Work Efficient Higher Order Vectorisation

Ben Lippmeier[✿] Manuel Chakravarty[✿] Gabriele Keller[✿]
Roman Leshchinskiy Simon Peyton Jones[★]

[✿]University of New South Wales
[★]Microsoft Research Ltd

ICFP 2012

Data Parallel Haskell (DPH)

- Nested data parallel programming on shared memory multicore.

Data **P**arallel **H**askell (**DPH**)

- Nested data parallel programming on shared memory multicore.
- Compiling common classes of programs used to break their asymptotic work complexity.

Data **P**arallel **H**askell (**DPH**)

- Nested data parallel programming
  on shared memory multicore.

- Compiling common classes of programs used
  to break their asymptotic work complexity.

- Not anymore!

```
retrieve [[A B]   [C D E] [F G] [H]]    (xss)
         [[1 0 1] [2]     [1 0] [0]]    (iss)
     ==> [[B A B] [E]     [G F] [H]]
```

```
retrieve [[A B]   [C D E] [F G] [H]]     (xss)
         [[1 0 1] [2]     [1 0] [0]]     (iss)
    ==>  [[B A B] [E]     [G F] [H]]
```

User written source code.

```
retrieve :: A (A Char) -> A (A Int) -> A (A Char)
retrieve xss iss
  = zipWithP mapP (mapP indexP xss) iss
```

```
              indexP :: A Char -> Int -> Char
     mapP indexP xss :: A (Int -> Char)
```

```
retrieve [[A B]   [C D E] [F G] [H]]     (xss)
         [[1 0 1] [2]     [1 0] [0]]     (iss)
     ==> [[B A B] [E]     [G F] [H]]
```

User written source code.

```
retrieve :: A (A Char) -> A (A Int) -> A (A Char)
retrieve xss iss
  = zipWithP mapP (mapP indexP xss) iss
```

nested parallel computation

```
         indexP :: A Char -> Int -> Char
mapP indexP xss :: A (Int -> Char)
```

```
retrieve [[A B]   [C D E] [F G] [H]]     (xss)
         [[1 0 1] [2]     [1 0] [0]]     (iss)
    ==>  [[B A B] [E]     [G F] [H]]
```

User written source code.

```
retrieve :: A (A Char) -> A (A Int) -> A (A Char)
retrieve xss iss
  = zipWithP mapP (mapP indexP xss) iss
```

builds an array of partially applied functions

```
         indexP :: A Char -> Int -> Char
mapP indexP xss :: A (Int -> Char)
```

Vectorisation

```
retrieve [[A B]   [C D E] [F G] [H]]     (xss)
         [[1 0 1] [2]     [1 0] [0]]     (iss)
    ==>  [[B A B] [E]     [G F] [H]]
```

User written source code.

```
retrieve :: A (A Char) -> A (A Int) -> A (A Char)
retrieve xss iss
  = zipWithP mapP (mapP indexP xss) iss
```

Vectorised data-parallel version.

```
retrieve_v :: A (A Char) -> A (A Int) -> A (A Char)
retrieve_v xss iss
  = let ns = takeLengths iss
    in  unconcat iss
        $ index_l (replicates ns xss)
        $ concat iss
```

```
retrieve [[A B]   [C D E] [F G] [H]]     (xss)
         [[1 0 1] [2]     [1 0] [0]]     (iss)
    ==>  [[B A B] [E]     [G F] [H]]
```

```
retrieve [[A B]   [C D E] [F G] [H]]     (xss)
         [[1 0 1] [2]     [1 0] [0]]     (iss)
    ==>  [[B A B] [E]     [G F] [H]]

ns   = takeLengths iss
     = [3 1 2 1]
```

```
 retrieve [[A B]   [C D E] [F G] [H]]    (xss)
          [[1 0 1] [2]     [1 0] [0]]    (iss)
     ==>  [[B A B] [E]     [G F] [H]]

 ns   = takeLengths iss
      = [3 1 2 1]

xss1 = replicates ns xss
     = [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
```

```
 retrieve [[A B]   [C D E] [F G] [H]]     (xss)
          [[1 0 1] [2]     [1 0] [0]]     (iss)
     ==>  [[B A B] [E]     [G F] [H]]

 ns   = takeLengths iss
      = [3 1 2 1]

xss1 = replicates ns xss
     = [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]

iss2 = concat iss
     = [1      0     1     2       1     0     0]
```

```
 retrieve [[A B]   [C D E] [F G] [H]]    (xss)
          [[1 0 1] [2]     [1 0] [0]]    (iss)
     ==>  [[B A B] [E]     [G F] [H]]

 ns   = takeLengths iss
      = [3 1 2 1]

xss1  = replicates ns xss
      = [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]

iss2  = concat iss
      = [1     0     1     2       1     0     0]

xss2  = index_l xss1 iss2
      = [B     A     B     E       G     F     H]
```

```
 retrieve [[A B]   [C D E] [F G] [H]]     (xss)
          [[1 0 1] [2]     [1 0] [0]]     (iss)
     ==>  [[B A B] [E]     [G F] [H]]

 ns   = takeLengths iss
      = [3 1 2 1]

xss1 = replicates ns xss
     = [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]

iss2 = concat iss
     = [1     0     1     2       1     0     0]

xss2 = index_l xss1 iss2
     = [B     A     B     E       G     F     H]

res  = unconcat iss xss2
     = [[B A B] [E] [G F] [H]]
```

```
 retrieve [[A B]   [C D E] [F G] [H]]     (xss)
          [[1 0 1] [2]     [1 0] [0]]     (iss)
     ==>  [[B A B] [E]     [G F] [H]]

 ns   = takeLengths iss
      = [3 1 2 1]

xss1 = replicates ns xss
     = [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]

iss2 = concat iss
     = [1     0     1     2       1     0     0]

xss2 = index_l xss1 iss2
     = [B     A     B     E       G     F     H]

res  = unconcat iss xss2
     = [[B A B] [E] [G F] [H]]
```

```
xss1 = replicates ns xss
     = [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
```

```
xss1 = replicates [3       1        2      1]
                  [[A B]   [C D E] [F G] [H]]
     = [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
```

```
xss1 = replicates [3       1       2     1]
                  [[A B]   [C D E] [F G] [H]]
     = [[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
```

```
retrieve :: A (A Char) -> A (A Int) -> A (A Char)
retrieve xss iss
  = zipWithP mapP (mapP indexP xss) iss
```

partial application => sharing

array value

```
[[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
```

NESL-style array representation

segment descriptor

```
lengths:  [2 2 2 3 2 2 1]

indices:  [0 2 4 6 9 11 13]

   data:  [A B A B A B C D E F G F G H]
```

- NESL-style nested array representation cannot represent physical sharing of elements between among sub-arrays.

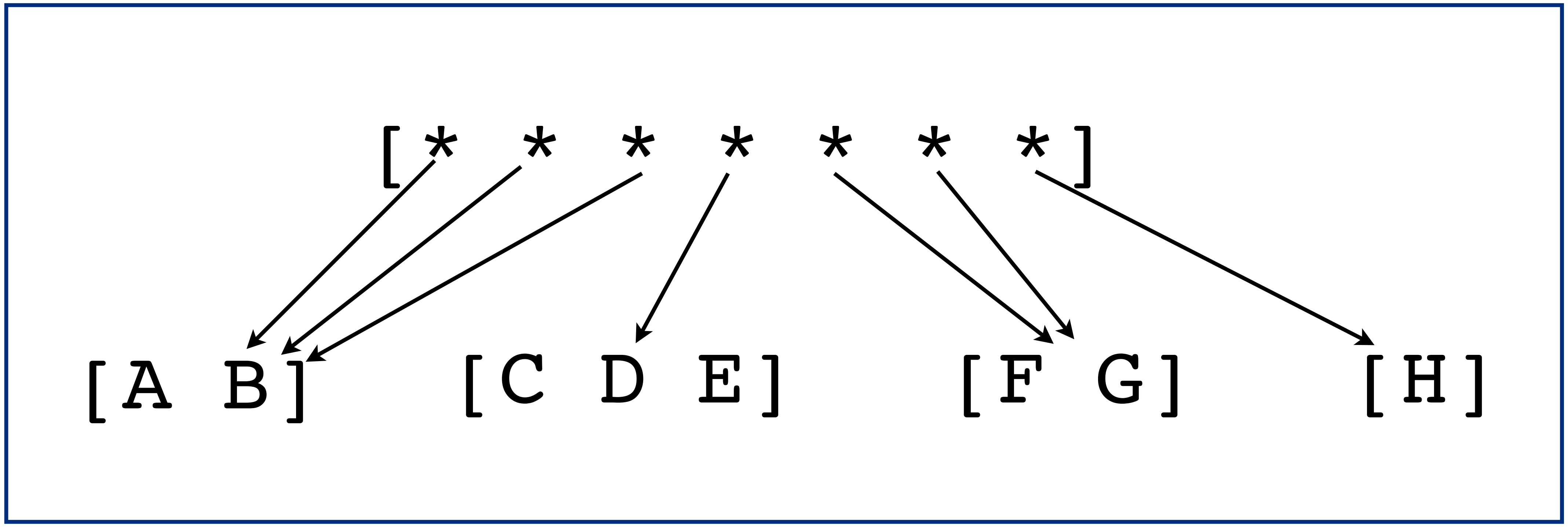
```
retrieve [[A B C D E F G H]] [[0 1 2 3 4 5 6 7]]
      ==> [A B C D E F G H]
```

- O(size) when using sequential lists (or arrays)
- O(size^2) when vectorised. **--- BAD!**

array value

`[[A B] [A B] [A B] [C D E] [F G] [F G] [H]]`

pointer-based array representation



- Naive pointer based representation has poor locality.
- Hard to distribute array across processors.

Complexity Goal

Suppose the source program is evaluated using pointer-based nested arrays.

The vectorised program should have the same asymptotic work complexity,
  but run in parallel,
  and be faster than the sequential version.

Operators

```
replicate     ::   Int    ->   e -> A e
replicates    :: A Int    -> A e -> A e
concat        :: A (A e) -> A e
unconcat      :: A (A e) -> A e -> A (A e)
pack          :: A Bool   -> A e -> A e
combine2      :: A Bool   -> A e -> A e -> A e
index         :: Int      -> A e -> e
index_l       :: A (A e) -> A Int    -> A e
append        :: A e      -> A e      -> A e
append_l      :: A (A e) -> A (A e) -> A (A e)
```

Operators

	Pointer Based	Old DPH (NESL Style)
`replicate`	O(length result)	O(**size** result)
`replicates`	O(max (len source, len result))	O(max (len source, **size** result))
`concat`	O(max (len source, len result))	O(1)
`unconcat`	O(len result)	O(1)
`pack`	O(len source)	O(max (len source, **size** result))
`combine2`	O(len result)	O(**size** result)
`index`	O(1)	O(1)
`index_l`	O(len result)	O(max (len source, **size** result))
`append`	O(len result)	O(**size** result)
`append_l`	O(len (concat result))	O(**size** (concat result))

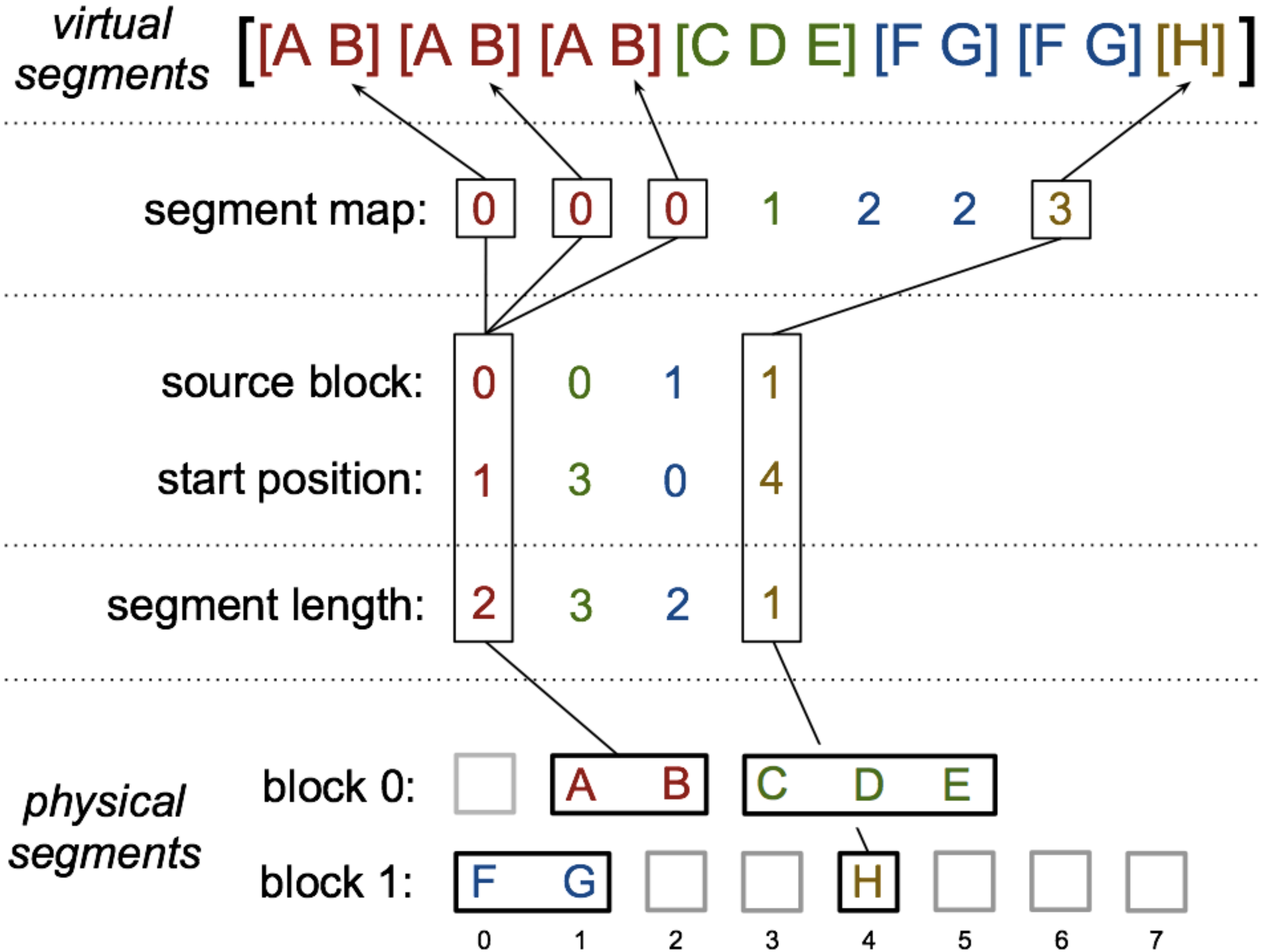
You don’t need O(1) concat

User written source code.

```
retrieve :: A (A Char) -> A (A Int) -> A (A Char)
retrieve xss iss
  = zipWithP mapP (mapP indexP xss) iss
```

Vectorised data-parallel version.

```
retrieve_v :: A (A Char) -> A (A Int) -> A (A Char)
retrieve_v xss iss
  = let ns = takeLengths iss
    in  unconcat iss
          $ index_l (replicates ns xss)
          $ concat iss
```



virtual segments
[[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
segment map:
0
0
0
1
2
2
3
source block:
0
0
1
1
start position:
1
3
0
4
segment length:
2
3
2
1
physical segments
block 0:
A
B
C
D
E
block 1:
F
G
H
0
1
2
3
4
5
6
7

Operators

	Pointer Based	Old DPH (NESL Style)
`replicate`	O(len result)	O(**size** result)
`replicates`	O(max (len source, len result))	O(max (len source, **size** result))
`concat`	O(max (len source, len result))	O(1)
`unconcat`	O(len result)	O(1)
`pack`	O(len source)	O(max (len source, **size** result))
`combine2`	O(len result)	O(**size** result)
`index`	O(1)	O(1)
`index_l`	O(len result)	O(max (len source, **size** result))
`append`	O(len result)	O(**size** result)
`append_l`	O(len (concat result))	O(**size** (concat result))

Operators

	Pointer Based	New DPH
`replicate`	O(len result)	O(len result)
`replicates`	O(max (len source, len result))	O(max (len source, len result))
`concat`	O(max (len source, len result))	O(max (len source, len result))
`unconcat`	O(len result)	O(len result)
`pack`	O(len source)	O(len source)
`combine2`	O(len result)	O(len result)
`index`	O(1)	O(1) / **O(len result) for nested result**
`index_l`	O(len result)	O(max (len source, len result))
`append`	O(len result)	O(len result)
`append_l`	O(len (concat result))	O(len (concat result))

Operators

	Pointer Based	New DPH
`replicate`	O(len result)	O(len result)
`replicates`	O(max (len source, len result))	O(max (len source, len result))
`concat`	O(max (len source, len result))	O(max (len source, len result))
`unconcat`	O(len result)	O(len result)
`pack`	O(len source)	O(len source)
`combine2`	O(len result)	O(len result)
`index`	O(1)	O(1) / **O(len result) for nested result**
`index_l`	O(len result)	O(max (len source, len result))
`append`	O(len result)	O(len result)
`append_l`	O(len (concat result))	O(len (concat result))

All implemented using parallel vector operations!

In the paper

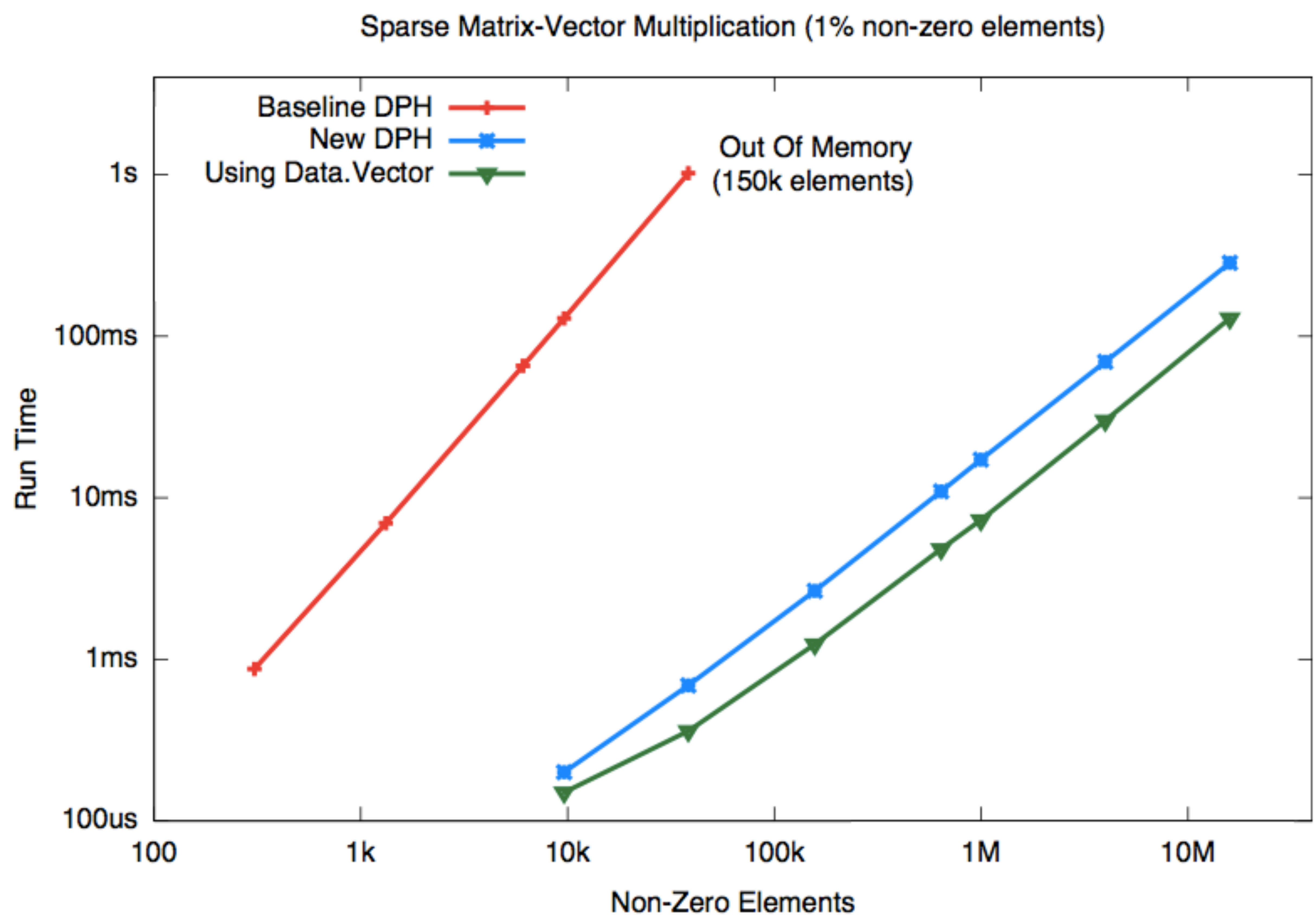
- Reference implementation of all operators.
- Other operators (besides replicates) also cause problems.
- Discussion of what complexity operators need to have.
- Invariants needed to maintain complexity.
- How to convert between old and new representations.
- How to avoid using extra descriptor fields when not needed.
- Example arrays and tests

Benchmarks

Sparse Matrix-Vector Multiplication

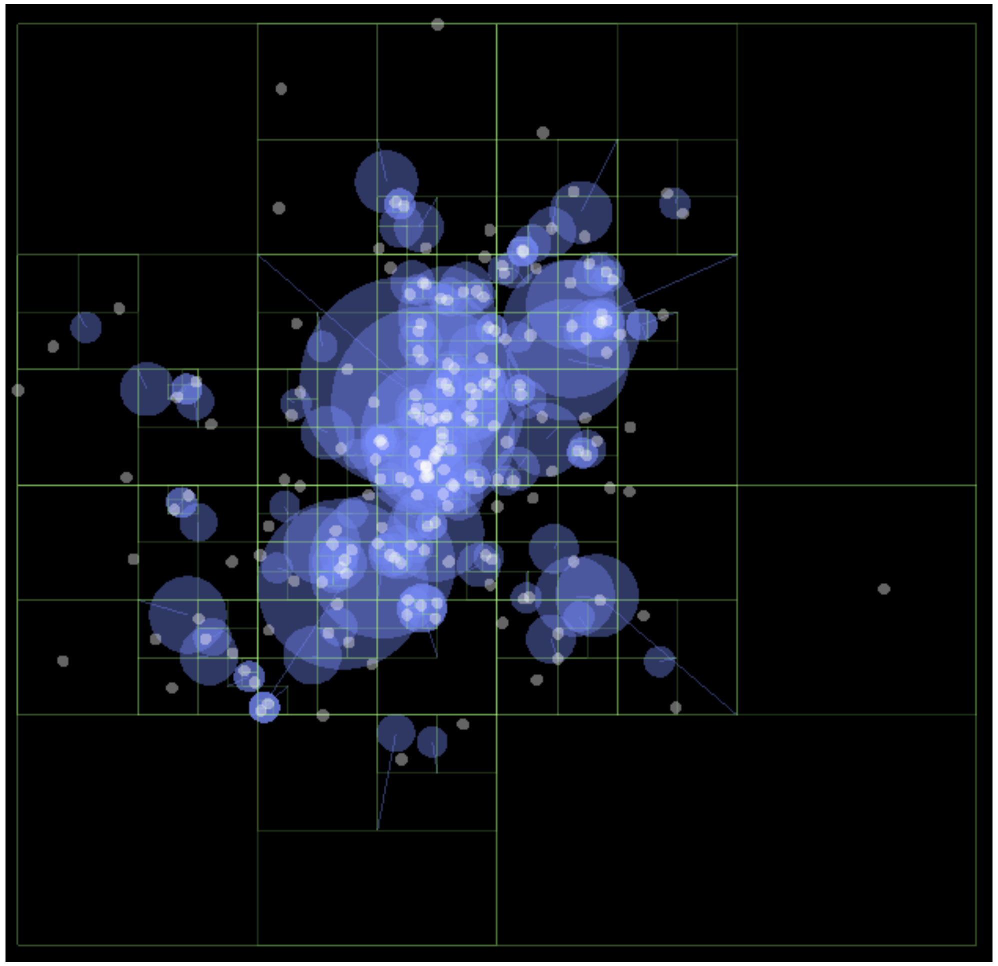
$$
\begin{bmatrix}
1 & 0 & 0 & 0 & 2 & 0 & 0 & 2 \\
2 & 0 & 0 & 0 & 0 & 5 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 3 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix}
=
\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix}
$$

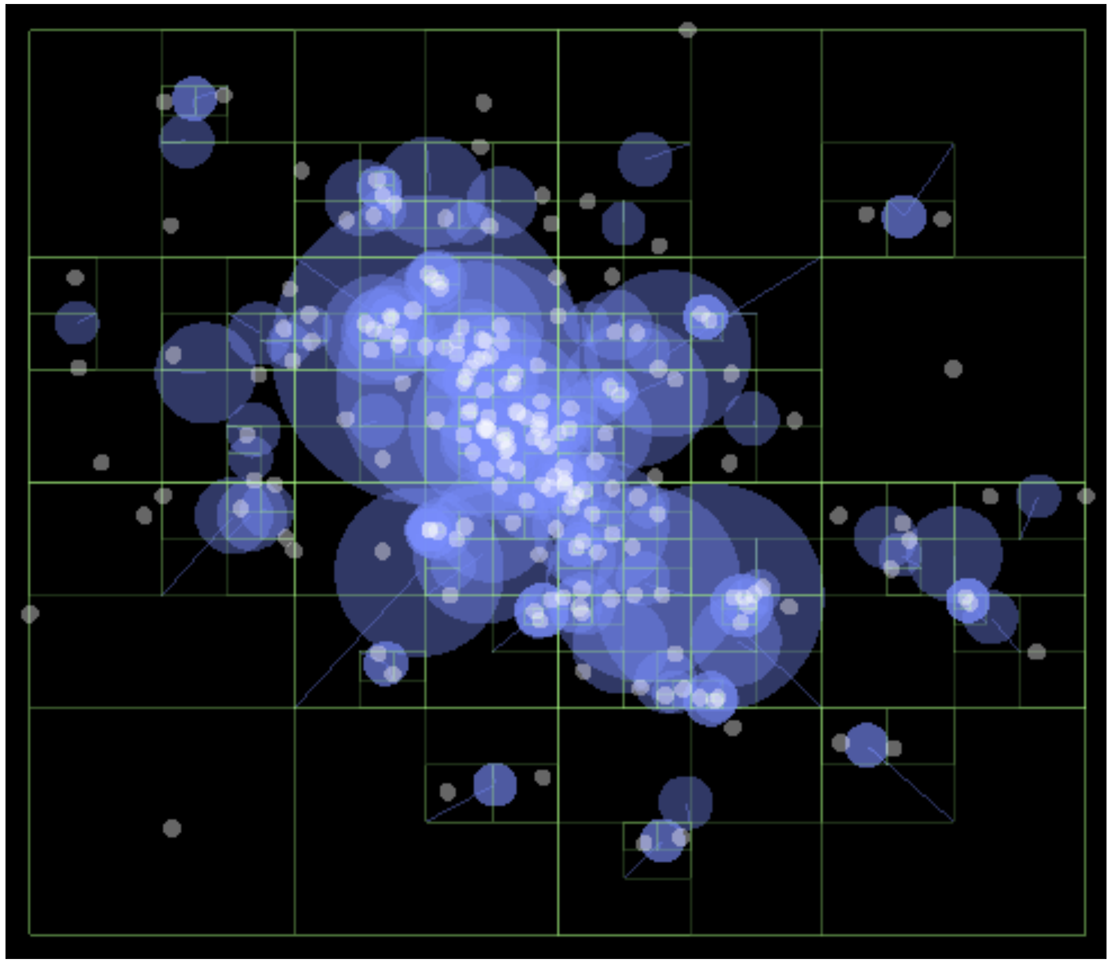
```
smvm :: A (A (Int, Double))
     -> A Double -> A Double

smvm matrix vector
 = let term (ix, coeff) = coeff * (vector ! ix)
   in  mapP (\row -> sumP (mapP term row)) matrix
```



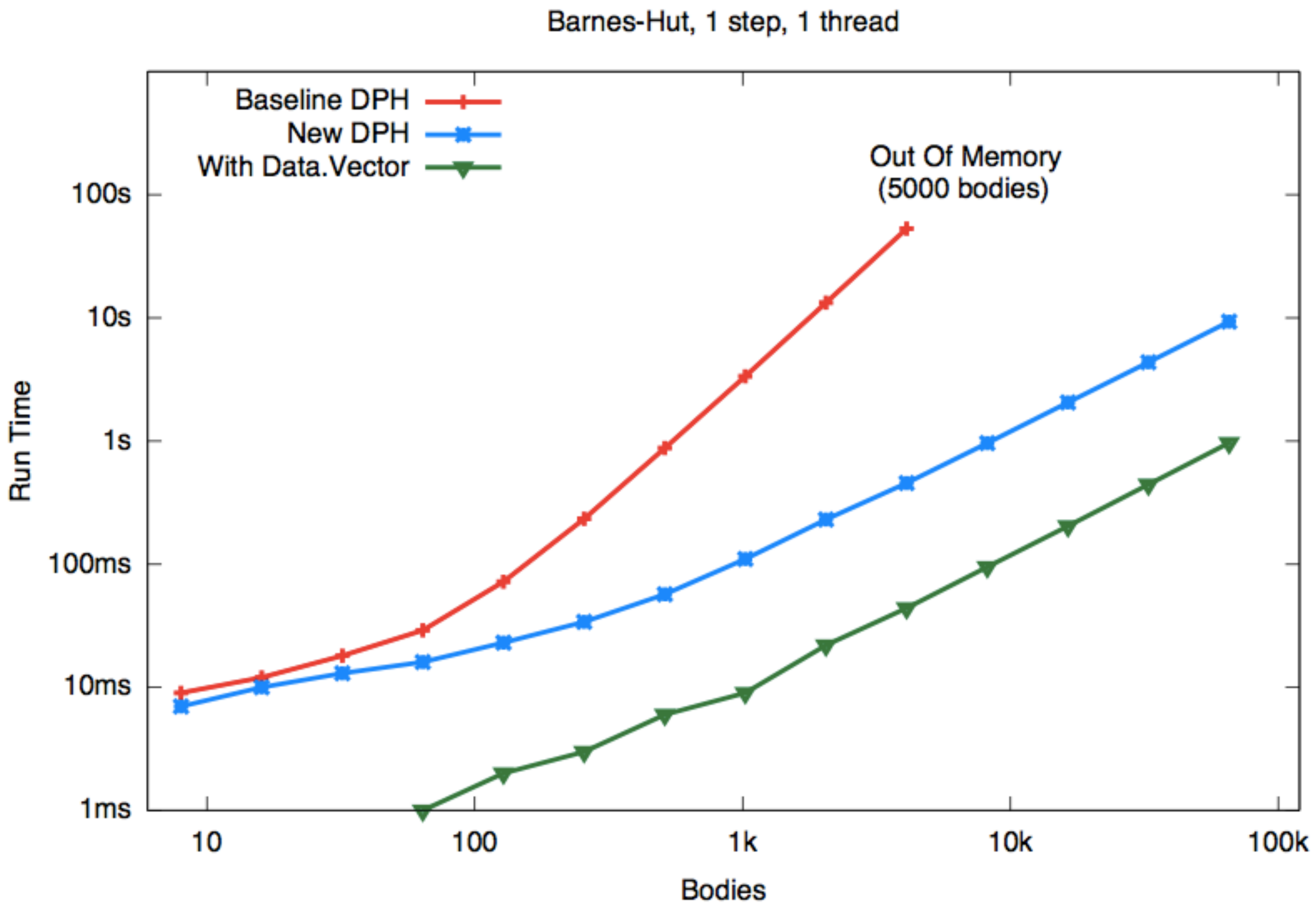
Sparse Matrix-Vector Multiplication (1% non-zero elements)
Baseline DPH
New DPH
Using Data.Vector
Out Of Memory
(150k elements)
Run Time
1s
100ms
10ms
1ms
100us
100
1k
10k
100k
1M
10M
Non-Zero Elements

Barnes-Hut Gravitation Simulation





```
calcAccels :: Double -> Box -> A MassPoint -> A Accel
calcAccels epsilon boundingBox points
 = mapP (\m -> calcAccel epsilon m tree) points
 where tree = buildTree boundingBox points
```


Barnes-Hut, 1 step, 1 thread
Baseline DPH
New DPH
With Data.Vector
Out Of Memory
(5000 bodies)
Run Time
100s
10s
1s
100ms
10ms
1ms
10
100
1k
10k
100k
Bodies

Things that need to be fixed

- Need array fusion for new representation.
- Integrate with vectorisation avoidance.
- Current parallel implementation is slower than sequential.
- Avoid performing fine grained computations in parallel.

Questions?

Spare Slides

Append Example

Operators

```
replicate     ::   Int    ->   e -> A e
replicates    :: A Int    -> A e -> A e
concat        :: A (A e) -> A e
unconcat      :: A (A e) -> A e -> A (A e)
pack          :: A Bool   -> A e -> A e
combine2      :: A Bool   -> A e -> A e -> A e
index         :: Int      -> A e -> e
index_l       :: A Int    -> A (A e) -> A e
append        :: A e      -> A e     -> A e
append_l      :: A (A e) -> A (A e) -> A (A e)
```

```
[[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
      segmap: [0 0 0 1 2 2 3]
source block: [0 0 1 1]
 start index: [1 3 0 4]
      length: [2 3 2 1]
    Blocks 0: [X A B C D E]
           1: [F G X X H X X X]
```

```
[[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
      segmap: [0 0 0 1 2 2 3]
source block: [0 0 1 1]
 start index: [1 3 0 4]
      length: [2 3 2 1]
    Blocks 0: [X A B C D E]
           1: [F G X X H X X X]

 [[K] [] [L M N O]]
      segmap: [0 1 2]
source block: [0 0 0]
 start index: [0 1 1]
      length: [1 0 4]
    Blocks 0: [K L M N O]
```

```
[[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
      segmap: [0 0 0 1 2 2 3]
source block: [0 0 1 1]
 start index: [1 3 0 4]
      length: [2 3 2 1]
    Blocks 0: [X A B C D E]
           1: [F G X X H X X X]

 [[K] [] [L M N O]]
      segmap: [0 1 2]
source block: [0 0 0]
 start index: [0 1 1]
      length: [1 0 4]
    Blocks 0: [K L M N O]

[[A B] [A B] [A B] [C D E] [F G] [F G] [H] [K] [] [L M N O]]
      segmap: [0 0 0 1 2 2 3 4 5 6]
source block: [0 0 1 1 2 2 2]
 start index: [1 3 0 4 0 1 1]
      length: [2 3 2 1 1 0 4]
    Blocks 0: [X A B C D E]
           1: [F G X X H X X X]
           2: [K L M N O]
```

```
[[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
      segmap: [0 0 0 1 5 5 6]
source block: [0 0 1 1 0 1 1 1 1]
 start index: [1 3 2 2 0 5 7 0 4]
      length: [2 3 2 2 1 2 1 2 1]
    Blocks 0: [X A B C D E]
           1: [F G X X H X X X]
```

```
 [[K] [] [L M N O]]
      segmap: [0 1 2]
source block: [0 0 0]
 start index: [0 1 1]
      length: [1 0 4]
    Blocks 0: [K L M N O]
```

```
[[A B] [A B] [A B] [C D E] [F G] [F G] [H] [K] [] [L M N O]]
      segmap: [0 0 0 1 5 5 6 7 8 9]
source block: [0 0 1 1 0 1 1 1 1 2 2 2]
 start index: [1 3 2 2 0 5 7 0 4 0 1 1]
      length: [2 3 2 2 1 2 1 2 1 1 0 4]
    Blocks 0: [X A B C D E]
           1: [F G X X H X X X]
           2: [K L M N O]
```

```
[[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
      segmap: [0 0 0 1 5 5 6]
source block: [0 0 1 1 0 1 1 1 1]
 start index: [1 3 2 2 0 5 7 0 4]
      length: [2 3 2 2 1 2 1 2 1]
    Blocks 0: [X A B C D E]
           1: [F G X X H X X X]

 [[K] [] [L M N O]]
      segmap: [0 1 2]
source block: [0 0 0]
 start index: [0 1 1]
      length: [1 0 4]
    Blocks 0: [K L M N O]
```

Invariant

All physical segment descriptors must be reachable from the virtual segment map

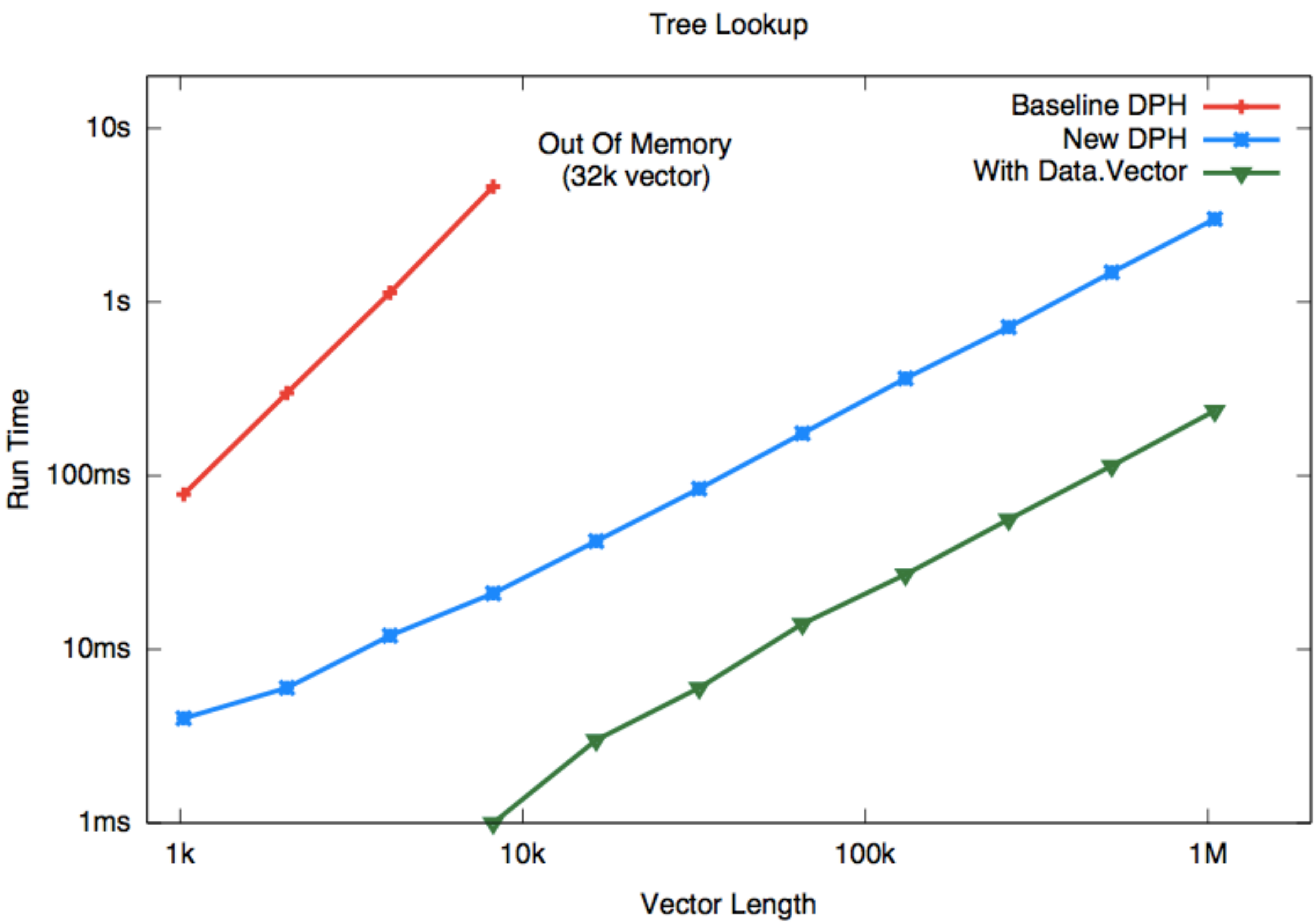
```
[[A B] [A B] [A B] [C D E] [F G] [F G] [H] [K] [] [L M N O]]
      segmap: [0 0 0 1 5 5 6 7 8 9]
source block: [0 0 1 1 0 1 1 1 1 2 2 2]
 start index: [1 3 2 2 0 5 7 0 4 0 1 1]
      length: [2 3 2 2 1 2 1 2 1 1 0 4]
    Blocks 0: [X A B C D E]
           1: [F G X X H X X X]
           2: [K L M N O]
```

Spare Slides

Tree Lookup Benchmark

Tree Reconstruction

```
treeLookup :: A Int -> A Int -> A Int
treeLookup table indices
  | lengthP indices == 1
  = [:table !: (indices !: 0):]

  | otherwise
  = let half = lengthP indices `div` 2
        s1   = sliceP 0    half indices
        s2   = sliceP half half indices
    in concatP (mapP (treeLookup table) [: s1, s2 :])
```



Tree Lookup
Baseline DPH
New DPH
With Data.Vector
Out Of Memory
(32k vector)
10s
1s
100ms
10ms
1ms
Run Time
1k
10k
100k
1M
Vector Length

Spare Slides

Index Space Overflow

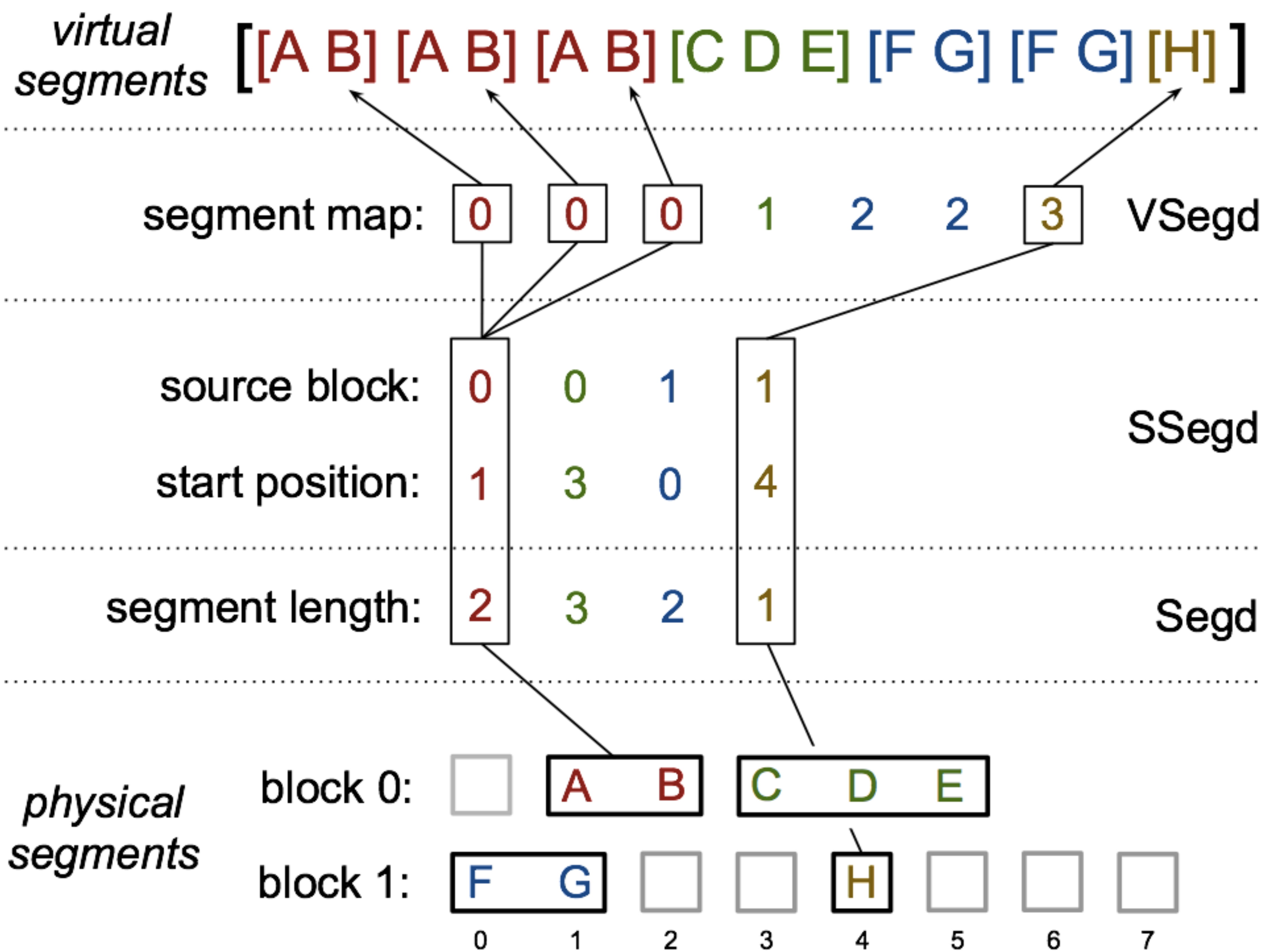
Index space overflow

```
retsum :: A (A Int) -> A (A Int) -> A (A Int)
retsum xss iss
 = zipWithP mapP
           (mapP (\xs i. indexP xs i + sumP xs) xss) iss
```

```
retsum [[1 2]   [4 5 6] [8]] (xss)
       [[1 0 1] [1 2]   [0]] (iss)
  ==>  [[5 4 5] [20 21] [16]]
```

Spare Slides

Rewriting to Simplified Segds


virtual segments
[[A B] [A B] [A B] [C D E] [F G] [F G] [H]]
segment map:
0 0 0 1 2 2 3
VSegd
source block:
0 0 1 1
start position:
1 3 0 4
SSegd
segment length:
2 3 2 1
Segd
physical segments
block 0:
A B C D E
block 1:
F G H
0 1 2 3 4 5 6 7

Backing off to simpler segment descriptors

```
sum_vs :: VSegd -> PDatas Int -> PData Int

sum_s  :: Segd  -> PData  Int -> PData Int
```

```
RULE "sum_vs/promote"
  forall segd arr
  . sum_vs (promoteSSegd (promoteSegd  segd))
           (singletondPR arr)
  = sum_s segd arr
```

Spare Slides

Space Complexity

Space Complexity

```
furthest :: PA (Float, Float) -> Float
furthest ps = maxP (mapP (\p. maxP (mapP (dist p) ps)) ps)
```

```
furthest_v xs
 = let c = length xs
       xss' = replicate c xs
       ns   = lengths xss'
   in  max  $ max_l c
            $ unconcat xss'
            $ dist_l (U.sum ns)
                     (replicates ns xs) (concat xss')
```