Data Parallel Programming II

Mary Sheeran

Example (as requested)

Associative non-commutative binary operator

Define

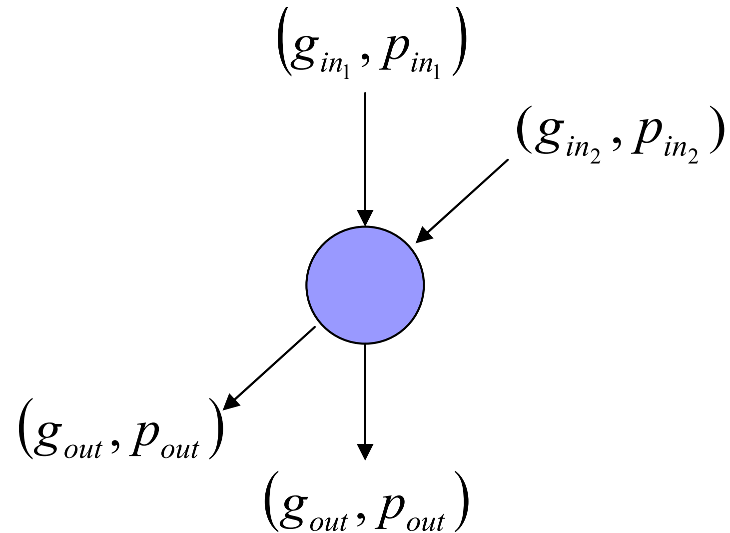
$a*b = a$

$a*(b*c) = a*b = a$

$(a*b) * c = a*c = a$

$a*b = a$
$b*a = b$

Another example from prefix adders



$$(g_{out}, p_{out}) = (g_{in1} + p_{in1} \cdot g_{in2}, p_{in1} \cdot p_{in2})$$

Part 1: simple language based performance model

Call-by-value λ-calculus

$$\lambda x.\, e \Downarrow \lambda x.\, e \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.\, e \quad e_2 \Downarrow v \quad e[v/x] \Downarrow v'}{e_1\, e_2 \Downarrow v'} \qquad \text{(APP)}$$

slide from Blelloch's ICFP10 invited talk

The Parallel λ-calculus: cost model

$$e \Downarrow v; w, d$$

Reads: expression *e* evaluates to *v* with work *w* and span *d.*

- **Work** (W): sequential work
- **Span** (D): parallel depth

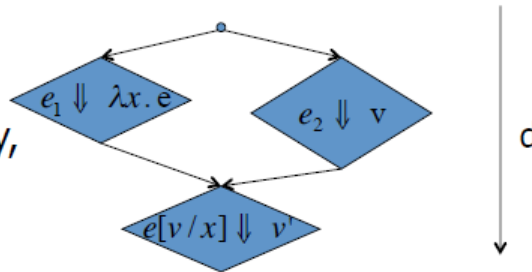
slide from Blelloch's ICFP10 invited talk

The Parallel λ-calculus: cost model

$$\lambda x.\mathrm{e} \Downarrow \lambda x.\mathrm{e}; 1,1 \qquad \text{(LAM)}$$

$$\frac{e_1 \Downarrow \lambda x.e; w_1, d_1 \quad e_2 \Downarrow v; w_2, d_2 \quad e[v/x] \Downarrow v'; w_3, d_3}{e_1\, e_2 \Downarrow v'; 1 + w_1 + w_2 + w_3, 1 + \max(d_1, d_2) + d_3} \qquad \text{(APP)}$$

Work adds
Span adds sequentially,
and max in parallel



slide from Blelloch's ICFP10 invited talk

Adding Functional Arrays: NESL

$$\{e_1 : x \text{ in } e_2 \mid e_3\}$$

$$\frac{e'[v_i / x] \Downarrow v_i';\ w_i, d_i \qquad i \in \{1 \ldots n\}}{\{e' : x \text{ in } [v_1 \ldots v_n]\} \Downarrow [v_1' \ldots v_n'];\ 1 + \sum_{i=1}^{n} w_i,\ 1 + \max_{i=1}^{|v|} d_i}$$

Primitives:

```
<- : 'a seq * (int,'a) seq -> 'a seq
• [g,c,a,p] <- [(0,d),(2,f),(0,i)]
     [i,c,f,p]
```

[ICFP95]

```
elt, index, length
```

slide from Blelloch's ICFP10 invited talk

Adding Functional Arrays: NESL

$$\{e_1 : x$$

$$\frac{e'[v_i / x] \Downarrow v_i'; w_i, d_i}{\{e' : x \text{ in } [v_1 \ldots v_n]\} \Downarrow [v_1' \ldots v_n']}$$

Arrays are purely functional (not mutable)

Primitives:

```
<- : 'a seq * (int,'a) seq -> 'a seq
• [g,c,a,p] <- [(0,d),(2,f),(0,i)]
     [i,c,f,p]

elt, index, length
```

[ICFP95]

slide from Blelloch's ICFP10 invited talk

Adding Functional Arrays: NESL

$$\{e_1 : x \text{ in } e_2 \mid e_3\}$$

Blelloch:
programming based cost models could change the way people think about costs and open door for other kinds of abstract costs
doing it in terms of machines.... "that's so last century"

```
<- : 'a seq * (int,'a) seq -> 'a seq
• [g,c,a,p] <- [(0,d),(2,f),(0,i)]
      [i,c,f,p]

elt, index, length
```

[ICFP95]

slide from Blelloch's ICFP10 invited talk

The Second Half: Provable Implementation Bounds

Theorem [FPCA95]:If $e \Downarrow v; w, d$ then v can be calculated from e on a CREW PRAM with p processors in $O\left(\frac{w}{p} + d \log p\right)$ time.

Can't really do better than: $\max\left(\frac{w}{p}, d\right)$

If w/p > d log p then "work dominates"

We refer to w/p as the parallelism.

d

(Typo fixed by MS based on the video)

slide from Blelloch's ICFP10 invited talk

Brent’s theorem

If a computation can be performed in d steps with w operations on a parallel computer (formally, a PRAM) with an unbounded number of processors, then the computation can be performed in

$d + (w-d)/p$ steps with p processors

using a greedy scheduler

http://maths-people.anu.edu.au/~brent/pd/rpb022.pdf

Proof

PROOF. Suppose that s_i operations are performed at step i, for $i = 1, 2, \cdots, t$. Thus $\sum_{i=1}^{t} s_i = q$. Using p processors, we can simulate step i in time $\lceil s_i/p \rceil$. Hence, the computation C can be performed with p processors in time

$$\sum_{i=1}^{t} \lceil s_i/p \rceil \leq (1 - 1/p)t + (1/p) \sum_{i=1}^{t} s_i = t + (q - t)/p.$$

from the aforementioned paper

Why?

s_i number of operations at time i

Time for s_i on p processors

$$\lceil s_i / p \rceil \leq \frac{s_i + p - 1}{p}$$

Overall time

$$\sum_{i=1}^{d} \lceil s_i / p \rceil$$

$$\leq \sum_{i=1}^{d} \frac{s_i + p - 1}{p} = \sum_{i=1}^{d} \frac{p}{p} + \sum_{i=1}^{d} \frac{s_i - 1}{p}$$

$$= \quad d + \frac{\sum_{i=1}^{d} s_i - \sum_{i=1}^{d} 1}{p}$$

$$= d + \frac{\sum_{i=1}^{d} s_i - \sum_{i=1}^{d} 1}{p}$$

$$= d + \frac{w - d}{p}$$

Now have both lower and upper bound on running time for p processors

$$\max(w/p, d) \leq T_p \leq d + \frac{w - d}{p}$$

Now have both lower and upper bound on running time for p processors

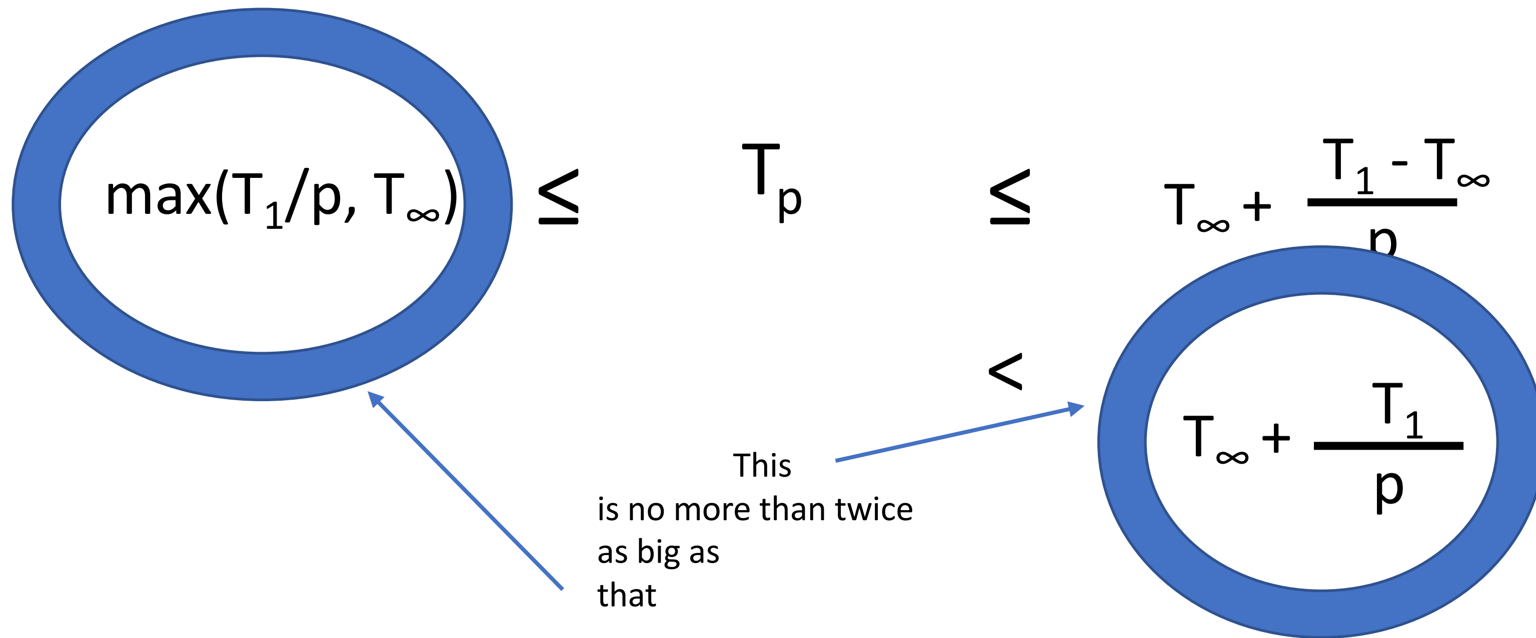
$$\max(T_1/p,\ T_\infty) \quad \leq \quad T_p \quad \leq \quad T_\infty + \frac{T_1 - T_\infty}{p}$$

Now have both lower and upper bound on running time for p processors

$$\max(T_1/p,\ T_\infty) \quad \leq \quad T_p \quad \leq \quad T_\infty + \frac{T_1 - T_\infty}{p}$$

$$< \quad T_\infty + \frac{T_1}{p}$$

Now have both lower and upper bound on running time for p processors



So a greedy scheduler does pretty close to the best possible

So a greedy scheduler does pretty close to the best possible

Though note that no real scheduler is perfect
will cause delays between task becoming ready and starting
(sometimes called scheduler friction)

can also cause memory effects. Movement of computations can cause additional data movement

Ideal scheduler => Rough but useful estimate

The Second Half: Provable Implementation Bounds

Theorem [FPC...]: If ... w, d then v can be calculated ...om e on a CR... PRAM with p processo... in $O\left(\frac{w}{p} + d \log p\right)$ tim...

Can't really do be... ...an: max...

If w/p > d log p then "work domi...

We refer to w/p as the parallelis...

The log p is related to load balancing cost

d

(Typo fixed by MS based on the video)

slide from Blelloch's ICFP10 invited talk

degree of parallelism

$$\frac{T_1}{T_\infty} \qquad \frac{w}{d}$$

An idea of how many processors we can usefully use

Why?

$$T_p < \frac{w}{p} + d$$

$$= \frac{w}{p} + \frac{w}{w/d}$$

$$= \frac{w}{p} \left(1 + \frac{p}{w/d} \right)$$

Why?

$$T_p < \frac{w}{p} + d$$

$$= \frac{w}{p} + \frac{w}{w/d}$$

$$= \frac{w}{p}\left(1 + \frac{p}{w/d}\right)$$

If p << degree of parallelism (w/d) then we get near perfect speedup

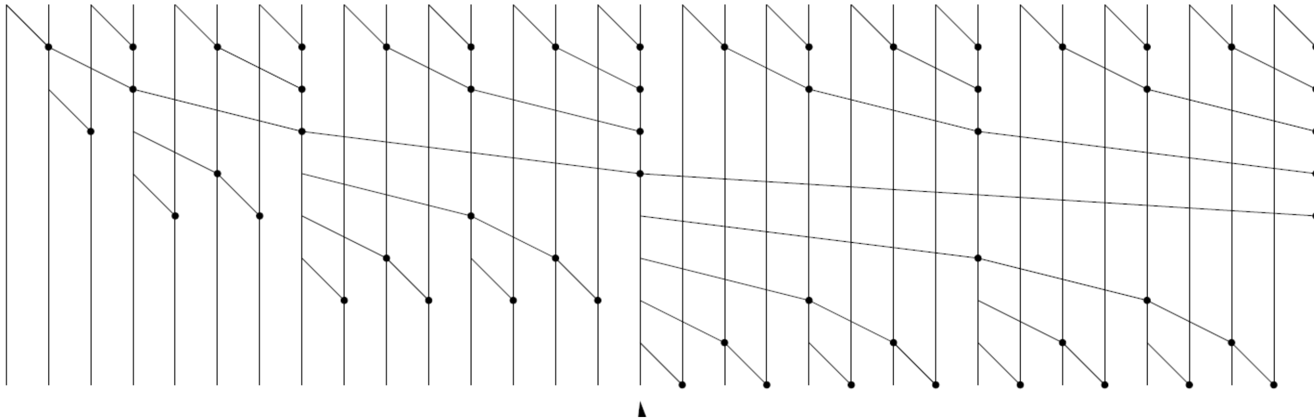
Work efficiency

Parallel algorithm is work efficient if it has work that is asymptotically the same as that of the optimal sequential algorithm

Aim first for low work

Then try to lower depth (span)

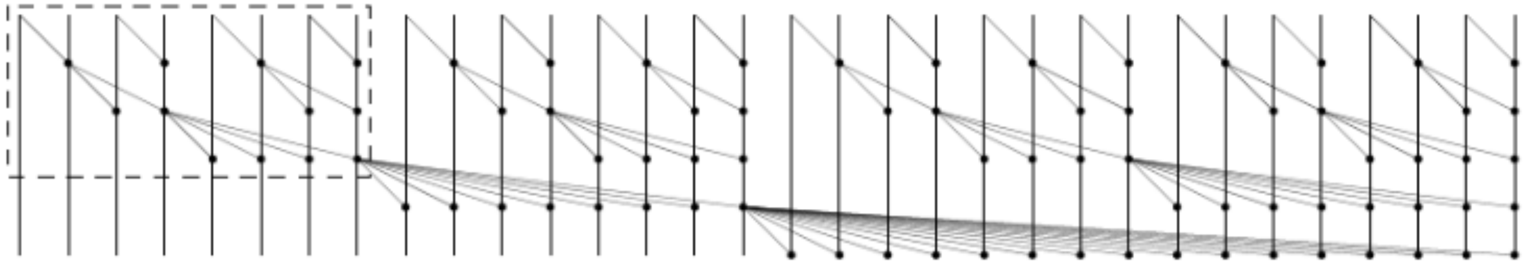
Back to our scan



oblivious or data independent computation

$N = 2^n$ inputs, work of dot is 1
work = ?
depth = ?

Another scan (Sklansky)



For N inputs Depth log N

What about work??

Quicksort

```
function Quicksort(A) = if (#A < 2) then A else
                let pivot = A[#A/2];
                    lesser = {e in A| e < pivot};
                    equal = {e in A| e == pivot};
                    greater = {e in A| e > pivot};
                    result = {quicksort(v): v in [lesser,greater]};
                in result[0] ++ equal ++ result[1];
```

Analysis in ICFP10 video gives depth = O(log N) work = O(N logN)

Quicksort

```
function Quicksort(A) = if (#A < 2) then A else
            let pivot = A[#A/2];
               lesser = {e in A| e < pivot};
               equal = {e in A| e == pivot};
               greater = {e in A| e > pivot};
               result = {quicksort(v): v in [lesser,greater]};
            in result[0] ++ equal ++ result[1];
```

Analysis in ICFP10 video gives depth = O(log N) work = O(N logN)

(The depth is improved over the example with trees, due to the addition of parallel arrays as primitive.)

From the NESL quick reference

Basic Sequence Functions

Basic Operations	Description	Work	Depth
#a	Length of a	O(1)	O(1)
a[i]	ith element of a	O(1)	O(1)
dist(a,n)	Create sequence of length n with a in each element.	O(n)	O(1)
zip(a,b)	Elementwise zip two sequences together into a sequence of pairs.	O(n)	O(1)
[s:e]	Create sequence of integers from s to e (not inclusive of e)	O(e-s)	O(1)
[s:e:d]	Same as [s:e] but with a stride d.	O((e-s)/d)	O(1)
Scans			
plus_scan(a)	Execute a scan on a using the + operator	O(n)	O(log n)
min_scan(a)	Execute a scan on a using the minimum operator	O(n)	O(log n)
max_scan(a)	Execute a scan on a using the maximum operator	O(n)	O(log n)
or_scan(a)	Execute a scan on a using the or operator	O(n)	O(log n)
and_scan(a)	Execute a scan on a using the and operator	O(n)	O(log n)

See the DIKU NESL interpreter!

Lesson 1: Sequential Semantics

- Debugging is much easier without non-determinism
- Analyzing correctness is much easier without non-determinism
- If it works on one implementation, it works on all implementations
- Some problems are inherently concurrent—these aspects should be separated

Slide borrowed from Blelloch's retrospective talk on NESL.
glew.org/damp2006/Nesl.ppt

Lesson 2: Cost Semantics

- Need a way to analyze cost, at least approximately, without knowing details of the implementation
- Any cost model based on processors is not going to be portable – too many different kinds of parallelism

Slide borrowed from Blelloch's retrospective talk on NESL.
glew.org/damp2006/Nesl.ppt

Lesson 3: Too Much Parallelism

Needed ways to back out of parallelism

- Memory problem
- The "flattening" compiler technique was too aggressive on its own
- Need for Depth First Schedules or other scheduling techiques
- Various bounds shown on memory usage

Slide borrowed from Blelloch's retrospective talk on NESL.
glew.org/damp2006/Nesl.ppt

NESL : what more should be done?

Take account of LOCALITY of data and
account for communication costs
(Blelloch has been working on this.)

Deal with exceptions and randomness

Reduce amount of parallelism where appropriate
(see Futhark lecture)

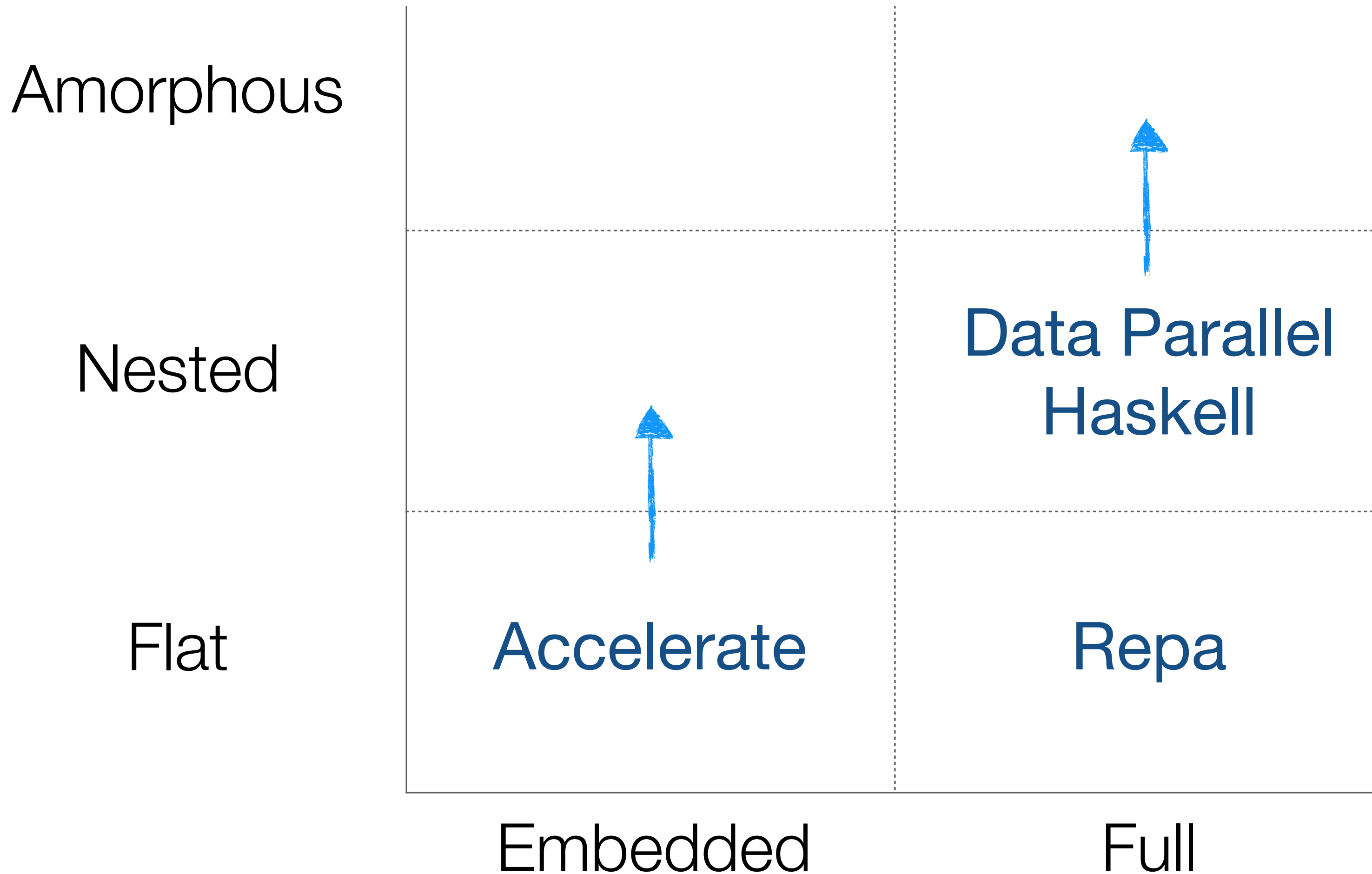
NESL also influenced

The Java 8 streams that you will see on Monday next week

Intel Array Building Blocks (ArBB)
That has been retired, but ideas are reappearing as C/C++ extensions

Futhark, which you will see on Thursday next week

Collections seem to encourage a functional style even in non functional languages
(remember Backus' paper from first lecture)



Slide borrowed from lecture by G. Keller

Data Parallel Haskell (DPH) intentions

NESL was a seminal breakthrough but, fifteen years later it remains largely un-exploited.Our goal is to adopt the key insights of NESL, embody them in a modern, widely-used functional programming language, namely Haskell, and implement them in a state-of-the-art Haskell compiler (GHC). The resulting system, Data Parallel Haskell, will make nested data parallelism available to real users.

Doing so is not straightforward. NESL a first-order language, has very few data types, was focused entirely on nested data parallelism, and its implementation is an interpreter. Haskell is a higher-order language with an extremely rich type system; it already includes several other sorts of parallel execution; and its implementation is a compiler.

http://www.cse.unsw.edu.au/~chak/papers/fsttcs2008.pdf

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:])
```

Examples

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

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

Nested data parallelism
Parallel op (svMul) on each row

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 (look for PRG-56)

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

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 wor
Meertens forma

Provides shape po
compiler like SAC, but using the Haskell type system

And you will have a lecture on Single Assignment C later in the course

terminology

Regular arrays

dense, rectangular, most elements non-zero

shape polymorphic

functions work over arrays of arbitrary dimension

terminology

Regular arrays

dense, recta

shape polyr

functions w   on

note: the arrays are purely
functional and immutable

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

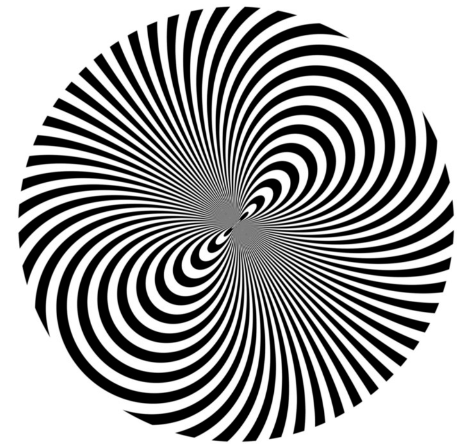
P processing elements, n array
elements => n/P consecutive
elements on each proc. element

Delayed (or pull) arrays great idea!

Represent array as function from index to value

Not a new idea
Originated in Pan in the functional world I think

See also
Compiling Embedded Langauges



But this is 100* slower than expected

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1 arr2
  = map (* 2) $ zipWith (+) arr1 arr2
```

Fast but cluttered

```
doubleZip arr1@(Manifest !_ !_) arr2@(Manifest !_ !_)
  = force $ map (* 2) $ zipWith (+) arr1 arr2
```

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

Repa's real strength

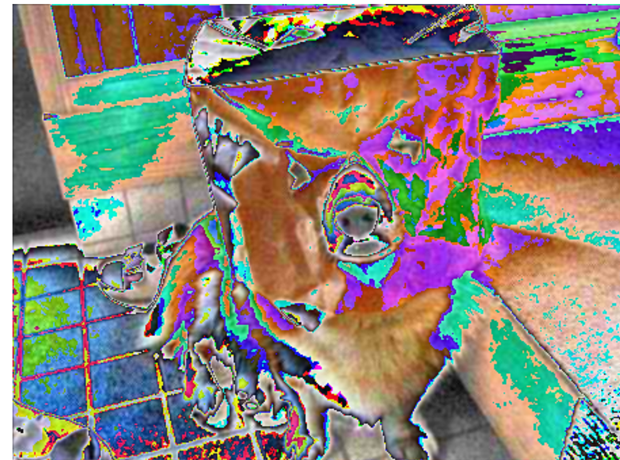
Stencil computations!

```
[stencil2| 0 1 0
           1 0 1
           0 1 0 |]
```

```
do
  (r, g, b) <- liftM (either (error . show) R.unzip3) readImageFromBMP "in.bmp"
  [r', g', b'] <- mapM (applyStencil simpleStencil) [r, g, b]
  writeImageToBMP "out.bmp" (U.zip3 r' g' b')
```

Repa's real strength

http://www.cse.chalmers.se/edu/year/2015/course/DAT280_Parallel_Functional_Programming/Papers/RepaTutorial13.pdf



Fancier array type (Repa 2)

```
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 -   ool
                 , rangeRects   :: [R     ] }
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

Repa info

http://repa.ouroborus.net/

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

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

Example: sorting

Batcher’s bitonic sort

(see lecture from last week)

“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

1 2 3 4 5 6 7 8 9 10 8 6 4 2 1 0

1 2 3 4 5 6 7 8 9 10 8 6 4 2 1 0

1 9

1 2 3 4 5 6 7 8 9 10 8 6 4 2 1 0

1 2 9 10

1 2 3 4 5 6 7 8 9 10 8 6 4 2 1 0

1 2 3 4 9 10 8 6

1 2 3 4 5 6 7 8 9 10 8 6 4 2 1 0

1 2 3 4 4 9 10 8 6 5

Swap!

1 2 3 4 5 6 7 8 9 10 8 6 4 2 1 0

1 2 3 4 4 2 9 10 8 6 5 6

1 2 3 4 5 6 7 8 9 10 8 6 4 2 1 0

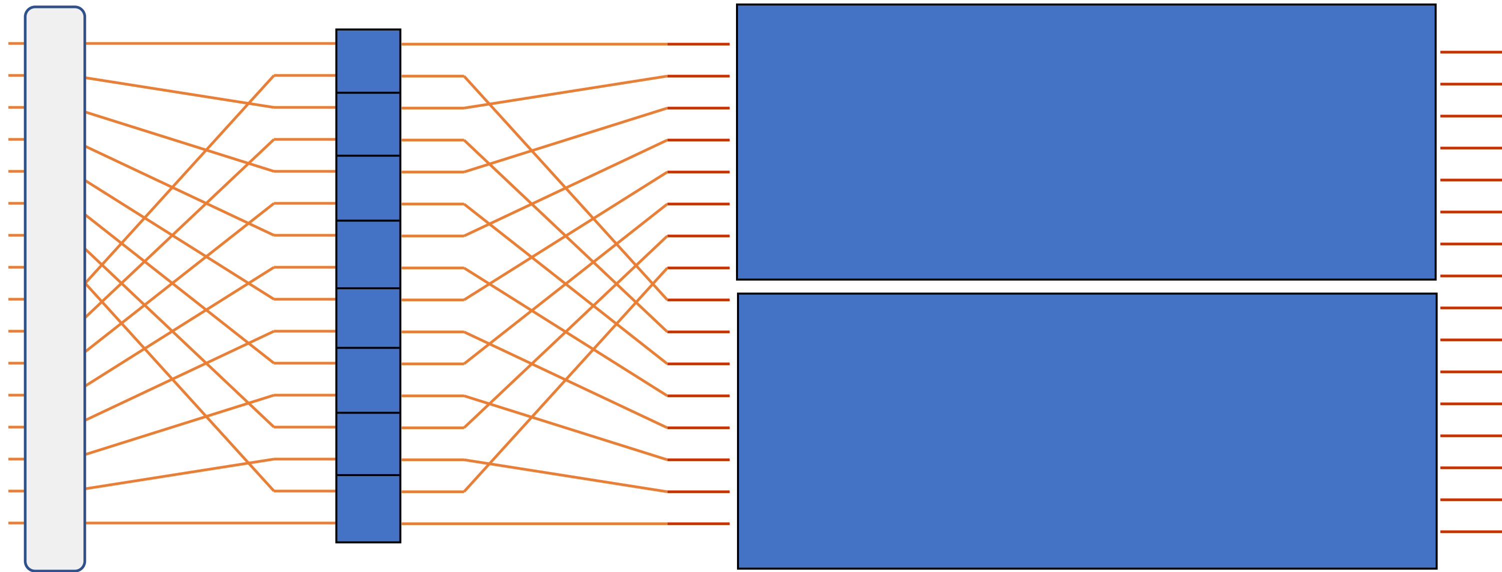
1 2 3 4 4 2 1 0 9 10 8 6 5 6 7 8

1 2 3 4 5 6 7 8 9 10 8 6 4 2 1 0

1 2 3 4 4 2 1 0 9 10 8 6 5 6 7 8

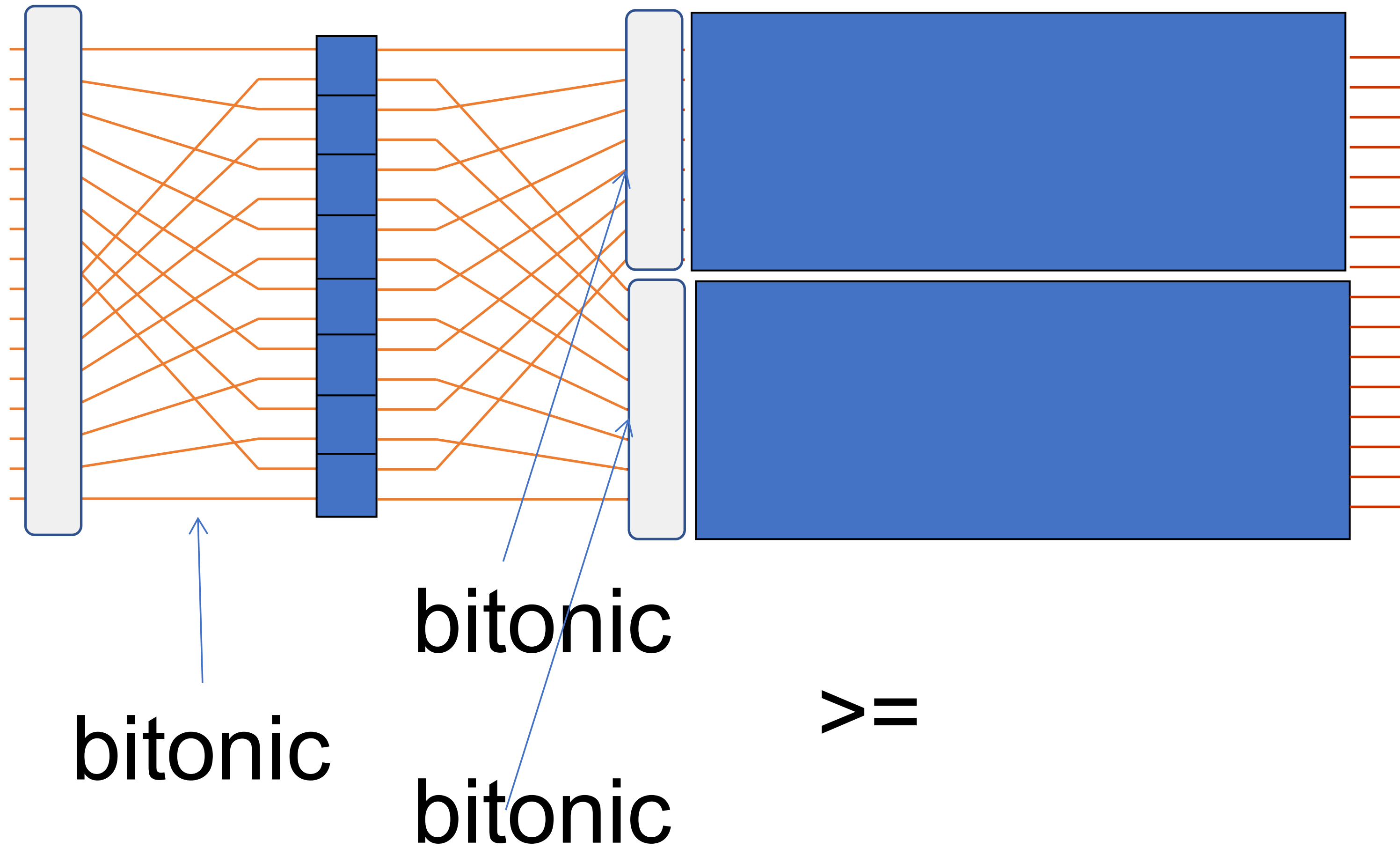
bitonic $\leq$ bitonic

Butterfly

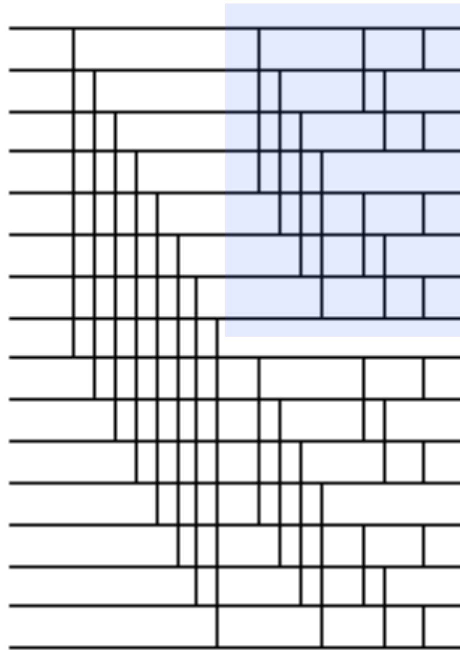


bitonic

Butterfly



bitonic merger



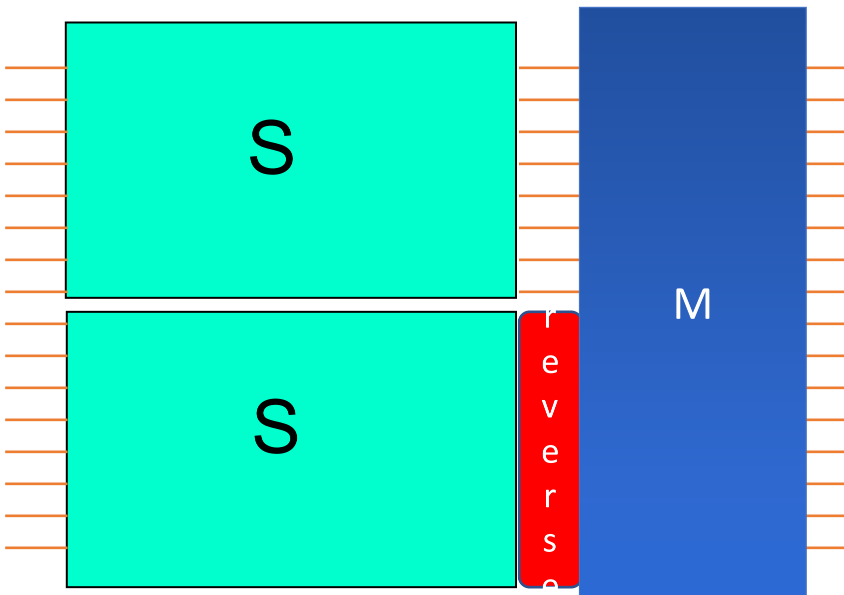
Question

What are the work and depth (or span) of bitonic merger?

Making a recursive sorter (D&C)

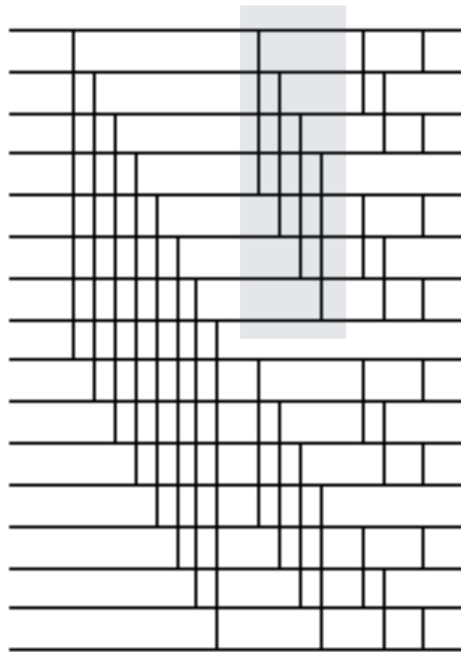
Make a bitonic sequence using two half-size sorters

Batcher's sorter (bitonic)

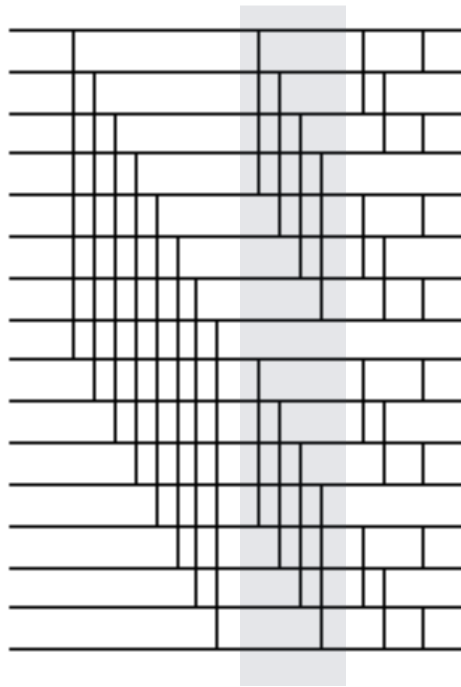


Let's try to write this sorter down in Repa

bitonic merger



bitonic merger



whole array operation

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

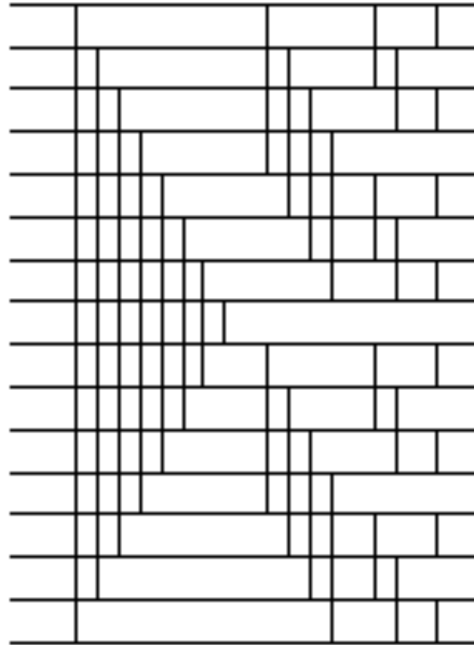
dee for diamond

```
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
```

dee f g 3 gives index i matched with index (i xor 8)

```
bitonicMerge n = compose [dee min max (n-i) | i <- [1..n]]
```

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 = extent arr in computeUnboxedP $ fromFunction sh 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
```

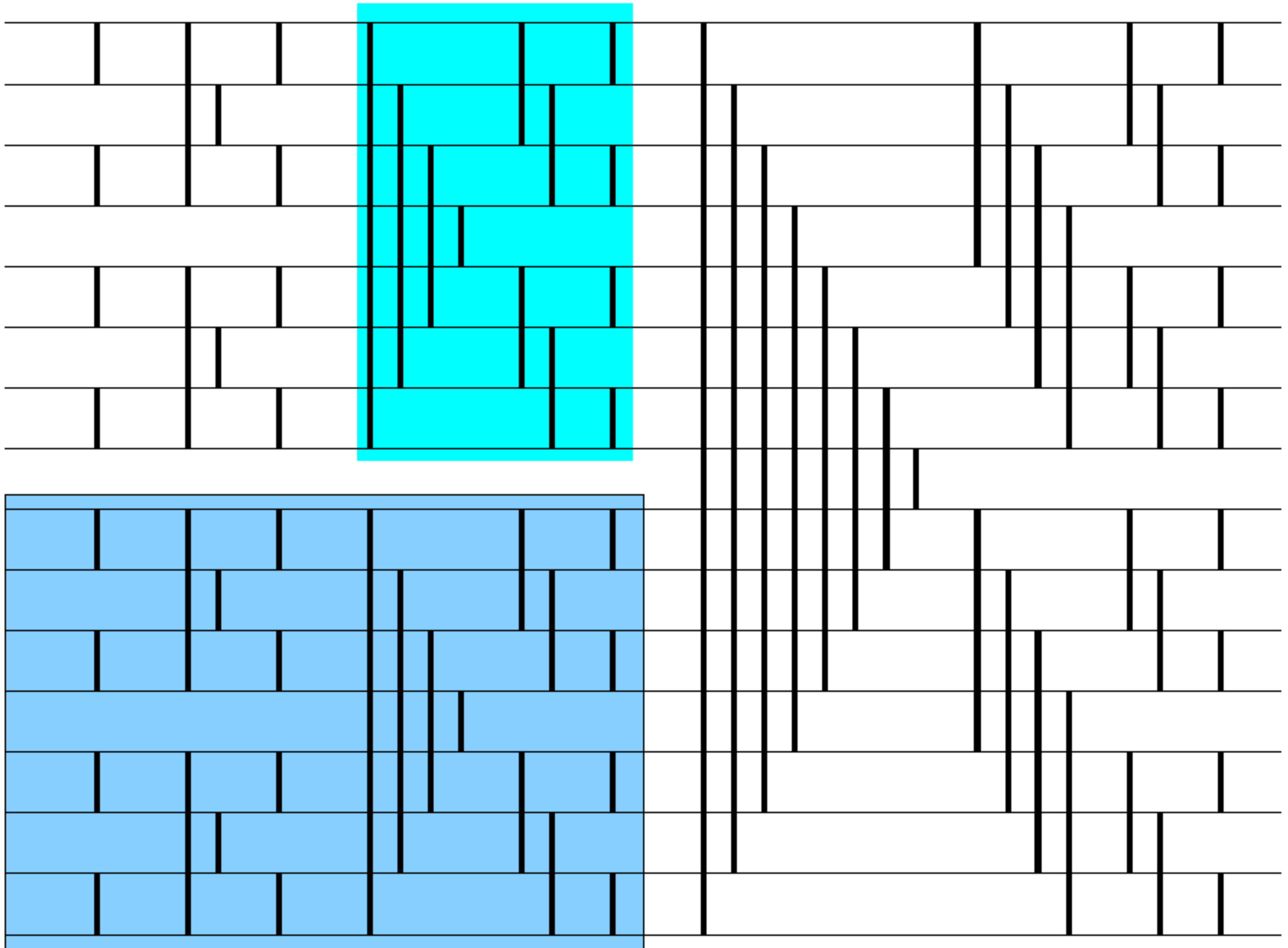
vee

```
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 = extent arr in computeUnboxedP $ fromFunction sh 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
```

vee f g 3 out(0) -> f a(0) a(7)
out(7) -> g a(7) a(0)
out(1) -> f a(1) a(6)
out(6) -> g a(6) a(1)

tmerge

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



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

Question

What are work and depth of this sorter??

Performance is decent!

Initial benchmarking for 2^20 Ints

Around 800ms on 4 cores on my previous laptop

Compares to around 1.6 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

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)

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

Idiomatic Repa (written by experts) is about 3 times slower.
Genericity costs here!

Message: map, fold and scan are not enough. We need to think more about higher order functions on arrays (e.g. with binary operators)

Nice success story at NYT

Haskell in the Newsroom

Haskell in Industry

stackoverflow

is your friend

See for example

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

Conclusions (Repa)

Based on DPH technology

Good speedups!

Neat programs

Good control of Parallelism

BUT CACHE AWARENESS needs to be tackled

Conclusions

Development seems to be happening in Accelerate, which now works for both

multicore and GPU (work ongoing)

Array representations for parallel functional programming is an important, fun and frustrating research topic ☺

par and pseq

Strategies

Par monad

Repa

(Accelerate)

(Obsidian)

NESL

Futhark

SAC

Haxl

Questions to think about

What is the right set of whole array operations?

(remember Backus from the first lecture)

A big question (at least for me)

How much should one put in the types?

More research needed

Combinators for parallel programming (influenced by Skeletons perhaps?)

Support for benchmarking, granularity control

Support for chunking (for example much needed in the Par monad)

Expressing locality Dealing with cache hierarchies

Need better ways to reinvent and assess parallel (functional) algorithms
(I find this paper about parallelising an important algorithm in visualisation very inspiring)

Oh and we are looking for doctoral students to work on secure programming of IoT!!

Octopi   Job ad