Sprachkonzepte der Parallelen Programmierung Vorlesung SS 11, WS 12, SS 13, WS 15–17

Johannes Waldmann, HTWK Leipzig

22. Januar 2018

Introduction

Sprachkonzepte der parallelen Programmierung

- programming language concepts

  for concurrent, distributed, and parallel computing
- why? 1. application requires it, 2. hardware allows it
- optional course for BSc. computer science students, in their 5th semester (of 6)
- each week (of 14): one lecture, one lab class (discussing homework, programming exercises)
- finally, written exams (closed book) 120 min

Concepts of Parallelism

- non-deterministic
  - concurrent (nebenläufig)
    interleaved execution of components
  - distributed (verteilt)
    as above, plus: explicit data transfer (messages)

  e.g., responsive multi-user system requires concurrency

  (can be simulated on sequential hardware: OS)
- deterministic

  for application probl. without concurrency requirement,

  use hardware parallelism to solve them faster

  e.g., matrix multiplication, text/image analysis

From Concept to Implementation

notice the gap:

- quite often, we want deterministic parallelism
- but hardware is concurrent (e.g., several cores)
  and distributed (e.g., per-core memory/cache)

who bridges the gap?

- WRONG: the *programmer* (application program handles sequencing, synchronisation and messaging)
- RIGHT: the *language* (with libraries, compiler, run-time system) (program expresses intent)

note the difference between: $\sum_{0\le i<n} x_i$ (intent)
and: `for(i=0;i<n;i++){s+=x[i];}` (sequencing)

Abstract! Abstract! Abstract!

main thesis

- *higher* abstraction level of the language
- $\Rightarrow$ *easier* for compiler and RTS to use hardware specifics (e.g., parallelism) for efficient execution

example (C#, mono) just one annotation expresses the intent of parallel execution:

```
Enumerable.Range(0,1<<25)
      .Select(bitcount).Sum()
Enumerable.Range(0,1<<25).AsParallel()
      .Select(bitcount).Sum()
```

this is why we focus on functional programming
(e.g., `Select` is a higher-order function)

Why did this work, exactly?

`Enumerable.Range(...).AsParallel().Sum()`

- technically, `AsParallel()`
  produces `ParallelQuery<T>` from `IEnumerable<T>`,
  and `Sum()` has a clever implementation for that type
- mathematically (“clever”), addition is *associative*,
  so we can group partial sums as needed
- if an operation is not associative?
  e.g., the final carry bit in addition of bitvectors

  then we should find a modification that is!

  because that allows for *straightforward*
  and *adaptable* (e.g., to number of cores) parallelism

Types for Pure Computations

measure run-times and explain (C# vs. Haskell)

```
Enumerable.Range(0,1<<25).Select(bitcount).Sum()
Enumerable.Range(0,1<<25).Select(bitcount).Count()
Enumerable.Range(0,1<<25)                 .Count()
length $ map bitcount $ take (2^25) [ 0 .. ]
length                $ take (2^25) [ 0 .. ]
```

- elements of list are not needed for counting
- computation of elements cannot be observed
  it has no side effects (Nebenwirkung)
  (this follows from `bitcount:: Int -> Int`)

  the Haskell RTS never calls `bitcount`,
- the C# type `int->int` includes side effects,
  so the RTS must call the function.

If we absolutely must program imperatively,

(imp. program execution = sequence of state changes)

- then we are on dangerous ground
  already for the sequential case
  proving an imperative program correct requires
  complicated machinery (Hoare calculus)
  hence it is often not done
  (for functional programs just use equational reasoning)
- we need even more caution (and discipline)
  for concurrent imperative programs
  need concurrency primitives
  - that have clear semantics
  - that solve typical problems
  - that are composable (to larger programs)

Typical Concurrency Problems

- mutual exclusion

  at most one process gets to access a shared resource (e.g., a shared memory location)

- producers and consumers, readers and writers
  - cannot consume item before it is produced,
  - cannot consume item twice

- concurrent mutable data structures
  - counters
  - collections (hash maps, ...)

Semantics for Concurrent Systems

...via mathematical models:

- Petri nets (Carl Adam Petri, 1926–2010)

  (automata with distributed state)

- process algebra (Bergstra and Klop 1982, Hoare 1985)

  (regular process expressions and rewrite rules)

  `http://theory.stanford.edu/~rvg/process.html`

- modal logic

  (statements about time-dependent properties)

  application: model checking,

  e.g., SPIN `http://spinroot.com/`

Concurrency Primitives

- locks (Semaphores, E.W. Dijkstra 1974) `http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF`
  of historical importance, but ... *locks are bad*
  (in particular, not composable)

- no locks
  atomic, non-blocking ("optimistic") execution of
  - elementary operations (compare-and-swap) realized in hardware `http://stackoverflow.com/questions/151783/`
  - transactions (STM, software transactional memory) `http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/` in Haskell, Clojure

Homework

1. which are associative? (give proof or counter-example)
   (a) on $\mathbb{Z}$: multiplication, subtraction, exponentiation
   (b) on $\mathbb{B}$ (booleans): equivalence, antivalence, implication
   (c) on $\mathbb{N}^2$: $(a,b) \circ (c,d) := (a \cdot c, a \cdot d + b)$

2. sum-of-bitcounts
   (a) re-do the C# bitcounting example in Java (hint: `java.util.stream`)
   (b) discuss efficient implementation of `int bitcount (int x);` (hint: time/space trade-off)
   (c) discuss efficient implementation of sum-of-bitcounts
      i. from 0 to $2^e - 1$
      ii. bonus: from 0 to $n - 1$ (arbitrary n)

   hint:
   how did little Carl Friedrich Gauß do the addition?
   morale:
   the computation in the example should never be done in real life, but it makes a perfect test-case since it keeps the CPU busy and we easily know the result.

3. sorting network exercise: `https://autotool.imn.htwk-leipzig.de/new/aufgabe/2453`

4. on your computer, install compilers/RTS for languages:
   Haskell (ghc), C# (mono), Java 8/9, Scala, Clojure, Go

   or make sure that you can ssh to Z423/Z430 computers

Petri-Netze

Einleitung

- Verhalten nebenläufiger Systeme *spezifizieren* und *modellieren*
- Spezifikation (Beispiel): Spursprache (Menge der möglichen Reihenfolgen von atomaren Aktionen)
- Modell (Beispiel): Petri-Netz (nebenläufiger Automat) eingeführt von Carl Adam Petri, 1962

Vergleiche: Beschreibung/Modellierung sequentieller Systeme durch reguläre Sprachen/endliche Automaten
Vorgehen hier: erst konkrete Modelle,
dann Spezifikationssprache (Logik).

Definition: Netz

Stellen/Transitions-Netz $N = (S, T, F)$

- S eine Menge von *Stellen*
- T eine Menge von *Transitionen*, $S \cap T = \emptyset$
- $F \subseteq (S \times T) \cup (T \times S)$ eine Menge von *Kanten*

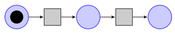
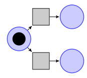
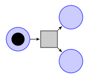
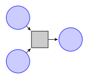
das ist ein gerichteter bipartiter Graph
Bezeichnungen:

- Vorbereich (Eingänge) einer Transition: $\text{Vor}_N(t) = \{s \mid (s,t) \in F\}$
- Nachbereich (Ausgänge) einer Transition: $\text{Nach}_N(t) = \{s \mid (t,s) \in F\}$.

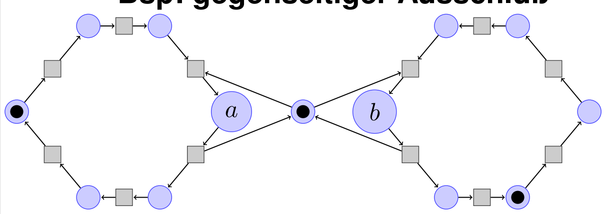
Zustände, Übergänge

- *Zustand* eines Netzes N ist Abbildung $z : S \to \mathbb{N}$ (für jede Stelle eine Anzahl von Marken)
- in Zustand z ist eine Transition t *aktiviert*, wenn jede Stelle ihres Vorbereiches wenigstens eine Marke enthält: $\forall s \in \text{Vor}(t) : z(s) \geq 1$
- eine aktivierte Transition *schaltet*: verbraucht Marken im Vorbereich, erzeugt Marken im Nachbereich.
- Bezeichnung $z_1 \xrightarrow{t} z_2$: aus Zustand z_1 entsteht durch Schalten von t der Zust. z_2.
- Def. $z_1 \xrightarrow{t} z_2$: erfordert 4 Fälle: alle Kombinationen von: $s \in \text{Vor}(t), s \notin \text{Vor}(t)$ mit $s \in \text{Nach}(t), s \notin \text{Nach}(t)$ effizientere Notation in Modell mit Kantengewichten.

Petri-Netze modellieren...

- sequentielle Ausführung 
- Auswahl (ein Zweig von mehreren) 
- nebenläufige Verzweigung (mehrere Zweige) 
- Synchronisation 
- Konflikte (gegenseitiger Ausschluß)

Bsp: gegenseitiger Ausschluß



Für jeden erreichbaren Zust. z gilt: $z(a) = 0 \vee z(b) = 0$.
Beispiel aus: Kastens und Kleine Büning: *Modellierung*, Hanser, 2008.
`http://www.hanser-elibrary.com/isbn/9783446415379`
Zeichnung mit TIKZ, vgl.
`http://www.texample.net/tikz/examples/nodetutorial/`

Petri-Netze und UML

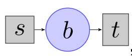
UML-2.5, Sect. 13.2.1
A variety of behavioral specification mechanisms are supported by UML, including:

- StateMachines that model finite automata (see C. 14)
- Activities defined using Petri-net-like graphs (see C. 15)
- Interactions that model partially-ordered sequences of event occurrences (see C. 17).

Sprache eines Netzes (I)

- Folge von Zuständen $z_0 \to_{t_1} z_1 \to_{t_2} z_2 \dots \to_{t_k} z_k$
  Spur $w = t_1 t_2 \dots t_k \in T^*$, Notation $z_0 \to_w z_k$
- für gegebenes Netz N und Startzustand z_0:
  Menge aller möglichen Spuren (Spursprache)
  $\text{spur}(N, z_0) = \{w \mid w \in T^* \wedge \exists z_k : z_0 \to_w z_k\}$
  vergleiche: Sprache eines endlichen Automaten
- Def: L ist präfix-abgeschlossen : $\iff$
  $\forall w \in L : \forall u \sqsubseteq_{\text{prefix}} w : u \in L$
  Satz: jede Spursprache ist präfix-abgeschlossen
  Bsp: $(ab)^*$ ist keine Petrinetz-Spursprache,
  aber $(ab)^*(a + \epsilon)$ ist eine PN-Spursprache.

Sprache eines Netzes (II)

- es gibt Petri-Netze mit komplizierten (= nicht regulären) Spursprachen
- Bsp: $N =$ , $z_0 = \{(b, 0)\}$.
  Beispiele: $\epsilon, s, ss, st, sss, sst, sts, \dots \in \text{spur}(F, z_0)$,
  $t, ts, stt \notin \text{spur}(F, z_0)$.
  allgemein: $\text{spur}(F, z_0) = \{w \mid \forall u \sqsubseteq w : |w|_s \geq |w|_t\}$
  Satz: $\text{spur}(F, z_0) \cap s^* t^* = \{s^x t^y \mid x \geq y\} \notin \text{REG}$,
  Beweis mit Schleifensatz (pumping lemma).
  Diese Spursprache ist kontextfrei.
  (Nicht jede PN-Spursprache ist CF.)

Kapazitäten und -Schranken

Erweiterung:

- jede Kante bekommt eine *Gewicht* (eine positive Zahl),

  beschreibt die Anzahl der Marken, die bei jedem Schalten durch die Kante fließen sollen.

Einschränkung:

- Stellen können einer *Kapazität* bekommen (eine positive Zahl), beschreibt die maximal erlaubte Anzahl von Marken in dieser Stelle

falls alle Kapazitäten beschränkt ⇒ Zustandsmenge endlich (aber mglw. groß) ⇒ vollständige Analyse des Zustandsübergangsgraphen (prinzipiell) möglich

Formale Definition der Ü.-Relation

- Netz mit Kantengewichten: $F : (S \times T) \cup (T \times S) \to \mathbb{N}$
- Beziehung zu einfachem Modell:

  keine Kante: Gewicht 0, einfache Kante: Gewicht 1
- Transition t ist in Zustand z aktiviert: $\forall s : z(s) \geq F(s,t)$.
- Zustandsübergang: $z_1 \xrightarrow{t} z_2$:

  $\forall s : z_1(s) - F(s,t) + F(t,s) = z_2(s)$
- beachte: durch *Verallgemeinerung* des Modells wird Notation hier *einfacher* ...
  und damit auch Beweise, die Notation benutzen.

Bedingung/Ereignis-Netze

...erhält man aus allgemeinem Modell durch:

- jede Kante hat Gewicht 1
- jede Kapazität ist 1

  (dadurch wird der Zustandsraum endlich!)

Beispiele:

- Ampelkreuzung

  (zwei Ampeln grün/gelb/rot, nicht gleichzeitig grün)
- speisende Philosophen
- Definition und Analyse von Lebendigkeit, Fairness

Eigenschaften von Petri-Netzen

Definitionen (für Netz N mit d Stellen, Zustand $m \in \mathbb{N}^d$)

- $M \subseteq \mathbb{N}^d$: Nachfolger $\text{Post}_N(M) = \{y \mid m \in M, m \to_N y\}$
- Mehr-Schritt-Nachfolger: $\text{Post}^*_N(M)$

Eigenschaften (Beispiele):

- Erreichbarkeit: gilt $m_0 \to^*_N m_1$?
- Beschränktheit: ist $\text{Post}^*_N(m_0)$ endlich?
- Platz-Beschränktheit: $\{m(p) \mid m \in \text{Post}^*_N(m_0)\}$ endlich?

Alain Finkel und Jerome Leroux: *Neue, einfache Algorithmen für Petri-Netze*, Informatik-Spektrum 3/2014, S. 229–236

Beschränktheit ist entscheidbar

$\text{Post}^*_N(m_0)$ endlich?
Entscheidungsverfahren: wir zählen abwechselnd auf:

- A: Elemente von $\text{Post}^*(m_0)$ (z.B. Breitensuche)
- B: Kandidaten für Zeugen für Unbeschränktheit:
  Kandidat ist $(s,t) \in T^* \times T^+$,
  ist Zeuge, wenn $m_0 \to^s x \to^t y$ mit $x \leq y$ und $x \neq y$

zu zeigen ist: $\text{Post}^*_N(m_0)$ unendlich $\iff$ Zeuge existiert
„⇐“: ist klar. Für „⇒“:

- $\to$ auf $\text{Post}^*(m_0)$ ist unendlichen Baum endlichen Grades
- enthält unendlichen Pfad (Lemma von König)
- dieser Pfad enthält passendes (x,y) (Lemma v. Higman)

Lemma von Higman, WQO

- Def. eine Relation $\leq$ auf M
  heißt *WQO (wohl-quasi-Ordnung)*, falls gilt:
  es gibt keine unendliche $\leq$-Antikette.
- Def. eine Menge $A \subseteq M$ heißt *Antikette*,
  falls $\forall x,y \in A : x \neq y \Rightarrow x \not\leq y$.
  (Def. ... *Kette*, falls ... $\Rightarrow x \leq y \vee y \leq x$.)
- Bsp: $(\mathbb{N}, \leq)$ ist WQO. $(\mathbb{N}, |)$ (Teilbarkeit) ist ?
- Versionen von Higmans Lemma:
  Satz: $(\mathbb{N}^d, \leq)$ (komponentenweise $\leq$) ist WQO.
  Satz: $(\Sigma^*, \sqsubseteq)$ ist WQO, wobei $(u \sqsubseteq v) :\iff$
  u entsteht aus v durch Löschen von Buchstaben

Aufgaben

- Vergleich Wechselschalter (Kastens/Kleine Büning Abb. 7.23) mit Netz aus Vorlesung
- Zustandsgraph (K/KB Aufg. 7.16)
- autotool-Aufgaben (reachability, deadlock)
- zu Petri-Netz für gegens. Ausschluß (Ampelsteuerung):

  formuliere eine Verschärfung der angegebenen Invariante, die man durch Induktion über Länge der Schaltfolge beweisen kann.
- Beispiele Petri-Netze zur Modellierung (Modell)eisenbahn:
  - Zug darf Streckenabschnitt nur befahren, wenn Signal

grün zeigt
  - Zug darf Weiche nicht aus dem falschen Zweig befahren
- XOR-Verzweigung (mit späterer Zusammenführung) durch Petri-Netz ist einfach. Wie geht das für OR? (ein Zweig *oder beide* Zweige werden ausgeführt)
- Diskutiere, verallgemeinere, formalisiere diese Simulation einer Kapazitätsbeschränkung: `http://www.texample.net/tikz/examples/nodetutorial/`
- unendliche Antiketten für die Relationen
  - Teilbarkeit auf $\mathbb{N}$
  - Teilwort-Relation $\leq$ auf $\{a,b\}^*$, mit
    $u \leq v \iff \exists p,q : puq = v$

- möglichst große Antikette für $(\mathbb{N}^2, \leq)$, die $(5, 3)$ enthält.
- …für $(\mathbb{N}^3, \leq)$ an, die $(5, 3, 1)$ enthält.

Spezifikation und Verifikation nebenläufiger Prozesse

Einleitung

wie überall,

- Trennung von Spezifikation und Implementierung
- jeweils ein mathematisches Modell
- Sätze über Eigenschaften, Beziehungen dieser Modelle
- Algorithmen zur Beantwortung der Frage: erfüllt die Implementierung die Spezifikation?

so auch hier:

- Spezifikation: PLTL (propositional linear time logic)
- Implementierung: Omega-Wörter, -Sprachen, -Automaten

Literatur

- Mordechai Ben-Ari: *Principles of Concurrent and Distributed Programming*, Prentice Hall 1990
- Beatrice Berard et al.: *Systems and Software Verification*, Springer 2001

erfordert eigentlich eine eigene Vorlesung, vergleiche

- Bertrand Meyer: *Concepts of Concurrent Computation*, `http://se.inf.ethz.ch/courses/2012a_spring/ccc/`
- Sibylle Schwarz: Verifikations- und Spezifikationsmethoden (Abschnitt 3: Model Checking) `http://whz-cms-10.zw.fh-zwickau.de/sibsc/lehre/ws11/veri/`

Kripke-Strukturen, Omega-Wörter

allgemein: Kripke-Struktur zu Variablenmenge V ist

- Graph (S, T) mit $S =$ Menge der Systemzustände, $T \subseteq S \times S$ Menge der Zustandsübergänge
- Knotenbeschriftung $b : S \to (V \to \mathbb{B})$ d.h., $b(s)$ ist eine Belegung der Variablen V

hier speziell:

- $S = \mathbb{N}$ (Zeitpunkte $0, 1, \ldots$)
- $T = \{(s, s + 1) \mid s \in \mathbb{N}\}$ (linear time)

Beispiel:

- $V = \{p, q\}$,
- $b(s) = \{(p, (s \geq 3)), (q, (2 \mid s))\}$

Omega-Wörter und -Sprachen

- jede lineare Kripke-Struktur über V

  entspricht einem unendlichen Wort über $\Sigma = 2^V$

  Bsp: $(0, 1)(0, 0)(0, 1)(1, 0)(1, 1)(1, 0)(1, 1)\ldots$
- ein unendliches Wort (Omega-Wort) über Σ

  ist eine Abbildung $\mathbb{N} \to \Sigma$
- Σ^ω bezeichnet die Menge aller Omega-Wörter über Σ
- Schreibweise für Omega-Wörter mit schließlich periodischer Struktur:

  $(0, 1)(0, 0)(0, 1)\ ((1, 0)(1, 1))^\omega$

  vgl. unendl. Dezimalbrüche $3/22 = 0.1\overline{36}$

PLTL: propositional linear time logic

Syntax:

- Variablen $p, q, \ldots$, logische Operatoren $\neg, \vee, \wedge, \Rightarrow, \ldots$
- temporale Operatoren: immer $\Box$, irgendwann $\Diamond$,…

Beispiele: $\Diamond(p \vee q)$, $\Box\Diamond p$, $\Diamond\Box p$

Semantik: Wert der Formel F in Struktur K zur Zeit s:

- für $v \in V$: $\mathrm{wert}(v, K, s) = b_K(s)(v)$
- $\mathrm{wert}(F_1 \wedge F_2, K, s) = \min\{\mathrm{wert}(F_1, K, s), \mathrm{wert}(F_2, K, s)\}$
- $\mathrm{wert}(\Box F_1, K, s) = \min\{\mathrm{wert}(F_1, K, s') \mid s' \in \mathbb{N}, s' \geq s\}$
- $\mathrm{wert}(\Diamond F_1, K, s) = \max\{\mathrm{wert}(F_1, K, s') \mid s' \in \mathbb{N}, s' \geq s\}$

Übung: $\Diamond\Box\phi \Rightarrow \Box\Diamond\phi$ ist allgemeingülitg (gilt in jeder Struktur), …aber die Umkehrung nicht

PLTL-Spezifikationen von Systemeigenschaften

- gegenseitiger Ausschluß (*mutual exclusion*):

  Variablen: $p_i :=$ Prozeß i besitzt eine Ressource
  - Spezifikation (2 Prozesse): $\Box\neg(p_1 \wedge p_2)$
  - Übung: für 3 Prozesse lautet die Formel nicht $\Box\neg(p_1 \wedge p_2 \wedge p_3)$. Warum nicht? Wie dann?
- Fairness (kein Verhungern, *no starvation*)

  Variablen: $A_i :=$ Prozeß i beantragt Ressource; P_i

  Spezifikation: $\Box(A_1 \Rightarrow \Diamond P_1) \wedge \ldots \wedge \Box(A_n \Rightarrow \Diamond P_n)$

PLTL: Algorithmen

Satz: die folgenden Fragen sind entscheidbar:

- Modell-Problem:
  - Eingaben: eine PLTL-Formel F über V, ein schließlich periodisches Wort $w \in \Sigma^\omega$ mit $\Sigma = \mathbb{B}^V$
  - Frage: gilt $1 = \mathrm{wert}(F, w, 0)$
- Erfüllbarkeits-Problem:
  - Eingabe: eine PLTL-Formel F
  - Frage: gibt es $w \in \Sigma^\omega$ mit $1 = \mathrm{wert}(F, w, 0)$

Beweis-Idee: die Mengen $\{w \in \Sigma^\omega \mid 1 = \mathrm{wert}(F, w, 0)\}$ sind ω-regulär (Def. auf nächster Folie) und lassen sich durch endliche Automaten beschreiben. (J. R. Büchi 1962, A. Pnueli 1977)

ω-(reguläre) Sprachen

- Alphabet Σ,
- ω-Wort $w \in \Sigma^\omega$: Abbildung $\mathbb{N} \to \Sigma$
- ω-Sprache $L \subseteq \Sigma^\omega$: Menge von ω-Wörtern
- ω-reguläres Wort: hat die Form $u \cdot v^\omega$ mit $v \neq \epsilon$.

  Achtung: es gibt kein Produkt von ω-Wörtern, also auch keine geschachtelten Omegas.
- ω-reguläre Sprache: beschrieben durch ω-regulären Ausdruck: $P_1 \cdot K_1^\omega \cup \ldots \cup P_n \cdot K_n^\omega$ mit P_i, K_i regulär und $\neq \emptyset$, $\epsilon \notin K_i$

  Achtung: eine ω-reguläre Sprache (Bsp. $(a+b)^\omega$) kann auch nicht-reguläre ω-Wörter enthalten.

Übung PLTL

- Ist die Struktur $0(001)^\omega$ ein Modell der Formel $\Diamond\Box p$?
- Gibt es ein Modell für $\Box(p \iff \Diamond\neg p)$?
- Formalisieren Sie (mit den Variablen p_i für „Prozeß i besitzt Ressource“)
  - $F =$ Prozeß 1 besitzt die Ressource unendlich oft,
  - $G =$ Prozesse 1 und 2 besitzen die Ressource nie gleichzeitig,
  - $H =$ immer, wenn Prozeß 1 die Ressource besitzt, dann besitzt Prozeß 2 diese nicht, wird sie aber später erhalten.
- Für alle 8 Konjunktionen von $\{F, G, H, \neg F, \neg G, \neg H\}$:

geben Sie jeweils ein Modell als ω-reguläres Wort an (falls möglich).

- durch die Operatoren
  - $F \,\mathsf{U}\, G$: Es gibt einen Zeitpunkt, zu dem G gilt. Bis dahin gilt F.
  - $\mathsf{X}\, F$: im nächsten Zeitpunkt gilt F.

  wird PLTL erweitert.
  - Gegeben Sie die formale Semantik von U und X an.
  - Wie kann man $\Diamond$ durch Until realisieren?
  - Realisieren Sie Until durch Next (mit einer Hilfsvariablen p, die weder in F noch in G vorkommt) $p \leftrightarrow (F \,\mathsf{U}\, G)$ gdw. $G \vee \ldots$

Konkrete Syntax der PLTL-Operatoren

deutsch	Symbol	englisch	autotool NuSMV	Spin
irgendwann	$\Diamond$	finally (eventually)	F	<>
immer	$\Box$	globally generally	G	[ ]
bis		until	U	U
nächster		next	X	X

Vergleich der PLTL-Operatoren

Ausdrucksstärke von PLTL(M) für $M \subseteq \{\mathsf{F}, \mathsf{G}, \mathsf{U}, \mathsf{X}\}$:

- PLTL(F, G) kann $(abc)^\omega$ nicht von $(acb)^\omega$ unterscheiden

  dabei ist a Abkürzung für $\{a = 1, b = 0, c = 0\}$, usw.
- ... aber PLTL(U), durch $(a \,\mathsf{U}\, b) \,\mathsf{U}\, c$
- PLTL$(\mathsf{F}, \mathsf{G}, \mathsf{U})$ kann $(ab)^\omega$ nicht von $(abb)^\omega$ unterscheiden,
- ... aber PLTL(X), durch ... ?

Model-Checking mit NuSMV

Überblick

- `http://nusmv.fbk.eu/`
- Eingabe:
  - Zustands-Übergangs-System
    Bsp: `init(p) := 0; next(p):= !p;`
  - Spezifikation (LTL-Formel, Bsp: $\Box\Diamond p$, notiert als `G F p`)
- NuSMV kann
  - System simulieren (zufällig, benutzergesteuert)
  - Erfüllung der Spezifikation beweisen
  - für Nicht-Erfüllung einen Beweis (Spur) berechnen

NuSMV: Programm und Simulation

- Programm (beschreibt Zustands-Übergangs-Relation)

```
MODULE main
VAR p : boolean; q: boolean;
ASSIGN init (p) := FALSE;
    next(p) := !p; next(q) := {q,FALSE};
```

- interaktive Simulation (`nusmv -int ex1.smv`, dann)

```
NuSMV > go
NuSMV > pick_state -i
NuSMV > simulate -i
```


NuSMV: Spezifikation und Verifikation

- Programm mit Spezifikation

```
MODULE main
VAR p : boolean; q: boolean;
ASSIGN init (p) := FALSE;
    next(p) := !p; next(q) := {q,FALSE};
LTLSPEC  F G p
```

- Verifikation: `nusmv  ex1.smv`

```
-- specification  F ( G p)  is false
-- as demonstrated by the following executi
Trace Description: LTL Counterexample
...
```


NuSMV: Modellierungssprache

- Zweck: Beschreibung der Zustands-Übergangs-Relation
- Datentypen: `boolean`,`int`, Aufzählungen,. . .
- Ausdrücke: ähnlich C, aber *nebenwirkungsfrei*
- Zustandsvariablen:
  - Deklaration: `VAR p : {red, green};`
  - Initialisierung: `ASSIGN init(p) := green;`
- Zustandsübergänge
  - deterministisch `next(p) := red;`
  - nichtdeterministisch `next(p) := {red,green};`
  - bedingt:
    `next(q) := case  p = red : TRUE  esac;`

NuSMV: Simulation Petrinetz

```
MODULE transition(vor,nach)
ASSIGN
  next (vor)  :=
    case vor > 0 :  vor - 1; TRUE : vor ; esac;
  next (nach) :=
    case vor > 0 & nach < 7 : nach + 1; TRUE : nach ;
MODULE main
VAR s1 : 0 .. 7; s2 : 0 .. 7; s3 : 0 .. 7;
    t1 : process transition(s1, s2);
    t2 : process transition(s2, s1);
    t3 : process transition(s1, s3);
ASSIGN init (s1):=1; init(s2):=2; init (s3):=0;
LTLSPEC G s1 > 0;
```

Ü: überprüfe mit dieser Methode Eigenschaften von Petrinetzen aus Vorlesung (z.B. Ampelschaltung)

Entscheiden der PLTL-Erfüllbarkeit

- Beispiel: „gibt es ein Modell für $\mathsf{G}(p \iff \mathsf{F}\,\neg p)$“?
- `MODULE main VAR p : boolean;`
  `ASSIGN next(p) := {FALSE,TRUE};`
  `LTLSPEC ! G (p <-> F ! p)`

  Verifikation mit NuSMV liefert: *spec. is true*
- Antwort: Formel ist nicht erfüllbar. — Begründung:
  - Spursprache ist $\{0,1\}^\omega$ (d.h. *alle* Wörter), wegen nichtdeterministischer Auswahl in jedem Schritt
  - LTLSPEC ist die Negation der Formel ϕ aus der Aufgabenstellung. NuSMV beweist, daß diese in jeder Spur wahr ist, also ist ϕ für jede Spur falsch, also besitzt ϕ kein Modell.

NuSMV – build from source

falls Fehler

```
NuSMV-2.6.0/cudd-2.4.1.1/util/pipefork.c:46:
     union wait status;
```

dann in ändere in dieser Datei Zeile 43 zu

```
#if (defined __linux__) || (defined __hpux)
```

Nebenläufige Java-Programme

Threads erzeugen und starten

Thread-Objekt implementiert `run()`,
diese Methode wird aufgerufen durch `start()`,
das aber sofort zurückkehrt (mglw. bevor `run()` endet).

```
for (int t=0; t<8; t++) { new Thread() {
        public void run() {
            System.out.println (t);
        }
    }.start();
}
```

alternative Notation (Java $\geq$ 8)

```
new Thread( () -> System.out.println(t) );
```


Auf das Ende von Threads warten

`t.join()` blockiert aufrufenden Thread
und kehrt erst zurück, wenn `t` beendet ist:

```
t1.start(); t2.start();
...
t1.join() ; t2.join();
```

- das ist die einfachste Möglichkeit der Synchronisation, benutzt nur Threads selbst
- es gibt viele weitere Möglichkeiten,

  diese benutzen zusätzliche Objekte (Sperren)

Gemeinsamer Speicher

(Vorsicht, Code ist absichtlich falsch)

```
int s = 0; // gemeinsamer Speicher
// Threads erzeugen:
for (int t=0; t<threads; t++) {
 new Thread ( () ->
  { for (int i = 0; i<steps; i++)  s++; });
// Threads starten: ...
// auf Threads warten: ...
System.out.println (s);
```

- Ausgabe ist i.A. deutlich kleiner als `threads*steps`.
- `s++` ist nicht atomar
- tatsächlich wird noch viel weniger garantiert

Das Java-Speichermodell

- beschreibt garantiertes Verhalten beim Zugriff nebenläufiger Programme auf gemeinsamen Speicher

  (Objekt-Attribute, Klassen-Attribute, Array-Inhalte)
- Definition: JLS Kap. 17.4 `http://docs.oracle.com/javase/specs/jls/se9/html/jls-17.html#jls-17.4`
- Erläuterungen: William Pugh: `http://www.cs.umd.edu/~pugh/java/memoryModel/`

Sequentielle Konsistenz (Plan)

- moderne Compiler und Hardware (Prozessoren/Caches) können elementare Anweisungen umordnen und verändern,
- wenn trotzdem das Ergebnis berechnet wird, das der Programmierer erwartet, nämlich das *ohne Umordnung*
- falls das so ist, heißt der Compiler *sequentiell konsistent*
- sequentielle Konsistenz gilt in Java innnerhalb jedes Threads, aber *nicht* zwischen Threads
- das Erreichen der gewünschten Konsistenz bleibt dem Programmierer überlassen
  (zu erreichen durch Sprache und Bibliotheken)

Beispiel Umordnung

```
vorher  : A == 0; B == 0;
Thread 1: r2 = A ; B = 1;
Thread 2: r1 = B ; A = 2;
```

- Ist schließlich `r1 == 1; r2 == 2` möglich?
- in NuSMV nicht (Ü: ausprobieren)
- in Java doch, denn …

Beispiel Code-Änderung

```
vorher: p == q und p.x == 0
Thread 2:
  r6=p; r6.x=3;
Thread 1:
  r1=p; r2=r1.x; r3=q; r4=r3.x; r5=r1.x;
```

- in `r2,r4,r5` wird gleiche Stelle gelesen, wir erwarten Resultate `0,0,0` oder `0,0,3` oder `0,3,3` oder `3,3,3`
- transformiere Code in Thread 1 zu

  ```
  r1=p; r2=r1.x; r3=q; r4=r3.x; r5=r2;
  ```

  dann ist Resultat `0,3,0` möglich.

Def. Speichermodell (Plan)

Merksatz:

- in sequentiellen Java-Programmen bestimmt die Programm-Ordnung die Ausführungsordnung

  beides sind totale Ordnungen
- in nebenläufigen Java-Programmen *gibt es keine totale Ausführungsordnung*

  …das Speichermodell definiert nur eine *partielle* „happens-before“-Halbordnung

Def. Speichermodell (Detail)

- (JLS 17.4.3) Programm-Ordnung (P.O.)
  (total je Thread, Quelltextreihenfolge der Aktionen)
- (.4) Synchronisations-Ordnung (S.O.)
  (total, auf bestimmten Aktionen: volatile-write/read, Thread-start, Beginn, Ende, join)
- (.4) *synchronises-with* (S.W.) (partiell)
  Schreiben auf volatile-Variable v S.W. jedem folgenden (bzgl. S.O.) Lesen von v
- (.5) *happens-before* (vereinfacht): (H.B.)
  transitive Hülle von (P.O. $\cup$ S.W.)
- jede well-formed (.7) und kausale (.8) Ausführung ist erlaubt.

Data Races

- Def: ein *data race*

  ist eine Menge von Zugriffen auf eine gemeinsame Variable (Speicherstelle),

  die durch *happens-before* nicht total geordnet ist.
- Def: ein *korrekt synchronisiertes* Programm

  ist ein Programm ohne *data races*.

volatile-Variablen

vereinfachtes Denkmodell
(= Veranschaulichung der happens-before-Relation)

- jeder Thread hat eine Kopie (einen Cache) der gemeinsamen Variablen (des Heaps)
- Synchronisation nur bei bestimmten Aktionen, u.a.

  Benutzen von `volatile` deklarierten Variablen:
  - beim Lesen ein *refresh* (Heap $\rightarrow$ Cache)
  - beim Schreiben ein *flush* (Cache $\rightarrow$ Heap)

  Thread-Verwaltung:
  - start: refresh
  - join: flush

Übung JMM

- Quelltexte aus VL: `https://gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws17`

  `kw45/BC` reparieren, ausprobieren, messen (auch andere Werte für threads/steps)
- Beispiele für totale und nicht totale Halbordnungen

  Präfix-Ordnung auf Wörtern, lexikografische O. auf Wörtern, lex. O. auf Zahlenpaaren, komponentenweise O. auf Zahlenpaaren,

  Def. komponentenweises Produkt von Relationen, lex. Produkt von Relationen.
- Funktionale Schnittstellen und Lambda-Ausdrücke in

  Java $\geq$ 8

  Notation, Benutzung, Typinferenz
- Beispiel aus Goetz: JCP, Abschn. 3.2

```
volatile
    boolean running = true;
Thread 1:
    while (running) { }
Thread 2:
    running = false;
```

  (evtl. `Thread.sleep(1000)` benutzen) beobachte und diskutiere Verhalten mit und ohne `volatile`.

  Beziehung zu Definition happens-before in JLS 17.4

- weitere Beispiele für happens-before:

  finale und nicht finale Attribute bei Objektkonstruktion.

Semaphore und Monitore

Semaphore

(allgemeiner) Semaphor ist abstrakter Datentyp
mit Zustand $S \in \mathbb{N}$ und *atomaren* Operationen:

- `Wait(S)` : wenn $S > 0$ dann $S := S - 1$, sonst blockiere
- `Signal(S)` : wenn es Prozesse gibt, die auf S warten, dann wecke einen davon auf, sonst $S := S + 1$

Invariante: $S = S_0 + \#$`Signal` $- \#$`Wait`
($\#$`Wait` $=$ Anzahl der *abgeschlossenen* Aufrufe
von $\#$`Wait`, entspr. für $\#$`Signal`)
Beweis der Invarianz: Induktionsanfang
und 4 Fälle im Induktionsschritt

Semaphor: Geschichte

- E. W. Dijsktra: *Cooperating Sequential Processes, 4. The General Semaphore*, TU Eindhoven 1965 `http://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html`
- J. B. Calvert: The Origin of the Railway Semaphore (Its evolution from the optical telegraph) `http://mysite.du.edu/~jcalvert/railway/semaphor/semhist.htm`
- Monty Python: Semaphore Version of Wuthering Heights

Gegenseitiger Ausschluß (grundsätzlich)

```
Semaphore s := 1;
Gemeinsame Ressource r;
Prozeß Nr i { non_critical_section;
              Wait (s);
              critical_section; // benutze r
              Signal (s);    }
```

Eigenschaften:

- gegenseitiger Ausschluß
- *fairness* für 2 Prozesse
- für ≥ 3 Prozesse nur *progress*
- fairness für ≥ 3, wenn blockierte Prozesse in Queue (statt Menge) verwaltet werden

Gegenseitiger Ausschluß (Korrektheit)

Bezeichnungen:

- S: Zustand des Semaphors
- C: Anzahl der Prozesse in kritischem Abschnitt

Zeige Invariante: $S + C = 1$.
Beweis:

- $C = \#$`Wait` $- \#$`Signal` lt. Programmtext
- $S = 1 + \#$`Signal` $- \#$`Wait` lt. Semaphor-Invariante

aus Invariante folgt Korrektheit ($C \leq 1$)

Gegenseitiger Ausschluß in SMV

`https://gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws17/blob/master/kw46/semaphore.smv`
formuliere und prüfe Bedingungen:

- Korrektheit (gegenseitiger Ausschluß)
  `G (! (proc1.state=have & proc2.state=have))`
- Fairness?
  Nein. Deswegen ist das keine korrekte Simulation eines Semaphors.
- Liveness?

Semaphore und Monitore

- Semaphore (ist Objekt)
  Aufgabe ist Synchronisation von *Threads*
- Monitor (ist Menge von Methoden)
  Aufgabe ist Zugriffskontrolle für Daten
- Monitor kann durch Semaphor realisiert werden:
  jeder Zugriff (jede Methode) muß Semaphor erwerben und wieder abgeben
- einfache Notation in Java durch `synchronized`

Monitore in Java

- die `synchronized`-Methoden einer Klasse C bilden für jedes Objekt von C einen Monitor
  (für jedes Objekt von C kann jederzeit höchstens eine Monitor-Methode laufen)
- diese Methoden sind *re-entrant*:
  während man eine Monitor-Methode ausführt, kann man weitere Methoden des gleichen Monitors aufrufen
  (deswegen funktioniert Implementierung mit Semaphor doch nicht — diese würde verklemmen)
- Spezialfall: ein-elementiger Monitor: Code-Block

```
Object lock = new Object ();
synchronized (lock) { ... }
```


Explizites wait/notify für Monitore

- durch `synchronized` sind Erwerb und Freigabe der Sperre versteckt und automatisch richtig geschachtelt
- explizites Sperren und Freigeben sind auch möglich: Methoden `wait`, `notify`, `notifyAll`:

```
synchronized void take (..) {
  while (taken) this.wait (); taken = true;
synchronized void drop (..) {
  taken = false; this.notifyAll (); }
```

- Benutzung nur innerhalb eines Monitors,
- `wait()` in Schleife, siehe *spurious wake-ups* in `http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait--`

Beispiel: Philosophen in der Mensa

(Edsger Dijkstra, Tony Hoare, ca. 1965)

- Prozess = Philosoph
- gemeinsame Ressource = Gabel

gewünschte System-Eigenschaften:

- liveness (kein Verklemmen)

  die Folge der Aktionen ist unendlich
- fairness (kein Verhungern)

  falls ein Prozeß eine Ressource anfordert, bekommt er sie nach endlich vielen Aktionen tatsächlich

Modellierung des Ressourcenzugriffs

Modellierung des ausschließlichen Ressourcenzugriffs:

```
class Fork {
    private boolean taken = false;
    synchronized void take () {
        while (taken) { this.wait (); }
        taken = true;               }
    synchronized void drop () {
        taken = false; this.notifyAll ();
```

Q: warum wird expliziter Semaphor (wait/notify) benutzt?

A: jeder Prozeß (Philosoph) benötigt zwei Ressourcen (Gabeln) gleichzeitig, kann aber nicht zwei `synchronized`- Methoden gleichzeitig ausführen (kann die erste Gabel nicht festhalten, während die zweite geholt wird)

5 Philosophen

```
class Fork { void take() ; void drop () }
Philosoph i : new Thread () { void run () {
 while(true) { this.nachdenken();
    fork[i].take(); fork[i+1].take();
    this.essen();
    fork[i].drop(); fork[i+1].drop();
}}} . start();
```

welche Eigenschaften? wie kann man das ggf. reparieren?

- global: progress oder deadlock?
- lokal: fairness?

Quelltexte: `https://gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws17`

Übung Monitor

Verhalten dieses Programmes ausprobieren, diskutieren:

```
final Object lock = new Object();
Thread t = new Thread(() -> {
  synchronized (lock) {  lock.wait(); } });
t.start();
synchronized (lock) { lock.notifyAll(); }
t.join();
```


Übung Dining Philosphers

Algorithmen implementieren und Eigenschaften (Liveness, Fairness) diskutieren/beweisen:

- Philosoph 0 ist Linkshänder
- ein Kellner (Platzanweiser), der immer nur maximal 4 Leute an den Tisch läßt

Realisierung des Modells

- in Java, in NuSMV, mit Petrinetz

Übung Binärer Semaphor

binärer Semaphor: $S \in \{0, 1\}$ und ...

`Signal(S)` : ... sonst $S := 1$

Simulation allgemeiner Semaphor durch binären Semaphor `http://www.csc.uvic.ca/~mcheng/460/notes/gensem.pdf`

- Warum ist Solution 1 falsch,
- worin besteht Unterschied zu Solution 2?

weiter Beispiele zu Semaphoren: Allen B. Downey: *The Little Book of Semaphores*, `http://greenteapress.com/semaphores/downey08semaphores.pdf`

Software Transactional Memory

Motivation/Plan

für nebenläufige Programme, die gemeinsamen Speicher benutzen:

- bisher: Synchronisation durch Sperren (locks)

  wesentlicher Nachteil: nicht modular

- jetzt: nichtblockierende Synchronisation

Quelle: Simon Peyton Jones: *Beautiful Concurrency*, = Kapitel 24 in: Andy Oram und Greg Wilson (Hrsg.): *Beautiful Code*, O'Reilly, 2007. `http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/`

Beispiel: Kontoführung (I)

das ist das (bisher) naheliegende Modell:

```
class Account { int balance;
  synchronized void withdraw (int m)
    { balance -= m; }
  synchronized void deposit (int m)
    { withdraw (-m); }
```

welche Fehler können hier passieren:

```
void transfer
      (Account from, Account to, int m)
{
  from.withdraw (m);
  to.deposit (m);
}
```


Beispiel: Kontoführung (II)

ist das eine Lösung?

```
void transfer
       (Account from, Account to, int m)
{
  from.lock(); to.lock ();
  from.withdraw (m);
  to.deposit (m);
  from.unlock(); to.unlock();
}
```


Beispiel: Kontoführung (III)

wann funktioniert diese Lösung und wann nicht?

```
  if (from < to) { from.lock(); to.lock() }
  else           { to.lock(); from.lock() }
  ...
```


Locks are Bad

- taking too few locks
- taking too many locks
- taking the wrong locks
- taking locks in the wrong order
- error recovery
- lost wakeups, erroneous retries

locks do not support modular programming
John Ousterhout: *Why Threads are a Bad Idea (for most puroposes)* USENIX 1996, `https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf`

Speicher-Transaktionen (Benutzung)

```
from <- atomically $ newTVar 10
atomically $ do x <- readTVar from
                if x < a then retry
                else writeTVar from (x-a)
```

- Transaktions-Variablen
- Lese- und Schreibzugriffe nur innerhalb einer Transaktion
  Zugriff außerhalb ist
  - statischer (Typ-)Fehler in Haskell
  - Laufzeitfehler in Clojure
- Transaktion wird atomar und isoliert ausgeführt
  - atomar: findet komplett statt oder überhaupt nicht
  - isoliert: ... `http://blog.franslundberg.com/2013/12/acid-does-not-make-sense.html`

Speicher-Transaktionen (Implementierung)

- während der Transaktion:
  Zugriffe (Schreiben und Lesen) in Log schreiben
- am Ende (commit): prüfen, ob Log konsistent mit aktuellem Speicherzustand ist
  konsistent := die während der Transaktion gelesenen Werte stimmen mit den aktuellen überein
- ... wenn ja, dann schreiben (atomar)
- ..., wenn nicht, dann Transaktion wiederholen
- ... bei Wert-Änderung einer der gelesenen Variablen

Einzelheiten, Erweiterungen: `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/STM`

Deklaration von Nebenwirkungen in Typen

in Java (u.v.a.m.) ist der Typ nur ein Teil der Wahrheit:

```
public static int f (int x) {
  y++ ; new File ("/etc/passwd").delete();
  return x+1;
}
```

in Haskell: Typ zeigt mögliche Nebenwirkungen an.
damit kann man trennen:

- Aktion (`IO Int`)

  von Resultat (`Int`)

- Aktion, die in Außenwelt sichtbar ist (`IO Int`)

  von Aktion, die in Transaktion erlaubt ist (`STM Int`)

Nebenwirkungen in Haskell: IO a

Werte:

```
4 :: Int  ;   "foo" ++ "bar" :: String
```

Aktionen mit Resultat und Nebenwirkung:

```
writeFile "foo.text" "bar" :: IO ()
readFile "foo.text" :: IO String
putStrLn (show 4) :: IO ()
```

Nacheinanderausführung von Aktionen:

```
do s <- readFile "foo.text"
   putStrln (show (length s))
```

Start einer Aktion: im Hauptprogramm

```
main :: IO ()
main = do ...
```


Nebenwirkungen auf den Speicher

```
import Data.IORef
data IORef a -- abstrakt
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

- damit kann man die üblichen imperativen Programme schreiben (jede Variable ist eine IORef)
- die Kombinatoren zur Programmablaufsteuerung kann man sich selbst bauen, z. B.

  ```
  while :: IO Bool -> IO () -> IO ()
  ```

  Übung: while implementieren, Fakultät ausrechnen

Transaktionen: STM a

jede Transaktion soll *atomar sein*
⇒ darf keine IO-Aktionen enthalten (da man deren Nebenwirkungen sofort beobachten kann)
⇒ neuer Typ `STM a` für Aktionen mit Nebenwirkungen *nur auf Transaktionsvariablen* `TVar a`

```
type Account = TVar Int
withdraw :: Account -> Int -> STM ()
withdraw account m = do
    balance <- readTVar account
    writeTVar account ( balance - m )
transfer :: Account -> Account -> Int -> IO
transfer from to m = atomically
  ( do withdraw from m ; deposit to m    )
```


Bedingungen und Auswahl

- eine Transaktion abbrechen: `retry`
- eine Transaktion nur ausführen, wenn eine Bedingung wahr ist

  ```
  check :: Bool -> STM ()
  check b = if b then return () else retry
  ```

- eine Transaktion nur ausführen, wenn eine andere erfolglos ist: `orElse`

STM-Typen und -Operationen

```
data STM a -- Transaktion mit Resultat a
data IO  a -- (beobachtbare) Aktion
           -- mit Resultat a
atomically :: STM a -> IO a
retry      :: STM a
orElse     :: STM a -> STM a -> STM a

data TVar a -- Transaktions-Variable
            -- mit Inhalt a
newTVar    :: a -> STM ( TVar a )
readTVar   ::
writeTVar  ::
```

(= Tab. 24-1 in Beautiful Concurrency)

vgl. `http://hackage.haskell.org/packages/archive/stm/2.2.0.1/doc/html/Control-Monad-STM.html`

STM - Beispiele, Übungen

The Santa Claus Problem

Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.

J. A. Trono: *A new Exercise in Concurrency*, SIGCSE Bull. 26, 1994.

Lösung mit STM in Peyton Jones: *Beautiful Concurrency*, 2007

Philosophen mit STM

```
forM [ 1 .. num ] $ \ p -> forkIO $ forever
    atomically $ do
        take $ left  p ; take $ right p
    atomically $ drop $ left  p
    atomically $ drop $ right p
take f = do
    busy <- readTVar f
    when busy $ retry
    writeTVar f True
```

kein Deadlock (trivial). — nicht fair, siehe `http://thread.gmane.org/gmane.comp.lang.haskell.parallel/305`

Quelltexte: `https://gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws17`

Übung STM

- ein Haskell-Hauptprogramm schreiben,

  ```
  main :: IO ()
  main = do putStrLn "hello world"
  ```

  kompilieren, ausführen (Optionen sind hier nicht nötig, aber später)

  ```
  ghc -threaded -rtsopts -O2 Main.hs
  ./Main +RTS -N
  ```

- dining philosophers in Haskell, in `https://gitlab.imn.htwk-leipzig.de/waldmann/skpp-ws17`, kompilieren, ausführen. Diskutiere

  `https://mail.haskell.org/pipermail/haskell-cafe/2015-November/122233.html` Was passiert bei `if p > 1 then ...`?

- Nach Aktivieren der Anweisung

  ```
  hSetBuffering stdout NoBuffering
  ```

  wird deutlich, daß der gemeinsame Zugriff der Philosophen auf `stdout` (in `putStrLn`) ebenfalls eine Konfliktquelle ist.

  Lösen Sie dieses Problem durch einen weiteren Thread. Nur dieser darf `putStrLn` ausführen, die anderen müssen ihm das mitteilen. Benutzen Sie dazu `TVar String` (Inhalt der Ausgabe) und `TVar Bool`

  (Anforderung/Freigabe).

- Cigarette smokers problem (Allen B. Downey: The Little Book of Semaphores, `http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf`, Abschnitt 4.5) Diskutiere Deadlock in Semaphor-Lösung, schreibe (triviale) STM-Lösung.
- ggf. STM-Test mit dejafu `https://mail.haskell.org/pipermail/haskell-cafe/2015-November/122224.html`

STM in Clojure (Beispiele)

Clojure = LISP für JVM

```
(def foo (ref "bar"))  -- newTVar

(deref foo)            -- readTVar
@foo

(ref-set foo "oof")    -- writeTVar
(dosync (ref-set foo "oof"))
```

Quellen:

- Kap. 6 *Concurrency* aus: Stuart Halloway, *Programming Clojure*, Pragmatic Bookshelf, 2009;
- `http://clojure.org/refs`

STM in Clojure (Sicherheit)

Transaktionsvariablen ohne Transaktion benutzen:

- Haskell: statischer Typfehler
- Clojure: Laufzeitfehler

IO innerhalb einer Transaktion:

- Haskell: statischer Typfehler
- Clojure: “I/O and other activities with side-effects should be avoided in transaction. . . ”

Übung: ein Programm konstruieren, bei dem eine IO-Aktion innerhalb einer Transaktion stattfindet, aber die Transaktion nicht erfolgreich ist.

Transaktion mit Nebenwirkung

Transaktionen:

```
(def base 100)
(def source (ref (* base base)))
(def target (ref 0))
(defn move [foo]
   (dotimes [x base]
      (dosync (ref-set source (- @source 1))
      (ref-set target (+ @target 1))) ))
(def movers (for [x (range 1 base)] (agent n
(dorun (map #(send-off % move) movers))
```

Nebenwirkung einbauen:

```
(def c (atom 0)) ... (swap! c inc) ...
(printf c)
```


STM und persistente Datenstrukturen

“The Clojure MVCC STM is designed to work with the persistent collections, and it is strongly recommended that you use the Clojure collections as the values of your Refs. Since all work done in an STM transaction is speculative, it is imperative that there be a low cost to making copies and modifications.”

“The values placed in Refs must be, or be considered, immutable!!”

Beispiel Suchbäume:

- destruktiv: Kind-Zeiger der Knoten verbiegen,
- persistent: neue Knoten anlegen.

Bsp: persistenter Suchbaum in Haskell

Nicht blockierende Synchronsiation

Einleitung

Synchronisation (geordneter Zugriff auf gemeinsame Ressourcen) durch

- explizite Sperren (lock)
  pessimistische Ausführung
  Gefahr von Deadlock, Livelock, Prioritätsumkehr
- ohne Sperren (lock-free)
  optimistische Ausführung
  ein Prozeß ist erfolgreich (andere müssen wiederholen)
  – nur feingranular (`AtomicLong`, `compareAndSet()`)
  – atomare zusammengesetze Transaktionen

Literatur

- *Atomic Variables and Nonblocking Synchronization*, Kapitel 15 in Brian Goetz et al.: *Java Concurrency in Practice*
- *Counting, Sorting and Distributed Coordination*, Kapitel 12 in Maurice Herlihy and Nir Shavit: *The Art of Multiprocessor Programming*
- Which CPU architectures support Compare And Swap (CAS)?

  `http://stackoverflow.com/questions/151783/`

Compare-and-Set (Benutzung)

Der Inhalt einer Variablen soll um 1 erhöht werden.
Mit STM wäre es leicht:

```
atomically $ do
    v <- readTVar p ; writeTVar p $! (v+1)
```

ohne STM, mit einfachen atomaren Transaktionen:

```
AtomicInteger p;  boolean ok;
do { int v = p.get();
     ok = p.compareAndSet(v,v+1);
} while ( ! ok);
```

- Vorteil: das geht schnell (evtl. sogar in Hardware)
- Nachteil: nicht modular (keine längeren Transaktionen)
- Auswirkung: kompliziertere Algorithmen

Compare-and-Set (Implementierung)

Modell der Implementierung:

```
class AtomicInteger {  private int value;
  synchronized int get () { return value; }
  synchronized boolean
    compareAndSet (int expected, int update)
      if (value == expected) {
        value = update ; return true;
      } else {
        return false; } } }
```

moderne CPUs haben CAS (oder Äquivalent)
im Befehlssatz (Ü: suche Beispiele in x86-Assembler)

JVM (ab 5.0) hat CAS für Atomic{Integer,Long,Reference}

Compare-and-Set (JVM)

Assembler-Ausgabe (des JIT-Compilers der JVM):

```
javac CAS.java
java -Xcomp -XX:+UnlockDiagnosticVMOptions -
```

Vorsicht, Ausgabe ist groß. Mit `nohup` in Datei umleiten, nach `AtomicInteger::compareAndSet` suchen.

Non-Blocking Stack

Anwendung: Scheduling-Algorithmen:
(jeder Thread hat Stack mit Aufgaben, andere Threads können dort Aufgaben hinzufügen und entfernen)

```
private static class Node<E> {
  E item; Node<E> next;
}
class Stack<E> {
  AtomicReference<Node<E>> top
    = new AtomicReference<Stack.Node<E>>();
  public void push (E x)
  public E pop ()
}
```


Spezifikation f. Concurrent Stacks

Stack-spezifisch:

- correct set semantics

allgemein:

- linearizability
- lock-free (lebendig), wait-free (fair)

vgl. Hendler, Shavit, Yerushalmi: *A Scalable Lock-free Stack Algorithm* (Sect. 5) (16th ACM Symp. on Parallelism in Algorithms and Architectures) `http://www.cs.bgu.ac.il/~hendlerd/papers/scalable-stack.pdf`

Abstraktion, Linearisierbarkeit

- nebenläufige Implementierung N einer Datenstrukturspezifikation P
- mit *abstraction map* a von N zu einer sequentiellen Implementierung S
- N heißt *linearisierbar*, falls es für jede nebenläufige Ausführung von N eine Folge von *Linearisierungpunkten* $n_1, n_2, \ldots$ gibt,

  so daß $a(n_1), a(n_2), \ldots$ eine P-korrekte Ausführung von S ist.

vgl. Shavit: *Art of Multiproc. Prog.* Sect. 9.3 *Concurrent Reasoning*

Non-Blocking Queue (Problem)

- einfach verkettete Liste

```
private static class Node<E> {
   E item; AtomicReference<Node<E>> next; }
```

- Zeiger `head,tail` auf Anfang/Ende, benutze Sentinel (leerer Startknoten)

Auslesen (am Anfang) ist leicht,
Problem beim Einfügen (am Ende):

- zwei Zeiger `next` und `tail` müssen geändert werden,
- aber wir wollen keinen Lock benutzen.

Non-Blocking Queue (Lösung)

(Michael and Scott, 1996)
`http://www.cs.rochester.edu/research/synchronization/pseudocode/queues.html`

Idee: die zwei zusammengehörigen Änderungen mglw. durch verschiedene Threads ausführen (!)

Queue hat zwei Zustände:

- A: tail zeigt auf letzten Knoten
- B: tail zeigt auf vorletzten Knoten

wer B bemerkt, muß reparieren.

in Java realisiert als ConcurrentLinkedQueue

Non-Blocking Übung

- Simulation von CAS durch STM
- CAS-Maschinenbefehle in JVM-Assemblercode
- Messungen für Zähler mit CAS, mit `synchronized`
- Testfälle für nebenläufige Stacks/Queues
- push/pop für non-blocking Stack
- enqueue/dequeue für non-blocking Queue

Lokale Prozeßkommunikation (I)

Motivation

bisher betrachtete Modelle zur Thread-Kommunikation:

- Datenaustausch über gemeinsamen Speicher
- Synchronisation durch Locks, Transaktionen

jetzt:

- kein gemeinsamer Speicher
- Datentransport durch Nachrichten
- dabei ggf. Synchronisation

Beispiel: Rendezvous (Ada), Actors (Scala), Channels (Go)

Communicating Sequential Processes (CSP)

- abstraktes Modell für Kommunikation von Prozessen
- Abstraktion: (endliches) Alphabet von (einfachen) Nachrichten, synchrone Kommunikation
- entwickelt 1978 von C. A. R. Hoare

  `http://research.microsoft.com/en-us/people/thoare/`

- Grundlage für Prozeßmodell in Occam, Ada, Go, ...

CSP: Syntax

E ist eine Menge von Ereignissen
Die Menge $\mathbb{P}(E)$ der Prozesse über E definiert durch:

- STOP $\in \mathbb{P}$,
- wenn $e \in E$ und $P \in \mathbb{P}$, dann $(e \to P) \in \mathbb{P}$
- wenn $P_1, P_2 \in \mathbb{P}$, dann sind in $\mathbb{P}$:
  - Nacheinanderausführung: $P_1; P_2$
  - Auswahl, intern: $P_1 \sqcap P_2$, extern: $P_1 \square P_2$
  - nebenläufige Ausführung mit Kommunikationsalphabet $C \subseteq E$: $P_1 \|_C P_2$
- Wiederholung: $P_1^* \in \mathbb{P}$

CSP: Semantik

Semantik eines Prozesses $P \in \mathbb{P}(E)$ definiert durch:

- Definition des Zustandsübergangs-Systems $A(P)$
  - endlicher Automat mit Alphabet E und ϵ-Übergängen
  - Zustände sind Prozesse (d.h. Terme)
  - P ist initial, alle Zustände sind final (akzeptierend)
  - Übergänge beschrieben durch Term-Ersetzungs-Regeln
- Definition der Semantik von $A(P)$, zwei Möglichkeiten:
  - die Spur-Semantik (= die Sprache von $A(P)$)
  - die Ablehnungs-Semantik

CSP: von Prozess zu Automat

Übergangsrelation von $A(P)$ definiert durch Regeln zu

- Nacheinanderausführung:
  - $(a \to P) \xrightarrow{a} P$
  - (STOP; Q) $\xrightarrow{\epsilon} Q$,
  - wenn $P \xrightarrow{a} P'$, dann $(P; Q) \xrightarrow{a} (P'; Q)$,

  (vgl. Nil, Cons, Append für Listen)
- Wiederholung:
  - $P^* \xrightarrow{\epsilon}$ STOP, $\quad P^* \xrightarrow{\epsilon} (P; P^*)$.

  (vgl. Kleene-Hülle als Sprachoperation)
- sowie (nächste Folien) Kommunikation, Verzweigung

Regeln zur Kommunikation

das Ereignis gehört zum Kommunikations-Alphabet: beide Prozesse führen es gemeinsam (synchron) aus

- $a \in C \land P \xrightarrow{a} P' \land Q \xrightarrow{a} Q' \Rightarrow (P \|_C Q) \xrightarrow{a} (P' \|_C Q')$,

das Ereignis gehört nicht zum Kommunikations-Alphabet oder ist ein ϵ-Übergang: einer der beiden Prozesse führt es aus (der andere wartet)

- $(a = \epsilon \lor a \in E \setminus C) \land P \xrightarrow{a} P' \Rightarrow (P \|_C Q) \xrightarrow{a} (P' \|_C Q)$,
- $(a = \epsilon \lor a \in E \setminus C) \land Q \xrightarrow{a} Q' \Rightarrow (P \|_C Q) \xrightarrow{a} (P \|_C Q')$,

definiert *synchrone* Kommunikation, realisiert u.a. in Ada (Rendezvous), Scala (Operation `!?`), Go (Kanal mit Kapazität 0).

Regeln für Auswahloperatoren

- interne Auswahl (Nichtdeterminismus)
  $P \sqcap Q \xrightarrow{\epsilon} P, \quad P \sqcap Q \xrightarrow{\epsilon} Q.$
- externe Auswahl
  $P \xrightarrow{a} P' \Rightarrow (P \square Q) \xrightarrow{a} P', \quad Q \xrightarrow{a} Q' \Rightarrow (P \square Q) \xrightarrow{a} Q'.$

Beispiel: (mit verkürzter Notation a für $a \to$ STOP)

- $P_1 = (a \sqcap b)$: $A(P_1) =$ STOP $\xleftarrow{b} b \xleftarrow{\epsilon} P_1 \xrightarrow{\epsilon} a \xrightarrow{a}$ STOP
- $P_2 = (a \square b)$: $A(P_2) =$ STOP $\xleftarrow{b} P_2 \xrightarrow{a}$ STOP

diese Automaten sind verschieden, aber die Sprachen stimmen überein.

Verschiedene Prozeß-Semantiken

- Spur-Semantik:
  Menge der (unvollständigen) Spuren (*partial traces*) (jeder Automatenzustand ist akzeptierend)
  gestattet keine Beschreibung von Verklemmungen (*deadlocks*), keine Unterscheidung von interner und externer Auswahl, deswegen
- Ablehnungs-Semantik
  zur genaueren Beschreibung des Prozeßverhaltens
  Bemerkung: wenn man das nicht klar definiert, dann beginnt das große Rätselraten darüber, was *Nichtdeterminismus* für Prozesse bedeuten soll, vgl. `http://lambda-the-ultimate.org/node/4689`

Ablehnungs-Semantik

Ab-Semantik eines Prozesses ist Menge von Paaren von

- partieller Spur $s \in E^*$
- und Folge-Menge $F \subseteq E$ (mögliche nächste Ereignisse)

$(s, F) \in \text{Ab}(P) :\iff \exists Q : P \xrightarrow{s} Q \land F = \{e \mid \exists R : Q \xrightarrow{e} R\}$.
Beispiel: Ab-Semantik ist genauer als Sp-Semantik:

- $\text{Sem}_{\text{Sp}}(b \square c) = \text{Sem}_{\text{Sp}}(b \sqcap c) = \{\epsilon, b, c\}$
- $\text{Sem}_{\text{Ab}}(b \square c) = \{(\epsilon, \{b, c\}), (b, \emptyset), (c, \emptyset)\}$
- $\text{Sem}_{\text{Ab}}(b \sqcap c) = \{(\epsilon, \{b, c\}), (\epsilon, \{b\}), (b, \emptyset), (\epsilon, \{c\}), (c, \emptyset)\}$

Rendez-Vous (I) in Ada

```
task body Server is
   Sum : Integer := 0;
begin loop
      accept Foo (Item : in Integer)
        do Sum := Sum + Item; end Foo;
      accept Bar (Item : out Integer)
        do Item := Sum; end Bar;
   end loop;
end Server;
A : Server; B : Integer;
begin
   A.Foo (4); A.Bar (B); A.Foo (5); A.Bar (B
end B;
```


Rendezvous (II)

- ein Prozeß (Server) führt `accept` aus,
  anderer Prozeß (Client) führt Aufruf aus.
- beide Partner müssen aufeinander warten
- `accept Foo ( .. ) do .. end Foo` ist atomar

Rendezvous (III)

allgemeinere Formen von `accept`:

- ```
  select accept Foo (Item : in Integer)
              do .. end;
      or accept Bar ( ... )
  end select;
  ```
- ```
  when X < Y => accept Foo (...)
  select ... or terminate; end select;
  select ... or delay 1.0 ; ... end select;
  select ... else .. end select;
  ```

`http://en.wikibooks.org/wiki/Ada_Programming/Tasking`
`http://www.adaic.org/resources/add_content/`

`standards/05aarm/html/AA-9-7-1.html`

Lokale Prozeßkommunikation (II)

Kommunikations-Kanäle

zur asynchronen Kommunikation
(Eigenschaften: vgl. Postbrief statt Rendezvous)

- Kapazität des Kanals/Briefkastens
  (Kapazität 0 $\Rightarrow$ Rendezvous)
- Ordnung der Nachrichten (FIFO oder ungeordnet)
- Typisierung der Nachrichten

Bsp. in Go: (`http://golang.org`)

```
ch := make (chan int) // anlegen
ch <- 41 // schreiben
x := <- ch // lesen
```


Kanäle in Haskell

Kanal ist *typisiert*, FIFO, *unbeschränkt*.

```
data Chan a -- abstrakt
newChan   :: IO (Chan a)
writeChan ::
readChan  ::
```

Dok.: `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent-Chan.html`

Übungen

- Implementierung ansehen
- Anwendung: Aufsammeln von Teilergebnissen
- Anwendung: Mergesort in Aktor-Style
- vergleiche mit `Control.Concurrent.STM.TChan`

Haskell: MVar

ist Kanal der Kapazität 1

```
data MVar a = ...

takeMVar :: MVar a      -> IO a
-- blockiert, wenn leer

putMVar  :: MVar a -> a -> IO ()
-- blockiert, wenn voll
```


Actors (Scala)

- Briefkasten ist nicht typisiert
- Nachrichten sind typisiert

`http://www.scala-lang.org/node/242`

```
object Stop
class Server extends Actor { def act() {
    var running = true;
    while (running) { receive {
        case x : Int => println(x)
        case Stop => running = false; } } }
var s = new Server()
s.start ; s ! 42 ; s ! Stop
```


Good Actors Style

Kap. 30.5 in: Odersky, Spoon, Villers: *Programming in Scala*, Artima 2007,

- ein Aktor soll nicht blockieren
  ...sondern lieber Arbeit an andere Aktoren weitergeben
- kommuniziere mit Aktoren nur durch Nachrichten
  ...und nicht durch gemeinsame Variablen
- Nachrichten sollten *immutable* sein
  ...sonst Gefahr von inkonsistenten Daten
- Nachrichten sollten *self-contained* sein
  ...damit der Aktor nicht nachfragen muß
  unveränderliche Objekte kann man billig mitschicken

Rendezvous-Zusammenfassung

- unmittelbar synchron, kein Puffer:
  - Ada-Rendezvous (task entry call/accept)
  - Go:
    `ch = make(chan int); ch <- .. ; .. <- ch`
  - Scala: `Actor a ; ... = a !? msg`
- gepuffert synchron (Nachrichten der Reihe nach)
  - beschränkte Kapazität:
    Go: `make(chan int, 10)`
    `java.util.concurrent.LinkedBlockingQueue`
  - unbeschränkt:
    Haskell: `Control.Concurrent.newChan`
- asynchron Scala: `Actor a ; ... = a ! msg`

Übung: Kanäle in Go

Sprachdefinition: `http://golang.org/`
Compiler/Runtime:

- google: `go run hello.go`
- gcc: `gcc-go -o hello hello.go ; ./hello`

Kanäle:

- Syntax (Deklaration, Benutzung)
- Kapazität
- Schließen von Kanälen

Übung:

- berechne Summe der Bitcounts von 0 bis $2^n - 1$
- verteile Rechnung auf p Prozesse

Futures in Java

`submit` startet eine asynchrone Berechnung,
`get` ist der blockierende Zugriff auf das Resultat.
Implementierung könnte einen Kanal benutzen
(die Rechnung schreibt, `get` liest)

```
package java.util.concurrent;
interface Callable<R> { R call() }
interface ExecutorService {
  <R> Future<R> submit (Callable<R>)
  void shutdown() }
interface Future<R> { R get() }
class Executors {
  ExecutorService newFixedThreadPool(int) }
```

Ü: Bitcount-Summation. `Stream<Future<Integer>>`

Paralleles Sortieren

Einleitung

- VL SKPP: über Sprachkonzepte, nicht über Algorithmen
- trotzdem an einem Beispiel Zusammenhänge zeigen
- Sortieren ist der Standard-Testfall für Algorithmen-Entwurf und -Analyse (vgl. 2. Semester)
- aber wer tatsächlich sortiert, macht etwas falsch (hat die falsche Datenstruktur gewählt — Liste oder Array statt balancierter Baum)
- wer Sortieren auch noch selbst implementiert, macht es erst recht falsch (sollte Standardbibliotheken verwenden)
- ... außer: der Kandidat ist Student und soll etwas lernen

Kostenmodelle für parallele Alg.

- Datenabhängigkeitsgraph für (sequentiellen oder funktionalen) Algorithmus in *single-assignment*-Form

  `int f(int x) {a=x; a=a+3; b=x*5; return a+b;}`

  ⇒ `i f(i x) {a0=x; a1=a0+3; b0=x*5; return a1+b0;}`
  - Knoten: Variablen (benutzte Speicherstellen)
  - Kanten: $\{x \to a_0, a_0 \to a_1, x \to b_0, a_1 \to r, b_0 \to r\}$
- interessante Maße sind:
  - work (Anzahl aller Operationen, d.h. Knoten) (Bsp: 4)
  - depth/span (Länge eines längsten Weges) (Bsp: 3)

  span ist Laufzeit bei bestmöglicher Parallelisierung
- vgl. Guy Blelloch (`http://www.cs.cmu.edu/~guyb/`): *Parallel Thinking* (PPoPP 2009) u.ä.

Work und Span für einfache Sortierverfahren

- Auswählen: bestimme Index für Minimum $a[1 \dots n]$
  - sequentielle Implementierung: work $= $ span $= n$
  - balancierter Baum: work $= n$, span $= \log n$
- Sortieren durch Auswählen:

```
for i from 1 to n - 1:
 m := Index für minimum (a[i .. n]); a[i] <-> a[m]
```

  - sequentielle Implementierung: work $=$ span $= n^2$
  - mit balanciertem Auswählen: work $= n^2$, span $= n \log n$.
- Merge-Sort? span(sort) $= \log n \cdot$ span(merge)

  aber span(merge) $= n$ bei sequentieller Impl.

Massiv paralleles Sortieren

```
for i,j :
  boolean c[i,j] := (a[i] < a[j])
for j :
  out[j] := a[sum (c[1,j] .. c[n,j])]
```

- korrekt?

  (falls Elemente von `a` paarweise verschieden sind)
- work?
- span?
- praktisch?

Sortiernetze

- elementarer Baustein: *Komparator*

  Eingänge x, y, Ausgänge $\max(x, y), \min(x, y)$
- Sortierverfahren (für Eingabebreite n)

  $=$ Anordnung (Netz, azyklischer Graph, Schaltkreis) von Komparatoren
- unabhängige Komparatoren können parallel arbeiten
- Programm-Ablauf hängt nur von n ab, nicht von Daten.

  solche Programme heißen *data oblivious*
- Ziel: Merge-Sort, dabei Merge mit span $\log n$

Konkrete Sortiernetze geringer Breite

- $n = 1$
- $n = 2$
- $n = 3$
- $n = 4$?
- $n = 8$?

Das 0-1-Prinzip

- Satz: Netz N sortiert *alle* Eingaben aus $\mathbb{N}^n$

  $\iff N$ sortiert alle Eingaben aus $\{0, 1\}^n$
- Beweis ($\Leftarrow$)

  falls Eingabe $\vec{x} = [x_1, \dots, x_n] \in \mathbb{N}^n$ falsch behandelt,

  dann Ausgabe $\vec{y} = [y_1, \dots, y_n]$ mit $i < j$ und $y_i > y_j$.

  betrachte $f : a \mapsto$ if $a > y_j$ then 1 else 0

  f ist monoton, also schalten Komparatoren bei Eingabe $\vec{x}$ genauso wie bei $f(\vec{x})$.

  also entsteht Ausgabe $[\dots, 1, \dots, 0, \dots]$.

Odd-Even-Merge

- M_n: Merge-Netz für $n + n$ Eingaben. Spezifikation: Falls $\vec{x}$ monoton und $\vec{y}$ monoton, dann $M_n(\vec{x}, \vec{y})$ monoton.
- M_1 ?, M_2 ?, M_{2n} aus M_n ?
- $M_{2n}(x, y) = M_n(\mathrm{odd}(x), \mathrm{odd}(y)); M_n(\mathrm{even}(x), \mathrm{even}(y)); \dots$
- Ergänze (zunächst für Bsp $n = 2$, dann allgemein)
- Beweise mit 0-1-Prinzip
- work, span für Merge? für Sort?
- das so konstruierte 8-Sortiernetz ist optimal
- Ü: Implem.? (1. konstruieren, 2. effizient ausführen)
- Ü: Merge für andere Eingabebreiten, z.B. $M_{3,4}$

Verteiltes Zählen

Motivation

Motivation: zentrale Ausgabe von Tickets (mit eindeutigen und aufsteigenden Nummern).

mit höherem Durchsatz als mit einen *zentralen* Zähler

```
class Counter { int count;
synchronized int next () { return count++;}}
```

James Aspnes, Maurice Herlihy, and Nir Shavit.
Counting networks, JACM 41(5):1020–1048, Sept. 1994
`http://www.cs.yale.edu/homes/aspnes/papers/ahs-abstract.html`

wesentlicher Baustein: `AtomicBoolean.negate()`

Spezifikation für Zählnetze

korrekte Behandlung der Token:

- Netzwerk mit n Eingängen, n Ausgängen, Tiefe d
- jedes Token, das einen Eingang betritt, verläßt das Netzwerk nach $\leq d$ Schritten an einem Ausgang (das Netzwerk vergißt und erfindet keine Token)

gute Verteilung der Token:

- (informal) bei *beliebiger* Verteilung der Token auf die Eingänge: jeder Ausgang wird (etwa) *gleich oft* benutzt.
- (formal) betrachte Anzahlen $[x_1, \dots, x_n]$ der Token je Eingang, Anzahlen $[y_1, \dots, y_n]$ der Token je Ausgang; im *Ruhezustand* ($\sum_{i=1}^n x_i = \sum_{i=1}^n y_i$) soll gelten: $[y_1, \dots, y_n]$ ist *Schrittfolge*: $y_1 \geq \dots \geq y_n \geq y_1 - 1$

Folgerung aus Spezifikation für Zählnetze

Satz: für jedes $n > 0, S \geq 0$ gibt es *genau eine* Schrittfolge $[z_1, \dots, z_n]$ mit $S = \sum z_i$.

Satz: für *jeden* Zustand jedes Zählnetzes gilt:

- wenn $\sum x_i - \sum y_i = D > 0$

  (es befinden sich noch D Token im Netz),
- dann gilt $\forall i : z_i - \boxed{} \leq y_i \leq z_i$

  wobei $[z_1, \dots]$ die (eindeutige) Schrittfolge mit $\sum z_i = \sum x_i$

Folgerung: auch wenn der Ruhezustand nie eintritt, sind die Ausgänge gut verteilt
(hoher Durchsatz $\implies$ kleines D $\implies$ gute Verteilung)

Netzwerke aus Verteilern

Verteiler:

- ein *Verteiler* (balancer) ist Schaltkreis mit zwei Eingängen, zwei Ausgängen, einem Zustand.
- Wenn Zustand *hoch*, erscheint nächstes Eingangstoken am oberen Ausgang. Wenn Zustand *tief*, am unteren.
- Nach jedem Token wechselt der Zustand.

Eigenschaften/Fragen:

- jeder Verteiler ist ein Zählnetz für 2 Eingänge
- gibt es Zählnetze aus Verteilern (z. B. für 4 Eingänge)?
- kann man diese systematisch konstruieren?

Bitonisches Zählen und Zusammenfügen (I)

Ansatz für Konstruktion eines 2^k-Zählnetzes aus Verteilern:

- Zählnetze C benutzen Teilnetzwerke M, deren *Eingangsfolgen* (nach Induktion) Schrittfolgen sind.

  (vergleiche `mergesort`: die Funktion `merge` wird nur auf geordnete Folgen angewendet)
- Konstruktion der Zählnetze: Induktionsanfang: $C_1(x_1) =$

  Induktionsschritt: $C_{2n}(x_1, \dots, x_{2n}) =$
  $C_n(x_1, \dots, x_n); C_n(x_{n+1}, \dots x_{2n}); M_{2n}(x_1, \dots, x_n; x_{n+1}, \dots, x_2$
- Konstruktion der Merge-Netze: (Spezifikation?)

  Induktionsanfang: $M_2(x_1, x_2)$;   Induktionsschritt?

Bitonisches Zählen und Zusammenfügen (II)

Induktionsschritt:

$$M_{2n}(\vec{x}, \vec{y}) = \begin{cases} M_n(\operatorname{odd} \vec{x}, \operatorname{even} \vec{y}); \\ M_n(\operatorname{even} \vec{x}, \operatorname{odd} \vec{y}); \\ V(x_1, x_2); \dots; V(y_{n-1}, y_n) \end{cases}$$

mit $V(p, q)$ = Verteiler, $\operatorname{odd}(x_1, x_2, \dots) = (x_1, x_3, \dots)$, $\operatorname{even}(x_1, x_2, \dots) = (x_2, x_4, \dots)$.

Satz: jedes solche M_n erfüllt die Spezifikation.

Übung: konstruiere C_4, M_4

Übung: Beweis für M_8 mit Eingangsfolge $(3, 3, 3, 2; 9, 9, 8, 8)$, unter der Annahme, daß der Satz für M_4 gilt.

Übung: Beweis für M_{2n} mit beliebiger Eingangsfolge, unter der Annahme, daß der Satz für M_n gilt.

Implementierung für Verteiler und Netze

Plan:

```
struct Balancer {
    AtomicBoolean state;
    Balancer [Boolean] next;
}
traverse (Balancer b) {
    while (nicht fertig) {
        boolean i = b.state.getAndNegate();
        traverse(b.next[i]);          } }
```

Aufgaben:

- implementiere `negate`
- implementiere Verteiler mit STM

Anwendungen von Zählnetzen

`http://www.cs.yale.edu/homes/aspnes/papers/ahs-abstract.html` Section 5

- verteiltes Zählen

  n Ein/Ausgänge, an jedem Ausgang ein Zähler mit Schrittweite n
- verteilter Producer/Consumer-Puffer

  benutzt zwei Netze der Breite n zum verteilten Zählen

  sowie n 1-Element-Container
- Synchronisationsbarriere (vgl. CyclicBarrier)

Übung Zählnetze

Beweise: die folgenden Bedingungen sind äquivalent:

- $(x_1, \ldots, x_n)$ ist Schrittfolge
- $\forall 1 \le i < j \le n : 1 \ge x_i - x_j \ge 0$.
- Wenn $m = \sum x_i$, dann $\forall i : x_i = \lceil \frac{m-i+1}{n} \rceil$

Wenn x eine Schrittfolge ist, welche Beziehungen gelten zwischen $\sum \text{odd}(x), \sum(x)/2, \sum \text{even}(x)$?
(Möglichst genau! Benutze ggf. $\lceil\cdot\rceil, \lfloor\cdot\rfloor$)

Beweise: Wenn x und y gleichlange Schrittfolgen mit $\sum x = 1 + \sum y$, dann gilt für alle bis auf ein i: $x_i = y_i$.
Was gilt stattdessen für dieses i?

periodische Zählnetze

Parallele Auswertungsstrategien

Überblick

- bei Ausdrücken $f(X, Y)$ kann man Werte von X und Y parallel und unabhängig berechnen,
- wenn die Auswertung von X und Y nebenwirkungsfrei ist.
- im einfachsten Fall sind *alle* Ausdrücke nebenwirkungsfrei (Haskell)
- Haskell benutzt Bedarfsauswertung.
  Strategie-Kombinatoren und -Annotationen erzwingen frühere/verteilte Auswertung von Teilausdrücken:
  - Kombinatoren: `par X (pseq Y (f X Y))`
  - Strategie-Annotationen: `` xs `using` parList rseq ``

Algebraische Datentypen und Pattern Matching

ein Datentyp mit zwei Konstruktoren:

```
data List a
   = Nil              -- nullstellig
   | Cons a (List a) -- zweistellig
```

Programm mit Pattern Matching:

```
length :: List a -> Int
length xs = case xs of
    Nil       -> 0
    Cons x ys -> 1 + length ys
```

beachte: Datentyp rekursiv ⇒ Programm rekursiv

```
append :: List a -> List a -> List a
```


Alg. Datentypen (Beispiele)

```
data Bool = False | True
data Maybe a = Nothing | Just a
data Tree a =
    Leaf | Branch ( Tree a ) a ( Tree a)
```

Ü: inorder, preorder, leaves, depth

Ü: Schlüssel in Blättern

```
data N = Z | S N
```

Ü: Rechenoperationen

Notation für Listen in Haskell:

anstatt `data List a = Nil | Cons a (List a)`

wird benutzt `data [a] = [] | (a : [a])`

Bedarfsauswertung

- Konstruktoren werten Argumente (zunächst) nicht aus

  statt Wert (struct) wird Programm (thunk) gespeichert, das den Wert ausrechnen kann

  (ähnlich einer `Future`, die noch nicht gestartet wurde)
- Konstruktor wird erst bestimmt, wenn er für Pattern Matching benötigt wird

  (Vergleich mit einem Konstruktor eines Musters in `case`)
- dann wird thunk durch Wert überschrieben
- Wert kann selbst wieder thunks enthalten

Normalformen

- ein Objekt, das kein thunk ist, heißt (schwache) *Kopfnormalform* (whnf)

  d.h., der *oberste* Konstruktor ist bestimmt

  Teilobjekte können thunks sein
- ein Objekt, das kein thunk ist und keine thunks enthält, heißt *Normalform*

  d.h., *alle* Konstruktoren sind bestimmt
- Zahlen, Zeichen sind auch Normalformen

  aber Typ `Int` bedeutet „thunk oder Zahl“:

  `data Int = GHC.Types.I# GHC.Prim.Int#`

Experiment zu Liste, Sequence, whnf

```
ghci
:set +s

let xs = [1..10^6] :: [Int]
seq xs () ; head xs ; last xs ; last xs

import qualified Data.Sequence as S
let ys = S.fromList xs
seq ys () ; S.index ys 0
S.index ys (10^6-1)
```


Bedarfsauswertung und Parallelisierung

- GHC RTS verwaltet *spark pool*,

  Spark ist (Zeiger auf) thunk, Pool ist Menge von Sparks
- parallele worker threads bestimmen whnf der sparks
- Spark (für `x`) wird erzeugt durch `par x y`
- Schicksal der Sparks (Statistik: `./Main +RTS -N -s`)
  - *converted*: worker überschreibt thunk durch whnf
    (das ist anzustreben: worker hilft Hauptprogramm)
  - *fizzled*: Hauptprogramm überschreibt thunk durch whnf
    (das ist schlecht: worker hat umsonst gearbeitet)
  - *garbage collected*: whnf wurde nicht benötigt

Beispiel: Mergesort

Sequentieller Algorithmus, wesentlicher Teil:

```
msort :: Ord a => [a] -> [a] -> [a]
msort xs =
  let ( here, there ) = split xs
      sh = msort here ; st = msort there
  in  merge sh st
```

parallelisiert durch: `import Control.Parallel`

```
  .. in  par sh $ pseq st $ merge sh st
```

- Datentyp `[a]` nützt hier wenig:
  Cons ist lazy, whnf enthält ggf. nur ein Cons.
- `Data.Sequence.Seq a` ist besser
  ist balancierter Baum, deswegen whnf $\approx$ nf

Beispiel: Primzahlen

Aufgabe: bestimme $\pi(n) :=$ Anzahl der Primzahlen in $[1..n]$ auf naive Weise (durch Testen und Abzählen)

```
num_primes_from_to :: Int -> Int -> Int
num_primes_from_to lo hi
 = length $ filter id $ map prime [lo .. hi]
prime :: Int -> Bool
```

parallele Auswertung durch Strategie-Annotation

```
withStrategy ( parListChunk 100000 rdeepseq
   ( map prime [lo..hi] )
```

getrennte Beschreibung von *Wert* und *Strategie*

Beschreibungssprache (EDSL) für Strategien

http://hackage.haskell.org/package/parallel/docs/Control-Parallel-Strategies.html

Parallel LINQ

Beispiel:

```
(from n in Enumerable.Range(lo, hi-lo)
                    .AsParallel()
 where Prime(n)  select true).Count ();
```

Typen:

- `System.IEnumerable<E>`
- `System.Linq.ParallelEnumerable<E>`

http://msdn.microsoft.com/en-us/library/dd997425.aspx

Übung:

- paralleles `foreach`
- Steuerung der Parallelität durch `Partitioner`

Übung

- Laufzeitanalyse mit Threadscope

```
ghc -O2 -eventlog -threaded Pr.hs
./Pr 6 +RTS -N2 -l
threadscope Pr.eventlog
```

- Primzahlen:
  - finde beste *chunk size* für `map isprime [..]`
  - warum ist parallele Auswertung innerhalb von `isprime` unzweckmäßig?
- bitcounts summieren

Flexible Parallelisierung durch assoziative Operationen

Motivation

Beispiel Summe eine Liste durch beliebige Klammerung von zweistelligen Additionen

```
sum [3,1,2,4] = ((3+1)+2)+4 = 3+(1+(2+4))
```

wähle den Berechnungsbaum, der am besten zu verfügbarer Hardware (Anzahl Prozessoren) paßt

```
 ... = (3+1)+(2+4)
```

Verallgemeinerung statt Addition: beliebige assoziative Operation, nicht notwendig kommutativ, d.h. Blatt-Reihenfolge muß erhalten bleiben

Monoide

- eine Struktur A mit Operation $\circ_A$ und Element 1_A heißt *Monoid*, falls:
  - $\circ_A$ ist assoziativ
  - 1_A ist links und rechts neutral für $\circ_A$
- Beispiele:

  Listen, Verkettung, leerer Liste

  natürliche Zahlen, Addition, 0
- Monoid-Morphismen von Listen lassen sich flexibel parallelisieren
- Beispiele: Länge, Summe.

Homomorphismen

homo-morph = gleich-förmig

Signatur Σ (= Menge von Funktionssymbolen)

Abbildung h von Σ-Struktur A nach Σ-Struktur B ist Homomorphismus, wenn:

$\forall f \in \Sigma, x_1, \dots, x_k \in A :$
$$h(f_A(x_1, \dots, x_k)) = f_B(h(x_1), \dots, h(x_k))$$

Beispiel:

- Σ = Monoid-Signatur (Element 1, binäre Operation $\circ$)
- $A = \text{List a}$ (Listen) mit $1_A = \text{Nil}, \circ_A = \text{append}$
- $B = \text{N}$ (Zahlen) mit $1_B = 0, \circ_B = \text{plus}$
- $h = \text{length}$

Homomorphismen von Listen

Abbildung `[a] -> b` ist gegeben durch

- Bild der leeren Liste `:: b`
- Bild der Einerliste `:: a -> b`
- Verknüpfung `:: b -> b -> b`

```
foldb :: b -> (a->b)->(b->b->b) -> [a]-> b
foldb n e f xs = case xs of
  [] -> n ; [x] -> e x
  _ -> let (l,r) = splitAt ... xs
       in  f (foldb n e f l) (foldb n e f r)
```

Satz: `f` assoziativ und `n` links und rechts neutral für `f` ⇒ `foldb n e f` ist Monoid-Homomorphismus von `([a],[],++)` nach `(b,n,f)`

Maximale Präfix-Summe

```
mps :: [Int] -> Int
mps xs = maximum $ map sum $ inits xs

mps [1,-1,0,2,0,-1] = maximum [0,1,0,0,2,2,1
```

ist kein Homomorphismus (Gegenbeispiel?) aber das:

```
mpss :: [ Int ] -> ( Int, Int )
mpss xs = ( mps xs, sum xs )
```

Bestimme die Verknüpfung in `(Int,Int)`

(die Darstellung als `foldb`)

beweise Assoziativität

Sequentielle Folds

für sequentielle Berechnung sind geeignet

```
foldr :: ( a -> b -> b) -> b -> [a] -> b
foldr f e xs = case xs of
  [] -> e ;  x : ys -> f x (foldr f e ys)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f e xs = case xs of
  [] -> e ;  x : ys -> foldl f (f e x) ys
```

`foldl` paßt zum Iterator-Muster (Berechnung über einen Stream, mit einem Akkumulator), vgl. `Aggregate` in C#/Linq

Homomorphie-Sätze

1. für jeden Hom exist. Zerlegung in map und reduce — und das reduce kann man flexibel parallelisieren!

   Bsp: `length = reduce (+) . map (const 1)`

   `map`: parallel ausrechnen, `reduce`: balancierter Binärbaum.
2. jeden Hom. kann man als foldl und als foldr schreiben
3. (Umkehrung von 2.) Wenn eine Funktion sowohl als foldl als auch als foldr darstellbar ist, dann ist sie ein Hom. — und kann (nach 1.) flexibel parallelisiert werden

   m.a.W: *aus der Existenz zweier sequentieller Algorithmen folgt die Existenz eines parallelen Alg.*

Literatur

- Jeremy Gibbons: *The Third Homomorphism Theorem*, Journal of Functional Programming, May 1995.

  `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.45.2247&rep=rep1&type=pdf`
- Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, Masato Takeichi: *Automatic Inversion Generates Divide-and-Conquer Parallel Programs*, PLDI 2007.

Übung Assoc.

- ist `f::[a] -> Bool` mit `f xs =` „die Länge von `xs` ist gerade " ein Morphismus?

  desgl. für „. . . ist durch 3 teilbar"

  wenn nein: Gegenbeispiel lt. Definition,

  wenn ja: Implementierung mit `foldb`

- desgl. für

```
f :: [a] -> Maybe a
f xs = case xs of
   [] -> Nothing ; x:ys -> Just x
```

- `mss :: [Int] -> Int` maximale Segmentsumme

(über alle zusammenhängenden Teilfolgen)

durch ein assoziative Verknüpfung auf angereicherter Struktur (Hinweis: 3-Tupel von Zahlen)

- `carry :: [(Bool,Bool)] -> Bool` Übertrag bei Addition von Binärzahlen

  Hinweis: betrachte `f :: [(Bool,Bool)]->(Bool,Bool)` mit Spezifikation

  `f xys = (p,q)` gdw. `\cin -> p `xor` (q && cin)` beschreibt ausgehenden Übertrag

- Diskutiere Parallelisierung von

`up :: Ord a => [a] -> Int` Länge einer längsten aufsteigenden (mglw. lückenhaften) Teilfolge.

Diskussion: Kommutativität

Methode aus PLINQ (`http://msdn.microsoft.com/en-us/library/ff963547.aspx`)

```
Aggregate<S, A, R>(
  this ParallelQuery<S> source,
  Func<A> seedFactory,
  Func<A, S, A> updateAccumulatorFunc,
  Func<A, A, A> combineAccumulatorsFunc,
  Func<A, R> resultSelector);
```

- in Haskell nachbauen (Typ und Implementierung)
- `combine` muß kommutativ sein (`http://blogs.msdn.com/b/pfxteam/archive/2008/01/22/7211660.aspx`) — warum?

Das Map/Reduce-Framework

Schema und Beispiel

```
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

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. `https://research.google.com/archive/mapreduce.html`
- Ralf Lämmel: *Google's MapReduce programming model - Revisited*, Science of Computer Programming - SCP , vol. 70, no. 1, pp. 1-30, 2008 `https://userpages.uni-koblenz.de/~laemmel/MapReduce/paper.pdf`

Implementierungen

- Haskell:

  wenige Zeilen, zur Demonstration/Spezifikation
- Google:

  C++, geheim
- Hadoop:

  Java, frei (Apache-Projekt, Hauptsponsor: Yahoo)

  `http://hadoop.apache.org/`

Implementierung in Haskell

```
import qualified Data.Map as M

map_reduce :: ( Ord ki, Ord ko )
  => ( (ki, vi) -> [(ko,vm)] )  -- ^ distrib
  -> ( ko -> [vm] -> vo ) -- ^ collect
  -> M.Map ki vi -- ^ eingabe
  -> M.Map ko vo -- ^ ausgabe
map_reduce distribute collect input
    = M.mapWithKey collect
    $ M.fromListWith (++)
    $ map ( \ (ko,vm) -> (ko,[vm]) )
    $ concat $ map distribute
    $ M.toList $ input
```


Anwendung: Wörter zählen

```
main :: IO ()
main = do
  files <- getArgs
  texts <- forM files readFile
  let input = M.fromList $ zip files texts
      output = map_reduce
        ( \ (ki,vi) -> map ( \ w -> (w,1) )
                           ( words vi ) )
        ( \ ko nums -> Just ( sum nums))
        input
  print $ output
```

wo liegen die Möglichkeiten zur Parallelisierung?
(in diesem Programm nicht sichtbar.)

Hadoop

Bestandteile:

- verteiltes Dateisystem
- verteilte Map/Reduce-Implementierung

Betriebsarten:

- local-standalone (ein Prozeß)
- pseudo-distributed (mehrere Prozesse, ein Knoten)
- fully-distributed (mehrere Knoten)

Voraussetzungen:

- java
- ssh (Paßwortfreier Login zwischen Knoten)

Hadoop-Benutzung

- (lokal) konfigurieren

  `conf/{hadoop-env.sh,*-site.xml}`
- Service-Knoten starten

  `bin/start-all.sh --config /path/to/conf`
- Job starten

```
bin/hadoop --config /path/to/conf \\
    jar examples.jar terasort in out
```

Informationen:

- Dateisystem: `http://localhost:50070`,
- Jobtracker: `http://localhost:50030`

Wörter zählen

```
public static class TokenizerMapper
   extends Mapper<Object, Text, Text, IntWri
   public void map(Object key, Text value, C
public static class IntSumReducer
   extends Reducer<Text,IntWritable,Text,Int
   public void reduce(Text key, Iterable<Int
 }
public static void main(String[] args) { ...
   job.setMapperClass(TokenizerMapper.class)
   job.setCombinerClass(IntSumReducer.class)
   job.setReducerClass(IntSumReducer.class);
```

`hadoop/src/examples/org/apache/hadoop/examples/WordCount.java`

Sortieren

vgl. `http://sortbenchmark.org/`, Hadoop gewinnt 2008.
Beispielcode für

- Erzeugen der Testdaten
- Sortieren
- Verifizieren

(jeweils mit map/reduce)

Index-Berechnung

- Eingabe: `Map<Quelle,List<Wort>>`
- Ausgabe: `Map<Wort,List<Quelle>>`

Spezialfall: Quelle = Wort = URL, ergibt „das Web".

Page Rank (I)

„Definition": eine Webseite (URL) ist wichtig, wenn wichtige Seiten auf sie zeigen.

- Eingabe: Matrix `link :: (URL, URL) -> Double` mit `link(u,v)` = Wahrscheinlichkeit, daß der Besucher von u zu v geht.
- Gesucht: Vektor `w :: URL -> Double` mit `w * link = w`

Modifikationen für

- eindeutige Lösbarkeit
- effiziente Lösbarkeit

Page Rank (Eindeutigkeit)

- aus der Link-Matrix: Sackgassen entfernen (dort zufällig fortsetzen)
- diese Matrix mit völlig zufälliger Verteilung überlagern

Resultat ist (quadr.) stochastische Matrix mit positiven Einträgen, nach Satz von Perron/Frobenius

- besitzt diese einen eindeutigen größten reellen Eigenwert
- und zugehöriger Eigenvektor hat positive Einträge.

Page Rank (Berechnung)

durch wiederholte Multiplikation:
beginne mit $w_0 =$ Gleichverteilung,
dann $w_{i+1} = L \cdot w_i$ genügend oft
(bis $|w_{i+1} - w_i| < \epsilon$)

diese Vektor/Matrix-Multiplikation kann ebenfalls mit Map/Reduce ausgeführt werden.
(Welches sind die Schlüssel?)
(Beachte: Matrix ist dünn besetzt. Warum?)

Quelle: Massimo Franceschet: *PageRank: Standing on the Shoulders of Giants* Comm. ACM 6/2011, `http://cacm.acm.org/magazines/2011/6/108660`

Übung Map/Reduce

- PLINQ/Aggregate:

  wie oft und für welche Argumente werden aufgerufen: update, combine? (abhängig von MaxDegreeOfParallelism)

- Map/Reduce (Haskell):

  Wordcount ausprobieren (Beispieltext: `http://www.gutenberg.org/etext/4300`)

  Wie bestimmt man die häufigsten Wörter? (allgemeinster Typ? Implementierung mit map/red?)

  wie multipliziert man (dünn besetze) Matrizen mit map/red?

- Map/Reduce mit Hadoop:

  `http://www.michael-noll.com/tutorials/`
  `running-hadoop-on-ubuntu-linux-multi-node-c`

Hardwarenahe Parallelität: CUDA

Beispiel

- Aufgabe: gesucht ist
  - Konfiguration von P Punkten im Einheitsquadrat,
  - deren minimaler gegenseitiger Abstand maximal ist.

  (`http://www2.stetson.edu/~efriedma/cirinsqu/`)
- Lösungsplan: zufällige lokale Suche
- Teilaufgabe dabei: bewerte eine Liste von K Konfigurationen
- Parallelisierungsmöglichkeiten:
  - Bewertung für alle Konfigurationen gleichzeitig
  - Abstandsberechnung für alle Punktepaare gleichzeitig
- das ist *massiv* parallel $(K = 128, P = 16)\ldots$

  solche Prozessoren gibt es *preiswert*

  CUDA ist eine C-Erweiterung zur Programmierung dafür

Sequentielle und parallele Lösung (Ansatz)

```
for (int k = 0; k < K; k++) {
  float mindist = +infinity;
  for (int p = 0; p < P; p++) {
    for (int q = p + 1; q < P; q++ ) {
      float s = abstand (c[k][p], c[k][q]);
      mindist = MIN (mindist, s); } }
  c[k].fitness = mindist;  }

dim3 blocks (K, 1) ; dim3 threads (P, P);
kernel <<<blocks,threads>>> (c);
__global__ kernel (config *c) {
  int k = blockIdx.x;
  int p = threadIdx.x; int q = threadIdx.y;
  if (p < q) { float s = abstand (c[k][p], c[k][q]);
  // Berechnug des MIN?
}
```


Threadkommunikation

Threads in einem Block haben gemeinsamen *shared memory*

```
__global__ kernel (config *c) {
  __shared__ float dist [P*P];

  if (p < q) dist[p*P+q] = abstand(c[k][p], c[k][q]);
  else       dist[p*P+q] = +infinity;
```

Synchronisationspunkt für *alle* (!) Threads in einem Block:

```
  __syncthreads();
```

erst danach kann man Ergebnisse aufsammeln

Binäre Reduktion

sequentiell (linear)

```
float accu = +infty;
for ( ... ) { accu = MIN (accu, dist[ ..]); }
```

parallel (logarithmisch)

```
int tid = P * p + q; int gap = threadsPerBlock / 2;
while ( gap > 0 ) {
  if (dist[tid] > dist[tid + gap])
         dist[tid] = dist[tid + gap];
  __syncthreads ();
  gap /= 2;
}
```

Resultat verarbeiten

```
if (0 == tid) c[k] = dist[0];
```


Datentransport zwischen Host und Decive

```
config * dev_c ;
cudaMalloc ( (void**) &dev_c, K*sizeof(config) );

cudaMemcpy ( dev_c, c, K*sizeof(config),
   cudaMemcpyHostToDevice );
...
cudaMemcpy ( c, dev_c, K*sizeof(config),
   cudaMemcpyDeviceToHost );
cudaFree (dev_c);
```

. . . das ist teuer (⇒ möglichst wenige Daten transportieren)

Zusammenfassung CUDA

CUDA-Programmierung ist

- programmiersprachlich einfach
- algorithmisch anspruchsvoll

  (imperative parallele Programmierung)

jeweils projektspezifisch sind festzulegen

- was läuft auf Device, was auf Host?
- Abmessung der Threadblöcke auf dem Device?
- Host realisiert Software-API (Steuerprogramm in Hochsprache)

Alternativen:

- OpenCL `https://www.khronos.org/opencl/`
- accelerate/CUDA-backend

  `http://hackage.haskell.org/package/accelerate`

Literatur CUDA

- David B Kirk, Wen-mei W Hwu, *Programming Massively Parallel Processor*, Morgan Kaufmann, 2010
- Jason Sanders, Edward Kandrot: *CUDA by Example*, Addison-Wesley, 2011 `http://developer.nvidia.com/cuda-example-introduction-general-purpose-gpu-progr`

Zusammenfassung,Ausblick

Zusammenfassung

- Zustandsübergangssystem, Petri-Netz
- Spursprache, Omega-reguläre Wörter, Temporal-Logik
- sicherer Zugriff auf gemeinsamen Speicher: Semaphore, Transaktionen (STM), atomare Transaktionen (CAS)
- Prozesse mit lokalem Speicher: CSP, Bisimulation, Kanäle, Aktoren, verteilte Programme
- deterministischer Parallelismus: Strategie-Annotationen, balancierte folds, map/reduce

Beispielklausur (alt) `http://www.imn.htwk-leipzig.`

`de/~waldmann/edu/ss11/skpp/klausur/`

Parallele Algorithmen

- in SKPP haben wir programmiersprachliche Konzepte betrachtet, um parallele Algorithmen zu implementieren.
- woher kommen die Algorithmen?
  paralleles Suchen, Sortieren, Matrixmultiplizieren, ...
  Bsp: `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/twodim/shear/shearsorten.htm`
  dazu braucht man eine eigenen Vorlesung, vgl.
- Joseph JaJa: *Introduction to Parallel Algorithms* (Addison-Wesley, 1992) ISBN-10: 0201548569
- *Algorithm Design: Parallel and Sequential* (CMU, aktuell) `http://www.parallel-algorithms-book.com/`

Komplexitätstheorie für parallele Algorithmen

Klassen:

- NC $=$ polylogarithmische Zeit, polynomielle Anzahl von Prozessoren
- P $=$ polynomielle Zeit
- NC $\subseteq$ P

Reduktionen:

- $\leq_L$ logspace-Reduktion, Eigenschaften
- P-vollständige Probleme (Bsp: Auswertung beliebiger Schaltkreise)

(vgl. für sequentielle Algorithmen: Klasse NP, Polynomialzeitreduktion $\leq_P$, NP-Vollständigkeit)