4. Verteilte Numerische Algorithmen

Bisher im wesentlichen Beschreibung von
- parallelen/verteilten Architekturen
- Programmiermethoden
- einfachen numerischen Basisoperationen

Im folgenden Kapitel

Überblick über numerische Algorithmen aus verschiedenen Anwendungsgebieten und deren Parallelisierung

Parallele Versionen der Algorithmen
- lassen sich teilweise durch einfache Anwendung paralleler numerischer Basisoperationen realisieren
(inkl. Basiskommunikationsoperationen)
- erfordern zum Teil aber auch völlig neue Konzepte

4.1 Numerische Basisalgorithmen

4.2 Lösung linearer Gleichungssysteme

4.3 Nichtlineare Gleichungssysteme

4.4 Schnelle Fourier-Transformation

4.5 Optimierungsprobleme

4.6 Asynchrone Algorithmen

4.1 Numerische Basisalgorithmen

- Numerische Basisalgorithmen bilden die Basis für komplexere Algorithmen
- Effiziente (parallele) Realisierung der Algorithmen ist notwendig für eine effiziente Realisierung komplexerer numerischer Verfahren

Ansatz in der Numerik:
- Definition der benötigten Operationen (auf Vektoren und Matrizen)
- Festlegung der Funktionsschnittstellen (ursprünglich Fortran, inzwischen auch C)
- Implementierung der Funktionen
  - in Fortran/C als portable Versionen
  - in Assembler mit maschinenspezifischer Optimierung

⇒ Basic Linear Algebra Subprograms (BLAS) über die Netlib (`www/netlib/org`) frei verfügbar

Zur parallelen Realisierung wurde PBLAS entwickelt:
- parallele Realisierung einer Untermenge von BLAS Routinen
- Kommunikation mit Hilfe von BLACS
  - portable auf Basis von MPI oder PVM
  - architekturspezifisch von einzelnen Herstellern

Unterteilung der BLAS Routinen in drei Gruppen
- Vektor-Vektor-Operationen (BLAS 1)
- Vektor-Matrix-Operationen (BLAS 2)
- Matrix-Matrix-Operationen (BLAS 3)

für unterschiedliche Datentypen (real, double, complex) und Matrizen (dicht, bandstrukturiert, symmetrisch, ...)

Sequentielle Implementierung und Implementierung mit geteiltem Speicher recht einfach protabel zu halten.

Bei verteiltem Speicher bestimmt die Rechnerstruktur den Algorithmus (siehe vorheriges Kapitel)

Deshalb
- Festlegung einer Topologie (hier naheliegend ein Gitter)
- Realisierung der Algorithmen auf der Topologie
- Einbettung der Topologie in die Topologie des Parallelrechners

BLACS Routinen wurden primär für Kommunikation in einem Gitter entworfen
- Operationen zum Versenden und Empfangen von Vektoren und Matrizen
- Möglichkeit der Gruppenadressierung (Zeile, Spalte, Alle)

Aufbau der Operationen in PBLAS
- Verwendung von BLAS Routinen zur lokalen Berechnung
- Verwendung von BLACS Routinen zur Kommunikation

Datenverteilung in PBLAS: Blockzyklische Schachbrettverteilung von Matrizen auf dem Gitter
- seien r, c die Matrixdimensionen,
- s, t die Gitterdimensionen ($p = s \cdot t$)
- rl, cl die Dimensionen der Matrixblöcke
- so, to Adresse des Prozessors, der den ersten Block speichert

Für gegebene (r, c) und (s, t) definiert (rl, cl, so, to) eine eindeutige Zuordnung

Beispiel Datenverteilung einer 5×5 Matrix auf einem 2×2 Gitter bzgl. $(2, 2, 0, 0)$:

A_{11}	A_{12}	A_{15}	A_{13}	A_{14}
A_{21}	A_{22}	A_{25}	A_{23}	A_{24}
A_{51}	A_{52}	A_{55}	A_{53}	A_{54}
A_{31}	A_{32}	A_{35}	A_{33}	A_{34}
A_{41}	A_{42}	A_{45}	A_{43}	A_{44}

Verteilung möglichst so, dass
- die Last gleichmäßig verteilt wird
- zusammenhängende Blöcke auf einem Prozessor in den Hauptspeicher passen
- die Speicherung mehrerer mittelgroßer Blöcke auf einem Prozessor kann gegenüber der Speicherung eines großen Blocks vorteilhaft sein (Überlappung Berechnung Kommunikation)

BLAS 1 Operationen (Vektor-Vektor-Operationen)

x, y sind Vektoren, α, β Skalare

- $x \leftrightarrow y$ (vertauschen der Elemente in x und y)
- $x \leftarrow \alpha y$ (Vektor-Skalar-Multiplikation)
- $y \leftarrow x$ (kopieren eines Vektors)
- $y \leftarrow \alpha x + y$ (Addition eines Vektor-Skalar-Produkts)
- $\beta \leftarrow x^T Y$ (Vektor-Produkt von Vektoren)
- Berechnung von Vektornormen

Beispielimplementierung Vektor-Produkt

jeder Prozessor k speichert Subvektoren $x_{(k-1)g,\dots,kg}$ und $y_{(k-1)g,\dots,kg}$ wobei $n = p/g$
(n Vektorlänge, p `Prozessorzahl`)

Algorithmus für einen Prozessor

```
double x[g], y[g] ; /* Enthalten die Teilvektoren */
double res=0.0 ; /* zur Aufnahem des Ergebnisses */
int i ;
for (i = 0; i < g; i++)
   res += x[i] * y[i] ;
MPI_Reduce(..) oder MPI_Allreduce(...) ;
```


A, B, C sind Matrizen, T eine obere Dreiecksmatrix, x, y sind Vektoren, α, β Skalare

BLAS 2 Operationen (Matrix-Vektor-Operationen)

- $y \leftarrow \alpha Ax + \beta y$ und $y \leftarrow \alpha A^T x + \beta y$ (Matrix-Vektor-Produkte)
- $A \leftarrow \alpha xy^T + A$ (Rang-1-Modifikationen)
- $A \leftarrow \alpha xx^T + A$ und $A \leftarrow \alpha xy^T + +\alpha yx^T + A$ (Rang-1/2-Modifikationen symmetrischer Matrizen)
- $x \leftarrow Tx$ und $x \leftarrow T^T x$ (Mult. mit Dreiecksmatrix)
- $x \leftarrow T^{-1}x$ (Lsg. lin. Gl.Sys. mit Dreiecksmatrix)

Beispielimplementierung Matrix-Vektor-Produkt siehe Kap. 2

Beispielimplementierung Lsg. lin. Gl.Sys. mit Dr.Mat. siehe 4.2.

BLAS 3 Operationen (Matrix-Matrix-Operationen)

- $C \leftarrow \alpha AB + \beta C$, $C \leftarrow \alpha A^T B + \beta C$, ... (Matrix-Matrix-Produkte)
- $C \leftarrow \alpha xAA^T + C$, $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$, ... (Rang-k-Modifikationen symmetrischer Matrizen)
- $B \leftarrow \alpha TB$, $B \leftarrow \alpha T^T B$, ... (Mult. mit Dreiecksmatrix))
- $B \leftarrow \alpha T^{-1} B$, $B \leftarrow \alpha BT^{-1}$ (Lsg. lin. Gl.Sys. mit Dreiecksmatrix)
- $C \leftarrow \beta C + \alpha A^T$ (Trans. Matrix)

Matrix-Matrix-Multiplikation

- oft verwendetees Benchmarkproblem,
- welches in der Praxis recht selten auftritt

Sequentielles Vorgehen

```
for (i = 0; i < n; i++)
   for (j = 0; j < m; j++)
      for (k = 0; k < r; k++)
         C[i][k] += A[i][j] * B[j][k] ;
```

Sequentielle Implementierungsalternativen durch unterschiedliche Anordnung der Schleifen

Parallele Implementierung blockorientierte Verfahren:

Matrizen werden in Blöcke $A^{(x,y)}$und $B^{(y,z)}$zerlegt, mit $0 \le x < X \le n$, $0 \le y < Y \le m$ und $0 \le z < Z \le r$

Matrix $A^{(x,y)}$enthält $u \in [\lfloor n/X \rfloor, \lceil n/X \rceil]$ Zeilen
und $v \in [\lfloor m/Y \rfloor, \lceil m/Y \rceil]$ Spalten

Matrix $A^{(y,z)}$enthält $u \in [\lfloor m/Y \rfloor, \lceil m/Y \rceil]$ Zeilen
und $w \in [\lfloor r/Z \rfloor, \lceil r/Z \rceil]$ Spalten

Verwendung von $X \cdot Y \cdot Z$ Prozessoren

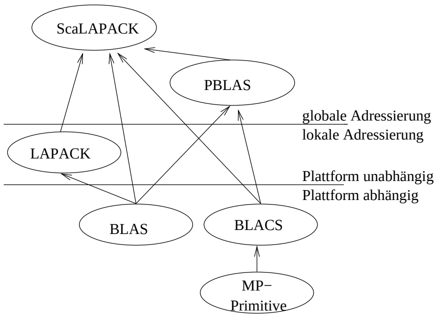
Prozessor (x, y, z) berechnet $C^{(i,j,k)} = A^{(i,k)} \cdot B^{(k,j)}$

Berechnung von $C^{(i,j)} = \sum_{k=1}^{Y} C^{(i,j,k)}$ per Akkumulation

Numerische Programme/Routinen sind unter
www.netlib.org zu finden

Umfangreiche Sammlung von numerischen Algorithmen
(oft Fortran, teilweise auch C)

Einige parallele Algorithmen z.B.ScaLAPACK
für Systeme mit verteiltem Speicher



Enthaltene Routinen zur Lösung von

- linearen Gleichungssystemen
  - dichtbesetzt, bandstrukturiert, tridiagonal
- Lineare Regression
- Eigenwerten/-vektoren
- SVD
- "Out-of-core" Löser (LU, QR, Cholesky, ...)

4.2 Lösung linearer Gleichungssysteme

Lösen linearer Gleichungssysteme

- ist zentrale Operation im wissenschaftlichen Rechnen
- wird oftmals als Subprogramm komplexer numerischer Algorithmen verwendet

Wir betrachten Lösungsverfahren für
$Ax = b$ mit $A \in \mathbb{R}^{n\times n}$ und $x, b \in \mathbb{R}^n$
auf einem Rechner mit verteiltem Adressraum
Andere Sichtweise:
Darstellung von b als Linearkombination der Spalten von A
Gleichungssystem lösbar falls A nichtsingulär
Klassifizierung von Lösungsverfahren

- direkte Verfahren
  - berechnen (bis auf Rundungsfehler) exakte Lösung
  - Aufwand fest und abhängig von Matrixgröße und Besetzungsstruktur
- iterative Verfahren
  - berechnen konsekutive Näherungen der Lösung
  - Aufwand abhängig von gewünschter Genauigkeit sowie Größe und Besetzungsstruktur der Matrix

Vorteile direkter Verfahren

- Berechnung der exakten Lösung
  (Vorsicht Rundungsfehler!)
- Speicherplatz und Laufzeit vorhersagbar

Nachteile direkter Verfahren

- oft hoher Aufwand und Speicherplatzbedarf (durch *fill-in*)
- parallele Implementierung oft komplex

Vorteile iterativer Verfahren

- Matrixstruktur bleibt erhalten
  (kein *fill-in*)
- geringer Speicherbedarf bei spärlich besetzten Matrizen
- oft einfach parallelisierbar

Nachteile iterativer Verfahren

- Konvergenz nicht immer gesichert
- Laufzeit zum Erreichen einer vorgegebenen Genauigkeit nicht vorhersagbar

In der Praxis werden oft iterative Verfahren verwendet, da Matrizen oft groß und sehr spärlich besetzt sind

4.2.1 Direkte Verfahren für allgemeine Matrizen

Lösung mittels der Gauß-Elimination
Wir betrachten erst den sequentiellen Algorithmus,
dann dessen Parallelisierung

$$\begin{array}{ccccccccc}
A_{11}x_1 & + & A_{12}x_2 & + & \dots & + & A_{1n}x_n & = & b_1 \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
A_{i1}x_1 & + & A_{i2}x_2 & + & \dots & + & A_{in}x_n & = & b_i \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
A_{n1}x_1 & + & A_{n2} & + & \dots & + & A_{nn}x_n & = & b_n
\end{array}$$

Sei $A^{(1)} = A$ und $b^{(1)} = b$
Erzeugen von $A^{(2)}, \dots, A^{(n)}$ und $b^{(2)}, \dots, b^{(n)}$, so dass $A^{(k)}x = b^{(k)}$ gilt und $A^{(k)}$ die folgende Gestalt hat

$$\begin{pmatrix}
A_{11} & A_{12} & \cdots & A_{1,k-1} & A_{1k} & \cdots & A_{1n} \\
0 & A_{22}^{(2)} & \cdots & A_{2,k-1}^{(2)} & A_{2k}^{(2)} & \cdots & A_{2n}^{(2)} \\
\vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\
\vdots & & \ddots & A_{k-1,k-1}^{(k-1)} & A_{k-1,k}^{(k-1)} & \cdots & A_{k-1,n}^{(k-1)} \\
\vdots & & & 0 & A_{kk}^{(k)} & \cdots & A_{kn}^{(k)} \\
\vdots & & & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & \cdots & 0 & A_{nk}^{(k)} & \cdots & A_{nn}^{(k)}
\end{pmatrix}$$

$\Rightarrow A^{(n)}$ ist eine obere Dreiecksmatrix!

Berechnung der Lösung
$A^{(n)}x = b^{(n)}$
per Rücksubstitution

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

sequentielle Implementierung offensichtlich
Parallele Implementierung mit p-Prozessoren:

- Gleichmäßige Verteilung der Matrixzeilen und zugehörigen Vektorelemente auf die Prozessoren (sei $k = n/p$)
- Prozessor i erhält Zeilen $i \cdot k + 1, \dots, (i+1)\cdot k$ (Prozessornummerierung $0, \dots, p-1$)
- Idee des Algorithmus
  - Prozessoren starten mit lokaler Berechnung
  - immer, wenn ein Prozessor fertig geworden ist, verteilt er seinen Vektor an alle anderen mit kleinerer Adresse
  - nach Empfang eines Vektors berechnen Prozessoren zugehöriges Vektor-Matrix Produkt

Aufwand

- p Berechnungsschritte mit jeweils k^2 bzw. $k^2/2$ Multiplikationen und Additionen
- $p-1$ Multicasts mit jeweils k Werten

Algorithmus für Prozessor pr

```
double b[k] /* enthält Teilvektor von b^(n) */
double x[k], y[k] /*Teillösungsvektor und Empfangspuffer*/
double a[k][n] /* Zeilen von A^(n) */
for (i = p-1; i >= 0; i–) {
  if (pr <= i) {
    if (i == p-1) {
      for (j = k-1; j >= 0; j–)
        x[j] = b[j] ;
      if (pr == p-1)
        for (j = k-1; j >= 0; j–)
          for (l = j+1; l < k; l++)
            x[j] = (x[j] + a[j][l + i * k] * x[l]); }
    else
      for (j = k-1; j >= 0; j–)
        for (l = j+1; l < k; l++)
          x[j] = x[j] + a[j][l + i · k] · y[l] ;
    if (pr == i) {
      for (j = k-1; j >= 0; j–)
        x[j]/ = a[j][j + pr * k]
      /* verschicke x an Prozessoren 0, ... pr − 1*/ }
    if (pr < i)
      /* empfange Vektor x[i * k, ..., (i + 1) · k − 1]
                in y */
} }
```


Berechnung von $A^{(k+1)}$ aus $A^{(k)}$

Ziel: Nullsetzen der Elemente $A_{ik}^{(k)}$ für $i = k + 1, \ldots, n$ durch Subtraktion eines Vielfachen der k-ten Zeile von der i-ten Zeile

Unter der Voraussetzung $A_{kk}^{(k)} \neq 0$ berechne

- $L_{ik} = A_{ik}^{(k)} / A_{kk}^{(k)}$
- $A_{ij}^{(k+1)} = A_{ij}^{(k)} - L_{ik} A_{kj}^{(k)}$
- $b_i^{(k+1)} = b_i^{(k)} - L_{ik} b_k^{(k)}$

für $i, j = k + 1, \ldots, n$

Falls $A_{kk}^{(k)} = 0$ (oder klein) $\Rightarrow$

- Spaltenpivotsuche
  - suche $r = \max_{i=k,\ldots,n}(|A_{ik}^{(k)}|)$ (Pivotelement)
  - vertausche Zeile k und r sowie $b_r^{(k)}$ und $b_k^{(k)}$
- Zeilenpivotsuche
  - suche $c = \max_{i=k,\ldots,n}(|A_{ki}^{(k)}|)$ (Pivotelement)
  - vertausche Spalte k und c

Lösungsaufwand $O(n^3)$

Übliche Realisierung der Gauß-Elimination durch LR-Faktorisierung

Darstellung der Matrix A durch

- obere Dreiecksmatrix $R = A^{(n)}$ und
- untere Dreiecksmatrix L mit

$$L = \begin{pmatrix} 1 & 0 & 0 & \ldots & & 0 \\ L_{21} & 1 & 0 & \ldots & & 0 \\ L_{31} & L_{32} & 1 & & & 0 \\ \vdots & & \ddots & \ddots & & \vdots \\ \vdots & & & \ddots & \ddots & 0 \\ L_{n1} & L_{n2} & \ldots & \ldots & L_{n,n-1} & 1 \end{pmatrix}$$

so dass $Ax = LA^{(n)}x = Ly = b$

Berechnungsschritte

- Bildung von $A^{(n)}$ und L
- Rücksubstitution $Ly = b$ (Lösung entspricht $b^{(n)}$)
- Rücksubstitution $A^{(n)}x = y$

Vorteil: mehrfache Verwendung der Dreiecksmatrizen für unterschiedliche Vektoren b

Parallele Implementierung

Datenverteilung beeinflusst Berechnungsschritte und Kommunikation $\Rightarrow$

blockweise Ablage von Zeilen oder Spalten ungeeignet (da Prozessor, der die ersten Zeilen speichert nur in den ersten Schritten beschäftigt ist!)

Hier betrachtet zyklische Speicherung der Zeilen

- Prozessoren $1, \ldots, q$
- Prozessor q speichert die Zeilen $q, q + p, q + 2p, \ldots$ und die zugehörigen Elemente von b

Schritte des Algorithmus

1. Bestimmung des lokalen Pivotelements
2. Bestimmung des globalen Pivotelements per Akkumulation
3. Vertauschen der Pivotzeilen
4. Verteilen der Pivotzeile per Broadcast
5. Berechnung der Eliminationsfaktoren L_{ik}
6. Berechnung der Matrixelemente in $A^{(k+1)}$ und Vektorelement in $b^{(k+1)}$

Realisierung des Programms im SPMD-Stil

4.2.2 Direkte Verfahren für Matrizen spezieller Struktur

Tridiagonale Systeme $A_{ij} = 0$ falls $|i-j| > 1$

Struktur der Matrix

$$A = \begin{pmatrix} d_1 & u_1 & & & 0 \\ l_2 & d_2 & u_2 & & \\ & \ddots & \ddots & \ddots & \\ & & l_{n-1} & d_{n-1} & u_{n-1} \\ 0 & & & l_n & d_n \end{pmatrix}$$

liefert die Gleichungen

$$\begin{aligned} d_1x_1 + u_1x_2 &= b_1 \\ l_ix_{i-1} + d_ix_i + u_ix_{i+1} &= b_i \quad (i = 2, \dots n-1) \\ l_nx_{n-1} + d_nx_n &= b_n \end{aligned}$$

Sei $x_0 = x_{n+1} = 0$, dann gilt

$x_i = \frac{1}{d_i}(b_i - l_ix_{i-1} - u_ix_{i+1})$ falls $d_i \neq 0$

Einsetzen der Gleichungen für Elemente mit ungeradem Index liefert ein Gleichungssystem der Form

$l^1_{2i}x_{2i-2} + d^1_{2i}x_{2i} + u^1_{2i}x_{2i+2} = b^1_{2i}$

für $i = 1, \dots, n/2$

$l^1_{2i} = -\frac{l_{2i}l_{2i-1}}{d_{2i-1}} \qquad u^1_{2i} = -\frac{u_{2i}u_{2i+1}}{d_{2i+1}}$

$d^1_{2i} = -\frac{u_{2i-1}l_{2i}}{d_{2i-1}} + d_{2i} - \frac{u_{2i}l_{2i+1}}{d_{2i+1}}$

$b^1_{2i} = b_{2i} - \frac{l_2b_{2i-1}}{d_{2i-1}} - \frac{u_{2i}b_{2i+1}}{d_{2i+1}}$

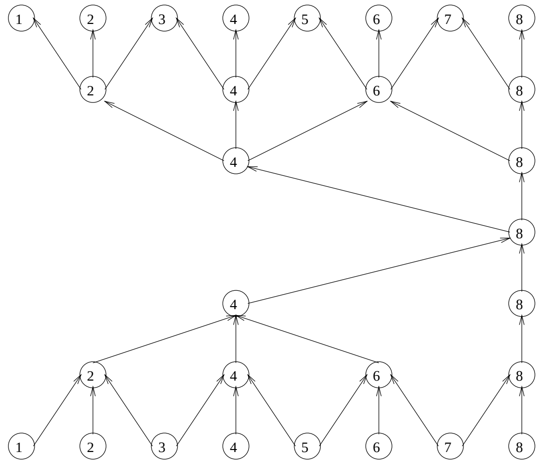
⇒ Gleichungssystem mit $\lfloor n/2 \rfloor$ Variablen und tridiagonaler Struktur

⇒ wiederholte Anwendung des Vorgehens reduziert die Variablenzahl auf 1

⇒ direkte Lösung dieses Systems

⇒ Rücksubstitution zur Berechnung aller Variablenwerte

Ablauf für $n = 8$:



Algorithmus setzt voraus, dass keine 0 in der Diagonalen auftritt!

Das gilt sicher bei diagonaldominanten und bei symmetrischen positiv definiten Matrizen

Parallelisierung

mit n Prozessoren in $\Theta(\log n)$ Schritten

Sequentieller Algorithmus benötigt $O(n)$ Schritte

⇒ Effizienz $1/\log n$

Effizienterer Algorithmus mit $O(n/\log n)$ Prozessoren in $O(\log n)$ Schritten (siehe Skalaraddition!)

Realisierung auf verschiedenen Topologien

Lineares Array mit p Prozessoren

- jeder Prozessor berechnet $\lfloor n/p \rfloor$ oder $\lceil n/p \rceil$ Variablen
- ersten $m = \lfloor n/p \rfloor$ Berechnungsschritte lokal in $O(n/p)$
- Lösung des resultierenden tridiagonalen Systems mit p Variablen auf einen Prozessor (Kommunikations- und Berechnungsaufwand jeweils $O(p)$)
- Übertragung der Resultate an alle Prozessoren per Broadcast (Aufwand $O(p)$)

Optimale Prozessorzahl $p = \Theta(\sqrt{n})$ mit Gesamtlaufzeit $O(\sqrt{n})$

⇒größere Prozessorzahl ineffizient (z.B $O(n)$ Prozessoren benötigen Laufzeit $O(n)$)

Hyperwürfel mit $O(n)$ Prozessoren

Einbettung des linearen Arrays und Verwendung des Array-Algorithmus unter Benutzung zusätzlicher Kommunikationswege

- Kommunikation findet zwischen benachbarten Prozessoren oder zwischen Prozessoren mit Adressen $j2^k$ und $(j+1)2^k$ statt
- Prozessoren $j2^k$ und $(j+1)2^k$ haben im Array Abstand 2^k, im Hyperwürfel einen Abstand von 2
- alle Prozessorpaare $(j2^k, (j+1)2^k)$ können im Hyperwürfel simultan in 2 Schritten kommunizieren
- konstante Kommunikationzeit pro Schritt
- Gesamtlaufzeit $O(\log n)$ mit n Prozessoren

Erweiterung auf blocktridiagonale Systeme

- aus x_i und b_i werden k-dimensionale Vektoren
- aus l_i, u_i und d_i werden $k \times k$ Matrizen
- Division wird durch Multiplikation mit der inversen Matrix ersetzt
- Prozessoren invertieren $k \times k$ Matrizen (Aufwand $O(k^3)$)
- Kommunikation durch Verschicken von $k \times k$ Matrizen (Aufwand $O(k^2)$)

4.2.3 Klassische Iterationsverfahren

Ziel Lösung von $Ax = b$

Sei x^* die gesuchte Lösung

Iterative Verfahren basieren auf der Zerlegung

$$A = M - N \text{ mit } M, N \in \mathbb{R}^{n\times n}$$

M ist eine nichtsinguläre Matrix,
deren Inverse leicht berechenbar ist

Umformen obiger Gleichung liefert

$$\begin{aligned} Ax &= b & \Leftrightarrow \\ Mx &= Nx + b & \Leftrightarrow \\ x &= M^{-1}Nx + M^{-1}b \end{aligned}$$

mit $C = M^{-1}N$ und $d = M^{-1}b$ erhalten wir
das Iterationsschema

$$x^{(k+1)} = Cx^{(k)} + d$$

Verfahren ist konvergent,
falls $\lim_{k\to\infty} x^{(k)} = x^*$ für alle $x^{(0)}$

Offensichtlich gilt bei konvergenten Verfahren

$$\begin{aligned} \lim_{k\to\infty} \left\|x^{(k)} - x^*\right\| &= \lim_{k\to\infty} C^k \left\|x^{(0)} - x^*\right\| &= 0 \\ \Rightarrow \qquad & \lim_{k\to\infty} C^k &= 0 \end{aligned}$$

Folgende Aussagen sind äquivalent

- Die Iteration $x^{(k+1)} = Cx^{(k)} + d$ konvergiert
- $\lim_{k\to\infty} C^k = 0$
- $\rho(C) < 1$ (Spektralradius von C)
  $\rho(C) = \max_{\lambda\in EW} |\lambda|$

Typische Iterationsverfahren

Jacobi-Verfahren

Zerlegung $A = D - L - R$

wobei D Diagonalmatrix, L untere Dreiecksmatrix und
R obere Dreiecksmatrix

Iterationsschema des Jacobi-Verfahrens

$$Dx^{(k+1)} = (L + R)x^{(k)} + b$$

oder

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

bei sequentieller Berechnung ist der Wert $x_j^{(k+1)}$ zum Zeitpunkt der Berechnung von $x_i^{(k+1)}$ $(j < i)$ bekannt

$\Rightarrow$ Ausnutzung der Kenntnis erhöht oft
Konvergenzgeschwindigkeit

Gauß-Seidel-Verfahren

Iterationsschema

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

oder

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

Konvergenzkriterium (für Jacobi- und Gauß-Seidel)

starkes Zeilensummenkriterium

$$|A_{ii}| > \sum_{i=1, i\neq j}^{n} |A_{ij}|$$

$\Rightarrow$ Jacobi und Gauß-Seidel konvergieren

Erhöhung der Konvergenzgeschwindigkeit durch Relaxation

$$\text{Setze } x^{(k+1)} = \omega x^{(k+1)} + (1 - \omega)x^{(k)}$$

"gute" Werte für ω oft nur über Heuristiken bestimmbar!

Parallele Implementierung des Jacobi-Verfahrens

Benötigte Operationen

- Matrix-Vektor-Multiplikation mit der Iterationsmatrix C
- Vektor-Addition

Spezielle Strukturen der Matrix erfordern
unterschiedliche Vorgehensweisen

Paralleles Jacobi-Verfahren

- Matrix-Vektor-Multiplikation $L + R$ mit $x^{(k)}$
- Vektor-Addition des Ergebnisses mit b
- Matrix-Vektor-Multiplikation mit D^{-1}
  (als Diagonalmatrix leicht zu invertieren)

Parallelisierung im wesentlichen durch
Parallelisierung der Vektor-Matrix-Multiplikation

Da Matrizen oft spärlich besetzt sind, sollte Struktur bei
der Parallelisierung ausgenutzt werden

Blockweise Zerlegung der Matrix A in

$$\begin{pmatrix} A_{11} & \cdots & A_{1p} \\ \vdots & \ddots & \vdots \\ A_{p1} & \cdots & A_{pp} \end{pmatrix}$$

wobei $A_{qr} \in \mathbb{R}^{k\times k}$ mit $k = n/p$

Jeder Prozessor speichert

- k Zeilen von A
- Subvektor der Länge k von x

Sei $\Omega_{in}(q) = \{r | A_{rq} \neq 0\}$ und
$\Omega_{out}(q) = \{r | A_{qr} \neq 0\}$

Mögliche Realisierung des parallelen Jacobi Verfahrens
(für Prozessor q, blockweise Speicherung der Spalten)

x_q ist der lokale Iterationsvektor

`verschicke` x_q `per Multicast an alle Prozessoren in` $\Omega_{out}(q)$ `;`
`while (!stop) {`
  $x_q = A_{qq} x_q$ `;`
  `while (noch nicht alle Vektoren` x_r
      `mit` $r \in \Omega_{in}(q)$ `empfangen) {`
    `empfange` x_r `in` y `;`
    $x_q += A_{rq} y$ `;`
  `}`
  `verschicke` x_q `per Multicast an alle`
      `Prozessoren in` $\Omega_{out}(q)$ `;`
`}`

nur drei Vektoren der Länge k werden benötigt
(beim blockierendem Senden aus x_q)

ansonsten muss Kommunikationssystem
bis zu $|\Omega_{in}(q)|$ `Vektoren zwischenspeichern`

Abbruch der Iteration

Iteration ist abzubrechen, falls $\left\| x^* - x^{(k)} \right\| < \varepsilon$

Da x^* unbekannt, kann dies nicht realisiert werden

Deshalb oft Überprüfung des relativen Fehlers
$\frac{\left\| x^{k+1)} - x^{(k)} \right\|}{\left\| x^{(k)} \right\|} < \varepsilon$
oder der Residualnorm
$\left\| A x^{(k)} - b \right\| < \varepsilon$

Gauß-Seidel-Verfahren

- Matrix-Vektor-Multiplikation R mit $x^{(k)}$
- Vektor-Addition des Ergebnisses mit b
- Lösen des Gleichungssystems mit unterer Dreiecksmatrix $(D - L)$

Parallelisierung deutlich schwieriger, da

- zur Berechnung von $x_i^{(k+1)}$ Werte $x_j^{(k+1)}$ $(j < i)$
  benötigt werden (inhärent sequentiell)

Parallelisierung durch

- lokale Berechnung von Teilsummen
  (für dicht besetzte Matrizen)
- Ausnutzung der Abhängigkeitsstruktur
  (für spärlich besetzte Matrizen)

Gauß-Seidel für dichtbesetzte Matrizen

GS-Algorithmus führt für Zeile i ein inneres Produkt mit

$$(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, 0, x_{i+1}^{(k)}, \ldots, x_n^{(k)})$$

aus

$\Rightarrow$ innere Produkte müssen nacheinander berechnet werden

$\Rightarrow$ Parallelität nur innerhalb der Berechnung der
inneren Produkte

Algorithmus mit blockweiser Verteilung der Spalten

- Annahme $k = n/p$ ganzzahlig
- Prozessor q $(0 \leq q < p)$ speichert Spalten
  $q \cdot k, \ldots, (q+1) \cdot k - 1$

Idee des Vorgehens:

- Verteilung der Berechnung der Teilsummen auf die Prozessoren
- Einzel-Akkumulation zur Berechnung eines Vektorelements (Ziel der Akkumulation ist Prozessor, der dieses Element speichert)
- Algorithmus auf folgende Folie berechnet elementweise:
  Besser, da realistischer Berechnung über Blöcke der Größe k

MPI-Realisierung (eine mögliche Variante, inkl. Konvergenztest)

```
do {
  delta_x = 0.0 ;
  for (i = 0; i < n; i++) {
    s_k = 0.0 ;
    for (j = 0; j < k; j++)
      if (j + q * k != i)
        s_k += local_A[i][j] * x[j] ;
    root = i / k ;
    i_local = i % k ;
    MPI_Reduce(&s_k, &x[i_local], 1, MPI_FLOAT,
               MPI_SUM, root, MPI_COMM_WORLD) ;
    if (q == 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) ;
```

nur sehr eingeschränkte Parallelität
durch häufige Kommunikation

Gauß-Seidel für dünnbesetzte Matrizen

bei dünnbesetzten Systemen hängt x_i nicht von x_j ab, falls $A_{ji} = 0$

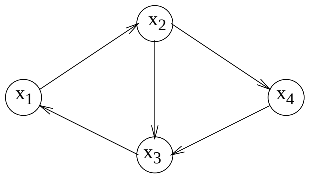
$\Rightarrow$ damit kann $x_i^{(k+1)}$ ohne Kenntnis von $x_j^{(k+1)}$ berechnet werden, auch wenn $j < i$

Darstellung der Datenabhängigkeit durch Abhängigkeitsgraph

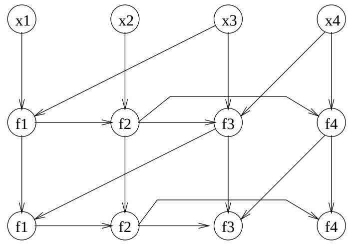
Abhängigkeitsgraph für iterative Verfahren

- ein Knoten pro Variable x_i
- eine gerichtete Kante zwischen Knoten j und Knoten i, falls x_j zur Berechnung von x_i benötigt wird (d.h. $A_{ji} \neq 0$)

Beispiel:

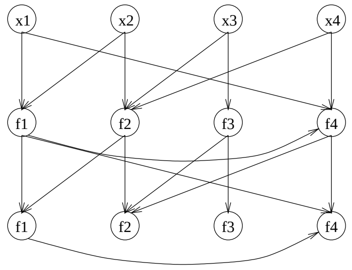


Resultierender Präzendenzgraph der ersten zwei Iterationsschritte des GS-Verfahrens



x_3 und x_4 können parallel berechnet werden!

Umnummerieren der Variablen $2 \to 4$, $4 \to 3$ und $3 \to 2$ liefert den Präzedenzgraphen



x_1, x_2 und x_3 können parallel berechnet werden

$\Rightarrow$ mögliche Parallelität (und auch Konvergenz) des GS-Verfahrens hängt von der Variableanordnung ab

Was ist eine gute Anordnung bzgl. Parallelisierung?

Rückführung auf ein Färbeproblem im Abhängigkeitsgraphen

Folgende Aussagen sind äquivalent

- Es existiert eine Variablenanordnung, mit der da GS-Verfahren in K parallelen Schritten ausgeführt werden kann
- Es existiert eine Färbung der Knoten des Abhängigkeitsgraphen mit K Farben, so dass keine Zyklen identisch gefärbter Knoten existieren

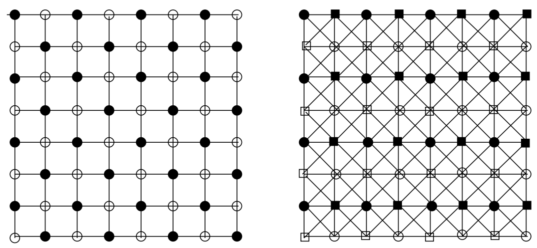
Ermittlung eines minimalen Wertes für K ist NP-vollständig!

Paralleles GS-Verfahren

- Ermittlung eines möglichst kleinen K
- Auswahl der zugehörigen Variablenordnung
  - sortiere Farben in einer beliebigen Reihenfolge
  - für alle Variablen x, y einer Farbe
    - falls Kante von y nach x existiert muss x kleineren Index als y bekommen
    - ansonsten können beide beliebig angeordnet werden
- parallele Realisierung des GS-Verfahrens: Für alle Farben
  - berechne alle Variablenwerte der aktuellen Farbe parallel
  - sende Resultate an andere Prozessoren

Oft sind dünnbesetzte Matrizen stark strukturiert

Beispiel Gitterstrukturen (bidirektionale Abhängigkeiten!)



optimale Färbung mit 2 bzw. 4 Farben

$\Rightarrow$ für diese Fälle effiziente parallele Realisierung des GS-Verfahrens

Beispiel Poisson-Gleichung

$\frac{\partial^2 f}{\partial x^2}(x,y) + \frac{\partial^2 f}{\partial y^2}(x,y) = g(x,y)$ mit $(x,y) \in [0,1]^2$

Approximation der Ableitungen

$\frac{\partial^2 f}{\partial x^2}(x,y) \approx \frac{f(x+h,y) - 2f(x,y) + f(x-h,y)}{h^2}$

$\frac{\partial^2 f}{\partial y^2}(x,y) \approx \frac{f(x,y+h) - 2f(x,y) + f(x,y-h)}{h^2}$

Diskretisierung in n äquidistante Teile liefert

$f(\frac{i}{n}, \frac{j}{n}) = f_{ij} = \frac{f_{i+1,j} + f_{i-1,j} + f_{i,j-1} + f_{i,j+1}}{4} - \frac{g_{ij}}{4n^2}$

5-Punkt-Diskretisierung

bei zusätzlichen Termen der Form $\frac{\partial^2 f}{\partial x \partial y}$ entsteht 9-Punkt-Diskretisierung

Koeffizientenmatrix nach Linearisierung zweidimensionaler Gitter

Zeilenweise Anordnung

$(0,0),(0,1),\dots,(0,n-1),(1,0),\dots(n-1,n-1)$

Nichtnullelemente in Zeile i $(0 < i < n-1)$

Bei Fünfpunktdiskretisierung:

$(i,i-n),(i,i-1),(i,i),(i,i+1),(i,i+n)$

Für $i = 1$ und $i = n$ fallen einzelne Elemente weg

Besetzungsstruktur der Matrix:

$$\left(\begin{array}{cccccccccccccccc}
* & * & & & & & * \\
& * & * & & & & & * \\
& * & * & * & & & & & * \\
& & * & * & * & & & & & * \\
& & & * & * & * & & & & & * \\
& & & & * & * & * & & & & & * \\
& & & & & * & * & * & & & & & * \\
& * & & & & & * & * & * & & & & & * \\
& & * & & & & & * & * & * & & & & & * \\
& & & * & & & & & * & * & * & & & & & * \\
& & & & * & & & & & * & * & * \\
& & & & & * & & & & & * & * & * \\
& & & & & & * & & & & & * & * & * \\
& & & & & & & * & & & & & * & * & * \\
& & & & & & & & * & & & & & * & * & * \\
& & & & & & & & & * & & & & & * & *
\end{array}\right)$$

Bei Neunpunktdiskretisierung:

$$(i,i-n-1),(i,i-n),(i,i-n+1),(i,i-1),(i,i),$$
$$(i,i+1),(i+n-1),(i,i+n),(i,i+n+1)$$

Besetzungstruktur der Matrix

$$\left(\begin{array}{cccccccccccccccc}
* & * & & & & * & * & * \\
& * & * & & & & * & * & * \\
& * & * & * & & & & * & * & * \\
& & * & * & * & & & & * & * & * \\
& & & * & * & * & & & & * & * & * \\
& & & & * & * & * & & & & * & * & * \\
& * & & & & * & * & * & & & & * & * & * \\
& * & * & & & & * & * & * & & & & * & * & * \\
& * & * & * & & & & * & * & * & & & & * & * & * \\
& & * & * & * & & & & * & * & * & & & & * & * \\
& & & * & * & * & & & & * & * & * & & & & * \\
& & & & * & * & * & & & & * & * & * \\
& & & & & * & * & * & & & & * & * & * \\
& & & & & & * & * & * & & & & * & * & * \\
& & & & & & & * & * & * & & & & * & * & * \\
& & & & & & & & * & * & * & & & & * & *
\end{array}\right)$$

Weitere Möglichkeiten der Anordnung existieren z.B.

- spaltenweise
- blockweise

4.2.4 Nichtstationäre Iterationsverfahren

Konvergenz von Jacobi- und GS-Verfahren ist oft langsam

Bestimmung von guten/optimalen Relaxationsparametern für JOR und SOR oft schwierig

$\Rightarrow$ großer Bedarf an schnellen und robusten Verfahren

Eine Klasse solcher Verfahren sind nichtstationäre Verfahren

Name resultiert daher, dass Iterationsmatrix nicht konstant bleibt

(im Gegensatz zur Iterationsmatrix stationärer Verfahren wie Jacobi, GS)

Hier einige allgemeine Ideen und Vorstellung eines speziellen Verfahrens

Im folgenden sei Matrix A

- symmetrisch d.h. $A = A^T$
- positiv definit d.h. $x^T A x > 0$ für alle $x \in \mathbb{R}^n$ $x \neq 0$

Die Lösung x^* eines linearen Gleichungssystems $Ax - b = 0$ mit symmetrischer, positive definiter Matrix A entspricht dem Minimum der Funktion

$$f(x) = \tfrac{1}{2} x^T A x - b^T x$$

$f(x)$ ist konvex und besitzt eindeutiges Minimum x^*

Idee iterativer Lösungsverfahren auf dieser Basis

bestimme ausgehend von $x^{(0)}$ neue Approximationen $x^{(k)}$ des Minimums bis $x^{(k)}$ nahe genug bei x^* liegt

Veränderung des Vektors in einem Iterationsschritt durch

$x^{(k+1)} = x^{(k)} + \alpha(k) p^{(k)}$

$p^{(k)}$ Richtungsänderung und $\alpha(k)$ Schrittweite

Bestimmung der optimalen Schrittweite bei bekanntem $p^{(k)}$

Ziel: Wähle $\alpha(k)$ so, dass

$$f(x + \alpha(k) p^{(k)}) = \min_{\alpha \in \mathbf{R}} \left(f(x^{(k)} + \alpha p^{(k)} \right)$$

Index k wird im folgenden weggelassen!

Notwendige Bedingung für das Minimum

$$\tfrac{df(x+\alpha p)}{d\alpha} = \alpha p^T A p + p^t (Ax - b) = 0$$

Bedingung auch hinreichend, da

$$\tfrac{df^2(x+\alpha p)}{d\alpha^2} = p^T A p > 0 \text{ (da } A \text{ positiv definit)}$$

Für das Minimum gilt $\alpha_{min} = \frac{-p^T r}{p^T A p}$

wobei $r = Ax - b$ Residuenvektor

$\Rightarrow$ Vermeide Wahl von p, so dass $p^T r = 0$!

Unterschiedliche Wahl von p führt zu
unterschiedlichen Iterationsverfahren

Einfachste Form: Methode des steilsten Abstiegs

setze $p^{(k)} = -r^{(k)}$

$\Rightarrow$ in der Praxis sehr schlechte Konvergenz

Besser Verfahren des konjugierten Gradienten

Zwei Vektoren $p, q \in \mathbb{R}^n$ sind bzgl. A konjugiert

$$\Rightarrow q^T A p = p^T A q = 0$$

Linear unabhängige Vektoren $p^1, \ldots, p^n$

$$\text{mit } \left(p^i\right)^T A p^j = 0 \text{ für all } i, j$$

bilden eine konjugierte Basis des $\mathbb{R}^n$ bzgl. A

> Falls die Richtungsvektoren zur Minimierung eine konjugierte Basis des $\mathbb{R}^n$ bzgl A bilden, so bestimmt ein Minimierungsverfahren mit exakter Arithmetik nach maximal n Schritten die exakte Lösung.

CG-Verfahren wählt Richtungsvektoren so, dass nach maximal n Schritten eine konjugierte Basis entsteht

Algorithmus CG-Verfahren

Initialisierung

wähle initiale Belegung von $x^{(0)}$

$p^{(0)} = -r^{(0)} = b - Ax^{(0)}$

Iteration

while ($\left\|r^{(k)}\right\| > \varepsilon$ && $k < k_{\max}$) do {

$$q^{(k)} = Ap^{(k)} \quad (1)$$
$$\alpha_k = \left(r^{(k)}\right)^T r^{(k)} / \left(\left(p^{(k)}\right)^T q^{(k)}\right) \quad (2)$$
$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)} \quad (3)$$
$$r^{(k+1)} = r^{(k)} + \alpha_k q^{(k)} \quad (4)$$
$$\beta_k = \left(r^{(k+1)}\right)^T r^{(k+1)} / \left(\left(r^{(k)}\right)^T r^{(k)}\right) \quad (5)$$
$$p^{(k+1)} = -r^{(k+1)} + \beta_k p^{(k)} \quad (6)$$
$$k = k + 1 \quad (7)$$

}

$x^{(k)}, r(k), p^{(k)} \in \mathbb{R}^n$ Vektoren des k-ten CG-Schritts

Folgende Aussagen gilt für das CG-Verfahren

- Falls $A \in \mathbb{R}^{n \times n}$ symmetrische, positiv definite Matrix und $x^{(0)}, b \in \mathbb{R}^n$
- $\exists i \leq n$, so dass $p^{(i)} = 0$ und $Ax^{(i)} = b$

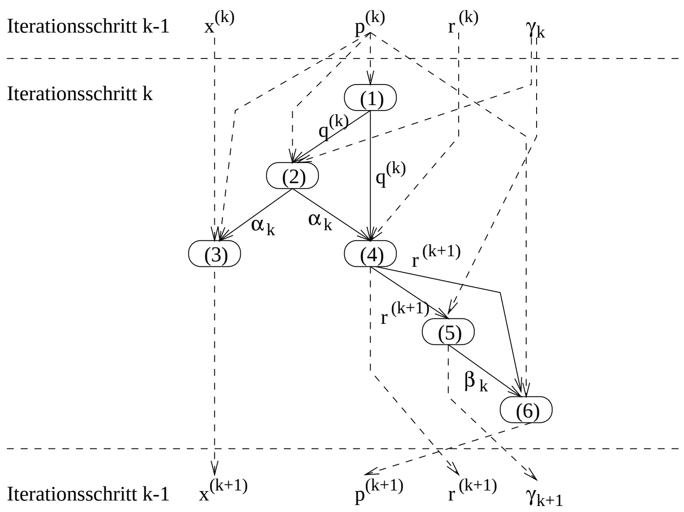
Auszuführende Berechnungsschritte

Basisoperationen in jedem Iterationsschritt

- Matrix-Vektor-Multiplikation in Schritt (1)
- zwei innere Produkt in Schritt (2),
  eines davon bereits in der vorherigen Iteration in Schritt (5) berechnet
  (zwischenspeichern von $\gamma_k = \left(r^{(k)}\right)^T r^{(k)}$)
- zwei Vektor-Additionen in den Schritten (3) und (4)
  (`saxpy`-Operationen `a x plus y`)
- zwei innere Produkte in Schritt (5),
  eines davon bereits aus der vorherigen Iteration bekannt
- eine Vektor-Addition in Schritt (6)

$\Rightarrow$ 1 Vektor-Matrix-Multiplikation, 2 innere Produkte,
3 Vektor-Additionen

Datenabhängigkeit zwischen den Iterationsschritten



Parallelisierung

Blockweise Verteilung der Zeilen von A und blockweise Verteilung der Vektorelemente von $x^{(k)}, p^{(k)}, r^{(k)}, q^{(k)}$

Schritte des parallelen Algorithmus

- Multi-Broadcast zur Verteilung $p^{(k)}$
- Parallele Matrix-Vektor-Multiplikation zur Berechnung von $q^{(k)}$
- Parallele Berechnung der inneren Produkte $\left(p^{(k)}\right)^T q^{(k)}$ und von α_k
  - jeder Prozessor berechnet lokale Teilsummen
  - Addition der Werte per Einzelakkumulation in einem Prozessor i
  - Bildung von α_k in i
  - Verteilung von α_k per Broadcast
- Lokale Berechnung von $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$ und $r^{(k+1)} = r^{(k)} + \alpha_k q^{(k)}$
- Parallele Berechnung des inneren Produkts $\left(r^{(k+1)}\right)^T r^{(k+1)}$ und von β_k
  - jeder Prozessor berechnet lokale Teilsummen
  - Addition der Werte per Einzelakkumulation in einem Prozessor i
  - Bildung von β_k in i
  - Verteilung von β_k per Broadcast
- Lokale Berechnung von $p^{(k+1)} = -r^{(k+1)} + \beta_k p^{(k)}$

Weitere nichtstationäre Iterationsverfahren

Falls A nicht symmetrisch ist, kann das CG-Verfahren auf

$$A^T A x = A^T b$$

mit identischer Lösung und

symmetrischer Koeffizientenmatrix angewendet werden

$\Rightarrow$ CGNR-Verfahren

(i.a. schlechtere Konvergenz als CG-Verfahren!)

Weitere Verfahren existieren für allgemeine Matrizen

Aber es gibt kein Verfahren für allgemeine Matrizen, welches

- eine garantierte Konvergenz in maximal n Schritten hat
- nur wenige Vektoren zur Berechnung der Resultate benötigt (kurze Rekurrenz)

4.2.5 Mehrgitterverfahren

Zur Analyse von großen Differentialgleichungssystemen werden oft Mehrgitterverfahren als effiziente Löser eingesetzt.

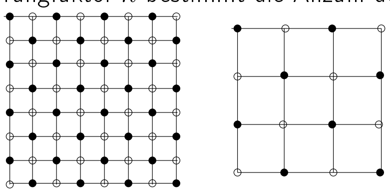
An dieser Stelle eine grobe Einführung für zweidimensionale Probleme und eine mögliche Parallelisierung

Diskretisierung führt zum Gleichungssystem $Ax = b$, welches mittels Iterationsverfahren

$x^{(k)} = Hx^{(k-1)} - b$ für eine Iterationsmatrix H gelöst wird

Falls das Verfahren konvergiert gilt $\lim_{k\to\infty} x^{(k)} = x$

Bei Differentialgleichungen in $\Omega \subset \mathbb{R}^2$entsteht A aus der Linearisierung einer zweidimensionalen Gitterstruktur, der Diskretisierungfaktor h bestimmt die Anzahl der Gitterpunkte



Unterschiedliche Werte von h liefern unterschiedliche Systeme

- Verkleinerung von h fügt zusätzliche Variablen ein und
- sorgt für eine genauere Lösung (zumindest solange bis die numerischen Ungenauigkeiten überwiegen)
- bei vielen iterativen Verfahren wächst die Iterationszahl mit der Variablenzahl

Idee der Mehrgitterverfahren

- Iteriere auf unterschiedlichen Gittern und
- transferiere die aktuellen Approximationen von feineren auf gröbere Gitter und umgekehrt

Sei

- $x^{jh,(k)}$ k-ter Iterationvektor auf dem Gitter mit Punktabstand $j \cdot h$ (h ist das feinste und $J \cdot h$ das gröbste Gitter)
- H^{jh}ist die Iterationsmatrix auf dem Gitter $j \cdot h$
- $r^{jh,(k)} = Ax^{jh,(k)} - b^{jh}$ der Fehler der k-ten Iteration auf dem Gitter jh

Betrachten wir das Verhalten des Fehlers

- der Fehler oszilliert und wird durch Iterationen geglättet
- der Fehler besteht aus Komponenten unterschiedlicher Frequenz (Abstand der Amplituden ist unterschiedlich)
- Iterationen auf einem Gitter glätten Fehlerkomponenten unterschiedlicher Frequenz

$\Rightarrow$ Verwendung unterschiedlicher Gitter glättet die verschiedenen Fehlerfrequenzen simultan und führt zu schneller Konvergenz

Vorgehen beim Mehrgitterverfahren (hier erst einmal 2 Gitter)

1. Iteriere $x^{h,(k)} = H^h x^{h,(k-1)} - b^h$ ausgehend von $x^{h,(0)}$für $k = 1, ..., K$ (i.a. recht klein $2, ..10$)
2. Berechne $r^h = Ax^{h,(K)} - b^h$
3. Transferiere r^h nach r^{2h}
4. Iteriere $x^{2h,(k)} = H^{2h} x^{2h,(k-1)} - r^{2h}$ ausgehen von z.B. $x^{2h,(0)} = 0$ für $k = 1, ..., L$
5. Transferiere $x^{2h,(L)}$ nach y^h und berechne $x^h = x^{h,(K)} + y^h$
6. Setze $x^{h,(0)} = x^h$ und iteriere $x^{h,(k)} = H^h x^{h,(k-1)} - b^h$ für $k = 1, ..., M$
7. Berechne $r^h = Ax^{h,(M)} - b^h$
8. Falls $\|r^h\| < \epsilon$ Abbruch sonst setze $x^{h,(0)} = x^{h,(M)}$ und fahre bei 1. fort

Vorgehen intuitiv einfach, transferieren von Vektoren auf gröbere/feinere Gitter ist noch zu definieren.

Pro Mehrgitteriterationsschritt

- eine Projektion von h nach $2h$
- eine Projektion von $2h$ nach h
- $K + M$ Iterationen auf Gitter h
- L Iterationen auf Gitter $2h$

Übergang vom groben auf ein feineres Gitter (Prolongation)

Wir nehmen an, dass das feinere Gitter genau die doppelte Schrittweite des groben hat.

Prolongation erfolgt per Interpolation

Wir benutzen zweidimensionale Adressierung zur Angabe der Koordinaten

Element i, j im Gitter der Schrittweite $2h$ wird auf Element $2i, 2j$ im Gitter der Schrittweite h abgebildet

d.h. Elemente mit ungeraden Indizes im feineren Gitter haben keine direkte Entsprechung im groben Gitter

$$
\begin{aligned}
v^h_{2i,2j} &= v^{2h}_{i,j} && 1 \le i,j \le \tfrac{n}{2} - 1 \\
v^h_{2i+1,2j} &= \tfrac{1}{2}\left(v^{2h}_{i,j} + v^{2h}_{i+1,j}\right) && 0 \le i \le \tfrac{n}{2} - 1, 1 \le j \le \tfrac{n}{2} - 1 \\
v^h_{2i,2j+1} &= \tfrac{1}{2}\left(v^{2h}_{i,j} + v^{2h}_{i,j+1}\right) && 1 \le i \le \tfrac{n}{2} - 1, 0 \le j \le \tfrac{n}{2} - 1 \\
v^h_{2i+1,2j+1} &= \tfrac{1}{4}\left(v^{2h}_{i,j} + v^{2h}_{i,j+1} + v^{2h}_{i+1,j} + v^{2h}_{i+1,j+1}\right) && 0 \le i,j \le \tfrac{n}{2} - 1
\end{aligned}
$$

n ist die Dimension des feineren Gitter

Realisierung Prolongation per Vektor-Matrix-Multiplikation

$v^h = I^h_{2h} v^{2h}$

Übergang vom feineren auf das grobere Gitter (Restriktion)

Unterschiedliche Alternativen

$v^{2h}_{i,j} = v^h_{2i,2j}$oder

$v^{2h}_{i,j} = \frac{1}{8}\left(4v^h_{2i,2j} + v^h_{2i-1,2j} + v^h_{2i+1,2j} + v^h_{2i,2j-1} + v^h_{2i,2j+1}\right)$

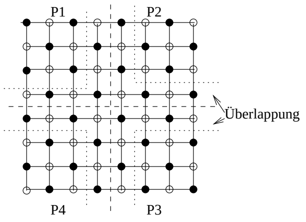
Restriktion per Vektor-Matrix-Multiplikation $v^{2h} = I^{2h}_h v^h$

Eine Version des Mehrgitterverfahrens (weitere existieren)

1. Iteriere $\nu_1$mal $x^h H^h - b^h$ausgehend von u^0.
   Dies liefert v^h.
2. for $(j = 2, \ldots, J)$ do
3. $\quad r^{jh} = I^{jh}_{(j-1)h}\left(v^{(j-1)h} H^{(j-1)h} - b^{(j-1)h}\right)$;
4. $\quad$ if $(j == J)$
   Löse $r^{jh} = x^{jh} H^{jh} - b^{jh}$. Dies liefert v^{jh}.
5. $\quad$ else
   Iteriere $\nu_1$mal $x^{jh} H^{jh} - b^{jh}$ ausgehend von $x^{jh} = 0$. Dies liefert v^{jh}.
6. for $(j = J - 1, \ldots, 1)$ do
7. $\quad$ Iteriere $\nu_2$mal $x^{jh} H^{jh} - b^{jh}$ ausgehend von $x^{jh} = v^{jh} + I^{jh}_{(j+1)h} v^{(j+1)h}$. Dies liefert eines neues v^{jh}.
8. Test Konvergenz, falls noch keine Konvergenz setze $u^0 = v^h$ und fahre bei 1. fort.

Parallelisierung von Mehrgitterverfahren

Unterteilung des Gebietes in gleich große Teile, so dass die Anzahl der Ranpunkte der Gebiete möglichst klein ist (möglichst Quadrate oder Rechtecke verwenden)



Werte der Randpunkte der Nachbargebiete werden jeweils als Kopien gespeichert und in jedem Iterationsschritt ausgetauscht

- bei Verwendung des Jacobi-Verfahrens werden
  zuerst die Randpunkte neu berechnet und verschickt,
  anschließend werden die restlichen Punkt berechnet
  nach Empfang der neuen Werte für die Randpunkte kann mit der nächsten Iteration begonnen werden
- bei Verwendung des Gauß-Seidel-Verfahrens
  werden zuerst die schwarzen Randpunkte berechnet und verschickt,
  dann die schwarzen Punkte im Inneren berechnet
  nach Empfang der schwarzen Randpunkte werden die weißen Punkte in gleicher Weise berechnet und verschickt

Prolongation und Restriktion können bei Kenntnis der Werte der Randpunkte lokal berechnet werden

Aufwandsbetrachtung für ein Gitter mit n^2 Variablen auf p Prozessoren (sei $m^2 = n^2/p$)

- Berechnungsaufwand $O(m^2)$ (konkret bei Fünfpunktdiskretisierung $9m^2 t_A$ wobei t_A die Zeit für eine Multiplikation/Addition ist)
- Kommunikationsaufwand $O(m)$ (konkret $4(t_B + mt_S)$ bei sequentiellem Senden und Übertragungszeit $t_B + mt_S$ für m Werte)

Beim Übergang auf ein gröberes Gitter werden $n^2/4$ Variablen auf p Prozessoren verteilt, wobei jeder Prozessor $m^2/4$ Werte erhält

- dadurch wird der Berechnungsaufwand um den Faktor 4 verringert
- während der Kommunikationsaufwand um einen Faktor < 2 verringert wird

$\Rightarrow$ Verhältnis Berechnungszeit zu Kommunikationszeit wird ungünstiger

4.3 Nichtlineare Gleichungssysteme

Anwendungsgebiete:

- nichtlineare Schwingungen
- Strömungen
- Produktionsplanung
- ...

Allgemeine Gestalt

$$\begin{aligned} G_1(x_1, \dots, x_n) &= b_1 \\ &\vdots \\ G_n(x_1, \dots, x_n) &= b_n \end{aligned}$$

$G_i : \mathbb{R}^n \to \mathbb{R}$ (z.B. Polynom)

Andere Schreibweise $G(x) = b$

mit $G = (G_1, \dots, G_n) : \mathbb{R}^n \to \mathbb{R}^n$ und $x, b \in \mathbb{R}^n$

Darstellungsformen

- Fixpunktgleichung $F(x) = x$
- Nullstellenbestimmung $F(x) = 0$

Lösung x^* i.d.R. nicht in geschlossener Form berechenbar $\Rightarrow$ iterative numerische Verfahren z.B. Newton Verfahren

Fixpunktiterationen

Gleichungssystem $G(x) = b$ wird in eine Fixpunktgleichung

$$x = \Phi(x)$$

mit $\Phi(x) = G(x) - b + x$ transformiert

Ein Vektor x^* heißt Fixpunkt von Φ, falls $\Phi(x^*) = x^*$
(in diesem Fall gilt auch $G(x^*) = b$)

Allgemeines Verfahren zur Lösung von Fixpunktgleichungen

Fixpunktiteration $x^{(k+1)} = \Phi(x^{(k)})$

Iteration konvergiert, falls $\lim_{k\to\infty} x^{(k)} = x^*$

Zentral für Konvergenz: Kontraktionsabbildung

Abbildung $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ erfüllt Lipschitzbedingung auf $X \subseteq \mathbb{R}^n$, falls $\|\Phi(x) - \Phi(y)\| \leq \alpha \|x - y\|$ für eine Konstante α und alle $x, y \in X$

(da alle Normen im $\mathbb{R}^n$ äquivalent sind, kann eine beliebige genommen werden)

Φ ist Kontraktionsabbildung auf X, falls Φ die Lipschitzbedingung mit Konstante $0 \leq \alpha < 1$ auf X erfüllt

Falls Φ Kontraktionsabbildung und $x^{(0)}$ "geeignet gewählt", so konvergiert die Fixpunktiteration

Für die Fehlerabschätzung gilt dann

$$\left\|x^{(k)} - x^*\right\| \leq \frac{\alpha^k}{1-\alpha} \left\|x^{(1)} - x^{(0)}\right\|$$

Spezialfall $n = 1$ ($\phi : \mathbb{R} \to \mathbb{R}$):

falls ϕ stetig differenzierbar gilt

$$|\phi(x) - \phi(y)| \leq \phi'(\xi) \cdot |x - y|$$

für ein $\xi \in (x, y)$

$\Rightarrow$ Lipschitzkonstante L in Umgebung $S_r(x^{(0)})$ entspricht

$$L = \max_{\xi \in S_r(x^{(0)})} \phi'(\xi)$$

Fixpunktiteration konvergiert $\Rightarrow$
$|\phi'(\xi)| \leq L < 1$ in $S_r(x^{(0)})$

Parallelisierung der Fixpunktiteration

ähnlich zu linearen Gleichungssystemen

Sei $k = n/p$ ganzzahlig

"Blockweise" Verteilung der Funktionen $\Phi_i(x)$

Prozessor j ($j = 0, \dots, p-1$) berechnet
Funktionen $j \cdot k + 1, \dots, (j+1) \cdot k$

Ablauf des Algorithmus

- Berechnung der Komponenten des Lösungsvektors lokal auf den Prozessoren
- Multi-Broadcast zum Austausch des Lösungsvektors

Newton Verfahren

Nullstellenbestimmung einer zweimal stetig differenzierbaren Funktion $f : \mathbb{R} \rightarrow \mathbb{R}$ mit Hilfe eines Iterationsverfahrens

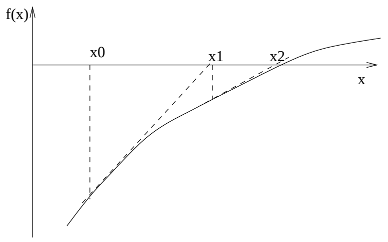
$$x^{(k+1)} = \Phi(x^{(k)})$$

Φ wird in Abhängigkeit von f konstruiert $(\Phi[f](x))$

Iterationsvorschritt des Newton-Verfahrens

$$\Phi[f](x^{(k+1)}) = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

Berechnung des Schnittpunkts der Tangente im Punkt $x^{(k)}$ mit der x-Achse



$\Rightarrow$ Linearisierung der Funktion

Anwendung auf nichtlineare Gleichungssysteme

Sei $F(x) = G(x) - b$

Taylor-Entwicklung um den Startpunkt $x^{(0)}$ liefert für die Nullstelle x^* in erster Näherung

$$0 = F(x^*) = F(x^0) + DF(x^{(0)}) \cdot (x^* - x^{(0)}) \; (*)$$

mit der Jacobimatrix

$$DF(x) = \begin{pmatrix} \frac{\partial F_1}{\partial x_1}(x) & \cdots & \frac{\partial F_1}{\partial x_n}(x) \\ \vdots & & \vdots \\ \frac{\partial F_n}{\partial x_1}(x) & \cdots & \frac{\partial F_n}{\partial x_n}(x) \end{pmatrix}$$

(*) kann als Funktion von x aufgefasst werden

$$\overline{F}(x) = F(x^{(0)}) + DF(x^{(0)}) \cdot (x - x^{(0)})$$

mit der Nullstelle

$$x^{(1)} = x^{(0)} - \left(DF(x^{(0)})\right)^{-1} \cdot F(x^{(0)})$$

falls die Matrix $DF(x^{(0)})$ invertierbar ist

Daraus kann ein Iterationsverfahren abgeleitet werden

$$x^{(k+1)} = x^{(k)} - \left(DF(x^{(k)})\right)^{-1} \cdot F(x^{(k)})$$

Inverse $\left(DF(x^{(k)}\right)^{-1}$ muss nicht explizit berechnet werden!

Iterationsvorschritt $x^{(k+1)} = x^{(k)} + w^{(k)}$

mit $w^{(k)}$ Lösung des linearen Gleichungssystems

$$DF(x^{(k)}) \cdot w^{(k)} = -F(x^{(k)})$$

Jacobi-Matrix wir meist nicht exakt bestimmt, sondern über die Approximation

$$\frac{\partial F_i}{\partial x_j} \approx \frac{F_i(x^{(k)} + r_j e_j) - F_i(x^{(k)})}{r_j} \; (**)$$

wobei e_j j-ter Einheitsvektor

r_j der Abstand zur Ermittlung der Ableitung ist

Vorschlag zur Wahl von r_j:

Wähle r_j so, dass

$$\left\| F(x^{(k)} + r_j e_j) - F(x^{(k)}) \right\| \approx \sqrt{eps} \cdot F(x^{(k)})$$

bei Maschinengenauigkeit eps

Aufwand zur Bildung der Jacobi-Matrix mittels (**)

- $n + 1$ Funktionsauswertungen pro Zeile (insgesamt $n^2 + n$)

Newton Verfahren angewendet auf Funktion F mit Lipschitzkonstante $0 \leq L < 1$ kann abgebrochen werden, falls $\left\| x^{(k+1)} - x^{(k)} \right\| \leq \varepsilon \cdot \frac{1-L}{L}$

Erweiterung zur Steigerung der Konvergenzgeschwindigkeit

gedämpftes Newton-Verfahren

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} \cdot w^{(k)}$$

mit $0 < \alpha^{(k)} \leq 1$

oft mit dynamischer Berechnung von $\alpha^{(k)}$

Sequentielle Implementierung

```
double *newton_seq (double *F(), double eps)
{
  double x_old[n], x_new[n], *temp, w[n], r[n] ;
  double f[n], df[n][n], L;
  int i, j ;
  /* initialisiere x_old, x_new, r */
  do {
   temp = x_new; x_new = x_old; x_old = temp ;
   for (i = 0; i < n; i++) {
     f[i] = F(i,x_old) ;
     for (j = 0; j<n; j++) {
       x_old[j] += r[j] ;
       df[i][j] = (F(j,x_old)-f[i]) / r[j] ;
       x_old[j] -= r[j] ;
   }}
   w = gauss_seq(df, f) ; /* Lsg. lin. Gl. */
   for (i = 0; i < n; i++)
     x_new[i] = x_old[i] + w[i] ;
  } while (norm(x_new, x_old) > eps * (1-L) / L;
  return x_new ;
}
```

Parallele Implementierung

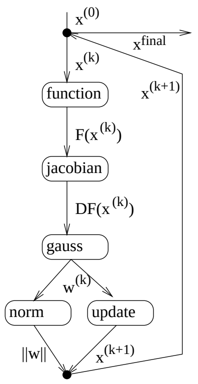
Iterationen müssen nacheinander durchgeführt werden, da Ergebnisvektor in Schritt k in Schritt $k+1$ verwendet wird

Teilaufgaben in jedem Schritt

- Berechnung der Jacobi-Matrix
- Durchführung der Gauß-Elimination
- Berechnung des nächsten Approximationsvektors
- Konvergenzkontrolle

Parallelisierung des Verfahrens $\Rightarrow$
geeignete Parallelisierung der Teilaufgaben

Datenabhängigkeiten:



Bei Parallelisierung auf geeignete Datenverteilung achten!

Gauß-Elimination benötigt den größten Teil der Laufzeit
$\Rightarrow$ Datenverteilung bzgl. Gauß-Elimination optimieren

Zeilenzyklische Verteilung: Prozessor q speichert

- Funktionen F_i mit $i = q, 2q, \ldots, kq$
- Zeilen $q, 2q, \ldots, kq$ der Jacobi-Matrix
- Vektorelemente $q, 2q, \ldots, kq$ des Vektor $x^{(k)}$

und berechnet

- Werte der gespeicherten Funktionen
- zugehörige Zeilen der Jacobi-Matrix
- zugehörige Zeilen der Dreiecksmatrizen bei der Gauß-Elimination
- zugehörige Werte in den Vektoren $w^{(k)}$ und $x^{(k+1)}$
- zugehörige Komponenten der Norm $\|w^{(k)}\|$

Programmablauf

- Berechnung der Zeilen der Jacobi-Matrix
- Durchführung der Gauß-Elimination
- Rücksubstitution
- Verteilung Lösungsteilvektoren per Broadcast

Zeilenzyklische parallele Implementierung

```
double *newton_cyclic (double *F())
{
  double x_old[n], x_new[n], *temp, w[n], r[n] ;
  double f[n], df[n][n] ;
  /* initialisiere x_old, x_new, r */
  do {
   temp = x_new; x_new = x_old; x_old = temp ;
   for (i = q; i <= k; i+=p) {
     f[i] = F(i,x_old) ;
     for (j = 0; j<n; j++) {
       x_old[j] += r[j] ;
       df[i][j] = (F(j,x_old)-f[i]) / r[j] ;
       x_old[j] -= r[j] ;
   } }
   w = gauss_cyclic(df, f) ;
   for (i = 0; i < n; i++)
     x_new[i] = x_old[i] + w[i] ;
  } while (norm(x_new, x_old) > eps * (1-L) / L;
  return x_new ;
}
```

Kommunikation findet in der `gauss_cyclic()` Funktion statt