Parallel Functional Programming
Lecture 7
Data Parallelism I

Mary Sheeran

http://www.cse.chalmers.se/edu/course/pfp

Data parallelism

Introduce parallel data structures and make operations on them parallel

Often data parallel arrays

Canonical example : NESL (NESted-parallel Language) (Blelloch)

NESL

concise (good for specification, prototyping)

allows programming in familiar style (but still gives parallelism)

allows nested parallelism (see later)

associated language-based cost model

gave decent speedups on wide-vector parallel machines of the day

Hugely influential!

Image: © Thinking Machines Corporation, 1986.
Photo: Steve Grohe.

http://www.venturenavigator.co.uk/content/152

Connection Machine

First commercial massively parallel machine

65k processors

can see CM-1 and CM-5 (from 1993) at Computer History Museum, Mountain View

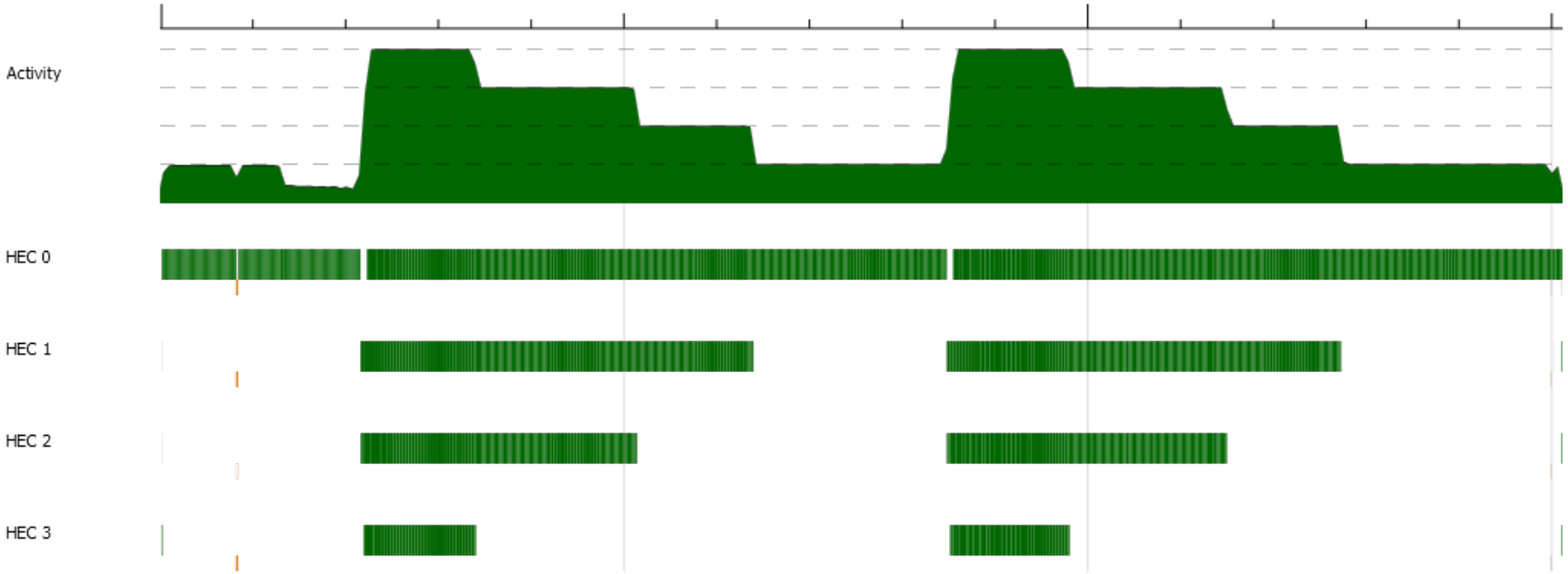
NESL array operations

```
function factorial(n) =
  if (n <= 1) then 1
  else n*factorial(n-1);

{factorial(i) : i in [3, 1, 7]};
```

apply to each = parallel map (works with user-defined functions => load balancing)

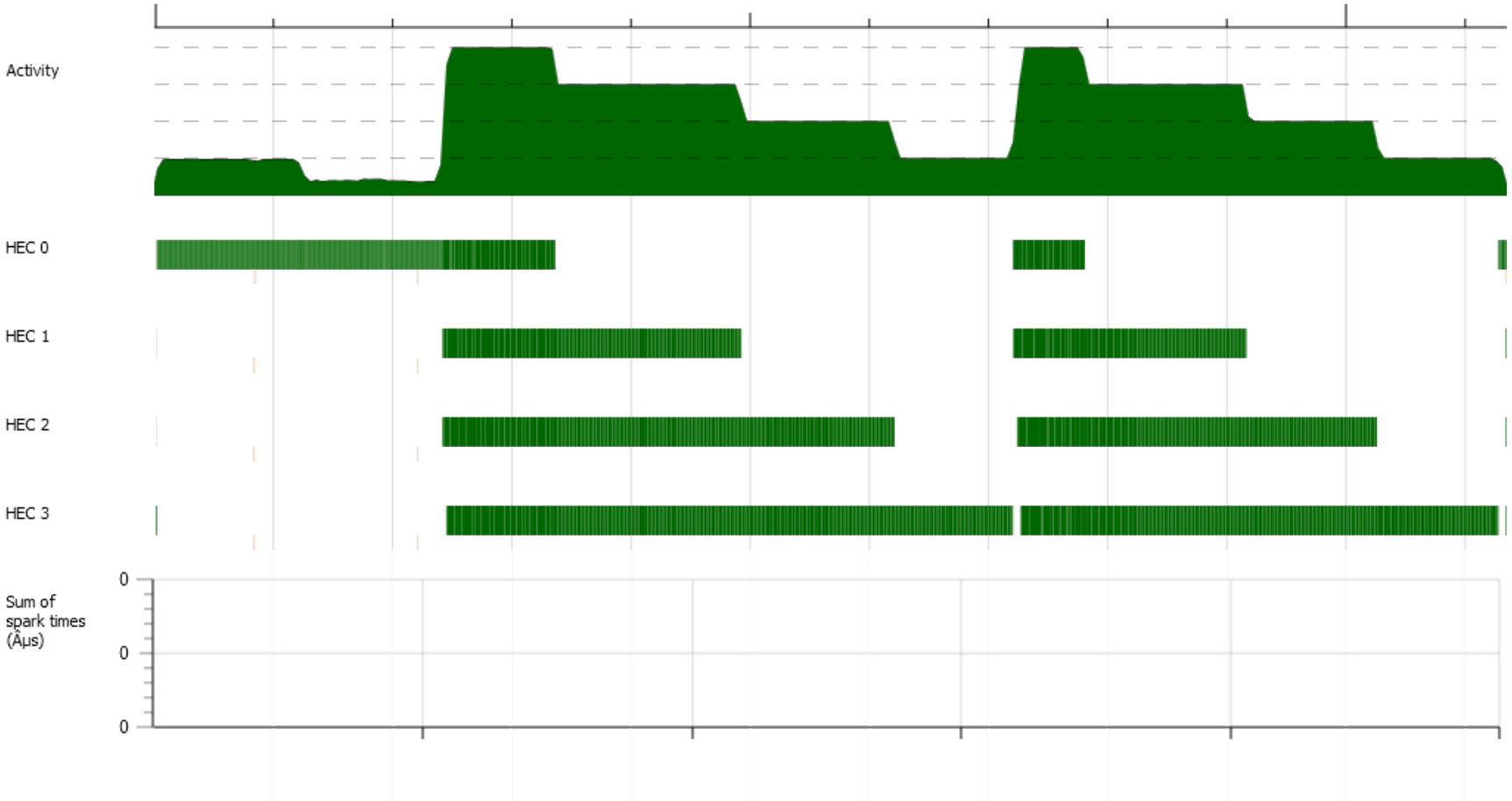
list comprehension style notation

map fac $ fromList (Z :. (n::Int)) (reverse [1..n]) (twice)



in Repa (parallel array library in Haskell, flat, more later)

reverse the list



apply to each (multiple sequencs)

The result of:

```
{a + b : a in [3, -4, -9]; b in [1, 2, 3]};
```

is:

```
it = [4, -2, -6] : [int]

Bye.
```

http://www.cs.cmu.edu/~scandal/nesl/tutorial2.html interactive tutorial!

Filtering too

The result of:

```
{a * a : a in [3, -4, -9, 5] | a > 0};
```

is:

```
it = [9, 25] : [int]

Bye
```

scan (Haskell first)

```
*Main> scanl1 (+) [1..10]
[1,3,6,10,15,21,28,36,45,55]

*Main> scanl1 (*) [1..10]
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

1 2 3 4 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

3

1 2 3 4 5 6 7 8 9 10

3

6

1 2 3 4 5 6 7 8 9 10

3

6

10

1 2 3 4 5 6 7 8 9 10

3

6

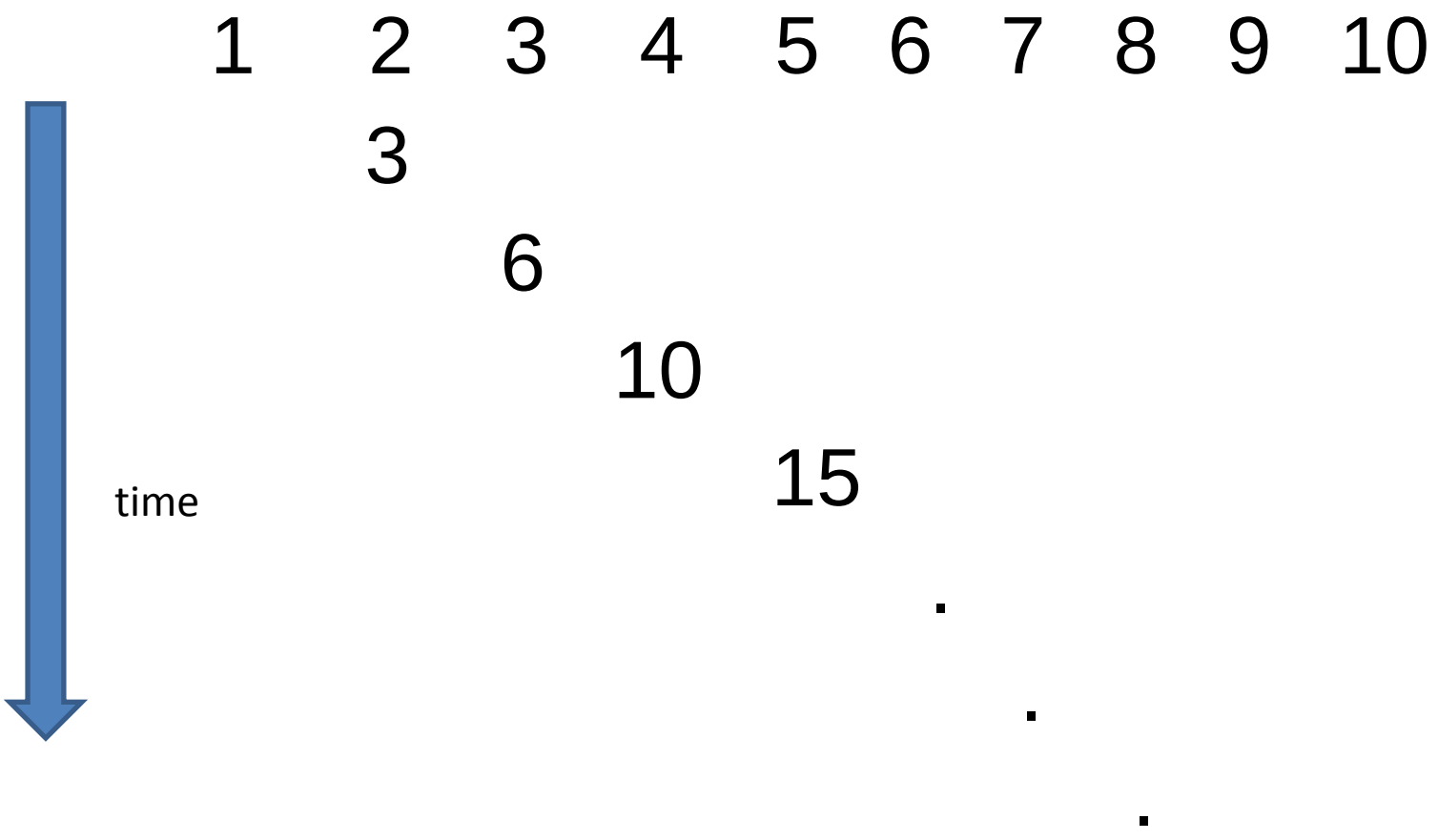
10

15

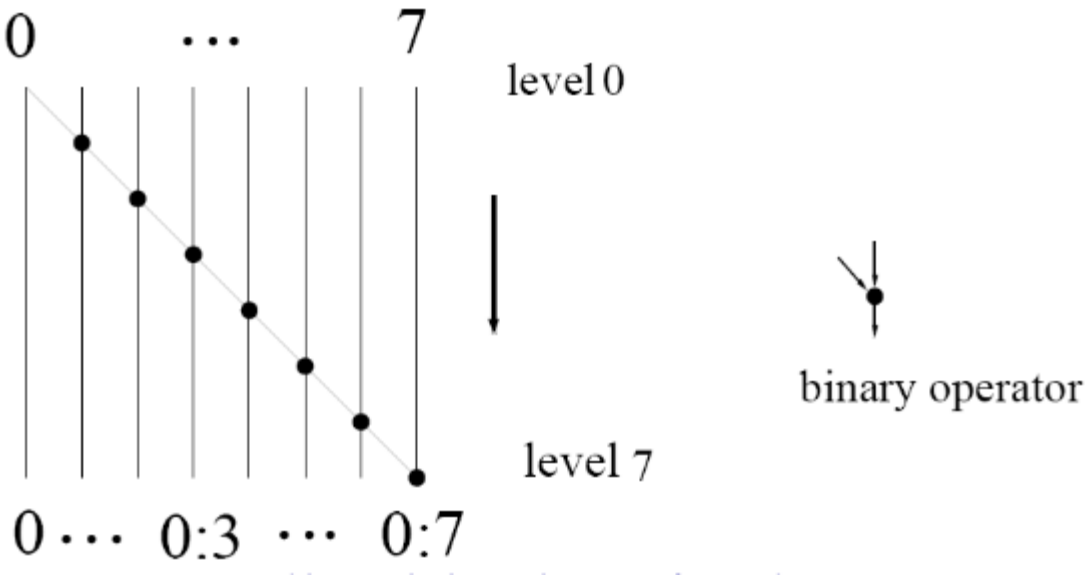
.

.

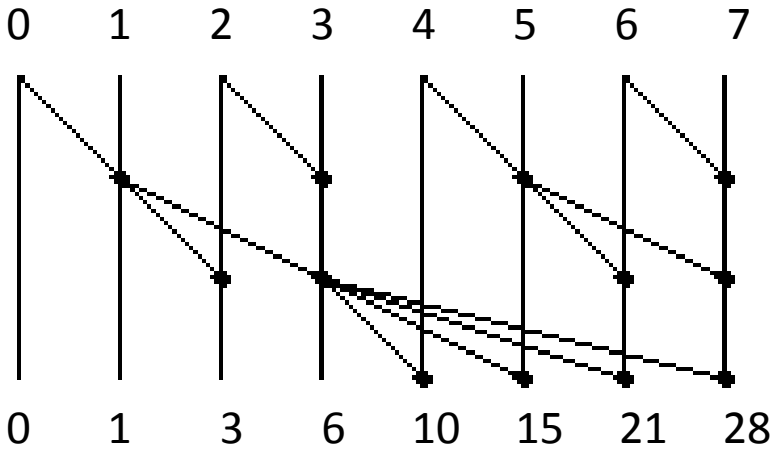
.


1 2 3 4 5 6 7 8 9 10
3
6
10
15
time

scan diagram

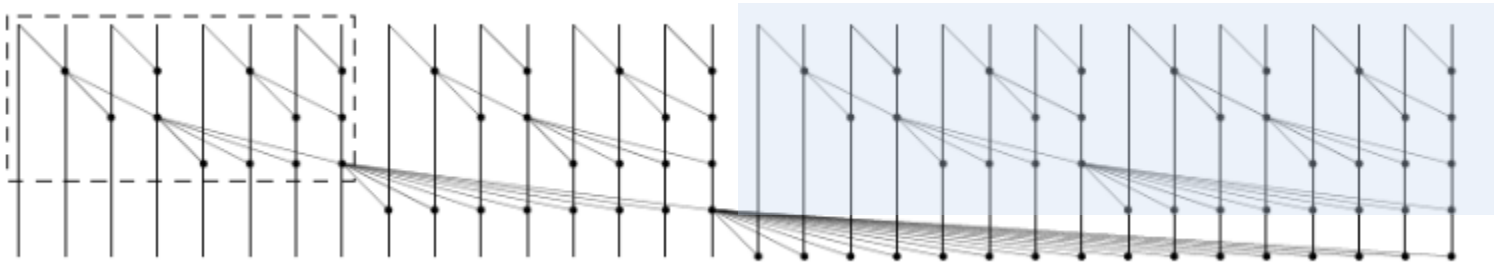


Recursive, log depth (Sklansky)

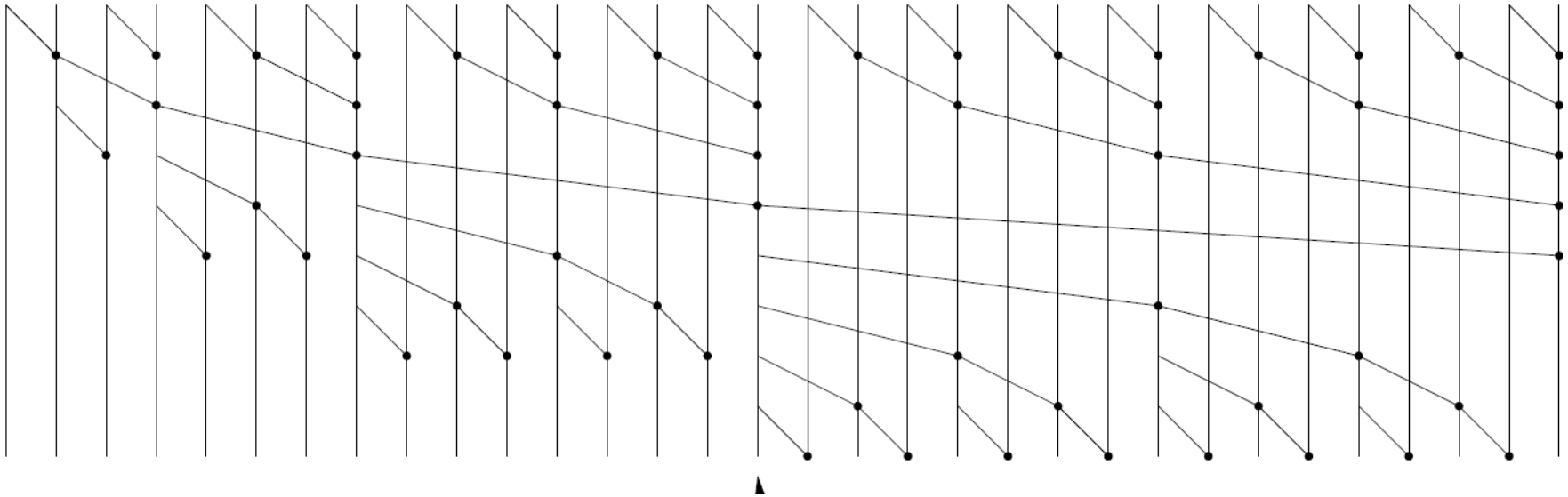


(operator associative, not necessarily commutative)

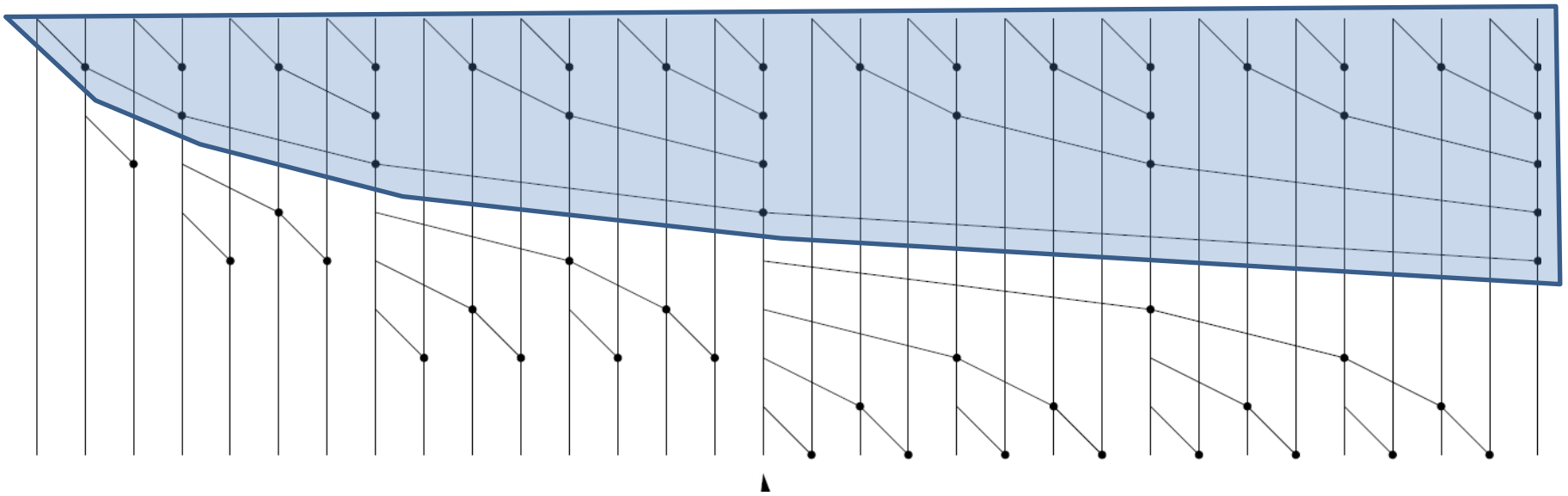
recursive (log depth) Sklansky



Brent Kung (’79)

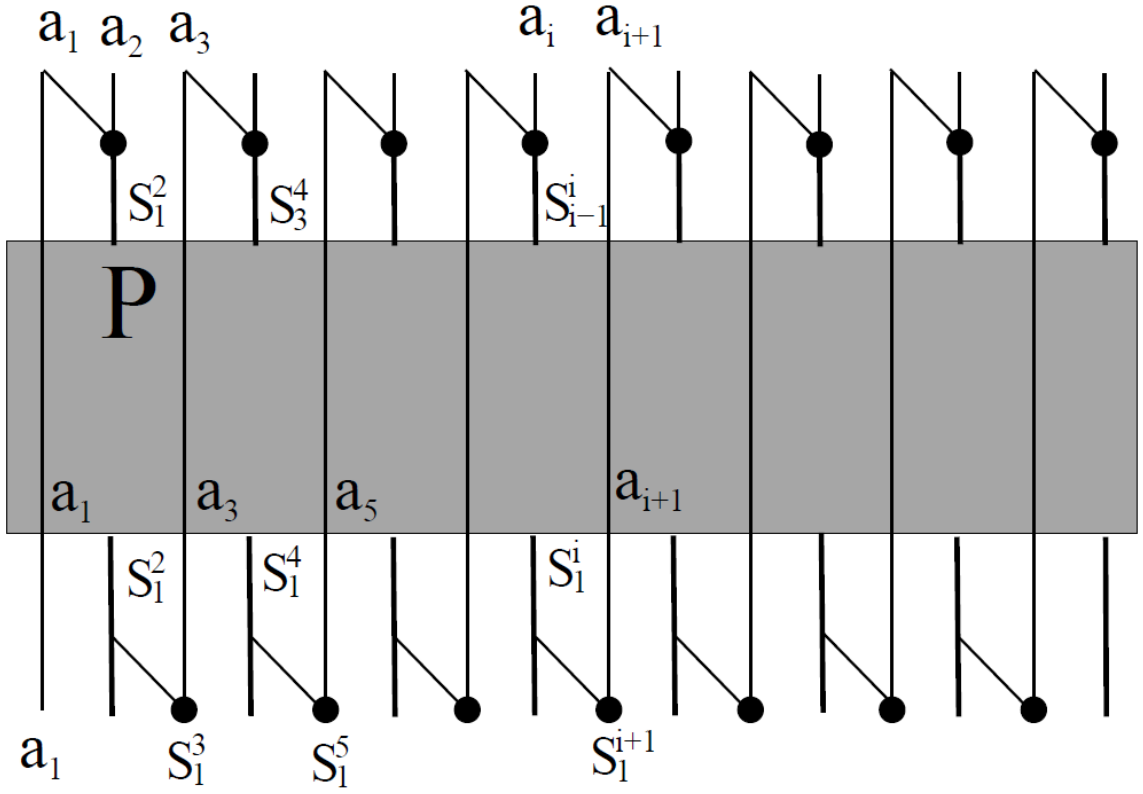


Brent Kung



forward tree + several reverse trees

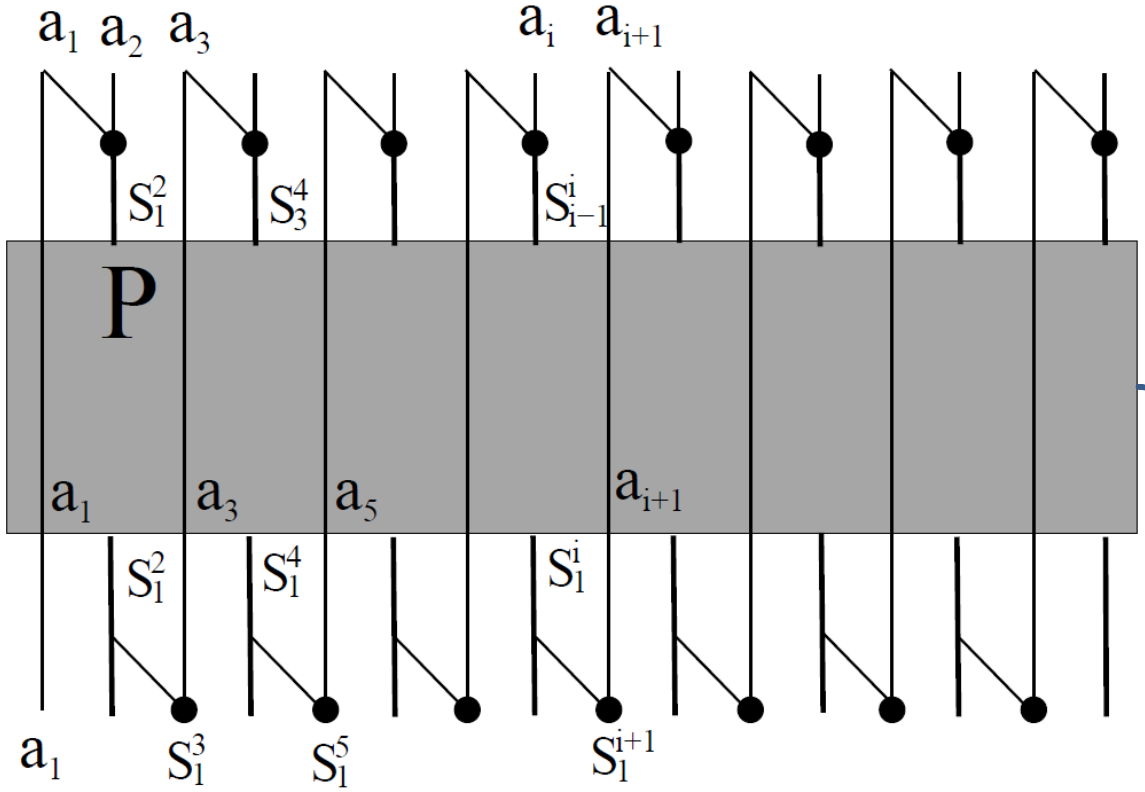
recursive decomposition



$S_i^j = a_i * a_{i+1} * \ldots * a_j$

indices from 1 here

recursive decomposition



$S_i^j = a_i * a_{i+1} * \ldots * a_j$

one recursive call on n/2 inputs

divide
conquer
combine

prescan

scan ”shifted right by one”

prescan of

[a_1 , a_2, ,a_3, ,a_4, . . . , a_n]

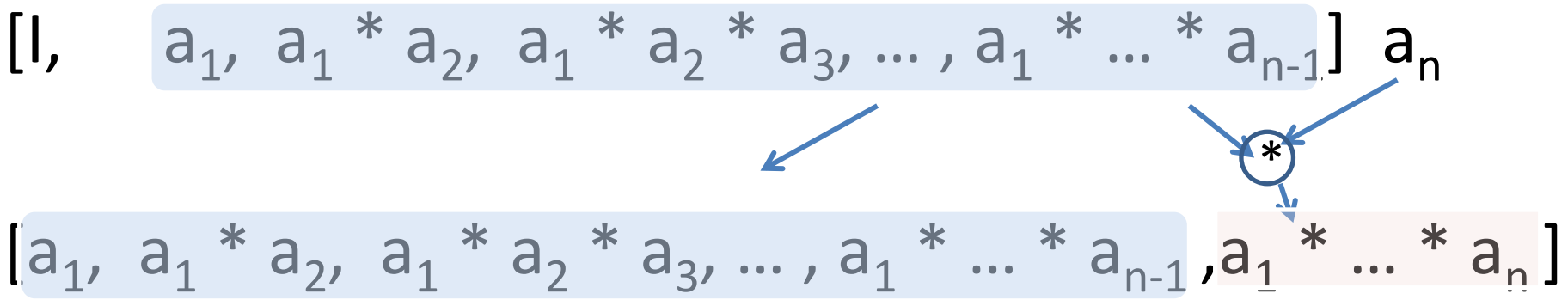
is

[I, a_1, $a_1 * a_2$, $a_1 * a_2 * a_3$, … , $a_1 * … * a_{n-1}$]

identity element

scan from prescan

easy (constant time)



prescan in NESL

```
function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});
```

prescan

```
function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});
```

```
scan_op('+, 0, [2, 8, 3, -4, 1, 9, -2, 7]);

is:

scan_op = fn : ((b, b) -> b, b, [b]) -> [b] :: (a in any; b in any)

it = [0, 2, 10, 13, 9, 10, 19, 17] : [int]
```

prescan

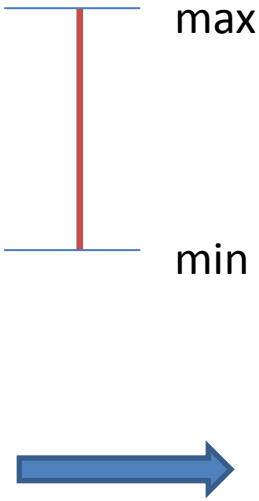
```
function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});
```

```
scan_op(max, 0, [2, 8, 3, -4, 1, 9, -2, 7]);

is:

scan_op = fn : ((b, b) -> b, b, [b]) -> [b] :: (a in any; b in any)


it = [0, 2, 8, 8, 8, 8, 9, 9] : [int]
```

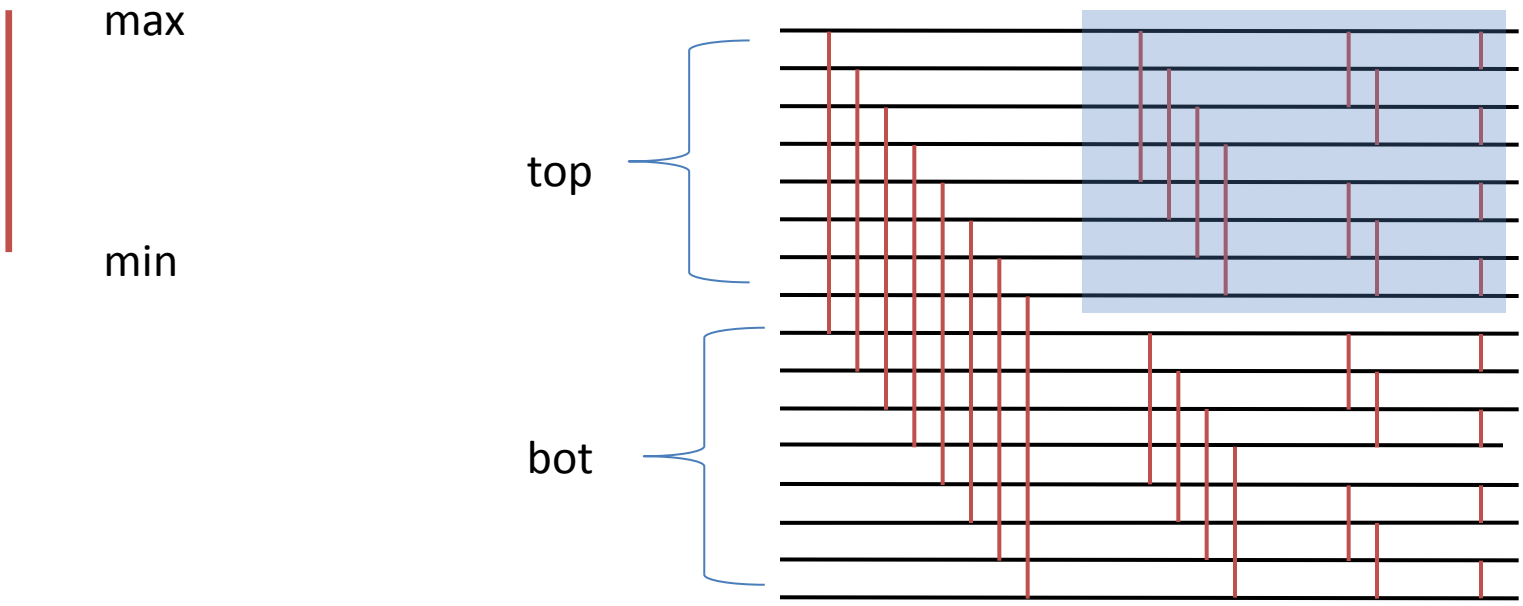
Batcher’s bitonic merge

```
function bitonic_sort(a) =
if (#a == 1) then a
else
    let
            bot = subseq(a,0,#a/2);
            top = subseq(a,#a/2,#a);
            mins = {min(bot,top):bot;top};
            maxs = {max(bot,top):bot;top};
    in flatten({bitonic_sort(x) : x in [mins,maxs]});
```

bitonic_sort (merger)



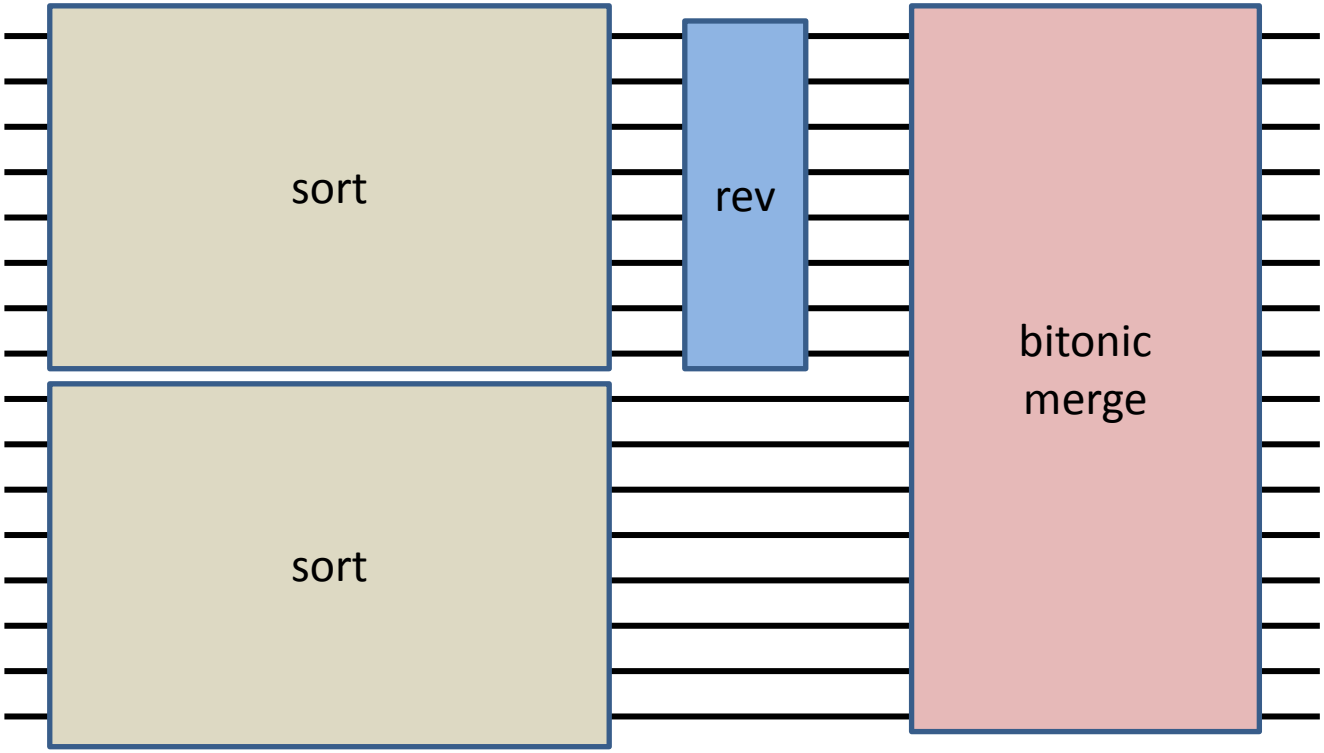
bitonic_sort (merger)



bitonic sort

```
function batcher_sort(a) =
if (#a == 1) then a
else
    let b = {batcher_sort(x) : x in bottop(a)};
    in bitonic_sort(b[0]++reverse(b[1]));
```

bitonic sort



parentheses matching

For each index, return the index of the matching parenthesis

```
function parentheses_match(string) =
let
    depth = plus_scan({if c==`( then 1 else -1 : c in string});
    depth = {d + (if c==`( then 1 else 0): c in string; d in depth};
    rnk = permute([0:#string], rank(depth));
    ret = interleave(odd_elts(rnk), even_elts(rnk))
in permute(ret, rnk);
```

(	)	(	(	)	(	)	)	(	(	(	)	)	)
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1

(	)	(	(	)	(	)	)	(	(	(	)	)	)
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1
0	1	0	1	2	1	2	1	0	1	2	3	2	1

prescan
(+)

(	)	(	(	)	(	)	)	(	(	(	)	)	)
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1
0	1	0	1	2	1	2	1	0	1	2	3	2	1
1	1	1	2	2	2	2	1	1	2	3	3	2	1

+1 if (
+0 if)

(	)	(	(	)	(	)	)	(	(	(	)	)	)	
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1	
0	1	0	1	2	1	2	1	0	1	2	3	2	1	
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth

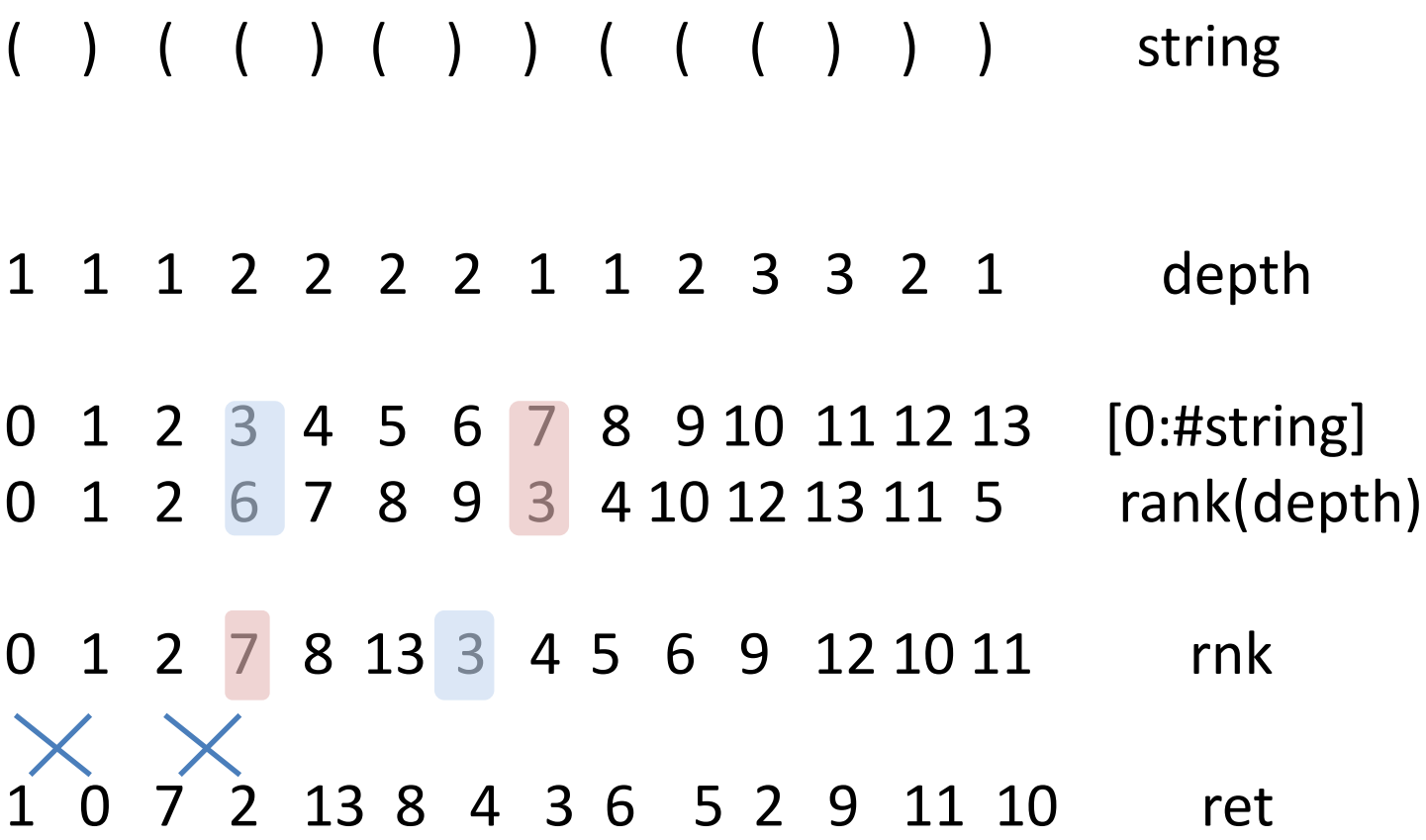
+1 if (
+0 if)

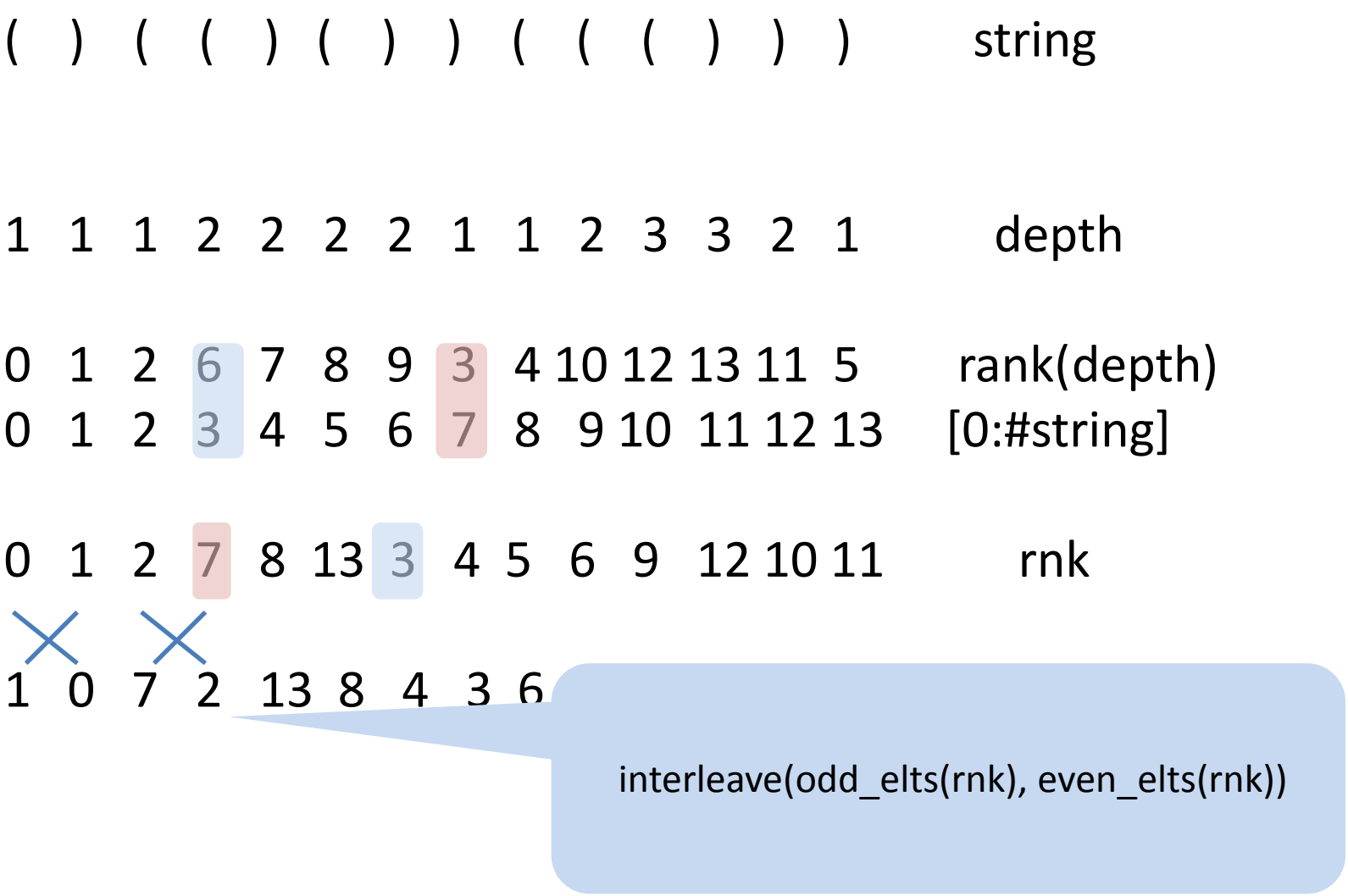
(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1	
0	1	0	1	2	1	2	1	0	1	2	3	2	1	
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)

(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1	
0	1	0	1	2	1	2	1	0	1	2	3	2	1	
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	3	4	5	6	7	8	9	10	11	12	13	[0:#string]
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)
0	1	2	7	8	13	3	4	5	6	9	12	10	11	rnk

(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1	
0	1	0	1	2	1	2	1	0	1	2	3	2	1	
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	3	4	5	6	7	8	9	10	11	12	13	[0:#string]
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)
0	1	2	7	8	13	3	4	5	6	9	12			

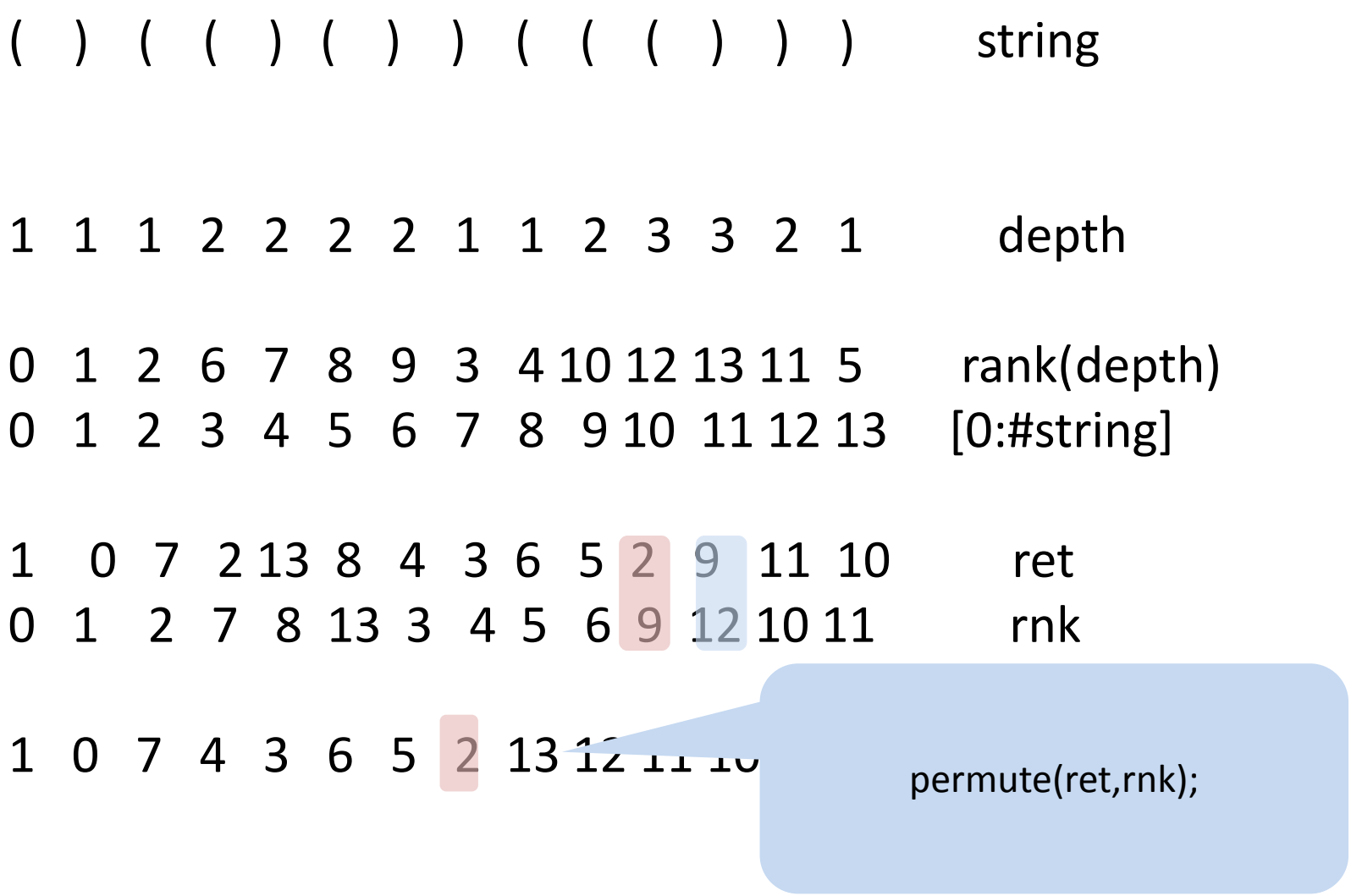
permute
([0:#string),rank(depth));

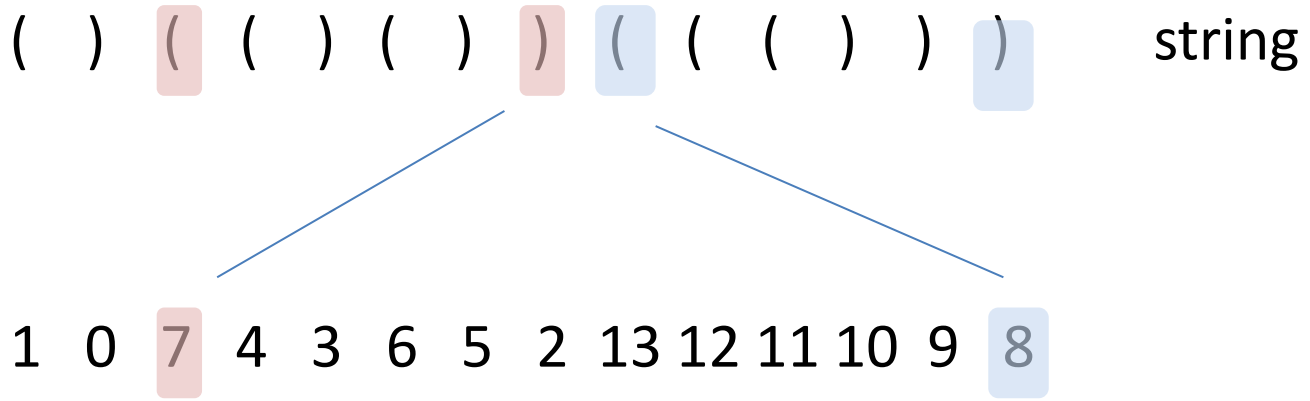

( ) (( ) ( )) ((( ))) string
1 1 1 2 2 2 2 1 1 2 3 3 2 1 depth
0 1 2 3 4 5 6 7 8 9 10 11 12 13 [0:#string]
0 1 2 6 7 8 9 3 4 10 12 13 11 5 rank(depth)
0 1 2 7 8 13 3 4 5 6 9 12 10 11 rnk
1 0 7 2 13 8 4 3 6 5 2 9 11 10 ret


( ) (( ) ( )) ((( ))) string
1 1 1 2 2 2 2 1 1 2 3 3 2 1 depth
0 1 2 6 7 8 9 3 4 10 12 13 11 5 rank(depth)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 [0:#string]
0 1 2 7 8 13 3 4 5 6 9 12 10 11 rnk
1 0 7 2 13 8 4 3 6
interleave(odd_elts(rnk), even_elts(rnk))

(	)	(	(	)	(	)	)	(	(	(	)	)	)	string
1	1	1	2	2	2	2	1	1	2	3	3	2	1	depth
0	1	2	6	7	8	9	3	4	10	12	13	11	5	rank(depth)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	[0:#string]
1	0	7	2	13	8	4	3	6	5	2	9	11	10	ret
0	1	2	7	8	13	3	4	5	6	9	12	10	11	rnk

1	0	7	4	3	6	5	2	13	12	11	10	9	8


( ) (( ) ( )) ((( ))) string
1 1 1 2 2 2 2 1 1 2 3 3 2 1 depth
0 1 2 6 7 8 9 3 4 10 12 13 11 5 rank(depth)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 [0:#string]
1 0 7 2 13 8 4 3 6 5 2 9 11 10 ret
0 1 2 7 8 13 3 4 5 6 9 12 10 11 rnk
1 0 7 4 3 6 5 2 13
permute(ret,rnk);


( ) (( ) ( )) ((( )))
string
1 0 7 4 3 6 5 2 13 12 11 10 9 8

parentheses matching code again

For each index, return the index of the matching parenthesis

```
function parentheses_match(string) =
let
   depth = plus_scan({if c==`( then 1 else -1 : c in string});
   depth = {d + (if c==`( then 1 else 0): c in string; d in depth};
   rnk = permute([0:#string], rank(depth));
   ret = interleave(odd_elts(rnk), even_elts(rnk))
in permute(ret, rnk);
```

one scan, a map, a zipWith, two permutes and an interleave, also rank and odd_elts and even_elts

What does Nested mean??

```
{plus_scan(a) : a in [[2,3], [8,3,9], [7]]};
```

```
[[0, 2], [0, 8, 11], [0]] : [[int]], 11], [0]]
```

What does Nested mean??

```
{plus_scan(a) : a in [[2,3], [8,3,9], [7]]};
```

sequnece of sequences
apply to each of a PARALLEL
function

```
[[0, 2], [0, 8, 11], [0]] : [[int]]
```

What does Nested mean??

```
{plus_scan(a) : a in [[2,3], [8,3,9], [7]]};
```

sequnece of sequences apply to each of a PARALLEL function

```
[[0, 2], [0, 8, 11], [0]] : [[int]], 11], [0]]
```

Implemented using Blelloch's Flattening Transformation, which converts nested parallelism into flat. Brilliant idea, challenging to make work in fancier languages (see DPH and good work on Manticore (ML))

Back to examples

this prescan is actually flat

```
function scan_op(op,identity,a) =
if #a == 1 then [identity]
else
  let e = even_elts(a);
      o = odd_elts(a);
      s = scan_op(op,identity,{op(e,o): e in e; o in o})
  in interleave(s,{op(s,e): s in s; e in e});
```

same prescan in Repa (clear version)

```
import Data.Array.Repa   as   A

prescan0 :: (Elt a) => (a -> a -> a) -> a  -> (Array (Z :. Int) a) -> (Array (Z :. Int) a)

prescan0 f _ as | size (extent as) == 1 = fromList (Z :. (1 :: Int)) [i]

prescan0 f i as | otherwise = let es = selEven as
                                  os = selOdd  as
                                  ss = prescan0 f i (A.zipWith f es os)
                              in interleave2 ss (A.zipWith f ss es)
```

using repa 2.1.1.5 because that works with GHC 7.0.4, the one from the latest Haskell platform.

If you are using GHC 7.4, get a later Repa version

same prescan in Repa (my fastest so far)

```
-- assumes input of length a power of 2
prescan :: (Elt a) => (a -> a -> a) -> a  -> (Array (Z :. Int) a) -> (Array (Z :. Int) a)
{-# INLINE prescan #-}
prescan f !i !as = sc as
   where
       sc as | size (extent as) == 1 = force $ fromList (Z :. (1 :: Int)) [i]
       sc as | otherwise =
           let es = force $ selEven as
               os = force $ selOdd  as
               ss = force $ sc (A.zipWith f es os)
           in as `deepSeqArray` interleave2M ss (A.zipWith f ss es)
```

5 or 6 times faster for sumAll . prescan (+) (0::Int) on 2^20 inputs

still 3-4 times slower than scanl1 ☹ but good speedup on 2 cores -N4

and hopefully on more

the power of scan

Blelloch pointed out that once you have scan
you can do LOTS of interesting algorithms, inc.

To lexically compare strings of characters. For example, to determine that "strategy" should appear before "stratification" in a dictionary

To evaluate polynomials

To solve recurrences. For example, to solve the recurrences

$$x_i = a_i x_{i-1} + b_i x_{i-2} \text{ and } x_i = a_i + b_i / x_{i-1}$$

To implement radix sort

To implement quicksort

To solve tridiagonal linear systems

To delete marked elements from an array

To dynamically allocate processors

To perform lexical analysis. For example, to parse a program into tokens

and many more

http://www.cs.cmu.edu/~blelloch/papers/Ble93.pdf

Back to examples
Batcher's bitonic merge IS NESTED

```
function bitonic_sort(a) =
if (#a == 1) then a
else
    let
            bot = subseq(a,0,#a/2);
            top = subseq(a,#a/2,#a);
            mins = {min(bot,top):bot;top};
            maxs = {max(bot,top):bot;top};
    in flatten({bitonic_sort(x) : x in [mins,maxs]});
```

and so is the sort

Back to examples
Batcher's bitonic merge IS NESTED

nestedness is good for D&C
and for irregular computations

```
function bitonic_sort(a) =
if (#a == 1) then a
else
    let
            bot = subseq(a,0,#a/2);
            top = subseq(a,#a/2,#a);
            mins = {min(bot,top):bot;top};
            maxs = {max(bot,top):bot;top};
    in flatten({bitonic_sort(x) : x in [mins,maxs]});
```

and so is the sort

Back to examples
parentheses matching is FLAT

For each index, return the index of the matching parenthesis

```
function parentheses_match(string) =
let
   depth = plus_scan({if c==`( then 1 else -1 : c in string});
   depth = {d + (if c==`( then 1 else 0): c in string; d in depth};
   rnk = permute([0:#string], rank(depth));
   ret = interleave(odd_elts(rnk), even_elts(rnk))
in permute(ret, rnk);
```

What about a cost model?

Blelloch empasises

1) work : total number of operations
   represents total cost (integral of needed resources over time)

2) depth or span: longest chain of sequential dependencies
   best possible running time on an unlimited number of processors

claims:

1) easier to think about algorithms based on work and depth than to use running time on machine with P processors (e.g. PRAM)
2) work and depth predict running time on various different machines
   (at least in the abstract)

Part 1: simple language based performance model

Call-by-value λ-calculus

$$\lambda x.\, e \Downarrow \lambda x.\, e \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.\, e \quad e_2 \Downarrow v \quad e[v/x] \Downarrow v'}{e_1\, e_2 \Downarrow v'} \qquad \text{(APP)}$$

slide from Blelloch's ICFP10 invited talk

The Parallel λ-calculus: cost model

$$e \Downarrow v; w, d$$

Reads: expression *e* evaluates to *v* with work *w* and span *d*.

- **Work** (W): sequential work
- **Span** (D): parallel depth

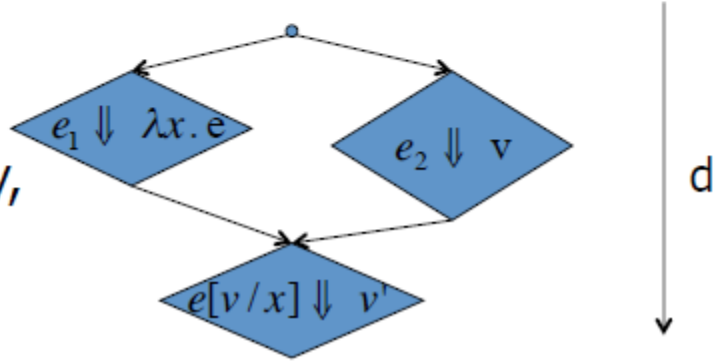
slide from Blelloch's ICFP10 invited talk

The Parallel λ-calculus: cost model

$$\lambda x.\,\mathrm{e} \Downarrow \lambda x.\,\mathrm{e};\ 1,1 \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.\,e;\ w_1, d_1 \quad e_2 \Downarrow v;\ w_2, d_2 \quad e[v/x] \Downarrow v';\ w_3, d_3}{e_1\ e_2 \Downarrow v';\ 1 + w_1 + w_2 + w_3,\ 1 + \max(d_1, d_2) + d_3} \qquad \text{(APP)}$$

Work adds
Span adds sequentially,
and max in parallel



slide from Blelloch's ICFP10 invited talk

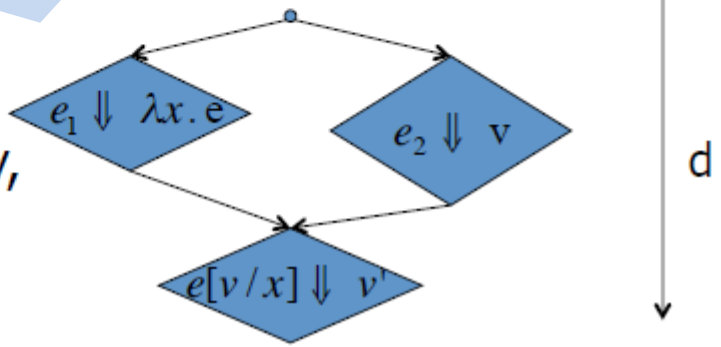
The Parallel λ-calculus: cost model

$\lambda x.e \Downarrow \lambda x.e$ (LAM)

$$\frac{e_1 \Downarrow \quad}{e_1}$$

Oops. The colour is wrong. Should be pink (not blue)

Work adds
Span adds sequentially,
and max in parallel



slide from Blelloch's ICFP10 invited talk

The Parallel λ-calculus cost model

$$\lambda x.\,\mathrm{e} \Downarrow \lambda x.\,\mathrm{e};\ 1,1 \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.\,e;\ w_1,d_1 \quad e_2 \Downarrow v;\ w_2,d_2 \quad e[v/x] \Downarrow v';\ w_3,d_3}{e_1\ e_2 \Downarrow v';\ 1+w_1+w_2+w_3,\ 1+\max(d_1,d_2)+d_3} \qquad \text{(APP)}$$

$$c \Downarrow c;\ 1,1 \qquad \text{(CONST)}$$

$$\frac{e_1 \Downarrow c;\ w_1,d_1 \quad e_2 \Downarrow v;\ w_2,d_2 \quad \delta(c,v) \Downarrow v'}{e_1\ e_2 \Downarrow v';\ 1+w_1+w_2,\ 1+\max(d_1,d_2)} \qquad \text{(APPC)}$$

$$c_n = 0,\cdots,n,+,+_0,\cdots,+_n,<,<_0,\cdots,<_n,\times,\times_0,\cdots,\times_n,\cdots \qquad \text{(constants)}$$

slide from Blelloch's ICFP10 invited talk

Adding Functional Arrays: NESL

$$\{e_1 : x \text{ in } e_2 \mid e_3\}$$

$$\frac{e'[v_i/x] \Downarrow v_i';\ w_i, d_i \quad i \in \{1 \ldots n\}}{\{e' : x \text{ in } [v_1 \ldots v_n]\} \Downarrow [v_1' \ldots v_n'];\ 1 + \sum_{i=1}^{n} w_i,\ 1 + \max_{i=1}^{|v|} d_i}$$

Primitives:

```
<- : 'a seq * (int,'a) seq -> 'a seq
```

```
• [g,c,a,p] <- [(0,d),(2,f),(0,i)]
     [i,c,f,p]
```

[ICFP95]

```
elt, index, length
```

slide from Blelloch's ICFP10 invited talk

Adding Functional Arrays: NESL

$$\{e_1 : x \text{ in } e_2 \mid e_3\}$$

Blelloch:
programming based cost models could change the way people think about costs and open door for other kinds of abstract costs
doing it in terms of machines.... "that's so last century"

```
<- : 'a seq * (int,'a) seq -> 'a seq
• [g,c,a,p] <- [(0,d),(2,f),(0,i)]
     [i,c,f,p]

elt, index, length
```

[ICFP95]

slide from Blelloch's ICFP10 invited talk

The Second Half:
Provable Implementation Bounds

Theorem [FPCA95]:If $e \Downarrow v; w, d$ then v can be calculated from e on a CREW PRAM with p processors in $O\left(\frac{w}{p} + d \log p\right)$ time.

Can't really do better than: $\max\left(\frac{w}{p}, d\right)$

If w/p > d log p then "work dominates"

We refer to w/p as the parallelism.

slide from Blelloch's ICFP10 invited talk

The Second Half:

Blelloch on programming parallel algorithms:

Start with work same as best sequential algorithm.
Work is most important.
Next reduce span.
Want work over p term to dominate.

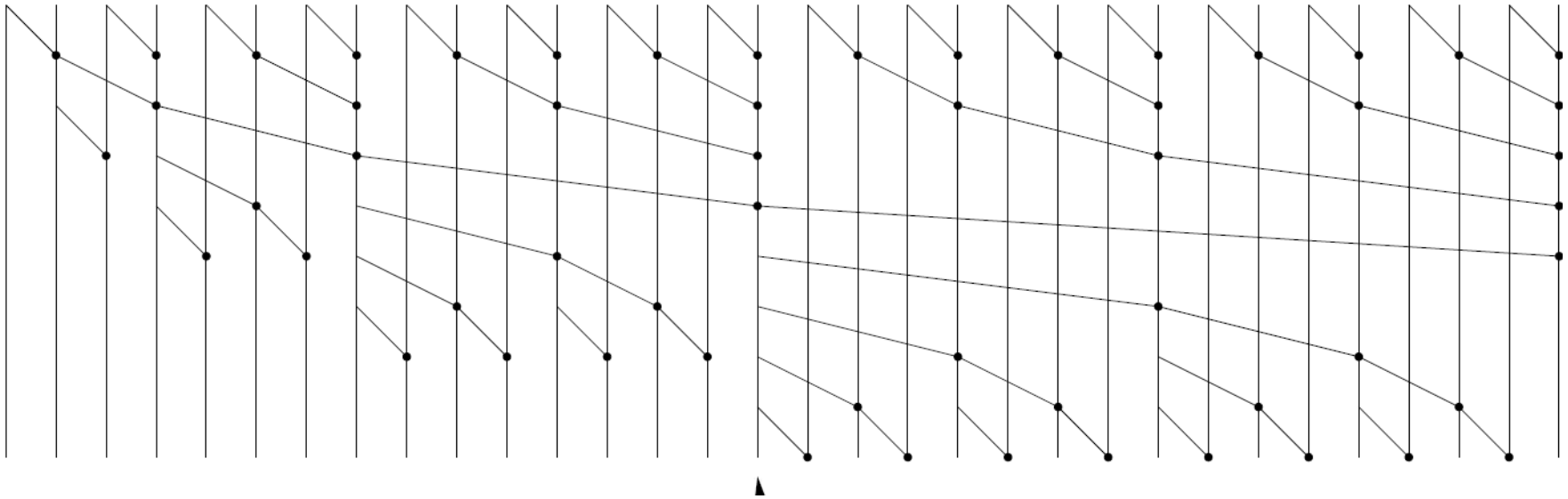
Can't really do better than: $\max\left(\frac{w}{p}, d\right)$

If w/p > d log p then "work dominates"

We refer to w/p as the parallelism.

slide from Blelloch's ICFP10 invited talk

Back to our scan



oblivious or data independent computation

N = 2^n inputs, work of dot is 1
work = ?
depth = ?

and for Sklansky? and bitonic sort? and quicksort in NESL?

Next lecture (Thursday)

Repa (flat)