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 7

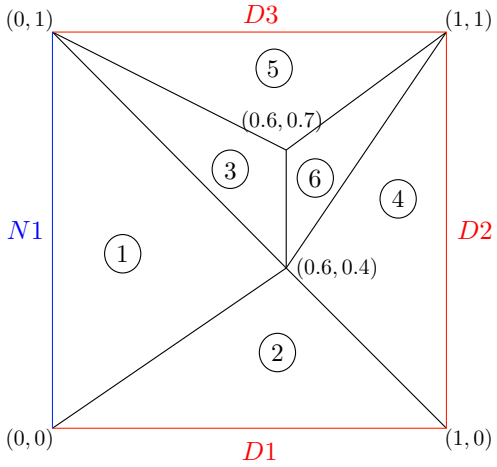
(Präsenzübung 10. Juni 2013)

Teil 1: Theorie verstehen (Hausaufgabe!)

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

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

Um diese partielle Differentialgleichung mit der Finite Elemente Methode numerisch lösen zu können, muss ein Gitter aus Dreiecken, eine sogenannte Triangulierung, erzeugt werden, welches Omega überdeckt. Die dafür benötigten Datenstrukturen sollen anhand von folgendem Beispiel eingeführt werden. Die Abbildung zeigt eine Triangulierung von $\Omega = [0,1]^2$.



Um diese Triangulierung zu beschreiben, werden zwei Matrizen `coordinates` und `elements` benötigt. Die Matrix `coordinates` enthält die Koordinaten der Eckpunkte der Dreiecke und in der Matrix `elements` stehen die Nummern der Eckpunkte, die das jeweilige Dreieck begrenzen. Für unser Beispiel sehen die Matrizen wie folgt aus:

$$coordinates = \begin{pmatrix} 0.0 & 0.0 \\ 1.0 & 0.0 \\ 0.6 & 0.4 \\ 0.6 & 0.7 \\ 0.0 & 1.0 \\ 1.0 & 1.0 \end{pmatrix} \qquad elements = \begin{pmatrix} 1 & 3 & 5 \\ 1 & 2 & 3 \\ 3 & 4 & 5 \\ 2 & 6 & 3 \\ 4 & 6 & 5 \\ 3 & 6 & 4 \end{pmatrix}$$

Um die Randbedingungen numerisch verarbeiten zu können brauchen wir zwei weitere Datenstrukturen. Am unteren, rechten und oberen Rand des Quadrats geben wir Dirichlet Randbedingungnen vor (rot), links Neumann Randbedingungen (blau). Entsprechend benötigen wir zwei Matrizen `dirichlet` und `neumann`, welche die Kanten des jeweiligen Randes enthalten. In unserem Fall sind die Matrizen gegeben durch

$$dirichlet = \begin{pmatrix} 1 & 2 \\ 2 & 6 \\ 6 & 5 \end{pmatrix} \quad neumann = \begin{pmatrix} 5 & 1 \end{pmatrix}.$$

Um Speicher zu sparen werden die Randdaten am Computer nicht als $n \times 2$ Matrizen gespeichert, sondern als Vektoren. Für obiges Beispiel werden die Vektoren

$$dirichlet = \begin{pmatrix} 1 & 2 & 6 & 5 \end{pmatrix} \quad neumann = \begin{pmatrix} 5 & 1 \end{pmatrix}$$

gespeichert. Dies funktioniert natürlich nur für zusammenhängende Randstücke. Um dies zu verdeutlichen betrachten wir ein zweites Beispiel. Gegeben seien die Matrizen

$$coordinates = \begin{pmatrix} 0.0 & 0.0 \\ 1.0 & 0.0 \\ 1.0 & 1.0 \\ 0.0 & 1.0 \\ 0.5 & 0.0 \\ 1.0 & 0.5 \\ 0.5 & 1.0 \\ 0.0 & 0.5 \end{pmatrix} \quad dirichlet = \begin{pmatrix} 1 & 5 \\ 5 & 2 \\ 3 & 7 \\ 7 & 4 \end{pmatrix} \quad neumann = \begin{pmatrix} 2 & 6 \\ 6 & 3 \\ 4 & 8 \\ 8 & 1 \end{pmatrix}.$$

Machen Sie sich die Struktur des Randes mit Hilfe einer Zeichnung klar. Um nun den Dirichlet Rand zu speichern verwenden wir die Datenstruktur

```
std::vector<IndexVector> dirichlet(2);
```

das heißt einen `vector` bei dem jeder Eintrag ein `IndexVector` ist. In unserem Beispiel enthält `dirichlet[0]` den `IndexVector` $\begin{pmatrix} 1 & 5 & 2 \end{pmatrix}$ und `dirichlet[1]` den `IndexVector` $\begin{pmatrix} 3 & 7 & 4 \end{pmatrix}$.

Aufgabe 1

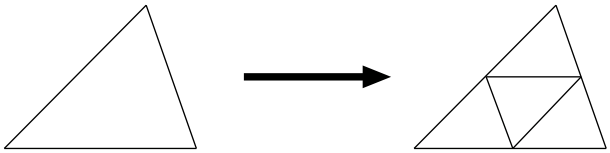
Machen Sie sich an Hand von obigem Beispiel klar, wie die Datenstrukturen `coordinates`, `elements`, `dirichlet` und `neumann` aufgebaut sind. Zeichnen Sie das Gitter für

$$coordinates = \begin{pmatrix} 0.0 & 0.0 \\ 1.0 & 0.0 \\ 0.7 & 0.7 \\ 0.0 & 1.0 \\ 1.0 & 1.0 \end{pmatrix} \qquad elements = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 5 \\ 2 & 3 & 4 \\ 5 & 3 & 4 \end{pmatrix}$$

$$dirichlet = \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix} \quad neumann = \begin{pmatrix} 2 & 4 \\ 5 & 1 \end{pmatrix}.$$

und schreiben Sie `dirichlet` und `neumann` als `vector` von `IndexVector`en.

Der nächste Schritt ist die Gitter Verfeinerung. Hier verwenden wir eine uniforme Verfeinerung (d.h. jedes Dreieck wird gleich verfeinert), die so genannte Rot-Verfeinerung. Hierbei wird jedes Dreieck in 4 kleinere Dreiecke zerlegt:



Um ein ganzes Gitter zu verfeinern geht man wie folgt vor:

- Berechne für jede Kante des existierenden Gitters den Mittelpunkt und hänge die Mittelpunkte unten an `coordinates` an.
- Durchlaufe alle Elemente (d.h. Dreiecke) des groben Gitters und generiere für jedes Element 4 neue Dreiecke.
- Durchlaufe alle Dirichlet (bzw. Neumann) Kanten des groben Gitters und generiere für jede Kante zwei neue Kanten.

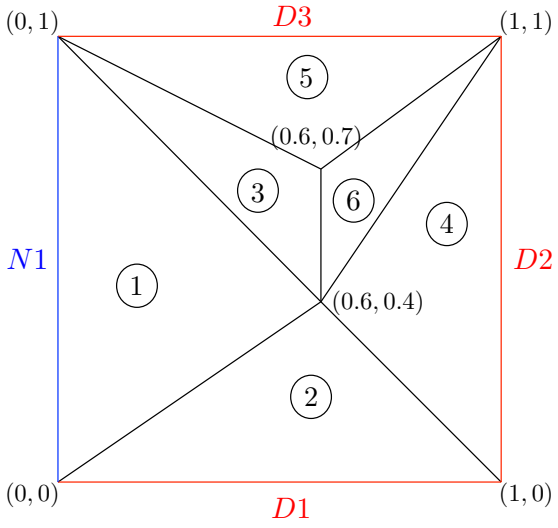
Um dies algorithmisch realisieren zu können muss man sich merken an welche Stelle in `coordinates` man die neu erzeugten Punkte (Mittelpunkte der alten Kanten) schreibt. Hierfür nummerieren wir die Kanten (edges) und führen wir die Datenstrukturen `edge2nodes` und `element2edges` ein. Die k-te Zeile der $n_e \times 2$ Matrix `edge2nodes` enthält die Nummern der beiden Knoten, die zu der k-ten Kante gehören. Die k-te Zeile der $n \times 3$ Matrix `element2edges` enthält die Nummern der drei Kanten, die zum k-ten Element gehören.

Aufgabe 2

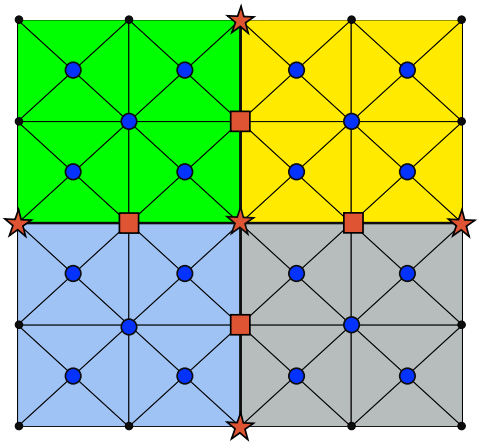
Führen Sie im Beispiel rechts Kantennummern ein (beliebig) und schreiben Sie die Matrizen `edge2nodes` und `element2edges` explizit auf. (Die Matrizen `coordinates` und `elements` sind die von der ersten Seite)

Aufgabe 3

Zeichnen Sie für das Gitter aus Aufgabe 2 das Gitter, das durch Rot-Verfeinerung entsteht.



Bei der Parallelisierung durch Gebietszerlegung wird das Gebiet Ω (und im Diskreten die Triangulierung) auf die einzelnen Prozesse aufgeteilt. Jeder Prozessor erhält nur das Gitter seines Teilgebietes und führt eine lokale Nummerierung ein. Um das globale Gitter auf die einzelnen Prozesse aufzuteilen, wird der Vektor `elements2procs`$\in \mathbb{R}^n$ benötigt. Die Datenstruktur gibt an, welches Element zu welchem Prozessor gehört (`elements2procs`(j)=4 bedeutet, dass Element j zu Prozessor 4 gehört). Außerdem ergeben sich durch die Gebietszerlegung besondere Kanten und Punkte, die wir im Folgenden genauer beschreiben (siehe Abbildung).



Gebietsaufteilung bei 4 Prozessen mit cross points (Sterne), Kopplungsknoten (Quadrate), inneren Knoten (blaue Kreise) und Randknoten (schwarz).

Die Ränder zwischen zwei Teilgebieten bezeichnen wir als Kopplungsränder (in unserem Beispiel hat jeder Prozessor 2 Koplungsränder), die Eckknoten eines Kopplungsrandes bezeichnen wir als cross points (Sterne) und die Knoten zwischen zwei cross points heißen Kopplungsknoten (Quadrate). Die anderen Knoten, d.h. die Knoten, die auf keinem Kopplungsrand liegen, nennen wir innere Knoten. Durch diese Definition ergibt sich sofort, dass Kopplungsknoten immer in den Teilgebieten von zwei Prozessen enthalten sind, cross points können in beliebig vielen Teilgebieten enthalten sein. Außerdem können cross points auch Randknoten sein, d.h. es können an diesen Knoten auch Randbedingungen gegeben sein.
Neben der Datenstruktur `elements2procs` benötigt unser paralleler Algorithmus zusätzlich die Datenstruktur `skeleton`, die die Kopplungskanten beschreibt. `skeleton` ist eine Matrix mit fünf Spalten. Dabei schreibt jede Zeile einen Kopplungsrand. In den ersten beiden Spalten sind die Eckpunkte des Kopplungsrandes definiert (Index des Eckpunktes in `coordinates` **im mathematisch positiven Sinne**), Spalte drei und vier enthalten die Indizes der Prozesse der Kopplungskante, **erst der linke Prozess, dann der Rechte** (eine Kopplungskante gehört immer zu genau zwei Prozessen). Die fünfte Spalte gibt jedem Kopplungsrand eine Farbe. Was genau die Farbe bedeutet und wozu man sie braucht wird später noch ausführlich beschrieben.
`elements2procs` und `skeleton` sind, ebenso wie `coordinates`, `elements`, `dirichlet` und `neumann` als Text-Dateien gespeichert und müssen vom Programm eingelesen werden.
Um Kopplungsränder im Program zu verwalten führen wir die Klasse `Coupling` ein. Ein Coupling Objekt beschreibt **alle** Kopplungsränder eines Prozesses und hat die Variablen

- `IndexVector neighbourProcs`: Enthält die Nummern der Nachbarprozesse
- `std::vector<IndexVector> boundaryNodes`: Ein `IndexVector` für jeden Kopplungsrand. Jeder `IndexVector` enthält die lokale Nummer der Kopplungsknoten. (Kante wird im Mathematisch positiven Sinn durchlaufen)
- `std::vector<IndexVector> coupling2edges`: Ein `IndexVector` für jeden Kopplungsrand. Jeder `IndexVector` enthält die lokale Nummer einer Kopplungskante.
- `IndexVector local2globalCrossPoints`: `local2globalCrossPoints(k) = j` heißt, dass der lokal k-te cross point den globalen Index j hat.
- `IndexVector crossPointsBdryData`: Gibt an, ob ein cross point ein Dirichlet Knoten ist (1) oder nicht (0).
- `IndexVector crossPointsNumProcs`: Gibt an, wie viele Prozessoren an einem cross point liegen.

Teil 2: Gegebenen Code verstehen (Hausaufgabe!)

Laden Sie den Code zu diesem Übungsblatt von der Homepage herunter und bearbeiten Sie folgende Aufgaben:

1. Öffnen Sie die Dateien `Matrix.hpp` und `Vector.hpp` und machen Sie sich klar, welche Variablen und Methoden die Klassen `Matrix`, `IndexMatrix`, `Vector` und `IndexVector` besitzen.
   - Wie wird auf einen Vektor bzw. eine Matrix zugegriffen?
   - Wie erhält bzw. ändert man die Größe eine Matrix bzw. die Länge eines Vektors?
   - Wie wird das Matrix-Vektor Produkt bzw. das Skalarprodukt zweier Vektoren berechnet?
2. Öffnen Sie die Datei `mesh.hpp` und machen Sie sich klar, welche Variablen und Methoden die Klasse Mesh besitzt.
3. Öffnen Sie die Dateien `mesh.cpp`, `main-serial.cpp` und `main-parallel.cpp` und schauen Sie, wo etwas programmiert werden muss. Lesen Sie die Anweisungen im Code.

Teil 3: Programmieren (In der Übung)

Bearbeiten Sie die folgenden Aufgaben:

1. Lesen Sie den Konstruktor der Klasse `Mesh` (`mesh.cpp` Zeile 35) und versuchen Sie nachzuvollziehen was gemacht wird.
2. Vervollständigen Sie die Funktion `refineRed` in der Datei `mesh.cpp` (Zeile 413). Folgen Sie den Anweisungen im Code.
3. Vervollständigen Sie die Hauptprogramme `main-serial.cpp`. Folgen Sie den Anweisungen im Code. Starten Sie das Programm `main-serial`. Das Programm gibt das verfeinerte Gitter in Text-Dateien im Ordner examples/output aus. Verwenden Sie das MATLAB Skript `run_me_triangulation.m` um das verfeinerte Gitter zu zeichnen.
4. Vervollständigen Sie die Hauptprogramme `main-parallel.cpp`. Folgen Sie den Anweisungen im Code. Starten Sie das Programm `main-parallel`. Das Programm gibt das verfeinerte Gitter in Text-Dateien im Ordner examples/output aus. Verwenden Sie das MATLAB Skript `run_me_triangulation.m` um das verfeinerte Gitter zu zeichnen.