Tutorium 09: Parallelität in Java

Paul Brinkmeier

19. Januar 2020

Tutorium Programmierparadigmen am KIT

Heutiges Programm

Parallelprogrammierung

ProPa-Stoff zu Parallelprogrammierung:

- Grundlegende Begriffe
- Message Passing, wurde in OS kurz behandelt ("message queues")
- Shared Memory + Synchronisierung, wie in SWT1, OS, etc.
 - In Java, mit ein paar Details zur JVM ← wir sind hier
- Dieses Jahr: Manuelles Threading und Monitore in Java (siehe bspw. auch SWT1) sind nicht Bestandteil der VL.

Parallelprogrammierung

Heute: Verschiedene Java-/Parallelprogrammierungskonzepte

- Amdahlsches Gesetz
- Lambdas, @FunctionalInterface
- Threads (.start(), .run())
- Happens-before-Beziehung

Wiederholung

Flynns Taxonomie

- SISD: Single Instruction, Single Data
 Ein Datum wird von einer Ausführungsarbeit bearbeitet
- SIMD: Single Instruction, Multiple Data
 Eine Ausführungseinheit bearbeitet mehrere Daten gleichzeitig
- MIMD: Multiple Instruction, Multiple Data
 ≈ Mehrere Ausführungseinheiten arbeiten gleichzeitig
- MISD: Multiple Instruction, Single Data
 ≈ Mehrere Ausführungseinheiten arbeiten gleichzeitig an einem Datum

Daten- und Taskparallelismus

Parallele Probleme sind üblicherweise entweder

- "datenparallel": Problem kann auf identische Ausführungseinheiten verteilt werden Beispiel: map primeFactors [1432793, 651433, ...]
- "taskparallel": Problembestandteile sind nicht homogen Beispiel: Videospiel mit Render-, Netzwerk- und Logikprozessen

Datenparallele Probleme sind i.d.R. einfacher zu behandeln (auch: "embarrassingly parallel"). Bei manchen Problemen verschwimmt die Grenze auch (bspw. Webserver).

MPI

MPI ("Message Passing Interface") ist ein Standard für Parallelprogrammierung. Es existieren verschiedene Implementierungen für verschiedene Sprachen. Die VL verwendet Open MPI, eine Open-Source-Implementierung.

- MPI-,, Prozesse" beziehen sich i.d.R. auf Prozessorkerne
- Message Passing statt Shared Memory:
 - Daten werden explizit über Send und Recv geteilt
- MPI-Prozesse werden in sog. Communicators eingeteilt. Wir verwenden immer den Communicator, der alle Prozesse enthält (MPI_COMM_WORLD))

MPI: Kollektive Operationen

Statt Send und Recv nutzt man in MPI meistens "kollektive Operationen".

- Selber Aufruf in jedem Prozess
- Meistens mit root Parameter, um Datenquelle zu bestimmen

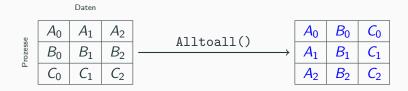
Beispiel: Bcast verteilt ein Datum auf alle Prozesse.



Cheatsheet: Liste an kollektiven MPI-Operationen

Folgende kollektiven Operationen kennen wir:

- MPI_Bcast
- MPI_Scatter, MPI_Gather
- MPI_Allgather ("Gather" + Bcast)
- MPI_Alltoall ("transponiert")
- MPI_Reduce ("wie fold")



Gegeben den parallelisierbaren Anteil eines Algorithmus $p \in [0,1]$, berechnet

$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{(1-p) + \frac{p}{n}}$$

den *maximalen Speedup* durch parallele Ausführung auf *n* Prozessoren.

- In der Praxis nicht erreichbar durch OS-Overhead!
- Trotzdem gute Annäherung für die etwaige Größenordnung des tatsächlichen Speedups (wenn man p kennt)

Gegeben den parallelisierbaren Anteil eines Algorithmus $p \in [0,1]$, berechnet

$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{(1-p) + \frac{p}{n}}$$

den *maximalen Speedup* durch parallele Ausführung auf *n* Prozessoren.

Beispielalgorithmus (Histogramm eines Graustufenbildes berechnen):

- Berechne Histogramme für Bildauschnitte (7s, parallelisierbar)
- Summiere einzelne Histogramme (3s, nicht parallelisierbar)

Gegeben den parallelisierbaren Anteil eines Algorithmus $p \in [0, 1]$, berechnet

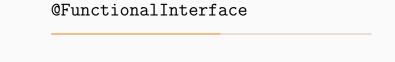
$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{(1-p) + \frac{p}{n}}$$

den *maximalen Speedup* durch parallele Ausführung auf *n* Prozessoren.

$$P = \frac{7s}{7s + 3s} = 0.7$$

$$T(n) = 0.3 + \frac{0.7}{n}$$

$$S(4) = \frac{1}{0.3 + 0.175} \approx 2.1$$



Lambdas

Seit Java 8 gibt es Lambda-Ausdrücke, bspw.:

```
Function<Float, Float> f = x -> 2 * x;
Float tau = f(pi);
```

Ein Lambda ist hier Syntaxzucker für eine anonyme Klassendeklaration (gabs auch schon vor Java 8):

```
Function<Float, Float> f =
  new Function<Float, Float>() {
    @Override
    public Float apply(Float x) {
      return 2 * x;
    }
    };
Float tau = f.apply(pi);
```

Das Interface Function

```
interface Function<A, B> {
   B apply(A x);
}
```

- java.util.function enthält alle möglichen solchen Interfaces (ziemlicher Clusterfuck)
- Eigene Typen für Lambdas?

@FunctionalInterface

Um eigene Typen für Lambdas zu definieren, können wir Interfaces mit einer einzelnen Methode schreiben und diese als @FunctionalInterface annotieren:

```
@FunctionalInterface
interface PixelTransformation {
  byte transform(byte input);
}

PixelTransformation bw =
  x -> x < 128 ? 0 : 255;</pre>
```

Das können wir verwenden, um nicht überall Function<A, B> stehen zu haben.

Threads in Java

Manuelle Threadverwaltung in Java

```
Runnable r = ...; (new Thread(r)).start();
// Runnable is ein functional interface:
Thread t = new Thread(() -> {
   calculate999999thPrime();
});
t.start();
t.join();
```

- Functional interfaces k\u00f6nnen f\u00fcr ad-hoc Threads verwendet werden
- Threadverwaltung:
 - t.start() lässt den Thread t anlaufen
 - t.join() wartet bis t durchgelaufen ist
 - Außerdem: interrupt()/isInterrupted()



- Programm lädt Graustufenbild als Array von Bytes (0 = schwarz, 255 = weiß)
- Histogramm: Wie oft kommen alle Grauwerte als Pixel vor?
- Parallelisiert den Code in demos/java/Histogram.java!
 - Verwendet Thread, start() und join()
 - Vergleicht die resultierende Zeit mit der nicht parallelisierten Version.
 - Wieviele Kerne hat euer Rechner? Lohnt es sich, mehr Threads zu starten als Kerne verfügbar sind?



- Programm lädt Graustufenbild als Array von Bytes (0 = schwarz, 255 = weiß)
- Histogramm: Wie oft kommen alle Grauwerte als Pixel vor?
- Parallelisiert den Code in demos/java/Histogram.java!
 - Verwendet Thread, start() und join()
 - Vergleicht die resultierende Zeit mit der nicht parallelisierten Version.
 - Wieviele Kerne hat euer Rechner? Lohnt es sich, mehr Threads zu starten als Kerne verfügbar sind?

Happens-before

Happens-before-Beziehung

poopy

Ende

Ende

- Im Campus-System könnt ihr euch bis zum 23.03. für die PP-Klausur anmelden
 - Termin: 09.04.2020 um 17:00, Zelt auf dem Forum :(
- Rückmelden ist möglich bis zum 15.02.