Nested data parallelism in Haskell

Simon Peyton Jones (Microsoft)
Manuel Chakravarty, Gabriele Keller, Roman Leshchinskiy
(University of New South Wales)

2009

Paper: "Harnessing the multicores"
At http:://research.microsoft.com/~simonpj

Road map

Multicore

Parallel programming essential

Task parallelism

- Explicit threads
- Synchronise via locks, messages, or STM

Modest parallelism
Hard to program

Data parallelism

Operate simultaneously on bulk data

Massive parallelism
Easy to program

- Single flow of control
- Implicit synchronisation

Haskell has **three** forms of concurrency

- Explicit threads
  - Non-deterministic by design
  - Monadic: **forkIO** and **STM**
- Semi-implicit
  - Deterministic
  - Pure: **par** and **seq**
- Data parallel
  - Deterministic
  - Pure: parallel arrays
  - Shared memory initially; distributed memory eventually; possibly even GPUs

```
main :: IO ()
  = do { ch <- newChan
    ; forkIO (ioManager ch)
    ; forkIO  (worker 1 ch)
    ... etc ... }
```

```
f :: Int -> Int
f x = a `par` b `seq` a + b
    where
       a = f (x-1)
       b = f (x-2)
```

- **General attitude**: using **some of** the parallel processors you already have, **relatively easily**

Data parallelism
The key to using multicores

Flat data parallel

Apply sequential operation to bulk data

- The brand leader
- Limited applicability (dense matrix, map/reduce)
- Well developed
- Limited new opportunities

Nested data parallel

Apply parallel operation to bulk data

- Developed in 90's
- Much wider applicability (sparse matrix, graph algorithms, games etc)
- Practically un-developed
- **Huge opportunity**

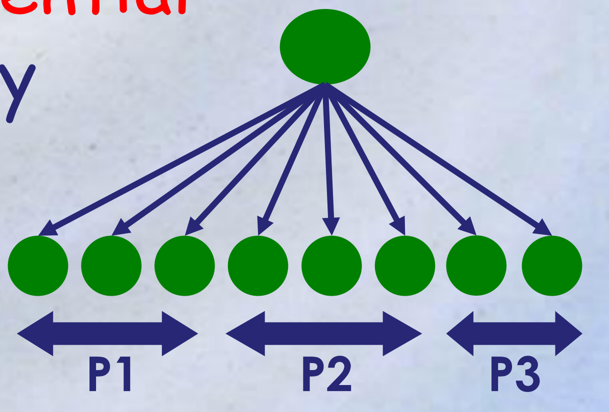
Flat data parallel

e.g. Fortran(s), *C
MPI, map/reduce

- The brand leader: widely used, well understood, well supported

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- BUT: "**something**" is sequential
- Single point of concurrency
- Easy to implement: use "chunking"
- Good cost model



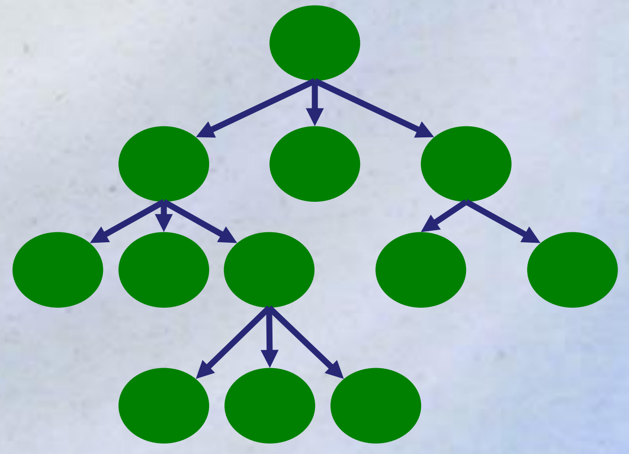
1,000,000's of (small) work items

Nested data parallel

- Main idea: **allow "something" to be parallel**

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- Now the parallelism structure is recursive, and un-balanced
- Still good cost model



Still 1,000,000's of (small) work items

Nested DP is great for **programmers**

- Fundamentally more modular
- Opens up a much wider range of applications:
  - Sparse arrays, variable grid adaptive methods (e.g. Barnes-Hut)
  - Divide and conquer algorithms (e.g. sort)
  - Graph algorithms (e.g. shortest path, spanning trees)
  - Physics engines for games, computational graphics (e.g. Delauny triangulation)
  - Machine learning, optimisation, constraint solving

Nested DP is tough for **compilers**

- ...because the concurrency tree is both irregular and fine-grained
- But it can be done! NESL (Blelloch 1995) is an existence proof
- Key idea: “flattening” transformation:

Array comprehensions

[:Float:] is the type of parallel arrays of Float

```
vecMul :: [:Float:] -> [:Float:] -> Float
vecMul v1 v2 = sumP [: f1*f2 | f1 <- v1 | f2 <- v2 :]
```

sumP :: [:Float:] -> Float

An array comprehension: "the array of all f1*f2 where f1 is drawn from v1 and f2 from v2"

Operations over parallel array are computed in parallel; that is the only way the programmer says "do parallel stuff"

NB: no locks!

Sparse vector multiplication

A sparse vector is represented as a vector of (index,value) pairs

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

v!i gets the i'th element of v

Parallelism is proportional to length of sparse vector

Sparse matrix multiplication

A sparse matrix is a vector of sparse vectors

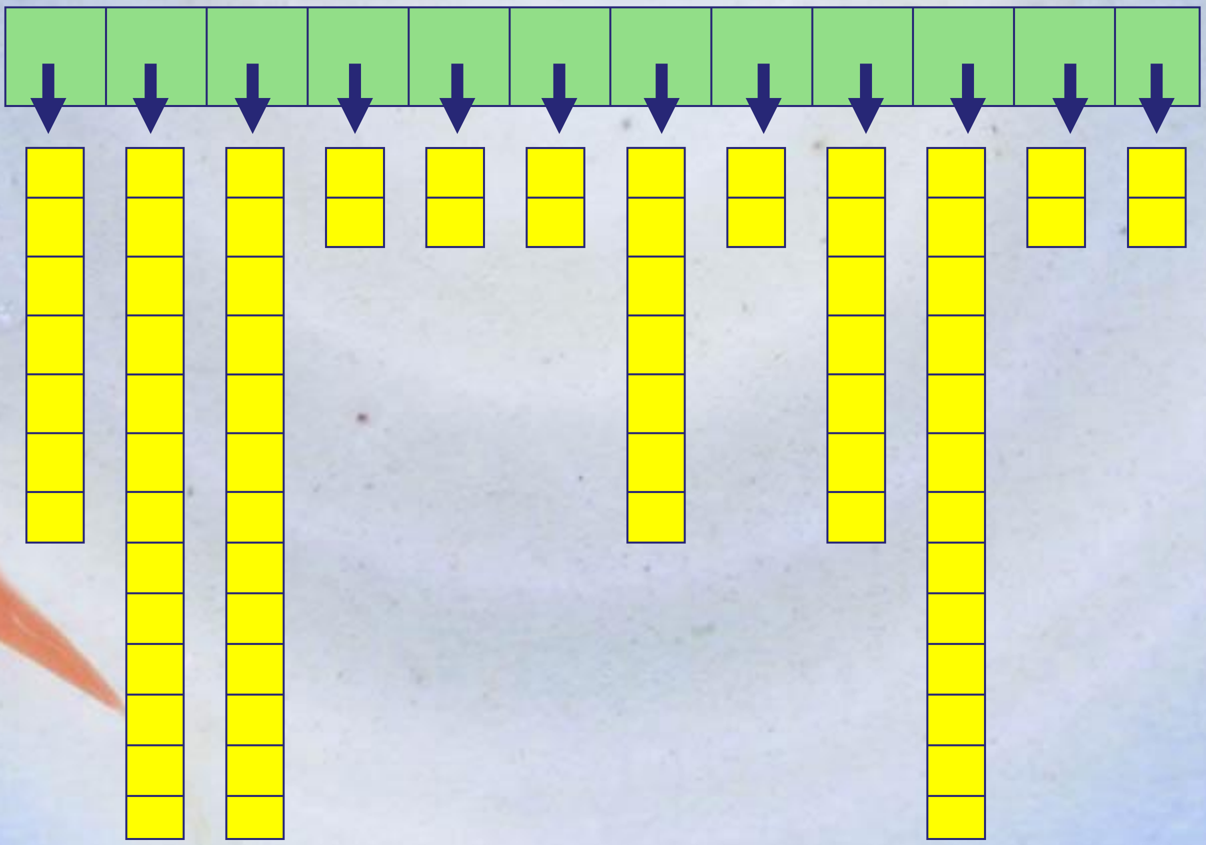
```
smMul :: [:[:(Int,Float):]:] -> [:Float:] -> Float
smMul sm v = sumP [: svMul sv v | sv <- sm :]
```

Nested data parallelism here!

We are calling a parallel operation, svMul, on every element of a parallel array, sm

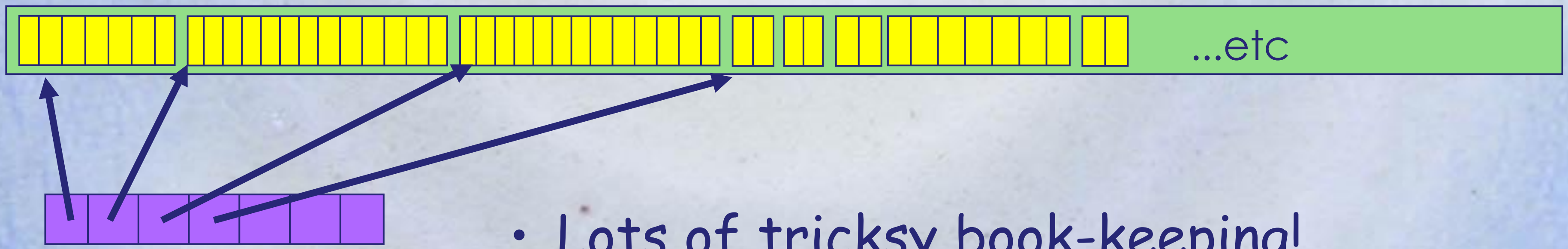
Hard to implement well

- Evenly chunking at top level might be **ill-balanced**
- Top level along might **not be very parallel**



The flattening transformation

- Concatenate sub-arrays into one big, flat array
- Operate in parallel on the big array
- Segment vector keeps track of where the sub-arrays are



- Lots of tricksy book-keeping!
- Possible to do by hand (and done in practice), but very hard to get right
- Blelloch showed it could be done systematically

Parallel search

```
type Doc = [: String :] -- Sequence of words
type DocBase  = [: Document :]

search :: DocBase -> String -> [: (Doc,[:Int:]):]
```

Find all Docs that mention the string, along with the places where it is mentioned (e.g. word 45 and 99)

Parallel search

```
type Doc = [: String :]
type DocBase  = [: Document :]

search :: DocBase -> String -> [: (Doc,[:Int:]):]

wordOccs :: Doc -> String -> [: Int :]
```

Find all the places where a string is mentioned in a document
(e.g. word 45 and 99)

Parallel search

```
type Doc = [: String :]
type DocBase  = [: Document :]

search :: DocBase -> String -> [: (Doc,[:Int:]):]
search ds s = [: (d,is) | d <- ds
                        , let is = wordOccs d s
                        , not (nullP is) :]

wordOccs :: Doc -> String -> [: Int :]
```

```
nullP :: [:a:] -> Bool
```

Parallel search

```
type Doc = [: String :]
type DocBase  = [: Document :]

search :: DocBase -> String -> [: (Doc,[:Int:]):]

wordOccs :: Doc -> String -> [: Int :]
wordOccs d s = [: i | (i,s2) <- zipP positions d
                         , s == s2 :]
    where
      positions :: [: Int :]
      positions = [: 1..lengthP d :]
```

```
zipP    :: [:a:] -> [:b:] -> [:(a,b):]
lengthP :: [:a:] -> Int
```

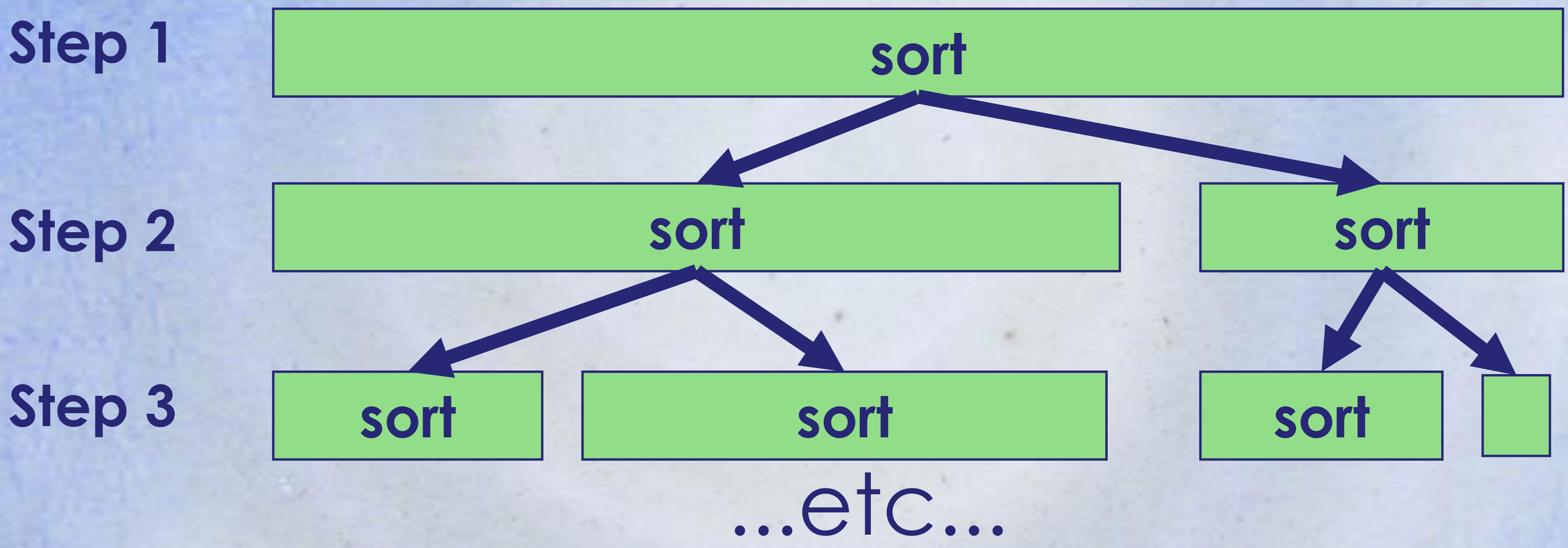
Data-parallel quicksort

```
sort :: [:Float:] -> [:Float:]
sort a = if (lengthP a <= 1) then a
         else sa!0 +++ eq +++ sa!1
   where
      m = a!0
      lt = [: f | f<-a, f<m :]
      eq = [: f | f<-a, f==m :]
      gr = [: f | f<-a, f>m :]
      sa = [: sort a | a <- [:lt,gr:] :]
```

Parallel filters

2-way nested data parallelism here!

How it works



- All sub-sorts at the same level are done in parallel
- Segment vectors track which chunk belongs to which sub problem
- Instant insanity when done by hand

In the paper…

- All the examples so far have been small
- In the paper you’ll find a much more substantial example: the Barnes-Hut N-body simulation algorithm
- Very hard to fully parallelise by hand

Fusion

- Flattening is not enough

```
vecMul :: [:Float:] -> [:Float:] -> Float
vecMul v1 v2 = sumP [: f1*f2 | f1 <- v1 | f2 <- v2 :]
```

- Do not
  1. Generate [: f1*f2 | f1 <- v1 | f2 <- v2 :] (big intermediate vector)
  2. Add up the elements of this vector
- Instead: multiply and add in the same loop
- That is, **fuse** the multiply loop with the add loop
- Very general, aggressive fusion is required

What we are doing about it

NESL

a mega-breakthrough but:

- specialised, prototype
- first order
- few data types
- no fusion
- interpreted

Substantial improvement in

- **Expressiveness**
- **Performance**

Haskell

- broad-spectrum, widely used
- higher order
- very rich data types
- aggressive fusion
- compiled

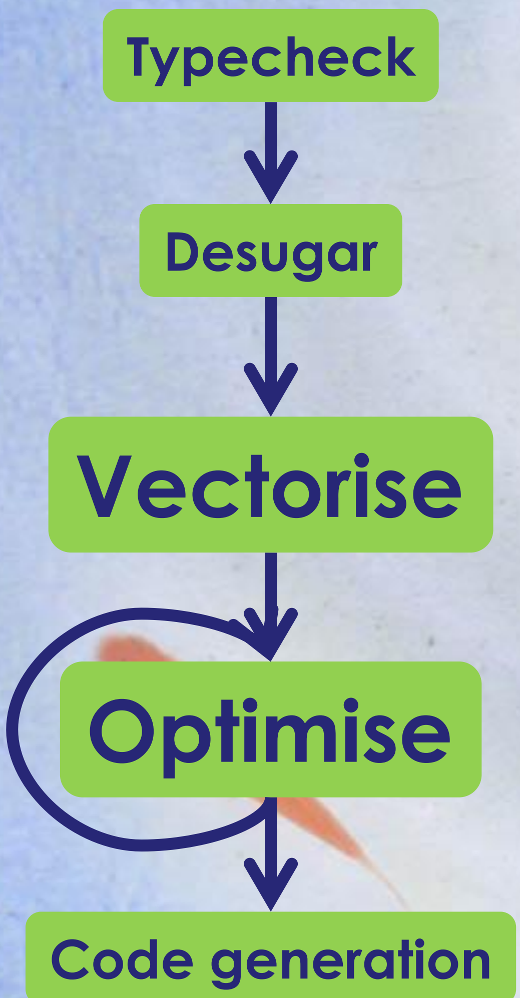
- **Shared memory initially**
- **Distributed memory eventually**
- **GPUs anyone?**

Main contribution: an optimising data-parallel compiler implemented by modest enhancements to a full-scale functional language implementation

Four key pieces of technology

1. Flattening
   - specific to parallel arrays
2. Non-parametric data representations
   - A generically useful new feature in GHC
3. Chunking
   - Divide up the work evenly between processors
4. Aggressive fusion
   - Uses "rewrite rules", an old feature of GHC

Overview of compilation



Not a special purpose data-parallel compiler!
Most support is either useful for other things, or is in the form of library code.

The flattening transformation
(new for NDP)
Main focus of the paper

Chunking and fusion
("just" library code)

Step 0: desugaring

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

```
sumP :: Num a => [:a:] -> a
mapP :: (a -> b) -> [:a:] -> [:b:]
```

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

Step 1: Vectorisation

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

```
sumP      :: Num a => [:a:] -> a
*^        :: Num a => [:a:] -> [:a:] -> [:a:]
fst^      :: [:(a,b):] -> [:a:]
bpermuteP :: [:a:] -> [:Int:] -> [:a:]
```

```
svMul :: [:(Int,Float):] -> [:Float:] -> Float
svMul sv v = sumP (snd^ sv  *^  bpermuteP v (fst^ sv))
```

Scalar operation * replaced by vector operation *^

Vectorisation: the basic idea

```
mapP f v
```

➡

```
f^ v
```

```
f  :: T1 -> T2
f^ :: [:T1:] -> [:T2:]  -- f^ = mapP f
```

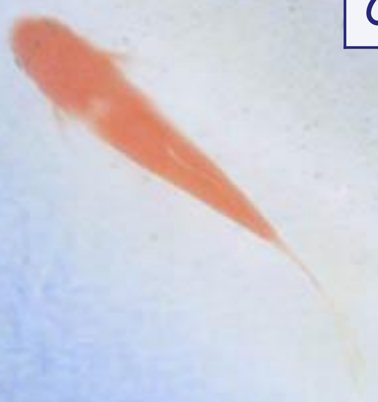
- For every function f, generate its **lifted version**, namely f^
- Result: a functional program, operating over flat arrays, with a fixed set of primitive operations *^, sumP, fst^, etc.
- Lots of intermediate arrays!

Vectorisation: the basic idea

```
f  :: Int -> Int
f x = x+1

f^ :: [:Int:] -> [:Int:]
f^ x = x +^ (replicateP (lengthP x) 1)
```

This	Transforms to this
Locals, x	x
Globals, g	g^
Constants, k	replicateP (lengthP x) k



```
replicateP :: Int -> a -> [:a:]
lengthP :: [:a:] -> Int
```

Vectorisation: the key insight

```
f  :: [:Int:] -> [:Int:]
f a = mapP g a = g^ a

f^ :: [:[:Int:]:] -> [:[:Int:]:]
f^ a = g^^ a      --???
```

Yet another version of g???

Vectorisation: the key insight

```
f  :: [:Int:] -> [:Int:]
f a = mapP g a = g^ a

f^ :: [:[:Int:]:] -> [:[:Int:]:]
f^ a = segmentP a (g^ (concatP a))
```

First concatenate, then map, then re-split

```
concatP  :: [:[:a:]:] -> [:a:]
segmentP :: [:[:a:]:] -> [:b:] -> [:[:b:]:]
```

Shape

Flat data

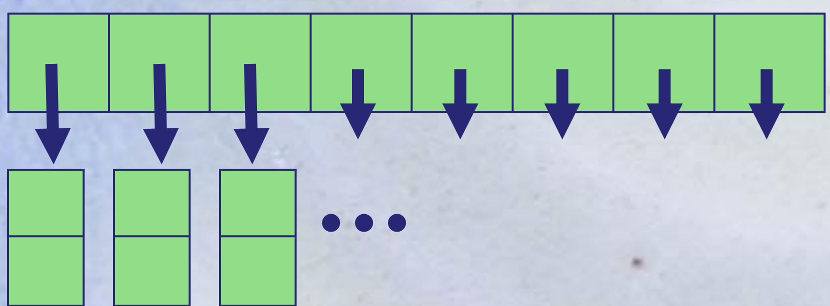
Nested data

Payoff: f and f^ are enough. No f^^

Step 2: Representing arrays

`[:Double:]` Arrays of pointers to boxed numbers are Much Too Slow

`[:(a,b):]` Arrays of pointers to pairs are Much Too Slow



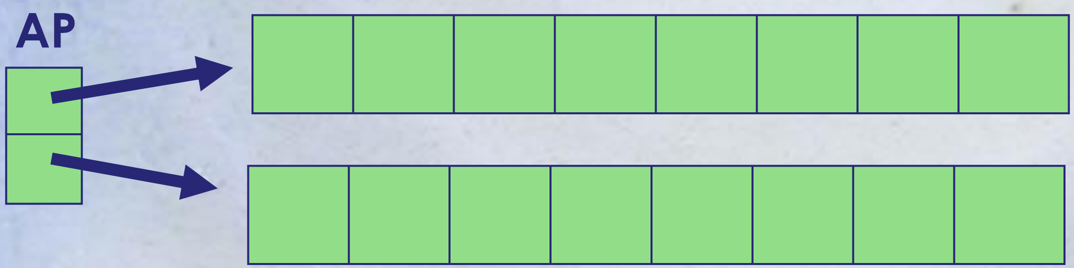
Idea!
Representation of an array depends on the element type

Step 2: Representing arrays

[POPL05], [ICFP05], [TLDI07]

```
data family [:a:]

data instance [:Double:] = AD ByteArray
data instance [:(a,b):]  = AP [:a:] [:b:]
```



- Now *^ is a fast loop
- And fst^ is constant time!

```
fst^ :: [:(a,b):] -> [:a:]
fst^ (AP as bs) = as
```

Step 2: Nested arrays

Shape

Flat data

```
data instance [:[:a:]:] = AN [:Int:] [:a:]

concatP  :: [:[:a:]:] -> [:a:]
concatP (AN shape data) = data

segmentP :: [:[:a:]:] -> [:b:] -> [:[:b:]:]
segmentP (AN shape _) data = AN shape data
```

Surprise: concatP, segmentP are constant time!

Higher order complications

```
f   :: T1 -> T2 -> T3

f1^ :: [:T1:] -> [:T2:] -> [:T3:] -- f1^ = zipWithP f
f2^ :: [:T1:] -> [:(T2 -> T3):]   -- f2^ = mapP f
```

- f1^ is good for [: f a b | a <- as | b <- bs :]
- But the type transformation is not uniform
- And sooner or later we want higher-order functions anyway
- f2^ forces us to find a representation for [:(T2->T3):]. Closure conversion [PAPP06]

Step 3: chunking

```
svMul :: [:(Int,Float):] -> [:Float:] -> Float
svMul (AP is fs) v = sumP (fs  *^  bpermuteP v is)
```

- Program consists of
  - Flat arrays
  - Primitive operations over them (*^, sumP etc)
- Can directly execute this (NESL).
  - Hand-code assembler for primitive ops
  - All the time is spent here anyway
- But:
  - intermediate arrays, and hence memory traffic
  - each intermediate array is a synchronisation point
- Idea: chunking and fusion

Step 3: Chunking

```
svMul :: [:(Int,Float):] -> [:Float:] -> Float
svMul (AP is fs) v = sumP (fs  *^  bpermuteP v is)
```

1. **Chunking**: Divide is,fs into chunks, one chunk per processor
2. **Fusion**: Execute sumP (fs *^ bpermute v is) in a tight, sequential loop on each processor
3. **Combining**: Add up the results of each chunk

Step 2 alone is not good for a parallel machine!

Expressing chunking

```
sumP :: [:Float:] -> Float
sumP xs = sumD (mapD sumS (splitD xs))
```

```
splitD    :: [:a:] -> Dist [:a:]
mapD      :: (a->b) -> Dist a -> Dist b
sumD      :: Dist Float -> Float

sumS      :: [:Float:] -> Float      -- Sequential!
```

- **sumS** is a tight sequential loop
- **mapD** is the true source of parallelism:
  - it starts a "gang",
  - runs it,
  - waits for all gang members to finish

Expressing chunking

```
*^ :: [:Float:] -> [:Float:] -> [:Float:]
*^ xs ys = joinD (mapD mulS
              (zipD (splitD xs) (splitD ys))
```

```
splitD    :: [:a:] -> Dist [:a:]
joinD     :: Dist [:a:] -> [:a:]
mapD      :: (a->b) -> Dist a -> Dist b
zipD      :: Dist a -> Dist b -> Dist (a,b)

mulS :: ([:Float:],[: Float :]) -> [:Float:]
```

- Again, mulS is a tight, sequential loop

Step 4: Fusion

```
svMul :: [:(Int,Float):] -> [:Float:] -> Float
svMul (AP is fs) v = sumP (fs  *^  bpermuteP v is)
 = sumD . mapD sumS . splitD . joinD . mapD mulS $
      zipD (splitD fs) (splitD (bpermuteP v is))
```

- Aha! Now use rewrite rules:

```
{-# RULE
    splitD (joinD x)   = x
    mapD f (mapD g x)  = mapD (f.g) x #-}
```

```
svMul :: [:(Int,Float):] -> [:Float:] -> Float
svMul (AP is fs) v = sumP (fs  *^  bpermuteP v is)
 = sumD . mapD (sumS . mulS) $
      zipD (splitD fs) (splitD (bpermuteP v is))
```

Step 4: Sequential fusion

```
svMul :: [:(Int,Float):] -> [:Float:] -> Float
svMul (AP is fs) v = sumP (fs  *^  bpermuteP v is)
 = sumD . mapD (sumS . mulS) $
      zipD (splitD fs) (splitD (bpermuteP v is))
```

- Now we have a sequential fusion problem.
- Problem:
  - lots and lots of functions over arrays
  - we can't have fusion rules for every pair
- New idea: stream fusion

In the paper

- The paper gives a much more detailed description of
  - The vectorisation transformation
  - The non-parametric representation of arrays

  This stuff isn't new, but the paper gathers several papers into a single coherent presentation
- (There's a sketch of chunking and fusion too, but the main focus is on vectorisation.)

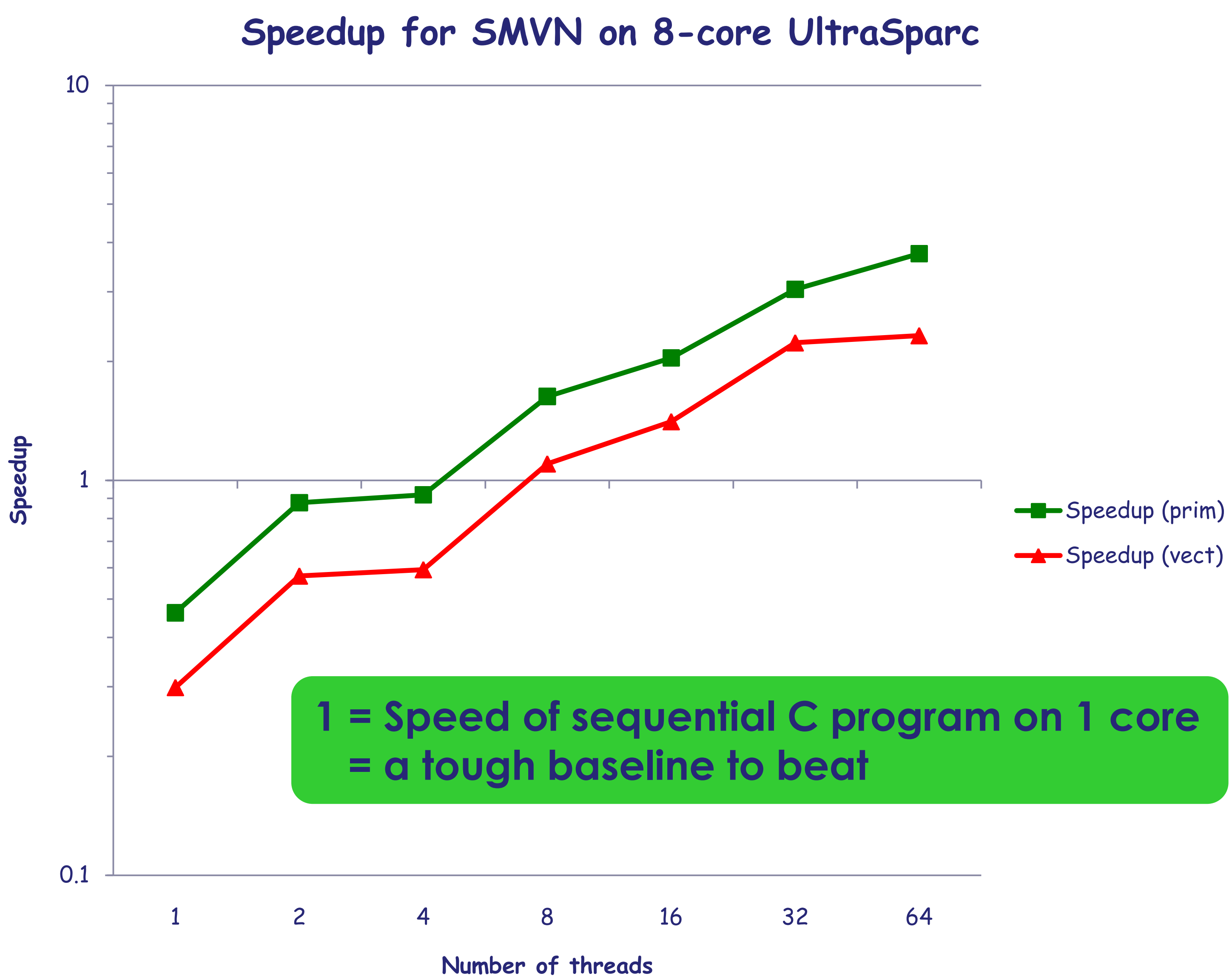
So how far have we got?

Four key pieces of technology

1. Flattening
2. Non-parametric data representations
3. Chunking
4. Aggressive fusion

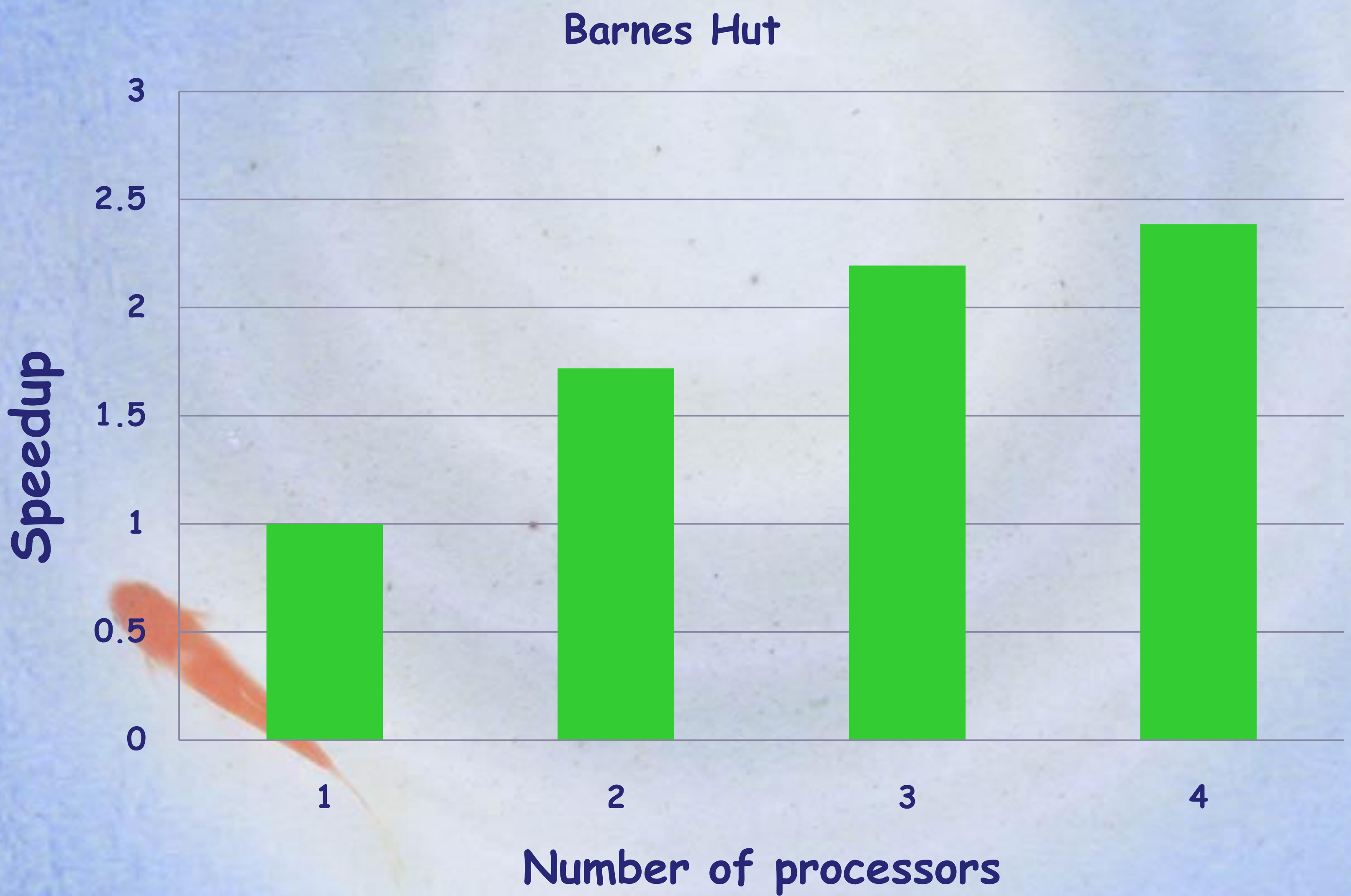
An ambitious enterprise; but version 1 now implemented and released in GHC 6.10

Does it work?



1 = Speed of sequential C program on 1 core
= a tough baseline to beat

Less good for Barnes-Hut



Summary

- **Data parallelism** is the only way to harness 100's of cores
- **Nested DP** is great for programmers: far, far more flexible than flat DP
- Nested DP is tough to implement. We are optimistic, but have some way to go.
- **Huge opportunity**: almost no one else is dong this stuff!
- Functional programming is a massive win in this space: Haskell prototype in 2008
- **WANTED**: friendly guinea pigs

http://haskell.org/haskellwiki/GHC/Data_Parallel_Haskell

Paper: "Harnessing the multicores" on my home page

Purity pays off

- Two key transformations:
  - Flattening
  - Fusion
- Both depend utterly on purely-functional semantics:
  - no assignments
  - every operation is a pure function

The data-parallel languages of the future will be functional languages

Extra slides

Stream fusion for lists

```
map f (filter p (map g xs))
```

- Problem:
  - lots and lots of functions over lists
  - and they are **recursive** functions
- New idea: make map, filter etc non-recursive, by defining them to work over **streams**

Stream fusion for lists

```
data Stream a where
    S :: (s -> Step s a) -> s -> Stream a

data Step s a = Done | Yield a (Stream s a)

toStream :: [a] -> Stream a
toStream as = S step as
  where
    step [] = Done
    step (a:as) = Yield a as

fromStream :: Stream a -> [a]
fromStream (S step s) = loop s
  where
    loop s = case step s of
               Yield a s'  -> a : loop s'
               Done        -> []
```

Non-recursive!

Recursive

Stream fusion for lists

```
mapStream :: (a->b) -> Stream a -> Stream b
mapStream f (S step s) = S step' s
  where
    step' s = case step s of
                Done        -> Done
                Yield a s' -> Yield (f a) s'

map :: (a->b) -> [a] -> [b]
map f xs = fromStream (mapStream f (toStream xs))
```

Non-recursive!

Stream fusion for lists

```
map f (map g xs)

= fromStream (mapStream f (toStream
    (fromStream (mapStream g (toStream xs))))

=                 -- Apply (toStream (fromStream xs) = xs)
  fromStream (mapStream f (mapStream g (toStream xs)))

=                 -- Inline mapStream, toStream
  fromStream (Stream step xs)
   where
     step [] = Done
     step (x:xs) = Yield (f (g x)) xs
```

Stream fusion for lists

```
fromStream (Stream step xs)
     where
       step [] = Done
       step (x:xs) = Yield (f (g x)) xs

=                       -- Inline fromStream
  loop xs
    where
        loop [] = []
        loop (x:xs) = f (g x) : loop xs
```

- Key idea: mapStream, filterStream etc are all non-recursive, and can be inlined
- Works for arrays; change only fromStream, toStream