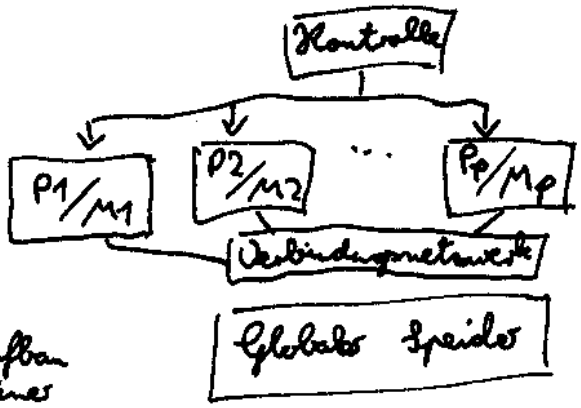
1. Einführung

18.04.2007
VL Logen
Par Verf Alg.

1.1 Das PRAM-Rechnermodell

PRAM - Parallel Random Access Machine
abstraktes Rechnermodell zur Beschreibung
paralleler Alg. (Fortune, Wyllie, 1978)

einfach, aber unrealistisch
→ Konzentration auf inhärente P. im Alg.
→ SIMD (Single Instruction Multiple Data) Modell
⇒ synchrone P.



Prozessor-Speicher-Paare
(Prozessorelemente)

Aufbau eines PRAM

Arbeitsweise einer PRAM

- Ein-/Ausgabe erfolgt über globalen Speicher
- Die Berechnung beginnt mit einem einzelnen aktiven Prozessorelement (PE)
- In einem Berechnungsschritt kann ein aktives PE
  - ein anderes PE aktivieren
  oder
  - einen lokalen oder einen globalen Speicherplatz lesen oder schreiben
  oder
  - eine einzelne RAM-Operation ausführen, d.h. eine ALU-Operation, einen Sprung, ...
- Alle aktiven PEs führen dieselbe Instruktion aus, allerdings auf unterschiedlichen Speicherzellen.
- Die Berechnung terminiert, wenn das letzte PE stoppt.

Kosten einer PRAM-Berechnung

#PEs $\star$ parallele Zeitkomplexität
(# Berechnungsschritte)

Speicherorganisation

- EREW (Exclusive Read Exclusive Write)
  keine Lese-/Schreibkonflikte
- CREW (Concurrent Read Exclusive Write)
  gleichzeitiges Lesen, exklusives Schreiben
  → Default-Modell
- CRCW (Concurrent Read Concurrent Write)
  gleichzeitiges Lesen und Schreiben
  Auflösung von Schreibkonflikten:
  - COMMON: Alle müssen denselben Wert schreiben
  - ARBITRARY: Zufällige Auswahl eines Schreibers
  - PRIORITY: z.B. Der Prozessor mit kleinstem Index oder größtem Index darf schreiben

Satz 1.1: Eine p-Prozessor CRCW-PRIORITY-PRAM kann durch eine p-Prozessor EREW-PRAM simuliert werden. Dabei wird die Zeitkomplexität um einen Faktor $\Theta(\log p)$ erhöht. 19.04.2007 Θ theta

Bsp. Wir betrachten PRAMs mit $p = 2^k$ Prozessoren.

Bsp. In einem Datenbereich mit $n \gg p$ Werten soll ein Element x gesucht werden.

seq. Alg.: Durchlauf durch Datenbereich, bis x gefunden wurde oder Ende erreicht.
Zeitkomplexität: im Mittel: $\frac{n}{2}$ Schritte

EREW-PRAM
Sei P_i der i-te Prozessor mit $0 \le i \le p-1$

Vorphase Aktivierung aller p Proz.
und 'Broadcast' des Wertes x
(wegen ER erforderlich)

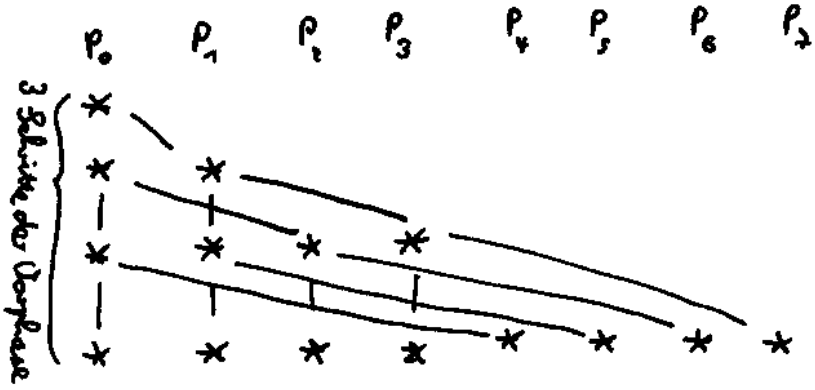
Stufe 1: P_0 liest x, aktiviert P_1 und gibt x an P_1 weiter
("P_0 schreibt x in Speicherzelle, wo P_1 liest")

Stufe 2: P_0 und P_1 aktivieren P_2 und P_3
und geben x an diese weiter.

⋮

Stufe (j): Alle P_i, $0 \le i \le 2^{j-1} - 1$,
aktivieren alle $P_{i+2^{j-1}}$
und geben x an diese weiter

Nach $\log p = k$ Schritten sind alle Proz aktiv und kennen x.



Für die eigentliche Parallelverarbeitung wird der Datenbereich
in p Teilbereiche der Größe $(n \operatorname{div} p)$ bzw. $(n \operatorname{div} p)+1$
zerlegt.

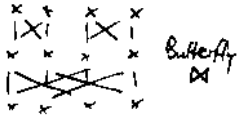
Jeder Proz P_i, $0 \le i \le p-1$, durchläuft im Gleichtakt seinen
Datenbereich und sucht den Wert x.

Nach jedem Vergleichsschritt muss geprüft werden, ob x gefunden wurde.
Da CW nicht möglich ist, schreibt jeder Proz sein Endsignal in
einen eigenen Platz im globalen Speicher.

Durch einen „umgekehrten Broadcast" werden in $\log p$ Schritten
alle Endsignale mit der log. Disjunktion (oder) zu einem Endsignal reduziert.
Dieses wird anschließend per Broadcast allen Proz mitgeteilt
$\Rightarrow 1 + 2 \log p$ Schritte pro Iteration

Insgesamt: worst case Zeitkomplexität $c * \log p + \left\lceil \frac{n}{p} \right\rceil (1 + 2 \log p)$ Schritte.

CREW-PRAM



Anfänglicher Broadcast von x ist nicht notwendig, da alle Prozessoren x in einem Schritt lesen können.
Bei der Propagation und Reduktion der Endsignale kann jeder Prozessor nun gleichzeitig die Endsignale seiner Partner lesen und verknüpfen, so dass nach $\log p$ Schritten alle Proz. wissen, ob x gefunden wurde. $\tilde{c} \log p + \lceil \frac{n}{p} \rceil (1 + \log p)$ $\quad \tilde{c} < c$

Beweis von Satz 1.1:

Es wird gezeigt, dass jeder CW-Schritt einer CRCW-PRIORITY-PRAM in $\Theta(\log p)$ Schritten auf einer EREW-PRAM simuliert werden kann.

Hilfsresultat (Beweis später):
Eine p-Prozessor EREW-PRAM kann ein Feld mit p Elementen im globalen Speicher in $\Theta(\log p)$ Schritten sortieren.
Annahme, die PRIORITY-PRAM besitzt die Proz. $P_1, \dots, P_p$ und die globalen Speicherplätze $M_1, \dots, M_m$.
Simulation eines CW-Schrittes, bei dem Prozessor P_i auf die Speicherzelle M_{i_j} schreiben möchte:
Die EREW-PRAM verwendet folgende Hilfsspeicherplätze im globalen Speicher:
$T_1, \dots, T_p$ „Schreibwünsche"
und $S_1, \dots, S_p$ „Schreibgenehmigungen"

* Prozessor P_i der EREW-PRAM schreibt exklusiv in die Speicherzelle T_i das Paar (j_j, i) „P_i schreibt in M_j"

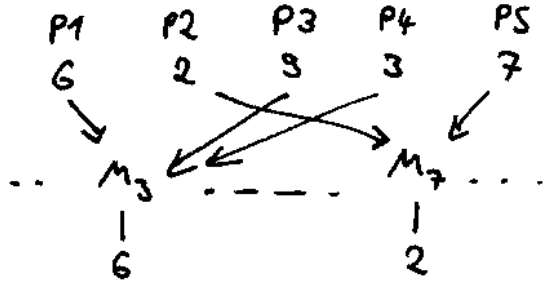
* Die EREW-PRAM sortiert das Feld $T_1, \dots, T_p$ in $\Theta(\log p)$ Schritten.

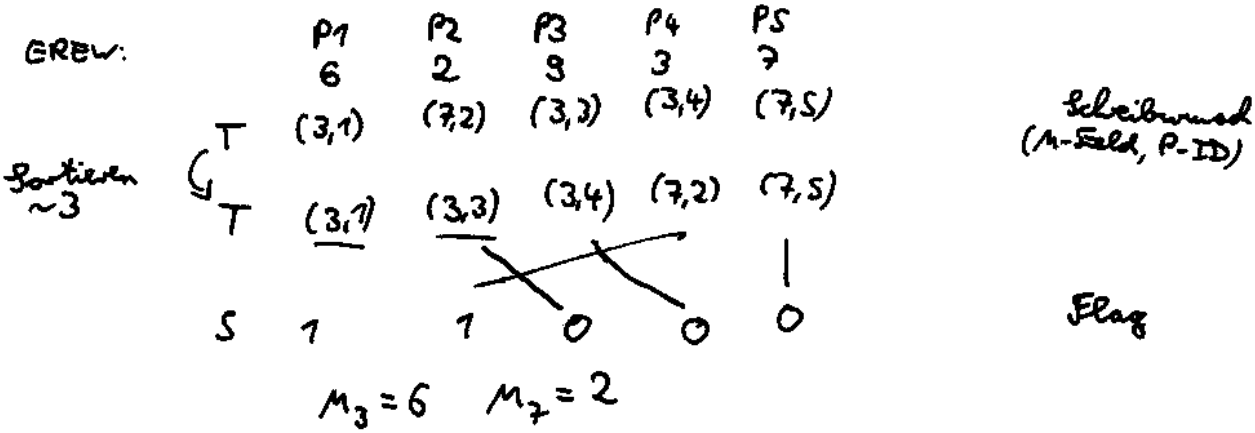
- P_1 liest T_1 etwa (j_1, i_1) und schreibt eine 1 in S_{i_1}.
- Die Prozessoren P_k, $2 \le k \le p$, lesen nun jeweils $T_k = (j_k, i_k)$ und $T_{k-1} = (j_{k-1}, i_{k-1})$
  Falls $j_k = j_{k-1}$ (Schreibkonflikt) wird $S_{i_k} = 0$ gesetzt.
  Falls $j_k \neq j_{k-1}$, wird $S_{i_k} = 1$ gesetzt, d.h. es darf geschrieben werden.

- Das S-Feld gibt an, welche Prz. exklusiv schreiben dürfen.

⇒ Simulationsaufwand pro CW-Schritt:
$\Theta(\log p)$ [Sortieren].

Bsp. $p = 5$



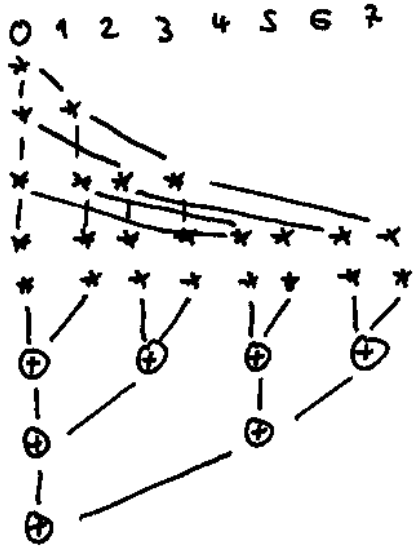


ÜB HS IV

25.04.2007

Elementare PRAM-Algorithmen

* Broadcast / Reduktion



meist Verwendung eines Binärbaums
- Aktivierung von p Prozessoren bzw. Broadcast eines Wertes an p Prozessoren in $\lceil \log p \rceil$ möglich.
- Reduktion von n Werten mit assoziativer binärer Operation $\oplus$ ist auf $\frac{n}{2}$ Prz. in $\Theta(\log n)$ Schritten möglich

* Präfixsummenberechnung (Scans)

Gegeben n Werte $a_0, \dots, a_{n-1}$ und eine binäre, assoziative Operation $\oplus$

Gesucht:
a_0
$a_0 \oplus a_1$
$a_0 \oplus a_1 \oplus a_2$
$\vdots$
$a_0 \oplus \dots \oplus a_{n-1}$

} n Werte

CREW-PRAM-Verfahren

globale Variablen: n, $A[0..(n-1)]$

Anfangsbedingung: $A[0..(n-1)]$ enthält Eingabewerte

Endbedingung: $A[i]$ enthält $A[0] \oplus \dots \oplus A[i]$.

```
begin
  spawn (P_1, ..., P_{n-1})                    | aktiviere P_1 bis P_{n-1}
  for all P_i where 1 ≤ i ≤ n-1 do
    for j=0 to ⌈log n⌉ - 1 do                  ← ein Schleifendurchlauf pro Takt
                  =2
      if i ≥ 2^j then
        A[i] ← A[i] + A[i - 2^j]
      fi
    od
  od
end
```

Bsp.

A	0	1	2	3	4	5	6
	3	1	0	4	2	6	5
	3	4	1	4	6	8	11
	3	4	4	8	7	12	17
					10	16	21

Aufwand: $n-1$ Proz., $\Theta(\log n)$ par. Schritte

Kosten: $\Theta(n \log n)$ } keine Kostenoptimalität
seq. Aufwand: $\Theta(n)$ }

Def.: Ein paralleler Algorithmus heißt kostenoptimal, wenn seine Kosten (# Proz. * par. Laufzeit) in derselben Komplexitätsklasse wie ein optimaler seq. Algorithmus liegen

Aufwand der Reduktion

par. Kosten: $\frac{n}{2}$ Proz. $\Theta(\log n)$
$\rightsquigarrow \Theta(n \log n)$ } nicht kostenoptimal
seq. Aufwand: $\Theta(n)$

Satz 1.2 (Brent, 1974)

Sei A ein paralleler Algorithmus mit Ausführungszeit t (# par. Schritte). Falls A m Operationen ausführt, so kann A mit p Prozessoren in der Zeit $t + \frac{m-t}{p}$ ausgeführt werden.

26.04.2007

Beweis: Sei s_i die Anzahl der Operationen, die im i-ten Schritt von A ausgeführt werden: $\sum_{i=1}^{t} s_i = m$

Mit p Prozessoren benötige wir $\lceil \frac{s_i}{p} \rceil$ Schritte zur Ausführung von s_i Operationen.

$$\Rightarrow \sum_{i=1}^{t} \lceil \frac{s_i}{p} \rceil \leq \sum_{i=1}^{t} \frac{s_i + p - 1}{p} = \sum_{i=1}^{t} \left(\frac{s_i - 1}{p} + \overset{1}{\underset{m}{\frac{p}{p}}} \right)$$

$$= t + \sum_{i=1}^{t} \frac{s_i - 1}{p}$$

$$= t + \frac{m-t}{p} \qquad \oplus$$

Verbesserung der Reduktion / Präfixsummenberechnung durch die Reduktion der #Proz.

Reduktion

#Ops im seq. Fall: $n-1$

im par. Fall: $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots + 1$

$$= \sum_{i=0}^{\log n - 1} 2^i = n - 1$$

Kostenopt. falls $\Theta(p * \log n) = \Theta(n)$

$$\rightarrow p = \Theta\left(\frac{n}{\log n}\right)$$

Zeitkomplexität mit $p = \Theta\left(\frac{n}{\log n}\right)$ Proz. nach **Brent** $\Theta\left(\log n + \frac{n-1-\log n}{\frac{n}{\log n}}\right)$

$$= \Theta\left(\log n + \log n - \frac{\log n}{n} - \frac{\log^2 n}{n}\right)$$

$$= \Theta(\log n)$$

$\Rightarrow$ Die Komplexität des par. Alg. wird durch die Reduktion der Proz.-Zahl auf $\Theta\left(\frac{n}{\log n}\right)$ Proz. nicht verändert.

$\Rightarrow$ Kostenopt. der Reduktion auf $\Theta\left(\frac{n}{\log n}\right)$ Proz.

Bsp. $n=16$ $\rightsquigarrow$ Wähle $\frac{n}{\log n} = 4$ Proz. statt $\frac{n}{2} = 8$ Proz.

$\Rightarrow$ 1 zusätzlicher Schritt bei nur 4 Proz.

Präfixsummenberechnung

Ops im seq. Fall: $n-1$

im par. Fall:

$$\sum_{j=0}^{\lceil \log n \rceil - 1} (n - 2^j) = n \lceil \log n \rceil - \left(2^{\lceil \log n \rceil} - 1\right)$$
$$= \Theta(n \log n)$$

$\rightsquigarrow$ mehr Ops als im seq. Fall

$\rightsquigarrow$ Reduktion der # Proz. führt nicht zu kostenoptimalem Verfahren

Besser: Verwende bei $p < n-1$ Proz. für Teilaufgabenbereiche opt. seq. Verfahren und kombiniere anschließend Werte in geeigneter Weise.

Berechnung der Präfixsummen von n Werten mit $p < n-1$ Proz.

1.) Aufteilen der n Werte in p Teilbereiche mit nicht mehr als $\lceil \frac{n}{p} \rceil$ Elementen.

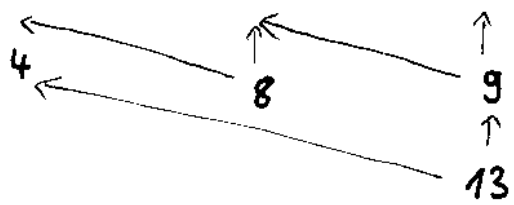
Bsp. $n=14$, $\lceil \log n \rceil = 4$, $p=4$

A 2 1 4 -3 | 0 -2 5 1 | -1 2 4 0 | 3 7

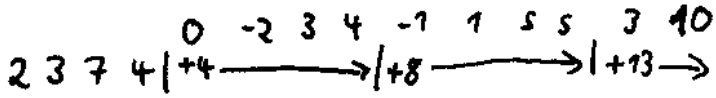
2.) Die ersten p-1 Proz. berechnen opt. seq. Verfahren auf ihren Teilbereichen $\Rightarrow \lceil \frac{n}{p} \rceil - 1$ Schritte

2 3 7 4 | 0 -2 3 4 | -1 1 5 5 | 3 10

3.) Die ersten p-1 Proz. berechnen die Präfixsumme der Gesamtsummen ihrer Teilbereiche mit dem par. Verfahren $\Rightarrow \lceil \log (p-1) \rceil$ Schritte



4.) Die letzten $p-1$ Proz. addieren die Gesamtsumme der Vorgängerblöcke auf die Elemente ihrer Teilbereiche
$\Rightarrow \lceil \frac{n}{p} \rceil$ Additionen



Gesamtaufwand:
$$\lceil \tfrac{n}{p} \rceil - 1 + \lceil \log(p-1) \rceil + \lceil \tfrac{n}{p} \rceil \in \Theta\left(\tfrac{n}{p} + \log p\right)$$

Gesamtzahl der Ops:
$$p \cdot \left(\lceil \tfrac{n}{p} \rceil - 1\right) + \Theta(p \log p) + p \cdot \lceil \tfrac{n}{p} \rceil \in \Theta(n + p \log p)$$

Mit $p = \frac{n}{\log n}$ Proz. ergibt sich der Gesamtaufwand zu: $\Theta(\log n)$
Gesamtzahl der Ops: $\Theta(n)$
$\Rightarrow$ Das Verfahren ist kostenoptimal

Einordnung der PRAM-Modelle in die Komplexitätstheorie

offenes Problem: $P = NP$?

- polynomiell lösbare Probleme
- nichtdeterministisch polyn. lösbare Probleme

NP-vollständige Probleme:
Teilklasse aus NP, für die gilt:
Wenn für ein NP-vollständiges Problem ein polynomielles Verfahren gefunden wird, dann folgt $P = NP$.

Nach Nick Pippenger wurde die folgende Nick Class (NC) benannt.

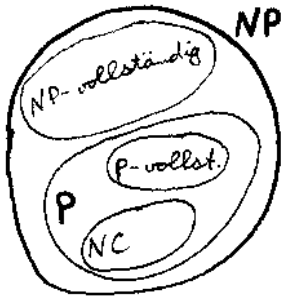
Def. Ein Problem gehört zu der Nick Class (NC), falls ein paralleler Lösungsalgorithmus A existiert, der für geeignete k_1, k_2 folgende Bedingungen erfüllt:

a) # Proz. ist polynomial in Problemgröße beschränkt, d.h. $p(n) \in O(n^{k_1})$

b) Laufzeit von A ist polylogarithmisch in Problemgröße, d.h. $T(n) \in O(\log^{k_2} n)$

Für Probleme in NC erhält man durch Sequentialisierung des par. Lösungsverfahrens lin. Verfahren mit polynomialer Laufzeit, d.h. $NC = P$

Def. Ein Problem L in P heißt P-vollständig, falls jedes andere Problem in P in polylogarithmischer paralleler Zeit mit einer PRAM mit polynomialer Anzahl von Proz. in L transformiert werden kann.



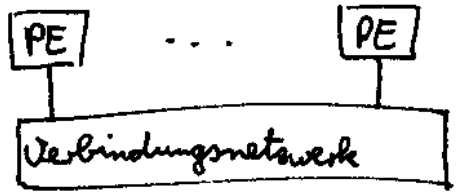
Bsp. für P-vollst. Probleme
- Tiefensuche in Graphen
- Best. max. Flüsse in Graphen

Parallele Berechnungsthese (nicht bewiesen)

Die Klasse der Probleme, die von einer PRAM in Zeit $T(n)^{O(1)}$ lösbar ist, entspricht der Klasse von Problemen, die durch eine RAM mit Speicherplatz $T(n)^{O(1)}$ lösbar ist, sofern $T(n) \geq \log n$.

1.2 Rechnermodelle mit verteiltem Speicher

bestimmendes Element: Verbindungsnetzwerk
feste Knoten-zu-Knoten-Verbindungen



PE = Prozessoren mit privaten Speicher

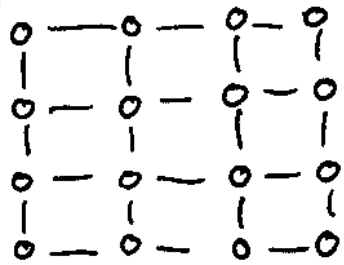
Bewertungskriterien für Verbindungsstrukturen

- Durchmesser ≙ längster Abstand zwischen zwei Knoten
- Halbierungsbreite ≙ minimale Anzahl von Verbindungslinien, die durchtrennt werden muss, um das Netzwerk in zwei Teile zu zerlegen
- Verbindungsgrad: Anzahl der Verbindungen pro Knoten

02.05.2007

Gitternetzwerke (mesh networks)
regelmäßiger q-dimensionaler Verband mit k Knoten pro Dimension
↝ k^q Knoten

z.B. $q=2,\ k=4$

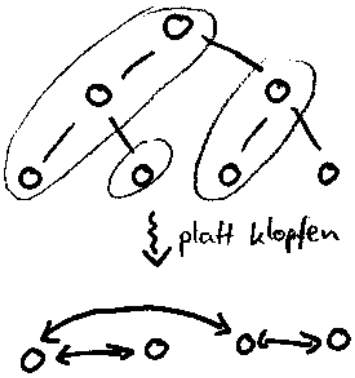


Torus: zusätzlich 'wrap-around' Verbindungen der Randknoten
→ Ringe pro Dimension

Kennzahlen	Gitter	Torus
Durchmesser	$q \cdot (k-1)$	$q \cdot \lfloor \frac{k}{2} \rfloor$
Halbierungsbreite	k^{q-1}	$2\,k^{q-1}$
Verbindungsgrad	$2q$	$2q$

Aufwand für Reduktion / Broadcast entspricht dem Durchmesser des Gitters

Binärbäume vollständig, Tiefe k $\sim 2^{k+1}-1$ Knoten und 2^k Blattknoten



$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

Kennzahlen

Durchmesser: $2k$

Halbierungsbreite: 1

Verbindungsgrad: 3

Aufwand Reduktion / Broadcast: k

Hypercube

2^k Knoten sind in einem Hypercube der Dimension k wie folgt vernetzt:

Die Knoten werden durch binäre Adressen identifiziert

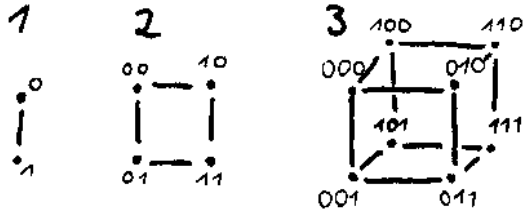
$b_{k-1} \cdots b_0 \quad (b_i \in \{0,1\},\ 0 \le i \le k-1)$

Je zwei Knoten sind genau dann verbunden, wenn sich ihre Adresse in genau einer Bitposition unterscheiden.

induktiver Aufbau

Dimension $k = 0$ 1 2 3

Hypercube



Kennzahlen

Durchmesser: k

Halbierungsbreite: 2^{k-1}

Verbindungsgrad: k

Hypercube-Reduktion

Pseudocode: Parameter k = Dimension des Hypercubes

```
local sum
begin
for j := 0 to k-1 do
    for all P_i where 0 ≤ i < 2^(k-j) do
        if bit_j (i) = 1
        then send sum to P_(i-2^j)
        else receive tmp from P_(i+2^j)
             sum := sum + tmp
        fi
    od
od
end
```

Broadcast im Hypercube

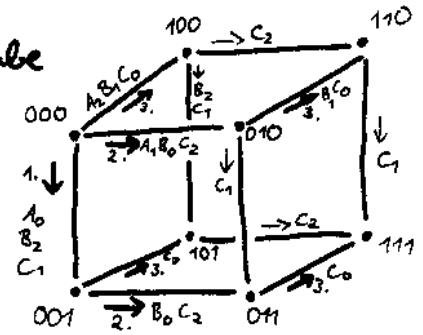
→ umgekehrte Reduktion

Optimierung von Johnsson und Ho (1989)
für große Nachrichten:

Beobachtung: Das Standard-Broadcast-Verfahren benutzt maximal 2^{k-1} Verbindungsleitungen gleichzeitig bei $k \cdot 2^{k-1}$ insgesamt vorhandenen.

03.05.2007

Hypercube



1., 2., 3.: Standard-Broadcast

Idee: Nutze alle Verbindungsleitungen durch Aufspalten der Broadcast-Nachricht in k Teile und Durchführung des Broadcast in verschiedenen Dimensionen.

Der Algorithmus nutzt $k = \log p$ kantendisjunkte spannende Bäume des Hypercubes mit identischer Wurzel. Bei einer Nachricht der Größe M werden Nachrichtenteile der Größe $\frac{M}{\log p}$ parallel verschickt, so dass der Broadcast insgesamt $\frac{M}{\log p} * \log p = M$ Zeit benötigt.

Shuffle-Exchange-Netzwerke

2^k Knoten, die nummeriert sind von 0 bis $2^k - 1$,

2 Verbindungsarten:

- Exchange-Verbindungen: $\leftrightarrow$ letztes Bit invertiert

  $b_{k-1} \dots b_1 0 \leftrightarrow b_{k-1} \dots b_1 1$

- Shuffle-Verbindungen: $\rightarrow$ binär rotiert

  $b_{k-1} \dots b_1 b_0 \rightarrow b_{k-2} \dots b_1 b_0 b_{k-1}$

Bsp. $k = 3$

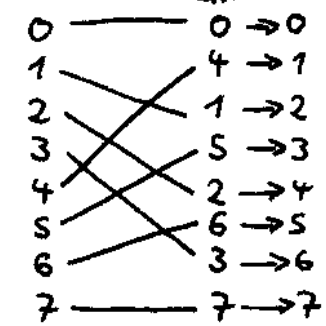
Kennzahlen

Durchmesser: $\underbrace{k}_{\text{Exchange}} + \underbrace{k-1}_{\text{Shuffle}} = 2k-1$

Halbierungsbreite: 4

Verbindungsgrad: 4

perfektes Mischen von Karten:



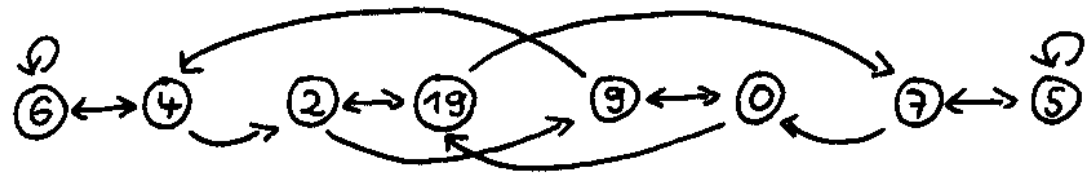
„perfect shuffle"

Shuffle-Exchange-Reduktion

Ansatz: Reduktion von zwei Zwischenergebnissen pro Schritt und Knoten

→ logarithmisch viele Schritte insgesamt

Bsp.



Schritt								
1.	-3	-3	-4	-4	9	9	24	24
2.	6	6	6	6	20	20	20	20
3.	26	...						

allgem. Pseudocode Parameter: $n = 2^k$

```
local val, temp
begin
    for j := 0 to k-1 do
        for all P_i where 0 ≤ i < n do
            send val to P_←i        } „shuffle"
            receive val from P_→j   }
            send val to P_<i>       } „exchange"
            receive temp from P_<i> }
            val := val ⊕ temp
        od
    od
end
```

wobei: $\overleftarrow{i}$, $\overrightarrow{i}$: i binär rotiert

$<i>$: letztes Bit invertiert

1.3 Bewertung paralleler Algorithmen

Analyse - Zeitkomplexität
- Kosten (# Proz. * par. Zeit)
- Beschleunigung
- Effizienz
- Skalierbarkeit

$$\text{Beschleunigung (speedup)} = \frac{\text{optimale seq. Ausführungszeit}}{\text{parallele Ausführungszeit}}$$

$$\text{Effizienz (efficiency)} = \frac{\text{optim.}}{\text{parallele Kosten}}$$

$$= \frac{\text{Beschleunigung}}{\text{\# Proz.}}$$

Bezeichne n die Problemgröße

p die Anzahl der Proz.

$T_p(n)$ die Ausführungszeit eines Algorithmus auf p Prozessoren bei Problemgröße n

$S_p(n)$ die Beschleunigung

$E_p(n)$ die Effizienz

$C_p(n)$ die Kosten

Dann gilt:

$$C_p(n) = p * T_p(n)$$

$$S_p(n) = \frac{T_1^{opt}(n)}{T_p(n)}, \quad \text{i.a. } 0 \leq S_p(n) \leq p$$

$$E_p(n) = \frac{S_p(n)}{p}, \quad \text{also: } 0 \leq E_p(n) \leq 1$$

Sei $\sigma(n)$ der Zeitanteil des seq. Alg., der nicht parallelisiert werden kann.

Sei $\varphi(n)$ der Zeitanteil, der parallelisiert ist und $K_p(n)$ der Mehraufwand für die parallele Ausführung auf p Proz.

Dann gilt:

$$T_1(n) = \sigma(n) + \varphi(n)$$

$$T_p(n) \geq \sigma(n) + \frac{\varphi(n)}{p} + K_p(n)$$

$$\curvearrowright \quad S_p(n) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p} + \underbrace{K_p(n)}_{\geq 0}} \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p}}$$

$$E_p(n) \leq \frac{\sigma(n) + \varphi(n)}{p\,\sigma(n) + \varphi(n) + p K_p(n)}$$

Amdahls Gesetz

Sei $f = \frac{\sigma(n)}{\sigma(n) + \varphi(n)}$ der nicht par. Anteil an seq. Ausführungszeit

Dann gilt: $\sigma(n) = f \cdot T_1(n) = f(\sigma(n) + \varphi(n))$

$$\varphi(n) = (1-f)\, T_1(n) = (1-f)(\sigma(n) + \varphi(n))$$

Dann gilt: $S_p(n) \leq \dfrac{1}{\frac{\sigma(n)}{\sigma(n)+\varphi(n)} + \frac{1}{p}\frac{\varphi(n)}{\sigma(n)+\varphi(n)}}$

$$= \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

Bsp. Schon bei $f = 25\%$ ist der Speedup durch 4 beschränkt.

Gesetz von Gustafson und Barsis

Sei $s = \dfrac{\sigma(n)}{\sigma(n) + \frac{\varphi(n)}{p}}$ } $T_p(n)$ ohne Mehraufwand

seq. Anteil an paralleler Berechnung.

$$\rightsquigarrow 1-s = \frac{\frac{\varphi(n)}{p}}{\sigma(n) + \frac{\varphi(n)}{p}}$$

$$\rightsquigarrow \sigma(n) = s * \left(\sigma(n) + \frac{\varphi(n)}{p}\right)$$

$$\varphi(n) = (1-s) * p * \left(\sigma(n) + \frac{\varphi(n)}{p}\right)$$

$$\rightsquigarrow S_p(n) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p}} = s + (1-s)p = p + (1-p)s$$

Bsp. $p = 64$

$T_p(n) = 220\,sec$

$s = 5\%$

$\rightsquigarrow S_p(n) \leq 64 + (1-64) * 0{,}05 = \underline{\underline{60{,}85}}$

09.05.2007

Bewertung par. Alg.

Beschleunigung $S_p(n) = \dfrac{T_1(n)}{T_p(n)}$

Effizienz $E_p(n) = \dfrac{S_p(n)}{p} = \dfrac{T_1(n)}{p \cdot T_p(n)}$

Unterscheide:

$\sigma(n)$ – seq. Laufzeit, die nicht parallelisierbar ist

$\varphi(n)$ – seq. Laufzeit, die parallelisierbar ist

$\rightsquigarrow T_1(n) = \sigma(n) + \varphi(n)$

$T_p(n) \geq \sigma(n) + \varphi(n)/p$

Amdahls Gesetz

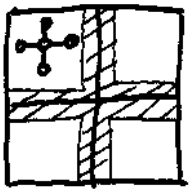
Sei $f_n := \dfrac{\sigma(n)}{T_1(n)} = \dfrac{\sigma(n)}{\sigma(n) + \varphi(n)}$

$$S_p(n) \leq \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

Amdahl-Effekt:

Bei wachsender Problemgröße wächst die Beschleunigung bei gleichbleibender Prozessorzahl.

Bsp.



Berechnungskomplexität $O(n^2)$

Kommunikationskomplexität $O(n)$

Die Karp-Flatt-Metrik (CACM 33(5), May 1990)

Auflösung von $S_p(n) \leq \frac{1}{f + \frac{(1-f)}{p}}$

nach f liefert: $f + \frac{(1-f)}{p} \leq \frac{1}{S_p(n)}$

$$\Leftrightarrow \quad f\left(1 - \frac{1}{p}\right) \leq \frac{1}{S_p(n)} - \frac{1}{p}$$

$$\Leftrightarrow \quad \boxed{f = \frac{\frac{1}{S_p(n)} - \frac{1}{p}}{1 - \frac{1}{p}}}$$

Der Term $\frac{\frac{1}{S_p(n)} - \frac{1}{p}}{1 - \frac{1}{p}}$ wird als

experimentell ermittelter serieller Anteil bezeichnet.

Da f von p unabhängig ist, sollte e möglichst konstant sein. In e geht aber auch der Mehraufwand für die Parallelverarbeitung ein, was zu wachsendem e führen kann.

⇒ Über die Analyse der Entwicklung von e bei steigender Prozessorzahl und gleichbleibender Problemgröße können Rückschlüsse auf die Güte der Parallelisierung gezogen werden.

Bewertung der Skalierbarkeit

10.05.2007

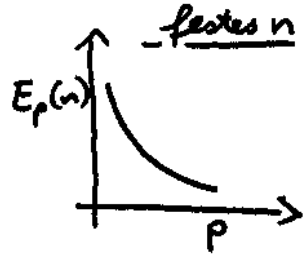
Ziel: gleichbleibende Effizienz bei steigender Prozessorzahl durch Vergrößerung der Problemgröße

Overhead-Funktion

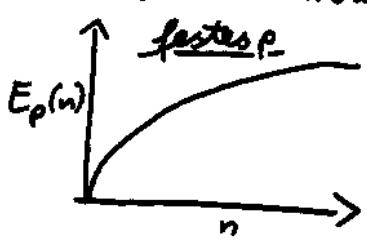
$$O_p(n) = p * T_p(n) - T_1(n)$$
$$= p \cdot \sigma(n) + \varphi(n) + p \cdot K_p(n) - \sigma(n) - \varphi(n)$$
$$= (p-1)\sigma(n) + p \cdot K_p(n)$$

Es gilt:
$$E_p(n) = \frac{T_1(n)}{p \cdot T_p(n)} = \frac{T_1(n)}{T_1(n) + O_p(n)} = \frac{1}{1 + \frac{O_p(n)}{T_1(n)}}$$

Bei wachsender Prozessorzahl sinkt $E_p(n)$ bei fester Problemgröße.



bei skalierbarem System



Aus $E_p(n) = \frac{1}{1 + \frac{O_p(n)}{T_1(n)}}$ folgt: $\frac{O_p(n)}{T_1(n)} = \frac{1}{E_p(n)} - 1 = \underbrace{\frac{1 - E_p(n)}{E_p(n)}}$

konstant bei gleichbleibender Effizienz

$$\Leftrightarrow T_1(n) = \underbrace{\frac{E_p(n)}{1 - E_p(n)}}_{=c} \cdot O_p(n)$$

$\Rightarrow$ Isoeffizienzrelation f mit $n = f(p)$

Bsp. Addition von n Werten auf p Prozessoren

$T_1(n) = O(n)$; $T_p(n) = O(\frac{n}{p} + \log p)$

$O_p(n) = O(p \cdot \log p)$

$\Rightarrow n = c \cdot p \log p$

2. Paralleles Sortieren

interne Sortierverfahren, d.h. alle zu sortierenden Daten liegen im Hauptspeicher.

2.1 Ein CRCW-PRAM-Verfahren mit konstantem Zeitaufwand

n^2 Prozessoren einer CRCW-PRAM können n Elemente in konstanter Zeit sortieren, falls

- die Aktivierungszeit der n^2 Prozessoren ($O(\log n)$) vernachlässigt wird

und

- beim CW die Summe der Werte geschrieben wird.

Ansatz: enumeration sort (rank sort)

Vgl. jedes Element mit jedem anderen und zähle die Anzahl der kleineren Elemente → Position in sortierter Liste

Algorithmus:

Parameter: n = # Elemente

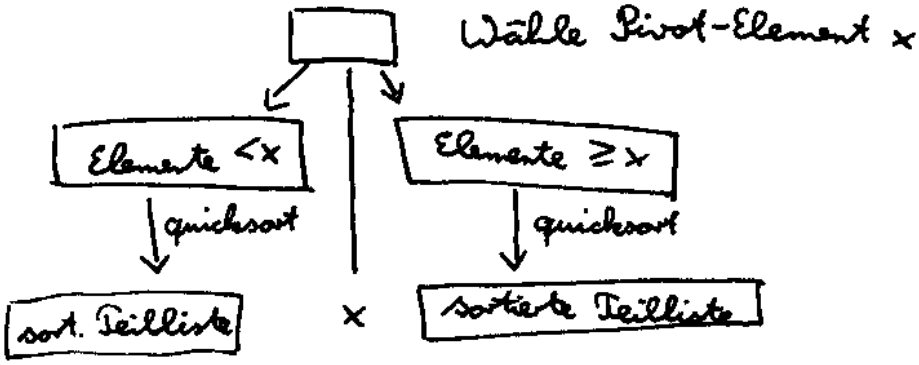
Globale Var.: a [0, ..., (n-1)] Elemente
pos [0, ..., (n-1)] Positionen in sortierter Liste
sorted [0, ..., (n-1)] sortiertes Feld

```
begin
  spawn P_{i,j} with 0 ≤ i,j < n
  for all P_{i,j} with 0 ≤ i,j < n do
O(1) { pos[i] := 0;
       if a[i] > a[j] or (a[i] = a[j] and i < j) then
         pos[i] := 1      ⇐ CW mit Addition
       fi
  od
  for all P_{i,0}, 0 ≤ i < n do
      sorted[pos[i]] := a[i]
  od
end
```

2.2 Quicksort-basierte Verfahren

Quicksort hat im Mittel opt. seq. Aufwand $O(n \log n)$

rekursives divide-et-impera-Verfahren



Probleme: - Wahl des Pivotelements

- sequentielle Anfangsverfahren → Amdahl

2.2.1 Hyperquicksort

Paralleles Quicksort auf Hypercubes

Initialisierung: Ausgangsliste mit n Werten wird gleichmäßig auf die 2^k Knoten eines k-dimensionalen Hypercubes verteilt.

→ Jeder Knoten enthält $\frac{n}{2^k}$ Werte

Ziel: 1.) Die Teillisten auf allen Prozessoren sind sortiert.

2.) Alle Elemente auf P_i sind ≤ allen Elementen auf P_{i+1} $(0 \le i \le 2^k - 2)$

Eine gleichmäßige Verteilung aller Elemente (wie zu Beginn) ist nicht erforderlich.

Verfahren:

1.) Jeder Prozess sortiert die ihm zugeordneten Teillisten mit einem optimalen sequentiellen Sortierverfahren
→ Ziel 1 erfüllt.

2.) rekursive divide-and-conquer-Schritte

for $d = k$ to 2 do

Zerlege den Teilhypercube der Dim. d in zwei Hypercubes der Dim. $d-1$

Sammle in einem Teilhypercube die „kleineren" Elemente und im anderen die „größeren" Elemente (im Vergleich zu einem Pivotelement), indem jeder Prozess mit seinem direkten Nachbarn im anderen Teilhypercube Werte austauscht.

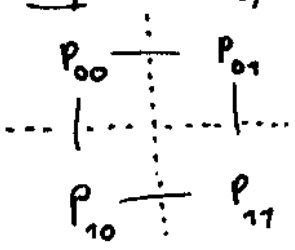
Festlegung des Pivotelements: mittleres Element eines ausgezeichneten Prozessors im Hypercube der Dim. d, Broadcast dieses Elementes an alle Prozessoren des Teilhypercubes.

Jeder Proz. führt split- und merge-Schritte aus:

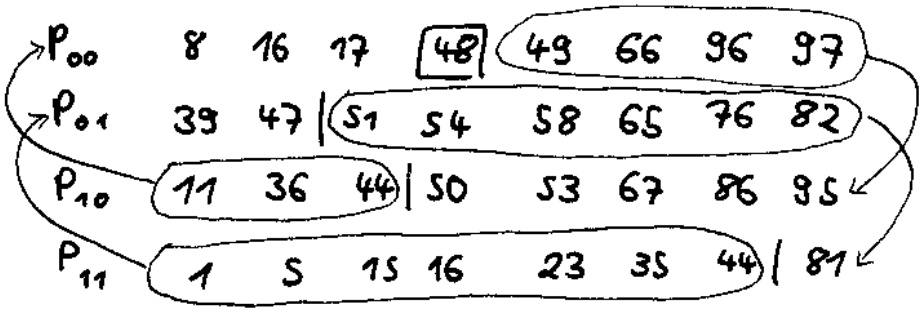
- split: Aufteilen gemäß Pivotelement
  Verschicken einer „Hälfte" an Partner
- merge: Empfang von Werten vom Partner und Mischen mit eigener „Hälfte"

Nach k solchen Schritten ist das Ziel 2 erfüllt.

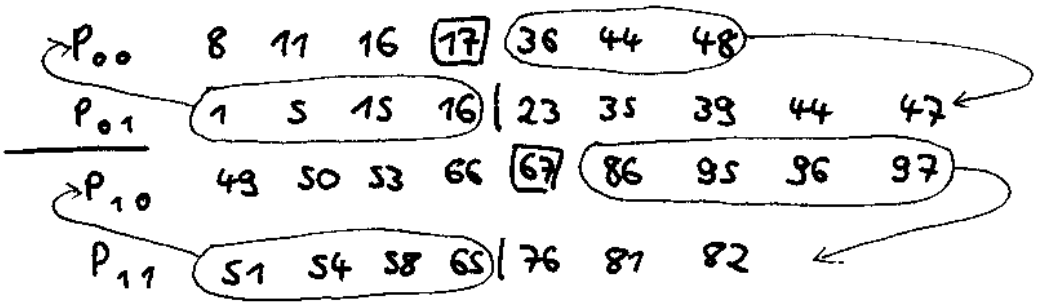
Bsp. $n = 32$, $k = 2$



nach lokalem Sortieren



nach dem ersten split-& merge-Schritt



nach dem zweiten split-& merge-Schritt

P_{00}	1	5	8	11	15	16	16	17
P_{01}	23	35	36	39	44	44	47	48
P_{10}	49	50	51	53	54	58	65	66	67
P_{11}	76	81	82	86	95	96	97		

→ Ziel erreicht.

Analyse

1. Schritt: $\Theta\left(\frac{n}{2^k} \cdot \log \frac{n}{2^k}\right) = \Theta\left(\frac{n}{2^k}(\log n - k)\right)$

Schritte 2.1 ... 2.k: Aufteilung in Teilhypercubes

$k \underset{2.1}{\rightsquigarrow} k-1 \rightsquigarrow \dots \underset{2.k}{\rightsquigarrow} 1$

im besten Fall: gleichmäßige Aufteilung der Elemente auf Proz.
$\Theta\left(\frac{n}{2^k} \cdot k\right)$ parallele Vergleichsschritte $\Rightarrow O\left(\frac{n \log n}{2^k}\right)$
parallele Rechenschritte

Kommunikationsoverhead:

Broadcast der Pivotelemente: $\leq k$
Elementaustausch: $\frac{n}{2^{k+1}}$
$\Big\} O(k^2 + \frac{n-k}{2^{k+1}})$
$\rightsquigarrow O\left(\frac{n \log p}{p}\right)$

Isoeffizienzrelation:

$$T_1(n) = C \cdot O_p(n)$$
$$n \cdot \log n = C \cdot \frac{n \cdot \log p}{p} \cdot p$$
$$\log n = c \cdot \log p$$
$$\boxed{n = p^c} \Rightarrow \text{schlechte Skalierbarkeit falls } c \geq 2$$

2.2.1 Hyperquicksort

1. lokale Sortierung von p Teillisten mit $\frac{n}{p}$ Elementen
2. Auswahl des Pivotelementes durch P_0
   und Broadcast an alle Knoten
3. parallele Aufteilung aller sortierten Teillisten
   gemäß Pivotelement
   $P_{0b_{k-2}\dots b_0}$ schickt seine "großen" Elemente an $P_{1b_{k-2}\dots b_0}$

   $P_{1b_{k-2}\dots b_0}$ schickt seine "kleinen" Elemente an $P_{0b_{k-2}\dots b_0}$
4. Alle Prozesse mischen empfangene Teillisten mit
   eigenem Rest

Fahre fort mit Schritt 2 in beiden Teilhypercubes
$\langle P_{0b_{k-2}\dots b_0}\rangle$ und $\langle P_{1b_{k-2}\dots b_0}\rangle$

Nach $k = \log p$ Iterationen liegt die Gesamtliste
verteilt sortiert vor.

Aufwand: $O\left(\frac{n \log n}{p} + n \log p\right)$

Nachteile von Hyperquicksort

- kritische Pivotwahl lokal durch ausgezeichneten Prozessor
- hoher Kommunikationsoverhead
  "Elemente wandern über evtl. mehrere Zwischenprozesse
  zur Zielposition"

2.2.2 PSRS-Algorithmus (Lit '92)

(Parallel Sorting by Regular Sampling)

1. bessere Pivotauswahl
2. Elemente werden höchstens einmal kommuniziert

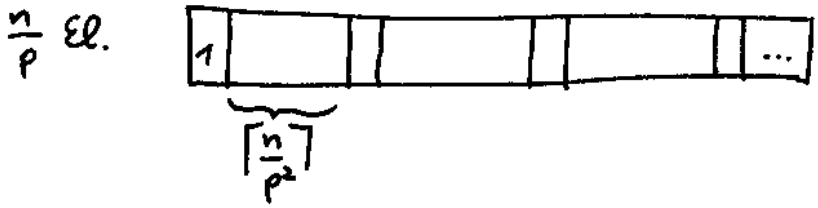
4 Phasen

Phase I: sequentielles Quicksort auf Teilsegmenten mit bis zu $\lceil \frac{n}{p} \rceil$ Elementen der zu sortierenden Liste

→ p sortierte Teillisten mit $\leq \lceil \frac{n}{p} \rceil$ Elementen

→ Auswahl von p Elementen an den p Positionen

$1,\ \lceil \frac{n}{p^2} \rceil + 1,\ 2 \cdot \lceil \frac{n}{p^2} \rceil + 1,\ \ldots,\ (p-1) \cdot \lceil \frac{n}{p^2} \rceil + 1$



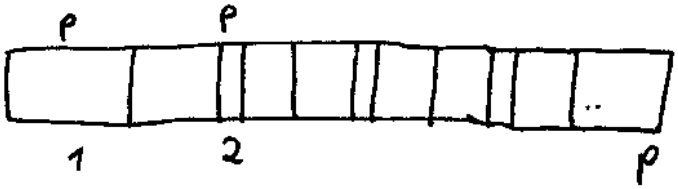
⇒ gleichmäßige Stichprobe aus sortierter Teilliste

Phase II: Ein Prozessor sammelt alle p Proben mit je p Elementen und mischt diese zu einer sortierten Gesamtprobe

→ 1 sortierte Teilliste mit p^2 Elementen

→ Auswahl von $p-1$ Pivotelementen aus dieser Probe und zwar an den Stellen

$p + \lfloor \frac{p}{2} \rfloor,\ 2p + \lfloor \frac{p}{2} \rfloor,\ \ldots,\ (p-1)p + \lfloor \frac{p}{2} \rfloor$



und Broadcast dieser $p-1$ Pivotelemente an alle Prozesse.

Phase III: Jeder Prozessor teilt seine Teilliste mit $\lceil \frac{n}{p} \rceil$ El. in p Teillisten gemäß den p-1 Pivotel. auf.
Jeder Prozessor i behält seine i-te Partition und versendet die j-te Partition an Prozessor j für $1 \leq j \leq p$, $j \neq i$

Phase IV: Jeder Prozessor mischt die ihm zugewiesenen p sortierten Teillisten zu einer sortierten Teilliste.

Die Konkatenation aller sortierten Teillisten ergibt die sortierte Gesamtliste.

Bsp. $p=3$, $n=27$

nach Phase I:

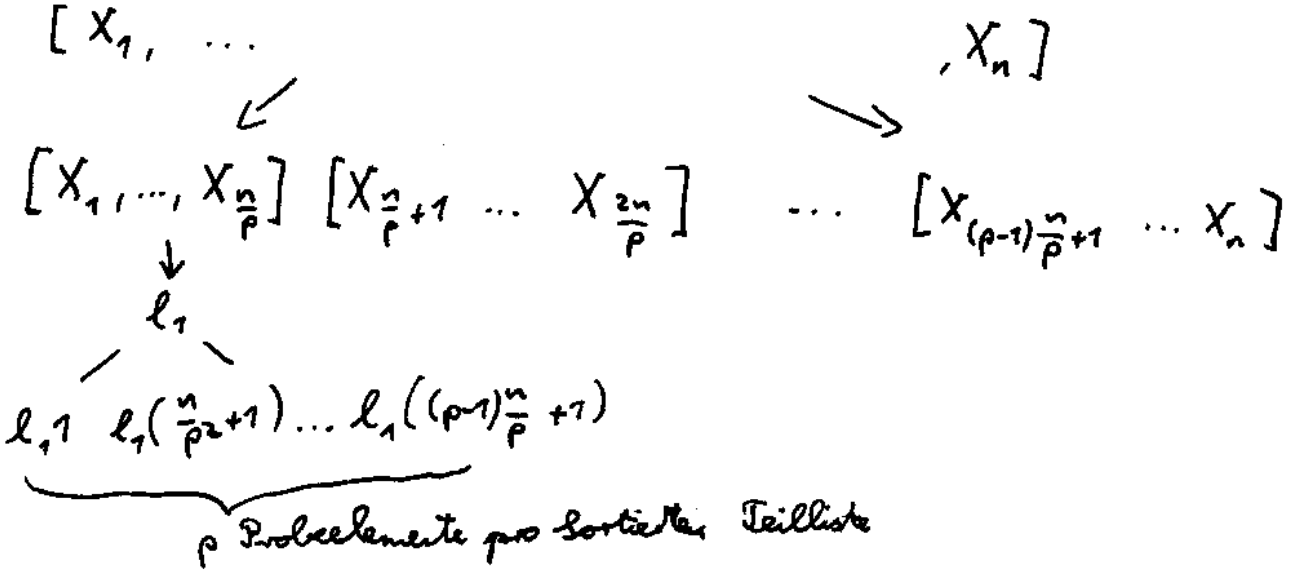
p 1:	6	14	15	33	46	48	69	91	93
p 2:	12	21	36	40	54	61	69	89	97
p 3:	20	27	32	33	53	58	69	84	97

Phase II:

6 12 20 33 39 40 69 72 72

23.05.2007

PSRS



Beobachtung: Jedes Probeelement repräsentiert $\frac{n}{p^2}$ Elemente einer sortierten Teilliste, die größer oder gleich dem Probeelement sind.

p^2 Probeelemente sortieren

$[Y_1, \ldots, Y_{p^2}]$

↓ Auswahl von $p-1$ Pivotelementen

$\{Y_{p+\frac{p}{2}}, \ldots, Y_{(p-1)p+\frac{p}{2}}\}$

↙ ↓ ↘ Broadcast an alle Proz.

Aufteilen von l_j in p Teilsegmente gemäß Pivotelementen

Mischen der Teilsegmente

Analyse des Verfahrens:

vereinfachende Annahmen:

p Prozessoren, p geradzahlig,

$n = p^2 k$ paarweise verschiedene El. mit $k > 1$

Phase I: paralleles seq. Sortieren der Teillisten der Länge $\frac{n}{p}$

$\rightarrow O(\frac{n}{p} \log \frac{n}{p})$

Phase II: Sortieren der Listenprobe mit p^2 Elementen

$\rightarrow O(p^2 \log p)$

Phase III: Aufteilen in p Partitionen

$\rightarrow O(\frac{n}{p})$

Phase IV: Jeder Prozess mischt p sortierte Teillisten

Satz: Jeder Prozess hat höchstens $\frac{2n}{p}$ Elemente zu mischen.

Beweis: später

$\xrightarrow{\text{ÜB}} O(\frac{n}{p} \log p)$

Gesamtkomplexität

$$O\left(\frac{n}{p}\log\frac{n}{p} + p^2 \log p + \frac{n}{p} + \frac{n}{p}\log p\right)$$

$$= O\left(\frac{n}{p}\log n + p^2 \log p\right)$$

Falls $n \geq p^3$ dominiert der erste Term so dass die Kompl. $O\left(\frac{n \log n}{p}\right)$ optimal wird.

Der in Phase IV verwendete Satz ist ein Korollar des folgenden Theorems:

Bezeichnet ϕ_i die Anzahl der Listenelemente, die in Phase IV von Prozess i gemischt werden müssen, so gilt:

$$\max_{1 \leq i \leq p} \phi_i \leq \frac{2n}{p} - \frac{n}{p^2} - p + 1 \leq \frac{2n}{p}$$

Notation $X = \{X_1, \ldots, X_n\}$ zu sortierende Liste

$Y = \{Y_1, \ldots, Y_{p^2}\}$ sortierte Listenprobe

$N_X(\text{cond})$ Anzahl der El. von X, die die Bedingung cond erfüllen

z.B. $N_X(\leq Y_2)$

Hilfslemmata

① Sei $1 \le i \le p$

$$N_X\left(\le \underbrace{Y_{(i-1)p+\frac{p}{2}}}_{(i-1)\text{te Pivotelement}}\right) \geq \begin{cases} \frac{p}{2} & \text{falls } i=1 \\ \frac{n}{p^2}\left((i-1)p-\frac{p}{2}\right)+p & \text{falls } i>1 \end{cases}$$

Beweis:

$i=1$ $Y_{(i-1)p+\frac{p}{2}} = Y_{\frac{p}{2}}$ Die $\frac{p}{2}$ Elemente der sortierten Probeliste $\langle Y_j \mid 1 \le j \le \frac{p}{2} \rangle$ sind $\le Y_{\frac{p}{2}}$ und Elemente aus X

$i>1$ $(i-1)p+\frac{p}{2} > p$, d.h. in $\langle Y_\ell \mid 1 \le \ell \le (i-1)p+\frac{p}{2} \rangle$ sind Proben Y_j und Y_k vom selben Prozess.

Sei $j<k$. Alle $\frac{n}{p^2}$ Werte, die durch Y_j repräsentiert werden, sind $\le Y_k \le Y_{(i-1)}$

Für $(i-1)p+\frac{p}{2}-p$ Werte Y_j gibt es ein Y_k mit $Y_k > Y_j$ und Y_k kommt vom selben Prozessor. Für die p größeren Werte Y_k gilt ebenfalls, dass sie $\le Y_{(i-1)p+\frac{p}{2}}$ sind, d.h.

$$N_X\left(\le Y_{(i-1)p+\frac{p}{2}}\right) \geq \frac{n}{p^2}\underbrace{\left((i-1)p-\frac{p}{2}\right)}_{\#Y_j} + \underbrace{p}_{\#Y_k}$$

$$\max_{1 \le i \le p} \phi_i \le \frac{2n}{p} - \frac{n}{p^2} - p + 1$$

Hilfslemmata

1. $$N_X\left(\le Y_{(i-1)p+\frac{p}{2}}\right) \ge \begin{cases} \frac{p}{2} & \text{falls } i=1 \\ \frac{n}{p^2}\left((i-1)p - \frac{p}{2}\right) + p & \text{falls } i>1 \end{cases}$$

2. $$N_X\left(> \underbrace{Y_{ip+\frac{p}{2}}}_{\text{i-tes Pivotelement}}\right) \ge \frac{n}{p^2}\left((p-i)p - \frac{p}{2} + 1\right) - 1$$

Beweis: In der Probeliste der Y_i gibt es

$p^2 - \left(p + \frac{p}{2}\right) = (p-i)p - \frac{p}{2}$ Werte $> Y_{ip+\frac{p}{2}}$

und je $\frac{n}{p^2}$ Werte aus X, die durch diese repräsentiert werden.

Hinzu kommen die $\frac{n}{p^2} - 1$ Werte $> Y_{ip+\frac{p}{2}}$, die durch $Y_{ip+\frac{p}{2}}$ repräsentiert werden ($Y_{ip+\frac{p}{2}}$ ausgenommen).

somit: $N_X\left(> Y_{ip+\frac{p}{2}}\right) \ge \frac{n}{p^2}\left((p-i)p - \frac{p}{2}\right) + \frac{n}{p^2} - 1$ ⊕

Beweis des Theorems:

3 Fälle: $i=1$, $i=p$, $1<i<p$

1. Fall $i=1$:

Alle Elemente, die von Prozess 1 gemischt werden, sind $\le Y_{p+\frac{p}{2}}$

Mit Lemma 2 folgt: $N_X\left(> Y_{p+\frac{p}{2}}\right) \ge \frac{n}{p^2}\left(p^2 - \frac{3}{2}p + 1\right) - 1$

$$= n - \frac{3}{2}\frac{n}{p} + \frac{n}{p^2} - 1$$

$$\phi_1 = n - N_X\left(> Y_{p+\frac{p}{2}}\right)$$

$$\leq \frac{3}{2}\frac{n}{p} - \frac{n}{p^2} + 1$$

$$\overset{!}{\leq} \frac{2n}{p} - \frac{n}{p^2} - p + 1, \quad \text{weil } \frac{1}{2}\frac{n}{p} - p \geq 0 \Leftrightarrow n \geq 2p^2$$

2. Fall $i = p$

Alle von Prozess p zu misdenden Elemente müssen größer als $Y_{(p-1)p+\frac{p}{2}}$ sein.

Mit Lemma 1 folgt: $N_X\left(\leq Y_{(p-1)p+\frac{p}{2}}\right)$

$$\geq \frac{n}{p^2}\left(p^2 - \frac{3}{2}p\right) + p$$

$$= n - \frac{3}{2}\frac{n}{p} + p$$

Damit folgt

$$\phi_p = n - N_X\left(\leq Y_{(p-1)p+\frac{p}{2}}\right)$$

$$\leq \frac{3}{2}\frac{n}{p} - p \overset{!}{\leq} 2\frac{n}{p} - \frac{n}{p^2} - p + 1$$

falls $\frac{1}{2}\frac{n}{p} - \frac{n}{p^2} + 1 \geq 0$ gilt, weil $p \geq 2$

3. Fall $1 < i < p$

$$\phi_i = n - N_X\left(\leq Y_{(i-1)p+\frac{p}{2}}\right) - N_X\left(> Y_{ip+\frac{p}{2}}\right)$$

$$\leq n - \left(\frac{n}{p^2}\left((i-1)p - \frac{p}{2}\right) + p\right) - \left(\frac{n}{p^2}\left((p-i)p - \frac{p}{2} + 1\right) - 1\right)$$

$$= 2\frac{n}{p} - \frac{n}{p^2} - p + 1$$

2.3 Sortiernetze

Batcher 1968: Netze zum Sortieren in polylogarithmischer Zeit

hier: odd-even-merge-sort

Zentrale Idee: Mischen sortierter Teilfolgen

$A = (a_1 \dots a_i)$
$B = (b_1 \dots b_j)$
→ MERGE → $C = (c_1 \dots c_{i+j})$ sortiert

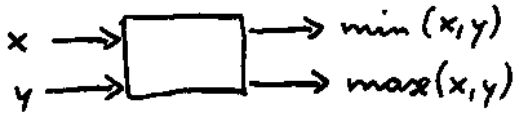
im schlechtesten Fall: $i+j-1$ Vergleiche

Das klassische Mergesort wird in Abhängigkeit von den Vergleichsergebnissen gesteuert.

→ wissensabhängig (non-oblivious)

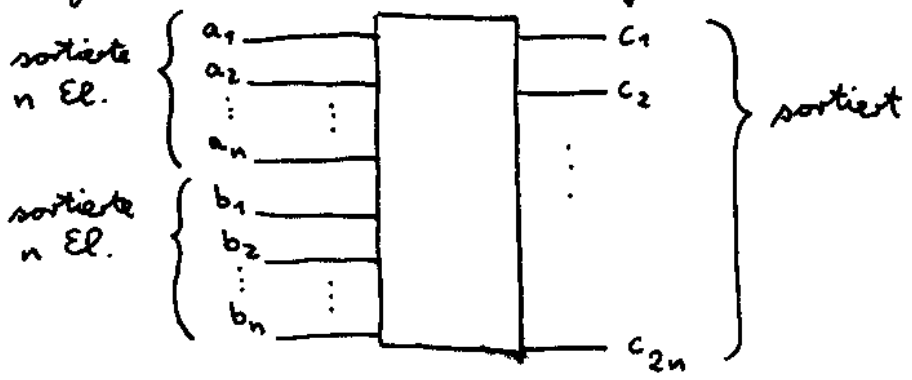
Sortiernetze arbeiten wissensunabhängig (oblivious) mit fest verdrahteten Komparator-Bausteinen.

Komparatorbaustein



im Folgenden stets: $n = 2^k$ zu sortierende Werte.

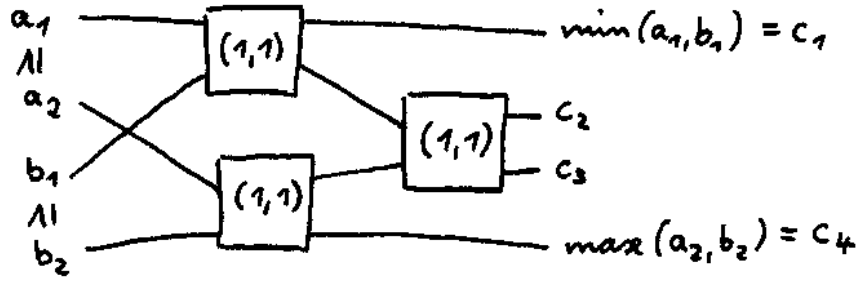
Gesucht ist ein (n,n)-Merger:



induktive Konstruktion:

$n = 1$: Ein Komparator ist ein $(1,1)$-Merger

$n = 2$:



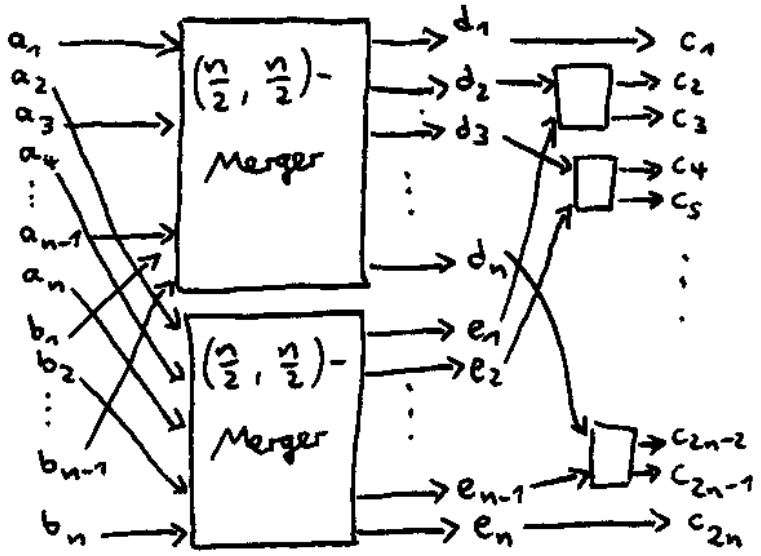
bel. $n > 1$

Voraussetzung: Es stehen $(\frac{n}{2}, \frac{n}{2})$-Merger zur Verfügung.

Motivation: $\frac{x_{[k]}}{l}$ bezeichne Teilfolge $\langle x_{k+il} \mid i \geq 0 \rangle$

Die sortierten Teilfolgen $\frac{a_{[1]}}{2}$ und $\frac{b_{[1]}}{2}$ (ungerade Pos.) werden mit einem $(\frac{n}{2}, \frac{n}{2})$-Merger zu einer oberen Folge d verschmolzen.

Parallel werden die Teilfolgen $\frac{a_{[2]}}{2}$ und $\frac{b_{[2]}}{2}$ (gerade Pos.) mit einem weiteren $(\frac{n}{2}, \frac{n}{2})$-Merger zu einer sortierten unteren Folge e verschmolzen

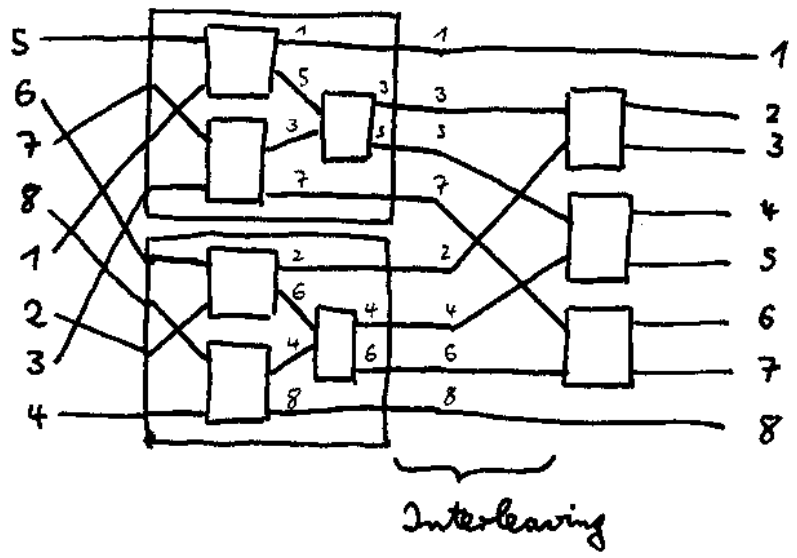


Schließlich werden die Folgen $d \setminus \{d_1\}$ und $e \setminus \{e_n\}$ mit $n-1$ Komparatoren elementweise verglichen und umsortiert.

Satz 2.1 Die Resultatfolge c ist sortiert.

Beispiel $a = (5, 6, 7, 8)$

$b = (1, 2, 3, 4)$



Satz 2.2 (0-1-Prinzip)

Ein Sortieralgorithmus bestehe nur aus eingabeunabhängigen Vergleich-Austausch-Anweisungen. Dann gilt:

Sortiert der Algorithmus jede Eingabefolge, die nur aus Nullen und Einsen besteht, so sortiert er jede Eingabefolge.

Beweis durch Widerspruch:

Annahme, die Eingabefolge $x_1, \ldots, x_n$ werde nicht in die korrekte Reihenfolge

$$x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$$

sortiert. Sei k die erste Stelle, an der sich die sortierte Folge von der Ausgabe des Algorithmus $x_{\sigma(1)} \ldots x_{\sigma(1)}$ unterscheidet, d.h. $x_{\sigma(k)} > x_{\pi(k)}$

Def. zu $x_1 \ldots x_n$ und k eine 0-1-Folge

$$y_i = \begin{cases} 0 & \text{falls } x_i \leq x_{\pi(k)} \\ 1 & \text{falls } x_i > x_{\pi(k)} \end{cases}$$

Wird diese Folge dem wissensunabhängigen Sortierverfahren übergeben, so finden die gleichen Vergleichs-/Austauschschritte statt, denn $x_i \geq x_j \rightsquigarrow y_i \geq y_j$

Insbesondere steht an der k-ten Stelle der Ausgabefolge $y_{\sigma(k)} = 1$ und irgendwo rechts davon der Wert $y_{\pi(k)} = 0$.

Die 0-1-Folge $y_1 \dots y_n$ wird also nicht richtig sortiert. ↯

30.05.2007

Interleaving-Schema

$d_1 \quad d_2 \quad d_3 \quad d_4 \quad \dots \quad d_n$

$\quad e_1 \quad e_2 \quad \dots \quad e_{n-1} \quad e_n$

$d_1 \quad \{d_2 \; e_1\} \quad \{d_3 \; e_2\} \quad \dots \quad \{d_n \; e_{n-1}\} \quad e_n$

Vergleichs-/Austauschschritte

Satz 2.1: Die Resultatfolge c ist sortiert

Beweis: a und b seien sortierte 0-1-Folgen.

Sei a_k die letzte Null der Folge a

und b_ℓ die letzte Null der Folge b.

$a = (0, \dots, 0, 1, \dots, 1)$ (1, k, n)

$b = (0, \dots, 0, 1, \dots, 1)$ (ℓ)

$k, \ell \in \{0, \dots, n\}$

Für die Teilfolgen gilt:

$\frac{a_{[1]}}{2}$ hat $\lceil \frac{k}{2} \rceil$ Nullen, $\frac{b_{[1]}}{2}$ entsprechend $\lceil \frac{\ell}{2} \rceil$

$\frac{a_{[2]}}{2}$ hat $\lfloor \frac{k}{2} \rfloor$ Nullen, $\frac{b_{[2]}}{2}$ entsprechend $\lfloor \frac{\ell}{2} \rfloor$.

Damit hat die Folge d $\gamma := \lceil \frac{k}{2} \rceil + \lceil \frac{\ell}{2} \rceil$ Nullen

und die Folge e $\delta := \lfloor \frac{k}{2} \rfloor + \lfloor \frac{\ell}{2} \rfloor$ Nullen.

Für die Differenz $\Delta := \gamma - \delta$ gilt: $\Delta \in \{0, 1, 2\}$.

Wir betrachten das Interleaving-Schema für diese 3 Fälle:

$\Delta = 2$

d ... 0 0 0 1 1 ...

e ... 0 0 1 1 1 ...

$\Delta = 1$

d ... 0 0 1 1 1 ...

e ... 0 0 1 1 1 ...

$\Delta = 0$

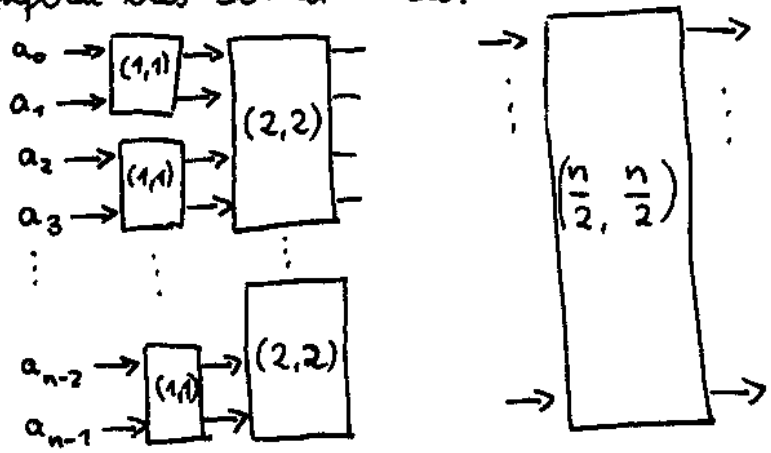
d ... 0 1 1 1 ...

e ... 0 0 1 1 ...

!

Die Resultatfolge ist in jedem Fall eine sortierte 0-1-Folge. ⊕

Aufbau des Sortiernetzes:



Mit $n = 2^{k+1}$ gibt es k solche Mischstufen.

Durchlaufzeit (= Zahl vertikaler Komparatorstufen)

$t(n) = \sum_{i=0}^{k} \tau(2^i)$, wobei $\tau(n)$ die Durchlaufzeit eines (n,n)-Mergers sei.

Komparatorzahl

$N(n) = \sum_{i=0}^{k} 2^{k-i} \, v(2^i)$, wobei $v(n)$ die Komparatorzahl eines (n,n)-Mergers sei.

Bestimme $\tau(n)$ und $v(n)$ durch Analyse eines (n,n)-Mergers:

Es gilt: $\tau(1) = 1$
$\tau(2n) = \tau(n) + 1 \qquad (n = 2^k, \; k \geq 0)$

$v(1) = 1$
$v(2n) = 2 \cdot v(n) + 2n - 1$

Mit vollständiger Induktion zeigt man, dass

$\tau(n) = \log n + 1$, $\quad v(n) = n \cdot \log n + 1$

Damit folgt:

$t(n) = \sum_{i=0}^{k} (i+1) = \frac{(k+1)(k+2)}{2} \in O(k^2) = O(\log^2 n)$

$\Rightarrow$ polylogarithmische Laufzeit

$N(n) = \sum_{i=0}^{k} 2^{k-i} \, (i \cdot 2^i + 1) = \ldots = 2^{k+1} - 1 + 2^k \frac{k(k+1)}{2}$
$\in O(n \log^2 n)$

Kosten: $t(n) \cdot N(n) \in O(n \log^4 n)$

2.4 Der Algorithmus von Cole

optimale Lösung auf CREW-PRAM, d.h. $O(\log n)$ Zeit mit $O(n)$ Proz.
paralleles Mischsortieren in vollständigen Binärbaum
Hilfsmittel: Skelette von Folgen
→ Konstanter Mischaufwand

Def. 2.3

X und Y seien sortierte Folgen ganzer Zahlen.

$[x]$ entstehe aus X durch Hinzunehmen der Elemente $-\infty$ und ∞.

(a) Sei $a < b$ und $x \in X$

x heißt zwischen a und b $:\Leftrightarrow$ $a < x \leq b$

(b) X heißt Skelett von Y,

in Zeichen $X \propto Y$,

falls für alle $k \geq 2$ zwischen je k Elementen von $[x]$ höchstens $2k-1$ Elemente von Y liegen.

Notation: X gemeinsames Skelett von Y und Z

$$X \propto \frac{Y}{Z}$$

$Y \,\&\, Z$ sei die Verschmelzung (merge) von Y und Z.

Bsp. $Y = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20)$

$X = (5, 10, 12, 17)$

$Z = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21)$

Es gilt $X \propto \frac{Y}{Z}$

Skelett-Merging

Die beiden Folgen werden anhand des gemeinsamen Skeletts in Teilfolgen zerlegt. Die entsprechenden Teilfolgen können dann in konstanter Zeit $O(1)$ verschmolzen werden.

Lemma 2.4

Seien Y und Z sortierte endliche Folgen mit gemeinsamem Skelett X.

Sei $Y(i) := \{y \in Y : x_{i-1} < y \leq x_i\}$

$Z(i) := \{z \in Z : x_{i-1} < z \leq x_i\}$

Dann gilt:

$$Y \& Z = Y(1) \& Z(1),\ Y(2) \& Z(2),\ \ldots,\ Y(|X|+1) \& Z(|X|+1)$$

$$-\infty \mid \begin{matrix}1,\ 4\\ 2,\ 3\end{matrix} \mid 5 \mid \begin{matrix}6,\ 9\\ 7,8,10\end{matrix} \mid 10 \mid \begin{matrix}11,\ 12\\ \varnothing\end{matrix} \mid 12 \mid \begin{matrix}13,\ 16\\ 14,15,17\end{matrix} \mid 17 \mid \begin{matrix}19,20\\ 18,\ 21\end{matrix} \mid \infty$$

Grundidee des Algorithmus von Cole

- Verschmelzen von Folgen mit gem. Skelett
- Zu sortierende Liste ist zu Beginn auf die Blattknoten eines vollständigen Binärbaums verteilt.
- Innere Knoten haben die Aufgabe, die Liste der Blattknoten ihres Unterbaums zu sortieren.
- Die Verschmelzung von Teillisten erfolgt in mehreren Baumebenen fließbandartig.
- Da immer Skelett-Merging durchgeführt wird, bleibt der Aufwand pro Baumebene konstant und insgesamt ergibt sich ein logarithmischer Aufwand.

Bezeichnungen

Gegeben sei der vollst. binäre Baum mit $k = 2^n$ Blättern.

$T(v)$ sei der Unterbaum mit Wurzel v

$val(v)$ sei die momentane Folge des Knotens v

$list(v)$ sei die sortierte Liste, die aus den Werten der Blätter von $T(v)$ entsteht.

Es gilt: $val(v)$ ist stets sortierte Teilliste von $list(v)$.

Ein Knoten v heißt vollständig, falls $val(v) = list(v)$, sonst unvollständig.

Arbeitsweise eines bel. inneren Knotens v:

Zu Beginn ist $val(v) = \emptyset$, d.h. $|val(v)| = 0$.
Der Knoten v wird aktiv, wenn er vom linken Kind eine Folge X_1 der Länge 1 erhält und vom rechten Kind eine Folge Y_1 der Länge 1, die er zu einer Folge $val(v)$ der Länge 2 verschmilzt.
In jedem weiteren Schritt j erhält der Knoten von seinen Kindknoten Folgen X_j und Y_j, so dass

$$X_{j-1} = \frac{X_{j,[1]}}{2} =: \frac{X_j}{2}, \quad Y_{j-1} = \frac{Y_j}{2}$$

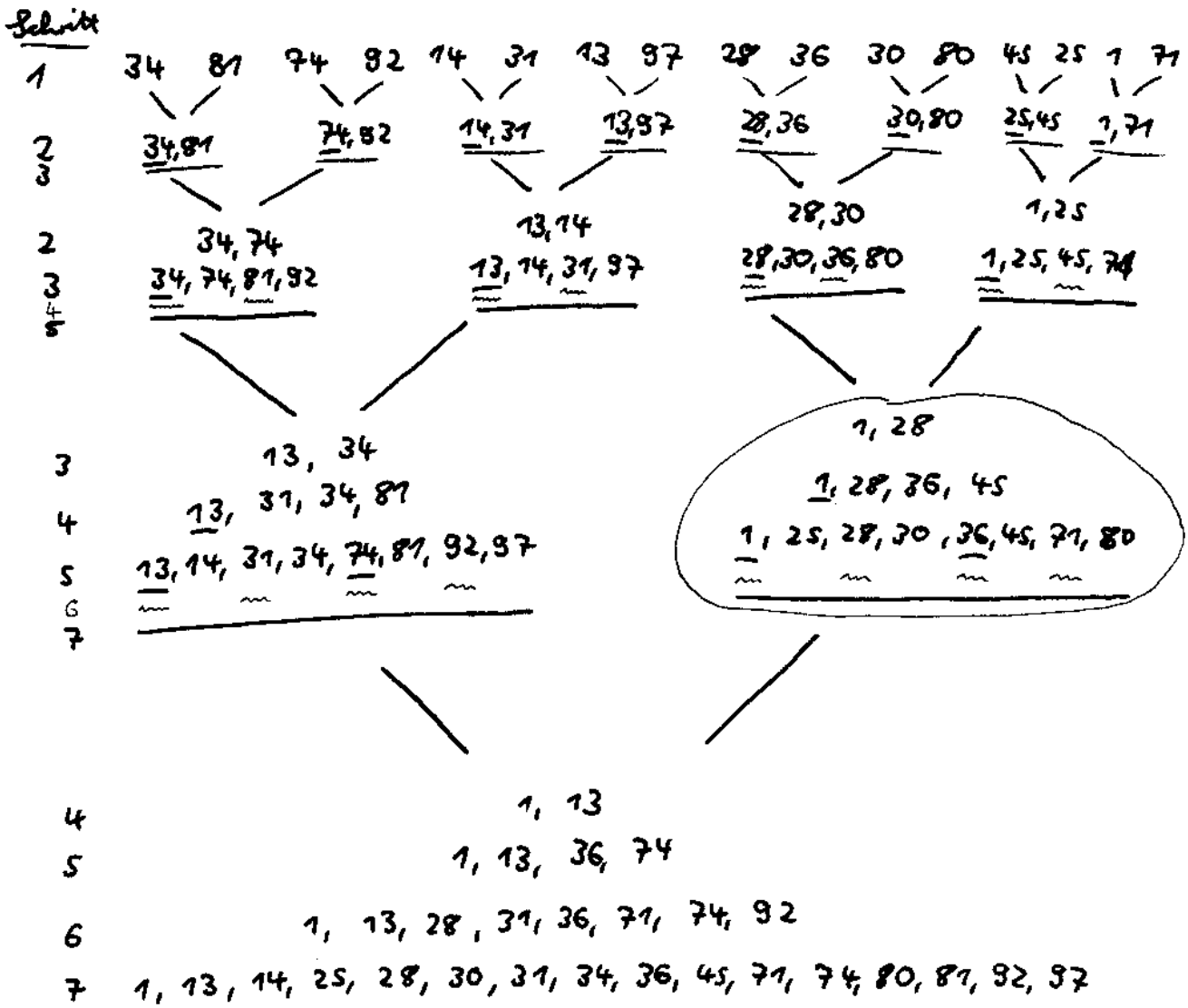
In jedem Schritt werden X_j und Y_j zu einem neuen $val(v)$ gemischt.
In jedem Schritt wird die Länge von $val(v)$ verdoppelt, bis $list(v)$ erreicht wird.
$\rightarrow$ Knoten v wird vollständig.

Ausgabevorschriften:

① Ist v unvollständig und $|val(v)| \geq 4$, so sendet v die Folge $\frac{val(v)}{4}$.

② Ist v vollständig, so bleibt v noch zwei Schritte aktiv. Im vorletzten Schritt sendet v $\frac{list(v)}{2}$ und im letzten Schritt $list(v)$. Danach wird v inaktiv.

Bsp.: 16 Zahlen sortieren



Schritt 1 $X_1 = (28)$ $Y_1 = (1)$

$val_1(v) = (1, 28)$

$\emptyset$

Schritt 2 $X_2 = (28, 36)$ $Y_2 = (1, 45)$

$val_2(v) = (1, 28, 36, 45)$

$Z_1 = (1)$

Schritt 3 v wird vollständig

$X_3 = (28, 30, 36, 80)$ $Y_3 = (1, 25, 45, 71)$

$val_3(v) = (1, 25, \ldots, 80) = list(v)$

$Z_2 = (1, 36)$

vorletzter Schritt $Z_3 = (1, 28, 36, 71)$

Schritt 5: $Z_4 = \underline{list}(v)$ $\rightsquigarrow$ wird inaktiv

Satz 2.5: Sei v ein Knoten mit $|list(v)| \geq 4$.
Seien X_{i+1}, Y_{i+1} die Eingabefolgen und Z_i die Ausgabefolge in seinem $(i+1)$-ten Schritt. Dann gilt für alle $i \in \mathbb{N}$:

$X_{i+1} \propto X_{i+2}$ und $Y_{i+1} \propto Y_{i+2}$ $\rightsquigarrow$ $Z_i \propto Z_{i+1}$

Lemma 2.6: $X \propto X'$, $Y \propto Y'$ $\rightsquigarrow$ $\frac{X \,\&\, Y}{4} \propto \frac{X' \,\&\, Y'}{4}$

Beweisskizze: Gelte $X \propto X'$, $Y \propto Y'$.

Zeige: (a) $X \,\&\, Y \propto X'$, $X \,\&\, Y \propto Y'$

(b) Im allgemeinen gilt nicht: $X \,\&\, Y \propto X' \,\&\, Y'$
Jedoch liegen zwischen je k aufeinanderfolgenden Elementen von $X \,\&\, Y$ höchstens $2k+2$ Elemente von $X' \,\&\, Y'$.

(c) Mit (a) und (b) folgt die Aussage des Lemmas. ⌖

Beweis von Satz 2.5

Sei $|list(v)| = 2^{j-1}$ $j \geq 3$,
d.h. Knoten v ist nach seinem $(j-1)$-ten Schritt vollständig.
Mit der Ausgabevorschrift ① erhält man mit Lemma 2.6 für alle i mit $1 \leq i < j-2$

$$Z_i = \frac{val(v)}{4} = \frac{X_{i+1} \,\&\, Y_{i+1}}{4} \propto \frac{X_{i+2} \,\&\, Y_{i+2}}{4} = Z_{i+1}$$

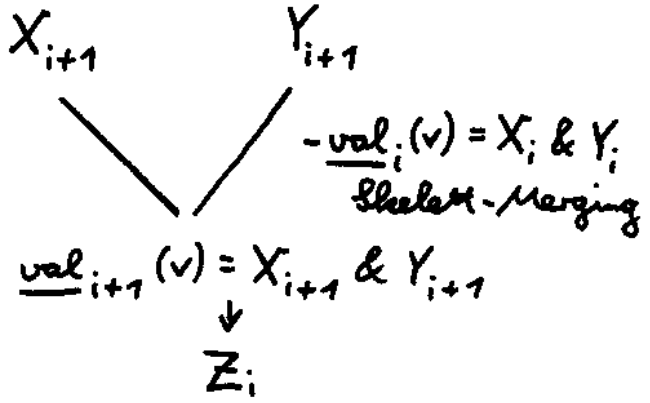
Sobald v vollständig, tritt Vorschrift ② in Kraft und es gilt: $Z_{j-1} = \frac{list(v)}{4} \propto \frac{list(v)}{2} = Z_j$

$$Z_j = \frac{list(v)}{2} \propto list(v) = Z_{j+1}$$ ⌖

06.06.2007

Algorithmus von Cole

Knoten v in seinem $(i+1)$-ten aktiven Schritt



Es gilt

$$Z_i = \begin{cases} \dfrac{X_{i+1} \,\&\, Y_{i+1}}{4} & \text{falls } |\underline{val}_{i+1}(v)| \geq 4 \\ \dfrac{\underline{list}(v)}{2} & \text{im vorletzten Schritt} \\ \underline{list}(v) & \text{im letzten Schritt} \end{cases}$$

Invariante: $X_{i+1} \propto X_{i+2}$, $Y_{i+1} \propto Y_{i+2} \rightsquigarrow Z_i \propto Z_{i+1}$

Zeitlicher Ablaufplan

Phase	gesendete Folgen (mit X_i wird gleichzeitig Y_i gesendet)
1	$Z_1^{Blatt} \to X_1^A$
2	$Z_1^A \to X_1^B$
3	$Z_2^A \to X_2^B$
4	$Z_1^B \to X_1^C$
5	$Z_2^B \to X_2^C$
6	$Z_3^B \to X_3^C$ $\quad Z_1^C \to X_1^D$
	$Z_2^C \to X_2^D$
	$Z_3^C \to X_3^D$
	$Z_4^C \to X_4^D$

Beobachtung

- Wird Knoten v im Schritt i inaktiv, so wird sein Elternknoten im Schritt $i+3$ inaktiv.
  $\rightsquigarrow$ Laufzeit $t(n) = 3 \log n = \underline{O(\log n)}$
- Prozessoren werden nur für die aktiven Knoten benötigt. Jeder aktive Knoten benötigt $|val(v)|$ Prozessoren. Die gesamte zu sortierende Folge liegt in der Baumebene, die vor ihrem letzten Schritt steht. In jeder darüberliegenden Ebene findet man höchstens halb so viele Elemente, d.h.

$$N(n) = \sum_{v \text{ aktiv}} |val(v)| = O(2n) = O(n)$$

$\Rightarrow$ Algorithmus ist kostenoptimal

Skalett-Merging auf CREW-PRAM mit konstantem Aufwand

Def. 2.7 Seien A und B sortierte endliche Folgen ganzer Zahlen. Der Rang von $a \in A$ in B ist definiert als relative Koordinate von a bzgl. B, d.h.

$$b_i < a \leq b_{i+1} \text{ für } b_i, b_{i+1} \in B$$

$\rightsquigarrow \Sigma g_B(a) = i$

Die Rangfunktion von A bzgl. B, $\Sigma g_B(A)$, sei ein Feld mit $|A|$ Einträgen. Der i-te Eintrag sei $\Sigma g_B(a_i)$.

Bsp. $Y = (1, 4, 6, 9, 11, 12, 13, 16, 19, 20)$

$X = (5, 10, 12, 17)$

$Z = (2, 3, 7, 8, 10, 14, 15, 17, 18, 21)$

Dann gilt:

$\text{rg}_X(Y) = (0, 0, 1, 1, 2, 2, 3, 3, 4, 4)$

$\text{rg}_X(Z) = (0, 0, 1, 1, 1, 3, 3, 3, 4, 4)$

$\text{rg}_Y(X) = (2, 4, 5, 8)$

$\text{rg}_Z(X) = (2, 4, 5, 7)$

Lemma 2.7 (Rangberechnung)

Ist $B = (b_1, \dots, b_k)$ eine sortierte Folge, so kann für ein a der Rang $\text{rg}_B(a)$ in $O(1)$ mit $O(k)$ Prozessoren auf einer CREW-PRAM berechnet werden.

Beweis: Betrachte die Folge $:= (\underbrace{-\infty}_{b_0}, b_1, \dots, b_k, \underbrace{\infty}_{b_{k+1}})$

PRAM-Code:

for all i in $0 \le i \le k$ do

if $\underbrace{b_i < a \le b_{i+1}}_{\text{CR}}$ then $\underbrace{\text{rg}_B(a) := i}_{\text{EW}}$

Lemma 2.8: Seien X, Y, Z sortierte Folgen und es sei $Z \propto_Y^X$. Falls $\text{rg}_X([Z])$, $\text{rg}_Y([Z])$, $\text{rg}_Z([Y])$ bekannt sind, kann das Skelett-Merging von X und Y bzgl. Z in $O(1)$-Zeit mit $|X| + |Y|$ Prozessoren auf einer CREW-PRAM durchgeführt werden.

13.06.2007

3. Matrixoperationen

Transposition, Multiplikation, Inversion

o.B.d.A. $n \times n$-Matrizen

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} = (a_{ij})_{1 \le i,j \le n}$$

3.1 Transposition

Spiegelung an der Diagonalen

Zeilenvektoren $\leftrightarrow$ Spaltenvektoren

Aufwand seq. Transposition: $O(n^2) = \Omega(n^2)$

3.1.1 EREW-PRAM-Transposition

$\binom{n}{2} = \frac{n(n-1)}{2}$ Proz. P_{ij}, $1 \le i < j \le n$

A liegt im gemeinsamen Speicher

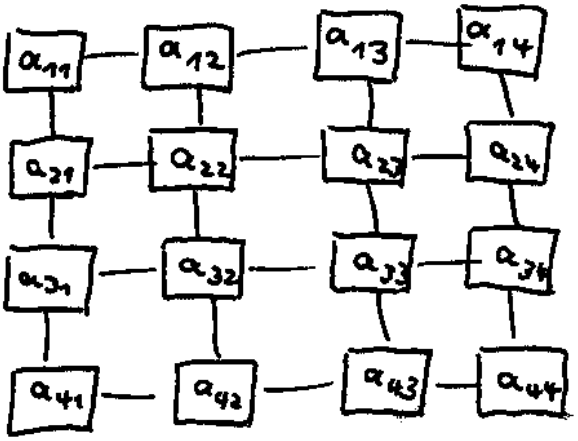
Alle Proz. führen parallel in $O(1)$ Schritten den Befehl $a_{ij} \leftrightarrow a_{ji}$ aus.

P_{ij} arbeitet exklusiv auf a_{ij} und a_{ji} $\rightarrow$ EREW.

3.1.2 Gitter-Transposition

$n \times n$ Gitter $\rightarrow$ natürliche 1-1-Verteilung der Matrixelemente.

Bsp. $n=4$



Jeder Prozessor P_{ij}, $1 \le i,j \le n$, $i \ne j$

habe drei Register.

$A(i,j)$ speichert zu Beginn a_{ij} und am Ende a_{ji}

$B(i,j)$ empfängt Daten von $P_{i,j+1}$ (rechter Nachbar),
falls $i \leq j$,
und von $P_{i-1,j}$ (oberer Nachbar), falls $i > j$.

$C(i,j)$ empfängt Daten von $P_{i,j-1}$ (linker Nachbar),
falls $i < j$,
und von $P_{i+1,j}$ (unterer Nachbar), falls $i \geq j$.

Das Matrixelement a_{ij} wird als Tripel (a_{ij}, j, i) gesendet.
Zuerst vertikal ($i > j$) bzw. horizontal ($i < j$) zu P_{jj}
und dann horizontal bzw. vertikal zu P_{ji}.
In jedem Schritt wird getestet, ob die Position (j,i)
im Datentripel mit der Prozessorpos. übereinstimmt.
Wenn ja, wird das Matrixelement ins A-Reg. kopiert.

Analyse: längster Weg bei Matrixelement a_{1n} und a_{n1}
$\rightarrow 2(n-1)$ Schritte
$\Rightarrow$ keine Kostenoptimalität, da parallele Kosten $O(n^3)$.

3.1.3 Shuffle-Transposition (Stone 1971)

Sei $n = 2^q$
Perfect-Shuffle-Netzwerk mit $n^2 = 2^{2q}$ Proz. $P_0 \dots P_{n^2-1}$
Zeilenweise Speicherung der Matrix
$a_{ij} \mapsto P_k := P_{ij}$ mit $k = 2^q(i-1) + j - 1$
Transposition bedeutet Übergang zu spaltenweiser Speicherung der Matrixelemente.
$a_{ij} \mapsto P_l$ mit $l = (j-1) \cdot 2^q + (i-1)$

Satz 3.1 Nach q Shuffle-Schritten enthält Prozessor P_k das Element a_{ji}.

Beweis: Zu Beginn enthält P_k mit $k = (i-1)\cdot 2^q + (j-1)$ das Element a_{ij}.

Binärdarstellung von k:

Bei jedem Shuffle-Schritt wandert ein Matrixelement zu dem Prozessor, dessen Prozessor-ID durch einen zyklischen Linksshift entsteht.
Nach q zyklischen Linksshifts befindet sich a_{ij} in P_l mit l:

⊕

Analyse: parallele Zeit: $q = O(\log n)$ Schritte
parallele Kosten: $O(n^2 \log n)$
nicht optimal, aber besser als Gittertransposition.

14.06.2007

3.2 Multiplikation

Geg. Matrizen $\mathcal{A} = (a_{ij})_{1 \le i,j \le n}$,

$\mathcal{B} = (b_{ij})_{1 \le i,j \le n}$

Gesucht Matrix $\mathcal{C} = (c_{ij})_{1 \le i,j \le n}$

mit $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

seq. Aufwand: $O(n^3)$ Op.s bei Schulmethode

3.2.1 CRCW-PRAM-Multiplikation

Auflösen von Schreibkonflikten durch Bitsektion Summation aller Schreibwünsche

n^3 Proz. P_{ijk}, Matrizen $\mathcal{A}$ und $\mathcal{B}$ im gemeinsamen Speicher.

Alle Proz. P_{ijk} führen parallel die Berechnung

$c_{ij} = a_{ik} * b_{kj}$ durch.

Analyse: par. Laufzeit: $O(1)$
par. Kosten: $O(n^3)$
relativ opt. zur Schulmethode

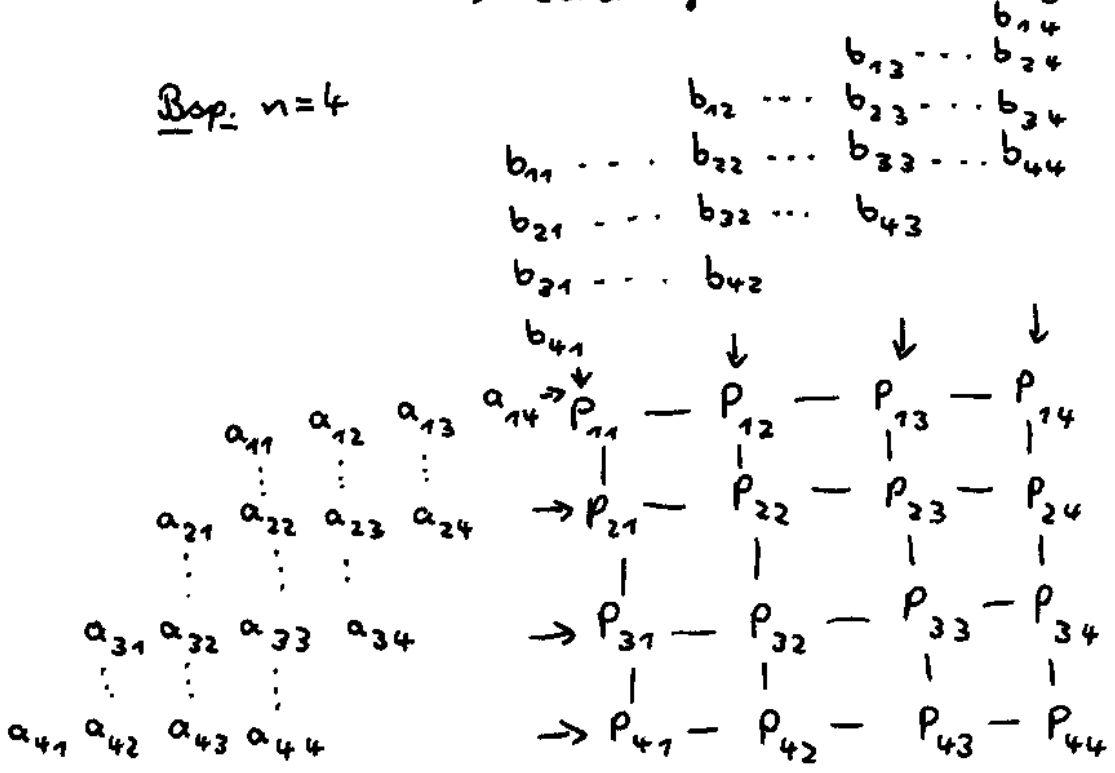
3.2.2 Gitter-Multiplikation

n^2 Proz. P_{ij}

Eingabe der Matrizen Or und Lr erfolgt über Randprocessoren.
Eingabe von Or zeilenweise verschoben
Eingabe von Lr spaltenweise verschoben

→ Scherungsschema (skewing scheme)

Bsp.: $n=4$



Die Zeile i der Matrix Or wird einen Schritt später als die Zeile $i-1$ $(2 \leq i \leq n)$ eingelesen.
Analog wird Spalte j einen Schritt später als Spalte $j-1$ $(2 \leq j \leq n)$ eingelesen.

Die versetzte Eingabe sichert, dass die Matrixelemente a_{ik} und b_{kj} im Schritt $\boxed{i + (j-1) + (n-k)}$ gleichzeitig den Prozessor P_{ij} erreichen.

Wenn ein Prozessor die Eingaben a und b erhält, berechnet er $c \leftarrow c + a * b$ und schickt a horizontal an den rechten Nachbarn und b vertikal an den unteren Nachbarn weiter.

Analyse: Die Elemente a_{n1} und b_{1n} sind die letzten, die Proz. P_{nn} erreichen. Dazu werden

$$\underset{i}{n} + \underset{j}{(n-1)} + \underset{k}{(n-1)} = 3n-2$$ Schritte benötigt.

$\Rightarrow$ parallele Laufzeit: $O(n)$
parallele Kosten: $O(n^3)$

Variation dieses Verfahrens für Torus statt Gitter nach Gentleman 78, Cannon?

Die Matrizen A und B seien elementweise auf Torusstruktur verteilt, d.h. P_{ij} enthalte a_{ij} und b_{ij}.

1. Verschiebung der Matrizen, damit El. der a_{ik} und b_{kj} auf P_{ij} liegen. Dazu: Rotiere i-te Zeile von A um i-1 Pos. nach links. Rotiere j-te Spalte von B um j-1 Pos. nach oben.

2. Schrittweises Multiplizieren und Aufaddieren sowie Rotation der Zeilen von A um eine Pos. nach links/rechts. Rotation der Spalten von B um eine Pos. nach oben/unten.

3.2.3 Hypercube-Multiplikation

[Dekel, Nassimi, Sahni 1981]

n^3 Proz., $O(\log n)$ par. Laufzeit

Grundidee: Verteilung der n^3 zu multiplizierenden Paare (a_{ij}, b_{jk}) auf die n^3 Proz., so dass alle Mult. parallel in einem Schritt durchgeführt werden können.
Logarithmische Laufzeit durch Broadcast zum Umschichten der Matrixelemente sowie Summation der Produkte (→ Reduktion).

Sei $n = 2^q$. Die $n^3 = 2^{3q}$ Prozessoren P_r werden in einem $(n \times n \times n)$-Gitter angeordnet. Proz. P_r wird die Adresse (i, j, k) zugeordnet, wobei r eine Binärzahl mit $3q$ Bits ist und

$$r = \underbrace{r_{3q-1} \cdots r_{2q}}_{q} \; \underbrace{r_{2q-1} \cdots r_{q}}_{q} \; \underbrace{r_{q-1} \cdots r_{0}}_{q}$$

$\Rightarrow i \qquad \Rightarrow j \qquad \Rightarrow k$

d.h. $r = i \cdot 2^{2q} + j \cdot 2^q + k = in^2 + jn + k$

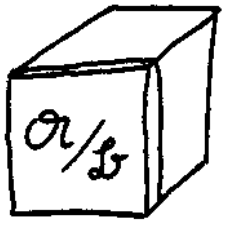
für $i, j, k \in \{0, \dots, n-1\}$

Proz. $P_{(i,j,k)}$ habe lokale Register $A(i,j,k)$, $B(i,j,k)$, $C(i,j,k)$

Initialisierung:
$$\left.\begin{aligned} A(0,j,k) &= a_{jk} \\ B(0,j,k) &= b_{jk} \end{aligned}\right\} \; 0 \le j,k \le n-1$$

Alle anderen Reg. seien leer.

Ziel des Alg.: Am Ende gilt: $C(0,j,k) = c_{jk} = \sum_{i=0}^{n-1} a_{ji} \cdot b_{ik}$



Der Algorithmus hat 4 Phasen:

1 Verteilen der Matrixelemente im Hypercube

2 Umspeicherung der Matrixelemente

3 parallele Multiplikation in allen PEs

4 Akkumulation und Summation der Produkte

Phase 1

kopiere $A(0,j,k)$ nach $A(i,j,k)$ für alle $1 \le i \le n-1$.
kopiere $B(0,j,k)$ nach $B(i,j,k)$ für alle $1 \le i \le n-1$.

→ paralleler Broadcast in 1. Dimension des Würfels

for $l := 0$ to $q-1$ do in parallel

if bit_{3q-1-l} (Prozessorindex) $= 0$

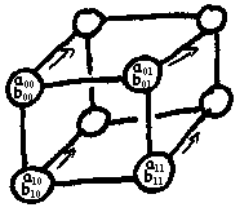
then sende A- und B- Register
an Nachbarn mit $bit_{3q-1-l} = 1$

else empfange A-und-B- Reg.wert
von Nachbarn mit $bit_{3q-1-l} = 0$

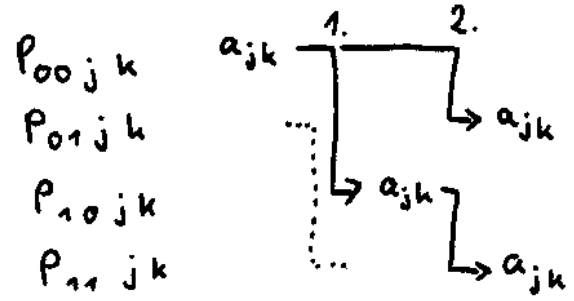
⇒ $2q$ Schritte

Jeder zweite Prozess sendet, die anderen empfangen

Bsp: $n=2, q=1$ ↝ Hypercube der Dim 3



$n=4, q=2$ ↝ Hypercube der Dim 6
4 x 4 x 4 Würfel



Phase 2: Umfächerung der Koeffizienten, damit zu multiplizierende Werte zusammenkommen.

Kopiere a_{ji} aus $A(i, j, i)$ in $A(i,j,\underline{k})$ für alle $0 \le k < n$

(i: beliebig)

Kopiere b_{ik} aus $B(i, i, k)$ in $B(i,j,k)$ für alle $0 \le j < n$

durch parallele Broadcasts in entsprechenden Dimensionen.

for $l := 0$ *to* $q-1$ *do*

nur für $k=i$ stets erfüllt { *if* $r_{\underbrace{3q-1-l}_{i}} = r_{\underbrace{2q-1-l}_{k/j}}$

then sende ^{B}A-Reg. an Prozessornachbarn mit Bit $2q-1-l$ komplementiert.

else empfange ^{B}A-Reg.-wert von Nachbarn mit Bit $2q-1-l$ komplementiert

$\Rightarrow 2q$ Schritte

Nach dieser Phase enthält P_{ijk} die El. a_{ji} und b_{ik}.

20.06.2007

DNS-Verfahren für Matrixmultiplikation im Hypercube

$n = 2^q$, HC-Dim $3q \sim n^3$ Knoten

$\langle P_r \mid 0 \le r < n^3 \rangle$

$r = \underbrace{r_{3q-1} \cdots r_{2q}}_{i} \; \underbrace{r_{2q-1} \cdots r_{q}}_{j} \; \underbrace{r_{q-1} \cdots r_{0}}_{k}$

4 Phasen

1. Phase: Verteilen der Matrixelemente im HC

$a_{jk} \to A(0,j,k)$, $b_{jk} \to B(0,j,k)$ } Broadcast { $A(i,j,k)$, $B(i,j,k)$ für $0 \le i < n$

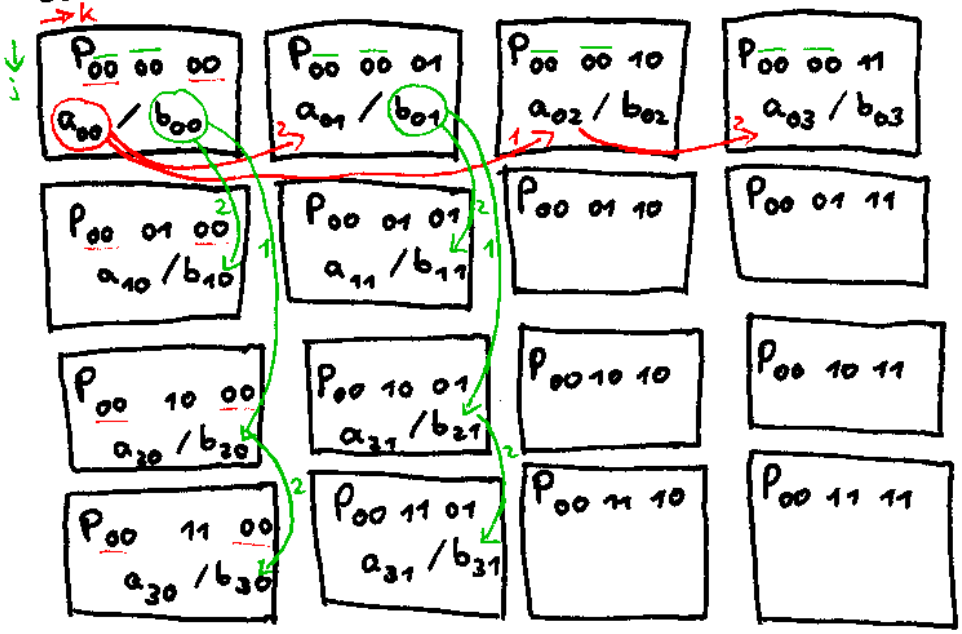
2. Phase: Umspeichern der Matrixelemente im Hypercube

Broadcast von a_{ji} aus $A(i,j,\underline{i})$ nach $A(i,j,k)$
für alle $0 \le k < n$

Broadcast von b_{ik} aus $B(i,\underline{i},k)$ nach $B(i,j,k)$
für alle $0 \le j < n$

Bsp. $n=4$, $q=2$ ↷ Dim 6

Ebene $i = 00$



im Bsp.: Verteilung der 1. Spalte von $\mathcal{A}$
Verteilung der 2. Spalte von $\mathcal{B}$

Situation nach Umspeicherung:
$P(i,j,k)$ enthält a_{ji} und b_{ik}

3. Phase: Skalare Multiplikation

Alle PEs führen parallel den Schritt

$$C(i,j,k) = \underbrace{A(i,j,k)}_{a_{ji}} * \underbrace{B(i,j,k)}_{b_{ik}} \quad \text{aus.}$$

4. Phase: Parallele Reduktion der n^2 Summen

$$c_{jk} = \sum_{i=0}^{n-1} a_{ji} \cdot b_{ik}$$

zur Ebene $i = 0$.

for $l := 0$ to $q-1$ do

if $r_{2q+l} = 1$

then sende C-Reg. an Prozessornachbarn mit $r_{2q+l} = 0$.

else empfange C-Reg. vom Nachbarn
und addiere diesen Wert zu lokalem C-Reg.

$\Rightarrow q$ Schritte.

Ergebnis: c_{jk} steht im Reg. C $(0, j, k)$.

Laufzeitanalyse:

$$T(n) = O(\underbrace{2q}_{\text{Phase 1}} + \underbrace{2q}_{2} + \underbrace{1}_{3} + q) = O(q) = O(\log n)$$

Kosten: $O(n^3 \log n)$

21.06.2007

3.3 Matrize-Inversion

wichtige Grundop., insbesondere Lösung linearer Gleichungssysteme.

$$Ax = b \quad \rightsquigarrow \quad x = A^{-1} b$$

falls A^{-1} existiert

im folgenden: exakter, paralleler Inversionsalg. in $O(\log^2 n)$ Schritten.

Grundbegriffe aus der linearen Algebra:

- charakteristisches Polynom einer Matrix A

$$\chi(\lambda) = \det(A - \lambda I_n)$$
$$= \lambda^n + c_1 \lambda^{n-1} + \dots + c_{n-1}\lambda + c_n$$
$$= \lambda^n + \sum_{i=1}^{n} c_i \lambda^{n-i}$$

Die Nullstellen von χ heißen Eigenwerte von A.

Es gilt: A ist invertierbar $\rightsquigarrow$ $c_n \neq 0$.

Satz von Cayley-Hamilton

Ist $\lambda^n + \sum_{i=1}^{n} c_i \lambda^{n-i}$ das char. Polynom der Matrix A,

so gilt: $$A^n + \sum_{i=1}^{n} c_i A^{n-i} = 0$$

Voraussetzung: A^{-1} existiert.

Dann folgt:

$$A^{-1}\left(A^n + \sum_{i=1}^{n} c_i A^{n-i}\right) = 0$$

$$\Leftrightarrow A^{n-1} + \sum_{i=1}^{n-1} c_i A^{n-i-1} + c_n A^{-1} = 0$$

$$\Leftrightarrow A^{-1} = -\frac{1}{c_n}\left(A^{n-1} + \sum_{i=1}^{n-1} c_i A^{n-i-1}\right)$$

⊛ siehe unten

Damit kann die Inversion von A durch Bestimmung der Potenzen von A und der Koeffizienten des char. Polynoms erfolgen.

1. Berechnung der Potenzen von A

Gegeben sei ein Hypercube mit n^4 Prozessoren.
Ein Hypercube mit n^4 Proz. kann als Hypercube mit n Proz. betrachtet werden, dessen Knoten nicht einzelne Proz., sondern Hypercubes mit je n^3 Proz. sind.

↘ Hypercubes $H_0, \ldots, H_{n-1}$

Zwischen je zwei benachbarten Knoten des logischen n-Hypercubes existieren entsprechend n^3 „parallele" Verbindungen zwischen den entsprechenden Proz. der Knotenhypercubes.

Ziel: Berechnung von A^i in H_i.

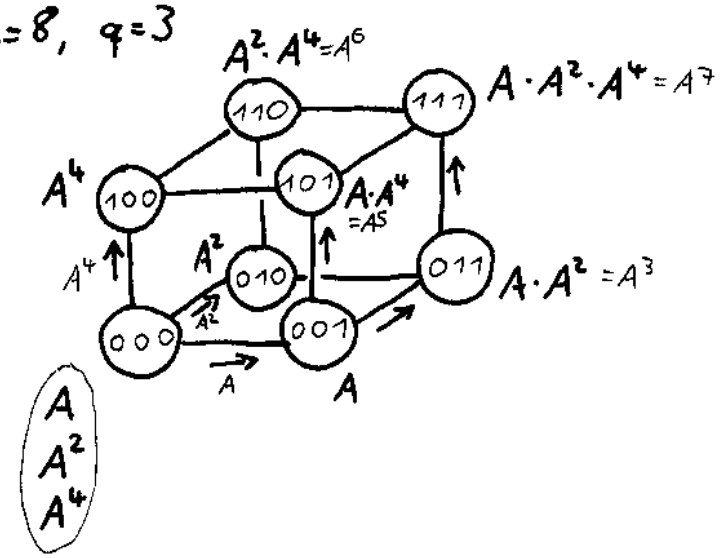
Sei $i = i_{q-1} \dots i_0$, falls $q = \log n$

Algorithmus:

Lade A nach H_0
und broadcaste A zu allen H_i mit $i_0 = 1$.

for $j = 1$ to $q-1$ do
quadriere Inhalt von H_0 ($\rightarrow A^{2^j}$)
broadcaste Inhalt von H_0 zu allen H_i mit $i_j = 1$
multipliziere in den H_i mit $i_j = 1$ die aktuelle Matrixpotenz mit A^{2^j}

Bsp. $n=8$, $q=3$



Aufwandsanalyse

- Matrix-Mult. in Knoten-HCs: $O(\log n)$ DNS
- for-Schleife: $\underbrace{O(\log n)}_{\#\text{Schleifendurchläufe}} \cdot \underbrace{O(\log n)}_{\text{Schleifenrumpf}} = O(\log^2 n)$

Berechnung der gewichteten Matrixsumme aus (*):

- Broadcast des Koeffizientenvektors an alle H_i ($1 \le i < n$): $O(\log n)$
- Broadcast von c_i innerhalb der H_i: $O(\log n)$
- skalare Multiplikation: $O(1)$
- Addition der gemischten Matrizen: $O(\log n)$

$\rightsquigarrow$ insgesamt: $O(\log n)$

$\Rightarrow$ Gesamtaufwand $O(\log^2 n)$ auf n^4 Proz.

2. Berechnung der Koeffizienten des char. Polynoms mit dem Algorithmus von Csanky (1976)

Lemma von de Verrier:

Die Koeff. des char. Polynoms einer Matrix A erfüllen das folgende Gleichungssystem:

$$\begin{pmatrix} 1 & & & & 0 \\ s_1 & 2 & & & \\ s_2 & s_1 & 3 & & \\ \vdots & s_2 & s_1 & \ddots & \\ & & s_2 & \ddots & \ddots \\ s_{n-1} & & & s_2 & s_1 \; n \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = - \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix}$$

Dabei bezeichne s_k die Spur von A^k, $1 \le k \le n$,
d.h. die Summe der Hauptdiagonalelemente
(= Summe der Eigenwerte von A)

$\Rightarrow$ Berechnung der Spuren s_k $(1 \le k \le n)$
+ Dreiecksgl.sys. lösen

$\Rightarrow$ $\log n$ Schritte

26.07.2007 und 28.07.2007

Siehe Anhang A: Methods for Exhaustive Searching

04.07.2007

4. Parallele Suche

Unterscheide

a Entscheidungsprobleme
→ Erfüllen einer Menge von Bedingungen

Bsp. n-Damenproblem

Formulierung als Suche in Baumstrukturen
(state space trees)

Standardverfahren: Backtracking = Depth First Search

Unterscheide:
- Finden einer Lösung
- Finden aller Lösungen

b Optimierungsprobleme
→ zusätzlich: Minimierung/Maximierung einer Zielfkt.

Bsp. 0/1 - Rucksackproblem

Standardverfahren: Branch & Bound
→ Beschneidung des Suchbaums durch Ausschließen von Teilbäumen (Pfaden), deren Lösungen schlechter als die bisher gefundene beste Lösung sind.

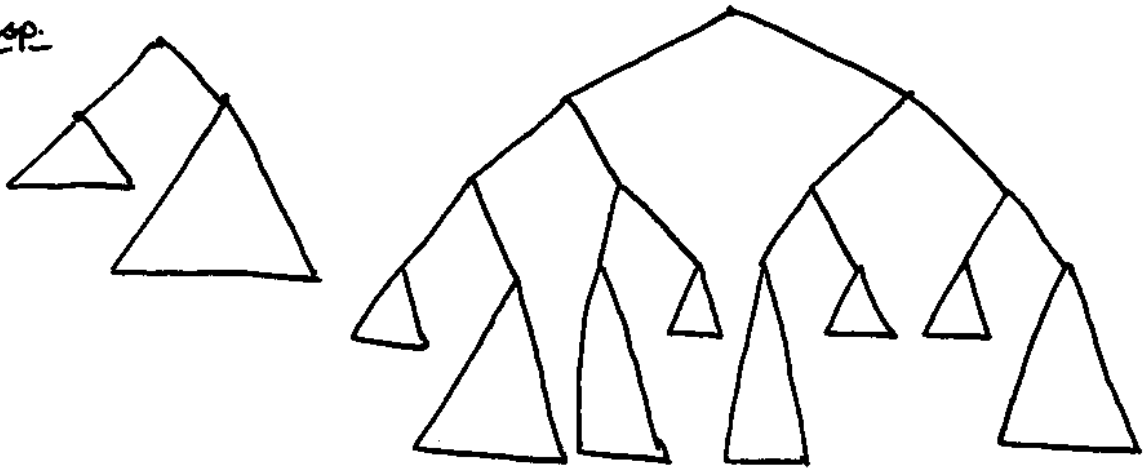
Grundidee paralleler Baumsuchverfahren

Suchbaum parallel durch mehrere Prozessoren durchlaufen

⇒ Mehraufwand
* notwendige Kommunikationen bzw. Konflikte beim Zugriff auf gemeinsame Datenstrukturen
* redundante Arbeit bzw. Leerlaufzeiten bei Lastungleichheit

Eine statische Lastverteilung ist nicht sinnvoll.

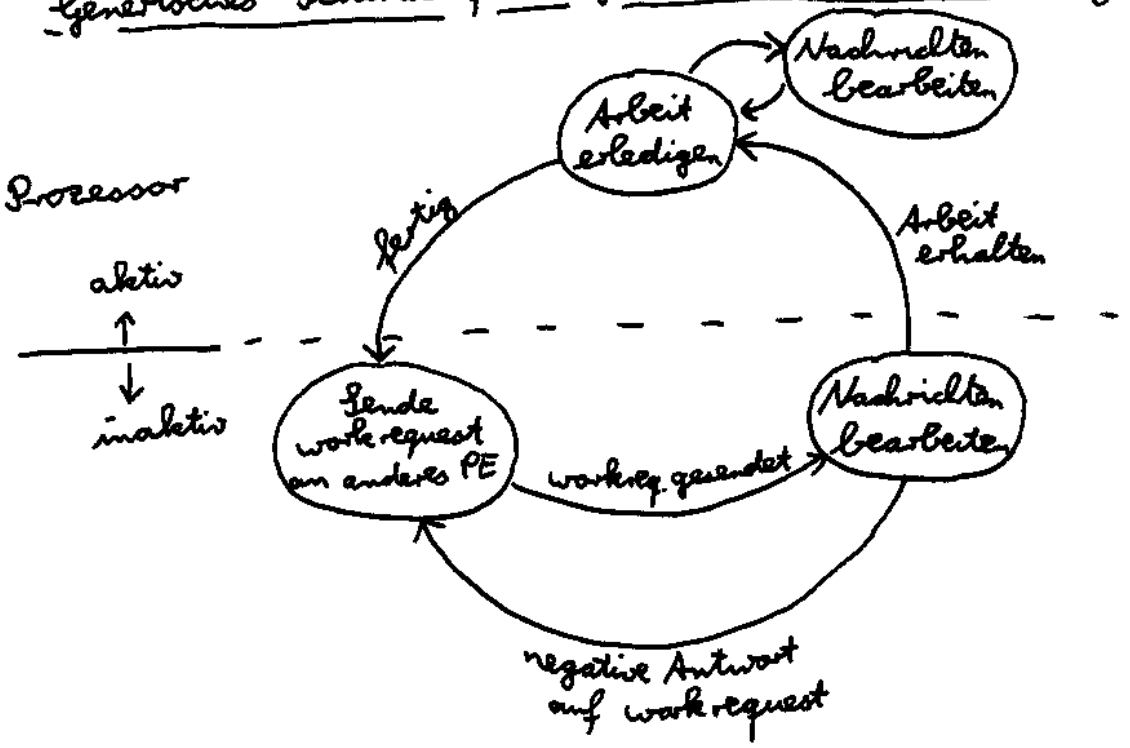
Bsp:



besser: dynamische Lastverteilung

Grundidee: Arbeitslose Prozessoren fordern Arbeit
von zentraler Verwaltung (→ Manager / Master)
oder anderen Prozessoren → Master-Worker-Schema

Generisches Schema für dynamische Lastverteilung



kritische Aspekte

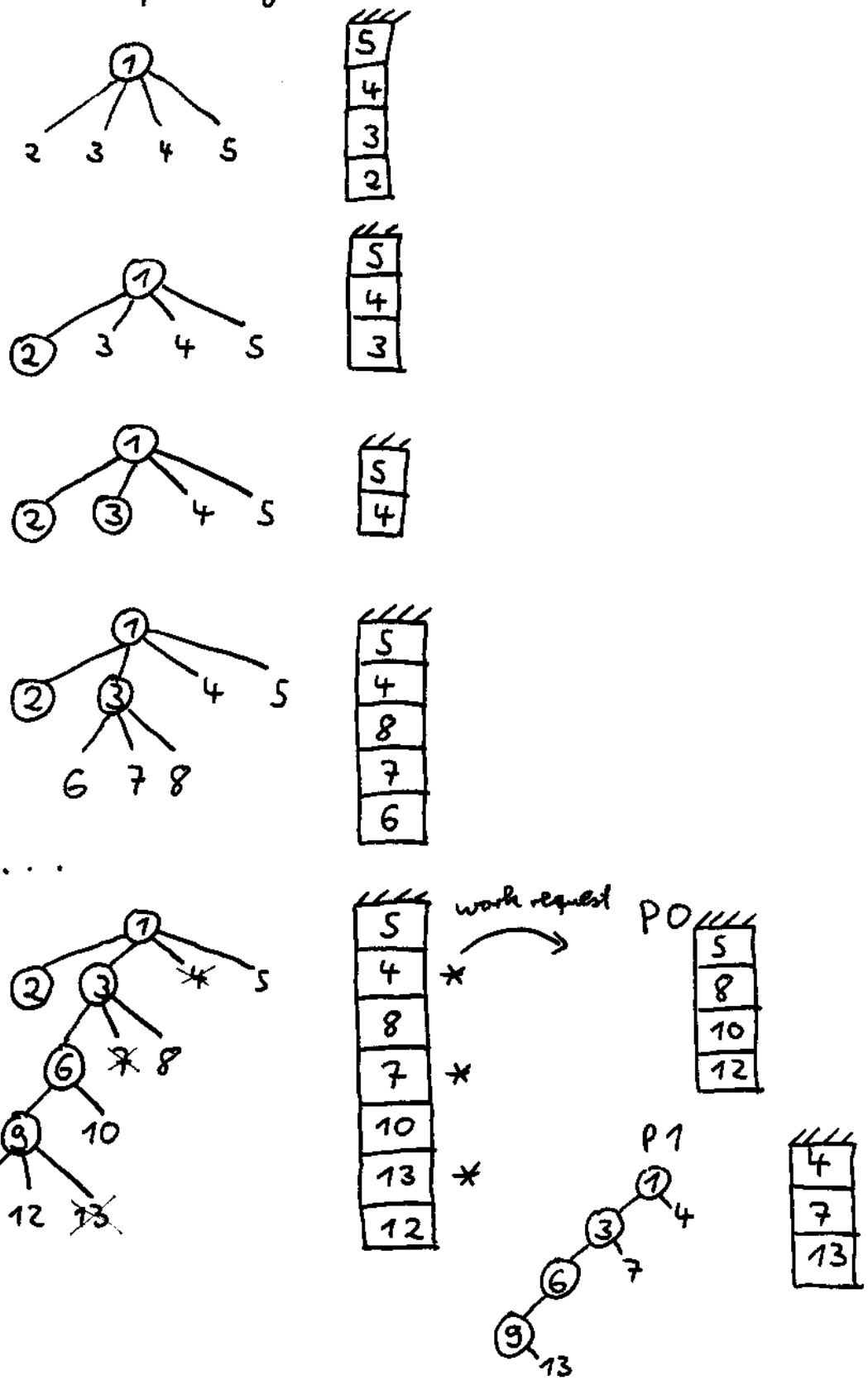
1 Aufteilen der Arbeit bei eintreffender Anfrage arbeitsloser Prozessoren

2 Bestimmung des Proz, an den work request geschickt wird.

ad 1 paralleles Backtracking mit explisiter Stackverwaltung

→ Arbeitsaufteilung durch Stack splitting, meist half split.

Bsp. Stack splitting



Um das Verschicken zu kleiner Arbeitspakete zu vermeiden, werden Knoten jenseits einer vorgegebenen Tiefe (= cutoff depth) nicht mehr verschickt.

05.07.2007

ad 2 Lastbalancierungsstrategien

* Asynchrones Round Robin

Jeder Prozess verwaltet eine lokale Variable req_id mit der Id des Prozesses, der als nächstes nach Arbeit gefragt werden soll. Nach Senden einer Arbeitsanforderung wird req_id um 1 (mod # Proz.) erhöht.
Nachteil: evtl. mehrere Workreq.s an denselben Prozess.

* Globales Round Robin

Prozess 0 verwaltet globale Var. req_id. Arbeitslose Prozessoren müssen vor workrequest Wert von req_id erfragen.
Nachteil: Globale Var. req_id wird schnell zum Engpass.

* Random Polling

Zufällige Auswahl der Adressaten von workrequests.

Problem: Terminationserkennung, falls nach erster gefundener Lösung gestoppt werden soll.
MPI_FINALIZE sollte z.B. von allen Prozessen erst ausgeführt werden, wenn alle Nachrichten empfangen wurden, sonst Laufzeitfehler.

Dijkstras Token-Verfahren

Ziel: Feststellung, ob

1 alle Prozesse ohne Arbeit

2 keine Nachrichten mehr unterwegs

Prozesse und umlaufendes Token erhalten Nachrichtenzähler.

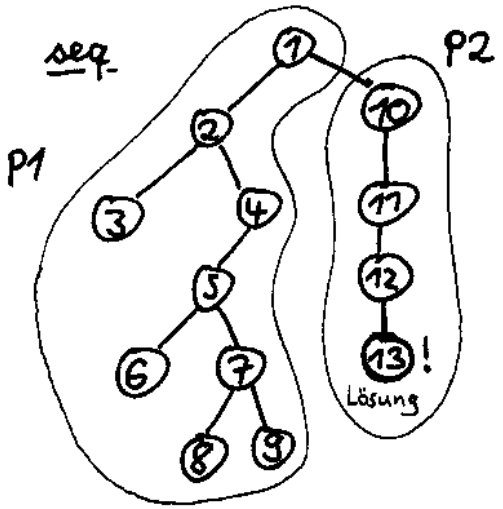
In den Prozessen gibt der Zähler jeweils die # gesendeter Nachrichten minus # empfangener Nachrichten an. Das Token akkumuliert die Zähler der Prozesse.

Jeder Prozess und das Token erhalten die Farbe weiß oder schwarz. Schwarz bedeutet „es ist noch Arbeit geleistet worden."

Siehe Anhang B: Figure 16.8: Dijkstra et al.'s algorithm

Speedup-Anomalien

Bsp. superlinearer Speedup



$T_1 = 13$
$T_2 = 4$
$\Big\} S_2 = 3{,}25 > 2$

sublinearer Speedup

obiges Bsp. mit Lösung im Knoten 6

$$\left.\begin{array}{l} T_1 = 6 \\ T_2 = 6 \end{array}\right\} S_2 = 1 \quad < 2$$

Paralleles Branch & Bound

* Priority queue
* optimale Lösung muss gefunden werden

globale 'priority queue' (PQ) wird schnell zum Engpass
→ Verwaltung einer PQ pro Prozess

Problem: Eventuell bearbeiten Prozesse global ungünstige Teilprobleme.

Siehe Anhang C: Figure 16.16: Parallel branch-and-bound algorithm

5. Graphenalgorithmen

5.1 Grundbegriffe

Def. 5.1 Ein endlicher Graph $G = (V, E)$ besteht aus einer endlichen Knotenmenge V (vertices) und einer endlichen Menge E von Kanten (edges), $E \subseteq V \times V$.

Def. 5.2 (Adjazenzmatrix)
Sei $G = (V, E)$ mit $V = \{v_0, \dots, v_{n-1}\}$.
Die Einträge a_{ij}, $0 \le i,j \le n-1$, der $n \times n$-Adjazenzmatrix von G sind def. durch

$$a_{ij} := \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$

Ein Graph heißt ungerichtet, falls zu jedem $(v, v') \in E$ auch $(v', v) \in E$, ansonsten gerichtet.

G heißt gewichtet, falls jeder Kante mittels einer Gewichtsfkt. $w: E \to \mathbb{R}_0^+$ eine nicht-negative reelle Zahl zugeordnet ist.

w kann zu $\tilde{w}: V \times V \to \mathbb{R}_0^+ \cup \{\infty\}$ ergänzt werden:

$$\tilde{w}(v_i, v_j) = \begin{cases} w(v_i, v_j) & \text{falls } (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases}$$

Eine Folge von Kanten $(v_{i_1}, v_{i_2})(v_{i_2}, v_{i_3}) \dots (v_{i_k}, v_{i_{k+1}})$ heißt

- Pfad, falls alle Knoten $v_{i_1}, \dots, v_{i_{k+1}}$ verschieden sind.

- Zykel, falls $v_{i_{k+1}} = v_{i_1}$ und
  $v_{i_1}, \dots, v_{i_{k+1}}$ voneinander verschieden sind.

- Weg, falls alle Kanten voneinander verschieden sind

Ein Graph heißt azyklisch, falls er keinen Zykel enthält.
Ein Graph (V', E') heißt Subgraph von $G=(V, E)$, falls $V' \subseteq V$ und $E' \subseteq E$.
Ein ungerichteter Graph heißt zusammenhängend, falls zu je zwei Knoten v_i und v_j ein Pfad von v_i nach v_j existiert.

11.07.2007

5.4.2. Bestimmung von Zusammenhangskomponenten

Siehe auch Anhang D: Zusammenhangskomponenten.

Zusammenhangskomponenten:
minimale Teilmenge zusammenhängender Subgraphen

1. Verfahren: Zurückführung auf Matrixmultiplikation.

Die $n \times n$ - Zusammenhangsmatrix

$$C = (c_{ij})_{0 \le i,j < n}$$

wird def. durch

$$c_{ij} = \begin{cases} 1 & \text{falls } v_i \text{ und } v_j \text{ durch einen Weg} \\ & \text{der Länge} \geq 0 \text{ verbunden sind} \\ 0 & \text{sonst} \end{cases}$$

C ergibt sich als reflexiver, transitiver Abschluss der Adjazenzmatrix unter der Boolesschen Matrixmultiplikation, bei der als Multiplikation die

Konjunktion und als Addition die Disjunktion verwendet wird.

Statt $c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}$ wird

$$c_{ij} = \bigvee_{k=0}^{n-1} (a_{ik} \wedge b_{kj})$$ berechnet.

Anstelle der Adjazenematrix wird die auf der Diagonalen modifizierte Matrix B verwendet:

$$B = (b_{ij})_{0 \le i,j \le n-1}$$

$$b_{ij} = \begin{cases} a_{ij} & \text{falls } i \neq j \\ 1 & \text{falls } i = j \end{cases} \Leftarrow \text{reflexiver Abschluss}$$

Für die Matrix B gilt:

$$b_{ij} = \begin{cases} 1 & \text{falls es einen Weg der Länge} \\ & 0 \text{ oder } 1 \text{ von } v_i \text{ nach } v_j \text{ gibt} \\ 0 & \text{sonst} \end{cases}$$

Für die Einträge der k-ten Potenz von B mit Boolescher MM gilt

$$(b_{ij})^k = \begin{cases} 1 & \text{falls es einen Weg der Länge} \le k \\ & \text{von } v_i \text{ nach } v_j \text{ gibt} \\ 0 & \text{sonst} \end{cases}$$

Satz 6.1 Für die Zusammenhangsmatrix C gilt:
$C = B^{n-1}$, wobei n die Anzahl der Knoten sei.

Beweis: Falls v_i und v_j durch einen Weg verbunden sind, existiert auch ein Weg der Länge $\le n-1$. Würde es nur Wege der Länge $> n-1$ geben, so würde mindestens ein Knoten mehrfach durch-

laufen und der Weg würde einen Zykel enthalten, der entfernt werden könnte.

Sei o.B.d.A. $n-1$ Zweierpotenz.
Dann kann C in $\log(n-1)$ Booleschen MM berechnet werden.
Mit Hypercube-MM hätte man einen parallelen Aufwand von $O(\log^2 n)$ auf n^3 Proz.

Algorithmus von Hirschberg (1976)

Siehe auch Anhang D: Hirschberg-Algorithmus

Grundidee: Zusammenfassen von verbundenen Knoten zu Superknoten, bis jeder Superknoten einer Zusammenhangskomponente entspricht.

Modell: CREW-PRAM
Komplexität: $O(\log^2 n)$ mit $O(n^2)$ Proz.
n = # Knoten im Graphen

Eingabe: Adjazenzmatrix A, $V = \{1, \dots, n\}$
Ausgabe: Vektor C der Länge n, so dass $c(i) = c(j) = k$, falls i und j in derselben Zshgskomp. und k ist der „kleinste" Knoten der Zshgskomp. = Index

Der Algorithmus iteriert 3 Phasen $\log n$-mal.

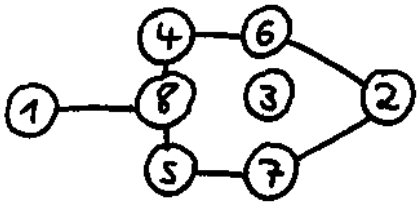
1.) Finde zu jedem Knoten den benachbarten Superknoten mit kleinstem Index.

2. Verbinde die Wurzel jedes Superknotens (= Knoten mit kleinstem Wert) mit der Wurzel des benachbarten Superknoten, mit dem kleinsten Index

3. Alle neu verbundenen Superknoten werden zu einem neuen Superknoten zusammengefasst.

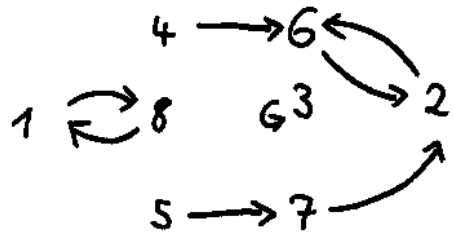
12.07.2007

Bsp.:



i	1	2	3	4	5	6	7	8
C(i)	1	2	3	4	5	6	7	8
B(i)=T(i)	8	6	3	6	7	2	2	1

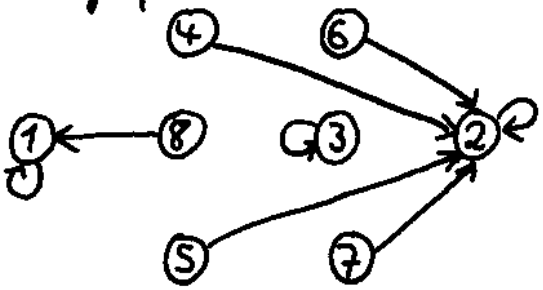
⇝ T-Graph



nach Phase 3 (1. Iteration)

	1	2	3	4	5	6	7	8
T(i)	1	2	3	2	2	6	6	8
C(i)	1	2	3	2	2	2	2	1

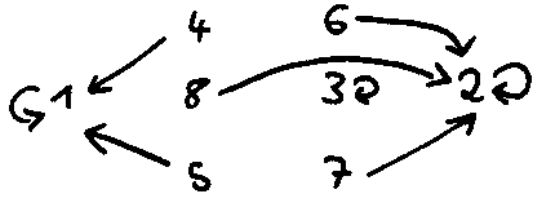
⇝ C-Graph



2. Iteration

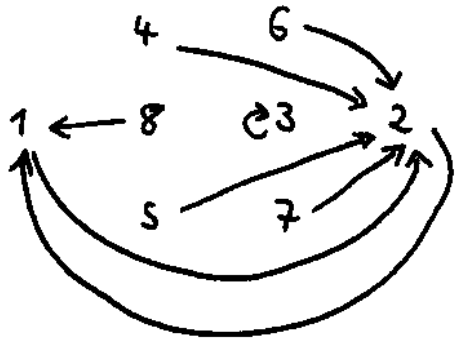
1. Phase	1	2	3	4	5	6	7	8
T(i)	1	2	3	1	1	2	2	2

→ T-Graph



2. Phase

T(i)	2	1	3	2	2	2	2	1



3. Phase

C(i)	1	1	3	1	1	1	1	1

5.3 Minimal spannende Bäume

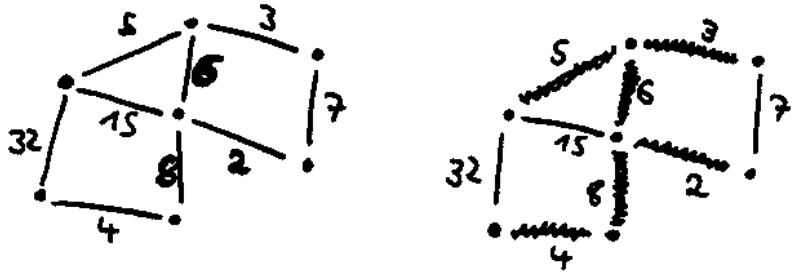
Def. Ein Baum ist ein zusammenhängender, ungerichteter, azyklischer Graph.

Ein spannender Baum eines Graphen G ist ein Subgraph, der alle Knoten von G umfasst und ein Baum ist.

In einem gewichteten Graphen ist ein minimal spannender Baum (MST) ein spannender Baum mit

der minimalen Summe von Kantengewichten.

Bsp.:



Falls $|V| = n$, so hat ein MST nach Def. $n-1$ Kanten.

Da jede der potentiellen $\frac{n(n-1)}{2}$ Kanten mindestens einmal betrachtet werden muss, ist die untere Grenze der Laufzeit eines seq. Alg. zur Bestimmung eines MST $\Omega(n^2)$.

klassische Verfahren:

1956	Kruskal	$O(n^2)$
1957/59	Prim - Dijkstra	$O(n^2)$
1977	Sollin	$O(n^2 \log n)$

Siehe Anhang E: Algorithmus von Prim, Algorithmus von Sollin

7. Verteilte Algorithmen

parallele Alg.	vs.	verteilte Alg.
mehrere, oft gleichartige Prozesse		mehrere Prozesse
Kommunikation: gem. Speicher oder Nachrichten		Kommunikation über Nachrichten
Zweck: Beschleunigung		Zweck: Kooperation

Phänomene verteilter Berechnungen

* Mehrere Prozesse können niemals gleichzeitig beobachtet werden
  ↝ Aussagen über den globalen Zustand sind schwierig

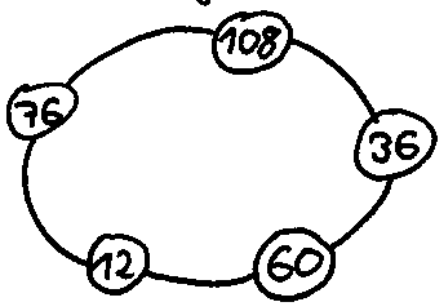
typische Probleme:

Siehe auch Anhang F: Verteilte Terminierung

* Terminationserkennung
* Schnappschussproblem
* Deadlockerkennung
* Leader Election ...

18.07.2007

Bsp. verteilte Berechnung des gg^T

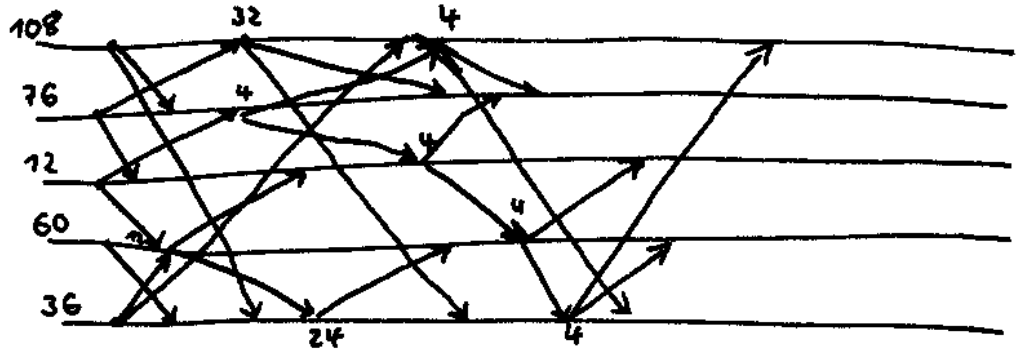


Ansatz:
- Initial Nachbarn informieren
- danach nur auf Nachrichten reagieren
- neue Erkenntnisse an alle Nachbarn senden

P_i : { Nachricht $\langle y_i \rangle$ trifft ein }

atomar
- if $y_i < M_i$ then
  - $M_i := \text{mod}(M_i - 1, y_i) + 1$
  - sende $\langle M_i \rangle$ an alle Nachbarn
- fi

Visualisierung einer mögl. Berechnung



Abstraktes Modell verteilter Berechnungen

Algorithmen als Folge von Aktionen

zwei Arten von Aktionen:

* interne Aktionen

I_p : { Bedingung }
- lokale Zustandstransf
- oder Nachrichten senden

M_p : { Nachricht $\langle M \rangle$ trifft ein }
- lokale Zustandstransf.
- oder Nachrichten senden

7.1 Der Echo-Algorithmus (Chang '82)

paralleler Durchlauf eines bel. zsh. Graphen

→ Verteilen Info des Initiators an alle Knoten

→ Durch Echos Rückfluss von Infos möglich

Echo-Nachrichten durchlaufen einen spannenden Baum des Graphen

Grundidee: Ein Prozess sendet Echo (Quittung) erst dann, wenn er für alle von ihm versendeten Nachrichten Quittungen erhalten hat.

Spezifikation über zwei atomare Aktionen:

I_i : { ¬ informed }

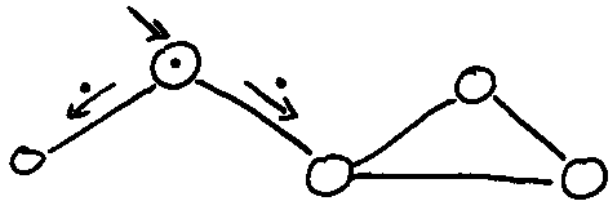
initiator := true
informed := true
N := 0
send <INFO> to neighbours

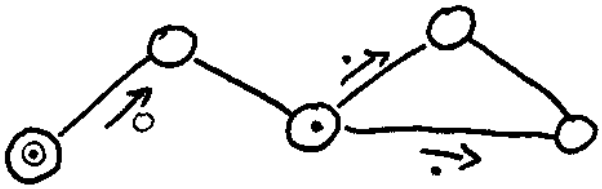
X_i : { receive <INFO> or <ECHO> from p }

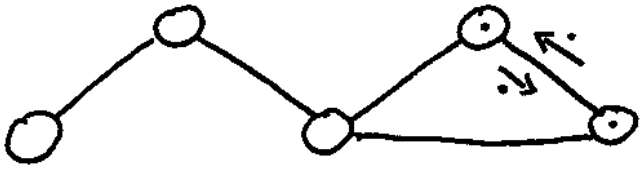
if ¬ informed
  then informed := true „rot werden"
    N := 0; Pred := p;
    send <INFO> to neighbours \ {p}
fi

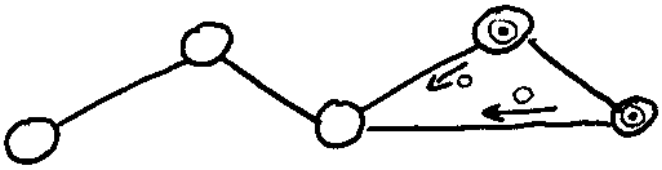
N := N + 1;
if N = |neighbours|
  then informed := false „grün werden"
    if initiator then 'terminated'
    else send <ECHO> to Pred
    fi
fi

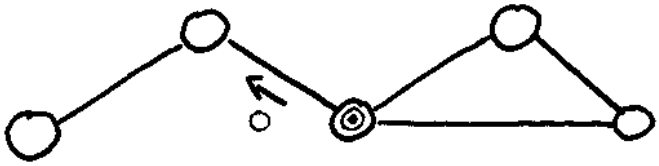
Bsp.:

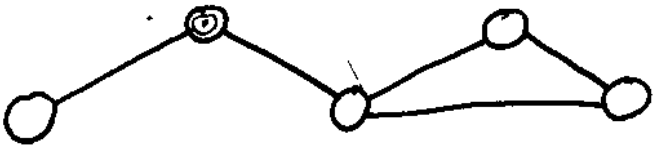












wichtig:
- 1 Initiator
- über jede Graphkante laufen genau zwei Nachrichten
- 2 Wellen

  Hinwelle „rot werden"

  Rückwelle „grün werden"

Anhang A:

①

Methods for Exhaustive Searching

Yolanda Ortega-Mallén

Dept. Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Contents

The 8-Queens Problem

Try every possibility:

- Each queen is tried at each square:
  Comb(64,8)= 4.426.165.368 !!!
- Queens must be set in different rows:
  8^8= 16.777.216
- Queens must be set in different rows and different columns:
  8!= 40.320
- At stage i set a queen in the i-row.
  If it generates a threat then abort this solution:
  $\sum_{i=1}^{8} 8^i = (8^{8+1} - 1)/(8 - 1) = 19{,}173{,}961$ (partial) states,
  only 15.721 are analyzed, just 2.057 are found *promising*

Solution Spaces

The last recourse to solve certain problems is the exhaustive searching of the whole space of possible solutions.

- Impracticable if the solution space is huge $\longrightarrow$ structure the searching to discard big blocks of unsatisfactory solutions.
- Constructing the solution by steps: $(x_1, \ldots, x_n)$
  $x_i \in S_i$ represents the decision taken at i-step.
  The solution must satisfy/optimize a criterion function
  - Explicit restrictions : define the sets S_i.
  - Implicit restrictions : relations among the components x_i to satisfy the criterion function.

Exploration Trees

The set of tuples satisfying the explicit restrictions is structured as an exploration tree.

- at each level of the tree the decision of the corresponding step is taken.
  - State node: tuple (partial or complete) satisfying the explicit restrictions.
  - Solution node: complete tuple satisfying the explicit and implicit restrictions.
- Pruning the tree: obtain a promising test from the criterion function to determine if a state node cannot lead to a solution $\Rightarrow$ discard this branch of the exploration tree.

Example: The n-Queens Problem

- Solutions: n-tuples $(x_1, \ldots, x_8)$
  where x_i= column occupied by queen in row i
- Explicit restrictions: $x_i \in [1..n]$
- Implicit restrictions:
  Queens share neither column nor diagonal
  $i \neq j \Rightarrow ((x_i \neq x_j) \wedge |x_i - x_j| \neq |i - j|)$
- Exploration tree:
  Permutations tree with $\sum_{i=1}^{n} n^i$ state nodes
- Solution nodes: at leaves

Exploration Tree Traversal

The exploration tree is traversed in a certain order:
find the first solution, find every solution, find the best solution.

- For each node of the tree its children are generated.
  - Alive node: there are still children to be generated.
  - Expanding node: its children are being generated.
  - Dead node: cannot be expanded
    - It does not satisfy the promising test, or
    - All its children have been generated.
- Possible traversals:
  - Backtracking: depth-first traversal → alive nodes are managed with an stack ⇒ Simple and space efficient.
  - Branch-and-Bound: the *most promising* alive node is expanded
    → alive nodes are managed with a priority queue.

Time Costs

- The worst case cost is in the order of the solution space size
  ⇒ usually is at least exponential.
- Only practical if the pruning is effective:
  - detect as many nonpromising nodes as posible
    → the upper in the tree the better
  - moderate cost of computation
    → the pruning must compensate the effort
- Very difficult to estimate theoretically beforehand
  ⇒ empirical measures

Backtracking: The General Recursive Scheme

```
proc backtracking(in/out sol : tuple, in k : integer)
    prepare-run-level(k)
    while ¬last-child-level(k) do
        sol[k] := next-child-level(k)
        if is-solution?(sol, k) then
            handle-solution(sol)
        else
            if is-promising?(sol, k) then
                backtracking(sol, k + 1)
            eif
        eif
    ewhile
eproc
```

Backtracking: The n-Queens Problem (I)

```
proc queens-BT(solution[1..n], k)
    for column = 1 to n do
        solution[k] := column
        if ¬threat(solution, k) ∧ k = n { is-solution? } then
            print(solution)
        else
            if ¬threat(solution, k)    { is-promising? } then
                queens-BT(solution, k + 1)
            eif
        eif
    efor
eproc
```

- initial call: queens-BT(*solution*,1)

Backtracking: The n-Queens Problem (II)

Procedure to test threat:

{ there is no threat in *solution*$[1..k-1]$ }
fun threat(*solution*$[1..n], k$) **return** *yes*
    yes := false ; $j := 1$
    while $j \neq k \wedge \neg$*yes* **do**
        yes := (*solution*$[k]$ = *solution*$[j]$)
                $\wedge$ (|*solution*$[k]$ − *solution*$[j]$| = $k - j$)
        $j := j + 1$
    ewhile
efun
{ *yes* $\Leftrightarrow$ there is a threat in *solution*$[1..k]$ }

Backtracking: Saving time by Marking

- The promising test can be too much time consuming.
- Time can be saved with extra memory space. Associate to each node information relative to *partial computations* of the promising test.
- This technique is denominated marking.
- Can be implemented with in/out parameters $\Rightarrow$ minor space increase.

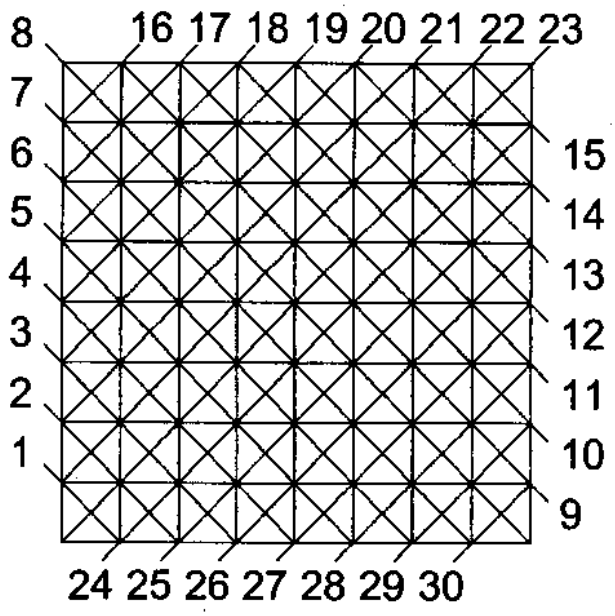
Backtracking with Marking: The n-Queens Problem

Each cell *threatens* one row, one column, and two diagonals:

- Keep a boolean array with the cells that are already threatened.
  - space cost: an array of size $n \times n$.
  - time cost: linear with respect to the size of the array $\Rightarrow$ There is no improvement!
- Keep two boolean vectors representing columns and diagonals that are already threatened.
  $\Rightarrow$ Numbering of diagonals.

Backtracking with Marking: The n-Queens Problem

Diagonals numbering:



The cell $\langle i, j \rangle$ belongs to the $j - i + n$ descending diagonal and to the $i + j + 2n - 2$ ascending diagonal.

Backtracking with Marking: The n-Queens Problem (II)

```
proc queens-BT-M(sol[1..n], k, C[1..n], D[1.,4n − 2])
    for column = 1 to n do
        sol[k] := column
        if ¬C[sol[k]] ∧ ¬D[sol[k] − k + n] ∧ ¬D[sol[k] + k + 2n − 2] then
            { marking }
            C[sol[k]] := true
            D[sol[k] − k + n] := true ; D[sol[k] + k + 2n − 2] := true
            if k = n then print(sol) else queens-BT-M(sol, k + 1, C, D) eif
            { unmarking }
            C[sol[k]] := false
            D[sol[k] − k + n] := false ; D[sol[k] + k + 2n − 2] := false
        eif
    efor
eproc

proc queens(n)
var sol[1..n], C[1..n], D[1.,4n − 2]
    C[1..n] := [false, ..., false]
    D[1.,4n − 2] := [false, ..., false]
    queens-BT-M(sol, 1, C, D)
eproc
```

Backtracking: The Recursive Scheme with Marking

```
proc backtracking-with-marking(in/out sol : tuple, in k : integer,
    in/out m : mark)
    prepare-run-level(k)
    while ¬last-child-level(k) do
        sol[k] := next-child-level(k)
        m := marking(m, sol[k])
        if is-solution?(sol, k) then
            handle-solution(sol)
        else
            if is-promising?(sol, k, m) then
                backtracking-with-marking(sol, k + 1, m)
            eif
        eif
        m := unmarking(m, sol[k])
    ewhile
eproc
```

Backtracking: Optimization Problems

- Keep the best solution found so far, to compare each new solution with it.
- Gain efficiency by keeping also the *value* associated to the best solution.
- Facilitate the computation of the value associated to a solution by storing the (partial) value associated to each partial tuple (marking).
- Extra pruning: when it can be guaranteed that none of the descendants of an expanding node can lead to a better solution than the obtained so far.

Backtracking: The Optimization Recursive Scheme

```
roc backtracking-opt(in/out sol, in k, in/out value, best-sol, best-value)
  prepare-run-level(k)
  while ¬last-child-level(k) do
      sol[k] := next-child-level(k)
      value := update(value, sol, k)
      if solution?(sol, k) then
          if better?(value, best-value) then
              best-sol := sol ; best-value := value
          eif
      else
          if promising?(sol, k) ∧ worth-it?(sol, k, value, best-value) then
              backtracking-opt(sol, k + 1, value, best-sol, best-value)
          eif
      eif
      value := recover(value, sol, k)
  ewhile
proc
```

Backtracking: The 0/1 Knapsack Problem (I)

Alternatives for the Solution Space:

- Stage i: which object should be introduced after having introduced $i-1$ objects?
  - $\Rightarrow$ tuples with the objects introduced so far.
  - Each state node can be considered a solution node.
  - The order in the introduction of objects is irrelevant $\Rightarrow$ An order between objects can be fixed.
- Stage i: should be introduced the i-object?
  - $\Rightarrow$ complete binary tree.
  - Solution nodes are only at the leaves.
  - Explicit restrictions: $x_i \in \{0,1\}$.
  - Implicit restrictions: $\sum_{i=1}^{n} w_i.x_i \leq M$.

Backtracking: The 0/1 Knapsack Problem (II)

```
{ V[1]/W[1] ≥ V[2]/W[2] ≥ ... ≥ V[n]/W[n] }
proc knapsack-BT(W, V, M, sol, k, weight, value, best-sol, best-value)
   sol[k] := 1    { object is taken - no new estimation is calculated }
   weight := weight + W[k] ;  value := value + V[k]    { mark }
   if weight ≤ M then
      if k = n then best-sol := sol ;  best-value := value
      else knapsack-BT(W, V, M, sol, k + 1, weight, value, best-sol, best-value) eif
   eif
   weight := weight − W[k] ;  value := value − V[k]    { unmark }
   sol[k] := 0    { object is not taken - no marking, but new estimation is calculated }
   estimated-value := value + knapsack-greedy(W[k + 1..n], V[k + 1..n], M − weight)
   if estimated-value > best-value then
      if k = n then best-sol := sol ;  best-value := value
      else knapsack-BT(W, V, M, sol, k + 1, weight, value, best-sol, best-value) eif
   eif
eproc

fun knapsack(W[1..n], V[1..n], M) return (best-sol[1..n], best-value)
var sol[1..n]
   weight := 0 ;  value := 0 ;  best-value := −∞
   knapsack-BT(W, V, M, sol, 1, weight, value, best-sol, best-value)
efun
```

Branch-and-Bound

- Backtracking (depth-first-search) can get lost in unbounded depth branches
  $\Rightarrow$ alternative traversal of the exploration tree.
- Try to find a solution more quickly by exploring first the most promising node.
- For optimization problems we define an estimated value function:
  - minimization problems: the estimated value in a node is a lower bound of the value associated to the best solution reachable from the node.
    estimated-cost(X) $\leq$ real-cost(X)
  - When solution nodes are only at leaves usually we have:
    estimated-cost(X) = real-cost(X) for X solution node.
  - It is expected that nodes with a low estimated cost will lead to solutions with a low real cost
    $\Rightarrow$ the alive node with the lowest estimated cost is the most promising.
  - The estimated value provides an extra pruning (as in Backtracking).
  - maximization problems: estimated-benefit(X) $\geq$ real-benefit(X)

Branch-and-Bound: The Scheme for Minimization

```
fun branch-and-bound-min(T : tree) return ⟨best-sol, best-cost⟩
var X, Y : node, Q : priority-queue
   Y := root(T) ; create-empty(Q) ; enqueue(Q, Y)
   best-cost := +∞
   while ¬is-empty?(Q) ∧ estimated-cost(min-elem(Q)) < best-cost do
      Y := min-elem(Q) ; dequeue(Q)
      for for every child X of Y do
         if is-solution?(X) then
            if real-cost(X) < best-cost then
               best-cost := real-cost(X) ; best-sol := X
            elf
         else
            if is-promising?(X) ∧ (estimated-cost(X) < best-cost) then
               enqueue(Q, X)
            elf
         elf
      efor
   ewhile
efun
```

Branch-and-Bound: Optimist-Pessimist (Minimization)

- best-cost = cost corresponding to the best solution obtained so far
  ⇒ only updated when a better solution is found
  ⇒ initialized with the worst possible value → $+\infty$
  ⇒ little effective until a solution is found, thereafter it improves slowly.
- optimist-cost(X) = lower bound for the cost of the best solution that can be ever reached from node X (before estimated-cost(X)).
- pessimist-cost(X) = upper bound for the cost of the best solution that can be ever reached from node X.
- optimist-cost(X) $\leq$ real-cost(X) $\leq$ pessimist-cost(X)
- If the pessimist-cost is used to update the best-cost (even if the solution has still not been found) then the pruning would be more effective.
- The cost of any solution reachable from X can be used as pessimist-cost(X).

Branch-and-Bound: Optimist-Pessimist Scheme (Min.)

```
fun bb-opt-pes-min(T : tree) return ⟨best-sol, best-cost⟩
var X, Y : node, Q : priority-queue
    Y := root(T) ; create-empty(Q) ; enqueue(Q, Y)
    best-cost := pessimist-cost(Y)
    while ¬is-empty?(Q) ∧ optimist-cost(min-elem(Q)) ≤ best-cost do
        Y := min-elem(Q) ; dequeue(Q)
        for for every child X of Y do
            if is-solution?(X) then
                if real-cost(X) ≤ best-cost then
                    best-cost := real-cost(X) ; best-sol := X
                eif
            else
                if is-promising?(X) ∧ (optimist-cost(X) ≤ best-cost) then
                    enqueue(Q, X)
                    if pessimist-cost(X) < best-cost then best-cost := pessimist-cost(X) eif
                eif
            eif
        efor
    ewhile
efun
```

Branch-and-Bound: The 0/1 Knapsack Problem (I)

type $node =$ **reg**
$sol[1..n]$ **of** $\{0,1\}$
k : *nat*
$weight, benefit, optBenefit$: *real*
freg

Branch-and-Bound: The 0/1 Knapsack Problem (II)

```
{ V[1]/W[1] ≥ V[2]/W[2] ≥ ... ≥ V[n]/W[n] }
proc knapsack-BB(W, V, M) return ⟨best-sol[1..n], best-benefit⟩
    Y.k := 0 ; Y.weight := 0 ; Y.benefit := 0 { root is generated }
    ⟨Y.optBenefit, best-benefit⟩ := estimations(W, V, M, Y.k, Y.weight, Y.benefit)
    create-empty(Q) ; enqueue(Q, Y)
    while ¬is-empty?(Q) ∧ (max-elem(Q).optBenefit ≥ best-benefit) do
        Y := max-elem(Q) ; dequeue(Q)
        X.k := Y.k + 1 ; Z.sol := Y.sol
        { try to insert the object in the knapsack }
        if Y.weight + W[X.k] ≤ M then { estimations remain the same as for Y }
            X.sol[X.k] := 1 ; X.weight := Y.weight + W[X.k]
            X.benefit := Y.benefit + V[X.k] ; X.optBenefit := Y.optBenefit
            if X.k = n then { ben-real(X) = ben-opt(X) }
                best-sol := X.sol ; best-benefit := X.benefit
            else
                enqueue(Q, X) { best-benefit cannot be improved }
            elf
        elf
```

Branch-and-Bound: The 0/1 Knapsack Problem (III)

```
        { object is discarded }
        ⟨X.optBenefit, pes⟩ := estimations(W, V, M, X.k, Y.weight, Y.benefit)
        if X.optBenefit ≥ best-benefit then
            X.sol[X.k] := 0 ; X.weight := Y.weight ; X.benefit := Y.benefit
            if X.k = n then
                best-sol := X.sol ; best-benefit := X.benefit
            else
                enqueue(Q, X)
                best-benefit := max(best-benefit, pes)
            eif
        eif
    ewhile
efun
```


Branch-and-Bound: The 0/1 Knapsack Problem (IV)

```
{ V[1]/W[1] ≥ V[2]/W[2] ≥ ... ≥ V[n]/W[n] }
fun estimations(W[1..n], V[1..n], M, k, weight, benefit) return ⟨opt, pes⟩
    empty-space := M − weight ; pes := benefit ; opt := benefit
    j := k + 1
    while j ≤ n ∧ W[j] ≤ empty-space do { object j can be entirely taken }
        empty-space := empty-space − W[j]
        opt := opt + V[j] ; pes := pes + V[j]
        j := j + 1
    ewhile
    if j ≤ n then { there are still objects to try }
        opt := opt + (empty-space/W[j]) * V[j]    { object j is fractionated (greedy solution) }
        { extend to a 0/1 solution }
        j := j + 1
        while j ≤ n ∧ empty-space > 0 do
            if W[j] ≤ empty-space then
                empty-space := empty-space − W[j] ; pes := pes + V[j]
            eif
            j := j + 1
        ewhile
    eif
efun
```

Bibliography

- Estructuras de Datos y Métodos algorítmicos
  Martí Oliet, N., Ortega Mallén, Y., Verdejo López, J.A.
  Pearson Educación, 2004
- Computer Algorithms (3rd edition)
  Horowitz, E., Sahni, S. Rajasekaran, S.
  Computer Science Press, 1998
- Foundations of Algorithms
  Neapolitan, R., Naimipour, K.
  Jones & Bartlett Publishers, 1997

Anhang B:

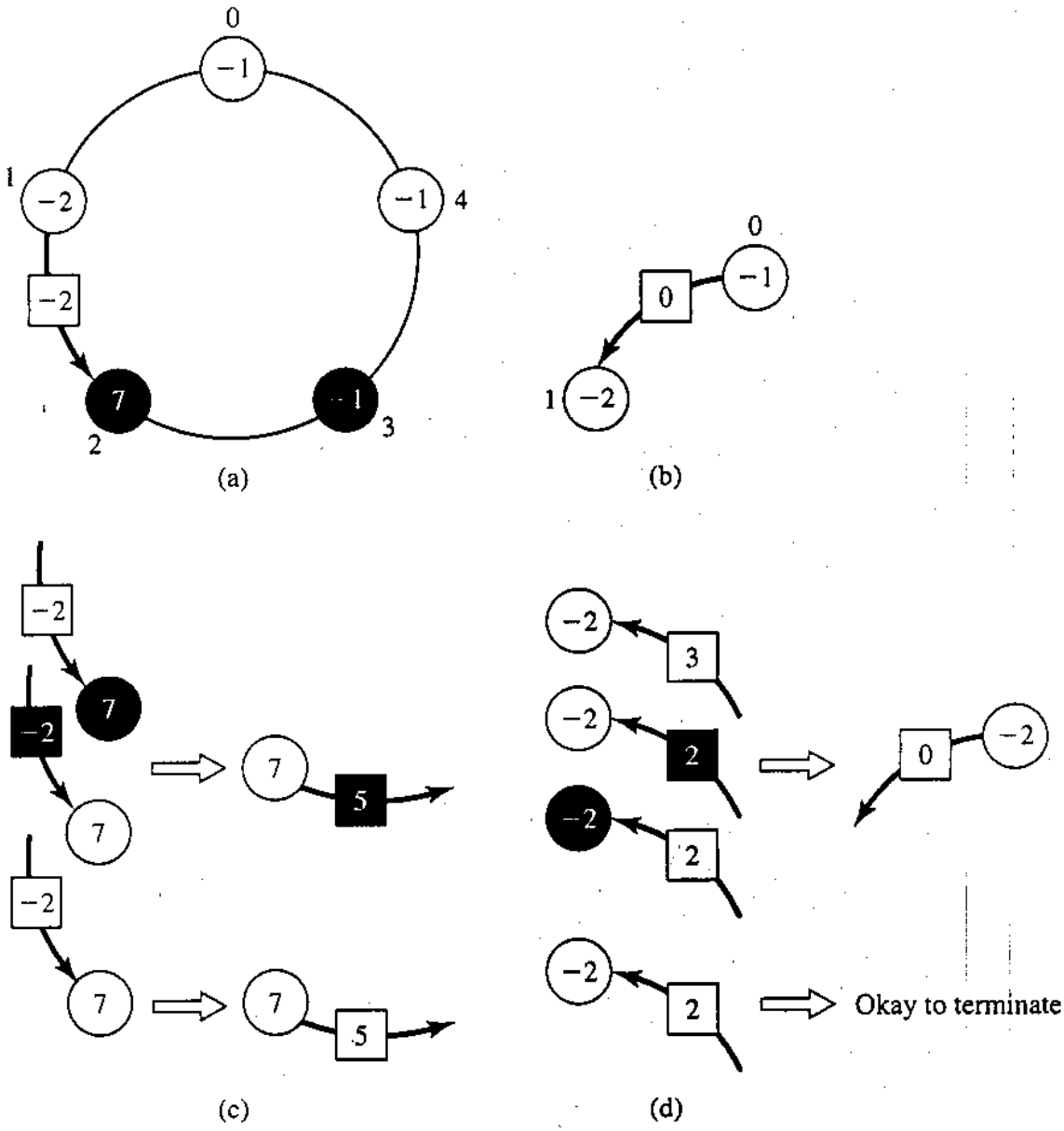


Figure 16.8 Dijkstra et al.'s algorithm to solve the distributed termination problem. (a) A token (square) is passed around a logical ring of processes (circles). (b) Process 0 initiates a probe. (c) An intermediate process modifies the token and passes it along. (d) The token returns to process 0.

```
Constants:
  Comm_Interval — Time between communication steps
  Termination — Tags termination messages
  Token — Tags token message
  Unexamined_Subproblem — Tags message containing unexamined subproblem

Functions:
  Current_Time() — Wall clock time
  Delete_Min() — Delete subproblem with least lower bound from priority queue
  First_Element() — Returns first element from priority queue without deleting it
  Initialize() — Set priority queue size to 0
  Insert() — Insert subproblem into priority queue
  Is_Empty() — Returns true if priority queue is empty
  Lower_Bound() — Returns lower bound associated with unexplored subproblem

Variables:
  color — Process color (for termination detection)
  global_c — Cost of globally best solution found so far
  id — Process rank
  initial — Initial problem
  last_comm — Time of last communication
  local_c — Cost of best solution found so far by this process
  local_s — Best solution found so far by this process
  msg_count — Messages sent minus messages received
  q — Priority queue
  token — Token passed around ring for termination detection
  u — State space tree node
  v — New node with additional constraint

Parallel Best-First Branch and Bound (minimization):
Initialize (q)
if id = 0 then
  Insert (q, initial)
  token.c ← ∞
  token.color ← WHITE
  token.count ← 0
  Send token to successor process
endif
local_c ← ∞
best_soln ← ∞
last_comm ← Current_Time()
msg_count ← 0
color ← WHITE
repeat
  if Is_Empty(q) or (Current_Time()−last_comm > Comm_Interval) then
    BandB_Communication()
    last_comm ← Current_Time()
  else if not Is_Empty(q) then
    u ← Delete_Min(q)
    if Lower_Bound(u) < best_c then
      color ← BLACK
      if u is a solution then
        if Lower_Bound(u) < global_c then
          local_s ← u
          local_c ← Lower_Bound(local_s)
        endif
```

Figure 16.16 Parallel branch-and-bound algorithm.

```
        else
          for i ← 1 to Possible_Constraints(u) do
            Add constraint i to u, creating v
            if Lower_Bound(v) < global_c then
              Insert(q, v)
            endif
          endfor
        endif
      endif
    endif
  forever

  BandBCommunication():
  if there is a pending message with a Termination tag then Halt endif
  if there is a pending message with a Token tag then
    Receive message containing token
    if local_c < token.c then
      token.c ← local_c
      token.s ← local_s
    endif
    if token.c ≤ Lower_Bound(First_Element(q)) then Initialize(q) endif
    global_c ← token.c
    if id = 0 then
      if (color = WHITE) and (token.color = WHITE) and
          (token.count + msg_count = 0) then
        Send messages with a Termination tag to all other processes
        Halt
      else
        token.color ← WHITE
        token.count ← 0
      endif
    else
      if color = BLACK then token.color ← BLACK
      token.count ← token.count + msg_count
    endif
    Send token to successor
    color ← WHITE
  endif
  while there are pending messages with tag Unexamined_Subproblem do
    Receive message with unexamined subproblem u
    msg_count ← msg_count − 1
    color ← BLACK
    if Lower_Bound(u) < global_c then Insert (q, u)
  endwhile
  if there is more than one unexamined subproblem in q then
    Send unexamined subproblem to another process
    msg_count ← msg_count + 1
    color ← BLACK
  endif
  return
```

Figure 16.16 (contd.) Parallel branch-and-bound algorithm.

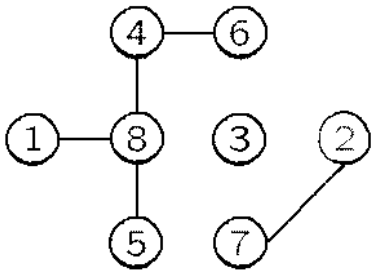
Bestimmung der Zusammenhangskomponenten

Zur Zusammenhangskomponente eines Knoten v_k gehören alle Knoten v_j mit $c_{kj} = 1 = c_{jk}$.

Konstruiere daher die Matrix D mit

$$d_{jk} = \begin{cases} v_k & \text{falls } c_{jk} = 1 \\ 0 & \text{sonst.} \end{cases}$$

Beispiel:



Adjazenzmatrix

	1	2	3	4	5	6	7	8
1								1
2							1	
3								
4						1		1
5								1
6				1				
7		1						
8	1			1	1			

Zusammenhangsmatrix

	1	2	3	4	5	6	7	8
1	1			1	1	1		1
2		1					1	
3			1					
4	1			1	1	1		1
5	1			1	1	1		1
6	1			1	1	1		1
7		1					1	
8	1			1	1	1		1

Matrix D

	1	2	3	4	5	6	7	8
1	1			4	5	6		8
2		2					7	
3			3					
4	1			4	5	6		8
5	1			4	5	6		8
6	1			4	5	6		8
7		2					7	
8	1			4	5	6		8

Pseudocode Hirschberg-Algorithmus

Iteriere $\lceil \log n \rceil$-mal die folgenden 3 Phasen:
Initialisierung des C-Vektors:
$C(i) := i$ für alle $1 \leqslant i \leqslant n$.

Phase 1:

for all vertices i in parallel do

$$T(i) = \begin{cases} \min_j\{\, C(j) \mid & \text{falls existent} \\ \qquad A(i,j) = 1, & \\ \qquad C(j) \neq C(i)\} & \\ C(i) & \text{sonst} \end{cases}$$

Phase 2:

for all vertices i in parallel do

$$T(i) = \begin{cases} \min_j\{\, T(j) \mid & \text{falls existent} \\ \qquad C(j) = i, & \\ \qquad T(j) \neq i\} & \\ C(i) & \text{sonst} \end{cases}$$

Phase 3:

for all vertices i in parallel do $B(i) = T(i)$
repeat $\log n$ times
    for all vertices i in parallel do
        $T(i) = T(T(i))$
for all vertices i in parallel do
    $C(i) = \min\{B(T(i)), T(i)\}$

Detaillierung und Analyse von Phase 1/2

Phase 1 und 2 sind ähnlich,
hier Detaillierung von Phase 1:

1(a) for all i, j $(1 \leqslant i, j \leqslant n)$ in parallel do

$Temp(i,j) :=$ if $A(i,j) = 1$ and $C(i) \neq C($
then $C(j)$ else ∞

1(b) for all i $(1 \leqslant i \leqslant n)$ in parallel do

$Temp(i,1) := \min_{1 \leqslant j \leqslant n}\{Temp(i,j)\}$

1(c) for all i $(1 \leqslant i \leqslant n)$ in parallel do

$T(i) :=$ if $Temp(i,1) = \infty$ then $C(i)$
else $Temp(i,1)$

∞ repräsentiert eine Zahl $> n$.

Aufwandsanalyse von Phase 1 und 2:

1/2(a)	$O(1)$	mit $O(n^2)$ Proz.
1/2(b)	$O(\log n)$	mit $O(n)$ Proz.
1/2(c)	$O(1)$	mit $O(n)$ Proz.

$\curvearrowright$ $O(\log n)$ mit $O(n^2)$ Proz.

Begründung für Phase 3

Der T-Graph, der in Phase 2 gebildet wird, besitzt in jedem neu zu bildenden Superknoten eine Schleife aus 2 Kanten, wobei der kleinste Knoten des neuen Superknotens einer der beiden Schleifenknoten ist.

Da jeder Superknoten aus höchstens n Knoten besteht, kommt man beim Durchlaufen des T-Graphen nach n Schritten in die Schleife und ist somit höchstens einen Schritt vom Minimum entfernt.

Gesamtaufwand:

Phase 1/2: $O(\log n)$ mit $O(n^2)$ Proz.

Phase 3:	3(a)	$O(1)$	mit $O(n)$ Proz.
	3(b)	$O(\log n)$	mit $O(n)$ Proz.
	3(c)	$O(1)$	mit $O(n)$ Proz.

$\curvearrowright O(\log n)$ mit $O(n)$ Proz.

$\log n$ Iterationen: $\boxed{O(\log^2 n) \text{ mit } O(n^2) \text{ Proz.}}$

Algorithmus von Prim

Invariante:
Alle Knoten v_i außerhalb eines Baumes T_i kennen den Knoten in T_i mit minimalem Abstand zu ihnen.
$c(v_i) :=$ nächster Nachbar von v_i in T_i

Initialisierung: Ein Anfangsknoten v_0 wird festgelegt. T_0 besteht nur aus v_0.
$c(v_i) := v_0$ für alle $v_i \neq v_0$

Pseudo-Code:

for $i := 1$ to $n-1$ do

i) Suche unter den Knoten außerhalb von T_i einen Knoten $\bar{v}$ mit $w(\bar{v}, c(\bar{v}))$ minimal und füge $\bar{v}$ mit der Kante $(\bar{v}, c(\bar{v}))$ zu T_i hinzu

ii) Die Knoten v_j, die außerhalb von T_i bleiben, berechnen $c(v_j)$ neu
$c(v_j) :=$ if $w(v_j, \bar{v}) < w(v_j, c(v_j))$
then $\bar{v}$ else $c(v_j)$

Analyse

sequentielle Laufzeit:

$$T(n) = 1 + O(n) + (n-1) * O(n) = O(n^2)$$

Implementierung auf CREW-PRAM:

n Prozessoren mit
1-1-Zuordnung Knoten $\leftrightarrow$ Prozessoren

↷ Initialisierung: $O(1)$
Schleife (i) Minimumbildung: $O(\log n)$
Schleife (ii): $O(1)$

↷ parallele Laufzeit:

$O(n \log n)$ mit $O(n)$ Prozessoren

↷ parallele Kosten: $O(n^2 \log n)$

Durch Re-Scheduling kann Kostenoptimalität erreicht werden.

Algorithmus von Sollin

Arbeitsweise ähnlich zu Hirschberg-Algorithmus

Initialisierung: Wald mit n isolierten Knoten, die als Bäume betrachtet werden

Iteration:

für jeden Baum:
  bestimme die Kante mit dem kleinsten Gewicht, die diesen Baum mit einem anderen verbindet

Alle diese minimalen Kanten werden hinzugefügt. Dabei werden eventuell entstehende Zykel beliebig durchbrochen.

Die Anzahl der Bäume wird pro Iteration mindestens halbiert.
↷ maximal $\lceil \log n \rceil$ Iterationen sind erforderlich

Pro Iteration werden maximal $O(n^2)$ Vergleiche zur Bestimmung der minimalen Kanten benötigt.

↷ sequentielle Laufzeit: $O(n^2 \log n)$

Sequentieller Pseudo-Code

Parameter: n = Anzahl der Knoten

Globale Variablen:

closest[]	Abstand zu nächstem Baum
edge[]	Kante zu nächstem Baum
i	
T	MST (als Kantenmenge)
v,w	Endpunkte der aktuellen Kante
weight[]	Kantengewichte
Baum[]	Wald als Knotenmengen

Hilfsfunktionen:
FIND(v) liefert zu einem Knoten, den Baum, in dem v enthalten ist.

UNION(v, w) vereinigt die Bäume, in denen zwei Knoten v und w enthalten sind.

Pseudo-Code:

```
begin
    for i := 1 to n do Baum[i] := {v_i} od
    T:= ∅
    while |T| < n − 1 do
        für jeden Baum i setze closest[i] := ∞
        für jede Kante (v, w) do
            if FIND(v) ≠ FIND(w) then
              if weight[{v, w}] < closest[FIND(v)]
              then closest[FIND(v)] := weight[{v, w
                   edge[FIND(v)] = {v, w}
              fi
            fi
        od
        für jeden Baum i do
            {v, w} := edge[i]
            if FIND(v) ≠ FIND(w) then (*)
              T := T ∪ {v, w}          (*)
              UNION(v, w)              (*)
            fi
        od
    od
end
```


Parallelisierung

Die äußere While-Schleife kann wegen Abhängigkeiten nicht parallelisiert werden.

Die erste innere for-Schleife kann parallel ausgeführt werden. Bei p Prozessoren bearbeitet jeder Prozessor $\frac{1}{p}$ Bäume.

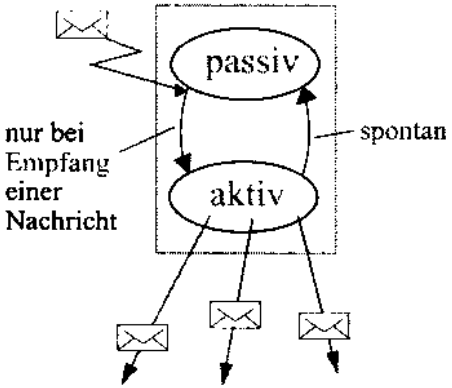
Zweite innere Schleife: Jeder Prozessor kann für einen Anteil an Knoten jeweils die von diesen Knoten ausgehenden Kanten untersuchen.

Dritte innere Schleife: Unkontrollierte Parallelverarbeitung kann zu Fehlern führen. Hier ist eine Synchronisation auf dem durch $(*)$ markierten kritischen Bereich erforderlich.

Verteilte Terminierung: Problemdefinition

Nachrichtengesteuertes Modell einer vert. Berechnung:

- Prozesse sind *aktiv* oder *passiv*
- Nur aktive Prozesse versenden Nachrichten
- Prozeß kann "spontan" passiv werden
- Prozeß wird durch ankommende Nachricht reaktiviert



Problem:

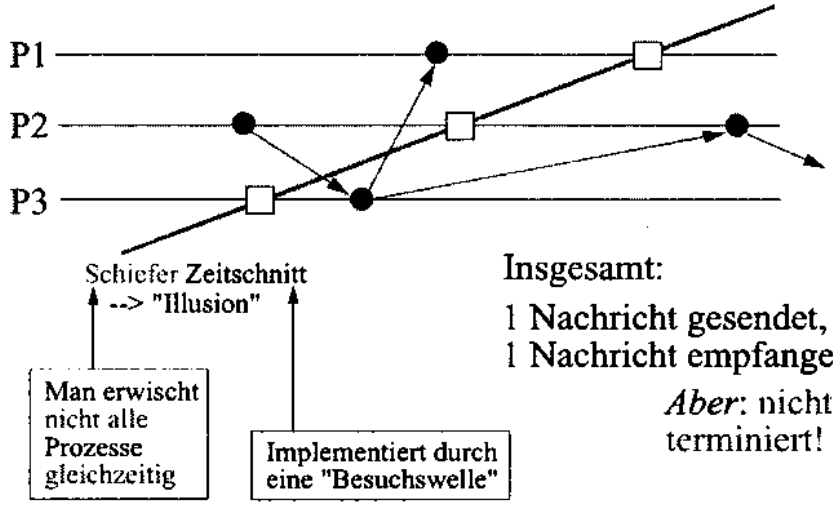
Feststellen, ob (zu einem Zeitpunkt)
- alle Prozesse passiv sind
- keine Nachricht unterwegs ist

- "Globales Prädikat"
- "Stabiler Zustand"

Verteilte Terminierung: Lösungen durch Zählen von Nachrichten?

- Genügt das (verteilte) Zählen von gesendeten und empfangenen Nachrichten?

- Einfaches Zählen genügt nicht, Gegenbeispiel:



Insgesamt:
1 Nachricht gesendet,
1 Nachricht empfangen.

Aber: nicht terminiert!

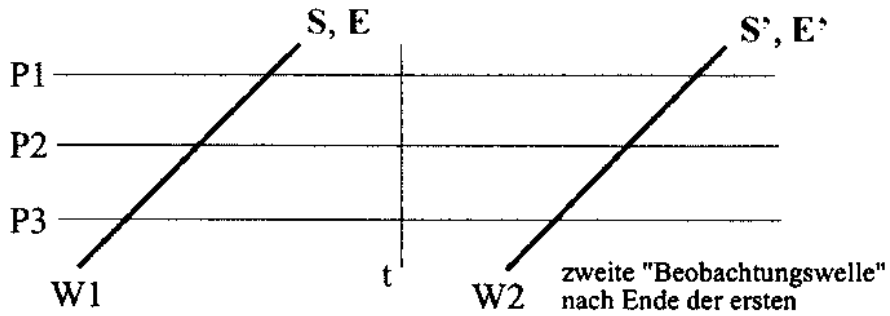
Ursache (informell):

- Nachricht aus der "Zukunft"
  - kompensiert die Zähler
- Inkonsistenter Schnitt
  - ist nicht äquivalent zu einem senkrechten Schnitt

Lösung durch "Ursachenvermeidung"? Ideen vielleicht:

- Nachrichten aus der Zukunft *vermeiden* oder zumindest *erkennen*?
- Senkrechten Schnitt simulieren durch *Einfrieren* der Prozesse?

Das Doppelzählverfahren



Behauptung: S=E=S'=E' ==> terminiert

Beweis (Skizze; läßt sich auch formalisieren):

S=S' ==> Keine Nachricht zwischen W1, W2 gesendet.
E=E' ==> ———— "" ———— empfangen.
==> Werte bei t = Werte von W1.

Also: S=E ==> zum globalen Zeitpunkt t gilt:

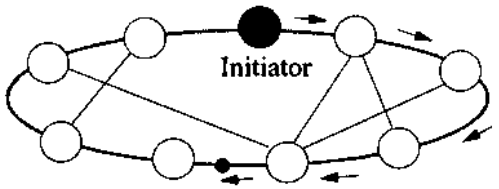
Anzahl gesendeter = Anzahl empfangener Nachrichten
==> zum Zeitpunkt t ist keine Nachricht unterwegs
==> zum Zeitpunkt t terminiert
==> Berechnung war nach W1 terminiert □

Es gelingt also, für einen bestimmten Zeitpunkt eine kausaltreue Beobachtung (als senkrechten Schnitt) im Nachhinein zu rekonstruieren!

Kontrolltopologien

- Die für den Terminierungsalgorithmus benötigten "Wellen" können unterschiedlich realisiert werden:

1.) *Ring / Hamilton'scher Zyklus:*



Kontrollnachricht ("Token") besucht zyklisch zwei Mal alle Prozesse und akkumuliert dabei die Zählerstände.

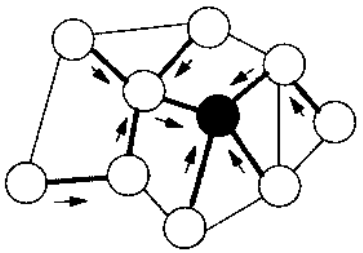
"Logischer" Ring genügt!

- In einem zusammenhängenden ungerichteten Graphen kann ein logischer Ring immer gefunden werden, indem man einen Spannbaum "umfährt".

- Zeit- und Nachrichtenkomplexität: O(n)

2.) *Spannbaum:*

- Vereinfachter Echo-Algorithmus: An den Blättern reflektierte Welle sammelt die Zählerstände in akkumulierender Weise ein.

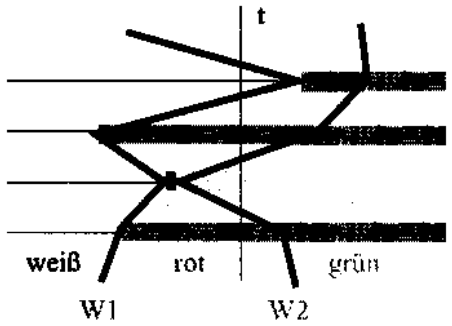


- Auch hier werden *zwei* solche zum Initiator hinfließende Wellen benötigt

- Bei nicht-entartetem Spannbaum: Viele Nachrichten parallel unterwegs ==> bessere Zeitkomplexität!

Echo-Algorithmus für die beiden Wellen des Doppelzählverfahrens?

- Anwendbar für beliebige zusammenhängende Topologien.
- Ausnutzen der beiden "Halbwellen" eines einzigen Laufs des Echo-Algorithmus für die beiden Kontrollrunden!

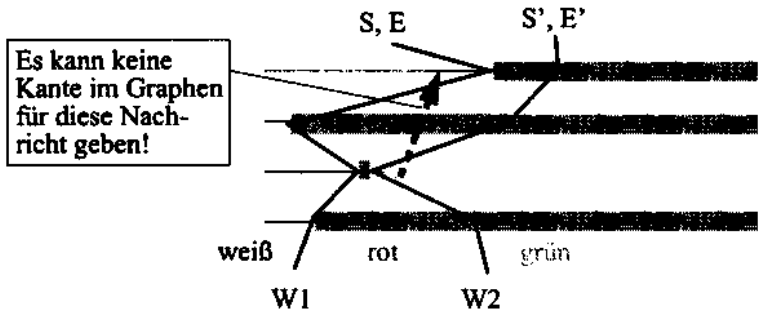


Der Echo-Algorithmus wird als Transportdienst zur Realisierung von zwei Wellen benutzt, wobei durch die Echo-Nachrichten sowohl die bei W1 lokal gemerkte Information als auch die bzgl. W2 relevante lokale Information an den Initiator übermittelt wird.

- Welle W1: "rot werden"; Welle W2: "grün werden"

- *Behauptung*: Das so realisierte Doppelzählverfahren ist korrekt.
- *Problem*: Formeller oder informaler Beweis lassen sich so nicht anwenden, da sich W1 und W2 *überlappen*!

Bemerkung:
Auf vorhandenen Spannbäumen kann man aber nicht einfach die vom Initiator ausgesendete Hinwelle und die reflektierte Rückwelle verwenden!



- "Trick": Es gibt keine nach W2 gesendete Nachricht, die vor W1 ankommt (grüne Knoten haben keine weißen Nachbarn!)
  - Wenn ein Knoten grün wird, sind seine Nachbarn bereits vorher rot geworden
  - --> Täuschung der Zähler durch Kompensation mittels Nachrichten "rückwärts" über 2 Wellen ist unmöglich!

- *Beweisskizze*:

1) Im roten Gebiet (d.h. zwischen W1 und W2) findet keine Aktivität statt, wenn das Terminierungskriterium S=E=S'=E' gilt:
Da W1 nur passive Prozesse "schneidet", müßte dazu eine vor W1 (im weißen Gebiet) ausgesendete Nachricht im roten Gebiet ankommen. Dann ist aber E' > E.

2) Es kann Nachrichten geben, die beide Wellen "vorwärts" überqueren (d.h. im weißen Gebiet gesendet werden und im grünen ankommen). Solche Nachrichten werden auf S (und S') als gesendet registriert, jedoch weder auf E noch auf E' als empfangen registriert. Da eine Kompensation der Zähler S, E mittels Nachrichten "rückwärts" über 2 Wellen unmöglich ist (wie im Gegenbsp. zum einfachen Zählen), ist dann S > E.