Dr. Andreas Borchert
Prof. Dr. Stefan Funken
Prof. Dr. Karsten Urban
Markus Bantle, Kristina Steih

Institut für Numerische Mathematik
Universität Ulm
Sommersemester 2013

# High Performance Computing – Blatt 8

(Präsenzübung 17. Juni 2013)

## Teil 1: Theorie verstehen (Hausaufgabe!)

Gegeben Sei das Gebiet $\Omega \subset \mathbb{R}^2$. Wir betrachten das Poisson-Problem

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega \\ u &= u_D && \text{auf } \Gamma_D \subset \Gamma = \partial\Omega \quad \text{(Dirichlet-Randbedingung)} \\ \frac{\partial u}{\partial n} &= g && \text{auf } \Gamma_N = \Gamma \setminus \Gamma_D \quad \text{(Neumann-Randbedingung).} \end{aligned}$$

Diese Differentialgleichung soll mit der Finiten Elemente Methode gelöst werden. Nachdem wir uns letzte Woche mit der Generierung und Verwaltung von Triangulierungen beschäftigt haben, beschäftigen wir uns in dieser Übung mit der Assemblierung der Galerkin-Matrix und mit verteilten Vektoren, den `DataVectors`.
Um die obige Differentialgleichung numerisch zu lösen führen wir eine Basis des Raumes der stückweise linearen Funktionen auf $\Omega$ bzgl. einer gegebenen Triangulierung ein. Hierbei wird zu jedem Knoten des Gitters $x_i \in \overline{\Omega}$ eine Basisfunktion $\varphi_i$ eingeführt mit

$$\varphi_i(x_j) = \delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j. \end{cases},$$

welche auf jedem Dreieck linear ist.
**Aufgabe 1**
Zeichnen Sie eine Basisfunktion $\varphi_i(x)$.

Die Einträge der Galerkin-Matrix $\mathbf{A} = (a_{i,j}) \in \mathbb{R}^{N \times N}$ ($N$ ist hier die Anzahl der Basisfunktionen bzw. der Knoten) sind dann gegeben durch

$$a_{i,j} = \int_\Omega \nabla\varphi_i(x)\ \nabla\varphi_j(x)\, dx.$$

Bevor wir uns um die Assemblierung der Galerkin-Matrix kümmern, betrachten wir die Struktur der Galerkin-Matrix genauer.

**Aufgabe 2**

Erklären Sie die folgende Aussage:

*Da die Basisfunktionen $\varphi_i$ lokale Träger haben, ist die Galerkin-Matrix dünn besetzt.*

Um die Assemblierung und Speicherung der Galerkin-Matrix sowie die Matrix-Vektor-Multiplikation effizient (bzgl. Speicherbedarf und Rechenzeit) realisieren zu können, brauchen wir ein spezielles Speicherformat, das so genannte *compressed-row-storage* (CRS) Format. Eine dünsbesetzte Matrix im CRS-Format wird über die Datenstrukturen `values`, `colIndex` und `rowPtr` beschrieben (siehe Abbildung).

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 & 4 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & -3 \\ -4 & 0 & 0 & -4 & 0 \\ 0 & 3 & 0 & 2 & 0 \end{pmatrix}$$

| data | 2 | -1 | 4 | 1 | 2 | -3 | -4 | -4 | 3 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| colIndex | 0 | 1 | 4 | 2 | 2 | 4 | 0 | 3 | 1 | 3 |
| rowPtr | 0 | 3 | 4 | 6 | 8 | 10 | | | | |

**Aufgabe 3**

Erklären Sie was die Vektoren `values`, `colIndex` und `rowPtr` enthalten. Wie groß ist der Speicherbedarf einer Matrix mit $n$ Nicht-Null-Einträgen?

Für die Assemblierung der Galerkin-Matrix gehen wir wieder (wie in 1D) elementweise vor. Das heißt, wir assemblieren für jedes Element eine $3 \times 3$ Element-Steifigkeits-Matrix, die die Beiträge aller 9 Kombinationen der drei Basis-Funktionen, deren Träger dieses Element einschließen, enthält. Anders als im 1D-Projekt werden jetzt die lokalen Beiträge jedoch nicht sofort auf die globale Matrix addiert, sondern wir speichern die Zeilen- und Spalten-Indizes der Stellen, an denen wir die Werte addieren würden, sowie die entsprechenden Werte in drei Vektoren. Nachdem die Werte für alle Elemente in den drei Vektoren gesammelt sind, werden diese Vektoren verwendet um ein CRS-Matrix zu initialisieren.
Der nächste Schritt ist die Assemblierung der rechten Seite des Gleichungssystems. Diese ist gegeben durch

$$\begin{aligned} b &= b^{(1)} + b^{(2)} - b^{(3)}, \\ b_j &= \int_\Omega f(x)\varphi_j(x)\,dx + \int_{\Gamma_N} g(x)\varphi_j(x)\,ds_x - \int_\Omega \nabla\widetilde{u}_D \nabla\varphi_j(x)\,dx =: b_j^{(1)} + b_j^{(2)} - b_j^{(3)}, \end{aligned}$$

wobei $\widetilde{u}_D$ eine Fortsetzung der Dirichlet-Daten in Omega ist (die Fortsetzung wird so gewählt, dass $\widetilde{u}_D(x_i) = 0$ für alle inneren Knoten des Gitters gilt).
Der Anteil von $b_j^{(1)}$ bzgl. eines Dreiecks $T$ (das im Träger von $\varphi_j$ liegt) wird approximiert durch

$$\int_T f(x)\varphi_j(x)\,dx \approx \frac{|T|}{3} f(x_S),$$

wobei $x_S$ der Schwerpunkt von $T$ ist. Für ein Neumann-Randstück $S_N$ (das an den Träger von $\varphi_j$ angrenzt) wird das Integral $b_j^{(2)}$ approximiert durch

$$\int_{S_N} g(x)\varphi_j(x)\,ds_x \approx \frac{|S_N|}{2} g(x_M),$$

wobei $x_M$ der Mittelpunkt von $S_N$ ist. Um den Vektor $b^{(3)}$ zu berechnen, initialisieren wir eine Hilfsvektor $d$ mit $d(j) = u_D(x_j)$, wenn $x_j$ ein Dirichlet Knoten ist, und $d(j) = 0$ wenn $x_j$ kein Dirichlet Knoten ist. Der Vektor $b^{(3)}$ ist dann gegeben durch $b^{(3)} = A \cdot d$.
Wir können nun also die Galerkin Matrix $A$ und die rechte Seite $b$ assemblieren. Für die parallele FEM ist das Gitter aufgeteilt, und jeder Prozess assembliert nur einen Teil von

$A$ bzw. $b$. Jeder Prozess hat also eine lokale Steiffigkeitsmatrix $A_{loc}$ und eine lokale rechte Seite $b_{loc}$.

### Aufgabe 3

Erklären Sie (evtl anhand einer Grafik) wieso die Einträge von $A_{loc}$ bzw. $b_{loc}$ nicht immer mit den Einträgen der globalen Steifigkeitsmatrix $A$ bzw. der globalen rechten Seite $b$ überein stimmen.

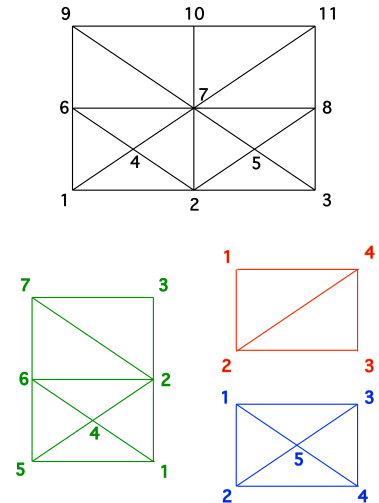
Für die parallele FEM arbeiten wir mit verschiedenen Vektoren, bei denen jeder Eintrag einem Gitterpunkt entspricht (z.B. die Vektoren $u$ und $b$, die die Lösung bzw. die rechte Seite des linearen Gleichungssystems enthalten). Da das Gitter auf verschiedene Prozessoren verteilt ist, sind auch die Vektoren verteilt. Man unterscheidet zwei Typen von verteilten Vektoren:

- typeI: der Vektor enthält an den Kopplungsknoten und cross points den gesamten Wert (global).
- typeII: der Vektor enthält an den Kopplungsknoten und cross points nur einen Teil des gesamten Wertes (lokal).

Hier ein einfaches Beispiel:

Zum globalen Gitter (rechts, Schwarz) ist der Vektor $u = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$ gegeben. Das Gitter (und somit auch der Vektor $u$) wird nun auf drei Prozesse aufgeteilt (grün, rot und blau). Die lokalen Vektoren (typeII) lauten dann

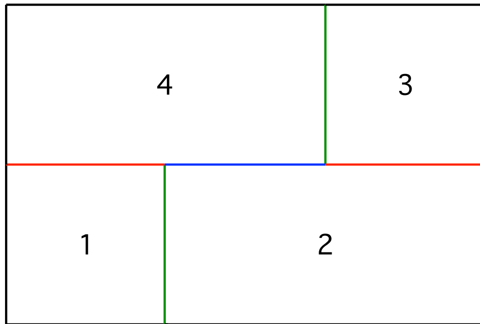
$$u^{(1)} = (0.5, 0.25, 0.5, 1, 1, 1, 1)$$
$$u^{(2)} = (0.5, 0.25, 1, 1)$$
$$u^{(3)} = (0.25, 0.5, 1, 1, 1).$$




Wären die Vektoren vom typeI, dann wären alle Einträge, genauso wie im globalen Vektor, eins.
Die Konvertierung zwischen den Typen wird wie folgt gemacht:

> typeII $\rightarrow$ typeI: Für alle Kopplungsknoten und alle cross points werden die Werte der jeweils umliegenden Prozesse aufaddiert (Kommunikation nötig).
>
> typeII $\rightarrow$ typeI: Der Wert an einem Kopplungsknoten wird halbiert, der Wert an jedem cross point wird durch die Anzahl der jeweils anliegenden Prozesse geteilt (keine Kommunikation nötig).

Für die Kommiunikation der Kopplungsknoten muss darauf geachtet werden, dass geordnet kommuniziert wird, damit sich die Prozesse nicht blockieren. Deshalb bekommt jeder Kopplungsrand eine Farbe zu geweisen. Die Kommunikation für Kopplungsränder der gleichen Farbe kann dann gleichzeitig ablaufen. Um dies zu verstehen betrachten wir folgendes Beispiel:




Hier sind die Farben so verteilt, dass

- jeder Prozess immer nur mit genau einem anderen Kommuniziert,
- so wenig Kommunikation wie möglich stattfindet,
- es keine Deadlocks gibt.

**Aufgabe 4**

Welche MPI Befehle wird für die Kommunikation in der Umwandlung typeII $\rightarrow$ typeI benötigt?

## Teil 2: Gegebenen Code verstehen (Hausaufgabe!)

Laden Sie den Code zu diesem Übungsblatt von der Homepage herunter.
In unserem FEM-Softwarepaket werden verteilte Vektoren mit der Klasse `DataVector` verwaltet. Ein `DataVector` besteht aus einer Länge (int), einem Datenfeld (double*) und einem Typ (enum: typeI, typeII oder nonMPI). Zudem bietet ein `DataVector` Funktionen für die Typ-Konvertierung, die Änderung der Größe und für die Ausgabe der Daten in eine Datei. Funktionen für Operationen mit `DataVector`en (Skalarprodukt, Norm, Matrix-Vektor-Produkt) sind in den Dateien `MathOperationsMPI.*` deklariert und implementiert.

**Aufgabe 5**

Gehen Sie durch den Code in den Dateien `DataVector.*` und `MathOperationsMPI.*` und schauen Sie wo etwas implementiert werden muss.

**Aufgabe 6**

Die zentrale Datenstruktur des FEM-Projekts ist die Klasse FEM.
Lesen Sie den Code in der Datei `Fem.hpp` und erstellen Sie eine Liste der Variablen und Funktionen der Klasse FEM. Verschaffen Sie sich eine Überblick über den Code in der Datei `Fem.cpp` und schauen Sie wo etwas implementiert werden muss.

## Teil 3: Programmieren (In der Übung)

Zunächst beschäftigen wir uns mit der Assemblierung der Galerkin-Matrix und der rechten Seite des linearen Gleichungssystems.

(1) Vervollständigen Sie die Funktion `assemble` (`Fem.cpp`, Zeile 41) und die Datei `main-serial.cpp`.

(2) Kompilieren Sie alles und starten Sie `./main-serial ./input/Rectangle`
Vergleichen Sie die erzeugte rechte Seite (./output/b_serial.txt) mit der Musterlösung (./output/b_solution.txt).
Betrachten Sie die Einträge der Matrix $A$ (in der Datei ./output/A_serial.txt). Versuchen Sie das zu Grunde liegende Gitter (Grafik unten) mit den Matrix-Einträgen in Verbindung zu bringen.

Nun wenden wir uns der Implementierung von `DataVectors` zu.

(3) Vervollständigen Sie die Funktionen (in der Datei `DataVector.cpp`)

- `communicationCrossPoints`
- `communicationBoundaryNodes`
- `typeII_2_typeI`
- `typeI_2_typeII`

Bearbeiten Sie die Funktionen in dieser Reihenfolge.

In der Datei `main-parallel.cpp` ist nun zunächst Folgendes implementiert:
Prozess 0 liest das Gitter ein und schickt die entsprechenden Daten an die anderen Prozesse (wie letzte Woche). Jeder Prozess legt dann ein lokales Mesh Objekt und ein lokales FEM Objekt an. Danach assembliert jeder Prozess seine lokale Steifigkeitsmatrix A und gibt diese in eine Datei aus.

(4) Kompilieren und starten Sie das Programm `main-parallel`.
Die lokalen Matrizen werden in die Dateien ./output/A_0_local.txt und ./output/A_1_local.txt ausgegeben. Versuchen Sie die Werte in den Matrizen mit den beiden lokalen Gittern (Grafiken unten) in Verbindung zu bringen.

Schließlich möchten wir für Prozess 0 den Teil der globalen Steifigkeitsmatrix $A$ rekonstruieren, der den Knoten von Prozess 0 entspricht. Hierzu multiplizieren wir die Matrix $A$ nach und nach mit Einheitsvektoren (Teil II im Code in der Datei `main-parallel.cpp`). Hierbei muss beachtet werden dass gilt:

$$\text{Lokale Matrix} \cdot \text{typeI-Vektor} = \text{typeII-Vektor}.$$

(5) Vervollständigen Sie Teil II in der Datei `main-parallel.cpp`

(6) Vervollständigen Sie die Funktion `dot` in der Datei `MathOperationsMPI.cpp` und testen Sie die Funktion indem Sie den dritten Teil in der Datei `main-parallel.cpp` einkommentieren.