The Grand Challenge

How to properly use multi-cores?
Need new programming models!

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

Candidate models in Haskell

**Explicit threads**
- Non-deterministic by design
- Monadic: `forkIO` and `STM`

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

**Semi-implicit parallelism**
- Deterministic
- Pure: `par` and `pseq`

**Data parallelism**
- Deterministic
- Pure: parallel arrays
- Memory: Shared -> Distributed -> GPUs?
The `\par\` Combinator

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

\[
\text{blurRows \par} (\text{mix blurCols blurRows})
\]
The GHC Runtime

- Multiple virtual CPUs
  - Each virtual CPU has a pool of OS threads.
  - CPU-local spark pools for additional work.
  - Work-stealing queue to run sparks.
- Lightweight Haskell threads map many-to-one onto OS threads.
- Automatic thread migration and load balancing.
- Parallel, generational garbage collection.

The meaning of `par`

`par` does not guarantee new Haskell thread

Hint that it may be good to evaluate the first argument in parallel.

`par` is very cheap

Runtime converts spark depending on load.

Programmers can safely use it anywhere. Over-approximates program parallelism.

Example: One processor

```
x `par` (y + x)
```

- `y` is evaluated
- `x` is evaluated
- `x` is sparked
- `x` fizzes
**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**

- `a `par` b`
  - spark for a created
  - eval b
  - thread 2
    - eval a
    - spark converted to thread
    - time

**Model: One Processor**

- `f `par` (f + e)`
  - spark for f created
  - thread 1
    - eval f
    - eval e
    - eval f + e

**No parallelism?**

- No extra resources, so spark for f fizzes
- Main thread demands f, so spark fizzles
That’s lame! How to force computation?

A second combinator: `pseq`

\[
pseq :: a \rightarrow b \rightarrow b \\
x \text{ `pseq` } y
\]

evaluate x in the current thread then return y

\[
x \text{ `pseq` } y = \text{diverge if } x \text{ diverge} \\
= y \text{ otherwise.}
\]

Control evaluation order

\[
f \text{ `par` } (e \text{ `pseq` } (f + e))
\]
A second combinator: \texttt{pseq}

\begin{align*}
\text{f \texttt{par} (e \texttt{pseq} (f + e))}
\end{align*}

ThreadScope: Generated Event Logs

\textbf{Visualize and track spark behavior}

\begin{itemize}
\item Thread 1
\item Thread 2 (Idle)
\end{itemize}

\texttt{f `par` (f + e)}

\textbf{Sample Program}

\begin{verbatim}
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
\end{verbatim}

\begin{itemize}
\item \texttt{fib} and \texttt{sumEuler} are unchanged
\end{itemize}

\textbf{Strategies}

\textbf{Performance Numbers}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{performance.png}
\caption{Performance Numbers}
\end{figure}
**Summary:** Semi-implicit parallelism

Deterministic: Parallel result = Sequential result
No races or errors.

**Good for Reasoning**

Erase `par`, `pseq` to get the original program

**Cheap:** Sprinkle `par` measure and refine

---

**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
- Memory: Shared -> Distributed -> GPUs?

---

**Data Parallelism**

**Flat Data Parallel**
Apply *Sequential Op* Over Bulk Data

**Nested Data Parallel**
Apply *Parallel Op* Over Bulk Data

**Brand Leader**
Fortran, MPI, Map-Reduce

**Limited applicability**
Dense matrix, Map-Reduce

**Well developed**
Limited new opportunities

**Recently developed (90s)**
Wider Applicability
Sparse matrix, Graph Algs, Games.

**Practically Undeveloped**
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

1,000,000’s of small work items

Easy to implement (“chunking”)

Good cost model

Nested Data Parallel

Main idea: Allow *something* to be parallel

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

Parallelism is recursive & unbalanced

Still Good Cost Model

But hard to implement!

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

Nested DP Great for Programmers

Modularity opens up range of applications

Divide and conquer (sort)

Nested DP Great for Programmers

Modularity opens up range of applications

Graph Algorithms

(Shortest Paths, Spanning Trees)
Nested DP Great for Programmers

Modularity opens up range of applications
Sparse Arrays, Variable-Grid Adaptive Methods (Barnes-Hut)

But, Nested DP hard for compilers!
As NDP “tree” is irregular and fine-grained
But it can be done! [NESL, Blelloch 1995]

Key idea: “Flattening Transformation”

Nested Data Parallel Code (what we want to write)  Compiler  Flat Data Parallel Code (what we want to run)
Data Parallel Haskell

Substantial improvement in
- Expressiveness
- Performance

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

- Shared memory now
- Distributed memory later
- GPUs someday?

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

Array Comprehensions

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.”

NOTE: no locks!

Sparse Vector Multiplication

A sparse vector represented as 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 i-th 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:] :]
```

2-way nested data parallelism here.

Parallel filters

Segment vectors map chunks to sub-problems
Instant insanity when done by hand

Example: Parallel Sub-Sorts At Same Level

```
Step 1

Step 2

Step 3

...etc...
```

Example: Parallel Search

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

type Corpus = [: Doc :]

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)

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

type Corpus = [: Doc :]

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

Find all the places where a string is mentioned in a doc (e.g. word 45, 99).
The Flattening Transformation

Concatenate sub-arrays into one big flat array
Operate in parallel on the big array
The Flattening Transformation

Concatenate sub-arrays into one big flat array

Segment vector tracks extent of sub-arrays

Lot of Tricky Book-Keeping

Possible, but hard to do by hand

The Flattening Transformation

Fusion

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

Lot of Tricky Book-Keeping

Blelloch showed how to do it systematically

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

Bad idea: Generate Intermediate Array
- [: f1*f2 | f1 <- v1 | f2 <- v2 :]
- Add elements of this big intermediate vector
Fusion

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

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

Good Idea: Multiply and Add in Same Loop!
- That is, fuse the multiply loop with add loop
- Very general, aggressive fusion is required

VecMul v1 v2 = sumP [: f1*f2 | f1 <- v1 | f2 <- v2 :]

Step 0: Desugaring

Rewrite Haskell source into simpler core
e.g., remove array comprehensions

sDotP sv v = sumP [: x * (v!i) | (i,x) <- sv :]

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

Step 1: Vectorization

Replace scalar-fun f by vector-fun f^\

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

sDotP sv v = sumP (msp^ sv *^ bpermuteP v (fst^ sv))

Step Implementation Technique

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
Vectorization (Basic idea)

mapP $f$ $v$ $\rightarrow$ $f^\wedge$ $v$

For every function $f$, generate the lifted version $f^\wedge$
Resulting program operates over flat arrays with a fixed set of primitive operations: $*^\wedge$, $\text{sumP}$, $\text{fst}^\wedge$, ...

Vectorization (Basic idea)

But, lots of intermediate arrays!

Vectorization (Basic idea)

$f :: \text{Int} \rightarrow \text{Int}$
$f \ x = x + 1$

$f^\wedge :: [[:\text{Int}:]] \rightarrow [[:\text{Int}:]]$ 
$f^\wedge$ $xs = xs +^\wedge$ (replicateP $\left(\text{lengthP} \ xs\right)$ $1$)

Vectorization: Problem

How to lift functions that have already been lifted?

$f :: [[:\text{Int}:]] \rightarrow [[:\text{Int}:]]$
$f \ xs = \text{mapP} \ g \ xs = g^\wedge$ $xs$

$f^\wedge :: [[:[[:\text{Int}:]]:] : ] \rightarrow [[:[[:\text{Int}:]]:] : ]$
$f^\wedge$ $xss = g^\wedge^\wedge$ $xss$ 

Yet another version of $g$???
Vectorization: Key insight

**First concat, then map, then re-split**

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

\[
f^:: [:[:\text{Int}:]:] \rightarrow [:[:\text{Int}:]:]
f^\ xsss = \text{segmentP} \ xsss (g^\ (\text{concatP} \ xsss))
\]

**Shape**

**Flat data**

**Nested data**

**Payoff:** \( f, f^ \) are enough (avoid \( f^{^\^} \))

### Step 2: Representing Arrays

Extend Haskell with construct to specify families of structures with a different implementations.

**Data family**

\[
data\ family\ [:a:]
\]

**Data instance**

\[
data\ instance\ [:\text{Double}:] = \text{AD} \ \text{Int} \ \text{ByteArray}
data\ instance\ [::(a, b):] = \text{AP} [:a:] [:b:]
\]

**Data family**

**Data instance**

**AP**

Now *\(^\^\)* can be a fast loop

Array elements are not boxed

**References:** [POPL05], [ICFP05], [TLDI07]

Too Slow: Array of pointers to boxed nums
Too Slow: Arrays of pointers to pairs

Idea: Pick representation using element type...
### Step 2: Representing Arrays

<table>
<thead>
<tr>
<th>data family</th>
<th>[:a:]</th>
</tr>
</thead>
<tbody>
<tr>
<td>data instance</td>
<td>[:Double:] = AD Int ByteArray</td>
</tr>
<tr>
<td>data instance</td>
<td>[:(a, b):] = AP [:a:] [:b:]</td>
</tr>
</tbody>
</table>

**And $\text{fst}^\wedge$ is constant time!**

\[
\text{fst}^\wedge :: [:(a, b):] \rightarrow [:a:]
\]

\[
\text{fst}^\wedge (\text{AP as bs}) = \text{as}
\]

### Step 2: Nested arrays

Represent with (shape descriptor, flat array)

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

### Step 2: Nested arrays

Representation supports efficient op-lifting

<table>
<thead>
<tr>
<th>f :: T1 -&gt; T2 -&gt; T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1^ :: [:T1:] -&gt; [:T2:] -&gt; [:T3:]</td>
</tr>
<tr>
<td>f2^ :: [:T1:] -&gt; [:T2 -&gt; T3:]</td>
</tr>
</tbody>
</table>

**Surprise:** concatP, segmentP are constant time!
Step 3: Distribution

After steps 0-2,
Program = flat arrays + vec-ops (*^, sumP...)

NESL directly executes this version
Hand-coded assembly for primitive ops
All the time is spent here anyway

But, many intermediate arrays!
Increase memory traffic, synchronization

Idea: Distribution and Fusion

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

Expressing Distribution

New type Dist a
Describes a collection of distributed `a` values

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

(Selected) Operations
splitD Distribute data among processors
joinD Collect result data
mapD Run sequential function on each processor
sumD Sum numbers from each processor

Step 2 alone is not good on a parallel machine!
Distributing sumP

sumP is Composition of Primitive Functions

\[ \text{sumP} :: [:\text{Float}:] \rightarrow \text{Float} \]
\[ \text{sumP} \; \text{xs} = \text{sumD} \; (\text{mapD} \; \text{sumS} \; \text{splitD} \; \text{xs}) \]

Processor 1

\[ \text{xs} = [:2,1,4,9,5:] \]
\[ \text{xs}_1 = [:2,1,4:] \]
\[ t_1 = \text{sumS} \; \text{xs}_1 \]
\[ t_1 = 7 \]
\[ \text{xs}_2 = [:9,5:] \]
\[ t_2 = \text{sumS} \; \text{xs}_2 \]
\[ t_2 = 14 \]
\[ \text{result} = 21 \]

Processor 2

\[ \text{splitD} \]
\[ \text{mapD} \; \text{sumS} \]
\[ \text{sumD} \]
\[ \text{result} = 21 \]

mapD : The Source of Parallelism

Processor 1

\[ \text{xs} = [:2,1,4,9,5:] \]
\[ \text{xs}_1 = [:2,1,4:] \]
\[ t_1 = \text{sumS} \; \text{xs}_1 \]
\[ t_1 = 7 \]
\[ \text{xs}_2 = [:9,5:] \]
\[ t_2 = \text{sumS} \; \text{xs}_2 \]
\[ t_2 = 14 \]
\[ \text{result} = 21 \]

Processor 2

\[ \text{splitD} \]
\[ \text{mapD} \; \text{sumS} \]
\[ \text{sumD} \]
\[ \text{result} = 21 \]

sumS : A Tight Sequential Loop

Processor 1

\[ \text{xs} = [:2,1,4,9,5:] \]
\[ \text{xs}_1 = [:2,1,4:] \]
\[ t_1 = \text{sumS} \; \text{xs}_1 \]
\[ t_1 = 7 \]
\[ \text{xs}_2 = [:9,5:] \]
\[ t_2 = \text{sumS} \; \text{xs}_2 \]
\[ t_2 = 14 \]
\[ \text{result} = 21 \]
**sumD: Collecting the Result**

- **splitD**: 
  \[\text{:[:a:] -> Dist [:a:]\}]

- **mapD**: 
  \[(a->b) -> Dist a -> Dist b\] 

- **sumD**: 
  \[\text{Dist Float -> Float}\]

- **sumS**: 
  \[\text{[:Float:] -> Float}\]

```
xs = [:2,1,4,9,5 :]
t1 = sumS xs1
  = 7

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

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

**Processor 1**

- `splitD`
- `mapD`
- `sumD`

**Processor 2**

- `splitD`
- `mapD`
- `sumD`

**Result**

- `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 :]
```

**Processor 1**

- `splitD`
- `mapD`
- `mulS`
- `zipD`
- `joinD`

- `sumS` for `xs1` = 7
- `sumS` for `xs2` = 14

**Processor 2**

- `splitD`
- `mapD`
- `mulS`
- `zipD`
- `joinD`

- `sumS` for `zs1` = 6
- `sumS` for `zs2` = 9

**Result**

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

---

**Step 4: Fusion**

**Idea: Rewrite Rules Remove Synchronizations**

```haskell
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))
```

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

---

**Step 4: Fusion**

**Idea: Rewrite Rules Remove Synchronizations**

```haskell
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))
```

- `splitD (joinD x) = x`
- `{# RULE #-}`
Step 4: Fusion

Idea: Rewrite Rules Remove Synchronizations

\[ \text{sDotP} \text{(AP is fs) v} \]
\[ \Rightarrow \text{sumP} (\text{fs} \ ^* \ ^* \ \text{bpermuteP v is}) \]
\[ \Rightarrow \text{sumD} \ . \ \text{mapD} \ \text{sumS} \ . \ \text{mapD} \ \text{mulS} \ $ \]
\[ \text{zipD} (\text{splitD fs}) (\text{splitD (bpermuteP v is)}) \]

Fuse Successive uses of mapD
Removes synchronization points

\[-\# \text{ RULE } \#-\]
\[ \text{mapD f (mapD g x) = mapD (f.g) x} \]

Step 4: Sequential fusion

Now we have a sequential fusion problem
Lots and lots of functions over arrays
Can’t have fusion rules for every pair...

... new idea: Stream Fusion [ICFP 07]

4-Step Implementation Technique

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!

Key optimizations: Flattening & Fusion
Rely on purely-functional semantics

No Assignments, No Shared Memory
Every operation is pure

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

And it goes fast too...

Pinch of salt
1-processor version
30% slower than C
2-processor version
is a performance win

Nested Data Parallel Summary
NDP is a promising way to harness 100’s of cores
Great for programmers: far more flexible than flat DP

NDP is tough to implement
But we (think we) know how to do it.

Functional programming is a big win in this space
Work in progress, available in GHC 6.10 and 6.12.

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

Distribution & Fusion Used Elsewhere
FlumeJava @ Google
Dryad/Linq @ Microsoft
**Candidate models in Haskell**

### Explicit threads
- Non-deterministic by design
- Monadic: `forkIO` and `STM`

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

### Semi-implicit parallelism
- Deterministic
- Pure: `par` and `pseq`

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

### Data parallelism
- Deterministic
- Pure: parallel arrays
- Memory: Shared -> Distributed -> GPUs?

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