cs242

PARALLELISM IN HASKELL

Kathleen Fisher

Reading: A Tutorial on Parallel and Concurrent Programming in Haskell
Skip Section 5 on STM

Thanks to Simon Peyton Jones, Satnam Singh, and Don Stewart for these slides.

Announcements

- Submit course evaluations in Axess
  - Open: Nov. 30 to Dec. 14 at 8am.
  - Registrar: Students who submit evaluations will see grades when submitted by faculty; others will see grades on Jan. 4.
  - Your feedback is crucial to improving the course!
  - Please participate.
- Final exam:
  - Monday, December 7, 12:15-3:15pm in Gates B01.
  - Local SCPD students should come to campus for exam.

The Grand Challenge

- Making effective use of multi-core hardware is the challenge for programming languages now.
- Hardware is getting increasingly complicated:
  - Nested memory hierarchies
  - Hybrid processors: GPU + CPU, Cell, FPGA...
  - Massive compute power sitting mostly idle.
- We need new programming models to program new commodity machines effectively.

Candidate models in Haskell

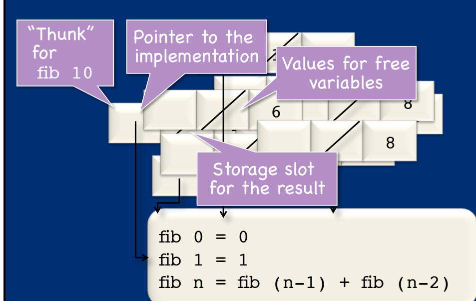
- Explicit threads
  - Non-deterministic by design
  - Monadic: **forkIO** and **STM**
- Semi-implicit parallelism
  - Deterministic
  - Pure: **par** and **pseq**
- Data parallelism
  - Deterministic
  - Pure: parallel arrays
  - Shared memory initially; distributed memory eventually; possibly even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

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

Parallelism vs Concurrency

- A parallel program exploits real parallel computing resources to *run faster* while computing the *same answer.*
  - Expectation of genuinely simultaneous execution
  - Deterministic
- A concurrent program models independent agents that can communicate and synchronize.
  - Meaningful on a machine with one processor
  - Non-deterministic

Haskell Execution Model



Functional Programming to the Rescue?

- No side effects makes parallelism easy, right?
  - It is always safe to speculate on pure code.
  - Execute each sub-expression in its own thread?
- Alas, the 80s dream does not work.
  - Far too many parallel tasks, many of which are too small to be worth the overhead of forking them.
  - Difficult/impossible for compiler to guess which are worth forking.

Idea: Give the user control over which expressions might run in parallel.

The `par` combinator

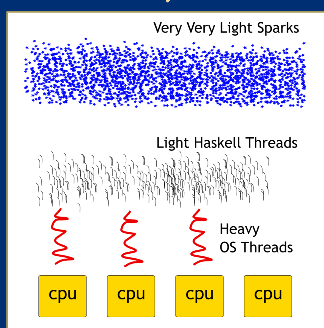
```
par :: a -> b -> b
x `par` y
```

- Value (ie, thunk) bound to `x` is **sparked** for speculative evaluation.
- Runtime may instantiate a spark on a thread running in parallel with the parent thread.
- Operationally, ``x `par` y = y``
- Typically, `x` is used inside `y`:

```
blurRows `par` (mix blurCols blurRows)
```

- All parallelism built up from the `par` combinator.

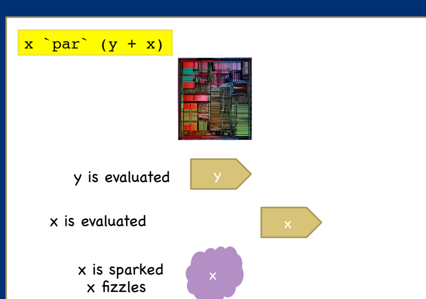
Concurrency Hierarchy



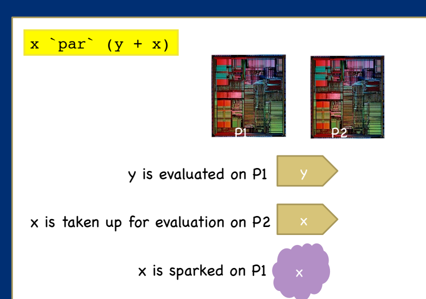
The meaning of `par`

- `par` does not guarantee a new Haskell thread.
- It hints that it would be good to evaluate the first argument in parallel.
- The runtime decides whether to convert spark
  - Depending on current workload.
- This allows `par` to be very cheap.
  - Programmers can use it almost anywhere.
  - Safely over-approximate program parallelism.

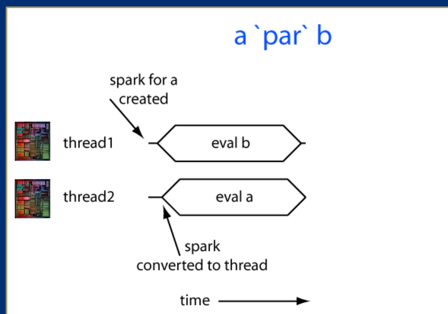
Example: One processor



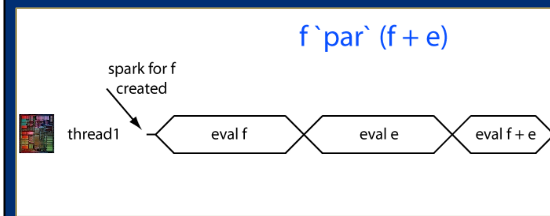
Example: Two Processors



Model: Two Processors

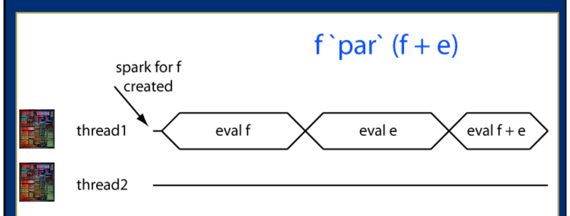


Model: One Processor



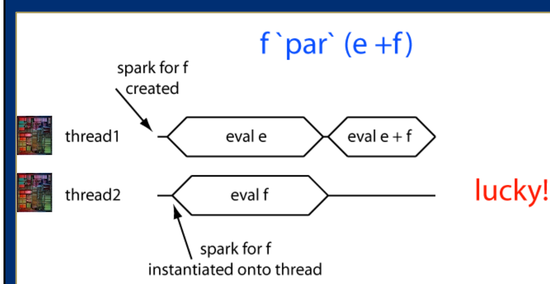
- No extra resources, so spark for `f` fizzles.

No parallelism?



- Main thread demands `f`, so spark fizzles.

Lucky parallelism



A second combinator: pseq

```
pseq :: a -> b -> b
x `pseq` y
```

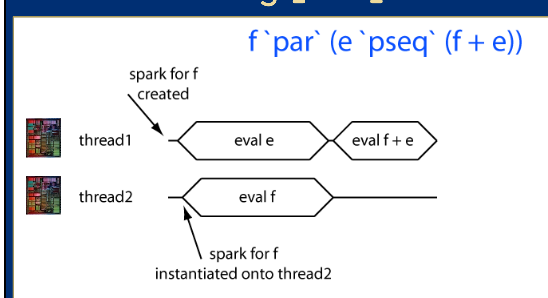
- pseq: Evaluate `x` in the current thread, then return `y`.
- Operationally,

```
x `pseq` y = bottom if x -> bottom
           = y      otherwise.
```

- With pseq, we can control evaluation order.

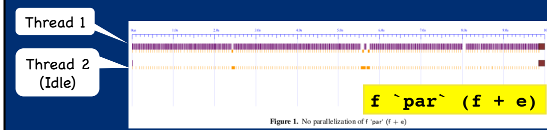
```
e `par` f `pseq` (f + e)
```

Using pseq



ThreadScope

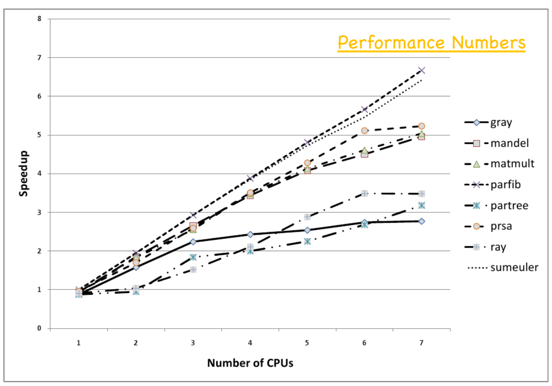
- ThreadScope (in Beta) displays event logs generated by GHC to track spark behavior:



Sample Program

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib(n-2)

sumEuler :: Int -> Int
sumEuler n = … in ConcTutorial.hs …

parSumFibEulerGood :: Int -> Int -> Int
parSumFibEulerGood a b = f `par` (e `pseq` (f + e))
  where
    f = fib a
    e = sumEuler b
```

- The `fib` and `sumEuler` functions are unchanged.



Summary: Semi-implicit parallelism

- Deterministic:
  - Same results with parallel and sequential programs.
  - No races, no errors.
  - Good for reasoning: Erase the `par` combinator and get the original program.
- Relies on purity.
- Cheap: Sprinkle `par` as you like, then measure with ThreadScope and refine.
- Takes practice to learn where to put `par` and `pseq`.
- Often good speed-ups with little effort.

Candidate Models in Haskell

- Explicit threads
  - Non-deterministic by design
  - Monadic: **forkIO** and **STM**
- Semi-implicit
  - Deterministic
  - Pure: **par** and **pseq**
- Data parallelism
  - 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 `pseq` a + b
  where
    a = f1 (x-1)
    b = f2 (x-2)
```

Road map

Multicore → Parallel programming essential

Task parallelism
Each thread does something different.
- Explicit: threads, MVars, STM
- Implicit: par & pseq

Modest parallelism
Hard to program

Data parallelism
Operate simultaneously on bulk data

Massive parallelism
Easy to program
- Single flow of control
- Implicit synchronisation

Data parallelism

Flat data parallel
Apply *sequential* operation to bulk data

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

Nested data parallel
Apply *parallel* operation to bulk data

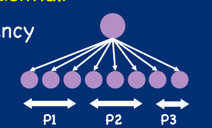
- Developed in 90's
- Much wider applicability (sparse matrix, graph algorithms, games etc)
- Practically un-developed
- Huge opportunity

Flat data parallel

- 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



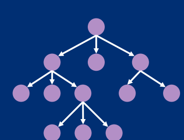
1,000,000's of (small) work items

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.
- Hard to implement!


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:
  - Divide and conquer algorithms (e.g. sort)
  - Graph algorithms (e.g. shortest path, spanning trees)
  - Sparse arrays, variable grid adaptive methods (e.g. Barnes-Hut)
  - Physics engines for games, computational graphics (e.g. Delauny triangulation)
  - Machine learning, optimization, 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:

Nested data parallel program (the one we want to write) → Compiler → Flat data parallel program (the one we want to run)

Data Parallel Haskell

NESL (Blelloch)
A mega-breakthrough but:
- specialized, prototype
- first order
- few data types
- no fusion
- interpreted

Substantial improvement in
- Expressiveness
- Performance

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

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

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

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`

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

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

NB: no locks!

Sparse vector multiplication

A sparse vector is represented as a vector of (index, value) pairs: `[:(0,3),(2,10):]` instead of `[:3,0,10,0:]`.

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

`v!i` gets the ith element of `v`

```
sDotP [:(0,3),(2,10):][:2,1,1,4:]
 = sumP [: 3 * 2, 10 * 1 :]
 = 16
```

Parallelism is proportional to length of sparse vector.

Sparse matrix multiplication

A sparse matrix is a vector of sparse vectors: `[:[:(1,3),(4,10):], [:(0,2),(1,12),(4,6):]:]`

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

Nested data parallelism here!
We are calling a parallel operation, **`sDotP`**, on every element of a parallel array, `sm`.

Example: Data-parallel Quicksort

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

Parallel filters

2-way nested data parallelism here.

Example: Parallel Search

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

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

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

Example: Parallel Search

```
type Doc    = [: String :]
type Corpus = [: Doc :]

search :: Corpus -> 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) .

Example: Parallel Search

```
type Doc    = [: String :]
type Corpus = [: Doc :]

search :: Corpus -> 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
```

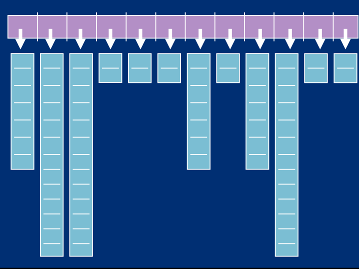
Example: Parallel Search

```
type Doc    = [: String :]
type Corpus = [: Doc :]

search :: Corpus -> 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
```

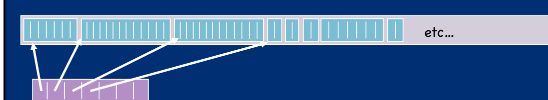
Hard to implement well!

- Evenly chunking at top level might be **ill-balanced**.
- Top level alone 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** tracks extent of sub-arrays.



- 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.

Fusion

Flattening enables load balancing, but it is not enough to ensure good performance. Consider:

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

- **Bad idea:**
  1. Generate `[: f1*f2 | f1 <- v1 | f2 <-v2 :]`
  2. Add the elements of this big intermediate vector.
- **Good idea:** Multiply and add in the same loop.
  - That is, **fuse** the multiply loop with the add loop.
  - Very general, aggressive fusion is required.

Implementation Techniques

Four key pieces of technology:

1. **Vectorization**
   - Specific to parallel arrays
2. **Non-parametric data representations**
   - A generically useful new feature in GHC

} Flattening

3. **Distribution**
   - Divide up the work evenly between processors
4. **Aggressive fusion**
   - Uses "rewrite rules," an old feature of GHC

Main advance: an optimizing data-parallel compiler implemented by modest enhancements to a full-scale functional language implementation.

Step 0: Desugaring

- Rewrite Haskell source into simpler core, e.g, removing array comprehensions:

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

```
sDotP sv v = sumP (mapP (\(i,f) -> f * (v!i)) sv)
```

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

Step 1: Vectorization

- Replace scalar function `f` by the lifted (vectorized) version, written `f^`.

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

```
svMul sv v = sumP (snd^ sv  *^  bpermuteP v (fst^ sv))
```

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

Vectorization: Basic idea

```
mapP f v   =>   f^ v
```

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

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

Vectorization: Basic idea

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

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

Source	Transformed to…
Locals, x	x
Globals, g	g^
Constants, k	replicateP (lengthP x) k

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

Vectorization: Problem

- How do we lift functions that have already been lifted?

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

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

Yet another version of g???

Vectorization: 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:]:]
```

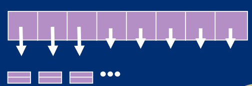
Shape — Flat data — Nested data

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 also Much Too Slow.

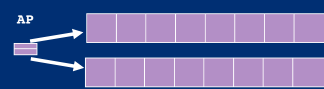


Idea!
Select representation of array based on its element type…

Step 2: Representing arrays

- Extend Haskell with construct to specify families of data structures each with a different implementation.

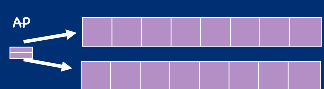
```
data family [:a:]

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



[POPL05], [ICFP05], [TLDI07]

Step 2: Representing arrays

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

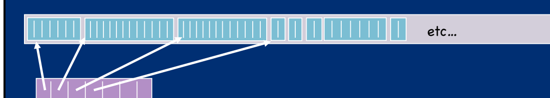


- Now `*^` can be a fast loop because array elements are not boxed.
- And `fst^` is constant time!

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

Step 2: Nested arrays

- Represent nested array as a pair of a shape descriptor and a flat array:



Shape

Flat data

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

Step 2: Nested arrays

- Representation supports operations needed for lifting efficiently:

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!

Step 3: Distribution

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

1. Distribution: 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 the results of each chunk.

Step 2 alone is not good on a parallel machine!

Expressing distribution

- Introduce new type to mark distribution.
  - Type `Dist a` denotes collection of distributed `a` values.
- (Selected) Operations:
  - `splitD`: Distribute data among processors.
  - `joinD`: Collect result data.
  - `mapD`: Run sequential function on each processor.
  - `sumD`: Sum numbers returned from each processor.

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

Distributing sumP

`sumP` is the composition of more primitive functions:

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

```
xs = [: 2,1,4,9,5 :]
```

splitD

Processor 1

```
xs1 = [: 2,1,4 :]
t1  = mapD sumS xs1
    = 7
```

Processor 2

```
xs2 = [: 9,5 :]
t2  = mapD sumS xs2
    = 14
```

mapD

sumD

```
result = 21
```

Distributing Lifted Multiply

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

```
xs = [: 2,1,4,9,5 :]
ys = [: 3,2,2,1,1 :]
```

splitD

Processor 1

```
xs1 = [: 2,1,4 :]
ys1 = [: 3,2,2 :]
zs1 = zipD …
t1  = mapD mulS zs1
    = [: 6,2,8 :]
```

Processor 2

```
xs2 = [: 9,5 :]
ys2 = [: 1,1 :]
zs2 = zipD …
t2  = mapD sumS zs2
    = [: 9,5 :]
```

mapD

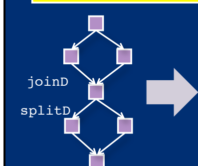
joinD

```
result = [: 6,2,8,9,5 :]
```

Step 4: Fusion

Idea: Rewriting rules eliminate synchronizations.

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



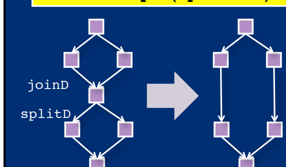
A `joinD` followed by a `splitD` can be replaced by doing nothing.

```
{-# RULE
   splitD (joinD x) = x
#-}
```

Step 4: Fusion

Idea: Rewriting rules eliminate synchronizations.

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



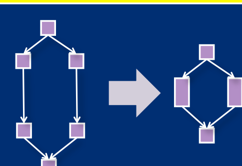
A `joinD` followed by a `splitD` can be replaced by doing nothing.

```
{-# RULE
   splitD (joinD x) = x
#-}
```

Step 4: Fusion

Idea: Rewriting rules eliminate synchronizations.

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



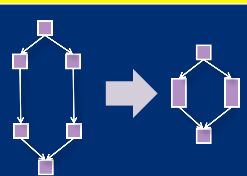
Successive uses of `mapD` can be coalesced, which removes a synchronization point.

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

Step 4: Fusion

Idea: Rewriting rules eliminate synchronizations.

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



Successive uses of mapD can be coalesced, which removes a synchronization point.

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

Step 4: Sequential fusion

```
sDotP :: [:(Int,Float):] -> [:Float:] -> Float
sDotP (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
  - Can't have fusion rules for every pair
- New idea: stream fusion.

Implementation Techniques

Four key pieces of technology:

1. Vectorization
   - Specific to parallel arrays
2. Non-parametric data representations
   - A generically useful new feature in GHC

(1 and 2: Flattening)

3. Distribution
   - Divide up the work evenly between processors
4. Aggressive fusion
   - Uses "rewrite rules," an old feature of GHC

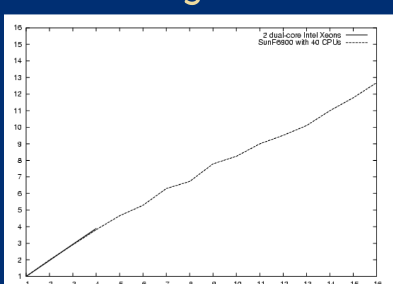
Main advance: an optimizing data-parallel compiler implemented by modest enhancements to a full-scale functional language implementation.

Purity pays off

- Two key transformations:
  - Flattening
  - Fusion
- Both rely on purely-functional semantics:
  - No assignments.
  - Every operation is pure.

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

And it goes fast too...



Speedup of smvm (x-axis is number of PEs)

1-processor version goes only 30% slower than C

Perf win with 2 processors

Data Parallel Summary

- Data parallelism is the most promising 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, but we (think we) know how to do it.
- Functional programming is a massive win in this space.
- Work in progress: starting to be available in GHC 6.10 and 6.12.

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

Candidate Models in Haskell

- Explicit threads
  - Non-deterministic by design
  - Monadic: **forkIO** and **STM**
- Semi-implicit parallelism
  - Deterministic
  - Pure: **par** and **pseq**
- Data parallelism
  - 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 `pseq` a + b
    where
        a = f1 (x-1)
        b = f2 (x-2)
```

The Grand Challenge

- Making effective use of multicore hardware is the challenge for programming languages now.
- Hardware is getting increasingly complicated:
  - Nested memory hierarchies
  - Hybrid processors: GPU + CPU, Cell, FPGA...
  - Massive compute power sitting mostly idle.
- We need new programming models to program new commodity machines effectively.
- Language researchers are working hard to answer this challenge…