Parallelism 3: Parallel Collections

COS 326
David Walker
Princeton University

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

Credits

- Material on Parallel Complexity from the last couple of lectures:
  - Blelloch, Harper, Licata (CMU, Wesleyan)

- Material on parallel prefix sum:
  - Dan Grossman, UW
  - http://homes.cs.washington.edu/~djg/teachingMaterials/spac

Last Time

Futures: A simple abstraction for parallel programming

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

  val future : ('a->'b) -> 'a -> 'b future

  val force : 'a future -> 'a
end
```

Key idea: supports equational reasoning

- force (future f x) == f x
- when f is a pure function
- reasoning about parallelism via futures is as easy as reasoning about sequential programs

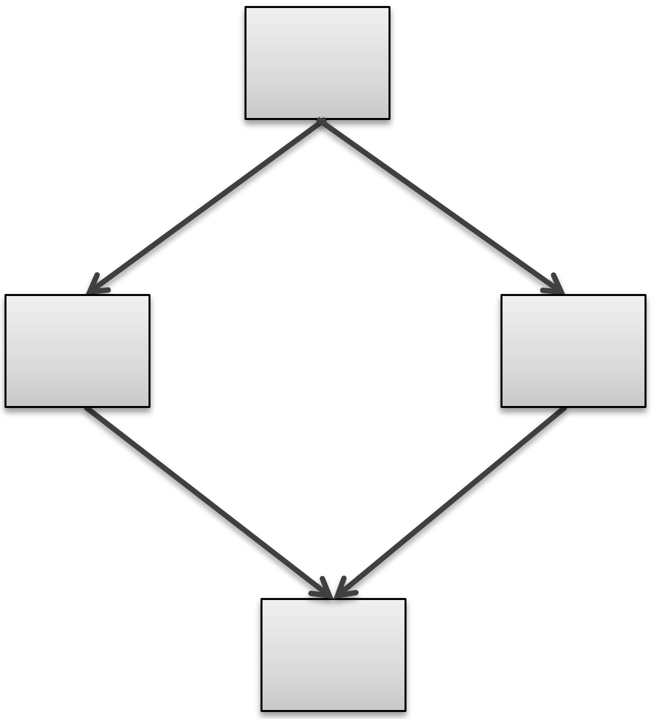
Last Time

The complexity of parallel programs

- Work: Cost of executing a program with just 1 processor
- Span: Cost of executing a program with infinite processors

We can visualize computations:

- Work: add up the blocks
- Span: length of the longest path

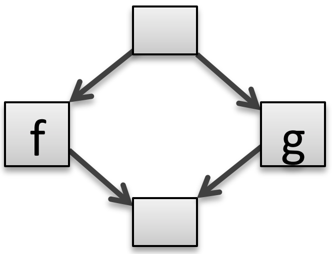


How you allocate computations to processors (ie, *scheduling*) matters, but greedy schedulers do a pretty good job and are used in practice.

Analyzing Program Complexity

Recall the combinator both f x g y

- executes f x and g y in parallel
- visually

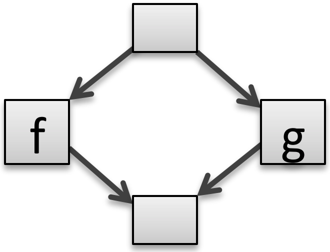


- used in divide-and-conquer parallel programming

Analyzing Program Complexity

Recall the combinator both f x g y

- executes f x and g y in parallel
- visually
- used in divide-and-conquer parallel programming



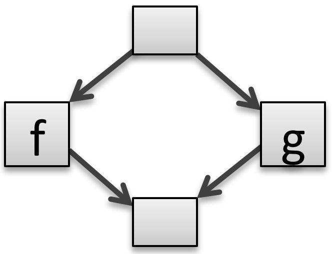
Analyzing complexity:

- Work: Just like analyzing a sequential program
  - both f x g y
  - cost = cost(f x) + cost(g y) + 1
  - mirrors summing the cost of all blocks in the diagram

Analyzing Program Complexity

Recall the combinator both f x g y

- executes f x and g y in parallel
- visually
- used in divide-and-conquer parallel programming



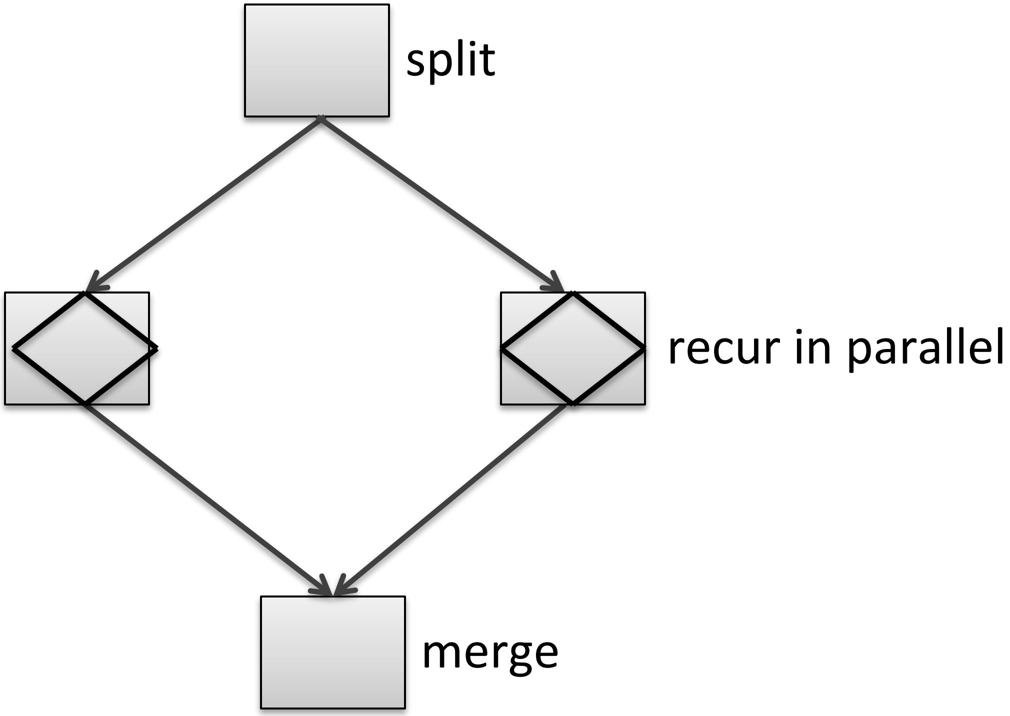
Analyzing complexity:

- Work: Just like analyzing a sequential program
  - both f x g y
  - cost = cost(f x) + cost(g y) + 1
  - mirrors summing the cost of all blocks in the diagram
- Span: Also similar to analyzing a sequential program
  - with one key difference
  - both f x g y
  - cost = max (cost(f x), cost(g y)) + 1
  - mirrors finding the length of the longest path through the diagram

COMPLEXITY OF PARALLEL PROGRAMS

Divide-and-Conquer Parallel Algorithms

- Split your input in 2 or more subproblems
- Solve the subproblems recursively in parallel
- Combine the results to solve the overall problem



Parallel Map

```
let rec map f l =
  match l with
     []   -> []
   | h1::t1 ->
      let (h2,t2)  =
        both f hd
             (map f) tail
      in
      h2::t2
```

Parallel Map

```
let rec map f l =
  match l with
    []   -> []
  | h1::t1 ->
     let (h2,t2)   =
       both f hd
            (map f) tail
     in
     h2::t2
```

Assume function f takes constant C span,
Assume input list of size n,
work_map(n) = B + (C + work_map(n-1))
 = (B+C)*n

Parallel Map

```
let rec map f l =
  match l with
     []   -> []
  | h1::t1 ->
       let (h2,t2)   =
         both f hd
              (map f) tail
       in
       h2::t2
```

Assume function f takes constant C span,
Assume input list of size n,

$$\begin{aligned} span\_map(n) &= B + \max(C, span\_map(n-1)) \\ &\sim= B + span\_map(n-1) \qquad \text{(if } B*n >> C\text{)} \\ &= B*n \end{aligned}$$

Parallel Map

```
let rec map f l =
  match l with
    []   -> []
  | h1::t1 ->
     let (h2,t2)   =
       both f hd
            (map f) tail
     in
     h2::t2
```

work_map(n) = (B+C)*n

span_map(n) = B*n

parallelism(n) = work_map(n)/span_map(n)

= (B+C)*n/B*n

~= C

Parallel Map

```
let rec map f l =
  match l with
    []   -> []
  | h1::t1 ->
     let (h2,t2)   =
       both f hd
             (map f) tail
     in
     h2::t2
```

we can speed the algorithm up by a small fixed constant, but that won't help us process big lists

work_map(n) = (B+C)*n
span_map(n) = B*n
parallelism(n) = work_map(n)/span_map(n)
= (B+C)*n/B*n
~= C

Parallel Map

```
let rec map f l =
  match l with
    []   -> []
  | h1::t1 ->
      let (h2,t2)   =
        both f hd
             (map f) tail
      in
      h2::t2
```

work_map(n) = (B+C)*n
span_map(n) = B*n
parallelism(n) = work_map(n)/span_map(n)
= (B+C)*n/B*n
~= C

we can only make use of a (small) constant number of machines

Parallel Map

```
let rec map f l =
  match l with
     []   -> []
   | h1::t1 ->
       let (h2,t2)   =
         both f hd
```

Problem: splitting and merging lists take linear time – can't get good speedups

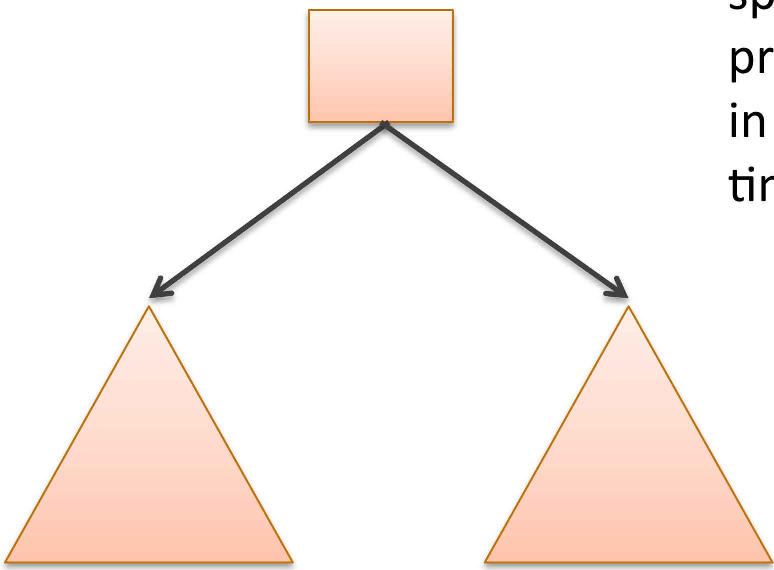
Problem: cutting a list in half takes at least time proportional to n/2

Problem: stitching 2 lists together of size n/2 takes n/2 time

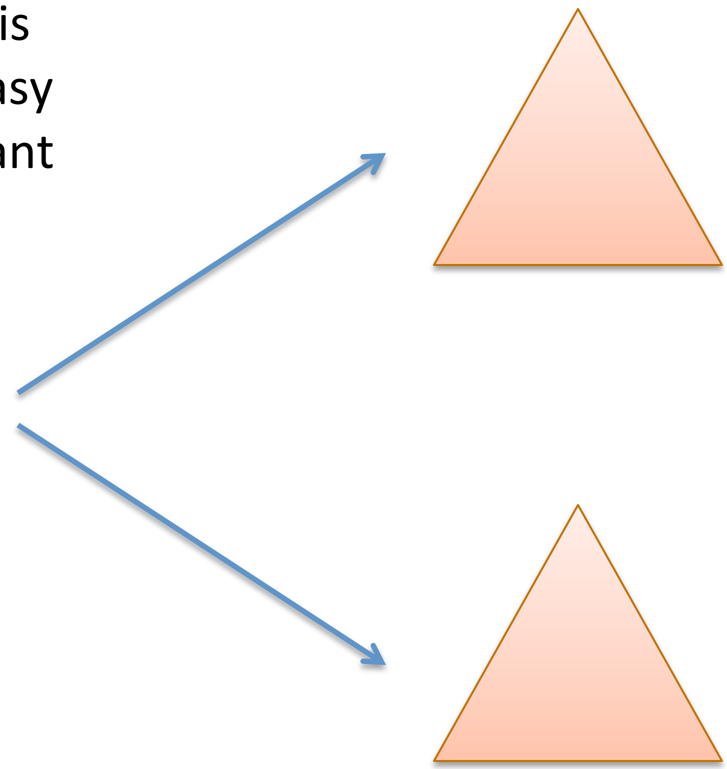
Conclusion: lists are a bad data structure to choose for divide-and conquer parallel programming

Complexity

Consider balanced trees:



splitting is pretty easy in constant time



merging is harder, but can be done in poly-log time

Parallel TreeMap

```
type tree = Empty | Node of tree * int * tree

let rec treemap f l =
  match t with
    Empty  -> Empty
  | Node(left, i, right) ->

      let j = future f i in
      let left2, right2 =
        both (treemap f) left
             (treemap f) right
      in
      Node (left2, force j, right2)
```

Parallel TreeMap

```
type tree = Empty | Node of tree * int * tree

let rec treemap f l =
  match t with
    Empty  -> Empty
  | Node(left, i, right) ->

       let j = future f i in
       let left2, right2 =
         both (treemap f) left
              (treemap f) right
       in
       Node (left2, force j, right2)
```

<u>Assume balanced tree of size n, executing f costs C:</u>

work(n) = work(f i) + work(n/2) + work(n/2)) + B

Parallel TreeMap

```
type tree = Empty | Node of tree * int * tree

let rec treemap f l =
  match t with
    Empty  -> Empty
  | Node(left, i, right) ->

       let j = future f i in
       let left2, right2 =
         both (treemap f) left
              (treemap f) right
       in
       Node (left2, force j, right2)
```

<u>Assume balanced tree of size n, executing f costs C:</u>

work(n) = work(f i) + work(n/2) + work(n/2)) + B

= C + 2*work(n/2) + B

= (C+B) * n

Parallel TreeMap

```
type tree = Empty | Node of tree * int * tree

let rec treemap f l =
  match t with
    Empty  -> Empty
  | Node(left, i, right) ->

       let j = future f i in
       let left2, right2 =
         both (treemap f) left
              (treemap f) right
       in
       Node (left2, force j, right2)
```

Assume balanced tree of size n, executing f costs C:

work(n) = work(f i) + work(n/2) + work(n/2)) + B
= C + 2*work(n/2) + B
= (C+B) * n

roughly the same work as listmap

Parallel TreeMap

```
type tree = Empty | Node of tree * int * tree

let rec treemap f l =
  match t with
    Empty  -> Empty
  | Node(left, i, right) ->

       let j = future f i in
       let left2, right2 =
         both (treemap f) left
              (treemap f) right
       in
       Node (left2, force j, right2)
```

<u>Assume balanced tree of size n, executing f costs C:</u>

span(n) = max (span(f i), max(span(n/2), span(n/2)) + B

Parallel TreeMap

```
type tree = Empty | Node of tree * int * tree

let rec treemap f l =
  match t with
    Empty  -> Empty
  | Node(left, i, right) ->

       let j = future f i in
       let left2, right2 =
         both (treemap f) left
              (treemap f) right
       in
       Node (left2, force j, right2)
```

Assume balanced tree of size n, executing f costs C:

span(n) = max (span(f i), max(span(n/2), span(n/2)) + B

= max(C, max(span(n/2), span(n/2))) + B

Parallel TreeMap

```
type tree = Empty | Node of tree * int * tree

let rec treemap f l =
  match t with
    Empty  -> Empty
  | Node(left, i, right) ->

       let j = future f i in
       let left2, right2 =
         both (treemap f) left
              (treemap f) right
       in
       Node (left2, force j, right2)
```

Assume balanced tree of size n, executing f costs C:

span(n) = max (span(f i), max(span(n/2), span(n/2)) + B

= max(C, max(span(n/2), span(n/2))) + B

= span(n/2) + B

Parallel TreeMap

```
type tree = Empty | Node of tree * int * tree

let rec treemap f l =
  match t with
    Empty  -> Empty
  | Node(left, i, right) ->

       let j = future f i in
       let left2, right2 =
         both (treemap f) left
              (treemap f) right
       in
       Node (left2, force j, right2)
```

<u>Assume balanced tree of size n, executing f costs C:</u>

span(n) = max (span(f i), max(span(n/2), span(n/2)) + B
= max(C, max(span(n/2), span(n/2))) + B
= span(n/2) + B
= B log n

Parallel TreeMap

```
type tree = Empty | Node of tree * int * tree

let rec treemap f l =
  match t with
    Empty  -> Empty
  | Node(left, i, right) ->

       let j = future f i in
       let left2, right2 =
         both (treemap f) left
              (treemap f) right
       in
       Node (left2, force j, right2)
```

Assume balanced tree of size n, executing f costs C:

span(n) = max (span(f i), max(span(n/2), span(n/2)) + B
= max(C, max(span(n/2), span(n/2))) + B
= span(n/2) + B
= B log n

asymptotically better than for lists

Lists vs Trees

Lists:
work(n) = (B+C)*n
span(n) = B*n
parallelism(n) = work(n)/span(n)
~= C

Trees:
work(n) = (B+C)*n
span(n) = B log n
parallelism(n) = work(n)/span(n)
~= C n / log n

Trees or arrays, which can be split into even-sized pieces in constant time speed parallel divide-and-conquer algorithms

PARALLEL COLLECTIONS

What if you had a really big job to do?

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

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

Eg: search facebook for a friend or twitter for a tweet

To get big jobs done, we typically need to harness 1000s of computers at a time, 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

PARALLEL SEQUENCES

Parallel Sequences

Parallel sequences

< e1 , e2 , e3 , ... , en >

Operations:

- creation (called tabulate)
- indexing an element in constant span
- map
- scan -- like a fold: <u, u + e1, u + e1 + e2, ...> log n span!

Languages:

- Nesl [Blelloch]
- Data-parallel Haskell
- Lots of cool stuff in Scala too

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

Write a function that creates the sequence <0, ..., n-1>
with Span = O(1) and 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

Write a function that creates the sequence <0, ..., n-1>
with Span = O(1) and 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

Write a function such that given a sequence <v0, ..., vn-1>, maps f over each element of the sequence with Span = O(1) and 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
tabulate f n	n	1
nth i s	1	1
length s	1	1

Example

Write a function such that given a sequence <v0, ..., vn-1>, maps f over each element of the sequence with Span = O(1) and 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 -> nth s i) (length s)
```

Operations:

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

Example

Write a function such that given a sequence <v1, ..., vn-1>, reverses the sequence. with Span = O(1) and Work = O(n)

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

Operations:

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

Example

Write a function such that given a sequence <v1, ..., vn-1>, reverses the sequence. with Span = O(1) and Work = O(n)

```
(* reverse <v0, ..., vn-1> == <vn-1, ..., v0> *)
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

	Work	Span
`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`	O(N+M)	O(1)
`split : 'a seq -> int -> 'a seq * 'a seq`	O(N)	O(1)

For efficient implementations, see Blelloch's NESL project:
http://www.cs.cmu.edu/~scandal/nesl.html

A Parallel Sequence API

	Work	Span
`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`	O(N+M)	O(1)
`split : 'a seq -> int -> 'a seq * 'a seq`	**O(N)**	**O(1)**

For efficient implementations, see Blelloch's NESL project:
http://www.cs.cmu.edu/~scandal/nesl.html

Fold and Reduce

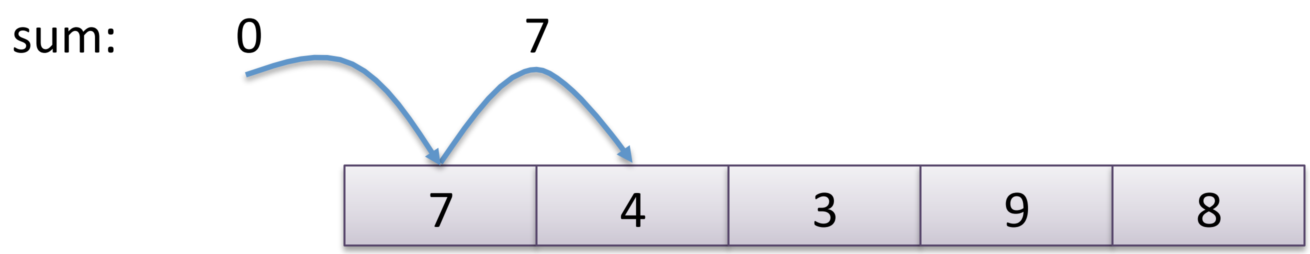
We have seen many sequential algorithms can be programmed succinctly using fold or reduce. Eg: sum all elements:

sum: 0

7	4	3	9	8

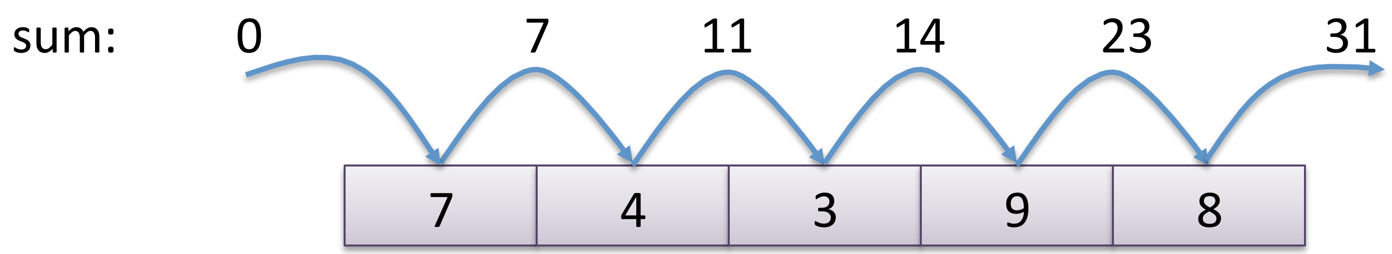
Fold and Reduce

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



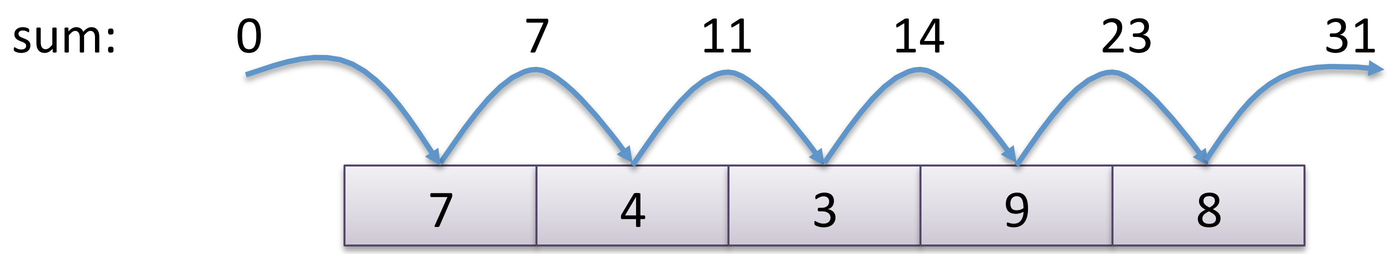
Fold and Reduce

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



Fold and Reduce

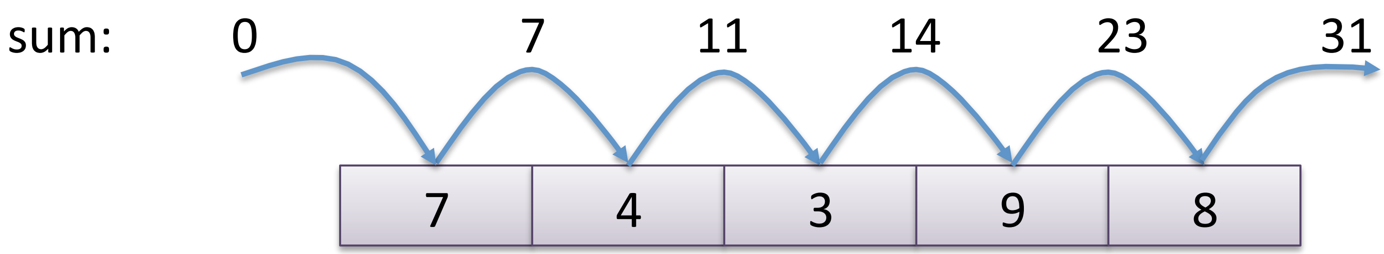
We have seen many sequential algorithms can be programmed succinctly using fold or reduce. Eg: sum all elements:




```
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) + v6)

add on processor 1

add on processor 2

Side Note: Associativity vs Commutativity

Associativity admits parallelism

(((((init + v1) + v2) + v3) + v4) + v5) == ((init + v1) + v2) + ((v3 + v4) + v6)

add on processor 1

add on processor 2

Commutativity allows us to reorder the elements:

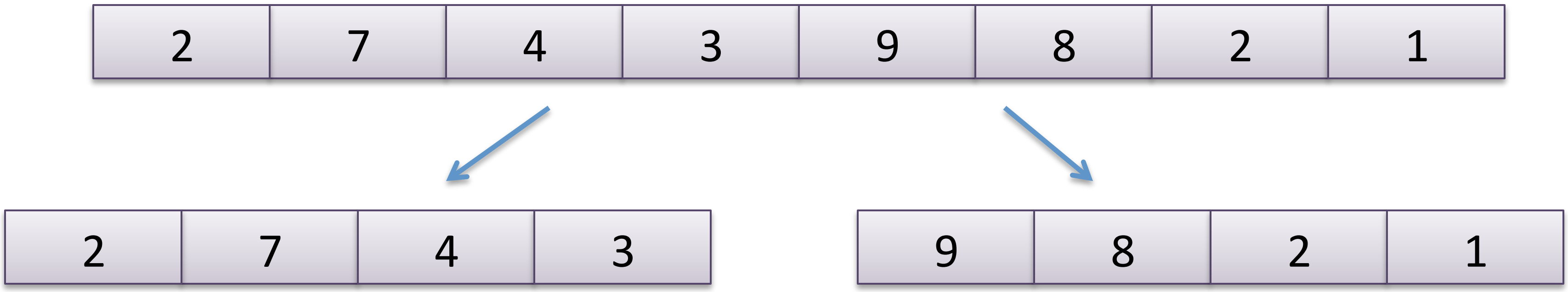
v1 + v2 == v2 + v1

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

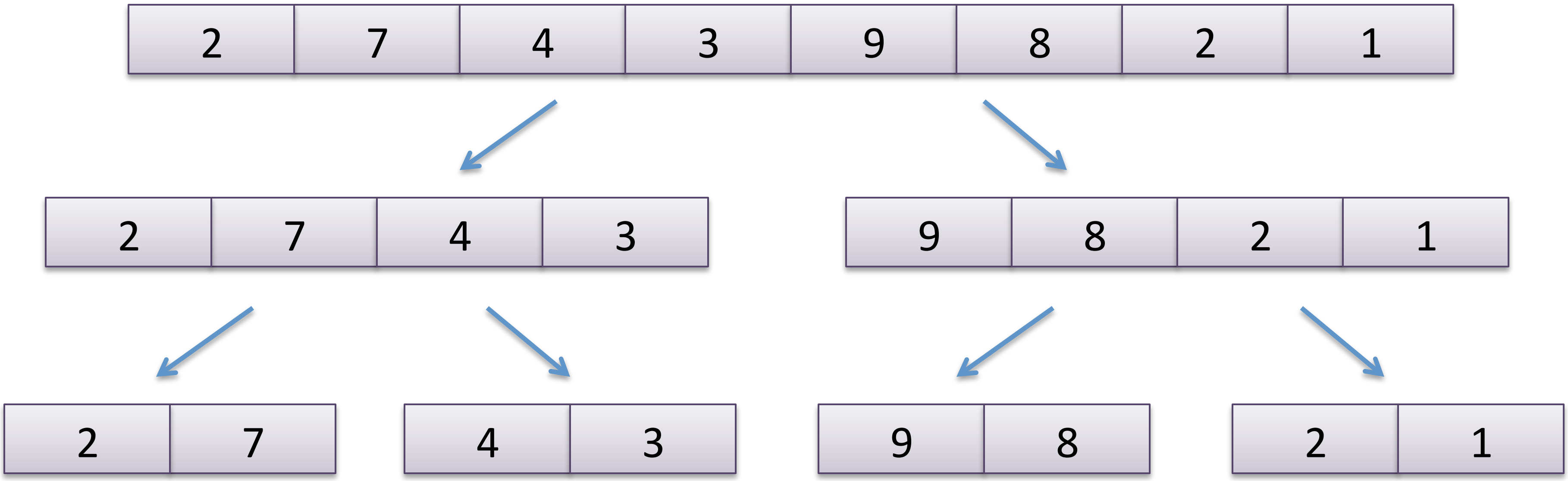
Parallel Sum

2	7	4	3	9	8	2	1

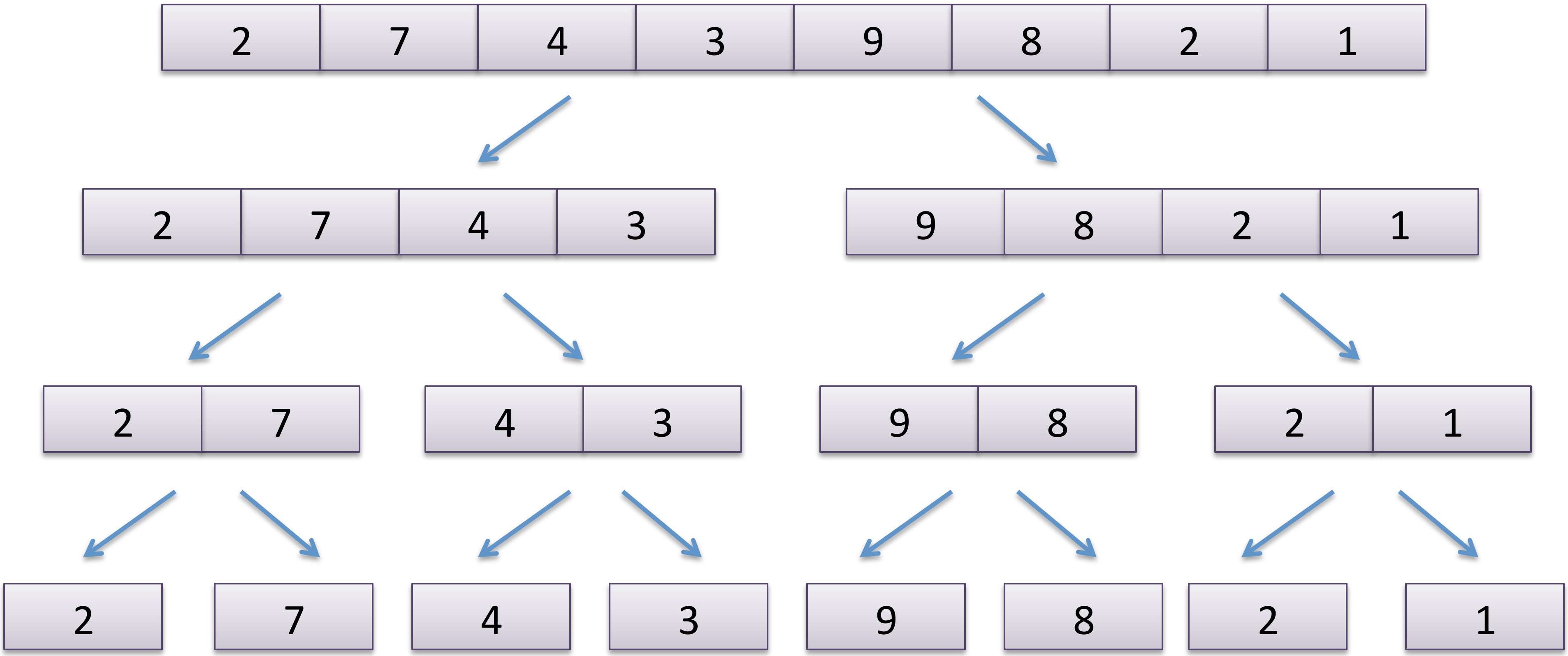
Parallel Sum



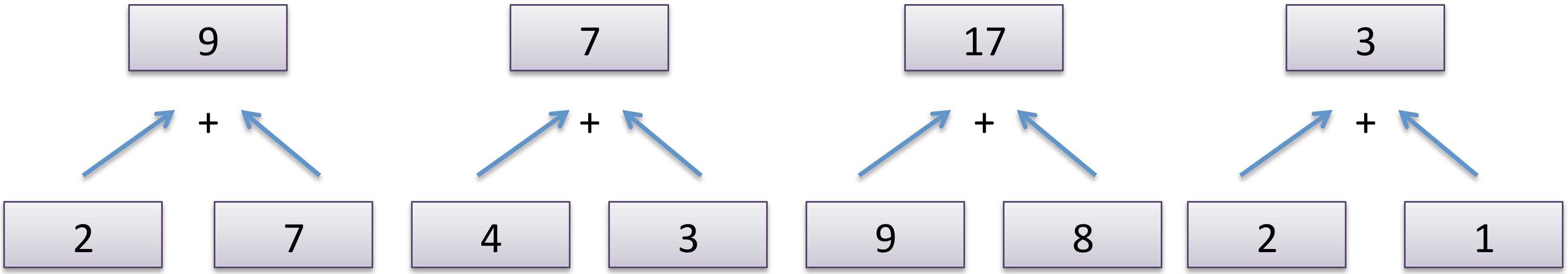
Parallel Sum



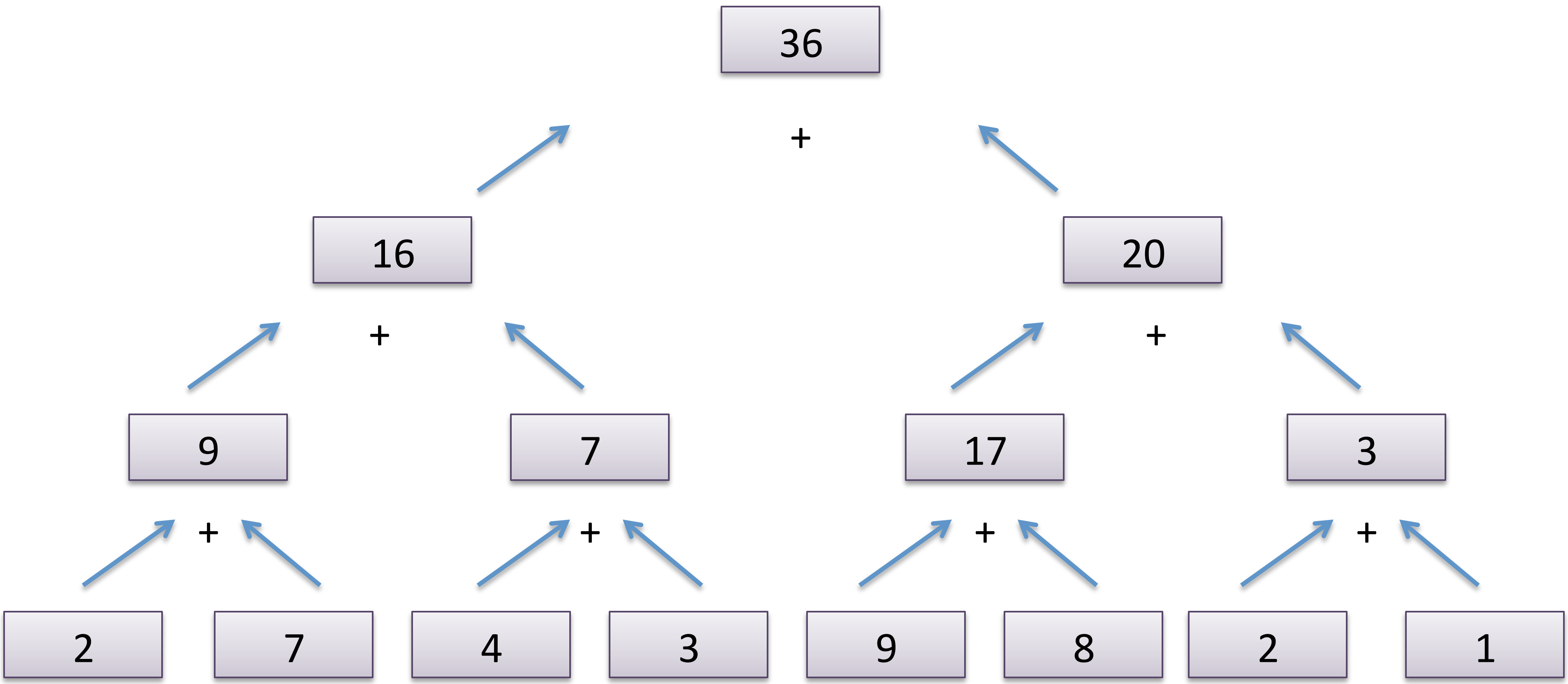
Parallel Sum



Parallel Sum



Parallel Sum



Splitting Sequences

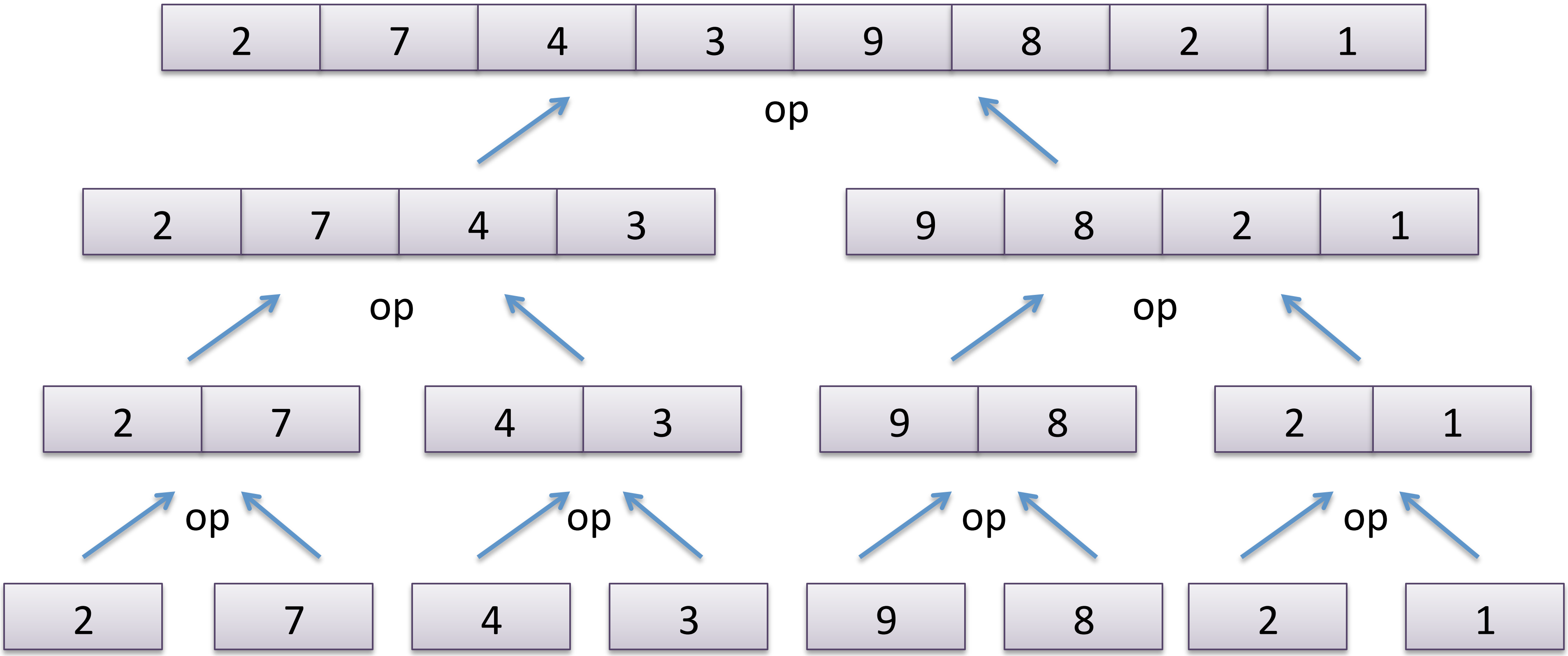
```
type 'a treeview =
  Empty
| One of 'a
| Pair of 'a seq * 'a seq

let show_tree (s:'a seq) : 'a treeview =
  match length s with
    0 -> Empty
  | 1 -> One (nth s 0)
  | n -> Pair (split s (n/2))
```

Parallel Sum

```
let rec psum (s : int seq) : int =
  match treeview s with
    Empty -> 0
   | One v -> v
   | Pair (s1, s2) ->
      let (n1, n2) = both psum s1
                          psum s2 in
      n1 + n2
```

Parallel Reduce



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

op base X == X   op X 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 treeview s with
    Empty -> base
   | One v -> f base v
   | Pair (s1, s2) ->
       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 treeview s with
    Empty -> base
  | One v -> f base v
  | Pair (s1, s2) ->
      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 treeview s with
    Empty -> base
  | One v -> inject v
  | Pair (s1, s2) ->
      let (r1, r2) = both mapreduce s1
                          mapreduce s2 in
      combine r1 r2
```

A little more general

```
let rec mapreduce
  (in:'a -> 'b)(comb:'b -> 'b -> 'b)(b:'b)(s:'a seq) =
  let mr = mapreduce in comb b in
  match treeview s with
    Empty -> b
  | One v -> in v
  | Pair (s1, s2) ->
      let (r1, r2) = both mr s1
                          mr s2 in
      comb r1 r2
```

```
let count s = mapreduce (fun x -> 1) (+) 0 s
```

A little more general

```
let rec mapreduce
  (in:'a -> 'b)(comb:'b -> 'b -> 'b)(b:'b)(s:'a seq) =
  let mr = mapreduce in comb b in
  match treeview s with
    Empty -> b
  | One v -> in v
  | Pair (s1, s2) ->
      let (r1, r2) = both mr s1
                          mr s2 in
      comb r1 r2
```

```
let count s = mapreduce (fun x -> 1) (+) 0 s
```

```
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
```

Parallel Reduce with Sequential Cut-off

When data is small, the overhead of parallelization isn't worth it.
You should revert to the sequential version.

```
type 'a treeview =
  Small of 'a seq | Big of 'a treeview * 'a treeview

let show_tree (s:'a seq) : 'a treeview =
  if length s < sequential_cutoff then
    Small s
  else
    Big (split s (n/2))
```

```
let rec reduce f b s =
  match treeview s with
    Small s -> sequential_reduce f b s
  | Big (s1, s2) ->
      let (n1, n2) = both (reduce f b) s1
                          (reduce f b) s2
      in
      f n1 n2
```

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(eft) or R(ight) paren *)

let balanced (ps : paren seq) : bool = ...
```

First, a sequential approach

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

(	(	)	)	)	(	)	(

0

First, a sequential approach

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

(	(	)	)	)	(	)	(

0 1

First, a sequential approach

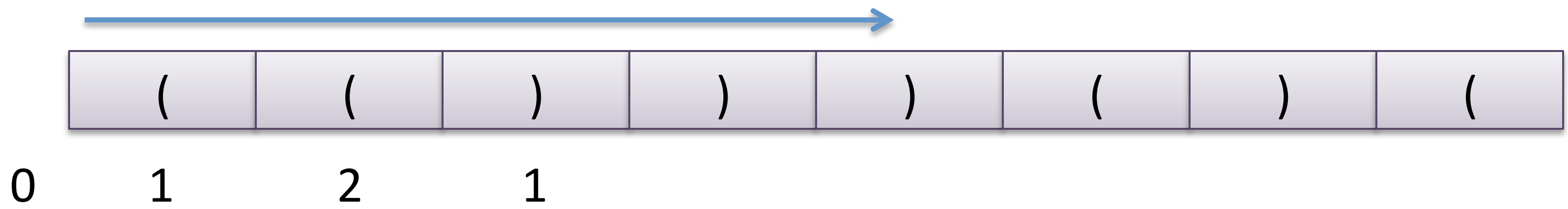
fold from left to right, keep track of
# of unmatched left parens

(	(	)	)	)	(	)	(

0 1 2

First, a sequential approach

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



First, a sequential approach

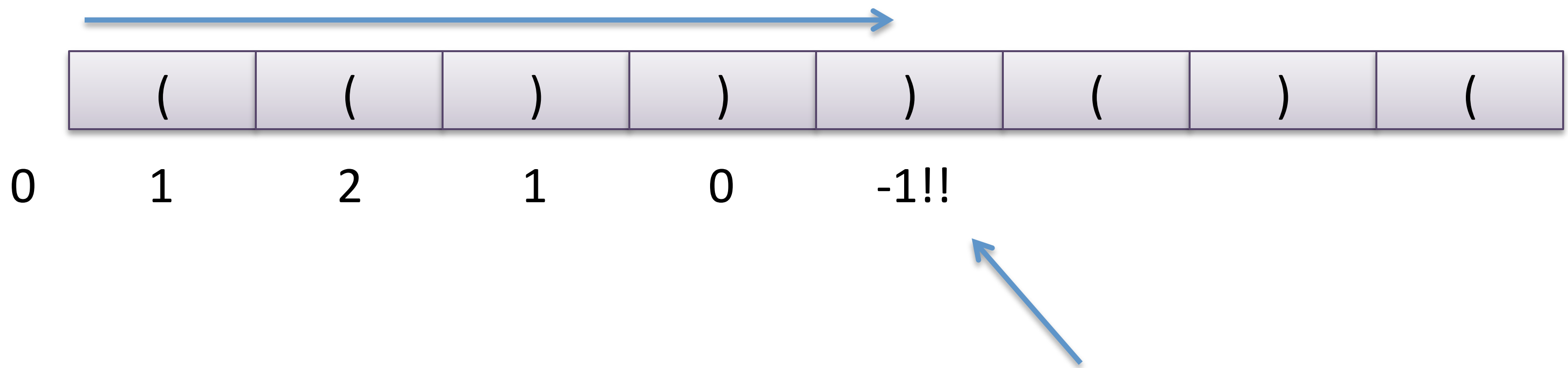
fold from left to right, keep track of
# of unmatched left parens

(	(	)	)	)	(	)	(

0 1 2 1 0

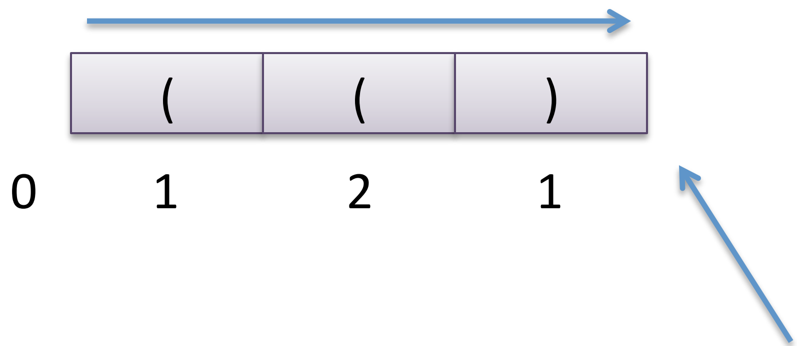
First, a sequential approach

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



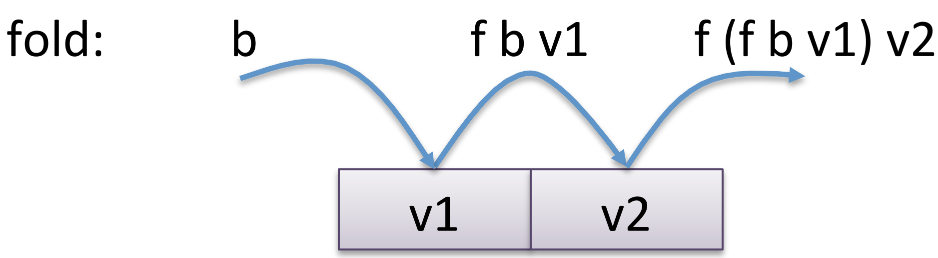
too many right parens
indicates no match

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
```

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 > 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:
  -))) … 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 > k then))) … j …)))))) … x – k …))) (((… y … (((

- if x < k then))) … j …))) (((… k – x … ((((((… y … (((

Parallel Matcher

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

    ))) ... j ... ))) ((( ... k ... (((

 *)

let rec matcher s =
    match show_tree s with
      Empty -> (0, 0)
    | One L -> (0, 1)
    | One R -> (1, 0)
    | Pair (left, right) ->
       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 Matcher

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

    ))) ... j ... ))) ((( ... k ... (((

 *)

let rec matcher s =
    match show_tree s with
      Empty -> (0, 0)
    | One L -> (0, 1)
    | One R -> (1, 0)
    | Pair (left, right) ->
       let (j, k), (x, y) = both matcher left
                                 matcher right    in
       if x > k then
         (j + (x - k), y)
       else
         (j, (k - x) + y)
```

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

Parallel Balance

```
(*   *)
let matcher s = ...

(* true if s is a sequence of balanced parens *)
let balanced s =
    match matcher 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)

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
```

For correctness,
check the associativity
of combine

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

PARALLEL SCAN AND PREFIX SUM

The prefix-sum problem

Sum of Sequence:

`input`	6	4	16	10	16	14	2	8
`output`	76							

Prefix-Sum **of Sequence:**

`input`	6	4	16	10	16	14	2	8
`output`	6	10	26	36	52	66	68	76

The prefix-sum problem

val prefix_sum : int seq -> int seq

`input`	6	4	16	10	16	14	2	8
`output`	6	10	26	36	52	66	68	76

The simple sequential algorithm: accumulate the sum from left to right

- Sequential algorithm: Work: $O(n)$, Span: $O(n)$
- Goal: a parallel algorithm with Work: $O(n)$, Span: O(log n)

Parallel prefix-sum

The trick: *Use two passes*

- Each pass has $O(n)$ work and $O(\texttt{log}\ n)$ span
- So in total there is $O(n)$ work and $O(\texttt{log}\ n)$ span

First pass *builds a tree of sums bottom-up*

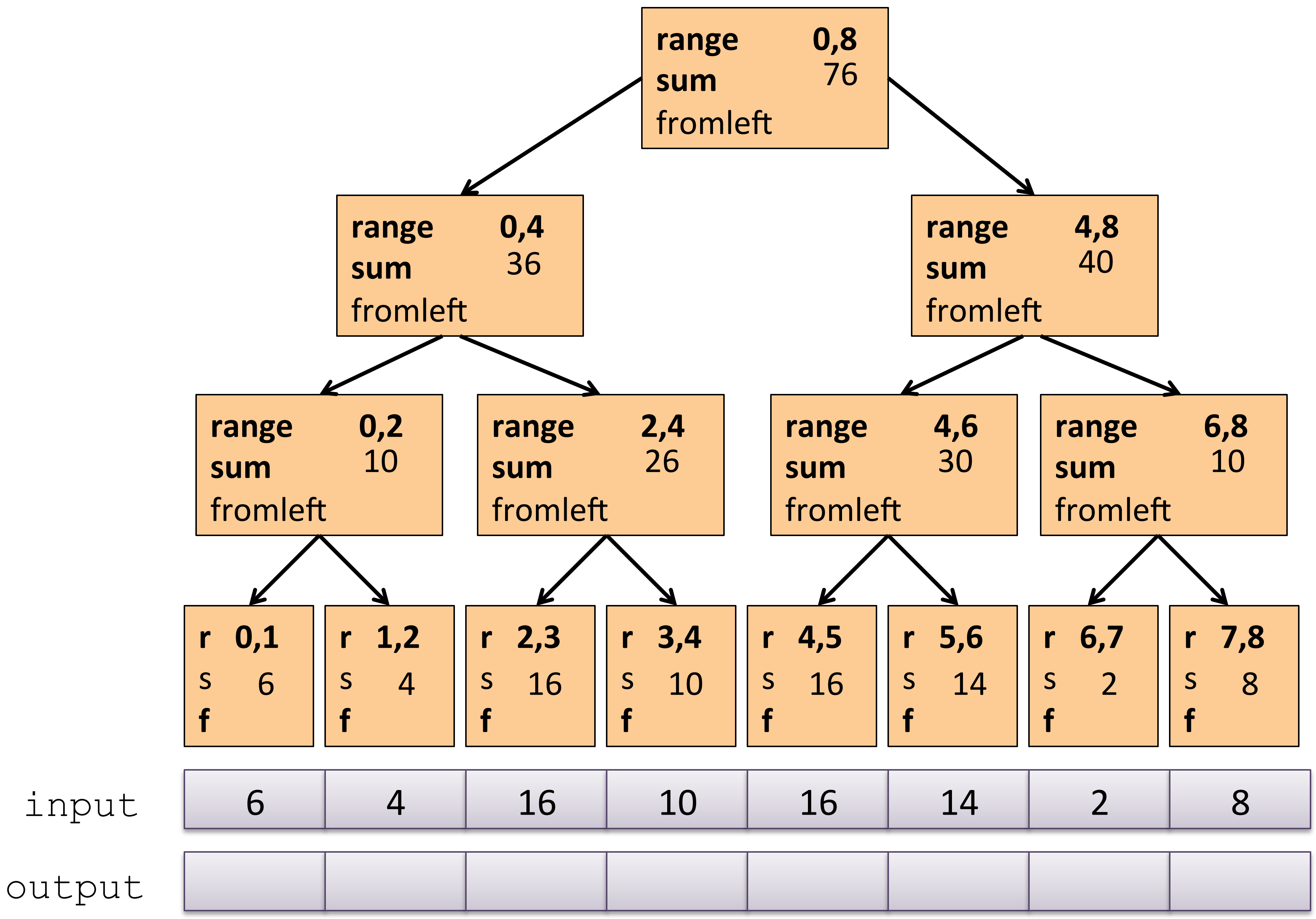
- the "up" pass

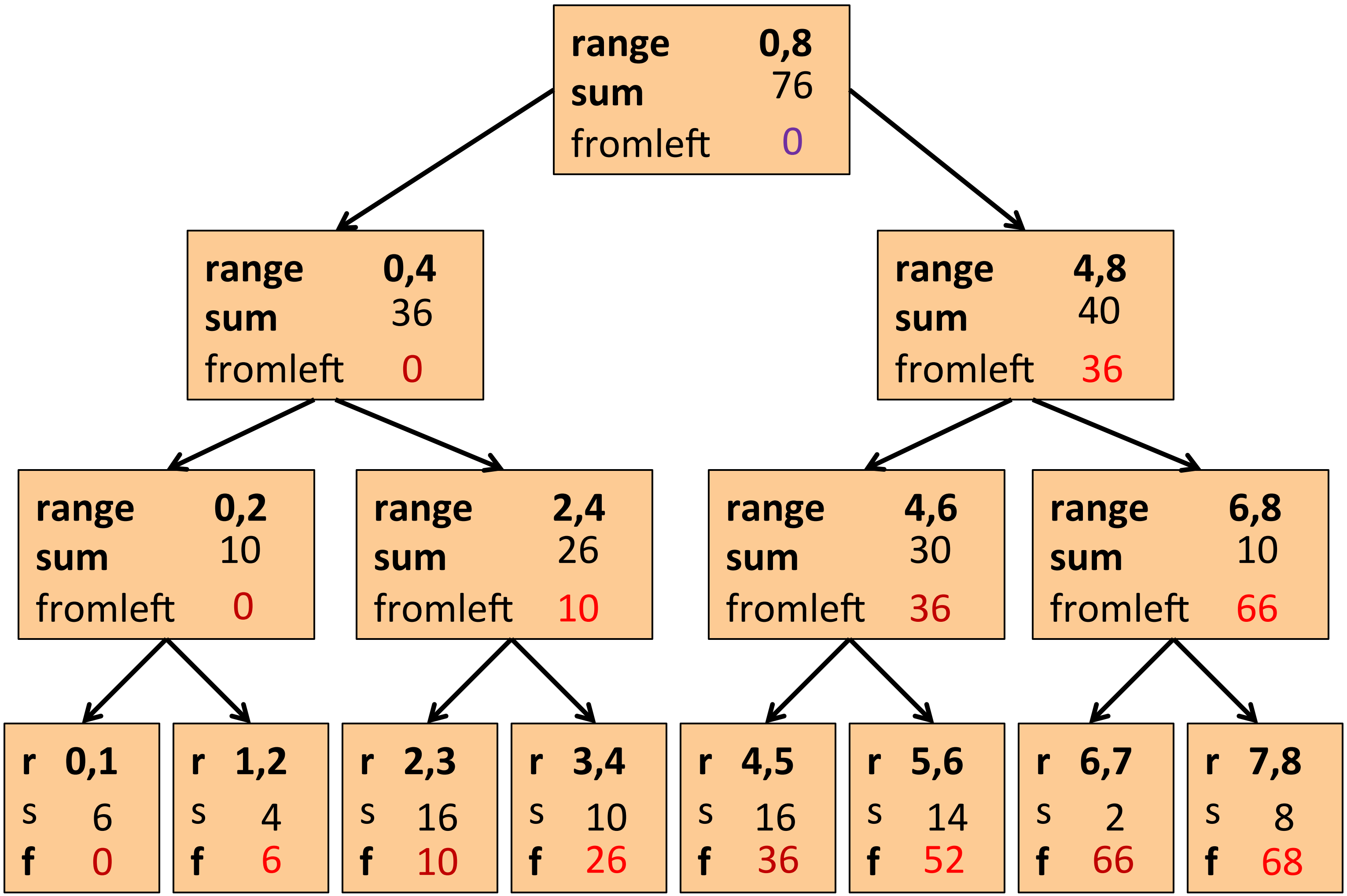
Second pass *traverses the tree top-down to compute prefixes*

- the "down" pass computes the "from-left-of-me" sum

Historical note:

- Original algorithm due to R. Ladner and M. Fischer, 1977


range 0,8
sum 76
fromleft
range 0,4
sum 36
fromleft
range 4,8
sum 40
fromleft
range 0,2
sum 10
fromleft
range 2,4
sum 26
fromleft
range 4,6
sum 30
fromleft
range 6,8
sum 10
fromleft
r 0,1
s 6
f
r 1,2
s 4
f
r 2,3
s 16
f
r 3,4
s 10
f
r 4,5
s 16
f
r 5,6
s 14
f
r 6,7
s 2
f
r 7,8
s 8
f
input
6
4
16
10
16
14
2
8
output



input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76

The algorithm, pass 1

1. Up: Build a binary tree where
   - Root has sum of the range [`x,y`)
   - If a node has sum of [`lo,hi`) and `hi>lo`,
     - Left child has sum of [`lo,middle`)
     - Right child has sum of [`middle,hi`)
     - A leaf has sum of [`i,i+1`), i.e., `nth input i`

This is an easy parallel divide-and-conquer algorithm: "combine" results by actually building a binary tree with all the range-sums

- Tree built bottom-up in parallel

Analysis: $O(n)$ work, $O(\log n)$ span

The algorithm, pass 2

2. Down: Pass down a value **`fromLeft`**
   - Root given a **`fromLeft`** of **`0`**
   - Node takes its **`fromLeft`** value and
     - Passes its left child the same **`fromLeft`**
     - Passes its right child its **`fromLeft`** plus its left child's **`sum`**
       - as stored in part 1
   - At the leaf for sequence position **`i`**,
     - **`nth output i == fromLeft + nth input i`**

This is an easy parallel divide-and-conquer algorithm: traverse the tree built in step 1 and produce no result

- Leaves create **`output`**
- Invariant: **`fromLeft`** is sum of elements left of the node's range

Analysis: $O(n)$ work, $O(\mathbf{log}\ n)$ span

Sequential cut-off

For performance, we need a sequential cut-off:

- Up:
  - just a sum, have leaf node hold the sum of a range
- Down:
  - do a sequential scan

Parallel prefix, generalized

Just as map and reduce are the simplest examples of a common pattern, prefix-sum illustrates a pattern that arises in many, many problems

- Minimum, maximum of all elements *to the left of* `i`
- Is there an element *to the left of* `i` satisfying some property?
- Count of elements *to the left of* `i` satisfying some property
  - This last one is perfect for an efficient parallel filter …
  - Perfect for building on top of the "parallel prefix trick"

Parallel Scan

scan (o) <x1, ..., xn>

==

<x1, x1 o x2, ..., x1 o ... o xn>

like a fold, except return
the folded prefix at each step

pre_scan (o) base <x1, ..., xn>

==

<base, base o x1, ..., base o x1 o ... o xn-1>

sequence with o applied to all items
to the left of index in input

More Algorithms

- To add multiprecision numbers.
- To evaluate polynomials
- To solve recurrences.
- To implement radix sort
- To delete marked elements from an array
- To dynamically allocate processors
- To perform lexical analysis. For example, to parse a program into tokens.
- To search for regular expressions. For example, to implement the UNIX grep program.
- To implement some tree operations. For example, to find the depth of every vertex in a tree
- To label components in two dimensional images.

See Guy Blelloch "Prefix Sums and Their Applications"

Summary

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

Scans are trickier and use a 2-pass algorithm that builds a tree.

The map-reduce-fold paradigm, inspired by functional programming, is a big winner when it comes to big data processing.

Hadoop is an industry standard but higher-level data processing languages have been built on top.