Data Parallel Haskell

Data Parallel Haskell

Inhalt:

(1) Ziele und Syntax von DPH

(2) Komplexes Beispiel: Das n-Körper Problem

(3) Implementierung von DPH

(4) Benchmark

Was ist Data Parallel Haskell?

- Erweiterung von Haskell um Sprachelemente mit denen Parallelität in Programmen ausgedrückt werden kann
- Implementiert im Glasgow Haskell Compiler (GHC)
  → derzeit noch in einem frühen Stadium
- Wesentliches Merkmal: Verschachtelte („Nested“) Datenparallelität

Parallelrechner

Obere Leistungsgrenze für Rechenleistung einzelner Prozessoren scheint allmählich erreicht.

Vermehrt Einsatz von Parallelrechnern:
Viele Einheiten, die gleichzeitig Berechnungen ausführen können.

Sichtbarkeit für den Programmierer:

implizite Parallelität

- automatisierte Parallelisierung
- Datenparallele Sprachen

Implizite Kontroll-parallelität
(Glasgow Parallel Haskell - GPH)

Prozesskontroll-sprachen
(Eden)

explizite Parallelität

- Haskell + MPI
- Concurrent Haskell

Programmierer drückt aus, wo Parallelität vorhanden ist (Annotationen, Datenparallelität)

Programmierer muss Prozesserzeugung und Kommunikation weitgehend selbst kontrollieren

(Flache) Datenparallelität

Idee: Führe die gleiche (sequentielle) Berechnung auf einer großen Menge unterschiedlicher Daten aus (SIMD)

```
-- f sei eine rechenintensive Funktion
f :: Integer -> Integer


x = map f [1..1000]
```

Vollständige Auswertung von x aufwändig.

Wenn viele Prozessoren vorhanden sind kann die Auswertung der Listenelemente auf diese verteilt werden.

(Flache) Datenparallelität

Vorteile:

- Überschaubar, da sequentielle Denkweise („Vektoroperationen“)
- Auf vielen Architekturen verfügbar (Vektorrechner, verteilte Systeme, Mehrprozessorsysteme, Multicores)
- Gute Skalierbarkeit

Nachteil:

- Beschränkte Anwendbarkeit:
  Viele Algorithmen mit flacher Datenparallelität nicht formulierbar

  (unregelmäßige Datenstrukturen z.B. dünn besetzte Matrizen nicht bequem handhabbar)

Verschachtelte Datenparallelität

Idee:
Erhöhe die Flexibilität durch Möglichkeiten Parallelverarbeitung zu verschachteln:

```
-- f sei eine rechenintensive Funktion
f :: Integer -> Integer

g :: [:[:Integer:]:] -> [:[:Integer:]:]
g = mapP (mapP f)

x :: [:[:Integer:]:]
x = [:[:1, 2, 3:], [:812, 44, 9:], [:24:]:]

-- Aufruf mit verschachtelter Parallelität
g x
```

Verschachtelte Datenparallelität

Aufruf g x führt zu:

```
g x ⇒ mapP (mapP f) [:[:1, 2, 3:], [:812, 44, 9:], [:24:]:]

    ⇒ [: (mapP f [:1, 2, 3:]), (mapP f [:812, 44, 9:]),
         (mapP f [:24:]):]

    ⇒ [: [:(f 1), (f 2), (f 3):], [:(f 812), (f 44), (f 9):],
         [:(f 24):] :]
```

Verschachtelte Datenparallelität

Aufruf g x führt zu:

```
g x ⇒ mapP (mapP f) [:[:1, 2, 3:], [:812, 44, 9:], [:24:]:]

    ⇒ [: (mapP f [:1, 2, 3:]), (mapP f [:812, 44, 9:]),
         (mapP f [:24:]):]

    ⇒ [: [:(f 1), (f 2), (f 3):], [:(f 812), (f 44), (f 9):],
         [:(f 24):] :]
```

Auswertung eines Elements führt auf weitere parallele Auswertung

→ Keine flache Datenparallelität

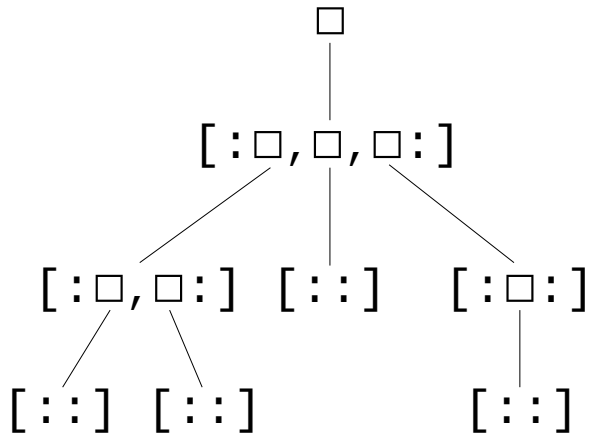
→ Verschachtelte Datenparallelität !

Verschachtelte Datenparallelität

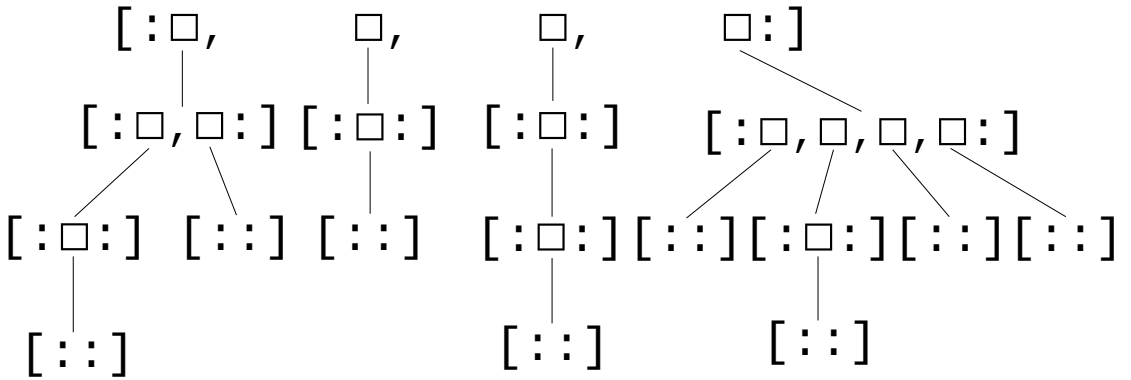
Wunsch: Verschachtelte Parallelität auch für algebraische Datenstrukturen

```
data Tree a = Node a [:Tree a:]

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Node val [::] = Node (f val) [::]
mapTree f Node val ts   = Node (f val) (mapP (mapTree f) ts)
```

Baum



paralleles Array von Bäumen



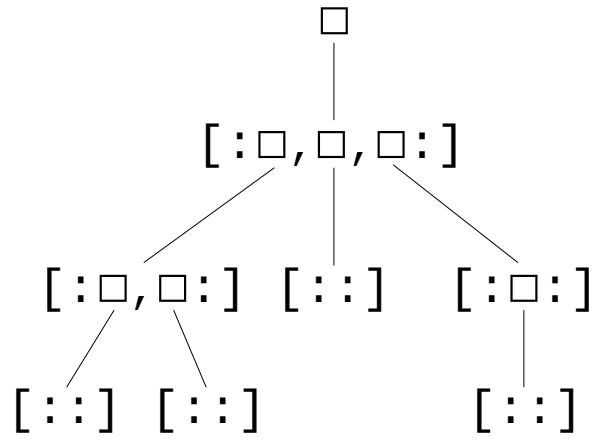
Verschachtelte Datenparallelität

Wunsch: Vers… …isch…
Datenstruktur…

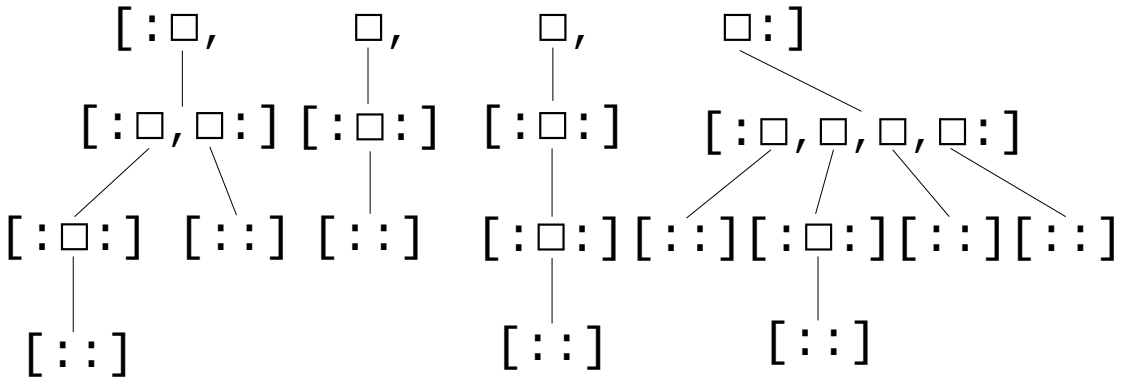
Parallelverarbeitung wird nur „ebenenweise“ möglich sein.

```
data Tree a = Nod…

mapTree :: (a -> b) -> Tr…  -> Tree b
mapTree f Node val [::] =  …de (f val) [::]
mapTree f Node val ts   =  …ode (f val) (mapP (mapTree f) ts)
```

Baum



paralleles Array von Bäumen

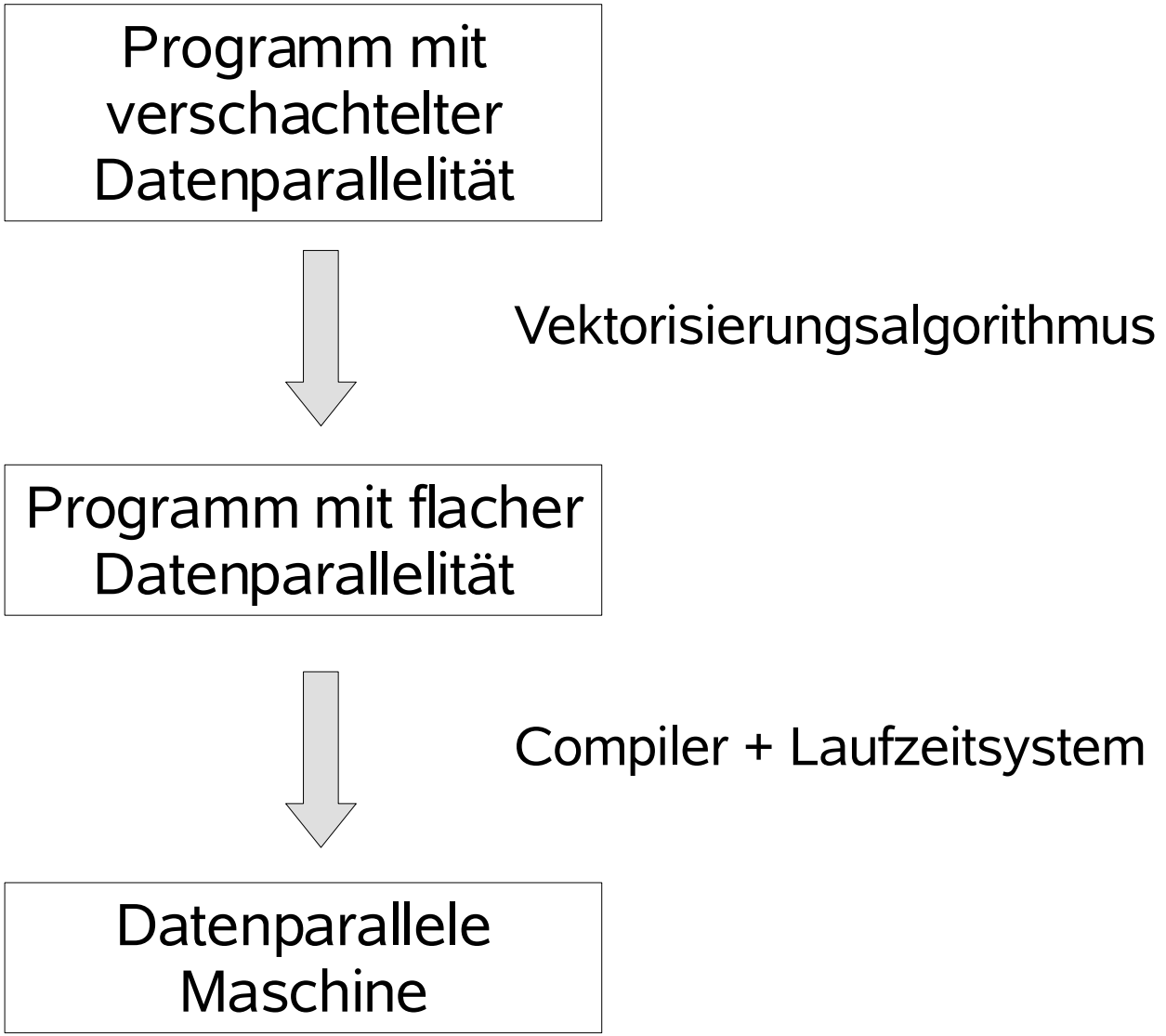


Data Parallel Haskell

Ziele:

- Stelle verschachtelte Datenparallelität bereit
- Verwende dazu eine weit verbreitete Sprache: Haskell
- Implementierung soll von der Architektur nur die Möglichkeit der flachen Parallelverarbeitung fordern
  (!)

Vektorisierung



Data Parallel Haskell

Erweiterung von Haskell um:

(1) Polymorphen Typ: Parallele Arrays vom Typ e:

```
[: e :]
```

(2) Parallele Funktionen: Operieren auf parallelen Arrays, z.B.

```
mapP :: (a->b) -> [:a:] -> [:b:]
```

Data Parallel Haskell

(1) Parallele Arrays vom Typ e: `[: e :]`

Weitgehend analog zu Listen:

```
[:1, 2, 3, 4, 5:] :: [:Integer:]
[:sin, cos:] :: Floating a => [:a -> a:]

-- parallel array comprehensions
add1 :: [:Integer:] -> [:Integer:]
add1 xs = [:x + 1 | x <- xs:]
```

Wesentliche Unterschiede:

- Striktheit: Auswertung eines Elements eines parallelen Arrays führt zur Auswertung **aller** Elemente
- Die **Auswertung** erfolgt parallel

Data Parallel Haskell

(2) Parallele Funktionen

Soweit möglich analog zu bekannten Haskell Funktionen für Listen

```
mapP        :: (a->b) -> [:a:] -> [:b:]
zipP        :: [:a:] -> [:b:] -> [:(a,b):]
zipWithP    :: (a->b->c) -> [:a:] -> [:b:] -> [:c:]
filterP     :: (a->Bool) -> [:a:] -> [:a:]
lengthP     :: [:a:] -> Int

replicateP :: Int -> a -> [:a:]
concatP     :: [:[:a:]:] -> [:a:]
unconcatP   :: [:[:a:]:] -> [:b:] -> [:[:b:]:]
transposeP :: [:[:a:]:] -> [:[:a:]:]
combineP    :: [:Bool:] -> [:a:] -> [:a:] -> [:a:]
expandP     :: [:[:a:]:] -> [:b:] -> [:b:]
```

Data Parallel Haskell

Beispiele für parallele Funktionen:

```
x :: [:[:Integer:]:]
x = [:[:1, 2, 3:], [:4, 5:], [:6, 7:]:]

concatP x
```

liefert `[:1, 2, 3, 4, 5, 6, 7:]`

```
unconcatP [:[:sin, sin, sin:], [:sin, sin:], [:sin, sin:]:]
                                              [:1, 2, 3, 4, 5, 6, 7:]
```

oder

```
unconcatP x [:1, 2, 3, 4, 5, 6, 7:]
```

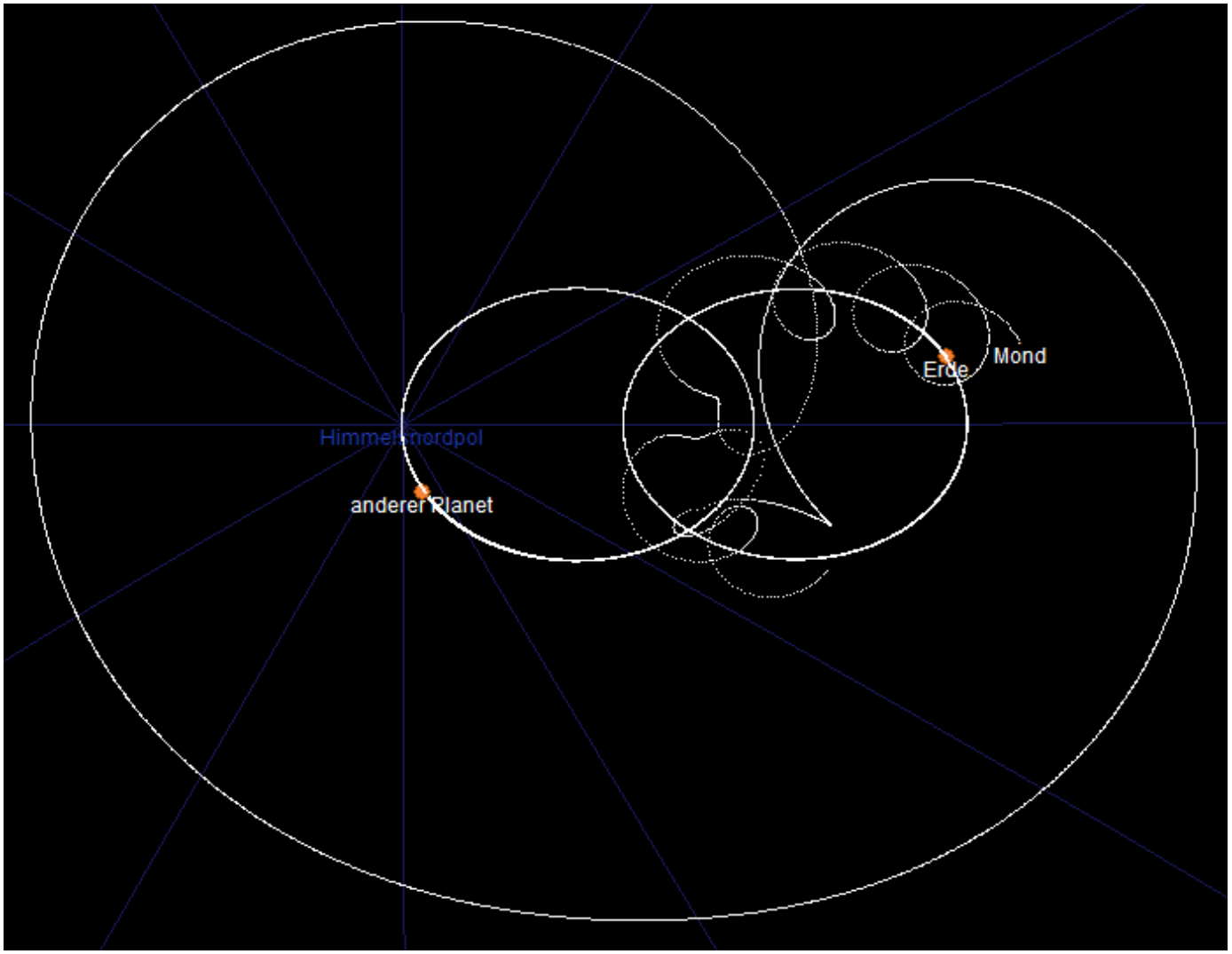
liefert `[:[:1, 2, 3:], [:4, 5:], [:6, 7:]:]`

```
transposeP x
```

liefert

```
[:[:1, 4, 6:], [:2, 5, 7:], [:3:]:]
```

Komplexes Beispiel: n-Körper-Problem



Das n-Körper-Problem

Gegeben: n Körper im $\mathbb{R}^m$ mit Massen $m_i \in \mathbb{R}_+ \quad i \in \{1,...,n\}$

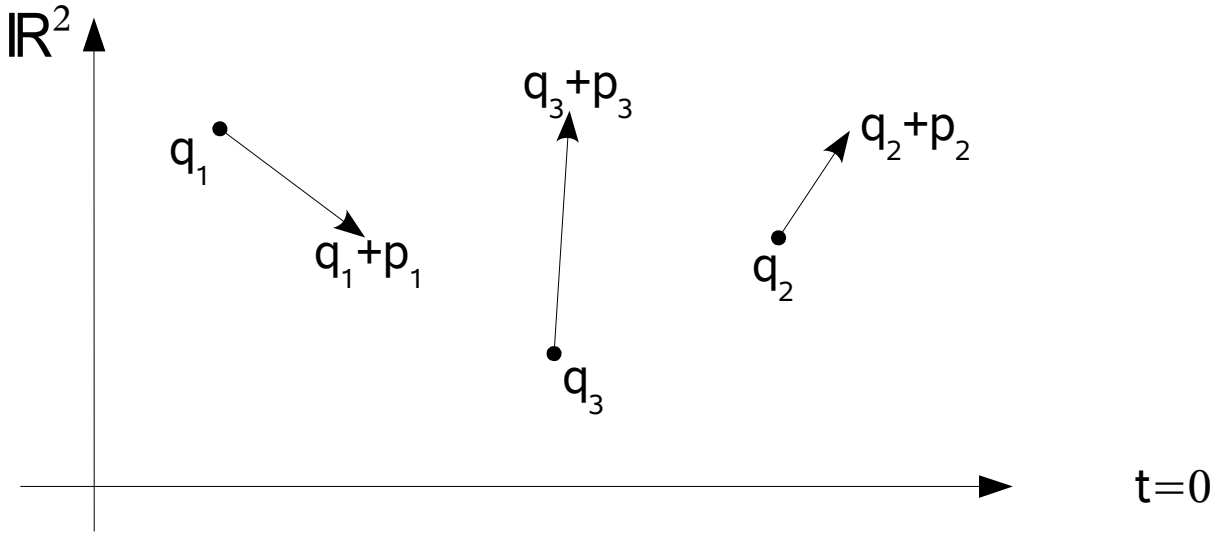
Näherung: Körper sind punktförmig, also durch

$$q_i \in \mathbb{R}^m \quad i \in \{1,...,n\}$$

gegeben. Außerdem ist für jeden Körper seine Geschwindigkeit

$$p_i \in \mathbb{R}^m \quad i \in \{1,...,n\}$$

zum Anfangszeitpunkt $t=0$ gegeben.



Das n-Körper-Problem

Gesucht: Bewegungsbahnen

Position: $Q_i: [0,T] \to \mathbb{R}^m \quad t \to Q_i(t) \quad \text{mit } Q_i(0) = q_i$

Geschwindigkeit: $P_i: [0,T] \to \mathbb{R}^m \quad t \to P_i(t) \quad \text{mit } P_i(0) = p_i$

$i \in \{1,\ldots,n\}$

unter der Annahme, dass die Gravitationskräfte aus der Newtonschen (klassischen) Mechanik wirken:
Ein Körper $q_1 \in \mathbb{R}^m$ mit Masse $m_1 \in \mathbb{R}_+$ übt auf einen Körper $q_2 \in \mathbb{R}^m$ mit Masse $m_2 \in \mathbb{R}_+$ die Kraft F_{12} mit

$$F_{12} = -G \frac{m_1 m_2}{\|q_2 - q_1\|^2} \cdot \frac{q_2 - q_1}{\|q_2 - q_1\|} \in \mathbb{R}^m$$

aus. Dabei ist

$$G = (6{,}67428 \pm 0{,}00067) \cdot 10^{-11} \frac{m^3}{kg \cdot s^2}$$

die Gravitationskonstante.

Das n-Körper-Problem

Dazu:

- Diskretisiere Zeitintervall $[0,T]$ durch äquidistante Unterteilung mit Schrittweite s

- Approximiere die Lösung zu diesen diskreten Zeitpunkten
  $t_k := k \cdot s \quad k \in \{0,\ldots,l\}$

→ Es ist ein Algorithmus zum Berechnen eines Zeitschritts gesucht

Das n-Körper-Problem

Naive Idee:

- Berechne in jedem Zeitschritt die Kraft F_{ij}, die der Körper i auf den Körper j ausübt für alle $i,j \in \{1,\ldots,n\}$ und damit die Summen

$$F^j := \sum_{\substack{i=1,\ldots,n \\ i \neq j}} F_{ij}$$

- Berechne neue Positionen und Geschwindigkeiten der Körper:

$$Q_i(t_{k+1}) = Q_i(t_k) + P_i(t_k) \cdot s$$

$$P_i(t_{k+1}) = P_i(t_k) + \frac{F^i}{m_i} \cdot s$$

wobei die Newtonsche Formel

$$F = m \cdot a$$

benutzt wurde um die neue Geschwindigkeit zu berechnen.

Das n-Körper-Problem

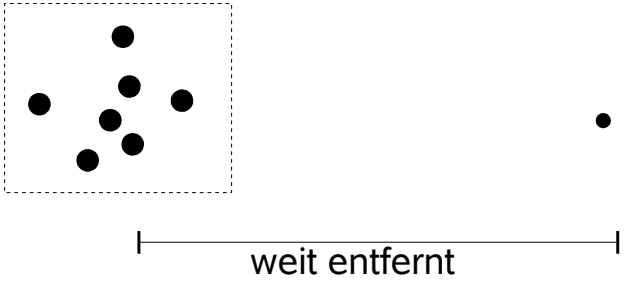
Problem:

Es müssen n^2 Wechselwirkungen berechnet werden

Komplexität: $O(n^2)$

Idee: Barnes-Hut Algorithmus

Behandle Körper die nahe beieinanderliegen aus Sicht eines weit entfernten Körpers als einen einzelnen Körper.



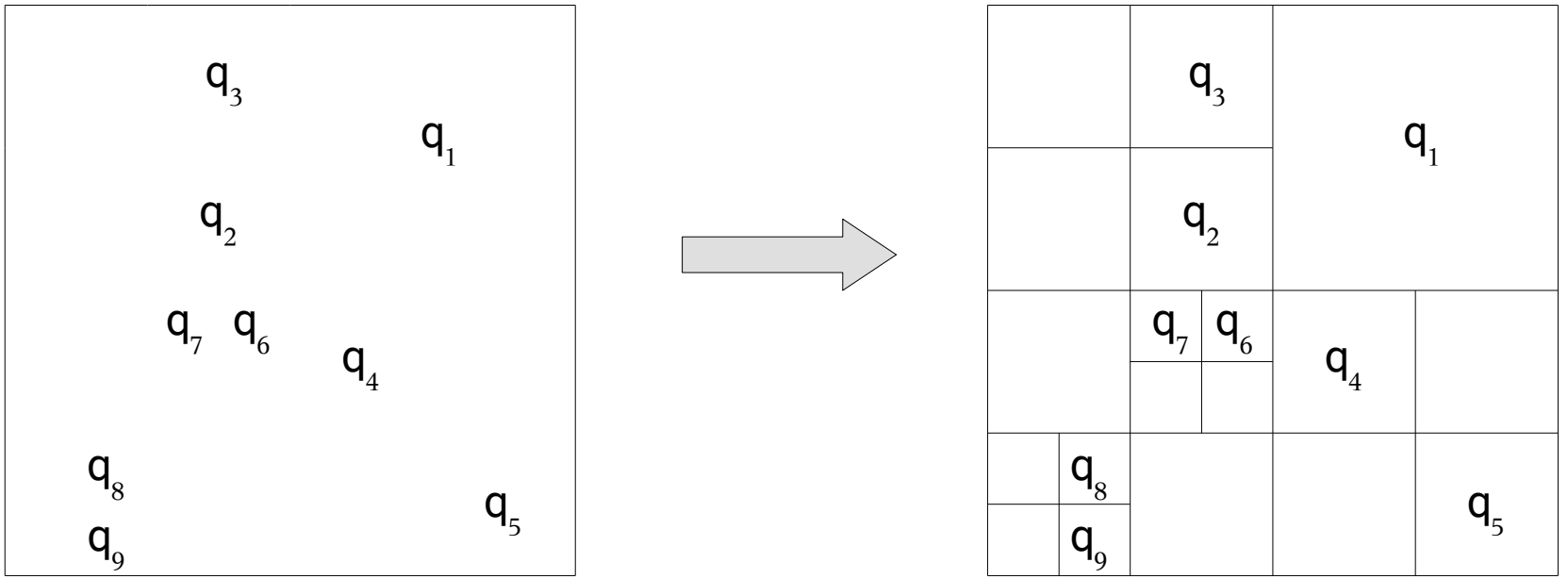
Man kann zeigen: Komplexität $O(n \cdot \log(n))$ erreichbar

Der Übersichtlichkeit halber sei von nun an $m=2$ (zweidimensionaler Raum)

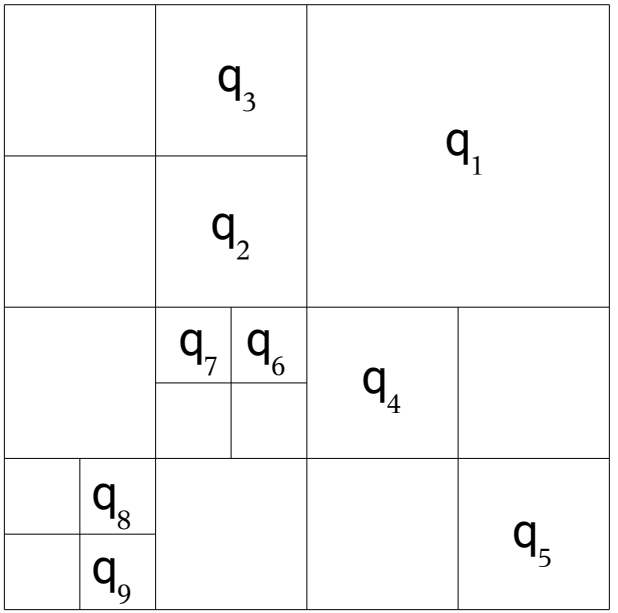
Barnes-Hut Algorithmus

(1) Hierarchische Gruppierung der Körper bestimmen

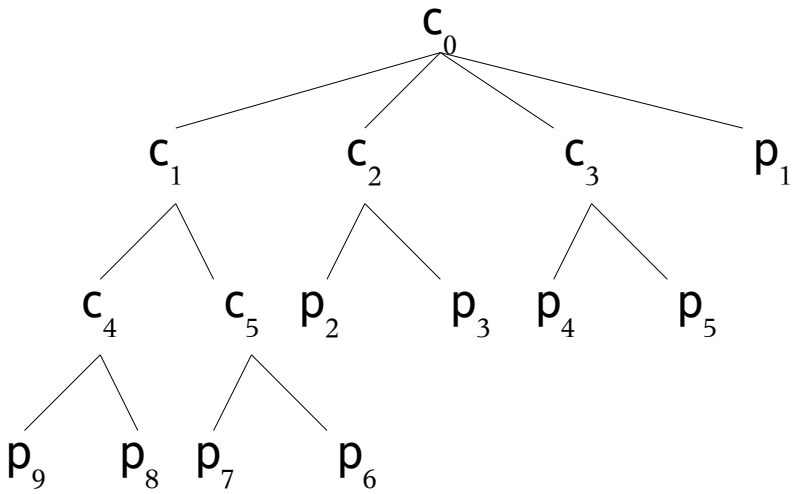
- Gesamtbereich in dem sich die Körper befinden in vier Teilbereiche aufteilen
- Körper anhand der Teilbereiche gruppieren
- Wiederholen, bis in einem Bereich höchstens ein Körper vorhanden



Barnes-Hut Algorithmus



Gruppierung wird in einer Baumstruktur gespeichert:



Barnes-Hut Algorithmus

(2) Kräfte berechnen, die auf jeden der Körper wirken

Für jeden Körper i:

- Durchlaufe Baum von Wurzel an
- An aktuellem Knoten:
  - Ist Körper vom Zentrum des Knotens weit entfernt:
    → Berechne vom Zentrum wirkende Kraft → Fertig
  - Sonst:
    → Berechne rekursiv Kräfte der Unterbäume und summiere

Implementierung mit DPH

Bereitstellen notwendiger Datentypen:

```
type Mass     = Float
type Vector   = (Float, Float)
type Area     = (Vector, Vector)
type Force    = Vector
type Velocity = Vector
type Location = Vector

-- beschreibt einen Körper der Simulation
data Particle = Particle { mass     :: Mass
                         , location :: Location
                         , velocity :: Velocity}

-- beschreibt den Baum, der die Gruppierungs-
-- information enthält
data Tree = Node Mass Location [:Tree:]
```

Implementierung mit DPH

Bereitstellen von Basisoperationen:

```
add :: Vector -> Vector -> Vector
add (x1,x2) (y1,y2) = (x1 + y1, x2 + y2)

mul :: Float -> Vector -> Vector
mul s (x1,x2) = (s*x1, s*x2)
```

Implementierung mit DPH

Implementierung des Algorithmus durch Funktion:

```
oneStep :: Area -> Float -> [:Particle:] -> [:Particle:]
```

Bereich in dem sich die Körper bewegen

Zeitintervall s (Schrittweite)

Die Körper mit ihren Daten zum Zeitpunkt t

Die Körper mit ihren Daten zum Zeitpunkt t + s

Implementierung mit DPH

Struktur der Implementierung:

(1) Baum mit Gruppierungsinformationen aufbauen:

```
buildTree
```

(2) Vom Baum auf die Körper wirkende Kräfte berechnen:

```
calcForces
```

(3) Neue Körperpositionen und -geschwindigkeiten berechnen:

```
moveParticles
```

(1) buildTree

Aufbauen des Baums mit den Gruppierungsinformationen:

```
buildTree :: Area -> [:Particle:] -> Tree
```

Bereich in dem sich die Körper bewegen

Die Körper mit ihren Daten zum Zeitpunkt t

Baum mit Gruppierungs-informationen

(1) buildTree

Hilfsfunktionen:

```
-- Gibt zurück, ob ein Körper in einem Gebiet liegt
-- oder nicht
inArea :: Area -> Particle -> Bool

-- Zerlegt ein Gebiet in die 4 Teilgebiete
splitArea :: Area -> [:Area:]

-- Berechnet das Zentrum einer Menge von Bäumen
calcCentroid :: [:Tree:] -> (Mass, Location)
```

Implementierung erfolgt direkt – muss nicht angegeben werden.

(1) buildTree

Aufbauen des Baums mit den Gruppierungsinformationen:

```
buildTree :: Area -> [:Particle:] -> Tree

buildTree area [: p :]   = Node (mass p) (location p) [::]
buildTree area particles = Node m l subtrees
  where
   (m,l)    = calcCentroid subtrees
   subtrees = [: buildTree a ps
              | a <- splitArea area
              , let ps = [:p | p <- particles, inArea a p:]
              , lengthP ps > 0 :]
```

(1) buildTree

Aufbauen des Baums mit den Gruppierungsinformationen:

```
buildTree :: Area -> [:Particle:] ->

buildTree area [: p :]   = Node (                    ::]
buildTree area particles = Node m
  where
   (m,l)    = calcCentroid subtrees
   subtrees = [: buildTree a ps
              | a <- splitArea area
              , let ps = [:p | p <- particles, inArea a p:]
              , lengthP ps > 0 :]
```

Datenparallele Auswertung (verschachtelt)

(2) calcForces

Berechnung der wirkenden Kräfte:

```
calcForces :: Float -> Tree -> [:Particle:] -> [:Force:]
```

Länge des Bereichs in dem sich die Körper bewegen

Wird zur Definition was „weit entfernt“ ist benutzt

Baum mit Gruppierungs-informationen

Definiert das Gravitationsfeld der Simulation

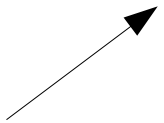
Körper die sich im Gravitationsfeld befinden

Kräfte die auf die einzelnen Körper wirken

(2) calcForces

Hilfsfunktionen:

```
-- Berechnet die Kraft, die auf den Körper
-- (1. Argument) von der Masse am gegebenen Ort
-- (2. und 3. Argument) ausgeübt wird nach dem
-- Newtonschen Gravitationsgesetz
forceOn :: Particle -> Mass -> Location -> Force

-- Summiert Kräfte
sumForces :: [:Force:] -> Force
sumForces fs = foldP add (0,0) fs

-- Entscheidung was "weit entfernt" bedeutet, lassen
-- wir offen
isFar :: Float -> Location -> Particle -> Bool
```



Länge des
Gesamtbereichs

(2) calcForces

Berechnung der wirkenden Kräfte (Version 1):

```
calcForce :: Float -> Tree -> Particle -> Force
calcForce len (Node m l ts) p
    | (isFar len l p) || (lengthP ts = 0)
                      = forceOn p m l
    | otherwise       = sumForces [: calcForce len t p | t <- ts :]
```

(2) calcForces

Berechnung der wirkenden Kräfte (Version 1):

```
calcForces :: Float -> Tree -> [:Particle:] -> [:Force:]
calcForces len tree ps = mapP (calcForce len tree) ps


calcForce :: Float -> Tree -> Particle -> Force
calcForce len (Node m l ts) p
    | (isFar len l p) || (lengthP ts = 0)
                      = forceOn p m l
    | otherwise       = sumForces [: calcForce len t p | t <- ts :]
```

(2) calcForces

Berechnung der wirkenden Kräfte (Version 1):

```
calcForces :: Float -> Tree -> [:Particle:] -> [:Force:]
calcForces len tree ps = mapP (calcForce len tree) ps


calcForce :: Float -> Tree -> Particle -> Force
calcForce len (Node m l ts) p
    | (isFar len l p) || (lengthP ts = 0)
                      = forceOn p m l
    | otherwise       = sumForces [: calcForce len t p | t <- ts :]
```

Datenparallele Auswertung (verschachtelt)

(2) calcForces

Problem (Einschub zu Problemen beim Erstellen paralleler Programme)

Der komplette Baum ist potentiell für die Auswertung jedes Arrayelements erforderlich.

```
mapP (calcForce len tree) ps
```

→ Er muss auf allen Recheneinheiten ggf. komplett verfügbar sein

→ Großer Bedarf an Kommunikation

(2) calcForces

Berechnung der wirkenden Kräfte (Version 2):

```
calcForces :: Float -> Tree -> [:Particle:] -> [:Force:]
calcForces len (Node m l ts) ps
 = let
    far_forces    = [: forceOn p m l | p <- ps, isFar len l p :]
    near_ps       = [: p | p <- ps, not (isFar len l p) :]
    near_forces_s = [: calcForces (len / 2) t near_ps | t <- ts :]
    near_forces   = [: sumForces p_forces
                     | p_forces <- transposeP near_forces_s :]
   in
    combineP [:isFar len l p | p <- ps:] far_forces near_forces
```

(2) calcForces

Berechnung der wirkenden Kräfte (Version 2):

```
calcForces :: Float -> Tree -> [:Particle:] -> [:Force:]
calcForces len (Node m l ts) ps
 = let
    far_forces    = [: forceOn p m l | p <- ps, isFar len l p :]
    near_ps       = [: p | p <- ps, not (isFar len l p) :]
    near_forces_s = [: calcForces (len / 2) t near_ps | t <- ts :]
    near_forces   = [: sumForces p_forces
                     | p_forces <- transposeP near_forces_s :]
   in
    combineP [:isFar len l p | p <- ps:] far_forces near_forces
```

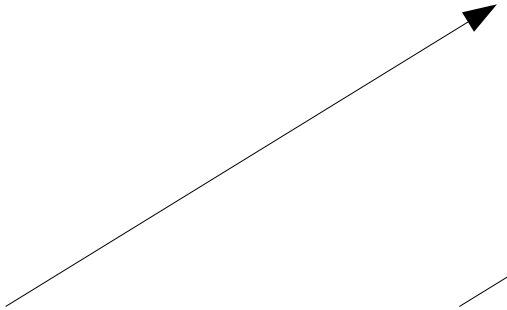
Welche Version ist „besser“?

Derartige Probleme sind kaum vermeidbar wenn man parallele Programme schreibt.

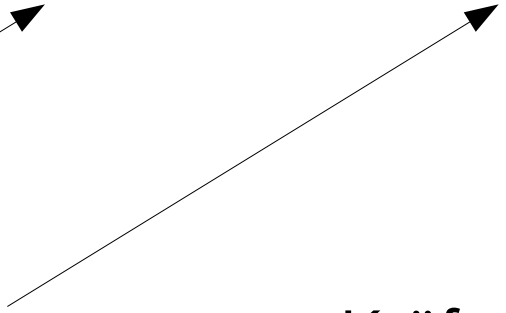
(3) moveParticles

Anwenden der Kräfte auf die Körper:

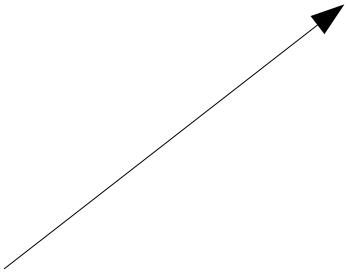
```
moveParticles :: Float -> [:Particle:] -> [:Force:] ->
                                          [:Particle:]
```



Zeitintervall s (Schrittweite)



Die Körper mit ihren Daten zum Zeitpunkt t



Kräfte die zum Zeitpunkt t auf die Körper wirken

Die Körper mit ihren Daten zum Zeitpunkt t+s

(3) moveParticles

Anwenden der Kräfte auf die Körper:

```
moveParticle :: Float -> Particle -> Force -> Particle
moveParticle timestep
             Particle { mass = m
                      , location = loc
                      , velocity = vel}
             force
             = Particle { mass = m
                        , location = newloc
                        , velocity = newvel}
             where
              accel = (1/m) `mul` force
              newloc = loc `add` (timestep `mul` vel)
              newvel = vel `add` (timestep `mul` accel)
```

(3) moveParticles

Anwenden der Kräfte auf die Körper:

```
moveParticles :: Float -> [:Particle:] -> [:Force:] ->
                                                  [:Particle:]
moveParticles ts ps fs = zipWithP (moveParticle ts) ps fs

moveParticle :: Float -> Particle -> Force -> Particle
moveParticle timestep
             Particle { mass = m
                      , location = loc
                      , velocity = vel}
             force
             = Particle { mass = m
                        , location = newloc
                        , velocity = newvel}
             where
              accel = (1/m) `mul` force
              newloc = loc `add` (timestep `mul` vel)
              newvel = vel `add` (timestep `mul` accel)
```

(3) moveParticles

Anwenden der Kräfte auf die Körper:

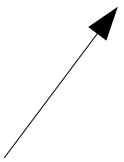
```
moveParticles :: Float -> [:Particle:] -> [:Force:] ->
                                                  [:Particle:]
moveParticles ts ps fs = zipWithP (moveParticle ts) ps fs

moveParticle :: Float -> Particle -> Force -> Particle
moveParticle timestep
             Particle { mass =
                      , lo
                      , v
             force
             = Particle {
                        , locati        wloc
                        , velocity = newvel}
             where
              accel = (1/m) `mul` force
              newloc = loc `add` (timestep `mul` vel)
              newvel = vel `add` (timestep `mul` accel)
```

Datenparallele Auswertung (flach)

Implementierung von oneStep

Erinnerung:

```
oneStep :: Area -> Float -> [:Particle:] -> [:Particle:]
```



Bereich in dem sich die Körper bewegen

Zeitintervall s (Schrittweite)

Die Körper mit ihren Daten zum Zeitpunkt t

Die Körper mit ihren Daten zum Zeitpunkt t + s

Implementierung von oneStep

Hilfsfunktion:

```
-- Berechnet die Länge eines Gebiets
lengthOf :: Area -> Float
```

Implementierung von oneStep

Wir haben eben implementiert:

```
buildTree     :: Area -> [:Particle:] -> Tree
calcForces    :: Float -> Tree -> [:Particle:] -> [:Force:]
moveParticles :: Float -> [:Particle:] -> [:Force:] ->[:Particle:]
```

Implementierung von oneStep

Wir haben eben implementiert:

```
buildTree     :: Area -> [:Particle:] -> Tree
calcForces    :: Float -> Tree -> [:Particle:] -> [:Force:]
moveParticles :: Float -> [:Particle:] -> [:Force:] ->[:Particle:]
```

Damit kann der Algorithmus formuliert werden:

```
oneStep :: Area -> Float -> [:Particle:] -> [:Particle:]
oneStep area time particles = moveParticles time particles forces
  where
    tree   = buildTree area particles
    forces = calcForces (lengthOf area) tree particles
```

Verwendung von Parallelität

`buildTree`: Verschachtelte Datenparallelität

```
[: buildTree a ps
| a <- splitArea area
, let ps = [:p | p <- particles, inArea a p:]
, lengthP ps > 0 :]
```

`calcForces`: Verschachtelte Datenparallelität

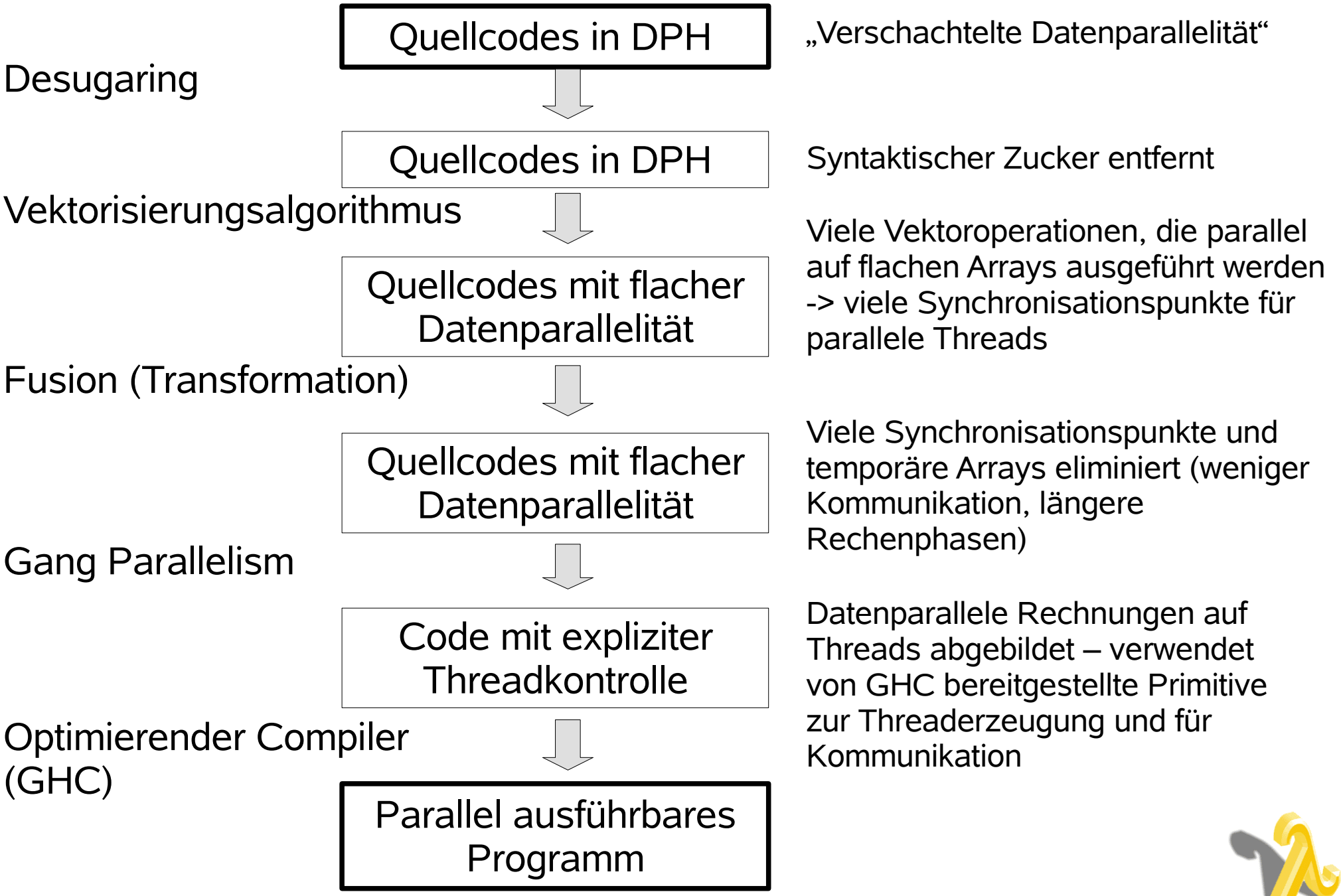
```
mapP (calcForce len tree) ps
sumForces [: calcForce len t p | t <- ts :]
```

`moveParticles`: Flache Datenparallelität

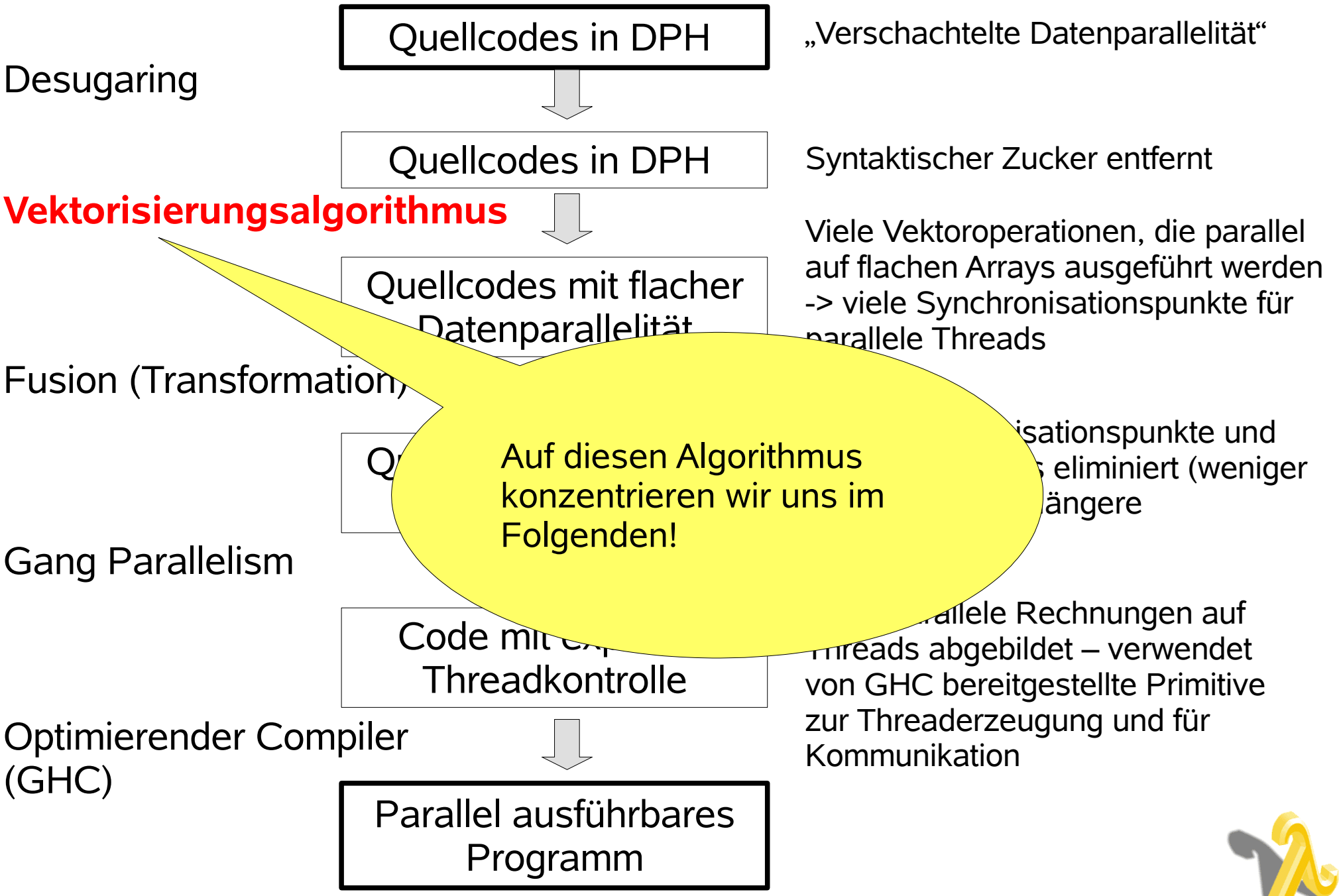
```
zipWithP (moveParticle ts) ps fs
```

Implementierung von DPH

Implementierung von DPH



Implementierung von DPH



Vektorisierung

Transformation der Funktionen, Datentypen und algebraischen Datenstrukturen der Quellcodes

$$V_t[\![]\!] : \text{Type} \rightarrow \text{Type}$$

$$V[\![]\!] : \text{Function} \rightarrow \text{Vect. Function}$$

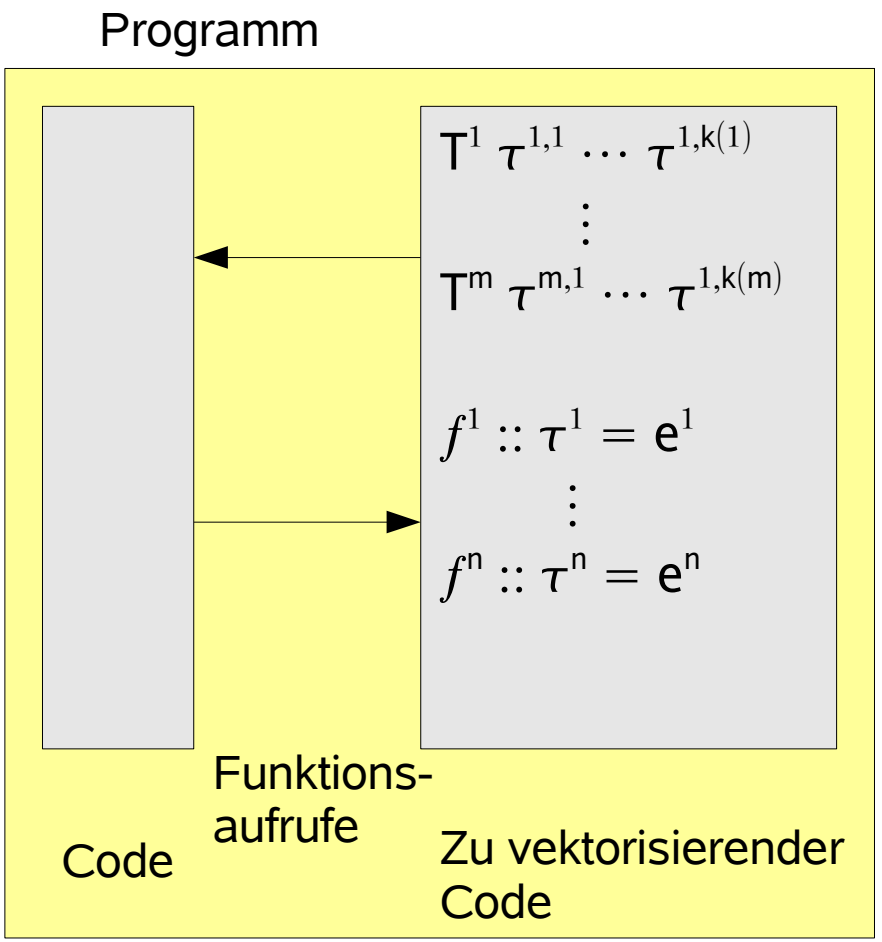
Werden sehen: Vektorisierte Funktionen besitzen eigenen Typ und verwenden nicht den Haskell-Funktionstyp ->

Vektorisierung transformiert die Quellcodes also massiv!

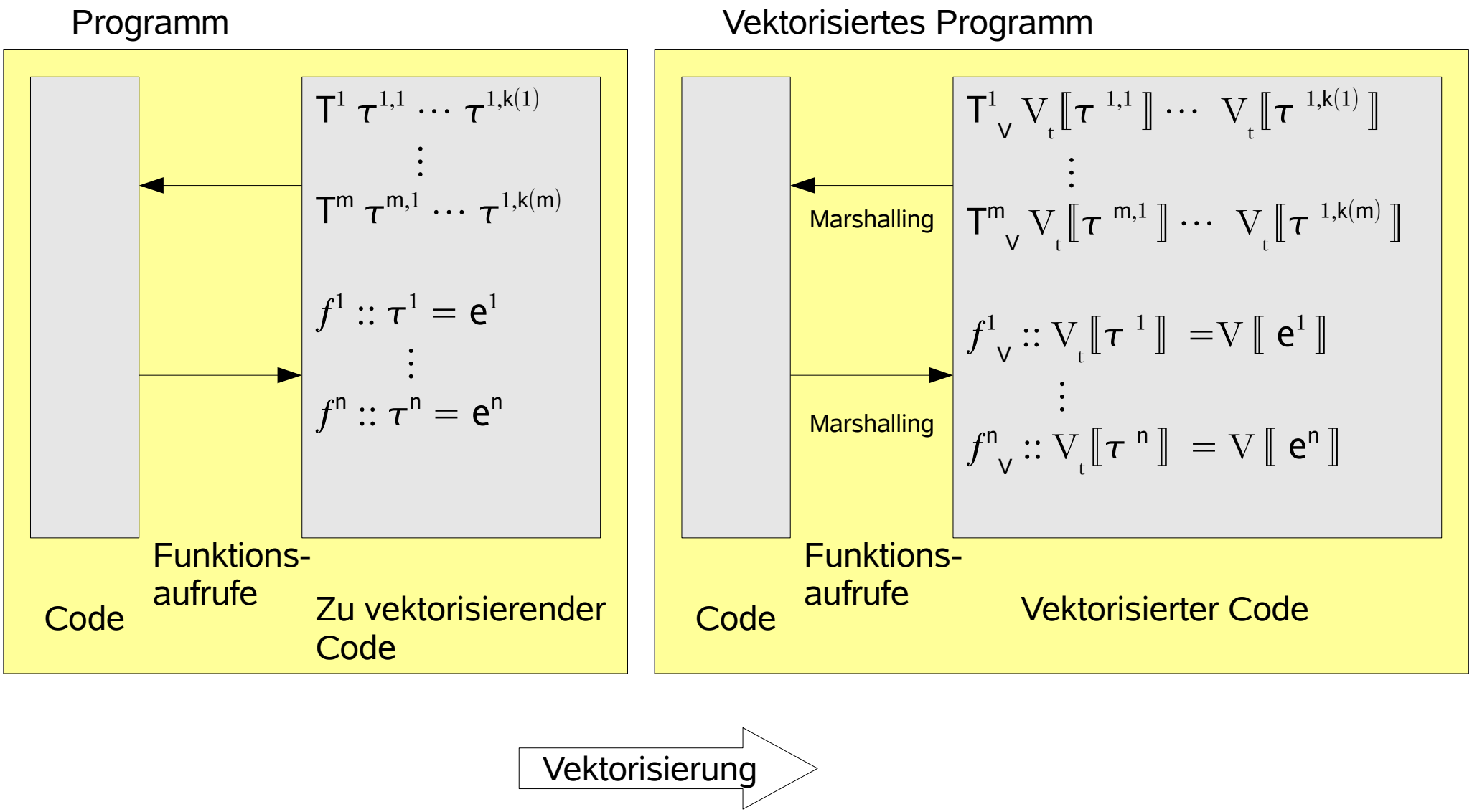
Im allgemeinen: Nur ein Teil des Programms wird vektorisiert (partielle Vektorisierung).

Problem: Marshalling nötig an der Grenze zwischen vektorisiertem und nicht vektorisiertem Code.

Vektorisierung



Vektorisierung



Vektorisierung

Konzentrieren uns zunächst auf verschachtelte Arrays.

Die eingeführten Konzepte erweisen sich später als tragfähig genug um komplexere Haskell-Konstrukte z.B. algebraische Datentypen zu handhaben.

Beispiel:

```
-- f sei eine rechenintensive Funktion
f :: a -> b

-- hier ist Auswertung mit flacher Datenparallelität
-- kein Problem
fl :: [:a:] -> [:b:]
fl = mapP f

-- mit flacher Datenparallelität problematisch
fll :: [:[:a:]:] -> [:[:b:]:]
fll = mapP fl
```

Vektorisierung

Weiteres Problem: Funktionen höherer Ordnung

```
f :: [:Integer:] -> [:Integer -> Integer:]
f xs = mapP (\x y -> x*x + y*y) xs


g :: [:Integer:] -> Integer -> [:Integer:]
g xs i = [:d i | d <- f xs:]
```

Problem:
Der Code in jedem Element der parallelen array comprehension ist verschieden.

Wie soll das mit flacher Datenparallelität ausgewertet werden?

Es muss stets die **selbe** Operation auf alle Arrayelemente angewendet werden.

Vektorisierung

Grundsätzliche Idee anhand eines Beispiels:

Zur Funktion

```
f :: Float -> Float
f x = x*x + 1
```

konstruiere geliftete Funktion

```
f_L :: [:Float:] -> [:Float:]
f_L x = (x *_L x) +_L (replicateP n 1)
        where
          n = lengthP x
```

wobei $*_L$ und $+_L$ Vektoroperationen sind, die parallel ausgeführt werden. Es gilt (semantisch):

```
f_L = mapP f
```

Vektorisierung

Grundsätzliche Idee anhand eines Beispiels:

Zur Funktion

```
f :: Float -> Float
f x = x*x + 1
```

konstruiere geliftete Funktion

```
f_L :: [:Float:] -> [:Float:]
f_L x = (x *_L x) +_L (replicateP n 1)
        where
          n = lengthP x
```

Ersetze:
- Funktionen durch geliftete Funktionen
- Konstanten durch Aufrufe von `replicateP`

wobei $*_L$ und $+_L$ Vektoroperationen sind, die parallel ausgeführt werden. Es gilt (semantisch):

```
f_L = mapP f
```

Vektorisierung

Durch Liften aller Funktionen können wir also stets zu einer Funktion

```
g :: a -> b
```

eine geliftete Funktion

```
g_L :: [:a:] -> [:b:]
```

konstruieren, welche direkt parallel ausgeführt werden kann und die

```
g_L = mapP g
```

erfüllt.
Verwende diese Gleichung als Ersetzungsregel im Quellcode.

Vektorisierung

Problem: Verschachtelte Parallelität

```
f :: a -> b

g :: [:[:a:]:] -> [:[:b:]:]
g xs = mapP (mapP f) xs
```

Hierfür müsste f_{LL} konstruiert werden.

Die Tiefe dieser „Verschachtelung“ ist nicht statisch beschränkt, also zur Compilezeit unbekannt.

Idee (Blelloch):

```
f_LL :: [:[:a:]:] -> [:[:b:]:]
f_LL xss = unconcatP xss (f_L (concatP xss))
```

Werden sehen: `concatP` und `unconcatP` in konstanter Zeit möglich.

Vektorisierung

Klar: Müssen Repräsentation paralleler Arrays geeignet wählen!

Verschachtelte Arrays müssen flach mit Zusatzinformationen über Struktur gespeichert werden.

Repräsentation des Arrays wird vom Typ seiner Elemente abhängen!

Um die nötige Flexibilität zu erhalten benötigen wir:

- Repräsentation der parallelen Arrays -> Assoziierte Datentypen

- Funktionen höherer Ordnung -> Closures

Repräsentation von Arrays aus DPH

Der Vektorisierer ersetzt ein paralleles Array `[:a:]` durch einen Typ `PA a`

Die Repräsentation der Arrays `PA a` ist abhängig vom Typ a z.B.

`PA Int`
repräsentiert durch einen Block von 32-Bit Integer-Werten

`PA Double`
repräsentiert durch Block von 64-Bit-Double-Werten

Problem: Polymorphe Funktionen wie

```
lengthPA :: PA a -> Int
```

sind nicht mehr möglich, da für jede Repräsentation anders zu implementieren.

Repräsentation von Arrays aus DPH

Lösung: Assoziierte Datentypen

```
class PAElem a where
 data PA a                  -- !ASSOZIIERTER DATENTYP!
 indexPA     :: PA a -> Int -> a
 lengthPA    :: PA a -> Int
 replicatePA :: Int -> a -> PA a
-- ... und weitere polymorphe Operationen
-- für parallele Arrays
```

Repräsentation von Arrays aus DPH

Lösung: Assoziierte Datentypen

```
class PAElem a where
 data PA a                    -- !ASSOZIIERTER DATENTYP!
 indexPA     :: PA a -> Int -> a
 lengthPA    :: PA a -> Int
 replicatePA :: Int -> a -> PA a
-- ... und weitere polymorphe Operationen
-- für parallele Arrays
```

Instanz für jeden Typ der Element paralleler Arrays sein kann:

```
instance PAElem Int where
 data PA Int = AInt ByteArray
 indexPA (AInt ba) i = indexIntArray ba i
 lengthPA (AInt ba) = lengthIntArray ba
 replicatePA n i = Aint (replicateIntArray n i)
-- ...
```

(alle primitiven Typen analog hierzu)

Repräsentation von Arrays aus DPH

Lösung: Assoziierte Datentypen

```
class PAElem a where
 data PA a                    -- !ASSOZIIERTER DATENTYP!
 indexPA     :: PA a -> Int -> a
 lengthPA    :: PA a -> Int
 replicatePA :: Int -> a ->
-- ... und weitere polymo
-- für parallele Arrays
```

Primitiver Datentyp ByteArray mit Operationen vom GHC bereitgestellt

Instanz für jeden Typ der Element paral Arrays sein kann:

```
instance PAElem Int where
 data PA Int = AInt ByteArray
 indexPA (AInt ba) i = indexIntArray ba i
 lengthPA (AInt ba) = lengthIntArray ba
 replicatePA n i = Aint (replicateIntArray n i)
-- ...
```

(alle primitiven Typen analog hierzu)

Repräsentation von Arrays aus DPH

Arrays von Tupeln: `PA (a, b)`

```
instance (PAElem a, PAElem b) => PAElem (a, b) where
 data PA (a, b) = ATup Int (PA a) (PA b)
 indexPA (ATup _ arr1 arr2) i
          = (indexPA arr1 i, indexPA arr2 i)
 lengthPA (Atup n _ _) = n
-- ...
```

Repräsentation von Arrays aus DPH

Arrays von Tupeln: `PA (a, b)`

```
instance (PAElem a, PAElem b) => PAElem (a, b) where
 data PA (a, b) = ATup Int (PA a) (PA b)
 indexPA (ATup _ arr1 arr2) i
          = (indexPA arr1 i, indexPA arr2 i)
 lengthPA (Atup n _ _) = n
-- ...
```

Damit `zipPA` und `unzipPA` in konstanter Zeit:

```
zipPA :: PAElem a => PA a -> PA b -> PA (a, b)
zipPA as bs = ATup (lengthPA as) as bs

unzipPA :: PA (a, b) -> (PA a, PA b)
unzipPA (ATup _ as bs) = (as, bs)
```

Repräsentation von Arrays aus DPH

(!) Verschachtelte Arrays: `PA (PA a)`

Repräsentiert durch:
- flaches Array vom Typ `PA a` das alle Daten enthält

- einen Segmentdeskriptor des Typs `PA (Int, Int)` der die Anfangsposition und Länge der inneren Arrays festhält

```
instance PAElem a => PAElem (PA a) where
 data PA (PA a) = AArr (PA a) (PA (Int, Int))
 indexPA (AArr arr segd) i
          = slicePA arr (indexPA segd i)
 lengthPA (AArr _ segd) = lengthPA segd
-- ...
```

(wobei man `slicePA` in konstanter Zeit erreichen kann)

Repräsentation von Arrays aus DPH

Beispiele zu verschachtelten Arrays:

```
[:[:1,2,3:], [:4,5:], [::], [:6,7:]:] :: PA (PA Int)
```

Wird repräsentiert durch:

```
AArr [#1,2,3,4,5,6,7#]                    -- Daten
     (ATup [#0,3,5,5#] [#3,2,0,2#]) -- Segmentdeskriptor
```

Wobei `[#..#]` ein Literal für ein `ByteArray` ist.

Repräsentation von Arrays aus DPH

Beispiele zu verschachtelten Arrays:

```
[:[:1,2,3:], [:4,5:], [::], [:6,7:]:] :: PA (PA Int)
```

Wird repräsentiert durch:

```
AArr [#1,2,3,4,5,6,7#]                    -- Daten
     (Atup [#0,3,5,5#] [#3,2,0,2#]) -- Segmentdeskriptor
```

Wobei `[#..#]` ein Literal für ein `ByteArray` ist.

```
[:[:(0,15),(2,9),(3,20):], [::], [:(3,46):]:]
                                  :: PA (PA (Int, Int))
```

Wird repräsentiert durch:

```
AArr (Atup [#0,2,3,3#] [#15,9,20,46#]) -- Daten
     (Atup [#0,3,3#] [#3,0,1#])      -- Segmentdeskriptor
```

Repräsentation von Arrays aus DPH

(!) `concatPA` und `unconcatPA` in konstanter Zeit:

```
concatPA :: PA (PA a) -> PA a
concatPA (AArr data _) = data
```

„Extraktion“ des flachen Arrays mit dem Inhalt

```
unconcatPA :: PA (PA a) -> PA b -> PA (PA b)
unconcatPA (AArr _ segd) data = AArr data segd
```

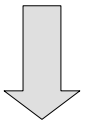
Zusammensetzen des flachen Datenarrays mit dem Segmentdeskriptor

Repräsentation von Funktionen

Haben: Zu jeder Funktion g geliftete Funktion g_L die parallel verarbeitet wird.

Problem: Funktionen höherer Ordnung

```
f :: (Int -> Bool) -> (Bool, [:Bool:])
f g = (g 2, mapP g [:1,2,3:])
```



```
f :: (Int -> Bool) -> (Bool, [:Bool:])
f g = (g 2, g_L [:1,2,3:])
```

Beim Compilieren: Unklar welches g

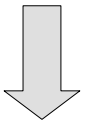
Wenn f aufgerufen wird muss ihm sowohl g als auch g_L übergeben werden!

Repräsentation von Funktionen

Haben: Zu jeder Funktion g geliftete Funktion g_L die parallel verarbeitet wird.

Problem: Funktionen höherer Ordnung

```
f :: (Int -> Bool) -> (Bool, [:Bool:])
f g = (g 2, mapP g [:1,2,3:])
```



```
f :: (Int -> Bool) -> (Bool, [:Bool:])
f g = (g 2, g_L [:1,2,3:])
```

Beim Compilieren: Unklar welches

Wenn f aufgerufen wird muss i geben werden!

Brauchen:
Neuen Funktionstyp,
der g und g_L enthält

Repräsentation von Funktionen

Weiteres Problem: Arrays von Funktionen

```
dist :: Float -> Float -> Float
dist x y = sqrt(x*x + y*y)

distance :: [:Float:] -> [:Float -> Float:]
distance xs = mapP dist xs


-- Wie soll x repräsentiert werden?
x :: [:Float -> Float:]
x = distance [:1.0, 4.5, 6.7:]
```

Funktionen in x unterscheiden sich.

Können nicht parallel auswerten, da dafür der Code **gleich** sein müsste!

Repräsentation von Funktionen

Weiteres Problem: Arrays vo

```
dist :: Float -> Flo
dist x y = sqrt(x*x +

distance :: [:Float:] -> [:F
distance xs = mapP dist xs


-- Wie soll x repräsentiert werden?
x :: [:Float -> Float:]
x = distance [:1.0, 4.5, 6.7:]
```

Idee:
Funktionen sind identisch.
Unterschiedlich ist das Argument an die freie Variable x.

Funktionen in x unterscheiden sich.

Können nicht parallel auswerten, da dafür der Code **gleich** sein müsste!

Repräsentation von Funktionen

Also: Eine vektorisierte Funktion besteht aus:

- ursprünglicher Funktion
- gelifteter Funktion
- Werte für freie Variablen -> Closures

```
-- einfaches Closure in Haskell
f :: Float -> (Float -> Float)
f z = (\ x y -> x + y) z
```

Typ: Vektorisierte Funktion

Führen Datentyp für vektorisierte Funktion ein.

Vektorisierer ersetzt Haskell-Funktionen `a -> b` durch vektorisierte Funktionen `a :-> b`

```
data (a :-> b) = forall e. PAElem e =>
                 Clo { env  :: e
                     , clo_s :: e -> a -> b
                     , clo_l :: PA e -> PA a -> PA b
                     }
```

Er muss selbst den Code für clo_s und clo_l erzeugen.

Typ: Vektorisierte Funktion

Zur Applikation vektorisierter Funktionen muss ein Applikationsoperator bereitgestellt werden:

```
($:) :: (a :-> b) -> a -> b
($:) (Clo env fs fl) = fs env
```

Analog der Applikationsoperator für die geliftete Funktionen:

```
($:L) :: a :-> b -> PA a -> PA b
($:L) (Clo env fs fl) arr
        = fl (replicatePA (lengthPA arr) env) arr
```

Arrays vektorisierter Funktion

Damit:

```
instance PAElem (a :-> b) where
 data PA (a :-> b) = forall e. PAElem e =>
                     AClo { aenv :: PA e
                          , aclo_S :: e -> a -> b
                          , aclo_L :: PA e -> PA a -> PA b
                          }

 indexPA (AClo env f_S f_L) i
         = Clo (indexPA env i) f_S f_L

 lengthPA (AClo env _ _)
         = lengthPA env

 replicatePA i (Clo env f_S f_L)
         = AClo (replicatePA i env) f_S f_L
-- ...
```

Arrays vektorisierter Funktion

Damit:

```
instance PAElem (a :-> b) where
 data PA (a :-> b) = forall e. PAElem e =>
                     AClo { aenv :: PA e
                          aclo_S :: e -> a -> b
                          aclo_L :: PA e -> PA a -> PA b
                          }

 indexPA (AClo env f_S f_L) i = Clo (f_S env i) f_S f_L

 lengthPA (AClo env f_S f_L) = f_S f_L

-- ...
```

Man sieht: Array besitzt
- Eintrag für **eine** Funktion (mit Liftung)
- Ein Array von Umgebungsvariablen

Arrays vektorisierter Funktion

Achtung: Ein Array der Form

```
[:sin, cos, tan, exp:] :: Floating a => [:a -> a:]
```

ist nicht direkt repräsentierbar!

Beispiel einer vektorisierten Funktion

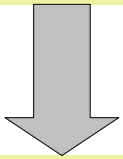
```
inc :: Float -> Float
inc = (\x -> x + 1)
```

Nachtrag: Unit-Array (für Closure ohne Umgebungsvariablen)

```
instance PAElem () where
 data PA () = AUnit Int
 indexPA (AUnit _) i = ()
 lengthPA (AUnit n) = n
-- ...
```

Beispiel einer vektorisierten Funktion

```
inc :: Float -> Float
inc = (\x -> x + 1)
```



Vektorisierer

```
inc_v :: Float :-> Float
inc_v = Clo () inc_s inc_L

inc_s :: () -> Float -> Float
inc_s = (\e x -> case e of
                   () -> (+)_v $: x $: 1
        )

inc_L :: PA () -> PA Float -> PA Float
inc_L = (\e x -> case e of
                   AUnit n -> (+)_v $:_L x $:_L (replicatePA n 1)
        )
```

Implementierung von mapP

Vektorisierung:

Verschachtelte Datenparallelität -> Flache Datenparallelität

Kern ist Implementierung der von DPH bereitgestellten Funktionen.

Haben nun alles zurechtgelegt um diese Implementierung am Beispiel von `mapP` durchzuführen.

(Hier findet sich Blellochs Idee wieder)

Implementierung von mapP

```
mapP :: (a->b) -> [:a:] -> [:b:]
```

Vektorisierte Funktion (von Hand implementiert!):

```
mapP_V :: (a :-> b) :-> PA a :-> PA b
mapP_V = Clo () mapP_1 mapP_2

mapP_1 :: () -> (a :-> b) -> (PA a :-> PA b)
mapP_1 _ f = Clo f mapP_S mapP_L

mapP_2 :: PA () -> PA (a :-> b) -> PA (PA a :-> PA b)
mapP_2 _ fs = AClo fs mapP_S mapP_L

mapP_S :: (a :-> b) -> PA a -> PA b
mapP_S (Clo env _ fl) xss
    = fl (replicatePA (lengthPA xss) env) xss

mapP_L :: PA (a :-> b) -> PA (PA a) -> PA (PA b)
mapP_L (AClo env _ fl) xss
    = unconcatPA xss (fl (expandPA xss env) (concatPA xss))
```

Implementierung von mapP

```
mapP :: (a->b) -> [:a:] -> [:b:]
```

Vektorisierte Funktion (von Hand implementiert!):

```
mapP_v :: (a :-> b) :-> (PA a :-> PA b)
mapP_v = Clo () mapP_1 mapP_2
```

Zunächst ist für diesen vektorisierten Funktionspfeil das Objekt zu definieren

```
mapP_1 :: () -> (a :-> b) -> (PA a :-> PA b)
mapP_2 :: PA () -> PA (a :-> b) -> PA (PA a :-> PA b)
```

Implementierung von mapP

```
mapP :: (a->b) -> [:a:] -> [:b:]
```

Vektorisierte Funktion (von Hand implementiert!):

```
mapP_V :: (a :-> b) :-> PA a :-> PA b
mapP_V = Clo () mapP_1 mapP_2

mapP_1 :: () -> (a :-> b) -> (PA a :-> PA b)
mapP_1 _ f = Clo f mapP_S mapP_L

mapP_2 :: PA () -> PA (a :-> b) -> PA (PA a :-> PA b)
mapP_2 _ fs = AClo fs mapP_S mapP_L
```

Implementierung von mapP

```
mapP :: (a->b) -> [:a:] -> [:b:]
```

Vektorisierte Funktion (von Hand implementiert!):

```
mapP_V :: (a :-> b) :-> PA a :-> PA b
mapP_V = Clo () mapP_1 mapP_2

mapP_1 :: () -> (a :-> b) -> (PA a :-> PA b)
mapP_1 _ f = Clo f mapP_S mapP_L

mapP_2 :: PA () -> PA (a :-> b) -> PA (PA a :-> PA b)
mapP_2 _ fs = AClo fs mapP_S mapP_L

mapP_S :: (a :-> b) -> PA a -> PA b
mapP_S (Clo env _ fl) xss
    = fl (replicatePA (lengthPA xss) env) xss

mapP_L :: PA (a :-> b) -> PA (PA a) -> PA (PA b)
mapP_L (AClo env _ fl) xss
    = unconcatPA xss (fl (expandPA xss env) (concatPA xss))
```

Implementierung von DPH

Noch offen: (Vektorisierung)

- Konkrete Definition der Vektorisierungstransformation für Typen und Funktionen
- Arrays algebraischer Datentypen \ algebraische Datentypen allgemein
- Laziness
- Partielle Vektorisierung (und damit verbundenes Marshalling)
- Viele subtile Probleme (u.a. bei der Implementierung im GHC)
- .....

Implementierung von DPH

Noch offen: DPH

- Fusion – Programmtransformation zur Vereinigung von Vektoroperationen (Elimination von Synchronisierungspunkten)
- Gang Parallelism: Wie wird der datenparallele Code auf Threads abgebildet?
- Optimierung des Programmcodes durch GHC
- .....

Implementierung von DPH

Noch offen: DPH

- Fusion – Programmtransformation zur Vereinigung von Vektoroperationen (Elimination von Synchronisierungspunkten)
- Gang Parallelism: Wie wird der datenparallele Code auf Threads abgebildet?
- Optimierung des Programmcodes durch GHC
- .....

Ist in funktionalen Sprachen **viel** einfacher als in imperativen – funktioniert für DPH sehr gut.

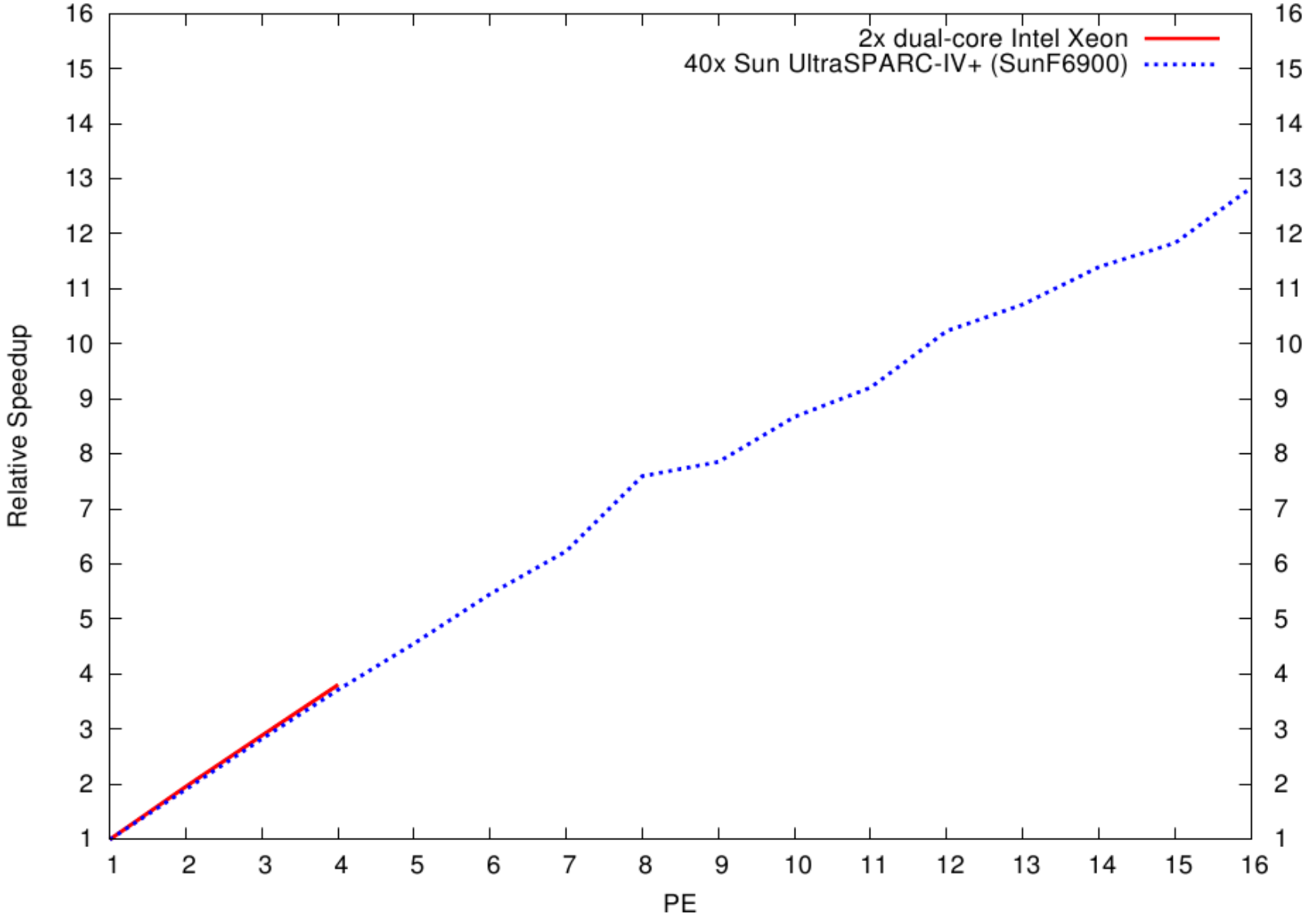
Zusammenfassung

Data Parallel Haskell:

Verschachtelte Datenparallelität:

- Einfach und flexibel einsetzbar, vielversprechender Ansatz!
- **Aber**: Schwierig zu implementieren

Benchmark



Speedup bei Multiplikation einer 10000x10000 dünn besetzten Matrix mit ungefähr 1 Million Double-Einträgen ungleich 0