Algorithmen für vollbesetzte Matrizen III

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

Datenparallele Algorithmen für vollbesetzte Matrizen

- LU-Zerlegung

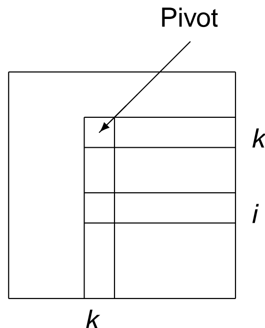
LU-Zerlegung: Formulierung

Zu lösen sei das lineare Gleichungssystem

$$Ax = b \tag{1}$$

mit einer $N \times N$-Matrix A und entsprechenden Vektoren x und b.
Gauß'sche Eliminationsverfahren (Sequentielles Verfahren)

(1) **for** $(k = 0; k < N; k++)$
(2) **for** $(i = k + 1; i < N; i++)$ {
(3) $l_{ik} = a_{ik} / a_{kk}$;
(4) **for** $(j = k + 1; j < N; j++)$
(5) $a_{ij} = a_{ij} - l_{ik} \cdot a_{kj}$;
(6) $b_i = b_i - l_{ik} \cdot b_k$;
}



transformiert das Gleichungssystem (1) in das Gleichungssystem

$$Ux = d \tag{2}$$

mit einer oberen Dreiecksmatrix U.

LU-Zerlegung: Eigenschaften

Obige Formulierung hat folgende Eigenschaften:

- Die Matrixelemente a_{ij} für $j \geq i$ enthalten die entsprechenden Einträge von U, d.h. A wird überschrieben.
- Vektor b wird mit den Elementen von d überschrieben.
- Es wird angenommen, das a_{kk} in Zeile (3) immer von Null verschieden ist (keine Pivotisierung).

LU-Zerlegung: Ableitung aus Gauß-Elimination

Die *LU*-Zerlegung läßt sich aus der Gauß-Elimination ableiten:

- Jeder einzelne Transformationsschritt, der für festes k und i aus den Zeilen (3) bis (5) besteht, lässt sich als Multiplikation des Gleichungssystems mit einer Matrix schreiben:

$$\hat{L}_{ik} = \quad \begin{matrix} \\ \\ \\ \\ i \\ \\ \end{matrix} \overset{\qquad\qquad k}{\begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & -l_{ik} & & \ddots & \\ & & & & & 1 \end{pmatrix}} = I - l_{ik} E_{ik}$$

(E_{ik} ist die Matrix deren Element $e_{ik} = 1$ ist, und die sonst aus lauter Nullen besteht) von links verstehen, mit l_{ik} aus Zeile (3) des Gauß'schen Eliminationsverfahrens.

LU-Zerlegung

- Somit gilt

$$\hat{L}_{N-1,N-2} \cdot \dots \cdot \hat{L}_{N-1,0} \cdot \dots \cdot \hat{L}_{2,0}\hat{L}_{1,0}A = \\ = \hat{L}_{N-1,N-2} \cdot \dots \cdot \hat{L}_{N-1,0} \cdot \dots \cdot \hat{L}_{2,0}\hat{L}_{1,0}b \tag{3}$$

  und wegen (2) gilt

$$\hat{L}_{N-1,N-2} \cdot \dots \cdot \hat{L}_{N-1,0} \cdot \dots \cdot \hat{L}_{2,0}\hat{L}_{1,0}A = U. \tag{4}$$

LU-Zerlegung: Eigenschaften

- Es gelten folgende Eigenschaften:
  1. $\hat{L}_{ik} \cdot \hat{L}_{i',k'} = I - l_{ik} E_{ik} - l_{i'k'} E_{i'k'}$ für $k \neq i'$ ($\Rightarrow E_{ik} E_{i'k'} = 0$) .
  2. $(I - l_{ik} E_{ik})(I + l_{ik} E_{ik}) = I$ für $k \neq i$, d.h. $\hat{L}_{ik}^{-1} = I + l_{ik} E_{ik}$.
- Wegen 2 und der Beziehung (4)

$$A = \underbrace{\hat{L}_{1,0}^{-1} \cdot \hat{L}_{2,0}^{-1} \cdot \dots \cdot \hat{L}_{N-1,0}^{-1} \cdot \dots \cdot \hat{L}_{N-1,N-2}^{-1}}_{=:L} U = LU \tag{5}$$

- Wegen 1, was sinngemäß auch für $\hat{L}_{ik}^{-1} \cdot \hat{L}_{i'k'}^{-1}$ gilt, ist L eine untere Dreiecksmatrix mit $L_{ik} = l_{ik}$ für $i > k$ und $L_{ii} = 1$.
- Den Algorithmus zur LU-Zerlegung von A erhält man durch weglassen von Zeile (6) im Gauß-Algorithmus oben. Die Matrix L wird im unteren Dreieck von A gespeichert.

LU-Zerlegung: Parallele Variante mit zeilenweiser Aufteilung

Zeilenweise Aufteilung einer $N \times N$-Matrix für den **Fall** $N = P$:

P_0								
P_1								
P_2			(k,k)					
P_3								
P_4								
P_5								
P_6								
P_7								

- Im Schritt k teilt Prozessor P_k die Matrixelemente $a_{k,k}, \ldots, a_{k,N-1}$ allen Prozessoren P_j mit $j > k$ mit, und diese eliminieren in ihrer Zeile.
- Parallele Laufzeit:

$$T_P(N) = \underbrace{\sum_{m=N-1}^{1}}_{\text{Anz. zu eliminierender Zeilen}} (t_s + t_h + \underbrace{t_w \cdot m}_{\text{Rest der Zeile } k}) \underbrace{\operatorname{ld} N}_{\text{Broadcast}} + \underbrace{m 2 t_f}_{\text{Elimination}} \qquad (6)$$

$$= \frac{(N-1)N}{2} 2t_f + \frac{(N-1)N}{2} \operatorname{ld} N t_w + N \operatorname{ld} N (t_s + t_h)$$

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

LU-Zerlegung: Analyse der parallelen Variante

- Sequentielle Laufzeit der LU-Zerlegung:

$$T_S(N) = \sum_{m=N-1}^{1} \underbrace{m}_{\text{Zeilen sind zu elim.}} \underbrace{2mt_f}_{\text{Elim. einer Zeile}} = \tag{7}$$

$$= 2t_f \frac{(N-1)\,(N(N-1)+1)}{6} \approx \frac{2}{3} N^3 t_f .$$

- Wie man aus (6) sieht, wächst $N \cdot T_P = \Omega(N^3 \operatorname{ld} N)$ (beachte $P = N$!) asymptotisch schneller als $T_S = \Omega(N^3)$.
- Der Algorithmus ist also nicht kostenoptimal (Effizienz kann für $P = N \longrightarrow \infty$ nicht konstant gehalten werden).
- Dies liegt daran, dass Prozessor P_k in seinem Broadcast wartet, bis alle anderen Prozessoren die Pivotzeile erhalten haben.
- Wir beschreiben nun eine *asynchrone* Variante, bei der ein Prozessor sofort losrechnet, sobald er die Pivotzeile erhalten hat.

LU-Zerlegung: Asynchrone Variante

Programm (Asynchrone LU-Zerlegung für $P = N$)

```
parallel lu-1
{
    const int N = ...;
    process Π[int p ∈ {0, ..., N − 1}]
    {
        double A[N];                                   // meine Zeile
        double rr[2][N];                               // Puffer für Pivotzeile
        double *r;
        msgid m;
        int j, k;

        if (p > 0) m =arecv(Π_{p−1},rr[0]);
        for (k = 0; k < N − 1; k + +)
        {
            if (p == k) send(Π_{p+1},A);
            if (p > k)
            {
                while (¬success(m));                   // warte auf Pivotzeile
                if (p < N − 1) asend(Π_{p+1},rr[k%2]);
                if (p > k + 1) m =arecv(Π_{p−1},rr[(k + 1)%2]);
                r = rr[k%2];
                A[k] = A[k]/r[k];
                for (j = k + 1; j < N; j + +)
                    A[j] = A[j] − A[k] · r[j];
            }
        }
    }
}
```

LU-Zerlegung: Zeitlicher Ablauf

Wie verhält sich der parallele Algorithmus über die Zeit?

Zeit →

P_0: **send** $k=0$

P_1: **recv** $k=0$, **send** $k=0$; Eliminieren $k=0$, **send** $k=1$

P_2: **recv** $k=0$, **send** $k=0$, **recv** $k=1$, **send** $k=1$; Eliminieren $k=0$, Eliminieren $k=1$

P_3: **recv** $k=0$, **send** $k=0$, **recv** $k=1$, **send** $k=1$; Eliminieren $k=0$, Eliminieren $k=1$

LU-Zerlegung: Parallele Laufzeit und Effizienz

- Nach einer Einlaufzeit von p Nachrichtenübertragungen ist die Pipeline gefüllt, und alle Prozessoren sind ständig mit eliminieren beschäftigt. Somit erhält man folgende Laufzeit ($N = P$, immer noch!):

$$
\begin{aligned}
T_P(N) &= \underbrace{(N-1)(t_s + t_h + t_w N)}_{\text{Einlaufzeit}} + \sum_{m=N-1}^{1} \Big(\underbrace{2mt_f}_{\text{Elim.}} + \underbrace{t_s}_{\substack{\text{Aufsetzzeit}\\\text{(Rech-}\\\text{nen+}\textbf{send}\\\text{parallel)}}} \Big) = \qquad (8)\\
&= \frac{(N-1)N}{2} 2t_f + (N-1)(2t_s + t_h) + N(N-1)t_w \approx \\
&\approx N^2 t_f + N^2 t_w + N(2t_s + t_h).
\end{aligned}
$$

- Den Faktor ld N von (6) sind wir somit los. Für die Effizienz erhalten wir

$$
\begin{aligned}
E(N,P) &= \frac{T_S(N)}{NT_P(N,P)} = \frac{\frac{2}{3}N^3 t_f}{N^3 t_f + N^3 t_w + N^2(2t_s + t_h)} = \qquad (9)\\
&= \frac{2}{3} \frac{1}{1 + \frac{t_w}{t_f} + \frac{2t_s + t_h}{Nt_f}}.
\end{aligned}
$$

- Die Effizienz ist also maximal $\frac{2}{3}$. Dies liegt daran, dass Prozessor k nach k Schritten idle bleibt. Verhindern lässt sich dies durch mehr Zeilen pro Prozessor (gröbere Granularität).

LU-Zerlegung: Der Fall $N \gg P$

LU-Zerlegung für den **Fall** $N \gg P$:

- Programm 0.1 von oben lässt sich leicht auf den Fall $N \gg P$ erweitern. Dazu werden die *Zeilen* zyklisch auf die Prozessoren $0, \ldots, P-1$ verteilt. Die aktuelle Pivotzeile erhält ein Prozessor vom Vorgänger im Ring.
- Die parallele Laufzeit ist

$$\begin{aligned} T_P(N,P) &= \underbrace{(P-1)(t_s + t_h + t_w N)}_{\text{Einlaufzeit der Pipeline}} + \sum_{m=N-1}^{1} \Big(\underbrace{\frac{m}{P}}_{\text{Zeilen pro Prozessor}} \cdot m 2 t_f + t_s \Big) = \\ &= \frac{N^3}{P}\frac{2}{3} t_f + N t_s + P(t_s + t_h) + N P t_w \end{aligned}$$

und somit hat man die Effizienz

$$E(N,P) = \frac{1}{1 + \frac{P t_s}{N^2 \frac{2}{3} t_f} + \ldots}.$$

LU-Zerlegung: Der Fall $N \gg P$

- Wegen der zeilenweisen Aufteilung gilt jedoch in der Regel, dass einige Prozessoren eine Zeile mehr haben als andere.
- Eine noch bessere Lastverteilung erzielt man durch eine zweidimensionale Verteilung der Matrix. Dazu nehmen wir an, dass die Aufteilung der Zeilen- und Spaltenindexmenge

$$I = J = \{0, \dots, N-1\}$$

durch die Abbildungen p und μ für I und q und ν für J vorgenommen wird.

LU-Zerlegung: Allgemeine Aufteilung

- Die nachfolgende Implementierung wird vereinfacht, wenn wir zusätzlich noch annehmen, dass die Datenverteilung folgende Monotoniebedingung erfüllt:

$$\begin{array}{lll} \text{Ist } i_1 < i_2 \text{ und } p(i_1) = p(i_2) & \text{so gelte} & \mu(i_1) < \mu(i_2) \\ \text{ist } j_1 < j_2 \text{ und } q(j_1) = q(j_2) & \text{so gelte} & \nu(j_1) < \nu(j_2) \end{array}$$

- Damit entspricht einem Intervall von globalen Indizes $[i_{min}, N-1] \subseteq I$ eine Anzahl von Intervallen lokaler Indizes in verschiedenen Prozessoren, die wie folgt berechnet werden können:

$$\begin{aligned} & \text{Setze} \\ \tilde{I}(p,k) &= \{m \in \mathbf{N} \mid \exists i \in I, i \geq k \colon p(i) = p \wedge \mu(i) = m\} \\ & \text{und} \\ ibegin(p,k) &= \begin{cases} \min \tilde{I}(p,k) & \text{falls } \tilde{I}(p,k) \neq \emptyset \\ N & \text{sonst} \end{cases} \\ iend(p,k) &= \begin{cases} \max \tilde{I}(p,k) & \text{falls } \tilde{I}(p,k) \neq \emptyset \\ 0 & \text{sonst.} \end{cases} \end{aligned}$$

- Dann kann man eine Schleife

  for $(i = k; i < N; i{+}{+}) \ldots$

  ersetzen durch lokale Schleifen in den Prozessoren p der Gestalt

  for $(i = ibegin(p,k); i \leq iend(p,k); i{+}{+}) \ldots$

LU-Zerlegung: Allgemeine Aufteilung

Analog verfährt man mit den Spaltenindizes:

$$
\begin{aligned}
&\text{Setze} \\
\tilde{J}(q,k) &= \{n \in \mathbf{N} \mid \exists j \in j, j \geq k \colon q(j) = q \wedge \nu(j) = n\} \\
&\text{und} \\
jbegin(q,k) &= \begin{cases} \min \tilde{J}(q,k) & \text{falls } \tilde{J}(q,k) \neq \emptyset \\ N & \text{sonst} \end{cases} \\
jend(q,k) &= \begin{cases} \max \tilde{J}(q,k) & \text{falls } \tilde{J}(q,k) \neq \emptyset \\ 0 & \text{sonst.} \end{cases}
\end{aligned}
$$

Damit können wir zur Implementierung der *LU*-Zerlegung für eine allgemeine Datenaufteilung schreiten.

LU-Zerlegung: Algorithmus mit allg. Aufteilung

Programm (Synchrone *LU*-Zerlegung mit allg. Datenaufteilung)

```
parallel lu-2
{
    const int N = ..., √P = ...;

    process Π[int (p, q) ∈ {0, ..., √P − 1} × {0, ..., √P − 1}]
    {
        double A[N/√P][N/√P], r[N/√P], c[N/√P];
        int i, j, k;

        for (k = 0; k < N − 1; k++)
        {
            I = μ(k); J = ν(k);                           // lokale Indizes

            // verteile Pivotzeile:
            if (p == p(k))
            {                                             // Ich habe Pivotzeile
                for (j = jbegin(q, k); j ≤ jend(q, k); j++)
                    r[j] = A[I][j];                       // kopiere Segment der Pivotzeile
                Sende r an alle Prozessoren (x, q) ∀x ≠ p
            }
            else recv(Π_{p(k),q}, r);

            // verteile Pivotspalte:
            if (q == q(k))
            {                                             // Ich habe Teil von Spalte k
                for (i = ibegin(p, k + 1); i ≤ iend(p, k + 1); i++)
                    c[i] = A[i][J] = A[i][J] / r[J];
                Sende c an alle Prozessoren (p, y) ∀y ≠ q
            }
            else recv(Π_{p,q(k)}, c);

            // Elimination:
            for (i = ibegin(p, k + 1); i ≤ iend(p, k + 1); i++)
                for (j = jbegin(q, k + 1); j ≤ jend(q, k + 1); j++)
                    A[i][j] = A[i][j] − c[i] · r[j];
        }
    }
}
```

LU-Zerlegung: Analyse I

- Analysieren wir diese Implementierung (synchrone Variante):

$$T_P(N,P) = \sum_{m=N-1}^{1} \underbrace{\left(t_s + t_h + t_w \frac{m}{\sqrt{P}}\right) \operatorname{ld}\sqrt{P}\, 2}_{\substack{\text{Broadcast}\\ \text{Pivotzeile/-spalte}}} + \left(\frac{m}{\sqrt{P}}\right)^2 2t_f =$$

$$= \frac{N^3}{P}\frac{2}{3}t_f + \frac{N^2}{\sqrt{P}} \operatorname{ld}\sqrt{P} t_w + N \operatorname{ld}\sqrt{P}\, 2(t_s + t_h).$$

- Mit $W = \frac{2}{3}N^3 t_f$, d.h. $N = \left(\frac{3W}{2t_f}\right)^{\frac{1}{3}}$, gilt

$$T_P(W,P) = \frac{W}{P} + \frac{W^{\frac{2}{3}}}{\sqrt{P}} \operatorname{ld}\sqrt{P} \frac{3^{2/3} t_w}{(2t_f)^{\frac{2}{3}}} + W^{\frac{1}{3}} \operatorname{ld}\sqrt{P} \frac{3^{1/3} 2(t_s + t_h)}{(2t_f)^{\frac{1}{3}}}.$$

LU-Zerlegung: Analyse II

- Die Isoeffizienzfunktion erhalten wir aus $PT_P(W,P) - W \stackrel{!}{=} KW$:

$$\sqrt{P} W^{\frac{2}{3}} \operatorname{ld} \sqrt{P} \frac{3^{2/3} t_w}{(2t_f)^{\frac{2}{3}}} = KW$$

$$\iff \quad W = P^{\frac{3}{2}} (\operatorname{ld} \sqrt{P})^3 \frac{9 t_w^3}{4 t_f^2 K^3}$$

also

$$W \in O(P^{3/2} (\operatorname{ld} \sqrt{P})^3).$$

- Programm 0.2 kann man auch in einer asynchronen Variante realisieren. Dadurch können die Kommunikationsanteile wieder effektiv hinter der Rechnung versteckt werden.

LU-Zerlegung: Pivotisierung

- Die *LU*-Faktorisierung allgemeiner, invertierbarer Matrizen erfordert Pivotisierung und ist auch aus Gründen der Minimierung von Rundungsfehlern sinnvoll.
- Man spricht von voller Pivotisierung, wenn das Pivotelement im Schritt k aus allen $(N-k)^2$ verbleibenden Matrixelementen ausgewählt werden kann, bzw. von teilweiser Pivotisierung (*engl.* „partial pivoting“), falls das Pivotelement nur aus einem Teil der Elemente ausgewählt werden darf. Üblich ist z.B. das maximale Zeilen- oder Spaltenpivot, d.h. man wählt a_{ik}, $i \geq k$, mit $|a_{ik}| \geq |a_{mk}| \quad \forall m \geq k$.
- Die Implementierung der *LU*-Zerlegung muss nun die Wahl des neuen Pivotelements bei der Elimination berücksichtigen. Dazu hat man zwei Möglichkeiten:
  - Explizites Vertauschen von Zeilen und/oder Spalten: Hier läuft der Rest des Algorithmus dann unverändert (bei Zeilenvertauschungen muss auch die rechte Seite permutiert werden).
  - Man bewegt die eigentlichen Daten nicht, sondern merkt sich nur die Vertauschung von Indizes (in einem Integer-Array, das alte Indizes in neue umrechnet.

LU-Zerlegung: Pivotisierung

- Die parallelen Versionen besitzen unterschiedliche Eignung für Pivotisierung. Folgende Punkte sind für die parallele *LU*-Zerlegung mit teilweiser Pivotisierung zu bedenken:
  - Ist der Bereich in dem das Pivotelement gesucht wird in einem einzigen Prozessor (z.B. zeilenweise Aufteilung mit maximalem Zeilenpivot) gespeichert, so ist die Suche sequentiell durchzuführen. Im anderen Fall kann auch sie parallelisiert werden.
  - Allerdings erfordert diese parallele Suche nach dem Pivotelement natürlich Kommunikation (und somit Synchronisation), die das Pipelining in der asynchronen Variante unmöglich macht.
  - Permutieren der Indizes ist schneller als explizites Vertauschen, insbesondere, wenn das Vertauschen den Datenaustausch zwischen Prozessoren erfordert. Allerdings kann dadurch eine gute Lastverteilung zerstört werden, falls zufällig die Pivotelemente immer im gleichen Prozessor liegen.
- Einen recht guten Kompromiss stellt die zeilenweise zyklische Aufteilung mit maximalem Zeilenpivot und explizitem Vertauschen dar, denn:
  - Pivotsuche in der Zeile k ist zwar sequentiell, braucht aber nur $O(N-k)$ Operationen (gegenüber $O((N-k)^2/P)$ für die Elimination); ausserdem wird das Pipelining nicht gestört.
  - explizites Vertauschen erfordert nur Kommunikation des Index der Pivotspalte, aber keinen Austausch von Matrixelementen zwischen Prozessoren. Der Pivotspaltenindex wird mit der Pivotzeile geschickt.
  - Lastverteilung wird von der Pivotisierung nicht beeinflusst.

LU-Zerlegung: Lösen der Dreieckssysteme

- Wir nehmen an, die Matrix A sei in $A = LU$ faktorisiert wie oben beschrieben, und wenden uns nun der Lösung eines Systems der Form

$$LUx = b \tag{10}$$

zu. Dies geschieht in zwei Schritten:

$$\begin{aligned} Ly &= b \quad (11)\\ Ux &= y. \quad (12) \end{aligned}$$

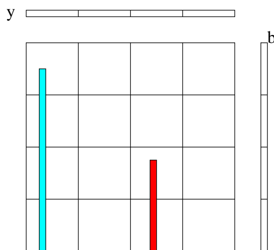
- Betrachten wir kurz den sequentiellen Algorithmus:

```
// Ly = b:
for (k = 0; k < N; k++) {
    y_k = b_k;          l_kk = 1
    for (i = k+1; i < N; i++)
        b_i = b_i - a_ik y_k;
}
// Ux = y:
for (k = N-1; k >= 0; k--) {
    x_k = y_k / a_kk
    for (i = 0; i < k; i++)
        y_i = y_i - a_ik x_k;
}
```

- Dies ist eine spaltenorientierte Version, denn nach Berechnung von y_k bzw. x_k wird sofort die rechte Seite für alle Indizes $i > k$ bzw. $i < k$ modifiziert.

LU-Zerlegung: Parallelisierung

- Die Parallelisierung wird sich natürlich an der Datenverteilung der *LU*-Zerlegung orientieren müssen (falls man ein Umkopieren vermeiden will, was wegen $O(N^2)$ Daten und $O(N^2)$ Operationen sinnvoll erscheint). Betrachten wir hierzu eine zweidimensionale blockweise Aufteilung der Matrix:



- Die Abschnitte von b sind über Prozessorzeilen kopiert und die Abschnitte von y sind über Prozessorspalten kopiert. Offensichtlich können nach Berechnung von y_k nur die Prozessoren der Spalte $q(k)$ mit der Modifikation von b beschäftigt werden. Entsprechend können bei der Auflösung von $Ux = y$ nur die Prozessoren $(*, q(k))$ zu einer Zeit beschäftigt werden. Bei einer zeilenweise Aufteilung ($Q = 1$) sind somit immer alle Prozessoren beschäftigt.

LU-Zerlegung: Parallelisierung bei allg. Aufteilung

Programm (Auflösen von $LUx = b$ bei allgemeiner Datenaufteilung)

```
parallel lu-solve
{
    const int N = ...;
    const int √P = ...;
    process Π[int (p, q) ∈ {0, ..., √P − 1} × {0, ..., √P − 1}]
    {
        double A[N/√P][N/√P];
        double b[N/√P]; x[N/√P];
        int i, j, k, I, K;

        // Löse Ly = b, speichere y in x.
        // b spaltenweise verteilt auf Diagonalprozessoren.
        if (p == q) sende b an alle (p, ∗);
        for (k = 0; k < N; k + +)
        {
            I = μ(k); K = ν(k);
            if(q(k) == q)                                    // nur die haben was zu tun
            {
                if (k > 0 ∧ q(k) ≠ q(k − 1))                 // brauche aktuelle b
                    recv(Π_{p,q(k−1)}, b);
                if (p(k) == p)
                {                                            // habe Diagonalelement
                    x[K] = b[I];                             // speichere y in x!
                    sende x[K] an alle (∗, q);
                }
                else recv(Π_{p(k),q(k)}, x[k]);
                for (i = ibegin(p, k + 1); i ≤ iend(p, k + 1); i + +)
                    b[i] = b[i] − A[i][K] · x[K];
                if (k < N − 1 ∧ q(k + 1) ≠ q(k))
                    send(Π_{p,q(k+1)}, b);
            }
        }
        ...
}
```

LU-Zerlegung: Parallelisierung

Programm (Auflösen von $LUx = b$ bei allgemeiner Datenaufteilung cont.)

```
parallel lu-solve cont.
{
        ...
        // { y steht in x; x ist spaltenverteilt und zeilenkopiert. Für Ux = y wollen wir y in b speichern Es ist also x
           { in b umzukopieren, wobei b zeilenverteilt und spaltenkopiert sein soll.
        for (i = 0; i < N/√P; i++)                                          // löschen
                b[i] = 0;
        for (j = 0; j < N − 1; j++)
                if (q(j) = q ∧ p(j) = p)                                    // einer muss es sein
                        b[μ(j)] = x[ν(j)];
        summiere b über alle (p, ∗), Resultat in (p, p);

        // Auflösen von Ux = y (y ist in b gespeichert)
        if (p == q) sende b and alle (p, ∗);
        for (k = N − 1; k ≥ 0; k − −)
        {
                I = μ(k); K = ν(k);
                if (q(k) == q)
                {
                        if (k < N − 1 ∧ q(k) ≠ q(k + 1))
                                recv(Π_{p,q(k+1)}, b);
                        if (p(k) == p)
                        {
                                x[K] = b[I]/A[I][K];
                                sende x[K] an alle (∗, q);
                        }
                        else recv(Π_{p(k),q(k)}, x[K]);
                        for (i = ibegin(p, 0); i ≤ iend(p, 0); i++)
                                b[i] = b[i] − A[i][K] · x[K];
                        if (k > 0 ∧ q(k) ≠ q(k − 1))
                                send(Π_{p,q(k−1)}, b);
                }
        }
    }
}
```

LU-Zerlegung: Parallelisierung

- Da zu einer Zeit immer nur $\sqrt{P}$ Prozessoren beschäftigt sind, kann der Algorithmus nicht kostenoptimal sein. Das Gesamtverfahren aus *LU*-Zerlegung und Auflösen der Dreieckssysteme kann aber immer noch isoeffizient skaliert werden, da die sequentielle Komplexität des Auflösens nur $O(N^2)$ gegenüber $O(N^3)$ für die Faktorisierung ist.
- Muss man ein Gleichungssystem für viele rechte Seiten lösen, sollte man ein rechteckiges Prozessorfeld $P \times Q$ mit $P > Q$ verwenden, oder im Extremfall $Q = 1$ wählen. Falls Pivotisierung erforderlich ist, war das ja ohnehin eine sinnvolle Konfiguration.