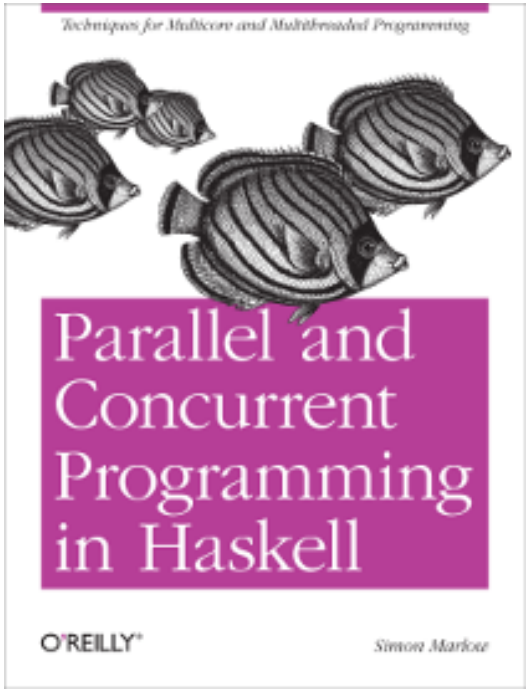
Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

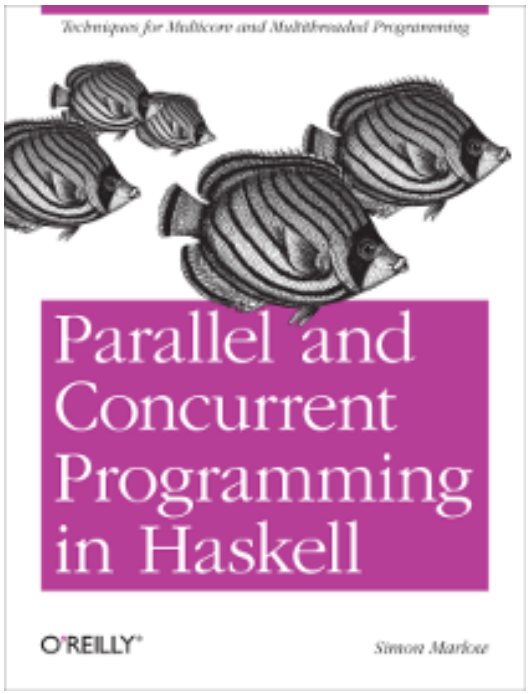
Übersicht I

Leseempfehlung:



Wunderbares Buch zum Thema von Simon Marlow. Gratis im Internet verfügbar, inklusive Beispielcode auf `Hackage`.

Leseempfehlung:



Wunderbares Buch zum Thema von Simon Marlow. Gratis im Internet verfügbar, inklusive Beispielcode auf `Hackage`.

S.a.: HaskellCast, Episode 4

Motivation

Free Lunch is over!

Herb Sutter (2005)

Free Lunch is over!

Herb Sutter (2005)

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

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.

Definitionen

Parallelism vs. Concurrency:

Beides ist ein Ausdruck von „Dinge gleichzeitig tun“; in der Programmierung haben sie aber grundverschiedene Bedeutungen.

Parallelism vs. Concurrency:

Beides ist ein Ausdruck von „Dinge gleichzeitig tun“; in der Programmierung haben sie aber grundverschiedene Bedeutungen.

Programme arbeiten *parallel*, wenn sie mehrere Prozessorkerne einsetzen, um schneller an die Antwort einer bestimmten Frage zu kommen.

Parallelism vs. Concurrency:

Beides ist ein Ausdruck von „Dinge gleichzeitig tun“; in der Programmierung haben sie aber grundverschiedene Bedeutungen.

Programme arbeiten *parallel*, wenn sie mehrere Prozessorkerne einsetzen, um schneller an die Antwort einer bestimmten Frage zu kommen.

Nebenläufige Programme hingegen haben mehrere „threads of control“. Oft dient das dazu, gleichzeitig mit mehreren externen Agenten (dem User, einer Datenbank, . . .) zu interagieren (das bedeutet `IO`).

Eine ähnliche Unterscheidung ist zwischen *deterministischem* und *nicht-deterministischem* Code.

Eine ähnliche Unterscheidung ist zwischen *deterministischem* und *nicht-deterministischem* Code.

Nebenläufiger Code ist zwangsläufig nicht-deterministisch (wegen der Interaktion mit Externa), paralleler Code kann jedoch oft deterministisch formuliert werden. Das erlaubt es z.B. das Programm auf einem Kern zu testen und dann einfach auf mehr Kerne zu schmeißen ohne, dass sich das Ergebnis ändert.

Eine ähnliche Unterscheidung ist zwischen *deterministischem* und *nicht-deterministischem* Code.

Nebenläufiger Code ist zwangsläufig nicht-deterministisch (wegen der Interaktion mit Externa), paralleler Code kann jedoch oft deterministisch formuliert werden. Das erlaubt es z.B. das Programm auf einem Kern zu testen und dann einfach auf mehr Kerne zu schmeißen ohne, dass sich das Ergebnis ändert.

Manchmal wird Concurrency an der falschen Stelle eingesetzt (wenn eigentlich Parallelism gewollt wäre). Das ist oft eine unkluge Entscheidung weil es den Determinismus des Programmes opfert und damit das Programm schwerer zu verstehen und zu optimieren macht.

Eine ähnliche Unterscheidung ist zwischen *deterministischem* und *nicht-deterministischem* Code.

Nebenläufiger Code ist zwangsläufig nicht-deterministisch (wegen der Interaktion mit Externa), paralleler Code kann jedoch oft deterministisch formuliert werden. Das erlaubt es z.B. das Programm auf einem Kern zu testen und dann einfach auf mehr Kerne zu schmeißen ohne, dass sich das Ergebnis ändert.

Manchmal wird Concurrency an der falschen Stelle eingesetzt (wenn eigentlich Parallelism gewollt wäre). Das ist oft eine unkluge Entscheidung weil es den Determinismus des Programmes opfert und damit das Programm schwerer zu verstehen und zu optimieren macht.

Es ist auch ganz natürlich, beides im gleichen Programm verwenden zu wollen.

(WH)NF:

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

(WH)NF:

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

Die **NF** eines Ausdrucks ist der gesamte Ausdruck, vollständig berechnet. Es gibt keine Unterausdrücke, die weiter ausgewertet werden könnten.

(WH)NF:

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

Die **NF** eines Ausdrucks ist der gesamte Ausdruck, vollständig berechnet. Es gibt keine Unterausdrücke, die weiter ausgewertet werden könnten.

Die **WHNF** 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 **NF** auch in **WHNF**.

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

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

(WH)NF Zuschauer-Wachzustands-Überprüfungs-Quiz:

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

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

(WH)NF Zuschauer-Wachzustands-Überprüfungs-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 Zuschauer-Wachzustands-Überprüfungs-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 Zuschauer-Wachzustands-Überprüfungs-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.

(WH)NF Zuschauer-Wachzustands-Überprüfungs-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")
```

(WH)NF Zuschauer-Wachzustands-Überprüfungs-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 `(:)`.

(WH)NF Zuschauer-Wachzustands-Überprüfungs-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 Zuschauer-Wachzustands-Überprüfungs-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
```

⇒ Weder noch! Äußerster Part ist Anwendung der Funktion.

Ein paar technische Feinheiten:

Ein paar technische Feinheiten:

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

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

Ein paar technische Feinheiten:

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.hs +RTS -N2 -s -RTS
```

Ein paar technische Feinheiten:

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.hs +RTS -N2 -s -RTS
```

Dokumentation findet sich leicht via beliebiger Suchmaschine.
Eine Kurzübersicht gibt es zum Beispiel unter
`cheatography.com/nash/cheat-sheets/ghc-and-rts-options/`

Parallelism

- Die `Eval`-Monade und Strategies
- Überblick: Die `Par`-Monade
- Überblick: Die `RePa`-Bibliothek und `Accelerate`

Parallelism

- ○ Die `Eval`-Monade und Strategies
- Überblick: Die `Par`-Monade
- Überblick: Die `RePa`-Bibliothek und `Accelerate`

Das Modul `Control.Parallel.Strategies` (aus dem Paket `parallel`) stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung, ...

Das Modul `Control.Parallel.Strategies` (aus dem Paket `parallel`) stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung, ...

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

Das Modul `Control.Parallel.Strategies` (aus dem Paket `parallel`) stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung, ...

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

... insbesondere die Strategies `rpar` und `rseq`. Dazu gleich mehr.

Das Modul `Control.Parallel.Strategies` (aus dem Paket `parallel`) stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung, ...

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

... insbesondere die Strategies `rpar` und `rseq`. Dazu gleich mehr.

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

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

Das Modul `Control.Parallel.Strategies` (aus dem Paket `parallel`) stellt uns die `Eval`-Monade und einige Funktionen vom Typ *Strategy* zur Verfügung, ...

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

... insbesondere die Strategies `rpar` und `rseq`. Dazu gleich mehr.

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

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

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

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

`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` 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.

Protips:

`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.

Protips:

- Ausgewertet wird jeweils zur WHNF (wenn nichts anderes angegeben wurde).

`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.

Protips:

- 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.

`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.

Protips:

- 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.

Sehen wir uns das mal *in action* an...

Ein Beispiel (1):

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

Ein Beispiel (1):

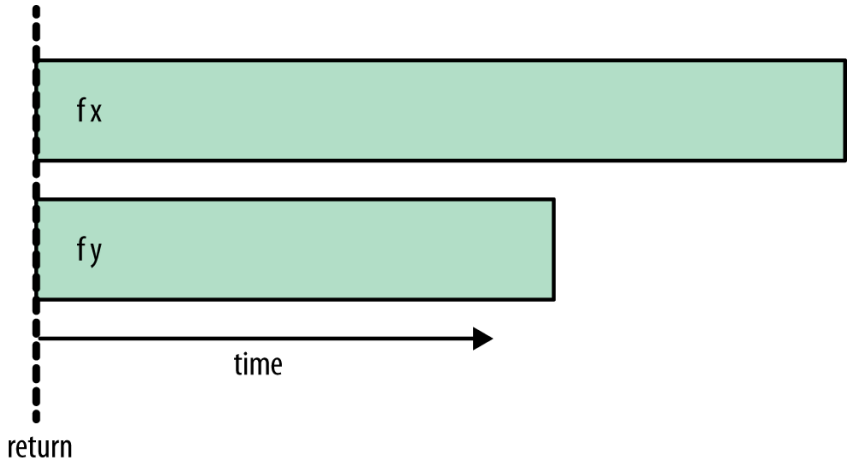
Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

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

Ein Beispiel (1):

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

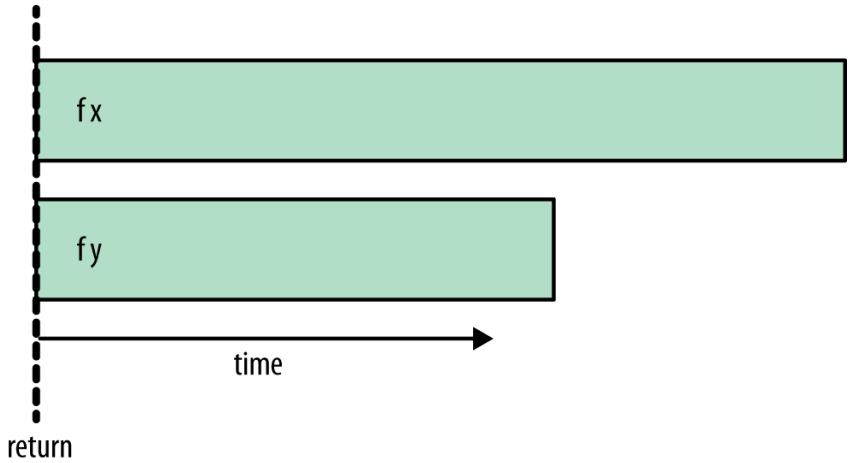
```
-- don't wait for evaluation
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    return (a,b)
```



Ein Beispiel (1):

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)
    return (a,b)
```



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

Ein Beispiel (2):

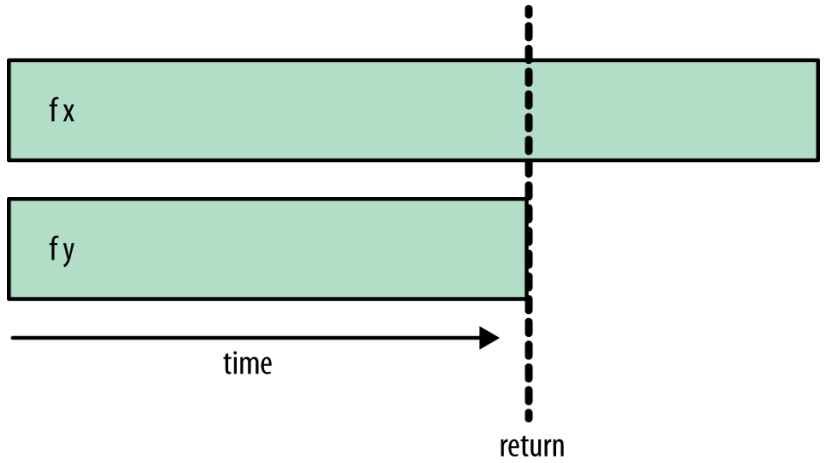
Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

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

Ein Beispiel (2):

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

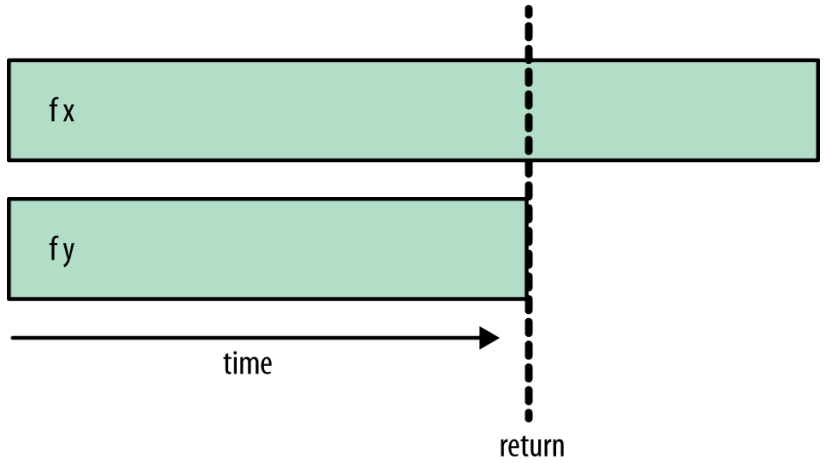
```
-- wait for (f y)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y)  -- wait
    return (a,b)
```



Ein Beispiel (2):

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
    return (a,b)
```



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

Ein Beispiel (3):

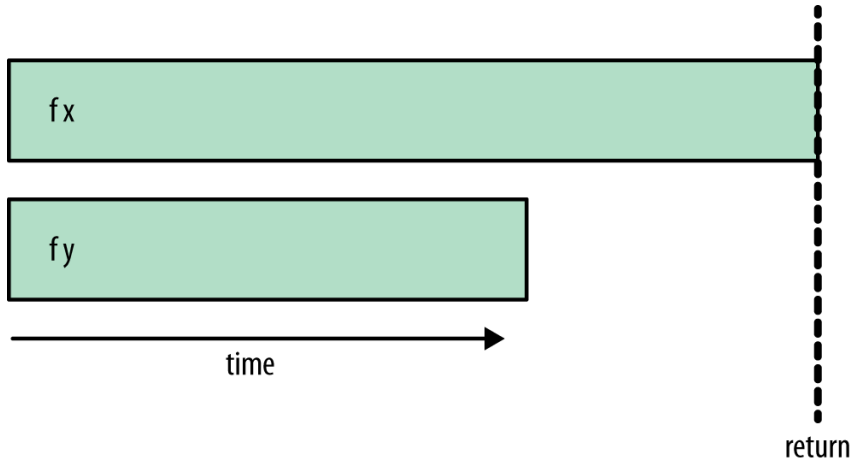
Wir wollen die Ausdrücke `(f x)` und `(f y)` mit der `Eval`-Monade parallel auswerten. O.B.d.A. benötigt `(f x)` mehr Zeit.

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

Ein Beispiel (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.

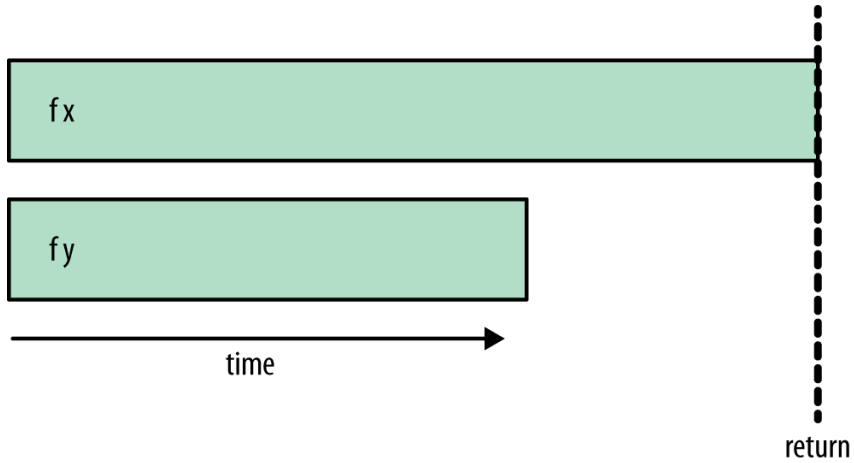
```
-- wait for (f y) and (f x)
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y)  -- wait
    rseq a           -- wait
    return (a,b)
```



Ein Beispiel (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.

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

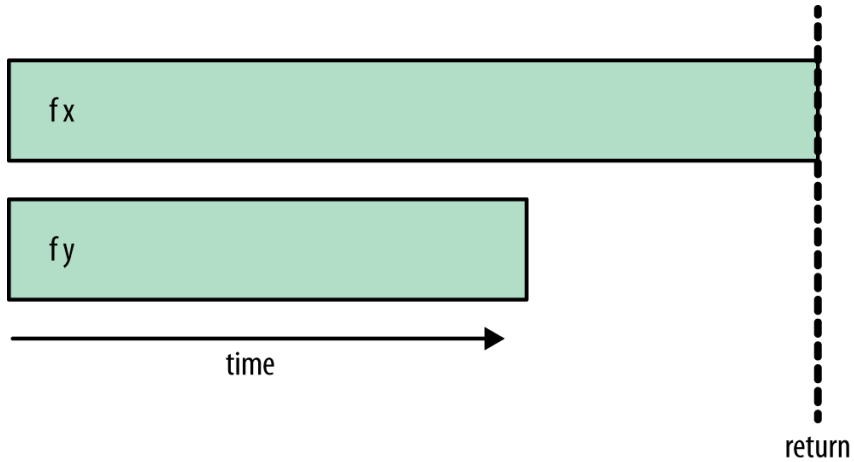


In diesem Code wird sowohl auf `(f x)` als auch auf `(f y)` gewartet, bevor etwas zurück gegeben wird.

Ein Beispiel (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
    return (a,b)
```



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

Ein weiteres Beispiel: Wir wollen ein Programm zum Lösen von Sudokus parallelisieren.
Wir nehmen dazu an wir haben bereits die folgende Funktion:

```
solve :: String -> Maybe Grid
```

Ein weiteres Beispiel: Wir wollen ein Programm zum Lösen von Sudokus parallelisieren.
Wir nehmen dazu an wir haben bereits die folgende Funktion:

```
solve :: String -> Maybe Grid
```

Dann wäre ein mögliches (sequentielles) Programm das folgende:

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

    let puzzles   = lines file
        solutions = map solve puzzles

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

Nun wollen wir Liste der Lösungen parallel auf zwei Kernen ausführen lassen:

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

  let puzzles = lines file

      (as,bs) = splitAt (length puzzles `div` 2) puzzles

      solutions = runEval $ do
                    as' <- rpar (force (map solve as))
                    bs' <- rpar (force (map solve bs))
                    rseq as'
                    rseq bs'
                    return (as' ++ bs')

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

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

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

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 Evaluation zu forcen.

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 Evaluation zu forcen.

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

Einschub: Die `NFData`-Typklasse:

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

Einschub: Die `NFData`-Typklasse:

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

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

Einschub: Die `NFData`-Typklasse:

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

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

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

`rnf` bringt Standardimplementation mit, dies erleichtert Instanzen von simplen Datentypen ohne Substrukturen:

```
instance NFData Bool
```

Einschub: Die `NFData`-Typklasse:

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

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

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

`rnf` bringt Standardimplementation mit, dies erleichtert Instanzen von simplen Datentypen ohne Substrukturen:

```
instance NFData Bool
```

Instanzen von Typen mit Substrukturen funktionieren 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
```

Zurück zu unserem Beispiel:

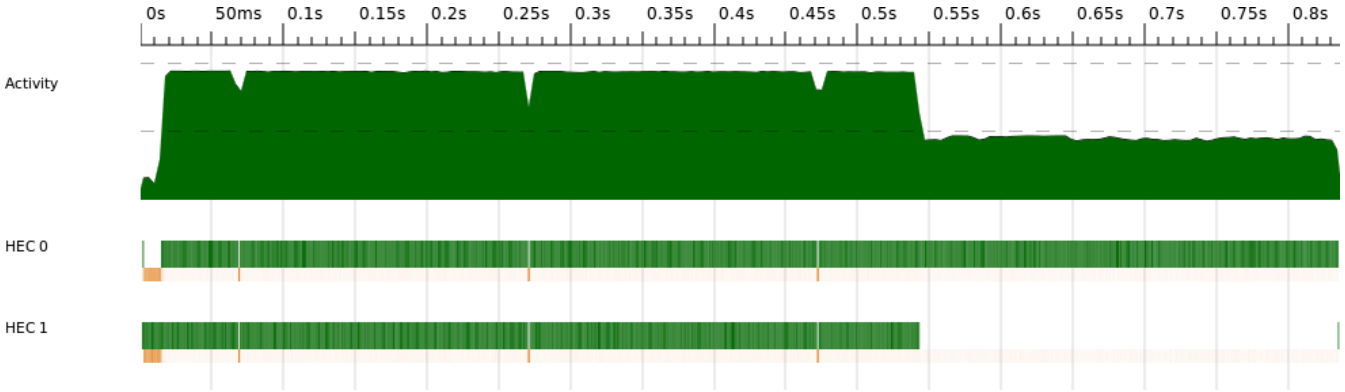
```
main :: IO ()
main = do
  [f]  <- getArgs
  file <- readFile f
  let puzzles = lines file
      (as,bs) = splitAt (length puzzles `div` 2) puzzles
      solutions = runEval $ do
                    as' <- rpar (force (map solve as))
                    bs' <- rpar (force (map solve bs))
        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

Zurück zu unserem Beispiel:

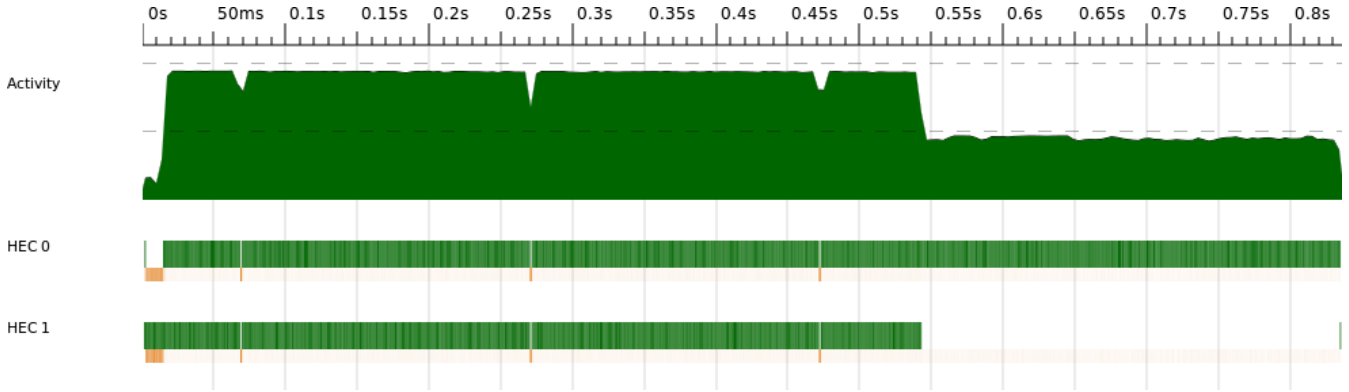
```
main :: IO ()
main = do
  [f]  <- getArgs
  file <- readFile f
  let puzzles = lines file
      (as,bs) = splitAt (length puzzles `div` 2) puzzles
      solutions = runEval $ do
                    as' <- rpar (force (map solve as))
                    bs' <- rpar (force (map solve bs))
        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

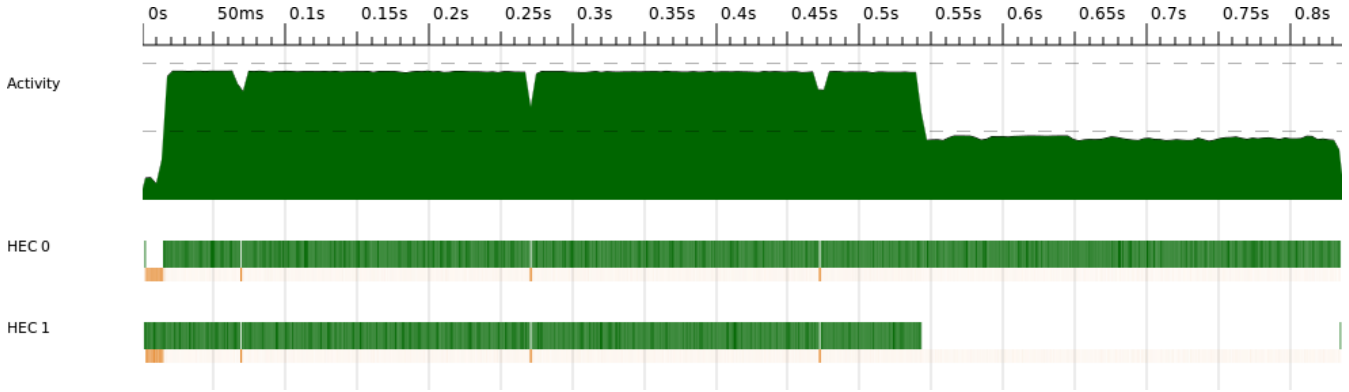


Analyse mit *ThreadScope*:



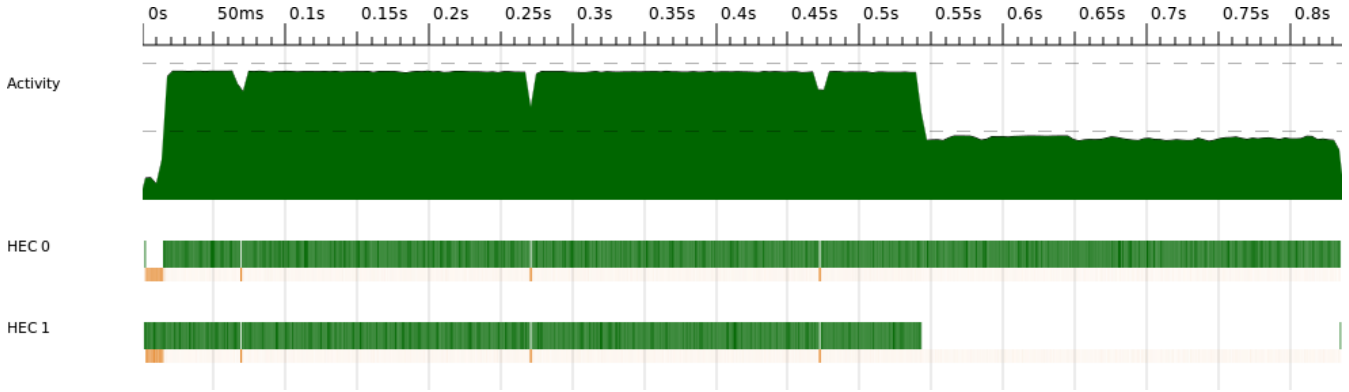
Wir bemerken: Unsere parallelen Berechnungen sind ungleich groß. Eine benötigt deutlich länger als die andere.

Analyse mit *ThreadScope*:



Wir bemerken: Unsere parallelen Berechnungen sind ungleich groß. Eine benötigt deutlich länger als die andere.
Auch dies ist ein häufiges Problem mit Parallelisierung: Chunks von voraus bestimmter Größe (*static partitioning*) enthalten nur selten tatsächlich gleich viel Arbeit.

Analyse mit *ThreadScope*:



Wir bemerken: Unsere parallelen Berechnungen sind ungleich groß. Eine benötigt deutlich länger als die andere.
Auch dies ist ein häufiges Problem mit Parallelisierung: Chunks von voraus bestimmter Größe (*static partitioning*) enthalten nur selten tatsächlich gleich viel Arbeit.

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

Diese Probleme können wir lösen, indem wir von *static partitioning* auf *dynamic partitioning* wechseln.

Das bedeutet, anstatt von Hand ein paar große Chunks anzugeben, auf denen Parallelism angewendet werden soll, geben wir viele kleine Chunks an, die dann zur Laufzeit unter den Prozessorkernen aufgeteilt werden.

Diese Probleme können wir lösen, indem wir von *static partitioning* auf *dynamic partitioning* wechseln.

Das bedeutet, anstatt von Hand ein paar große Chunks anzugeben, auf denen Parallelism angewendet werden soll, geben wir viele kleine Chunks an, die dann zur Laufzeit unter den Prozessorkernen 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 magische (sprich: schlaue) Weise parallel evaluiert werden können.

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

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

- mehrere Milliarden Sparks,

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

- mehrere Milliarden Sparks,
- ca. eine Million lightweight Haskell-Threads,

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

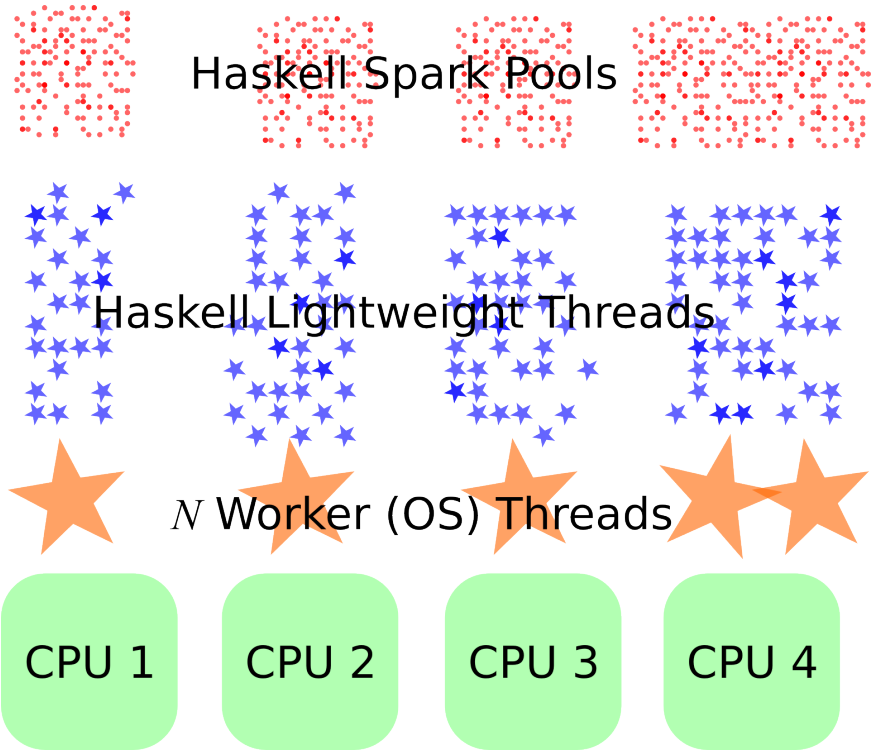
- mehrere Milliarden Sparks,
- ca. eine Million lightweight Haskell-Threads,
- ein Dutzend OS-Thread,

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

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

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

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



Grafik von Don Stewart, Quelle:
`http://stackoverflow.com/questions/958449/what-is-a-spark-in-haskell`

Hands on: Wir definieren uns folgende monadische Version von `map`:

Hands on: Wir definieren uns folgende monadische Version von `map`:

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

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

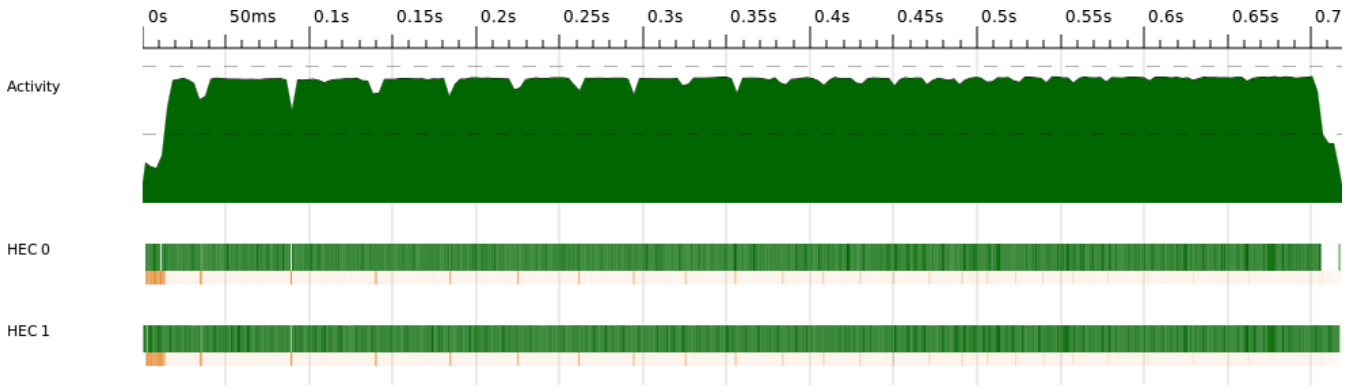
Hands on: Wir definieren uns folgende monadische Version von `map`:

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

Nun wird für jede Funktionsanwendung `(f a)` ein *Spark* erstellt, die alle parallel arbeiten und vom RTS des GHC gemanaged werden.
Eingesetzt in unser Beispiel:

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

Analyse mit *ThreadScope*:



Schon viel besser. Der Speedup beträgt jetzt $\sim 1,8$. Den optimalen Wert von 2 zu erreichen ist (praktisch) unmöglich, da immer ein Overhead für das Management der Sparks entstehen muss.

Wir können uns auch eine Übersicht darüber geben lassen, was mit den Sparks passiert ist:

```
SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

Wir können uns auch eine Übersicht darüber geben lassen, was mit den Sparks passiert ist:

```
SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

Was bedeutet das jetzt im Einzelnen?

Wir können uns auch eine Übersicht darüber geben lassen, was mit den Sparks passiert ist:

```
SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

Was bedeutet das jetzt im Einzelnen?

- `converted`: der Spark wurde erfolgreich für Parallelism verwendet.

Wir können uns auch eine Übersicht darüber geben lassen, was mit den Sparks passiert ist:

```
SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

Was bedeutet das jetzt im Einzelnen?

- `converted`: der Spark wurde erfolgreich für Parallelism verwendet.
- `overflowed`: Die maximale Anzahl Sparks wurde überschritten, Spark gelöscht.

Wir können uns auch eine Übersicht darüber geben lassen, was mit den Sparks passiert ist:

```
SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

Was bedeutet das jetzt im Einzelnen?

- `converted`: der Spark wurde erfolgreich für Parallelism verwendet.
- `overflowed`: Die maximale Anzahl Sparks wurde überschritten, Spark gelöscht.
- `dud`: Es wurde ein Spark für einen Ausdruck erstellt, der bereits ausgewertet wurde.

Wir können uns auch eine Übersicht darüber geben lassen, was mit den Sparks passiert ist:

```
SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

Was bedeutet das jetzt im Einzelnen?

- `converted`: der Spark wurde erfolgreich für Parallelism verwendet.
- `overflowed`: Die maximale Anzahl Sparks wurde überschritten, Spark gelöscht.
- `dud`: Es wurde ein Spark für einen Ausdruck erstellt, der bereits ausgewertet wurde.
- `GC'd`: Der evaluierte Ausdruck wurde nicht benötigt und garbage collected.

Wir können uns auch eine Übersicht darüber geben lassen, was mit den Sparks passiert ist:

```
SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

Was bedeutet das jetzt im Einzelnen?

- `converted`: der Spark wurde erfolgreich für Parallelism verwendet.
- `overflowed`: Die maximale Anzahl Sparks wurde überschritten, Spark gelöscht.
- `dud`: Es wurde ein Spark für einen Ausdruck erstellt, der bereits ausgewertet wurde.
- `GC'd`: Der evaluierte Ausdruck wurde nicht benötigt und garbage collected.
- `fizzled`: Ausdruck wurde an anderer Stelle schneller vom Programm ausgewertet.

Das Rabbithole zu Strategien geht ziemlich tief. Andere Funktionen, die zur Verfügung stehen, sind zum Beispiel...

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

r0        :: Strategy a                -- evaluiert nichts
rdeepseq :: NFData a => Strategy a -- evaluiert komplett

dot :: Strategy a -> Strategy a -> Strategy a -- Kombination

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

Das Rabbithole zu Strategien geht ziemlich tief. Andere Funktionen, die zur Verfügung stehen, sind zum Beispiel...

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

r0        :: Strategy a                -- evaluiert nichts
rdeepseq :: NFData a => Strategy a -- evaluiert komplett

dot :: Strategy a -> Strategy a -> Strategy a -- Kombination

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

All diese sind wunderschöne Werkzeuge um schnell und einfach adequaten Parallelism zu erzeugen. Außerdem können wir einfach die Parallelisierung vom Algorithmus trennen.

Es gibt aber auch andere Use Cases. Etwas mehr Finetuning bietet zum Beispiel...

Parallelism

- Die `Eval`-Monade und Strategies
- ○ Überblick: Die `Par`-Monade
- Überblick: Die `RePa`-Bibliothek und `Accelerate`

Alternative zu Strategies, mit anderen Schwerpunkten und anderen Tradeoffs.

Alternative zu Strategies, mit anderen Schwerpunkten und anderen Tradeoffs.

Das Interface heißt - Überraschung! - `Par`.

```
newtype Par a
instance Monad Par

runPar :: Par a -> a
```

Alternative zu Strategies, mit anderen Schwerpunkten und anderen Tradeoffs.

Das Interface heißt - Überraschung! - `Par`.

```
newtype Par a
instance Monad Par

runPar :: Par a -> a
```

Wir können explizit Parallelism erzeugen mit einer Funktion namens `fork`, die ihr Argument (das „Kind“) parallel zum Aufrufenden Programm („Elter“) ausführt.

```
fork :: Par () -> Par ()
```

Alternative zu Strategies, mit anderen Schwerpunkten und anderen Tradeoffs.

Das Interface heißt - Überraschung! - `Par`.

```
newtype Par a
instance Monad Par

runPar :: Par a -> a
```

Wir können explizit Parallelism erzeugen mit einer Funktion namens `fork`, die ihr Argument (das „Kind“) parallel zum Aufrufenden Programm („Elter“) ausführt.

```
fork :: Par () -> Par ()
```

Aber `fork` gibt nichts an das Elter zurück, wie können wir dann Informationen austauschen?

Introducing: *IVars*

```
data IVar a  -- instance Eq

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

Introducing: *IVars*

```
data IVar a  -- instance Eq

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

IVars können benutzt werden, um Daten zwischen Berechnungen in `Par` zu übergeben.

Stellt euch eine Kiste vor, die zunächst mal leer startet (`new`). Mit `put` können Daten von einer Berechnung in der `Par`-Monade in die Kiste hinein gelegt und mit `get` von einer anderen ausgelesen werden. Ist noch nichts in der Kiste wartet `get` so lange bis sich das ändert, bevor es weiter geht.

Introducing: *IVars*

```
data IVar a  -- instance Eq

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

IVars können benutzt werden, um Daten zwischen Berechnungen in `Par` zu übergeben.

Stellt euch eine Kiste vor, die zunächst mal leer startet (`new`). Mit `put` können Daten von einer Berechnung in der `Par`-Monade in die Kiste hinein gelegt und mit `get` von einer anderen ausgelesen werden. Ist noch nichts in der Kiste wartet `get` so lange bis sich das ändert, bevor es weiter geht.
(Ein ähnlicher Datentyp, *MVar*, wird uns im Kontext von `Haskell` und Concurrency noch begegnen.)

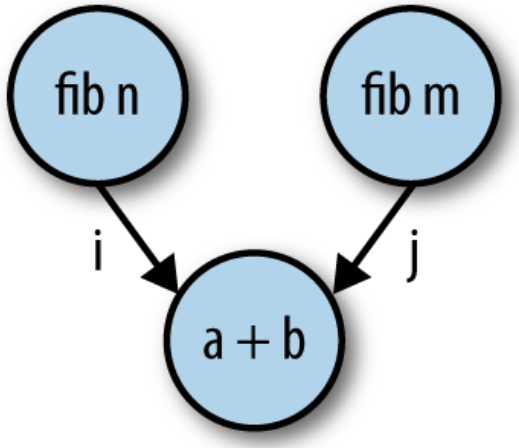
Ein einfaches Beispiel mit `Par`:

```
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)
```

Ein einfaches Beispiel mit `Par`:

```
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)
```

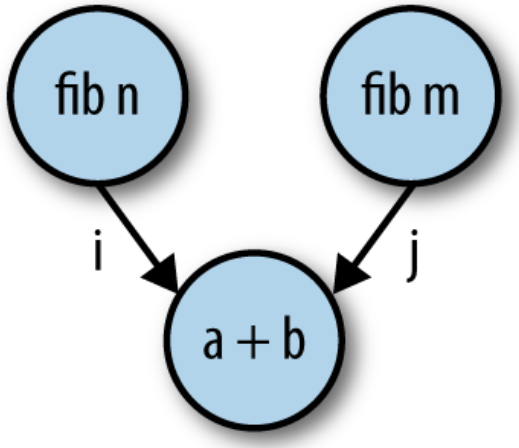
Graph:



Ein einfaches Beispiel mit `Par`:

```
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)
```

Graph:



`fork`: Neuer Knoten
`new`: Neue Kante
`put, get`: Kanten & Knoten verbinden

Die `Par`-Monade kann allerdings noch mehr als nur hübsche Datenfluss-Graphen! Wie würden wir unsere parallele Version von `map` von vorhin hier formulieren?

Die Par-Monade kann allerdings noch mehr als nur hübsche Datenfluss-Graphen! Wie würden wir unsere parallele Version von map von vorhin hier formulieren?

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

parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do ibs <- mapM (spawn . f) as
                  mapM get ibs
```

Die `Par`-Monade kann allerdings noch mehr als nur hübsche Datenfluss-Graphen! Wie würden wir unsere parallele Version von `map` von vorhin hier formulieren?

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

parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do ibs <- mapM (spawn . f) as
                  mapM get ibs
```

Da `f :: (a -> Par b)` kann auch in `f` wieder Parallelism verwendet werden.

Ein paar Faustregeln zur `Par`-Monade (im Vergleich zu Strategies):

Ein paar Faustregeln zur `Par`-Monade (im Vergleich zu Strategies):

- Wenn euer Programm eine lazy Datenstruktur ausspuckt, die ihr mit Strategien parallelisieren wollt, funktioniert das in der Regel gut. Ansonsten bietet sich `Par` an.

Ein paar Faustregeln zur `Par`-Monade (im Vergleich zu Strategies):

- Wenn euer Programm eine lazy Datenstruktur ausspuckt, die ihr mit Strategien parallelisieren wollt, funktioniert das in der Regel gut. Ansonsten bietet sich `Par` an.
- `runEval` ist quasi umsonst, `runPar` ist teuer. Idealerweise `runPar` um alle Stellen wickeln, die Parallelism brauchen und nicht rekursiv aufrufen.

Ein paar Faustregeln zur `Par`-Monade (im Vergleich zu Strategies):

- Wenn euer Programm eine lazy Datenstruktur ausspuckt, die ihr mit Strategien parallelisieren wollt, funktioniert das in der Regel gut. Ansonsten bietet sich `Par` an.
- `runEval` ist quasi umsonst, `runPar` ist teuer. Idealerweise `runPar` um alle Stellen wickeln, die Parallelism brauchen und nicht rekursiv aufrufen.
- Kein „speculative Parallelism“ in der `Par`-Monade.

Ein paar Faustregeln zur `Par`-Monade (im Vergleich zu Strategies):

- Wenn euer Programm eine lazy Datenstruktur ausspuckt, die ihr mit Strategien parallelisieren wollt, funktioniert das in der Regel gut. Ansonsten bietet sich `Par` an.
- `runEval` ist quasi umsonst, `runPar` ist teuer. Idealerweise `runPar` um alle Stellen wickeln, die Parallelism brauchen und nicht rekursiv aufrufen.
- Kein „speculative Parallelism“ in der `Par`-Monade.
- Strategies erlauben, den eigentlichen Code komplett vom Parallelism zu trennen. `(expr `using` strat)` vs. `(expr)`

Ein paar Faustregeln zur `Par`-Monade (im Vergleich zu Strategies):

- Wenn euer Programm eine lazy Datenstruktur ausspuckt, die ihr mit Strategien parallelisieren wollt, funktioniert das in der Regel gut. Ansonsten bietet sich `Par` an.
- `runEval` ist quasi umsonst, `runPar` ist teuer. Idealerweise `runPar` um alle Stellen wickeln, die Parallelism brauchen und nicht rekursiv aufrufen.
- Kein „speculative Parallelism“ in der `Par`-Monade.
- Strategies erlauben, den eigentlichen Code komplett vom Parallelism zu trennen. `(expr `using` strat)` vs. `(expr)`
- `Par` ist eine reine Haskell-Bibliothek, leichter zu modifizieren (Scheduler)

Ein paar Faustregeln zur `Par`-Monade (im Vergleich zu Strategies):

- Wenn euer Programm eine lazy Datenstruktur ausspuckt, die ihr mit Strategien parallelisieren wollt, funktioniert das in der Regel gut. Ansonsten bietet sich `Par` an.
- `runEval` ist quasi umsonst, `runPar` ist teuer. Idealerweise `runPar` um alle Stellen wickeln, die Parallelism brauchen und nicht rekursiv aufrufen.
- Kein „speculative Parallelism“ in der `Par`-Monade.
- Strategies erlauben, den eigentlichen Code komplett vom Parallelism zu trennen. `(expr `using` strat)` vs. `(expr)`
- `Par` ist eine reine Haskell-Bibliothek, leichter zu modifizieren (Scheduler)
- Strategies haben besseren Support vom RTS und ThreadScope

Parallelism

- Die `Eval`-Monade und Strategies
- Überblick: Die `Par`-Monade
- ○ Überblick: Die `RePA`-Bibliothek und `Accelerate`

Die bisherigen Ansätze für Parallelism waren gut geeignet für herkömmliche Datenstrukturen in `Haskell`. Für manche Probleme (oft solche, die große Arrays voller ungeboxter Daten verarbeiten) ist das aber nicht der richtige Weg.

Die bisherigen Ansätze für Parallelism waren gut geeignet für herkömmliche Datenstrukturen in `Haskell`. Für manche Probleme (oft solche, die große Arrays voller ungeboxter Daten verarbeiten) ist das aber nicht der richtige Weg.

Beispiele:

- Bildverarbeitung

Die bisherigen Ansätze für Parallelism waren gut geeignet für herkömmliche Datenstrukturen in `Haskell`. Für manche Probleme (oft solche, die große Arrays voller ungeboxter Daten verarbeiten) ist das aber nicht der richtige Weg.

Beispiele:

- Bildverarbeitung
- high-performance number crunching

Die bisherigen Ansätze für Parallelism waren gut geeignet für herkömmliche Datenstrukturen in `Haskell`. Für manche Probleme (oft solche, die große Arrays voller ungeboxter Daten verarbeiten) ist das aber nicht der richtige Weg.

Beispiele:

- Bildverarbeitung
- high-performance number crunching
- ...

Die bisherigen Ansätze für Parallelism waren gut geeignet für herkömmliche Datenstrukturen in `Haskell`. Für manche Probleme (oft solche, die große Arrays voller ungeboxter Daten verarbeiten) ist das aber nicht der richtige Weg.

Beispiele:

- Bildverarbeitung
- high-performance number crunching
- ...

Für diese Zwecke wurden die Bibliotheken `REPA` und `Accelerate` entwickelt. Ihre Datenstrukturen sind sich im Kern ähnlich...

Ein kurzer Blick auf die wichtigsten Datentypen von `REPA` (Abk: REgular PArallel arrays):

Ein kurzer Blick auf die wichtigsten Datentypen von `REPA` (Abk: REgular PArallel arrays):

```
data Array r sh e
```

- `e` ist der Datentyp, der gespeichert werden soll (unboxed).

Ein kurzer Blick auf die wichtigsten Datentypen von `REPA` (Abk: REgular PArallel arrays):

```
data Array r sh e
```

- `e` ist der Datentyp, der gespeichert werden soll (unboxed).
- `sh` ist die Form (*shape*) des Arrays.

Ein kurzer Blick auf die wichtigsten Datentypen von `REPA` (Abk: REgular PArallel arrays):

```
data Array r sh e
```

- `e` ist der Datentyp, der gespeichert werden soll (unboxed).
- `sh` ist die Form (*shape*) des Arrays.
- `r` ist der *representation type*. Dazu gleich mehr.

Ein kurzer Blick auf die wichtigsten Datentypen von `REPA` (Abk: REgular PArallel arrays):

```
data Array r sh e
```

- `e` ist der Datentyp, der gespeichert werden soll (unboxed).
- `sh` ist die Form (*shape*) des Arrays.
- `r` ist der *representation type*. Dazu gleich mehr.

Die Form eines Arrays ist entweder ein Skalar oder eine Form „mal“ eine zusätzliche (immer durch `Int` indizierte) Dimension.

```
data Z = Z -- Scalar
data tail :. head
          = tail :. head
```

Ein kurzer Blick auf die wichtigsten Datentypen von `REPA` (Abk: REgular PArallel arrays):

```
data Array r sh e
```

- `e` ist der Datentyp, der gespeichert werden soll (unboxed).
- `sh` ist die Form (*shape*) des Arrays.
- `r` ist der *representation type*. Dazu gleich mehr.

Die Form eines Arrays ist entweder ein Skalar oder eine Form „mal“ eine zusätzliche (immer durch `Int` indizierte) Dimension.

```
data Z = Z -- Scalar
data tail :. head
          = tail :. head
```

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
```

Ein kurzer Blick auf die wichtigsten Datentypen von `REPA` (Abk: REgular PArallel arrays):

```
data Array r sh e
```

- `e` ist der Datentyp, der gespeichert werden soll (unboxed).
- `sh` ist die Form (*shape*) des Arrays.
- `r` ist der *representation type*. Dazu gleich mehr.

Die Form eines Arrays ist entweder ein Skalar oder eine Form „mal“ eine zusätzliche (immer durch `Int` indizierte) Dimension.

```
data Z = Z -- Scalar
data tail :. head
          = tail :. head
```

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
```

Es gibt einige Operationen auf `REPA`-Arrays, die denen auf normalen Arrays zumindest ähneln:

Es gibt einige Operationen auf REPA-Arrays, die denen auf normalen Arrays zumindest ähneln:

```
fromListUnboxed :: (Shape sh, Unbox a) => sh -> [a]
                                       -> Array U sh a
(!)  :: (Shape sh, Source r e) => Array r sh e -> sh -> e
size :: Shape sh => sh -> Int

Repa.map :: (Shape sh, Source r a)
         => (a -> b) -> Array r sh a -> Array D sh b
```

Es gibt einige Operationen auf `REPA`-Arrays, die denen auf normalen Arrays zumindest ähneln:

```
fromListUnboxed :: (Shape sh, Unbox a) => sh -> [a]
                                       -> Array U sh a
(!)  :: (Shape sh, Source r e) => Array r sh e -> sh -> e
size :: Shape sh => sh -> Int

Repa.map :: (Shape sh, Source r a)
         => (a -> b) -> Array r sh a -> Array D sh b
```

Bei der Anwendung von `Repa.map` erhalten wir ein Array von `D` (Delayed) werten.
Das ist der Mechanismus, um mehrere Funktionsapplikationen zu einer zusammen zu schmelzen (genannt „Fusion“).

Mit `REPA` und `Accelerate` zu arbeiten ist etwas extra Arbeit im Design...

Mit REPA und Accelerate zu arbeiten ist etwas extra Arbeit im Design...

```
let a = fromListUnboxed (Z :. 10) [1..10] :: Array U DIM1 Int
computeS (Repa.map (+1) a) :: Array U DIM1 Int
> AUnboxed (Z :. 10) (fromList [2,3,4,5,6,7,8,9,10,11])

computeS (Repa.map (+1) (Repa.map (^2) a)) :: Array U DIM1 Int
> AUnboxed (Z :. 10) (fromList [2,5,10,17,26,37,50,65,82,101])
```

Mit `REPA` und `Accelerate` zu arbeiten ist etwas extra Arbeit im Design...

```
let a = fromListUnboxed (Z :. 10) [1..10] :: Array U DIM1 Int
computeS (Repa.map (+1) a) :: Array U DIM1 Int
> AUnboxed (Z :. 10) (fromList [2,3,4,5,6,7,8,9,10,11])

computeS (Repa.map (+1) (Repa.map (^2) a)) :: Array U DIM1 Int
> AUnboxed (Z :. 10) (fromList [2,5,10,17,26,37,50,65,82,101])
```

...dafür können wir diese Berechnungen aber dann mit `computeP` oder `foldP` extrem fix quasi automatisch parallelisieren.

Mit REPA und Accelerate zu arbeiten ist etwas extra Arbeit im Design...

```
let a = fromListUnboxed (Z :. 10) [1..10] :: Array U DIM1 Int
computeS (Repa.map (+1) a) :: Array U DIM1 Int
> AUnboxed (Z :. 10) (fromList [2,3,4,5,6,7,8,9,10,11])

computeS (Repa.map (+1) (Repa.map (^2) a)) :: Array U DIM1 Int
> AUnboxed (Z :. 10) (fromList [2,5,10,17,26,37,50,65,82,101])
```

...dafür können wir diese Berechnungen aber dann mit computeP oder foldP extrem fix quasi automatisch parallelisieren.

REPA unterstützt auch „stencil convolutions“. Wenn z.B. eine Funktion auf jedes Pixel angewendet werden soll, die die Nachbarpixel mit einbezieht, kann REPA hochspezialisierten Code generieren, der extrem schnell durchläuft.