The Par Monad: Dataflow Parallelism

Stephen A. Edwards

Columbia University

Fall 2020
The Par Monad: Dataflow Parallelism

In Control.Monad.Par,

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

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

```haskell
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

```
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

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

**spawn** :: NFData a => Par a -> Par (IVar a) -- Spawn a process

```
spawn p = do i <- new  -- Create a new IVar i
    fork $ do x <- p  -- Run p
              put i x  -- Put the result in i
    return i  -- Return the IVar i
```

**spawnP** :: NFData a -> a -> Par (IVar a) -- Evaluate pure expression

```
spawnP = spawn . return
```

**runPar** $ do

```
i <- spawnP (fib n)  -- Start fib n in parallel with
j <- spawnP (fib m)  -- Start fib m
a <- get i  -- Wait for fib n to finish
b <- get j  -- Wait for fib m to finish
return (a+b)
```
parMapM applies a function that works in the monad:

```haskell
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:

```haskell
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,

```
shortestPath :: Graph -> Vertex -> Vertex -> Vertex -> Weight
shortestPath g i j 0 = weight g i j
shortestPath g i j k = \text{min} \ (shortestPath g i j (k-1))
                        \hspace{1cm} \text{(shortestPath g i k (k-1) +}
                        \hspace{1cm} \text{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

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

\textit{mapWithKey} is an unusual map over an \texttt{IntMap}.

We need a map that runs in the \texttt{Par} monad. Fortunately, \texttt{IntMap} provides

\begin{verbatim}
traverseWithKey :: Applicative t =>
                (Key -> a -> t b) -> IntMap a -> t (IntMap b)
\end{verbatim}

and in \texttt{Traversable},

\begin{verbatim}
traverse :: (Traversable t, Applicative f) =>
           (a -> f b) -> t a -> f (t b)
\end{verbatim}

So we can update our \texttt{update} function to spawn \texttt{shortmap} in parallel:

\begin{verbatim}
update g k = runPar $ do
  m <- Map.traverseWithKey
      (\i jmap -> spawnP (shortmap i jmap)) g
  traverse get m -- get each IVar in the IntMap
\end{verbatim}
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