Karlsruher Institut für Technologie

KASTEL / IPD
Prof. Dr. Sebastian Erdweg erdweg@kit.edu
Prof. Dr. Ralf H. Reussner reussner@kit.edu

Programmierparadigmen – WS 2025/26

https://dsis.kastel.kit.edu/1181_1197.php

Blatt 2: Java Parallelprogrammierung Besprechung: 09.01.2026

1 Amdahlsches Gesetz

Ein Programm mit mehreren Threads arbeitet auf einem Puffer. Ein Teil der Threads liest Daten aus dem Puffer (Leser), der andere Teil schreibt Daten in den Puffer hinein (Schreiber). Eine beliebige Anzahl lesender Threads kann gleichzeitig auf dem Puffer arbeiten. Wenn ein Schreiber auf den Puffer zugreift, kann kein anderer Thread (Leser oder Schreiber) auf dem Puffer aktiv sein.

Gegeben ist ein Thread Pool, wobei jeder Leser und Schreiber durch einen Thread repräsentiert wird. Dabei sind 90% der Threads Leser und 10% Schreiber. Jeder Leser benötigt 2 Sekunden und jeder Schreiber 3 Sekunden zur Ausführung. Das Programm ist beendet, wenn alle Threads im Pool beendet sind.

Aufgabe: Unter Berücksichtigung des Amdahlschen Gesetzes, wo liegt die obere Grenze der Beschleunigung (Speedup) der oben beschriebenen Implementierung auf einem 4-Kern-Prozessor?

Beispiellösung: Amdahls Gesetz berechnet die maximale Beschleunigung, die durch Parallelverarbeitung erreicht werden kann.

P: Anteil eines Programms, der parallelisiert werden kann
N: Anzahl der Prozessoren

$S(P) = \frac{1}{(1-P)+\frac{P}{N}}$

$P = \frac{(2*0.9)}{2*0.9+3*0.1} \approx 0.86$

Mit $N = 4$ resultiert dies in: $S(P) = \frac{1}{(1-0.86)+\frac{0.86}{4}} \approx 2.82$

2 Threading-Grundlagen: Producer-Consumer

Ein häufig verwendetes Muster in der Parallelverarbeitung zur Verteilung von Arbeit auf ein oder mehrere Arbeiter-Threads ist das Producer-Consumer-Muster. Hierbei werden in eine Datenstruktur Elemente von Erzeuger-Threads (Producer) eingefügt und von Verarbeitungs-Threads (Consumer) entfernt und verwendet.

Aufgabe: In dieser Aufgabe sollen Sie für ein Producer-Consumer-Szenario mit Integern die Basis-Funktionalität und das Threading implementieren. Die benötigten Source-Code-Dateien, in denen Sie arbeiten werden, finden Sie im ILIAS-Kurs.

1. Ergänzen Sie die Methoden `produce()` und `int consume()` in der bereitgestellten `Buffer`-Klasse in der `Buffer.java`-Quelldatei. In `produce()` soll der erzeugte Integer (`element`) der Warteschlange `queue` thread-sicher hinzugefügt werden. Falls dieser Vorgang die gegebene Maximal-Kapazität (`capacity`) von `queue` überschreiten würde, soll der Thread warten bis wieder Kapazitäten frei sind. In der Methode `int consume()` soll ein Element aus der Queue `queue` Thread-sicher entfernt werden, wenn mindestens ein Element vorhanden ist. Ansonsten soll der Thread warten bis Elemente hinzugefügt wurden.
   Sie können für das Warten auf Kapazitäten oder auf Elemente die Methoden `wait()` und `notifyAll()` verwenden.
   Tipp: Falls Sie die Ergebnisse auf der Konsole einfacher betrachten möchten, können Sie in die `produce` und `consume` Methoden Wartezeiten einfügen, damit die Ausgaben seltener erfolgen (z.B. durch Aufrufe von `Thread.sleep(int milliseconds)`).

   Beispiellösung: Der Quellcode der Lösung wird im ILIAS-Kurs zur Verfügung gestellt.
   In `produce()` und `consume()` ist jeweils eine Synchronisation auf `this` erforderlich, um eine Ausführung zu garantieren, die Thread-sicher ist. Solange die Warteschlange in der `consume()`-Methode leer ist (`queue.size() == 0`) bzw. ihre Kapazität in der `produce()`-Methode erreicht hat (`queue.size() > capacity`), wird `wait()` aufgerufen. Wenn die jeweilige Operation durchgeführt wurde, wird `notifyAll()` aufgerufen, da sich der Zustand der Warteschlange `queue` geändert hat und andere Threads somit potentiell weiterarbeiten können. Hierbei ist es wichtig, bei der `wait()`-Methode die `InterruptedException` zu fangen und zu behandeln, um das Beenden aller Threads korrekt zu ermöglichen. Außerdem sollte das `interrupt`-Flag durch einen Aufruf von `Thread.currentThread().interrupt()` erneut gesetzt werden, damit der Aufrufer zwischen einer erfolgreichen und einer unterbrochenen Ausführung der Methode unterscheiden kann.

2. Als nächstes sollen Sie zwei Producer-Threads und einen Consumer-Thread erstellen und starten. Modifizieren Sie hierfür die unten angegebene `main`-Methode der `ProducerConsumer.java`-Quelldatei. Alle Threads sollen die gleiche Instanz der `Buffer`-Klasse verwenden. Jeder Producer- bzw. Consumer-Thread soll die Methode `produce()` bzw. `consume()` der `Buffer`-Klasse solange aufrufen, bis der Wert der Kontroll-Variable `running` der `Buffer`-Klasse auf `false` gesetzt wird. Sie können den Wert durch die getter-Methode `isRunning()` abrufen. Bei der Implementierung des Threadings dürfen Sie alle in der Vorlesung vorgestellten Ansätze verwenden. Achten Sie darauf, eventuell wartende Threads zu beenden. Ergänzen Sie die folgende `main`-Methode und erstellen Sie bei Bedarf bis zu zwei weiteren Klassen:

```
public static void main(String[] args){
    Buffer buffer = new Buffer();

    //Erstellen und starten Sie die Threads hier.
```

```
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e1) {
            buffer.stopRunning();
        } finally {
            buffer.stopRunning();
        }

        // Beenden Sie ihre Threads hier und warten Sie auf Vollendung

    }
```

Beispiellösung: Der Quellcode der Lösung wird im ILIAS-Kurs zur Verfügung gestellt.
Eine mögliche Lösung das Threading zu realisieren ist es zwei neue Klassen `Producer` und `Consumer` zu erstelle,n in deren Konstrukteren die `Buffer`-Instanz übergeben wird. Diese Klassen implementieren entweder das Interface `Runnable` oder erben von `Thread` und implementieren die `run()`-Methode. In der `run()`-Methode wird in einer while-Schleife, die solange läuft wie `buffer.isRunning()` `true` zurückliefert, `buffer.produce()` bzw. `buffer.consume()` aufgerufen. Eine weitere Möglichkeit bei dem das Erstellen von Klassen umgangen wird, ist es in der `main`-Methode Threads mit anonymen Methoden auszuführen. Hierbei werden die Implementierungen der vorher erwähnten `run()`-Methoden der `Producer`- und `Consumer`-Klassen in die anonyme Methode verlagert.

3. Warum ist es in diesem Producer-Consumer Schema besser `notifyAll()` und nicht `notify()` zu verwenden?

   Beispiellösung: Ein Aufruf von `notify()` weckt nach einem nicht genauer spezifizierten Verfahren einen Thread auf. Ein Problem kann hier auftreten, wenn nur noch ein `Producer` oder `Consumer` arbeitet und alle anderen Threads warten. Ist danach der `Buffer` voll und wird ein `Producer` durch das `notify()` aufgeweckt, oder ist er leer und wird ein `Consumer` durch `notify()` aufgeweckt, so ruft dieser sofort wieder `wait()` auf und das Programm endet in einem Deadlock. Ein möglicher Ablauf mit zwei Producern `P1` und `P2`, sowie zwei Consumern `C1` und `C2` auf einem `Buffer` der Größe 1 ist wie folgt:

   (a) `P1` ruft `produce()` auf, ist erfolgreich, `Buffer` ist danach voll, ruft `notify()` auf (verpufft, da kein Thread wartet)

   (b) `P2` ruft `produce()` auf, `Buffer` ist bereits voll, ruft `wait()` auf

   (c) `P1` ruft `produce()` auf, `Buffer` ist bereits voll, ruft `wait()` auf

   (d) `C1` ruft `consume()` auf, ist erfolgreich, `Buffer` ist danach leer, ruft `notify()` auf, welches `P1` aufweckt

   (e) `C2` ruft `consume()` auf, `Buffer` ist bereits leer, ruft `wait()` auf

   (f) `C1` ruft `consume()` auf, `Buffer` ist bereits leer, ruft `wait()` auf

   (g) `P1` ruft `produce()` auf, ist erfolgreich, `Buffer` ist danach voll, ruft `notify()` auf, welches `P2` aufweckt

   (h) `P2` ruft `produce()` auf, `Buffer` ist bereits voll, ruft `wait()` auf

   (i) `P1` ruft `produce()` auf, `Buffer` ist bereits voll, ruft `wait()` auf

   (j) Alle Thread warten → Deadlock

   Da immer die gleichen Threads (z.b. zwei `Producer`-Threads) aufgeweckt werden können, würde das Programm selbst mit einem `notify()` vor einem `wait()` möglicherweise nicht

fortschreiten. Das Programm befände sich in einem Livelock. Bei `notifyAll()` werden alle Threads aufgeweckt und somit ist diesbezüglich kein Deadlock möglich.

3 Synchronisation: Barrieren

Implementieren Sie eine einfache zyklische Barriere. Die Barriere bekommt als einzigen Konstruktorparameter die Anzahl an Threads, welche `await` aufrufen müssen, damit die Barriere alle Threads freigibt. Nach dem Freigeben soll die Barriere erneut benutzbar sein. Die Methode `freeAll` löst eine manuelle Freigabe aus. Alle Threads, welche vor dem Zeitpunkt des `freeAll`-Aufrufs `await` aufgerufen haben, werden freigegeben. Weitere Threads, welche nach dem `freeAll`-Aufruf kommen, müssen wieder warten. Implementieren Sie folgendes Interface:

```
public interface IBarrier {
  void await() throws InterruptedException;
  void freeAll();
}
```

Testen Sie Ihre Barriere.

Beispiellösung:

```
public class Barrier implements IBarrier {
  private final int maxThreads;
  private int threadsInCurrentWaitingPhase = 0;
  private int currentWaitingPhase = 1;

  public Barrier(int maxThreads) {
    this.maxThreads = maxThreads;
  }

  @Override
  public synchronized void await() throws InterruptedException {
    // Threads bleiben in await, solange ihre waitingPhase aktiv ist.
    // Eine Phase ist beendet, wenn entweder maxThreads viele Threads
    // await aufgerufen haben oder aber freeAll aufgerufen wird.
    final int myPhase = currentWaitingPhase;
    ++threadsInCurrentWaitingPhase;
    if (threadsInCurrentWaitingPhase == maxThreads) {
      freeAll();
    }
    while (currentWaitingPhase == myPhase) {
      this.wait();
    }
  }

  @Override
  public synchronized void freeAll() {
    // Durch Erhöhen von currentWaitingPhase verlassen alle Threads die
    // await-Methode, da ihre Phase kleiner ist. Alle Threads, die await
    // nach diesem freeAll aufrufen, haben dann eine neue waitingPhase.
    ++currentWaitingPhase;
    threadsInCurrentWaitingPhase = 0;
    this.notifyAll();
  }
}
```

4 Speicherkonsistenz: Happens-before-Beziehung

Die Happens-before-Beziehung zwischen zwei Statements in Java garantiert, dass ein Statement den geschriebenen Wert des anderen Statements sieht. Fehlende Happens-before-Beziehungen können im Multi-Threading schwer ersichtliche Auswirkungen auf die Programm-Korrektheit haben.

In folgendem Codeausschnitt ist ein Programm zu sehen, das ein Ping-Pong Spiel zwischen zwei Threads dargestellt. Hierbei wird eine `ping`-Variable von dem PPingThread `pingThread` auf `true` gesetzt und eine entsprechende Ausgabe auf die Konsole erstellt. Der PPongThread `pongThread` setzt die `ping`-Variable auf `false` und erstellt ebenfalls eine Ausgabe auf die Konsole.

```
public class HappensBefore {
    public static boolean ping = false;
    public static final int maxRuns = 100;

    public static void main(String[] args) {
        Thread pingThread = new Thread(()-> {
            for(int i = 0; i < maxRuns; i++) {
                while(ping) {}

                ping = true;
                System.out.println("Ping␣-␣Round␣" + i);
            }
        });
        Thread pongThread = new Thread(()-> {
            for(int i = 0; i < maxRuns; i++) {
                while(!ping) {}

                ping = false;
                System.out.println("Pong␣-␣Round␣" + i);
            }
        });

        pingThread.start();
        pongThread.start();
    }
}
```

Aufgabe: Bei mehrfacher Ausführung des Programmes terminiert es in einigen Fällen nicht. Wieso kommt es zu diesem Verhalten? Verwenden Sie für Ihre Erklärung als Argumentationsgrundlage Happens-before-Beziehungen. Wie lassen sich die Probleme beheben?

Beispiellösung: Es besteht die Möglichkeit, dass das Programm in einer Endlos-Ausführung stecken bleibt (es terminiert nicht). Der Grund hierfür ist, dass zwischen dem Lesen und Schreiben von `ping` keine Happens-before-Beziehung besteht. Das heißt, dass es möglich ist, dass in den while-Schleifen ein nicht aktueller Wert von `ping` gelesen werden kann. Ein Grund hierfür ist das Caching der `ping`-Variable in einem Prozessor. Wird der Wert der Variable in einem Prozessor-Cache verändert und nicht zurückgeschrieben, kann der andere Thread diese Änderung nicht lesen. Weiterhin ist es möglich, dass der Wert nur aus dem Cache gelesen wird ohne dass der Wert im Cache mit den anderen Speichern synchronisiert wird. Hierdurch kann ein eventuelles Schreiben eines anderen Prozessors (sogar in den Hauptspeicher) ebenfalls unentdeckt bleiben. Beispielsweise liest `pingThread` den von ihm selbst geschriebenen `true`-Wert von `ping`, wohingegen `pongThread` den von ihm gesetzten `false`-Wert

von `ping` liest. Durch das Lesen der alten werte, werden die while-Schleifen nicht verlassen.

Dieses Problem kann gelöst werden, in dem die `ping`-Variable als `volatile` deklariert wird. Hierdurch wird eine Happens-before-Beziehung zwischen jedem Schreiben der Variable und jedem folgenden Lesen der Variable hergestellt. In *vielen* Implementierungen der Java Virtual Machine wird hierbei erzwungen, dass die Werte direkt in den Hauptspeicher geschrieben und aus diesem gelesen werden. Dies ist jedoch nur eine Lösung und laut der Java-Dokumentation sind auch andere Realisierungen möglich (z.B. die Ausnutzung von Cache-Kohärenz-Protokollen, falls vorhanden).

5 Fortgeschrittene Parallelisierungsprinzipien: Max Of Max

Gegeben sei die Klasse MaxOfMax welche die `calculateMax` bereitstellen soll. Die `calculateMax` Methode soll den größten Wert einer Sequenz von `Integer`-Zahlen parallel berechnen. In einem Vorverarbeitungsschritt wird die Sequenz in disjunkte Blöcke aufgeteilt und über den Parameter `blocksOfNumbers` der Methode übergeben. Für jeden Block soll der größte Wert berechnet und folgend der größte Wert über die Ergebnisse aller Blöcke bestimmt werden. Der größte Wert einer Folge von `Integer`-Zahlen wird mittels der ebenfalls bereitgestellten Methode `findMax` berechnet. Gehen Sie in den folgenden Aufgaben davon aus, dass alle Datenstrukturen und `Integer`-Objekte in dem Parameter `blocksOfNumbers` instanziiert sind, weswegen Sie das Auftreten und Behandeln von `null`-Werten nicht berücksichtigen müssen.

Aufgabe: In dieser Aufgabe sollen Sie Teile der Klasse MaxOfMax implementieren. Verwenden Sie hierfür die Source-Code Datei `MaxOfMax.java`, welche Ihnen im ILIAS-Kurs zur Verfügung gestellt wird.

1. Vervollständigen Sie die Implementierung der `calculateMax`-Methode in der Datei `MaxOfMax.java` unter Verwendung von `ExecutorService` und `Futures`. Das Auftreten von Exceptions muss nicht beachtet werden.

   Beispiellösung:

```
package edu.kit.kastel.sdq.parallelsum;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class MaxOfMax {
    public int calculateMax(Collection<List<Integer>>
       blocksOfNumbers,
                            int numberThreads) throws
                               ExecutionException,
                               InterruptedException {

        if (blocksOfNumbers.size() == 0) {
            return Integer.MIN_VALUE;
        }

        List<Integer> results = new ArrayList<>();


        // Beginn Lösung!

        List<Future<Integer>> futures = new ArrayList<>();
        ExecutorService executor =
           Executors.newFixedThreadPool(numberThreads);
```

```
        for (List<Integer> numberBlock : blocksOfNumbers) {

            futures.add(executor.submit(() -> {
                return findMax(numberBlock);
            }));
        }

        for (Future<Integer> future : futures) {
            results.add(future.get());
        }

        executor.shutdown();

        // Ende Lösung!

        return findMax(results);
    }

    private Integer findMax(Collection<Integer> numbers) {
        Integer maxValue = Integer.MIN_VALUE;
        for (Integer number : numbers) {
            if (number > maxValue) {
                maxValue = number;
            }
        }
        return maxValue;
    }

    private Integer findMaxStream(Collection<Integer> numbers) {
        return numbers.parallelStream()
        .mapToInt(x -> x.intValue())
        .max()
        .orElse(Integer.MIN_VALUE);
    }
}
```

2. Ergänzen Sie die Methode `findMaxStream` in der Datei `MaxOfMax.java`, welche als Ersatz für die Methode `findMax` dienen soll. Verwenden Sie hierfür parallele Java Streams und die Stream-Operation `max`.
   Hinweis: Beachten Sie den Rückgabe-Typ der Stream-Operation `max` des von Ihnen verwendeten Stream-Typs. Der Rückgabe-Typ von `Stream::max` ist beispielsweise ein `Optional<T>`[1].

   Beispiellösung:

```
public Integer findMax(Collection<Integer> numbers){
        return numbers.parallelStream()
                        .mapToInt(x -> x.intValue())
                        .max()
                        .orElse(Integer.MIN_VALUE);
}
```

[1] https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html