Parallel Sequences

COS 326
Speaker: Andrew Appel
Princeton University

Credits:
Dan Grossman, UW
http://homes.cs.washington.edu/~djg/teachingMaterials/spac
Blelloch, Harper, Licata (CMU, Wesleyan)
Programming with shared mutable data is very hard!

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?

Example: Create an index of every web page on the planet.
   – Google does that regularly!
   – There are billions of them!

Example: Search facebook for a friend or twitter for a tweet

To get big jobs done, we typically need 1000s of computers, 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

x = 1 + 2
3 + x

cost = 1

cost = 1
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

\[ x = 1 + 2 \]

\[ 3 + x \]

cost = 1

cost = 1

Cost so far: 0
Visualizing Computational Costs

step 1:
execute first block

\[
x = 1 + 2
\]

\[
3 + x
\]

Cost so far: 1

cost = 1

cost = 1
step 2: execute second block because all of its predecessors have been completed

Cost so far: 1
Visualizing Computational Costs

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

\[
\begin{align*}
  \text{cost} &= 1 \\
  3 + x &\quad \text{cost} = 1 \\
  x &= 1 + 2 \\
  \text{total cost} &= 1 + 1 \\
  &= 2
\end{align*}
\]
(1 + 2 || f 3)

parallel pair:
compute both left and right-hand sides independently
return pair of values
(easy to implement using futures)
Visualizing Computational Costs

\[(1 + 2 \parallel f 3)\]

- **A**: Cost = 1
- **B**: Cost = 1
- **C**: Cost = 7
- **D**: Cost = 1

- **B** = 1 + 2
- **C** = f 3
- **D** = ( , )
Visualizing Computational Costs

Suppose we have 1 processor. How much time does this computation take?
Suppose we have 1 processor. How much time does this computation take? Schedule 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
Suppose we have 2 processors. How much time does this computation take?
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

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!
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
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
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
E
J
F
H I
E 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
I E J
J F
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
F
J
```

**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!
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

- \( T(p) \geq \max(\frac{\text{work}}{p}, \text{span}) \)
  - 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:

– as span gets small relative to \(\frac{\text{work}}{p}\)
  • \(\frac{\text{work}}{p} + \text{span} \Rightarrow \frac{\text{work}}{p}\)
  • \(\max\left(\frac{\text{work}}{p}, \text{span}\right) \Rightarrow \frac{\text{work}}{p}\)
  • so \(T(p) \Rightarrow \frac{\text{work}}{p}\) -- greedy schedulers converge to the optimum!

– if span approaches the work
  • \(\frac{\text{work}}{p} + \text{span} \Rightarrow \text{span}\)
  • \(\max\left(\frac{\text{work}}{p}, \text{span}\right) \Rightarrow \text{span}\)
  • so \(T(p) \Rightarrow \text{span}\) – greedy schedulers converge to the optimum!
And therefore

Even though greedy schedulers are simple to implement,

they can be effective in building a parallel programming system.

and

This *supports* the idea that **work and span** are useful ways to reason about the cost of parallel programs.
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

```
tabulate f n  == <f 0, f 1, ..., f (n-1)>
work = O(n)  span = O(1)
```
Parallel Sequences: Selected Operations

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

\[
\text{tabulate } f \ n \ \Rightarrow \ <f\ 0,\ f\ 1,\ \ldots,\ f\ (n-1)> \\
\text{work} = O(n) \quad \text{span} = O(1)
\]

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

\[
\text{nth } <e_0,\ e_1,\ \ldots,\ e_{(n-1)}> \ i \ \Rightarrow \ e_i \\
\text{work} = O(1) \quad \text{span} = O(1)
\]
tabulate : (int -> 'a) -> int -> 'a seq

\[
\text{tabulate } f \text{ } n \text{ } == \text{ } \langle f \text{ } 0, \text{ } f \text{ } 1, \text{ ...}, \text{ } f \text{ } (n-1)\rangle
\]
work = O(n) \quad \text{span} = O(1)

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

\[
\text{nth } \langle e_0, \text{ } e_1, \text{ ...}, \text{ } e(n-1)\rangle \text{ } i \text{ } == \text{ } e_i
\]
work = O(1) \quad \text{span} = O(1)

length : 'a seq -> int

\[
\text{length } \langle e_0, \text{ } e_1, \text{ ...}, \text{ } e(n-1)\rangle \text{ } == \text{ } n
\]
work = O(1) \quad \text{span} = O(1)
Write a function that creates the sequence \(<0, \ldots, n-1>\) 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>
Example Problems

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

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

Operations:

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

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

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

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

let map \(f\) \(s\) =

<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>(\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)

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

Operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{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>
Write a function such that given a sequence \(<v_0, ..., v_{n-1}>\), reverses the sequence. with Span = O(1) and 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>
Example Problems

Write a function such that given a sequence \(<v_0, \ldots, v_{n-1}>\), reverses the sequence. with Span = \(O(1)\) and Work = \(O(n)\)

\[
\begin{aligned}
(* \text{ reverse } <v_0, \ldots, v_{n-1}> == <v_{n-1}, \ldots, v_0> *) \\
\text{let reverse } s =
\end{aligned}
\]

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 \(<v_0, ..., v_{n-1}>\), reverses the sequence. with Span = O(1) and Work = O(n)

(* reverse <v_0, ..., v_{n-1}> == <v_{n-1}, ..., 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>type 'a seq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tabulate</td>
<td>(int -&gt; 'a) -&gt; int -&gt; 'a seq</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>length</td>
<td>'a seq -&gt; int</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>nth</td>
<td>'a seq -&gt; int -&gt; 'a</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>append</td>
<td>'a seq -&gt; 'a seq -&gt; 'a seq</td>
<td>O(N+M)</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>(can build this from tabulate, nth, length)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>split</td>
<td>'a seq -&gt; int -&gt; 'a seq * 'a seq</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
We have seen many sequential algorithms can be programmed succinctly using fold or reduce. Eg: sum all elements:

\[
\text{sum: } \quad 0
\]

\[
\begin{array}{cccccc}
7 & 4 & 3 & 9 & 8 \\
\end{array}
\]
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
```

```
7 4 3 9 8
```

sum: 7
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
```

```
let sum_all [7; 4; 3; 9; 8] = 31
```
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:

\[
(((\text{init} + \text{v1}) + \text{v2}) + \text{v3}) + \text{v4} + \text{v5}) \equiv ((\text{init} + \text{v1}) + \text{v2}) + ((\text{v3} + \text{v4}) + \text{v5})
\]
The key is **associativity**:

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

**Commutativity not needed!**

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

\[
v_1 + v_2 = v_2 + v_1
\]

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

2  7  4  3  9  8  2  1

2  7  4  3  9  8  2  1
Parallel Sum

9 + 7 = 16

17 + 3 = 20

2 + 7 + 4 + 3 = 16

9 + 8 + 2 + 1 = 20
Parallel Sum

\[ 36 = (16 + (9 + 7)) + (20 + (17 + 3)) \]

\[ = (16 + 9 + 7) + (20 + 17 + 3) \]

\[ = 32 + 40 \]

\[ = 72 \]
let rec psum (s : int seq) : int =
    match length s with
    | 0  -> 0
    | 1  -> nth s 0
    | n  ->
        let (s1, s2) = split (n/2) s in
        let (a1, a2) = both psum s1
                        psum s2 in
        a1 + a2
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 length s with
  0 -> base
  | 1 -> nth s 0
  | n ->
    let (s1,s2) = split (n/2) s in
    let (n1, n2) = both (reduce f base) s1
                    (reduce f base) s2 in
    f n1 n2
let rec reduce (f:'a -> 'a -> 'a) (base:'a) (s:'a seq) =
match length s with
  0 -> base
| 1 -> nth s 0
| n ->
  let (s1,s2) = split (n/2) s in
  let (n1, n2) = both (reduce f base) s1
                     (reduce f base) 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 length s with
    | 0     -> base
    | 1     -> inject (nth s 0)
    | n     ->
      let (s1, s2) = split (n/2) s in
      let (n1, n2) = both
        (mapreduce inject combine base) s1
        (mapreduce inject combine base) s2 in
      combine n1 n2
A little more general

```ocaml
let rec mapreduce (inject: 'a -> 'b) (combine:'b -> 'b -> 'b) (base:'b) (s:'a seq) =
    match length s with
    | 0 -> base
    | 1 -> inject (nth s 0)
    | n -> let (s1,s2) = split (n/2) s in
            let (n1, n2) = both
                (mapreduce inject combine base) s1
                (mapreduce inject combine base) s2 in
            combine n1 n2

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
```
DON’T PARALLELIZE AT TOO FINE A GRAIN
Parallel Reduce with Sequential Cut-off

When data is small, the overhead of parallelization isn't worth it. Revert to the sequential version!

let Sequential_reduce f base (s:'a seq) =
  let rec g i x =
    if i<0 then x else g (i-1) (f (nth a i) x)
  in g (length s - 1)

let SHORT = 1000

let rec reduce (f:'a -> 'a -> 'a) (base:'a) (s:'a seq) =
  if length s < SHORT
  then sequential_reduce f base s
  else let (s1,s2) = split ((length s)/2) s in
    let (n1, n2) = both (reduce f base) s1
    (reduce f base) s2 in
    f n1 n2
BALANCED PARENTHESES
The Balanced Parentheses Problem

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

- balanced: ()(()())
- not balanced: ( ( )
- not balanced: )()
- not balanced: ()())

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

```
type paren = L | R     (* L(eft) or R(ight) paren *)

let balanced (ps : paren seq) : bool = ...
```
First, a sequential approach

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

0

Warning! This solution does not generalize to a parallel map/reduce!
First, a sequential approach

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

Warning! This solution does not generalize to a parallel map/reduce!
First, a sequential approach

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

0 1 2

Warning! This solution does not generalize to a parallel map/reduce!
First, a sequential approach

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

Warning! This solution does not generalize to a parallel map/reduce!
First, a sequential approach

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

Warning! This solution does not generalize to a parallel map/reduce!
First, a sequential approach

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

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

if you reach the end of the 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

That was easy enough. But the “check” function is not associative, that means it can’t be used in a parallel “reduce”. That’s what I was warning about!
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
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 \( x \geq k \) then ))) ... j ... ))) ))) ... x - k ... ))) ((( ... y ... (((

– if \( x \leq k \) then ))) ... j ... ))) ((( ... k - x ... ((( ((( ... y ... (((
let rec matcher s =
  match length s with
  0 -> (0, 0)
  | 1 -> (match nth s 0 with
          | L -> (0, 1)
          | R -> (1, 0))
  | n ->
      let (left, right) = split (n/2) s in
      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 matcher s = ...

(* true if s is a sequence of balanced parens *)
let balanced s = 
  match matcher s with 
  | (0, 0) -> true 
  | (j,k) -> false
let rec matcher s =
    match length s with
    0 -> (0, 0)
    | 1 -> (match nth s 0 with
            | L -> (0, 1)
            | R -> (1, 0))
    | n ->
        let (left, right) = split (n/2) s in
        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 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

Work: O(N)  
Span: O(log N)
Using a Parallel Fold

```ocaml
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 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

The map-reduce paradigm, inspired by functional programming, is a winner when it comes to big-data processing (more about that in the next lecture).
Sanity checks

let combine \((j, k) \ (x, y)\) =
  if \(x > k\) then \((j + (x - k), y)\)
  else \((j, (k - x) + y)\)

base = (0,0)

check the associativity of combine

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

Prove for yourself:

\[\text{combine (combine (j,k) \ (x,y)) \ (a,b) = combine (j,k) \ (combine (x,y)\ (a,b))}\]

\[\text{combine (j,k) \ (0,0) = (j,k)}\]

\[\text{combine (0,0) \ (j,k) = (j,k)}\]