Programmierung und Modellierung mit Haskell

Teil 11: Parallele Auswertung

Steffen Jost

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

2. Juli 2018

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 11: Parallele Auswertung

Motivation

Das Thema **Paralleles Rechnen** ist sehr umfangreich. Mit dieser Vorlesung wollen wir lediglich einen *sehr kurzen* Einblick bieten.

Wichtig: **Paralleles Rechnen in Haskell muss nicht mit einer Monade abgewickelt werden!**

Wir behandeln hier ausschließlich die `Par`-Monade, da es unsere Hauptmotivation für dieses Kapitel ist, lediglich ein weiteres interessantes Beispiel für die Anwendung einer Monade zu haben.

Wer mehr über die Themen *Paralleles Rechnen* und *Nebenläufigkeit* in Haskell erfahren mag, kann die Vorlesung "Fortgeschrittene Funktionale Programmierung" in kommenden Wintersemester besuchen (6 ECTS "Vertiefende Themen der Informatik").

Grundlagen

Paralleles Rechnen: Ziel ist *schnelle* Ausführung von Programmen durch gleichzeitige Verwendung mehrerer Prozessoren.
Computer mit mehreren Kernen und Cloud-Architekturen sind inzwischen Standard und ermöglichen paralleles Rechnen.

Dagegen bedeutet **Nebenläufigkeit** **engl. Concurrency**
nicht-deterministische Berechnungen durch zufällig abwechselnd ausgeführte (interagierende) Prozesse.
Dies muss nicht unbedingt parallel erfolgen: man kann auf einem Prozessorkern verschiedene Prozesse in schneller Folge abwechseln, so dass z.B. eine GUI noch auf Benutzereingaben reagieren kann, während eine aufwendige Berechnung läuft; oder das ein Webserver Anfragen verschiedener Benutzer scheinbar gleichzeitig beantwortet.

Grundlagen

Parallele Berechnung ist schwierig:

Berechnung

effizienter Algorithmus zur Berechnung gesucht, der das gleiche Ergebnis wie bei sequentieller Berechnung liefert

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 unterbrochen, um alle Threads gleichmäßig 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

⇒ 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 in imperativen Sprachen 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 nicht mehr vorhersagbar.

Deadlock Ein Thread wartet auf das Ergebnis eines anderen Threads, welcher 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 oft leicht 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 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

  Aufruf mit `+RTS -qa` bestimmt Anzahl Kerne

- **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.
Das Programm muss speziell übersetzt werden:

```
> 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
```

```
                                                     individual     inherited
COST CENTRE                    MODULE          no.     entries  %time %alloc   %time %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
```

Runtime-system statistics

Für einen ersten Blick zur Performance eines Programmes reichen oft auch schon die 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
```

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

Die Par-Monade

`Par`-Monade aus Modul `Control.Monad.Par`
ggf. mit `stack install monad-par` installieren.

Vorteile

- Relativ simples Konzept mit durchschaubarer Beschleunigung
- Kümmert sich um globales Scheduling
- Berechnung bleibt garantiert voll deterministisch
  $\Rightarrow$ es kommt immer der gleiche Wert heraus

Nachteile

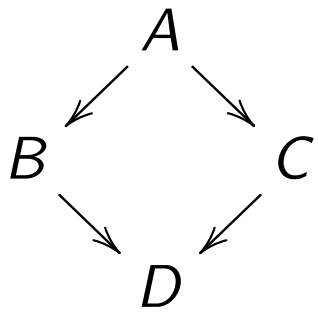
- Erlaubt nur Parallelität, aber keine Nebenläufigkeit
- Deutlich teurer Overhead im Vergleich zu anderen Ansätzen
- Es können immer noch Deadlocks auftreten
- Innerhalb `Par` darf kein IO geschehen klar: sonst nebenläufig!

Explizite Aufteilung der Arbeit

In der `Par`-Monade muss sich der Programmierer explizit um die Aufteilung der Arbeit in parallele Prozesse kümmern.
Der Programmierer muss auch Abhängigkeiten im Datenfluss der Berechnung explizit angeben.

Beispiel

Vier Berechnungen A, B, C, D:
B & C hängen vom Ergebnis von A ab. Berechnung D hängt von B & C ab. Lediglich B & C können parallel ausgeführt werden.



Zur Parallisierung verwenden wir die `Par`-Monade:
Typ `Par a` steht für eine Berechnung mit Endergebnis von Typ `a`, welche möglicherweise parallel ausgeführt wird.

Explizite Synchronisation

Kommunikation der Daten zwischen den parallelen Berechnungen werden mit "Postfächern" des Typs `IVar` realisiert:

- `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; wartet ggf. bis ein Wert vorliegt

Achtung:

- Dabei können Deadlocks auftreten!
- `IVar`-Referenzen dürfen keinesfalls zwischen verschiedenen `Par`-Monaden herumgereicht werden!
- `class NFData a where`
  `  rnf :: a -> ()`
  Klasse aller Typen, welche voll ausgewertet können. vgl. `seq`

Das Problem mit der Faulheit

Idee Mit `put :: IVar a -> a -> Par ()` aufwändige Berechnung von Typ `a` *parallel* durchführen, danach das Ergebnis in `IVar a` speichern.

Problem Wegen verzögerter Auswertung wird in `IVar a` nur ein Thunk (oder WHNF) gespeichert, dessen Auswertung dann erst später (nicht-parallel) erfolgen könnte!

Lösung Klasse aller Typen, welche voll ausgewertet können.

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

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

`put` erzwingt die vollständige Auswertung des Ergebnis noch während der parallelen Berechnung!

Das Problem mit der Faulheit

Idee Mit `put :: NFData a => IVar a -> a -> Par ()` aufwändige Berechnung von Typ `a` *parallel* durchführen, danach das Ergebnis in `IVar a` speichern.

Problem Wegen verzögerter Auswertung wird in `IVar a` nur ein Thunk (oder WHNF) gespeichert, dessen Auswertung dann erst später (nicht-parallel) erfolgen könnte!

Lösung Klasse aller Typen, welche voll ausgewertet können.

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

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

`put` erzwingt die vollständige Auswertung des Ergebnis noch während der parallelen Berechnung!

Das Problem mit der Faulheit

Idee: Mit put :: NFData a => IVar a -> a -> Par () au
da
P
T
sp
L

Instanzen von NFData wenden seq einfach auf alle Teile an:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
  deriving (Eq,Show)

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

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

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

put erzwingt die vollständige Auswertung des Ergebnis noch während der parallelen Berechnung!

Das Problem mit der Faulheit

Idee Mit put :: NFData a => IVar a -> a -> Par ()
au
da
P
T
sp
L

Instanzen von NFData wenden seq einfach auf alle Teile an:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
  deriving (Eq,Show)

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

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

GHC-Spracherweiterungen DeriveGeneric und DeriveAnyClass ermöglichen auch automatische Instanzdeklarationen für NFData mit deriving wie gewohnt.

put erzwingt die vollständige Auswertung des Ergebnis noch während der parallelen Berechnung!

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.

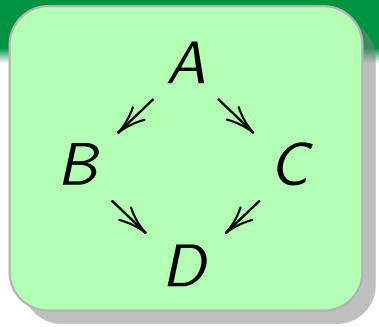
Beispiel: fork

```
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



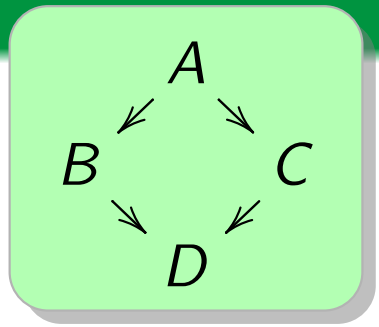
Damit können wir parallele Berechnungen bauen.
Abhängigkeiten können implizit ausgedrückt werden:

```
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.

Starten der Par-Monade

Zum tatsächlichen Ausführen einer `Par`-Berechnung gibt es:

```
runPar   :: Par a ->    a
runParIO :: Par a -> IO a     -- zum Aufruf aus main
```

Entspricht dieser Typ nicht `unpure :: Monad m => m a -> a`
was es laut Folie 9.44 nicht geben kann?!

Ja! Die `Par`-Monade ist ein Beispiel einer Monade, die man wieder "abschütteln" kann!

Das ist auch okay:
Einziger Seiteneffekt ist, dass die Berechnung parallel stattfindet.
Egal wie viele Prozessoren eingesetzt werden, für gleiche Argumente kommt garantiert immer das gleiche Ergebnis heraus!

Achtung: Ergebnistyp von `runPar` darf nie `IVar` enthalten!
Typsystem könnte das verhindern, wurde hier aber nicht eingesetzt.

Übersicht Par-Monade

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

fork     :: Par () -> Par ()

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

- `runPar` bzw. `runParIO` stößt die die Monade an
- `fork` weist parallele Auswertung einer Berechnung an
- `new` erzeugt "Postfach" zur Kommunikation zwischen parallelen Berechnungen; `IVar` nicht zwischen verschiedenen `runPar` tauschen
- `put` belegt "Postfach"; nur einmal pro Fach anwenden!
- `get` liest Fach aus; wartet ggf. bis der Wert verfügbar ist

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



        -- Applicative strikes again:
        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

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 ein paralleles `map` bauen:

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

mapM :: Monad m => (a -> m b) -> [a] -> m [b]
ivs  :: [IVar b]
```

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
```

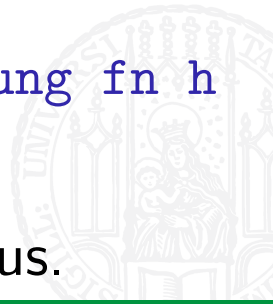
Idee Da jede nicht-leere Liste in einer `IVar` endet, kann der Anfang verwendet werden, bevor das Ende konstruiert wurde.
Der *eine* `fork` lässt den Producer in einem eigenen Thread laufen.

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 <- seq acc (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.

GHC Spracherweiterung `BangPatterns` erlaubt matching mit vorangestelltem Ausrufezeichen, hier `!acc`
Dies erzwingt Auswertung des Arguments zu WHNF, d.h. der Akkumulator wird vereinfacht, bevor das nächste Element der Pipeline erwartet wird.

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.

GHC Spracherweiterung `BangPatterns` erlaubt matching mit vorangestelltem Ausrufezeichen, hier `!acc`
Dies erzwingt Auswertung des Arguments zu WHNF, d.h. der Akkumulator wird vereinfacht, bevor das nächste Element der Pipeline erwartet 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 Garbage Collection verlangsamt $\Rightarrow$ nur jedes x-te Listenglied in `IVar` einpacken

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 Garbage Collection verlangsamt $\Rightarrow$ nur jedes x-te Listenglied in `IVar` einpacken

Zusammenfassung Par-Monade

- Ausschließlich zur Beschleunigung der Berechnung durch paralleles Auswerten.
- Seiteneffekt: Beschreiben von `IVar`-Variablen; weitere Seiteneffekte wie IO innerhalb der `Par`-Monade verboten
- Explizite Synchronisation über `IVar`-Variablen
- Deadlocks möglich.
- Berechnung mit `Par`-Monade ist deterministisch, d.h. liefert immer das gleiche Resultat nur evtl. schneller
- Programmierer muss über Aufteilung der Arbeit selbst entscheiden.
- Die `Par`-Monade kann jederzeit verwendet werden, ein Durchschleifen des Monaden-Typ ist nicht notwendig; aber empfehlenswert, da jeder Aufruf von `runPar` teuer ist

Weitere Ansätze zur Parallelität in Haskell

- **Glasgow Parallel Haskell**: Hinweise zur Partitionierung werden ins Programm eingestreut. Kontrolle der Parallelität erfolgt automatisch im Laufzeitsystem.
- **Software Transactional Memory:** Bibliothek erlaubt *spekulativ* parallele Berechnungen durchzuführen. Am Ende der Berechnung wird auf mögliche Konflikte mit anderen Berechnungen getestet ("lock-free"). Erlaubt Nebenläufigkeit.
- **forkIO**: klassische, explizite Nebenläufigkeit
- **Data Parallelism** Parallelität ist beschränkt auf gleichzeitiges Ausführen einer Operation auf großen Datenstrukturen, z.B. für Arrays Paket `Repa`.
- **GPU Acceleration** Parallelität mithilfe der leistungsfähigen Grafikkartenprozessoren erreicht, z.B. Paket `Accelerate` mit CUDA und OpenCL.