

Lazy and Parallel Evaluation

Stephen A. Edwards

Columbia University

Fall 2020



Techniques for Multicore and Multithreaded Programming



Parallel and Concurrent Programming in Haskell

O'REILLY*

Simon Marlow

This material adapted from

Simon Marlow's book

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

Mary Sheeran and John Hughes's class

http://www.cse.chalmers.se/edu/year/2018/course/DAT280_Parallel_Functional_Programming/lectures.html

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**.

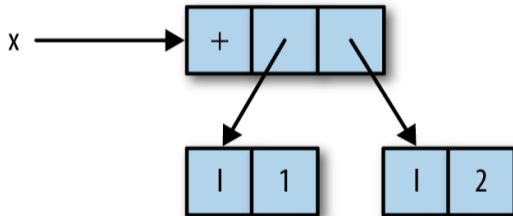
```
Prelude> let x = 1 + 2 :: Int
Prelude> :t x
x :: Int
Prelude> :sprint x
x = _
Prelude> x + 1
4
Prelude> :sprint x
x = 3
```

_ 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



Thunk Crood



[Marlow, Figure 2-1]

Thunks all the way down: seq also forces evaluation

```
seq :: a -> b -> b
```

seq x y = evaluate x and y; return y

```
Prelude> let x = 1 + 2 :: Int
```

```
Prelude> let y = x + 1
```

```
Prelude> :sprint x
```

```
x = _
```

```
Prelude> :sprint y
```

```
y = _
```

```
Prelude> seq y ()
```

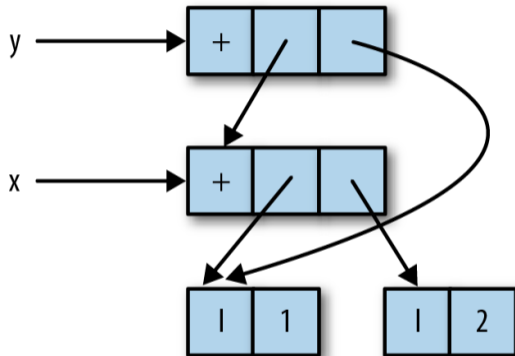
```
()
```

```
Prelude> :sprint x
```

```
x = 3
```

```
Prelude> :sprint y
```

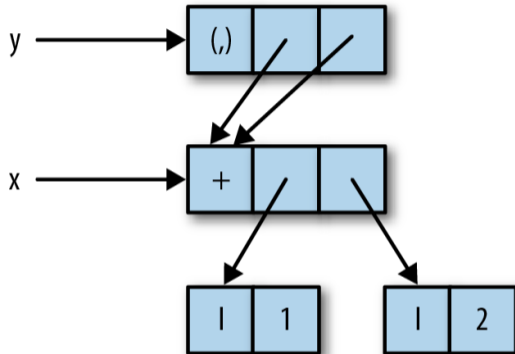
```
y = 4
```



[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)
```



[Marlow, Figure 2-3]

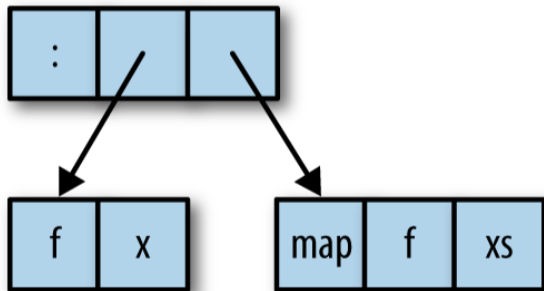
Weak head normal form: top is data constructor or lambda, not application

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  

```



[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)
```

<i>n</i>	nfib <i>n</i>
10	177
20	21891
25	242785
30	2692537
35	29860703
40	331160281

```
$ 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

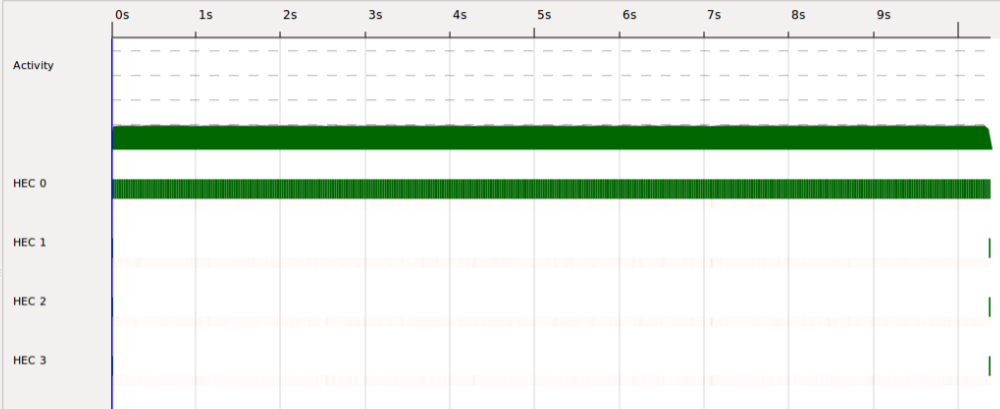
Google "Haskell Runtime Control" or look in the GHC User Guide



Key Traces Bookmarks

Timeline

- running
- GC
- create thread
- seq GC req
- par GC req
- migrate thread
- thread wakeup
- shutdown
- user message
- perf counter
- perf tracepoint
- create spark
- dud spark
- overflowed spark
- run spark
- fizzled spark
- GCed spark



Time

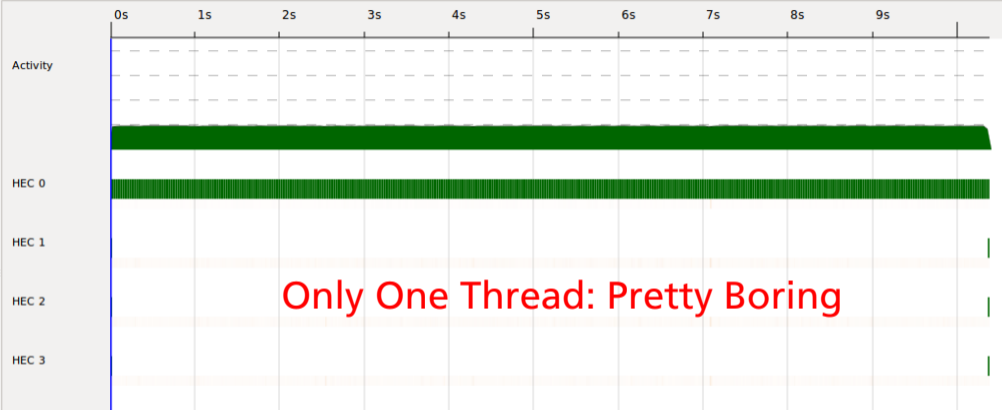
Total time: 10.37s
 Mutator time: 10.21s
 GC time: 0.16s
 Productivity: 98.4% of mutator vs total



Key Traces Bookmarks

Timeline

- running
- GC
- create thread
- seq GC req
- par GC req
- migrate thread
- thread wakeup
- shutdown
- user message
- perf counter
- perf tracepoint
- create spark
- dud spark
- overflowed spark
- run spark
- fizzled spark
- GCed spark



Time Heap GC Spark stats Spark sizes Process info Raw events

Total time: 10.37s
Mutator time: 10.21s
GC time: 0.16s
Productivity: 98.4% of mutator vs total

Asking for Parallelism

In `Control.Parallel`, (`stack install parallel`)

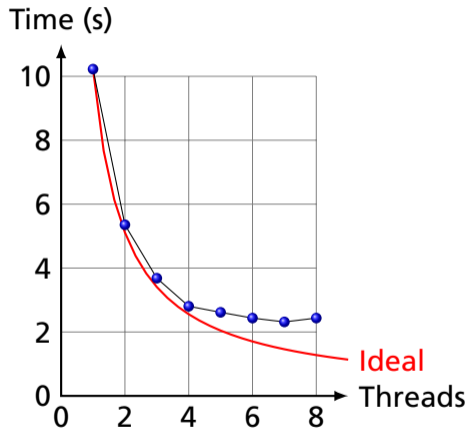
```
par : a -> b -> b
```

`par x y` “sparks” the evaluation of `x` in parallel with `y`; returns `y`.

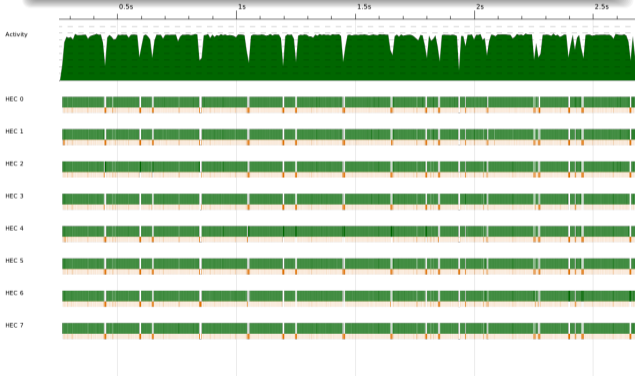
The run-time system *may* convert a spark into work for a thread

```
import Control.Parallel(par)  
  
nfib2 :: Integer -> Integer  
nfib2 n | n < 2 = 1  
nfib2 n = par nf (nf + nfib2 (n-2) + 1)  
  where nf = nfib2 (n-1)
```

Performance of nfib2 (using par)

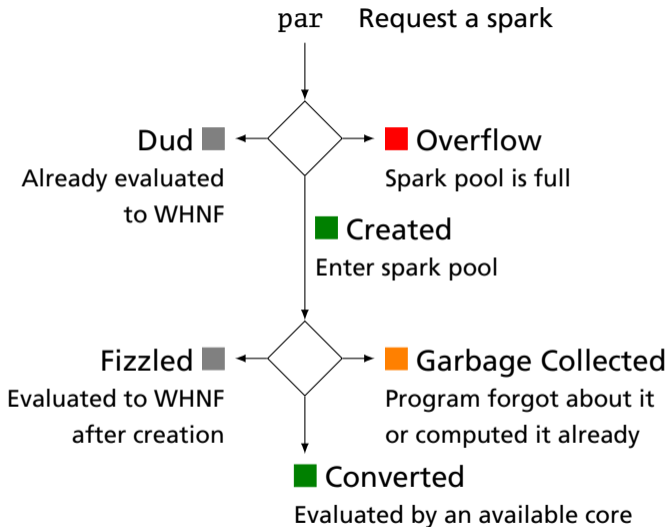


```
$ time ./nfib2 +RTS -N8 -ls
331160281
real 2.604s
```



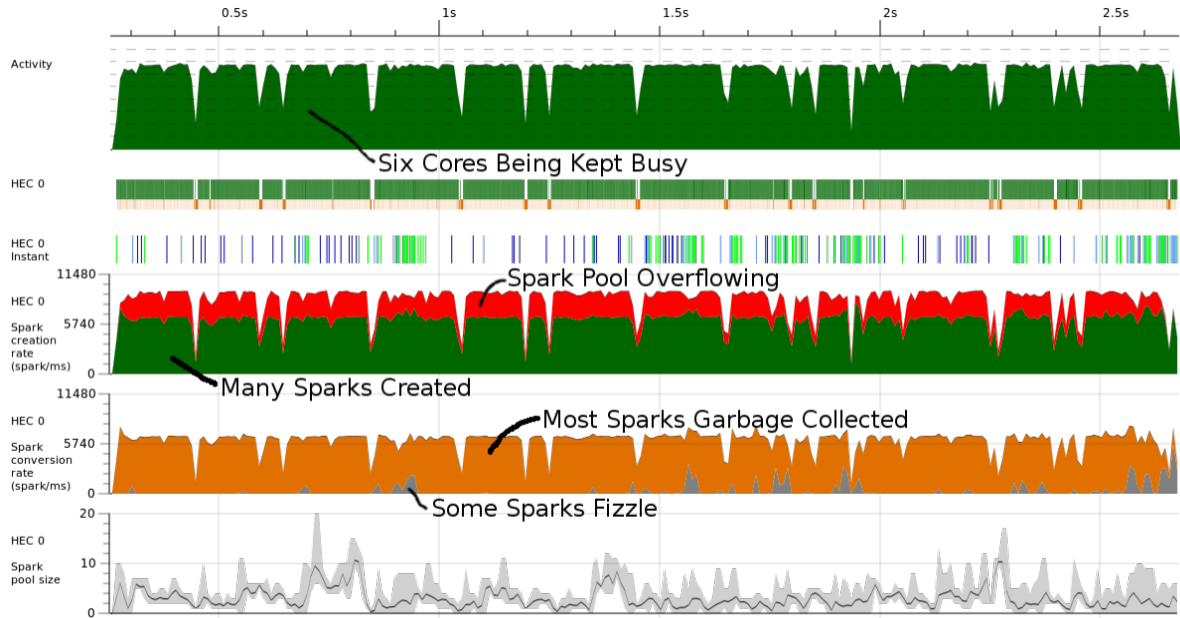
A speedup of 7.44: Pretty good for a first try

Sparks



```
$ ./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.



Asking more precisely for parallelism

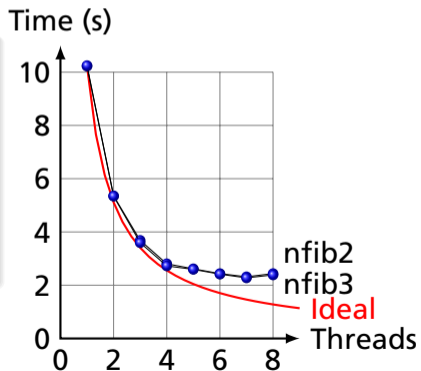
Also in `Control.Parallel`,

```
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)
```

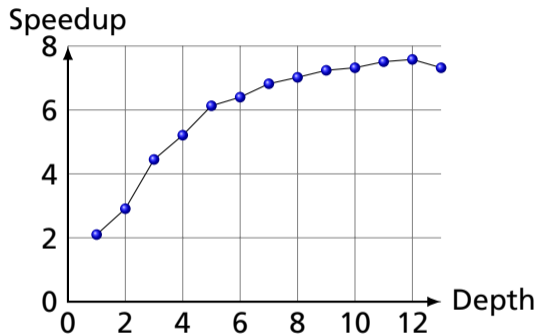
It doesn't make sense to be creating 168 million pieces of work when we only have 8 cores on which to do work; only 2351 ever did useful work.

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
```

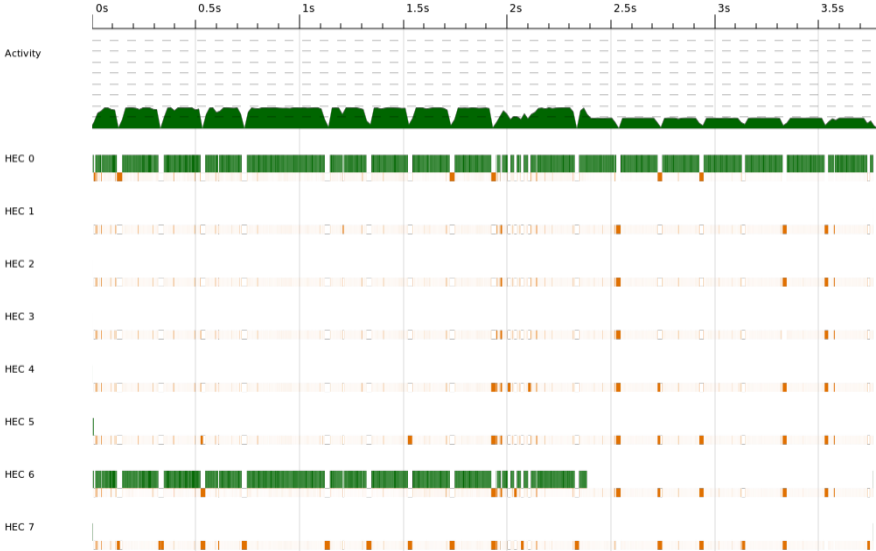


Computing nfib4 40 on an 8-thread i7

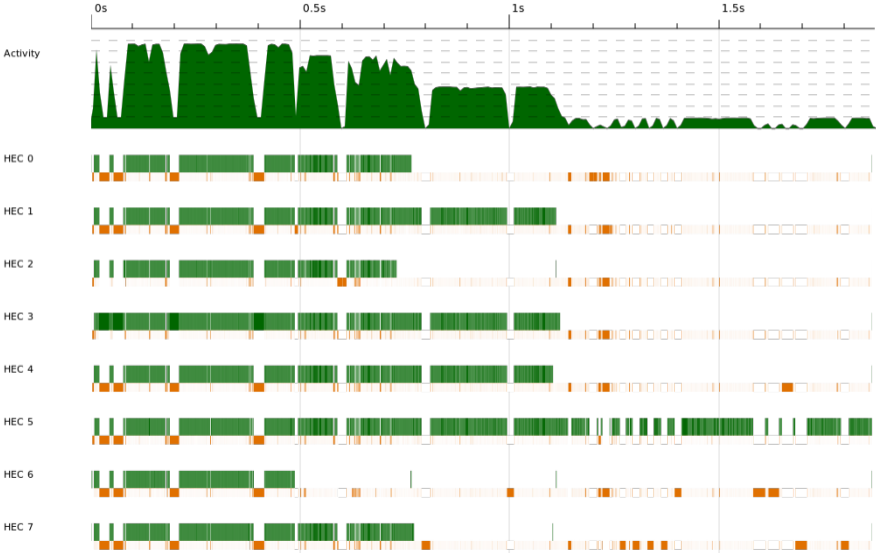
Depth	Sparks				Time (s)		Speedup
	total	converted	GC'ed	fizzled	total	elapsed	
1	1	1	0	0	8.00	3.80	2.10
2	3	3	0	0	6.80	2.34	2.91
3	7	7	0	0	8.83	1.98	4.45
4	15	12	0	2	7.89	1.51	5.21
5	31	19	0	11	7.58	1.24	6.13
6	63	30	0	32	8.14	1.27	6.40
7	127	39	0	87	8.62	1.26	6.82
8	256	48	1	206	7.51	1.07	7.02
9	511	78	0	432	7.57	1.05	7.24
10	1026	98	4	923	7.53	1.03	7.32
11	2052	162	49	1840	7.33	0.98	7.51
12	4106	160	436	3509	7.04	0.93	7.58
13	8226	249	2109	5867	7.62	1.04	7.32
25	30833310	2855	28605093	398402	10.17	1.50	6.77

3.6 GHz 4-core, 8-thread i7-3820, +RTS -N8 -s, 4-run averages, -O2 -threaded -rtsopts

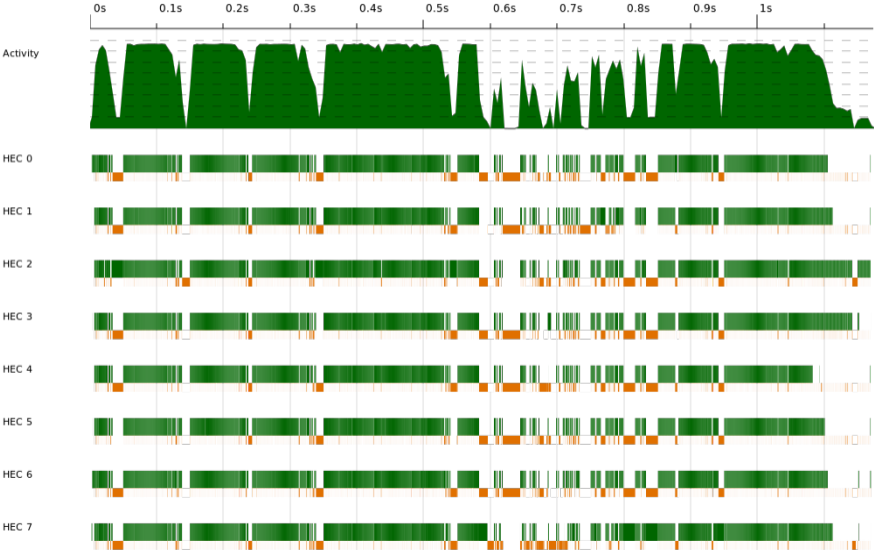
Depth = 1: Only two-way parallelism



Depth = 4: 16-way parallelism but unbalanced



Depth = 7: 32 sparks, better balancing



Depth = 12: 4000+ sparks, excellent balancing

