Concurrent programming in Haskell

Funktionale Programmierung – WS 2010

Florian Grabbe Philip Müller

fhwedel
UNIVERSITY OF APPLIED SCIENCES

10. Dezember 2010

Gliederung

Gliederung

Nebenläufigkeit

- Aufteilung eines Problems in verschiedenartige Teilprobleme
- Strukturierungsmethode
- Beispiel
- Nicht zwingend schneller als vollsequentielle Verarbeitung
- Sinnvoll auch ohne parallele Hardware (Timeslicing)

Parallelität

- Aufteilung eines Problems in gleichartige Teilprobleme
- Beispiel
- Ausnutzung paralleler Hardware zur Beschleunigung

Gliederung

Anwendungsbeispiel

- Vergleich von Mengen in Listendarstellung
- erst sortieren, dann vergleichen

Anwendungsbeispiel

- Vergleich von Mengen in Listendarstellung
- erst sortieren, dann vergleichen

Beispiel

```
eqList        :: (Ord a) => [a] -> [a] -> Bool
eqList xs ys = ssort xs == ssort ys
```

Parallelisieren mit `par`

- mehrere Berechnungen parallel ausführen
- Kennzeichnung mit `par` (in Modul `Control.Parallel`)
- „*semi explizite*“ Verteilung der Aufgaben

Parallelisieren mit `par`

- mehrere Berechnungen parallel ausführen
- Kennzeichnung mit `par` (in Modul `Control.Parallel`)
- „*semi explizite*“ Verteilung der Aufgaben

Signatur

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

Parallelisieren mit `par`

- mehrere Berechnungen parallel ausführen
- Kennzeichnung mit `par` (in Modul `Control.Parallel`)
- „*semi explizite*“ Verteilung der Aufgaben

Signatur

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

- erster Parameter kann parallel zum Zweiten ausgewertet werden
- gibt den Wert des zweiten Parameters zurück
- `a `par` b` ist semantisch äquivalent zu `b`

Anwendungsbeispiel mit par

Beispiel

```
import Control.Parallel

eqList          :: (Ord a) => [a] -> [a] -> Bool
eqList xs ys = b `par` (a == b)
               where a = ssort xs
                     b = ssort ys
```

Anwendungsbeispiel mit par

Beispiel

```
import Control.Parallel

eqList          :: (Ord a) => [a] -> [a] -> Bool
eqList xs ys = b `par` (a == b)
               where a = ssort xs
                     b = ssort ys
```

Problem: Auswertungsreihenfolge nicht festgelegt

seq und pseq

- erster Parameter wird zuerst ausgewertet
- Wert des zweiten Parameters wird zurück gegeben

Signaturen

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

seq und pseq

- erster Parameter wird zuerst ausgewertet
- Wert des zweiten Parameters wird zurück gegeben

Signaturen

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

- `seq` ist in beiden Parametern strikt
- Compiler kann `a `seq `b` in `b `seq `a `seq `b` umwandeln

seq und pseq

- erster Parameter wird zuerst ausgewertet
- Wert des zweiten Parameters wird zurück gegeben

Signaturen

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

- `seq` ist in beiden Parametern strikt
- Compiler kann `a `seq `b` in `b `seq `a `seq `b` umwandeln
- Lösung: `pseq` ist nur im ersten Parameter strikt

Anwendungsbeispiel mit par und pseq

Beispiel

```
import Control.Parallel

eqList          :: (Ord a) => [a] -> [a] -> Bool
eqList xs ys = a `par` (b `pseq` (a == b))
               where a = ssort xs
                     b = ssort ys
```

Anwendungsbeispiel mit par und pseq

Beispiel

```
import Control.Parallel

eqList          :: (Ord a) => [a] -> [a] -> Bool
eqList xs ys = a `par` (b `pseq` (a == b))
               where a = ssort xs
                     b = ssort ys
```

Problem: keine vollständige Auswertung durch pseq

Normal form (NF)

Normal form

Form, in der ein Ausdruck vollständig ausgewertet ist

Normal form (NF)

Normal form

Form, in der ein Ausdruck vollständig ausgewertet ist

Beispiel

THUNK
(*THUNK*, *THUNK*)
("hallo", *THUNK*)
("hallo", 2:*THUNK*)
("hallo", 2:1:*THUNK*)
("hallo", 2:1:[]) ← normal form

Thunk

Ein noch nicht ausgewerteter Ausdruck

Weak head normal form (WHNF)

Weak head normal form

Form, in der ein Ausdruck noch nicht vollständig ausgewertet ist aber wenigstens der äußerste Konstruktor erreicht wurde

Weak head normal form (WHNF)

Weak head normal form

Form, in der ein Ausdruck noch nicht vollständig ausgewertet ist aber wenigstens der äußerste Konstruktor erreicht wurde

Beispiel

THUNK
(*THUNK*, *THUNK*) ← weak head normal form
("hallo", *THUNK*) ← weak head normal form
("hallo", 2:*THUNK*) ← weak head normal form
("hallo", 2:1:*THUNK*) ← weak head normal form
("hallo", 2:1:[])

Auswertung erzwingen

Beispiel

```
import Control.Parallel

eqList          :: (Ord a) => [a] -> [a] -> Bool
eqList xs ys = a `par` (b `pseq` (a == b))
               where a = force $ ssort xs
                     b = force $ ssort ys

force     :: [a] -> ()
force xs = go xs `pseq` ()
           where go []     = 1
                 go (y:ys) = go ys
```

Gliederung

forkIO

- Nebenläufigkeit mit Hilfe von Threads
- mit `forkIO` erzeugen (in Modul `Control.Concurrent`)

Signatur

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

- Thread wird sofort ausgeführt
- erzeugt *lightweight* Thread (kein OS-Thread)
- `ThreadId` repräsentiert ein *Handle* auf den Thread
- *main*-Thread wartet nicht

forkIO

Beispiel

```
import Control.Concurrent
import System.Directory

main = do
    forkIO (writeFile "xyz" "thread was here")
    v <- doesFileExist "xyz"
    print v
```

forkIO

Beispiel

```
import Control.Concurrent
import System.Directory

main = do
    forkIO (writeFile "xyz" "thread was here")
    v <- doesFileExist "xyz"
    print v
```

Ausgabe

?

Determinismus

Ausführungsreihenfolge

Die Laufzeitkomponente von GHC legt die Ausführungsreihenfolge der Threads nicht fest.

im Beispiel

zuerst `writeFile`

oder `doesFileExist ?`

Nutzen von Wartezeiten

Nebenläufige Datenverarbeitung

- Ein-/Ausgabe
- Netzwerkkommunikation
- Datenbankzugriffe
- ...

Beispiel

```
import Control.Concurrent (forkIO)
import Codec.Compression.GZip (compress)
import qualified Data.ByteString.Lazy as L

gzip f = L.writeFile (f ++ ".gz") . compress

main = do
    putStrLn "Enter a file to compress:"
    name <- getLine
    if null name
        then return ()
        else do
            content <- L.readFile name
            forkIO (gzip name content)
            main
```

Threadkommunikation über Immutables

- Unveränderliche Werte können sicher gemeinsam genutzt werden
- Aber: Kein Datenaustausch
- $\rightarrow$ Kommunikation zwischen Threads nicht möglich

Was tun, wenn Kommunikation gewünscht ist?

- Zustand eines Threads abfragen?
- Spiel-Beispiel

Was tun, wenn Kommunikation gewünscht ist?

- Zustand eines Threads abfragen?
- Spiel-Beispiel
- $\rightarrow$ Veränderbare Variablen

Veränderbare Variablen in Haskell?

- MVars sind keine Variablen wie in C/Java

Veränderbare Variablen in Haskell?

- MVars sind keine Variablen wie in C/Java
- Restriktionen geben gewisse Sequentialität vor
- Ein-Element-Warteschlange

Grundlegende Operationen

```
putMVar :: MVar a -> a -> IO ()

takeMVar :: MVar a -> IO a
```

Einsatzzwecke

- Nachrichten zwischen Threads versenden bzw. Daten austauschen
- als Semaphore

MVars angewandt

Ein einfaches Beispiel

```
import Control.Concurrent

communicate = do
  m <- newEmptyMVar
  forkIO $ do
    v <- takeMVar m
    putStrLn ("received " ++ show v)
  putStrLn "sending"
  putMVar m "wake up!"
```

Auf Thread-Ende warten

```
myForkIO :: IO () -> IO (MVar ())
```

Auf Thread-Ende warten

```
myForkIO :: IO () -> IO (MVar ())
```

```
import Control.Exception (finally)
myForkIO io = do
  mvar <- newEmptyMVar
  forkIO (io `finally` putMVar mvar ())
  return mvar
```

Weitere Operationen

```
newEmptyMVar :: IO (MVar a)

newMVar :: a -> IO (MVar a)

tryTakeMVar :: MVar a -> IO (Maybe a)
```

Sichere Modifikation von MVars

```
modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b
```

- `takeMVar`, Modifikation und `putMVar`
- Exception Handling

Sichere Modifikation von MVars

```
modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b
```

- `takeMVar`, Modifikation und `putMVar`
- Exception Handling
- $\rightarrow$ Kein Deadlock

Warteschlangen

- FIFO zur Thread-Kommunikation
- Kann Geschwindigkeitsunterschiede abpuffern (Producer-Consumer)
- Andere Restriktionen als bei MVars

```
newChan :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan :: Chan a -> IO a
```

Channel-Beispiel

```
import Control.Concurrent
import Control.Concurrent.Chan

main = do
  ch <- newChan
  forkIO $ do
    writeChan ch "hello world"
    writeChan ch "now i quit"
  readChan ch >>= print
  readChan ch >>= print
```

Vorsicht mit der Bedarfsauswertung!

```
import Control.Concurrent

notQuiteRight = do
  mv <- newEmptyMVar
  forkIO $ expensiveComputation mv
  someOtherActivity
  result <- takeMVar mv
  print result
```

Aus anderen Sprachen bekannte Probleme

- unbeschränkte FIFOs können „explodieren“
- Bei gemeinsam genutzten veränderbaren Daten lauern immer tückische Fehler
- Nebenläufigkeitsprobleme häufig schwer zu reproduzieren

Aus anderen Sprachen bekannte Probleme

Deadlock z.B. Wenn MVars als Semaphoren genutzt werden

Starvation z.B. `modifyMVar` mit sehr aufwändiger Funktion blockiert MVar

Lösung? Software Transactional Memory

Gliederung

Software Transactional Memory

- alternative Möglichkeit, Threads in Haskell zu synchronisieren
- „Transaktionen“ statt expliziter „locks“
- vermeiden von deadlocks
- spezielle Variablen: TVars
- atomare Blöcke

Software Transactional Memory

Ausschnitt aus `Control.Concurrent.STM`

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

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

STM Beispiel

b :: TVar Int

7

Thread 1:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben

Transaction log

TVar	gelesen	geschrieben

STM Beispiel

b :: TVar Int

7

Thread 1:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben
b		

Transaction log

TVar	gelesen	geschrieben
b		

STM Beispiel

b :: TVar Int

7

Thread 1:

```
atomically ( do
      v <- readTVar b
      writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
      v <- readTVar b
      writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben
b		

Transaction log

TVar	gelesen	geschrieben
b	7	

STM Beispiel

b :: TVar Int

7

Thread 1:

```
atomically ( do
     v <- readTVar b
     writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
     v <- readTVar b
     writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben
b	7	

Transaction log

TVar	gelesen	geschrieben
b	7	

STM Beispiel

b :: TVar Int

7

Thread 1:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben
b	7	8

Transaction log

TVar	gelesen	geschrieben
b	7	

STM Beispiel

b :: TVar Int

7

Thread 1:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben
b	7	8

Transaction log

TVar	gelesen	geschrieben
b	7	4

STM Beispiel

b :: TVar Int

8

Thread 1:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben

Transaction log

TVar	gelesen	geschrieben
b	7	4

STM Beispiel

b :: TVar Int

8

Thread 1:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben

Transaction log

TVar	gelesen	geschrieben
b		

STM Beispiel

b :: TVar Int

8

Thread 1:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben

Transaction log

TVar	gelesen	geschrieben
b	8	

STM Beispiel

b :: TVar Int

8

Thread 1:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben

Transaction log

TVar	gelesen	geschrieben
b	8	5

STM Beispiel

b :: TVar Int

5

Thread 1:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v+1)
)
```

Thread 2:

```
atomically ( do
    v <- readTVar b
    writeTVar b (v-3)
)
```

Transaction log

TVar	gelesen	geschrieben

Transaction log

TVar	gelesen	geschrieben

Gliederung

Echte vs. Pseudo-Parallelität

- Haskell Threads $\neq$ Betriebssystem-Threads
- Zwei verschiedene Haskell-Laufzeitumgebungen
- „Normal“ vs threaded
- Entscheidung *zur Link-Zeit* mit `-threaded`
- Threads und MVars viel billiger in „normaler“ Laufzeitumgebung

Threaded Runtime

- Auch die „threaded runtime“ nutzt standardmäßig nur einen OS-Thread
- Option `+RTS -N` erlaubt Nutzung mehrerer OS-Threads
- Anzahl verfügbarer OS Threads lässt sich auslesen (Beispiel)

Wahl der richtigen Laufzeitumgebung

- Linken mit oder ohne Unterstützung von OS-Threads?
- Threaded: Ausnutzung paralleler Hardware *aber* Overhead für Haskell-Threads
- Häufig wird ein Geschwindigkeitsvorteil vor allem durch das „Verstecken“ von IO-Wartezeiten erreicht. Dafür reicht die normale Laufzeitumgebung.
- Fazit: Nebenläufige Programme sind mit der threaded runtime nicht automatisch schneller.

Quellen


Bryan O'Sullivan, Don Stewart and John Goerzen
Real World Haskell.
O'Reilly, November 2008.


Simon Peyton Jones and Satnam Singh
A Tutorial on Parallel and Concurrent Programming in Haskell
Lecture Notes from Advanced Functional Programming
MS Research Cambridge, 2008.

- Don Stewart
  Multicore Haskell Now!
  Edinburgh, Scotland, September 2009.