

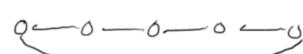
## Verbindungs topologien

### Prozessor - Prozessor

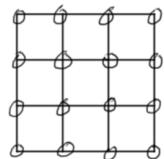
-) Lineares Feld



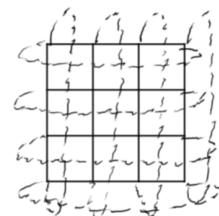
-) Ring



-) Gitter



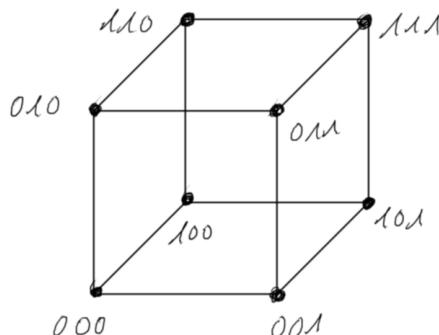
-) Torus



-) Hypercube: Sei  $d \in \mathbb{N}$  (Dimension). Dann Anzahl Knoten  $2^d$

$\Rightarrow$  Ein Knoten mit Binäradresse  $m_0, m_1, \dots, m_{d-1}$  ist mit einem anderen Knoten verbunden, wenn sich Binäradressen genau um einen Bit unterscheiden.

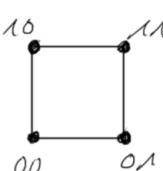
$d = 3:$



$d = 1:$

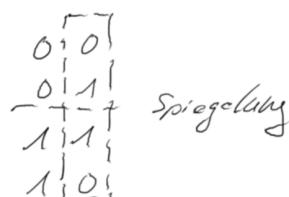


$d = 2:$



Einbettung Ring in Hypercube:

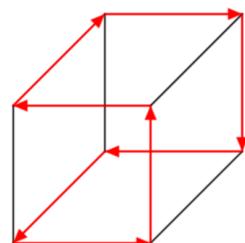
$d = 2:$



$\Rightarrow d = 3:$

0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0

binary reflected Gray code



Früher: „Store and Forward Routing“:  $P_0 \rightarrow P_1 \rightarrow P_2$

Realisation:  $P_0 \rightarrow P_1, P_1 \rightarrow P_2$

$\rightarrow$  Kosten basieren auf Anzahl Zwischenschritte

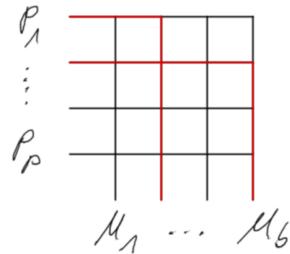
Heute: „wormhole Routing“:  $P_0 \rightarrow P_1 \rightarrow P_2$

Realisation: einmal Kanal freischalten und anschließend durchrouten

$\Rightarrow$  Kostenmodell:  $T(N) = a + \beta N$

Prozessor - Speicher:

•) Crossbar:



$\Rightarrow$  Verbindung von  $P_i - M_j$  gleichzeitig möglich mit  $P_k - M_l$ , wenn  $i \neq k$  und  $j \neq l$

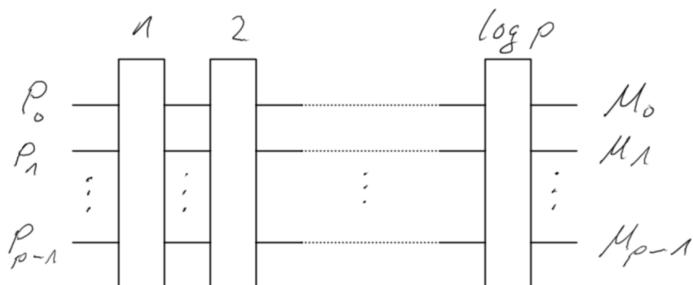
$\Rightarrow$  Kosten skaliieren mit  $O(p \cdot b)$

$\Rightarrow$  effizient, aber teuer für große Rechner  
(Bus: preisgünstig, ineffizient)

•) Mehrstufiges Netzwerk:

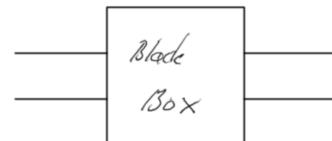
- $p$  Prozessoren sind mit  $b$  Bänken verbunden ( $p, b \in \mathbb{N}$ )  
(Bsp: S2 - Netzwerk)

$\Rightarrow$   $\log p$  identische Stufen, um  $p$  Prozessoren mit  $b$  Bänken zu verbinden

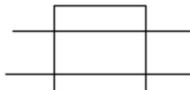


Schaltelemente:

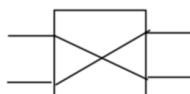
- ) Jede Stufe enthält  $\frac{p}{2}$  Schaltelemente:



Funktionen: a) „Pass Through“:



b) „Cross Over“:



Verknüpfung der Schaltelemente nach:

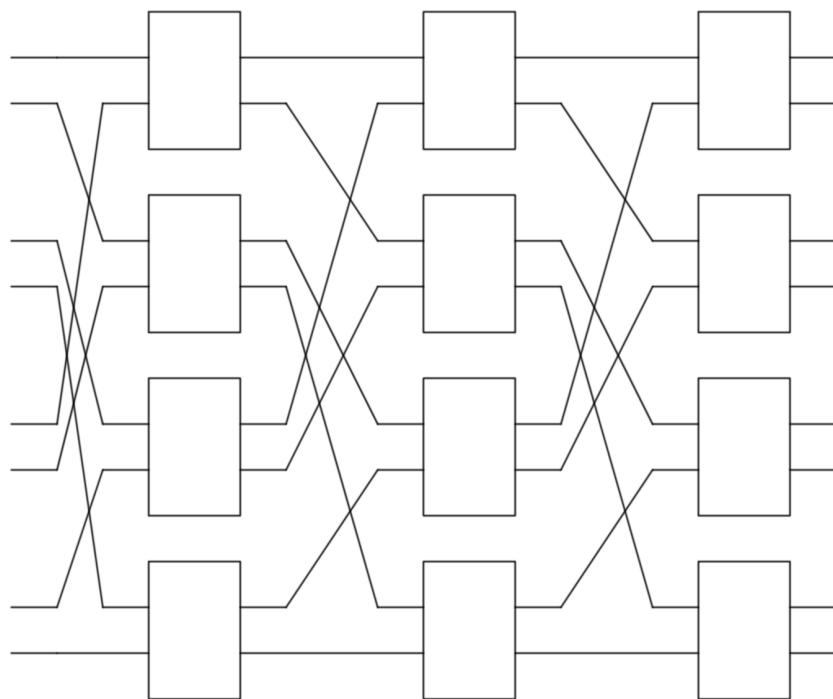
- ) Ausgang  $i$  von Stufe  $k$  wird mit Eingang  $j$  von Stufe  $k+1$  verknüpft:

$$j = \begin{cases} 2i & : \text{falls } 0 \leq i < \frac{p}{2} \\ 2i+1-p & : \text{sonst} \end{cases}$$

— dies wird auch „perfect shuffle“ genannt

Beispiel:  $p = 8$

0	000	000	0	$\Rightarrow$ zyklischer links - shift in Binärdarstellung
1	001	001	1	
2	010	010	2	
3	011	011	3	$\Rightarrow$ Anzahl Schalter: $\frac{p}{2} \log p$
4	100	100	4	
5	101	101	5	im $\Omega$ -Netzwerk
6	110	110	6	
7	111	111	7	



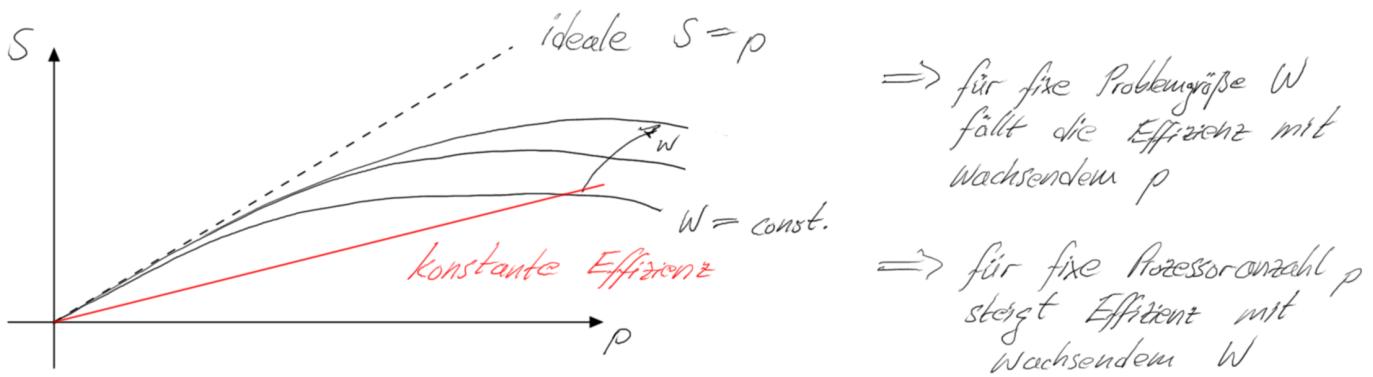
Methode für Test: „Destination tag routing“

### Skalierbarkeit von parallelen Systemen

Definition: Ein paralleles System ist ein Algorithmus und eine Architektur, auf der dieser Algorithmus ausgeführt wird.

$$\left. \begin{array}{l} \text{Speedup: } S(p) := \frac{T(1)}{T(p)} \\ \text{Effizienz: } E(p) := \frac{S(p)}{p} \end{array} \right\} \begin{array}{l} \text{messet Einfluss} \\ \text{von parallelen Re-} \\ \text{sourcen bei gleich-} \\ \text{bleibender Problemgröße} \end{array}$$

Bemerkung: Problemgröße  $W$  (von Work) ist hier die Anzahl der arithmetischen Operationen des seitlichen Algorithmus.  $W = T(1)$



- Parallelle Systeme werden eingesetzt, um große Probleme darstellen zu können, Zeit zu reduzieren.
- Aus Benutzersicht ist folgende Frage relevant:

Wenn ein Problem mit Größe  $W$  mit  $p$  Prozessoren in einer Zeit  $T$  gelöst werden kann, kann dann in gleicher Zeit ein Problem der Größe  $2W$  mit  $2p$  Prozessoren gelöst werden?

$$E(2W, 2p) = E(W, p)$$

- Skalierbarkeit misst die Fähigkeit eines parallelen Systems, den Speedup mit der Anzahl der Prozessoren zu erhöhen.

$$\Rightarrow \text{Overhead-Funktion: } T_0(W, p) = p \cdot T(p) - W$$

$$\Rightarrow E(W, p) = \frac{S(W, p)}{p} = \frac{T(W)}{p \cdot T(p)} = \frac{W}{p \cdot T(p)} = \frac{W}{T_0(W, p) + W}$$

$$\Rightarrow E(W, p) = \left(1 + \frac{T_0(W, p)}{W}\right)^{-1}$$

$\Rightarrow$  um Effizienz auf  $E_0$  konstant zu halten, muss gelten:

$$1 + \frac{T_0}{W} = \frac{1}{E_0} \Rightarrow W = \frac{E_0}{1-E_0} T_0(W, p) \quad (*)$$

Die Isoeffizienz-Funktion gibt Rote an, mit der  $W$  wachsen muss, um Effizienz, bei steigender Prozessoranzahl konstant zu halten.  
etwa:

a)  $W = \Theta(e^p)$  (sehr schlecht skalierbar)

b)  $W = \Theta(p)$  (sehr gut skalierbar)

Definition: Falls eine Lösung von (\*), also  $W$  als Funktion von  $p$  gibt, heißt die Lösung Isoeffizient-Funktion. Das parallele System heißt skalierbar.

Beispiel: (Iscoffizienzanalyse)

-) Addition von  $n$  Zahlen auf Hypercube mit  $p$  Prozessoren:

serielle Laufzeit:  $T(1) = W = (n-1) \cdot T_A \approx n \cdot T_A$

mit  $T_A$  als Laufzeit für arithmetische Operationen

$$\text{parallele Laufzeit: } T(p) = \underbrace{\frac{n}{p} T_A}_{\text{lokaler Anteil}} + \log p (\beta + 1 \cdot \gamma + T_A)$$

↗      ↗      ↗  
 Latenz Daten skalares „+“

$$\text{Overhead-Funktion: } T_0(W_{IP}) = pT(p) - w = n \cdot T_A + p \log p (\beta + \alpha + T_A) - n \cdot T_A$$

$$\Rightarrow T_0(\omega, p) = p \cdot \log p (\beta + \alpha + T_A)$$

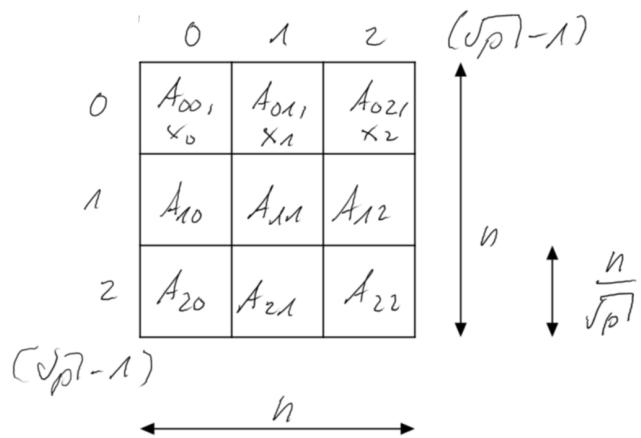
$$\Rightarrow W = \frac{E}{1-E} T_0 = \frac{E}{1-E} \rho \cdot \log \rho (\beta + \tau + T_A)$$

$\Rightarrow$  Isoeffizienz-Funktion:  $\Theta(p \log p)$   $\Rightarrow$  paralleles System ist skalierbar

Beispiel: (Matrix- Vektor- Produkt auf  $\sqrt{p} \times \sqrt{p}$ -Gitter)

Sei  $A$  eine  $n \times n$ -Matrix, die blockweise auf ein Prozessorenrgitter verteilt sei. Ein Vektor  $x \in \mathbb{R}^n$  sei „konform“ zur  $A$  partitioniert und liegt in Gitterzeile 0. Ziel:  $y = Ax$

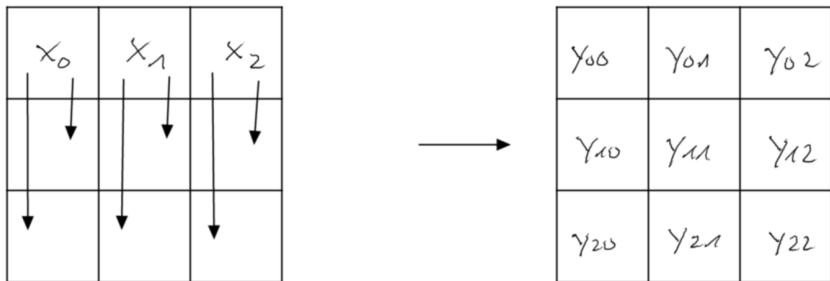
$$\Leftrightarrow x = \begin{bmatrix} x_0 \\ \vdots \\ x_{\sqrt{p}l-1} \end{bmatrix} \xrightarrow{\frac{1}{\sqrt{p}l}}$$



```

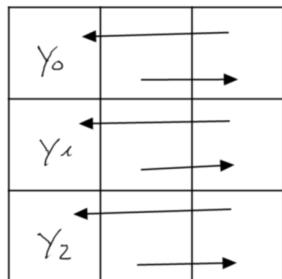
(my-row, my-col) = my-id()
column-context = {Pij | j = my-col}
row-context = {Pij | i = my-row}
broadcast (xloc, source = P0, my-col, context =
            column-context)
y-loc = MatrixVectorProd (Aloc, xloc)
reduce (in = y-loc, result = y, op = t, context =
            row-context, dest = (my-row, 0))

```



mit  $y_{ij} = A_{ij} x_j$   
 ( $\rho$  Matrix-Vektor-Produkte  
 mit  $\frac{n}{\sqrt{\rho}}$   $\times \frac{n}{\sqrt{\rho}}$  Matrizen)

$\sqrt{\rho}$ -Broadcasts von Vektoren der Länge  $\frac{n}{\sqrt{\rho}}$  entlang der Spalten mit  $\sqrt{\rho}$  Prozessoren simultan



$\sqrt{\rho}$ -Reduktionen von Vektoren der Länge  $n/\sqrt{\rho}$  entlang Zeilen mit  $\sqrt{\rho}$  Prozessoren

$$y_i = \sum_{j=0}^{\sqrt{\rho}-1} A_{ij} x_j$$

$$\text{es ist dann wieder: } y = [y_0, \dots, y_{\sqrt{\rho}-1}]$$

Koeffizienzanalyse:

$$\text{serielle Laufzeit: } W \approx 2n^2 T_A$$

$$\begin{aligned} \text{parallele Laufzeit: } T(\rho) &\approx \log \sqrt{\rho} \cdot (\beta + \frac{n}{\sqrt{\rho}} \tau) \text{ "Broadcast x"} \\ &+ 2 \left( \frac{n}{\sqrt{\rho}} \right)^2 T_A \text{ "Matrix-Vect.-P."} \\ &+ \log \sqrt{\rho} \cdot \left( \beta + \frac{n}{\sqrt{\rho}} (\tau + T_A) \right) \text{ "Reduktion"} \end{aligned}$$

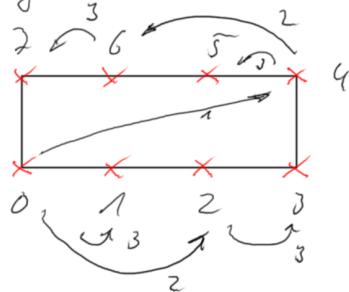
$$\begin{aligned} \text{Overhead-Funktion: } T_0 &= \rho \cdot T(\rho) - W = \rho \log \sqrt{\rho} \cdot 2\beta \\ &+ \rho \log \sqrt{\rho} \frac{n}{\sqrt{\rho}} (2\tau + T_A) \\ &+ \underbrace{\rho \cdot \frac{2n^2}{\rho} T_A - 2n^2 T_A}_{=0} \\ &= \underbrace{\beta \cdot \rho \log \rho}_{\text{nicht abhängig von } W} + \underbrace{\sqrt{\rho} \log \sqrt{\rho} n (2\tau + T_A)}_{\text{abhängig von } W} \end{aligned}$$

$$\Rightarrow \text{Lösen der Gleichung: } \underline{\underline{W = \Theta(\rho \cdot \log \rho)}}$$

- getroffene Annahmen:
- ) Zeit zum Datentransfer ist unabhängig von relativer Entfernung der Knoten
  - $\Rightarrow \tau = \beta + m \tau$  gilt für benachbige Knoten
  - ) cut through routing
  - ) Verbindungen sind bidirektional
  - ) single-port-communication: (immer nur eine Nachricht zu einem bestimmten Zeitpunkt)
  - ) Senden und Empfangen funktioniert über gleichzeitig

one-to-all broadcast:

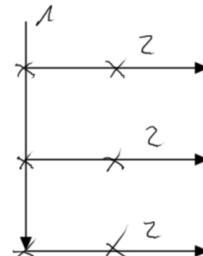
$$\text{a) Ring: } \tau = (\beta + m \tau) \log p$$



b)  $\sqrt{p} \times \sqrt{p}$  - Gitter:

$$\tau = (\beta + m \tau) \log \sqrt{p} + (\beta + m \tau) \log \sqrt{p}$$

$$\Rightarrow \boxed{\tau = (\beta + m \tau) \log p}$$



all-to-all broadcast:

Bsp: Ring:

```

me = my_id()
left = (me - 1) mod p
right = (me + 1) mod p
result = data
msg = data
  
```

```

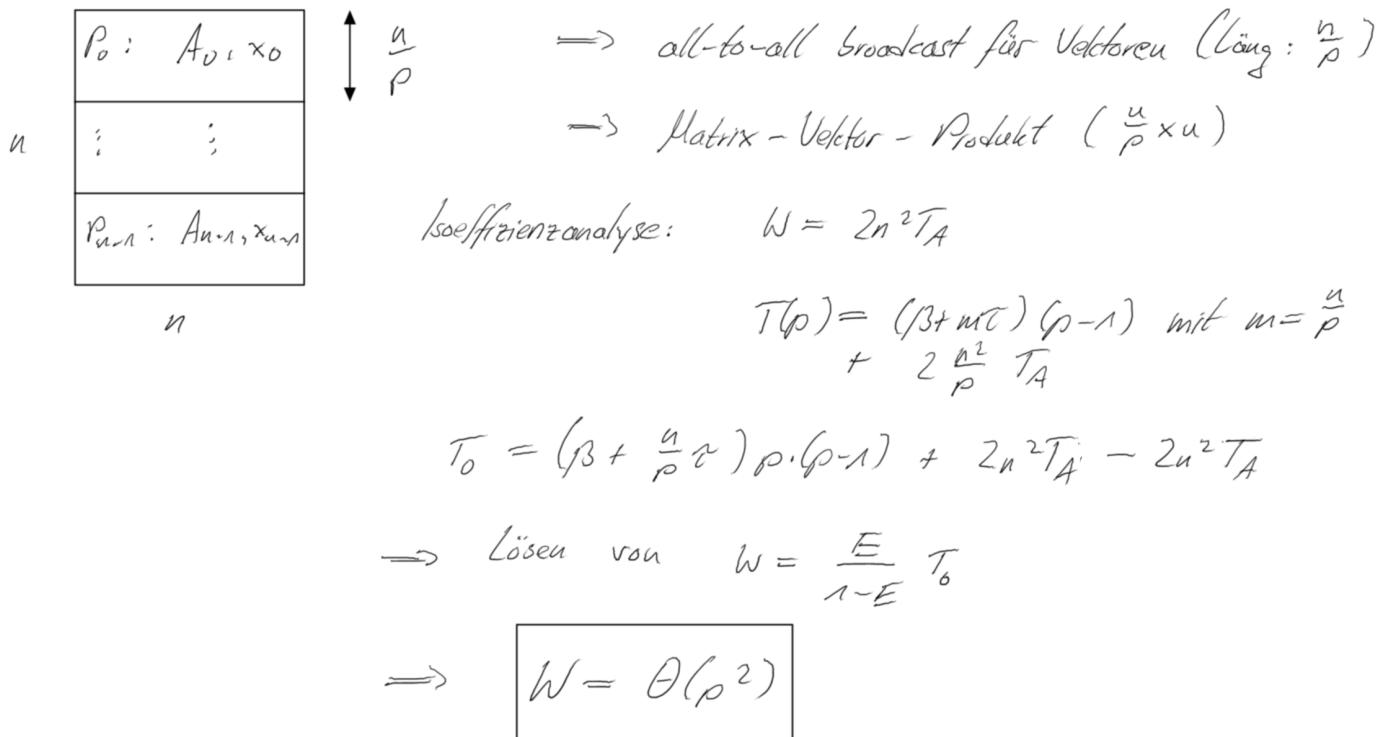
for i = 1 to p-1 do
  send msg, rechts
  get msg, links
  result += msg
  
```

$$\Rightarrow \boxed{\tau = (\beta + m \tau) \cdot (p-1)}$$

Matrix - Vektor - Produkt auf Ring:

$p$  Prozessoren,  $A \in M(n, n)$ ,  $n \in N$ ,  $x \in R^n$

⇒ blockzeilenweise auf Prozessoren aufteilen



## Matrix - Matrix - Multiplikation

Seien  $A, B \in \mathbb{R}^{u \times n}$  zwei  $u \times n$ -Matrizen. Berechne  $C := A \cdot B \in \mathbb{R}^{u \times n}$

$$\Rightarrow T_{\text{serial}} = W \approx 2n^3 T_A$$

## Blockorientierter Algorithmus:

Sei  $n \bmod \sqrt{p} = 0 \Rightarrow$  Aufteilung der Matrizen in  $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$  Blöcke.

for  $(i=0)$  to  $(\sqrt{p7}-1)$  do  
 for  $(j=0)$  to  $(\sqrt{p7}-1)$  do  
 $c_{ij} = 0$   
 for  $(k=0)$  to  $(\sqrt{p7}-1)$  do  
 $c_{ij} = c_{ij} + A_{ik} \cdot B_{kj}$

$(*) [my\_row, my\_col] = my\_id()$   
 column-context =  $\{ p_{ij} \mid j = my\_col \}$   
 row-context =  $\{ p_{ij} \mid i = my\_row \}$   
 all-to-all-broadcast ( $A_{loc}$ , source =  $p_{my\_row, my\_col}$ , context = row-context)  
 all-to-all-broadcast ( $B_{loc}$ , source =  $p_{my\_row, my\_col}$ , context = column-context)

Analyse: (1) all-to-all-broadcast:

$$\Rightarrow \text{innerhalb einer Gitterzeile mit } \sqrt{p} \text{-Prozessoren}$$

$$\Rightarrow \text{Datengröße } n^2/p \text{ (simultan für } \sqrt{p} \text{-Gitterzeilen)}$$

$$\Rightarrow (\alpha + \frac{n^2}{p} \tau) (\sqrt{p}-1) \quad (\text{analog für Gitterspalten})$$

(2)  $\sqrt{p}$ - serielle Matrix-Matrix-Multiplikation der Größe  $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$

$$\Rightarrow \sqrt{p} \cdot 2 \cdot \left(\frac{n}{\sqrt{p}}\right)^3 \cdot T_A$$

$$\Rightarrow T(p) = 2 \frac{n^3}{p} T_A + 2 \left( \alpha + \frac{n^2}{p} \tau \right) (\sqrt{p}-1)$$

$$\approx 2 \frac{n^3}{p} T_A + 2\sqrt{p} \left( \alpha + \frac{n^2}{p} \tau \right)$$

$$\Rightarrow T_0 = p T_p - w = 2 \alpha p^{\frac{3}{2}} + 2n^2 \tau \sqrt{p}$$

$$\Rightarrow W = \Theta(p^{\frac{3}{2}})$$

Problem: Algorithmus benötigt sehr viel Speicher. Für jeden Prozess  $2n \cdot \frac{n}{\sqrt{p}}$  Skalarwerten

$$\Rightarrow \text{für alle Prozesse: } \sqrt{p} \cdot 2n^2$$

$\Rightarrow$  Das ist das  $\sqrt{p}$ -fache des seriellen Algorithmus!

Algorithmus von Cannon:

-) Datenverteilung wie im Blockalgorithmus

) Idee: Reduziere Speicherbedarf vom Blockalgorithmus durch systematische Shifts von  $n/\sqrt{p} \times n/\sqrt{p}$  Blöcke der Matrizen A und B beim Überschreiben dieser Daten

$\Rightarrow$  Umnummerierung der innersten Schleife:

$$C_{ij} = C_{ij} + \sum_{k=0}^{\sqrt{p}-1} A_{ik} B_{kj} = C_{ij} + \sum_{k=0}^{\sqrt{p}-1} A_{i, (i+j+k) \bmod \sqrt{p}} B_{(i+j+k) \bmod \sqrt{p}, j}$$

for all  $(i=0)$  to  $(\sqrt{p}-1)$  do

zirkulärer Links-Shift von Zeile  $i$  von  $A$ , sodass  $A_{ij}$  durch  $A_{i,(i+1) \text{ mod } \sqrt{p}}$  überschrieben wird

for all  $(j=0)$  to  $(\sqrt{p}-1)$  do

zirkulärer Hoch-Shift von Spalte  $j$  von  $B$ , sodass  $B_{ij}$  durch  $B_{(i+1) \text{ mod } \sqrt{p}, j}$  überschrieben wird.

$C = 0$

for  $(k=0)$  to  $(\sqrt{p}-1)$  do

forall  $(i=0)$  to  $(\sqrt{p}-1)$  do

forall  $(j=0)$  to  $(\sqrt{p}-1)$  do

$$C_{ij} = C_{ij} + A_{ij}B_{ij}$$

zirkulärer Links-Shift von Zeile  $i$  von  $A$  ( $A_{ij} = A_{i,(j+1) \text{ mod } \sqrt{p}}$ )

zirkulärer Hoch-Shift von Zeile  $j$  von  $B$  ( $B_{ij} = B_{(i+1) \text{ mod } \sqrt{p}, j}$ )

Analyse: (1) Links-Shift (analog Hoch-Shift):  $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} = \frac{n^2}{p}$  Elemente mit  $\sqrt{p}$  Prozessoren

$$\Rightarrow \alpha + \frac{n^2}{p} \tau$$

(2)  $\sqrt{p}$ - serielle - Matrix - Matrix - Multiplikation der Größe  $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$

$$\Rightarrow \sqrt{p} \cdot 2 \cdot \left(\frac{n}{\sqrt{p}}\right)^3 \cdot T_A = \frac{2n^3}{p} T_A$$

$$\Rightarrow W = \Theta(p^{\frac{3}{2}}) \Rightarrow \text{genau wie oben + bessere Speicherverwaltung}$$

Bemerkung: \*) es können maximal  $p = n^2$  Prozessoren eingesetzt werden  
 $\Rightarrow \Theta(n)$

### Algorithmus von Dekel, Nassimi, Sarni

\*) Ziel: Verwendung von mehr als  $n^2$  Prozessoren

Idee:  $n^3$  Prozesse in Form eines  $n \times n \times n$ -Gitters: Es müssen rund  $n^3$  Multiplikationen ausgeführt werden. Jeder Prozess soll nun eine Multiplikation übernehmen.

- $\Rightarrow$  Sei  $P_{ijk}$  der Prozess, der  $a_{ik} b_{kj}$  berechnet.  
 $\Rightarrow$  Nach Multiplikation müssen Produkte noch über  $k$  summiert werden  
 (Reduktion)

Durchführung:

(1) Schicke  $a_{ij}$  von  $P_{ijo}$  nach  $P_{ijj}$  simultan für  $n^2$  Prozesse

$$\Rightarrow \alpha + 1 \cdot \tilde{\tau}$$

(2) Schicke  $a_{ij}$  von  $P_{ijo}$  nach  $P_{i*\bar{j}}$  ( $*$  steht für alle möglichen Werte)  
 one-to-all broadcast entlang jedes linearen Arrays mit  $n$  Prozessen

$$\Rightarrow (\alpha + 1 \cdot \tilde{\tau}) \cdot \log n$$

(3) Schicke  $b_{ij}$  von  $P_{ijo}$  nach  $P_{ijj}$  (analog zu 1)

(4) Schicke  $b_{ij}$  von  $P_{ijj}$  nach  $P_{jji}$  (analog zu 2)

(5) Multiplikation auf  $P_{ijk}$ :  $a_{ik} b_{kj}$  (simultan für  $n^3$  Prozesse)

(6)  $P_{ijo}$  akkumuliert Reduktion entlang  $P_{ijj*}$  mit  $n$  Prozessen

$$\Rightarrow (\alpha + \tau + \tau_A) \log n$$

$$\begin{aligned}
 \Rightarrow \text{Laufzeit: } T(p) &= 2(\alpha + \tilde{\tau})(1 + \log n) + \tau_A + (\alpha + \tilde{\tau} + \tau_A) \log n \\
 &\Rightarrow T \in \Theta(\log n) \quad (\text{wenn } p = n^3)
 \end{aligned}$$

$$\begin{aligned}
 \text{Overhead-Funktion: } T_0 &= p T(p) - N = 3(\alpha + \tilde{\tau})n^3 \log n \\
 &\quad + \tau_A n^3 \log n + [2(\alpha + \tilde{\tau}) + \tau_A] n^3 \\
 &\quad - 2n^3 \frac{\tau_A}{\tau_A}
 \end{aligned}$$

$$\text{für } p = n^3 \text{ gilt: } T_0 = (\alpha + \tau)p \log p + \frac{1}{3}\tau_A p \log p + [2(\alpha + \tau) - \tau_A] n^3$$

$$\Rightarrow N \in \Theta(p \log p)$$

## Algorithmus von Dekhtsi, Nassimi, Sahni blockorientiert

⇒ Modifikation auf Blöcke (Erhöhe „granularität“)

Sei  $p = q^3$  für ein  $q < n$ .

⇒ In Ebene  $k=0$  sind zu Beginn auf allen  $P_{ij0}$  mit  $i, j = 1, \dots, q$  zwei Blöcke der Größe  $\frac{n}{q} \times \frac{n}{q}$

Dann ist dieser Algorithmus analog zu vorher, nur das die skalare Multiplikation durch Multiplikation der  $\frac{n}{q} \times \frac{n}{q}$ -Blöcke ersetzt wird mit  $q^3$  Prozessen.

⇒ Analyse: (1) Sende Blöcke zu anderem Prozess für  $q^2$  Prozesse

$$\Rightarrow 2(\alpha + \frac{n^2}{q^2} \tau)$$

(2) one-to-all-broadcast von  $\frac{n}{q} \times \frac{n}{q}$ -Blöcke entlang des linearen Arrays mit  $q$  Prozessen

$$\Rightarrow 2(\alpha + \frac{n^2}{q^2} \tau) \log q$$

(3) Matrix-Matrix-Multiplikation von  $\frac{n}{q} \times \frac{n}{q}$ -Matrizen

$$\Rightarrow 2(\frac{n^3}{q}) T_A$$

(4) Reduktion von  $\frac{n}{q} \times \frac{n}{q}$ -Matrizen entlang des linearen Feldes mit  $q$  Prozessen

$$\Rightarrow (\alpha + \frac{n^2}{q^2} \tau + \frac{n^2}{q^2} T_A) \log q$$

$$\Rightarrow \text{Laufzeit: } T(p) = 2 \frac{n^3}{q^3} T_A + \left[ 3 \left[ \alpha + \frac{n^2}{q^2} \tau \right] + T_A \frac{n^2}{q^2} \right] \log q + 2 \left( \alpha + \frac{n^2}{q} \tau \right)$$

$$\Rightarrow T \in O(\log^3 n), \text{ wenn } p = \frac{n^3}{\log^3 n}$$

$$\rightarrow \omega \in \Theta(p \cdot \log^3 p)$$

(\*) damit Algorithmus kosten optimal ist

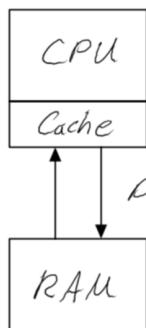
## Untere Schranken für Datenbewegungen

Zeit zur Ausführung eines Algorithmus mit  $F$  Rechenoperationen und  $S$  Datenbewegungen, welche insgesamt  $W$  Wörter bewegen:

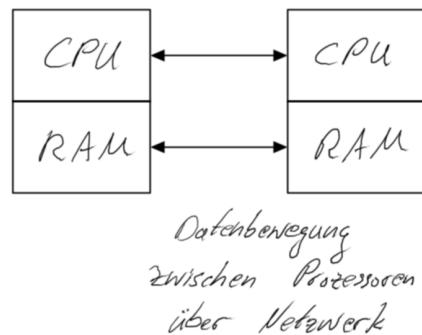
$$T = F \cdot T_A + S \cdot \alpha + W \cdot \tau$$

Technologietrend:  $\alpha \gg \tau \gg T_A$

seriell:



parallel:



Datenbewegung zwischen unterschiedlichen Ebenen einer Speicherhierarchie

Datenbewegung zwischen Prozessoren über Netzwerk

$\Rightarrow$  Ziel: Entwickle Algorithmen so, dass auf allen Ebenen

$$L_1 \leftrightarrow L_2 \leftrightarrow \dots \leftrightarrow L_k \leftrightarrow \text{RAM} \leftrightarrow \text{Netz}$$

die Datenbewegungen minimiert werden (Nicht nur Überlappungen von Datenbewegungen und Rechenoperationen)

Vorgehensweise:

- ) leite (möglichst große) untere Schranke für Datenbewegung her
- ) finde Algorithmus, der möglichst nah an dieser Schranke ist

## Dreifach-geschachtelte-Schleifen - Algorithmen

Beispiel: (Matrix-Matrix-Produkt)

```

for (i=0) to (n-1)
    for (j=0) to (n-1)
        for (k=0) to (n-1)
            Cij = Cij + aik bkj
    
```

Annahme: jede Traversierung liefert korrektes Ergebnis (numerische Stabilität)

Bemerkung: Hier ist eine Verallgemeinerung möglich.

Sieben  $n \in \mathbb{N}$  und  $S_A, S_B, S_C \subset \{1, \dots, n\} \times \{1, \dots, n\}$  die Mengen der Indizes auf die in  $A$  bzw.  $B, C$  zugegriffen wird.  
 Sei weiterhin  $a: S_A \rightarrow M$ ,  $b: S_B \rightarrow M$ ,  $c: S_C \rightarrow M$ , wobei  $M$  die Menge aller Positionen im „langsamsten“ Spalten beschreibt. (Bsp: Column-major-Format)

$$\begin{aligned} \text{Bsp: } c(i,j) &= \text{Offset}_c + i-1 + (j-1) \cdot n \\ a(i,h) &= \text{Offset}_a + i-1 + (h-1) \cdot n \\ b(k,j) &= \text{Offset}_b + k-1 + (j-1) \cdot n \end{aligned}$$

Die Funktionen  $a, b, c$  seien so gewählt, dass  $\text{Bild } a$ ,  $\text{Bild } b$ ,  $\text{Bild } c$  paarweise disjunkt sind und alle Funktionen injektiv sind.

Den Wert an einer Speicherposition  $i \in M$  bezeichnen wir mit  $\text{mem}(i)$ .  
 (d.h.  $\text{mem}: M \rightarrow X$ , wobei  $X$  die Menge aller möglichen Werte darstellt, welche gespeichert werden können)

Definition: (Dreifach-geschachtelte-Schleife)

Eine Berechnung heißt dreifach-geschachtelte-Schleife, falls sie

$$\text{mem}(c(i,j)) = f_{ij}\left(\left\{g_{jk}(\text{mem}(a(i,h)), \text{mem}(b(k,j)))\right\}_{k \in S_j}\right)$$

für alle  $(i,j) \in S_C$  berechnet, wobei  $S_{ij} \subset \{1, \dots, n\}$  die Menge aller Indizes ist, über die  $g_{jk}$  läuft (für  $k$ ).

Eine Operation ist dann die Auswertung einer Funktion  $g_{jk}$ .

Bemerkung:  $\cdot$ )  $S_C, f_{ij}, S_{ij}$  oder  $g_{jk}$  können auch erst zur Laufzeit bekannt werden z.B. bei Pivotierung von Matrizen

Nach Loomis und Whitney (1949) gilt:

Sei  $V$  eine Menge von Gitterpunkten in  $\mathbb{R}^3$  (d.h.  $(i,j,k) \in \mathbb{Z}^3$ ). Sei  $V_i$  die Projektion von  $V$  in Dimension  $i$ . Seien  $V_j, V_k$  analog. Dann gilt:

$$\#V \leq (\#V_i \cdot \#V_j \cdot \#V_k)^{\frac{1}{2}}$$

Definition: Bewegung von Daten der Größe  $W$  zwischen langsamem und schnellem Speicher (seriell) bzw. zwischen zwei Prozessoren (parallel) benötigt Zeit  $\alpha + W\tau$  für bestimmte  $\alpha, \tau > 0$ .  
 Dann sind die Bandbreitenkosten  $W$ .

Lemma: (untere Schranke Bandbreitenkosten)

Eine untere Schranke für die Bandbreitenkosten einer Dreifach-Schleifen-Berechnung mit 6 Operationen und Größe des schnellen Speichers  $M$  ist:

$$W \geq \frac{6}{8\sqrt{M}} - \mu \quad \text{serieller Fall}$$

$$W \geq \frac{6}{8P\sqrt{M}} - \mu \quad \text{im parallelen Fall}$$

Spezialfall  $G \in \Theta(n^3)$ : seriell:  $W = \mathcal{O}\left(\left(\frac{n}{\sqrt{M}}\right)^3 M\right)$

parallel:  $W = \mathcal{O}\left(\left(\frac{n}{\sqrt{M}}\right)^3 \frac{M}{P}\right)$

## Shared - Memory - Programming

### pthreads

Thread: lightweight-process

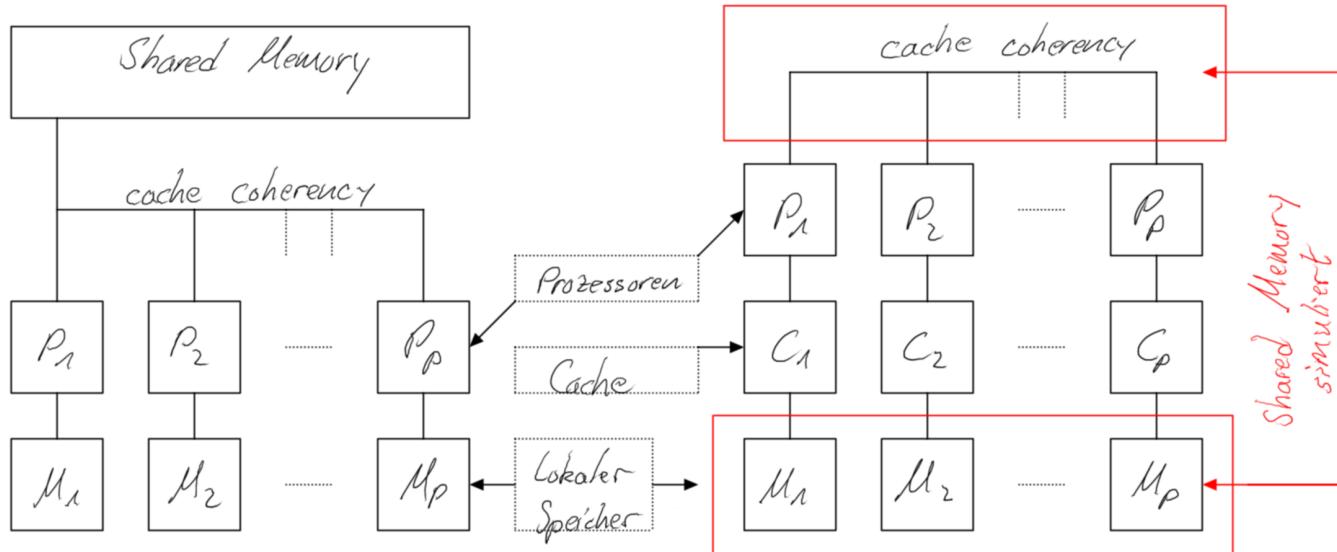
- unabhängigen Kontrollfluss innerhalb eines Prozesses
- dynamisch erzeugbar und terminierbar
- alle Threads innerhalb eines Prozesses teilen globale Variablen bzw. Daten
- besitzt eigenen Stack, lokale Variablen und Programmzähler
- Kontextwechsel zwischen Threads schneller als bei Prozessen

Befehle zum: •) Erzeugen, Zusammenführen und Beenden von Threads

### shared - memory mit OpenMP

Idee / Modell: ( $p \in \mathbb{N}$ )

Realität:



- Speicherzugriffzeit für alle Prozessoren auf jede Zelle gleich  
→ in Realität nicht möglich ⇒ Zugriff auf Speicher in der „Nähe“ schneller
- cache coherency: Abruf von Variablen wird immer für mehrere Prozessoren ausgeführt ⇒ kürzere Zugriffszeiten im koherrenten Fall, aber schlechterer im inkoharenten

•) OpenMP: Parallelisierung serialen Quelltextes durch einfache Präprozessor-Direktiven

## Partitionsierte Systeme

Sei  $A \in \mathbb{R}^{n \times n}$  eine symmetrische, positiv definite Matrix (d.h.  $A = A^T$ ,  $x^T A x > 0$  für alle  $x \in \mathbb{R}^n \setminus \{0\}$ ) (auch spd)

Sei  $b \in \mathbb{R}^n$

Aufgabe: Löse  $Ax = b$

1. Cholesky-Zerlegung: Es existiert eine eindeutige Faktorisierung der Form

$$A = L \cdot L^T \text{ mit } L \text{ als untere Dreiecksmatrix und } l_{ii} > 0$$

$$Ax = b \Leftrightarrow L(L^T x) = b \Leftrightarrow Ly = b, L^T x = y$$

Eine Umsortierung der Zeilen/Spalten ist möglich, da die symmetrische Permutation von  $A$  definiert durch  $\tilde{A} = PAP^T$  mit  $P$  als Permutationsmatrix ebenfalls spd ist.

$$\Rightarrow PAx = PAP^T P x = Pb \Rightarrow \tilde{A}\tilde{x} = \tilde{b} \text{ mit } \tilde{x} = Px, \tilde{b} = Pb$$

Falls  $A$  dünn besetzt ist, kann es Fill-In geben, d.h. Positionen  $(i,j)$  bei denen  $a_{ij} = 0$  ist, aber  $l_{ij} \neq 0$  ist.

$\Rightarrow$  mehr Speicheranwendung für  $L \Rightarrow$  unter Umständen längere Berechnungszeit

Dieses Problem lässt sich durch Ummumerierung beheben.

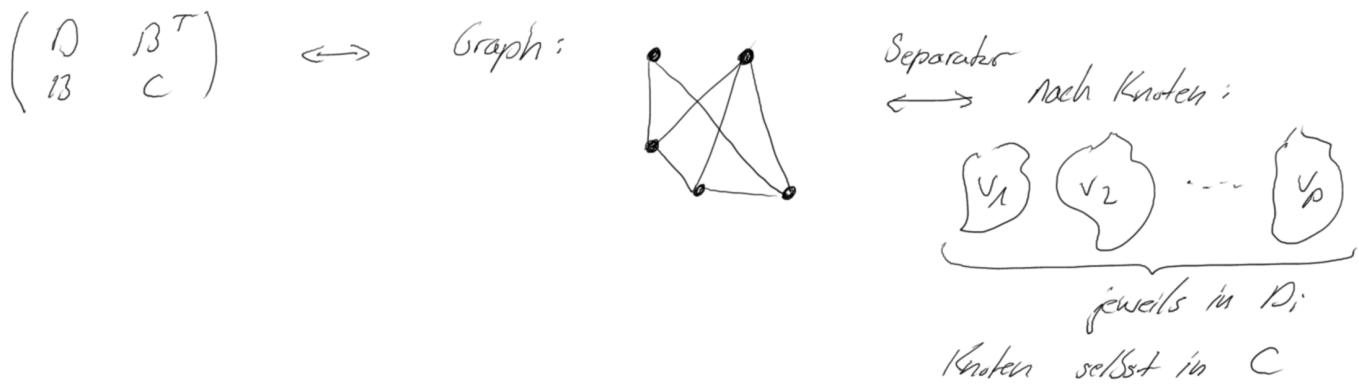
Anzahl Fill-In's hängt aber stark vom Muster der Nicht-Null-Einträge ab.  
Berechnung einer "guten" Umsortierung ist aber zeitaufwändig.

2. Block-Cholesky-Zerlegung

$$\text{Sei } A = \begin{pmatrix} D & B^T \\ B & C \end{pmatrix} \text{ mit } D, B, C \text{ Matrizen (nicht unbedingt quadratisch)}$$

eine Block-Matrix. Dann gibt es eine Block-Cholesky-Zerlegung  $A = L \cdot L^T$  mit

$$L = \begin{pmatrix} E & 0 \\ F & G \end{pmatrix} \text{ wobei } E, G \text{ untere Dreiecksmatrizen sind.}$$



Oft muss man auch Kanten statt Knoten aus „Berechnungsgraph“ herausnehmen.  
Knoten modellieren Rechenlast einer Teilaufgabe.  
Kanten modellieren Austausch von Daten (Kommunikation) zwischen Teilaufgaben.

**Definition:**  $G = (V, E, s, \omega)$  sei ungerichteter Graph mit Knotenmenge  $V$ ,  
Kantemenge  $E$ ,  $s$  Knotengewichte ( $s: V \rightarrow \mathbb{N}$ ),  $\omega: E \rightarrow \mathbb{N}$  Kantengewichte. Eine Abbildung  $\rho: V \rightarrow \{1, \dots, p\}$ , die die Knoten  $G$  in disjunkte  $V_i$  (zusammenhangskompl.) mit  $V = \bigcup_{i=1}^p V_i$  aufteilt, heißt  $p$ -Partition.  
Eine 2-Partition heißt Bisektion.

Sei für  $V_i \subset V$ :

$$s(V_i) = \sum_{v \in V_i} s(v)$$

Eine  $p$ -Partition heißt  $\epsilon$ -balanciert, falls

$$s(V_i) = \frac{1+\epsilon}{p} \sum_{j=1}^p s(v_j) \quad \text{mit } \epsilon > 0$$

Außerdem ist der Schnitt von  $\rho$

$$\text{cut } \rho := \sum_{\substack{(u,v) \in E \\ \rho(u) \neq \rho(v)}} \omega((u,v)) \quad (\text{"Summe der Gewichte der Schnittkante"})$$

„Bestimmen eine  $\epsilon$ -balancierte  $p$ -Partition mit minimalem Schnitt“  
heißt  $p$ -weg Partitionierungsproblem.

Verteile Teilaufgaben so auf  $p$  Prozesse, dass Rechenlast gleichmäßig verteilt ist und Kommunikation zwischen Prozessen minimiert wird.

Problem schon für  $s = \omega = 1$ ,  $\epsilon = 0$ ,  $p = 2$  (Bisektionsproblem)  
ist NP-vollständig.

Verwandtes Problem: Partitionsverfeinerung

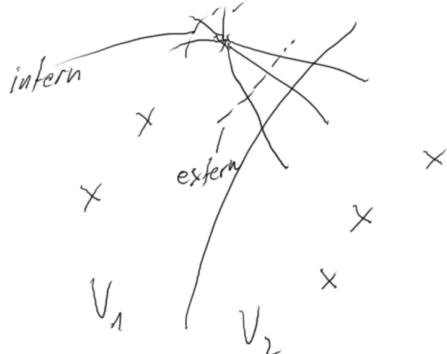
Gegeben ein Graph mit nicht minimaler Partierung (balanciert)

Finde neue Partition (balanciert) mit kleinerem Schnitt.

Kernighan-Lin - Verfeinerung:

Idee: Ausgehend von einer Anfangspartitionierung in zwei Komponenten  $V_1$  und  $V_2$  finde zwei gleich große Teilmengen  $U_1 \subset V_1$  und  $U_2 \subset V_2$ , sodass Austauschen von  $U_1$  nach  $V_2$  und  $U_2$  nach  $V_1$  maximale Reduzierung des Schnitts bewirkt.

Bilde Teilmengen  $U_1$  und  $U_2$  durch iterativen Prozess über Knotenpaare. Betrachte  $u \in V_1, v \in V_2$



$$\text{Definition: } \text{ext}(u) := \sum_{(u,v) \in E} \omega((u,v)) \\ P(u) \neq P(v)$$

$$\text{int}(u) := \sum_{(u,v) \in E} \omega((u,v)) \\ P(u) = P(v)$$

$$\text{diff}(u) := \text{ext}(u) - \text{int}(u)$$

$\text{diff}(u)$  ist ungefährs Maß für die Veränderung des Schnitts, wenn man  $u$  von  $V_1$  nach  $V_2$  verschieben würde. Knoten  $u$  mit großem  $\text{diff}(u)$  sind Kandidaten für Vertauschungsmaengen  $U_1$  und  $U_2$ .

Definition: Für ein  $(u,v) \in E$  mit  $P_u \neq P_v$  heißt

$$\text{gain}(u,v) := \text{diff}(u) + \text{diff}(v) - 2\omega(u,v)$$

der Gewinn. (mit  $\omega(u,v) = 0$  für  $(u,v) \notin E$ )

Algorithmus: Sei  $P$  initiale Partitionierung.

Berechne für alle  $u \in V$   $\text{diff}(u)$ .

Setze alle Knoten auf nicht markiert

Berechne  $c_0 := \text{cut } P$

für  $i = 1, i \leq \min(|V_1|, |V_2|)$

Bestimme ein Paar von noch nicht markierten Knoten  $u_i \in V_1, v_i \in V_2$  mit maximalem  $\text{gain}(u_i, v_i)$   
Markiere  $u_i, v_i$

für  $x \in \text{Adj}(u_i) \cup \text{Adj}(v_i)$  // Nachbarknoten

$$\text{diff}(x) = \text{diff}(x) + \begin{cases} 2\omega(x_i, u_i) - 2\omega(x_i, v_i); & x \in V_{P(u_i)} \\ 2\omega(x_i, v_i) - 2\omega(x_i, u_i); & \text{sowohl} \end{cases}$$

$$c_i := c_{i-1} - \text{gain}(u_i, v_i)$$

Bestimme kleinstes  $j$  mit  $c_j = \min_i c_i$

Vertausche  $\{u_1, \dots, u_f\}$ ,  $\{v_1, \dots, v_f\}$