Verteilung von Daten & Programmen

Inhalt

- Leistungsbewertung
- Parallelisierung
- Datenabhängigkeiten
- Software für verteilte Systeme

Leistungsbewertung

Bestimmung der Leistung (Performance)

- Wie kann die Leistung eines verteilten Systems bestimmt werden?

Kenngrößen (Auswahl)

- Ausführungszeit [sec]
- Instruktionen [MIPS]
- Floating-Point Op. [FLOPS]
- Bandbreite [Byte/s]

Bewertung/Messung mit Benchmarks

- Bsp.: LINPACK, SPEC, PARKBENCH etc.

Modellierung des parallelen Algorithmus

- Bsp.: Analyse anhand des PRAM Modells

Leistungsbewertung(2)

Bewertung der Ausführungszeit

Als Funktion des Parallelisierungsgrades und der Problemgröße

Leistungsgewinn (Speedup)

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

T(1) = seq. Ausführungszeit
T(n) = parallele Ausführungszeit bei n Prozessoren

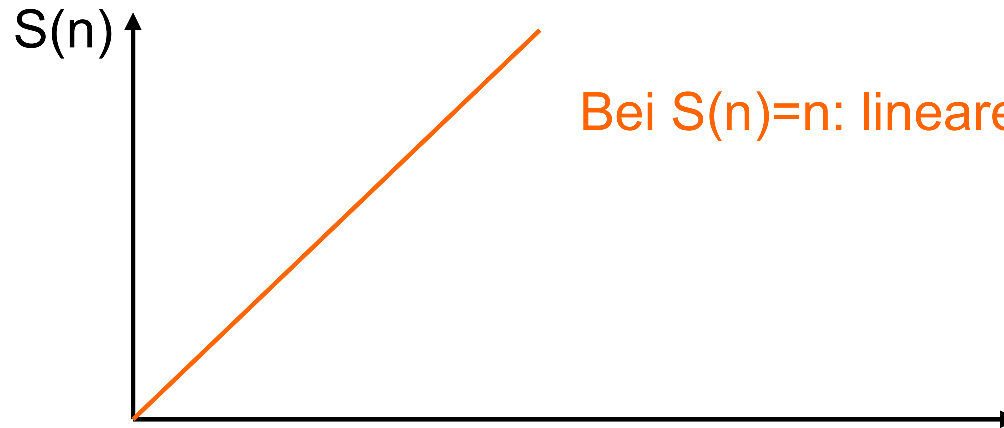
Wirkungsgrad, Auslastung (Efficiency)

$$E(n) = \frac{S(n)}{n}$$

Leistungsbewertung(3)

Grafische Darstellung des Speedups

$S(n) = f(n)$



Anzahl der Prozessoren n

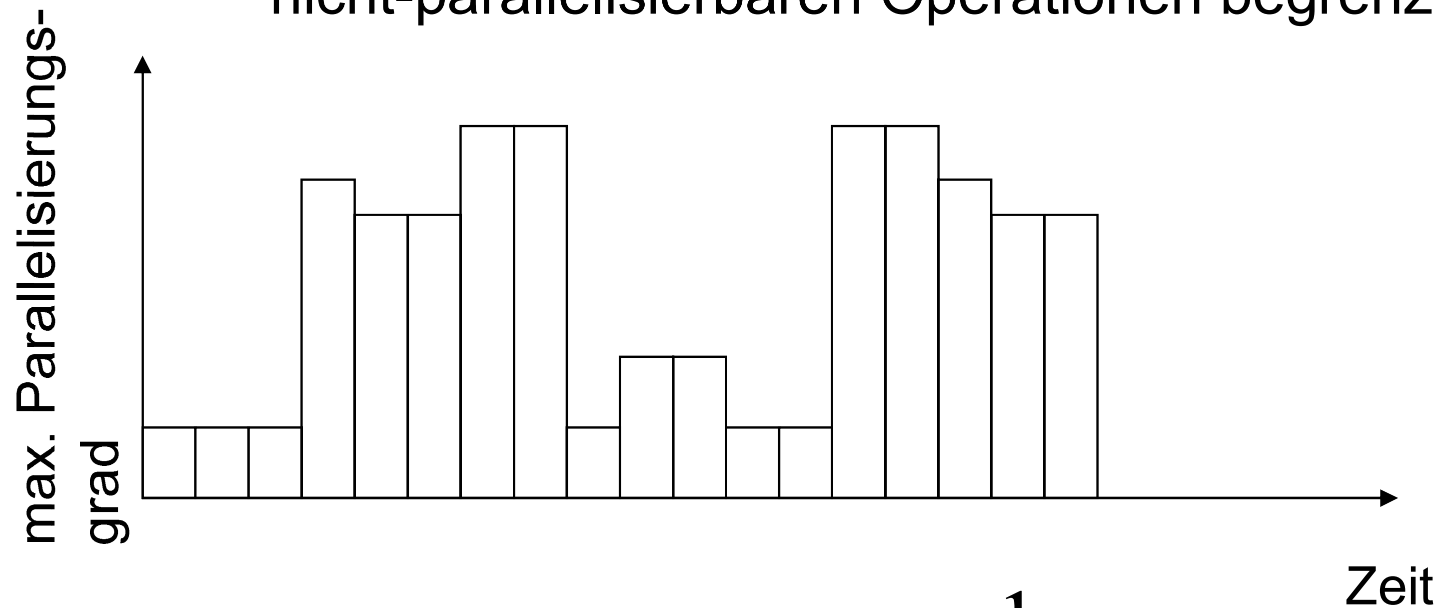
- Meistens ist **S(n)<n**, da die Anwendung nicht optimal parallelisierbar ist (Datenabhängigkeiten) und Kommunikationsoverhead auftritt.

- Bei **S(n) > n**: „superlinearer Speedup“ (zu hinterfragen !!)

Leistungsbewertung(4)

„Amdahl-Gesetz“

der erzielbare Speedup ist durch den Anteil der nicht-parallelisierbaren Operationen begrenzt.



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

f = Anteil der nicht-parallelisierbaren Operationen

„Parallelisierung" v. Programmen

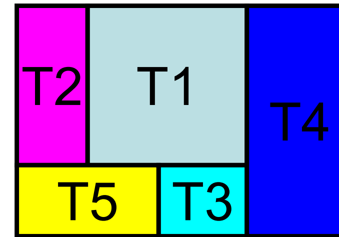
1. Partitionieren des sequentiellen Programms
   - Aufteilung in Teilprogramme (Tasks) anhand der Datenabhängigkeiten.

2. Zuordnen von Tasks zu Prozessoren (Mapping)
   - Welcher Task wird auf welchem Prozessor ausgeführt?

3. Bestimmen der zeitlichen Abfolge der Tasks (Scheduling)
   - Wann wird der Task auf dem Prozessor ausgeführt?

„Parallelisierung“ v. Programmen

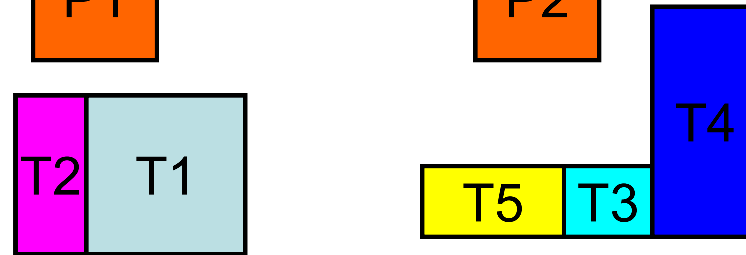
1. Partitionierung



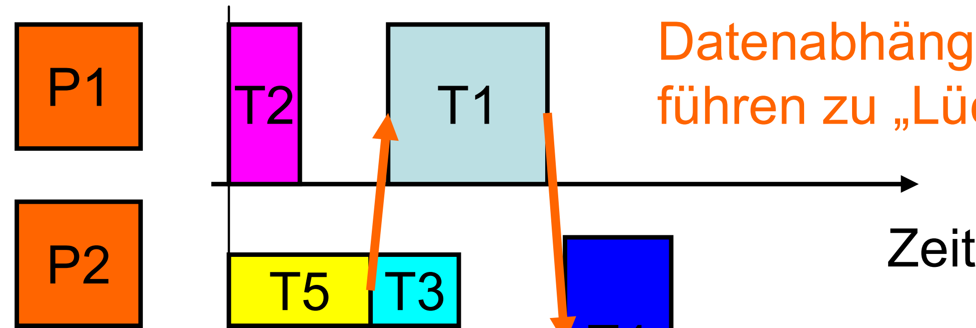
2. Mapping

Multiprozessor System

zugeordnete Tasks



3. Scheduling



Datenabhängigkeiten führen zu „Lücken“

Datenabhängigkeiten

Erkennung von Parallelität

– wann können Tasks unabhängig (parallel) voneinander ausgeführt werden?

Beschreibung von Tasks

– Tasks verarbeiten Eingabedaten zu Ausgabedaten
  ➔ Menge von Eingangsvariablen I
  ➔ Menge von Ausgangsvariablen O

Beispiel

– T1: A=B+C I1={B,C};O1={A}
– T2: B=A+E I2={A,E};O2={B}
– T3: A=A+B I3={A,B};O3={A}

Arten von Datenabhängigkeiten

Datenflussabhängigkeit

- Eingangsvariable ist eine Ausgangsvariable eines vorigen Task
- „write-before-read“

„Data antidependence“

- Ausgangsvariable ist eine Eingangsvariable eines vorigen Task
- „read-before-write“

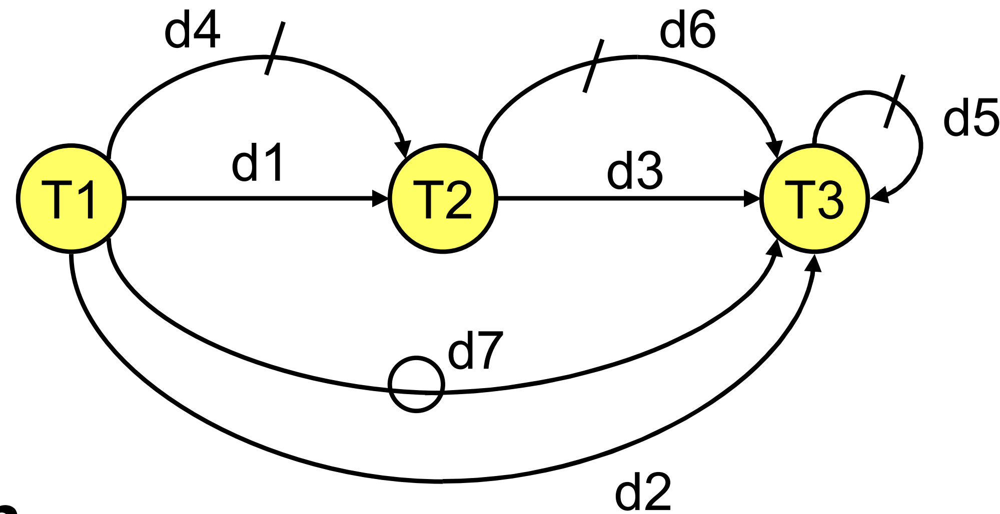
Datenausgangsabhängigkeit

- mehrere Tasks haben dieselbe Ausgangsvariable
- „write-before-write“

Datenabhängigkeitsgraph

Beispiel

T1: A=B+C
T2: B=A+E
T3: A=A+B



Abhängigkeiten

d1: $O1 \subseteq I2$ (Variable A)
d2: $O1 \subseteq I3$ (Variable A)
d3: $O2 \subseteq I3$ (Variable B)
d4: $O2 \subseteq I1$ (Variable B)
d5: $O3 \subseteq I3$ (Variable A)
d6: $O3 \subseteq I2$ (Variable A)
d7: $O3 \subseteq O1$ (Variable A)

„write-before-read“ (d1–d3)
„read-before-write“ (d4–d6)
„write-before-write“ (d7)

Erkennen von Parallelität

Bedingung für parallele Abarbeitung

- Es dürfen keine (Daten)abhängigkeiten zwischen den Tasks herrschen

„Bernstein" Bedingung

- Zwei Tasks können parallel ausgeführt werden (T1 || T2), wenn gilt:

$$I1 \cap O2 = \phi$$
$$I2 \cap O1 = \phi$$
$$O1 \cap O2 = \phi$$

➔ Tasks von vorigem Beispiel können nicht parallelisiert werden!

Erkennen von Parallelität (2)

Beispiel:

T1: C = D * E
T2: M = G + C
T3: A = B + C
T4: C = L + M
T5: F = G / E

Variablen der Tasks:

Eingangsvariablen	Ausgangsvariablen
I1 = {D, E}	O1 = {C}
I2 = {G, C}	O2 = {M}
I3 = {B, C}	O3 = {A}
I4 = {L, M}	O4 = {C}
I5 = {G, E}	O5 = {F}

Erkennen von Parallelität (3)

T1 || T2 ?

$I1 \cap O2 = \phi$
$I2 \cap O1 \neq \phi$ **Nicht parallelisierbar!**
$O1 \cap O2 = \phi$

T1 || T5 ?

$I1 \cap O5 = \phi$
$I5 \cap O1 = \phi$ **Parallelisierbar!**
$O1 \cap O5 = \phi$

Weiters

T2 || T3 }
T2 || T5 } **T2 || T3 || T5**
T3 || T5
T4 || T5

Datenflussgraph

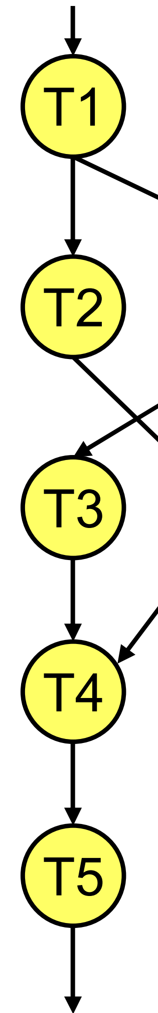
Grafische Darstellung

Knoten: Verarbeitungseinheiten (Tasks)

Kanten: Datenaustausch zwischen den Tasks (Variablen)

➔Datenflussabhängigkeiten erkennbar

➔Parallelisierbare Tasks erkennbar!



Software für verteilte Systeme

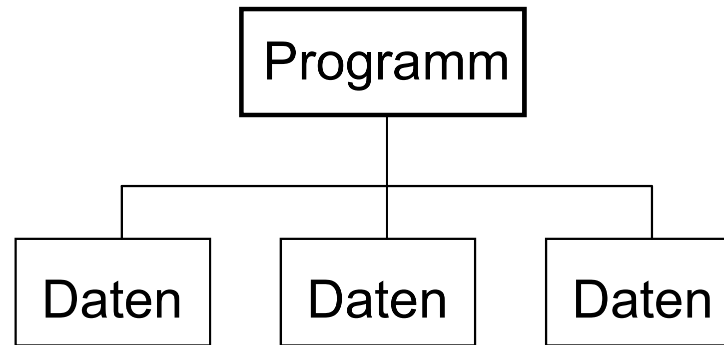
Verfahren zur Software-Erstellung für verteilte (parallele) Systeme

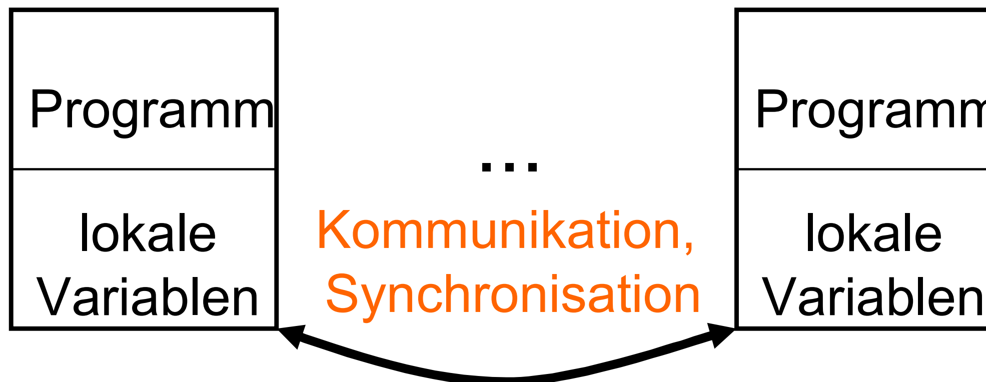
- 1. Numerische Bibliotheken für sequentielle Sprachen (nur für spezielle Funktionen)
- 2. Bibliotheken für Kommunikationsfunktionen Anwender für „Parallelisierung“ verantwortlich
- 3. Spezialbefehle für existierende Sprachen Bsp. Abspalten par. Prozesse, parallele Vektor Operationen etc.
- 4. Neue Sprachen für Parallelverarbeitung Bsp. OCCAM, CSP, Cilk

Software-Schwerpunkte

1. Control Models

Ein oder mehrere „threads of control“ (Kontrollfluß)

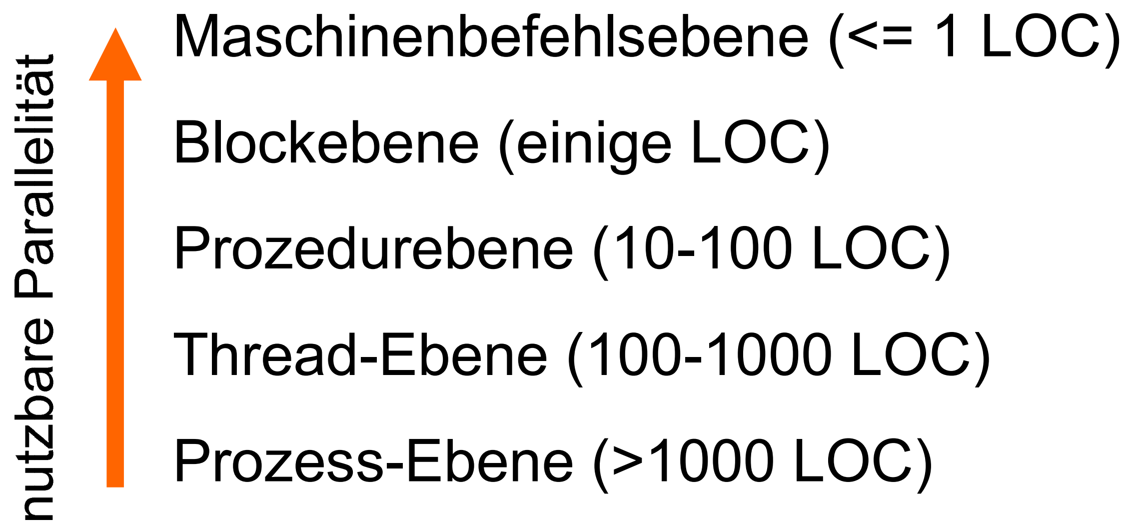




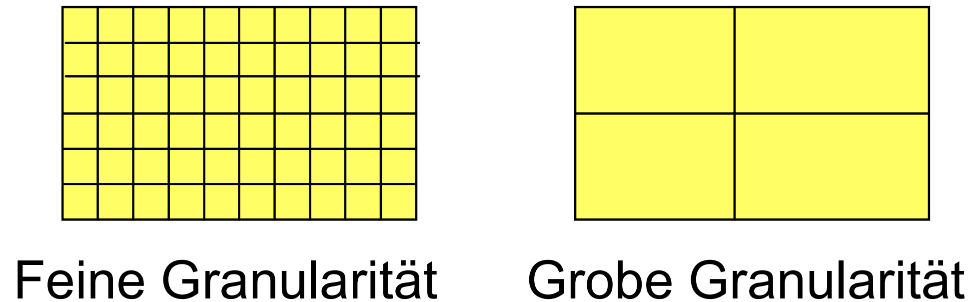
Software-Schwerpunkte(2)

2. Granularität

auf welcher Ebene wird die Parallelität ausgenutzt?



- Maschinenbefehlsebene (<= 1 LOC)
- Blockebene (einige LOC)
- Prozedurebene (10-100 LOC)
- Thread-Ebene (100-1000 LOC)
- Prozess-Ebene (>1000 LOC)



Feine Granularität   Grobe Granularität

Software-Schwerpunkte(3)

3. Rechenmodell

- Wie wird die Abarbeitung der Tasks/Threads organisiert?
- Single Program Multiple Data (wie SIMD)
- Pipes & Filters
- Phasenweise Berechnung (+Synchronisation)
- „Divide & Conquer“ (Aufspaltung in (kleinere) parallele Tasks)
- „Task Farm“ („freie“ Tasks warten in einer Warteschlage auf nächsten Job)

Software-Schwerpunkte(4)

4. Kommunikationsmethoden

- Wie können Daten zwischen Tasks ausgetauscht werden?
- Gemeinsame Variablen
- Botschaften
  - Punkt-zu-Punkt
  - Broadcast → einer an alle
  - Multicast →
- Synchrone/asynchrone Kommunikation

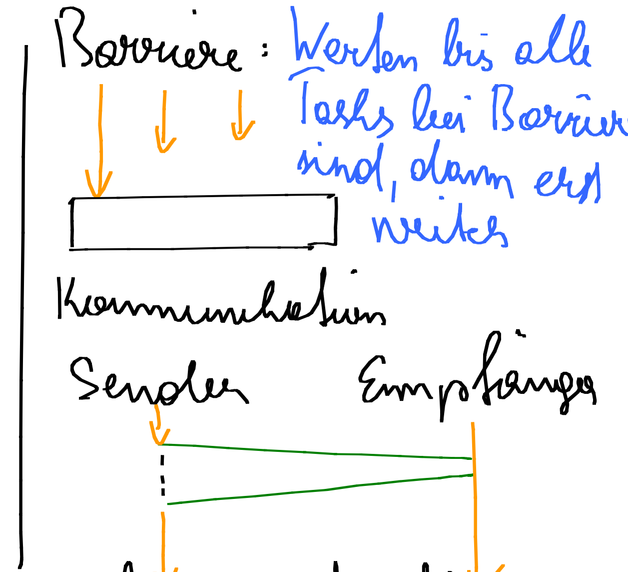
Software-Schwerpunkte(5)

5. Synchronisationsmethoden

- Wie kann die (zeitliche) Abfolge von Tasks erzielt werden?
- Semaphore (z.B. für wechselseitigen Ausschluss)
- „Barriere“ (Warten auf „langsamsten“ Task)
- Kommunikation kann auch zu Synchronisation führen (z.B. bei „Rendez-vous“)

Semaphore: eine Variable mit 2 Methoden up & down.
will ein Prozess schreiben, führt er down aus;
andere Zugreifer werden schlafen gelegt;
⇒ ist Zugriff beendet, up ausführen;
andere Zugreifer wieder aufwecken;

Bei Implementierung ist höchste Vorsicht geboten



Sender wartet bis Empfänger antwortet -> Synchr.