Multicore-Parallelität in Haskell

Vortrag im Rahmen des Seminars *Multicore-Programmierung*

Stefan Boxleitner

Fakultät für Informatik und Mathematik
Universität Passau

20. Januar 2011

Gliederung

Concurrent Haskell

Concurrent Haskell

- explizites Erzeugen von Threads mit
  `forkIO :: IO () -> IO ThreadId`
- nur innerhalb der `IO`-Monade
- `forkIO x` startet einen neuen Thread, der sofort mit der Auswertung von `x` beginnt
  Beispiel:

```
let                                                    1
 loop ch = hPutChar stdout ch >> loop ch               2
in                                                     3
do { forkIO (loop 'a'); loop 'z' }                     4
```

- Nichtdeterminismus

Kommunikation

- Container `MVar a`: enthält Datum vom Typ `a` oder ist leer
- Blockierender Zugriff, nur von `IO` aus
  - `takeMVar :: MVar a -> IO a`
    `takeMVar m` blockiert bis `m` gefüllt, entfernt dann enthaltenes Datum und gibt es zurück
  - `putMVar :: MVar a -> a -> IO ()`
    `putMVar m x` blockiert bis `m` leer, füllt dann `m` mit `x`

Software Transaktionsspeicher

Nachteile Sperren-basierter Programme

- Fehleranfälligkeit durch explizites Setzen von Locks
  - zu wenige (Verlust der Konsistenz)
  - zu viele (Deadlocks)
- schlechte Skalierbarkeit:

```
//atomar                                    1
add() {...}                                 2
remove() {...}                              3
//nicht atomar                              4
c() {add(); remove();}                      5
```

lock-based programs do not compose

Lösung aus dem Datenbankdesign: *Transaktionen*

- Idee: Deklaration von Code-Blöcken als Transaktionen

```
atomar add() {...}
```
1

- Verwaltung durch Transaktionsmanager mit ausgereiften Strategien im Laufzeitsystem

Transaktionsmanagement

- Transaktionen werden parallel ausgeführt
- für jede Transaktion: Änderungen am System werden nur zwischengespeichert und in Transaktionslog mitgeschrieben
- Beim Abschluss einer Transaktion t: Überprüfung auf Konflikte
  - keine andere Transaktion, die von t gelesene Werte geschrieben hat, darf in der Zwischenzeit erfolgreich abgeschlossen worden sein.
  - wenn Bedingung erfüllt: Übernehmen der Änderungen von t
  - ansonsten: Neustart der Transaktion

Umsetzung in Haskell

- Ansatz ähnlich monadischem I/O
- gekapselt in `STM`-Monade
- Transaktionen dürfen zustandsbehaftete Operationen durchführen, aber nur wenn sie reversibel sind.
- Transaktionaler Speicher: `TVar`
  - nur innerhalb der STM-Monade

```
readTVar :: TVar a -> STM a                    1
writeTVar :: TVar a -> a -> STM ()             2
```

- Ausführung von Transaktionen nur aus der `IO`-Monade mit `atomic :: STM a -> IO a`

Beispiel: Ressourcenmanager

- Bestand ist repräsentiert durch Int

```
type Resource = TVar Int                                    1
```

- Operationen getR und putR

Implementierung von putR:

```
putR :: Resource -> Int -> STM ()                           1
putR r i = do { v <- readTVar r                             2
              ; writeTVar r (v+i) }                         3
```

Blockierende Transaktionen

- Mechanismus zum Warten auf bestimmte Ereignisse
- `retry :: STM a`
- expliziter Abbruch und Neustart der Transaktion
- Neustart nur, wenn Zustand der bis dato gelesenen `TVars` verändert wurde

Im Beispiel:

```
getR :: Resource -> Int -> STM ()                          1
getR r i = do { v <- readTVar r                            2
              ; if (v < i) then retry                      3
                else writeTVar r (v-i) }                   4
```

Sequentielle Komposition

Hintereinanderreihung von Transaktionen in der do-Notation

```
tSeq = do { t1 ; t2 }
```
1

- Ausführen von `t1`
- bei Blockieren von `t1`: Blockieren von `tSeq`
- Ausführen von `t2`
- bei Blockieren von `t2`: Blockieren von `tSeq`

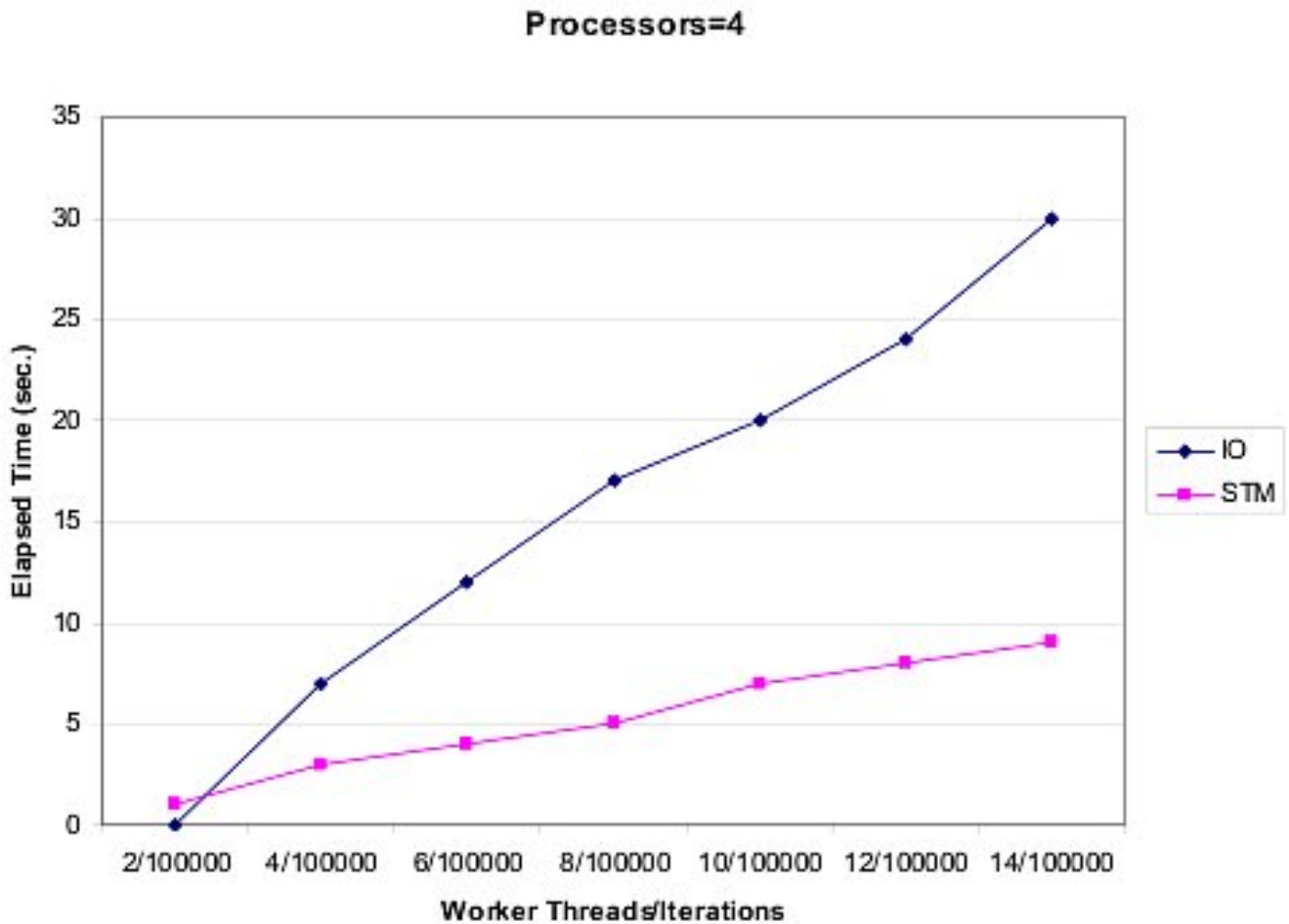
Alternativen

Kombinator `orElse`, z.B.

```
tSeq = do { t1 `orElse` t2 }
```
1

- Ausführen von `t1`
- bei Blockieren von `t1`: Ausführen von `t2`
- bei Blockieren von `t2`: Blockieren von tSeq
- Neustart erst, wenn gelesene `TVars` von `t1` *oder* `t2` modifiziert wurden

Performanz: Locks gegen STM



Glasgow Parallel Haskell

Glasgow Parallel Haskell

neue Primitiven

- `par :: a -> b -> b`
  - `par x y` erzeugt einen Spark für x im Spark-Pool und gibt y zurück
  - unausgelastete Threads holen sich Elemente aus dem Spark-Pool und werten sie aus.
- `pseq :: a -> b -> b`
  - `pseq x y` wertet zuerst x nach WHNF aus und gibt dann y zurück

Beispiel: Fibonacci-Funktion

```
fib :: Int -> Int                                    1
fib n                                                2
  | n <= 1      = 1                                  3
  | otherwise = let                                  4
                      x = fib (n-1)                  5
                      y = fib (n-2)                  6
                  in                                 7
                      x `par` y `pseq` x + y         8
```

Strategien

- bilden Abstraktionsschicht über `par` und `pseq`
- werten einen Ausdruck bis zu einem bestimmten Grad aus
- ermöglichen das Bestimmen einer Auswertungsreihenfolge
- grundsätzliche Strategien:
  - `r0`
  - `rseq`
  - `rpar`
  - `rdeepseq :: NFData a => Strategy a`

Umsetzung

- `Eval` - die Auswertungsreihenfolgen-Monade
- Anwendung von Strategien nur innerhalb der `Eval`-Monade
- entfernen des `Eval`-Kontextes durch
  `runEval :: Eval a -> a`
- praktische und flexible Notation zur Festlegung der Auswertungsreihenfolge
- `Eval`-Monade als eingebettete domänenspezifische Sprache (EDSL) zur Bestimmung der Auswertungsreihenfolge

Beipiel: Fibonacci-Funktion angepasst

```
runEval $ do                          1
  x <- rpar (fib (n-1))               2
  y <- rseq (fib (n-2))               3
  return x + y                        4
```

Komponierte Strategien

- Kompositionsoperator `dot :: Strategy a -> Strategy a -> Strategy a`
- Kombinator `evalList :: Strategy a -> Strategy [a]`
- `parList` kann aus `evalList` und `rpar` komponiert werden:

```
parList :: Strategy a -> Strategy [a]                    1
parList s = evalList (rpar `dot` s)                      2
```

Data Parallel Haskell

(Flache) Datenparallelität

- eine sequentielle Funktion wird parallel auf eine Menge von Werten angewendet
- Auf aktuelle Architekturen wie Multicore, MIMD, GPGPU, etc. optimal abbildbar
- gute Granulariät
- Daten-Lokalität
- gute Lastverteilung

Verschachtelte Datenparallelität

- Code enthält verschachtelte Datenstrukturen oder Funktionen
- dadurch komplexe Paradigmen, wie Divide-and-Conquer umsetzbar
- klar strukturiert
- komponierbar
- nicht ohne Weiteres auf moderne Architekturen abbildbar

Data Parallel Haskell

Idee:

- Schreiben von Programmen verschachtelter Datenparallelität
- Automatische Transformation in Programme flacher Datenparallelität

Programmieren mit Data Parallel Haskell

- neuer Typ `[:e:]`: Parallel Array vom Typ `e`
- parallele Operationen für Parallel Arrays; Benennung angelehnt an entsprechende Operationen für Listen
- Comprehensions für Parallel Arrays

Beispiel: Wortsuche

```
type Document = [: String :]
type DocColl = [: Document :]

wordOccs :: Document -> String -> [:Int:]
wordOccs d s = [: i | (i,s2) <- zipP [:1..lengthP d:] d
                    , s == s2 :]

search :: [: Document :] -> String
          -> [: (Document, [:Int:]) :]
search ds s = [: (d,is) | d <- ds
                , let is = wordOccs d s
                , not (nullP is) :]
```

Repräsentation von Daten

flache Parallel Arrays

```
data instance [: Int    :] = PI Int ByteArray      1
data instance [: Float  :] = PF Int ByteArray      2
data instance [: Double :] = PD Int ByteArray      3
```

Arrays von Paaren

```
data instance [: (a,b) :] = PP [:a:] [:b:]         1
```

Arrays von Arrays

```
data instance [: [:a:] :] = PA [:Int:] [:a:]       1
```

Beispiel: dünnbesetzte Matrix

```
type SparseVector = [: (Int, Float) :]                    1
type SparseMatrix = [: SparseVector :]                    2
```

Eine Instanz davon könnte so aussehen:

```
m :: SparseMatrix                                         1
m = [: [:(1,2.0), (7,1.9):], [:(3,3.0:] :]                2
```

Repräsentation in DPH:

```
PA [:0,2] (PP [:1,   7,   3  :]                           1
              [:2.0, 1.9, 3.0:])                          2
```

Vektorisierung von Funktionen

- Nach *Desugaring*: viele Ausdrücke der Form `mapP f`
- Idee: *Lifting* der Funktionen

```
f :: Float -> Float                    1
fL :: [:Float:] -> [:Float:]           2
```

- Ersetzen der Ausdrücke `mapP f` durch `fL`
- Regeln (stark vereinfacht):
  - Konstanten durch `replicateP` ersetzen
  - Funktionen durch ihrer gelifteten Versionen ersetzen, z.B. + wird zu +L
  - Parameter bleiben

Beispiel:

```
f x = x*x +1                           1
fL x = (x *L x) +L (replicateP n 1)    2
  where                                3
    n = lenghtP x                      4
```

Verschachtelte Funktionen

Beispiel:

```
g :: [:Float:] -> [:Float:]                    1
g xs = mapP f xs                               2
```

Ersetzen von `mapP f` durch `fL`

```
g :: [:Float:] -> [:Float:]                    1
g xs = fL xs                                   2
```

Liften von `g` ergibt

```
gL :: [:[:Float:]:] -> [:[:Float:]:]           1
gL xs = fLL xs                                 2
```

Doppelt geliftete Version von `f`? n-fach geliftete Versionen?

Lösung

Definition von `fLL` durch `fL`.

```
fLL :: [:[:Float:]:] -> [:[:Float:]:]                    1
fLL xs = unconcatP xs (fL (concatP xs))                  2
```

- Konkatenieren aller Reihen von `xs` zu einem flachen Array

```
concatPA :: [: [:a:] :] -> [:a:]                         1
concatPA (PA segs xs) = xs                               2
```

- Ausführen von `fL`
- Zerschneiden von `xs` in ursprüngliche Form

```
unconcatP :: [:a:] -> [:a:] -> [: [:a:] :]               1
unconcatP (PA segs xs) ys = PA segs ys                   2
```

- `concatP` und `unconcatP` in konstanter Zeit

Performanz DPH