15-150 Fall 2013
Lecture 19

Stephen Brookes

today

parallel programming

- parallelism and functional style
- cost semantics
- Brent's Theorem and speed-ups
- sequences: an abstract type with efficient parallel operations

parallelism

- exploiting *multiple processors*
- evaluating *independent* code *simultaneously*

- low level implementation
  - *scheduling* work onto processors
- high level planning
  - designing code *abstractly*
  - without *baking in a schedule*

our approach

- design code *abstractly*
    - specify *behavioral correctness*
    - specify *asymptotic runtime (work, span)*
- reason about code *abstractly*
    - *independently* of schedule
    - *cost semantics* and *evaluation*
- You design the code
- The compiler schedules the work

functional benefits

- No side effects, so *evaluation order* doesn't affect *behavioral correctness*
- Can build *abstract types* that support efficient *parallel-friendly* operations
- Can use *work* and *span* to predict how *parallelizable* our code is
- Work and span are *independent* of scheduling details

caveat

- In practice, it's hard to achieve speed-up
- Current languages don't make it easy to implement good scheduling strategies
- Problems include:
  - scheduling overhead
  - locality of data (cache problems)
  - runtime sensitive to scheduling choices

why bother?

- It's good to learn to think *abstractly* first and figure out details later
  - Focus on *data dependencies* when you design your code
- Our thesis: this approach to parallelism will *prevail*...

(plus, 15-210 builds on these ideas...)

cost semantics

- We've already introduced *work* and *span*
- *Work* estimates the *sequential* running time on a *single* processor
- *Span* takes account of data dependency, estimates the *parallel* running time with *unlimited* processors
  - *critical path length* of computation

cost semantics

- We showed how to calculate *work* and *span* for *recursive functions* with ***recurrence relations***
- Now we introduce ***cost graphs***, another way to deal with work and span
- Cost graphs also allow us to talk about *schedules*...
- ... and the potential for *speed-up*

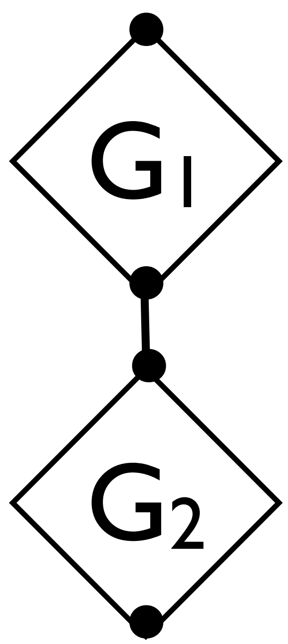
cost graphs

A *cost graph* is a *series-parallel graph*

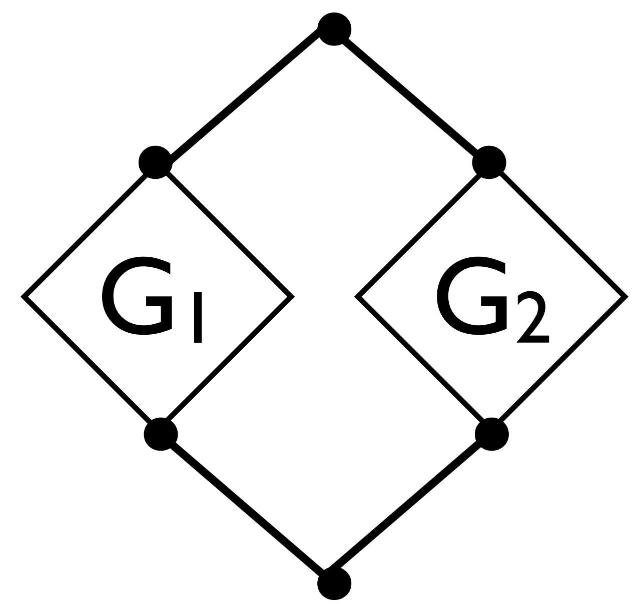
- *directed* graph, with *source* and *sink*
- nodes represent *units of work* (constant time)
- edges represent *data dependencies*
- branching indicates *potential parallelism*

cost graphs

- a single node



sequential
composition



parallel
composition

work and span

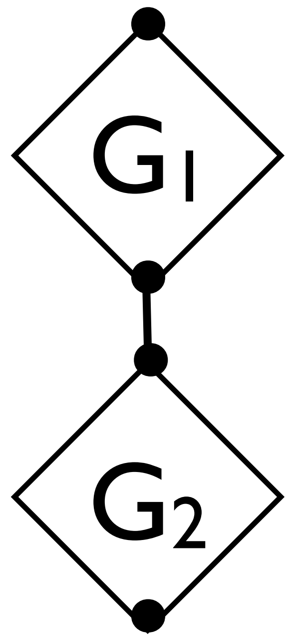
of a cost graph

- The ***work*** is the *number of nodes*
- The ***span*** is the *length of the longest path* from *source* to *sink*

span(G) $\leq$ ***work***(G)

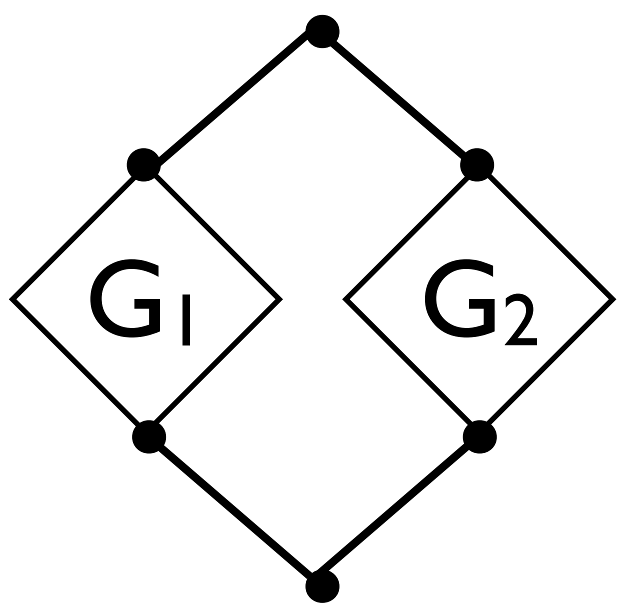
work

work



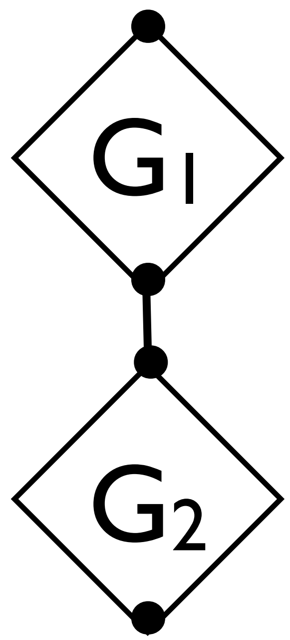
$=$ work G_1 + work G_2 + c

work

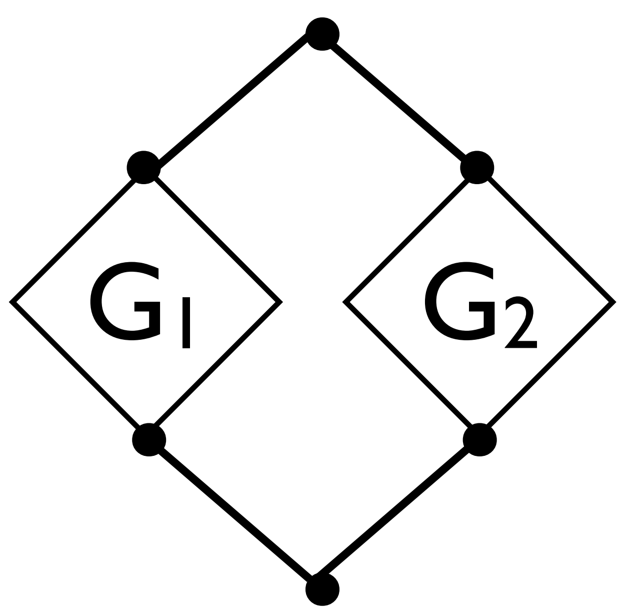


$=$ work G_1 + work G_2 + c

span

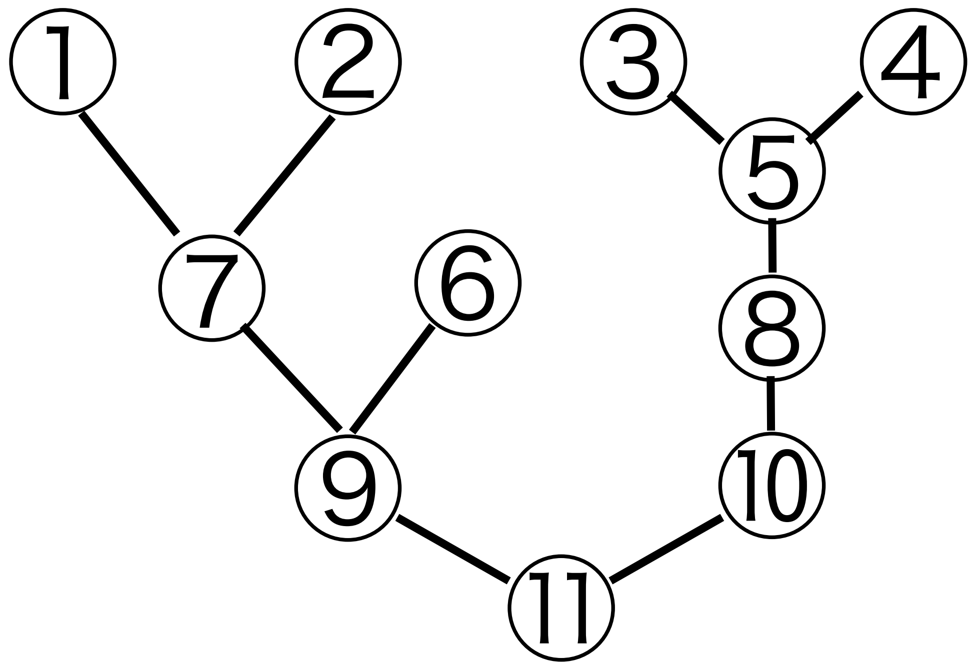
span 

$= \text{span } G_1 + \text{span } G_2 + c$

span 

$= \max(\text{span } G_1 \text{ , span } G_2) + c$

example



work = 11 (number of nodes)
span = 4 (longest path length)

using graphs

- Every *expression* can be given a *cost graph*
- Calculate the *work* and *span* using the graph
  - These are *asymptotically* the same as the work and span derived from *recurrence relations*

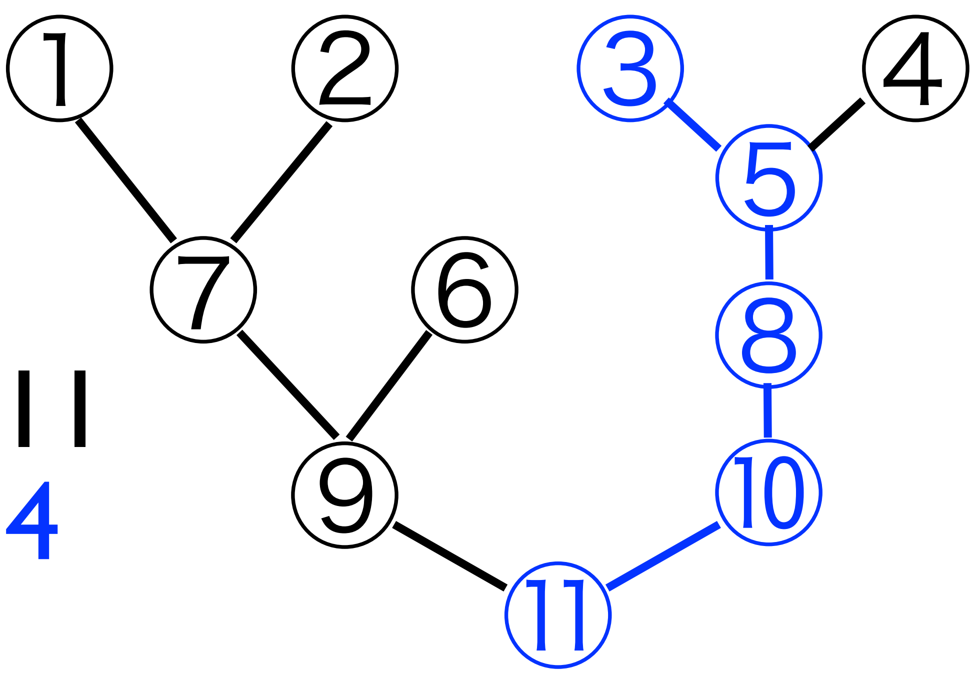
But what do work and span
tell us about the
actual running time?

scheduling

assign units of work to processors
respecting data dependency

- Work: number of steps taken by sequential scheduler on a single processor
- Span: number of steps taken by an optimal parallel scheduler with unlimited processes

w = 11
s = 4



(i) ① ② ⑥ ③ ④
(ii) ⑦ ⑤
(iii) ⑨ ⑧
(iv) ⑩
(v) ⑪

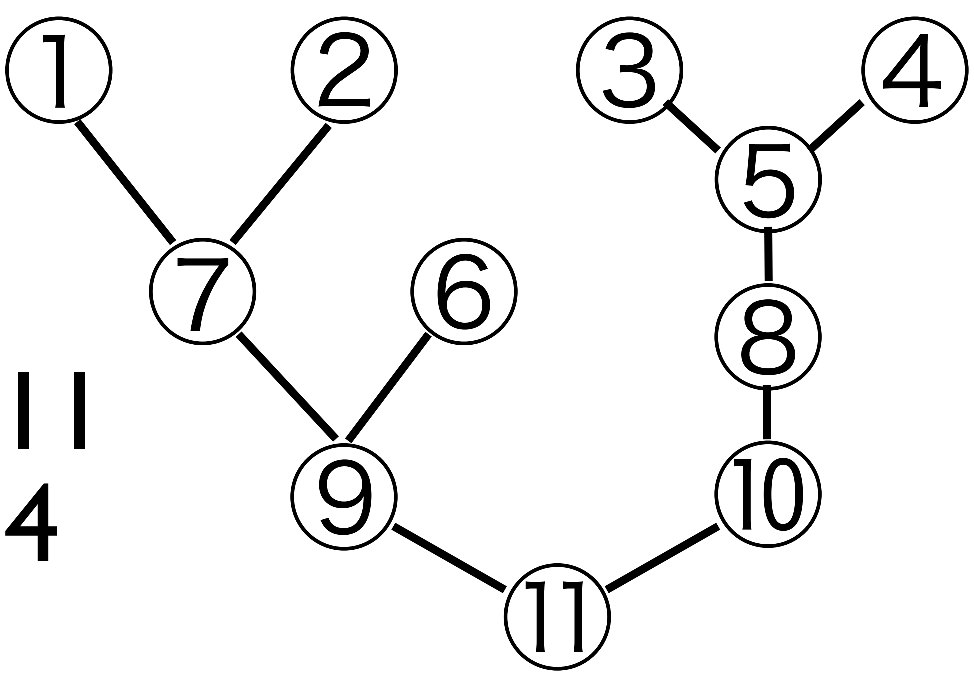
an optimal parallel schedule

(5 rounds, or 4 steps)

(uses 5 processors)

example

What if there are only 2 processors?



w = 11
s = 4

(i)	1	2
(ii)	3	4
(iii)	7	5
(iv)	6	8
(v)	9	10
(vi)		11

a best schedule for 2 processors

(6 rounds,
5 steps)

2 processors
cannot do the job as fast as 5 processors (!)

Brent's Theorem

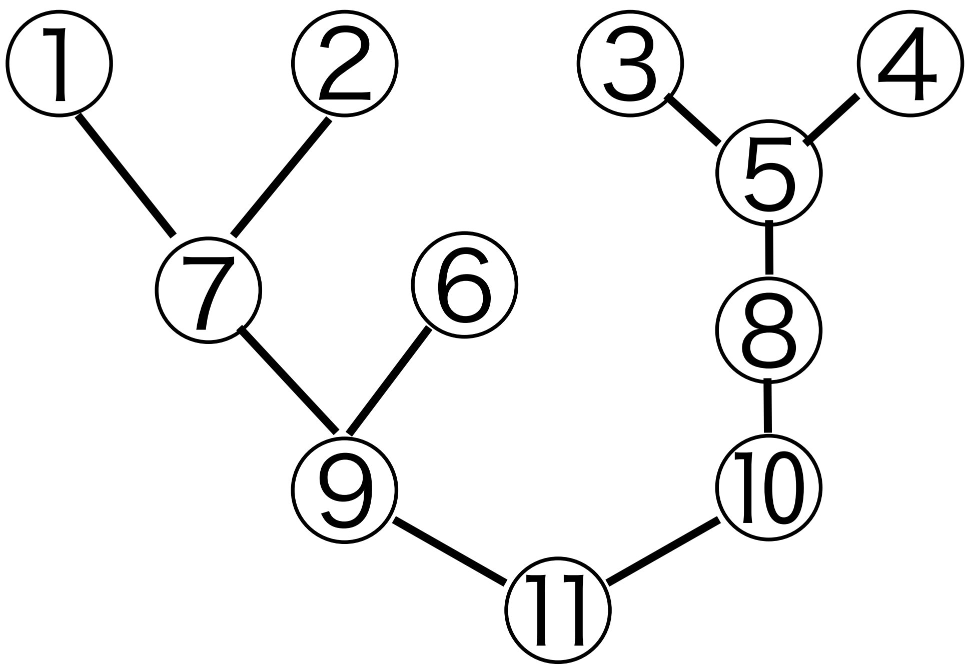
An expression with *work* **w** and *span* **s**
can be evaluated on a **p**-processor machine
in time O(max(**w**/**p**, **s**)).

Optimal schedule using **p** processors:
Do (up to) **p** units of work each round
Total work to do is **w**
Needs at least **s** steps

What's the significance of
the smallest **p** such that
w/**p** $\leq$ **s** ?

Using more than
this many processors
won't yield any speed-up

example



min {**p** | 11/**p** ≤ 4} is 3

work = 11
span = 4

(i) ① ③ ④
(ii) ② ⑥ ⑤
(iii) ⑦ ⑧
(iv) ⑨ ⑩
(v) ⑪

a best schedule for 3 processors

(5 rounds,
4 steps)

3 processors
can do the job as fast as 5(!)

next

- Exploiting parallelism in ML
- A signature for *parallel collections*
- *Cost analysis* of implementations
- *Cost benefits* of parallel algorithm design

sequences

```
signature SEQ =
sig
  type 'a seq
  exception Range
  val tabulate : (int -> 'a) -> int -> 'a seq
  val length : 'a seq -> int
  val nth : int -> 'a seq -> 'a
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val reduce : (('a * 'a) -> 'a) -> 'a -> 'a seq -> 'a
  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
end
```

implementations

- Many ways to implement the signature
  - lists, balanced trees, arrays, ...
- For each one, can give a *cost analysis*
- There may be implementation *trade-offs*
  - *arrays: item access is O(1)*
  - *trees: item access is O(log n)*

Seq : SEQ

- An abstract parameterized type of *sequences*
- Think of a sequence as a *parallel collection*
- With *parallel-friendly* operations
  - *constant-time* access to items
  - *efficient* map and reduce

We'll work today with an implementation
Seq : SEQ
based on *vectors*

notation

- We use math notation like

$$\langle v_1, ..., v_n \rangle$$

$$\langle v_0, ..., v_{n-1} \rangle$$

$$\langle \ \rangle$$

for sequence values

$$\langle 1, 2, 4, 8 \rangle : \text{int seq}$$

equality

- Two sequence values are (extensionally) *equal* iff they have the same length and their items are equal

$$\langle v_1, ..., v_n \rangle = \langle u_1, ..., u_m \rangle$$

if and only if

$$n = m \text{ and for all } i, v_i = u_i$$

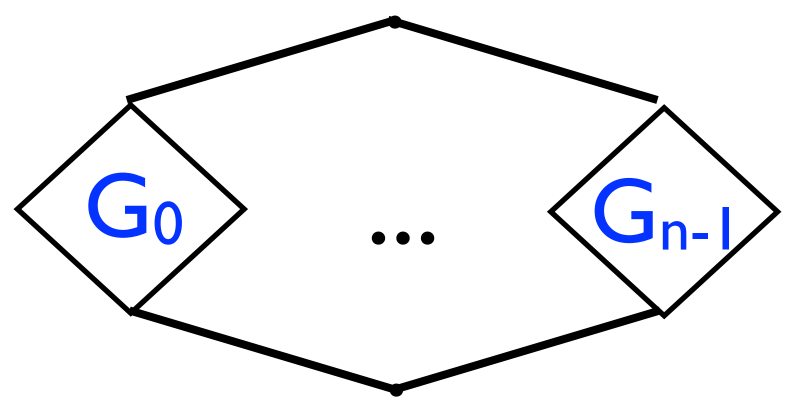
operations

- For each operation in the signature SEQ we specify the (extensional) behavior of the operation implemented in Seq and discuss its cost semantics
- Other structures with the same signature may implement the operations with *different* work and span profile
- Learn to choose wisely!

tabulate

tabulate f n = ⟨f 0, ..., f(n-1)⟩

- If G_i is cost graph for f(i), the cost graph for tabulate f n is



work?
span?

If f is O(1), the work for tabulate f n is O(n)

If f is O(1), the span for tabulate f n is O(1)

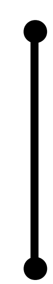
examples

- tabulate (**fn** x:int => x) 6
- tabulate (**fn** x:int => x*x) 6
- tabulate (**fn** _ => **raise** Range) 0

length

length $\langle v_1, ..., v_n \rangle = n$

- Work is $O(1)$
- Span is $O(1)$
- Cost graph is



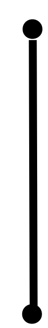
Contrast: List.length $[v_1,...,v_n] = n$

work, span $O(n)$

nth

$$
\begin{aligned}
\text{nth } i\ \langle v_0, \ldots, v_{n-1} \rangle &= v_i && \text{if } 0 \leq i < n \\
&= \textbf{raise}\ \text{Range} && \text{otherwise}
\end{aligned}
$$

- Work is O(1)
- Span is O(1)
- Cost graph is



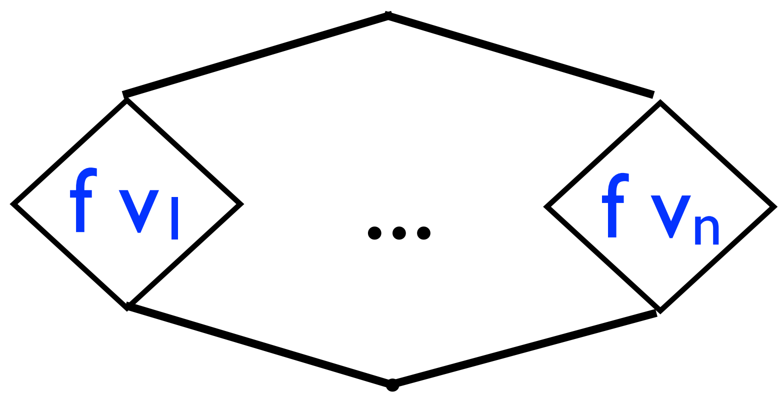
Seq *provides*

constant-time access to items

map

$\text{map } f \langle v_1, \ldots, v_n \rangle = \langle f\ v_1, \ldots, f\ v_n \rangle$

cost graph



- If f is constant time, *work* O(n)
  span O(1)

(contrast with List.map)

reduce

reduce should be used to *combine* a sequence using an *associative* function g with *identity element* z

- g : t * t -> t is ***associative*** iff for all x_1, x_2, x_3:t
  $g(x_1, g(x_2, x_3)) = g(g(x_1, x_2), x_3)$
- z is an *identity* for g iff for all x:t, $g(x,z) = x$

- When g is associative and z an identity we write

$$v_1 \ g \ v_2 \ g \ \ldots \ g \ v_n \ g \ z$$

for the result of combining $v_1, v_2, \ldots, v_n$, z using g

$$\text{reduce } g \ z \ \langle v_1, \ldots, v_n \rangle = v_1 \ g \ v_2 \ g \ \ldots \ g \ v_n \ g \ z$$
$$= v_1 \ g \ v_2 \ g \ \ldots \ g \ v_n$$

reduce

- When g is associative and z is an identity

  $\text{reduce } g\ z\ \langle v_1, \ldots, v_n\rangle = v_1\ g\ v_2\ g \ldots g\ v_n\ g\ z$

- If g is constant time,

  $\text{reduce } g\ z\ \langle v_1, \ldots, v_n\rangle$

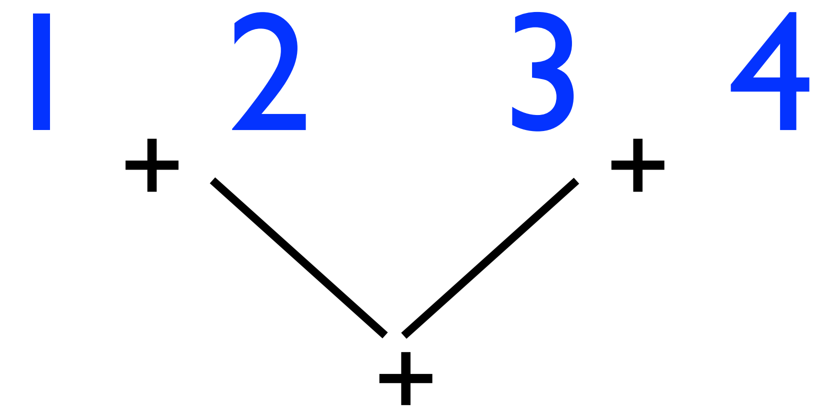
  has *work* O(n)

  and *span* O(log n)

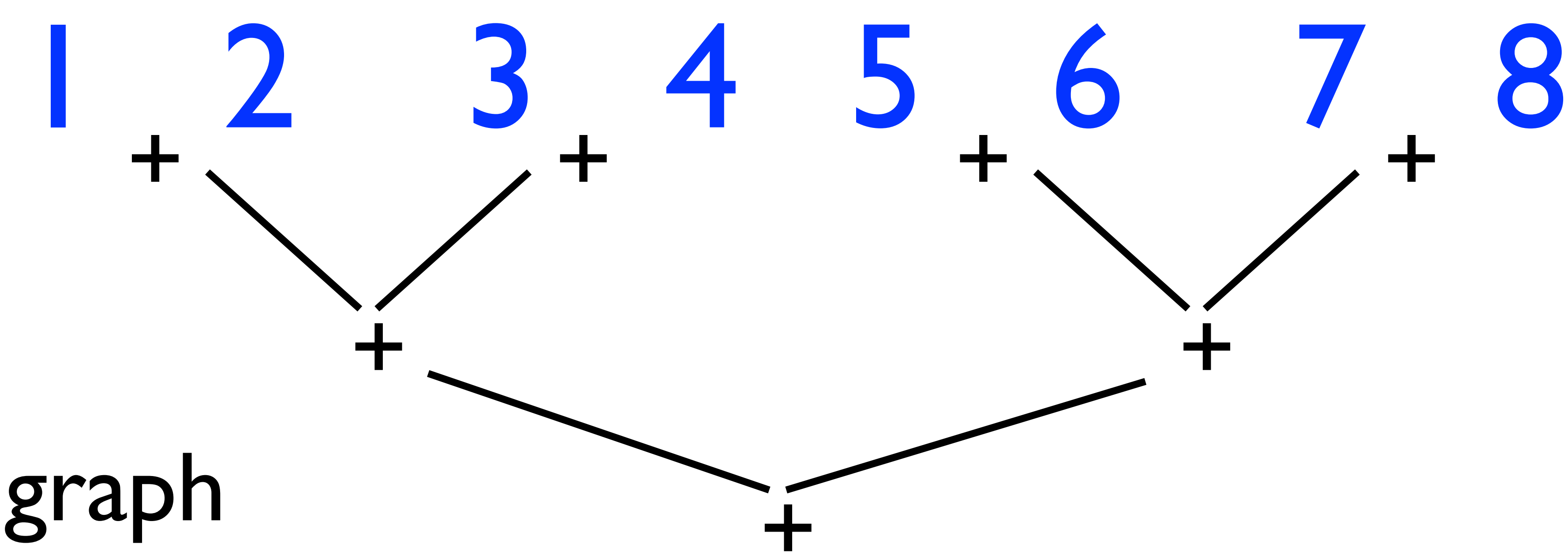
(Contrast with foldr, foldl on lists)

reduce (op +) 0 ⟨1, 2, 3, 4⟩

cost graph



reduce (op +) 0 ⟨1, 2, 3, 4, 5, 6, 7, 8⟩

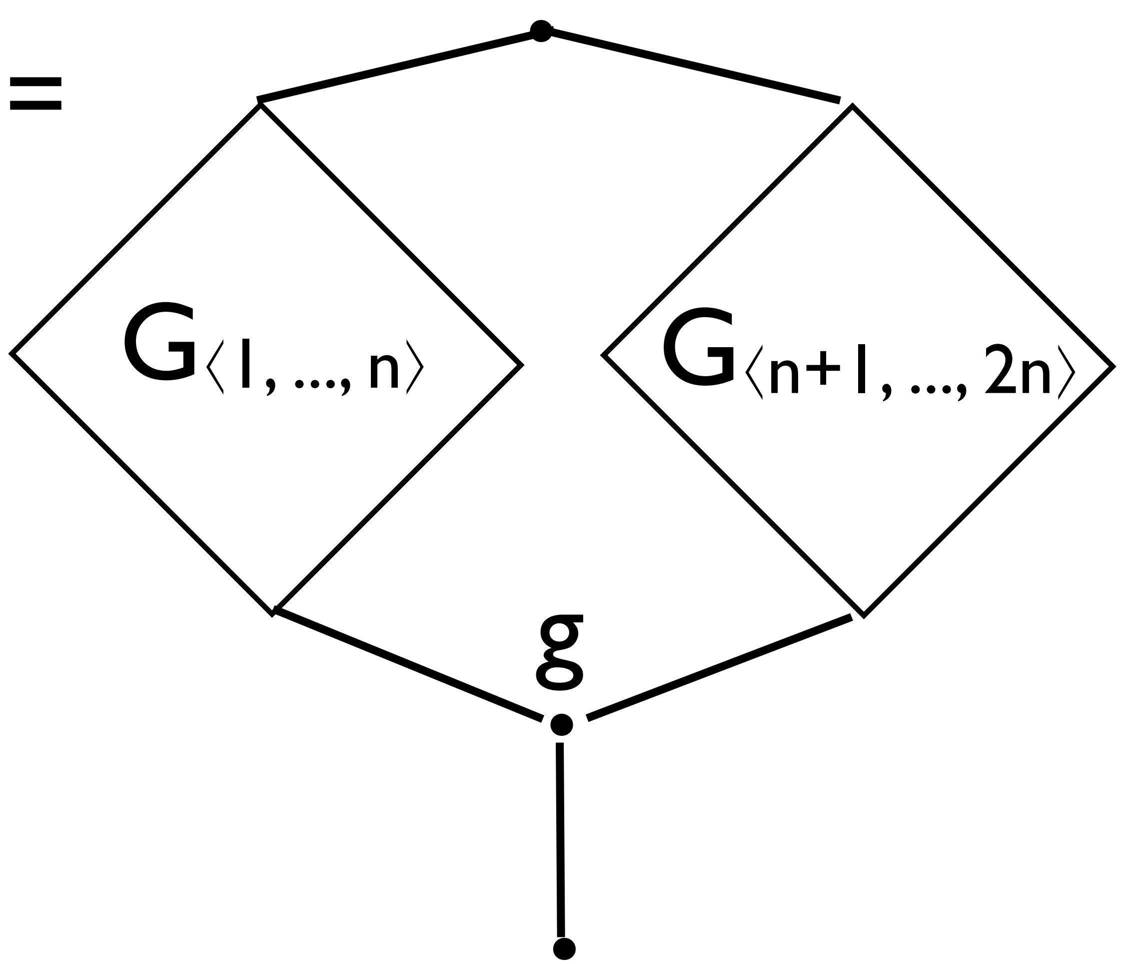


cost graph

reduce cost

reduce g z $\langle v_1, \ldots, v_{2n} \rangle$ =
g(reduce g z $\langle v_1, \ldots, v_n \rangle$, reduce g z $\langle v_{n+1}, \ldots, v_{2n} \rangle$)

$G_{\langle 1, \ldots, 2n \rangle}$ =



W(2n) = 2*W(n) + c
S(2n) = S(n) + c

W(n) is O(n)
S(n) is O($\log_2$ n)

mapreduce

- When g is associative and z is an identity,

$$\text{mapreduce } f\ z\ g\ \langle v_1, \ldots, v_n \rangle = (f\ v_1)\ g \ldots g\ (f\ v_n)\ g\ z$$

- When f, g are constant time,

mapreduce f z g $\langle v_1, \ldots, v_n \rangle$

has *work* O(n)

and *span* O(log n)

examples

```
fun sum (s : int seq) : int =
            reduce (op +) 0 s

fun count (s : int seq seq) : int =
            sum (map sum s)
```

analysis

```
fun sum (s : int seq) : int = reduce (op +) 0 s
fun count (s : int seq seq) : int = sum (map sum s)
```

- Let **s** be a value of type **int seq seq** consisting of n *rows*, each of length n
- What are the work and span for

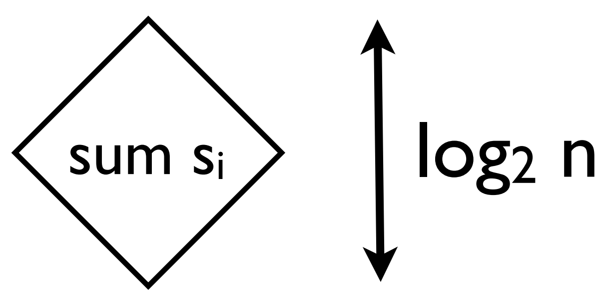
count s ?

analysis

Let $s = \langle s_1, \ldots, s_n \rangle$, $s_i = \langle x_{i1}, \ldots, x_{in} \rangle$, t_i = sum s_i

For each i, sum s_i = reduce(op +) 0 $\langle x_{i1}, \ldots, x_{in} \rangle$

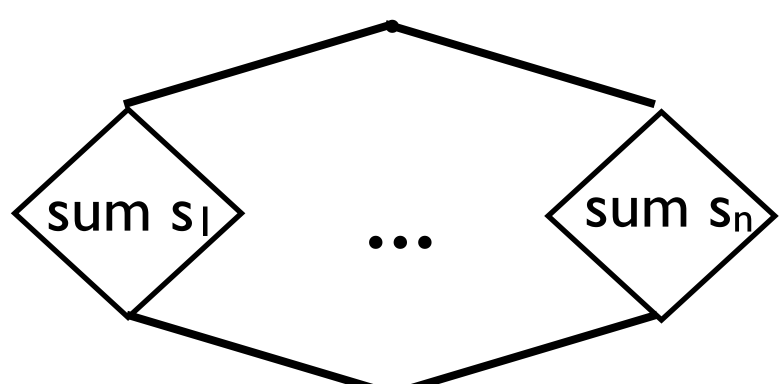
cost graph of sum s_i



work is O(n)
span is O(log n)

map sum s = $\langle$sum s_1, ..., sum $s_n\rangle$

cost graph of map sum s



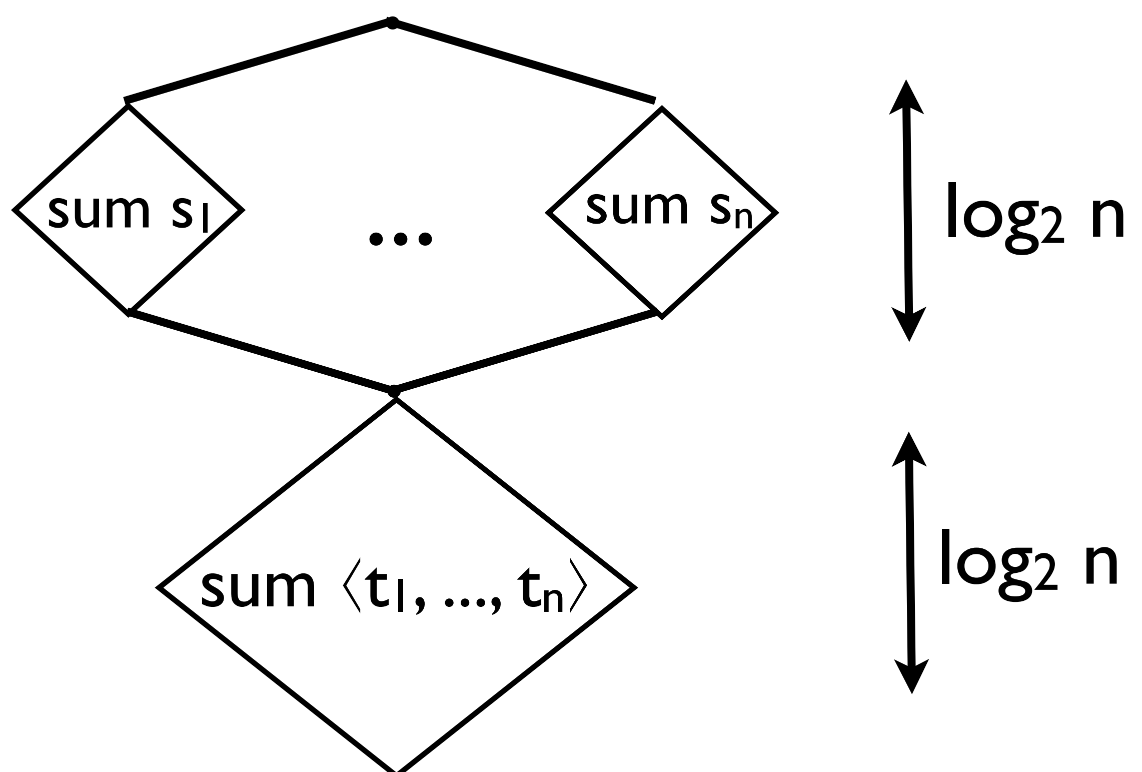
work is $O(n^2)$
span is O(log n)

analysis

Let t_i = sum s_i

count s = sum $\langle t_1, \ldots, t_n \rangle$

cost graph of
sum (map sum s)



work is $O(n^2)$
span is $O(\log n)$