Laziness
  Forcing Evaluation with seq
  Weak Head Normal Form

Parallelism
  ThreadScope
  Sparking Parallelism with par
  Sparks
  Limiting Granularity

The Eval Monad: rpar and rseq

Bibliography
This material adapted from

Simon Marlow’s book

https://simonmar.github.io/pages/pcph.html

Mary Sheeran and John Hughes’s class

Laziness in Haskell

Haskell follows a *call-by-need†* evaluation strategy in which expressions are evaluated only when their values are needed and at most once.

```haskell
Prelude> let x = 1 + 2 :: Int
Prelude> :t x
x :: Int
Prelude> :sprint x
x = _
Prelude> x + 1
4
Prelude> :sprint x
x = 3
```

_ denotes an unevaluated “thunk”

†C, Java, etc. are *call-by-value*: arguments are evaluated before a function call; Algol-68 is *call-by-name*: arguments are (re)evaluated at each reference.
Thunks all the way down: seq also forces evaluation

\[ \texttt{seq} :: \texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{b} \]

\[ \texttt{seq} \ x \ y = \text{evaluate} \ x \ \text{and} \ y; \ \text{return} \ y \]

Prelude> \texttt{let} \ x = 1 + 2 :: \texttt{Int}
Prelude> \texttt{let} \ y = x + 1
Prelude> \texttt{:sprint} \ x
x = _
Prelude> \texttt{:sprint} \ y
y = _
Prelude> \texttt{seq} \ y \ ()
()  
Prelude> \texttt{:sprint} \ x
x = 3
Prelude> \texttt{:sprint} \ y
y = 4

[Marlow, Figure 2–2]
Weak Head Normal Form: Lazy Data Structures

Prelude> let x = 1 + 2 :: Int
Prelude> let y = (x, x)
Prelude> let swap(a, b) = (b, a)
Prelude> let z = swap (x,x+1)
Prelude> :sprint z
z = _
Prelude> seq z ()
()  
Prelude> :sprint z
z = (_,_)
Prelude> seq x ()
()  
Prelude> :sprint z
z = (_,3)

Weak head normal form: top is data constructor or lambda, not application

[Marlow, Figure 2–3]
Functions Build Thunks

Prelude> let xs = 
    map (+1) [1..10] :: [Int]
Prelude> :sprint xs
xs = _
Prelude> seq xs ()
()  
Prelude> :sprint xs
xs = _ : _
Prelude> seq (tail xs) ()
()  
Prelude> :sprint xs
xs = _ : _ : _
Prelude> length xs
10  
Prelude> :sprint xs
xs = [_,_,_,_,_,_,_,_,_,_]

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = let x' = f x
               xs' = map f xs
               in x' : xs'

[Marlow, Figure 2–4]
Let’s Speed Up a Dumb† Program

nfib1 :: Integer -> Integer
nfib1 n | n < 2 = 1
nfib1 n = nfib1 (n-1) + nfib1 (n-2) + 1

main :: IO ()
main = print (nfib1 40)

<table>
<thead>
<tr>
<th>n</th>
<th>nfib n</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>177</td>
</tr>
<tr>
<td>20</td>
<td>21891</td>
</tr>
<tr>
<td>25</td>
<td>242785</td>
</tr>
<tr>
<td>30</td>
<td>2692537</td>
</tr>
<tr>
<td>35</td>
<td>29860703</td>
</tr>
<tr>
<td>40</td>
<td>331160281</td>
</tr>
</tbody>
</table>

$ stack ghc -- -O2 \ # Optimize
    -threaded \ # Enable parallel execution
    -rtsopts \ # Enable run–time system flags +RTS
    -eventlog \ # Enable parallel profiling
    nfib1.hs

†This should be iterative, not recursive
Running the Program

$ TIMEFORMAT="real %Rs"  # for bash time builtin
$ time ./nfib1
331160281
real 9.984s
$ time ./nfib1 +RTS -N1  # +RTS = Run Time System, -N1 = 1 core
331160281
real 9.994s
$ time ./nfib1 +RTS -N4  # -N4 = use 4 cores
331160281
real 10.214s
$ time ./nfib1 +RTS -N4 -ls  # -ls = Record events in nfib1.eventlog
331160281
real 10.378s
ThreadScope

ThreadScope: the Haskell parallel execution event log viewer

Under Ubuntu, I was able to install it using Aptitude:

```sh
$ sudo apt install threadscope
```

The Haskell stack may also be able to install it (```stack install threadscope```), but it didn’t work automatically on my machine.

A Haskell executable compiled with `-rtsopts` enables the `+RTS ... -RTS` syntax for passing arguments to the Haskell runtime system.

The `-l` option enables event logging (in a binary file `executable.eventlog`); `s` includes scheduler events.

Google “Haskell Runtime Control” or look in the GHC User Guide.
Only One Thread: Pretty Boring
Asking for Parallelism

In Control.Parallel, (stack install parallel)

\[
\text{par} : \text{a} \rightarrow \text{b} \rightarrow \text{b}
\]

\text{par} x y "sparks" the evaluation of \text{x} in parallel with \text{y}; returns \text{y}.

The run-time system may convert a spark into work for a thread

\[
\text{import Control.Parallel(par)}
\]

\[
\text{nfib2} :: \text{Integer} \rightarrow \text{Integer}
\]

\[
\text{nfib2} n | n < 2 = 1
\]

\[
\text{nfib2} n = \text{par} \text{nf} (\text{nf} + \text{nfib2} (n-2) + 1)
\]

where \text{nf} = \text{nfib2} (n-1)
Performance of nfib2 (using par)

\[
\text{\$ time ./nfib2 +RTS -N8 -ls}
\]

331160281
real 2.604s

A speedup of 7.44: Pretty good for a first try
Sparks

Request a spark

Overflow
Spark pool is full

Created
Enter spark pool

Dud
Already evaluated to WHNF

Fizzled
Evaluated to WHNF after creation

Garbage Collected
Program forgot about it or computed it already

Converted
Evaluated by an available core

$ ./nfib2 +RTS -N8 -s 331160281
SPARKS:
166651588 total
1210 converted,
47083668 overflowed,
0 dud,
117359879 GC'd,
2206831 fizzled

Conclusion: Far too many sparks created; majority were garbage collected; 25% didn’t even fit in the spark pool. Only 1210 (0.0007%) did useful work.

From https://wiki.haskell.org/ThreadScope_Tour
Six Cores Being Kept Busy

Sparks Pool Overflowing

Many Sparks Created

Most Sparks Garbage Collected

Some Sparks Fizzle
Asking more precisely for parallelism

Also in Control.Parallel,

\[
pseq : a \rightarrow b \rightarrow b
\]

Like \texttt{seq}, but only strict in its first argument. \texttt{pseq x y} means “make sure \texttt{x} is evaluated before starting on \texttt{y}”

\[
\text{import Control.Parallel(par, pseq)}
\]

\[
nfib3 :: \text{Integer} \rightarrow \text{Integer}
\]

\[
nfib3 n \mid n < 2 = 1
\]

\[
nfib3 n = nf1 \`par` nf2 \`pseq` nf1 + nf2 + 1 \\
\text{where } nf1 = nfib3 (n-1) \\
\text{nf2 = nfib3 (n-2)}
\]

No visible change in performance; the compiler may have automatically done this for us
Controlling Granularity

We are creating a *lot* of sparks, most of which are pointless:

```
./nfib3 +RTS -N8 -s
SPARKS: 168073361 (  
    2351 converted,  
    48159769 overflowed,  
    0 dud,  
    115072423 GC'd,  
    4838818 fizzled)
```

It doesn’t make sense to be creating 168 million pieces of work when we only have 8 cores on which to do work; only 2351 ever did useful work.

Idea: let’s go parallel *only to a certain depth*
Running Parallel to a Certain Depth

\[
\text{nfib4} :: \text{Int} \to \text{Int} \to \text{Integer}
\]
\[
\text{nfib4} \ 0 \ n \ = \ \text{nfib} \ n
\]
\[
\text{nfib4} \ _ \ n \ | \ n \ < \ 2 \ = \ 1
\]
\[
\text{nfib4} \ d \ n \ = \ \text{nfib4} (d-1) (n-1) \ \text{par} \ \text{nib4} (d-1) (n-2) \ \text{pseq} \ \text{nib4} + \ \text{nib4} + \ 1
\]
\[
\text{where} \ \text{nib4} = \ \text{nib4} (d-1) (n-1)
\]
\[
\text{nib4} = \ \text{nib4} (n-1) + \ \text{nib4} (n-2) + \ 1
\]

Speedup

Computing \text{nfib4} 40 on an 8-thread i7
<table>
<thead>
<tr>
<th>Depth</th>
<th>Sparks</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>converted</td>
<td>GC'ed</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>63</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>127</td>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>511</td>
<td>78</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1026</td>
<td>98</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>2052</td>
<td>162</td>
<td>49</td>
</tr>
<tr>
<td>12</td>
<td>4106</td>
<td>160</td>
<td>436</td>
</tr>
<tr>
<td>13</td>
<td>8226</td>
<td>249</td>
<td>2109</td>
</tr>
<tr>
<td>25</td>
<td>30833310</td>
<td>2855</td>
<td>28605093</td>
</tr>
</tbody>
</table>

3.6 GHz 4-core, 8-thread i7-3820, +RTS -N8 -s, 4-run averages, -O2 -threaded -rtsopts
Depth = 1: Only two-way parallelism
Depth = 4: 16-way parallelism but unbalanced
Depth = 7: 32 sparks, better balancing
Depth = 12: 4000+ sparks, excellent balancing
The Eval Monad

When in doubt, create a Monad

\[
\text{module Control.Parallel.Strategies where}
\]
\[
data \text{Eval} \ a = \ldots
\]
\[
\text{instance Monad Eval where} \ldots
\]
\[
\text{runEval} :: \text{Eval} \ a \rightarrow \ a \quad -- \text{Get the result}
\]
\[
\text{rpar} :: \ a \rightarrow \text{Eval} \ a \quad -- \text{Spark evaluation}
\]
\[
\text{rseq} :: \ a \rightarrow \text{Eval} \ a \quad -- \text{Wait for evaluation to WHNF}
\]
runEval $ do
  a <- rpar (f x)
  b <- rpar (f y)
  return (a, b)

Start parallel evaluation of f x and f y
Return immediately
runEval $ do
  a <- rpar (f x)
  b <- rseq (f y)
  return (a, b)

Start parallel evaluation of f x and f y
Wait for f y to finish, then return

*Do we really know f y is faster?*
runEval $ do
  a <- rpar (f x)
  b <- rseq (f y)
  rseq a
  return (a, b)

Equivalent and symmetrical:

runEval $ do
  a <- rpar (f x)
  b <- rpar (f y)
  rseq a
  rseq b
  return (a, b)

Start parallel evaluation of f x and f y
Wait for both to finish, then return

