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

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 a Simple (Contrived) Program
   - Flattening Prime-Number (Sieve) Computation
   - Flattening Quicksort

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])` $\equiv 7 \uparrow 4 \equiv 7$
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 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 a Simple (Contrived) Program
   - Flattening Prime-Number (Sieve) 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` :: $[\alpha] \to [\beta] \to [\gamma]$,
  `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*

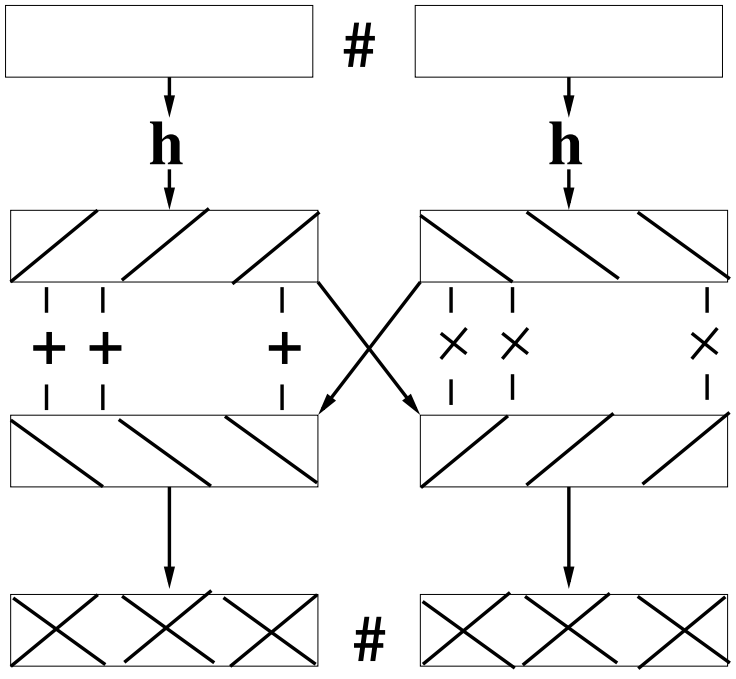
`dhop` :: *[a]* $\to$ *[a]* $\to$ *[a]*
`dhop` *u v* = *(zipWith* $\oplus$ *u v) ++ (zipWith* $\otimes$ *u v)*

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)* ‘`dhop`‘ *(*$\oplus \updownarrow \otimes$ *y)*

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

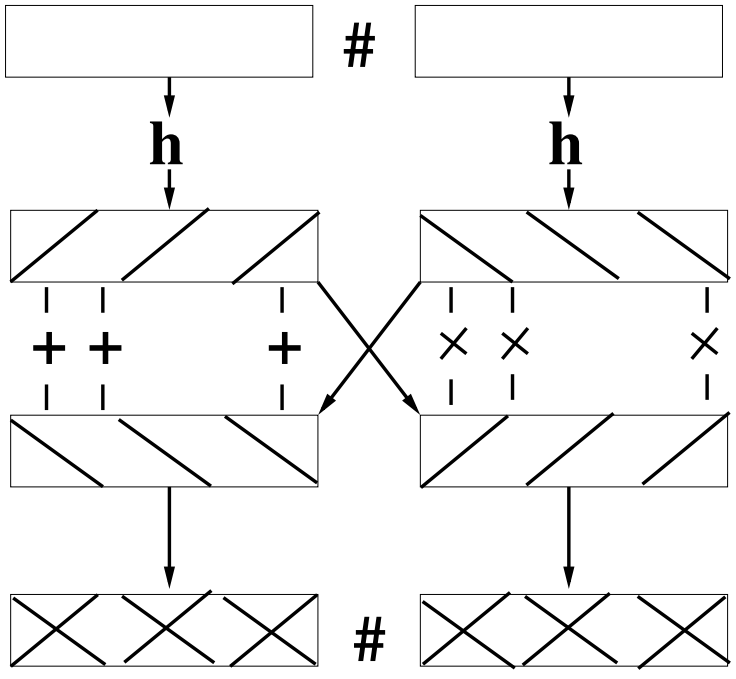
Distributed Reduce is a DH



- `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 $\odot = \odot \updownarrow \odot$

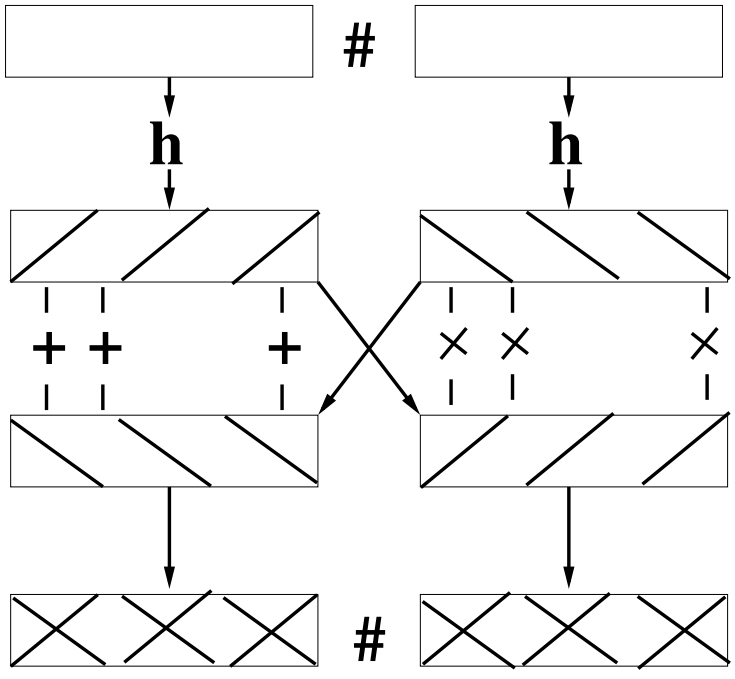
Scan is a DH



- `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` $\odot$ $e_\odot$ `(x ++ y)` $= S_1 \oslash S_2 =$ `S1 ++ (map (`$\odot$ `s) S2)`, 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)`, $(s_1,r_1) \oplus (s_2,r_2) =$ `(?, ?)` $(s_1,r_1) \otimes (s_2,r_2) =$ `(?, ?)`

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)` $= S_1 \oslash S_2 =$ `S1 ++ (map (`$\odot$ `s) S2)`, where S_1= `scan` $\odot$ $e_\odot$ `x`, S_2= `scan` $\odot$ $e_\odot$ `y`, and `s=last S1`
- `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

- `map` :: $(\alpha \to \beta) \to [\alpha] \to [\beta]$
  `map f` $[x_1, \ldots, x_n]$ = `[f(`x_1`),...,f(`x_n`)]`,
  i.e., $x_i :: \alpha, \forall i$, and `f` :: $\alpha \to \beta$.

- `reduce` :: $(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$
  `reduce` $\odot$ `e` $[x_1, x_2, .., x_n]$ = `e`$\odot x_1 \odot x_2 \odot \ldots \odot x_n$,
  i.e., `e::`α, $x_i :: \alpha, \forall i$, and $\odot :: \alpha \to \alpha \to \alpha$.

- $\texttt{scan}^{exc}$ :: $(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]$
  $\texttt{scan}^{exc}$ $\odot$ `e` $[x_1, \ldots, x_n]$ = `[e,e`$\odot x_1, \ldots, $`e`$\odot x_1 \odot \ldots x_{n-1}$`]`
  i.e., `e::`α, $x_i :: \alpha, \forall i$, and $\odot :: \alpha \to \alpha \to \alpha$.

- $\texttt{scan}^{inc}$ :: $(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]$
  $\texttt{scan}^{inc}$ $\odot$ `e` $[x_1, \ldots, x_n]$ = `[e`$\odot x_1, \ldots, $`e`$\odot x_1 \odot \ldots x_n$`]`
  i.e., `e::`α, $x_i :: \alpha, \forall i$, and $\odot :: \alpha \to \alpha \to \alpha$.

Parallel Random Access Machine (PRAM)

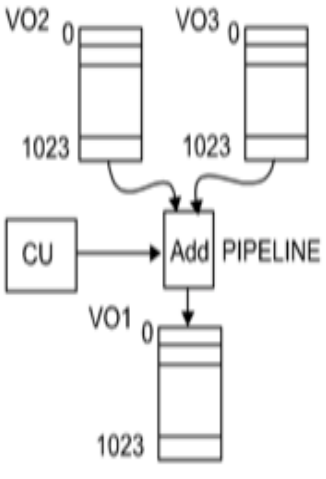
PRAM focuses exclusively on parallelism and ignores issues related to synchronization and communication:

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.

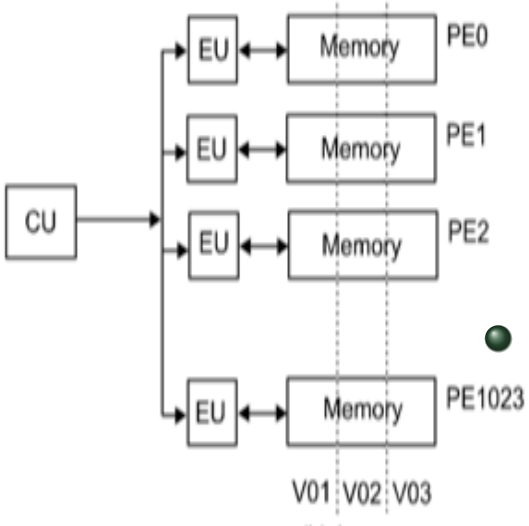
Parallel Random Access Machine (PRAM)

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.



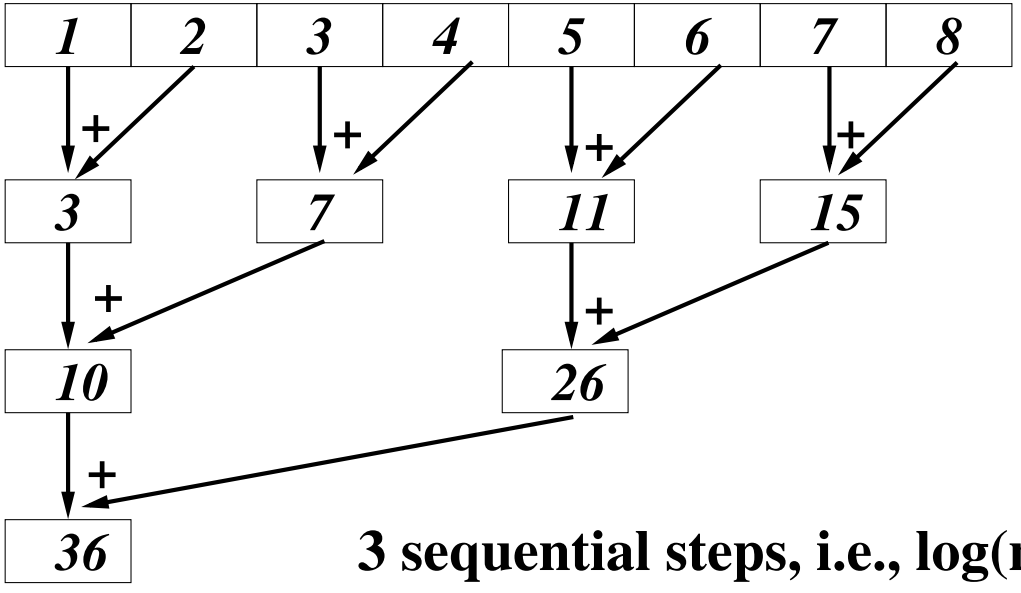
(a) Vector



(b) Array

- 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

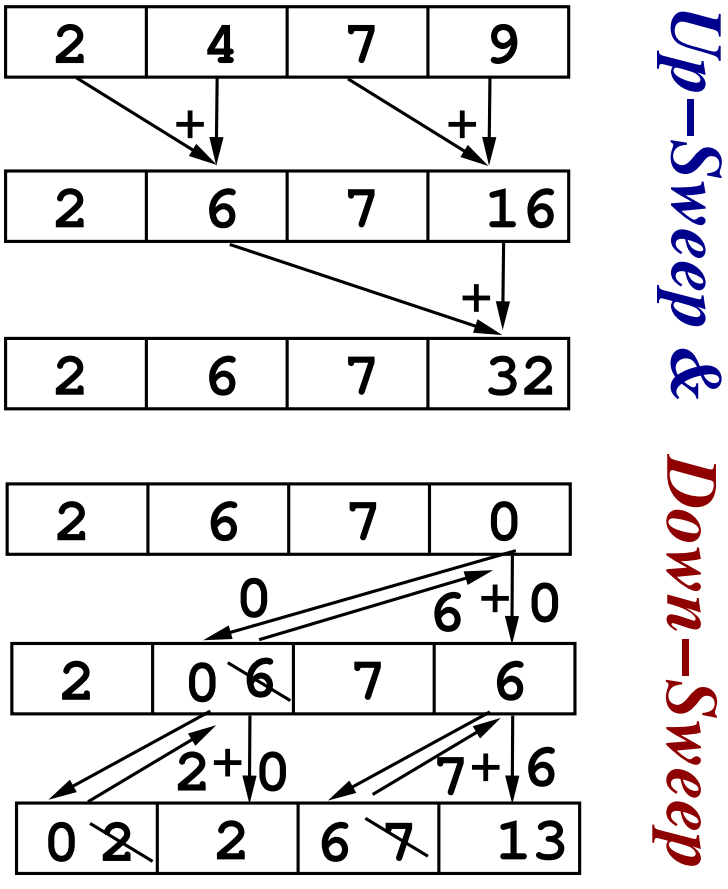
```
Input:  array A of n=2^k elems of type T
        ⊕ :: T × T → T associative
Output: S = ⊕_{j=1}^{n} a_j

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 $\oplus$



Up–Sweep & Down–Sweep

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 = $[0, a_1, a_1 \oplus a_2, \ldots, \oplus_{j=1}^{n-1} a_j]$

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

4.  for d = 0 to k-1 do // up-sweep
5.    forall i = 0 to n-1 by 2^(d+1) do
6.      B[i+2^(d+1)-1] ← B[i+2^d -1] ⊕
                         B[i+2^(d+1)-1]
7.    enddo
8.  enddo
9.  B[n-1] = 0
10. for d = k-1 downto 0 do // down-sweep
11.   forall i = 0 to n-1 by 2^(d+1) do
12.     tmp ← B[i+2^d-1]
13.     B[i+2^d-1] ← B[i+2^(d+1)-1]
14.     B[i+2^(d+1)-1] ← tmp ⊕ B[i+2^(d+1)-1]
15.   enddo
16. 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 :: [`α_1`] → [`α_2`] → [(`α_1`,`α_2`)]`
- `zip [a`$_1$`,...,a`$_n$`] [b`$_1$`,...,b`$_m$`]` $\equiv$ `[(a`$_1$`,b`$_1$`),...,(a`$_q$`,b`$_q$`)]`, where `q = min(m,n)`.

Zip, ZipWith

- `zip :: [`α_1`] → [`α_2`] → [(`α_1`,`α_2`)]`
- `zip [a`$_1$`,...,a`$_n$`] [b`$_1$`,...,b`$_m$`]` $\equiv$ `[(a`$_1$`,b`$_1$`),...,(a`$_q$`,b`$_q$`)]`, where `q = min(m,n)`.
- `unzip :: [(`α_1`,`α_2`)] → ([`α_1`],[`α_2`])`
- `unzip [(a`$_1$`,b`$_1$`),...,(a`$_n$`,b`$_n$`)]`$\equiv$`([a`$_1$`,...,a`$_n$`],[b`$_1$`,...,b`$_n$`])`,

Zip, ZipWith

- `zip :: [`α_1`] → [`α_2`] → [(`α_1`,`α_2`)]`
- `zip [a1,...,an] [b1,...,bm]` $\equiv$ `[(a1,b1),...,(aq,bq)]`, where `q = min(m,n)`.
- `unzip :: [(`α_1`,`α_2`)] → ([`α_1`],[`α_2`])`
- `unzip [(a1,b1),...,(an,bn)]`$\equiv$`([a1,...,an],[b1,...,bn])`,
- In some sense `zip` is syntactic sugar, for example one could work with the tuple of array representation, e.g.,
- `mapT :: ((`α_1`,...,`α_m`)→(`β_1`,...,`β_n`)) → [`α_1`] → ... → [`α_m`] → ([`β_1`],...,[`β_n`]])`
- `mapT f` $\equiv$ `unzip`^{n} `. map f .` `zip`^{n}

Zip, ZipWith

- `zip :: [`α_1`] → [`α_2`] → [(`α_1`,`α_2`)]`
- `zip [a`$_1$`,...,a`$_n$`] [b`$_1$`,...,b`$_m$`]` $\equiv$ `[(a`$_1$`,b`$_1$`),...,(a`$_q$`,b`$_q$`)]`, where `q = min(m,n)`.
- `unzip :: [(`α_1`,`α_2`)] → ([`α_1`],[`α_2`])`
- `unzip [(a`$_1$`,b`$_1$`),...,(a`$_n$`,b`$_n$`)]`$\equiv$`([a`$_1$`,...,a`$_n$`],[b`$_1$`,...,b`$_n$`])`,
- In some sense `zip` is syntactic sugar, for example one could work with the tuple of array representation, e.g.,
- `mapT :: ((`α_1`,...,`α_m`)→(`β_1`,...,`β_n`)) →`
  `[`α_1`] → ... → [`α_m`] → ([`β_1`],...,[`β_n`])`
- `mapT f` $\equiv$ `unzip`^{n} `. map f . zip`^{n}
- `zipWith :: (`$\alpha_1 \to \alpha_2 \to \beta$`) → [`$\alpha_1$`] → [`$\alpha_2$`] → [`$\beta$`]`
- `zipWith` $\odot$ `[a`$_1$`,...,a`$_n$`] [b`$_1$`,...,b`$_n$`]` $\equiv$ `[a`$_1$$\odot$`b`$_1$`,...,a`$_n$$\odot$`b`$_n$`]`
- `zipWith` $\odot$ $\equiv$ `map (\(u,v) → u` $\odot$ `v) . zip`

Permute, Write, Split, Filter

- Operator to permute in parallel based on a set (array) of indices:
  permute :: [Int] $\rightarrow [\alpha] \rightarrow [\alpha]$, 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] $\rightarrow [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$
  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 :: 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 :: 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`$) \rightarrow$ `[`α`]` $\rightarrow$ `[`α`]`

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`$) \rightarrow$ `[`α`]` $\rightarrow$ `[`α`]`

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 $\oplus$

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] ]
```

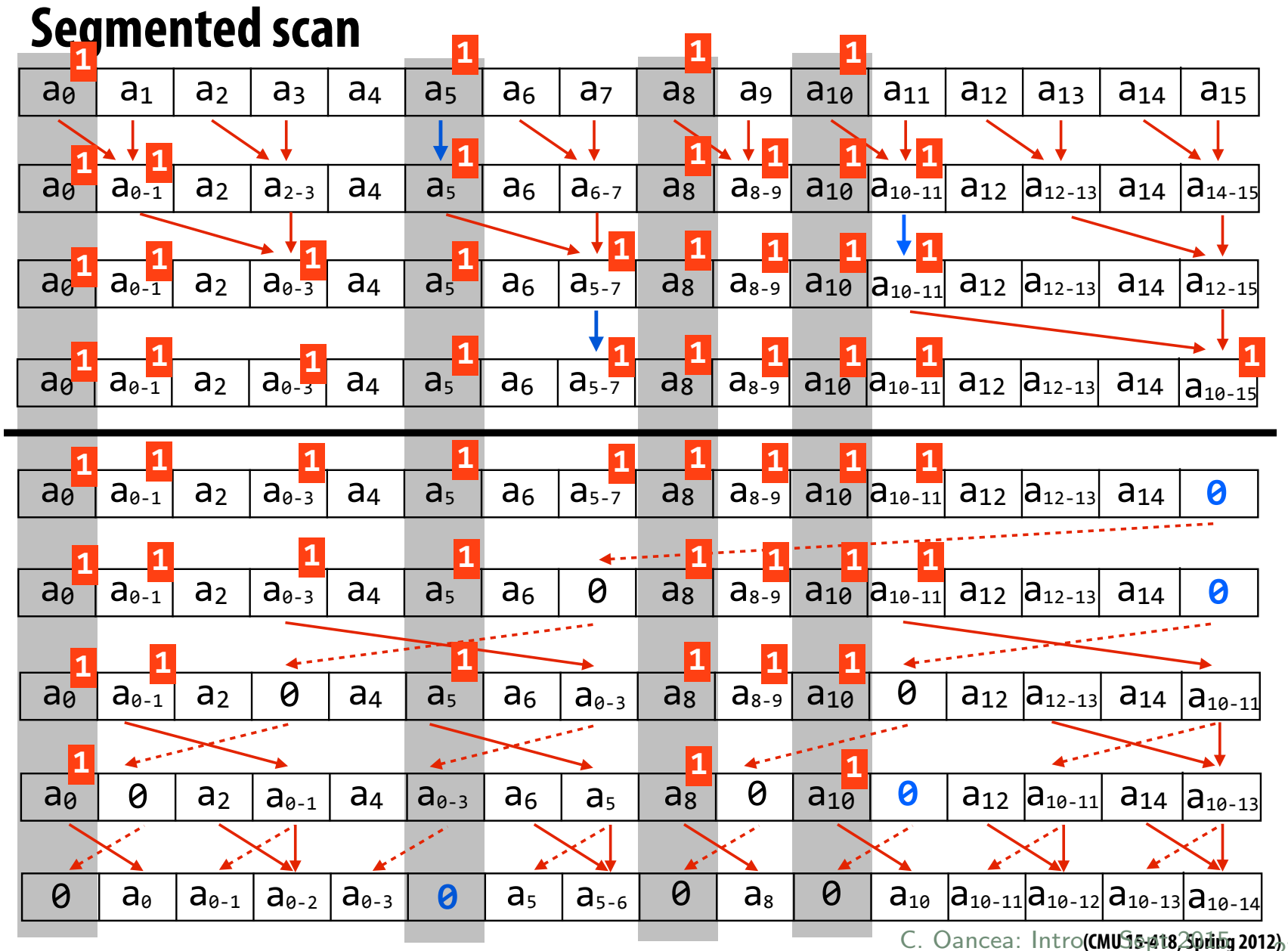
```
-- Flags & Flat Data Representation:
sgmScanInc (+) 0 [1,0,0,1,0,0,0] -- flag
                 [1,2,3,1,2,3,4] -- data
    ≡
( [1,0,0,1,0,0, 0],              -- flag
  [1,3,6,1,3,6,10] )             -- data
```

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 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
1.  FORALL i = 0 to n-1 do B[i] ← A[i] ENDDO
2.  FOR d = 0 to k-1 DO // up-sweep
3.    FORALL i = 0 to n-1 by 2^(d+1) DO
4.      IF F[i+2^(d+1)-1] == 0 THEN
5.          B[i+2^(d+1)-1] ← B[i+2^d-1] ⊕ B[i+2^(d+1)-1]
6.      ENDIF
7.      F[i+2^(d+1)-1] ← F[i+2^d-1] .|. F[i+2^(d+1)-1]
8.  ENDDO ENDDO
9.  B[n-1] ← 0
10. FOR d = k-1 downto 0 DO // down-sweep
11.   FORALL i = 0 to n-1 by 2^(d+1) DO
12.     tmp ← B[i+2^d-1]
13.     IF F_original[i+2^d] ≠ 0 THEN
14.          B[i+2^(d+1)-1] ← 0
15.     ELSE IF F[i+2^d-1] ≠ 0 THEN
16.          B[i+2^(d+1)-1] ← tmp
17.     ELSE B[i+2^(d+1)-1] ← tmp ⊕ B[i+2^(d+1)-1]
18.     ENDIF
19.     F[i+2^(d+1)-1] ← 0
20. 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 a Simple (Contrived) Program
   - Flattening Prime-Number (Sieve) Computation
   - Flattening Quicksort

Nested *vs* Flattened Parallelism: Scan inside a Map

(1) Scan inside a nested 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] ]
```

Nested *vs* Flattened Parallelism: Scan inside a Map

(1) Scan inside a nested 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] ]
```

becomes a segmented scan, which requires a flag array as arg:

```
sgmScan^inc (+) 0 [2, 0, 3, 0, 0] [1, 3, 2, 4, 6] ≡ [ 1, 4, 2, 6, 12 ]
```

The flag array [2, 0, 3, 0, 0] encodes the fact that the flat-data array [1, 3, 2, 4, 6] has two segments:

- one of length 2 starting at index 0
- one of length 3 starting at index 2

(i.e., an non-zero element in the flag array denotes the length of the segment that start at that point.)

Nested *vs* Flattened Parallelism: Map inside a Map

(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)] ]
```

Nested *vs* Flattened Parallelism: Map inside a Map

(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)] ]
```

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 assumed known and is preserved [2, 0, 3, 0, 0]

How To Distribute the Segment Size?

Assume flag array: `[2, 0, 3, 0, 0]`.

How do we get `[2, 2, 3, 3, 3]`?

How To Distribute the Segment Size?

Assume flag array: `[2, 0, 3, 0, 0]`.

How do we get `[2, 2, 3, 3, 3]`?

$\texttt{scan}^{inc}$ `(+) 0 flags flags`

Nested *vs* Flattened Parallelism: Replicate in a Map

(4) Replicate nested inside a map:

```
map (\(n,m) -> replicate n m) [(1,7),(3,8),(2,9)] ≡
[ replicate 1 7, replicate 3 8, replicate 2 9 ] ≡
[ [7], [8,8,8], [9,9] ]
```

Nested *vs* Flattened Parallelism: Replicate in a Map

(4) Replicate nested inside a map:

```
map (\(n,m) -> replicate n m) [(1,7),(3,8),(2,9)] ≡
[ replicate 1 7, replicate 3 8, replicate 2 9 ] ≡
[ [7], [8,8,8], [9,9] ]
```

becomes a composition of scans and write:

```
1. (ns, ms)  = unzip([(1,7), (3,8), (2,9)]
2. inds = sgmScan^exc (+) 0 ns                    -- [0,1,4]
3. size = (last inds) + (last ns)                  -- 4 + 2 = 6
4. flag = write inds  ns  (replicate size 0)       -- [1, 3, 0, 0, 2, 0]
5. vals = write inds  ms  (replicate size 0)       -- [7, 8, 0, 0, 9, 0]
6. sgmScan^inc (+) 0 flag vals                     -- [7,8,8,8,9,9]
```

2. builds the indices at which segment start
3. get the size of the flat array (equivalent to summing `arr`)

4-5. write the array elems at the position where a segment starts

6. distribute the start-elem of a segment throughout the segment.

Nested *vs* Flattened Parallelism: Iota in a Map

(5) Iota nested inside a map ((iota n)≡[0,...,n-1]):

```
map (\i -> iota i) [1,3,2] ≡
[ iota 1, iota 3, iota 2 ] ≡ [ [0], [0,1,2], [0,1] ]
```

Nested *vs* Flattened Parallelism: Iota in a Map

(5) Iota nested inside a map ((iota n)≡[0,...,n-1]):

```
map (\i -> iota i) [1,3,2] ≡
[ iota 1, iota 3, iota 2 ] ≡ [ [0], [0,1,2], [0,1] ]
```

becomes a composition of scans and write:

```
1. arr  = [1, 3, 2]
2. inds = sgmScan^exc (+) 0 arr       -- [0,1,4]
3. size = (last inds) + (last arr) -- 4 + 2 = 6
4. flag = write inds  -- [0,1,4]
                arr   -- [1,3,2]
                (replicate size 0)
--              [1, 3, 0, 0, 2, 0]
5. tmp  = replicate size 1  --[1, 1, 1, 1, 1, 1]
6. sgmScan^exc (+) 0 flag tmp --[0, 0, 1, 2, 0, 1]
```

2. builds the indices at which segment start
3. get the size of the flat array (equivalent to summing `arr`)
4. write the array elems at the position where a segment starts
6. segmented scan an array of ones.

Nested *vs* Flattened Parallelism: If Inside a Map

(6) An If-Then-Elese nested inside a map:

```
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
-- should result in [6, 3, 5, 14]
```

Nested *vs* Flattened Parallelism: If Inside a Map

(6) An If-Then-Elese nested inside a map:

```
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
-- should result in [6, 3, 5, 14]
```

translates to a scatter-map-gather composition:

```
1. ais = zip arr (iota (length arr))       -- [(3,0), (4,1), (6,2), (7,3)]
2. (ais',flg)=parFilter(\(x,_)->odd x) ais--([(3,0),(7,3),(4,1),(6,2)],[2,0,2,0])
3. (ais^t,ais^f) = split flg[0] ais'        --([(3,0),(7,3)], [(4,1),(6,2)])
4. (arr^t,inds^t) = unzip ais^t               --([3,7], [0,3])
5. (arr^f,inds^f) = unzip ais^f               --([4,6], [1,2])
6. (arr^then,arr^else) = (map f arr^t, map g arr^f) --([6,14], [3,5])
7. result = write inds^f arr^else (write inds^t arr^then [0,...,0]) --[6, 3, 5, 14]
```

1-2. zip array with indices and permute based on if predicate,

3-5. unzip the array segments and indices,

6. map the two data arrays

7. write back the resulted elements at original positions.

Nested *vs* Flattened Parallelism: Filter inside a Map

Rule 7: Assignment I, Task 3!

How to Flatten? A Relatively Simple Case

```
let arr = [1, 2, 3, 4] in
map (\i -> map (+(i+1)) (iota i)) arr
-- Result: [[2],[3,4],[4,5,6],[5,6,7,8]]
```

How to Flatten? A Relatively Simple Case

```
let arr = [1, 2, 3, 4] in
map (\i -> map (+(i+1)) (iota i)) arr
-- Result: [[2],[3,4],[4,5,6],[5,6,7,8]]
```

Normalize the code:

```
map (\i -> let ip1 = i+1 in
           let iot = (iota i) in
           let ip1r= (replicate i ip1) in
           map (+) (zip iot ip1r)            ) arr
```

Distribute the map over every instruction in the body
(bottom-up if nest > 2), where $\mathcal{F}$ denotes the flattening transf:

How to Flatten? A Relatively Simple Case

```
let arr = [1, 2, 3, 4] in
map (\i -> map (+(i+1)) (iota i)) arr
-- Result: [[2],[3,4],[4,5,6],[5,6,7,8]]
```

Normalize the code:

```
map (\i -> let ip1 = i+1 in
           let iot = (iota i) in
           let ip1r= (replicate i ip1) in
           map (+) (zip iot ip1r)            ) arr
```

Distribute the map over every instruction in the body
(bottom-up if nest > 2), where $\mathcal{F}$ denotes the flattening transf:

```
F(map (\i -> map (+(i+1)) (iota i)) [0..n-1]) ≡
1. let ip1s = map (\i -> i+1) arr in -- [2, 3, 4, 5]
2. let iots = F(map (\i -> (iota i)) arr)) in
3. let ip1rs= F(map (\(i,ip1) -> (replicate i ip1)) (zip arr ip1s)))
4. in  F(map (\(z) -> map (+) z) (zip ip1rs iots))
```

How to Flatten? A Relatively Simple Case

According to rule (4) iota nested inside a map

(assuming `arr = [1,2,3,4]`):

```
2. let iots = F(map (\i -> iota(i)) arr)

≡

inds = sgmScan^exc (+) 0 arr -- [0,1,3,6]
size = (last inds) + (last arr) -- 6 + 4 = 10
flag = write inds arr
             (replicate size 0)
--           [1, 2, 0, 3, 0, 0, 4, 0, 0, 0]
tmp  = replicate(size, 1)
iots = sgmScan^exc (+) 0 flag tmp --[0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

How to Flatten? A Relatively Simple Case

According to rule (5) replicate nested inside a map

(assuming `arr = [1,2,3,4]`):

```
3. let ip1rs= F(map (\(i,ip1) -> replicate i ip1) (zip arr ip1s)))
≡
vals = write inds  ip1s (replicate size 0) -- [2, 3, 0, 4, 0, 0, 5, 0, 0, 0]
ip1rs= sgmScan^inc (+) 0 flag vals          -- [2, 3, 3, 4, 4, 4, 5, 5, 5, 5]
```

According to rule (2) map nested inside a map

```
F(map (\(z) -> map (+) z) (zip ip1rs iots))
≡
4. result = map (+) (zip ip1rs iots)
-- [2, 3, 3, 4, 4, 4, 5, 5, 5, 5]
-- [0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
--  +  +  +  +  +  +  +  +  +  +
-------------------------------
-- [2, 3, 4, 4, 5, 6, 5, 6, 7, 8] values
-- [1, 2, 0, 3, 0, 0, 4, 0, 0, 0] flags
```

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 a Simple (Contrived) Program
   - Flattening Prime-Number (Sieve) Computation
   - Flattening Quicksort

How Does One Flattens Prime Numbers?

The important bit with nested parallelism:

```
sqrt_primes = primesOpt (sqrt (fromIntegral n))
nested = map (\p -> let m = (n `div` p)
                    in  map (\j -> j*p) [2..m]
             ) sqrt_primes
not_primes  = reduce (++) [] nested
```

How Does One Flattens Prime Numbers?

The important bit with nested parallelism:

```
sqrt_primes = primesOpt (sqrt (fromIntegral n))
nested = map (\p -> let m = (n `div` p)
                    in  map (\j -> j*p) [2..m]
             ) sqrt_primes
not_primes  = reduce (++) [] nested
```

Normalize the nested map:

```
sqrt_primes = primesOpt (sqrt (fromIntegral n))
nested = map (\p ->
                    let m   = n `div` p        in
                    let mm1 = m - 1            in
                    let iot = iota mm1         in          -- F rule 5
                    let twom= map (+2) iot     in          -- F rule 2
                    let rp  = replicate mm1 p  in          -- F rule 4
                    in  map (\(j,p) -> j*p) (zip twom rp) -- F rule 2
             ) sqrt_primes
not_primes  = reduce (++) [] nested
```

Flattened Prime-Number (Sieve) Computation

```
sqrt_primes = primesOpt (sqrt (fromIntegral n))

ms   = map (\p -> n `div` p) sqrt_primes
mm1s = map (-1) ms

inds = sgmScan^exc (+) 0 mm1s
size = (last inds) + (last mm1s)
flag = write inds nm1s (replicate size 0)
tmp0 = replicate size 1
iots = sgmScan^exc (+) 0 flag tmp0

twoms= map (+2) iots

tmp1 = write inds sqrt_primes (replicate size 0)
rps  = sgmScan^inc (+) 0 flag tmp0

nested= map (\(j,p) -> j*p) (zip twoms rps) -- flag array

-- nested is already flattened, hence (reduce (++) [] nested) has no effects
not_primes  = nested
```

Recounting Quicksort

Recount the classic nested-parallel definition:

```
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
  in  (nestedQuicksort s1) ++ (nestedQuicksort s2)
  -- can be re-written as:
  -- rs = map nestedQuicksort [s1, s2]
  -- in (rs !! 0) ++ (rs !! 1)
```

Normalizing Quicksort

Key Idea: write a function with the semantics of `map nestedQuicksort`, i.e., it operates on array of arrays, and, for simplicity, use `parFilter ::` $(\alpha \rightarrow \mathit{Bool}) \rightarrow [\alpha] \rightarrow ([\alpha], [\mathit{Int}])$.

```
quicksort^lift :: [[a]] -> [[a]]
quicksort^lift arrofarrs =
  map (\arr ->
          if  (length arr) <= 1 then arr else -- F rule 6
          let i  = getRand (0, (length arr) - 1)
              a  = arr !! i
              (s, flag) = parFilter (<a) arr -- -- F rule 7
              (s1, s2)  = split flag[0] s
              rs = quicksort^lift [s1, s2]
          in  (rs !! 0) ++ (rs !! 1)
      ) arrofarrs
```

Flattened Quicksort

Semantically operates on a 2-dim array ⇒ requires sizes array arg.

```
flatQuicksort :: a -> [Int] -> [a] -> [a]
flatQuicksort sizes arrofarrs = --Assume sizes=[4,0,0,0], arrofarrs=[3,2,4,1]
sizes_distr = sgmScan^inc (+) 0 sizes sizes -- [4,4,4,4]
ais = zip3 arrofarrs sizes_distr sizes (iota (length arrofarrs))
(ais',flg)=parFilter(\(_,s,_)->s <= 1) ais--[(3,4,0),(2,4,1),(4,4,2),(1,4,3)]
(ais^t,ais^f) = split flg[0] ais'          --([],[(3,4,0),(2,4,1),(4,4,2),(1,4,3)])
(arr^t,sizes^t,_,inds^t) = unzip3 ais^t                  --([],[],[])
(arr^f,sizes^f,_,inds^f) = unzip ais^f                   --([3,2,4,1], [4,0,0,0], [0,1,
arr^then = arr^t

sgm_beg_ind = sgmScan^exc (+) 0 sizes^f
is = map (\(s,j)->getRand j 0 (s-1)) (zip sizes_distr (length arrofarrs))--[0]
as = map (\ (i,beg_sgm) -> arrofarrs !! (i+beg_sgm)) (zip is sgm_beg_ind)--[3]
tot_size^f = reduce (+) 0 sizes^f
a_distr = sgmScan^inc (+) 0 sizes $ -- [3,3,3,3]
          write sgm_beg_ind as (replicate 0 tot_size^f)

(sizes^else,arr^else) = segmSpecialFilter (\(r,x)->(x < r)) -- ([2,1,3,4], [2,0,2,0])
                        sizes^f (zip a_distr arr^f)
arr'^else = flatQuicksort sizes^else arr^else [1,2,3,4]
write inds^f arr'^else (write inds^t arr^then [0,...,0]) --[1,2,3,4]
```

Intuitive Flattened Quicksort

```
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'
```

Intuitive Flattened Quicksort

```
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])

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 a Simple (Contrived) Program
   - Flattening Prime-Number (Sieve) Computation
   - Flattening Quicksort