Best Practices für moderne Multicore-Programmierung

Jörg Hettel

Hochschule
Kaiserslautern
University of
Applied Sciences

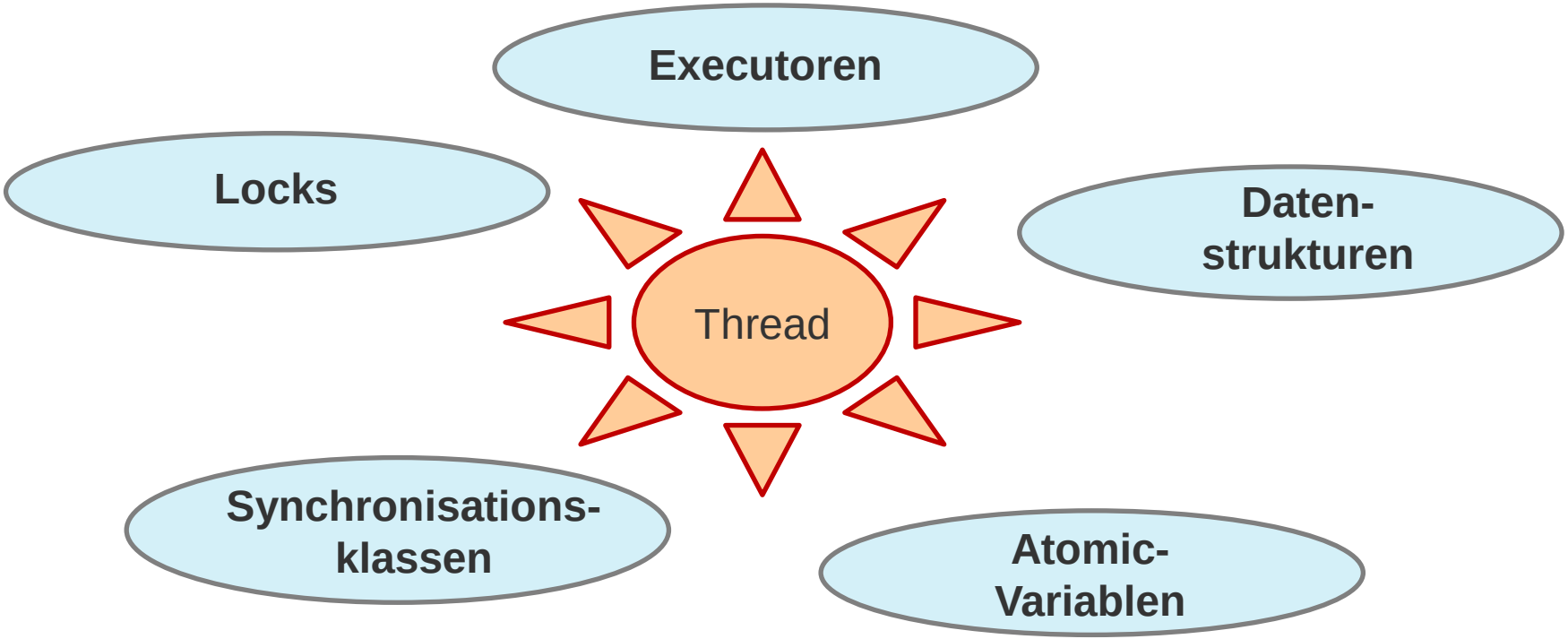
Agenda

- **Allgemeine Parallelisierungsstrategien und -Tools**
- **Best Practices**
  - Engpass Anwendungsstart
  - Engpass Vorschaubilderzeugung
  - Erhalt der Reaktivität der Anwendung
  - Engpass Algorithmus
- **Take Home Message**

Hochschule Kaiserslautern
University of Applied Sciences

Parallelisierungsstrategien und -Tools

Concurrency-Tools von Java



Concurrency-Tools von Java

Hochschule Kaiserslautern
University of Applied Sciences

Arbeiten mit rudimentären "Concurrency-Tools" ist mühsam und fehleranfällig

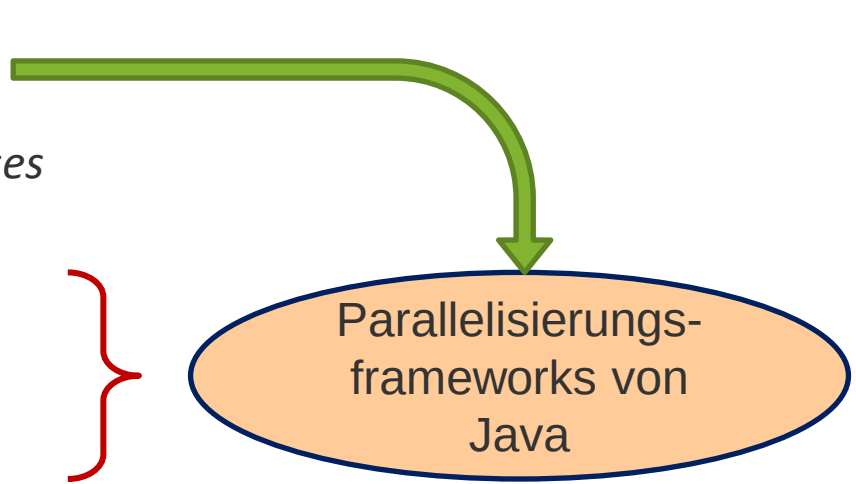
Problem: **Mutable Shared State**

- Falsche Synchronisation führt z.B. zu Deadlocks
- Nichtberücksichtigung von "Sichtbarkeitsregeln" führt zu inkonsistenten Daten
- Falsche Aufteilung der "Load" führt zu "Ressourcenverschwendung"
- Etc.

Blick über den Zaun

Concurrency-Modelle:

- *Thread-Programmierung*
- *Funktionale Programmierung*
- *Communicating Sequential Processes*
- *Aktoren*
- *Datenparallelität*
- *Event-Basierte Programmierung*
- *Reaktive Programmierung*
- *...*



Parallelisierungsframeworks

Hochschule Kaiserslautern
University of Applied Sciences

Einführung von Abstraktionen für die Multicore-Programmierung:

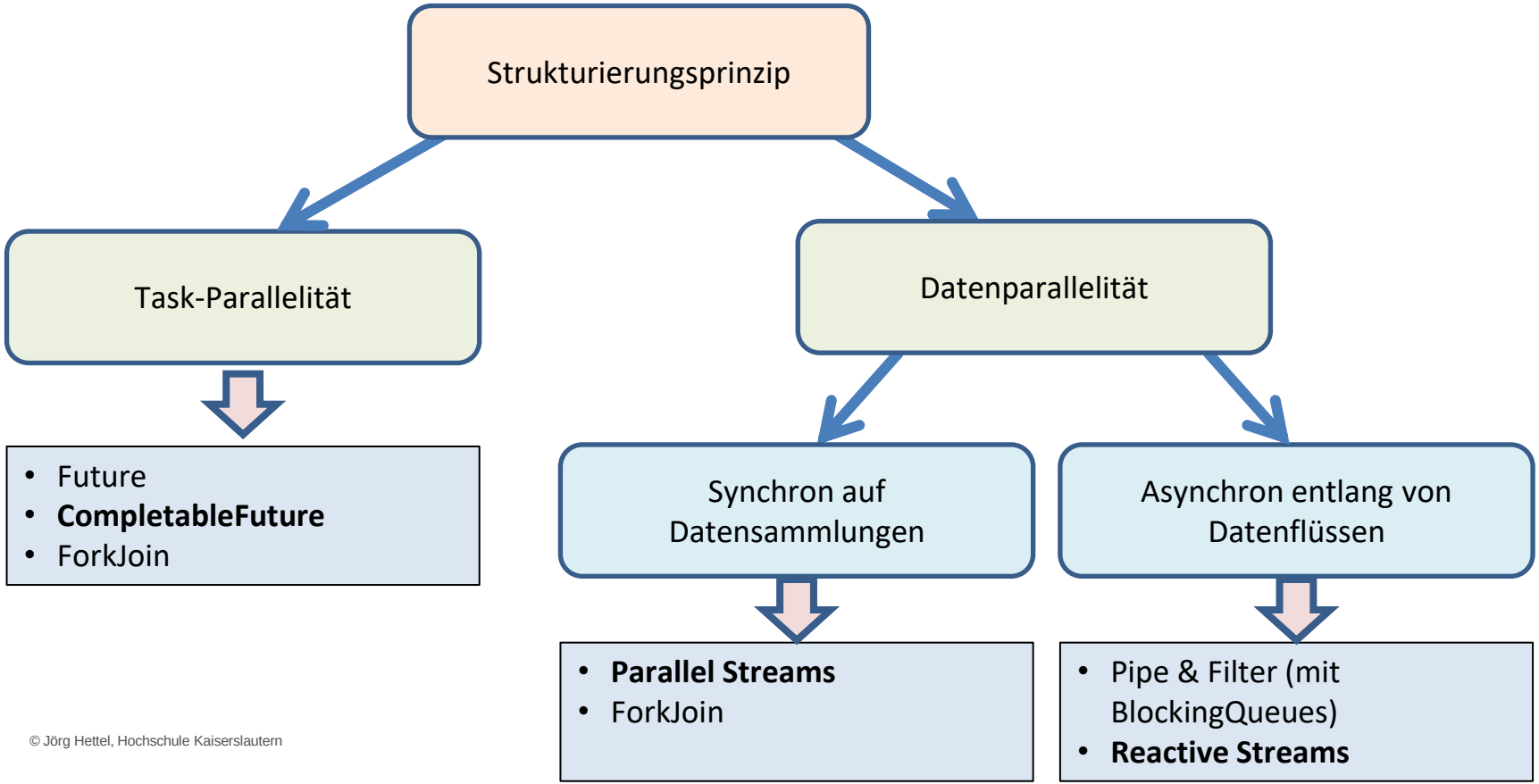
High-Level-Frameworks

- *Parallel Streams*
- *CompletableFuture*-Klasse
- *Flow-Klassen* (+ Frameworks)

Ziele:

- Einfache Implementierung einer Parallelverarbeitung zur Beschleunigung von Anwendungen
- Nutzung eines anderen Concurrency-Modells

Parallelisierungsstrategien (mit Java)



Parallelisierungsstrategien mit Java

Hochschule Kaiserslautern
University of Applied Sciences

Definition: **Task** als Abstraktion für eine Sammlung von zusammengehörigen (abhängigen) Anweisungen

Heuristik für die Parallelisierung

- **Task-Parallelität**

  Tasks, die keine Abhängigkeiten haben, können parallel ausgeführt werden

- **Datenparallelität**

  Ein Task bearbeitet eine Menge von Daten, wobei jedes Element auf die selbe Art und Weise verarbeitet wird

- **Reaktive Streams**

  Ausführung eines Tasks wird durch eine äußeres Signal "getriggert"

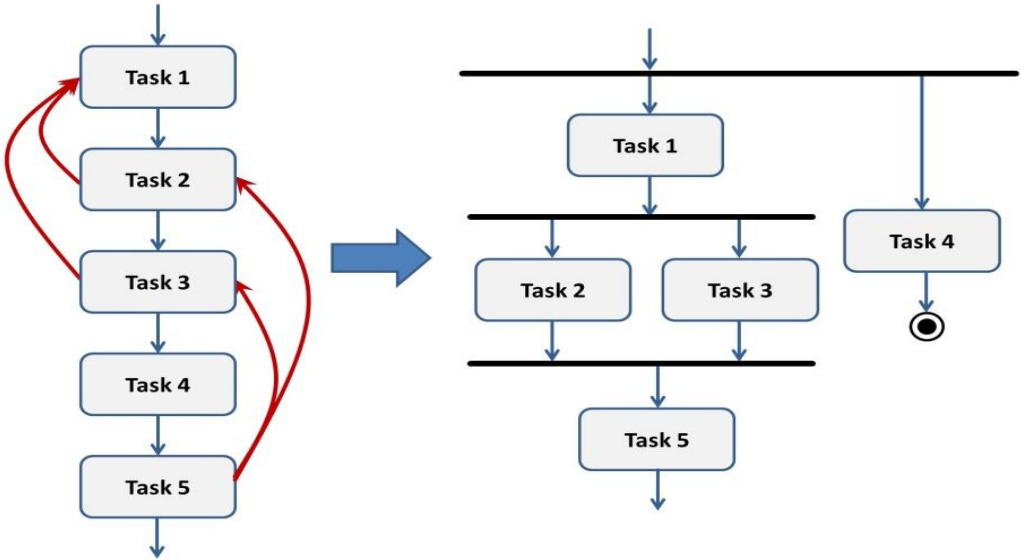
Hochschule Kaiserslautern
University of Applied Sciences

Best Practices

Task-Parallelität

Task-Parallelität

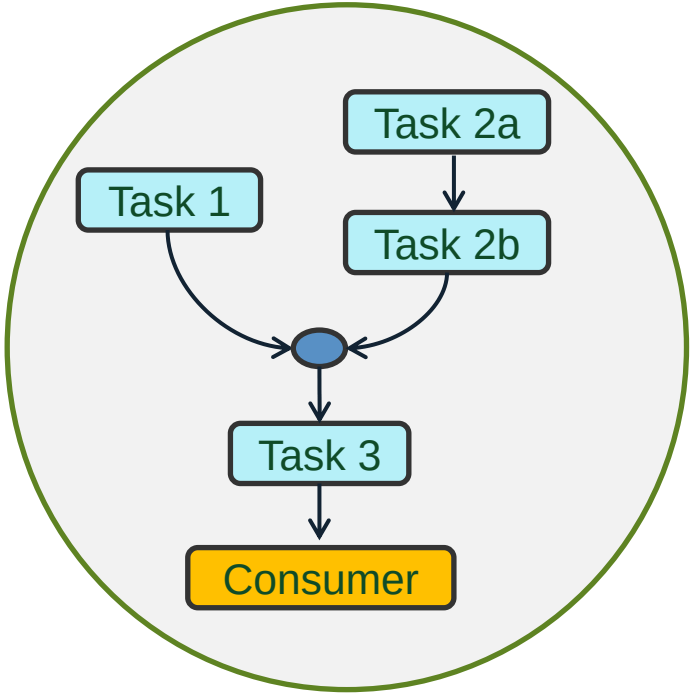
Abhängigkeiten von Tasks legen die Parallelisierungs-möglichkeiten fest



Java CompletableFuture

CompletableFuture–Klasse für Task-Parallelität

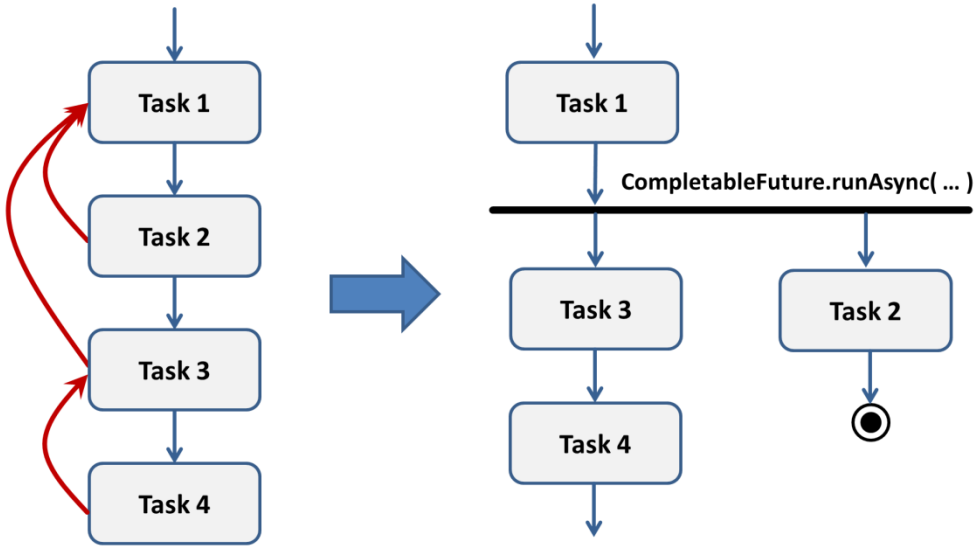
- Definition von asynchronen nebenläufigen Abläufen



Fire-and-Forget-Task

Ein Task 2 hat keine weiteren Abhängigkeiten mehr

- Kann asynchron ausgeführt werden!



CompletableFuture-Idiom

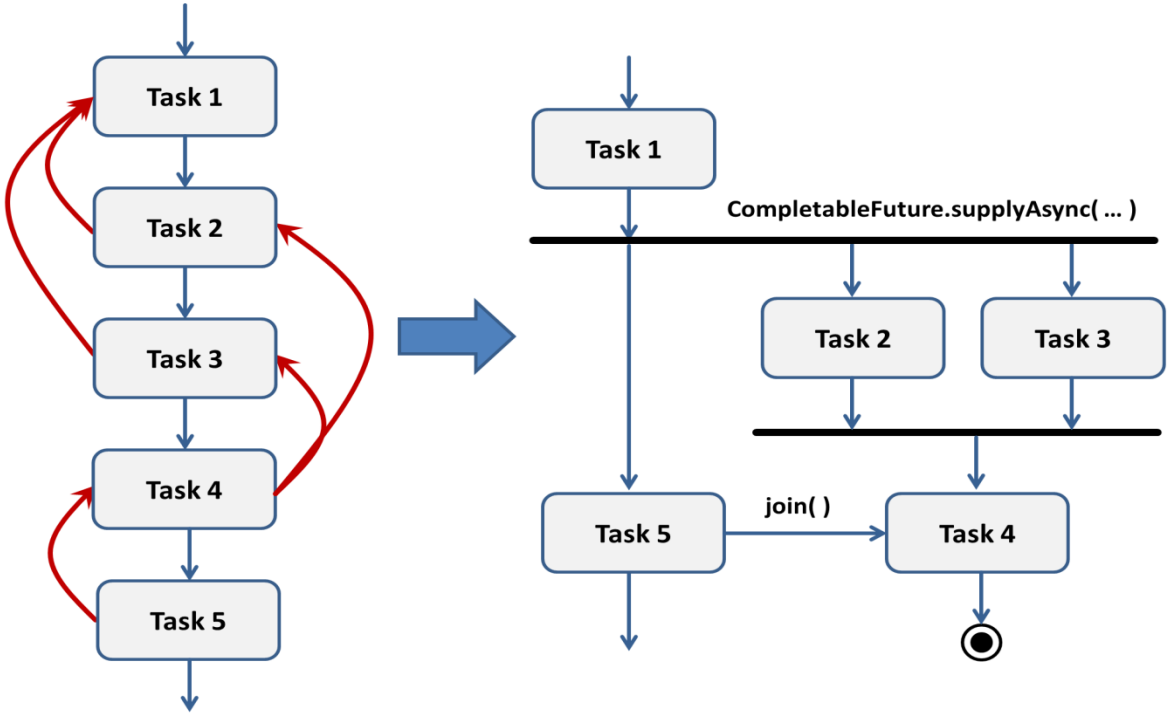
Umsetzung einer einfachen Task-Parallelität (*fire-and-forget*)

- **Wichtig:** Exceptions sollten immer explizit "behandelt" werden
- Oft ist es wichtig Zeitüberschreitungen zu melden
  - Bemerkung: Der eigentliche Task wird nicht abgebrochen!

```
CompletableFuture.runAsync( () ->
{
   // asynchrone Ausführung
   // ...
})
.orTimeout(2000, TimeUnit.MILLISECONDS)
.exceptionally( (ex) -> { ... } );
```

Split-and-Join-Tasks

Tasks besitzen keine lineare Abhängigkeiten



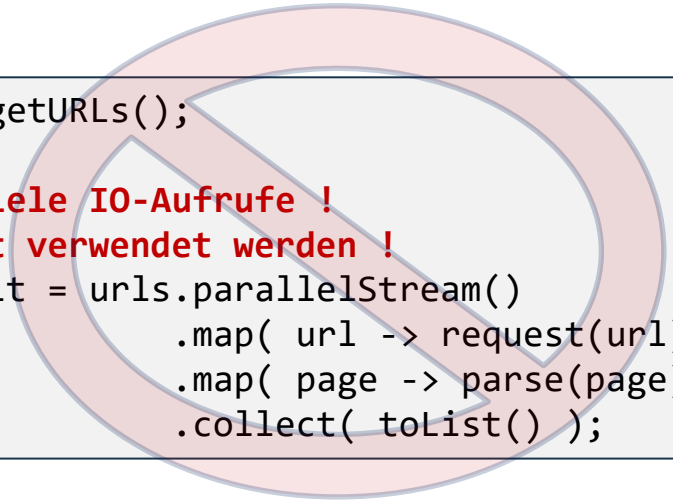
Einfaches CompletableFuture-Idiom

Einfaches Warten, bis asynchrone Ausführung beendet ist

```
...

CompletableFuture<Double> task2CF  =
                CompletableFuture.supplyAsync( () -> { } );

CompletableFuture<Double> task3CF =
                CompletableFuture.supplyAsync( () -> { } );


// UND-Verknüpfung
CompletableFuture<Double> task4CF =
     task2CF.thenCombine(task3CF, (resTask2, resTask3) -> { } );

// ...

R result = task4CF.join();
```

Paralleles IO (Variante 1)

- Don't do this!!
- Codesequenz entspricht einem blockierenden Aufruf
  - Es werden die Threads des Common-Pool benutzt

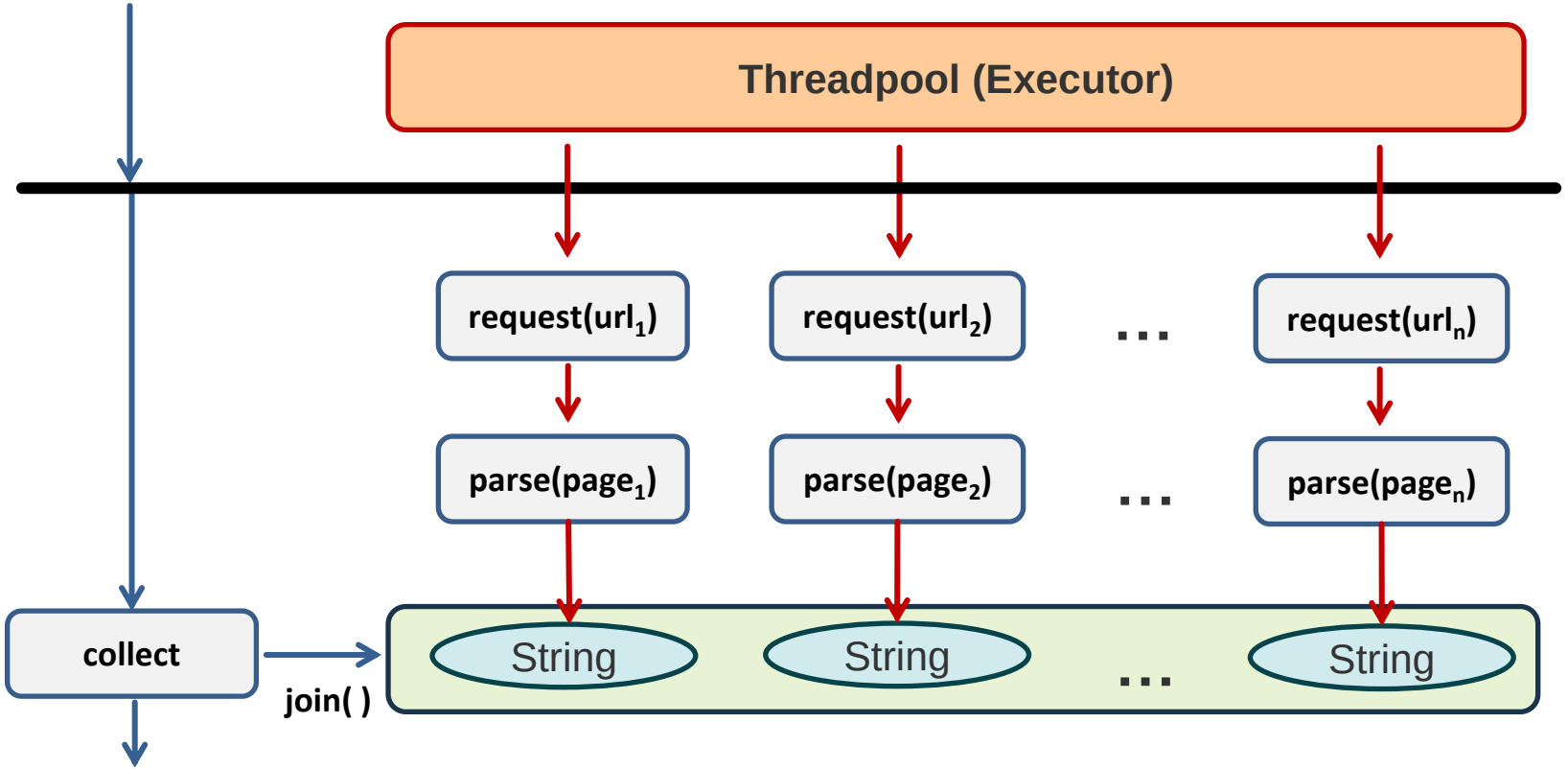


```
List<URL> urls = getURLs();

// VORSICHT parallele IO-Aufrufe !
// Sollte so nicht verwendet werden !
List<String> result = urls.parallelStream()
                          .map( url -> request(url) )
                          .map( page -> parse(page) )
                          .collect( toList() );
```

Paralleles IO (Variante 1)

```
List<URL> urls = getURLs();
int maxThreads = ...;
ExecutorService executor = Executors.newFixedThreadPool(
    Math.min( urls.size(), maxThreads),
    new ThreadFactory() { // Erzeuge Daemon-Thread } );

// Asynchrones IO
List<CompletableFuture<String>> futures =
     urls.stream()
         .map( url -> CompletableFuture.supplyAsync(
                              () -> request(url), executor ) )
         .map( future -> future.thenApplyAsync(
                              page -> parse(page), executor ) )
         .collect( toList() );

List<String> result = futures.stream()
                             .map( CompletableFuture::join )
                             .collect( toList() );
executor.shutdown();
```

Paralleles IO (Variante 1)



Hochschule Kaiserslautern
University of Applied Sciences

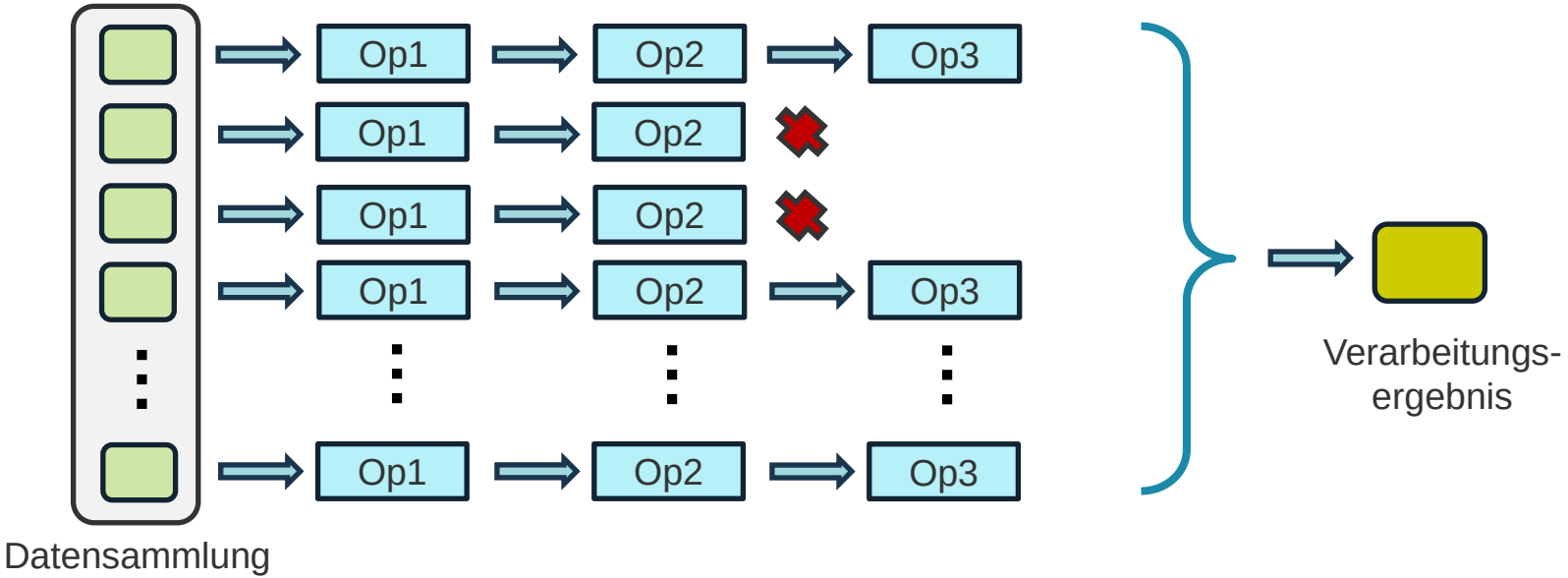
Best Practices

Datenparallelität

Datenparallelität

Gleichförmige Verarbeitung einer "Datensammlung"

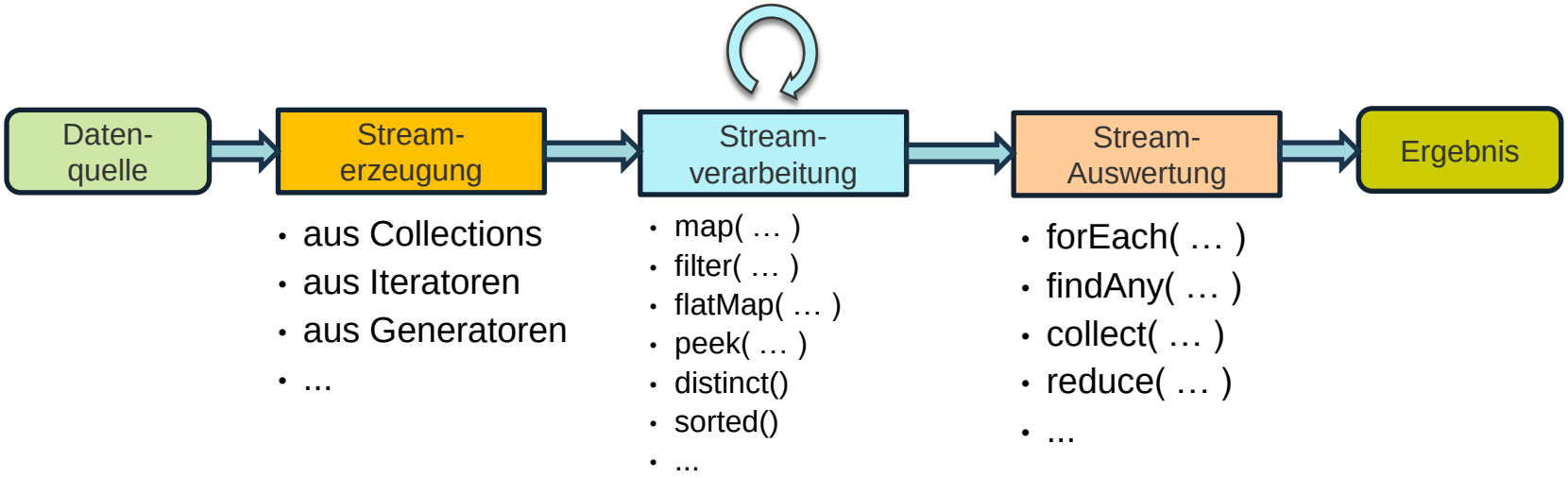
- "Map-Reduce"- bzw. "Map-Collect"-Prinzip



Java-Streams

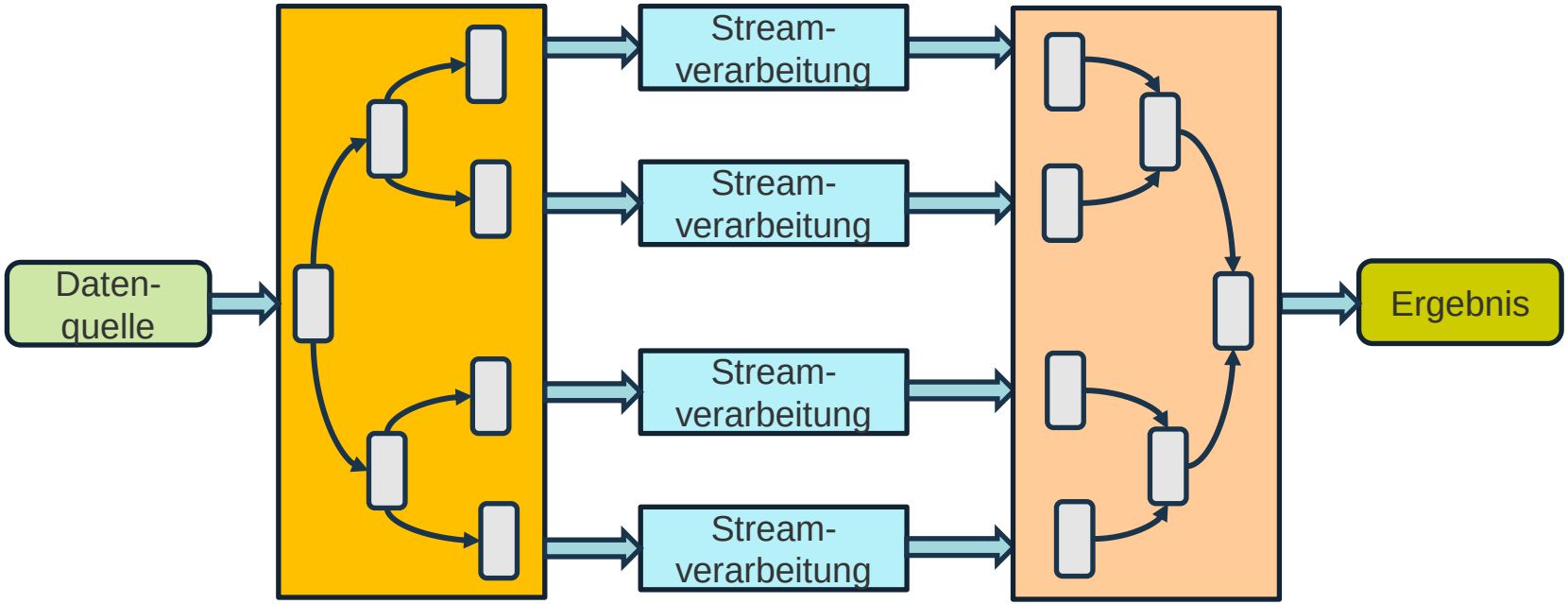
„Java 8“-Streams für Daten-Parallelität

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



Parallele Java-Streams

Parallelisierung durch Fork/Join-Mechanismus



Bemerkungen zu Parallel-Streams

Hochschule Kaiserslautern
University of Applied Sciences

- Stream-Verarbeitung muss „seiteneffektfrei“ (*parallel ready*) sein!
- 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

- Sowohl in den Split-Prozess als auch in den Reduce/Collect-Prozess kann „eingegriffen“ werden
  - Benutzerdefinierte Spliteratoren bzw. Collectoren

Paralleles IO (Variante 2)

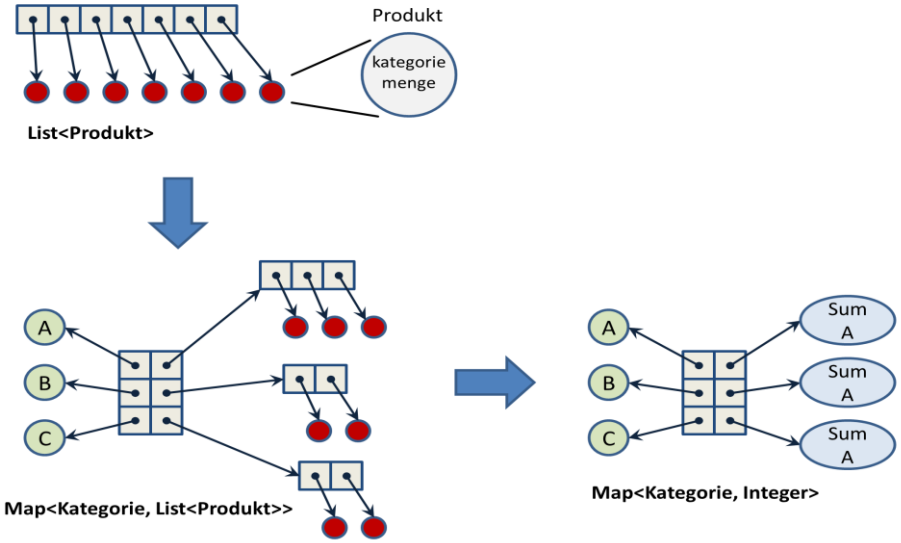
Hochschule Kaiserslautern
University of Applied Sciences

- Mit parallelen Streams, aber eigenem Threadpool (Executor)

```
List<URL> urls = getURLs();
int maxThreads = ...;
ForkJoinPool pool = new ForkJoinPool( Math.min( urls.size(), maxThreads );

// Asynchrones IO
ForkJoinTask<List<String>> task = pool.submit(
  () ->  urls.parallelStream()
             .map( url -> request(url) )
             .map( page -> parse(page) )
             .collect( toList() )
  );


List<String> result = task.join();

pool.shutdown();
```

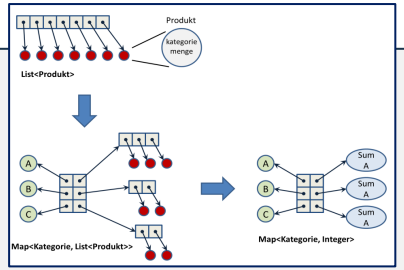
Benutzerdefinierte Collectoren

- **Beispiel:** Bestimme den Bestand der einzelnen Kategorien.
  - Es gibt verschiedene Produkte, die jeweils einer Kategorie zugeordnet sind



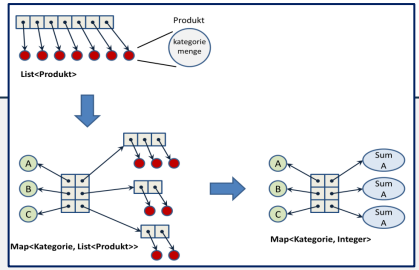
Formulierung mit Streams

- Naive (entspricht dem Bild)




```
List<Produkt> produkte = ...;

Map<Kategorie, List<Produkt>> kategorieMap =
     produkte.stream()
             .collect(
                 Collectors.groupingBy( Produkt::getKategorie) );

Map<Kategorie, Integer> kategorieSum =
     kategorieMap.entrySet()
                 .stream()
                 .collect( Collectors.toMap(
                   entry -> entry.getKey(),
                   entry -> entry.getValue()
                                 .stream()
                                 .mapToInt( Produkt::getMenge ).sum()));
```

Downstream Collection

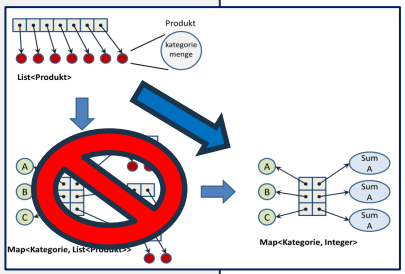



```
List<Produkt> produkte = ...;

Map<Kategorie, Integer>  kategorieMap =
       produkte.stream()
               .collect( Collectors.groupingBy(
                           Produkt::getKategorie,
                           Collectors.summingInt( Produkt::getMenge)) );
```

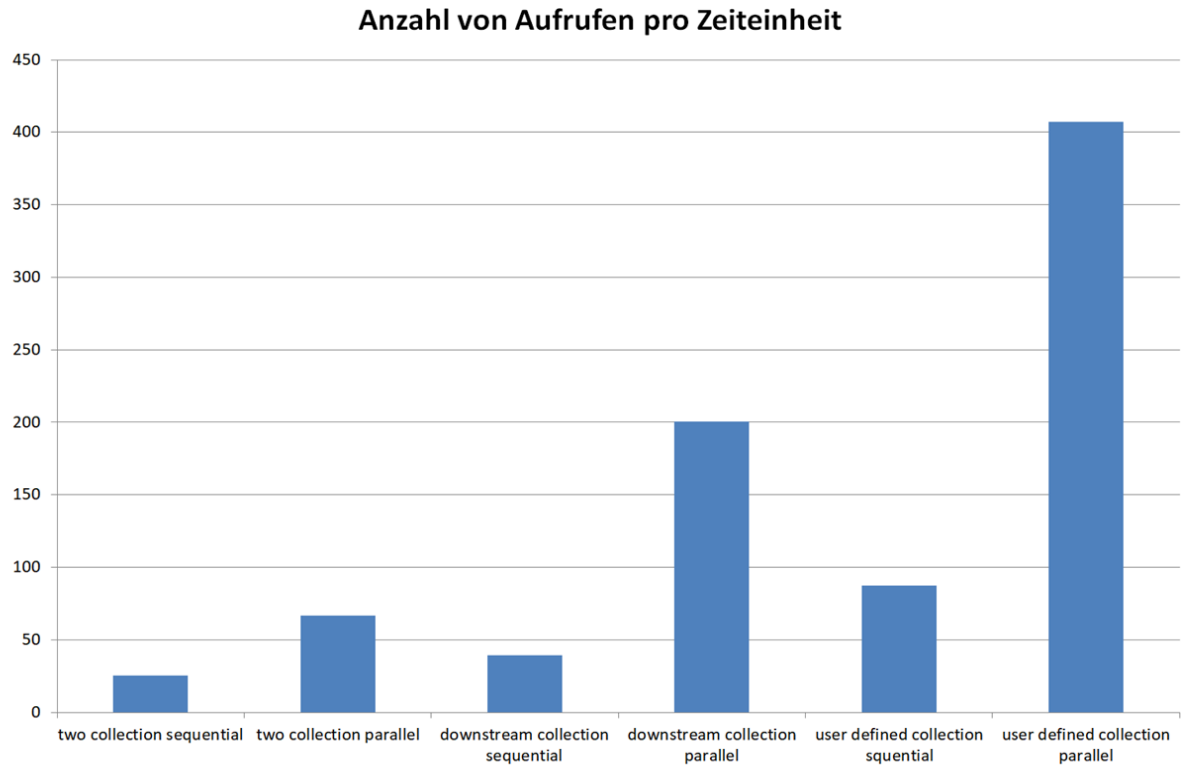
Benutzerdefinierter Collector

- Gruppierung und Summation können zusammenngefasst werden

```
class ProduktCollector implements  Collector<...>
{
  public Supplier<int[]> supplier() {
    return () -> new int[3];  // Drei Kategorien
  }

  public BiConsumer<int[], Produkt> accumulator() {
    return (array, produkt) -> {  // Zählt Kategorien };
  }

  public BinaryOperator<int[]> combiner() {
    return (left,right) -> { // Vereinigt Teilergebnisse  };
  }
  ...
}
```



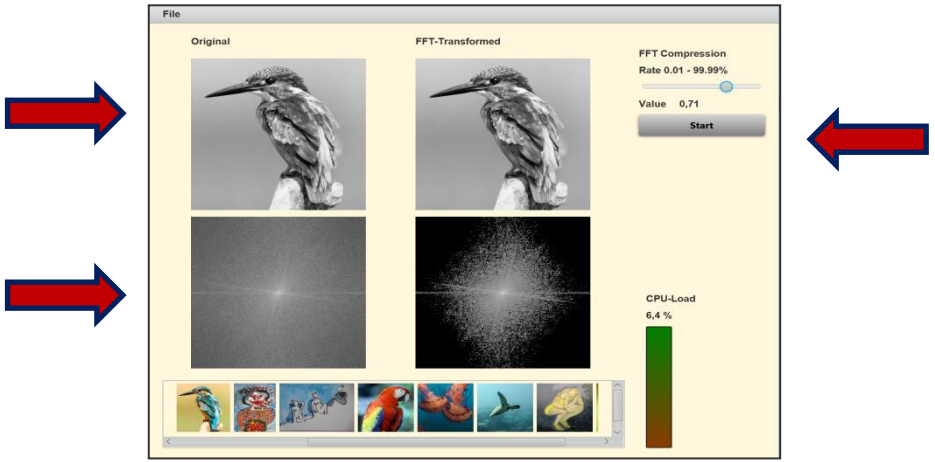
Benchmark



Beispiel

Verbesserungspotential

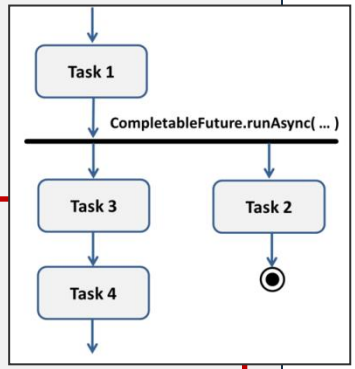
- Programmstart
- Anzeige der Vorschaubilder
- Berechnung der Bildkompression
- Reaktivität der Anwendung



Beispiel: https://github.com/hettel/ImageCompression-CaseStudy

Asynchrone Initialisierung der Hardware

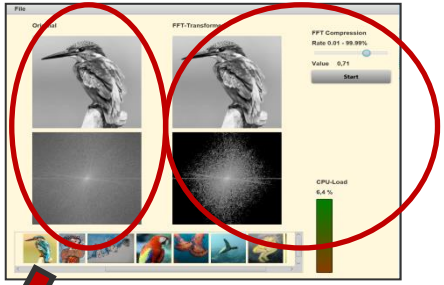
```
private CpuInfoPublisher publisher;

@Override
public void initialize(URL location, ResourceBundle resources)
{
  clusterCount.addAll(1, 2, 3, 4, 5, 6, 7, 8, 9);
  ...
  cpuLoadBar.setFill(Color.GREEN);

  CompletableFuture.runAsync( () ->
  {
    publisher = CpuInfoPublisher.getInstance();
    publisher.subscribe(value -> Platform.runLater(() -> {
      repaintGradient(value);
      cpuLabel.setText(df.format(100 * value) + " %");
    }));
  }).exceptionally(
        ex -> { ex.printStackTrace(); return (Void) null; });
}
```

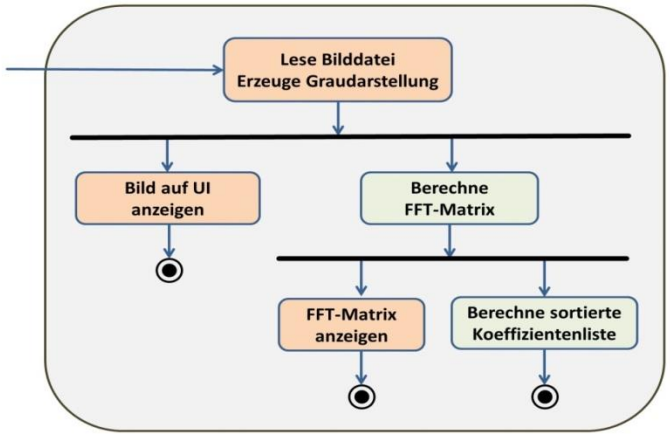


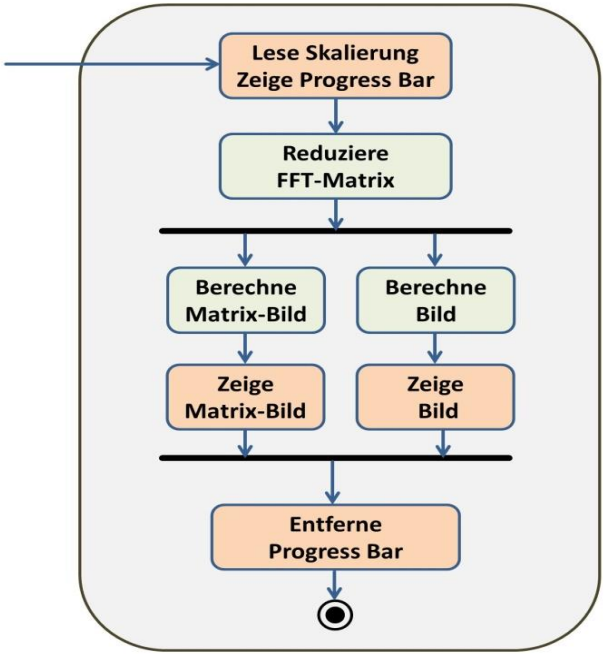
Erzeugung der Vorschaubilder

```
File[] listOfFiles = imageFolder.listFiles( ... );

int numOfProcessors = Runtime.getRuntime().availableProcessors();
int numOfPoolThreads = Math.min( listOfFiles.length , numOfProc*16);

ForkJoinPool pool = new ForkJoinPool(numOfPoolThreads);
ForkJoinTask<List<ImageView>> task = pool.submit(
        () ->
            Arrays.stream(listOfFiles)
                  .parallel()
                  .map( FileIOHelper::createPreviewImage )
                  .map( FileIOHelper::createImageView )
                  .collect( toList() )

        );

List<ImageView> resultList = task.join();
pool.shutdown();
```

Berechnung der FFTs







Take Home Message (1)

Hochschule Kaiserslautern
University of Applied Sciences

- Es gibt viele potentielle Stellen für den Einsatz von Parallelisierung
  - Oft mehr als man auf den ersten Blick vermutet!
  - Immer auch den Overhead beachten
  - Frameworkeinsatz birgt auch Fehlerpotential
    - Erschwert das Debugging und das "Code-Verstehen"

Drei Parallelisierungsstrategien bei Java:

- **Daten-Dekomposition**
  - Parallel Stream oder Fork/Join
- **Task-Dekomposition**
  - CompletableFuture oder Future
- **Asynchrone Daten-Auslieferungen**
  - Publish/Subscribe

Take Home Message (2)

Hochschule Kaiserslautern
University of Applied Sciences

- **Task-Parallelität** (*CompletableFuture*)
  - Besitzt großes Anwendungspotential
    - "Seiteneffekte" und "Regeln" beachten
    - Einsatz dokumentieren und nicht "übertreiben"
  - Augenmerk auf Fehlersituationen legen
    - Ggf. mit Timeouts arbeiten

- **Daten-Parallelität** (*parallel Streams*)
  - Parallele Streams lohnen sich nicht immer
    - "Seiteneffekte" und "Regeln" beachten

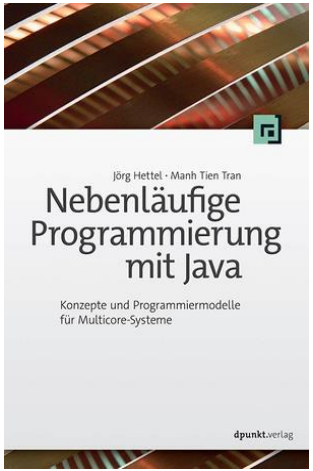
Hochschule Kaiserslautern
University of Applied Sciences

Gibt es Fragen?

Jörg Hettel

Hochschule Kaiserslautern
Campus Zweibrücken
Fachbereich Informatik

eMail: joerg.hettel@hs-kl.de



andrena
OBJECTS

Bitte geben Sie uns jetzt Ihr Feedback!

Best Practices für moderne Multicore-Programmierung

Prof. Dr. Jörg Hettel

Nächste Vorträge in diesem Raum

13:30 GraphQL als Schnittstelle zum Backend, *Christian Kumpe, Thorben Hischke*

14:30 Integration ist schwer, Integrationstests umso mehr, *Benjamin Muskalla*

15:45 Java 9 ist tot, lang lebe Java 11, *Falk Sippach, Steffen Schäfer*