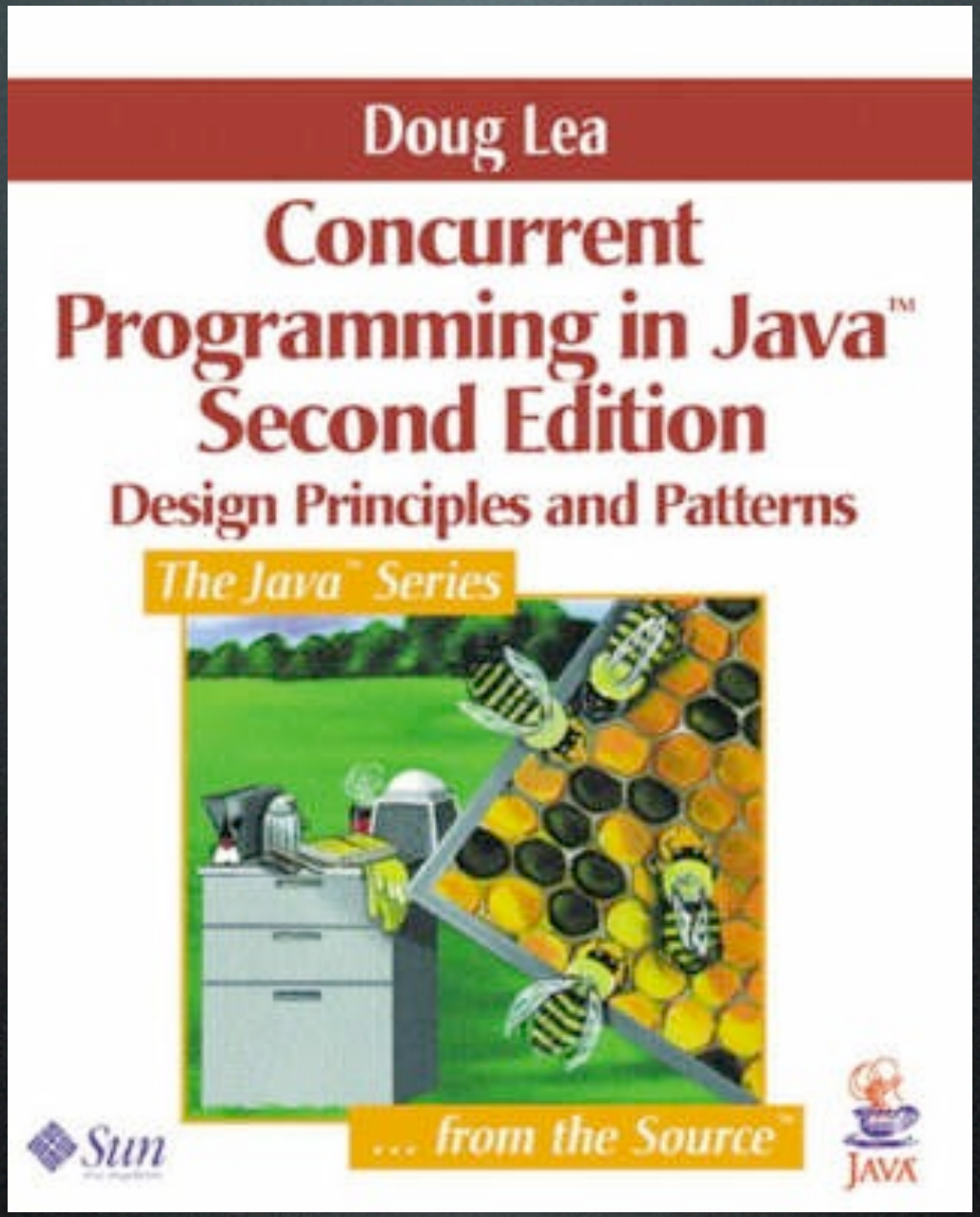
Mein paralleles Leben als Java-Programmierer

Johannes Link

johanneslink.net

Stephan Mosel - http://www.flickr.com/photos/moe/9474926/

2000


Doug Lea
Concurrent Programming in Java™ Second Edition
Design Principles and Patterns
The Java™ Series
... from the Source™
Sun
JAVA

```java
public class MyClass {
    private volatile String myProp = "";
    public synchronized String getMyProp() {
        return myProp;
    }
    public void setMyProp(String value) {
        synchronized (this) {
            myProp = value;
            this.notifyAll();
        }
    }
    public synchronized void waitUntilPropNotEmpty()
                                throws InterruptedException {
        while (myProp.equals("")) {
            this.wait();
        }
    }
}
```

```java
public class MyClassTest...
    private volatile boolean succeeded = false;
    @Test public void waitingForAValue() throws Exception {
        final MyClass my = new MyClass();
        Thread myThread = new Thread(new Runnable() {
            public void run() {
                try {
                    my.waitUntilPropNotEmpty();
                    succeeded = true;
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
        myThread.start();
        my.setMyProp("");
        assertFalse(succeeded);
        my.setMyProp("test");
        myThread.join();
        assertTrue(succeeded);
    }
```

Angenehme Position des Halbwissens

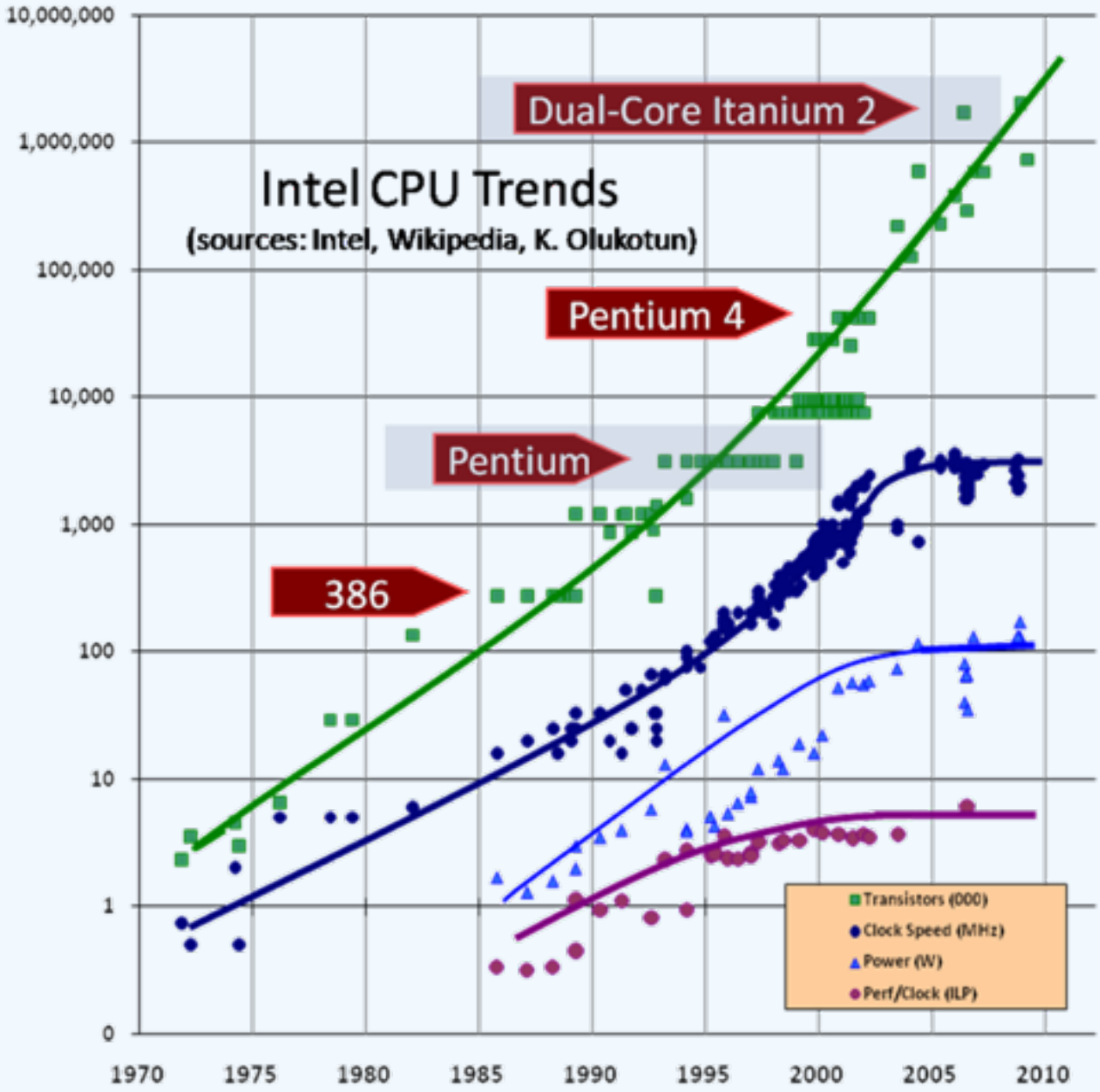
"Nebenläufigkeit wird völlig überbewertet"

"Das erledigen die Frameworks für mich"

"Ich trenne meine Fachlogik von Nebenläufigkeit und Threading"

"Ich denke mal ganz intensiv darüber nach und dann funktioniert das"

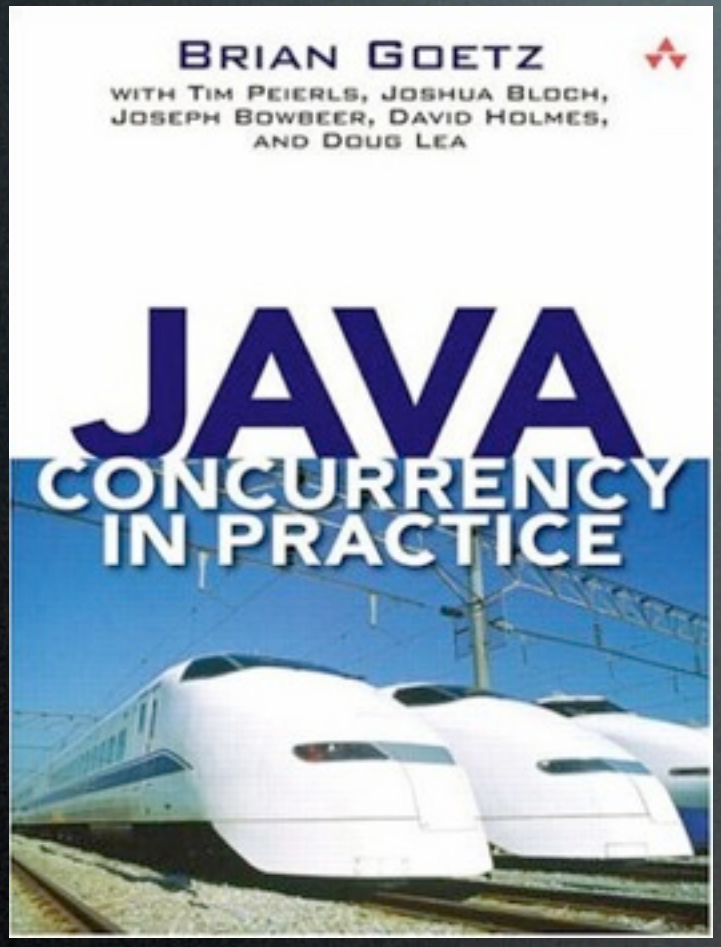
Kein kostenloses Mittagessen mehr...

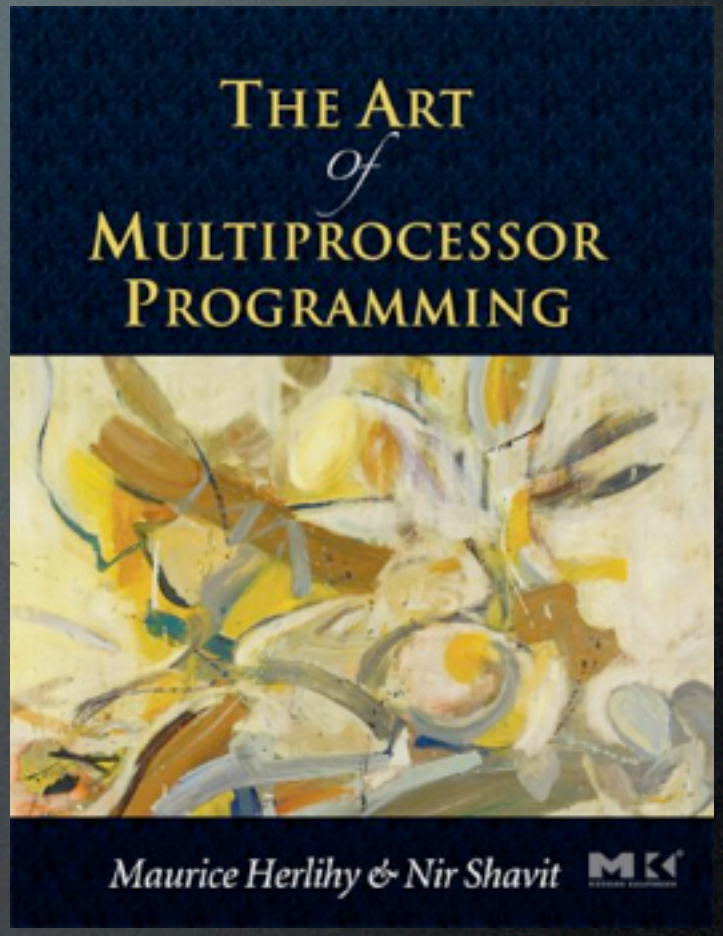


aus Herb Sutter: "The free lunch is over"

"Applications will increasingly need to be concurrent if they want to fully exploit continuing exponential CPU throughput gains."

Herb Sutter


BRIAN GOETZ
WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA
JAVA
CONCURRENCY
IN PRACTICE


THE ART
Of
MULTIPROCESSOR
PROGRAMMING
Maurice Herlihy & Nir Shavit

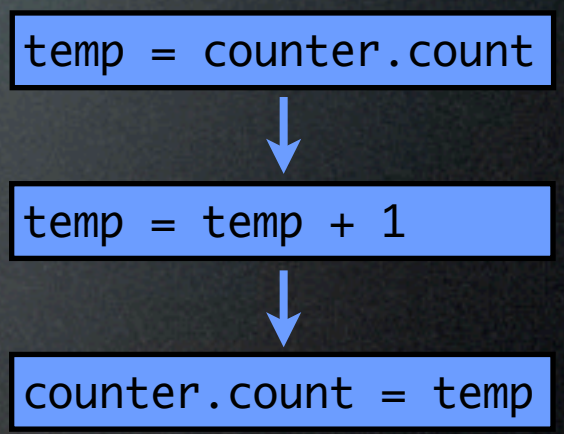
Was ist das Problem?

- Safety:
  Nothing bad ever happens
- Liveness:
  Something good eventually happens
- Performance:
  Things happen faster

Safety - Alles ist korrekt

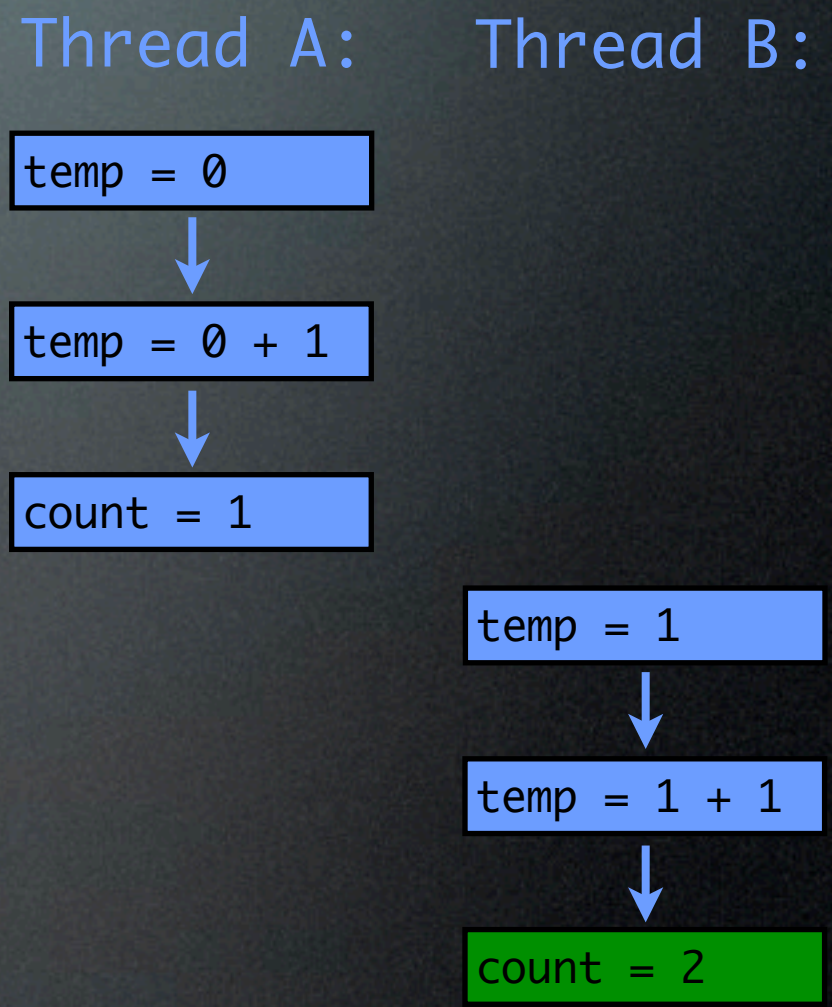
```
public class Counter {
    private long count = 0;
    public long value() {
        return count;
    }
    public void inc() {
        count++;
    }
}

Counter counter = new Counter();
//in Thread A:
counter.inc();
//in Thread B:
counter.inc();
//danach:
assert counter.value() == 2;
```

counter.inc():



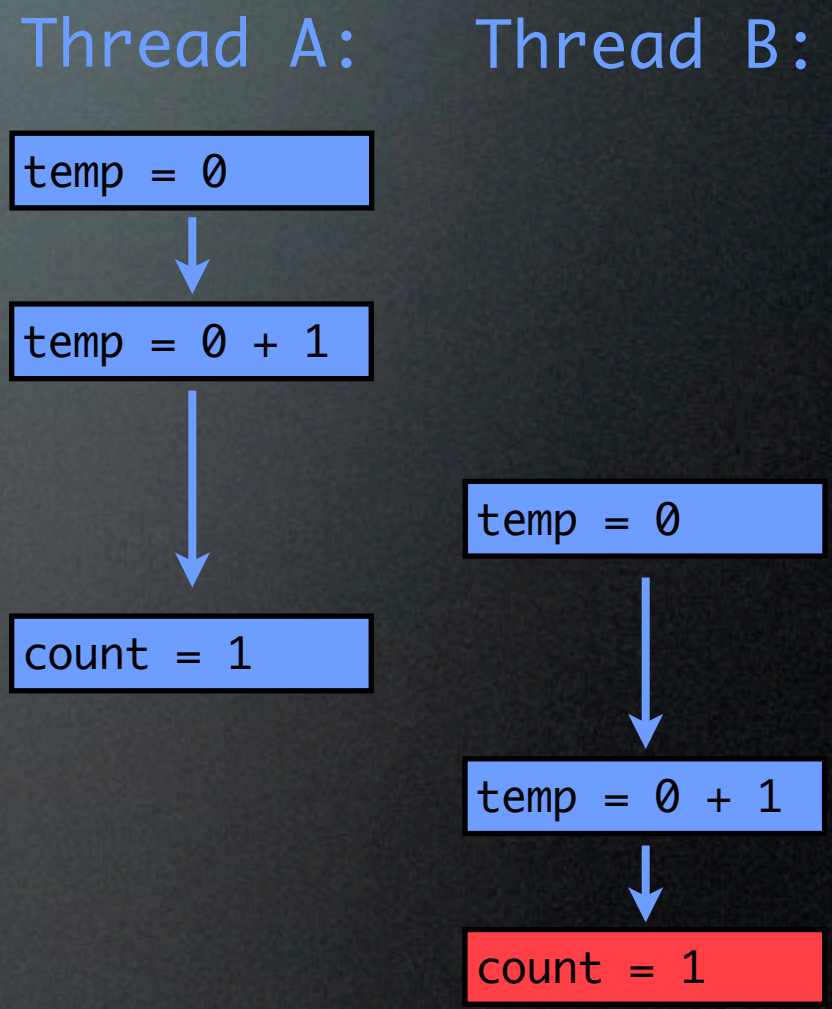
Safety - Alles ist korrekt

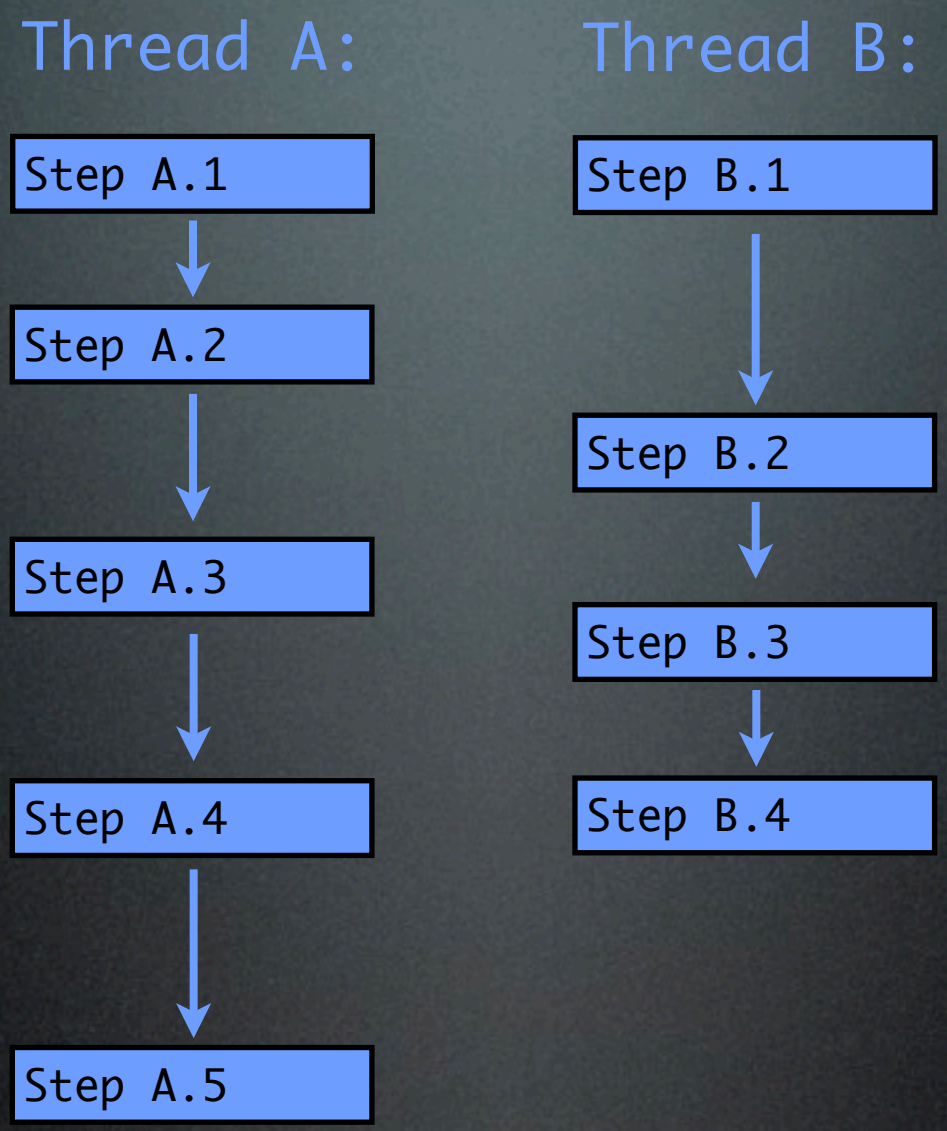
```
public class Counter {
    private long count = 0;
    public long value() {
        return count;
    }
    public void inc() {
        count++;
    }
}

Counter counter = new Counter();
//in Thread A:
counter.inc();
//in Thread B:
counter.inc();
//danach:
assert counter.value() == 2;
```

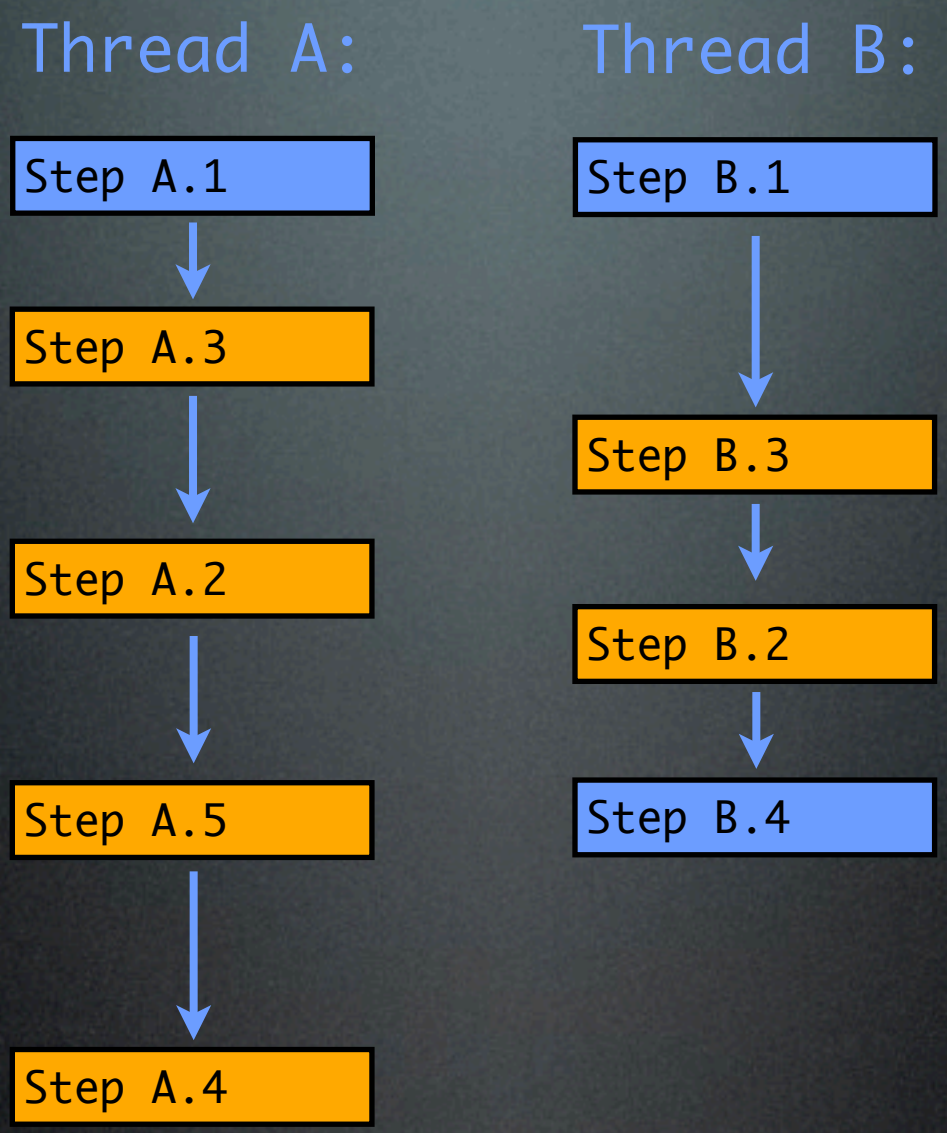


Safety - Alles ist korrekt

```
public class Counter {
    private long count = 0;
    public long value() {
        return count;
    }
    public void inc() {
        count++;
    }
}

Counter counter = new Counter();
//in Thread A:
counter.inc();
//in Thread B:
counter.inc();
//danach:
assert counter.value() == 2;
```




Thread A:
Step A.1
Step A.2
Step A.3
Step A.4
Step A.5
Thread B:
Step B.1
Step B.2
Step B.3
Step B.4


Thread A:
Step A.1
Step A.3
Step A.2
Step A.5
Step A.4
Thread B:
Step B.1
Step B.3
Step B.2
Step B.4

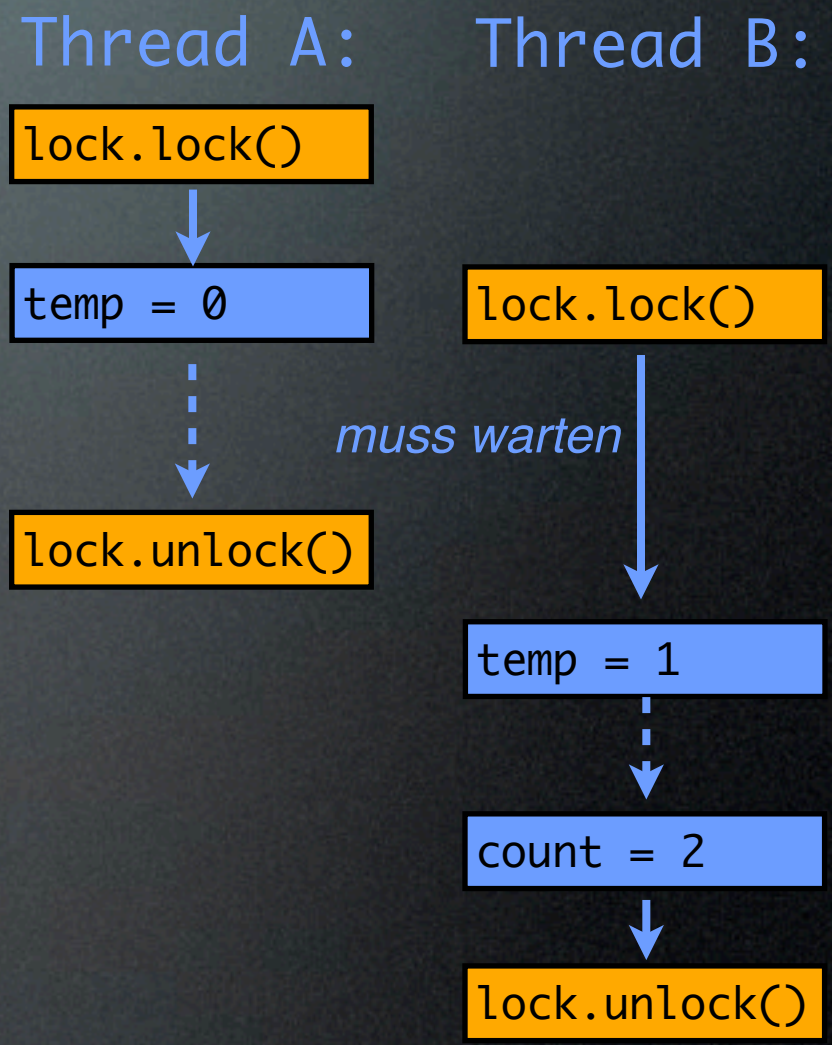
Was bedeutet korrekt in einem nebenläufigen Programm?

Synchronisation nebenläufiger Objekte

Bestimmte Programmbereiche werden durch Schlösser (Lock) in eine definierte sequenzielle Reihenfolge gebracht

Locks

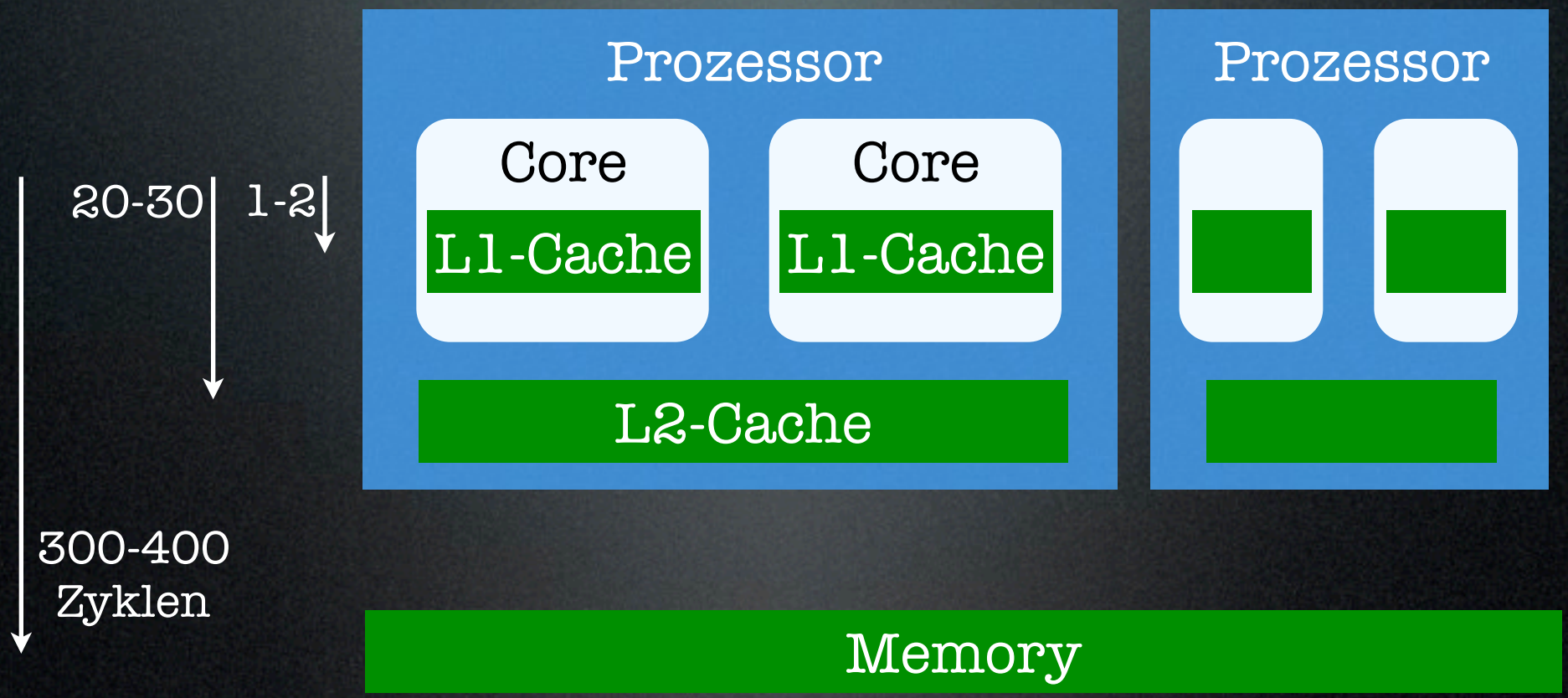
```
public class LockedCounter {
    private long count = 0;
    private Lock lock
          = new ReentrantLock();

    public long value() {
        lock.lock();
        long value = count;
        lock.unlock();
        return value;
    }

    public void inc() {
        lock.lock();
        count++;
        lock.unlock();
    }
}
```



Locks sind nicht umsonst

- Sie bergen das Risiko von Deadlocks
- Sie führen zu sequenzialisierter Programmausführung
- Sie benötigen zusätzliche Prozessorzyklen
  - Basieren auf primitiven Operationen (TAS oder CAS), welche (oft) auf "richtiges" Memory zugreifen müssen

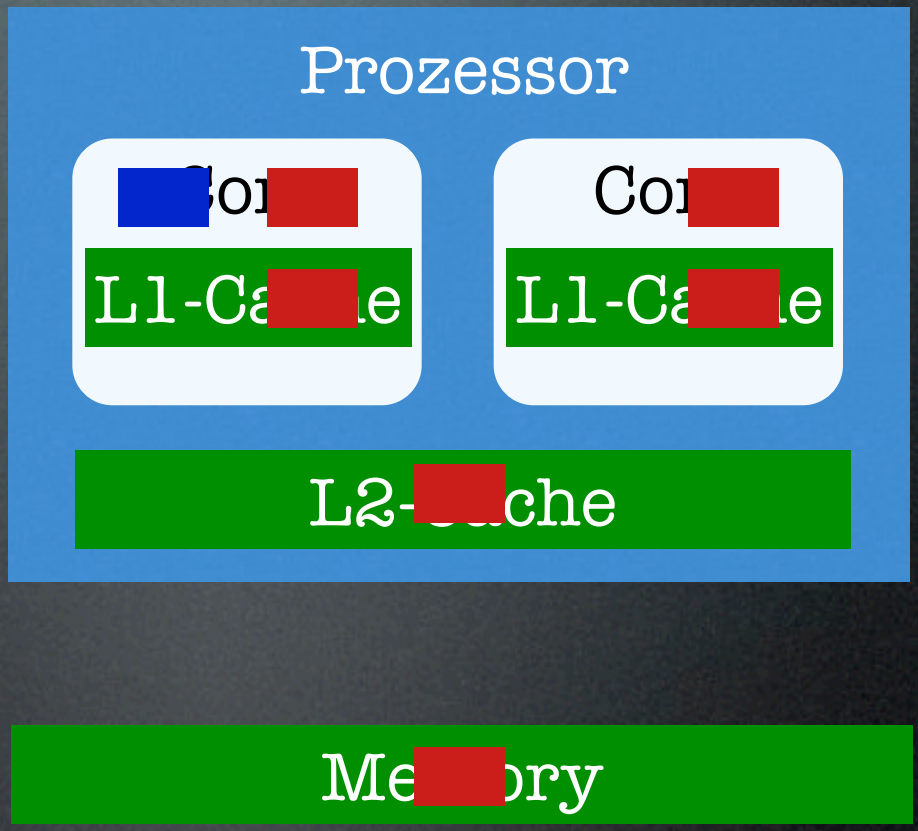
Caching, Memory & Co



Wie machen wir ein Objekt für andere Threads sichtbar?

Unshared Object

Shared Object



Sicheres Publizieren - Safe Publication

- Initialisierung in statischem Initializer
- Speichern in `volatile`-Feld oder in `AtomicReference`
- Ein unveränderliches Objekt
- Speichern in einem Feld, das vollständig durch einen Lock abgesichert ist

Grundsätze für die Verwendung von Locks

Halte genau dann einen Lock,

- wenn du auf gemeinsamen und veränderlichen Zustand zugreifst
- wenn du atomare Operationen ausführst
  - check then act
  - read-modify-write

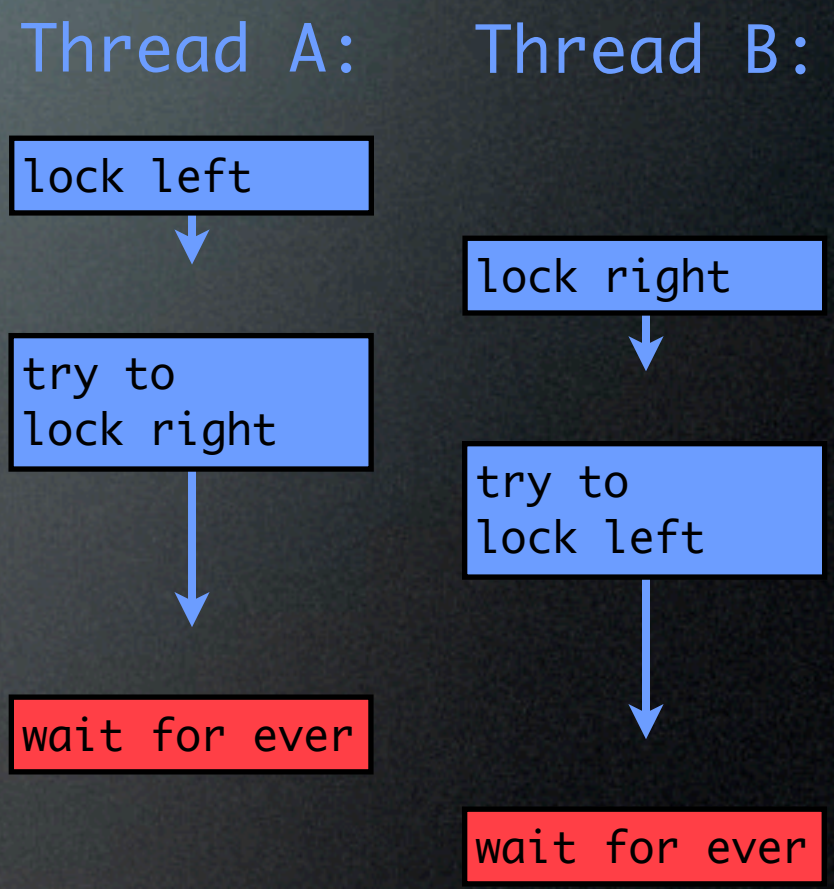
Halte den Lock nicht länger als nötig

Liveness - Jeder sollte an die Reihe kommen

- Gefahr 1: Deadlocks
- Gefahr 2: Starvation

Deadlock

```
public class SimpleDeadlock...
  private final Object left
      = new Object();
  private final Object right
      = new Object();
  public void leftRight() {
    synchronized (left) {
      synchronized (right) {
        doSomething();
      }
    }
  }
  public void rightLeft() {
    synchronized (right) {
      synchronized (left) {
        doSomething();
      }
    }
  }
```



Starvation

Bei hohem "Wettbewerb" (contention) um die gleiche Ressource (z.B. Lock) kommen nicht alle wartenden Threads an die Reihe

Performance

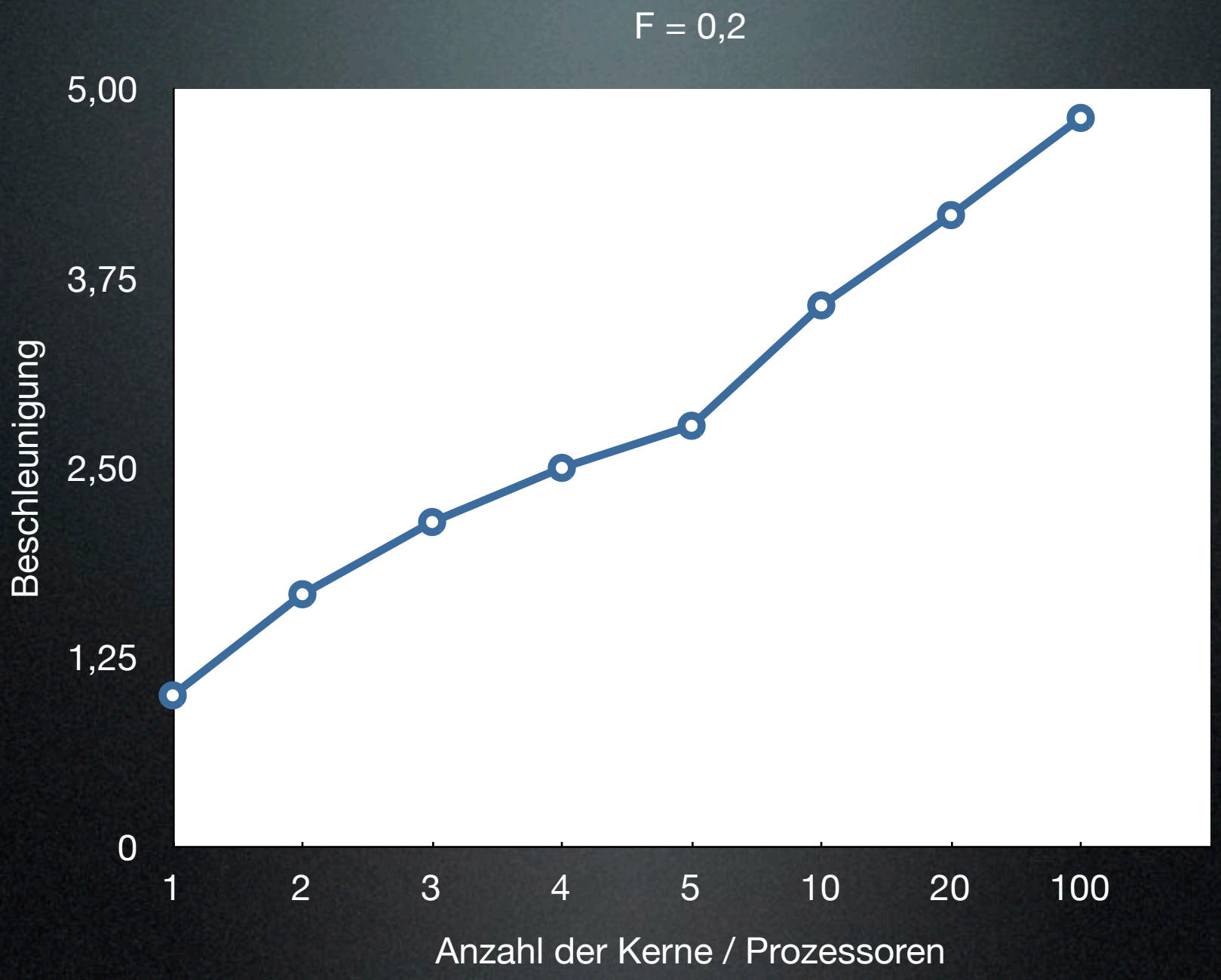
- Mehr Prozessoren/Kerne sollten zu besserer Performance führen
  - Besseres Antwortverhalten durch Verlagerung lang-laufender Berechnungen in parallele Threads
  - Höherer Durchsatz durch Aufteilung von Berechnungen in mehrere Threads

Amdahl's Law

$$speedup \leq \frac{1}{\left(F + \frac{(1 - F)}{N}\right)}$$

F: nicht parallelisierbare Anteil eines Programms

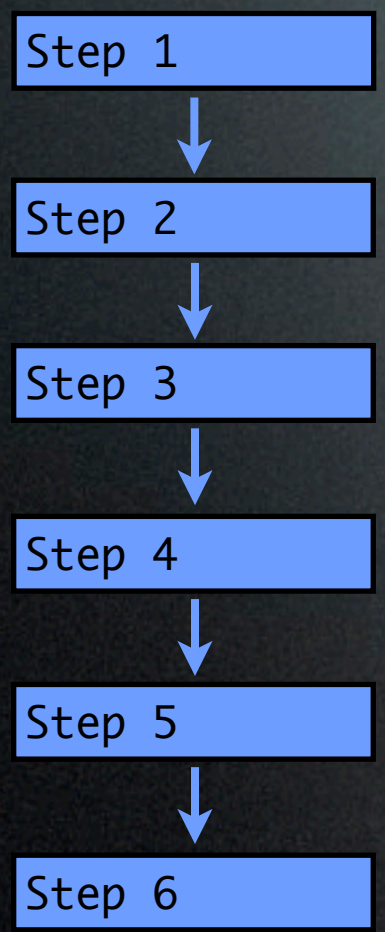
N: Anzahl der Kerne / Prozessoren

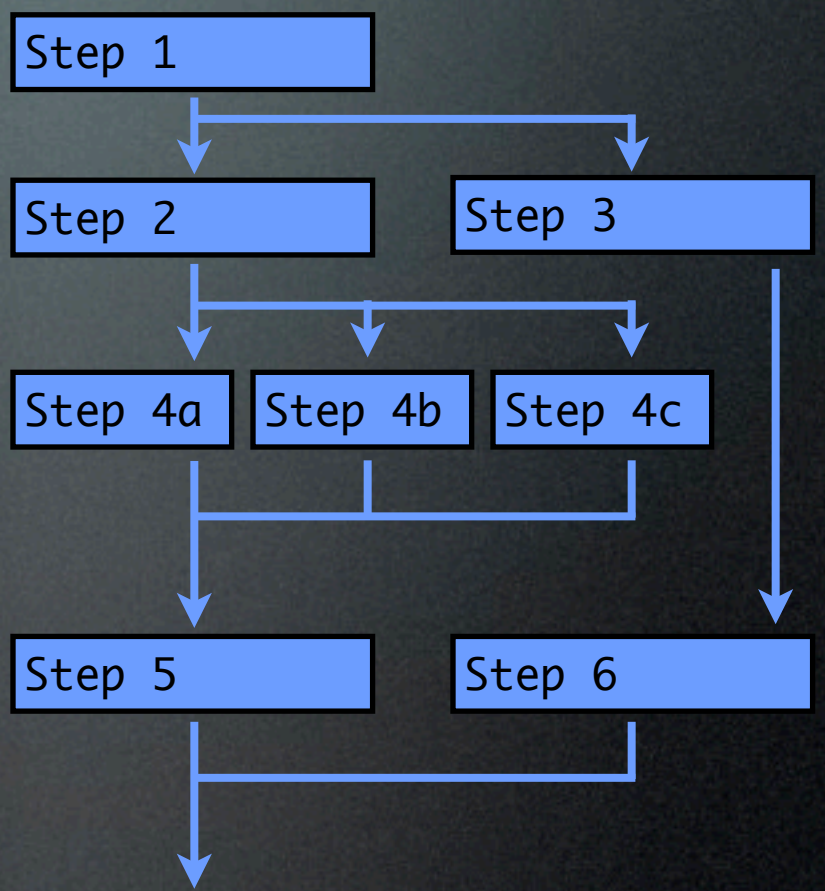

F = 0,2
5,00
3,75
2,50
1,25
0
Beschleunigung
1
2
3
4
5
10
20
100
Anzahl der Kerne / Prozessoren

Wie parallelisiert man ein Programm?

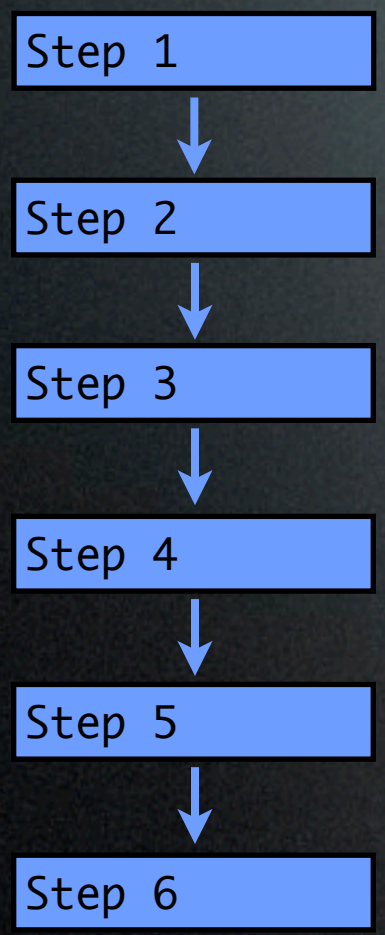
- Parallelisierung des Kontrollflusses
- Parallellisierung der Daten

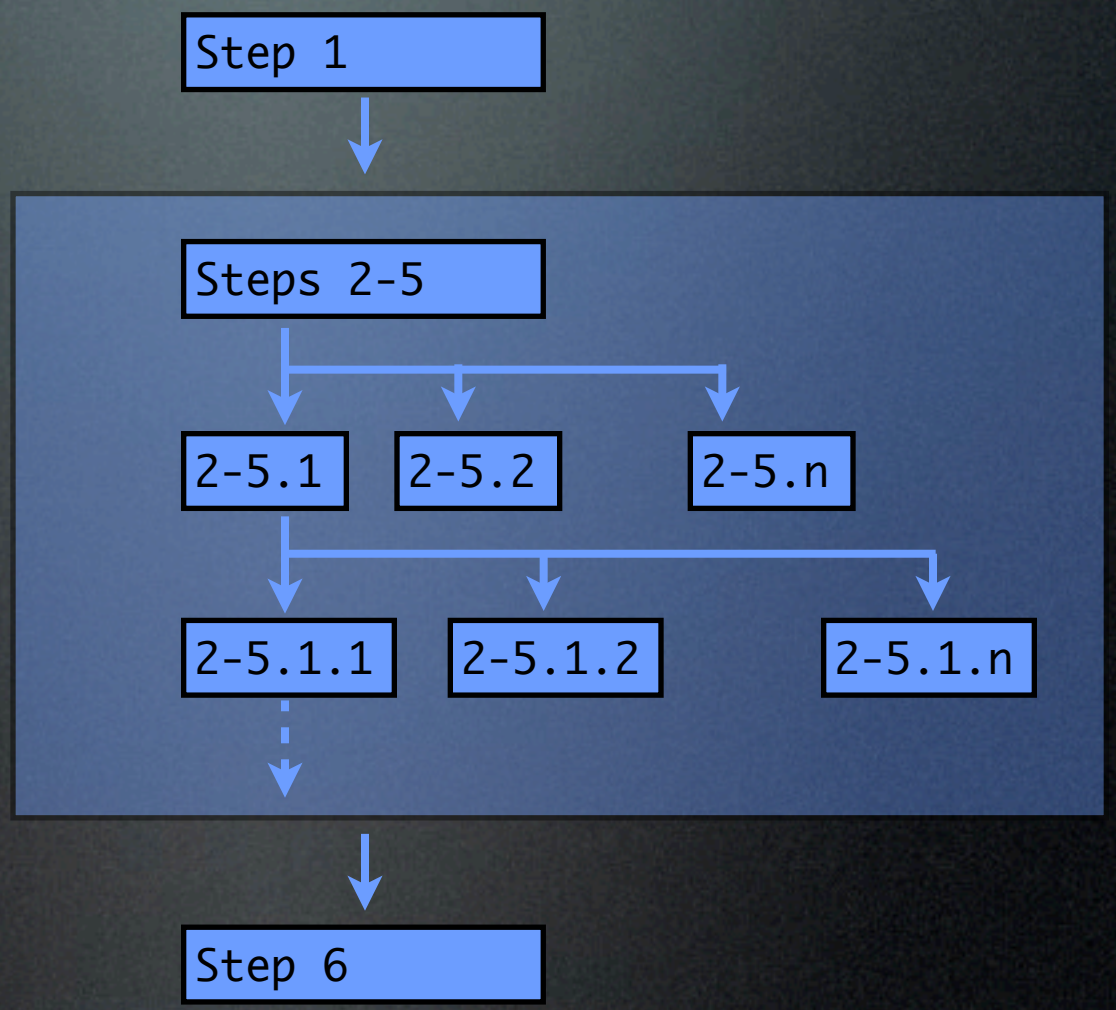
Parallelisierung des Kontrollflusses



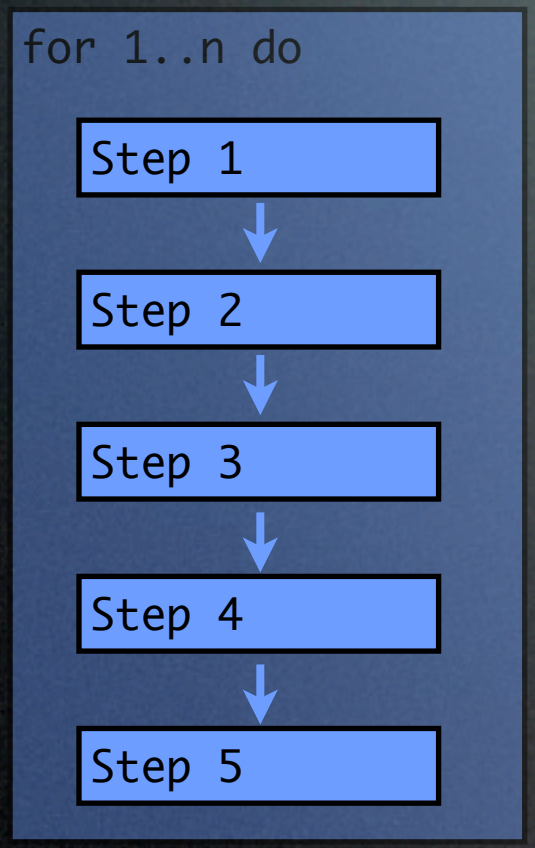


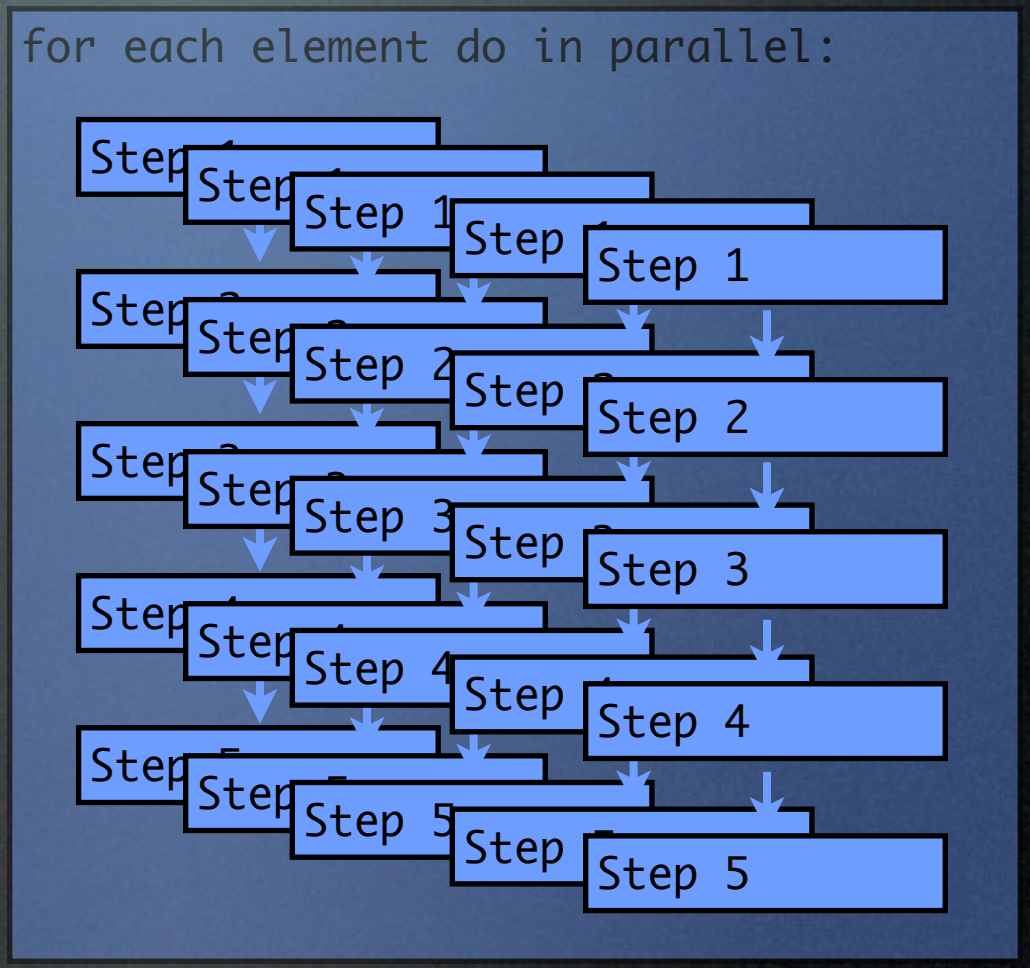
Rekursive Zerlegung des Kontrollflusses (Fork/Join)





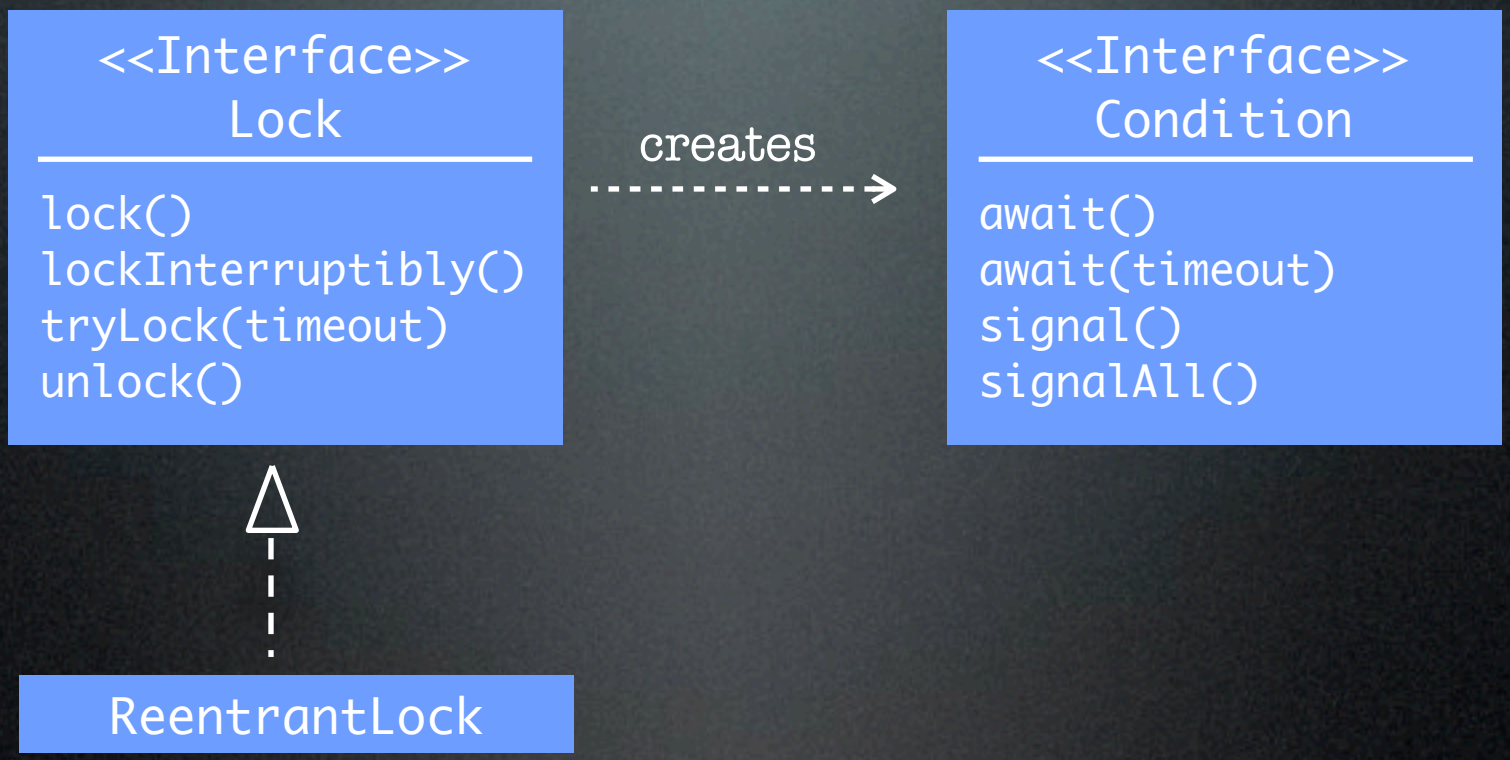
Parallelisierung der Daten





java.util.concurrent [.atomic|locks]

Locks und Conditions



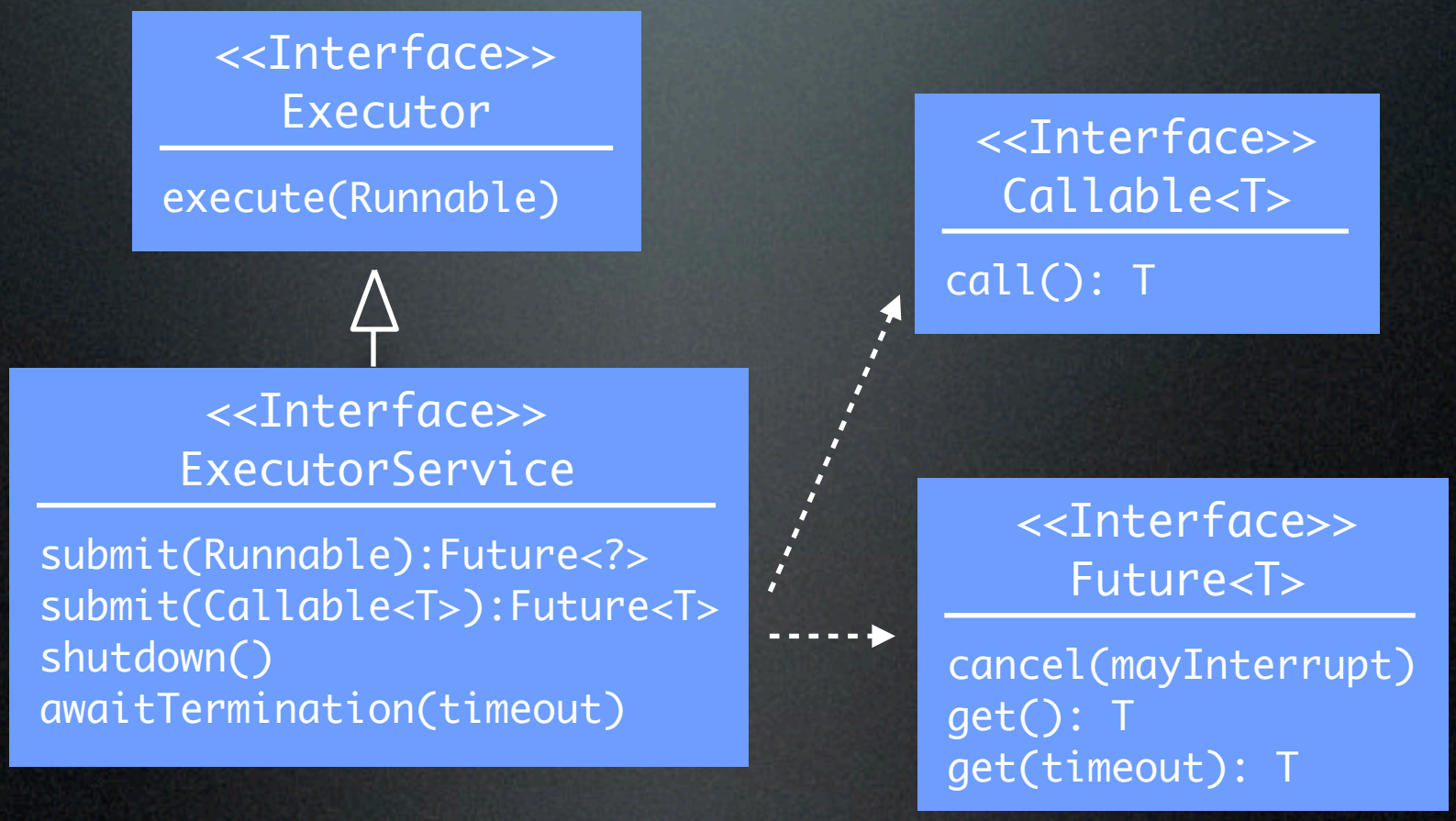
```java
public class MyNewClass...
private String myProp = "";
private Lock lock = new ReentrantLock();
private Condition propNotEmpty = lock.newCondition();

public String getMyProp() {
    lock.lock();
    try {
        return myProp;
    } finally {lock.unlock();}
}

public void setMyProp(String value) {
    lock.lock();
    try {
        myProp = value;
        propNotEmpty.signalAll();
    } finally {lock.unlock();}
}

public void waitUntilPropNotEmpty() throws InterruptedException {
    lock.lock();
    try {
        while (myProp.equals("")) {
            propNotEmpty.await();
        }
    } finally {lock.unlock();}
}
```

Implizite oder explizite Locks / Conditions

- Verwende implizite Locks und Conditions, wenn dir deren Feature-Set genügt
  - Performance seit Java 6 praktisch gleich
- Verwende explizite Locks und Conditions, wenn du mehr brauchst
  - Unterbrechbarkeit
  - tryLock()
  - Timeouts
  - Mehr als eine Condition-Queue

Das Executor-Framework

Nie wieder `new Thread(...).start()` !



```java
public class MyNewClassTest...

private volatile boolean succeeded = false;

@Test
public void waitingForAValue() throws Exception {
    final MyNewClass my = new MyNewClass();
    ExecutorService executor = Executors.newCachedThreadPool();
    Runnable task = new Runnable() {
        @Override
        public void run() {
            try {
                my.waitUntilPropNotEmpty();
                succeeded = true;
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    };
    Future<?> result = executor.submit(task);
    my.setMyProp("");
    assertFalse(succeeded);
    my.setMyProp("test");
    result.get(); //wait for end of task
    assertTrue(succeeded);
}
```

Atomics

```
public class SynchronizedCounter...
    private long count = 0;

    public synchronized long value() {
        return count;
    }

    public synchronized void inc() {
        count++;
    }
```

```
public class AtomicCounter...
    private AtomicLong count = new AtomicLong(0);

    public long value() {
        return count.longValue();
    }

    public void inc() {
        count.incrementAndGet();
    }
```

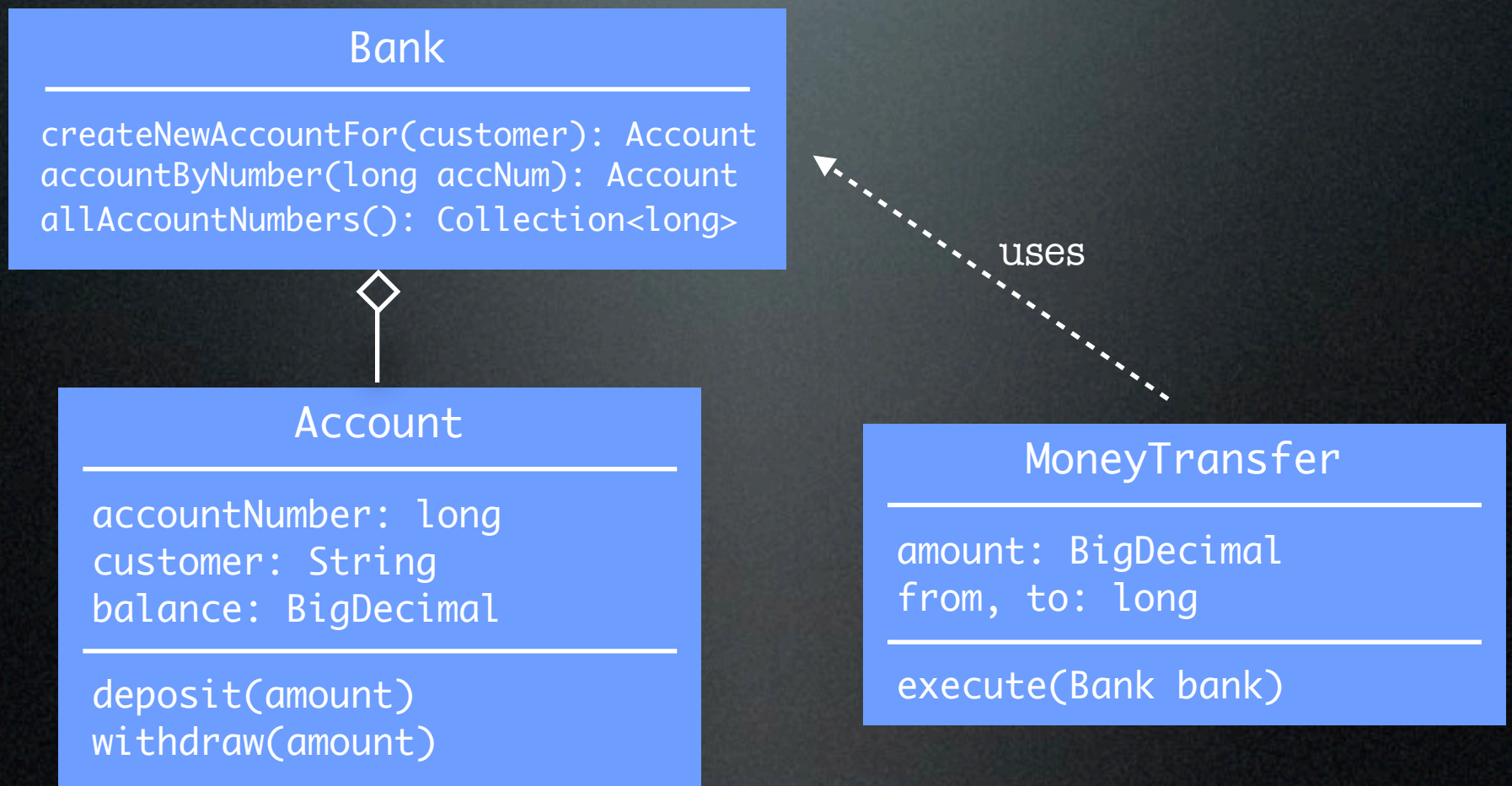
Compare and Set

```
public class HandmadeAtomicReference<V>...
    private V ref;

    public HandmadeAtomicReference(V initialValue) {...}

    public synchronized V get() { return ref; }

    /**
    * Atomically sets the value to the given updated value
    * if the current value {@code ==} the expected value.
    * @param expect the expected value
    * @param update the new value
    * @return true if successful.
    */
    public synchronized boolean compareAndSet(V expect, V update) {
        if (ref != expect)
            return false;
        ref = update;
        return true;
    }
```

Und noch mehr...

- Concurrent Collections
- BlockingQueue
- Latches, Barriers, Exchangers & Co

Mein Lieblingsbeispiel: Banking



```java
public class Account {
    private BigDecimal balance = BigDecimal.ZERO;
    private final long accountNumber;
    private final String customer;

    public Account(long accountNumber, String customer) {...}

    public void deposit(BigDecimal amount) {
        balance = balance.add(amount);
    }

    public void withdraw(BigDecimal amount) {
        balance = balance.subtract(amount);
    }

    public BigDecimal getBalance() {
        return balance;
    }

    public long getAccountNumber() {
        return accountNumber;
    }

    public String getCustomer() {
        return customer;
    }
}
```

```java
public class Bank {

    private Map<Long, Account> accounts = new HashMap<Long, Account>();
    private long lastAccountNumber = 0;

    public Account createNewAccountForString(String customer) {
        Account account = new Account(++lastAccountNumber, customer);
        accounts.put(account.getAccountNumber(), account);
        return account;
    }

    public Account accountByNumber(long accountNumber) {
        return accounts.get(accountNumber);
    }
}
```

```java
public class AccountCreationTest extends ParallelTestCase...
    private volatile Bank bank = new Bank();
    @Test
    public void createAccountsInManyThreads() throws Exception {
        final int numberOfAccounts = 100;
        final AtomicInteger account = new AtomicInteger(0);
        Runnable createAccountTask = new Runnable() {
            @Override
            public void run() {
                String customer = "c" + account.incrementAndGet();
                long accountNumber = bank.createNewAccountForString(
                                                customer).getAccountNumber();
                assertEquals(customer, bank.accountByNumber(
                                                accountNumber).getCustomer());
            }
        };
        runInParallelThreads(numberOfAccounts, createAccountTask);
        assertAllAccountsHaveBeenCreated(numberOfAccounts);
    }
```

"No account with number 99"

```
public class Bank {
    private Map<Long, Account> accounts = new HashMap<Long, Account>();
    private long lastAccountNumber = 0;

    public Account createNewAccountForString(String customer) {
        Account account = new Account(++lastAccountNumber, customer);
        accounts.put(account.getAccountNumber(), account);
        return account;
    }

    public Account accountByNumber(long accountNumber) {
        return accounts.get(accountNumber);
    }
}
```

NullPointerException

```
public class Bank {
    private Map<Long, Account> accounts = new HashMap<Long, Account>();
    private AtomicLong lastAccountNumber = new AtomicLong(0L);

    public Account createNewAccountForString(String customer) {
        Account account =
            new Account(lastAccountNumber.incrementAndGet(), customer);
        accounts.put(account.getAccountNumber(), account);
        return account;
    }

    public Account accountByNumber(long accountNumber) {
        return accounts.get(accountNumber);
    }
}
```

ok

```
public class Bank {
    private final Map<Long, Account> accounts =
                    new ConcurrentHashMap<Long, Account>();
    private final AtomicLong lastAccountNumber = new AtomicLong(0L);

    public Account createNewAccountForString(String customer) {
        Account account =
            new Account(lastAccountNumber.incrementAndGet(), customer);
        accounts.put(account.getAccountNumber(), account);
        return account;
    }

    public Account accountByNumber(long accountNumber) {
        return accounts.get(accountNumber);
    }
}
```

```java
public class Account...
    private final AtomicReference<BigDecimal> balance =
            new AtomicReference<BigDecimal>(BigDecimal.ZERO);
    public void deposit(BigDecimal amount) {
        while (true) {
            BigDecimal oldAmount = balance.get();
            BigDecimal newAmount = oldAmount.add(amount);
            if (balance.compareAndSet(oldAmount, newAmount))
                return;
        }
    }
    public void withdraw(BigDecimal amount) {
        while (true) {
            BigDecimal oldAmount = balance.get();
            BigDecimal newAmount = oldAmount.subtract(amount);
            if (balance.compareAndSet(oldAmount, newAmount))
                return;
        }
    }
    public BigDecimal getBalance() {
        return balance.get();
    }
```

```java
public class Account...

    private BigDecimal balance = BigDecimal.ZERO;

    public synchronized void deposit(BigDecimal amount) {
        balance = balance.add(amount);
    }

    public synchronized void withdraw(BigDecimal amount) {
        balance = balance.subtract(amount);
    }

    public synchronized BigDecimal getBalance() {
        return balance;
    }
```

```java
public class MoneyTransfer {

    private final BigDecimal amount;
    private final long from;
    private final long to;

    public MoneyTransfer(BigDecimal amount, long from, long to) {
        this.amount = amount;
        this.from = from;
        this.to = to;
    }

    public synchronized void execute(Bank bank) {
        Account source = bank.accountByNumber(from);
        Account target = bank.accountByNumber(to);
        if (source.getBalance().compareTo(amount) < 0)
            throw new InsufficientFundsException();
        source.withdraw(amount);
        target.deposit(amount);
    }
}
```

```java
public class MoneyTransfer...
    public void execute(Bank bank) {
        Account source = bank.accountByNumber(from);
        Account target = bank.accountByNumber(to);
        synchronized (source) {
            synchronized (target) {
                doTransfer(source, target);
            }
        }
    }
    private void doTransfer(Account source, Account target) {
        if (source.getBalance().compareTo(amount) < 0)
            throw new InsufficientFundsException();
        source.withdraw(amount);
        target.deposit(amount);
    }
```

```java
// Thread A:
new MoneyTransfer(new BigDecimal(2), 234, 789).execute(bank);

// Thread B:
new MoneyTransfer(new BigDecimal(2), 789, 234).execute(bank);
```

```java
public class MoneyTransfer...
    public void execute(Bank bank) {
        Account source = bank.accountByNumber(from);
        Account target = bank.accountByNumber(to);
        Object[] locks = new Object[] { source, target };
        Arrays.sort(locks);
        synchronized (locks[0]) {
            synchronized (locks[1]) {
                doTransfer(source, target);
            }
        }
    }
```

```java
public class Account implements Comparable<Account>...
    public int compareTo(Account other) {
        return new Long(accountNumber).
                    compareTo(new Long(other.accountNumber));
    }
```

Für den normal sterblichen Entwickler ist die Erstellung korrekter Multi-Thread-Programme außerhalb ihrer Fähigkeiten

```java
public class ValueHolder {
    private List<Listener> listeners = new LinkedList<Listener>();
    private int value;

    public static interface Listener {
        public void valueChanged(int newValue);
    }

    public void addListener(Listener listener) {
        listeners.add(listener);
    }

    public void setValue(int newValue) {
        value = newValue;
        for (Listener each : listeners) {
            each.valueChanged(newValue);
        }
    }
}
```

The Problem with Threads

http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf

Zugrundeliegende Programmiermodell

Shared mutable state (aka objects)

is accessed in multiple **concurrent threads,**

and we use **locks** to synchronize / sequentialize access to the state

Das **Objekt** wird zur "undichten" Abstraktion

Um aus einzelnen thread-sicheren Objekten komplexere thread-sichere Objekte zu bauen, muss der Locking-Mechanismus der Einzelobjekte bekannt sein

- Verletzt Prinzip der Kapselung
- Verletzt Prinzip der Modularisierung

Können uns andere Programmierparadigmen retten?

Alternative Paradigmen

- Transactional Memory
- Immutability
- Message Passing
- Parallel Collection Processing
- Dataflow Concurrency

Transactional Memory (TM)

- Heap als transaktionale Datenmenge
- Transaktionseigenschaften ähnlich wie bei einer Datenbank (ACI~~D~~)
- Optimistische Transaktionen
- Transaktionen werden bei einer Kollision automatisch wiederholt
- Transaktionen können geschachtelt und komponiert werden

```java
public class Account...
    public void deposit(BigDecimal amount) {
        atomic {
            balance = balance.add(amount);
        }
    }
    public void withdraw(BigDecimal amount) {
        atomic {
            balance = balance.subtract(amount);
        }
    }
```

```java
public class MoneyTransfer...
    public void execute(Bank bank) {
        atomic {
            Account source = bank.accountByNumber(from);
            Account target = bank.accountByNumber(to);
            if (source.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            source.withdraw(amount);
            target.deposit(amount);
        }
    }
```

Eigenschaften von TM

- Vorteile:
  - Wir können weiter in shared state und Transaktionen denken
  - Die korrekte Verwendung ist wesentlich einfacher als bei Locks: Deadlocks sind ausgeschlossen!
- Nachteile:
  - Keinerlei Hilfe, wie wir unser sequenzielles Programm parallelisieren
  - Der Fortschritt ist nicht garantiert. Livelocks sind möglich
  - Die performante Implementierung ist noch Forschungsthema

TM-Implementierungen

- Hardware Transactional Memory
- Software Transactional Memory
  - Clojure: Programmiersprache auf der JVM, die STM eingebaut hat
  - Java-STM-Frameworks:
    Deuce, Akka, Multiverse

Immutability

- Ohne veränderlichen Zustand, müssen Veränderungen auch nicht synchronisiert werden
- Systeme haben aber doch einen Zustand der sich ändert, oder?
- Trick: Wir unterscheiden zwischen unveränderlichen Werten (values) und einer bestimmten Entity zugeordnenten aktuellen Wert

```
@Immutable
class Account {
    BigDecimal balance
    long accountNumber
    String customer

    static Account create(long accountNumber, String customer) {
        new Account(balance: 0.0, accountNumber: accountNumber,
                    customer: customer)
    }

    Account deposit(amount) {
        cloneWithChange(balance: balance + amount)
    }

    Account withdraw(amount) {
        cloneWithChange(balance: balance - amount)
    }

    private Account cloneWithChange(changes) {
        def newProps = ...
        new Account(newProps)
    }
}
```

```
@Immutable
class Account...
    BigDecimal balance
    long accountNumber
    String customer
    long version

    static Account create(long accountNumber, String customer) {
        new Account(balance: 0.0, accountNumber: accountNumber,
                    customer: customer, version: 0)
    }

    Account incrementVersion() {
        cloneWithChange(version: version + 1)
    }

    Account deposit(amount) {
        cloneWithChange(balance: balance + amount)
    }

    Account withdraw(amount) {
        cloneWithChange(balance: balance - amount)
    }
```

```
class Bank...
    private final accounts = new ConcurrentHashMap()

    Account createNewAccountFor(String customer) {
        Account account =
          Account.create(nextAccountNumber(), customer).incrementVersion();
        accounts[account.getAccountNumber()] = account
    }

    Account accountByNumber(long accountNumber) {
        accounts[accountNumber]
    }

    synchronized boolean update(Account... accountsToUpdate) {
        if (accountsToUpdate.any {acc ->
            acc.version != accounts[acc.accountNumber]?.version
        }) {
            return false
        }
        accountsToUpdate.each {acc ->
            accounts[it.accountNumber] = it.incrementVersion()
        }
        return true
    }
```

```
@Immutable
class MoneyTransfer {

    BigDecimal amount
    long from, to

    boolean execute(Bank bank) {
        while(true) {
            Account source = bank.accountByNumber(from);
            Account target = bank.accountByNumber(to);
            if (source.balance < amount)
                return false
            source = source.withdraw(amount)
            target = target.deposit(amount)
            if (bank.update(source, target))
                return true
        }
    }
}
```

Immutability im Großen

- Performante Implementierung von "Immutable Data Types" ist möglich
- In funktionale Programmiersprachen sind unveränderliche Werte die Norm und veränderlicher State die Ausnahme
  - Beispiel Clojure: Fokus auf unveränderliche Werte und explizite Mechanismen zur Manipulation des aktuellen Werts einer Entity

Message Passing - Actors

- Vollständiger Verzicht auf "shared state"
- Aktoren empfangen und verschicken Nachrichten
  - Asynchron und nicht-blockierend
  - Jeder Actor hat seine "Mailbox"
- Immer nur ein aktiver Thread pro Aktor

```
class BankActor extends AbstractPooledActor...
    void act() {
        loop {
            react { message ->
                switch(message) {
                    case CreateAccount:
                         createNewAccountFor(message.customer); break
                    case GetAccount:
                         accountByNumber(message.accountNumber); break
                    case GetAllAccountNumbers:
                         allAccountNumbers(); break
                    case UpdateAccounts:
                         update(message.accounts); break
                    default:
                      println "${this.class}: I don't understand $message"
    }   }   }   }

@Immutable class CreateAccount { String customer }
@Immutable class UpdateAccounts { Account[] accounts }
@Immutable class GetAccount { long accountNumber }
@Immutable class GetAllAccountNumbers {}
```

```
class BankActor extends AbstractPooledActor...
    private accounts = [:]
    private lastAccountNumber = 0
    void createNewAccountFor(String customer) {
        Account account = ...
        reply account
    }
    void accountByNumber(long accountNumber) {
        reply accounts[accountNumber]
    }
    void allAccountNumbers() {
        reply accounts.keySet() as List
    }
    void update(Account[] accountsToUpdate) {
        if (accountsToUpdate.any { acc ->
            acc.version != accounts[acc.accountNumber]?.version
        }) {
            reply false
        } else {
            accountsToUpdate.each {acc ->
                accounts[acc.accountNumber] = acc.incrementVersion()
            }
            reply true
    }   }
```

```
class MoneyTransferActor...
  BankActor bank
  void act() {
    ...
      switch(message) {
          case TransferMoney: transferMoney(message); break;
  }
  void transferMoney(transfer) {
    while(true) {
      Account source = bank.sendAndWait(new GetAccount(transfer.from));
      Account target = bank.sendAndWait(new GetAccount(transfer.to));
      if (source.balance < transfer.amount) {
        reply new TransferResult(transfer.transferId, false)
        break
      }
      source = source.withdraw(transfer.amount)
      target = target.deposit(transfer.amount)
      if (bank.sendAndWait(new UpdateAccounts(accounts: [source, target]))) {
        reply new TransferResult(transfer.transferId, true)
        break
  } } }

@Immutable class TransferMoney {
    long transferId
    BigDecimal amount
    long from, to
}
```

Actors - Vorteile

- Unabhängige Aktoren
- Skalierbar
- Verteilbar
- Leichtere Vermeidung von
  - Race Conditions
  - Dead Locks
  - Starvation

Fehler-tolerant & hoch-verfügbar

- Mehrere redundante Aktoren stehen für die gleiche Aufgabe zur Verfügung
- Organisation in Supervisor-Hierarchien
  - Kein lokales Exception-Handling, sondern Neustart eines Aktors im Fehlerfalle

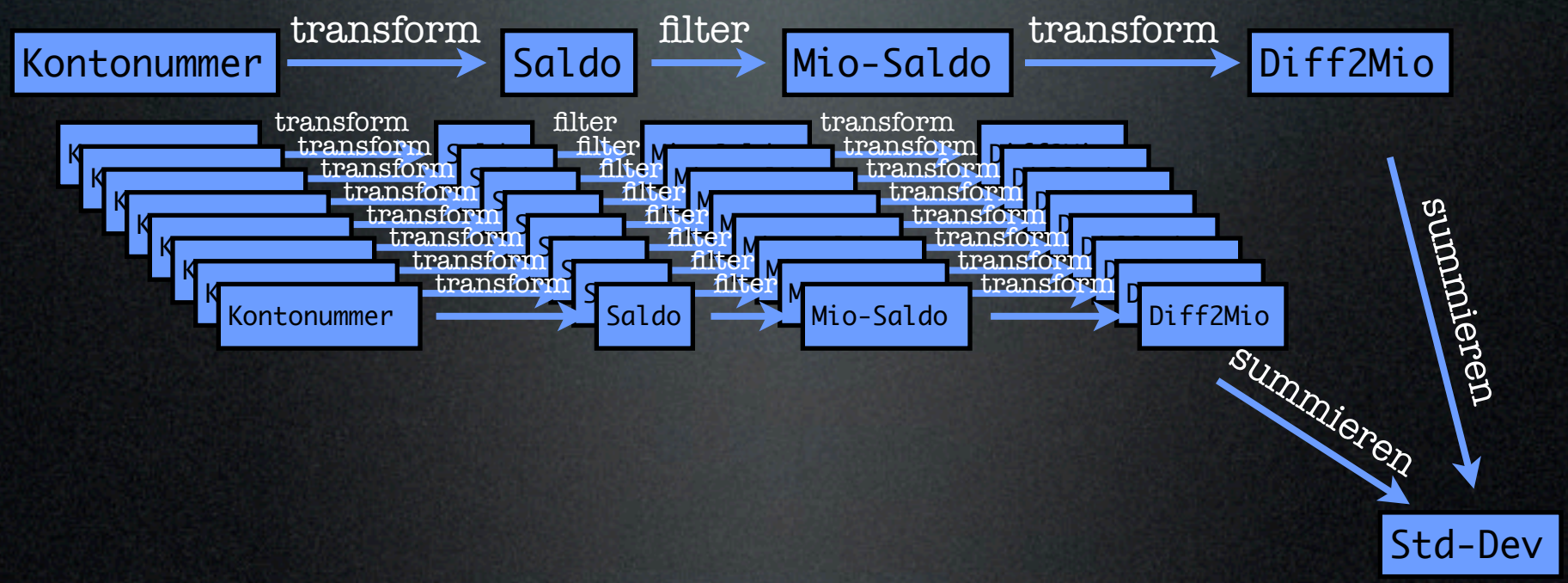
Actors - Nachteile

- Nicht geeignet für Probleme, die einen echten Konsens über gemeinsame Objekte erfordern
- Kommunikation über asynchrone Nachrichten ist für viele Problemstellungen eine Komplizierung

Parallele Collections

- Wir arbeiten auf allen Elementen einer Collection gleichzeitig
- Voraussetzung: Die Einzeloperationen sind unabhängig voneinander
- Typische parallele Aktionen:
  - Filtern
  - Transformieren

Beispiel: Standardabweichung aller Millionärskonten von 1.000.000



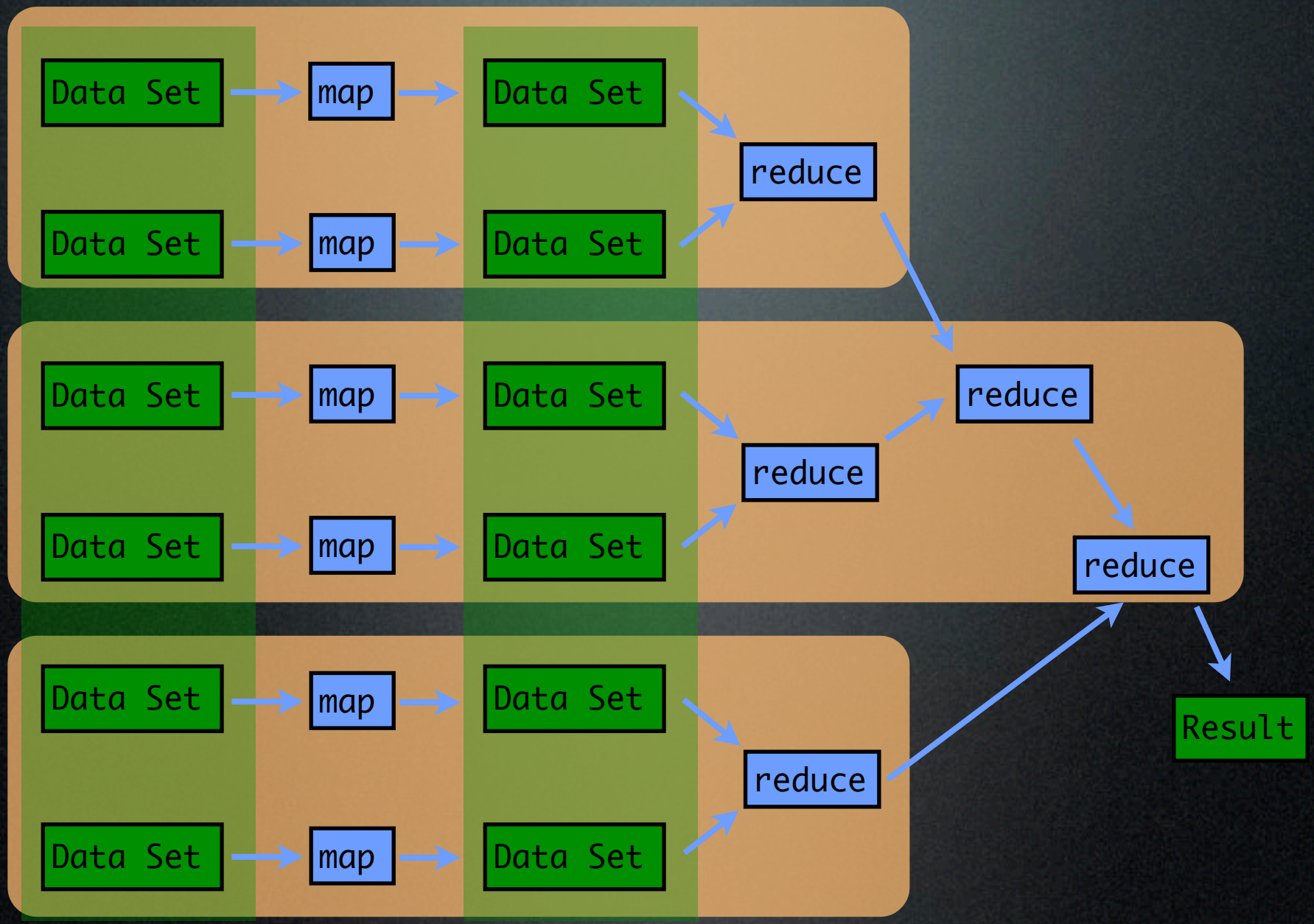
gpars' parallele Collections

```
Parallelizer.doParallel(numThreads) {
    def allBalances = bank.allAccountNumbers().collectParallel {
        return bank.accountByNumber(it).balance
    }
    def millionaireBalances = allBalances.findAllParallel { it >= MILLION }
    def deviations = millionaireBalances.collectParallel { balance ->
        def diffToMean = balance - MILLION
        return (diffToMean * diffToMean)
    }
    def count = deviations.size()
    def average = Math.sqrt(deviations.sumParallel() / count)
}
```

Collection
..work..
..work..
..work..
..work..
..work..
Collection
..work..
..work..
..work..
..work..
..work..

MapReduce

- Von Google patentierter Ansatz zur verteilten Bearbeitung großer Datencluster
- Name stammt von den map- und reduce-Funktionen funktionaler Sprachen
- Idee: Einzelne Datensätze werden durch eine Pipeline von Arbeitsschritten transformiert (**map**) und am Ende zu einem Ergebnis zusammengeführt (**reduce**)


Data Set
map
Data Set
reduce
Data Set
map
Data Set
Data Set
map
Data Set
reduce
reduce
Data Set
map
Data Set
reduce
Data Set
map
Data Set
Result

Map-Reduce mit gpars

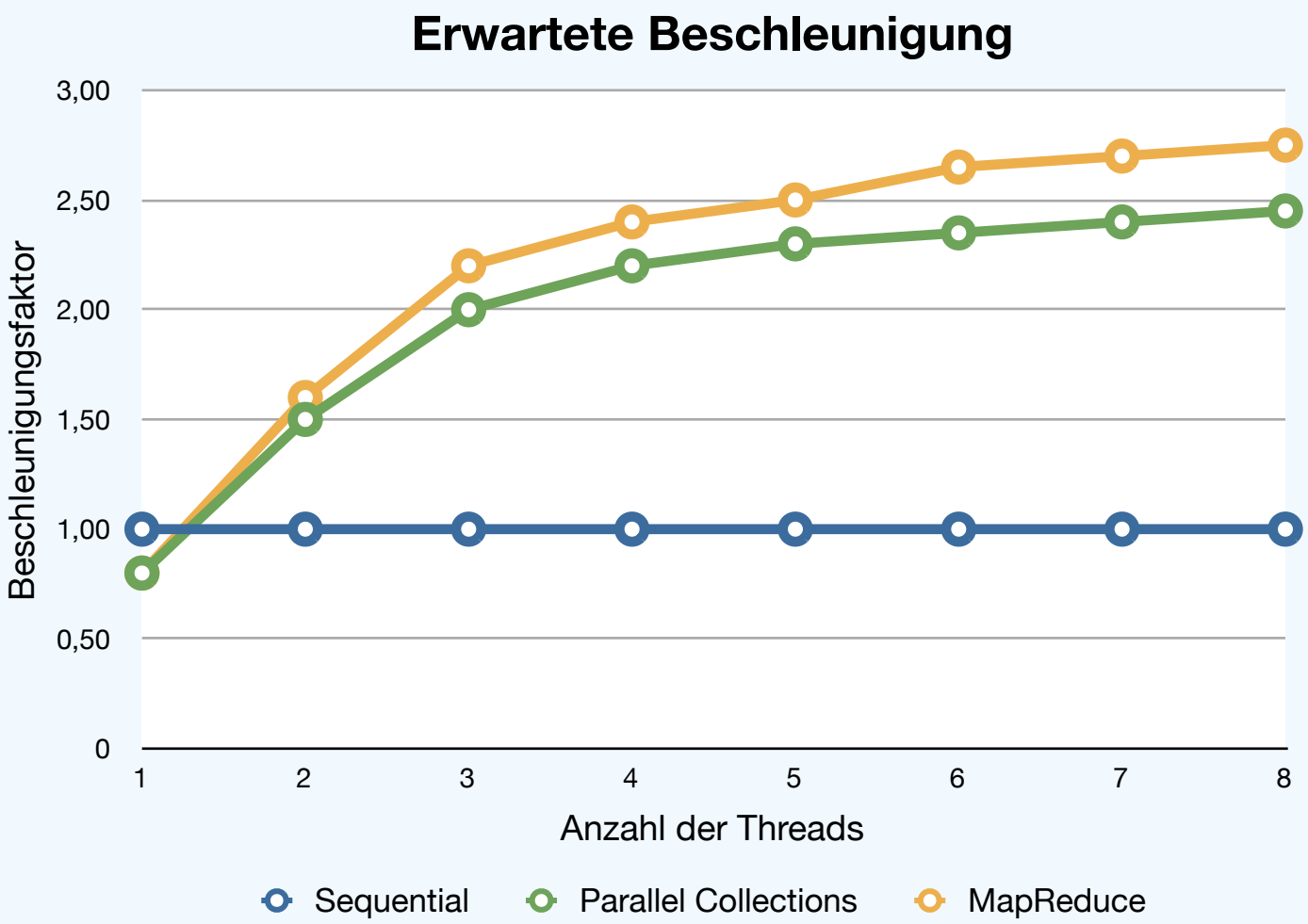
```
Parallelizer.doParallel(numThreads) {
    def deviations = bank.allAccountNumbers().parallel.map {accountNumber ->
        bank.accountByNumber(accountNumber).balance
    }.filter { balance ->  balance >= MILLION  }.map {balance ->
        def diffToMean = balance - MILLION
        (diffToMean * diffToMean)
    }
    //def sum = deviations.sum()
    def sum = deviations.reduce {a,b -> a + b}
    def count = deviations.size()
    def average = Math.sqrt(sum / count)
}
```

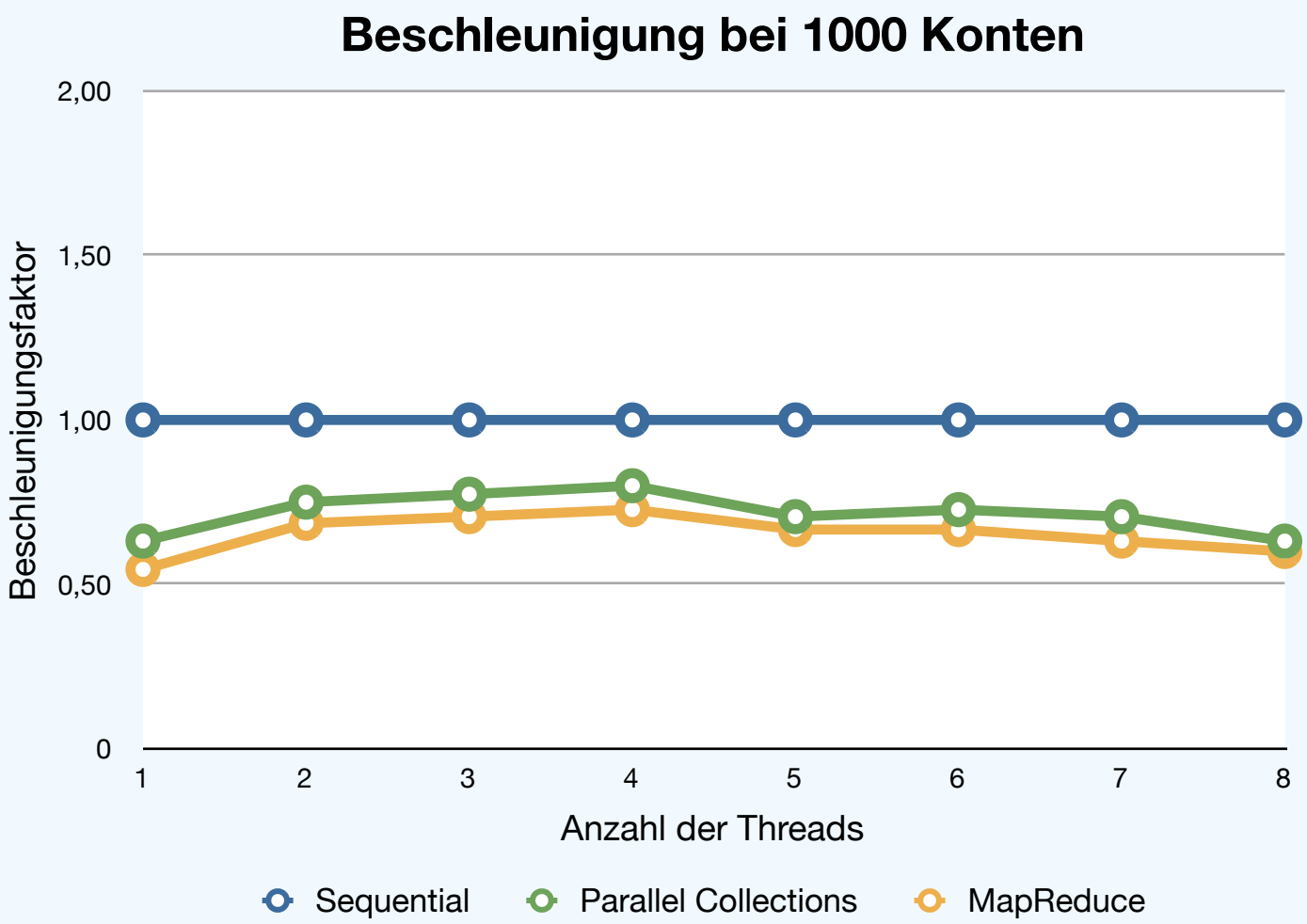
Kritik an MapReduce

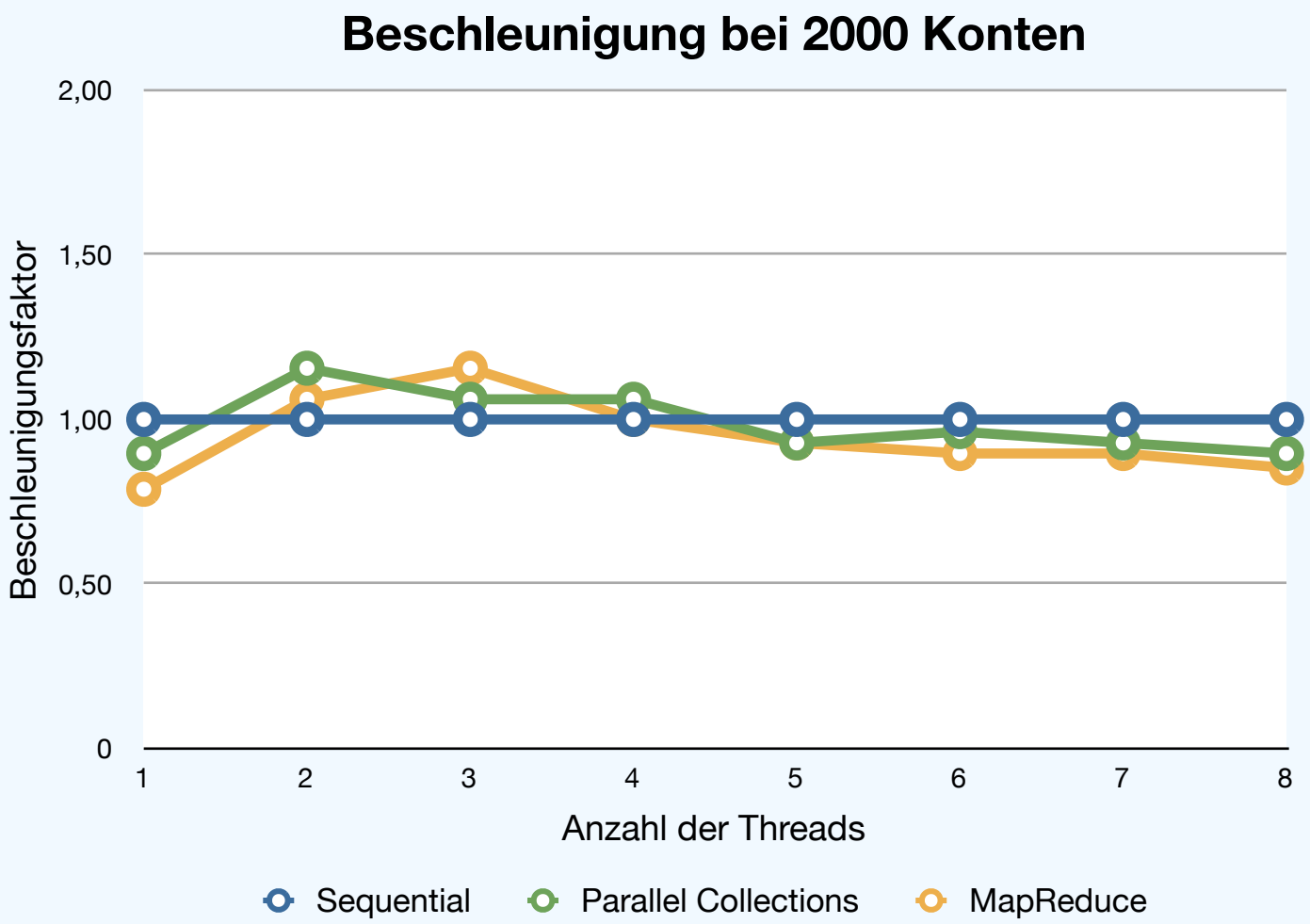
- Patentierbarkeit (claim of novelty) wird angezweifelt
- Wie breit ist der Problemraum, für den MapReduce sinnvoll angewandt werden kann?
- Wie sinnvoll ist MapReduce im nicht verteilten Kontext?

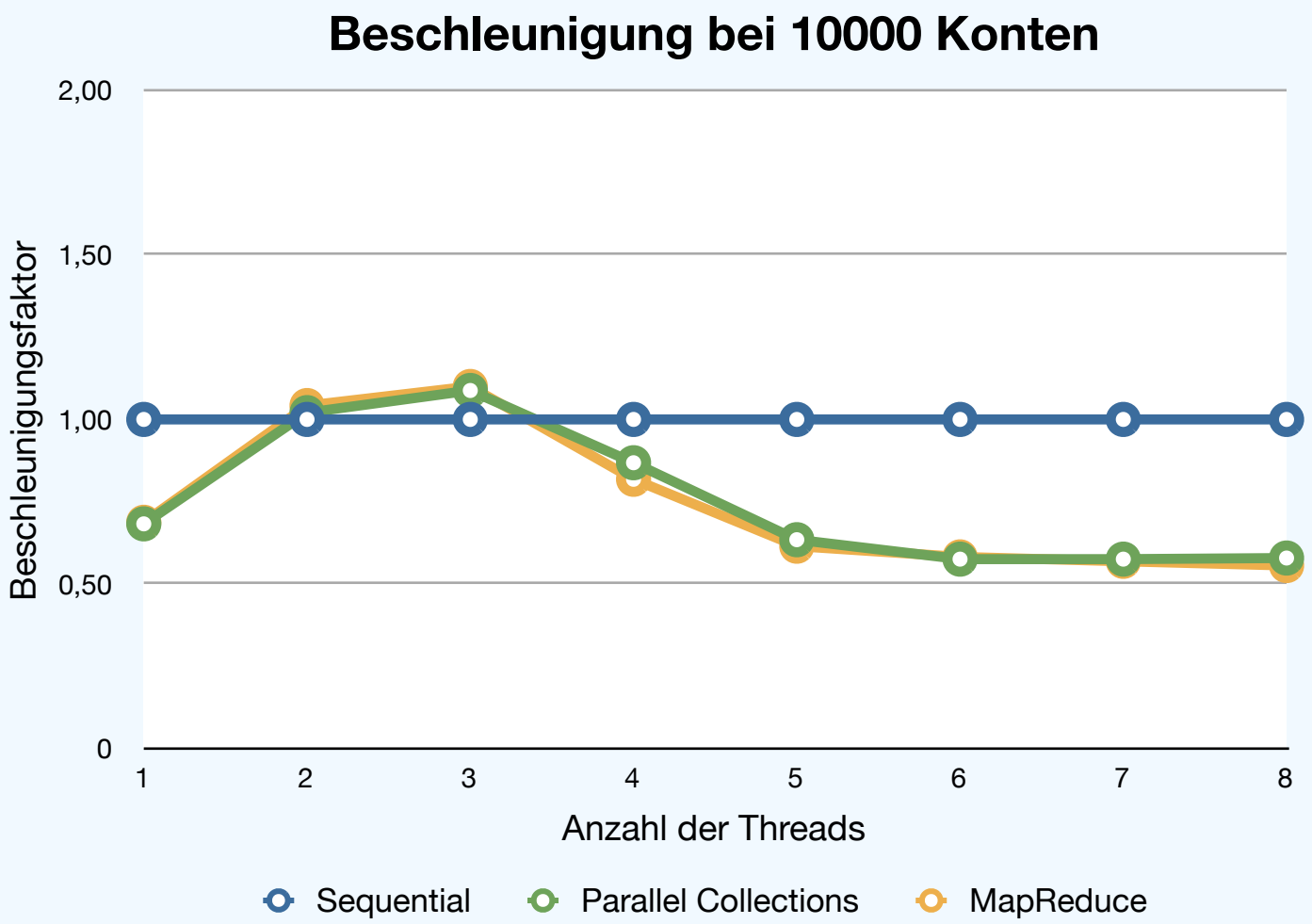
Performance-Vergleich

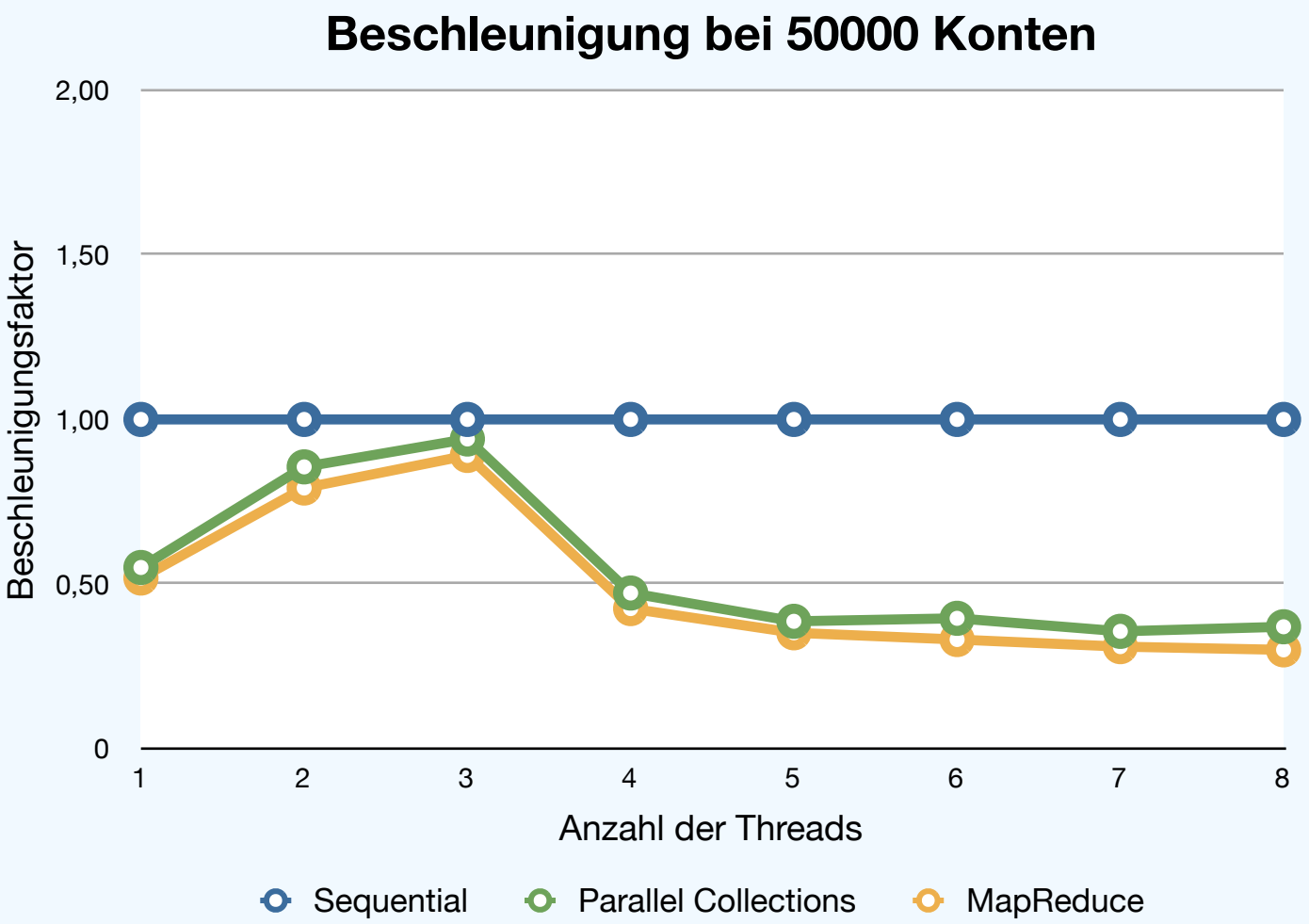
- Hardware:
  - Intel i7: 4 cores, 8 hyper threads
- Wettbewerber
  - Sequenziell
  - Parallel Collections
  - MapReduce
- Vefahren
  - jeweils 1 - 8 Threads
  - jeweils 10 Samples
  - 1000, 2000, 10.000, 50.000, 100.000 Konten

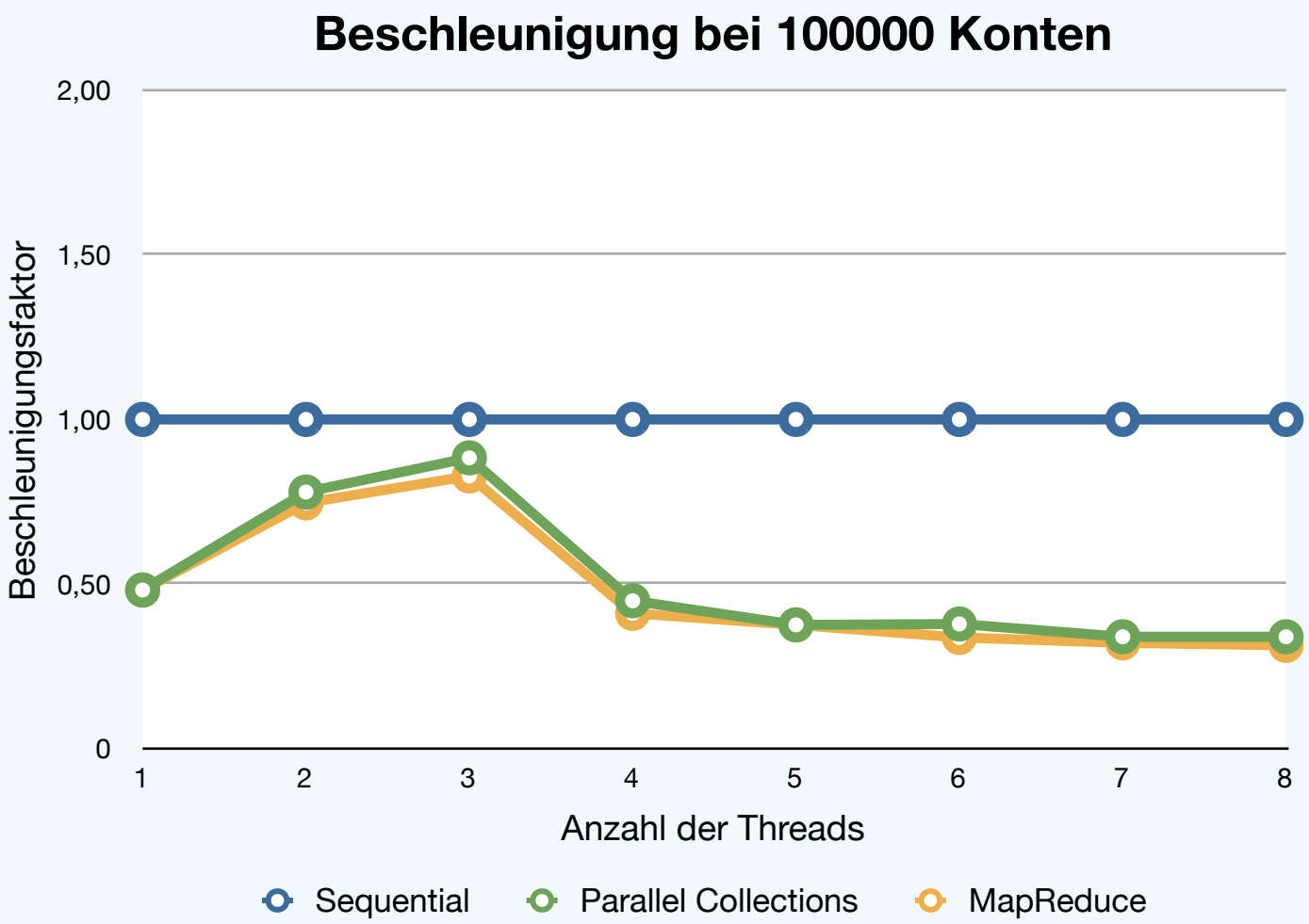

Erwartete Beschleunigung
3,00
2,50
2,00
1,50
1,00
0,50
0
Beschleunigungsfaktor
1
2
3
4
5
6
7
8
Anzahl der Threads
Sequential
Parallel Collections
MapReduce


Beschleunigung bei 1000 Konten
2,00
1,50
1,00
0,50
0
Beschleunigungsfaktor
1
2
3
4
5
6
7
8
Anzahl der Threads
Sequential
Parallel Collections
MapReduce


Beschleunigung bei 2000 Konten
2,00
1,50
1,00
0,50
0
Beschleunigungsfaktor
1
2
3
4
5
6
7
8
Anzahl der Threads
Sequential
Parallel Collections
MapReduce


Beschleunigung bei 10000 Konten
2,00
1,50
1,00
0,50
0
Beschleunigungsfaktor
1
2
3
4
5
6
7
8
Anzahl der Threads
Sequential
Parallel Collections
MapReduce


Beschleunigung bei 50000 Konten
2,00
1,50
1,00
0,50
0
Beschleunigungsfaktor
1
2
3
4
5
6
7
8
Anzahl der Threads
Sequential
Parallel Collections
MapReduce


Beschleunigung bei 100000 Konten
2,00
1,50
1,00
0,50
0
Beschleunigungsfaktor
1
2
3
4
5
6
7
8
Anzahl der Threads
Sequential
Parallel Collections
MapReduce

Data Flows

- Beschreibung der Abhängigkeiten von Daten untereinander
- Grundkonzept: Dataflow-Variablen
  - Wert kann nur einmal gebunden werden
  - Abfrage des Wertes blockiert bis er gebunden wurde

gpars - DataFlows

```
def x = new DataFlowVariable()
def y = new DataFlowVariable()
def z = new DataFlowVariable()

task { z << x.val + y.val }
task { x << 40 }
task { y << 2 }

assert 42 == z.val
```

```
def df = new DataFlows()

task { df.z = df.x + df.y }
task { df.x = 40 }
task { df.y = 2 }

assert 42 == df.z
```

Vorteile von Dataflows

- Kein Locking notwendig
- Keine Race-Conditions
- Deterministisch
  - Deadlocks sind möglich, sie passieren dann aber immer!
- Kein Unterschied zwischen sequenziellem und nebenläufigem Code

Beispiel: Depotwert

```
def depot = ['AAPL':10, 'GOOG':2, 'IBM':20, 'ORCL':500]
def stocks = depot.keySet()

def prices = new DataFlows()
def value = new DataFlowVariable()

stocks.each {stock ->
    task {
        prices[stock] = getStockPrice(stock)
    }
}

task {
    value << stocks.collect { stock ->
        prices[stock] * depot[stock]
    }.sum()
}

println value.val
```

Mehr über Data Flows

- Einschränkungen
  - Nicht für jede Problemstellung geeignet
  - Berechnungen dürfen keine Seiteneffekte haben
- Erweiterungen
  - Data flow streams, Data flow operators
- JVM-Implementierungen
  - Scala Dataflow, FlowJava (akademisch & tot)

Nicht behandelt

- Fork/Join (jsr166y)
- CSP:
  Communicating Sequential Processes
- Declarative Concurrency
- und andere...

Fazit

- Multi-thread-orientierte und Lock-basierte Programmierung ist schwierig
  - Hauptgrund: Thread-sichere Objekte sind zusammengesetzt nicht komponierbar
- "Immutability" erleichtert das parallele Leben sehr
- Andere Paradigmen können helfen, der heilige Kral der Multi-Core-Entwicklung ist aber (noch) nicht gefunden

Links

Zum Lesen

http://www.infoq.com/presentations/goetz-concurrency-past-present

http://www.slideshare.net/jboner/state-youre-doing-it-wrong-javaone-2009

http://ajaxonomy.com/2008/news/toward-better-concurrency

http://igoro.com/archive/gallery-of-processor-cache-effects/

Zum Ausprobieren

http://gpars.codehaus.org/

http://clojure.org/

http://github.com/jboner/scala-dataflow