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...

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
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.

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.

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

Blur 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

x `par` (y + x)
y is evaluated
x is evaluated
x is sparked
x fizzles

Example: Two Processors

x `par` (y + x)
y is evaluated on P1
x is taken up for evaluation on P2
x is sparked on P1
Model: Two Processors

Model: One Processor

No parallelism?

Lucky parallelism

A second combinator: pseq

Using pseq
ThreadScope

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

Test 2

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Idle)</td>
<td>(Busy)</td>
</tr>
</tbody>
</table>

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

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

Performance Numbers

- 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 ()
main = do
  ch <- newChan
  forkIO (ioManager ch)
  forkIO (worker 1 ch)

f :: Int -> Int
f x = a `par` b `pseq` a + b
  where
    a = f1 (x-1)
    b = f2 (x-2)
```

Multicore

**Parallel programming essential**

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

- Data parallelism
  - Operate simultaneously on bulk data

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

Road map

- **Modest parallelism**
  - Hard to program

- **Data parallelism**
  - Operate simultaneously on bulk data
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

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!

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)

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

Data Parallel Haskell
- NESL (Blelloch)
  A mega-breakthrough but:
  - specialized, prototype
  - first order
  - few data types
  - in Haskell
  - interpreted

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

Substantial improvement in
- Expressiveness
- Performance
Array comprehensions

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

\[
\text{vecMul} \ v_1 \ v_2 = \text{sumP} \ [ f_1 \cdot f_2 | f_1 \gets v_1 \ | f_2 \gets v_2 ]
\]

Operations over parallel arrays 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:

\[
\{ (0,3), (2,10) \}
\]

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

\[
\text{sDotP} \ sv \ v = \text{sumP} \ [ f \cdot (v!i) | (i,f) \gets sv ]
\]

NB: no locks!

Parallelism is proportional to length of sparse vector.

Sparse matrix multiplication

A sparse matrix is a vector of sparse vectors:

\[
\{ ([1,3],[4,10]), ([0,2],[1,12],[4,6]) \}
\]

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

\[
\text{smMul} \ sm \ v = \text{sumP} \ [ \text{sDotP} \ sv \ v | sv \gets sm ]
\]

Nested data parallelism here!

We are calling a parallel operation, \text{sDotP}, on every element of a parallel array, \text{sm}.

Example: Data-parallel Quicksort

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

\[
\text{sort} \ a = \begin{cases} 
\text{if} \ (\text{lengthP} \ a \leq 1) \ \text{then} \ a \\
\text{else} \ \text{sa}!0 \ \text{++: eq} \ \text{++: sa}!1 
\end{cases}
\]

where

\[
\begin{align*}
\text{p} &= a!0 \\
\text{lt} &= [ f | f < \text{p}, f \text{ in } a ] \\
\text{eq} &= [ f | f < \text{p}, f = \text{p} ] \\
\text{gr} &= [ f | f < \text{p}, f > \text{p} ] \\
\text{sa} &= [ \text{sort} \ a | a \leftarrow [ \text{lt}, \text{gr} ] ]
\end{align*}
\]

Parallel filters

2-way nested data parallelism here.

Example: Parallel Search

\[
\begin{align*}
\text{type Doc} &= \{ \text{String} \} \rightarrow \text{sequence of words} \\
\text{type Corpus} &= \{ \text{Doc} \} \rightarrow \text{Document} \\
\text{search} : &\ Corpus \rightarrow \text{String} \rightarrow \{ (\text{Doc},[\text{Int}]) \}
\end{align*}
\]

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

\[
\begin{align*}
\text{type Doc} &= \{ \text{String} \} \\
\text{type Corpus} &= \{ \text{Doc} \} \\
\text{search} : &\ Corpus \rightarrow \text{String} \rightarrow \{ (\text{Doc},[\text{Int}]) \}
\end{align*}
\]

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 ss = [: (d, is) |
    d <- ds , let is = wordOccs d ss
    , not (nullP is) :
] where
    wordOccs :: Doc -> String -> [: Int :]
    wordOccs d s = [: i |
        (i, s2) <- zipP positions d
        , s == s2 :
    ]
    positions :: [: Int :]
    positions = [: 1..lengthP d :
    ]
nullP :: [:a:] -> Bool
```

Hard to implement well!
- Evenly chunking at top level might be **ill-balanced**.
- Top level alone might **not be very parallel**.

```
- Concatenate sub-arrays into one big, flat array.
- Operate in parallel on the big array.
- Segment vector tracks extent of sub-arrays.
- Lots of tricky 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.

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.

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
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:

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

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

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

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

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

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

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

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

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

Payoff: \( f \) and \( f^\wedge \) are enough. No \( f^{^\wedge} \).
Step 2: Representing arrays

- Arrays of pointers to boxed numbers are Much Too Slow.
- Arrays of pointers to pairs are also Much Too Slow.

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

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:]
```

AP

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

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

Surprise: concatP, segmentP are constant time!

Step 2: Nested arrays

- Represent nested array as a pair of a shape descriptor and a flat array:
  
  Data instance 
  
  Representation supports operations needed for lifting efficiently:

  Distribution: Divide is, fs into chunks, one chunk per processor.
  Fusion: Execute sumP (fs ^ hpermuteP v) in a tight sequential loop on each processor.
  Combining: Add the results of each chunk.

Step 3: Distribution
# 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.

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

## Distributing Lifted Multiply

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

### Processor 1
- `xs = [2,1,4,9,5]`
- `ys = [2,2,2,1,1]`

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs = [2,1,4]</code></td>
<td><code>xs = [9.5]</code></td>
</tr>
<tr>
<td><code>ys = [2,2,2]</code></td>
<td><code>ys = [1.1]</code></td>
</tr>
<tr>
<td><code>zs = zipD t1</code></td>
<td><code>zs = zipD t2</code></td>
</tr>
<tr>
<td><code>t1 = mapD mulS xs</code></td>
<td><code>t2 = mapD mulS ys</code></td>
</tr>
<tr>
<td><code>t1 = [9.5]</code></td>
<td><code>t2 = [9.5]</code></td>
</tr>
</tbody>
</table>

### Result
- `result = [6,2,8,9,5]`

## Distributing sumP

**sumP** is the composition of more primitive functions:

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

### Processor 1
- `xs = [2,1,4,9,5]`

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs = [2,1,4]</code></td>
<td><code>xs = [9.5]</code></td>
</tr>
<tr>
<td><code>t1 = mapD sumS xs</code></td>
<td><code>t2 = mapD sumS xs</code></td>
</tr>
<tr>
<td><code>t1 = 14</code></td>
<td><code>t2 = 14</code></td>
</tr>
</tbody>
</table>

### Result
- `result = 21`

## Step 4: Fusion

Idea: Rewriting rules eliminate synchronizations.

```
sDotP :: [:((Int,Float)):] -> [:Float:] -> Float
sDotP AP is fs = mapD mulS $(zipD (splitD fs) (splitD (bpermuteP v is)))
```

### Processor 1
- `xs = [2,1,4,9,5]`

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs = [2,1,4]</code></td>
<td><code>xs = [9.5]</code></td>
</tr>
<tr>
<td><code>t1 = mapD mulS xs</code></td>
<td><code>t1 = mapD mulS xs</code></td>
</tr>
<tr>
<td><code>t1 = 14</code></td>
<td><code>t1 = 14</code></td>
</tr>
</tbody>
</table>

### Result
- `result = 21`

### Step 4: Fusion

Idea: Rewriting rules eliminate synchronizations.

```
sDotP :: [:((Int,Float)):] -> [:Float:] -> Float
sDotP AP is fs = mapD mulS $(zipD (splitD fs) (splitD (bpermuteP v is)))
```

### Processor 1
- `xs = [2,1,4,9,5]`

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs = [2,1,4]</code></td>
<td><code>xs = [9.5]</code></td>
</tr>
<tr>
<td><code>t1 = mapD mulS xs</code></td>
<td><code>t1 = mapD mulS xs</code></td>
</tr>
<tr>
<td><code>t1 = 14</code></td>
<td><code>t1 = 14</code></td>
</tr>
</tbody>
</table>

### Result
- `result = 21`

Successive uses of `mapD` can be coalesced, which removes a synchronization point.
### Step 4: Fusion

**Idea:** Rewriting rules eliminate synchronizations.

\[
\text{mapD} \ f \ (\text{mapD} \ g \ x) = \text{mapD} \ (f \cdot g) \ x
\]

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

- 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
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

### Data Parallel Summary

- **Data parallelism** is the most promising way to harness 100% 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](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)
```

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...