Parallelism and concurrency in Haskell

David Luposchainsky

2017-12-07

Introduction

Concurrency vs. parallelism

Parallelism

- Goal: speedup
- Orthogonal to correctness
- Comparatively simple

Concurrency vs. parallelism

Parallelism

- Goal: speedup
- Orthogonal to correctness
- Comparatively simple

Concurrency

- Affects logical structure of a program
- Heavily interleaved with correctness
- Often very complicated

Talk outline

1. **Parallelism**
   Excellent case for laziness
2. **Concurrency with forks and mutable state**
   Nice abstractions, language agnostic
3. **Transactional memory**
   Unique to Haskell, envied by everyone else

Parallelism

“Go faster”

Checking primes

Problem: find primes in a list (e.g. for crypto)

```
isPrime :: Natural -> Bool

filterPrime :: [Natural] -> [Natural]
filterPrime = filter isPrime

primesBetween lo hi = filterPrime [lo..hi]
```

Checking primes

Problem: find primes in a list (e.g. for crypto)

```
isPrime :: Natural -> Bool

filterPrime :: [Natural] -> [Natural]
filterPrime = filter isPrime

primesBetween lo hi = filterPrime [lo..hi]
```

Unfortunately, this won't parallelize well.

`map` is simple enough to parallelize (why?). Can we use that fact?

map is simple enough to parallelize (why?). Can we use that fact?

```haskell
isPrime :: Natural -> Bool

filterPrime :: [Natural] -> [Natural]
filterPrime = map snd . filter fst . map (\n -> (isPrime n, n))
          -- `isPrime` isolated in `map` argument
          -- (using the decorate-operation-undecorate idiom)

primesBetween lo hi = filterPrime [lo..hi]
```

What's in the list?

```
map (\n -> (isPrime n, n))
    (take 1e5 [1e10..])
```

What's in the list?

```
map (\n -> (isPrime n, n))
    (take 1e5 [1e10..])
```

Thunks! Literally function applications of the lambda.

```
map (\n -> (isPrime n, n)) [1,2,3]  =  [ (\n -> (isPrime n, n)) 1
                                       , (\n -> (isPrime n, n)) 2
                                       , (\n -> (isPrime n, n)) 3 ]
```

Forcing

- $=$ Evaluation to a certain degree (e.g. *WHNF*)
- Primitive task that allows parallelism
- "I'll probably need this, so maybe evaluate it"

Evaluation strategies

```haskell
-- Control.Parallel.Strategies

-- A way to evaluate 'a's
type Strategy a = a -> Eval a
withStrategy :: Strategy a -> (a -> a)
```

Evaluation strategies

```haskell
-- Control.Parallel.Strategies

-- A way to evaluate 'a's
type Strategy a = a -> Eval a
withStrategy :: Strategy a -> (a -> a)

-- Primitives
r0   :: Strategy a -- "NOOP"
rseq :: Strategy a -- Sequential forcing
rpar :: Strategy a -- Parallel forcing
```

Evaluation strategies

```haskell
-- Control.Parallel.Strategies

-- A way to evaluate 'a's
type Strategy a = a -> Eval a
withStrategy :: Strategy a -> (a -> a)

-- Primitives
r0   :: Strategy a -- "NOOP"
rseq :: Strategy a -- Sequential forcing
rpar :: Strategy a -- Parallel forcing

-- Combining functions
evalList        :: Strategy a -> Strategy [a]
evalTraversable :: Traversable t => Strategy a -> Strategy (t a)
evalTuple2      :: Strategy a -> Strategy b -> Strategy (a,b)
```

Back to primes

```
filterPrime =
    map snd . filter fst . map (\n -> (isPrime n, n))
    --                        ^^^
    -- List of thunks we'd like to force in parallel
```

Back to primes

```
filterPrime =
    map snd . filter fst . map (\n -> (isPrime n, n))
    --                        ^^^
    -- List of thunks we'd like to force in parallel

filterPrimeParallel =
    map snd . filter fst . parallelize . map (\n -> (isPrime n, n))
```

Back to primes

```haskell
filterPrime =
    map snd . filter fst . map (\n -> (isPrime n, n))
    --                      ^^^
    -- List of thunks we'd like to force in parallel

filterPrimeParallel =
    map snd . filter fst . parallelize . map (\n -> (isPrime n, n))
  where
    parallelize :: [(a, b)] -> [(a, b)]
    parallelize = withStrategy tupleFstList

    tupleFstList :: Strategy [(a, b)]
    tupleFstList = parList parFst

    parFst :: Strategy (a,b)
    parFst = parTuple2 rseq r0
```

Back to primes

```
filterPrime =
    map snd . filter fst . map (\n -> (isPrime n, n))
    --                     ^^^
    -- List of thunks we'd like to force in parallel

filterPrimeParallel =
    map snd . filter fst . parallelize . map (\n -> (isPrime n, n))
  where
    parallelize :: [(a, b)] -> [(a, b)]
    parallelize = withStrategy tupleFstList

    tupleFstList :: Strategy [(a, b)]
    tupleFstList = parList parFst

    parFst :: Strategy (a,b)
    parFst = parTuple2 rseq r0

filterPrimeParallel =
    map snd . filter fst . parallelize . map (\n -> (isPrime n, n))
  where
    parallelize = withStrategy (parList (parTuple2 rseq r0))
```

Demo

```
 . ./run
[1 of 1] Compiling Main ( Primes.hs, Primes.o )
Linking Primes …
Running sequentially
45.38 s
Running in parallel
16.37 s
```

Recap

`Control.Parallel` for easy parallelism:

1. Define strategy according to your needs
2. Apply strategy, done

Threads are cheap

- Couple of hundred (heap) bytes per thread
- Literally millions are possible
- Don't be afraid of using them!

```
>>> time ./Main.hs 1234567
Forking 1234567 threads
Start the counting cascade
Counter: 1234567
./Main.hs 1234567  5,38s user 0,66s system 100% cpu 6,001 total
```

Other parallelism libraries

- **monad-par**: Explicit dataflow networks for complicated (pure or effectful) computations
- **speculation**: the one-liner to solve all pure speculative parallelism needs
- **Haxl**: See Simon Marlow's BTD 10 talk

Concurrency

Basic primitives

```haskell
-- Threads
forkIO :: IO () -> IO ThreadId
forkFinally :: IO a
            -> (Either SomeException a -> IO ())
            -> IO ThreadId
```

Basic primitives

```haskell
-- Threads
forkIO :: IO () -> IO ThreadId
forkFinally :: IO a
            -> (Either SomeException a -> IO ())
            -> IO ThreadId

-- Futures/promises
async :: IO a -> IO (Async a)
wait :: Async a -> IO a
```

Basic primitives

```haskell
-- Threads
forkIO :: IO () -> IO ThreadId
forkFinally :: IO a
            -> (Either SomeException a -> IO ())
            -> IO ThreadId

-- Futures/promises
async :: IO a -> IO (Async a)
wait :: Async a -> IO a

-- Locking
data MVar a
newMVar  :: a -> IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar  :: a -> MVar a -> IO ()
```

Example: webserver

```
main :: IO ()
main = do
    numClients <- newMVar 0
    forever (acceptClient numClients)

acceptClient :: MVar Int -> IO ()
acceptClient numClients = do
    connection <- listenOn (Port 8080)
    forkFinally (\_ -> close connection)
                (handleClient numClients connection)
```

Example: webserver

```haskell
main :: IO ()
main = do
    numClients <- newMVar 0
    forever (acceptClient numClients)

acceptClient :: MVar Int -> IO ()
acceptClient numClients = do
    connection <- listenOn (Port 8080)
    forkFinally (\_ -> close connection)
                (handleClient numClients connection)

handleClient :: MVar Int -> Handle -> IO ()
handleClient numClients connection = do
    modifyMVar numClients (+1)
    getHttpMethod connection >>= \case
        Head   -> …
        Get    -> …
        Delete -> …
        …
    modifyMVar numClients (subtract 1)
```

Example: service chaining

```
main = do
    let customerId = 123
    addressA <- async (queryAddress customerId)
    inventoryA <- async (queryInventory customerId)
```

Example: service chaining

```
main = do
    let customerId = 123
    addressA <- async (queryAddress customerId)
    inventoryA <- async (queryInventory customerId)

    address <- wait addressA
    capabilitiesA <- async (queryCapabilities customerId address)
```

Example: service chaining

```haskell
main = do
    let customerId = 123
    addressA <- async (queryAddress customerId)
    inventoryA <- async (queryInventory customerId)

    address <- wait addressA
    capabilitiesA <- async (queryCapabilities customerId address)

    capabilities <- wait capabilities

    -- more stuff here

    inventory <- wait inventoryA
    doStuff address inventory address
```

Higher abstractions

- Channels: queues built with `MVars`
- Semaphores: more flexible locking
- Concurrent data structures

STM

Transactional memory

```haskell
modifyMVar :: (a -> a) -> MVar a -> IO ()
modifyMVar f mVar = do
    value <- takeMVar mVar
    putMVar mVar (f value)
```

Transactional memory

```haskell
modifyMVar :: (a -> a) -> MVar a -> IO ()
modifyMVar f mVar = do
    value <- takeMVar mVar
    putMVar mVar (f value)
```

What happens when another thread fills the `MVar` between the take and put?

Standard API

```
data IO a
instance Monad IO where …

data MVar a
newMVar    :: a -> IO (MVar a)
takeMVar   :: MVar a -> IO a
putMVar    :: MVar a -> a -> IO ()
modifyMVar :: (a -> a) -> MVar a -> IO () -- Dangerous
```

Transactional API

```
data STM a
instance Monad STM where …

data TVar a
newTVar    :: a -> STM (TVar a)
takeTVar   :: TVar a -> STM a
putTVar    :: TVar a -> a -> STM ()
modifyTVar :: (a -> a) -> TVar a -> STM ()

retry  :: STM a
orElse :: STM a -> STM a -> STM a

atomically :: STM a -> IO a
```

Transactional API

```haskell
data STM a
instance Monad STM where …

data TVar a
newTVar    :: a -> STM (TVar a)
takeTVar   :: TVar a -> STM a
putTVar    :: TVar a -> a -> STM ()
modifyTVar :: (a -> a) -> TVar a -> STM ()

retry  :: STM a
orElse :: STM a -> STM a -> STM a

atomically :: STM a -> IO a

modifyTVar :: (a -> a) -> TVar a -> STM ()
modifyTVar f tVar = do
    value <- readTVar tVar
    writeTVar tVar (f value)
```

How does it work?

- STM actions write to a log of changes
- `atomically` attempts to commit the log
- Changed circumstances lead to `retry`ing the action
- Repeat until successful

Should I use STM?

Pros

- No deadlocks by design
- Guaranteed atomicity

Cons

- Livelocks still possible
- Stampending herd
- No fairness guarantees
- Long transactions undesirable

Speculative parallelism

“I think this is the answer
so I’ll continue with that assumption
and verify it in parallel”

…is a one-liner

```haskell
spec :: Eq a => a -> (a -> b) -> a -> b
spec g f a = let s = f g in s `par` if g == a then s else f a in spec
```

…is a one-liner

```haskell
spec :: Eq a => a -> (a -> b) -> a -> b
spec g f a = let s = f g in s `par` if g == a then s else f a in spec
```

- par is an old version of `rpar` from `Control.Parallel`
- Implemented by Edward Kmett
- …after reading a 12-page paper attempting the same in C#
- …in a Reddit comment

Add two minor modifications,

- Parametrize over the equality function
- No speculation with only one thread

```haskell
-- Copied verbatim from the `speculation` package

specBy :: (a -> a -> Bool) -> a -> (a -> b) -> a -> b
specBy cmp guess f a
  | numCapabilities == 1 = f $! a
  | otherwise = speculation `par`
      if cmp guess a
          then speculation
          else f a
  where speculation = f guess

spec :: Eq a => a -> (a -> b) -> a -> b
spec = specBy (==)
```

How does it work?

- Threads are heap objects
- Heap objects are tracked by the GC
- Obsolete threads are simply GC'd
- Fork as long as you have memory and CPU free

Things not covered

- Exceptions
- Debugging
- FFI integration
- Distributed processes
- Data parallel computations
- GPGPU
- Other semiautomatic parallelization libs

Questions?