Parallelisierung eines adaptiven Verfahrens zur numerischen Lösung partieller Differentialgleichungen

von

Sven Groß

Matrikelnummer 211 054

Diplomarbeit in Mathematik

vorgelegt der

FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND NATURWISSENSCHAFTEN

der Rheinisch-Westfälischen Technischen Hochschule Aachen

im September 2002

angefertigt im

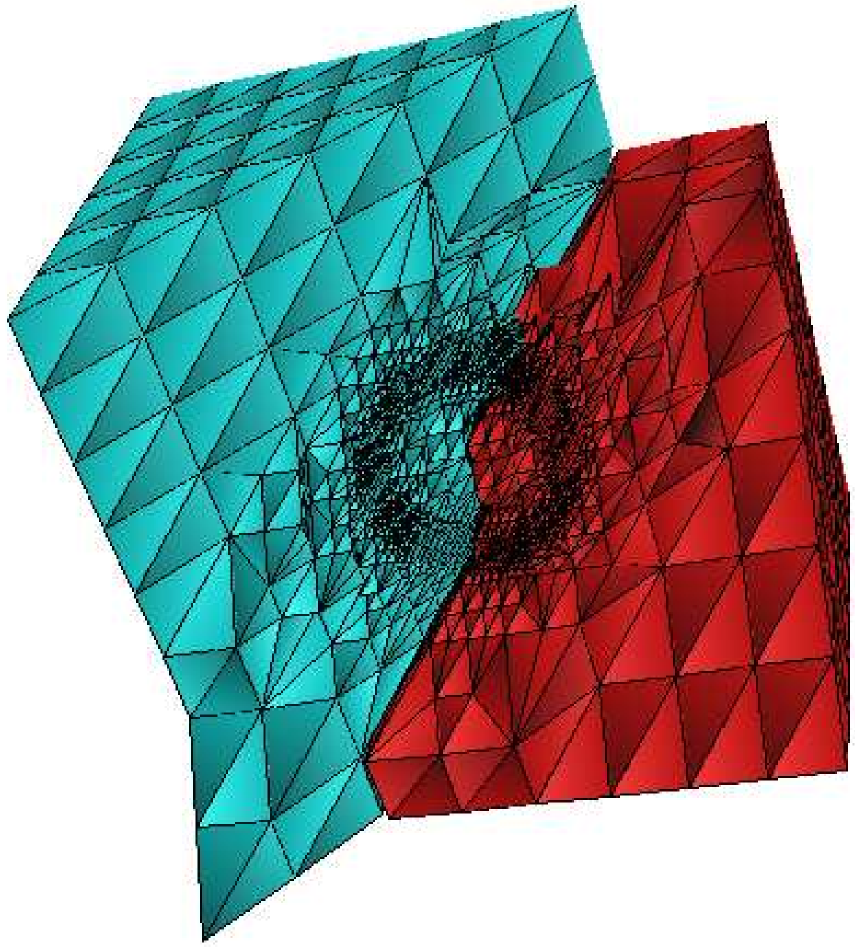
Institut für Geometrie und Praktische Mathematik (IGPM),
Lehrstuhl für Numerische Mathematik

Prof. Dr. Arnold Reusken

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtliche oder sinngemäße Wiedergaben aus anderen Quellen sind kenntlich gemacht und durch Zitate belegt.

Mönchengladbach, den 23. September 2002

Sven Groß



Vorwort

Bei der Diskretisierung von partiellen Differentialgleichungen mit Finite-Elemente-Verfahren entstehen große dünnbesetzte Gleichungssysteme, die bei sehr großen Unbekanntenzahlen unweigerlich zu einem Speicherproblem führen. Die Anzahl der Elemente in einem regelmäßigen dreidimensionalen Tetraedergitter wächst z.B. mit dem Faktor 8, wenn die Länge der Elementseiten halbiert wird, weil man eine genauere Approximation wünscht. Bei solch einem Ansatz ist die Güte der Lösung letztlich durch die Speicherausstattung des verwendeten Rechners beschränkt.

Ein Weg, dies zu umgehen, besteht darin, das Wachstum der Unbekanntenzahlen massiv einzuschränken, indem man nur soviele Unbekannte hinzunimmt, wie benötigt werden, um die Approximationsgüte in gewünschter Weise zu verbessern. D.h. das Gitter wird nur lokal an solchen Stellen verfeinert, wo die Lösung noch zu ungenau ist (*Adaptivität*). Ein anderer Ansatz besteht darin, den zur Verfügung stehenden Speicher zu vergrößern, indem man das Problem auf mehrere vernetzte Rechner aufteilt (*Parallelisierung*).

In der vorliegenden Arbeit werden beide Ideen miteinander verknüpft und am Beispiel der Poissongleichung ein paralleles adaptives Verfahren entwickelt. Dazu wurde der Verfeinerungsalgorithmus von Bey parallelisiert und das für adaptive parallele Verfahren auftretende Problem der Lastbalancierung behandelt. Als Ausgangspunkt diente dabei das Programmpaket DROPS, das zur Zeit am Institut für Geometrie und Praktische Mathematik der RWTH Aachen im Rahmen des Sonderforschungsbereichs 540 entwickelt wird und zur Simulation von reaktiven mehrphasigen Strömungen eingesetzt werden soll.

In Kapitel 1 wird aufbauend auf dem Konzept der schwachen Lösungen das Finite-Elemente-Verfahren hergeleitet und die Poissongleichung entsprechend diskretisiert. Darüber hinaus werden einige theoretische Resultate (Existenz und Eindeutigkeit der schwachen Lösung, Fehlerabschätzungen usw.) vorgestellt.

Kapitel 2 beschäftigt sich mit ersten Aspekten der Parallelisierung, und die nachfolgenden Kapitel bauen teilweise darauf auf. Dabei werden zunächst die grundlegenden Konzepte und Ideen vorgestellt, bevor die Parallelisierung des CG-Algorithmus aus der verteilten Speicherung der Matrizen und Vektoren hergeleitet wird. Das Kapitel endet mit einem längeren Abschnitt über DDD, einer Programmbibliothek zur konsistenten Verwaltung von verteilt gespeicherten Daten, auf der ein Großteil der Parallelisierung in DROPS basiert. Es wird gezeigt, dass das DDD-Konzept der verteilten Objekte den verwalteten Datengraph korrekt auf die Prozessoren abbildet (Satz 2.10).

Kapitel 3 und 4 nehmen in dieser Arbeit eine zentrale Rolle ein, da sie die Parallelisierung des Verfeinerungsalgorithmus vorstellen. Kapitel 3 beginnt mit einer Einführung in die Idee der Adaptivität. Basierend auf [2] werden anschließend die Rot-Grün-Verfeinerung von Triangulierungen erläutert und die verwandten Begriffe „Triangulierungshierarchie“ und „Multilevelgitter“ eingeführt. Das Kapitel endet mit der Präsentation der ersten Version des parallelen Verfeinerungsalgorithmus; die Wirkung der einzelnen darin enthaltenen Programmfunktionen wird detailliert erläutert.

Die erste Version des Algorithmus arbeitet nur auf verteilten Triangulierungshierarchien, die keine überlappt gespeicherten Tetraeder enthalten. In Kapitel 4 wird allerdings dargelegt, dass eine sinnvolle Lastverteilung eine Überlappung von Tetraedern erfordert. Deswegen wird eine geeignete Form der verteilten Speicherung des Multilevelgitters hergeleitet und in einer formalen Spezifikation genau beschrieben. Ausgehend davon wird die endgültige Version des parallelen Verfeinerungsalgorithmus entwickelt und bewiesen, dass dieser den Anforderungen der formalen Spezifikation genügt (Satz 4.8).

Die Herleitung des parallelen Verfeinerungsalgorithmus in Kapitel 3 und 4 erfolgt – im Sinne einer leichteren Verständlichkeit, und um eine größere Transparenz in Bezug auf die Funktion der einzelnen Kommunikationsaufrufe zu schaffen – in zwei Stufen: Die erste Version des Algorithmus in Kapitel 3 ist noch eng an den von Bey in [2] vorgestellten seriellen Verfeinerungsalgorithmus angelehnt und soll den Leser mit den wenigen zur Parallelisierung nötigen Kommunikationsaufrufen vertraut machen. Die zweite Version des Algorithmus in Kapitel 4 erfordert demgegenüber einige grundlegende Änderungen und Neuerungen, da sie auf einer allgemeineren und komplizierteren verteilten Speicherung der Tetraeder beruht.

Kapitel 5 ist dem Thema der Lastverteilung gewidmet, welches für die Effizienz paralleler Algorithmen gerade im Kontext von Adaptivität (und damit sich dynamisch verändernder Datenmengen) eine zentrale Rolle spielt. Zunächst wird das Lastverteilungsproblem auf das verallgemeinerte Graphpartitionierungsproblem zurückgeführt, anschließend werden einige Verfahren zur Graphpartitionierung vorgestellt. Schließlich wird die in DROPS realisierte Form der Lastverteilung beschrieben und im Rahmen dessen das dabei entwickelte Konzept des reduzierten dualen Graphen erläutert. Es wird gezeigt, dass dieses Vorgehen verteilte Multilevelgitter erzeugt, die zulässig sind, d.h. die Anforderungen der formalen Spezifikation aus Kapitel 4 erfüllen.

Kapitel 6 stellt die Ergebnisse einiger paralleler numerischer Experimente vor. Dabei werden einige parallelisierte Komponenten von DROPS wie der parallele Verfeinerungsalgorithmus und der parallele CG-Algorithmus auf ihre Skalierbarkeit hin untersucht. In diesem Rahmen werden die Aussagekraft von Speedup- und Effizienzmessungen kritisch diskutiert und deren Möglichkeiten und Grenzen aufgezeigt. In einem weiteren Experiment wird das Zusammenspiel der einzelnen Komponenten in einem parallelen Lauf eines adaptiven Zyklus getestet, dessen Resultate abschließend präsentiert werden.

Inhaltsverzeichnis

Kapitel 1

Die Poissongleichung und ihre Diskretisierung

1.1 Die Poissongleichung

Sei $\Omega \subset \mathbb{R}^n$ ein beschränktes Gebiet mit polygonalem Lipschitzrand $\Sigma = \partial\Omega$. Dann heißt zu vorgegebenem $f \in C^0(\overline{\Omega})$

$$-\Delta u = f \qquad \text{in } \Omega$$

die *Poissongleichung*. Physikalisch wird damit z.B. ein elektrisches Potential u beschrieben, wobei f der Quellterm (eine äußere Kraft, in diesem Fall die Ladungsdichte) ist. Um die Eindeutigkeit einer evtl. existierenden Lösung zu erhalten, muss man eine zusätzliche Randbedingung stellen:

$$u = g \qquad \text{auf } \Sigma\ ,$$

eine sogenannte *Dirichlet-Randbedingung*.

$u \in C^2(\Omega) \cap C^0(\overline{\Omega})$ heißt klassische Lösung der Randwertaufgabe (RWA)

$$\left\{ \begin{array}{rcll} -\Delta u &=& f & \text{in } \Omega \\ u &=& g & \text{auf } \Sigma\ , \end{array} \right.$$

falls u diese Gleichung punktweise erfüllt.

Falls eine klassische Lösung existiert, ist diese eindeutig: Die Differenz zweier Lösungen $d := u - \tilde{u}$ ist harmonisch und erfüllt deswegen das Maximumprinzip

$$\sup_{\overline{\Omega}} d = \sup_{\Sigma} d\ .$$

Wegen $d_{|\Sigma} = g - g = 0$ folgt $d \leq 0$. Analog schließt man für $-d$ und erhält somit schließlich $d = 0$, also die Eindeutigkeit $u = \tilde{u}$.

Man kann auch andere Randbedingungen wie die *Neumann-Randbedingung*

$$\frac{\partial u}{\partial n} = h \qquad \text{auf } \Sigma$$

vorgeben, allerdings ist dann eine Lösung nur noch eindeutig bis auf eine Konstante. Auch gemischte Randwerte sind möglich: Dazu wird Σ in den Dirichletrand Σ_D und den Neumannrand Σ_N aufgeteilt ($\Sigma = \Sigma_D \cup \Sigma_N$) und entsprechende Randwerte dort vorgegeben; Randpunkte, an denen Σ_N und Σ_D zusammentreffen, werden als Dirichletrandpunkte behandelt.

Die Existenz klassischer Lösungen ist dagegen eine problematische Angelegenheit: Für jede klassische Lösung u ist zwar $\Delta u \in C^0(\overline{\Omega})$, umgekehrt gibt es aber Funktionen $f \in C^0(\overline{\Omega})$, für die auch bei glattem Rand und glatten Randdaten keine Lösung $u \in C^2(\Omega) \cap C^0(\overline{\Omega})$ existiert. Die Räume $C^k(\Omega)$ sind also ungeeignet zur Suche von Lösungen elliptischer Gleichungen. Erst das Zulassen sogenannter *schwacher* Lösungen ermöglicht es, bereits Existenz für $f \in L_2(\Omega)$ zu erzielen. Diese sind Thema des folgenden Abschnitts und bilden die Grundlage für die Finite-Elemente-Methode.

1.2 Variationsformulierung und schwache Lösungen

Für Lösungen elliptischer Gleichungen scheint der Raum der klassischen Lösungen nicht geeignet zu sein; auf der Suche nach angemessenen Funktionenräumen stößt man neben den Räumen $C^{k,\alpha}(\Omega)$ der hölderstetigen Funktionen auf die Sobolevräume, welche für den Finite-Elemente-Ansatz den geeigneten Rahmen darstellen. Zunächst einige Definitionen:

$$L_2(\Omega) := \{u : \Omega \to \mathbb{R} : |u|^2 \text{ ist Lebesgue-integrabel und } \int_\Omega |u|^2 \, dx < \infty\}$$

bildet mit dem L_2-Skalarprodukt

$$(u, v)_{L_2(\Omega)} := (u, v)_0 := \int_\Omega u \cdot v \, dx$$

und der davon abgeleiteten Norm $||u||_{L_2(\Omega)} = \sqrt{(u, u)_0}$ einen Hilbertraum.

$$C_0^\infty(\Omega) := \{u \in C^\infty(\Omega) : \operatorname{supp}(u) \subset \Omega\}$$

liegt dicht in $L_2(\Omega)$ und wird als „Testfunktionenraum“ benutzt. Beachte, dass der *Träger* einer Funktion $u : \Omega \to \mathbb{R}$

$$\mathrm{supp}(u) := \overline{\{x \in \Omega : u(x) \neq 0\}}$$

per Definition eine *abgeschlossene* Menge ist.

Für einen Multiindex $\alpha \in \mathbb{N}_0^n$ ($\mathbb{N}_0 = \mathbb{N} \cup \{0\}$) sei D^α der partielle Ableitungsoperator $D^\alpha = \frac{\partial^{\alpha_1}}{\partial x_1^{\alpha_1}} \cdots \frac{\partial^{\alpha_n}}{\partial x_n^{\alpha_n}}$.

Definition 1.1 (schwache Ableitung) $u \in L_2(\Omega)$ besitzt eine schwache D^α-Ableitung $v \in L_2(\Omega)$, falls

$$(w, v)_0 = (-1)^{|\alpha|}(D^\alpha w, u)_0 \qquad \forall\, w \in C_0^\infty(\Omega)$$

gilt mit $|\alpha| = \sum_i \alpha_i$. In diesem Fall schreibt man $D^\alpha u = v$.

Besitzt u eine starke D^α-Ableitung, so stimmt diese fast überall mit der schwachen überein, was man leicht mittels partieller Integration nachvollziehen kann.

Definition 1.2 (Sobolev-Raum) Für $k \in \mathbb{N}_0$ sei $H^k(\Omega) \subset L_2(\Omega)$ die Menge aller Funktionen u, die schwache Ableitungen $D^\alpha u \in L_2(\Omega)$ für $|\alpha| \leq k$ besitzen:

$$H^k(\Omega) := \{u \in L_2(\Omega) : D^\alpha u \in L_2(\Omega) \quad \forall\, |\alpha| \leq k\}\ .$$

$H^k(\Omega)$ bildet mit dem Skalarprodukt

$$(u, v)_{H^k(\Omega)} := (u, v)_k := \sum_{|\alpha \leq k} (D^\alpha u, D^\alpha v)_{L_2(\Omega)}$$

und der davon induzierten Norm $||u||_{H^k(\Omega)} := \sqrt{(u,u)_k}$ einen Hilbertraum, dieser heißt *Sobolev-Raum.* Die Vervollständigung von $C_0^\infty(\Omega)$ in $L_2(\Omega)$ unter der Sobolev-Norm $||\cdot||_{H^k(\Omega)}$ wird mit $H_0^k(\Omega)$ bezeichnet.

Um nun die Poissongleichung mit homogener Dirichlet-Randbedingung

$$\left\{ \begin{array}{rcll} -\Delta u & = & f & \text{in } \Omega \\ u & = & 0 & \text{auf } \Sigma \end{array} \right. \tag{1.1}$$

in eine Variationsformulierung umzuschreiben, wird diese mit einer Testfunktion $\phi \in H_0^1(\Omega)$ multipliziert; anschließend werden beide Seiten über Ω integriert:

$$-\int_\Omega \Delta u \cdot \phi \, dx = \int_\Omega f \cdot \phi \, dx\ .$$

Die linke Seite wird partiell integriert:

$$-\int_\Omega \Delta u \cdot \phi \, dx = \int_\Omega \nabla u \cdot \nabla \phi \, dx$$

(das Randintegral verschwindet wegen $\phi|_\Sigma \equiv 0$), und man erhält

$$\int_\Omega \nabla u \cdot \nabla \phi \, dx = \int_\Omega f \cdot \phi \, dx \ .$$

Beachte, dass u jetzt nur noch schwache Ableitungen erster Ordnung besitzen muss; der geeignete Sobolevraum, in dem die Lösung gesucht werden sollte, ist also $V := H_0^1(\Omega)$. Mithilfe der Bilinearform

$$a(u,v) := \int_\Omega \nabla u \cdot \nabla \phi \, dx$$

kann nun das Variationsproblem formuliert werden:

$$\begin{cases} \text{Finde } u \in H_0^1(\Omega), \text{ so dass} \\ a(u,\phi) = (f,\phi)_0 \qquad \forall\, \phi \in H_0^1(\Omega) \text{ gilt.} \end{cases} \tag{1.2}$$

Falls eine klassische Lösung der RWA (1.1) existiert, so stimmt diese mit der zugehörigen schwachen Lösung von (1.2) fast überall überein (partielle Integration!). Die klassischen Lösungen sind also in diesem Sinne in das allgemeinere Konzept der schwachen Lösungen eingebettet. Umgekehrt ist klar, dass es in vielen Fällen eine schwache Lösung geben kann, in denen eine klassische Lösung nicht existiert. In der Tat kann man zeigen, dass für jedes $f \in L_2(\Omega)$ bereits eine eindeutige schwache Lösung existiert. Entscheidend dafür ist, dass $a(\cdot,\cdot)$ eine beschränkte elliptische Bilinearform ist, diese Eigenschaften erbt sie von dem elliptischen Differentialoperator $L = -\Delta$.

Satz 1.3 ($H_0^1(\Omega)$-Elliptizität und Beschränktheit von $a(\cdot,\cdot)$) *Die Bilinearform $a(\cdot,\cdot)$ ist symmetrisch sowie $H_0^1(\Omega)$-elliptisch und beschränkt, d.h.*

$$a(u,u) \geq \alpha ||u||_1^2 \qquad \forall\, u \in H_0^1(\Omega)$$

und

$$a(u,v) \leq M ||u||_1 ||v||_1 \qquad \forall\, u,v \in H_0^1(\Omega) \ .$$

Beweis: Die Symmetrie von $a(\cdot,\cdot)$ ist klar. Aus der Cauchy-Schwarz-Ungleichung und der Definition der Sobolev-Norm $||\cdot||_1$ folgt für alle $u,v \in H_0^1(\Omega)$

$$\begin{aligned} a(u,v) = (\nabla u, \nabla v)_0 &\leq ||\nabla u||_0 \, ||\nabla v||_0 \\ &\leq ||u||_1 \, ||v||_1 \ , \end{aligned}$$

also die Beschränktheit von $a(\cdot,\cdot)$. Zum Nachweis der Elliptizität sei $u \in H_0^1(\Omega)$. Aus der Friedrichs-Ungleichung folgt dann

$$||u||_1^2 = ||u||_0^2 + |u|_1^2 \leq (C+1)\ |u|_1^2 = (C+1)\ a(u,u)$$

mit $|u|_1^2 = \sum_{|\alpha|=1} ||D^\alpha u||_0^2$. □

Der Schlüssel zum Beweis der Existenz schwacher Lösungen liegt im folgenden Lemma von Lax und Milgram:

Satz 1.4 (Lax-Milgram-Lemma) *Sei H ein Hilbertraum und $a(\cdot,\cdot) : H \times H \to \mathbb{R}$ eine H-elliptische beschränkte Bilinearform. Dann existiert zu jedem beschränkten linearen Funktional $F \in H^*$ ein eindeutig bestimmtes $x \in H$ mit*

$$a(x,y) = F(y) \qquad \forall\, y \in H\ .$$

Dies führt geradewegs zu folgendem

Satz 1.5 (Existenz und Eindeutigkeit schwacher Lösungen) *Für jede rechte Seite $f \in L_2(\Omega)$ besitzt das Variationsproblem (1.2) eine eindeutige schwache Lösung $u \in H_0^1(\Omega)$.*

Beweis: Nach Satz 1.3 ist $a(\cdot,\cdot)$ eine symmetrische, beschränkte, $H_0^1(\Omega)$-elliptische Bilinearform. Weiterhin ist $F(v) := (f,v)_0$, $v \in H_0^1(\Omega)$ für jedes $f \in L_2(\Omega)$ ein beschränktes lineares Funktional auf $H_0^1(\Omega)$, denn

$$(f,v)_0 \leq ||f||_0\, ||v||_0 \leq ||f||_0\, ||v||_1 \qquad \forall\, v \in H_0^1(\Omega)\ .$$

Damit sind die Voraussetzungen des Lax-Milgram-Lemmas 1.4 erfüllt, also existiert zu jedem $f \in L_2(\Omega)$ ein $u \in H_0^1(\Omega)$ mit

$$a(u,v) = F(v) \qquad \forall\, v \in H_0^1(\Omega)\ .$$

□

1.3 Diskretisierung

Das unendlich-dimensionale Variationsproblem (1.2) wird durch eine Galerkin-Approximation in ein endlich-dimensionales Problem überführt, indem man einen endlich-dimensionalen Ansatzraum $V_h \subset V$ wählt: Dazu wird eine Triangulierung $\mathcal{T}$ des polyedrisch berandeten Gebietes $\Omega \subset \mathbb{R}^n$ gewählt; diese Arbeit ist beschränkt auf dreidimensionale Gebiete $\Omega \subset \mathbb{R}^3$ sowie reine Tetraederzerlegungen ohne hängende Knoten, sogenannte *konsistente Triangulierungen*.

Definition 1.6 (Triangulierung) Eine endliche Menge $\mathcal{T}$ von Tetraedern $T \in \overline{\Omega}$ heißt *Triangulierung* von Ω, wenn folgende Bedingungen erfüllt sind:

1. $|T| > 0 \qquad$ für alle $T \in \mathcal{T}$,
2. $\bigcup_{T\in\mathcal{T}} T = \overline{\Omega}$,
3. $\overset{\circ}{S} \cap \overset{\circ}{T} = \emptyset \qquad$ für alle $S, T \in \mathcal{T}$ mit $S \neq T$.

Dabei bezeichne $\overset{\circ}{M}$ das Innere einer Menge $M \subset \mathbb{R}^3$.

Eine Triangulierung heißt *konsistent*, wenn der Schnitt zweier Tetraeder $S, T \in \mathcal{T}$ entweder leer ist oder jeweils mit einem *Subsimplex* (d.h. Knoten, Kante oder Seitenfläche) von S und T übereinstimmt.

Für eine Seitenfläche F eines Tetraeders $T \in \mathcal{T}$, die auf dem Rand $\partial\Omega$ liegt, gelte zusätzlich $F \subset \Sigma_N$ oder $F \subset \Sigma_D$.

Zusätzlich wird ein Finite-Elemente-Typ gewählt, in diesem Fall P_1-Elemente, die stückweise linear auf den Tetraedern sind. Triangulierung und FE-Typ legen nun den Ansatzraum fest:

$$V_h := \mathcal{P}_{1,D}(\mathcal{T}) := \{\phi \in V : \phi|_T \in \mathcal{P}_1(T) \quad \forall T \in \mathcal{T}\},$$

wobei $\mathcal{P}_1(S)$ den Raum der linearen Polynome auf $S \subset \mathbb{R}^3$ bezeichne.

Zur Wahl einer Basis von V_h seien die Freiheitsgrade der $T \in \mathcal{T}$ auf den Knoten so durchnummeriert, dass zuerst Knoten im Inneren, dann solche auf Neumannrändern und zuletzt solche auf Dirichleträndern auftreten. Weiter bezeichne $x_i \in \mathbb{R}^3$ die Koordinate des Knotens, auf dem Freiheitsgrad i liegt. Definiere nun für $M \subset \overline{\Omega}$ folgende Indexmenge:

$$I(M) := \{i : x_i \in M\}\ .$$

Die Hutfunktion $\phi_i : \overline{\Omega} \to \mathbb{R}$ sei nun durch

$$\phi_i|_T \in \mathcal{P}_1(T) \qquad \forall T \in \mathcal{T}$$

und

$$\phi_i(x_j) = \delta_{ij} \qquad \forall j \in I(\overline{\Omega}) \tag{1.3}$$

definiert. Es sei bemerkt, dass die Menge aller Hutfunktionen $\{\phi_i : i \in I(\overline{\Omega})\}$ aufgrund der Eigenschaft (1.3) linear unabhängig ist.

1.3.1 Homogene Dirichletrandwerte

Zunächst wird der einfachste Fall, das Problem mit homogenen Dirichletrandwerten betrachtet, d.h. $u = 0$ auf $\Sigma_D = \partial\Omega$, also $V = H_0^1(\Omega)$. Das endlichdimensionale Galerkin-Problem lautet nun:

$$\begin{cases} \text{Finde } u_h \in V_h \text{, so dass} \\ a(u_h, \phi) = (f, \phi)_0 \qquad \forall\, \phi \in V_h \text{ gilt.} \end{cases} \tag{1.4}$$

Da eine lineare Funktion $f(x, y, z) = ax + by + cz + d$ auf einem Tetraeder durch Vorgabe von Werten auf seinen vier Knoten bereits eindeutig bestimmt ist, gilt $\dim V_h = |I(\Omega)|$ sowie

$$V_h = \langle \phi_i \mid i \in I(\Omega) \rangle \ .$$

Somit ist $\{\phi_i : i \in I(\Omega)\}$ eine Basis von V_h (die sogenannte Knotenbasis), welche der Umwandlung von (1.4) in ein lineares Gleichungssystem von $|I(\Omega)|$ Gleichungen und Unbekannten dient:

$$\begin{cases} \text{Finde } \underline{u} \in \mathbb{R}^{|I(\Omega)|} \text{, so dass} \\ a(\sum_{j \in I(\Omega)} \underline{u}_j \phi_j, \phi) = (f, \phi)_0 \qquad \forall\, \phi \in V_h \text{ gilt.} \end{cases} \tag{1.5}$$

Dies ist äquivalent zur Matrixnotation

$$A\underline{u} = \underline{b}$$

mit

$$\begin{aligned} A_{ij} &= a(\phi_j, \phi_i) \ , \\ \underline{b}_i &= (f, \phi_i)_0, \qquad i, j \in I(\Omega) \ . \end{aligned}$$

1.3.2 Inhomogene Dirichlet- und Neumannrandwerte

Soweit zur Diskretisierung eines Poissonproblems mit homogenen Dirichletrandwerten.

Ein Poissonproblem mit inhomogenen Dirichlet- und Neumannrandwerten

$$\begin{cases} -\Delta u &= f & \text{in } \Omega \\ u &= g_D & \text{auf } \Sigma_D \\ \frac{\partial u}{\partial n} &= g_N & \text{auf } \Sigma_N \end{cases} \tag{1.6}$$

wird abgesehen von kleinen Änderungen nach demselben Prinzip diskretisiert wie zuvor das Poissonproblem mit homogenen Randwerten. Im Endeffekt werden sich dabei die inhomogenen Randwerte in der rechten Seite $\underline{b}$ des linearen Gleichungssystems wiederfinden.

Durch Multiplikation von (1.6) mit einer Funktion $\phi \in H^1(\Omega)$ und partielle Integration erhält man unter Beachtung der Neumann-Randbedingung

$$\int_\Omega \nabla u \cdot \nabla \phi \, dx - \oint_{\Sigma_D} \frac{\partial u}{\partial n} \cdot \phi \, d\sigma - \oint_{\Sigma_N} g_N \cdot \phi \, d\sigma = \int_\Omega f \cdot \phi \, dx \ .$$

Wählt man nun als Testfunktionenraum

$$V_N := \{\phi \in H^1(\Omega) : \phi_{|\Sigma_D} = 0\} \ ,$$

so ergibt sich das Variationsproblem

$$\begin{cases} \text{Finde } u \in H^1(\Omega), \quad u_{|\Sigma_D} = g_D, \text{ so dass} \\ a(u,\phi) = (f,\phi)_0 + \oint_{\Sigma_N} g_N \cdot \phi \, d\sigma \qquad \forall \, \phi \in V_N \text{ gilt.} \end{cases} \tag{1.7}$$

Die Neumann-Randbedingung findet sich also direkt in der Variationsgleichung wieder, wogegen die Dirichlet-Randbedingung explizit im Ansatz gefordert werden muss. Deswegen wird eine Dirichlet-Randbedingung auch als *wesentliche Randbedingung* bezeichnet; Neumann-Randbedingungen zählen zu den *natürlichen Randbedingungen.*

Die Diskretisierung erfolgt wiederum über die Wahl eines endlichdimensionalen Ansatzraumes

$$V_{h,N} := \langle \phi_j \mid j \in I(\Omega \cup \Sigma_N) \rangle \ .$$

Beachte die Konformität des Ansatzes: $V_{h,N} \subset V_N$. Da die Menge der Hutfunktionen $\{\phi_i : i \in I(\overline{\Omega})\}$ linear unabhängig ist, ist die Dimension von $V_{h,N}$ gerade $n := \dim V_{h,N} = |I(\Omega \cup \Sigma_N)|$, und die Menge

$$\mathcal{B} := \{\phi_i : i \in I(\Omega \cup \Sigma_N)\}$$

ist eine Basis von $V_{h,N}$. Die Unbekannten befinden sich also auf den inneren Knoten sowie auf den Knoten des Neumannrandes; auf dem Dirichletrand Σ_D sind die Werte von u ja bereits bekannt.

Beachte für $j \in I(\Sigma_N)$, also eine Unbekannte auf dem Neumannrand, die spezielle partielle Integration

$$\begin{aligned} \int_\Omega (-\Delta u)\phi_j \, dx &= \int_\Omega \nabla u \nabla \phi_j \, dx - \oint_{\partial\Omega} \underbrace{(\nabla u \cdot \underline{n})}_{=g_N \text{ auf } \Sigma_N} \phi_j \, d\sigma \\ &= a(u,\phi_j) - \oint_{\Sigma_N} g_N \phi_j \, d\sigma \ . \end{aligned}$$

Für $j \in I(\Omega)$ gilt $\phi_j = 0$ auf $\partial\Omega \supset \Sigma_N$, so dass das Randintegral in diesem Fall verschwindet.

Setze $X := \mathbb{R}^n, \quad X^\star := \mathbb{R}^n$ und definiere den Operator $J_h : X \to V_{h,N}$ durch

$$J_h\underline{u} := \sum_{j\in I(\Omega\cup\Sigma_N)} \underline{u}_j\phi_j + \sum_{j\in I(\Sigma_D)} g_D(x_j)\phi_j \ ,$$

welcher die Beziehung zwischen den Lösungsvektoren $\underline{u} \in X$ des linearen Gleichungssystems auf der einen Seite und dem Funktionenraum $V_{h,N}$ auf der anderen Seite herstellt. Weiter sei der Operator $M_h : V_{h,N} \to X^\star$ durch

$$M_h f := \underline{v} \quad \text{mit} \quad \underline{v}_i = (f, \phi_i)_0$$

definiert, welcher bei der Diskretisierung der rechten Seite eine Rolle spielt.

Mit Hilfe von J_h und M_h lässt sich das Galerkinproblem nun schreiben als

$$\begin{cases} \text{Finde } \underline{u} \in X, \text{ so dass gilt} & \\ a(J_h\underline{u}, \phi_i) \;=\; M_h f & \forall\, i \in I(\Omega) \\ a(J_h\underline{u}, \phi_i) \;=\; M_h f + \oint_{\Sigma_N} g_N\phi_i \, d\sigma & \forall\, i \in I(\Sigma_N) \end{cases} \tag{1.8}$$

Dies definiert das lineare Gleichungssystem

$$A\underline{u} = \underline{b} \qquad \text{mit } A \in \mathbb{R}^{n\times n}, \quad \underline{u} \in X \ , \quad \underline{b} \in X^\star$$

und

$$\begin{aligned} A_{ij} &= a(\phi_j, \phi_i) \ , && (1.9) \\ \underline{b}_i &= M_h f - \sum_{j\in I(\Sigma_D)} g_D(x_j) a(\phi_j, \phi_i) \\ &\quad + \oint_{\Sigma_N} g_N\phi_i \, d\sigma \ , && (1.10) \end{aligned}$$

wobei das Randintegral in (1.10) für $i \in I(\Omega)$ verschwindet.

Bemerkung 1.7 X und $X^\star$ sind zwar formal dieselben Räume, tragen aber unterschiedliche Bedeutungen:

- Für $\underline{u} \in X$ entspricht $\underline{u}_i = u(x_i)$ einem *Funktionswert*. X ist also der Raum der Lösungen $\underline{u}$. $J_h\underline{u}$ ist gerade die gesuchte Lösungsfunktion der Differentialgleichung.

- $\underline{v} \in X^\star$ ist die Darstellung eines linearen Funktionals $F \in V^*_{h,N}$ in der dualen Basis $\mathcal{B}^*$. Es gilt also $\underline{v}_i = F(\phi_i)$, was nach dem Darstellungssatz von Riesz einem *Integralausdruck*

$$\underline{v}_i = (w, \phi_i)_1 = \int_\Omega w \cdot \phi_i + \nabla w \cdot \nabla \phi_i \, dx$$

  für ein gewisses $w \in V_{h,N}$ entspricht. Im Falle $\underline{v} = M_h f$ für ein $f \in V_{h,N}$ gilt gerade $F(\phi) = (f, \phi)_0$ für alle $\phi \in V_{h,N}$.

  Die rechte Seite $\underline{b}$ eines durch Diskretisierung des Variationsproblems (1.7) entstandenen linearen Gleichungssystems $A\underline{u} = \underline{b}$ ist Element von $X^\star$; im Falle homogener Dirichletranddaten gilt gerade $\underline{b} = M_h f$.

- Die Matrix A ist die Matrixdarstellung der Bilinearform $a(\cdot, \cdot)$ zur Basis $\mathcal{B}$. Sie lässt sich auch als Operator $A : X \to X^\star$ auffassen: Im Falle homogener Dirichletrandwerte entspricht für $\underline{x} \in X$ der Vektor $A\underline{x} \in X^\star$ gerade dem linearen Funktional

$$\phi \in V_{h,N} \quad \mapsto \quad a(J_h \underline{x}, \phi) \ .$$

Bemerkung 1.8 Auf Gleichung (1.8) kann man auch durch einen anderen Ansatz für inhomogene Dirichletranddaten stoßen. Man ersetzt das inhomogene Problem für u durch ein homogenes Problem für $\tilde{u} = u - u_0$, wobei u_0 eine beliebige $H^1(\Omega)$-Funktion mit $u_0 = g_D$ auf Σ_D und $\frac{\partial u_0}{\partial n} = 0$ auf Σ_N ist:

$$\begin{cases} \text{Finde } \tilde{u} \in H^1_0(\Omega), \text{ so dass gilt} \\ \begin{array}{rcll} a(\tilde{u}, \phi_i) & = & \tilde{f}(\phi_i) & \forall\, i \in I(\Omega) \\ a(\tilde{u}, \phi_i) & = & \tilde{f}(\phi_i) + \oint_{\Sigma_N} g_N \phi_i \, d\sigma & \forall\, i \in I(\Sigma_N) \end{array} \end{cases}$$

mit

$$\tilde{f}(\phi) := (f, \phi)_0 - a(J_h \underline{u_0}, \phi) \ .$$

Problem (1.8) erhält man dann aus obigem Ansatz durch die spezielle Wahl

$$u_0 := \sum_{i \in I(\Sigma_D)} g_D(x_i) \cdot \phi_i \ .$$

Beachte, dass die Randwerte also aufgrund der linearen Finiten Elemente nur linear approximiert werden.

1.3.3 Abschätzung des Diskretisierungsfehlers

Entscheidend für die Güte der numerischen Lösung ist die Frage nach dem durch die Galerkin-Approximation entstehenden Diskretisierungsfehler $||u - u_h||$ in einer geeigneten Norm $||\cdot||$. Dabei hilft das nachstehende Lemma von Céa, welches die Abschätzung des Diskretisierungsfehlers auf eine Abschätzung des Approximationsfehlers zurückführt.

Satz 1.9 (Lemma von Céa) *Sei $(V, ||\cdot||)$ ein reeller Hilbertraum und $V_h \subset V$ ein endlichdimensionaler Teilraum. $a(\cdot,\cdot) : V \times V \to \mathbb{R}$ sei eine beschränkte V-koerzive Bilinearform und $f^* \in V^*$ ein beschränktes lineares Funktional auf V. Dann gilt für die Lösungen u bzw. u_h von*

$$a(u, v) = f^*(v) \qquad \forall\, v \in V \quad \textit{bzw.} \quad a(u, v_h) = f^*(v) \qquad \forall\, v \in V_h$$

die Fehlerabschätzung

$$||u - u_h|| \leq \frac{M}{\alpha} \inf_{v \in V_h} ||u - v_h|| \ , \tag{1.11}$$

wobei α, M die Elliptizitäts- bzw. Stetigkeitskonstanten aus Satz 1.3 sind.

Es wird der Fall homogener Dirichletrandwerte betrachtet, also $V := H_0^1(\Omega)$ und $V_h := \mathcal{P}_{1,D}(\mathcal{T}_h)$, wofür die Voraussetzungen des Lemmas von Céa bereits in Abschnitt 1.2 gezeigt wurden. Die Frage nach der Approximationsgüte von V_h bezüglich V führt zu der Untersuchung von Projektionsoperatoren $\Pi_h : V \to V_h$. Dabei spielt neben der Regularität der Lösung u (u soll zusätzlich in $H^2(\Omega)$ liegen) die Entartetheit der Triangulierung $\mathcal{T}_h$ eine besondere Rolle. Anschaulich bedeutet dies, dass die Innenwinkel der Tetraeder gleichmäßig von 0 und π weg beschränkt sind.

Definition 1.10 (Entartungsmaß, Stabilität)

1. Für einen Tetraeder T bezeichne $h(T)$ die Länge seiner längsten Kante sowie $\rho(T)$ den Durchmesser der größten in T enthaltenen Kugel. Das Verhältnis dieser beiden Größen
$$\delta(T) := \frac{h(T)}{\rho(T)}$$
heißt *Entartungsmaß* für T. T heißt *entartet*, wenn $\delta(T) = \infty$ gilt.

2. Für eine Triangulierung $\mathcal{T}$ eines polyedrischen Gebietes $\Omega \subset \mathbb{R}^3$ definiere
$$h(\mathcal{T}) := \max_{T \in \mathcal{T}} h(T)$$

sowie

$$\delta(\mathcal{T}) := \max_{T \in \mathcal{T}} \delta(T) \ .$$

Eine Familie $\{\mathcal{T}_h\}_{h>0}$ von Triangulierungen heißt *stabil*, falls ihre Entartung

$$\delta := \sup_{\mathcal{T}_h} \delta(\mathcal{T}_h)$$

endlich ist.

Satz 1.11 (Approximationssatz) *Sei $\Omega \subset \mathbb{R}^3$ ein polyedrisches Lipschitzgebiet, $\{\mathcal{T}_h\}_{h>0}$ eine stabile Familie konsistenter Triangulierungen von Ω. Dann existieren Projektionsoperatoren $\Pi_h : H^2(\Omega) \to \mathcal{P}_{1,D}$ mit*

$$||u - \Pi_h u||_1 \leq Ch|u|_2 \qquad \forall\, u \in H_0^1(\Omega) \cap H^2(\Omega) \ , \tag{1.12}$$

wobei $C = C(\delta)$ unabhängig von u und h ist.

Aus den Sätzen 1.9 und 1.11 folgt nun

Satz 1.12 (Fehlerabschätzung in der H^1-Norm) *Es gelte die Voraussetzung von Satz 1.11. Weiterhin sei das Variationsproblem (1.2) $H^2(\Omega)$-regulär. Dann besitzt für jede der Triangulierungen $\mathcal{T}_h$ das zugehörige Galerkinproblem (1.4) eine eindeutige Lösung $u_h \in V_h$, und es gilt die Fehlerabschätzung*

$$||u - u_h||_1 \leq \frac{CM}{\alpha} h \ |u|_2$$

mit der von u und h unabhängigen Konstanten $C = C(\delta)$ aus Satz 1.11, sowie $|u|_2^2 := \sum_{|\alpha|=2} ||D^\alpha u||_0$.

Beweis: Zunächst zur Existenz von u_h: Wegen $V_h \subset V$ (Konformität der Ansatzräume) ist $a(\cdot,\cdot)$ auch auf V_h eine elliptische beschränkte Bilinearform. Ebenso ist die Abbildung $(\cdot, v_h)_0$ für alle $v_h \in V_h$ ein beschränktes lineares Funktional auf V_h. Also lässt sich das Lax-Milgram-Lemma 1.4 auch auf das entsprechende Galerkinproblem (1.4) anwenden, und es folgt die Existenz und Eindeutigkeit einer Lösung u_h.

Die Abschätzung des Diskretisierungsfehlers folgt unmittelbar aus den Sätzen 1.9 und 1.11. □

Für $u \in H^2(\Omega)$ folgt aus vorstehendem Satz auch gleichzeitig die Konvergenz des Finite-Elemente-Verfahrens für $h \to 0$. Zwar liefert die Variationsgleichung per se nur $u \in H^1(\Omega)$, Regularitätssätze zeigen aber, dass die Lösung u $H^2(\Omega)$-regulär ist, falls der Rand und die Randwerte genügend glatt (C^2) sind.

Obige Abschätzung in der H^1-Norm lässt sich über die Betrachtung des *adjungierten Problems* auch in der L_2-Norm durchführen, wobei man eine h-Potenz hinzugewinnt:

Satz 1.13 (Fehlerabschätzung in der L_2-Norm) *Zusätzlich zu den Voraussetzungen von Satz 1.12 sei auch das* adjungierte Problem

$$\begin{cases} \textit{Finde } w \in H_0^1(\Omega), \textit{ so dass} \\ a(\phi, w) = (\tilde{f}, \phi)_0 \qquad \forall\, \phi \in H_0^1(\Omega) \textit{ gilt.} \end{cases} \tag{1.13}$$

H^2-regulär. (Ist die Bilinearform $a(\cdot,\cdot)$ symmetrisch, wie im Falle des Poissonproblems, so stimmen das adjungierte Problem (1.13) und das ursprüngliche Variationsproblem (1.2) überein, so dass keine zusätzlichen Voraussetzungen getroffen werden müssen.) Dann gilt folgende Abschätzung in der L_2-Norm:

$$||u - u_h||_0 \leq \frac{CM^2}{\alpha} h^2\ |u|_2 \tag{1.14}$$

für alle $u \in H^2(\Omega)$.

Beweis: Zu $\tilde{f} := u - u_h$ sei w die zugehörige Lösung des adjungierten Problems. Dann gilt insbesondere

$$||u - u_h||_0^2 = (u - u_h, u - u_h)_0 = a(u - u_h, w)\ .$$

Der Fehler $u - u_h$ ist a-orthogonal zum Ansatzraum V_h, d.h.

$$a(u - u_h, v_h) = (f, v_h)_0 - (f, v_h)_0 = 0 \quad \forall\, v_h \in V_h\ .$$

Daraus folgt unter Beachtung der Stetigkeit von $a(\cdot,\cdot)$

$$||u - u_h||_0^2 = a(u - u_h, w - v_h) \leq M||u - u_h||_1\ ||w - v_h||_1 \quad \forall\, v_h \in V_h.$$

Aus dem vorangegangenen Satz 1.12 lässt sich $||u - u_h||_1$ weiter abschätzen, somit ergibt sich

$$||u - u_h||_0^2 \leq M\frac{CM}{\alpha} h\ |u|_2\ \inf_{v_h \in V_h} ||w - v_h||_1$$

für $u \in H^2(\Omega)$. Auch der aus dem Céa-Lemma bekannte Infimum-Ausdruck lässt sich mit Hilfe von Satz 1.11 in dieser Weise abschätzen. Da das adjungierte Problem H^2-regulär ist, gilt zudem

$$\inf_{v_h \in V_h} ||w - v_h||_1 \leq Ch\ |w|_2 \leq CC'h\ ||\tilde{f}||_0\ .$$

Insgesamt folgt also

$$||u - u_h||_0^2 \leq \frac{C''M^2}{\alpha} h^2\ |u|_2\ ||\tilde{f}||_0\ ,$$

und unter Beachtung von $\tilde{f} = u - u_h$ ergibt sich schließlich die gewünschte L_2-Fehlerabschätzung (1.14). □

Dieses theoretische Resultat bedeutet in der Praxis: Bei Halbierung von h (d.h. bei einmaliger uniformer Verfeinerung) erwartet man, dass der L_2-Fehler $||u - u_h||_0$ etwa mit einem Faktor 4 sinkt.

Kapitel 2

Aspekte der Parallelisierung

Die numerische Simulation von Prozessen aus dem CFD-Bereich führt häufig zu Problemstellungen von hoher numerischer Komplexität, sowohl was den Rechen- als auch den Speicheraufwand angeht. Da im Rahmen des SFB 540 gerade solche Fragestellungen interessant sind (instationäre reaktive mehrphasige Strömungen in 3D), ist für DROPS die Parallelisierung eine wichtige Aufgabe, da diese das Rechenzeit- und Speicherplatzproblem entschärfen kann.

Als Parallelrechnermodell wird von einer *distributed-memory-Maschine* ausgegangen, d.h. einer gewissen Anzahl an Prozessoren, die jeweils mit einem lokalen Speicher ausgestattet und durch ein Netzwerk miteinander verbunden sind, über welches sie kommunizieren können. Der Datenaustausch erfolgt über das Senden und Empfangen von Nachrichten (*message passing*); den Quasistandard in diesem Bereich stellt MPI (*message passing interface*) dar.

Für die Parallelisierung von DROPS wurde deswegen MPI als Grundlage gewählt. Der parallele Code von DROPS gehört zur Klasse der SPMD-Programme (*single program – multiple data*), d.h. alle Prozessoren verarbeiten dasselbe Programm, aber mit unterschiedlichen (lokalen) Daten. Die Prozessoren können sich zu einem bestimmten Zeitpunkt an unterschiedlichen Stellen des Programms befinden. Aus diesem Grund gehört die Klasse SPMD gemäß der Grobklassifikation nach FLYNN der Oberklasse MIMD (*multiple instruction stream, multiple data stream*) an.

2.1 Leistungsbewertung paralleler Algorithmen

Die Güte eines parallelen Algorithmus wird an seiner Skalierbarkeit gemessen, welche die Eignung des Algorithmus für größere Prozessorzahlen beschreibt. Dazu wird die Effizienz des Algorithmus, d.h. sein Verhalten bei Ausführung auf verschiedenen Prozessorzahlen, betrachtet. Der ersten Erläuterung dieser Begriffe dient folgende

Definition 2.1 (Speedup, Effizienz, Skalierbarkeit)

- Der *(parallele) Speedup* $S_n(P)$ beschreibt den Zeitgewinn eines parallelen Algorithmus auf P Prozessoren gegenüber der seriellen Version bei einer festen Problemgröße n:
$$S_n(P) = \frac{T_n(1)}{T_n(P)} \ .$$
$T_n(j)$ bezeichnet die Rechenzeit des Algorithmus für ein Problem der Größe n auf j Prozessoren.

- Die *(parallele) Effizienz* $E_n(P)$ ist definiert als
$$E_n(P) = \frac{S_n(P)}{P} \ .$$

- Ein paralleler Algorithmus heißt *skalierbar* für P Prozessoren, wenn $E_n(P)$ fast 1 ist, wobei eine angemessene Gesamtproblemgröße n vorausgesetzt wird.

In der Praxis kann man nur für eine hinreichend hohe Problemgröße erwarten, dass ein Algorithmus auf P Prozessoren auch etwa P-mal so schnell läuft wie auf einem Prozessor. Wenn bei steigender Prozessorzahl P das lokale Problem auf einem Prozessor zu klein wird, ist eine Messung der Rechenleistung so nicht mehr sinnvoll, weil der Anteil des Rechenaufwandes gegenüber dem des Kommunikationsaufwandes stark abfällt.

Die obengenannte Definition ist unter diesen Überlegungen in der Praxis somit nur bedingt zur Beurteilung der Skalierbarkeit geeignet, da in dieser Definition für das serielle und das parallele Problem dieselbe fixierte Gesamtproblemgröße n verwendet wird. Um für den parallelen Fall eine ausreichend große lokale Problemgröße n' zu gewährleisten, muss aber eine Gesamtgröße von $n = P{\cdot}n'$ gewählt werden, was bei größeren Prozessorzahlen P dazu führt, dass das Problem für den seriellen Lauf nicht mehr in einen einzelnen Speicher passt. Die parallele Effizienz eines Algorithmus kann auf diese Weise also nur für kleinere Prozessorzahlen ermittelt werden.

Für Algorithmen von der Komplexität $\mathcal{O}(n)$ bietet sich ein anderer Weg an: Aus der ermittelten Laufzeit $T_{n'}(1)$ kann auf die serielle Laufzeit des P-mal größeren Problems $T_{P\cdot n'}(1) \approx P \cdot T_{n'}(1)$ geschlossen werden, so dass nun eine Gesamtproblemgröße $n = n(P) = P \cdot n'$ proportional zu P möglich wird. n' ist dabei unabhängig von P gewählt und stellt die lokale Teilproblemgröße dar. Deswegen ist für solche Algorithmen die folgende Definition der Anwendung in der Praxis angemessener:

Definition 2.2 (Skalierter Speedup, skalierte Effizienz, Skalierbarkeit)

- Der *skalierte Speedup* $S_{n'}^{\mathrm{sc}}(P)$ beschreibt den Zeitgewinn eines parallelen Algorithmus auf P Prozessoren gegenüber der seriellen Version bei fester *Teilproblemgröße* n':
$$S_{n'}^{\mathrm{sc}}(P) = \frac{P \cdot T_{n'}(1)}{T_{P \cdot n'}(P)} \; .$$
- Die *skalierte Effizienz* $E_{n'}^{\mathrm{sc}}(P)$ ist (analog zur parallelen Effizienz) definiert als
$$E_{n'}^{\mathrm{sc}}(P) = \frac{S_{n'}^{\mathrm{sc}}(P)}{P} = \frac{T_{n'}(1)}{T_{P \cdot n'}(P)} \; .$$
- Ein paralleler Algorithmus heißt *skalierbar* für P Prozessoren, wenn $E_{n'}^{\mathrm{sc}}(P)$ fast 1 ist für eine angemessen große lokale Problemgröße n'.

Bemerkung 2.3 (Interpretation der beiden Speedup-Begriffe)
Der Unterschied von parallelem und skaliertem Speedup lässt sich gut an folgendem Beispiel aus der Arbeitswelt verdeutlichen: Wenn 1 Arbeiter für das Mähen von 1000 m^2 Rasen 1 Stunde braucht, so schaffen 100 Arbeiter in 1 Stunde etwa 100 000 m^2, wenn jeder von ihnen ein zusammenhängendes 1000-m^2-Stück zugeteilt bekommt. Dies entspricht der Idee des skalierten Speedups.

Man kann aber nicht erwarten, dass 100 Arbeiter ein 1000 m^2 großes Rasenstück in 1/100 Stunde, also 36 Sekunden, mähen. Die Arbeiter würden sich gegenseitig behindern, und die Absprache der Arbeiter untereinander (Organisationsaufwand) würde einen großen Teil der Zeit verschlingen. Darin kommt der Nachteil bei der Messung des parallelen Speedups zum Ausdruck.

Bei den bisherigen Betrachtungen wurden einige idealisierende Annahmen getroffen, die so in der Praxis nicht gelten. Insbesondere sind drei wichtige Aspekte nicht berücksichtigt worden, die in der Praxis i.A. dazu führen, dass die Ausführung des Algorithmus auf P Prozessoren länger dauert als die P-fache serielle Ausführungszeit:

1. Die beiden Speedup-Begriffe vernachlässigen den Anteil der Kommunikation am Gesamtaufwand. Der Aufwand für die Kommunikation fällt nur im parallelen Fall an und ist nicht unerheblich (einige hundert Gleitkommaoperationen pro Kommunikation).
2. (Fast) jeder Algorithmus enthält Teile, die sich nicht parallelisieren lassen, weil ihnen eine stark serielle Struktur innewohnt.

3. Ist die Rechenlast nicht gleichmäßig auf die Prozessoren verteilt, so liegt der maximal erreichbare Speedup (wiederum unter Vernachlässigung des Kommunikationsaufwandes) bei

$$S_n(P) \leq \frac{T_n(1)}{\max_{1 \leq p \leq P} T_{n_p}(1)} ,$$

wobei n_p die Größe des lokalen Problems auf Prozessor p bezeichne.

Im Unterschied zu Punkt 1 und 2, die sich zwingend aus der Parallelisierung bzw. dem Algorithmus ergeben und zwar berücksichtigt, aber nicht umgangen werden können, kann das in Punkt 3 angesprochene Problem des Lastungleichgewichtes durch entsprechende Lastverteilungsstrategien (siehe Kapitel 5) nahezu vollständig behoben werden, wobei auch dem zu erwartenden Kommunikationsaufwand durch eine Minimierung der entstehenden Prozessorränder Rechnung getragen wird. Deswegen werden zunächst die Auswirkungen der ersten beiden Punkte weiter betrachtet.

Zu Punkt 1: Die Kommunikation einer Nachricht der Länge n von einem Prozessor zum anderen benötigt typischerweise die Zeit

$$T_{\text{comm}}(n) = (\alpha + \beta n)\, T_A ,$$

wobei T_A der Zeit für die Berechnung einer Addition entspreche. $T_{\text{comm}}(n)$ setzt sich also aus einer sogenannten mittleren *Aufsetzzeit* der Größe αT_A, die aus der Latenzzeit des verwendeten Netzwerks resultiert, sowie der reinen Sendezeit $\beta n T_A$ zusammen, die proportional zur Länge der Nachricht ist. α und β sind von der Qualität des Netzwerks abhängige Konstanten. Realistische Werte sind z.B. $\alpha = 1000$ und $\beta = 10$; unter diesen Voraussetzungen würde z.B. die reine Sendezeit für eine Nachricht der Länge $n = \alpha/\beta = 100$ genau der vorher benötigten Aufsetzzeit entsprechen.

Um den Effekt der langen Aufsetzzeiten zu vermindern, sollten also möglichst lange Nachrichten kommuniziert werden. Statt viele kleine Nachrichten an einen anderen Prozessor zu senden, lautet also die Strategie, diese besser zu möglichst großen Datenpaketen zu bündeln und dann en bloc zu verschicken. Die von DDD (siehe Abschnitt 2.4) über das *message passing interface* abgesetzte Kommunikation ist genau diesem Prinzip verschrieben. Dabei muss das Bündeln und verzögerte Senden der Daten immer dagegen abgewogen werden, ob der andere beteiligte Prozessor diese nicht frühzeitig benötigt, um weiter arbeiten zu können.

Zu Punkt 2: Das *Amdahlsche Gesetz* untersucht die Auswirkung von Punkt 2 auf den Speedup:

Satz 2.4 (Amdahlsches Gesetz) *Seien s der serielle und p der parallele Anteil eines Algorithmus mit $s+p=1$. Dann ergibt sich als obere Schranke für den maximal erreichbaren Speedup*

$$S_n(P) \leq \frac{1}{s} \ .$$

Beweis: Da der serielle Teil nicht parallelisierbar ist, ergibt sich für die parallele Ausführungszeit

$$T_n(P) \geq (s + \frac{p}{P})T_n(1)$$

und somit

$$S_n(P) = \frac{T_n(1)}{T_n(P)} \leq \frac{1}{s + \frac{1-s}{P}} \leq \frac{1}{s} \ .$$

□

Für einen seriellen Anteil von z.B. $s = 1\%$ bedeutet dies einen maximalen Speedup von 100, unabhängig von der Anzahl der Prozessoren. Überdies berücksichtigt das Amdahlsche Gesetz nicht die Auswirkung von Kommunikationszeiten, so dass das reale Verhalten des Speedup bei zu kleinen Problemgrößen noch schlechter als von Amdahls Gesetz prognostiziert ausfällt.

Sinnvolle Aussagen über die Skalierbarkeit lassen sich also nur für angemessene Problemgrößen treffen, statt $S_n(p)$ sollte also besser der *skalierte* Speedup $S_n^{\mathrm{sc}}(P)$ betrachtet werden. Da hierbei die lokale Problemgröße n' unabhängig von P festgelegt ist, jeder Prozessor also unabhängig von der Anzahl der eingesetzten Prozessoren stets dieselbe Rechenlast besitzt, ergibt sich in Analogie zum Ahmdahlschen Gesetz für den skalierten Speedup folgendes Resultat:

Satz 2.5 *Es bezeichne n' eine fest gewählte lokale Teilproblemgröße und $n = n'P$ die Gesamtproblemgröße auf P Prozessoren. Der parallele Anteil des Algorithmus sei von der Komplexität $\mathcal{O}(n)$. Die Ausführungszeit $T_{\mathrm{ser}} = s \cdot T_{n'}(1)$ des seriellen Anteils des Algorithmus sei unabhängig von n. Dann gilt*

$$S_{n'}^{\mathrm{sc}}(P) \approx \frac{T_{n'P}(1)}{T_{n'P}(P)} = s + P(1-s) \ .$$

Beweis: Mit $p = 1 - s$ ist

$$\frac{T_{n'P}(1)}{T_{n'P}(P)} = \frac{(s+pP)T_{n'}(1)}{(s+p)T_{n'}(1)} = s + pP \ .$$

□

Geht man wieder von $s = 1\%$ aus, so bedeutet dies, dass der skalierte Speedup fast P beträgt, die skalierte Effizienz ist also beinahe 1. Parallele $\mathcal{O}(n)$-Algorithmen (wie der in dieser Arbeit vorgestellte parallele Verfeinerungsalgorithmus oder ein paralleles Mehrgitterverfahren) lassen also eine gute Skalierbarkeit erwarten, wenn man eine feste Teilproblemgröße wählt und so die Gesamtproblemgröße mit der Anzahl der Prozessoren skaliert.

2.2 Verteilung der Daten in DROPS

Im Folgenden soll zunächst erläutert werden, welche grundlegenden Überlegungsschritte dazu führen, die Aufteilung der geometrischen und numerischen Daten unter den Prozessoren so vorzunehmen, wie in DROPS geschehen, damit die parallelen Algorithmen effizient ausgeführt werden können. Die Idee dabei ist die folgende:

Eine erfolgreiche Parallelisierung sollte zuallererst bei dem rechenintensivsten Teil eines Programmes ansetzen, der den größten Anteil der Gesamtrechenzeit ausmacht, erst dann kann man die gewünschte Skalierbarkeit des Algorithmus überhaupt erwarten. Im vorliegenden Fall liegt die Hauptrechenlast in der Lösung des linearen Gleichungssystems $Ax = b$, also im CG-Algorithmus, und dort speziell in der Matrix-Vektor-Multiplikation, die den Aufwand eines Iterationsschrittes größenordnungsmäßig bestimmt. Auch in anderen iterativen Lösern ist die Matrix-Vektor-Multiplikation die ausschlaggebende Operation, so dass bei der numerischen Lösung von partiellen Differentialgleichungen generell die Parallelisierung bei dem Matrix-Vektor-Produkt ansetzen sollte.

Parallele Multiplikation setzt parallele Datenspeicherung voraus, d.h. jeder Prozessor speichert einen bestimmten Abschnitt der Matrizen und Vektoren derart, dass das Produkt einfach parallel berechnet werden kann. Dies wird näher in Abschnitt 2.3 diskutiert.

Die parallele Speicherung der Geometriedaten ergibt sich nun indirekt aus der Aufteilung der numerischen Daten, da die Diskretisierung und damit die Erzeugung der Steifigkeitsmatrix und des Vektors der rechten Seite auf den Geometriedaten fußt: Ein Prozessor p speichert deswegen diejenigen Matrix- und Vektoranteile, die den Unbekannten auf den Tetraederelementen

$$\{T \in \mathcal{T} : T \text{ ist auf Prozessor } p \text{ gespeichert } \}$$

entsprechen, welche auf diesem Prozessor gespeichert sind.

Die Speicherung der Tetraeder sei zunächst als überlappungsfrei vorausgesetzt, d.h. jeder Tetraeder T ist auf genau einem Prozessor gespeichert. Für den Verfeinerungsalgorithmus sowie die Diskretisierungsphase ist es unabdingbar, dass zu jedem Tetraeder auch seine Knoten, Kanten und Seitenflächen lokal abgespeichert sind. Daraus ergibt sich automatisch eine überlappende Speicherung für die Knoten, Kanten und Seitenflächen. Aufgrund der gewählten P_1-Diskretisierung mit Unbekannten auf den Knoten sind also auch die Unbekannten und damit ebenso die numerischen Daten überlappend gespeichert (dies ist im Übrigen für alle anderen FE-Typen genauso, bis auf die P_0-Elemente, welche nur eine Unbekannte im Schwerpunkt des Elementes besitzen).

Bei der Verteilung der Tetraeder auf die Prozessoren ist zu beachten, dass die Anzahl der Unbekannten auf jedem Prozessor etwa die gleiche ist. Ansonsten würde ein Großteil der Ausführungszeit darauf entfallen, dass die Prozessoren mit wenig Rechenlast auf die mit viel Rechenlast warten müssten und in dieser Zeit keine sinnvolle Arbeit verrichten könnten. Weiterhin ist anzustreben, die Überlappung der numerischen Daten möglichst gering zu halten, weil der bei parallelen Berechnungen vergleichsweise zeitintensive Kommunikationsaufwand proportional zur Überlappung der Unbekannten ist. Diese beiden Forderungen – gleichmäßige Verteilung bei minimaler Überlappung – münden in ein Lastverteilungsproblem, welches mit Load-Balancing-Algorithmen (Kapitel 5) gelöst werden muss.

2.3 Verteilte Matrizen und Vektoren

Wie schon gesagt, erfordert die paralle Berechnung des Matrix-Vektor-Produkts auch eine parallele verteilte Speicherung der Matrizen und Vektoren. Jeder Prozessor speichert dabei einen bestimmten Anteil der numerischen Daten. Manche Werte werden von mehreren Prozessoren gespeichert, es liegt also eine überlappende Speicherung der numerischen Daten vor. Um zu erläutern, welche Art der Überlappung den numerischen Operationen (Matrix-Vektor-Produkt, Skalarprodukt) angemessen ist, werden zunächst die Begriffe der verteilten und akkumulierten Speicherung eingeführt.

Sei dazu n die Anzahl der Unbekannten und n_p die Anzahl der Unbekannten auf Prozessor p. Definiere die Koinzidenzmatrix $I_p \in \mathbb{R}^{n_p \times n}$ zu Prozessor p durch

$$(I_p)_{i,j} := \begin{cases} 1 & \text{Unbekannte mit globaler Nummer } j \\ & \text{existiert auf Prozessor } p \text{ mit lokaler Nummer } i \\ 0 & \text{sonst} \end{cases}$$

Definition 2.6 (Akkumulierter Vektor) $\overline{\mathbf{x}} \in \mathbb{R}^n$ wird in Prozessor p als $\overline{\mathbf{x}}_p$ *akkumuliert* gespeichert, wenn

$$\overline{\mathbf{x}}_p = I_p \overline{\mathbf{x}} \ ,$$

d.h. jeder Prozessor besitzt den gleichen globalen Wert in seinen Unbekannten wie die entsprechenden überlappenden Unbekannten auf anderen Prozessoren.

Definition 2.7 (Verteilter Vektor und verteilte Matrix) $\mathbf{x} \in \mathbb{R}^n$ wird in Prozessor p als $\mathbf{x}_p$ *verteilt* gespeichert, wenn

$$\mathbf{x} = \sum_{p=1}^{P} I_p^T \mathbf{x}_p \ ,$$

d.h. die überlappenden Unbekannten auf den Interfaces enthalten nur einen Teil des vollen Wertes, erst die Summe der zu einer verteilten Unbekannten gehörenden lokalen Werte ergibt den vollen globalen Wert. Dieselbe Bemerkung gilt für eine verteilte Matrix $\mathbf{M}$, deren Anteil $\mathbf{M}_p$ auf Prozessor p gespeichert ist:

$$\mathbf{M} = \sum_{p=1}^{P} I_p^T \mathbf{M}_p I_p \ .$$

Bemerkung 2.8 (Aufstellen der verteilten Steifigkeitsmatrix) Ist $\mathbf{M}$ eine Steifigkeitsmatrix, so ist $\mathbf{M}_p$ gerade die zum Teilgebiet Ω_p von Prozessor p gehörige Steifigkeitsmatrix. Insofern erweist sich die verteilte Form der Abspeicherung von Matrizen als ideal für unsere Belange: Die Matrizen $\mathbf{M}_p$ können in der Diskretisierungsphase völlig unabhängig voneinander aufgestellt werden, also ohne Einsatz von Kommunikation.

Offensichtlich sind die Basisoperationen Addition, Subtraktion und skalare Multiplikation zwischen Vektoren gleichen Typs ohne Kommunikation möglich. Die Umwandlung eines verteilten Vektors in einen akkumulierten (Akkumulation) benötigt dagegen Kommunikation, da die überlappten Unbekannten ihre jeweiligen Werte unter den Prozessoren aufsummieren müssen:

$$\overline{\mathbf{x}}_p = I_p \sum_{i=1}^{P} I_i^T \mathbf{x}_i \ .$$

Die andere Umwandlungsrichtung ist dagegen nicht mehr eindeutig. Eine Möglichkeit besteht darin, unter allen Kopien einer Unbekannten auf einem Interface eine auszuzeichnen (nämlich als Master, die anderen heißen Ghosts), z.B. die mit der kleinsten Prozessornummer, und diesem Master den globalen Wert zuzuweisen; die Ghosts erhalten den Wert Null. Eine andere Möglichkeit besteht darin, den globalen Wert gleichmäßig aufzuteilen:

$$\mathbf{x}_p = I_p R^{-1} I_p^T \overline{\mathbf{x}}_p$$

mit der Diagonalmatrix $R = \sum_{p=1}^{P} I_p^T I_p \in \mathbb{R}^n$, deren Diagonaleintrag R_{ii} die Anzahl der Prozessoren angibt, auf denen die Unbekannte mit der (globalen) Nummer i vorkommt.

Welcher Alternative der Vorzug gegeben wird, kann nur aus der konkreten Implementierungssituation heraus entschieden werden: Wenn auch andere Komponenten des Programmes eine Unterteilung der verteilten Unbekannten in Master und Ghosts benötigen, so ist die erste Alternative anzustreben; andernfalls kann auch die zweite Alternative sinnvoll sein, wenn die entsprechenden Diagonaleinträge von R auf jedem Rechner ohne großen Aufwand (insbesondere ohne Kommunikation) zur Verfügung stehen.

Das Skalarprodukt kann sinnvoll nur für Vektoren unterschiedlichen Typs parallel berechnet werden und benötigt dafür lediglich die globale Summation einer reellen Zahl (Reduce-Operation):

$$(\overline{\mathbf{x}}, \mathbf{y}) = \overline{\mathbf{x}}^T \mathbf{y} = \overline{\mathbf{x}}^T \sum_{p=1}^{P} I_p^T \mathbf{y}_p = \sum_{p=1}^{P} (I_p \overline{\mathbf{x}})^T \mathbf{y}_p = \sum_{p=1}^{P} (\overline{\mathbf{x}}_p, \mathbf{y}_p) \ .$$

Die Skalarprodukte werden also zunächst lokal berechnet und dann die lokalen Ergebnisse unter den Prozessoren aufsummiert. Liegen die Vektoren nicht in unterschiedlicher Form vor, muss zuvor eine Typumwandlung stattfinden, die evtl. zusätzliche Kommunikation erfordert.

Nun zur Matrix-Vektor-Multiplikation:

$$\mathbf{M} \cdot \overline{\mathbf{x}} = \sum_{p=1}^{P} I_p^T \mathbf{M}_p I_p \cdot \overline{\mathbf{x}} = \sum_{p=1}^{P} I_p^T \underbrace{\mathbf{M}_p \overline{\mathbf{x}}_p}_{:=\mathbf{y}_p} = \mathbf{y} \ .$$

Wie man sieht, ergibt das Produkt einer verteilten Matrix mit einem akkumulierten Vektor einen verteilten Vektor. Dabei werden die Produkte lokal parallel berechnet, Kommunikation ist nicht nötig. Soll ein verteilter Vektor mit einer verteilten Matrix multipliziert werden, so muss der Vektor zuvor wie oben beschrieben akkumuliert werden (Kommunikation).

2.3.1 Der parallele CG-Algorithmus

Mit den oben beschriebenen parallelen Matrix-Vektor-Operationen ist es nun ein Leichtes, den seriellen CG-Algorithmus zu parallelisieren: In jeder Iteration entfällt die Hauptrechenlast auf die Berechnung eines Matrix-Vektor-Produktes sowie zweier Skalarprodukte; beide Operationen werden nun durch ihre parallelen Versionen ersetzt.

Ausgehend von der seriellen Fassung des CG-Algorithmus in Abbildung 2.1, ergibt sich daraus zugleich, welche Vektoren verteilt und welche akkumuliert abgespeichert werden müssen: Die Berechnung des Matrix-Vektor-Produktes zwischen A und $\underline{x}$ in (1) erfordert die verteilte Speicherung der Steifigkeitsmatrix $\mathbf{A}$ sowie

$$
\begin{array}{ll}
\textbf{Algorithm } CG(\ A, \underline{x}, \underline{b}, \mathrm{tol}\ ) & \\
\{ & \\
\quad \underline{r} := A\,\underline{x} - \underline{b}; & (1) \\
\quad \underline{d} := -\underline{r}; & (2) \\
\quad \gamma := (\underline{r}, \underline{r}); & (3) \\
\quad \textbf{for } (\ k = 1, \ldots, n\ ) \textbf{ do} & \\
\quad \{ & \\
\qquad \underline{e} := A\,\underline{d}; & (4) \\
\qquad \delta := (\underline{d}, \underline{e}); & (5) \\
\qquad \alpha := \gamma / \delta; & (6) \\
\qquad \underline{x} := \underline{x} + \alpha\,\underline{d}; & (7) \\
\qquad \underline{r} := \underline{r} + \alpha\,\underline{e}; & (8) \\
\qquad \gamma := (\underline{r}, \underline{r}); & (9) \\
\qquad \textbf{if } (\ \gamma < \mathrm{tol}^2\ ) \textbf{ then stop}; & \\
\qquad \underline{d} := \frac{\gamma}{\beta}\,\underline{d} - \underline{r}; & (10) \\
\quad \} & \\
\} &
\end{array}
$$

Abbildung 2.1: Der serielle CG-Algorithmus.

eine akkumulierte Speicherung des Vektors $\overline{\mathbf{x}}$. Da das Ergebnis der Matrix-Vektor-Multiplikation dann in verteilter Form vorliegt, müssen die Vektoren $\mathbf{b}$ sowie $\mathbf{r}$ ebenfalls verteilt abgespeichert werden.

Das in jeder Iteration berechnete Matrix-Vektor-Produkt (4) benötigt analog die akkumulierte Speicherung des Vektors $\overline{\mathbf{d}}$ sowie die verteilte Speicherung des Vektors $\mathbf{e}$.

Nun zu den Skalarprodukten in (3), (5) und (9): Die parallele Berechnung des Skalarproduktes benötigt einen der beiden Vektoren in akkumulierter sowie den anderen in verteilter Form. Für das Skalarprodukt δ in (5) zwischen $\overline{\mathbf{d}}$ und $\mathbf{e}$ ist dies wegen der vorangegangenen Überlegungen zum Matrix-Vektor-Produkt (4) bereits gegeben.

Die Auswertung des Skalarproduktes γ in (3) bzw. (9) erfordert, da $\mathbf{r}$ bislang nur in verteilter Form vorliegt, einen weiteren Vektor $\overline{\mathbf{r}}$, der zuvor aus $\mathbf{r}$ akkumuliert werden muss. Beachte, dass $\mathbf{r}$ und $\overline{\mathbf{r}}$ zwei *verschiedene* Vektoren (insbesondere mit unterschiedlichen Speicherbereichen) darstellen!

Ein Blick auf die verbleibenden Vektoradditionen bzw. -subtraktionen in (7), (8) und (10) zeigt, dass die jeweils beteiligten Vektoren gleichen Typs sind. Bis auf die Akkumulation von $\mathbf{r}$ für die parallele Berechnung des Skalarproduktes γ sind also keine weiteren Typumwandlungen (und damit Kommunikation) nötig.

Die in einem parallel ausgeführten Iterationsschritt anfallende Kommunikation besteht also hauptsächlich aus der Akkumulation von $\mathbf{r}$, hinzu kommt die parallele Berechnung zweier Skalarprodukte, welche aber jeweils nur die globale Summation einer reellen Zahl bedeuten.

Algorithm $CG(\ \mathbf{A}, \overline{\mathbf{x}}, \mathbf{b}, \mathrm{tol}\ )$
{
  $\mathbf{r} := \mathbf{A}\,\overline{\mathbf{x}} - \mathbf{b}$;
  $\overline{\mathbf{r}} :=$ `Akkumuliere(` $\mathbf{r}$ `);`
  $\overline{\mathbf{d}} := -\overline{\mathbf{r}}$;
  $\gamma := (\overline{\mathbf{r}}, \mathbf{r})$;
  for $(\ k = 1, \ldots, n_p\ )$ **do**
  {
    $\mathbf{e} := \mathbf{A}\,\overline{\mathbf{d}}$;
    $\delta := (\overline{\mathbf{d}}, \mathbf{e})$;
    $\alpha := \gamma/\delta$;
    $\overline{\mathbf{x}} := \overline{\mathbf{x}} + \alpha\,\overline{\mathbf{d}}$;
    $\mathbf{r} := \mathbf{r} + \alpha\,\mathbf{e}$;
    $\overline{\mathbf{r}} :=$ `Akkumuliere(` $\mathbf{r}$ `);`
    $\gamma := (\overline{\mathbf{r}}, \mathbf{r})$;
    if $(\ \gamma < \mathrm{tol}^2\ )$ **then stop**;
    $\overline{\mathbf{d}} := \frac{\gamma}{\beta}\,\overline{\mathbf{d}} - \overline{\mathbf{r}}$;
  }
}

Abbildung 2.2: Der parallele CG-Algorithmus.

2.4 Die DDD-Bibliothek: dynamic distributed data

Die von Birken entwickelte Programmbibliothek DDD[1] wurde bereits in UG[2] erfolgreich eingesetzt, um den seriellen Code zu parallelisieren. DDD schafft dabei eine abstrakte Schnittstelle, um auf mehreren Prozessoren verteilt gespeicherte Daten konsistent zu verwalten, die sich graphartig untereinander referenzieren. Die eigentliche DDD-Implementierung (z.B. Kopplungslisten) ist hinter der Schnittstelle verborgen, die ein weitreichendes Angebot an Funktionen wie Datentransfer, Interface-Kommunikation usw. bereithält. Gleichzeitig hat der Anwender über die Definition von Handler-Funktionen die Möglichkeit, die DDD-

[1] `http://cox.iwr.uni-heidelberg.de/~ddd/`
[2] `http://cox.iwr.uni-heidelberg.de/~ug/`

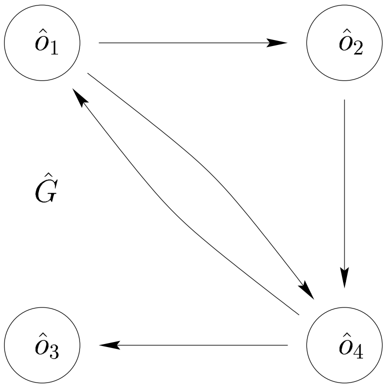


Abbildung 2.3: Beispiel eines globalen Graphen.

Funktionalität auf sein spezielles Problem hin anzupassen, so dass DDD trotz seines Umfangs ausreichend große Flexibilität bietet.

Für DROPS ist zunächst die parallele Verwaltung der Geometriedatenstruktur durch DDD von Interesse, darauf aufbauend wird die angebotene Interface-Kommunikation über Prozessorgrenzen hinweg eingesetzt. So wird z.B. die Akkumulation von Vektoren im CG-Löser oder der Update der Kantenverfeinerungszähler im Verfeinerungsalgorithmus einfach und effizient möglich.

2.4.1 Verteilte Objekte

Dem DDD-Konzept liegt folgendes theoretische Modell zugrunde: Die Daten des seriellen Programms liegen als gerichteter Graph $\hat{G} = (\hat{O}, \hat{R})$ der globalen Objekte $\hat{O} = \{\hat{o}_1, \ldots, \hat{o}_n\}$ vor, die Kanten $\hat{R} \subset \hat{O} \times \hat{O}$ entsprechen Referenzen der Objekte untereinander. Im Falle von DROPS sind die Objekte O_{DROPS} die Tetraeder, Seitenflächen, Kanten und Knoten; eine Referenz $(o_1, o_2) \in R_{\text{DROPS}}$ bedeutet, dass das Objekt o_1 einen Zeiger auf das Objekt o_2 enthält.

Bei einer auf einem Parallelrechner verteilten Datenstruktur taucht erstmals der Begriff der Lokalität auf: Bei distributed-memory-Maschinen[3] ist kein globaler Adressraum (Speicher) mehr vorhanden, sondern jeder Prozessor besitzt seinen eigenen lokalen Adressraum. Das bedeutet, dass der Graph $\hat{G}$ aus dem seriellen Fall bei verteilter, paralleler Speicherung in lokale Graphen G^p zerfällt.

Sei O^p die Menge der auf Prozessor p lokal gespeicherten Objekte. Die auf verschiedenen Prozessoren gespeicherten Objektkopien eines überlappt gespeicherten

[3]Bei shared-memory-Maschinen oder NUMA-Architekturen gibt es zwar das Konzept eines globalen Adressraumes, der Lokalitätsbegriff findet aber auch hier seine sinnvolle Anwendung, da der Speicher sich aus verschiedenen Bereichen mit unterschiedlichen Zugriffszeiten zusammensetzt.

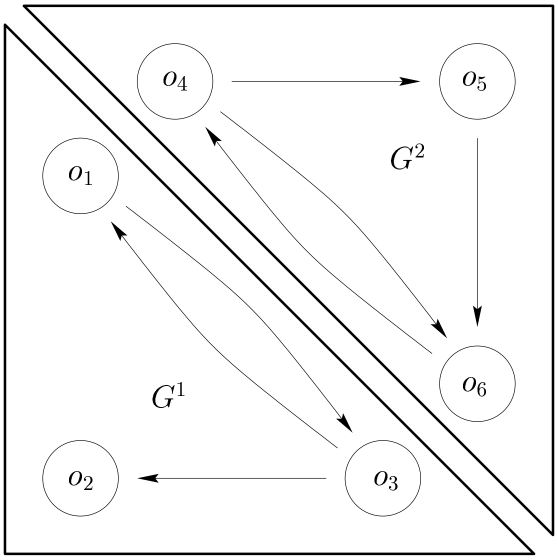


Abbildung 2.4: Beispiel zweier lokaler Graphen.

globalen Objektes werden dabei als *unterschiedliche lokale* Objekte aufgefasst. Aus der Lokalität der Daten folgt dann also

$$O^p \cap O^q = \emptyset \qquad \text{für } p \neq q \ .$$

Die Vereinigung der auf den verschiedenen Prozessoren gespeicherten lokalen Objekte

$$O = \bigcup_{p=1}^{P} O^p$$

bildet die Menge aller auf dem Parallelrechner gespeicherten lokalen Objekte. Beachte, dass die Mengen $O^p, \quad p = 1, \ldots, P$, eine Partition von O bilden.

$G^p = (O^p, R^p)$ stellt den lokalen Graph auf Prozessor p dar, wobei die $R^p \subset O^p \times O^p$ Referenzen zwischen lokalen Objekten entsprechen, die ausschließlich auf Prozessor p gespeichert sind. Da Referenzen über Prozessorgrenzen hinweg nicht möglich sind, muss die Verkopplung der lokalen Graphen G^p zu einem verteilten Graph $\tilde{G}$ also auf Objektebene statt auf Referenzebene stattfinden. Dies geschieht durch Vergabe von *globalen Nummern*, welche jedem Objekt durch die Abbildung

$$\mathsf{gid} : O \to I, \qquad I = \{1, \ldots, |\hat{O}|\} \ ,$$

zugeordnet werden. Da alle globalen Objekte lokal repräsentiert werden sollen, sei die Surjektivität von gid vorausgesetzt.

Die Kopplung der lokalen Graphen G^p zu einem verteilten Graphen $\tilde{G}$, der $\hat{G}$ entspricht, wird nun durch folgende Äquivalenzrelation $\sim_{\mathsf{gid}} \subset O \times O$ mit

$$a \sim_{\mathsf{gid}} b \quad :\Leftrightarrow \quad \mathsf{gid}(a) = \mathsf{gid}(b)$$

erreicht: $\sim_{\mathsf{gid}}$ definiert eine Unterteilung von O in Äquivalenzklassen. Die Menge dieser Äquivalenzklassen bilden $\tilde{O} := \{\tilde{o}_i : i \in I\}$, wobei eine solche Äquivalenzklasse

$$\tilde{o}_i = \{o \in O : \mathsf{gid}(o) = i\} \tag{2.1}$$

als das verteilte Objekt $\tilde{o}_i$ mit globaler Nummer $i \in I$ bezeichnet wird. $o \in \tilde{o}$ heißt lokale Kopie des verteilten Objekts $\tilde{o} \in \tilde{O}$.

Die lokalen Referenzrelationen R^p induzieren eine verteilte Referenzrelation $\tilde{R}$ und damit den verteilten Graphen $\tilde{G} = (\tilde{O}, \tilde{R})$:

$$(\tilde{o}_i, \tilde{o}_j) \in \tilde{R} \quad :\Leftrightarrow \quad \exists\, a \in \tilde{o}_i \quad \exists\, b \in \tilde{o}_j \quad \exists\, p : \quad (a, b) \in R^p. \tag{2.2}$$

Die lokalen Referenzrelationen $R^p \subset O^p \times O^p$ sind nun durch folgende zwei Bedingungen charakterisiert:

1. Das Fehlen eines globalen Adressraumes bedeutet, dass Referenzen über Prozessorgrenzen hinweg nicht zulässig sind; diese müssen innerhalb eines lokalen Adressraums repräsentiert sein, d.h.

$$(\hat{o}_i, \hat{o}_j) \in \hat{R} \quad \Rightarrow \quad \exists\, p \quad \exists\, a, b \in O^p : \mathsf{gid}(a) = i \wedge \mathsf{gid}(b) = j. \tag{2.3}$$

2. Des Weiteren müssen lokale Referenzen einer globalen Referenz entsprechen:

$$\forall\, p : \qquad (a, b) \in R^p \quad \Rightarrow \quad \big(\hat{o}_{\mathsf{gid}(a)}, \hat{o}_{\mathsf{gid}(b)}\big) \in \hat{R}. \tag{2.4}$$

In [3] wird zwar die grundlegende Idee konsistenter verteilter Speicherung ebenso anhand des globalen und verteilten Datengraphs $\hat{G}$ bzw. $\tilde{G}$ beschrieben. Es wird aber nicht untersucht, unter welchen Bedingungen die Isomorphie von $\hat{G}$ und $\tilde{G}$ gezeigt werden kann. Dies wird in dieser Arbeit durch den folgenden Satz nun nachgeholt.

Definition 2.9 (Graph-Homomorphismus) Gegeben seien zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ mit $E_1 \subset V_1 \times V_1$ und $E_2 \subset V_2 \times V_2$. Eine Abbildung

$$\Gamma : V_1 \to V_2$$

heißt Graph-Homomorphismus, wenn sie die Kantenrelation respektiert, d.h. die Bedingung

$$(v, w) \in E_1 \iff \big(\Gamma(v), \Gamma(w)\big) \in E_2$$

erfüllt.

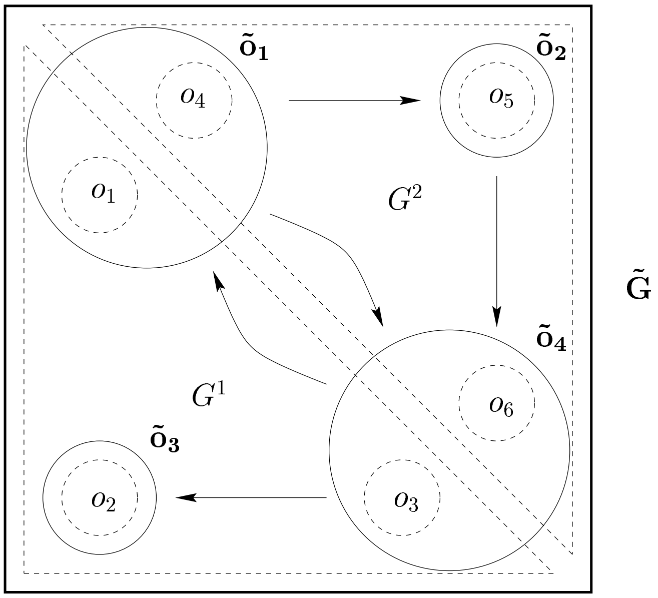


Abbildung 2.5: Beispiel eines verteilten Graphen.

Satz 2.10 (Isomorphie von $\hat{G}$ und $\tilde{G}$) *Gegeben seien der globale Graph $\hat{G} = (\hat{O}, \hat{R})$ sowie die lokalen Graphen $G^p = (O^p, R^p), \quad p = 1, \ldots, P$, welche die Bedingungen (2.3) und (2.4) erfüllen. Dann ist der durch (2.1) und (2.2) induzierte Graph $\tilde{G} = (\tilde{O}, \tilde{R})$ isomorph zu $\hat{G}$.*

Beweis: Definiere $\Gamma_1 : \hat{O} \to \tilde{O}$ durch $\Gamma_1(\hat{o}_i) := \tilde{o}_i$. Zeige nun, dass Γ_1 ein Graph-Homomorphismus ist, d.h.

$$(a, b) \in \hat{R} \quad \Longleftrightarrow \quad \big(\Gamma_1(a), \Gamma_1(b)\big) \in \tilde{R} \ .$$

1. Sei $(\hat{o}_i, \hat{o}_j) \in \hat{R}$. Dann folgt mit (2.3), dass es einen Prozessor p und dort eine lokale Referenz $(a, b) \in R^p$ gibt mit

$$\mathsf{gid}(a) = i, \quad \mathsf{gid}(b) = j,$$

   es gilt also $a \in \tilde{o}_i$ und $b \in \tilde{o}_j$. Nach der Definition (2.2) von $\tilde{R}$ ist also $(\tilde{o}_i, \tilde{o}_j) \in \tilde{R}$.

2. Seien nun $\hat{o}_i, \hat{o}_j \in \hat{O}$ mit $\big(\Gamma_1(\hat{o}_i), \Gamma_1(\hat{o}_j)\big) = (\tilde{o}_i, \tilde{o}_j) \in \tilde{R}$. Wegen (2.2) gibt es Objektkopien $a \in \tilde{o}_i$ und $b \in \tilde{o}_j$ sowie einen Prozessor p mit $(a, b) \in R^p$. Mit (2.4) gilt also

$$(\hat{o}_{\mathsf{gid}(a)}, \hat{o}_{\mathsf{gid}(b)}) \in \hat{R} \ ,$$

   wegen $\mathsf{gid}(a) = i, \mathsf{gid}(b) = j$ folglich $(\hat{o}_i, \hat{o}_j) \in \hat{R}$.

Somit ist Γ_1 ein Graph-Homomorphismus.

Wegen der Surjektivität von gid gilt $|\hat{O}| = |\tilde{O}|$, also ist die Abbildung $\Gamma_2 : \tilde{O} \to \hat{O}, \quad \Gamma_2(\tilde{o}_i) := \hat{o}_i$ wohldefiniert. Analog zu Γ_1 ist Γ_2 ebenfalls ein Graph-Homomorphismus. Es gilt zudem $\Gamma_1^{-1} = \Gamma_2$, und somit sind Γ_1, Γ_2 Graph-Isomorphismen.
$\Rightarrow \quad \hat{O} \cong \tilde{O}$. □

Zusätzlich befolgt DDD zwei weitere Regeln bei der verteilten Speicherung von Objekten:

Definition 2.11 (konsistent definiert, effizient gespeichert)

- Die Relationen R^p heißen *konsistent definiert*, wenn gilt

$$\begin{aligned} &\forall\,(\tilde{o}_1, \tilde{o}_2) \in \tilde{R} \quad \forall\, a \in \tilde{o}_1 \quad \forall\, b \in \tilde{o}_2 : \\ &\qquad a, b \in O^p \quad \Rightarrow \quad (a, b) \in R^p \ . \end{aligned}$$

  Das bedeutet, dass eine globale Referenz auf allen Prozessoren, die entsprechende Objektkopien bereithalten, als lokale Referenz repräsentiert sein muss.

- Die Objekte O^p heißen *effizient gespeichert*, wenn gilt

$$\forall\, p \quad \forall\, a, b \in O^p : \quad \big(\mathsf{gid}(a) = \mathsf{gid}(b) \quad \Rightarrow \quad a = b\big) \ .$$

  Jedes globale Objekt wird also auf jedem Prozessor höchstens einmal gespeichert.

Bemerkung 2.12 Wenn die lokalen Graphen $G^p = (O^p, R^p)$ konsistent definiert und effizient gespeichert sind, dann sind für alle Prozessoren p die Abbildungen

$$\Gamma_p : O^p \to \tilde{O}, \quad \Gamma_p(o) := \tilde{o}_{\mathsf{gid}(o)}$$

Graph-Monomorphismen, d.h. Einbettungen der lokalen Graphen G^p in den verteilten Graphen $\tilde{G}$.

Die konsistente Verwaltung der Daten wird vollständig von DDD übernommen. Diese Arbeit wird hauptsächlich von vier Modulen geleistet: dem Verwaltungsmodul, dem Transfermodul, dem Interface-Modul und dem Identifikationsmodul.

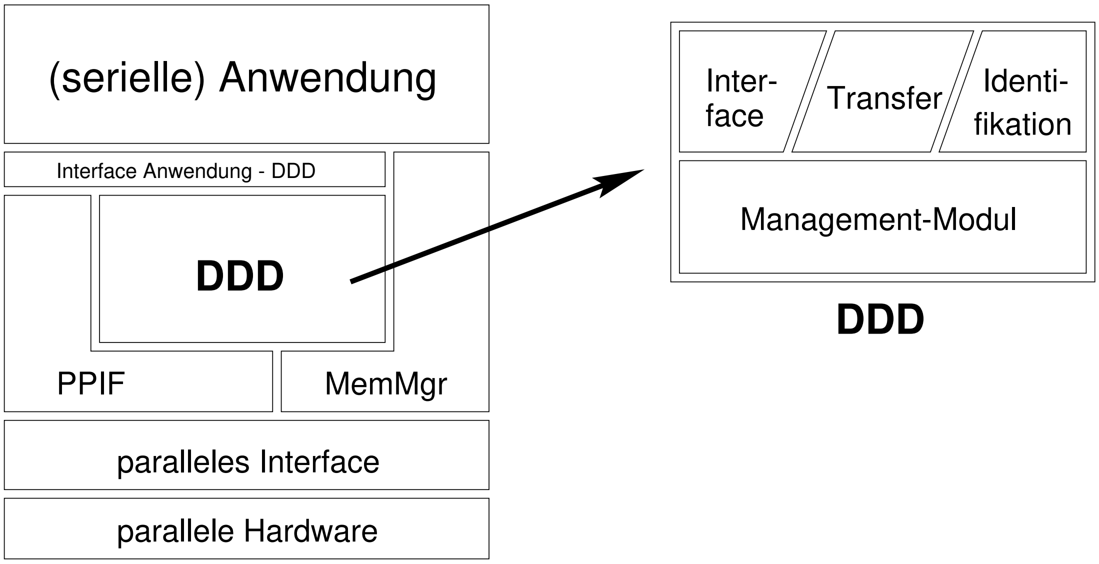


Abbildung 2.6: Die Struktur von DDD (nach [3]).

2.4.2 Verwaltungsfunktionen

Die Datentypen, welche verteilt verwaltet werden sollen, müssen bei DDD als *DDD-Datentyp* spezifiziert und registriert werden. Die Registrierung des Datentyps erfolgt zur Laufzeit, wobei der neue Typ zuerst bei der DDD-Umgebung angemeldet (`DDD_TypeDeclare`) und anschließend definiert wird (`DDD_TypeDefine`). Durch `DDD_TypeDefine` wird DDD die Struktur des Datentyps mitgeteilt, d.h. die Position und Größe der einzelnen Datenkomponenten und Referenzen.

Referenzen auf ein anderes verteiltes Objekt werden durch `OBJPTR` gekennzeichnet. Die Datenkomponenten können entweder als global (`GDATA`) oder lokal (`LDATA`) gültig spezifiziert werden. Wird eine Datenkomponente durch den Benutzer nicht an DDD mitgeteilt, so wird dieser Speicherbereich als `LDATA` behandelt.

Global gültige Datenkomponenten besitzen bei allen Objektkopien eines verteilten Objektes den identischen Wert, lokal gültige können dagegen auf unterschiedlichen Prozessoren unterschiedliche Werte annehmen. Die Koordinaten eines Knotens sind z.B. global gültig; eine Markierung, ob der Knoten gelöscht werden soll, besitzt dagegen nur lokale Gültigkeit: Es kann sein, dass die lokale Objektkopie nur auf einem bestimmten Prozessor wegfallen soll, die übrigen Prozessoren, die auch Objektkopien des globalen verteilten Objekts speichern, diese jedoch beibehalten sollen.

Als Beispiel ist die Geometriedatenstruktur `EdgeCL`, die in DROPS eine Kante beschreibt, in Tabelle 2.1 angegeben.

Die den DDD-Datentypen zugeordneten *lokalen Objekte* werden in zwei unter-

Tabelle 2.1: Die Datenstruktur `EdgeCL` als DDD-Typ.

Datenstruktur `EdgeCL`			
Datentyp	Name	Bedeutung	Art
`DDD_HEADER`	`_ddd`	DDD-Header	`DDDHDR`
`(VertexCL*)[2]`	`_Vertices`	Feld von Endknotenzeigern	`OBJPTR`
`VertexCL*`	`_MidVertex`	Zeiger auf Mittelpunktsknoten	`OBJPTR`
`BndIdxT[2]`	`_Bnd`	Feld von Randindizes	`GDATA`
`short int`	`_MFR`	Kantenzähler, verteilt gespeichert	`LDATA`
`short int`	`_AccMFR`	akkumulierter Kantenzähler	`GDATA`
`bool`	`_RemoveMark`	Entfernungsmarkierung	`LDATA`

schiedliche Klassen unterteilt: *DDD-Objekte* und *registrierte Datenobjekte*. Zu welcher Klasse ein Objekt gehört, ist bereits durch den DDD-Datentyp festgelegt.

- Jedes *DDD-Objekt* enthält eine `DDD_HEADER`-Datenstruktur, die alle Laufzeitinformationen des DDD-Laufzeitsystems enthält. Der DDD-Header besteht aus der globalen Nummer (gid) des Objekts, dem DDD-Datentyp (type), zwei weiteren Eigenschaftsmerkmalen „Priorität“ (prio) und „Attribut“ (attr) sowie internen Daten zur Adressierung in den Kopplungslisten. Die Position des DDD-Headers wird bei Definition des Datentyps mit `DDDHDR` gekennzeichnet.

- Ein *registriertes Datenobjekt* ist dadurch gekennzeichnet, dass sein Typ als DDD-Typ registriert ist, es aber keinen DDD-Header enthält. Damit enthält es keine Informationen über sein verteiltes Objekt; von DDD verwaltete Referenzen (`OBJPTR`) auf lokale registrierte Datenobjekte sind deswegen nicht möglich. Registrierte Datenobjekte können aber ihrerseits Referenzen auf lokale DDD-Objekte enthalten, wobei diese Referenzen durch DDD konsistent verwaltet werden. Registrierte Datenobjekte erfordern somit keinen zusätzlichen Speicheraufwand, der gerade bei kleinen Objekten stark ins Gewicht fallen würde; dafür müssen gewisse Einschränkungen in Kauf genommen werden.

Bei Erzeugung neuer DDD-Objekte müssen diese mittels `DDD_HdrConstructor` in das DDD-Laufzeitsystem eingebunden werden. Werden sie wieder gelöscht, so werden sie mit `DDD_HdrDestructor` dort wieder abgemeldet.

2.4.3 Transferfunktionen

Mithilfe dieser Funktionen können DDD-Objektkopien gelöscht oder auf andere Prozessoren kopiert werden. Die entsprechenden Befehle heißen `DDD_Xfer-`

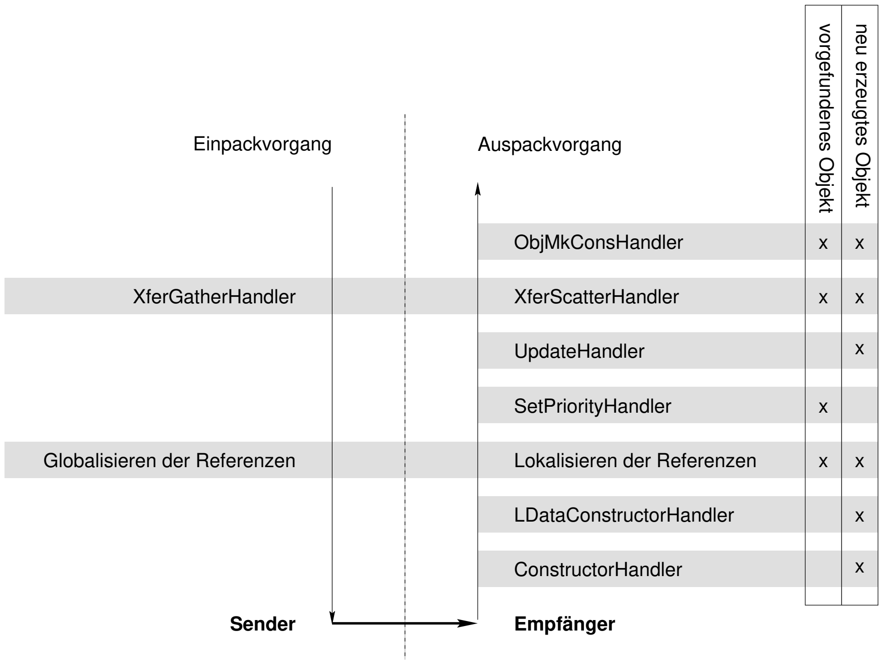


Abbildung 2.7: Übersicht der Handlerfunktionen beim Transfer (nach [3]). Die Tabelle rechts gibt an, unter welchen Bedingungen der jeweilige Handler beim Auspackvorgang aufgerufen wird.

`CopyObj` und `DDD_XferDeleteObj`. Dabei werden Referenzen auf andere Objekte (`OBJPTR`) von DDD automatisch angepasst, außerdem wird das Packen und Entpacken der Daten auf Sender- bzw. Empfängerseite von DDD abgewickelt. Dies ist möglich, da DDD die Struktur von DDD-Objekten vollständig aus der Typdefinition (`DDD_TypeDefine`) bekannt ist.

Zu den Transfer-Anweisungen zählt ebenfalls der Befehl `DDD_PrioChange`, mit dem die Priorität eines lokalen DDD-Objekts verändert werden kann. Dabei wird zwar das Objekt nicht auf einen anderen Prozessor verschoben, gleichwohl müssen Prozessoren, die andere Objektkopien des verteilten Objektes speichern, von dieser Änderung in Kenntnis gesetzt werden.

Die Transfer-Anweisungen werden von `DDD_XferBegin()` und `DDD_XferEnd()` eingeschlossen. Der eigentliche Datentransfer findet erst durch `DDD_XferEnd()` statt, so dass die gesamte Kommunikation zwischen zwei Prozessoren zusammengefasst werden kann. Dies ist optimal (im Sinne der benötigten Anzahl gesendeter Nachrichten), weil auf diese Weise lange Aufsetzzeiten vermieden werden. Die beim Transfer nötigen Operationen auf Sender- und Empfängerseite können für jeden DDD-Typ durch vielfältige Handler-Funktionen auf das jeweilige Objekt angepasst werden (siehe Abb. 2.7).

Das Transfermodul wird in DROPS zur Verteilung des Anfangsgitters und zur

Lastmigration nach einer Lastbalancierung eingesetzt.

2.4.4 Interface-Modul

Interfaces sind Mengen von verteilten Objekten eines Typs, die auf den Prozessorrändern liegen und für die gemeinsame synchronisierende Kommunikation durchgeführt werden kann. Nach der einmaligen Definition eines Interfaces durch `DDD_IFDefine` kann dieses jederzeit zu synchonisierender Kommunikation benutzt werden. Die zu verschickenden Daten werden dabei gebündelt, so dass von Prozessor p zu Prozessor $q \neq p$ höchstens eine Nachricht verschickt wird.

Bei topologischen Änderungen, z.B. durch das Transfermodul, werden Interfaces automatisch von DDD angepasst. `DDD_IFDisplay` zeigt Informationen über ein bestimmtes Interface an, z.B. die Anzahl der enthaltenen verteilten Objekte. `DDD_IFDisplayAll` zeigt alle bis zu diesem Zeitpunk in der DDD-Umgebung definierten Interfaces an. In DROPS werden Interfaces z.B. zur Akkumulation von verteilten Vektoren eingesetzt.

Ein Paar von Gather-/Scatter-Handlern bestimmt dabei, was die jeweilige Interface-Kommunikation leisten soll: Der Gather-Handler sammelt die zu kommunizierenden Daten auf der jeweiligen Senderseite von jedem Objekt ein, das zum Interface gehört. Der Scatter-Handler beschreibt, wie die empfangenen Daten in den entsprechenden Interface-Objekten der Empfängerseite verarbeitet werden sollen.

Die Kommunikation über ein Interface kann in nur eine Richtung (`DDD_IFOneway`) oder in beide Richtungen (`DDD_IFExchange`) ausgeführt werden. Zusätzlich besteht durch `DDD_IFExecLocal` die Möglichkeit, auf jedes Objekt eines Interfaces, welches lokal auf dem Prozessor gespeichert ist, eine durch einen Execute-Handler definierte Funktion anzuwenden. Bei Ausführung von `DDD_IFExecLocal` findet natürlich keine Kommunikation zwischen den Prozessoren statt.

Durch Angabe von Attributen (`attr`) kann die Kommunikation über ein Interface genauer spezifiziert werden. Z.B. kann die Interface-Kommunikation nach Gitterleveln getrennt ausgeführt werden, wenn das Level eines Objektes in `attr` gespeichert wird. Diese attributgesteuerte Interface-Kommunikation wird durch die Befehle `DDD_IFAOneway`, `DDD_IFAExchange` bzw. `DDD_IFAExecLocal` angestoßen.

2.4.5 Identifikationsmodul

Sollen lokale Objekte auf verschiedenen Prozessoren zu einem verteilten Objekt zusammengefasst werden, so geschieht dies durch Identifikation mittels Identifikationstupeln, die aus ganzen Zahlen, Zeichenketten oder auch anderen verteilten

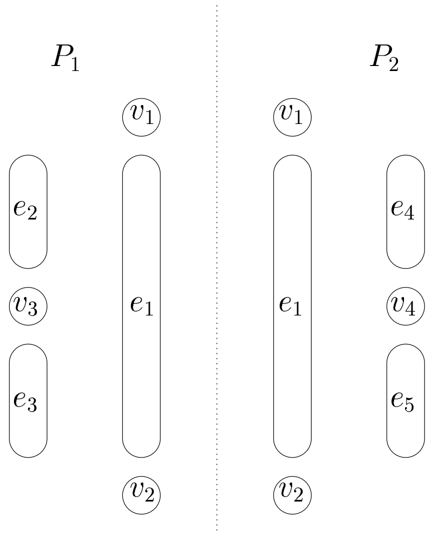


Abbildung 2.8: Beispiel für einen Identifikationsvorgang.

Objekten, die beiden Prozessoren bereits bekannt sind, bestehen können. Die entsprechenden Identifikationskommandos lauten `DDD_IdentifyNumber`, `DDD_IdentifyString` und `DDD_IdentifyObject`. Die auf den beiden Prozessoren abgesetzten Identifikationskommandos müssen sich jeweils paarweise entsprechen.

Alle Identifikationskommandos einer Identifikationsphase müssen von den beiden Befehlen `DDD_IdentifyBegin()` und `DDD_IdentifyEnd()` eingeschlossen sein. Wird `DDD_IdentifyEnd()` aufgerufen, so wird damit das Ende der Identifikationsphase angezeigt, und alle erforderlichen Informationen zwischen den Prozessoren werden ausgetauscht. Der zur Identifikation benötigte Kommunikationsaufwand ist wie beim Transfer- und Interface-Modul optimal.

Die Identifikation wird in DROPS zum einen bei der Erzeugung eines verteilten Anfangsgitters eingesetzt, zum anderen bei der Erzeugung neuer verteilter Objekte durch den Verfeinerungsalgorithmus benötigt. Auch für das Wiederaufsetzen nach einer abgebrochenen Rechnung wird Identifikation benötigt.

Als Beispiel für einen Identifikationsvorgang wird die Verfeinerung einer verteilten Kante vorgestellt. Die Kante e_1 und ihre Endknoten v_1 und v_2 seien auf den Prozessoren P_1 und P_2 gespeichert. Aus der Sicht von DDD stellen e_1, v_1, v_2 verteilte Objekte mit lokalen Objektkopien auf den Prozessoren P_1, P_2 dar.

Durch Verfeinerung von e_1 entstehen nun auf beiden Prozessoren jeweils zwei Teilkanten und ein Mittelpunktsknoten. Auf Prozessor P_1 seien diese mit e_2, e_3, v_3 und auf Prozessor P_2 mit e_4, e_5, v_4 bezeichnet (siehe Abb. 2.8). Für DDD handelt es sich dabei zunächst um völlig verschiedene Objekte. Durch eine Identifikationsphase müssen nun die entsprechenden Objekte über Identifikationstupel miteinander identifiziert werden, um sie zu einem verteilten Objekt zu verschmelzen:

- Die Knotenobjekte v_3, v_4 besitzen als gemeinsames Merkmal die Kante e_1, deren Mittelpunktsknoten sie darstellen. Deswegen wird (e_1) als Identifikationstupel benutzt.
- Die Kantenobjekte e_2, e_4 repräsentieren beide eine Teilkante von e_1. Um sie von der anderen Teilkante zu unterscheiden, wird zusätzlich die Nummer des anliegenden Knotens v_1 herangezogen. Als Identifikationstupel ergibt sich somit $(e_1, 1)$. Alternativ hätte man auch (e_1, v_1) verwenden können.
- Die Kantenobjekte e_3, e_5 repräsentieren beide eine weitere Teilkante von e_1. Um sie von der anderen Teilkante zu unterscheiden, wird zusätzlich die Nummer des anliegenden Knotens v_2 herangezogen. Als Identifikationstupel ergibt sich somit $(e_1, 2)$. Alternativ hätte man auch (e_1, v_2) verwenden können.

Dazu werden auf beiden Prozessoren jeweils folgende Identifikationskommandos ausgeführt:

Prozessor P_1:	Prozessor P_2:
`DDD_IdentifyBegin();`	`DDD_IdentifyBegin();`
`DDD_IdentifyObject(` v_3, P_2, e_1 `);`	`DDD_IdentifyObject(` e_5, P_1, e_1 `);` `DDD_IdentifyNumber(` $e_5, P_1, 2$ `);`
`DDD_IdentifyObject(` e_2, P_2, e_1 `);` `DDD_IdentifyNumber(` $e_2, P_2, 1$ `);`	`DDD_IdentifyObject(` e_4, P_1, e_1 `);` `DDD_IdentifyNumber(` $e_4, P_1, 1$ `);`
`DDD_IdentifyObject(` e_3, P_2, e_1 `);` `DDD_IdentifyNumber(` $e_3, P_2, 2$ `);`	`DDD_IdentifyObject(` v_4, P_1, e_1 `);`
`DDD_IdentifyEnd();`	`DDD_IdentifyEnd();`

Die drei Parameter der Identifikationskommandos `DDD_IdentifyXXX(` $o_{\mathrm{loc}}, p, \lambda$ `)` haben dabei folgende Bedeutung: Der erste Parameter o_{loc} bezeichnet das lokal gespeicherte, zu identifizierende Objekt. Der zweite Parameter p gibt den Prozessor an, auf dem sich das andere Objekt befindet, mit dem o_{loc} identifiziert werden soll. Der dritte Parameter λ stellt schließlich das zur Identifikation benutzte Identifikationsmerkmal dar. Beachte, dass die Reihenfolge, in der die Identifikationsbefehle gegeben werden, unerheblich ist.

Kapitel 3

Adaptivität und Verfeinerung

Die numerische Lösung partieller Differentialgleichungen mittels Finite-Elemente-Verfahren läuft auf die iterative Lösung großer dünnbesetzter Gleichungssysteme hinaus, deren Aufwand größenordnungsmäßig von der Anzahl der Unbekannten n abhängt. Um die Lösung u einer partiellen Differentialgleichung mit einer vorher festgelegten Genauigkeit effizient zu bestimmen, ist es deswegen eine sinnvolle Forderung, nur so viele Unbekannte zu verwenden, wie dafür nötig sind. Die Verteilung der Unbekannten muss also problemangepasst erfolgen, d.h. viele Unbekannte bzw. eine feine Triangulierung in Bereichen, in denen sich u stark ändert, wenige Unbekannte bzw. eine grobe Triangulierung in Bereichen schwacher Änderung von u. Diese Idee wird als *Adaptivität* bezeichnet.

3.1 Der adaptive Zyklus

Die Umsetzung dieser Idee mündet in folgenden adaptiven Zyklus, welcher in Abbildung 3.1 dargestellt ist: Das Problem wird auf einer Anfangstriangulierung $\mathcal{T}_0$ diskretisiert und das entstehende Gleichungssystem gelöst. Anschließend beurteilt ein Fehlerschätzer die Qualität der erhaltenen Lösung; basierend auf der elementweisen Schätzung werden im Zuge einer Markierungsstrategie bestimmte Tetraeder durch entsprechende Markierungen zur Verfeinerung/Entfeinerung vorgesehen. Diese Markierungen werden durch einen Verfeinerungsalgorithmus[1] umgesetzt, der durch Verfeinerung von $\mathcal{T}_0$ eine neue Triangulierung $\mathcal{T}_1$ erzeugt. Dieser Zyklus wird solange wiederholt, bis der Fehlerschätzer die diskrete Lösung u akzeptiert.

Die durch sukzessive Verfeinerung entstehende Hierarchie $(\mathcal{T}_0, \ldots, \mathcal{T}_J)$ ermöglicht den Einsatz von Mehrgitterverfahren, die derzeit zu den modernsten und

[1] Eigentlich müsste es Verfeinerungs-/Entfeinerungsalgorithmus heißen, der Einfachheit halber wird im Folgenden stellvertretend der Begriff „Verfeinerungsalgorithmus“ verwendet.

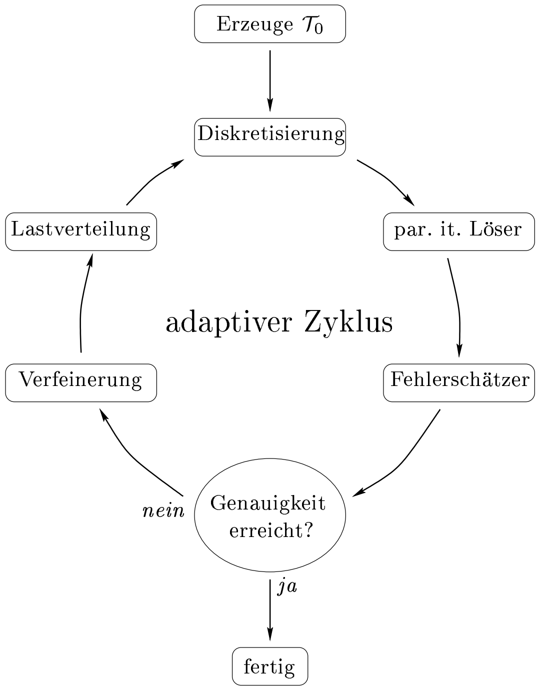


Abbildung 3.1: Ein adaptiver Zyklus zur parallelen numerischen Lösung partieller Differentialgleichungen.

schnellsten Lösern der aus der Diskretisierung von partiellen Differentialgleichungen stammenden linearen Gleichungssysteme zählen. Im Rahmen dieser Arbeit wurde wegen des großen Parallelisierungsaufwandes kein solches Mehrgitterverfahren realisiert, dies soll in naher Zukunft geschehen und der Funktionalität von DROPS hinzugefügt werden. Der Untersuchungsansatz dieser Arbeit beschränkt sich auf die Lösung des Problems auf der feinsten Triangulierung $\mathcal{T} = \mathcal{T}_J$ mithilfe eines parallelen CG-Lösers (vgl. Abschnitt 2.3.1).

Der Einsatz adaptiver Verfeinerungstechniken führt im Vergleich zu uniformer (d.h. globaler) Verfeinerung zu einer mitunter drastischen Reduktion der Unbekanntenzahlen, erfordert dafür aber auch einen vielfach erhöhten Implementierungsaufwand. Neben der Bereitstellung komplexer Datenstrukturen für die Speicherung der Gitterhierarchie müssen Fehlerschätzungs- und Markierungstechniken implementiert und vor allem ein adaptiver Verfeinerungsalgorithmus entwickelt werden. Im Falle eines parallelen Programmes erfordert die Adaptivität zwischenzeitliche dynamische Lastbalancierung, um die geänderte Rechenlast gleichmäßig auf die Prozessoren aufzuteilen – uniforme Verfeinerung benötigt dagegen nur eine einmalige gleichmäßige Verteilung der Anfangstriangulierung, da sich bei uniformer Verfeinerung die relative Rechenlast jedes einzelnen Prozessors nicht ändert.

3.2 Adaptive Verfeinerung

Die durch den Verfeinerungsalgorithmus erzeugten Triangulierungshierarchien sollten folgenden zwei Anforderungen genügen: Die einzelnen Triangulierungen sollen *konsistent* sein (d.h. keine hängenden Knoten enthalten, vgl. Definition 1.6), und die erzeugten Triangulierungshierarchien sollen *stabil* sein (d.h. nicht beliebig entarten, vgl. Definition 1.10). Während die Konsistenz vor allem aus algorithmischen Gesichtspunkten erwogen wird (keine aufwändige Behandlung hängender Knoten während der Diskretisierung), hat die Stabilität entscheidenden Einfluss auf das Konvergenzverhalten der Diskretisierung (siehe Abschnitt 1.3.3) und die Kondition der Steifigkeitsmatrix und somit auf die Konvergenzgeschwindigkeit iterativer Lösungsverfahren.

Der Verfeinerungsalgorithmus erhält als Eingabe eine Hierarchie

$$(\mathcal{T}_0, \ldots, \mathcal{T}_J)$$

von konsistenten Triangulierungen $\mathcal{T}_k$, die sukzessive Verfeinerungen der Triangulierung $\mathcal{T}_0$ sind (vgl. Definition 3.4). Der Begriff der Triangulierungshierarchie wird in Abschnitt 3.3 genauer definiert. Die Blätter dieser Triangulierungshierarchie, d.h. alle unverfeinerten Tetraeder, seien durch einen Markierer jeweils für *Verfeinerung*, *Nichtverfeinerung* oder *Entfernung* markiert. Ausgabe ist eine gemäß diesen Markierungen gebildete Hierarchie

$$(\mathcal{T}'_0, \ldots, \mathcal{T}'_{J'})$$

konsistenter Triangulierungen $\mathcal{T}'_k$, welche ebenfalls sukzessive Verfeinerungen von $\mathcal{T}'_0 = \mathcal{T}_0$ sind. **Beachte aber, dass die Ausgabetriangulierungen $\mathcal{T}'_k$ keinesfalls** (bis auf $k = 0$) **mit den entsprechenden Eingabetriangulierungen $\mathcal{T}_k$ übereinzustimmen noch Verfeinerungen von diesen zu sein brauchen!** Das Beispiel in Abbildung 3.2 soll genau dies verdeutlichen: Beachte, dass in diesem Beispiel $\mathcal{T}_1 \neq \mathcal{T}'_1$ gilt!

Die Anzahl der Triangulierungen ändert sich um höchstens eine, d.h.

$$|J' - J| \leq 1.$$

Beachte, dass insbesondere auch $J' = J - 1$ möglich ist, z.B. im Fall globaler Entfeinerung.

Der in DROPS implementierte adaptive Verfeinerungsalgorithmus reiht sich in die Klasse der sogenannten „Rot-/Grün-Verfeinerungsalgorithmen“ ein, da er wie diese aus folgenden grundlegenden Elementen besteht:

1. einer stabilen *regulären* („roten“) Verfeinerungsstrategie für die vom Markierer zur Verfeinerung vorgesehenen Tetraeder,

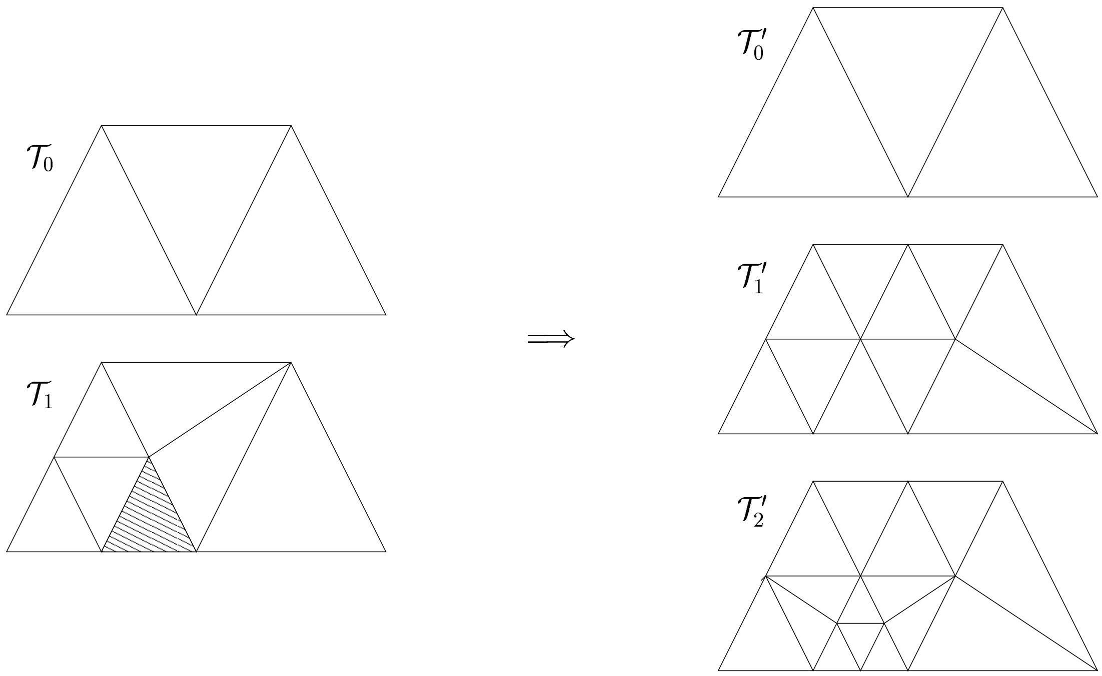


Abbildung 3.2: Beispiel für Triangulierungshierarchien als Eingabe und Ausgabe des Verfeinerungsalgorithmus. In der Eingabehierarchie $(\mathcal{T}_0, \mathcal{T}_1)$ sei ein Dreieck zur Verfeinerung markiert. Die Triangulierungshierarchie $(\mathcal{T}_0', \mathcal{T}_1', \mathcal{T}_2')$ auf der rechten Seite stellt die entsprechende Ausgabe des Verfeinerungsalgorithmus dar. Beachte $\mathcal{T}_1 \neq \mathcal{T}_1'$!

2. zusätzlichen *irregulären* („grünen“) Verfeinerungsregeln, welche der Konsistenzerhaltung dienen und Tetraeder aufeinanderfolgender Verfeinerungsstufen miteinander verbinden („grüner Abschluss“),

3. einem globalen Algorithmus, der reguläre und irreguläre Verfeinerungsregeln derart umsetzt, dass die Konsistenz- und Stabilitätsbedingung gleichermaßen erfüllt werden.

Bevor dieses Konzept anschaulich erläutert wird, muss zunächst der Begriff der Verfeinerung genauer geklärt werden.

Definition 3.1 (Verfeinerung) Für einen Tetraeder T heißt eine Triangulierung $\mathcal{K}(T)$ von T *Verfeinerung* von T, falls $|\mathcal{K}(T)| \geq 2$ und alle Eckpunkte von Tetraedern $T' \in \mathcal{K}(T)$ mit Eckpunkten oder Kantenmittelpunkten von T übereinstimmen. In diesem Fall heißt $T' \in \mathcal{K}(T)$ *Kind* von T, umgekehrt wird aus Sicht des Kindes $T' \in \mathcal{K}(T)$ der Tetraeder T als *Elterntetraeder* $E(T')$ bezeichnet.

Eine Verfeinerung $\mathcal{K}(T)$ von T heißt *regulär*, falls T 8 Kinder besitzt, andernfalls heißt sie *irregulär*.

Bemerkung 3.2 Obige Definition lässt sich auf n-Simplices ($n \geq 2$) verallgemeinern, wenn man in obiger Definition den Begriff „Tetraeder" durch „n-Simplex" ersetzt. Eine Verfeinerung eines n-Simplex S heißt *regulär*, falls S genau 2^n Kinder besitzt.

Definition 3.3 (Kantenverfeinerungsmuster) Sei T ein Tetraeder, dessen Kanten $E_1, \ldots, E_6$ lokal nummeriert seien. Weiterhin sei $\mathcal{K}(T)$ eine Verfeinerung von T. Eine Kante E von T heißt *verfeinert* in $\mathcal{K}(T)$, wenn der Mittelpunktsknoten von E mit einem der Eckpunkte eines Kindtetraeders $T' \in \mathcal{K}(T)$ übereinstimmt. Das 6-Tupel

$$\underline{r} = (r_1, \ldots, r_6) \in \{0,1\}^6$$

heißt Kantenverfeinerungsmuster der Verfeinerung $\mathcal{K}(T)$, falls

$$r_i = \begin{cases} 0 & E_i \text{ ist nicht in } \mathcal{K}(T) \text{ verfeinert} \\ 1 & E_i \text{ ist in } \mathcal{K}(T) \text{ verfeinert} \end{cases} \qquad 1 \leq i \leq 6 \; .$$

Bei einem regulär verfeinerten Tetraeder sind sämtliche Kanten unterteilt. Das andere Extrem bildet ein unverfeinerter Tetraeder, bei dem keine der Kanten verfeinert ist. Alle dazwischenliegenden kombinatorischen Möglichkeiten von verfeinerten Kanten bilden die irregulären Verfeinerungen. Insgesamt kann man also alle Verfeinerungsregeln nach ihrem jeweiligen Kantenverfeinerungsmuster einteilen.

Zu jedem der möglichen $2^6 = 64$ Kantenverfeinerungsmuster existiert eine entsprechende Verfeinerung (wobei die Nichtverfeinerung, die der Verfeinerung keiner Kante entspricht, streng genommen keine Verfeinerung ist). Beachte aber, dass zu manchen Kantenverfeinerungsmustern mehrere verschiedene Verfeinerungen möglich sind (vgl. Abschnitte 3.2.1 und 3.2.2), Eindeutigkeit gibt es hierbei also nicht. Um konsistente Triangulierungen zu erzeugen, muss aber darauf geachtet werden, dass nicht nur die Verfeinerungen angrenzender Kanten, sondern auch die Verfeinerung angrenzender Seitenflächen zueinander passen. Dieses Problem taucht bei Verfeinerungen in 2D nicht auf. Bei Beachtung bestimmter Regeln (vgl. Abschnitt 3.2.2) kann man aber trotzdem mit einem festen Regelsatz von genau 64 Verfeinerungsvorschriften auskommen.

Anhand von 2D-Beispielen soll im Folgenden die Idee der Rot-Grün-Verfeinerung deutlich gemacht werden. Beim Übergang zur Verfeinerung in drei Dimensionen tauchen zwar neue Effekte auf, die bedacht werden müssen; die grundlegenden Ideen sind hierbei aber dieselben, so dass zur ersten Veranschaulichung die zweidimensionale Verfeinerung ihren exemplarischen Zweck erfüllt.

Dazu stelle man sich eine Dreiecks-Triangulierung $\mathcal{T}$ vor, bei der manche Dreiecke $T \in \mathcal{T}$ für Verfeinerung markiert seien (vgl. Abbildung 3.3). Die markierten Dreiecke werden regulär verfeinert (rote Verfeinerung), es entstehen durch Verbindung

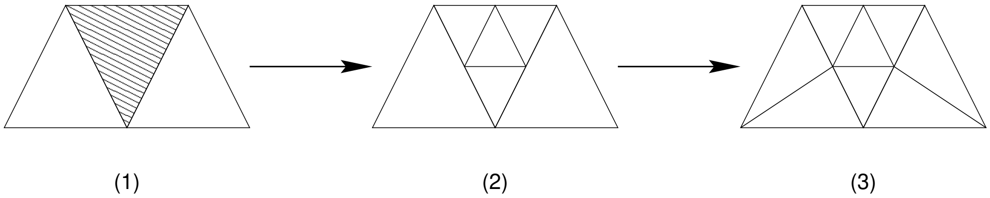


Abbildung 3.3: (1) Triangulierung mit Markierung, (2) rote Verfeinerung und (3) grüner Abschluss.

der Kantenmittelpunkte vier Kinder, die alle ähnlich zu ihrem Elterndreieck sind. Eventuell entstandene hängende Knoten werden durch entsprechende irreguläre Verfeinerungen beseitigt (grüner Abschluss).

Diese Strategie stellt sicher, dass wieder eine Triangulierung entsteht (Dreiecke $T \in \mathcal{T}$ bleiben entweder unverfeinert oder werden durch eine Verfeinerung ersetzt) und dass diese konsistent ist (wegen des grünen Abschlusses, auch in 2D gibt es einen kompletten Satz von Verfeinerungsregeln, wie Abbildung 3.4 zeigt). Diese Überlegungen zur Verfeinerung einer Triangulierung schlagen sich in folgender Definition nieder:

Definition 3.4 (Verfeinerung einer Triangulierung) Eine Triangulierung $\mathcal{T}'$ heißt *Verfeinerung* einer Triangulierung $\mathcal{T} \neq \mathcal{T}'$, wenn für jedes Element $T \in \mathcal{T}$ entweder $T \in \mathcal{T}'$ gilt oder es eine Teilmenge $\mathcal{K}(T) \subset \mathcal{T}'$ gibt, welche eine Verfeinerung von T ist. Im Falle $T \in \mathcal{T}'$ sagt man, dass T beim Übergang von $\mathcal{T}$ auf $\mathcal{T}'$ *unverfeinert* bleibt.

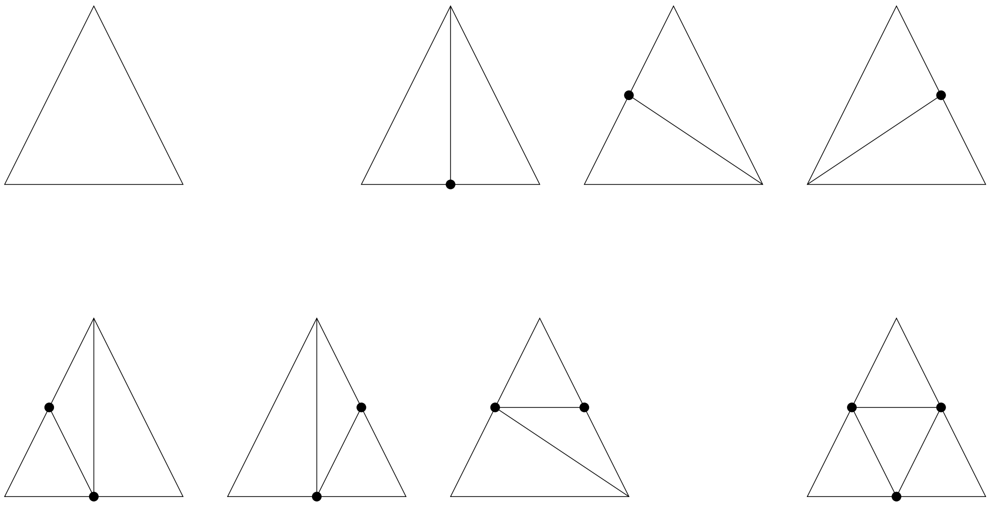


Abbildung 3.4: Ein kompletter Satz von Verfeinerungsregeln im $\mathbb{R}^2$.

Die noch zu erfüllende Forderung der Stabilität kann nur dann eingehalten werden, wenn grüne Kinder (d.h. Kinder, die bei irregulärer Verfeinerung entstanden sind) nicht beliebig oft weiter verfeinert werden dürfen. Würde z.B. das linke grün verfeinerte Dreieck in Abbildung 3.3 immer weiter nach dieser grünen Regel verfeinert werden (weil z.B. fortwährende rote Verfeinerung auf das mittlere Dreieck der Anfangstriangulierung angewendet würde), dann entstünden Teildreiecke, deren einer Innenwinkel immer spitzer würde, die also mehr und mehr entarteten. Um das Entartungsmaß möglichst klein zu halten, wird also gefordert: *Grüne Kinder dürfen nicht weiter verfeinert werden!*

Stattdessen werden grüne Kinder, die (durch eine Markierung oder eine angrenzende Kante) zur Verfeinerung vorgesehen sind, mitsamt ihrer Geschwister durch eine rote Verfeinerung ersetzt, wie in den Abbildungen 3.5 und 3.6 dargestellt. Da durch rote Verfeinerung im 2D-Fall keine neuen Ähnlichkeitsklassen entstehen und irreguläre Elemente nicht weiter verfeinert werden dürfen, liefert das Verfahren Folgen von Triangulierungen, die das Stabilitätskriterium erfüllen. Im Falle der 3D-Verfeinerung führt der Verzicht auf Verfeinerung von irregulären Kindern aus ähnlichen Gründen auf die gewünschte Stabilität.

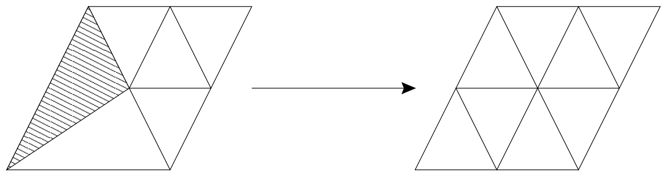


Abbildung 3.5: Ersetzung irregulärer Verfeinerungen aufgrund von Markierungen.

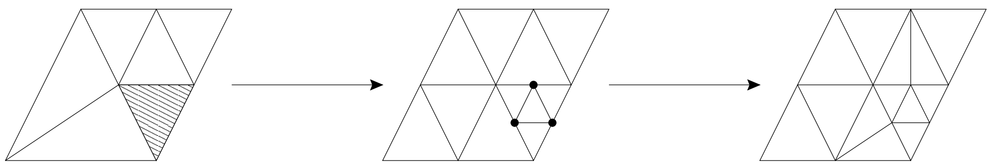


Abbildung 3.6: Ersetzung irregulärer Verfeinerungen aufgrund von verfeinerten Kanten.

3.2.1 Reguläre Verfeinerung von Tetraedern

Zur Beschreibung der regulären Verfeinerungsstrategie sei $T \in \mathbb{R}^3$ ein beliebiger (nichtentarteter) Tetraeder. Durch reguläre Verfeinerung seiner Seitenflächen (im 2D-Sinne) erhält man eine Aufteilung wie in Abb. 3.7 gezeigt.

Diese Abbildung zeigt, dass an den Ecken von T vier Teiltetraeder abgeschnitten werden können, wobei ein Oktaeder zurückbleibt. Im Inneren dieses Oktaeders

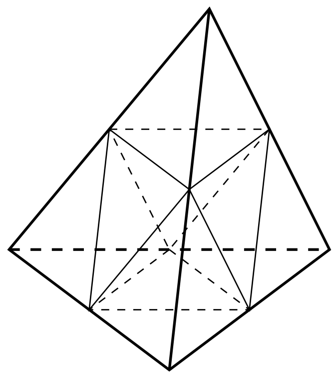


Abbildung 3.7: Rot verfeinerte Seitenflächen.

liegen drei verschiedene Schnittparallelogramme, jeweils von vier Eckpunkten des Oktaeders begrenzt. Je zwei dieser Parallelogramme, die sich in einer Raumdiagonalen schneiden, unterteilen den Oktaeder in vier weitere Teiltetraeder, insgesamt gibt es also drei verschiedene Möglichkeiten der Unterteilung, die jeweils der Wahl einer anderen Diagonalen entsprechen und allesamt T in acht Kinder verfeinern.

Die vier äußeren Teiltetraeder sind ähnlich zu T, was für die vier inneren i.A. nicht mehr gilt. Damit bei sukzessiver roter Verfeinerung nicht eine unbegrenzte Anzahl von Ähnlichkeitsklassen entsteht, kommt der richtigen Wahl der Raumdiagonalen eine zentrale Rolle zu. Folgende in [2] vorgestellte reguläre Verfeinerungsstrategie garantiert, dass bei wiederholter Anwendung nicht mehr als drei Ähnlichkeitsklassen entstehen.

Zur Formulierung der regulären Verfeinerungsstrategie wird folgende Notation eingeführt: $T = [x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}]$ wird als geordnetes Tupel seiner Eckpunkte $x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}$ aufgefasst. Diese totale Ordnung auf den Knoten des Tetraeders[2], *lokale Knotenordnung* genannt, ist entscheidend für die Umsetzung der Verfeinerung.

$$x^{(ij)} := \left(x^{(i)} + x^{(j)}\right)/2, \quad 1 \le i < j \le 4$$

bezeichne den Kantenmittelpunkt der Kante zwischen $x^{(i)}$ und $x^{(j)}$. Die acht Kinder $\mathcal{K}(T) := \{T_1, \dots, T_8\}$ von T sind gegeben durch

$$\begin{aligned} T_1 &:= [x^{(1)}, x^{(12)}, x^{(13)}, x^{(14)}]\,, & T_5 &:= [x^{(12)}, x^{(13)}, x^{(14)}, x^{(24)}]\,,\\ T_2 &:= [x^{(12)}, x^{(2)}, x^{(23)}, x^{(24)}]\,, & T_6 &:= [x^{(12)}, x^{(13)}, x^{(23)}, x^{(24)}]\,,\\ T_3 &:= [x^{(13)}, x^{(23)}, x^{(3)}, x^{(34)}]\,, & T_7 &:= [x^{(13)}, x^{(14)}, x^{(24)}, x^{(34)}]\,,\\ T_4 &:= [x^{(14)}, x^{(24)}, x^{(34)}, x^{(4)}]\,, & T_8 &:= [x^{(13)}, x^{(23)}, x^{(24)}, x^{(34)}]\,. \end{aligned} \tag{3.1}$$

[2]Bei der tatsächlichen Implementierung in DROPS sind nicht nur die vier Knoten, sondern basierend darauf auch die sechs Kanten und vier Seitenflächen eines Tetraeders geordnet abgespeichert, jeweils in einem Feld aus Zeigern auf die jeweiligen Subsimplices.

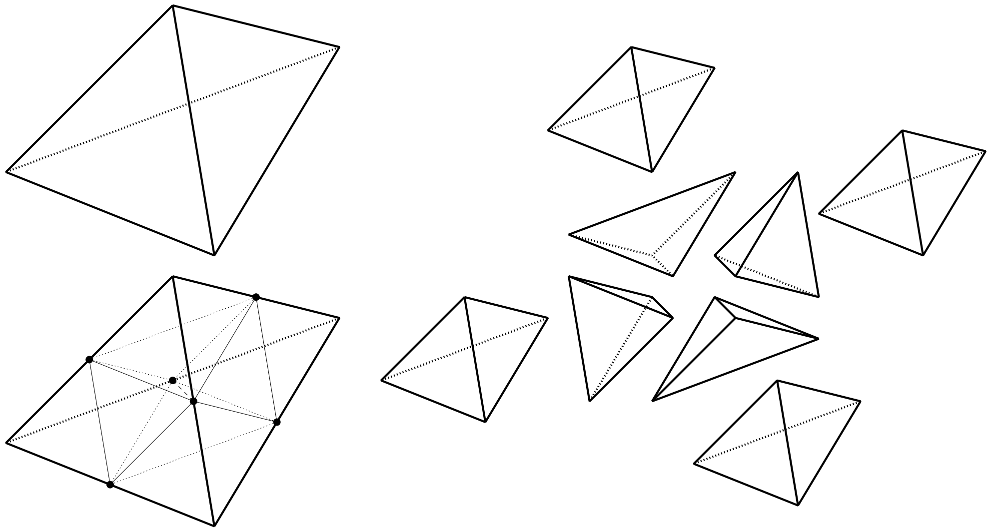


Abbildung 3.8: Ein regulär verfeinerter Tetraeder.

T_1 bis T_4 entsprechen gerade den abgeschnittenen Ecken von T, T_5 bis T_8 den vier inneren Tetraedern, die durch den inneren Oktaeder durch Wahl der Diagonalen zwischen $x^{(13)}$ und $x^{(24)}$ entstanden sind. Die Diagonale und damit die konkrete Verfeinerung wird also implizit durch die Reihenfolge der Knoten bestimmt. Für die Stabilität der regulären Verfeinerungsstrategie (3.1) ist somit die lokale Knotenordnung der Kinder verantwortlich, welche von der Ordnung der Elternknoten bei Anwendung der regulären Verfeinerungsregel vererbt wird.

Bemerkung 3.5 (Stabilität)

1. Die Verfeinerungsregel (3.1) stimmt für $n = 3$ mit dem Algorithmus von Freudenthal überein, welcher in allgemeiner Weise die reguläre Verfeinerung von n-Simplices beschreibt. Bey hat in [2] bewiesen, dass bei wiederholter Anwendung des Algorithmus von Freudenthal auf einen Simplex $T \in \mathbb{R}^n$ höchstens $n!/2$ Ähnlichkeitsklassen entstehen. Insbesondere ergeben sich für die reguläre Verfeinerung (3.1) eines Tetraeders höchstens $3!/2 = 3$ Ähnlichkeitsklassen.

2. Werden grüne Kinder nicht weiter verfeinert, so entstehen stabile Folgen von Triangulierungen. Dies hängt damit zusammen, dass die Anzahl der entstehenden Ähnlichkeitsklassen insgesamt endlich ist: Die Anfangstriangulierung setzt sich aus Tetraedern endlich vieler Ähnlichkeitsklassen zusammen, deren Anzahl sich durch Kinder, die aus regulärer Verfeinerung entstanden sind, höchstens verdreifachen kann. Hinzu kommen noch Kinder aus irregulärer Verfeinerung; werden diese aber nicht weiter verfeinert, entstehen dadurch nur endlich viele neue Ähnlichkeitsklassen. Somit bleibt

das Entartungsmaß in jedem Fall beschränkt, was die Stabilität der entstehenden Triangulierungshierarchie garantiert.

3.2.2 Irreguläre Verfeinerung von Tetraedern

Zur adaptiven Verfeinerung wird neben regulärer auch irreguläre Verfeinerung gebraucht, um Konsistenz herzustellen und damit reguläre Tetraeder verschiedener Level miteinander zu verbinden (grüner Abschluss). Die $2^6 = 64$ möglichen Kantenverfeinerungsmuster verlangen – lässt man das leere und das volle Muster weg – 62 verschiedene grüne Verfeinerungen[3].

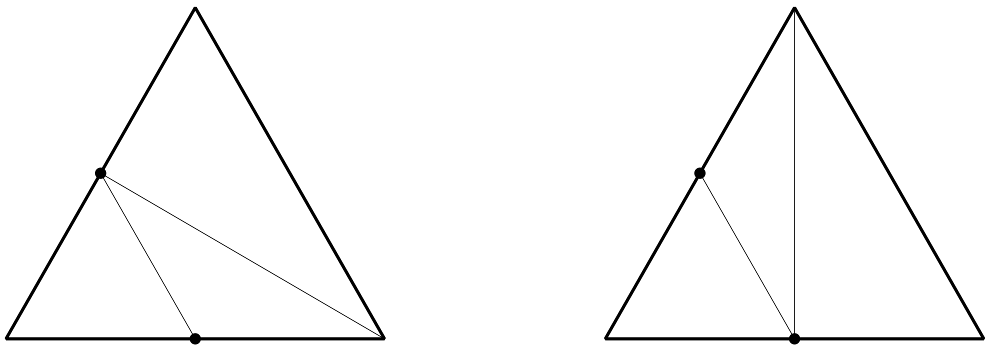


Abbildung 3.9: Nichteindeutige grüne Verfeinerung im $\mathbb{R}^2$.

Wie vorher schon erwähnt wurde, gibt es zu jedem Kantenverfeinerungsmuster eine entsprechende Tetraederverfeinerung, allerdings muss diese im Falle grüner Verfeinerungsregeln nicht eindeutig sein (siehe Abb. 3.9 als entsprechendes 2D-Beispiel). Einfach eine Regel auszuwählen, wäre im 2D-Fall legitim und praktikabel, in drei Dimensionen müssen aber nicht nur die Kanten in konsistenter Weise unterteilt werden, sondern auch das Verfeinerungsmuster auf gemeinsamen Seitenflächen benachbarter Tetraeder übereinstimmen. Die Regeln nun aufgrund dieser Seitenflächenverfeinerungsmuster auszuwählen, ist aber i.A. nicht möglich: Man kann nämlich auf den vier Seitenflächen eines Tetraeders bestimmte Verfeinerungmuster vorgeben, zu denen es keine entsprechende Verfeinerung des Tetraeders gibt. Um diese Situation zu vermeiden, muss eine andere Strategie angewendet werden:

Diese Idee basiert wiederum auf der Einführung einer Ordnung auf den Knoten, diesmal auf allen Knoten der Anfangstriangulierung $\mathcal{T}_0$, und zwar, indem diese

[3] Tatsächlich existieren in DROPS 63 grüne Verfeinerungsregeln, weil das volle Muster hinzugenommen wird. Diese grüne „reguläre“ Verfeinerung erzeugt dieselben Kinder wie die rote Verfeinerungsregel. Der Unterschied besteht alleine darin, dass der Verfeinerungsalgorithmus auf grün verfeinerten Tetraedern die Verfeinerungsregel leicht ersetzen kann, falls ein verändertes Kantenverfeinerungsmuster es erfordert. Auf rot verfeinerten Tetraedern dagegen ändert sich die Verfeinerungsregel nur dann, wenn sämtliche Kinder zur Entfernung markiert sind.

global durchnummeriert werden. Diese Nummerierung induziert eine lokale Ordnung der vier Knoten eines jeden Tetraeders. Offensichtlich ist die Verfeinerung einer Seitenfläche dann und nur dann nicht eindeutig, wenn zwei der drei Kanten zur Verfeinerung vorgesehen sind.

Taucht beim grünen Abschluss also eine Situation wie in Abb. 3.9 auf, so wird jeweils der Knoten mit der kleineren Nummer mit dem Mittelpunkt der gegenüberliegenden Kante verbunden. Diese Vorschrift kann aufgrund der lokalen Knotenordnung auf den einzelnen Tetraedern unabhängig voneinander angewendet werden; da diese lokale Ordnung aber gleichzeitig auf der globalen Nummerierung der Knoten von $\mathcal{T}_0$ basiert, führt sie zu konsistenter Verfeinerung auch auf den Seitenflächen.

Ähnlich wie bei der roten Verfeinerungsregel (3.1) wird bei jeder grünen Regel die lokale Knotenordnung des Elterntetraeders an die Kinder vererbt, und zwar so, dass die Knotenreihenfolge auf einer gemeinsamen Seitenfläche gemäß der jeweiligen lokalen Knotenordnung der angrenzenden Tetraeder identisch ist. Diese Eigenschaft benachbarter Tetraeder wird als *konsistente Knotennummerierung* bezeichnet.

Definition 3.6 (konsistente Knotennummerierung) Zwei Tetraeder T_1 und T_2, $T_1 \neq T_2$, mit gemeinsamer Seitenfläche $F = T_1 \cap T_2$ seien mit einer lokalen Knotenordnung

$$T_l = [x_l^{(1)}, x_l^{(2)}, x_l^{(3)}, x_l^{(4)}], \qquad l = 1, 2$$

versehen. Die Knotenmenge von F ergibt sich als Schnitt der Knotenmengen von T_1 und T_2:

$$\begin{aligned} \{x_1^{(i)} : i = 1, \ldots, 4\} \cap \ \{x_2^{(j)} : j = 1, \ldots, 4\} \quad &=: \quad \{x_1^{(i_1)}, x_1^{(i_2)}, x_1^{(i_3)}\} \\ &= \quad \{x_2^{(j_1)}, x_2^{(j_2)}, x_3^{(j_3)}\} \ . \end{aligned}$$

O.B.d.A. gelte $i_1 < i_2 < i_3$ und $j_1 < j_2 < j_3$. Das Paar (T_1, T_2) heißt *konsistent nummeriert*, falls gilt

$$x_1^{(i_k)} = x_2^{(j_k)}, \quad k = 1, \ldots, 3 \ ,$$

d.h. dass die auf F durch T_1 induzierte Knotenreihenfolge mit der durch T_2 induzierten übereinstimmt.

Eine konsistente Triangulierung $\mathcal{T}$ heißt *konsistent nummeriert*, falls je zwei benachbarte Tetraeder ein konsistent nummeriertes Paar bilden.

Bemerkung 3.7 Mit Hilfe dieser Definition lässt sich zusammenfassend sagen:

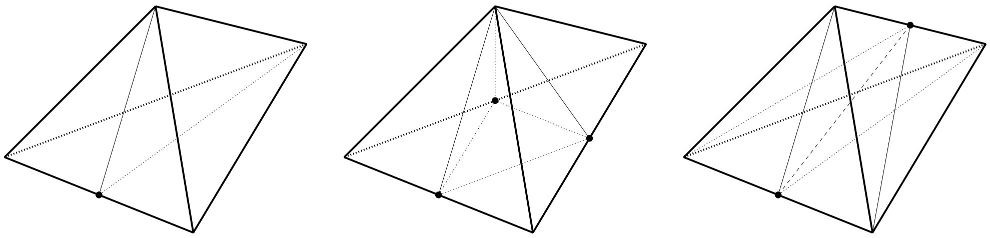


Abbildung 3.10: Irregulär verfeinerte Tetraeder.

1. Durch eine beliebige globale Nummerierung der Knoten von $\mathcal{T}_0$ wird eine lokale Knotenordnung auf den einzelnen Tetraedern induziert, die zur konsistenten Nummerierung von $\mathcal{T}_0$ führt.

2. Die in DROPS implementierten (roten und grünen) Verfeinerungsregeln sind so konstruiert, dass die lokale Knotenordnung des Elterntetraeders eine lokale Knotenordnung der Kinder induziert, welche jeweils die konsistente Nummerierung respektiert.

3. Ist also eine Triangulierung $\mathcal{T}$ konsistent nummeriert, so ist ihre gemäß dieser Regeln enstandene Verfeinerung $\mathcal{T}'$ es ebenfalls. Somit geht die Konsistenz von $\mathcal{T}$ ebenfalls auf $\mathcal{T}'$ über.

4. Das Konzept der konsistenten Nummerierung erlaubt die Anwendung der jeweils adäquaten Regel für jeden einzelnen Tetraeder einzig aufgrund seiner lokalen Knotennummerierung. Das bedeutet, dass keine Information vom Nachbartetraeder gebraucht wird und die Verfeinerung somit tetraederweise unabhängig voneinander stattfinden kann. Diese Eigenschaft ist insbesondere für die Parallelisierung von großem Vorteil.

5. Durch solche lokale Knotenordnungen, welche insgesamt eine konsistente Nummerierung ergeben, ist die Auswahl der Verfeinerungsregel zu einem vorgegebenen Kantenverfeinerungsmuster eindeutig. Deswegen kommt man mit genau 64 verschiedenen Verfeinerungsregeln aus, um konsistente Verfeinerung durchführen zu können.

3.3 Triangulierungshierarchien und Multilevelgitter

Zur Beschreibung des adaptiven Verfeinerungsalgorithmus muss zunächst der Begriff der Triangulierungshierarchie genauer erläutert und die für die Implementation besser geeigneten Multilevelgitter eingeführt werden.

Wenn im Folgenden von Triangulierung gesprochen wird, wird stets stillschweigend angenommen, dass diese konsistent sei (d.h. ohne hängende Knoten).

Eine Anfangstriangulierung und ihre sukzessiven Verfeinerungen bilden zusammen eine Triangulierungshierarchie. Darüber hinaus wird eine zusätzliche Bedingung gefordert, die es ermöglicht, Triangulierungshierarchien gleichbedeutend mit Multilevelgittern (die in Kürze definiert werden) zu betrachten.

Definition 3.8 (Triangulierungshierarchie) Eine endliche Folge von Triangulierungen $\mathcal{H} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ heißt *Triangulierungshierarchie* von Ω, falls gilt:

1. Die Anfangstriangulierung $\mathcal{T}_0$ ist eine konsistente Triangulierung von Ω.
2. Für $0 \leq k < J$ gilt: $\mathcal{T}_{k+1}$ ist Verfeinerung von $\mathcal{T}_k$.
3. Für $0 \leq k < J$ gilt: $T \in \mathcal{T}_k \cap \mathcal{T}_{k+1} \Rightarrow T \in \mathcal{T}_J$.

$\mathcal{T}_0$ wird auch die *gröbste*, $\mathcal{T}_J$ die *feinste* Triangulierung von $\mathcal{H}$ genannt.

Ein Element T von $\mathcal{H}$ heißt *Blatt*, wenn $T \in \mathcal{T}_J$ gilt, d.h. T in $\mathcal{H}$ nicht verfeinert wird.

Ein Element T von $\mathcal{H}$ heißt *regulär*, falls $T \in \mathcal{T}_0$ gilt oder T durch reguläre Verfeinerung entstanden ist. Andernfalls heißt T *irregulär*.

Eine Triangulierungshierarchie $\mathcal{H}$ heißt *regulär*, falls irreguläre Elemente in $\mathcal{H}$ nicht verfeinert werden, also alle irregulären Elemente von $\mathcal{H}$ Blätter sind.

Der in DROPS implementierte Verfeinerungsalgorithmus erzeugt ausschließlich reguläre Triangulierungshierarchien, weil dies Stabilität garantiert (vgl. Abschnitt 3.2.1). Beispiele für reguläre Triangulierungshierarchien sind in Abbildung 3.2 dargestellt. Beachte, dass nach der Verfeinerung insbesondere die Bedingung 3 in vorstehender Definition für alle Stufen k der ausgegebenen Triangulierungshierarchie wieder erfüllt ist. Durch Verfeinerung kommt also nicht einfach nur eine neue Triangulierung $\mathcal{T}_{J+1}$ hinzu, sondern es werden während des Verfeinerungsprozesses auch Triangulierungen auf tieferen Stufen $0 < j < J+1$ modifiziert.

Die Bedingung 3 in vorstehender Definition sagt aus, dass ein beim Übergang von einer auf die nächstfeinere Triangulierung unverfeinert gebliebener Tetraeder auch bei allen nachfolgenden Übergängen unverfeinert bleibt. Aufgrund dieser Eigenschaft kann jedem Element T von $\mathcal{H}$ in wohldefinierter Weise der Index der Triangulierung zugewiesen werden, in welcher T zum ersten Mal vorkommt:

$$\ell(T) := \min\{k : T \in \mathcal{T}_k\}$$

heißt *Level von T*.

Bemerkung 3.9 Wegen Bedingung 3 aus Definition 3.8 beschreibt der Level eines Tetraeders gerade die Anzahl seiner Vorfahren.

Bei der realen Umsetzung des Verfeinerungsalgorithmus auf einem Rechner stellt sich unmittelbar die Frage nach einer geeigneten Datenstruktur, um $\mathcal{H}$ zu repräsentieren. Jede Triangulierung $\mathcal{T}_k$, $k = 0, \ldots, J$ separat abzuspeichern, z.B. als Liste, ist ungeeignet, weil dies mehr Speicher beansprucht als tatsächlich notwendig, da dann Tetraeder, die in mehreren Triangulierungen vorkommen, auch mehrfach abgespeichert würden. Dies stellt gerade im Falle lokal adaptiver Verfeinerung eine ziemliche Speicherplatzverschwendung dar.

Werden dagegen jeweils alle Elemente von $\mathcal{H}$ mit dem Level k zu einer Menge $\mathcal{G}_k$ zusammengefasst, so findet sich jedes Element T aus $\mathcal{H}$ in genau einer dieser Mengen, nämlich $\mathcal{G}_{\ell(T)}$, wieder.

Definition 3.10 (Multilevelgitter)

1. Sei $\mathcal{H} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ eine Triangulierungshierarchie.

$$\mathcal{G}_k := \{T \in \mathcal{T}_k : \ell(T) = k\}, \quad k = 0, \ldots, J$$

   heißt *Gitter* der Stufe k. Die Folge der Gitter

$$\mathcal{M} = \mathcal{M}(\mathcal{H}) = (\mathcal{G}_0, \ldots, \mathcal{G}_J)$$

   wird als das zu $\mathcal{H}$ gehörige *Multilevelgitter* bezeichnet.

2. Sei nun keine explizite Triangulierungshierarchie vorgegeben. Eine Folge

$$\mathcal{M} = (\mathcal{G}_0, \ldots, \mathcal{G}_J)$$

   von Tetraedermengen $\mathcal{G}_k$ heißt *Multilevelgitter*, falls es eine Triangulierungshierarchie $\mathcal{H}$ gibt, so dass $\mathcal{M}$ mit dem zu $\mathcal{H}$ gehörigen Multilevelgitter $\mathcal{M}(\mathcal{H})$ übereinstimmt.

Bemerkung 3.11 Es gilt

$$\mathcal{G}_0 = \mathcal{T}_0, \qquad \mathcal{G}_k = \mathcal{T}_k \setminus \mathcal{T}_{k-1}, \quad 1 \leq k \leq J.$$

Umgekehrt kann aus alleiniger Kenntnis des Multilevelgitters die zugehörige Triangulierungshierarchie eindeutig rekonstruiert werden. Sei $\Omega_k := \bigcup_{T \in \mathcal{G}_k} T$ der durch das Gitter $\mathcal{G}_k$ überdeckte Bereich. Dann gilt

$$\overline{\Omega} = \overline{\Omega}_0 \supset \overline{\Omega}_1 \supset \cdots \supset \overline{\Omega}_J \ ,$$

und die Triangulierungshierarchie $\mathcal{H}$ kann aus dem Multilevelgitter $\mathcal{M}$ auf rekursive Weise folgendermaßen rekonstruiert werden:

$$\mathcal{T}_0 = \mathcal{G}_0, \qquad \mathcal{T}_k = \mathcal{G}_k \cup \{T \in \mathcal{T}_{k-1} : T \not\subset \overline{\Omega}_k\}, \quad 1 \leq k \leq J.$$

Somit sind Multilevelgitter und Triangulierungshierarchien einander eindeutig paarweise zugeordnet. Dies erlaubt es, beide Begriffe äquivalent zu gebrauchen. Alle bisher definierten Eigenschaften von Triangulierungshierarchien lassen sich deswegen analog auf Multilevelgitter übertragen.

Somit stellt das Multilevelgitter die geeignete Form dar, die zugehörige Triangulierungshierarchie speichereffizient als Datenstruktur abzubilden. In DROPS werden daher dieser Idee folgend alle Tetraeder nach ihrem Level geordnet in Listen abgespeichert.

Bemerkung 3.12 (Geometriedatenstruktur von DROPS) Neben den Tetraedern werden auch die Knoten, Kanten und Seitenflächen der Tetraeder levelweise abgespeichert. Jedem dieser *Subsimplices* S wird auf folgende Weise sein Level $\ell(S)$ zugeordnet:

- Die Subsimplices auf Level 0 entsprechen gerade den Subsimplices der Tetraeder auf Level 0.
- Ist S Subsimplex eines Tetraeders T auf Level $k > 0$, aber von keinem der Tetraeder auf Stufe $k-1$, so ist $\ell(S) = k$.

Das bedeutet, dass Subsimplices auf dem Level abgespeichert werden, auf dem sie zuerst auftauchen. Insbesondere werden Subsimplices nur genau einmal abgespeichert, und nicht – wie in Beys AGM^{3D} – auf jedem Level, wo es einen Tetraeder mit diesem Subsimplex gibt.

Zusätzlich sind die Objekte eines Levels untereinander und mit Objekten aus benachbarten Levels in geeigneter Weise durch Zeiger verknüpft:

- Jeder **Tetraeder** hat Zugriff auf:
  - Eckpunkte (4)
  - Kanten (6)
  - Seitenflächen (4)
  - Kinder (≤ 8)
  - Elterntetraeder (≤ 1)

- Jede **Seitenfläche** hat Zugriff auf:
  - angrenzende Tetraeder (≤ 4)
- Jede **Kante** hat Zugriff auf:
  - Eckpunkte (2)
  - Mittelpunkt (≤ 1)
- Von einem **Knoten** aus ist kein Zugriff auf andere Objekte möglich.

Beachte, dass bei bestimmten grünen Regeln Seitenflächen irregulär verfeinerter Kinder mit manchen Seitenflächen ihrer Elterntetraeder zusammenfallen können. In solchen Fällen kann es bis zu vier verschiedene Tetraeder mit gemeinsamer Seitenfläche geben, nämlich zwei benachbarte irregulär verfeinerte Eltern mit ihren grünen Kindern.

Bei paralleler verteilter Speicherung ist von einer Seitenfläche aus natürlich nur Zugriff auf die lokal gespeicherten angrenzenden Tetraeder möglich. Gleiches gilt für Kinder und Eltern von Tetraedern.

3.4 Der serielle Verfeinerungsalgorithmus

Der in DROPS implementierte serielle Verfeinerungsalgorithmus ist stark an den von Bey in [2] beschriebenen Verfeinerungsalgorithmus angelehnt. Es gibt aber auch einige grundlegende Unterschiede:

- Im Gegensatz zu Bey wird ein voller Satz an Verfeinerungsregeln verwendet.
- Die Seitenflächen werden in der Datenstruktur als eigenständige Objekte repräsentiert, was die Darstellung der Nachbarschaftsrelation vereinfacht und auch Finite-Element-Typen mit Unbekannten auf den Seitenflächen zulässt. Beys Datenstruktur berücksichtigt die Seitenflächen dagegen nicht.
- Knoten, Kanten und Seitenflächen tauchen nur auf einem Level auf (vgl. Bem. 3.12) und nicht – wie bei Bey – auf jedem Level, auf dem ein Tetraeder existiert, der diesen Subsimplex referenziert.

Letzteres führt in AGM^{3D} bei grüner Verfeinerung dazu, dass Kanten (und Seitenflächen, wenn diese repräsentiert würden) teilweise doppelt gespeichert werden; Knoten werden sowohl bei roter als auch grüner Verfeinerung mehrfach gespeichert. Weil die Einträge der Vektoren und Matrizen direkt an die Geometriedatenstruktur gekoppelt werden, ist diese Mehrfachspeicherung von Subsimplices in AGM^{3D} notwendig, da sonst die Gleichungssysteme verschiedener Stufen für das Mehrgitterverfahren nicht auf diese Weise abgespeichert werden könnten. Da bei DROPS numerische Daten und Geometriedaten getrennt voneinander gespeichert werden, kann hier auf die mehrfache Speicherung der Subsimplices verzichtet werden.

Der Verfeinerungsalgorithmus erhält als Eingabe ein Multilevelgitter

$$\mathcal{M} = (\mathcal{G}_0, \ldots, \mathcal{G}_J) \ ,$$

dessen Blätter (d.h. alle unverfeinerten Tetraeder) jeweils durch den Fehlerschätzer zur *Verfeinerung*, *Nichtverfeinerung* oder *Entfernung* markiert seien; diese Markierung eines Tetraeders T wird im Algorithmus entsprechend durch $\mathsf{mark}(T) \in$ {Ref, NoRef, Del} notiert. Alle Nichtblätter sind (noch aus dem letzten Verfeinerungsschritt) entsprechend ihrer Verfeinerungsregel markiert.

Ausgabe ist ein Multilevelgitter

$$\mathcal{M}' = (\mathcal{G}'_0, \ldots, \mathcal{G}'_{J'})$$

mit $\mathcal{G}'_0 = \mathcal{G}_0$ und $|J' - J| \leq 1$, welches alle Nichtblätter von $\mathcal{M}$ enthält. Alle Tetraeder von $\mathcal{M}'$ sind entsprechend ihrer Verfeinerungsregel markiert, d.h. jeder Tetraeder T besitzt eine Markierung $\mathsf{mark}(T)$, die mit seinem Kantenverfeinerungsmuster übereinstimmt. Dabei werden die Markierungen des Fehlerschätzers auf den Blättern von $\mathcal{M}$ folgendermaßen umgesetzt:

(a) Reguläre Blätter T mit $\mathsf{mark}(T)$ =Ref werden regulär verfeinert.

(b) Irreguläre Blätter T mit $\mathsf{mark}(T)$ =Ref werden durch reguläre ersetzt.

(c) Sind alle Kinder eines verfeinerten Tetraeders zur Entfernung (Del) markiert, so wird die Verfeinerung – wenn möglich – rückgängig gemacht.

Das bedeutet, dass auch solche Blätter, welche die Markierung NoRef oder Del tragen, unter Umständen zur Konsistenzerhaltung verfeinert werden, wenn ein grüner Abschluss dies nötig macht.

Da der grüne Abschluss alleine aufgrund des Kantenverfeinerungsmusters eindeutig bestimmt werden kann, muss für jede Kante effizient (d.h. insbesondere ohne Betrachtung aller an dieser Kante liegenden Tetraeder) entschieden werden können, ob sie zur Verfeinerung vorgesehen ist oder nicht. Dazu wird jede Kante mit einem Zähler versehen, welcher alle angrenzenden Elemente zählt, die zur regulären Verfeinerung vorgesehen sind. Eine Kante ist also genau dann zur Verfeinerung vorgesehen, wenn ihr Zähler einen Wert größer Null besitzt.

```
Algorithm SerVerfeinerung( G_0, ..., G_J )
{
    for ( k = J, ..., 0 ) do
    {
        BestimmeMarkierungen( G_k );                          (1)
        SchließeGitter( G_k );                                (2)
    }
    for ( k = 0, ..., J ) do if ( G_k ≠ ∅ ) then
    {
        if ( k > 0 ) then SchließeGitter( G_k );              (3)
        if ( k < J ) then VergröbereGitter( G_k );            (4)
        VerfeinereGitter( G_k );                              (5)
    }
    if ( G_J = ∅ ) then
        J := J − 1;                                           (6)
    else if ( G_{J+1} ≠ ∅ ) then
        J := J + 1;                                           (7)
}
```

Abbildung 3.11: Der serielle Verfeinerungsalgorithmus.

Der Aufbau des Gesamtalgorithmus gliedert sich nun in zwei Phasen (vgl. Abb. 3.11): In Phase I (1–2) werden die Gitter der einzelnen Stufen von oben nach unten (*top-down*), in Phase II (3–5) von unten nach oben (*bottom-up*) abgearbeitet. Dabei werden in Phase I nur die Markierungen der Elemente verändert; die Generierung neuer bzw. das Löschen nicht mehr gebrauchter Elemente findet erst in Phase II statt.

Phase I: In der Funktion *BestimmeMarkierungen* werden die Elemente der Stufe k je nach Verfeinerungszustand und Markierungen der Kinder auf Stufe $k+1$ gemäß den Punkten (a)–(c) markiert. Die Markierungen für den grünen Abschluss werden anschließend in *SchließeGitter* vorgenommen.

Phase II: Die Markierungen werden nun in die Tat umgesetzt: Nicht mehr benötigte Tetraeder, Seitenflächen, Kanten und Knoten auf Stufe $k+1$ werden für $k < J$ durch *VergröbereGitter* gelöscht. In *VerfeinereGitter* werden dann die neu hinzukommenden Tetraeder, Seitenflächen, Kanten und Knoten auf Stufe $k+1$ erzeugt und miteinander verknüpft. Da dabei neue reguläre Elemente auf Stufe $k+1$ entstehen, die evtl. im Zuge des grünen Abschlusses verfeinert werden müssen, wird wiederum *SchließeGitter* auf der nächsten Stufe aufgerufen, also im nächsten Schleifendurchlauf für $k > 0$. Für $k = 0$ wäre ein Aufruf im Übrigen wirkungslos, da bereits in Phase I unmittelbar vorher *SchließeGitter* aufgerufen wurde.

Anstatt die aufgerufenen Funktionen an dieser Stelle schon näher zu erläutern, wird vorher schon ein erster Schritt zur Parallelisierung des Verfeinerungsalgorithmus gewagt. Dieser bewirkt keine wesentlichen Veränderungen des Gesamtalgorithmus und der Unterfunktionen, so dass die detaillierteren Ausführungen direkt an dem ersten parallelen Entwurf des Verfeinerungsalgorithmus erfolgen sollen.

3.5 Erste Fassung des parallelen Verfeinerungsalgorithmus

Wird die dynamische Lastverteilung zunächst außer Acht gelassen, so kann man sich auf eine einfache parallele Speicherung der Geometriedaten beschränken. Alle Ausführungen dieses Kapitels basieren also auf der Annahme, dass die Anfangstriangulierung $\mathcal{T}_0 = \mathcal{G}_0$ zu Anfang gleichmäßig auf die Prozessoren verteilt sei und dass außer dieser anfänglichen Lastverteilung keine weitere Lastverteilung mehr erfolgt.

Die verteilte Speicherung der Anfangstriangulierung sei derart, dass jeder Tetraeder auf genau einem Prozessor gespeichert wird; die Tetraeder sind also *überlappungsfrei* gespeichert. Es ist klar, dass dann durch Verfeinerung auch keine solche Überlappung von Tetraedern mehr entstehen kann, da jeder Tetraeder einer Stufe $k > 0$ eindeutig seinem Elterntetraeder und damit einem bestimmten Prozessor zugeordnet ist.

Die Menge der auf Prozessor p gespeicherten Tetraeder auf Stufe k sei mit $G_k(p)$ bezeichnet und heißt *lokales Gitter* der Stufe k auf Prozessor p. Die Vereinigung dieser lokalen Gitter einer Stufe k ergibt gerade das entsprechende Gitter $\mathcal{G}_k$, welches aus dem seriellen Algorithmus bekannt ist:

$$\mathcal{G}_k = \bigcup_{p=1,\ldots,P} \mathcal{G}_k(p) \ .$$

$\mathcal{G}_k$ wird auch als *globales Gitter* der Stufe k bezeichnet, um im Kontext einer verteilten Speicherung des Multilevelgitters den Unterschied zu den lokalen Gittern deutlich zu machen. Entsprechend wird $\mathcal{M}(p) = (\mathcal{G}_0(p), \ldots, \mathcal{G}_J(p))$ ein *lokales Multilevelgitter* auf Prozessor p genannt.

Da im Laufe des Verfeinerungsalgorithmus von den Tetraedern aus auf die jeweiligen Subsimplices häufig zugegriffen wird, sollte dieser Zugriff lokal (d.h. ohne Kommunikation zwischen den Prozessoren) erfolgen. Um dies zu ermöglichen, müssen zu einem auf einem bestimmten Prozessor gespeicherten Tetraeder auch alle seine Subsimplices auf demselben Prozessor gespeichert werden. Dies impliziert eine überlappende Speicherung der Knoten, Kanten und Seitenflächen.

Algorithm *ParVerfeinerung*$_{\text{Version 1}}$ ($\mathcal{G}_0(p), \ldots, \mathcal{G}_J(p)$)
{
  for ($k = J, \ldots, 0$) **do**
  {
    BestimmeMarkierungen($\mathcal{G}_k(p)$); (1)
    `AkkumuliereKantenzähler`($\mathcal{G}_k(p)$); (2)
    SchließeGitter($\mathcal{G}_k(p)$); (3)
  }
  for ($k = 0, \ldots, J$) **do if** ($\mathcal{G}_k(p) \neq \emptyset$) **then**
  {
    if ($k > 0$) **then** *SchließeGitter*($\mathcal{G}_k(p)$); (4)
    if ($k < J$) **then** *VergröbereGitter*($\mathcal{G}_k(p)$); (5)
    `DDD_IdentifyBegin`(); (6)
      VerfeinereGitter($\mathcal{G}_k(p)$); (7)
    `DDD_IdentifyEnd`(); (8)
  }
  `BestimmeLetztesLevel`(); (9)
}

Abbildung 3.12: Die erste Version des parallelen Verfeinerungsalgorithmus.

Geht man von dieser einfachen parallelen Datenstruktur aus (bei Hinzunahme der dynamischen Lastverteilung wird im Hinblick auf die Tetraeder eine kompliziertere verteilte Speicherung nötig sein), so ist die Parallelisierung des Verfeinerungsalgorithmus verhältnismäßig einfach. Insbesondere müssen keine massiven Eingriffe in den Algorithmus vorgenommen werden, es kommen lediglich einige Kommunikationsaufrufe hinzu, so dass diese erste Version des parallelen Algorithmus inklusive der Unterroutinen sich fast wie dessen serielle Version liest.

Die verteilte Speicherung der Geometrie wird mit Hilfe von DDD verwaltet, ebenso wird die gesamte Prozessorkommunikation über DDD abgewickelt. Dazu müssen alle verteilt gespeicherten Objekte, in diesem Fall also die Simplices, bei DDD angemeldet werden (`DDD_HdrConstructor`) und bei Wegfall auch wieder abgemeldet werden (`DDD_HdrDestructor`). Dies geschieht mehr oder weniger automatisch, indem entsprechende Aufrufe in den Konstruktor bzw. Destruktor der entsprechenden Simplextypen aufgenommen werden.

Beim Erzeugen von Objekten auf Prozessorrändern (z.B. der Verfeinerung einer Kante, die auf mehreren Prozessoren gespeichert ist) muss dafür Sorge getragen werden, dass die auf den einzelnen Prozessoren erzeugten lokalen Objekte zu den entsprechenden verteilten Objekten zusammengefasst werden. Dies geschieht in DDD mittels des Identifikationsmoduls. Bei dem Beispiel der überlappend gespeicherten Kante sollen durch Verfeinerung ein Mittelpunktsknoten und zwei Kindkanten entstehen, die auf denselben Prozessoren überlappend gespeichert sind. Nach unabhängiger Erzeugung des Knotens und der zwei Kanten auf den beteiligten Prozessoren müssen diese Subsimplices untereinander identifiziert werden, wobei als *ein* Identifkationsmerkmal die gemeinsame Elternkante dient (vgl. auch Abbildung 2.8). Analog wird bei der Identifikation neu erzeugter Seitenflächen auf Prozessorrändern auf die Elternseitenfläche zurückgegriffen, aus welcher die entsprechenden Seitenflächen durch Verfeinerung entstanden sind. Genaueres dazu wird in der Beschreibung der Funktion *VerfeinereGitter* ausgeführt.

Der Wert der für den Algorithmus zentralen Kantenzähler, die das Verfeinerungsmuster der einzelnen Tetraeder festlegen, wird in verteilter Form gespeichert (vgl. Def. 2.7), und zwar aus folgendem Grund: Ein solcher Kantenzähler kann jeweils durch die verschiedenen einer Kante anliegenden Tetraeder modifiziert werden, die sich evtl. auf verschiedenen Prozessoren befinden können. Um eine Unabhängigkeit der Prozessoren bei der Modifikation der Kantenzähler zu erreichen, ist die verteilte Speicherung des Wertes also die adäquate Wahl. Zur Entscheidung, ob eine Kante verfeinert werden soll oder nicht, ist dann allerdings vorher die Akkumulation ihres Kantenzählers nötig.

Im Einzelnen treten beim Übergang vom seriellen Algorithmus zur ersten parallelen Version also folgende zusätzliche Aufrufe hinzu:

- Nach der Modifikation der Kantenzähler in *BestimmeMarkierungen* werden die Kantenzähler auf den Prozessorgrenzen akkumuliert (`AkkumuliereKantenzähler`). Dies wird durch Benutzung eines DDD-Interfaces für die Kanten realisiert.
- Bei Wegfall von (Sub-)Simplizes müssen diese bei DDD abgemeldet werden (*VergröbereGitter*).
- Neu hinzukommende (Sub-)Simplizes müssen bei DDD angemeldet und auf Prozessorrändern zusätzlich identifiziert werden (*VerfeinereGitter*).
- Nach Ablauf von Phase II muss unter den Prozessoren ermittelt werden, wie die Nummer J' des letzten Levels lautet (`BestimmeLetztesLevel`).

Die entsprechenden Kommunikationsaufrufe sind im Algorithmus typographisch durch Verwendung der folgenden `Schriftart` kenntlich gemacht. Streicht man jene Zeilen, die Kommunikationsaufrufe enthalten, erhält man die serielle Version des Algorithmus zurück.

Die Arbeitsweise des parallelen Algorithmus soll im Folgenden anhand eines 2D-Beispiels erläutert werden, und zwar für $P = 4$ Prozessoren. Die Ausgangstriangulierungshierarchie $\mathcal{H} = (\mathcal{T}_0, \mathcal{T}_1)$ umfasst zwei Triangulierungen $\mathcal{T}_0$ und $\mathcal{T}_1$;

entsprechend besteht das zugehörige Multilevelgitter $\mathcal{M} = \mathcal{M}(\mathcal{H})$ aus zwei Gittern $\mathcal{G}_0$ und $\mathcal{G}_1$. Die Triangulierung $\mathcal{T}_0$ bestehe aus vier Dreiecken, von denen je eines mitsamt seinen Kindern auf einem der vier Prozessoren gespeichert sei. Zwei der regulären Kinder auf Prozessor P_3 sind zur Verfeinerung markiert (vgl. Abb. 3.13).

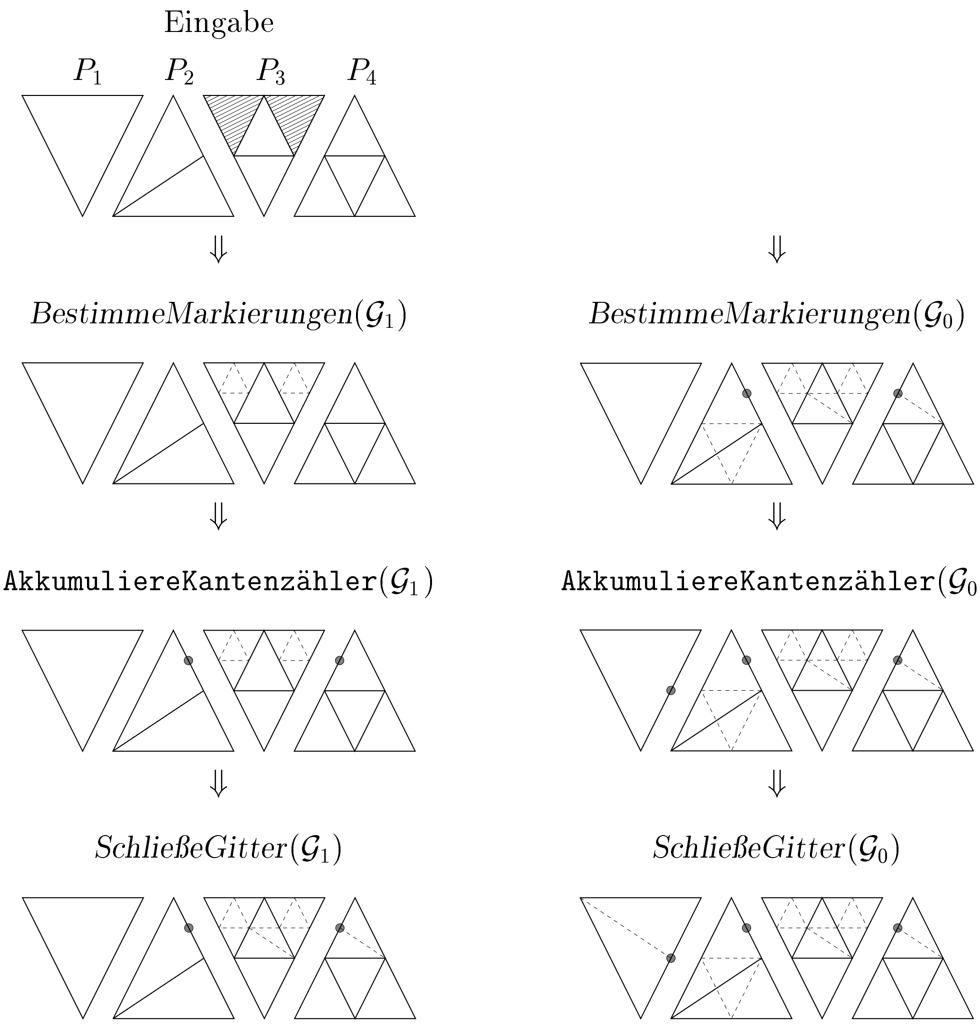


Abbildung 3.13: Die Arbeitsweise des parallelen Verfeinerungsalgorithmus in Phase I.

Phase I startet mit *BestimmeMarkierungen*$(\mathcal{G}_1)$, wo die zur Verfeinerung markierten Dreiecke zu regulärer Verfeinerung markiert werden. Die Verfeinerungsmarkierungen sind jeweils durch gestrichelte Linien kenntlich gemacht. Gleichzeitig werden die Zähler der angrenzenden Kanten erhöht, diese Kanten sind dadurch zur Verfeinerung markiert.

Da dies nur auf den lokal gespeicherten Kopien der Kanten geschieht, werden als Nächstes die Kantenverfeinerungsinformationen durch `AkkumuliereKantenzäh-`

`ler`$(\mathcal{G}_1)$ auch auf die Objektkopien der übrigen Prozessoren übertragen. Eine solche Markierung zur Kantenverfeinerung, die von einer Objektkopie eines anderen Prozessors herrührt, ist durch einen Kreis auf der jeweiligen Kante dargestellt.

Da nun die nötigen Kantenverfeinerungsinformationen auf allen Prozessoren bekannt sind, kann für das Gitter $\mathcal{G}_1$ der grüne Abschluss berechnet werden. Dies geschieht in *SchließeGitter*$(\mathcal{G}_1)$. Beachte, dass das *irreguläre* Kinddreieck auf Prozessor P_2 keine Verfeinerungsmarkierung erhält, da grüne Kinder nicht verfeinert werden dürfen.

Derselbe Zyklus wird nun auf Gitter $\mathcal{G}_0$ wiederholt: In *BestimmeMarkierungen*$(\mathcal{G}_0)$ wird das grün verfeinerte Dreieck auf Prozessor P_2 zur regulären Verfeinerung markiert, da eine der Kindkanten zur Verfeinerung vorgesehen ist. Die dadurch ausgelöste Erhöhung der angrenzenden Kantenzähler wird durch `AkkumuliereKantenzähler`$(\mathcal{G}_0)$ auf die Objektkopien der anderen Prozessoren übertragen. Anschließend wird in *SchließeGitter*$(\mathcal{G}_0)$ der grüne Abschluss auf $\mathcal{G}_0$ bestimmt.

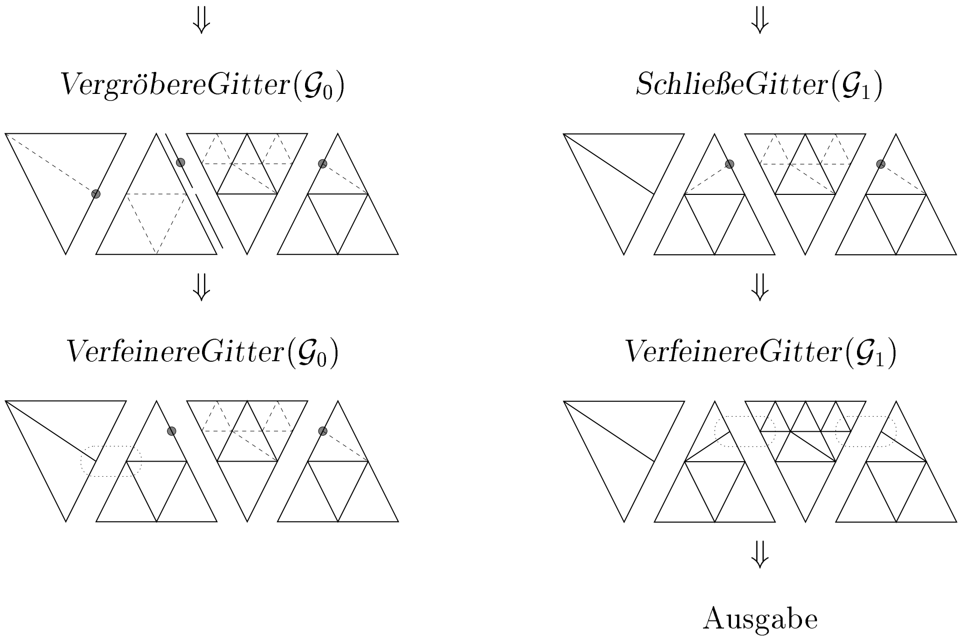


Abbildung 3.14: Die Arbeitsweise des parallelen Verfeinerungsalgorithmus in Phase II.

Nun tritt der Algorithmus in *Phase II* (vgl. Abb. 3.14). In *VergröbereGitter*$(\mathcal{G}_0)$ wird zunächst die grüne Verfeinerung des Dreiecks auf Prozessor P_2 entfernt. Die zwei Kindkanten der vormals unterteilten Kanten sowie deren Mittelpunktsknoten können für die neu vorgesehene reguläre Verfeinerung wiederverwendet werden und bleiben deswegen erhalten. Beachte, dass dies auch unbedingt notwendig ist, da sonst die Kantenverfeinerungsinformation der einen Kindkante verlorenginge.

VerfeinereGitter($\mathcal{G}_0$) setzt die Verfeinerungsmarkierungen der Dreiecke in $\mathcal{G}_0$ in reale Verfeinerung um. Dabei werden die wiederverwendbaren Subsimplices in die neue Verfeinerung integriert. Neu entstandene Subsimplices (Knoten, Kanten) auf Prozessorrändern werden miteinander identifizert; dies wird in der Abbildung durch eine punktierte Umkreisung dargestellt.

Auf Prozessor P_2 befindet sich nun ein neues, reguläres Kinddreieck, welches eine zur Verfeinerung vorgesehene Kante besitzt. Dieses neu entstandene Dreieck muss zur Konsistenzerhaltung des Multilevelgitters entsprechend irregulär verfeinert werden. Die Entscheidung über den passenden grünen Abschluss wird in *SchließeGitter*($\mathcal{G}_1$) getroffen.

Die Verfeinerungsmarkierungen der Dreiecke des Gitters $\mathcal{G}_1$ werden schließlich durch *VerfeinereGitter*($\mathcal{G}_1$) umgesetzt und die entsprechenden Kinddreiecke erzeugt. Auf Prozessorrändern neu entstandene Subsimplices werden wiederum geeignet miteinander identifiziert.

Nach Abschluss der Phase II ist ein neues konsistentes Multilevelgitter mit drei Leveln entstanden, entsprechend trägt das letzte Level die Nummer $J' = 2$. Beachte, dass auf Prozessor P_1 keine Dreiecke aus dem letzten Gitterlevel $\mathcal{G}_2$ gespeichert sind. Damit aber alle Prozessoren die gleiche Levelzahl besitzen, muss J' in `BestimmeLetztesLevel`() durch globale Maximumsbildung unter den Prozessoren berechnet werden.

3.5.1 Beschreibung der aufgerufenen Funktionen

Nach diesem verdeutlichenden Beispiel für die Arbeitsweise des Algorithmus *ParVerfeinerung*$_{\text{Version 1}}$ sollen nun nacheinander die einzelnen Funktionen aufgeführt und näher erläutert werden, die im Laufe des Algorithmus aufgerufen werden.

BestimmeMarkierungen

Für alle Elemente der Stufe k werden die Markierungen des Fehlerschätzers folgendermaßen umgesetzt:

- Zur regulären Verfeinerung (RegRef) werden markiert
  - reguläre Blätter, die für Verfeinerung (Ref) markiert sind (1),
  - irregulär verfeinerte Elemente, die mindestens ein Kind besitzen, welches verfeinert werden soll (7):
    * entweder wegen einer entsprechenden Markierung (Ref)
    * oder wegen einer zur Verfeinerung vorgesehenen Kante.

```
Function BestimmeMarkierungen( $\mathcal{G}_k(p)$ )
{
    for ( $T \in \mathcal{G}_k(p)$ ) do
    {
        if ( T ist unverfeinert ) then
        {
            if ( T ist regulär ) and ( mark(T) = Ref ) then
            {
                mark(T) := RegRef;                                  (1)
                Erhöhe Kantenzähler;                                (2)
            }
            if ( $k = 0$ ) and ( mark(T) = Del ) then
                mark(T) := NoRef;                                   (3)
        }
        else if ( T ist regulär verfeinert ) then
        {
            if ( $\forall T' \in \mathcal{K}(T)$ : mark(T') = Del ) then
            {
                mark(T) := NoRef;                                   (4)
                Erniedrige Kantenzähler;                            (5)
            }
            for ( $T' \in \mathcal{K}(T)$ ) do
                if ( mark(T') = Del ) then
                    mark(T') := NoRef;                              (6)
        }
        else // T ist irregulär verfeinert
        {
            if  ( $\exists T' \in \mathcal{K}(T)$ : mark(T') = Ref )
            or  ( mind. eine Kindkante, die nicht Kante von T ist,
                ist zur Verfeinerung vorgesehen ) then
            {
                mark(T) := RegRef;                                  (7)
                Erhöhe Kantenzähler;                                (8)
            }
            else
                mark(T) := NoRef;                                   (9)
            for ( $T' \in \mathcal{K}(T)$ ) do
                mark(T') := NoRef;                                  (10)
        }
    }
}
```

Wird ein Tetraeder zur regulären Verfeinerung markiert, so werden außerdem seine sechs Kantenzähler erhöht (2, 8).

- Zur Nichtverfeinerung (NoRef) werden markiert
  - alle regulär verfeinerten Elemente, deren Kinder alle für Entfernung (Del) markiert sind (4). Weil die reguläre Verfeinerung wegfällt, müssen die jeweiligen Kantenzähler erniedrigt werden (5),
  - alle irregulär verfeinerten Elemente, deren Kinder nicht zur weiteren Verfeinerung vorgesehen sind (9). Die bestehende grüne Verfeinerung muss erst durch den grünen Abschluss in *SchließeGitter* entweder erneut bestätigt oder durch eine andere ersetzt werden.

  Außerdem werden alle Markierungen zur Entfernung (Del) der Kinder auf Stufe $k+1$ durch solche zur Nichtverfeinerung (NoRef) ersetzt (6, 10). Gleiches gilt für Elemente auf Stufe 0, die nicht entfernt werden dürfen (3). Da in *SchließeGitter* keine Markierungen für Entfernung gesetzt werden, gibt es also nach Abschluss der Phase I keine Elemente T mit $\mathsf{mark}(T) =$ Del mehr.

`AkkumuliereKantenzähler`

Diese Funktion besteht aus einem einzigen Aufruf, der die DDD-Interface-Kommunikation auf den überlappend gespeicherten Kanten eines bestimmten Levels auslöst (`DDD_IFAExchange`). Der aufgerufene Gather-Handler sammelt die Werte der verteilt gespeicherten Kantenzähler ein, der Scatter-Handler addiert diese unter den Prozessoren auf, so dass zum Schluss jeweils der akkumulierte Wert in der Kante vorliegt. Die Akkumulation erfolgt dabei nicht im selben Kantenzähler, sondern in einer weiteren Zählervariablen auf der Kante, damit der verteilt gespeicherte Kantenzähler nicht verändert wird. Ansonsten würden nachfolgende Akkumulationen falsche Ergebnisse liefern, und aufgrund dessen käme es evtl. zur Auswahl falscher Verfeinerungsregeln.

SchließeGitter

In dieser Funktion wird der grüne Abschluss bestimmt. Deswegen werden nur solche Elemente der Stufe k betrachtet, welche

1. regulär sind (irreguläre Elemente, also grüne Kinder, dürfen nicht verfeinert werden), und
2. nicht bereits für reguläre Verfeinerung (RegRef) markiert sind.

Function *SchließeGitter(* $\mathcal{G}_k(p)$ *)*
{
  for *(* $T \in \mathcal{G}_k(p)$ *)* **do**
    if *(T ist regulär)* **and** *(* $\mathsf{mark}(T) \neq$ *RegRef)* **then**
    {
      Bestimme anhand der Kantenzähler
      die passende Verfeinerungsregel $0 \leq R \leq 63$ *;* *(1)*
      if *(* $R =$ *NoRef)* **and** *(* $\mathsf{mark}(T) =$ *Del)* **then**
        mache nichts;
        // NoRef-Markierung wird in
          BestimmeMarkierungen($\mathcal{G}_{k-1}(p)$ *) gesetzt!*
      else
        $\mathsf{mark}(T) := R$; *(2)*
    }
}

Aufgrund des vollen Satzes an Verfeinerungsregeln ist die Bestimmung der richtigen grünen Verfeinerungsregel einfach: Sie ist durch das Kantenverfeinerungsmuster eindeutig festgelegt (vgl. Abschnitt 3.2.2). In DROPS werden die Markierungen als Integer repräsentiert, das Bitmuster der adäquaten Regel stimmt mit dem entsprechenden Kantenverfeinerungsmuster überein. Insbesondere gilt NoRef $= 000000_2 = 0$ (Nichtverfeinerung) und RegRef $= 111111_2 = 63$ (reguläre Verfeinerung).

Nur im Falle, dass keine Kante zur Verfeinerung vorgesehen ist, die entsprechende Regel also „NoRef" (Nichtverfeinerung) lautet, muss beachtet werden, dass eine Markierung „Del" nicht überschrieben werden darf, weil sie in Zeile (6) von *BestimmeMarkierungen* auf der nächsttieferen Stufe noch von Bedeutung ist: Dort wird der Elterntetraeder für Entfeinerung markiert, falls alle seine Kinder diese Del-Markierung tragen; zusätzlich werden dort alle Del-Markierungen der Kinder in NoRef-Markierungen umgewandelt, eine in *SchließeGitter* bestimmte Regel „NoRef" wird hier also schließlich auch als Markierung umgesetzt.

Bisher wurden nur die Markierungen der Tetraeder modifiziert. Die beiden nachfolgenden Funktionen *VergröbereGitter* bzw. *VerfeinereGitter* sind für das Löschen bzw. das Erzeugen der entsprechenden Objekte zuständig.

VergröbereGitter

Diese Funktion entfernt bei Aufruf auf Stufe $k < J$ die nicht mehr benötigten Knoten, Kanten, Seitenflächen und Tetraeder der Stufe $k + 1$.

Da die Knoten, Kanten und Seitenflächen von mehreren Tetraedern geteilt werden, dürfen diese Objekte nur dann wegfallen, wenn keine der Verfeinerungsmar-

Function *VergröbereGitter*($\mathcal{G}_k(p)$)
{
    Markiere alle Knoten, Kanten und Seitenflächen
    auf Level $k+1$ *für Entfernung;* (1)
    for ($T \in \mathcal{G}_k(p)$) **do if** (T *ist verfeinert*) **then**
    {
        if (T *ist entsprechend seiner Verfeinerung markiert*) **then**
            Lösche die Del-Markierungen aller
            Knoten, Kanten, Seitenflächen und Kinder von T; (2)
        else
        {
            for ($T' \in \mathcal{K}(T)$) **do**
                $\mathsf{mark}(T') :=$ *Del;* (3)
            if ($\mathsf{mark}(T) \neq$ *NoRef*) **then**
                Behalte alle Knoten, Kanten, Seitenflächen und
                Kinder von T*, welche für die neue Verfeinerung*
                gemäß $\mathsf{mark}(T)$ *wiederbenutzt werden können;* (4)
        }
    }
    `DDD_XferBegin`(); (5)
        Melde alle für Entfernung markierten Knoten, Kanten,
        Seitenflächen und Tetraeder auf Level $k+1$ *bei DDD*
        mittels `DDD_XferDeleteObj` *ab;* (6)
    `DDD_XferEnd`(); (7)
    Lösche alle für Entfernung markierten Knoten,
    Kanten, Seitenflächen und Tetraeder auf Level $k+1$; (8)
}

kierungen auf den Tetraedern der Stufe k diese mehr benötigen. Anders gesagt: Gibt es nur einen Tetraeder, dessen Verfeinerungsmarkierung einen bestehenden Subsimplex der Stufe $k+1$ weiterhin braucht, so muss dieser behalten werden. Deswegen wird folgende Strategie angewendet: Zunächst werden alle Subsimplices der Stufe $k{+}1$ für Entfernung markiert (1). Anschließend werden diese Markierungen teilweise zurückgenommen, wenn der entsprechende Elterntetraeder entweder gemäß seiner Verfeinerung markiert ist (2) oder die neue Verfeinerungsmarkierung manche der bestehenden Kindsubsimplices wiederverwenden kann (4).

Die Bestimmung der zu löschenden Tetraeder ist einfacher, da ein Kindtetraeder auf Stufe $k+1$ zu genau einem Elterntetraeder auf Stufe k (und nicht zu mehreren) gehört. Deswegen kann von diesem Elterntetraeder aus entschieden werden, welches der Kinder wegfallen bzw. wiederverwendet werden kann (3,4). Beachte, dass vor Aufruf von *VergröbereGitter* auf Stufe k keine Del-Markierungen auf Tetraedern der Stufe $k+1$ existieren, da diese sämtlich während Phase I entfernt wurden.

Alle anschließend noch für Entfernung markierten Tetraeder und Subsimplices der Stufe $k+1$ werden tatsächlich nicht mehr gebraucht und können gelöscht werden (8), nachdem sie bei DDD abgemeldet wurden (5–7).

VerfeinereGitter

Diese Funktion erzeugt die neuen Objekte auf Stufe $k+1$. Ist speziell $k = J$, so wird ein neues Gitter $\mathcal{G}_{J+1}(p)$ angelegt (1). Anschließend wird für jedes Element der Stufe k, welches nicht entsprechend seiner Verfeinerung markiert ist, seine neue Verfeinerung erzeugt. Dazu werden in (2, 4) alle dazu benötigten Kindtetraeder, -seitenflächen und -kanten zunächst auf Stufe $k+1$ gesucht (alte Objekte werden wiederverwendet); wenn diese nicht existieren, werden sie entsprechend erzeugt (3, 5). Anschließend müssen die Objekte untereinander durch Zeiger verknüpft werden. Nach Beendigung von *VerfeinereGitter* sind alle Tetraeder der Stufe k entsprechend ihrer Verfeinerung markiert.

Bemerkung 3.13 (Implementationshinweis) Um das Suchen wiederverwendbarer Objekte auf Stufe $k+1$ effizient durchführen zu können, wird wiederum das Konzept der konsistenten Nummerierung ausgenutzt: In *VergröbereGitter* wird für jedes wiederverwendbare Objekt (Kindkante, -seitenfläche oder -tetraeder) in dessen ersten Knoten temporär ein Verweis darauf gespeichert. In *VerfeinereGitter* muss die Suche nach dem entsprechenden Objekt nun nicht unter allen Objekten der Stufe $k+1$, sondern nur unter den Verweisen in dessen ersten Knoten durchgeführt werden. Dieser Knoten, in dem gesucht werden muss, ist dank der konsistenten Nummerierung eindeutig. Nach Beendigung von Phase II werden schließlich alle temporären Verweise wieder gelöscht.

Bei der Erzeugung von Knoten, Kanten oder Seitenflächen auf Prozessorrändern muss darauf geachtet werden, dass die lokal auf den einzelnen Prozessoren erzeugten Subsimplices zu jeweils einem verteilten Objekt zusammengefasst werden. Um den Begriff des Prozessorrandes genauer zu fassen, ist er Gegenstand der folgenden Definition. (Diese Definition wird in Kapitel 4 erweitert werden, um einer komplexeren Datenstruktur gerecht zu werden.)

Definition 3.14 (Prozessorrand) Zu einem Subsimplex S (das kann ein Knoten, eine Kante oder eine Seitenfläche sein) gebe die Menge

$$\mathsf{procs}(S) := \{p : 1 \le p \le P, \quad S \text{ ist auf Prozessor } p \text{ gespeichert}\}$$

Function *VerfeinereGitter(* $\mathcal{G}_k(p)$ *)*
{
  if *(* $k = J$ *)* **then** $\mathcal{G}_{k+1}(p) := \emptyset$; *(1)*
  for *(* $T \in \mathcal{G}_k(p)$ *)* **do**
    if *(T ist nicht entsprechend seiner Verfeinerung markiert)* **then**
    {
      Verfeinere T entsprechend $\mathsf{mark}(T)$*, d.h.*
        Finde alle existierenden Kinder von T; *(2)*
        Erzeuge alle fehlenden Kinder von T; *(3)*
      for *(* $T' \in \mathcal{K}(T)$ *)* **do**
      {
        Finde alle existierenden Knoten,
        Kanten und Seitenflächen von T'; *(4)*
        Erzeuge alle fehlenden Knoten,
        Kanten und Seitenflächen von T'; *(5)*
        Identifiziere neu erzeugte Simplices
        auf Prozessorrändern durch Aufrufe
        von `DDD_IdentifyObject` *miteinander;* *(6)*
      }
    }
}

gerade die an der verteilten Speicherung von S beteiligten Prozessoren an. Man sagt, S liege auf einem *Prozessorrand*, wenn S auf mehreren Prozessoren gespeichert ist, also

$$|\mathsf{procs}(S)| \geq 2$$

gilt. In diesem Fall spricht man auch von einer *verteilten Speicherung* bzw. von *Überlappung*.

Um DDD mitzuteilen, dass es sich bei den auf den verschiedenen Prozessoren gespeicherten neu erzeugten Einzelobjekten um Objektkopien ein und desselben (verteilten) Objektes handelt, werden die entsprechenden Objekte durch den Identifikationsmechanismus von DDD einander zugeordnet (6). Miteinander identifizierte Objekte erhalten die gleiche globale Nummer und werden innerhalb der DDD-Datenstrukturen in die Kopplungslisten eingetragen.

Zur Identifikation dienen sogenannte Identifikationstupel, deren Einträge Zahlen, Zeichenketten oder andere DDD-Objekte sein können. Objekte auf verschiedenen Prozessoren, die dieselben Identifikationstupel tragen, werden miteinander identifiziert. Neben der Frage, *welche Objekte* auf *welchen Prozessoren* miteinander identifiziert werden müssen, stellt sich also auch die Frage nach dazu *geeigneten Identifikationstupeln*, welche eine eindeutige Zuordnung ermöglichen.

Da durch Verfeinerung keine Überlappung von Tetraedern entstehen kann, ist die Identifikation nur für Subsimplices von Belang. Wie entstehen aber neue Subsimplices durch Verfeinerung?

Ein neuer *Knoten* V kann nur bei der Verfeinerung einer Kante E entstehen, und zwar gerade als deren Mittelpunktsknoten. Insofern ist im Falle von Knoten die Identifikation einfach: Der neue Knoten V muss genau dann identifiziert werden, wenn E auf einem Prozessorrand liegt. Die Identifikation muss mit allen Prozessoren $\mathsf{procs}(E)$ stattfinden, auf denen die Kante E gespeichert ist, denn auch dort entstehen im Zuge ihrer Verfeinerung neue Mittelpunktsknoten. Als eindeutiges Identifizierungsmerkmal kann E selber benutzt werden, weil eine Kante und ihr Mittelpunktsknoten einander eindeutig zugeordnet sind.

Anders verhält es sich im Falle von *Kanten*, diese können auf verschiedene Arten bei der Verfeinerung eines Tetraeders T neu entstehen:

- entweder als innere Diagonale von T: Diese liegt nicht auf einem Prozessorrand, da sie im Inneren des Tetraeders T liegt, welcher eindeutig einem Prozessor zugeordnet ist. Es ist also keine Identifikation erforderlich.

- als Teilkante E' einer Kante E von T: Liegt E auf einem Prozessorrand, so findet Identifikation mit allen Prozessoren $\mathsf{procs}(E)$ statt, auf denen E gespeichert ist. Das Identifikationstupel setzt sich aus zwei Merkmalen zusammen: dem DDD-Objekt E sowie einer Zahl $i \in \{0, 1\}$, die die lokale Ordnung der Teilkante bestimmt. Dabei ist die erste Teilkante dadurch ausgezeichnet, dass sie am ersten Endknoten von E, die zweite Teilkante dadurch, dass sie am zweiten Endknoten von E anliegt. Aufgrund der Eigenschaft der konsistenten Knotennummerierung ist diese lokale Ordnung der Teilkanten auf allen Prozessoren eindeutig, somit ist sichergestellt, dass tatsächlich die richtigen Teilkanten miteinander identifiziert werden.

- als Kante E' auf einer Seitenfläche F von T: Falls F auf einem Prozessorrand liegt, so findet Identifikation mit allen Prozessoren $\mathsf{procs}(F)$ statt, auf denen F gespeichert ist. Als Identifikationstupel dienen zwei DDD-Objekte, nämlich die beiden Endknoten der Kante. Dass mindestens einer dieser Endknoten ein Mittelpunktsknoten einer Kante von T ist und daher evtl. selber noch in der Identifikationsphase steht, ist dabei kein Hindernis. DDD erlaubt ausdrücklich eine Identifikation durch DDD-Objekte, die in derselben Identifikationsphase identifiziert werden, solange dadurch keine zyklischen Abhängigkeiten entstehen.

  Dies ist hierbei aber nicht der Fall, da während der Identifikationsphase in *VerfeinereGitter*($\mathcal{G}_k(p)$) immer nur Objekte eines Levels (nämlich $k+1$) identifiziert werden; ein zur Identifikation von E' genutzter Knoten aus Level $k+1$ wird aber schließlich durch eine Kante identifiziert, die auf Level k liegt und damit bereits auf jeden Fall identifiziert ist.

```
Function BestimmeLetztesLevel()
{
    if ( 𝒢_J(p) = ∅ ) then
        J_p := J − 1;                                      (1)
    else if ( 𝒢_{J+1}(p) ≠ ∅ ) then
        J_p := J + 1;                                      (2)
    else
        J_p := J;                                          (3)
    Berechne J' := max_{1≤p≤P} J_p
    mittels eines MPI_Allreduce-Aufrufes;                  (4)
    if ( J_p < J' )
        for ( k = J_p + 1, ..., J' )
            𝒢_k(p) := ∅;                                   (5)
}
```

Eine neue *Seitenfläche* F' kann durch Verfeinerung von T nur auf zwei Arten entstehen: entweder im Inneren von T ($F' \subset \mathring{T}$) oder andernfalls ($F' \subset \partial T$) auf einer Seitenfläche F von T. Nur im letzten Fall ist Identifikation nötig, die dann mit allen Prozessoren procs(F) stattfindet, auf denen F gespeichert ist. Als Identifikationstupel dient wiederum ein DDD-Objekt, nämlich F, und eine ganze Zahl j, die die Nummer von F' in F angibt:

Durch Verfeinerung wird eine Elternseitenfläche je nach Kantenverfeinerungsmuster der anliegenden Kanten auf unterschiedliche Weise in Kinderseitenflächen (sogenannte *Subfaces*) zerlegt. Alle dabei auftauchenden Subfaces werden mit einer eindeutigen Nummer versehen, die das jeweilige Subface repräsentiert. Diese Nummer ist zum einen bei der Konstruktion neu entstehender Subfaces hilfreich, zum anderen kann damit eindeutig F' durch F identifiziert werden: Die wegen der konsistenten Knotennummerierung auf jedem Prozessor eindeutige lokale Knotenordnung in F gewährleistet, dass die Nummer von F' auf jedem Prozessor auch dieselbe ist.

`BestimmeLetztesLevel`

In dieser Funktion wird dafür gesorgt, dass nach Abschluss des Verfeinerungsalgorithmus alle Prozessoren dieselbe Anzahl von Leveln besitzen. Dazu bestimmt jeder Prozessor p die Nummer J_p seines letzten nichtleeren Gitterlevels (1–3). Anschließend wird das Maximum $J' = \max_p J_p$ dieser Nummern gebildet (`MPI_Allreduce`) (4). J' gibt gerade die Nummer des letzten Levels des zugehörigen globalen Multilevelgitters an. Falls $J_p \neq J'$ gilt, werden entsprechend viele leere Gitterlevel hinzugefügt (5).

Basierend auf der Korrektheit des seriellen Algortithmus, welche in [2] gezeigt wird, lässt sich zeigen:

Satz 3.15 *Der oben vorgestellte parallele Verfeinerungsalgorithmus liefert, auf ein konsistentes Multilevelgitter angewendet, wieder ein konsistentes Multilevelgitter als Ausgabe.*

Beweis: Für den seriellen Algorithmus wurde diese Aussage von Bey bereits bewiesen. Da jeder Prozessor p diesen Algorithmus auf sein lokales Multilevelgitter $\mathcal{M}(p)$ anwendet und dieses einer Triangulierunghierarchie zur Anfangstriangulierung $\mathcal{T}_0(p) = \mathcal{G}_0(p)$ entspricht, ist klar, dass das entstehende lokale Multilevelgitter $\mathcal{M}'(p)$ jeweils für sich genommen konsistent ist.

Für die Konsistenz des *verteilten* Multilevelgitters $\tilde{\mathcal{M}}' = \big(\mathcal{M}'(1), \dots \mathcal{M}'(P)\big)$ muss darüber hinaus gezeigt werden, dass für jedes Level $k = 0, \dots, J'$ die lokalen Gitter $\mathcal{G}_k(p)$, $p = 1, \dots, P$ auch „aneinanderpassen". Dies ergibt sich aus der Akkumulation der Kantenzähler sowie der konsistenten Nummerierung der Knoten:

Wegen `AkkumuliereKantenzähler` sind die Verfeinerungsmarkierungen auf den Kanten von $\tilde{\mathcal{M}}$ konsistent unter den Prozessoren bekannt. Dies bedeutet, dass alle darauf basierenden Entscheidungen bezüglich der Verfeinerungsregeln genau so getroffen werden, wie der serielle Algorithmus sie treffen würde, wenn er auf das *globale* Multilevelgitter

$$\mathcal{M} = (\mathcal{G}_0 \dots, \mathcal{G}_j)$$

angewendet würde, welches aus den globalen Gittern gebildet wird.

Bei der Umsetzung der Verfeinerung werden auch die auf Prozessorrändern liegenden Seitenflächen von den jeweiligen Tetraedern aus konsistent verfeinert: Da die konsistente Knotennummerierung über Prozessorgrenzen hinweg gültig ist (das liegt daran, dass jeder Knoten eine global gültige Nummer besitzt), werden die Seitenverfeinerungsmuster einer überlappt gespeicherten Seitenfläche von jedem der beiden verfeinernden Tetraeder aus zueinander passend gewählt, und das unabhängig voneinander: Die beiden beteiligten Tetraeder müssen dazu, obwohl sie auf verschiedenen Prozessoren gespeichert sind, keine weiteren Informationen austauschen. □

3.6 Fehlerschätzung

Als weitere Komponente eines adaptiven Verfahren spielt die Fehlerschätzung bei der Steuerung der adaptiven Gitteranpassung eine Rolle: Diese setzt sich aus einem lokalen Fehlerindikator η, welcher auf jedem Tetraeder lokal die Größe des

Fehlers abschätzt, sowie aus einer Markierungsstrategie zusammen, welche die lokalen Fehlerschätzungen in Verfeinerungs-/Entfeinerungsmarkierungen umsetzt. Aufgrund dieser Markierungen kann dann mit Hilfe des adaptiven Verfeinerungsalgorithmus ein verbessertes Gitter erzeugt werden.

Da der Fehlerschätzung in dieser Arbeit eher eine Nebenrolle zukommt und das Gewicht mehr auf dem adaptiven Verfeinerungsalgorithmus liegt, wurde eine leicht zu parallelisierende Fehlerschätzungstechnik implementiert.

Zu einer Triangulierung $\mathcal{T}$ von Ω sei $u_h = J_h \underline{u}$ die mit der Finite-Elemente-Methode gewonnene Lösungsfunktion. Als Fehlerindikator $\eta_G : \mathcal{T} \to \mathbb{R}$ kommt folgender simple Fehlerschätzer zum Einsatz:

$$\eta_G(T) := \max_{x,y \in \overline{T}} |u_h(x) - u_h(y)| \,.$$

Beachte, dass es aufgrund der gewählten Diskretisierung (lineare finite Elemente) zur Berechnung des Maximums ausreicht, für $x, y \in \overline{T}$ die Eckpunkte von T einzusetzen. Dieser Fehlerschätzer sorgt dafür, dass Bereiche, in denen der Gradient von u_h groß ist, stärker verfeinert werden, und zählt deswegen zu den Gradientenfehlerschätzern.

Die Berechnung von η_G ist trivial parallelisierbar, weil keine Informationen von benachbarten Tetraedern gebraucht werden; alle benötigten Daten sind lokal auf dem jeweiligen Tetraeder verfügbar.

Als Markierungsstrategie wird ein Algorithmus aus [1] gewählt: Bei Vorgabe einer Toleranz ε und eines geeigneten Fehlerschätzers $\eta : \mathcal{T} \to \mathbb{R}$ werden die Tetraeder aus $\mathcal{T}$ folgendermaßen markiert:

Algorithm *Markiere*($\mathcal{T}, \eta, \varepsilon$)
{
  $m := \max_{T \in \mathcal{T}} \eta(T)$; *(1)*
  if ($m \le \varepsilon$) **then return**; *(2)*
  $\delta := \min\{\varepsilon, m/C\}$; *(3)*
  for ($T \in \mathcal{T}$) **do**
    if ($\eta(T) \ge \delta$) **then**
      Markiere T zur Verfeinerung; *(4)*
}

Der Parameter $C > 0$ kann frei gewählt werden. In dem adaptiven Experiment in Abschnitt 6.4 wurde z.B. $C = 4$ gewählt. Die zur parallelen Markierung einzig nötige Kommunikation ist die Bildung des globalen Maximums (`MPI_Allreduce`) zu Anfang in (1). Alle folgenden Anweisungen können dann prozessorweise unabhängig voneinander ausgeführt werden.

Bemerkung 3.16 In der seriellen Version von DROPS wird für die Poissongleichung eine Fehlerschätzungstechnik benutzt, die aus einem Residuenfehlerschätzer nach VERFÜRTH und einer Markierungsstrategie nach DÖRFLER besteht. Ob und wie diese geeignet parallelisiert werden könnten, soll im Folgenden kurz diskutiert werden.

Der Residuenfehlerschätzer $\eta_R : \mathcal{T} \to \mathbb{R}$ wertet auf einem Tetraeder T neben der $L_2(T)$-Norm der rechten Seite f den Sprung des Gradienten ∇u_h über die Seitenflächen aus. Bei Seitenflächen, die auf Prozessorrändern liegen, ist dazu Kommunikation nötig. Die nötigen Daten können über ein DDD-Interface für die Seitenflächen ausgetauscht werden.

Die DÖRFLER-Markierungsstrategie beruht darauf, diejenigen Tetraeder T mit den größten lokalen Schätzungen $\eta_R(T)$ zu markieren, die zusammen einen bestimmten Anteil des Gesamtfehlers ausmachen. Dazu werden im seriellen Algorithmus zuerst die Tetraeder absteigend nach der lokalen Fehlerschätzung $\eta_R(T)$ sortiert: $\mathcal{T} = \{T_1, \dots, T_m\}$ mit

$$\eta_R(T_1) \geq \eta_R(T_2) \geq \cdots \geq \eta_R(T_m)$$

und gleichzeitig der Gesamtfehler $E := \sum_{T \in \mathcal{T}} \eta_R(T)$ akkumuliert. Dann werden diejenigen Tetraeder $T_1, \dots, T_k$ zur Verfeinerung markiert, welche für einen Anteil σ_{rel} des Gesamtfehlers verantwortlich sind:

$$\sum_{j=1}^{k} \eta_R(T_j) \approx \sigma_{\text{rel}}\, E \ .$$

Der Parameter $\sigma_{\text{rel}} \in [0, 1]$ kann vom Benutzer frei gewählt werden.

Die Parallelisierung dieser Markierungsstrategie fällt vermutlich schwieriger aus.

Kapitel 4

Adaptive Verfeinerung und Lastverteilung

4.1 Warum Tetraederüberlappung notwendig ist

Bei der verteilten Speicherung des Multilevelgitters $\mathcal{M}$ auf P Prozessoren seien wie bereits in Abschnitt 3.5 die auf Prozessor p gespeicherten Elemente aus Gitterlevel k mit

$$\mathcal{G}_k(p) := \{T \in \mathcal{G}_k : T \text{ wird auf Prozessor } p \text{ gespeichert}\}$$

bezeichnet. $\mathcal{G}_k(p)$ heißt *lokales Gitter* der Stufe k auf Prozessor p. Das globale Gitter $\mathcal{G}_k$ erhält man wiederum durch Vereinigung der lokalen Gitter $\mathcal{G}_k(p)$ aller Prozessoren $p = 1, \ldots, P$. Allerdings wird diese Vereinigung i.A. nicht mehr disjunkt sein wie zuvor bei der ersten Fassung des parallelen Verfeinerungsalgorithmus. Dies ergibt sich zwingend aus der dynamischen Lastverteilung, wie im Folgenden dargestellt wird.

Die Art der Aufteilung der globalen Gitter auf die Prozessoren muss den verschiedenen Anforderungen der einzelnen Programmkomponenten an die Lokalität der Daten gleichermaßen gerecht werden: Daten, auf die regelmäßige Zugriffe erfolgen, sollten lokal gespeichert werden und nicht durch teure Kommunikation erst beschafft werden müssen. Die einzigen Komponenten des Adaptionszyklus, welche auf den Geometriedaten aufbauen und damit deren Verteilung bestimmen, sind der Verfeinerungsalgorithmus mit anschließender Lastverteilung sowie die Diskretisierung und Fehlerschätzung.

Für die *Diskretisierung*, welche ja elementweise abläuft, wäre eine überlappungsfreie Speicherung der Tetraeder ausreichend, d.h. $\mathcal{G}_k(p) \cap \mathcal{G}_k(q) = \emptyset$ für $p \neq q$. Die Knoten, Kanten und Seitenflächen müssen überlappend gespeichert sein, um einen lokalen Zugriff der Tetraeder auf ihre Subsimplices zu ermöglichen.

Der *Verfeinerungsalgorithmus* arbeitet prinzipiell ebenfalls elementweise, benötigt aber neben dem Zugriff auf die Subsimplices auch Zugriff auf die Kinder eines Tetraeders. Solange ein zu Anfang überlappungsfrei aufgeteiltes Anfangsgitter ohne zwischenzeitliche Lastumverteilung verfeinert wird, befinden sich zu einem Elterntetraeder alle seine Kinder auf demselben Prozessor und umgekehrt zu einem Kind auch der zugehörige Elterntetraeder. Zwischenzeitliche Lastbalancierung ist aber im adaptiven Fall unbedingt vonnöten, da sonst große Lastunterschiede zwischen verschiedenen Prozessoren auftreten können, wenn z.B. immer nur Elemente *eines* Prozessors verfeinert werden.

Wird ein Tetraeder T auf Level $k > 0$ im Zuge der *Lastverteilung* auf einen anderen Prozessor verschoben, so muss, um die Referenz auf den Elterntetraeder $E(T)$ im Level $k-1$ zu repräsentieren, entweder eine Kopie von T auf dem alten Prozessor verbleiben ober aber $E(T)$ auch auf dem neuen Prozessor gespeichert werden.[1] Im letzteren Fall bedeutet dies entweder wiederum das Verschieben von $E(T)$ auf den neuen Prozessor oder eine dortige Speicherung als Kopie.

Das Zulassen von Kopien bedeutet zwar die Aufgabe der überlappungsfreien Tetraederspeicherung und somit eine Verkomplizierung der Datenstrukturen und des Verfeinerungsalgorithmus, aber der einzige Ausweg, der darin besteht, die Elterntetraedern ebenfalls zu verschieben, führt zu vielerlei Einschränkungen. Diese können dazu führen, dass eine balancierte Lastverteilung unmöglich wird, wie im Folgenden dargestellt wird.

Es wird nämlich eine Art Dominoeffekt angestossen, denn das Verschieben von $E(T)$ führt zum Mitverschieben von $E(E(T))$ usw. Gleichzeitig müssen aber auch alle Geschwistertetraeder von T mitverschoben werden, da sonst eine Überlappung des Elterntetraeders $E(T)$ entstünde. Dasselbe gilt genauso für die Geschwister von $E(T)$, $E(E(T))$ usw. Ingesamt ergibt sich also, dass immer eine ganze *Familienhierarchie* zusammen auf einem Prozessor gespeichert werden muss und nicht auseinandergerissen werden darf.

Zu einem Tetraeder T_0 auf Level $\ell(T_0) = 0$ (als allererstem Vorfahren) sei die Familienhierarchie $\mathsf{Fam}_J(T_0)$ folgendermaßen rekursiv definiert:

$$
\begin{aligned}
\mathsf{Fam}_0(T_0) &:= T_0\ , \\
\mathsf{Fam}_k(T_0) &:= \left(\bigcup_{T \in \mathsf{Fam}_{k-1}(T_0)} \mathcal{K}(T) \right) \cup T_0\ , \qquad k = 1, \ldots, J\ .
\end{aligned}
$$

Das Verschieben *eines* Tetraeders aus einer solchen Familienhierarchie $\mathsf{Fam}_J(T_0)$ würde also immer das Verschieben *aller* Tetraeder aus dieser Menge bedeuten.

[1] Beachte, dass Referenzen über Prozessorgrenzen hinweg nicht erlaubt sind, vgl. Bedingung (2.3).

Neben einer hohen Migrationsrate (und damit einem immensen Kommunikationsaufwand) bedeutet dies insbesondere, dass die Last nur extrem grob verteilt werden kann: Die Verteilung kann nicht tetraederweise (was dem Idealfall der feinstmöglichen Lastverteilung entspricht), sondern nur familienweise erfolgen.

Zum einen ist diese Art der familienbasierten Lastverteilung im Hinblick auf die Parallelisierung von Mehrgitterverfahren ungeeignet, da diese eine weitestgehend unabhängige Verteilung der einzelnen Gitterlevel erfordern. Zum anderen ergeben sich bei lokal adaptiver Verfeinerung zwangsläufig einschneidende Folgen für die Güte der Lastverteilung:

Bei stark lokaler Verfeinerung führt eine solch grobgranulare Lastverteilung zu einer äußerst schlecht balancierten (und damit unbrauchbaren) Partitionierung, wie folgendes zweidimensionale Beispiel zeigt:

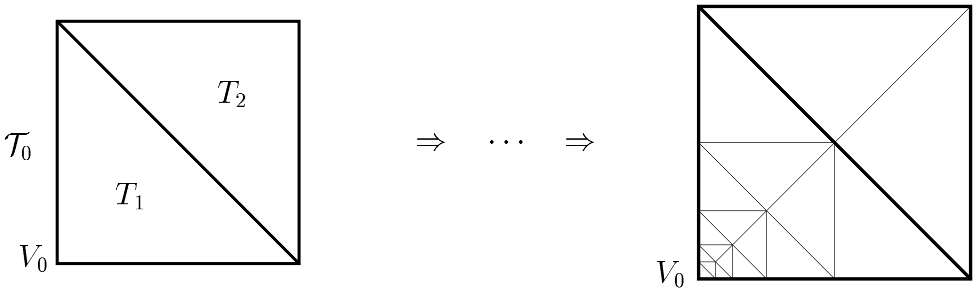


Abbildung 4.1: Ein Beispiel für ein stark lokal verfeinertes Gitter.

Das Einheitsquadrat $Q = [0,1]^2$ wird in zwei Dreiecke T_1 und T_2 zerlegt, welche die Anfangstriangulierung $\mathcal{T}_0 = \{T_1, T_2\}$ bilden. Nun wird sukzessive das Dreieck, welches am Ursprung V_0 anliegt, regulär verfeinert und der grüne Abschluss gebildet. Das Ergebnis nach viermaliger Verfeinerung zeigt Abbildung 4.1.

Bei einer Lastverteilung dieser Triangulierungshierarchie können nur die beiden Mengen $\mathsf{Fam}(T_1)$ und $\mathsf{Fam}(T_2)$ als Ganzes verteilt werden, eine Verteilung auf mehr als zwei Prozessoren ist also nicht möglich. Entscheidend aber ist das enorme Ungleichgewicht zwischen diesen beiden Mengen: Betrachtet man nur die feinste Triangulierung $\mathcal{T}_4$, so enthält $\mathsf{Fam}(T_1)$ alleine 16 der 18 Dreiecke in $\mathcal{T}_4$. Bezogen auf das gesamte Multilevelgitter entfallen 23 von 26 Dreiecken auf $\mathsf{Fam}(T_1)$. Da das Paket $\mathsf{Fam}(T_1)$, das fast 90% der Gesamtlast enthält, nicht aufgeteilt werden kann, ist eine balancierte Lastverteilung auf diese grobgranulare Art also nicht möglich.

Beachte, dass die Anzahl der Familienhierarchien (und damit die Anzahl der verteilbaren Stücke) gerade $|\mathcal{T}_0|$ entspricht; die Feinheit der Anfangstriangulierung bestimmt also die maximal mögliche Feinheit aller nachfolgenden Lastverteilungen. Bei stark lokaler Verfeinerung wie im obigen Beispiel müsste, um die Qualität der Lastverteilung zu sichern, allerdings $\mathcal{T}_0$ so fein gewählt werden, dass dies im

Widerspruch zur Idee der Adaptivität treten würde: Es soll ja nicht ganz Q, sondern nur die Ecke um V_0 fein aufgelöst werden! Außerdem wäre es dann nur eine Frage weniger weiterer lokaler Verfeinerungsschritte, um wiederum dasselbe Problem zu erhalten: die Existenz einer nicht aufteilbaren Familienhierarchie, die den überwiegenden Teil der Gesamtlast umfasst.

Will man also auf Überlappung der Tetraeder verzichten, so handelt man sich damit automatisch eine sehr grobgranulare Lastverteilung ein, die vor allem daher rührt, dass durch den Verzicht auf Überlappung Tetraeder *aller* Verfeinerungsstufen untrennbar zusammengefasst werden. Dies lässt sich nur umgehen, indem man jeweils nur Tetraeder weniger aufeinanderfolgender Level zusammenfasst, was bedeutet, dass Familienhierarchien auf einer bestimmten Stufe getrennt werden müssen. Da trotz dieser Trennlinie die Eltern-Kind-Referenzen weiterhin realisiert werden müssen, ist eine Überlappung der entsprechenden Elterntetraeder unumgänglich.

Das sich aus all diesen Überlegungen ergebende Fazit lautet also:

$\triangleright\triangleright$ *Adaptivität benötigt eine genügend feingranulare Lastverteilung. Diese kann nur realisiert werden, wenn man eine Überlappung der Elemente zulässt.*

Somit ist klar, dass das Zusammenspiel von Verfeinerung und Lastbalancierung eine „vertikale“ Überlappung der Tetraeder zwingend erfordert („vertikal“, weil durch die Überlappung die Eltern-Kind-Beziehung etabliert wird). Die genaue Anforderung an die verteilte Speicherung des Multilevelgitters soll im Folgenden formal spezifiziert werden.

4.2 Formale Spezifikation des verteilten Multilevelgitters

Um die Überlappung der Tetraeder genauer beschreiben zu können und bestimmte Eigenschaften dieser Überlappung zu beweisen, wird eine formale Spezifikation des verteilten Multilevelgitters aufgestellt. Dazu werden die Elemente aus $\mathcal{G}_k(p)$ in zwei Mengen $\mathsf{Ma}_k(p), \mathsf{Gh}_k(p) \subset \mathcal{G}_k(p)$ eingeteilt, welche eine Partition von $\mathcal{G}_k(p)$ bilden:

$$\mathsf{Ma}_k(p) \cup \mathsf{Gh}_k(p) = \mathcal{G}_k(p) \quad \text{und} \quad \mathsf{Ma}_k(p) \cap \mathsf{Gh}_k(p) = \emptyset. \tag{4.1}$$

Elemente $T \in \mathsf{Ma}_k(p)$ werden *Masterelemente*, $T \in \mathsf{Gh}_k(p)$ *Ghostelemente* genannt; man sagt auch, T werde auf Prozessor p als Master bzw. als Ghost gespeichert. Die Implementierung dieser Klassifizierung erfolgt im Rahmen von DDD durch die Vergabe entsprechender Prioritäten. Bedingung (4.1) besagt gerade, dass jeder auf einem Prozess p gespeicherte Tetraeder T entweder als Master oder als Ghost gespeichert ist, keinesfalls aber ein Master- und Ghostelement eines Tetraeders sich gleichzeitig auf ein und demselben Prozess befinden.

Die Master- und Ghostelemente besitzen folgende Bedeutung: Zu jedem Tetraeder T des globalen Multilevelgitters gebe es genau ein Masterelement auf einem der Prozessoren. Entstehen (durch Lastverteilung) Überlappungen des Masterelementes, also Kopien von T auf anderen Prozessoren, so werden diese dort als Ghost gespeichert. Die im Folgenden spezifizierte Form der Überlappung führt übrigens dazu, dass höchstens *eine* Ghostkopie entstehen kann, eine eventuelle Überlappung ist also „so dünn wie möglich".

Masterelemente des Levels k auf Prozessor p, die eine Überlappung in Form von Ghosts auf anderen Prozessoren besitzen, werden in der Menge $\mathsf{HasGh}_k(p)$ zusammengefasst:

$$\mathsf{HasGh}_k(p) := \mathsf{Ma}_k(p) \cap \bigcup_{q=1,\ldots,P} \mathsf{Gh}_k(q) \ .$$

Die grundlegende Idee bei der Überlappung ist folgende: Da im (seriellen) Verfeinerungsalgorithmus der Zugriff eines Tetraeders auf alle seine Kinder gewährleistet sein muss und dies bei paralleler Ausführung möglichst keine Kommunikation kosten soll, wird gefordert, dass alle Kinder eines verfeinerten Tetraeders T zusammen auf einem Prozessor p als Master gespeichert werden müssen. Ist T als Master selber auf einem anderen Prozessor $q \neq p$ gespeichert, so wird auf dem Prozess p eine Ghostkopie von T gespeichert, um die Eltern-Kind-Referenzen realisieren zu können.

Die Einschränkung, dass alle Kinder eines verfeinerten Tetraeders auf demselben Prozessor gespeichert sein müssen, zeigt sich bei der Lastverteilung in der Zusammenfassung von Geschwistertetraedern. Dies bedeutet – verglichen mit der feinstmöglichen, *tetraederweisen* Verteilung – eine gewisse Vergröberung, die aber vergleichsweise gering ist. Die Granularität ist immer noch fein genug, um auch bei stark lokaler Verfeinerung eine ausreichend balancierte Lastverteilung zu ermöglichen, da Tetraeder *verschiedener* Verfeinerungsstufen nicht zusammengefasst sind.

Ein wichtiger Schritt dieser Arbeit besteht nun darin, die obige Verteilungsidee in einer formalen Spezifikation festzuhalten. Zur Darstellung der Eltern-Kind-Beziehung wird dabei die in Definition 3.1 eingeführte Menge $\mathcal{K}(T)$ der Kinder eines Tetraeders T benutzt. Zur einfacheren Darstellung sei diese Definition folgendermaßen erweitert: Ist ein Tetraeder T unverfeinert, so soll $\mathcal{K}(T) = \emptyset$ gelten. Darüber hinaus wird der Einfachheit halber $\mathsf{Ma}_{J+1} := \emptyset$ eingeführt.

Definition 4.1 (zulässige verteilte Multilevelgitter) Die Folge

$$\tilde{\mathcal{M}} = (\mathcal{M}(1), \ldots \mathcal{M}(P))$$

der lokalen Multilevelgitter heißt *verteiltes Multilevelgitter*.

$\tilde{\mathcal{M}}$ heißt *zulässig*, falls für jedes Level $k = 0, \ldots, J$ die Bedingung (4.1) sowie die folgenden Bedingungen (M1) – (M5) gelten:

(M1) **Existenz:** Jedes Element aus Level k wird auf mindestens einem Prozessor durch ein Masterelement repräsentiert.

$$\bigcup_{p=1}^{P} \mathsf{Ma}_k(p) = \mathcal{G}_k$$

(M2) **Eindeutigkeit:** Zu jedem Element aus Level k gibt es höchstens ein Masterelement.

$$\forall\, p_1, p_2 : \qquad \mathsf{Ma}_k(p_1) \cap \mathsf{Ma}_k(p_2) \neq \emptyset \;\Rightarrow\; p_1 = p_2$$

(M3) **Elternlokalität:** Ein Masterelement und sein Elterntetraeder (als Master oder Ghost) sind auf demselben Prozessor gespeichert.

$$\forall\, p \quad \forall\, T \in \mathcal{G}_k \quad \forall\, T' \in \mathcal{K}(T) : \qquad T' \in \mathsf{Ma}_{k+1}(p) \;\Rightarrow\; T \in \mathcal{G}_k(p)$$

(M4) **Ghosteltern:** Ghostelemente besitzen stets Kinder.

$$\forall\, p \quad \forall\, T \in \mathsf{Gh}_k(p) : \qquad \mathcal{K}(T) \neq \emptyset$$

(M5) **Kinder von Ghosts:** Ghostelemente und deren Kinder sind auf demselben Prozessor gespeichert.

$$\forall\, p \quad \forall\, T \in \mathsf{Gh}_k(p) : \qquad \mathcal{K}(T) \subset \mathsf{Ma}_{k+1}(p)$$

Bemerkung 4.2

- Die Bedingung (M5) drückt gerade die oben beschriebene zentrale Eigenschaft der Verteilung aus, dass die Kinder eines verfeinerten Tetraeders zusammen als Master auf einem gemeinsamen Prozessor gespeichert sein sollen.

- Die Bedingungen (M1), (M2) sagen aus, dass die Masterelemente $\mathsf{Ma}_k(p)$, $p = 1, \ldots, P$ eine Partition des Gitters $\mathcal{G}_k$ bilden. Aus diesem Grund kann jedem Tetraeder eindeutig die Prozessornummer zugeordnet werden, auf dem das zugehörige Masterelement gespeichert ist. Die Abbildung $\mathsf{master} : \bigcup_{k=0}^{J} \mathcal{G}_k \to \{1, \ldots, P\}$ ist deswegen durch

$$\mathsf{master}(T) = p \quad :\Leftrightarrow \quad T \in \mathsf{Ma}_{\ell(T)}(p)$$

  wohldefiniert.

- Die Bedingungen (M3), (M4) schließlich sorgen dafür, dass Ghosts nur zur Realisierung einer Eltern-Kind-Referenz erzeugt werden und somit keine unnötige Überlappung von Tetraedern entsteht.

Aus den Bedingungen (M1) – (M5) lassen sich bereits einige Eigenschaften von zulässigen verteilten Multilevelgittern herleiten:

Satz 4.3

1. *Zu jedem Element aus Level k gibt es unter allen Prozessoren höchstens ein Ghostelement.*

$$\forall\, T \in \mathcal{G}_k \quad \forall\, p, q : \qquad T \in \mathsf{Gh}_k(p) \cap \mathsf{Gh}_k(q) \;\Rightarrow\; p = q$$

2. *Alle Kinder eines Tetraeders sind gemeinsam auf demselben Prozessor als Masterelemente gespeichert.*

$$\forall\, T \in \mathcal{G}_k \quad \exists\, p : \qquad \mathcal{K}(T) \subset \mathsf{Ma}_{k+1}(p)$$

Beweis:

1. Seien $T \in \mathcal{G}_k$ und p_1, p_2 mit $T \in \mathsf{Gh}_k(p_1) \cap \mathsf{Gh}_k(p_2)$. Dann folgt aus (M4) und (M5)

$$\emptyset \neq \mathcal{K}(T) \subset \mathsf{Ma}_{k+1}(p_1) \cap \mathsf{Ma}_{k+1}(p_2).$$

   Wegen (M2) ist also $p_1 = p_2$.

2. Sei $T \in \mathcal{G}_k$. Für $\mathcal{K}(T) = \emptyset$ ist nichts zu zeigen, sei also o.B.d.A. $\mathcal{K}(T) \neq \emptyset$. Wähle $T' \in \mathcal{K}(T) \subset \mathcal{G}_{k+1}$. (M1) impliziert

$$\exists\, p : \qquad T' \in \mathsf{Ma}_{k+1}(p),$$

   und zwar $p = \mathsf{master}(T')$. Mit (M3) folgt, dass T ebenfalls auf Prozessor p gespeichert wird: $T \in \mathcal{G}_k(p)$.

1. Fall: $T \in \mathsf{Gh}_k(p)$. Dann impliziert (M5): $\mathcal{K}(T) \subset \mathsf{Ma}_{k+1}(p)$.

2. Fall: $T \in \mathsf{Ma}_k(p)$. Angenommen, es existiere $T'' \in \mathcal{K}(T)$ mit $q := \mathsf{master}(T'') \neq p$. Dann ist wiederum wegen (M3) $T \in \mathcal{G}_k(q)$. Wegen $\mathsf{master}(T) = p \neq q$ muss

$$T \in \mathsf{Gh}_k(q)$$

gelten. Also ist wegen (M5) $\mathcal{K}(T) \subset \mathsf{Ma}_{k+1}(q)$, insbesondere gilt auch $T' \in \mathsf{Ma}_{k+1}(q)$. Somit folgt

$$T' \in \mathsf{Ma}_{k+1}(p) \cap \mathsf{Ma}_{k+1}(q) \; .$$

Dies ist aber wegen $p \neq q$ ein Widerspruch zu (M2). Also gilt für alle $T'' \in \mathcal{K}(T) \quad \mathsf{master}(T'') = p$.

□

4.3 Ein paralleler Verfeinerungsalgorithmus für adaptive Gitterverfeinerung auf zulässigen verteilten Multilevelgittern

Die erste Fassung des parallelen Verfeinerungsalgorithmus wurde in Abschnitt 3.5 vorgestellt; die Frage der Lastverteilung wurde dabei zunächst ausgeklammert, was die Verteilung der Geometriedaten stark vereinfachte: Die dort angenommene überlappungsfreie Speicherung der Tetraeder stellt einen Spezialfall eines verteilten zulässigen Multilevelgitters dar.

Aufbauend darauf soll nun der Algorithmus auf *allgemeine* zulässige verteilte Multilevelgitter ausgeweitet werden. Während in der ersten vorgestellten Version des Verfeinerungsalgorithmus die überlappungsfreie Speicherung der Tetraeder die Parallelisierung vereinfachte, muss nun die Überlappung der Tetraeder entsprechend berücksichtigt werden, was die Parallelisierung erheblich verkompliziert und weitere Kommunikation nötig macht.

Wie aus obigem Algorithmus ersichtlich, treten gegenüber der ersten Fassung des parallelen Verfeinerungsalgorithmus nur die Funktionen `ÜbermittleMarkierungen` und `SetzePrios` neu hinzu. Die übrigen Funktionen tauchten bereits in der ersten Fassung des Algorithmus auf, sind aber in der neuen Fassung teilweise verändert. Da diese Funktionen bereits vorgestellt wurden, werden jeweils nur die Änderungen beschrieben, die sich durch die Überlappung der Tetraeder ergeben.

```
Algorithm ParVerfeinerung( G_0(p), ..., G_J(p) )
{
    for ( k = J, ..., 0 ) do
    {
        BestimmeMarkierungen( G_k(p) );                          (1)
        ÜbermittleMarkierungen( G_k(p) );                        (2)
        AkkumuliereKantenzähler( G_k(p) );                       (3)
        SchließeGitter( G_k(p) );                                (4)
    }
    for ( k = 0, ..., J ) do if ( G_k(p) ≠ ∅ ) then
    {
        if ( k > 0 ) then SchließeGitter( G_k(p) );              (5)
        if ( k < J ) then VergröbereGitter( G_k(p) );            (6)
        DDD_IdentifyBegin();                                     (7)
            VerfeinereGitter( G_k(p) );                          (8)
        DDD_IdentifyEnd();                                       (9)
    }
    SetzePrios();                                                (10)
    BestimmeLetztesLevel();                                      (11)
}
```

Abbildung 4.2: Der parallele Verfeinerungsalgorithmus.

Um das grundlegende Prinzip dieser Änderungen besser verstehen zu können, sollen zu Anfang die Konsequenzen der Überlappung für die Speicherung einzelner Tetraeder genauer betrachtet werden. Dies wird die nötigen Änderungen in den einzelnen Funktionen motivieren und leichter verständlich machen.

Liegt bei einem Tetraeder T eine Überlappung vor, so bedeutet dies, dass T auf einem Prozessor p als Master und auf einem anderen Prozessor $q \neq p$ als Ghost gespeichert ist. Darüber hinaus gibt es nach Satz 4.3 keine weiteren Ghostkopien von T.

Wegen (M4) folgt weiterhin, dass T verfeinert ist, und mit (M5), dass seine Kinder $\mathcal{K}(T)$ sich allesamt auf Prozessor q als Masterelemente befinden. Deswegen werden alle Referenzen von T auf Kinder $T' \in \mathcal{K}(T)$ nur auf dem Ghostelement auf Prozessor q gespeichert, auf dem Masterelement von T auf Prozessor p hingegen nicht.

Bei Referenzen auf den Elterntetraeder dagegen verhält es sich genau umgekehrt: Wegen (M3) ist sichergestellt, dass zu jedem Masterelement auf Level $k > 0$ auch sein Elterntetraeder auf demselben Prozessor gespeichert ist. Dies legt nahe, die Referenz auf den Elterntetraeder (falls dieser existiert) immer nur auf dem Master zu speichern, nicht aber auf dem entsprechenden Ghost.

Zusammenfassend sei für die Arbeit des parallelen Verfeinerungsalgorithmus also Folgendes festgehalten:

- Aktionen, die auf die *Kinder* $\mathcal{K}(T)$ eines Tetraeders T wirken, werden im Fall einer überlappten Speicherung von T immer vom *Ghosttetraeder* aus vorgenommen.
- Aktionen, die auf den *Elterntetraeder* $E(T)$ eines Tetraeders T gerichtet sind, werden im Fall einer überlappten Speicherung von T immer vom *Mastertetraeder* aus durchgeführt.

Unter Berücksichtigung dieser Aussagen folgt nun die Beschreibung der in den einzelnen Funktionen erfolgten Änderungen.

4.3.1 Beschreibung der aufgerufenen Funktionen

BestimmeMarkierungen

Die bei einer Überlappung auftretenden Konsequenzen bezüglich der Speicherung von Kinder- bzw. Elternreferenzen haben folgende Auswirkungen auf *BestimmeMarkierungen*: Da überlappte Tetraeder immer verfeinert sind, kann der erste Block (1) – (3) für unverfeinerte Tetraeder unverändert übernommen werden.

Veränderungen ergeben sich erst für den Fall verfeinerter Tetraeder. Da dort die Verfeinerungsmarkierungen aufgrund von Informationen auf den Kindern gesetzt werden, kann dies nicht auf überlappten Mastern geschehen, da diese keine Referenzen auf ihre Kinder besitzen. Die entsprechende Entscheidung muss also auf den Ghostkopien getroffen werden, die Zugriff auf die Kindtetraeder haben. Anweisungen (4) – (10) des zweiten Blocks werden deswegen nur für $T \in \mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p)$ ausgeführt.

Die durch einen Ghost in (4) bzw. (7) gesetzte Markierung muss auch auf dem zugehörigen Master gesetzt werden, nachdem sie diesem per Kommunikation bekannt gemacht worden ist. Um das Absetzen vieler kleiner Nachrichten zu vermeiden, geschieht diese Kommunikation nicht an Ort und Stelle, sondern erst in `ÜbermittleMarkierungen`, wo die gesammelten Informationen zusammengefasst und über ein DDD-Interface ausgetauscht werden.

Ein weiteres Augenmerk gilt dem Erniedrigen bzw. Erhöhen der Kantenzähler in (5) bzw. (8), welches dem Setzen der Markierung folgt. Im Falle einer Überlappung dürfen die Kantenzähler nicht sowohl auf Master- als auch auf Ghostseite modifiziert werden, weil eine entsprechende Erhöhung bzw. Erniedrigung

```
Function BestimmeMarkierungen( $\mathcal{G}_k(p)$ )
{
    for ( $T \in \mathcal{G}_k(p)$ ) do {
        if ( T ist unverfeinert ) then
        {
            if ( T ist regulär ) and ( $\mathsf{mark}(T) = Ref$ ) then
            {
                $\mathsf{mark}(T) := RegRef$;                                  (1)
                Erhöhe Kantenzähler;                                            (2)
            }
            if ( $k = 0$ ) and ( $\mathsf{mark}(T) = Del$ ) then
                $\mathsf{mark}(T) := NoRef$;                                   (3)
        }
        else if ( $T \notin \mathsf{HasGh}_k(p)$ ) {
            if ( T ist regulär verfeinert ) then
            {
                if ( $\forall\, T' \in \mathcal{K}(T) : \mathsf{mark}(T') = Del$ ) then
                {
                    $\mathsf{mark}(T) := NoRef$;                               (4)
                    if ( $T \notin \mathsf{Gh}_k(p)$ )
                        Erniedrige Kantenzähler;                                (5)
                }
                for ( $T' \in \mathcal{K}(T)$ ) do
                    if ( $\mathsf{mark}(T') = Del$ ) then
                        $\mathsf{mark}(T') := NoRef$;                          (6)
            }
            else // T ist irregulär verfeinert
            {
                if  ( $\exists\, T' \in \mathcal{K}(T) : \mathsf{mark}(T') = Ref$ )
                or  ( mind. eine Kindkante, die nicht Kante von T ist,
                      ist zur Verfeinerung vorgesehen ) then
                {
                    $\mathsf{mark}(T) := RegRef$;                              (7)
                    if ( $T \notin \mathsf{Gh}_k(p)$ )
                        Erhöhe Kantenzähler;                                    (8)
                }
                else
                    $\mathsf{mark}(T) := NoRef$;                               (9)
                for ( $T' \in \mathcal{K}(T)$ ) do
                    $\mathsf{mark}(T') := NoRef$;                              (10)
            }
        }
    }
}
```

```
Function ÜbermittleMarkierungen( G_k(p) )
{
    for ( T ∈ Gh_k(p) ) do
        m_T := mark(T);                              (1)
    for ( q = 1,...,P,   q ≠ p ) do
        Sende {m_T : T ∈ Gh_k(p) ∩ Ma_k(q)}
        an Prozessor q;                              (2)
    Empfange entsprechende Nachrichten
    von anderen Prozessoren;                         (3)
    for ( T ∈ HasGh_k(p) ) do
        if ( T ist regulär verfeinert ) then
        {
            if ( m_T ≠ RegRef ) then
            {
                mark(T) := NoRef;                    (4)
                Erniedrige Kantenzähler;             (5)
            }
        }
        else // T ist irregulär verfeinert
        {
            if ( m_T = RegRef ) then
            {
                mark(T) := RegRef;                   (6)
                Erhöhe Kantenzähler;                 (7)
            }
            else
                mark(T) := NoRef;                    (8)
        }
}
```

eines Kantenzählers sonst in der akkumulierten Bilanz doppelt zu Buch schlagen würde.

Gemäß dem Grundsatz *„Modifikation von Kantenzählern nur auf Mastern“*, muss im Falle einer Überlappung die Modifikation in (6) bzw. (10) auf dem Master nachgeholt werden; dies geschieht ebenfalls in `ÜbermittleMarkierungen`.

`ÜbermittleMarkierungen`

Den obigen Ausführungen folgend, werden die auf überlappten Mastertetraedern in *BestimmeMarkierungen* aufgeschobenen Anweisungen in `ÜbermittleMarkierungen` nachgeholt. Dazu müssen die durch Ghosts in (4) bzw. (7) von *BestimmeMarkierungen* gesetzten Markierungen an die entsprechenden Master übermittelt werden. Die empfangenden Master setzen dann ihrerseits die jeweils empfangene Markierung und modifizieren die Kantenzähler entsprechend.

Einige Worte zur Implementierung: Die Kommunikation wird über ein DDD-Interface abgewickelt, das die Menge der überlappten Tetraeder auf Level k umfasst. Schritt (1) entspricht dabei dem Gather-Handler, der die in *Bestimme-Markierungen* gesetzten Markierungen auf den Ghosts einsammelt. Der Scatter-Handler wird durch die Schritte (4) – (8) repräsentiert, welche auf den empfangenden Mastern ausgeführt werden. Die Kommunikationsanweisungen (2) und (3) werden DDD-intern abgewickelt.

`AkkumuliereKantenzähler`

Die Funktion `AkkumuliereKantenzähler` ist gegenüber der ersten Fassung des parallelen Verfeinerungsalgorithmus unverändert.

SchließeGitter

Auch *SchließeGitter* ist gegenüber der ersten Fassung des parallelen Verfeinerungsalgorithmus unverändert: Da alle Kanten nach Aufruf von `Akkumuliere-Kantenzähler` die akkumulierten globalen Zählerwerte enthalten, ist das Kantenverfeinerungsmuster auf jedem Tetraeder bekannt, im Fall einer Überlappung also sowohl auf Master- als auch auf Ghostseite. Die Verfeinerungsregel kann also auf *jedem* Tetraeder ohne Kommunikationsaufwand wie gehabt bestimmt und gesetzt werden.

Function *SchließeGitter*($\mathcal{G}_k(p)$)
{
    for ($T \in \mathcal{G}_k(p)$) **do**
        if (*T ist regulär*) **and** ($\mathsf{mark}(T) \neq$ *RegRef*) **then**
        {
            Bestimme anhand der Kantenzähler
            die passende Verfeinerungsregel $0 \leq R \leq 63$; (1)
            if ($R =$ *NoRef*) **and** ($\mathsf{mark}(T) =$ *Del*) **then**
                mache nichts;
                // *NoRef-Markierung wird in*
                    BestimmeMarkierungen($\mathcal{G}_{k-1}(p)$) *gesetzt!*
            else
                $\mathsf{mark}(T) := R$; (2)
        }
}

VergröbereGitter

Die Aufgabe von *VergröbereGitter* besteht darin, diejenigen Knoten, Kanten, Seitenflächen und Tetraeder auf Level $k+1$ zu entfernen, welche aufgrund geänderter Verfeinerungsregeln nicht länger benötigt werden. Da hierbei Zugriff von den Eltern auf die Kindtetraeder erfolgt (3–5), werden beim Schleifendurchlauf die überlappten Master $\mathsf{HasGh}_k(p)$ ausgelassen.

Fallen durch eine geänderte Verfeinerungsregel gar alle Kinder eines Tetraeders T weg, d.h. ist $\mathsf{mark}(T) = \text{NoRef}$, so muss dafür Sorge getragen werden, dass die Bedingung (M4) nicht verletzt wird, welche besagt, dass Ghostelemente stets verfeinert sind. Besitzt also ein Ghost die NoRef-Markierung, so muss er im Sinne der Zulässigkeit des verteilten Multilevelgitters wegfallen.

Da zusätzlich der bislang überlappte Master, der sich auf einem anderen Prozess befindet, über den Wegfall der Überlappung in Kenntnis gesetzt werden muss, wird der Ghost in (6) mittels `DDD_XferDeleteObj` bei DDD abgemeldet. Beachte, dass dies das Vorziehen des `DDD_XferBegin`-Kommandos in (2) zur Folge hat.

Durch den Wegfall von Ghosts auf Level k werden eventuell auch einige Knoten, Kanten und Seitenflächen nicht mehr benötigt. Da diese sich aber nicht auf Level $k+1$, sondern auf darunterliegenden befinden, können diese nicht im Rahmen der generellen Funktionalität von *VergröbereGitter* aufgespürt und gelöscht werden. Dies muss also in einem Extraschritt (9) erfolgen. Beachte, dass dadurch auch Objekte auf Level 0 wegfallen können. Dies ist neu im Vergleich zur ersten Fassung des parallelen bzw. des seriellen Verfeinerungsalgorithmus, wo das Ausgangsgitter $\mathcal{G}_0$ immer unverändert blieb.

Bemerkung 4.4 Tatsächlich müssen eigentlich nur die Subsimplices auf Level 0 extra behandelt werden: Subsimplices auf Level $k > 0$, die aufgrund wegfallender Ghosts nicht mehr gebraucht werden, werden nämlich bereits im vorhergehenden Aufruf von *VergröbereGitter*($\mathcal{G}_{k-1}(p)$) gelöscht. Die Bestimmung dieser Subsimplices erfolgt dabei durch die Funktionen *BehandleGhosts* und *RetteGhostKnoten* in den Schritten (7) und (8), diese Funktionen werden weiter unten im Text vorgestellt.

Die Implementation des parallelen Verfeinerungsalgorithmus behandelt deswegen bei der Realisierung von Schritt (9) tatsächlich nur Objekte der Stufe 0. Um die Formulierung des Algorithmus nicht zusätzlich zu verkomplizieren, wurde dieses Detail im Hinblick auf eine bessere Verständlichkeit nicht erwähnt.

Zurück zur Hauptaufgabe von *VergröbereGitter*: dem Auffinden und Löschen nicht mehr benötigter Objekte auf Stufe $k+1$. Dazu werden zunächst alle Objekte dieser Stufe zur Entfernung markiert und anschließend diese Markierungen

Function *VergröbereGitter(* $\mathcal{G}_k(p)$ *)*
{
  Markiere alle Knoten, Kanten und Seitenflächen
  auf Level $k+1$ *für Entfernung;* (1)

  `DDD_XferBegin`*();* (2)

  for *(* $T \in \mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p)$ *)* **do if** *(* T *ist verfeinert)* **then**
  {
    if *(* T *ist entsprechend seiner Verfeinerung markiert)* **then**
      Lösche die Del-Markierungen aller
      Knoten, Kanten, Seitenflächen und Kinder von T*;* (3)
    else
    {
      for *(* $T' \in \mathcal{K}(T)$ *)* **do**
        $\mathsf{mark}(T') := Del$*;* (4)
      if *(* $\mathsf{mark}(T) \neq NoRef$ *)* **then**
        Behalte alle Knoten, Kanten, Seitenflächen und
        Kinder von T*, welche für die neue Verfeinerung*
        gemäß $\mathsf{mark}(T)$ *wiederbenutzt werden können;* (5)
      else if *(* $T \in \mathsf{Gh}_k(p)$ *)* // $\mathsf{mark}(T) = NoRef$
        Melde T *mittels* `DDD_XferDeleteObj`
        bei DDD ab und lösche T *anschließend;* (6)
    }
  }

  BehandleGhosts($\mathsf{Gh}_{k+1}(p)$ *);* (7)
  RetteGhostKnoten($\bigcup_{i \geq k+1} \mathsf{Gh}_i(p)$ *);* (8)

  Wenn (6) ausgeführt wurde, so bestimme alle Knoten,
  Kanten und Seitenflächen der weggefallenen Ghostelemente
  und markiere die nicht mehr gebrauchten für Entfernung; (9)

  Melde alle für Entfernung markierten Knoten, Kanten,
  Seitenflächen und Tetraeder bei DDD
  mittels `DDD_XferDeleteObj` *ab;* (10)

  `DDD_XferEnd`*();* (11)

  Lösche alle für Entfernung markierten Knoten,
  Kanten, Seitenflächen und Tetraeder; (12)
}

evtl. wieder rückgängig gemacht. Das Löschen der Entfernungmarkierungen findet dabei von den jeweiligen Elterntetraeder des Levels k aus statt. Im Falle eines Ghostelementes $T \in \mathsf{Gh}_{k+1}(p)$ muss aber beachtet werden, dass sich der Elterntetraeder $E(T)$ nicht auf demselben Prozess befinden muss wie T. Dies hat zur Folge, dass die Ghosts auf Level $k+1$ gesondert behandelt werden müssen.

Diese Arbeit übernimmt die Funktion *BehandleGhosts*: Alle Ghosts aus Level

$k + 1$, welche keine NoRef-Markierung besitzen, werden beibehalten und entfernen deshalb die Del-Markierung von ihren Knoten, Kanten und Seitenflächen. Mit NoRef markierte Ghosts dagegen fallen beim folgenden Aufruf von *VergröbereGitter* auf der nächsten Stufe weg (Schritt (6)) und modifizieren daher die Markierungen ihrer Subsimplices nicht.

Um die Menge $\mathsf{Gh}_{k*1}(p)$ aller Ghosts auf Level $k+1$ zu durchlaufen, wäre es ineffizient, über ganz $\mathcal{G}_{k+1}(p)$ zu iterieren, da der Anteil der Ghosts i.A. eher gering ist. Deswegen wird stattdessen das DDD-Interface, welches die Menge der überlappten Tetraeder auf Level $k+1$ umfasst, eingesetzt: Mittels `DDD_IFAExecLocal` wird über die Menge $\mathsf{Gh}_{k+1}(p) \cup \mathsf{HasGh}_{k+1}(p)$ iteriert und jeweils ein Handler aufgerufen, der auf den Ghostelementen die oben beschriebene Funktionalität ausführt. Da nur der lokale Teil des Interfaces angesprochen wird, findet dabei keinerlei Kommunikation statt.

Durch *BehandleGhosts*($\mathsf{Gh}_{k+1}(p)$) werden alle Kanten und Seitenflächen von verbleibenden Ghosts aus Level $k + 1$ vor dem Löschen gerettet. Eine gewisse Sonderstellung nehmen allerdings die Knoten ein: Ein regulärer Tetraeder T besitzt stets nur Kanten und Seitenflächen des gleichen Levels $\ell(T)$. Dies gilt insbesondere für Ghosts, da diese wegen (M4) stets verfeinert und damit regulär sind. Die Knoten von T können dagegen aus den Leveln $j \leq \ell(T)$ stammen.

Im Falle der überlappungsfreien Speicherung der Tetraeder, wie sie in der ersten Fassung des parallelen Verfeinerungsalgorithmus vorausgesetzt wurde, gibt es auf Prozessor p zu jedem Knoten V aus Level j einen Tetraeder $T \in \mathcal{G}_j(p)$ aus demselben Level, der V als Knoten besitzt. Um dies einzusehen, sei T' ein beliebiger auf dem Prozess p gespeicherter Tetraeder, der V besitzt; ein solches T' existiert, da es sonst den Knoten V nicht gäbe. Nach obiger Überlegung gilt

$$\ell(T') \geq j \ .$$

Im Falle $j = \ell(T')$ ist $T = T'$ gefunden, ansonsten ist T unter den Vorfahren von T' zu finden (betrachte E(T'), E(E(T')) usw.). Die Überlappungsfreiheit der Tetraeder garantiert dabei, dass alle Vorfahren von T' und somit auf jeden Fall auch T ebenfalls auf Prozessor p gespeichert sind.

Genau dies ist aber für ein zulässiges verteiltes Multilevelgitter im Allgemeinen nicht mehr der Fall: Die Überlappung der Tetraeder und damit die Trennung der Familienhierarchie kann dazu führen, dass es zu dem oben betrachteten Knoten V aus Level j keinen Tetraeder $T \in \mathcal{G}_j(p)$ gibt, welcher V besitzt.

Um zu einem solchen Knoten V aus Level j mit Sicherheit festzustellen, ob er noch weiterhin gebraucht wird, reicht es also nicht, die Menge der Tetraeder $\mathcal{G}_j(p)$ aus Level j zu betrachten. Zusätzlich müssen auch alle Ghosts der darüber liegenden Level miteinbezogen werden. Diese Arbeit leistet *RetteGhostKnoten*: Dazu entfernt jeder Ghost $T \in \bigcup_{i \geq j} \mathsf{Gh}_i(p)$ evtl. vorhandene Del-Markierungen von allen seinen Knoten. Realisiert wird dies wiederum wie in *BehandleGhosts* durch Aufrufe von `DDD_IFAExecLocal` für die Level i mit $j = k + 1 \leq i \leq J$.

Bemerkung 4.5 Die Funktion *VergröbereGitter* ist ein Beispiel dafür, dass die Einführung der Überlappung von Tetraedern mitunter weitreichende Folgen für die Parallelisierung birgt und diese zum Teil subtiler Art sind, weil sie im überlappungsfreien Fall nie auftreten. Während der Parallelisierung des seriellen Algorithmus tauchten immer wieder solche Probleme auf und erforderten teilweise Maßnahmen, die nur durch gewisse grundlegende Änderungen des Ausgangsalgorithmus realisiert werden konnten. Das Verstehen der Ursachen und Hintergründe dieser Probleme (sowie deren adäquate Lösung) haben sicherlich den Großteil der Entwicklungsarbeit des parallelen Verfeinerungsalgorithmus ausgemacht (und gleichzeitig die meisten Nerven gekostet).

VerfeinereGitter

Diese Funktion erzeugt die neuen Objekte der Stufe $k + 1$. Da im Falle einer Überlappung die Kinder nur durch den Ghost gespeichert werden, übernimmt dieser auch die Erzeugung der durch eine geänderte Verfeinerungsregel neu entstehenden Objekte. Deswegen werden beim Schleifendurchlauf die überlappten Master $\mathsf{HasGh}_k(p)$ ausgelassen.

Eine wesentliche Veränderung ergibt sich für die Identifizierung neu erzeugter Simplices, da durch die Überlappung neue, quasi „künstliche“ Prozessorränder entstehen. Dies soll durch ein kleines 2D-Beispiel illustriert werden:

Betrachte drei benachbarte Dreiecke A, B und C, die einen Ausschnitt aus einer Triangulierung darstellen. A und B sind jeweils grün verfeinert und auf zwei verschiedenen Prozessoren p und q überlappend gespeichert. C ist unverfeinert und nur auf Prozessor q gespeichert. Die beschriebene Ausgangssituation ist in Abbildung 4.3 dargestellt, wobei die Verfeinerungsregeln auf A und B durch gestrichelte Linien angedeutet sind. Die auf dem entsprechenden Prozessor gespeicherten Kinder sind unter ihrem Elterntetraeder angeordnet. A ist also auf Prozessor p als Master und auf Prozessor q als Ghost gespeichert, bei B verhält es sich gerade umgekehrt. Wie beschrieben, werden die Kinder jeweils auf der Ghostseite gespeichert.

Die Kinder von A und B werden nun zur Verfeinerung markiert, die Verfeinerungsmarkierungen von A und B werden also durch den Verfeinerungsalgorithmus auf reguläre Verfeinerung gesetzt, auf C wird der entsprechende grüne Abschluss gesetzt. Nachdem die grünen Kinder auf den Ghosts von A und B durch *VergröbereGitter* entfernt wurden, werden in *VerfeinereGitter* die neuen Kinder erzeugt und dabei u.a. auch neue Knoten erzeugt, die in Abbildung 4.4 fett dargestellt sind.

```
Function VerfeinereGitter( G_k(p) )
{
    if ( k = J ) then G_{k+1}(p) := ∅;                                  (1)
    for ( T ∈ G_k(p) \ HasGh_k(p) ) do
        if ( T ist nicht entsprechend seiner Verfeinerung markiert ) then
        {
            Verfeinere T entsprechend mark(T), d.h.
                Finde alle existierenden Kinder von T;                  (2)
                Erzeuge alle fehlenden Kinder von T
                als Master;                                             (3)
            for ( T' ∈ K(T) ) do
            {
                Finde alle existierenden Knoten,
                Kanten und Seitenflächen von T';                        (4)
                Erzeuge alle fehlenden Knoten,
                Kanten und Seitenflächen von T';                        (5)
                Identifiziere neu erzeugte Simplices
                auf Prozessorrändern durch Aufrufe
                von DDD_IdentifyObject miteinander;                     (6)
            }
        }
}
```

Die Frage ist nun: Welche der neu erzeugten Knoten müssen identifiziert werden und, wenn ja, mit welchen anderen Knoten? Sie liegen zwar alle auf überlappten Kanten, diese Überlappung täuscht aber, denn Knoten V_3 muss nicht identifiziert werden: Er liegt zwar auf einer überlappten Kante, diese Überlappung ist aber künstlich durch die Tetraederüberlappung von A entstanden.

Um nun echte Überlappung von künstlicher Überlappung zu unterscheiden, zieht man sich – jedenfalls gedanklich – wieder auf die überlappungsfreie Speicherung von Tetraedern zurück; die Darstellung der Eltern-Kind-Relationen spielt dabei keine Rolle, Eltern- und Kindtetraeder dürfen demnach auf verschiedenen Prozessoren gespeichert sein.

Man erhält diese überlappungsfreie Verteilung der Tetraeder, wenn man die überlappten Tetraeder $\mathsf{HasGh}_k(p)$ weglässt, denn diese dienen nur dazu, die Eltern-Kind-Relation darstellen zu können. Dann ergibt sich ein Bild wie in Abbildung 4.5, in dem die tatsächlich überlappten Kanten nun erkennbar sind. Damit ist klar, dass V_1 mit V_4 und V_2 mit V_5 identifiziert werden müssen, V_3 dagegen nicht.

Da das Weglassen von $\mathsf{HasGh}(p)$ technisch nicht möglich ist, kann man dies stattdessen „simulieren“, indem man zu allen Knoten, Kanten und Seitenflächen als zusätzliche Information speichert, ob sie

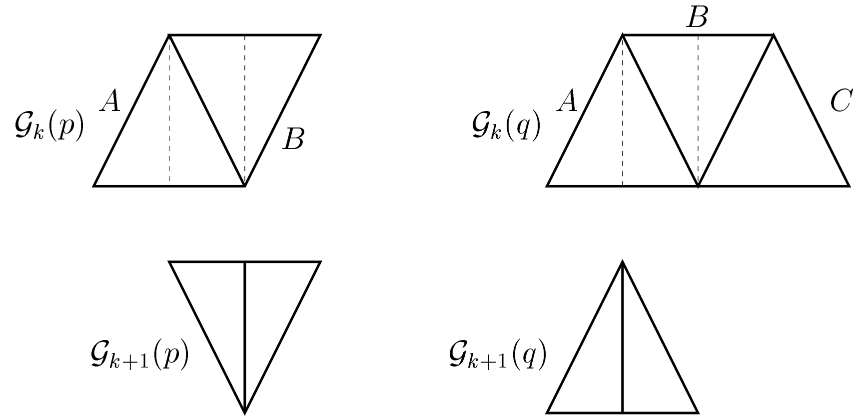


Abbildung 4.3: Beispiel eines verteilten Gitters mit Tetraederüberlappung.

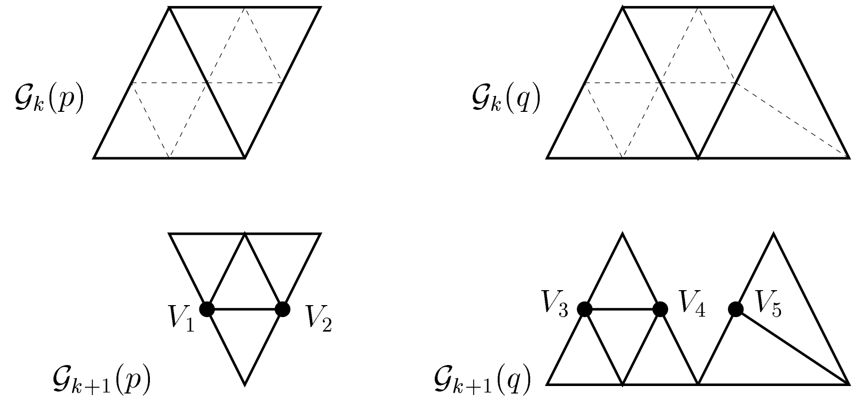


Abbildung 4.4: Nach der Verfeinerung sind neue Knoten entstanden.

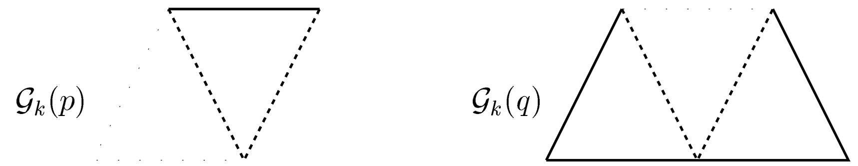


Abbildung 4.5: Gitter ohne überlappte Master. Die tatsächlich überlappten Kanten sind gestrichelt dargestellt.

(A) zu einem Ghost oder nichtüberlappten Master gehören, oder aber

(B) nur zu überlappten Mastern gehören.

Dies wird in der Implementierung wiederum durch Prioritäten realisiert, diese seien mit PrioA bzw. PrioB bezeichnet. Die tatsächlichen Überlappungen erhält man nun, wenn man Simplices mit PrioB nicht beachtet, weil sie ausschließlich zu überlappten Mastertetraedern gehören.

Dieser Idee folgend, wird nun die bei der ersten Fassung des parallelen Verfeinerungsalgorithmus gegebene Definition 3.14 des Prozessorrandes modifiziert und durch folgende ersetzt:

Definition 4.6 (Prozessorrand) Für einen Subsimplex S bezeichne die Menge

$$\mathsf{procs}_A(S) \quad := \quad \{p : 1 \leq p \leq P,\ S \text{ ist auf Prozessor } p \text{ mit Priorität PrioA gespeichert}\}$$

gerade die Prozessoren, auf denen S mit Priorität PrioA gespeichert ist; d.h. S gehört zu einem Tetraeder, der kein überlappter Master ist. Man sagt, S liege auf einem *Prozessorrand*, wenn S auf mehreren Prozessoren mit Priorität PrioA gespeichert ist, also

$$|\mathsf{procs}_A(S)| \geq 2$$

gilt. In diesem Fall spricht man auch von einer *essentiellen Überlappung.*

Bemerkung 4.7 Die alte Definition 3.14 ist als Spezialfall weiterhin in Definition 4.6 enthalten: Im Falle der überlappungsfreien Speicherung von Tetraedern besitzt nämlich jeder Subsimplex S die Priorität PrioA, mithin sind die Mengen $\mathsf{procs}(S)$ und $\mathsf{procs}_A(S)$ identisch.

Bei der Vorstellung der ersten Fassung des Verfeinerungsalgorithmus wurde in der dortigen Beschreibung der Funktion *VerfeinereGitter* bereits der Identifikationsvorgang zur Zusammenfassung neu erzeugter Subsimplices dargestellt. Der für die hier beschriebene Fassung von *VerfeinereGitter* gültige Identifikationmechanismus leitet sich daraus ab, indem man jeweils die geänderte Definition des Prozessorrandes zugrunde legt sowie jeweils die Menge $\mathsf{procs}(S)$ durch $\mathsf{procs}_A(S)$ ersetzt. Auf eine nochmalige Darstellung wird deswegen verzichtet. Bis auf die Tatsache, dass nur noch PrioA-Objekte miteinander identifiziert werden, gelten die dargestellten grundsätzlichen Überlegungen zur Identifizierung unverändert.

`SetzePrios`

In dieser Funktion werden alle Subsimplices durch Prioritäten gekennzeichnet, wie in der Beschreibung der Funktion *VerfeinereGitter* erläutert. Da diese Prioritäten sich durch die Ent- und Verfeinerung von Tetraedern teilweise geändert haben könnten, müssen sie, nachdem die Phase II abgeschlossen wurde, neu bestimmt werden.

Dazu markiert zunächst jeder überlappte Master alle seine Knoten, Kanten und Seitenflächen mit PrioB (1). Anschließend markieren alle übrigen Tetraeder (also Ghosts und nichtüberlappte Master) ihre Subsimplices mit PrioA (2). Danach sind tatsächlich nur noch solche Subsimplices mit PrioB markiert, welche ausschließlich zu überlappten Mastertetraedern gehören, denn ansonsten wären sie in Schritt (2) mit PrioA markiert worden.

Function `SetzePrios`*()*
{
  for *(* $T \in \bigcup_{k=0,\dots,J} \mathsf{HasGh}_k(p)$ *)* **do**
    Markiere alle Knoten, Kanten und
    Seitenflächen von T mit PrioB; *(1)*
  for *(* $T \in \bigcup_{k=0,\dots,J} (\mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p))$ *)* **do**
    Markiere alle Knoten, Kanten und
    Seitenflächen von T mit PrioA; *(2)*
}

`BestimmeLetztesLevel`

Die Funktion `BestimmeLetztesLevel` ist gegenüber der ersten Fassung des parallelen Verfeinerungsalgorithmus unverändert.

4.4 Korrektheit des parallelen Verfeinerungsalgorithmus

Dieser Abschnitt beschäftigt sich mit der Korrektheit des vorgestellten parallelen Verfeinerungsalgorithmus. Damit ist gemeint, ob – ein zulässiges Multilevelgitter als Eingabe vorausgesetzt – der Algorithmus durchgeführt werden kann und das Ausgabemultilevelgitter wieder zulässig ist. Die positive Antwort hält folgender Satz bereit:

Satz 4.8 (Wohldefiniertheit und Zulässigkeit) *Sei das verteilte Multilevelgitter*

$$\tilde{\mathcal{M}}^{\mathrm{alt}} = \left(\mathcal{M}^{\mathrm{alt}}(1), \dots, \mathcal{M}^{\mathrm{alt}}(P)\right)$$

zulässig, wobei die Folge $\mathcal{M}^{\mathrm{alt}}(p) = \left(\mathcal{G}_0^{\mathrm{alt}}(p), \dots, \mathcal{G}_J^{\mathrm{alt}}(p)\right)$ *jeweils das lokale Multilevelgitter auf Prozessor* p *bezeichne.* $\tilde{\mathcal{M}}^{\mathrm{alt}}$ *diene dem parallelen Verfeinerungsalgorithmus als Eingabemultilevelgitter. Dann gilt:*

a) Der Verfeinerungsalgorithmus ist wohldefiniert.

b) Das als Ausgabe durch den Verfeinerungsalgorithmus erzeugte verteilte Multilevelgitter $\tilde{\mathcal{M}}^{\mathrm{neu}} = \left(\mathcal{M}^{\mathrm{neu}}(1), \dots, \mathcal{M}^{\mathrm{neu}}(P)\right)$ *ist wieder zulässig.*

Beweis:

a) Zu zeigen ist, dass Daten, auf die zugegriffen wird, auch tatsächlich lokal vorhanden sind. Da ein Tetraeder stets alle seine Knoten, Kanten und Seitenflächen speichert, ist der Zugriff auf die Subsimplices sichergestellt. Kritisch ist allein der Zugriff auf den Elterntetraeder sowie auf die Kindtetraeder – jeweils vorausgesetzt, dass diese existieren.

   Im gesamten Verfeinerungsalgorithmus findet kein expliziter Zugriff auf den Elterntetraeder statt. Implizit erfolgt dieser allerdings bei dem Test, ob ein Tetraeder T regulär ist; dies ist nämlich genau dann wahr, wenn $\ell(T) = 0$ oder $E(T)$ regulär verfeinert ist.

   Dieser für ein Ghostelement T kritische Zugriff kann aber vermieden werden, weil Ghosts stets regulär sind: Wäre $T \in \mathsf{Gh}_k(p)$ irregulär, so würde dies bedeuten, dass T unverfeinert wäre, was im Widerspruch zu (M4) steht.

   Der Zugriff von Tetraedern auf ihre Kindtetraeder, der an vielen Stellen des Algorithmus auftaucht, ist nur von solchen Tetraedern möglich, die keine überlappten Master darstellen, d.h. nur für $T \in \mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p)$. Nach einem detaillierten Blick auf die einzelnen aufgerufenen Funktionen ist aber klar, dass dies immer gegeben ist.

   Somit ist der parallele Verfeinerungsalgorithmus auf einem zulässig verteilten Multilevelgitter durchführbar.

b) Sei $\tilde{\mathcal{M}}^{\text{neu}} = \big(\mathcal{M}^{\text{neu}}(1), \ldots, \mathcal{M}^{\text{neu}}(P)\big)$ das erzeugte Multilevelgitter mit

$$\mathcal{M}^{\text{neu}}(p) = \big(\mathcal{G}_0^{\text{neu}}(p), \ldots, \mathcal{G}_{J'}^{\text{neu}}(p)\big) \ .$$

   Für die Zulässigkeit von $\tilde{\mathcal{M}}^{\text{neu}}$ muss für jedes Gitterlevel $k = 0, \ldots, J'$ gezeigt werden, dass die Bedingungen (M1) – (M5) gelten. Um diese Bedingungen einfacher von denen für das Eingabemultilevelgitter $\tilde{\mathcal{M}}^{\text{alt}}$ unterscheiden zu können, werden sie für das Ausgabemultilevelgitter $\tilde{\mathcal{M}}^{\text{neu}}$ mit (M1') – (M5') bezeichnet.

   Zunächst sei bemerkt, dass die neuen Master- und Ghostelemente $\mathsf{Ma}_k^{\text{neu}}(p)$ und $\mathsf{Gh}_k^{\text{neu}}(p)$ wiederum eine Partition von $\mathcal{G}_k^{\text{neu}}(p)$ bilden, die Bedingung (4.1) gilt also auch für das neue Multilevelgitter $\tilde{\mathcal{M}}^{\text{neu}}$. Dies wird klar, wenn man sich überlegt, dass $\mathcal{G}_k^{\text{neu}}(p)$ aus $\mathcal{G}_k^{\text{alt}}(p)$ durch Wegfallen oder Hinzufügen von Tetraedern entsteht. Durch das Wegfallen von Tetraedern wird die Bedingung (4.1) nicht verletzt. Ein neu hinzukommender Tetraeder T wird in *VerfeinereGitter* (3) immer nur als Master erzeugt, es entstehen also aus $\mathcal{G}_k(p) = \mathsf{Ma}_k(p) \dot\cup \mathsf{Gh}_k(p)$ die Mengen

$$\begin{gathered}\mathcal{G}_k'(p) = G_k(p) \cup \{T\}, \\ \mathsf{Ma}_k'(p) = \mathsf{Ma}_k(p) \cup \{T\} \qquad \text{und} \qquad \mathsf{Gh}_k'(p) = \mathsf{Gh}_k(p) \ ,\end{gathered}$$

die wiederum der Partitionsbedingung (4.1) genügen.

Zu zeigen bleiben die Bedingungen (M1') – (M5') für die Gitter $\mathcal{G}_k^{\text{neu}}(p)$, $k = 0, \ldots, J'$. Wegen $|J - J'| \leq 1$ werden drei Fälle unterschieden:

1. $J' = J - 1$: Dieser Fall wird in Satz 4.9 bewiesen.
2. $J' = J$: Die Gitter $\mathcal{G}_k^{\text{neu}}(p), k = 0, \ldots, J-1$ erfüllen (M1') – (M5') (Satz 4.9). Für das Gitter $\mathcal{G}_J^{\text{neu}}(p)$ gilt $\mathcal{G}_J^{\text{neu}}(p) = \mathsf{Ma}_J^{\text{neu}}(p)$, denn
   - das Eingabegitter $\mathcal{G}_J^{\text{alt}}(p)$ enthält wegen (M4) nur Master, und
   - $\mathcal{G}_J^{\text{neu}}(p)$ entsteht aus $\mathcal{G}_J^{\text{alt}}(p)$, dabei fallen entweder einige Tetraeder aus $\mathcal{G}_J^{\text{alt}}(p)$ weg oder aber es entstehen neue durch Verfeinerung eines Tetraeders $T \in \mathcal{G}_{J-1}^{\text{neu}}$. Letzteres geschieht in *VerfeinereGitter* (3), wo alle neu erzeugten Kinder aus $\mathcal{K}^{\text{neu}}(T)$ als Master angelegt werden.

   Insbesondere gilt $\mathsf{Gh}_J^{\text{neu}}(p) = \emptyset$, also gibt es keine Tetraederüberlappung, und (M1') – (M5') sind damit automatisch erfüllt.
3. $J' = J + 1$: Für die Gitterlevel $k = 0, \ldots, J$ gelten die Bemerkungen zu Fall 2 genauso, also muss die Behauptung nur noch für $\mathcal{G}_{J+1}^{\text{neu}}(p)$ gezeigt werden. Da $\mathcal{G}_{J+1}^{\text{neu}}(p)$ nur aus neu erzeugten Tetraedern besteht, die allesamt Master sind, gilt $\mathcal{G}_{J+1}^{\text{neu}}(p) = \mathsf{Ma}_{J+1}^{\text{neu}}(p)$ bzw. $\mathsf{Gh}_{J+1}^{\text{neu}}(p) = \emptyset$. Da also keinerlei Überlappung von Tetraedern vorliegt, folgt wiederum direkt, dass $\mathcal{G}_{J+1}^{\text{neu}}(p)$ die Bedingungen (M1') – (M5') erfüllt.

□

Die Hauptarbeit des Beweises wurde in folgenden Satz verlagert:

Satz 4.9 (Zulässigkeit von $\tilde{\mathcal{M}}^{\text{neu}}$) *Seien wie in Satz 4.8 die verteilten Multilevelgitter $\tilde{\mathcal{M}}^{\text{alt}}$ die Eingabe bzw. $\tilde{\mathcal{M}}^{\text{neu}}$ die Ausgabe des parallelen Verfeinerungsalgorithmus. Weiterhin sei $\tilde{\mathcal{M}}^{\text{alt}}$ zulässig, d.h. $\tilde{\mathcal{M}}^{\text{alt}}$ erfülle (M1) – (M5) für die Level $k = 0, \ldots, J$. Dann erfüllt auch $\tilde{\mathcal{M}}^{\text{neu}}$ die entsprechenden Bedingungen (M1') – (M5') für die Level $k = 0, \ldots, J-1$.*

Beachte, dass die Level $k = 0, \ldots, J-1$ in $\tilde{\mathcal{M}}^{\text{neu}}$ auf jeden Fall existieren und bereits in $\tilde{\mathcal{M}}^{\text{alt}}$ vorhanden waren, so dass für die alten Gitterlevel die Bedingungen (M1) – (M5) ausgenutzt werden können. Für den Beweis dieses zentralen Satzes werden allerdings zunächst einige Hilfsaussagen benötigt, die eine einfachere Beweisführung ermöglichen.

Das erste Lemma beschäftigt sich mit den Tetraedern, die beim Übergang von $\tilde{\mathcal{M}}^{\text{alt}}$ zu $\tilde{\mathcal{M}}^{\text{neu}}$ neu hinzugekommen sind. Die Aussage ist zwar mehr oder weniger trivial, aber nützlich, um beim Beweis von Satz 4.9 nicht mehrfach den Code des Algorithmus zitieren zu müssen.

Lemma 4.10 (Lemma 1) Aus dem Level $k \geq 1$ sei $T \in \mathcal{G}_k^{\text{neu}}$ ein Tetraeder, der im alten Multilevelgitter noch nicht vorhanden war, d.h. $T \notin \mathcal{G}_k^{\text{alt}}$. Dann gilt:

(a) Alle neu entstandenen Tetraeder sind Master.

$$\exists\, p \quad T \in \mathsf{Ma}_k^{\text{neu}}(p)$$

(b)

$$T \in \mathsf{Ma}_k^{\text{neu}}(p) \quad \Rightarrow \quad E(T) \in \mathcal{G}_{k-1}^{\text{neu}}(p)$$

Beweis:

(a) Ist $T \in \mathcal{G}_k^{\text{neu}} \setminus \mathcal{G}_k^{\text{alt}}$ ein neu hinzugekommener Tetraeder, so kann dieser nur in *VerfeinereGitter* (3) auf einem Prozessor $p \in \{1, \ldots, P\}$ entstanden sein. Dort wird T aber als Master erzeugt, also $T \in \mathsf{Ma}_k^{\text{neu}}(p)$.

(b) Da die Verfeinerung vom Elterntetraeder $E(T) \in \mathcal{G}_{k-1}^{\text{alt}}$ aus stattfindet, hat dieser beim Aufruf von *VerfeinereGitter*($\mathcal{G}_{k-1}(p)$) noch existiert. $E(T)$ ist auch *danach* nicht gelöscht worden, da dies nur im Aufruf von *VergröbereGitter* auf einem Level $\ell < k$ geschehen sein könnte; Phase II wird aber *bottom-up* abgearbeitet. Also ist $E(T)$ auch nach Ablauf des Verfeinerungsalgorithmus noch auf Prozessor p gespeichert.

□

Das zweite Lemma trifft im Gegensatz zu Lemma 1 einige Aussagen zu Tetraedern aus $\tilde{\mathcal{M}}^{\text{neu}}$, die aus $\tilde{\mathcal{M}}^{\text{alt}}$ übernommen wurden.

Lemma 4.11 (Lemma 2) Aus dem Level $k \geq 0$ sei $T \in \mathcal{G}_k^{\text{neu}}$ ein Tetraeder, der bereits im alten Multilevelgitter vorhanden war, d.h. $T \in \mathcal{G}_k^{\text{alt}}$. Dann gilt:

(a) Übernommene Masterelemente wechseln nicht den Prozessor.

$$T \in \mathsf{Ma}_k^{\text{alt}}(p) \quad \Leftrightarrow \quad T \in \mathsf{Ma}_k^{\text{neu}}(p)$$

(b) Übernommene Tetraeder, die in $\tilde{\mathcal{M}}^{\text{neu}}$ verfeinert sind, bleiben auf denselben Prozessoren gespeichert wie zuvor.

$$T \in \mathcal{G}_k^{\text{alt}}(p) \text{ und } \mathcal{K}^{\text{neu}}(T) \neq \emptyset \quad \Rightarrow \quad T \in \mathcal{G}_k^{\text{neu}}(p)$$

Beweis: Sei $T \in \mathcal{G}_k^{\text{alt}} \cap \mathcal{G}_k^{\text{neu}}$ ein aus $\tilde{\mathcal{M}}^{\text{alt}}$ übernommener Tetraeder.

(a) Teil (a) folgt aus der Tatsache, dass im gesamten Verfeinerungsalgorithmus die Prioritäten der Tetraeder nicht verändert werden und keine Verschiebung der Tetraeder in andere Level oder auf andere Prozessoren erfolgt.

(b) Teil (b) muss nur noch für $T \in \mathsf{Gh}_k^{\text{alt}}(p)$ gezeigt werden, ist nämlich T ein Master, so folgt (b) direkt aus (a). Sei also

$$T \in \mathsf{Gh}_k^{\text{alt}}(p) \text{ mit } \mathcal{K}^{\text{neu}}(T) \neq \emptyset \ ,$$

d.h. T ist in $\tilde{\mathcal{M}}^{\text{neu}}$ verfeinert. Deswegen trägt T die neue Markierung

$$\mathsf{mark}^{\text{neu}}(T) \neq \text{NoRef} \ .$$

Die einzigen Stellen, aufgrund derer T im Verfeinerungsalgorithmus auf Prozessor p wegfallen könnte, sind die Anweisungen (5) und (6) von *VergröbereGitter*; aufgrund der Markierung kommt nur noch (5) in Frage. Falls der Elterntetraeder $T_0 := E(T)$ von T nicht auf Prozessor p gespeichert ist, also $T_0 \notin \mathcal{G}_{k-1}^{\text{alt}}(p)$, so ist nichts zu zeigen, da dann Anweisung (5) nicht für $T \in \mathsf{Gh}_k^{\text{alt}}(p)$ ausgeführt wird. Sei also

$$T_0 = E(T) \in \mathcal{G}_{k-1}^{\text{alt}}(p) \ .$$

Dann wird die Anweisung (5) in *VergröbereGitter*($\mathcal{G}_{k-1}(p)$) vom Elterntetraeder T_0 aus auf seinen Kindern $\mathcal{K}^{\text{alt}}(T_0)$ ausgeführt. Die Frage ist nun, ob die in (4) gesetzte Del-Markierung in Anweisung (5) wieder von $T \in \mathcal{K}^{\text{alt}}(T_0)$ entfernt wird oder nicht.

Wegen $T \in \mathcal{G}_k^{\text{neu}}$ ist aber T_0 in $\tilde{\mathcal{M}}^{\text{neu}}$ ebenfalls verfeinert, und es gilt $T \in \mathcal{K}^{\text{neu}}(T_0)$. Da T in der neuen Verfeinerung von T_0 vorkommt, löscht also T_0 die Del-Markierung von $T \in \mathsf{Gh}_k^{\text{alt}}(p)$ in Anweisung (5), um T für die neue Verfeinerung von T_0 zu behalten. Deswegen bleibt T auf Prozessor p als Ghostkopie erhalten, und somit gilt $T \in \mathcal{G}_k^{\text{neu}}(p)$.

□

Beachte, dass Aussage (b) für in $\tilde{\mathcal{M}}^{\text{neu}}$ unverfeinerte Tetraeder nicht immer gelten muss, da eventuelle Ghostkopien in diesem Fall auf den entsprechenden Prozessoren wegfallen. Ebenso kann es in $\tilde{\mathcal{M}}^{\text{neu}}$ verfeinerte Tetraeder geben, die in $\tilde{\mathcal{M}}^{\text{alt}}$ noch gar nicht existierten.

Das dritte Lemma besagt, dass in $\tilde{\mathcal{M}}^{\text{neu}}$ vorkommende Ghosts aus $\tilde{\mathcal{M}}^{\text{alt}}$ stammen müssen:

Lemma 4.12 (Lemma 3)

$$T \in \mathsf{Gh}_k^{\text{neu}}(p) \quad \Rightarrow \quad T \in \mathsf{Gh}_k^{\text{alt}}(p)$$

Beweis: Da neu erzeugte Tetraeder immer als Master angelegt werden, muss $T \in \mathsf{Gh}_k^{\text{neu}}(p)$ auch schon vorher als Ghost auf Prozessor p gespeichert gewesen sein. $\square$

Nun sind alle Hilfsaussagen für den im Folgenden geführten Beweis von Satz 4.9 zusammengetragen:

Beweis: (Satz 4.9)

zu (M1'): Es ist nach Definition $\mathcal{G}_k^{\text{neu}} = \bigcup_{p=1}^{P} \mathcal{G}_k^{\text{neu}}(p)$. Zeige

$$\bigcup_{p=1}^{P} \mathsf{Ma}_k^{\text{neu}}(p) = \mathcal{G}_k^{\text{neu}} .$$

Wegen $\mathsf{Ma}_k^{\text{neu}}(p) \subset \mathcal{G}_k^{\text{neu}}(p)$ ist die Inklusion „$\subseteq$" trivial. Zum Nachweis von „$\supseteq$" sei $T \in \mathcal{G}_k^{\text{neu}}$.

1. Fall: $T \in \mathcal{G}_k^{\text{alt}}$, d.h. T war schon in $\tilde{\mathcal{M}}^{\text{alt}}$ vorhanden.

$$\stackrel{\text{(M1)}}{\Rightarrow} \qquad \exists\, p \quad T \in \mathsf{Ma}_k^{\text{alt}}(p) \quad \stackrel{\text{La2(a)}}{\Rightarrow} \quad T \in \mathsf{Ma}_k^{\text{neu}}(p) .$$

2. Fall: $T \notin \mathcal{G}_k^{\text{alt}}$. Da auf Level 0 keine neuen Tetraeder entstehen, gilt $k \geq 1$, und Lemma 1(a) liefert:

$$\exists\, p \quad T \in \mathsf{Ma}_k^{\text{neu}}(p) .$$

zu (M2'): Sei

$$T \in \mathsf{Ma}_k^{\text{neu}}(p) \cap \mathsf{Ma}_k^{\text{neu}}(q) .$$

Zeige, dass dann $p = q$ gilt.

Der Beweis erfolgt per Induktion über das Gitterlevel k. Für $k = 0$ ist wegen $\mathsf{Ma}_0^{\text{neu}}(r) = \mathsf{Ma}_0^{\text{alt}}(r)$ für $r = 1, \ldots, P$ nichts zu zeigen: Da die Master auf Level 0 immer unverändert bleiben, folgt (M2') sofort aus (M2) für $\tilde{\mathcal{M}}^{\text{alt}}$.

Sei nun $k \geq 1$, und (M2') sei für die Level ℓ, $0 \leq \ell < k$ bereits bewiesen.

1. Fall: $T \in \mathcal{G}_k^{\text{alt}}$, d.h. T war schon in $\tilde{\mathcal{M}}^{\text{alt}}$ vorhanden. Dann liefert Lemma 2(a)

$$T \in \mathsf{Ma}_k^{\text{alt}}(p) \quad \text{und} \quad T \in \mathsf{Ma}_k^{\text{alt}}(q) .$$

Mit (M2) folgt $p = q$.

2. Fall: $T \notin \mathcal{G}_k^{\text{alt}}$. Wegen $k \geq 1$ folgt aus Lemma 1(b)

$$E(T) \in \mathcal{G}_{k-1}^{\text{neu}}(p) \cap \mathcal{G}_{k-1}^{\text{neu}}(q) .$$

Hieraus ergeben sich drei Alternativen für die Speicherung von $E(T)$ als Master oder Ghost auf Prozessor p und q:

1. $E(T)$ ist auf Prozessor p und q als Master gespeichert, also $E(T) \in \mathsf{Ma}^{\text{neu}}_{k-1}(p) \cap \mathsf{Ma}^{\text{neu}}_{k-1}(q)$. Nach Induktion folgt dann aus (M2') für Level $k-1$ die Gleichheit $p = q$.
2. $E(T)$ ist auf Prozessor p und q als Ghost gespeichert, also $E(T) \in \mathsf{Gh}^{\text{neu}}_{k-1}(p) \cap \mathsf{Gh}^{\text{neu}}_{k-1}(q)$. Lemma 3 impliziert dann
$$E(T) \in \mathsf{Gh}^{\text{alt}}_{k-1}(p) \cap \mathsf{Gh}^{\text{alt}}_{k-1}(q) \ .$$
Da $\tilde{\mathcal{M}}^{\text{alt}}$ zulässig ist, folgt unter Beachtung von Satz 4.3, dass $p = q$ ist.
3. $E(T)$ ist auf dem einen Prozessor als Ghost, auf dem anderen als Master gespeichert. Ohne Einschränkung gelte
$$E(T) \in \mathsf{Gh}^{\text{neu}}_{k-1}(p) \quad \text{und} \quad E(T) \in \mathsf{Ma}^{\text{neu}}_{k-1}(q) \ .$$
Lemma 3 besagt $E(T) \in \mathsf{Gh}^{\text{alt}}_{k-1}(p)$, also insbesondere $E(T) \in \mathcal{G}^{\text{alt}}_{k-1}$. Somit liefert Lemma 2(a) $E(T) \in \mathsf{Ma}^{\text{alt}}_{k-1}(q)$, was nach Definition von HasGh
$$E(T) \in \mathsf{HasGh}^{\text{alt}}_{k-1}(q)$$
bedeutet, E(T) ist also auf Prozessor q als überlappter Master gespeichert. Da die Master und Ghostelemente eines lokalen Gitterlevels stets eine Partition bilden (beachte Beweis zu Satz 4.8), ist sichergestellt, dass $E(T)$ auf Prozessor q nicht noch zusätzlich als Ghost gespeichert ist. Da $T \in \mathsf{Ma}^{\text{neu}}_{k}(q)$ nur in *VerfeinereGitter* (3) neu entstanden sein kann, diese Anweisung aber nur für $E(T) \notin \mathsf{HasGh}^{\text{alt}}_{k-1}(q)$ ausgeführt wird, ergibt sich ein Widerspruch. Fall 3 kann also nicht auftreten.

zu (M3'): Sei $T \in \mathcal{G}^{\text{neu}}_{k}$. Ist T unverfeinert, so ist nichts zu zeigen. Sei also o.B.d.A. $\mathcal{K}^{\text{neu}}(T) \neq \emptyset$ und $T' \in \mathcal{K}^{\text{neu}}(T)$ mit $T' \in \mathsf{Ma}^{\text{neu}}_{k+1}(p)$.

1. Fall: $T' \in \mathcal{G}^{\text{alt}}_{k+1}$, d.h. T' war schon in $\tilde{\mathcal{M}}^{\text{alt}}$ vorhanden. Dann war aber auch der Elterntetraeder T bereits in $\tilde{\mathcal{M}}^{\text{alt}}$ vorhanden, und es gilt $T' \in \mathcal{K}^{\text{alt}}(T)$. Sei q der Prozessor, auf welchem T' als Master in $\tilde{\mathcal{M}}^{\text{alt}}$ gespeichert war (wegen (M1) existiert ein solches q):
$$T' \in \mathsf{Ma}^{\text{alt}}_{k+1}(q) \ .$$
Lemma 2(a) impliziert $T' \in \mathsf{Ma}^{\text{neu}}_{k+1}(q)$, und wegen (M2') ist $q = p$. Andererseits liefert $T' \in \mathsf{Ma}^{\text{alt}}_{k+1}(p)$ zusammen mit (M3)
$$T \in \mathcal{G}^{\text{alt}}_{k}(p) \ .$$
Da T in $\tilde{\mathcal{M}}^{\text{neu}}$ verfeinert ist, ergibt sich mit Lemma 2(b) $T \in \mathcal{G}^{\text{neu}}_{k}(p)$.

2. Fall: $T' \notin \mathcal{G}_k^{\text{alt}}$, d.h. T' ist neu hinzugekommen. Nach Lemma 1(a) gilt

$$\exists\, q \quad T' \in \mathsf{Ma}_{k+1}^{\text{neu}}(q)\ ,$$

wegen der bereits bewiesenen Eindeutigkeit (M2') ist aber $q = p$. Mit Lemma 1(b) folgt weiterhin $T = E(T') \in \mathcal{G}_k^{\text{neu}}(p)$.

zu (M4'): Sei $T \in \mathsf{Gh}_k^{\text{neu}}(p)$. Lemma 3 impliziert dann

$$T \in \mathsf{Gh}_k^{\text{alt}}(p)\ .$$

Angenommen, $\mathcal{K}^{\text{neu}}(T) = \emptyset$, d.h. T ist in $\tilde{\mathcal{M}}^{\text{neu}}$ unverfeinert. Dann besitzt T nach Durchführung des Verfeinerungsalgorithmus die Markierung

$$\mathsf{mark}^{\text{neu}}(T) = \text{NoRef}\ .$$

Dann wäre T als Ghost auf Prozessor p aber in *VerfeinereGitter* (6) weggefallen, also $T \notin \mathsf{Gh}_k^{\text{neu}}(p)$. Dies ist aber ein Widerspruch, somit gilt doch $\mathcal{K}^{\text{neu}}(T) \neq \emptyset$.

zu (M5'): Sei $T \in \mathsf{Gh}_k^{\text{neu}}(p)$. Lemma 3 impliziert wiederum $T \in \mathsf{Gh}_k^{\text{alt}}(p)$. Aus (M4) folgt $\mathcal{K}^{\text{alt}}(T) \neq \emptyset$. Wegen (M4') ist T auch in $\tilde{\mathcal{M}}^{\text{neu}}$ verfeinert, also $\mathcal{K}^{\text{neu}}(T) \neq \emptyset$. Sei

$$T' \in \mathcal{K}^{\text{neu}}(T) \subset \mathcal{G}_{k+1}^{\text{neu}}$$

eines der Kinder von T in $\tilde{\mathcal{M}}^{\text{neu}}$.

1. Fall: $T' \in \mathcal{K}^{\text{alt}}(T)$, d.h. T' war bereits unter den alten Kindern von T. Wegen $T \in \mathsf{Gh}_k^{\text{alt}}(p)$ impliziert (M5)

$$\mathcal{K}^{\text{alt}}(T) \subset \mathsf{Ma}_{k+1}^{\text{alt}}(p)\ .$$

Also ist $T' \in \mathsf{Ma}_{k+1}^{\text{alt}}(p)$, und Lemma 2(a) liefert $T' \in \mathsf{Ma}_{k+1}^{\text{neu}}(p)$.

2. Fall: $T' \notin \mathcal{K}^{\text{alt}}(T)$. Dann ist T' in *VerfeinereGitter* (3) entstanden. Wegen

$$E(T') = T \in \mathsf{Gh}_k^{\text{alt}}(p)$$

ist dies auf Prozessor p geschehen. Also folgt $T' \in \mathsf{Ma}_{k+1}^{\text{neu}}(p)$.

□

Kapitel 5

Lastverteilung und Graphpartitionierung

5.1 Die Notwendigkeit der Lastverteilung

Um ein gegebenes Problem effizient parallel lösen zu können, muss gewährleistet werden, dass die Rechenlast gleichmäßig auf die einzelnen Prozessoren verteilt wird. Der Ansatz zur Erreichung dieses Ziels wird *Lastverteilung* genannt und besteht aus zwei Schritten: der *Lastbalancierung*, welche eine Zuordnung der Einzellasten zu den verschiedenen Prozessoren berechnet (diese Arbeit wird oft durch ein Lastbalancierungs-Tool übernommen) und der anschließenden *Lastmigration*, die diese Zuordnung auch tatsächlich umsetzt, wobei die entsprechenden Daten zwischen den Prozessoren ausgetauscht werden. Auch in DROPS wird ein solcher Lastbalancierer, nämlich ParMETIS, eingesetzt, die Lastmigration wird mit Hilfe des Transfer-Moduls von DDD abgewickelt.

Die Folgen eines Lastungleichgewichtes liegen auf der Hand: Zum einen führt dies zu einer verlängerten Gesamtrechenzeit, da der Prozessor mit dem größten Lastanteil am längsten mit der Berechnung, z.B. eines lokalen Skalarproduktes, beschäftigt ist, während die anderen Prozessoren schon mit der Berechnung fertig sind und am nächsten Synchronisationspunkt, bei diesem Beispiel der globalen Aufsummierung der lokalen Skalarprodukte, auf den langsamsten Prozessor warten, ohne weitere Berechnungen durchführen zu können. Dies wirkt sich ungünstig auf die Effizienz des betrachteten Programmabschnittes aus.

Zum anderen bedeutet unbalancierte Last aber auch eine ungleichmäßige Nutzung der lokalen Arbeitsspeicher, was eine stark verringerte maximale Problemgröße zur Folge hat, die auf einer bestimmten Maschine noch behandelt werden kann. Diese liegt dann weit unter der theoretisch erreichbaren Problemgröße, welche sich aus dem Gesamtspeicherplatz der Maschine ergibt; der brach liegende Speicherplatz der Prozessoren mit kleinem Lastanteil wird quasi verschenkt.

Es sei zusätzlich angemerkt, dass die Berechnung einer Lastbalancierung schwieriger wird, wenn diese für ein heterogenes Rechnernetzwerk gelten soll, also z.B. für einen Cluster, dessen Einzelrechner verschieden schnelle Prozessoren und verschieden große Arbeitsspeicher besitzen und evtl. über verschieden schnelle Netzwerkanbindungen verbunden sind. Deswegen sind die folgenden Ausführungen auf homogene Netzwerke beschränkt, für die die Berechnung einer Lastbalancierung bereits alles andere als trivial ist.

Die Frage der Lastverteilung bei einem Finite-Elemente-Verfahren ist die Frage nach der Verteilung der Elemente (in 3D also der Tetraeder) auf die Prozessoren, also einer Abbildung

$$m : \mathcal{T} \to \{1, \ldots, P\} \ .$$

m steht dabei für *master*, $m(T)$ gibt also den Prozessor an, auf dem der Tetraeder $T \in \mathcal{T}$ als Master gespeichert ist.

Auch wenn die Anfangstriangulierung $\mathcal{T}_0$ gleichmäßig verteilt ist, entsteht durch adaptive Verfeinerung nach wenigen Verfeinerungsschritten ein Lastungleichgewicht, welches erneute Lastverteilung erfordert. Diese Art der wiederholten zwischenzeitlichen Lastverteilung wird *dynamisch* genannt, im Gegensatz zur *statischen*, welche nur einmal zu Anfang vorgenommen wird und nur im Fall uniformer Verfeinerung sinnvoll ist.

Ein weiterer wichtiger Punkt besteht darin, dass die Berechnung der dynamischen Lastbalancierung selber parallel erfolgen muss, da serielle Lastbalancierung zu einem Flaschenhals des gesamten parallelen Verfahrens würde, und zwar nicht nur was die dafür benötigte Rechenzeit, sondern auch den dazu benötigten Speicherplatz angeht. Somit ist klar, dass für die parallele numerische Lösung partieller Differentialgleichungen mit Hilfe adaptiver Methoden eine *dynamische* und *parallele* Lastbalancierung erforderlich ist.

5.2 Das Graphpartitionierungsproblem

Der Lastbalancierungsalgorithmus muss nicht nur für eine gleichmäßige Datenverteilung sorgen (was mehr oder weniger trivial wäre), sondern dabei gleichzeitig die aus der Verteilung resultierende Überlappung der Daten minimieren (um den Kommunikationsaufwand möglichst gering zu halten). Um dies mathematisch zu beschreiben, wird zu einer Triangulierung $\mathcal{T}$ der sogenannte *duale Graph* $G(\mathcal{T}) = (V, E)$ definiert durch $V = \mathcal{T}$ sowie

$$(T_1, T_2) \in E \subset \mathcal{T} \times \mathcal{T} \quad :\Leftrightarrow \quad \text{die Tetraeder } T_1 \text{ und } T_2 \text{ besitzen eine gemeinsame Seitenfläche.}$$

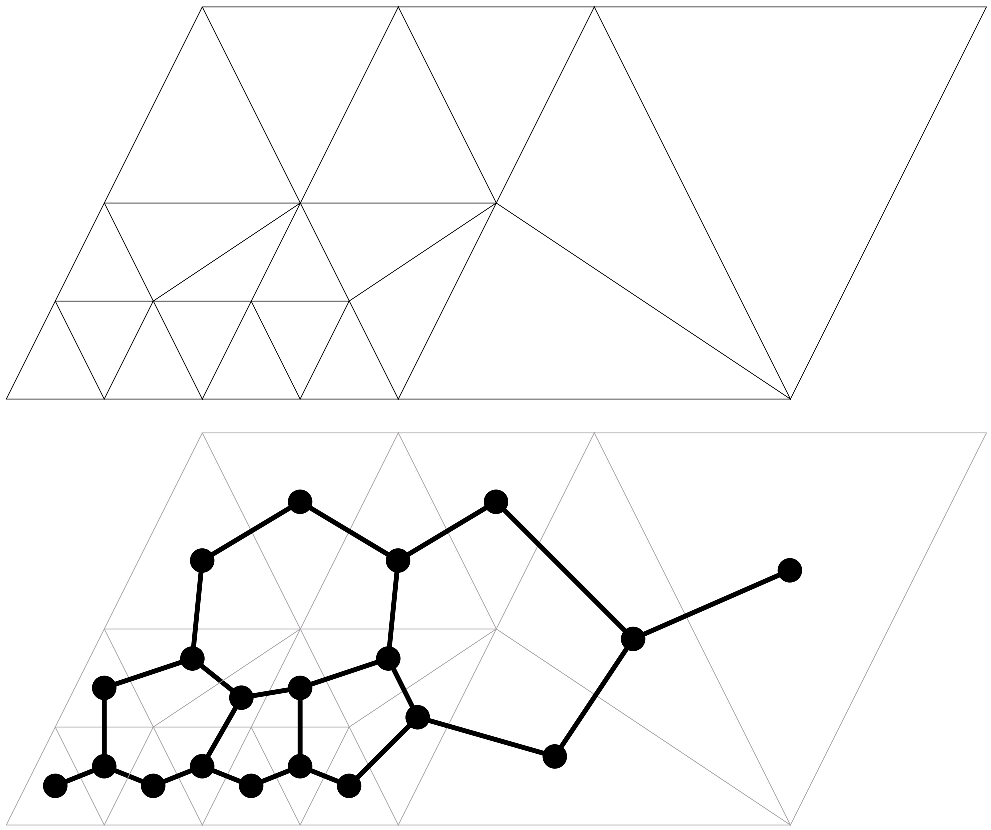


Abbildung 5.1: Triangulierung $\mathcal{T}$ und zugehöriger dualer Graph $G(\mathcal{T})$.

$G(\mathcal{T})$ ist offensichtlich ein ungerichteter Graph. Ein Beispiel für einen dualen Graphen $G(\mathcal{T})$ einer Triangulierung $\mathcal{T}$ ist in Abbildung 5.1 gegeben.

Zusätzlich werden Gewichtungsfunktionen $\alpha : V \to \mathbb{R}_+$ und $\beta : E \to \mathbb{R}_+$ für die Knoten und Kanten des Graphen eingeführt, um den Lastanteil $\alpha(v_T)$ jedes Tetraeders T (d.h. Knoten v_T des dualen Graphen) sowie den Kommunikationsumfang $\beta(e_F)$ über jede Seitenfläche F (d.h. Kante e_F des dualen Graphen) zu beschreiben. Insgesamt erhält man auf diese Weise einen *gewichteten Graphen* $G = (V, E, \alpha, \beta)$. Das Gewicht einer Knotenteilmenge $\tilde{V} \subset V$ wird definiert als

$$\alpha(\tilde{V}) := \sum_{T \in V} \alpha(T)$$

und spiegelt die gesammelte Last der Tetraeder aus $\tilde{V}$ wider. Für eine gegebene Partitionierung $m : V \to \{1, \ldots, P\}$ entspricht die Menge

$$E_{\mathrm{cut}}(m) := \{(T_1, T_2) : m(T_1) \neq m(T_2)\}$$

denjenigen Seitenflächen, welche bei dieser Verteilung überlappend gespeichert werden und so die Prozessorränder bilden, über welche die Kommunikation zwischen den Prozessoren stattfindet.

Nun kann das Lastbalancierungsproblem als ein Graphpartitionierungsproblem formuliert werden, wobei der Parameter C den Grad der noch tolerierten Unbalancierung steuert:

Definition 5.1 (Verallgemeinertes Graphpartitionierungsproblem) Seien $P \in \mathbb{N}$, $C \in \mathbb{R}$, $C \geq 1$ und ein ungerichteter gewichteter Graph $G = (V, E, \alpha, \beta)$ gegeben. Finde eine Abbildung $m : V \to \{1, \dots, P\}$, so dass gilt

$$\mathrm{cost}_{\mathrm{comm}}(m) := \sum_{e \in E_{\mathrm{cut}}(m)} \beta(e) \to \min \ ,$$

unter der Nebenbedingung

$$\alpha(V_p) \leq C \frac{\alpha(V)}{P}$$

mit $V_p := m^{-1}(p) = \{T \in V : m(T) = p\}$.

Das Graphpartitionierungsproblem gehört zur Klasse der NP-vollständigen Probleme, somit gilt selbiges auch für das dahintersteckende Lastbalancierungsproblem. Die Berechnung einer *optimalen* Partitionierung ist also nicht effizient durchführbar. Dagegen gibt es zahlreiche heuristische Ansätze, die teilweise relativ brauchbare Ergebnisse liefern. Einem Überblick über verschiedene Verfahren der Graphpartitionierung soll der folgende Abschnitt dienen.

5.3 Überblick über verschiedene Methoden

Dieser Abschnitt, der auf einem Übersichtsartikel zur Graphpartitionierung [5] basiert, soll einen Einblick in verschiedene Methoden zur Graphpartitionierung bieten und einige der vielen unterschiedlichen Ansätze vorstellen. Auf eine detaillierte Darstellung wird allerdings verzichtet, weil dies den Rahmen dieser Arbeit sprengen würde.

Die Verfahren können je nach ihren prinzipiellen Grundideen in drei Klassen eingeteilt werden:

- Verfahren, die die geometrischen Eigenschaften eines Graphen ausnutzen,
- Verfahren, die auf strukturellen Eigenschaften eines Graphen basieren,
- Verfahren, die die Idee der Umverteilung verwenden, um eine bestehende Partition zu verbessern.

Auch auf dem Gebiet der Graphpartitionierung wird die Idee des Multilevel-Konzepts seit neuerer Zeit erfolgreich angewendet. Dies soll abschließend näher erläutert werden.

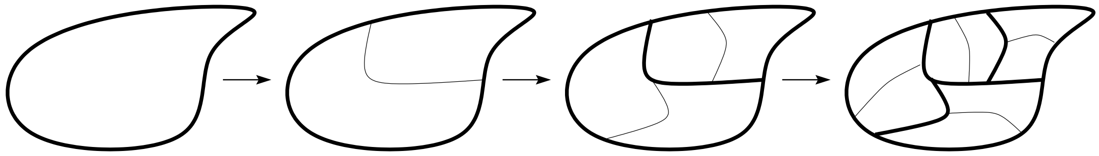


Abbildung 5.2: Schematische Darstellung der rekursiven Bisektion.

5.3.1 Rekursive Bisektion

Einen wichtigen Spezialfall der Graphpartitionierung stellt die Graphbisektion dar, d.h. die Unterteilung des Graphen in zwei Teile. Auch das Graphbisektionsproblem ist NP-vollständig, denn das Graphpartitionierungsproblem lässt sich durch rekursive Anwendung von Bisektionsschritten darauf zurückführen.

Dieser Divide-and-Conquer-Ansatz wird *rekursive Bisektion* genannt und von den meisten Algorithmen zur Bestimmung einer P-Partitionierung verwendet. Dabei wird der Ausgangsgraph mittels Bisektion in zwei Teilgraphen zerlegt, welche wiederum durch Bisektion geteilt werden usw. Nach r Rekursionsschritten ist eine 2^r-Partitionierung entstanden. Abb. 5.2 stellt den Ablauf der rekursiven Bisektion schematisch dar.

5.3.2 Geometriebasierte Verfahren

Geometrische Verfahren nutzen zusätzliche geometrische Informationen über einen Graphen aus, die in Form von Koordinaten $x_v \in \mathbb{R}^d$ der einzelnen Graphknoten $v \in V$ gegeben seien. Diese Koordinaten stammen aus der zugrunde liegenden Problemstellung; im Falle finiter Elemente, wo die Graphknoten den einzelnen Tetraedern entsprechen, sind die Koordinaten gerade durch den Schwerpunkt des jeweiligen Tetraeders gegeben. Dabei wird angenommen, dass die räumliche Nähe zweier Knoten sich auch in einem kurzen Verbindungpfad im Graphen selbst ausdrückt. In der Tat benutzen geometrische Algorithmen alleine die geometrischen Informationen und nicht die Konnektivität des Graphen, so dass die Qualität der erzeugten Partitionierung stark vom Bestehen dieser Annahme abhängt.

Da natürlich nicht jeder Graph über Knotenkoordinaten verfügt, sind die geometrischen Verfahren nicht allgemein anwendbar. Weil aber sehr viele zu partitionierende Graphen einer großen Problemklasse entstammen, bei der in natürlicher Weise den Graphknoten räumliche Koordinaten zugeordnet werden können, und diese Verfahren zudem relativ einfach zu implementieren sind, spielen sie in der Praxis eine wichtige Rolle.

Rekursive Koordinatenbisektion

Als ein Vertreter der geometrischen Verfahren soll die *rekursive Koordinatenbisektion (RCB)* vorgestellt werden, die auch als *rekursive orthogonale Bisektion* bekannt ist. Sie basiert auf dem soeben erläuterten Prinzip der rekursiven Bisektion.

Jeder Bisektionsschritt beginnt mit der Sortierung der Knoten nach ihren Raumkoordinaten für jede Raumdimension. Dadurch entstehen d Anordnungen der Knoten, für die jeweils der Median M_j, $j = 1, \ldots, d$, der Koordinatenwerte berechnet wird. Für jede Dimension j definiert dies eine Hyperebene H_j senkrecht zur jeweiligen Koordinatenachse, welche diese im Median M_j schneidet:

$$H_j: \; x_j = M_j \qquad j = 1, \ldots, d$$

Nach Konstruktion teilt jede dieser Hyperebenen den Graphen in zwei Teile (Knoten, die auf der Hyperebene liegen, werden jeweils zur Hälfte einem der beiden Teilgraphen zugeordnet). Der Algorithmus wählt nun diejenige Hyperebene aus, die die wenigsten Kanten des Graphen schneidet.

Diese recht simple Methode besitzt viele Varianten. Eine wichtige davon, die bessere Ergebnisse als die RCB liefert, ist die *recursive inertial bisection*, welche die Hauptträgheitsachse I der Knoten berechnet und eine Hyperebene senkrecht dazu auswählt. Die Idee dabei besteht darin, dass viele Knoten längs dieser Achse und in großer Nähe dazu liegen, so dass zu I senkrechte Hyperebenen relativ wenige Kanten des Graphen schneiden.

5.3.3 Strukturbasierte Verfahren

Im Gegensatz zu geometrischen Verfahren können strukturelle Verfahren auf allgemeine Graphen angewendet werden, weil sie alleine die Konnektivität der Knoten und damit die Struktur des Graphen verwenden.

Zu dieser Klasse von Verfahren zählen einige *Greedy*-Algorithmen (greedy, *engl.* gierig). Der Greedy-Algorithmus von Farhat beispielsweise teilt den Graph durch Breitensuche direkt in P Teilgraphen ein und ist somit auch ein Beispiel für einen Algorithmus, der nicht auf rekursiver Bisektion basiert. Von einem beliebigen Knoten aus wird eine Breitensuche gestartet, bis $\lceil |V|/P \rceil$ Knoten beisammen sind, welche die erste Partition V_1 bilden. Als Ausgangspunkt für die nächste Breitensuche wird einer derjenigen Randknoten der zuletzt gebildeten Partition V_i gewählt, welche die wenigsten noch nicht besuchten Kanten besitzen. Nach $P-1$ Breitensuchen ist schließlich eine P-Partitionierung entstanden. Die Breitensuche ersetzt gewissermaßen die den geometrischen Verfahren zur Verfügung stehende euklidische Entfernung der Knoten: Stattdessen wird die Entfernung zweier Knoten im Graphen herangezogen; diese ist als Länge des kürzesten Verbindungspfades in G definiert.

Der Greedy-Algorithmus von Karypis und Kumar basiert auf rekursiver Bisektion und benutzt ein ähnliches Prinzip wie der von Farhat zur Bildung der Partitionen. Allerdings wird zur Auswahl der hinzukomenden Knoten keine Breitensuche verwendet, stattdessen werden die Knoten nach ihrem Nutzen bewertet: Durch Hinzunahme eines Knotens ändert sich die Zahl $|E_{\mathrm{cut}}|$ der durchschnittenen Kanten; der Knoten mit der jeweils günstigsten Bilanz wird hinzugenommen.

Alle diese pfadorienterten Verfahren kranken an einem entscheidenden Nachteil: Die Struktur des Graphen wird jeweils nur lokal beurteilt, die globale Struktur fließt nur unzureichend ein.

Diesen Nachteil beheben *spektrale Verfahren*, indem sie die globale Konnektivität des Graphen mit in Betracht ziehen. Die in der Praxis oft benutzte *rekursive spektrale Bisektion (RSB)* soll im Folgenden erläutert werden.

Rekursive spektrale Bisektion

Wichtige Ausgangsgrundlage der RSB ist die sogenannte Laplace-Matrix $L(G) = L \in \mathbb{R}^{|V|\times|V|}$ eines Graphen G, deren Einträge folgendermaßen definiert sind:

$$L_{ij} = \begin{cases} \deg(v_i) & i = j \\ -1 & (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$

$\deg(v_i)$ bezeichnet dabei die Anzahl der in v_i zusammentreffenden Kanten und wird als *Knotengrad* bezeichnet.

In $L(G)$ sind nun die Informationen über die Konnektivität von G enthalten, so dass sich das Graphbisektionsproblem mittels L beschreiben lässt. Dazu betrachte einen Vektor $\underline{x}$ mit $|V|$ Einträgen, die jeweils den Wert 1 oder -1 besitzen und bestimmen, ob der jeweilige Knoten in V_1 oder V_2 liegt. $\underline{x}$ heißt *Bisektionsvektor*.

Der Einfachheit halber sei $|V|$ gerade. Dann ist Gleichverteilung genau für $\sum_i \underline{x}_i = 0$ gegeben. Man macht sich leicht klar, dass

$$\underline{x}^T L \underline{x} = 4|E_{\mathrm{cut}}|$$

gilt, also die Güte der Bisektion beschreibt. Das Graphbisektionsproblem lässt sich jetzt äquivalent so formulieren:

> Finde einen Bisektionsvektor $\underline{x}$, so dass $\sum_i \underline{x}_i = 0$ gilt und $\underline{x}^T L \underline{x}$ minimal ist.

Selbstverständlich ist dieses äquivalente Problem ebenfalls NP-vollständig. Erweitert man dieses diskrete Problem zu einem kontinuierlichen, indem man Werte $\underline{x}_i \in \mathbb{R}, \quad i = 1, \ldots, |V|$, zulässt, so wird daraus ein mit Hilfe von Eigenwertmethoden behandelbares Problem:

$$\text{Finde } \underline{x} \in \mathbb{R}^{|V|}, \text{ so dass } \textstyle\sum_i \underline{x}_i = 0 \text{ gilt und } \left|\frac{\underline{x}^T L \underline{x}}{\underline{x}^T \underline{x}}\right| \text{ minimal ist.} \tag{5.1}$$

Aufgrund ihrer Symmetrie besitzt die Laplace-Matrix L ein reelles Spektrum und eine orthogonale Eigenvektorbasis. Mithilfe der Gershgorin-Kreise lässt sich das Spektrum von L weiter eingrenzen: Wegen der Konstruktion von L gilt gerade $L_{ii} = \sum_{j \neq i} |L_{ij}| = \deg(v_i)$, also gilt für das Spektrum

$$\sigma(L) \;\subset\; \bigcup_{v \in V} [0,\, 2\deg(v)] \;\subset\; [0,\, 2\max_{v \in V} \deg(v)] \;.$$

L ist jedoch nicht positiv definit, denn $0 \in \sigma(L)$: der Vektor $\underline{x}^{(1)}$, der aus lauter Einsen besteht, ist ein Eigenvektor von L zum Eigenwert $\lambda_1 = 0$. (Aus Sicht der Graphpartitionierung interpretiert, entspricht $\underline{x}^{(1)}$ einer „Partitionierung" von G in $V_1 = V$ und $V_2 = \emptyset$.) Der Eigenvektor zum zweitgrößten Eigenwert von L dagegen stellt den Schlüssel zur Lösung des Problems dar:

Satz 5.2 Sei $\underline{x}^{(1)}, \underline{x}^{(2)}, \ldots, \underline{x}^{(|V|)}$ eine (wegen der Symmetrie von L existierende) Eigenvektorbasis zu den reellen Eigenwerten

$$0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_{|V|}$$

von L, wobei die Normierung $||x^{(i)}||_2 = \sqrt{|V|}$ gelte. Dann ist $\underline{x}^{(2)}$, also der Eigenvektor zum zweitkleinsten Eigenwert λ_2 von L, eine Lösung des kontinuierlichen Problems (5.1).

Beweis: Wegen der Orthogonalität der Eigenvektorbasis gilt insbesondere

$$\sum_i \underline{x}_i^{(2)} = {\underline{x}^{(1)}}^T \underline{x}^{(2)} = 0 \;.$$

Also erfüllt $\underline{x}^{(2)}$ die erste Bedingung von (5.1).

Zum Beweis der zweiten Bedingung (Minimalität) sei $\underline{y} \in \mathbb{R}^{|V|}$ ein beliebiger Vektor mit $\sum_i \underline{y}_i = 0$, seine Darstellung in der Eigenvektorbasis laute $\underline{y} = \sum_i a_i \underline{x}^{(i)}$. Wie oben gilt wiederum ${\underline{x}^{(1)}}^T \underline{y} = \sum_i \underline{y}_i = 0$, also liegt $\underline{y}$ im Orthogonalraum zu $\underline{x}^{(1)}$, somit ist $a_1 = 0$.

Wegen der Orthogonalität der Eigenvektorbasis gilt nun

$$\underline{y}^T \underline{y} = \sum_{i=\mathbf{2}}^{|V|} a_i^2 ||\underline{x}^{(i)}||_2^2 \;,$$

sowie

$$\begin{aligned} \underline{y}^T L \underline{y} &= y^T \sum_{i=2}^{|V|} a_i \lambda_i \underline{x}^{(i)} \\ &= \sum_{i=2}^{|V|} a_i^2 \lambda_i ||\underline{x}^{(i)}||_2^2 \ . \end{aligned}$$

Es folgt also wegen $\lambda_i \geq \lambda_2$ für $i \geq 2$

$$\underline{y}^T L \underline{y} \ \geq \ \lambda_2 \, \underline{y}^T \underline{y} \ = \ \underline{y}^T \underline{y} \, \frac{\underline{x}^{(2)T} L \underline{x}^{(2)}}{\underline{x}^{(2)T} \underline{x}^{(2)}}$$

und damit die Behauptung. □

Bemerkung 5.3 $\underline{x}^{(2)}$ wird auch als *Fiedler-Vektor* von G bezeichnet.

Trotz dieses schönen Resultates stellt sich natürlich augenblicklich die Frage, ob die Lösung des kontinuierlichen Problems überhaupt Relevanz für das diskrete Problem hat. Hier zeigt sich, dass die Lösung des kontinuierlichen Problems mit dem diskreten Ausgangsproblem in direkter Beziehung steht: Fiedler analysierte den Eigenvektor $\underline{x}^{(2)}$ der Laplace-Matrix $L(G)$ und zeigte, dass die Differenz zweier Einträge von $\underline{x}^{(2)}$ umso kleiner ist, je kürzer der kleinste verbindende Pfad in G zwischen den beiden entsprechenden Knoten ist. Es ist somit sinnvoll, den Fiedler-Vektor zur Berechnung einer Graphbisektion heranzuziehen.

Somit gestaltet sich ein spektraler Bisektionsschritt wie folgt: Berechne den Fiedler-Vektor $\underline{x}^{(2)}$ mit einem (speziell modifizierten) Lanczos-Verfahren und bestimme den Median M seiner Einträge. Knoten v_i mit $\underline{x}_i^{(2)} < M$ gehören zu V_1, solche mit $\underline{x}_i^{(2)} > M$ zu V_2 (solche mit $\underline{x}_i^{(2)} = M$ werden gleichmäßig aufgeteilt). Beachte, dass L nicht explizit aufgestellt werden muss, sondern eine Multiplikation mit L mithilfe der Struktur von G berechnet werden kann.

RSB erzeugt qualitativ hochwertige Partitionierungen, weil die globale Struktur des Graphen berücksichtigt wird, benötigt aber wegen der Berechnung des Fiedler-Vektors auch entsprechend lange Rechenzeiten. Im Falle paralleler Graphpartitionierung lässt sich diese Berechnung allerdings gut parallelisieren und die Rechenzeit entsprechend mindern.

5.3.4 Umverteilungsbasierte Verfahren (Partitionsverfeinerung)

Im Gegensatz zu den beiden vorher erläuterten Klassen der geometrischen und strukturellen Verfahren, die einen Graphen von Grund auf neu partitionieren, setzen umverteilungsbasierte Verfahren auf einer bestehenden Partitionierung auf und versuchen, diese durch Umverteilung von Knoten zu verbessern. Dies wird auch als *Verfeinerung* einer Partition bezeichnet, deswegen tragen diese Verfahren im Englischen den Namen *(graph) refinement methods*.

Obwohl umverteilungsbasierte Verfahren prinzipiell auch auf eine zufällig erzeugte Partition angewendet werden können, arbeiten sie am besten, wenn die Eingabepartition bereits eine gewisse Qualität besitzt. Deswegen werden sie oft als Postprocessing-Schritt nach einer erfolgten Partitionierung eingesetzt, um diese weiter zu verbessern, und sind bedeutend im Rahmen der dynamischen Lastbalancierung, wenn z.B. im Fall adaptiver Gitter die Last von Zeit zu Zeit neu verteilt wird und sich der zugehörige duale Graph nur lokal verändert hat.

Den prominentesten Vertreter dieser Klasse stellt das bereits aus den 60er-Jahren stammende *Kernighan-Lin-Verfahren (KL)* dar. Es handelt sich dabei um ein deterministisches Optimierungsverfahren mit der *hill-climbing*-Eigenschaft. Der Grundgedanke eines Bisektionsschrittes besteht darin, einzelne Knoten von der einen in die andere Partition zu verlagern mit dem Ziel, das Gewicht der durchschnittenen Kanten $\mathrm{cost}_{\mathrm{comm}}(m)$ zu verringern.

Bei der Auswahl des zu verschiebenden Knotens v spielt sein Gewinn (*gain*) $g(v)$ eine zentrale Rolle. Sei v o.B.d.A. in der ersten Partition enthalten, also $v \in V_1$. Dann ist $g(v)$ definiert als Differenz aus dem Gewicht der Kanten, die v mit der anderen Partition V_2 verbinden, und dem Gewicht der Kanten, durch die v mit der eigenen Partition V_1 verbunden ist:

$$g(v) := \sum_{e \in E \cap (\{v\} \times V_2)} \beta(e) - \sum_{e \in E \cap (\{v\} \times V_1)} \beta(e) \ .$$

Beachte, dass $g(v)$ auch negative Werte annehmen kann. Ein Greedy-Algorithmus würde nun nacheinander die Knoten mit dem jeweils maximalen Gewinn verschieben, und zwar solange, bis kein Knoten mit positivem Gewinn mehr vorhanden wäre. Der Algorithmus hat dann ein lokales Minimum von $\mathrm{cost}_{\mathrm{comm}}$ gefunden, ist aber auch in diesem gefangen.

Um dies zu vermeiden, wird im KL-Algorithmus auch das Verschieben von Knoten mit negativem Gewinn zugelassen, wenn keine Knoten mit positivem Gewinn mehr existieren. Damit wird also ein Anstieg von $\mathrm{cost}_{\mathrm{comm}}$ (*hill climbing*) akzeptiert, in der Hoffnung, dass dadurch im Endeffekt bessere Minima gefunden werden.

Jeder Knoten darf nur einmal verschoben werden, um zu gewährleisten, dass der Algorithmus nach endlicher Zeit abbricht. Nachdem alle Knoten verschoben worden sind (oder eine bestimmte Anzahl von verschobenen Knoten mit negativem Gewinn erreicht worden ist), werden die im Verlauf des Knotenverschiebens gebildeten Partitionierungen $m_0, m_1, \ldots, m_K$ nach ihrer Qualität beurteilt. Diejenige Partitionierung m_k, $0 \leq k \leq K$, welche $\text{cost}_{\text{comm}}$ minimiert, stellt dann die Ausgabe des Algorithmus dar.

In gewisser Weise erinnert das KL-Verahren an die Methode des *simulated annealing*, welches sich in vielen Bereichen der Optimierung als erfolgreiches Verfahren erwiesen hat. Im Bereich der Graphpartitionierung aber liefert es nur unzureichende Resultate.

5.3.5 Multilevel-Graphpartitionierung

Auch im Bereich der Graphpartitionierung hat die Multilevel-Idee in den letzten Jahren gewinnbringenden Einzug gehalten. Dabei hat sich in der Praxis gezeigt, dass dieser erfolgsversprechende Ansatz nicht nur die Erzeugung von Partitionen beschleunigt, sondern oft auch deren Qualität im Vergleich zu den entsprechenden One-Level-Algorithmen verbessert.

Analog zu Mehrgitterverfahren im Bereich der numerischen Lösung partieller Differentialgleichungen besteht die Multilevel-Graphpartitionierung aus der Idee, durch Vergröberung des Ausgangsproblems eine Hierarchie von stufenweise verkleinerten Graphpartitionierungsproblemen zu erzeugen, die eine Approximation des ursprünglichen Problems darstellen. Das gröbste Graphpartitionierungsproblem wird mit einem Standardalgorithmus (z.B. RSB) gelöst, was wegen der reduzierten Problemgröße nur relativ wenig Rechenzeit benötigt. Die erhaltene Grobgraphpartition wird nun stufenweise auf die feineren Graphen übertragen, bis schließlich eine Partition des Ausgangsgraphen gefunden ist. Der schematische Ablauf eines Multilevel-Graphpartitionierungsalgorithmus ist in Abb. 5.3 dargestellt.

Die grundlegenden Bausteine eines Multilevel-Graphpartitionierungsalgorithmus sind also folgende:

1. **Vergröberung** eines Graphen, wobei sich die globale Struktur des Ausgangsgraphen in dem vergröberten Graphen in gewissem Sinne niederschlagen soll.

2. **Partitionierung** des gröbsten Graphen mit einem Standardverfahren.

3. **Interpolation** einer Grobgraphpartition auf den nächstfeineren Graphen. Das Vorgehen bei der Interpolation hängt dabei direkt von den zur Vergröberung und Partitionierung gewählten Methoden ab. Der Interpolation folgt oft ein Verfeinerungsschritt, um die interpolierte Partition weiter zu verbessern und an die feinere Struktur des Graphen anzupassen.

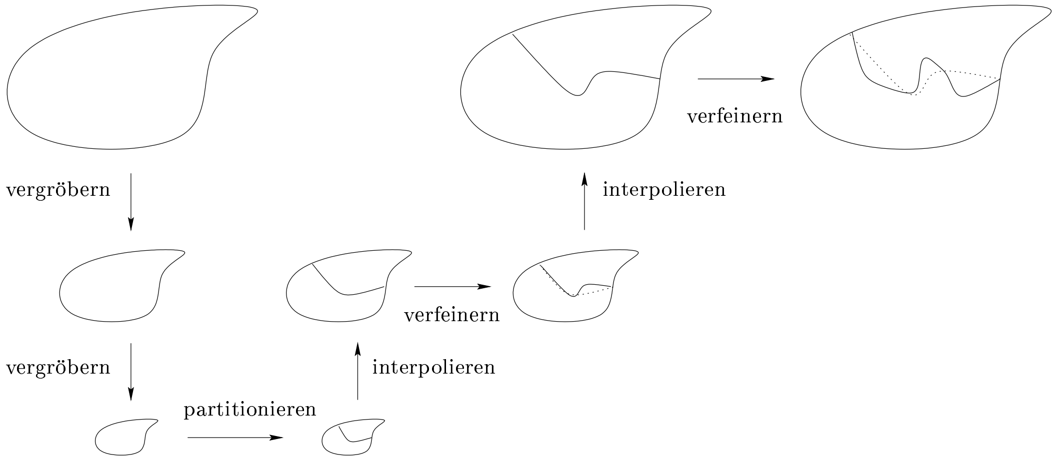


Abbildung 5.3: Schema einer Multilevel-Graphpartitionierung.

Für die Komponente der Partitionierung kann z.B. einer der vorgestellten Algorithmen aus 5.3.2 und 5.3.3 verwendet werden.

Als eine Möglichkeit der Graphvergröberung sei hier das sogenannte *Graphmatching* vorgestellt.

Definition 5.4 (Matching) Ein *Matching* eines ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge $E' \subset E$ ungerichteter Kanten, so dass zwei verschiedene ungerichtete Kanten aus E' keinen gemeinsamen Knoten besitzen:

$$\forall\,(v_1, v_2) \in E' : \quad (v_2, v_1) \in E' \tag{5.2}$$

$$\forall\, e = (v_1, v_2),\, e' = (v_1', v_2') \in E' : \quad \{v_1, v_2\} \cap \{v_1', v_2'\} \neq \emptyset \Rightarrow e = e' \tag{5.3}$$

Ein *maximales Matching* von G ist ein Matching $E'_{\max}$, welches bei Hinzufügen einer weiteren Kante aus $E \setminus E'_{\max}$ die Matching-Eigenschaft (5.3) verliert.

Ein maximales Matching von G kann durch einen einfachen Greedy-Algorithmus in $\mathcal{O}(|E|)$-Laufzeit berechnet werden. Mit Hilfe eines maximalen Matchings $E'_{\max}$ von G kann nun ein vergröberter Graph $\tilde{G} = (\tilde{V}, \tilde{E})$ konstruiert werden, indem jeweils zwei Knoten einer Kante aus $E'_{\max}$ zu einem sogenannten *Multiknoten* zusammengefasst werden; alle Knoten, die mit keiner Kante aus $E'_{\max}$ verbunden sind, werden einfach übernommen:

$$\begin{aligned}
\tilde{V}_M &= \big\{\{v_1, v_2\} : (v_1, v_2) \in E'_{\max}\big\}\ , \\
\tilde{V}_S &= \Big\{\{v\} : v \in V, v \notin \bigcup_{\tilde{v} \in \tilde{V}_M} \tilde{v}\Big\}\ , \\
\tilde{V} &= \tilde{V}_M \cup \tilde{V}_S \subset Pot(V)\ , \\
\tilde{E} &= \{(\tilde{v}_1, \tilde{v}_2) : \quad \exists\, v_1 \in \tilde{v}_1, v_2 \in \tilde{v}_2 : (v_1, v_2) \in E\} \\
&\quad \setminus \{(\tilde{v}, \tilde{v}) : v \in V\}\ .
\end{aligned}$$

Ist G ein gewichteter Graph mit Gewichtung (α, β), so kann daraus leicht eine Gewichtung $(\tilde{\alpha}, \tilde{\beta})$ des vergröberten Graphen $\tilde{G}$ abgeleitet werden: Das Gewicht eines Multiknotens $\{v_1, v_2\} \in \tilde{V}_M$ ist gerade die Summe der Gewichte von v_1 und v_2, die Gewichte der übernommenen Knoten aus $\tilde{V}_S$ ändern sich nicht. Das Gewicht einer Kante $(\tilde{v}_1, \tilde{v}_2) \in \tilde{E}$ ist gerade die Summe der Gewichte aller Kanten $(v_1, v_2) \in E \setminus E'_{\max}$, die Knoten aus $\tilde{v}_1$ und $\tilde{v}_2$ miteinander verbinden:

$$\tilde{\beta}\big((\tilde{v}_1, \tilde{v}_2)\big) := \sum_{e \in (E \setminus E'_{\max}) \cap (\tilde{v}_1 \times \tilde{v}_2)} \beta(e)$$

Damit geht die Gewichtung von G in angemessener Weise auf $\tilde{G}$ über. Abb. 5.4 zeigt die Vergröberung eines gewichteten Graphen durch ein maximales Matching.

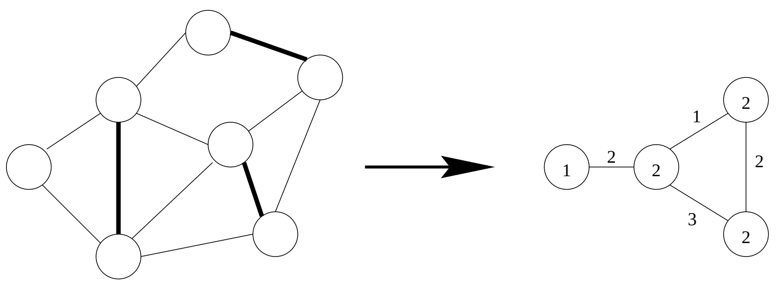


Abbildung 5.4: Vergröberung durch maximales Matching, induzierte Gewichtung: Die Kanten aus $E'_{\max}$ sind fett dargestellt. Alle Knoten und Kanten des Ausgangsgraphen seien mit 1 gewichtet; die Gewichte des vergröberten Graphen sind durch die Beschriftung der Knoten und Kanten ersichtlich.

Bemerkung 5.5 (Interpolation) Da die Zuordnung der Knoten $v \in V$ zu den Grobgraphknoten $\tilde{v} \in \tilde{V}$ eindeutig ist, ergibt sich aus einer Grobgraphpartitionierung $(\tilde{V}_1, \ldots, \tilde{V}_K)$ von $\tilde{G}$ direkt die im Interpolationsschritt gebildete Feingraphpartitionierung $(V_1 \ldots, V_K)$ in trivialer Weise:

$$V_k := \bigcup_{\tilde{v} \in \tilde{V}_k} \tilde{v} \ .$$

Die bestehenden Implementationen von Multilevel-Graphpartitionierungsverfahren unterscheiden sich jeweils in der Kombination der für die drei Grundbausteine gewählten Methoden. Neben der Multilevel-RSB sei der in ParMETIS implementierte und in DROPS eingesetzte Multilevel-Algorithmus von Karypis/Kumar erwähnt, der zur Vergröberung ein modifiziertes Graphmatching (*heavy edge matching*), den vorgestellten Greedy-Algorithmus zur Grobgraphpartitionierung sowie eine Variante des KL-Algorithmus in der Verfeinerungsphase verwendet.

5.3.6 Frei verfügbare Lastbalancierungstools

Im Internet gibt es einige Software-Bibliotheken zur Graphpartitionierung, die entweder frei oder zu Forschungszwecken heruntergeladen werden können. Zu den populärsten Tools zählen sicherlich METIS/ParMETIS[1] und JOSTLE[2]. Das METIS/ParMETIS-Paket wird von Karypis und Kumar an der University of Minnesota entwickelt und enthält sowohl serielle als auch parallele Graphpartitionierungsalgorithmen, darunter auch Multilevelalgorithmen. Inzwischen wird auch *multi constraint partitioning*, also Aufteilung unter mehreren Nebenbedingungen, unterstützt.

Auch JOSTLE, welches von Walshaw an der University of Greenwich entwickelt wird, bietet ähnlich wie METIS/ParMETIS serielle und parallele Graphpartionierung, darunter auch Multilevelalgorithmen; ein Schwerpunkt liegt darauf, die Anzahl der zu migrierenden Graphknoten möglichst gering zu halten. Beide Bibliotheken werden laufend gewartet und erweitert, gerade in diesem Jahr sind neue Versionen herausgekommen.

Weitere Tools sind ZOLTAN[3] (Sandia National Laboratories) und DRAMA[4] (NEC Europe), die neben eigenen Graphpartitionierungsroutinen auch die Funktionen der Bibliotheken ParMETIS und JOSTLE zur Lastverteilung einsetzen. Dies ist dann interessant, wenn man ParMETIS und JOSTLE gleichzeitig nutzen oder miteinander vergleichen möchte, ohne dafür den dualen Graphen zweimal (im jeweils geforderten Format) aufstellen zu müssen.

CHACO[5] (Hendrickson/Leland, Sandia National Laboratories), SCOTCH[6] (Pellegrini, Université Bordeaux I) und PARTY[7] (Preis, Universität Paderborn) sind Bibliotheken zur statischen Lastverteilung. CHACO und PARTY scheinen aktuell nicht mehr weiterentwickelt zu werden.

5.4 Lastverteilung in DROPS

Dieser Abschnitt widmet sich der Implementierung der Lastverteilung in DROPS. Im Rahmen dieser Arbeit wird ein CG-Verfahren eingesetzt, um das diskrete Problem auf der feinsten Triangulierung $\mathcal{T}_J$ zu lösen, so dass die gewählte Art der

[1] `http://www-users.cs.umn.edu/~karypis/metis/parmetis/`
[2] `http://www.gre.ac.uk/~c.walshaw/jostle/`
[3] `http://www.cs.sandia.gov/Zoltan/`
[4] `http://www.ccrl-nece.technopark.gmd.de/DRAMA/`
[5] `http://www.cs.sandia.gov/HPCCIT/chac.html`
[6] `http://www.labri.fr/Perso/~pelegrin/scotch/`
[7] `http://www.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/PART/party.html`

Lastverteilung hierauf abgestimmt ist. Dabei müssen die Tetraeder $T \in \mathcal{T}_J$ so verteilt werden, dass die Rechenlast für eine Matrix-Vektor-Multiplikation möglichst gleich verteilt ist und gleichzeitig der Kommunikationsaufwand für die Akkumulation möglichst gering ist.

Dazu ist eine gleichmäßige Verteilung der Tetraeder aus $\mathcal{T}$ anzustreben, wobei die Anzahl der Seitenflächen auf Prozessorrändern möglichst klein sein sollte. Dies legt folgender Satz nahe ($\tilde{\mathcal{T}}$ bezeichne dabei den Teil der Triangulierung, den ein bestimmter Prozessor speichert):

Satz 5.6 *Sei $\tilde{\mathcal{T}}$ eine Triangulierung eines einfach zusammenhängenden Gebietes $\tilde{\Omega} \subset \mathbb{R}^3$. $\tilde{\mathcal{T}}$ umfasse $n_T := |\tilde{\mathcal{T}}|$ Tetraeder, n_{F_B} gebe die Anzahl der Seitenflächen auf dem Rand $\partial\tilde{\Omega}$ an. Die dünnbesetzte Matrix A sei die Steifigkeitsmatrix, welche durch Diskretisierung einer partiellen Differentialgleichung mit konformen linearen Finiten Elementen auf $\tilde{\mathcal{T}}$ entsteht. Dann gilt für die Zahl der Gleitkommaoperationen für eine Matrix-Vektor-Multiplikation $y = Ax$:*

$$FP(Ax) \leq 16n_T + \frac{7}{2}n_{F_B} + 2 \ .$$

Beweis: Seien n_T die Anzahl der Tetraeder, n_F die Anzahl der Seitenflächen, n_E die Anzahl der Kanten und n_V die Anzahl der Knoten in $\tilde{\mathcal{T}}$. n_{F_B}, n_{E_B} und n_{V_B} bezeichne die Anzahl der entsprechenden Subsimplices auf dem Rand $\partial\tilde{\Omega}$. Dann gelten folgende zwei Gleichungen und die folgende Ungleichung:

$$\begin{aligned} n_T + n_E + 1 &= n_F + n_V && (1)\\ 2n_F - n_{F_B} &= 4n_T && (2)\\ 2n_E &\leq 3n_F && (3) \end{aligned}$$

(1) ist die Eulersche Formel für einfach zusammenhängende Gebiete in drei Dimensionen. Gleichung (2) ergibt sich aus der Beobachtung, dass jede innere Seitenfläche an zwei Tetraeder angrenzt, es sei denn, sie liegt auf dem Rand. Ungleichung (3) besagt, dass an jeder Kante mindestens zwei Seitenflächen angrenzen.

Der Graph der Steifigkeitsmatrix A ist ein Teilgraph des Graphen, der aus den Knoten und Kanten in $\tilde{\mathcal{T}}$ besteht. Betrachtet man eine Zeile von A (die einem gewissen Knoten V entspricht), so benötigt man für jedes Nichtnullelement dieser Zeile eine Addition und eine Multiplikation bei der Berechnung des Matrix-Vektorproduktes. Jedes Nichtnullelement dieser Zeile entspricht dabei entweder einer Kopplung mit einem Knoten V', der über eine Kante mit V verbunden ist, oder aber einer Kopplung von V mit sich selbst. Insgesamt ergibt sich damit

$$FP(Ax) = 2(n_V + 2n_E) \ .$$

Daraus folgt

$$\begin{aligned}
FP(Ax) &\overset{(1)}{=} 2n_T + 6n_E - 2n_F + 2 \\
&\overset{(2)}{=} 2n_T + 6n_E - 2(2n_T + \frac{1}{2}n_{F_B}) + 2 \\
&= -2n_T + 6n_E - n_{F_B} + 2 \\
&\overset{(3)}{\leq} -2n_T + 9n_F - n_{F_B} + 2 \\
&\overset{(2)}{=} -2n_T + 9(2n_T + \frac{1}{2}n_{F_B}) - n_{F_B} + 2 \\
&= 16n_T + \frac{7}{2}n_{F_B} + 2\ .
\end{aligned}$$

□

Bemerkung 5.7

- In zwei Dimensionen gilt eine ähnliche Aussage (siehe [1], S. 93):

  $$FP(Ax) = 7n_F + 3n_{V_B} + 2\ ,$$

  wobei n_F entsprechend die Anzahl der Dreiecke der zweidimensionalen Triangulierung bezeichne.

- Analog zu den Gleichungen (1) und (2) aus dem Beweis von Satz 5.6 gilt für die Anzahl der Subsimplices auf dem Rand:

  $$\begin{aligned}
  n_{F_B} + n_{V_B} &= n_{E_B} + 2 && (1') \\
  2n_{E_B} &= 3n_{F_B} && (2')
  \end{aligned}$$

  Daraus ergibt sich $n_{V_B} - 2 = \frac{1}{2}n_{F_B} = \frac{1}{3}n_{E_B}$. Das bedeutet: Die Aufgabe, die Anzahl der Seitenflächen auf dem Prozessorrand zu minimieren, ist äquivalent zu den Aufgaben, die Anzahl der Kanten oder Knoten auf dem Prozessorrand zu minimieren.

Bemerkung 5.8 (Lastverteilung bei Mehrgitterverfahren)
Bei Einsatz eines parallelen Mehrgitterverfahrens ist in den Prolongations-, Restriktions- und Glättungsschritten zusätzliche Kommunikation nötig, welche bei der Lastbalancierung berücksichtigt werden muss. Außerdem müssen neben $\mathcal{T}_J$ sämtliche übrigen Triangulierungsstufen (bis auf evtl. die Anfangstriangulierung $\mathcal{T}_0$) geeignet auf die Prozessoren verteilt werden. Laut [1] ist für ein multiplikatives Mehrgitterverfahren die gleichmäßige Verteilung jedes Gitterlevels anzustreben. Die gemeinsame Speicherung von Elterntetraedern und deren Kindern auf einem

Prozessor (vgl. Regel (M5) aus Definition 4.1 sowie Satz 4.3 (2)) führt bereits zu verminderter vertikaler Kommunikation im Prolongations-/Restriktionsschritt, eine weitere Verbesserung kann durch Zusammenfassen von Eltern und Kindern mehrerer Generationsstufen (Clustering) erreicht werden, was gleichzeitig die Komplexität des zugehörigen dualen Graphen senkt.

Einen bedeutenden Punkt bei der Lastverteilung stellt die Einhaltung der Regeln (M1) bis (M5) dar, die ein zulässiges Multilevelgitter beschreiben (vgl. Definition 4.1), insbesondere der Eigenschaft, dass Geschwistertetraeder zusammen mit ihrem gemeinsamen Elterntetraeder auf einem Prozessor abgespeichert werden müssen. Dies kann auf zwei Weisen gewährleistet werden:

(A) Zu $\mathcal{T}_J$ wird der duale Graph $G(\mathcal{T}_J)$ aufgestellt und durch einen Lastbalancierer eine Partitionierung m berechnet, welche folgendermaßen modifiziert wird: alle Geschwistertetraeder $T' \in \mathcal{K}(T)$ des gemeinsamen Elterntetraeders $T \in \mathcal{T}_{J-1}$ werden auf *den* Prozessor migriert, welcher den meisten Geschwistern durch die Partitionierung m zugeordnet wurde.

(B) Geschwistertetraeder werden von vorneherein zusammengefasst und dem Lastbalancierer als ein einzelner Knoten des dualen Graphen bekannt gemacht.

Es ist unmittelbar klar, dass der Ansatz B dem ad-hoc-Ansatz A überlegen ist: Der entscheidende Nachteil von A liegt darin, dass sich durch die Modifikation der Partitionierung die Menge $E_{\text{cut}}(m)$ unkontrolliert ändert und deswegen unter Umständen die durch den Graphpartitionierer minimierte Größe $\text{cost}_{\text{comm}}(m)$ und damit der Kommunikationsaufwand ungünstig zunimmt.

Demgegenüber wird in Ansatz B die Information über Geschwisterzusammengehörigkeit direkt im Graphen kodiert und bei der Lastbalancierung entsprechend berücksichtigt. Ein weiterer Vorteil von Ansatz B liegt darin, dass die Anzahl der Knoten des dualen Graphen geringer ist als bei Ansatz A (bei uniformer Verfeinerung um den Faktor 8), was einen Speicherplatz- und Rechenzeitgewinn während der Lastbalancierungsphase bedeutet.

Die Verfolgung des Ansatzes B führt zur Einführung sogenannter *Multiknoten*, die Idee ist ähnlich der des Matching (vgl. Definition 5.4). Im Unterschied dazu werden aber nicht gewisse Graphkanten, sondern die Elternrelation der Tetraeder zur Bestimmung von solchen Knoten des dualen Graphen benutzt, die zu einem Multiknoten zusammengefasst werden.

Dazu wird die Elternabbildung

$$E : \bigcup_{k=1,\dots,J} \mathcal{G}_k \quad \to \quad \bigcup_{k=1,\dots,J} \mathcal{G}_{k-1}$$

aus Definition 3.1 formal erweitert um $E(T) := T$ für $\ell(T) = 0$. Definiere nun folgende Relation $\approx$ auf der Menge aller Tetraeder einer Triangulierungshierarchie $\mathcal{H}$ durch

$$T_1 \approx T_2 \quad :\Leftrightarrow \quad E(T_1) = E(T_2) \ .$$

Zwei Tetraeder stehen in der Relation $\approx$, wenn sie beide denselben Elterntetraeder besitzen. Offensichtlich ist $\approx$ eine Äquivalenzrelation.

Definition 5.9 (Reduzierter dualer Graph) $\mathcal{T}$ sei eine Triangulierung aus $\mathcal{H}$. Der *reduzierte duale Graph* $G'(\mathcal{T}) = (V', E')$ wird nun folgendermaßen definiert: V' ist die Menge der Äquivalenzklassen auf $V = \mathcal{T}$ bezüglich $\approx$, d.h.

$$\begin{aligned} V' := V/_{\approx} &= \{[T]_{\approx} : T \in \mathcal{T}\} \\ &= \big\{\{T \in \mathcal{T} : E(T) = S\} : S \in \mathcal{E}(\mathcal{T})\big\} \ , \end{aligned}$$

wobei $\mathcal{E}(\mathcal{T}) := \{E(T) : T \in \mathcal{T}\}$ die Menge der Eltern von Tetraedern aus $\mathcal{T}$ bezeichne. $v' \in V'$ heißt *Multiknoten.*

V' induziert nun die Kantenmenge

$$\begin{aligned} E' = \ &\{(v_1', v_2') : \quad \exists\, v_1 \in v_1', v_2 \in v_2' : (v_1, v_2) \in E\} \\ &\setminus \{(v', v') : v' \in V'\} \ . \end{aligned}$$

Die Gewichtung der Knoten und Kanten geht dabei auf die übliche Weise vom dualen Graphen $G(\mathcal{T})$ auf den reduzierten dualen Graphen $G'(\mathcal{T})$ über:

$$\begin{aligned} \alpha(v') &:= \sum_{v \in v'} \alpha(v) \ , \\ \beta\big((v_1', v_2')\big) &:= \sum_{e \in E \cap (v_1' \times v_2')} \beta(e) \ . \end{aligned}$$

Das Gewicht eines Multiknotens ist die Summe der Gewichte der darin zusammengefassten Knoten. Das Gewicht einer Kante zwischen zwei Multiknoten v_1' und v_2' ist die Summe aus dem Gewicht der Kanten, die im dualen Graphen $G(\mathcal{T})$ Knoten aus v_1' mit Knoten aus v_2' verbinden.

In Abbildung 5.5 ist ein Beispiel für einen reduzierten dualen Graphen $G'(\mathcal{T})$ gegeben. Die zu einem Multiknoten zusammengefassten Tetraeder sind jeweils dick umrahmt. Beachte, dass der entsprechende duale Graph $G(\mathcal{T})$ (siehe Abbildung 5.1) 19 Knoten und 23 Kanten, der reduzierte duale Graph dagegen nur 8 Knoten und 9 Kanten besitzt.

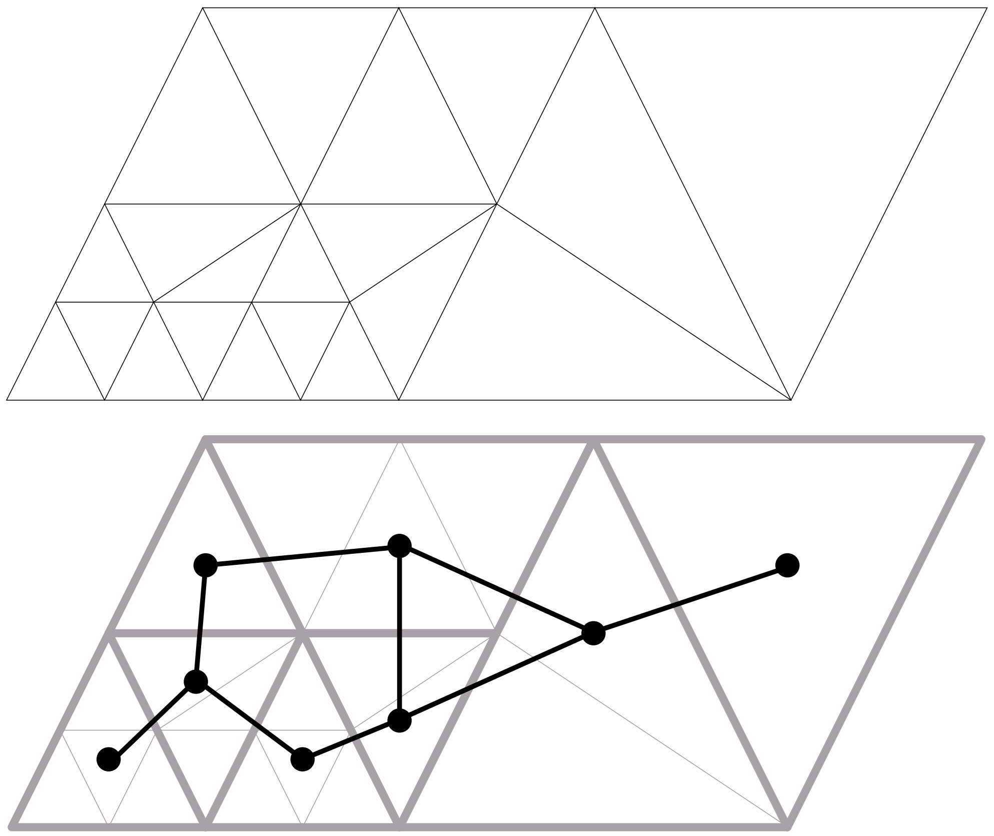


Abbildung 5.5: Triangulierung $\mathcal{T}$ und zugehöriger reduzierter dualer Graph $G'(\mathcal{T})$.

Bemerkung 5.10

- Im Falle der Anfangstriangulierung stimmen der duale Graph $G(\mathcal{T}_0)$ und der reduzierte duale Graph $G'(\mathcal{T}_0)$ überein. Zur Verteilung des Anfangsgitters muss also kein gesondertes Verfahren implementiert werden.

- Zur Aufstellung des reduzierten dualen Graphen wurde ein Iterator über die Menge $\mathcal{E}(\mathcal{T})$ implementiert, der folgende Charakterisierung von $\mathcal{E}(\mathcal{T})$ verwendet:

$$S \in \mathcal{E}(\mathcal{T}) \quad \Leftrightarrow \quad \begin{cases} S \in \mathcal{T} \text{ und } \ell(S) = 0 \quad \textbf{oder} \\ S \text{ ist verfeinert und } \mathcal{K}(S) \cap \mathcal{T} \neq \emptyset \ . \end{cases} \tag{5.4}$$

  Diese Äquivalenz folgt sofort aus obiger Definition der Elternabbildung E.

- Im Falle der überlappten Speicherung von $S \in \mathcal{E}(\mathcal{T})$ kann der Zugriff auf die Kinder $\mathcal{K}(S)$ nur von der Ghostkopie von S aus erfolgen; falls S nicht überlappt ist, erfolgt der Zugriff vom Master aus. Deswegen definiert man für ein Level k und einen Prozessor p die Menge

$$\mathcal{E}_k(\mathcal{T}, p) := \mathcal{E}(\mathcal{T}) \cap [\mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p)] \ .$$

  Die Mengen $\mathcal{E}_k(\mathcal{T}, p)$, $p = 1, \ldots, P$, bilden eine Partition von $\mathcal{E}_k(\mathcal{T}) := \mathcal{E}(\mathcal{T}) \cap \mathcal{G}_k$, und zwar für jedes Level $k = 0, \ldots, J$:
  Beweis: Wegen

$$\bigcup_{p=1}^{P} [\mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p)] = \mathcal{G}_k$$

  folgt $\bigcup_{p=1}^{P} \mathcal{E}_k(\mathcal{T}, p) = \mathcal{E}(\mathcal{T}) \cap \mathcal{G}_k = \mathcal{E}_k(\mathcal{T})$. Sei weiterhin $T \in \mathcal{E}_k(\mathcal{T}, p) \cap \mathcal{E}_k(\mathcal{T}, q)$. Ist T nicht überlappt gespeichert, so folgt direkt $p = q$. Ist dagegen T überlappt gespeichert, so gibt es nach Satz 4.3 höchstens zwei verschiedene Objektkopien von T, nämlich

$$T \in \mathsf{HasGh}_k(r) \quad \text{und} \quad T \in \mathsf{Gh}_k(s)$$

  für genau ein r sowie für genau ein $s \neq r$. Es folgt $T \notin \mathcal{E}_k(\mathcal{T}, r)$ aufgrund der Definition von $\mathcal{E}_k(\mathcal{T}, r)$. Damit ergibt sich $s = p = q$. □

Für jeden Multiknoten $v' \in V'$ gibt es nach Konstruktion genau einen Elterntetraeder $S \in \mathcal{E}(\mathcal{T})$ mit

$$v' = v_S := \{T \in \mathcal{T} : E(T) = S\} \ .$$

Außerdem gibt es genau einen Prozessor p mit $S \in \mathcal{E}_{\ell(S)}(\mathcal{T}, p)$, von dem aus Zugriff auf die Kinder von S möglich ist (falls S überhaupt Kinder besitzt, also verfeinert ist).

Die Lastverteilung in DROPS besteht nun aus folgenden Schritten: Der reduzierte duale Graph zur feinsten Triangulierung $\mathcal{T}_J$ wird aufgestellt und dem Lastbalancierungs-Tool übergeben, welches eine Partitionierung

$$m_{\mathcal{T}_J} : V' \to \{1, \ldots, P\}$$

berechnet. Damit ist eine Zuordnung der Tetraeder aus $\mathcal{T}_J$ auf die Prozessoren gegeben.

Anschließend folgt die Lastmigration: Sie wird durch folgenden Algorithmus in Abbildung 5.6 beschrieben und unter Zuhilfenahme des Transfermoduls von DDD abgewickelt.

$$m = m_{\mathcal{T}}$$

bezeichne dabei eine durch einen Lastbalancierer berechnete Partitionierung des reduzierten dualen Graphen $G'(\mathcal{T})$ zur Triangulierung $\mathcal{T}$. Für die momentane Lastmigration in DROPS wird aufgrund des eingesetzten CG-Lösers nur $\mathcal{T} = \mathcal{T}_J$ gewählt.

Aus gegebenem Anlass sei nochmals darauf hingewiesen, dass der tatsächliche Transfer erst durch `DDD_XferEnd`() angestoßen wird, und zwar entsprechend den vorher abgesetzten Transferkommandos. Werden für einen Tetraeder mehrere `DDD_XferCopyObj`-Kommandos gegeben, die einen Transfer auf denselben Prozessor, aber einmal als Ghost und einmal als Master bewirken, so wird nur der entsprechende Mastertransfer wirksam. Wird für einen Tetraeder T auf einem Prozessor p ein `DDD_XferDeleteObj`-Kommando gegeben, gleichzeitig auf einem anderen Proyessor mit `DDD_XferCopyObj` ein Transfer von T nach p veranlasst, so bleibt T auf Prozessor p erhalten und erhält die gemäß dem Transfer geforderte Master- oder Ghostpriorität.

Zur Unterscheidung der einzelnen Transfers sind diese in E- und *M-Transfers* eingeteilt. E-Transfers betreffen die Transfers der $S \in \mathcal{E}_k(\mathcal{T}, p)$, M-Transfers die Mastertransfers der Kinder $\mathcal{K}(S)$. Die Menge der Tetraeder, die auf Prozessor p in Level $k \geq 1$ gespeichert sind und für welche möglicherweise (falls $d \neq p$) M-Transfers durchgeführt werden, ist als

$$M_k(\mathcal{T}, p) := \bigcup_{S \in \mathcal{E}_{k-1}(\mathcal{T}, p)} \mathcal{K}(S)$$

definiert. Beachte, dass der Schnitt der Mengen $M_k(\mathcal{T}, p)$ und $\mathcal{E}(\mathcal{T})$ i.A. nichtleer ist, d.h. es kann Tetraeder geben, für die sowohl E- als auch M-Transfers durchgeführt werden. In dem 2D-Beispiel, welches in Abbildung 5.5 dargestellt ist, trifft dies z.B. für das Elterndreieck des Multiknotens in der linken unteren Ecke zu.

```
Algorithm Migration( G_0(p), ..., G_J(p); m, T )
{
    DDD_XferBegin();

    for ( k = 0, ..., J ) do
        for ( S ∈ E_k(T, p) ) do
        {
            d := m(v_S);                                        (1)
            if ( d ≠ p ) then
            { // E-Transfer
                if ( K(S) = ∅ ) then
                { // E_1-Transfer
                    DDD_XferCopyObj( S, Ma, d );                (2)
                    DDD_XferDeleteObj( S );                     (3)
                }
                else
                { // E_2-Transfer
                    DDD_XferCopyObj( S, Gh, d );                (4)
                    if ( S ∈ Gh_k(p) ) then
                        DDD_XferDeleteObj( S );                 (5)
                }
                for ( T ∈ K(S) )
                { // M-Transfer
                    if ( K(T) = ∅ or T ∈ HasGh_{k+1}(p) ) then
                    { // M_1-Transfer
                        DDD_XferCopyObj( T, Ma, d );            (6)
                        DDD_XferDeleteObj( T );                 (7)
                    }
                    else // beachte Kindtetraeder von T!
                    { // M_2-Transfer
                        DDD_XferCopyObj( T, Ma, d );            (8)
                        if ( T ∈ E(T, p) and m(v_T) ≠ p )
                            DDD_XferDeleteObj( T );             (9)
                        else
                            DDD_XferPriorityChange( T, Gh );    (10)
                    }
                }
            }
        }

    DDD_XferEnd();
}
```

Abbildung 5.6: Der Migrationsalgorithmus.

Lemma 5.11

a) Alle lokalen Tetraederobjekte, für die evtl. ein M-Transfer vorgenommen wird, sind Master.

$$\forall p \quad \forall 1 \leq k \leq J: \qquad M_k(\mathcal{T},p) \subset \mathsf{Ma}_k(p)$$

b) Alle lokalen Tetraederobjekte, für die evtl. ein E- und ein M-Transfer auf demselben Prozessor vorgenommen wird, sind nichtüberlappend gespeichert.

$$\forall p \quad \forall 1 \leq k \leq J: \qquad M_k(\mathcal{T},p) \cap \mathcal{E}_k(\mathcal{T},p) \subset \mathsf{Ma}_k(p) \setminus \mathsf{HasGh}_k(p)$$

c) Die Mengen $M_k(\mathcal{T},p)$, $p = 1,\ldots,P$ bilden für jedes Level $1 \leq k \leq J$ eine Partition von

$$M_k(\mathcal{T}) := \bigcup_{S \in \mathcal{E}_{k-1}(\mathcal{T})} \mathcal{K}(S) \; .$$

Beweis: Seien $1 \leq k \leq J$ ein Level und $1 \leq p \leq P$ ein Prozessor.

a) Sei $T \in M_k(\mathcal{T},p)$. Nach Definition von $M_k(\mathcal{T},p)$ ist $S := E(T) \in \mathcal{E}_{k-1}(\mathcal{T},p)$. Ist S überlappend gespeichert, so gilt aufgrund der Definition von $\mathcal{E}_{k-1}(\mathcal{T},p)$ $S \in \mathsf{Gh}_{k-1}(p)$, und (M5) liefert die Behauptung. Ist S nicht überlappend gespeichert, so folgt $S \notin \mathcal{G}_k(q)$ für alle $q \neq p$. Wegen (M3) muss T dann als Master auf Prozessor p gespeichert sein.

b) Sei $T \in M_k(\mathcal{T},p) \cap \mathcal{E}(\mathcal{T},p)$. Dann ist nach Teil a) $T \in \mathsf{Ma}_k(p)$. Nach Definition von $\mathcal{E}_k(\mathcal{T},p)$ folgt $T \notin \mathsf{HasGh}_k(p)$.

c) folgt aus der Tatsache, dass die Mengen $\mathcal{E}_{k-1}(\mathcal{T},p)$, $p = 1,\ldots,P$ eine Partition von $\mathcal{E}_{k-1}(\mathcal{T})$ bilden:

$$\begin{aligned} M_k(\mathcal{T}) &= \bigcup_{S \in \bigcup_{p=1}^{P} \mathcal{E}_{k-1}(\mathcal{T},p)} \mathcal{K}(S) \\ &= \bigcup_{p=1}^{P} \bigcup_{S \in \mathcal{E}_{k-1}(\mathcal{T},p)} \mathcal{K}(S) \\ &= \bigcup_{p=1}^{P} M_k(\mathcal{T},p) \; . \end{aligned}$$

Sei $T \in M_k(\mathcal{T},p) \cap M_k(\mathcal{T},q)$. Dann ist $S := E(T) \in \mathcal{E}_{k-1}(\mathcal{T},p) \cap \mathcal{E}_{k-1}(\mathcal{T},q)$, und wegen der Partitionseigenschaft der Elternmengen folgt $p = q$.

□

Bemerkung 5.12 (Transfers eines Tetraeders) Aufgrund der Partitionseigenschaften von $\mathcal{E}_k(\mathcal{T}, p)$, $p = 1, \ldots, P$, und $M_k(\mathcal{T}, p)$, $p = 1, \ldots, P$ wird für jeden Tetraeder $T \in \mathcal{G}_k$ höchstens *ein* E-Transfer und höchstens *ein* M-Transfer durchgeführt. Aufgrund der Struktur des Algorithmus handelt es sich dabei entweder um einen E_1- oder einen E_2-Transfer bzw. entweder um einen M_1- oder einen M_2-Transfer.

Nun zur Beschreibung des Migrationsalgorithmus: Auf Prozessor p wird für jedes Level $k = 0, \ldots, J$ die Menge $\mathcal{E}_k(\mathcal{T}, p)$ durchlaufen. Sei also $S \in \mathcal{E}_k(\mathcal{T}, p)$ und $d := m(v_S)$ der Prozessor, auf den der Multiknoten v_S migriert werden soll. Gilt $d = p$, so ist nichts zu tun. Im Folgenden werden die im Falle $d \neq p$ durchgeführten Transfers beschrieben:

Ist S unverfeinert, so ist S wegen (M4) nicht überlappend gespeichert. Also wird S als Master auf den Prozessor $m(v')$ verschoben (E_1-Transfer, Befehle (2), (3)).

Ist S verfeinert, so wird für S ein E_2-Transfer durchgeführt: S wird als Ghost auf Prozessor q kopiert (4); war S auf Prozessor p als Ghost gespeichert, so wird die Ghostkopie auf Prozessor p gelöscht (5). Für die Kinder von S wird je einer der M-Transfers durchgeführt:

- Ist $T \in \mathcal{K}(S)$ selbst verfeinert und wird T auf Prozessor p zur Realisierung der Eltern-Kind-Referenz benötigt (d.h. $T \notin \mathsf{HasGh}_{k+1}(p)$), so muss dem mit einem M_2-Transfer Sorge getragen werden: T wird als Master auf Prozessor p kopiert (8). Falls für T noch ein E-Transfer (in Level $k+1$) durchgeführt wird, so wird die lokale Kopie von T auf Prozessor p gelöscht (9), da die Kinder von T im Zuge dieses Transfers auf Level $k+1$ mitverschoben werden. Falls kein solcher E-Transfer von T ansteht, muss zur Realisierung der Eltern-Kind-Referenz eine Ghostkopie auf Prozessor p verbleiben (10).
- Andernfalls reicht das Verschieben von T als Master auf Prozessor d aus (M_1-Transfer, Befehle (6), (7)); auf eventuelle Kinder von T muss nicht geachtet werden.

Bemerkung 5.13 (M-Transfers) Beachte, dass aufgrund von Lemma 5.11 M-Transfers nur für Masterelemente durchgeführt werden. Im Falle eines M_2-Transfers lässt sich eine noch genauere Aussage treffen: Da M_2-Transfers nur für $T \notin \mathsf{HasGh}_k(p)$ ausgeführt werden, folgt, dass solche Tetraeder stets nichtüberlappend gespeichert sind. Es stellt deswegen auch keine Einschränkung dar, in dem **if** -Statement vor (9) nur $T \in \mathcal{E}(\mathcal{T}, p)$ statt allgemeiner $\mathcal{T} \in \mathcal{E}(\mathcal{T})$ abzufragen.

Die Transfers sind so gewählt, dass die Zulässigkeit eines verteilten Multilevelgitters durch den Migrationsprozess erhalten bleibt. Dies ist Aussage des folgenden Satzes:

Satz 5.14 (Korrektheit der Lastverteilung) *Gegeben sei ein zulässiges verteiltes Multilevelgitter*

$$\tilde{\mathcal{M}}^{\mathrm{alt}} = \left(\mathcal{M}^{\mathrm{alt}}(1), \ldots, \mathcal{M}^{\mathrm{alt}}(P)\right) ,$$

dessen zugehörige Triangulierungshierarchie gerade $\mathcal{H} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ entspreche. Außerdem sei

$$m : V' \to \{1, \ldots, P\}$$

eine Partitionierung des reduzierten dualen Graphen einer Triangulierung $\mathcal{T}$, wobei $\mathcal{T} \subset \mathcal{T}_j$ für ein j mit $0 \leq j \leq J$. Dann ist der auf $\tilde{\mathcal{M}}^{\mathrm{alt}}$ angewendete Migrationsalgorithmus wohldefiniert, und das aus $\tilde{\mathcal{M}}^{\mathrm{alt}}$ durch die Migration gemäß m hervorgehende verteilte Multilevelgitter

$$\tilde{\mathcal{M}}^{\mathrm{neu}} = \left(\mathcal{M}^{\mathrm{neu}}(1), \ldots, \mathcal{M}^{\mathrm{neu}}(P)\right)$$

ist wiederum zulässig.

Beweis: Wie im Beweis zu Satz 4.8 seien wieder (M1') – (M5') die für das neu entstandene verteilte Multilevelgitter $\tilde{\mathcal{M}}^{\mathrm{neu}}$ zu beweisenden Bedingungen. Entsprechend bezeichnen (M1) – (M5) die für das Ausgangsmultilevelgitter $\tilde{\mathcal{M}}^{\mathrm{alt}}$ gültigen Zulässigkeitsbedingungen.

Sei k ein Level, $0 \leq k \leq J$.

zur Wohldefiniertheit: Zu zeigen ist, dass zu jedem $S \in \mathcal{E}_k(\mathcal{T}, p)$ – falls S verfeinert ist – lokal Zugriff auf die Kinder $\mathcal{K}(S)$ möglich ist. Sei also $S \in \mathcal{E}_k(\mathcal{T}, p)$ und o.B.d.A. $\mathcal{K}(S) \neq \emptyset$ (sonst ist nichts zu zeigen). Dann liefert Lemma 5.11 die Behauptung:

$$\mathcal{K}(S) \subset M_{k+1}(\mathcal{T}, p) \subset \mathsf{Ma}_{k+1}(p) .$$

Alle Kinder von S sind also ebenso lokal auf Prozessor p gespeichert.

zu (4.1): $\mathcal{G}_k^{\mathrm{neu}}(p) = \mathsf{Ma}_k^{\mathrm{neu}}(p) \cup \mathsf{Gh}_k^{\mathrm{neu}}(p)$ gilt per Definition von $\mathcal{G}_k^{\mathrm{neu}}(p)$. Die Partitionseigenschaft

$$\mathsf{Ma}_k^{\mathrm{neu}}(p) \cap \mathsf{Gh}_k^{\mathrm{neu}}(p) = \emptyset$$

folgt aus der Tatsache, dass DDD für jedes verteilte Objekt auf jedem Prozessor höchstens eine lokale Kopie speichert (*effiziente Speicherung*, vergleiche Definition 2.11). Diese Eigenschaft wird insbesondere durch das DDD-Transfermodul eingehalten.

zu (M1'): Da im Zuge der Migration keine neuen *verteilten* Objekte entstehen (nur evtl. neue lokale Objektkopien), gilt $\mathcal{G}_k^{\text{neu}} \subset \mathcal{G}_k^{\text{alt}}$. Außerdem fallen auch keine verteilten Objekte weg, denn zu jedem `DDD_XferDeleteObj`-Kommando in (3), (5), (7) bzw. (9) gibt es auch ein vorhergehendes `DDD_XferCopyObj`-Kommando in (2), (4), (6) bzw. (8).

$$\Rightarrow \qquad \mathcal{G}_k^{\text{neu}} = \mathcal{G}_k^{\text{alt}} \ .$$

Deswegen wird im Folgenden einfach die Bezeichnung $\mathcal{G}_k$ gewählt, also ohne einen hochgestellten Zusatz zur Unterscheidung von „alt" oder „neu" zu verwenden.

Es muss nur die Inklusion

$$\mathcal{G}_k \subset \bigcup_{q=1}^{P} \mathsf{Ma}_k^{\text{neu}}(q)$$

gezeigt werden. Sei also $T \in \mathcal{G}_k$. Dann existiert wegen (M1) ein Prozessor p mit $\mathcal{T} \in \mathsf{Ma}_k^{\text{alt}}(p)$.

1. Fall: Für T wurde kein Transfer oder nur ein E_2-Transfer durchgeführt. Dann ist der Master nach wie vor auf Prozessor p gespeichert: $T \in \mathsf{Ma}_k^{\text{neu}}(p)$.

2. Fall: Es gab einen E_1- oder M-Transfer für T. Dann folgt

$$T \in \mathsf{Ma}_k^{\text{neu}}(d)$$

für $d = m(v_T)$ (E_1-Transfer) bzw. $d = m(v_{E(T)})$ (M-Transfer).

zu (M2'): Sei $T \in \mathsf{Ma}_k^{\text{neu}}(p) \cap \mathsf{Ma}_k^{\text{neu}}(q)$. Wegen (M1), (M2) gibt es genau ein r mit $T \in \mathsf{Ma}_k^{\text{alt}}(r)$.

Angenommen, es sei $p \neq q$, d.h. es gibt zwei Masterkopien von T auf unterschiedlichen Prozessoren. Da T vorher nur auf r als Master gespeichert war, muss dies durch zwei `DDD_XferCopyObj`-Kommandos ausgelöst worden sein:

$$\texttt{DDD\_XferCopyObj}(\ T, \mathsf{Ma}, d_i\ )\ , \quad i = 1, 2\ .$$

Für T werden höchstens ein E- und ein M-Transfer durchgeführt, vgl. Bemerkung 5.12. Deswegen wurde eines der beiden Kommandos in einem E-Transfer, das andere in einem M-Transfer von T gegeben. Da im Rahmen eines E-Transfers dafür nur (2) in Frage kommt (Anweisung (4) würde T als Ghost kopieren), handelt es sich um einen E_1-Transfer. Dann ist aber

$$T \in \mathcal{E}(\mathcal{T}) \quad \text{und} \quad \mathcal{K}(T) = \emptyset\ .$$

Aus der Charakterisierung (5.4) von $\mathcal{E}(\mathcal{T})$ folgt dann aber $\ell(T) = 0$. Andererseits muss wegen des M-Transfers von T gelten, dass $\ell(T) > 0$ ist. Widerspruch, also ist doch $p = q$.

zu (M3'): Sei $T \in \mathcal{G}_k$, o.B.d.A. gelte $\mathcal{K}(T) \neq \emptyset$ (sonst ist nichts zu zeigen). Sei weiter $T' \in \mathcal{K}(T)$ und

$$T' \in \mathsf{Ma}_{k+1}^{\text{neu}}(p) \ .$$

Zu zeigen ist $T \in \mathcal{G}_{k+1}^{\text{neu}}(p)$.

1. Fall: Für T und T' wurden keine Transfers durchgeführt. Dann ist $T' \in \mathsf{Ma}_{k+1}^{\text{alt}}(p)$, und die Behauptung folgt aus (M3).

2. Fall: Nur für T wurden Transfers durchgeführt. Dann ist $T' \in \mathsf{Ma}_{k+1}^{\text{alt}}(p)$, und wegen (M3) gilt

$$T \in \mathcal{G}_k^{\text{alt}}(p) \ .$$

Für T kann kein E-Transfer durchgeführt worden sein, sonst wäre auch für $T' \in \mathcal{K}(T)$ ein M-Transfer ausgeführt worden. Da aber ein Transfer von T' in diesem Fall ausgeschlossen wurde, muss es sich bei T um einen M-Transfer handeln. Also gilt

$$T \in M_k(\mathcal{T}, q) \quad \text{und} \quad d = m(v_{E(T)}) \neq q$$

für einen Prozessor q.

1. T war vor der Migration nicht überlappt. Dann ist $q = p$, und wegen

$$T \notin \mathsf{HasGh}_k^{\text{alt}}(p) \quad \text{und} \quad \mathcal{K}(T) \neq \emptyset$$

handelt es sich um einen M_2-Transfer. Da für T kein E-Transfer auftritt, wird neben (8) zusätzlich Befehl (10) gegeben. Also wird T als Master auf den Prozessor $d \neq p$ kopiert, auf Prozessor p verbleibt aber eine Ghostkopie von T. Es folgt $T \in \mathcal{G}_k^{\text{neu}}(p)$.

2. T war vor der Migration überlappt. Dann folgt aus (M5) sowie (M2)

$$T \in \mathsf{Gh}_k^{\text{alt}}(p) \ .$$

Wegen Lemma 5.11 ist $q \neq p$ und $T \in \mathsf{HasGh}_k^{\text{alt}}(q)$. Der M-Transfer von T wird also auf Prozessor q ausgeführt, genauer gesagt, handelt es sich um einen M_1-Transfer. Da darüber hinaus kein E-Transfer von T stattfindet, folgt

$$\left.\begin{array}{lcl} d = p & \Rightarrow & T \in \mathsf{Ma}_k^{\text{neu}}(p) \\ d \neq p & \Rightarrow & T \in \mathsf{Gh}_k^{\text{neu}}(p) \end{array}\right\} \quad \Rightarrow \quad T \in \mathcal{G}_k^{\text{neu}}(p) \ .$$

3. Fall: Nur für T' wurden Transfers durchgeführt. Dann ist kein M-Transfer von T' denkbar, da sonst auch ein E-Transfer für T durchgeführt worden wäre. Also wurde für T' nur ein E-Transfer ausgeführt, d.h.

$$T' \in \mathcal{E}_{k+1}(\mathcal{T}, q)$$

für einen Prozessor q. Aufgrund der Charakterisierung (5.4) von $\mathcal{E}(\mathcal{T})$ folgt aus $\ell(T') > 0$ direkt $\mathcal{K}(T') \neq \emptyset$. Also handelt es sich um einen E_2-Transfer. Es werden folgende zwei Fälle unterschieden:

1. T' war vor der Migration nicht überlappt, also gilt $T' \in \mathsf{Ma}^{\text{alt}}_{k+1}(q)$ und (5) wird nicht ausgeführt. Dann ist auch $T' \in \mathsf{Ma}^{\text{neu}}_{k+1}(q)$ und (M2') liefert $q = p$. Also liefert (M3)

$$T \in \mathcal{G}^{\text{alt}}_k(p) = \mathcal{G}^{\text{neu}}_k(p) \ ,$$

da für T kein Transfer stattfand.

2. T' war vor der Migration schon überlappt, also $T' \in \mathsf{Gh}^{\text{alt}}_{k+1}(q)$. Also werden (4) und (5) ausgeführt, d.h. die Ghostkopie wird auf einen anderen Prozessor verschoben; der Master dagegen bleibt, wo er ist.
$\Rightarrow \quad T' \in \mathsf{Ma}^{\text{alt}}_{k+1}(p) \quad \overset{(M3)}{\Rightarrow} \quad T \in \mathcal{G}^{\text{alt}}_k(p) = \mathcal{G}^{\text{neu}}_k(p)$.

4. Fall: Sowohl für T als auch für T' wurden Transfers durchgeführt. Betracht man die Transfers von T, so lassen sich dabei folgende Fälle unterschieden:

1. Es gibt einen E-Transfer von T, also

$$T \in \mathcal{E}_k(\mathcal{T}, q)$$

für einen Prozessor q und $d = m(v_T) \neq q$. Dann gibt es auch einen M-Transfer von T' nach d. Wegen (M2') folgt $d = p$. Durch den E-Transfer von T nach $d = p$ ist

$$T \in \mathcal{G}^{\text{neu}}_k(p)$$

gewährleistet.

2. Es gibt keinen E-Transfer von T, demnach gibt es auch keinen M-Transfer von $T' \in \mathcal{K}(T)$. Also muss es einen E-Transfer von T' geben. Wegen $\ell(T') > 0$ und (5.4) ist dies ein E_2-Transfer, der Master ändert sich also nicht:
$\Rightarrow \quad T' \in \mathsf{Ma}^{\text{alt}}_{k+1}(p) \quad \overset{(M3)}{\Rightarrow} \quad T \in \mathcal{G}^{\text{alt}}_k(p)$.
Nun zu T: Da es keinen E-Transfer von T gibt, findet für T also nur ein M-Transfer statt. Dann

$$T \in M_k(\mathcal{T}, q)$$

für ein q. Zwei Fälle werden unterschieden:

(a) T war vor der Migration nicht überlappt. Dann ist wegen $T \in \mathcal{G}^{\text{alt}}_k(p)$ in diesem Fall $q = p$. Außerdem gilt insbesondere $T \notin \mathsf{HasGh}^{\text{alt}}_k(p)$, und wegen $\mathcal{K}(T) \neq \emptyset$ handelt es sich um einen M_2-Transfer. Genauer gesagt, werden die Anweisungen (8) und (10) ausgeführt, da es keinen E-Transfer von T gibt. Es folgt also

$$T \in \mathcal{G}^{\text{neu}}_k(p) \ .$$

(b) T war vor der Migration überlappt. Wegen $T' \in \mathsf{Ma}_{k+1}^{\text{alt}}(p)$ muss unter Beachtung von (M1), (M5) gelten, dass $T \in \mathsf{Gh}_k^{\text{alt}}(p)$ und $T \in \mathsf{HasGh}_k^{\text{alt}}(q)$ mit $q \neq p$ ist. Es findet für T also ein M_1-Transfer nach $d = m(v_{E(T)}) \neq q$ statt.

$$\left.\begin{array}{lcl} d = p & \Rightarrow & T \in \mathsf{Ma}_k^{\text{neu}}(p) \\ d \neq p & \Rightarrow & T \in \mathsf{Gh}_k^{\text{neu}}(p) \end{array}\right\} \quad \Rightarrow \quad T \in \mathcal{G}_k^{\text{neu}}(p) \ .$$

zu (M4'): Sei $T \in \mathsf{Gh}_k^{\text{neu}}(p)$. Angenommen, T wäre unverfeinert, d.h. $\mathcal{K}(T) = \emptyset$. Dann folgt aus (M4)

$$T \notin \bigcup_{q=1}^{P} \mathsf{Gh}_k^{\text{alt}}(q) \ .$$

Die Ghostkopie von T auf Prozessor p nach der Migration kann also nur durch einen Transfer entstanden sein. Wegen $\mathcal{K}(T) = \emptyset$ kommen nur E_1- und M_1-Transfers in Frage. Durch diese entstehen aber keine Ghostkopien. Widerspruch, also ist T doch verfeinert: $\mathcal{K}(T) \neq \emptyset$.

zu (M5'): Sei

$$T \in \mathsf{Gh}_k^{\text{neu}}(p) \ .$$

Wegen (M4') ist $\mathcal{K}(T) \neq \emptyset$. Sei $T' \in \mathcal{K}(T)$. Zeige $T' \in \mathsf{Ma}_{k+1}^{\text{neu}}(p)$.

1. Fall: T war vor der Migration nicht überlappt gespeichert, also

$$T \in \mathsf{Ma}_k^{\text{alt}}(q) \setminus \mathsf{HasGh}_k^{\text{alt}}(q)$$

für genau einen Prozessor q. Die Ghostkopie nach der Migration kann nur in Anweisung (4) während eines E_2-Transfers oder in Anweisung (10) während eines M_2-Transfers entstanden sein. Es ist nicht möglich, dass sowohl (4) als auch (10) für T ausgeführt worden sind, weil Anweisung (10) für T nur dann aktiv wird, falls es keinen E-Transfer von T gibt (beachte **if** -Statement vor (9)!). Es werden also zwei Fälle unterschieden:

1. Für T wurde Anweisung (10) ausgeführt (im Rahmen eines M_2-Transfers). Dann gibt es – wie soeben ausgeführt – keinen E-Transfer von T, d.h. es werden nur die Befehle (8) und (10) für T durchgeführt. Somit wird T als Master auf Prozessor

$$d = m(v_{E(T)}) \neq q$$

kopiert und auf Prozessor q selbst in einen Ghost umgewandelt, d.h.

$$T \in \mathsf{Gh}_k^{\text{neu}}(q) \ .$$

Da keine weiteren Anweisungen für T ausgeführt wurden, folgt $q = p$.
Da T vor der Migration nicht überlappt war, folgt mit (M3)

$$T' \in \mathsf{Ma}_{k+1}^{\mathrm{alt}}(p) \ .$$

Da es für T keinen E-Transfer gibt, gibt es für T' keinen M-Transfer. Es sind also folgende Alternativen denkbar:

(a) Gar kein Transfer von T': Dann ist nach wie vor $T' \in \mathsf{Ma}_{k+1}^{\mathrm{neu}}(p)$.

(b) E-Transfer von T': Wegen $\ell(T') > 0$ und (5.4) handelt es sich um einen E_2-Transfer, also ebenso $T' \in \mathsf{Ma}_{k+1}^{\mathrm{neu}}(p)$.

2. Für T wurde Anweisung (4) ausgeführt (im Rahmen eines E_2-Transfers). Dann ist für einen Prozessor q

$$T \in \mathcal{E}_k(\mathcal{T}, q) \quad \text{und} \quad d = m(v_T) \neq q \ .$$

Wegen $T \in \mathsf{Ma}_k^{\mathrm{alt}}(q)$ wird (5) nicht ausgeführt. Außerdem wird – wie oben erwähnt – (10) nicht ausgeführt, so dass (4) die einzige Anweisung ist, um eine Ghostkopie auf Prozessor p zu erzeugen. Somit folgt $d = p$.
Für alle Kinder von T wird ein M-Transfer nach $d = p$ durchgeführt, also auch für T'. Somit folgt wegen (6) bzw. (8): $T' \in \mathsf{Ma}_{k+1}^{\mathrm{neu}}(p)$.

2. Fall: T war vor der Migration bereits überlappt gespeichert. Dann gibt es nach Satz 4.3 genau ein q mit

$$T \in \mathsf{Gh}_k^{\mathrm{alt}}(q) \ .$$

(M5) liefert darüber hinaus

$$T' \in \mathsf{Ma}_{k+1}^{\mathrm{alt}}(q) \ .$$

Es werden zwei Fälle unterschieden:

1. Es gab einen E-Transfer von T. Dann

$$T \in \mathcal{E}_k(\mathcal{T}, q) \quad \text{und} \quad d = m(v_T) \neq q \ .$$

Wegen $\mathcal{K}(T) \neq \emptyset$ wird für T ein E_2-Transfer ausgeführt, und zwar Anweisungen (4) und (5). Da weiterhin (10) für T nicht aufgerufen wird (sonst gäbe es keinen E-Transfer), gibt es nach der Migration genau einen Prozessor, auf dem T als Ghost gespeichert ist, und zwar Prozessor d:

$$T \notin \mathsf{Gh}_k^{\mathrm{neu}}(r) \quad \forall\, r \neq d \ .$$

Es folgt also $d = p$.
Wegen des E-Transfers von T werden für alle Kinder von T M-Transfers nach $d = p$ ausgeführt, also auch für T'. Folglich $T' \in \mathsf{Ma}_{k+1}^{\text{neu}}(p)$.

2. Es gab keinen E-Transfer von T. Somit bleiben folgende Möglichkeiten:
   - M-Transfer von T: Dann ist $T \in M_k(\mathcal{T}, r)$ für einen Prozessor r sowie $d = m(v_{E(T)}) \neq r$. Lemma 5.11 liefert $T \in \mathsf{Ma}_k^{\text{alt}}(r)$, wegen der Überlappung genauer

     $$T \in \mathsf{HasGh}_k^{\text{alt}}(r) \ .$$

     Es handelt sich also um einen M_1-Transfer. Da sich dadurch die Position der Ghostkopie von T nicht ändert, folgt $T \in \mathsf{Gh}_k^{\text{alt}}(p)$, somit folgt mit Satz 4.3 $q = p$.
   - T wurde gar nicht transferiert: Dann ändert sich die Position der Ghostkopie von T ebenso nicht. Also folgt wiederum $q = p$.

   In beiden Fällen folgt $q = p$, also insbesondere

   $$T' \in \mathsf{Ma}_{k+1}^{\text{alt}}(p) \ .$$

   Da es keinen E-Transfer von T gab, wurde auch für T' kein M-Transfer durchgeführt. Die beiden möglichen Alternativen sind:

   (a) E-Transfer von T': Wegen $\ell(T') > 0$ und (5.4) handelt es sich um einen E_2-Transfer, der Master von T' hat also seinen Prozessor nicht gewechselt. Es folgt also $T' \in \mathsf{Ma}_{k+1}^{\text{neu}}(p)$.

   (b) T' wurde gar nicht transferiert: Dann ist ebenso $T' \in \mathsf{Ma}_{k+1}^{\text{neu}}(p)$.

□

Kapitel 6

Parallele numerische Experimente

Im folgenden Kapitel wird die Skalierbarkeit verschiedener Module getestet, die parallelisiert wurden. Dies sind

- der parallele Verfeinerungsalgorithmus,
- die parallele Diskretisierung (d.h. das Aufstellen der verteilten Steifigkeitsmatrix **A** sowie der verteilten rechten Seite **b**) und
- der parallele CG-Algorithmus zur Lösung des aus der Diskretisierung gewonnenen Gleichungssystems.

Zusätzlich wird ein kompletter paralleler adaptiver Zyklus betrachtet (Anfangstriangulierung aufstellen, diskretisieren, lösen, Fehler schätzen, verfeinern, Last verteilen usw.) und die Rechenzeiten für die einzelnen Module gemessen.

Zur Ermittlung der skalierten Effizienz für einen der genannten parallelen Algorithmen werden die Ausführungszeiten auf verschiedenen Prozessorzahlen gemessen und miteinander verglichen, wobei vorausgesetzt wird, dass die Teilprobleme auf den einzelnen Prozessoren alle etwa von der gleichen Größe n' sind. Aus diesem Grund wird im Rahmen der Effizienzmessungen nur uniforme Verfeinerung eingesetzt. Pro Versuch werden jeweils zwei Läufe durchgeführt und aus den gemessenen Zeiten das arithmetische Mittel gebildet.

Die parallele Zeitmessung selber wird folgendermaßen durchgeführt: Zunächst werden alle Prozessoren durch einen `MPI_Barrier`-Befehl synchronisiert, um zu gewährleisten, dass die lokalen Zeitmessungen auf den einzelnen Prozessoren zum selben Zeitpunkt beginnen. Die lokalen Messungen werden gestoppt, wenn das untersuchte Teilproblem gelöst ist. Die lokal ermittelten Rechenzeiten unterscheiden sich natürlich i.A. voneinander; ihr *Maximum* wird als parallele Ausführungszeit bezeichnet und mittels eines `MPI_Reduce`-Aufrufs bestimmt.

In Abschnitt 6.3 wird die Aussagefähigkeit von Effizienz und Speedup kritisch diskutiert. Dabei wird gezeigt, dass diese Größen nicht alleine von der Qualität des parallelen Codes, sondern ebenso stark von externen Bedingungen abhängig sind.

Die Rechnungen wurden auf einem Linux-Cluster mit 16 SMP-Knoten durchgeführt, die durch ein 100-Mbit-Ethernet-Netzwerk miteinander verbunden sind. Jeder Knoten besteht aus einem PC mit Dualboard, ausgestattet mit zwei 1.2-GHz-Prozessoren (Intel Pentium III) und 4 GB Hauptspeicher, so dass insgesamt 32 Prozessoren zur Verfügung stehen. Der C++-Quellcode wurde mit dem gcc-3.1 bei höchster Optimierungsstufe (-O3 -funroll-loops -march=pentium3) übersetzt.

Die Experimente wurden für die Prozessorzahlen $P = 1, 2, 4, 8, 16, 32$ durchgeführt. Bis zur Anzahl von 16 Prozessoren wurden immer die Prozessoren verschiedener Rechner eingesetzt, so dass auf jedem Rechner höchstens ein Prozess des parallelen Laufs ausgeführt wurde. Erst für $P = 32$ wurden auf jedem Rechner alle beiden Prozessoren eingesetzt. Entgegen der ursprünglichen Erwartung hat es sich nämlich gezeigt, dass ein paralleler Lauf auf zwei Prozessoren *verschiedener* Rechner schneller ist als einer auf den beiden Prozessoren *eines* Rechners, obwohl die Kommunikation im letzteren Fall eigentlich viel schneller verlaufen sollte. Vergleiche dazu Abschnitt 6.1.

6.1 Skalierbarkeit des parallelen Verfeinerungsalgorithmus

Zur Messung der skalierten Effizienz des parallelen Verfeinerungsalgorithmus wird als Testproblem die Verfeinerung des Würfels $\Omega = (0,1)^3$ betrachtet. Dieser wird in n_0 Teilwürfel zerlegt, von denen jeder aus sechs Tetraedern besteht. Diese $6n_0$ Tetraeder bilden die Anfangstriangulierung $\mathcal{T}_0$, welche gleichmäßig auf die P Prozessoren verteilt wird.

n_0 wird dabei in Abhängigkeit von P so gewählt, dass auf jeden Prozessor durchschnittlich $6^4 = 1\,296$ Tetraeder der Anfangstriangulierung entfallen. Das Gitter wird zunächst zweimal uniform verfeinert, bevor die eigentliche Zeitmessung beginnt, um eine ausreichend hohe Problemgröße zu gewährleisten. Die feinste Triangulierung $\mathcal{T}_2$ besteht dann aus $P \cdot 82\,944$ Tetraedern. Aufgrund der gleichmäßigen Verteilung der Anfangstriangulierung ist die lokale Teilproblemgröße n' jedes beteiligten Prozessors unabhängig von P, was Voraussetzung für die Messung der skalierten Effizienz ist.

Ausgehend von dieser Triangulierungshierarchie $(\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2)$ werden nun zwei uniforme Verfeinerungsschritte parallel durchgeführt, deren Ausführungszeiten jeweils gemessen werden und in Tabelle 6.1 dargestellt sind. Nach Beendigung der Verfeinerung besteht die dann feinste Triangulierung $\mathcal{T}_5$ aus $P \cdot 5\,308\,416$ Tetraedern, das gesamte Multilevelgitter umfasst etwa $P \cdot 6$ Millionen Tetraeder. Beachte, dass die zur Berechnung der Effizienz herangezogenen Zeitmessungen in dieser wie in den anderen folgenden Tabellen nicht mit allen Nachkommastellen angegeben sind.

P	1	2	4	8	16	32
$T_1(P)$	3.1	3.5	4.2	5.2	5.8	9.3
$T_2(P)$	24.9	27.2	30.5	33.0	36.9	55.0
$E_1^{sc}(P)$	100 %	90 %	75 %	60 %	54 %	34 %
$E_2^{sc}(P)$	100 %	92 %	82 %	76 %	68 %	45 %

Tabelle 6.1: Ausführungszeiten des parallelen Verfeinerungsalgorithmus.

Die Tabelle 6.1 zeigt die Ausführungszeiten der zwei aufeinanderfolgenden uniformen Verfeinerungsschritte in Sekunden, sowie die sich daraus jeweils ergebende skalierte Effizienz in %. Im Falle $P = 1$ verdeutlichen die seriellen Ausführungszeiten für die beiden Verfeinerungsschritte die lineare Komplexität des Verfeinerungsalgorithmus: Die achtfache Tetraederanzahl verursacht eine etwa achtmal so große Ausführungszeit.

Nun zur Bewertung der Effizienzergebnisse: Die achtfach höhere Problemgröße beim zweiten Verfeinerungsschritt und damit ein günstigeres Verhältnis von Rechenzeit zu Kommunikationsaufwand spiegelt sich in einer besseren Effizienz gegenüber dem ersten Verfeinerungsschritt wider, besonders für höhere Prozessorzahlen (vgl. die beiden letzten Zeilen von Tabelle 6.1).

Zu bemerken ist außerdem der Einbruch der Effizienzrate beim Übergang von 16 auf 32 Prozessoren bei beiden Verfeinerungsschritten. Das scheint an der Tatsache zu liegen, dass dabei zum ersten Mal jeder Rechnerknoten mit zwei Prozessen statt einem ausgelastet wird. Als Beleg dafür dient folgendes Experiment, in dem die Messung der Effizienz auf $P = 2$ Prozessoren wiederholt wurde; statt zwei Prozessoren auf verschiedenen Rechnern wurden diesmal die zwei Prozessoren *eines* Rechners eingesetzt.

Die Resultate sind in Tabelle 6.2 angegeben, als Vergleich wurden diesen die Werte aus Tabelle 6.1 für $P = 2$ Prozessoren auf unterschiedlichen Rechnern gegenübergestellt. Die Ausführungszeiten sind um bis zu 41 % gesteigert! Dies liegt möglicherweise an Speicherzugriffskonflikten, da die beiden Prozessoren eines Rechners auf denselben Speicher zugreifen. Finden viele Speicherzugriffe und Speicherallokationen statt, so behindern sich die Prozessoren dabei gegenseitig

$P = 2$	gleicher Rechner	unterschiedliche Rechner
$T_1(P)$	4.7	3.5
$T_2(P)$	38.5	27.2
$E_1^{sc}(P)$	66 %	90 %
$E_2^{sc}(P)$	65 %	92 %

Tabelle 6.2: Ausführungszeiten des parallelen Verfeinerungsalgorithmus auf $P = 2$ Prozessoren, die sich auf demselben Rechner befinden. Zum Vergleich dieselbe Messung für $P = 2$ Prozessoren auf unterschiedlichen Rechnern.

und müssen mehrere Taktzyklen ungenutzt verstreichen lassen, bevor sie die eigenen Daten laden können, die sie zur Ausführung der gerade anstehenden Operation benötigen.

6.2 Skalierbarkeit des parallelen CG-Algorithmus

Als Testproblem wird folgende Poissongleichung auf dem Würfel $\Omega = (0,1)^3$ betrachtet, wobei auf der Seitenfläche $\Sigma_1 := [0,1]^2 \times \{0\}$ Dirichletrandwerte und auf den übrigen Seitenflächen Neumannrandwerte vorgeben werden. Die Funktion

$$u(x,y,z) := x(1-x)y(1-y)z(1-z)$$

erfüllt die Differentialgleichung

$$\begin{aligned} -\Delta u &= f \qquad \text{in } \Omega && (6.1)\\ u &= 0 \qquad \text{auf } \Sigma_1 && (6.2)\\ \frac{\partial u}{\partial n} &= g_N \qquad \text{auf } \partial\Omega \setminus \Sigma_1 \;, && (6.3) \end{aligned}$$

wobei $f := -\Delta u$ und $g_N := \frac{\partial u}{\partial n}$ passend gewählt werden.

Die Triangulierung besteht aus n_0 Teilwürfeln, von denen jeder in sechs Tetraeder unterteilt ist. Die Unterteilung von Ω in Teilwürfel wird wie zuvor abhängig von der Prozessoranzahl P so gewählt, dass insgesamt $P \cdot 262\,144$ Teilwürfel entstehen und damit auf jeden Prozessor durchschnittlich 1 572 864 Tetraeder bzw. 274 625 Knoten (und damit Unbekannte) entfallen.

Die Messung der Skalierbarkeit erfolgt zum einen für die Diskretisierung (Aufstellen der verteilten Steifigkeitsmatrix A und der verteilten rechten Seite $\mathbf{b}$) und zum anderen für den parallelen CG-Algorithmus. Da die Durchführung der Diskretisierung trivial parallelisierbar ist und keine Kommunikation dabei stattfindet, ergibt sich eine fast optimale Skalierbarkeit, wie Tabelle 6.3 zeigt.

P	1	2	4	8	16	32
$T_{disc}(P)$	13.0	13.1	13.3	13.4	13.5	14.5
$E^{sc}_{disc}(P)$	100 %	99 %	97 %	97 %	96 %	89 %

Tabelle 6.3: Skalierte Effizienz der parallelen Diskretisierung.

Die leichte Zunahme der Ausführungszeiten für $P = 1, \ldots, 16$ Prozessoren lässt sich durch leichte Lastungleichgewichte zwischen den Prozessoren erklären. Beim Übergang von 16 auf 32 Prozessoren kommt immerhin eine ganze Sekunde hinzu, deren Ursache wiederum aufgetretene Speicherzugriffskonflikte zu sein scheinen.

Bei der Messung der skalierten Effizienz für den parallelen CG-Algorithmus muss beachtet werden, dass dieser keine $\mathcal{O}(n)$-Laufzeit besitzt: Da die Anzahl der CG-Iterationen, um ein bestimmtes Residuum (hier $\varepsilon = 10^{-7}$) zu unterschreiten, mit wachsender Unbekanntenzahl ansteigt (vgl. Tabelle 6.4), können die Laufzeiten $T_{CG}(P)$ für den CG-Löser auf verschiedenen Prozessoranzahlen P nicht direkt miteinander verglichen werden. Stattdessen wird aus der Anzahl der Iterationen auf die durchschnittliche Rechenzeit $T_{iter}(P)$ für einen CG-Iterationsschritt geschlossen.

Der Aufwand für eine Iteration wird bestimmt durch eine Matrix-Vektor-Multiplikation und zwei Skalarprodukte, hinzu kommen Linearkombinationen von Vektoren. All diese Operationen sind in $\mathcal{O}(n)$-Zeit durchführbar, auch das Matrix-Vektor-Produkt, da es sich um eine dünnbesetzte Matrix handelt: In jeder Zeile kommen höchstens k_{nz} Nichtnulleinträge vor; da k_{nz} aber durch die maximale Anzahl der Knoten bestimmt ist, mit welchen ein Knoten durch Kanten in der Triangulierung $\mathcal{T}$ verbunden sein kann, folgt, dass k_{nz} unabhängig von n ist, eine Matrix-Vektor-Multiplikation also höchstens $2k_{\mathrm{nz}}\, n = \mathcal{O}(n)$ Additionen und Multiplikationen benötigt.

P	1	2	4	8	16	32
$T_{CG}(P)$	68.9	117.5	178.8	223.0	309.4	715.8
# iter	464	650	863	893	1207	1450
$T_{iter}(P)$	0.148	0.181	0.207	0.250	0.256	0.494
$E^{sc}_{iter}(P)$	100 %	82 %	72 %	60 %	58 %	30 %

Tabelle 6.4: Skalierte Effizienz für eine parallele CG-Iteration.

Die skalierte Effizienz für eine CG-Iteration ist in Tabelle 6.4 dargestellt. Die nur mittelmäßigen Effizienzwerte bleiben hinter den Erwartungen zurück. Dies liegt eindeutig daran, dass der Rechenaufwand pro CG-Iteration relativ gering ist. Erst bei Kombination mit (parallelen) Präkonditionierungstechniken ist zu erwarten, dass deutlich mehr Rechenlast auftritt und sich infolgedessen bessere Effizienzergebnisse einstellen.

Der nachfolgende Abschnitt demonstriert, dass sich unter anderen Bedingungen bessere Effizienzwerte erzielen lassen, und zwar *ohne* Modifikation des Codes. Insofern sei davor gewarnt, der Effizienz bei der Beurteilung eines parallelen Codes zuviel Beachtung zu schenken.

6.3 Aussagekraft von Speedup und Effizienz

Im vorangegangenen Abschnitt wurde die Effizienz einer CG-Iteration gemessen. Um die Aussagekraft der gewonnenen Effizienzzahlen kritisch zu hinterfragen, wurden zwei zusätzliche Experimente durchgeführt. Dabei wurde wiederum die Effizienz einer CG-Iteration gemessen, diesmal aber unter veränderten Bedingungen:

- Messung auf demselben Cluster, diesmal wurde das Programm aber ohne Optimierung übersetzt.
- Messung auf einem anderen (älteren) Linux-Cluster mit leistungsschwächeren Prozessoren.

Beispiel 1 Das Programm, welches vorher mit der höchsten Optimierungsstufe (Compileroption -O3) übersetzt worden war, wurde diesmal ohne Optimierung (-O0) übersetzt und die Effizienzmessungen wiederholt. Wie zu erwarten, steigen die Ausführungszeiten im Vergleich zur optimierten Version an. Für die Effizienz ergeben sich gegenüber der optimierten Version deutlich verbesserte Werte.

P	1	2	4	8	16	32
$T_{iter}(P)$	0.313	0.338	0.363	0.426	0.426	0.603
$E^{sc}_{iter}(P)$	100 %	93 %	86 %	73 %	73 %	52 %

Tabelle 6.5: Skalierte Effizienz für eine parallele CG-Iteration, ohne Optimierung (Beispiel 1).

Beispiel 2 Die Messung wurde auf einem Linux-Cluster mit vier SMP-Knoten (Dualboards, jeweils zwei 600-MHz-Prozessoren, Pentium III) und 100-Mbit-Ethernet-Netzwerkverbindung wiederholt. Wegen des kleineren Arbeitsspeichers (1 GB pro Rechnerknoten) wurde die Problemgröße entsprechend angepasst (eine Verfeinerung weniger, also nur etwa ein Achtel der Unbekannten). Das Programm wurde ohne Optimierung übersetzt. Hierbei werden vergleichbare Effizienzraten wie in Beispiel 1 erzielt. Wieder ist ein Einbruch für 8 Prozessoren zu verzeichnen, da dann auf jedem Rechnerknoten beide Prozessoren eingebunden sind.

Diese Verbesserung der Effizienz in beiden Beispielen gegenüber der Messung in Abschnitt 6.2 ist nicht weiter verwunderlich, wenn man sich veranschaulicht,

P	1	2	4	8
$T_{iter}(P)$	0.108	0.115	0.120	0.166
$E^{sc}_{iter}(P)$	100 %	93 %	90 %	65 %

Tabelle 6.6: Skalierte Effizienz für eine parallele CG-Iteration, langsameres Cluster, ohne Optimierung (Beispiel 2).

dass die Effizienz stark von dem Verhältnis von Kommunikations- zu Rechenzeit abhängt.

Die parallele Ausführungszeit $T(P)$ einer bestimmten Aktion auf P Prozessoren setzt sich zusammen aus der reinen Rechenzeit $T_R(P)$ und der Kommunikationszeit $T_K(P)$:

$$T(P) = T_R(P) + T_K(P) \ .$$

Zur Berechnung der skalierten Effizienz $E^{sc}(P)$ wird nun diese parallele Ausführungszeit $T(P)$ ins Verhältnis zur seriellen Ausführungszeit $T(1)$ gesetzt. $T(1)$ besteht nur aus der Rechenzeit $T_R(1)$, da keine Kommunikation stattfindet; wegen der gleichen lokalen Problemgröße gilt zudem $T_R(1) \approx T_R(P)$. Zusammengefasst ergibt sich:

$$E^{sc}(P) = \frac{T(1)}{T(P)} = \frac{T_R(1)}{T_R(P) + T_K(P)} \approx \frac{T_R(1)}{T_R(1) + T_K(P)}$$

bzw.

$$E^{sc}(P) \approx \big(1 + Q(P)\big)^{-1} \qquad \text{mit } Q(P) = \frac{T_K(P)}{T_R(1)} \ .$$

Um die skalierte Effizienz $E^{sc}(P)$ auf P Prozessoren zu *verbessern*, muss also das Verhältnis $Q(P)$ von Kommunikations- zu Rechenzeit *verkleinert* werden. Dies kann durch Verkleinerung der Kommunikationszeit $T_K(P)$, aber auch durch Vergrößerung der Rechenzeit $T_R(1)$ geschehen.

Dazu muss nicht unbedingt das Programm selber verändert werden, auch die Modifikation äußerer Bedingungen kann Ursache einer Steigerung der Effizienzrate sein:

- In Beispiel 1 wurde eine Vergrößerung der Rechenzeit $T_R(1)$ durch Verzicht auf Optimierung erreicht.

- In Beispiel 2 änderte sich die Rechenzeit pro Iteration gegenüber Tabelle 6.4 kaum, da der Effekt des langsameren Clusters durch die kleinere Problemgröße ausgeglichen wurde. Die kleinere Problemgröße führt aber zu einem gesunkenen Kommunikationsvolumen, was sich offensichtlich in einer kleineren Kommunikationszeit $T_K(P)$ niederschlägt.

Trotz besserer Effizienzwerte bedeuten aber beide Beispiele einen *Rückschritt* in Bezug auf die eigentlichen Ziele der Parallelisierung, nämlich

- eine Verkleinerung der Gesamtlaufzeit bzw.
- eine Vergrößerung der Gesamtproblemgröße

im Vergleich zur seriellen Behandlung des Problems.

Dies alles zeigt, dass man die Aussagekraft von Effizienz und Speedup nicht zu hoch einschätzen sollte, weil diese stark von äußeren Faktoren abhängen. Insbesondere lassen sich Effizienzangaben über andere parallele Implementationen in der Literatur (z.B. [3], [1]) mit der Effizienz des eigenen Programms nicht direkt vergleichen. Das wäre nur dann möglich, wenn beide unter denselben Bedingungen gemessen würden, d.h. auf einer identischen Maschine, bei Verwendung des gleichen Compilers und derselben Optimierungsstufe, sowie einer vergleichbaren lokalen Problemgröße. In der Praxis lässt sich das aber kaum realisieren, schon alleine deswegen, weil die Autoren ihre Experimentierbedingungen meist nicht vollständig offenlegen.

Trotz all dieser Kritik haben Effizienzmessungen aber ihre sinnvolle Berechtigung: Sie können helfen, das eigene Programm daraufhin zu bewerten, ob sich der Kommunikationsaufwand im Bezug auf den Rechenaufwand in vernünftigen Grenzen hält. Darüber hinaus geben Effizienzmessungen Auskunft darüber, bis zu welcher Prozessorzahl der Einsatz des Programmes auf einer bestimmten Maschine im Sinne einer proportionalen Beschleunigung der Rechenzeit noch lohnenswert ist. Für manche Problemstellungen ist aber gar nicht die Dauer der Rechnung, sondern der zur Verfügung stehende Speicherplatz von Bedeutung. In diesen Fällen lohnt sich ein Einsatz auf vielen Prozessoren auch dann noch, wenn die Effizienz unbefriedigende Werte annimmt.

6.4 Ein paralleler adaptiver Zyklus

Um das Zusammenspiel der einzelnen Komponenten zu testen, wird die parallele Ausführung eines adaptiven Zyklus betrachtet. Als Testbeispiel dient die Poissongleichung auf dem Würfel $\Omega = [-1, 1]^3$ mit Dirichletrandbedingungen. Die rechte Seite f sowie die Dirichletrandwerte g_D sind so gewählt, dass die Funktion

$$u(x, y, z) := \frac{1}{1 + e^{-60(r(x)-0.3)}} \qquad \text{mit } r(x, y, z) := \sqrt{x^2 + y^2 + z^2}$$

die kontinuierliche Lösung darstellt. u ändert sich stark in einer kleinen Umgebung der Kugeloberfläche $S := \{(x, y, z) \in \mathbb{R}^3 : r(x, y, z) = 0.3\}$ und ändert sich

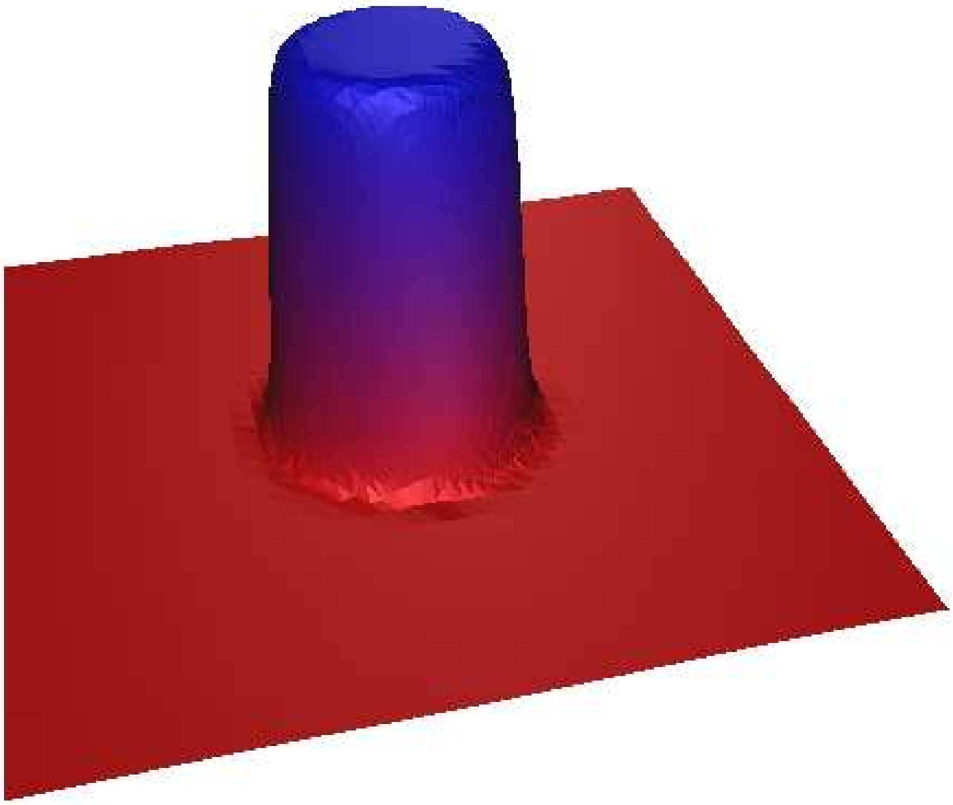


Abbildung 6.1: Diskrete Lösung u_h des adaptiven Verfahrens. Aus Visualisierungsgründen ist $-u_h$ dargestellt.

ansonsten nur wenig: Innerhalb der Kugel liegen die Werte von u nahe bei 0, außerhalb davon nahe bei 1 (siehe auch Abbildung 6.1). Man würde also erwarten, dass der Bereich um S vom Fehlerschätzer markiert und damit durch den Verfeinerungsalgorithmus stark verfeinert wird. Diese Erwartungen erfüllen sich tatsächlich, wie die Abbildungen 6.2 und 6.3 zeigen.

Als Ausgangstriangulierung wird Ω in $4 \times 4 \times 4$ Teilwürfel unterteilt, von denen jeder in 6 Tetraeder zerlegt wird. Diese Ausgangstriangulierung wird gleichmässig auf die Prozessoren verteilt und einmal uniform verfeinert, bevor der adaptive Zyklus (siehe Abbildung 3.1) gestartet wird. Der CG-Algorithmus erhält jeweils $x_0 = 0$ als Startwert, es handelt sich also nicht um eine geschachtelte Iteration. Der adaptive Zyklus endet, wenn die Fehlerschätzungsstrategie weniger als 1 % der Blätter der Triangulierungshierarchie zur Verfeinerung markiert hat. Zur Feh-

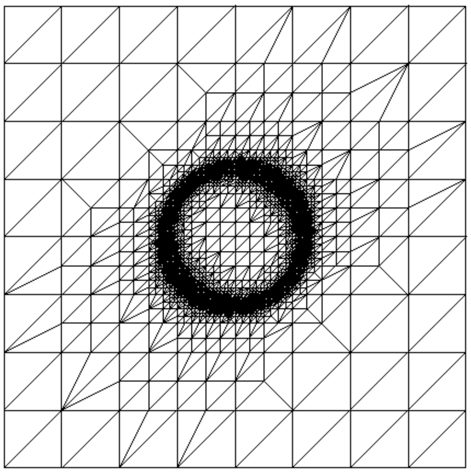


Abbildung 6.2: Adaptives Gitter in der Ebene $z = 0$.

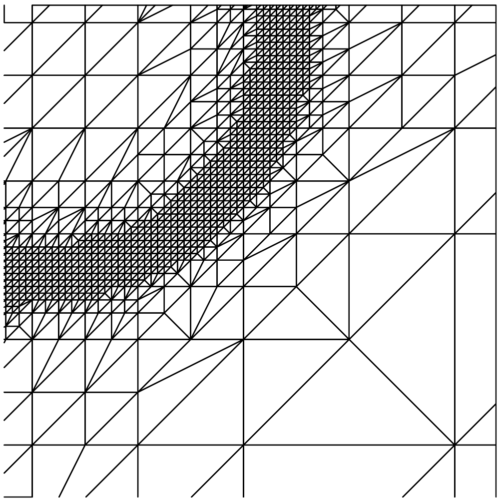


Abbildung 6.3: Ausschnitt.

lerschätzung wird die in Abschnitt 3.6 vorgestellte Strategie angewendet. Dabei wurden $C = 4$ und $\varepsilon = 0.25$ gewählt.

Tabelle 6.7 stellt die einzelnen Ausführungszeiten des seriellen Laufes (also $P = 1$) vor. Insgesamt wird der adaptive Zyklus sechsmal durchlaufen, bevor das Verfahren abbricht. Für jeden Durchlauf sind die Anzahl der Gitterlevel J, die Gesamtanzahl der Tetraeder $\#T$ sowie die Anzahl der Unbekannten n angegeben. Weiterhin aufgeführt sind die Rechenzeiten für die Diskretisierung (T_{disc}), das Lösen mittels CG-Algorithmus bis zu einem Residuum $< 10^{-7}$ (T_{CG}), die Fehlerschätzung (T_{est}) sowie die Verfeinerung (T_{ref}).

Zyklus	J	$\#T$	n	T_{disc}	T_{CG}	T_{est}	T_{ref}
1	1	3456	343	0.033	0.001	0.003	0.015
2	2	5028	555	0.053	0.003	0.004	0.058
3	3	14322	1851	0.176	0.022	0.010	0.254
4	4	55896	7687	0.738	0.465	0.039	1.209
5	5	263202	37047	3.622	8.095	0.200	6.917
6	6	1510224	215758	20.878	144.054	0.526	–
Gesamt		—		25.500	152.638	0.782	8.452

Tabelle 6.7: Serielle Ausführungszeiten für die einzelnen Komponenten des adaptiven Zyklus.

Im sechsten und letzten Durchlauf wird ein Gleichungssystem mit $n = 215\,758$ Unbekannten gelöst, das Multilevelgitter besteht dann aus insgesamt 1.5 Millionen Tetraedern. Aus den Gesamtzeiten für die einzelnen Komponenten wird ersichtlich, dass dabei der letzte Durchlauf am meisten zu Buche schlägt (beim

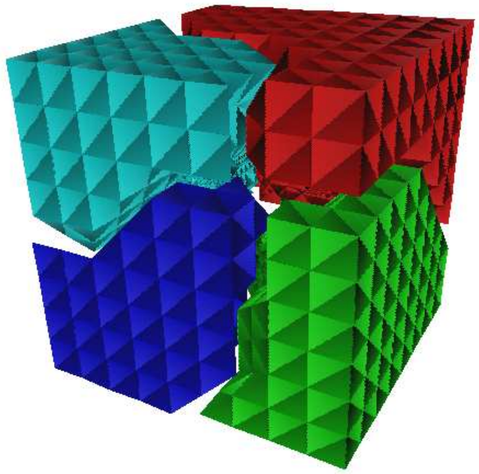


Abbildung 6.4: Verteilt gespeicherte Triangulierung des Würfels Ω auf $P = 4$ Prozessoren.

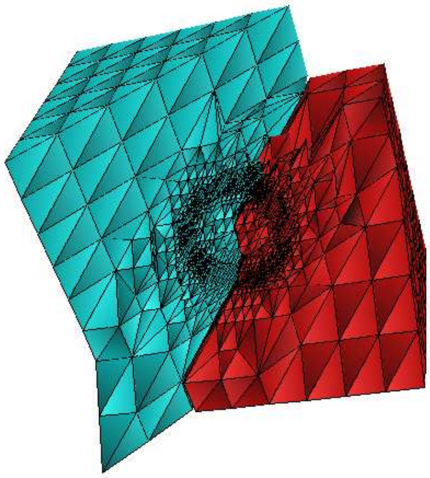


Abbildung 6.5: Innenansicht.

CG-Löser etwa mit 95 % der im CG-Algorithmus aufgewendeten Zeit); demgegenüber fallen die vorhergehenden Durchläufe kaum ins Gewicht. Im Vergleich der einzelnen Komponenten wird außerdem deutlich, dass von der Gesamtrechenzeit für den adaptiven Zyklus (187.4 Sekunden) der größte Anteil auf den CG-Algorithmus entfällt (über 80 %).

Anschließend wird dasselbe Problem parallel auf $P = 2, 4, 8, 16$ gerechnet, und zwar unter Beibehaltung der Gesamtproblemgröße, also im Sinne des *parallelen* Speedup. Die Wahl einer zur Prozessoranzahl P proportionalen Gesamtproblemgröße n wäre zwar wünschenswert, um *skalierte* Effizienzmessungen betreiben zu können, ist wegen der lokal adaptiven Verfeinerung des Gitters aber leider nicht möglich.

In Tabelle 6.8 werden die Gesamtausführungszeiten für die einzelnen Komponenten miteinander verglichen. Beachte, dass im parallelen Fall die Lastverteilung hinzukommt, die sich jeweils an die Verfeinerung anschließt. Aufgeführt sind die Zeiten für das Aufstellen des reduzierten dualen Graphen (T_{dual}), die Graphpartitionierung mit `ParMETIS_V3_PartKway` (T_{lb}) sowie die Migration (T_{migr}). Es wird deutlich, dass der überwiegende Anteil der Lastverteilung auf die Migration entfällt. Die Zeit T_{JOB} ist die Summe der vorangegangenen sieben Ausführungszeiten und beschreibt die Gesamtzeit des kompletten Laufes.

Betrachtet man die Laufzeiten $T_{CG}(P)$ für den parallelen CG-Algorithmus, so wird deutlich, dass bis zu $P = 4$ Prozessoren noch ein gutes Skalierungsverhalten erzielt wird. Auf $P = 8$ Prozessoren fällt der Gewinn gegenüber den auf 4 Prozessoren erzielten Ausführungszeiten eher gering aus, so dass sich hier bereits der

P	$T_{disc}(P)$	$T_{CG}(P)$	$T_{est}(P)$	$T_{ref}(P)$	$T_{dual}(P)$	$T_{lb}(P)$	$T_{migr}(P)$	$T_{JOB}(P)$
1	25.5	152.6	0.8	8.5	–	–	–	187.4
2	12.9	83.8	0.4	6.7	3.4	1.2	11.9	120.3
4	6.6	44.6	0.2	4.4	1.8	1.6	16.0	75.2
8	3.3	38.1	0.2	4.1	1.2	3.3	17.9	68.1
16	1.7	62.2	0.2	6.4	1.1	7.5	21.2	100.3

Tabelle 6.8: Vergleich der Gesamtausführungszeiten für die einzelnen Komponenten des adaptiven Zyklus. Im letzten Durchlauf gibt es insgesamt $\#T = 1\,510\,224$ Tetraeder, $n = 215\,758$ Unbekannte zur feinsten Triangulierung $\mathcal{T}_6$.

P	$E_{disc}(P)$	$E_{CG}(P)$	$E_{JOB}(P)$
1	100 %	100 %	100 %
2	99 %	91 %	78 %
4	97 %	86 %	62 %
8	97 %	50 %	34 %
16	94 %	15 %	12 %

Tabelle 6.9: Parallele Effizienzen einzelner Komponenten und des Gesamtlaufs.

Einsatz doppelt so vieler Prozessoren kaum noch lohnt. Bei $P = 16$ Prozessoren ist gar ein deutlicher Anstieg der Ausführungszeit zu verbuchen.

Dies schlägt sich auch in der parallelen Effizienz $E_{CG}(P)$ des CG-Algorithmus sowie der parallelen Effizienz $E_{JOB}(P)$ für den Gesamtlauf nieder (siehe Tabelle 6.9): Für $P = 1, 2, 4$ lassen sich relativ gute Ergebnisse erzielen, ab $P = 8$ Prozessoren sind unbefriedigende Effizienzraten zu verzeichnen.

Dieses Verhalten liegt an der relativ kleinen Gesamtproblemgröße von $n = 215\,758$ Unbekannten im letzten Durchlauf des adaptiven Zyklus: Auf 4 Prozessoren liegt die durchschnittliche Teilproblemgröße noch bei etwa $n' = 54\,000$ Unbekannten, bei 8 bzw. 16 Prozessoren sind es nur noch $n' = 27\,000$ bzw $n' = 13\,500$ Unbekannten, und damit fällt der Anteil der Rechenzeit verglichen mit dem Kommunikationsaufwand zu gering aus. Im Fall $P = 16$ scheint der Kommunikationsaufwand sogar zu überwiegen, so dass die Einsparung der Rechenzeit aufgrund des doppelten Einsatzes an Prozessoren durch das gleichzeitig erhöhte Kommunikationsvolumen zunichte gemacht wird.

Die sinkenden Effizienzraten sind gleichzeitig ein Effekt der parallelen Effizienz: Die Größe s_p des Interfaces eines Prozessors p – also die Anzahl der Unbekannten auf dem Prozessorrand – liegt in der Größenordnung

$$s_p = \mathcal{O}(n_p^{2/3}) \ ,$$

wobei n_p die Anzahl der Unbekannten auf Prozessor p bezeichne. Da sich der Kommunikationsaufwand pro Iterationsschritt proportional zu s_p verhält, ist das

Verhältnis von Kommunikations- zu Rechenzeit proportional zu $n_p^{-1/3}$ (Verhältnis von Oberfläche zu Volumen). Da sich dieses Verhältnis mit wachsendem P vergrößert, ist zu erwarten, dass gleichzeitig die parallele Effizienz $E(P)$ sinkt.

Um sinnvolle Effizienzmessungen auch für Prozessorzahlen von $P = 8, 16$ durchführen zu können, wurde ein zweites Experiment mit größeren Unbekanntenzahlen durchgeführt. Die Gesamtproblemgröße wurde heraufgesetzt, indem die Parameter der Markierungsstrategie modifiziert wurden ($C = 1$, $\varepsilon = 0.1$). Der adaptive Zyklus wird in diesem Fall siebenmal durchlaufen, das resultierende Multilevelgitter besteht aus 7 Leveln mit insgesamt $\#T = 5\,926\,536$ Tetraedern. Das diskrete Problem auf der feinsten Triangulierung $\mathcal{T}_7$ umfasst $n = 848\,622$ Unbekannte. In Tabelle 6.10 sind die Gesamtausführungszeiten für diesen Lauf dargestellt.

P	$T_{disc}(P)$	$T_{CG}(P)$	$T_{est}(P)$	$T_{ref}(P)$	$T_{dual}(P)$	$T_{lb}(P)$	$T_{migr}(P)$	$T_{JOB}(P)$
1	107.7	1254.0	5.3	31.6	–	–	–	1398.6
2	55.9	707.2	2.8	27.4	13.8	4.5	75.4	887.0
4	28.0	360.6	1.5	21.3	7.4	3.8	49.0	471.6
8	14.1	214.4	0.8	13.9	4.2	6.4	44.6	298.4
16	7.1	175.4	0.5	20.3	3.0	14.6	124.4	345.3

Tabelle 6.10: Vergleich der Gesamtausführungszeiten für die einzelnen Komponenten des adaptiven Zyklus. Im letzten Durchlauf gibt es insgesamt $\#T = 5\,926\,536$ Tetraeder, $n = 848\,622$ Unbekannte zur feinsten Triangulierung $\mathcal{T}_7$.

P	$E_{disc}(P)$	$E_{CG}(P)$	$E_{JOB}(P)$
1	100 %	100 %	100 %
2	96 %	89 %	79 %
4	96 %	87 %	74 %
8	96 %	73 %	59 %
16	95 %	45 %	25 %

Tabelle 6.11: Parallele Effizienzen einzelner Komponenten und des Gesamtlaufs.

Hierbei lassen sich insgesamt bessere Effizienzergebnisse als für das zuerst durchgeführte Experiment erzielen (siehe Tabelle 6.11). Aufgrund der etwa vierfachen Tetraederzahl ergeben sich längere Migrationszeiten, im Fall $P = 16$ ist diese ungewöhnlich hoch, so dass beim Übergang von 8 auf 16 Prozessoren die Ausführungszeit $T_{JOB}(16)$ trotz gesunkener Rechenzeit für den CG-Algorithmus insgesamt ansteigt.

6.4.1 Prozentuale Anteile der einzelnen Komponenten am Gesamtverfahren

Abschließend sollen die Ergebnisse aus beiden Experimenten unter dem Merkmal des Anteils der Einzelkomponenten an der Gesamtrechenzeit betrachtet werden. Die Tabellen 6.12 und 6.13 zeigen die entsprechenden prozentualen Anteile aller Komponenten für beide Experimente; diese Daten sind jeweils in den Abbildungen 6.6 bzw. 6.7 grafisch dargestellt. Qualitativ ergibt sich bei beiden Experimenten ein ähnliches Bild:

P	disc	CG	est	ref	dual	lb	migr
1	13.6	81.4	0.4	4.5	–	–	–
2	10.7	69.7	0.3	5.6	2.8	1.0	9.9
4	8.8	59.3	0.3	5.9	2.4	2.1	21.3
8	4.8	56.0	0.3	6.0	1.8	4.9	26.3
16	1.7	62.0	0.2	6.4	1.1	7.5	21.1

Tabelle 6.12: Erstes Experiment ($n = 215\,758$): Prozentualer Anteil der einzelnen Komponenten an der Gesamtlaufzeit $T_{JOB}(P)$.

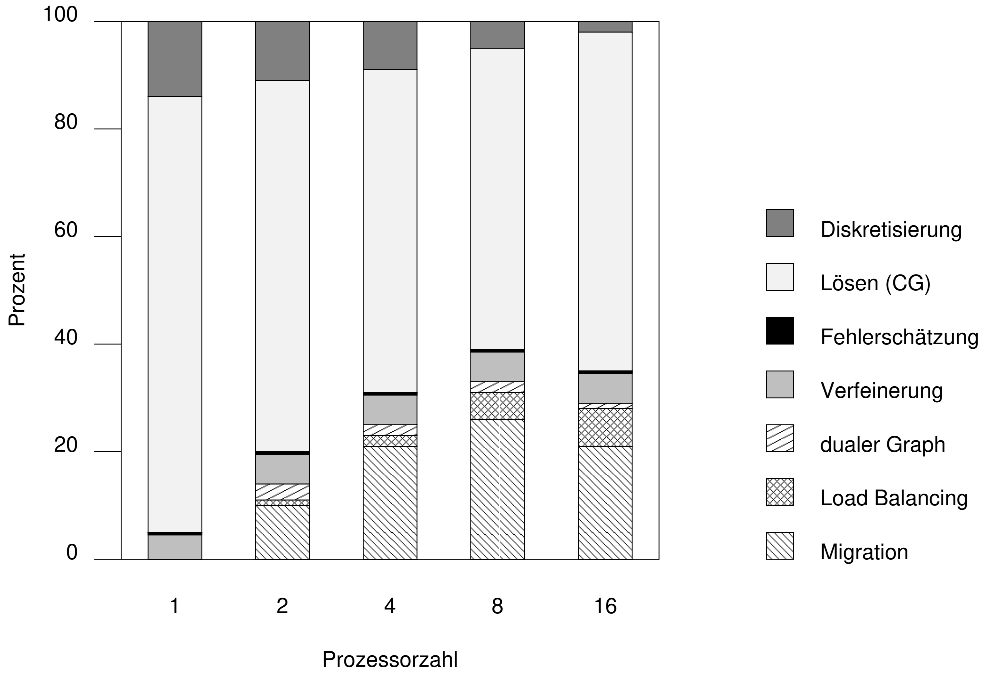


Abbildung 6.6: Erstes Experiment ($n = 215\,758$): Prozentualer Anteil der einzelnen Komponenten an der Gesamtlaufzeit $T_{JOB}(P)$.

Der CG-Algorithmus bildet den größten Anteil der Gesamtrechenzeit, gefolgt von der Migration, welche den zweitgrößten Anteil ausmacht: Für das zweite Experiment z.B. entfällt jeweils über 70 % auf den CG-Algorithmus, nur für $P = 16$

sinkt der Anteil auf 50 % ab, da hier der Migrationsanteil mit 36 % relativ groß ist. Das Aufstellen des reduzierten dualen Graphen sowie die Graphpartitionierung fallen verglichen mit dem Aufwand für die Migration kaum ins Gewicht. Der Anteil der Diskretisierung nimmt aufgrund seiner optimalen Parallelisierbarkeit mit wachsendem P stetig ab. Der Anteil der Verfeinerung bleibt im ersten Experiment etwa gleich; im zweiten Experiment nimmt er leicht zu, bleibt aber in beiden Fällen deutlich unter 10 %.

P	disc	CG	est	ref	dual	lb	migr
1	7.7	89.7	0.4	2.3	–	–	–
2	6.3	79.7	0.3	3.1	1.6	0.5	8.5
4	5.9	76.5	0.3	4.5	1.6	0.8	10.4
8	4.7	71.9	0.3	4.7	1.4	2.1	15.0
16	2.1	50.8	0.2	5.9	0.9	4.2	36.0

Tabelle 6.13: Zweites Experiment ($n = 848\,622$): Prozentualer Anteil der einzelnen Komponenten an der Gesamtlaufzeit $T_{JOB}(P)$.

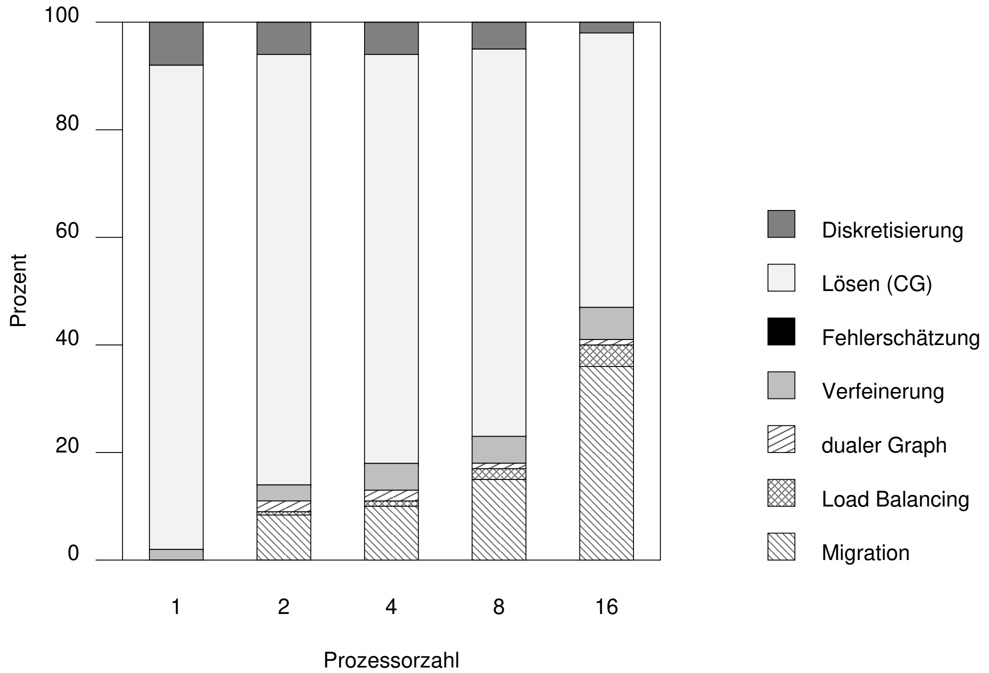


Abbildung 6.7: Zweites Experiment ($n = 848\,622$): Prozentualer Anteil der einzelnen Komponenten an der Gesamtlaufzeit $T_{JOB}(P)$.

Da die Migration den zweitgrößten Anteil am Gesamtverfahren ausmacht, lohnt es sich, für spätere Versionen von DROPS in diesem Bereich über Verbesserungen nachzudenken. Im Hinblick auf die eigentlich interessanten Problemstellungen des SFB 540 verliert die zeitkritische Migrationsphase allerdings evtl. etwas an Bedeutung: Bei der numerischen Behandlung der Stokes- bzw. Navier-Stokes-Gleichung (z.B. mit quadratischen Elementen für die Geschwindigkeit sowie linearen Elementen für den Druck) ergibt sich im Vergleich zur Poissongleichung ein größeres Verhältnis

$$\frac{n}{\#T}$$

von numerischen Daten zu Geometriedaten, so dass zu erwarten ist, dass dann der Anteil der Migration am Gesamtaufwand deutlich niedriger liegen wird. Wie dieser tatsächlich ausfällt, wird sich dann zeigen, wenn erste Strömungsprobleme parallel berechnet werden können.

Trotzdem ist zu beachten, dass eine verbesserte Approximation von u durch u_h durch weitere Adaptionsschritte mit feineren Triangulierungen und damit höheren Tetraederzahlen einhergeht. Somit wächst neben dem Aufwand des iterativen Lösers auch der Migrationsaufwand an, wenn die Genauigkeit der diskreten Lösung u_h gesteigert werden soll. Insofern sollten Konzepte entwickelt werden, um den Migrationsumfang möglichst einzudämmen.

Als Ausblick auf die zukünftige Weiterentwicklung des parallelen DROPS-Code seien abschließend zwei denkbare Lösungsansätze genannt, die das Problem der hohen Migrationsrate von unterschiedlichen Seiten her anzugehen versuchen:

- Zur Graphpartitionierung sollte ein umverteilungsbasiertes Verfahren eingesetzt werden, welches auch die Anzahl der zu migrierenden Graphknoten beachtet und zugunsten einer geringeren Migrationsrate einen etwas schlechteren Balancierungsgrad akzeptiert. Bedauerlicherweise scheint die entsprechende Routine `ParMETIS_V3_AdaptiveRepart` fehlerhaft zu sein, da sie zu einer Speicherverletzung und damit zum Programmabsturz führt.

Noch vielversprechender ist aber der folgende Ansatz, da er die Komplexität des Migrationsproblems entscheidend senkt:

- Die Lastverteilung sollte *vor* der Verfeinerung stattfinden, da dann die Anzahl der Tetraeder noch erheblich kleiner ist. Die Knoten des dualen Graphen müssen dabei je nach Anzahl der Kinder gewichtet werden, die durch die nachfolgende Verfeinerung des entsprechenden Tetraeders voraussichtlich entstehen werden. Eine solche Lastverteilung müsste also zu einem Zeitpunkt vorgenommen werden, zu dem die Verfeinerungsmarkierungen der Tetraeder bereits feststehen, die eigentliche Verfeinerung aber noch nicht stattgefunden hat. Geeignet wäre also eine Lastverteilung in der Mitte des Verfeinerungsalgorithmus, und zwar unmittelbar nach Beendigung der Phase I, bevor Phase II die Verfeinerungen tatsächlich realisiert.

Literaturverzeichnis

[1] Peter Bastian: *Parallele adaptive Mehrgitterverfahren.* Teubner, Stuttgart 1996.

[2] Jürgen Bey: *Finite-Volumen- und Mehrgitterverfahren für elliptische Randwertprobleme.* Dissertation, Universität Tübingen, 1997.

[3] Klaus Birken: *Ein Modell zur effizienten Parallelisierung von Algorithmen auf komplexen, dynamischen Datenstrukturen.* Dissertation, Universität Stuttgart, 1998.

[4] Klaus Birken: *Dynamic Distributed Data (DDD) — a software tool for distributed memory parallelization.* Dokumentation, 1997.
`http://cox.iwr.uni-heidelberg.de/~ddd/`

[5] B. L. Chamberlain: *Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations.* Technical report, University of Washington, 1998.
`http://www.cs.washington.edu/homes/brad/cv/pubs/degree/generals.html`

[6] Sven Groß, Jörg Peters, Volker Reichelt, Arnold Reusken: *The DROPS Package for Numerical Simulations of Incompressible Flows Using Parallel Adaptive Multigrid Techniques.* IGPM Report 211, 2002.

[7] Gundolf Haase: *Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen.* Teubner, Stuttgart; Leipzig 1999.

[8] Wolfgang Hackbusch: *Theorie und Numerik elliptischer Differentialgleichungen.* Teubner, Stuttgart 1986.

[9] Jürgen Jost: *Partielle Differentialgleichungen. Elliptische (und parabolische) Gleichungen.* Springer, Berlin 1998.

[10] George Karypis, Kirk Schloegel, Vipin Kumar: *ParMETIS. Parallel Graph Partitioning and Sparse Matrix Ordering Library.* Dokumentation, University of Minnesota, Army HPC Research Center Minneapolis, 1998.
`http://www-users.cs.umn.edu/~karypis/metis/parmetis/`

[11] George Karypis, Kirk Schloegel, Vipin Kumar: *Parallel Static and Dynamic Multi-Constraint Graph Partitioning.* University of Minnesota, Supercomputing Institute, 2001.

[12] Chris Walshaw, Mark Cross, Martin Everett: *Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes.* J. Parallel Distrib. Comput. 47 (1997), 102–108.