Parallel Collections

COS 326
David Walker
Princeton University

Credits:
Dan Grossman, UW
http://homes.cs.washington.edu/~djg/teachingMaterials/spac
Bleloch, 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
• 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?
Need high-level interfaces to shield application programmers from the complex details. Complex implementations solve the problems of distribution, fault tolerance and performance.

Common abstraction: Parallel collections

Example collections: sets, tables, dictionaries, sequences
Example bulk operations: create, map, reduce, join, filter
COMPLEXITY OF PARALLEL ALGORITHMS
let x = 1 + 2 in
3 + x
let x = 1 + 2 in
3 + x
let \( x = 1 + 2 \) in \( 3 + x \)

dependence:
\( x = 1 + 2 \) happens before \( 3 + x \)
Execution of dependency diagrams: A processor can only begin executing the computation associated with a block when the computations of all of its predecessor blocks have been completed.
step 1: execute first block

Cost so far: 0
step 1: execute first block

Cost so far: 1
step 2: execute second block because all of its predecessors have been completed

Cost so far: 1
step 2: execute second block because all of its predecessors have been completed.

Cost so far: 1 + 1
let x = 1 + 2 in
3 + x

\[
\begin{align*}
\text{cost} &= 1 \\
\text{total cost} &= 1 + 1 \\
&= 2
\end{align*}
\]
parallel pair:
compute both left and right-hand sides independently
return pair of values
(easy to implement using futures)
Visualizing Computational Costs

\[(1 + 2 \mid\mid f\ 3)\]
Suppose we have 1 processor. How much time does this computation take?
Suppose we have 1 processor. How much time does this computation take? Scheduled A-B-C-D: 1 + 1 + 7 + 1
Suppose we have 1 processor. How much time does this computation take?
Schedule A-C-B-D: $1 + 1 + 7 + 1$
Visualizing Computational Costs

Suppose we have 2 processors. How much time does this computation take?
Visualizing Computational Costs

Suppose we have 2 processors. How much time does this computation take? Cost so far: 1
Suppose we have 2 processors. How much time does this computation take? Cost so far: $1 + \max(1,7)$
Suppose we have 2 processors. How much time does this computation take?
Cost so far: $1 + \max(1,7) + 1$
Suppose we have 2 processors. How much time does this computation take? Total cost: $1 + \max(1,7) + 1$. We say the schedule we used was: A-CB-D
Suppose we have 3 processors. How much time does this computation take?
Suppose we have 3 processors. How much time does this computation take? Schedule A-BC-D: \(1 + \max(1,7) + 1 = 9\)
Suppose we have infinite processors. How much time does this computation take? Schedule A-BC-D: $1 + \max(1,7) + 1 = 9$
Understanding the complexity of a parallel program is a little more complex than a sequential program

- the number of processors has a significant effect

One way to *approximate* the cost is to consider a parallel algorithm independently of the machine it runs on is to consider *two* metrics:

- **Work**: The cost of executing a program with just 1 processor.
- **Span**: The cost of executing a program with an infinite number of processors

Always good to minimize work

- Every instruction executed consumes energy
- Minimize span as a second consideration
- Communication costs are also crucial (we are ignoring them)
Parallelism

The parallelism of an algorithm is an estimate of the maximum number of processors an algorithm can profit from.

- parallelism = work / span

If work = span then parallelism = 1.
- We can only use 1 processor
- It's a sequential algorithm

If span = ½ work then parallelism = 2
- We can use up to 2 processors

If work = 100, span = 1
- All operations are independent & can be executed in parallel
- We can use up to 100 processors
Series-parallel graphs arise from execution of functional programs with parallel pairs. Also known as well-structured, nested parallelism.
let both \( f \ x \ g \ y = \)  
let \( ff = \) future \( f \ x \) in  
let \( gv = g \ y \) in  
(force \( ff, gv \) )
In general, a series-parallel graph has a source and a sink and is:
- a single node, or
- two series-parallel graphs in sequence, or
- two series-parallel graphs in parallel
However:
The results about greedy schedulers (next few slides) do apply to DAG schedules as well as series-parallel schedules!
Work and Span of Acyclic Graphs

Let's assume each node costs 1.

- **Work**: sum the nodes.
- **Span**: longest path from source to sink.
Let's assume each node costs 1.

**Work**: sum the nodes.

**Span**: longest path from source to sink.

work = 10
span = 5
Let's assume each node costs 1.

Let's assume we have 2 processors. How do we schedule computation?
Let's assume each node costs 1.

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

Option 1:
A
B G
C D
Let's assume each node costs 1.

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

Option 1:
A
B G
C D
Let's assume each node costs 1.

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

Option 1:
A
B G
C D
E H
I
Let's assume each node costs 1.

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

Option 1:
A
B G
C D
E H
I
J

F
Let's assume each node costs 1.

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

Option 1:
A
B G
C D
E H
I
J
F
Let's assume each node costs 1.

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

Option 1:
A
B G
C D
E H
H I
I
J
F
Let's assume each node costs 1.

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

Option 1:

A
B G
C D
E H
I
J
F
Let's assume each node costs 1.

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

Option 1:
A
B G
C D
E H
H I
J
J
F
Let's assume each node costs 1.

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

Option 1:
A
B G
C D
E H
H I
J
I
F

Conclusion:
How you schedule jobs can have an impact on performance.
Greedy Schedulers

Greedy schedulers will schedule some task to a processor as soon as that processor is free.

- Doesn't sound so smart!

Properties (for p processors):

- $T(p) < \frac{\text{work}}{p} + \text{span}$
  - won't be worse than dividing up the data perfectly between processors, except for the last little bit, which causes you to add the span on top of the perfect division

- $T(p) \geq \max\left(\frac{\text{work}}{p}, \text{span}\right)$
  - can't do better than perfect division between processors ($\frac{\text{work}}{p}$)
  - can't be faster than span
Greedy Schedulers

Properties (for \( p \) processors):

\[
\max\left(\frac{\text{work}}{p}, \text{span}\right) \leq T(p) < \frac{\text{work}}{p} + \text{span}
\]

Consequences:

\(\quad\) as span gets small relative to \( \frac{\text{work}}{p} \)

\(\quad\) work\( /p \) + span \(\Rightarrow\) work\( /p \)

\(\quad\) max(work\( /p \), span) \(\Rightarrow\) work\( /p \)

\(\quad\) so \( T(p) \Rightarrow\) work\( /p \) -- greedy schedulers converge to the optimum!

\(\quad\) if span approaches the work

\(\quad\) work\( /p \) + span \(\Rightarrow\) span

\(\quad\) max(work\( /p \), span) \(\Rightarrow\) span

\(\quad\) so \( T(p) \Rightarrow\) span -- greedy schedulers converge to the optimum!
PARALLEL SEQUENCES
Parallel Sequences

Parallel sequences

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

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

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

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

work = O(n) \quad \text{span} = O(1)
Parallel Sequences: Selected Operations

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

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

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

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

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

work = \(O(1)\)  \hspace{1cm} \text{span} = O(1)
Parallel Sequences: Selected Operations

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

\[\text{tabulate } f \ n = \langle f \ 0, f \ 1, \ldots, f \ (n-1) \rangle\]

\[\text{work} = O(n) \quad \text{span} = O(1)\]

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

\[\text{nth } \langle e_0, e_1, \ldots, e_{(n-1)} \rangle \ i = e_i\]

\[\text{work} = O(1) \quad \text{span} = O(1)\]

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

\[\text{length } \langle e_0, e_1, \ldots, e_{(n-1)} \rangle = n\]

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

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

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

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

\[
(* \text{ create } n = <0, 1, \ldots, n-1> *)
let create n =
  tabulate (fun i -> i) n
\]

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

<table>
<thead>
<tr>
<th>Operations</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>
Example Problems

Write a function such that given a sequence \(<v_0, \ldots, v_{n-1}>\), 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)

\[
(* \text{map } f \ <v_0, \ldots, v_{n-1}> \ == \ <f \ v_0, \ldots, f \ v_{n-1}> *)
\]

\[
\text{let map } f \ s =
\]

**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>(\text{nth } i\ s)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(\text{length } s)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example Problems

Write a function such that given a sequence \(<v_0, ..., v_{n-1}>\), 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)

\[
(* \text{map } f <v_0, ..., v_{n-1}> == <f v_0, ..., f v_{n-1}> *) \\
\text{let map } f \ s = \\
\quad \text{tabulate (fun } i -> \text{nth } s \ i) \ (\text{length } s)
\]

<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>
Example Problems

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

<table>
<thead>
<tr>
<th>Operations</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>
Example Problems

Write a function such that given a sequence \(<v_1, ..., v_{n-1}>\), reverses the sequence. with \(\text{Span} = O(1)\) and \(\text{Work} = O(n)\)

\[
(* \text{ reverse } <v_0, ..., v_{n-1}> \implies <v_{n-1}, ..., v_0> *)
let reverse s =
\]

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>
Example Problems

Write a function such that given a sequence \(<v_1, \ldots, v_{n-1}\>\), reverses the sequence. with \(\text{Span} = O(1)\) and \(\text{Work} = O(n)\)

(* reverse \(<v_0, \ldots, v_{n-1}\> == <v_{n-1}, \ldots, v_0> *\)

let reverse s =
    let n = length s in
    tabulate (fun i -> nth s (n-i-1)) n

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>
A Parallel Sequence API

type 'a seq

<table>
<thead>
<tr>
<th>Function</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabulate</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>length</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>nth</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>append</td>
<td>$O(N+M)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>split</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
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 |
```

sum:  0
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:

```
sum: 0 7 11 14 23 31
```

```
7 4 3 9 8
```
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
```

```
7 4 3 9 8
```

```
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:

\[
\text{let sum\_all } (l:\text{int list}) = \text{reduce (+) 0} l
\]

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

\[
(((init + v1) + v2) + v3) + v4) + v5) \equiv ((init + v1) + v2) + ((v3 + v4) + v6)
\]

add on processor 1

add on processor 2
The key is **associativity**:

\[
((((\text{init} + v1) + v2) + v3) + v4) + v5) = ((\text{init} + v1) + v2) + ((v3 + v4) + v6)
\]

- 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

<table>
<thead>
<tr>
<th>2</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>9</th>
<th>8</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>
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

2  7

4  3

9  8  2  1

9  8

2  1
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
Parallel Reduce

<table>
<thead>
<tr>
<th>2</th>
<th>7</th>
<th>4</th>
<th>3</th>
<th>9</th>
<th>8</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>

If \( \text{op} \) is associative and the base case has the properties:

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

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 reduce s2 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
    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.

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

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

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

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

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

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:**

<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: \textit{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 \textit{builds a tree of sums bottom-up}

- the “up” pass

Second pass \textit{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: 6 4 16 10 16 14 2 8

output: 

- range sum fromleft
  - range 0,1 sum 6
    - r 0,1 s f
  - range 1,2 sum 4
    - r 1,2 s f
  - range 2,3 sum 16
    - r 2,3 s f
  - range 3,4 sum 10
    - r 3,4 s f
  - range 4,5 sum 16
    - r 4,5 s f
  - range 5,6 sum 14
    - r 5,6 s f
  - range 6,7 sum 2
    - r 6,7 s f
  - range 7,8 sum 8
    - r 7,8 s f

- range sum 0,8
  - range 4,6 sum 30
    - range 4,8 sum 40
      - range 0,8 sum 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 $[lo, \text{middle})$
     - Right child has sum of $[\text{middle}, hi)$
     - A leaf has sum of $[i, i+1)$, i.e., $\text{nth input } i$

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
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 \( i \),
     • \( \text{nth output } i = \text{fromLeft} + \text{nth input } i \)

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

\[
\text{scan (o) } <x_1, ..., x_n>
\]

\[
= \ <x_1, x_1 \circ x_2, ..., x_1 \circ ... \circ x_n>
\]

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

\[
\text{pre\_scan (o) base } <x_1, ..., x_n>
\]

\[
= \ <\text{base, base } \circ x_1, ..., \text{base } \circ x_1 \circ ... \circ x_{n-1}>
\]

sequence with \( \circ \) 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 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[i] to output[bitsum[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:

   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 $2T(n/2)$

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

```
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 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 gmys 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
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 (e.g., 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
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)
    }
}
Architecture

Input Data

Mapper

Local Storage

Reducer

Combine

Mapper

Local Storage

Reducer

Mapper

Local Storage

Reducer

Output Data

Map

Shuffle/Sort

Reduce
Iterative Jobs are Common

Input Data

Mapper

Reducer

Mapper

Reducer

Mapper

Reducer

Output Data

Mapper

Reducer

Mapper

Reducer

Output Data

Working Set

Worker

Worker
A Modern Software Stack

For more: See COS 418, distributed systems
Parallel complexity can be described in terms of work and span.

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

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.