Parallel Basic Blocks and Flattening Nested Parallelism

Cosmin E. Oancea and Troels Henriksen
[cosmin.oancea,athas]@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

September 2015 PMPH Lecture Notes

Almost Homomorphisms (Gorlatch)

"Systematic Extraction and Implementation of Divide-and-Conquer Parallelism", Sergei Gorlatch, 1996. (attached in TeachingMaterial/AdditionalMaterial/ListHom-Flattening).

Intuition: a non-homomorphic function g can be sometimes "lifted" to a homomorphic one f, by computing a baggage of *extra info*.

The initial problem obtained by projecting the homomorphic result: $g \ = \pi \ . \ f$

Maximum-Segment Sum Problem (MSS):
Given a list of integers, find the contiguous segment of the list whose members have the largest sum among all such segments.
The result is only the maximal sum (not the segment's members).

E.g., `mss` [1, -2, 3, 4, -1, 5, -6, 1] = 11
(the corresponding segment is [3, 4, -1, 5]).

Maximum Segment Sum

Incorrect list-homomorphism implementation

```
mss []         = 0
mss (x ++ y) = (mss x) ↑ (mss y) -- ↑ denotes Max
```

Incorrect: `(mss [1,-2,3,4])`↑`(mss [-1,5,-6,1])` $\equiv$ 7 ↑ 4 $\equiv$ 7
The correct result of `(mss [1, -2, 3, 4, -1, 5, -6, 1])` is 11, corresponding to segment [3, 4, -1, 5].

The segment of interest may lie partly in `x` and partly in `y`. To construct a homomorphism we need to compute extra information:

Maximum Segment Sum

Incorrect list-homomorphism implementation

```
mss []       = 0
mss (x ++ y) = (mss x) ↑ (mss y) -- ↑ denotes Max
```

Incorrect: `(mss [1,-2,3,4])`↑`(mss [-1,5,-6,1])` $\equiv 7 \uparrow 4 \equiv 7$
The correct result of `(mss [1, -2, 3, 4, -1, 5, -6, 1])` is 11, corresponding to segment [3, 4, -1, 5].

The segment of interest may lie partly in `x` and partly in `y`. To construct a homomorphism we need to compute extra information:

- maximum concluding segment: `mcs x = mcs [1,-2,3,4] = 7`
- maximum initial segment: `mis y = mis [-1,5,-6,1] = 4`
- total segment sum: `ts [1,-2,3,4] = 6`

Maximum Segment Sum

Incorrect list-homomorphism implementation

```
mss []         = 0
mss (x ++ y) = (mss x) ↑ (mss y) -- ↑ denotes Max
```

Incorrect: `(mss [1,-2,3,4])↑(mss [-1,5,-6,1])` $\equiv 7 \uparrow 4 \equiv 7$
The correct result of `(mss [1, -2, 3, 4, -1, 5, -6, 1])` is 11, corresponding to segment [3, 4, -1, 5].

The segment of interest may lie partly in `x` and partly in `y`. To construct a homomorphism we need to compute extra information:

- maximum concluding segment: `mcs x = mcs [1,-2,3,4] = 7`
- maximum initial segment: `mis y = mis [-1,5,-6,1] = 4`
- total segment sum: `ts [1,-2,3,4] = 6`
- `mis (x++y) = (mis x) ↑ ((ts x)+(mis y))`, similar for `mcs`
- `mss (x++y) = (mss x) ↑ (mss y) ↑ ((mcs x) + (mis y))`

Maximum-Segment Sum = Near Homomorphism

Correct Solution – Test it in Haskell!

```
-- x ↑ y = if(x >= y) then x else y
(mssx, misx, mcsx, tsx) ⊙ (mssy, misy, mcsy, tsy) = (
        (mssx ↑ mssy ↑ (mcsx+misy),
         misx ↑ (tsx+misy),
        (mcsx+tsy) ↑ mcsy,
         tsx + tsy
    )

f x = (x ↑ 0, x ↑ 0, x ↑ 0, x)

emss = (reduce ⊙ (0,0,0,0)) . (map f)

mss  = π1 . emss
       where π1 (a, _, _, _) = a
```

The baggage: 3 extra integers (`misx`, `mcsx`, `tsx`) and a constant number of integer operations per communication stage.

Longest Satisfying Segment Problems

- Class of problems which requires to find the longest segment of a list for which some property holds, such as:
- longest sequence of zeros, or longest sequence made from the same number, or longest sorted sequence.
- Not all predicates can be written as a list homomorphism, e.g., longest sequence whose sum is 0.

Restrict The Shape of the Predicate to:

```
p []              = True
p [x]             = ...
p [x, y]          = ...
p [x : y : zs] = (p [x,y]) ∧ p (y : zs)
```

Longest Satisfying Segment Problems

Restrict the Shape of the Predicate:

```
zeros [x]   = x == 0                same [x]   = True     sorted [x]   = True
zeros [x,y] =   (zeros [x])         same [x,y] = x == y   sorted [x,y] = x <= y
              ∧ (zeros [y])
```

Extra Baggage:

- As before, the length of the longest initial/concluding satisfying segments (`lis`/`lcs`), and the total list length (`tl`).
- When considering the concatenation of the `(lcs, lis)` pair, it is not guaranteed that the result satisfies the predicate e.g., `(sorted x)` $\wedge$ `(sorted y)` $\not\Rightarrow$ `sorted x++y`.
- We also need the *last* element of `lcs` and the *first* elem of `lis`,
- in order to compute whether `(lcs x)` is *connected* to `(lis y)`, i.e., `p [lastx,firsty] == True`
- Boolean indicating whether the whole list satisfies `p` (`ok`).

Longest Satisfying Segment Problem: Exercise

Exercise: fill in the blanks, test in Haskell for zeros/same/sorted

```
(lssx, lisx, lcsx, tlx, firstx, lastx, okx) ⊙
(lssy, lisy, lcsy, tly, firsty, lasty, oky)

  = (newlss, newlis, newlcs, tlx+tly, firstx, lasty, newok)
     where
        connect = ...
        newlss  = ...
        newlis  = ...
        newlcs  = ...
        newok   = ...

f x = (xmatch, xmatch, xmatch, 1, x, x, p [x])
    where xmatch = if (p [x]) then 1 else 0

elss = (reduce (⊙) (0,0,0,0,0,0,True)) . (map f)

lss  = π1 . elss
        where π1 (a, _, _, _, _, _, _) = a
```

1. Homomorphisms (Continuation)
   - Almost Homomorphisms Gorlatch'96
   - Scan as a Distributable Homomorphism
2. Implementation of Flat Bulk Operators
   - Implementation of Reduce and Scan
   - Other Second-Order Bulk Operators
   - Implementation of Segmented Scan
3. Nested Data-Parallel Applications
   - Sieve: Prime-Numbers Computation
   - Nested Parallel Quicksort
4. Flattening Nested Parallelism
   - Rules For Flattening
   - Flattening Prime-Number Computation
   - Flattening Quicksort

All Homomorphism Are Efficient?

If the combine operator involves concatenation then does map-reduce provides efficient parallelization?

Merge Sort

```
-- merge two sorted lists
merge :: Ord T => [T] -> [T] -> [T]
merge [] y  = y
merge x  [] = x
merge (x:xs) (y:ys) =
  if ( x <= y )
  then x : merge xs (y:ys)
  else y : merge (x:xs) ys
```

```
-- mSort = hom merge [.] []
-- [.] x = [x]
mSort :: Ord T => [T] -> [T]
mSort []     = []
mSort [x]    = [x]
mSort (x++y) = (mSort x) `merge`
               (mSort y)
```

In the naive merged sort, the `merge` reduction operator traverses sequentially the whole list, hence this map-reduce does not give efficient parallelization!

Distributable Homomorphism (DH)

- DH: a class of homomorphisms that allows efficient parallel implem even if concatenation appears in the reduction operator.
- Requires that the length of the list is a power of 2, and at every step the list is split in half.
- `zipWith ::` $[\alpha] \rightarrow [\beta] \rightarrow [\gamma]$,
  `zipWith` $\odot$ $[x_1,\ldots,x_n]$ $[y_1,\ldots,y_n] \equiv [x_1 \odot y_1,\ldots,x_n \odot y_n]$

Definition (Distributable Homomorphism (DH))

Given two associative binary operators $\oplus$ and $\otimes$ we define operator

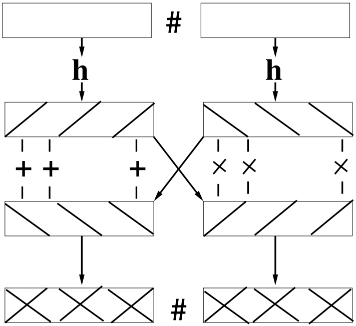
$$\texttt{dhop} :: [a] \rightarrow [a] \rightarrow [a]$$
$$\texttt{dhop}\ u\ v = (zipWith \oplus u\ v) +\!\!+ (zipWith \otimes u\ v)$$

We write $\oplus \updownarrow \otimes$ for the LH with combine operator `dhop` $\oplus$ $\otimes$

$$\oplus \updownarrow \otimes\ [a] = [a]$$
$$\oplus \updownarrow \otimes\ (x +\!\!+ y) = (\oplus \updownarrow \otimes\ x)\ \text{`}\texttt{dhop}\text{`}\ (\oplus \updownarrow \otimes\ y)$$

Function $h :: [T] -> [T]$ is a distributable homomorphism iff $h = \oplus \updownarrow \otimes$ for some binary associative operators $\oplus$ and $\otimes$.

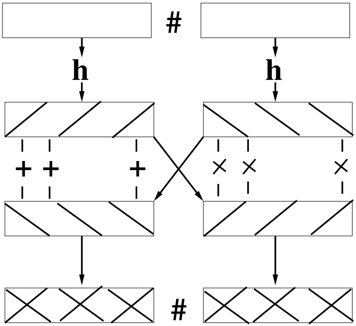
Distributed Reduce is a DH



- `dhop u v = (zipWith` $\oplus$ `u v) ++`
  `(zipWith` $\otimes$ `u v)`
- $\oplus \updownarrow \otimes$ `(x ++ y) = (`$\oplus \updownarrow \otimes$ `x)`
  `` `dhop` `` `(`$\oplus \updownarrow \otimes$ `y)`

- For example, distributed reduction: `distrRed (`$\odot$`)` $e_\odot$ `x =`
  `[reduce` $\odot$ $e_\odot$ `x,...,` `reduce` $\odot$ $e_\odot$ `x]`
- is a DH: distrRed $\odot = \odot \updownarrow \odot$

Scan is a DH

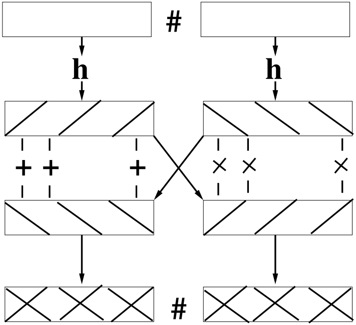


- `dhop u v = (zipWith` $\oplus$ `u v) ++`
  `(zipWith` $\otimes$ `u v)`
- $\oplus \updownarrow \otimes$ `(x ++ y) = (`$\oplus \updownarrow \otimes$ `x)`
  `` `dhop` `` `(`$\oplus \updownarrow \otimes$ `y)`
- This implementation of scan is not work efficient, i.e., work $O(N\ lg\ N)$!

- `scan` $\odot$ $\texttt{e}_\odot$ $(\texttt{x} + +\texttt{y}) = S_1 \oslash S_2 = S_1$ `++ (map (`$\odot$ `s)` S_2`)`,
  where S_1= `scan` $\odot$ $\texttt{e}_\odot$ `x`, S_2= `scan` $\odot$ $\texttt{e}_\odot$ `y`, and `s=last` S_1

- `dhScan` $\odot$ $\texttt{e}_\odot$ `= (map` π_1`) . (`$\oplus \updownarrow \otimes$`) . (map pair)`,
  where π_1 `(a,b) = a,` `pair a = (a,a)`,
  $(\texttt{s}_1,\texttt{r}_1) \oplus (\texttt{s}_2,\texttt{r}_2) = (?,\ ?)$
  $(\texttt{s}_1,\texttt{r}_1) \otimes (\texttt{s}_2,\texttt{r}_2) = (?,\ ?)$

Scan is a DH



- `dhop u v = (zipWith` $\oplus$ `u v) ++`
  `(zipWith` $\otimes$ `u v)`
- $\oplus \updownarrow \otimes$ `(x ++ y) = (`$\oplus \updownarrow \otimes$ `x)`
  `` `dhop` `` `(`$\oplus \updownarrow \otimes$ `y)`
- This implementation of scan is not work efficient, i.e., work $O(N\ lg\ N)$!

- `scan` $\odot$ $e_\odot$ $(x++y) = S_1 \oslash S_2 = S_1$ `++ (map (`$\odot$ `s)` S_2`)`,
  where S_1= `scan` $\odot$ $e_\odot$ `x`, S_2= `scan` $\odot$ $e_\odot$ `y`, and `s=last` S_1

- `dhScan` $\odot$ $e_\odot$ `= (map` π_1`) . (`$\oplus \updownarrow \otimes$`) . (map pair)`,
  where π_1 `(a,b) = a,` `pair a = (a,a)`,
  $(s_1, r_1) \oplus (s_2, r_2) = (s_1,\ r_1 \odot r_2)$
  $(s_1, r_1) \otimes (s_2, r_2) = (r_1 \odot s_2,\ r_1 \odot r_2)$

Map, Reduce, and Scan Types and Semantics

- $\texttt{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$
  $\texttt{map f } [\texttt{x}_1, \ldots, \texttt{x}_n] = [\texttt{f}(\texttt{x}_1), \ldots, \texttt{f}(\texttt{x}_n)]$,
  i.e., $\texttt{x}_i :: \alpha, \forall i$, and $\texttt{f} :: \alpha \rightarrow \beta$.

- $\texttt{reduce} :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha] \rightarrow \alpha$
  $\texttt{reduce} \odot \texttt{ e } [\texttt{x}_1, \texttt{x}_2, \ldots, \texttt{x}_n] = \texttt{e} \odot \texttt{x}_1 \odot \texttt{x}_2 \odot \ldots \odot \texttt{x}_n$,
  i.e., $\texttt{e} :: \alpha$, $\texttt{x}_i :: \alpha, \forall i$, and $\odot :: \alpha \rightarrow \alpha \rightarrow \alpha$.

- $\texttt{scan}^{exc} :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha] \rightarrow [\alpha]$
  $\texttt{scan}^{exc} \odot \texttt{ e } [\texttt{x}_1, \ldots, \texttt{x}_n] = [\texttt{e}, \texttt{e} \odot \texttt{x}_1, \ldots, \texttt{e} \odot \texttt{x}_1 \odot \ldots \texttt{x}_{n-1}]$
  i.e., $\texttt{e} :: \alpha$, $\texttt{x}_i :: \alpha, \forall i$, and $\odot :: \alpha \rightarrow \alpha \rightarrow \alpha$.

- $\texttt{scan}^{inc} :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha] \rightarrow [\alpha]$
  $\texttt{scan}^{inc} \odot \texttt{ e } [\texttt{x}_1, \ldots, \texttt{x}_n] = [\texttt{e} \odot \texttt{x}_1, \ldots, \texttt{e} \odot \texttt{x}_1 \odot \ldots \texttt{x}_n]$
  i.e., $\texttt{e} :: \alpha$, $\texttt{x}_i :: \alpha, \forall i$, and $\odot :: \alpha \rightarrow \alpha \rightarrow \alpha$.

Parallel Random Access Machine (PRAM)

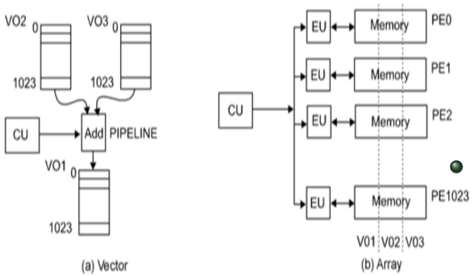
PRAM focuses exclusively on parallelism and ignores issues related to synchronization and communication:

1. p processors connected to shared memory
2. each processor has an unique id (index) i, $1 \leq i \leq p$
3. SIMD execution, each parallel instruction requires unit time,
4. each processor has a flag that controls whether it is active in the execution of an instruction.

Parallel Random Access Machine (PRAM)

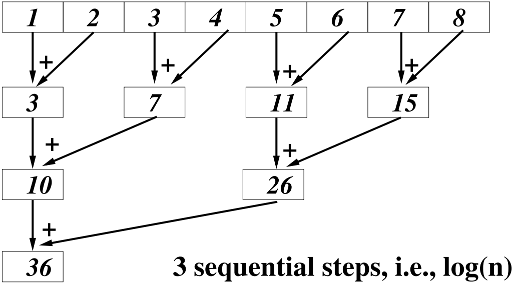
PRAM focuses exclusively on parallelism and ignores issues related to synchronization and communication:

1. p processors connected to shared memory
2. each processor has an unique id (index) i, $1 \leq i \leq p$
3. SIMD execution, each parallel instruction requires unit time,
4. each processor has a flag that controls whether it is active in the execution of an instruction.



- Work Time Algorithm (WT):
  - Work Complexity W(n): is the total # of ops performed,
  - Depth/Step Complexity D(n): is the # of sequential steps.
- If we know WT's work and depth, then Brent Theorem gives good complexity bounds for a PRAM.

Reducing in Parallel



Reducing an array of length `n` with `n/2` processors requires:

- work $W(n) = n$ and
- depth $D(n) = lg\ n$, i.e., number of sequential steps.
- optimized runtime with P processors: $(n/P) + lg\ P$.

Theorem (Brent Theorem)

A Work-Time Algorithm of depth $D(n)$ and work $W(n)$ can be simulated on a P-processor PRAM in time complexity T such that:

$$\frac{W(n)}{P} \leq T < \frac{W(n)}{P} + D(n)$$

Reduce: Algorithm and Complexity

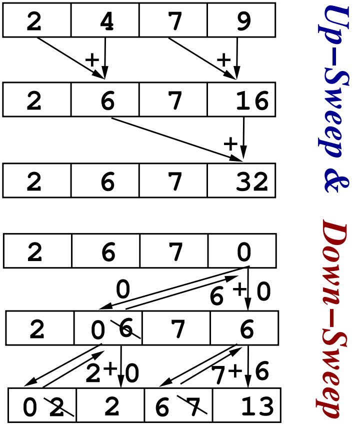
```
Input:   array A of n=2^k elems of type T
         ⊕ :: T × T → T associative
Output:  S = ⊕_{j=1}^{n} a_j

1.  forall i = 0 to n-1 do
2.    B[i] ← A[i]
3.  enddo

4.  for h = 1 to k do
5.    forall i = 0 to n-1 by 2^h do
6.      B[i] ← B[i] ⊕ B[i+2^(h-1)]
7.    enddo
8.  enddo
9.  S ← B[0]
```

- $D_{1-3}(n) = \Theta(1)$, $W_{1-3}(n) = \Theta(n)$,
- $D_{5-7}(n) = \Theta(1)$,
  $W_{5-7}(n, h) = \Theta(n/2^h)$,
- $D_{4-8}(n) = k \times D_{5-7}(n) = \Theta(lg\ \ n)$
- $W_{4-8}(n) = \sum_{h=1}^{k} W_{5-7}(n, h) =$
  $\Theta(\sum_{h=1}^{k}(n/2^h)) = \Theta(n)$
- $D_9(n) = \Theta(1)$, $W_9(n) = \Theta(1)$,

- $D(n) = \Theta(lg\ \ n), W(n) = \Theta(n)!$

$$\frac{n-1}{P} \leq Runtime < \frac{n-1}{P} + lg\ \ n$$

Parallel Exclusive Scan with Associative Operator $\oplus$



Two Steps:

- Up-Sweep: similar with reduction
- Root is replaced with neutral element.
- Down-Sweep:
  - the left child sends its value to parent and updates its value to that of parent.
  - the right-child value is given by $\oplus$ applied to the left-child value and the (old) value of parent.
  - note that the right child is in fact the parent, i.e., in-place algorithm.

Parallel Exclusive Scan Algorithm And Complexity

Input: array A of $n=2^k$ elems of type T
$\oplus :: T \times T \rightarrow T$ associative
Output: B = $[0,\ a_1,\ a_1 \oplus a_2, \ldots, \oplus_{j=1}^{n-1}\ a_j]$

```
forall i = 0 : n-1 do
  B[i] ← A[i]
enddo

for d = 0 to k-1 do // up-sweep
  forall i = 0 to n-1 by 2^(d+1) do
    B[i+2^(d+1)-1] ← B[i+2^d  -1] ⊕
                     B[i+2^(d+1)-1]
  enddo
enddo
B[n-1] = 0
for d = k-1 downto 0 do // down-sweep
  forall i = 0 to n-1 by 2^(d+1) do
    tmp ← B[i+2^d-1]
    B[i+2^d-1] ← B[i+2^(d+1)-1]
    B[i+2^(d+1)-1] ← tmp ⊕ B[i+2^(d+1)-1]
  enddo
enddo
```

- The code show exponentials for clarity, but those can be computed by one multiplication/division operation each sequential iteration.
- $D(n) = \Theta(lg\ n), W(n) = \Theta(n)!$
- Similar reasoning as with reduce.

Zip, ZipWith

- `zip ::` $[\alpha_1] \rightarrow [\alpha_2] \rightarrow [(\alpha_1,\alpha_2)]$
- `zip` $[a_1,\ldots,a_n]$ $[b_1,\ldots,b_m] \equiv [(a_1,b_1),\ldots,(a_q,b_q)]$, where `q = min(m,n)`.

Zip, ZipWith

- zip :: $[\alpha_1] \rightarrow [\alpha_2] \rightarrow [(\alpha_1,\alpha_2)]$
- zip $[a_1,\ldots,a_n]$ $[b_1,\ldots,b_m] \equiv [(a_1,b_1),\ldots,(a_q,b_q)]$, where q = min(m,n).
- unzip :: $[(\alpha_1,\alpha_2)] \rightarrow ([\alpha_1],[\alpha_2])$
- unzip $[(a_1,b_1),\ldots,(a_n,b_n)] \equiv ([a_1,\ldots,a_n],[b_1,\ldots,b_n])$,

Zip, ZipWith

- `zip ::` $[\alpha_1] \to [\alpha_2] \to [(\alpha_1,\alpha_2)]$
- `zip` $[a_1,\ldots,a_n]\ [b_1,\ldots,b_m] \equiv [(a_1,b_1),\ldots,(a_q,b_q)]$, where `q = min(m,n)`.
- `unzip ::` $[(\alpha_1,\alpha_2)] \to ([\alpha_1],[\alpha_2])$
- `unzip` $[(a_1,b_1),\ldots,(a_n,b_n)] \equiv ([a_1,\ldots,a_n],[b_1,\ldots,b_n])$,
- In some sense `zip` is syntactic sugar, for example one could work with the tuple of array representation, e.g.,
- `mapT ::` $((\alpha_1,\ldots,\alpha_m) \to (\beta_1,\ldots,\beta_n)) \to$
  $[\alpha_1] \to \ldots \to [\alpha_m] \to ([\beta_1],\ldots,[\beta_n]])$
- `mapT f` $\equiv$ `unzip`^{n} `. map f .` `zip`^{n}

Zip, ZipWith

- `zip ::` $[\alpha_1] \rightarrow [\alpha_2] \rightarrow [(\alpha_1,\alpha_2)]$
- `zip` $[a_1,\ldots,a_n]$ $[b_1,\ldots,b_m] \equiv [(a_1,b_1),\ldots,(a_q,b_q)]$, where `q = min(m,n)`.
- `unzip ::` $[(\alpha_1,\alpha_2)] \rightarrow ([\alpha_1],[\alpha_2])$
- `unzip` $[(a_1,b_1),\ldots,(a_n,b_n)] \equiv ([a_1,\ldots,a_n],[b_1,\ldots,b_n])$,
- In some sense `zip` is syntactic sugar, for example one could work with the tuple of array representation, e.g.,
- `mapT ::` $((\alpha_1,\ldots,\alpha_m) \rightarrow (\beta_1,\ldots,\beta_n)) \rightarrow$
  $[\alpha_1] \rightarrow \ldots \rightarrow [\alpha_m] \rightarrow ([\beta_1],\ldots,[\beta_n]])$
- `mapT f` $\equiv$ `unzip`^{n} `. map f . zip`^{n}
- `zipWith ::` $(\alpha_1 \rightarrow \alpha_2 \rightarrow \beta) \rightarrow [\alpha_1] \rightarrow [\alpha_2] \rightarrow [\beta]$
- `zipWith` $\odot$ $[a_1,\ldots,a_n]$ $[b_1,\ldots,b_n] \equiv [a_1 \odot b_1,\ldots,a_n \odot b_n]$
- `zipWith` $\odot \equiv$ `map (\(u,v)` $\rightarrow$ `u` $\odot$ `v) . zip`

Permute, Write, Split, Filter

- Operator to permute in parallel based on a set (array) of indices:
  permute :: [Int] $\rightarrow$ [α] $\rightarrow$ [α], e.g.,
  A (data vector) = `[a0, a1, a2, a3, a4, a5]`
  I (index vector) = `[3, 2, 0, 4, 1, 5 ]`
  permute I A = `[a2, a4, a1, a0, a3, a5]`

- Operator to write in parallel a set of values to correspond indices:
  write :: [Int] $\rightarrow$ [α] $\rightarrow$ [α] $\rightarrow$ [α]
  A (data vector) = `[b0, b1, b2]`
  I (index vector) = `[2, 4, 1]`
  X (input array) = `[a0, a1, a2, a3, a4, a5]`
  write I A X = `[a0, b2, b0, a3, b1, a5]`

- `split ::` $\text{Int} \rightarrow [\alpha] \rightarrow ([\alpha],[\alpha])$
  `split i` $[a_0,\ldots,a_n] \equiv ([a_0,\ldots,a_{i-1}],\ [a_i,\ldots,a_n])$
- `replicate ::` $\text{Int} \rightarrow \alpha \rightarrow [\alpha]$
  `replicate N a` $\equiv$ `[a, a,..., a]`, i.e., a is replicated `N` times.

Filter: Implementation based on Map and Scan

`filter ::` $(\alpha \rightarrow$`Bool`$) \rightarrow [\alpha] \rightarrow [\alpha]$

Filters out the input-list elements that do NOT satisfy the predicate.

Can `filter` be implemented via `map` and `reduce` (and `scan`)?

Filter: Implementation based on Map and Scan

`filter` :: $(\alpha \rightarrow$ `Bool`$) \rightarrow [\alpha] \rightarrow [\alpha]$

Filters out the input-list elements that do NOT satisfy the predicate.

Can `filter` be implemented via `map` and `reduce` (and `scan`)?

```
parFilter :: (a->Bool) -> [a] -> ([a], [Int])
parFilter cond X =
let n   = length arr
    cs  = map cond X
    tfs = map (\f->if f then 1
                        else 0) cs
    isT = scan^inc (+) 0 tfs
    i   = last isT

    ffs = map (\f->if f then 0
                        else 1) cs
    isF = (map (+ i) . scan^inc (+) 0) ffs

    inds= map (\ (c,iT,iF) ->
                  if c then iT-1 else iF-1 )
              (zip3 cs isT isF)
    flags = write [0,i] [i,n-i] (replicate n 0)
in  (permute X inds, flags)
```

```
Assume X = [5,4,2,3,7,8], and
cond is T(rue) for even nums.
n   = 6
cs  = [F, T, T, F, F, T]
tfs = [0, 1, 1, 0, 0, 1]

isT = [0, 1, 2, 2, 2, 3]
i   = 3

ffs = [1, 0, 0, 1, 1, 0]

isF = [4, 4, 4, 5, 6, 6]

inds= [3, 0, 1, 4, 5, 2]


flags  = [3, 0, 0, 3, 0, 0]
Result = [4, 2, 8, 5, 3, 7]
```

Segmented Inclusive Scan with Operator ⊕

Equiv with Mapping a Scan op on each segment of an irregular array.

```
-- iota n = [0..n-1]
map (\i-> scan (+) 0 [1..i]) [3,4] ≡
[ scan^inc (+) 0 [1,2,3],
  scan^inc (+) 0 [1,2,3,4] ]
      ≡
[ [1,3,6], [1,3,6,10] ]
```

```
-- Flags & Flat Data Representation:
sgmScanInc (+) 0 [1,0,0,1,0,0,0] -- flag
                 [1,2,3,1,2,3,4] -- data
    ≡
( [1,0,0,1,0,0, 0],              -- flag
  [1,3,6,1,3,6,10] )             -- data
```

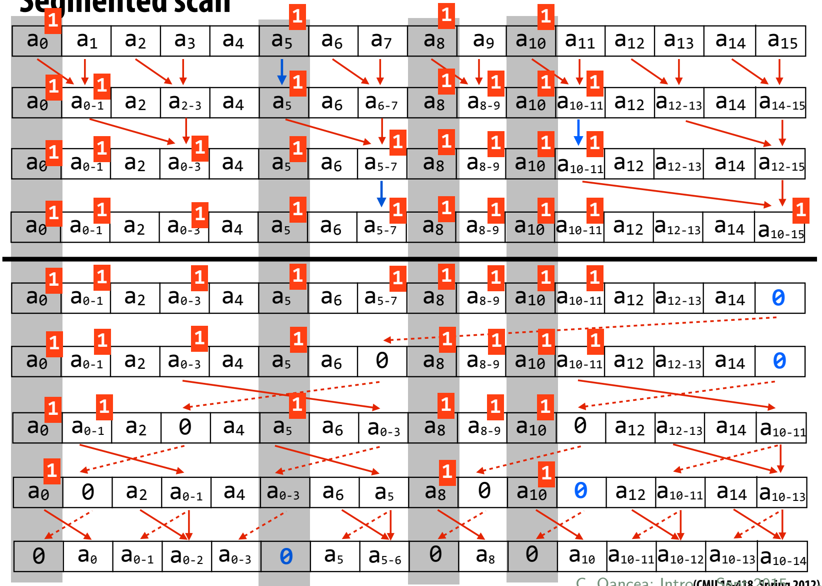
Can be obtained by replacing the following operator:

```
sgmScanInc :: (a->a->a) -> a -> [Int] -> [a] -> [a]
sgmScanInc ⊙ ne flags data =
  let fds = zip flags data
      (_,r) = unzip $
              scan^inc (\(f1,v1) (f2,v2) ->
                         let f = f1 .|. f2  -- bitwise or
                         in  if f2 == 0
                             then (f, v1 ⊙ v2)
                             else (f, v2)
                     ) (0,ne) fvs
  in  r
```

How about Exclusive Scan?

Slide from CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Segmented scan



(CMU 15-418, Spring 2012)

Segmented Exclusive Scan Alg And Complexity

```
Input:  flag array F of n=2^k of ints
        data array A of n=2^k elems of type T
        ⊕ :: T × T → T associative
Output: B = segmented scan of 2-dim array A
FORALL i = 0 to n-1 do B[i] ← A[i] ENDDO
FOR d = 0 to k-1 DO // up-sweep
  FORALL i = 0 to n-1 by 2^(d+1) DO
    IF F[i+2^(d+1)-1] == 0 THEN
        B[i+2^(d+1)-1] ← B[i+2^d-1] ⊕ B[i+2^(d+1)-1]
    ENDIF
    F[i+2^(d+1)-1] ← F[i+2^d-1] .|. F[i+2^(d+1)-1]
ENDDO ENDDO
B[n-1] ← 0
FOR d = k-1 downto 0 DO // down-sweep
  FORALL i = 0 to n-1 by 2^(d+1) DO
    tmp ← B[i+2^d-1]
    IF F_original[i+2^d] ≠ 0 THEN
        B[i+2^(d+1)-1] ← 0
    ELSE IF F[i+2^d-1] ≠ 0 THEN
        B[i+2^(d+1)-1] ← tmp
    ELSE B[i+2^(d+1)-1] ← tmp ⊕ B[i+2^(d+1)-1]
    ENDIF
    F[i+2^(d+1)-1] ← 0
ENDDO ENDDO
```

- While there are more branches, the asymptotics does not change:
- $D(n) = \Theta(lg\ n)$, $W(n) = \Theta(n)$!

Computing Prime Numbers: First Attempt

See also "Scan as Primitive Parallel Operation" [Bleelloch] (attached in TeachingMaterial/AdditionalMaterial/ListHom-Flattening).

Start with an array of size n filled intially with 1, i.e., all are primes, and iteratively zero out all multiples of numbers up to $\sqrt{n}$.

```
int res[n] = {0, 0, 1, 1, 1, ..., 1}
for(i = 2; i < sqrt(n); i++) {  //sequential
    if ( res[i] != 0 ) {
        forall m ∈ multiples of i ≤ n do {
            res[m] = 0;
        }
    }
}
```

Work: $O(n\ lg\ lg\ n)$ but Depth: $O(\sqrt{n})$ (Not Good Enough!)

Computing Prime Numbers: First Attempt

Start with an array of size n filled intially with 1, i.e., all are primes, and iteratively zero out all multiples of numbers up to $\sqrt{n}$.

```
primes :: Int -> [Int]
primes n =
  let a = map (\i -> if i==0 || i==1
                     then 0
                     else 1 ) [0..n]
      sqrtN = floor (sqrt (fromIntegral n))
  in  primesHelp 2 n sqrtN a
  where
    primesHelp :: Int -> Int -> Int
               -> [Int] -> [Int]
    primesHelp i a =
      if i > sqrtN then a
      else let m    = (n `div` i) - 1
               inds = map (\k -> (k+2)*i)
                          [0..m-1] --(iota m)
               vals = replicate m 0
               a'   = write inds vals a
           in  primesHelp (i+1) a'
```

```
Assume n = 9, sqrtN = 3
a = [0,0,1,1,1,1,1,1,1,1]

call primesHelp 2 a
m    = (9 `div` 2) - 1 = 3
inds = [4, 6, 8]
vals = [0, 0, 0]
a' = [0,0,1,1,0,1,0,1,0,1]

call primesHelp 3 a'
m    = (9 `div` 3) - 1 = 2
inds = [6, 9]
vals = [0, 0]
a''= [0,0,1,1,0,1,0,1,0,0]

call primesHelp 4 a''
result: [0,0,1,1,0,1,0,1,0,0]
  i.e., [0,1,2,3,4,5,6,7,8,9]
```

Work: $O(n\ lg\ lg\ n)$ but Depth: $O(\sqrt{n})$ (Not Good Enough!)

Computing Prime Numbers: Nested Parallelism

If we have all primes from 2 to $\sqrt{n}$ we could generate all multiples of these primes at once: `{[2*p:n:p]: p in sqr_primes}` in NESL.

Also call algorithm recursively on $\sqrt{n} \Rightarrow$ Depth: $O(lg\ lg\ n)$!

(solution of $n^{(1/2)^m} = 2$).

Computing Prime Numbers: Nested Parallelism

If we have all primes from 2 to $\sqrt{n}$ we could generate all multiples of these primes at once: `{[2*p:n:p]: p in sqr_primes}` in NESL.

Also call algorithm recursively on $\sqrt{n} \Rightarrow$ Depth: $O(lg\ lg\ n)$!
(solution of $n^{(1/2)^m} = 2$).

```
primesOpt :: Int -> [Int]
primesOpt n =
  if n <= 2 then [2]
  else
   let sqrtN = floor (sqrt (fromIntegral n))
       sqrt_primes = primesOpt sqrtN
       nested = map (\p->let m = (n `div` p)
                         in  map (\j-> j*p)
                                 [2..m]
                    ) sqrt_primes
       not_primes  = reduce (++) [] nested
       mm = length not_primes
       zeros = replicate mm False
       prime_flags= write not_primes zeros
                    (replicate (n+1) True)
       (primes,_)= unzip $ filter (\(i,f)->f)
                    $ (zip [0..n] prime_flags)
   in drop 2 primes
```

Computing Prime Numbers: Nested Parallelism

If we have all primes from 2 to $\sqrt{n}$ we could generate all multiples of these primes at once: `{[2*p:n:p]: p in sqr_primes}` in NESL.
Also call algorithm recursively on $\sqrt{n} \Rightarrow$ Depth: $O(lg\ lg\ n)$!
(solution of $n^{(1/2)^m} = 2$).

```
primesOpt :: Int -> [Int]
primesOpt n =
  if n <= 2 then [2]
  else
   let sqrtN = floor (sqrt (fromIntegral n))
       sqrt_primes = primesOpt sqrtN
       nested = map (\p->let m = (n `div` p)
                         in  map (\j-> j*p)
                                 [2..m]
                    ) sqrt_primes
       not_primes  = reduce (++) [] nested
       mm = length not_primes
       zeros = replicate mm False
       prime_flags= write not_primes zeros
                    (replicate (n+1) True)
       (primes,_)= unzip $ filter (\(i,f)->f)
                    $ (zip [0..n] prime_flags)
   in drop 2 primes
```

```
Assume n = 9, sqrtN = 3

call primesOpt 3
n = 3,sqrtN = 1,sqrt_primes=[2]
nested = [[]]; not_primes = []
mm = 0; zeros = []
prime_flags = [T,T,T,T]
primes = [0,1,2,3]; returns [2,3]

in primesOpt 9, afer
return from primesOpt3,
sqrt_primes = [2,3]
nested = [[4,6,8],[6,9]]
not_primes = [4,6,8,6,9]
mm=5;zeros= [F,F,F,F,F]
prime_flags= [T,T,T,T,F,T,F,T,F,F]
primes = [0,1,2,3,5,7]
returns [2,3,5,7]
```

Quicksort with Nested Parallelism

```
nestedQuicksort :: [a] -> [a]
nestedQuicksort arr =
  if (length arr) <= 1 then arr else
  let i = getRand (0, (length arr) - 1)
      a = arr !! i
      s1 = filter (\x -> (x <  a)) arr
      s2 = filter (\x -> (x >= a)) arr
      rs = map nestedQuicksort [s1, s2]
  in  (rs !! 0) ++ (rs !! 1)

-- Average Depth and Work ?
```

Quicksort with Nested Parallelism

```
nestedQuicksort :: [a] -> [a]
nestedQuicksort arr =
  if (length arr) <= 1 then arr else
  let i = getRand (0, (length arr) - 1)
      a = arr !! i
      s1 = filter (\x -> (x <  a)) arr
      s2 = filter (\x -> (x >= a)) arr
      rs = map nestedQuicksort [s1, s2]
  in  (rs !! 0) ++ (rs !! 1)

-- Average Depth and Work ?
```

```
Assume input array [3,2,4,1]
Assume random i = 0 ⇒ a = 3

s1 = [2,1]
s2 = [3,4]

nestedQuicksort [2,1]:
i = 0, a = 2
s1 = [1]
s2 = [2]
results in [1]++[2]==[1,2]

nestedQuicksort [3,4]: ...
results in [3,4]

After recursion concat:
[1,2] ++ [3,4] = [1,2,3,4]
```

Denoting by n the size of the input array: Average Work is $O(n\ lg\ N)$.

If filter would have depth 1, then Average Depth: $O(lg\ n)$.

In practice we have depth: $O(lg^2\ n)$.

1. Homomorphisms (Continuation)
   - Almost Homomorphisms Gorlatch'96
   - Scan as a Distributable Homomorphism
2. Implementation of Flat Bulk Operators
   - Implementation of Reduce and Scan
   - Other Second-Order Bulk Operators
   - Implementation of Segmented Scan
3. Nested Data-Parallel Applications
   - Sieve: Prime-Numbers Computation
   - Nested Parallel Quicksort
4. Flattening Nested Parallelism
   - Rules For Flattening
   - Flattening Prime-Number Computation
   - Flattening Quicksort

Nested *vs* Flattened Parallelism

Nested Arrays/Parallelism

```
-- 1. scan nested inside a map:
map (\row->scan^inc (+) 0 row)
    [[1,3], [2,4,6]] ≡
[ scan^inc (+) 0 [1,3],
  scan^inc (+) 0 [2,4,6] ] ≡

[ [ 1, 4], [2, 6, 12] ]

-- 2. map nested inside a map:
map ((\row->map f row))
    [[1,3], [2,4,6]] ≡
[ map f [1, 3],  map f [2, 4, 6] ] ≡
[ [f(1),f(3)], [f(2),f(4),f(6)] ]

-- 3. Distrib segment size to each elem [2, 2, 3, 3, 3] ≡
--E.g., flags = [2, 0, 3, 0, 0]
```

Flat Arrays/Parallelism

```
-- becomes a segmented scan
sgmScan^inc (+) 0
-- flags: ≠0 starts a new segment
    [2, 0, 3, 0, 0]
    [1, 3, 2, 4, 6] ≡
--  [ 2, 0, 3, 0, 0  ]
    [ 1, 4, 2, 6, 12 ]

-- becomes a map on the flat array
map f [1, 3, 2, 4, 6] ≡
[ f(1), f(3), f(2), f(4), f(6) ]
-- & the flag array is preserved:
-- [2, 0, 3, 0, 0]
```

Nested *vs* Flattened Parallelism

Nested Arrays/Parallelism

```
-- 1. scan nested inside a map:
map (\row->scan^inc (+) 0 row)
    [[1,3], [2,4,6]] ≡
[ scan^inc (+) 0 [1,3],
  scan^inc (+) 0 [2,4,6] ] ≡

[ [ 1, 4], [2, 6, 12] ]

-- 2. map nested inside a map:
map ((\row->map f row))
    [[1,3], [2,4,6]] ≡
[ map f [1, 3],  map f [2, 4, 6] ] ≡
[ [f(1),f(3)], [f(2),f(4),f(6)] ]

-- 3. Distrib segment size to each elem
--E.g., flags = [2, 0, 3, 0, 0]
```

Flat Arrays/Parallelism

```
-- becomes a segmented scan
sgmScan^inc (+) 0
-- flags: ≠0 starts a new segment
    [2, 0, 3, 0, 0]
    [1, 3, 2, 4, 6] ≡
--  [ 2, 0, 3, 0, 0  ]
    [ 1, 4, 2, 6, 12 ]

-- becomes a map on the flat array
map f [1, 3, 2, 4, 6] ≡
[ f(1), f(3), f(2), f(4), f(6) ]
-- & the flag array is preserved:
-- [2, 0, 3, 0, 0]

[2, 2, 3, 3, 3] ≡
    scan^inc (+) 0 flags flags
```

Nested *vs* Flattened Parallelism

Nested Arrays/Parallelism

```
-- 4. replicate nested inside a map:
let arr = [1,3,2] in
map (\n -> replicate(n, n)) arr
≡
[ replicate(1,1)
, replicate(3,3)
, replicate(2,2) ]
≡
[ [1], [3,3,3], [2,2] ]

-- 7. if nested inside a filter
let arr = [3, 4, 6, 7] in
map(\x ->
        if (odd x)
        then f x  -- assume f is *2
        else g x  -- assume g is -1
   ) arr
-- ⇒ [6, 3, 5, 14]

-- 6. filter nested inside a map:
map(\row -> filter p row)
```

Flat Arrays/Parallelism

```
--build the flag array and sgmScan it!
inds = sgmScan^exc (+) 0 arr
size = (last inds) + (last arr)
flag = write arr   -- [1,3,2]
             inds  -- [0,1,4]
             replicate(size, 0)
--           [1,3,0,0,2,0]
sgmScan^inc (+) 0 flag
-- [1,3,3,3,2,2]

-- scatter o map o gather
ais = zip arr (iota (length arr))
(ais',flg)=parFilter(\(x,_)->p e) ais
(ais^t,ais^f) = split flg[0] ais'
(arr^t,inds^t) = unzip ais^t
(arr^f,inds^f) = unzip ais^f
(arr^then,arr^else) = (map f arr^t, map g arr^f)
restmp = write inds^t arr^then [0,...,0]
result = write inds^f arr^else restmp
-- write filter in terms of map/scan
-- the flatten each operation!
```

How to Flatten? A Relatively Simple Case

```
map (\i -> map (+1) [0..i]) [0..n-1]
```

Any difference between [0..n-1] and [0..i]? How does one write [0..i]?

How to Flatten? A Relatively Simple Case

```
map (\i -> map (+1) [0..i]) [0..n-1]
```

Any difference between [0..n-1] and [0..i]? How does one write [0..i]?

```
map (\i -> let ip1 = i + 1
               tmp1 = replicate ip1 1
               tmp2 = scan^exc (+) 0 tmp1
           in        map (+1) tmp2     )
    [0..n-1]
```

How to Flatten? A Relatively Simple Case

```
map (\i -> let ip1  = i + 1
               tmp1 = replicate ip1 1
               tmp2 = scan^exc (+) 0 tmp1
           in         map (+1) tmp2      ) [0..n-1]
```

Assume N = 4. Expected Result:
[[1],[1,2],[1,2,3],[1,2,3,4]]

How to Flatten? A Relatively Simple Case

```
map (\i -> let ip1  = i + 1
               tmp1 = replicate ip1 1
               tmp2 = scan^exc (+) 0 tmp1
           in       map (+1) tmp2      ) [0..n-1]
```

Assume N = 4. Expected Result:
[[1],[1,2],[1,2,3],[1,2,3,4]]

1. Size of row i is i+1, hence array's shape = map (+1) [0..n-1], i.e., shape = [1,2,3,4]
2. Result Array # of all elements: flat_size = $\sum_{i=0}^{n-1}(i+1)$ = 10!
3. start index of each segment:
   segm_beg=scan^{exc} (+) 0 shape
   = [0,1,3,6]
   shape = [1,2,3,4]
4. write ind-val pairs into zero array
   sizes = [1,2,0,3,0,0,4,0,0,0]
   flags = [1,1,0,1,0,0,1,0,0,0]
5. Distributing map across each stmt:
6. tmp1^{flat} = map (replicate ip1 1) =
   segmScan^{inc} (+) 0 sizes flags
   = [1,1,1,1,1,1,1,1,1,1]
7. tmp2^{flat} = sgmScan^{exc} (+) 0 flags tmp1^{flat}
   = [0,0,1,0,1,2,0,1,2,3]
8. add 1 to all elements:
   vals = map (+1) tmp2^{flat}
   = [1,1,2,1,2,3,1,2,3,4]
   & sizes [1,2,0,3,0,0,4,0,0,0]
9. What if I want to add i instead of 1?

How to Flatten? A Relatively Simple Case

```
map (\i -> let ip1  = i + 1
               tmp1 = replicate ip1 1
               tmp2 = scan^exc (+) 0 tmp1
           in         map (+1) tmp2      ) [0..n-1]
```

Assume N = 4. Expected Result:
[[1],[1,2],[1,2,3],[1,2,3,4]]

1. Size of row i is i+1, hence
   array's shape = map (+1) [0..n-1],
   i.e., shape = [1,2,3,4]
2. Result Array # of all elements:
   flat_size = $\sum_{i=0}^{n-1}(i+1)$ = 10!
3. start index of each segment:
   segm_beg=scan^{exc} (+) 0 shape
   = [0,1,3,6]
   shape = [1,2,3,4]
4. write ind-val pairs into zero array
   sizes = [1,2,0,3,0,0,4,0,0,0]
   flags = [1,1,0,1,0,0,1,0,0,0]
5. Distributing map across each stmt:
6. tmp1^{flat} = map (replicate ip1 1) =
   segmScan^{inc} (+) 0 sizes flags
   = [1,1,1,1,1,1,1,1,1,1]
7. tmp2^{flat} = sgmScan^{exc} (+) 0 flags tmp1^{flat}
   = [0,0,1,0,1,2,0,1,2,3]
8. add 1 to all elements:
   vals = map (+1) tmp2^{flat}
   = [1,1,2,1,2,3,1,2,3,4]
   & sizes [1,2,0,3,0,0,4,0,0,0]
9. What if I want to add i instead of 1?

```
iis = write segm_beg [0..n-1] zeros
            [0,1,3,6] [0,1,2,3]
            [0,0,0,0,0,0,0,0,0,0]
          ------------------------------
            [0,1,0,2,0,0,3,0,0,0]
diis= sgmScan^inc (+) 0 flags iis
          [1,1,0,1,0,0,1,0,0,0]
          [0,1,0,2,0,0,3,0,0,0]
         -------------------------------
        = [0,1,1,2,2,2,3,3,3,3]
vals = map (+i) vals0
     = zipWith (+) diis vals0
     = [0,1,2,2,3,4,3,4,5,6]
```

10. Flaten op would get rid of the flags

How Does One Flattens Prime Numbers?

```
nested = map (\p->let m = (9 `div` p)
                  in  map (\j-> j*p)
                          [2..m]
             ) [2,3]
not_primes  = reduce (++) [] nested
```

```
n = 9, sqrtN = 3.
mult_lens denotes # of multi-
  ples of a given prime upto n.
mult_lens = [4-1,3-1]=[3,2] i.e.,
                excludes the prime*1
mult_scan = [0,3] via scan^exc
mult_tot_len = 5, total # of multiples
flags      = [1, 0, 0, 1, 0],
  the segments of the array
        of prime multiples
ps         = [2, 0, 0, 3, 0]
  each segments has its prime
p_vals     = [2, 2, 2, 3, 3]
              *  *  *  *  *
p_inds     = [2, 3, 4, 2, 3]
              =  =  =  =  =
not_primes= [4, 6, 8, 6, 9]
zeros     = [F, F, F, F, F]
prime_flg=[T,T,T,T,F,T,F,T,F,F]
  transformed to prime indexes
primes    =[0,1,2,3,  5,  7    ]
Result is:[    2,3,  5,  7]
```

Prime Numbers: Flattening Nested Parallelism

```
primesFlat :: Int -> [Int]
primesFlat n = if n <= 2 then [2] else
 let sqrtN = floor (sqrt (fromIntegral n))
     sqrt_primes = primesFlat sqrtN
     mult_lens   = map (\p->(n `div` p)-1)
                       sqrt_primes
     mult_scan   = scanExc (+) 0 mult_lens
     mult_tot_len= (last mult_scan) +
                   (last mult_lens)
     flags = write mult_scan
          (replicate (length sqrt_primes) 1)
          (replicate mult_tot_len 0)
     ps    = write mult_scan sqrt_primes
                  (replicate mult_tot_len 0)
     p_vals= segmScanInc (+) 0 flags ps
     p_inds= map (+1) $ segmScanInc (+) 0 flags
                  (replicate mult_tot_len 1)
     not_primes= zipWith (*) p_inds p_vals
     zeros = replicate (length p_inds) False
     prime_flg  = write not_primes zeros
                  (replicate (n+1) True)
     (primes,_) = unzip $ filter (\(i,f)->f))
                   $ zip [0..n] prime_flags
 in drop 2 primes
```

```
Assume n = 9, sqrtN = 3, and
that (primesOpt 3) results in
sqrt_primes = [2,3].
mult_lens denotes # of multi-
  ples of a given prime upto n.
mult_lens = [3,2]
mult_scan = [0,3] start indices
mult_tot_len = 5, total # of
  prime multiples.
flags      = [1, 0, 0, 1, 0],
  the segments of the array
        of prime multiples
ps         = [2, 0, 0, 3, 0]
  each segments has its prime
p_vals     = [2, 2, 2, 3, 3]
               *  *  *  *  *
p_inds     = [2, 3, 4, 2, 3]
               =  =  =  =  =
not_primes = [4, 6, 8, 6, 9]
zeros     = [F, F, F, F, F]
prime_flg=[T,T,T,T,F,T,F,T,F,F]
  transformed to prime indexes
primes    =[0,1,2,3,  5,  7  ]
```

Result is: [2, 3, 5, 7]

How to Flatten Quicksort

```
nestedQuicksort :: [a] -> [a]
nestedQuicksort arr =
  if (length arr) <= 1 then arr else
  let i = getRand (0, (length arr) - 1)
      a = arr !! i
      s1 = filter (\x -> (x <  a)) arr
      s2 = filter (\x -> (x >= a)) arr
      rs = map nestedQuicksort [s1, s2]
  in  (rs !! 0) ++ (rs !! 1)
```

Key Idea is to write a function that has the semantics of:

`map nestedQuicksort`

by distributing the map against the bindings ⇒
it operates on arrays of arrays!

`quicksort`^{lift} `:: [[a]] -> [[a]]` or flattened

`flatQuicksort`^{lift} `:: [Int] -> [a] -> [a]`,
where the first arg are the flags and the second the flat data.

For example, we will have an array of `i`s, an array of `a`s, of `s1`s, etc.

Quicksort: Flattened Nested Parallelism

```
flatQuicksort :: a -> [Int] -> [a] -> [a]
flatQuicksort ne sizes arr =
  if reduce (&&) True $
        map (\s->(s<2)) sizes
  then arr else
  let si = scanInc (+) 0 sizes
      r_inds=
        map (\(l,u,s)->
                if s<1 then ne else
                     arr !! (getRand (l,u-1))
            )  (zip3 (0:si) si sizes)
      rands = segmScan^inc (+) ne sizes r_inds

      (sizes',arr_rands) = segmSpecialFilter
                  (\(r,x)->(x < r))
                  sizes (zip rands arr)
      (_,arr') = unzip arr_rands
  in  flatQuicksort ne sizes' arr'
```

Quicksort: Flattened Nested Parallelism

```
flatQuicksort :: a -> [Int] -> [a] -> [a]
flatQuicksort ne sizes arr =
  if reduce (&&) True $
         map (\s->(s<2)) sizes
  then arr else
  let si = scanInc (+) 0 sizes
      r_inds=
        map (\(l,u,s)->
                if s<1 then ne else
                     arr !! (getRand (l,u-1))
            )  (zip3 (0:si) si sizes)
      rands = segmScan^inc (+) ne sizes r_inds

      (sizes',arr_rands) = segmSpecialFilter
                     (\(r,x)->(x < r))
                     sizes (zip rands arr)
      (_,arr') = unzip arr_rands
  in  flatQuicksort ne sizes' arr'
```

```
Key idea: use a 2D array:
Input: ne = 0, sizes = [4,0,0,0],
               arr   = [3,2,4,1]
Condition does not hold
since one size is 4 > 2
si = [4,4,4,4] distrib inner sizes
Compute the indexes of random
   nums, one for each segment
r_sparse= [a!!0,0,0,0]=[3,0,0,0]
& distrib it across each segm
rands    = [3,3,3,3]
For 1D case this is ≡
 with our parFilter. Generalize it
[4,0,0,0], [(3,3),(3,2),(3,4),(3,1
sizes' =    [2,    0,    2,    0],
arr_rands= [(3,2),(3,1),(3,3),(3,4
arr'      = [   2,    1,    3,    4
Recursively with the new array & s
```

`segmSpecialFilter::(a->Bool)->[Int]->[a]->([Int],[a])`

Intuitive Semantics: `segmSpecialFilter odd [2,0,2,0] [4,1,3,3]` ⇒ `([1,1,2,0],[1,4,3,3])`