Fortgeschrittene Funktionale Programmierung

Teil 6: Paralleles Rechnen

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

8. Januar 2018

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 6: Paralleles Rechnen

Grundlagen: Parallelität≠Nebenläufigkeit

Paralleles Rechnen: Ziel ist *schnelle* Ausführung von Programmen durch gleichzeitige Verwendung mehrerer Prozessoren.

Computer mit mehreren Kernen und Cloud-Architekturen sind Standard und ermöglichen paralleles Rechnen.
Problem: Viele Ansätze für paralleles Rechnen sind sehr "low-level" und schwierig zu handhaben.

Parallele funktionale Sprachen bieten einen hochsprachlichen Ansatz, in dem nur wenige Aspekte der parallelen Berechnung bestimmt werden müssen!

Dagegen bedeutet **Nebenläufigkeit** *engl. Concurrency* nicht-deterministische Berechnungen durch zufällig abwechselnd ausgeführte interagierende Prozesse, z.B. Reaktion auf verschiedene externe Ereignisse (nicht unbedingt parallel – auch auf einem Prozessorkern kann abgewechselt werden).

z.B. Anfragen an Webserver

Grundlagen

Parallele Berechnung ist schwierig:

Berechnung

korrekter und effizienter Algorithmus zur Berechnung des Gewünschten (wie in sequentieller Berechnung)

Koordination

Sinnvolle Einteilung der Berechnung in unabhängige Einheiten, welche parallel ausgewertet werden können

Beurteilung der Effizienz erfolgt primär durch Vergleich der Beschleunigung relativ zur Berechnung mit einem Prozessor.

Beispiel: Faktor 14 ist gut, wenn statt 1 Prozessor 16 verwendet werden, aber schlecht, wenn 128 verwendet werden (dürfen).

Amdahl's Law maximale Beschleunigung $\leq \frac{1}{(1-P)+P/N}$

mit $N =$ Anzahl Kerne, $P =$ parallelisierbarer Anteil des Programms

Grundbegriffe

Thread Ausführungsstrang eines Programmes, welcher sequentiell abläuft.
Ein Programm oder Prozess kann aus mehreren Threads bestehen.

HEC Ein Thread eines Haskell Programmes wird auch als Haskell Execution Context bezeichnet.

Core Prozessorkern, welcher jeweils einen Thread abarbeiten kann.

Üblicherweise gibt es mehr Threads als Prozessorkerne. Das Betriebssystem kümmert sich darum, alle Threads auf die Prozessorkerne zu verteilen.

Dabei gibt es verschiedene Strategien. Meist werden Threads regelmäßig unterbrochen, um alle Threads scheinbar gleichzeitig auszuführen.

Koordinierung der parallelen Ausführung

Partitionierung Aufspaltung des Programms in unabhängige, parallel berechenbare Teile, **Threads**

- Wie viele Threads?
- Wie viel macht ein einzelner Thread?

Synchronisation
Abhängigkeiten zwischen Threads identifizieren

Kommunikation / Speicher Management
Austausch der Daten zwischen den Threads

Mapping Zuordnung der Threads zu Prozessoren

Scheduling Auswahl lauffähiger Threads auf einem Prozessor

$\Rightarrow$ Explizite Spezifizierung durch den Programmierer sehr aufwändig und auch sehr anfällig für Fehler ↯
(z.B. drohen Deadlocks und Race-Conditions)

Probleme bei parallelen Berechnung

Bei parallelen Berechnungen mit mehreren Threads können normalerweise u.a. folgende Probleme auftreten:

Race-Condition Verschiedene Threads können sich durch Seiteneffekte gegenseitig beeinflussen. Da manchmal ein Thread schneller als ein anderer abgehandelt wird und die Möglichkeiten der Verzahnung immens sind, ist das Gesamtergebnis der Berechnung nicht vorhersagbar.

Deadlock Ein Thread wartet auf das Ergebnis eines anderen Threads, welcher direkt oder indirekt selbst auf das Ergebnis des ersten Threads wartet. Die Berechnung kommt somit zum erliegen.

Diese Probleme lassen sich in *nebenläufigen* Programmen nicht generell vermeiden. In rein funktionalen *parallelen* Programmen können diese Probleme aber von vornherein eliminiert werden!

Parallele Funktionale Sprachen

Rein funktionale Programmiersprachen haben keine Seiteneffekte und sind **referentiell transparent**.

Stoy (1977):

> *The only thing that matters about an expression is its value, and any sub-expression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same wherever it occurs.*

Insbesondere ist die Auswertungsreihenfolge (nahezu) beliebig.

$\Rightarrow$ Ideal, um verschiedene Programmteile parallel zu berechnen!

Aber: Ansätze zur vollautomatischen Parallelisierung von Programmen sind bisher leider gescheitert!

Verwendung mehrerer Kerne in GHC

Anzahl der verwendbaren Kerne in GHC einstellbar:

- **Kompiler-Parameter, statisch:** Für 32 Kerne kompilieren mit `ghc prog.hs -threaded -with-rtsopts="-N32"`
- **Kommandozeilenparameter:** Einmal kompilieren mit
  `ghc prog.hs -threaded -rtsopts`
  und dann Aufrufen mit `./prog +RTS -N32`
  RTS=RunTime System; autom. Anzahl Kerne `+RTS -qa`
- **Dynamisch im Programm** mit Modul *Control.Concurrent*

```
getNumCapabilities :: IO Int
setNumCapabilities :: Int -> IO ()
setNumCapabilities 32
```

Ebenfalls kompilieren mit `-threaded`

⇒ Dies *erlaubt* jeweils die Benutzung mehrerer Threads, das Programm selbst muss aber noch angepasst werden!

Profiling

GHC erlaubt Profiling, d.h. zur Laufzeit wird protokolliert, was das Programm wie lange wirklich macht. Spezielles kompilieren mit:

```
> ghc MyProg.hs -O2 -prof -fprof-auto -rtsopts
> ./MyProg +RTS -p
```

erstellt Datei `MyProg.prof`, in der man sehen kann wie viel Zeit bei der Auswertung der einzelnen Funktionen verwendet wurde.

- Genutzte Module müssen mit Profiling-Unterstützung installiert sein `cabal install mein-modul -p` `stack build --profile; stack exec -- Main +RTS -p`
- Viele Optionen verfügbar. Ohne `-fprof-auto` werden z.B. nur komplette Module abgerechnet.
- `+RTS -h` für Speicher-Profiling

Beispiel: Profiling

```
COST CENTRE    MODULE  %time %alloc

fakultät       Main     56.8   88.8
numberLength   Main      9.9    1.4
collatzLength  Main      9.9    0.9
hanoi          Main      8.6    4.6
collatzStep    Main      7.4    2.1
fibs           Main      4.9    2.0
main0          Main      2.5    0.0
```

COST CENTRE	MODULE	no.	entries	individual %time	individual %alloc	inherited %time	inherited %alloc
MAIN	MAIN	58	0	0.0	0.0	100.0	100.0
main0	Main	118	0	2.5	0.0	2.5	0.0
printTimeDiff	Main	137	1	0.0	0.0	0.0	0.0
printLocalTime	Main	120	0	0.0	0.0	0.0	0.0
CAF	Main	115	0	0.0	0.0	97.5	100.0
fakNumber	Main	135	1	0.0	0.0	0.0	0.0
cseqLength	Main	133	1	0.0	0.0	0.0	0.0
numbersWithCSequenceLength	Main	130	1	0.0	0.1	17.3	3.2
collatzLength	Main	132	173813	9.9	0.9	17.3	3.0
collatzStep	Main	134	171350	7.4	2.1	7.4	2.1
hanoiHeight	Main	125	1	0.0	0.0	0.0	0.0
fibs	Main	124	1	4.9	2.0	4.9	2.0
fibNumber	Main	121	1	0.0	0.0	0.0	0.0
printLocalTime	Main	119	1	0.0	0.0	0.0	0.0
main0	Main	117	1	0.0	0.0	75.3	94.8
printTimeDiff	Main	138	0	0.0	0.0	0.0	0.0
fakultät	Main	136	1	56.8	88.8	56.8	88.8
numberWithCSequenceLength	Main	129	1	0.0	0.0	0.0	0.0
numberWithCSequenceLength.\	Main	131	2463	0.0	0.0	0.0	0.0
main0.r	Main	126	1	0.0	0.0	8.6	4.6
hanoi	Main	127	32767	8.6	4.6	8.6	4.6

Runtime-system statistics

Für einen ersten Blick zur Performance eines Programmes reichen oft auch schon die einfachen Laufzeit-Statistiken.

Dafür reicht die Kompiler-Option `-rtsops` bereits aus.

Erstellt wird die Statistik beim Aufruf mit der RTS-Option `-s`

```
> ./MyProg +RTS -s
```

Optional kann auch ein Dateiname angegeben werden, um die Statistiken abzuspeichern.

Beispiel: Laufzeit-Statistik

```
> ./rpar 2 +RTS -N3 -s
1,876,383,792 bytes allocated in the heap
    1,043,816 bytes copied during GC
       46,856 bytes maximum residency (2 sample(s))
       39,160 bytes maximum slop
            2 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0      2307 colls,  2307 par    0.07s    0.02s     0.0000s    0.0005s
Gen  1         2 colls,     1 par    0.00s    0.00s     0.0002s    0.0002s

Parallel GC work balance: 24.95% (serial 0%, perfect 100%)
TASKS: 5 (1 bound, 4 peak workers (4 total), using -N3)
SPARKS: 1 (1 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.00s  (  0.00s elapsed)
MUT     time    2.78s  (  1.66s elapsed)
GC      time    0.07s  (  0.03s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    2.85s  (  1.68s elapsed)

Alloc rate    676,021,486 bytes per MUT second

Productivity  97.4% of total user, 165.2% of total elapsed
```

Beispiel: Laufzeit-Statistik

```
> ./rpar 2 +RTS -N3 -s
1,876,383,792 bytes allocated in the heap
    1,043,816 bytes copied during GC
       46,856 bytes maximum residency (2 sample(s))
       39,160 bytes maximum slop
            2 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0      2307 colls,  2307 par    0.07s    0.02s     0.0000s    0.0005s
Gen  1         2 colls,     1 par    0.00s    0.00s     0.0002s    0.0002s

Parallel GC work balance: 24.95% (serial 0%, perfect 100%)
TASKS: 5 (1 bound, 4 peak workers (4 total), using -N3)
SPARKS: 1 (1 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.00s  (  0.00s elapsed)
MUT     time    2.78s  (  1.66s elapsed)
GC      time    0.07s  (  0.03s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    2.85s  (  1.68s elapsed)

Alloc rate    676,021,486 bytes per MUT second

Productivity  97.4% of total user, 165.2% of total elapsed
```

Man kann Speicherverbrauch, Zeitaufwand für Garbage Collection und Eckdaten zur parallelen Auswertung ablesen.

Threadscope

Speziell für das Profiling von parallel/nebenläufig ausgeführten Haskell Programmen gibt es den **Threadscope** Profiler:

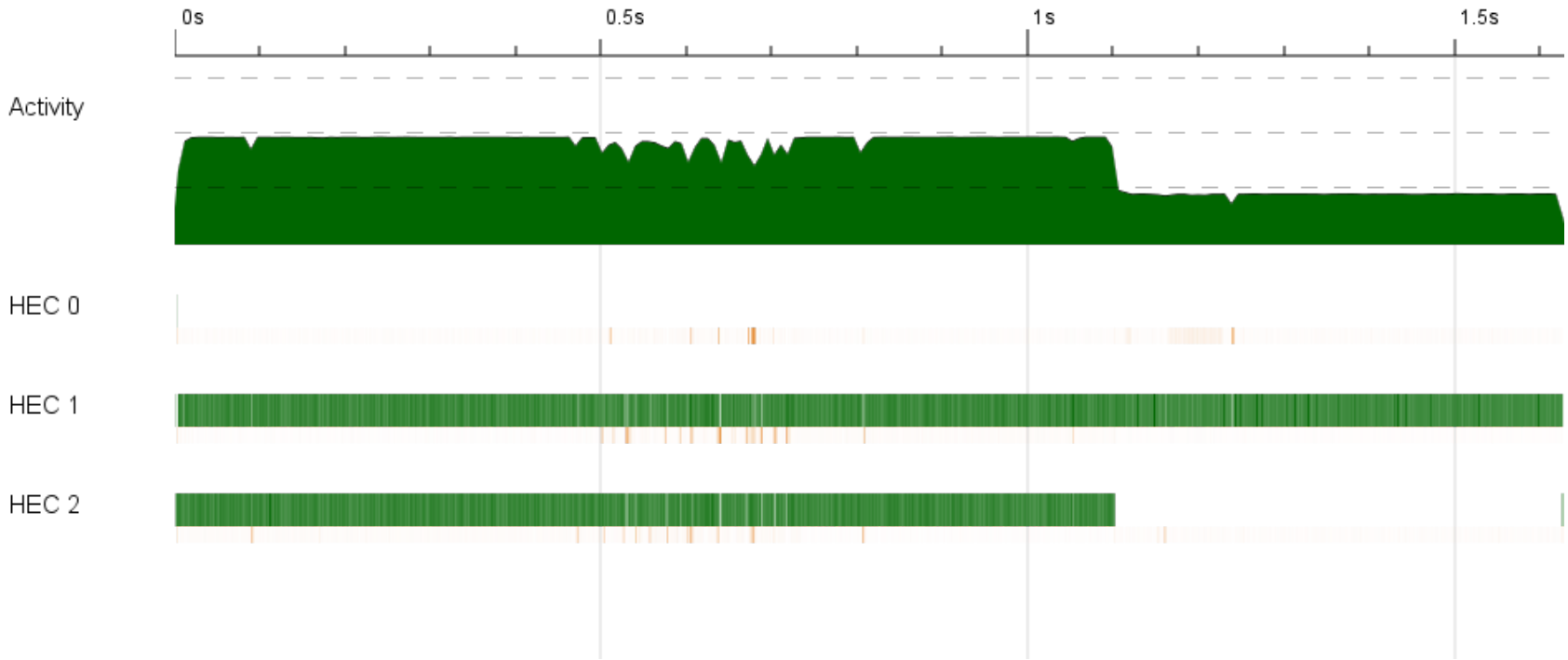
www.haskell.org/haskellwiki/ThreadScope

Verwendung

```
> ghc MyPrg.hs -threaded -eventlog -rtsopts
> ./MyPrg +RTS -N4 -l
> threadscope MyPrg.eventlog
```

- Zuerst Programm mit Event-Logging zu übersetzen
- Dann mit Event-Logging ausführen
- Threadscope visualisiert dann das Event-Log

Threadscope



Threadscope zeigt uns hier, das anfangs zwei Kerne genutzt wurden (HEC1, HEC2), während der dritte Kern (HEC0) komplett ungenutzt blieb.
Arbeitsphasen sind in Grün dargestellt;
GarbageCollection in Orange.

Semi-explizite Parallelität: GpH

Glasgow parallel Haskell (GpH) 1996 Modul *Control.Parallel*

erweitert Haskell durch zwei Primitive:

```
par :: a -> b -> b
```

`x `par` e` erlaubt parallele Auswertung von `x` und `e`

- gibt Hinweis auf parallele Auswertung; nicht erzwungen
- erzeugt **Spark** für `x`; Sparks werden zu WHNF ausgewertet, falls ein Prozessor frei ist

```
pseq :: a -> b -> b
```

`x `pseq` e` definiert sequentielle Auswertung von `x` und `e`

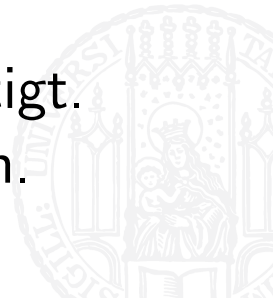
- `x` wird zur Weak Head Normal Form ausgewertet
- verbietet dem Kompilier einige Optimierungsmöglichkeiten im Vergleich zum konventionellen `seq` für strikte Auswertung

Beispiel: Fibonacci I

```
import Control.Parallel

fib :: Integer -> Integer
fib n | n < 2      = 1
      | otherwise = fib (n-1) + fib (n-2)

main = do
  let x = fib 41
  let y = fib 41
  let s = x `par` y `pseq` x+y
  print s
```

Beobachtungen:

Mit 2 Kernen wird nur noch ungefähr die Hälfte der Zeit benötigt.
Allerdings ist die Summe der Rechenzeit beider Kerne gestiegen.
$\Rightarrow$ Verwaltung der Sparks kostet ebenfalls Zeit!



Beispiel: Fibonacci II

```
pfib1 :: Integer -> Integer
pfib1 n | n < 2      = 1
        | otherwise = let fn1 = pfib1 (n-1)
                          fn2 = pfib1 (n-2)
          in fn1 `par` fn2 `pseq` fn1 + fn2

main = do
  let x = pfib1 41
  let y = pfib1 41
  let s = x+y
  print s
```

Beobachtungen:

Berechnung dauert nun plötzlich länger! Was ist passiert?

Beispiel: Fibonacci II

```
pfib1 :: Integer -> Integer
pfib1 n | n < 2      = 1
        | otherwise = let fn1 = pfib1 (n-1)
                          fn2 = pfib1 (n-2)
          in fn1 `par` fn2 `pseq` fn1 + fn2

main = do
  let x = pfib1 41
  let y = pfib1 41
  let s = x+y
  print s
```

Beobachtungen:

Berechnung dauert nun plötzlich länger! Was ist passiert?

$\Rightarrow$ Zuviel Overhead durch zu viele Sparks!

Sparks

Ein **Spark** steht für eine *mögliche* parallele Berechnung. Zur Laufzeit gibt es mehrere Möglichkeiten für einen Spark:

Konvertiert engl. converted Spark ausgerechnet

Abgeschnitten engl. pruned Spark nicht ausgerechnet

- **Dud**: Spark war schon ein ausgerechneter Wert
- **Fizzled**: Inzwischen durch anderen Thread berechnet
- **Garbage Collected**: Keine Referenz zum Spark vorhanden, d.h. Wert wird gar nicht mehr benötigt
- **Overflowed**: Spark wurde gleich verworfen, da der Ringpuffer der Sparks momentan voll ist

Anzeige der Laufzeit Statistik mit:

```
> ghc Program.hs -make -O2 -threaded -rtsopts
> ./Program +RTS -N3 -s
```

Idealerweise sollten deutlich mehr Spark konvertiert als abgeschnitten werden!

Sparks

Ein **Spark** steht für eine *mögliche* parallele Berechnung. Zur Laufzeit gibt es mehrere Möglichkeiten für einen Spark:

Konvertiert engl. converted Spark ausgerechnet

Abgeschnitten engl. pruned Spark nicht ausgerechnet

- **Dud**: Spark war schon ein ausgerechneter Wert
- **Fizzled**: Inzwischen durch anderen Thread berechnet
- **Garbage Collected**: Keine Referenz zum Spark vorhanden, d.h. Wert wird gar nicht mehr benötigt
- **Overflowed**: Spark wurde gleich verworfen, da der Ringpuffer der Sparks momentan voll ist

Anzeige der Laufzeit Statistik mit:

```
> ghc Program.hs -make -O2 -threaded -rtsopts
> ./Program +RTS -N3 -s
```

Idealerweise sollten deutlich mehr Spark konvertiert als abgeschnitten werden!

Problem: Granularität

Ein **Spark** ist sehr billig (deutlich einfacher als ein **Thread**), da der Spark-Pool lediglich ein Ringpuffer von Thunks ist. Es ist akzeptabel mehr Sparks einzuführen als letztendlich ausgeführt werden, da die Anzahl der verfügbaren Kerne ja variieren kann. Dennoch ist es schädlich, wenn zu viele Sparks angelegt werden (z.B. wenn die Ausführung des Sparks schneller geht als das Anlegen des Sparks selbst).

Problem für parallelles Rechnen

Die **Granularität**, also die Größe der einzelnen Aufgaben/Threads/Sparks sollte

- *nicht zu klein sein*, damit sich die parallele Behandlung gegenüber dem erhöhten Verwaltungsaufwand auch lohnt
- *nicht zu groß sein*, damit genügend Aufgaben für alle verfügbaren Kerne bereit stehen

Problem: GC'd Sparks

```
pfib1 :: Integer -> Integer
pfib1 n | n < 2      = 1
        | otherwise = pfib1 (n-1) `par`
                      pfib1 (n-2) `pseq`
                      (pfib1 (n-1)) + (pfib1 (n-2))
main = do
  let x = pfib1 41
  let y = pfib1 41
  let s = x+y
  print s
```

```
> stack ghc -- parfib01.hs -O2 -threaded -rtsopts
> ./parfib01 +RTS -N4 -s
...
  SPARKS: 866457019 (190014985 converted, 248820243 overflowed,
                                  0 dud, 427621790 GC'd, 1 fizzled)
```

Die Sparks werden sofort vom Garbage Collector wieder entfernt, da das Ergebnis des Sparks ja nicht mehr referenziert wird, sondern eine neue Berechnung gestartet wird.

Laziness und Parallele Auswertung

Ein weiteres Problem für parallele Auswertung kann durch die verzögerter Auswertung entstehen:

```
genlist :: Integer -> Integer -> [[Integer]]
genlist _ 0 = []
genlist x n = let f = fib x
                  h = (:) f []
                  t = genlist x (n-1)
              in h `par` t `pseq` h : t
main = do
  let l = genlist 38 6
  mapM_ print l
```

Die Liste wird zwar parallel zusammengebaut, aber Fibonacci-Berechnungen finden erst sequentiell bei bei der Bildschirmausgabe statt, da die parallele Auswertung nur bis zur WHNF erfolgte.

parfib10.hs

Laziness und Parallele Auswertung

Ein weiteres Problem für parallele Auswertung kann durch die verzögerter Auswertung entstehen:

```
genlist :: Integer -> Integer -> [[Integer]]
genlist _ 0 = []
genlist x n = let f = fib x
                  h = (:) f []
                  t = genlist x (n-1)
              in f `par` t `pseq` h : t  -- parallel
main = do
  let l = genlist 38 6
  mapM_ print l
```

Die Liste wird zwar parallel zusammengebaut, aber Fibonacci-Berechnungen finden erst sequentiell bei bei der Bildschirmausgabe statt, da die parallele Auswertung nur bis zur WHNF erfolgte.

parfib10.hs

Problem: Auswertegrad

Aufgrund der verzögerten Auswertung wird schnell unklar, wann der wesentliche Teil einer Berechnung ausgeführt wird.
Dies beeinflusst letztendlich die Granularität.

In einer Sprache mit verzögerter Auswertung muss sich der Programmierer daher explizit um den **Auswertegrad** und die **Auswertereihenfolge** der Daten kümmern, um die Granularität korrekt abschätzen zu können.

$\Rightarrow$ Primitive `par` und `pseq` in der Praxis schlecht brauchbar

Modul Control.Parallel.Strategies

Eine Lösungsmöglichkeit bietet das Modul `Control.Parallel.Strategies`.

Dort werden **Strategien** zur Koordination der parallelen Berechnung angeboten, um diese von der eigentlichen Berechnung zu trennen. Die Strategie legt Auswertegrad und -reihenfolge fest.

Die Abhängigkeiten verschiedener paralleler Berechnungen werden grundlegend durch eine Monade explizit ausgedrückt.

Die Komposition verschiedener paralleler Berechnungen erfolgt mit den üblichen monadischen Kombinatoren, z.B.

```
join     :: Monad m => m (m a) -> m a
sequence :: Monad m => [m a] -> m [a]
```

Parallele Berechnung als Monade

Modul `Control.Parallel.Strategies` definiert die Monade `Eval` zur Erzeugung von Sparks:

```
runEval :: Eval a -> a

rpar :: a -> Eval a -- kreiert Spark
rseq :: a -> Eval a -- wartet auf Ergebnis
```

- Drückt Abhängigkeiten zwischen parallelen Berechnungen aus: `m >>= f` erzwingt Berechnung von `m` vor `f`
- Komposition durch monadische Kombinatoren.
- Monade ist "rein", also keine Seiteneffekte/Kontext

$\Rightarrow$ Lediglich Koordination durch Abstraktion von `par` & `pseq`

Beispiel: Fibonacci III

```
import Control.Parallel
import Control.Parallel.Strategies

fib :: Integer -> Integer
fib n | n < 2      = 1
      | otherwise = fib (n-1) + fib (n-2)

main = do
  let myfib = fib
  let s = runEval $ do
            x <- rpar $ myfib 40
            y <- rseq $ myfib 38
            return $ (x+y)
  print s
```

parfib2.hs

Beispiel: Fibonacci-Liste II

```
import Control.Parallel
import Control.Parallel.Strategies

fib :: Integer -> Integer
fib n | n < 2      = 1
      | otherwise = fib (n-1) + fib (n-2)

genlist :: Integer -> Integer -> Eval [[Integer]]
genlist _ 0 = return []
genlist x n = do f <- rpar $ fib x
                 let h = [f]
                 t <- genlist x (n-1)
                 return $ h : t
main = do
  let l = runEval $ genlist 38 6
  mapM_ print l
```

Implementation der Eval Monade

```
data Eval a = Done a

instance Monad Eval where
  -- return :: a -> Eval a
  return x = Done x
  -- (>>=) :: Eval a -> (a -> Eval b) -> Eval b
  Done x >>= k = k x

runEval :: Eval a -> a
runEval (Done a) = a

rpar :: a -> Eval a
rpar x = x `par` return x

rseq :: a -> Eval a
rseq x = x `pseq` return x
```

$\Rightarrow$ Sehr simple Implementation der Monade!

Implementation der Eval Monade

```
data Eval a = Done a

instance Monad Eval where
  -- return :: a -> Eval a
  return x = Done x
  -- (>>=) :: Eval a -> (a -> Eval b) -> Eval b
  Done x >>= k = k x

runEval :: Eval a -> a
runEval (Done a) = a

rpar :: a -> Eval a
rpar x = x `par` return x

rseq :: a -> Eval a
rseq x = x `pseq` return x
```

Seit 7.2.1 ist die Implementation durch Verwendung von Primitiven geringfügig anders

⇒ Sehr simple Implementation der Monade!

Auswertestrategien

Folgende Typabkürzung erlaubt uns im Zusammenhang mit der Monade eine schöne Vereinfachung:

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

Anwendung einer Auswertestrategie erfolgt mit `using`:

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

Beispiel:

```
foo1 x y = someexpr
foo2 x y = someexpr `using` someParallelStrategy
```

Beide Funktionen sehen gleich aus, aber die zweite wird parallel ausgewertet!

Auswertestrategien mit Auswertungsgrad

Einfache Strategien zur Kontrolle von Auswertegrad, Auswertereihenfolge und Parallelität:

- `r0` führt keine Auswertung durch
- `rpar` führt Auswertung parallel durch
- `rseq` führt eine Auswertung zur WHNF durch (default)
- `rdeepseq` führt eine komplette Auswertung durch

```
r0 :: Strategy a
r0 x = Done x

rpar :: Strategy a
rpar x = x `par` Done x

rseq :: Strategy a
rseq x = x `pseq` Done x
```

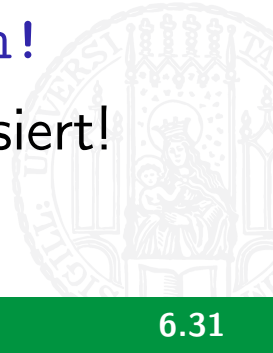
Beispiel: Strategie für Paare

```
evalTuple2 :: Strategy a -> Strategy b -> Strategy (a,b)
evalTuple2 sa sb (a,b) = (,) <$> sa a <*> sb b
```

```
parTuple2 :: Strategy (a,b)
parTuple2 = evalTuple2 rpar rpar

main = do
  let x = fib 40
  let y = fib 40
  let p = (x, y) `using` parTuple2
  print $ fst p + snd p  -- p muss verwendet werden!
```

Das Programm wird lediglich durch den Zusatz `using` parallelisiert!

Achtung: `p` muss verwendet werden, sonst GC'd.

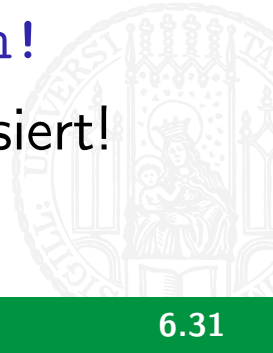


Beispiel: Strategie für Paare

```
evalTuple2 :: Strategy a -> Strategy b -> Strategy (a,b)
evalTuple2 sa sb (a,b) = do a' <- sa a
                            b' <- sb b
                            return (a',b')

parTuple2 :: Strategy (a,b)
parTuple2 = evalTuple2 rpar rpar

main = do
  let x = fib 40
  let y = fib 40
  let p = (x, y) `using` parTuple2
  print $ fst p + snd p  -- p muss verwendet werden!
```

Das Programm wird lediglich durch den Zusatz `using` parallelisiert!

Achtung: `p` muss verwendet werden, sonst GC'd.



Auswertestrategien kombinieren

Auswertestrategien können auch kombiniert werden:

Komposition

```
dot :: Strategy a -> Strategy a -> Strategy a
s2 `dot` s1 = s2 . runEval . s1
```

Parallele Anwendung auf Tupel

```
parTuple2' :: Strategy a -> Strategy b -> Strategy (a,b)
parTuple2' strat1 strat2 =
  evalTuple2 (rpar `dot` strat1) (rpar `dot` strat2)
```

Auswertestrategien kombinieren

Sequentielle Anwendung auf Listen

```
evalList :: Strategy a -> Strategy [a]
evalList --  :: (a -> Eval a) -> ([a] -> Eval [a])
evalList s [] = return []
evalList s (x:xs) = (:) <$> s x <*> evalList s xs
```

Parallele Anwendung auf Listen

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

Damit können wir schnell ein paralleles `map` definieren:

```
parMap :: (a -> b) -> [a] -> [b]
parMap f xs = map f xs `using` parList rseq
```

Auswertestrategien kombinieren

Sequentielle Anwendung auf Listen

```
evalList :: Strategy a -> Strategy [a]
evalList --  :: (a -> Eval a) -> ([a] -> Eval [a])
evalList s [] = return []
evalList s (x:xs) = (:) <$> s x <*> evalList s xs
```

Parallele Anwendung auf Listen

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

Damit können wir schnell ein paralleles `map` definieren:

```
parMap :: (a -> b) -> [a] -> [b]
parMap f xs = map f xs `using` parList rseq
```

Alternative für faule Listen, falls `parList` versagt:

```
parBuffer :: Int -> Strategy a -> Strategy [a]
```

Beispiel: parMap

```
import Control.Parallel
import Control.Parallel.Strategies

fib :: Integer -> Integer
fib n | n < 2      = 1
      | otherwise = fib (n-1) + fib (n-2)

main = do
  let l1 = [40,39..34]
  let l2 = map fib l1 `using` parList rseq
  print l2
```

Änderung des Programms zur parallelen Berechnung ist minimal!

Kontrolle des Auswertegrad

`r0`, `rseq` und `rdeepseq` regulieren Auswertegrad eines Ausdrucks

`rdeepseq` wertet vollständig zur Normal-Form (NF) aus; was mithilfe der Klasse `NFData` erreicht wird:

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

Für Basistypen reicht die Default-Implementierung. Weitere Instanzen definiert Modul `Control.Parallel.Strategies`

Eigene Instanzen sind leicht zu definieren:

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
instance NFData a => NFData (Tree a) where
  rnf Leaf = ()
  rnf (Branch l a r) = rnf l `seq` rnf a `seq` rnf r
```

Kontrolle des Auswertegrad: rdeepseq

Generische `NFData` Instanz für Listen:

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs
```

Während `seq` das erste Argument zu WHNF auswertet, wertet `deepseq` sein erstes Argument zur NF, also vollständig aus:

```
deepseq :: NFData a => a -> b -> b
deepseq a b = rnf a `seq` b
```

Die `rdeepseq` Auswertestrategie wird oft verwendet:

```
rdeepseq :: NFData a => Strategy a
rdeepseq x = x `deepseq` Done x
```

Semi-explizite Parallelität mit GpH

Zusammenfassung GpH

- Paket `parallel` ggf.: `cabal install parallel`
- Erlaubt parallele Auswertung von Teilausdrücken/Thunks
- Verwaltung der Parallelität erfolgt im Laufzeitsystem
  Overhead ist geringer als in anderen Ansätzen
- Programmierer entscheidet über *Auswertereihenfolge*
- Programmierer entscheidet über *Auswertegrad*
- Kombinierbare Auswertestrategien erfassen
  Auswertereihenfolge und Auswertegrad
- Funktionaler Code mit wenigen Änderungen parallelisierbar!
  Korrektheit bleibt unverändert, so fern ein Wert geliefert wird!

Par-Monade

Alternative: **Par**-Monade aus Modul `Control.Monad.Par`

ggf.: `stack install monad-par`

Vorteile

- Nachdenken über Lazy Evaluation entfällt komplett
- Kümmert sich um globales Scheduling
- Berechnung bleibt voll deterministisch

  $\Rightarrow$ Es kommt immer der gleiche Wert heraus

Nachteile

- Deutlich teurer Overhead im Vergleich zu GpH-Sparks
- Erlaubt nur Parallelität, aber keine Nebenläufigkeit
- Es können immer noch Deadlocks auftreten
- Innerhalb `Par` darf kein IO geschehen wäre sonst nebenläufig!

Explizite Synchronisation

Synchronisation erfolgt explizit durch Programmierer.
Dazu werden `IVar`-Referenzen bereitgestellt:

- `new :: Par (IVar a)`
  erzeugt Referenz auf leere Speicherstelle eines konkreten Typs
- `put :: NFData a => IVar a -> a -> Par ()`
  beschreibt die Speicherstelle. Dies kann nur einmal ausgeführt werden! Ein zweiter Schreibversuch führt zu einer Ausnahme!
- `get :: IVar a -> Par a`
  liest Speicherstelle aus und wartet ggf. bis ein Wert vorliegt

Achtung:

- Es können jetzt natürlich Deadlocks auftreten!
- `IVar`-Referenzen dürfen keinesfalls zwischen verschiedenen `Par`-Monaden herumgereicht werden!
  Kein forall-Trick wie in `ST` Monade $\Rightarrow$ `lvish` package

Beispiel: IVar Verwendung

```
example :: a -> Par (b,c)
example x = do
    vb <- new
    vc <- new
    --  Parallele Berechnungen werden nun gestartet
    --  und befüllen vb and vc mit Aufruf an put
    rb <- get vb
    rc <- get vc
    return (rb,rc)
```

- `new`-Befehle erzeugen leere Speicherstellen, deren Namen `vb` und `vc` an die parallelen Berechnungen übergeben werden.
- Innerhalb der parallelen Berechnungen erfolgen die Aufrufe `put vb somevalue` und `put vc othervalue` welche die leeren Speicherstellen unabhängig voneinander füllen.
- `get`-Befehle **synchronisieren**: mit dem Auslesen wird gewartet, bis die Speicherstelle gefüllt ist.

Beispiel: IVar Verwendung

```
new :: Par (IVar t)
get :: IVar t -> Par t
```

```
example :: a -> Par (b,c)
example x = do
    vb <- new
    vc <- new
    --  Parallele Berechnungen werden nun gestartet
    --  und befüllen vb and vc mit Aufruf an put
    rb <- get vb
    rc <- get vc
    return (rb,rc)
```

- `new`-Befehle erzeugen leere Speicherstellen, deren Namen `vb` und `vc` an die parallelen Berechnungen übergeben werden.
- Innerhalb der parallelen Berechnungen erfolgen die Aufrufe `put vb somevalue` und `put vc othervalue` welche die leeren Speicherstellen unabhängig voneinander füllen.
- `get`-Befehle **synchronisieren**: mit dem Auslesen wird gewartet, bis die Speicherstelle gefüllt ist.

Beispiel: IVar Verwendung

```
new :: Par (IVar t)
get :: IVar t -> Par t
```

```
example :: a -> Par (b,c)
example x = do
    vb <- new                          -- vb :: IVar b
    vc <- new                          -- vc :: IVar c
    --  Parallele Berechnungen werden nun gestartet
    --  und befüllen vb and vc mit Aufruf an put
    rb <- get vb
    rc <- get vc
    return (rb,rc)
```

- `new`-Befehle erzeugen leere Speicherstellen, deren Namen `vb` und `vc` an die parallelen Berechnungen übergeben werden.
- Innerhalb der parallelen Berechnungen erfolgen die Aufrufe `put vb somevalue` und `put vc othervalue` welche die leeren Speicherstellen unabhängig voneinander füllen.
- `get`-Befehle **synchronisieren**: mit dem Auslesen wird gewartet, bis die Speicherstelle gefüllt ist.

Beispiel: IVar Verwendung

```
new :: Par (IVar t)
get :: IVar t -> Par t
```

```
example :: a -> Par (b,c)
example x = do
    vb <- new                          -- vb :: IVar b
    vc <- new                          -- vc :: IVar c
    --  Parallele Berechnungen werden nun gestartet
    --  und befüllen vb and vc mit Aufruf an put
    rb <- get vb                       -- rb :: b
    rc <- get vc                       -- rc :: c
    return (rb,rc)
```

- `new`-Befehle erzeugen leere Speicherstellen, deren Namen `vb` und `vc` an die parallelen Berechnungen übergeben werden.
- Innerhalb der parallelen Berechnungen erfolgen die Aufrufe `put vb somevalue` und `put vc othervalue` welche die leeren Speicherstellen unabhängig voneinander füllen.
- `get`-Befehle **synchronisieren**: mit dem Auslesen wird gewartet, bis die Speicherstelle gefüllt ist.

Beispiel: fork

```
example :: a -> Par (b,c)
example x = do
    vb <- new                          -- vb :: IVar b
    vc <- new                          -- vc :: IVar c
    fork $ put vb $ taskB x            -- taskB :: a -> b
    fork $ put vc $ taskC x            -- taskC :: a -> c
    rb <- get vb                       -- rb == taskB x
    rc <- get vc                       -- rc == taskC x
    return (rb,rc)
```

- `fork :: Par () -> Par ()`
  startet übergebene Berechnung parallel zum aktuellen Thread.
- Typ `Par ()` sagt: parallele Berechnungen haben kein Ergebnis!
  Also müssen Ergebnisse als *Seiteneffekte* in der `Par`-Monade übergeben werden — durch Beschreiben von `IVar`-Variablen, der einzige erlaubte Seiteneffekt in der `Par`-Monade.

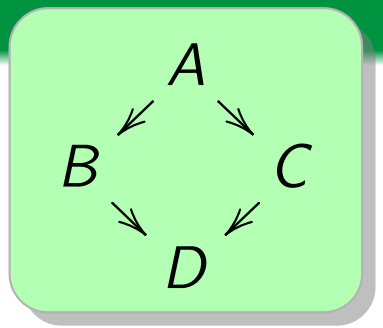
Beispiel: fork

```
put ::IVar t -> t -> Par ()
```

```
example :: a -> Par (b,c)
example x = do
    vb <- new                          -- vb :: IVar b
    vc <- new                          -- vc :: IVar c
    fork $ put vb $ taskB x            -- taskB :: a -> b
    fork $ put vc $ taskC x            -- taskC :: a -> c
    rb <- get vb                       -- rb == taskB x
    rc <- get vc                       -- rc == taskC x
    return (rb,rc)
```

- `fork :: Par () -> Par ()`
  startet übergebene Berechnung parallel zum aktuellen Thread.
- Typ `Par ()` sagt: parallele Berechnungen haben kein Ergebnis!
  Also müssen Ergebnisse als *Seiteneffekte* in der `Par`-Monade übergeben werden — durch Beschreiben von `IVar`-Variablen, der einzige erlaubte Seiteneffekt in der `Par`-Monade.

Beispiel: implizite Abhängigkeiten



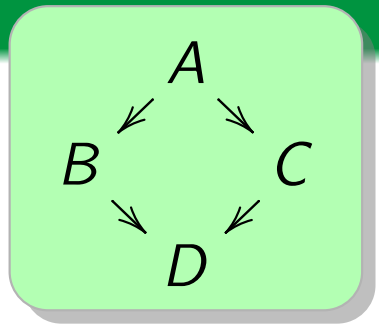
Damit können wir parallele Berechnungen bauen.
Abhängigkeiten können implizit ausgedrückt werden:

```
example2 :: Par d
example2 = do
  va <- new; vb <- new; vc <- new; vd <- new
  fork $ put va taskA                          -- A
  fork $ do ra <- get va; put vb $ taskB ra    -- B
  fork $ do ra <- get va; put vc $ taskC ra    -- C
  fork $ do rb <- get vb
            rc <- get vc
            put vd $ taskD rb rc -- D
  get vd
```

```
taskA :: a
taskB :: a -> b
taskC :: a -> c
taskD :: b->c->d
```

- Reihenfolge der `fork`-Anweisungen ist egal!
  Abhängigkeiten werden bereits implizit durch Lesen und Schreiben der `IVar`-Variablen ausgedrückt!

Beispiel: implizite Abhängigkeiten



Damit können wir parallele Berechnungen bauen.
Abhängigkeiten können implizit ausgedrückt werden:

```
example2 :: Par d
example2 = do
  va <- new; vb <- new; vc <- new; vd <- new
  fork $ do ra <- get va; put vc $ taskC ra     -- C
  fork $ do ra <- get va; put vb $ taskB ra     -- B
  fork $ do rb <- get vb                        -- D
            rc <- get vc
            put vd $ taskD rb rc
  fork $ put va taskA    -- A
  get vd
```

```
taskA :: a
taskB :: a -> b
taskC :: a -> c
taskD :: b->c->d
```

- Reihenfolge der `fork`-Anweisungen ist egal!
  Abhängigkeiten werden bereits implizit durch Lesen und Schreiben der `IVar`-Variablen ausgedrückt!

Beispiel: Deadlock

```
deadlockExample :: Par (b,c)
deadlockExample = do
  vb <- new
  vc <- new
  fork $ do rc <- get vc; put vb $ task1 rc
  fork $ do rb <- get vb; put vc $ task2 rb
  get vb
  get vc
  return (vb,vc)
```

```
task1 :: c -> b
task2 :: b -> c
```

- `task1` wird erst ausgeführt, wenn `vc` gefüllt ist;
  `task2` wird erst ausgeführt, wenn `vb` gefüllt ist.
  Beide Threads warten also zuerst darauf, dass der andere fertig wird. Die Berechnung kommt zum Stillstand: **Deadlock**!
- Mehrfache `get`-Aufrufe mit gleicher `IVar` sind unproblematisch.

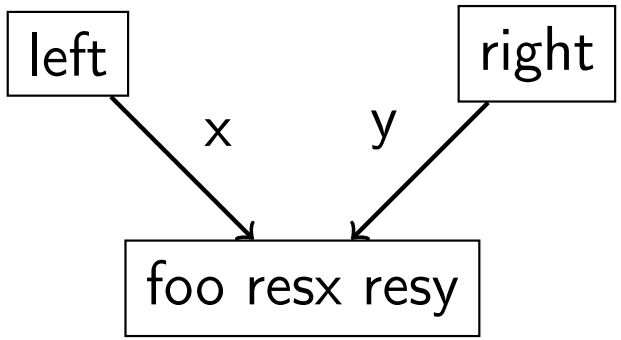
Übersicht Par-Monade

```
runPar   :: Par a  ->    a
runParIO :: Par a  -> IO a

fork     :: Par () -> Par ()

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

- `runPar` führt die Monade aus
- `fork` startet parallele Auswertung
- `new` erzeugt Postfach für Synchronisation
- `put` erzwingt die volle Auswertung seines Argumentes
- `get` wartet bis der Wert verfügbar ist
- `IVar` Variablen müssen *lokal* pro `runPar` sein und dürfen nur *einmal beschrieben* werden; Laufzeitfehler sonst

Beispiel

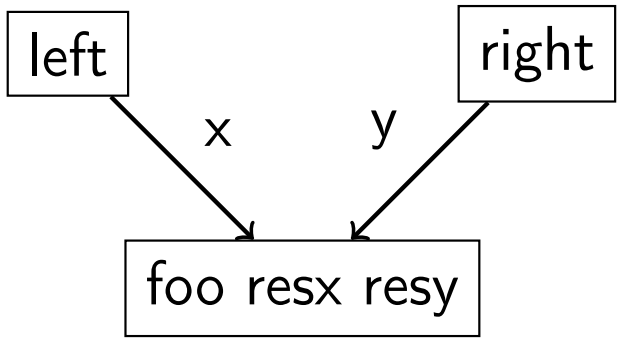
```
foo = runPar $ do
        x <- new
        y <- new
        fork $ put x left
        fork $ put y right
        resx <- get x
        resy <- get y
        return $ foo resx resy
where
  left    = ...Ausdruck mit aufwändiger Auswertung...
  right   = ...Ausdruck mit aufwändiger Auswertung...
  foo a b = ..Berechnung abhängig von beiden Werten..
```



1. `IVar` Variablen anlegen
2. Parallele Berechnung starten
3. Auf Ende der Berechnung mit `get` warten

Beispiel

```
foo = runPar $ do
        x <- spawnP  left
        y <- spawnP  right


        resx <- get x
        resy <- get y
        return $ foo resx resy
  where
    left    = ...Ausdruck mit aufwändiger Auswertung...
    right   = ...Ausdruck mit aufwändiger Auswertung...
    foo a b = ..Berechnung abhängig von beiden Werten..
```

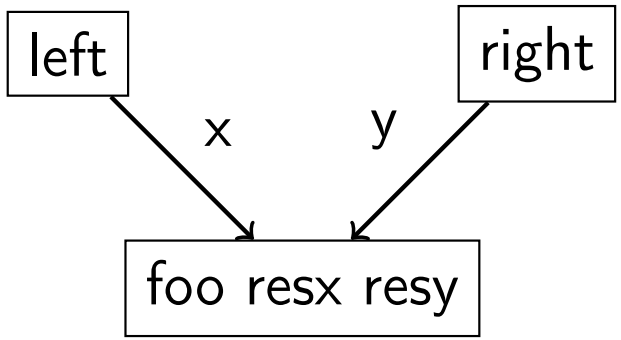



```
spawnP :: NFData a => a -> Par (IVar a)
```

Abkürzung zum Anlegen und Ausführen einer parallelen Berechnung

Beispiel

```
foo = runPar $ do
        x <- spawnP  left
        y <- spawnP  right

        foo <$> get x <*> get y
  where
    left    = ...Ausdruck mit aufwändiger Auswertung...
    right   = ...Ausdruck mit aufwändiger Auswertung...
    foo a b = ..Berechnung abhängig von beiden Werten..
```




```
spawnP :: NFData a => a -> Par (IVar a)
```

Abkürzung zum Anlegen und Ausführen einer parallelen Berechnung

Paralleles Map in der Par-Monade

`spawn` führt eine Berechnung in `Par` parallel aus, und liefert einen Verweis auf das zukünftige Ergebnis:

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

spawnP :: NFData a => a -> Par (IVar a)
spawnP = spawn . return
```

Damit können wir nun wieder das parallele Map innerhalb der `Par`-Monade ausdrücken:

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

Parallelles Map in der Par-Monade

`spawn` führt eine Berechnung in `Par` parallel aus, und liefert einen Verweis auf das zukünftige Ergebnis:

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

spawnP :: NFData a => a -> Par (IVar a)
spawnP = spawn . return
```

Damit können wir nun wieder das parallele Map innerhalb der `Par`-Monade ausdrücken:

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

Pipelining

Explizites **paralleles Pipelining**:

Producer stellt eine Folge von Werten eines Typs bereit
Mapper transformiert Typ einzelner Werte (oft mehrere Mapper)
Consumer sammelt Werte auf

Alle drei Einheiten arbeiten parallel an je einem Element mit eigenem Tempo: Der Producer arbeitet so schnell er kann, Mapper/Consumer arbeiten sobald ein Element vorliegt.

Wir bauen uns nun eine solche explizit parallele Pipeline mit Hilfe der Par-Monade...

Pipelining: Producer

```
data IList a = Nil | Cons a (IVar (IList a))
type Stream a = IVar (IList a)

streamFromList :: NFData a => [a] -> Par (Stream a)
streamFromList xs = do
    outstrm <- new              -- Referenz auf Ergebnis
    fork $ loop xs outstrm      -- merkt Listen- & Stream-Position
    return outstrm
  where
    loop   []   var = put var Nil
    loop (x:xs) var = do
      tail <- new               -- Neue Ref.für nächstes Element
      put var (Cons x tail)     -- Auswertung von x erzwingen
      loop xs tail
```

Idee Da jede nicht-leere Liste in einer `IVar` endet, kann der Anfang verwendet werden, bevor das Ende konstruiert wurde.
Der *eine* `fork` lässt den Producer in einem eigenen Thread laufen.

Pipelining: Mapper

```
data IList a = Nil | Cons a (IVar (IList a))
type Stream a = IVar (IList a)

streamMap :: NFData b => (a -> b) -> Stream a -> Par (Stream b)
streamMap fn instrm = do
    outstrm <- new
    fork $ loop instrm outstrm      -- Merkt In-/Out-Stream Pos.
    return outstrm
  where
    loop vin vout = do
      ilist <- get vin              -- Warte hier auf Eingabe
      case ilist of
        Nil             -> put vout Nil
        Cons h oldtail -> do
          newtail <- new
          put vout $ Cons (fn h) newtail    -- Auswertung fn h
          loop oldtail newtail
```

Der `fork` führt gesamten Mapper parallel in eigenen Thread aus.

Pipelining: Consumer

```
data IList a = Nil | Cons a (IVar (IList a))
type Stream a = IVar (IList a)

streamFold :: (a -> b -> a) -> a -> Stream b -> Par a
streamFold fn !acc instrm = do
  ilist <- get instrm                    -- Wartet hier auf Eingabe
  case ilist of Nil       -> return acc
                Cons h t -> streamFold fn (fn acc h) t
```

Führt selbst keinen Fork durch; ggf. mit `spawn` aufrufen, falls Hauptthread eigenständig weiter laufen soll.

Erinnerung: Bang-Pattern `!acc` erzwingt Auswertung zu WHNF, d.h. der Akkumulator wird vereinfacht, bevor das nächste Element verarbeitet wird.

Anwendung Pipeline

```
data IList a = Nil | Cons a (IVar (IList a))
type Stream a = IVar (IList a)

streamFromList ::                NFData a => [a] -> Par (Stream a)
streamMap :: NFData b => (a -> b) -> Stream a -> Par (Stream b)
streamFold ::  (b -> a -> b) -> b -> Stream a -> Par b

pipeline ::  NFData a => [a] -> b
pipeline input = runPar $ do
  s0 <- streamFromList input
  s1 <- streamMap expensiveFun1 s0
  s2 <- streamMap expensiveFun2 s1
  streamFold expensiveFold s2
```

Mögliches Problem: Producer arbeitet zu schnell und erzeugt gesamte Stream-Struktur im Speicher, was GC verlangsamt

Anwendung Pipeline

```
data IList a = Nil | Cons a (IVar (IList a))
type Stream a = IVar (IList a)

streamFromList ::                NFData a => [a] -> Par (Stream a)
streamMap :: NFData b => (a -> b) -> Stream a -> Par (Stream b)
streamFold ::  (b -> a -> b) -> b -> Stream a -> Par b

pipeline ::  NFData a => [a] -> b
pipeline input = runPar $
      streamFromList input
  >>= streamMap expensiveFun1
  >>= streamMap expensiveFun2
  >>= streamFold expensiveFold
```

Mögliches Problem: Producer arbeitet zu schnell und erzeugt gesamte Stream-Struktur im Speicher, was GC verlangsamt

Zusammenfassung Par-Monade

- Ausschließlich zur Beschleunigung der Berechnung durch paralleles Auswerten
- Berechnung mit `Par`-Monade ist deterministisch, d.h. liefert immer das gleiche Resultat — nur evtl. schneller
- IO-Operationen innerhalb `Par`-Monade nicht erlaubt
- Erheblich mehr interner Verwaltungsaufwand als bei GpH, d.h. die parallel ausgeführten Berechnungseinheiten sollten alle wesentlich größere Berechnungen durchführen

  ...da im Gegensatz zu Sparks alle...
- Parallele Einheiten werden immer vollständig ausgewertet
- Die `Par`-Monade kann jederzeit verwendet werden, ein Durchschleifen des Monaden-Typ ist nicht notwendig

  im Gegensatz zu `IO`
- Profiling etwas schwieriger
- Parallelität nur innerhalb eines `runPar`, mehrere `runPar` werden immer sequentiell ausgewertet.

Weitere Ansätze zur Parallelität in Haskell

- **Data Parallelism** Parallelität ist beschränkt auf gleichzeitiges Ausführen einer Operation auf großen Datenstrukturen, z.B. für Arrays, siehe Paket `Repa`.
  Solche Spezialfälle kommen häufig vor, weshalb solche speziellen Lösungen generischen Lösungen überlegen sind.
- **GPU Acceleration** Parallelität wird heute häufig mithilfe der dazu viel leistungsfähigeren Grafikkartenprozessoren erreicht, z.B. mit CUDA und OpenCL.
  Das Paket `Accelerate` stellt eine Schnittstelle bereit, um diese in Haskell zu nutzen.
  Auch hier dreht es sich primär um spezialisierte Operationen auf großen Datenstrukturen wie Arrays.