Parallele und Funktionale Programmierung

Autor: Julian Kotzur

Sinnlose Wissensachen

Grundlagen
→ Lastverteilung: große Mengen von Anfragen auf mehrere parallel arbeitende Systeme verteilen
→ Begrenzte Resourcen um Dos-Attacken zu verhindern
→ Unterschiedliche JVMs können indeterministisches Verhalten verursachen
→ bei kleinen Problemen ist Parallelität unnötig
→ Thread-Erzeugung kostet Zeit und Resourcen
→ Granularität: feingranulare Aktivitätsfäden schlecht für Parallelisierung. Deshalb so grobgranular halten wie möglich

Speedup
→ $Speedup = \frac{Laufzeit\ Sequentiell}{Laufzeit\ Parallel}$
→ Idealer Speedup: Speedup = Prozessorzahl
→ Negative Aspekte für Parallelisierung: Sequentieller Anteil, Verwaltungskosten, IO-Wartezeiten
→ Effizienz: Speedup / Prozessoranzahl
→ Gesetz von Amdahl:
⇒ Maximaler Speedup nie erreichbar, jedes Parallele Programm sequentiellen Teilcode enthält

Lemma von Brent
→ Lemma von Brent: Wenn es einen p-verteilten statischen Algorithmus gibt, der in O(t) liegt, und insgesamt s Berechnungen/Schritte ausführt, dann gibt es einen s/ t-verteilten Algorithmus mit gleichen Aufwand

Petri-Netze

Stellen:
→ Werden als Kreise dargestellt
→ Innerhalb der Kreise sind Zahlen bzw. Tokens
→ Kapazität ist hochgestellte Zahl ⇒ Max. Tokenanzahl

Transitionen:
→ Darstellung als Striche bzw. Rechtecke

Pfeile:
→ Verbinden genau eine Transition mit einer Stellen
→ Pfeile können ganzzahlig bewertet werden
→ Keine Bewertung impliziert Bewertung 1

Schaltregeln:
→ Transition kann feuern, wenn jede Eingangsstelle mind. so viele Tokens hat, wie das Gewicht des Pfeiles angibt.
→ Zudem, wenn in jeder Ausgangsstelle die Kapazität nicht überschritten wird.
→ Beim Schalten werden in den Eingangsstellen die Tokens entsprechend dem Pfeilgewicht entfernt und ebenso den Ausgangsstellen hinzugefügt.
→ Das Schalten benötigt keine Zeit
→ Petrinetz feuert, wenn Transition feuert

Eigenschaften

→ Beschränkt: Keine Stelle kann unendlich Token enthalten
→ Lebendig: Transition kann unendlich oft feuern. Wenn das für alle Transitionen gilt, ist das Petri-Netz lebendig. Hilfreich hierbei ist zu schauen, ob sich Marken vernichten/erzeugen lassen. Am Einfachsten Lebendig zu widerlegen ist es, eine Schaltreihenfolge zu finden, bei der alle Marken verschwinden. Zudem kann man einen Endloszyklus finden, der nicht alle Transitionen beinhaltet, und aus dem man nicht rauskommt, um Lebendigkeit zu widerlegen

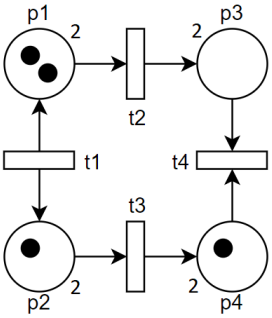
Erreichbarkeitsgraph

→ Entscheidungsbaum, bestehend aus Belegungen
→ Aus Belegung gehen Äste, abhängig von den Transitionen die feuer können, aus
→ Alle möglichen Belegungen sind enthalten
→ Es schaltet in jedem Zustand nur eine Transition

Petrie Netze und Matrizen

→ Transitionsinvariante I: Matrix * I = 0
→ Stelleninvariante S: S * Matrix = 0
→ Pro Stelle eine Zeile
→ Pro Transition eine Spalte
→ negative Zahl: Stelle → Transition
→ positive Zahl: Stelle ← Transition
→ Anfangsbelegung: Vektor mit Tokens in den Stellen
→ Schaltvektor: Anzahl der Feuer einer Transition
⇒ Multiplikation mit Matrix ergibt Belegungsvektor
⇒ Enthält als Zeilen die Transitionen mit Feueranzahl
→ Endbelegung = Matrix * Schaltvektor + Startbelegung

Beispiel zum selbst bearbeiten:



Checkfragen für die Klausur:

→ Threads erstellen und starten? Runnable?
→ Unterschied "implements Runnable"/"extends Thread"?
→ ExecutorService? Welche Methoden? Was bekommt er übergeben? Implementierung? Vorteile? Callalbe? Future?
→ Arten von Wettlaufsituationen?
→ Synchronisation: join, Future ,synchronized
→ Threadsicherheit: Kapselung, Faustregel Verwendung synchronized
→ volatile? Atomic?
→ Verklemmungen? Lösungsmöglichkeiten, Arten, Vermeidungsstrategien
→ Speedup + Berechnung? Lastbalance?
→ Collection-Klassen? Map-Reduce? CyclicBarrier? Giftpille?, work stealing? geometrische Dekomposition?
→ Maximumssuche? Parallele Summenbildung? Turnieralgorithmus?
→ Petri - Netz : lebendig ? beschränkt ? Invarianten ? Ergebnisvektor ? Matrix ? Ereignisgraph ? Deadlock ? LifeLock ?
→ Scala : Listen? Nil? NUll? null? Seiteneffekte? Pattern Matching ? Mustervergleich? Bibiliotheksfunktionen, Binäre Bäume? Parallel?

Erzeugung von Parllelität

Threads-Objekt:

```
import java.lang.Thread;
public class Threads extends Thread{
 public void run(){
  System.out.println("Hello World");
}}
Thread thread1 = new Threads();
thread1.start(); // starten
try{ thread1.sleep( <Zeit> ); // warten
thread1.join(); //Beenden
}catch(Exception e){} } //Exceptionbehandlung
// mehrere Threads:
Thread[] ts = new Thread[<Anzahl n>];
for (int i=0; i<n;i++) ts[i]= new Threads(i);
for (Thread t : ts) t.start();// starten
for (Thread t : ts) t.join(); // Beenden
```

Runnable-Objekt:

```
import java.lang.Runnable;
public class Runner implements Runnable{
 public void run(){
  System.out.println("Hello World");
}}
// Einen Runnable berechnen:
new Thread(new Runner()).start();
// mehrere Runnables;
Thread[] ts = new Thread[<Anzahl n>];
for(int i=0;i<n;i++){
ts[i] = new Thread(new Runner());
ts[i].start();}
for(Thread t : ts){try{ t.join();
}catch(Exception e) {} }
```

ExecutorService:

```
import java.util.concurrent;
public class Execut implements Runnable{
public void run(){
System.out.println("Hello World");
}}
public static void main(String args[]){
 ExecutorService e;
 e = Executors.newFixedThreadPool(<Anzahl>);
 e.execute(new Execut());
 for (int i = 0; i < <Anzahl>; i++) {
 e.execute(new MyRunnable(i)); }
 e.shutdown(); }
```

Bemerkungungen:
→ Thread-Lösungen auf Runnable-Lösungen zurückführbar
→ Executer benötigt Klassen, die Runnable implementieren
→ join() sowie ExecutorService.awaitTermination() benötigen InterruptedException-Behandlung.

Callables

→ Haben im Gegensatz zu Runable einen Rückgabewert

```
import java.util.concurrent.Callable;
class Starter implements Callable<Integer>{
public Integer call(){ return 1; } }
Callable<Integer> c = new Starter();
ExecutorService e {bla...};
Future<Integer> result=e.submit(c); //starten
int ausgabe = result.get(); // Wert umlagern
```

Probleme die Auftreten:

Sichtbarkeitsprobleme

→ Wird ein Wert in eine Variable geschrieben, dann kann man ohne Sichtbarkeitssynchronisation nicht annehmen, dass andere Aktivitätsfäden den neuen Wert jemals sehen
→ Ursachen: Umordnungsprobleme und Datenlogistik intern im Rechner(Register, Caches)
→ Lösung: volatile

Umordnungsprobleme

→ Keine Datenabhängigkeit in Thread-Methoden
⇒ Zuweisungen müssen nicht sequentiell passieren

Wettlaufsituationen

→ Tritt auf, wenn Thread "gleichzeitig" ausgeführt werden
→ nicht deterministisches Verhalten
→ Abarbeitung der Threads in unterschiedlicher Reihenfolge
→ Tritt gerne bei Variablen mit mehrfachem Zugriff auf
→ Lösung: Atomare Aktionen, Synchronisation

Deadlock

→ Auch Verklemmung genannt
→ Gutes Beispiel: Philosophenproblem
→ Erfüllte Bedingungen:
⇒ Gegenseitiger Ausschluss (Löffel kann nicht von mehreren Philosophen gehalten werden)
⇒ Kein Entzug (Philosophen prügeln sich nicht um Löffel)
⇒ Iterative Anforderung (Es wird ein weiterer Löffel zur exklusiven Nutzung angefordert)
⇒ Lösung: Atomar zwei Löffel nehmen
→ Lösung: Löffel nummerieren, kleiner Zahl privorisieren, Locks nach gewisser Zeit freigeben
→ Hilfsmittel: atomic anstatt synchronized
→ Verhungern: Spezielfall, hier konkurrieren zwei Threads und einer wird am weiterkommen gehindert.
Beispielcode:

```
private final Object left = new Object();
private final Object right = new Object();
public void leftRight() {
 synchronized(left) {
  synchronized(right) { // eat } } }
public void rightLeft() {
 synchronized(right) {
  synchronized(left) { // eat } } }
```

Hilfreiche Werkzeuge:

Arbeit gleichmäßig verteilen:

```
public class Threads extends Thread {
 int id;
 public MyThread(int i) { id = i; }
 public void run() {
  if (number == (id % <AnzahlThreads>)) { }
} }
```

Synchronized

→ Wann synchronized verwenden:
⇒ Parallel lesender und schreibender Zugriff auf eine Klassenvariable sollte synchronisiert werden
→ Erzeugung:
⇒ Alle Threads joinen
⇒ Auf Dummie-Objekt syhcronisieren:

```
static Object dummie = new Object();
synchronized(dummie){ }
```

→ Auch auf eine Klasse möglich und sinnvoll: Name.class
→ Auch auf Klassen "außerhalb" möglich
→ synchronized(this) ist meist doof
→ synchronized überwacht einen CodeBlock und lässt nur jeweils einen Thread auf einmal daran arbeiten.
→ Warten mehrere Threads bei einem synchronized, wird "zufällig" der nächste ausgewählt
→ Hat ein Aktivitätsfaden bereits eine Marke, so kann er weitere synchronized-Methoden bzw. Blöcke betreten, die von der selben Marke geschützt werden. Die Marke wird erst beim Verlassen der zuerst aufgerufenen synchronized-Methode freigegeben
→ Hat eine Klasse synchronized-Methoden und normale Methoden, dann können die normalen Methoden beliebig nebenläufig ausgeführt werden
→ Nachteile:
⇒ Es ist immer nur genau 1 Aktivitätsfaden erlaubt
⇒ Laufzeitzunahme durch wartende Threads
⇒ Das Anfordern und das Freigeben der Marke müssen in der selben Methode erfolgen

korrekte Klasse

→ die Klasseninvariante gilt ab der Erzeugung
→ beliebiger Methodenaufruf problemlos möglich
→ Jede Methode muss das von ihr erwartete Verhalten haben
→ Verifikation mittels wp-Kalkül

thread-sichere Klasse

→ Klasse ist korrekt
→ jede Methodenaufrufenkombination zerstört nicht die Klasseninvariante bzw. das Verhalten der Methoden
→ ist eine Klasse thread-sicher, sorgt sie also selbst für eine geeingete Datensynchronisation
→ Verifikation mittels Petri-Netze
→ Wettlaufsituationen weiterhin möglich

Thread-Sichere Konstruktoren

→ Konstrukoren können nicht synchronized werden
→ Instanzierungsprobleme bei Objekterzeugung möglich
→ Möglichkeiten der Umsetzung:
⇒ Objektreferenz als volatile-Variable
⇒ Objektreferenz als synchronizedgeschütze Instanzvariable
⇒ Statische Inizialisierung beim Laden der Klasse

volatile

→ schwächer als synchronized
→ Modifikator vor Variablen
→ Variable steht nur im Hauptspeicher
→ Wird bei Variablen, die für Abbruchbedingungn zuständig sind, verwendet
→ Trotz volatile können Wettlaufsituationen auftreten
→ Beispiel:

```
public class VolaTest {
 private volatile static int MY_INT = 0;
 public static void main(String[] args) {
  new Listener().start();
  new Maker().start();
 }
 static class Listener extends Thread {
  public void run() {
   int local_value = MY_INT;
   while (local_value < 5) {
    if (local_value != MY_INT) {
     System.out.println("change" + MY_INT);
     local_value = MY_INT;
 }}}}
 static class Maker extends Thread {
  public void run() {
   int local_value = MY_INT;
   while (MY_INT < 5) {
    System.out.println("increment"+ MY_INT);
    MY_INT = ++local_value;
    try {Thread.sleep(500);}
    catch (InterruptedException e) { }
}}}}
```

Atomic-Klassen

→ mächtiger als volatile
→ get() und set() Zugriffe wie bei volatile
→ Vorteil: Es gibt weitere Methoden:
⇒ decrementAndGet();
⇒ incrementAndGet();
⇒ addAndGet(int delta);
⇒ getAndDecrement();
⇒ getAndIncrement();
⇒ getAndAdd(int delta)
→ thread-sicher, da Methoden atomar

LinkedBlockingQueue

→ Vorteil: Keine Fehler bei zu vielen/wenigen Elementen
→ FIFO Prinzip der Eingabe/Entnahme
→ poll() : Gibt und löscht Kopfelement bzw. null
→ take() : Wartet auf Kopfelement und macht dann poll()
→ offer(): Einfügen falls möglich, sonst false
→ put() : Wartet bis einfügen möglich

CyclicBarrier

→ Bei Inizialisierung wird Anzahl Threads übergeben
→ barrier.await() wartet bis alle Threads angekommen
→ Nützlich für Sichtbarkeitssynchronisation

ReentrantLock

→ ähnlich wie synchronized
→ lässt nur einen Thread ab dem lock() weiterarbeiten
→ nächster Thread kann erst nach unlock() arbeiten

Task-parallele Vorgehensweise

→ Gliederung des Problems in Arbeitspakete

Lineare Organisationsform

Unabhängige Arbeitspaktete:

client and server:
→ Anfragen werden an einen Server gestellt
→ Der Server arbeitet diese ab oder:
→ Der Server erstellt Threads, welche abarbeiten
→ Nachteile:
⇒ Zu viele Anfragen: Server hat keinen Speicher mehr
⇒ Zu viele Arbeitspakete, Threads, Erzeugung ist teuer

master and worker:
→ work stealing:
⇒ Threads erzeugen Arbeitspaktete
⇒ Es werden so auch Arbeitspakete erzeugt
⇒ Diese Arbeitspakete werden in einer Queue gespeichert
⇒ Threads nehmen sich nach Beendigung ihrer Arbeit neue Arbeit aus der Queue

Abhängige Arbeitspakete:

Producer-Comsumer:
⇒ seiteneffektfreie Funktionsauswertung wichtig
⇒ Producer: Stellt Arbeitspakete für Consumer bereit
⇒ Consumer: Nimmt Arbeitspakete arbeitet diese ab
⇒ Java stellt dafür LinkedBlockingQueue zur Verfügung
⇒ Poison-Pill: Beendet Consumer. Wird von Producer erzeugt und von Consumer weitergegeben

Fließbandarbeit:
⇒ Arbeitspakete werden sequentiell bearbeitet
⇒ Parallelität tritt erst bei mehreren Arbeitspaketen ein
⇒ Fließbandstufen sollten ähnlichen Umfang haben
⇒ Fließbandstufen sollten parallel arbeiten
→ Aufgabe: n Stufen mit $x_1, ..., x_n$ Zeit und m Paketen:
⇒ Sequentiell: m*(summe x_n)
⇒ Parallel: (Summe x_i) + (m-1)*(größtes x_i)
⇒ Speedup: Sequentiell / Parallel

Rekursive Organisationsform

→ Fibonacci:
⇒ Jeder Rekursive Aufruf erzeugt einen Thread
⇒ Extrem viel Resourcen für Threads benötigt
⇒ Langsamer als Sequentielle Variante
⇒ Daher Parallelität/Threadanzahl begrenzen
⇒ Zusätzlich umstieg auf Sequentielle Lösung ab einer gewissen tiefe möglich

Beispiel Monte Carlo

→ Chef und Arbeiter Prinzip
→ Wird z.b. zur Berechnung von Pi verwendet
→ Interessant ist die Arbeitsverteilung:

```
// Das ist hier stark vereinfacht:
int pts = 42; // Anzahl der Punkte
int mySum = 0; // Anzahl der "Treffer"
for (int i = id; i < pts; i += numThreads) {
 double x = rnd.nextDouble();
 double y = rnd.nextDouble();
 if (x*x + y*y <= 1.0) mySum++;
}
```

Datenparallele Vorgehensweise

→ Gliederung des Problems in Datenstrukturen

Geometrische Dekomposition

→ keine Datenabhängigkeit zwischen einzelnen Werten
→ Funktion f wird auf jedes Datenelement angewendet
→ Funktionsweise:
⇒ Aufteilung der Datenelemente auf mehrere Threads
⇒ Bei mehrfacher Iteration über die Datenelemente, Threads nicht neu erzeugen
⇒ Lösung CyclicBarrier verwenden
⇒ mit await() kann man die Threads synchronisieren
⇒ Threads arbeiten weiter, wenn alle angekommen sind ⇒ Beispiel: GameOfLife

→ Rekursive Datenstrukturen
⇒ Beispiel: Listen, Bäume, Graphen

→ Parallele Maximumssuche:
⇒ Sequentiller Algorithmus ittriert über alle Elemente
⇒ Parallele Methode: Tunier
⇒ Laufzeit: O(1 / log n)
⇒ Anzahl der Threads habliert sich pro Stufe
⇒ Vorgehen: Datenstruktur auf Threads aufteilen. Alle Threads berechnen Maximum. Danach werden die Ergebnisse auf Threads aufgeteilt und wieder das maximum berechnet. Das wird solange gemacht, bis nur noch ein Thread übrig ist

→ Parallele Summenbildung:
⇒ Statt der Max-Operation Werte der Reihung addieren
⇒ Laufzeit: O(log n)

Map-Reduktion

→ Ähnlich wie Chef/Arbeiter-Verfahren
→ map:
⇒ Funktion: (in-key , in-value) → (out-key , tmp-value)
⇒ Erzeugung von Zwischenergebnissen

→ reduce:
⇒ Funktion: (out-key , tmp-value) → out-value
⇒ Berechnet aus Zwischenergebnis das Endergebnis

→ Beispiel: WordCount:
⇒ map: erstellt 26 Listen und hängt jeweils ein 1er Element pro vorkommenden Buchstaben an
⇒ reduce: summiert die 1er Elemente auf
→ Beispiel Dijksta:
⇒ map: Berechnet für alle Nachfolger Knoten von einem Knoten aus den Weg und speichert diese
⇒ reduce: Wählt aus den von map berechneten Werten das Minimum aus

Wichtige Begriffe:
→ Lokalitätsoptimierung: der map-Arbeiter wird möglichst auf dem Rechner gestartet, der die zugehörige Datei speichert
→ Lastbalance: Zuteilung von map-Arbeitern berücksichtigt die Belastungssituation der Rechner

Scala

Grundlage:
→ Seiteneffektfrei
→ Rückgabewerte einer Funktion sind zeitlos (werttreu)
→ Programmierung durch Funktionen und Anwendungen
→ deklarative Sprache: Ergebnis wird formal beschrieben
→ Lazy Evaluation: Ausdrücke werden erst ausgewertet, wenn sie gebraucht werden (Schlüsselwort lazy val)
Beispielcode zu Fibonacci:

```
object Fibonacci {
 def fibonacci : Int => Long = {
  case 1 => 1
  case 2 => 1
  case n => fibonacci(n-1) + fibonacci(n-2)
}
 def main(args: Array[String]) {
  println(fibonacci(30))
} }
```

→ object deklariert ein Singleton
→ Singleton ist das Klassenäquivalent zu Java

Bedingte Ausdrücke:

→ Zu jedem "if" muss ein "else" gehören

```
def fibonacci : Int => Long = n =>{
 if(n==1 || n == 2) 1
 else fibonacci(n-1) + fibonacci(n-2)
}
```

Tupel

→ Tupel können mehrere Elemente beinhalten
→ Tupelelemente können unterschiedliche Datentypen haben
→ Zugriff auf n-te Komponente: ._n
Beispiel Fibonaccitupel:

```
def fibtupel :(Int , Int)=>(Long , Long)={
 case(n , m) => (fibonacci(n) , fibonacci(m))
}
```

Zeichenketten - Strings

```
val x = "fibonacci" // String value
x.toCharArray() // Umwandlung in CharArray
for (n <- x) print(n) // Arrayiteration
println("fibonacci")
println("fibonacci" : String)
```

Pattern Matching

→ Scala Switch-Case-Äquivalent nur mächtiger

```
val  matched : Any = List(1)
matched match{
 // Unterschiedliche Datentypen
 case m : String => println("String")
 // Listen matchen
 case n::Nil => println("List")
 // Zahlen matchen
 case 1 | 1 | 2 => println("Fibonacci")
 // Default Fall
 case _ => println("nothing")
}
```

Funktion

→ Signatur der Funktion: f: $T_1 => T_2$

```
\\ Allgemeine Definition
def <Name> : <Input> => <Output> =
\\ Definition mit Generischem Typ:
def pair[A]: A => (A, A) = x => (x, x)
\\ Currying: Mehrere Parameterlisten:
def addd: Int => Int => Int => Int =
x => y => z => x + y + z
```

Funktionen höherer Ordnung

→ Liefert oder erwartet eine Funktion als Argument

```
def doppelfib:(Long => Long)=>Int=>Long={
 f => x => f( f(x) )
}
\\ Funktion aufrufen Beispiel:
println(doppelfib(fibonacci) (9))
```

Listen

→ Sequenz von Werten gleichen Typs
→ Any ist die Oberklasse aller Typen in Scala
→ Liste Zerlegen: p::ps (p = Listenkopf , ps = Restliste)
→ Liste an Liste anhängen: p ::: ps

```
def fiblist :(List[Long],Int)=>List[Long]={
 case (Nil , x)       => Nil
 case (n :: Nil , x) => Nil
 case (n::m::q , x)  =>
  if( x == 1) List(n+m):::List(n,m):::q
  else fiblist2(List(n+m,n,m):::q , x-1)
}
List(false, true, false) : List[Boolean]
List(List(1, 1)) : List[List[Int]]
List("Hello", List(false) , 'b') : List[Any]
Listengenerator:
val liste = List.range(3,6) // List(3,4,5)
for(x <-liste)) yield x * x // List(9,16,25)
```

Wichtige Bibliotheksfunktionen

.toList	String zu Liste
.mkString	Char-Array zu String
.sum	Summe aller Elemente einer Liste
.product	Produkt aller Elemente einer Liste
.reverse	Listenindizies invertieren
(n)	Auswahl Element Index n (ohne .)
.take(n)	Liste der ersten n Elemente
.drop(n)	Liste ohne die ersten n Elemente
.length()	Ausgabe der Länge
.map(f(x))	Funktion auf jedem Listenelement
.filter(x=>x)	filtert Elemente aus
.takeWhile(x=>x)	Liste Elemente bis !Prädikat
.dropWhile(x=>x)	filter solange bis !Prädikat
.forall(x=>x)	true, alle Elemente erfüllen Prädikat
.exists(x=>x)	true, ein Element erfüllt Prädikat
.foldRight(x=>x)	Funktionsaufruf von Links nach Rechts
.foldLeft(x=>x)	Funktionsaufruf von Rechts nach Links

Stream

→ eine unendliche Datenstruktur
→ speichert bereits berechnete Elemente
→ Wichtigstes Werkzeug: #::

```
def fibstream:Int=>Stream[Long]=x=>{
 fibonacci(x)  #:: fibstream(x+1)
}
 def streamtoList:Int=>List[Long]=n=>{
 for(x<-List.range(0,n)) yield fibstream(1)(x)
}
println(streamtoList(10))
// List(1, 1, 2, 3, 5, 8, 13, 21, 34, 55)
```

Type Aliasing

→ Eigene Datenstruktur erstellen

```
type <Name> = <Datenstruktur>
\\Beispiel:
type Coordinate = (Int,Int)
```

Typverbund

→ auch Algebraischer Datentyp genannt
→ besteht i.d.R. aus mehreren Datentypen
→ Beispiel

```
abstract class Shape
case class Circle(r: Double) extends Shape
case class Rect(l: Int, w: Int)extends Shape
def square: Int => Shape =
 n => Rect(n, n)
def area: Shape => Double = {
 case Circle(r) => math.Pi * r * r
 case Rect(l, w) => l * w }
```

→ Circle und Rect sind Konstruktoren
→ Dürfen auch rekursiv definiert werden

Parallelisierung

→ scala.collection.parallel.immutable
→ Programmierer muss sich um nichts kümmern
→ Parallelisierung mit .par

```
val seq = 4711::42::List.fill(300000)(666)
val par = seq.par
```

→ Java Synchronisierungskonzepte möglich:
⇒ synchronized
⇒ wait und notify
→ Beispiel

```
def fib: Int => Long = n =>{
 if(n==1 || n==2) 1 else fib(n-1) + fib(n-2)
}
// Sequentiell:
val seq = List.range(1,45)
// Parallel:
val par = List.range(1, 45).par
// DualCore i5 Auswertung:
// seq = 12468t
// par = 10802t
```

Binäre Bäume

```
abstract class Tree[+T]
case object Empty extends Tree[Nothing]
case class Node[T]
(l:Tree[T], v:T, r:Tree[T]) extends Tree[T]

def leaf[T]: T => Tree[T] =
v => Node(Empty, v, Empty)

val myTreeI =Node(Node(leaf(1),leaf(4)),
5,(Node (leaf(6), 7, leaf(9))))

\\ Suchen in Binärbäumen:
def hat[T]: Tree[T] => T => Boolean = {
case Empty => v => false
case Node(l, n, r) =>
v => v == n || hat(l)(v) || hat(r)(v) }
\\ Aufruf: contains(myTreeS)("Echo")
\\ Traversierung in-order:
def flatten[T]: Tree[T] => List[T] = {
case Empty => Nil
case Node(l, n, r) =>
flatten(l):::n::flatten(r) }
```

Quicksort

```
def qsort: List[Int] => List[Int] = {
 case Nil => Nil
 case p::ps => qsort(ps.filter(_ <= p)) :::
  p :: qsort(ps.filter(_ > p)) }
```

Map-Reduce

```
def countrufer: String => List[(Char,Int)] =
 string => count(vowels.toList, string.toList)
def reduce: (Char, List[Int]) => (Char, Int)=
 {case (key,values) => ((key, values.sum))}
//Hilfsfunktion
def count[T]: (List[T], List[T]) =>
 List[(T, Int)] = (ks, vs) =>
 for (k <- ks if (vs.contains(k)))
 yield (k, vs.count(_==k))
//Ausgabe:
val bsp2 = List("uuu","oo")
=> List[(Char, Int)] = List((o,2), (u,3))
```