Parallel Prefix Scan and Assignment 7

COS 326

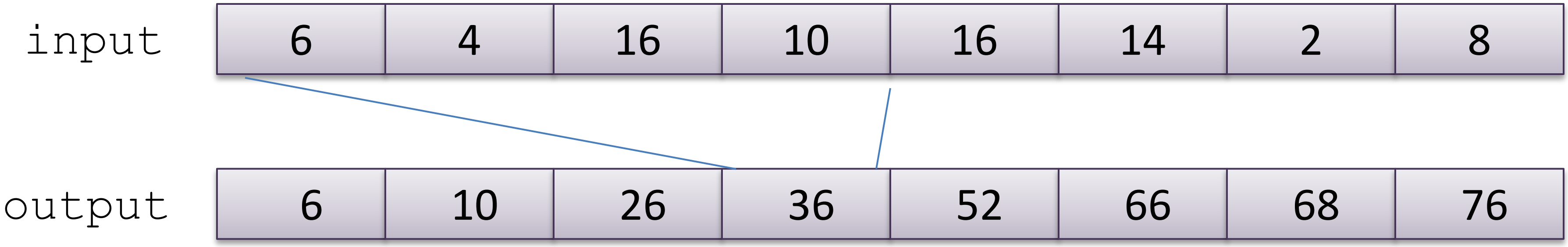
Andrew Appel

Princeton University

Credits:
Dan Grossman, U.Wash.
Guy Blelloch, Bob Harper (CMU), Dan Licata (Wesleyan)

The prefix-sum problem

prefix_sum : int seq -> int seq



`input`	6	4	16	10	16	14	2	8
`output`	6	10	26	36	52	66	68	76

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)

Parallel prefix-sum

The trick: *Use two passes*

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

First pass *builds a tree of sums bottom-up*

- the “up” pass

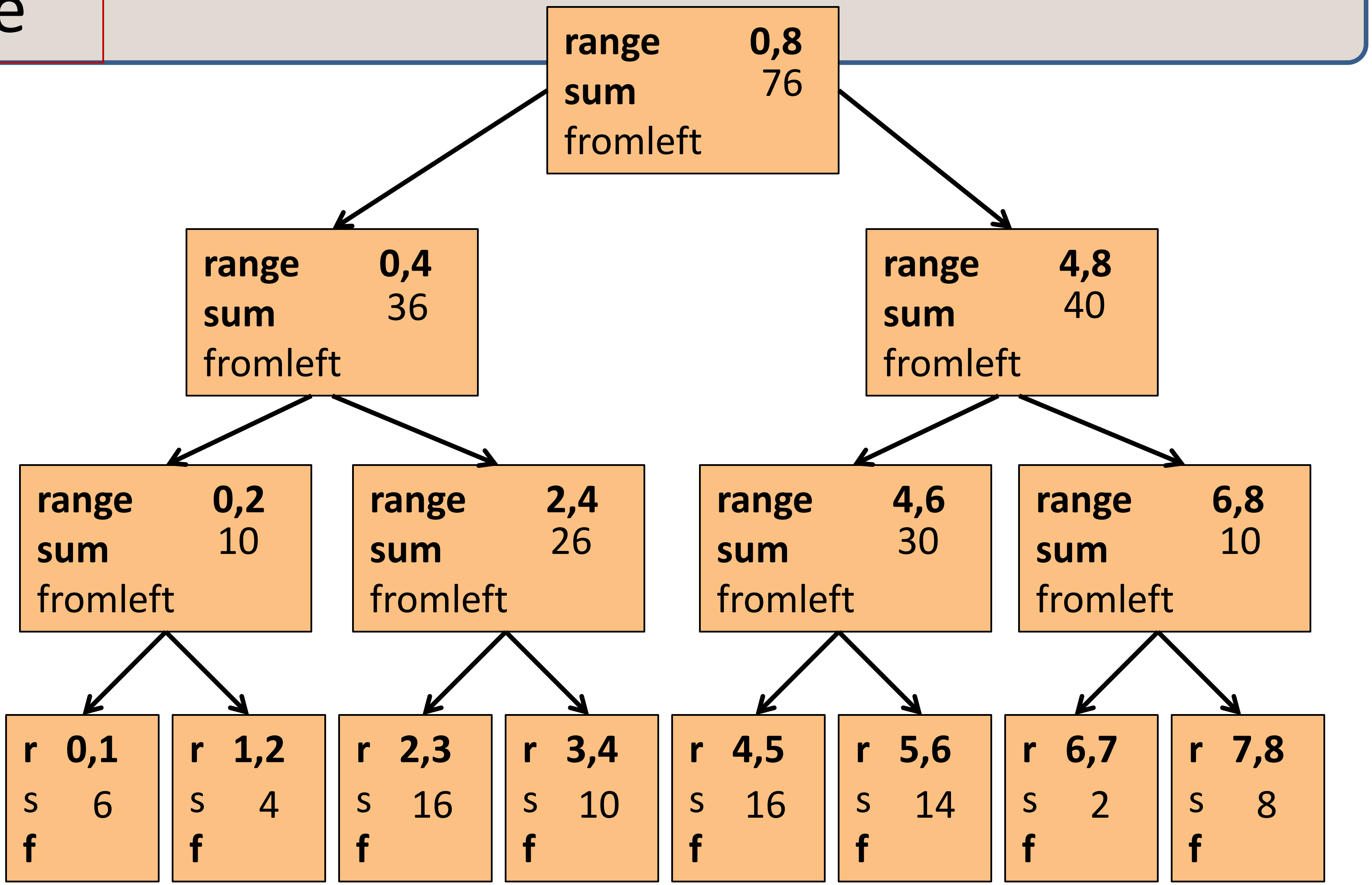
Second pass *traverses the tree top-down to compute prefixes*

- the “down” pass computes the "from-left-of-me" sum

Historical note:

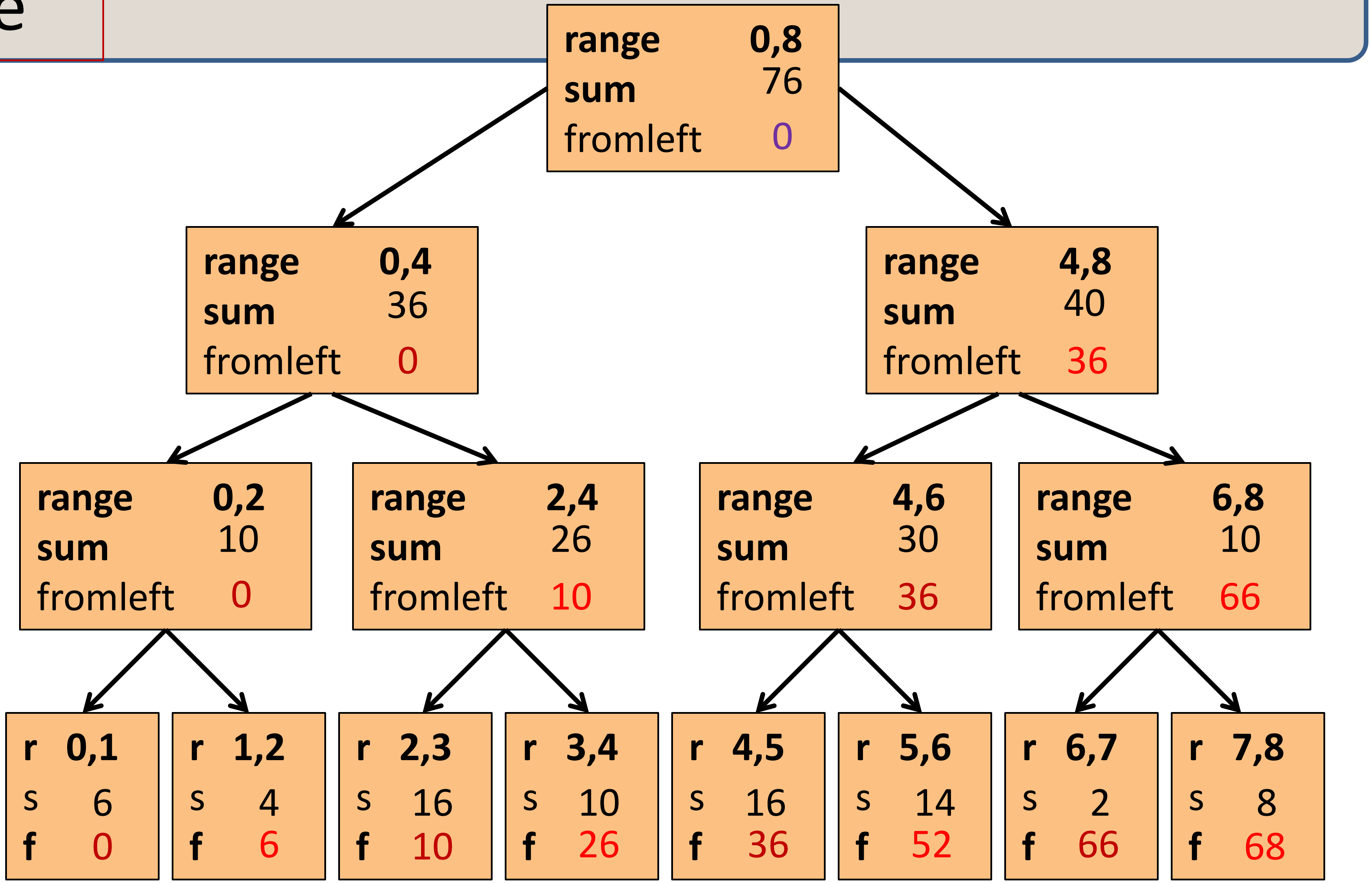
- Original algorithm due to R. Ladner and M. Fischer, 1977

Example



`input`	6	4	16	10	16	14	2	8
`output`								

Example



range 0,8
sum 76
fromleft 0

range 0,4
sum 36
fromleft 0

range 4,8
sum 40
fromleft 36

range 0,2
sum 10
fromleft 0

range 2,4
sum 26
fromleft 10

range 4,6
sum 30
fromleft 36

range 6,8
sum 10
fromleft 66

r 0,1
s 6
f 0

r 1,2
s 4
f 6

r 2,3
s 16
f 10

r 3,4
s 10
f 26

r 4,5
s 16
f 36

r 5,6
s 14
f 52

r 6,7
s 2
f 66

r 7,8
s 8
f 68

input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	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,middle**)
     - Right child has sum of [**middle,hi**)
     - A leaf has sum of [**i,i+1**), i.e., **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

The algorithm, pass 2

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`**,
     - **`nth output i == 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 **`output`**
- Invariant: **`fromLeft`** is sum of elements left of the node's range

Analysis: $O(n)$ work, $O(\mathtt{log}\ n)$ 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

scan (o) $<x_1, \ldots, x_n>$

==

$<x_1, x_1 \text{ o } x_2, \ldots, x_1 \text{ o } \ldots \text{ o } x_n>$

Operator o
must be associative!

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

pre_scan (o) base $<x_1, \ldots, x_n>$

==

$<\text{base}, \text{base o } x_1, \ldots, \text{base o } x_1 \text{ o } \ldots \text{ o } x_{n-1}>$

base must be a unit
for operator o

sequence with o applied to all items
to the left of index in input

Parallel Filter

Given a sequence **`input`**, produce a sequence **`output`** containing only elements v such that (**`f v)`** is **`true`**

Example: let f x = x > 10

```
    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*

Parallel prefix to the rescue

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:	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:	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)

Easy: Do the two recursive calls in parallel

- Work: unchanged. Total: $O(n \log n)$
- Span: now $T(n) = O(n) + 1T(n/2) = O(n)$

Doing better

As with mergesort, we get a $O(\mathtt{log}\ n)$ 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…

*These days, most hits get this right, and discuss parallel 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, $O(\texttt{log}\ n)$ 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 $O(\texttt{log}\ n)$ span for partition, the total best-case and expected-case span for quicksort is

$$T(n) = O(\texttt{log}\ n) + 1T(n/2) = O(\texttt{log}^2\ n)$$

Example

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"

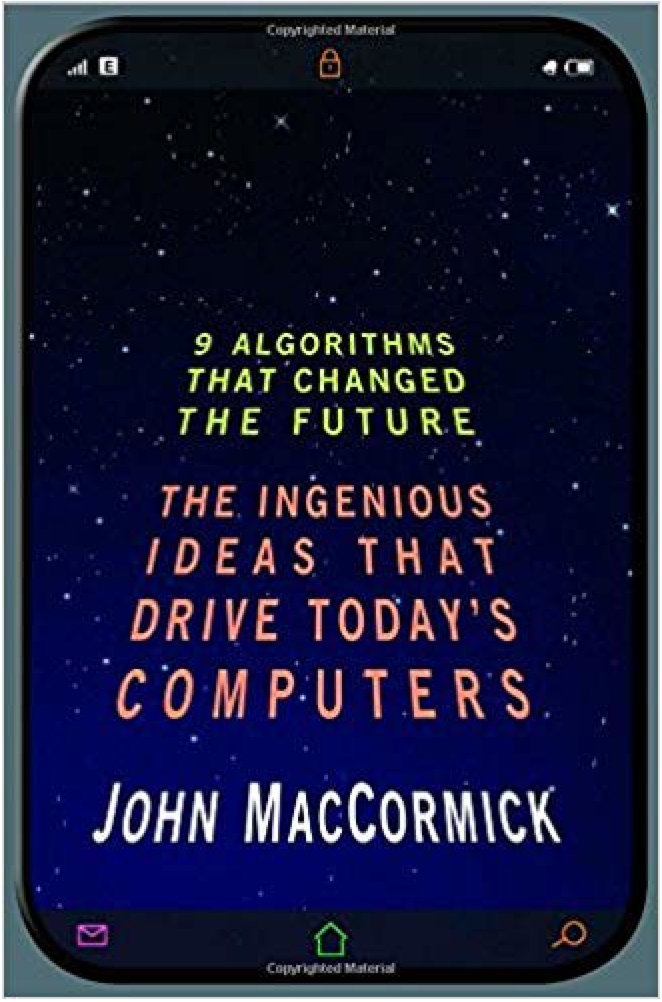
Summary

- 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

ASSIGNMENT #7: PROGRAMMING WITH PARALLEL SEQUENCES

Do the reading . . .


Copyrighted Material
9 ALGORITHMS THAT CHANGED THE FUTURE
THE INGENIOUS IDEAS THAT DRIVE TODAY'S COMPUTERS
JOHN MACCORMICK
Copyrighted Material

Chapter 2, “Search Engine Indexing”

(On reserve for this course, available at canvas.princeton.edu,

select this course, then “reserves”)

(Read also Chapter 3, “Page Rank” so you can appreciate what you were doing in Assignment 5 . . .)

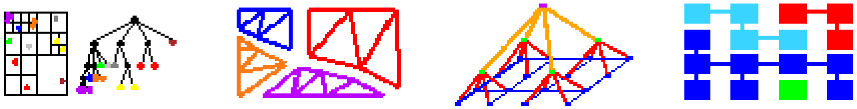
map-reduce API for Assignment 7

		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

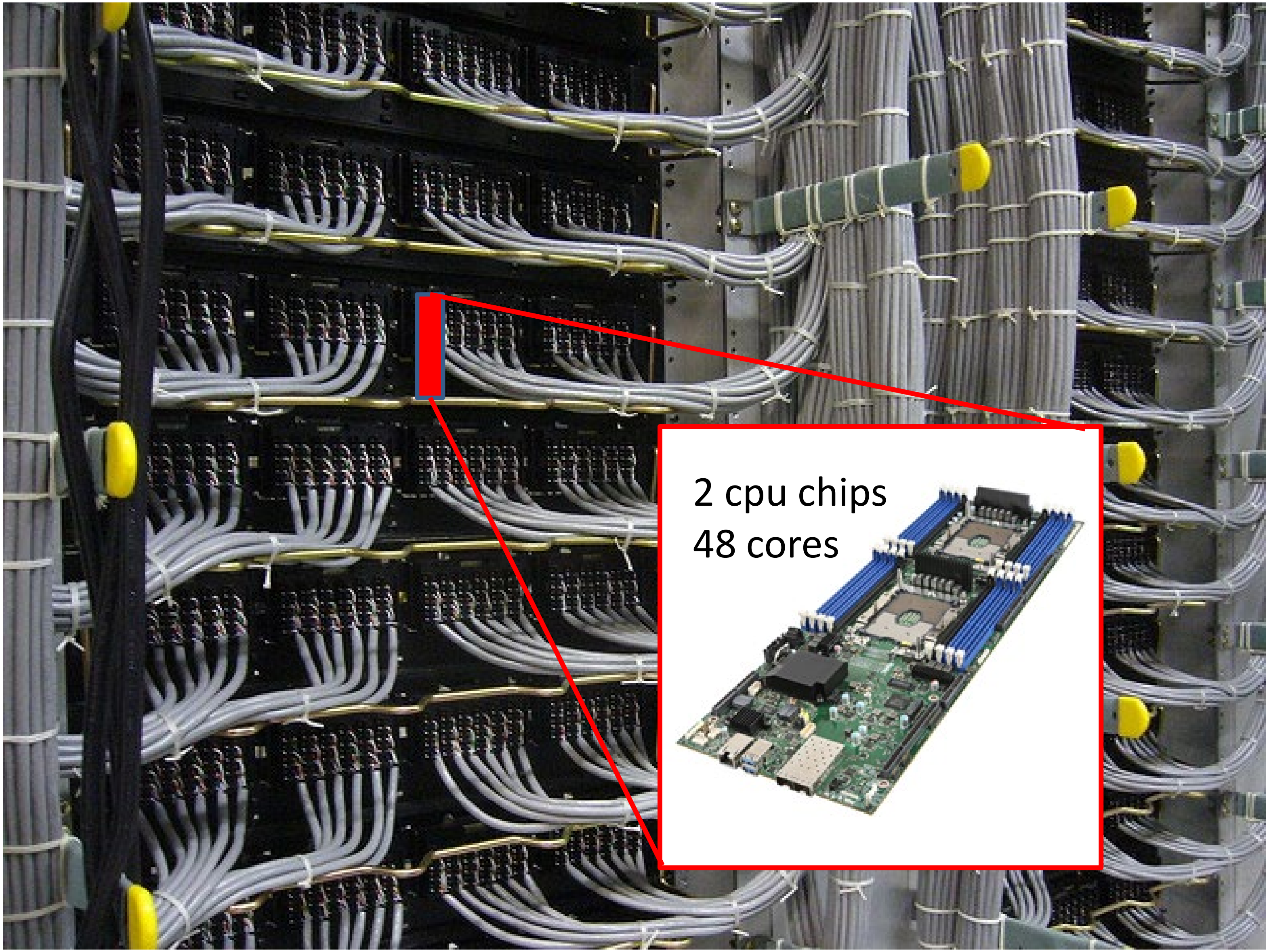


NESL is a parallel language developed at Carnegie Mellon. It integrates ideas from the theory community (parallel algorithms), the languages community (functional languages) and the systems community (many of the implementation techniques). The most important new ideas behind NESL are

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

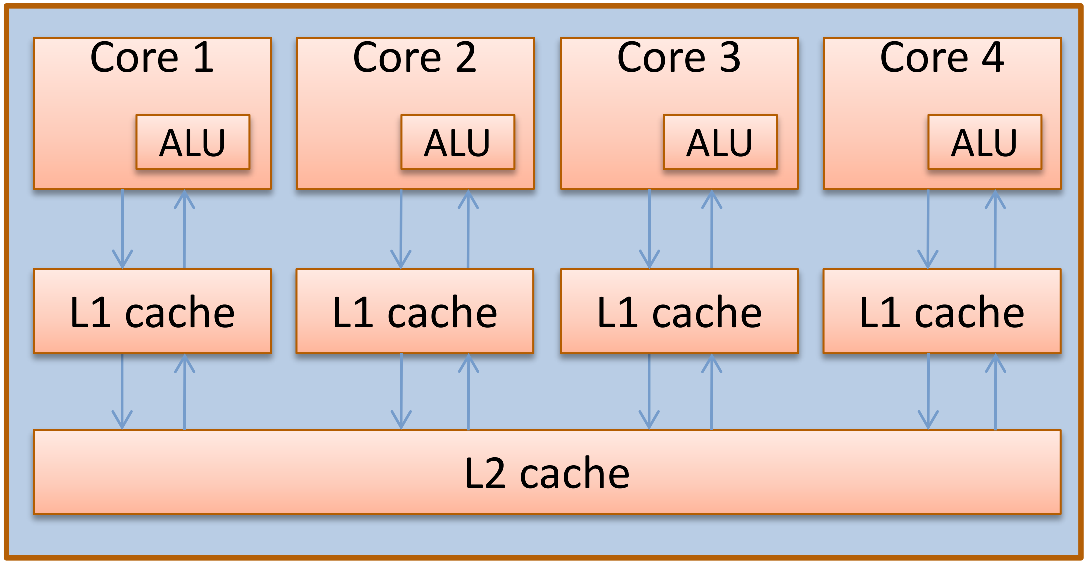
Data Centers: *Lots* of Connected Computers!



2 cpu chips
48 cores

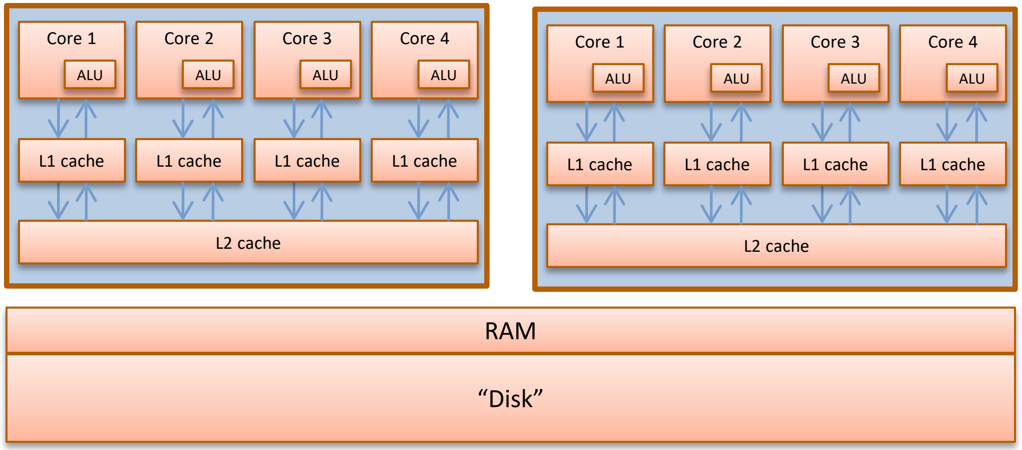
Real Machines

Chip



Real Machines

Board


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

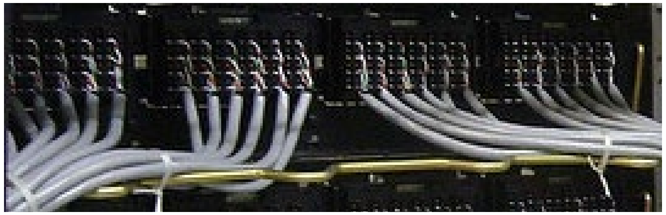
Core 1 ALU L1 cache
Core 2 ALU L1 cache
Core 3 ALU L1 cache
Core 4 ALU L1 cache
L2 cache

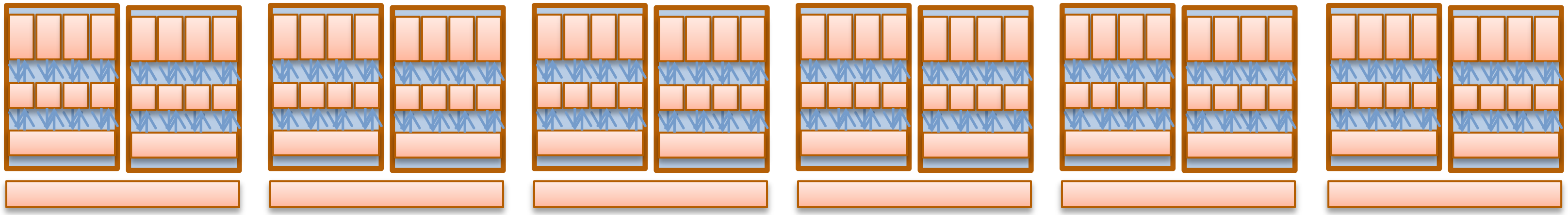
RAM

“Disk”

Real Machines

Shelf

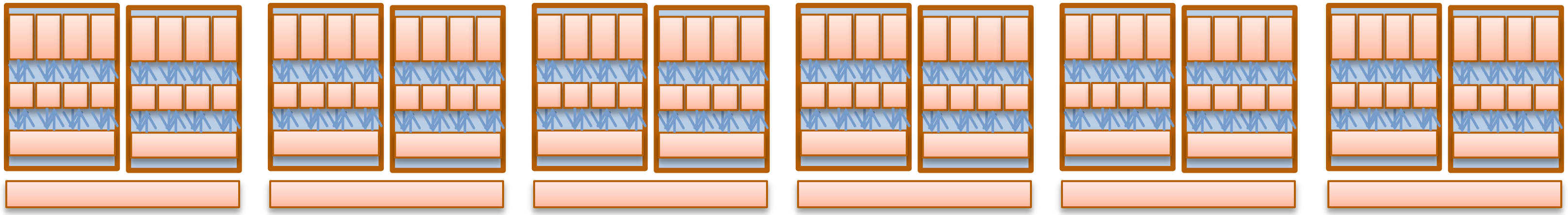




Rack

Server room

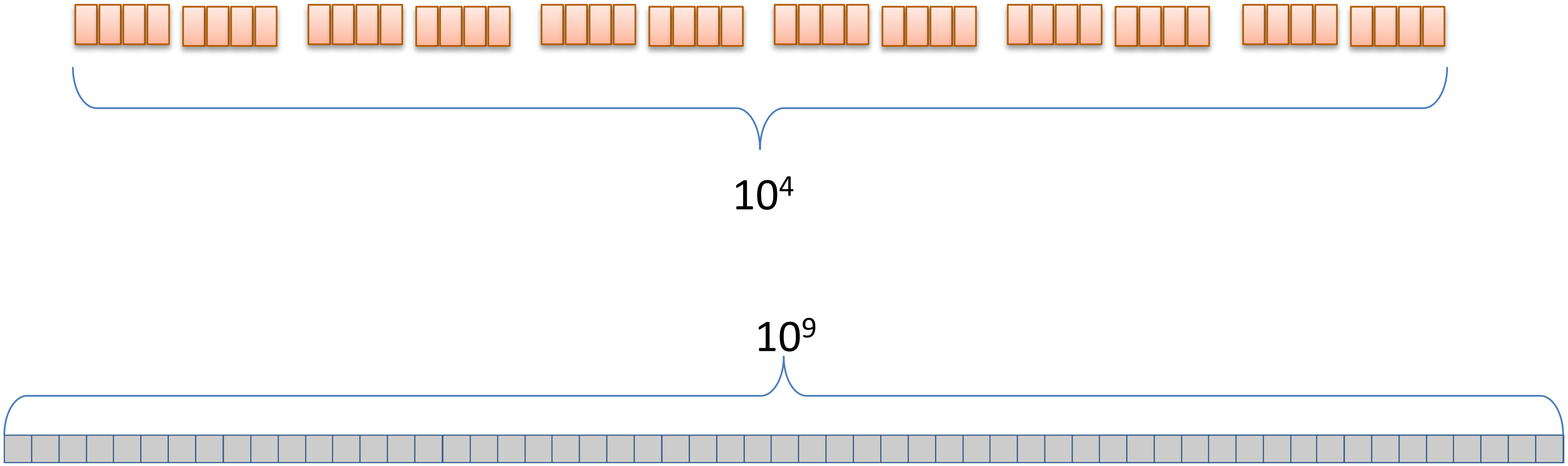
Real Machines



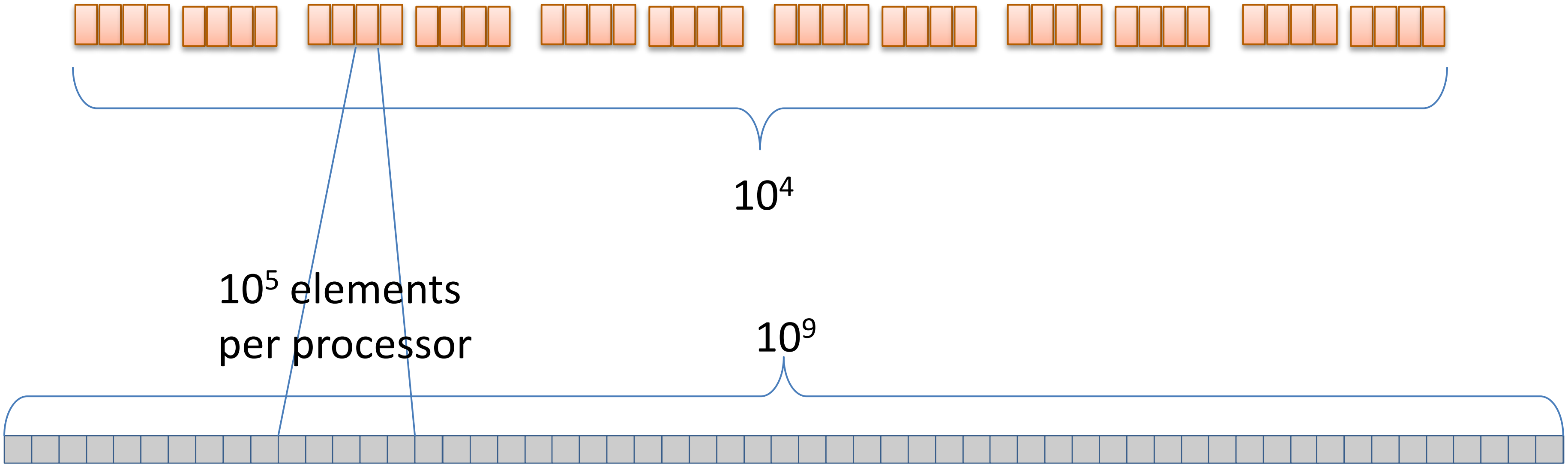
s: int seq

length(s) = 10^9

Real Machines



Real Machines

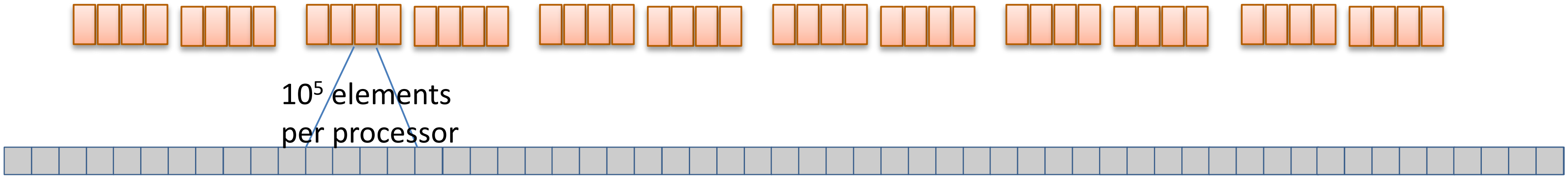


10^4

10^5 elements
per processor

10^9

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]
```

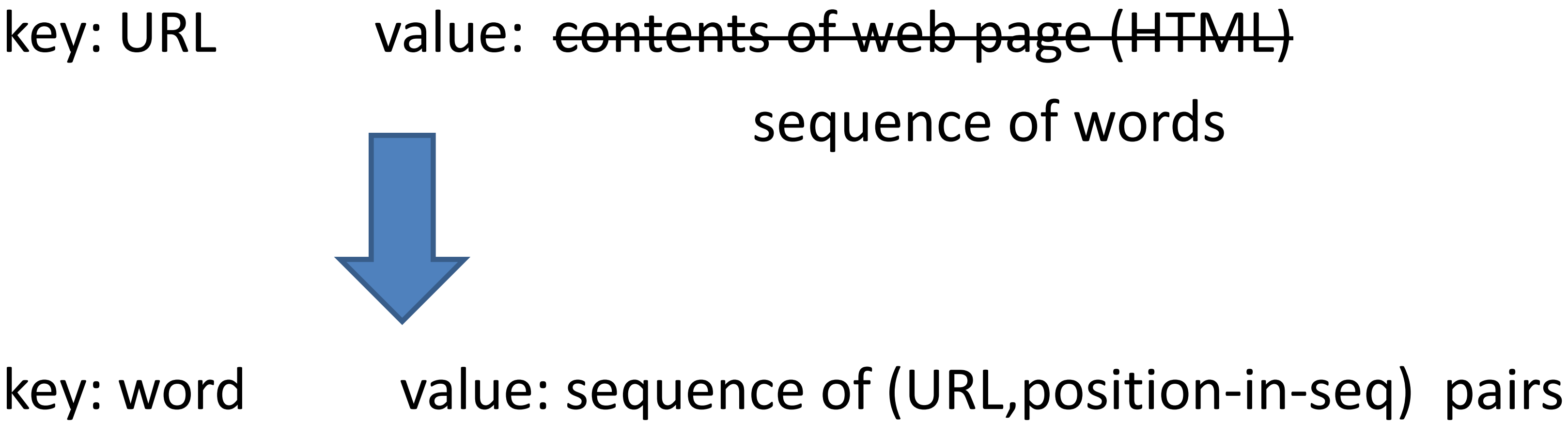
```
let s1 = M.seq_of_array [|1;2;3;4;5|]
let f (s: int M.seq) = M.map (fun i -> i+1) s
let s2 =  ~~A.reporting  "test1"~~  f  s1
let r = Array.to_list (M.array_of_seq s2)

(* Prints:  nothing *)

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

Discussion

How to use these operators to make an inverted index?

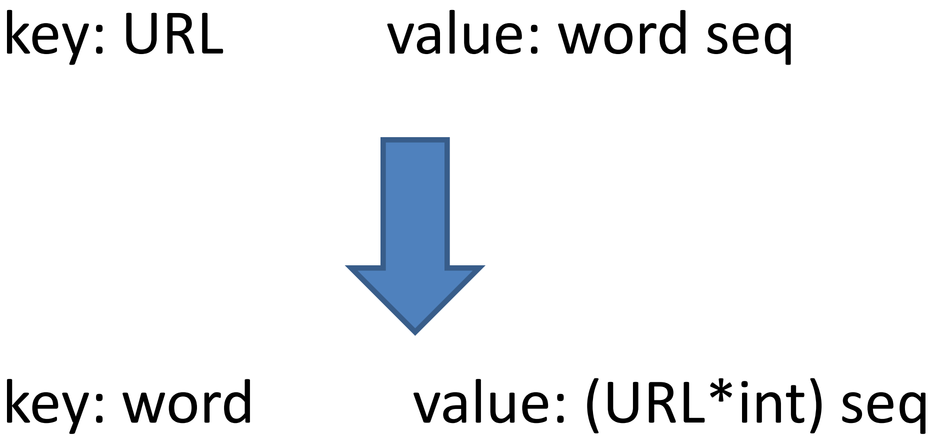


key: URL value: ~~contents of web page (HTML)~~ sequence of words

key: word value: sequence of (URL,position-in-seq) pairs

Discussion

How to use these operators to make an inverted index?

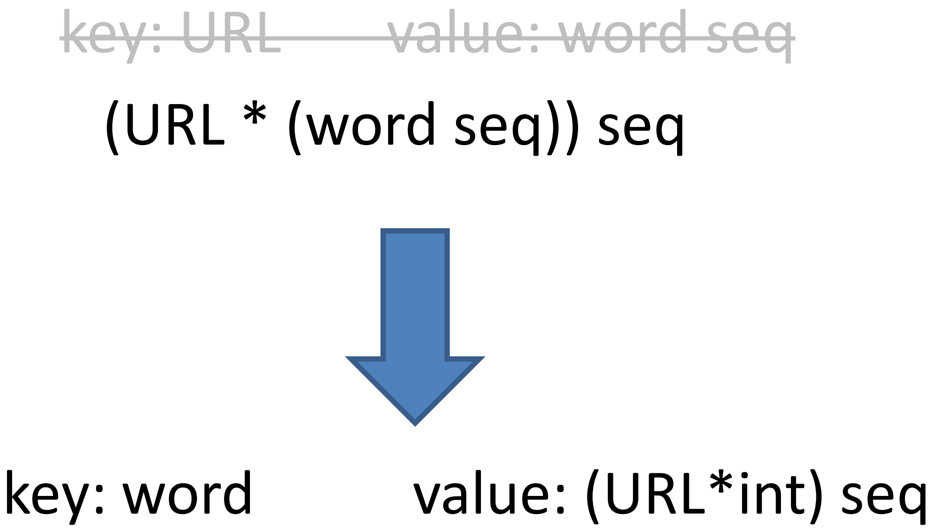


key: URL value: word seq

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

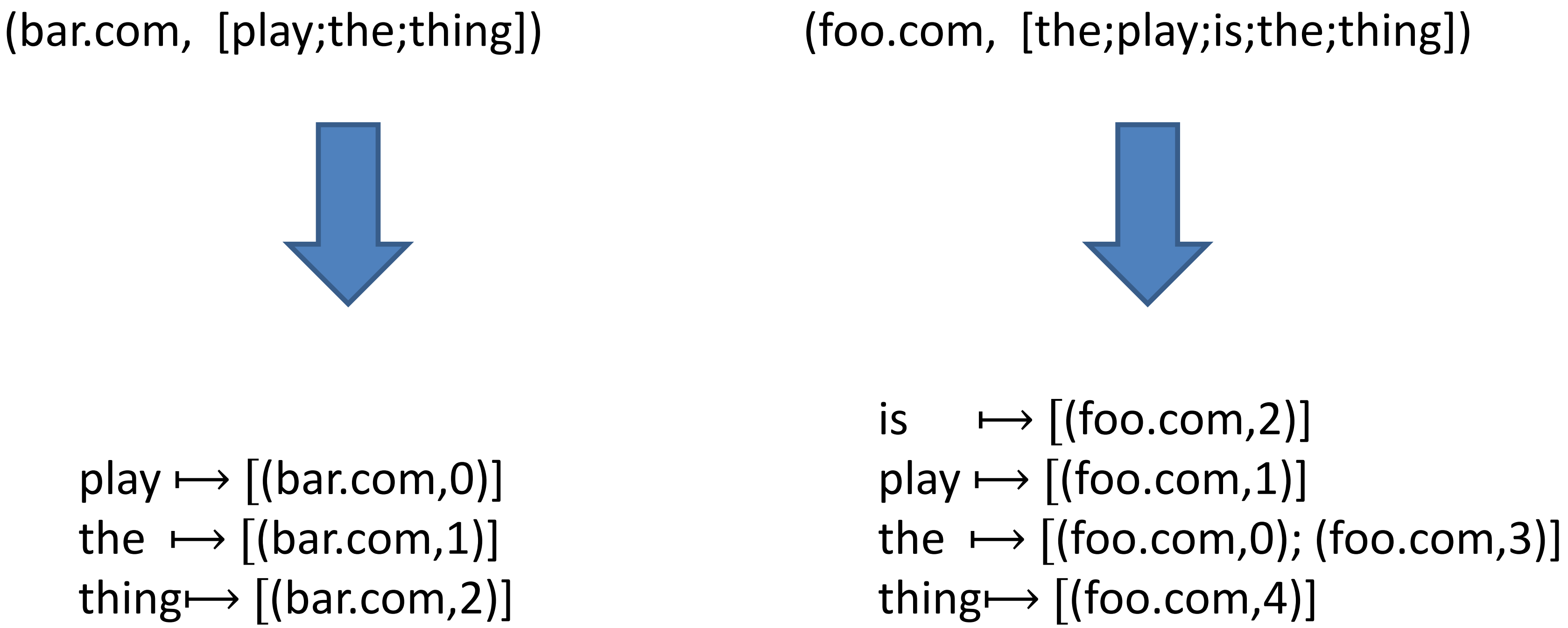
(URL* (word seq))

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

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

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

Discussion



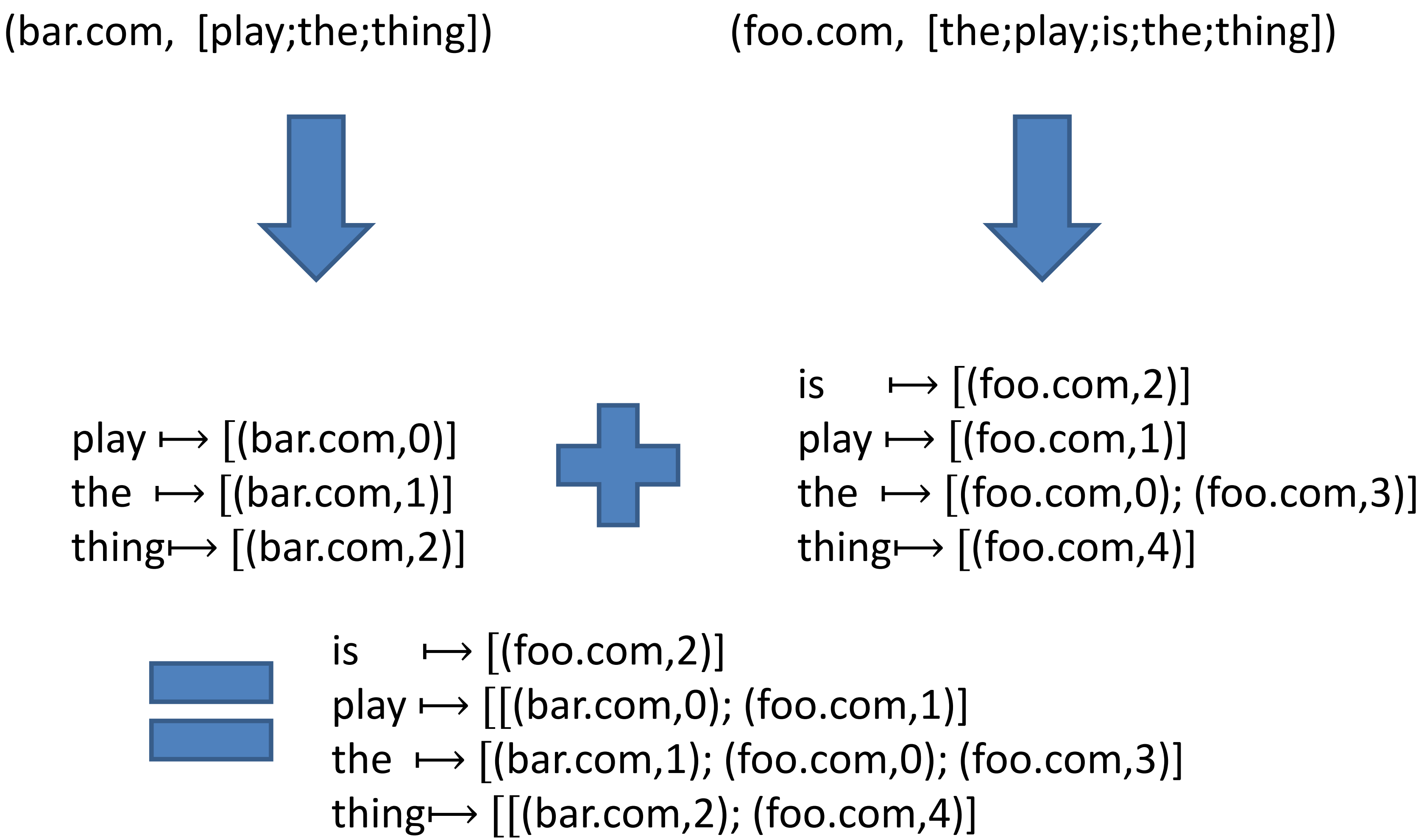
(bar.com, [play;the;thing])

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)]

Discussion



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

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

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

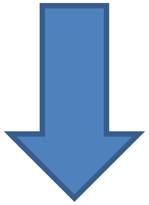
is ⟼ [(foo.com,2)]
play ⟼ [(foo.com,1)]
the ⟼ [(foo.com,0); (foo.com,3)]
thing⟼ [(foo.com,4)]

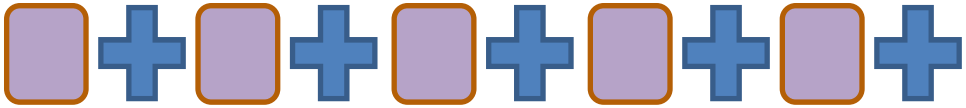
is ⟼ [(foo.com,2)]
play ⟼ [[(bar.com,0); (foo.com,1)]
the ⟼ [(bar.com,1); (foo.com,0); (foo.com,3)]
thing⟼ [[(bar.com,2); (foo.com,4)]

Discussion

How to use these operators to make an inverted index?

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





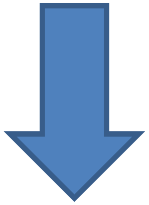
Reduce!

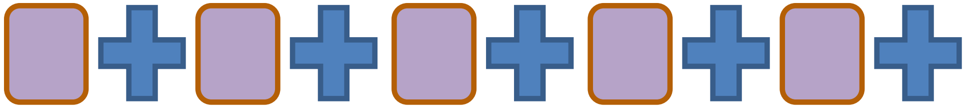
word ((URL*int)seq) Map.t

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.