Freie Universität Berlin

Iterative Verfahren zum Lösen von linearen Gleichungssystemen

Referenten:
Christoph Graebnitz
Benjamin Zengin
Dozentin: Prof. Dr. M. Esponda
Modul: Proseminar Parallel Programming
Datum: 06.02.2013

Freie Universität Berlin

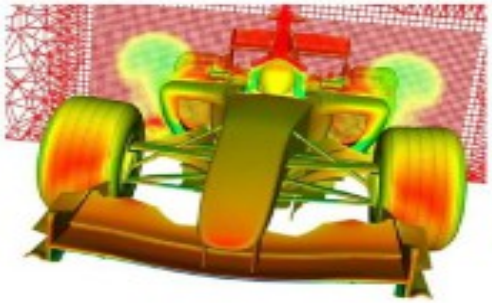
Gliederung

1. Iterative Algorithmen und Parallelisierung
   i. Gauß - Seidel – Verfahren
   ii. Anwendung auf dünnbesetzte Matrizen
   iii. SOR – Verfahren
2. Cholesky - Faktorisierung
   i. Definition
   ii. Strukturierung für dünnbesetzte Matrizen
   iii. Algorithmen
   iv. Parallelisierung

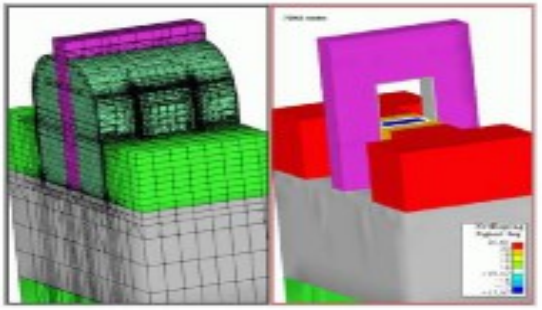
Freie Universität Berlin

Anwendungsbereiche

Strömungsdynamik

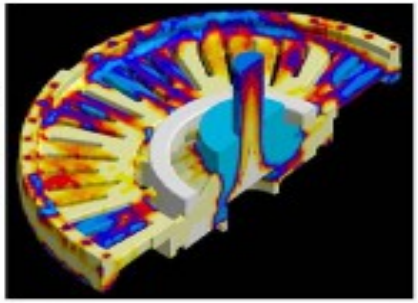


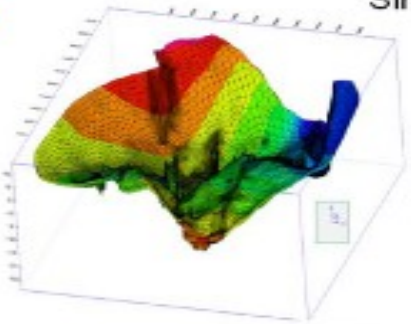
Öl&Gas-Reservoir-Simulation



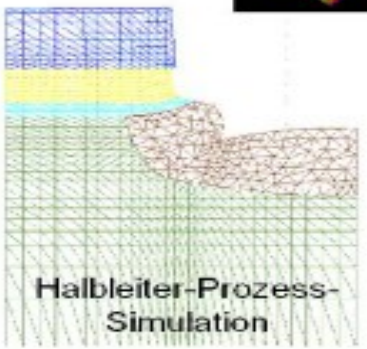
Halbleiter-Bauelemente-Simulation

Struktur-Mechanik / Gieß-simulation



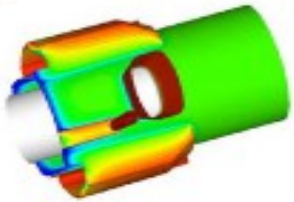


Grundwasser-Simulation, Geophysik



Halbleiter-Prozess-Simulation

Elektrochemie



Schaltungs-simulation, Elektromagnetisch e Verträglichkeit

2. Gauß-Seidel-Verfahren
Motivation

$$A = \begin{pmatrix} 4 & 1 & 2 \\ 1 & 3 & 4 \\ 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 13 \\ 9 \end{pmatrix}$$

Konvergenzbeispiel:

	Jacobi-Verfahren			Gauß-Seidel-Verf.			Jacobi-Relaxationsv.		
m	x_1^m	x_2^m	x_3^m	x_1^m	x_2^m	x_3^m	x_1^m	x_2^m	x_3^m
0	10	10	10	10	10	10	10	10	10
5	-3.4722	-3.5463	-2.6528	0.9785	1.8258	3.0979	0.0876	0.8770	1.5945
10	4.0059	5.7443	6.6438	1.0000	1.9991	3.0005	1.1603	2.2050	3.1545
15	-0.9840	-0.4692	0.5793	1.0000	2.0000	3.0000	0.9800	1.9760	2.9697
20	2.3131	3.6345	4.6005	1.0000	2.0000	3.0000	1.0035	2.0044	3.0033
50	1.1101	2.1371	3.1342	1.0000	2.0000	3.0000	1.0000	2.0000	3.0000
100	1.0018	2.0022	3.0022	1.0000	2.0000	3.0000	1.0000	2.0000	3.0000
150	1.0000	2.0000	3.0000	1.0000	2.0000	3.0000	1.0000	2.0000	3.0000

2. Gauß-Seidel-Verfahren Konvergenz

Freie Universität Berlin

Starkes Zeilensummenkriterium garantiert Konvergenz

$$\sum_{j=1\ j\neq i}^{n} |a_{ij}| < |a_{ii}| \qquad i = 1, \ldots, n$$

Garantierte Konvergenz außerdem für
Symmetrische positiv definite Matrizen

Freie Universität Berlin

2. Gauß-Seidel-Verfahren

Aufteilung der Matrix A in:

Sei: $A \in \mathbb{R}^{n \times n}$ $\quad A = D - L - R$

D ist die Diagonalmatrix
-L ist die Untere Dreiecksmatrix
-R ist die Obere Dreiecksmatrix

$$D = \begin{pmatrix} * & 0 & 0 & 0 \\ 0 & * & 0 & 0 \\ 0 & 0 & * & 0 \\ 0 & 0 & 0 & * \end{pmatrix} \qquad -L = \begin{pmatrix} 0 & 0 & 0 & 0 \\ * & 0 & 0 & 0 \\ * & * & 0 & 0 \\ * & * & * & 0 \end{pmatrix} \qquad -R = \begin{pmatrix} 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

* steht für beliebige Werte

2. Gauß-Seidel-Verfahren

Daraus ergibt sich die Matrixvorschrift pro Iterationsschritt

$$(D-L)\,x^{(k+1)} = Rx^{(k)} + b$$

Umgeformt:

$$x^{(k+1)} = -(D+L)^{-1}\,Rx^{(k)} + (D+L)^{-1}\,b$$

Die Berechnung pro Iterationsschritt

2. Gauß-Seidel-Verfahren

Komponentenvorschrift:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{(k)} \right) \qquad i = 1,2,\ldots,n$$

$x_i^{(k+1)}$ Enthält Ergebnis der Aktuellen Iteration

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{ii} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \qquad x^{(k)} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \qquad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

2. Gauß-Seidel-Verfahren Sequentielle Implementierung

Freie Universität Berlin

repeat:

$$x^{(k)} := x^{(k+1)}$$

for i = 1 to n:

$$x_i^{(k+1)} := \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

until $\| x^{(k+1)} - x^{(k)} \| \leq \varepsilon \| x^{(k+1)} \|$

$x_i^{(k+1)}$ Enthält Ergebnis des Aktuellen Iteration

$x^{(k)}$ Wird für Kovergenztest benötigt

ε Toleranzgrenze

$\| . \|$ Vektornorm $\| x \|_\infty = max | x |_i$ oder $\| x \|_2 = \left(\sum_{i=1}^{n} | x_i |^2 \right)^{\frac{1}{2}}$,

$i = 1, 2, \ldots, n$

2. Gauß-Seidel-Verfahren
Parallele Implementierung

Freie Universität Berlin

Komponentenvorschrift:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

Parallelität nur innerhalb eines Iterationsschrittes möglich

$x_i^{(k+1)}$ Zur Berechnung werden alle $x_i^{(k+1)}$ der vorherigen Iterationen benötigt

2. Gauß-Seidel-Verfahren Parallele Implementierung

Freie Universität Berlin

Verwenden spaltenorientierte Blockverteilung
n ist ein Vielfaches von p Anzahl Prozessoren

$$A = \begin{pmatrix} P_1 & P_1 & & P_n & P_n \\ a_{11} & a_{12} & \cdots & a_{1n-1} & a_{1n} \\ a_{21} & a_{ii} & \cdots & a_{2n-1} & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{(n-1)1} & a_{(n-1)2} & \cdots & a_{n-1n-1} & a_{n-1n} \\ a_{n1} & a_{n2} & \cdots & a_{nn-1} & a_{nn} \end{pmatrix}$$

Zeilenweise parallel rechnen!

2. Gauß-Seidel-Verfahren
Parallele Implementierung

Freie Universität Berlin

Teilsumme der Prozessoren P_q mit $1 \le q \le p$

$$s_{qi} = \sum_{\substack{j=(q-1)\cdot n/p+1 \\ j<i}}^{q\cdot n/p} a_{ij}\, x_j^{(k+1)} + \sum_{\substack{j=(q-1)\cdot n/p+1 \\ j>i}}^{q\cdot n/p} a_{ij}\, x_j^{(k)}$$

$$A = \begin{array}{c} \begin{matrix} P_1 & P_1 & & P_n & P_n \end{matrix} \\ \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n-1} & a_{1n} \\ a_{21} & a_{ii} & \cdots & a_{2n-1} & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{(n-1)1} & a_{(n-1)2} & \cdots & a_{n-1n-1} & a_{n-1n} \\ a_{n1} & a_{n2} & \cdots & a_{nn-1} & a_{nn} \end{pmatrix} \end{array}$$

2. Gauß-Seidel-Verfahren
Parallele Implementierung

Freie Universität Berlin

Programmfragment:

```
n_local = n/p;
do {
   delta_x = 0.0;
   for (i = 0; i < n; i++) {
      s_k = 0.0;
      for (j = 0; j < n_local; j++)
         if (j + me * n_local != i)
           s_k = s_k + local_A[i][j] * x[j];
      root = i/n_local;
      i_local = i % n_local;
      MPI_Reduce(&s_k, &x[i_local], 1, MPI_FLOAT, MPI_SUM, root,
                 MPI_COMM_WORLD);
      if (me == root) {
        x_new = (b[i_local] - x[i_local]) / local_A[i][i_local];
        delta_x = max(delta_x, abs(x[i_local] - x_new));
        x[i_local] = x_new;
      }
   }
   MPI_Allreduce(&delta_x, &global_delta, 1, MPI_FLOAT,
                 MPI_MAX, MPI_COMM_WORLD);
} while(global_delta > tol);
```

Teilsumme S_{qi}

2. Gauß-Seidel-Verfahren
Parallele Implementierung

Freie Universität Berlin

Programmfragment:

Teilsummen werden Zusammengefasst

```
n_local = n/p;
do {
   delta_x = 0.0;
   for (i = 0; i < n; i++) {
      s_k = 0.0;
      for (j = 0; j < n_local; j++)
         if (j + me * n_local != i)
           s_k = s_k + local_A[i][j] * x[j];
      root = i/n_local;
      i_local = i % n_local;
      MPI_Reduce(&s_k, &x[i_local], 1, MPI_FLOAT, MPI_SUM, root,
                 MPI_COMM_WORLD);
      if (me == root) {
        x_new = (b[i_local] - x[i_local]) / local_A[i][i_local];
        delta_x = max(delta_x, abs(x[i_local] - x_new));
        x[i_local] = x_new;
      }
   }
   MPI_Allreduce(&delta_x, &global_delta, 1, MPI_FLOAT,
                 MPI_MAX, MPI_COMM_WORLD);
} while(global_delta > tol);
```

2. Gauß-Seidel-Verfahren
Parallele Implementierung

Freie Universität Berlin

Programmfragment:

$x_i^{(k+1)}$ wird berechnet

```
n_local = n/p;
do {
   delta_x = 0.0;
   for (i = 0; i < n; i++) {
      s_k = 0.0;
      for (j = 0; j < n_local; j++)
         if (j + me * n_local != i)
           s_k = s_k + local_A[i][j] * x[j];
      root = i/n_local;
      i_local = i % n_local;
      MPI_Reduce(&s_k, &x[i_local], 1, MPI_FLOAT, MPI_SUM, root,
                 MPI_COMM_WORLD);
      if (me == root) {
        x_new = (b[i_local] - x[i_local]) / local_A[i][i_local];
        delta_x = max(delta_x, abs(x[i_local] - x_new));
        x[i_local] = x_new;
      }
   }
   MPI_Allreduce(&delta_x, &global_delta, 1, MPI_FLOAT,
                 MPI_MAX, MPI_COMM_WORLD);
} while(global_delta > tol);
```

2. Gauß-Seidel-Verfahren
Parallele Implementierung

Freie Universität Berlin

Programmfragment:

```
n_local = n/p;
do {
   delta_x = 0.0;
   for (i = 0; i < n; i++) {
      s_k = 0.0;
      for (j = 0; j < n_local; j++)
         if (j + me * n_local != i)
           s_k = s_k + local_A[i][j] * x[j];
      root = i/n_local;
      i_local = i % n_local;
      MPI_Reduce(&s_k, &x[i_local], 1, MPI_FLOAT, MPI_SUM, root,
                 MPI_COMM_WORLD);
      if (me == root) {
        x_new = (b[i_local] - x[i_local]) / local_A[i][i_local];
        delta_x = max(delta_x, abs(x[i_local] - x_new));
        x[i_local] = x_new;
      }
   }
   MPI_Allreduce(&delta_x, &global_delta, 1, MPI_FLOAT,
                 MPI_MAX, MPI_COMM_WORLD);
} while(global_delta > tol);
```

delta_x wird für Kovergenztest übermittelt

Freie Universität Berlin

2. Gauß-Seidel-Verfahren
Für dünnbesetzte Matrizen

Freie Universität Berlin

2. Gauß-Seidel-Verfahren
Für dünnbesetzte Matrizen

Einschränkung der Parallelität durch Datenabhängigkeiten

Nur für große Matrizen annehmbare Beschleunigung

Beschleunigung der Konvergenz bei dünnbesetzten Matrizen

Quadratische Matrix mit $O(n \cdot \log(n))$ Einträgen ist dünnbesetzt

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Wodurch entsteht die Beschleunigung?

Falls $A=(a_{ij})\ i,\ j=1,2,\ldots,n$ dünnbesetzt

Wenn $a_{ij}=0$

$$x_i^{(k+1)}=\frac{1}{a_{ii}}\left(b_i-\sum_{j=1}^{i-1}0\,\underbrace{x_j^{(k+1)}}-\sum_{j=i+1}^{n}a_{ij}\,x_j^{(k)}\right)$$

Nicht gebraucht

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Dünnbesetzte Matrix mit Bandstruktur

$$\begin{pmatrix} a_{11} & \cdots & a_{1(q+1)} & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & \ddots & & \ddots & \ddots & & & \vdots \\ a_{(p+1)1} & & \ddots & & \ddots & \ddots & & \vdots \\ 0 & \ddots & & \ddots & & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & & \ddots & & \ddots & 0 \\ \vdots & & \ddots & \ddots & & \ddots & & a_{(n-q)n} \\ \vdots & & & \ddots & \ddots & & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & a_{n(n-p)} & \cdots & a_{nn} \end{pmatrix}$$

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Spezielle Bandmatrix

$a_{ij} \neq 0$ nur für $j = i - \sqrt{n}, i-1, i+1, i, i+\sqrt{n}$

Bsp. einer 9×9

$$\begin{pmatrix}
a_{11} & a_{1(1+1)} & 0 & a_{1(1+\sqrt{9})} & 0 & 0 & 0 & 0 & 0 \\
a_{2(2-1)} & a_{22} & a_{2(2+1)} & 0 & a_{2(2+\sqrt{9})} & 0 & 0 & 0 & 0 \\
0 & a_{3(3-1)} & a_{33} & a_{3(3+1)} & 0 & a_{3(3+\sqrt{9})} & 0 & 0 & 0 \\
a_{4(4-\sqrt{9})} & 0 & a_{4(4-1)} & a_{44} & a_{4(4+1)} & 0 & a_{4(4+\sqrt{9})} & 0 & 0 \\
0 & a_{5(5-\sqrt{9})} & 0 & a_{5(5-1)} & a_{55} & a_{5(5+1)} & 0 & a_{5(5+\sqrt{9})} & 0 \\
0 & 0 & a_{6(6-\sqrt{9})} & 0 & a_{6(6-1)} & a_{66} & a_{6(6+1)} & 0 & a_{6(6+\sqrt{9})} \\
0 & 0 & 0 & a_{7(7-\sqrt{9})} & 0 & a_{7(7-1)} & a_{77} & a_{7(7+1)} & 0 \\
0 & 0 & 0 & 0 & a_{8(8-\sqrt{9})} & 0 & a_{8(8-1)} & a_{88} & a_{8(8+1)} \\
0 & 0 & 0 & 0 & 0 & a_{9(9-\sqrt{9})} & 0 & a_{9(9-1)} & a_{99}
\end{pmatrix}$$

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Durch die Spezielle Bandmatrix reduziert sich die Komponentenvorschrift auf

Daten Abhängigkeit!

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - a_{i(i-\sqrt{n})} \cdot x_{i-\sqrt{n}}^{(k+1)} - a_{i(i-1)} \cdot x_{i-1}^{(k+1)} - a_{i(i+1)} \cdot x_{i+1}^{(k)} - a_{i(i+\sqrt{n})} \cdot x_{i+\sqrt{n}}^{(k)} \right)$$

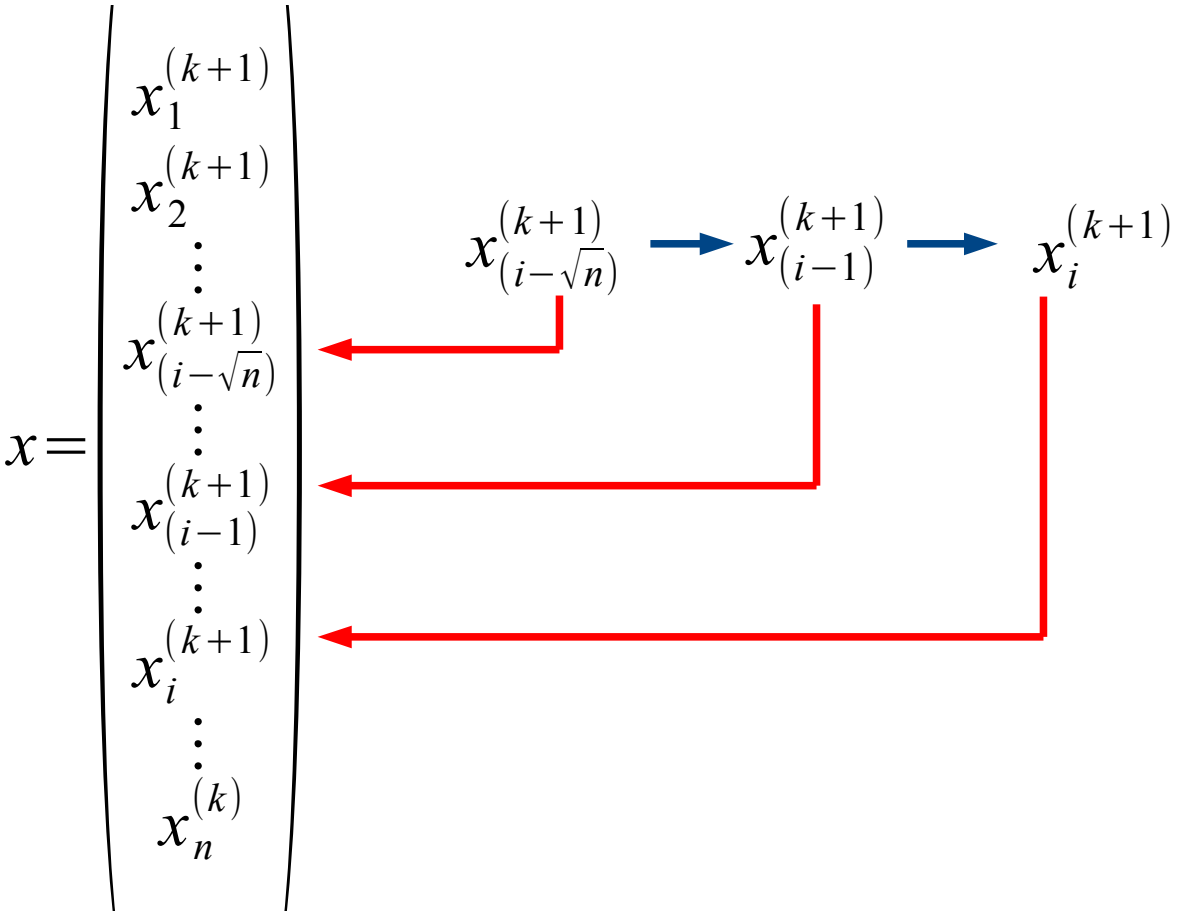
$$i = 1, \ldots, n$$

$$j = i - \sqrt{n}, i-1, i+1, i, i+\sqrt{n}$$

2. Gauß-Seidel-Verfahren
Für dünnbesetzte Matrizen

Freie Universität Berlin

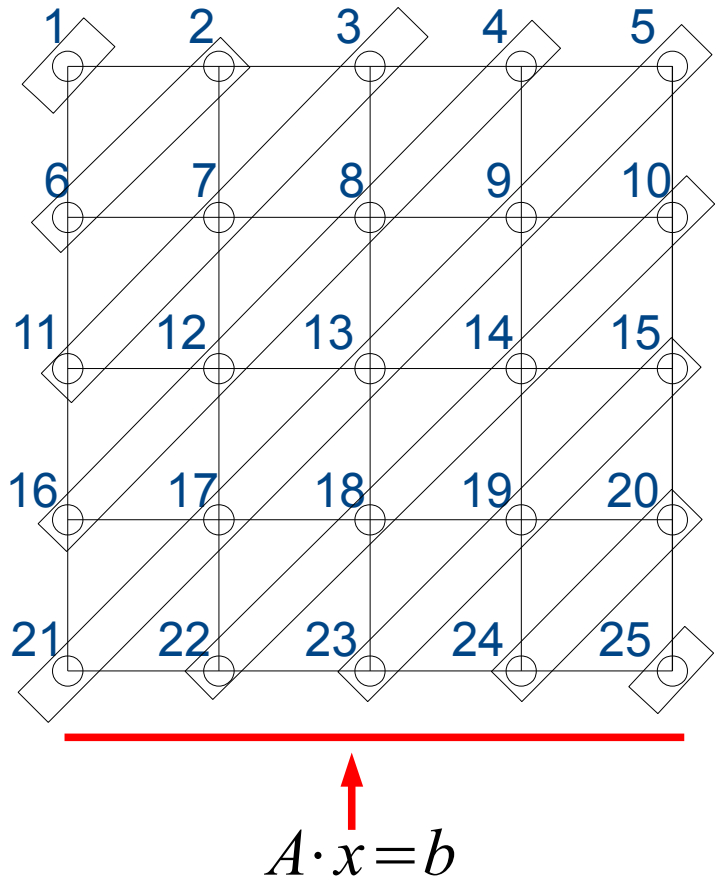
Abhängigkeiten von $x^{(k+1)}$ während einer Iteration



2. Gauß-Seidel-Verfahren
Für dünnbesetzte Matrizen

Freie Universität Berlin

Betrachtung der Komponenten von x als $\sqrt{n} \times \sqrt{n}$ Gittermatrix wobei $n = 25$



Gitterpunkt x_{ij} hängt von

$x_{i(j-1)}$ und $x_{(i-1)j}$ ab

Teilweise Aufhebung der Datenabhängigkeit
Diagonalen sind nicht abhängig voneinander!

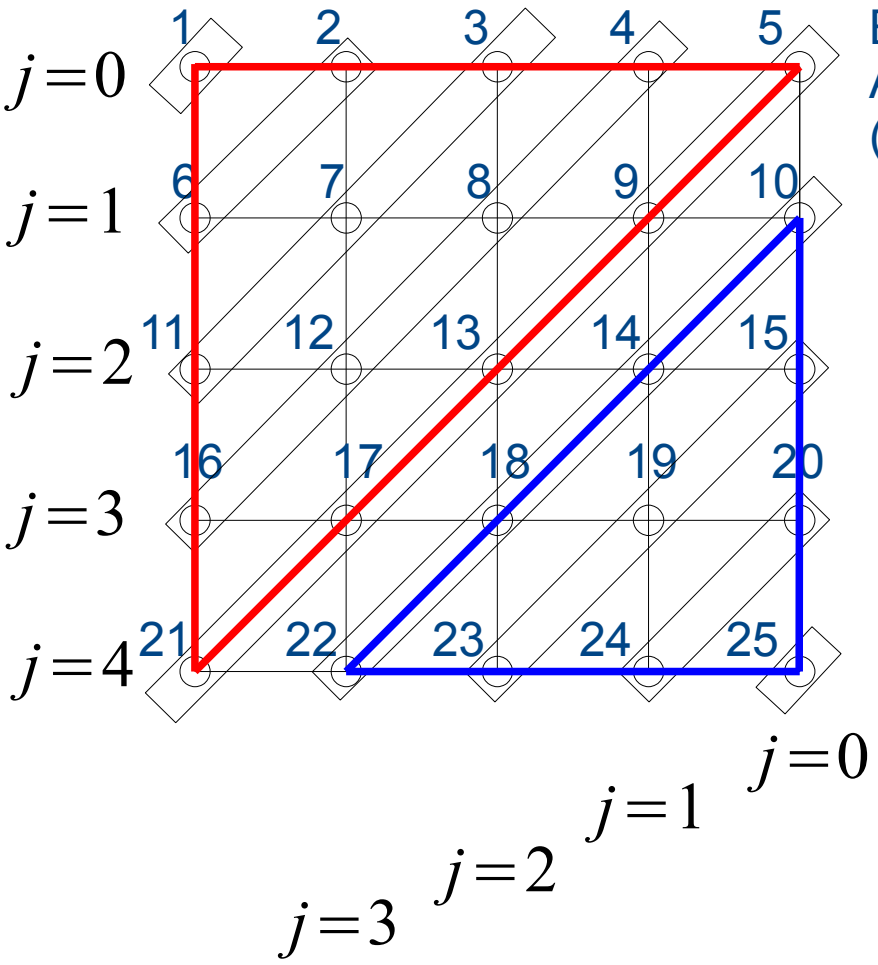
Verteilung der Gitterpunkte auf

pro Diagonale max. $\sqrt{n}$ Prozesse

2. Gauß-Seidel-Verfahren
Für dünnbesetzte Matrizen

Freie Universität Berlin

Berechnung der Indizes i von x



Bei Paralleler Implementierung
Aufteilung der Diagonalen in zwei Teile
(zwei Schleifen)

Für $l=1,\ldots,\sqrt{n}$

$i=l+j\cdot(\sqrt{n}-1)$ für $0\le j<l$

Für $l=2,\ldots,\sqrt{n}$

$i=l\cdot\sqrt{n}+j\cdot(\sqrt{n}-1)$ für $0\le j\le\sqrt{n}-l$

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Programmfragment:

```
sqn = sqrt(n);
do {
   for (l = 1; l <= sqn; l++) {
      for (j = me; j < l; j+=p) {
         i = l + j * (sqn-1) - 1; /* starte Nummerierung bei 0 */
         x[i] = 0;
         if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
         if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
         if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
         if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
         x[i] = (x[i] + b[i]) / a[i][i];
      }
      collect_elements(x,l);
   }
   for (l = 2; l <= sqn; l++) {
      for (j = me -l +1; j <= sqn -l; j+=p) {
         if (j >= 0) {
           i = l * sqn + j * (sqn-1) - 1;
           x[i] = 0;
           if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
           if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
           if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
           if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
           x[i] = (x[i] + b[i]) / a[i][i];
         }
      }
      collect_elements(x,l);
   }
} while(convergence_test() < tol);
```

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Erste Schleife:

Berechnung des Index für Feld x

```
for (l = 1; l <= sqn; l++) {
   for (j = me; j < l; j+=p) {
      i = l + j * (sqn-1) - 1; /* starte Nummerierung bei 0 */
      x[i] = 0;
      if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
      if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
      if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
      if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
      x[i] = (x[i] + b[i]) / a[i][i];
   }
   collect_elements(x,l);
}
```

Für $l = 1,\ldots,\sqrt{n}$

$i = l + j \cdot (\sqrt{n} - 1)$ für $0 \leq j < l$

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Erste Schleife:

Berechnung von x_i^{k+1}

```
for (l = 1; l <= sqn; l++) {
   for (j = me; j < l; j+=p) {
      i = l + j * (sqn-1) - 1; /* starte Nummerierung bei 0 */
      x[i] = 0;
      if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
      if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
      if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
      if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
      x[i] = (x[i] + b[i]) / a[i][i];
   }
   collect_elements(x,l);
}
```

$$x_i^{(k+1)} = \frac{1}{a_{ii}} (b_i - a_{i(i-\sqrt{n})} \cdot x_{i-\sqrt{n}}^{(k+1)} - a_{i(i-1)} \cdot x_{i-1}^{(k+1)} - a_{i(i+1)} \cdot x_{i+1}^{(k)} - a_{i(i+\sqrt{n})} \cdot x_{i+\sqrt{n}}^{(k)})$$

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Erste Schleife:

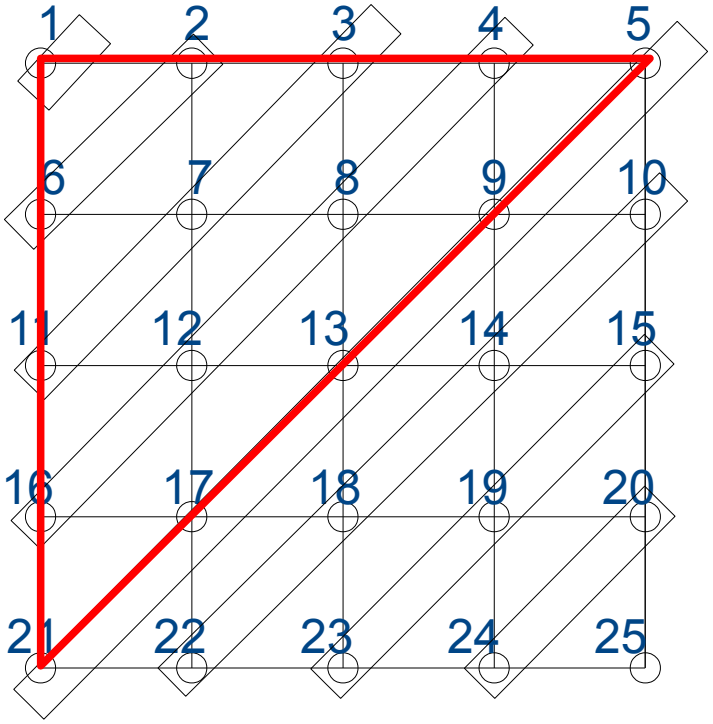
Prozesskommunikation

```
for (l = 1; l <= sqn; l++) {
   for (j = me; j < l; j+=p) {
      i = l + j * (sqn-1) - 1; /* starte Nummerierung bei 0 */
      x[i] = 0;
      if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
      if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
      if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
      if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
      x[i] = (x[i] + b[i]) / a[i][i];
   }
   collect_elements(x,l);
}
```

2. Gauß-Seidel-Verfahren
Für dünnbesetzte Matrizen

Freie Universität Berlin

Bis jetzt errechnete $x_i^{(k+1)}$



2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Zweite Schleife

Berechnung des Index i für Feld x

```
for (l = 2; l <= sqn; l++) {
   for (j = me -l +1; j <= sqn -l; j+=p) {
      if (j >= 0) {
        i = l * sqn + j * (sqn-1) - 1;
        x[i] = 0;
        if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
        if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
        if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
        if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
        x[i] = (x[i] + b[i]) / a[i][i];
      }
   }
   collect_elements(x,l);
}
```

$$l = 2,\ldots,\sqrt{n}$$

$$i = l\cdot\sqrt{n} + j\cdot\left(\sqrt{n}-1\right) \text{ für } 0 \le j \le \sqrt{n}-l$$

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Zweite Schleife

Berechnung von $x_i^{(k+1)}$

```
for (l = 2; l <= sqn; l++) {
   for (j = me -l +1; j <= sqn -l; j+=p) {
      if (j >= 0) {
        i = l * sqn + j * (sqn-1) - 1;
        x[i] = 0;
        if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
        if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
        if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
        if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
        x[i] = (x[i] + b[i]) / a[i][i];
      }
   }
   collect_elements(x,l);
}
```

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - a_{i(i-\sqrt{n})} \cdot x_{i-\sqrt{n}}^{(k+1)} - a_{i(i-1)} \cdot x_{i-1}^{(k+1)} - a_{i(i+1)} \cdot x_{i+1}^{(k)} - a_{i(i+\sqrt{n})} \cdot x_{i+\sqrt{n}}^{(k)} \right)$$

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Zweite Schleife

Berechnung von $x_i^{(k+1)}$

```
for (l = 2; l <= sqn; l++) {
   for (j = me -l +1; j <= sqn -l; j+=p) {
      if (j >= 0) {
        i = l * sqn + j * (sqn-1) - 1;
        x[i] = 0;
        if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
        if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
        if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
        if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
        x[i] = (x[i] + b[i]) / a[i][i];
      }
   }
   collect_elements(x,l);
}
```

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - a_{i(i-\sqrt{n})} \cdot x_{i-\sqrt{n}}^{(k+1)} - a_{i(i-1)} \cdot x_{i-1}^{(k+1)} - a_{i(i+1)} \cdot x_{i+1}^{(k)} - a_{i(i+\sqrt{n})} \cdot x_{i+\sqrt{n}}^{(k)} \right)$$

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Zweite Schleife

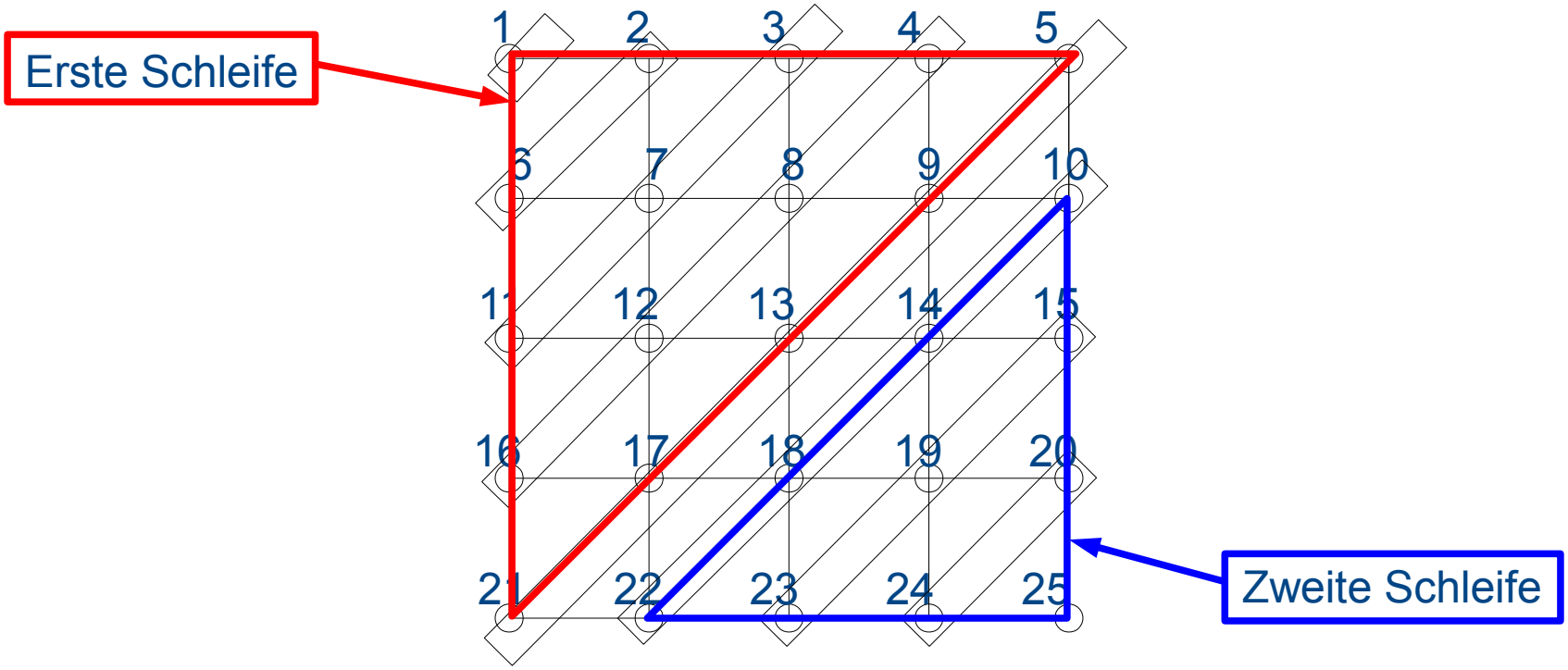
Prozesskommunikation

```
for (l = 2; l <= sqn; l++) {
   for (j = me -l +1; j <= sqn -l; j+=p) {
      if (j >= 0) {
        i = l * sqn + j * (sqn-1) - 1;
        x[i] = 0;
        if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
        if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
        if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
        if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
        x[i] = (x[i] + b[i]) / a[i][i];
      }
   }
   collect_elements(x,l);
}
```

2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

$x_i^{(k+1)}$ Ist für einen Schritt komplett berechnet



2. Gauß-Seidel-Verfahren Für dünnbesetzte Matrizen

Freie Universität Berlin

Programmfragment:

```
sqn = sqrt(n);
do {
   for (l = 1; l <= sqn; l++) {
      for (j = me; j < l; j+=p) {
         i = l + j * (sqn-1) - 1; /* starte Nummerierung bei 0 */
         x[i] = 0;
         if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
         if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
         if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
         if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
         x[i] = (x[i] + b[i]) / a[i][i];
      }
      collect_elements(x,l);
   }
   for (l = 2; l <= sqn; l++) {
      for (j = me -l +1; j <= sqn -l; j+=p) {
         if (j >= 0) {
           i = l * sqn + j * (sqn-1) - 1;
           x[i] = 0;
           if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
           if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
           if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
           if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
           x[i] = (x[i] + b[i]) / a[i][i];
         }
      }
      collect_elements(x,l);
   }
} while(convergence_test() < tol);
```

2. SOR-Verfahren

Freie Universität Berlin

Das SOR-Verfahren relaxiertes Gauß-Seidel-Verfahren
(engl. successive overrelaxtion)
Konvergiert schneller als Gauß-Seidel-Verfahren

Zerlegung der Matrix A in:

Sei: $A \in \mathbb{R}^{n \times n}$

$$A = D - L - R$$ Gauß-Seidel-Verfahren

$$A = \frac{1}{\omega} D - L - R - \left(\frac{(1-w)}{\omega}\right) D$$ SOR-Verfahren

2. SOR-Verfahren

Daraus ergibt sich die Matrixvorschrift pro Iterationsschritt

$$(D-\omega L)\,x^{(k+1)}=(1-\omega)\,Dx^{(k)}+\omega\,R\,x^{(k)}+\omega\,b$$

Umgeformt:

$$x^{(k+1)}=(D-\omega L)^{-1}\left((1-\omega)\,Dx^{(k)}+\omega\,R\,x^{(k)}+\omega\,b\right)$$

Die Berechnung pro Iterationsschritt

Freie Universität Berlin

2. SOR-Verfahren

Komponentenvorschrift:

$$\hat{x}_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) \qquad i = 1,2,\ldots,n$$

$\hat{x}_i^{(k+1)}$ Enthält Zwischenergebnis der Aktuellen Iteration

$$x_i^{(k+1)} = x_i^{(k)} + \omega \left(\hat{x}_i^{(k+1)} - x_i^{(k)} \right)$$

Enthält Ergebnis der Aktuellen Iteration

Freie Universität Berlin

2. SOR-Verfahren

Zusammengesetzte Komponentenvorschrift: $i = 1,2,\ldots,n$

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{(k)}\right) + (1-\omega)\, x_i^{(k)}$$

$x_i^{(k+1)}$ Enthält Zwischenergebnis der Aktuellen Iteration

Konvergiert für jeden Startvektor $x^{(0)}$
Bei positiv definite symmetrische Matrizen $\omega \in (0,2)$

Freie Universität Berlin

Cholesky-Faktorisierung für dünnbesetzte Matrizen

Voraussetzungen:

$$A\ symmetrisch \rightarrow a_{ij} = a_{ji}$$

$$A\ positiv\ definit \rightarrow \forall\, x \in R^n,\ x \neq 0 : x^T\ A\, x > 0$$

Dann gilt:

$$A = L \cdot L^T$$

,wobei L untere Dreiecksmatrix mit positiver Diagonalen ist.

Definition

Freie Universität Berlin

Lösung des Gleichungssystems:

$$L\,y = b \qquad L^T\,x = y$$

, denn es gilt:

$$L\,y = L\,L^T\,x = A\,x = b$$

Faktorisierung:

$$\text{for } (\,j = 0\,;\; j < n\,;\; j{+}{+}) :$$

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=0}^{j-1} l_{jk}^2}\,;$$

$$\text{for } (i = j + 1\,;\, i < n\,;\, i{+}{+}) :$$

$$l_{ij} = \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=0}^{j-1} l_{jk}\, l_{ik} \right);$$

Strukturierung

Freie Universität Berlin

Wir definieren zwei Strukturen:

$$Struct(L_{*j}) = \{ k > j \mid l_{kj} \neq 0 \}$$, nicht null in Spalte j

$$Struct(L_{i*}) = \{ k < i \mid l_{ik} \neq 0 \}$$, nicht null in Zeile i

und zwei Operationen:

$$cmod(j,k) =$$

$$\text{for each } i \in Struct(L_{*k}) \text{ with } i \geq j: \; a_{ij} = a_{ij} - l_{jk} l_{ik};$$

$$cdiv(j) =$$

$$l_{jj} = \sqrt{a_{jj}};$$

$$\text{for each } i \in Struct(L_{*j}) \text{ with } i \geq j: \; l_{ij} = \frac{a_{ij}}{l_{jj}};$$

Algorithmen

Freie Universität Berlin

Left-Looking-Algorithmus:

$$left\_cholesky =$$

$$\text{for } (\, j = 0\,;\; j < n\,;\; j{+}{+}) :$$

$$\text{for each } k \in Struct(L_{j*}):\; cmod(j, k);$$

$$cdiv(j);$$

Right-Looking-Algorithmus:

$$right\_cholesky =$$

$$\text{for } (\, j = 0\,;\; j < n\,;\; j{+}{+}) :$$

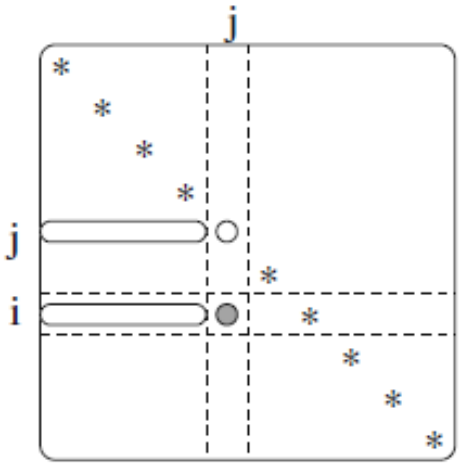
$$cdiv(j);$$

$$\text{for each } k \in Struct(L_{*j}):\; cmod(k, j);$$

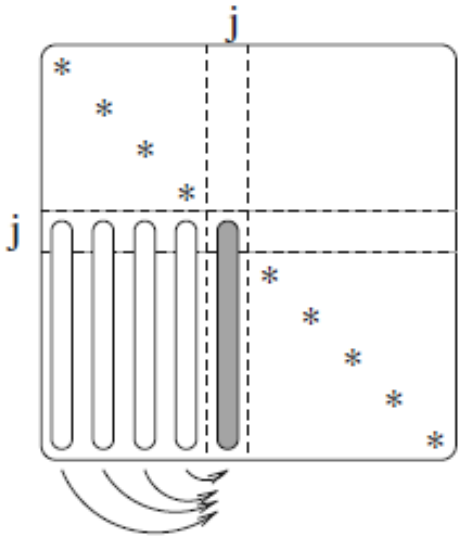
Zugriffsstrukturen

Freie Universität Berlin

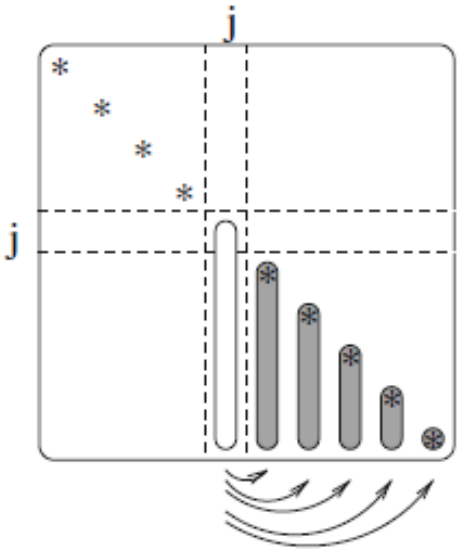
Zugriffsstruktur
zur Berechnung von l_{ij}



Zugriffsstruktur
beim Left-Looking-Verfahren



Zugriffsstruktur
beim Right-Looking-Verfahren



aktualisierte Datenelemente

benutzte Datenelemente

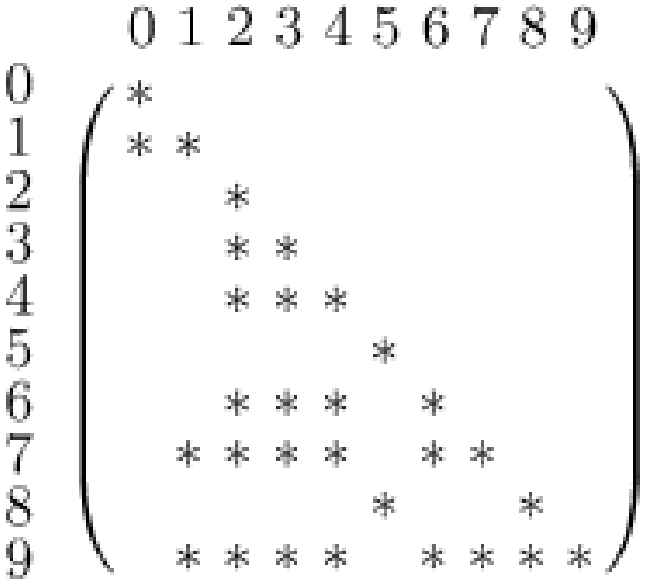
Superknoten

Definition:

$$\forall i,\ p \le i \le p+q-1:$$

$$Struct(L_{*i}) = Struct(L_{*\,p+q-1}) \cup \{i+1, \ldots, p+q-1\}$$

Beispiel:



Superknoten in Spalten:

{0, 1, 2, 5, 6, 8}

Superknoten

Freie Universität Berlin

Definieren Operation:

$$smod(j, S) =$$
$$r = min\{ j-1, last(S)\};$$
$$\text{for } (k = first(S); k \le r; k++):$$
$$cmod(j, k);$$

Ist Spalte j innerhalb von S, dann nutze Spalten in S links von j, sonst nutze alle Spalten von S.

Superknoten

Freie Universität Berlin

Right-Looking-Superknoten-Algorithmus:

$$supernode\_cholesky =$$

for each $supernode\ S$ do from left to right :

$cdiv(first(S));$

for ($j = first(S)+1$; $j \le last(S)$; $j++$) :

$smod(j, S);$

$cdiv(j);$

for each $k \in Struct(L_{*\,last(S)})$:

$smod(k, S);$

Abspeicherungsschemata

Freie Universität Berlin

Folgende Variablen werden definiert:

nz	Anzahl der Nicht-Null-Einträge
Nonzero[nz]	Nicht-Null-Einträge (spaltenweise linear)
Row[nz]	Zeilenindizes der Nicht-Null-Einträge
StartColumn[n]	Index des ersten Eintrags aller Spalten (innerhalb *Nonzero*)

Fordert viel Overhead-Speicher, deshalb nur sinnvoll für dünnbesetzte Matrizen.

Abspeicherungsschemata

Freie Universität Berlin

Bei Verwendung von Superknoten:

Definieren zusätzliches Array *StartRow[n]*, das die Indizes von Row, von den ersten Nicht-Null-Zeilenindizes der Spalten enthält.

Aufgrund der Besetzungsstruktur von Superknoten, müssen in Row nur noch die Zeilenindizes der ersten Superknotenspalten gespeichert werden.

$$Struct(L_{*j}) = \{ Row[StartRow[j]+i] \mid 1 \le i \le StartColumn[j+1] - StartColumn[j] \}$$

Datenabhängigkeiten

Freie Universität Berlin

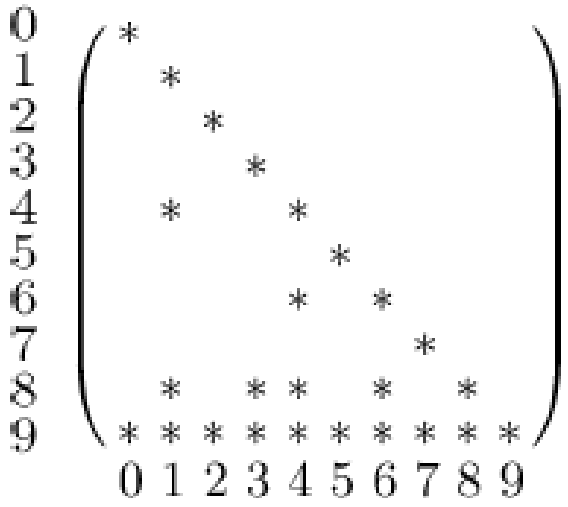
Wollen Datenabhängigkeiten bestimmen.

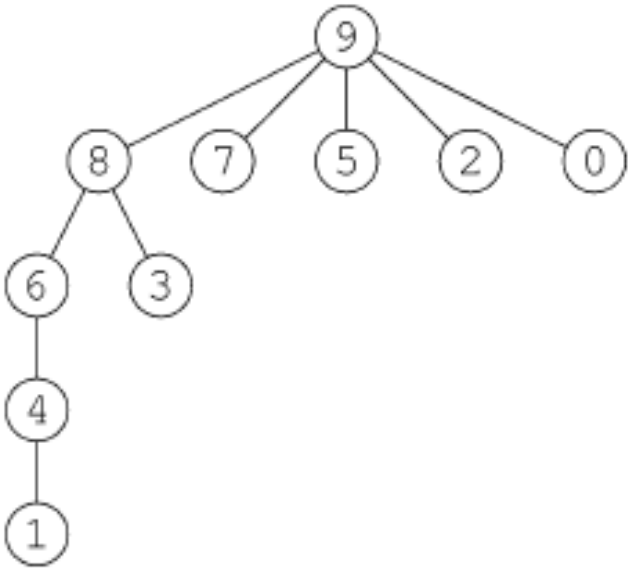
Definieren $parent(j) = min\{ i \mid i \in Struct(L_{*j}) \}$

(Erste Spalte *i*, die von Spalte *j* abhängt)

Erzeugen „Eliminationsbaum“, der eine Kante *(i, j)* hat, wenn *i = parent(j)*.

Beispiel:





Datenabhängigkeiten

Freie Universität Berlin

Sei *G[j]* Unterbaum mit Wurzel *j*.

Zwei Spalten *j* und *i* sind genau dann parallel berechenbar, wenn *G[j]* und *G[i]* disjunkte Teilbäume sind.
(Blätter sowieso unabhängig von einander.)

Also gilt:

Grad der Parallelität ist antiproportional zur Tiefe des Eliminationsbaumes.

Parallelisierung (Left-Looking-Algorithmus)

Freie Universität Berlin

Definieren Spaltentasks *Tcol(0),..., Tcol(n – 1)*
Spaltentask *Tcol(j)* berechnet Spalte *j*.

Also: $\forall k \in Struct\, L_{j*} : cmod(j,k);$ und $cdiv(j);$

Spaltentasks sind in zentralem Taskpool.
→ dynamische Taskverwaltung
→ Lock wegen konkurrierenden Zugriffen benötigt

Einfügen der Spaltentasks in zentralen Pool

Freie Universität Berlin

1. Variante

Tcol(j) wird erst eingefügt, wenn alle *Tcol(k)* mit *G[k]* Teilbaum von *G[j]* ausgeführt wurden.
→ Prozessor kann Task ohne Unterbrechung ausführen

Initialisierung: Blätter des Eliminationsbaums

2. Variante

Tcol(j) kann bearbeitet werden auch, wenn noch nicht alle entsprechenden *Tcol(k)* ausgeführt wurden.
→ Prozessor kann *cmod(j,k)* nur für Spalten *k* ausführen, für die *Tcol(k)* bereits ausgeführt wurde.

Initialisierung: Alle Knoten sortiert nach Abhängigkeit

Einfügen der Spaltentasks in zentralen Pool

Freie Universität Berlin

Zum Verwalten der durchführbaren Operationen, definiere für jede Spalte *j* einen Stack C_j.

C_j enthält Zeilenindizes der berechenbaren Spalten.

Hat ein Prozessor $|Struct(L_{j*})|$ Elemente aus C_j entfernt, ist *Tcol(j)* fertig und der Prozessor muss in jedes C_k mit $k \in Struct(L_{*j})$ j hinein schreiben.

Alle Stacks C müssen mit einem Lock geschützt werden.

Implementierung

Freie Universität Berlin

$$
\begin{aligned}
&parallel\_left\_cholesky = \\
&\quad c = 0; \\
&\quad \text{while } \left(\left(j = MPADD(c,1) \right) < n \right) : \\
&\qquad \text{for } \left(i = 0; i < |Struct(L_{j*})|; i++ \right) : \\
&\qquad\quad \text{while } \left(C_j empty \right) : \ wait(); \\
&\qquad\quad k = pop(C_j); \\
&\qquad\quad cmod(j,k); \\
&\qquad cdiv(j); \\
&\qquad \text{for } \left(i \in Struct(L_{*j}) \right) : \ push(j, C_i);
\end{aligned}
$$

Parallelisierung (Right-Looking-Algorithmus)

Freie Universität Berlin

Definieren Spaltentasks *Tcol(0),..., Tcol(n – 1)*
Spaltentask *Tcol(j)* beendet Spalte *j* und führt Berechnung auf alle abhängigen Spalten aus.

Also: $cdiv(j);$ und $\forall k \in Struct\, L_{*j} : cmod(k, j);$

Taskpool wie beim Left-Looking.

Einfügen der Spaltentasks in zentralen Pool

Freie Universität Berlin

Tcol(j) wird erst eingefügt, sobald *cdiv(j)* ausgeführt werden kann. Dazu definiere Zähler c_j (Lock-geschützt), der die Anzahl *cmod(j,k)* Operationen zählt.

Zielspalten *k* der *cmod(k,j)* Operationen müssen mit Lock geschützt werden.

Initialisierung: Blätter des Eliminationsbaums

Implementierung

Freie Universität Berlin

```
parallel_right_cholesky=
    c=0;
    initialize_task_pool(TP);
    while ((j=MPADD(c,1))<n) :
        while (!filled_pool(TP, j)) : wait();
        getColumn(j);
        cdiv(j);
        for (k∈Struct(L*j)) :
                lock(k); cmod(k, j); unlock(k);
                if (MPADD(ck,1)+1==|Struct(Lk*)|) :
                    addColumn(k);
```

Parallelisierung (Superknoten-Algorithmus)

Freie Universität Berlin

Definieren Superknotentasks *Tsup(S_0),..., Tsup(S_n)*, wobei S_i fundamentaler Superknoten.

Fundamentaler Superknoten:
Knoten *(p + i)* ist einziges Kind von *(p + i + 1)*, $0 \leq i \leq q-2$

Ist ein Superknoten fundamental, so können alle Spalten innerhalb des Knotens berechnet werden, sobald die erste berechnet werden kann.

Taskpool wie gehabt.

Einfügen der Spaltentasks in zentralen Pool

Freie Universität Berlin

Tsup(S) wird erst eingefügt, wenn alle nötigen *smod(k,S)* ausgeführt worden sind. Dazu definiere Zähler c_S *(Lock)*.

Zielspalten *k* der *smod(k,S)* Operationen müssen mit Lock geschützt werden.

Initialisierung: Superknoten die an Blättern des Eliminationsbaums beginnen.

Implementierung

Freie Universität Berlin

```
parallel_supernode_cholesky=
    c=0;
    initialize_task_pool(TP);
    while ((S=MPADD(c,1))<N) :
        while (!filled_pool(TP,S)) : wait();
        get_supernode(S);
        cdiv(first(S));
        for (j=first(S)+1; j≤last(S); j++) :
            smod(j,S);
            cdiv(j);
```

Implementierung

Freie Universität Berlin

$$\text{for } (k \in Struct(L_{*last(S)})):$$

$$lock(k);\ smod(k, S);\ unlock(k);$$

$$MPADD(c_k, 1);$$

$$\text{if } (k == first(K)\ \&\&\ c_k + 1 == |Struct(L_{k*})|):$$

$$add\_supernode(K);$$

Diskussion der Varianten

Freie Universität Berlin

	Left-Looking	Right-Looking	Superknoten
Vorteile	Kein Lock-Mechanismus auf Spalten	Geringer Aufwand bei Datenverwaltung	Weniger Lock-Konkurrenz Geringer Aufwand bei Datenverwaltung
Nachteile	Größerer Aufwand bei Datenverwaltung	Tasks müssen bei leerem Pool warten	Tasks müssen bei leerem Pool warten

Auf SB_PRAM am Besten: Left-Looking

Quellen

Freie Universität Berlin

Literatur:

Thomas Rauber, Gudula Rünger: Parallele Programmierung.
Springer - Verlag, 2012

Internet:

http://www.scai.fraunhofer.de/fileadmin/download/mathematik_praxis/gls/gls_skript_und_arbeitsblaetter.pdf

http://www.math.uni-hamburg.de/home/hofmann/lehrveranstaltungen/sommer05/prosem/Vortrag5.pdf

http://de.wikipedia.org/wiki/Dünnbesetzte_Matrix

http://de.wikipedia.org/wiki/Bandmatrix

http://mo.mathematik.uni-stuttgart.de/kurse/kurs48/seite46.html

http://www.hdm-stuttgart.de/redaktionzukunft/fokus_einzel_uebersicht.html?dossier_ID=63