Sprachkonzepte der parallelen Programmierung

Johannes Waldmann

HTWK-FIMN

2. Mai. 2012

Die Zukunft ist parallel. . .

- Prozessoren werden nicht (wesentlich) schneller,
- sondern kleiner und sparsamer:
- d. h. man kann (auf einem Chip, in einem Gehäuse) viele parallel betreiben und das auch bezahlen . . .
- aber wie programmiert man für solche Hardware?
- und wie lehren wir das?

Die Zukunft ist parallel. . .

- Prozessoren werden nicht (wesentlich) schneller,
- sondern kleiner und sparsamer:
- d. h. man kann (auf einem Chip, in einem Gehäuse) viele parallel betreiben und das auch bezahlen . . .
- aber wie programmiert man für solche Hardware?
- und wie lehren wir das?

. . . die Zukunft von *parallel* ist *deklarativ*!

Nebenläufig oder parallel?

Nebenläufigkeit (concurrency)

- sichtbar in der Aufgabenstellung (z. B. Simulation)
- und in der Lösung (z. B. Threads)
- nicht notwending in der Hardware (z. B. time sharing)

Nebenläufig oder parallel?

Nebenläufigkeit (concurrency)

- sichtbar in der Aufgabenstellung (z. B. Simulation)
- und in der Lösung (z. B. Threads)
- nicht notwending in der Hardware (z. B. time sharing)

Parallelität

- vorhanden in der Hardware (z. B. mehrere Kerne)
- nicht sichtbar in der Aufgabenstellung (z. B. Sortieren)
- im Idealfall unsichtbar in der Lösung
- . . . oder *orthogonal* dazu

Quellen

- Maurice Herlihy, Nir Shavit: *The Art of Multiprocessor Programming*, Elsevier 2008.
- Simon Peyton Jones: *Beautiful Concurrency*, in: Oram, Wilson: Beautiful Code, O'Reilly 2007
- Brian Goetz: *Java Concurrency in Practice*, Addison-Wesley 2006.
- Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, Masato Takeichi: *Automatic Inversion Generates Divide-and-Conquer Parallel Programs*, PLDI 2007.
- J. Waldmann: Skript Sprachkonzepte der par. Prog. `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss11/skpp/folien/main/`

Modelle der nebenl. und par. Prog.

gemeinsamer Speicher

- Modelle: CREW-PRAM, Kompl.-Klasse NC, Petrinetz ($=$ Automat mit verteiltem Zustand)
- Anwendung: Threads, Transaktionen

Modelle der nebenl. und par. Prog.

gemeinsamer Speicher

- Modelle: CREW-PRAM, Kompl.-Klasse NC, Petrinetz (= Automat mit verteiltem Zustand)
- Anwendung: Threads, Transaktionen

privater Speicher

- Modell: CSP (communicating sequential processes)
- Anwendung: Kanäle, Nachrichten, Aktoren

Modelle der nebenl. und par. Prog.

gemeinsamer Speicher

- Modelle: CREW-PRAM, Kompl.-Klasse NC, Petrinetz (= Automat mit verteiltem Zustand)
- Anwendung: Threads, Transaktionen

privater Speicher

- Modell: CSP (communicating sequential processes)
- Anwendung: Kanäle, Nachrichten, Aktoren

gar kein (veränderlicher) Speicher

- Modell: Funktion
- Anw.: Strategie-Annotationen, paralleles *fold* über assoziative Funktion (Bsp. map/reduce)

Shared State

- mit Sperren (locks)
  klassisches Beispiel: 5 Philosophen
- sperrfrei (Compare-And-Set, . . .)
  Bsp.: Elimination Backoff Stack
- optimistisch (STM: Software Transactional Memory)

Modularität nur durch STM! — Herlihy/Shavit, Kap. 18:

- locks are hard to manage effectively,
- atomic primitives (CAS) result in complex algorithms,
- it is difficult to compose multiple calls to multiple objects into atomic units.

Software Transactional Memory (Bsp)

```
transfer :: Account -> Account -> Amount
transfer from to amount =
  atomically $ do  deposit to amount
                   withdraw from amount
type Account = TVar Amount
withdraw :: Account -> Amount -> STM ()
withdraw account amount = do
  balance <- readTVar account
  if amount > balance then retry
  else writeTVar account (balance - amour
atomically :: STM a -> IO a
```

aus: Simon Peyton Jones: *Beatiful Concurrency*, 2007.

Software Transactional Memory

Transaktionen sind *atomar* und *isoliert*.

mögliche Implementierung mit Logs (vgl. Cache)

- Log enthält Lese- und Schreibzugriffe
- Commit nur, wenn Log konsistent ist
- Retry nur nach Änderung

Software Transactional Memory

Transaktionen sind *atomar* und *isoliert*.

mögliche Implementierung mit Logs (vgl. Cache)

- Log enthält Lese- und Schreibzugriffe
- Commit nur, wenn Log konsistent ist
- Retry nur nach Änderung

Realisierung von STM z. B.

- als Bibliothek in Haskell
- als Sprachkonzept in Clojure

Typsystem beschränkt Aktionen in Transaktionen:

```
atomically :: STM a -> IO a
readTVar account :: STM Amount
putStrLn "foobar" :: IO ()
```

Communicating Sequential Processes

(Tony Hoare, 1985, `http://www.usingcsp.com/`)

- Syntax: Prozeß $=$ Term (Prozeß-Algebra)
- Semantik:
  - Spursprache
  - Ablehnungs-Semantik

Anwendungen:

- Rendezvous zwischen Tasks in Ada
- Nachrichten für Aktoren in Scala
- Kanäle in Go

Funktionales Paralleles Programmieren

Ausdrücke haben *Wert*, aber keine *Wirkung*.
Ermöglicht sehr einfach Parallelität auf sehr hohem (d. h. dem richtigen) Abstraktionsniveau:

- Argumente jedes Funktionsaufrufes können parallel ausgewertet werden;
  ergibt massiven Parallelismus,
  ggf. Steuerung durch Annotationen sinnvoll
- geschachtelte Aufrufe von *assoziativen Operatoren* können umgruppiert werden, so daß balancierter Aufrufbaum entsteht

Parallelität durch Annotation

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys ; merge xs [] = xs
merge (x:xs) (y:ys) =
    if x < y then x : merge xs (y:ys)
             else y : merge (x:xs) ys
msort :: Ord a => [a] -> [a]
msort [] = [] ; msort [x] = [x]
msort xs =
 let (l, r) = splitAt (div (length xs) 2)
 in merge (msort l `using` rpar `dot` eva
          (msort r `using` evalList r0)
```

Iterierte assoziative Operationen

Prinzip: wenn $\oplus$ assoziativ, dann sollte man

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_7 \oplus$$

so auswerten:

$$((x_1 \oplus x_2) \oplus (x_3 \oplus x_4)) \oplus ((x_5 \oplus x_6) \oplus (x_7 \oplus x_8))$$

Beispiel: carry-lookahead-Addierer
(die assoziative Operation ist die Verkettung der Weitergabefunktionen des Carry)

Konstruktion assoz. Operatoren

. . . durch 3. Homomorphiesatz für Listen:
Wenn eine Funktion f sowohl als fold-left also auch als fold-right darstellbar ist,

$$\begin{aligned} f[x_1, \ldots, x_n] &= ((s \oplus_l x_1) \oplus_l x_2) \ldots \oplus_l x_n \\ f[x_1, \ldots, x_n] &= x_1 \oplus_r \ldots (x_{n-1} \oplus_r (x_n \oplus_r t)) \end{aligned}$$

dann auch durch ein balanciertes fold über eine assoziative Operation.
Beispiel: maximale Präfixsumme.
Beachte: aus Existenz zweier sequentieller Algorithmen folgt hier die Existenz eines effizienten parallelen Algorithmus!

Map/Reduce-Algorithmen

```
map_reduce
   :: ( (ki, vi) -> [(ko,vm)] )   -- ^ map
   -> ( (ko, [vm]) -> [vo] ) -- ^ reduce
   -> [(ki,vi)] -- ^ eingabe
   -> [(ko,vo)] -- ^ ausgabe
```

Beispiel (word count)

```
ki = Dateiname, vi = Dateiinhalt
ko = Wort     , vm = vo = Anzahl
```

- parallele Berechnung von `map`
- parallele Berechnung von `reduce`
- verteiltes Dateisystem für Ein- und Ausgabe

Map-Reduce: Literatur

- Jeffrey Dean and Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004. `http://labs.google.com/papers/mapreduce.html`
- Ralf Lämmel: *Google's MapReduce programming model - Revisited*, Science of Computer Programming - SCP , vol. 70, no. 1, pp. 1-30, 2008 `http://www.systems.ethz.ch/education/past-courses/hs08/map-reduce/reading/mapreduce-progmodel-scp08.pdf`

Folgerungen für die Lehre

Aspekte der Parallelen Programmierung

- Modelle, Kalküle
- Komplexität (L-Reduktion, P-Vollständigkeit)
- Algorithmen
- Sprachkonzepte
- Hardwarekonzepte (Architektur, Prozessoren)

Grundlagen für Programmierung:

- deklarativ (funktional).

Sequentielle imperative Programme lassen sich nur mit größter Mühe parallelisieren—das ist unökonomisch (auch in der Lehre)

Lehre (konkret)

(= meine Stunden lt. Modulplänen)

Bachelor

- Pflicht (4. Sem): Deklarative (fortgeschrittene) Programmierung

Master

- Pflicht (1. Sem): Prinzipien v. Progr.-Spr.
- Wahl (Kern): Compilerbau
- Wahl: Sprachkonzepte der parallelen Prog.

Änderungen:

- (leicht?) SKPP „ordnungskonform“ hochstufen
- (schwer?) Deklarative Programmierung *früher*