HSR ParProg Zusammenfassung

Parallele Programmierung

Emanuel Duss

2014-08-29 16:39

Inhaltsverzeichnis

1 Multi-Threading Grundlagen (Threads)

1.1 Begriffe

Prozess (Schwergewichtsprozess) Parallel laufende Programm-Instanz. Eigener Adressraum pro Prozess

Thread (Leichtgewichtsprozess) Parallele Ablaufsequenz innerhalb eines Programms. Teilen gleichen Adressraum im Prozess. Jeder Thread hat einen eigenen Stack (braucht Speicher).

1.2 Thread erzeugen

1.2.1 Erben von der Klasse `Thread`

```
class SimpleThread extends Thread {
  @Override
  public void run() {
    // DO STUFF
  }
}
Thread myThread = new SimpleThread(); // Thread instanzieren
myThread.start(); // Thread starten (führt run-Methode aus)
```

1.2.2 Implementieren des Interfaces `Runnable`

```
class SimpleLogic implements Runnable {
  @Override
  public void run() {
    // DO STUFF
  }
}
Thread myThread = new Thread(new SimpleLogic());
```

1.2.3 Anonyme innere Klasse

```
void startMyThread(final String label, final int nofIt) { // Explizit final!
  new Thread(new Runnable() {
    @Override public void run() {
      for (int i = 0; i < nofIt; i++) {
        System.out.println(i + " " + label);
      }
    }
  }).start();
}

startMyThread("A", 10);
startMyThread("B", 10);
```

1.2.4 Java 8 Lambda

```
Thread myThread = new Thread(() -> {
  // thread behavior
});
myThread.start();
```

1.3 Threads Kontrollieren

- Ein Thread kann nur einmal gestartet werden.
- Thread in Wartezustand schicken: `Thread.sleep(milliseconds)`
- Prozessor freigeben (gibt mehr Wechsel): `Thread.yield()`
- Auf die Beendigung eines Threads warten: `Thread.join()`
- Thread unterbrechen: `Thread.interrupt()`
- Aktuellen Thread anzeigen: `static Thread currentThread()`
  - `Thread.currentThread().join()` würde den Code blockieren
- Thread als Daemon markieren (Beendet sich wenn sich das Programm beendet): `void setDaemon(boolean on)`
- Name setzen und lesen: `String getName()` und `void setName(String name)`

2 Monitor Synchronisation

2.1 Synchronisation

- Synchronisation = Einschränkung der Nebenläufigkeit
- Ohne Synchronisation laufen Threads beliebig verzahnt oder parallel
- Wenn die Threads nicht unabhängig sind, kann das Probleme geben
- Adressraum und Heap wird geteilt: Zugriff auf selbe Objekte und Variabeln: Instanzvariabeln, statische Variabeln, Elemente in einem Array
- Beschränkung mittels Synchronisation auf gegenseitiger Ausschluss und warten auf Bedingungen
- Race Condition: Unkontrollierte oder falsche Interaktionen zwischen Threads
- Kritische Abschnitte markieren (Gegenseitiger Ausschluss = Mutual Exclusion)

2.2 Java `synchronized`

- In einer Instanz kann nur ein Thread in genau einer `synchronized` Methode sein.
- Lock wird beim Exit freigegeben (Block, Return Statement, Unbehandelte Exception)
- Rekursive Locks sind möglich.

Methode als `synchronized`:

```
public synchronized void foo(int bar) { /* baz */ } // Object Lock
static public synchronized void bar(int foo) { /* baz */ } // Class Lock
```

Block als `synchronized`:

```
public void foo(int bar) { // Object Lock
  synchronized(this) { // Monitor Lock auf `this`
    // foo
  }
}
static public void bar(int foo) { // Class Lock
  synchronized(this) { // Monitor Lock auf `this`
    // foo
  }
}
```

2.3 Monitor Konzept

- Nur ein Thread operiert zur gleichen Zeit im Monitor
- Non-private Methoden alle synchronized, private Variabeln
- `synchronized`: Trete in Monitor ein
- `wait()`: Gibt Monitor-Lock temporär frei (wichtig: throws `InterruptedException`‘)
- `notify()`: beliebigen wartenden Thread im Monitor aufweckern
- `notifyAll()`: alle Threads im Monitor aufweckern

```
class BankAccount {
  private int balance = 0;
  public synchronized void withdraw(int amount) throws InterruptedException {
    while (amount > balance)
      wait(); // Warte auf Bedingung (oder `wait(1000)` mit Timeout (InterruptedException)
    balance -= amount;
  } // Verlasse Monitor

  public synchronized void deposit(int amount) {
```

```
    balance += amount;
    notifyAll(); // Signalisiere Bedingung, Wecke alle im Monitor wartende Threads
  }
}
```

Problem 1: Wait mit If: Aufgeweckter Thread muss um Monitor-Eintritt kämpfen: Während der Zeit die zum Eintreten in den Monitor benötigt wird kann ein anderer Thread den aktuellen unterbrechen ("überholen") und die Wartebedingung invalidieren (overrun issue). Zudem können Spurious Wakeups von Java dazu führen, dass Threads einfach so aufgeweckt werden.

Lösung: Wartebedingung in Schlaufe testen:

```
while (!condition) { wait(); }
```

Problem 2: Single Notify

Lösung: Bedingung profilaktisch an alle informieren: `notifyAll()` verwenden. Vielleicht weckt man jemand, der nicht auf die abgearbeitete Bedingung wartet und deshalb weiterhin blockiert.

Lösung: Einzelne Objekte Locken:

```
private Queue<T> queue;
public T get() {
  synchronized(queue){ // Lock auf queue
    while(queue.isEmpty())
      queue.wait(); // Wichtig: Hier queue.wait() statt nur wait()
    return queue.remove();
  }
}
```

Wenn Lock auf einzelnes Objekt, muss auch das Notify auf die queue aufgerufen werden (`queue.NotifyAll()`).

2.4 Wann Single Notify?

- Wenn nur *eineßemantische Bedingung (Bedingung interessiert* jeden* wartenden Thread.)
- Bedingung gilt jeweils nur für einen: One-In/One-Out (Nur ein einziger wartender Thread kann weitermachen).
- Aber: Fairness-Problem in Java: Java weckt beliebigen Thread (Grund für `notifyAll()`, aber immer noch nicht ganz fair)

3 Spezifische Synchronisationsprimitiven

3.1 Semaphore

- Objekt mit Zähler: Anzahl noch freie Ressourcen (nur positiv)
- Methode `acquire()`:
  - Beziehe Ressource (auch mehrere mit `acquire(23)` möglich)
  - Warten, bis Ressource verfügbar (Zähler == 0)
  - Sonst Zähler dekrementieren
- Methode `release()`:
  - Ressource freigeben
  - Zähler inkrementieren
- Fairness kann in Konstruktor mit `true` erzwungen werden (FIFO).

Beispiel normaler Semaphor:

```
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  private Semaphore upperLimit = new Semaphore(Capacity, true); // Neuer Semaphor
  private Semaphore lowerLimit = new Semaphore(0, true); // true: Fairness erzwingen (FIFO)

  public void put(T item) throws InterruptedException {
    upperLimit.acquire(); // Nicht voll
    synchronized (queue) { queue.add(item); }
    lowerLimit.release();
  }
  public T get() throws InterruptedException {
    T item;
    lowerLimit.acquire(); // Nicht Leer
    synchronized(queue){ item = queue.remove(); }
    upperLimit.release(); // Ressource freigeben
    return item;
  }
}
```

Beispiel binärer Semaphor (Mutex = Mutial Exclusion):

```
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  private Semaphore upperLimit = new Semaphore(Capacity, true);
  private Semaphore lowerLimit = new Semaphore(0, true);
  private Semaphore mutex = new Semaphore(1, true); // Mutual exclusion

  public void put(T item) throws InterruptedException {
    upperLimit.acquire();
    mutex.acquire(); queue.add(item); mutex.release();
    lowerLimit.release();
  }
  public T get() throws InterruptedException {
    lowerLimit.acquire();
    mutex.acquire(); T item = queue.remove(); mutex.release();
    upperLimit.release();
    return item; // Besser noch in try-finally setzen
  }
}
```

3.2 Lock & Conditions

- Auch ein Monitor Konzept: Monitor mit mehreren Wartelisten für verschiedene Bedingungen.
- Pro Bedingung gibt es eine Warteschleife.
- Fairness ist im Konstruktor mit `true` aktivierbar.
- Vorteil: Gezieltes Notifizieren der wartenden Threads (statt alle zu notifizieren).

```
import java.util.concurrent.locks.Lock;
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  private Lock monitor = new ReentrantLock(true); // Fairness aktivieren (Default false)
  private Condition nonFull = monitor.newCondition();
  private Condition nonEmpty = monitor.newCondition();

  public void put(T item) throws InterruptedException {
    monitor.lock();
    try {
      while (queue.size() == Capacity) // Schlaufe wegen Überholproblem & Spurious Wakeup
        nonFull.await();
      queue.add(item);
      nonEmpty.signal();
    } finally { monitor.unlock(); } // Damit bei InterruptedException der Lock nicht behalten wird
  }
  public T get() throws InterruptedException {
    monitor.lock();
    try {
      while (queue.size() == 0)
        nonEmpty.await();
      T item = queue.remove();
      nonFull.signal(); // Gezieltes Einzel-Signal für diese Bedingung
      return item;
    } finally { monitor.unlock(); }
  }
}
```

3.3 ReadWrite Locks

- Problem: Exklusives Recht: Es kann gleichzeitig nur einer im Monitor sein.
- Paralleles `read` möglich. Sobald aber ein `write` vorkommt, nicht mehr möglich.
- Conditions nur auf Write-Locks, da sich auch nur dort etwas ändern kann.
- Write-Locks haben priorität vor den Read-Locks, mit `true` beim Konstruktoraufruf wird der Lock auf fair geschaltet.
- Bei Verschachtelung können DeadLocks entstehen (`R`-Lock, `RW`-Lock, `R`-Unlock, `RW`-Unlock).

```
ReadWriteLock rwLock = new ReentrantReadWriteLock(true); // Fairer Lock

rwLock.readLock().lock(); // Shared Lock (nur lesen)
// read-only accesses
rwLock.readLock().unlock();

rwLock.writeLock().lock(); // Exclusive Lock (Mit schreiben)
// write (and read) accesses
rwLock.writeLock().unlock();
```

Beispiel:

```
class NameDatabase {
  private Collection<String> names = new HashSet<>();
  private ReadWriteLock rwLock = new ReentrantReadWriteLock(true);

  public Collection<String> find(String pattern) {
    rwLock.readLock().lock();
    try {
      return names.stream().filter(n -> n.matches(pattern)); // Read-only
    } finally {
      rwLock.readLock().unlock();
    }
  }
  public void put(String name) {
    rwLock.writeLock().lock();
    try {
      names.add(name); // Write
    } finally {
      rwLock.writeLock().unlock();
    }
  }
}
```

3.4 Count Down Latch

- Synchronisationsprimitive mit Countdown Zähler.
- Bleibt für immer offen.
- `await()`: Warten bis Countdown `0` ist (kann `InterruptedException` werfen)
- Blockiert bei > 0.
- `countDown()`: Zähler zählt um 1 runter.

```
CountDownLatch carsReady = new CountDownLatch(N); // Warte auf N cars
CountDownLatch startSignal = new CountDownLatch(1); // Einer gibt Signal

carsReady.countDown();
startSignal.await();
carsReady.await();
startSignal.countDown();
```

3.5 Cyclic Barrier

- Für zyklische Synchronisationspunkte.
- `await()` blockiert, bis so viele Threads `await()` aufgerufen haben (kann `InterruptedException` werfen).
- Man muss am Anfang wissen, wieviele Threads es sind.
- Rükgabewert: > 0 warten und $== 0$ Barriere öffnen und Warteende aufwecken
- Falls mehrere Threads jeweils ein Objekt der Klasse haben, muss die Barriere `static` sein, damit sie auf dieselbe Barriere zugreifen.

```
CyclicBarrier raceStart = new CyclicBarrier(N); // N Cars
while (true) {
  raceStart.await(); // Warten bis alle bereit
  // play concurrently with others ...
}
```

3.6 Phaser

- Verallgemeinerte `Cyclic Barrier`, mit etwas mehr Funktionen.
- Während der Laufzeit Threads mehrere hinzufügen/entfernen.

```
Phaser phaser = new Phaser(0); // Anfangs keine Player
phaser.register(); // Anmelden: In der nächsten Runde bin ich dabei
while (...) {
  phaser.arriveAndAwaitAdvance(); // Anmelden
  playRound();
}
phaser.arriveAndDeregister(); // Abmelden
```

3.7 Exchanger

- Daten austauschen
- Blockiert, bis anderer Thread auch `exchange()` aufruft.
- Liefert Argument `x` des jeweils anderen Threads.

```
Exchange<V> e;
e.exchange(objA); // Thread 1
e.exchange(objB); // Thread 2
```

Beispiel

```
Exchanger<Integer> exchanger = new Exchanger<>();
  for (int k = 0; k < 2; k++) {
    new Thread(() -> {
      for (int in = 0; in < 5; in++) {
        try {
          int out = exchanger.exchange(in);
          System.out.println(Thread.currentThread().getName() + " got " + out);
      } catch (InterruptedException e) { }
    }
  }).start();
}
```

3.8 Vergleich der Synchronisationspunkten

- Monnitor: Nur 1 Warteraum.
- Lock & Condition: Mehrere Warteräume.
- Semaphore: Wenn `== 0`, erhöhbar (Ressourcen beziehen und freigeben).
- Count Down Latch: Wenn `> 0`, nicht erhöhbar, nicht rezyklierbar.
- Cyclic Barrier: Rezyklierbar.

4 Gefahren der Nebenläufigkeit

4.1 Race Conditions

- Race Condition: Ungenügend synchronisierte Zugriffe auf gemeinsame Ressourcen.
- Sobald auf eine Variable ein Write-Zugriff geschieht, können Race Conditions auftreten.
- Unerwartete Resultate und Effekte.
- Fehler können sporadisch oder selten auftreten.
- Schwierig durch Tests zu finden.
- Data Races (low level)
  - Unsynchronisierter Zugriff auf gleichen Speicher (Variablen, Array-Element)
  - Mindestens ein Write-Zugriff (Reine Read-Zugriffe sind nicht problematisch)
- Semantisch höhere Race Conditions (high-level)
  - Nicht genügend grosse synchronisierte Blöcke (Critical Sections nicht geschützt)
  - Sollte z. B. zusammengehörenh: `account.setBalance(account.getBalance() + 100);`
- Lösung: Genügend synchronisieren.

4.1.1 Was synchronisieren?

- Einfach alles synchronisieren?
  - Weitere Nebenläufigkeitsfehler
  - Synchronisation ist teuer
- Confinement (Einsperrung)
  - Nur Read-Only Zugriffe
  - Objekt gehört nur einem Thread zu einer Zeit
- Verstecktes Multi-Threading
  - Finalizers: Über speziellen Finalizer Thread
  - Timers: Handler durch separaten Thread
  - Externe Libraries & Frameworks

4.1.2 Immutable Object (Unveränderlichkeit)

- Objekte nur lesendem Zugriff
  - Keine verändernde Operationen möglich (z. B: nur final Variabeln und keine Setter)
- Instanzvariabeln sind alle `final`
  - Primitive Datentypen

```
class Configuration {
  // Instanzvariabeln müssen final sein
  private final String server;
  private final int version;

  public Configuration(String server, int version) {
    this.server = server; this.version = version;
  }

  // Neues Objekt zurückgeben
  public Configuration adjust(int newVersion) {
```

```
    return new Configuration(server, newVersion)
  }

  // Read-Only
  public String getServer() { return server; }
  public int getVersion() { return version; }

  // equals and hashCode based on content
}
```

4.1.3 Confinement (Kapselung)

- Thread Confinement: Objekte nur über Referenzen von einem einzigen Thread erreichbar.
- Object Confinement: Objekt in anderem bereits synchronisieren Objekte eingekapselt (Achtung Kapselungsbruch: Falsch wäre, wenn eine Referenz zurückgegeben wird, welche die Einkapselung kaputt macht.)
- Objekte, welche in einem Thread erzeugt werden, sind nur von diesem Thread erreichbar.

4.1.4 Thread Safe

- Klassen / Methoden, welche intern synchronisiert sind (Keine Low-Level Data Races innerhalb dieses Codes).
- Semantisch höhere Race Conditions bleiben aber möglich
- Keine durchgängige Definition
- Moderne Colections sind nicht Thread-Save
- Auch wenn Thread-Save können ausserhalb Synchronisationsfehler passieren.

4.1.5 Synchronized Wrappers

- Wrapper einer Collection, der alle Methoden synchronisiert
- Elemente werden dadurch nicht synchronisiert
- Beispiel: `List list = Collections.synchronizedList(new ArrayList())`

4.1.6 Concurrent Collections

- Effiziente Thread-sichere Collections (`java.util.concurrent`‘).
- Schwachkonsistente Iteratoren (keine Exceptions, nebenläufige Updates bei Iteration)

4.1.7 Iterieren mit Nebenläufigkeit

- Iteration einer synchronisierten Collection ist nicht als Ganzes synchronisiert (Nur Einzelzugriffe)
- Anderer Thread kann Collection parallel ändern (Semantisch höhere Race Condition, welche eventuell sogar eine Exception wirft)

Client-Side Locking:

```
Collection<T> c = Collections.synchronizedCollection(myCollection);
...
synchronized(c) {
  for (T element : c) {
    // use element
  }
```

```
}

Map<String, T> m = Collections.synchronizedMap(myHashMap);
...

Set<String> s = m.keySet(); // Collection View
synchronized(m) { // Muss m locken, nicht s!
  for (String key : s) {
    // use key
  }
}
```

4.2 Deadlocks

- Deadlocks: Gegenseitiges Aussperren von Threads.
- Livelock: Verbreauch der CPU während Deadlock.
- Betriebsmittelgraph
  - Thread T wartet auf Lock von Ressource R: $T \rightarrow R$.
  - Thread T besitzt Lock auf Ressource R: $R \rightarrow T$.
- Alle vier Bedingungen müssen zutreffen
  - Gegenseitiger Ausschluss (Locks)
  - Geschachtelte Locks
  - Zyklische Warteabhängigkeiten
  - Sperren ohne Timeout/Abbruch

4.2.1 Vermeidung

- Lineare Ordnung der Ressourcen einführen (nur geschachtelt in aufsteigender Reihenfolge sperren.) Beispiel: Konten nur nach aufsteigender Nummer sperren: Kein Zyklus im Betriebsmittelgraph möglich.
- Grobgranulare Locks wählen (Wenn Ordnung nicht möglich/sinnvoll). Beispiel: Gesamte Bank sperren

4.3 Starvation

- Starvation: Ein Thread kriegt nie die Chance, eine Ressource zuzugreifen.
  - Obwohl Ressource immer wieder frei wird (kein Deadlock oder Livelock).
  - Andere Threads können ihn dauernd überholen und Ressource wegschnappen.

4.3.1 Vermeidung

- Faire Synchronisationskonstrukte (länger wartende Threads haben Vortritt, Fairness einschalten).
- Java Monitor ist bei vielen Threads Starvation-Anfällig

4.3.2 Prioritäten & Starvation

- Priorität setzen: `myThread.setPriority(priority)`
  - 1: MIN_PRIORITY, 5: NORM_PRIORITY, 10: HIGH_PRIORITY
- Scheduling kann vom OS abhängig sein.
- Priority Inversion: Hoch prioritärer Thread wartet auf Bedingung von tief prioritärem Thread
  - Normale Threads können dazwischenlaufen
  - Tief und hoch prioritärer Thread verhungern

4.3.3 Beispiel

Dining Philosophers:

Nimmt linke Gabel und versucht Rechte zu nehmen. Wenn diese bereits besetzt ist, legt er die Linke wieder hin und probiert es erneut. (Wenn Prio zu niedrig verhungert Thread / Philosoph).

Code Beispiel:

```
do {
    boolean success = other.withdraw(100);
} while (!success);

mine.deposit(100);
```

5 Thread Pools, Asynchronität

5.1 Thread Pools Idee

- Task Queue: Potentielle Parallele Arbeitspakete (Task) werden in Warteschlange eingereiht.
- Thread Pool: Beschränkte Anzahl von Worker-Threads holen Tasks aus der Warteschlange und führen sie aus.
- Anzahl von Threads kann eingeschränkt werden (`Anz. Worker Threads = Anz. Prozessoren + Anz. I/O-Aufrufe`) (Während warten bei einem I/O können andere Threads arbeiten.
- Höhere Abstraktion: Was parallel ist, aber nicht wie.
- Free Lunch: Programme laufen automatisch schneller auf parallelen Maschinen.
- Einschränkung: Tasks im Thread-Pool dürfen nicht aufeinander warten (Deadlockgefahr); Keine gegenseitige Abhängigkeiten. (Ausnahme bei sauber geschachtelten Subtasks, da diese auf einem Stack ausgeführt werden können).
- Task muss bis zum Ende laufen, bevor Worker Thread beliebig anderen Task ausführen kann.

5.2 Implementierung

→ Task (Aufgabe, welche vorher einem Thread war):

```
// Thread-Pool-Auftrag mit Integer Resultat
// Ohne Rückgabetyp wird Runnable implementiert
class ComplexCalculation1 implements Callable<Integer> {
  @Override
  public Integer call() throws Exception {
    int value = ...; // long calculation
    return value;
  }
}
```

→ Thread Pool: Ausführung der Tasks: (Task in Queue einreihen; Handle für Abfrage als Resultat)

```
// Erzeuge Thread-Pool mit 2 Worker Threads
ExecutorService threadPool = Executors.newFixedThreadPool(2);

Future<Integer> future1, future2; // Handle auf Task Resultat

// Schicke Task an Thread-Pool
future1 = threadPool.submit(new ComplexCalculation1());
future2 = threadPool.submit(new ComplexCalculation1());

// Warte auf Task Ende und hole Resultat (oder auch Exceptions)
Integer result1 = future1.get(); // Quasi Join (warte, bis Thread fertig ist)
Integer result2 = future2.get(); // Blockiert, bis Task beendet ist

threadPool.shutdown(); // Thread Pool nach Gebrauch abstellen (sonst keine Beendigung)
```

Ab Java 8 auch als Lambda möglich:

```
Future<Integer> future = threadPool.submit(() -> {
  int value = ...; // long calculation
  return value;
});
```

Task abbrechen:

```
future1.cancel(true); // Abbrechen; True: macht noch einen Interrupt
```

Thread Pool Typen (Klasse `Executors`):

- `newFixedThreadPool(int nofThreads)`: Fixe Anzahl Worker Threads
  - `Runtime.getRuntime().availableProcessors` für nofThreads
- `newCachedThreadPool()`: Automatisch Anzahl Threads erzeugen
- `newSingleThreadExecutor()`: Nur ein Worker Thread

5.3 Rekursive Tasks ab Java 7

- Kein Shutdown mehr nötig, da sie als Daemon-Threads laufen.
- Schneller, wenn Joins geschachtelt sind.
- Subtasks (mittels `fork()` kommen zuvorderst in die lokale Task-Queue rein (LIFO).
- Hierarchische Abhängigkeiten mit Untertasks ist erwünscht (keine Gefahr für Deadlocks)
- `RecursiveAction`, falls kein Rückgabetyp.

→ Task erstellen

```
class MySuperTask extends RecursiveTask<Integer> {
  @Override
  protected Integer compute() { // Task Implementierung
    MySubTask sub = new MySubTask(...);
    sub.fork(); // Starte Unter-Task
    // other work
    sub.join(); // Warte auf Beendigung
    return ...;
  }
}
```

→ Task lancierung

```
ForkJoinPool threadPool = new ForkJoinPool(); // Thread-Pool für rekursive Tasks
// ...
threadPool.invoke(new MySuperTask()); // Haupt-Task ausführen und auf Beendigung warten
```

→ Tasks Joinen (Am besten geschachtelt)

```
task1.fork(); // <-------*
task2.fork(); // <----*  |
task2.join(); // <----*  |
task1.join(); // <-------*
invokeAll(task3, task4); // Oder einfach beides zusammen
```

→ Keine überparallelisierung mittels Tresholds

```
class SumTask extends RecursiveTask<Double> {
  private double[] array; private int from, to;

  public SumTask(double[] a, int f, int t) {
    array = a; from = f; to = t;
  }

  protected Double compute() {
    if (to - from >= 1000) { // Tuning mit Schwellwert durch Programmierer
      int middle = (to + from) / 2;
      SumTask subTask1 = new SumTask (array, from, middle);
      SumTask subTask2 = new SumTask (array, middle, to);
      subTask1.fork(); subTask2.fork();
```

```
      return subTask1.join() + subTask2.join();
    } else { // Dieser Teil wird sequenziell gelöst
      return sum(array, from, to);
    }
  }
}

public class ParallelSum {
  public static double parallel_sum(double[] array, int from, int to) {
    ForkJoinTask<Double> task = new SumTask(array, from, to);
    return new ForkJoinPool().invoke(task);
  }
}
```

5.4 Asynchrone Aufrufe mittels Completion Callback

- Asynchrone Operation informiert Caller über Resultat
- Achtung: Callback wird von anderem Thread als der vom Aufrufer ausgefhrt.
  - Synchronisation ist nötig, zwischen Zugriffen des Aufrufers und der Callback-Methode.

```
// Strategie-Interface für Callback:
interface CallbackHandler<T> {
  void handleResult(T result);
}

class MyMath {
ExecutorService threadPool = ...;
// Callback als Parameter mitgeben:
void asyncLongOperation(long input, CallbackHandler<Long> callback) {
  // Callback am Schluss der asynchronen Arbeit:
  threadPool.submit(() -> {
    long result = longOperation(input);
    callback.handleResult(result);
  });
}
}
```

Ohne Lambda als anonyme innere Klasse:

```
threadPool.submit(new Runnable() {
  @Override
  public void run() {
    long result = longOperation(input);
    callback.handleResult(result);
  }
});
```

Asynchrone Aufrufe in Java 8 mit Lambda:

```
// Asynchroner Aufruf mittels Thread-Pool mit einem worker Thread
ExecutorService threadPool = Executors.newSingleThreadExecutor();
Future<Long> future = threadPool.submit(() -> longOperation()); // Task als Lambda
otherWork();
process(future.get()); // Resultat über Future
threadPool.shutdown(); // Nicht vergessen
```

5.5 Problem: Work Stealing

- Neue Tasks haben Vorrang
- Neue Tasks werden lokal eingereiht
- Starvation Gefahr

6 .NET Task Parallel Library

6.1 Threading in .NET mit C# 5

6.1.1 Threads

```
Thread myThread = new Thread(() => { // Lambda
  for (int i = 0; i < 100; i++) {
    Console.WriteLine("MyThread step {0}", i);
  }
});
myThread.Start();
// ...
myThread.Join();
```

- Keine Vererbung, nur Deletage bei Konstruktur
- Exception in einem Thread führt zu Abbruch des gesamtesn Programms
- Lambdas haben Zugriff auf umgebende Variabeln → Prädestiniert für Data Races

6.1.2 Monitor

```
class BankAccount {
  private decimal balance;
  private object syncObject = new object(); // Monitor auf Hilfsobjekt als Best Practice

  public void Withdraw(decimal amount) {
    lock(syncObject) { // Analog zu synchronized Statement
      while (amount > balance) { // Schlaufe auch notwendig
        Monitor.Wait(syncObject);
      }
      balance -= amount;
    }
  }
  public void Deposit(decimal amount) {
    lock(syncObject) {
      balance += amount;
      Monitor.PulseAll(syncObject); // Analog zu notifyAll()
    }
  }
}
```

- FIFO Warteschlange, Pulse informiert längst wartenden
- `wait()` in Schlaufe
- `PulseAll()` bei mehreren Bedingungen oder Erfüllungen mehrerer Threads (wie in Java)
- Synchronisation mit Hilfsobjekt als Best Practice

6.1.3 Synchronisationsprimitiven

- Kein Fairness-Flag, kein Lock & Condition
- ReadWriteLockSlim, Semaphoren, Mutex
- Collections nicht Thread-Save

6.2 .NET Task Parallel Library (TPL)

6.2.1 Task Parallelisierung

```
Task task = Task.Factory.StartNew(() => {
  // task implementation
});
// perform other activity
task.Wait(); // Blockiert, bis Task beendet ist
```

6.2.2 Task mit Rückgabe

```
Task<int> task = Task.Factory.StartNew(() => { // Rückgabetyp int
  int total = ... // some calculation
  return total;
});
// ...
Console.Write(task.Result); // Blockiert bis Task Ende und liefert dann Resultat
```

6.2.3 Task Parameterübergabe

Man sollte nicht direkt auf die Variabeln zugreifen, wegen Data Races!

```
Task task = Task.Factory.StartNew((obj) => { // Typ object
  int myParamValue = (int)obj;  // Cast nötig
  // task implementation using myParamValue
}, 10); // Argument übergeben
```

6.2.4 Multi Task Start & Wait

- `Task.WaitAll(taskArray);`: Ende von allen Tasks abwarten
- `Task.WaitAny(taskArray);`: Ersten beendete Task abwarten

6.2.5 Geschachtelte Tasks

Tasks können Subtasks starten und abwarten:

```
Task.Factory.StartNew(() => {
  // outer task
  Task<bool> left = Task.Factory.StartNew(() => Search(leftPart));
  Task<bool> right = Task.Factory.StartNew(() => Search(rightPart));
  bool found = left.Result || right.Result;
  // ...
});
```

6.2.6 Exceptions in Tasks

- Propagierung an Aufrufer von `Wait()` oder Result
- Abonnierung über Event `TaskScheduler.UnobservedTaskException`

6.2.7 Kooperatives Task Abbrechen

```
CancellationTokenSource source = new CancellationTokenSource();
CancellationToken target = source.Token;
Task.Factory.StartNew(() => {
  while (...) {
    // work
    target.ThrowIfCancellationRequested();
    // Task {Logik bricht sich selbst ab (OperationCancelledException)
  }
});
source.Cancel(); // Abbrech-Signal auslösen
```

6.2.8 Thread Pool

- Erzeugt neue Worker Threads
- Warteabhängigkeiten unter Tasks
- Deadlocks mit `ThreadPool.SetMaxThreads()`
- Daemon-Threads möglich
- Fire-and-Forget

```
static void Main() {
  Task.Factory.StartNew(() => {
    // some calculation
    Console.WriteLine("Task finished");
  });
}
```

6.2.9 Datenparallelität mit Barrier am Schluss

Parallel Statement Block (Unabhängige Statements, egal welche Reihenfolge):

```
Parallel.Invoke(
  () => MergeSort(l, m),
  () => MergeSort(m, r)
);
```

Unabhängige Schlaufen-Bodies (Unabhängige Schlaufen Bodies, egal welche Reihenfolge):

```
Parallel.For(0, array.Length, (i) =>
  DoComputation(array[i])
);
Parallel.ForEach(collection, (item) =>
  DoComputation(item)
);
Parallel.Foreach(files,
  f => Convert(f)
);
```

6.2.10 Effiziente parallele Aggregation

- Thread-Lokales Teilresultat
- Resultat gefunden, Schleife stoppen: `loopState.Stop();`
- Alle Tasks abbrechen: `loopState.Break();`

```
Parallel.ForEach<FileInfo, long>(files,
  // Initialisierung pro Worker-Thread
  () => { subTotal = 0; },
  (f, _, subTotal) => {
    // Teilresultat pro Worker- Thread berechnen
    subTotal += CountWords(f);
    return subTotal;
  },
  // Teilresultat aller Worker- Threads aggregieren
  (subTotal) => {
    lock(someLockObj) {
      totalWords += subTotal;
    }
  }
);
```

6.2.11 Parallel Loop

Parallel Loop:

```
Parallel.For(0, N, (i) => {
  for (int j = 0; j < M; j++) {
    array[i, j] = Compute(...);
  }
});
```

Parallel Loop Partitionierung:

```
Parallel.ForEach(Partitioner.Create(0, array.Length),
  (range, _) => {
  for (int i = range.Item1; i < range.Item2; i++) {
    DoCalculation(array[i]);
  }
});
```

6.2.12 Parallel LINQ (PLINQ)

- Seiteneffekte nur per Kriterium (where, select) möglich.
- Race Conditions, Deadlocks möglich.

```
var result = from foo in numbers.AsParallel() select _IsPrime(foo);
return result.ToArray();
```

7 GUI und Threading

7.1 Java Swing

7.1.1 Threads in GUIs

- GUI Frameworks erlauben nur Single-Threading (Single-Worker-Thread: Nur spezieller GUI-Thread darf auf UI Komponente zugreifen)
- GUI Thread macht Loop zur Ausführung der Ereignisse aus einer Event Queue
- GUI Thread: `java.awt.EventDispatchThread`
- Events in Eventquele: `java.awt.EventQueue`
- Somit keine langen Operationen in UI Events, da sonst das UI blockiert
- Kein Zugriff auf UI Komponenten duch fremde Threads (sont Race Conditions)
- Swing ist nicht Thread-safe
  - Thread-Safe sind `repaint()`, `revalidate()`, `addListener()` und `removeListener()`

7.1.2 Dispatching an UI Thread

- UI Zugriffe an Event Dispatch Thread delegieren
- Event Dispatch Thread wird bei `setVisible()` und `pack()` erzeugt
- Benutzung der Klasse `SwingUtilities`
  - `static void invokeLater(Runnable doRun)`: Führt `run()`-Methode des Runnable -Objektes im Event Dispatch Thread aus
  - `static void invokeAndWait(Runnable doRun)`: Wartet zusätzlich, bis Ausführung fertig ist
- Statt `Thread.start()`, auch ein Task möglich.

```
class MyButtonActionListener implements ActionListener { // UI Thread
  @Override
  public void actionPerformed(ActionEvent arg) { // UI Thread
    new Thread(() -> { // UI Thread erzeugt neuen Thread
      String text = readHugeFile(); // Neuer Thread
      SwingUtilities.invokeLater(() -> { // Neuer UI Thread
        textArea.setText(text); // UI Thread
      });
    }).start();
  }
}
```

7.1.3 Potentielle Race Condition

setVisible() darf nicht mehr nach pack() aufgerufen werden, da das GUI dann schon existiert und nichts mehr am Thread geändert werden darf.

```
f.pack();
f.setVisible(true);
```

Lösung:

```
SwingUtilities.invokeLater(() -> { // Jetzt beides als ein einziger Event eingereiht
  frame.pack();
  frame.setVisible(true);
});
```

7.1.4 Swing Background Worker (Hilfsklasse)

- Zeitaufwändige Operationen in separatem Thread: `doInBackground()`
- UI-Zugriffe durch EventDispatchThread: `done()`

```
public abstract class SwingWorker<Result, Temp> {
  protected abstract Result doInBackground(); // läuft in separatem Thread (keine UI-Zugriffe)
  protected void done(); // UI-Thread
}
```

Beispiel:

```
// Integer: Resultat von Background
// Void: Keiine Zwischenresultate
class BackgroundCalculator extends SwingWorker<Integer, Void> {
  @Override
  public Integer doInBackground() { // Background Worker Thread
    return longComputation();
  }
  @Override
  protected void done() {
    try {
      Integer result = get(); // Resultat von doInBackground()
      label.setText("Result: " + result);
    } catch (InterruptedException | ExecutionException e) {
      e.printStackTrace(); // UI-Thread
    }
  }
  // ...
}
```

- Ausführen mittels `new BackgroundCalculator().execute();`

7.2 .NET GUI Threading Model

7.2.1 Asynchronität mit Async/Await

- Rückgabetypen: `void`: Fire-and-Forget, `Task`: Keine Rückgabe, aber erlaubt Warten auf Ende, `Task<T>`: Für Methode mit Rückgabetype `T`
- `async`-Methode muss ein `async` enthalten (sonst wäre sie gar nicht asynchron)
- Zwei Stücke: Zuerst synchron, danach asynchron

Beispiel:

```
async Task<string> ConcatWebSitesAsync(string url1, string url2) {
  HttpClient client = new HttpClient();
  Task<string> download1 = client.GetStringAsync(url1);
  Task<string> download2 = client.GetStringAsync(url2);
  string site1 = await download1;
  string site2 = await download2;
  return site1 + site2;
} // Async Methode

// ...

Task<string> task = ConcatWebSitesAsync(s1, s2);
OtherWork();
```

```
int result = await task; // Warte auf Beendigung der Async Methode
//
```

8 Memory Models

8.1 Alternativen zu Synchronisation

- Vermeide Synchronisation: Confinement, Immutability
- Low-Level Cache Konsistenz: Speichergarantien ohne Locks
- Vermeide Kontextwechsel: Spin Waits
- Einsatz von Concurrent Collections: Für massive Nebenläufigkeit optimiert

8.2 Ursachen für Probleme

- Weak Consistenc: Zugriff können in verschiedenen Reihenfolgen geschehen.
- Compiler, Laufzeitsystem und CPUs ordnen die Reihenfolge der Instruktionen.
- Compiler/Laufzeitsystem kann Variabeln in Register ablegen oder unnötige Zuweisungen wegoptimieren.

8.3 Java Memory Model

- Atomicy: Unteilbarkeit von Lese- und Schreibzugriffe auf Speicher.
- Visibility: Sichtbarkeit aktueller Speicherwerte zwischen Threads.
- Ordering: Reihenfolge von Lese-/Schreiboperationen auf Speicher.
- JVM könnte stärkere Garantien implementieren, man sollte sich aber nicht drauf verlassen.

8.3.1 Atomicy

- Atomicity: (Unteilbarkeit) von Lese- und Schreibzugriffe auf Speicher
- Bei Primitive Datentypen falls bis 32 Bit
- Bei Objektreferenzen
- Bei `volatile` Keyword

```
int i = 1; // Nicht atomar, da in zwei Anweisungen unterteilt (default = 0)
s = "second"; // Atomar, da nur Zuweisung der Referenz
```

8.3.2 Visibility

- Visibility (Sichtbarkeit): aktueller Speicherwerte zwischen Threads
- Änderungen eines anderen Threads wird evtl. erst später gesehen
- Verursacht durch Compiler-Optimierung (Hält Variabelnwerte evtl. in Register)
- Java visibility Garantien:
  - Locks Release & Acquire: SchreibenderThread ruft Release und lesender thread ruft Acquire für denselben Lock auf
  - Volatile Variabeln: lesen und schreiben
  - Initialisierung von `final` Variabeln (nach Ende des Konstruktors)
  - Thread-Start und Join, Task Start und Ende
- Alle Änderungen vor dem Unlock/Release werden für jeden sichtbar, der danach Lock/Acquire auf demselben Objekt bezieht.
- alle Änderungen vor dem volatile Zugriff werdenfür jeden sichtbar, der danach auf dieselbe volatile Variable zugreift.

8.3.3 Ordering

- Ordering (Reihenfolge) von Lese-/Schreibeoperationen auf Speicher
- Java Ordering Garantien:
  - Innerhalb eines Threads: Umsortieren erlaubt
  - Zwischen Threads: Reihenfolge nur erhalten für Synchronisationsbefehle (synchronized, Lock Acquire/Release, volatile Variabeln)
  - Nicht-volatile Zugriffe werden nicht über Grenzen von Synchronisation oder volatile Zugriffe optimiert

8.4 Java `volatile` Keyword

- Atomicity: Atomares Lesen und schreiben (aber nicht i++)
- Visibility: Lese und Schreibzugriffe via Hauptspeicher propagiert
- Reordering: Keine Umordnung durch Compiler / Laufzeitsystem
- Verhindert Data Race auf Variable

8.5 Java Atomic Klassen

- Kein Blockieren oder Warten auf Locks (komplexer als nur atomares Lesen und Schreiben)
- Java Atomic Variables: Atomares Exchange, Test-And-Set und Inkrementieren
- Effiziente Implementierung (benutzt atomare Instruktionen von Maschine)
- Garantiert auch Visibility und Ordering
- `AtomicBoolean`:
  - `getAndSet(boolean newValue)`: Liefert alten Wert und setzt den neuen (atomar)
  - `compareAndSet(boolean expect, boolean update)`: Liefert alten Wert und setzt den neuen (atomar)
- `AtomicInteger`:
  - `int addAndGet(int delta)`: `current += delta; return current`
  - `int getAndAdd(int delta)`: `old = current; current += delta; return old`
  - `int decrementAndGet()`: `return --current`
  - `int getAndDecrement()`: `return currentValue--`
  - `int incrementAndGet()`: `return ++currentValue`
  - `int getAndIncrement()`: `return currentValue++`
- `AtomicLong`
- `AtomicReference<V>`
- `AtomicIntegerArray`, `AtomicLongArray`, `AtomicReferenceArray<V>`
- `ConcurrentLinkedQueue<V>`, `ConcurrentLinkedDeque<V>`, `ConcurrentSkipListSet<V>`, `ConcurrentHashMap<K, V>`, `ConcurrentSkipListMap<K, V>`

Die Atomic Klassen machen eine optimistische Synchronisation:

- Datenstrukturen ohne Locks nur mit atomaren Operationen implementieren
- Es gibt schon Lockfreie Datenstrukturen (`ConcurrentLinkedQueue<V>`)
- Optimistische Synchronisation: Schreibe Änderungen nur, wenn kein anderer Thread zwischenzeltich geschrieben hat. (Wiederholung bei Fehlschlag → Starvation möglich).
- ABA-Problem: Ein anderen Thread überschreibt unbemerkt dazwischen dasselbe Resultat (muss nicht immer ein Problem sein, je nach Anwendung).

8.6 NET Memory Model

- `long`/`double` auch ohne `volatile` atomar
- Visibility: Nicht definiert, sondern durch Ordering gegeben
- Ordering: Volatile ist nur partielle Fence
- Atomare Instruktion: `Interlocked` Klasse (Atomares Exchange, Add, Increment, CompareExchange)
- Volatile Read: Bleibt vor den nachfolgende Zugriffen (umordnen nicht möglich)
- Volatile Write: Bleibt nach den vorherigen Zugriffen (umordnen nicht möglich)
- Full Fences: `Thread.MemoryBarrier();`

9 GPU Parallelisierung

9.1 GPU Co-Prozessor Architektur

- GPU ist Co-Prozessor zur CPU
- Eine GPU besteht aus mehreren (z. B. 1-30) Streaming Multiprocessors (SM), welche aus mehreren (z. B. 8-192) Streaming Processors (SP) besteht.
- Single Instruction Multiple Data (SIMD): Vektorparallele Ausführung: Alle Cores innerhalb SM müssen zur gleichen Zeit die gleiche Instruktion ausführen, aber auf verschiedenen Daten. (Voneinander abhängige Daten oder sequenzielle Arbeiten sind nicht möglich).
- SIMT: Single Instruction Multiple Threads: Dieselbe Operationen werden gleichzeitig in verschiedenen Threads aufgerufen, aber auf verschiedenen Daten.
- GPU: Hohe Datenparallelität, hohe Memorybandbreite, viele einfache Cores, kleine Caches pro Core
- Non-Uniform Memory Access (NUMA): Kein gemeinsamer Hauptspeicher zwischen GPU und CPU. Daten müssen explizit transportiert ewrden.
- Computer Unified Device Architecture (CUDA) ist ein Programmiermodell und eine API für C.

9.2 Kernel Definition und Launch

```
// GPU (Device)
__global__
void VectorAddKernel(float *A, float *B, float *C) {
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

// CPU (Host)
int main() {
  // ...
  // kernel invocation
  VectorAddKernel<<<1, N>>>(A, B, C);
  // ...
}
```

9.3 Grid, Block, Thread Aufteilung

- Ein Grid hat mehrere Blöcke, welcher mehrere Threads hat.
- Ein Block ist immer im selben Streaming Multiprozessor.
- Threads können innerhalb eines Blocks interagieren.
- Thread = Virtueller Skalarprozessor
- Block = virtueller Multiprozessor
- Einzelne Blöcke müssen unabhängig sein, können nicht auf andere warten.
- Jeder Kernel läuft in einem eigenen Grid.
- Blocksize: Anzahl Threads per Block

Jeder Kernel hat eigenen Datenteil:

- `threadIdx.x`: Nummer des Threads innerhalb Block
- `blockIdx.x`: Nummer des Blocks
- `blockDim.x`: Blockgrösse
- Weitere Dimensionen `y` und `z` nutzbar

9.4 Grundgerüst für CUDA

Aufteilung in Blöcke (Beispiel Vektoraddition):

```
__global__
void VectorAddKernel(float *A, float *B, float *C, int N) {
  // Eindeutiger Index basierend auf (Block ID, Thread ID)
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) { // Überflüssige Threads machen nichts
    C[i] = A[i] + B[i];
  }
}

// Kernel invocation: 4 Blöcke * 512 Threads = 2048 Elemente
VectorAddKernel<<<4, 512>>>(A, B, C, 2048);
```

Grundgerüst:

```
void CudaVectorAdd(float* A, float* B, float* C, int N) {
  size_t size = N * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc(&d_A, size);
  cudaMalloc(&d_B, size);
  cudaMalloc(&d_C, size);

  cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

  int blockDim = 512;
  int gridDim = (N + blockDim - 1) / blockDim;
  VectorAddKernel<<<gridDim, blockDim>>>(d_A, d_B, d_C, N);

  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Verbessert mit Fehlerbehandlung:

```
handleCudaError(cudaMalloc(&d_A, size));
handleCudaError(cudaMalloc(&d_B, size));
handleCudaError(cudaMalloc(&d_C, size));

handleCudaError(cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice));
handleCudaError(cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice));

int blockDim = 512, gridDim = (N + blockDim - 1) / blockDim;
VectorAddKernel<<<gridDim, blockDim>>>(d_A, d_B, d_C, N);
handleCudaError(cudaGetLastError());

handleCudaError(cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost));

handleCudaError(cudaFree(d_A));
handleCudaError(cudaFree(d_B));
handleCudaError(cudaFree(d_C));

void handleCudaError(cudaError error) {
  if (error != cudaSuccess) {
```

```
    fprintf(stderr, "CUDA: %s!\n",
      cudaGetErrorString(error));
    exit(EXIT_FAILURE);
  }
}
```

9.4.1 Launch Configuration

- Blockgrösse als Vielfaches von 32; Sonst sehr ineffizient
- Überflüssige Threads vermeiden; 2 Blöcke à 1024 => 548 unnütze Threads
- Maximale Anzahl Threads per Block; Abhängig von GPU, z.B. 512 oder 1024
- Streaming Multiprocessor ausschöpfen; Limite für Resident Blöcke und Threads, z.B. 8 und 1536
- Grosse Blockgrösse hat Vorteile; Threads können nur in Block interagieren

Beispiel:

- Maximale Anzahl Threads per Block = 512
- Maximale Anzahl Resident Blocks = 8
- Maximale Anzahl Resident Threads = 1536
- → 3 Blöcke, 512 Threads pro Block = Nur 36 unnütze Blöcke

9.5 Mehrere Dimensionen

- Limitation: C kennt keine mehrdimensionale Arrays:

```
// Problem/Gewünscht:
float[,] matrix = new float[NofRows, NofCols];
matrix[row, col] = ...

// Lösung: Low-Level Initialisierung:
float *matrix = (float *)malloc(NofRows * NofCols * sizeof(float));
matrix[row * NofCols + col] = ...
```

- Jeder Block wird dreidimensional adressiert (Würfel)
- Jeder Block kann wieder in weitere Würfel unterteilt werden

```
dim3 gridDim(3, 2, 1);
dim3 blockDim(4, 3, 1);
Call<<<gridDim, blockDim>>>(...);
```

Index berechnen mit mehrere Dimensionen:

```
id = threadIdx.x + blockIdx.x * blockDim.x
  + (threadIdx.y + blockIdx.y * blockDimx.y) * line_length
```

Kernel für Matrixmultiplikation (2D Modell):

```
__global__ // Kernel
void multiply(float *A, float *B, float *C) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (i < N && j < M) {
    float sum = 0;
    for (int k = 0; k < K; k++) {
      sum += A[i * K + k] * B[k * M + j];
    }
```

```
    C[i * M + j] = sum;
  }
}
```

9.6 CUDA-Optimierungstechniken

9.6.1 Speichermodell

- Global Memory ist teuer: ca. 600 Zyklen
- Threads lesen wiederholt selbe Elemente von A und B
- Shared Memory
  - Per Streaming Multiprocessor
  - Sehr schnell
  - nur zwischen Threads innerhalb Block sichtbar
  - Paar KB
  - `__shared__ float x;`
- Global Memory
  - Per GPU
  - Langsam
  - in allen Threads sichtbar
  - Mehrere GB
  - `cuMalloc()` oder `__device__ float x;`
- Cache/Schnellen Speicher "Shared Memory" explizit gemeinsam nutzen

9.6.2 Synchronisation

- Idee: Zwischenresultate (z. B. Spalten und Zeilen bei einer Matrixmultiplikation) in Shared Memory Laden.
- Threads synchronisieren mittels `__syncthreads()`
- Jedes `__syncthreads()` Statement ist eine andere Barriere
- In If-Else nur erlaubt, falls alle Threads eines Blocks nur das genau selbe `__syncthreads` nutzen

```
// Deklaration des Shared memory, statische Array-Grösse nötig
__shared__ float Asub[TILE_SIZE][TILE_SIZE];
__shared__ float Bsub[TILE_SIZE][TILE_SIZE];

int tx = threadIdx.x, ty = threadIdx.y;
int col = blockIdx.x * TILE_SIZE + tx;
int row = blockIdx.y * TILE_SIZE + ty;

// Matrix Multiplikation mit Shared Memory
for (int tile = 0; tile < nofTiles; tile++) {
  Asub[ty][tx] = A[row * K + tile * TILE_SIZE + tx];
  Bsub[ty][tx] = B[(tile * TILE_SIZE + ty) * M + col];
  __syncthreads();
  // Multipliziere Zeile von A-Tile mit
  // Spalte von B-Tile aus dem Shared Memory
  __syncthreads();
}
```

9.6.3 Warps

- Block wird intern in Warps zerlegt ($\rightarrow$ Jedes Warp hat 32 Threads).
- Alle Threads in Warp führen gleiche Instruktion aus.
- SMP kann alle Warps eines Blocks beherbergen, da alle dasselbe shared Memory sehen und alle müssen die Barriere erreichen (es laufen aber nicht alle parallel).
- Ein Warp läuft immer auf einem Streaming Prozessor (deshalb SIMD)
- Falls ein Warp auf Speicher wartet, führt SP nächsten Warp aus.

9.6.4 Divergenz

- Unterschiedliche Verzweigungen im selben Warp.
- SM führt Instruktion der einen Verzweigung durch und die anderen Threads müssen warten. In der nächsten Verzweigung warten dann die anderen Threads.
- Performance Problem

Deshalb bessere Verzweigungen machen:

```
if (threadIdx.x / 32 > 1) {
  // foo
} else {
  // bar
}
```

9.6.5 Coalescing

- Zugriffsmuster der Threads sind entscheidend.
- Aufeinanderfolgender Zugriff von Threads auf Daten.
- In einer Transaktion auf mehrere Daten zugreifen (Memory Burst)
- Threads in einem Warp greifen auf alle Elemente einer Burst Section zu
- Ohne Coalescing gibt es mehrere Einzelzugriffe.

9.6.6 Performance Empfehlungen

- Datenaustausch zwischen CPU/GPU minimieren
- Divergenz innerhalb Warp vermeiden
- Global Memory Zugriffe reduzieren (Shared Mem)
- Unnütze Threads pro Block minimieren
- Maximale Blockgrösse möglichst ausschöpfen
- Wenig lokale Variablen (=Register) pro Thread

10 Actors Model

10.1 Motivation

- Herkömmliche Programmiersprachen sind nicht für Nebenläufigkeit designed.
- Korrekte nebenläufige Programme zu schreiben ist daher besonders schwierig!
- Fehleranfällig: Race Conditions
- Schlecht verteilbar: Shared Memory Modell, gemeinsamer Speicher von Threads
- Threads operieren auf passiven Objekten

10.2 Actor Modell und CSP

- Anderes Programmierkonzept: Aktive Objekte: Objekte haben nebenläufiges Innenleben und können miteinander kommunizieren.
- Kommunikation zwischen Objekten geschieht über austausch von Nachrichten
- Objekte senden und Empfangen auf einem Kanal.
- Kein Shared Memory: Austausch von Nachrichten über Kanäle/Mailboxen
- Implementierungen: Erlang Java Communication Sequencial Process, .NET

10.3 Actors

- Aktor umfasst: Processing, Storage, Communication
- Aktor kann
  - Neue Actors erstellen
  - Nachrichten an Actors senden
  - Entscheiden wie die nächste Nachricht behandelt werden soll

10.4 Vorteile vom Actor Modell

- Nebenläufigkeit
  - Alle Actors (Objekte) laufen nebenläufig.
  - Maschine kann grad an Nebenläufigkeit ausnutzen.
- Keine Race Conditions
  - Da kein Shared Memory.
  - Nachrichtenaustausch zur Synchronisation.
- Gute Verteilbarkeit
  - Da kein Shared Memory
  - Nachrichtenaustausch für Netz prädestiniert

10.5 Communicating Sequential Processes (CSP)

- von Sir C.A.R. Hoare
- Nur ein Modell, keine Programmiersprache
- Ähnlich zu Actors, nur blockierendes Senden möglich
- Prozesse kommunizieren über Kanäle
- Nachrichtenaustausch erfolgt sofort und synchron

Ablauf

- Aktives Objekt P sendet Naricht E an Kanal C
- Aktives Objekt Q Empfange Naricht E von Kanal c

Beispiel des Modells (nur zur Veranschaulichung)

```
actor A channel c {
  produce x;
  c!(x);
}
actor B channel c {
  c?(x);
  consume x;
}
```

10.6 Actors mit Akka

- Aktive Objekte mit privatem Zustand.
- Akka (für JVM und weitere), Meta-Programmiermodell auf normaler Sprache.
- Jeder Actor hat eine Mailbox
  - Senden ist immer asynchron.
  - Ein Buffer für alle Nachrichten.
- Empfangen
  - Spezielle Methode wird ausgefhrt.
  - Nur eine Nachricht pro Client auf einmal bedienbar.
  - Effekte: Privater Zustand ändern, Nachrichten Senden, neue Actors erzeugen.

Actor als Empfänger:

```
public class NumberPrinter extends UntypedActor {
  public void onReceive(final Object message) {
    if (message instanceof Integer) {
      System.out.print(message);
    }
  }
}
```

Senden an den Actor:

```
ActorSystem system = ActorSystem.create("System");

// Spezielle Referenz, Erzeugung per Reflection
ActorRef printer = system.actorOf(Props.create(NumberPrinter.class));

// Einfaches asynchrones Senden
for (int i = 0; i < 100; i++) {
  printer.tell(i, ActorRef.noSender()); // Einfaches asynchrones Senden
}

system.shutdown(); // Gebe "End-Signal" an alle Actors
```

10.7 Actor Referenzen

- Verhindert Methodenaufrufe / Variablenzugriffe
- Kommunikation über eine Art Kanal
- Beispiele für Actors

  - Alternative zu Threads
  - Transaction-Processing
  - Backend für Service
  - Kommunikations-Hub
- Patterns für das Senden und Empfangen
  - Pattern: Producer - Consumer
  - Pattern: Pipeline
  - Pattern: Map-Reduce: Aufgabe an Worker-Actors verteilen

10.8 Verteilung

- Senden und Empfangen von serialisierten immutable Nachrichten

Server:

```
ActorSystem system = ActorSystem.create("System"); //
ActorRef printer = system.actorOf(Props.create(...), "printer");
```

Config:

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "server"
      port = 2552
    }
  }
}
```

Client:

```
ActorSystem system = ActorSystem.create("producer"); // ActorSelection, nichtActorRef
ActorSelection printer = system.actorSelection("akka.tcp://consumer@server:2552/user/printer");
printer.tell(123, ActorRef.noSender());
```

10.9 Beispiel Ping Pong

```
public class PingPong {

  static final FiniteDuration FIVE_SECONDS = Duration.create(5, TimeUnit.SECONDS);

  public static void main(String[] args) {
    ActorSystem system = ActorSystem.create("PingPong");

    ActorRef p1 = system.actorOf(Props.create(PingPongActor.class), "P1");
    ActorRef p2 = system.actorOf(Props.create(PingPongActor.class), "P2");

    p1.tell(new Start(), p2);

    // Aufgabe c) Sending a reset message
```

```
    system.scheduler().schedule(FIVE_SECONDS, FIVE_SECONDS, p1, new Reset(),
      system.dispatcher(), ActorRef.noSender());
  }

  static class Start { }
  static class Reset { }
  static class Ping {
    final int count;
    public Ping(int count) { this.count = count; }
  }

  static class PingPongActor extends UntypedActor {
    private boolean reset = false;

    public void onReceive(Object message) {
      if (message instanceof Start) { handleStart((Start) message); }
      else if (message instanceof Ping) { handlePing((Ping) message); }
      else if (message instanceof Reset) { handleReset(); }
      else { unhandled(message); }
    }

    private void handleReset() { reset = true; }

    private void handlePing(Ping msg) {
      System.out.println(getSelf().path().name() + ": Ping " + msg.count);
      try {
        Thread.sleep((long) (Math.random() * 1000) + 300);
      } catch (InterruptedException e) { }

      // Aufgabe b
      if(reset) {
        reset = false;
        getSender().tell(new Ping(0), getSelf()); }
      else if (msg.count == 10) { getContext().system().shutdown(); }
      else { getSender().tell(new Ping(msg.count + 1), getSelf()); }
    }

    private void handleStart(Start message) {
      System.out.println("Starting ...");
      getSender().tell(new Ping(0), getSelf());
    }
  }
}
```

10.10 Remoting

- `system.actorSelection` mit URL.
- ActorSelection: Leichtgewichtiger als ActorRef.
- Remote Erzeugen: `system.actorOf(...)`, `application.conf` spezifiziert, wo Actor erstellt wird.

10.11 Hierarchien

- Passend zu URL Adressierungsschema
- Erzeuger is Parent

- `ActorSelection` selektiert Teilbaum, auch Broadcast möglich (`/foo/bar/*`)

Akka Sender

```
tell(msg, sender) // Sender mitgeben, zB getSelf() oder getSender() bei Forward
```

Akka Synchrones Senden (Empfänger muss antworten, sonst Timeout)

```
Future<Object> result = Patterns.ask(actorRef, msg, timeout); // Antwort ist Untyped
```

10.12 Messages

- Serializable Classes
- Immutable (Value Objects), Attribute Final, Collections in `Collections.unmodifiableList` wrappen, keine Methoden mit Seiteneffekten
- Viel Schreibaufwand für Message-Klassen → einfacher mit Scala-API

10.13 Akka Laufzeitsysten

- Dispatches zur Ausführung
- Typischerweise ein Java Fork-Join Thread Pool
- Nicht ein Thread pro Actor!
- Warteabhängigkeit zwischen Actors:
- Synchrones Send & Receive daher nicht empfohlen!

10.13.1 Akka Supervision

- Andere Actors überwachen (z. B. nach Execptions)
- Bei Exeptions wird der Supervisor benachrichtigt
- Parents überwachen standarmässig ihre Kinder
- Supervisor hat verschiedene Möglichkeiten: Resume, Restart, Stop, Escalate
- Parent von `/usr` ist der Root Guardian
- Zusätzlicher `/system` Actor für Logging, Shutdown

10.14 Shutdown

- Wenn Maiblox leer? Actor könnte noch beschäftigt sein.
- Applikation muss Actor selber stoppen.

```
getContext().stop(actorRef); // Stoppt nach Bearbeitung der akuellen Massage
getContext().stop(getSelf()); // immer Rekursiv
getContext().system().shutdown();
actor.tell(PoisonPill.getInstance(), sender); // Stoppt bei Behandlung der Poison Pill
victim.tell(Kill.getInstance(), sender); // Startet Supervision Behandlung!
```

10.15 Schwächen

- Kein richtiges Protokoll (nur implizit vorhanden: jeder nimmt alle entgegen, ich sehe nicht wer was wohin schickt)
- Keine Typsicherheit
- Diskrepanz JVM und Actor Model

11 Cluster Parallelisierung

11.1 High-Performance Computing (HPC) Cluster Architektur

- Cluster: Verbund leistungsfähiger Rechenknoten.
- Compute Nodes hinter Head Node, mit welchem der Client kommuniziert.

11.2 Jobs und Tasks, Input / Output

- HPC Job: Zusammengehörige Ausführung im Cluster; vom Client lanciert; 1 oder mehrere Tasks
- HPC Task: Ausführung eines Executables, operiert auf Files i Fileshare des Clusters, Abhängigkeiten zwischen mehreren Tasks möglich
- Job/Task Modell ist zu Low-Level: Batch Commands und nur Files als Austausch
  - Kein Shared Memory zwischen Nodes
  - Aber Shared Memory für Cores innerhalb Node.
- Ein Nachrichtenaustausch ist nötig

11.3 MPI (Message Passing Interface)

- Verteiltes Programmiermodell, Prinzip von Actor Model
- Industriestandard einer Library (Aktuell 3.0)
- MPI Programm wird in mehrere Prozesse gestartet mit eindeutiger ID
- Jeder Prozess hat ein unabhängiger Adressraum
- Prozesse starten und terminieren synchron
- Kommunikation untereinander möglich (Senden/Empfangen, Synchronisation mit Barrieren)

Beispiel in C:

```c
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]){
  MPI_Init(&argc, &argv); // MPI Initialisierung

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Prozess Identifikation

  printf("MPI process %i", rank);

  MPI_Finalize(); // MPI Finalisierung
}
```

Beispiel in C# .NET

```c#
using MPI;
using System;
class Program {
  public static void Main(string[] args) {
    using (new MPI.Environment(ref args)) { // Jede Prozessinstanz
    int rank = Communicator.world.Rank;
    Console.WriteLine("MPI process {0} ", rank);
    }
  }
}
```

Ausführung (Anzahl Prozesse und CPUs angebbar):

```
mpiexec {n 16 FirstMpiProgram.exe
```

11.4 Single Programm Multiple Data (SPMD)

- MPI Programm wird in mehrere Prozesse gestartet
  - Jeder Prozess hat eine eigene ID (Rank)
  - Prozesse arbeiten unabhängig in ihrem eigenen Adressraum (kein Shared Memory)
  - Prozesse starten und terminieren Synchron
- Prozesse können untereinander Kommunizieren
  - Senden/Empfangen von Nachrichten
  - Synchronisation mit Barrieren

11.4.1 Nachrichtenaustausch über Communicator

- Communicator: Gruppe von MPI-Prozessen
- Erlaubt Kommunikation zwischen Prozessen
- `Communicator.world`: Alle Prozesse einer MPI-Programmausführung; aber auch eigene Gruppen definierbar
- Prozess Identifikation
  - Rank (= Nummer ab 0 innerhalb einer Gruppe): `Communicator.world.Rank`
  - Eindeutige ID: Rank, Communicator: `Communicator.world.Size`

11.4.2 Direkte Kommunikation

```
var world = Communicator.world;
int rank = world.Rank; // Prozessnummer
int size = world.Size; // Anzahl Prozesse
int tag = 1;
if (rank == 0) {
  int value = new Random().Next();
  for (int to = 1; to < size; to++) {
    world.Send(value, to, tag); // Senden
  }
} else {
  int value;
  world.Receive(0, tag, out value); // Empfangen
  // out value = in value einen Wert schreiben
  Console.WriteLine("{0} received by {1}", value, rank);
}
```

11.4.3 Broadcast

`for`-Schleife fällt weg:

```
int value = 0;
if (rank == 0) {
  value = new Random().Next();
}
// Alle erhalten Wert von Prozess 0
```

```
Communicator.world.Broadcast(ref value, 0);
// 0 ist der Sender: Er sendet, alle anderen warten
Console.WriteLine("{0} by {1}", value, rank);
```

11.4.3.1 Barriere

Warte, bis alle Prozesse die Barriere erreichen:

```
Communicator.world.Barrier()
```

11.4.4 Reduktion

Aggregation von Teilresultaten zwischen Prozessen:

```
world.Allreduce(value, (a, b) => a + b)
```

- `T Allreduce(T value, Op<T>)`:
- `T Reduce(T value, Op<T>, int rank)`:

11.5 Weitere Nachrichtentypen

- All Total: Jeder sendet verschiedenen Wert an alle anderen
- Scatter: Einer verteilt verschiedene Werte an alle anderen
- Gather: Alle senden verschiedenen Wert an einen

12 Reactive Programming

- Automatische Parallelisierbarkeit ohne Programm-Redesign
- Einfache Skalierung: Grössere Datenmengen und mehr Cores
- Imperativ: Task/Daten Parallelisierung
- Deskriptiv: PLINQ, Reactive Programming
- Parallele Datenflüsse:
  - Horizontal: Parallel arbeiten
  - Vertikal: Datenmenge teilen
- Vorteile:
  - Aktive Datenflüsse statt nur passive LINQ
  - Skalierbare Parallelität durch Wahl der scheduler
  - Durchgängig asynchron
- Nachteile
  - Zerstückelung komplexer Logiken in Handler
  - Allfälliger Kontext muss durchgeschleust werden
  - Komplizierte Aggregation (Observable statt Skalar)

12.1 Datenfluss als PLINQ

- Pull-Mechanismus: Nicht Reactive
- Auswertung beginnt durch Iterieren der Abfrage
- Schritte werden rückwärts ausgelüst
- Input-Quelle ist passiv
  - Input muss vollständig vorliegen
  - Wird durch query ausgelesen und verarbeitet
- Problem
  - Geht nicht, falls Input sukzessife mit Pausen ankommt (User Input, Netzwerk) oder länge des Streams unbekannt ist.
- Push Mechanismus (Reactive)
  - Input unbekannter Länge mit Pausen
  - Asynchron

Beispiel PLINQ:

```
from entry in
  salesEurope.AsParallel()
  Union(salesAsia.AsParallel()).
  Union(salesAmerica.AsParallel())
group
  entry by entry.Article into category
  let sum = category.Sum(e => e.Volume)
where
  sum >= 1000
select
  new { category.Key, sum };
```

12.2 Reactive Programming

- Input und Arbeitsschritte sind aktiv: Lösen pro Wert ein Ereignis aus (`OnNext`).
- Verkettung der Arbeitsschritte: Nachfolgeschritt abonniert Events des Vorgängers.
- Asynchrone Events als Input verwendbar
- Ähnlich dem Observer-Pattern.

12.3 .NET Rx (Reactive Extensions)

- Vorgänger (Observable) → Nachfolger (Observer)
- `IObservable<T>`: `Subscribe(Observer<T>)`
- `IObserver<T>`: `OnNext(value: T)`, `OnError(Exception)`, `OnCompleted()`
- Pipelining möglich: Zwischenschritte haben zwei Rollen (Observer des Vorgängers, Observable des Nachfolgers)
- Subject = Observer + Observable (Beide Interfaces in einem)

Beispiel:

```
var subject = new Subject<string>();

// Observable: Registrieren
subject.Subscribe(Console.WriteLine);

// Observer: Füttern
subject.OnNext("A");
subject.OnNext("B");
subject.OnNext("C");
subject.OnCompleted(); // oder OnError()
```

12.3.1 Sequenzende

- Observer kann beliebig viele Werte erhalten
- Beliebige Verzögerung zwischen `OnNext()`
- Ende der Sequenz: Erfolgreich mit `OnCompleted()` und fehlerhaft mit `OnError()`.

12.3.2 Ad-Hoc Observer Erzeugung

```
subject.Subscribe(
  value => Console.WriteLine("{0} received", value),
  exception => Console.WriteLine("{0} thrown", exception),
  () => Console.WriteLine("completed", value)
);
```

12.3.3 Buffer Varianten

- `Subject`: Observer erhält zukünftige Werte; kein Buffer
- `ReplaySubject`: Observer erhält alle alten Werte; Unbeschränkter Buffer
- `BehaviorSubject`: Observer erhält letzten Wert; Buffer von einem Element
- `AsyncSubject`: Schickt letzten Wert bei OnCompleted; Buffer von einem Element

Beispiel ReplaySubject:

```
var subject = new ReplaySubject<string>();
subject.OnNext("A");
subject.OnNext("B");
subject.Subscribe(Console.WriteLine);
subject.OnNext("C");
```

12.3.4 Weitere Techniken

- Passive Enumerable zu aktiven Observable umwandeln: `foo.ToObservable()`
- Observables kombinieren: var combinedSales = salesEurope.ToObservable().Merge(alesAsia.ToObservable());‘
- Default ist jedoch alles sequentiell (nacheinander) aber asynchron (kein Warten)
- Concurrency ist aber einfach einstellbar: `observable.ObserveOn(TaskPoolScheduler.Default)`

12.4 Parallele Verarbeitung mit TPL

- Synchrone Ausführung (gleicher Thread, Default): `TaskPoolScheduler.Default`
- Parallele Ausführung in TPL: `NewThreadScheduler.Default`
- Alle Aufrufe dieses Observable in neuen Thread (GUI Thread): `DispatcherScheduler.Current`

```
var sales1 = salesEurope.ToObservable().ObserveOn(TaskPoolScheduler.Default);
var sales2 = salesAsia.ToObservable().ObserveOn(TaskPoolScheduler.Default);
var sales3 = salesAmerica.ToObservable().ObserveOn(TaskPoolScheduler.Default);
var combinedSales = sales1.Merge(sales2).Merge(sales3);
combinedSales.Subscibe(Console.WriteLine);
```

12.5 Mögliche Concurrency Fehler

- Race Conditions
  - Mit Seiteneffekten in Observer möglich
  - Vermeiden oder synchronisieren
- Deadlocks
  - Bei Warteabhängigkeiten in Observer
  - Blockierende Aufrufe wie `First()`, `Last()` vermeiden

12.6 Kombination mit mit LINQ

- Rx ist mit LINQ kombinierbar

13 Software Transactional Memory (STM)

- Problem Shared Memory:
  - Explizite Synchronisation gibt Deadlocks, Starvation, Kosten
  - Race Conditions bei ungenügend Synchronisation
- Idee aus der Datenbankwelt.
- Ziel von STM: Keine Race Conditions, keine Deadlocks, keine Starvation
- Atomare Sequenz von Operationen (Read/Write, wie eine grosse atomare Aktion, keine inkonsistenten Zwischenstände bemerkbar)
- ACI Transaktionen:
  - Atomicity: Vollständig oder gar nicht
  - Consistency: Programm vor und nach Trasaktion gültig
  - Isolation: Effekte wie eine seriele Ausführung (korrekt wie in der seriellen Welt), Parallel aber selbes Verhalten
- Konzept
  - Deskriptiv: Was ist atomar? (Nicht wie!)
  - Automatische Isolation: Überlasse korrekte Ausführung dem System
  - Einschränkungen: Speiherzugriffe sind isoliert, Seiteneffekte nicht
  - Implementierung: Meist optimistisches Concurrency Control (unsynchronisiert ausführen, bei Konflikt: Rollback)
- Sieht gleich aus wie Locking mit `synchronized(this){...}` (Imperativ)
- Nested Transactions = Beliebig schachtelbar: atomic Blöcke wieder in atomic Blöcke packen
  - `synchronized` nicht lockbar: Deadlocks, Niemand sieht Stand zwischen Überweisung
- Transaktionsausführung
  - Optimistisches Concurrency Control: Kann Folgefehler geben (wenn zwischendurch eine andere Transaktion z. B. eine Variable auf 0 setzt: Div by Zero)
  - Transaktionen können unerwartet abbrechen (automatisches Retry, unbemerkbar für Anwendungen)
  - Seiteneffekte bleiben sichtbar: Keine IOs/Logs/Files/
  - Starvation: Hat Transaktion immer Pech: Wird diese nie evtl. fertig werden
- Warten auf Bedingungen
- Auch auf Hardware möglich (Haswell Prozessoren), auch Nested Transaktionen
- Vorteil: Keine Deadlock-Gefahr, auch im Fehlerfall atomar

```
atomic {
  if (balance >= amount) {
    retry; // Transaktion ausdrücklich zurücksetzen, später nochmals probieren
  }
  balance -= amount;
}
```

13.1 ScalaSTM

- Implementation auf JVM: ScalaSTM
- Wrapping von Variabeln, damit es im STM System richtig verwendet wird
  - `Ref.View<T>` Wrapper: `T` muss immutable sein (sonst auch nicht atomar)
- Vordefinierte transaktionelle Collections (weil `Map<K,V>>` nicht transaktionell)
  - `STM.newMap()`, `STM.newSet()`, `STM.newArrayAsList()` (fixe Grösse), Aber: normale List fehlt
  - `final Map<String, BankAccount> accounts = STM.newMap();`

- Falls etwas nicht immutable: Dann muss es in `atomic` eingepackt werden

```
final Ref.View<Integer> balance = STM.newRef(0);
final Ref.View<Date> lastUpdate = STM.newRef(new Date());

import scala.concurrent.stm.japi.STM;
void withdraw(int amount) {
  STM.atomic(() -> {
    if (balance.get() < amount) { // Für amount gibt es kein Grund zur Isolation
      STM.retry(); // Nur weiter, falls Kontostand OK
    }
    balance.set(balance.get() - amount);
    lastUpdate.set(new Date());
  });
}
```

Rollback und Abbrechen mit Exceptions (weil retry bricht ab):

```
STM.atomic(() -> {
  if (balance.get() >= Limit) {
    throw new RuntimeException("Balance limit"); // Abbruch mit Rollback
  }
  ...
}
```

- Begriff der Serialisierbarkeit: Es kann parallel laufen, muss aber aussehen wie es seriell geschehen wäre.
- Seiteneffekte: Generell kein IO machen.
- Write Skew Problem: Isolationsfehler (Bereitschaftsdienst, Schleife), in Scala STM nicht möglich
- Starvation Probleme: Wiederholter Abbruch einer Transaktion (immer wieder Konflikte, lang-laufende Transaktionen, keine Fortschrittsgarantie), ScalaSTM: beliebige Wiederholung möglich