Funktionale Programmierung

Parallele Auswertung

Andreas Abel und Steffen Jost

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

21. Mai 2012

Verwendung mehrerer Kerne in GHC

Anzahl der verwendeten Kerne in GHC einstellen mit

- **Kompiler-Parametern:**
  Für 32 Kerne kompilieren mit

  ```
  ghc prog.hs -threaded -with-rtsopts="-N32"
  ```

  Alternativ kompilieren mit

  ```
  ghc prog.hs -threaded -rtsopts
  ```

  und dann Aufrufen mit `./prog +RTS -N32`
- **Dynamisch im Programm** mit

  ```
  GHC.Conc.setNumCapabilities :: Int -> IO ()
  GHC.Conc.setNumCapabilities 32

  GHC.Conc.getNumCapabilities :: IO Int
  ```

  Wert kann zur Laufzeit nur erhöht, nie herabgesetzt werden!

Ansätze zur Parallelität

Ansätze zur Parallelität, welche Haskell-basiert unterstützt werden:

- **Semi-explizite Parallelität**: Im Programm muss lediglich die Partitionierung ausgedrückt werden. Die Kontrolle der Parallelität erfolgt automatisch im Laufzeitsystem.
- **Software Transactional Memory:** Sprache wird um Bibliothek erweitert, welche es erlaubt **spekulativ** parallele Berechnungen durchzuführen. Erst am Ende der Berechnung wird auf mögliche Konflikte mit anderen Berechnungen getestet ("lock-free").
- **Nested Data Parallelism:** Parallelität ist beschränkt auf gleichzeitiges Ausführen einer Operation auf (großen) Datenstrukturen, z.B. mittels Array-comprehensions. Diese können geschachtelt werden.

Semi-explizite Parallelität mit GpH

GpH: Glasgow parallel Haskell Control.Parallel

- erlaubt parallele Auswertung von Teilausdrücken/Thunks
- Programmierer entscheidet über **Auswertereihenfolge**
- Programmierer entscheidet über **Auswertegrad**
- Kombinierbare Auswertestrategien erfassen Auswertereihenfolge und Auswertegrad
- einfach Verwendung mit geringen Änderungen am Codes

Explizite Parallelität

`par` kann rein funktionalen Code einfach parallelisieren, aber:

- in vielen großen Anwendung findet man oft zustandsbasierten (monadischen) Code, den man parallel abarbeiten will
- manche Anwendung bestehen ganz natürlich aus einer Menge nebenläufiger Threads

$\Rightarrow$ **Software Transactional Memory**

Nebenläufigkeit bedeutet nicht automatisch die Verwendung mehrer Kerne, sondern strukturiert ein Programm in mehrere asynchron ablaufende Einheiten!

Concurrent Haskell

Concurrent Haskell (Modul: Control.Concurrent) bietet Bibliotheksfunktionen zum Erzeugen und zur Kontrolle von nebenläufigen Berechnungen (IO-Thread).

Im Gegensatz zu GpH sind diese IO-Threads explizite Objekte, die im Code kontrolliert werden.

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

`forkIO` erzeugt einen IO-Thread, der mittels `ThreadId` identifiziert wird.

Beispiel Nebenläufigkeit

```
import Control.Concurrent
x = 35
f = fib
sillyA = putStrLn $ "SillyA " ++ (show $ f (x+9))
sillyB = putStrLn $ "SillyB " ++ (show $ f (x+0))
sillyC = putStrLn $ "SillyC " ++ (show $ f (x-9))

main =
  do
    putStrLn "Creating Threads."
    forkIO sillyA
    forkIO sillyB
    forkIO sillyC
    putStrLn "Waiting for completion."
    threadDelay 15000000
    putStrLn "Done."
```

Beispiel Nebenläufigkeit

Achtung: Implizite Sychronisation durch gemeinsame Thunks!

```
import Control.Concurrent
x = fib 35
f = id
sillyA = putStrLn $ "SillyA " ++ (show $ f (x+9))
sillyB = putStrLn $ "SillyB " ++ (show $ f (x+0))
sillyC = putStrLn $ "SillyC " ++ (show $ f (x-9))

main =
  do
    putStrLn "Creating Threads."
    forkIO sillyA
    forkIO sillyB
    forkIO sillyC
    putStrLn "Waiting for completion."
    threadDelay 15000000
    putStrLn "Done."
```

Explizite Synchronisation: MVars

Kommunikation zwischen IO-Threads mit synchronisierten shared-memory Variablen **MVar** Control.Concurrent.MVar

- `MVar a` ist eine Speicherstelle des Typs `a`
- Zugriff innerhalb der IO-Monade möglich
- MVar kann leer oder gesetzt sein

Die zwei grundlegenden Operationen sind

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

Semantik:

	MVar Leer	MVar Gesetzt
takeMVar	Blockiert	liefert & leert MVar
putMVar	Setzt MVar	Blockiert

Explizite Synchronisation: MVars

Zahlreiche Operationen im Modul `Control.Concurrent.MVar` verfügbar:

`newEmptyMVar :: IO (MVar a)` erzeugte neue leere MVar

`newMVar :: a -> IO (MVar a)` erzeugte initialisierte MVar

`takeMVar :: MVar a -> IO a` nimmt nicht-leere MVar

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

`readMVar :: MVar a -> IO a` liest nicht-leere MVar

`swapMVar :: MVar a -> a -> IO a` tauscht MVar

`isEmptyMVar :: MVar a -> IO Bool` testet ob MVar leer ist

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

`tryPutMVar :: MVar a -> a -> IO Bool` non-blocking

Beispiel für MVars: Rendevous

```
threadA :: MVar Int -> MVar Double -> IO ()
threadA valueToSendMVar valueReceiveMVar = do -- work
    putMVar valueToSendMVar 46 -- perform rendevous
    v <- takeMVar valueReceiveMVar
    putStrLn (show v )

threadB :: MVar Int -> MVar Double -> IO ()
threadB valueToReceiveMVar valueToSendMVar = do -- work
    -- perform rendevous by waiting on value
    z <- takeMVar valueToReceiveMVar
    putMVar valueToSendMVar (1.5 * (fromIntegral z))

main = do
  aMVar <- newEmptyMVar
  bMVar <- newEmptyMVar
  forkIO (threadA aMVar bMVar )
  forkIO (threadB aMVar bMVar )
```

Explizite Synchronisation: MVars

MVars erlauben direkt

- gemeinsame Verwendung von Datenstrukturen, z.B. write-locks für Dateien
- Kommunikationskanäle ohne Buffer

Da MVars für beliebige Typen deklariert werden können, ist es relativ leicht möglich, Kommunikationskanäle mit unbegrenzten Buffer (FIFO Warteschlangen) zu erstellen.

Diese gibt es aber auch schon fertig: Control.Concurrent.Chan

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

Software Transactional Memory

Software Transactional Memory (`STM`) erlaubt die Ausführung von IO-Threads auf gemeinsamen Variablen (`TVar`) ohne diese bei Beginn der Benutzung zu sperren ("lock-free").

Am Ende der Benutzung wird getestet ob Konflikte auftraten.

In dem Fall wird die Berechnung "zurückgespult".

Dazu müssen alle Operationen auf `TVar` in der `STM` Monade ausgeführt werden.

STM Operationen

```
newtype STM a -- Monade welche atomare Speicherzugriffe
atomically :: STM a -> IO a -- STM Aktion atomar ausfueh
retry      :: STM a         -- STM Aktion wiederholen
orElse     :: STM a -> STM a -> STM a
 -- Komposition. Wert des ersten Arguments,
 -- falls kein retry auftrat, sonst zweites

data TVar a -- Speicherzelle fuer atomare Zugriffe
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM()
```

Beispiel für STM

```
-- in thread 1:
atomically $
  do
    v <- readTVar acc
    writeTVar acc (v + 1)

-- in thread 2:
atomically $
  do
    v <- readTVar acc
    writeTVar acc (v - 3)
```

Weitere Funktionen der STM Monade

Die Funktion *retry* ermöglicht es ein roll-back zu erzwingen. Dies ist hilfreich wenn im Programm erkannt wird, dass eine Transaktion nicht erfolgreich abgeschlossen werden kann.

Beispiel: abheben von einem leeren Konto

```
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n = do
  bal <- readTVar acc
  if bal < n
    then retry
    else writeTVar acc (bal - n)
```

Weitere Funktionen der STM Monade

Die Funktion *orElse* ermöglicht es eine Auswahl ("choice") zwischen 2 STM Transaktionen zu implementieren. Wenn eine Transaktion fehlschlägt, so wird die andere ausgeführt. Nur wenn beide Transaktionen fehlschlagen, schlägt auch die gesamte Transaktion fehl.

Beispiel: abheben von einem von zwei Konten

```
withdraw2 :: TVar Int -> TVar Int -> TVar Int -> Int ->
withdraw2 acc1 acc2 acc3 n =
  do
    (withdraw acc1 n
    `orElse`
    withdraw acc2 n)
    deposit acc3 n
```

Beispiel: Gemeinsame Warteschlange

Es soll eine Warteschlange ("queue") implementiert werden, die von mehreren Threads gemeinsam benutzt werden kann.

```
data Queue e = Queue { shead :: TVar Int,
                       stail :: TVar Int,
                       sa :: Array Int (TVar e)  }

enqueue :: Queue a -> a -> IO ()
enqueue q x =  atomically $
  do
    h <- readTVar (shead q)
    t <- readTVar (stail q)
    when (t==h-1) retry
    val <- writeTVar ((sa q)!t) x
    writeTVar (stail q) ((t+1) `mod` (len (sa q)))

len ary = let (lb,ub) = bounds ary in ub-lb
```

Beispiel: Gemeinsame Warteschlange

Es soll eine Warteschlange ("queue") implementiert werden, die von mehreren Threads gemeinsam benutzt werden kann.

```
data Queue e = Queue { shead :: TVar Int,
                       stail :: TVar Int,
                       sa :: Array Int (TVar e)  }

dequeue :: Queue a -> IO a
dequeue q =  atomically $
  do
    h <- readTVar (shead q)
    t <- readTVar (stail q)
    when (h==t) retry
    val <- readTVar ((sa q)!h)
    writeTVar (shead q) ((h+1) `mod` (len (sa q)))
    return val

len ary = let (lb,ub) = bounds ary in ub-lb
```

Zusammenfassung Explizite Parallelität

- Echte explizite Nebenläufigkeit
- Implizite Synchronisation über gemeinsame Teilausdrücke
- Explizite Synchronisation über `MVar` oder `Chan`
- "lock-free" Code auf gemeinsamen Variablen (`TVars`) in `STM` Monade
- Eignet sich zur Parallelisierung von monadischem Code

Nested Data Parallelism

Im "**nested data parallelism**" Ansatz wird nur eine Form von Parallelität unterstützt: Datenparallelität, d.h. eine Funktion wird parallel auf eine Menge (meist Liste) von Daten angewendet.

Dies schränkt die Art der Parallelität ein.

Durch die Möglichkeit, Datenparallelität zu verschachteln, erhält man dennoch mächtige Konstrukte zum Beschreiben paralleler Berechnungen.

Weiters bietet Datenparallelität ein einfaches Kostenmodell, das bei der Vorhersage der parallelen Performanz hilft.

Beispiel für Datenparallelität

Data Parallel Haskell add-on für GHC 7.4

via cabal install dph-examples

`[: :]` steht für **parallele Arrays**, welche wie strikte Listen behandelt werden. Keine induktive Verarbeitung möglich!

Summe der Quadrate:

```
sumSq :: [:Float:] -> Float
sumSq a = sumP [: x * x | x <- a :]
```

Vektor Multiplikation:

```
vecMul :: [:Float:] -> [:Float:] -> Float
vecMul a b = sumP [: x * y | x <- a | y <-b :]
```

Pros und Cons von Datenparallelität

- Für eine n Prozessor Maschine werden n Threads erzeugt, die jeweils ein Segment der Liste berechnen
- Dadurch erzeugt man gute Granularität
- Da immer benachbarte Listenelemente zusammengefasst werden, erzeugt man gute Datenlokalität.
- Da genauso viele Threads erzeugt werden wie Prozessoren vorhanden sind, ist die Lastverteilung sehr gut.

Beachte:

- Parallele Listen haben eine strikte Semantik.
- Diese Implementierung ist zunächst nur für flache Datenparallelität möglich.

Modul **Data.Array.Prallel.Prelude** stellt für viele gängigen Funktionen auf Listen parallele Varianten bereit:

```
emptyP     :: [:a:]
singletonP:: a -> [:a:]
(!:)       :: [:a:] -> Int -> a
(+:+)      :: [:a:] -> [:a:] -> [:a:]
lengthP    :: [:a:] -> Int
replicateP:: Int -> a -> [:a:]
mapP       :: (a -> b) -> [:a:] -> [:b:]
zipP       :: [:a:] -> [:b:] -> [:(a, b):]
unzipP     :: [:(a, b):] -> ([:a:], [:b:])
zipWithP   :: (a -> b -> c) -> [:a:] -> [:b:] -> [:c:]
```

Beispiel: Quicksort

```
qsort :: [:Double:] -> [:Double:]
qsort xs
  | lengthP xs 1 = xs -- kein Pattern Match
  | otherwise = rs !: 0 +:+ eq +:+ rs !: 1
 where
  p  = xs !: (lengthP xs `div` 2)
  lt = [:x | x <- xs, x <  p:]
  eq = [:x | x <- xs, x == p:]
  gt = [:x | x <- xs, x >  p:]
  rs = mapP qsort [:lt, gt:]
```

Beachte: Der divide-and-conquer Algorithmus wird durch Daten-Parallelität auf der Liste $[:lt, gt:]$ ausgedrückt.

Data Parallel Haskell

- Data Parallel Haskell bietet ein Modell **impliziter** Parallelität.
- Data Parallel Haskell ist eine Haskell Erweiterung, die verschachtelte Datenparallelität (in obiger Syntax) erlaubt.
- Autoren: Manuel Chakravarty, Gabriele Keller.
- Kern der Implementierung ist die Umwandlung von verschachtelter Daten-Parallelität in flache Daten-Parallelität ("vectorisation"), im 2 Phasen
  - Umwandlung von verschachtelten Arrays in flache Arrays mit primitiven Elementen. Dabei sollen verwandte Daten in benachbarten Speicherstellen zu finden sein.
  - Entsprechende Umwandlung des Codes ("code vectorisation"). Um Zwischen-Datenstrukturen zu vermeiden, wird Code-Fusion verwendet.
- Dies beruht auf Arbeiten von Guy Blelloch zu NESL.

Literatur

- S.L. Peyton Jones, S. Singh, "A Tutorial on Parallel and Concurrent Programming in Haskell" (see `http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/`)
- P.W. Trinder, K. Hammond, H-W. Loidl, S.L. Peyton Jones. "Algorithm + Strategy = Parallelism." *J. of Functional Programming*, 8(1):23–60, Jan 1998. (see `http://www.macs.hw.ac.uk/~dsg/gph/papers/`)
- P.W. Trinder, H-W. Loidl, R. Pointon. "Parallel and Distributed Haskells." *J. of Functional Programming*, 12(5):469–512, Jul 2002.
- GHC Users Guide, `http://www.haskell.org/ghc/docs/latest/html/users_guide`