Faculty of Science

Parallel Basic Blocks and Flattening Nested Parallelism

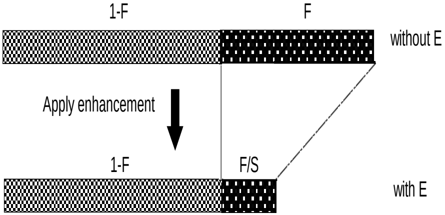
Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

September 2020 PMPH Lecture Slides

1. Implementation of Flat Bulk Operators
   - Amdahl's Law
   - Work-Depth Asymptotic
   - Implementation of Reduce
   - Implementation of Scan
   - Implementation of Segmented Scan
   - Other Second-Order Parallel Operators
2. Nested Data-Parallel Applications
   - Sieve: Prime-Numbers Computation
   - Nested Parallel Quicksort
3. Flattening Nested Parallelism
   - Rules For Flattening
   - Flattening a Simple (Contrived) Program
   - Flattening Prime-Number (Sieve) Computation
   - Flattening Quicksort

Amdahl's Law



Enhancement accelerates a fraction F of the task by a factor S:

$$T_{exe}(withE) = T_{exe}(withoutE) \times [(1-F) + \frac{F}{S}]$$

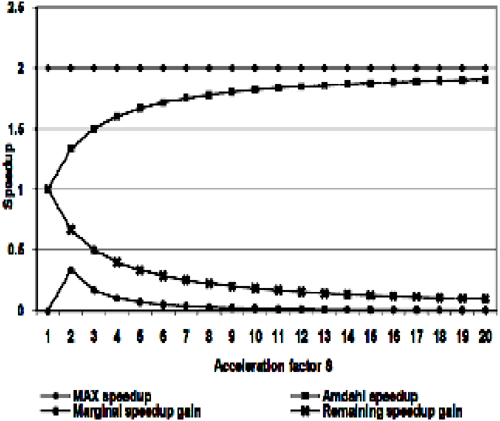
$$Speedup(E) = \frac{T_{exe}(withoutE)}{T_{exe}(withE)} = \frac{1}{(1-F)+\frac{F}{S}}$$

Amdahl's Law

1 Improvement is limited by the $1 - F$ part of the execution that cannot be optimized: $Speedup(E) < \frac{1}{1-F}$

2 Optimize the common case & execute the rare case in software.

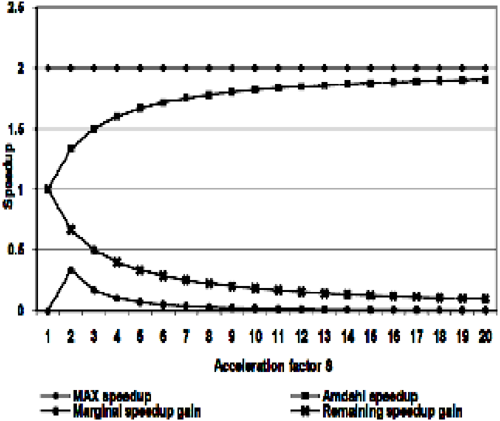
3 Low of diminishing returns



F=0.5

Amdahl's Law

1. Improvement is limited by the $1 - F$ part of the execution that cannot be optimized: $Speedup(E) < \frac{1}{1-F}$
2. Optimize the common case & execute the rare case in software.
3. Low of diminishing returns

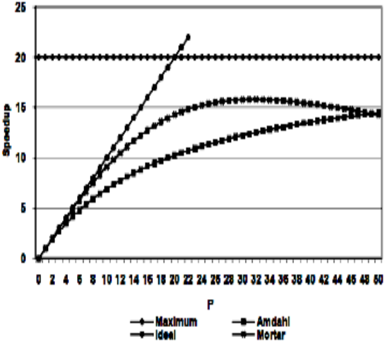


F=0.5

- every increment of S consumes new resources and is less rewarding:
- $S = 2 \Rightarrow 33\%$ speedup,
- $S = 5 \Rightarrow 6.67\%$ speedup.

Amdahl's Law: Parallel Speedup

$$S_P = \frac{T_1}{T_P} = \frac{P}{F+P(1-F)} < \frac{1}{1-F}$$

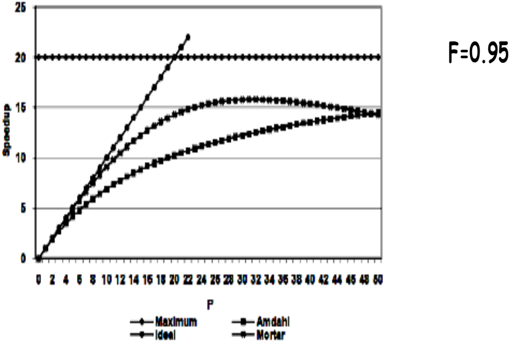


F=0.95

- Typically: speedup is sublinear, e.g., due to inter-thread communic.
- Sometimes superlinear speedup due to cache effects.
- Unforgiving Law: even if 99% is parallelized, $S_\infty < 100$.

Amdahl's Law: Parallel Speedup

$$S_P = \frac{T_1}{T_P} = \frac{P}{F+P(1-F)} < \frac{1}{1-F}$$



- Typically: speedup is sublinear, e.g., due to inter-thread communic.
- Sometimes superlinear speedup due to cache effects.
- Unforgiving Law: even if 99% is parallelized, $S_\infty < 100$.

Hardware Trend is to ever increase the number of cores.

Amdhal's Law: reason about parallelism asymptotically (∞ # cores), i.e., systematically exploit all levels of application's parallelism.

1. Implementation of Flat Bulk Operators
   - Amdahl's Law
   - Work-Depth Asymptotic
   - Implementation of Reduce
   - Implementation of Scan
   - Implementation of Segmented Scan
   - Other Second-Order Parallel Operators
2. Nested Data-Parallel Applications
   - Sieve: Prime-Numbers Computation
   - Nested Parallel Quicksort
3. Flattening Nested Parallelism
   - Rules For Flattening
   - Flattening a Simple (Contrived) Program
   - Flattening Prime-Number (Sieve) Computation
   - Flattening Quicksort

Parallel Random Access Machine (PRAM)

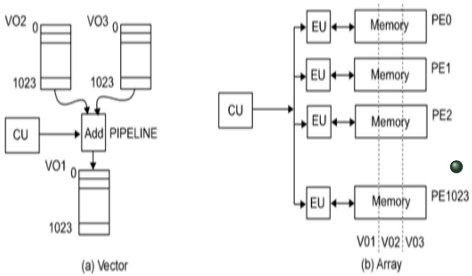
PRAM focuses exclusively on parallelism and ignores issues related to synchronization and communication:

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:

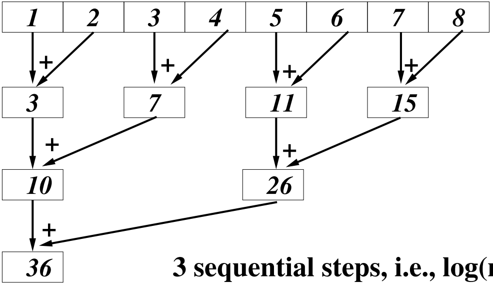
1. p processors connected to shared memory
2. each processor has an unique id (index) i, $1 \leq i \leq p$
3. SIMD execution, each parallel instruction requires unit time,
4. each processor has a flag that controls whether it is active in the execution of an instruction.



- 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.

1. Implementation of Flat Bulk Operators
   - Amdahl's Law
   - Work-Depth Asymptotic
   - Implementation of Reduce
   - Implementation of Scan
   - Implementation of Segmented Scan
   - Other Second-Order Parallel Operators
2. Nested Data-Parallel Applications
   - Sieve: Prime-Numbers Computation
   - Nested Parallel Quicksort
3. Flattening Nested Parallelism
   - Rules For Flattening
   - Flattening a Simple (Contrived) Program
   - Flattening Prime-Number (Sieve) Computation
   - Flattening Quicksort

Reducing in Parallel



3 sequential steps, i.e., log(n)

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: $O(\frac{n}{P} + lg\ P)$.

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
$\oplus : T \times T \rightarrow T$ associative
Output: S = $\oplus_{j=1}^{n} a_j$

```
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]
```

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

$$O(\frac{n}{P}) \leq O(Runtime) < O(\frac{n}{P} + lg\ n)$$

Note that the program is hardware-agnostic, i.e., has no notion of the number of cores.

Reduce: Naive Implementation in Futhark

```
-- Reduction by hand in Futhark: red-by-hand.fut
-- ==
-- compiled input { [1.0f32, -2.0, 3.0] }
-- output { 2.0f32 }
-- compiled input @ data/f32-arr-16777216.in
-- output { 3091.746094f32 }

let main [n] (a : [n]f32) : f32 = -- assumes n = 2^k
  let k = t32 <| f32.log2 <| r32 n
  let b =
    loop b = a for h < k do
        let n' = n >> (h+1)
        in  map (\i -> unsafe (b[2*i]+b[2*i+1]) ) (iota n')
  in b[0]
```

Performance w.r.t. the native reduce? Compile and Run with:

```
$ futhark opencl red-by-hand.fut
$ futhark dataset --f32-bounds=-1.0:1.0 -g [8388608]f32
| ./red-by-hand -t /dev/stderr -r 10
```

1. Implementation of Flat Bulk Operators
   - Amdahl's Law
   - Work-Depth Asymptotic
   - Implementation of Reduce
   - Implementation of Scan
   - Implementation of Segmented Scan
   - Other Second-Order Parallel Operators
2. Nested Data-Parallel Applications
   - Sieve: Prime-Numbers Computation
   - Nested Parallel Quicksort
3. Flattening Nested Parallelism
   - Rules For Flattening
   - Flattening a Simple (Contrived) Program
   - Flattening Prime-Number (Sieve) Computation
   - Flattening Quicksort

Zip, Unzip, iota, replicate

- `zip :` $[n]\alpha_1 \rightarrow [n]\alpha_2 \rightarrow [n](\alpha_1,\alpha_2)$
- `zip` $[a_1,\ldots,a_n]$ $[b_1,\ldots,b_n] \equiv [(a_1,b_1),\ldots,(a_n,b_n)]$,

Zip, Unzip, iota, replicate

- `zip :` $[n]\alpha_1 \rightarrow [n]\alpha_2 \rightarrow [n](\alpha_1,\alpha_2)$
- `zip` $[a_1,\ldots,a_n]$ $[b_1,\ldots,b_n] \equiv [(a_1,b_1),\ldots,(a_n,b_n)]$,
- `unzip :` $[n](\alpha_1,\alpha_2) \rightarrow ([n]\alpha_1,[n]\alpha_2)$
- `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`/`unzip` are syntactic sugar

Zip, Unzip, iota, replicate

- zip : [n]α_1 $\rightarrow$ [n]α_2 $\rightarrow$ [n](α_1,α_2)
- zip [a_1,...,a_n] [b_1,...,b_n] $\equiv$ [(a_1,b_1),...,(a_n,b_n)],
- unzip : [n](α_1,α_2) $\rightarrow$ ([n]α_1,[n]α_2)
- unzip [(a_1,b_1),...,(a_n,b_n)]$\equiv$([a_1,...,a_n],[b_1,...,b_n]),
- In some sense `zip`/`unzip` are syntactic sugar
- replicate : (n: int) $\rightarrow$ α $\rightarrow$ [n]α
- replicate n a $\equiv$ [a, a,..., a],

Zip, Unzip, iota, replicate

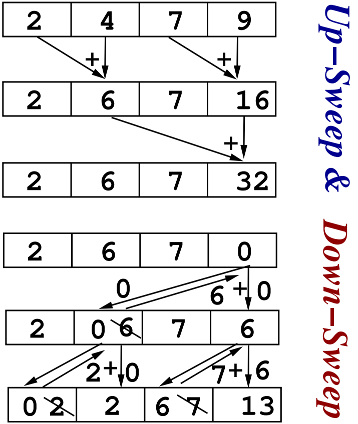
- `zip :` $[n]\alpha_1 \rightarrow [n]\alpha_2 \rightarrow [n](\alpha_1,\alpha_2)$
- `zip` $[a_1,\ldots,a_n]$ $[b_1,\ldots,b_n] \equiv [(a_1,b_1),\ldots,(a_n,b_n)]$,
- `unzip :` $[n](\alpha_1,\alpha_2) \rightarrow ([n]\alpha_1,[n]\alpha_2)$
- `unzip` $[(a_1,b_1),\ldots,(a_n,b_n)] \equiv ([a_1,\ldots,a_n],[b_1,\ldots,b_n])$,
- In some sense `zip`/`unzip` are syntactic sugar
- `replicate :` $(n:\ \texttt{int}) \rightarrow \alpha \rightarrow [n]\alpha$
- `replicate n a` $\equiv$ `[a, a,..., a]`,
- `iota :` $(n:\ \texttt{int}) \rightarrow [n]\texttt{int}$
- `iota n` $\equiv$ `[0, 1,..., n-1]`

Note: in Haskell `zip` does not expect same-length arrays; in Futhark it does!

Map, Reduce, and Scan Types and Semantics

- $[n]\alpha$ denotes the type of an array of n elements of type α.
- $\texttt{map} : (\alpha \to \beta) \to [n]\alpha \to [n]\beta$
  $\texttt{map f } [x_1,\ldots,x_n] = [\texttt{f } x_1,\ldots, \texttt{ f } x_n]$,
  i.e., $x_i : \alpha, \forall i$, and $\texttt{f} : \alpha \to \beta$.
- $\texttt{reduce} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to \alpha$
  $\texttt{reduce} \odot \texttt{ e } [x_1,x_2,..,x_n] = e \odot x_1 \odot x_2 \odot \ldots \odot x_n$,
  i.e., $e : \alpha$, $x_i : \alpha, \forall i$, and $\odot : \alpha \to \alpha \to \alpha$.
- $\texttt{scan}^{exc} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$
  $\texttt{scan}^{exc} \odot \texttt{ e } [x_1,\ldots,x_n] = [e, e \odot x_1, \ldots, e \odot x_1 \odot \ldots x_{n-1}]$
  i.e., $e : \alpha$, $x_i : \alpha, \forall i$, and $\odot : \alpha \to \alpha \to \alpha$.
- $\texttt{scan}^{inc} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$
  $\texttt{scan}^{inc} \odot \texttt{ e } [x_1,\ldots,x_n] = [e \odot x_1, \ldots, e \odot x_1 \odot \ldots x_n]$
  i.e., $e : \alpha$, $x_i : \alpha, \forall i$, and $\odot : \alpha \to \alpha \to \alpha$.

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

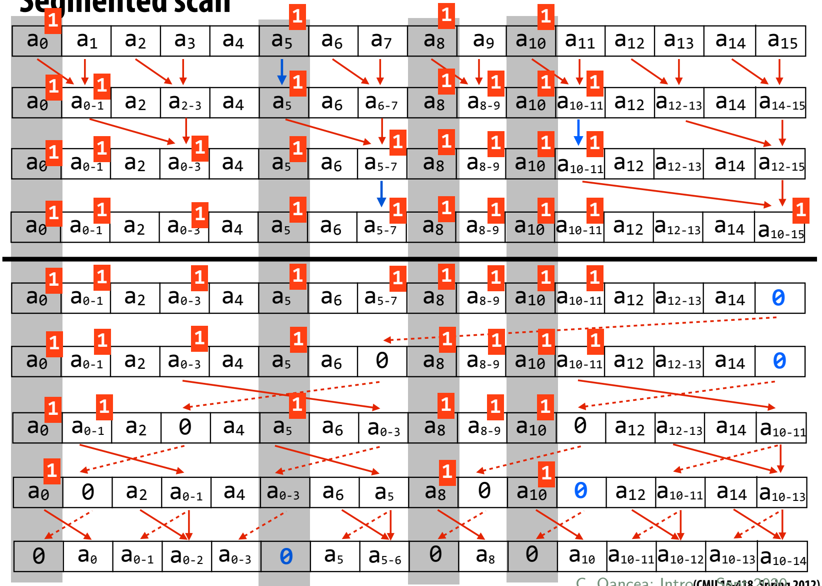
```
forall i = 0 : n-1 do
  B[i] ← A[i]
enddo

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.

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

Segmented scan



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

Segmented Inclusive Scan with Operator $\oplus$ (Haskell)

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] ≡
```

Segmented Inclusive Scan with Operator $\oplus$ (Haskell)

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
    ≡

  [1,3,6,1,3,6,10]       -- scanned data
```

Segmented Inclusive Scan with Operator ⊕ (Haskell)

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
        ≡
  [1,3,6,1,3,6,10]       -- scanned data
```

Can be obtained by replacing the following Futhark operator:

```
let segmented_scan [n] 't (op: t -> t -> t) (ne: t)
                   (flags: [n]bool) (arr: [n]t) : [n]t =
  let (_, res) = unzip <|
    scan (\(x_flag,x) (y_flag,y) ->
            let fl = x_flag || y_flag
            let vl = if y_flag then y else op x y
            in  (fl, vl)
         ) (false, ne) (zip flags arr)
  in  res
```

How about Exclusive Scan?

Map2, Filter

- map2 : $(\alpha_1 \to \alpha_2 \to \beta) \to$ [n]α_1 $\to$ [n]α_2 $\to$ [n]β
- map2 $\odot$ [a_1,...,a_n] [b_1,...,b_n] $\equiv$ [$a_1 \odot b_1$,...,$a_n \odot b_n$]
- map3 ...

Map2, Filter

- `map2 :` $(\alpha_1 \to \alpha_2 \to \beta) \to [\mathtt{n}]\alpha_1 \to [\mathtt{n}]\alpha_2 \to [\mathtt{n}]\beta$
- `map2` $\odot$ $[\mathtt{a}_1, \ldots, \mathtt{a}_n]$ $[\mathtt{b}_1, \ldots, \mathtt{b}_n]$ $\equiv$ $[\mathtt{a}_1 \odot \mathtt{b}_1, \ldots, \mathtt{a}_n \odot \mathtt{b}_n]$
- `map3 ...`
- `filter :` $(\alpha \to \mathtt{Bool}) \to [\mathtt{n}]\alpha \to [\mathtt{m}]\alpha$ (where $\mathtt{m} \leq \mathtt{n}$)
- `filter p` $[\mathtt{a}_1, \ldots, \mathtt{a}_n]$ = $[\mathtt{a}_{k_1}, \ldots, \mathtt{a}_{k_m}]$ such that $\mathtt{k}_1 < \mathtt{k}_2 < \ldots < \mathtt{k}_m$, and denoting by $\overline{\mathtt{k}} = \mathtt{k}_1, \ldots, \mathtt{k}_m$, we have $(\mathtt{p}\ \mathtt{a}_j == \mathtt{true})\ \forall j \in \overline{\mathtt{k}}$, **and** $(\mathtt{p}\ \mathtt{a}_j == \mathtt{false})\ \forall j \notin \overline{\mathtt{k}}$.

Note: in Haskell `map2, map3` do not expect same-length arrays; in Futhark they do!

Scatter: A Parallel Write Operator

Scatter updates in parallel a base array with a set of values at specified indices:

scatter : $*[m]\alpha \rightarrow [n]\text{int} \rightarrow [n]\alpha \rightarrow *[m]\alpha$

A (data vector) = [b0, b1, b2, b3]
I (index vector) = [2, 4, 1, -1]
X (input array) = [a0, a1, a2, a3, a4, a5]
scatter X I A = [a0, b2, b0, a3, b1, a5]

Scatter: A Parallel Write Operator

Scatter updates in parallel a base array with a set of values at specified indices:

scatter : $*[m]\alpha \rightarrow [n]\text{int} \rightarrow [n]\alpha \rightarrow *[m]\alpha$

```
A (data vector)   =[b0, b1, b2, b3]
I (index vector)  =[2,  4,  1,  -1]
X (input array)   =[a0, a1, a2, a3, a4, a5]
scatter X I A     =[a0, b2, b0, a3, b1, a5]
```

scatter has $D(n) = \Theta(1)$ and $W(n) = \Theta(n)$,
i.e., requires `n` update operations (`n` is the size of I or A, not of X!).

1. Array X is consumed by `scatter`; following uses of X are illegal!
2. Similarly, X can alias neither I nor A!

In Futhark, `scatter` check and ignores the indices that are out of bounds (no update is performed on those). This is useful for padding the iteration space in order to obtain regular parallelism.

Permute, Split, Replicate, Iota

- Operator to permute in parallel based on a set (array) of indices:
  permute : [n]int→[n]α→[n]α.
  `permute I A ≡ scatter (replicate n e) I A`
  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]

- `split : (i:int)` $\rightarrow$ `[n]`α $\rightarrow$ `([i]`α`,[n-i]`α`)`
  split i $[a_0,\ldots,a_{n-1}] \equiv ([a_0,\ldots,a_{i-1}], [a_i,\ldots,a_{n-1}])$

- `replicate : (n:int)` $\rightarrow \alpha \rightarrow$ `[n]`α
  `replicate n a` $\equiv$ `[a, a,..., a]`, i.e., a is replicated `n` times.

- `iota : (n:int)` $\rightarrow$ `[n]int`
  `iota n = [0,...,n-1]`

Partition2/Filter Implementation

`partition2 :` $(\alpha \rightarrow$`Bool`$) \rightarrow$ `[n]`$\alpha \rightarrow$ `([n]i32,[n]`α`)`

In result, the elements satisfying the predicate occur before the others.

Can be implemented by means of `map`, `scan` and `scatter`.

Partition2/Filter Implementation

`partition2 :` $(\alpha \rightarrow$`Bool`$) \rightarrow$ `[n]`$\alpha \rightarrow$ `([n]i32,[n]`α`)`

In result, the elements satisfying the predicate occur before the others.

Can be implemented by means of `map`, `scan` and `scatter`.

```
let partition2 't [n] (dummy: t)
      (cond: t -> bool) (X: [n]t) :
                        (i32, [n]t) =
  let cs = map cond X
  let tfs= map (\ f->if f then 1
                          else 0) cs
  let isT= scan (+) 0 tfs
  let i   = isT[n-1]

  let ffs= map (\f->if f then 0
                          else 1) cs
  let isF= map (+i) <| scan (+) 0 ffs
  let inds=map (\(c,iT,iF) ->
                    if c then iT-1
                         else iF-1
               ) (zip3 cs isT isF)
  let tmp = replicate n dummy
  in (i, scatter tmp inds X)
```

```
Assume X = [5,4,2,3,7,8], and
cond is T(rue) for even nums.
```

Partition2/Filter Implementation

partition2 : $(\alpha \to \text{Bool}) \to [n]\alpha \to ([n]\text{i32}, [n]\alpha)$

In result, the elements satisfying the predicate occur before the others.

Can be implemented by means of map, scan and scatter.

```
let partition2 't [n] (dummy: t)
      (cond: t -> bool) (X: [n]t) :
                     (i32, [n]t) =
  let cs = map cond X
  let tfs= map (\ f->if f then 1
                         else 0) cs
  let isT= scan (+) 0 tfs
  let i  = isT[n-1]

  let ffs= map (\f->if f then 0
                        else 1) cs
  let isF= map (+i) <| scan (+) 0 ffs
  let inds=map (\(c,iT,iF) ->
                 if c then iT-1
                      else iF-1
             ) (zip3 cs isT isF)
  let tmp = replicate n dummy
  in (i, scatter tmp inds X)
```

```
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]
```

Computing Prime Numbers: First Attempt

See also "Scan as Primitive Parallel Operation" [Bleelloch] (attached in Additional Teaching Material module).

Start with an array of size n filled initially 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!)

Why `i < sqrt(n)`?

Computing Prime Numbers: First Attempt

See also "Scan as Primitive Parallel Operation" [Bleelloch] (attached in Additional Teaching Material module).

Start with an array of size n filled initially 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!)

Why `i < sqrt(n)`?
Because a non-prime number has to have a multiple less than $\sqrt{n}$.

Computing Prime Numbers: 1st Attempt (Futhark)

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 $\sqrt{n}$.

```
let primesHelp [np1] (sq : i32)
        (a : *[np1]i32) : [np1]i32 =
  let n = np1 - 1 in
  loop(a) for j < (sq-1) do
    let i    = j + 2
    let m    = (n / i) - 1
    let inds= map (\k->(k+2)*i)(iota m)
    in  scatter a inds (replicate m 0)

let main (n : i32) : []i32 =
   let a = map (\i->if i==0 || i==1
                    then 0 else 1)
               (iota (n+1))
   let sq= i32 (f32.sqrt (f32 n))
   let fl= primesHelp sq a
   in  filter (\i->unsafe fl[i]!=0)
              (iota (n+1))
```

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

iteration j = 0, i = 2
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]

iteration j = 1, i = 3
m    = (9 ‘div‘ 3) - 1 = 2
inds = [6, 9]
vals = [0, 0]
a’’= [0,0,1,1,0,1,0,1,0,0]

iteration j = 2, i = 4
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!)

Prime Numbers: Nested Parallelism in Haskell

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 $n^{(1/2)^{depth}} = 2$).

Prime Numbers: Nested Parallelism in Haskell

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 $n^{(1/2)^{depth}} = 2$).

```
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=scatter(replicate (n+1) True)
                           not_primes zeros
       (primes,_)= unzip $ filter (\(i,f)->f)
                    $ (zip [0..n] prime_flags)
   in drop 2 primes
```

Prime Numbers: Nested Parallelism in Haskell

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 $n^{(1/2)^{depth}} = 2$).

```
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=scatter(replicate (n+1) True)
                            not_primes zeros
       (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)

-- Is this implementation correct?
-- 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)

-- Is this implementation correct?
-- 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: $O(lg^2\ n)$.

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]`?

sgmScan^{inc} `(+) 0 flags flags`

Nested *vs* Flattened Parallelism: Replicate in a Map

(4) Replicate nested inside a map:

```
map2 (\ n m -> replicate n m) [1,3,2] [7,8,9] ≡
[ 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:

```
map2 (\ n m -> replicate n m) [1,3,2] [7,8,9] ≡
[ replicate 1 7, replicate 3 8, replicate 2 9 ] ≡
[ [7], [8,8,8], [9,9] ]
```

becomes a composition of scans and scatter:

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

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

3-4. write the array elems at the position where a segment starts

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

- Implementation shortcomings:

Nested *vs* Flattened Parallelism: Replicate in a Map

(4) Replicate nested inside a map:

```
map2 (\ n m -> replicate n m) [1,3,2] [7,8,9] ≡
[ replicate 1 7, replicate 3 8, replicate 2 9 ] ≡
[ [7], [8,8,8], [9,9] ]
```

becomes a composition of scans and scatter:

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

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

3-4. write the array elems at the position where a segment starts

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

- Implementation shortcomings: replicate 0 7?

Nested *vs* Flattened Parallelism: Replicate in a Map

(4) Replicate nested inside a map:

```
map2 (\ n m -> replicate n m) [1,3,2] [7,8,9] ≡
[ replicate 1 7, replicate 3 8, replicate 2 9 ] ≡
[ [7], [8,8,8], [9,9] ]
```

becomes a composition of scans and scatter:

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

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

3-4. write the array elems at the position where a segment starts

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

- Implementation shortcomings: replicate 0 7? sgmScan^{inc} $(+)$?

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 scatter:

Note that iota n ≡ $\texttt{scan}^{exc}$ (+) 0 (replicate n 1)

```
1. arr  = [1, 3, 2]
2. inds = scan^exc (+) 0 arr          -- [0,1,4]
3. size = (last inds) + (last arr) -- 4 + 2 = 6
4. flag = scatter (replicate size 0)
                  inds  -- [0,1,4]
                  arr   -- [1,3,2]
--                [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 with inner parallelism 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 with inner parallelism 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', s)=partition2(\(x,_)->odd x) ais--([(3,0),(7,3),(4,1),(6,2)], 2)
3. (ais^t,ais^f) = split s 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 = scatter (scatter [0,...,0] inds^t arr^then) inds^f arr^else --[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, i.e., here you apply flattening recursively on each map to unveil more parallelism,

7. write back the resulted elements at original positions.

Nested *vs* Flattened Parallelism: Reduce Inside Map

(7) Reduce Inside a Map or Segmented Reduce:

```
let arr = [[1, 3, 4], [6, 7]] in
map (\x -> reduce + 0 x) arr
-- should result in [8, 13]
```

Nested *vs* Flattened Parallelism: Reduce Inside Map

(7) Reduce Inside a Map or Segmented Reduce:

```
let arr = [[1, 3, 4], [6, 7]] in
map (\x -> reduce + 0 x) arr
-- should result in [8, 13]
```

translates to a scan-pack composition:

```
1. shp    = [3, 2]
2. flags  = [1, 0, 0, 1, 0]
3. arr    = [1, 3, 4, 6, 7]
4. n      = length arr
5. indsp1 = scan^inc (+) 0 shp           -- [3, 5]
6. sc_arr = sgmScan^inc (+) 0 flags arr -- [1, 4, 8, 6, 13]
7. res    = map (\ ip1 -> sc_arr[ip1-1]) indsp1
```

1. Implementation of Flat Bulk Operators
   - Amdahl's Law
   - Work-Depth Asymptotic
   - Implementation of Reduce
   - Implementation of Scan
   - Implementation of Segmented Scan
   - Other Second-Order Parallel Operators

2. Nested Data-Parallel Applications
   - Sieve: Prime-Numbers Computation
   - Nested Parallel Quicksort

3. Flattening Nested Parallelism
   - Rules For Flattening
   - Flattening a Simple (Contrived) Program
   - Flattening Prime-Number (Sieve) Computation
   - Flattening Quicksort

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
           map2 (+) ip1r iot            ) arr
```

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

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
           map2 (+) ip1r iot            ) arr
```

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

```
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(map2 (\ i ip1 -> (replicate i ip1)) arr ip1s)
4. in  𝓕(map2 (\ ip1r iot -> map2 (+) ip1r iot) 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 =` $\mathcal{F}$`(map (\i -> iota i) arr)`

$\equiv$

```
inds = scan^exc (+) 0 arr -- [0,1,3,6]
size = (last inds) + (last arr) -- 6 + 4 = 10
flag = scatter (replicate size 0)
               inds arr
--             [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(map2 (\ i ip1 -> replicate i ip1) arr ip1s)
≡
vals = scatter (replicate size 0) inds  ip1s -- [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(map2 (\ ip1r iot -> map2 (+) ip1r iot) ip1rs iots)
≡
4. result = map (+) 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. Implementation of Flat Bulk Operators
   - Amdahl's Law
   - Work-Depth Asymptotic
   - Implementation of Reduce
   - Implementation of Scan
   - Implementation of Segmented Scan
   - Other Second-Order Parallel Operators
2. Nested Data-Parallel Applications
   - Sieve: Prime-Numbers Computation
   - Nested Parallel Quicksort
3. 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           -- distribute map
                 let mm1  = m - 1            in           -- distribute map
                 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                  -- ignore, already flat
```

Flattening PrimeOpt is part of Weekly Assignment 1!

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 `partition2 ::` $(\alpha \rightarrow \mathit{Bool}) \rightarrow [\alpha] \rightarrow ([\alpha], [\mathit{Int}])$.

```
quicksort^lift :: [[a]] -> [[a]]
quicksort^lift arrofarrs =
  map (\arr ->
          if  (length arr) < 2 then arr else -- F rule 6
          let i  = getRand (0, (length arr) - 1)
              a  = arr !! i
              (s, flag) = partition2 (<a) arr
              (s1, s2)  = split flag[0] s
              rs = quicksort [s1, s2]
          in  (rs !! 0) ++ (rs !! 1)
      ) arrofarrs
```

`(length arr) < 2` will not work correctly if the input array has duplicated elements (?)

What should the result be of distributing the `map` over `rs = quicksort [s1,s2]`?

Normalizing Quicksort

Flattening quicksort could be a project if anyone is interested!

- Try to distribute the outer `map` across the inner code;
- The recursion can be re-written as a loop in which the stopping condition is that all elements of the array are sorted (expressed as a map-reduce composition);
- Distributing the puter `map` over `rs = quicksort [s1,s2]` results in quicksort^{lift},
- The difficult step is to flatten `map (partition2)`.