Programming with Parallel Sequences

COS 326
Speaker: Andrew Appel
Princeton University
Last Time: Parallel Sequences, Parallel Collections

The parallel sequence abstraction is powerful:

- tabulate
- nth
- length
- map
- split
- scan
  - used to implement prefix-sum
  - clever 2-phase implementation
  - used to implement filters
- sorting
ASSIGNMENT #7: PROGRAMMING WITH PARALLEL SEQUENCES
Do the reading . . .

Chapter 2, “Search Engine Indexing”

(On reserve for this course, available at blackboard.princeton.edu, select this course, then “reserves”)

(Read also Chapter 3, “Page Rank” so you can appreciate what you were doing in Assignment 5 . . .)
End goal: develop a system for efficiently computing US population queries by geographic region
### map-reduce API for Assignment 7

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tabulate (f: int-&gt;α) (n: int) : α seq</code></td>
<td>Create seq of length n, element i holds f(i)</td>
<td>( n )</td>
</tr>
<tr>
<td><code>seq_of_array: α array -&gt; α seq</code></td>
<td>Create a sequence from an array</td>
<td>1</td>
</tr>
<tr>
<td><code>array_of_seq: α seq -&gt; α array</code></td>
<td>Create an array from a sequence</td>
<td>1</td>
</tr>
<tr>
<td><code>iter (f: α -&gt; unit): α seq -&gt; unit</code></td>
<td>Applying f on each element in order.</td>
<td>( n )</td>
</tr>
<tr>
<td><code>length: α seq -&gt; int</code></td>
<td>Return the length of the sequence</td>
<td>1</td>
</tr>
<tr>
<td><code>empty: unit -&gt; α seq</code></td>
<td>Return the empty sequence</td>
<td>1</td>
</tr>
<tr>
<td><code>cons: α -&gt; α seq -&gt; α seq</code></td>
<td>Cons a new element on the beginning</td>
<td>( n )</td>
</tr>
<tr>
<td><code>singleton: α -&gt; α seq</code></td>
<td>Return the sequence with a single element</td>
<td>1</td>
</tr>
<tr>
<td><code>append: α seq -&gt; α seq -&gt; α seq</code></td>
<td>(nondestructively) concatenate two sequences</td>
<td>( m+n )</td>
</tr>
<tr>
<td><code>nth: α seq -&gt; int -&gt; α</code></td>
<td>Get the nth value in the sequence. Indexing is zero-based.</td>
<td>1</td>
</tr>
<tr>
<td><code>map (f: α -&gt; β) -&gt; α seq -&gt; β seq</code></td>
<td>Map the function f over a sequence</td>
<td>1</td>
</tr>
<tr>
<td><code>reduce (f: α -&gt; α -&gt; α) (base: α): α seq -&gt; α</code></td>
<td>Fold a function f over the sequence. f must be associative, and base must be the unit for f.</td>
<td>( n )</td>
</tr>
<tr>
<td><code>mapreduce: (α-&gt;β)-&gt;(β-&gt;β-&gt;β)-&gt;β -&gt; α seq -&gt; β</code></td>
<td>Combine the map and reduce functions.</td>
<td>( n )</td>
</tr>
<tr>
<td><code>flatten: α seq seq -&gt; α seq</code></td>
<td>Flatten ( [[a0;a1]; [a2;a3]] = [a0;a1;a2;a3] )</td>
<td>( n )</td>
</tr>
<tr>
<td><code>repeat (x: α) (n: int) : α seq</code></td>
<td>Repeat x 4 = ([x;x;x;x])</td>
<td>( n )</td>
</tr>
<tr>
<td><code>zip: (α seq * β seq) -&gt; (α * β) seq</code></td>
<td>Zip [a0;a1] [b0;b1;b2] = ([(a0,b0);(a1,b1)])</td>
<td>( n )</td>
</tr>
<tr>
<td><code>split: α seq -&gt; int -&gt; α seq * α seq</code></td>
<td>Split ([a0;a1;a2;a3] = ([a0],[a1;a2;a3]))</td>
<td>( n )</td>
</tr>
<tr>
<td><code>scan: (α-&gt;α-&gt;α) -&gt; α -&gt; α seq -&gt; α seq</code></td>
<td>Scan f b ([a0;a1;a2;...]= ) ([f b a0; f (f b a0) a1; f (f (f b a0) a1) a2; ...])</td>
<td>( n )</td>
</tr>
</tbody>
</table>
NESL

These parallel-sequence operators are inspired by the NESL language (and system) developed by Guy Blelloch.

http://www.cs.cmu.edu/~scandal/nesl.html

NESL is a parallel language developed at Carnegie Mellon. It integrates ideas from the theory community (parallel algorithms), the languages community (functional languages) and the systems community (many of the implementation techniques). The most important new ideas behind NESL are

1. **Nested data parallelism**: this feature offers the benefits of data parallelism, concise code that is easy to understand and debug, while being well suited for irregular algorithms, such as algorithms on trees, graphs or sparse.

2. **A language-based performance model**: this gives a formal way to calculate the **work and depth** of a program. These measures can be related to running time on parallel machines.
IMPLEMENTATION OF PARALLEL SEQUENCES
Data Centers: *Lots* of Connected Computers!

- 2 CPU chips
- 48 cores
Real Machines

Chip

Core 1
ALU
L1 cache

Core 2
ALU
L1 cache

Core 3
ALU
L1 cache

Core 4
ALU
L1 cache

L2 cache
Real Machines

Board

- Core 1
- Core 2
- Core 3
- Core 4
- L1 cache
- L1 cache
- L1 cache
- L1 cache
- L2 cache
- ALU
- ALU
- ALU
- ALU

- Core 1
- Core 2
- Core 3
- Core 4
- L1 cache
- L1 cache
- L1 cache
- L1 cache
- L2 cache
- ALU
- ALU
- ALU
- ALU

RAM

“Disk”
Real Machines

Shelf

Rack

Server room
Real Machines

s: int seq
length(s) = 10^9
Real Machines

10^5 elements per processor

10^4

10^9
tabulate (f: \(\text{int} \rightarrow \alpha\)) (n: \text{int}) : \alpha \text{ seq}

Create seq of length n, element i holds f(i)

Real Machines

10^5 elements per processor

<table>
<thead>
<tr>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1</td>
</tr>
</tbody>
</table>
module type S = sig
  type 'a t
  val tabulate : (int -> 'a) -> int -> 'a t
  val seq_of_array : 'a array -> 'a t
  val array_of_seq : 'a t -> 'a array
  val iter : ('a -> unit) -> 'a t -> unit
  val length : 'a t -> int
  val empty : unit -> 'a t
  val cons : 'a -> 'a t -> 'a t
  val singleton : 'a -> 'a t
  val append : 'a t -> 'a t -> 'a t
  val nth : 'a t -> int -> 'a
  val map : ('a -> 'b) -> 'a t -> 'b t
  val map_reduce : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'a t -> 'b
  val reduce : ('a -> 'a -> 'a) -> 'a t -> 'a
  val flatten : 'a t t -> 'a t
  val repeat : 'a -> int -> 'a t
  val zip : ('a t * 'b t) -> ('a * 'b) t
  val split : 'a t -> int -> 'a t * 'a t
  val scan: ('a -> 'a -> 'a) -> 'a -> 'a t
end

module ArraySeq : S = struct
  type 'a t = 'a array
  let length = Array.length
  let empty () = Array.init 0 (fun _ -> raise (Invalid_argument ""))
  let singleton x = Array.make 1 x
  let append = Array.append
  let cons (x:'a) (s:'a t) = append (singleton x) s
  let tabulate f n = Array.init n f
  let nth = Array.get
  let map = Array.map
  let map_reduce = (...)
  let reduce = (...)
  let flatten = (...)
  let repeat = (...)
  let zip = (...)
  let split = (...)
  let scan = (...)
  ...
end
module type S = sig
  type 'a t
  val tabulate : (int -> 'a) -> int -> 'a t
  val seq_of_array : 'a array -> 'a t
  val array_of_seq : 'a t -> 'a array
  val iter: ('a -> unit) -> 'a t -> unit
  val length : 'a t -> int
  val empty : unit -> 'a t
  val cons : 'a -> 'a t -> 'a t
  val singleton : 'a -> 'a t
  val append : 'a t -> 'a t -> 'a t
  val nth : 'a t -> int -> 'a
  val map : ('a -> 'b) -> 'a t -> 'b t
  val map_reduce : ('a -> 'b) -> ('b -> 'b -> 'b) -> 'b -> 'a t -> 'b
  val reduce : ('a -> 'a -> 'a) -> 'a t -> 'a
  val flatten : 'a t t -> 'a t
  val repeat : 'a -> int -> 'a t
  val zip : ('a t * 'b t) -> ('a * 'b) t
  val split : 'a t -> int -> 'a t * 'a t
  val scan: ('a -> 'a -> 'a) -> 'a -> 'a t
end

module Accounting (M: S) : SCount =
  struct
    let work = ref 0
    let span = ref 0
    let reporting name f x = ...
    module SM = struct
      type 'a t = 'a M.t
      let tabulate f n = (cost n 1;
        let s = !span in
        let smax = ref s in
        let z = M.tabulate (fun x -> let y = f x in
          smax := max (!smax) (!span);
          span := s; y) n
          in span := !smax; z)
      let length a = (cost 1 1; M.length a)
      let append a b = (cost (M.length a + M.length b) 1;
        M.append a b)
      ...
    end
  end
Open Sequence
module A = Accounting(ArraySeq)
module M = A.SM

let s1 = M.seq_of_array [|1;2;3;4;5|]
let f (s: int M.seq) = M.map (fun i -> i+1) s
let s2 = A.reporting "test1" f s1
let r = Array.to_list (M.array_of_seq s2)

(* Prints: *)
test1 work=5 span=1

r : int list = [2;3;4;5;6]
Discussion

How to use these operators to make an inverted index?

key: URL     value: contents of web page (HTML)

sequence of words

key: word     value: sequence of (URL,position-in-seq) pairs
How to use these operators to make an inverted index?

- **key: URL**          **value: word seq**
- **key: word**          **value: (URL*int) seq**
How to use these operators to make an inverted index?

\[
\text{key: URL} \quad \text{value: word seq} \\
(\text{URL} \ast (\text{word seq})) \quad \text{seq}
\]

\[
\text{key: word} \quad \text{value: (URL*int) seq}
\]
Discussion

How to use these operators to make an inverted index?

Input web pages: \((URL^* \ (word \ seq)) \ seq\)

key: word \quad value: (URL^*int) seq

finite map: \(\text{word} \rightarrow ((URL^*int)\text{seq})\)

Implement by balanced binary search tree (such as 2-3 tree) from OCaml’s Map library
Now, let’s focus on a single web page, one element of this sequence of web pages.

Input web pages: \((\text{URL}^* \ (\text{word seq}))\) seq

word \((\text{URL}^*\text{int}\text{seq})\) Map.t
Discussion

(URL* (word seq))

word ((URL*int)seq) Map.t

(is \mapsto [(foo.com,2)])
(play \mapsto [(foo.com,1)])
(the \mapsto [(foo.com,0); (foo.com,3)])
(thing \mapsto [(foo.com,4)])
Discussion

(bar.com, [play;the;thing])

(play → [(bar.com,0)])
(the → [(bar.com,1)])
(thing → [(bar.com,2)])

(foo.com, [the;play;is;the;thing])

(is → [(foo.com,2)])
(play → [(foo.com,1)])
(the → [(foo.com,0); (foo.com,3)])
(thing → [(foo.com,4)])
(bar.com, [play;the;thing])

play $\rightarrow$ [(bar.com,0)]
the $\rightarrow$ [(bar.com,1)]
thing $\rightarrow$ [(bar.com,2)]

(foo.com, [the;play;is;the;thing])

is $\rightarrow$ [(foo.com,2)]
play $\rightarrow$ [(foo.com,1)]
the $\rightarrow$ [(foo.com,0); (foo.com,3)]
thing $\rightarrow$ [(foo.com,4)]
How to use these operators to make an inverted index?

Input web pages: \((URL^* (\text{word seq}))\) \(\text{seq}\)

word \(((URL^*\text{int})\text{seq})\) Map.t

Reduce!
Discussion

How to use these operators to make an inverted index?

Input web pages: \((URL* (word \text{seq}))\) \text{seq}

word \(((URL*\text{int})\text{seq})\) \text{Map.t}

This has been a brief introduction to give you a flavor of what you have to do. More details in the homework . . . but not necessarily a lot more – you’ll have to think for yourself.

And: There is not “one true solution” to this homework.
Don’t “hide” work and span!

Open Sequence
module A = Accounting(ArraySeq)
module M = A.SM

let rec costly (n: int) = if n=0 then 1 else costly (n-1) + costly (n-1)

let s1 = M.seq_of_array [|51;52;53;54;55|]
let f (s: int M.seq) = M.map costly s
let s2 = A.reporting "test2" f s1
let r = Array.to_list (M.array_of_seq s2)

(* Prints: *)

test2 work=5 span=1

r : int list = [2;3;4;5;6]
CONCLUSION
Summary

By using the Parallel Sequence operators to combine pure-functional implementations of primitive functions, you can:

• Write highly parallel programs
• that scale to many processors
• with fault-tolerance built in
• that compute the same answer deterministically no matter how the parallel execution goes
• while still thinking at a high level of abstraction, independent of the gory details of your parallel machine.