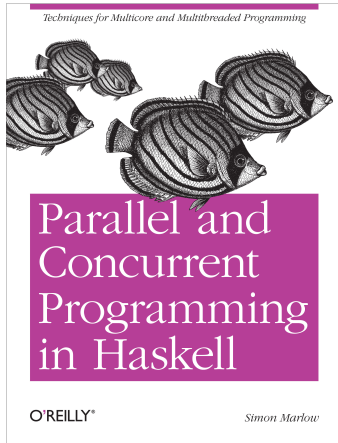
Fortgeschrittene Funktionale Programmierung in Haskell

Jonas Betzendahl
Stefan Dresselhaus

Vorlesung 8: *Parallelism & Concurrency*
Stand: 3. Juni 2016

Leseempfehlung

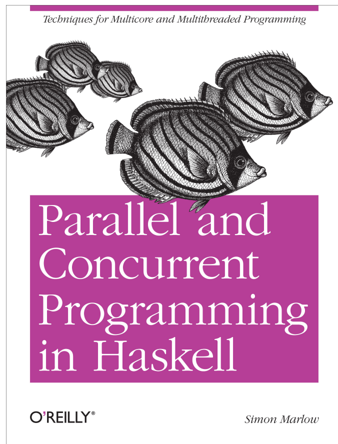


Wunderbares Buch „*Parallel and Concurrent Programming in Haskell*" von Simon Marlow (Microsoft Research, Facebook), auch kostenlos im Internet verfügbar[1].

[1] http://chimera.labs.oreilly.com/books/1230000000929

[2] http://www.haskellcast.com/episode/004-simon-marlow-on-parallelism-and-concurrency/

Leseempfehlung



Wunderbares Buch „*Parallel and Concurrent Programming in Haskell*“ von Simon Marlow (Microsoft Research, Facebook), auch kostenlos im Internet verfügbar[1].

Es gibt außerdem eine Episode des *Haskellcast*[2] mit ihm zum gleichen Thema.

[1] http://chimera.labs.oreilly.com/books/1230000000929
[2] http://www.haskellcast.com/episode/004-simon-marlow-on-parallelism-and-concurrency/

Motivation (I)

Moore's Law: [3]

Microprocessor Transistor Counts 1971-2011 & Moore's Law

curve shows transistor count doubling every two years

Transistor count

Date of introduction

---

[3] By Wgsimon - Own work, CC BY-SA 3.0
https://commons.wikimedia.org/w/index.php?curid=15193542

Motivation (II)

Free Lunch is over!
Herb Sutter (2005)

Motivation (II)

Free Lunch is over!
Herb Sutter (2005)

Die Hardware unserer Computer wird seit mehreren Jahren schon schneller breiter (*mehr* Kerne) als tiefer (*schnellere* Kerne).

Um technischen Fortschritt voll auszunutzen ist es also essentiell, gute Werkzeuge für einfache und effiziente Parallelisierung bereit zu stellen.

Parallelism vs. Concurrency

Sowohl Parallelismus *(Parallelism)* als auch Nebenläufigkeit *(Concurrency)* sind Arten mehrere Dinge „gleichzeitig“ zu machen. Es sind aber durchaus unterschiedliche Konzepte.

Parallelism vs. Concurrency

Sowohl Parallelismus *(Parallelism)* als auch Nebenläufigkeit *(Concurrency)* sind Arten mehrere Dinge „gleichzeitig“ zu machen. Es sind aber durchaus unterschiedliche Konzepte.

Parallel laufende Programme benutzen mehr Hardware (z.B. mehrere CPUs) um schneller Berechnungen durchführen zu können.

(deterministisch)

Parallelism vs. Concurrency

Sowohl Parallelismus *(Parallelism)* als auch Nebenläufigkeit *(Concurrency)* sind Arten mehrere Dinge „gleichzeitig“ zu machen. Es sind aber durchaus unterschiedliche Konzepte.

Parallel laufende Programme benutzen mehr Hardware (z.B. mehrere CPUs) um schneller Berechnungen durchführen zu können.

(deterministisch)

Nebenläufige Programme haben mehrere „threads of control“ oft um mit mehreren Externa (z.B. dem User, einer Datenbank, ...) zu Interagieren (das bedeutet `IO`).

(nichtdeterministisch)

(WH)NF:

Im Themenbereich Parallelism wird oft darüber gesprochen, wann und „wie weit“ Ausdrücke ausgewertet werden. Es gibt dafür zwei wichtige Vokabeln: *Normal Form* und *Weak Head Normal Form*.

(WH)NF:

Im Themenbereich Parallelism wird oft darüber gesprochen, wann und „wie weit“ Ausdrücke ausgewertet werden. Es gibt dafür zwei wichtige Vokabeln: *Normal Form* und *Weak Head Normal Form*.

Die Normal Form eines Ausdrucks ist der vollständig berechnete Ausdruck. Es gibt keine Unterausdrücke, die weiter ausgewertet werden könnten.

(WH)NF:

Im Themenbereich Parallelism wird oft darüber gesprochen, wann und „wie weit" Ausdrücke ausgewertet werden. Es gibt dafür zwei wichtige Vokabeln: *Normal Form* und *Weak Head Normal Form*.

Die Normal Form eines Ausdrucks ist der vollständig berechnete Ausdruck. Es gibt keine Unterausdrücke, die weiter ausgewertet werden könnten.

Die Weak Head Normal Form eines Ausdrucks ist der Ausdruck, evaluiert zum äußersten Konstruktor oder zur äußersten λ-Abstraktion (dem „head"). Unterausdrücke können berechnet sein oder auch nicht. Ergo ist jeder Ausdruck in Normal Form auch automatisch in Weak Head Normal Form.

(WH)NF-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

(WH)NF-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

(WH)NF-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

$\Rightarrow$ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

(WH)NF-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

⇒ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```
\x -> 2 + 2
```

(WH)NF-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```haskell
(1337, "Hello World!")
```

$\Rightarrow$ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```haskell
\x -> 2 + 2
```

$\Rightarrow$ **WHNF**! Der *head* ist eine λ-Abstraktion.

(WH)NF-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

$\Rightarrow$ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```
\x -> 2 + 2
```

$\Rightarrow$ **WHNF**! Der *head* ist eine λ-Abstraktion.

```
'f' : ("oo" ++ "bar")
```

(WH)NF-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```haskell
(1337, "Hello World!")
```

$\Rightarrow$ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```haskell
\x -> 2 + 2
```

$\Rightarrow$ **WHNF**! Der *head* ist eine λ-Abstraktion.

```haskell
'f' : ("oo" ++ "bar")
```

$\Rightarrow$ **WHNF**! Der *head* ist der Konstruktor `(:)`.

(WH)NF-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

⇒ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```
\x -> 2 + 2
```

⇒ **WHNF**! Der *head* ist eine λ-Abstraktion.

```
'f' : ("oo" ++ "bar")
```

⇒ **WHNF**! Der *head* ist der Konstruktor `(:)`.

```
(\x -> x + 1) 2
```

(WH)NF-Quiz:

Sind diese Ausdrücke in **NF** oder **WHNF**? Wenn ja welche davon?

```
(1337, "Hello World!")
```

$\Rightarrow$ **NF** und **WHNF**! Der komplette Ausdruck ist evaluiert.

```
\x -> 2 + 2
```

$\Rightarrow$ **WHNF**! Der *head* ist eine λ-Abstraktion.

```
'f' : ("oo" ++ "bar")
```

$\Rightarrow$ **WHNF**! Der *head* ist der Konstruktor `(:)`.

```
(\x -> x + 1) 2
```

$\Rightarrow$ Weder noch! Äußerster Part ist Anwendung der Funktion.

Optionen für den GHC

Um Programme in Haskell parallel ausführen zu können, müssen sie wie folgt kompiliert werden:

```
$ ghc --make -rtsopts -threaded Main.hs
```

[4] oder z.B. unter folgender Adresse:
cheatography.com/nash/cheat-sheets/ghc-and-rts-options/

Optionen für den GHC

Um Programme in `Haskell` parallel ausführen zu können, müssen sie wie folgt kompiliert werden:

```
$ ghc --make -rtsopts -threaded Main.hs
```

Danach können sie auch mit RTS (Run Time System) - Optionen wie z.B. diesen hier ausgeführt werden:

```
$ ./Main +RTS -N2 -s -RTS
```

[4] oder z.B. unter folgender Adresse:
`cheatography.com/nash/cheat-sheets/ghc-and-rts-options/`

Optionen für den GHC

Um Programme in Haskell parallel ausführen zu können, müssen sie wie folgt kompiliert werden:

```
$ ghc --make -rtsopts -threaded Main.hs
```

Danach können sie auch mit RTS (Run Time System) - Optionen wie z.B. diesen hier ausgeführt werden:

```
$ ./Main +RTS -N2 -s -RTS
```

Dokumentation für bestimmte Features findet sich leicht via beliebiger Suchmaschine[4].

---

[4] oder z.B. unter folgender Adresse:
cheatography.com/nash/cheat-sheets/ghc-and-rts-options/

Parallelism

Eval und Strategy

Das Modul `Control.Parallel.Strategies` stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung.

```
type Strategy a = a -> Eval a
```

Strategien sind Funktionen, die beschreiben, *wie* bestimmte Ausdrücke ausgewertet werden sollen. Dazu gleich mehr.

Eval und Strategy

Das Modul `Control.Parallel.Strategies` stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung.

```
type Strategy a = a -> Eval a
```

Strategien sind Funktionen, die beschreiben, *wie* bestimmte Ausdrücke ausgewertet werden sollen. Dazu gleich mehr.

Desweiteren stellt es die Operation `runEval` bereit, die die monadischen Berechnungen ausführt und das Ergebnis zurück gibt.

```
runEval :: Eval a -> a
```

Eval und Strategy

Das Modul `Control.Parallel.Strategies` stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung.

```
type Strategy a = a -> Eval a
```

Strategien sind Funktionen, die beschreiben, *wie* bestimmte Ausdrücke ausgewertet werden sollen. Dazu gleich mehr.

Desweiteren stellt es die Operation `runEval` bereit, die die monadischen Berechnungen ausführt und das Ergebnis zurück gibt.

```
runEval :: Eval a -> a
```

Wohlgemerkt: `runEval` ist *pur!*
Wir müssen nicht gleichzeitig auch in der `IO`-Monade sein.

`rpar` und `rseq`

`rpar` ist die Strategie, die ihr Argument parallel auswertet und währenddessen das Programm weiter laufen lässt.

`rpar` und `rseq`

`rpar` ist die Strategie, die ihr Argument parallel auswertet und währenddessen das Programm weiter laufen lässt.

`rseq` ist die Strategie, die auf das Ergebnis ihres Argumentes wartet und erst dann mit dem Programm weiter macht.

`rpar` und `rseq`

`rpar` ist die Strategie, die ihr Argument parallel auswertet und währenddessen das Programm weiter laufen lässt.

`rseq` ist die Strategie, die auf das Ergebnis ihres Argumentes wartet und erst dann mit dem Programm weiter macht.

Ein paar Hinweise:

- Ausgewertet wird jeweils zur WHNF (wenn nichts anderes angegeben wurde).
- Wird `rpar` ein bereits evaluierter Ausdruck übergeben, passiert nichts, weil es keine Arbeit parallel auszuführen gibt.

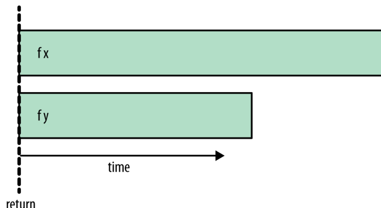
Wartezeiten (I):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

Wartezeiten (I):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

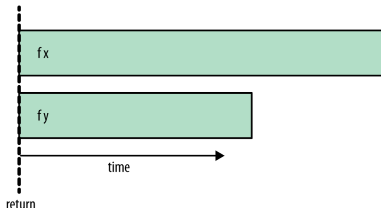
```
-- don't wait for evaluation
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    pure (a,b)
```



Wartezeiten (I):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- don't wait for evaluation
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    pure (a,b)
```

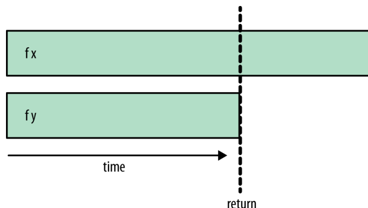


Hier passiert das `pure` sofort. Der Rest des Programmes läuft weiter, während `(f x)` und `(f y)` (parallel) ausgewertet werden.

Wartezeiten (II):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

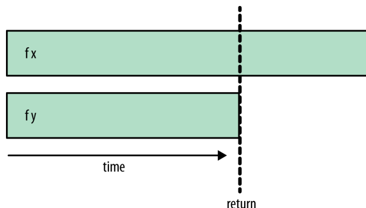
```
-- wait for (f y)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y) -- wait
    pure (a,b)
```



Wartezeiten (II):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- wait for (f y)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y) -- wait
    pure (a,b)
```

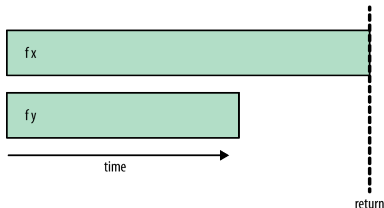


Hier werden `(f x)` und `(f y)` ebenfalls ausgewertet, allerdings wird mit `pure` gewartet, bis `(f y)` zu Ende evaluiert wurde.

Wartezeiten (III):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

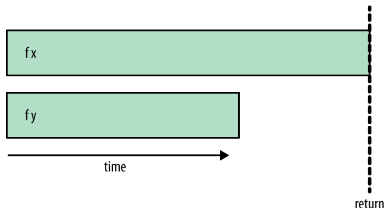
```
-- wait for (f y) and (f x)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y) -- wait
    rseq a          -- wait
    pure (a,b)
```



Wartezeiten (III):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- wait for (f y) and (f x)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y) -- wait
    rseq a          -- wait
    pure (a,b)
```

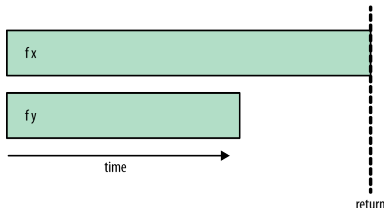


In diesem Code wird sowohl auf `(f x)` als auch auf `(f y)` gewartet, bevor etwas zurück gegeben wird.

Wartezeiten (3):

Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

```
-- perhaps more readable:
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    rseq a          -- wait
    rseq b          -- wait
    pure (a,b)
```



In diesem Code wird sowohl auf `(f x)` als auch auf `(f y)` gewartet, bevor etwas zurück gegeben wird.

Beispiel: Sudoku

Wir nehmen an, wir haben bereits die folgende Funktion:

```
solve :: String -> Grid
solve = undefined -- magic goes here
```

Beispiel: Sudoku

Wir nehmen an, wir haben bereits die folgende Funktion:

```
solve :: String -> Grid
solve = undefined -- magic goes here
```

Dann sähe ein mögliches (sequentielles) Programm so aus:

```
main :: IO ()
main = do
    [f]  <- getArgs
    file <- readFile f

    let puzzles   = lines file
    let solutions = map solve puzzles

    print (length (filter isJust solutions))
```

Parallelisierung von Sudoku

Nun wollen wir die Lösungen parallel auf zwei Kernen berechnen:

```
main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f

  let puzzles = lines file
  let (as,bs) = splitAt (length puzzles `div` 2) puzzles
  let solutions = runEval $ do
                    as' <- rpar (force (map solve as))
                    bs' <- rpar (force (map solve bs))
                    rseq as'
                    rseq bs'
                    pure (as' ++ bs')

  print (length (filter isJust solutions))
```

force und seine Bedeutung

Was tut die Funktion `force` und warum wird sie hier benötigt?

```
force :: NFData a => a -> a
```

`force` und seine Bedeutung

Was tut die Funktion `force` und warum wird sie hier benötigt?

```haskell
force :: NFData a => a -> a
```

`rpar` evaluiert nur zur **WHNF**, nicht zur vollen Lösung. Dies ist ein häufiger Fehler bei Parallelisierung von `Haskell`-Programmen. Die Lösung ist, die komplette Evaluation zu erzwingen.

`force` und seine Bedeutung

Was tut die Funktion `force` und warum wird sie hier benötigt?

```
force :: NFData a => a -> a
```

`rpar` evaluiert nur zur **WHNF**, nicht zur vollen Lösung. Dies ist ein häufiger Fehler bei Parallelisierung von `Haskell`-Programmen. Die Lösung ist, die komplette Evaluation zu erzwingen.

Allerdings muss hierbei bedacht werden, dass `force` $\mathcal{O}(n)$ Zeit benötigt, um die Datenstruktur komplett zu evaluieren.

Die `NFData`-Typklasse

Diese Typklasse umfasst alle Typen, die zu einer Normalform ausgewertet werden können (keine Funktionstypen).

Die NFData-Typklasse

Diese Typklasse umfasst alle Typen, die zu einer Normalform ausgewertet werden können (keine Funktionstypen).

```
class NFData a where
  rnf :: a -> ()
  rnf x = x `seq` ()
```

```
-- Zur Erinnerung:
seq :: a -> b -> b
```

Die NFData-Typklasse

Diese Typklasse umfasst alle Typen, die zu einer Normalform ausgewertet werden können (keine Funktionstypen).

```
class NFData a where
  rnf :: a -> ()
  rnf x = x `seq` ()
```

```
-- Zur Erinnerung:
seq :: a -> b -> b
```

`rnf` bringt Standardimplementation mit, dies erleichtert Instanzen von simplen Datentypen ohne Substrukturen. Instanzen von Typen *mit* Substrukturen nutzen rekursive Aufrufe von `rnf` und `seq`:

```
data Tree a = Empty
            | Branch (Tree a) a (Tree a)

instance NFData a => NFData (Tree a) where
  rnf Empty            = ()
  rnf (Branch l a r) = rnf l `seq` rnf a `seq` rnf r
```

Beispiel, fortgesetzt

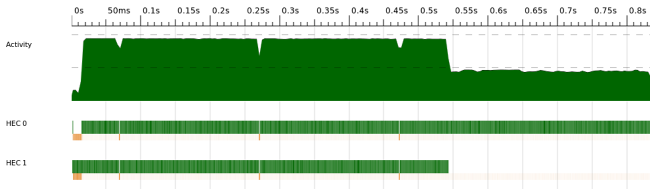
```
main :: IO ()
main = do
  [f]  <- getArgs
  file <- readFile f
  let puzzles = lines file
  let (as,bs) = splitAt (length puzzles `div` 2) puzzles
  let solutions = runEval $ do
                    as' <- rpar (force (map solve as))
                    bs' <- rpar (force (map solve bs))
                    rseq as'
                    rseq bs'
                    pure (as' ++ bs')
  print (length (filter isJust solutions))
```

Beispiel, fortgesetzt

```
main :: IO ()
main = do
  [f]  <- getArgs
  file <- readFile f
  let puzzles = lines file
  let (as,bs) = splitAt (length puzzles `div` 2) puzzles
  let solutions = runEval $ do
                  as' <- rpar (force (map solve as))
                  bs' <- rpar (force (map solve bs))
                  rseq as'
                  rseq bs'
                  pure (as' ++ bs')
  print (length (filter isJust solutions))
```

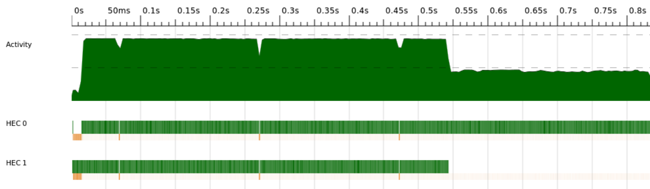
Wenn wir diesen Code auf zwei Kernen laufen lassen, bekommen wir einen Speedup in Wall-clock-time, allerdings „nur“ um einen Faktor von $\sim 1,5$.

Analyse mit ThreadScope (I)



Wir bemerken: Unsere parallelen Berechnungen sind ungleich groß. Eine benötigt deutlich länger als die andere. Dies ist ein häufiges Problem bei Parallelisierung mit *static partitioning*.

Analyse mit ThreadScope (I)



Wir bemerken: Unsere parallelen Berechnungen sind ungleich groß. Eine benötigt deutlich länger als die andere. Dies ist ein häufiges Problem bei Parallelisierung mit *static partitioning*.

Außerdem sind wir so durch die Anzahl der Chunks beschränkt. Werden nur zwei Chunks parallel evaluiert, können wir keinen Speedup > 2 erreichen, egal wie viele Kerne wir einsetzen.

Sparks

Ein Weg, hier weiter zu optimieren, ist, von *static partitioning* auf *dynamic partitioning* wechseln.

Anstatt von Hand ein paar große Chunks anzugeben geben wir viele kleine Chunks an, die dann zur Laufzeit automatisch auf die Prozessorkerne aufgeteilt werden.

Sparks

Ein Weg, hier weiter zu optimieren, ist, von *static partitioning* auf *dynamic partitioning* wechseln.

Anstatt von Hand ein paar große Chunks anzugeben geben wir viele kleine Chunks an, die dann zur Laufzeit automatisch auf die Prozessorkerne aufgeteilt werden.

Es gibt ein Fachwort für dieses Konzept: *Spark*. Ein Spark ist ein noch nicht ausgewerteter Ausdruck in einer Queue, die vom RTS auf schlaue Weise parallel evaluiert werden können.

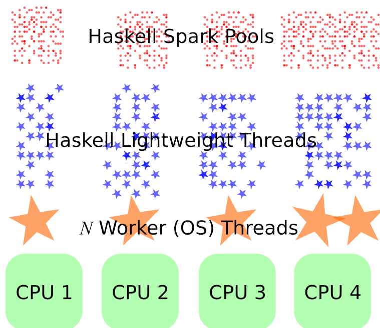
Sparks und Threads

Sparks sind *nicht* das gleiche wie `Haskell`-Threads, und diese wiederum sind etwas anderes als Betriebssystem-Threads.

Zum Vergleich: Ein etwas größeres Programm hätte vielleicht...

- mehrere Milliarden Sparks,
- ca. eine Million lightweight Haskell-Threads,
- ein Dutzend OS-Thread,
- auf sechs Kernen.

Eine Visualisierung von Don Stewart[5]:



[5] `https://stackoverflow.com/questions/958449`

`map` in der `Eval`-Monade

```
parMap :: (a -> b) -> [a] -> Eval [b]
parMap f []     = pure []
parMap f (a:as) = do b  <- rpar (f a)
                     bs <- parMap f as
                     pure (b:bs)
```

Nun wird für jede Funktionsanwendung `(f a)` ein *Spark* erstellt, alle arbeiten parallel und werden vom RTS gemanaged.

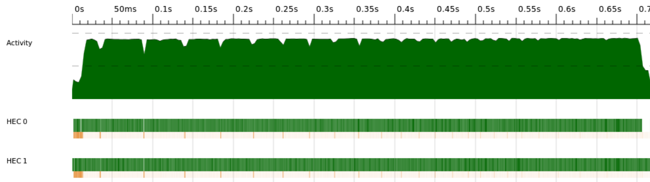
map in der Eval-Monade

```
parMap :: (a -> b) -> [a] -> Eval [b]
parMap f []     = pure []
parMap f (a:as) = do b  <- rpar (f a)
                     bs <- parMap f as
                     pure (b:bs)
```

Nun wird für jede Funktionsanwendung `(f a)` ein *Spark* erstellt, alle arbeiten parallel und werden vom RTS gemanaged.

```
main :: IO ()
main = do
  [f]  <- getArgs
  file <- readFile f
  let puzzles   = lines file
  let solutions = runEval (parMap solve puzzles)
  print (length (filter isJust solutions))
```

Analyse mit ThreadScope (II)



Der Speedup beträgt nun $\sim 1,8$. Den optimalen Wert von 2 zu erreichen ist in der Praxis unmöglich, da immer ein gewisser Overhead für das Management der Sparks entsteht.

Mehr Strategien

Es gibt natürlich noch mehr Inhalt zu Strategien. Andere Funktionen sind zum Beispiel:

```
using :: a -> Strategy a -> a
x `using` strat = runEval (strat x)

-- evaluiert nichts
r0 :: Strategy a

-- evaluiert komplett
rdeepseq :: NFData a => Strategy a

-- Kombination von zwei Strategien
dot :: Strategy a -> Strategy a -> Strategy a

parList :: Strategy a -> Strategy [a]
```

Mehr Strategien

Es gibt natürlich noch mehr Inhalt zu Strategien. Andere Funktionen sind zum Beispiel:

```
using :: a -> Strategy a -> a
x `using` strat = runEval (strat x)

-- evaluiert nichts
r0 :: Strategy a

-- evaluiert komplett
rdeepseq :: NFData a => Strategy a

-- Kombination von zwei Strategien
dot :: Strategy a -> Strategy a -> Strategy a

parList :: Strategy a -> Strategy [a]
```

All diese sind gute Werkzeuge um schnell Parallelismus zu erhalten. Außerdem können wir Parallelisierung und Algorithmus trennen.

Concurrency

Die Grundlagen

Wir beginnen mit der Funktion, die einen neuen *Thread* erstellt:

```
forkIO :: IO () -> IO ThreadId
```

Die Grundlagen

Wir beginnen mit der Funktion, die einen neuen *Thread* erstellt:

```
forkIO :: IO () -> IO ThreadId
```

Threads interagieren notwendigerweise mit der Welt, ergo ist die Berechnung, die wir übergeben vom Typ `IO ()`.

Der Rückgabewert - die `ThreadId` - kann später benutzt werden um z.B. den Thread vorzeitig abzubrechen oder ihm eine Exception zuzuschmeißen.

Beispiel: Output via Threads

```
import Control.Concurrent
import Control.Monad
import System.IO

main :: IO ()
main = do hSetBuffering stdout NoBuffering
          forkIO (replicateM_ 100000 (putChar 'A'))
          replicateM_ 100000 (putChar 'B')
```

Frage: Wie sieht hier der Output aus?

Beispiel: Output via Threads

```
import Control.Concurrent
import Control.Monad
import System.IO

main :: IO ()
main = do hSetBuffering stdout NoBuffering
          forkIO (replicateM_ 100000 (putChar 'A'))
          replicateM_ 100000 (putChar 'B')
```

Frage: Wie sieht hier der Output aus?

```
AAAAAAAAABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABAB....
```

`MVars` als Übergabemöglichkeit

Frage: Wie kriegen wir jetzt Ergebnisse aus nebenläufigen Berechnungen raus? Der Typ ist `IO ()`, das liefert nichts (interessantes) zurück.

`MVars` als Übergabemöglichkeit

Frage: Wie kriegen wir jetzt Ergebnisse aus nebenläufigen Berechnungen raus? Der Typ ist `IO ()`, das liefert nichts (interessantes) zurück.

Introducing: Mutable Variables, kurz `MVars`!

```
data MVar a  -- abstract

newEmptyMVar :: IO (MVar a)           -- create
newMVar      :: a -> IO (MVar a)      -- create with value
takeMVar     :: MVar a -> IO a        -- take (blocks)
putMVar      :: MVar a -> a -> IO () -- put  (blocks)

readMVar     :: MVar a -> IO a        -- atomically read
```

Ein Beispiel zu MVars

```
main :: IO ()
main = do m <- newEmptyMVar
          forkIO $ do putMVar m 'x'; putMVar m 'y'
          r <- takeMVar m
          print r
          r <- takeMVar m
          print r
```

Ein Beispiel zu MVars

```
main :: IO ()
main = do m <- newEmptyMVar
          forkIO $ do putMVar m 'x'; putMVar m 'y'
          r <- takeMVar m
          print r
          r <- takeMVar m
          print r
```

Wie wir sehen kann die gleiche `MVar` über Zeit mehrere Zustände annehmen und erfolgreich zur Kommunikation zwischen Threads benutzt werden.

Nutzung von `MVars`

Generell haben `MVars` drei Hauptaufgaben:

Nutzung von `MVars`

Generell haben `MVars` drei Hauptaufgaben:

- **Channel mit nur einem Slot**
  Eine `MVar` kann als Nachrichtenkanal zwischen Threads dienen, allerdings nur eine Nachricht auf einmal halten.

Nutzung von `MVars`

Generell haben `MVars` drei Hauptaufgaben:

- **Channel mit nur einem Slot**
  Eine `MVar` kann als Nachrichtenkanal zwischen Threads dienen, allerdings nur eine Nachricht auf einmal halten.
- **Behältnis für shared mutable state**
  In Concurrent Haskell brauchen oft mehrere Threads Zugriff auf einen shared state. Ein beliebtes Designpattern ist, das dieser State als normaler (immutable) `Haskell`-Datentyp repräsentiert und in einer `MVar` verpackt wird.

Nutzung von `MVars`

Generell haben `MVars` drei Hauptaufgaben:

- **Channel mit nur einem Slot**
  Eine `MVar` kann als Nachrichtenkanal zwischen Threads dienen, allerdings nur eine Nachricht auf einmal halten.
- **Behältnis für shared mutable state**
  In Concurrent Haskell brauchen oft mehrere Threads Zugriff auf einen shared state. Ein beliebtes Designpattern ist, das dieser State als normaler (immutable) `Haskell`-Datentyp repräsentiert und in einer `MVar` verpackt wird.
- **Baustein für kompliziertere Strukturen**

MVars und das RTS

Was passiert, wenn wir folgenden Code ausführen?

```
main :: IO ()
main = do m <- newEmptyMVar -- stays empty
          takeMVar m
```

MVars und das RTS

Was passiert, wenn wir folgenden Code ausführen?

```
main :: IO ()
main = do m <- newEmptyMVar -- stays empty
          takeMVar m
```

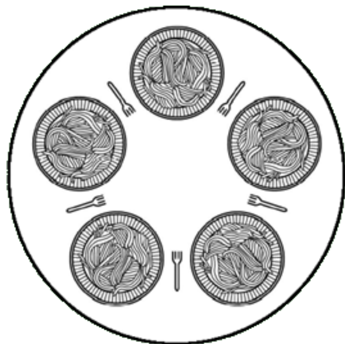
Wir bekommen eine Fehlermeldung, dass das Programm hängt, statt einfach nur ein hängendes Programm.

```
$ ./mvar3
mvar3: thread blocked indefinitely in an MVar operation
```

Deadlock Detection (I)

Threads und `MVars` sind Objekte auf dem Heap. Das `RTS` (i.e. der Garbage collector) durchläuft den Heap um alle lebendigen Objekte zu finden, angefangen bei den laufenden Threads und ihren Stacks.

Alles was so nicht erreichbar sind (z.B. ein Thread der auf eine `MVar` wartet, die nirgendwo sonst referenziert wird), blockiert und bekommt eine Exception geschmissen.



Ein Tisch aus dem Problem der „dining philosophers“[6]

[6] Quelle: `http://jason.mchu.com/SDP/`

Deadlock Detection (II)

Dieses Vorgang funktioniert allerdings nicht immer wie man zunächst denkt. Was passiert mit diesem Code?

```
main :: IO ()
main = do
  lock     <- newEmptyMVar
  complete <- newEmptyMVar
  forkIO $ takeMVar lock `finally` putMVar complete ()
  takeMVar complete
```

Deadlock Detection (II)

Dieses Vorgang funktioniert allerdings nicht immer wie man zunächst denkt. Was passiert mit diesem Code?

```
main :: IO ()
main = do
  lock     <- newEmptyMVar
  complete <- newEmptyMVar
  forkIO $ takeMVar lock `finally` putMVar complete ()
  takeMVar complete
```

Da nicht nur der geforkte Thread sondern auch der ursprüngliche gedeadlocked sind, wird hier die Fehlermeldung geprintet, statt die rettende Exception an das Kind zu senden.

Motivation: STM

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)[7]

[7] „The Future is Parallel, and the Future of Parallel is Declarative“: `https://www.youtube.com/watch?v=hlyQjK1qjw8`

Motivation: STM

Trotz aller Unterstützung durch das `RTS`:

Locks are absurdly hard to get right! (SPJ)[7]

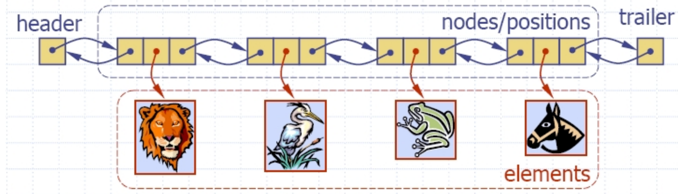
Beliebte Fehler:

- **Races** (vergessene Locks)
- **Deadlocks** (Locks in falscher Reihenfolge genommen)
- **Lost wakeups** (Conditional-Variable nicht bescheid gesagt)
- **Error Recovery** (Exceptionhandler müssen Locks freigeben und teilweise Ursprungszustand restaurieren)

[7] „The Future is Parallel, and the Future of Parallel is Declarative“: `https://www.youtube.com/watch?v=hlyQjK1qjw8`

Parallel Queue (I)

Angenommen wir möchten gerne eine Queue[8] parallel bearbeiten:



Problem: offensichtlich, race conditions etc.

[8] Quelle: www.math.bas.bg/~nkirov/2015/NETB201/slides/ch04/ch04.html

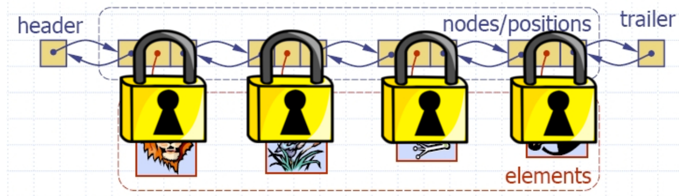
Parallel Queue (II)

Angenommen wir möchten gerne eine Queue parallel bearbeiten:

Problem: Nicht gerade nebenläufig.

Parallel Queue (III)

Angenommen wir möchten gerne eine Queue parallel bearbeiten:



Problem: Fehleranfälligkeit bei kleinen Listen

Introducing: `STM`

Software Transactional Memory stellt die Möglichkeit zur Verfügung, Berechnungen in atomaren Blöcken auszuführen. Das Interface ist aber das gleiche, wie in wie jeder anderen Monade.

Introducing: `STM`

Software Transactional Memory stellt die Möglichkeit zur Verfügung, Berechnungen in atomaren Blöcken auszuführen. Das Interface ist aber das gleiche, wie in wie jeder anderen Monade.

<u>Vorteile von atomarem `STM`:</u>

- Deadlocks werden unmöglich *weil es keine Locks mehr gibt*!
- Automatisierte error recovery. `STM` stellt den Ausgangszustand von selbst wieder her.
- `TVars` (Transaction Variables). Wie `MVars`, nur in der `STM`-Monade.

STM auf einen Blick

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

atomically :: STM a -> IO a

data TVar a                                 -- abstract

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

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

throwSTM   :: Exception e => e -> STM a
catchSTM   :: Exception e => STM a -> (e -> STM a) -> STM a
```

Ein Blick auf: atomically

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

Ein Blick auf: `atomically`

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

- *Kein* Sprachkonstrukt!
- Führt Berechnungen in `STM` in der echten Welt aus
- ...und zwar in einem Rutsch, ohne Unterbrechung!
- Stellt bei Fehlschlag Ursprungszustand wieder her.
- Deshalb: Kein `IO` in Transaktionen!
- Ebenfalls: Keine genesteten `atomically`s (Typen!)

Ein Blick auf: `retry`

```
retry :: STM a
```

Ein Blick auf: `retry`

```
retry :: STM a
```

- *Kein* Sprachkonstrukt!
- Rollt zurück und versucht die gleiche Transaktion erneut durchzuführen
- ... allerdings erst zu einem angebrachten Zeitpunkt. CPU wird nicht unnützerweise zur Heizung.

Ein Blick auf: orElse

```
orElse :: STM a -> STM a -> STM a
```

Ein Blick auf: `orElse`

```
orElse :: STM a -> STM a -> STM a
```

- *Kein* Sprachkonstrukt!
- `orElse a b` führt `b` aus, wenn `a` `retry` aufruft.
- Komposition von `STM`-Berechnungen:
  `>>=` ist `AND`; `orElse` ist `OR`

Ein Blick auf: `orElse`

```
orElse :: STM a -> STM a -> STM a
```

- *Kein* Sprachkonstrukt!
- `orElse a b` führt `b` aus, wenn `a` `retry` aufruft.
- Komposition von `STM`-Berechnungen:
  `>>=` ist `AND`; `orElse` ist `OR`

```
takeEitherTMVar :: TMVar a -> TMVar b -> STM (Either a b)
takeEitherTMVar ma mb =
  fmap Left (takeTMVar ma)
    `orElse`
  fmap Right (takeTMVar mb)
```

Beispiel: Banksoftware (I)

Stellen wir uns vor, wir wollen eine simplifizierte Bankensoftware schreiben, die in der Lage sein soll, Konten und Überweisungen zu repräsentieren.

Beispiel: Banksoftware (I)

Stellen wir uns vor, wir wollen eine simplifizierte Bankensoftware schreiben, die in der Lage sein soll, Konten und Überweisungen zu repräsentieren.

Eine naive Implementation wäre die folgende:

```haskell
type Account = IORef Integer

transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    fromVal <- readIORef from
    toVal   <- readIORef to
    writeIORef from (fromVal - amount)
    writeIORef to (toVal + amount)
```

Beispiel: Banksoftware (II)

Diese Implementation hätte jedoch in einem Nebenläufigen Setting einige Probleme. Man beachte folgende Zeile:

```
fromVal <- readIORef from
```

Beispiel: Banksoftware (II)

Diese Implementation hätte jedoch in einem Nebenläufigen Setting einige Probleme. Man beachte folgende Zeile:

```
fromVal <- readIORef from
```

Finden nun mehrere Aktionen gleichzeitig statt, so könnte es sein, dass mehrere Threads denselben Wert als `fromVal` lesen, bevor die jeweils andere Transaktion durchgeführt wurde.

Dies hätte zur Folge, dass später ein inkorrekter Kontostand berechnet und gesetzt würde.

Beispiel: Banksoftware (III)

Führen wir also Locks ein (durch Benutzung von `MVars`), um diese race condition zu verhindern.

```
type Account = MVar Integer

credit :: Integer -> Account -> IO ()
credit amount account = do
    current <- takeMVar account
    putMVar account (current + amount)

debit :: Integer -> Account -> IO ()
debit amount account = do
    current <- takeMVar account
    putMVar account (current - amount)
```

Beispiel: Banksoftware (IV)

In dieser Implementation sähe eine Funktion für Überweisungen etwa so aus:

```
transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Beispiel: Banksoftware (IV)

In dieser Implementation sähe eine Funktion für Überweisungen etwa so aus:

```
transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Diese verhindert, dass durch fehlerhaftes Zusammenspiel von Threads Geld geschaffen oder vernichtet wird.

Beispiel: Banksoftware (IV)

In dieser Implementation sähe eine Funktion für Überweisungen etwa so aus:

```
transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Diese verhindert, dass durch fehlerhaftes Zusammenspiel von Threads Geld geschaffen oder vernichtet wird.

Es existiert allerdings immer noch eine race condition: Der Thread, der die Überweisung ausführt, könnte direkt nach dem `debit`-Schritt von der CPU verdrängt werden und die Bankensoftware dadurch in einem inkonsistenten Zustand zurück lassen.

Beispiel: Banksoftware (V)

Wie sähe diese Software mit STM aus?

```
type Account = TVar Integer

credit :: Integer -> Account -> STM ()
credit amount account = do
    current <- readTVar account
    writeTVar account (current + amount)

debit :: Integer -> Account -> STM ()
debit amount account = do
    current <- readTVar account
    writeTVar account (current - amount)

transfer :: Integer -> Account -> Account -> STM ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Beispiel: Banksoftware (VI)

Vergleich zur Variante mit `MVars`:

```
type Account = MVar Integer

credit :: Integer -> Account -> IO ()
credit amount account = do
    current <- takeMVar account
    putMVar account (current + amount)

debit :: Integer -> Account -> IO ()
debit amount account = do
    current <- takeMVar account
    putMVar account (current - amount)

transfer :: Integer -> Account -> Account -> IO ()
transfer amount from to = do
    debit  amount from
    credit amount to
```

Beispiel: Banksoftware (VII)

Der Unterschied hier besteht hauptsächlich in den Rückgabetypen.

```
transfer :: Integer -> Account -> Account -> IO ()
```

Diese Funktion führt die Überweisung vollkommen in der echten Welt durch, mit allen möglichen Fehlern, die dabei auftreten können.

Beispiel: Banksoftware (VII)

Der Unterschied hier besteht hauptsächlich in den Rückgabetypen.

```
transfer :: Integer -> Account -> Account -> IO ()
```

Diese Funktion führt die Überweisung vollkommen in der echten Welt durch, mit allen möglichen Fehlern, die dabei auftreten können.

```
transfer :: Integer -> Account -> Account -> STM ()
```

Diese Funktion stellt uns nur eine Berechnung in der `STM`-Monade bereit, die wir später entweder direkt oder auch als Baustein einer größeren Transaktion ausführen können.

Der Vorteil ist, dass wir nur einen Weg haben, die Berechnung auszuführen:

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

`MVars` vs. `TVars`

Auch wenn `STM` einige Vorteile bietet, so gibt es dennoch einige gute Gründe, `MVars` in bestimmten Situationen einzusetzen:

- **Performance:** `STM` ist eine Abstraktion, und wie (fast) alle Abstraktionen hat es Laufzeitkosten.
- **Fairness:** Blockieren mehrere Threads auf einer `MVar`, werden sie garantiert FIFO wieder aufgeweckt. `STM` hingegen hat keine Garantie für Fairness.

Fragen?

Cartesian Closed Comic by Maria Kovalyova & Roman Cheplyaka
https://ro-che.info/ccc/19