

Paralleles Höchstleistungsrechnen



Lösung tridiagonaler und dünnbesetzter linearer Gleichungssysteme

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 425
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

20. Januar 2010



Lösung tridiagonaler und dünnbesetzter linearer Gleichungssysteme

- Optimaler sequentieller Algorithmus
- Zyklische Reduktion
- Gebietszerlegung
- LU -Zerlegung dünnbesetzter Matrizen
- Parallelisierung



Optimaler sequentieller Algorithmus

- Als Extremfall eines dünnbesetzten Gleichungssystems betrachten wir

$$Ax = b$$

mit $A \in \mathcal{R}^{N \times N}$ tridiagonal.

$$(1) \quad \begin{pmatrix} * & * & & & & & \\ * & * & * & & & & \\ & * & * & * & & & \\ & & * & * & * & & \\ & & & * & * & * & \\ & & & & * & * & * \\ & & & & & * & * \end{pmatrix}$$

- Der optimale Algorithmus ist die Gauß-Elimination, manchmal auch Thomas-Algorithmus genannt.



Optimaler sequentieller Algorithmus

- Gauß-Elimination für tridiagonale Systeme

// Vorwärtselimination (diesmal lösen, nicht LU -Zerlegen):

for ($k = 0$; $k < N - 1$; $k++$) {

$$l = a_{k+1,k}/a_{k,k};$$

$$a_{k+1,k+1} = a_{k+1,k+1} - l \cdot a_{k,k+1}$$

$$b_{k+1} = b_{k+1} - l \cdot b_k;$$

} // $(N - 1) \cdot 5$ Rechenoperationen

// Rückwärtseinsetzen:

$$x_{N-1} = b_{N-1}/a_{N-1,N-1};$$

for ($k = N - 2$; $k \geq 0$; $k--$) {

$$x_k = (b_k - a_{k,k+1} \cdot x_{k+1})/a_{k,k};$$

} // $(N - 1)3 + 1$ Rechenoperationen

- Die sequentielle Komplexität beträgt

$$T_S = 8Nt_f$$

Offensichtlich ist der Algorithmus streng sequentiell!



Zyklische Reduktion

- Betrachte eine Tridiagonalmatrix mit $N = 2M$ (N gerade).
- *Idee*: Eliminiere in jeder *geraden* Zeile k die Elemente $a_{k-1,k}$ und $a_{k+1,k}$ mit Hilfe der ungeraden Zeilen darüber bzw. darunter.
- Jede gerade Zeile ist damit nur noch mit der vorletzten und übernächsten gekoppelt; da diese gerade sind, wurde die Dimension auf $M = N/2$ reduziert.
- Das verbleibende System ist wieder tridiagonal, und die Idee kann rekursiv angewandt werden.

0	*	*	□			
1	*	*	*			
2	□	*	*	□		
3		*	*	*		
4		□	*	*	□	
5			*	*	*	
6			□	*	*	□
7				*	*	*
8				□	*	*
9					*	*

⊗ werden entfernt, dabei entsteht fill-in (□).



Zyklische Reduktion

- Algorithmus der Zyklischen Reduktion

// Elimination aller ungeraden Unbekannten in geraden Zeilen:

for ($k = 1; k < N; k += 2$)

{ // Zeile k modifiziert Zeile $k - 1$

$$l = a_{k-1,k} / a_{k,k};$$

$$a_{k-1,k-1} = a_{k-1,k-1} - l \cdot a_{k,k-1};$$

$$a_{k-1,k+1} = -l \cdot a_{k,k+1}; \quad // \text{fill-in}$$

$$b_{k-1} = b_{k-1} - l \cdot b_k;$$

} // $\frac{N}{2} 6t_f$

for ($k = 2; k < N; k += 2$);

{ // Zeile $k - 1$ modifiziert Zeile k

$$l = a_{k,k-1} / a_{k-1,k-1};$$

$$a_{k,k-2} = l \cdot a_{k-1,k-2}; \quad // \text{fill-in}$$

$$a_{k,k} = l \cdot a_{k-1,k};$$

} // $\frac{N}{2} 3t_f$

- Alle Durchläufe beider Schleifen können parallel bearbeitet werden (nehmen wir eine Maschine mit gemeinsamem Speicher an)!



Zyklische Reduktion

- Resultat dieser Elimination ist

	0	1	2	3	4	5	6	7	8	9
0	*		*							
1	*	*	*							
2	*		*		*					
3			*	*	*					
4			*		*		*			
5					*	*	*			
6					*		*		*	
7							*	*	*	
8							*		*	
9								*	*	*

bzw. nach Umordnen

	1	3	5	7	9	0	2	4	6
1	*					*	*		
3		*					*	*	
5			*					*	*
7				*					*
9					*				
0						*	*		
2						*	*	*	
4							*	*	*
6								*	*
8									*

- Sind die x_{2k} , $k = 0, \dots, M - 1$, berechnet, so können die ungeraden Unbekannten mit

for ($k = 1$; $k < N - 1$; $k += 2$)

$$x_k = (b_k - a_{k,k-1} \cdot x_{k-1} - a_{k,k+1} \cdot x_{k+1}) / a_{k,k};$$

// $\frac{N}{2} 5t_f$

$$x_{N-1} = (b_{N-1} - a_{N-1,N-2} \cdot x_{N-2}) / a_{N-1,N-1};$$

parallel berechnet werden.



Zyklische Reduktion

- Der *sequentielle* Aufwand für die zyklische Reduktion ist somit

$$\begin{aligned}T_S(N) &= (6 + 3 + 5)t_f \left(\frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \dots + 1 \right) \\ &= 14Nt_f\end{aligned}$$

- Dies ist fast doppelt so viel wie der optimale sequentielle Algorithmus benötigt. Dafür kann die zyklische Reduktion parallelisiert werden. Die maximal erreichbare Effizienz ist jedoch

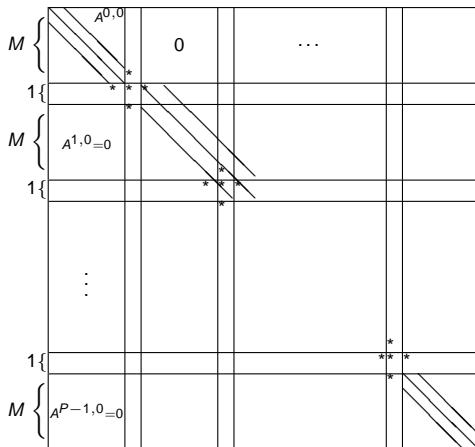
$$E_{\max} = \frac{8}{14} \approx 0.53,$$

wobei angenommen wurde, dass alle Operationen optimal parallel ausgeführt werden und Kommunikation umsonst ist (Rückwärtseinsetzten beschäftigt nur $\frac{N}{2}$ Prozessoren!). Nicht berücksichtigt wurde auch, dass die zyklische Reduktion mehr Indexrechnung benötigt!



Gebietszerlegungsmethoden

- Ein anderer Ansatz lässt sich im Prinzip auf allgemeinere Problemstellungen erweitern lässt: Gebietszerlegungsverfahren
- Es sei P die Zahl der Prozessoren und $N = MP + P - 1$ für ein $M \in \mathbf{N}$. Wir unterteilen dann die $N \times N$ -Matrix A in P Blöcke à M Zeilen mit je einer Zeile zwischen den Blöcken:



Gebietszerlegung

- Die Unbekannten zwischen den Blöcken bilden das *Interface*. Jeder Block ist mit höchstens zwei Interface-Unbekannten gekoppelt.
- Nun sortieren wir Zeilen und Spalten der Matrix so um, dass die Interfaceunbekannten am Ende stehen. Dies ergibt folgende Gestalt:

$$\begin{array}{|ccc|c}
 A^{0,0} & & & A^{0,l} \\
 & A^{1,1} & & A^{1,l} \\
 & & \ddots & \vdots \\
 & & & A^{P-1,P-1} \\
 \hline
 A^{l,0} & A^{l,1} & \dots & A^{l,P-1} \\
 \hline
 & & & A^{l,l}
 \end{array} ,$$

wobei $A^{p,p}$ die $M \times M$ -Tridiagonalmatrix und $A^{l,l}$ eine $P-1 \times P-1$ -Diagonalmatrix ist. Die $A^{p,l}$ haben die allgemeine Form

$$A^{p,l} = \left(\begin{array}{c|c|c} \dots & * & \dots \\ \hline & * & \\ \hline \dots & * & \dots \end{array} \right) .$$



Gebietszerlegung

- Idee: Eliminiere Blöcke $A^{l,*}$ in der Blockdarstellung. Dadurch wird $A^{l,l}$ modifiziert, genauer entsteht folgende Blockdarstellung:

$$\left(\begin{array}{cccc|c} A^{0,0} & & & & A^{0,l} \\ & A^{1,1} & & & A^{1,l} \\ & & \ddots & & \vdots \\ & & & A^{P-1,P-1} & A^{P-1,l} \\ \hline 0 & 0 & \dots & 0 & S \end{array} \right),$$

$$\text{mit } S = A^{l,l} - \sum_{p=0}^{P-1} A^{l,p} (A^{p,p})^{-1} A^{p,l}.$$

- S wird allgemein als „Schurkomplement“ bezeichnet. Alle Eliminationen in $\sum_{p=0}^{P-1}$ können *parallel* durchgeführt werden.
- Nach Lösen eines Systems $Sy = d$ für die Interfaceunbekannten können die inneren Unbekannten wieder parallel berechnet werden.
- S hat Dimension $P - 1 \times P - 1$ und ist selbst dünn besetzt, wie wir gleich sehen werden.



Ausführen des Plans

1. Transformiere $A^{p,p}$ auf Diagonalgestalt.

($a_{i,j}$ bezeichnet $(A^{p,p})_{i,j}$, falls nicht anders angegeben):

$\forall p$ parallel

for ($k = 0$; $k < M - 1$; $k++$) // untere Diagonale

{
 $l = a_{k+1,k} / a_{k,k}$;
 $a_{k+1,k+1} = a_{k+1,k+1} - l \cdot a_{k,k+1}$;
 if ($p > 0$) $a_{k+1,p-1}^{p,l} = a_{k+1,p-1}^{p,l} - l \cdot a_{k,p-1}$;
 // fill-in linker Rand

$b_{k+1}^p = b_{k+1}^p - l \cdot b_k^p$;
} // $(M - 1)7t_f$

for ($k = M - 1$; $k > 0$; $k--$) // obere Diagonale

{
 $l = a_{k-1,k} / a_{k,k}$;
 $b_{k-1}^p = b_{k-1}^p - l \cdot b_k^p$;
 if ($p > 0$) $a_{k-1,p-1}^{p,l} = a_{k-1,p-1}^{p,l} - l \cdot a_{k,p-1}^{p,l}$;
 // linker Rand
 if ($p < P - 1$) $a_{k-1,p}^{p,l} = a_{k-1,p}^{p,l} - l \cdot a_{k,p}^{p,l}$;
 // rechter Rand, fill-in

} // $(M - 1)7t_f$



Ausführen des Plans

2. Eliminiere in $A^{l,*}$.

$\forall p$ parallel:

if ($p > 0$)

{

$$l = a_{p-1,0}^{l,p} / a_{0,0}^{p,p};$$

$$a_{p-1,p-1}^{l,l} = a_{p-1,p-1}^{l,l} - l \cdot a_{0,p-1}^{p,l};$$

$$\text{if } (p < P - 1) \ a_{p-1,p}^{l,l} = a_{p-1,p}^{l,l} - l \cdot a_{0,p}^{p,l};$$

$$b_{p-1}^l = b_{p-1}^l - l \cdot b_0^p;$$

}

if ($p < P - 1$)

{

$$l = a_{p,M-1}^{l,p} / a_{M-1,M-1}^{p,p};$$

$$\text{if } (p > 0) \ a_{p,p-1}^{l,l} = a_{p,p-1}^{l,l} - l \cdot a_{M-1,p-1}^{p,l};$$

$$a_{p,p}^{l,l} = a_{p,p}^{l,l} - l \cdot a_{M-1,p}^{p,l};$$

$$b_p^l = b_p^l - l \cdot b_{M-1}^p;$$

}

// linker Rand $P - 1$ im Interface

// Diagonale in S

// obere Diag. in S , fill-in

// rechter Rand

// fill-in untere Diag von S



Ausführen des Plans

3. Löse Schurkomplement. S ist *tridiagonal* mit Dimension $P - 1 \times P - 1$. Nehme an, dass $M \gg P$ und löse sequentiell. $\rightarrow 8Pt_f$ Aufwand.
4. Berechne innere Unbekannte Hier ist nur eine Diagonalmatrix je Prozessor zu lösen.

$\forall p$ parallel:

for ($k = 0$; $k < M - 1$; $k++$)

$$x_k^p = (b_k^p - a_{k,p-1}^{p,l} \cdot x_{p-1}^l - a_{k,p}^{p,l} \cdot x_p^l) / a_{k,k}^{p,p};$$

// $M5t_f$



- Gesamtaufwand parallel:

$$\begin{aligned}T_P(N, P) &= 14Mt_f + O(1)t_f + 8Pt_f + 5Mt_f = \\ &= 19Mt_f + 8Pt_f\end{aligned}$$

(ohne Kommunikation!)

$$\begin{aligned}E_{\max} &= \frac{8(MP + P - 1)t_f}{(19Mt_f + 8Pt_f)P} \approx \\ &\approx \frac{1}{\frac{19}{8} + \frac{P}{M}} \leq \frac{8}{19} = 0.42 \\ &\text{für } P \ll M\end{aligned}$$

- Der Algorithmus benötigt zusätzlichen Speicher für das fill-in. Zyklische Reduktion arbeitet mit Überschreiben der alten Einträge.



LU-Zerlegung dünnbesetzter Matrizen

Was ist eine dünnbesetzte Matrix

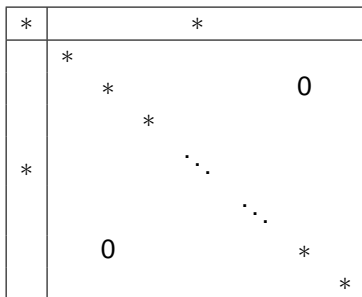
- Im allgemeinen spricht man von einer dünnbesetzten Matrix, wenn sie in (fast) jeder Zeile nur eine konstante Anzahl von Nichtnullelementen besitzt.
- Ist $A \in \mathcal{R}^{N \times N}$, so hat A dann nur $O(N)$ statt N^2 Einträge.
- Für genügend großes N ist es dann bezüglich Rechenzeit und Speicher günstig, diese große Anzahl von Nullen nicht zu bearbeiten bzw. zu speichern.



LU-Zerlegung dünnbesetzter Matrizen

Fill-in

- Bei der LU -Zerlegung können im Verlauf des Eliminationsprozesses Elemente, die anfänglich Null waren, zu Nichtnullelementen werden.
- Man spricht dann von „*Fill-in*“.
- Dies hängt stark von der Struktur der Matrix ab. Als extremes Beispiel betrachte die „Pfeilmatrix“



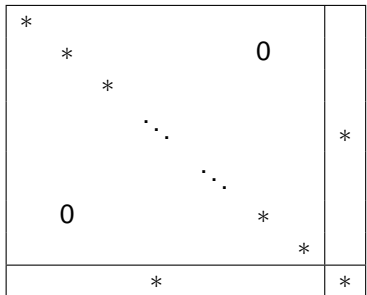
- Bei der Elimination in der natürlichen Reihenfolge (ohne Pivotisierung) läuft die gesamte Matrix voll.



LU-Zerlegung dünnbesetzter Matrizen

Umordnung der Matrix

- Bringt man die Matrix durch Zeilen- und Spaltenvertauschung auf die Form



- es entsteht offensichtlich kein Fill-in.
- Ein wichtiger Punkt bei der LU -Zerlegung dünnbesetzter Matrizen ist das Finden einer Anordnung der Matrix, so dass das Fill-in minimiert wird.
- Umordnung ist eng gekoppelt mit der Pivotisierung



Pivotisierung

- Ist die Matrix A symmetrisch positiv definit (SPD), so ist die LU -Faktorisierung immer numerisch stabil, und es ist *keine* Pivotisierung notwendig.
- Die Matrix kann also vorab so umgeordnet werden, dass das Fill-in klein wird.
- Bei einer allgemeinen, invertierbaren Matrix wird man pivotisieren müssen.
- Dann muss dynamisch während der Elimination ein Kompromiss zwischen numerischer Stabilität und Fill-in gefunden werden.
- Deshalb beschränken sich fast alle Codes auf den symmetrisch positiven Fall und bestimmen vorab eine Eliminationsreihenfolge, die das Fill-in minimiert.
- Eine exakte Lösung dieses Minimierungsproblems ist \mathcal{NP} -Vollständig.
- Man verwendet daher heuristische Verfahren.



Graph einer Matrix

Matrixgraph

- Im symmetrisch positiven Fall lässt sich das Fill-in rein anhand der Null-Struktur der Matrix untersuchen.
- Zu einem beliebigen, nun nicht notwendigerweise symmetrischen $A \in \mathcal{R}^{N \times N}$ definieren wir einen ungerichteten Graphen $G(A) = (V_A, E_A)$ mit

$$\begin{aligned} V_A &= \{0, \dots, N-1\} \\ (i, j) \in E_A &\iff a_{ij} \neq 0 \vee a_{ji} \neq 0. \end{aligned}$$

- Dieser Graph beschreibt die direkten Abhängigkeiten der Unbekannten untereinander.

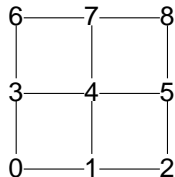


Graph einer Matrix

Beispiel:

	0	1	2	3	4	5	6	7	8
0	*	*		*					
1	*	*	*		*				
2		*	*			*			
3	*			*	*		*		
4		*		*	*	*		*	
5			*		*	*			*
6				*			*	*	
7					*		*	*	*
8						*		*	*

A



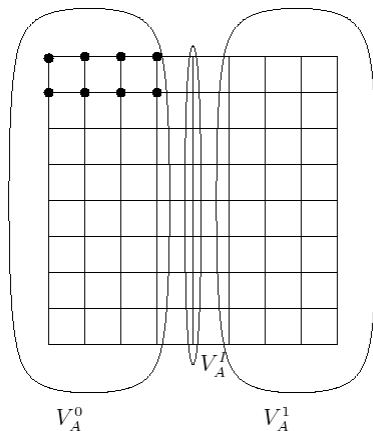
$G(A)$



Matrix-Anordnungsstrategien

Nested Dissection

- Ein wichtiges Verfahren zur Anordnung von SPD-Matrizen zum Zwecke der Fill-in-Minimierung ist die „*nested dissection*“.
- Beispiel:
Der Graph $G(A)$ der Matrix A sei ein quadratisches Gitter



Matrix-Anordnungsstrategien

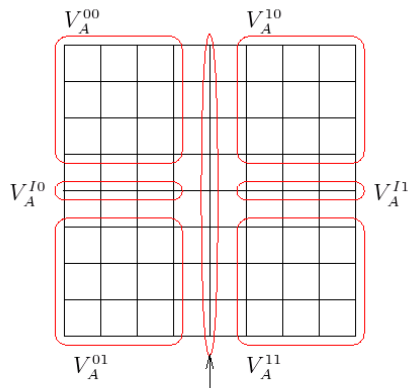
Nun teilen wir die Knotenmenge V_A in drei Teile: V_A^0 , V_A^1 und V_A^l , so dass

- V_A^0 und V_A^1 sind (möglichst) groß,
- V_A^l ist ein Separator, d.h. bei Herausnahme von V_A^l aus dem Graphen zerfällt dieser in zwei Teile. Somit gibt es *kein* $(i, j) \in E_A$, so dass $i \in V_A^0$ und $j \in V_A^1$.
- V_A^l ist möglichst klein.
- Die Abbildung zeigt eine Möglichkeit für eine solche Aufteilung.



Matrix-Anordnungsstrategien

- Nun ordnet man die Zeilen und Spalten so um, dass zuerst die Indizes V_A^0 , dann V_A^1 und zum Schluss V_A^l kommen.
- Dann wendet man das Verfahren *rekursiv* auf die Teilgraphen mit den Knotenmengen V_A^0 und V_A^1 an.
- Das Verfahren stoppt, wenn die Graphen eine vorgegebene Größe erreicht haben.
- Beispielgraph nach zwei Schritten:



Matrix-Anordnungsstrategien

	V_A^{00}	V_A^{01}	V_A^{10}	V_A^{10}	V_A^{11}	V_A^{11}	V_A^j
V_A^{00}	*	0	*				*
V_A^{01}	0	*	*				*
V_A^{10}	*	*	*				*
V_A^{10}				*		*	*
V_A^{11}					*	*	*
V_A^{11}				*	*	*	*
V_A^j	*	*	*	*	*	*	*

A, umgeordnet

Komplexität der Nested Dissection

- Für obiges Beispiel führt die nested dissection Nummerierung auf eine Komplexität von $O(N^{3/2})$ für die LU -Zerlegung.
- Zum Vergleich benötigt man bei lexikographischer Nummerierung (Bandmatrix) $O(N^2)$ Operationen.



Datenstrukturen für dünnbesetzte Matrizen

- Es gibt eine ganze Reihe von Datenstrukturen zur Speicherung von dünnbesetzten Matrizen.
- Ziel ist eine effiziente Implementierung von Algorithmen
- Somit ist auf Datenlokalität und möglichst wenig Overhead durch zusätzliche Indexrechnung zu achten.
- Eine oft verwendete Datenstruktur ist „*compressed row storage*“
- Hat die $N \times N$ -Matrix insgesamt M Nichtnullelemente, so speichert man die Matrixelemente Zeilenweise in einem eindimensionalen Feld ab:

double $a[M]$;

- Die Verwaltung der Indexinformation geschieht mittels dreier Felder **int** $s[N]$, $r[N]$, $j[M]$;
- Deren Belegung zeigt die Realisierung des Matrix-Vektor-Produktes $y = Ax$:

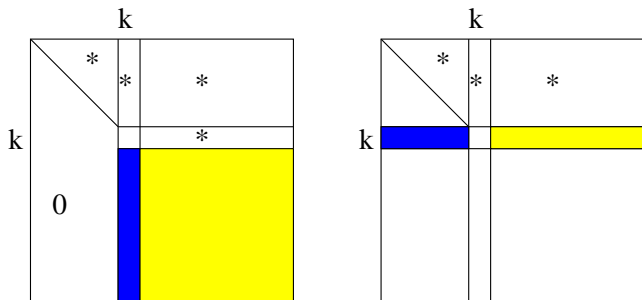
```
for (i = 0; i < N; i++) {  
    y[i] = 0;  
    for (k = r[i]; k < r[i] + s[i]; k++)  
        y[i] += a[k] · x[j[k]];  
}
```

- r gibt den Zeilenanfang, s die Zeilenlänge, und j den Spaltenindex.



Eliminationsformen

- Bei der LU -Zerlegung für vollbesetzte Matrizen haben wir die sogenannte kij -Form der LU -Zerlegung verwendet.



- Dabei werden in jedem Schritt k alle a_{ik} für $i > k$ eliminiert, was eine Modifikation aller a_{ij} mit $i, j > k$ erfordert.
- Diese Situation ist links dargestellt.
- Bei der kji -Variante eliminiert man im Schritt k alle a_{kj} mit $j < k$.
- Dabei werden die a_{ki} mit $i \geq k$ modifiziert. Beachte, dass die a_{kj} von links nach rechts eliminiert werden müssen!
- Bei der folgenden sequentiellen LU -Zerlegung für dünnbesetzte Matrizen werden wir von dieser kji -Variante ausgehen.



Sequentieller Algorithmus

- Im folgenden setzen wir voraus:
 - ▶ Die Matrix A kann in der gegebenen Anordnung ohne Pivottisierung faktorisiert werden. Die Anordnung wurde in geeigneter Weise gewählt, um das Fill-in zu minimieren.
 - ▶ Die Datenstruktur speichert alle Elemente a_{ij} mit $(i, j) \in G(A)$. Wegen der Definition von $G(A)$ gilt:

$$(i, j) \notin G(A) \Rightarrow a_{ij} = 0 \wedge a_{ji} = 0.$$

Ist $a_{ij} \neq 0$, so wird in jedem Fall auch a_{ji} gespeichert, *auch wenn dies Null ist*. Die Matrix muss nicht symmetrisch sein.

- Die Erweiterung der Struktur von A geschieht rein aufgrund der Information in $G(A)$. So wird, falls $(k, j) \in G(A)$, auch ein a_{kj} formal eliminiert. Dies kann möglicherweise ein fill-in a_{ki} erzeugen, obwohl $a_{ki} = 0$ gilt.
- Der nun folgende Algorithmus verwendet die Mengen $S_k \subset \{0, \dots, k-1\}$, welche im Schritt k genau die Spaltenindizes enthalten, die eliminiert werden müssen.



Sequentieller Algorithmus

```
for (k = 0; k < N; k++) Sk = ∅;
for (k = 0; k < N; k++)
{
  // 1. erweitere Matrixgraph
  for (j ∈ Sk)
  {
    G(A) = G(A) ∪ {(k, j)};
    for (i = k; i < N; i++)
      if ((j, i) ∈ G(A))
        G(A) = G(A) ∪ {(k, i)};
  }

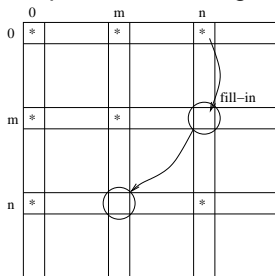
  // 2. Eliminiere
  for (j ∈ Sk)
  {
    ak,j = ak,j/aj,j;
    for (i = j + 1; i < N; i++)
      ak,i = ak,i - ak,j · aj,i;
  }
  // eliminiere ak,j
  // L-Faktor

  // 3. update Si für i > k, geht wegen Symmetrie von EA
  for (i = k + 1; i < N; i++)
    if ((k, i) ∈ G(A))
      Si = Si ∪ {k};
  // ⇒ (i, k) ∈ G(A)
}
```



Sequentieller Algorithmus

- Betrachten wir in einem Beispiel, wie die S_k gebildet werden.



- Zu Beginn enthalte $G(A)$ die Elemente

$$G(A) = \{(0, 0), (m, m), (n, n), (0, n), (n, 0), (0, m), (m, 0)\}$$

- Für $k = 0$ wird in Schritt 3 $S_m = \{0\}$ und $S_n = \{0\}$ gesetzt.
- Nun wird als nächstes bei $k = m$ das $a_{m,0}$ eliminiert.
- Dies erzeugt das Fill-in (m, n) , was wiederum im Schritt 3 bei $k = m$ die Anweisung $S_n = s_n \cup \{m\}$ zur Folge hat.
- Somit gilt am Anfang des Schleifendurchlaufes $k = n$ korrekt $S_n = \{0, m\}$ und in Schritt 1 wird korrekt das Fill-in $a_{n,m}$ erzeugt, bevor die Elimination von $a_{n,0}$ durchgeführt wird. Dies gelingt wegen der Symmetrie von E_A .

Parallelisierung

LU -Zerlegung dünnbesetzter Matrizen besitzt folgende Möglichkeiten zu einer Parallelisierung:

- *grobe Granularität*: In allen 2^d Teilmengen von Indizes, die man durch nested dissection der Tiefe d gewinnt, kann man parallel mit der Elimination beginnen. Erst für die Indizes, die den Separatoren entsprechen, ist Kommunikation erforderlich.
- *mittlere Granularität*: Einzelne Zeilen können parallel bearbeitet werden, sobald die entsprechende Pivotzeile lokal verfügbar ist. Dies entspricht dem Parallelismus bei der dichten LU -Zerlegung.
- *feine Granularität*: Modifikationen einer einzelnen Zeile können parallel bearbeitet werden, sobald Pivotzeile und Multiplikator verfügbar sind. Dies nutzt man bei der zweidimensionalen Datenverteilung im dicht besetzten Fall.



Parallelisierung: Fall $N = P$

Programm (*LU*-Zerlegung für dünnbesetzte Matrizen und $N = P$)

parallel *sparse-lu-1*

```
{  
  const int  $N = \dots$ ;  
  process  $\Pi$ [int  $k \in \{0, \dots, N - 1\}$ ]  
  {  
    // (nur Pseudocode!)  
     $S = \emptyset$ ; // 1. Belege S  
    for ( $j = 0; j < k; j++$ ) // der Anfang  
      if ( $((k, j) \in G(A))$ )  $S = S \cup \{j\}$ ;  
    for ( $j = 0; j < k; j++$ ) // 2. Zeile k bearbeiten  
      if ( $(j \in S_k)$ )  
      {  
        recv( $\Pi_j, r$ ); // warte, bis  $\Pi_j$  die Zeile j schickt  
        // erweitere Muster  
        for ( $i = j + 1; i < N; i++$ )  
        {  
          if ( $(i < k \wedge (j, i) \in G(A))$ ) // Info ist in r  
             $S = S \cup \{i\}$ ; // Prozessor i wird Zeile i schicken  
          if ( $((j, i) \in G(A)) \cup \{(k, i)\}$ ) // Info ist in r  
             $G(A) = G(A) \cup \{(k, i)\}$ ;  
        }  
        // eliminiere  $a_{k,j} = a_{k,j} / a_{j,j}$ ; // Info ist in r  
        for ( $i = j + 1; i < N; i++$ ) // Info ist in r  
           $a_{k,i} = a_{k,i} - a_{k,j} \cdot a_{j,i}$ ;  
      }  
    for ( $i = k + 1; i < N; i++$ ) // 3. wegschicken  
      if ( $((k, i) \in G(A))$ ) // lokale Info!  
        send Zeile k an  $\Pi_i$ ; // k weiss, dass i die Zeile k braucht.  
  }  
}
```


Parallelisierung für $N \gg P$

Der Fall $N \gg P$

- Jeder Prozessor hat nun einen ganzen Block von Zeilen. Drei Dinge sind zu tun:
- *Empfange Pivotzeilen* von anderen Prozessoren und speichere diese in der Menge R .
- *Sende fertige Zeilen* aus einem Sendepuffer S zu den Zielprozessoren.
- *Lokale Elimination*
 - ▶ Wähle eine Zeile j aus dem Empfangspuffer R .
 - ▶ Eliminiere damit lokal alle möglichen $a_{k,j}$.
 - ▶ Wenn eine Zeile fertig wird, stecke sie in den Sendepuffer (es kann mehrere Zielprozessoren geben).

