Monad-Par

Xavier de Bondt Philip de Bruin

November 1, 2018

The Monad-Par

- Based around a monad called Par

The Monad-Par

- Based around a monad called Par
- Run a function parallel by using
  `runPar :: Par a -> a`
- Create parallel tasks with
  `fork :: Par () -> Par ()`

The Monad-Par

- Based around a monad called Par
- Run a function parallel by using
  `runPar :: Par a -> a`
- Create parallel tasks with
  `fork :: Par () -> Par ()`
- Problem: How do we pass data back to the parent?

IVar data type

- `IVar` introduces the following functions:

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

- We can think of `IVar` as an empty box

IVar data type

- IVar introduces the following functions:

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

- We can think of IVar as an empty box
- NOTE: put calls deepseq on the value you try to put in

Example: Fibonacci

- We want to write the function `fib k`
- Assumptions: $n = k - 1$, $m = k - 2$

Example: Fibonacci

- We want to write the function `fib k`
- Assumptions: $n = k - 1$, $m = k - 2$

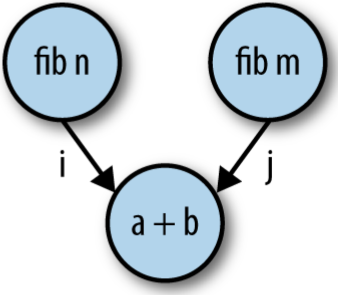


Figure: Dataflow diagram

Example: Fibonacci

Actual code

```
runPar $ do i <- new
            j <- new
            fork (put i (fib n))
            fork (put j (fib m))
            a <- get i
            b <- get j
            return (a + b)
```

Some other functions

spawn

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do i <- new
             fork (do x <- p; put i x)
             return i
```

Some other functions

spawn

```haskell
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do i <- new
             fork (do x <- p; put i x)
             return i
```

So we can write our Fibonacci as

```haskell
runPar $ do i <- (spawn . return . fib) n
            j <- (spawn . return . fib) m
            a <- get i
            b <- get j
            return (a + b)
```

Some other functions

parMap

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

- For this example we go back to Assignment 2 to use Roses!

```
data Rose a = a :> [Rose a]
  deriving (Eq, Show)
```

Examples

Back to Rose!

- For this example we go back to Assignment 2 to use Roses!

```
data Rose a = a :> [Rose a]
  deriving (Eq, Show)
```

- We will make the function `leaves` parallel

```
leaves :: Rose a -> Int
leaves (_ :> []) = 1
leaves (_ :> b ) = sum (map leaves b)
```

Examples

Making leaves parallel

- Our first instinct would say: "Re-use the fibonacci example!"

```
parleaves2 :: Rose a -> Int
parleaves2 (_ :> [a,b])
= runPar $ do i <- new
              j <- new
              fork (put i (leaves a))
              fork (put j (leaves b))
              a <- get i
              b <- get j
              return (a + b)
```

Examples

These parleaves become messy..

- This gets messy very fast.. for let's say 9 children

```
parleaves9 (_ :> [a,b,c,d,e,f,g,h,i])
= runPar $ do j <- new
              ...
              s <- new
              fork (put j (leaves a))
              ...
              fork (put s (leaves i))
              a <- get j
              ...
              i <- get s
              return (a + b + c + d + e + f + g + h + i)
```

Examples

Making leaves parallel with parMap

- So we use `parMap`

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

- So we use parMap

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

- Making this look easy!

```
parleaves' (_ :> b)
    = runPar $ do solutions <- parMap (leaves) b
                  return (sum solutions)
```

- We can also make `mergesort` parallel

```
mergesort :: (Ord a, NFData a) => [a] -> [a]
mergesort xs
  | (length xs) > 1 = merge (mergesort ls) (mergesort rs)
  | otherwise       = xs
 where (ls, rs) = splitinhalf xs
```

- We can also make `mergesort` parallel

```
mergesort :: (Ord a, NFData a) => [a] -> [a]
mergesort xs
  | (length xs) > 1 = merge (mergesort ls) (mergesort rs)
  | otherwise       = xs
   where (ls, rs) = splitinhalf xs
```

- We will only use this part of the mergesort-algorithm since we can split it up right here by adding this to the `where`

```
[l,r] = runPar $ do merged <- parMap mergesort (ls:[rs])
                    return merged
```

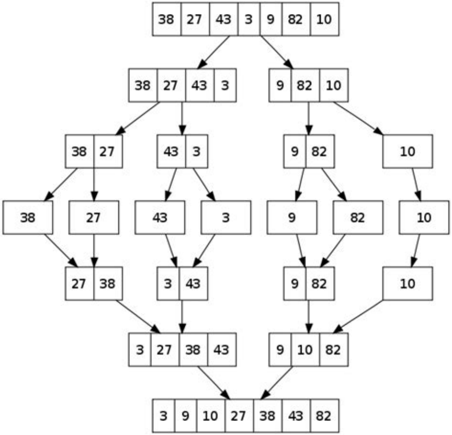
  and replacing

```
| (length xs) > 1 = merge l r
```

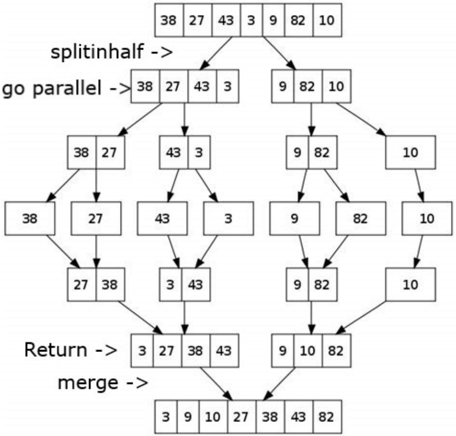
Examples

What are we sorting parallel?

- The normal `mergesort` flow diagram



- Where we go parallel in `mergesort`



Examples

Generalize - parFuncOnList

- To make things easier

```
parFuncOnList :: (NFData b) => [a] -> (a -> b) -> [b]
parFuncOnList xs f = runPar $ do
                        solutions <- parMap' f xs
                        return (solutions)
```

- To make things easier

```
parFuncOnList :: (NFData b) => [a] -> (a -> b) -> [b]
parFuncOnList xs f = runPar $ do
                            solutions <- parMap' f xs
                            return (solutions)
```

- Simplified `parleaves`

```
parleaves (_ :> b) = parFuncOnList b (leaves)
```

Examples

Generalize - parFuncOnList

- To make things easier

```
parFuncOnList :: (NFData b) => [a] -> (a -> b) -> [b]
parFuncOnList xs f = runPar $ do
                            solutions <- parMap' f xs
                            return (solutions)
```

- Simplified `parleaves`

```
parleaves (_ :> b) = parFuncOnList b (leaves)
```

- Simplified part of `parmergesort`

```
[l,r]                   = parFuncOnList (ls:[rs]) mergesort
```

- We can do a similar thing for `quicksort`

```
quicksort :: (Ord a) => [a] -> [a]
quicksort []     = []
quicksort (x:xs) = quicksort [y | y <- xs, y <= x]
                   ++ [x] ++
                   quicksort [y | y <- xs, y > x]
```

Examples
Quicksort

- We can do a similar thing for `quicksort`

```
quicksort :: (Ord a) => [a] -> [a]
quicksort []     = []
quicksort (x:xs) = quicksort [y | y <- xs, y <= x]
                   ++ [x] ++
                   quicksort [y | y <- xs, y > x]
```

- We can simply say that

```
parquicksort :: (Ord a, NFData a) => [a] -> [a]
parquicksort []     = []
parquicksort (x:xs) = l ++ [x] ++ r
    where [ls,rs] = [y | y <- xs, y <= x]
                    : ([y | y <- xs, y > x] : [])
          [l, r ] = parFuncOnList [ls,rs] quicksort
```

ThreadScoping

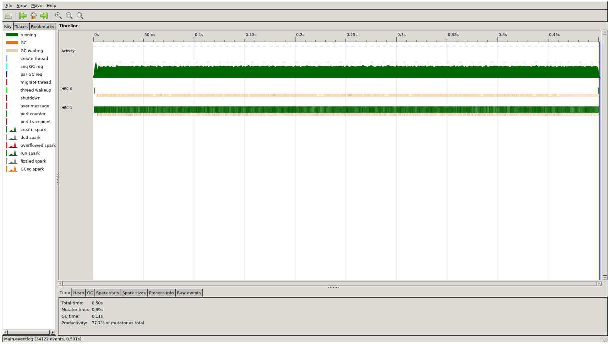
Is it faster?

- To test if these variants are faster we use ThreadScope which allows us to see the total runtime (and more)!

ThreadScoping

Is it faster?

- To test if these variants are faster we use ThreadScope which allows us to see the total runtime (and more)!
- It looks something like this



- When testing `parleaves` with some small handmade examples we note that there is no real time difference, so we had to use something big...

ThreadScoping

- When testing `parleaves` with some small handmade examples we note that there is no real time difference, so we had to use something big...
- Remember Assignment 2?

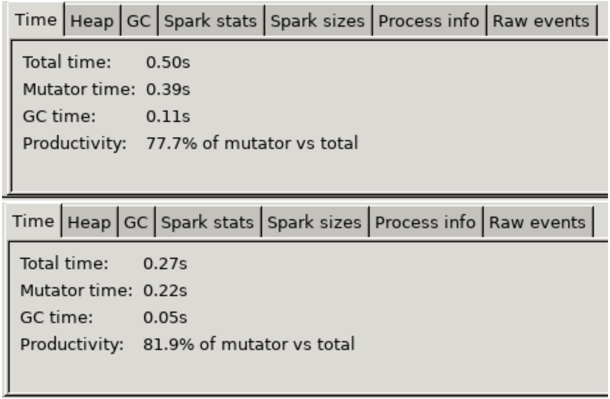
```
gameTreeComplexity :: Int
gameTreeComplexity = leaves (gameTree P1 emptyBoard)
```

Let's use this as our big data!

ThreadScoping

Results parleaves

- With `leaves` above and `parleaves` on the bottom



- So it's very fast!

- Similarly for our big data we used 1000 random baby names and put them into a JSON array

- Similarly for our big data we used 1000 random baby names and put them into a JSON array
- Here's the result for `quicksort` with the parallel variant on the bottom

- Similarly for our big data we used 1000 random baby names and put them into a JSON array
- Here's the result for `quicksort` with the parallel variant on the bottom

- Hard to see, but the normal one is 11ms and the parallel one is 6.6ms! The same can be shown for `mergesort`.

Conclusion
Faster and understandable

- `monad-par` often much faster when using big data.
- Next to that, it's not very hard to use!