Parallel Functional Programming
Lecture 9
Data Parallelism II

Mary Sheeran

(with thanks to Ben Lippmeier for borrowed slides)

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

DPH

Parallel arrays [: e :] (which can contain arrays)

DPH

Parallel arrays [: e :] (which can contain arrays)

Expressing parallelism = applying collective operations to parallel arrays

Note: demand for any element in a parallel array results in eval of all elements

DPH array operations

```
(!:) :: [:a:] -> Int -> a
sliceP :: [:a:] -> (Int,Int) -> [:a:]
replicateP :: Int -> a -> [:a:]
mapP :: (a->b) -> [:a:] -> [:b:]
zipP :: [:a:] -> [:b:] -> [:(a,b):]
zipWithP :: (a->b->c) -> [:a:] -> [:b:] -> [:c:]
filterP :: (a->Bool) -> [:a:] -> [:a:]
concatP :: [:[:a:]:] -> [:a:]
concatMapP :: (a -> [:b:]) -> [:a:] -> [:b:]
unconcatP :: [:[:a:]:] -> [:b:] -> [:[:b:]:]
transposeP :: [:[:a:]:] -> [:[:a:]:]
expandP :: [:[:a:]:] -> [:b:] -> [:b:]
combineP :: [:Bool:] -> [:a:] -> [:a:] -> [:a:]
splitP :: [:Bool:] -> [:a:] -> ([:a:], [:a:])
```

Parallel array comprehensions

```
[: forceOn p m l | p <- ps, isFar len l p :]
```

Examples

```
svMul :: [:(Int,Float):] -> [:Float:] -> Float
svMul sv v = sumP [: f*(v !: i) | (i,f) <- sv :]
```

```
smMul :: [:[:(Int,Float):]:] -> [:Float:] -> Float
smMul sm v = sumP [: svMul row v | row <- sm :]
```

Examples

```
svMul :: [:(Int,Float):] -> [:Float:] -> Float
svMul sv v = sumP [: f*(v !: i) | (i,f) <- sv :]
```

```
smMul :: [:[:(Int,Float):]:] -> [:Float:] -> Float
smMul sm v = sumP [: svMul row v | row <- sm :]
```

Nested data parallelism
Parallel op (svMul) on each row

Barnes Hut N-body simulation

Reduces cost from O(N^2) to O(N log N)

Uses octree to represent the hierarchical grouping of particles

Particles close to each other are grouped and their centre of gravity (centroid)
Is calculated.

When a particle with which they should interact is sufficiently far away, then the centroid can be used.

Usually done in 3D. This DPH example is in 2D (and slightly simplified), so uses quad tree.

The Barnes Hut paper is GREAT.
Barnes, Josh, and Hut Piet. "A heirarchical O(N logN) force-calculation algorithm."
Nature. 324. (1986)

http://www.nature.com/nature/journal/v324/n6096/pdf/324446a0.pdf
(Access when on a Chalmers computer)

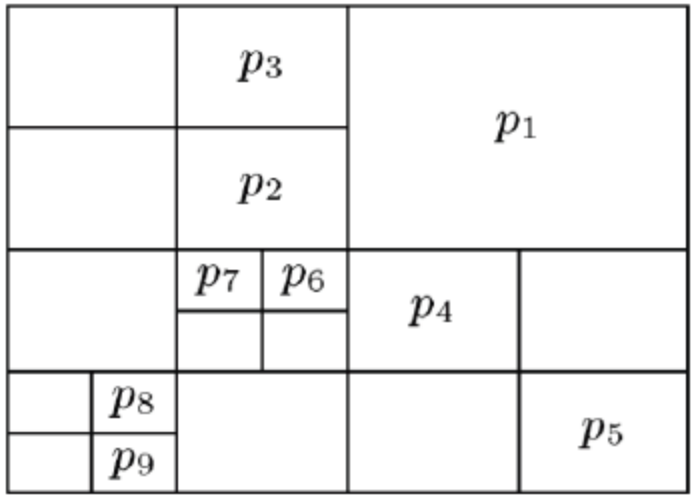
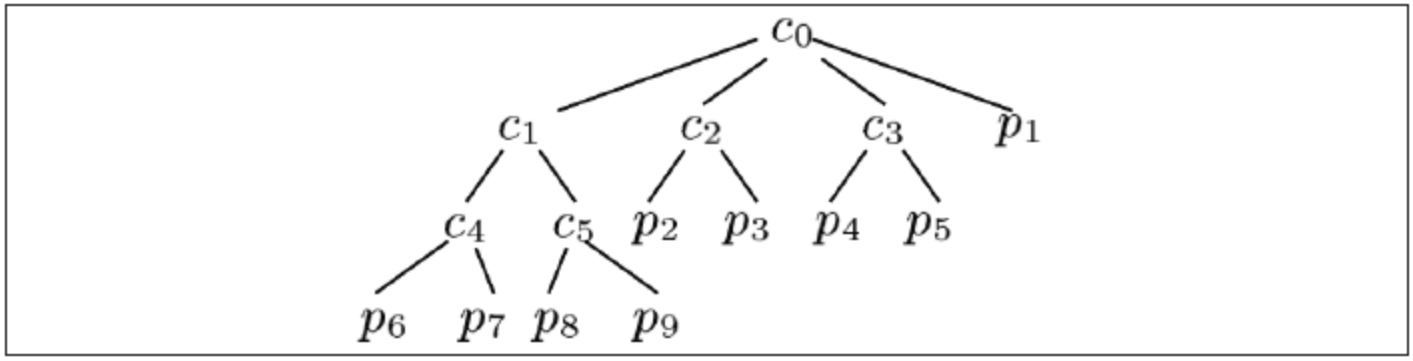
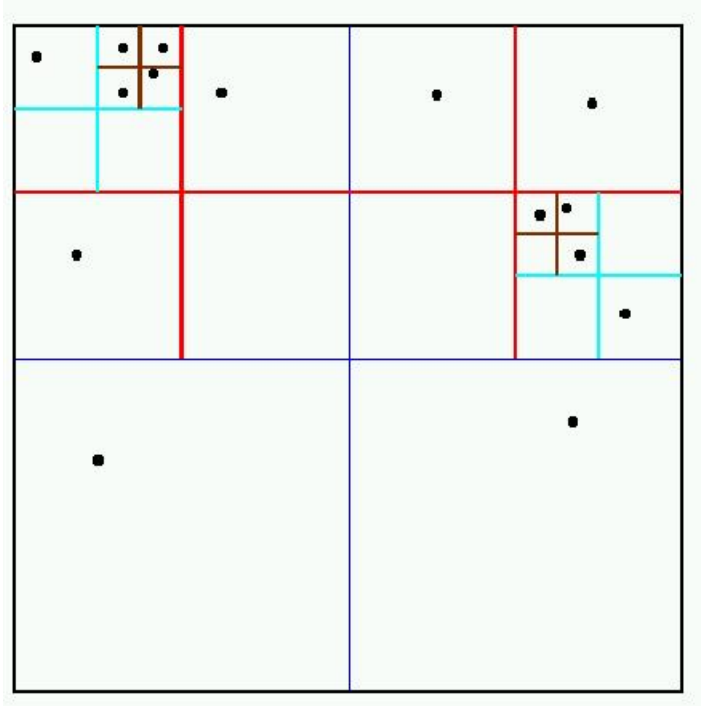
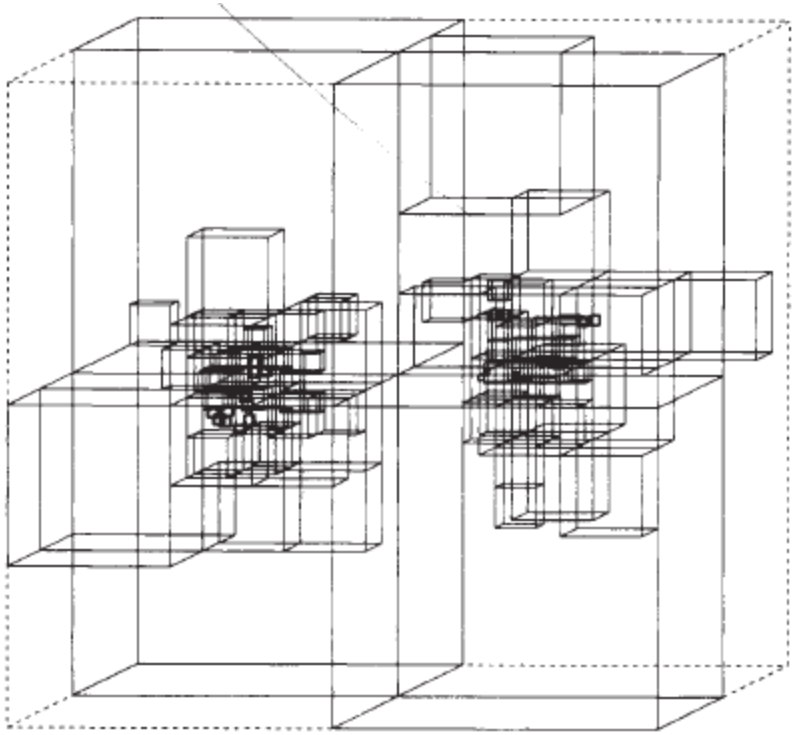


Figure 2: Subdivision of area







Having constructed such a tree, the force on any particle p may be approximated by a simple recursive calculation. Start at the root cell of the tree, which contains the entire system. Let l be the length of the cell currently being processed and D the distance from the cell's centre-of-mass to p. If $l/D < \theta$, where θ is a fixed accuracy parameter ~ 1, then include the interaction between this cell and p in the total being accumulated. Otherwise, resolve the current cell into its eight subcells, and recursively examine each one in turn. The core of the force calculation routine may be compactly expressed in SCHEME, a dialect of LISP:

```
(define (acceleration particle ensemble)
  (cond ((singleton?  ensemble)
         (newton-acceleration particle (the-element ensemble)))
        ((< (/ (diameter ensemble)
               (distance particle (centroid ensemble)))
           theta)
         (newton-acceleration particle (centroid ensemble)))
        (else
         (reduce sum-vector
                 (map (lambda (e) (acceleration particle e))
                      (subdivisions ensemble))))))
```

Barnes Hut (2D) in DPH

```
-- Compute one step of the n-body simulation
oneStep :: [:Particle:] -> [:Particle:]
oneStep particles = moveParticles particles forces
  where
    tree   = buildTree initialArea particles
    forces = calcForces (lengthOf initialArea) tree particles

buildTree     :: Area -> [:Particle:] -> Tree
calcForces    :: Float -> Tree -> [:Particle:] -> [:Force:]
moveParticles :: [:Particle:] -> [:Force:] -> [:Particle:]
lengthOf      :: Area -> Float
```

```
moveParticles :: [:Particle:] -> [:Force:] -> [:Particle:]
moveParticles ps fs = zipWithP moveParticle ps fs

moveParticle :: Particle -> Force -> Particle
moveParticle (Particle { mass     = m
                       , location = loc
                       , velocity = vel })
             force
  = Particle { mass     = m
             , location = loc + vel   * timeStep
             , velocity = vel + accel * timeStep }
  where
    accel = force / m
```

```
data Tree = Node Mass Location [:Tree:]
            -- Rose tree for spatial decomposition
```

```
data Tree = Node Mass Location [:Tree:]
          -- Rose tree for spatial decomposition
```

The only way to get parallelism over sub-trees

```
data Tree = Node Mass Location [:Tree:]
            -- Rose tree for spatial decomposition
```

```
-- Perform spatial decomposition and build the tree
buildTree :: Area -> [:Particle:] -> Tree
buildTree area [: p :]   = Node (mass p) (location p) [::]
buildTree area particles = Node m l subtrees
  where
    (m,l)    = calcCentroid subtrees
    subtrees = [: buildTree a ps
               | a <- splitArea area
               , let ps = [:p | p <- particles, inArea a p:]
               , lengthP ps > 0 :]
```

```
data Tree = Node Mass Location [:Tree:]
            -- Rose tree for spatial decomposition
```

```
-- Perform spatial decomposition and build the tree
buildTree :: Area -> [:Particle:] -> Tree
buildTree area [: p :]    = Node (mass p) (location
buildTree area particles = Node m l subtrees
  where
    (m,l)     = calcCentroid subtrees
    subtrees = [: buildTree a ps
                | a <- splitArea area
                , let ps = [:p | p <- particles, inArea a p:]
                , lengthP ps > 0 :]
```

Up to 4 areas

```
data Tree = Node Mass Location [:Tree:]
            -- Rose tree for spatial decomposition
```

```
-- Perform spatial decomposition and build the tree
buildTree :: Area -> [:Particle:] -> Tree
buildTree area [: p :]    = Node (mass p) (location p) [::]
buildTree area particles = Node m l subtrees
  where
    (m,l)     = calcCentroid subtrees
    subtrees = [: buildTree a ps
                | a <- splitArea area
                , let ps = [:p | p <- particles, inArea a p:]
                , lengthP ps > 0 :]
```

Tons of parallelism!

1) From recursive calls of parallel function buildTree
2) From nested parallel arrays

```
calcForces :: Float -> Tree -> [:Particle:] -> [:Force:]
calcForces len (Node m l ts) ps
  = let
      far_forces     = [: forceOn p m l | p <- ps, isFar len l p :]
      near_ps        = [: p | p <- ps, not (isFar len l p) :]
      near_forces_s  = [: calcForces (len / 2) t near_ps | t <- ts :]
      near_forces    = [: sumForces p_forces
                        | p_forces <- transposeP near_forces_s :]
    in
    combineP [:isFar len l p | p <- ps:] far_forces near_forces
```

Performance

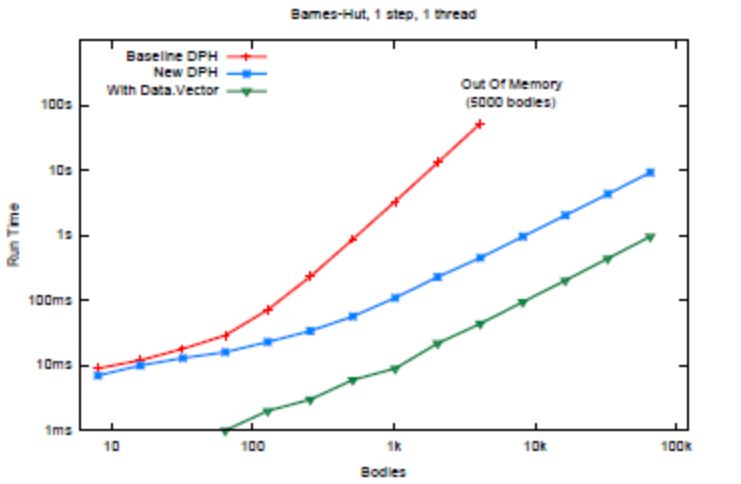


Figure 6. Benchmark Runtime Performance

Summary of example

Nestedness is essential in this example

Feels like just replacing [] by [: :] but authors caution that deciding on parallelisation needs thought and has influence on communication needed

Doesn’t yet run faster than using Data.Vector, but getting there!

Data parallelism

Perform *same* computation on a collection of *differing* data values

examples: HPF (High Performance Fortran)
CUDA

Both support only flat data parallelism

Flat : each of the individual computations on (array) elements is sequential

those computations don't need to communicate

parallel computations don't spark further parallel computations

Regular, Shape-polymorphic, Parallel Arrays in Haskell

Gabriele Keller[†] Manuel M. T. Chakravarty[†] Roman Leshchinskiy[†]
Simon Peyton Jones[‡] Ben Lippmeier[†]

[†]Computer Science and Engineering, University of New South Wales
{keller,chak,rl,benl}@cse.unsw.edu.au

[‡]Microsoft Research Ltd, Cambridge
simonpj@microsoft.com

API for purely functional, collective operations over dense, rectangular, multi-dimensional arrays supporting shape polymorphism

ICFP 2010

Ideas

Purely functional array interface using collective (whole array) operations like map, fold and permutations can

- combine efficiency and clarity
- focus attention on structure of algorithm, away from low level details

Influenced by work on algorithmic skeletons based on Bird Meertens formalism

Provides shape polymorphism not in a standalone specialist compiler like SAC, but using the Haskell type system

terminology

Regular arrays

dense, rectangular, most elements non-zero

shape polymorphic

functions work over arrays of arbitrary dimension

terminology

Regular arrays

dense, rectan

shape polym

functions wo

note: the arrays are purely functional and immutable

All elements of an array are demanded at once -> parallelism

P processing elements, n array elements => n/P consecutive elements on each proc. element

But things moved on!

Repa from ICFP 2010 had ONE type of array (that could be either delayed or manifest, like in many EDSLs)

A paper from Haskell'11 showed efficient parallel stencil convolution

http://www.cse.unsw.edu.au/~keller/Papers/stencil.pdf

Fancier array type

```
data Array sh a
   = Array       { arrayExtent  :: sh
                 , arrayRegions :: [Region sh a] }
data Region sh a
   = Region      { regionRange  :: Range sh
                 , regionGen    :: Generator sh a }
data Range sh
   = RangeAll
   | RangeRects  { rangeMatch   :: sh -> Bool
                 , rangeRects   :: [Rect sh] }
data Rect sh
   = Rect sh sh

data Generator sh a
   = GenManifest { genVector    :: Vector a }

   | forall cursor.
     GenCursored { genMake      :: sh -> cursor
                 , genShift     :: sh -> cursor -> cursor
                 , genLoad      :: cursor -> a }
```

Figure 5. New Repa Array Types

Fancier array type

```
data Array sh a
   = Array       { arrayExtent  :: sh
                 , arrayRegions :: [Region sh a] }
data Region sh a
   = Region      { regionRange  :: Range sh
                 , regionGen    :: Generator sh a }
data Range sh
   = RangeAll
   | RangeRects  { rangeMatch   :: sh -> Bool
                 , rangeRects   :: [Rect sh] }
data Rect sh
   = Rect sh sh

data Generator sh a
   = GenManifest { genVector    :

   | forall cursor.
     GenCursored { genMake
                 , genShift
```

But you need to be a guru to get good performance!

Put Array representation into the type!

The fundamental problem with Repa 1 & 2 is the following: at an particular point in the code, the programmer typically has a clear idea of the array representation they desire. For example, it may consist of three regions, left edge, middle, right edge, each of which is a delayed array. Although this knowledge is statically known to the the programmer, it is invisible in the types and only exposed to the compiler if very aggressive value inlining is used. Moreover, the programmer's typeless reasoning can easily fail, leading to massive performance degradation.

The solution is to expose static information about array representation to Haskell's main static reasoning system; its type system.

Repa 3 (Haskell'12)

Guiding Parallel Array Fusion with Indexed Types

Ben Lippmeier[†] Manuel M. T. Chakravarty[†] Gabriele Keller[†] Simon Peyton Jones[‡]

[†]Computer Science and Engineering
University of New South Wales, Australia
{benl,chak,keller}@cse.unsw.edu.au

[‡]Microsoft Research Ltd
Cambridge, England
{simonpj}@microsoft.com

Abstract

We present a refined approach to parallel array fusion that uses indexed types to specify the internal representation of each array. Our approach aids the client programmer in reasoning about the performance of their program in terms of the source code. It also makes the intermediate code easier to transform at compile-time, resulting in faster compilation and more reliable runtimes. We demonstrate how our new approach improves both the clarity and performance of several end-user written programs, including a fluid flow solver and an interpolator for volumetric data.

Categories and Subject Descriptors D.3.3 [*Programming Lan-*

This second version of `doubleZip` runs as fast as a hand-written imperative loop. Unfortunately, it is cluttered with explicit pattern matching, bang patterns, and use of the `force` function. This clutter is needed to guide the compiler towards efficient code, but it obscures the algorithmic meaning of the source program. It also demands a deeper understanding of the compilation method than most users will have, and in the next section, we will see that these changes add an implicit precondition that is not captured in the function signature. The second major version of the library, Repa 2, added support for efficient parallel stencil convolution, but at the same time also increased the level of clutter needed to achieve efficient code [8].

http://www.youtube.com/watch?v=YmZtP11mBho

quote on previous slide was from this paper

version

I use the latest Repa (3.2.3.3) (which works with the GHC that you get with the current Haskell platform)

cabal update

cabal install repa

http://hackage.haskell.org/packages/archive/repa/3.2.3.3/doc/html/Data-Array-Repa.html

(I don't have llvm installed. Using GHC's llvm backend speeds things up significantly, apparently.)

Repa Arrays

Repa arrays are wrappers around a linear structure that holds the element data.

The representation tag determines what structure holds the data.

Delayed Representations (functions that compute elements)
D -- Functions from indices to elements.
C -- Cursor functions.

Manifest Representations (real data)
U -- Adaptive unboxed vectors.
V -- Boxed vectors.
B -- Strict ByteStrings.
F -- Foreign memory buffers.

Meta Representations
P -- Arrays that are partitioned into several representations.
S -- Hints that computing this array is a small amount of work, so computation should be sequential rather than parallel to avoid scheduling overheads.
I -- Hints that computing this array will be an unbalanced workload, so computation of successive elements should be interleaved between the processors
X -- Arrays whose elements are all undefined.

10 Array representations!

- `D` – Delayed arrays (delayed) §3.1
- `C` – Cursored arrays (delayed) §4.4
- `U` – Adaptive unboxed vectors (manifest) §3.1
- `V` – Boxed vectors (manifest) §4.1
- `B` – Strict byte arrays (manifest) §4.1
- `F` – Foreign memory buffers (manifest) §4.1
- `P` – Partitioned arrays (meta) §4.2
- `S` – Smallness hints (meta) §5.1.1
- `I` – Interleave hints (meta) §5.2.1
- `X` – Undefined arrays (meta) §4.2

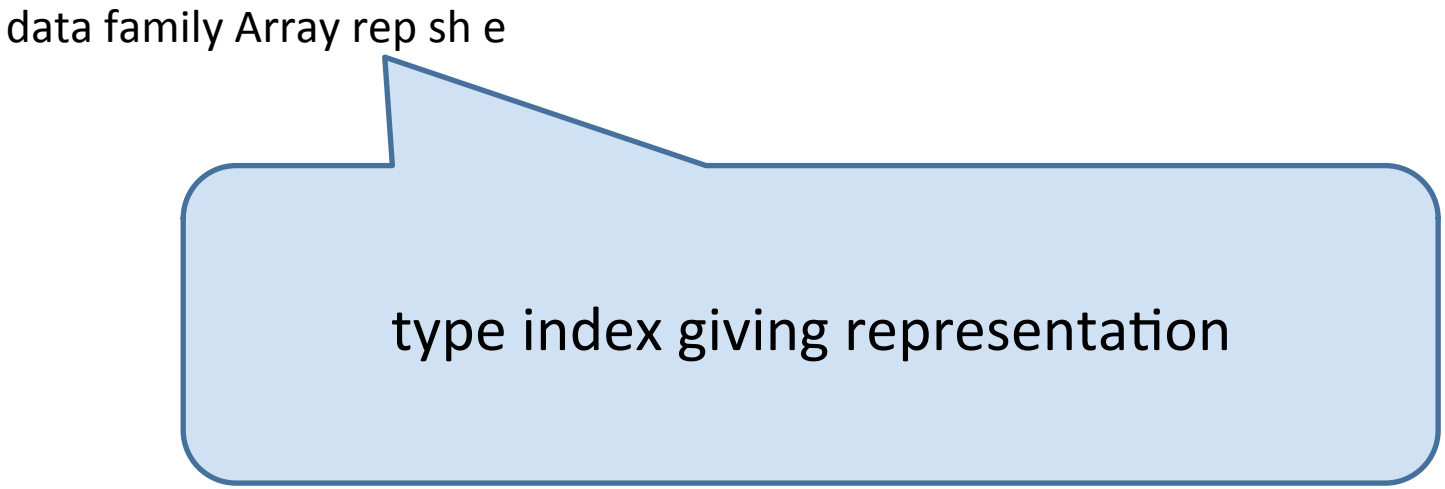
10 Array representations!

- `D` – Delayed arrays (delayed) §3.1
- `C` – Cursored arrays (delayed) §4.4
- `U` – Adaptive unboxed vectors (manifest) §3.1
- `V` – Boxed vectors (manifest) §4.1
- `B` – Strict byte arrays (manifest) §4.1
- `F` – Foreign memory buffers (manifest) §4.1
- `P` – Partitioned arrays (meta) §4.2
- `S` – Smallness hints (meta) §5.1.1
- `I` – Interleave hints (meta) §5.2.1
- `X` – Undefined arrays (meta) §4.2

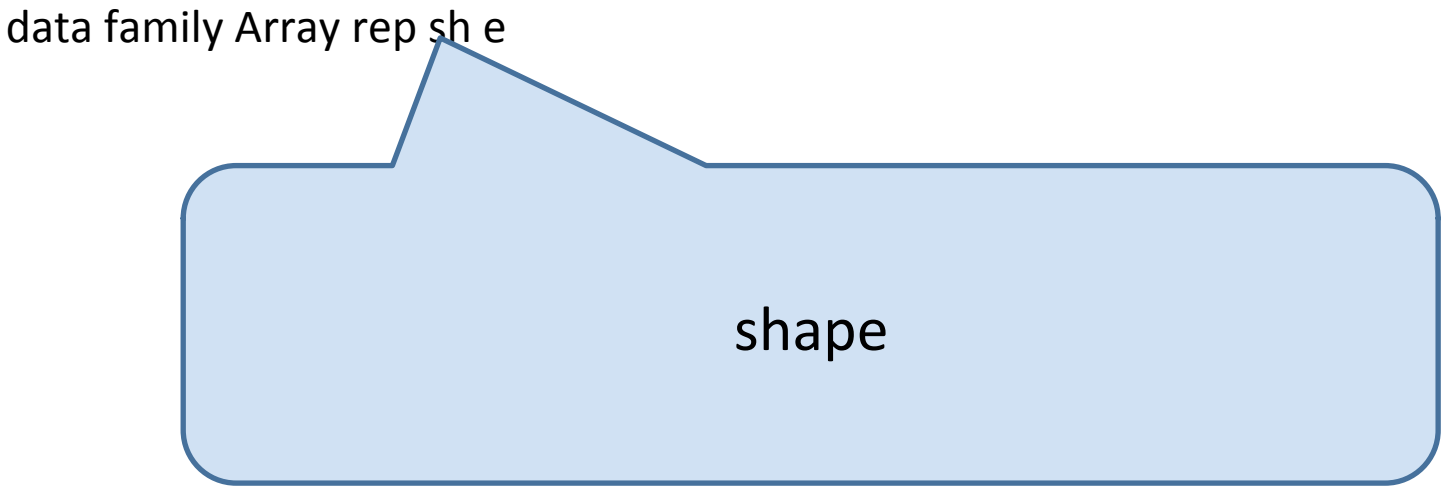
But the 18 minute presentation at Haskell'12 makes it all make sense!! Watch it!

http://www.youtube.com/watch?v=YmZtP11mBho

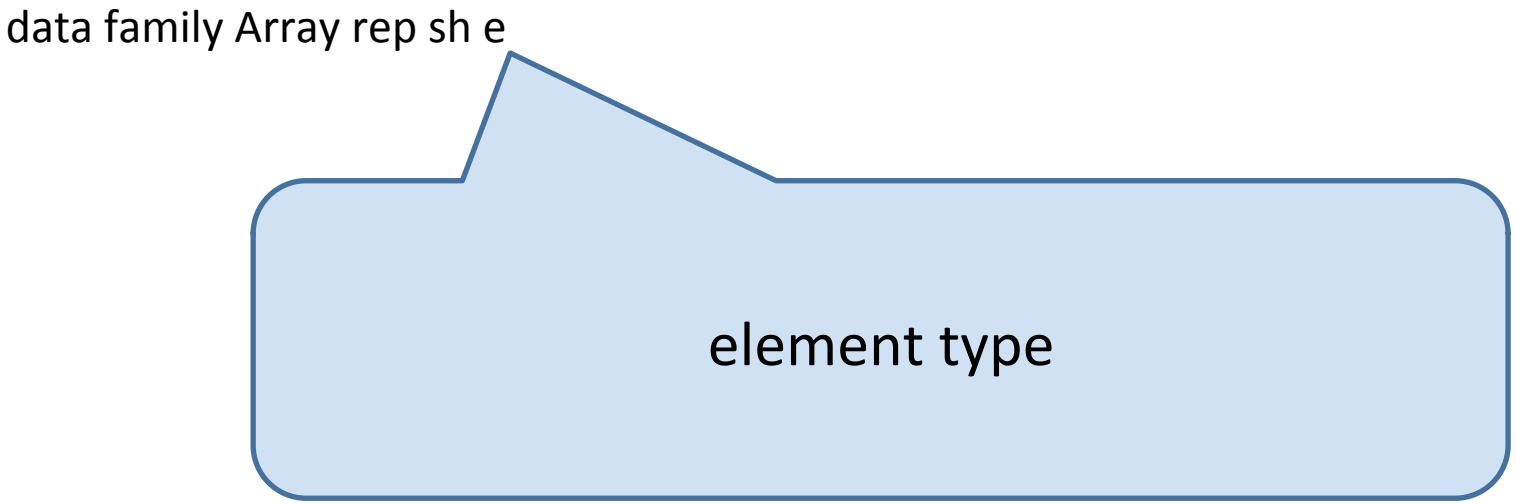
Type Indexing



Type Indexing



Type Indexing



map

```
map
:: (Shape sh, Source r a) =>
   (a -> b) -> Array r sh a -> Array D sh b
```

map

```
map
:: (Shape sh, Source r a) =>
   (a -> b) -> Array r sh a -> Array D sh b


map f arr = case delay arr of ADelayed sh g ->
                  ADelayed sh (f . g)
```

Fusion

Delayed (and cursored) arrays enable fusion that avoids intermediate arrays

User-defined worker functions can be fused

This is what gives tight loops in the final code

Parallel computation of array elements

```
computeP :: (Load r1 sh e, Target r2 e, Source r2 e, Monad m)
         => Array r1 sh e -> m (Array r2 sh e)
```

example

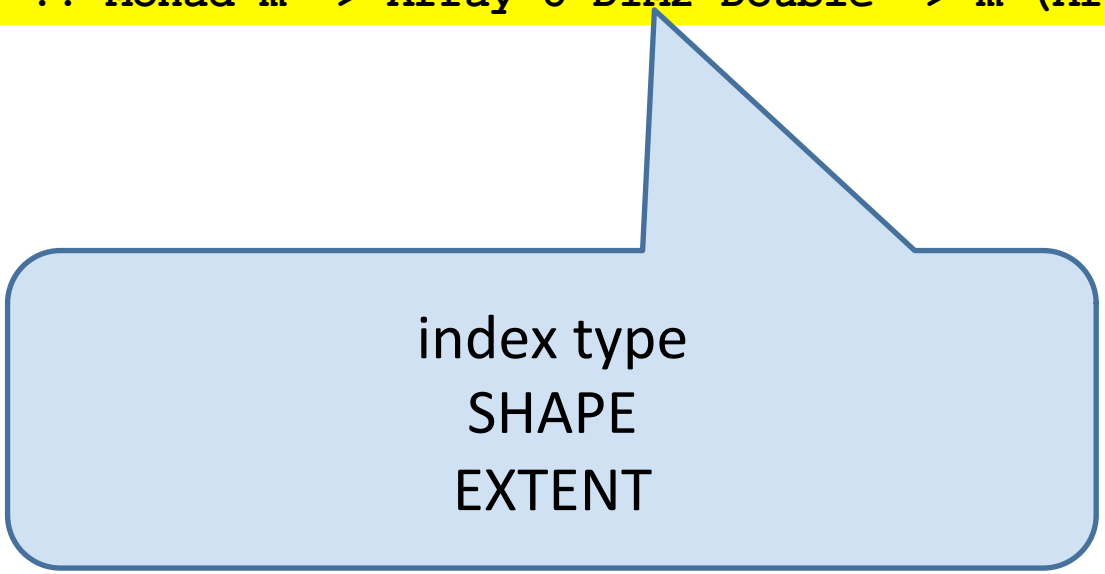
```
import Data.Array.Repa as R
```

```
transpose2P :: Monad m => Array U DIM2 Double -> m (Array U DIM2 Double)
```

example

```
import Data.Array.Repa as R
```

```
transpose2P :: Monad m => Array U DIM2 Double -> m (Array U DIM2 Double)
```



example

```
import Data.Array.Repa as R
```

```
transpose2P :: Monad m => Array U DIM2 Double -> m (Array U DIM2 Double)
```

DIM0 = Z (scalar)
DIM1 = DIM0 :. Int
DIM2 = DIM1 :. Int

snoc lists

Haskell lists are cons lists
1:2:3:[] is the same as [1,2,3]

Repa uses snoc lists at type level for shape types
and at value level for shapes

DIM2 = Z :. Int :. Int is a shape type

Z :. i :. j read as (i,j) is an index into a two dim. array

transpose 2D array in parallel

```
transpose2P
        :: Monad m
        => Array U DIM2 Double
        -> m (Array U DIM2 Double)

transpose2P arr
 = arr `deepSeqArray`
   do   computeUnboxedP
         $ unsafeBackpermute new_extent swap arr
 where  swap (Z :. i :. j)      = Z :. j :. i
        new_extent              = swap (extent arr)
```

more general transpose (on inner two dimensions)

```
transpose
  :: (Shape sh, Source r e) =>
     Array r ((sh :. Int) :. Int) e
     -> Array D ((sh :. Int) :. Int) e
```

more general transpose (on inner two dimensions) is provided

```
transpose
  :: (Shape sh, Source r e) =>
     Array r ((sh :. Int) :. Int) e
     -> Array D ((sh :. Int) :. Int) e
```

This type says an array with at least 2 dimensions.
The function is shape polymorphic

more general transpose (on inner two dimensions) is provided

```
transpose
  :: (Shape sh, Source r e) =>
     Array r ((sh :. Int) :. Int) e
     -> Array D ((sh :. Int) :. Int) e
```

Functions with at-least constraints become a parallel map over the unspecified dimensions (called rank generalisation)

Important way to express parallel patterns

Remember

Arrays of type (Array D sh a) or (Array C sh a) are *not real arrays*. They are represented as functions that compute each element on demand. You need to use computeS, computeP, computeUnboxedP and so on to actually evaluate the elements.

(quote from

http://hackage.haskell.org/packages/archive/repa/3.2.1.1/doc/html/Data-Array-Repa.html

which has lots more good advice, including about compiler flags)

Example: sorting

Batcher's bitonic sort

"hardware-like" data-independent

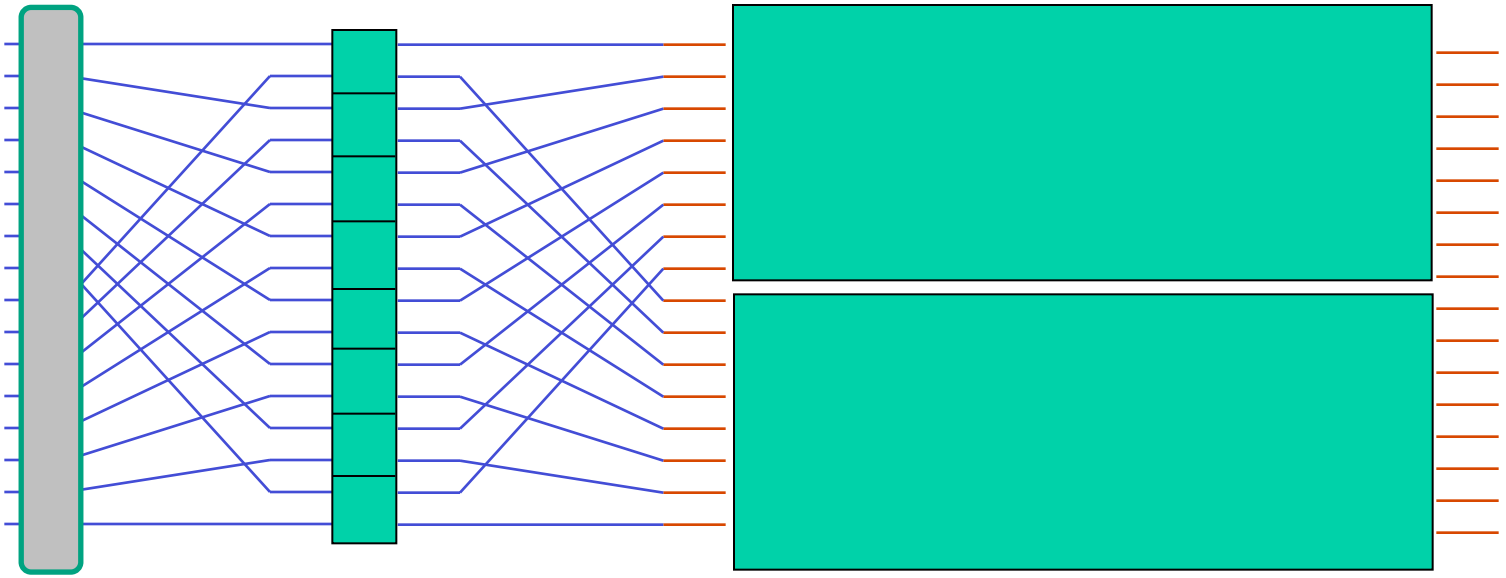
http://www.cs.kent.edu/~batcher/sort.pdf

bitonic sequence

inc (not decreasing)
then
dec (not increasing)

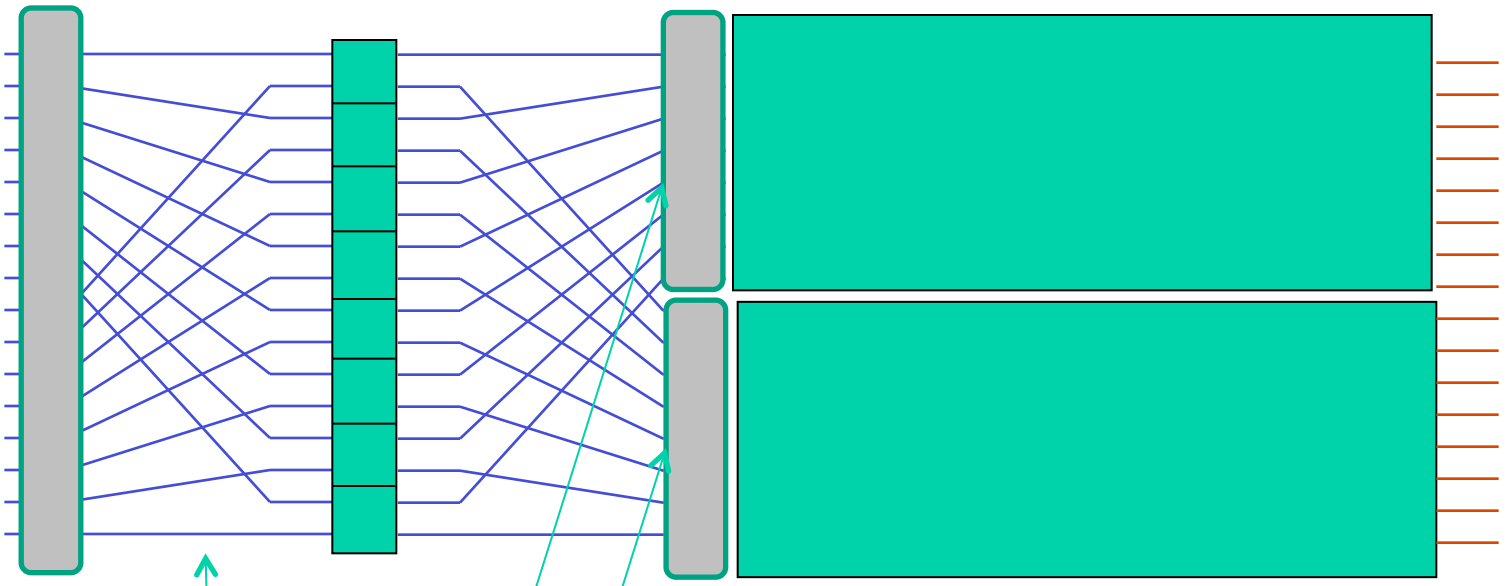
or a cyclic shift of such a sequence

Butterfly



bitonic

Butterfly



bitonic

bitonic

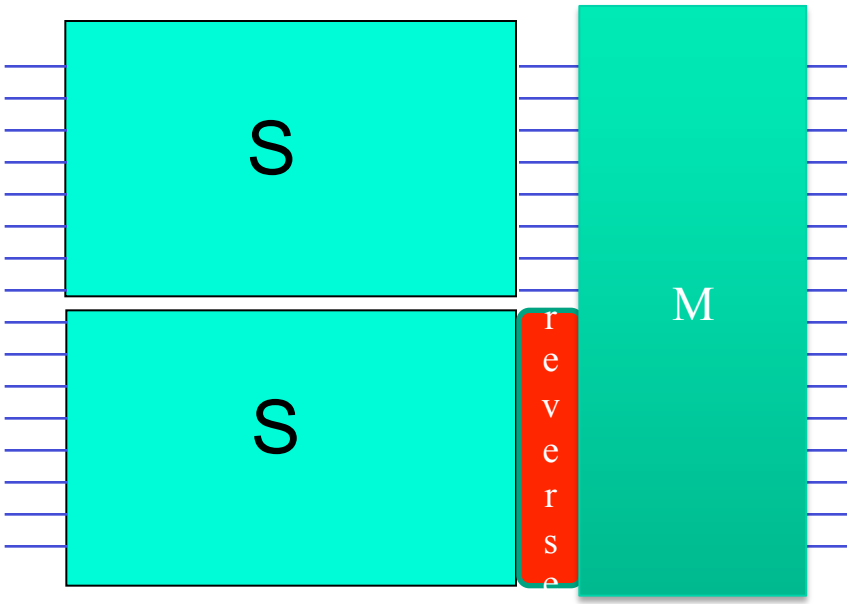
bitonic

>=

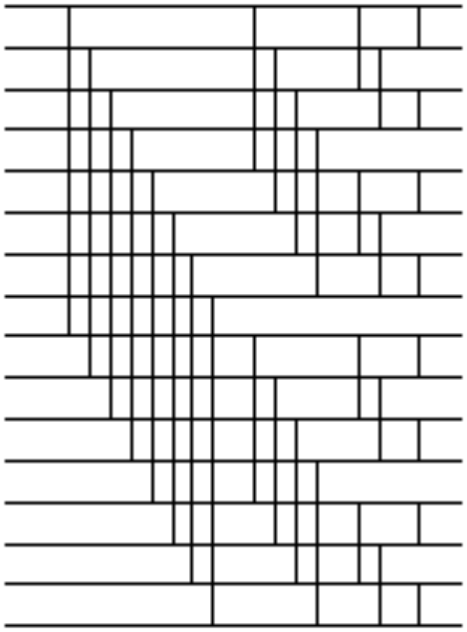
Making a recursive sorter (D&C)

Make a bitonic sequence using two half-size sorters

Batcher's sorter (bitonic)



bitonic merger



dee for diamond

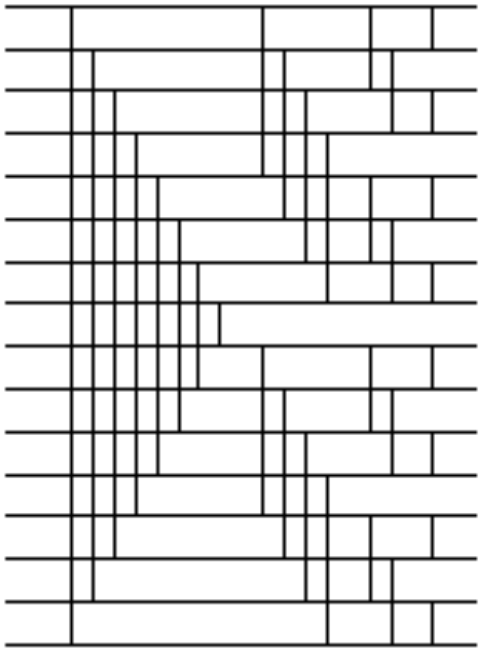
```
dee :: (Shape sh, Monad m) => (Int -> Int -> Int) -> (Int -> Int -> Int)
       -> Int -> Array U (sh :. Int) Int -> m (Array U (sh :. Int) Int)
dee f g s arr = let sh = extent arr in computeUnboxedP $ fromFunction sh ixf
  where
    ixf (sh :. i) = if (testBit i s) then (g a b) else (f a b)
      where
        a = arr ! (sh :. i)
        b = arr ! (sh :. (i `xor` s2))
        s2 = (1::Int) `shiftL` s
```

assume input array has length a power of 2, s > 0 in this and later functions

```
bitonicMerge
  :: (Monad m, Shape sh) =>
     Int -> Array U (sh :. Int) Int -> m (Array U (sh :. Int) Int)
bitonicMerge n = compose [dee min max (n-i) | i <- [1..n]]
```

```
compose :: Monad m => [a -> m a] -> a -> m a
compose [] arr = return arr
compose (f:fs) arr
  = do
      arr1 <- f arr
      compose fs arr1
```

tmerge



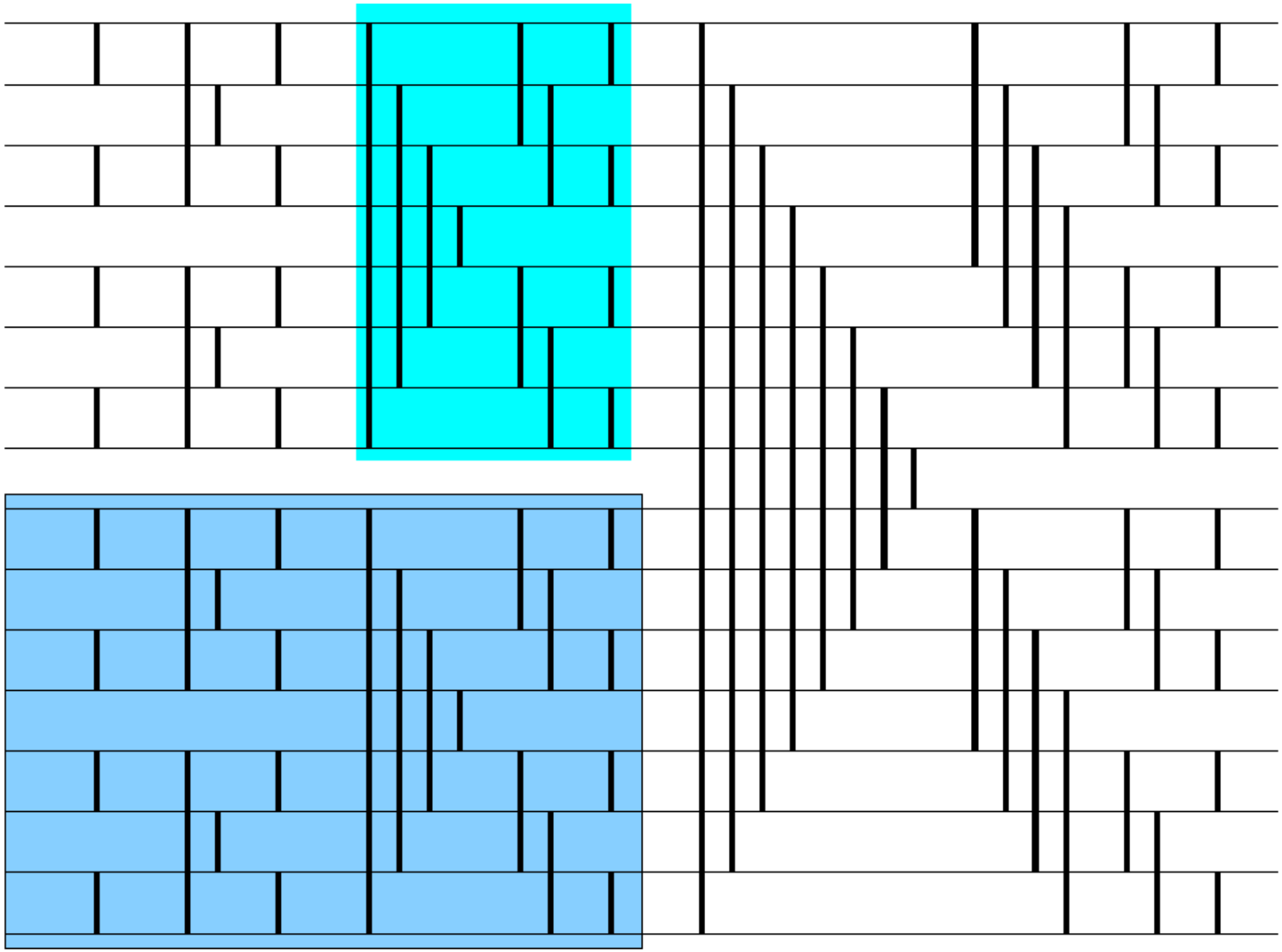
vee

```
vee :: (Shape sh, Monad m) => (Int -> Int -> Int) -> (Int -> Int -> Int)
        -> Int -> Array U (sh :. Int) Int -> m (Array U (sh :. Int) Int)
vee f g s arr = let (sh :. len)
   = extent arr in computeUnboxedP $ fromFunction (sh :. len) ixf
  where
    ixf (sh :. ix) = if (testBit ix s) then (g a b) else (f a b)
      where
        a = arr ! (sh :. ix)
        b = arr ! (sh :. newix)
        newix = flipLSBsTo s ix
```

tmerge

```
tmerge
  :: (Monad m, Shape sh) =>
     Int -> Array U (sh :. Int) Int -> m (Array U (sh :. Int) Int)


tmerge n = compose $ vee min max (n-1) : [dee min max (n-i) | i <- [2..n]]
```



```
tsort
  :: (Monad m, Shape sh) =>
     Int -> Array U (sh :. Int) Int -> m (Array U (sh :. Int) Int)


tsort n = compose [tmerge i | i <- [1..n]]
```

Performance is decent!

Initial benchmarking for 2^20 Ints

Around 744ms on 4 cores on this laptop

Compares to around 1.61 seconds for Data.List.sort (which is seqential)

Still slower than Persson's non-entry from the sorting competition in the 2012 course (which was at 400ms) -- a factor of a bit under 2, which is about what you would expect when comparing Batcher's bitonic sort to quicksort

Comments

Should be very scalable

Can probably be sped up! Need to add sequentialness ☺

Similar approach might greatly speed up the FFT in repa-examples
(and I found a guy running an FFT in Haskell competition)

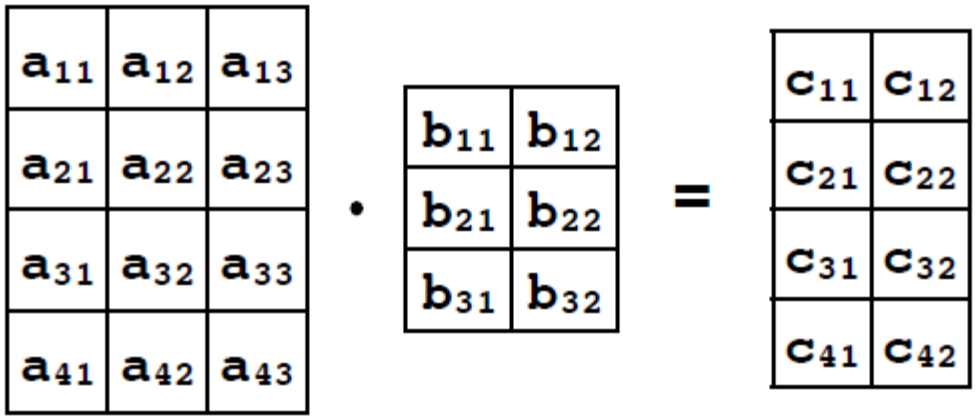
I wonder if more standard higher order functions (without bit hackery) could be made to work well (= fast) (zipWith, interleave etc.)

Note that this approach turned a nested algorithm into a flat one

Did you notice that I didn't mention scan ?? (Repa needs one!)

Study examples written by the master

Matrix Multiplication $(A.B)_{i,j} = \sum_{k=1}^{n} A_{i,k} \,.\, B_{k,j}$



slide from Lippmeier's ICFP 2010 talk on Repa

```
mmultP  :: Monad m
        => Array U DIM2 Double
        -> Array U DIM2 Double
        -> m (Array U DIM2 Double)

mmultP arr brr
 = [arr, brr] `deepSeqArrays`
   do   trr      <- transpose2P brr
        let (Z :. h1  :. _)  = extent arr
        let (Z :. _   :. w2) = extent brr
        computeP
         $ fromFunction (Z :. h1 :. w2)
         $ \ix   -> R.sumAllS
                  $ R.zipWith (*)
                        (unsafeSlice arr (Any :. (row ix) :. All))
                        (unsafeSlice trr (Any :. (col ix) :. All))
```

```
mmultP  :: Monad m
        => Array U DIM2 Double
        -> Array U DIM2 Double
        -> m (Array U DIM2 Double)

mmultP arr brr
 = [arr, brr] `deepSeqArrays`
   do   trr      <- transpose2P brr
        let (Z :. h1  :. _)  = extent arr
        let (Z :. _   :. w2) = extent brr
        computeP
         $ fromFunction (Z :. h1 :. w2)
         $ \ix   -> R.sumAllS
                  $ R.zipWith (*)
                        (unsafeSlice arr (Any :. (row ix) :. All))
                        (unsafeSlice trr (Any :. (col ix) :. All))
```

```
row :: DIM2 -> Int
row (Z :. r :. _) = r
```

stackoverflow

is your friend

See for example

http://stackoverflow.com/questions/14082158/idiomatic-option-pricing-and-risk-using-repa-parallel-arrays?rq=1

Conclusions

Based on DPH technology

Good speedups!

Neat programs

Good control of Parallelism

BUT CACHE AWARENESS needs to be tackled (see lecture later by Nick Frolov)

Array representations for parallel functional programming is an important, fun and frustrating research topic ☺ (see lecture on Monday next week)

Questions to think about

Can my bitonic sorter in Repa be sped up?

(I will put the code up on the web page.)

Can you implement a fast scan in Repa?

Next lecture (tomorrow)

Erlang!

Feel free to mail questions

MAKE USE of Nick! He knows a lot and is happy to guide you.