Verteilte und Parallele Programmierung

Mit Virtuellen Maschinen

PD Stefan Bosse

Universität Bremen - FB Mathematik und Informatik

Metriken und Verhalten von Parallelen Programmen

Wie können parallele und nebenläufige Prozesse quantitativ erfasst werden?

Was kann schief gehen?

Was begrenzt die Parallelisierung?

Zustands-Raum Diagramme

Für die Visualisierung und Analyse von nebenläufigen Prozessen und Aktionen

Zustands-Raum Diagramme beschreiben die möglichen Zustands-Entwicklungen - das temporale Verhalten - von parallelen Programmen.

- Computer sind endliche Zustandsautomaten. Der Zustandsübergang wird durch die Programminstruktionen hervorgerufen.
- Der Zustand **S** eines parallelen Programms welches aus *N* Prozessen P_i besteht setzt sich zusammen aus folgenden Tupeln:
  - Globale Variablen des Programms:
  - Lokale Variablen und Instruktionszeiger von Prozess 1:
  - Lokale Variablen und Instruktionszeiger von Prozess 2: usw.
- Die Berechnung ändert globale und lokale Variablen (Datenfluss) sowie Instruktionszeiger (Kontrollfluss) und führt zu einer Zustandsänderung *comp*: $s_i \rightarrow s_j$.

Nebenläufige Programmierung: Zustands-Raum Diagramme

- Wesentliche Zustandsänderung ist die Änderung von lokalen und globalen Speicher (Daten).
- Variablen sind Bestandteil von Ausdrücken und erlauben zwei Operationen: {read,write}

```
var x: read(x) ⇔ RHS value(x) → ε (value(x))
       write(x,v) ⇔ LHS reference(x) → x := v
```

Beschreibung eines parallelen Programms auf Programmierebene

- Der Zugriff auf globale und somit geteilte Variablen muss atomar sein, d.h. immer nur ein Prozess kann den Wert einer Variable lesen oder einen neuen Wert schreiben.
- Der parallele Zugriff auf eine geteilte Ressource muss aufgelöst werden (*Konflikt*): i.A. mutualer Ausschluss durch *Sequenzialisierung* der parallelen Zugriffe!

Nebenläufige Programmierung: Zustands-Raum Diagramme

- Beispiel

```
write(X,v1) || write(X,v2) || x3:=read(X) →
write(X,v1); read(X,x3); write(X,v2) |
x3=read(X);  write(X,v2); write(X,v2) | ..
```

- Algebraisch ausgedrückt ergibt sich die Transformation:

$$\frac{\begin{array}{c}(WRITE(x,v_1) \rightarrow p_1)\| \\ (WRITE(x,v_2) \rightarrow p_2)\end{array}}{\begin{array}{c}(WRITE(x,v_1) \rightarrow (WRITE(x,v_2) \rightarrow (p_1 \parallel p_2)))| \\ (WRITE(x,v_2) \rightarrow (WRITE(x,v_1) \rightarrow (p_1 \parallel p_2)))| \ldots\end{array}}$$

- Es gibt eine Menge von möglichen Entwicklungen des parallelen Systems!

Nebenläufige Programmierung: Zustands-Raum Diagramme

- Instruktionen von Prozessen werden sequenziell ausgeführt. Instruktionen können auf prozesslokale und programmglobale Variablen zugreifen.
- Definition eines parallelen Programms: globale Variablen (V) und Prozesse mit lokalen Variablen (v)

```
var V1,V2,...
process p1(a1,a2,...)
  var v1,v2,....


  Vi := ε(ai,vi,Vi)
  vi := ε(ai,vi,Vi)
end
```

```
process p2(a1,a2,...)
  var v1,v2,....


  Vi := ε(ai,vi,Vi)
  vi := ε(ai,vi,Vi)
end
```

Nebenläufige Programmierung: Zustands-Raum Diagramme

- Bei einem sequenziellen Programm ist das Ergebnis einer Berechnung (d.h. die Werte aller Variablen) deterministisch allein durch die Anweisungssequenz und die Eingabedaten gegeben, und kann beliebig oft wiederholt werden - immer mit dem gleichen Ergebnis → Reihenfolge aller Anweisungen der Berechnung vorgegeben und fest
- Bei einem parallelen Programm können mehrere Anweisungen verschiedener Prozesse gleichzeitig oder überlappend ausgeführt werden bzw. konkurrieren.
- Die Reihenfolge parallel ausgeführter Anweisungen von einzelnen Prozessen kann hingegen undeterministisch = zufällig sein!!
- Jeder der Prozesse kann als nächster den Zustand des Programms ändern.
- Ein Zustands-Raum Diagramm beschreibt die Änderung des Programmzustandes als sequenzielle Auswertung aller möglichen Prozessaktivitäten.

Nebenläufige Programmierung: Zustands-Raum Diagramme

- Das Diagramm ist ein gerichteter Graph, dessen Knoten den aktuellen Programmzustand $s \in \mathbf{S}$ beschreiben, und dessen Kanten die möglichen Zustandsübergänge beschreiben.
- Es gibt einen ausgewiesen Startzustand (Initialisierung) und einen oder mehrere Endzustände.
- Gibt es mehrere Endzustände liegt wohl möglich ein Entwurfsfehler vor, da das Programm unterschiedliche Endergebnisse liefern kann.

Algorithmus: Entwicklung des Zustands-Raum Diagramms

- Es gebe ein paralleles Programms welches aus N Prozessen $\mathbf{P}=\{p_1,p_2,..,p_N\}$ besteht.

1. Initialisiere den Startzustand $s_x=s_0$ und erzeuge Wurzel-Knoten s_0 im Diagramm.
2. Aktueller Zustand: s_x. Setze $\mathbf{P}^*:=\mathbf{P}$ mit $\mathbf{P}=\{p_1,p_2,..,p_N\}$ als Prozessliste und $\mathbf{S}^*=\{\}$
3. Wähle (entferne) einen beliebigen Prozess p_x aus der Prozessliste $\mathbf{P}^*$ und setze $\mathbf{P}^*:= \{p_n | \ p_n \in \mathbf{P}^* \wedge p_n \notin p_x\}$

Nebenläufige Programmierung: Zustands-Raum Diagramme

4. Evaluiere die nächste Instruktion $i_{x,n}$ von p_x und berechne die Wirkung auf lokale und globale Variablen sowie den Instruktionszeiger $I_{x,n}$.

5. Erzeuge einen neuen Zustandsknoten $s_{j/=x}$, füge ihn zur Liste $\mathbf{S}^*:=\mathbf{S}^* \cup \{s_j\}$ hinzu und verbinde ihn mittels einer Kante $t_{x\rightarrow j}$ mit dem aktuellen Ausgangszustand s_x

6. Wiederhole Schritt 3 bis 5 für alle anderen verbleibenden Prozesse $p \in \mathbf{P}$ bis $\mathbf{P}^*=\{\}$

7. Setze $\mathbf{S}^{**}:=\mathbf{S}^*$. Für jeden Knoten $s_x \in \mathbf{S}^{**}$ wiederhole die Schritte 2 bis 6. Iteriere Schritt 7 bis alle Prozesse terminiert sind oder keine Zustandsänderung mehr auftritt.

Nebenläufige Programmierung: Zustands-Raum Diagramme

Beispiel

Sequenzielles Programm

```
for i := 1 to 3 do i1
```

Paralleles Programm

```
process p1
  for i := 1 to 3 do i1
end
process p2
  for j := 1 to 3 do i2
end
```

Nebenläufige Programmierung: Zustands-Raum Diagramme

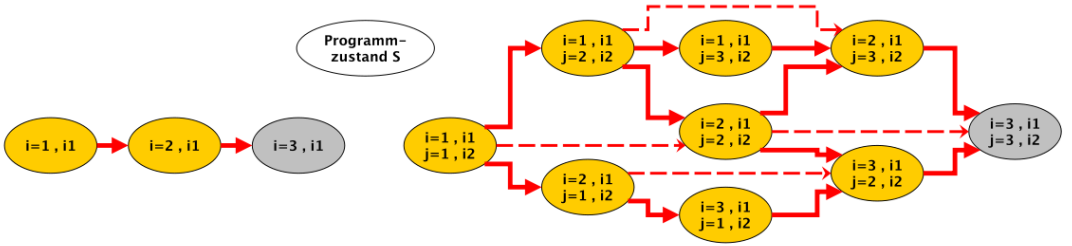


Abb. 1. Beispiel und mögliche Entwicklung des Programmzustandes (lokale Var. i,j)

Globale Ressourcen und Synchronisation

Globale Ressourcen

- In dem bisherigen Programmiermodell gibt es nur globale Variablen als globale geteilte Ressourcen, die konkurrierend von Prozessen gelesen und verändert werden können.
- Die globalen Variablen dienen dem Datenaustausch = Kommunikation zwischen den Prozessen (Shared Memory Model!).
  - Shared Memory dient aber nur dem Datenaustausch, es bietet keine höhere Synchronisation zwischen Prozessen
  - Bei nachrichtenbasierter Kommunikation (und auch Channels/Queues) ist das anders.

Globale Ressourcen und Synchronisation

Atomare Aktionen

- *Atomare Operationen*, Schreibweise $|A|$ werden in einem Schritt ausgeführt und können nicht durch andere Prozesse unterbrochen werden (keine Interferenz).
- Einzelne Lese- und Schreibe-Operationen mit globalen Variablen sind atomar. Nebenläufigkeit wird durch Sequenzialisierung aufgelöst.

```
t1: |X := v|
```

Globale Ressourcen und Synchronisation

Nicht atomare Aktionen und Synchronisation

Zusammengesetzte Ausdrücke können aus mehreren Berechnungsschritten bestehen, so ist z. B. `X := X + 1` als Ganzes nicht mehr atomar!

```
t1: |temp=X|
t2: |X:=temp+1|
⇔
    |X:=X+1| ??
```

Globale Ressourcen und Synchronisation

Beispiel einer unzureichend geschützten nicht atomaren Aktion

```
var X := 0
INCR(X)      DECR(X)
process p1     process p2
  var t:=0       var t:=0
  t := X         t := X
  X := t + 1     X := t - 1
end            end
```

Globale Ressourcen und Synchronisation

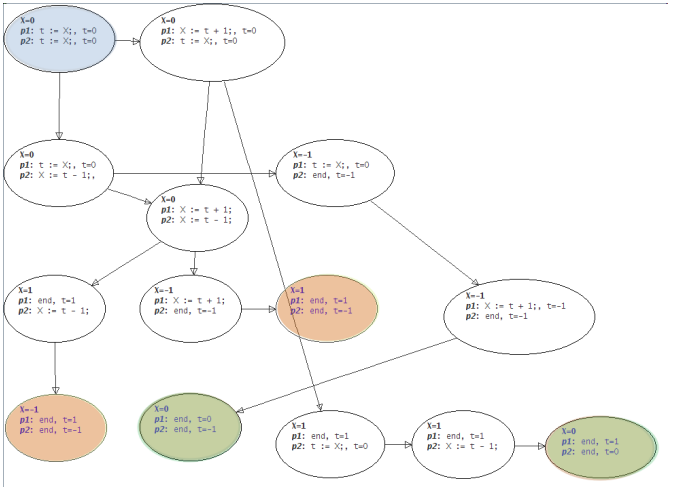


Abb. 2. Zustands-Raum Diagramm mit unterschiedlichen Endergebnissen (X=0,-1,1)

Experiment

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT Counter

Globale Ressourcen und Synchronisation

> Nicht atomare Aktionen (Sequenz von Aktionen) mit globalen Ressourcen müssen durch Mutuale Ausschlusssynchronisation geschützt werden → **Schutz von kritischen Kodebereichen ⇒ Datenkonsistenz!**

Semaphor

- Ein Semaphor S ist ein Zähler $s>0$ der niemals negative Werte annehmen kann.
- Es gibt zwei atomare Operationen die von einer Menge von Prozessen $P=\{p_1,p_2,...\}$ auf einem Semaphor S angewendet werden können:

Globale Ressourcen und Synchronisation

```
operation down(S) ∈ p:
  |if s > 0
   then s := s - 1
   else WAITERS+(S,p),block(p)|
operation up(S) ∈ p:
  |if s = 0 and WAITERS(S) ≠ []
   then pi=WAITERS-(S),unblock(pi)
   else s := s + 1|
```

Def. 1. Operationen der Semaphore

Globale Ressourcen und Synchronisation

- Ein Semaphor mit einem Startwert s=1 entspricht einer Mutex (binärer Semaphor). Ein *DOWN(S)* leitet einen kritischen Bereich in einer Sequenz i_1, i_2, i_3, ... ein, und ein *UP(S)* gibt ihn wieder frei:

```
semaphore S(1)
... down(S) |i1,i2,i3,..,in)| up(S) ...
```

Globale Ressourcen und Synchronisation

Beispiel einer mit einem Semaphor geschützten nicht atomaren Aktion

```
var X := 1
semaphore S := 1
INCR(X)          DECR(X)
process p1       process p2
  var t:=0         var t:=0
  down(S)          down(S)
  t := X           t := X
  X := t + 1       X := t - 1
  up(S)            up(S)
end              end
```

Globale Ressourcen und Synchronisation

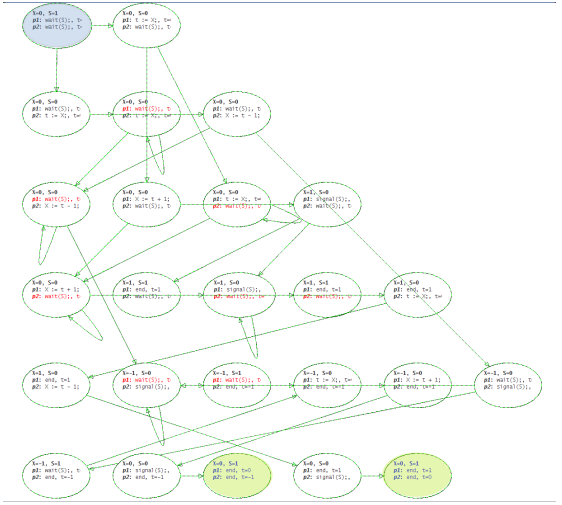


Abb. 3. Zustands-Raum Diagramm mit einem Endergebnis (X=0)

Eigenschaften von Parallelen Systemen

Konkurrieren mehrere Prozesse um die Ressource K, so gewinnt maximal einer, die anderen verlieren den Wettbewerb.

Def. 2. Wettbewerb

Ein solches **Wettbewerbsproblem** ist gekennzeichnet durch die Eigenschaften **Sicherheit** und **Lebendigkeit**.

Sicherheit bedeutet: Bedingung für Anzahl der Prozesse im kritischen Bereich K muss $|\{ p \mid I(p) \in K\}| \leq 1$ immer erfüllt sein.

Eigenschaften von Parallelen Systemen

Eigenschaft (Liveness) Lebendigkeit → Starvation Freiheit

- Starvation Freiheit bedeutet dass ein Prozess p_i der eine Ressource/den kritischen Bereich anfragt maximal eine endliche Anzahl von Zeiteinheiten / Wettbewerben $n \neq \infty$ durch jeden anderen Prozess umgangen werden kann (Bypass, die anderen haben den Wettbewerb gewonnen).

Eigenschaften von Parallelen Systemen

Eigenschaft Deadlock Freiheit

- Wenn bis zu einem beliebigen Zeitpunkt t eine Reihe von Prozessen versucht haben die Ressource / den kritischen Bereich zu erlangen, ihn aber nicht erhalten haben, so wird es in der Zukunft eine Zeit $t' > t$ geben wo sie erfolgreich sind (d.h. die Anfrage-Operation terminiert).
- D.h. eine Anfrage der Ressource terminiert mit einem gewonnen Wettbewerb garantiert irgendwann.
    - Starvation Freiheit impliziert Deadlock Freiheit!
    - Worst case eines Deadlocks: alle Prozesse verlieren den Wettbewerb, keiner kann die Ressource mehr nutzen/zugeteilt bekommen!

Eigenschaften von Parallelen Systemen

Eigenschaft Begrenztes (Bounded) Bypass Kriterium

- Es gibt eine Funktion $f(N)$ für N Prozesse die die maximale Anzahl von verlorenen Wettbewerben (bei Konkurrenz) angibt.

Benötigt First-In-First-Out Puffer Scheduler!

Eigenschaften von Parallelen Systemen

Service gegen Klient Sichtweise

Der Service ist die Ressource, der kritische Bereich. Der Klient ist der Prozess, der die Ressource anfragt.

- Aus Sicht des Service (Ressource) ist die Bewahrung von Invarianten (z.B. Semaphore immer positiv) wichtigstes Kriterium (Sicherheit) gefolgt von der Deadlock Freiheit. Der konkurrierende Wettbewerb führt immer dazu, dass irgendein Prozess die Ressource nutzen kann → *die Ressource gewinnt immer*.
- Aus Sicht des Klienten (eines Prozesses) ist die Starvation Freiheit wichtigstes Kriterium. Wenn immer ein Prozess die Ressource anfragt, wird er sie (eventuell) zugeteilt bekommen → *der Prozess gewinnt immer*.
- Aus der Sicht aller (um eine Ressource konkurrierenden) Prozesse sind die Freiheit von Deadlock und Starvation wichtigstes Kriterien.

Eigenschaften von Parallelen Systemen

- Es gibt daher eine Hierarchie der Lebendigkeits- und Sicherheitseigenschaften (entsprechend ihrer Stärke):

1. Bounded Bypass $f(N)=X \gg$
2. Starvation Freiheit $\equiv$ Endlicher Bypass $f(N) \neq \infty \gg$
3. Deadlock Freiheit $\gg$
4. Bewahrung von Invarianten (Konsistenz der Ressource)

Eigenschaften von Parallelen Systemen

Deadlock

- Tritt häufig ein wenn zwei (oder mehrere) Prozesse gegenseitig eine Ressource (Lock) beanspruchen die aber jeweils vom anderen bereits belegt ist.
  - Prozesse, die bereits Sperren halten, fordern neue Sperren an.
  - Die Anforderungen für neue Sperren werden gleichzeitig gestellt.
  - Zwei oder mehr Prozesse bilden eine kreisförmige Kette, in der jeder Prozesse auf eine Sperre wartet, die vom nächsten Prozess in der Kette gehalten wird → Dining Philosopher Problem.

Eigenschaften von Parallelen Systemen

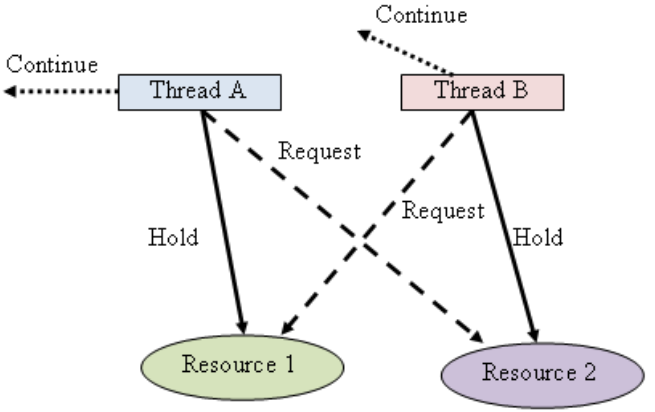


Abb. 4. Deadlocksituation zweier Prozesse die wechselseitig gesperrte Ressourcen anfordern

Experimentf

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT DiningPhil

Parallelität

- Unterteilung in räumliche und zeitliche Parallelität
- Parallele Datenverarbeitung bedeutet Partitionierung eines seriellen Programms in eine Vielzahl von Subprogrammen oder Tasks
- Weitere Unterteilung beider Dimensionen in Abhängigkeit von:
  - Art der Tasks/Algorithmen
  - Ausführungsmodell der Tasks und verwendete Rechnerarchitektur
  - Art und Umfang der Wechselwirkung zwischen Tasks
  - Kontroll- und Datenfluss eines Tasks

Parallelität

Räumliche Parallelität

Die Datenmenge D kann in Teilmengen $d_i \supset D$ zerlegt werden. Die minimal erreichbare Größe der Teilmenge gibt Granularität bei der Parallelisierung wieder. Die Datenmenge D wird durch eine Verarbeitungsstufe in eine neue Datenmenge D' transformiert, die dann von nachfolgenden Verarbeitungsstufen weiter verarbeitet wird.
Beispiel : D=Bild ⇒ Glättung ⇒ D_1 ⇒ Objekterkennung ⇒ D_2

Zeitliche Parallelität

Zeitliche Parallelität ist vorhanden, wenn eine Folge von gleichartigen Datenmengen $D(n)$ repitierend mit dem gleichen Algorithmus verarbeitet werden → **Pipeline-Verfahren**

Datenabhängigkeit

- Räumliche und zeitliche Parallelität führen zu räumlicher und zeitlicher Datenabhängigkeit.
- Räumliche Datenabhängigkeit findet auf Intra- und Intertaskebene statt.
  - Intrataskebene: Subtasks tauschen Daten aus → Sequenzielle Ausführung. Beispiel: ST_1: `a=x+y;` ST_2: `b=a+1;` → $b(a) \rightarrow ST_2(ST_1)$
  - Intertaskebene: Übertragung von Daten zwischen Tasks in einer Pipeline.
- Zeitliche Datenabhängigkeit: Ergebnisse aus der Vergangenheit gehen in aktuelle Datenberechnung ein. Beispiel: Bewegungserkennung aus einer Bild-Sequenz.

Datenabhängigkeit

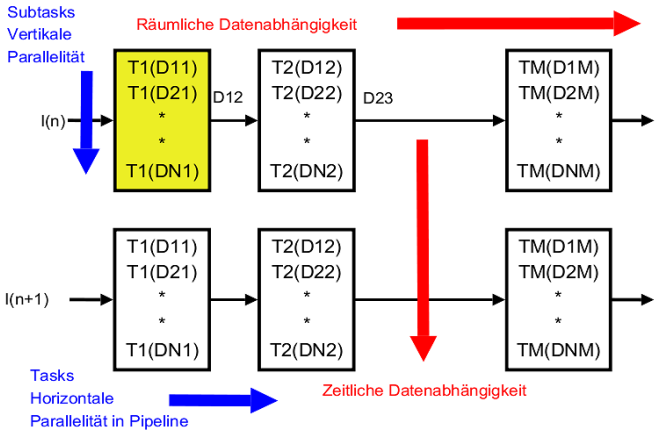


Abb. 5. Räumliche und zeitliche Datenabhängigkeit ⇔ Vertikale und horizontale Parallelisierung

Rechenzeit

- Die Rechenzeit t_{tot} für die Ausführung einer Pipeline $T_1 ... T_M$ enthält:

  1. Zeit zum Einlesen der Daten $I(n)$,
  2. Zeit für die Ausgabe der Daten und Ergebnisse,
  3. Summe aller Ausführungszeiten der Tasks in der Pipeline, die sich aus Rechen- und Kommuniaktionszeiten zusammensetzten.

$$t_{tot} = \sum_{i=1}^{m} \tau_i + \sum_{i=1}^{m-1} t_d(D_{i,i+1}) + t_{in} + t_{out}$$

$$\tau_i = \max\{t_{comp}(T_{i,j}(d_j)) | 1 \leqslant j \leqslant n_i\} + t_{comm}(T_i)$$

als die Zeit die benötigt wird, einen Task i unter Berücksichtigung von Datenabhängigkeiten der n_i Subtasks $T(d_j)$ zu bearbeiten.

Rechenzeit

- Längste Bearbeitungszeit eines Subtasks bestimmt Bearbeitungszeit des Tasks τ_i!
- T_i ist der i-te Task in der horizontalen Pipeline,
- d_j die Teildatenmenge eines Subtasks $T_{i,i}(d_j)$ eines Tasks T_i,
- t_{comp} ist die Rechenzeit,
- t_{comm} die Intertaskkommunikationszeit,
- t_{in} und t_{out} die Zeit zum Datentransfer in und aus der Pipeline, und
- $t_d(D_{i,i+1})$ die Datentransferzeit zwischen zwei Tasks.

- In Vision-Systemen sind die ersten Tasks i.A. low- und mid-level Algorithmen, und die letzten Tasks high-level Algorithmen, die auf den Daten der unteren Ebenen aufbauen. Die einzelnen Datenströme D_i können daher von unterschiedlicher Größe und Art sein.

Klassifikation von parallelen Algorithmen

Datenabhängigkeit

Lokal, statisch

Ausgangsdaten (≡Ergebnisse) hängen nur von eng begrenzter kurzreichweitiger Region der Eingangsdaten ab. Die Größe der Eingangsdatenregion ist unveränderlich (statisch). → Kommunikationsbedarf ist gering.

Lokal, dynamisch

Die Größe der Eingangsdaten-Region ist parametrisiert und veränderlich (dynamisch). Z.B. bei mathematischer Faltung von Bildmatrizen ändert sich Größe der Region.

Klassifikation von parallelen Algorithmen

Global, statisch

Die Ausgangsdaten hängen gänzlich vom gesamter Eingangsdatenmenge ab. Abhängigkeit hängt nur von der Größe des Bildes, aber nicht von dessen Inhalt ab. Z.B. Fouriertransformation oder Histogramm-Funktionen. → Kommunikationsbedarf ist groß.

Global, dynamisch

Ausgangsdaten hängen von variierenden Ausschnittsregion der Eingangsdaten ab. Die Berechnung ist vollständig datenabhängig. Z.B. Hough-Transformation.

Klassifikation von parallelen Algorithmen

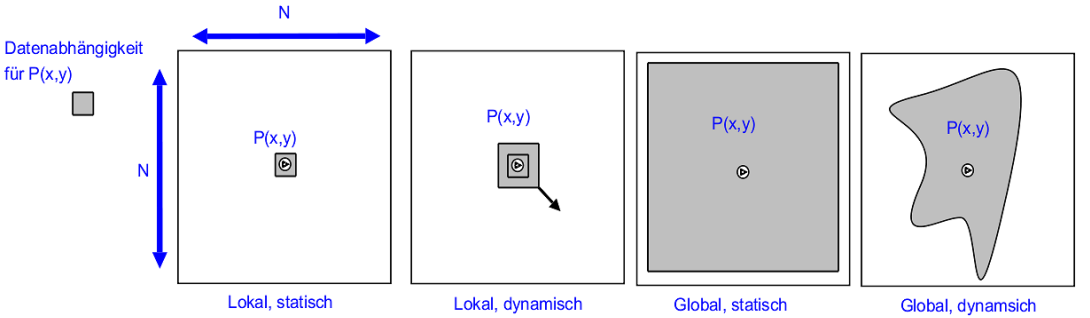


Abb. 6. Klassifikation nach Kommunikationsbedarf aufgrund Datenabhängigkeit zwischen einzelnen Tasks eines parallelen Programms

- Weiterhin Klassifikation nach:
  - Eingabedatenabhängigkeit
  - Ergebnisabhängigkeit

Speed-up

Bei der Datenverarbeitung gibt es drei Randbedingungen:

1. Gesamte Rechenzeit → Zeitdimension
2. Gesamte Ressourcenbelegung → Flächendimension
3. Bei der Parallelverarbeitung wird eine weitere Dimension hinzugefügt: Anzahl der Verarbeitungseinheiten *N*

- Die Nutzung von Parallelität führt zu einem Performanzgewinn (Speedup) durch Vergleich sequenzielles Programm (*N*=1) mit parallelen Programm (*N* Prozessoren):

$$Speedup = S(N) = \frac{Performanz(N)}{Performanz(1)}, S_t(n) = \frac{Rechnenzeit(1)}{Rechenzeit(N)}$$

Speed-up

- Die sog. Skalierung bei der Parallelisierung ist i.A. nicht linear:

$$0 < S(N) < N$$

- Kommunikation ist weitere Randbedingung bei der parallelen Datenverarbeitung.

Kosten und Laufzeit

Beispiel: Matrixmultiplikation

```
FUN matmult(A:ARRAY (p,q), B: ARRAY(p,q)) -> C:ARRAY (p,r)
DEF matmult(A, B) =
  FOR i = 1 to p DO
    FOR j = 1 to r DO
      C[i,j] <- 0;
      FOR k = 1 to q DO
       C[i,j] <- C[i,j] + A[i,k] * B[k,j]
      END FOR k
    END FOR j
  END FOR i
END
```

Kosten und Laufzeit

- Partitionierung kann beliebig erfolgen, da die einzelnen Ergebniswerte $C_{i,j}$ nicht voneinander abhängen.
- Datenabhängigkeit des Problems: Die Berechnung eines $C_{i,j}$ Wertes hängt außerhalb der FOR-k-Schleife von keinem anderen Wert $C_{n,m}$ mit $n \neq i$ und $m \neq j$ ab.

Kosten und Laufzeit

- Mögliche Partitionierungen der drei For-Schleifen auf *N* parallel arbeitenden Verarbeitungseinheiten (VE):

1. Jede VE berechnet einen $C_{i,j}$-Wert. D.h. eine VE führt die FOR-k-Schleife für ein gegebenes *i* und +j* durch.
   - Jede VE benötigt dazu die i-te Zeile von *A* und die j-te Spalte von *B*.
   - Keine weitere Datenabhängigkeit!
   - Es werden $N=p \bullet r$ VEs benötigt.

2. Eine VE berechnet eine Ergebnisspalte, d.h. führt die FOR-k und FOR-j-Schleifen durch.
   - Jede VE benötigt *A* und eine Spalte von *B*.
   - Es werden $N=p$ VEs benötigt.

Kosten und Laufzeit

3. Eine VE führt eine Multiplikation und Addition innerhalb der FOR-K-Schleife durch.
   - Jede VE benötigt einen *A* und *B*-Wert.
   - Es werden $N=p \bullet r \bullet q$ VEs benötigt.
   - Jeweils eine VE für einen gegebenen $C_{i,j}$-Wert führt die Zusammenführung der Zwischenwerte der FOR-k-VEs durch.

- Zusammenführung der Ergebnisdaten in den Fällen 1&2 trivial. Im Fall 3 besteht Zusammenführung im wesentlichen in der Summation der Zwischenergebnisse.

Kosten und Laufzeit

Beispiel: p=q=r=2

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1*5+2*7 & 1*6+2*8 \\ 3*5+4*7 & 3*6+4*8 \end{pmatrix}$$

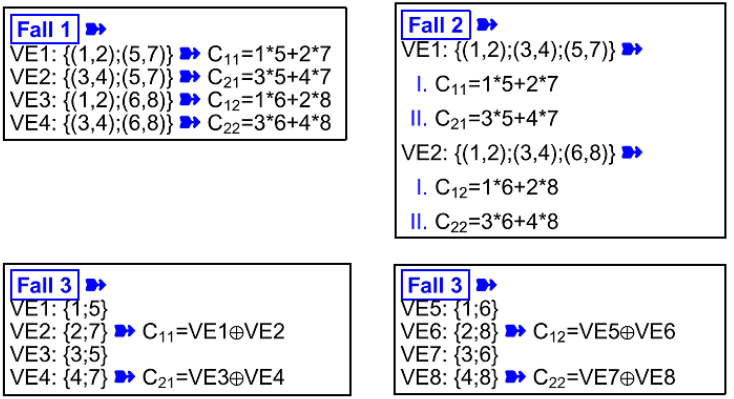


Abb. 7. Mögliche Partitionierungen der Matrixmultiplikation

Kosten und Laufzeit

Laufzeit- und Kostenanalyse:
Vereinfachung: n=p=q=r

Fall 0: Sequenzieller Algorithmus
- Anzahl VE: 1
- Laufzeit: $\Theta(n^3)$
- Kosten: $\Theta(n^3)$

Fall 1
- Anzahl VE: n^2
- Laufzeit: $\Theta(n)$
- Kosten: $\Theta(n^3)$

Fall 2
- Anzahl VE: n
- Laufzeit: $\Theta(n^2)$
- Kosten: $\Theta(n^3)$

Fall 3
- Anzahl VE: n^3
- Laufzeit: $\Theta(\log n)$
- Kosten: $\Theta(n^3 \log n)$

Fall 3b (Master=+Worker)
- Anzahl VE: $n^3/\log n$
- Laufzeit: $\Theta(\log n)$
- Kosten: $\Theta(n^3)$!!!

Berechnung der C-Summe im Fall 3

$$c_{i,j} = \sum_k t_{i,j,k}$$

wobei je eine VE einen t-Wert berechnet:

$$V_{i,j,k} \Rightarrow t_{i,j,k} \text{ mit Laufzeit } \Theta(1)$$

Implementierung der Summe:

1. Sequenziell mit einem Addierer
   ➥ Laufzeit: $\Theta(n)$
2. Kette aus (n-1) Addierern
   ➥ Laufzeit $\Theta(n)$
3. Baum-Kaskade aus (n-1) Addierern
   ➥ Laufzeit $\Theta(\log n)$

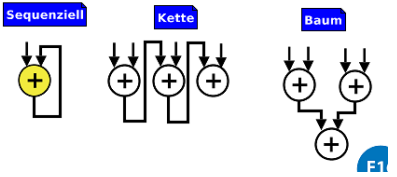


Abb. 8. Kosten und Laufzeitanalyse der verschiedenen Parititionierungen

Kommunikation

- Berechnung der Kommunikation mit Einheitswerten eines Nachrichtenaustauschs: Message Passing (MP) → Aufwand eine Zelle einer Matrix zu versenden ist *MP*=1.

Vereinfachung: n=p=q=r
Berechnung eines C-Wertes mit einer VE erfordert:

$$VE_{i,j} \Rightarrow c_{i,j} \Rightarrow MP = MP(\rightarrow A_i \oplus \rightarrow B_j) + MP(C_{i,j} \rightarrow) = 2n + 1$$

Summe aller Nachrichten für Berechnung von C mit n^2 VEs:

$$\sum_{i,j} VE_{i,j} \Rightarrow C \Rightarrow MP = n^2(2n + 1) = 2n^3 + n^2 \Rightarrow \Theta(n^3)$$

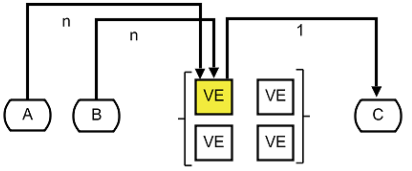


Abb. 9. Kommunikation Fall 1

Kommunikation

Vereinfachung: n=p=q=r
Berechnung einer C-Spalte mit einer VE erfordert:

$$VE_j \Rightarrow C_j \Rightarrow MP = MP(\rightarrow A_{i,j} \oplus \rightarrow B_j) + MP(C_{i,j} \rightarrow) = n^2 + n + n = n^2 + 2n$$

Summe aller Nachrichten für Berechnung von C mit n VEs:

$$\sum_j VE_j \Rightarrow C \Rightarrow MP = n(n^2 + 2n) = n^3 + 2n^2 \Rightarrow \Theta(n^3)$$

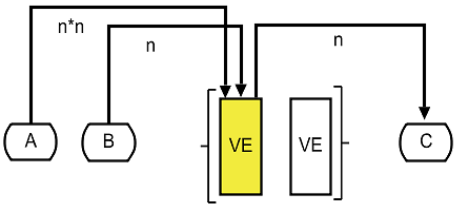


Abb. 10. Kommunikation Fall 2

Kommunikation

Vereinfachung: n=p=q=r
Berechnung eines t-Wertes mit einer VE erfordert:

$$VE_{i,j,k} \Rightarrow t_{i,j,k} \Rightarrow MP = MP(\rightarrow A_{i,j} \oplus \rightarrow B_{i,j}) + MP(t_{i,j,k} \rightarrow) = 3$$

Summe aller Nachrichten für Berechnung von C_{ij} mit n VEs:

$$\sum_{k} VE_{i,j,k} \Rightarrow C_{i,j} \Rightarrow MP = 3n$$

Summe aller Nachrichten für Berechnung von C mit n^3 VEs:

$$\sum_{i,j} VE_{i,j} \Rightarrow C \Rightarrow MP = n^2 3n = 3n^3 \Rightarrow \Theta(n^3)$$

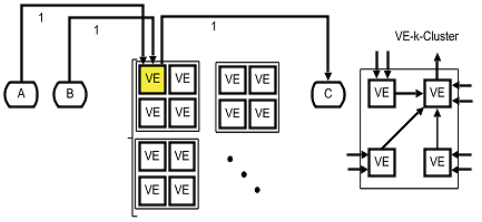


Abb. 11. Kommunikation Fall 3

Kommunikation

- Bisher konnten Matrizen ganzzahlig auf VEs verteilt werden. In der Realität ist aber $p=q=r,n$!
- Bisher wurde angenommen, dass alle VEs mit jeder anderen VE Daten mit einer Distanz/Ausdehnung $D=1$ austauschen kann. Nur möglich mit vollständig verbundenen Netzwerktopologien unter Verwendung von Kreuzschaltern.
- Gängige parallele und ökonomische Rechnertopologie: **Maschennetz**
  - Besteht aus $n \times n$ VEs.
  - Jede VE kann mit seinem direkten Nachbarn kommunizieren.
  - Eine Nachricht hat eine maximale Reichweite von $(2n-1)$ VEs, $\Theta(n)$.

Kommunikation

- Bisherige statische Partitionierung resultiert in zu hohen Kommunikationsaufwand in der Verteilung der Matrizen *A* und *B* sowie in der Zusammenführung der Matrix *C*.
  - Dynamisch veränderliche Partitionierung passt sich effizienter an Netzwerktopologie an.
  - Bei Matrixoperationen mit zwei Matrizen ($n=p=q=r$) ist dafür ein $2n \times 2n$ Netz optimal geeignet.

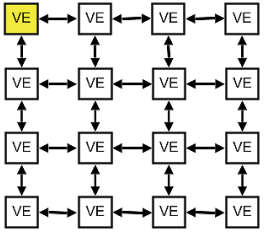


Abb. 12. Zweidimensionales Maschennetzwerk

Kommunikation

Dynamische Verteilung der Matrizen

- Das *2n x 2n* Netz wird in vier Quadranten der Größe $n \times n$ unterteilt.
  - Die Matrizen *A* und *B* befinden sich initial im linken unteren und rechten oberen Quadranten
  - Die Ergebnismatrix *C* wird schließlich im rechten unteren Quadranten zusammen- geführt.
- Alle VEs die ein $A_{1,j}$-Element besitzen werden dieses nach rechts verschieben.
- Alle VEs die ein $B_{i,1}$-Element besitzen werden dieses nach unten verschieben.
  - Dieser Vorgang wird für weitere Zeilen (*A*) und Spalten (*B*) fortgesetzt.
  - Die einzelnen Elemente von *A* und *B* passieren die VEs im C-Quadranten. Die einzelnen C-Werte können mittels Summation berechnet werden.

Kommunikation

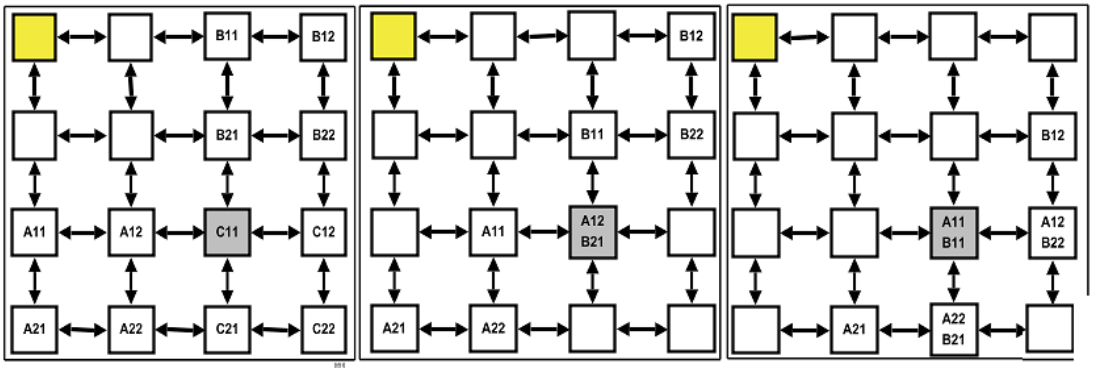


Abb. 13. Dynamische und überlagerte Verteilung der Matrizen *A*, *B*, und *C*

Kommunikation

- Die Laufzeit der Matrixmuliplikation auf dem Netz beträgt $\Theta(n)$, da n Verschiebungen durchgeführt werden.
- Die Kosten betragen $\Theta(n^3)$, vergleichbar mit sequenzieller Ausführung. D.h. diese Partitionierung und Architektur ist kostenoptimal.
- Das *2n x 2n* Netz wird nur in drei Quadranten genutzt. Benötigt werden daher nur $3n^2$ VEs.
- Aus Sicht der Rechnerarchitektur ungünstig zu implementieren und nicht generisch, d.h. abhängig vom verwendeten Algorithmus.

Kommunikation

- Reduktion dieses Verfahrens auf $n \times n$ Netz möglich
  - Dazu werden die Matrizen *A*, *B* und *C* überlagert, d.h. $VE_{1,1} \Rightarrow \{A_{1,1}, B_{1,1}, C_{1,1}\}$.
- Die Zeilenelemente von *A* werden dann nach vorherigen Ablaufschema gegen den Uhrzeigersinn rotiert (nur horizontal), und die Spaltenelemente von *B* im Uhrzeigersinn (vertikal).
- Man erhält gleiche asymptotische Grenzwerte für die Laufzeit $\Theta(n)$ und Kosten $\Theta(n^3)$!

Kommunikation

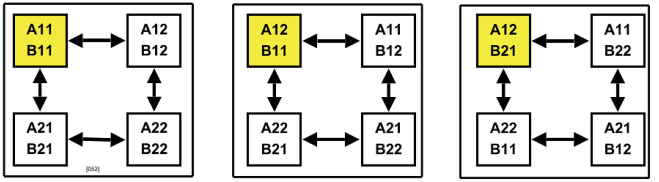


Abb. 14. Überlagerte Verarbeitung der Matrizen *A*, *B*, und *C* auf einem Maschennetzwerk

Maßzahlen für Parallele Systeme

Berechnungszeit

Die Berechnungszeit T_{comp} (computation time) eines Algorithmus als Zeit, die für Rechenoperationen verwendet wird.

Kommunikationszeit

Die Kommunikationszeit T_{comm} (communication time) eines Algorithmus als Zeit, die für den Daten- bzw. Nachrichtenaustausch (Sende- und Empfangsoperationen) zwischen Subprogrammen und Verarbeitungseinheiten verwendet wird.

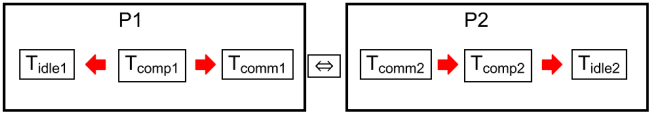
Untätigkeitszeit

Die Untätigkeitszeit T_{idle} (idle time) eines Systems als Zeit, die mit Warten (auf zu empfangende oder sendende Nachrichten) verbracht wird → Prozessblockierung trägt zur Untätigskeitszeit bei!

Maßzahlen für Parallele Systeme

Es gilt (bei *N* Prozessen):

$$T_{tot} = T_{comp} + T_{comm} + T_{idle} \approx 1/N \sum_{i=1..n} T_{comp,i} + T_{comm,i} + T_{idle,i}$$



Maßzahlen für Parallele Systeme

Übertragungszeit

Die Übertragunszeit T_{msg} (message time) ist die Zeit, die für das Übertragen einer Nachricht mit der Länge L Datenwörter zwischen zwei Prozessen oder Prozessoren benötigt wird.

- Sie setzt sich aus einer Startzeit T_s (message startup time) und einer Transferzeit T_w für ein Datenwort zusammen.
- Es gilt: $T_{msg} = T_s + L \bullet T_w$
- Voraussetzung: Verbindungsnetz arbeitet konfliktfrei.

Maßzahlen für Parallele Systeme

Startzeit

Die Startzeit wird durch die Kommunikationshard- und software bestimmt, die zur Initiierung eines Datentransfers benötigt wird, z.B. Overhead des Protokollstacks bei einer Software-Implementierung.

Transferzeit

Die Transferzeit wird durch die Bandbreite des Kommunikationsmediums und zusätzlich bei Software-Implementierung durch den Protokollstack (Datenkopie) bestimmt.

Maßzahlen für Parallele Systeme

Parallelisierungsgrad P

Die maximalze Anzahl von binären Stellen (bits) pro Zeiteinheit (Taktzyklus) die von einer Datenverarbeitungsanlage verarbeitet werden kann.

- Es gilt: $P = W \bullet B$

Wortlänge W

Die Wortlänge oder Wortbreite gibt die Anzahl der Bits eines Datenpfades an.

Bitslice-Länge B

Die Bitslice-Länge setzt sich zusammen aus der Anzahl von Verarbeitungseinheiten VE, die parallel ausgeführt werden können, und der Anzahl der Pipeline-Stufen einer VE.

- Es gilt: $B = N_{VE} \bullet N_{Stages}$

Maßzahlen für Parallele Systeme

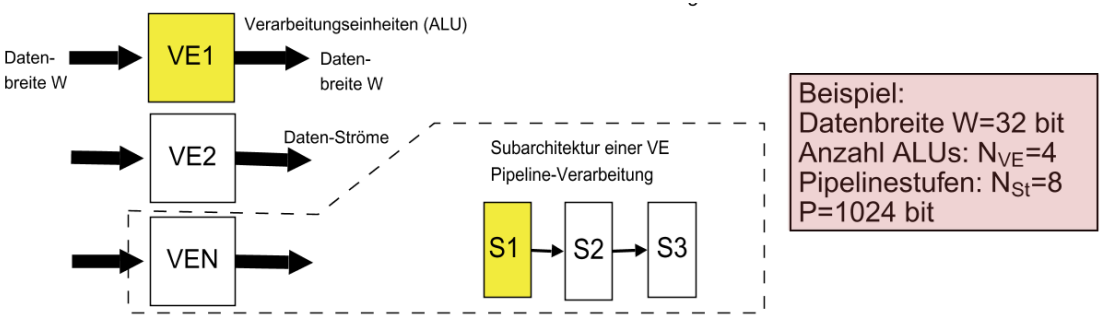


Abb. 15. Illustration Parallelisierungsgrad und Beispiel

Maßzahlen für Parallele Systeme

Beschleunigung S und Kosten C

Die Beschleunigung gibt die Steigerung der Verabeitungsgeschwindigkeit bzw. die Re-duzierung der Verarbeitungszeit T an beim Übergang Anzahl Prozessoren N=1 → N>1. Die Kosten C skaliert die Verarbeitungszeit (Komplexitätsklasse) mit den Ressourcen.

$$S(N) = \frac{T(1)}{T(N)}, C(N) = T(N)N$$

Maßzahlen für Parallele Systeme

Effizienz E

Die Effizienz gibt die relative Verbesserung in der Verarbeitungsgeschwindigkeit an, da die Leistungssteigerung *S* mit der Anzahl der Prozessoren normiert wird.

$$E(N) = \frac{S(N)}{N}, \text{ mit } \frac{1}{N} \leqslant E(N) \leqslant 1$$

Maßzahlen für Parallele Systeme

Mehraufwand R

Bezieht sich auf die Anzahl *X* der auszuführenden (Einheits-)Operationen des Programms:

$$R(N) = \frac{X(N)}{X(1)}$$

Parallelindex I

Der Parallelindex gibt die Anzahl der parallelen Operationen pro Zeit-/Takteinheit an.

$$I(N) = \frac{X(N)}{T(N)}$$

Maßzahlen für Parallele Systeme

Auslastung U

Entspricht dem normierten Parallelindex:

$$U(N) = \frac{I(N)}{N}$$

Maßzahlen für Parallele Systeme

Beispiel zur Auslastung

Ein Einprozessorsystem benötigt für die Ausführung von 1000 Operationen genau 1000 (Takt-)Schritte. Ein Multiprozessorsystem mit 4 Prozessoren benötigt dafür 1200 Operationen, die aber insgesamt in 400 Schritten ausgeführt werden können:

```
X(1)=1000 und T(1)=1000, X(4)=1200 und T(4)=400
S(4)=2.5 und E(4)=0.625,I(4)=3 und U(4)=0.75
```

Im Mittel sind 3 Prozessoren gleichzeitig aktiv, da jeder Prozessor nur zu 75% ausgelastet ist!

Amdahl's Gesetz

> Eine kleine Zahl von sequenziellen Operationen kann den Performanzgewinn durch Parallelisierung signifikant reduzieren.

- Sequenzieller Anteil der Berechnungszeit *T* eines Algorithmus in [%]: η
- Paralleler Anteil dann: $(1-\eta)$
- Kommunikation (Synchronisation) zwischen nebenläufig ausgeführten Tasks oder Verarbeitungseinheiten verursacht immer $\eta>0$!
  - Beispiel: Schutz einer geteilten Ressource mit einer Mutex.
  - Beispiel: Datenverteilung über eine Queue

Amdahl's Gesetz

- Zugriff auf geteilte Ressourcen verursacht immer $\eta>0$!
  - Beispiel: Geteilte Ressource Hauptspeicher in einer PRAM
- Der Kommunikationsanteil ist schwer im Voraus abzuschätzen, und der genaue Wert hängt auch von temporalen Konkurrenzverhalten ab (wie häufig gibt es verlorene Wettbewerbe)!

Das Amdahlsche gesetzt gilt nur für Probleme mit fester Größe!

Amdahl's Gesetz

- Es gilt dann für die gesamte Berechnungszeit eines parallelen Systems:

$$T(N,\eta) = \eta T(1) + \frac{(1-\eta)T(1)}{N}$$

- Daraus lässt sich eine Obergrenze der Beschleunigung *S* mit zusätzlicher Abhängigkeit von η ableiten:

$$S(N,\eta) = \frac{T(1)}{T(N,\eta)} \leqslant \frac{N}{(\eta N - 1) + 1}$$

Amdahl's Gesetz

Beispiel

Ein Algorithmus mit einem sequenziellen Anteil von η=10% und einem parallelen Teil von 90% wird A.) mit N=10 Prozessoren, und B.) mit N=20 Prozessoren ausgeführt. Die Obergrenze der Beschleunigung S ist dann:

```
A.) S(10)=5.26
B.) S(20)=6.0
```

- Verdopplung der Prozessoren erhöht nicht mehr signifikant die Beschleunigung!

Amdahl's Gesetz

(Berechnung)

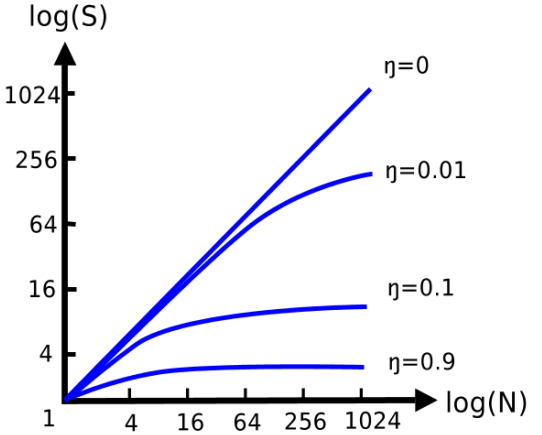


Abb. 16. Beschleunigung *S* in Abhängigkeit von η und Anzahl der Prozessoren *N*

Simulation eines einfachen Systems

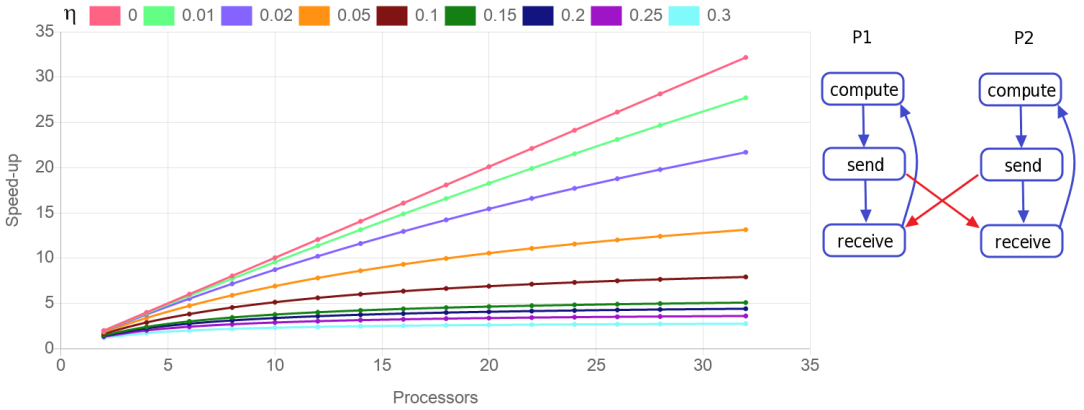


Abb. 17. Simulation der Beschleunigung *S* in Abhängigkeit des Kommunikationsanteils η% bei *N* Prozessen und einem vollständigen Kommunikationsnetzwerkgraphen *N*:*N*

Gustafson's Gesetz

- Amdahl's Gesetzt gilt bei Problemen mit fest vorgegebener Größe (also z.B. die Größe einer Matrix)
  - Für verteiltes Rechnen mit $N > 1000$ eine Katastrophe!
- Aber die theoretische Begrenzung der Beschleunigung S gilt so nicht mehr für Probleme die mitwachsen
  - Einfach ausgedrückt: Die Hinzunahme von Prozessoren bei zunehmender Größe des Problems bringt i.A. einen Vorteil!
- Es kann (min.) zwei Motivationen geben:
  - Die Rechenzeit soll effizient und wirtschaftlich reduziert werden
  - Die Rechenzeit soll beim Anwachsen des Problems (z.B. die Anzahl von WEB Seiten die eine Suchmaschine indizieren soll) nicht anwachsen (egal was es kostet, oder noch mit vertretbaren Kostenaufwand)!

Gustafson's Gesetz

Geschichte

In einer Konferenzdebatte von 1967 über die Verdienste von Parallelen Rechnen, IBM's Gene Amdahl argumentiert, dass ein wesentlicher Anteil der Arbeit von Computern inhärent seriell ist, sowohl aus algorithmischen als auch aus architektonischen Gründen. [C]

Er schätzte die serielle Fraktion f auf etwa 0.25-0.45. Er behauptete, dass dies den Ansatz stark einschränken würde parallele Verarbeitung zur Reduzierung der Ausführungszeit einzusetzen.

Amdahl argumentierte, dass selbst die Verwendung von zwei Prozessoren weniger kosteneffektiv als ein serieller Prozessor. Außerdem, die Verwendung einer großen Anzahl von Prozessoren würde niemals die Ausführungszeit um mehr als $1/f$ reduzieren, was durch seine Schätzung ein Faktor von etwa 2-4 war.

Trotz vieler Bemühungen, einen Fehler in Amdahls Gesetz zu finden, ist Amdahls Gesetz seit über 20 Jahren die Begründung für die fortgesetzte Verwendung von seriellem Rechnern und seriellen Programmiermodellen.

Gustafson's Gesetz

1. Die Zeit die ein Nutzer auf eine Berechnung warten will sei konstant (Einheitswert, *P*=1)
2. Der Anteil der Berechnung der seriell ist sei *f*, und unabhängig von der Parallelisierung (Stimmt das wirklich??)
   - Der serielle Anteil kann mit zunehmender Problemgröße (relativ!) abnehmen!
3. Der verbleibende Anteil 1-*f* parallelisiert ideal so dass ein serieller Prozessor *P* mal länger braucht um diesen zu berechnen (*P*: Anzahl der Prozessoren)
4. Das Verhältnis der parallelen zur seriellen Berechnung ist dann:

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

Gustafson's Gesetz

[C]

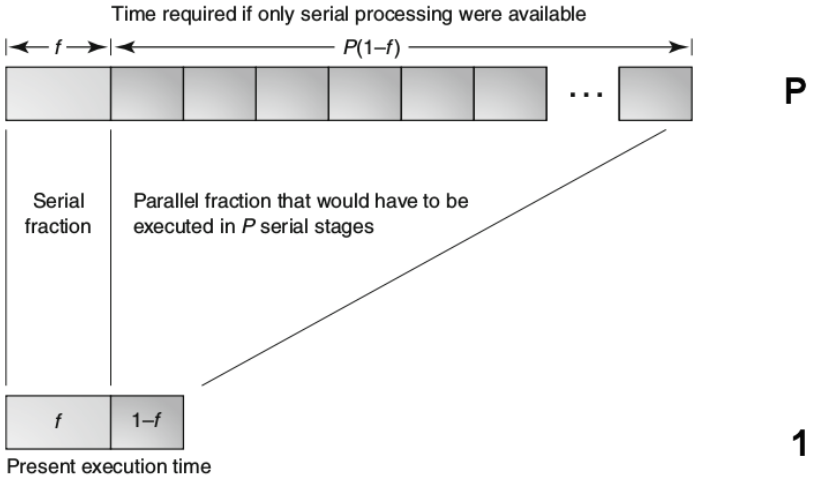


Abb. 18. Vergleich paralleler zu serieller Berechnung

Gustafson's Gesetz

[C]

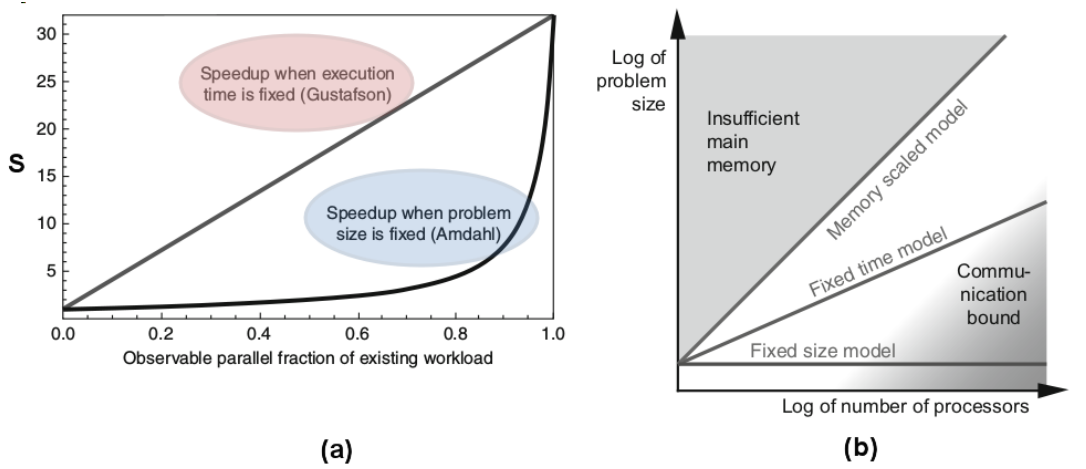


Abb. 19. (a) Beschleunigung *S* möglich mit 32 Prozessoren in Abhängigkeit vom parallelen Anteil, Gustafson's vs Amdahl's Gesetz (b) Verschiedene Skalierungstypen und Kommunikationskosten sowie Randbedingungen[]

Zusammenfassung

Parallele Programme können sich auf unterschiedliche Weise entwickeln. Zustandsraum Diagramme verdeutlichen die möglichen Entwicklungen von parallelen Systemen.

Es gibt eine Vielzahl von metrischen Größen um parallele und verteilte Systeme messbar und vergleichbar zu machen → Beschleunigung.

Es können Fehler in parallelen Systemen durch Wettrennen und Wettbewerbe auftreten: Deadlock, Starvation

Kommunikation ist der limitierende Faktor in der Parallelisierung.