# Grundlagen der Parallelisierung

Philipp Kegel, Sergei Gorlatch

AG Parallele und Verteilte Systeme
Institut für Informatik
Westfälische Wilhelms-Universität Münster

3. Juli 2009

# Inhaltsverzeichnis

# Warum Parallelität?

**Idee:** mehrere „Recheneinheiten“ (Prozessoren, Computer, etc.) arbeiten gleichzeitig, d. h. *parallel* an einer gemeinsamen Aufgabe

**Motivation:**

- Beschleunigung von Berechnungen für viele Anwendungsfelder nötig (*Grand Challenges*, z. B. Simulation von Wetter und Klimawandel, Medizinische Bildgebung, Windkanal, Motorenkonstruktion, Nukleartests, etc.)
- Erfüllung von großen Speicheranforderungen
- Höherer Durchsatz

# Warum Parallelität?

**Beobachtung:**

- Fortschritte bei Netzwerken: Geschwindigkeit, Bandbreite, etc.
- Fortschritte bei Prozessoren: kleiner, billiger

**aber**

- Fortschritte bei Einzelprozessorleistung werden wegen der natürlichen Lichtgeschwindigkeitsgrenze immer schwieriger:

  Beispiel: Prozessor 6 GHz $\Rightarrow$ 1 Zyklus in 0,166 ns.
  Das Licht legt in dieser Zeit nur 5 cm zurück!

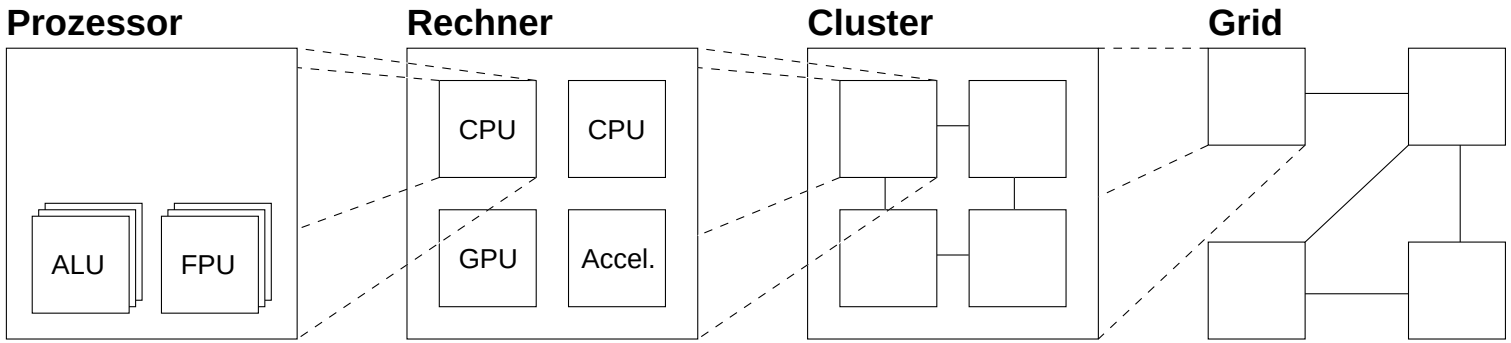
**Fazit:** Die Leistung eines Prozessors nähert sich der physikalischen Grenze. Dagegen wird es immer einfacher, viele Prozessoren miteinander zu verbinden

# Parallele Systeme

## Ebenen von Paralleltät

Parallelität durch Verbindung mehrere „Recheneinheiten“ auf verschiedenen Ebenen angesiedelt werden:

1. mehrere Funktionseinheiten eines Prozessors
2. mehrere Prozessoren eines Rechners
3. mehrere Rechner eines Clusters
4. mehrere Rechner/Cluster eines LAN/WAN (Computational Grids)

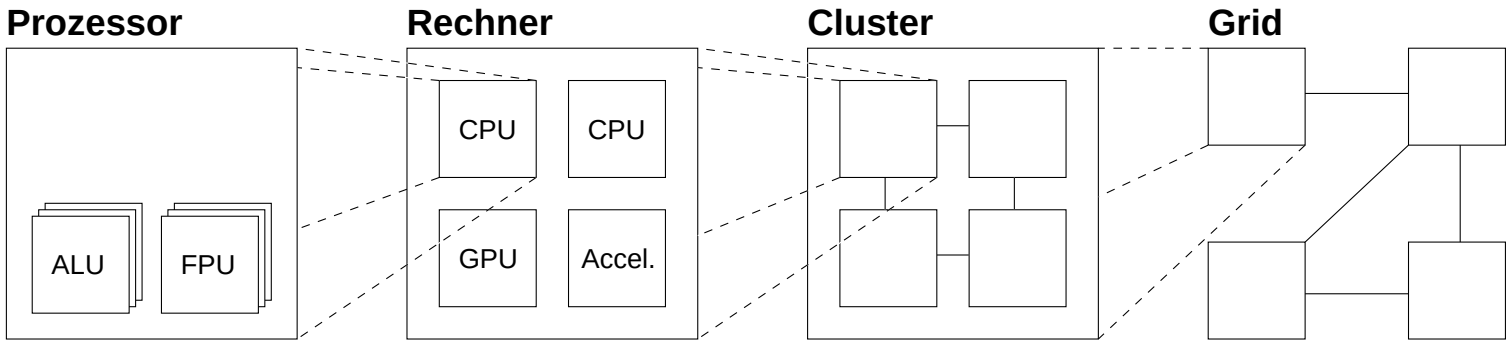


# Parallele Systeme

## Ebenen von Paralleltät

Parallelität durch Verbindung mehrere „Recheneinheiten“ auf verschiedenen Ebenen angesiedelt werden:

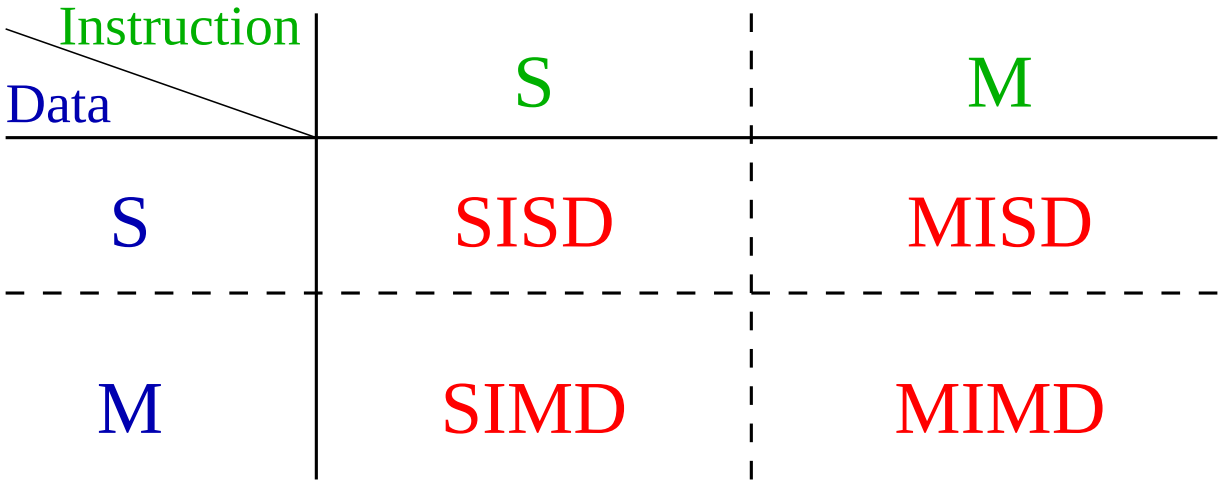
1. **mehrere Funktionseinheiten eines Prozessors**
2. **mehrere Prozessoren eines Rechners**
3. mehrere Rechner eines Clusters
4. mehrere Rechner/Cluster eines LAN/WAN (Computational Grids)



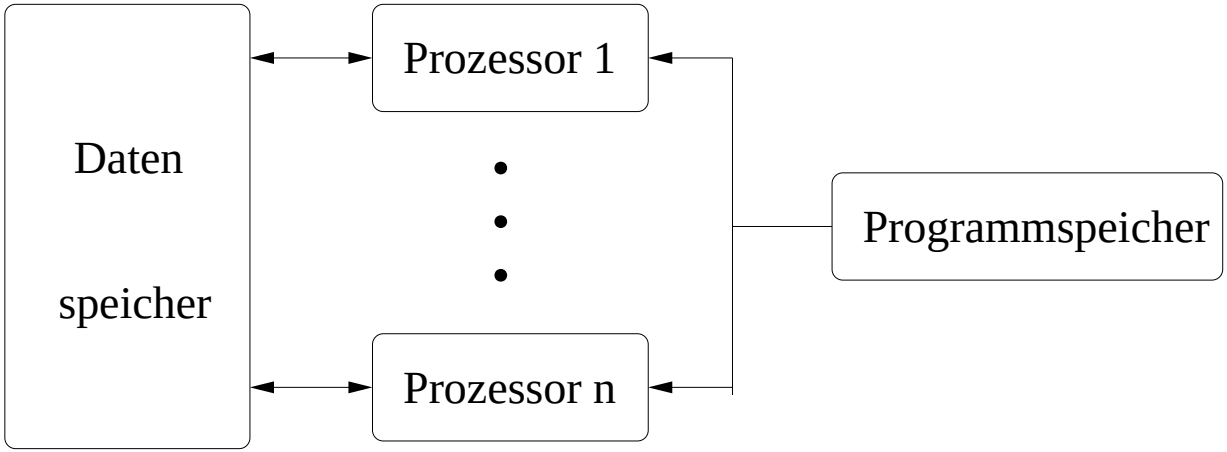

**aktueller Trend: Multi-/Many-Core-Architekturen**

# Klassifizierung nach M. Flynn [1972]

- Klassifiziert wird nach zwei Merkmalen:
  - **I** (*Instruction*): Organisation des Kontrollflusses
  - **D** (*Data*): Organisation des Datenflusses
- Jedes Merkmal kann einen der zwei Werte annehmen:
  - **S**: für *Single*
  - **M**: für *Multiple*
- Somit erhält man 4 mögliche Kombinationen:



# Single Instruction Multiple Data (SIMD)



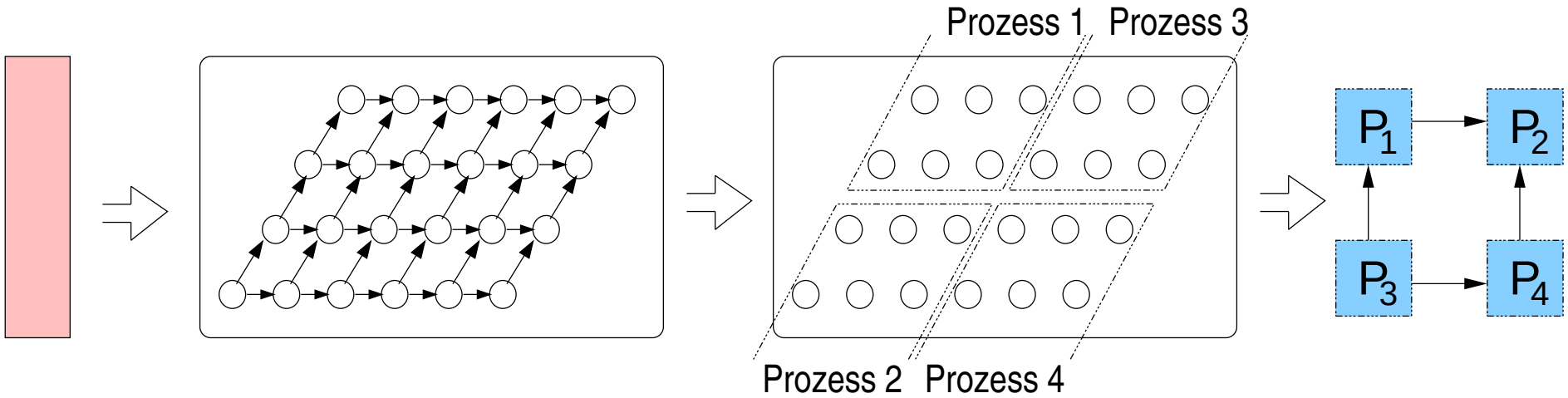

Ausprägungen der SIMD-Architektur:

- Vektorrechner (klassische SIMD-Architektur, z.B. Cray-2)
- SIMD Within A Register (SWAR): SIMD-Befehlserweiterung aktueller Prozessoren, z. B. SSE (Intel) oder 3DNow! (AMD)
- **GPUs**

# Parallelisierung von Programmen

**Parallelisierung = die Umwandlung eines sequentiellen Algorithmus/Programms zur parallelen Ausführung**

Wesentliche Schritte bei der Parallelisierung:




**Zerlegung** **Scheduling** **Mapping**

# Parallelisierung von Programmen

## Daten- vs. Taskparallelität

Möglichkeiten der parallelen Ausführung:

- **Datenparallelität:** Eine Operation parallel auf mehrere Elemente einer Datenstruktur anwenden
- **Taskparallelität:** Mehrere voneinander unabhängige (Gruppen von) Anweisungen (Tasks) eines Programm parallel ausführen
- Einzelne Tasks können ggf. von *mehreren* Prozessoren, d. h. datenparallel ausgeführt werden

**Grundidee:** Erzeugen von Berechnungsströmen, die auf mehreren Prozessoren gleichzeitig zur Ausführung gelangen und durch koordinierte Zusammenarbeit eine Anwendung implementieren

# Grenzen der Parallelisierung

Nicht alle Anwendungen können effizient parallelisiert werden.
Beispiel: Newtonsches Näherungsverfahren.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

**Problem:** Die Näherungswerte $x_n$ der Nullstelle können nur einer nach dem anderen, d. h. *sequentiell* berechnet werden.

- Typisches Problem vieler iterativer Verfahren

# Kosten der Parallelisierung

*Parallelisierungs-Schwierigkeiten* entstehen, wenn

- die Erzeugung der Berechnungsströme
- die Abbildung der Ströme auf die vorhandenen Rechnerarchitekturen
- die Organisation der Zusammenarbeit der Prozessoren

für konkrete Anwendung kompliziert sind und zusätzliche Aktivitäten bei der Programmausführung erfordern

Diese Schwierigkeiten bezeichnet man als **Overhead** (Mehrkosten).

# Kosten der Parallelisierung

## Ursachen für Overhead

- Die Anwendung besitzt niedrigen *Parallelitätsgrad*
- Die Berechnungsströme haben gegenseitige *Abhängigkeiten*
  - Die Zusammenarbeit erfordert *Synchronisation* $\Rightarrow$ Wartezeiten
  - Die Zusammenarbeit erfordert *Kommunikation* $\Rightarrow$ Extra-Arbeit
- Prozessoren nicht gleich ausgelastet $\Rightarrow$ *Load balancing* nötig
- Die Minimierung der Kommunikation und der Lastausgleich sind oft entgegengesetzt $\Rightarrow$ intelligentes *Scheduling* nötig
- *Granularität* der Teilaufgaben:
  - Hoher Parallelitätsgrad erfordert „feinkörnige“ Aufgaben
  - Minimierung der Kommunikation erfordert „grobkörnige“ Aufgaben

# Speedup

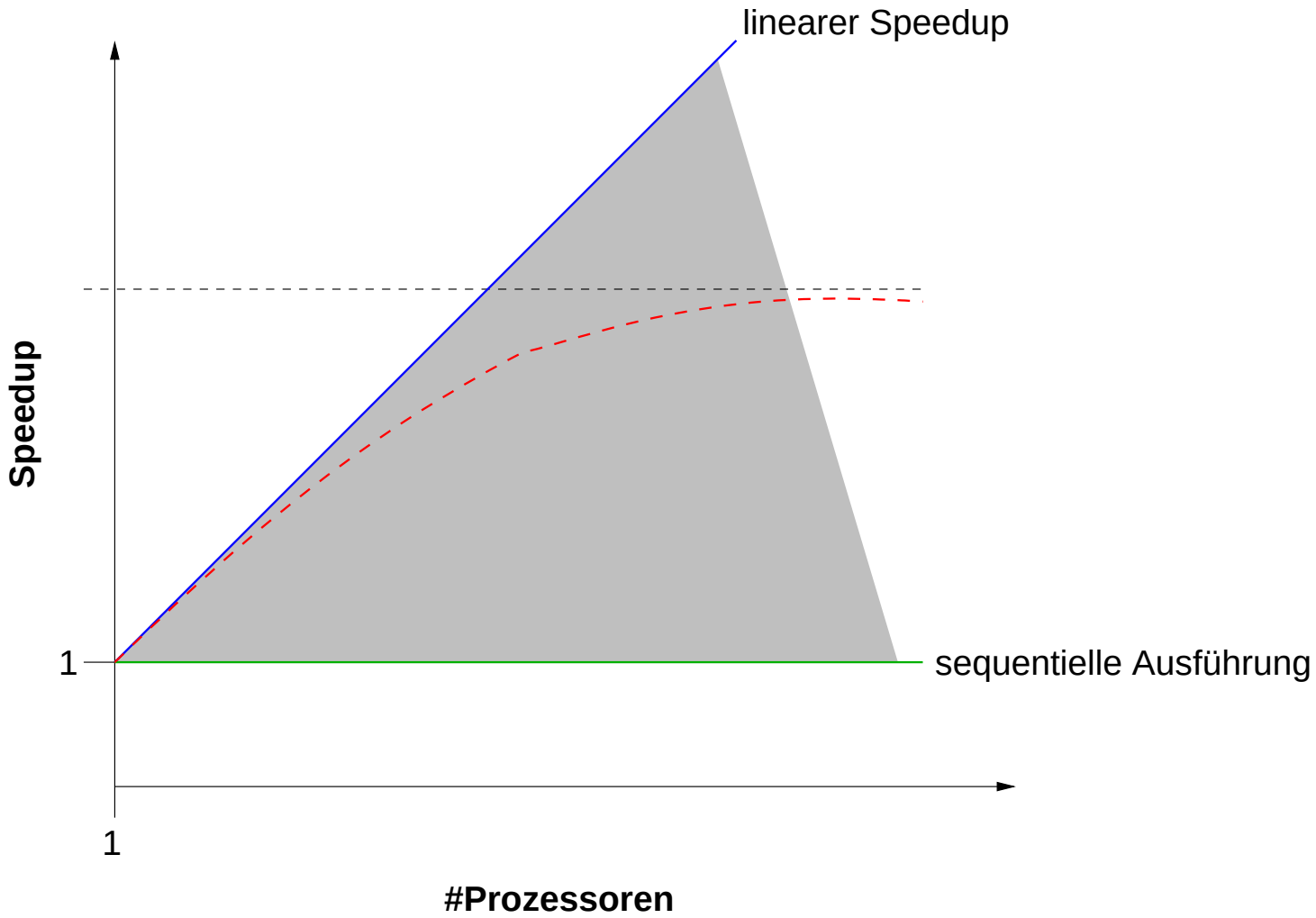
Ein gängiges Maß für den relativen Geschwindigkeitsgewinn durch Parallelisierung ist der sogenannte **Speedup** $S_p$.

$$S_p = \frac{T_1}{T_p}$$

wobei
$p$ – die Anzahl der verwendeten Prozessoren,
$T_p$ – die Laufzeit der Anwendung auf $p$ Prozessoren,
$T_1$ – die Laufzeit *derselben* Anwendung im sequentiellen Fall, d. h. auf einem Prozessor.

**Intuitiv:** Speedup $S_p$ ist der relative Geschwindigkeitsvorteil gegenüber der sequentiellen Implementierung, der durch den Einsatz von $p$ Prozessoren entsteht.

# Speedup


linearer Speedup
Speedup
1
sequentielle Ausführung
1
#Prozessoren

# Amdahlsches und Gustafson-Gesetz

Es werden zwei wesentliche Fälle unterschieden:

- Der inhärent sequentielle, d. h. **nicht parallelisierbare Anteil einer Anwendung ist konstant** (Amdahlsches Gesetz [1967])

  Das heißt, wenn z. B. 90% einer Anwendung perfekt parallelisierbar ist, kann diese Anwendung höchstens um Faktor 10 beschleunigt werden, egal wieviele Prozessoren man einsetzt.

- Der inhärent sequentielle Anteil einer Anwendung hat **konstante Laufzeit**, unabhängig von der Anwendungsgröße (Gustafson-Gesetz [1988])

# Amdahlsches und Gustafson-Gesetz

Es werden zwei wesentliche Fälle unterschieden:

- Der inhärent sequentielle, d. h. **nicht parallelisierbare Anteil einer Anwendung ist konstant** (Amdahlsches Gesetz [1967])

  Das heißt, wenn z. B. 90% einer Anwendung perfekt parallelisierbar ist, kann diese Anwendung höchstens um Faktor 10 beschleunigt werden, egal wieviele Prozessoren man einsetzt.

- Der inhärent sequentielle Anteil einer Anwendung hat **konstante Laufzeit**, unabhängig von der Anwendungsgröße (Gustafson-Gesetz [1988])

  Solche Anwendungen heißen **skalierbar**: man kann für wachsende Größe der Anwendung immer mehr Prozessoren einsetzen, mit steigendem Speedup.

## Parallelisierende Compiler

- Idee: Sequentielles Programm $\rightarrow$ Parallelprogramm
- Schwierigkeiten: *automatische* Analyse der Abhängigkei- ten, Erreichen einer regelmäßigen Lastverteilung zwischen den Prozessoren, Kommunikationsorganisation
- Grundproblem: Evtl. "unnötige" Abhängigkeiten im sequentiellen Programm erschweren Parallelisierung

$\Rightarrow$ Implizite Parallelisierung ist nur eingeschränkt anwendbar,
z. B. bei funktionalen Programmiersprachen:

- Keine vorgeschriebene Ausführungsreihenfolge, keine Seiteneffekte
  $\Rightarrow$ Funktionsargumente parallel auswertbar

In der Praxis erfolgt die Parallelisierung explizit.
Bekannte Programmiermodelle sind zum Beispiel:

- **Message Passing Interface (MPI):**
  Bibliotheksfunktionen für nachrichtenbasierte *Kommunikation* und *Synchronisation* von Prozessoren
- **OpenMP:**
  Definition paralleler Programmteile (insbesondere datenparallele Schleifen) durch Compilerdirektiven
- **CUDA:**
  Programmierung von GPUs durch Befehlserweiterungen der Sprache C

# Zusammenfassung

- Aktuelle Leistungsanforderungen lassen sich nicht mehr allein durch Performancesteigerungen sequentieller Prozessoren erfüllen
- SIMD-Architekturen gewinnen durch GPUs wieder an Bedeutung
- Parallelisierung hat Grenzen
  - Nicht alle Anwendungen sind effizient parallelisierbar
  - Parallelisierung erzeugt i. d. R. zusätzlich Berechnungsaufwand (Overhead)
- Automatische (effiziente) Parallelisierung ist nur in einigen Ausnahmefällen möglich

  $\Rightarrow$ Programmierer muss bereits bei der Entwicklung „parallel denken“

# Fragen?