PFP-Zusammenfassung

9.8.2022 - Sommersemester 2022

1. Parallele Programme

Prozessor

- Berechnungseinheit im Rechner mit Registern, Cache, etc. → Ein Stück Hardware

Prozesse

- Ein laufendes *Programm* mit eigenem Speicherbereich
- Werden für eigenständige (lose gekoppelte) Programme verwendet, die im Normalfall unabhängig voneinander arbeiten

Aktivität (Thread)

- Eine logische Programmeinheit (Thread)
- Werden für nebenläufige Arbeiten auf denselben Daten verwendet, da sie über gemeinsamen Speicher verfügen
- Bilden zusammen ein logisches, paralleles Programm

SMP-Rechner

- Ablaufplaner (Scheduler): Teilt den einzelnen Threads oder ganzen Prozessen best. Zeiteinheiten auf den einzelnen Prozessoren zu
- Falls einem ganzen Prozess eine Zeiteinheit zugeteilt wird, müsste er diese dann selbst an seine Threads weiterverteilen.

2. Flynns Taxonomie

Single Instruction Single Data (SISD)

Ein Programm arbeitet auf eigenen Daten ohne externe Kommunikation.

Single Instruction Multiple Data (SIMD)

Dasselbe Programm wird auf unterschiedliche Daten angewendet (z. B. Videodekodierung).

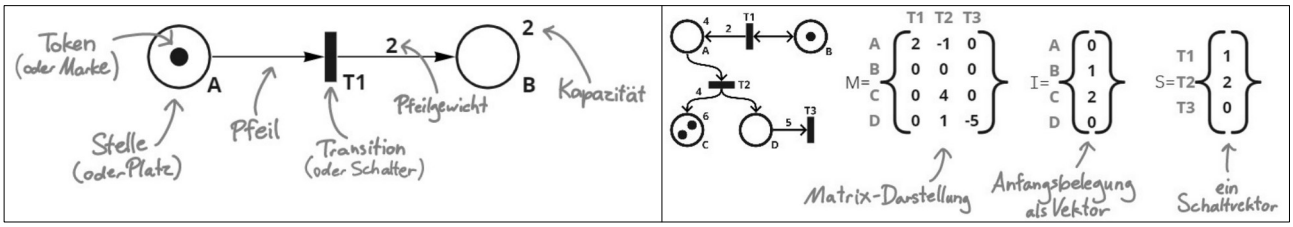
Multiple Instructions Single Data (MISD)

Verschiedene Programme, die auf gleichen Daten das gleiche Ergebnis berechnen (z. B. redundanter Bordcomputer in Flugzeug).

Multiple Instructions Multiple Data (MIMD)

Unterschiedliche Programme arbeiten auf jeweils eigenen Daten und kommunizieren untereinander.

3. PETRI-NETZE



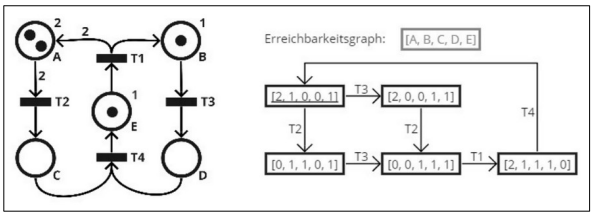
Matrixdarstellung: Zeilen: Stellen / Spalten: Transitionen (Obergrenzen sind nicht darstellbar)

Belegungsvektor: Anzahl der Tokens bei den jeweiligen Stellen.

Schaltvektor: Gibt an, wie oft die jeweiligen Transitionen schalten sollen.

Multipliziert man die Matrix eines Netzes mit einem Schaltvektor, erhält man einen Vektor wie sich die Belegungen der Stellen verändert haben.

Erreichbarkeitsgraph: Der Startzustand ist unterstrichen. Graph ist für *beschränkte* Petri-Netze endlich.



Lebendigkeit eines Petri-Netzes

Wenn von jedem Zustand des Erreichbarkeitsgraphen aus jede Transition (über Umwege) feuern könnte.

Beschränktheit eines Petri-Netzes

Wenn für jede Stelle im Petri-Netz eine „gedankliche Obergrenze“ an Marken angegeben werden könnte.

T-Invariante $C \cdot \vec{x} = \vec{0},\ x \neq \vec{0},\ x_i \geq 0$

Nach Ausführung des Schaltvektors $\vec{x}$ ist die Belegung des Netzes vor und nach dessen Ausführung gleich.

Überdeckung von T-Invarianten

Ein Petri-Netz, bei dem jede Transition in mindestens einer T-Invariante vorkommt. Ein solches Netz ist damit auch immer automatisch beschränkt und lebendig bzw. umgekehrt.

P-Invariante $\vec{y^T} \cdot C = \vec{0^T},\ \vec{y} \neq \vec{0},\ y_i \geq 0$

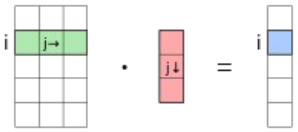
Egal wie oft alle Transitionen schalten, es gibt ein $\vec{y}$, das mit der jeweiligen Markenanzahl multipliziert immer eine Konstante ergibt.

Semipositive-Invariante: $y_i \neq 0$

Überdeckung von semipositiven P-Invarianten

Ein Petri-Netz, bei dem jede Stelle in mindestens einer semipositiven P-Invariante vorkommt, ist beschränkt.

(Matrix-Vektor-Multiplikation): Zeile der Matrix · Vektor



4. Thread-Erzeugung In Java

4.1 Thread

```
private static class myThread extends Thread{
    @Override
    public void run() {
    }
}
```

4.2 ExecutorService

```
ExecutorService e = Executors.newFixedThreadPool(4);

e.execute(/*Runnable*/);                          //Bei Runnables geht sowohl
e.submit(/*Runnable*/);                           //execute als auch submit

Future<Datentyp> f = e.submit(/*Callable*/);      //Bei Callables nur submit
try {
    Datentyp result = f.get();
} catch (Exception ex){
}

e.shutdown();
try {
    e.awaitTermination(60, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
}
```

4.3 Runnable

```
private static class myRunnable implements Runnable{
    @Override
    public void run() {
    }
}
```

4.4 Callable

```
private static class myCallable implements Callable<Datentyp>{
    @Override
    public Datentyp call() {
        return null;
    }
}
```

5. Synchronisation & Atomarität (in Java)

5.1 Datensynchronisation / Atomarität

Mehrere `Threads` können gemeinsam und gleichzeitig den Zustand eines Programms verändern.

Lösungen:

- ➔ Keine gleichzeitigen Zugriffe (Koordination der Zugriffe)
- ➔ Kein gemeinsamer Zustand (Methoden ohne Seiteneffekte, `Thread`-lokale Daten)
- ➔ Unveränderliche Datenstrukturen (`final` Variablen: Variablen die nach der Initialisierung nicht mehr änderbar sind)

5.2 Sichtbarkeitssynchronisation

Zustandsänderungen eines `Threads` sind nicht/nicht sofort von anderen `Threads` aus sichtbar.

Ursachen:

- Daten werden zur Zugriffsbeschleunigung in Registern gespeichert.
- Instruktionen dürfen zu Optimierungszwecken umsortiert werden, sofern dies die Logik nicht verändert.

→ Wird ein Wert in eine Variable geschrieben, kann nicht automatisch angenommen werden, dass andere `Threads` diesen Wert jemals sehen werden.

5.3 Lokale Korrektheit

Eine Klasse ist lokal korrekt, wenn keine Folge von Operationen zu einem nicht-spezifikationsgemäßen Zustand führen kann.

5.4 Thread-Sicherheit

Eine Klasse ist „thread-sicher“, wenn sie lokal korrekt ist und bei beliebig verschränkten, nebenläufigen Ausführungen ihrer Methoden das Verhalten spezifikationsgemäß bleibt.

Eine solche Klasse sorgt selbst für Datensynchronisation und verbirgt/kapselt diese vor dem Nutzer (Geheimnisprinzip (*boundary coordination*)).

5.5 SYNCHRONIZED

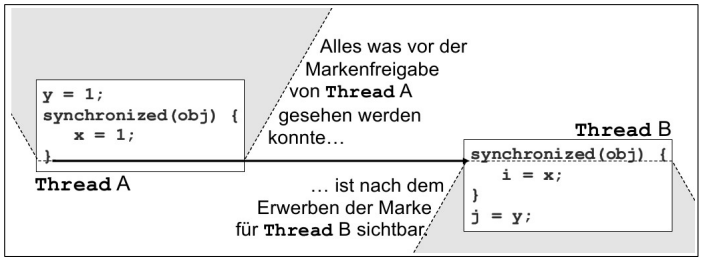
Jedes `Object` in Java ist mit einer Marke ausgestattet *(intrinsic lock)*.

Jeder `synchronized` Block, der mit derselben Marke synchronisiert ist, kann höchstens von einem `Thread` gleichzeitig ausgeführt werden.

→ Datensynchronisation im *wechselseitigen Ausschluss*

`Thread` ruft `synchronized`-Block/Methode auf:

1. Bekommt Marke oder wartet (wird vom System blockiert). Achtung: Kein FIFO-Prinzip es gibt keine feste Reihenfolge, in der wartende `Threads` Zugriff bekommen!
2. Alle, für einen vorherigen `Thread` mit gleichem Lock-Objekt sichtbaren, Variablen werden neu aus dem Speicher geladen.
3. Marke wird wieder freigegeben.
4. Alle (sichtbaren) Variablen werden (deterministisch nur) für nachfolgende `Threads` mit gleichem Lock-Objekt zurück in den Speicher geschrieben.



Achtung: Das Neuladen/Zurückschreiben der Variablen wird *nur beim gleichen Lock-Objekt* garantiert!

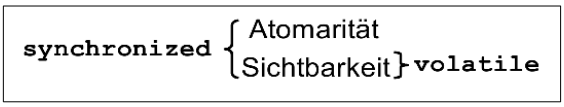
→ Nachteil: Die Verwaltung von wartenden `Threads` und das Neuladen kostet viel Laufzeit.

`this` zeigt für unterschiedliche Instanzen auf unterschiedliche Objekte → Unterschiedliche Marken

`static`-Methoden verwenden die Klassenmarke von `Klassenname.class`

Wird in einem `synchronized` Block ein weiterer `synchronized` Block mit derselben Marke aufgerufen, wird die Marke wiederverwendet und muss nicht vorher freigegeben werden (**reentrante Synchronisation**).

`synchronized` wird nicht vererbt!



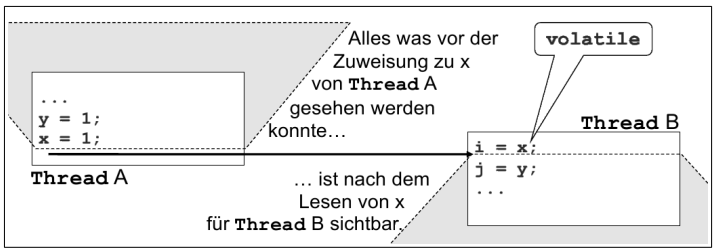
5.6 VOLATILE

`volatile` Variablen werden nicht in lokalen Registern der `Threads` gehalten.

Dazu begrenzen sie die Instruktionsumordnung. D. h. alle Befehle vor/hinter dem Zugriff auf eine `volatile`-Variable werden definitiv davor/dahinter ausgeführt.

→ Es ist immer der zuletzt hineingeschriebene Wert sichtbar.

→ Alle für vorherige `Threads` *sichtbaren* Variablen werden vor dem Zugriff in den Speicher geschrieben bzw. neu geladen.



Vereinfachende Vorstellung:

- Variablen stehen nur im Hauptspeicher
- Die Variable wird von einer Monitor-Klasse in einer `private` Variable gespeichert auf die nur mit `synchronized get-` und `set`-Methoden zugegriffen werden kann (aber ohne Markenverwaltung).

Referenzen:

Nur die Referenz ist `volatile`, nicht das Objekt, auf das verwiesen wird (Bsp.: Arrays, Instanzvariablen von Objekten). D. h. nur eine Zuweisung auf ein anderes Objekt ist für alle sichtbar, nicht ein geänderter Inhalt.

Primitive Datentypen:

werden immer atomar gelesen/geschrieben, außer `long` und `double`.

(64-Bit Werte werden intern als zwei 32-Bit Zahlen übertragen → Zwei Vorgänge, es wäre möglich, dass der erste Vorgang den alten ersten Teil liefert und der zweite Vorgang einen überschriebenen neuen Teil)

`volatile` hat also den gleichen Effekt wie `synchronized` (sowohl Atomarität als auch Sichtbarkeits-synchronisation), es können aber trotzdem Lesen-Schreiben-Ändern-Wettläufe (*race conditions*) auftreten!

`volatile` kann nur fehlerfrei verwendet werden (d. h. keine Lesen-Ändern-Schreiben-Wettläufe), wenn:

1. der neue Wert nicht vom alten abhängt oder höchstens ein Thread den Wert ändern kann.
2. die Variable nicht von anderen Variablen abhängt.

```
public class WrongZaehler {
   private volatile int zaehler;
   public int getNext() {
      return ++zaehler;
   }
   int getValue() {
      return zaehler;
   }
}
```

Trotz `volatile` kann die Lesen-Ändern-Schreiben-Wettlaufsituation auftreten.

5.7 Thread-Sichere Objekterzeugung

Konstruktoren sind nicht synchronized. D. h. wenn ein `Thread` ein Objekt für einen anderen `Thread` erzeugt, muss dieses neue Objekt (nach Initialisierung) auf einen Schlag (atomar) sichtbar gemacht werden.

Vier Möglichkeiten:

1. Die Referenz auf das neue Objekt in einer `volatile` Variable speichern.
2. Die Referenz auf das neue Objekt in einer `private` Variable eines anderen Objekts mittels `synchronized`-Methoden zwischenspeichern.
3. Die Referenz auf das neue Objekt in eine `final` Variable eines anderen `Thread`-sicher erzeugten Objekts zwischenspeichern.
4. Das neue Objekt mittels statischem Initialisierer erzeugen:

```
public static Object o = new Object();
```

5.8 Atomics

Wie `volatile`, nur dass es noch mehr *atomare* Methoden gibt. Beispielhaft die Methoden des `AtomicInteger` (weitere: `AtomicBoolean`, `AtomicIntegerArray`, `AtomicLong`, `AtomicLongArray`):

- `int get();`
- `int set(int value);`
- `boolean compareAndSet(int expect, int update);`
- `int decrementAndGet();`
- `int incrementAndGet();`
- `int addAndGet(int delta);`
- `int getAndDecrement();`
- `int getAndIncrement();`
- `int getAndAdd(int delta);`

Alle folgenden Klassen synchronisieren in ihren jeweiligen blockierenden Methoden die aufrufenden Threads.

Vorteil: Trotz der größeren Mächtigkeit kein aufwändiges Blockieren mit Marken.

5.9 ReentrantLock

IE = InterruptedException

- `void lock();` //Waiting if held by another thread (throws IE)
- `void unlock();` //Releases the lock
- `boolean tryLock();` //Does not wait if already held by another thread
- `boolean isLocked();` //Queries if this lock is held by any thread

5.10 Interface BlockingQueue<E>

- `void put(E);` //Inserts the specified element into this queue, //waiting if necessary for space (throws IE)
- `boolean offer(E);` //Inserts the specified element immediately, //returning false if no space available
- `E take();` //Retrieves and removes the head, waiting if //empty until an element is available (throws IE)
- `E poll(long timeout, TimeUnit unit);` //Retrieves and removes the head of this queue, //waiting up to the specified time (throws IE)

Eine Implementierung ist z. B. die `LinkedBlockingQueue<E>`:

```
BlockingQueue<E> queue = new LinkedBlockingQueue<E>();
```

Das Signal zum Beenden wird meistens über vorher definierte „Poison-Pills“ übermittelt.

5.11 Exchanger<V>

- `V exchange(V x);` //Waits for another thread and then transfers the //given obj to it, receiving its obj (throws IE)

5.12 CountDownLatch

Einmalig nutzbare Barriere, die einen (externen) (Countdown-)Zähler durch getrennte `await()`- und `countDown()`-Methoden implementiert:

- `CountDownLatch(int count);`
- `void await();` //Causes the current thread to wait until the //latch has counted down to zero (throws IE)
- `void countDown();` //Decrements the count of the latch, releasing all //waiting threads if the count reaches zero
- `long getCount();` //Returns the current count

5.13 CyclicBarrier

Eine wiederverwendbare Barriere, die einen (internen) (Countdown-)Zähler ohne getrennte `countDown()`-Methode implementiert:

- `CyclicBarrier(int n);`
- `int await();` //Waits until n threads invoked await (throws IE)

5.14 Weitere Synchronisierungsmöglichkeiten

- `void join();` //join() has a synchronization effect (throws IE)

6. PARALLELISIERUNGSARTEN

6.1 VERKLEMMUNG

Bedingungen einer Verklemmung

- Gegenseitiger Ausschluss
- Kein Entzug gemeinsam genutzter Ressourcen
- Iterative Anforderung
- Zirkuläre Abhängigkeit

Verhinderung einer Verklemmung

- Voll nebenläufiges Design
- Angeforderte Locks nach einer gewissen Zeit/Bedingung wieder freigeben
- Alle Locks atomar anfordern

Deadlock

Threads warten auf Ereignisse, ohne etwas zu tun (Bsp. Rechts-Vor-Links-Kreuzung).

Lifelock

Threads suchen (vergebens) *aktiv* nach einem Ausweg (Bsp. „unendliches“ Ausweichen).

6.2 SPEED-UP

Speed-Up $= \frac{T_{seq}}{T_{par}}$

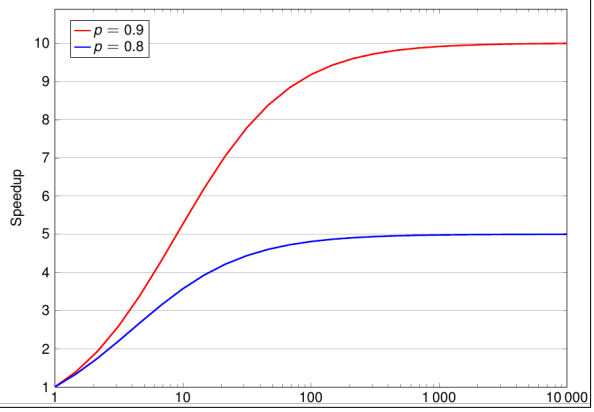
Gesetz von Amdahl

Der Speed-Up ist nur vom sequentiellen Anteil abhängig $\left(\frac{1}{s(n)},\ s = seq.\ Anteil\right)$.

- S_p : Speed-up
- T^* : sequentielle Laufzeit
- T_P : parallele Laufzeit mit P Prozessoren
- p : paralleler Anteil
- s : sequentieller Anteil
- P : Anzahl der verwendeten Prozessoren

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

$$S_p(n) = \frac{T^*(n)}{(s(n) + \frac{p(n)}{P}) * T^*(n)} = \frac{1}{s(n) + \frac{p(n)}{P}} \xrightarrow{P \to \infty} \frac{1}{s(n)}$$



6.3 Task-Parallele Vorgehensweisen

Client/Server (Klient/Dienstleister)

Clients stellen unabhängige Anfragen an den Server, die dieser bearbeitet (verteilt auf mehrere Prozesse).

Master/Worker (Chef/Arbeiter)

Der Master teilt eine Aufgabe in kleinere unabhängige Pakete auf, die von den Workers bearbeitet werden.

(Optional) Work-Stealing: Hat ein Worker keine Aufgabe, kann er anderen Workers Arbeit „abnehmen“.

Monte-Carlo-Simulation zur Berechnung von π

1. Verteile zufällig viele Punkte im Quadrat zwischen $(0, 0)$ und $(1, 1)$.
2. Zähle die Punkte, die im Einheitskreis liegen („im Einheitsviertelkreis“).
3. Das Verhältnis der Punkte im Kreis zur Gesamtanzahl der Punkte approximiert $\frac{\pi}{4}$.

Allgemein: Je mehr Zufallsereignisse durchgeführt werden, desto stärker nähert sich die Verteilung dem tatsächlichen Wert an.

Fließbandparallelisierung (Pipelining)

Einzelne Bearbeitungsschritte werden von verschiedenen Threads ausgeführt. Parallelität entsteht durch eine Parallelisierung der Arbeitsschritte auf unterschiedlichen (aufeinanderfolgenden) Daten.

Maximaler Speed-Up:

$$\frac{T_{seq}}{T_{par}} = \frac{n \cdot m}{n + m - 1} \quad ; \quad \lim_{m \to \infty} \frac{n \cdot m}{n + m - 1} = \lim_{m \to \infty} \frac{n}{\frac{n-1}{m} + 1} = n$$

Laufzeit: Bei n Paketen und k Stufenzeiten

$$T(n, k) = (n - 1) \cdot k_{max} + \sum_{i=1}^{k} k_i$$

Erklärung der Formel:
Das erste Paket durchläuft alle Stufen hintereinander $(\sum_{i=1}^{k} k_i)$. Alle darauffolgenden Pakete $(n - 1)$ werden durch die längste Stufe verzögert $(\cdot k_{max})$.

Implementierungsarten:

Exchanger	BlockingQueue
Direktes Weiterreichen der Ergebnisse → Speicherersparnis	Kommunikation über eine Warteschlange → Laufzeitschwankungen werden abgemildert

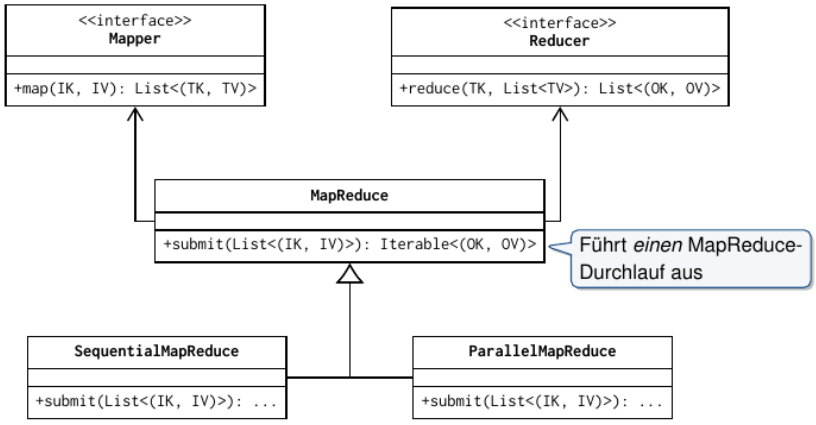
6.4 Daten-Parallele Vorgehensweisen

Geometrische Dekomposition

Voneinander unabhängige Daten, auf die jeweils die gleiche Funktion angewendet wird, werden gleichmäßig „räumlich“ aufgeteilt (z. B. in einem 2D-Array), bspw. beim Game-Of-Life.

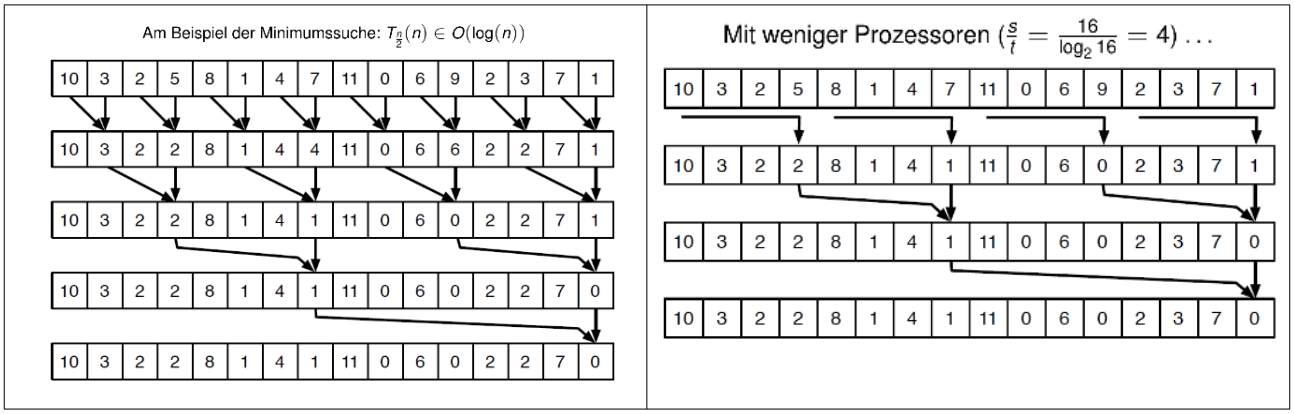
Map-Reduce

map	combine	reduce
`(in-key, in-value) → [(tmp-key, tmp-value)]`	`[(tmp-key, tmp-value)] → (tmp-key, [tmp-value])`	`(tmp-key, [tmp-value]) → (out-key, out-value)`
Eingabedaten werden als (Schlüssel, Wert)-Tupel eingelesen. Anschließend werden evtl. mehrere Zwischenergebnisse mit jeweils eigenem Schlüssel berechnet.	Alle Zwischenergebnisse mit gleichem Schlüssel werden zu einer Liste zusammengefasst.	Aus den Zwischenergebnissen eines Schlüssels wird das Endergebnis berechnet.



Lemma von Brent

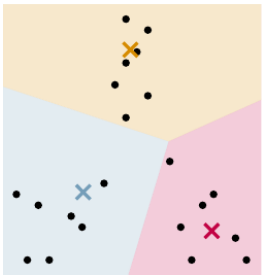
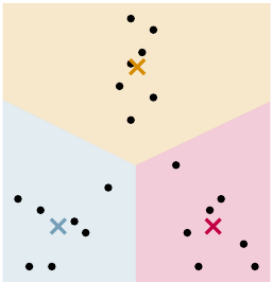
Wenn es einen p-verteilten statischen Algorithmus gibt (ein Algorithmus, dessen Zuordnung von Threads vorbestimmt ist), der in $O(t)$ liegt und insgesamt s Schritte ausführt, dann gibt es einen $\frac{s}{t}$-verteilten Algorithmus mit gleichem Aufwand.



_k_-means-Clustering

Dieses Verfahren erkennt die Struktur in Daten selbstständig („unsupervised learning“), ohne über den Inhalt Bescheid zu wissen. Allerdings muss die Anzahl k an Clustern zu Beginn gegeben sein:

1. Erzeuge k zufällige Mittelpunkte („Centroids“) (für jedes Cluster einen)
2. Folgende zwei Schritte wiederholen:
    1. Für jeden Punkt den naheliegendsten Mittelpunkt bestimmen bzw. das passende Cluster
    2. Neue Position der Mittelpunkte innerhalb ihres Clusters berechnen

Datenpunkte dem naheliegendsten Mittelpunkt zuordnen	Neue Mittelpunkte berechnen
	

Zeigerverdopplung

<u>Sequentiell</u>

1. Rang $r =$ Länge der Liste
2. Verfolge die Verweise, verringere bei jedem Schritt den Rang r um 1

Laufzeit: $T^*(n) = n$

<u>Parallel (Algorithmus von Wyllie)</u>

1. Für jedes Element der Liste eine Zahl Rang r auf 0 setzen, außer beim letzten auf $1\,(next = null)$
2. Distanz $d = 1$
3. Solange die Verweise der Elemente nicht auf $null$ zeigen:
    1. Zeigt Verweis auf ein Element mit Rang $r \neq 0$, dann setze eigenen Rang auf $r = d + r$
    2. Alle Verweise werden auf den Verweis des Nachfolgers gesetzt
    3. Verdopplung von d

Laufzeit: $T_n(n) \in O(log_2 n)$ Speichereffizienz: $E_n(n) = \frac{T^*(n)}{n \cdot T_n(n)} \in O\left(\frac{1}{log_2 n}\right)$

7. Scala

7.1 Syntaktische Konstrukte

Name	Beschreibung	Beispiel
Funktionsdeklaration (Lambda-Syntax)	(Anonyme) Funktionen, bei denen das Ergebnis der letzten Zeile zurückgegeben wird. Sind auch einer Variable zuweisbar.	`def a: Int => Int = x => 2 * x` `val y = (x: Int) => 2 * x`
Konstanten (`val`)	Unveränderbare (*immutable*) Variablen, analog zu `final` Variablen in Java.	`val x = 4` `a(2) == y(2) == x == 4`
Platzhaltersyntax	Kurzschreibweise für Funktionen mit nur einem Parameter.	`def a: Int => Int = 2 * _`

Currying (Partielle Auswertung)	Funktionen mit mehreren Parameterlisten. Können als Funktionen, die andere Funktionen zurückgeben, betrachtet werden. Sind entsprechend auch einer Variable zuweisbar.	`def f: Int => Int => Int = a => b => a - b`
		`val g = f(11)`
		`f(11)(5) == g(5) == 6`

Listengeneratoren (`for … yield`)	Generiert eine Liste, die für alle Werte des sog. *Generators*, für die der Wächter (*guard*) erfüllt ist, den Wert enthält, den der Ausdruck nach `yield` zurückgibt.	`for (x <- List.range(0, 10) if (x % 2 == 0))`
		`yield x * 2`
		`== List(0, 4, 8, 12, 16)`
		Generator
		Wächter (guard)

Pattern matching (Mustervergleich)	Vergleicht einen Wert mit verschiedenen Mustern. Die Anweisungen des ersten passenden Musters (von oben nach unten) werden ausgeführt. Zusätzliche Angabe eines Wächters (*guard*) möglich.	`x match {`
		`  case Nil => 0`
		`  case h :: tail if (h < 0) => 3`
		`  case h :: tail => 7`
		`}`

Lazy Evaluation (`lazy val`)	Wertet die rechte Seite der Definition erst aus, sobald sie zwingend benötigt wird.	`lazy val x: Int = 1 + x`
		`//Noch kein Fehler`
		`print(x) //StackOverflowError`

Type Aliasing (`type`)	Möglichkeit, bestehenden Datenstrukturen einen neuen Namen (*alias*) zu geben.	`type Coord = (Int, Int)` `type Move = Coord => Coord`
Aufzählungsdatentpyen (`enum`)	Analog zu Java. Können mittels *pattern matching* unterschieden werden.	`enum Answer:` `  case Yes, No, Unknown`

Generische Datentypen (Generics)	Typparameter werden mit eckigen Klammern angezeigt, Kovarianz durch ein `+` .	`abstract class Option[+T]`
		`case object None extends Option[Nothing]`
		`case class Some[+T](v: T) extends Option[T]`
		`x match {`
		`  case None => -1`
		`  case Some(z) => z`
		`}`

Interfaces (`trait`)	Schlüsselwort `trait` (ansonsten wie Java).	
Algebraische Datentypen (`case class` / `case object`)	Datentypen bzw. Objekte, die nur aus ihrem Konstruktor bestehen (also keine Methoden haben). Können mittels *pattern matching* entpackt werden.	

7.2 Tupel

Name	Beschreibung	Beispiel
`(x, …)`	Erzeugt ein Tupel bestehend aus den angegebenen Elementen (maximal 22).	`(4, 5)`
`_1, _2, …`	Liefert das *n*-te Element eines Tupels.	`(4, 5, 6)._2 == 5`

7.3 List-Methoden

Name	Beschreibung	Beispiel
`Nil`	Erzeugt eine leere `List`.	`Nil == List()`
`::`	Erzeugt eine `List` bestehend aus einem Kopfelement und einer Restliste.	`4 :: List(5, 6)` `== List(4, 5, 6)`
`:::`	Erzeugt eine `List`, bei der eine linke und rechte `List` zusammengehängt wurden.	`List(4, 5) ::: List(5, 6)` `== List(4, 5, 5, 6)`
`range(s, e)`	Erzeugt eine `List` mit allen Zahlen zwischen der unteren (eingeschlossenen) und der oberen (ausgeschlossenen) Grenze.	`List.range(4, 7)` `== List(4, 5, 6)`
`fill(n)(f)`	Erzeugt eine `List` mit *n* Elementen, deren Werte jeweils durch eine Funktion berechnet werden.	`List.fill(5)(3)` `== List(3, 3, 3, 3, 3)`

7.4 LazyList-Methoden

Name	Beschreibung	Beispiel
`#::`	Erzeugt eine `LazyList` bestehend aus einem Kopfelement und einer Restliste.	`4 #:: LazyList(5, 6)` `== LazyList(4, 5, 6)`

`#:::`	Erzeugt eine `LazyList`, bei der eine linke und rechte `LazyList` zusammengehängt wurden.	`LazyList(4, 5) #:::`
		`LazyList(5, 6)`
		`== LazyList(4, 5, 5, 6)`

`toList`	Wandelt eine endliche `LazyList` in eine `List` um (terminiert nicht bei unendl. `LazyList`).	`LazyList(4, 5, 6).toList` `== List(4, 5, 6)`

`iterate(x)(f)`	Erzeugt eine `LazyList` durch wiederholte Anwendung einer Funktion, beginnend bei einem Startwert ($[x, f(x), f(f(x)), ...]$).	`LazyList.iterate(4)(_ * 2)`
		`.take(3).toList`
		`== List(4, 8, 16)`

7.5 List- & LazyList-Methoden

Name	Beschreibung	Beispiel
`apply(n)`	Liefert das *n*-te Element einer Liste.	`List(4, 5, 6)(2) == 6`
`head`	Liefert das erste Element einer Liste, sofern vorhanden (sonst `NoSuchElementException`).	`List(4, 5, 6).head == 4` `Nil.head //Exception`
`tail`	Entfernt das erste Element einer Liste, sofern vorhanden (sonst `NoSuchElementException`).	`List(4, 5, 6).tail` `== List(5, 6)`
`last`	Liefert das letzte Element einer Liste, sofern vorhanden (sonst `NoSuchElementException`).	`List(4, 5, 6).last == 6`
`length`	Liefert die Länge einer Liste.	`List(4, 5, 6).length == 3`
`min / max`	Liefert das kleinste/größte Element einer Liste.	`List(5, 4, 6, 5).min == 4`
`sum / product`	Liefert die (Gesamt-)Summe/Produkt aller Elemente einer Liste.	`List(4, 5, 6).sum == 15` `List(4, 5, 6).product == 120`
`take(n)`	Liefert die ersten *n* Elemente einer Liste, sofern vorhanden (sonst so viele Elemente wie möglich).	`List(4, 5, 6).take(2)` `== List(4, 5)`
`drop(n)`	Entfernt die ersten *n* Elemente aus einer Liste (Gegenteil von `take`).	`List(4, 5, 6).drop(2)` `== List(6)`

`takeWhile`
`(T => Boolean)`
	Wählt solange Elemente aus einer Liste aus, wie die gegebene Funktion zutrifft.	`List(4, 6, 8, 9, 10, 12)`
		`.takeWhile(_ % 2 == 0)`
		`== List(4, 6, 8)`

`dropWhile`
`(T => Boolean)`
	Entfernt solange Elemente aus einer Liste, wie die gegebene Funktion zutrifft (Gegenteil von `takeWhile`).	`List(4, 6, 8, 9, 10, 12)`
		`.dropWhile(_ % 2 == 0)`
		`== List(9, 10, 12)`

`map` `(T => R)`	Wendet die gegebene Funktion auf jedes einzelne Element der Liste an.	`List(4, 5, 6).map(_ * 2)` `== List(8, 10, 12)`

`flatMap`
`(T => List[R])`
	Wie `map`, allerdings muss die gegebene Funktion eine Sammlung von Elementen (z. B. eine `List`) liefern. Diese Listen werden dann konkateniert.	`List(4, 5, 6).flatMap(`
		`  x => List.range(x - 2, x))`
		`== List(2, 3, 3, 4, 4, 5)`

`filter`
`(T => Boolean)`
	Entfernt (filtert) alle Elemente einer Liste, die eine gegebene Bedingung nicht erfüllen.	`List(4, 5, 6, 7)`
		`.filter(_ % 2 == 0)`
		`== List(4, 6)`

`forall`
`(T => Boolean)`
	Prüft, ob die gegebene Bedingung für alle Elemente zutrifft (∀).	`List(4, 5, 6, 7)`
		`.forall(_ % 2 == 0)`
		`== false`

`count`
`(T => Boolean)`
	Zählt, wie viele Elemente in einer Liste die gegebene Bedingung erfüllen.	`List(4, 5, 6, 7)`
		`.count(_ % 2 == 0)`
		`== 2`

Name	Beschreibung	Beispiel
`exists (T => Boolean)`	Prüft, ob die gegebene Bedingung für mindestens ein Element zutrifft (∃).	`List(4, 5, 6, 7) .exists(_ % 2 == 0) == true`
`contains(e)`	Prüft, ob das Element *e* in einer Liste enthalten ist.	`List(4, 5, 6, 7).contains(5) == true`
`zip(List)`	Fügt die Elemente zweier Listen paarweise (als Tupel) zusammen (die jeweils kürzere Liste bestimmt die Länge der Gesamtliste).	`List(4, 5, 6) .zip(List('A', 'B')) == List((4, 'A'), (5, 'B'))`
`unzip (List[(A, B)])`	Erzeugt aus einer Liste an Tupeln ein einziges Tupel bestehend aus zwei einzelnen Listen (Gegenteil von `zip`).	`List((4, 'A'), (5,'B')).unzip == (List(4, 5), List('A', 'B'))`
`foldLeft (s: R) ((R, T) => R)`	Beginnend bei einem „Startergebnis“, wird von links nach rechts mit jedem Element eine Funktion zusammen mit dem vorherigen Ergebnis aufgerufen. Das Ergebnis des letzten Elements wird als Endergebnis zurückgegeben.	`List(4, 5, 6).foldLeft(0) ((a, b) => a - b) == (((0 - 4) - 5) - 6) == -15`
`foldRight (s: R) ((T, R) => R)`	Analog zu `foldLeft` nur von rechts nach links. Außerdem ist das vorherige Ergebnis nun das zweite (rechte) Element im übergebenen Tupel (das erste ist das aktuelle Element).	`List(4, 5, 6).foldRight(0) ((a, b) => a - b) == (4 - (5 - (6 - 0))) == 5`
`fold (s: R) (R, T => R)`	Ähnlich zu `foldLeft/Right`, nur, dass die Elemente in beliebiger Reihenfolge verarbeitet werden (*divide/conquer*-Verfahren).	`List(4, 5, 6).fold(0) ((a, b) => a + b) == 15`
`sortWith ((A, B) => Boolean)`	Sortiert eine Liste nach einer gegebenen Bedingung (alle Elemente werden solange getauscht, bis die Bedingung für alle `true` ist).	`List(2, 9, 12, 1) .sortWith((a, b) => a < b) == List(1, 2, 9, 12)`
`reverse`	Liefert eine Liste, deren Elemente in umgekehrter Reihenfolge stehen.	`List(4, 5, 6).reverse == List(6, 5, 4)`

7.6 Parallelisierung

Name	Beschreibung	Beispiel
`par`	Konvertiert/wrapped eine API-Collection (z. B. `List`) in ihre jeweilige parallelisierte Variante.	`List.range(0, 1000) .par.map(_ * 2).toList`
`Future`	Startet eine nebenläufige Berechnung.	`val fut = Future { foo() } Await.result(fut, Duration.Inf)`
`Await.result`	Wartet auf Beendigung einer nebenläufigen Berechnung.	
`akka.actor`	Kommunikation ausschließlich durch Nachrichtenaustausch (Threads haben keine gemeinsamen Daten- oder Kontrollstrukturen → funktional & seiteneffektfrei). Ein *akka-actor* ist ein Software-Agent, vergleichbar mit einem erweiterten Zustandsautomaten (EFSM).	