

# 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  $O(\log n)$  span
- So in total there is  $O(n)$  work and  $O(\log n)$  span

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

- the “up” pass

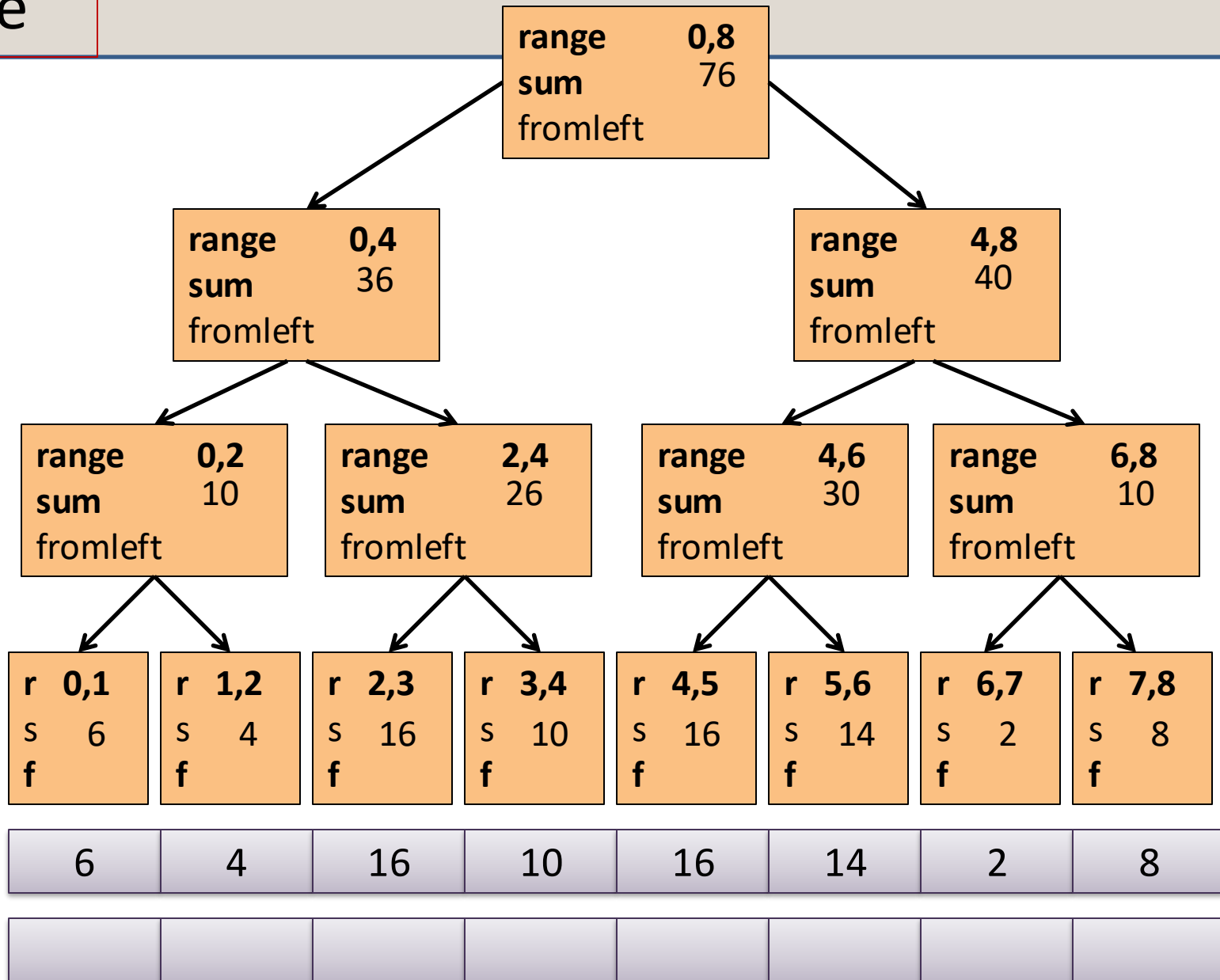
Second pass *traverses the tree top-down to compute prefixes*

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

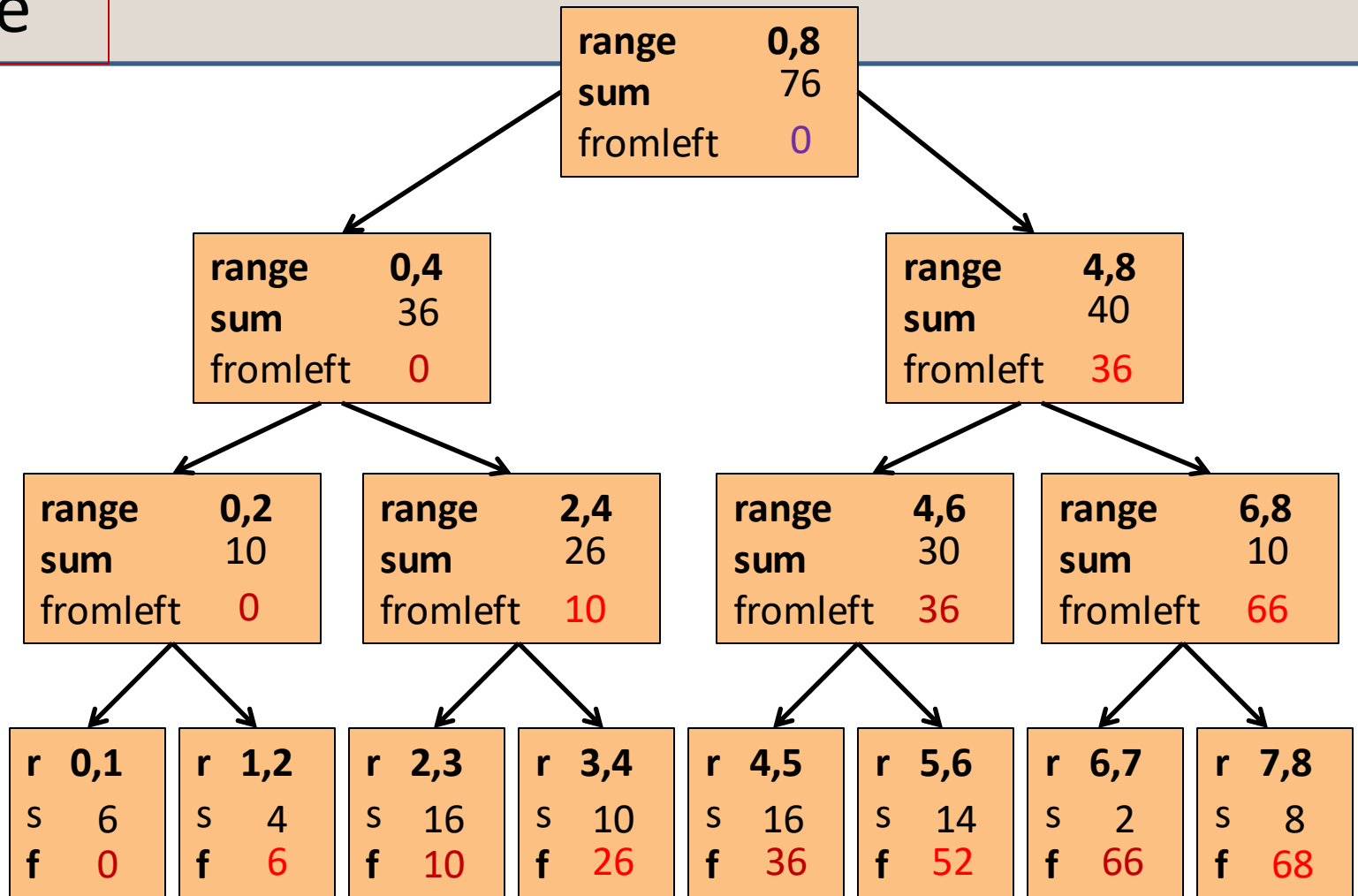
Historical note:

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

# Example



# Example



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,  $O(\log n)$  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

$$\begin{aligned} & \text{scan } (o) \langle x_0, \dots, x_{n-1} \rangle \\ == \\ & \langle x_0, x_0 \circ x_1, \dots, x_0 \circ \dots \circ x_{n-1} \rangle \end{aligned}$$

Operator  $o$   
must be associative!

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

$$\begin{aligned} & \text{pre\_scan } (o) \text{ base } \langle x_0, \dots, x_{n-1} \rangle \\ == \\ & \langle \text{base}, \text{base } \circ x_0, \dots, \text{base } \circ x_0 \circ \dots \circ x_{n-2} \rangle \end{aligned}$$

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  $\text{bits}[i] == 1$  then write  $\text{input}[i]$  to  $\text{output}[\text{bitsum}[i]]$  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 <i>work</i> |
|--|----------------------------------|
| 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 <i>work</i>
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  $O(\log n)$  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,  $O(\log n)$  span
- Parallel filter elements less than pivot into left side of **aux** array
- Parallel filter elements greater than pivot into right side 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

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

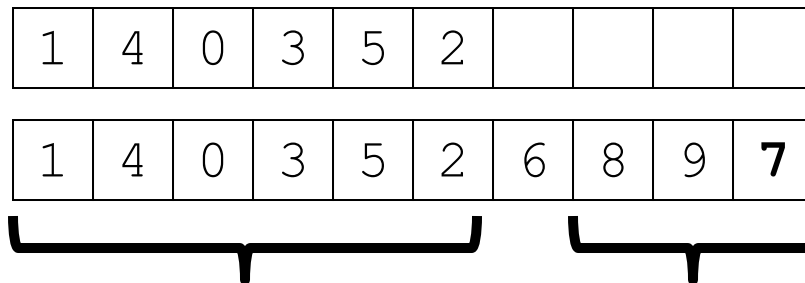
# Example

Step 1: pick pivot as median of three

<b>8</b>	1	4	9	<b>0</b>	3	5	2	7	<b>6</b>
----------	---	---	---	----------	---	---	---	---	----------

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	<b>7</b>



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

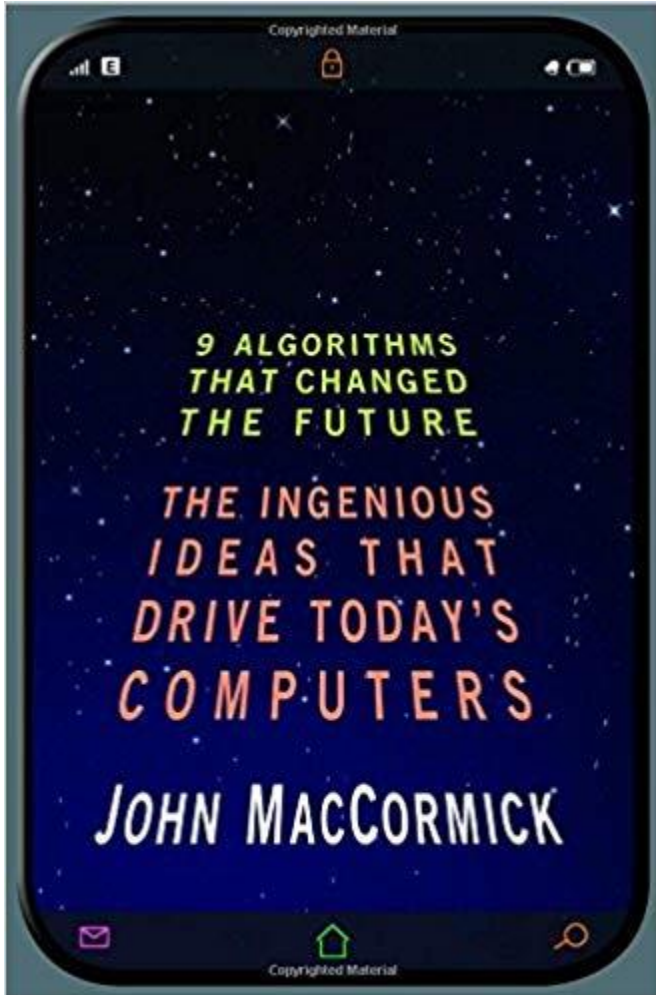
# **ASSIGNMENT #7: PROGRAMMING WITH PARALLEL SEQUENCES**

# Do the reading . . .

Chapter 2, “Search Engine Indexing”

(On reserve for this course, available at [canvas.princeton.edu](https://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

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



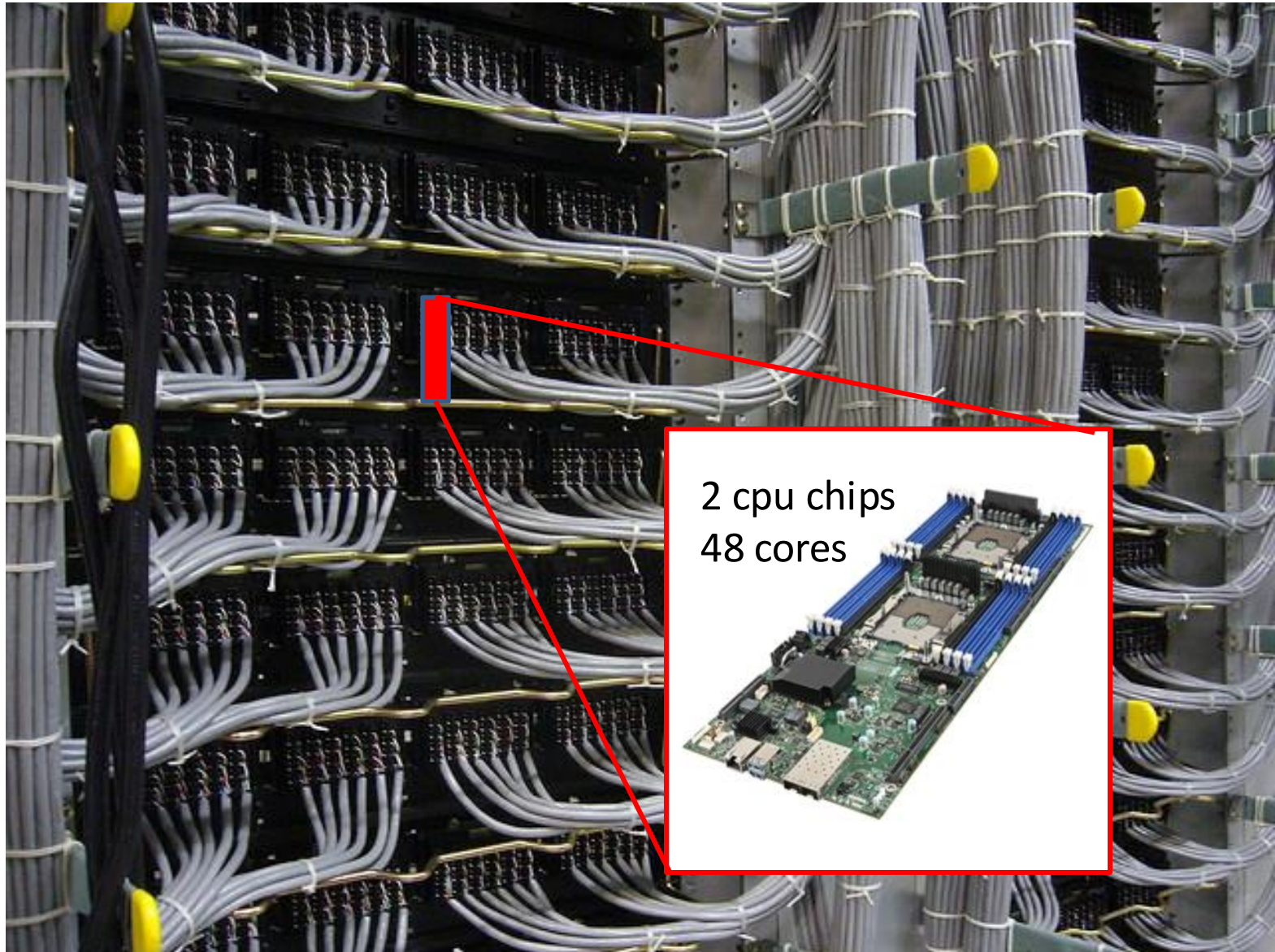
NESL is a parallel language developed at [Carnegie Mellon](#). It integrates ideas from the theory community (parallel algorithms), the languages community (functional languages) and the systems community (many of the implementation techniques). The most important new ideas behind NESL are

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.

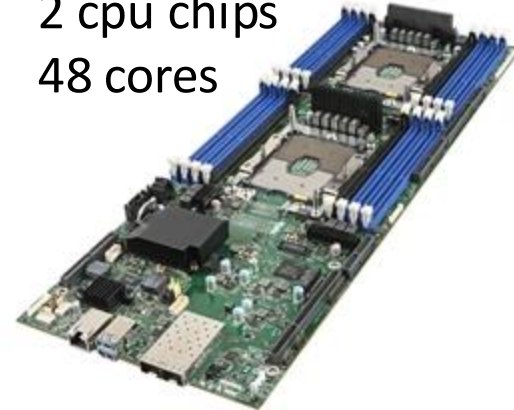


# **IMPLEMENTATION OF PARALLEL SEQUENCES**

# Data Centers: *Lots* of Connected Computers!

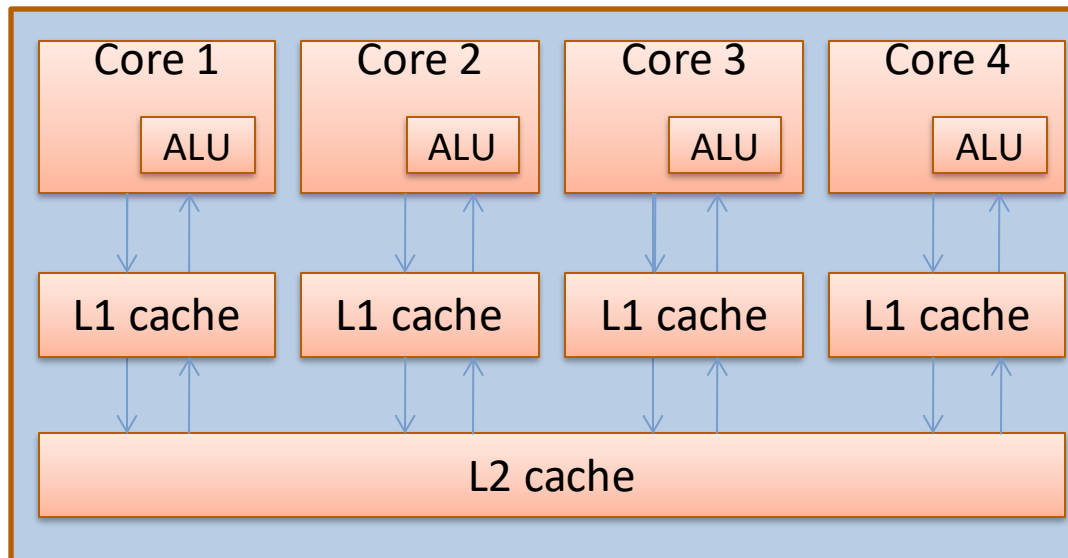


2 cpu chips  
48 cores



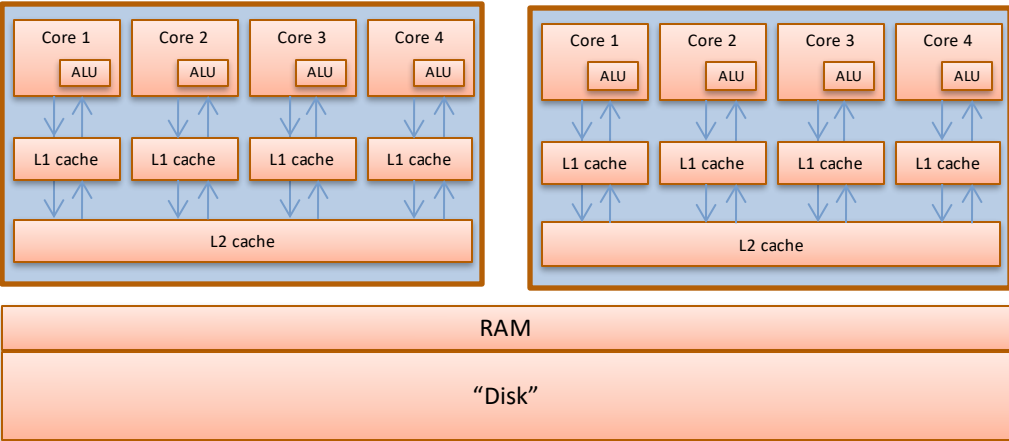
# Real Machines

Chip



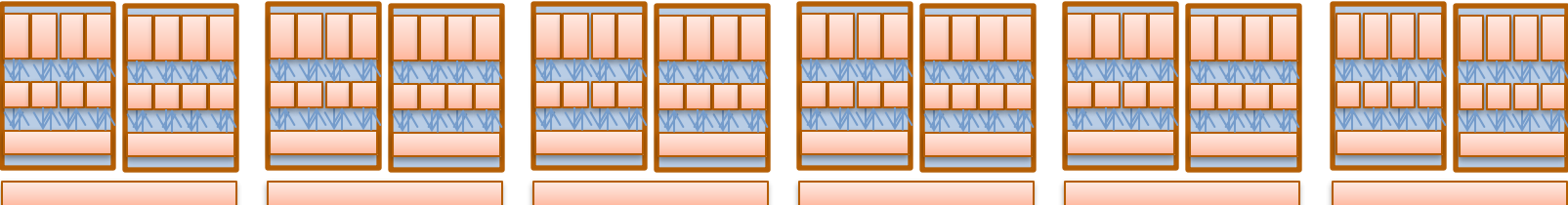
# Real Machines

## Board



# Real Machines

Shelf



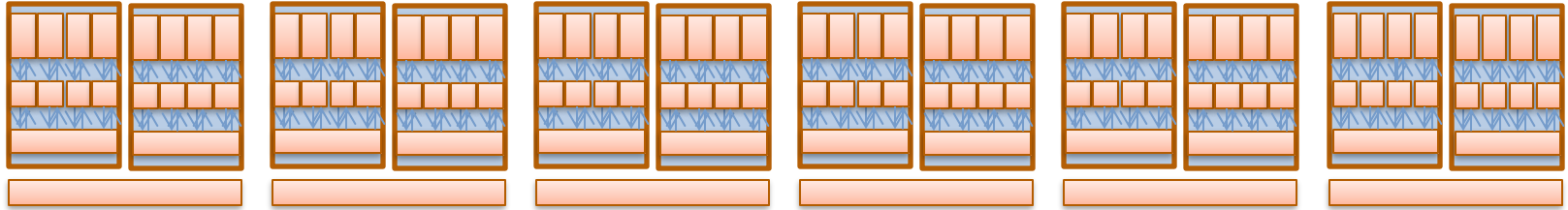
Rack



Server room



# Real Machines



```
s: int seq  
length(s) = 109
```

# Real Machines



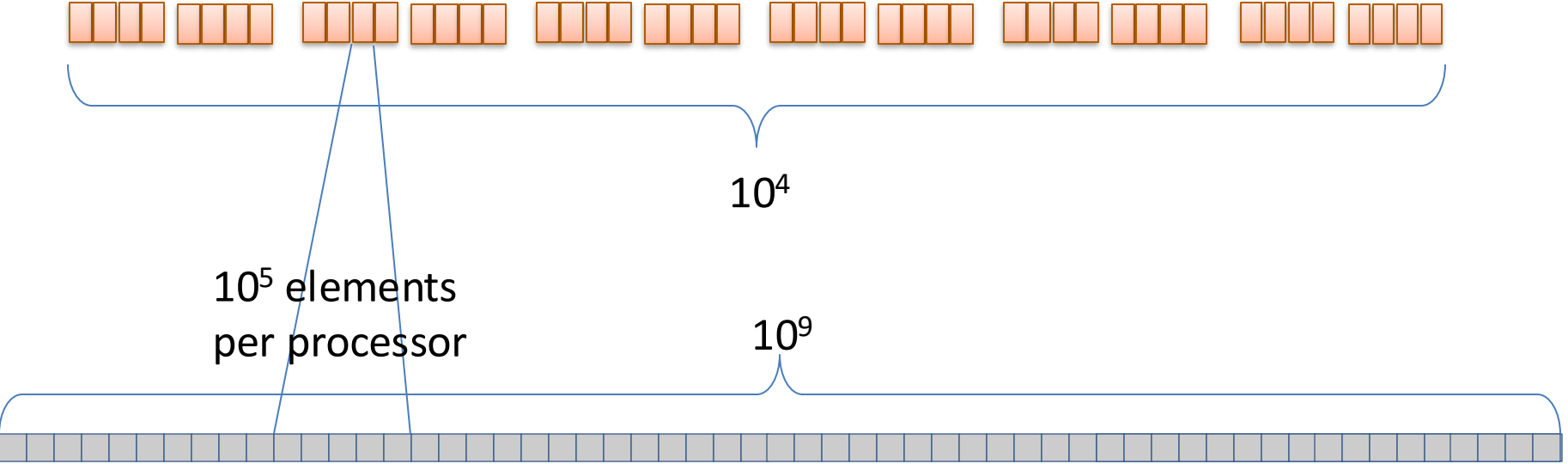
$10^4$

$10^9$



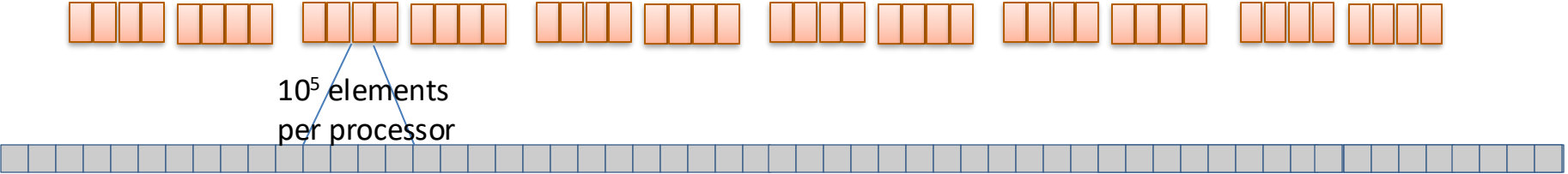


# Real Machines





# Real Machines



		Work	Span
<code>tabulate (f: int-&gt;<math>\alpha</math>) (n: int) : <math>\alpha</math> seq</code>	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 -> 'a t
  val nth : 'a t -> int -> 'a
  val map : ('a -> 'b) -> 'a t -> 'b t
  val map_reduce : ('a -> 'b -> 'c) -> 'a t -> 'c t
  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
  val scan: ('a -> 'a -> 'a) -> 'a t -> '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
  ...
end
```

# Work/Span estimation

```
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) -> 'a t -> 'b t
  val reduce : ('a -> '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 -> '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  $\mapsto$  [(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  $\mapsto$  [(bar.com,0)]  
the  $\mapsto$  [(bar.com,1)]  
thing  $\mapsto$  [(bar.com,2)]

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



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])

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



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



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



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

# Discussion

How to use these operators to make an inverted index?

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



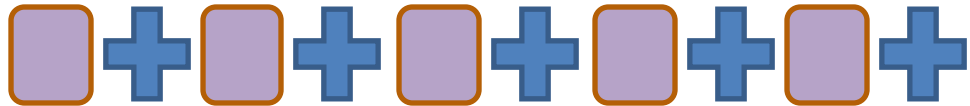
Reduce!

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
```

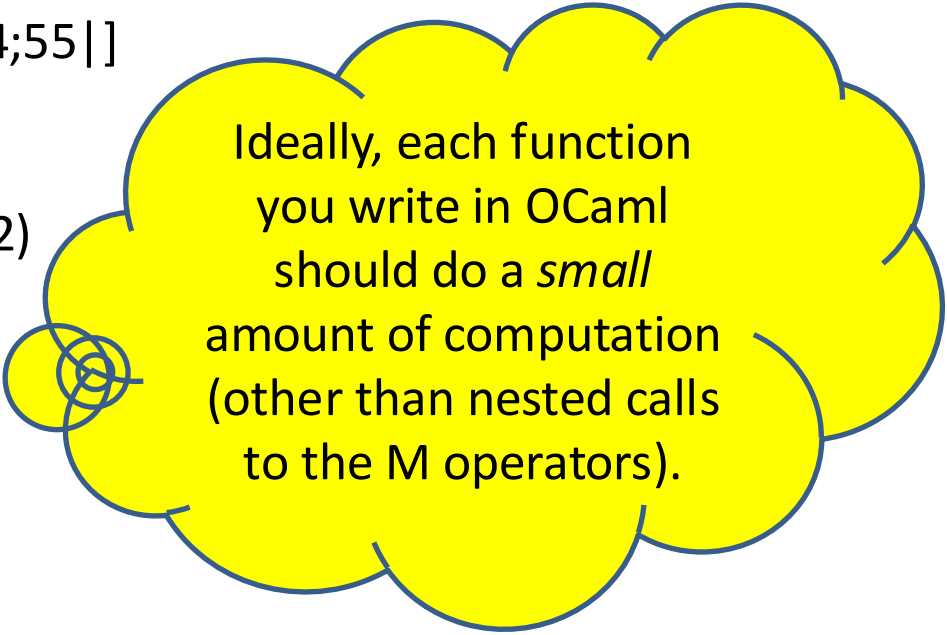
```
let s2 = A.reporting "test2" f s1
```

```
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).

# Parallel Benchmarking

The main.ml driver we provided for [testing](#) also provides support for benchmarking your code locally. If you pass the flag `-p`, your code will be run using a shared-memory parallel implementation of the Sequence module (see `SharedMemParallelS` in `sequence.ml`).

```
$ dune exec main -- -p make_index time=0.379550
```

In case you don't have a 16-core computer handy, you can benchmark your code on a computing cluster by including the string `"#benchmark"` somewhere in `inverted_index.ml` that you upload to Tigerfile. Then, after a few minutes, you'll get an e-mail with speedup results – please paste that into your `signature.txt` and discuss briefly.



# 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.