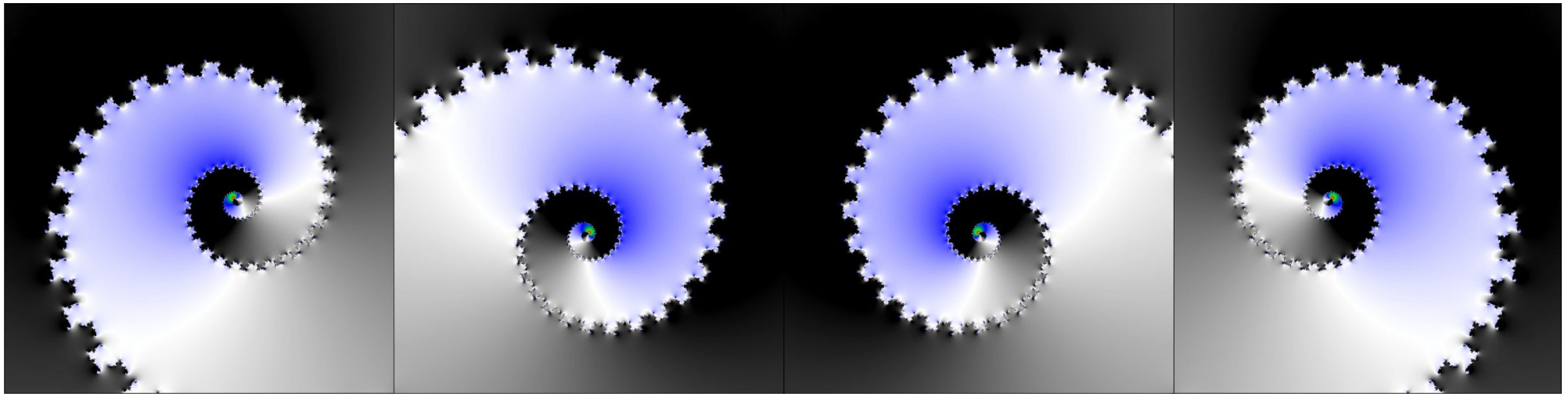
Java-Concurrency für Fortgeschrittene



arno haase consulting

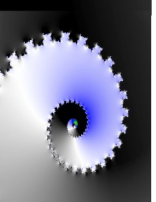
arno.haase@haase-consulting.com

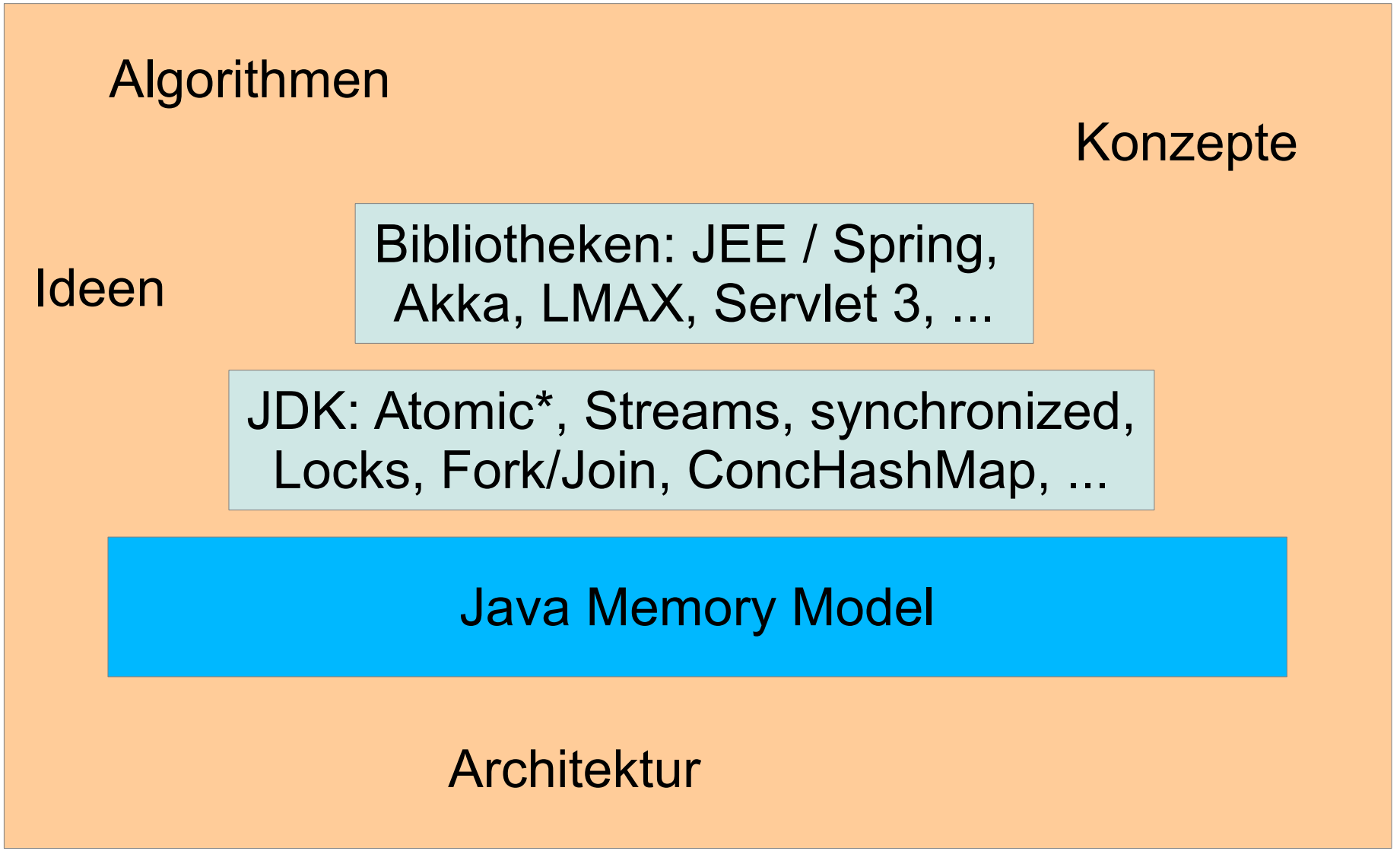
We should forget about small efficiencies, say about 97% of the time:

> Premature Optimization is the root of all evil.

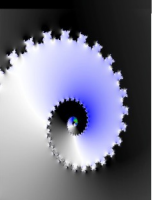
Yet we should not pass up our opportunities in that critical 3%.

Donald Knuth



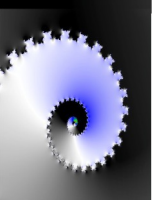

Algorithmen
Konzepte
Ideen
Bibliotheken: JEE / Spring,
Akka, LMAX, Servlet 3, ...
JDK: Atomic*, Streams, synchronized,
Locks, Fork/Join, ConcHashMap, ...
Java Memory Model
Architektur

1. Java Memory Model
2. Konzepte und Paradigmen
3. Performance

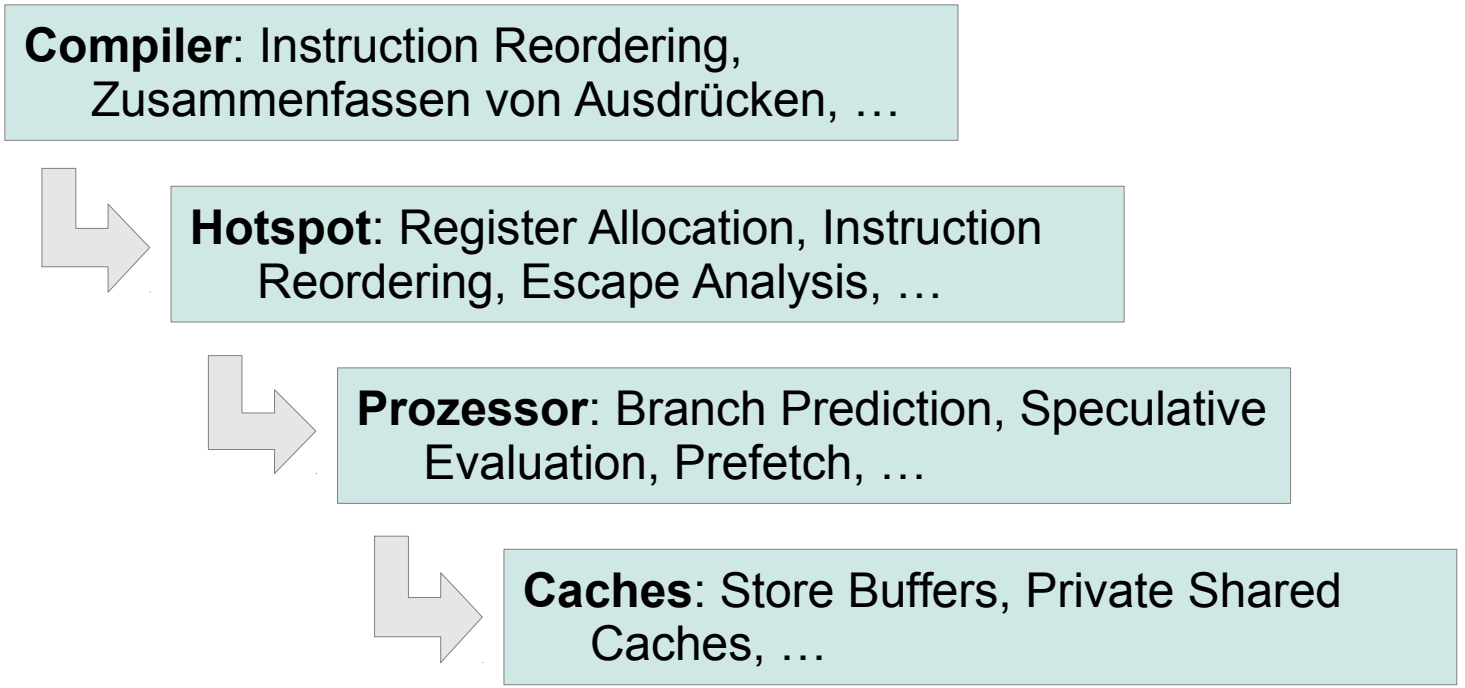


Tut Dein Computer,
was Du programmiert hast?

☐ ja ☒ nein



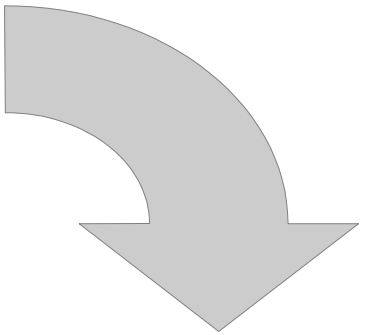
Transformationen



Die sichtbaren Auswirkungen sind auf allen Ebenen gleich.

Beispiel: Abbruchbedingung

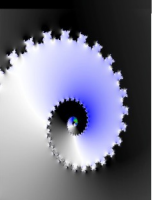
```
boolean shutdown = false;
…
void doIt() {
  while (!shutdown) {
    …
  }
}
```



```
boolean shutdown = false;
…
void doIt() {
  boolean b = shutdown;
  while (!b) {
    …
  }
}
```

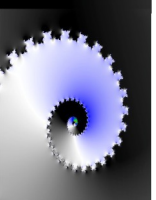
Threads (naiv)

- Threads arbeiten abwechselnd.
- Jeder Thread tut, was im Quelltext steht.
- Wenn er unterbrochen wird, sehen andere Threads den Zwischenzustand.
- Synchronisation dient dazu, Änderungen atomar zu machen.



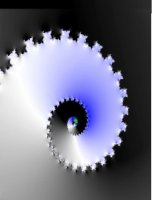
Threads (etwas weniger naiv)

- Es gibt mehrere CPUs, die Threads wirklich gleichzeitig abarbeiten.
- Jeder Thread tut, was im Quelltext steht.
- Datenzugriffe gehen ins RAM.
- Wenn Threads auf die selben Daten zugreifen, ist das automatisch nach einander.



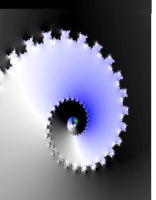
Die Wahrheit

- Innerhalb eines Threads sieht es aus, *als ob* er den Quelltext ausführen würde.
- Dinge in verschiedenen Threads passieren in einer definierten Reihenfolge, wenn die Synchronisation das vorschreibt („happens-before“).
- *Und nicht mehr!*



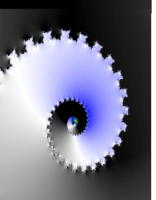
Korrekte Synchronisation

- Keine Race Conditions:
  - Wenn mehrere Threads auf eine Variable zugreifen
  - und mindestens einer schreibt,
  - müssen sie über „happens-before“ geordnet sein.
- Dann läuft das Programm, *als ob*
  - alle Speicherzugriffe sequentiell passieren,
  - tatsächlich auf RAM zugreifen,
  - und zwar in der „happens-before“-Reihenfolge



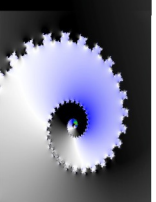
Beispiel: volatile

- Wenn
  - Thread T1 eine volatile-Variable *v* schreibt
  - Thread T2 anschließend *v* liest
- Dann
  - sind alle Änderungen aus T1 bis zum Zugriff auf *v* 'vor' dem Zugriff in T2

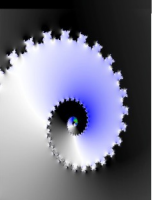


Memory Barriers

- Hilfsmittel zur Implementierung des JMM
  - Assembler-Befehle
  - „synchronisieren“ CPUs und Caches
  - Begrenzen Reordering
- Teuer!
  - Direkte Kosten: Cache-Flush
  - Begrenzen Optimierungen
- Wo setzt Java sie?
  - synchronized, Locks
  - volatile
  - nach Konstruktor (für final-Attribute), …

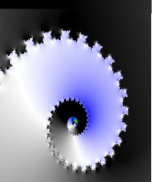


Concurrency
ist
komplex!



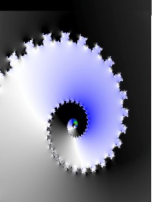
Ein einfacher Logger

```
public class Logger {
  public void log (String msg, Object... params) {
    String s = doFormat (msg, params);
    doLog (s);
  }

  private String doFormat (String msg, Object... params) {…}
  private void doLog (String msg) {…}
}
```



thread-sicher?

```
public class Logger {
  public synchronized void log (String msg, Object... params) {
    String s = doFormat (msg, params);
    doLog (s);
  }

  private String doFormat (String msg, Object... params) {…}
  private void doLog (String msg) {…}
}
```



versteckte Deadlocks

```
public class X {
  public synchronized void doIt () {
    log.log ("doing it");
    …
  }
  public synchronized String toString () { … }
  …
}
```

```
x.doIt();
```

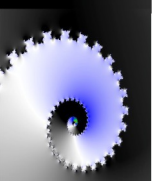
lock (x)

log.log("...") → lock (log)

```
log.log ("%s", x);
```

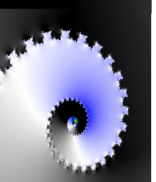
lock (log)

x.toString() → lock (x)



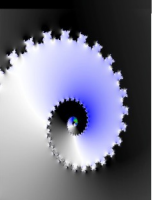
Asynchrones Logging

```
final ExecutorService exec =
                    Executors.newSingleThreadExecutor();

public void log (String msg, Object... args) {
  exec.execute (() -> {
    String formatted = doFormat (msg, args);
    doLog (formatted);
  });
}
```



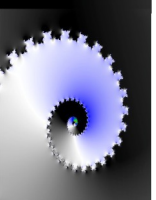
Parallelisieren?

- Formatieren kann teuer sein
  - toString(): Callbacks in Anwendungscode!
- Thread-Pool zum Logging?
  - Reihenfolge der Nachrichten geht verloren!
  - … und doLog ist nicht thread-sicher



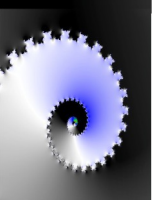
Future<String>

```
final ExecutorService exec =
                    Executors.newSingleThreadExecutor();

public void log (String msg, Object... args) {
  final Future<String> formatted =
      CompletableFuture.supplyAsync(() -> doFormat(msg, args));
  exec.execute (() -> {
    try {
      doLog (formatted.get());
    }
    catch (Exception exc) {
      exc.printStackTrace();
    }
  });
}
```



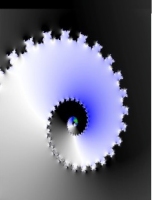
Concurrency: Shared Mutable State

- Locks: Blocking
  - Read/Write und andere Optimierungen
- *Ein* Worker-Thread mit Message-Queue: Non-Blocking
  - Actor als Variante
- Auch „Lock-Frei“ hat shared state
- Jeder Ansatz kostet!



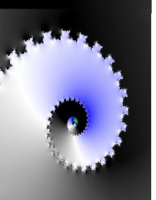
Worker Threads

- Queues
  - Bounded / Unbounded
  - Blocking / Non-Blocking
  - Single / Multi Producers / Consumers
  - Optimiert für Lesen oder Schreiben (oder Mix)
  - Sonder-Features: Priority, remove(), Batch, …
- Future für Ergebnisse
  - .thenAccept(...) / .thenAcceptAsync(...)

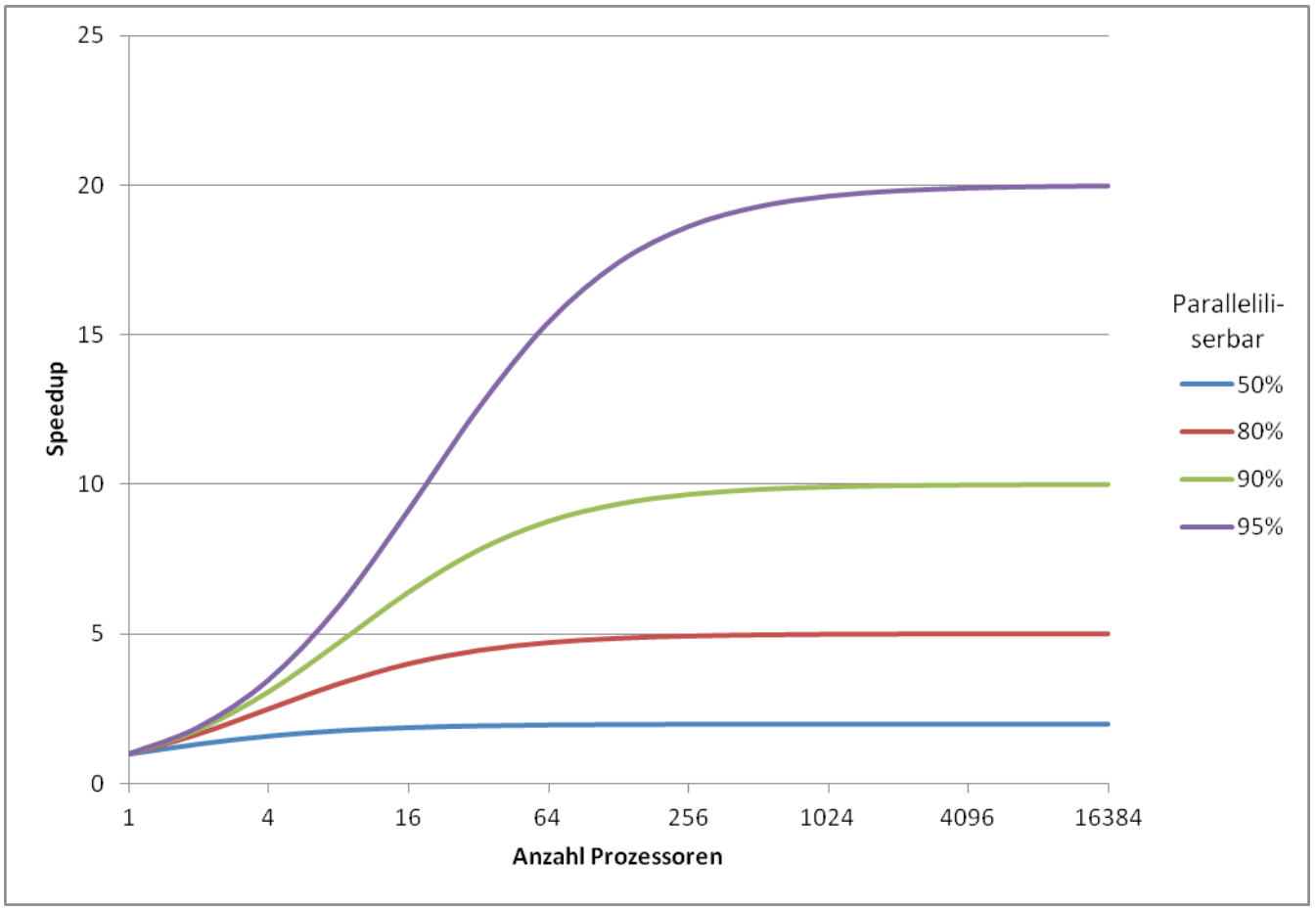


Welcher Thread-Pool?

- mit Seiteneffekten
  - (meist) Reihenfolge wichtig → Executors.newSingleThreadPool()
- non-blocking ohne Seiteneffekte
  - ForkJoinPool.commonPool()
- blocking
  - Eigenen ExecutorService je Kontext
  - Tuning, Monitoring, …
- Größere Probleme zerlegen und aufteilen
  - Messen, ob das hilft!
- *Nicht* ad hoc!

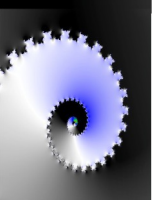


Amdahl's Law


25
20
15
10
5
0
Speedup
1
4
16
64
256
1024
4096
16384
Anzahl Prozessoren
Parallelili-
serbar
50%
80%
90%
95%

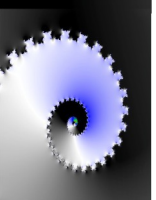
Geteilte Ressourcen zwingen zum Warten!

- genauer: *veränderliche* Ressourcen
  - Locks, I/O, …
- Verstecktes Sharing
  - ReadLock: Counter
  - UUID.randomUUID()
  - volatile: Geteilter Hauptspeicher
  - Cache Lines (Locality, Poisoning)
  - Festplatten, DVDs, Netzwerk
  - …
- Lösungen: Isolation oder Immutability



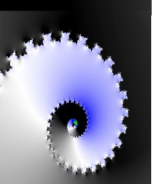
Lock-freie Programmierung

- Shared Mutable State
- Lock Free
  - Aufrufer müssen nie warten
  - Abarbeitung kann sich aber verzögern
  - z.B. asynchron entkoppelt
  - Responsive, schont Ressourcen
- Wait Free
  - Abarbeitung verzögert sich nicht
  - Spezielle Algorithmen und Datenstrukturen
  - Extremst schwierig; Grundlagenforschung, Work in Progress
  - ConcurrentHashMap, ConcurrentLinkedQueue



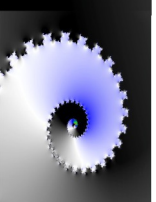
CAS-Schleife

```
final AtomicInteger n = new AtomicInteger (0);

int max (int i) {
  int prev, next;
  do {
    prev = n.get();
    next = Math.max (prev, i);
  } while (! n.compareAndSet (prev, next));
  return next;
}
```



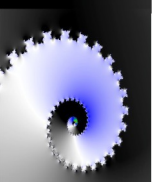
Funktionale Programmierung

- *Ohne* Seiteneffekte
  - Alle Objekte sind immutable, bei Änderung neues Objekt
  - != Verwendung von Lambdas: JDK-Collections, Guice, …
  - Scala, Clojure; a-base
- anderer Programmierstil
  - effizientes Kopieren: teilweise Wiederverwendung
  - funktionale Algorithmen
- Automatisch stabiler State, auch concurrent



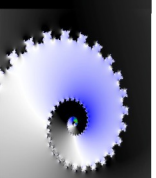
Performance-Tuning von Concurrency

```
public class StockExchange {
  private final Map<Currency, Double> rates = new HashMap<>();
  private final Map<String, Double> pricesInEuro = new HashMap<>();

  public void updateRate (Currency currency, double fromEuro) {
    rates.put (currency, fromEuro);
  }

  public void updatePrice (String wkz, double euros) {
    pricesInEuro.put (wkz, euros);
  }

  public double currentPrice (String wkz, Currency currency) {
    return pricesInEuro.get (wkz) * rates.get (currency);
  }
}
```



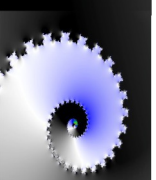
Wait-Free: ConcurrentHashMap

```
public class StockExchange {
  private final Map<Currency, Double> rates =
                                          new ConcurrentHashMap<>();
  private final Map<String, Double> pricesInEuro =
                                          new ConcurrentHashMap<>();

  …
}
```



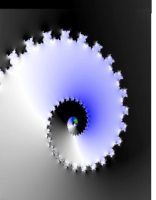
Feine Locks: Collections.synchronizedMap

```
public class StockExchange {
  private final Map<Currency, Double> rates =
                      Collections.synchronizedMap (new HashMap<>());
  private final Map<String, Double> pricesInEuro =
                      Collections.synchronizedMap (new HashMap<>());

  …
}
```



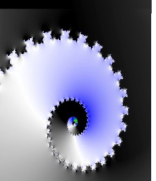
Grobe Locks

```
public class StockExchange {

  …

  public synchronized void updateRate (…) {
    …
  }

  public synchronized void updatePrice (…) {
    …
  }

  public synchronized double currentPrice (…) {
    …
  }
}
```



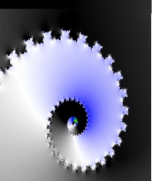
Funktional: Immutable Maps

```
public class StockExchange {
  private final AtomicReference<AMap<Currency, Double>> rates =
                              new AtomicReference<> (AHashMap.empty ());
  …

  public void updateRate (Currency currency, double fromEuro) {
    AMap<Currency, Double> prev, next;
    do {
      prev = rates.get ();
      next = prev.updated (currency, fromEuro);
    }
    while (! rates.compareAndSet (prev, next));
  }

  …
}
```



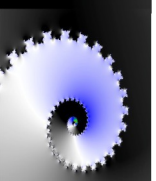
Variante: reduzierte Update-Garantie

```
public class StockExchange {
  private volatile AMap<Currency, Double> rates =
                                                AHashMap.empty ();
  …

  public void updateRate (Currency currency, double fromEuro) {
    rates = rates.updated (currency, fromEuro);
  }

  …
}
```



Queue mit Worker-Thread

```
public class StockExchange {
  private final Map<Currency, Double> rates = new HashMap<>();
  …

  private final BlockingQueue<Runnable> queue = …;

  public StockExchange() {
    new Thread(() -> {
      while (true) queue.take().run();
    }).start();
  }

  public void updateRate (Currency currency, double fromEuro) {
    queue.put (() -> rates.put (currency, fromEuro));
  }

  …
}
```

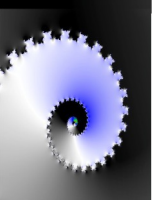


Vergleichstest

```
for (int i=0; i<1_000_000; i++) {
  stockExchange.updatePrice ("abc", 1.23);
}
```

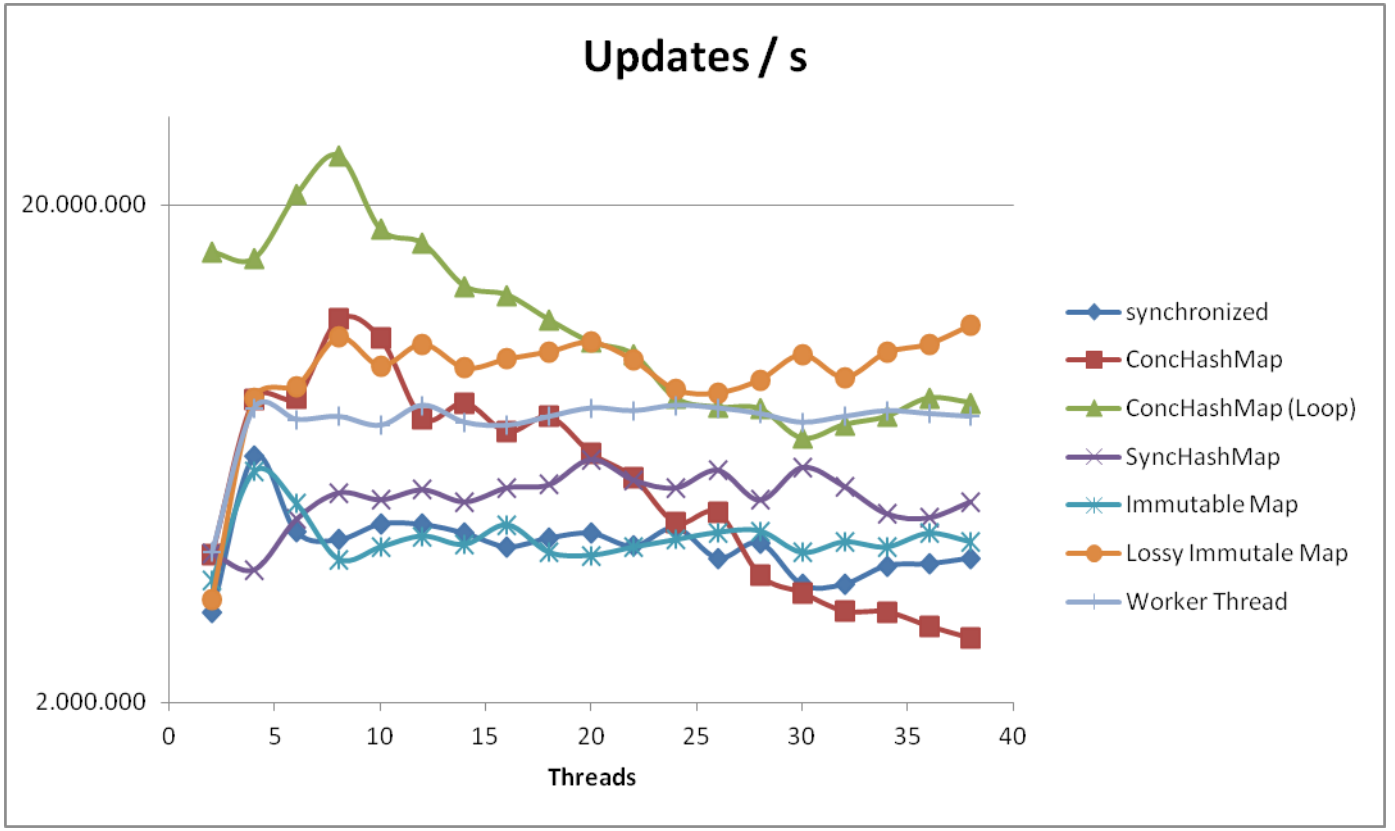
```
volatile int v=0;
…
for (int i=0; i<1_000_000; i++) {
  v=v;
  stockExchange.updatePrice ("abc", 1.23);
}
```

```
volatile int v=0;
final LinkedList<String> l = new LinkedList<>();
…
for (int i=0; i<1_000_000; i++) {
  l.add ("abc");
  v=v;
  stockExchange.updatePrice (l.remove(), 1.23);
}
```

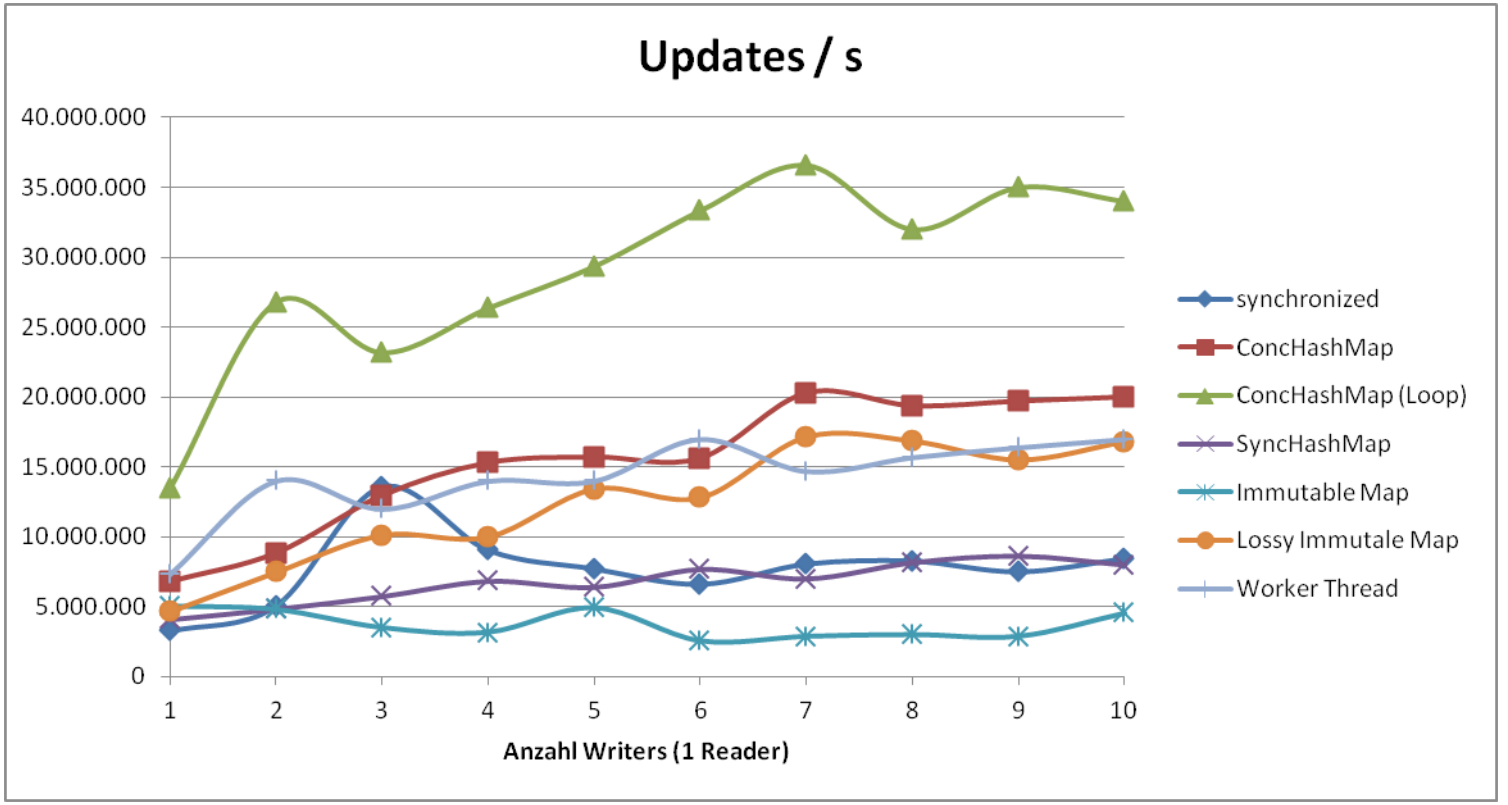


Gemischte Last

Gemischte Last

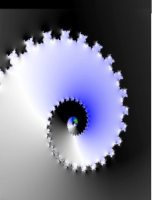


Update-Last


Updates / s
40.000.000
35.000.000
30.000.000
25.000.000
20.000.000
15.000.000
10.000.000
5.000.000
0
1
2
3
4
5
6
7
8
9
10
Anzahl Writers (1 Reader)
synchronized
ConcHashMap
ConcHashMap (Loop)
SyncHashMap
Immutable Map
Lossy Immutale Map
Worker Thread

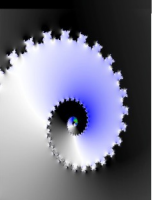
And the Winner is...

- ConcurrentHashMap
  - keine Konsistenz, primär Reads, algorithmische Zugriffe
- Queue mit Worker Thread
  - Transaktionaler Zugriff, zentrale Event-Queue
- Immutable Map
  - Stabile Daten während Read, primär Reads
  - lang laufende Reads
  - „lossy“ → auch schnelle Update
- Locks
  - nie besonders schnell → vorgegebenes Thread- und Datenmodell
  - Granularität: Lock je Operation



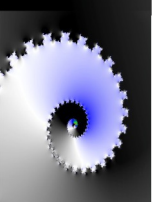
Testen (1): Korrektheit

- „es funktioniert“ reicht nicht
  - JMM vs. JVM, Hardware, …
- Reviews
- kontrollierte Unterbrechungen
- Shotgun



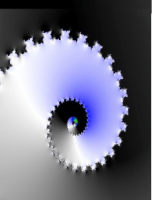
Testen (2): Performance

- Zeit einplanen!
- realistische Hardware
- große Hardware
  - Multi-Core vs. Multi-Prozessor → HW-Optimierung für Cache-Austausch
  - Skalierungseffekte
- realistische Last-Szenarien (→ kennen!!! Annahmen dokumentieren!!!)
  - Virtualisierte Hardware
- Konkrete Fragen stellen - viele Stellschrauben
  - Pool-Größen, Cut-Off für serielle Verarbeitung
  - HW-Größen: RAM, #Cores, …
  - Messreihen für alle Alternativen



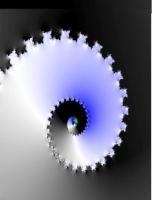
Praxis: Lokale Parallelisierung

- z.B. große Collection durchsuchen
  - Fork/Join-Beispiele
  - Stream-API
- Gewinne überprüfen
  - einfache APIs, laden zu „ad hoc“-Verwendung ein
  - ohne Grundlast wirkt es oft schnell
  - *vergrößert* in der Summe die CPU-Last – Vorteile nur bei CPU-Reserven. Mehr Kontext-Wechsel!
  - Messen: reale Hardware, reale Lastszenarien



Fazit

- Concurrency ist schwierig
- Probleme genau verstehen
- Messen, messen, messen!
- OS und Hardware haben qualitativ Einfluss
- Korrektheit vor Performance
- Möglichst grobe Concurrency
- Share Nothing



The End

- Links:
  - http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2
  - http://github.com/arnohaase/a-base