Parallelität und Nebenläufigkeit mit Haskell

Stefan Wehr (wehr@factisresearch.com)
factis research GmbH, Freiburg im Breisgau

16. Mai 2013
parallel2013, Karlsruhe

Warum funktional?

- Bewusster Umgang mit Seiteneffekten
- Wenig globaler Zustand
- Datenabhängigkeiten oft explizit

Warum Haskell?

- Ausdrucksstarkes Typsystem: Wenn das Programm kompiliert funktioniert es auch!
- Kurzer, prägnanter und lesbarer Code
- Einfaches Abstrahieren
- "World's finest imperative programming language"
  - Unterschied zwischen Aktion ("Statements") mit Seiteneffekten und reinen Ausdrücken
  - Aktionen sind Werte erster Klasse: Eigene Kontrollstrukturen, Programmatische Konstruktionen von Aktionen
- Reichhaltige Palette an Ansätzen zur parallelen und nebenläufigen Programmierung
  - Deterministischer Parallelismus
  - Datenparallelismus
  - GPU-Programmierung
  - STM
  - Leichtgewichtige Threads

Mein Hintergrund

- Haskell Benutzer seit 2003
- Zunächst vor allem im akademischen Bereich
- Seit 2010: Anwendung von Haskell in der Industrie
- factis research GmbH, Freiburg im Breisgau
    - Softwareprodukte für den Medizin- und Pflegebereich
    - Serverseitige Software fast ausschließlich in Haskell geschrieben
    - Große Erfahrung mit komplexen mobilen Anwendungen
    - Projekte und Schulungen im Bereich funktionale Programmierung und mobile Anwendungen

Nebenläufigkeit

- Nebenläufigkeit ist ein eigenständiges Ziel
  - Gleichzeitige Kommunikation mit mehreren externen Quellen
  - Reaktion auf Events der Außenwelt
- Notwendigerweise nicht-deterministisch

Schwierigkeiten

- Deadlocks
- Race conditions
- Nicht-deterministisches Verhalten
- Testabdeckung

Vorteile von Nebenläufigkeit

- Manche Probleme werden durch Nebenläufigkeit auch einfacher
- Beispiel: Netzwerkserver mit einem Thread pro Client
  - Sequentieller Ablauf pro Client
  - Alternative: Gleichzeitige Interaktion mit allen Clients
    - Komplizierte Zustandsmaschine

Parallelität

- Parallelität ist kein Ziel *per se*
- Ziel: Programm soll schneller laufen
- Parallelität ist ein möglicher Weg dieses Ziel zu erreichen
  - Benutzung mehrerer CPU Kerne
  - Benutzung von GPU Kernen
- Kann determinisisch sein

Schwierigkeiten bei der Parallelisierung

- Welche Teile des Programms sind langsam?
- Welche Teile des Programms können parallelisiert werden? Datenabhängigkeiten?
- Granularität
  - Zu fein: Overhead wird zu groß
  - Zu grob: nicht genug Parallelität möglich

Einführung in Haskell

- Haskell ist eine statische getypte Programmiersprache
  - Jeder Ausdruck hat zur Kompilierzeit einen Typ
  - Typsystem erlaubt Differenzierung nach Seiteneffekten
  - Typen müssen nicht explizit hingeschrieben werden (Typinferenz)
  - Gute Angewohnheit: Schreiben Typsignaturen an top-level Funktionen
- Haskell ist eine funktionale Programmiersprache
  - Funktionen nehmen eine zentral Rolle ein
  - Funktionen als Parameter und Rückgabewerte anderer Funktionen
- Lazy Auswertung
  - Ausdrücke werden erst dann ausgewertet wenn ihr Ergebnis gebraucht wird

Funktionen in Haskell

- Ohne Seiteneffekte

```
formatAge :: String -> Int -> String
formatAge name age =
    name ++ " ist " ++ show age ++ " Jahre alt!"

*Main> formatAge "Stefan" 34
"Stefan ist 34 Jahre alt!"
```

- Mit Seiteneffekten

```
sayAge :: String -> IO ()
sayAge name =
    do putStrLn ("Wie alt ist " ++ name ++ "? ")
       ageStr <- getLine
       putStrLn (formatAge name (read ageStr))
```

Listen in Haskell

- Eine Liste ...
  - ... ist entweder leer: `[]`
  - ... oder besteht aus einem Element `x` und einer Restliste `rest`: `x : rest`
- Typ einer Liste: `[a]` wobei `a` der Typ der Listenelemente ist

```haskell
countdown :: [Int]
countdown =
  10 : 9 : 8 : 7 : 6 : 5 : 4 : 3 : 2 : 1 : 0 : []

zutaten :: [String]
zutaten = ["Nudeln", "Tomaten", "Salz"]
```

Rekursive Funktionen in Haskell

```
incList :: [Int] -> [Int]
incList []        = []
incList (x:rest) = x + 1 : incList rest
```

```
*Main> incList countdown
[11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
incList' :: [Int] -> [Int]
incList' l = map (\x -> x + 1) l
```

```
map :: (a -> b) -> [a] -> [b]
map f []        = []
map f (x:rest) = f x : map f rest
```

Typparameter und Constraints in Haskell

- `map :: (a -> b) -> [a] -> [b]` ist generisch (oder polymorph) in den Typparametern `a` und `b`
- Constraints schränken Typparameter ein und machen gewissen Funktionalitäten verfügbar
- Constraint-System ist erweiterbar
- Beispiel: Gleichheitsoperator == nur verfügbar für Typen `a` mit Constraint `Eq a`

```
elem :: Eq a => a -> [a] -> Bool
elem y []       = False
elem y (x:rest) = if x == y then True else elem y rest
```

Monaden in Haskell

- Programmierbares Semikolon
- Abstraktion über Sequenzierung
- do-Notation
- IO ist eine Monade

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []        = return []
mapM f (x:rest) =
     do y <- f x
        rest' <- mapM f rest
        return (y:rest')

*Main> mapM sayAge ["Stefan", "Georg"]
```

- Definition eigener Monaden möglich

Deterministischer Parallelismus in Haskell

- Die `Par` Monade
- Explizite Datenflussinformationen durch Verwendung spezieller "Boxen"
- Semi-automatische Parallelisierung

Beispiel: Sudoku

- Parallelisierung durch gleichzeitiges Lösen mehrere Rätsel
- Keine Parallelisierung des Algorithmus zum Lösen *eines* Rätsels
- Gegeben:
  - Funktion zum Lösen eines Rätsels: `solve :: String -> Maybe Grid`
  - `data Maybe a = Just a | Nothing`
- Gesucht: Funktion zum Lösen mehrerer Rätsel: `computeSolutions :: [String] -> [Maybe Grid]`

Sequentielles Sudoku-Lösen

```
-- Datei: SudokuSeq.hs
computeSolutionsSeq :: [String] -> [Maybe Grid]
computeSolutionsSeq puzzles =
    map solve puzzles
```

- Kompilieren: `ghc --make -O2 -threaded -rtsopts SudokuSeq.hs`
- Ausführen: `./SudokuSeq +RTS -s -RTS sudoku17.1000.txt`
- Laufzeit: `Total time 1.52s ( 1.53s elapsed)`
  - CPU-Zeit: 1,52 Sekunden
  - Wall-Zeit: 1,53 Sekunden

Grundlagen der `Par` Monade

- Installation: `cabal install monad-par`
- `Par a`
  - Typ einer Aktion in der `Par`-Monad mit Ergebnistyp a
- `fork :: Par () -> Par ()`
  - Führt eine Aktion parallel aus
- `runPar :: Par a -> a`
  - Bringt Aktionen in der `Par`-Monade zurück in die Welt der normale Haskell-Berechnungen.
  - Ohne Seiteneffekte (sieht man am Typ!)
  - Deterministisch (sieht man am Typ!)

Datenfluss in der `Par` Monade

- Parallele Aktionen kommunizieren über Boxen
- `IVar a`: Box für Werte vom Typ a
- Boxen beschreiben den Datenfluss zwischen Aktionen
  - `new :: Par (IVar a)`
  - `get :: IVar a -> Par a`
  - `put :: NFData a => IVar a -> a -> Par ()`
  - Wichtig: Jede Box darf nur einmal beschrieben werden, sonst Aktion nicht mehr deterministisch

Sudoku: statische Paritionierung mit Par

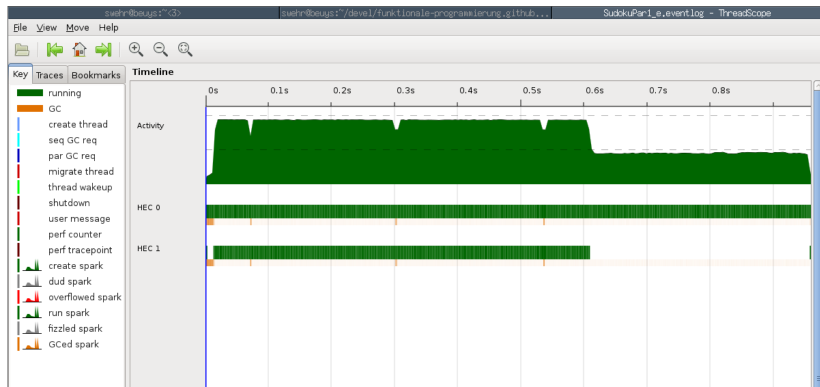
```haskell
-- Datei: SudokuPar1.hs
computeSolutionsPar1 :: [String] -> [Maybe Grid]
computeSolutionsPar1 puzzles =
  let n = length puzzles
      (as, bs) = splitAt (n `div` 2) puzzles
  in runPar (f as bs)
  where
    f :: [String] -> [String] -> Par [Maybe Grid]
    f as bs = do b1 <- new
                 b2 <- new
                 fork (put b1 (map solve as))
                 fork (put b2 (map solve bs))
                 res1 <- get b1
                 res2 <- get b2
                 return (res1 ++ res2)
```

Performance mit statischer Partitionierung

- Laufzeit auf einem Kern:
  - `./SudokuPar1 +RTS -s -RTS sudoku17.1000.txt`
  - 1,52s CPU-Zeit, 1,54s Wall-Zeit
- Mit 2 Kernen: Laufzeitoption `-N<K>` mit `K` Anzahl der Kerne
  - `./SudokuPar1 +RTS -s -N2 -RTS sudoku17.1000.txt`
  - 1,57s CPU-Zeit, 0,97s Wall-Zeit
- Speedup: $1,53/0,97 = 1,55$

Debugging mit Threadscope

- Kompilieren: `ghc --make -O2 -threaded -rtsopts -eventlog -o SudokuPar1_e SudokuPar1.hs`
- Ausführen: `./SudokuPar1_e +RTS -s -N2 -ls -RTS sudoku17.1000.txt`
- Eventlog in Threadscope öffnen



Probleme mit statischer Partitionierung

- Verschiedene Sudokurätsel brauchen unterschiedliche viel Zeit zum Lösen
  - Statische Partitionierung teilt Liste der Problem einfach in der Mitte
- Statische Partitionierung legt sich auf zwei parallele Berechnungen fest
  - Laufzeit mit vier Kernen schlechter als mit zwei Kernen
    - `./SudokuPar1 +RTS -s -N4 -RTS sudoku17.1000.txt`
    - 1,80s CPU-Zeit, 1,03s Wall-Zeit

Sudoku: dynamische Paritionierung

- Skaliert auf beliebig viele Kerne
- Kommt mit unterschiedlichen Größen der Teilprobleme zurecht
- Idee:
  - Teile Liste der Sudokurätsel in viele kleine Einheiten
  - Laufzeitsystem kümmert sich um Aufteilung der Einheiten auf Prozessorkerne

Dynamische Partitionierung mit Par

- Rückblick auf sequenzielle Variante:

```
computeSolutionsSeq puzzles = map solve puzzles
```

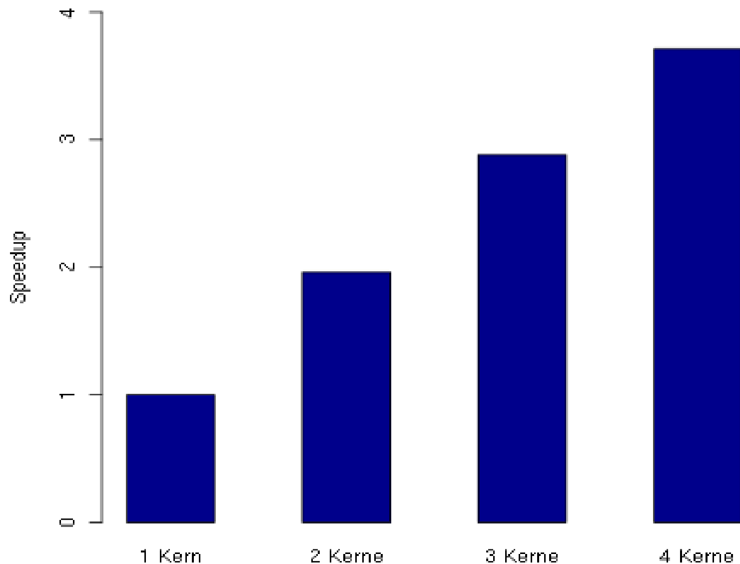
- Parallele Variante von map

```
map    ::                (a -> b) -> [a] ->     [b]
parMap :: NFData b => (a -> b) -> [a] -> Par [b]
```

- Parallele Variante des Sudoku-Solvers

```
computeSolutionsPar2 :: [String] -> [Maybe Grid]
computeSolutionsPar2 puzzles =
    runPar (parMap solve puzzles)
```

Speedup mit dynamischer Partitionierung



Implementierung von parMap

- Idee:
  - parMap f xs wertet f auf jedes Element aus xs in einer neuen, parallelen Berechung aus
  - Ergebnis einer solchen Berechnung landet in einer Box
  - Am Schluss werden alle Ergebnisse aus den Boxen eingesammelt

```
parMap :: NFData b => (a -> b) -> [a] -> Par [b]
parMap f xs =
    do bs <- mapM g xs
       mapM get bs
    where
      g x =
          do b <- new
             fork (put b (f x))
             return b
```

Was ist Datenparallelismus?

- Gleichzeitiges Anwenden derselben Operation auf unterschiedliche Daten
- SIMD-Instruktionen
- Flacher Datenparallelismus:
  - Paralleles Anwenden *sequentieller* Operationen
  - Weitverbreitet (MPI, HPF, ...)
  - Begrenzte Anwendungsmöglichkeiten
- Geschachtelter Datenparallelismus:
  - Paralleles Anwenden *paralleler* Operationen
  - Sehr viele Anwendungsmöglichkeiten
    - Divide-And-Conquer Algorithmen
    - Graphenalgorithmen
    - Machine Learning
    - ...
  - Modularer als flacher Datenparallelismus
  - Forschungsthema, Implementierung sehr schwierig

Accelerate: flacher Datenparallelismus auf der GPU

- GPUs sind parallele Mehrkernprozessoren
- Spezialisiert auf parallele Grafikoperationen (flacher Datenparallelismus)
- GPGPU: General Purpose Computation on Graphics Processing Unit
  - OpenCL, CUDA
  - Sehr aufwändig
- Idee: Erzeuge aus Haskell Code ein GPU Programm
  - Accelerate ist eine *EDSL* (Embedded Domain Specific Language)
  - Accelerate ist eine (kleine) Teilmenge von Haskell
  - Syntaxbaum der EDSL wird zur Laufzeit reifiziert und mittels CUDA auf die GPU gebracht
  - Performance: etwas langsamer als natives CUDA, aber akzeptabel

Accelerate, ein Beispiel

```
import qualified Data.Array.Accelerate as A

dotp :: A.Vector Float
     -> A.Vector Float
     -> A.Acc (A.Scalar Float)
dotp xs ys =
    let xs' = A.use xs
        ys' = A.use ys
    in A.fold (+) 0 (A.zipWith (*) xs' ys')
```

- `A.use` transferiert die Daten auf die GPU
- `A.fold` und `A.zipWith` werden auf der GPU ausgeführt

DPH: geschachtelter Datenparallelismus

- Paralleles Anwenden *paralleler* Operationen
- Viel flexibler und modularer als flacher Datenparallelismus
- *DPH*: Data Parallel Haskell
- Zentraler Begriff: parallele Arrays, z.B. `[: Double :]`
- Operationen über parallele Arrays werden automatisch parallelisiert

DPH, ein Beispiel

- Multiplikation dünnbesetzter Matrizen mit einem Vektor
- `[:Double:]`: dichtbesetzer Vektor
- `[:(Int, Double):]`: dünnbesetzter Vektor
- Dünnbesetzte Matrix: `[:[:(Int,Double):]:]`

```
svMul :: [:(Int,Double):] -> [:Double:] -> Double
svMul sv v = sumP [: (v !: i) * f | (i,f) <- sv :]

smMul :: [:[:(Int,Double):]:] -> [:Double:] -> Double
smMul sm v = sumP [: svMul row v | row <- sm :]
```

Nebenläufigkeit mit Haskell

- Threads sind extrem billig
  - `forkIO :: IO () -> IO ThreadId`
  - Green Threads
  - Mehrere Million Threads auf diesem Laptop keine Problem
  - Gute Interaktion mit blockierenden Systemcalls und nativen Threads
- Synchronisation mit STM
  - Relativ einfach (im Gegensatz zu Locks)
  - Modular (im Gegensatz zu Locks)

STM (Software Transactional Memory)

- Beispiel: Überweisung von einem Bankkonto
  - `transfer acc1 acc2 amount`: überweise Betrag `amount` von Konto `acc1` auf `acc2`
  - `transfer` soll thread-safe sein
  - Im Beispiel: Konten sind nicht persistenz
- Probleme mit Threads
  - Deadlocks, Race conditions
  - Unmodular
- Idee von STM:
  - Markiere Codeblöcke als “atomar”
  - Atomare Blöcke werden entweder ganz oder gar nicht ausgeführt (wie Datenbanktransaktionen)
- Vorteile von Haskell:
  - Immutability
  - Lazy Auswertung

Kontoüberweisungen mit STM

```
transfer :: Account -> Account -> Int -> IO ()
transfer acc1 acc2 amount =
    atomically (do deposit acc2 amount
                   withdraw acc1 amount)
```

- `atomically :: STM a -> IO a`
  - Führt die übergebene Aktion atomar aus
  - Auszuführende Aktion wird auch *Transaktion* genannt
  - `STM a`: Typ einer Transaktion mit Ergebnis vom Typ `a`
  - `STM` ist eine Monade

Transaktionsvariablen

- STM-Aktionen kommunizieren über *Transaktionsvariablen*
  - `TVar a`: Transaktionsvariablen die Wert vom Typ a enthält
- Lesen: `readTVar :: TVar a -> STM a`
- Schreiben: `writeTVar :: TVar a -> a -> STM ()`
- Außerhalb von `atomically` können Transaktionsvariablen weder gelesen noch geschrieben werden.

```haskell
type Account = TVar Int

deposit :: Account -> Int -> STM ()
deposit acc amount =
    do bal <- readTVar acc
       writeTVar acc (bal + amount)

withdraw :: Account -> Int -> STM ()
withdraw acc amount = deposit acc (- amount)
```

Blockieren mit STM

- Warten auf eine bestimmte Bedingung ist essential für nebenläufige Programme
- Beispiel: `limitedWithdraw` soll blockieren falls nicht genug Geld zum Abheben da ist

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount =
    do bal <- readTVar acc
       if amount > 0 && amount > bal
       then retry
       else writeTVar acc (bal - amount)
```

- `retry :: STM a`
  - Bricht die aktuelle Transaktion ab
  - Versucht zu einem späteren Zeitpunkt die Transaktion nochmals auszuführen

Kombinieren von Transaktionen

- Beispiel: Abheben mittels `limitedWithdraw` von mehreren Konto solange bis ein Konto genug Geld hat

```
limitedWithdrawMany :: [Account] -> Int -> STM ()
limitedWithdrawMany [] _ = retry
limitedWithdrawMany (acc:rest) amount =
    limitedWithdraw acc amount `orElse`
    limitedWithdrawMany rest amount
```

- `orElse :: STM a -> STM a -> STM a`
  - `orElse t1 t2` führt zuerst `t1` aus
  - Falls `t1` erfolgreich, so auch `orElse t1 t2`
  - Falls `t1` abbricht (durch Aufruf von `retry`) wird `t2` ausgeführt
  - Falls `t2` abbricht bricht auch die gesamte Transaktion ab, d.h. sie wird zu einem späteren Zeitpunkt nochmals ausgeführt

Zusammenfassung der STM API

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

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

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

Ausführungsmodell für STM

- `atomically t` wird *optimistisch* ausgeführt, ohne Locks
- `writeTVar v x` schreibt in ein *Log*, nicht in den Speicher
- `readTVar v` liest erst aus dem Log und nur bei Misserfolg aus dem Speicher
- Falls `readTVar v` den Wert nicht im Log findet wird der im Speicher gelesene Wert auch im Log vermerkt
- Am Ende einer Transaktion erfolgt die *Validierung*
  - Muss atomar erfolgen (Locks)
  - Validierung erfolgreich falls die Werte aller Leseoperation im Log zum Hauptspeicherinhalt passen
- Bei erfolgreicher Validierung: aktualisiere Hauptspeicher
- `retry` benutzt Log um Herauszufinden wann eine Neuausführung sich lohnt

Beispiel: Ausführung von STM

Thread A: `transfer acc1 acc2 50`

Beispiel: Ausführung von STM

Thread A: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
```

Log A

acc2: read 50

Beispiel: Ausführung von STM

Thread A: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (50 + 50)
```

Log A

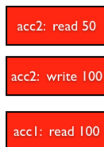
acc2: read 50

acc2: write 100

Beispiel: Ausführung von STM

Thread A: `transfer acc1 acc2 50`

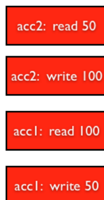
```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (50 + 50)
                bal1 <- readTVar acc1
```



Beispiel: Ausführung von STM

Thread A: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (50 + 50)
                bal1 <- readTVar acc1
                writeTVar acc1 (100 - 50))
```



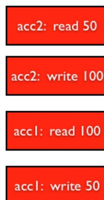
Beispiel: Ausführung von STM

Thread A: transfer acc1 acc2 50

Thread B: transfer acc1 acc2 50

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (50 + 50)
                bal1 <- readTVar acc1
                writeTVar acc1 (100 - 50))
```

Log A

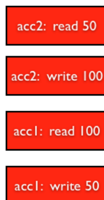


Beispiel: Ausführung von STM

Thread A: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (50 + 50)
                bal1 <- readTVar acc1
                writeTVar acc1 (100 - 50))
```

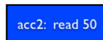
Log A



Thread B: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
```

Log B



Beispiel: Ausführung von STM

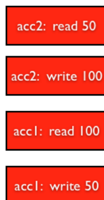
Thread A: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (50 + 50)
                bal1 <- readTVar acc1
                writeTVar acc1 (100 - 50))
```

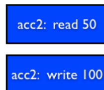
Thread B: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (50 + 50)
```

Log A



Log B



Beispiel: Ausführung von STM

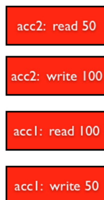
Thread A: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

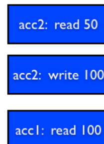
Thread B: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
```

Log A



Log B



Beispiel: Ausführung von STM

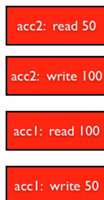
Thread A: transfer acc1 acc2 50

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Thread B: transfer acc1 acc2 50

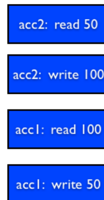
```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Log A



Validation

Log B



Beispiel: Ausführung von STM

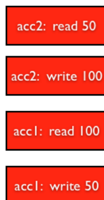
Thread A: `transfer acc1 acc2 50`

```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Thread B: `transfer acc1 acc2 50`

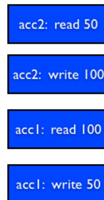
```
atomically (do bal2 <- readTVar acc2
               writeTVar acc2 (50 + 50)
               bal1 <- readTVar acc1
               writeTVar acc1 (100 - 50))
```

Log A

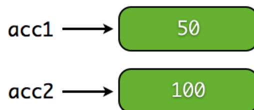


Commit

Log B



Beispiel: Ausführung von STM



Thread A

Thread B: `transfer acc1 acc2 50`

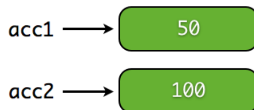
```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (50 + 50)
                bal1 <- readTVar acc1
                writeTVar acc1 (100 - 50))
```

Log B

- acc2: read 50
- acc2: write 100
- acc1: read 100
- acc1: write 50

Validation schlägt fehl!

Beispiel: Ausführung von STM



Thread A

Thread B: `transfer acc1 acc2 50`

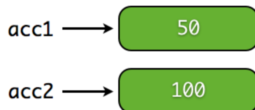
```
atomically (do  bal2 <- readTVar acc2
```

Rollback

Log B

acc2: read 100

Beispiel: Ausführung von STM



Thread A

Thread B: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (100 + 50)
```

Log B

acc2: read 100

acc2: write 150

Beispiel: Ausführung von STM

Thread A

Thread B: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (100 + 50)
                bal1 <- readTVar acc1
```

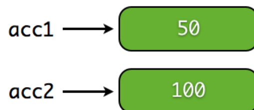
Log B

acc2: read 100

acc2: write 150

acc1: read 50

Beispiel: Ausführung von STM



Thread A

Thread B: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (100 + 50)
                bal1 <- readTVar acc1
                writeTVar acc1 (50 - 50))
```

Log B

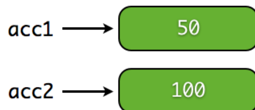
acc2: read 100

acc2: write 150

acc1: read 50

acc1: write 0

Beispiel: Ausführung von STM



Thread A

Thread B: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (100 + 50)
                bal1 <- readTVar acc1
                writeTVar acc1 (50 - 50))
```

Log B

- acc2: read 100
- acc2: write 150
- acc1: read 50
- acc1: write 0

Validation

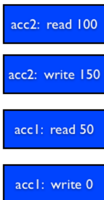
Beispiel: Ausführung von STM

Thread A

Thread B: `transfer acc1 acc2 50`

```
atomically (do  bal2 <- readTVar acc2
                writeTVar acc2 (100 + 50)
                bal1 <- readTVar acc1
                writeTVar acc1 (50 - 50))
```

Log B



Commit

Typsystem verhindert STM-Katastrophen!

- IO-Aktionen sind nicht beliebig wiederholbar
- Typsystem verhindert Ausführung von IO-Aktionen innerhalb einer STM-Transaktion

```haskell
launchMissiles :: IO ()
launchMissiles = -- ...

bad xv yv =
    atomically (do x <- readTVar xv
                   y <- readTVar yv
                   when (x > y) launchMissiles)
```

```
$ ghc Bad.hs
Bad.hs:12:25:
    Couldn't match type ‘IO’ with ‘STM’
    Expected type: STM ()
      Actual type: IO ()
```

Threads in Haskell

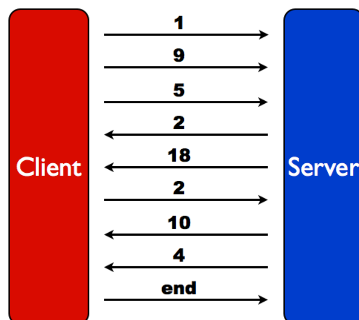
- Threads sind extrem billig
  - `forkIO : IO () -> IO ThreadId`
  - Green Threads
  - Platzbedarf $<$ 100 Bytes + Stack
  - Mehrere Million Threads auf diesem Laptop keine Problem
- Runtimesystem implementiert blockierende Aufrufe intern asynchron (z.B. mit `epoll` unter Linux)
- Interoperabilität mit nativem Code
  - Runtimesystem führt nativen Code in gesonderten OS-Threads aus
  - `forkOS ::  IO () -> IO ThreadId`
    - erstellt einen Haskell-Thread, dessen nativen Aufrufe immer im selben OS-Thread laufen
    - Wichtig für bestimmt Bibliotheken und "thread-local State"

Vergleich mit node.js

- node.js: Javascript Framework für serverseitige Netzwerkanwendungen
- Bekannt für gute Performance
- Asynchrone Schnittstelle zum Lesen von Sockets
  - "inversion of control"
  - Unbequem zum Programmieren
    - Logik für mehrere Clients vermischt
    - Zustand des Kontrollflusses muss explizit in globalen Variablen abgelegt werden
- Ausnutzung mehrerer Kerne nur durch mehrere Prozesse möglich
  - Benutzung von in-memory Datenstrukturen über mehrere Kerne hinweg schwierig

Benchmark Haskell vs. node.js

- Einfacher Server zum Verdoppeln von Zahlen
- Client schickt eine Zahl, Server schickt das Doppelte der Zahl zurück
- Client kann mehrere Zahlen schicken bevor der Server eine Antwort schickt
- Beispiel:



Haskell Server mit Conduits

- Conduits
  - Lösung für das Streaming Problem
    - Konstanter Speicher
    - Prompte Freigabe von Resourcen
  - Prinzip: Unix-Pipes

```
main =
    do let settings = serverSettings 44444 HostAny
       runTCPServer settings doubleApp
    where
      doubleApp x =
          appSource x $=
          (CB.lines =$= processLine) $$
          appSink x
```

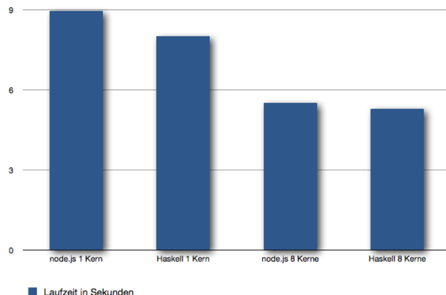
Die processLine Funktion

- `await`: Wartet auf neue Daten, *blockiert* falls keine da.
- `sendLine`: Erzeugt eine Ausgabe

```
processLine :: Monad m
            => Conduit BS.ByteString m BS.ByteString
processLine =
    do mBs <- await -- blockiert bis Daten da
       case mBs of
         Nothing -> return () -- EOF
         Just bs | bs == "end" -> sendLine "Thank you"
                 | otherwise ->
                   let l = case parseInt bs of
                             Just i -> show (2 * i)
                             Nothing -> "not an int"
                   in do sendLine l
                         processLine
```

Benchmark-Ergebnisse

- 5000 parallele Clients
- 10 Anfragen pro Client



Resourcen: Parallelität & Nebenläufigkeit

- Folien und Material zum Vortrag: `http://factisresearch.com/parallel2013/`
- Parallel and Concurrent Programming in Haskell, *Simon Marlow*
  - `http://community.haskell.org/~simonmar/par-tutorial.pdf`
  - Buch erscheint in Kürze bei O'Reilly
- Beautiful concurrency, *Simon Peyton Jones*
  - appeared in "Beautiful code", Herausgeber Greg Wilson, O'Reilly 2007
  - `http://research.microsoft.com/pubs/74063/beautiful.pdf`
- Deterministic Parallel Programming with Haskell, *Duncan Coutts* und *Andres Löh*
  - Computing in Science & Engineering, Volume 14, Issue 6, Dez. 2012
  - `http://www.well-typed.com/blog/aux/files/`

Resourcen: Haskell und funktionale Programmierung allgemein

- Haskell Homepage: `http://haskell.org`
- School of Haskell: `https://www.fpcomplete.com/`
- Learn You a Haskell for Great Good, *Miran Lipovača*, No Starch Press, 2011
  - `http://learnyouahaskell.com/`
- Real World Haskell, *Bryan O'Sullivan*, *Don Stewart* und *John Goerzen*, O'Reilly 2008
  - `http://book.realworldhaskell.org/`
- Programming in Haskell, *Graham Hutton*, Cambridge University Press, 2007
- Blog *Funktionale Programmierung*: `http://funktionale-programmierung.de/`

Fazit & Zusammenfassung

- Haskell bietet viel bzgl. Nebenläufigkeit und Parallelität
  - Deterministischer Parallelismus
  - Datenparallelismus
  - GPU-Programmierung
  - STM
  - Leichtgewichtige Threads
- Wichtige Eigenschaften von Haskell
  - Bewusster Umgang mit Seiteneffekten
  - Wenig globaler Zustand
  - Datenabhängigkeiten oft explizit
  - Ausdrucksstarkes Typsystem: Wenn das Programm kompiliert funktioniert es auch!
  - Kurzer, prägnanter und lesbarer Code
  - Einfaches Abstrahieren
- Haskell ist "World's finest imperative programming language"
- Haskell ist performant