Nested data parallelism in Haskell

Simon Peyton Jones (Microsoft)
Manuel Chakravartty, Gabriele Keller, Roman Leshchinskiy
(University of New South Wales)
2009

Paper: “Harnessing the multicores”
At http://research.microsoft.com/~simonpj
Multicore

Road map

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

**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

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

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

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

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

1,000,000's of (small) work items
Nested data parallel

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

```javascript
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:

  **Nested data parallel program** (the one we want to write)

  **Flat data parallel program** (the one we want to run)
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

\[ \text{svMul} :: [(\text{Int}, \text{Float})] \rightarrow [\text{Float}] \rightarrow \text{Float} \]

\[ \text{svMul sv v} = \text{sumP} [f \times (v!i) \mid (i, f) \leftarrow sv :] \]

Parallelism is proportional to length of sparse vector

\( v!i \) gets the i’th element of v
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 Definitions**

```haskell
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

\[
\text{type Doc} = \text{[[: String :]} \n\text{type DocBase} = \text{[[: Document :]} \n\text{search} :: \text{DocBase} \rightarrow \text{String} \rightarrow \text{[[: (Doc,[:Int:])]]} \n\text{wordOccs} :: \text{Doc} \rightarrow \text{String} \rightarrow \text{[: Int :]} \n\]

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

```haskell
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 :]
```

```haskell
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

Step 1  
- sort

Step 2  
- sort
    - sort

Step 3  
- sort
    - sort
    - sort

...etc...

- All subsorts at the same level are done in parallel
- Segment vectors track which chunk belongs to which subproblem
- 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

\[
\text{vecMul} :: [:\text{Float}:] \rightarrow [:\text{Float}:] \rightarrow \text{Float}
\]

\[
\text{vecMul} \; v1 \; v2 = \text{sumP} \; [: \text{f1}\*\text{f2} \mid \text{f1} \leftarrow v1 \; | \; \text{f2} \leftarrow v2 :]
\]

- Do not
  1. Generate [: \text{f1}\*\text{f2} \mid \text{f1} \leftarrow v1 \; | \; \text{f2} \leftarrow 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

- Typecheck
- Desugar
- Vectorise
- Optimise
- Code generation

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

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

Chunking and fusion ("just" library code)
Step 0: desugaring

\[
\text{svMul} :: [:(\text{Int,Float}):] \rightarrow [[:\text{Float}:]] \rightarrow \text{Float}
\]
\[
\text{svMul sv v} = \text{sumP} [\text{f*(v!i)} | (i,f) \leftarrow \text{sv} :]
\]

\[
\text{sumP :: Num a} \Rightarrow [[:a:]] \rightarrow a
\]
\[
\text{mapP :: (a} \rightarrow b) \rightarrow [[:a:]] \rightarrow [[:b:]]
\]

\[
\text{svMul} :: [:(\text{Int,Float}):] \rightarrow [[:\text{Float}:]] \rightarrow \text{Float}
\]
\[
\text{svMul sv v} = \text{sumP} (\text{mapP} (\lambda(i,f) \rightarrow f \times (v!i)) \text{ sv})
\]
Step 1: Vectorisation

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

Scalar operation * replaced by vector operation *^
Vectorisation: the basic idea

- For every function \( f \), generate its **lifted version**, namely \( f^\wedge \)
- Result: a functional program, operating over flat arrays, with a fixed set of primitive operations \( ^\wedge * \), \( \text{sumP} \), \( \text{fst}^\wedge \), etc.
- Lots of intermediate arrays!
Vectorisation: the basic idea

\[
\begin{align*}
 f &:: \text{Int} \to \text{Int} \\
 f \ x &= x + 1 \\
 f^ &:: [\text{Int}] \to [\text{Int}] \\
 f^ \ x &= x +^ (\text{replicateP} \ (\text{lengthP} \ x) \ 1)
\end{align*}
\]

<table>
<thead>
<tr>
<th>This</th>
<th>Transforms to this</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locals, (x)</td>
<td>(x)</td>
</tr>
<tr>
<td>Globals, (g)</td>
<td>(g^)</td>
</tr>
<tr>
<td>Constants, (k)</td>
<td>(\text{replicateP} \ (\text{lengthP} \ x) \ k)</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{replicateP} &:: \text{Int} \to a \to [\text{a}] \\
\text{lengthP} &:: [\text{a}] \to \text{Int}
\end{align*}
\]
Vectorisation: the key insight

\[ f :: [\text{Int}:] \rightarrow [\text{Int}:] \]
\[ f \ a = \text{mapP } g \ a = g^\ a \]

\[ f^ \ :: [+[\text{Int}:]:] \rightarrow [+[\text{Int}:]:] \]
\[ f^ \ a = g^{^\ a} \quad --??? \]

Yet another version of \( g \)???
Vectorisation: the key insight

\[
f :: [:Int:] \rightarrow [:Int:] \\
f a = mapP g a = g^\ a \]

\[
f^\ :: [:[[:Int:]:]] \rightarrow [:[[:Int:]:]] \\
f^\ a = segmentP a (g^\ (concatP a)) \]

\[
concatP :: [:[[:a:]:]] \rightarrow [:a:] \\
segmentP :: [:[[:a:]:]] \rightarrow [:b:] \rightarrow [:[[:b:]]:] \]

Payoff: \(f\) and \(f^\) are enough. No \(f^{^^}\)

First concatenate, then map, then re-split.
Step 2: Representing arrays

Arrays of pointers to boxed numbers are Much Too Slow

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]

```haskell
data family [:a:]

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

- Now \(^\ast\) is a fast loop
- And \(\text{fst}^\ast\) is constant time!

```haskell
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

\[ f :: T1 \to T2 \to T3 \]

\[
\begin{align*}
\text{f1}^\wedge &:: [:T1:] \to [:T2:] \to [:T3:] \quad \text{-- f1}^\wedge = \text{zipWithP f} \\\n\text{f2}^\wedge &:: [:T1:] \to [:(T2 \to T3):] \quad \text{-- f2}^\wedge = \text{mapP f}
\end{align*}
\]

- \( \text{f1}^\wedge \) 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
- \( \text{f2}^\wedge \) forces us to find a representation for [:(T2->T3):]. Closure conversion [PAPPO06]
Step 3: chunking

- **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

---

```
svMul :: [:((Int,Float):)] -> [:Float:] -> Float
svMul (AP is fs) v = sumP (fs *^ bpermuteP v is)
```
Step 3: Chunking

1. **Chunking**: Divide is, fs into chunks, one chunk per processor

2. **Fusion**: Execute \( \text{sumP} (fs \ ^\^ \ bpermuteP v \ is) \) in a tight, sequential loop on each processor

3. **Combining**: Add up the results of each chunk

\[
\text{svMul} :: [:\text{(Int,Float)}:] \rightarrow [:\text{Float}:] \rightarrow \text{Float}
\]

\[
\text{svMul} \ (\text{AP} \ is \ fs) \ v = \text{sumP} \ (fs \ ^\^ \ bpermuteP \ v \ is)
\]

Step 2 alone is not good for a parallel machine!
Expressing chunking

- **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

```haskell
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!
```
Expressing chunking

\[ ^* : \text{[Float]} \to \text{[Float]} \to \text{[Float]} \]
\[ ^* \text{xs ys} = \text{joinD} (\text{mapD mulS} \\
(\text{zipD (splitD xs) (splitD ys)})) \]

- Again, mulS is a tight, sequential loop

\[
\text{splitD} :: \text{[a]} \to \text{Dist [a]}
\]
\[
\text{joinD} :: \text{Dist [a]} \to \text{[a]}
\]
\[
\text{mapD} :: (a \to b) \to \text{Dist a} \to \text{Dist b}
\]
\[
\text{zipD} :: \text{Dist a} \to \text{Dist b} \to \text{Dist (a,b)}
\]
\[
\text{mulS} :: (\text{[Float]}, \text{[Float]}) \to \text{[Float]}
\]
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.)
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?
Speedup for SMVN on 8-core UltraSparc

1 = Speed of sequential C program on 1 core
   = a tough baseline to beat
Less good for Barnes-Hut

![Bar chart showing speedup vs. number of processors for Barnes-Hut algorithm. The x-axis represents the number of processors (1, 2, 3, 4), and the y-axis represents speedup. The chart shows a significant increase in speedup as the number of processors increases.]
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

- Problem:
  - lots and lots of functions over lists
  - and they are \textbf{recursive} functions
- New idea: make map, filter etc non-recursive, by defining them to work over \textit{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       -> []
Stream fusion for lists

\[
\text{mapStream} :: (a \to b) \to \text{Stream}
\ a \to \text{Stream}
\ b
\]

\[
\text{mapStream}
\ f
\ (S
\ \text{step}
\ s)
\ =
\ S
\ \text{step}'
\ s
\]

where

\[
\text{step}'
\ s
\ =
\ \text{case}
\ \text{step}
\ s
\ \text{of}
\]

\[
\text{Done}
\ \to
\ \text{Done}
\]

\[
\text{Yield}
\ a
\ s'
\ \to
\ \text{Yield}
\ (f
\ a)
\ s'
\]

map :: (a \to b) \to [a] \to [b]

\[
\text{map}
\ f
\ \text{xs}
\ =
\ \text{fromStream}
\ \text{(mapStream}
\ f
\ \text{(toStream}
\ \text{xs}))
\]
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

```haskell
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