Aufgabe 9 Grobgitterkorrektur

In der Vorlesung haben Sie bereits die Grobgitterkorrektur kennengelernt. Diese werden wir jetzt variationell formulieren. Sei $\mathcal{T}^H$ eine Triangulierung des Gebietes Ω und $\mathcal{T}^h$ eine Verfeinerung von $\mathcal{T}^H$. Die zugehörigen Finite-Elemente-Basen bezeichnen wir mit

$$\Phi^H = \{\varphi_i^H \mid i \in \mathcal{I}^H\},$$
$$\Phi^h = \{\varphi_i^h \mid i \in \mathcal{I}^h\}$$

und die Finite-Elemente-Räume mit

$$V^H = \operatorname{span} \Phi^H,$$
$$V^h = \operatorname{span} \Phi^h.$$

Wie definieren nun die Restriktionsabbildung $R^H\colon V^h \to V^H$, die bezüglich der oben angegebenen Basen durch die Matrix

$$R_{ij}^H = \varphi_i^H(x_j), \quad i \in \mathcal{I}^H, \quad j \in \mathcal{I}^h$$

beschrieben ist. Dabei sind die x_j die Knotenpositionen mit $\varphi_i^h(x_j) = \delta_{ij}$.

Sei nun ein $u_h^{(k+\frac{1}{2})}$ gegeben. Die Grobgitterkorrektur $w_H^{(k+\frac{1}{2})} \in V^H$ wird dann durch die Variationsgleichung

$$a(u_h^{(k+\frac{1}{2})} + w_H^{(k+\frac{1}{2})}, v) = l(v) \qquad \forall v \in V^H$$

charakterisiert. Der Startwert für die folgende Iteration ist dann

$$u_h^{(k+1)} = u_h^{(k+\frac{1}{2})} + w_H^{(k+\frac{1}{2})}.$$

a) Zeigen Sie, dass die Gleichung

$$\varphi_i^H = \sum_{j \in \mathcal{I}^h} R_{ij}^H \varphi_j^h$$

gilt.

b) Zeigen Sie die Beziehung

$$R^H A^h (R^H)^T = A^H,$$

wobei

$$A^h = a(\varphi_j^h, \varphi_i^h),$$
$$A^H = a(\varphi_j^H, \varphi_i^H).$$

c) Leiten Sie aus der Variationsformulierung die in der Vorlesung angegebene algebraische Schreibweise der Grobgitterkorrektur her.

8 Punkte

Aufgabe 10 Kommunikation von Knotendatenkorrekturen

In parallelen iterativen Lösern ist es häufig nötig, die Defektkorrektur $w^{(k)}$, die von den einzelnen Prozessoren berechnet wurde, auf die Näherungslösung $u^{(k)}$ zu addieren. Bei der Verwendung von P^1 finiten Elementen sind die Unbekannten $u_i^{(k)}$ den Knoten des Finite-Elemente-Gitters zugeordnet, und jeder Prozessor kennt nur einen Teil der Unbekannten und einen Teil des Gitters. Es gibt jedoch Überlappungsbereiche mit Knoten, die bei mehreren Prozessoren gespeichert sind. Für diese Knoten muss entschieden werden, welcher Prozessor die Defektkorrektur dazuaddiert. Wie sich die dazu nötige Kommunikation in DUNE implementieren lässt, ist Gegenstand dieser Aufgabe. Voraussetzung ist die Kenntnis des Kapitels „Parallelism“ aus dem DUNE Grid-Howto, dessen relevante Teile in der letzten Übung vorgestellt wurden. Da wir noch keinen vollständigen parallelen Löser implementieren werden, verwenden wir für $u^{(k)}$ und $w^{(k)}$ einfach Vektoren von Zufallszahlen.

Zunächst müssen sich die Prozessoren einigen, wer eigentlich für welchen Knoten zuständig ist. Für die inneren Knoten (Partitionstyp `InteriorEntity`) ist das klar, da jeder Knoten auf höchstens einem Prozessor innerer Knoten ist. Für die Randknoten (Partitionstyp `BorderEntity`) gibt es aber jeweils mehrere mögliche Kandidaten. Daher legt jeder Prozessor zunächst einen `std::vector<bool> vertexmask` an, in dem letztendlich stehen soll, ob er für den Knoten mit dem Index i zuständig ist oder nicht (`true` = zuständig, `false` = nicht zuständig). Dieser Vektor wird initialisiert, indem jeder Prozessor einmal über seinen Teil des Gitters iteriert und für alle inneren und Randknoten

```
vertexmask[mapper.map(*it)] = true;
```

setzt, für alle anderen `false`. Der Mapper ist dabei am besten vom Typ

```
Dune::LeafMultipleCodimMultipleGeomTypeMapper<Grid,P1Layout>
```

(Siehe das DUNE Grid-Howto für weitere Erklärungen un die Definition von `P1Layout`.)

Als nächstes müssen sich die Prozessoren einigen, wer für die Randknoten zuständig ist. Wir legen jetzt einfach fest, dass für jeden Knoten mit mehreren Kandidaten derjenige mit dem kleinsten Rank (`grid.comm().rank()`) zuständig ist. Wir müssen jetzt ein Data-Handle `OwnerExchange` schreiben, das in der `gather()`-Methode den Rank des lokalen Prozessors verteilt und in der `scatter()`-Methode die Einträge in der `vertexmask` passend aktualisiert. Danach können wir mit dem Aufruf

```
grid.template communicate<OwnerHandle>(handle,
    Dune::InteriorBorder_InteriorBorder_Interface, Dune::ForwardCommunication)
```

die eindeutige Zuständigkeitsverteilung erzeugen.

Wenn die Zuständigkeiten entschieden sind, generiert jeder Prozessor für seine Knoten ein zufälliges Update. Für Knoten, für die er nicht zuständig ist, kann er das Update einfach auf 0 setzen. Dieses Update muss über einen weiteren Aufruf von communicate übertragen werden, wiederum mit einem passenden Data-Handle. *12 Punkte*