Parallel Collections

COS 326
David Walker
Princeton University

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

slides copyright 2013-2015 David Walker and Andrew W. Appel
permission granted to reuse these slides for non-commercial educational purposes
Last Time: Parallel Programming Disciplines

• Programming with shared mutable data
• Very hard! Had to remember to:
  – acquire and release locks in the right places
  – acquire locks in the right order
  – once you are done writing your program, how do you test it?
  – how do you verify you haven't made a mistake?
• With pure functional code and parallel futures, many error modes disappear
• Are there more great abstractions like futures?
  – you betcha!
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

\[ < e_1, e_2, e_3, \ldots, e_n > \]

• Operations:
  – creation (called tabulate)
  – indexing an element in constant span
  – map
  – scan -- like a fold: \(<u, u + e_1, u + e_1 + e_2, \ldots> \log n \text{ span!} \)

• Languages:
  – Nesl [Blelloch]
  – Data-parallel Haskell
tabulate : (int -> 'a) -> int -> 'a seq

\text{tabulate } f \ n \ \Rightarrow \ <f \ 0, f \ 1, \ldots, f \ (n-1)>
\text{work} = O(n) \quad \text{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**: \((\text{int} \rightarrow 'a) \rightarrow \text{int} \rightarrow 'a \text{ seq}\)

\[
\text{tabulate } f \ n \ \equiv \ <f\ 0, f\ 1, \ldots, f\ (n-1)>
\]

work = \(O(n)\) \hspace{1cm} \text{span} = \(O(1)\)

**nth**: \('a \text{ seq} \rightarrow \text{int} \rightarrow 'a\)

\[
\text{nth } <e_0, e_1, \ldots, e(n-1)> \ i \equiv e_i
\]

work = \(O(1)\) \hspace{1cm} \text{span} = \(O(1)\)

**length**: \('a \text{ seq} \rightarrow \text{int}\)

\[
\text{length } <e_0, e_1, \ldots, e(n-1)> \equiv n
\]

work = \(O(1)\) \hspace{1cm} \text{span} = \(O(1)\)
Problems

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

(2) Write a function such that given a sequence <v0, ..., vn-1>, 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).

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

Try it!

Operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabulate f n</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>nth i s</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>length s</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Solutions

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

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

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

(* create \( n = \langle 0, 1, \ldots, n-1 \rangle \) *)

```ml
let create n =
  tabulate (fun i -> i) n
```

(* map \( f \langle v_0, \ldots, v_{n-1} \rangle = \langle f v_0, \ldots, f v_{n-1} \rangle \) *)

```ml
let map f s =
```

(* reverse \( \langle v_0, \ldots, v_{n-1} \rangle = \langle v_{n-1}, \ldots, v_0 \rangle \) *)

```ml
let reverse s =
```
let create n =
  tabulate (fun i -> i) n

let map f s =
  tabulate (fun i -> nth s i) (length s)

let reverse s =

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

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

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

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

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

(* reverse <v0, ..., vn-1> == <vn-1, ..., v0> *)
let reverse s =
  let n = length s in
  tabulate (fun i -> nth s (n-i-1)) n
One more problem

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

• Try formulating a divide-and-conquer parallel algorithm to solve this problem efficiently:

```plaintext
type paren = L | R  (* L(left) or R(right) paren *)

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

• You will need another function on sequences:

```plaintext
(* split s n divides s into (s1, s2) such that s1 is the first n elements of s and s2 is the rest
  Work = O(n) Span = O(1) *)
split : 'a sequence -> int -> 'a sequence * 'a sequence
```
### A Parallel Sequence API

<table>
<thead>
<tr>
<th>Function</th>
<th>Type Signature</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type 'a seq</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>tabulate</code></td>
<td><code>(int -&gt; 'a) -&gt; int -&gt; 'a seq</code></td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>length</code></td>
<td><code>'a seq -&gt; int</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>nth</code></td>
<td><code>'a seq -&gt; int -&gt; 'a</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>append</code></td>
<td><code>'a seq -&gt; 'a seq -&gt; 'a seq</code></td>
<td>$O(N+M)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>split</code></td>
<td><code>'a seq -&gt; int -&gt; 'a seq * 'a seq</code></td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

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

```
sum: 0 7
```

```
7 4 3 9 8
```
We have seen many sequential algorithms can be programmed succinctly using fold or reduce. Eg: sum all elements:

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>3</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
</table>

sum: 0 → 7 → 11 → 14 → 23 → 31
We have seen many sequential algorithms can be programmed succinctly using fold or reduce. Eg: sum all elements:

\[
\text{let } \text{sum} = \text{reduce} (+) 0 \text{ l}
\]

```
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
The key is **associativity**:

\[
(((((\text{init} + v_1) + v_2) + v_3) + v_4) + v_5) \equiv ((\text{init} + v_1) + v_2) + ((v_3 + v_4) + v_6)
\]

Add on processor 1: 
Add on processor 2:  

**Commutativity** allows us to reorder the elements:

\[
v_1 + v_2 \equiv v_2 + v_1
\]

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

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

\[
\begin{align*}
2 &+ 7 &+ 4 &+ 3 &+ 9 &+ 8 &+ 2 &+ 1 \\
9 & &7 & &17 & &3 
\end{align*}
\]
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))
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
If \( \text{op} \) is associative and the base case has the properties:

\[
\text{op base } X == X \quad \text{and} \quad \text{op } X \text{ base } == X
\]

then the parallel reduce is equivalent to the sequential left-to-right fold.
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
        in
        f n1 n2
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
        in
            reduce s2 in
        f n1 n2

let sum s = reduce (+) 0 s
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
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

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

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.

``` OCaml
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

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

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

```plaintext
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 right parens
First, a sequential approach

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

0 1
First, a sequential approach

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

0 1 2
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

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  0  -1!!
```

too many right parens indicates no match
First, a sequential approach

If you reach the end of the sequence, you should have no unmatched left parens
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)
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
• key insights
  – if you find () in a sequence, you can delete it without changing the balance
• 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 ... (((
• 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
• 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 ... (((

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)
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: \(O(\log N)\)
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
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)
Hey, wait a minute...

- 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 ... ))) ((( ... ))
    • for divide-and-conquer, splitting a sequence of parens is easy
    • combining two sequences where we have deleted all ()
      - ))) ... j ... ))) ((( ... ))
      - if x > k then ))) ... j ... ))) ))) ... x - k ... ))) ((( ... y ... (((
      - if x < k then ))) ... j ... ))) ((( ... k - x ... ((( ((( ... y ... (((

Dang! All that stuff about deleting parens seems complicated.
I liked the other way better, scanning from left to right, incrementing/decrementing the count.

0 1 2 1 0 -1!!
let rec mapreduce (inject: 'a -> 'b)
    (combine: 'b -> 'b -> 'b)
    (base: 'b)
    (s: 'a seq) = ...

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
  | (R, Some c) -> Some (c-1)
A nicer solution

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

type t = int option -> int option

let inject: paren -> t = (* you fill in the blanks!*)

let combine: t -> t -> t = (* you fill in the blanks!*)

let base: t = (* you fill in the blanks!*)

let finish: t -> bool = (* you fill in the blanks!*)

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

type t = int option -> int option

let inject: paren -> t = check

let combine: t -> t -> t = fun f g x → f (g x) (* compose *)

let base: t = fun x → x

let finish: t -> bool = fun f → match (f (Some 0)) with Some 0 → true | _ → false

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

type t = int option -> int option

let inject: paren -> t = check

let combine: t -> t -> t = fun f g x → f (g x) (* compose *)

let base: t = fun x → x

let finish: t -> bool = fun f → match (f (Some 0)) with Some 0 → true | _ → false

let balanced (s: paren seq) = finish (mapreduce inject combine base)

check the associativity of combine – super easy!

also check: combine base x == x super easy!
The “nicer solution” is beautiful but useless

mapreduce computes, efficiently, in parallel, a big function composition; then the “finish” function runs that function, which is when all the computation takes place, SEQUENTIALLY!

Double Dang!

type t = int option -> int option

let inject: paren -> t = check

let combine: t -> t -> t = fun f g x → f (g x) (* compose *)

let base: t = fun x → x

let finish: t -> bool = fun f → match (f (Some 0)) with Some 0 → true | _ → false

let balanced (s: paren seq) = finish (mapreduce inject combine base)
Exercise

Let \( s \) be a sequence of “digits”:

\[
\begin{align*}
\text{s} = & \quad 3 \quad 1 \quad 4 \quad 1 \quad 5 \quad 9 \quad 2 \quad 6 \\
\end{align*}
\]

Compute the decimal value of \( s \):

\[
\begin{align*}
\text{inject: } \text{int} \rightarrow \text{int} & = \quad \text{fun } d \rightarrow d \\
\text{combine (v: int) (d: int)} & = v \times 10 + d \\
\text{base} & = 0 \\
\text{combine (combine (combine (combine 0 3) 1) 4) 1} & \equiv 3141
\end{align*}
\]

Now, compute really fast in parallel:

\[
\begin{align*}
\text{mapreduce inject combine base s} & \equiv 31415926, \quad \text{right?}
\end{align*}
\]
Exercise

Let \( s \) be a sequence of “digits”:

\[
\begin{array}{c}
\text{s = } \\
\text{3 1 4 1 5 9 2 6 }
\end{array}
\]

Compute the decimal value of \( s \): 

\[
\text{inject: int -> int = fun d -> d}
\]

\[
\text{combine (v: int) (d: int) = v*10+d}
\]

\[
\text{base = 0}
\]

\[
\text{combine (combine (combine (combine 0 3) 1) 4) 1 == 3141}
\]

Now, compute really fast in parallel:

\[
\text{mapreduce inject combine base s == 31415926, right?}
\]
Another Exercise

\[ \sum_{i=0}^{n-1} f(i) \quad f : \text{int} \rightarrow \text{float} \]

inject: \( \text{int} \rightarrow \text{float} = f \)
combine \((x: \text{float}) (y: \text{float}) = x +. y \)
base = 0.

Now, compute really fast in parallel:
mapreduce inject combine base s

Is there a bug in this program?
Floating-point addition is not associative!

Consider 6-digit mantissas:

\[
\begin{align*}
.100000 \times 10^0 & \quad .400000 \times 10^{-6} \\
.0000004 \times 10^0 & \quad .400000 \times 10^{-6} \\
.000000 \times 10^0 & \quad .800000 \times 10^{-6}
\end{align*}
\]

\[
((.100000 + .0000004) + .0000004) + .0000004 = .100000
\]

\[
.100000 + (.0000004 + (.0000004 + .0000004)) = .100001
\]

For some summations, this matters a lot!
In other cases, it doesn’t matter.
So we can’t tell whether there’s a bug in the program.
PARALLEL SCAN AND PREFIX SUM
The prefix-sum problem

**Sum of Sequence:**

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>4</th>
<th>16</th>
<th>10</th>
<th>16</th>
<th>14</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Prefix-Sum of Sequence:**

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>4</th>
<th>16</th>
<th>10</th>
<th>16</th>
<th>14</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>6</td>
<td>10</td>
<td>26</td>
<td>36</td>
<td>52</td>
<td>66</td>
<td>68</td>
<td>76</td>
</tr>
</tbody>
</table>
The prefix-sum problem

val prefix_sum : int seq -> int seq

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>4</th>
<th>16</th>
<th>10</th>
<th>16</th>
<th>14</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>6</td>
<td>10</td>
<td>26</td>
<td>36</td>
<td>52</td>
<td>66</td>
<td>68</td>
<td>76</td>
</tr>
</tbody>
</table>

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: $O(\log n)$
The trick: *Use two passes*

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

input

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4</td>
<td>16</td>
<td>10</td>
<td>16</td>
<td>14</td>
<td>2</td>
</tr>
</tbody>
</table>

output

---

range sum fromleft

range 0,8 sum 76 fromleft

range 0,4 sum 36 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

range 4,8 sum 40 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

---

sum fromler

range 0,8

range 0,4

range 2,4

range 4,6

range 6,8

---

76
Example

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td>52</td>
</tr>
<tr>
<td>14</td>
<td>66</td>
</tr>
<tr>
<td>2</td>
<td>68</td>
</tr>
<tr>
<td>8</td>
<td>76</td>
</tr>
</tbody>
</table>
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., \(n\)th 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 \texttt{fromLeft}
   - Root given a \texttt{fromLeft} of 0
   - Node takes its \texttt{fromLeft} value and
     - Passes its left child the same \texttt{fromLeft}
     - Passes its right child its \texttt{fromLeft} plus its left child’s \texttt{sum}
       - as stored in part 1
   - At the leaf for sequence position \textit{i},
     - \texttt{nth output i \text{=} 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 \texttt{output}
- Invariant: \texttt{fromLeft} is sum of elements left of the node’s range

Analysis: \textit{O(n)} work, \textit{O(log n)} span
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
Given a sequence **input**, produce a sequence **output** containing only elements **v** such that \((f \ v)\) is **true**

Example: let \(f \ x = x > 10\)

\[
\text{filter } f \ <17, \ 4, \ 6, \ 8, \ 11, \ 5, \ 13, \ 19, \ 0, \ 24> \\
== \ <17, \ 11, \ 13, \ 19, \ 24>
\]

Parallelizable?

– Finding elements for the output is easy

– *But getting them in the right place seems hard*
Use parallel map to compute a \textbf{bit-vector} for true elements:

\[
\text{input} \ <17, 4, 6, 8, 11, 5, 13, 19, 0, 24> \\
\text{bits} \ <1, 0, 0, 0, 1, 0, 1, 1, 0, 1>
\]

Use parallel-prefix sum on the bit-vector:

\[
\text{bitsum} \ <1, 1, 1, 1, 2, 2, 3, 4, 4, 5>
\]

For each \(i\), if \(\text{bits}[i] == 1\) then write \(\text{input}[i]\) to \(\text{output}[\text{bitsum}[i]]\) to produce the final result:

\[
\text{output} \ <17, 11, 13, 19, 24>
\]
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:
   A. The elements less than the pivot \( O(n) \)
   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 \( T(n) = O(n) + 1T(n/2) = O(n) \)
As with mergesort, we get a $O(\log n)$ 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, $O(\log n)$ 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 $O(\log n)$ span for partition, the total best-case and expected-case span for quicksort is

$$T(n) = O(\log n) + 1T(n/2) = O(\log^2 n)$$
Step 1: pick pivot as median of three

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

Step 3: Two recursive sorts in parallel
   - Can copy back into original array (like in mergesort)
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

• 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"
If Google wants to index all the web pages (or images or gmas 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
Parallel Collections with Scala

The Bloom Programming Language

MapReduce

LINQ .NET

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

- based on 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
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:

\[
\begin{align*}
\text{map} &: \text{key1} \times \text{value1} \rightarrow (\text{key2} \times \text{value2}) \text{ list} \\
\text{combine} &: \text{key2} \times (\text{value2 list}) \rightarrow \text{value2 option} \\
\text{reduce} &: \text{key2} \times (\text{value2 list}) \rightarrow \text{key3} \times (\text{value3 list})
\end{align*}
\]
Architecture

Input Data

Map

Mapper

Local Storage

Mapper

Local Storage

Mapper

Local Storage

Reducer

Combine

Shuffle/Sort

Reducer

Output Data

Reduce
Iterative Jobs are Common

Input Data

Mapper → Reducer → Output Data

Mapper → Reducer → Output Data

Mapper → Reducer → Output Data

Mapper → Reducer → Output Data

Output Data

Working Set

Worker → Worker
A Modern Software Stack

- Workload Manager
  - High-level scripting language

- Hadoop
  - Cluster Node
  - Cluster Node
  - Cluster Node
  - Cluster Node
The Control Plane

User Program

Controller

Worker

Input Data

Worker

Input Data

Worker

Input Data
The flow of information

Worker
- Heartbeats
- Tasks to start
- Completed

Controller
- Job config.
- OK

User Program
Jobs, Tasks and Attempts

- A single *job* is split into 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: 1 failure per week
    - *repeated failures become the common case*
  - machines can partially fail or be slow for some reason
    - reducers can't start until *all* mappers complete
Hadoop interfaces:

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

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