Parallel Haskell

Troels Henriksen
Slides based on work by Ken Friis Larsen
based on material by Simon Marlow

DIKU
University of Copenhagen

2nd of December, 2019

Agenda

What is Parallel Haskell?

All you need is *X*

- Where *X* is:
  - Actors, threads, transactional memory, futures, love…
- In Haskell, the approach is to give you lots of different *X*s

All you need is *X*

- Where *X* is:
  - Actors, threads, transactional memory, futures, love...
- In Haskell, the approach is to give you lots of different *X*s

"Embrace diversity (but control side effects)"

–Simon Peyton Jones

All you need is *X*

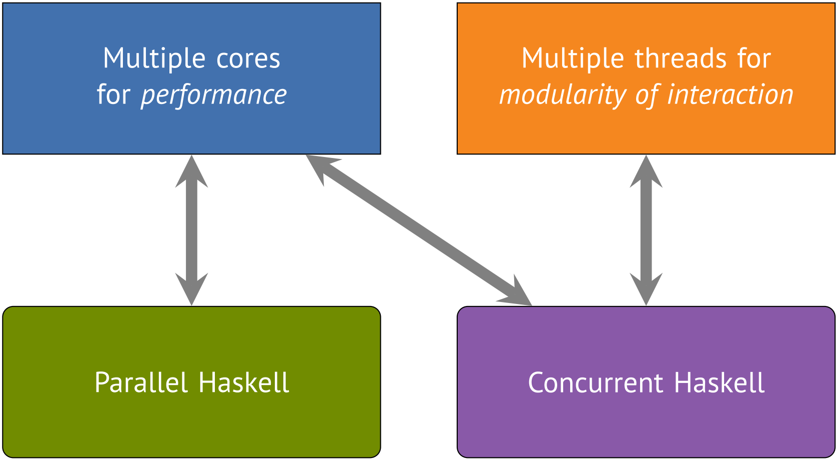
- Where *X* is:
  - Actors, threads, transactional memory, futures, love…
- In Haskell, the approach is to give you lots of different *X*s

“Embrace diversity (but control side effects)”

–Simon Peyton Jones

What is the difference between parallelism and concurrency?

Parallelism vs. Concurrency



Parallel Haskell

Task parallelism

Independent tasks executed on multiple processors, possibly with very different code and data.

Primary distinguishing feature of Parallel Haskell:

- The program does “the same thing” regardless of how many cores are used (*determinism*)
- No race conditions or deadlocks
  - add parallelism without sacrificing correctness
- Parallelism is used to speed up pure (non-IO monad) Haskell code

Lecture goals

- Understanding the GHC primitives for working with parallelism
- Tools and pitfalls
- Work with the `Par` Monad
- Use the `Par` monad for expressing dataflow parallelism

Running example: Sudoku

- Peter Norvig's algorithm with constraint propagation. Code from the Haskell wiki
- can solve all 49,000$^{(*)}$ problems in 2 mins

  $^{(*)}$ collected by Gordon Royle University of Western Australia
- input: a line of text representing a problem

					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

4						8		5
	3							
			7					
	2						6	
				8		4		
				1				
			6		3		7	
5			2					
1		4						

```
..............3.85..1.2.......5.7.....4...1...9.......5......73..2.1........4...9
4.....8.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......
```

Solving Sudoku problems

```
import Sudoku (solve, Grid)
import System.Environment

type Grid = ...
solve :: String -> Maybe Grid
evaluate :: a -> IO a

main :: IO ()
main = do
    [f] <- getArgs
    file <- readFile f
    let puzzles   = lines file
        solutions = map solve puzzles

    print (length (filter isJust solutions))
```

Compile and run the program

```
$ ghc -O2 sudoku1.hs -rtsopts
```

Compile and run the program

```
$ ghc -O2 sudoku1.hs -rtsopts
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku1.hs, sudoku1.o )
Linking sudoku1 ...
$
```

Compile and run the program

```
$ ghc -O2 sudoku1.hs -rtsopts
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku1.hs, sudoku1.o )
Linking sudoku1 ...
$ ./sudoku1 sudoku17.1000.txt +RTS -s
```

Compile and run the program

```
$ ghc -O2 sudoku1.hs -rtsopts
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku1.hs, sudoku1.o )
Linking sudoku1 ...
$ ./sudoku1 sudoku17.1000.txt +RTS -s
   2,352,239,176 bytes allocated in the heap
      39,000,728 bytes copied during GC
         213,616 bytes maximum residency (13 sample(s))
          81,576 bytes maximum slop
               2 MB total memory in use (0 MB lost due to fragmentatio

                                    Tot time (elapsed)
  Gen  0      4552 colls,     0 par    0.09s    0.09s
  Gen  1        13 colls,     0 par    0.00s    0.00s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    1.99s  (  2.01s elapsed)
  GC      time    0.09s  (  0.10s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    2.09s  (  2.11s elapsed) # <- what we care about
```

Let's use some cores

- Doing parallel computation entails specifying coordination in some way – compute *A* in parallel with *B*
- This is a constraint on evaluation order
- But by design, Haskell *does not have a specified evaluation order*
- So we need to add something to the language to express constraints on evaluation order

The Eval monad

```haskell
import Control.Parallel.Strategies

data Eval a
instance Monad Eval

runEval :: Eval a -> a
rpar :: a -> Eval a
rseq :: a -> Eval a
```

[1] *Seq no more: Better Strategies for Parallel Haskell*

The `Eval` monad

```
import Control.Parallel.Strategies

data Eval a
instance Monad Eval

runEval :: Eval a -> a
rpar :: a -> Eval a
rseq :: a -> Eval a
```

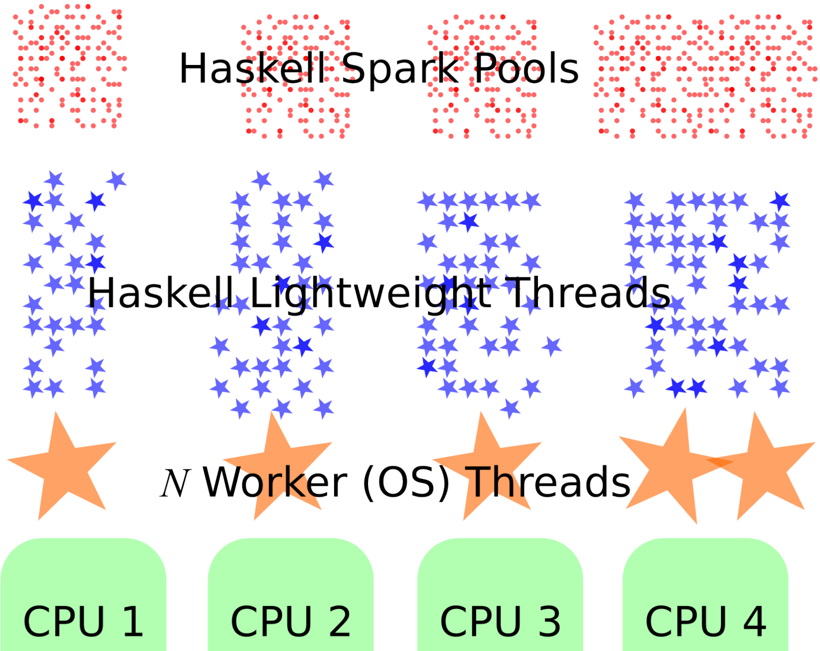
- `Eval` is pure
- Just for expressing sequencing between `rpar`/`rseq`
- Compositional – larger `Eval` sequences can be built by composing smaller ones using monad combinators
- Internal workings of `Eval` are simple[1]

[1] *Seq no more: Better Strategies for Parallel Haskell*

What does `rpar` actually do?

- `x <- rpar e`
- `rpar` creates a *spark* by writing an entry in the *spark pool*
  - `rpar` is cheap (not a OS thread)
- The spark pool is a circular buffer
- When a processor has nothing to do, it tries to remove an entry from its own spark pool, or steal an entry from another spark pool (*work stealing*)
- When a spark is found, it is evaluated
- If the value corresponding to a spark is needed before it is computed, it *fizzles*.
- The spark pool can be full — watch out for spark overflow

Illustration by Don Stewart



Basic Eval patterns

- To compute a in parallel with b, and return a pair of the results:

```
do a’ <- rpar a
   b’ <- rseq b
   rseq a’
   return (a’, b’)
```

Basic Eval patterns

- To compute a in parallel with b, and return a pair of the results:

```
do a' <- rpar a
   b' <- rseq b
   rseq a'
   return (a', b')
```

- Alternatively:

```
do a' <- rpar a
   b' <- rseq b
   return (a', b')
```

Basic Eval patterns

- To compute a in parallel with b, and return a pair of the results:

```
do a' <- rpar a
   b' <- rseq b
   rseq a'
   return (a', b')
```

- Alternatively:

```
do a' <- rpar a
   b' <- rseq b
   return (a', b')
```

- What is the difference between the two?

Back to Sudoku

Let's divide the work in two, so we can solve each half in parallel:

```
main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f
  let puzzles =
        lines file
      (as,bs) =
        splitAt (length puzzles `div` 2) puzzles
      solutions = runEval $ do
        as' <- rpar (map solve as)
        bs' <- rpar (map solve bs)
        return (as' ++ bs')
  print (length (filter isJust solutions))
```

Back to Sudoku

Let's divide the work in two, so we can solve each half in parallel:

```
main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f
  let puzzles =
        lines file
      (as,bs) =
        splitAt (length puzzles `div` 2) puzzles
      solutions = runEval $ do
        as' <- rpar (map solve as)
        bs' <- rpar (map solve bs)
        return (as' ++ bs')
  print (length (filter isJust solutions))
```

Does not work!

rpar evaluates its argument to Weak Head Normal Form (WHNF)

Refresher: Lazy Evaluation

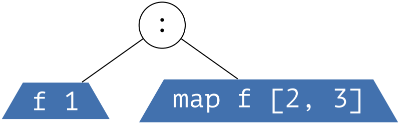
- Evaluating an expression creates a *thunk*

`map f [1,2,3]`

Refresher: Lazy Evaluation

- Evaluating an expression creates a *thunk*

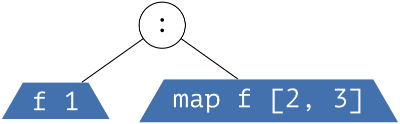
- Weak head normal form (WHNF) evaluates to first constructor



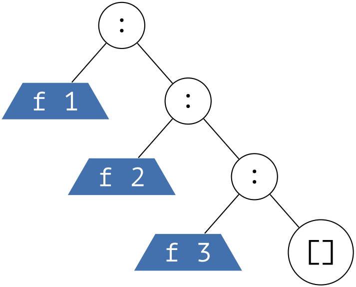
Refresher: Lazy Evaluation

- Evaluating an expression creates a *thunk*

- Weak head normal form (WHNF) evaluates to first constructor



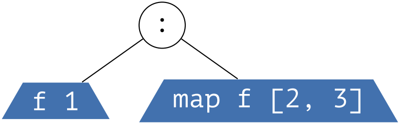
- Spine evaluated



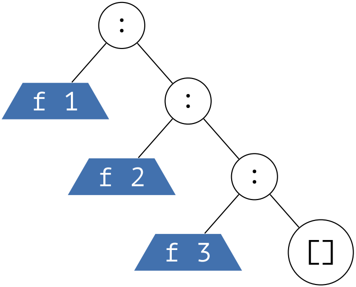
Refresher: Lazy Evaluation

- Evaluating an expression creates a *thunk*

- Weak head normal form (WHNF) evaluates to first constructor



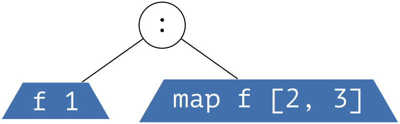
- Spine evaluated



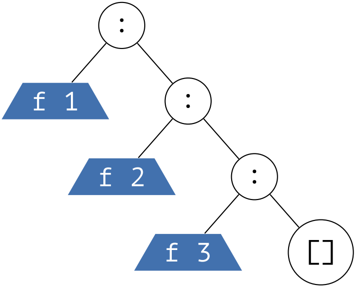
Refresher: Lazy Evaluation

- Evaluating an expression creates a *thunk*

- Weak head normal form (WHNF) evaluates to first constructor



- Spine evaluated



- **Stuff like this is why Haskell is not an ideal parallel language.**

We need to go deeper

```
import Control.DeepSeq

class NFData a where
  rnf :: a -> ()

deepseq a b = rnf a `seq` b

force :: NFData a => a -> a
force a = deepseq a a
```

- `force` fully evaluates a nested data structure and returns it
  - For example a list: the list is fully evaluated, including the elements. Example from library:

    ```
    instance NFData a => NFData [a] where
      rnf [] = ()
      rnf (x:xs) = x `deepseq` xs `deepseq` ()
    ```

Apply force

```
main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f
  let puzzles =
        lines file
      (as,bs) =
        splitAt (length puzzles `div` 2) puzzles
      solutions = runEval $ do
        as' <- rpar (force (map solve as))
        bs' <- rpar (force (map solve bs))
        return (as' ++ bs')
  print (length (filter isJust solutions))
```

Apply force

```
main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f
  let puzzles =
         lines file
      (as,bs) =
         splitAt (length puzzles `div` 2) puzzles
      solutions = runEval $ do
        as' <- rpar (force (map solve as))
        bs' <- rpar (force (map solve bs))
        return (as' ++ bs')
  print (length (filter isJust solutions))
```

Compile and run the program

```
$ ghc -O2 sudoku2.hs -rtsopts -threaded
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku2.hs, sudoku2.o )
Linking sudoku2 ...
$ ./sudoku2 sudoku17.1000.txt +RTS -s -N2
   2,383,492,264 bytes allocated in the heap
      49,765,200 bytes copied during GC
       2,489,872 bytes maximum residency (8 sample(s))
         259,184 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentatio

                                    Tot time (elapsed)
  Gen  0      2979 colls,  2978 par    0.12s    0.09s
  Gen  1         8 colls,     8 par    0.02s    0.01s

  SPARKS: 2 (1 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    2.36s  (  1.43s elapsed)
  GC      time    0.14s  (  0.10s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    2.50s  (  1.53s elapsed)
```

Calculating speedup

- Calculating speedup with 2 processors:
  - Elapsed time (1 proc) / Elapsed Time (2 procs)
  - **NOT** CPU time (2 procs) / Elapsed (2 procs)
  - Compare against sequential program, not parallel program running on 1 CPU (why?)
- Speedup for `sudoku2`: $2.11/1.53 = 1.38$

Why not 2?

There are two main reasons for lack of parallel speedup

1. Less than 100% *utilisation* (some processors idle for part of the time)
2. Extra *overhead* in the parallel version

Each of these has many possible causes.

A menu of ways to screw up

1. Less than 100% utilisation
   - Parallelism was not created, or was discarded
   - Algorithm not fully parallelised residual sequential computation
   - Uneven workloads
   - Poor scheduling
   - Communication latency
2. Extra overhead in the parallel version
   - Overheads from rpar, work-stealing, deep, ...
   - Lack of locality, cache effects...
   - Larger memory requirements leads to GC overhead
   - GC synchronisation
   - Duplicating work (but this might increase utilisation!)

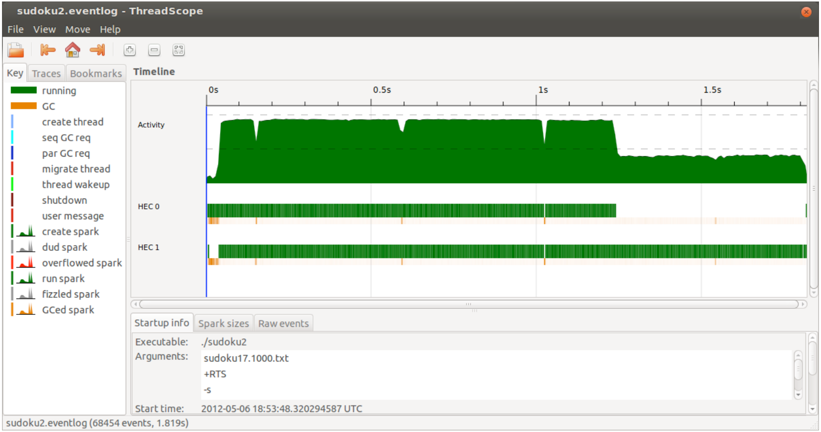
So we need tools

- We need tools to tell us why the program isn't performing as well as it could be
- For Parallel Haskell we have ThreadScope

So we need tools

- We need tools to tell us why the program isn't performing as well as it could be
- For Parallel Haskell we have ThreadScope

```
$ ghc -O2 sudoku2.hs -rtsopts -threaded -eventlog
$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -ls
$ threadscope sudoku2.eventlog
```



sudoku2.eventlog - ThreadScope
File
View
Move
Help
Key
Traces
Bookmarks
running
GC
create thread
seq GC req
par GC req
migrate thread
thread wakeup
shutdown
user message
create spark
dud spark
overflowed spark
run spark
fizzled spark
GCed spark
Timeline
0s
0.5s
1s
1.5s
Activity
HEC 0
HEC 1
Startup info
Spark sizes
Raw events
Executable:
./sudoku2
Arguments:
sudoku17.1000.txt
+RTS
-s
Start time:
2012-05-06 18:53:48.320294587 UTC
sudoku2.eventlog (68454 events, 1.819s)

Uneven workloads

- One of the tasks took longer than the other, leading to less than 100% utilisation

```
let (as, bs) =
      splitAt (length puzzles `div` 2)
              puzzles
```

- One of these lists contains more work than the other, even though they have the same length

Partitioning

```
let (as, bs) =
      splitAt (length puzzles `div` 2)
              puzzles
```

- Dividing up the work along fixed pre-defined boundaries like this example, is called *static partitioning*
- Static partitioning is simple, but can lead to under-utilisation if the tasks can vary in size
- Static partitioning does not adapt to varying availability of processors
- Our solution in this example can use only 2 processors

Dynamic Partitioning

- *Dynamic partitioning* involves
  - dividing the work into smaller units
  - assigning work units to processors dynamically at runtime using a scheduler
- Good for irregular problems and varying number of processors
- GHC's runtime system provides spark pools to track the work units, and a work-stealing scheduler to assign them to processors
- So all we need to do is use smaller tasks and more rpars, and we get dynamic partitioning

Back to Sudoku

```
runEval $ do
  as' <- rpar (force (map solve as))
  bs' <- rpar (force (map solve bs))
```

We want to push `rpar` down into the `map`, so that each call to solve will be a separate spark

A parallel map

```
parMap :: (a -> b) -> [a] -> Eval [b]
parMap f [] = return []
parMap f (a:as) = do
  b <- rpar (f a)
  bs <- parMap f as
  return (b:bs)
```

(Already provided in `Control.Parallel.Strategies`)

Putting it together

```
main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f

  let puzzles   = lines file
      solutions = runEval (parMap solve puzzles)

  print (length (filter isJust solutions))
```

Result with 2 cores

```
$ ./sudoku3 sudoku17.1000.txt +RTS -N2 -s
   2,386,461,968 bytes allocated in the heap
      68,963,504 bytes copied during GC
       2,717,872 bytes maximum residency (14 sample(s))
         179,824 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentatio

                                    Tot time (elapsed)
  Gen  0      2511 colls,  2510 par    0.14s    0.10s
  Gen  1        14 colls,    14 par    0.05s    0.03s

  Parallel GC work balance: 1.69 (8603353 / 5095344, ideal 2)

  SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    2.46s  (  1.24s elapsed)
  GC      time    0.19s  (  0.12s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    2.65s  (  1.37s Elapsed)
```

Result with 2 cores

```
$ ./sudoku3 sudoku17.1000.txt +RTS -N2 -s
   2,386,461,968 bytes allocated in the heap
      68,963,504 bytes copied during GC
       2,717,872 bytes maximum residency (14 sample(s))
         179,824 bytes maximum slop
               9 MB total memory in use (0 MB lost due to fragmentatio

                                    Tot time (elapsed)
  Gen  0      2511 colls,  2510 par    0.14s    0.10s
  Gen  1        14 colls,    14 par    0.05s    0.03s

  Parallel GC work balance: 1.69 (8603353 / 5095344, ideal 2)

  SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    2.46s  (  1.24s elapsed)
  GC      time    0.19s  (  0.12s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    2.65s  (  1.37s Elapsed)
```

Speedup 1.54

Evaluation Strategies

- So far we have used `Eval/rpar/rseq`
  - These are quite low-level tools
  - But its important to understand how the underlying mechanisms work
- Now: raise the level of abstraction
- Goal: encapsulate parallel idioms as re-usable components that can be composed together.

Two High-level Approaches

- Composing `Strategy`'s in the `Eval` monad, and exploit modularity via lazy evaluation (see the book)
- Use the `Par` monad to make parallel computations explicit (the rest of the lecture)

The `Par` monad

- Abandon modularity via lazy evaluation
- Get a more direct programming model
- Avoid some common pitfalls
- Modularity via higher-order skeletons
- A beautiful implementation

The Par monad

```
import  Control.Monad.Par

data Par
instance Monad Par
runPar :: Par a -> a
fork :: Par () -> Par ()

data IVar
new :: Par (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> a -> Par ()
```

The Par monad

```
import  Control.Monad.Par

data Par
instance Monad Par
runPar :: Par a -> a
fork :: Par () -> Par ()

data IVar
new :: Par (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> a -> Par ()
```

DANGER!
Must not contain an IVar

Sudoku is addictive

Let's use the Par monad to solve some sudoku puzzles in parallel

```
main :: IO ()
main = do
  [f] <- getArgs
  grids <- fmap lines (readFile f)
  let (as,bs) = splitAt (length grids `div` 2)
                        grids
      solutions = runPar $ do
                    ...
                    return (as' ++ bs')
  print (length (filter isJust solutions))
```

Building new abstractions

- `Par` primitives can be used for building new abstractions.
- For instance, spawning a computation and make a new `IVar` for the result is often useful (aka futures).

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do r <- new
             fork $ p >>= put r
             return r
```

Divide and conquer parallelism

```
parfib :: Int -> Int -> Par Int
parfib n
  | n <= 2 = return 1
  | otherwise = do
    x <- spawn $ parfib (n-1)
    y <- spawn $ parfib (n-2)
    x' <- get x
    y' <- get y
    return $ x' + y'
```

Another abstraction

- We can use `spawn` to build other primitives for regular parallelism, like `parMap`:

Another abstraction

- We can use spawn to build other primitives for regular parallelism, like `parMap`:

-
```
parMapM :: NFData b
        => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
  ibs <- mapM (spawn . f) as
  mapM get ibs
```

Another abstraction

- We can use `spawn` to build other primitives for regular parallelism, like `parMap`:

- ```
  parMapM :: NFData b
          => (a -> Par b) -> [a] -> Par [b]
  parMapM f as = do
    ibs <- mapM (spawn . f) as
    mapM get ibs
  ```

- Similar to `parMap`, except that the type is subtly different
  ```

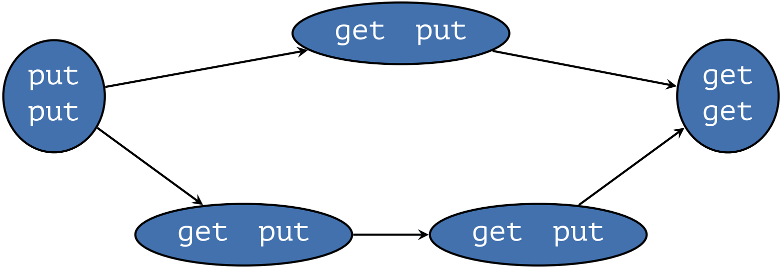
How did we avoid laziness?

- `put` is *hyperstrict*.
- (By default)
- There's also a WHNF version called `put_`
- **Why might we want to use `put_`?**

Dataflow problems

- `Par` really shines when the problem is easily expressed as a dataflow graph, particularly an irregular or dynamic graph (e.g., shape depends on the program input)
- Identify the nodes and edges of the graph
  - Each node is created by `fork`
  - Each edge is an `IVar`
- Many problems that are awkward in e.g. Futhark are straightforward with `Par`

`Par` expresses dynamic dataflow



Thoughts to take away

- Parallelism is not the goal
- Making your program *faster* is the goal (unlike Concurrency, which is a goal in itself)
- If you can make your program fast enough without parallelism, all well and good

Course stuff

- On Wednesday: Work with parallelism in Haskell
- Assignment:
  - Speeding up a ray tracer by computing pixels in parallel
  - Speeding up a ray tracer by computing *BVH* in parallel
  - Maybe other sources of parallelism, if you can find them!