Funktionale Programmierung

Parallele Auswertung

Andreas Abel und Steffen Jost

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

21. Mai 2012

Grundlagen

Paralleles Rechnen ermöglicht schnellere Ausführung von Programmen durch gleichzeitige Verwendung mehrerer Prozessoren.

Multi-core Maschinen oder **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.

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 anstatt 1 Prozessor 16 verwendet werden, aber schlecht, wenn 128 verwendet werden (dürfen).

Koordinierung der parallelen Ausführung

Es müssen koordiniert werden:

Partitionierung: Aufspaltung des Programs in unabhängige Teile, **Threads**, welche parallel berechnet werden können

- Wieviele Threads?
- Wieviel macht ein einzelner Thread?

Synchronisation
Identifizierung von Abhängigkeiten zwischen den Threads

Kommunikation / Speicher Management
Austausch der Daten zwischen den Threads

Mapping Zuordnung der Threads zu Prozessoren

Scheduling Auswahl lauffähiger Threads auf einem Prozessor

Komplette explizite Spezifizierung durch den Programmierer sehr aufwändig!

Koordinierung der parallelen Ausführung

Existierende Sprachen unterscheiden sich stark im Grad der Kontrolle dieser Aspekte. Es gibt verschiedene Ansätze, um den Aufwand zur Koordinierung der parallelen Ausführung für den Programmier zu reduzieren z.B. Skeletons

Manche Sprachen verwenden eigene **Koordinierungssprachen** die diese Aspekte kontrollieren

Viele der verwendeten Sprachen bieten lediglich Bibliotheken o.ä. mit denen eine explizite Kontrolle möglich ist z.B. MPI für message passing, OpenMP für shared-memory Programmierung

Parallele Funktionale Sprachen

Rein funktionale Programmiersprachen haben keine Seiteneffekte und sind daher **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.
Ideal, um verschiedene Teile des Programms parallel zu berechnen!

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: GpH

Glasgow parallel Haskell (GpH) ist eine konservative Erweiterung von Haskell und definiert zwei neue Primitive: Control.Parallel

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

`x 'par' e` definiert die parallele Auswertung von `x` und von `e`.

- gibt Hinweis auf parallele Auswertung; nicht erzwungen
- erzeugt **spark** für `x`; wird eventuell ausgewertet, falls ein Prozessor frei ist

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

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

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

Beispiel: parfact

Parallele Faktorial Berechnung — klassisches **divide and conquer**:

```
parfact :: Integer -> Integer
parfact n = parfact' 1 n

parfact' :: Integer -> Integer -> Integer
parfact' m n
  | m == n     = m
  | otherwise = left `par` right `pseq` (left * right)
    where mid   = (m + n) `div` 2
          left  = parfact' m mid
          right = parfact' (mid+1) n
```

- Auswertereihenfolge muss explizit kontrolliert werden, sonst könnte Multiplikation vor `left` ausgewertet werden
- `pseq` erzwingt Auswertung von `left` und `right` vor Auswertung der Multiplikation

Beispiel: parfact

Verbesserung des Codes durch "thresholding":

```
parfact2 :: Integer -> Integer
parfact2 n = parfact2' 1 n 50

parfact2' :: Integer -> Integer -> Integer -> Integer
parfact2' m n t
  | (n - m) <= t = product [m..n]
  | otherwise = left `par` right `pseq` (left * right)
      where mid = (m + n) `div` 2
            left = parfact2' m mid t
            right = parfact2' (mid + 1) n t
```

- Overhead für Verwaltung der Sparks ist teuer, daher besser "kleine" Sparks vermeiden

Beispiel: Quicksort

Naive Version von Quicksort erzeugt nur geringe Parallelität:

```
qsort1 :: Ord a => [a] -> [a]
qsort1 []  = []
qsort1 [x] = [x]
qsort1 (x : xs) = qlo `par`
                  qhi `par`
                  (qlo ++ (x:qhi))
  where qlo = qsort1 [ y | y <- xs, y <  x ]
        qhi = qsort1 [ y | y <- xs, y >= x ]
```

- Kaum parallele Auswertung, da alle Threads schnell WHNF erreichen (erste Cons-Zelle jeder Subliste)

Beispiel: quick-sort

Auswertung der Teillisten muss forciert werden:

```
qsort2 :: Ord a => [a] -> [a]
qsort2 []       = []
qsort2 [x]      = [x]
qsort2 (x : xs) = forcelist qlo `par`
                  forcelist qhi `par`
                  (qlo ++ (x:qhi))
  where qlo = qsort2 [ y | y <- xs, y <  x ]
        qhi = qsort2 [ y | y <- xs, y >= x ]

forcelist :: [a] -> ()
forcelist []     = ()
forcelist (x:xs) = x `seq` forcelist xs
```

Problem: “forcing” auf vielen verschiedenen Datenstrukturen benötigt — und deren Kompositionen listen, listen von listen, ...

Auswertestrategien

Evaluation strategies bieten von Berechnung getrennte Abstraktion der parallelen Koordination Control.Parallel.Strategies

Definiert lediglich Koordination, also Abstraktion über `par` & `pseq`

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

Eine simple Auswertungsfunktion

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

und `return` Operator zum Einführen in die `Eval` Monade:

```
return :: a -> Eval a
return x = Done x
```

Auswertestrategien

Anwendung einer Auswertestrategie erfolgt mit `using`:

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

GpH Beispiel:

```
somefun x y = someexpr `using` somestrat
```

Auswertestrategien mit Auswertungsgrad

Einfache Strategien zur Kontrolle von Auswertegrad, Auswertereihenfolge und Parallelität:

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

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

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

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

Beispiel: parfact mit Strategie

```
parfact3 :: Integer -> Integer
parfact3 n = parfact3' 1 n

parfact3' :: Integer -> Integer -> Integer
parfact3' m n
  | m == n     = m
  | otherwise = (left * right) `using` strategy
    where mid   = (m + n) `div` 2
          left  = parfact3' m mid
          right = parfact3' (mid+1) n
          -- strategy = r0
          strategy result = do
            rpar left
            rpar right
            return result
```

Kontrolle des Auswertegrad

Mit `r0`, `rseq` und `rdeepseq` wird der Auswertegrad eines Ausdruck reguliert.

`rdeepseq` wertet vollständig aus; dies wird über die `rnf` Strategie definiert, welche von der Klasse `NFData` bereitgestellt wird.

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

Für viele Basistypen entspricht `rnf` genau `rwhnf`;
viele Instanzen sind vordefiniert. Control.Parallel.Strategies

Kontrolle des Auswertegrad: rdeepseq

Instanzen für NFData werden beispielsweise so definiert:

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

Es gibt auch einen deepseq Operator, welcher sein erstes Argument vollständig auswertet:

```
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
```

Auswertestrategien kombinieren

Auswertestrategien können auch kombiniert werden:

Komposition

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

Sequentielle Anwendung auf Listen

```
evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x
                       xs' <- evalList s xs
                       return (x':xs')
```

Auswertestrategien kombinieren

Parallele Anwendung auf Listen

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

Parallele Anwendung auf Tupel

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

Zusammenfassung

Kontrolle von Auswerte-reihenfolge und -tiefe notwendig; erfolgt mittels Auswertestrategien
Strategien trennen Berechnung von der Koordination und spezifizieren:

- Auswertungsreihenfolge (mittels `pseq`)
- Auswertungsgrad (mittels `r0`, `rseq`, `rdeepseq`)
- Parallelität (mittels `par`)

GpH bietet ein Modell **semi-expliziter** Parallelität:

- Lediglich Programmteile zur parallelen Auswertung annotieren
- Verwaltung der Parallelität erfolgt im Laufzeitsystem
- Rein funktionaler Code mit nur wenigen Änderungen parallelisierbar
- Autoren: P.W. Trinder, K. Hammond, H-W. Loidl, Simon L. Peyton Jones.