Parallel Sequences: Fold, Reduce and Scan

COS 326

David Walker

Princeton University

Credits:
Dan Grossman, UW http://homes.cs.washington.edu/~djg/teachingMaterials/spac
Blelloch, Harper, Licata (CMU, Wesleyan)

A Parallel Sequence API

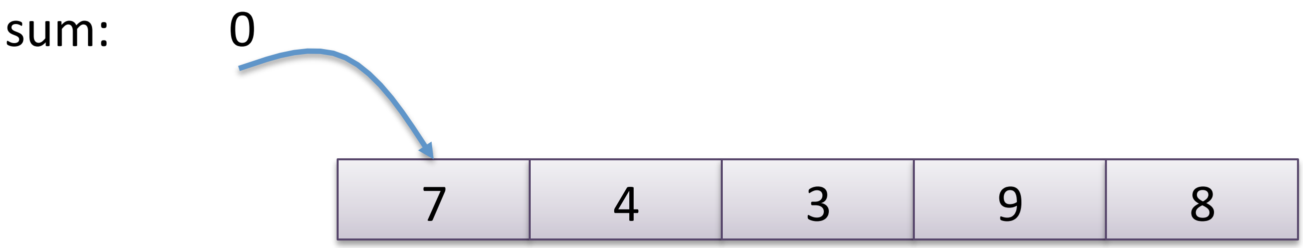
```
type 'a seq
```

	Work	Span
`tabulate : 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 -> '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

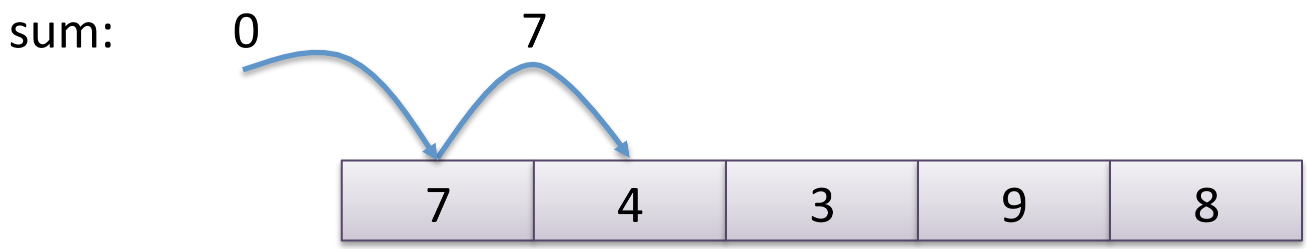
We have seen many sequential algorithms can be programmed succintly using fold or reduce. Eg: sum all elements:



7	4	3	9	8

Fold and Reduce

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

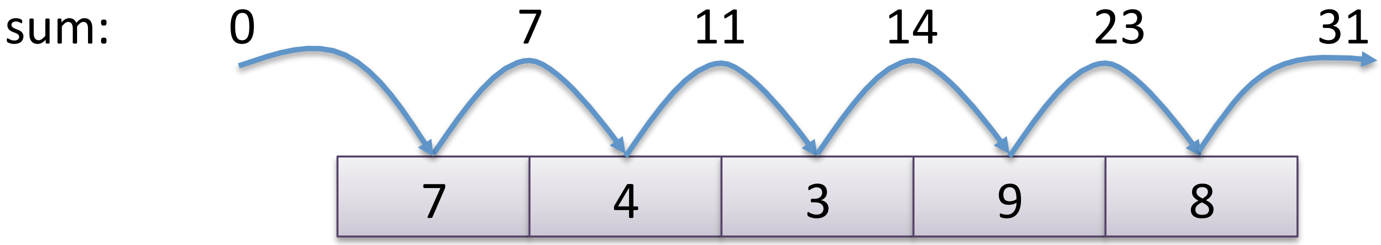


sum: 0 7

7	4	3	9	8

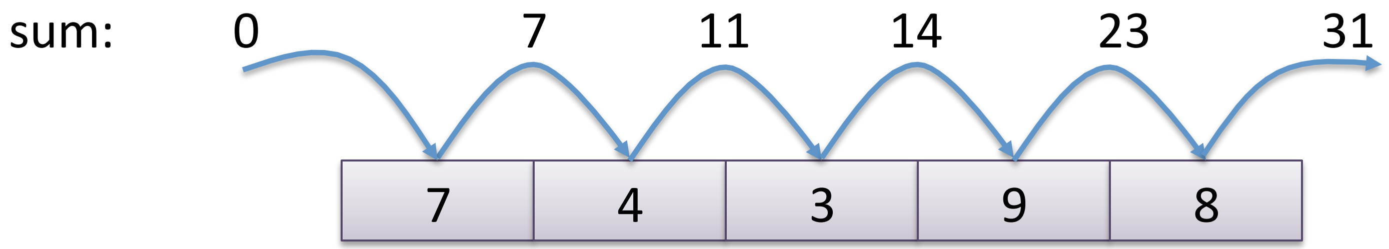
Fold and Reduce

We have seen many sequential algorithms can be programmed succintly 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:



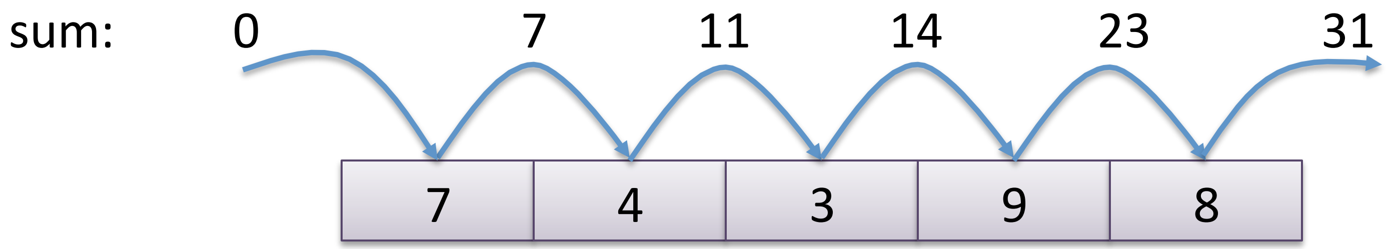
sum: 0 7 11 14 23 31

7 4 3 9 8

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

Fold and Reduce

We have seen many sequential algorithms can be programmed succintly 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

The key is *associativity*:

$$(((((init + v1) + v2) + v3) + v4) + v5) \;==\; \underbrace{((init + v1) + v2)}_{\text{add on processor 1}} + \underbrace{((v3 + v4) + v6)}_{\text{add on processor 2}}$$

Commutativity allows us to reorder the elements:

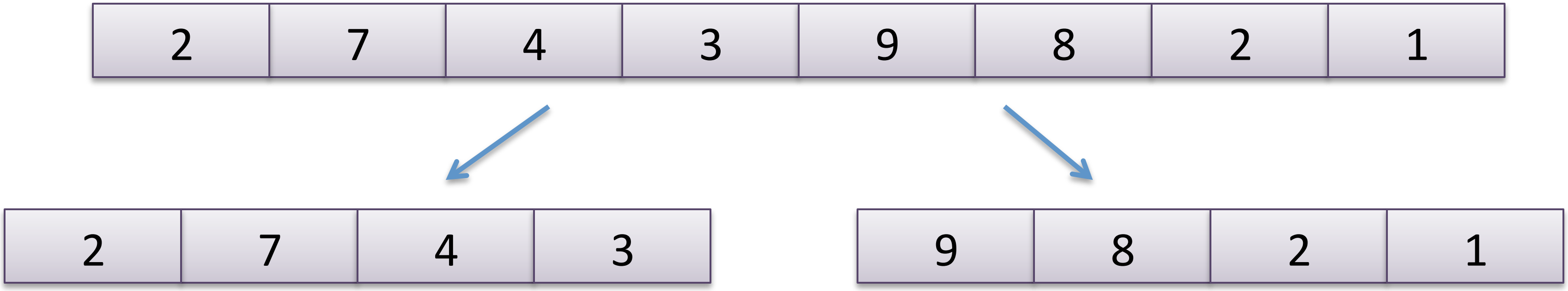
$$v1 + v2 == v2 + v1$$

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

Parallel Sum

2	7	4	3	9	8	2	1

Parallel Sum

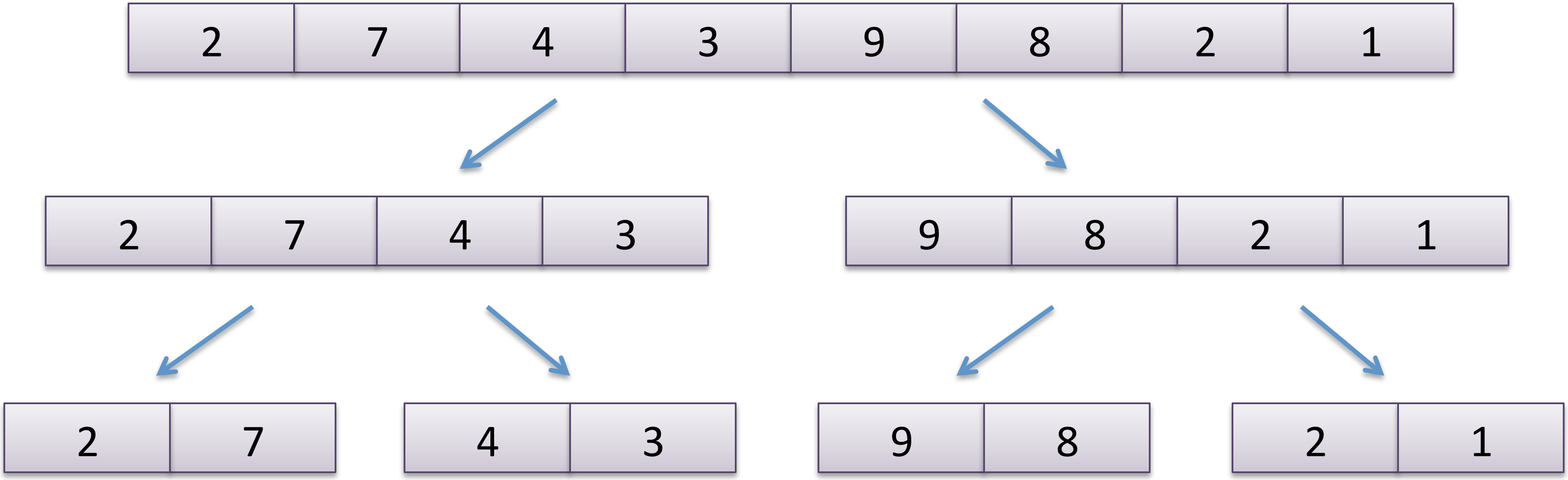


2	7	4	3	9	8	2	1

2	7	4	3

9	8	2	1

Parallel Sum



2	7	4	3	9	8	2	1

2	7	4	3

9	8	2	1

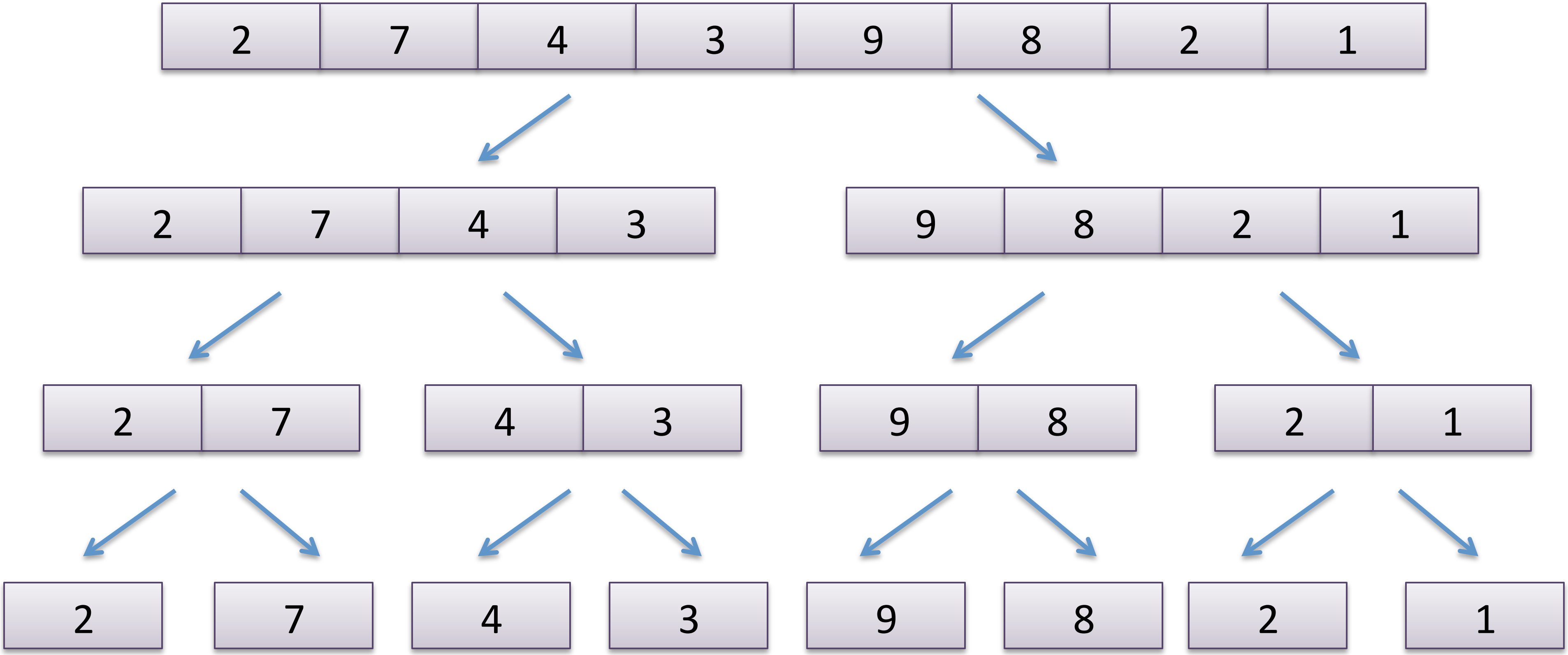
2	7

4	3

9	8

2	1

Parallel Sum



2	7	4	3	9	8	2	1

2	7	4	3

9	8	2	1

2	7

4	3

9	8

2	1

2

7

4

3

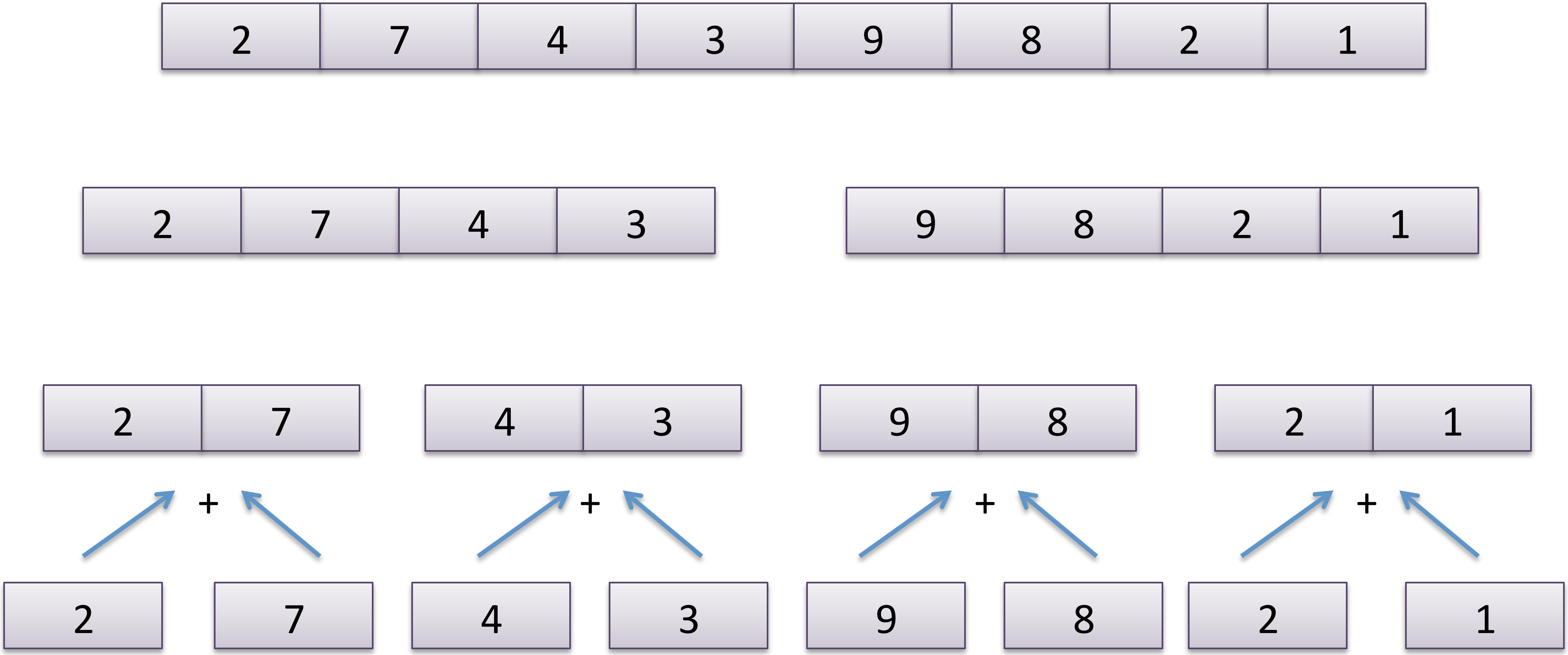
9

8

2

1

Parallel Sum



2 7 4 3 9 8 2 1

2 7 4 3

9 8 2 1

2 7

4 3

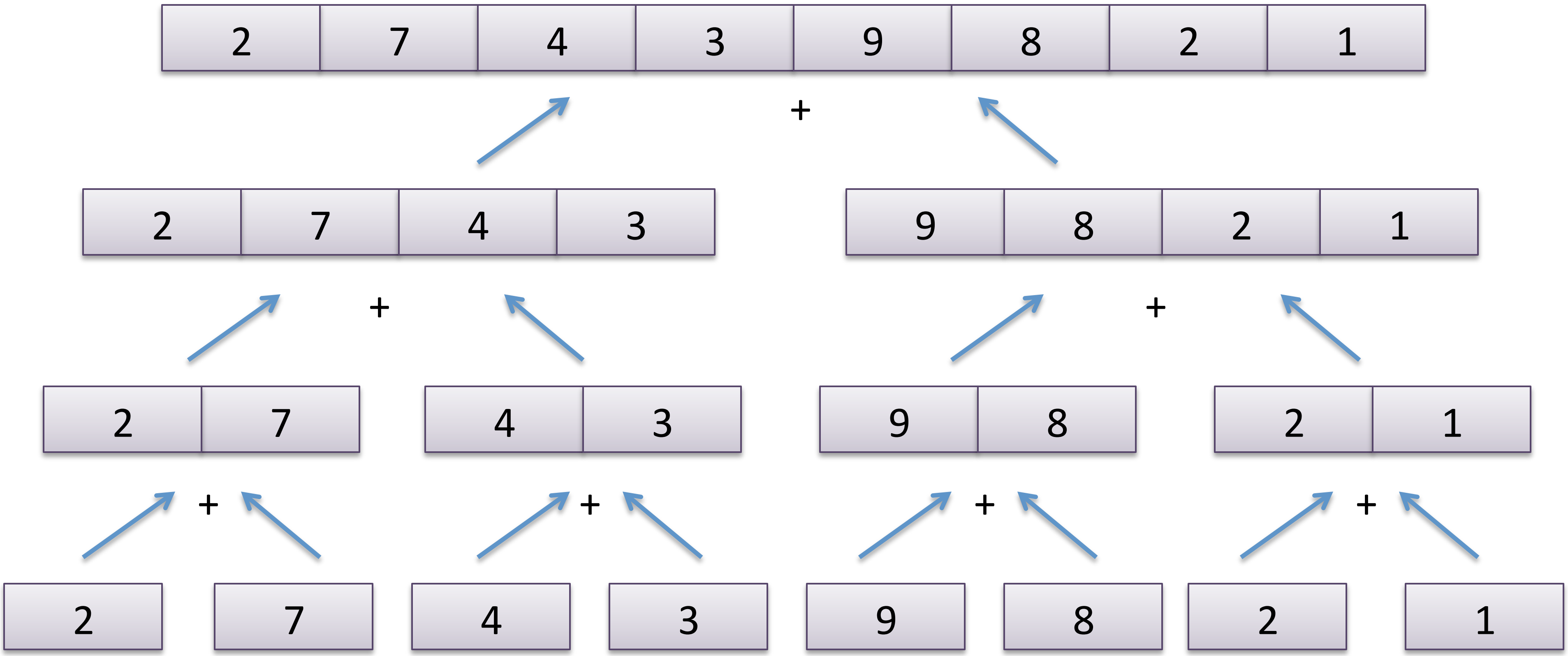
9 8

2 1

+ + + +

2 7 4 3 9 8 2 1

Parallel Sum



2 7 4 3 9 8 2 1

+

2 7 4 3 9 8 2 1

+ +

2 7 4 3 9 8 2 1

+ + + +

2 7 4 3 9 8 2 1

Splitting Sequences

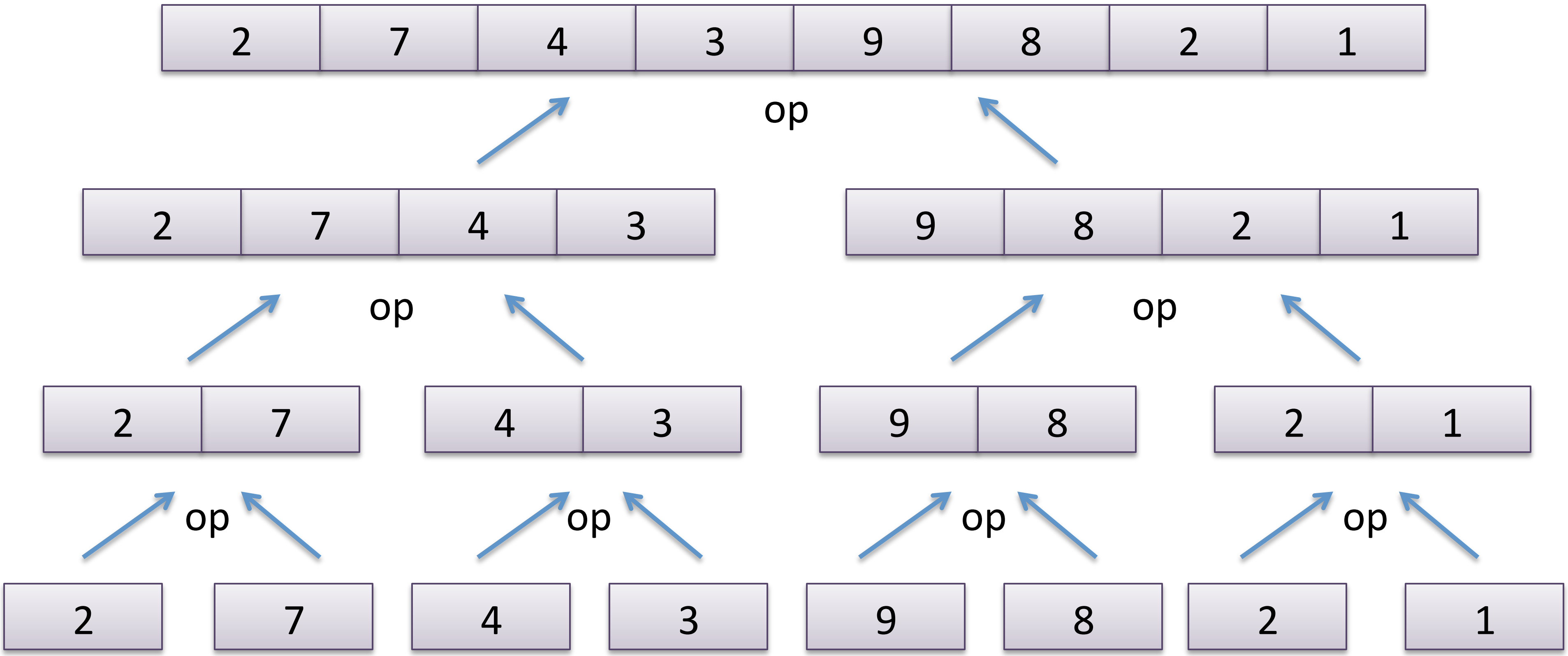
```
type 'a treeview =
  Empty
| One of 'a
| Pair of 'a seq * 'a seq

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 s1
                          reduce 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 s1
                          reduce s2 in
      f n1 n2
```

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

A little more general

```
let rec davefold (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 davefold s1
                          davefold s2 in
      combine r1 r2
```

A little more general

```
let rec davefold (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 davefold s1
                          davefold s2 in
      combine r1 r2
```

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

A little more general

```
let rec davefold (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 davefold s1
                          davefold s2 in
      combine r1 r2
```

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

```
let average s =
  let (count, total) =
    davefold (fun x -> (0,1))
             (fun (c1,t1) (c2,t2) -> (c1+c2, t1 + t2))
             0 s in
  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 seq * 'a seq

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 base s =
  match treeview s with
    Small s -> sequential_reduce f base s
  | Big (s1, s2) ->
      let (n1, n2) = both reduce s1
                          reduce 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: ()))

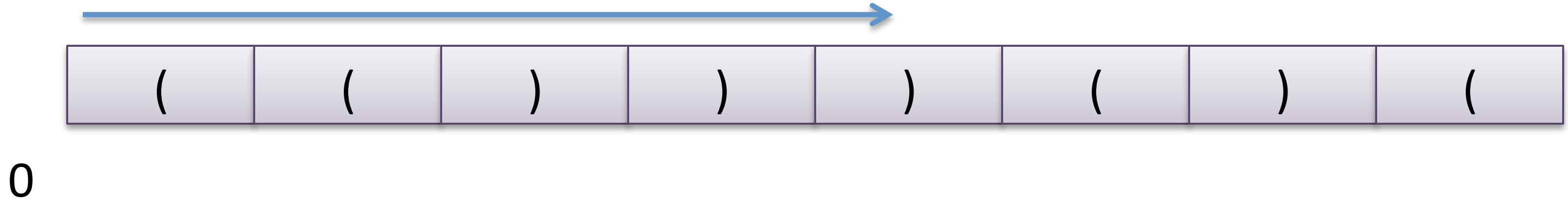
We will try formulating a divide-and-conquer parallel algorithm to solve this problem efficiently:

```
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 right parens

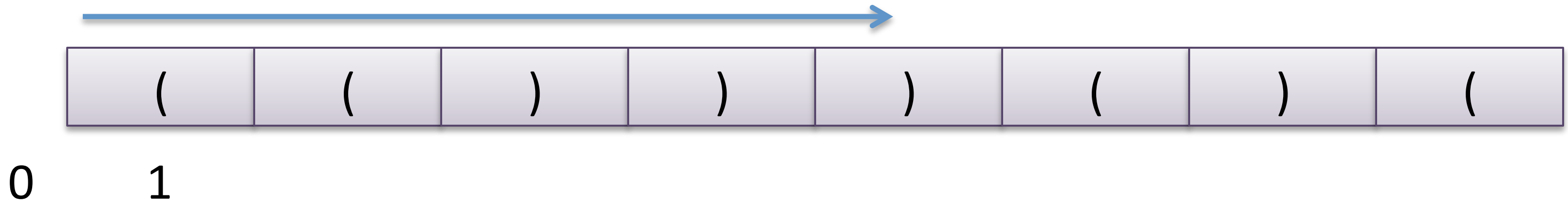


(	(	)	)	)	(	)	(

0

First, a sequential approach

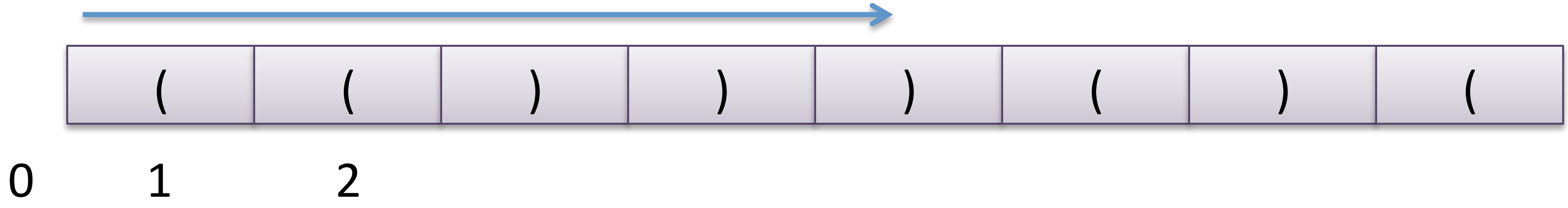
fold from left to right, keep track of
# of unmatched right parens



(	(	)	)	)	(	)	(

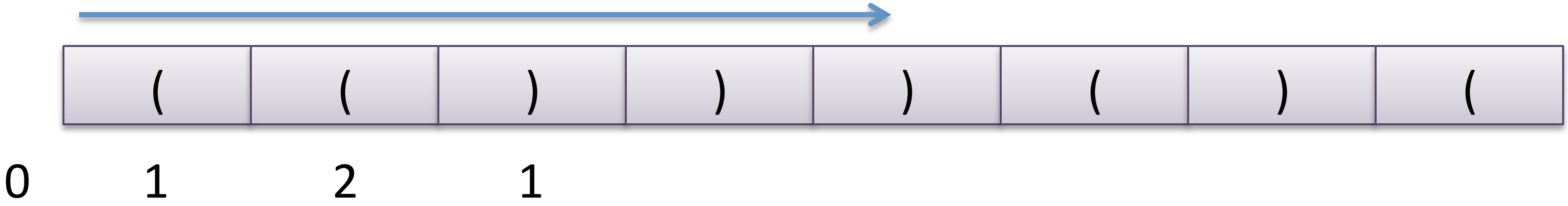
First, a sequential approach

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



First, a sequential approach

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

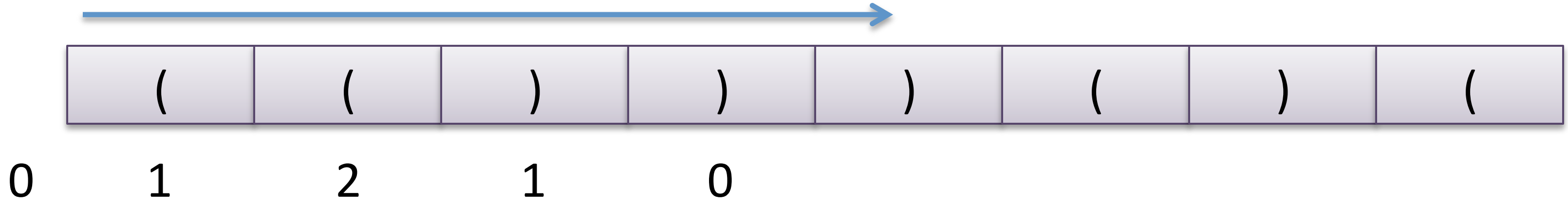


(( ))) ( ) (

0 1 2 1

First, a sequential approach

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

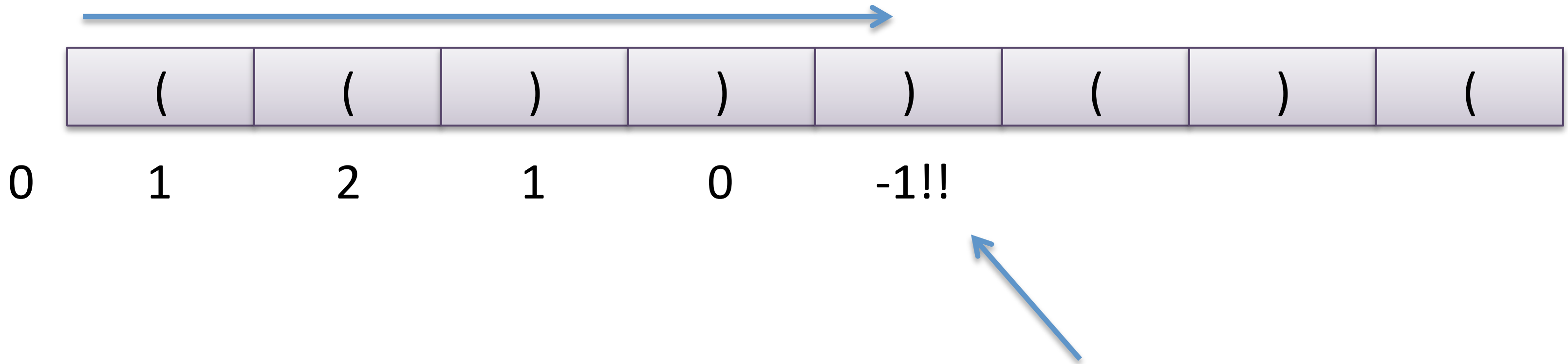


(( ))) ( ) (

0 1 2 1 0

First, a sequential approach

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

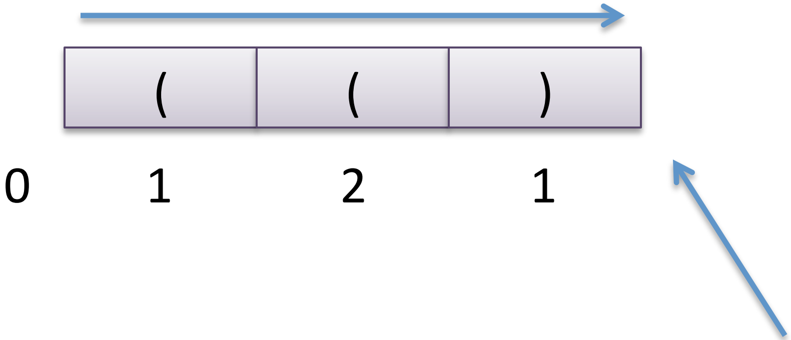


(	(	)	)	)	(	)	(

0 1 2 1 0 -1!!

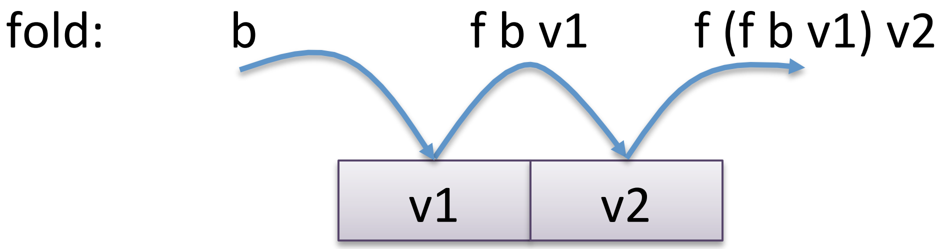
too many right parens
indicates no match

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



fold: b f b v1 f (f b v1) v2

v1 v2

```
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, l) = both matcher left
                                 matcher right    in
       if x > k then
         (j + (x - k), y)
       else
         (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, l) = 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 davefold (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 (i,j) (x,j) =
      if x > k then (j + (x - k), y)
      else          (j, (k - x) + y)

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

Using a Fold

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

let balanced s =
    match davefold 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

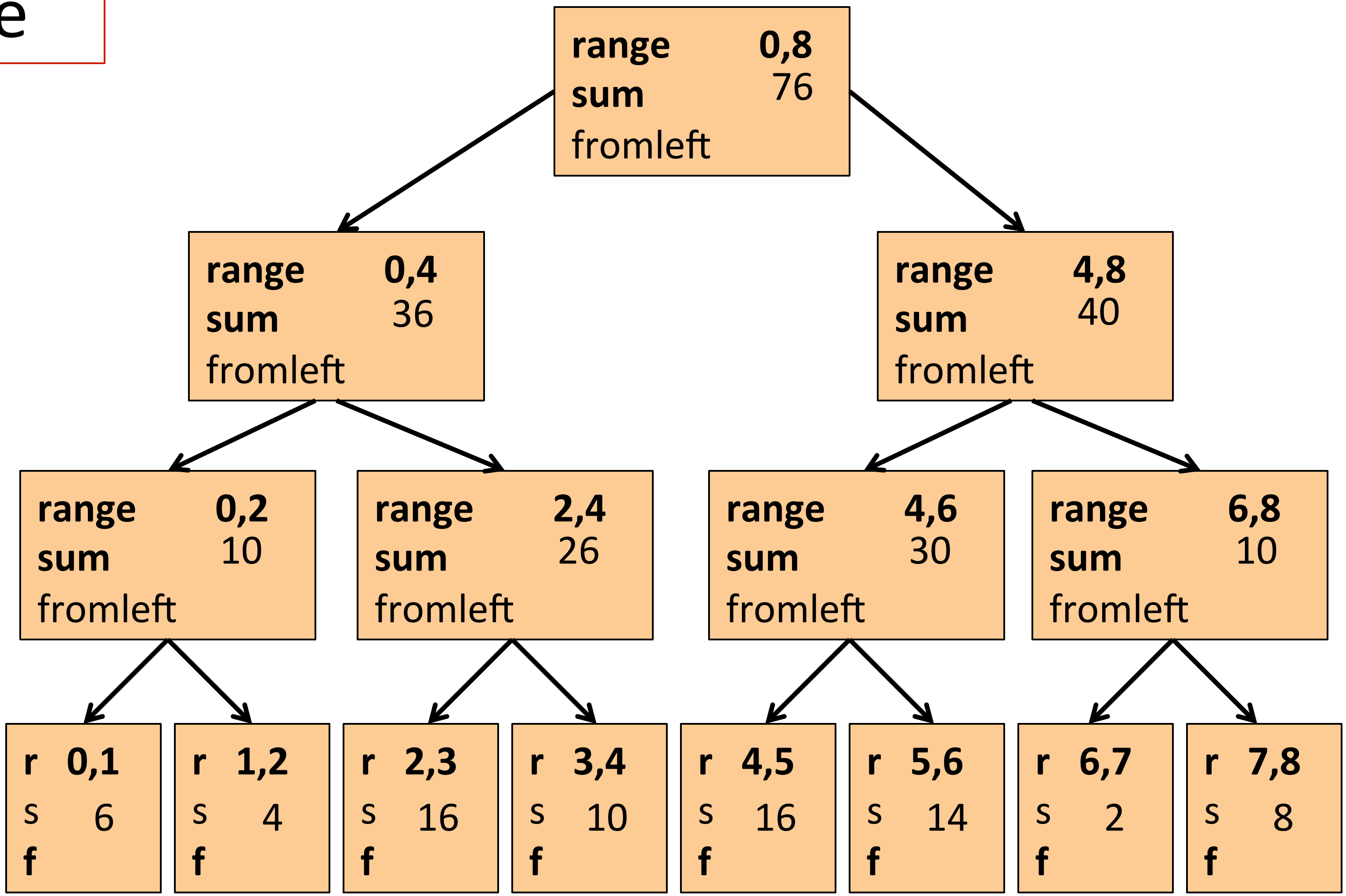
Second pass *traverses the tree top-down to compute prefixes*

- the “down” pass

Historical note:

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

Example



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

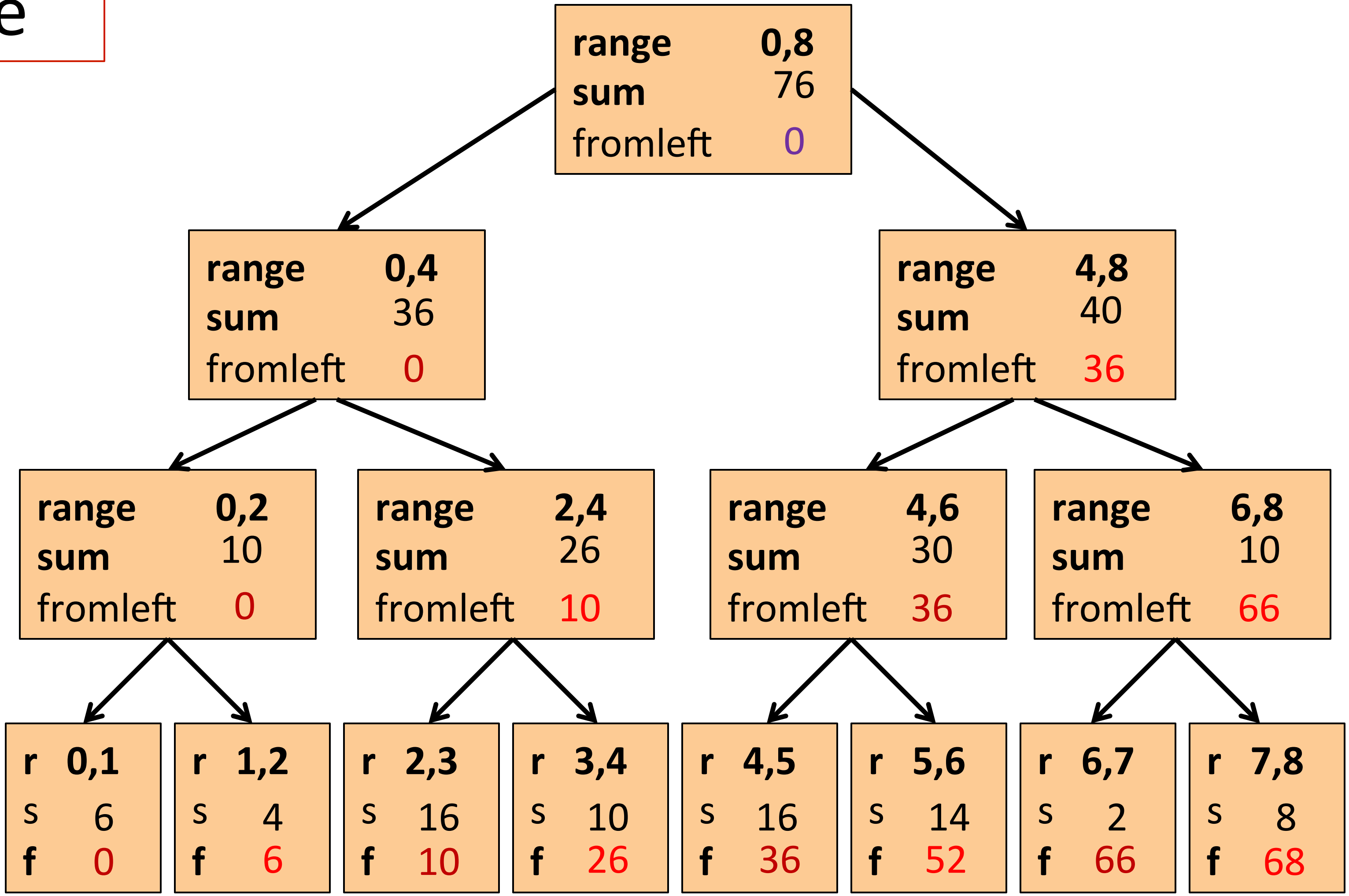
r 5,6
s 14
f

r 6,7
s 2
f

r 7,8
s 8
f

input	6	4	16	10	16	14	2	8
output								

Example



range 0,8
sum 76
fromleft 0

range 0,4
sum 36
fromleft 0

range 4,8
sum 40
fromleft 36

range 0,2
sum 10
fromleft 0

range 2,4
sum 26
fromleft 10

range 4,6
sum 30
fromleft 36

range 6,8
sum 10
fromleft 66

r 0,1
s 6
f 0

r 1,2
s 4
f 6

r 2,3
s 16
f 10

r 3,4
s 10
f 26

r 4,5
s 16
f 36

r 5,6
s 14
f 52

r 6,7
s 2
f 66

r 7,8
s 8
f 68

`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*(**`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

Parallel Filter

Given a sequence **`input`**, produce a sequence **`output`** containing only elements v such that **`(f v)`** is **`true`**

Example: let f x = x > 10

```
    filter f <17, 4, 6, 8, 11, 5, 13, 19, 0, 24>
 == <17, 11, 13, 19, 24>
```

Parallelizable?

- Finding elements for the output is easy
- *But getting them in the right place seems hard*

Parallel prefix to the rescue

Use parallel map to compute a bit-vector for true elements:

```
input  <17, 4, 6, 8, 11, 5, 13, 19, 0, 24>
bits   <1,  0, 0, 0,  1, 0,  1,  1, 0,  1>
```

Use parallel-prefix sum on the bit-vector:

```
bitsum <1,  1, 1, 1,  2, 2,  3,  4, 4,  5>
```

For each i, if bits[i] == 1 then write input[bitsum[i]] to output[i] to produce the final result:

```
output <17, 11, 13, 19, 24>
```

QUICKSORT

Quicksort review

Recall quicksort was sequential, in-place, expected time $O(n \log n)$

	Best / expected case *work*
1. Pick a pivot element	O(1)
2. Partition all the data into:	O(n)
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	2T(n/2)

How should we parallelize this?

Quicksort

		Best / expected case *work*
1.	Pick a pivot element	O(1)
2.	Partition all the data into:	O(n)
	A. The elements less than the pivot	
	B. The pivot	
	C. The elements greater than the pivot	
3.	Recursively sort A and C	2T(n/2)

Easy: Do the two recursive calls in parallel

- Work: unchanged. Total: $O(n \log n)$
- Span: now $\mathrm{T}(n) = O(n) + 1\mathrm{T}(n/2) = O(n)$

Doing better

As with mergesort, we get a $O(\texttt{log}\ n)$ speed-up with an *infinite* number of processors. That is a bit underwhelming

- Sort 10^9 elements 30 times faster

(Some) Google searches suggest quicksort cannot do better because the partition cannot be parallelized

- The Internet has been known to be wrong ☺
- But we need auxiliary storage (no longer in place)
- In practice, constant factors may make it not worth it

Already have everything we need to parallelize the partition…

Parallel partition (not in place)

Partition all the data into:

A. The elements less than the pivot
B. The pivot
C. The elements greater than the pivot

This is just two filters!

- We know a parallel filter is $O(n)$ work, $O(\texttt{log}\ n)$ span
- Parallel filter elements less than pivot into left side of **`aux`** array
- Parallel filter elements greater than pivot into right size of **`aux`** array
- Put pivot between them and recursively sort
- With a little more cleverness, can do both filters at once but no effect on asymptotic complexity

With $O(\texttt{log}\ n)$ span for partition, the total best-case and expected-case span for quicksort is

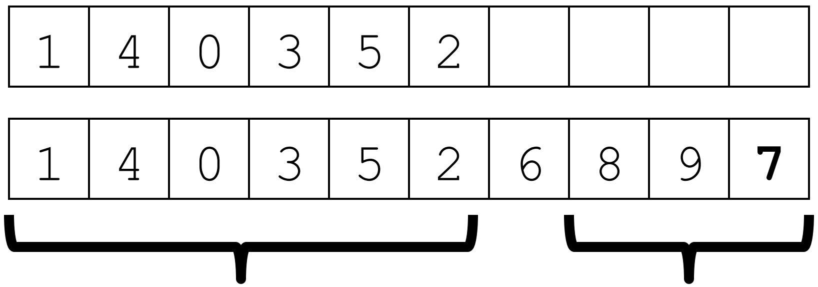
$$T(n) = O(\texttt{log}\ n) + 1T(n/2) = O(\texttt{log}^2\ n)$$

Example

Step 1: pick pivot as median of three

8	1	4	9	**0**	3	5	2	7	**6**

Steps 2a and 2c (combinable): filter less than, then filter greater than into a second array



1	4	0	3	5	2				

1	4	0	3	5	2	6	8	9	**7**

Step 3: Two recursive sorts in parallel

- Can copy back into original array (like in mergesort)

More Algorithms

- To add multi precision 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

- Parallel prefix sums and scans have many applications
  - A good algorithm to have in your toolkit!

- Key idea: An algorithm in 2 passes:
  - Pass 1: build a "reduce tree" from the bottom up
  - Pass 2: compute the prefix top-down, looking at the left-subchild to help you compute the prefix for the right subchild

PARALLEL COLLECTIONS IN THE "REAL WORLD"

Big Data

If Google wants to index all the web pages (or images or gmails or google docs or ...) in the world, they have a lot of work to do

- Same with Facebook for all the facebook pages/entries
- Same with Twitter
- Same with Amazon
- Same with ...

Many of these tasks come down to map, filter, fold, reduce, scan

Scala

Parallel Collections
with Scala

Jul 6' 2012 > Vikas Hazrati > vikas@knoldus.com > @vhazrati

knól x

The Bloom
Programming
Language

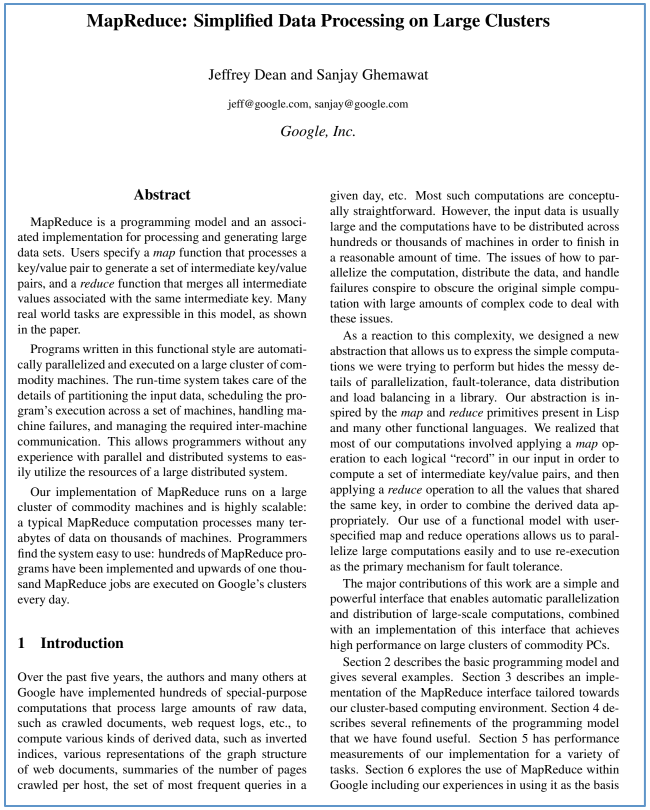
Google Map-Reduce

Google MapReduce (2004): a fault tolerant, massively parallel functional programming paradigm

- based around our friends "map" and "reduce"
- Hadoop is the open-source variant
- Database people complain that they have been doing it for a while
  - ... but it was hard to define

Fun stats circa 2012:

- Big clusters are ~4000 nodes
- Facebook had 100 PB in Hadoop
- TritonSort (UCSD) sorts 900GB/minute on a 52-node, 800-disk hadoop cluster



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

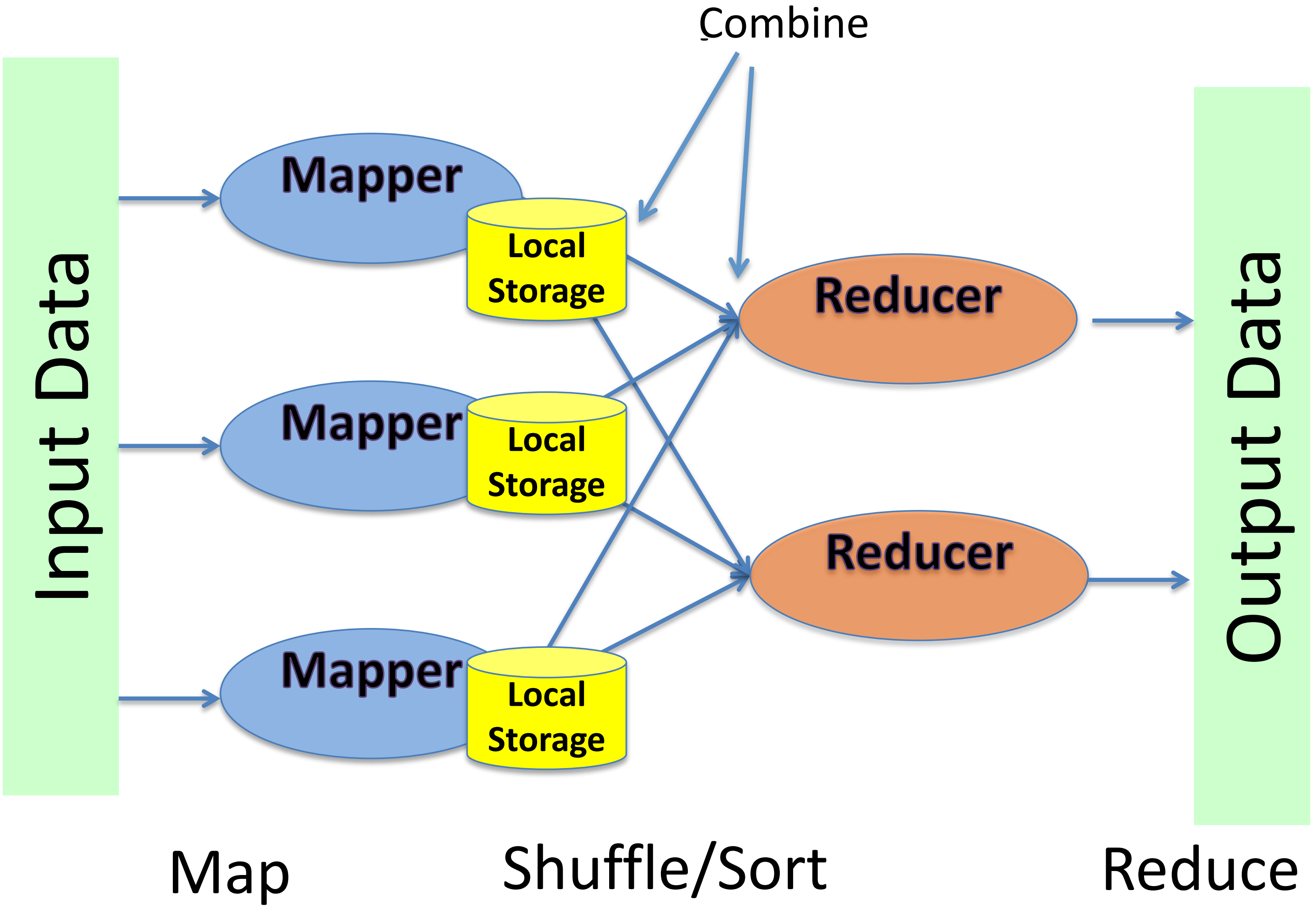
Data Model & Operations

- Map-reduce operates over collections of key-value pairs
  - millions of files (eg: web pages) drawn from the file system
- The map-reduce engine is parameterized by 3 functions:

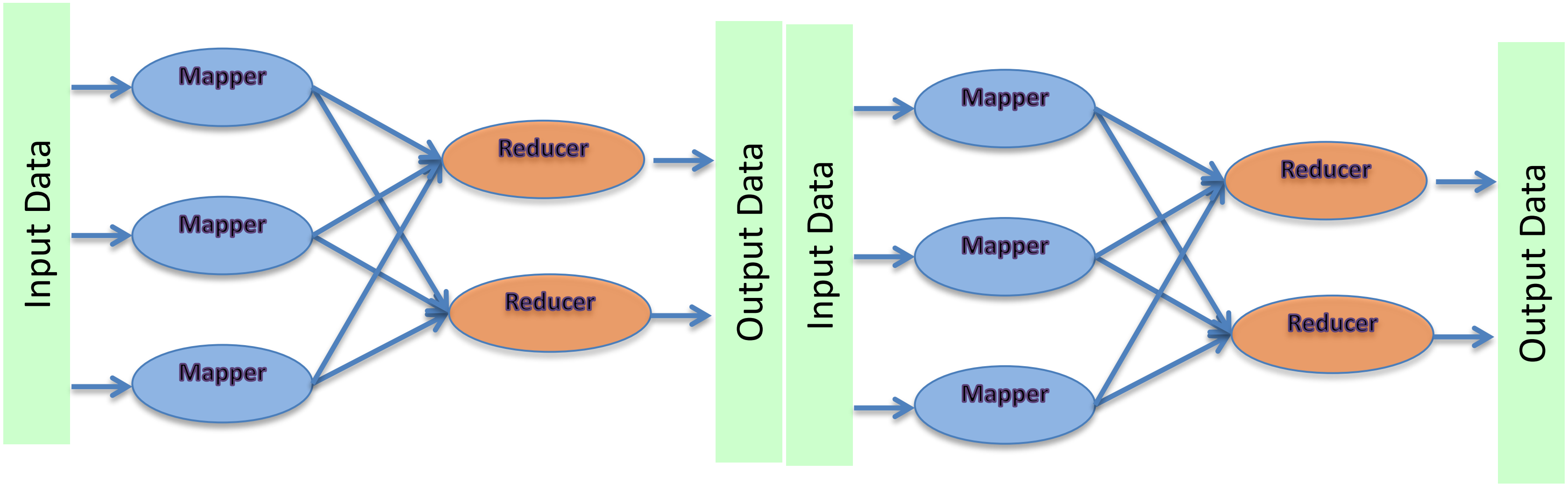
```
map     : key1 * value1        -> (key2 * value2) list
combine : key2 * (value2 list) -> value2 option
reduce  : key2 * (value2 list) -> key3 * (value3 list)
```

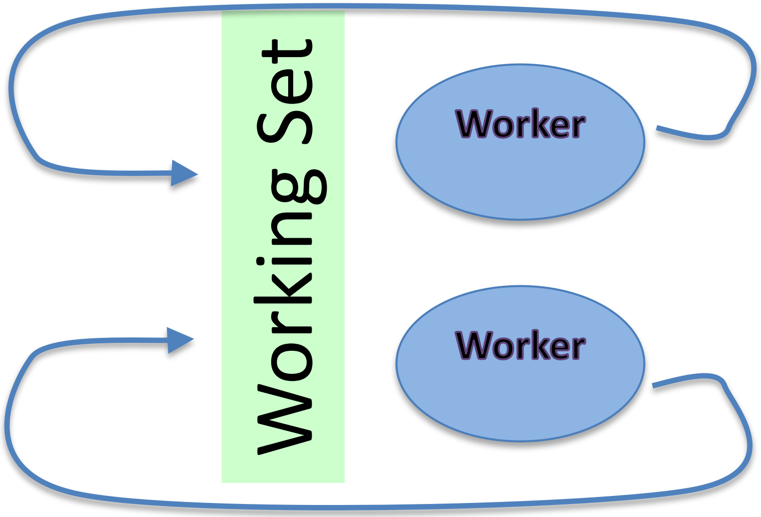
optional

Architecture

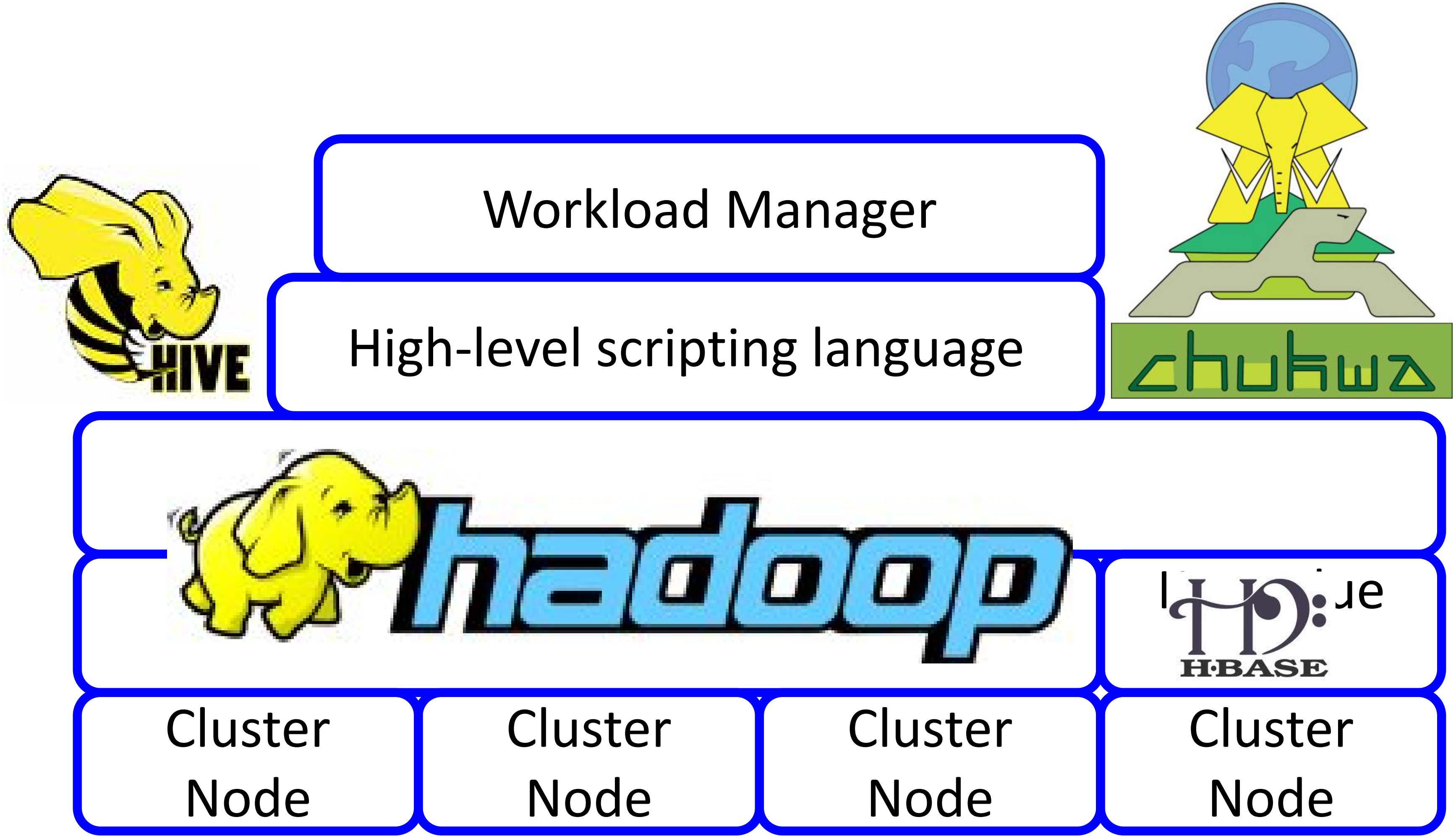


Interative Jobs are Common





A Modern Software Stack



Workload Manager

High-level scripting language

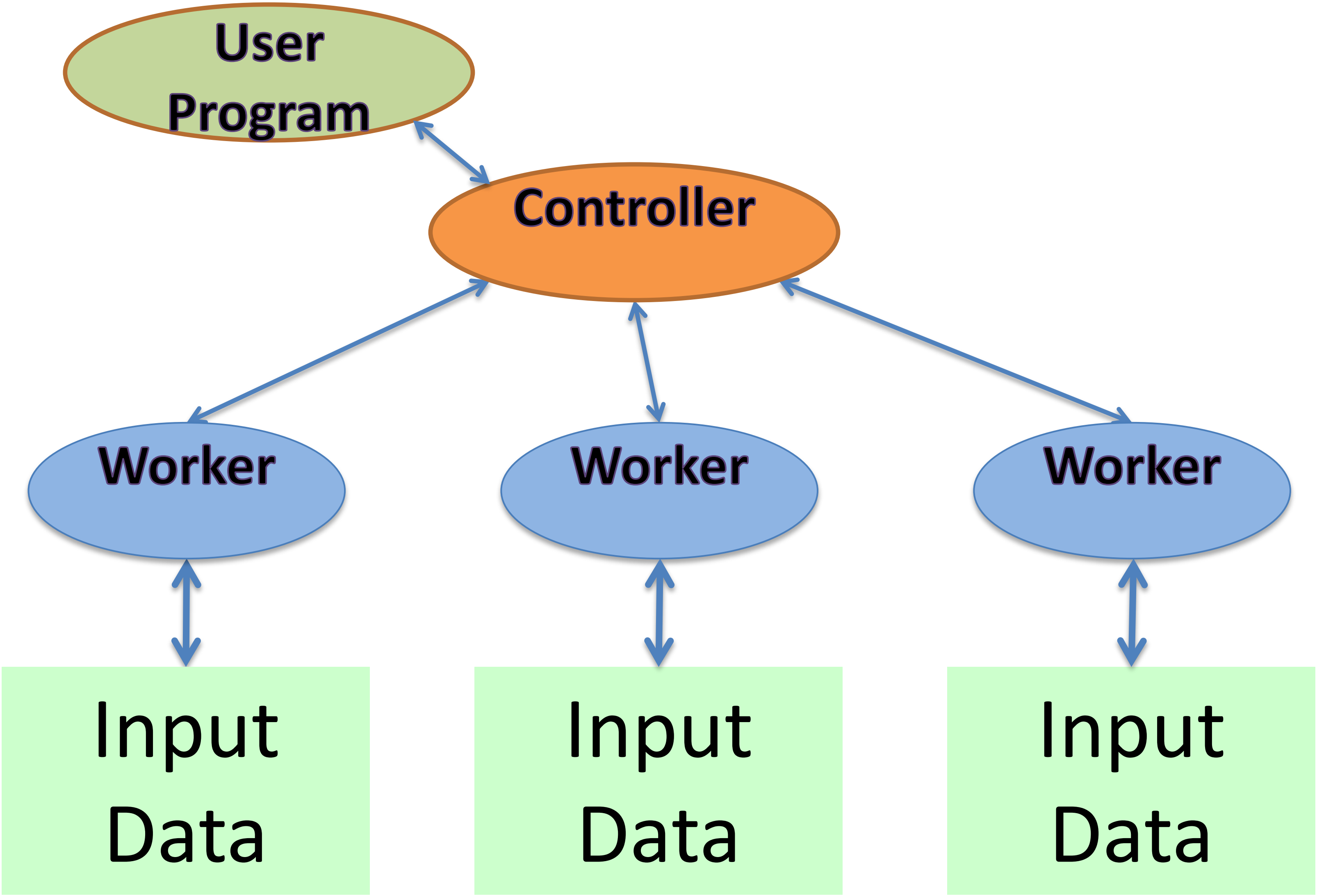
Cluster Node

Cluster Node

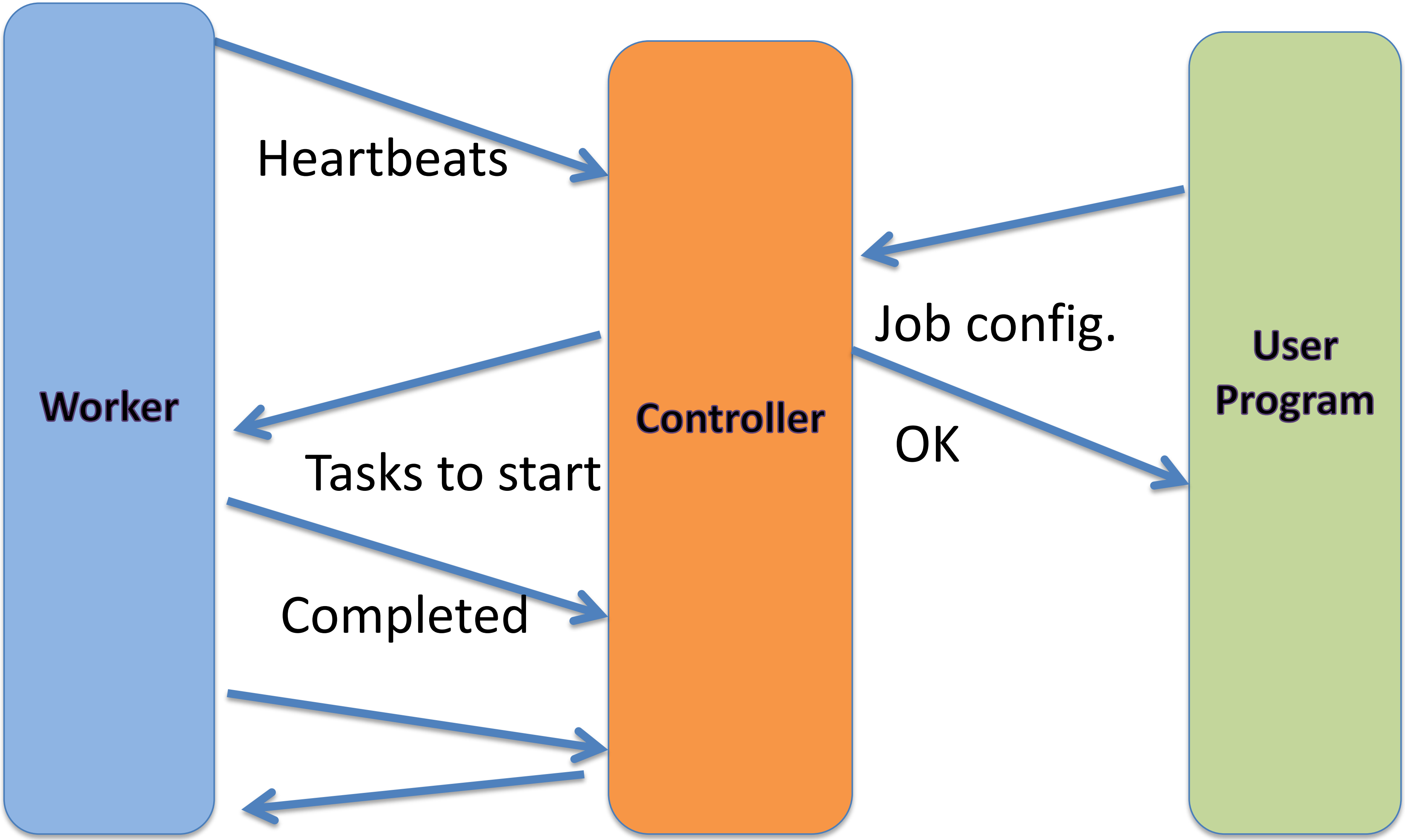
Cluster Node

Cluster Node

The Control Plane



Flow of Information



Heartbeats

Job config.

OK

Tasks to start

Completed

Jobs, Tasks and Attempts

- A single *job* is split in to many *tasks*
- Each *task* may include many calls to map and reduce
- *Workers* are long-running processes that are assigned many tasks
- Multiple workers may *attempt* the same task
  - each invocation of the same task is called an attempt
  - the first worker to finish "wins"
- Why have multiple machines attempt the same task?
  - machines will fail
    - approximately speaking: 5% of high-end disks fail/year
    - if you have 1000 machines: more than 1 failure per
    - *repeated failures become the common case*
  - machines can partially fail or be slow for some reason
    - reducers can't start until *all* mappers complete

Sort-of Functional Programming in Java

Hadoop interfaces:

```
interface Mapper<K1,V1,K2,V2> {
  public void map (K1 key,
                   V1 value,
                   OutputCollector<K2,V2> output)
  ...
}
```

```
interface Reducer<K2,V2,K3,V3> {
  public void reduce (K2 key,
                      Iterator<V2> values,
                      OutputCollector<K3,V3> output)
  ...
}
```

Word Count in Java

```
class WordCountMap implements Map {
  public void map(DocID key
                  List<String> values,
                  OutputCollector<String,Integer> output)
  {
     for (String s : values)
       output.collect(s,1);
  }
}
```

```
class WordCountReduce {
  public void reduce(String key,
                     Iterator<Integer> values,
                     OutputCollector<String,Integer> output)
  {
    int count = 0;
    for (int v : values)
      count += 1;
    output.collect(key, count)
  }
```

PLEASE RELAX
AND FOR THE SAKE OF HYGIENE,
WIPE THE
JAVA CODE OFF YOUR BRAIN

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.

END