle$  *)  
let reverse s =  
  let n = length s in  
  tabulate (fun i -> nth s (n-i-1)) n
```

Operations:

	Work	Span
tabulate f n	n	1
nth i s	1	1
length s	1	1

A Parallel Sequence API

	<u>Work</u>	<u>Span</u>
type 'a seq		
tabulate : (int -> 'a) -> int -> 'a seq	O(N)	O(1)
length : 'a seq -> int	O(1)	O(1)
nth : 'a seq -> int -> 'a	O(1)	O(1)
append : 'a seq -> 'a seq -> 'a seq (can build this from tabulate, nth, length)	O(N+M)	O(1)
split : 'a seq -> int -> 'a seq * 'a seq	O(N)	O(1)

For efficient implementations, see this paper by Andrew Tao '24:

<https://icfp23.sigplan.org/details/ocaml-2023-papers/2/Parallel-Sequences-in-Multicore-OCaml>

Fold and Reduce

We have seen many sequential algorithms can be programmed succinctly using fold or reduce. Eg: sum all elements:

sum:

0

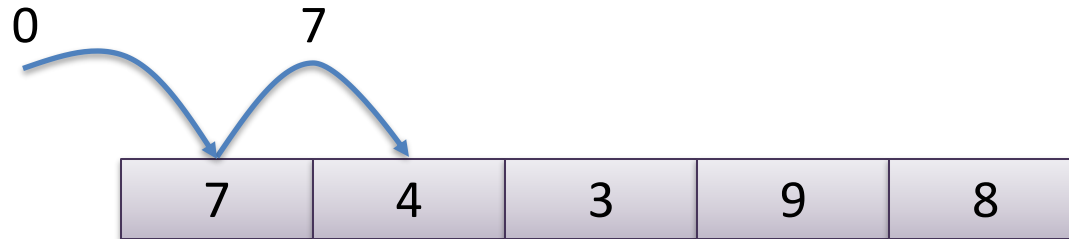


7	4	3	9	8
---	---	---	---	---

Fold and Reduce

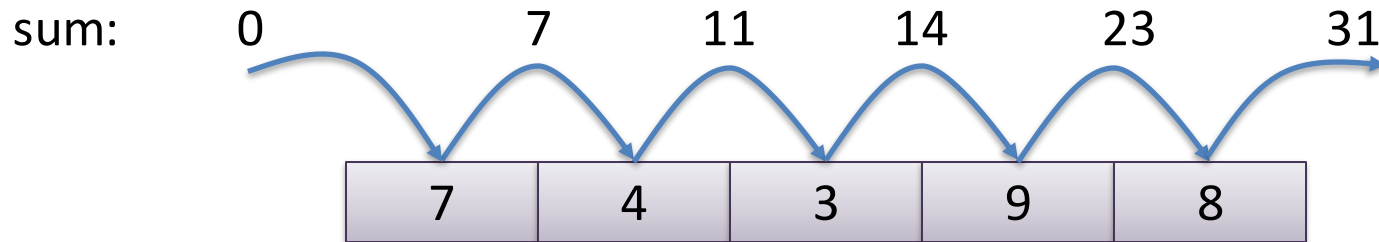
We have seen many sequential algorithms can be programmed succinctly using fold or reduce. Eg: sum all elements:

sum:



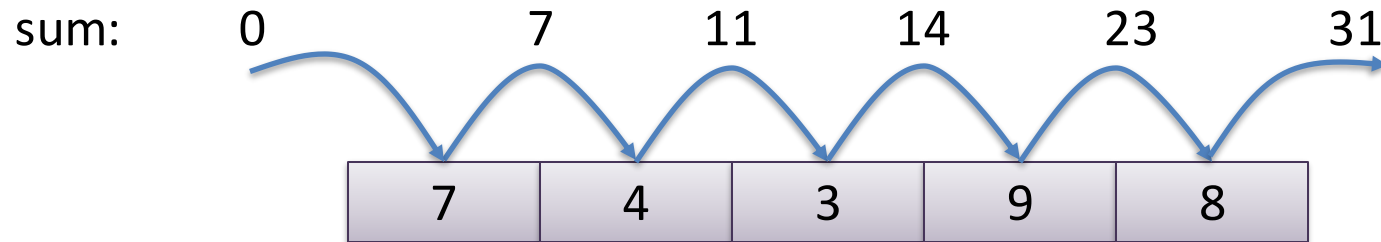
Fold and Reduce

We have seen many sequential algorithms can be programmed succinctly using fold or reduce. Eg: sum all elements:



Fold and Reduce

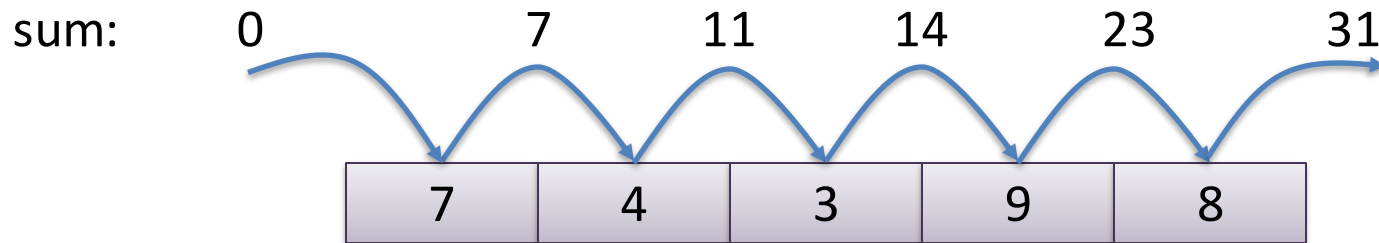
We have seen many sequential algorithms can be programmed succinctly using fold or reduce. Eg: sum all elements:



```
let sum_all (l:int list) = reduce (+) 0 l
```

Fold and Reduce

We have seen many sequential algorithms can be programmed succinctly using fold or reduce. Eg: sum all elements:



```
let sum_all (l:int list) = reduce (+) 0 l
```

Key to parallelization: Notice that because sum is an *associative* operator, we do not have to add the elements strictly left-to-right:

$$((((init + v1) + v2) + v3) + v4) + v5 == ((init + v1) + v2) + ((v3 + v4) + v5)$$

add on processor 1

add on processor 2

Side Note

The key is *associativity*:

$$((((init + v1) + v2) + v3) + v4) + v5 == ((init + v1) + v2) + ((v3 + v4) + v5)$$

add on processor 1

add on processor 2

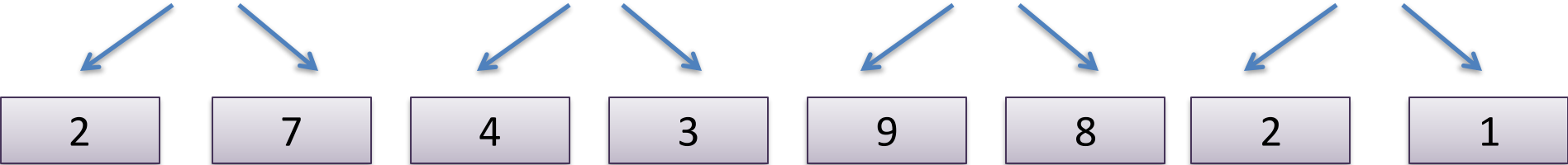
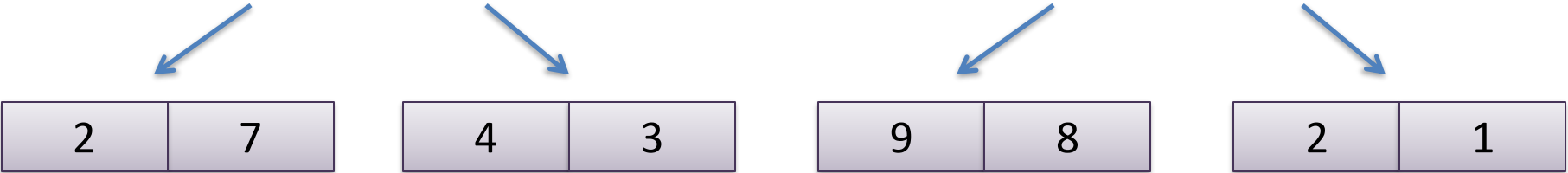
Commutativity not needed!

Commutativity allows us to reorder the elements:

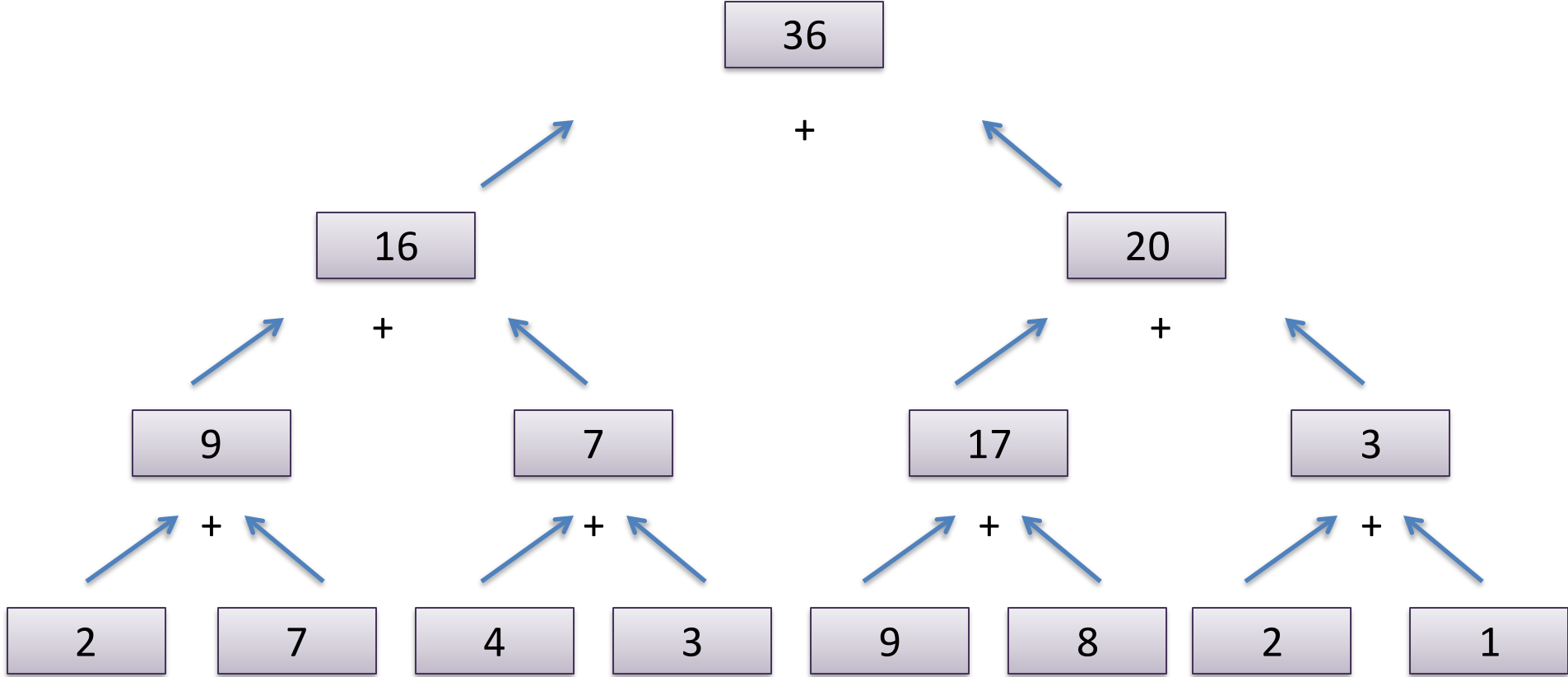
$$v1 + v2 == v2 + v1$$

But we don't have to reorder elements to obtain a significant speedup; we just have to reorder the execution of the operations.

Parallel Sum



Parallel Sum

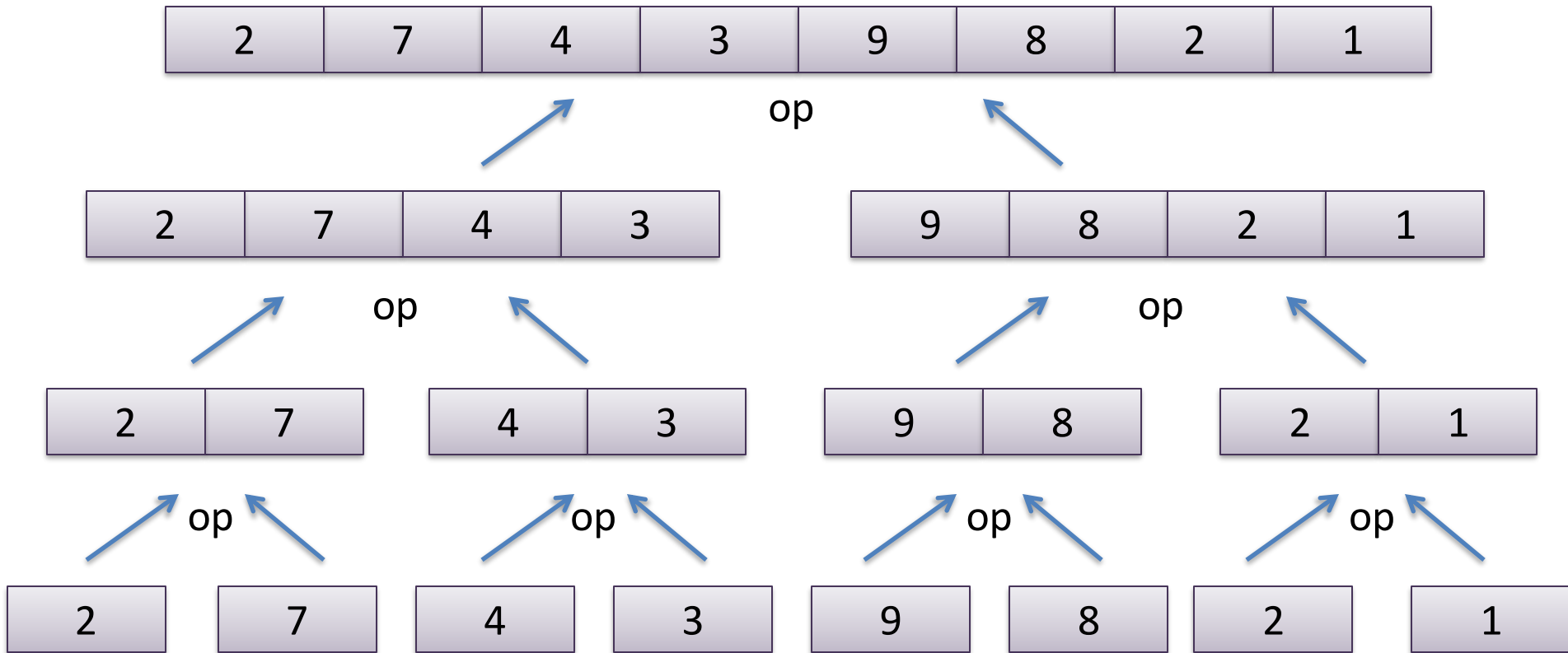


Parallel Sum

```
let both f x g y =  
  let ff = future f x in  
  let gv = g y in  
  (force ff, gv)
```

```
let rec psum (s : int seq) : int =  
  match length s with  
  | 0 -> 0  
  | 1 -> nth s 0  
  | n ->  
    let (s1, s2) = split (n/2) s in  
    let (a1, a2) = both psum s1  
                    psum s2 in  
    a1 + a2
```


Parallel Reduce



If op is associative and the base case has the properties:

$$\text{op base } X == X$$

$$\text{op } X \text{ base} == X$$

then the parallel reduce is equivalent to the sequential left-to-right fold.

Parallel Reduce

```
let rec reduce (f:'a -> 'a -> 'a) (base:'a) (s:'a seq) =
  match length s with
  | 0 -> base
  | 1 -> nth s 0
  | n ->
    let (s1,s2) = split (n/2) s in
    let (n1, n2) = both (reduce f base) s1
                  (reduce f base) s2 in
    f n1 n2
```

Parallel Reduce

```
let rec reduce (f:'a -> 'a -> 'a) (base:'a) (s:'a seq) =  
  match length s with  
  | 0 -> base  
  | 1 -> nth s 0  
  | n ->  
    let (s1,s2) = split (n/2) s in  
    let (n1, n2) = both (reduce f base) s1  
                    (reduce f base) s2 in  
    f n1 n2
```

```
let sum s = reduce (+) 0 s
```

A little more general

```
let rec mapreduce (inject: 'a -> 'b)
                  (combine:'b -> 'b -> 'b)
                  (base:'b)
                  (s:'a seq) =
  match length s with
  | 0 -> base
  | 1 -> inject (nth s 0)
  | n ->
    let (s1,s2) = split (n/2) s in
    let (n1, n2) = both
      (mapreduce inject combine base) s1
      (mapreduce inject combine base) s2 in
    combine n1 n2
```

A little more general

```
let rec mapreduce (inject: 'a -> 'b)
                  (combine:'b -> 'b -> 'b)
                  (base:'b)
                  (s:'a seq) =
  match length s with
  | 0 -> base
  | 1 -> inject (nth s 0)
  | n ->
    let (s1,s2) = split (n/2) s in
    let (n1, n2) = both
                (mapreduce inject combine base) s1
                (mapreduce inject combine base) s2 in
    combine n1 n2
```

```
let average s =
  let (count, total) =
    mapreduce (fun x -> (1,x))
              (fun (c1,t1) (c2,t2) -> (c1+c2, t1 + t2))
              (0,0) s in
  if count = 0 then 0 else total / count
```

**DON'T PARALLELIZE
AT TOO FINE A GRAIN**

BALANCED PARENTHESES

The Balanced Parentheses Problem

Consider the problem of determining whether a sequence of parentheses is balanced or not. For example:

- balanced: `()()()`
- not balanced: `(`
- not balanced: `)`
- not balanced: `)))`

We will try formulating a divide-and-conquer parallel algorithm to solve this problem efficiently:

```
type paren = L | R      (* L(left) or R(right) paren *)  
  
let balanced (ps : paren seq) : bool = ...
```

First, a sequential approach

fold from left to right, keep track of
of unmatched left parens



0

Warning! This solution
does not generalize to a
parallel map/reduce!

First, a sequential approach

fold from left to right, keep track of
of unmatched left parens



0 1

Warning! This solution
does not generalize to a
parallel map/reduce!

First, a sequential approach

fold from left to right, keep track of
of unmatched left parens

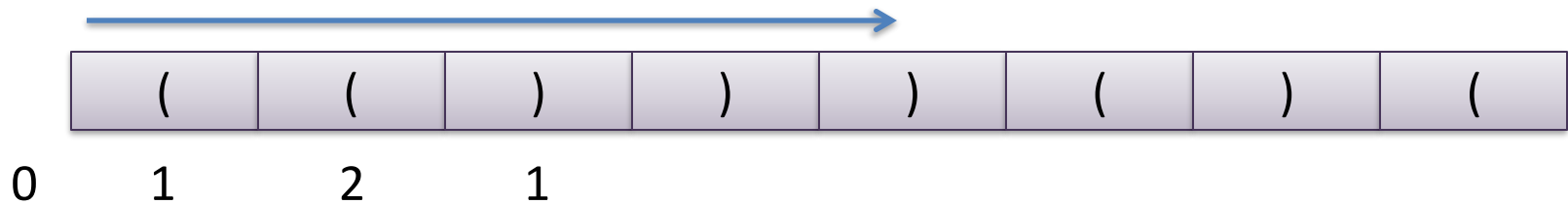


0 1 2

Warning! This solution
does not generalize to a
parallel map/reduce!

First, a sequential approach

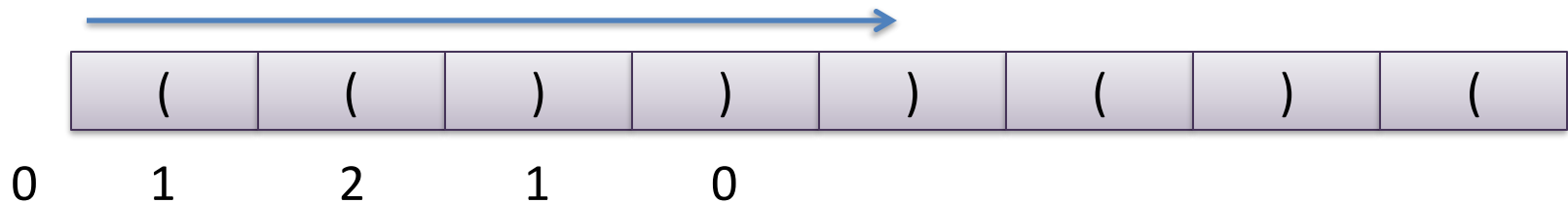
fold from left to right, keep track of
of unmatched left parens



Warning! This solution
does not generalize to a
parallel map/reduce!

First, a sequential approach

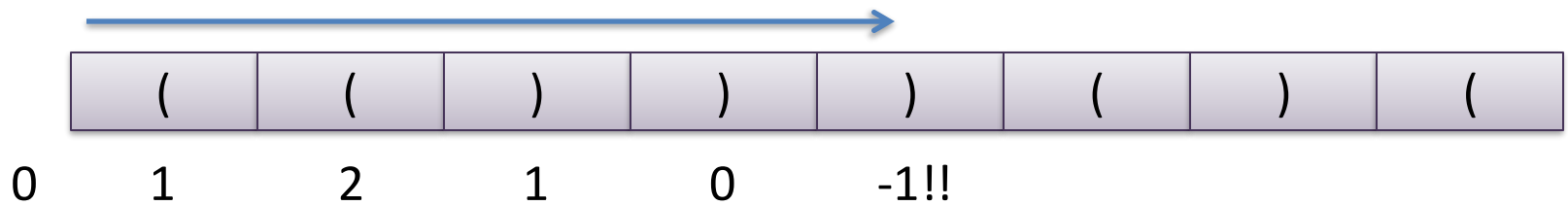
fold from left to right, keep track of
of unmatched left parens



Warning! This solution
does not generalize to a
parallel map/reduce!

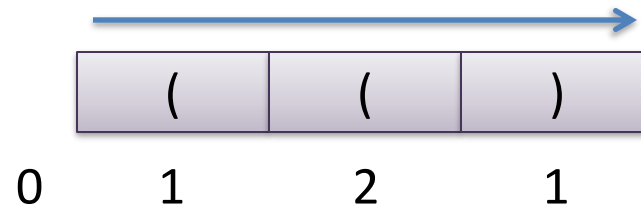
First, a sequential approach

fold from left to right, keep track of
of unmatched left parens



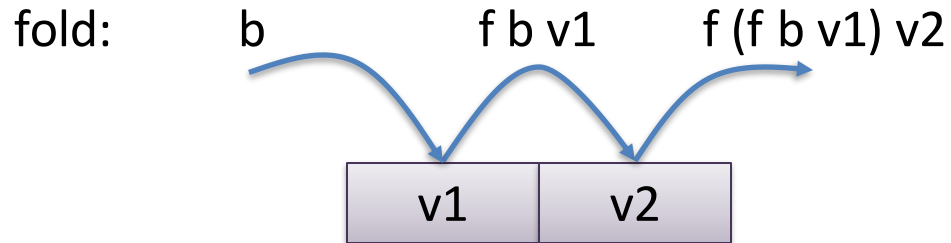
too many right parens
indicates no match

First, a sequential approach



if you reach the end of the end of the sequence, you should have no unmatched left parens

Easily Coded Using a Fold



```
let rec fold f b s =  
  let rec aux n accum =  
    if n >= length s then  
      accum  
    else  
      aux (n+1) (f (nth s n) accum)  
  in  
  aux 0 b
```

Easily Coded Using a Fold

```
(* check to see if we have too many unmatched R parens
```

```
    so_far : number of unmatched parens so far  
              or None if we have seen too many R parens
```

```
*)
```

```
let check (p:paren) (so_far:int option) : int option =  
  match (p, so_far) with  
  | (_, None) -> None  
  | (L, Some c) -> Some (c+1)  
  | (R, Some 0) -> None          (* violation detected *)  
  | (R, Some c) -> Some (c-1)
```

Easily Coded Using a Fold

```
let fold f base s = ...  
  
let check so_far s = ...  
  
let balanced (s: paren seq) : bool =  
  match fold check (Some 0) s with  
  | Some 0 -> true  
  | (None | Some n) -> false
```

That was easy enough. But the “check” function is not associative, that means it can’t be used in a parallel “reduce”.

That’s what I was warning about!

Parallel Version

Key insights

- if you find () in a sequence, you can delete it without changing the balance

Parallel Version

Key insights

- if you find () in a sequence, you can delete it without changing the balance
- if you have deleted all of the pairs (), you are left with:
 -))) ... j ...))) (((... k ... (((

Parallel Version

Key insights

- if you find () in a sequence, you can delete it without changing the balance
- if you have deleted all of the pairs (), you are left with:
 -))) ... j ...))) (((... k ... (((

For divide-and-conquer, splitting a sequence of parens is easy

Parallel Version

Key insights

- if you find () in a sequence, you can delete it without changing the balance
- if you have deleted all of the pairs (), you are left with:
 -))) ... j ...))) (((... k ... (((

For divide-and-conquer, splitting a sequence of parens is easy

Combining two sequences where we have deleted all ():

-))) ... j ...))) (((... k ... ((())) ... x ...))) (((... y ... (((

Parallel Version

Key insights

- if you find () in a sequence, you can delete it without changing the balance
- if you have deleted all of the pairs (), you are left with:
 -))) ... j ...))) (((... k ... (((

For divide-and-conquer, splitting a sequence of parens is easy

Combining two sequences where we have deleted all ():

-))) ... j ...))) (((... k ... ((())) ... x ...))) (((... y ... (((
- if $x \geq k$ then))) ... j ...)))))) ... $x - k$...))) (((... y ... (((

Parallel Version


Key insights

- if you find $()$ in a sequence, you can delete it without changing the balance
- if you have deleted all of the pairs $()$, you are left with:
 - $))) \dots j \dots))) (((\dots k \dots ((($

For divide-and-conquer, splitting a sequence of parens is easy

Combining two sequences where we have deleted all $()$:

– $))) \dots j \dots))) (((\dots k \dots (((\text{))) } \dots x \dots \text{))) (((\dots y \dots ((($



– if $x \geq k$ then $))) \dots j \dots))) \text{))) } \dots x - k \dots \text{))) (((\dots y \dots ((($

– if $x \leq k$ then $))) \dots j \dots))) (((\dots k - x \dots (((\text{ (((} \dots y \dots ((($

Parallel Matcher

(* delete all () and return the (j, k) corresponding to:

```
))) ... j ... ))) ((( ... k ... (((  
*)
```

```
let rec matcher s =  
  match length s with  
  | 0 -> (0, 0)  
  | 1 -> (match nth s 0 with  
          | L -> (0, 1)  
          | R -> (1, 0))  
  | n ->  
    let (left, right) = split (n/2) s in  
    let ((j, k), (x, y)) = both matcher left  
                          matcher right in  
    if x > k  
    then (j + (x - k), y)  
    else (j, (k - x) + y)
```

```
))) ... j ... ))) ((( ... k ... (((  
          ))) ... x ... ))) ((( ... y ... (((
```

Parallel Balance

```
(* *)  
let matcher s = ...  
  
(* true if s is a sequence of balanced parens *)  
let balanced s =  
  match matcher s with  
  | (0, 0) -> true  
  | (j,k) -> false
```

Parallel Matcher

(* delete all () and return the (j, k) corresponding to:

```
))) ... j ... ))) ((( ... k ... (((  
*)
```

```
let rec matcher s =  
  match length s with  
    0 -> (0, 0)  
  | 1 -> (match nth s 0 with  
          | L -> (0, 1)  
          | R -> (1, 0))  
  | n ->  
    let (left, right) = split (n/2) s in  
    let ((j, k), (x, y)) = both matcher left  
                          matcher right in  
    if x > k  
    then (j + (x - k), y)  
    else (j, (k - x) + y)
```

This looks just like mapreduce!

Using a Parallel Fold

```
let rec mapreduce (inject: 'a -> 'b)
                  (combine: 'b -> 'b -> 'b)
                  (base: 'b)
                  (s: 'a seq) = ...
```

```
let inject paren =
  match paren with
  | L -> (0, 1)
  | R -> (1, 0)
```

```
let combine (j,k) (x,y) =
  if x > k then (j + (x - k), y)
  else          (j, (k - x) + y)
```

```
let balanced s =
  match mapreduce inject combine (0,0) s with
  | (0, 0) -> true
  | (i,j)  -> false
```

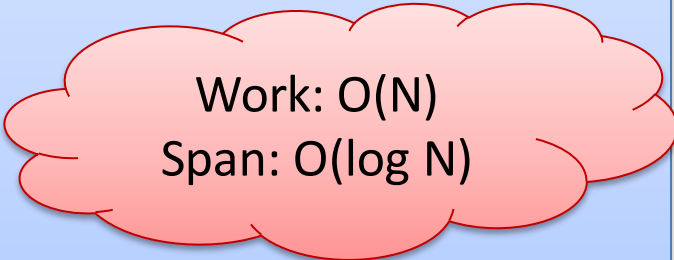
Using a Parallel Fold

```
let rec mapreduce (inject: 'a -> 'b)
                  (combine: 'b -> 'b -> 'b)
                  (base: 'b)
                  (s: 'a seq) = ...
```

```
let inject paren =
  match paren with
  | L -> (0, 1)
  | R -> (1, 0)
```

```
let combine (j,k) (x,y) =
  if x > k then (j + (x - k), y)
  else          (j, (k - x) + y)
```

```
let balanced s =
  match mapreduce inject combine (0,0) s with
  | (0, 0) -> true
  | (i,j)  -> false
```



Work: $O(N)$
Span: $O(\log N)$

Using a Parallel Fold

```
let rec mapreduce (inject: 'a -> 'b)
                  (combine: 'b -> 'b -> 'b)
                  (base: 'b)
                  (s: 'a seq) = ...
```

```
let inject paren =
  match paren with
  | L -> (0, 1)
  | R -> (1, 0)
```

For correctness,
check the associativity
of combine

also check:
combine base (i,j) == (i, j)

```
let combine (j,k) (x,y) =
  if x > k then (j + (x - k), y)
  else          (j, (k - x) + y)
```

```
let balanced s =
  match mapreduce inject combine (0,0) s with
  | (0, 0) -> true
  | (i,j) -> false
```

Summary

Parallel complexity can be described in terms of work and span

Folds and reduces are easily coded as parallel divide-and-conquer algorithms with $O(n)$ work and $O(\log n)$ span

The map-reduce paradigm, inspired by functional programming, is a winner when it comes to big-data processing (more about that in the next lecture).

Sanity checks

```
let combine (j,k) (x,y) =  
  if x > k then (j + (x - k), y)  
  else          (j, (k - x) + y)  
  
base = (0,0)
```

check the associativity
of combine

also check:
combine base (i,j) == (i, j)

Prove for yourself:

$\text{combine} (\text{combine} (j,k) (x,y)) (a,b) = \text{combine} (j,k) (\text{combine} (x,y)(a,b))$

$\text{combine} (j,k) (0,0) = (j,k)$

$\text{combine} (0,0) (j,k) = (j,k)$

Do the reading . . .

Chapter 2, “Search Engine Indexing”

(On reserve for this course,
available at canvas.princeton.edu,
select this course, then “reserves”)

(Read also Chapter 3, “Page Rank”
so you can appreciate what you were
doing in Assignment 5 . . .)

