Parallel Functional Programming

Lecture 2

Mary Sheeran
(with thanks to Simon Marlow for use of slides)

http://www.cse.chalmers.se/edu/course/pfp
Course reps

Could I have some volunteers (from Chalmers, GU) ?

(Seems better than using randomly generated names)
Remember nfib

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

- A trivial function that returns the number of calls made—and makes a very large number!

<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>
</tbody>
</table>
Sequential

sfib 40
Explicit Parallelism

`par x y`

- "Spark" x in parallel with computing y
  - (and return y)
- The run-time system *may* convert a spark into a parallel task—or it may not
- Starting a task is cheap, but not free
Explicit Parallelism

x `par` y
Explicit sequencing

\[ \text{pseq } x \; y \]

• Evaluate \( x \) before \( y \) (and return \( y \))

• Used to ensure we get the right evaluation order
Explicit sequencing

```
x `pseq` y
```

• Binds more tightly than par
import Control.Parallel

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

import Control.Parallel

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

• Evaluate nf1 in parallel with (Evaluate nf2 before ... )
Looks promising
Looks promising
What’s happening?

$ ./NF +RTS -N4 -s

-s to get stats
Hah

331160281
...

SPARKS: 165633686 (105 converted, 0 overflowed, 0 dud, 165098698 GC'd, 534883 fizzled)

<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>MUT</td>
<td>2.31s</td>
<td>1.98s</td>
</tr>
<tr>
<td>GC</td>
<td>7.58s</td>
<td>0.51s</td>
</tr>
<tr>
<td>EXIT</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Total</td>
<td>9.89s</td>
<td>2.49s</td>
</tr>
<tr>
<td>Operation</td>
<td>Time</td>
<td>Elapsed Time</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>--------------</td>
</tr>
<tr>
<td>INIT</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>MUT</td>
<td>2.31s</td>
<td>1.98s</td>
</tr>
<tr>
<td>GC</td>
<td>7.58s</td>
<td>0.51s</td>
</tr>
<tr>
<td>EXIT</td>
<td>0.00s</td>
<td>0.00s</td>
</tr>
<tr>
<td>Total</td>
<td>9.89s</td>
<td>2.49s</td>
</tr>
</tbody>
</table>

SPARKS: 165633686 (105 converted, 0 overflowed, 0 dud, 165098698 GC'd, 534883 fizzled)

converted = turned into useful parallelism
Controlling Granularity

- Let’s use a threshold for going sequential, \( t \)

```haskell
tfib :: Integer -> Integer -> Integer
tfib t n | n < t = sfib n
          tfib t n = nf1 `par` nf2 `pseq` nf1 + nf2 + 1
            where nf1 = tfib t (n-1)
                nf2 = tfib t (n-2)
```
Better

tfib 32 40 gives

SPARKS: 88 (13 converted, 0 overflowed, 0 dud, 0 GC'd, 75 fizzled)

INIT time 0.00s (0.01s elapsed)
MUT time 2.42s (1.36s elapsed)
GC time 3.04s (0.04s elapsed)
EXIT time 0.00s (0.00s elapsed)
Total time 5.47s (1.41s elapsed)
What are we controlling?

The division of the work into possible parallel tasks (par) including choosing size of tasks

GHC runtime takes care of choosing which sparks to actually evaluate in parallel and of distribution

Need also to control order of evaluation (pseq) and degree of evaluation

Dynamic behaviour is the term used for how a pure function gets partitioned, distributed and run

Remember, this is deterministic parallelism. The answer is always the same!
positive so far (par and pseq)

Don’t need to

express communication

express synchronisation

deal with threads explicitly
BUT

Original code + par + pseq + rnf etc. can be opaque
Separate concerns

Algorithm
Separate concerns

Algorithm

Evaluation Strategy
Evaluation Strategies

express dynamic behaviour independent of the algorithm

provide abstractions above par and pseq

are modular and compositional
(they are ordinary higher order functions)

can capture patterns of parallelism
Papers

Algorithm + Strategy = Parallelism

P.W. TRINDER
Department of Computing Science, University of Glasgow, Glasgow, UK

K. HAMMOND
Division of Computing Science, University of St Andrews, St Andrews, UK

H.-W. LOIDL AND S.I. PEYTON JONES †
Department of Computing Science, University of Glasgow, Glasgow, UK

Seq no more: Better Strategies for Parallel Haskell

Simon Marlow
Microsoft Research, Cambridge, UK
simonmar@microsoft.com

Patrick Maier
Heriot-Watt University, Edinburgh, UK
P.Maier@hw.ac.uk

Hans-Wolfgang Loidl
Heriot-Watt University, Edinburgh, UK
H.W.Loidl@hw.ac.uk

Mustafa K. Aswad
Heriot-Watt University, Edinburgh, UK
mka10@hw.ac.uk

Phil Trinder
Heriot-Watt University, Edinburgh, UK
P.W.Trinder@hw.ac.uk

JFP 1998

Haskell’10
Papers

Redesigns strategies

richer set of parallelism combinators
Better specs (evaluation order)
Allows new forms of coordination
generic regular strategies over data structures
speculative parallelism
monads everywhere 😊

Presentation is about New Strategies
The Eval monad

- Eval is pure
- Just for expressing sequencing between rpar/rseq – nothing more
- Compositional – larger Eval sequences can be built by composing smaller ones using monad combinators
- Internal workings of Eval are very simple (see Haskell Symposium 2010 paper)

```haskell
import Control.Monad

data Eval a
instance Monad Eval

runEval :: Eval a -> a
rpar :: a -> Eval a
rseq :: a -> Eval a
```
What does rpar *actually* do?

- rpar creates a *spark* by writing an entry in the *spark pool*—rpar is very cheap! (not a 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
- The spark pool can be full—watch out for spark overflow!

Slide borrowed from Simon Marlow’s CEFP slides, with thanks
Expressing evaluation order

\[
qfib :: \text{Integer} \rightarrow \text{Integer}
\]
\[
qfib \ n \mid n < 2 = 1
\]
\[
qfib \ n = \text{runEval $ do}$
\]
\[
\qquad \text{nf1 <- rpar (qfib (n-1))}
\]
\[
\qquad \text{nf2 <- rseq (qfib (n-2))}
\]
\[
\qquad \text{return} \ (\text{nf1 + nf2 + 1})
\]
Expressing evaluation order

qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
  nf1 <- rpar (qfib (n-1))
  nf2 <- rseq (qfib (n-2))
  return (nf1 + nf2 + 1)

"My argument could be evaluated in parallel"
Expressing evaluation order

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

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

\[
qfib \ n = \text{runEval} \ do
\]

\[
nf1 \leftarrow \text{rpar} \ (qfib \ (n-1))
\]

\[
nf2 \leftarrow \text{rseq} \ (qfib \ (n-2))
\]

\[
\text{return} \ (nf1 + nf2 + 1)
\]

"My argument could be evaluated in parallel"

Remember that the argument should be a thunk!
Expressing evaluation order

qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
  nf1 <- rpar (qfib (n-1))
  nf2 <- rseq (qfib (n-2))
  return (nf1 + nf2 + 1)

"Evaluate my argument and wait for the result."
Expressing evaluation order

qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
  nf1 <- rpar (qfib (n-1))
  nf2 <- rseq (qfib (n-2))
  return (nf1 + nf2 + 1)
Expressing evaluation order

```haskell
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
  nf1 <- rpar (qfib (n-1))
  nf2 <- rseq (qfib (n-2))
  return (nf1 + nf2 + 1)
```

pull the answer out of the monad
runEval $ do
  a <- rpar (f x)
  b <- rpar (f y)
  return (a, b)
runEval $ do
  a <- rpar (f x)
  b <- rpar (f y)
  return (a, b)
runEval $ do
  a <- rpar (f x)
  b <- rseq (f y)
  return (a,b)
runEval $ do
  a <- rpar (f x)
  b <- rseq (f y)
  return (a, b)

Not completely satisfactory
Unlikely to know which one to wait for
runEval \$ \text{do} \ vendorimageend{runEval$ do  
\text{a \leftarrow rpar}\ (f\ \text{x})  
\text{b \leftarrow rseq}\ (f\ \text{y})  
rseq\ a  
\text{return}\ (a, b)
runEval $ do
  a <- rpar (f x)
  b <- rseq (f y)
  rseq a
  return (a, b)

Choice between rpar/rpar and rpar/rseq/rseq will depend on circumstances (see PCPH ch. 2)
What do we have?

The Eval monad raises the level of abstraction for pseq and par; it makes fragments of evaluation order first class, and lets us compose them together. We should think of the Eval monad as an Embedded Domain-Specific Language (EDSL) for expressing evaluation order, embedding a little evaluation-order constrained language inside Haskell, which does not have a strongly-defined evaluation order.

(from Haskell 10 paper)
parallel map

\[ pMap :: (a \rightarrow b) \rightarrow [a] \rightarrow \text{Eval} [b] \]
\[ pMap \ f \ [] = \text{return} \ [] \]
\[ pMap \ f \ (a:as) = \text{do} \]
\[ \quad b \leftarrow \text{rpar} \ (f \ a) \]
\[ \quad bs \leftarrow pMap \ f \ as \]
\[ \text{return} \ (b:bs) \]
Using our pMap

print $ sum $ runEval $ (parMap foo (reverse [1..10000]))

foo :: Integer -> Integer
foo = \a -> sum [1 .. a]

SPARKS: 10000 (8194 converted, 1806 overflowed, 0 dud, 0 GC'd, 0 fizzled)
Using our pMap

```haskell
print \$ sum \$ runEval \$ (parMap foo (reverse [1..10000]))
```

foo :: Integer -> Integer
foo = \a -> sum [1 .. a]

SPARKS: 10000 (8194 converted, 1806 overflowed, 0 dud, 0 GC'd, 0 fizzled)

#sparks = length of list
parallel map

+ Captures a pattern of parallelism
+ good to do this for standard higher order function like map
+ can easily do this for other standard sequential patterns

return (b:bs)
BUT

- had to write a new version of map
- mixes algorithm and dynamic behaviour

return (b:bs)
Evaluation Strategies

Raise level of abstraction

Encapsulate parallel programming idioms as reusable components that can be composed
Strategy (as of 2010)

type Strategy a = a -> Eval a

function

evaluates its input to some degree

traverses its argument and uses rpar and rseq to express dynamic behaviour / sparking

returns an equivalent value in the Eval monad
using :: a -> Strategy a -> a

x `using` strat = runEval (strat x)

Program typically applies the strategy to a structure and then uses the returned value, discarding the original one (which is why the value had better be equivalent).

An almost identity function that does some evaluation and expresses how that can be parallelised.
Basic strategies

r0 :: Strategy a
r0 x = return x

rpar :: Strategy a
rpar x = x `par` return x

rseq :: Strategy a
rseq x = x `pseq` return x

rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x
Basic strategies

\[ r_0 :: \text{Strategy } a \]
\[ r_0 \ x = \text{return } x \]

\[ r_{\text{par}} :: \text{Strategy } a \]
\[ r_{\text{par}} \ x = x \ `\text{par}` \ \text{return } x \]

\[ r_{\text{seq}} :: \text{Strategy } a \]
\[ r_{\text{seq}} \ x = x \ `\text{pseq}` \ \text{return } x \]

\[ r_{\text{deepseq}} :: \text{NFData } a \Rightarrow \text{Strategy } a \]
\[ r_{\text{deepseq}} \ x = \text{rnf } x \ `\text{pseq}` \ \text{return } x \]
Basic strategies

r0 :: Strategy a
r0 x = return x

rpar :: Strategy a
rpar x = x `par` return x

rseq :: Strategy a
rseq x = x `pseq` return x

rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x

spark x
Basic strategies

\[
\begin{align*}
\text{r0} & : \text{Strategy } a \\
\text{r0 } x & = \text{return } x \\
\text{rpar} & : \text{Strategy } a \\
\text{rpar } x & = x \ `\text{par}` \ \text{return } x \\
\text{rseq} & : \text{Strategy } a \\
\text{rseq } x & = x \ `\text{pseq}` \ \text{return } x \\
\text{rdeepseq} & : \text{NFData } a \Rightarrow \text{Strategy } a \\
\text{rdeepseq } x & = \text{rnf } x \ `\text{pseq}` \ \text{return } x
\end{align*}
\]
Basic strategies

r0 :: Strategy a
r0 x = return x

rpar :: Strategy a
rpar x = x `par` return x

rseq :: Strategy a
rseq x = x `pseq` return x

rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x
evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x
                    xs' <- evalList s xs
                    return (x':xs')
**evalList**

```haskell
evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x
                      xs' <- evalList s xs
                      return (x':xs')
```

Takes a Strategy on `a` and returns a Strategy on lists of `a`. Building strategies from smaller ones.
parList

evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x
                    xs' <- evalList s xs
                    return (x':xs')

parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
parList

evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x
                              xs' <- evalList s xs
                              return (x':xs')

parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)

dot :: Strategy a -> Strategy a -> Strategy a
s2 `dot` s1 = s2 . runEval . s1
In reality

evalList :: Strategy a -> Strategy [a]
evalList = evalTraversable

parList :: Strategy a -> Strategy [a]
parList = parTraversable
evalList :: Strategy a -> Strategy [a]
evalList = evalTraversable

parList :: Strategy a -> Strategy [a]
parList = parTraversable

The equivalent of evalList and of parList are available for many data structures (Traversable). So defining parX for many X is really easy

=> generic strategies for data-oriented parallelism
another list strategy

parListSplitAt :: Int -> Strategy [a] -> Strategy [a] -> Strategy [a]

parListSplitAt n stratL stratR

n
par

stratL stratR
How do we use a Strategy?

```haskell
type Strategy a = a -> Eval a
```

- We could just use `runEval`
- But this is better:
  ```haskell
  x `using` s = runEval (s x)
  ```

  - e.g.
    ```haskell
    myList `using` parList rdeepseq
    ```

- Why better? Because we have a "law":
  - `x `using` s ≈ x`
  - We can insert or delete "`using` s" without changing the semantics of the program
Is that really true?

• Well, not entirely.

1. It relies on Strategies returning “the same value” (identity-safety)
   – Strategies from the library obey this property
   – Be careful when writing your own Strategies
2. `x `using` s` might do more evaluation than just `x`.
   – So the program with `x `using` s` might be `_|_`, but the program with just `x` might have a value

• If identity-safety holds, adding `using` cannot make the program produce a different result (other than `_|_`)
using yet another list strategy

\[ \text{parListChunk} :: \text{Int} \rightarrow \text{Strategy a} \rightarrow \text{Strategy [a]} \]

\[ \text{parListChunk n strat} \]

\[ \text{evalList strat} \]
using yet another list strategy

```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

Before

```
print $ sum $ runEval $ parMap foo (reverse [1..10000])
```

Now

```
print $ sum $ 
(map foo (reverse [1..10000]) `using` parListChunk 50 rdeepseq)
```

SPARKS: 200 (200 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
using yet another list strategy

```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

Before

```
print $ sum $(
  map foo (reverse [1..10000])
)
```

Now

```
print $ sum $(
  map foo (reverse [1..10000]) `using` parListChunk 50 rdeepseq
)
```

Remember not to be a control freak, though. Generating plenty of sparks gives the runtime the freedom it needs to make good choices (= Dynamic partitioning for free)

SPARKS: 200 (200 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
using is not always what we need

• Trying to pull apart algorithm and coordination in qfib (from earlier) doesn’t really give a satisfactory answer (see Haskell 10 paper)

(If the worst comes to the worst, one can get explicit control of threads etc. in concurrent Haskell, but determinism is lost... )
Divide and conquer

Capturing patterns of parallel computation is a major strong point of strategies. D&C is a typical example (see also parBuffer, parallel pipelines etc.)

```
divConq :: (a -> b)
  -> a
  -> (a -> Bool)
  -> (b -> b -> b)
  -> (a -> Maybe (a,a))
  -> b
```

- function on base cases
- input
- par threshold reached?
- combine
- divide
- result
Divide and Conquer

```haskell
divConq f arg threshold combine divide = go arg
  where
    go arg =
      case divide arg of
        Nothing -> f arg
        Just (l0,r0) -> combine l1 r1 'using' strat
          where
            l1 = go l0
            r1 = go r0
            strat x = do r l1; r r1; return x
              where r | threshold arg = rseq
                       | otherwise = rpar
```

Separates algorithm and strategy
A first inkling that one can probably do interesting things by programming with strategies
Skeletons

• encode fixed set of common coordination patterns and provide efficient parallel implementations (Cole, 1989)

• Popular in both functional and non-functional languages. See particularly Eden (Loogen et al, 2005)

A difference: one can / should roll ones own strategies
Strategies: summary

+ elegant redesign by Marlow et al (Haskell 10)
+ better separation of concerns
+ Laziness is essential for modularity
+ generic strategies for (Traversable) data structures
+ Marlow’s book contain a nice `kmeans` example. Read it!
- Having to think so much about evaluation order is worrying! Laziness is not only good here. *(Cue the Par Monad Lecture!)*
Strategies: summary

Algorithm

Evaluation Strategy
Better visualisation
Better visualisation
Better visualisation
Simon Marlow’s landscape for parallel Haskell

- **Parallel**
  - par/pseq
  - Strategies
  - Par Monad
  - Repa
  - Accelerate
  - DPH

- **Concurrent**
  - forkIO
  - MVar
  - STM
  - async
  - Cloud Haskell
In the meantime

• Do exercise 1 (not graded)
• Read papers and PCPH
• Start on Lab A (due midnight April 6)
• Note Nick’s office hours
  (room 5461, wed 13-14 and fri 13-14)

Use him! He is your best resource.