

Leistungsbewertung:

Amdahl - Gesetz

der erzielbare Speedup ist durch die Anzahl nicht parallelisierbarer Operationen begrenzt.

$$S(n) = \frac{n}{1 + (n-1) \cdot f} \leq \frac{1}{f}$$

↳ # nicht parallelisierbare Operationen

$$T_n = \frac{1}{m} \sum_{i=1}^m \frac{1}{R_i}$$

$$\begin{aligned} R_1 &= 1 & f \\ R_N &= n & 1 \cdot f \end{aligned}$$

$$S = \frac{T_1}{T_N} = \frac{1}{\frac{1}{1} + \frac{1-1}{n}}$$

Dieses Gesetz geht davon aus, dass es entweder voll sequentielle oder voll parallelisierbare Operationen gibt;

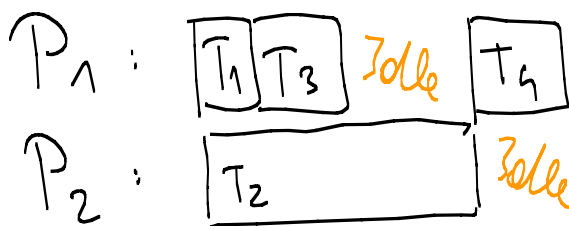
⇒ je genauer das Laufzeitprofil, desto genauer der Speedup.

# Parallelisierung von Programmen:

Positionieren: Teilen des Programmes in einzelne Teile

Mapping: Teile den Prozessoren zuteilen

Scheduling: Zeitlicher Ablauf der Teile  
⇒ Problematisch ist eine Datenabhängigkeit (Übergabewerte, ...)  
Teil<sub>i</sub> erst starten wenn T<sub>j</sub> abgeschlossen;  
⇒ Darstellung im Gantt-Diagramm



I/O Abhängigkeit: Ein Prozessor will auf eine I/O-Einheit zugreifen die von einem anderen Prozessor verwaltet wird;  
⇒ warten bis dieser Zeit hat.

## Datenabhängigkeiten:

Datenflussabhängigkeit:

↓  
Eingangsvariable ist Ausgangsvariable eines anderen Tasks (z.B.: Übergabewerte)

⇒ Darstellung im Gantt-Diagramm

Symbol

Data independency:

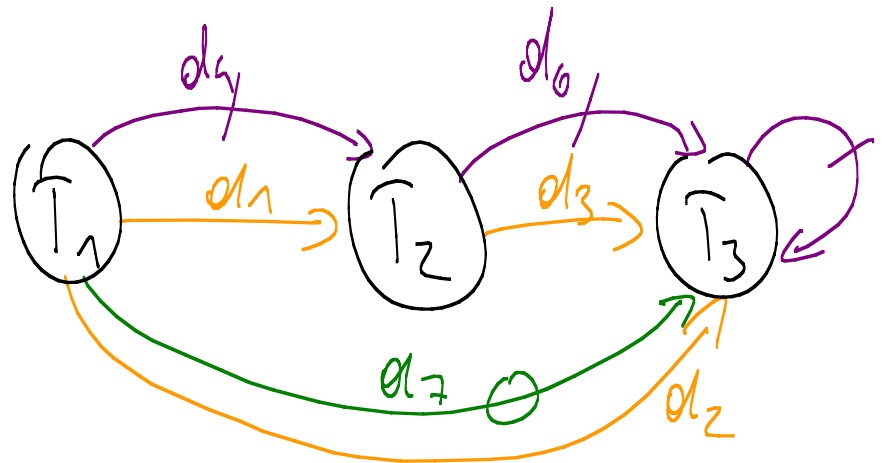
⊥ read before write (Ausgangsvar. ist Eingangsvariable eines anderen Tasks)

Datenausgangsabhängigkeit

⊕ write before write

Datenabhängigkeits-graph:

$$\begin{aligned} T_1 &: A = B + C \\ T_2 &: B = A + E \\ T_3 &: A = A + B \end{aligned}$$



- $d_2$ :  $T_1$  schreibt  $A$ ,  $T_2$  liest  $A$  (write before read)
- $d_7$ :  $T_1$  schreibt  $A$ ,  $T_3$  schreibt  $A$  (write before write)
- $d_6$ :  $T_2$  liest  $A$ ,  $T_3$  schreibt  $A$  (read before write)

⇒ nur wenn es keine Datenabhängigkeiten gibt können zwei Tasks parallel unabhängig ausgeführt werden

nach Bernstein

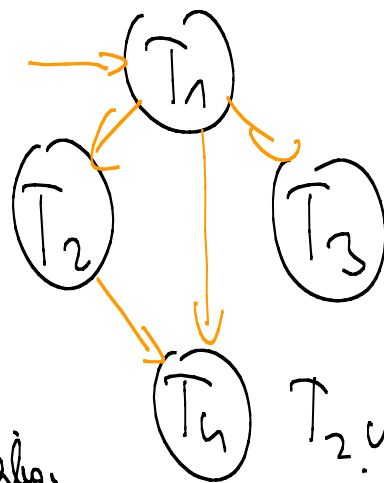
$$\begin{cases} I_1 \cap O_2 \stackrel{!}{=} \emptyset \\ I_2 \cap O_1 \stackrel{!}{=} \emptyset \\ O_1 \cap O_2 \stackrel{!}{=} \emptyset \end{cases}$$

alle inputs von 1 haben nichts gemeinsam mit outputs von 2

weiter kann mittels des Datenfluss-Graphen visualisiert werden:



nichts parallelisierbar



$T_2$  und  $T_3$  parallelisierbar

Will man auf Algorithmus-Ebene Abhängigkeiten erkennen reicht eine Datenfluss Analyse;

Viele Datenabhängigkeiten entstehen erst durch die Implementierung (da Variablen mehrfach belegt werden)

Software für verteilte Systeme:

sequentielle Sprache (z.B.: C);

→ für diese gibt es numerische Bibliotheken  
nur für Programmierer;  
eingeschränkte Funktionsbibliothek

→ Kommunikations Bibliotheken  
(sammle Daten und verteile sie auf Prozessor 1, 2)  
für den Anwender

→ Funktionsbibliothek  
(Prozess ... jeder hat seinen eigenen Speicher  
Threads -- alle sehen einen Speicher)

→ Programmiersprachen für Parallelisierung  
z.B.: CSP

Kontroll Modelle:

SIMD --- z.B.: auf Multi-RAI System  
(Datenparallelismus)  
es gibt nur einen Kontrollfluss

MIMD --- z.B.: Multicomputer System  
(Funktionsparallelismus)  
es laufen mehrere Kontrollflüsse  
nebeneinander die miteinander  
Daten austauschen

Granularität:

meist lässt man den Compiler auf

- Masinen - *sehr fein granular*
- Block -
- Prozedurebene

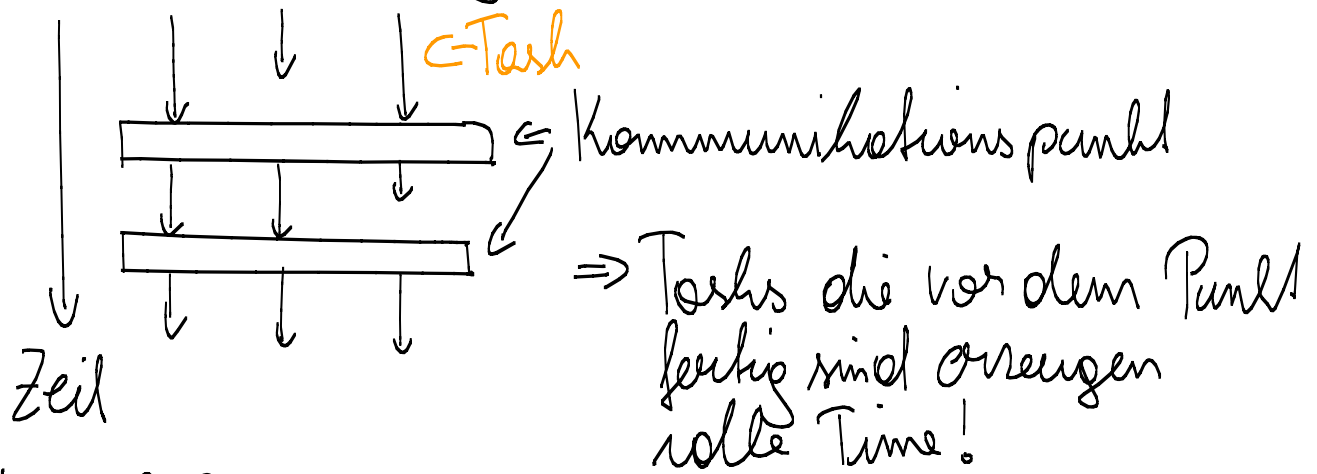
parallelisieren, der Programmierer / Betriebssystem:

- Thread -
- Prozessebene *grob granular*

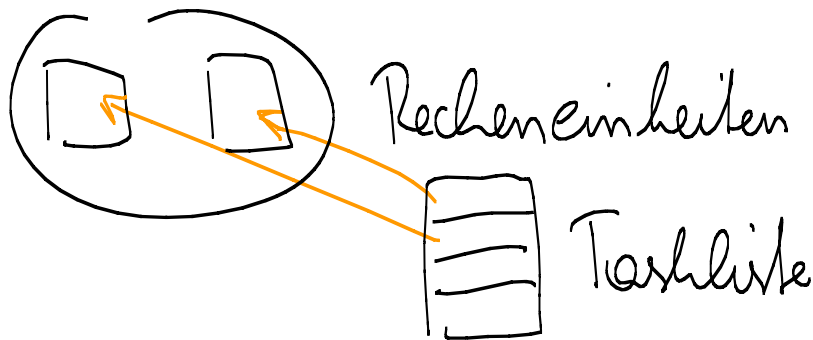
je feiner die Granularität, desto höher ist der  
Kommunikationsaufwand zwischen diesen Teilen;

# Rechenmodelle:

- Pipes & Filter
- Phasenweise Berechnung (Synchronisation)



- Divide & Conquer
- Task-Form



In der Taskliste stehen die Tasks, geht ein Prozessor auf volle bekommt er den nächsten Task, der aktuell geht zurück in die Tasklist;

=> keine Unterstützung für Kommunikation