The Par Monad: Dataflow Parallelism

Stephen A. Edwards

Columbia University

Fall 2019
The Par Monad
    spawn and spawnP
    parMap and parMapM

Example: Shortest Paths in a Graph
    Parallelizing Floyd-Warshall

Bibliography
The Par Monad: Dataflow Parallelism

In Control.Monad.Par,

```haskell
newtype Par a = ...
instance Applicative Par ...
instance Monad Par ...

runPar :: Par a -> a
fork :: Par () -> Par ()

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

`put` forces evaluation of its argument (NFData)

```haskell
runPar $ do
  −− parmonad.hs
  i <- new          −− Create IVar
  j <- new
  fork (put i (fib n))  −− Write result
  fork (put j (fib m))
  a <- get i        −− Wait for result
  b <- get j
  return (a+b)
```

An IVar is a write-once variable

`get` waits for data to be `put`

Multiple `puts` to the same IVar cause a runtime error

Restrict each IVar to a single `Par`
The Par Monad: Dataflow Parallelism

```haskell
runPar $ do
  i <- new  -- Create IVar
  j <- new
  fork (put i (fib n))  -- Write result
  fork (put j (fib m))
  a <- get i  -- Wait for result
  b <- get j
  return (a+b)
```

An IVar is a write-once variable

get waits for data to be put

Multiple puts to the same IVar cause a runtime error

Restrict each IVar to a single Par

Marlow, Fig. 4-1
Running Marlow’s parmonad.hs Example

$ stack ghc -- -O2 -threaded -rtsopts -eventlog parmonad.hs
$ ./parmonad 34 35 +RTS -N2 -ls
24157817

Works OK with \(-N2\); obvious load-balancing problem. \(-N8\) slows it down
Control.Monad.Par.{spawn,spawnP}: Fork and Return New IVar

\[
\text{spawn} :: \text{NFData} \ a \Rightarrow \text{Par} \ a \to \text{Par} \ (\text{IVar} \ a) \quad -- \text{Spawn a process}
\]
\[
\text{spawn} \ p = \text{do} \ i \leftarrow \text{new} \quad -- \text{Create a new IVar } i
\quad \text{fork} \$ \ do \ x \leftarrow p \quad -- \text{Run } p
\quad \quad \text{put} \ i \ x \quad -- \text{Put the result in } i
\quad \text{return} \ i \quad -- \text{Return the IVar } i
\]

\[
\text{spawnP} :: \text{NFData} \ a \Rightarrow a \to \text{Par} \ (\text{IVar} \ a) \quad -- \text{Evaluate pure expression}
\]
\[
\text{spawnP} = \text{spawn} \ . \ \text{return}
\]

\[
\text{runPar} \$ \ do
\quad i \leftarrow \text{spawnP} \ (\text{fib} \ n) \quad -- \text{Start fib } n \text{ in parallel with}
\quad j \leftarrow \text{spawnP} \ (\text{fib} \ m) \quad -- \text{Start fib } m
\quad a \leftarrow \text{get} \ i \quad -- \text{Wait for fib } n \text{ to finish}
\quad b \leftarrow \text{get} \ j \quad -- \text{Wait for fib } m \text{ to finish}
\quad \text{return} \ (a+b)
\]
Control.Monad.Par.{parMapM,parMap} 

parMapM applies a function that works in the monad:

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
  ibs <- mapM (spawn . f) as  -- Run each in parallel
  mapM get ibs               -- Wait for all list elements
```

parMap is similar but applies a pure function:

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

Actual implementations in Control.Monad.Par are more general
The Floyd-Warshall Shortest Paths Algorithm

The edge in $g$ from $i$ to $j$ has weight $g_{ij}$

Vertices are numbered $0 \ldots n$

In pseudocode,

```pseudocode
shortestPath :: Graph -> Vertex -> Vertex -> Vertex -> Weight
shortestPath g i j 0 = weight g i j
shortestPath g i j k = min (shortestPath g i j (k-1))
                      (shortestPath g i k (k-1) +
                       shortestPath g k j (k-1))
```

Like Fibonacci, a recursive definition that should be implemented bottom-up with results recorded. $O(n^3)$ overall
Sparse Graph Representation: Maps of Maps

Marlow’s code in fwsparse/SparseGraph.hs

```haskell
import qualified Data.IntMap.Strict as Map

type Vertex = Int

type Weight = Int

type Graph = IntMap (IntMap Weight)

weight :: Graph -> Vertex -> Vertex -> Maybe Weight
weight g i j = do
  jmap <- Map.lookup i g
  Map.lookup j jmap
```

IntMap is tuned to better work with Int keys
**$O(n^3)$ Sequential Implementation**

```haskell
shortestPaths :: [Vertex] -> Graph -> Graph
shortestPaths vs g = foldl' update g vs where -- For each vertex k
    update g k = Map.mapWithKey shortmap g where -- For each vertex i
        shortmap i jmap = foldr shortest Map.empty vs -- Shortest from i
            where
                shortest j m = case (old,new) of -- Update path from i to j via k
                    (Nothing, Nothing) -> m -- No path
                    (Nothing, Just w) -> Map.insert j w m -- Found a new path
                    (Just w, Nothing) -> Map.insert j w m -- Existing path only
                    (Just w1, Just w2) -> Map.insert j (min w1 w2) m -- Best
                        where
                            old = Map.lookup j jmap -- Previous i → j path
                            new = do w1 <- weight g i k -- i → k
                                     w2 <- weight g k j -- k → j
                                     return (w1+w2)
```
Running Sequential Floyd-Warshall

Random graph with 1000 vertices and 800 nodes:

$ stack ghc -- -O2 -rtsopts fwsparse.hs$
$ ./fwsparse 1000 800 +RTS -s$
Total time 14.531s (14.575s elapsed)

Fundamentally, three nested loops:

shortestPaths vs g = foldl' update g vs where
update g k = Map.mapWithKey shortmap g where
shortmap i jmap = foldr shortest Map.empty vs

Two are folds, which are difficult to parallelize unless operation is associative
However, mapWithKey is a map
Parallelizing Floyd-Warshall

*mapWithKey* is an unusual map over an IntMap.

We need a map that runs in the Par monad. Fortunately, IntMap provides

\[
\text{traverseWithKey} :: \text{Applicative } t \Rightarrow \text{ (Key } \rightarrow \text{ a } \rightarrow \text{ t b) } \rightarrow \text{ IntMap a } \rightarrow \text{ t (IntMap b)}
\]

and in Traversable,

\[
\text{traverse} :: (\text{Traversable } t, \text{ Applicative } f) \Rightarrow \text{ (a } \rightarrow \text{ f b) } \rightarrow \text{ t a } \rightarrow \text{ f (t b)}
\]

So we can update our *update* function to spawn *shortmap* in parallel:

```haskell
update g k = runPar $ do
  m <- Map.traverseWithKey
  (\i jmap -> spawnP (shortmap i jmap)) g
  traverse get m -- get each IVar in the IntMap
```
Running Parallel Floyd-Warshall

```
$ stack ghc -- -O2 -threaded -rtsopts -eventlog fwsparse1.hs
$ ./fwsparse1 1000 800 +RTS -s -N1
  Total time  6.091s ( 6.150s elapsed)
$ ./fwsparse1 1000 800 +RTS -s -N8
  Total time  12.071s ( 1.832s elapsed)
```

A 3.4× speedup on 8 cores, but we beat the sequential version (?)

Note the total time increased substantially (parallel overhead), but the elapsed time decreased anyway
Simon Marlow, Ryan Newton, and Simon Peyton Jones.
A monad for deterministic parallelism.