Haskell - Concurrency and parallelism

Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

Spring 2018

Outline

Parallelism and concurrency

Lightweight threads

STM

Data parallelism and Futhark

Concurrency

Definition: **code is split up and executed out-of-order**

Concurrency

Definition: **code is split up and executed out-of-order**

Advantage: Possible to exploit multi-core architecture

Concurrency

Definition: **code is split up and executed out-of-order**

Advantage: Possible to exploit multi-core architecture

Disadvantage:

1. Race-conditions
2. Deadlocking
3. Starvation

Definition: **code is split up and executed *at the same time***

Parallelism

Definition: **code is split up and executed *at the same time***

Code can thus be *concurrenc* but not *parallel*. What is an example?

Concurrency vs. parallelism

cphbusiness COPENHAGEN BUSINESS ACADEMY

- Concurrency is when many things are happening in random order
- Parallelism is when many things happen at the same time

Concurrency vs. parallelism

cphbusiness
COPENHAGEN BUSINESS ACADEMY

- Concurrency is when many things are happening in random order
- Parallelism is when many things happen at the same time

Problems of concurrency

- Not always *deterministic*
- Complicated to keep track of data flow
- Sometimes depending on concurrent processes (idle locks)

Correct parallelism is

- Deterministic (same outcome)
- High-level declarative (does not deal with synchronisation or communication)

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Haskell provides tons of tools for this

- Par Monad
- Eval Monad
- MVars
- IO Manager
- Asynchronous exceptions
- Software Transactional Memory (STM)
- Lightweight threads

Haskell provides tons of tools for this

- Par Monad
- Eval Monad
- MVars
- IO Manager
- Asynchronous exceptions
- **Software Transactional Memory (STM)**
- **Lightweight threads**

- Lastly: Data parallelism and Futhark

Credits

A lot of the material in these slides have been borrowed from a course on parallel functional programming on Copenhagen University. Especially from slides on Haskell by Ken Friis Larsen.

Some of the material is also borrowed from the book on Parallel and Concurrent Programming in Haskell:
`http://shop.oreilly.com/product/0636920026365.do`

In Java:

```
import java.util.Thread;
Thread t = new Thread(() -> ...);
```

Lightweight threads

In Java:

```
import java.util.Thread;
Thread t = new Thread(() -> ...);
```

In Haskell:

```
import Control.Concurrent
-- forkIO :: IO () -> IO ThreadId

main = do
  threadId <- forkIO $ putStrLn "FP rocks!"
```

Communicating with Threads

Same problem as in Java: How would you get data in/out of a thread?

Communicating with Threads

Same problem as in Java: How would you get data in/out of a thread?
Solution: external variables!

An `MVar` is exactly that: a *atomic* variable.

An `MVar` is exactly that: a *atomic* variable.

```
data MVar a -- abstract

newEmptyMVar :: IO (MVar a)
readMVar :: MVar a -> IO a
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
```

Threads example

```haskell
getURL :: String -> IO String
```

Threads example

```haskell
getURL :: String -> IO String
```

```haskell
main = do
  m1 <- newEmptyMVar
  m2 <- newEmptyMVar
  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Shovel"
    putMVar m1 r
  forkIO $ do
    r <- getURL "http://www.wikipedia.org/wiki/Spade"
    putMVar m2 r
  r1 <- takeMVar m1
  r2 <- takeMVar m2
  return (r1,r2)
```

```haskell
data Async a = Async (MVar a)

async :: IO a -> IO (Async a)
async action = do
  var <- newEmptyMVar
  forkIO $ action >>= putMVar var
  return $ Async var

wait :: Async a -> IO a
wait (Async var) = readMVar var
```

```haskell
main = do
  a1 <- async $
    getURL "http://www.wikipedia.org/wiki/Shovel"
  a2 <- async $
    getURL "http://www.wikipedia.org/wiki/Spade"
  r1 <- wait a1
  r2 <- wait a2
  return (r1,r2)
```

Using the async pattern

```haskell
main = do
  a1 <- async $
    getURL "http://www.wikipedia.org/wiki/Shovel"
  a2 <- async $
    getURL "http://www.wikipedia.org/wiki/Spade"
  r1 <- wait a1
  r2 <- wait a2
  return (r1,r2)
```

```haskell
getURLs :: [String] -> IO [ByteString]
getURLs sites =
  mapM (async.getURL) sites >>= mapM wait
```

Concurrency problems

Remember the concurrency problems:

1. Race-conditions
2. Deadlocking
3. Starvation

What is the root of these problems?

Parallel problems

In parallel computing we generally face two problems:

- Granularity
- Data dependency

In parallel computing we generally face two problems:

- Granularity
- Data dependency

Solution: Think of parallelism as a series of transactions

Just like a SQL transaction STMs work by

1. storing the current state of affairs

Just like a SQL transaction STMs work by

1. storing the current state of affairs
2. trying to execute transaction

Just like a SQL transaction STMs work by

1. storing the current state of affairs
2. trying to execute transaction
3. roll back to previous state if error

Software Transactional Memory

Just like a SQL transaction STMs work by

1. storing the current state of affairs
2. trying to execute transaction
3. roll back to previous state if error
4. commit change if success

Just like a SQL transaction STMs work by

1. storing the current state of affairs
2. trying to execute transaction
3. roll back to previous state if error
4. commit change if success

Just like a SQL transaction STMs work by

1. storing the current state of affairs
2. trying to execute transaction
3. roll back to previous state if error
4. commit change if success

This is a genious use case for monads: our monad simply *is* the state

cphbusiness
COPENHAGEN BUSINESS ACADEMY

```haskell
import Control.Concurrent.STM

data STM a -- abstract
instance Monad STM -- among other things

atomically :: STM a -> IO a
retry :: STM a
orElse :: STM a -> STM a -> STM a

data TVar a -- abstract
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

STM example: Banking

```haskell
type Amount = Int
type Account = TVar Amount
```

STM example: Banking

```haskell
type Amount = Int
type Account = TVar Amount
```

```haskell
transfer :: Account -> Account -> Amount -> IO ()
transfer from to amount = atomically $ do
  deposit  to   amount
  withdraw from amount
```

STM example: Banking

```haskell
type Amount = Int
type Account = TVar Amount
```

```haskell
transfer :: Account -> Account -> Amount -> IO ()
transfer from to amount = atomically $ do
  deposit  to   amount
  withdraw from amount
```

```haskell
deposit :: Account -> Amount -> STM ()
deposit account amount = do
  balance <- readTVar account
  writeTVar account $ balance + amount
withdraw :: Account -> Amount -> STM ()
withdraw account amount = deposit account (- amount)
```

STM example: Banking 2

```haskell
limitedWithdraw :: Account -> Amount -> STM ()
limitedWithdraw account amount = do
  balance <- readTVar account
  if amount > 0 && amount > balance
    then retry
    else writeTVar account $ balance - amount
```

```haskell
limitedWithdraw :: Account -> Amount -> STM ()
limitedWithdraw account amount = do
  balance <- readTVar account
  if amount > 0 && amount > balance
    then retry
    else writeTVar account $ balance - amount
```

```haskell
backupWithdraw :: Account -> Account -> Amount -> STM (
backupWithdraw acc1 acc2 amt =
  (limitedWithdraw acc1 amt)
    `orElse`
  (limitedWithdraw acc2 amt)
```

Performance

Threads and STM still have performance problems

Performance

Threads and STM still have performance problems

What about GPU acceleration you say?

Performance

Threads and STM still have performance problems

What about GPU acceleration you say?

Functional programming is historically bad at this

Performance

Threads and STM still have performance problems

What about GPU acceleration you say?

Functional programming is historically bad at this

Solution: Data parallelism

Data parallelism

Until now we are parallelising program logic, not data

Data parallelism

Until now we are parallelising program logic, not data

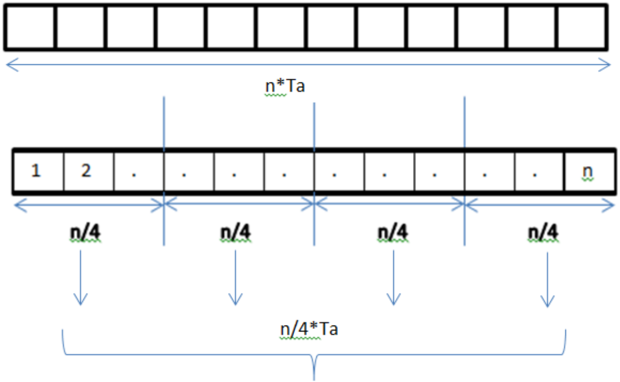
Task parallelism: Break problem into small parts, delegate to threads

Until now we are parallelising program logic, not data

Task parallelism: Break problem into small parts, delegate to threads

Data parallelism: Break data into small parts, delegate to threads

Work versus span



Work: The amount of instructions to do, assuming a single PU
Span: The longest series of instructions, assuming infinite PU

Using data parallelism

$$\underbrace{\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 1 & 3 & 2\end{pmatrix}}_{3 \times 3}\underbrace{\begin{pmatrix}10 & 11\\ 7 & 5\\ 2 & 4\end{pmatrix}}_{3 \times 2} = \underbrace{\begin{pmatrix}1*10+2*7+3*2 & 1*11+2*5+3*4\\ 4*10+5*7+6*2 & 4*11+5*5+6*4\\ 1*10+3*7+2*2 & 1*11+3*5+2*4\end{pmatrix}}_{3 \times 2}$$

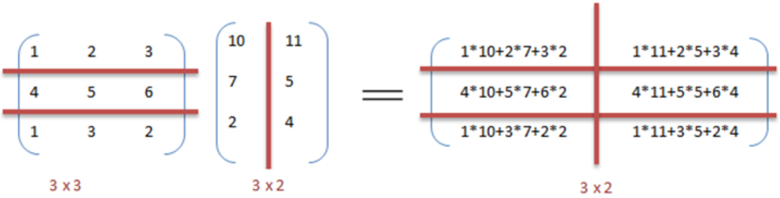
We can compute one cell at the time, giving $O(n^3)$

Using data parallelism

$$\underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 3 & 2 \end{pmatrix}}_{3 \times 3} \underbrace{\begin{pmatrix} 10 & 11 \\ 7 & 5 \\ 2 & 4 \end{pmatrix}}_{3 \times 2} = \underbrace{\begin{pmatrix} 1*10+2*7+3*2 & 1*11+2*5+3*4 \\ 4*10+5*7+6*2 & 4*11+5*5+6*4 \\ 1*10+3*7+2*2 & 1*11+3*5+2*4 \end{pmatrix}}_{3 \times 2}$$

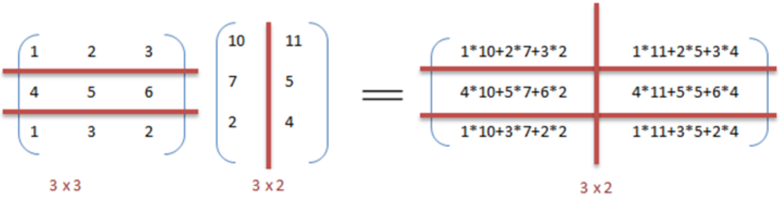
We can compute one cell at the time, giving $O(n^3)$ (work)

Using data parallelism



We can compute one cell at the time, giving $O(n^3)$ (work)

Or we can compute all cells in parallel, giving $O(1)$



We can compute one cell at the time, giving $O(n^3)$ (work)

Or we can compute all cells in parallel, giving $O(1)$ (span)

GPU acceleration

This is similar to the Single Instruction Mulitiple Data `SIMD` idea

GPU acceleration

This is similar to the Single Instruction Mulitiple Data `SIMD` idea

Incredibly useful in GPU (CUDA) programming

GPU acceleration

This is similar to the Single Instruction Mulitiple Data `SIMD` idea

Incredibly useful in GPU (CUDA) programming

Used in the OpenCL standard for GPUs

Futhark is a data parallel programming language developed here in Copenhagen

`https://futhark-lang.org/`

Futhark is a data parallel programming language developed here in Copenhagen

`https://futhark-lang.org/`

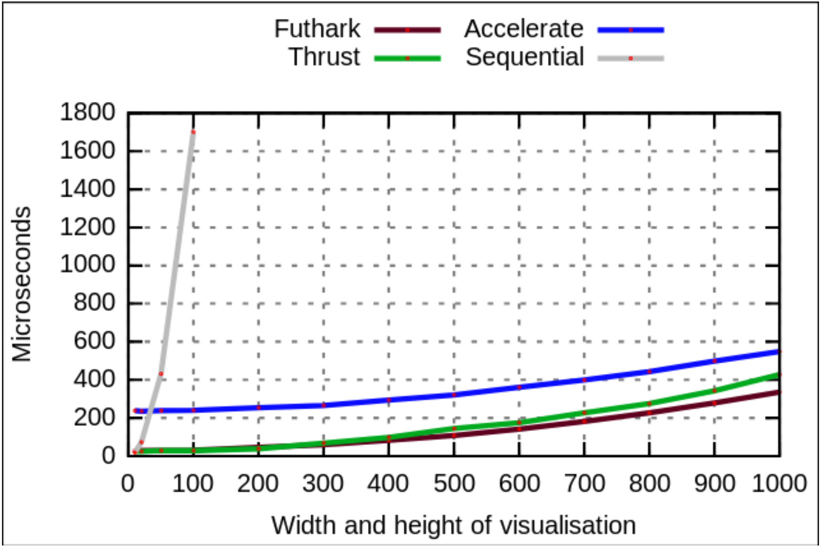
```
    let fact (n: i32): i32 = reduce ) 1 (1...n)
```

Futhark is a data parallel programming language developed here in Copenhagen

`https://futhark-lang.org/`

```
let fact (n: i32): i32 = reduce ) 1 (1...n)
```

Futhark performance



Recap

- Concurrency vs. parallelism
- Lightweight threading with `forkIO`
- Software Transactional Memory (STM)
- Task parallelism versus data parallelism
- Futhark and GPU accelerated functional programming