Parallele und funktionale Programmierung

Felix Leitl

17. Februar 2024

Petri-Netz

Grundlagen

Terminologie

- lebendig: wenn von jedem Knoten des Erreichbarkeitsgraphen ein Zustand erreichbar ist, von dem aus die Transition feuern kann
- beschränkt: Es werden nicht mehr Marken erzeugt
- Erreichbarkeitsgraph
- Schaltvektor
- Transitionsinnenvariante: Schaltvektor, durch welchen sich die Belegung nicht ändert
- Stelleninnenvarainte / P-Innenvarainte

Java

Grundlagen

Terminologie

- kritischer Abschnitt
- Verklemmung / Deadlock
- intrinsic lock (Marke jedes Objekts)
- Serialisierung (durch sycnchorized)
- Speichermodell
- Datensynchonisation:
  - Koordination der Zugriffe
  - Kein gemeinsamer Zustand
  - Unveränderlichke Datenstrukturen
- lokale Korrektheit: Eine Klasse C ist lokal korrekt, wenn keine Folge von Operationen zu einem nicht-spezifikationsgemäßen Zustand führen kann
- thread-sicher:
  - wenn sie lokal korrekt ist und
  - wenn bei beliebig verschränkten, nebenläufigen Ausführungen ihrer Methoden das Verhalten spezifikationsgemäß bezüglich der Pre- und Post-Bedingungen und der Klasseninvariante bleibt
- wechselseitiger Ausschluss
- Verhungerung

Speedup

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

- $T^*(n)$: ist die Laufzeit des schnellsten sequentiellen Algorithmus für das Problem bei Eingabegröße n
- $T_p(n)$: ist die Laufzeit des parallelen Programms mit p Prozessoren (bei Eingabegröße n)

Idealer Speedup bedeutet, dass der Speedup eines Programms mit der Prozessorzahl übereinstimmt
Gesetz von Amdahl:

$$S_p(n) = \frac{T^*(n)}{s(n) \cdot T^*(n) + \frac{p(n)}{p} \cdot T^*(n)}$$

- in sequentielle Anteile $s(n)$, die nicht parallelisierbar sind
- in parallele Anteile $p(n)$, die auf mehrere Threads verteilt werden können

Parallelisierung

Threads

```
public static class InnerThread extends Thread{
    private final int threadNumber:
    public InnerThread(int threadNumber){
        this.threadNumber = threadNumber;
    }
    public void run(){
        Sytsem.out.println("Thread: " + threadNumber)
    }
}
Thread[] innerThreads = new Thread[10];
for(i=0; i < 10; i++){
    innerThreads[i] = new InnerThread(i);
    innerThreads[i].start();
}
for thread in innerThread{
    try{
        thread.join();
    }catch(InterruptedExeption e){
        System.out.println(e + " has been thrown")
    }
}
```

Runnabels

```
public static class Runner implements Runnable{
    private final int threadNumber;
    public InnerThread(int threadNumber){
    this.threadNumber = threadNumber;
    }
    public void run(){
        System.out.println("Thread: " + i);
    }
}
Thread[] threads = new Thread[10];
for(i = 0; i < 10; i++){
    threads[i] = new Thread(new Runner(i))
    thread[i].start();
}
for thread in threads{
```

```
    try{
        thread.join();
    }catch(InterruptedExeption e){
        System.out.println(e + " has been thrown")
    }
}
```

ExecutorService

```
public static class Executor implements Runnable{
    private final int threadNumber;
    public Executor(int threadNumber){
        this.threadnumber = threadNumber;
    }
}
ExecutorService pool = Executors.newFixedThreadPool(10); // allows 10 parallel Threads
for(i = 0; i < 20; i++){
        pool.execute(new Executor(i));
}
pool.shutDown();
try{
        pool.awaitTermination(60, TimeUnit.SECONDS);
}catch(InterruptedExption e){
        System.out.printl(e + " has been thrown")
}
```

Sicherung

Synchronized

Synchronyzed garantiert nur bei Verwendungen der selben Marke, dass die Werte vor der Verwendung der Marke sichtbar werden

```
public synchronized void doSmth(){} // takes and gives intrinsic lock (this) back (atomic)
synchronized(this){ // takes and gives intrinsic lock (this / Object) back (atomic)
        // Do Something in here
}
```

volatile

Zur Sicherstellung der Sichtbarkeit und um primitive Type atomar zu machen

```
private volatile int number; // primitive types are atomic in Java, when used with volatile
```

AtomicInteger

```
int decrementAndGet();
int incrementAndGet();
```

```
int addAndGet(int delta);
int getAndDecrement();
int getAndIncrement();
int getAndAdd(int delta);

AtomicInteger number = new AtomicInteger(10);
number.incermentAndGet(5);
```

ReentrantLock

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try{
}finally{ // error handling
lock.unlock();
}
```

CyclicBarrier

```
CyclickBarrier barrier = new CyclicBarrier(3);
barrier.await();
barrier.await(60, timeUnit.SECONDS) // waits 60s
```

BlockingQueue

```
LinkedBlockingQueue<Integer> queue = new LinkedBlockingQueue
queue.put(10);
queue.offer(10); // returns true on success
queue.take(); // takes last element, when empty => blocks, till one element is in queue
queue.poll(); // returns last element or null
```

Scala

Grundlagen

Terminologie

- Seiteneffekt
- Werttreue: gleiche Eingabe ergibt gleiche Ausgabe
- REPL: Read-Eval-Print-Loop
- Objektorientiert: In Scala ist alles ein Objekt
- statisch typisiert: Typen werden zur Übersetzung festgelegt
- implizite Typinferenz
- Funktionsabstraktion: Funktionsdefinition
- Currying
- Paradigmen: imperativ vs. deklarativ
- Funktion höherer Ordnung: Parameter oder Rückgabewert sind Funktion
- anonyme Funktion: kann einer Variable zugeordnet werden
- n-stellige Funktion: n Parameterlisten
- Lazy Evaluation: Berechnung erst bei Verwendung
- LazyList: Unendlichkeit möglich
- Overhead: Mehrkosten durch Parallelisierung

Syntax

List

```
List(4,5,6): List[Int]
List(42, 'a', false): List[Any] // mehrere Typen in Liste => Supertyp Any
List.range(4,7) // 7 exclusiv
4 to 6
4::5::6::Nil // :: adds or removes the first element
List(4,5):::List(6) // adds two Lists together
List().length
List().head // first element
List().tail // List, without head
List().drop(3) // drops the first n elements
List().sum // adds all elements
List().product
```

```
List(4,5,6)(0) // returns first element
List(4,5,6).take(2) // returns first n elements
List().reverse
List(1,2,3).zip(List(4,5,6,7)) // => List((1,4),(2,5),(3,6)) drops the tail
ls.foldRight(Nil)((e, r) => if (r == Nil || e != r.head) e :: r else r)
List(1,2,3).filter(_ == 3) // returns with only 3
List(List(2), List(4,3)).flatten // returns List(2,3,4)
List(1,2,3).equals(List(1,2,3))
List().map(_ + 1) // increases every Element by one
"Scala".toList
List(1,2,3,4,5).takeWhile(_ < 3) // returns List(1,2)
List().dropWhile(_ < 5)
List('S', 'c', 'a').mkString
List(2,4,6).forall(_ % 2 == 0) // returns true, when all are true
List(2,3,4).exists(_ == 3) // returns true, when on is true
```

LazyList

```
1#::LazyList()
lazy val dikrim = b*b - 4  * a * c // calculation at first use
LazyList.iterate(BigInt(2))(_ + 1) // LazyList form 2 to infinity
```

Tupel

```
(false, 10): (Boolean, Int)
(false, 10, 'a'): (Boolean, Int, Char)
(false, 10)._1 // gives you the first element
```

Funktionen

```
def foo: Boolean => Char = a => if a 'a' else 'b'
def pair[A]: A => (A,A) = x => (x, x) // Generische Typparameter
def addd: Int => Int => Int => Int = x => y => z => x + y + z // Currying
def bar: List[Int] => Int = { // Mustervergleich
  case a::as => 1 + bar(as)
  case a => 0
}
def lol: Unit => Int = _ => 42 // Unit is nothing and _ ignores the constant
val f = (x: Int) => x + 1 // anonymus function
val f = _ + 1 // anonymus function
```

If-Else

```
if n>2 then n else -n
if (n>2) n else -n
if n>2 then n
else if n==2 then 2
```

```
else -n
case x if x > 2 // guard
```

For-comprehension

```
for x <- List.range(1,10) yield x*x
for x <- List.range(1,10) if x > 1 yield x*x
for x <- List.range(1,5)
    y <- List.range(x, 4) yield (x,y)
 for xs <- xss; x <- xs yield x // flatten an list
```

Type Aliasing

```
type Coordinate = (Int, Int)
type Move = Coordinate => Coordinate // function typ
def left: Move = {case (x, y) => (x-1, y)}
type Pair[T] = (T,T)
def mult: Pair[Int] => Int = (x, y) => x * y
def copy[T]: Pair[T] => T = x => (x, x) // generic typing
```

Typverbund

```
trait Shape // interface
case class Circle(r: Double) extends Shape // constructor is r: Double
case class Rect(l: Double, w: Double) extends Shape
case object Dot extends Shape // Singlton
def square: Double => Shape =
    n => Rect(n, n)
def area: Shape => Double ={
    case Circle(r) => math.Pi * r * r
    case Rect(l, w) => w * l
```

Enum

```
enum Answer:
        case Yes, No, Unknown // allows Yes instead of Answer.Yes
import Answer._
def flip: Answer => Answer = {
    case Yes => No
    case No => Yes
    case Unknown => Unknwon
```

MapReduce

```
val Letters: List[Char] = List.range('a', ('z' + 1).toChar)
def mapLetterCount: String => List[(Char, Int)] =
    string => count(letters, string.toList)
```

```
def reduceLetterCount: ((Char, List[Int])) => (Char, Int) = {
    case (key, values) => (key, values.sum)
}
mapReduce(mapLetterCount)(reduceLetterCount)(bsp1) // maps first, then reduces
```

Parallelisierung

Vorteeile

Alle parallelisierbaren Operationen arbeiten wegen „divide and conquer“ automatisch parallel
Programmierende muss sich nicht um Lastverteilung/Synchronisierung kümmern
Wird im Hintergrund mit einem schlafenden fork/join-Pool umgesetzt

.par

- Liefert Sicht auf Ursprüngliche Daten
- Die Verarbeitungsoperationen der Sicht sind jedoch parallelisiert
- Benötigt `import scala.collections.parallel.CollectionConverters._`
- Beispiele: `List(1,2,3).par.map(_ + 1)`

Future

- Einpacken einer Berechnung `f` in ein Futur: `val c = Future {f}`
- Warten auf das Ergebnis: `Await.result(c, Duration.Inf)`

Akka-Aktoren

- externe Bibliothek

Datentypen

- Boolean
- Int
- String
- Char
- BigInt
- BigDecimal
- Long

- Any: Übertyp aller Typen
- Nothing: Untertyp aller Typen

ParVector

Ist die parallele Version der Liste

```
342 +: 42 +: ParVector.fill(10)(100)
```

ParRange

```
ParRange(30, 47, 1, true) // == (30 to 47)
```

Future

```
val a = Future(x*x)
Await.result(a, Duration.Inf)
```