Lazy and Parallel Evaluation

Stephen A. Edwards

Columbia University

Fall 2022
Laziness
   Forcing Evaluation with seq
   Weak Head Normal Form

Parallelism
   ThreadScope
   Sparking Parallelism with par
   Sparks
   Limiting Granularity
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\(^\dagger\) evaluation strategy in which expressions are evaluated only when their values are needed and at most once.

\[
\text{Prelude}\gg \text{let } x = 1 + 2 :: \text{Int} \\
\text{Prelude}\gg :t x \\
x :: \text{Int} \\
\text{Prelude}\gg :\text{sprint } x \\
x = _ \\
\text{Prelude}\gg x + 1 \\
4 \\
\text{Prelude}\gg :\text{sprint } x \\
x = 3
\]

_ denotes an unevaluated “thunk”

\(^\dagger\)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: \textit{seq} also forces evaluation

\hspace{1cm} \textbf{seq} :: a \rightarrow b \rightarrow b

\textit{seq} \; x \; y = \text{evaluate } x \text{ and } y; \text{ return } y

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

[Marlow, Figure 2–2]
Weak head normal form: top is data constructor or lambda, not application
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:

```
$ 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)

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

\[
\text{par x y “sparks” the evaluation of x in parallel with y; returns y.}
\]

The run-time system \textit{may} convert a spark into work for a thread

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

\[
\text{nfib2 :: Integer -> Integer}
\]

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

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

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

$ time ./nfib2 +RTS -N8 -ls
331160281
real 2.604s

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

par Request a spark

Overflow Spark pool is full

Created Enter spark pool

Overflow

Dud Already evaluated to WHNF

Converted Evaluated by an available core

Fizzled Evaluated to WHNF after creation

Garbage Collected Program forgot about it or computed it already

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

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

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.
Six Cores Being Kept Busy

Spark Pool Overflowing

Many Sparks Created

Most Sparks Garbage Collected

Some Sparks Fizzle
Asking more precisely for parallelism

Also in Control.Parallel,

```
 pseq : a -> b -> b
```

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

```
import Control.Parallel(par, pseq)

nfib3 :: Integer -> Integer
nfib3 n | n < 2 = 1
nfib3 n = nf1 `par` nf2 `pseq` nf1 + nf2 + 1
  where nf1 = nfib3 (n-1)
    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

nfib4 :: Int -> Int -> Integer
nfib4 0 n = nfib n
nfib4 _ n | n < 2 = 1
nfib4 d n = nf1 `par` nf2 `pseq` nf1 + nf2 + 1
  where nf1 = nfib4 (d-1) (n-1)
        nf2 = nfib4 (d-1) (n-2)

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

Speedup

Computing 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>308333310</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