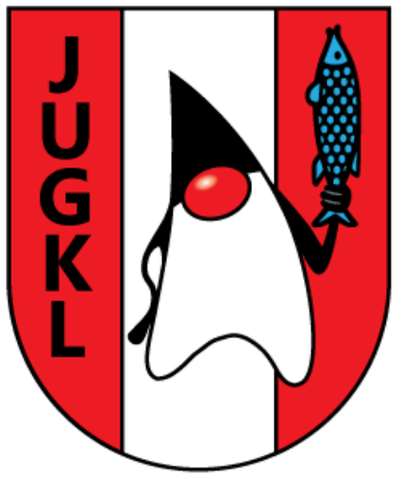
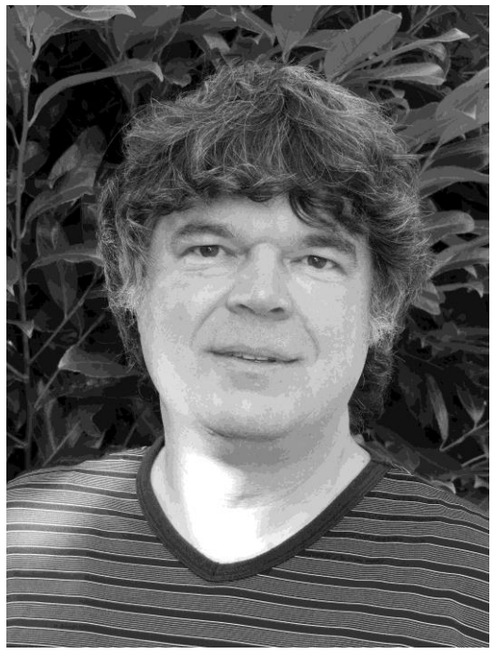


Hochschule Kaiserslautern
University of Applied Sciences

Java Concurrency-Konzepte

Jörg Hettel

Hochschule Kaiserslautern



Prof. Dr. Jörg Hettel

Hochschule Kaiserslautern
Campus Zweibrücken
Fachbereich Informatik

eMail: joerg.hettel@hs-kl.de

Hochschule Kaiserslautern
University of Applied Sciences

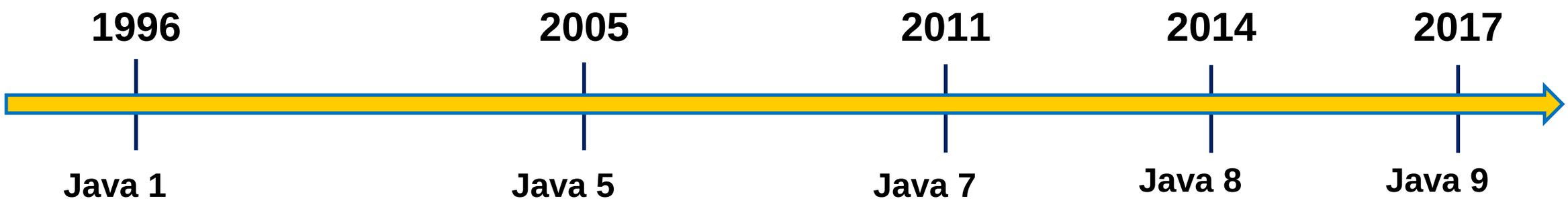
Lehr- und Interessensgebiete:

- Multicore-Programmierung
- Software-Architekturen
- Machine Learning
- Quantum Computing

Agenda

- Concurrency-Tools
- Parallelisierungsframeworks
  - **Parallel Streams**
  - **CompletableFuture**
- Java 9: *Reactive Streams*
  - **Datenflussorientierte Parallelverarbeitung**

Multicore-Features bei Java



Memory Modell
Klasse/Interface:
- **Thread,**
- **Runnable**

Schlüsselwörter:
- **synchronized**
- **volatile**

Methoden:
- **wait**
- **notify, notifyAll**

Überarbeitung des **Memory Modells**
Concurrency-Tools:
- **j.u.c-**Package

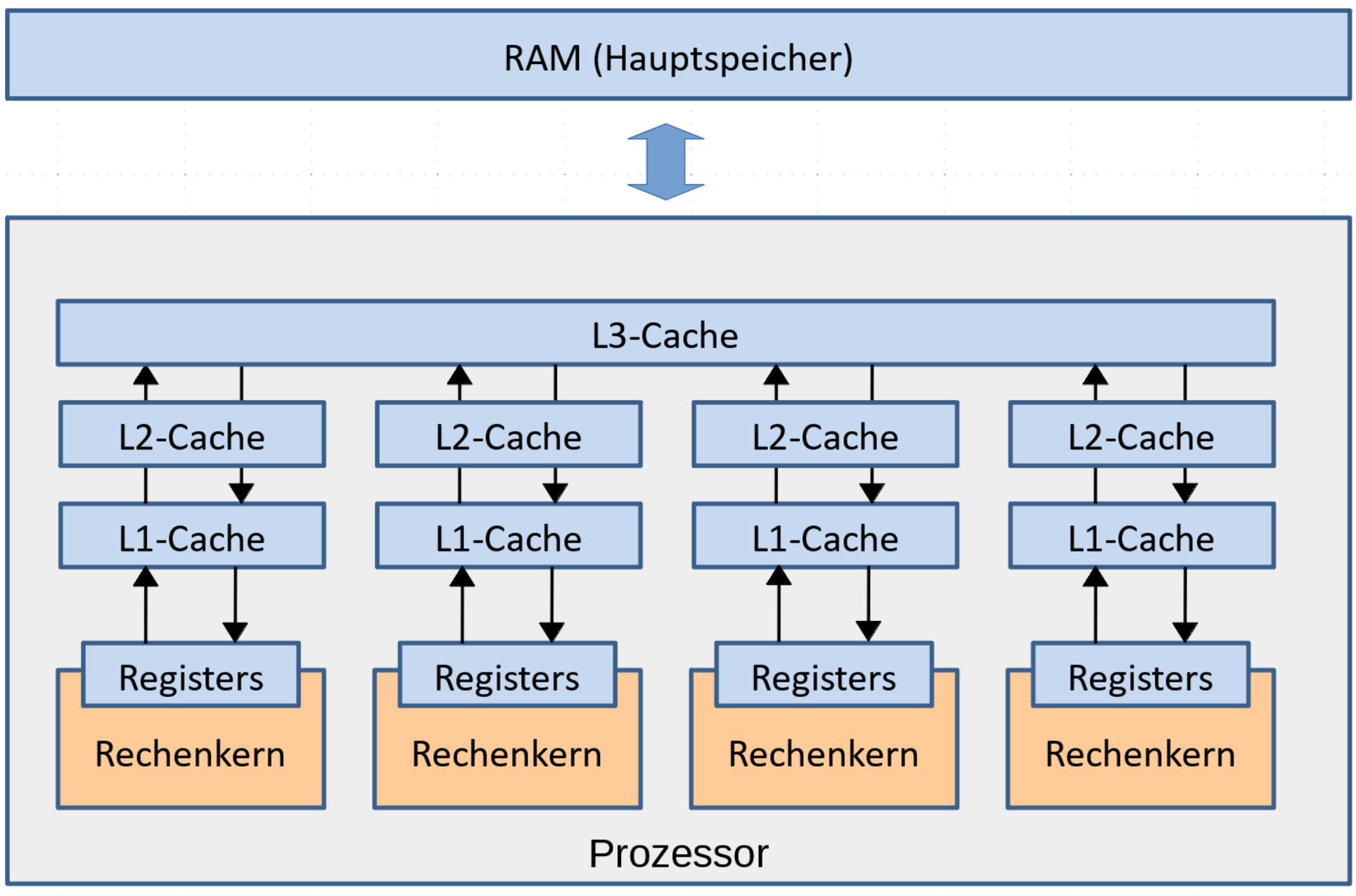
ForkJoin-Framework

Streams, Completable-Future

Flow-Klasse

Parallelisierungsframeworks

Es geht um: *Shared Memory Parallel Programming*



Hochschule Kaiserslautern
University of Applied Sciences

Concurrency-Tools

Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

Lock-Abstraktionen

- Ergänzung zum **synchronized**-Konzept
- Einführung verschiedener Lockarten:
  - **ReentrandLock**
    - kann faire Lockvergabe, unterbrechbares Warten
  - **ReadWriteLock**
    - Zusätzliche Unterscheidung zwischen Schreib- und Lesesperren
  - **StampedLock**
    - Zusätzlich optimistisches Read-Locking
- Definition beliebig vieler Bedingungsvariablen auf den Lockobjekten

Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

Synchronisationsabstraktionen

- Zur Koordinierung von mehreren Threads
  - Ressourcenverwaltung
    - **Semaphore**
  - Barrieren
    - **CountDownLatch**
    - **CyclicBarrier**
    - **Phaser**

Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

Atomic-Klassen

- Atomic-Klassen repräsentieren Variablen, die direkt „*compare and swap*“ – Operationen unterstützen
- Verschiedener Klassen:
  - **AtomicInteger**
  - **AtomicLong**
  - **AtomicReference, AtomicMarkableReference, AtomicStampedReference**
  - **AtomicAdder**
  - **…**
- Ermöglicht z.B. Entwicklung lockfreier Datenstrukturen

Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

Threadsichere Datenstrukturen

- Die „alten“ Standard-Collections sind nicht Threadsicher, z.B. keine parallele Lese- und Schreibzugriffe erlaubt
- „Parallele“ Arrays (Java 8)
  - Methoden: **parallelSort, parallelSetAll, parallelPrefix**
- Für Multithread-Anwendungen gibt es nun sichere Container:
  - **ConcurrentHashMap**
  - **CopyOnWriteArrayList**
  - **BlockingQueue, BlockingDeque…**
  - **…**

Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

Executoren bzw. Threadpools

- Einführung des **Future**-Pattern, d.h. asynchron ausgeführte Tasks mit Rückgabe
- Vorkonfigurierte Threadpools (Factory-Klasse **Executors**)
  - *Cached-, Fixed-, Scheduled-, ForkJoin-Threadpool* (*Common Pool*)
- Frei konfigurierbare Pools über Konstruktoraufruf

Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

Verschiedene abstrakte Konzepte zur Koordination und Synchronisation paralleler Threads

- **Für „Entwickler“ von Frameworks und Bibliotheken**
  - Locks zur Koordination des parallelen Zugriffs auf Ressourcen
  - Synchronisationsbarrieren
  - Unterstützung von atomaren Schreib- und Leseoperationen
- **Für „Anwendungsentwickler“**
  - Threadsichere Datenstrukturen
  - Threadpools (*Executoren*) zum einfachen Umgang mit Threads und Future-Unterstützung

Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

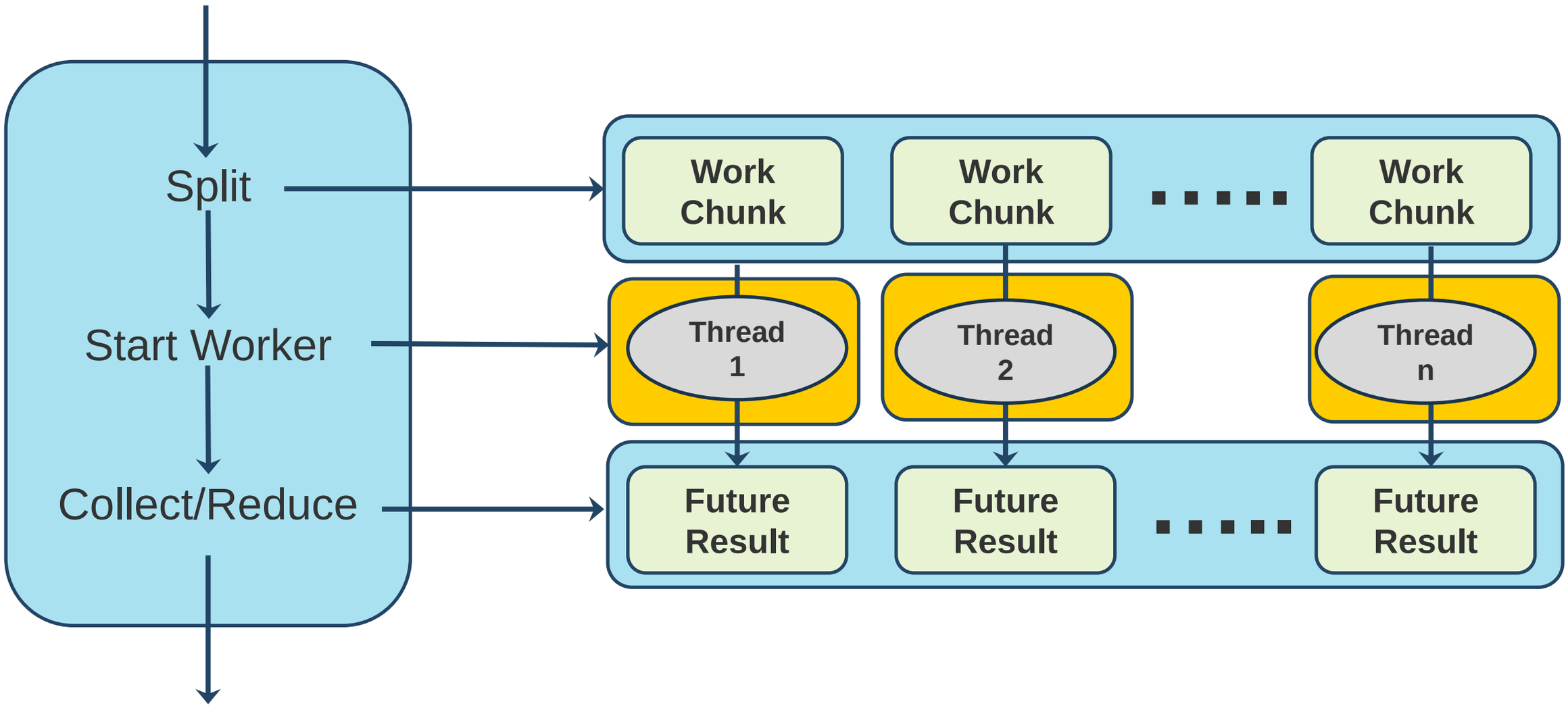
- Parallele Abläufe können mit den Concurrency-Tools „einfacher“ implementiert werden

- Es ist aber noch viel „Handarbeit“ notwendig!
  - Beispiel: Typisches Vorgehen für das ***„Parallel-For“-Pattern***
    - Task-Klasse für die „Work“ erstellen
    - Master muss Arbeit auf Taskobjekte verteilen
    - Taskobjekte werden einem Threadpool (Executor) übergeben
    - Master muss die Ergebnisse einsammeln und das Endergebnis ermitteln

Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

Beispiel: **„Parallel-For“** mit *Master-Worker-Pattern*



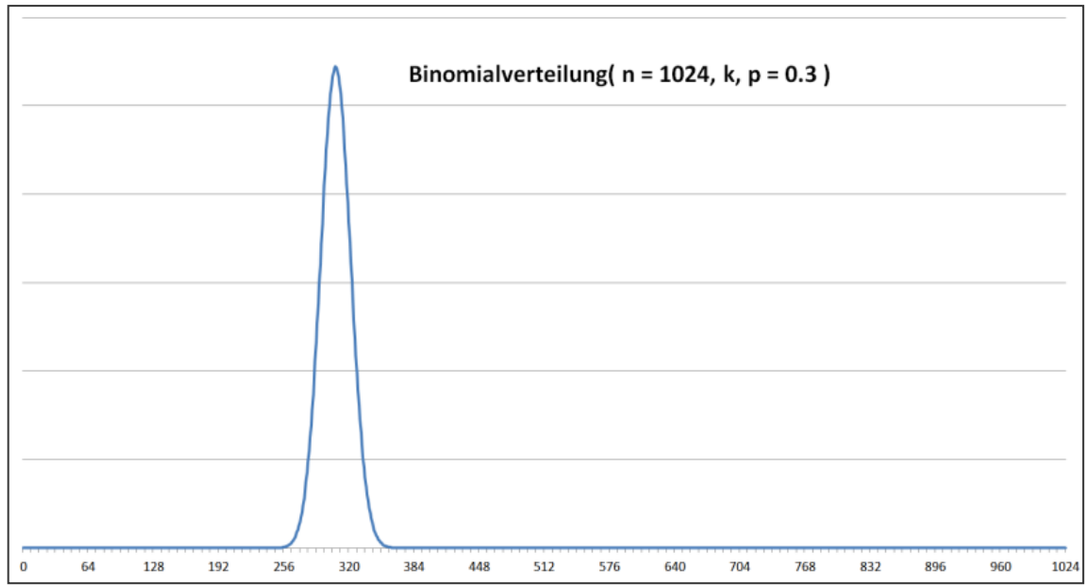
Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

Beispiel: Berechnung der Summe einer Binomialverteilung

$$Bin(n,k:p) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\sum_{k=0}^{n} Bin(n,k;p) = 1$$



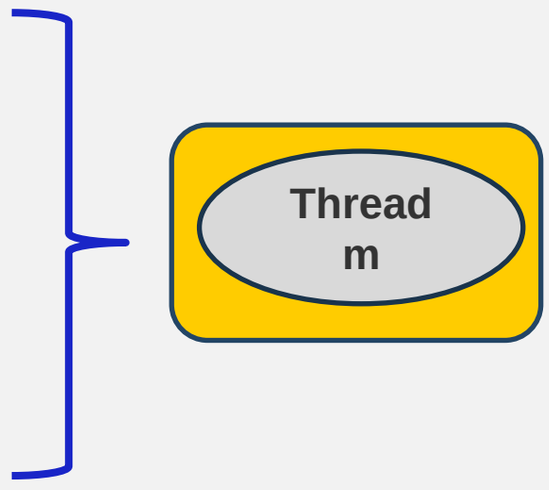

```
final int n = 1024;
final double p = 0.3;

BigDecimal res = BigDecimal.ZERO;
for(int k=0; k<=n; k++)
{
  res = res.add( Bin(N, k, p) );
}
```

Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

Task-Klasse (Worker)

```
class MasterWorkerTask implements Callable<BigDecimal>
{
 // Attribute

 private MasterWorkerTask(int start, int end, int n, double p)
 {
   ...
 }

 @Override
 public BigDecimal call() throws Exception
 {
   BigDecimal res = BigDecimal.ZERO;
   for (int k = start; k < end; k++)
   {
     res = res.add( Bin(n, k, p) );
   }
   return res;
 }
}
```



Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

Master-Klasse

```
ExecutorService executor = Executors.newCachedThreadPool();

final int n = 1024;
final double p = 0.03;
int numChunks = ...;

int chunkSize = n/numChunks;
List<MasterWorkerTask> tasks = new ArrayList<>();
for(int i=0; i < numChunks; i++ ) {
  int from = i*chunkSize;
  int to = (i == numChunks-1) ? n+1 : (i+1)*chunkSize;
  MasterWorkerTask task = new MasterWorkerTask( from, to, n, p );
  tasks.add( task );
}

List<Future<BigDecimal>> futures = executor.invokeAll(tasks);

BigDecimal res = BigDecimal.ZERO;
for( Future<BigDecimal> f : futures) {
   res = res.add( f.get() );
}

executor.shutdown();
```

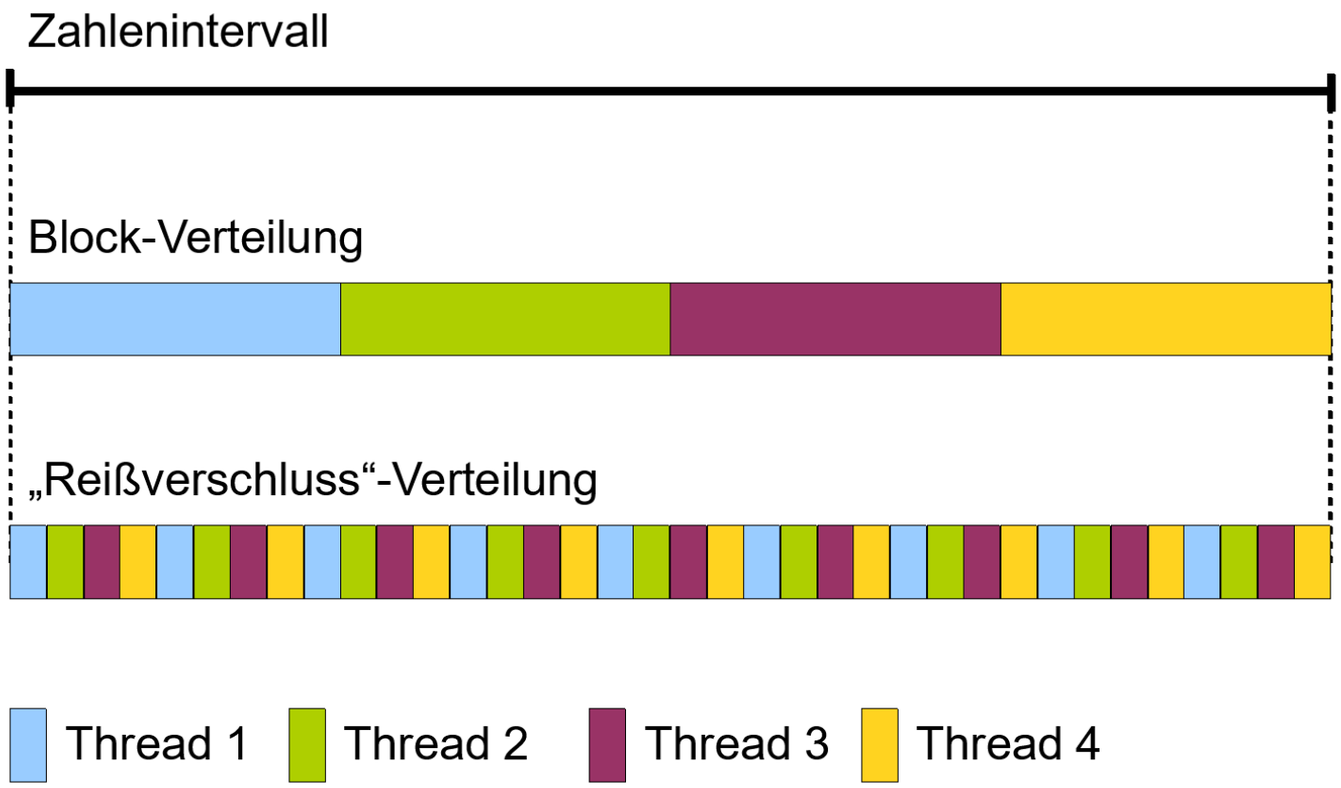
Split-Phase

Parallele Work-Phase

Collect / Reduce - Phase

Concurrency-Tools

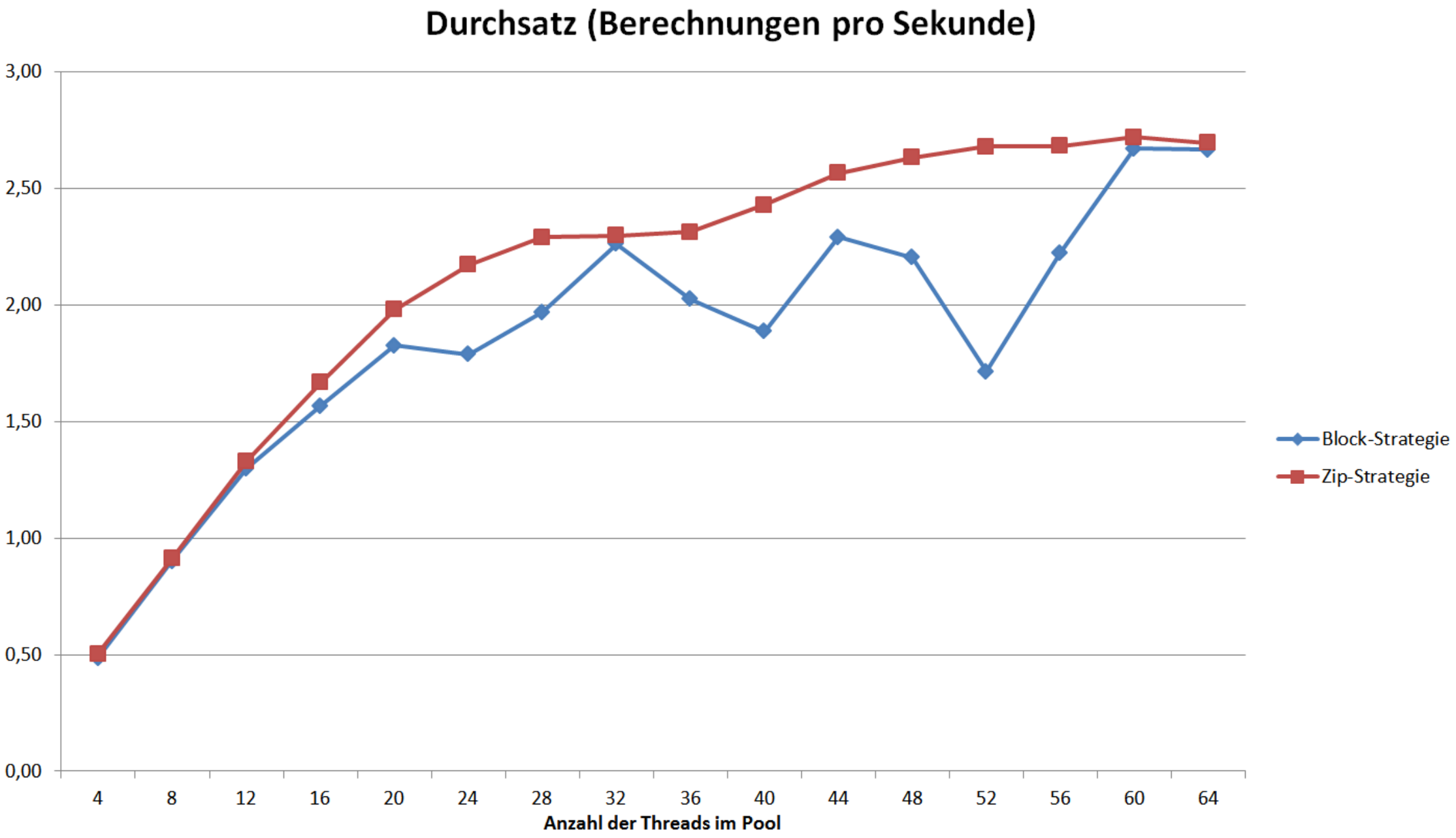
- Bemerkungen
  - Hoher Verwaltungsaufwand, der sequentiell abgearbeitet wird
    - Reduktion des maximalen Speedups (Amdahl´sches Gesetz)
  - Datenbereich kann verschieden portioniert werden



Concurrency-Tools

Hochschule Kaiserslautern
University of Applied Sciences

- Laufzeitvergleich



Frameworks für die Parallelisierung

Einsatz: Beschleunigung von Anwendungen

Streams

Streams entsprechen einer Abstraktion für Datensammlungen

- **Deklarative Beschreibung der Verarbeitung**
  - Interne Iteration anstatt äußere
  - Beschreibung von wie und was ist hier getrennt!
  - Nur noch Beschreibung von dem was gemacht werden soll:

```
streamOfStrings.forEach( ... );
```

```
str -> System.out.println(str)
```

Arbeiten mit Streams

Streams

- Verarbeitung von Datensammlungen, wie z.B. Collections
- Entspricht einer „Pipeline“-Verarbeitung

- aus Collections
- aus Iteratoren
- aus Generatoren
- ...

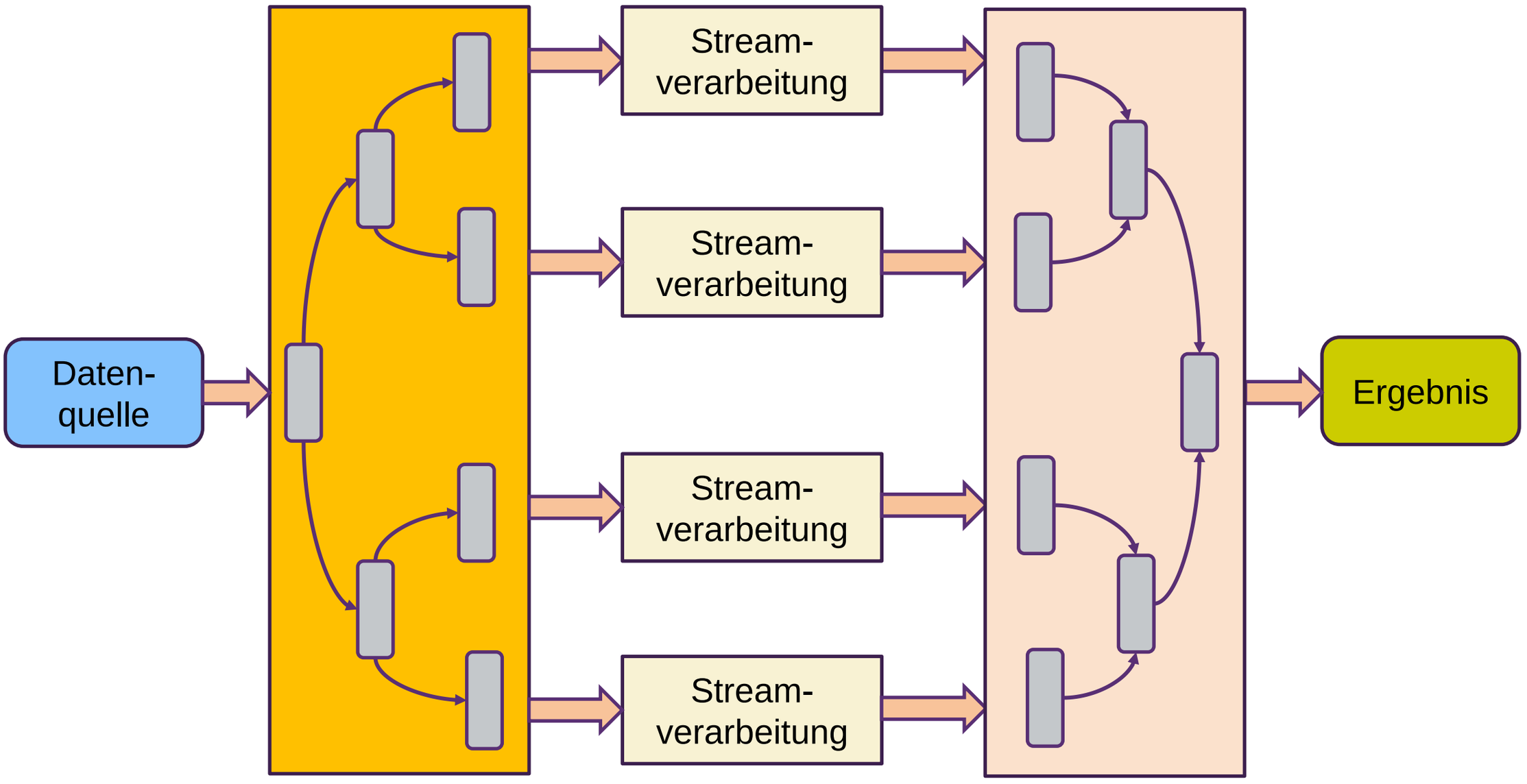
- map(…)
- filter(…)
- flatMap(…)
- peek(…)
- distinct()
- sorted()
- ...

- forEach(…)
- findAny(…)
- collect(…)
- reduce(…)
- ...

Parallelisierung mit Streams

Parallele Streams (Fork/Join-Konzept)

- ForkJoin-Framework kann auch direkt genutzt werden



Bemerkungen zu *Parallel*-Streams

- Explizite Steuerung des Split- und Reduce/Collect-Prozess:
  - Benutzerdefinierte Spliteratoren bzw. Collectoren
- Stream-Verarbeitung muss „seiteneffektfrei“ sein
  - Manipulationen (Lambdas) müssen „Parallel-Ready“
- Das Speicherlayout kann eine große Rolle spielen
- Parallelisierung lohnt sich nur
  - wenn Datenquelle effizient gesplittet werden kann
  - wenn effizient „reduziert“ bzw. „gesammelt“ werden kann
  - wenn genügend Arbeit (Daten) vorhanden ist
    - N*Q-Formel von Brian Götz: N*Q > 10.000
      - N - Anzahl der Elemente
      - Q - Aufwand der „Verarbeitung“

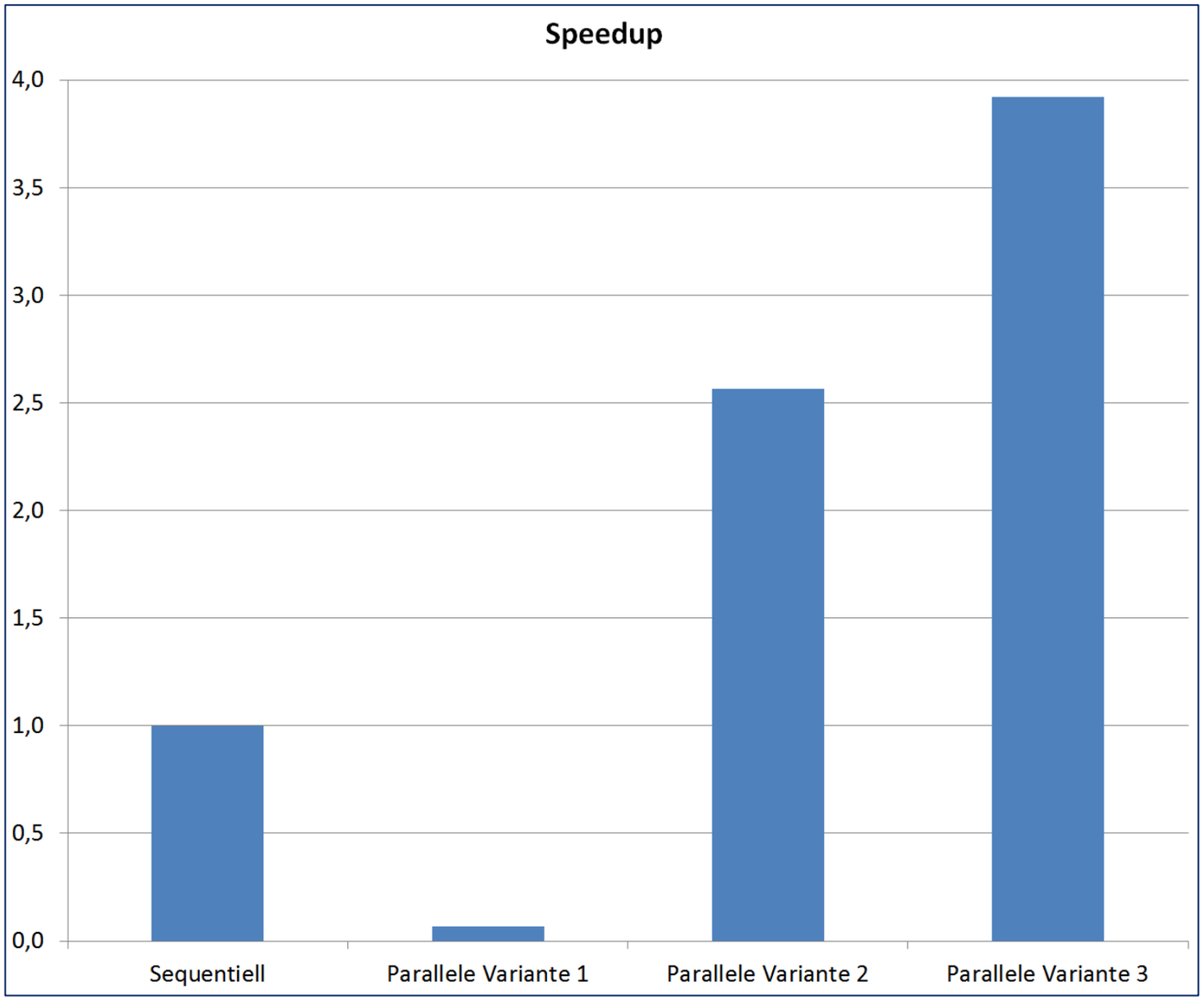
Beispiel: Paralleles befüllen eines Arrays

```
int[] array = ...;

// Variant 1 – keine gute Lösung !
Random rand = new Random();
Arrays.parallelSetAll(array, i -> rand.nextInt(100));

// Variante 2
ThreadLocal<Random> myThreadLocal =
                        ThreadLocal.withInitial(() -> new Random());
Arrays.parallelSetAll(array, i -> myThreadLocal.get().nextInt(100));

// Variante 3
Arrays.parallelSetAll(array, i ->
                            ThreadLocalRandom.current().nextInt(100));
```

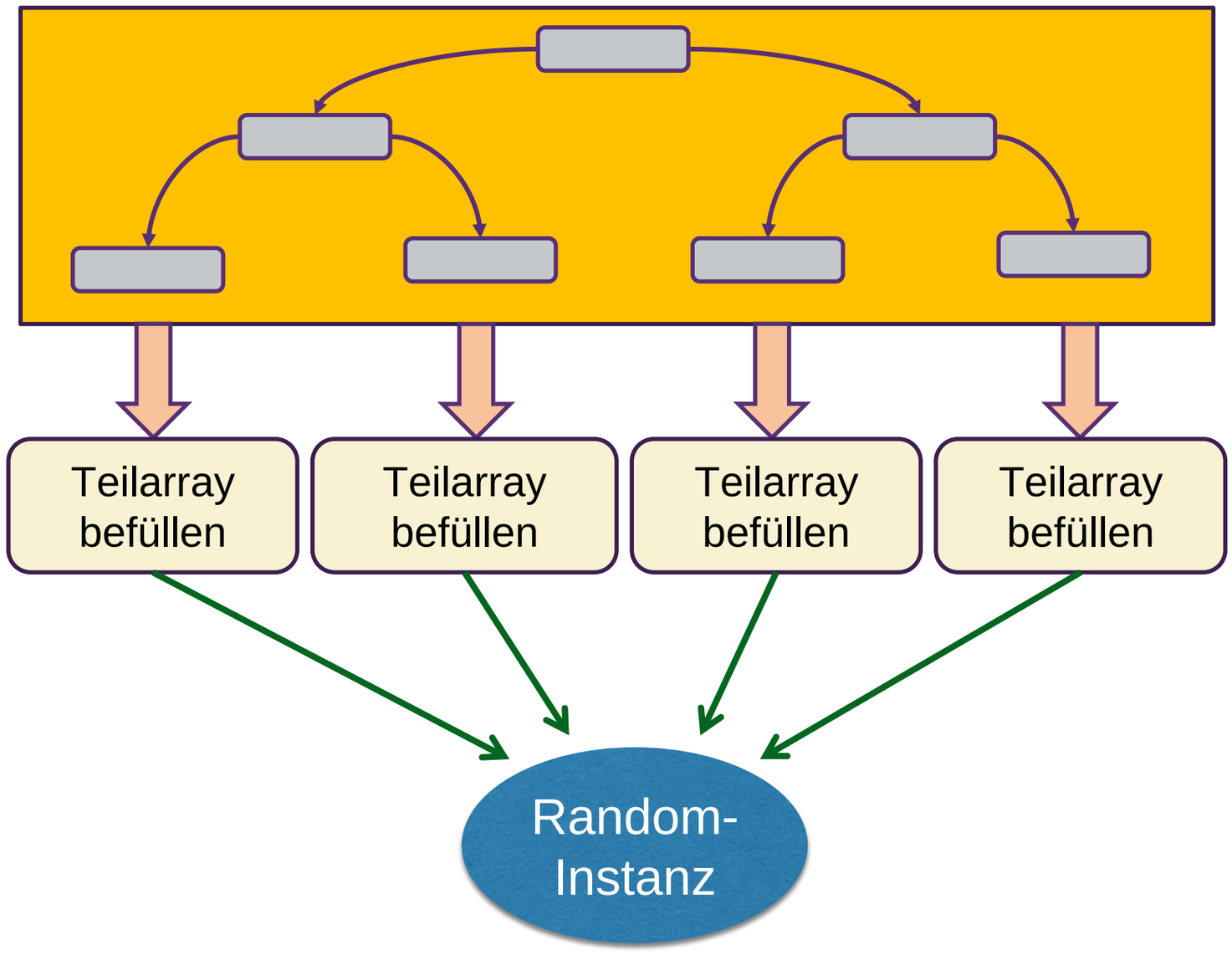
Beispiel: Paralleles befüllen eines Arrays

JMH-Benchmark auf einem Quadcore-Rechner:

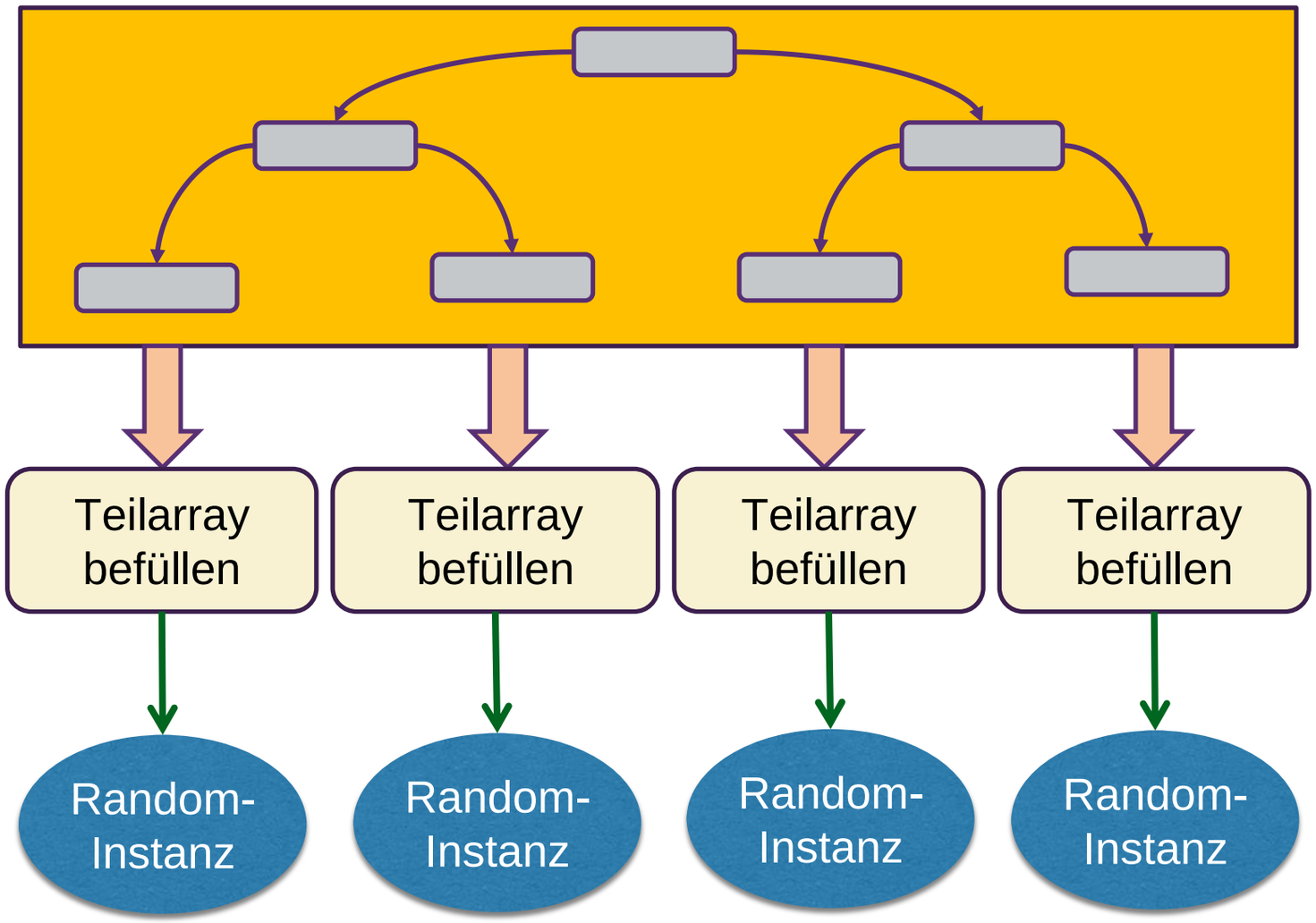
Befüllt wurde jeweils ein `int`-Array der Größe 100.000. Als Referenzwert für den Speedup wurde das sequentielle Befüllen mit `Arrays.setAll` benutzt.



Variante 1



Variante 2/3

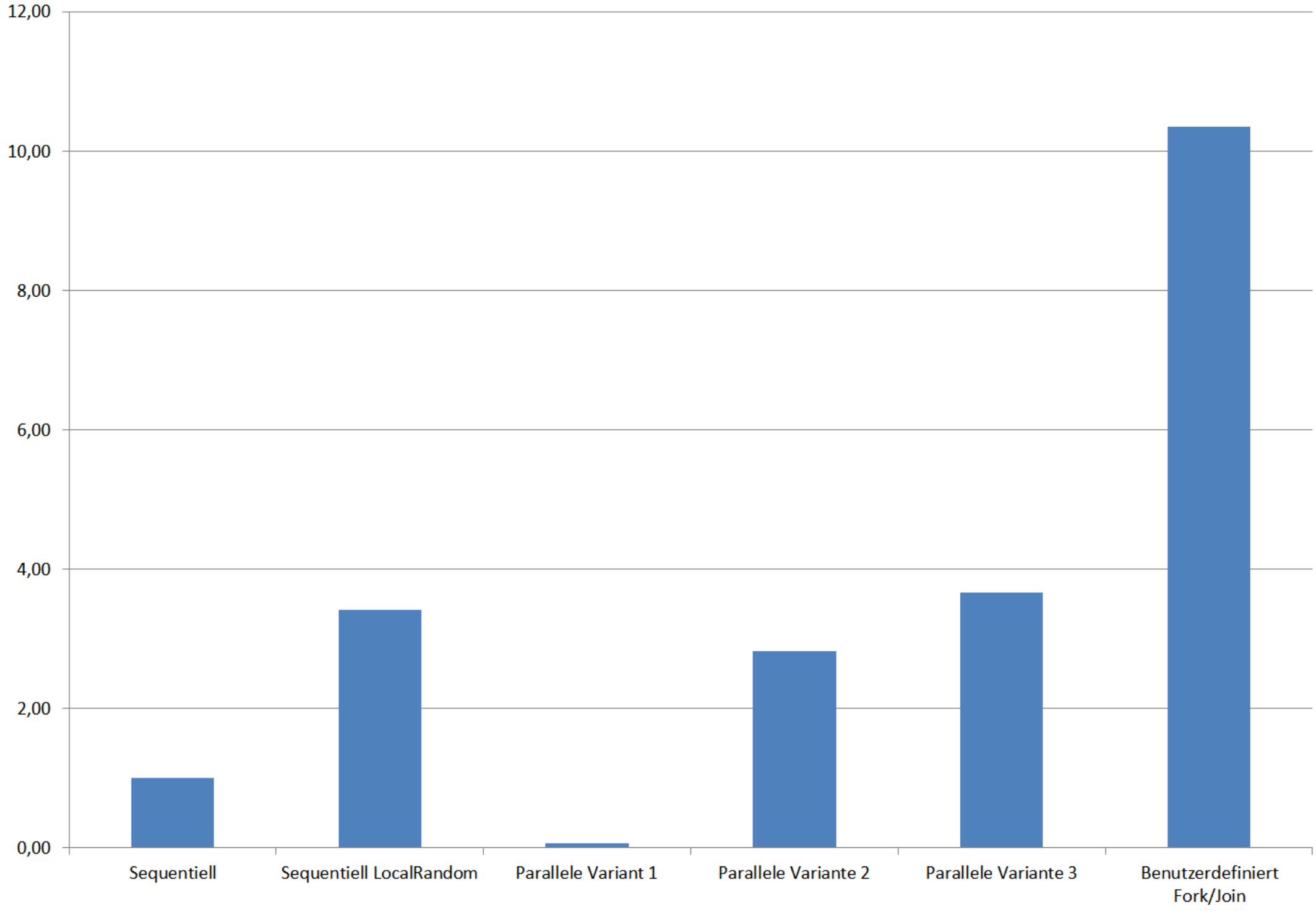


Variante 2 benutzt **synchronisierte Random-Instanz**, d.h. Änderungen müssen immer „sichtbar“ gemacht werden.

Variante 3 benutzt **unsynchronisierte** (thread-lokale) Random-Instanz.

Beispiel: Paralleles befüllen eines Arrays

Weitere Varianten



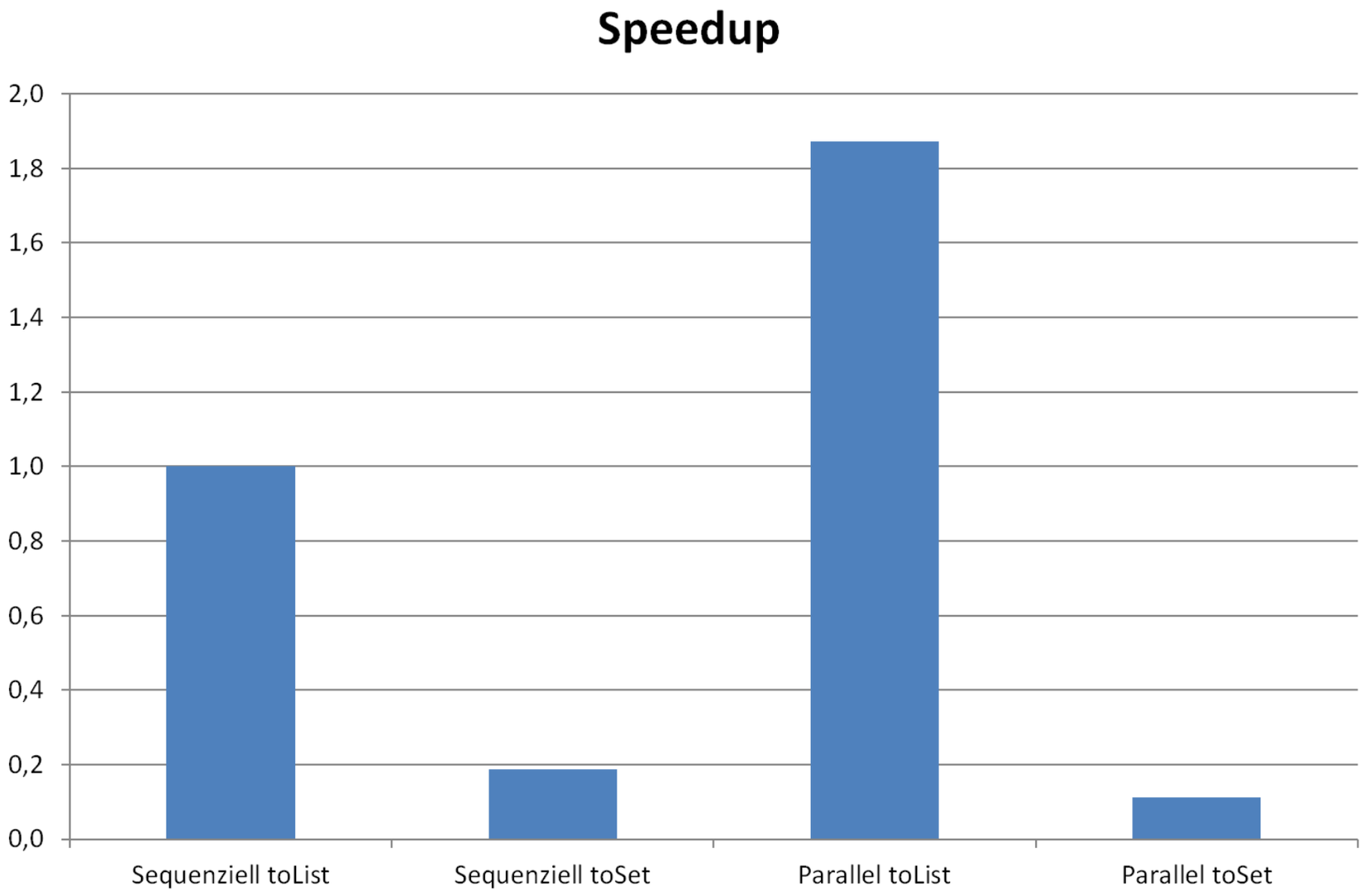
Beispiel: „Mutable Reduction“

```
List<Kunde> kunden = ...;

List<Kunde> seqToList = kunden.stream()
                              .filter( k -> k.getAlter() > 18 )
                              .collect(Collectors.toList());

Set<Kunde> seqToSet   = kunden.stream()
                              .filter( k -> k.getAlter() > 18 )
                              .collect(Collectors.toSet());

List<Kunde> parToList = kunden.parallelStream()
                              .filter( k -> k.getAlter() > 18 )
                              .collect(Collectors.toList());

Set<Kunde> parToSet   = kunden.parallelStream()
                              .filter( k -> k.getAlter() > 18 )
                              .collect(Collectors.toSet());
```

Beispiel: „Mutable Reduction“

JMH-Benchmark auf einem Quadcore-Rechner:

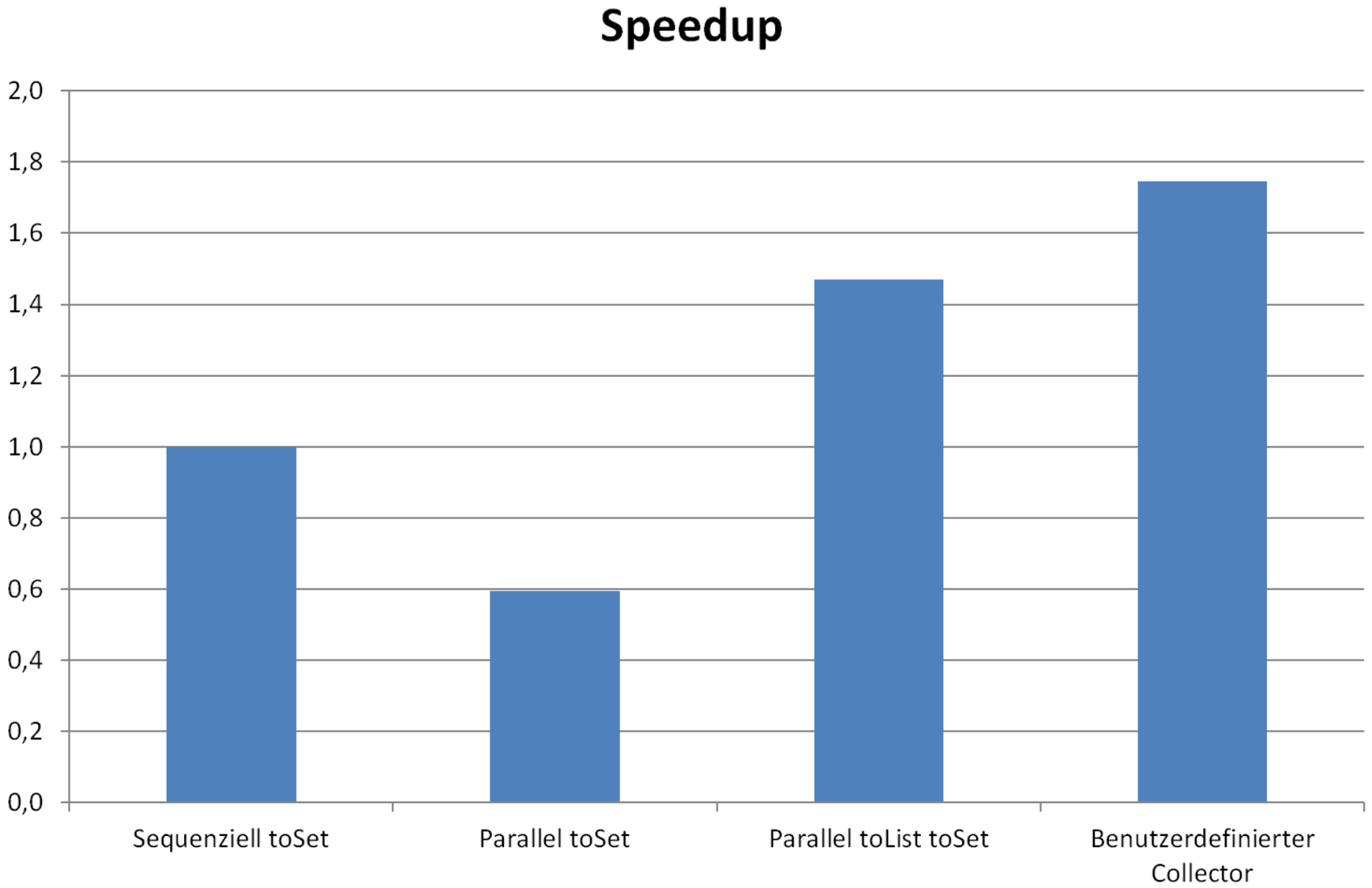
Die Ergebnislisten enthalten jeweils 10.000 Treffer (aus 30.000 Kunden). Als Referenzwert für den Speedup wurde das Sammeln mit `Collector.toList` auf einem sequentiellen Stream herangezogen.



Beispiel: „Ergebnistyp Set“

Soll die Verarbeitung eine `Set`-Datenstruktur liefern, erhält man in dem Beispiel mit einem benutzerdefinierten `Collector` das beste Ergebnis.

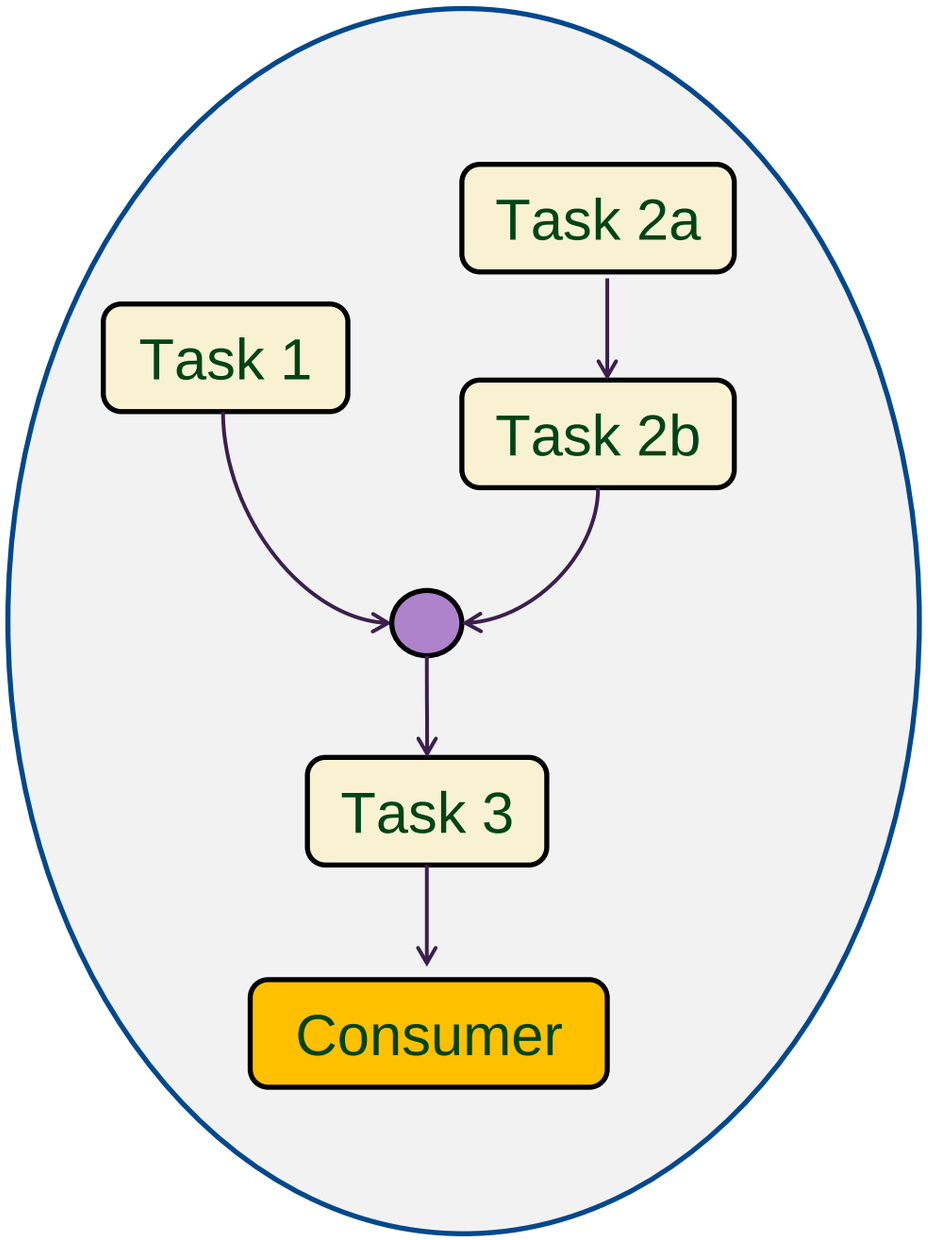
- Bem.: Das parallele Sammeln in eine `List` und dann umwandeln in eine `Set` ist hier schneller als direktes paralleles Sammeln in eine `Set`.



CompletableFuture

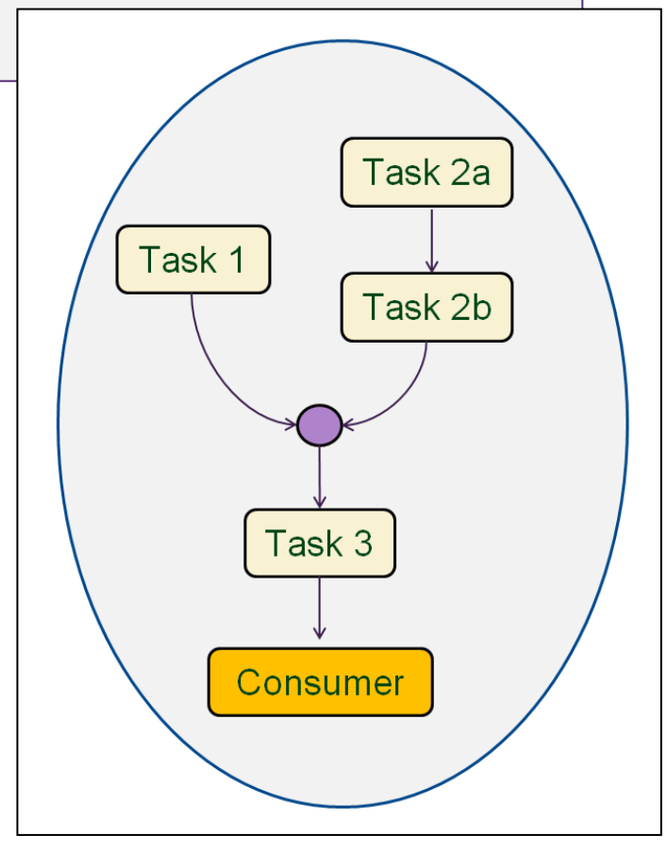
CompletableFuture–Klasse

- Realisierung von Task-Parallelität
- Definition von nebenläufigen Abläufen



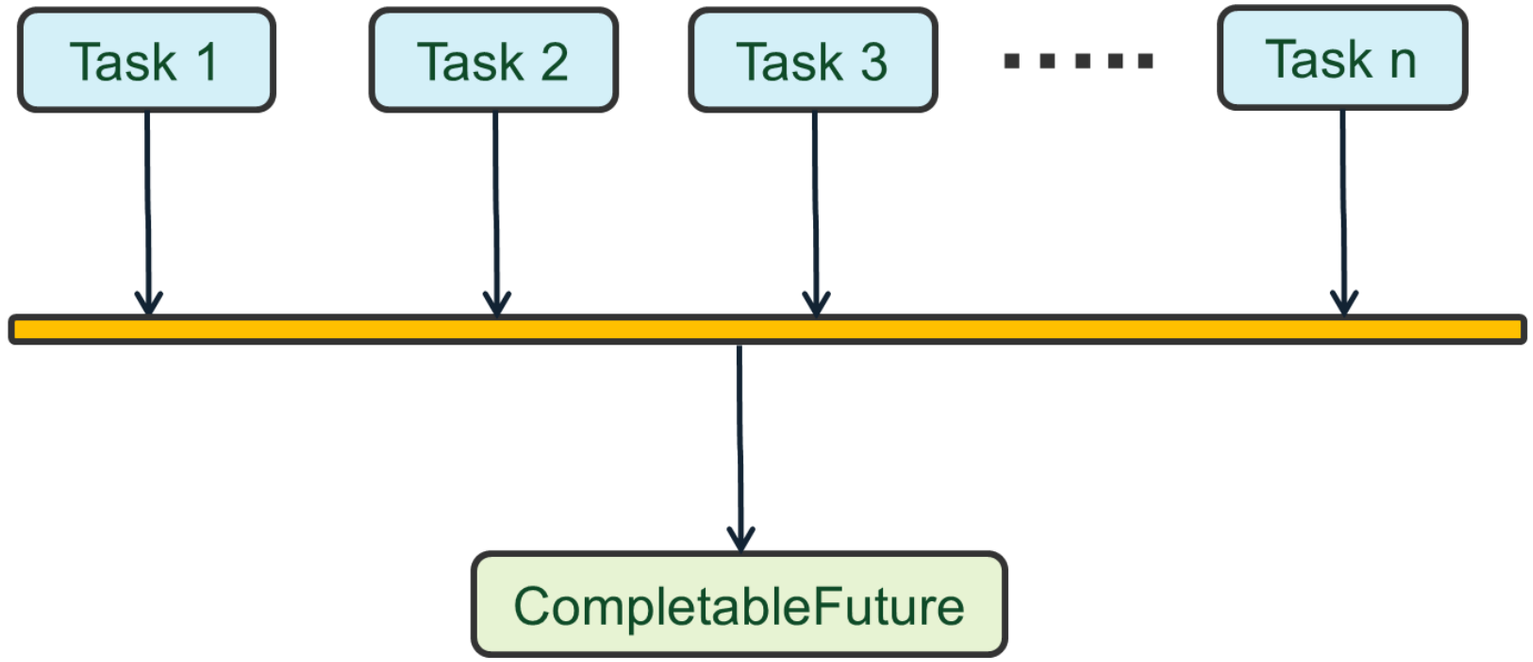
CompletableFuture

```
CompletableFuture<Integer> task1 =
          CompletableFuture.supplyAsync( () -> doTask1() );

CompletableFuture<Integer> task2a =
          CompletableFuture.supplyAsync( () -> doTask2a() );

task2a.thenApplyAsync( t -> doTask2b(t) )
      .thenCombineAsync( task1, (l,r) -> combine(l,r) )
      .thenApplyAsync( t -> doTask3(t) )
      .thenAcceptAsync( System.out::println );
```



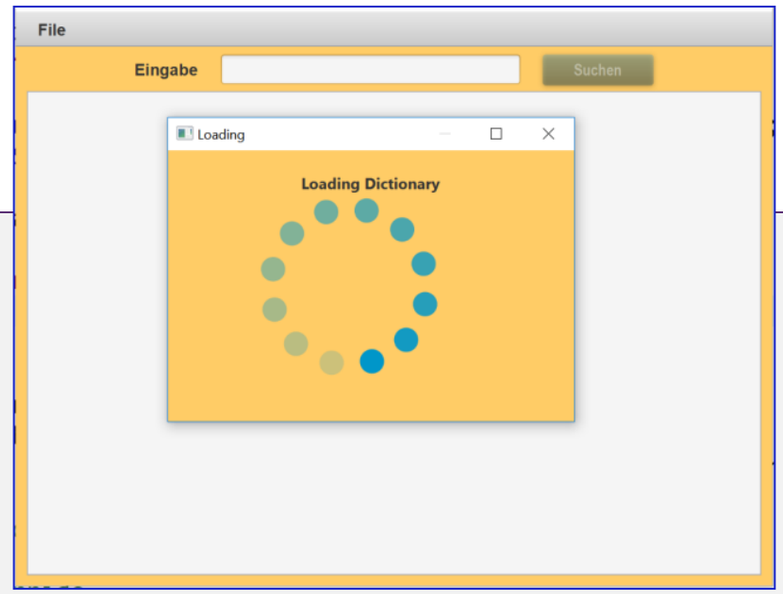
Barrieren

- Barrieren für **CompletableFutures**
  - **anyOf**
  - **allOf**



Beispiel: Asynchrones Laden von Daten

JavaFX-Anwendung




```
Task<Void> task = new Task<Void>()
{
  @Override
  protected Void call() throws Exception
  {
    // Langlaufende Aktion
  }
};
```

Task läuft **nicht** im GUI-Thread

Task läuft im GUI-Thread

```
CompletableFuture.runAsync( this::showProgress, Platform::runLater )
        .thenRunAsync( task )
        .thenRunAsync( this::closeProgress, Platform::runLater );
```

Bemerkung

Asynchrone Ausführung „anhängender“ Tasks

```
Supplier<Integer> supplier = () -> ...;
Consumer<Integer> consumer = (i) -> ...;

// Variante 1
CompletableFuture.supplyAsync( supplier )
                 .thenAccept( consumer );


// Variante 2
CompletableFuture.supplyAsync( supplier )
                 .thenAcceptAsync( consumer );
```

Wird **nicht** unbedingt asynchron ausgeführt

Wird **immer** asynchron ausgeführt

Flow-API: Reactive Streams

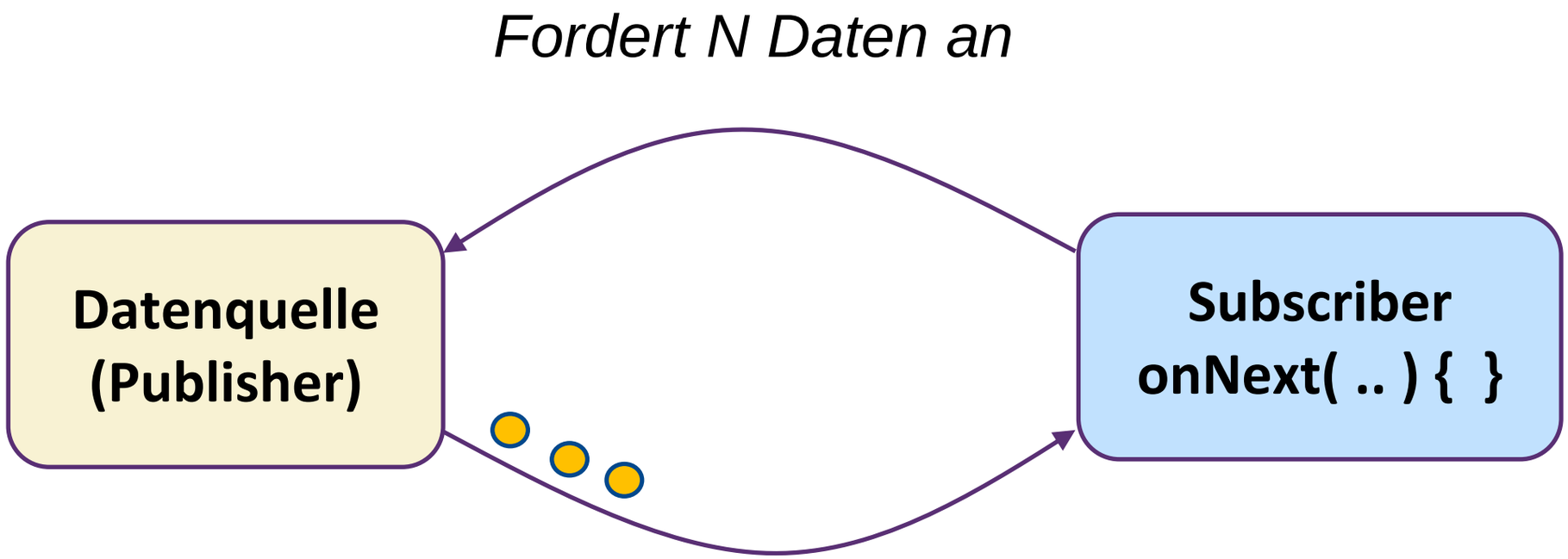
- Java 8-Streams unterstützen „pull“-Operationen
  - Explizite (interne) Dateniteration (*Kontrollflussorientiert*)

- *„Reactive Streams“* unterstützen „push“-Operationen
  - Entsprechen „aktiven“ Datenquellen (*Datenflussorientiert*)
    - Synchrone und asynchrone Datenauslieferung möglich
    - Java 9 bietet eine Andockstelle über Flow-Interfaces

- Anwendungsfälle
  - Verarbeitung von Echtzeitdatenquellen (Sensoren, etc.)
  - Monitor- und Analysewerkzeuge
  - Übertragung großer Datenmengen
  - …

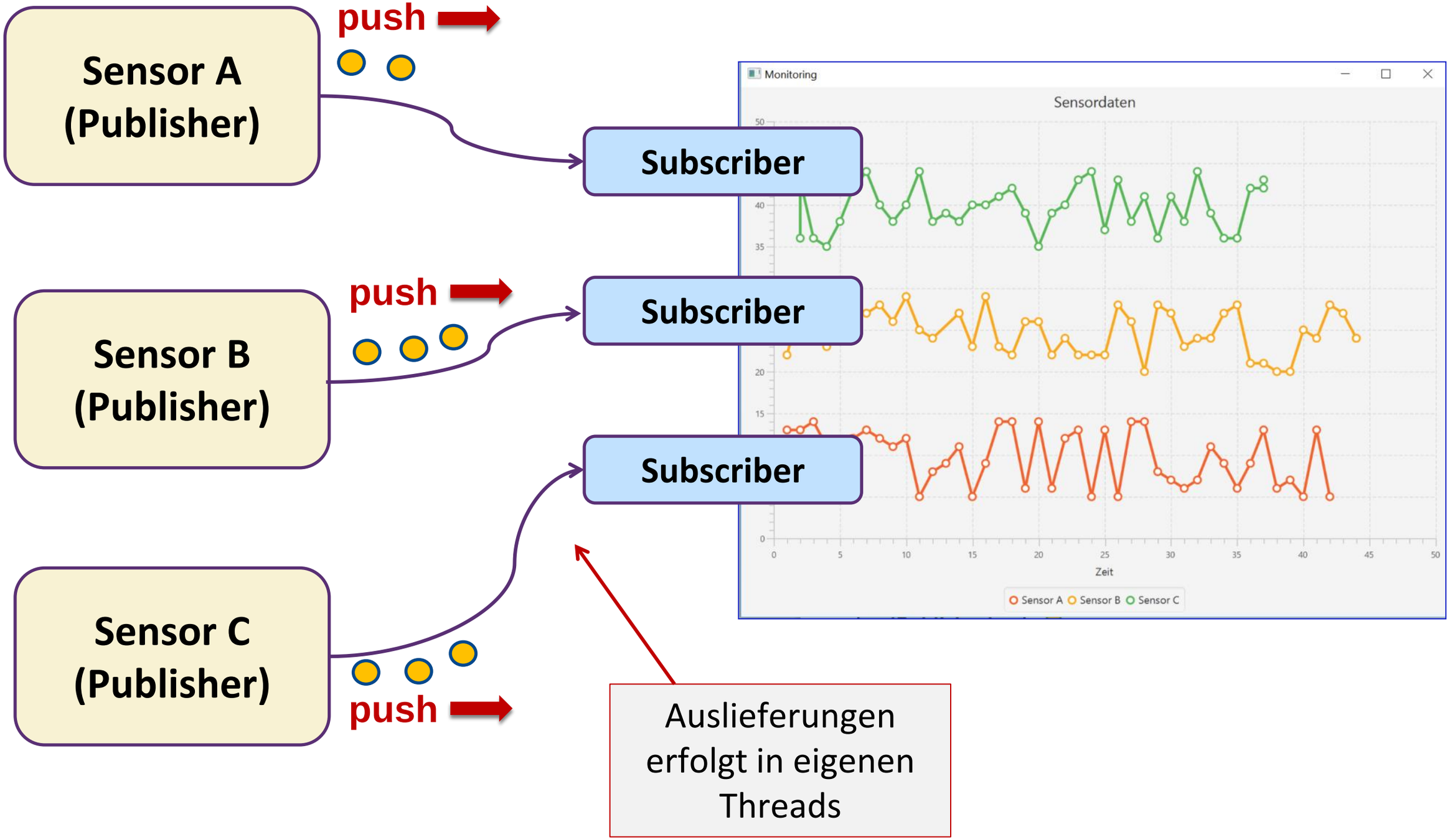
Reactive Streams

Erweitertes Observer-Pattern



Datenauslieferung an onNext() wird z.B. asynchron ausgeführt

Beispiel: Sensor-Monitoring



Beispiel: „Subscriber“

```
private static class SensorSubscriber implements Subscriber<Integer>
{
  // ...
  private SensorSubscriber( ... ) { ... }

  @Override
  public void onSubscribe(Subscription subscription)
  {
    this.subscription = subscription;
    this.subscription.request(50);
  }

  @Override
  public void onNext(Integer item)
  {
    ...
    Platform.runLater(() -> {
      // Aktualisiere Chart
    });
  }

  @Override public void onError(Throwable throwable) { ... }
  @Override public void onComplete() { ... }
}
```

Take Home Message

Java bietet heute

- **Concurrency-Tools**
  - Locks, Synchronisationsbarrieren, Threadpools und Futures, …
- **Threadsicherheit**
  - Atomic-Variablen, „Concurrent“-Datenstrukturen
- **Parallelisierungsframeworks**
  - ForkJoin
  - Parallel Streams
  - CompletableFuture-Klasse

Bau von Bibliotheken und Frameworks

Klassen für Multithread-Anwendungen

Beschleunigung von Anwendungen

Literatur

Web-Quellen:

[a] Doug Lea: *When to use parallel streams*
http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html

[b] Jörg Hettel und Manh Tien Tran: *Java goes Parallel*
https://www.informatik-aktuell.de/entwicklung/programmiersprachen/java-goes-parallel.html

Bücher:

[1] Brain Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes und Doug Lea: *Java Concurrency in Practice*.
Addison-Wesley, 2006.

[2] Jörg Hettel und Manh Tien Tran:
Nebenläufige Programmierung mit Java.
dpunkt.verlag, 2016.