Algorithmen für vollbesetzte Matrizen I

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 12/13

Themen

Algorithmen für vollbesetzte Matrizen als datenparallele Algorithmen

- Datenaufteilungen von Vektoren und Matrizen
- Matrixtransposition

Aufteilung von Vektoren I

- Vektor $x \in \mathcal{R}^N$ entspricht einer geordneten Liste von Zahlen.
- Jedem Index i aus der Indexmenge $I = \{0, \ldots, N-1\}$ wird eine reelle Zahl x_i zugeordnet.
- Anstelle von $\mathcal{R}^N$ schreiben wir $\mathcal{R}(I)$ schreiben, um die Abhängigkeit von der Indexmenge deutlich zu machen.
- Die natürliche (und effizienteste) Datenstruktur für einen Vektor ist das Feld.
- Da Felder in vielen Programmiersprachen mit dem Index 0 beginnen, tut dies auch unsere Indexmenge I.

Aufteilung von Vektoren II

- Eine Datenaufteilung entspricht nun einer Zerlegung der Indexmenge I in

$$I = \bigcup_{p \in P} I_p, \text{ mit } p \neq q \Rightarrow I_p \cap I_q = \emptyset,$$

  mit der Prozessmenge P.
- Bei guter Lastverteilung sollten die Indexmengen I_p, $p \in P$, jeweils (nahezu) gleich viele Elemente enthalten.
- Prozess $p \in P$ speichert somit die Komponenten x_i, $i \in I_p$, des Vektors x.
- In jedem Prozess möchten wir wieder mit einer zusammenhängenden Indexmenge $\tilde{I}_p$ arbeiten, die bei 0 beginnt, d.h.

$$\tilde{I}_p = \{0, \ldots, |I_p| - 1\}.$$

Aufteilung von Vektoren III

Die Abbildungen

$$\begin{aligned} p: &\quad I \to P \text{ bzw.} \\ \mu: &\quad I \to \mathbf{N} \end{aligned}$$

ordnen jedem Index $i \in I$ umkehrbar eindeutig einen Prozess $p(i) \in P$ und einen lokalen Index $\mu(i) \in \tilde{I}_{p(i)}$ zu:

$$I \ni i \mapsto (p(i), \mu(i)).$$

Die Umkehrabbildung

$$\mu^{-1}: \underbrace{\bigcup_{p \in P} \{p\} \times \tilde{I}_p}_{\subset P \times \mathbf{N}} \to I$$

liefert zu jedem lokalen Index $i \in \tilde{I}_p$ und Prozess $p \in P$ den globalen Index $\mu^{-1}(p, i)$, d.h.

$$p(\mu^{-1}(p, i)) = p \text{ und } \mu(\mu^{-1}(p, i)) = i.$$

Aufteilung von Vektoren IV

Gebräuchliche Aufteilungen sind vor allem die *zyklische Aufteilung* mit[1]

$$\begin{aligned} p(i) &= i \,\%\, P \\ \mu(i) &= i \div P \end{aligned}$$

und die *blockweise Aufteilung* mit

$$p(i) = \begin{cases} i \div (B+1) & \text{falls } i < R(B+1) \\ R + (i - R(B+1)) \div B & \text{sonst} \end{cases}$$

$$\mu(i) = \begin{cases} i \,\%\, (B+1) & \text{falls } i < R(B+1) \\ (i - R(B+1)) \,\%\, B & \text{sonst} \end{cases}$$

mit $B = N \div P$ und $R = N \,\%\, P$. Hier ist die Idee, dass die ersten R Prozesse $B+1$ Indizes bekommen und die restlichen je B Indizes.

[1] $\div$ bedeutet ganzzahlige Division; % die modulo-Funktion

Aufteilung von Vektoren V

Zyklische und blockweise Aufteilung für $N = 13$ und $P = 4$:
zyklische Aufteilung:

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	1	2	3	0	1	2	3	0	1	2	3	0
$\mu(i)$:	0	0	0	0	1	1	1	1	2	2	2	2	3

z.B. $I_1 = \{1, 5, 9\}$,
$\tilde{I}_1 = \{0, 1, 2\}$.

blockweise Aufteilung

I:	0	1	2	3	4	5	6	7	8	9	10	11	12
$p(i)$:	0	0	0	0	1	1	1	2	2	2	3	3	3
$\mu(i)$:	0	1	2	3	0	1	2	0	1	2	0	1	2

z.B. $I_1 = \{4, 5, 6\}$,
$\tilde{I}_1 = \{0, 1, 2\}$.

Aufteilung von Matrizen I

- Bei einer Matrix $A \in \mathcal{R}^{N \times M}$ wird jedem Tupel $(i,j) \in I \times J$, mit $I = \{0,\ldots,N-1\}$ und $J = \{0,\ldots,M-1\}$ eine reele Zahl a_{ij} zugeordnet.
- Prinzipiell beliebige Zuordnung von Matrixelementen zu Prozessoren
- Jedoch können die einem Prozessor zugeordneten Elemente im allgemeinen *nicht* wieder als Matrix dargestellt werden.
- Ausnahme: separate Zerlegung der eindimensionalen Indexmengen I und J.
- Dazu nehmen wir an, die Prozesse seien als zweidimensionales Feld organisiert, d.h.

$$(p,q) \in \{0,\ldots,P-1\} \times \{0,\ldots,Q-1\}.$$

Aufteilung von Matrizen II

- Die Indexmengen I, J werden zerlegt in

$$I = \bigcup_{p=0}^{P-1} I_p \text{ und } J = \bigcup_{q=0}^{Q-1} I_q$$

- Prozess (p, q) ist dann für die Indizes $I_p \times I_q$ verantwortlich.
- Lokal speichert Prozess (p, q) seine Elemente dann als $\mathcal{R}(\tilde{I}_p \times \tilde{J}_q)$-Matrix.
- Die Zerlegungen von I und J werden formal durch die Abbildungen p und μ sowie q und ν beschrieben:

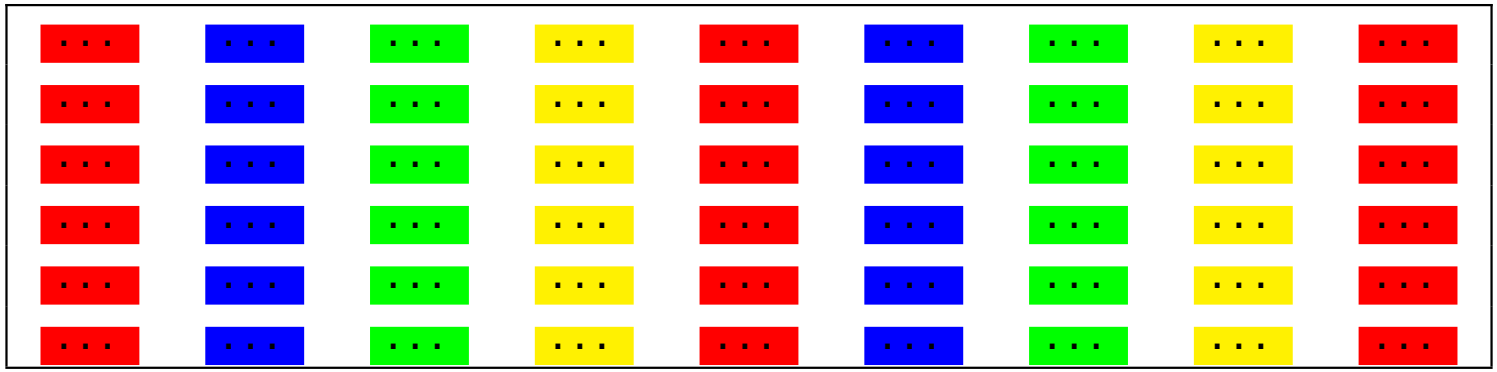
$$I_p = \{i \in I \mid p(i) = p\}, \quad \tilde{I}_p = \{n \in \mathbf{N} \mid \exists i \in I : p(i) = p \wedge \mu(i) = n\}$$
$$J_q = \{j \in J \mid q(j) = q\}, \quad \tilde{J}_q = \{m \in \mathbf{N} \mid \exists j \in J : q(j) = q \wedge \nu(j) = m\}$$

Aufteilung von Matrizen III

Beispiele zur Aufteilung einer 6×9-Matrix auf vier Prozessoren

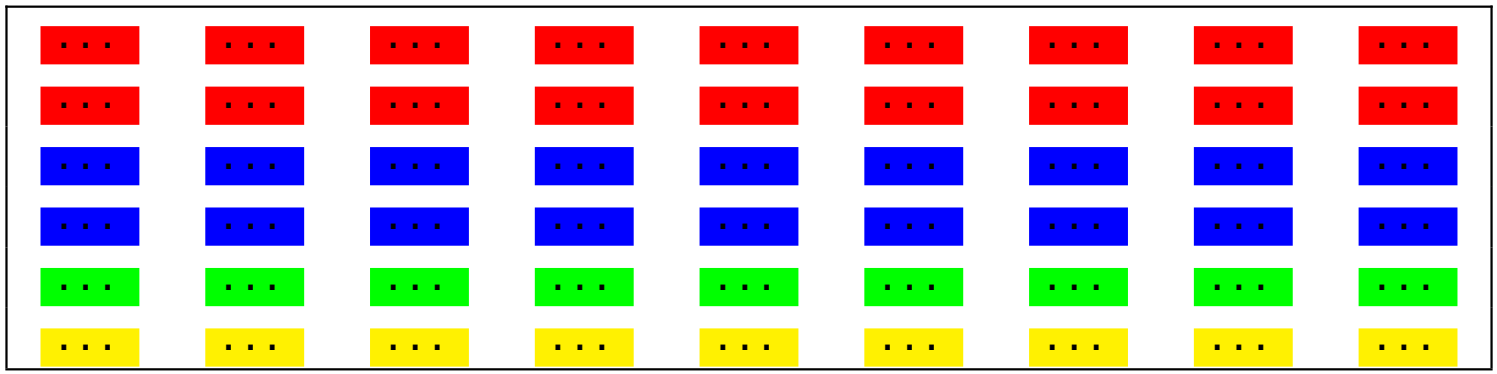
(a) $P = 1$, $Q = 4$ (Spalten), J: zyklisch:

0	1	2	3	4	5	6	7	8	J
0	1	2	3	0	1	2	3	0	q
0	0	0	0	1	1	1	1	2	ν



(b) $P = 4$,$Q = 1$ (Zeilen), I: blockweise:

0	0	0
1	0	1
2	1	0
3	1	1
4	2	0
5	3	0
I	p	μ



Aufteilung von Matrizen IV

(c) $P = 2$, $Q = 2$ (Feld), I: zyklisch, J: blockweise:

			0	1	2	3	4	5	6	7	8	J
			0	0	0	0	0	1	1	1	1	q
			0	1	2	3	4	0	1	2	3	ν
0	0	0	...	...	...	...	...	...	...	...	...	
1	1	0	...	...	...	...	...	...	...	...	...	
2	0	1	...	...	...	...	...	...	...	...	...	
3	1	1	...	...	...	...	...	...	...	...	...	
4	0	2	...	...	...	...	...	...	...	...	...	
5	1	2	...	...	...	...	...	...	...	...	...	
I	p	μ										

Aufteilung von Matrizen V

Welche Datenaufteilung ist nun die Beste?

- Generell liefert die Organisation der Prozesse als möglichst quadratisches Feld eine Aufteilung mit guter Lastverteilung.
- Wichtiger ist jedoch, dass sich unterschiedliche Aufteilungen unterschiedlich gut für verschiedene Algorithmen eignen.
- Wir werden sehen, dass sich ein Prozessfeld mit zyklischer Aufteilung sowohl der Zeilen aus auch der Spaltenindizes recht gut für die *LU*-Zerlegung eignet.
- Diese Aufteilung ist jedoch nicht optimal für die Auflösung der entstehenden Dreieckssysteme. Muss man das Gleichungssystem für viele rechte Seiten lösen, so ist ein Kompromiss anzustreben.
- Dies gilt generell für fast alle Aufgaben aus der linearen Algebra: Die Multiplikation zweier Matrizen oder die Transposition einer Matrix stellt nur einen Schritt eines größeren Algorithmus dar.
- Die Datenaufteilung kann somit nicht auf einen Teilschritt hin optimiert werden, sondern sollte einen guten Kompromiss darstellen. Eventuell kann auch überlegt werden, ob ein Umkopieren der Daten sinnvoll ist.

Transponieren einer Matrix I

Aufgabenstellung
Gegeben: $A \in \mathcal{R}^{N \times M}$, verteilt auf eine Menge von Prozessen;
Bestimme: A^T mit der selben Datenaufteilung wie A.

- Im Prinzip ist das Problem trivial.
- Wir könnten die Matrix so auf die Prozessoren aufteilen, dass nur Kommunikation mit nächsten Nachbarn notwendig ist (da die Prozesse paarweise kommunizieren).

12	1	3	5
0	13	7	9
2	6	14	11
4	8	10	15

Optimale Datenaufteilung für die Matrixtransposition (die Zahlen bezeichnen die Prozessornumern).

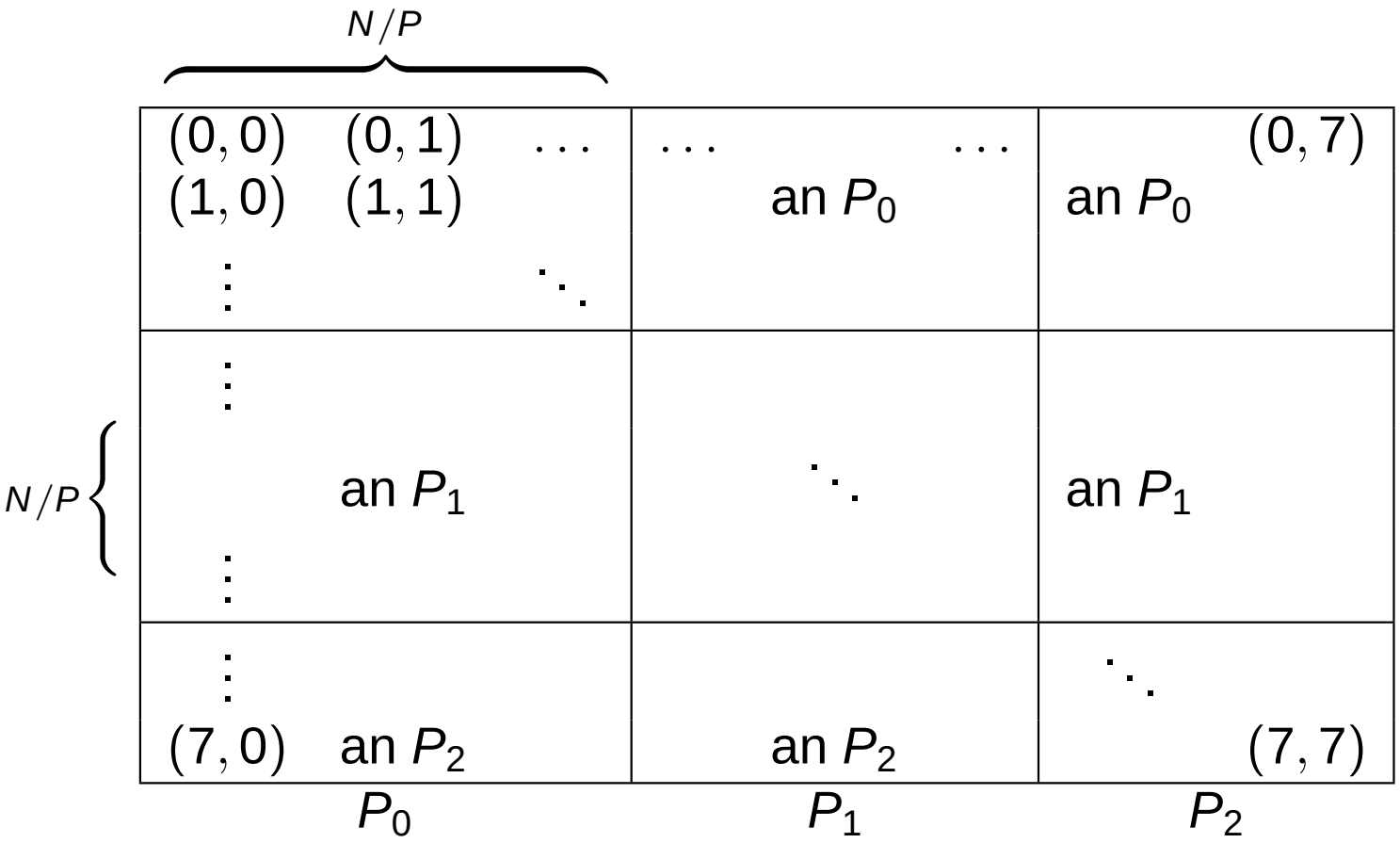
Transponieren einer Matrix II

Beispiel mit Ringtopologie:

- Offensichtlich ist nur Kommunikation zwischen direkten Nachbarn notwendig ($0 \leftrightarrow 1, 2 \leftrightarrow 3, \ldots, 10 \leftrightarrow 11$).
- Allerdings entspricht diese Datenaufteilung nicht dem Schema, das wir eben eingeführt haben und eignet sich z.B. weniger gut für die Multiplikation zweier Matrizen.

Transponieren einer Matrix: 1D Aufteilung

Betrachten wir o.B.d.A eine spaltenweise, geblockte Aufteilung



8×8-Matrix auf drei Prozessoren in spaltenweiser, geblockter Aufteilung.

Transponieren einer Matrix: 1D Aufteilung

- Offensichtlich hat in diesem Fall jeder Prozessor Daten an jeden anderen zu senden.
- Es handelt sich also um all-to-all mit persönlichen Nachrichten.
- Nehmen wir eine Hypercubestruktur als Verbindungstopologie an, so erhalten wir folgende parallele Laufzeit für eine $N \times N$-Matrix und P Prozessoren:

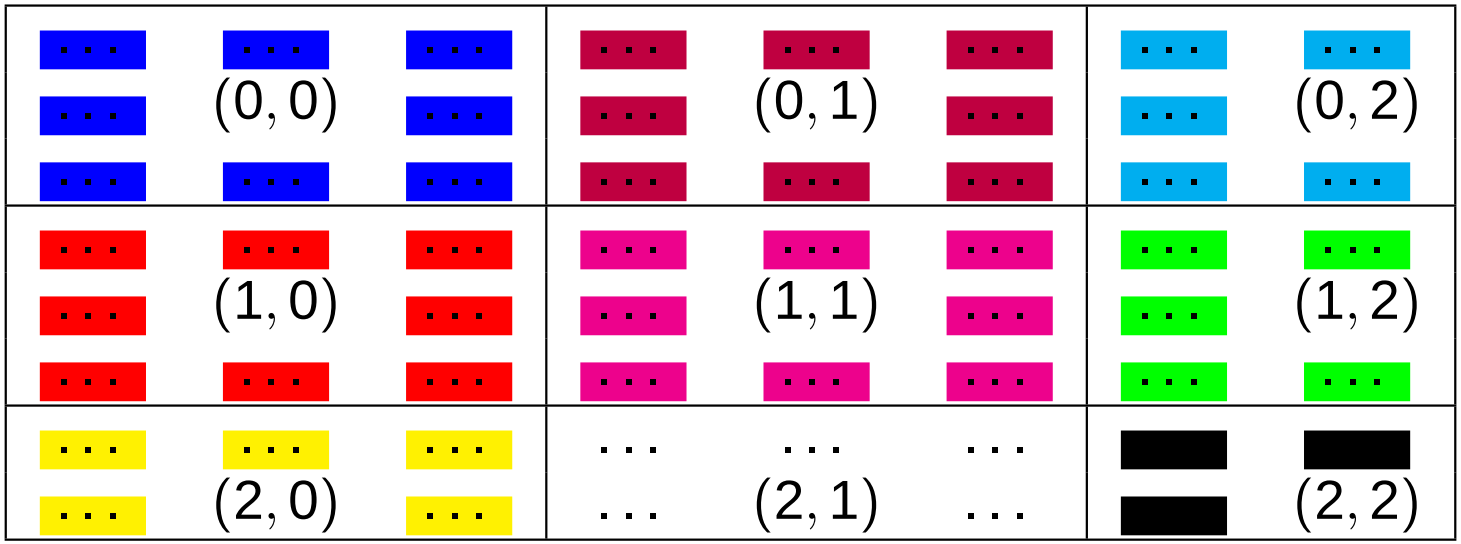
$$T_P(N,P) = \underbrace{2(t_s + t_h)\operatorname{ld} P}_{\text{Aufsetzen}} + \underbrace{t_w \frac{N^2}{P^2} P \operatorname{ld} P}_{\text{Datenübertragung}} + \underbrace{(P-1)\frac{N^2}{P^2}\frac{t_e}{2}}_{\text{Transponieren}} \approx$$

$$\approx \operatorname{ld} P(t_s + t_h)2 + \frac{N^2}{P} \operatorname{ld} P t_w + \frac{N^2}{P}\frac{t_e}{2}$$

- Selbst bei festem P und wachsendem N können wir den Anteil der Kommunikation an der Gesamtlaufzeit nicht beliebig klein machen.
- Dies ist bei allen Algorithmen zur Transposition so (auch bei der optimalen Aufteilung oben).
- Matrixtransposition besitzt also keine Isoeffizienzfunktion und ist somit nicht skalierbar.

Transponieren einer Matrix: 2D Aufteilung

Wir betrachten nun eine zweidimensionale, geblockte Aufteilung einer $N \times N$-Matrix auf ein $\sqrt{P} \times \sqrt{P}$-Prozessorfeld:

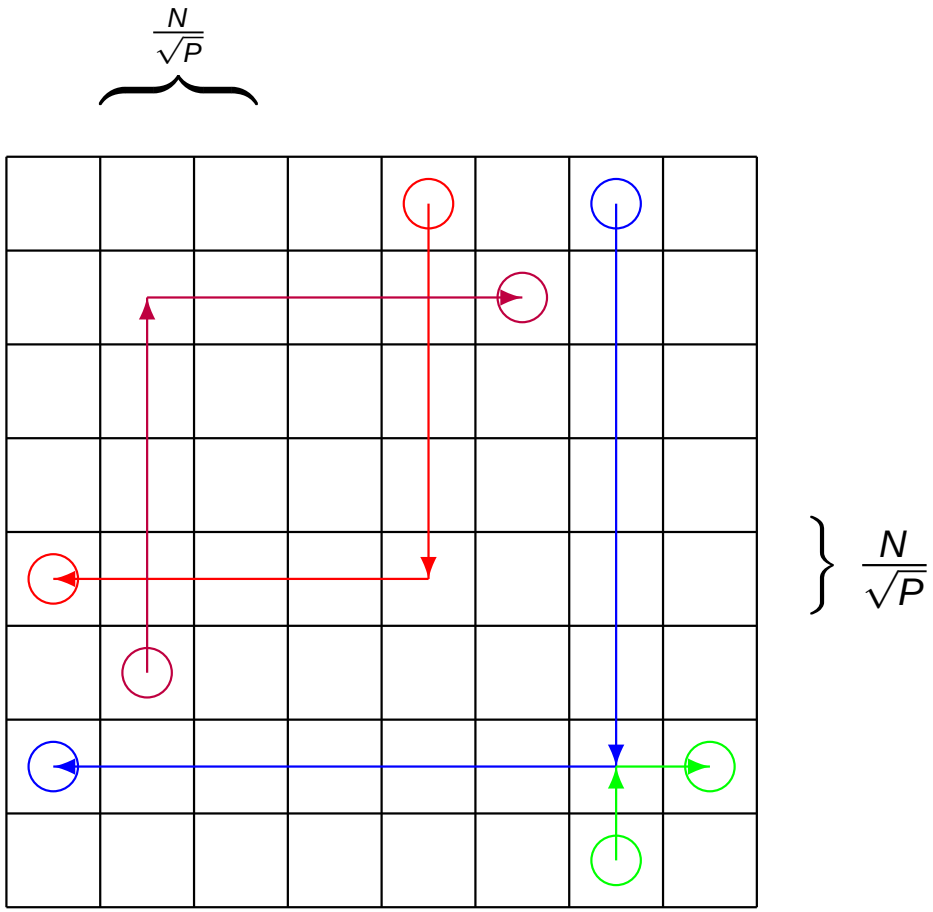


Beispiel für eine zweidimensionale, geblockte Aufteilung $N = 8$, $\sqrt{P} = 3$.

Transponieren einer Matrix

- Jeder Prozessor muss seine Teilmatrix mit genau einem anderen austauschen.
- Ein naiver Transpositionsalgorithmus für diese Konfiguration ist:
  - Prozessoren (p, q) unterhalb der Hauptdiagonalen ($p > q$) schicken Teilmatrix in der Spalte nach oben bis zu Prozessor (q, q), danach läuft die Teilmatrix nach rechts bis in die richtige Spalte zu Prozessor (q, p).
  - Entsprechend laufen Daten von Prozessoren (p, q) oberhalb der Hauptdiagonalen ($q > p$) erst in der Spalte q nach unten bis (q, q) und dann nach links bis (q, p) erreicht wird.

Transponieren einer Matrix



Diverse Wege von Teilmatrizen bei $\sqrt{P} = 8$.

Transponieren einer Matrix

- Offensichtlich leiten Prozessoren (p, q) mit $p > q$ Daten von unten nach oben bzw. rechts nach links und Prozessoren (p, q) mit $q > p$ entsprechend Daten von oben nach unten und links nach rechts.
- Bei synchroner Kommunikation sind in jedem Schritt vier Sende- bzw. Empfangsoperationen notwendig, und insgesamt braucht man $2(\sqrt{P} - 1)$ Schritte.
- Die parallele Laufzeit beträgt somit

$$\begin{aligned} T_P(N,P) &= 2(\sqrt{P}-1) \cdot 4 \left(t_s + t_h + t_w \left(\frac{N}{\sqrt{P}} \right)^2 \right) + \frac{1}{2} \left(\frac{N}{\sqrt{P}} \right)^2 t_e \approx \\ &\approx \sqrt{P} 8 (t_s + t_h) + \frac{N^2}{P} \sqrt{P} 8 t_w + \frac{N^2}{P} \frac{t_e}{2} \end{aligned}$$

- Im Vergleich zur eindimensionalen Aufteilung mit Hypercube hat man in der Datenübertragung den Faktor $\sqrt{P}$ statt $\operatorname{ld} P$.

Rekursiver Transpositionsalgorithmus

Dieser Algorithmus basiert auf folgender Beobachtung: Für eine 2×2-Blockmatrixzerlegung von A gilt

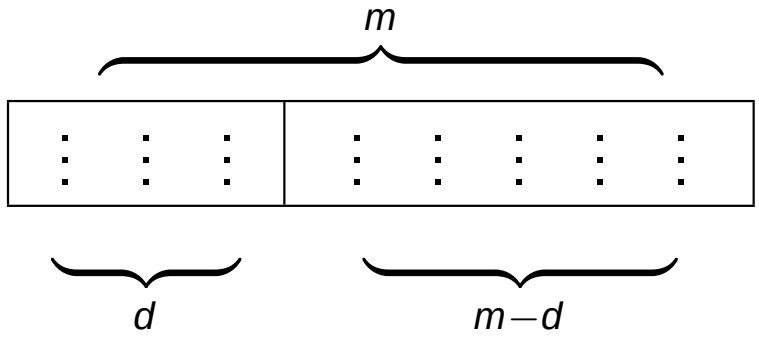
$$A^T = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}^T = \begin{pmatrix} A_{00}^T & A_{10}^T \\ A_{01}^T & A_{11}^T \end{pmatrix}$$

d.h. die Nebendiagonalblöcke tauschen die Plätze und dann muss jede Teilmatrix transponiert werden. Dies geschieht natürlich rekursiv bis eine 1×1-Matrix erreicht ist. Ist $N = 2^n$, so sind n Rekursionsschritte notwendig.

Rekursiver Transpositionsalgorithmus

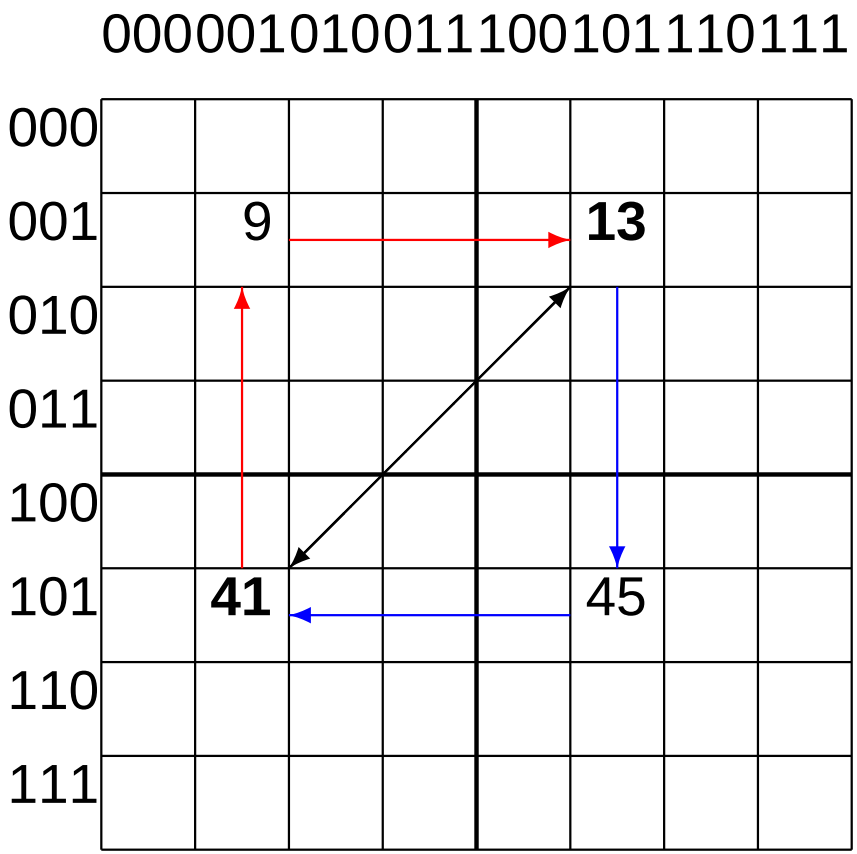
- Der Hypercube ist die ideale Verbindungstopologie für diesen Algorithmus.
- Mit $N = 2^n$ und $\sqrt{P} = 2^d$ mit $n \geq d$ geschieht die Zuordnung der Indizes $I = \{0, \ldots, N-1\}$ auf die Prozessoren mittels

$$\begin{aligned} p(i) &= i \div 2^{m-d}, \\ \mu(i) &= i \,\%\, 2^{m-d} \end{aligned}$$



- Die oberen d Bits eines Index beschreiben den Prozessor, auf den der Index abgebildet wird.
- Betrachten wir als Beispiel $d = 3$, d.h. $\sqrt{P} = 2^3 = 8$.
- Im Rekursionsschritt ist die Matrix in 2×2 Blöcke aus 4×4-Teilmatrizen zu teilen und $2 \cdot 16$ Prozessoren müssen Daten austauschen, z.B. Prozessor $101001 = 41$ und $001101 = 13$. Dies geschieht in zwei Schritten über die Prozessoren $001001 = 9$ und $101101 = 45$.
- Diese sind beide *direkte* Nachbarn der Prozessoren 41 und 13 im Hypercube.

Rekursiver Transpositionsalgorithmus



Kommunikation im rekursiven Transpositionsalgorithmus bei $d = 3$.

Der rekursive Transpositionsalgorithmus arbeitet nun rekursiv über die Prozessortopologie. Ist *ein* Prozessor erreicht, wird mit dem sequentiellen Algorithmus transponiert. Die parallele Laufzeit beträgt somit

$$T_P(N,P) = \mathrm{ld}\, P(t_s + t_h)2 + \frac{N^2}{P}\,\mathrm{ld}\,\sqrt{P}2t_w + \frac{N^2}{P}\frac{t_e}{2}$$

Rekursiver Transpositionsalgorithmus

Programm (Rekursiver Transpositionsalgorithmus auf Hypercube)

```
parallel recursive-transpose
{
    const int d = ..., n = ...;
    const int P = 2^d, N = 2^n;

    process Π[int (p,q) ∈ {0,...,2^d − 1} × {0,...,2^d − 1}]
    {
        Matrix A, B;                                   // A ist die Eingabematrix
        void rta(int r, int s, int k)
        {
            if (k == 0) { A = A^T; return; }
            int i = p − r, j = q − s, l = 2^(k−1);
            if (i < l)
            {
                if (j < l)
                {                                      // links oben
                    recv(B,Π_{p+l,q}); send(B,Π_{p,q+l});
                    rta(r,s,k − 1);
                }
                else
                {                                      // rechts oben
                    send(A,Π_{p+l,q}); recv(A,Π_{p,q−l});
                    rta(r,s + l,k − 1);
                }
            }
            ...
}
```

Rekursiver Transpositionsalgorithmus cont.

Programm (Rekursiver Transpositionsalgorithmus auf Hypercube cont.)

```
parallel recursive-transpose cont.
{
                ...
                else
                {
                    if (j < l) {                          // links unten
                        send(A,Π_{p-l,q}); recv(A,Π_{p,q+l});
                        rta(r + l,s,k − 1);
                    }
                    else
                    {                                     // rechts unten
                        recv(B,Π_{p-l,q}); send(B,Π_{p,q-l});
                        rta(r + l,s + l,k − 1);
                    }
                }
            }
            rta(0,0,d);
        }
}
```