Programming with Parallel Sequences

COS 326

Speaker: Andrew Appel

Princeton University

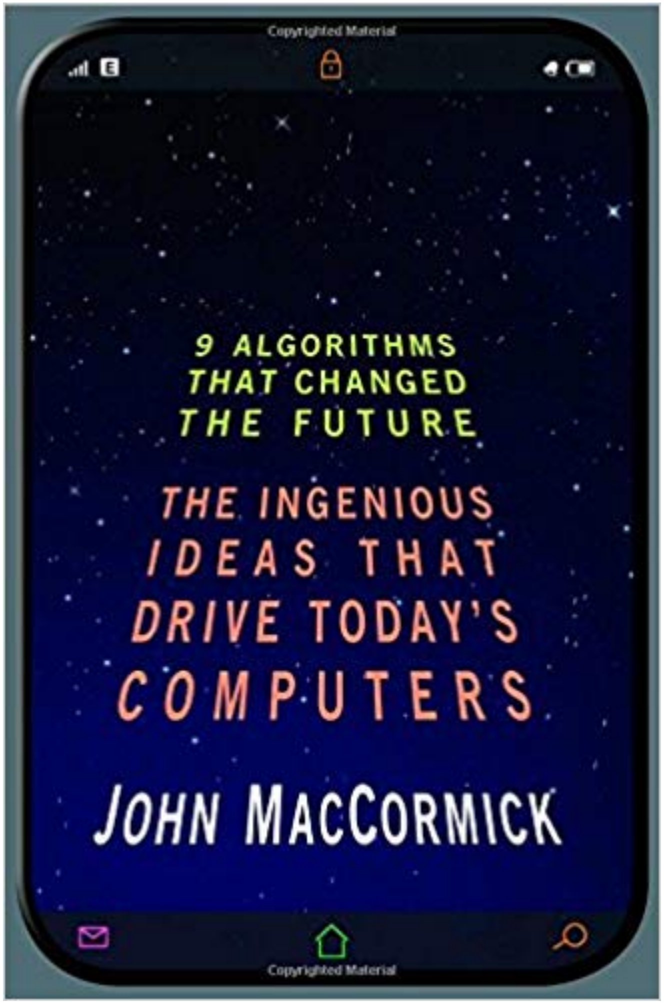
Last Time: Parallel Sequences, Parallel Collections

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 . . .



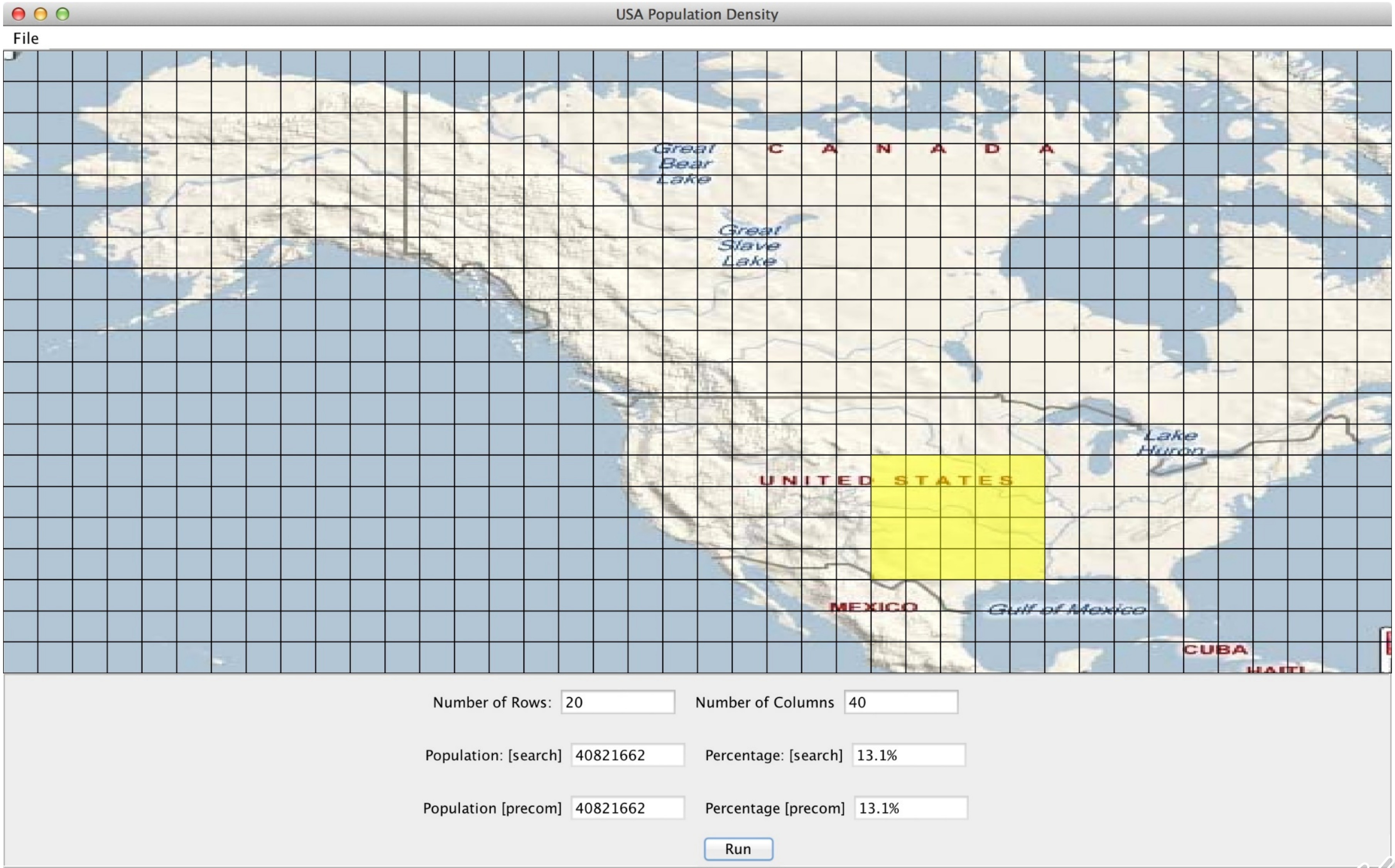
Chapter 2, "Search Engine Indexing"

(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 . . .)

US Census Queries



End goal: develop a system for efficiently computing US population queries by geographic region

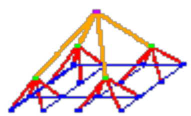
map-reduce API for Assignment 7

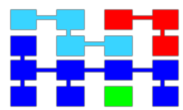
		Work	Span
tabulate (f: int->α) (n: int) : α seq	Create seq of length n, element i holds f(i)	n	1
seq_of_array: α array -> α seq	Create a sequence from an array	1	1
array_of_seq: α seq -> α array	Create an array from a sequence	1	1
iter (f: α -> unit): α seq -> unit	Applying f on each element in order.	n	n
length: α seq -> int	Return the length of the sequence	1	1
empty: unit -> α seq	Return the empty sequence	1	1
cons: α -> α seq -> α seq	cons a new element on the beginning	n	1
singleton: α -> α seq	Return the sequence with a single element	1	1
append: α seq -> α seq -> α seq	(nondestructively) concatenate two sequences	m+n	1
nth: α seq -> int -> α	Get the nth value in the sequence. Indexing is zero-based.	1	1
map (f: α -> β) -> α seq -> β seq	Map the function f over a sequence	n	1
reduce (f: α -> α -> α) (base: α): α seq -> α	Fold a function f over the sequence. f must be associative, and base must be the unit for f.	n	log n
mapreduce: (α->β)->(β->β->β)-> β -> α seq -> β	Combine the map and reduce functions.	n	log n
flatten: α seq seq -> α seq	flatten [[a0;a1]; [a2;a3]] = [a0;a1;a2;a3]	n	log n
repeat (x: α) (n: int) : α seq	repeat x 4 = [x;x;x;x]	n	1
zip: (α seq * β seq) -> (α * β) seq	zip [a0;a1] [b0;b1;b2] = [(a0,b0);(a1,b1)]	n	1
split: α seq -> int -> α seq * α seq	split [a0;a1;a2;a3] 1= ([a0],[a1;a2;a3])	n	1
scan: (α->α->α) -> α -> α seq -> α seq	scan f b [a0;a1;a2;...] = [f b a0; f (f b a0) a1; f (f (f b a0) a1) a2; ...]	n	log n

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



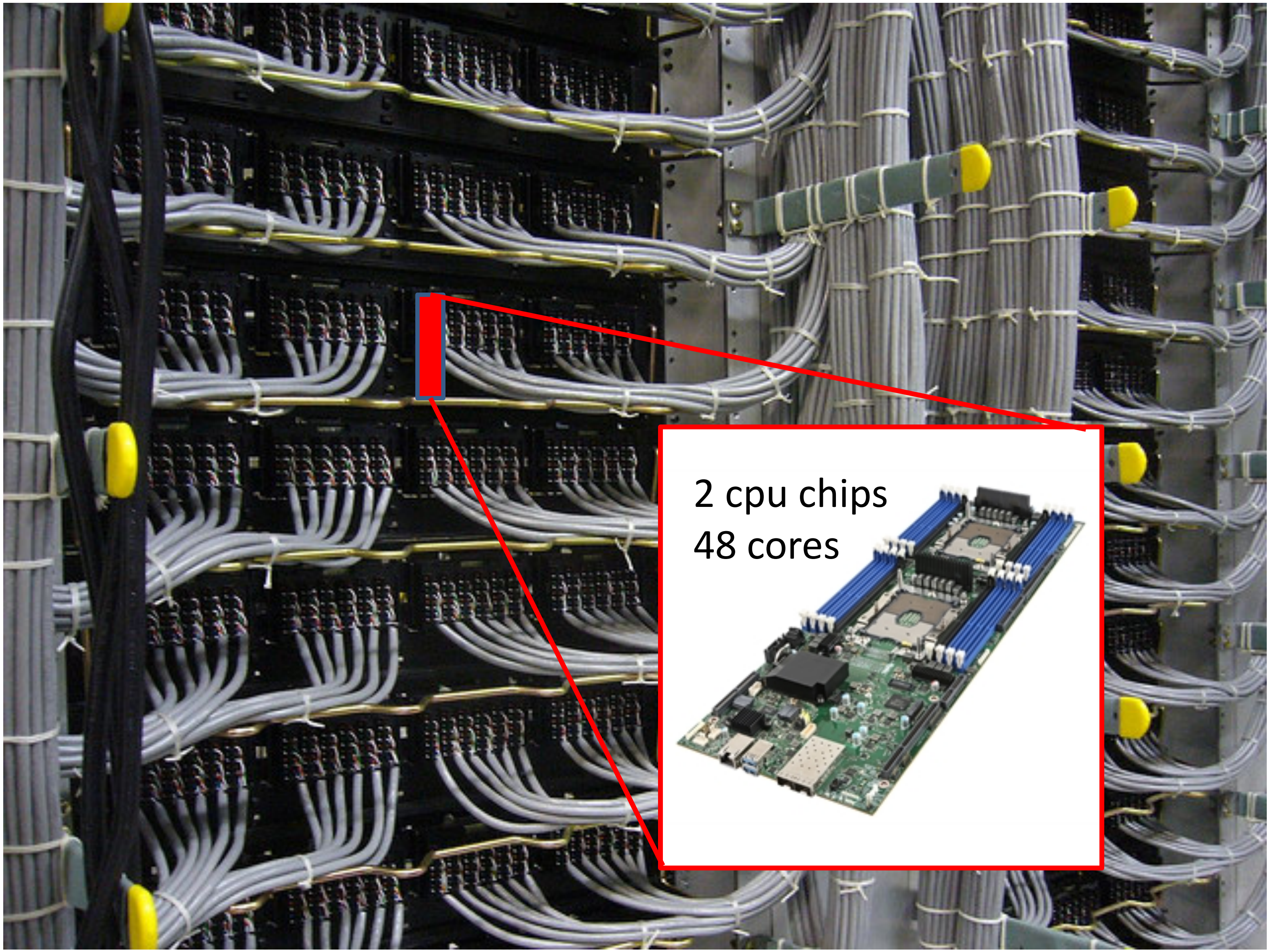


NESL is a parallel language developed at Carnegie Mellon. It integrates ideas from the theory community (parallel algorithms), the languages community (functional languages) and the systems community (many of the implementation techniques). The most important new ideas behind NESL are

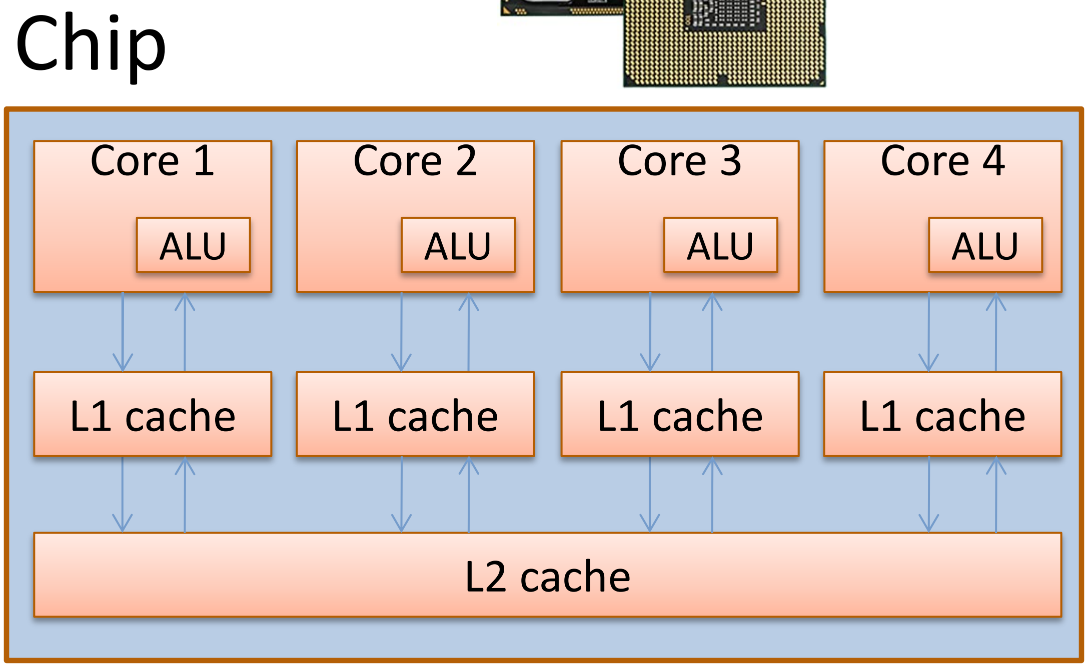
1. **Nested data parallelism:** this feature offers the benefits of data parallelism, concise code that is easy to understand and debug, while being well suited for irregular algorithms, such as algorithms on trees, graphs or sparse.
2. **A language-based performance model:** this gives a formal way 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!

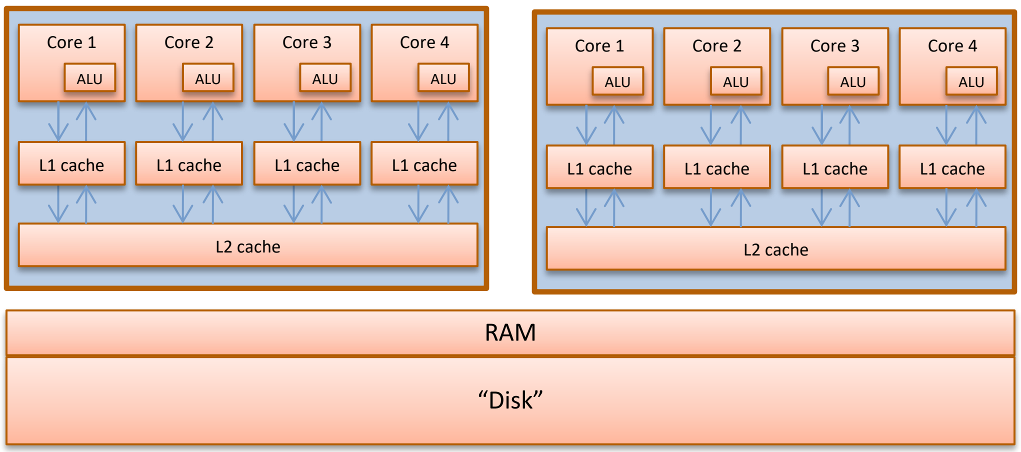


Real Machines



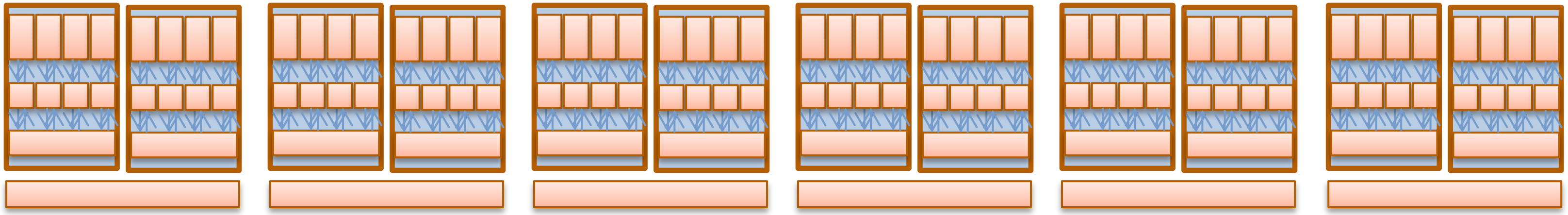
Real Machines

Board



Real Machines

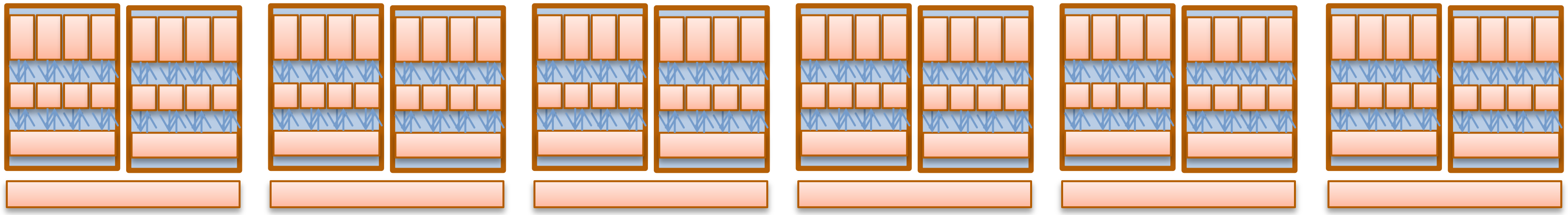
Shelf



Rack

Server room

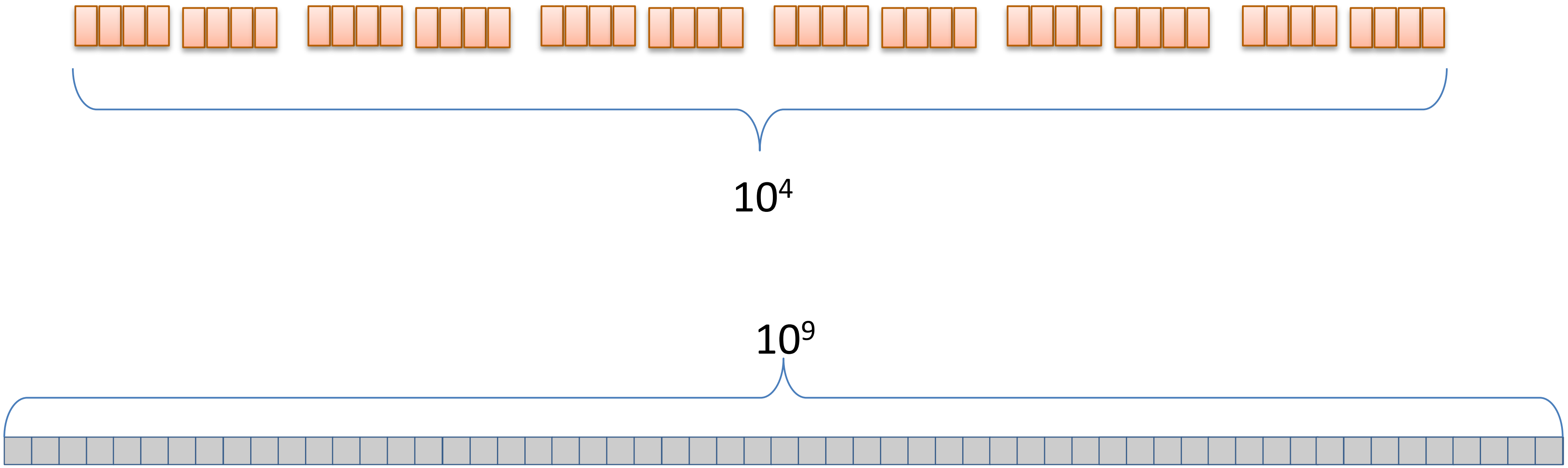
Real Machines



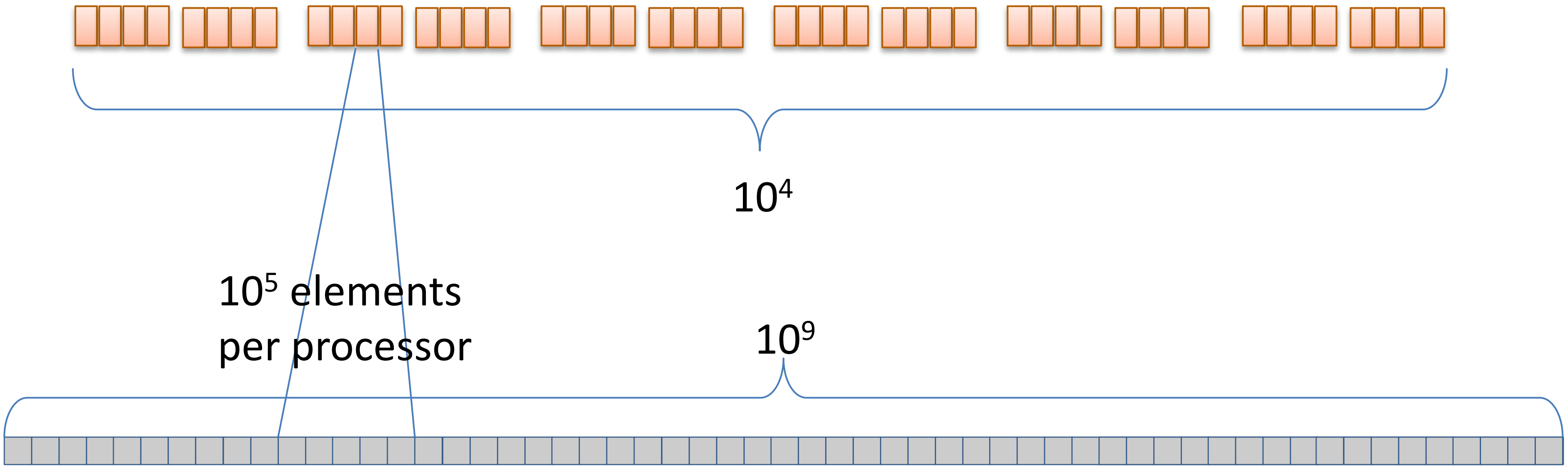
s: int seq

length(s) = 10^9

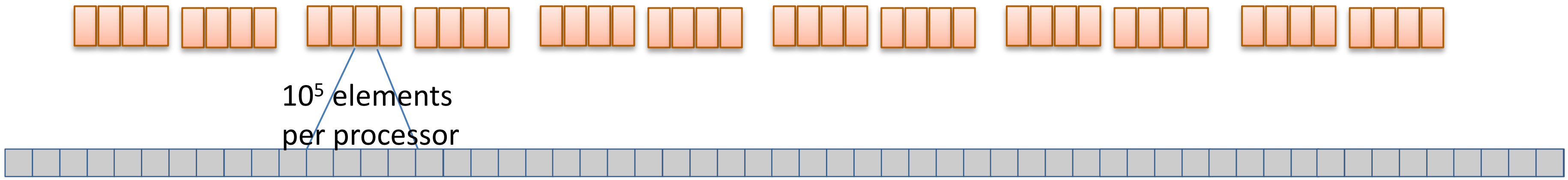
Real Machines



Real Machines



Real Machines



		Work	Span
tabulate (f: int->α) (n: int) : α seq	Create seq of length n, element i holds f(i)	n	1

API for Assignment 7

```
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 ->
  val nth : 'a t -> int -> 'a
  val map : ('a -> 'b) -> 'a t
  val map_reduce : ('a -> 'b
  val reduce : ('a -> 'a -> 'a)
  val flatten : 'a t t -> 'a t
  val repeat : 'a -> int -> 'a
  val zip : ('a t * 'b t) -> ('a
  val split : 'a t -> int -> 'a t
  val scan: ('a -> 'a -> 'a) ->
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
  . . .
end
```

Work/Span estimation

```
module type S = sig
  type 'a t
  val tabulate : (int -> 'a) -> int -> 'a
  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 -
  val reduce : ('a -> 'a -> 'a) -> '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
```

How to use it

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]

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: nothing *)

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

Discussion

How to use these operators to make an inverted index?

key: URL value: word seq

↓

key: word value: (URL*int) seq

Discussion

How to use these operators to make an inverted index?

~~key: URL value: word seq~~

(URL * (word seq)) seq

↓

key: word value: (URL*int) seq

Discussion

How to use these operators to make an inverted index?

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

key: word value: (URL *int) seq

finite map: word→((URL*int)seq)

Implement by balanced binary search tree (such as 2-3 tree) from OCaml's Map library

Discussion

How to use these operators to make an inverted index?

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

Now, let's focus on a *single* web page,
one element of this sequence of web pages

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

Discussion

(URL* (word seq))

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

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

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

Discussion

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

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

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

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

Discussion

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

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

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

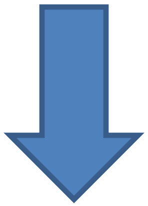
is $\longmapsto$ [(foo.com,2)]
play $\longmapsto$ [(foo.com,1)]
the $\longmapsto$ [(foo.com,0); (foo.com,3)]
thing $\longmapsto$ [(foo.com,4)]

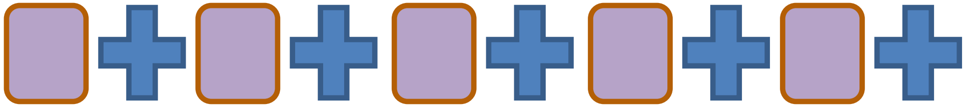
is $\longmapsto$ [(foo.com,2)]
play $\longmapsto$ [[(bar.com,0); (foo.com,1)]
the $\longmapsto$ [(bar.com,1); (foo.com,0); (foo.com,3)]
thing $\longmapsto$ [[(bar.com,2); (foo.com,4)]

Discussion

How to use these operators to make an inverted index?

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





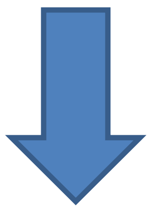
Reduce!

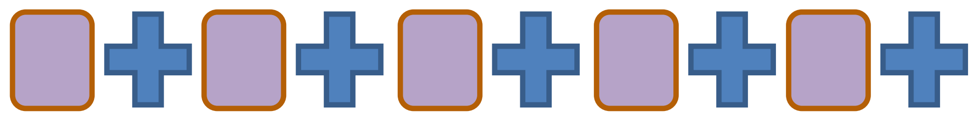
word ((URL*int)seq) Map.t

Discussion

How to use these operators to make an inverted index?

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





Reduce!

word ((URL*int)seq) 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]
```

Ideally, each function you write in OCaml should do a *small* amount of computation (other than nested calls to the M operators).

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.