4.4 Schnelle Fourier-Transformation

$f : \mathbb{R} \rightarrow \mathbb{R}$ ist eine periodische Funktion, falls

$\exists \tau$ so dass $f(t + \tau) = f(t)$ für alle $t \in \mathbb{R}$

Minimales τ, welches die Gleichung erfüllt ist die Periode und $1/\tau$ die Frequenz

Fourier-Transformation ist ein Werkzeug zur Analyse solcher Systeme

Formal Überführung einer Funktion $f(t)$ in eine komplexwertige Funktion $F(\omega)$ im Frequenzbereich

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{i \cdot 2\pi\omega t} dt$$

sowie die inverse Fourier-Transformation

$$f(t) = \int_{-\infty}^{\infty} F(\omega) e^{-i \cdot 2\pi\omega t} d\omega$$

$f(t)$ und $F(\omega)$ beschreiben das selbe Signal, aber

- $f(t)$ ist die "natürlichere" Darstellung, in der das Signal gemessen wird
- $F(\omega)$ lässt sich leichter manipulieren (Filterung, Glättung, Faltung, ...)
- durch Hin- und Rücktransformation sind die Vorteile beider Darstellungen nutzbar

Beispiel: Faltung $(f * g)(t) \equiv \displaystyle\int_{-\infty}^{\infty} g(\tau) f(t - \tau) d\tau$

Berechnung unangenehm, aber es gilt

$(f * g)(t) \Leftrightarrow F(\omega) \cdot H(\omega)$ Berechnung einfach

Anwendungen

- Bildverarbeitung: Kontrastverbesserung, Ausgleich bekannter Verzerrung (z.B. Hubble-Teleskop)
- Spracherkennung: Personenspezifische Zuordnungen sind im Frequenzbereich einfacher realisierbar
- Berechnung digitaler Filter
- Diskrete Fourier-Transformation zur schnellen Multiplikation von Polynomen

Historie:

- Euler (1707-1783), Clairaut (1713-1765), Lagrange (1736-1813), Gauß (1777-1855) entwickelten diskrete Fourier-Transformationen für bestimmte Reihen zur Untersuchung der Bewegung von Himmelskörpern
- Fourier (1768-1830) ist Namenpatron für das gesamte Gebiet und leistete wesentliche Beiträge zur Theorie
- Cooley und Tukey entwickelten 1965 den Algorithmus zur schnellen Fourier-Transformation (FFT)

In technischen Anwendungen ist $f(t)$ oftmals nicht bekannt

Stattdessen werden die Werte von $f(t)$ alle Δ Zeiteinheiten gemessen

$1/\Delta$ nennt man die Abtastrate (sampling rate)

$\omega_c = 1/(2\Delta)$ die Nyquist-Frequenz

Falls $F(\omega) = 0$ für $|\omega| \geq \omega_c$, dann gilt

$$f(t) = \Delta \sum_{k=-\infty}^{\infty} f_k \frac{sin(2\pi\omega_c(t - k\Delta))}{\pi(t - k\Delta)}$$

wobei $f_k = f(k\Delta)$

d.h. wenn die Abtastfrequenz groß genug ist, so kann die kontinuierliche Funktion durch eine diskrete Funktion repräsentiert werden

Für die praktische Berechnung betrachten wir nur endlich viele Summanden

Wie benutzen N Summanden und definieren:

$f_k = f(t_k) = f(k\Delta)$ für $k = 0, 1, 2, \ldots, N - 1$

Sei N gerade und $\omega_j = \frac{j}{N\Delta}$ für $j = -\frac{N}{2}, \ldots, \frac{N}{2}$

Damit hätten wir

$$\begin{aligned} F(\omega_j) = \int_{-\infty}^{\infty} f(t) e^{i \cdot 2\pi\omega_j t} dt \quad &\approx \sum_{k=0}^{N-1} f_k e^{2\pi i \omega_j t_k} \Delta \\ &= \Delta \sum_{k=0}^{N-1} f_k e^{2\pi i k j / N} \end{aligned}$$

$F_j = \displaystyle\sum_{k=0}^{N-1} f_k e^{i \cdot 2\pi k j / N}$ ist die diskrete Fourier-Transformation

es gilt $F(\omega_j) \approx \Delta F_j$

Analog lässt sich die diskrete inverse Fourier-Transformation entwickeln

$f_k = \frac{1}{N} \displaystyle\sum_{j=1}^{N-1} F_j e^{-i \cdot 2\pi k j / N}$

Sei $w = e^{i \cdot 2\pi / N}$ (komplexe Zahl)

Es gilt dann $F_j = \displaystyle\sum_{k=0}^{N-1} w^{kj} f_k$

Dies kann als ein Vektor-Matrix-Produkt aufgeschrieben werden

$f = (f_0, \ldots, f_{N-1})^T$, $F = (F_0, \ldots, F_{N-1})^T$

$$T = \begin{pmatrix} 1 & 1 & \cdots & \cdots & \cdots & 1 \\ 1 & w & w^2 & \cdots & \cdots & w^{N-1} \\ \vdots & w^2 & w^4 & \cdots & \cdots & w^{2(N-1)} \\ \vdots & \vdots & \vdots & \cdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \cdots & \cdots & \vdots \\ 1 & w^{N-1} & w^{2(N-1)} & \cdots & \cdots & w^{(N-1)(N-1)} \end{pmatrix}$$

Damit gilt $F = Tf$

wobei T eine symmetrische Matrix ist

Aufwand der Vektor-Matrix-Multiplikation $O(N^2)$

Geht es schneller?

Zur effizienteren Berechnung müssen spezielle Eigenschaften der Potenzen von w genutzt werden

Es gilt: $e^{i\varphi} = \cos(\varphi) + i \cdot \sin(\varphi)$

$\sin(\varphi) = \sin(\varphi + 2\pi) = -\sin(\varphi + \pi) = -\cos(\varphi + \pi/2)$

$\sin(\pi) = 0$ und $\sin(\pi/2) = -1$

Damit können folgende Identitäten bewiesen werden:

$$w^j = w^{j \ mod \ N} \qquad falls\ j \in \mathbb{N}$$
$$w^{N/2+j} = -w^j \qquad falls\ 0 \le j \le N/2 - 1$$
$$w^{N/2} = -1 \quad w^N = 1$$

Eingesetzt ergibt sich für die Matrix T(hier für $N = 8$):

$$T = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w^2 & -1 & -w^2 & 1 & w^2 & -1 & -w^2 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -w^2 & -1 & w^2 & 1 & -w^2 & -1 & w^2 \\ 1 & w & w^2 & w^3 & -1 & -w & -w^2 & -w^3 \\ 1 & w^3 & -w^2 & w & -1 & -w^3 & w^2 & -w \\ 1 & -w & w^2 & -w^3 & -1 & w & -w^2 & w^3 \\ 1 & -w^3 & -w^2 & -w & -1 & w^3 & w^2 & w \end{pmatrix}$$

Idee der Fast Fourier-Transformation (FFT):

Berechne die Fourier-Transformation der Länge N als Summe von zwei Fourier-Transformationen der Länge $N/2$:

$$\begin{aligned} F_j &= \sum_{k=0}^{N-1} w^{kj} f_k \\ &= \sum_{k=0}^{N/2-1} w^{2kj} f_{2k} + \sum_{k=0}^{N/2-1} w^{2kj+j} f_{2k+1} \\ &= \sum_{k=0}^{N/2-1} w^{2kj} f_{2k} + w^j \sum_{k=0}^{N/2-1} w^{2kj} f_{2k+1} \end{aligned}$$

$w^{2kj} = e^{i \cdot 4kj\pi/N}$ für Länge N und

$e^{i \cdot 4kj\pi/N} = e^{i \cdot 2kj\pi/(N/2)} = w^{kj}$ für Länge $N/2$

Damit gilt für $j = 0, \ldots, N/2 - 1$

$$\begin{aligned} F_j &= \sum_{k=0}^{N/2-1} w^{kj} f_{2k} + w^j \sum_{k=0}^{N/2-1} w_N^{kj} f_{2k+1} \\ &= F_n^g + w_N^j F_j^u \end{aligned}$$

und $F_{N/2+j} = F_j^g + w_N^j F_j^u$

wobei

- $w_N^j = \left(e^{i \cdot 2\pi/N}\right)^j$,
- F_j^g (F_j^u) $j$te Komponente einer Fourier-Transformation der Länge $N/2$, die nur die Werte mit geradem (ungeradem) Index berücksichtigt

Sei $N = 2^n$ dann kann die Idee rekursiv angewendet werden

Wir benutzen im Folgenden 0 für g und 1 für u und teilen Subsequenzen rekursiv auf

d.h. F_j^{00} (für $0 \le j \le N/4 - 1$) beschreibt den $j$ten Wert der Fourier-Tranformation der Länge $N/4$, der aus Werten mit Indizes $0, 4, 8, 12, \ldots$ (zweimal geradem Index) der ursprünglichen Werte entsteht

Jeder Index $k \in 0, \ldots, N-1 = 2^n - 1$ kann als Bitstring $k = (k_1, \ldots, k_n)$ beschrieben werden, so dass $k = \sum_{j=1}^n k_j \cdot 2^j$

Sei b ein Bitstring mit Länge $|b|$ und $b = \epsilon$ der leere Bitstring mit $|\epsilon| = 0$

Die zu einem Bitstring gehörige Nummer ist $\#b = \sum_{j=0}^{|b|-1} b_j \cdot 2^j$

und $b0 = b_0 \ldots b_{|b|-1}0$

Wir definieren rekursiv $F_0^b = f_{\#b}$ falls $|b| = n$:

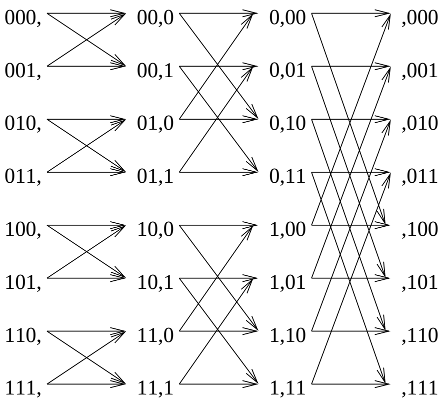
und für $|b| < n$:

$$F_j^b = \begin{cases} F_j^{b0} + w_{|b|}^j F_j^{b1} & falls\ j < 2^{n-|b|-1} \\ F_{j-2^{n-|b|-1}}^{b0} + w_{|b|}^{j-2^{n-|b|-1}} F_{j-2^{n-|b|-1}}^{b1} & sonst \end{cases}$$

Es gilt dann $F_j = F_j^\epsilon$ für $0 \le j \le N/2 - 1$

Beispiel für $N = 8$

Schreibweise: $b = b_1, b_2 \Rightarrow F_{\#b_2}^{b_1}$



Ein mögliches Schema, Umkehrung der Pfeile liefert ein anderes (verbreiteteres Schema, perfect shuffle)

Aufwandsbetrachtung:

n Stufen pro Stufe N Schritte $\Rightarrow$

Aufwand $O(Nn) = O(N\ log\ N)$

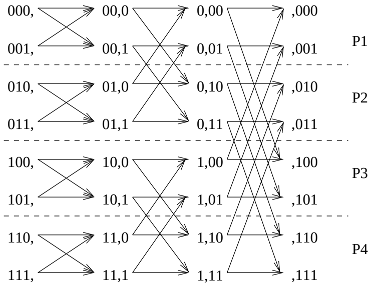
Falls N keine Potenz von 2:

- Daten mit Nullen auffüllen, so dass eine Potenz von 2 entsteht oder
- N in Primfaktoren zerlegen und Algorithmus verwenden, der auf Verallgemeinerung de Idee beruht

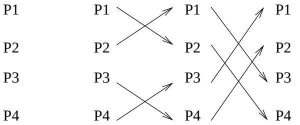
Parallele Implementierung: Verteilung von $N = 2^n$ Werten auf $P = 2^p$ Prozessoren $(p < n)$

Jeder Prozessor erhält 2^{n-p}Werte

Aufteilung $N = 8$ auf $P = 4$ Prozessoren



Kommunikationsstruktur



Keine Kommunikation im ersten Schritt, danach jeweils eine Kommunikationsoperation

(= Versenden eines Vektors mit 2^{n-p}Werten pro Prozessor)

Arithmetische Operationen pro Schritt:

- 2^{n-p}komplexe Additionen (2^{n+1-p} reelle Additionen)
- bis zu 2^{n-p}komplexe Multiplikationen (bis zu 2^{n+2-p} reelle Multiplikationen und 2^{n+1-p} reelle Additionen)

4.5 Optimierungsprobleme

Optimierungsprobleme sind neben Gleichungssystemen die wichtigste Problemklasse in der angewandten Mathematik

Optimierungsprobleme existieren in zahlreichen Varianten

Hier werden nur zwei Probleme der *dynamischen Programmierung* betrachtet, für deren Lösung einfache parallele Algorithmen existieren

Problem der kürzesten Pfade in einem Graphen (V, E)

Sei $V = \{0, \ldots, n-1\}$ die Menge der Knoten und

E die Menge der gerichteten Kanten $(E \subset V \times V)$

Jede Kante $(i, j) \in V$ ist mit Länge a_{ij} bewertet

$(a_{ij} \in \mathbb{R}_{\geq 0} \cup \infty)$

- falls $(i, j) \notin E \Rightarrow a_{ij} = \infty$
- falls $(i, j) \in E \Rightarrow a_{ij} < \infty$

Ein Weg $(i_1, i_2, \ldots, i_k)$ ist eine Sequenz von Knoten mit $i_l \in E$ und $(i_{l-1}, i_l) \in E$ $(0 < l \leq k)$

Die Länge eines Weges $(i_0, i_2, \ldots, i_k)$ ist

definiert als $\sum_{l=1}^{k} a_{i_{l-1} i_l}$

Gesucht wird der kürzeste Weg

von jedem Knoten i zu Knoten 0

Sei $A(i) = \{j | a_{ij} < \infty\}$

(Menge der Knoten, zu denen eine Kante von i aus geht)

Voraussetzungen

- Knoten 0 habe keine abgehenden Kanten $(A(0) = \emptyset)$
- jeder Zyklus im Graphen habe eine positive Länge
- von jedem Knoten $i \in V$ existiert ein Weg zu Knoten 0

Sei x_i^* die Länge des kürzesten Weges von i nach 0

x^* ist die eindeutige Lösung von

$$\begin{array}{rcl} x_i^* & = & \min_{j \in A(i)}(a_{ij} + x_j^*) \qquad (i = 2, \ldots n) \\ x_0^* & = & 0 \end{array}$$

Bellman-Ford-Algorithmus

- initialisiere $x_0^{(0)} = 0$ und $x_i^{(0)}$ $(i = 1, \ldots, n-1)$ beliebig
- iteriere $x_i^{(k)} = \min_{j \in A(i)}(a_{ij} + x_j^{(k-1)})$
- bis $x_i^{(k)} = x_i^{(k-1)}$ für alle $i = 1, \ldots, n-1$

Algorithmus konvergiert nach endlicher Iterationszahl

m^* (= maximale Kantenzahl eines kürzesten Weges), falls

$x_i^{(0)} \geq x_i^*$ für alle $i = 1, \ldots, n-1$

Konvergenz auch für andere Startwerte, dann aber u.U. nach zusätzlichen Iterationen

Synchrone parallele Implementierung des

Bellman-Ford Algorithmus

identisch zur Vektor-Matrix Multiplikation nach Ersetzung der Addition durch das Minimum und der Multiplikation durch die Addition

$\sum_{j=0}^{n-1} A_{ij} x_j \Rightarrow \min_{j=0}^{n-1}(a_{ij} + x_j)$

$\Rightarrow$ parallele Implementierung wie Vektor-Matrix-Multiplikation

(dicht- oder spärlich-besetzte Matrix)

Andere Algorithmen für das Problem des kürzesten Weges

Falls kürzeste Wege zwischen allen Paaren von Knoten berechnet werden sollen, muss der Bellman-Ford-Algorithmus n-mal aufgerufen werden

Laufzeit $O(n \cdot m^* * nz(A))$ mit

$nz(A) =$ Kantenzahl im Graphen

$\Rightarrow$ effizientere Algorithmen für dieses Problem

- Floyd-Warshall-Algorithmus
- Doubling-Algorithmus

$\Rightarrow$ Ergebnisse die einmal berechnet wurden werden

gespeichert und weiter verwendet

Laufzeit beider Algorithmen unabhängig von Kantenzahl im Graphen $\Rightarrow$ Laufzeit $O(n^3)$ bzw. $O(n^3 \cdot \log m^*)$

Floyd-Warshall-Algorithmus

Initiale Belegung

$$x_{ij}^{(0)} = \begin{cases} a_{ij} & falls\ j \in A(i) \\ \infty & sonst \end{cases}$$

Iterationsschema

$$x_{ij}^{(k+1)} = \begin{cases} \min\left(x_{ij}^{(k)}, x_{ik}^{(k)} + x_{kj}^{(k)}\right) & falls\ i \neq j \\ \infty & sonst \end{cases}$$

$x_{ij}^{(k)}$ enthält die Länge des kürzesten Wegs von i nach j über Zwischenknoten aus $\{0, \ldots, k\}$

Parallele Implementierung (für n Prozessoren)

Ablauf für Prozessor p

```
double x[n], y[n] ;
/* initialisiere x[i] = a[p][i] */
for (i = 0; i < n; i++) {
   if (p == i)
    MPI_Bcast(x,n,MPI_DOUBLE, i, MPI_COMM_WORLD);
   else {
    MPI_Bcast(y,n,MPI_DOUBLE, i, MPI_COMM_WORLD);
    for (j = 0; j < n; j++)
      if (i != j && x[j] > x[i] + y[j])
        x[j] = x[i] + y[j] ;
 } }
```


Doubling-Algorithmus

Initiale Belegung

$$x_{ij}^{(0)} = \begin{cases} a_{ij} & falls\ j \in A(i) \\ 0 & falls\ i = j \\ \infty & sonst \end{cases}$$

Iterationsschema

$$x_{ij}^{(k+1)} = \begin{cases} \min_m \left(x_{im}^{(k)} + x_{mj}^{(k)}\right) & falls\ i \neq j \\ 0 & sonst \end{cases}$$

$x_{ij}^{(k)}$ enthält die Länge des kürzesten Weges von i nach j mit 2^{k-1} oder weniger Kanten

$\Rightarrow$ nach maximal $\lceil \log(n) \rceil$ Schritten ist der Algorithmus beendet (praktisch werden wie beim Floyd-Warshall-Algorithmus m^* Schritte benötigt)

Sequentieller Aufwand höher als beim Floyd-Warshall-Algorithmus

Parallelisierung:

wie Matrix-Matrix-Multiplikation (+ durch $\min$ und * durch + ersetzen)

$\Rightarrow$ Realisierung in $O((\log n)(\log m^*))$ auf Hyperwürfel mit $O(n^3)$ Prozessoren

$\Rightarrow$ schnellste bekannte Lösung des Problem der kürzesten Wege zwischen allen Knoten!

Komplexere Probleme der dynamischen Programmierung

Eigenschaften des Problems der kürzesten Wege

- Kostenfunktion lässt sich durch einen Term beschreiben (monadische Kostenfunktion)
- Problem kann in Subprobleme zerlegt werden und Lösung aus Sublösungen zusammengesetzt werden (serielles Problem)

Algorithmen für diese Art von Problemen lassen sich auf Vektor-Matrix-Multiplikationen zurückführen

Nun betrachten wir ein polyadisches nicht-serielles Problem

- Kostenfunktion wird durch rekursive Terme beeinflusst (polyadisch)
- Lösung kann nicht aus Sublösungen zusammengesetzt werden (nicht-seriell)

Beispielproblem optimale Klammerung

- Gegeben n Matrizen $A^{(1)}, \ldots, A^{(n)}$
- Matrix $A^{(i)}$ mit r_{i-1} Zeilen und r_i Spalten
- Berechnet werden soll das Produkt $\prod_{i=1}^{n} A^{(i)}$

Matrix-Matrix-Multiplikation ist assoziativ $\Rightarrow$ es kann unterschiedlich geklammert werden

Gesucht eine Klammerung mit minmaler Zahl von Operationen

Beispiel: Multiplikation der Matrizen

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{pmatrix} \quad \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \end{pmatrix}$$

$$\begin{pmatrix} C_{11} \\ C_{21} \\ C_{31} \end{pmatrix} \quad \begin{pmatrix} D_{11} & D_{12} \end{pmatrix}$$

$$\begin{pmatrix} E_{11} & E_{12} \\ E_{21} & E_{22} \end{pmatrix} \quad \begin{pmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \end{pmatrix}$$

Multiplikation von links nach rechts:
84 Skalar-Multiplikationen und 44 Skalar-Additionen

Multiplikation von rechts nach links:
69 Skalar-Multiplikationen und 33 Skalar-Additionen

Falls B, C, F Dimension 300 statt 3 besitzen würden $\Rightarrow$

Multiplikation von links nach rechts:
6024 Skalar-Multiplikationen

Multiplikation von rechts nach links:
274200 Skalar-Multiplikationen

Zahl der möglichen Klammerungen $\approx 4^n/(n \cdot \sqrt{\pi n})$

Algorithmus der dynamischen Programmierung mit Aufwand $O(n^3)$ für n Matrizen

Sei c_{ij} die minimale Anzahl von Skalar-Multiplikationen zur Berechnung von $A^{(i)} \cdot \ldots \cdot A^{(j)}$ es gilt

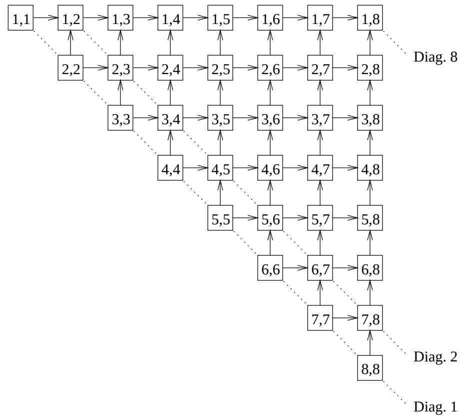
$$c_{ij} = \begin{cases} \min_{i \leq k \leq j}(c_{ik} + c_{(k+1)j} + r_{i-1} r_k r_j & falls\ 1 \leq i < j \leq n \\ 0 & sonst \end{cases}$$

Sequentieller Algorithmus

($A^{(i)}$ hat Dimension $r[i] \times r[i+1]$)

```
double c[n][n], b[n][n], r[n+1] ;
for (i = 0; i < n; i++)
  for (j = i+1; j < n; j++)
   c[i][j] = MAX_VAL ;
  for (i = 0; i < n; i++)
   c[i][i] = 0 ;
for (j = 0; j < n; j++)
  for (i = 0; i < n-j; i++)
   for (k = i+1; k < i+j; k++) {
     co = c[i][k-1]+c[k][i+j] + r[i]*r[k]*r[i+j+1];
     if (co < c[i][i+j]) {
       c[i][i+j] = co ;
       b[i][i+j] = k ;
   } }
```


Schematische Darstellung des Ablaufs



c_{1n} enthält die minimale Zahl von Multiplikationen

b_{ij} das letzte Produkt welches gebildet wird also

$$\prod_{k=i}^{j} A^{(k)} = \left(\prod_{k=i}^{b_{ij}-1} A^{(k)} \right) \cdot \left(\prod_{l=b_{ij}}^{j} A^{(l)} \right)$$

Anwendung auf das Beispiel:

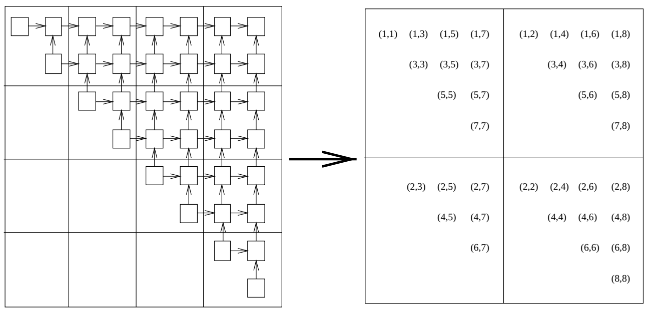
- Optimale Klammerung $(A(BC)((DE)F))$
- mit 36 Skalar-Multiplikationen und 13 Skalar-Additionen

Sequentielle Komplexität $\sum_{i=1}^{n-1} (n-i)i = \frac{n^3}{6} - \frac{n}{6} \in O(n^3)$

Parallele Realisierungen

Realisierung auf einem $\sqrt{p} \times \sqrt{p}$ Array von Prozessoren

Beste Datenverteilung shuffle-Anordnung am Beispiel



- Prozessoren berechnen Ergebnisse entlang der Diagonalen
- verschicken Werte sobald sie berechnet sind

⇒asymptotisch optimale Effizienz

4.6 Asynchrone Algorithmen

Bisher betrachtet synchrone Algorithmen:

- Prozessoren berechnen lokal ein Zwischenergebnis
- Zwischenergebnisse werden zwischen Prozessoren ausgetauscht
- Neuer Berechnungsschritt beginnt

Austausch von Ergebnissen führt zur Synchronisation

über alle Prozessoren ⇒ Synchroner Algorithmus

Vorteil

- Deterministischer Ablauf des Algorithmus

Nachteil

- Langsamster Prozessor bestimmt die Geschwindigkeit

⇒ gerade auf Workstation/PC-Netzen schlechte Leistung

Alternative asynchrone Algorithmen

- jeder Prozessor berechnet seine Zwischenergebnisse aus der vorhandenen Information über die Zwischenergebnisse anderer Prozessoren

⇒ keine Synchronisation und kein Warten

aber Konvergenz asynchroner Verfahren

oft schwer nachweisbar

4.6.1 Ein einfaches Beispiel

Gesucht wird die Lösung des folgenden Gleichungssystems:

$$x_1 = \quad ax_1 + bx_2$$
$$x_2 = \quad bx_1 + ax_2$$

Iterative Berechnung des Resultats auf
zwei Prozessoren P1 und P2

- $x_i(0)$ sind die Initialwerte
- $x_1(k)$ wird auf P1 und $x_2(k)$ auf P2 berechnet

Kommunikation zwischen P1 und P2 dauert D Zeiteinheiten

synchroner Algorithmus:

(führt Iteration aus, wenn Wert vom anderen Prozessor gelesen)

$$x_1(t+1) = \quad ax_1(t-D) + bx_2(t-D)$$
$$x_2(t+1) = \quad bx_1(t-D) + ax_2(t-D)$$

asynchroner Algorithmus:

(führt permanent Iterationen aus)

$$x_1(t+1) = \quad ax_1(t) + bx_2(t-D)$$
$$x_2(t+1) = \quad bx_1(t-D) + ax_2(t)$$

wobei $x_i(t) = x_i(0)$ für $t \leq 0$

sei $|a| + |b| < 1$

der Fixpunkt ist dann $x* = (0, 0)$ unabhängig vom Startvektor

Beide Verfahren konvergieren!

Konvergenz des asynchronen Verfahrens

Idee des Beweises (auch allg. verwendbar!):

Für $t' > 0$ und alle t mit $t' - D \leq t \leq t'$ gelte

$$|x_1(t)| < r, \ |x_2(t)| < r$$

$(x_1(t), x_2(t))$ ist eine l_∞ Approximation mit Radius r

Falls r gegen 0 konvergiert für wachsendes t ,
so ist die Konvergenz bewiesen

Wir zeigen: Für $t > t'$ ist $(x_1(t), x_2(t))$ eine l_∞ Approximation
mit Radius $r(|a| + |b|)$

Sei $t = t' + 1$:

$$|x_1(t)| = |ax_1(t') + bx_2(t' - D)| \leq$$
$$|ax_1(t')| + |bx_2(t' - D)| \leq (|a| + |b|)r$$

damit gilt für $t \geq t' + D + 2$: $|x_1(t)| \leq (|a| + |b|)^2 r$

Ausgehend von $r = \max(x_1(0), x_2(0))$ erhalten wir eine Abschätzung für alle $x_1(t)$ und in ähnlicher Form für $x_2(t)$.

Konvergenz der synchronen Iteration:

für ein $\rho > 0$ mit $|a|\rho^{-D} + |b|\rho^{-D} \leq \rho$

(bzw. $(|a| + |b|)^{1/(D+1)} \leq \rho$) gilt:

$|x_i(t)| \leq C\rho^t$ für alle $t = 0, 1, ...$

wobei $C = \max(x_1(0), x_2(0))$

Auswahl des kleinsten möglichen ρ liefert

$\rho_S = (|a| + |b|)^{1/(D+1)}$

Konvergenz der asynchronen Iteration:

Sei ρ so gewählt, dass $|a| + |b|\rho^{-D} \leq \rho$

dann gilt $|x_i(t)| \leq C\rho^t$ für alle $t = 0, 1, ...$

Sei ρ_A das minimale ρ , welches obige Ungleichung erfüllt

Allgemein gilt $\rho_A \leq \rho_S$

Anzahl übertragene Nachrichten

Iteration bis $|x_i(t)| \leq \epsilon$ (bzw. $\epsilon \leq C\rho^t$)

Womit gilt $t \geq |\log(\epsilon/C)|/|\log \rho|$

t_S und t_A Iterationszahlen der
synchronen und asynchronen Iteration

Auf Grund obiger Analyse gilt:

$\frac{t_S}{t_A} = \frac{|\log \rho_A|}{|\log \rho_S|} \geq 1$

- die Zahl der synchronen Nachrichten m_S ist höchstens $t_S/(D+1)$
- die Zahl der asynchronen Nachrichten m_A entspricht t_A

Damit gilt

$\frac{m_S}{m_A} \leq \frac{|\log \rho_A|}{(D+1)|\log \rho_S|} = \frac{|\log \rho_A|}{|\log(|a|+|b|)|}$

es gilt $|a| + |b| \leq \rho_A < 1$ für alle erlaubten a, b

$\Rightarrow m_S/m_A \leq 1$

Einige Beispiele ($\epsilon = 0.01$):

$a = b = 0.4, D = 3$:

$t_S = 67$ und $t_A = 43$, $m_s = 16$ und $m_A = 43$

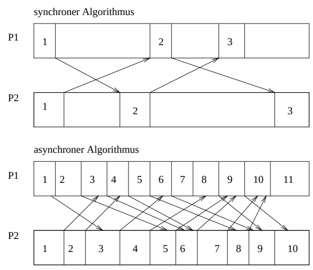
$a = b = 0.4, D = 5$:

$t_S = 100$ und $t_A = 62$, $m_s = 16$ und $m_A = 62$

$a = b = 0.2, D = 3$:

$t_S = 13$ und $t_A = 10$, $m_s = 3$ und $m_A = 10$

Ablauf der Verfahren



Viele Nachrichten

- verlangsamen die Kommunikation
- belasten den Prozessor und Speicher

Reduktion der Nachrichtenzahl im asynchronen Fall durch Senden der Ergebnisse nur nach der k-ten Neuberechnung:

- große k reduzieren die Nachrichtenzahl
- kleine k reduzieren die Iterationszahl

4.6.2 Ein asynchrones Algorithmen-Modell

Wir betrachten ein allgemeines Fixpunktproblem der Form
$x = f(x)$ mit $f = (f_1, \ldots, f_n)$ und $x = (x_1, \ldots, x_n)$
Gesucht wir Fixpunkt x^* mit $x_i^* = f(x^*)$

- T^i Menge der Zeitpunkte, zu denen neue Approximationen berechnet werden o.B.d.A sei $T^i = \{0, 1, , 2 \ldots\}$
- $\tau_i^j(t)$ Zeitpunkt zu dem der Wert von x_i auf dem Prozessor gehört, der x_j berechnet
  ($t - \tau_i^j(t)$ ist das Alter der Variablen)

Es gelte:

- $0 \leq \tau_i^j(t) \leq t$
- $\tau_i^i(t) = t$ (Prozessor kennt seine eigene Variable)
- $\lim_{t\to\infty} \tau_i^j(t) = \infty$

Falls Variable i zum Zeitpunkt $t \in T^i$ neu berechnet wird:
$x_i(t+1) = f_i(x_1(\tau_1^i), \ldots, x_n(\tau_n^i))$
Keine Beschränkungen bzgl. τ_i^j
<u>vollkommen asynchroner Algorithmus</u>
d.h. $\lim_{t\to\infty}(t - \tau_i^j(t)) = \infty$ kann gelten

Beliebig lange Kommunikationszeiten sorgen in manchen Fällen für Schwierigkeiten
$\Rightarrow$ partiell asynchrone Algorithmen
Annahmen:

- Jeder Prozessor berechnet mindestens einmal neue Werte für seine Variablen innerhalb jedem Intervall der Länge B
- Die Informationen eines Prozessors sind maximal B Zeiteinheiten alt

$\Rightarrow$Annahmen treffen für viele reale Systeme zu bzw. können durch Einführung von Synchronisationspunkten erzwungen werden

Es gibt Algorithmen, die

- für jedes endliche B konvergieren, aber für $B = \infty$ divergieren
- nur konvergieren, falls B "klein genug" ist

Konvergenzbeweise oft schwierig

4.6.3 Asynchrone Algorithmen für lineare Gleichungssysteme

Konvergenzbeweis ist ausweitbar auf Matrizen A mit

$$\sum_{j=1}^n |A_{i,j}| < 1$$

allgemeiner:
Für das Gleichungssystem $x = Ax + b$
konvergiert die asynchrone Iteration

$x_i(t+1) = \sum_{j=1}^{n} A_{i,j} x_j(\tau_j^i(t)) + b_i$ für $t \in T^i$

$x_i(t+1) = x_i(t)$ für $t \notin T^i$

für $\rho(|A|) < 1$ gegen den gesuchten Fixpunkt
$|A|$ entsteht aus A, indem alle Elemente durch die Beträge ersetzt werden
(deutlich einschränkender als die Bedingung $\rho(A) < 1$)
Es reicht aus $t - 2 \leq \tau_j^i(t) \leq t$ zuzulassen, um Divergenz für alle A mit $\rho(|A|) \geq 1$ zu induzieren
(auch wenn $\rho(A) < 1$)
dazu notwendig ist aber $t - 2 \leq \tau_i^i(t) \leq t$
andernfalls hängt Konvergenz-Divergenz von A und nicht nur von $\rho(|A|)$ ab

Anwendungsbeispiel Markov-Ketten:
stochastischer Prozess mit Zustandsraum $S = \{1, \ldots, n\}$
Folge von Zufallsvariablen $X_i \in S$ mit Übergangswahrscheinlichkeiten

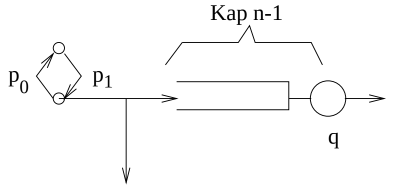
$$P\left(X_{k+1} = j | X_0 = i_0, \ldots, X_{k-1} = i_{k-1}, X_k = i\right)$$
$$P\left(X_{k+1} = j | X_k = i\right) = P_{ij}$$

Übergangswahrscheinlichkeiten werden in einer stochastischen Matrix $P \in \mathbb{R}^{n\times n}$ zusammengefasst
(d.h. $P_{ij} \geq 0$ und $\sum_{j=1}^{n} P_{ij} = 1$ für alle $i \in S$)
Weiterhin nehmen wird an, dass P eine primitive stochastische Matrix ist, so dass gilt

- Es existiert ein eindeutiger Zeilenvektor $\pi^* = \pi^* P$ und $\sum_{i=1}^{n} \pi_i = 1.0$
- Der Grenzwert $\lim_{k\to\infty} P^k$ entspricht einer Matrix, in der alle Zeilen gleich π^* sind
- Für jeden Vektor $\pi^{(0)} \geq 0$ mit $\sum_{i=1}^{n} \pi_i^{(0)} = 1.0$ konvergiert die Iteration $\pi^{(k+1)} = \pi^{(k)} P$ gegen π^*

π^* ist die stationäre Verteilung der Markov-Kette aus der Leistungsgrößen abgeleitet werden können

Anwendungsbeispiel einfaches Bediensystem



p_i und q sind Wahrscheinlichkeiten $\overline{p}_i = 1 - p_i$ und $\overline{q} = 1 - q$

$$P = \begin{pmatrix} P_{oo} & P_{01} & & & & \\ P_{10} & P_{11} & P_{12} & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & P_{(n-1)(n-2)} & P_{(n-1)(n-1)} & P_{(n-1)n} \\ & & & & P_{(n-1)n} & P_{nn} \end{pmatrix}$$

blocktridiagonale Struktur mit den Untermatrizen

$$P_{00} = \begin{pmatrix} \overline{p}_1 & p_1 \\ 0 & 0 \end{pmatrix} \; P_{01} = \begin{pmatrix} 0 & 0 \\ p_0 & \overline{p}_0 \end{pmatrix}$$

$$P_{i(i-1)} = q \begin{pmatrix} \overline{p}_1 & p_1 \\ 0 & 0 \end{pmatrix} \; P_{ii} = \begin{pmatrix} \overline{qp}_1 & \overline{q}p_1 \\ qp_0 & q\overline{p}_0 \end{pmatrix}$$

$$P_{i(i+1)} = \overline{q} \begin{pmatrix} 0 & 0 \\ p_0 & \overline{p}_0 \end{pmatrix}$$

$$P_{n(n-1)} = q \begin{pmatrix} \overline{p}_1 & p_1 \\ p_0 & \overline{p}_0 \end{pmatrix} \; P_{nn} = \overline{q} \begin{pmatrix} \overline{p}_1 & p_1 \\ p_0 & \overline{p}_0 \end{pmatrix}$$

Vollständig asynchroner Algorithmus

Ges. $\pi^* P = \pi^*$

wobei P eine primitive stochastische Matrix ist

da $P \geq 0$ gilt $\rho(P) = \rho(|P|) = 1$

Zerlegung von P

$$P = \begin{pmatrix} p_{11} & r \\ s & \widetilde{P} \end{pmatrix}$$

- r, s sind Zeilen-/Spaltenvektoren der Dim. $n-1$
- p_{11} ist ein Skalar und
- $\widetilde{P}$ ist eine substochastische $n-1 \times n-1$ Matrix

Es gilt $\rho(\widetilde{P}) = \rho(|\widetilde{P}|) < 1$

Löse für beliebiges $c > 0 : x\widetilde{P} + cr = x$ (asynchron)

$\Rightarrow$ Bedingungen für asynchrone Konvergenz sind erfüllt

Es gilt dann $\pi = (c/(xe^T + c), x/(xe^T + c))$

Bew. offensichtlich gilt $\pi e^T = 1$

weiterhin gilt $\pi P = \pi$, da

$(xe^T + c)^{-1}(cp_{11} + xs) =$

$(xe^T + c)^{-1}(c(1 - re^T) + x(I - \widetilde{P})e^T =$

$(xe^T + c)^{-1}(c - (cr + x - x - cr)e^T = c/(xe^T + c)^{-1}$

und ähnlich für x

Problem: schlechte Konvergenz

Ein Beispiel für ein partiell asynchrones Verfahren

$x = x - \gamma A x$

mit $A = \begin{pmatrix} 1+\epsilon & 1 & 1 \\ 1 & 1+\epsilon & 1 \\ 1 & 1 & 1+\epsilon \end{pmatrix}$

mit $0 < \epsilon < 1$ und $0 < \gamma < 1 - \epsilon$

Wir betrachten eine partiell asynchrone Iteration, bei der

- ein Prozessor eine Variable berechnet
- in jedem Schritt ein neuer Wert für die lokale Variable berechnet wird
- Werte zwischen den Prozessoren alle B Zeiteinheiten ausgetauscht werden

Sei $t_k = kB$ dann gilt für $t_k \leq t < t_{k+1}$:

$$x_1(t+1) = x_1(t) - \gamma((1+\epsilon)x_1(t) + x_2(t_k) + x_3(t_k))$$

damit gilt für $x_1(t_{k+1})$:

$$x_1(t_{k+1}) = \\ (1-\gamma(1+\epsilon))^B x_1(t_k) - \tfrac{1-(1-\gamma(1+\epsilon))^B}{1+\epsilon}(x_2(t_k) + x_3(t_k))$$

ähnlich werden $x_2(t_{k+1})$ und $x_3(t_{k+1})$ berechnet

Damit gilt: $x(t_{k+1}) = Cx(t_k)$

mit $C = \begin{pmatrix} \alpha & \beta & \beta \\ \beta & \alpha & \beta \\ \beta & \beta & \alpha \end{pmatrix}$

$\alpha = (1 - \gamma(1+\epsilon))^B$ und $\beta = -\frac{1-(1-\gamma(1-\epsilon))^B}{1+\epsilon}$

Konvergenz der Iteration falls $\rho(C) < 1$

C hat die Eigenwerte $\alpha + 2\beta$ und $\alpha - \beta$ (2fach)

Sei $\gamma(1+\epsilon) < 1$ dann gilt für $B \to \infty$:

$\alpha \to 0$ und $\beta \to -1/(1+\epsilon)$

damit gilt $|\alpha + 2\beta| \to 2/(1+\epsilon) > 1$

Falls γ sehr klein ist gilt

$$\alpha \approx 1 - B\gamma(1+\epsilon) \text{ und } \beta \approx -B\gamma$$

Eigenwerte $1 - B\gamma(3+\epsilon)$ und $1 - B\gamma\epsilon$

Für $B\gamma(3+\epsilon) < 1$ ist $\rho(C) < 1$

$\Rightarrow B < (\gamma(3+\epsilon))^{-1}$ muss gelten!

i.a. kann bei Problemen dieser Art eine Schranke für B nicht a priori bestimmt werden

Partiell asynchroner Algorithmus für Markov Ketten

Voraussetzungen

1. P ist primitive stochastische Matrix

2. es existiert ein i mit:

   (a) $P_{ii} > 0$
   (b) $\tau_{i,i}(t) = t$
   (c) $\pi_i(0) > 0$

3. es existiert ein $B < \infty$, so dass

   (a) $t - B \leq \tau_{i,j}(t) \leq t$ für alle $t \geq 0$ und $1 \leq i, j \leq n$
   (b) jedes Element alle B Zeiteinheiten neu berechnet wird

Unter diesen Bedingungen konvergiert die partiell asynchrone Iteration gegen $\widetilde{\pi} = c\pi$ mit $c > 0$

Die Konvergenz ist geometrisch

- Da kein festes B gefordert ist, funktioniert das Verfahren auf unterschiedlichsten Systemen
- Konvergenz ist i.d.R. schneller als beim vollständig asynchronen Verfahren
- Für große B oftmals langsame Konvergenz

Multisplitting Verfahren und chaotische Iterationen

wir betrachte $Ax = b$ wobei A monoton ist,
(d.h. A ist nicht singulär und $A^{-1} \geq 0$)
Splitting $A = M - N$ wobei M^{-1} existiert
Splitting ist regulär falls $M^{-1} \geq 0$ und $M^{-1}N \geq 0$
Multisplitting $A = M_l - N_l$ $(l = 1, ..., k)$ (regulär)
Sei E_l $(l = 1, ..., k)$ eine Menge von Gewichtsmatrizen, d.h. Diagonalmatrizen mit $\sum_{l=1}^{k} E_l = I$
Iteration $x(t+1) = \sum_{l=1}^{k} E_l M_l^{-1}(N_l x(t) + b)$
es gilt $\rho(\sum_{l=1}^{k} E_l M_l^{-1} N_l) < 1$

Idee der Parallelisierung (erst einmal synchron):

- jeder Prozessor bekommt ein $l \in \{1, ..., k\}$ zugeteilt und
- berechnet $E_l M_l^{-1}(N_l x(t) + b)$ (nur Zeilen x mit $E_l(x, x) \neq 0$ müssen berechnet werden)

Zwei konvergente asynchrone Varianten

Asynchrone Iteration Variante 1:

- jeder Prozessor führt eine Anzahl von Iterationen aus
- Vektor wird in einem synchronen Schritt neu berechnet

Asynchrone Iteration Variante 2:

- jeder Prozessor kann den globalen Vektor lesen
- jeder Prozessor kann den globalen Vektor schreiben
- schreiben und lesen sind atomar

Asynchroner Bellman-Forth Algorithmus

Voraussetzungen wie bei der synchronen Version

Darstellung der asynchronen Iteration:

- initialisiere $x_0^{(0)} = 0$ und $x_i^{(0)}$ $(i = 1, \ldots, n-1)$ beliebig
- iteriere $x_i(t+1) = \min_{j \in A(i)}(a_{ij} \cdot x_j(\tau_i^j(t)))$

Algorithmus berechnet nach endlicher Zeit die Lösung , falls

$x_i(0) \geq x_i^*$ für alle $i = 1, \ldots, n-1$

Implementierung:

Prozesse verschicken Nachrichten, wenn sie einen neuen Wert berechnet haben (Terminierungserkennung später)

Vergleich des asynchronen und synchronen Algorithmus

für $x_0(0) = 0$ und $x_i(0) = \infty$ für $i = 1, \ldots, n-1$

Definition eines äquivalenten Prozesses:

- Token welches von Knoten zur Knoten läuft und dabei die Pfadlängen addiert
- Erreicht Token einen Knoten, so wird es weitergereicht, falls noch kein Token mit kleinerer Pfadlänge eingetroffen ist

Prozess beschreibt den Ablauf des
Bellman-Forth Algorithmus

Notationen:

- Länge des Pfades $p = (i_1, \ldots, i_{k+1})$:
  $L_p = \sum_{m=1}^{k} a_{i_m i_{m+1}}$
- Kommunikationszeit Pfad p: $t_p = \sum_{m=1}^{k} t_{i_m i_{m+1}}$
  wobei t_{ij} Übertragungszeit von i nach j ist
- Für $i > 0$ $P_i(t)$ = Menge der Pfade p, die in i starten und in 0 mit $t_p \leq t$

Mit obiger Notation gilt:

$$x_i(t) = \begin{cases} \infty & P_i(t) = \emptyset \\ \min_{p \in P_i(t)} L_p & sonst \end{cases}$$

$x_i(t)$ Wert der asynchronen Iteration zum Zeitpunkt t

Ergebnis ist berechnet nach

$$t^A = \max_{i \in \{1, \ldots, n-1\}} \min \{t_p | p \in P_i(\infty),\ L_p = x_i^*\}$$

(Annahme Berechnungszeit vernachlässigbar bzw. in t_p enthalten)

Ablauf synchroner Algorithmus:

Prozess zur Berechnung von i sendet in jeder Iteration seinen aktuellen Wert an alle Prozesse die $j \in A(i)$ berechnen

- $P_i^k =$ Menge der Pfade von i nach 0 mit k oder weniger Kanten
- $k_i(t) = \max\{k | t_p \leq t, \forall p \in P_i^k\}$ Anzahl synchrone Iterationen zum Zeitpunkt t

damit gilt für den Wert der synchronen Iteration

$$x_i^S(t) = \begin{cases} \infty & P_i^{k_i(t)} = \emptyset \\ \min_{p \in P_i^{k_i}} L_p & sonst \end{cases}$$

da $P_i^{k_i} \subseteq P_i(t)$ gilt $x_i(t) \leq x_i^S(t)$ für alle i und t

$\Rightarrow$ der synchrone Algorithmus kann nicht schneller terminieren als der asynchrone Algorithmus

Nachrichtenzahl

Asynchrone Algorithmus überträgt mindestens so viele Nachrichten wie der synchrone Algorithmus, in Extremfällen werden deutlich mehr Nachrichten übertragen!

4.6.4 Organisation asynchroner Algorithmen

Bisher betrachtet: asynchrone Algorithmen und deren Konvergenz

Bei der praktischen Realisierung treten weitere Probleme auf:

- wie startet man einen verteilten Algorithmus?
- wie verteilt man die verfügbaren Ressourcen?
- wie terminiert man einen verteilten Algorithmus?
- wie erkennt man Konvergenz des Algorithmus?
- ...

Die meisten Fragen sind leichter für synchrone Probleme zu beantworten aber auch im synchronen Fall ist eine effiziente Realisierung notwendig

Wir betrachten *message-passing* Systeme

Terminierungserkennung

Zentrales Problem:

Wann kann der Algorithmus beendet werden?

Abbruch nach Zeitschranke ist unkritisch

Eigentlich gewünscht:

Terminierung wenn die Lösung genau genug berechnet wurde

Lokale Terminierungsbedingung: $|x_i - f_i(x^i)| \leq \epsilon$

Annahmen über Prozesse:

Falls die Terminierungsbedingung für Prozessor i erfüllt ist

- so versendet i keine Nachrichten mehr
- die Terminierungsbedingung bleibt erfüllt so lange keine Nachrichten eintreffen

Annahmen über das Kommunikationsnetz

- keine Verluste
- Nachrichtenreihenfolge bleibt nicht unbedingt erhalten

Terminierung ist erreicht wenn:

- lokale Terminierungsbedingung für alle Prozessoren gilt
- keine Nachrichten mehr unterwegs sind

Sei $\overline{t}$ der erste Zeitpunkt, zu dem die Bedingungen gelten (offensichtlich gelten sie auch für alle $t > \overline{t}$)

Annahmen zur Vereinfachung der Analyse

- Empfang einer Nachricht wird durch ein ACK bestätigt
- ACKs werden nicht bestätigt
- Nachrichten und ACKs erreichen nach endlicher Zeit den Empfänger
- jeder Prozessor sendet/empfängt in einem Schritt maximal eine Nachricht an/von jeden anderen Prozessor
- simultanes Senden und Empfangen ist nicht möglich
- wir betrachten getaktete Zeit

Protokoll zur Terminierungserkennung:

Prozesse können im aktiven oder inaktiven Zustand sein

- im inaktiven Zustand werden keine Nachrichten oder ACKs verschickt
- im aktiven Zustand werden Nachrichten und ACKs verschickt
- Übergang von inaktiv zu aktiv durch Empfang einer Nachricht

Ausgehend von einem Initiatorprozess werden werden Prozesse aktiviert

Terminierung ist eingetreten, falls Initiator inaktiv wird

Wenn p im inaktiven Zustand ist und zum Zeitpunkt t eine Nachricht von q empfängt

1. so erfolgt ein Übergang in den aktiven Zustand
2. die zugehörige Nachricht ist eine kritische Nachricht
3. q ist Elter von p und p Kind von q

Annahmen

- Aktive Prozesse senden (in endlicher Zeit) ACKs für empfangene Nachrichten mit Ausnahme der kritischen Nachricht
- ACK für die kritische Nachricht wird gesendet, wenn der Prozess vom aktiven in den inaktiven Zustand übergeht
- Nachrichten und ACK Übertragung dauert mindestens eine Zeiteinheit

Übergang vom aktiven in den inaktiven Zustand zum Zeitpunkt t falls

- keine Nachricht zum Zeitpunkt t empfangen wird
- die lokale Terminierungsbedingung gilt
- alle empfangenen Nachrichten, bis auf die kritische Nachricht, wurden per ACK bestätigt
- alle gesendeten Nachrichten wurden bestätigt

Bedingungen für Übergang aktiv - inaktiv gelten auch für Initiator ohne Elter

Initiator muss keine kritische Nachricht bestätigen

Falls j aktiv ist und Kind von i ist, so muss auch i aktiv sein (die kritische Nachricht kann noch nicht bestätigt worden sein)

Für jeden inaktiven Prozess gelten folgende Bedingungen

a) das lokale Terminierungskriterium ist erfüllt

b) alle gesendeten Nachrichten wurden bestätigt

c) alle empfangenen Nachrichten wurden bestätigt

Bei aktiven Prozessen ist mindestens eine Bedingung verletzt

Start der Prozedur mit Auswahl eines Initiatorprozesses

zum Zeitpunkt $t = 0$ ist der Initiator aktiv und alle anderen Prozesse sind inaktiv

d.h. Initiator weckt die anderen Prozesse durch Versenden von Nachrichten

Falls der Initiator zum ersten Mal inaktiv wird, ist die Terminierung des Algorithmus erkannt

Zu zeigen bleibt:

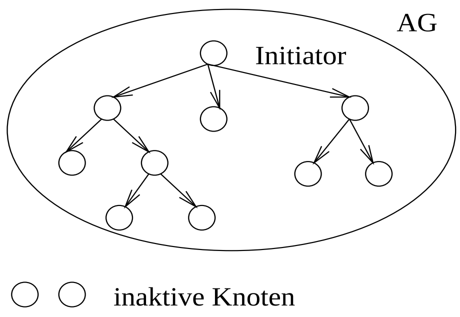
1. wenn zum Zeitpunkt t Terminierung erkannt wurde, so ist die Terminierung zum Zeitpunkt $t' \leq t$ eingetreten
2. wenn Terminierung zum Zeitpunkt t eintritt, so wird sie zum Zeitpunkt $t' \geq t$ erkannt

Beweis der Aussagen mit Hilfe des ***Activity Graphs*** (AG)

AG zum Zeitpunkt t beinhaltet alle aktiven Prozesse und eine gerichtete Kante vom Elter zum Kind

1. der Initiator gehört zu AG bevor Terminierung erkannt wurde
2. der Elterprozess eines Prozesses aus AG ist eindeutig und gehört zu AG
3. falls Prozess j zum Zeitpunkt t zu AG gehört, so gehört der Elterprozess im Intervall $[t-2, t+2]$ zu AG

<u>AG ist ein Baum mit dem Initiator als Wurzelknoten</u>



also keine Zyklen und ein eindeutiger Weg von der Wurzel zu jedem Knoten

dies gilt wg. 1. und 2. (oben) und dem Beweis, dass AG keine Zyklen enthält

<u>Bew. AG enthält keine Zyklen</u>:

- sei t Zeitpunkt zu dem Zyklus C erstmals entsteht
- d.h. jeder Prozess aus C hat Elter in C
- aus 3. (vorherige Folie) folgt: jeder Prozess aus C war aktiv bei $t - 1$
- eine Kante zwischen aktiven Prozessen wurde eingefügt

<u>Terminierung wurde zum Zeitpunkt t' entdeckt $\Rightarrow$</u>

<u>Terminierung erfolgte zum Zeitpunkt $t \leq t'$</u>

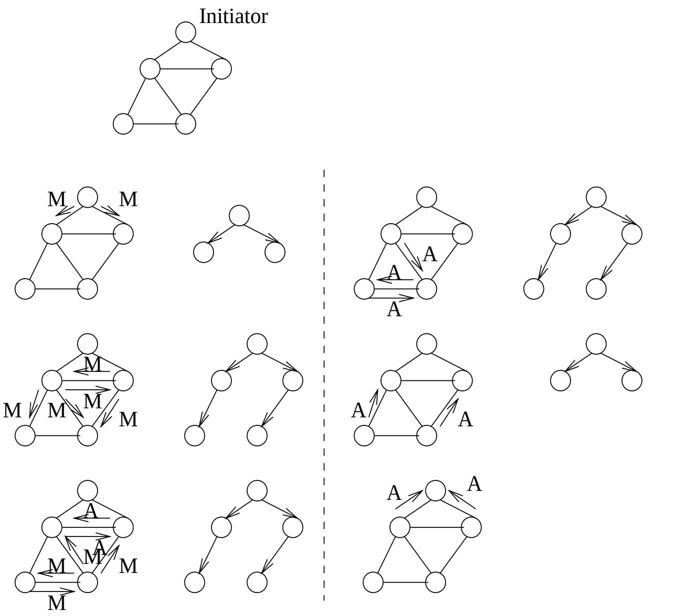
- nach Abschluss von $t' - 1$ ist der Initiator der einzige aktive Prozess (wg. 3.)
- zu t' kann kein Prozess aktiv werden (wg. 3.)
- damit wird AG zum Zeitpunkt t leer
- Regel für den Übergang von aktiv zu inaktiv legen fest:
  - alle lokalen Terminierungsbedingungen gelten
  - alle Nachrichten wurden bestätigt

<u>Terminierung erfolgte zum Zeitpunkt t $\Rightarrow$</u>

<u>Terminierung wurde zum Zeitpunkt t' entdeckt</u>

- nach t werden keine Nachrichten mehr verschickt
- damit kein Übergang mehr von inaktiv nach aktiv
- damit würde Terminierung nur nicht erkannt, wenn AG konstant: dies kann nicht gelten, da Prozesse ohne Kinder irgendwann ihre ACKs empfangen und inaktiv werden

Beispiel: Broadcast vom Initiator



Initiator sendet Pakete an alle Prozesse und möchte wissen, wann alle Pakete empfangen wurden

Zwei Arten von Nachrichten

- M für Broadcast-Nachrichten
- A für Acknowledgements

Schnappschuss

Im verteilten System hat Prozessor nur eine lokale Sicht

oft aber notwendig oder wünschenswert Information über den globalen Zustand des Systems

z.B. zur Beobachtung der Konvergenz eines asynchronen Iterationsverfahrens

unbefriedigende Lösung: Synchronisation aller Prozesse und Einsammeln der Ergebnisse durch einen Prozess

bessere Lösung: verteilte Bestimmung während des asynchronen Ablaufs ($\Rightarrow$ Schnappschussalgorithmus)

Annahmen bzgl. der Kommunikationsstruktur:

- fehlerfreie Übertragung
- FIFO Reihenfolge
- Übertragung nach endlicher Zeit

Gerichteter Graph $G = (N, A)$ beschreibt die Kommunikationsstruktur

Jeder Prozess hat lokale Zeit t_i und

Funktion $h_i(t_i)$ bildet lokale Zeit auf virtuelle globale Zeit eines externen Beobachters ab

(nicht notwendig aber hilfreich)

- x_i Wert der lokalen Variablen in Prozessor i
- M_{ij} mögliche Nachrichten von i nach j für $(i, j) \in A$
- Zustand von $(i, j) \in A$ umfasst alle Nachrichten, die von i nach j gesendet aber noch nicht empfangen wurden
- globaler Zustand umfasst Werte aller lokalen Variablen und Zustände der Kommunikationskanäle
- Berechnungen der Prozesse in Form von Ereignissen (selbst- oder nachrichteninduziert)
- Ereignis ändert den Wert der lokalen Variablen x_i und generiert u.U. Nachrichten m_{ij}

Annahmen:

1. Ereignisse sind atomar
2. unterschiedliche Ereignisse auf einem Prozessor treten zu unterschiedlichen Zeitpunkten auf
3. unterschiedliche Nachrichten werden zu unterschiedlichen Zeitpunkten empfangen
4. nachrichteninduzierte Ereignisse werden durch genau eine Nachricht induziert

- Falls Nachricht m_{ij} zum Zeitpunkt t_i gesendet und t_j empfangen wurde, gilt $h_i(t_i) < h_j(t_j)$
- Falls für eine andere Nachricht m_{jk} die Relation $h_j(t_j) < h_k(t_k)$ gilt, so gilt auch $h_i(t_i) < h_k(t_k)$ (Transitivität)
- Andererseits: falls zwei lokale Zeiten t_i und t_j nicht durch Nachrichten vergleichbar sind, so nennt man sie simultan

Etwas formaler:

Eine Menge lokaler Ereignisse $\{t_i | i \in N\}$ heißt simultan, falls jede Nachricht m_{ij}, die nach t_i gesendet wurde nach t_j empfangen wird

Ein zu einer Menge simultaner Ereignisse $\{t_i | i \in N\}$ gehöriger Schnappschuss umfasst die Variablenwerte x_i zu Zeitpunkten t_i und zusätzlich alle Nachrichten m_{ij}, die vor t_i gesendet und nach t_j empfangen wurden (aus Konsistenzgründen seien die t_i keine Ereigniszeitpunkte!)

Algorithmus basiert auf markierten Nachrichten

Wir nehmen an, dass markierte Nachrichten den Ablauf des Algorithmus nicht beeinflusst

Schnappschuss-Algorithmus (Chandy-Lamport85)

1. Initialisierung: Prozesse initiieren Schnappschuss durch Senden des Markers an ihre Nachbarn
(sei t_i^* die zugehörige lokale Zeit)

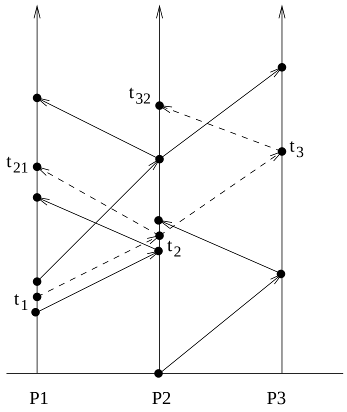
2. Protokollierung der Prozesszustände: Jeder Prozess protokolliert seinen lokalen Zustand, wenn er erstmals den Marker empfängt (sei t_i^* die zugehörige lokale Zeit)

3. Marker Generierung: Jeder Prozess, der erstmals einen Marker empfängt und nicht der Initiator ist, sendet einen Marker über alle Ausgangsleitungen

4. Protokollierung der Leitungszustände: Für $(i, j) \in A$ sei t_{ij}^* der Zeitpunkt, zu dem j erstmals einen Marker von i empfängt. Prozessor j protokollierte den Leitungszustand (i, j) durch protokollieren aller Nachrichten, die zwischen t_j^* und t_{ij}^* empfangen wurden

Zu zeigen bleibt, dass der Algorithmus korrekt ist.

Beispielablauf



Gestrichelte Linie beschreibt markierte Nachrichten

- t_1 Initiierung durch $P1$
- $P1$ speichert alle empfangenen Nachrichten, die zwischen t_1 und t_{21} auf Kanal $2 \to 1$ empfangen wurden
- dito für $P2$ und $P3$

Schnappschuss besteht aus

- Zustand $P1$ zum Zeitpunkt t_1 plus Nachricht $P2 \to P1$
- Zustand $P2$ zum Zeitpunkt t_2 plus Nachricht $P3 \to P2$
- Zustand $P3$ zum Zeitpunkt t_3

Der Algorithmus terminiert und jeder Prozess hat seinen Zustand und die Zustände hereinkommender Leitungen protokolliert

- Da der Kommunikationsgraph stark zusammenhängen ist empfängt jeder Prozess irgendwann einen Marker
(wg. 3.)
- Damit ist t_i^* definiert und jeder Prozess protokolliert seinen lokalen Zustand
- Über jede Leitung $(i, j) \in A$ wird genau ein Marker geschickt (wg. 1. und 3.)
- Damit ist t_{ij}^* definiert und jeder Leitungszustand wird protokolliert
- Da die Zahl der gesendeten Marker endlich ist und die Übertragungszeit jedes Markers endlich ist, terminiert der Algorithmus nach endlicher Zeit

Die lokalen Zeiten t_i^* sind simultan

- Sei $(i, j) \in A$ und sei Nachricht m_{ij} nach t_i^* von i nach j abgeschickt worden.
- Aufgrund der FIFO Kanäle kommt m_{ij} nach dem Marker und damit nach t_j^* bei j an

Protokollierte Nachrichten auf Kanal (i, j) wurden vor t_i^* abgeschickt und nach t_j^* empfangen

- Es werden Nachrichten protokolliert, die zwischen t_j^* und t_{ij}^* eintreffen
- Aufgrund der FIFO Kanäle entspricht dies den Nachrichten, die vor t_i^* gesendet und nach t_j^* empfangen wurden

Terminierung kann erkannt werden, indem jeder Prozess seine protokollierten Daten zum Initiator schickt

Damit erhält der Initiator eine Sicht auf den globalen Zustand (dies muss nicht ein Zustand sein, der real angenommen wurde!)

Was erkennt der Schnappschuss Algorithmus?

z.B. invariante Teilmengen $S_0 \subseteq S$

S_0 ist eine invariante Teilmenge, falls nachdem ein Zustand aus S_0 angenommen wurde, jeder Nachfolgezustand auch aus S_0 ist