Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Paderborn Center for Parallel Computing

Massiv parallele, adaptive FEM-Simulation auf Tetraedernetzen: Objektmodell, Algorithmen und Datenstrukturen

von

Oliver Marquardt

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

1. Gutachter: Prof. Dr. Burkhard Monien
2. Gutachter: Prof. Dr. Odej Kao

Vorwort

Ohne tatkräftige Unterstützung und Hilfe hätte diese Arbeit nicht entstehen können. An dieser Stelle möchte ich mich bei all denen bedanken, die es mir ermöglicht haben, diese Dissertation anzufertigen.

Bedanken möchte ich mich besonders bei Prof. Dr. Burkhard Monien, der mir den Zugang zum Verständnis paralleler Algorithmen und Verfahren ermöglichte und das *padfem*^{2}-Projekt immerwährend unterstützt hat.

Außerdem bedanke ich mich bei Prof. Dr. Odej Kao, der mir bei fachlichen Problemen immer mit Rat und Tat zur Seite stand und in kritischen Situationen den Rücken frei hielt.

Besonderen Dank gilt auch meinem Freund und Kollegen Dr. Stephan Blazy, der mich an das Thema der parallelen numerischen Simulation herangeführt und dafür begeistert hat. Zudem war er für mich ein besonderer Ansprechpartner sowohl bei fachlichen als auch häufig bei privaten Angelegenheiten und hatte immer ein offenes Ohr für mich.

Meinen Kollegen und den studentischen Hilfskräften des Paderborn Center for Parallel Computing (PC^2) sowie den Mitarbeitern der AG-Monien danke ich ebenfalls recht herzlich für ihre tatkräftige Unterstützung und dafür, dass sie mich in meinen oft phasenweise auftretenden Stimmungschwankungen ertragen haben.

"Felix, qui potuit rerum cognoscere causas."
Publius Vergilius Maro, 70-19 v. Chr.

Gewidmet meinen Eltern,
für aufopferungsvolle Hilfe und Beistand.

Abstract

The research and analysis of partial differential equations is the basis for fundamental understanding of natural phenomena, practical science procedures and industrial applications. The numerical solution of such applications based on partial differential equations on unstructured grids in two or three dimensions is one of the most important problems in mathematical computation and simulation nowadays. To obtain reasonable solutions the approximation usually involves a large number of unknowns. Thus, it can only be solved in a reasonable amount of time by utilizing (massively) parallel computer systems with large memory space.

One of the main problems of numerical simulation software running on parallel computer systems is scalability and efficient usage of such a system. Therefore, a distributed data and object model especially designed for massively parallel finite element applications is presented in this thesis. The main characteristic of this object and data model is a local namespace usage for all elements within a partition of a distributed mesh. Mesh consistency on partition boundaries is automatically maintained by the distributed object model itself.

The object and data model of a distributed mesh is a key component for a simulation environment, because it connects the three main modules of a numerical simulation software, namely the numerical module, the (geometric) adaptation module and the workload balancing and data migration module. All these modules work on their own data model. Thus, an efficient conversion technique between these data models is required. The distributed object and data model developed in this thesis, offers an efficient and scalable approach for this important requirement in parallel simulation applications. The utilization of this mechanism is presented in detail for all three modules.

Mesh modifying modules like the adaptation and the migration module represent bottlenecks for the efficiency of the data structure in the object model. For this reason, two algorithms for these modules, both especially developed for massively parallel usage and working on the distributed data and object model, are introduced in this thesis. The geometric adaptation algorithm is based on irregular refinement and extended with an additional set of rules for quality conservation of element shapes. The migration algorithm works efficiently on large distributed meshes and provides an automatic scalable partition boundary reconstruction, which maintains the local namespace consistency requirement.

To evaluate and verify the object model and the algorithms working on it, a practical implementation in the framework *padfem*^{2} has been carried out. Several artificial benchmark sets are used for analysis of the three main modules and the results are presented in this thesis. Finally, a comprehensive numerical simulation benchmark for computational fluid dynamics is evaluated within the *padfem*^{2}-environment to proof the efficiency of the developed framework including object model, data structures and algorithms.

Inhaltsverzeichnis

1 Einleitung

Motivation

Die Erforschung und Untersuchung partieller Differentialgleichungen stellen die Basis zum Verständnis naturwissenschaftlicher Phänomene dar (s. z.B. [GO96, Str92]). Zu solchen Phänomenen, die Wissenschaftler und Techniker interessieren, zählen z.B. Strömungs- und Mischverhalten von Flüssigkeiten und Gasen oder auch Verformungs- und Reißverhalten von Materialien. Das Studium dieser Phänomene ermöglicht es der Industrie hocheffiziente und kostengünstige Produktdesigns für unterschiedliche Anwendungsbereiche zu entwerfen und zu bauen. Als Beispiel sei hier die Luft- und Raumfahrtindustrie genannt, die auf der Basis von Untersuchungen des Strömungsverhaltens von einzelnen Tragflächen, ganzen Flugzeugen oder Raumfähren Geometriedesigns ermitteln, um Treibstoffeinsparungen, Geschwindigkeitsoptimierungen und Einhaltung von sicherheitsrelevanten Aspekten zu erreichen. Ähnliche Ziele verfolgt auch die Automobilindustrie. Hier werden z.B. ausgiebig Verformungsstudien (Crash-Tests) durchgeführt, um den Sicherheitsstandard von Automobilen immer weiter zu verbessern.

Da die Entwicklung von Prototypen eine teure und ressourcenaufwändige Angelegenheit ist, werden in der Industrie vor der Erstellung eines realen Prototypenmodells per Computersimulationen wichtige Eigenschaften des Prototypens ermittelt. Durch dieses Vorgehen können Entwicklungskosten und Zeitaufwand eingespart werden und unter Umständen sogar sicherheitskritische Probleme umgangen werden. Für bestimmte Produktgruppen ist dies manchmal auch der einzige Weg, diese zu entwickeln und zu testen, da diese bspw. nur einmal gebaut werden (z.B. Bauelemente in Kernkraftwerken, Fracht- und Personenschiffe). Der Bereich, der sich in der Industrie mit der Simulation von herzustellenden Produkten auf der Basis von Differentialgleichungen beschäftigt, fällt in das Gebiet des *Technical Computings*.

Auch in wissenschaftlichen Bereichen ohne industriellen bzw. kommerziellen Hintergrund werden partielle Differentialgleichungen studiert und theoretische Modelle in aufwändigen Computersimulationen analysiert und modifiziert. In der Astrophysik werden bspw. Modelle von Galaxien, Schwarzen Löchern oder ganze Universen mit Hilfe von Computern simuliert und deren Charakteristika untersucht, um die theoretischen Modelle weiter zu verfeinern und zu verbessern. Dieser Bereich, in der das wissenschaftliche Verständnis u.a. durch Simulationsmodelle erweitert wird, wird mit *Scientific Computing* bezeichnet.

Ein Ansatz, Systeme von partielle Differentialgleichungen mit Hilfe eines Computers zu behandeln, ist die Anwendung der sog. *Finite Elemente Methode* (FEM, FE-Methode, s. [JL01, Bra03])[1]. Sehr vereinfacht beschrieben arbeitet die FE-Methode folgendermaßen. Das Simulationsgebiet, auf das die partiellen Differentialgleichungen angewendet werden sollen, wird bei diesem Verfahren in kleine Teilgebiete mit bestimmten mathematischen Eigenschaften aufgeteilt (Diskretisierung in ein Netz). Dies können bspw. Dreiecke oder Rechtecke bei zwei-dimensionalen und Tetraeder bzw. Quader bei drei-

[1] Es existieren noch weitere Verfahren bzw. Methoden (bspw. Finite Volumen Methode), die aber in dieser Arbeit keine weitere Rolle spielen sollen.

dimensionalen Problemstellungen sein. Auch Kombinationen, sog. hybride Netze, sind möglich. Diese geometrischen Objekte werden Elemente genannt. Auf jedem Element werden nun für die Knoten sog. Ansatzfunktionen mit besonderen Eigenschaften definiert, die für die Problemstellung geeignet gewählt sind. Für das gesamte Simulationsgebiet erhält man so eine Menge von Funktionengleichungen, die untereinander in Beziehung stehen und mit deren Hilfe die ursprüngliche Problemstellung der partiellen Differentialgleichungen in diskreter Weise beschrieben bzw. umformuliert werden kann. Jeder Knotenpunkt im Netz entspricht hierbei einer Unbekannten im Gleichungssystem. Die Aufgabe besteht nun darin, für alle Knotenpunkte eine (diskrete) Lösung zu bestimmen, d.h. das aufgestellte Gleichungssystem zu lösen. Die Lösung des Gleichungssystems stellt dann – geeignet interpretiert – eine approximierte Lösung für die ursprüngliche Problemstellung dar.

Die Hauptaufgabe für eine numerische Simulation, die partielle Differentialgleichungen behandelt, ist es also, Gleichungssysteme aufzustellen und geeignet zu lösen. Um eine ausreichende Genauigkeit für die Lösung zu erhalten, gibt es nun mehrere Möglichkeiten eine numerische Behandlung der Problemstellung durchzuführen. Diese können u.a. sein:

- **Hohe Anzahl an Unbekannten in den Gleichungssystemen (feinere Diskretisierung)**: Die Anzahl der Gitterknoten im Netz (und damit die Zahl der Unbekannten) wird entweder im gesamten Netz gleichmäßig (Vollverfeinerung) oder aber in Teilbereichen des Netzes (lokale Verfeinerung) erhöht. Kann die Simulation selbst bestimmen, in welchen geometrischen Bereichen des Netzes die Anzahl der Unbekannten erhöht oder auch erniedrigt wird, so spricht man von lokaler Adaptivität der Simulationsrechnung. Eine höhere Zahl an Unbekannten im Gleichungssystem wird allerdings mit einem Mehraufwand im Lösungsverfahren (mehr Iterationen) erkauft.

- **Komplexe mathematische Verfahren zum Aufstellen der Gleichungssysteme**: Zur Lösung der numerischen Problemstellung können Verfahren mit unterschiedlicher Komplexität herangezogen werden. Komplexität bedeutet z.B. Ordnung eines Verfahrens. Je höher die Ordnung eines Verfahrens desto aufwändiger arbeitet es. Verfahren mit höherer Ordnung werden eingesetzt, wenn besondere Genauigkeit und/oder bessere numerische Stabilität gefordert wird.

- **Kombination beider Möglichkeiten**: Eine Kombination von Anzahlerhöhung der Unbekannten und Verfahren höherer Ordnungen stellt eine sinnvolle Erweiterung dar. Oftmals kann durch eine Kombination eine übermäßige Beanspruchung einer einzelnen Möglichkeit vermindert werden. Z.B. muss bei einer zu simulierenden Problemstellung die Zahl der Unbekannten nicht verdoppelt werden, wenn ein Verfahren benutzt wird, dass eine höhere Ordnung besitzt.

Eine hohe Anzahl von Unbekannten bedeutet anwendungstechnisch eine große Menge an Speicherplatz, um die Netzgeometrie bzw. die assoziierten Matrizen und Vektoren der Gleichungssysteme zu speichern. Komplexe Verfahren benötigen einen hohen Aufwand

an Rechenkapazität. Große Speicherplatz- und Rechenkapazitäten fallen auch heutzutage noch in die Domäne von Supercomputern und Parallelrechensystemen. Daher ist es nicht verwunderlich, dass ein Großteil numerischer Simulationen im industriellen und wissenschaftlichen Bereich auf dieser Architektur von Rechensystemen durchgeführt wird.

Supercomputer bzw. parallele Rechensysteme besitzen eine spezielle Architektur, die sich auf die Entwicklung von Softwareprogramme für solche Architekturen direkt auswirken. Das Speichermanagement basiert für gewöhnlich bei Parallelrechnern auf einem verteilten Ansatz. Entweder greift jede Prozessoreinheit in diesem Modell auf seinen eigenen, lokalen Speicher zu oder eine Gruppe von Prozessoreinheiten (Mehrprozessorsysteme) teilt sich einen Speicher, so dass jede Prozessorgruppe seinen eigenen physikalischen Speicher besitzt. Der Daten- und Informationsaustausch zwischen den Prozessoreinheiten bzw. -gruppen erfolgt bei parallelen Systemen entweder über einen (speziellen) Datenbus oder ein – für diesen Zweck meistens speziell dediziertes – Hochgeschwindigkeitsnetzwerk (vgl. hierzu z.B. [RR00, ALO02]). Bei Netzwerken, die lokale Speicher verbinden, erfolgen Zugriffe bzw. ein Datenaustausch nicht in der Art wie bei einem Mehrprozessorsystem (über ein cachekohärentes Protokoll), sondern über ein eigenes meist nachrichtenbasiertes Kommunikationsprotokoll (z.B. MPI, PVM, o.ä.). Bei der Entwicklung von numerischer Simulationssoftware für parallele Rechensysteme muss also auf die speziellen Bedingungen der Architektur eingegangen werden, um die Systeme effizient und ressourceneffektiv nutzen zu können. Nicht selten wirken sich diese Eigenarten paralleler Architekturen auf wesentliche Teile einer Simulationsumgebung sowohl in programm- als auch modelltechnischer Hinsicht aus. Die Entwicklung eines speziell für den massiv parallelen Einsatz konzipierten Datenmodells und darauf basierender Algorithmen und Datenstrukturen in der FEM-Simulation bilden die Ziele dieser Arbeit.

Ziele der Arbeit

Das Daten- bzw. Objektmodell bildet innerhalb einer FEM-Simulationssoftware das wichtigste Kernmodul. Es verbindet die drei Hauptsäulen einer parallelen numerischen Simulationsumgebung: numerische Behandlung, (lokale) geometrische Adaption sowie Lastverteilung und Migration. In dieser Arbeit wird ein verteiltes Objektmodell vorgestellt, das speziell für den massiv parallelen Einsatz konzipiert ist. Als Grundlage für das Diskretisierungsmodell wird hierbei das Tetraeder verwendet. Ein wichtiges Designmerkmal bei der Modellierung ist die Reduzierung oder sogar Vermeidung von Kommunikation zwischen den auf einzelne Partitionen (Datenaufteilung im parallelen Fall) verteilt arbeitenden Prozessoreinheiten. Dies hat entscheidende Bedeutung auf die Konsistenzwahrung der verteilten Daten- bzw. Objektmengen, die für die numerische Behandlung der Problemstellung unumgänglich ist. Um die Skalierbarkeit für die (massiv) parallele Nutzung durch Datenlokalität zu erhalten, werden in dem verteilten Objektmodell partitionslokale Namensräume für die Objekte eingeführt und

Abbildungsfunktionen für die Konsistenzwahrung definiert. Lokaler Namensraum bedeutet, dass ein geometrisches Objekt der Diskretisierung, also z.B. Knoten, Kanten oder Tetraeder, nur innerhalb einer Partition, die einem Prozessor bzw. einer Prozessorgruppe zugeordnet wird, eindeutig bekannt ist. An Partitionsgrenzen treten daher Mehrdeutigkeiten bei Objekten auf, so dass hier ein effizientes Auflösungsverfahren zur Namensbestimmung benötigt wird.

Jede der drei Hauptsäulen einer parallelen numerischen Simulation arbeitet in seinem eigenen Datenmodell, welches auf Effizienz und Ressourceneffektivität für die spezielle Aufgabe, die es ausführt, ausgelegt ist. Das Numerikmodul arbeitet mit Matrizen und Vektoren, die (geometrische) Adaption verwendet verkettete Datenstrukturen zur platzsparenden Speicherung und schnellen Verwaltung der Netzgeometrie, die Lastverteilung benötigt eine abstrahierte Datensicht auf das verteilte Netz und arbeitet auf einer kompakten Datenrepräsentation zur Migration. Das in dieser Arbeit entwickelte verteilte Objektmodell ist daher derart funktional aufgebaut, dass eine Konvertierung zwischen den verschiedenen Datenmodellen für die drei Simulationsmodule effizient, d.h. zeit- und speicherplatzsparend, durchgeführt werden kann.

Anhand von praxisrelevanten numerischen Gleichungslöseralgorithmen wird das verteilte Objektmodell dieser Arbeit für den massiv parallelen Fall untersucht und seine Effizienz (Skalierung) bewertet. Die geometrische Adaption stellt für das verteilte Objektmodell eine Herausforderung dar, da jede Veränderung der Netzgeometrie einen erheblichen Aufwand für die Objekt- und Datenstrukturverwaltung darstellt. Dies gilt sowohl für die Datenkonsistenzwahrung im parallelen Fall als auch für die speichertechnische Anforderung des Datenmodells. In dieser Arbeit wird daher anhand des Verfahrens zur irregulären Adaption (Rot-Grün Verfeinerung) ein konsistenzerhaltender, paralleler Algorithmus präsentiert, der im verteilten Objektmodell speziell für den massiv parallelen Einsatz optimiert ist und ein gutes Skalierungsverhalten besitzt. Die Eigenschaften dieses Adaptionsalgorithmus werden analysiert und eine reale Implementierung durch Messungen praxisrelevant bewertet. Neben der Adaption verändert auch die Lastverteilung bzw. die Datenmigration, die auf dem Ergebnis der Lastverteilung operiert, ebenfalls durch Modifizierung, das Datenmodell der Netzgeometrie. Daher erfolgt auch eine Untersuchung des verteilten Objektmodells im Hinblick auf Verschiebung großer Datenmengen zwecks Lastausgleich. Bei dieser Untersuchung interessiert in dieser Arbeit weniger das Verfahren der Lastausgleichsberechnung, als viel mehr die notwendigen Schritte, um die Konsistenz des Datenmodells nach einer Migration von Objekten der Netzgeometrie wieder herzustellen. Hierzu wird ein parallel arbeitender Algorithmus zur Migration präsentiert, der genauso wie der Adaptionsalgorithmus für den massiv parallelen Fall entwickelt wurde und gut skaliert. Dies wird ebenfalls durch praktische Messungen belegt.

Ein weiteres Ziel in dieser Arbeit stellt die Untersuchung zur Verbesserung der irregulären Adaption durch Erweiterung des verwendeten Regelsatzes dar. Der in der Literatur bekannte (reduzierte) Regelsatz erzeugt, abhängig von der Ausgangsverteilung der Tetraederqualitäten in der Netzgeometrie, in der Praxis u.U. Teiltetraeder in Abschlüssen,

die ungünstige, d.h. sehr kleine, Raumwinkel aufweisen. Da die Konditionszahl einer zur Diskretisierung assoziierten Matrix direkt vom kleinsten Winkel im Netz abhängt und somit das Konvergenzverhalten eines Gleichungslösers beeinflusst, stellt eine Verbesserung der Raumwinkel bei Abschlüssen auch eine Verbesserung für das Lösen eines Gleichungssystems bzw. des verwendeten Verfahrens dar. Aus diesem Grund wird in dieser Arbeit ein erweiterter Regelsatz zur irregulären Adaption präsentiert, der die Form eines Ausgangstetraeders vor Anwendung einer Abschlussregel berücksichtigt. Es wird gezeigt, dass durch wenige Änderungen im verteilten Objektmodell bzw. in dem präsentierten Adaptionsalgorithmus dieser erweiterte Regelsatz verwendet werden kann. Anhand einer praktischen Anwendung werden die Auswirkungen und Eigenschaften des erweiterten Regelsatzes unter Berücksichtigung verschiedener Qualitätsmetriken von Tetraedern aufgezeigt und bewertet.

Die in dieser Arbeit entwickelten Algorithmen und Datenstrukturen sowie das verteilte und spezialisierte Objektmodell bilden die Grundlage für die massiv parallel arbeitenden numerische Simulationsumgebung *padfem*^{2} (vgl. z.B. [BKM03, BM05a, BM06, BM07]). *padfem*^{2} wurde im Kontext des Sonderforschungsbereichs SFB 376 *Massive Parallelität*, Teilprojekt A3, am *Paderborn Center for Parallel Computing* (PC^2) der Universität Paderborn entwickelt. Die Hauptanwendungsbereiche des *padfem*^{2}-Projektes liegen in der numerischen Untersuchung strömungsmechanischer Problemstellungen sowie dem Studium von Lastverteilungsverfahren, skalierenden Adaptionsmethoden und verteilten Datenmodellen für den massiv parallelen Einsatz.

Aufbau der Arbeit

Diese Arbeit gliedert sich in die folgenden Kapitel:

Im Kapitel *Grundlagen* erfolgt eine Einführung über den generellen, technischen Aufbau einer parallelen numerischen Simulationsumgebung. Ausgehend vom Kreislauf innerhalb einer numerischen Simulation werden die drei Hauptmodule – Numerikmodul, Adaptionsmodul und Partitionierungs- bzw. Lastverteilungsmodul – in ihren Grundzügen und Funktionalitäten vorgestellt. Dabei geben die einzelnen Abschnitte, in denen die Module aufgezeigt werden, einen Überblick über die klassischen Verfahren und Strategien nebst Verweisen auf weiterführende Literatur. Da das Tetraeder im $\mathbb{R}^3$ die Grundlage für Diskretisierungen eines Simulationsgebietes darstellt, erfolgt eine Übersicht über benötigte mathematische Eigenschaften des Tetraeders. Hierbei wird zudem besonderer Wert auf Qualitätseigenschaften des allgemeinen Tetraeders gelegt, da diese die Grundlage für die in dieser Arbeit entwickelte Formadaptivität liefert. Um eine Grundlage für die Bewertung von parallelen Algorithmen und Rechensystemen zu haben, werden die wichtigsten Leistungsmaße in diesem Bereich definiert bzw. aufgezeigt. Abschließend erfolgt im Grundlagenkapitel ein Überblick über die wichtigsten Projekte im Bereich der parallelen numerischen Simulation. Hierbei wird nach Projekten aus dem akademischen und kommerziellen Umfeld unterschieden. Zudem werden die

bekanntesten Middleware-Softwarepakete, die eine Unterstützung bei der Entwicklung von Simulationsumgebungen bieten, vorgestellt.

Das Kapitel *Parallele Simulation mit partitionslokalen Objektnamensräumen* stellt den Hauptteil dieser Arbeit dar. Zu Beginn des Kapitels wird das in dieser Arbeit entwickelte, verteilte Objektmodell eingeführt und formal definiert. Hierbei wird detailliert auf die geforderten Eigenschaften (s.o.) eingegangen und eine mögliche Realisierung für eine Implementation (Datenstruktur und Operationen) für den parallelen Fall angeboten. Aufbauend auf dem verteilten Objektmodell werden die drei Hauptmodule einer parallelen Simulationsumgebung anhand konkreter Verfahren (Löser, Adaption, Lastverteilung) näher betrachtet. Dabei wird besonders auf die Objektverwaltung und die Transformationen zwischen den Datenmodellen der einzelnen Module Wert gelegt, da hier in (parallelen) Simulationsumgebungen für gewöhnlich der größte Leistungsengpass vorliegt. Neben den benötigten Datenstrukturen für das Objektmodell spielen auch die Algorithmen der Module und ihr Zusammenspiel eine wichtige Rolle. Bei numerischen Algorithmen (Löser und Abgleicher), die den Hauptteil der Simulationslaufzeit einnehmen, ist die Skalierbarkeit mit der Datenmenge (Matrixgröße) und der Zahl der verwendeten Prozessoren von entscheidender Bedeutung. Hierauf wird anhand dreier Gleichungslöser-Verfahren näher eingegangen. Durch lokale (geometrische) Adaption entsteht ein erheblicher und komplexer Verwaltungsaufwand für die Datenstrukturen des Objektmodells, das im parallelen Fall noch zusätzliche Komplexität wegen der Konformitäts- und Konsistenzbedingung der verteilten Diskretisierung erzwingt. In diesem Kapitel der Arbeit wird daher ein Algorithmus zur irregulären 3D-Adaption (Rot-Grün Adaption) vorgestellt, der speziell für den massiv parallelen Ansatz entwickelt wurde und mit dem verteilten Objektmodell korrespondiert. Zusätzlich wird eine Erweiterung des Regelsatzes zur irregulären Adaption eingeführt, die eine qualitätsgebundene Adaption bzgl. der Tetraederform ermöglicht. Neben der Adaption stellt die Lastverteilung und insbesondere die Datenmigration innerhalb einer parallelen Simulationsumgebung ein weiteres Modul dar, das einen stark modifizierenden Einfluss auf das Objektmodell und dessen Verwaltung hat. Aus diesem Grund wird abschließend im dritten Kapitel ein effizienter, skalierender Algorithmus zur Migration von Tetraedern und zur Rekonstruktion von verschobenen Partitionsgrenzen basierend auf dem verteilten Objektmodell vorgestellt.

Im Kapitel *Anwendung und Leistungsanalyse* wird einführend die Simulationsumgebung *padfem*^{2} vorgestellt, in der sämtliche Algorithmen und Datenstrukturen für das verteilte Objektmodell, die in dieser Arbeit vorgestellt werden, implementiert sind. *padfem*^{2} stellt aufgrund seiner Architektur und Konzepte einen erfolgreichen Ansatz für den Einsatz massiver Parallelität im Bereich der numerischen FEM-Simulation dar. Um das Leistungspotential zu zeigen, erfolgt in diesem Kapitel außerdem eine detaillierte Analyse der drei Hauptmodule einer parallelen numerischen Simulation anhand von typischen (Abschnitt Gleichungslöser) oder einzelne Aspekte besonders fordernde (Abschnitte Adaption und Lastverteilung) Problemstellungen. Abschließend verdeutlicht ein Beispiel einer realen Simulation (strömungsmechanische Untersuchung) die

Leistungsfähigkeit des verteilten Objektmodells und seiner Algorithmen und Datenstrukturen in einer praxisrelevanten Anwendung.

Das letzte Kapitel dieser Arbeit stellt schließlich eine bewertende Zusammenfassung vor.

Publikationen

Teile dieser Arbeit sowie die Anwendung von Konzepten, Algorithmen und Datenstrukturen der massiv parallelen *padfem*^{2}-Simulationsumgebung wurden auf folgenden Konferenzen bzw. in Publikationen veröffentlicht: *European PVM/MPI User's Group Meeting (EuroPVM/MPI)* 2003 [BKM03], *International Conference on Parallel and Distributed Computing and Systems (PDCS)* 2003 [BM03], *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* 2004 [BKM04b], *International Conference on Imaging Science, Systems, and Technology (CISST)* 2004 [BKM04a], *International Conference on Parallel and Distributed Computing and Networks (PDCN)* 2005 [BM05b], *Symposium on Simulationstechniques (ASIM)* 2005 [BM05a], *International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)* 2007 [BM07].

2 Grundlagen

Inhalt

2.1 Die 3 Säulen einer parallelen numerischen Simulation

Eine Simulationsumgebung für numerische Problemstellungen besteht in der Regel aus verschiedenen Modulen, die über einen Kommunikationsmechanismus Daten untereinander austauschen. Für die Mathematik existiert ein Modul, das die numerische Behandlung der Problemstellung, die durch die Simulation untersucht werden soll, durchführt. In diesem Numerikmodul sind verschiedene Verfahren bereitgestellt, die Teilaspekte zur Lösungsstrategie realisieren oder unterstützen. Dazu zählen z.B. im Bereich der FE-Lösungsverfahren die Gleichungssystemlöser, Hilfsfunktionen zur Problemaufstellung und -durchführung, Diskretisierung des Simulationsgebietes, Fehlerschätzer usw. Ein weiteres Modul beschäftigt sich mit der geometrischen Modifikation des Simulationsgebietes. Dies ist das Adaptionsmodul. Numerik- und Adaptionsmodul sind in jeder Simulationsumgebung vorhanden, die lokal adaptiv rechnet. Bietet die Simulationsumgebung auch eine parallele Anwendungsmöglichkeit, so ist noch ein weiteres Modul erforderlich. Hierbei handelt es sich um das Lastverteilungsmodul, das ein oder mehrere Verfahren anbietet, um einen Lastausgleich zwischen den mehrfach vorhandenen Recheneinheiten (Prozessoren, Rechenknoten) zu ermitteln. Durch Datenmigration zwischen den Recheneinheiten erfolgt dann der Ausgleich.

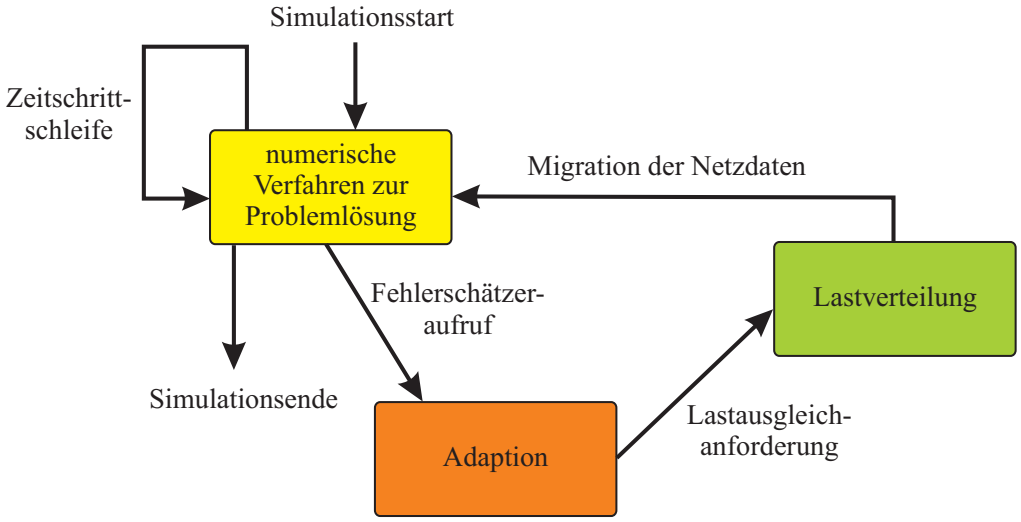
2.1.1 Der Simulationskreislauf

Abbildung 2.1 zeigt die drei wichtigen Module einer parallelen numerischen Simulation und ihre Interaktion untereinander in einem Flussdiagramm. Die Pfeile geben den typischen Verlauf und die Kommunikationspfade zwischen den Modulen an. Die Simulation einer Problemstellung beginnt im Numerikmodul. Zum Starten benötigt der Mathematikteil der Simulation Eingabeparameter, die üblicherweise aus einer Gebietsbeschreibung bzw. -diskretisierung sowie Rand- und Anfangsbedingungen besteht. Hinzu kommen noch weitere Parameter, die die Problemstellung aus anwendungstechnischer Sicht genauer spezifizieren[1].

Das Numerikmodul führt nun innerhalb einer Schleife auf der Basis der Eingabeparameter Berechnungen für einen einzelnen Zeitschritt aus. Ein Zeitschritt der Simulation besteht dabei aus mehreren Teilen, die abhängig von der Art der zu lösenden mathematischen Problemstellung sind. Bei strömungsmechanischen Problemstellungen, wie bspw. die Anwendung der Navier-Stokes Gleichungen, müssen der Transport von Partikeln (Charakteristikenverfahren) und die Diffusion des Fluids berechnet werden (s. z.B. [BM06, BM07]). Grundsätzlich werden aber zum größten Teil Gleichungssysteme aus der Diskretisierung des Gebietes und den zu berechnenden Teilproblemen aufgestellt und gelöst. Das Lösen dieser Gleichungssysteme benötigt gewöhnlich die meiste Zeit des Rechenschrittes bzw. des gesamten Simulationslaufs. Um diesen Zeitaufwand zu reduzieren, werden neben geeignet gewählten Gleichungslöseralgorithmen (z.B. vorkonditionierte Krylov-Unterraum-Methoden, Mehrgitter-Verfahren usw.) parallele Ansätze

[1] Dies können z.B. im Falle einer Strömungsberechnung physikalische Parameter des Fluids o.ä. sein.

Abbildung 2.1 Flussdiagramm einer numerischen Simulation



verfolgt (s.a. Abschnitt 2.1.2). Die parallelen Verfahren bedienen sich dabei Spezialisierungen bzw. Performanceoptimierungen, die von Berechnungen auf einem einzelnen Mehrprozessorrechner mit Hilfe von Threads bis zur massiv parallelen Anwendung mit mehreren hundert oder gar tausenden Rechenknoten reichen. Parallele Ansätze werden in der Praxis auch dann verfolgt, wenn die Datenmenge nicht mehr auf einen einzelnen Rechner abgebildet werden kann und deshalb auf mehrere Rechenknoten verteilt werden müssen.

Hat das Numerikmodul mehrere Zeitschritte berechnet, findet eine Überprüfung der Lösungsqualität statt. Dazu wird mit Hilfe eines Fehlerschätzers oder -indikators ein Maß für die Abweichung der bisher errechneten Lösung pro Knoten- oder Volumenelement bestimmt (s. z.B. [RB03, BD03, Aki05]). Finden sich Stellen in der Diskretisierung des Simulationsgebietes, die eine ausreichende oder eine zu grobe Lösungsqualität besitzen, so erfolgt eine Adaptionsphase. Bei ausreichender Lösungsqualität wird, im Falle einer geometrischen Adaption, eine lokale Vergröberung durchgeführt, bei zu grober Lösungsqualität findet eine lokale Verfeinerung der Diskretisierung statt (s.a. Abschnitt 2.1.3). Eine Adaption kann, je nach Problemstellung und gewünschter Lösungsqualität, nach jedem berechneten Zeitschritt stattfinden oder aber wiederholt im Abstand eines vorgegebenen Simulationszeitraumes. Die Parallelisierung eines Adaptionsverfahrens stellt eine Herausforderung dar, da gerade bei partitionierten Diskretisierungen an den Gebietsgrenzen Schwierigkeiten auftreten können. Aus Anwendungssicht ist es im parallelen Fall an Partitionsgrenzen aufwändig, eine Interpolation der Lösung eines Gitters auf ein lokal adaptiertes Gitter zu übertragen. Hierzu wird Kommunikation der benachbarten Partitionen benötigt. Aus geometrischer Sicht stellen die ausbreitenden Adaptionsfronten (zur Konformitätswahrung der Diskretisierung) über Partitionsgrenzen hinweg ebenfalls einen zusätzlichen Kommunikationsoverhead dar. Eine effiziente

Umsetzung des verwendeten Adaptionsverfahrens und die Reduzierung der Kommunikationsphase sind daher notwendige Optimierungsziele, um die Gesamtlaufzeit der Simulation möglichst gering zu halten.

Im parallelen Fall entsteht üblicherweise durch die Anwendung der Adaption ein Ungleichgewicht in der Anzahl der Volumen pro Partition. Die Partitionen werden in der Praxis meistens jeweils auf einen Rechenknoten oder eine CPU abgebildet. Die Folge des Ungleichgewichts ist eine unterschiedliche Berechnungsdauer für eine Iteration der Gleichungslöseralgorithmen auf den einzelnen Rechenknoten. Es entstehen Wartezeiten, in denen Rechenknoten während Kommunikationsphasen auf andere Rechenknoten warten müssen, da deren Iteration noch nicht fertig berechnet wurde. Um diese Wartezeiten zu reduzieren bzw. zu vermeiden, findet im Lastverteilungsmodul eine Berechnung eines Ausgleiches für die Anzahl der Volumen über alle Partitionen statt. Es existieren dazu Verfahren mit unterschiedlichen Optimierungszielen (s.a. Abschnitt 2.1.4). Die Berechnung eines Ausgleiches besteht aus den Informationen, welches Volumen einer Partition zu welcher Zielpartition migriert werden soll. Ein Ausgleich der Volumenzahl findet aus Performancegründen nach jeder Adaptionsphase statt. Kann der Ausgleich allein aus den Informationen des Fehlerschätzers bestimmt werden, d.h. die Ausbreitung der geometrischen Modifikationen kann vor ihrer Durchführung berechnet werden, so ist die Lastverteilungsphase auch vor der Adaptionsphase sinnvoll, da dann sowohl für die Berechnung des Ausgleiches als auch die Migration weniger Daten betrachtet werden müssen. Nach der Migration von Teilen der Diskretisierung startet die Berechnung des nächsten Schrittes wieder im Numerikmodul.

Der Kreislauf aus numerischer Problemlösung, Adaption und, im parallelen Fall, Lastverteilung endet, wenn eine vorgegebene Anzahl von Zeitschritten berechnet wurde (bei zeitlich abhängigen Problemstellungen) oder die Lösungsqualität (bestimmt durch einen Fehlerschätzer) eine vorgegebene Schranke unterschreitet.

2.1.2 Parallele Numerik

Der größte Aufwand während der numerischen Behandlung von FEM-Problemstellungen wird für die Lösung von linearen bzw. nicht-linearen Gleichungssystemen aufgewendet (vgl. hierzu [ALO02, JL01, Bra03, BJ93, GO96, BBC+94]). Durch Anwendung von parallelen Techniken wird das Lösen dieser Systeme stark beschleunigt bzw. wegen des großen Datenumfanges erst möglich gemacht. In diesem Abschnitt erfolgt ein kurzer Überblick über die klassischen parallelen Techniken, die hierfür angewendet werden.

Die Mathematik einer parallelen numerischen Simulation umfasst noch weitere Verfahren neben dem Lösen von linearen Gleichungssystemen (z.B. Partikelsuchmethoden, s.a. Abschnitt 2.1.1, [BM07, BM06, BM05a]). Diese sind aber zumeist problemspezifisch und sollen hier nicht weiter betrachtet werden.

Parallele Gleichungslöser

Für das Lösen eines linearen Gleichungssystems $Ax = b$ existieren eine Reihe von Verfahrensklassen. Hierzu zählen z.B. die Klasse der *direkten Verfahren*, der *Splitting-Verfahren*, der *Krylov-Unterraum Methoden* und der *Mehrgitter-Verfahren* (s. z.B. [Saa03, Hac93, Mei05, GL96, Bey98, SK04]).

In den folgenden Abschnitten sei $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ eine nicht-singuläre Matrix, $x = (x_i) \in \mathbb{R}^n$ der zu bestimmende Unbekannten- bzw. Lösungsvektor und $b = (b_i) \in \mathbb{R}^n$ die rechte Seite des linearen Gleichungssystems.

Direkte Verfahren Zu den direkten Verfahren zählt z.B. die klassische Gauß-Elimination bzw. der Gauß-Algorithmus. Die Grundlage des Algorithmus besteht aus den folgenden Operationen, die durch Multiplikation mit geeignet gewählten Elementarmatrizen durchgeführt werden: (a) Vertauschung von Zeilen, (b) Multiplikation einer ganzen Zeile mit einer Zahl $\neq 0$, (c) Addition eines Vielfachen einer Zeile zu einer anderen. Durch Anwenden dieser Operationen (a)-(c) auf die Matrix A zusammen mit der rechten Seite b soll A in eine Stufenform A' (obere Dreiecksmatrix) gebracht werden. Eine Pivotstrategie zur Bestimmung des betragsgößten Elementes (über Zeile, Spalte oder gesamte Restmatrix) erhöht zusätzlich die numerische Stabilität während der Matrixtransformation. Mittels Rücksubstitution können dann die einzelnen Komponenten x_i des Lösungsvektors aus der Stufenmatrix A' bestimmt werden.

Der Gauß-Algorithmus steht in enger Beziehung zur Bestimmung einer LU-Zerlegung von A. Hierbei wird das System $Ax = b$ sukzessiv in ein gleichwertiges System $LUx = b$ umgeformt. Anschließend wird durch Vorwärtssubstitution das System $Ly = b$ und danach mittels Rücksubstitution aus $Ux = y$ der Lösungsvektor bestimmt. Die LU-Zerlegung ist von Vorteil, wenn das Gleichungssystem für mehrere rechte Seiten b gelöst werden muss, da die Zerlegung dann nur einmal bestimmt werden muss[2].

Die hier vorgestellte Parallelisierung der Gauß-Elimination basiert auf einer zeilenzyklischen Zuordnung (s. z.B. [ALO02, RR00]) der Matrix auf die Prozessoren bzw. Rechenknoten. Ein Prozessor P_i speichert also die Zeilen $i, i+p, i+2p, \ldots$, wenn p Prozessoren bzw. Rechenknoten zur Verfügung stehen. Die Zerlegungsberechnung (Bestimmung von L und U) auf einem Prozessor gliedert sich nun in die folgenden Phasen:

- **Bestimmung des globalen Pivotelements:** Jeder Prozessor bestimmt in der zu betrachtenden Matrixspalte sein lokales, betragsgrößtes Element und kommuniziert mit allen anderen, um das globale, betragsgrößte Element zu identifizieren.
- **Austausch und Verteilung der Pivotzeile:** Der Prozessor mit dem global betragsgrößten Element tauscht seine komplette Zeile mit dem Prozessor aus, der die aktuell im Gauß-Algorithmus zu betrachtende Zeile hält. Danach erfolgt eine Verteilung der

[2] Die LU-Zerlegung spielt auch eine wichtige Rolle bei vorkonditionierten Krylov-Unterraum-Methoden, bei denen je Iterationsschritt ein Gleichungssystem $Mr = z$ mit $M = LU$ und variabler rechter Seite z gelöst werden muss.

Pivotzeile an alle anderen Prozessoren, damit diese lokal ihre Eliminationsschritte durchführen können.

- **Berechnung der Eliminationsfaktoren und Matrixelemente:** Die Prozessoren berechnen lokal anhand der empfangenen Zeile ihre Einträge für die Matrizen L (Eliminationsfaktormatrix) und U (Stufenmatrix) sowie b.

Nach der Durchführung dieser Phasen für alle Zeilen ist die Matrix A in L und U zerlegt. Es fehlt noch die Rückwärtssubstitution mit Hilfe der transformierten rechten Seite b. Aufgrund der zeilenzyklischen Verteilung der Matrix kann diese Phase nur sequenzialisiert durchgeführt werden. Ein Prozessor, der eine lokale Komponente des Lösungsvektors x aus b bestimmt, muss diese an die anderen verteilen, bevor andere Komponenten ausgerechnet werden können.

Eine alternative Parallelisierungsstrategie liegt in einer anderen Datenverteilung der Matrix A. Hierbei werden Zeilen und Spalten blockweise zyklisch auf die Prozessoren bzw. Rechenknoten verteilt (gesamtzyklisch, [RR00]). Der Kommunikations- und Verwaltungsaufwand für die Durchführung des Gauß-Algorithmus ist bei dieser Methode weitaus größer. Jedoch lässt sich dadurch ein besserer Parallelisierungsgrad erreichen, da die Rückwärtssubstitution besser parallel ausgeführt werden kann.

Ist A eine symmetrische, positiv definite Matrix, so kann das artverwandte Cholesky-Verfahren angewendet werden, um das lineare Gleichungssystem direkt zu lösen. Es lässt sich hinsichtlich Zeit- und Speicheraufwand günstiger implementieren. Die Parallelisierung dieses Verfahrens gestaltet sich in ähnlicher Weise wie für das klassische Gauß-Eliminationsverfahren. Ist die Matrix A zusätzlich noch dünn besetzt, was bei Matrizen der Fall ist, die aus FEM-Diskretisierungen gebildet werden, so existieren Re-Ordering-Verfahren mit besonderen Pivotsstrategien, um den *Fill-in*[3] zu verringern (s. z.B. [Sch00]). Aus speichertechnischen Gründen (komprimierte Speicherung dünn besetzter Matrizen) sind solche Verfahren vorzuziehen.

Splitting-Verfahren Splitting-Verfahren berechnen im Gegensatz zu den direkten Verfahren durch mehrfache Iteration einen Lösungsvektor für das lineare Gleichungssystem. Sie basieren auf einer Zerlegung der Koeffizientenmatrix $A = B + (A - B), B \in \mathbb{C}^{n\times n}$, wodurch das System $Ax = b$ transformiert werden kann in $x = B^{-1}(B-A)x+B^{-1}b$, d.h. in ein lineares Iterationsverfahren $x^{(k+1)} = \phi(x^{(k)}, b) = Mx^{(k)} + Nb$ für $k = 0, 1, 2, \ldots$ (vgl. z.B. [Mei05]). Zu dieser Klasse von Verfahren zählen z.B. das Jacobi-Verfahren (Gesamtschritt), das Gauß-Seidel-Verfahren (Einzelschritt) sowie das SOR-Verfahren (successive over-relaxation).

Die Berechnungsvorschrift im Jacobi-Verfahren für einen Lösungsvektor $x^{(k)}$ in der k-

[3] Null-Einträge der Matrix werden während des Algorithmus mit von Null verschiedenen Einträgen belegt.

ten Iteration lautet

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k-1)} \right), \; 1 \leq i \leq n. \tag{2.1}$$

Es ist ersichtlich, dass für die Berechnung der i-ten Komponente von $x^{(k)}$ nur Werte aus der vorherigen $(k-1)$-ten Iteration benötigt werden. Die Berechnung kann also für jede Komponente in einer Iteration (trivial) parallel ausgeführt werden, da die Daten unabhängig voneinander sind. Ist die Koeffizientenmatrix A und Lösungsvektor x auf mehrere Prozessoren bzw. Rechenknoten aufgeteilt, müssen vor der Berechnung die benötigten Daten kommuniziert werden.

Für das Gauß-Seidel-Verfahren und das SOR-Verfahren ist die Parallelisierung nicht so natürlich zu realisieren wie beim Jacobi-Verfahren, da für einen Berechnungsschritt einer Lösungsvektorkomponente $x_i^{(k)}$ eine Abhängigkeit der Daten sowohl zum vorherigen $(k-1)$-ten Iterationschritt als auch zur aktuellen Iteration besteht. Die Berechnungsvorschrift für das SOR-Verfahren lautet

$$x_i^{(k)} = (1-\omega) x_i^{(k-1)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right), \; 1 \leq i \leq n, \tag{2.2}$$

mit einem reellen Relaxationparameter $\omega \in]0, 2[$. Für $\omega = 1$ ergibt sich das klassische Gauß-Seidel-Verfahren.

Das Prinzip der Parallelisierung besteht nun darin, eine Transformation der Matrix A in eine äquivalente Matrix A' durchzuführen, in dem die Nummerierung der Unbekannten in der Diskretisierung bzw. des Lösungsvektors durch einen Umordnungsalgorithmus geändert wird. Diese Umordnung führt zu einer Segmentierung der Unbekannten in Mengen, die voneinander unabhägig und damit parallel berechnet werden können. Effiziente Umordnungsverfahren basieren auf Algorithmen für k-Färbungen von Graphen (s. z.B. [Saa03, ALO02, RR00, Bra94]) zur Bestimmung unabhängiger Knotenmengen. Ein bekanntes Verfahren für strukturierte Gitter und die daraus entstehenden Matrizen mit regelmäßiger Bandstruktur ist z.B. die Rot-Schwarz Nummerierung (Schachbrett-Ordnung), welches sowohl für 2D- als auch 3D-Probleme einfach zu implementieren ist (s. [ALO02]).

Die transformierte Matrix A' besitzt gemäß der Segmentierung eine besondere Form, wenn die Unbekannten pro Farbe sequenziell durchnummeriert werden (vgl. hierzu [Saa03, GL96, ALO02]). Wird mit $\mathrm{col}(A)$ die Anzahl der Farben bezeichnet, mit der die Diskretisierung aus n Knoten (entspricht n Zeilen) gefärbt wird, dann besteht die Struktur von A' aus einer Menge von Diagonalmatrixblöcken D_i, $1 \leq i \leq \mathrm{col}(A) := k$, sowie strikter oberer Dreiecksmatrix U (bestehend aus U_{ij}-Blöcken, $i < j$) und strikter

unterer Dreiecksmatrix L (bestehend aus L_{ij}-Blöcken, $i > j$):

$$Ax = b \hookrightarrow A'x' = b' \Leftrightarrow \begin{pmatrix} D_1 & U_{12} & U_{13} & \cdots & U_{1k} \\ L_{21} & D_2 & U_{23} & \cdots & U_{2k} \\ \vdots & \vdots & \ddots & & \vdots \\ L_{k1} & L_{k2} & L_{k3} & \cdots & D_k \end{pmatrix} \cdot \begin{pmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_k \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_k \end{pmatrix}. \quad (2.3)$$

Die Zeilen, die zu einem Diagonalmatrixblock D_i assoziiert sind, besitzen untereinander keine Abhängigkeit und sind daher parallel berechenbar. Sinnvollerweise sollte $\mathrm{col}(A) = \chi(A)$ (*chromatische Zahl*, s. z.B. [Bra94]) gelten, um die Anzahl der Zeilen für die Farben zu maximieren. In der Praxis wird jedoch aus Geschwindigkeitsgründen häufig ein Greedy-Algorithmus zur Bestimmung der Farbanzahl $\mathrm{col}(A)$ verwendet, da die Bestimmung von $\chi(G)$ für einen allgemeinen Graphen G *NP-schwer* ist. Ein paralleler Algorithmus bestimmt nun anhand der Umordnung für die Komponententeilmenge $x'_i \subset x'$ in der k-*ten Iteration* parallel eine Lösung (per Vorwärtssubstitution und berechneten Werten aus der $(k-1)$-ten Iteration, Gleichung 2.2) und nutzt diese dann zusammen mit den bisher berechneten Teilmengen $\{x'_1, \ldots, x'_{i-1}\}$, um x'_{i+1} zu bestimmen. Durch Zusammentragen der Komponenten (Kommunikation) und Rücktransformation von x' erhält man die Lösung des Systems $Ax = b$ in der k-ten Iteration.

Krylov-Unterraum Methoden Krylov-Unterraum Methoden zählen zu den modernen Verfahren, die besonders im Hinblick auf große (dünn besetzte) Matrizen entwickelt wurden. Verfahren aus dieser Klasse berechnen iterativ aus einer gegebenen Startlösung $x^{(0)}$ für ein Gleichungssystem $Ax = b$ (meistens $x^{(0)} = 0$ oder $x^{(0)} = b$) eine Lösung $x^{(k)}$, die bis auf einen Fehler $\|x^{(k)} - x^*\| = \epsilon_k$ (zur analytischen Lösung, falls gegeben) bzw. $\|x^{(k)} - x^{(k-1)}\| = \epsilon_k$ genau ist. Für die Konstruktion von $x^{(k)}$ verwenden Krylov-Unterraum Methoden orthogonalisierte Residuenvektoren, so dass $x^{(k)} = x^{(0)} + K_k$ mit $K_k = K_k(A, r^{(0)}) = \mathrm{span}(r^{(0)}, Ar^{(0)}, \ldots, A^{k-1}r^{(0)})$ und $r^{(0)} = b - Ax^{(0)}$ gilt.

Die Verfahren benutzen pro Iteration eine Menge von Grundoperationen, um eine Näherung zu berechnen. Dazu zählen hauptsächlich die Berechnung von (a) Vektoraktualisierungen $y = \alpha x + \beta$ (sog. DAXPY-Operationen), (b) Skalarprodukten $\langle x, y \rangle$ und (c) Matrix-Vektor-Multiplikationen $r = Mz$. Die Operationen (a)-(c) werden auch als BLAS-Operationen (*Basic Linear Algebra Subroutines*) bezeichnet. Durch die Anwendung der Vorkonditionierungstechnik kann i.A. ein Verfahren aus der Klasse der Krylov-Unterraum Methoden dahingehend verbessert werden, dass es weitaus weniger Iterationen benötigt als ein gewöhnliches Verfahren, um eine gleichwertige Lösung für das System $Ax = b$ zu bestimmen. Durch diese Technik kommen zu den Operationen (a)-(c) noch zusätzlich die Operationen (d) Bestimmung einer geeigneten Konditionierungsmatrix $\hat{A}$ (löse für $Ax = b$ transformiertes System $\hat{A}\hat{x} = \hat{b}$ mit $\hat{A} = SAS^T, \hat{x} = (S^{-1})^T x, \hat{b} = Sb$)[4] und (e) Anwendung der Vorkonditionierung inner-

[4] Idealerweise sollte S so gewählt sein, dass die Konditionszahl von $\hat{A}$ (wesentlich) kleiner als die von A ist und die im Verfahren verwendeten Operationen mit S (bspw. Berechnung von S^{-1} und S^T) müssen zeitlich schnell erfolgen, um einen Geschwindigkeitsgewinn des komplexeren Löserverfahrens zu erreichen (vgl. bspw. [ALO02, Mei05]).

halb jeder Iteration des Verfahrens (bspw. durch Vorwärts- und Rückwärtssubstitution). Zu der Klasse der Krylov-Unterraum Methoden zählen z.B. das CG-Verfahren, das BiCGStab-Verfahren und das GMRES-Verfahren (zu Verfahrensdetails s. z.B. [Hac93, GL96, Saa03, Mei05, She94]). Die Entscheidung, welches Verfahren auf das System $Ax = b$ angewendet wird, hängt i.A. von speziellen Eigenschaften der Koeffizientenmatrix A (z.B. Definitheit, Symmetrie, Eigenwerte) und dem zur Verfügung stehenden Speicherplatz (vgl. hierzu [BBC+94]) ab.

Die Parallelisierung der BLAS-Operationen (a)-(c) gestaltet sich relativ einfach. Die Vektoraktualisierung kann wegen der Datenunabhägigkeit der Vektorkomponenten trivial parallel ausgeführt werden (entsprechend der Datenverteilung der Komponenten auf die Prozessoren bzw. Rechenknoten). Ein Skalarprodukt $\langle x, y\rangle := \sqrt{\sum_{i=1}^{n} x_i \cdot y_i}$ beinhaltet eine globale Datenabhängigkeit, d.h. das Ergebnis liegt erst vor, wenn für die Operation alle Teildaten vorliegen. Eine Parallelisierungsstrategie hierfür basiert auf der lokalen Summierung der Produkte von prozessor- bzw. rechenknoteneigener Vektorkomponenten, Bildung der Gesamtsumme durch einen Prozessor bzw. einer Fan-in-Methode (vgl. [ALO02]), Wurzelbildung auf einem ausgezeichnneten Prozessor und Verteilung des Ergebnisses an alle anderen Recheneinheiten.

Die Strategie für einen parallelen Algorithmus für die Matrix-Vektor-Multiplikation hängt stark von der Datenverteilung der Matrix und des Vektors ab (s. [ALO02]). Hier sind drei klassische Verteilungen möglich: Blockzeilen, Blockspalten und zyklische Blockung. In der Blockzeilenverteilung erhält jeder Prozessor bzw. Rechenknoten eine Menge von aufeinander folgenden Zeilen zugeordnet, entsprechend bei der Blockspaltenverteilung aufeinander folgende Spalten der Matrix. Für die Blockzeilenverteilung liegt der Vektor repliziert auf allen Prozessoren vor, so dass die Einträge im Ergebnisvektor ohne Kommunikation berechnet werden können, d.h. die Matrix-Vektor-Multiplikation ist in dieser Datenverteilung trivial parallelisierbar, da keine Datenabhängigkeit bei den Operationen vorliegt. Sind die Spalten der Matrix in Blöcke den Prozessoren bzw. Rechenknoten zugeordnet, ist es erforderlich, den Vektor ebenfalls in Blöcke zu zerteilen, so dass die einem Prozessor zugeordneten Zeilenstücke mit den zugehörigen Vektorstücken zu Teilergebnissen zusammengetragen werden können. Eine Replizierung des Vektors ist hier nicht nötig. Durch Kommunikation (Kollektiv-/Reduce-Operationen) werden die Einträge für den Ergebnisvektor dann zusammengetragen. Bei der dritten Verteilungsmöglichkeit sind die Matrix und der Vektor in möglichst gleichgroße, rechteckige Blöcke zyklisch den Prozessoren bzw. Rechenknoten zugeordnet. Auch hier werden zugehörige Zeilen-/Spaltenstücke der Matrix und Vektorstücke zu Teilergebnissen zusammengeführt. Diese werden dann zyklisch zwischen den Prozessoren ausgetauscht und miteinander verknüpft, so dass am Ende für den Ergebnisvektor die Daten komplett vorhanden sind (vgl. [RR00, ALO02])). Die Kommunikation kann bei dieser Datenverteilung verzahnt mit den Berechnungen der Teilergebnisse ausgeführt werden.

Die Operationen (d)-(e) für die Bestimmung und Anwendung der Vorkonditionierung bilden den aufwändigsten Teil eines parallelen Algorithmus. Die Bestimmung der Konditionierungsmatrix erfolgt vor dem eigentlichen Löseralgorithmus. Hierfür werden ge-

wöhnlich Splitting-Verfahren (z.B. Jacobi-Verfahren) bzw. vollständige oder unvollständige Faktorisierungen der Koeffizientenmatrix herangezogen (vgl. z.B. [Sch00]). Unvollständige Faktorisierung bedeutet, das in der Zerlegungsmatrix die Besetzungsstruktur der Koeffizientenmatrix erhalten bleibt (s. z.B. [Saa03, GL96]). In der Praxis werden für große, dünn besetzte Matrizen am häufigsten die unvollständige LU- oder unvollständige Cholesky-Zerlegung verwendet (s. a. direkte Verfahren). Die Parallelisierung für die Bestimmung erfolgt daher in ähnlicher Weise wie bei der Gauß-Elimination bzw. dem Cholesky-Algorithmus. Dies gilt auch für die Anwendung der Vorkonditionierung, wenn in jeder Iteration der Krylov-Unterraum Methode ein zusätzliches Gleichungssystem $Mz = r$ durch Vorwärts- und Rückwartssubstitution zu lösen ist.

Mehrgitter-Verfahren Mehrgitter-Verfahren stellen derzeit die modernsten und effizientesten Verfahren zur Lösung von Gleichungssystemen dar, die aus FEM-Diskretisierungen abgeleitet werden (vgl. hierzu [Hac76, Hac85, Bey98, Bra03]). Der Ansatz dieser Verfahrensklasse basiert auf der geschickten Nutzung einer Netzhierarchie $\mathcal{M} = (M_0, M_1, \ldots, M_k)$ für die Lösungsberechnung. Die Hierarchie speichert von Hierarchiestufe M_i zu Hierarchiestufe M_{i+1} ein gröberes aber strukturell gleichendes Netz. Man unterscheidet in dieser Klasse das sog. *geometrische* und das *algebraische* Mehrgitter-Verfahren, beide gleichen sich bis auf die Hierarchiekonstruktion der Netze bzw. der dazu gehörenden Gleichungssysteme. Beim geometrischen Verfahren werden die Netze ausgehend von der (feinen) FEM-Diskretisierung durch geeignet gewählte Vergröberungsstrategien konstruiert. Als Grundlage für das algebraische Mehrgitter-Verfahren dient nicht die Diskretisierung selbst, sondern die Adjazenzmatrix, die aus der Koeffizientenmatrix des Gleichungssystems abgeleitet wird. Algebraische Mehrgitter-Verfahren eignen sich somit auch für die numerische Behandlung von Problemstellungen, bei denen keine Diskretisierungsansätze vorhanden oder möglich sind.

Das Grundprinzip eines Mehrgitter-Verfahrens basiert auf der Glättung von kurz- und langwelligen Fehleranteilen des Residuums (Fehlerfunktion) eines zugehörigen Gleichungssystems. Die wichtigen Operationen hierfür sind Vor- und Nachglättung, Defektberechnung, Restriktion (Interpolation des Resiuduums vom feinen auf das grobe Netz) und Prolongation (Interpolation der Grobnetzlösung auf die Lösung für das feine Netz) (s. [Bra03, ALO02]). Die Zusammenfassung dieser Operationen für eine Iteration (eine Netzebene) des Mehrgitterverfahrens nennt man Grobgitterkorrektur-Schritt. Als Glätter werden in der Praxis üblicherweise Splitting-Verfahren (Jacobi- oder Gauß-Seidel-Verfahren) verwendet, da sie die kurzwelligen (oszilierenden) Fehleranteile mit wenigen Iterationen schnell dämpfen können. Auf der gröbsten Netzstufe wird i.A. das Gleichungssystem mit einem direkten Verfahren gelöst, da die Zahl der Gitterknoten bzw. der Unbekannten gegenüber den feineren Netzen sehr gering ist und somit so ein Lösungsverfahren (Zeitaufwand) rechtfertigt.

Für die Parallelisierung des Mehrgitter-Verfahrens ist es notwendig (s. z.B. [ALO02, Zum03]), dass jede Hierarchiestufe partitioniert für die Prozessoren bzw. Rechenknoten vorliegt. Die Hierarchiekonstruktion erfolgt parallel derart, dass das feinste Netz, also dasjenige, das aus der FEM-Diskretisierung entsteht, auf die Prozessoren gleichmä-

ßig (und zusammenhängend) verteilt wird. Jeder Prozessor vergröbert nun lokal seinen Netzanteil mit Hilfe einer für die Mehrgitter-Methode geeignet gewählten Strategie und erzeugt somit eine lokale Netzhierachie. Um den Zusammenhang zwischen den lokalen Netzen zum globalen Netz (innerhalb einer Hierarchiestufe) zu gewährleisten, werden Überlappungsränder zwischen den lokalen Anteilen bestimmt (s. Halotechnik weiter unten), über die parallel Informationen für die Löser (Glätter) ausgetauscht werden können. Die Anzahl der Hierarchiestufen wird in der Praxis so gewählt, dass auf jedem Prozessor bzw. Rechenknoten im gröbsten Netz noch genug Knoten vorhanden sind, um ein ausgwogenes Verhältnis zwischen Kommunikation und lokaler Rechenlast zu erreichen. Bei zu wenigen Knoten bzw. Unbekannten wird ansonsten zu viel kommuniziert und die Gesamtlaufzeit des Verfahrens damit unnötig erhöht.

Der aus rechentechnischer Sicht aufwändigste Teil im Mehrgitter-Verfahren ist die Ausführung der Glätter. Da hier gewöhnlich Splitting-Verfahren bzw. auf dem gröbsten Gitter direkte Verfahren verwendet werden, können diese mit den bereits beschriebenen, klassischen Verfahren parallel ausgeführt werden. Die Interpolationsoperatoren für Restriktion und Prolongation sowie die Defektberechnung arbeiten zumeist lokal auf den Daten bzw. den Netzen und benötigen daher keine Kommunikation zwischen den Prozessoren.

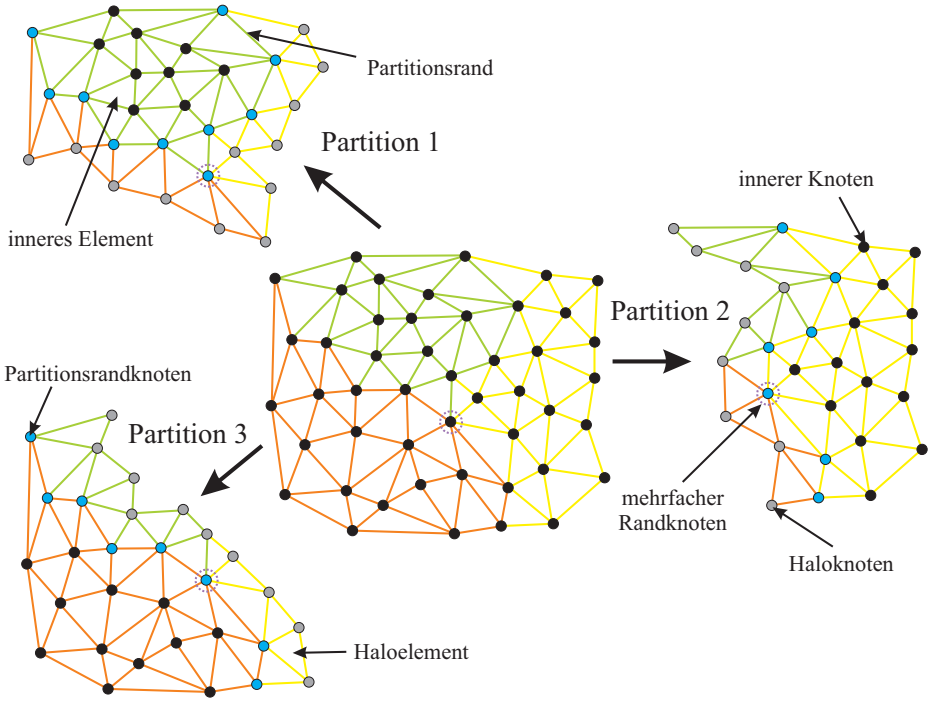
Die Technik des Halo

Die Finite Element Methode benötigt eine Diskretisierung des Simulationsgebietes. Diese Diskretisierung dient dazu, unter zu Hilfenahme geeignet gewählter Ansatzfunktionen für die zu berechnende Problemstellung, eine Koeffizientenmatrix für ein lineares Gleichungssystem aufzustellen (Matrix-Assemblierung). Ein Knoten in der Diskretisierung stellt zusammen mit seiner Konnektivität (Kanten zu den Nachbarknoten) eine Zeile der Matrix dar. Der Knoten selbst ist mit dem Diagonalelement der Matrixzeile assoziiert, die Nebendiagonaleinträge bilden die Kantenwerte ab.

Im parallelen Fall ist die Diskretisierung auf mehrere Prozessoren oder Rechenknoten aufgeteilt (partitioniert). Knoten, die auf dem Rand einer solchen Partition liegen, haben keine vollständigen Informationen über ihre geometrische Umgebung, d.h. es fehlen in der assoziierten Matrixzeile zum Knoten diejenigen Nebendiagonaleinträge der Kanten, die Teil einer Nachbarpartition sind. Um nun für die numerische Berechnung der Problemstellung auf einem Prozessor bzw. Rechenknoten diese notwendigen Daten vollständig zur Verfügung zu stellen, werden in der Diskretisierung Kopien der fehlenden Kanten und deren anhängigen Knoten aus den benachbarten Partitionen geometrisch hinzugefügt. Diese Kopien heißen Halo-, Schatten- oder auch Geistelemente (ghost elements). Die Gesamtheit aller Kopien für eine Partition ist der Halo.

Abbildung 2.2 verdeutlicht das Konzept des Halos anhand einer 3-Partitionierung für eine 2D-Gebietsdiskretisierung. In der Mitte ist die logische Partitionierung der Diskretisierung so dargestellt, als ob das Dreiecksnetz nicht aufgeteilt wäre. Die Dreiecke (Elemente) jeweils einer Farbe stellen eine eigene Partition dar (grün, gelb und orange).

Abbildung 2.2 Haloobjekte einer partitionierten Diskretisierung



Die Teilnetze in den Ecken der Abbildung zeigen diejenigen Teile der Diskretisierung, die ein Prozessor bzw. ein Rechenknoten tatsächlich speichert. Dabei ist zu erkennen, dass an dem Rand einer Partition, der nicht ein tatsächlicher Gebietsrand ist, ein Streifen aus Dreiecken (in 3D Tetraeder) der jeweiligen Nachbarpartition angehängt ist. Die Assemblierung der zur Teildiskretisierung gehörenden Matrix enthält als Haupt- und Nebendiagonalelemente die schwarzen Knoten (innere Knoten) und die blauen Knoten (Partitionsrandknoten), graue Knoten (Haloknoten) bilden nur Nebendiagonalelemente.

Für einen parallel arbeitenden Algorithmus, bspw. ein iterativer Gleichungslöser, muss nun durch eine Aktualisierungsfunktion (Halo-Aktualisierung, Halo-Update) sichergestellt werden, dass für die Partitionsrandknoten immer komplette, aktuelle Informationen vorliegen. Dies bedeutet, dass für eine verteilte Matrix die betroffenen Nebendiagonaleinträge aus den Nachbarpartitionen transferiert und in die lokalen, assoziierten Matrixeinträge kopiert werden müssen. Im Falle eines iterativen Gleichnugslösers muss dies in jeder Iteration erfolgen. Liegt eine k-Partitionierung mit $k > 2$ vor, so können Knoten bzw. Kanten in der lokalen Teildiskretisierung existieren, die Informationen von mehreren Nachbarpartitionen abbilden. In Abbildung 2.2 ist so ein Knoten durch einen gestrichelten Kreis markiert. Für solche Netzobjekte wird eine Entscheidungsmöglichkeit benötigt, wie mit den mehrfachen Daten beim Eintragen in die lokale Diskretisierung bzw. Matrix zu verfahren ist. Ein Beispiel für so eine Entscheidung wäre, dass immer nur von einem Nachbarn (dem mit der höchsten Partitionsnummer) die

Daten übernommen werden oder aber das arithmetische Mittel über alle Nachbardaten verwendet wird.

2.1.3 Adaption

Die Qualität der Lösung einer FEM-Simulation bzw. eines Simualtionsschrittes hängt stark mit der Diskretisierung des Simulationsgebietes für eine Problemstellung zusammen. Dynamische Vorgänge im Simulationsgebiet können mittels statischer Netze, d.h. Netze, die über den gesamten Simulationszeitraum keine geometrische Veränderung erfahren, schlecht oder nur mit erhöhtem numerischen Aufwand erfasst werden. Abhilfe für dieses Problem schafft die Adaption. Durch die Adaption wird an lokalen Stellen im Netz die Dichte der Knoten und damit die Zahl der Unbekannten verändert. Dadurch steigt die Lösungsqualität der numerischen Berechnung, da einerseits mehr Knoten an ausgewählten Stellen, wo die Lösung nicht genau genug ist, hinzukommen. Andererseits können Knoten an Stellen, an denen die Lösung genau genug ist, auch entfernt werden, wenn die Lösungsqualität dadurch nicht sinkt (Reduzierung der Unbekannten). Das Hinzufügen von Knoten zum Netz bezeichnet man als Verfeinerung, die Entfernung von Knoten bezeichnet man als Vergröberung.

Adaptionsstrategien

Es existieren mehrere Arten der Adaption (s. z.B. [Car97, GB98, Löh01]), die sich in ihren Strategien unterscheiden. Verfeinerung und Vergröberung zählen zu der sog. *geometrischen Adaption*, die auch als *h-Adaption* bekannt ist. Bei dieser Art von Adaption wird die geometrische Struktur eines Netzes, d.h. die Knotendichte und die Adjazenz der Knoten untereinander, im Verlauf der Simulation verändert. Zu der geometrischen Adaption gehört auch, neben der h-Adaption, die sog. *r-Adaption*. Der Unterschied zur h-Adaption besteht darin, dass keine neuen Knoten zum Netz hinzugefügt oder entfernt, sondern nur die Knotenpositionen verändert werden. Die Knotenkonnektivität bleibt erhalten. Die r-Adaption zählt somit auch zu den Netzoptimierungsverfahren, bei denen lokal und/oder global die möglichst optimale Form der Netzelemente (Maximierung der kleinsten Winkel) bestimmt wird.

Eine weitere Strategie für eine Adaption ist die *p-Adaption*. Im Gegensatz zur h- und r-Adaption werden im Netz keine Knoten hinzugefügt, entfernt oder verschoben. Eine Verbesserung der Lösungsqualität wird einzig durch die Anpassung der Polynomgrade der Ansatzfunktionen für die Finiten Elemente erreicht. Dies impliziert eine deutliche Erhöhung des numerischen Aufwandes (Interpolationsfunktionen höherer Grade). Die p-Adaption hat den Nachteil, dass sie für komplex berandete Geometrien (nicht linear bzw. planar) äußerst schwierig zu implementieren ist (vgl. [Car97, GB98]) und daher für solche Fälle selten angewendet wird. Die p-Adaption zählt nicht zu den geometrischen Adaptionen, sondern zu der numerischen Adaption. Eine Mischform stellt die

hp-Adaption dar, in der die geometrische Adaption mit Hilfe der p-Adaption erweitert wird, um zusätzliche Flexibilität zu erreichen.

Die *m-Adaption* stellt eine dritte Adaptionsstrategie dar, die ebenfalls zu der geometrischen Adaption zu zählen ist. Bei der m-Adaption werden entweder das gesamte Netz (globale m-Adaption) oder aber ein oder mehrere Teilbereiche (lokale m-Adaption) der Geometrie neu vernetzt. Werden Teilbereiche zur Neuvernetzung (Remeshing) ausgewählt, so werden diese aus der Diskretisierung komplett entfernt. Eine anschließende Knotenpositionsbestimmung auf der Basis der bisherigen berechneten Lösung und Fehlerschätzerergebnissen, fügt neue Knoten in diese gelöschten Bereiche ein, die durch einen Netzgenerierungsalgorithmus (z.B. Delaunay-Voronï Verfahren, Advancing-Front-Algorithmus, s. hierzu z.B. [TSW99]) zu einem neuen Netz verbunden werden.

Durch eine geometrische Adaption können Tetraeder entstehen, die in ihrer Form (sehr kleine Winkel oder Verletzung der Delaunay-Eigenschaft in der Nachbarschaft, s. [GB98]) ungünstigen Einfluß auf die Numerik bzw. die Problemstellung haben. Daher folgt nach einer geometrischen Adaption in der Regel eine Optimierungsphase auf lokaler (Teilbereiche) und/oder globaler (gesamtes Netz) Ebene, um solche Tetraederformen zu verbessern. Als Optimierungstechniken existieren sowohl topologische als auch geometrische Varianten. Zu den topologischen Optimierungen von Tetraederformen zählen Kanten- und Flächentausch benachbarter Tetraeder, Neuvernetzung[5] der direkten Umgebung eines Tetraeders, dem Zusammenspiel aus Knoten-, Kanten- und Dreiecksentfernung sowie Kantenaufspaltung und Minimierung des Knotengrades. Geometrische Optimierungen bestehen aus dem Verschieben von Knotenpositionen (Glättung) auf der Basis gewichteter baryzentrischer Mittelung. Dies kann sowohl lokal in Teilbereichen des Netzes bzw. in der direkten Umgebung eines Tetraeders als auch global mit allen Knoten des Netzes geschehen.

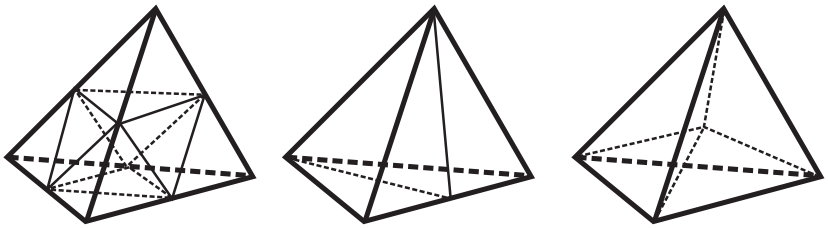
Techniken der geometrischen h-Adaption

Die h-Adaption, die für diese Arbeit die zentrale Adaptionstechnik darstellt, kann mit unterschiedlichen Varianten durchgeführt werden. Allen Varianten ist gemeinsam, dass sie einen oder mehrere Knoten in die Diskretisierung des Simulationsgebietes einfügen bzw. durch die inversen Operation Knoten entfernen. Es existieren drei Varianten, deren Verfeinerungsstrategie aus den jeweils folgenden Operationen bestehen:

- **Reguläre Verfeinerung** Das Tetraeder wird verfeinert, indem auf den sechs Kanten jeweils in der Mitte ein neuer Knoten eingefügt wird. Die sechs neuen Knoten werden so mit Kanten verbunden, dass in den Ecken des ursprünglichen Tetraeders vier kongruente Tetraeder und im Inneren ein Oktaeder entstehen. Das Oktaeder wird durch Einziehen einer Kante zwischen diagonal gegenüberliegenden Knoten in vier zum Ursprungstetraeder nicht-kongruente Tetraeder weiter unterteilt. Die

[5] Dies ist nicht zu verwechseln mit der m-Adaption, bei der größere Bereiche mit mehreren Tetraedern neu vernetzt werden.

Abbildung 2.3 Die drei Varianten der Tetraederverfeinerung



(a) (b) (c)

reguläre Verfeinerung erzeugt somit acht neue Teiltetraeder (s. Abbildung 2.3, Fall (a)).

- **Bisektion** Nur eine Kante des Tetraeders wird durch Einfügen eines neuen Knotens in der Mitte geteilt. Durch Hinzufügen von jeweils einer Kante auf den Dreiecksflächen, die zur gemittelten Kante adjazent sind, entstehen zwei neue Teiltetraeder. Diese Teiltetraeder sind nicht-kongruent zum Ursprungstetraeder (s. Abbildung 2.3, Fall (b)).
- **Baryzentrische Verfeinerung** Im Schwerpunktzentrum des Tetraeders wird ein neuer Knoten zur Diskretisierung hinzugefügt. Dieser wird mit den vier Eckpunkten des Tetraeders durch Kanten verbunden. Auf diese Weise entstehen vier neue, zum Ursprungstetraeder nicht-kongruente Teiltetraeder (s. Abbildung 2.3, Fall (c)).

Die Vergröberung für die drei Fälle wird durch die jeweilige Umkehroperation durchgeführt, d.h. durch die Entfernung der vormals hinzugefügten Knoten sowie der zusätzlichen Kanten und Flächen. Abbildung 2.3 zeigt die drei möglichen Varianten der Tetraederverfeinerung. Jede dieser drei Verfeinerungsvarianten hat seine Vor- und Nachteile.

Reguläre Verfeinerung Bei der regulären Verfeinerung entstehen vier kongruente und vier nicht-kongruente Teiltetraeder bzgl. des Ursprungstetraeder. Durch die Kongruenz bleiben die Raumwinkel, die für die Kondition des aus der Diskretisierung assemblierten Matrixsystems maßgeblich sind (vgl. [BA76, Kri92]), erhalten. Die vier nicht-kongruenten Teiltetraeder verschlechtern allerdings wegen der Raumwinkelverkleinerung die Kondition des Systems und beeinflussen somit die numerischen Löser, die auf den assemblierten Systematrizen arbeiten. Das Oktaeder, welches durch das Knoteneinfügen auf den sechs Außenkanten entsteht, kann durch jeweils drei Kanten in vier weitere Teiltetraeder unterteilt werden. Es muss bei dieser Unterteilung darauf geachtet werden, welche Kante dazu benutzt wird. Eine falsche Wahl führt bei fortgesetzter regulärer Verfeinerung des Tetraeders zu einer Degeneration der entstehenden Teiltetraeder, d.h. die Tetraeder werden immer flacher und ihr Volumen geht gegen Null (s. [Bey95, Bey98]). Wird für ein initiales, also ein im Simulationsverlauf

bisher nicht verfeinertes Tetraeder, die kürzeste Diagonale als Teilungskante ausgewählt und in nachfolgenden weiteren Verfeinerungen die Richtung dieser Diagonalen für die Teiltetraeder beibehalten, so entstehen maximal nur drei Kongruenzklassen und eine Degeneration wird verhindert (s. hierzu [Zha95, Bey00]). Die Festlegung der Diagonalrichtung kann erreicht werden, indem die Verfeinerung mit einer für das initialen Tetraeder lokalen Nummerierung durchgeführt und diese dann für die entstandenen Teiltetraeder beibehalten wird.

Verfeinerung durch Bisektion Bei der Bisektion eines Tetraeders entstehen im Gegensatz zur regulären Verfeinerung nur zwei Tetraeder. Dies bedeutet auch, dass durch den Verfeinerungsvorgang mit dieser Methode weniger neue Unbekannte zum Matrixsystem hinzugefügt werden als bei der regulären Verfeinerung. Da aber nicht-kongruente Teiltetraeder bei Anwenung dieser Strategie entstehen, gilt auch hier für die Raumwinkel das Problem der Konditionszahlverschlechterung des zur Diskretisierung assoziierten Matrixsystems. Ein weiteres Problem besteht in der fortgesetzten Anwendung dieser Verfeinerungsmethode auf ein Tetraeder. Da keine Kongruenz der entstehenden Teiltetraeder zum Ausgangstetraeder existiert, werden diese immer spitzwinkliger und degenerieren. Abhilfe für dieses Degenerationsverhalten kann dadurch erreicht werden, dass die Knoten immer nur auf der längsten Kante der Teiltetraeder eingefügt werden. Damit wird immer der größte Raumwinkel im Tetraeder geteilt. Eine Degeneration wird zwar nicht verhindert aber zumindest stark vermindert. Durch die "Längste-Kante-Bisektion" treten allerdings im Netz manchmal Situationen auf, in der die längste Kante eines Tetraeders nicht die längste Kante des Nachbartetraeders ist. In so einem Fall muss der Nachbar durch die Bisektionstrategie solange verfeinert werden, bis das an der gemeinsamen Kante anliegende Teiltetraeder des Nachbarn ebenfalls eine längste Kante an dieser hat. Bei dieser Situationsauflösung können durch die Bisektionsmethode sehr viele Teiltetraeder entstehen, die sich in einer Art Dominowelle um das ursprünglich verfeinerte Ausgangstetraeder herum ausbilden können.

Baryzentrische Verfeinerung Bei der baryzentrischen Verfeinerung wird der neue Knoten nicht auf einer Kante, sondern im Schwerpunkt des Tetraeders, welcher immer im Inneren ist (konvexes Polyeder), eingefügt. Durch diese Einfügeoperation entstehen wiederum vier nicht-kongruente Teiltetraeder. Als Vorteil für diese Verfeinerungsmethode gilt, dass benachbarte Tetraeder, also Tetraeder, die eine der äußeren Kanten mit dem verfeinerten Tetraeders gemeinsam haben, nicht von der Verfeinerung betroffen sind. Es entstehen keine sog. hängenden Knoten im Netz. Dies steht im Gegensatz zur Bisketion und der regulären Verfeinerung, bei der die Nachbarn durch die Knoteneinfügungen beeinflusst werden. Hängende Knoten, die für viele mathematischen Problemstellungen unerwünscht sind, da sie spezielle bzw. aufwändige numerische Verfahren zur Behandlung benötigen, haben somit eine entscheidende Bedeutung. Eine Auflösung von hängenden Knoten durch Abschlüsse (s. Abschnitt Irreguläre Adaption) ist daher für die beiden anderen Methoden notwendig. Die baryzentrische Verfeinerung kommt ohne diesen Zusatzaufwand aus. Dafür verhalten sich die Teiltetraeder bei fortgesetzter An-

wendung der Verfeinerung degenerativ. Diese kann nicht verhindert oder wie bei der regulären Verfeinerung begrenzt werden.

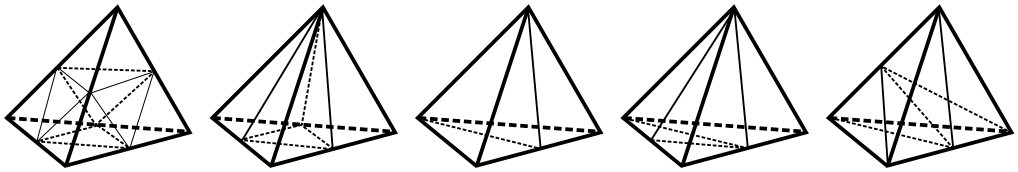
Irreguläre Adaption

Bisektion und reguläre Verfeinerung eines Tetraeders erzeugen auf den Kanten neue Knoten. Tetraeder in der direkten Nachbarschaft werden durch diese geteilte Kante mit ihrem Knoten beeinflusst, weil sie die ursprüngliche Kante gemeinsam haben. Da aber der neu eingefügte Knoten nicht immer eine direkte Verbindung zu allen Knoten eines Nachbartetraeder hat, bedarf dieser einer besonderen numerischen Behandlung für das Nachbartetraeder (Probleme bei den FEM-Ansatzfunktionen, s. z.B. [JL01, Bra03]). Solche Knoten werden als *hängend* bezeichnet. Die ursprüngliche Kante, die dieser Knoten teilt, wird ebenfalls als hängend bezeichnet. Dreiecksflächen können auch hängend sein, allerdings nur zwischen zwei Tetraedern, bei denen eines regulär verfeinert wurde, das andere aber nicht. Die drei Knoten auf den Seitenkanten der Dreiecksfläche sind demnach hängend. Hängende Dreiecksflächen können nicht bei Verfeinerung durch Bisektion auftreten.

Um den zusätzlichen (numerischen) Aufwand für solche hängenden Netzobjekte zu vermeiden, wird durch Anwenden der Verfeinerungsregel auf die betroffenen Nachbartetraeder versucht, die hängenden Objekte aufzulösen. D.h. Kantenverbindung vom hängenden Knoten zu allen anderen Knoten der beteiligten Tetraeder werden eingefügt. Im Falle der Bisektion ist dies ohne weiteres möglich. Für die reguläre Verfeinerungsregel geht das aber nicht, weil dadurch immer weiter neue Knoten auf den Kanten der beteiligten Nachbartetraeder eingefügt würden. Daher wird für die reguläre Verfeinerung ein spezieller Regelsatz an sog. Abschlüssen benötigt, der die Umgebung um hängende Netzobjekte korrigiert, d.h. alle hängenden Objekte beseitigt. Der Vorgang der regulären Verfeinerung mit Abschlüssen durch einen zusätzlichen Regelsatz heißt *irreguläre Adaption* oder auch *Rot-Grün Adaption* (s.a. Abschnitt 3.3). Die reguläre Verfeinerung wird auch als *rote Verfeinerung* bezeichnet und die Abschlüsse zum Auflösen der hängenden Netzobjekte *grüne Verfeinerung*.

Die irreguläre Adaption ist wesentlich komplexer für ein Simulationsprogramm zu realisieren als die Bisektion. Bei der Adaption durch Bisektion wird nur eine Regel benötigt, nämlich die Teilung eines Tetraeders in zwei weitere. Für die irreguläre Adaption existieren neben der Regel für die reguläre Verfeinerung noch 62 Möglichkeiten für Abschlüsse von hängenden Netzobjekten. Diese 62 Möglichkeiten berechnen sich aus der Anzahl und Position der verfeinerten bzw. geteilten Kanten eines Tetraeders, dessen Nachbarn durch die reguläre Verfeinerungsregel modifiziert wurden. Es gibt somit $2^6 - 2 = 62$ Möglichkeiten, wenn man die Fälle, in der keine oder alle Kanten verfeinert sind, nicht betrachtet. Aus Symmetriegründen kann diese Zahl weiter auf neun Fälle reduziert werden (s.a. Abschnitt 3.3). Da einige der Teiltetraeder, die bei Anwendung der Abschlussregeln im Inneren eines Tetraeders mit hängenden Kanten entstehen, sehr kleine Raumwinkel aufweisen können, kann aus numerischer Sicht (Verschlechterung der Kon-

Abbildung 2.4 Reduzierter Regelsatz der irregulären Verfeinerung



ditionszahl) die Zahl der neun Fälle weiter auf vier reduziert werden. Abbildung 2.4 zeigt die reguläre Verfeinerung (Fall a) mit den vier Abschlussregeln für verfeinerte Kanten eines Tetraeders (Fall b-e), die in der Praxis häufig Verwendung finden (s.a. Abbildung 3.4 zur Herleitung der Abschlüsse).

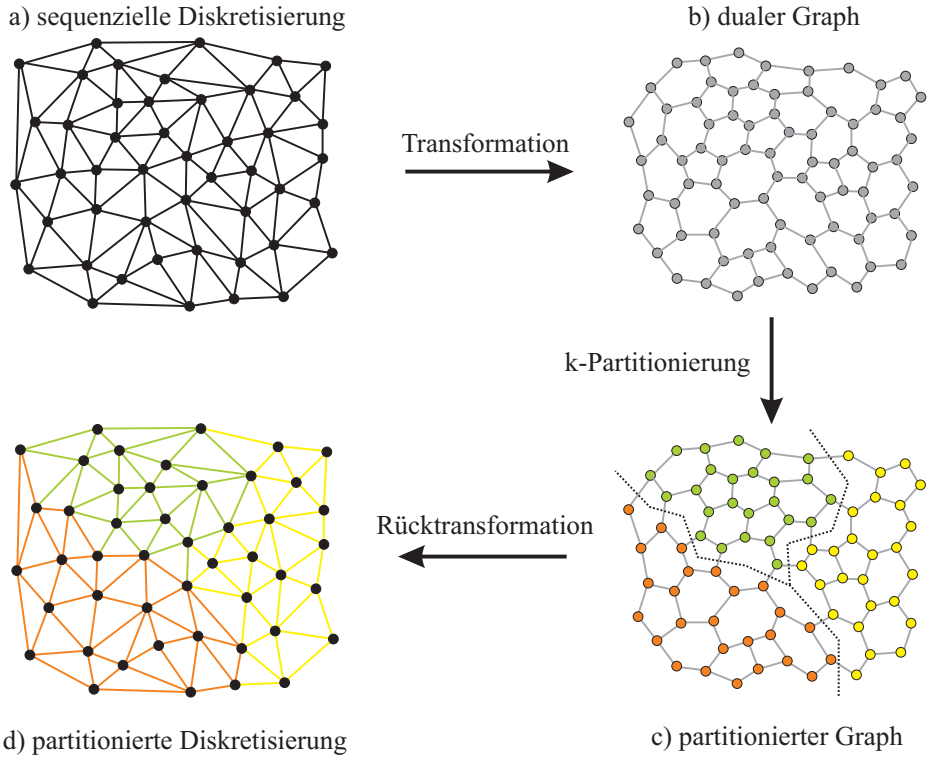
2.1.4 Partitionierung und Lastverteilung

Die Partitionierung stellt für die parallele Bearbeitung von numerischen Problemstellungen auf der Basis des FEM-Ansatzes eine fundamentale Grundlage dar. Die Aufgabe der Partitionierung ist es, eine FE-Diskretisierung (Netz) derart aufzuteilen, dass in jedem Teil der Diskretisierung (möglichst) gleich viele Elemente vorhanden sind oder aber der Rechenaufwand für diese Elemente auf allen Partitionen gleich ist.

Die Partitionierung einer FE-Diskretisierung erfolgt nicht an der Diskretisierung selbst, sondern an dem aus der Diskretisierung erzeugten dualen Graphen. In Abbildung 2.5 ist der Ablauf einer k-Partitionierung aufgezeigt. Eine Diskretisierung (a), die mit Hilfe eines Netzgenerierungsalgorithmus (s. z.B. [She97, Lis99, TSW99, Si04, Si07, SGG07]) erzeugt wurde, wird durch eine Transformationsoperation in den dualen Graphen umgeformt (b). Die Transformation bildet zu jedem Element der Diskretisierung (in 2D Dreieck, in 3D Tetraeder) im dualen Graphen einen Knoten. Die Kanten zwischen den Knoten bilden die Nachbarschaft der Elemente in der Diskretisierung ab. Zwei Elemente der Diskretisierung gelten hierbei als benachbart, wenn sie einen Knoten, eine Kante oder, wenn es sich bei der Diskretisierung um eine Tetraedierung handelt, auch eine Dreiecksfläche gemeinsam haben. Der duale Graph in Abbildung 2.5 nutzt als Nachbarschaftsrelation gemeinsame Kanten zwischen Elementen (Dreiecke). Durch einen Partitionierungsalgorithmus erfolgt nun die gleichmäßige Aufteilung der Knoten im dualen Graphen (c). Dieser Graph wird mit Hilfe der inversen Transformationsoperation wieder in die eigentliche Diskretisierung überführt (d).

Da in parallelen (verteilt arbeitenden) Programmen die Kommunikation zwischen Rechenknoten oftmals einen Flaschenhals in der Laufzeit bildet, wird als zusätzliches Optimierungsziel bei der Partitionierung eines Graphen bzw. einer FE-Diskretisierung die Minimierung der Kommunikationsoperationen bzw. der Datenmenge, die zwischen den Kommunikationspartnern ausgetauscht werden muss, gefordert. Dies bedeutet u.a.,

Abbildung 2.5 Partitionierung einer Diskretisierung



dass eine Partition möglichst wenig Nachbarpartitionen besitzen sollte und die Fläche des Partitionsrandes minimal ist, d.h. der Kantenschnitt im dualen Graphen zur Diskretisierung sollte minimal sein. Der Kantenschnitt dient hierbei als *Näherung für den Kommunikationsaufwand* (vgl. hierzu [Hen98]). Gewünscht wird weiterhin für einige Anwendungsfälle (z.B. vereinfachte Partikelsuche im parallelen Fall bei CFD-Problemstellungen), dass eine Partition zusammenhängend sein soll. Formal lässt sich das k-Partitionierungsproblem folgendermaßen beschreiben:

DEFINITION 2.1 (k-PARTITIONIERUNG EINES GRAPHEN)
Gegeben sei $k \in \mathbb{N}$ *und ein ungerichteter Graph* $G = (V, E)$, *wobei* $V = \{v_1, \ldots, v_m\}$ *eine Menge aus* $m \in \mathbb{N}$ *Knoten (*$m \geq k$ *sei hier vorausgesetzt) und* $E = \{e_{ij} = (v_i, v_j) \mid v_i, v_j \in V, i \neq j\}$ *eine Menge aus Kanten darstellt.*

Gesucht ist eine Funktion $\pi_k : V \mapsto \{1, \ldots, k\}$, *die* V *in* k *Teilmengen* V_i, $1 \leq i \leq k$, *aufteilt, so dass gilt:* $V = V_1 \cup \cdots \cup V_k$ *und* $V_i \cap V_j = \emptyset$ *für* $1 \leq i, j \leq k$ *und* $i \neq j$. *Für* π_k *wird zusätzlich gefordert, dass der Kantenschnitt* $\mathrm{cut}(\pi_k) = |\{e_{ij} = (v_i, v_j) \in E \mid \pi_k(v_i) \neq \pi_k(v_j)\}|$ *minimal ist.*

Es gilt: k-Partitionierungsproblem $\in \mathcal{NPC}$ *(NP-complete)* (s. z.B. [GJ79]). In der Praxis werden daher sehr häufig Heuristik-Algorithmen zur Bestimmung der Partitionierung verwendet.

Für $k = 2$ spricht man von einer *Bisektion eines Graphen*. k-Partitionierungen werden häufig auf eine Bisektion zurückgeführt (s. z.B. [DP97, Pre98]). Im Falle von $k = 2^r$ lässt sich die k-Partitionierung leicht auf r rekursive Bisketionen zurückführen, falls $k \neq 2^r$ wird die Bisektion derart modifiziert, dass die beiden Mengen V_1 und V_2 nicht gleichmäßig aufgeteilt werden, sondern in die Größen $|V_1| = \frac{m}{k}$ und $|V_2| = m - \frac{m}{k} = m(1 - \frac{1}{k})$. Auf die Menge V_2 wird dann wieder rekursiv die modifizierte Bisektion angewendet, bis schließlich k Teilmengen entstanden sind.

Partitionierungsstrategien

Es existieren eine Reihen von Strategien bzw. Heuristiken, um eine Partitionierung mit Kantenschnittminimierung für einen Graphen zu bestimmen. Einen umfangreichen Überblick über Partitionierungsstrategien sowie deren theoretische Betrachtungen (bspw. Schrankenbestimmung) kann in [Lül96, Els97, Die98, Pre00, Wie03, Sch06] und deren verwendeten Referenzen gefunden werden. An dieser Stelle soll ein kurzer Überblick über die klassischen Strategien erfolgen, die nicht auf einem Greedy-Ansatz basieren. Partitionierungen mittels Greedy-Ansatz erzeugen größtenteils schlechtere oder nicht eindeutige Ergebnisse bzgl. des Kantenschnitts (vgl. z.B. [ALO02]) als die folgenden aufgeführten Verfahren.

Globale Strategien Globale Strategien betrachten zur Bestimmung einer Partitionierung den gesamten Graphen. Aufgrund der Größe der Ausgangsgraphen benötigen die Verfahren mit globalem Ansatz oftmals lange Laufzeit (bspw. wegen numerischer Bestimmung von Eigenwerten und -vektoren, s.u.). Zu den in der Praxis am meisten verwendeten Verfahren für diese Strategie zählen die *spektrale Bisektion*, *Implizite Partitionierung durch lineare Nummerierung* und *koordinaten-basierte Partitionierung*.

Die spektrale Bisektion (s. z.B. [ALO02]) nutzt algebraische Eigenschaften der *Laplace-Matrix* aus, die aus dem Ausgangsgraphen konstruiert wird, um eine Partitionierung zu bestimmen. Die Laplace-Matrix $L(G) = (l_{ij})$ zu einem ungerichteten (zusammenhängenden) Graphen $G = (V, E)$ ist durch

$$l_{ij} = \begin{cases} -1 & : \; i \neq j \wedge e_{ij} \in E \\ 0 & : \; i \neq j \wedge e_{ij} \notin E \\ \deg(v_i) & : \; i = j \end{cases} \quad \text{für } 1 \leq i, j \leq |V| \tag{2.4}$$

definiert. Durch Berechnung des zweitkleinsten Eigenwertes $\lambda_2 > 0$ und dem zugehörigen Eigenvektor $u = (u_i)$ (der sog. *Fiedler-Vektor* [Fie73]), der die algebraische Konnektivität des Graphen G angibt (vgl. z.B. [PSL90]), kann eine Bipartitionierung ermittelt werden. Dazu wird der Median $\overline{u_i}$ über alle Komponenten von u bestimmt und V in die Mengen $V_1 = \{v_i \in V \mid u_i \leq \overline{u_i}\}$ und $V_2 = \{v_i \in V \mid u_i \geq \overline{u_i}\}$ mit $||V_1| - |V_2|| \leq 1$ aufgeteilt. Bei großen Eingabegraphen für dieses Verfahren ist aufgrund der numerischen Berechnung von Eigenwert/-vektor die Laufzeit entsprechend hoch.

Verfahren nach der Strategie impliziter Partitionierung durch lineare Nummerierung basieren auf einer bijektiven Zuordnung $f : V \mapsto \{1, \ldots |V|\}$ auf die Knoten des Graphen $G = (V, E)$ und gleichmäßiger Intervallaufteilung des Wertebereiches auf die k Partitionen (für $p \in \{1, \ldots, k\} : I_p := [a_{p_i}, b_{p_j}] \subset \{1, \ldots, |V|\}$ wobei $\forall l, m \in \{1, \ldots, k\}, l \neq m :$ $I_l \cap I_m = \emptyset$ und $||I_l| - |I_m|| \leq 1$). Der aufwändigste Teil bei dieser Art der Partitionierung besteht in der konsistenten Abbildungsbestimmung (Mapping) auf den Graphen aus der Diskretisierung. Zu dieser Verfahrensklasse gehören z.B. Partitionierung mittels raumfüllender Kurven (space-filling curves, SPC) und Partitionierung mittels graphfüllender Kurven (graph-filling curves, GFC). Raumfüllende Kurven (s. z.B. [Sag94])) basieren auf einem rekursiven Ansatz und nutzen u.a. Koordinateninformationen zur Konstruktion (vgl. hierzu [Zum01, Zum03, Wie03]). Durch die Konstruktionsvorschrift der Kurven (z.B. Hilbert-, Lebesgue- und Sierpinskikurve) werden die Knoten des Graphen, die geometrisch gesehen lokal zusammen liegen, in gleiche Intervallgrenzen abgebildet (Lokalitätsprinzip). Solche Kurven eignen sich gut für Graphen aus strukturierten Netzen, erzeugen allerdings bzgl. des Kantenschnittes schlechtere Ergebnisse als andere Verfahren bei Graphen aus unstrukturierten oder adaptiven Netzen mit hoher Lokalität oder Graphen aus Diskretisierungen, deren Simulationsgebiet Ω nicht sternförmig[6] ist. Für diese Klassen von Graphen eignen sich besser Partitionierungen auf der Basis graphfüllender Kurven (vgl. hierzu [SW03, Wie03, Sch06]). Bei dieser Art der Partitionierung wird ein Hierarchiebaum aus der Diskretisierung aufgebaut, indem das Diskretisierungsgebiet Ω_h rekursiv in $q \in \mathbb{N}$, q konstant, Teilgebiete aufgeteilt wird, bis ein Gebiet nur noch aus einem Knoten besteht. Es entsteht somit ein q-ärer Baum über die Teilgebiete. Eine Traversierung des Baumes ergibt dann eine lineare Nummerierung der Knoten. Durch Umordnung von Teilbäumen kann die Kantenschnittoptimierung zwischen den Teilgebieten erfolgen. Zur Bestimmung einer guten Partitionierung ist es erforderlich, ein geeignetes q zu finden und optimale Teilbaumvertauschungen auszuführen (s. hierzu [Sch06]).

Koordinaten-basierte Verfahren (s. z.B. [Lül96, Els97, Die98]) nutzen ausschließlich geometrische Informationen aus der Diskretisierung Ω_h des Gebietes Ω. Die bekanntesten Verfahren in dieser Strategieklasse sind die Partitionierung mit Koordinatensortierung und Inertialpartitionierung. Bei Partitionierung mit Koordinatensortierung werden die Knoten nach ihrer x-, y- und z-Koordinate sortiert und bzgl. des Abstandes in jeder Achsenrichtung zueinander in Teilmengen aufgeteilt. Dies lässt sich auch rekrusiv durchführen. Das Problem dieser Partitionierungsmethode ist, dass die Partitionierung abhängig von der Koordinatenlage der Knoten innerhalb der Diskretisierung ist. Eine Rotation des Gebietes um einen Punkt kann für den Kantenschnitt eine Verbesserung oder Verschlechterung ergeben. Diesem Problem versucht die Inertialpartitionierung entgegenzuwirken. Für die Inertialpartitionierung werden die Knoten mit ihren Koordinaten als Masseschwerpunkte aufgefasst und damit ein Masseschwerpunkt für die Diskretisierung bzw. den Graphen bestimmt. Über diesen Schwerpunkt werden dann

[6] Sternförmig bedeutet, dass es mindestens einen Punkt in Ω geben muss, so dass die Verbindungslinie zwischen diesem Punkt und jedem anderen Punkt aus Ω vollständig in Ω liegt. Anders ausgedrückt, Ω darf keine Löcher enthalten.

Aufteilungen der Knoten bzgl. des Abstandes zum Schwerpunkt und untereinander berechnet (Bestimmung eines eigenen Koordinatensystems mit Knotentransformation).

Lokale Strategien Lokale Strategien bzw. lokale Heuristiken berechnen keine initiale Partitionierung, sondern arbeiten auf einer vorgegebenen Partitionierung. Sie versuchen durch lokale Operationen, d.h. Austauschen von einzelnen oder Mengen von Graphknoten zwischen benachbarten Partitionen, Kantenschnittverbesserungen zu erhalten. Lokale Strategien arbeiten daher iterativ an einer verbesserten Lösung zum Graphpartitionierungsproblem. Zu den bekanntesten Verfahren in dieser Strategieklasse gehören die *Partitionierung nach Kernighan-Lin* (auch K/L-Algorithmus genannt) und die *Partitionierung durch die Methode der hilfreichen Mengen*.

Der Partitionierungsalgorithmus nach Kernighan-Lin (s. [KL70], mit Verbesserungen durch Fiduccia-Mattheyses [FM82]) zählt zu den ältesten lokalen Partitionierungsstrategien. Die grundlegende Idee für den K/L-Algorithmus ist, ausgehend von einer initialen Partitionierung (bspw. durch koordinaten-basierte Partitionierungsverfahren oder zufällige Zuordnung) durch lokal begrenzten Austausch oder Wanderung von Knotenpaaren zwischen benachbarten Partitionen, eine Verbesserung des Kantenschnitts zu erhalten. Die Bestimmung, welche Knoten zwischen zwei Partitionen V_a und V_b ausgetauscht werden bzw. wandern, erfolgt durch Berechnung des Kantenschnittgewinns für einen Knoten mit Hilfe einer Kantengewichtsfunktion $w_E : V \times V \mapsto \mathbb{R}$:

$$\text{für } v \in V_a : \text{gain}(v) := \text{gain}(v, V_a, V_b) = \sum_{\substack{(v,u)\in E \\ u\in V_b}} w_E(v,u) - \sum_{\substack{(v,u)\in E \\ u\in V_a}} w_E(v,u). \tag{2.5}$$

Zwei Knoten $v \in V_a$ und $u \in V_b$ tauschen nun ihre Partitionen, wenn $\text{gain}(v)+\text{gain}(u)-2 \cdot w_E(v,u)$ eine Verbesserung ergibt, d.h. den Kantenschnitt verringert. Da es u.U. mehrere Möglichkeiten gibt, einen Kantenschnittgewinn in einer lokalen Betrachtung zu erhalten und eine komplette Untersuchung aller Möglichkeiten aufwändig ist, wurden Verbesserungen für den ursprünglichen K/L-Algorithmus entwickelt, die die Laufzeit und die Qualitätsentscheidungen wesentlich verbessern (s. hierzu z.B. [FM82, Dut93, KK97]).

Partitionierung durch die Methode der hilfreichen Mengen (s. [HM91, DMP95]) stellt eine Erweiterung des K/L-Algorithmus in der Weise dar, dass nicht einzelne Knotenpaare, sondern Mengen von Knoten zwischen benachbarten Partitionen betrachtet werden. Für diese Methode arbeiten zwei Operationen auf einem Graphen, die Berechnung einer (i, g)-hilfreichen Menge und eine Balancierungsoperation (vgl. [Lül96, Die98, Pre00, Sch06]). Eine (i, g)-hilfreiche Menge S für eine k-Partitionierung $V_1, \ldots, V_k$ von V (gegeben durch Partitionierungsfunktion π, s. Definition 2.1) ist definiert durch:

$$\begin{aligned} &\text{für } i \in \{1, \ldots, k\} \text{ und } S \subset V \setminus V_i : \\ &\qquad \text{gain}(S, i) := \textstyle\sum_{v\in S} \text{ext}_{V_i}(v) - \text{ext}_{V_{\pi(v)}\setminus S}(v) + \text{int}_S(v) = g, \end{aligned} \tag{2.6}$$

wobei $\text{ext}_M(v)$ die Zahl der externen Kanten eines Knotens v bzgl. der Menge M bezeichnet und $\text{int}_M(v)$ die Anzahl der internen Kanten. Die (i, g)-hilfreiche Menge S beschreibt somit den Kantenschnittgewinn g, wenn die Menge S von Knoten zur Partition

V_i migriert. Nach der Migration von Knoten nach V_i durch eine (i, g)-hilfreiche Menge ist in den beteiligten Partitionen $V_j \neq V_i$ ein Ungleichgewicht entstanden, welches durch die Balancierungsoperation (s.o.) wieder ausgeglichen wird. Für eine detailiertere Beschreibung der Operationen bzw. des Verfahrens ist z.B. [Lül96, Die98, Pre00] zu empfehlen; Verbesserungen des Verfahrens sind in [Sch06] zu finden.

Partitionierung mit Hilfe der Multilevel-Methode Ein algorithmisches Hilfmittel für die Partitionierung ist die Multilevel-Methode (s. z.B. [ALO02]). Ähnlich wie beim Mehrgitter-Verfahren (s. Abschnitt 2.1.2) wird für einen zu partitionierenden Graphen $G = G^0$ eine Hierarchie $\mathcal{H}_G^m := (G^0, \ldots, G^i, G^{i+1}, \ldots, G^m)$ von Grobgraphen konstruiert. Die Vergröberung eines Graphen G^i zu G^{i+1} für die Hierarchie erfolgt durch Zusammenfassung von benachbarten (adjazenten) Knoten zu Mengen. Diese Knotenmengen, die untereinander disjunkt sind, werden *Multiknoten* genannt und ersetzen im Grobgraphen G^{i+1} die Knotenmengen durch neue Knoten. Dabei bilden die externen Kanten der Knotenmengen neue Kanten eines Multiknoten. Die Auswahl, welche Knoten zu Multiknoten zusammengefasst werden, kann bestimmten Kriterien folgen (bspw. den Grad der Knoten bzw. Multiknoten minimieren oder maximieren) oder aber zufällig sein (random matching).

Auf dem gröbsten Level m erfolgt die eigentliche Partitionierung anhand eines z.B. weiter oben beschriebenen Verfahrens. Aufgrund der (starken) Reduzierung der Knoten- und Kantenanzahl im Graphen G^m kann die Berechnung einer k-Partitionierung schnell und effizient durchgeführt werden. Die Anzahl der Stufen m sowie die Zahl der zu einem Multiknoten zusammengefassten Knoten sollten für die Hierarchiekonstruktion so gewählt werden, dass ausreichend Knoten in G^m für das verwendete Partitionierungsverfahren vorhanden sind. Idealerweise sind k Multiknoten bei einer k-Partitionierung in G^m, so dass jeder Partition ein Knoten zugeordnet werden kann.

Ist G^m partitioniert, so wird die gefundene Lösung von G^m auf G^0 übertragen, indem die Lösung in jeder Stufe i vom groben Graphen G^i auf seinen Vorgängergraphen G^{i-1} interpoliert wird. Dazu werden die in dieser Hierarchiestufe i zusammengefassten Multiknoten entfernt und die dazu assoziierten Knotenmengen samt interner Kanten hinzugefügt (rückverfeinert), um den Graphen G^{i-1} zu bilden. Da durch diesen Vorgang die "Optimalität" der Lösung von G^i nicht eine gute Partitionierung für G^{i-1} darstellen muss, dient diese als Startlösung für einen iterativen Verbesserungsalgorithmus (bspw. durch K/L-Algorithmus). Wurden alle Stufen der Hierarchie $\mathcal{H}_G^m$ rückwärts durchlaufen, bildet schließlich der Graph G^0 eine gültige Partitionierung für den Graphen G. Durch die Reduzierung der Datenmenge (Knote, Kanten und Konnektivität) in der konstruierten Hierarchie sowie der Reduzierung der zu betrachtenden Auswahlmöglichkeiten für lokale Strategien (z.B. Knotenpaare bzw. hilfreiche Mengen) arbeiten Partitionierungsstrategien mit der Multilevel-Methode besonders schnell und effizient (vgl. z.B. [KK98a]).

Erweiterung zur Lastverteilung

Die Lastverteilung bzw. Repartitionierung stellt einen Spezialfall der Partitionierung dar, in dem sich der Diskretisierungsgraph während der Laufzeit einer Simulation dynamisch verändert (bspw. durch eine geometrische Adaption) und somit ein Ungleichgewicht in der Verteilung der Elemente auf den Partitionen entsteht. Zu den Optimierungszielen der Lastverteilung gehört noch zusätzlich zur Ausgleichsbestimmung die Minimierung der Migrationsknotenzahl. Aus der Sicht von parallelen FEM-Anwendungen sind weitere Anforderungen an eine Lastverteilung gewünscht, beispielsweise Glattheit des Partitionsrandes (keine Ausbuchtungen oder Elementschläuche mit Durchmessern von wenigen Netzelementen), Form der Partitionen oder Zusammenhang jeder einzelnen Partition. Neben modifizierten und verbesserten Verfahren, die auf obige Partitionierungsstrategien basieren (vgl. bspw. [KK98a, KK99, Wal02]), existieren auch Verfahren, die mittels Diffusion (bspw. Bubble-FOS/C [MMS06, Sch06]) oder Netzwerkflussanalysen eine dynamische Lastverteilung berechnen. Eine ausführliche Behandlung und weiterführende Literaturreferenzen bietet hierzu u.a. [Lül96, Die98, Pre00, Sch06].

2.2 Mathematische Betrachtung des Tetraeders

2.2.1 Geometrie des Simplex

Das Tetraeder stellt in dieser Arbeit das grundlegende geometrische und mathematische Objekt dar. Daher ist es notwendig, seine Eigenschaften, die für das Verständnis der Betrachtungen in den nachfolgenden Abschnitten und Kapiteln wichtig sind, hier näher zu erläutern. Zu Beginn erfolgt eine Beschreibung der Klasse der *Simplizes* im $\mathbb{R}^n$ mit ihren Eigenschaften, in die das Tetraeder und das Dreieck als Konstruktionselement einzuordnen ist.

Das Simplex im $\mathbb{R}^n$

Für die Erzeugung einer FE-Diskretisierung eines Simulationsgebietes Ω benötigt man eine Zerlegung des Gebietes in endlich viele, nicht degenerierte Elemente, um auf dieser eine approximative Lösung der Problemstellung zu berechnen. Ein solches Element kann z.B. ein Simplex sein[7], welches im Folgenden genauer definiert wird (vgl. [Bey98] und [GB98]).

DEFINITION 2.2 (k-SIMPLEX IM $\mathbb{R}^n$)
Für $n, k \in \mathbb{N}_0$ *mit* $0 \leq k \leq n$ *definiert eine abgeschlossene Menge* $S \subset \mathbb{R}^n$ *ein* k-Simplex *im* $\mathbb{R}^n$, *wenn* S *die konvexe Hülle von* $k+1$ *Punkten* $x^{(0)}, \ldots, x^{(k)}$ *mit* $x^{(i)} = (x_1^{(i)}, \ldots, x_n^{(i)})^T$ *im* $\mathbb{R}^n$ *darstellt. Mit der Schreibweise* $S = [x^{(0)}, \ldots, x^{(k)}]$ *wird eine eindeutige Reihenfolge der Eckpunkte* $x^{(i)}$ *von* S *festgelegt.*

[7] Weitere Möglichkeiten sind u. a. Hexaeder, Dodekaeder oder hybride Kombinationen aus verschiedenen *platonischen Körpern*.

Die Eckpunkte $x^{(i)}$ werden auch oft Knoten genannt. Ist $k = n$ gegeben, so kann das k-Simplex S auch einfach nur als *Simplex* beschrieben werden. Das (2)- und (3)-Simplex im $\mathbb{R}^3$ wird gewöhnlich mit Dreieck bzw. Tetraeder bezeichnet.

Das Tetraeder wird in den folgenden Abschnitten und Kapiteln als ein aus Grundobjekten zusammengesetztes Objekt betrachtet. Diese Grundobjekte werden in der Mathematik als *Randsimplizes* bezeichnet. Dazu dient folgende Definition.

DEFINITION 2.3 (l-RANDSIMPLEX IM $\mathbb{R}^n$)
Sei durch $S = [x^{(0)}, \ldots, x^{(k)}]$, $0 < k \leq n$ *ein* (k)*-Simplex im* $\mathbb{R}^n$ *gegeben. Ein* (l)*-Simplex* S', $0 \leq l < k$ *heißt* (l)-Randsimplex *von* S*, wenn eine geordnete Indexmenge* $I = \{i_0, \ldots, i_l\}$ *mit* $0 \leq i_0 < i_1 < \ldots < i_l < k$ *existiert, so dass* $S' = [x^{(i_0)}, \ldots, x^{(i_l)}]$ *gilt.*

Die Forderung, dass die Indexmenge I geordnet sein muss, bedeutet, dass die Reihenfolge der Eckpunkte eines Randsimplizes S' mit der durch S induzierten Reihenfolge übereinstimmt. (0)-Randsimplizes sind gerade die Eckpunkte von S, (1)-Randsimplizes werden als Kanten bezeichnet. Im $\mathbb{R}^3$ definieren die (2)-Randsimplizes die Dreickesflächen von S und somit die eines Tetraeders.

Von großem Interesse für die nachfolgenden Qualitätsmetrik-Betrachtungen ist die Berechnung des Volumens eines Simplex bzw. in $\mathbb{R}^3$ des Tetraeders und der Flächeninhalt seiner Randsimplizes. Dazu dienen zur Berechnung die beiden folgenden Lemmata.

LEMMA 2.4 (VOLUMEN EINES (k)-SIMPLEX IM $\mathbb{R}^n$ [BEY98])
Gegeben sei ein (k)*-Simplex* $S = [x^{(0)}, \ldots, x^{(k)}]$, $0 < k \leq n$, $S \subset \mathbb{R}^n$. *Bezeichne* $G = (\langle l_i, l_j \rangle)_{0<i,j\leq k}$ *die Gramsche Matrix über den Kantenvektoren* $l_i := x^{(i)} - x^{(0)}$, $0 < i \leq k$, *mit Bezugsknoten* $x^{(0)}$. *Dann kann das Volumen* S_V *des* (k)*-Simplex* S *berechnet werden durch* $S_V = |\det(G)|^{\frac{1}{2}}/k!$.

Der Bezugsknoten für die Kantenvektoren l_i der Gramschen Matrix G kann ein beliebiger aber fester Knoten $x^{(j)}$ aus der Menge $\{x^{(0)}, \ldots, x^{(k)}\}$ sein. Die Volumenberechnung aus Lemma 2.4 ändert sich dadurch nicht.

Die Berechnung des Volumes eines (n)-Simplex im $\mathbb{R}^n$ gestaltet sich einfacher, da hierfür keine Gramsche Matrix aufgestellt werden muss. Dafür betrachtet man die folgende $(n+1) \times (n+1)$ Matrix V_S, die aus den Eckpunkten $x^{(i)} = (x_1^{(i)}, \ldots, x_n^{(i)})^T$ von S gebildet wird:

$$V_S := \begin{pmatrix} x_1^{(0)} & x_2^{(0)} & \cdots & x_n^{(0)} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_1^{(n)} & x_2^{(n)} & \cdots & x_n^{(n)} & 1 \end{pmatrix}. \tag{2.7}$$

LEMMA 2.5 (VOLUMEN EINES (n)-SIMPLEX IM $\mathbb{R}^n$ [BEY98])
Ist durch $S = [x^{(0)}, \ldots, x^{(k)}]$, $0 \leq k \leq n$, *mit* $x^{(i)} = (x_1^{(i)}, \ldots, x_n^{(i)})^T \in \mathbb{R}^n$ *ein* (n)*-Simplex im* $\mathbb{R}^n$ *gegeben, so kann das Volumen* S_V *von* S *berechnet werden durch* $S_V = |\det(V_S)|/n!$.

Für den konkreten Fall im $\mathbb{R}^3$ bedeutet dies nun für ein Tetraeder $T = [p^{(0)}, p^{(1)}, p^{(2)}, p^{(3)}]$ mit $p^{(i)} = (x_i, y_i, z_i)^T \in \mathbb{R}^3$

$$V_T = \frac{1}{6} \begin{vmatrix} x_0 & y_0 & z_0 & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} \tag{2.8}$$

bzw. nach Umformung die für Computer berechnungsgünstigere Variante mit $p^{(0)}$ als Bezugseckpunkt (abgeleitet aus dem Spatprodukt $(p^{(1)}-p^{(0)})\times(p^{(2)}-p^{(0)})\cdot(p^{(3)}-p^{(0)})$)

$$V_T = \frac{1}{6} \begin{vmatrix} x_1 - x_0 & x_2 - x_0 & x_3 - x_0 \\ y_1 - y_0 & y_2 - y_0 & y_3 - y_0 \\ z_1 - z_0 & z_2 - z_0 & z_3 - z_0 \end{vmatrix}. \tag{2.9}$$

Wie aus den Formeln (2.8) bzw. (2.9) ersichtlich ist, ist das Volumen V_T eines Tetraeders T vorzeichenbehaftet. Dies wird durch die Reihenfolge der Eckpunkte $x^{(i)}$ von T bestimmt, die somit auch für die Randsimplizes, d.h. die Dreiecke, eine Reihenfolge festlegen. Im Folgenden wird immer angenommen, dass die Aufzählungsreihenfolge der Eckpunkte eines Tetraeders im mathematisch positiven Sinn erfolgt. Damit ist auch das Volumen V_T stets positiv. Im Falle einer Reihenfolge der Eckpunkte mit der das Volumen negativ ist, genügt eine Vertauschung des ersten mit dem letzten Eckpunkt in der Liste, um ein positives Volumen zu erhalten.

Gilt für ein (k)-Simplex S, dass sein Volumen $V_S = 0$ ist, so nennt man S *entartet*. Geometrisch bedeutet dies, dass die Eckpunkte von S alle auf einer $(n-1)$-dimensionalen Hyperebene liegen, d.h. linear abhängig sind. Im $\mathbb{R}^3$ sagt man auch, die vier Knoten $p^{(0)}, \ldots, p^{(3)}$ eines Tetraeders T sind koplanar und bilden ein *degeneriertes Tetraeder* (s. a. Abschnitt 2.2.2).

Möchte man feststellen, ob ein gegebener Punkt $x' \in \mathbb{R}^n$ sich innerhalb eines (k)-Simplex $S \subset \mathbb{R}^n$ befindet, so kann man dies durch bestimmen der sog. *baryzentrischen Koordinaten* erreichen. Dazu wird folgende Definition benötigt.

Definition 2.6 (Baryzentrische Koordinaten)
Ist $S = [x^{(0)}, \ldots, x^{(k)}]$ *ein*(k)*-Simplex im* $\mathbb{R}^n$ *mit Volumen* $|V_S| > 0$ *, so lässt sich jeder Punkt* $x' \in S$ *eindeutig durch eine Linearkombination der Eckpunkte von* S *darstellen, d.h.*

$$x' = \sum_{i=0}^{k} \lambda_i x^{(i)} \quad \textit{mit} \quad \sum_{i=0}^{k} \lambda_i = 1,\ 0 \le \lambda_i \le 1. \tag{2.10}$$

Die Gewichte $\lambda_0, \ldots, \lambda_k$ *werden* baryzentrische Koordinaten *bzgl. S genannt. Häufig wird auch der Vektor* $\lambda = (\lambda_0, \ldots, \lambda_k)^T$ *als baryzentrische Koordinate bezeichnet.*

Um nun feststellen zu können, ob ein Punkt x' in einem Simplex S enthalten ist, müssen die baryzentrischen Koordinaten berechnet werden. Gilt für alle $\lambda_i > 0$, so ist der Punkt x' in S enthalten, ist mindestens ein $\lambda_i = 0$ und $\lambda_j > 0$ für $i \neq j$, so liegt der Punkt x' auf der konvexen Hülle von S, ansonsten befindet er sich außerhalb von S.

2.2.2 Qualitätsmetriken für Tetraeder

Degenerierte Tetraeder bzw. Dreiecke sind in der Netzgenerierung (s. z.B. [She97, GB98, Lis99, TSW99]) und in der numerischen Mathematik unerwünscht, weshalb gerade in der Netzgenerierung, d.h. der Diskretisierung des Simulationsgebietes Ω, hoher Aufwand durch Anwendung optimierender Qualitätsmetriken betrieben wird, um die Erzeugung dieser degenerierten Objekte zu verhindern. In [BA76] und [Kri92] wurde z.B. gezeigt, dass sehr kleine spitze oder große stumpfe Winkel der Simplizes in $\mathbb{R}^2$ und $\mathbb{R}^3$ maßgeblich die Konditionszahl der mit dem System verbundenen Matrix verschlechtern und somit zu einer schlechteren Konvergenzrate der Lösungsverfahren beitragen. Das gleichseitige Dreieck bzw. das gleichförmige Tetraeder bestimmt hier also das Optimum der Winkel. Leider lässt sich für gewöhnlich nicht jedes Gebiet ausschließlich mit diesen optimalen Objekten diskretisieren. Es ist daher wichtig, ein Qualitätsmaß für die Güte eines Tetraeders im $\mathbb{R}^3$ zu haben, um eine Aussage über die Diskretisierung und damit indirekt über die Lösungsverfahren machen zu können. Im Folgenden werden einige gebräuchliche Qualitätsmaße aufgezeigt, die die geometrischen Eigenschaften von Tetraedern berücksichtigen.

Qualitätsmaße, die ein Verhältnis aus zwei verschiedenen charakterisierenden Eigenschaften eines Tetraeders bilden, nennt man *Aspect-Ratio*. Für solche charakterisierenden Eigenschaften können z.B. Kantenlänge, Volumen, In-/Umkugelradius, Raum- und Flächenwinkel oder Oberflächengröße herangezogen werden. Die folgende Definition gibt das am häufigsten verwendete Aspect-Ratio Qualitätsmaß für Tetraeder an.

DEFINITION 2.7 (ENTARTUNGSMASS, ASPECT-RATIO)
Sei durch $T = [p^{(0)}, \ldots, p^{(3)}]$ *ein Tetraeder im* $\mathbb{R}^3$ *gegeben. Bezeichne* $l_{max}(T) > 0$ *die längste Kante von* T *und* $\rho_{in}(T) > 0$ *den Radius der größten Kugel, die in* T *eingebettet werden kann. Dann definiert*

$$Q(T) := \frac{l_{max}(T)}{\rho_{in}(T)} \tag{2.11}$$

ein Entartungsmaß bzw. Qualitätsmaß für T.

Möchte man also für ein allgemeines Tetraeder T das Qualitätsmaß $Q(T)$ bestimmen, so wird der Inkugelradius benötigt. Dieser kann mit Hilfe des Volumens V_T und den Flächeninhalten A_{F_i} der Dreickesflächen F_i bestimmt werden.

BEMERKUNG 2.8 (FLÄCHENINHALT EINES DREIECKS IM $\mathbb{R}^3$)
Sei durch die drei Punkte $p^{(0)}$, $p^{(1)}$ *und* $p^{(2)}$ *im* $\mathbb{R}^3$ *ein Dreieck* F *gegeben. Dann*

kann der Flächeninhalt A_F *von* F *bestimmt werden durch*

$$A_F = \frac{1}{2}\left|\left((p^{(1)} - p^{(0)}) \times (p^{(2)} - p^{(0)})\right)\right|. \tag{2.12}$$

BEMERKUNG 2.9 (INKUGELRADIUS EINES TETRAEDERS [GB98])
Sei durch $T = [p^{(0)}, \ldots, p^{(3)}]$ *im* $\mathbb{R}^3$ *ein Tetraeder gegeben. Bezeichne* V_T *das Volumen von* T *und* A_{F_i} *den Flächeninhalt des* (2)*-Randsimplex* F_i, $0 \leq i \leq 3$, *von* T. *Dann kann der Radius der Inkugel*[8] $\rho_{in}(T)$ *von* T *bestimmt werden durch*

$$\rho_{in}(T) = \frac{3V_T}{\sum_{i=0}^{3} A_{F_i}}. \tag{2.13}$$

Als ein alternatives Aspect-Ratio Qualitätsmaß zu $Q(T)$ für ein allgemeines Tetraeder T gilt das folgende Maß $Q^*(T)$. Aus numerischer Sicht ist es jedoch aufwändiger zu berechnen und wird daher in der Praxis seltener verwendet.

DEFINITION 2.10 (ALTERNATIVER ASPECT-RATIO)
Sei durch $T = [p^{(0)}, \ldots, p^{(3)}]$ *ein Tetraeder* T *im* $\mathbb{R}^3$ *gegeben. Bezeichne* l_i *die Länge der Kante* i, $0 \leq i \leq 5$. *Ferner sei durch* V_T *das Volumen des Tetraeders* T *gegeben. Dann bezeichnet*

$$Q^*(T) := \frac{\left(\sqrt{\sum_{i=0}^{5} l_i^2}\right)^3}{V_T} \tag{2.14}$$

ein alternatives Aspect-Ratio Entartungsmaß bzw. Qualitätsmaß von T.

In der Literatur wird gerne eine Normierung des Qualitätsmaßes auf das Interval $[0, 1]$ vorgenommen, wobei der Wert Eins für ein gleichförmiges Tetraeder angenommen wird und Null ein degeneriertes Tetraeder kennzeichnet. Dazu müssen die Kehrwerte der Qualitätsmaße aus Definition 2.7 und Definition 2.10 mit einem entsprechenden Proportionalitätsfaktor multipliziert werden.

DEFINITION 2.11 (NORMIERTER ASPECT-RATIO)
Seien das Tetraeder T *und die Qualitätsmaße* $Q(T)$ *und* $Q^*(T)$ *wie in Definition 2.7 bzw. Definition 2.10 definiert. Dann bilden*

$$Q_{ar}(T) := \frac{\alpha}{Q(T)} = \frac{12\,\rho_{in}(T)}{\sqrt{6}\,l_{max}(T)} = \frac{36\,V_T}{\sqrt{6}\,l_{max}(T)\,\sum_{i=0}^{3} A_{F_i}} \tag{2.15}$$

bzw.

$$Q^*_{ar}(T) := \frac{\beta}{Q^*(T)} = \frac{216\,V_T}{\sqrt{3}\left(\sqrt{\sum_{i=0}^{5} l_i^2}\right)^3} \tag{2.16}$$

die Entartungsmaße bzw. Qualitätsmaße von T *auf das Einheitsintervall* $[0, 1]$ *ab, wobei für das gleichmäßige Tetraeder der Wert Eins und für das degenerierte Tetraeder der Wert Null angenommen wird.*

[8] die größte Kugel, die in das Tetraeder eingebettet werden kann, d.h. die Dreiecksflächen als Tangentialflächen besitzt

Generell kann ein *normiertes geometrisches Qualitätsmaß* für ein Tetraeder folgendermaßen definiert werden (vgl. [DLGC98, LJ94b]).

DEFINITION 2.12 (NORMIERTES GEOMETRISCHES QUALITÄTSMASS)
Ein normiertes geometrisches Qualitätsmaß *ist eine stetige Funktion, die die geometrische Form eines Tetraeders bewertet. Sie ist invariant bzgl. Translation, Rotation, Reflexion und einheitlichem Skalieren des Tetraeders. Sie erreicht ihr Maximum beim gleichförmigen Tetraeder und ihr Minimum für ein degeneriertes Tetraeder. Die Funktion besitzt keine lokalen Maxima und Minima außer dem globalen Maximum für das gleichförmige Tetraeder und dem globalen Minimum für ein degeneriertes Tetraeder. Sie ist normiert auf das Einheitsintervall* $[0,1]$ *und nimmt den Wert Eins an für das gleichförmige Tetraeder und Null für ein degeneriertes Tetraeder.*

Neben dem "Standard"-Aspect-Ratio existieren noch weitere Quotienten, die ein Maß für die Qualität eines Tetraeders aufzeigen können. Hierzu zählt z.B. der *Kanten-Quotient* (oder auch *Kanten-Ratio* genannt).

DEFINITION 2.13 (KANTEN-RATIO QUALITÄTSMASS)
Sei durch $T = [p^{(0)}, \ldots, p^{(3)}]$ *ein Tetraeder* T *im* $\mathbb{R}^3$ *gegeben. Bezeichne* $l_{\min}(T)$ *die kürzeste und* $l_{\max}(T)$ *die längste Kante im Tetraeder* T*. Dann definiert*

$$Q_{er}(T) := \frac{l_{\min}(T)}{l_{\max}(T)} \tag{2.17}$$

das Kanten-Ratio Qualitätsmaß *bzgl. dem Tetraeder* T*.*

Ein weiteres Qualitätsmaß kann durch die Radien der Inkugel und der Umkugel definiert werden, welches dann den *Radius-Quotienten* bzw. *Radius-Ratio* bildet. Dazu wird noch der Radius der Umkugel eines Tetraeders benötigt, der mit folgender Bemerkung leicht bestimmt werden kann.

BEMERKUNG 2.14 (UMKUGELRADIUS EINES TETRAEDERS [GB98])
Sei durch $T = [p^{(0)}, \ldots, p^{(3)}]$ *ein Tetraeder im* $\mathbb{R}^3$ *gegeben. Bezeichne* V_T *das Volumen und die Konstanten* a, b, c *die Produkte der Kantenlängen von* T*, wobei*

$$\begin{aligned} a &= |p^{(1)} - p^{(0)}|\,|p^{(3)} - p^{(2)}| \\ b &= |p^{(2)} - p^{(0)}|\,|p^{(3)} - p^{(1)}| \\ c &= |p^{(3)} - p^{(0)}|\,|p^{(2)} - p^{(1)}| \end{aligned}$$

gilt. Dann kann der Radius der Umkugel[9] $\rho_{\text{circ}}(T)$ *von* T *bestimmt werden durch*

$$\rho_{\text{circ}}(T) = \frac{\sqrt{(a+b+c)(a+b-c)(a+c-b)(b+c-a)}}{24\,V_T}. \tag{2.18}$$

[9] die kleinste Kugel, welche das Tetraeder umschließt, d.h. durch alle vier Eckpunkte geht (Delaunay-Eigenschaft)

DEFINITION 2.15 (RADIUS-RATIO QUALITÄTSMASS)
Sei durch $T = [p^{(0)}, \ldots, p^{(3)}]$ *ein Tetraeder* T *im* $\mathbb{R}^3$ *gegeben. Ist durch* $\rho_{in}(T)$ *der Radius der Inkugel und durch* $\rho_{circ}(T)$ *der Radius der Umkugel von* T *gegeben, dann definiert*

$$Q_{rr}(T) := 3\,\frac{\rho_{in}(T)}{\rho_{circ}(T)} \tag{2.19}$$

das Radius-Ratio Qualitätsmaß *bzgl. des Tetraeders* T.

Die Radius-Ratio Metrik bildet für sich schon ein normiertes Qualitätsmaß, da es für ein gleichförmiges Tetraeder den Wert Eins und für ein degeneriertes Tetraeder (Inkugelradius $\to 0$) den Wert Null annimmt.

Jeweils zwei Dreieckseiten eines Tetraeders bilden zusammen einen inneren und einen äußeren Winkel. Diesen Winkel kann man ebenfalls für ein Qualitätsmaß heranziehen. Der innere Winkel wird *Flächenwinkel (engl. dihedral angle)* genannt und ist in einem nicht-degenerierten Tetraeder stets kleiner als 180°. Der Flächenwinkel ist gleich dem Winkel zwischen den beiden Ebenen, die durch die Eckpunkte der zwei Dreiecke gebildet werden, und kann daher über die beiden Normalenvektoren der Ebenen bestimmt werden.

DEFINITION 2.16 (FLÄCHENWINKEL QUALITÄTSMASS)
Sei durch $T = [p^{(0)}, \ldots, p^{(3)}]$ *ein Tetraeder* T *im* $\mathbb{R}^3$ *gegeben. Bezeichne* n_{ij1} *und* n_{ij2} *die beiden Normalenvektoren zu den beiden Dreieckseiten, die die Kante* e_{ij} *zwischen den Eckpunkten* $p^{(i)}$ *und* $p^{(j)}$ *gemeinsam haben. Dann kann der Flächenwinkel* $\phi_{ij}(T)$ *an dieser Kante* e_{ij} *bestimmt werden durch*

$$\phi_{ij}(T) = \pi - \arccos(n_{ij1} \cdot n_{ij2}).$$

Der kleinste Winkel über alle Flächenwinkel im Tetraeder T

$$Q_{da}(T) := \phi_{min}(T) = \min_{0 \le i < j \le 3} \{\phi_{ij}(T)\} \tag{2.20}$$

definiert ein Qualitätsmaß für das Tetraeder T.

Überträgt man die Winkeleigenschaft an einem Eckpunkt eines Dreiecks aus dem $\mathbb{R}^2$ (Verhältnis von Bogenstück zu Radius am Einheitskreis) in den $\mathbb{R}^3$, so erhält man die äquivalente Winkeleigenschaft für den Raum, den sog. *Raumwinkel (engl. solid angle)*.

DEFINITION 2.17 (RAUMWINKEL IM $\mathbb{R}^3$)
Gegeben sei ein Flächenstück F *der Oberfläche einer Kugel* K *im* $\mathbb{R}^3$ *mit Radius* r_K. *Bezeichne* A_F *den Flächeninhalt des Flächenstücks* F. *Dann heißt das Verhältnis*

$$\theta_F = \frac{A_F}{r_K^2}$$

der durch das Flächenstück F *bestimmte* Raumwinkel θ_F.

Die Einheit des Raumwinkels ist der *Steradiant*, abgekürzt *sterad*. Es ist leicht ersichtlich, dass der volle Raumwinkel den Wert 4π besitzt und der Raumwinkel θ_F selbst unanhängig vom Kugelradius r_K ist.

Auf das Tetraeder übertragen bedeutet dies nun, dass der Flächeninhalt des Kugeldreiecks, welches gebildet wird durch die drei Schnittpunkte der Einheitskugel ($r_K = 1$) mit den verlängerten bzw. verkürzten Kanten, ausgehend vom Eckpunkt, an dem der Winkel bestimmt werden soll, diesen Raumwinkel vollständig bestimmt. Zur genauen Berechnung müssen daher die Flächenwinkel an eben diesen Kanten bestimmt werden, da sie gleich den Winkeln im Kugeldreieck sind. Daraus folgt nun (vgl. [Bey81]):

DEFINITION 2.18 (RAUMWINKEL IM TETRAEDER)
Sei durch $T = [p^{(0)}, \ldots, p^{(3)}]$ *ein Tetraeder* T *im* $\mathbb{R}^3$ *gegeben. Bezeichne* $E_i = \{e_{ij} \mid 0 \leq j \leq 3, j \neq i\}$ *die Menge der Kanten, die alle den Eckpunkt* $p^{(i)}$ *gemeinsam haben. Weiter sei* $\phi_{ij}(T)$*, wie in Definition 2.16 angegeben, der Flächenwinkel an Kante* $e_{ij} \in E_i$*. Dann kann der Raumwinkel* $\theta_i(T)$ *am Eckpunkt* $p^{(i)}$ *bestimmt werden durch*

$$\theta_i(T) = \sum_{\substack{j=0 \\ j \neq i}}^{3} \phi_{ij}(T) - \pi. \tag{2.21}$$

Genauso wie beim Dreieck im $\mathbb{R}^2$ gilt, im übertragenen Sinn, für ein nicht degeneriertes Tetraeder T im $\mathbb{R}^3$ die Winkelsummen-Eigenschaft (vgl. [Gad52]), d.h. $0 < \sum_{i=0}^{3} \theta_i(T) \leq 2\pi$. Daraus folgt, wenn ein Raumwinkel in einem Tetraeder T sehr groß ist ($\theta_i \to 2\pi$), dann muss es demnach auch *mindestens einen* sehr kleinen Raumwinkel $\theta_{min}(T)$ in T geben und umgekehrt. Dies führt unmittelbar zu einem weiteren Qualitätsmaß basierend auf dem kleinsten Raumwinkel eines Tetraeders.

DEFINITION 2.19 (RAUMWINKEL QUALITÄTSMASS)
Sei durch $T = [p^{(0)}, \ldots, p^{(3)}]$ *ein Tetraeder* T *im* $\mathbb{R}^3$ *gegeben. Bezeichne* $\theta_i(T)$ *den Raumwinkel am Eckpunkt* $p^{(i)}$ *des Tetraeders* T *gemäß Definition 2.18. Dann definiert*

$$Q_{sa}(T) := \theta_{min}(T) = \min_{0 \leq i \leq 3} \{\theta_i(T)\} \tag{2.22}$$

ein Qualitätsmaß für das Tetraeder T*.*

Aus numerischer Sicht ist die Bestimmung des kleinsten Raumwinkels $\theta_{min}(T)$ nach Definition 2.18 aufwändig, da mehrfach für die Flächenwinkel $\phi_{ij}(T)$ trigonometrische Funktionen verwendet werden. In [LJ94b, DLGC98] wird eine alternative Methode angegeben, die numerisch weniger aufwändig ist, allerdings nur Raumwinkel $\leq \pi$ korrekt bestimmt. Da man aber nur den kleinsten Winkel haben will, ist diese Methode gerechtfertigt.

BEMERKUNG 2.20 (BESTIMMUNG DES KLEINSTEN RAUMWINKELS [LJ94B, DLGC98])
Sei durch $T = [p^{(0)}, \ldots, p^{(3)}]$ *ein Tetraeder* T *im* $\mathbb{R}^3$ *gegeben. Bezeichne* V_T *das Volumen von* T *und* l_{ij} *die Länge der Kante* $e_{ij} \in E_i = \{e_{ij} \mid 0 \leq j \leq 3, j \neq i\}$ *am*

Eckpunkt $p^{(i)}$ *von* T. *Dann kann der kleinste Raumwinkel* $\theta_{min}(T)$ *von* T *bestimmt werden durch*

$$\theta_{min}(T) = \alpha \min_{0 \le i \le 3} \{\theta_i(T)\} \tag{2.23}$$

mit

$$\alpha^{-1} = 6 \arcsin\left(\frac{\sqrt{3}}{3}\right) - \pi$$

und

$$\sin\left(\frac{\theta_i(T)}{2}\right) = \frac{12 V_T}{\sqrt{\prod_{\substack{0 \le j < k \le 3 \\ j,k \neq i}} (l_{ij} + l_{ik} + l_{jk})(l_{ij} + l_{ik} - l_{jk})}} = \sigma_i(T) \tag{2.24}$$

bzw.

$$\sigma_{min}(T) = \beta \min_{0 \le i \le 3} \{\sigma_i(T)\} \tag{2.25}$$

mit

$$\beta^{-1} = \arcsin\left(\frac{\alpha^{-1}}{2}\right) = \frac{\sqrt{6}}{9}.$$

Das *Mittelwert-Verhältnis* oder auch *Mittelwert-Ratio*, das hier als letztes Qualitätsmaß beschrieben wird, stellt eine Verbindung dar zwischen den in diesem Abschnitt behandelten *geometrischen Qualitätsmaßen* und den sog. *algebraischen Qualitätsmaßen*. Zur Bestimmung wird für den algebraischen Teil eine affine Transformationsmatrix M benötigt, die das gleichförmige Tetraeder T_{equ} in das zu betrachtende allgemeine Tetraeder T überführt. Von dieser Transformationsmatrix M werden die Eigenwerte λ_i entsprechend in ein Verhältnis gesetzt, welches dann eine Aussage über die Qualität von T gibt. Die Eigenwerte werden hier aber nicht direkt berechnet, sondern mittels geometrischen Eigenschaften dargestellt (vgl. [DLGC98, LJ94a]).

DEFINITION 2.21 (MITTELWERT-RATIO QUALITÄTSMASS)
Sei ein Tetraeder T *durch* $T = [p^{(0)}, \ldots, p^{(3)}]$ *im* $\mathbb{R}^3$ *gegeben. Bezeichne weiter* $T_{equ} = [r^{(0)}, \ldots, r^{(3)}]$ *das gleichförmige Tetraeder mit gleichem Volumen* V_T *wie* T. *Dann gibt es eine affine Transformation*

$$p^{(i)} = M r^{(\pi(i))} + t \quad \textit{für} \quad 0 \le i \le 3 \tag{2.26}$$

mit Permutation $\pi : \{0,1,2,3\} \to \{0,1,2,3\}$, *Transformationsmatrix* M *und Translationsvektor* t. *Seien* λ_i, $0 \le i \le 2$, *die Eigenwerte von* $M^T M$ *und* l_{ij} *die Länge der Kante zwischen Eckpunkt* $p^{(i)}$ *und* $p^{(j)}$ *von* T. *Dann definiert*

$$Q_{mr}(T) := \frac{3\sqrt[3]{\det(M^T M)}}{\text{Spur}(M^T M)} = \frac{3\sqrt[3]{\lambda_0 \lambda_1 \lambda_2}}{\lambda_0 + \lambda_1 + \lambda_2} = \frac{12\sqrt[3]{9 V_T^2}}{\sum_{0 \le i < j \le 3} l_{ij}^2} \tag{2.27}$$

das Mittelwert-Ratio Qualitätsmaß, *das das geometrische Mittel der Eigenwerte von* $M^T M$ *zum algebraischen Mittel ins Verhältnis setzt.*

Für positive Werte ist das geometrische Mittel stets kleiner oder gleich dem algebraischen Mittel. Dies gilt hier für alle Eigenwerte $\lambda_i \in \mathbb{R}^3$ der symmetrischen Matrix $M^T M$. Daher bildet die Mittelwert-Ratio Metrik $Q_{mr}(T)$ ebenfalls wie die Radius-Ratio Metrik $Q_{rr}(T)$ über dem allgemeinen Tetraeder T ein normiertes Qualitätsmaß.

Die Berechnung *reiner* algebraischer Qualitätsmaße ist ein aufwändiger und komplexer Vorgang und wird in der Praxis gewöhnlich aus Geschwindigkeitsgründen gar nicht oder höchstens approximativ bzw. in vereinfachter, hybrider Form (s. Mittelwert-Ratio) bestimmt. Im weiteren Verlauf dieser Arbeit wird auf reine algebraische Metriken nicht weiter eingegangen. Als Einstieg zu weiterführender Literatur bzgl. algebraischer Qualitätsmaße von Tetraedern kann [Knu01] dienen.

Vergleich geometrischer Qualitätsmaße

Da nun einige verschiedene Metriken bzw. Qualitätsmaße bekannt sind, stellt sich die Frage nach dem Verhalten dieser Metriken zueinander. Dazu dient die folgenden Definition, die die Äquivalenz zweier Metriken bestimmt.

DEFINITION 2.22 (ÄQUIVALENZ GEOMETRISCHER QUALITÄTSMETRIKEN)
Seien $Q_a(T)$ *und* $Q_b(T)$ *zwei verschiedene normierte geometrische Qualitätsmaße zu einem Tetraeder* T. $Q_a(T)$ *und* $Q_b(T)$ *sind genau dann* äquivalent, *wenn es* $\alpha, \beta, \gamma > 0$ *gibt, so dass gilt*

$$\alpha\, Q_a(T)^\gamma \leq Q_b(T) \leq \beta\, Q_a(T)^\gamma. \tag{2.28}$$

Dies bedeutet, wenn eine Metrik bzw. ein Qualitätsmaß für ein Tetraeder eine Degeneration, also einen Wert gegen Null, feststellt, so wird auch eine äquivalente Metrik zum selben Tetraeder eine Degeneration bestimmen, möglicherweiße aber mit einer anderen Geschwindigkeit. Eine Herleitung und einige Beispiele hierzu werden in [LJ94b] ausführlich diskutiert.

Degeneration von Tetraedern

Was bedeutet nun, aus geometrischer Sicht, ein schlecht geformtes Tetraeder? Eine mögliche Definition (vgl. [CDE+99]) geht über die Form der Dreiecke, aus denen das Tetraeder aufgebaut ist. Es gibt genau zwei Dreieckstypen, die kleine (spitze) Winkel besitzen. Der erste Typ hat eine kurze und demzufolge zwei lange Kanten. Er wird *dagger* genannt. Beide langen Kanten schließen einen spitzen Winkel ein. Der zweite Typ, mit *blade* benannt, besitzt keine kurzen Kanten aber dafür eine lange. An dieser langen Kante liegen zwei spitze Winkel an. Aus diesen beiden Dreieckstypen lassen sich nun zwei Klassen von degenerierten Tetraedern konstruieren, deren Volumen sehr nahe bei Null liegen *und auch* spitze Winkel besitzen, so dass z.B. für den Aspect-Ratio gilt $Q_{ar} < \epsilon \ll 1$ mit ϵ nahe bei Null. Tetraeder, die einer dieser beiden Klassen angehören, werden auch *Splitter (engl. sliver)* genannt.

Abbildung 2.6 Beispiele degenerierter Tetraeder mit nahezu kolinearen Eckpunkten

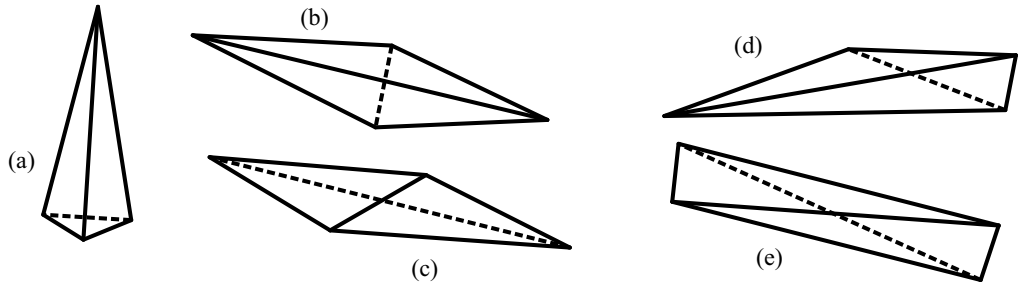
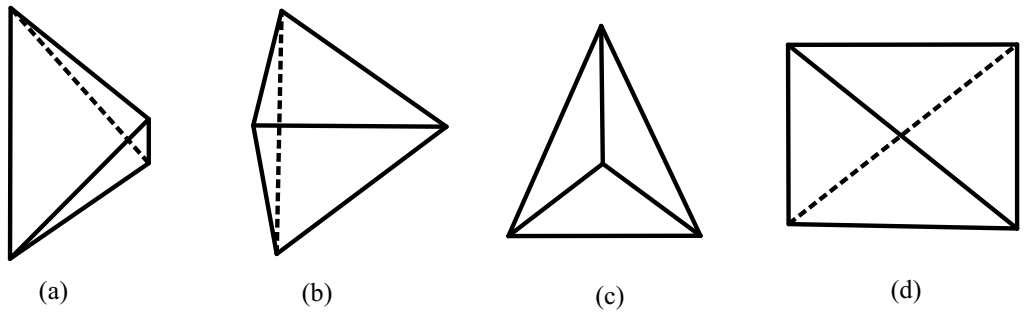


Abbildung 2.7 Beispiele degenerierter Tetraeder mit nahezu koplanaren Eckpunkten



Die erste Klasse der degenerierten Tetraeder kann dadurch konstruiert werden, dass die vier Eckpunkte des Tetraeders nahezu kolinear positioniert werden, d.h. sich alle nahe einer definierbaren Geraden befinden. In Abbildung 2.6 sind die fünf Grundtypen dieser Klasse dargestellt. Ihre Bezeichnungen sind (a) *spire*, (b) *spear*, (c) *spindle*, (d) *spike* und (e) *splinter*. Sie unterscheiden sich alle durch die Anzahl der beiden verwendeten Dreieckstypen.

Die zweite Klasse der degenerierten Tetraeder ist durch die Eckpunkte definiert, die nahezu koplanar im $\mathbb{R}^3$ liegen, d.h. alle Eckpunkte bis auf ein kleines $\epsilon > 0$ im Abstand in einer Ebene liegen. Das Volumen ist auch hier nahezu Null, aber es treten keine übermäßig kurzen oder langen Kanten auf. Abbildung 2.7 zeigt die vier Grundtypen dieser Klasse. Die Typen heißen (a) *wedge*, (b) *spade*, (c) *cap* und (d) *sliver*.

Eine andere Definition für degenerierte Tetraeder (in [BCER95]) geht über die Flächenwinkel benachbarter Dreiecke. Diese Ansatzdefinition wird hier aber nicht weiter verfolgt, da die Klassentypen der schlecht geformten Tetraeder sich gleichen.

Nimmt man nun Tetraeder dieser beiden Klassen als Referenz für die vorgestellten Qualitätsmaße, so stellt sich heraus, dass das Qualitätsmaß der Flächenwinkel Q_{da} und des Kanten-Ratio Q_{er} nicht für alle degenerierten Tetraeder einen minimalen Wert liefern, d.h. einen Wert nahe bei Null. So liefert z.B. der Splinter-Typ für das Flächenwinkel Qualitätsmaß einen guten Wert. Lässt man die Höhe dieses Tetraeders ins Unendliche

Tabelle 2.1 Char. Eigenschaften eines gleichförmigen Tetraeders T_a mit Kantenlänge a

min. Kantenlänge h_{min}	a	max. Kantenlänge h_{max}	a
Höhe h_i	$\frac{1}{3}\sqrt{3}\,a$	Volumen V_{T_a}	$\frac{1}{12}\sqrt{2}\,a^3$
Flächeninhalt Dreieck A_{F_i}	$\frac{1}{4}\sqrt{3}\,a^2$	Oberfläche S_{T_a}	$\sqrt{3}\,a^2$
Inkugelradius ρ_{in}	$\frac{1}{12}\sqrt{6}\,a$	Umkugelradius ρ_{circ}	$\frac{1}{4}\sqrt{6}\,a$
Flächenwinkel ϕ_{ij}	$\arccos(\frac{1}{3})$	Raumwinkel θ_i	$3\arccos(\frac{1}{3})-\pi$

wachsen, oder, was äquivalent dazu ist, die Länge der Kanten auf der Grundfläche zur Höhe gegen Null gehen, d.h. das Tetraeder wandelt sich zu einem Dorn, dann nähern sich die Flächenwinkel an der Grundseite gegen 90° und zur Spitze hin gegen 60°, obwohl das Volumen gegen Null strebt (s. hierzu [DLGC98]). Ähnlich verhält sich das Kanten-Ratio Qualitätsmaß. Wendet man beispielsweise den Sliver-Typ auf das Kanten-Ratio an, so liefert auch dieses Maß einen guten Wert, da die Längen der Kanten nahezu gleich groß sind, das Verhältnis kürzester zu längster Kante also nahe bei eins liegt, aber das Volumen gegen Null strebt. Beide Qualitätsmaße Q_{da} und Q_{er} bilden also nach Definition 2.12 keine *echten* normierten Qualitätsmaßen, obwohl sie eine große Anzahl von degenerierten Tetraedern erkennen können. Folglich sind diese auch nicht nach Definition 2.22 äquivalent zu den anderen Qualitätsmaßen.

Konstruktion eines gleichförmigen Tetraeders

Ein gleichförmiges Tetraeder T_a mit der Kantenlänge a lässt sich leicht mit folgender Konstruktionsvorschrift erzeugen. Die vier Eckpunkte von T_a lauten

$$p^{(0)} = \begin{pmatrix} 0 \\ \frac{\sqrt{3}}{3}a \\ 0 \end{pmatrix}, p^{(1)} = \begin{pmatrix} \frac{1}{2}a \\ -\frac{\sqrt{3}}{6}a \\ 0 \end{pmatrix}, p^{(2)} = \begin{pmatrix} -\frac{1}{2}a \\ -\frac{\sqrt{3}}{6}a \\ 0 \end{pmatrix}, p^{(3)} = \begin{pmatrix} 0 \\ 0 \\ -\frac{\sqrt{6}}{3}a \end{pmatrix}. \tag{2.29}$$

In Tabelle 2.1 sind einige charakteristische Werte für ein gleichförmiges Tetraeder T_a mit Kantenlänge a angegeben.

2.2.3 Tetraedernetze im $\mathbb{R}^3$

Da die Grundelemente eines unstrukturierten Netzes, die Elemente bzw. Tetraeder mit ihren Eigenschaften und Qualitätsbedingungen bekannt sind, kann nun eine Definition eines Tetraedernetzes erfolgen (vgl. [GB98]).

DEFINITION 2.23 (TETRAEDERNETZ IM $\mathbb{R}^3$)
Sei Ω eine abgeschlossene Teilmenge des $\mathbb{R}^3$. Die Menge $M_\Omega = \{T = [p^{(0)}, \ldots, p^{(3)}] \mid p^{(i)} \in \mathbb{R}^3\}$ wird als Tetraedernetz über dem Gebiet Ω bezeichnet, falls gilt

- M_Ω *ist eine endliche Überdeckung von* Ω, *d.h.* $\Omega = \bigcup_{T \in M_\Omega} T$.
- *Jedes Tetraeder* T *aus* M_Ω *ist nicht leer* ($V_T > 0$).
- *Die Schnittmenge der Rauminhalte zweier Tetraeder ist stets leer.*
- *Die Schnittmenge je zweier Tetraeder aus* M_Ω *ist entweder die leere Menge, ein Eckpunkt, eine Kante oder eine Dreiecksfläche.*

In den vorangegangenen Abschnitten wurde jeweils immer ein einzelnes Tetraeder bzgl. seiner geometrischen oder algebraischen Eigenschaften klassifiziert bzw. bewertet. Die Frage, die sich nun stellt, ist, wie kann man ein komplettes diskretisiertes (tetraediertes) Simulationsgebiet Ω bewerten? In der Literatur wird hierfür sehr häufig eine Metrik definiert (vgl. z.B. [She97, GB98, Lis99, TSW99]), welche das schlechteste Tetraeder im gesamten Netz als Maß für die Qualität heranzieht.

DEFINITION 2.24 (QUALITÄTSMASS FÜR EIN TETRAEDERNETZ)
Sei Ω *ein tetraediertes Teilgebiet des* $\mathbb{R}^3$. *Sei weiter* $Q^*(T)$ *ein normiertes Qualitätsmaß für ein Tetraeder* T *gemäß Definition 2.12. Dann definiert*

$$Q_\Omega := \min_{T \in \Omega} Q^*(T) \tag{2.30}$$

ein Maß für die Gesamtqualität von Ω.

Diese Definition reicht alleine aus, um eine Qualitätsbeurteilung für das Gebiet zu erhalten. Der Grund dafür liegt in den Konvergenzkriterien der Löser. Die Konditionszahl der Matrix, die aus dem diskretisierten Tetraedernetz für die unterschiedlichen Teilproblemstellungen generiert wird, hängt alleine von der kleinsten Kante bzw. dem kleinsten auftretenden Winkel und somit vom schlechtesten Tetraeder im Netz ab (vgl. hierzu [BA76, Kri92]).

2.3 Parallele Leistungsmaße

2.3.1 Leistungsbewertung einer parallelen Anwendung

Das grundlegende Leistungsmaß für eine parallele Anwendung bzw. eines parallelen Algorithmus ist die Laufzeit $T_p(n)$, wobei p die Zahl der Prozessoreinheiten (PE, bspw. Prozessoren oder Rechenknoten) und n die Länge bzw. Größe der Verarbeitungseingabe (Daten) bezeichnet. Die Laufzeit $T_p(n)$ setzt sich zusammen (vgl. [RR00]) aus der Zeit für die *Durchführung von lokalen Berechnungen*, der Zeit für die *Ausführung von Kommunikationsoperationen* zwischen den Prozessoreinheiten, der Summe der maximalen *Wartezeiten* während der Kommunikationsoperationen aller Prozessoreinheiten und der Zeit für *Synchronisationen*. Der akkumulierte Zeitaufwand für die lokalen

Berechnungen auf den Prozessoreinheiten entspricht bis auf den zusätzlichen Verwaltungsaufwand für die Parallelverarbeitung der Zeit, die eine sequenzielle Anwendung bzw. Algorithmus aufwendet. Die drei anderen Zeitanteile von $T_p(n)$ treten nur im parallelen Fall auf und werden zusammen mit dem lokalen Verwaltungsaufwand für die Parallelverarbeitung als *paralleler Zusatzaufwand (parallel overhead)* bezeichnet.

Speed-Up Für einen direkten Vergleich eines sequenziellen und eines parallelen Ansatzes für eine Problemstellung wird der *Speed-up* herangezogen:

DEFINITION 2.25 (SPEED-UP EINES PARALLELEN ALGORITHMUS)
Der Speed-up *eines parallelen Algorithmus* A*, der auf einer Problemstellung der Größe* n *mit* p *Prozessoreinheiten arbeitet, ist definiert als*

$$S_p^A(n) := \frac{T^*(n)}{T_p^A(n)}. \tag{2.31}$$

Hierbei bezeichnet $T^*(n)$ *die Laufzeit des besten bekannten* sequenziellen *Algorithmus für die Problemstellung.*

Statt $S_p^A(n)$ kann auch, wenn es aus dem Kontext ersichtlich ist, $S_p(n)$ oder $S_A(p)$ geschrieben werden. Steht $T^*(n)$ für die Bearbeitung einer Problemstellung nicht zur Verfügung (vgl. bspw. [ALO02, RR00]), so wird stattdessen die Ausführungszeit des parallelen Algorithmus mit nur einem Prozessor $T_1^A(n)$ als Approximation für den sequenziellen Zeitaufwand verwendet.

Der als ideale Speed-up bezeichnete Wert für einen Algorithmus A bei konstanter Problemgröße n ist $S_p^A(n) = p$, d.h. der Algorithmus skaliert linear mit der Zahl der Prozessoren. Dies gilt allerdings nur unter der Voraussetzung, dass die Teilprobleme über n ohne Zusatzaufwand beliebig klein werden können, unabhängig[10] von der Zahl p der Prozessoreinheiten. In der Theorie (s. z.B. [RR00]) wird für den Speed-up immer $S_p(n) \leq p$ angenommen. In der Praxis treten allerdings häufig auch Fälle mit $S_p(n) > p$ (superlinearer Speed-up) auf. Dies liegt hautpsächlich darin begründet, dass bei einer bestimmten Anzahl von Prozessoreinheiten die lokalen Problemgrößen für einen Prozessor so klein werden, dass Cache-Effekte die Ausführungszeit stark beeinflussen. In der Praxis gilt aber für einen parallelen Algorithmus wegen des parallelen Overheads für gewöhnlich $S_p(n) < p$.

Inkrementeller Speed-Up Häufig können Problemstellungen in praktischen Anwendungen nicht sequenziell berechnet werden und benötigen eine Mindestanzahl von Prozessoreinheiten. Dies kann z.B. durch die Größe der Problemstellung oder durch die Laufzeit im sequenziellen Fall bedingt sein. Hierfür wird der sog. *inkrementelle Speed-up* als Leistungsmaß eingeführt.

[10] In der Praxis ist diese Voraussetzung allerdings nicht haltbar, da der Verwaltungsaufwand (Kommunikation) für die Teilprobleme bei steigender Prozessorzahl unverhältnismäßig groß wird und den Speed-Up somit direkt beeinflusst.

DEFINITION 2.26 (INKREMENTELLER SPEED-UP EINES PAR. ALGORITHMUS [ALO02])
Der inkrementelle Speed-up *eines parallelen Algorithmus* A*, der auf einer Problemstellung der Größe* n *mit* $p \geq P$ *Prozessoreinheiten (*P *ist kleinste Anzahl an notwendigen Prozessoreinheiten) arbeitet, ist definiert als*

$$S_p^{inc,A}(n) := \frac{T_{p/2}^A(n)}{T_p^A(n)}. \tag{2.32}$$

Der inkrementelle Speed-up setzt also nur die Laufzeiten aus der parallelen Anwendung eines Algorithmus (bei halbierter Prozessoranzahl) ins Verhältnis.

Effizienz Eng verknüpft mit dem Speed-up ist die Effizienz eines Algorithmus. Sie gibt ein Maß für den Anteil der Laufzeit an, den eine Prozessoreinheit für Berechnungen benötigt, die auch im sequenziellen Algorithmus vorhanden sind (s. [RR00]).

DEFINITION 2.27 (EFFIZIENZ EINES PARALLELEN ALGORITHMUS)
Die Effizienz *eines parallelen Algorithmus* A*, der auf einer Problemstellung der Größe* n *mit* p *Prozessoreinheiten arbeitet, ist definiert als*

$$E_p^A(n) := \frac{S_p^A(n)}{p} = \frac{T^*(n)}{p \cdot T_p^A(n)}. \tag{2.33}$$

Hierbei bezeichnet $T^*(n)$ *die Laufzeit des besten bekannten* sequenziellen *Algorithmus für die Problemstellung.*

Amdahls Gesetz Enthält ein paralleler Algorithmus einen Anteil, der nur sequenziell ausgeführt werden kann (z.B. durch zwingende Serialisierung von Algorithmenabschnitten), so lässt sich nach dem *Gesetz von Amdahl* eine obere Schranke für den Speed-up angeben. Wird der sequenzielle (konstante) Anteil im Algorithmus mit f bezeichnet, dann setzt sich die Laufzeit des parallelen Algorithmus zusammen aus $f \cdot T^*(n)$ für den sequenziellen Teil und $\frac{1-f}{p} \cdot T^*(n)$ für den parallelen Teil, wobei $T^*(n)$ die Zeit für den besten (bekannten) sequenziellen Algorithmus angibt. Somit ergibt sich für den Speed-up bei konstanter Problemgröße n und idealer Problemaufteilung (vgl. [ALO02, RR00]):

$$S_p(n) = \frac{T^*(n)}{f \cdot T^*(n) + \frac{1-f}{p} \cdot T^*(n)} = \frac{1}{f + \frac{1-f}{p}} \Rightarrow \lim_{p \to \infty} S_p(n) = \frac{1}{f}. \tag{2.34}$$

2.3.2 Leistungsbewertung eines Parallelrechners

Die Leistungsbewertung eines Parallelrechners, sei es ein einzelner Mehrprozessorrechner (SMP) oder ein massiv paralleles System bestehend aus mehreren hundert oder tausend Rechenknoten (Cluster), wird durch sog. Benchmarkprogramme ermittelt. Benchmarkprogramme sind i.A. spezialisierte Anwendungen, die einen oder mehrere Aspekte des parallelen Systems untersuchen und bewerten. Benchmarkprogramme für Parallelrechner existieren in verschiedenen Kategorien. Die Kategorien (vgl. z.B. [RR00, Sim07]) gliedern sich folgendermaßen: (a) *Low-Level Benchmarks*, (b) *System-Level Benchmarks*, (c) *Kernel-Level Benchmarks* und (d) *Application-Level Benchmarks.*

Low-Level Benchmarks Low-Level Benchmarks sind Programme, die spezielle Eigenschaften der Hardware oder eine Kombination aus Soft- und Hardware untersuchen. Solche speziellen Eigenschaften können bspw. sein: Geschwindigkeit der Verarbeitungseinheiten (Fließkomma, Integer, Pipeline) auf einem Prozessor (z.B. Whetstone [www07g] oder Dhrystone [Wei84]) oder Transferleistung über mehrere Speicherhierarchien (z.B. STREAM-Benchmark [McC07]). Diese Kategorie von Benchmarkprogrammen kann nur über einzelne Komponenten des Rechners eine vergleichende Aussage geben. Die Gesamtleistung des Systems wird hierdurch in keiner Weise verdeutlich.

System-Level Benchmarks System-Level Benchmarks messen das Zusammenwirken mehrerer Komponenten des parallelen Rechensystems. Mit Aussagen dieser Benchmarkkategorie wird das parallele Rechensystem mit seiner Leistung als Ganzes beschrieben und bewertet. Leistung bedeutet hier z.B. Kommunikation (zwischen Rechenknoten oder Prozessoren), Festplatten-IO (z.B. für Datenbanken), Rechenleistung (z.B. für numerische Berechnungen) oder einer Kombination aus diesen (z.B. TPC-Benchmarks [www07f]). Da die Kommunikationsleistung auf parallelen Rechensystemen bei numerischen Anwendungen eine sehr große Rolle spielt, werden in der Praxis üblicherweise spezielle System-Level Benchmarks zum Bewerten herangezogen. Ein bedeutender Benchmark für diese Anwendungsklasse ist z.B. der *INTEL MPI Benchmark* (IMB, [www07b]). Dieser analysiert die Kommunikationsleistung zwischen Rechenknoten über ein (dediziertes) Verbindungsnetzwerk mit Hilfe der standardisierten Kommunikationsschnittstelle *MPI* (Message-Passing-Interface, [SO98, GFB+04]).

Kernel-Level Benchmarks Kernel-Level Benchmarks sind kleine Programmteile (Kernels), die aus meist technisch-wissenschaftlichen Applikationen extrahiert wurden und als Bewertungsmodell dienen. Diese Kernels sind deshalb für eine Bewertung eines parallelen Rechensystems von Bedeutung, da sie in den Applikationen entweder (akkumuliert) den größten Anteil der Laufzeit benötigen oder aber innerhalb der Applikation sehr häufig verwendet bzw. aufgerufen werden. Solche (mathematischen) Kernels können bspw. sein: Berechnung von Fast-Fourier Transformationen, Rechnungen auf schwach besetzten Matrizen (z.B. ILU-/QR-Zerlegung) oder das Lösen von linearen Gleichungssystemen. Zu den bekanntesten Kernel-Level Benchmarks zählen z.B. die *Livermore Loops Benchmarks* [McM86], die *NAS Parallel Benchmarks* [BBB+91] und der *LINPACK Benchmark* [Don87]. Der LINPACK Benchmark dient u.a. zur Bewertung der 500 schnellsten Supercomputer (s. [www07e]). Die Grundlage dieses Benchmarks ist das Lösen eines linearen Gleichungssystems $Ax = b$, wobei die erreichte Geschwindigkeit in GFLOP/s bzw. TFLOP/s (Fließkommaoperationen pro Sekunde) als Bewertungsmaßstab herangezogen wird.

Application-Level Benchmarks Application-Level Benchmarks beinhalten im Gegensatz zu den anderen Kategorien von Benchmarks nicht nur Teile eines Programmes oder Kernels, die spezielle Aspekte der parallelen Rechnerarchitektur unteruschen, sondern eine oder mehrere komplette Anwendungen. Als Grundlage für die Bewertung gilt, dass der verwendete Application-Level Benchmark alle relevanten Aspekte abdeckt. Dies ist

z.B. der Fall, falls die Applikation die einzige Anwendung ist, die auf dem System läuft (bspw. Strömungssimulation in der Auto- oder Luftfahrtindustrie, Datenbanken mit hohem Transaktionsaufkommen im Bankenwesen). Für Rechensysteme, seien es einzelne Rechner oder parallele Rechenarchitekturen, existiert eine Reihe von anerkannten Benchmarkanwendungen, die von der *Standard Performance Evaluation Corporation* (SPEC [www07d]) herausgebracht wird. Eine Benchmarkserie (benchmark suite) der SPEC besteht aus einer Anzahl von Programmen, die einzelne oder mehrere Leistungsaspekte des zu untersuchenden Rechensystems analysiert und in einen numerischen Wert ausdrückt. Die Werte der einzelnen Programme werden dann gewichtet zu einen einzigen numerischen Wert verknüpft, der schließlich die Gesamtleistung des Rechensystems ausdrückt (vgl. hierzu auch [RR00]). Da die Leistungsfähigkeit der (parallelen) Rechensysteme im Laufe der Zeit immer weiter wächst, aktualisiert die SPEC in unregelmäßigen Abständen die Auswahl der Programme innerhalb einer Benchmarkserie, um sie entsprechend dem Leistungssprung anzupassen. Zu den bekanntesten Benchmarkserien der SPEC gehören z.B. (s.a. [www07d]) *CPU2006* (CINT2006 und CFP2006, Prozessorbenchmark), *Viewperf 10* (Graphikbenchmark) sowie *HPC2002* (Benchmark für parallele Systeme), *MPI2007* (Benchmark für Implementierungen des MPI-Standards [SO98, GFB+04]) und *OMP2001* (Benchmark für Anwendungen auf OpenMP-Basis [Boa07, Cha00]).

2.4 Relevante Projekte im Bereich der parallelen numerischen Simulation

In diesem Abschnitt erfolgt ein kurzer Überblick über artverwandte Projekte zum *padfem*^{2}-Projekt (vgl. Abschnitt 4.1). Da im Bereich der numerischen Simulationsumgebungen eine große Anzahl an Projekten existieren, werden hier nur die bekanntesten bzw. wichtigsten Projekte und Programme zusammengefasst. Hierbei wird zwischen akademischer und kommerzieller Anwendung bzw. Ausrichtung unterschieden. Zusätzlich werden die bekanntesten unterstützenden Middleware-Projekte vorgestellt, die als Basis oder Ergänzung zur Entwicklung bzw. Erweiterung von numerischen Simulationen verwendet werden.

2.4.1 Projekte aus dem akademischen Bereich

Die Zielausrichtung numerischer Simulationen bzw. deren Anwendungen und Programme im akademischen Bereich liegen in der Erforschung von neuen Techniken und Verfahren sowie dem Studium neuartiger Modellprobleme. Die folgenden Projekte haben u.a. den akademischen Bereich maßgeblich beeinflusst.

UG (Unstructured Grid) Das UG-Projekt (s. z.B. [BBJ+97, Hei07]) stellt eine modulare numerische Simulationsumgebung für den (massiv) parallelen Einsatz bereit, die zwei-

und dreidimensionale, instationäre Problemstellungen aus den unterschiedlichsten naturwissenschaftlichen Bereichen bearbeiten kann. Der Schwerpunkt der numerischen Behandlung liegt in der Anwendung des Multigrid-Verfahrens. Die Basis für das Datenmanagement innerhalb von UG bildet das eigens entwickelte Abstraktionsmodell *DDD* (vgl. [Bir98]). Es ermöglicht aufgrund seines modularen Aufbaus und komplexen Schnittstellenumfangs eine komfortable Möglichkeit, sequenzielle Verfahren und Algorithmen in parallele zu transformieren. UG kann wegen seines Datenmodells auf unterschiedlichen Netztopologien arbeiten (u.a. Tetraeder-, Quader- und hybride Netzgeometrien).

OpenFOAM (Open Field Operation and Manipulation) OpenFOAM (s. [Ltd07]) stellt ein Baukastensystem für zahlreiche Anwendungsbereiche aus dem Umfeld der partiellen Differentialgleichungen bereit. Es ist modular aufgebaut und bietet eine Reihe von vorgefertigten Lösungsverfahren (Löser und Modelle) für Standardanwendungen. Neben den Lösern bietet OpenFOAM Werkzeuge für Pre- und Postprocessing (z.B. Netzgenerierung, Partitionierung, Visualisierung) an. Ein Bibliothekskonzept ermöglicht es, eigene physikalische Modelle in OpenFOAM umzusetzen oder vorgegebene zu erweitern. OpenFOAM verwendet primär die Finite Volumen Methode (FVM), um partielle Differentialgleichungssysteme im $\mathbb{R}^3$ zu approximieren. Als Netztopologien können hierbei die unterschiedlichsten Arten von Polyederdiskretisierungen herangezogen werden. Das Baukastensystem zeichnet sich u.a. dadurch aus, dass die parallele Anwendung im Design der Software tief verankert und zum Anwender hin abstrahiert wird. Dadurch kann der Anwender sowohl bei der Verwendung bestehender als auch beim Entwurf und der Realisierung eigener Verfahren und Modelle die Aspekte für eine parallele Benutzung vernachlässigen, da diese weitestgehend von OpenFOAM selbst behandelt werden.

ALBERTA ALBERTA (s. [SS05, SSK+07]) steht als Abkürzung für *Adaptive multi-Level finite element toolbox using Bisectioning refinement and Error control by Residual Techniques for scientific Applications*. Es handelt sich bei diesem Projekt ebenfalls um ein Baukastensystem, dessen Schwerpunkt auf dem Design von dimensionsunabhängigen Datenstrukturen und Algorithmen für numerische Simulationen auf der Basis partieller Differentialgleichungen liegt. Dabei unterscheidet ALBERTA Datenstrukturinformationen zu geometrischen Eigenschaften (Topologie), Informationen zu den finiten Elementen (Ansatzfunktionen und weitere Eigenschaften) sowie algebraischen Informationen, die die beiden erstgenannten Informationsstrukturen verbinden. ALBERTA arbeitet hierarchisch adaptiv auf den Netztopologien. Jede Ebene der Hierarchie speichert einen konsistenten Verfeinerungsgrad des Netzes. Mit ALBERTA können Problemstellungen aus dem ein-, zwei- und dreidimensionalen Raum bearbeitet werden. Das Baukastensystem arbeitet nur sequenziell. Es existiert eine Anbindung von ALBERTA an die Middleware-Software DUNE (AlbertaGrid).

DROPS Das DROPS-Projekt (s. [GPRR02]) stellt eine spezialisierte Anwendung für die numerische Simulation von inkompressiblen Fluiden dar und ist somit auf den Be-

reich der Strömungsmechanik begrenzt. Die Simulationsumgebung dient als Werkzeug zur Untersuchung neuer numerischer Verfahren in diesem Bereich. Als Schwerpunkt der numerischen Löser dient der adaptive Multigrid-Ansatz. DROPS arbeitet auf unstrukturierten dreidimensionalen Netzen (Tetraedernetze). Teile des Softwarepaketes sind parallelisiert. Dazu wird als Unterbau die abstrakte Datenmodellbibliothek DDD (s. UG) verwendet.

2.4.2 Projekte aus dem kommerziellen Bereich

Im industriellen Umfeld werden hauptsächlich kommerzielle Simulationsumgebungen eingesetzt. Dies liegt an der garantierten Robustheit und den hohen Genauigkeitsforderung an die verwendeten Algorithmen und Methoden. Die Industrie ist auf solche Eigenschaften bei der Entwicklung von Produkten angewiesen, da sowohl sicherheits-, ressourcen- als auch finanzkritische Anforderungen erfüllt werden müssen. Software aus dem akademischen Bereich kann diese Anforderungen nicht immer erfüllen. Die folgenden Applikationen stellen eine Auswahl aus dem kommerziellen Bereich dar.

ANSYS Die Firma ANSYS, Inc. (s. [ANS07a]), entwickelt und vertreibt eine Reihe von eigenständigen Softwarepaketen zur numerischen Simulation für die unterschiedlichsten Bereiche (Strömungsmechanik, Strukturmechanik, Multiphysics-Anwendungen, etc). Jede dieser Simulationssoftwarepakete stellt eine Komplettumgebung dar. Die Anwendungen arbeiten größtenteils parallel auf strukturierten und unstrukturierten Gittertopologien im Zwei- und Dreidimensionalen. Im Bereich der Strömungsmechanik bietet ANSYS Inc. die bekannten Softwarepakete Fluent und CFX an.

Fluent Fluent (s. [ANS07c]) ist ein spezialisiertes Komplettpaket zur numerischen Simulation von Problemstellungen der Strömungsmechanik. Komplettpaket bedeutet, dass in der Softwareumgebung alle Arbeitsschritte zur Erstellung einer Simulation durchgeführt werden können, d.h. das Preprocessing (Gebietsdefinition, Netzgenerierung, math. Modellbildung), die numerische Simulation selbst und das Postprocessing (Nachbearbeitung, Visualisierung) der Ergebnisse. Fluent arbeitet sowohl sequenziell als auch parallel. Unstrukturierte Gitter stellen in Fluent die Basis der Netztopologie dar. Im Zweidimensionalen verwendet Fluent Drei- und Rechtecke, im Dreidimensionalen Tetraeder, Quader, Prismen und Pyramiden als Elemente zur numerischen Simulation. Eine Vielzahl an mathematischen Lösungsverfahren werden in Fluent angeboten.

CFX CFX (s. [ANS07b]) ist neben Fluent eine weitere, weit verbreitete und angewandte numerische Simulationsumgebung im Bereich der Strömungsmechanik. An CFX können externe Modellierungsprogramme (CAD-Software) zum Geometrieentwurf von Gebieten angebunden werden. Zusätzlich bietet CFX verschiedene Netzgenerierungswerkzeuge zur Diskretisierung an. Als Basis für die numerische Behandlung der Problemstellungen setzt CFX u.a. ein effizient parallelisiertes Multigrid-Verfahren ein. Für die Visualisierung der numerischen Ergebnisse nutzt CFX eine Vielzahl von Techniken,

die von einfachen 3D-Ansichten bis hin zu komplexen Animationen von Strömungsproblemen reichen.

Star-CD Star-CD (s. [Ca07]) ist ein weiteres Werkzeugsystem zur numerischen Simulation von 3D Strömungsmechanikproblemen. Die Applikation importiert Geometriemodelle aus CAE-Anwendungen und diskretisiert selbständig und automatisch die Gebiete für die Simulation. Als Netzelemente können sowohl Tetraeder, Quader und anders geformte Polyedergeometrien verwendet werden. Das mathematische Grundmodell zur Berechnung von partiellen Differentialgleichungen in Star-CD ist die Finite Volumen Methode (FVM).

I-deas Das Softwarepaket I-deas (s. [UPS07]) ist ein Komplettpaket für das Design und die Simulation von Industrieprodukten. Es enthält einen umfangreichen Editor zur Konstruktion von zwei- und dreidimensionalen Modellen, industrie-erprobte Netzgenerierungswerkzeuge, ein komplexes Modul für numerische Simulationen und zahlreiche Visualisierungswerkzeuge zur Analyse der numerischen Ergebnisse. I-deas wird hauptsächlich eingesetzt im Bereich der Strukturmechanik. Als Netzelemente können Dreiecke und Rechtecke sowie Tetraeder und Quader eingesetzt werden. Neben einer umfangreichen Bibliothek von numerischen Lösungsverfahren (u.a. FEM, FVM) ermöglicht I-deas auch, externe Simulationssoftware anzubinden, so dass I-deas auch als reines Pre- und Postprocessing-Werkzeug angewendet werden kann.

2.4.3 Middleware

Als Middleware werden grundlegende oder ergänzende Softwarebibliotheken bzw. Pakete genannt, die einzelne Aspekte in der Entwicklung einer Simulationsumgebung übernehmen. Solche Bibliotheken konzentrieren sich meistens nur auf einen kleinen Teil der verwendeten Funktionalitäten. Es existieren Softwarepakete aus den unterschiedlichsten Bereichen, wie z.B. Netzgenerierung (Diskretisierung), numerische Grundfunktionen (Standardlöser für Gleichungssysteme), Visualisierung (Postprocessing) oder parallele Anwendung (Kommunikation, Partitionierung) usw. Die folgende Auswahl an Projekten zählen zu den am weitest verbreiteten bzw. verwendeten Middlewarepaketen.

DUNE DUNE (s. [BBE+06, BBD07]) steht für *Distributed and Unified Numerics Environment*. Durch dieses Projekt wird eine modulare Middleware zur Verfügung gestellt, deren Entwicklungsschwerpunkt auf der Bereitstellung einer vereinheitlichten Schnittstelle für numerische Simulationsumgebungen liegt. Die zwei wichtigsten Kernmodule behandeln die Datenstrukturen und iterative Lösungverfahren, die auf den Datenstrukturen arbeiten. DUNE stellt eine einheitliche Abstraktionssicht für sämtliche Operationen auf der Netzgeometrie bereit. Dadurch können dimensionsunabhängige Verfahren formuliert werden. DUNE bietet eine parallele Verwendung dieser Verfahren und der Datenstrukturen an. Es existieren eine Reihe von Anbindungen von DUNE an FEM-Simulationsumgebungen, die die Abstraktionsschichten der Datenstrukturen verwenden (z.B. UGGrid, AlbertaGrid, ALUGrid).

Pyramid Pyramid (s. z.B. [NLPT04, NL07]) ist eine Fortran90-Softwarebibliothek zur parallelen Adaption von unstrukturierten Gittern (Tetraedernetze). Die Bibliothek verwendet die MPI-Schnittstelle zur parallelen Ausführung und bietet eine Anbindung an die METIS-Bibliothek zur dynamischen Lastverteilung. Das implementierte Adaptionsverfahren verwendet eine automatische Qualitätskontrolle auf der Basis eines Patternmatching-Verfahrens.

PETSc Die PETSc-Bibliothek (Abkürzung für *Portable, Extensible Toolkit for Scientific Computation*, s. [SBK+07, BBE+07]) ist eine Zusammenstellung von mathematischen Verfahren (z.B. Gleichungslöser) und dazugehörigen Datenstrukturen (z.B. Matrizen und Vektoren) für den (massiv) parallelen Einsatz im Bereich des Scientific Computings. Als Grundlage werden in PETSc die BLAS- und die MPI-Bibliotheken sowie modifizierte Funktionen bekannter Pakete (u.a. LINPACK, MINPACK, SPARSPAK, SPARSEKIT2) verwendet. Darauf aufsetzend stellt PETSc parallele bzw. verteilte und sequenzielle Versionen von Krylov-Unterraum Verfahren und Präkonditionierern sowie Löser für nichtlineare Gleichungen und Unterstützungsmethoden für zeitabhängige partielle Differentialgleichungen (instationäre Lösungen) zur Verfügung. Zusätzlich bietet PETSc eine Anbindungsmöglichkeit für externe, spezialisierte Lösungsverfahren (z.B. Hypre, FFTW, Euclid, Trilinos/ML, etc.).

Trilinos Das Trilinos-Projekt (s. [Lab07, HW07]) ist eine Sammlung von eigenständigen und hocheffizienten numerischen Algorithmen für den Einsatz in parallelen und sequenziellen Systemen. Ebenso wie die PETSc-Bibliothek enthält es verschiedene spezialisierte Funktionen und Routinen, deckt aber ein größeres Anwendungsspektrum ab. Trilinos ist in eine Vielzahl von unabhängigen Unterbibliotheken, sog. Packages, aufgeteilt.

VTK Mit Hilfe der VTK-Bibliothek (Abkürzung für *Visualization Toolkit*, s. [Kit07, SML04]) können Ergebnisse aus numerischen Simulationen zwei- und dreidimensional aufbereitet und visuell dargestellt werden. VTK besteht aus einem Baukastensystem, in dem für die unterschiedlichsten Nachbearbeitungsschritte einer Simulation Funktionen bereitgestellt werden. Die Bibliothek kann eine umfangreiche Menge an topologischen Elementen verarbeiten. Für einen parallelen Einsatz bietet VTK spezialisierte Funktionen und Routinen auf der Basis der MPI- und PThreads-Bibliothek. VTK ist in der Sprache C++ geschrieben. Es existieren für eine Reihen von anderen Programmiersprachen (Python, Java, TCL/TK, etc.) Anbindungen.

3 Parallele Simulation mit partitionslokalen Objektnamensräumen

Inhalt

In diesem Kapitel erfolgt die Einführung eines spezialisierten, verteilten Objektmodells zur parallelen Simulation, welches auf partitionslokalen Namensräumen für die einzelnen Objekttypen basiert. Hierzu wird zu Beginn des Kapitels das Objektmodell formal definiert sowie Definitionen wichtiger Begriffe und Datenstrukturen, die zum Verständnis des Modells notwendig sind, eingeführt. Zur Förderung der Verständlichkeit werden im Verlauf des Kapitels immer wieder Notationen eingeführt und verwendet, mit deren Hilfe die parallelen Algorithmen formuliert werden können. Aufbauend auf dem spezialisierten, verteilten Objektmodell werden die drei wesentlichen Hauptmodule einer parallelen FEM-Simulationsumgebung genauer betrachtet: numerische Algorithmen, geometrische Adaption und Lastverteilung. Von den numerischen Algorithmen werden in dieser Arbeit nur Gleichungslöser und Partitionsrandabgleich untersucht. Dabei liegt der Fokus weniger auf der numerischen Mathematik als viel mehr auf den notwendigen Umformungen und Operationen aus dem verteilten Objektmodell in lokale Strukturen für die numerischen Algorithmen. Die geometrische Adaption im spezialisierten, verteilten Objektmodell stellt einen weiteren Schwerpunkt in diesem Kapitel dar. Hierbei wirkt sich die Struktur des verteilten Objektmodells sehr stark auf die Adaption aus, welche eine Algorithmenstruktur für die geometrische Adaption induziert, die in diesem Kapitel aufgezeigt und analysiert wird. Den dritten und letzten Schwerpunkt in diesem Kapitel bildet die Lastverteilung im verteilten Objektmodell. Auch hier liegt der Fokus weniger bei der Berechnung des Lastausgleichs, als viel mehr bei der Unterstützung der Simulationsumgebung bei der Migration von Daten im spezialisierten, verteilten Objektmodell.

3.1 Objektmodell

3.1.1 Definitionen und Bezeichnungen

Im Folgenden bezeichne T_Ω eine gültige Tetraedierung eines zusammenhängenden, nicht leeren und beschränkten Simulationsgebietes $\Omega \subset \mathbb{R}^3$ mit dem Gebietsrand $\Gamma = \partial\Omega$. Gültigkeit ist hier im Sinne der Definition 2.23 zu betrachten. Ein Punkt $n \in \mathbb{R}^3, n = (x, y, z)^T$, wird Netzknoten von Ω genannt, wenn er Teil der Tetraedierung T_Ω ist. Als Bezeichnung für diese Eigenschaft wird im weiteren Verlauf die Schreibweise $n \in T_\Omega$ benutzt. Die Menge $N_\Omega = \{n \in \mathbb{R}^3 \mid n \in T_\Omega\}$ wird als *Knotenmenge von* T_Ω bezeichnet. Eine ungerichtete Kante $e = \{n_0, n_1\}$, die zwei Netzknoten $n_0, n_1 \in N_\Omega$ verbindet wird als *Netzkante von* T_Ω bezeichnet, falls auch sie Teil der Tetraedierung T_Ω ist, d.h. $e \in T_\Omega$ gilt. Die dazugehörige Kantenmenge ist durch $E_\Omega = \{e \in N_\Omega^2 \mid e \in T_\Omega\}$ definiert. Analog erfolgt die Definition der Netzelemente und der Mengen für Dreiecksflächen $F_\Omega = \{f \in N_\Omega^3 \mid f \in T_\Omega\}$ und Tetraeder $V_\Omega = \{v \in N_\Omega^4 \mid v \in T_\Omega\}$. Eine gültige Tetraedierung wird in diesem Kapitel auch verkürzt mit folgender Tupelschreibweise verwendet: $T_\Omega = (N_\Omega, E_\Omega, F_\Omega, V_\Omega)$. Wird auf eine Menge des Tupels Bezug genommen, so wird dies mit der Schreibweise $\mathbb{K} \in T_\Omega$ mit $\mathbb{K} \in \{N_\Omega, E_\Omega, F_\Omega, V_\Omega\}$ verdeutlicht. Für ein $x \in \mathbb{K}$, wobei $\mathbb{K} \in T_\Omega$ ist, wird x auch als Objekt oder einfach nur als Element

bezeichnet. Knoten, Kante, Fläche und Tetraeder bzw. Volumen selbst beschreiben den sog. *Elementtyp* eines Objektes.

Referenzen und Referenzmengen

Für den weiteren Verlauf wird eine Konstruktionsmöglichkeit benötigt, welche die Elemente einer Tetraedierung aus Grundobjekten aufbaut. Dazu wird eine Funktion $\mathrm{cstr}()$ definiert. Die Funktion $\mathrm{cstr}(o_0, \ldots, o_n)$ konstruiert aus den als Argument übergebenen Objekten $o_0, \ldots, o_n$ Netzelemente höherer Dimension in Abhängigkeit von Typ und Anzahl der Objekte. Die Reihenfolge der Argumente spielt dabei keine Rolle in der Konstruktion. Sind beispielsweise $n_0, n_1 \in N_\Omega$, so erzeugt $\mathrm{cstr}(n_0, n_1) = e$ eine Kante e aus den beiden Konstruktionsknoten n_0 und n_1. Für $f_0, f_1, f_2, f_3 \in F_\Omega$ ergibt $\mathrm{cstr}(f_0, f_1, f_2, f_3) = \nu$ ein Tetraeder ν aus den einzelnen Dreiecksflächen. Für die Funktion $\mathrm{cstr}()$ wird vorausgesetzt, dass die Konstruktionsobjekte o_i auch tatsächlich zueinander passen und ein *korrektes* höherdimensionales Objekt erzeugen. Dies bedeutet, dass z.B. die Schnittmenge der Knotenmenge der Flächen $f_{0 \leq i \leq 3}$ aus dem letzten Beispiel genau vier nicht-identische Knoten enthalten muss.

Netzmodifizierende Algorithmen, wie z.B. Adaptions- oder Migrationsalgorithmen benötigen in vielen Bereichen Konstruktions- und Nachbarschaftsinformationen eines Elementes, um ihre Aufgaben korrekt ausführen zu können. Hierzu erfolgen nun Definitionen zu Hilfsmengen, den sog. Referenzmengen, die die genannten Informationen für ein beliebiges Element $x \in \mathbb{K}$ mit $\mathbb{K} \in T_\Omega$ zur Verfügung stellen.

DEFINITION 3.1 (REFERENZMENGEN EINES KNOTENS)
Sei durch $T_\Omega = (N_\Omega, E_\Omega, F_\Omega, V_\Omega)$ *eine gültige Tetraedierung gegeben. Ist* $x \in N_\Omega$ *ein Knoten aus dieser Tetraedierung, dann definieren folgende Mengen die sog. Referenzmengen bzgl. des Knotens* x*:*

$$\begin{aligned}
\mathcal{R}_N(x) &= \{n \in N_\Omega \mid \mathrm{cstr}(x, n) = e \in E_\Omega\} \\
\mathcal{R}_E(x) &= \{e \in E_\Omega \mid \exists n \in N_\Omega : \mathrm{cstr}(x, n) = e\} \\
\mathcal{R}_F(x) &= \{f \in F_\Omega \mid \exists n_0, n_1 \in N_\Omega : \mathrm{cstr}(n_0, n_1, x) = f\} \\
\mathcal{R}_V(x) &= \{\nu \in V_\Omega \mid \exists n_0, n_1, n_2 \in N_\Omega : \mathrm{cstr}(n_0, n_1, n_2, x) = \nu\}
\end{aligned}$$

Die Elemente der Referenzmengen heißen Referenzen.

$\mathcal{R}_N(x)$ gibt die direkten Nachbarknoten zu einem gegebenen Knoten x an. $\mathcal{R}_E(x)$, $\mathcal{R}_F(x)$ und $\mathcal{R}_V(x)$ definieren die Netzelemente, bei denen der Knoten x Teil der Konstruktion ist, d.h. sie geben die Knotenumgebungskugel erster, zweiter und dritter Dimension von x an.

Analog erfolgt die Definition der anderen Referenzmengen bzgl. einer Kante, Fläche und eines Tetraeders.

DEFINITION 3.2 (REFERENZMENGEN EINER KANTE)
Ist $x \in E_\Omega$ *eine Kante der gültigen Tetraedierung* $T_\Omega = (N_\Omega, E_\Omega, F_\Omega, V_\Omega)$, *dann sind die Referenzmengen bzgl. der Kante* x *definiert durch:*

$$\begin{aligned}
\mathcal{R}_N(x) &= \{\{n_0, n_1\} \subset N_\Omega \mid \mathrm{cstr}(n_0, n_1) = x\} \\
\mathcal{R}_E(x) &= \{e \in E_\Omega \mid \mathcal{R}_N(x) \cap \mathcal{R}_N(e) \neq \emptyset \wedge x \neq e\} \\
\mathcal{R}_F(x) &= \{f \in F_\Omega \mid \exists e_0, e_1 \in E_\Omega : \mathrm{cstr}(e_0, e_1, x) = f\} \\
\mathcal{R}_V(x) &= \{v \in V_\Omega \mid \exists e_0, e_1, e_2, e_3, e_4 \in E_\Omega : \mathrm{cstr}(e_0, e_1, e_2, e_3, e_4, x) = v\}
\end{aligned}$$

Durch $\mathcal{R}_N(x)$ werden die Konstruktionsknoten einer Kante $x \in N_\Omega$ bestimmt, $\mathcal{R}_E(x)$ liefert alle Nachbarkanten von x, die genau einen Knoten mit x gemeinsam haben. Die Mengen $\mathcal{R}_F(x)$ und $\mathcal{R}_V(x)$ besitzen genau die Dreiecksflächen bzw. Tetraeder als Element, bei denen die Kante x ein Konstruktionselement ist. Genauso wie im Fall, dass x ein Knoten ist, bestimmen hier die beiden Mengen die Umgebungskugel zweiter und dritter Dimension.

DEFINITION 3.3 (REFERENZMENGEN EINER FLÄCHE)
Ist $x \in F_\Omega$ *eine Dreiecksfläche der gültigen Tetraedierung* $T_\Omega = (N_\Omega, E_\Omega, F_\Omega, V_\Omega)$, *dann sind die Referenzmengen bzgl. der Fläche* x *definiert durch:*

$$\begin{aligned}
\mathcal{R}_N(x) &= \{\{n_0, n_1, n_2\} \subset N_\Omega \mid \mathrm{cstr}(n_0, n_1, n_2) = x\} \\
\mathcal{R}_E(x) &= \{\{e_0, e_1, e_2\} \subset E_\Omega \mid \mathrm{cstr}(e_0, e_1, e_2) = x\} \\
\mathcal{R}_F(x) &= \{f \in F_\Omega \mid \exists n_0, n_1, n_2, n_3, n_4 \in N_\Omega : \mathrm{cstr}(n_0, n_1, n_2) = x \wedge \\
&\qquad (\mathrm{cstr}(n_0, n_3, n_4) = f \vee \mathrm{cstr}(n_0, n_1, n_3) = f) \wedge x \neq f\} \\
\mathcal{R}_V(x) &= \{v \in V_\Omega \mid \exists f_0, f_1, f_2 \in F_\Omega : \mathrm{cstr}(f_0, f_1, f_2, x) = v\}
\end{aligned}$$

$\mathcal{R}_N(x)$ und $\mathcal{R}_E(x)$ bestimmen die Knoten- bzw. Kantenkonstruktionselemente einer Dreiecksfläche $x \in F_\Omega$. Die Menge $\mathcal{R}_F(x)$ beinhaltet alle Nachbardreiecke zu x, die mindestens einen Konstruktionsknoten gemeinsam haben. In $\mathcal{R}_V(x)$ sind die Tetraeder als Element enthalten, die x als Konstruktionsfläche besitzen. Es gilt stets $0 < |\mathcal{R}_V(x)| \leq 2$. Die Menge enthält genau dann ein Element, wenn das Tetraeder an einem Gebietsrand anliegt. Dies kann sowohl ein echter Rand von Ω aber auch ein durch die Datenpartitionierung künstlich erzeugter Rand sein (Partitionsrand oder Halorand).

DEFINITION 3.4 (REFERENZMENGEN EINES VOLUMENS)
Ist $x \in V_\Omega$ *ein Volumen/Tetraeder der gültigen Tetraedierung* $T_\Omega = (N_\Omega, E_\Omega, F_\Omega, V_\Omega)$, *dann sind die Referenzmengen bzgl. des Volumens* x *definiert durch:*

$$\begin{aligned}
\mathcal{R}_N(x) &= \{\{n_0, n_1, n_2, n_3\} \subset N_\Omega \mid \mathrm{cstr}(n_0, n_1, n_2, n_3) = x\} \\
\mathcal{R}_E(x) &= \{\{e_0, e_1, e_2, e_3, e_4, e_5\} \subset E_\Omega \mid \mathrm{cstr}(e_0, e_1, e_2, e_3, e_4, e_5) = x\} \\
\mathcal{R}_F(x) &= \{\{f_0, f_1, f_2, f_3\} \subset F_\Omega \mid \mathrm{cstr}(f_0, f_1, f_2, f_3) = x\} \\
\mathcal{R}_V(x) &= \{v \in V_\Omega \mid \exists n_{0 \leq i \leq 6} \in N_\Omega : \mathrm{cstr}(n_0, n_1, n_2, n_3) = x \wedge \\
&\qquad (\mathrm{cstr}(n_0, n_4, n_5, n_6) = v \vee \mathrm{cstr}(n_0, n_1, n_4, n_5) = v \vee \\
&\qquad \mathrm{cstr}(n_0, n_1, n_2, n_4) = v) \wedge x \neq v\}
\end{aligned}$$

Auch hier bestimmen die Referenzmengen $\mathcal{R}_N(x)$, $\mathcal{R}_E(x)$ und zusätzlich $\mathcal{R}_F(x)$ die Konstruktionselemente bzgl. Knoten, Kanten und Flächen eines Tetraeders $v \in T_\Omega$. $\mathcal{R}_V(x)$ ist die Menge aller Tetraeder zu x, die mindestens einen Knoten gemeinsam haben, d.h diese Menge bildet die klassische Umgebung/Kugel eines Tetraeders ab.

DEFINITION 3.5 (REFERENZSYSTEM)
Das System $\mathcal{R}_\Omega = (\mathcal{R}_N(\mathbb{K}), \mathcal{R}_E(\mathbb{K}), \mathcal{R}_F(\mathbb{K}), \mathcal{R}_V(\mathbb{K})), \mathbb{K} \in T_\Omega$ *wird als* Referenzsystem *über der Tetraedierung* T_Ω *des Gebietes* Ω *bezeichnet.*

Partitionen

Unter einer Partitionierung einer gültigen Tetraedierung T_Ω bzw. eines Gebietes Ω versteht man eine Aufteilung in n nicht leere und nicht überlappende Teiltetraedierungen $T_{\Omega_{1\leq i\leq n}}$ bzw. Teilgebiete $\Omega_{1\leq i\leq n}$. Für gewöhnlich wird eine Partition auf einen Rechenknoten bzw. einen Rechenprozessor abgebildet, um Parallelverarbeitung auf dem so verteilten Netz zu ermöglichen. Eine gleichmäßige Aufteilung der einzelnen Datenelemente der Teiltetraedierungen ist zumeist vorteilhaft, da der Rechenaufwand für die Teilgebiete ebenfalls gleichmäßig auf alle beteiligten Rechenknoten oder Prozessoren aufgeteilt ist[1]. Diese Aufteilung des Netzes ist Aufgabe der initialen Partitionierung bzw. der Lastverteilung in einer parallelen Simulation, auf die für dieses Objektmodell in Abschnitt 3.4 genauer eingegangen wird.

DEFINITION 3.6 (PARTITIONIERUNG EINES GEBIETES $\mathcal{P}_\Omega$)
Sei durch $T_\Omega = (N_\Omega, E_\Omega, F_\Omega, V_\Omega)$ *eine gültige Tetraedierung eines zusammenhängenden, nicht leeren und abgeschlossenen Gebietes* Ω *gegeben. Das System*

$$\mathcal{P}_\Omega = \{P_i\}_{1\leq i\leq n} = \{(N_{\Omega_i}, E_{\Omega_i}, F_{\Omega_i}, V_{\Omega_i}, \mathcal{R}_{\Omega_i})\}_{1\leq i\leq n} \tag{3.1}$$

bezeichnet eine Partitionierung in n *einzelne Partitionen* P_i *der Tetraedierung* T_Ω. *Die Elementmengen* $N_{\Omega_i} \subset N_\Omega$, $E_{\Omega_i} \subset E_\Omega$, $F_{\Omega_i} \subset F_\Omega$ *und* $V_{\Omega_i} \subset V_\Omega$ *beschreiben dabei den lokalen Anteil der Tetraedierung* T_Ω *für* P_i, $\mathcal{R}_{\Omega_i}$ *das Referenzsystem von* T_Ω *eingeschränkt auf die Elemente aus der Teiltetraedierung* T_{Ω_i}

BEMERKUNG 3.7 (EIGENSCHAFTEN EINER PARTITIONIERUNG)
Für eine n*-Partitionierung* $\mathcal{P}_\Omega = \{P_i\}_{1\leq i\leq n}$ *einer gültigen Tetraedierung* T_Ω *gilt:*

- *Jedes* $v \in V_\Omega$ *muss exakt einer Partition zugeordnet werden können, d.h.*

$$\bigcup_{1\leq i\leq n} V_{\Omega_i} = V_\Omega \wedge V_{\Omega_i} \cap V_{\Omega_j} = \emptyset \; \forall i, j \in \{1, \ldots, n\}, i \neq j. \tag{3.2}$$

- *Sind zwei Partitionen* P_i *und* P_j *benachbart, dann existiert* mindestens ein Partitionsrand *zwischen diesen beiden Partitionen.*

[1] Bei unterschiedlichem Rechenaufwand pro zu bearbeitendem Datenelement ist eine gleichmäßige Aufteilung der Elemente bzgl. des Rechenaufwandes besser geeignet als die reine Aufteilung über die Anzahl der Elemente.

- *Der Partitionsrand kann, muss aber nicht, aus einem Knoten, einer Kante oder aus einer oder mehreren zusammenhängenden Dreiecksflächen bestehen.*

- *Gilt für zwei Partitionen* P_i *und* P_j

$$N_{\Omega_i} \cap N_{\Omega_j} \neq \emptyset \vee E_{\Omega_i} \cap E_{\Omega_j} \neq \emptyset \vee F_{\Omega_i} \cap F_{\Omega_j} \neq \emptyset \tag{3.3}$$

  dann liegen die Elemente der Schnittmenge ausschließlich auf einem oder mehreren Partitionsrändern zwischen P_i *und* P_j*.*

- *Obwohl für das Gebiet* Ω *bzw. die Tetraedierung* T_Ω *gilt, dass es zusammenhängend sein muss, dürfen Partitionen* P_i *nicht zusammenhängend sein.*

Sind zwei Partitionen P_i und P_j benachbart, dann wird dafür verkürzt auch die Schreibweise $P_{i,j}$ benutzt. Mit $\text{neighbor}(P_i) = \{P_j \in \mathcal{P}_\Omega \mid \exists n \in N_{\Omega_i} : n \in N_{\Omega_j}\}$ wird die Menge aller Nachbarpartitionen P_j zu P_i bezeichnet, die Menge $\text{neighbor}^*(P_i) = \{j \in \{1, \ldots, n\} \mid \exists n \in N_{\Omega_i} : n \in N_{\Omega_j}\}$ ist die Menge der Partitionsindizes der Nachbarn zu P_i.

Der Rand einer Partition hat eine hohe Bedeutung für parallel arbeitende Algorithmen. In diesem Objektmodell wird der Rand zwischen einer Partition P_i und P_j durch das Tupel $\partial P_{i,j} = (N^*_{\Omega_{i,j}}, E^*_{\Omega_{i,j}}, F^*_{\Omega_{i,j}})$ dargestellt. Dabei bestehen die Mengen $N^*_{\Omega_{i,j}}$, $E^*_{\Omega_{i,j}}$ und $F^*_{\Omega_{i,j}}$ aus den Objekten, die auf dem Rand zwischen Partition P_i und P_j liegen, d.h.

$$\begin{aligned} N^*_{\Omega_{i,j}} &= \{n \in N_\Omega \mid n \in N_{\Omega_i} \wedge n \in N_{\Omega_j}\} \\ E^*_{\Omega_{i,j}} &= \{e \in E_\Omega \mid e \in E_{\Omega_i} \wedge e \in E_{\Omega_j}\} \\ F^*_{\Omega_{i,j}} &= \{f \in F_\Omega \mid f \in F_{\Omega_i} \wedge f \in F_{\Omega_j}\}. \end{aligned}$$

Da eine Partition P_i durchaus mehrere Nachbarn haben kann, kann der gesamte Rand von P_i beschrieben werden durch $\partial P_i = (N^*_{\Omega_i}, E^*_{\Omega_i}, F^*_{\Omega_i})$ mit den Mengen

$$\begin{aligned} N^*_{\Omega_i} &= \bigcup_{j \in \text{neighbor}^*(P_i)} N^*_{\Omega_{i,j}} \\ E^*_{\Omega_i} &= \bigcup_{j \in \text{neighbor}^*(P_i)} E^*_{\Omega_{i,j}} \\ F^*_{\Omega_i} &= \bigcup_{j \in \text{neighbor}^*(P_i)} F^*_{\Omega_{i,j}}. \end{aligned}$$

Durch $\partial P_i(N_{\Omega_i})$, $\partial P_i(E_{\Omega_i})$ und $\partial P_i(F_{\Omega_i})$ werden die Mengen der Randknoten, -kanten und -flächen einer Partition P_i beschrieben, bei einer expliziten Nachbarpartition P_j sind die Randelementmengen durch $\partial P_{i,j}(N_{\Omega_i})$, $\partial P_{i,j}(E_{\Omega_i})$ und $\partial P_{i,j}(F_{\Omega_i})$ gegeben. Ein *echtes* Gebietsrandobjekt des Netzes liegt zusätzlich in der Menge $\overline{\mathbb{K}_i}$ für $\mathbb{K}_i \in \{N_{\Omega_i}, E_{\Omega_i}, F_{\Omega_i}\}$.

Partitionslokaler Namensraum

Zur eindeutigen Identifizierung der einzelnen Objekte im verteilten Netz bzw. der verteilten Tetraedierung T_Ω, wird eine Abbildungsmöglichkeit benötigt, die die Objekte mit Namen versieht. Dabei sollen die Namen nur jeweils für eine Objekttypmenge eindeutig sein. Ein Name in diesem Objektmodell ist ein Tupel (n, p) aus Zahlen, die so gewählt werden, dass die Zahl n eindeutig innerhalb der Typmenge des Objektes in Partition P_p ist. Das Tupel (n, p) macht damit das partitionslokale Objekt global eindeutig in der Objekttypmenge.

DEFINITION 3.8 (PARTITIONSLOKALE NAMENSABBILDUNG label_p)
Sei $\mathbb{K}_p \in \{N_{\Omega_p}, E_{\Omega_p}, F_{\Omega_p}, V_{\Omega_p}\}$ *eine Objektmenge der Partition* $P_p \in \mathcal{P}_\Omega$. *Durch die Abbildungsvorschrift*

$$\text{label}_{\mathbb{K}_p} : \mathbb{K}_p \to \mathbb{I} \subset \mathbb{N} \times \mathbb{N}; \quad k \mapsto (n, p), \tag{3.4}$$

die jedem Element aus der Objektmenge $\mathbb{K}_p$ *ein Tupel* $(n, p) \in \mathbb{I}$ *zuordnet, wobei* $n \in \mathbb{N}$ *beliebig aber eindeutig in* $\mathbb{I}$ *unter* $\mathbb{K}_p$ *ist, wird eine Identifizierungsabbildung definiert. Das Tupel*

$$\text{label}_p = (\text{label}_{N_{\Omega_p}}, \text{label}_{E_{\Omega_p}}, \text{label}_{F_{\Omega_p}}, \text{label}_{V_{\Omega_p}}) \tag{3.5}$$

heißt partitionslokales Namenabbildungssystem *bzgl. Partition* P_p.

Wenn eindeutig ist, aus welcher Typmenge ein Objekt k ist, auf dem die Namensabbildung angewendet wird, so wird im Folgenden statt $\text{label}_{\mathbb{K}_i}(k)$ verkürzt die Schreibweise $\text{label}_i(k)$ verwendet.

Die Elemente einer Objekttypmenge $\mathbb{K}_i$ innerhalb einer Partition P_i werden also über die Identifizierungsabbildung $\text{label}_{\mathbb{K}_i}$ durch eine beliebige nicht-zusammenhängende bzw. nicht-kontinuierliche Zahlenfolge $(n_j)_{1 \le j \le |\mathbb{K}_i|} \subset \mathbb{N}$ dargestellt, welche durch den Zusatz des Partitionsindexes $(n_j, i)_{1 \le j \le |\mathbb{K}_i|}$ in der gesamten Tetraedierung des Gebietes global eindeutig gemacht werden. Diese Definition des Namensraumes im Objektmodell unterscheidet sich vom herkömmlichen Ansatz der Objektidentifikation zusätzlich dahingehend, dass im herkömmlichen Ansatz die Benennung, d.h. die Namen der Objekte aus einer kontinuierlichen und streng monoton steigenden Zahlenfolge $(n_i)_{1 \le i \le |\mathbb{K}|} = (1, \ldots, |\mathbb{K}|)$ mit $\mathbb{K} \in \{N_\Omega, E_\Omega, F_\Omega, V_\Omega\}$ bestehen muss[2], die zudem noch global eindeutig über alle Partitionen sein müssen[3]. Der herkömmliche Ansatz setzt also auf einer bijektiven Abbildung für die Benennung auf, die im verteilten Objektmodell nicht notwendig ist.

Aus dem verteilten Objektmodell ergibt sich, dass Partitionsrandobjekte lokal gesehen mehrere Namen haben. Hierbei gilt, dass eine Dreiecksfläche $f \in \partial P_{i,j}(F^*_{\Omega_i})$ nur die

[2] Der Grund hierfür ist die notwendige bijektive Abbildungsvorschrift der Objekte zu den Unbekannten für das zu lösende Matrixsystem.

[3] Es kann auch vorkommen, dass für alle Objekttypen der gleiche Namensraum verwendet wird, welches eine zusätzliche Einschränkung bei der Benennung darstellt.

Namen $\text{label}_i(f) = (r, i)$ und $\text{label}_j(f) = (s, j)$, $r, s \in \mathbb{N}$, haben kann, da im dreidimensionalen Fall eine Dreiecksfläche nur gemeinsamer Teil von exakt zwei Partitionen sein kann. Anders sieht es bei Randkanten und -knoten aus. Hier gibt es im Dreidimensionalen keine Einschränkung über die Anzahl der gemeinsamen Partitionen.

Um nun aus einem Identifikationstupel (n, p) die jeweiligen Komponenten, d.h. Objektname und Partitionsindex, zu extrahieren, werden zwei zusätzliche Funktionen benötigt. Mittels

$$\text{id} : \mathbb{I} \subset \mathbb{N} \times \mathbb{N} \to \mathbb{N}; \quad (n, p) \mapsto n$$

bzw. durch

$$\text{pid} : \mathbb{I} \subset \mathbb{N} \times \mathbb{N} \to \mathbb{N}; \quad (n, p) \mapsto p$$

werden diese Komponenten aus einem Identifikationstupel bestimmt.

Deadlock-freie Kommunikation

Eines der grundlegenden Probleme bei parallelen Algorithmen bzw. verteilter Datenhaltung ist die Bestimmung einer *optimalen Kommunikationstopologie*, um die Kosten für die Kommunikation möglichst gering halten zu können. In parallel arbeitenden Algorithmen spielt daher der Aufwand für Kommunikation bzw. Synchronisation zwischen den Rechenknoten eine große Rolle. Bei unstrukturierten Netzen ist zudem zu beachten, dass kein regelmäßiges Kommunikationsmuster bestimmbar ist, weil sich die Partitionsränder durch netzmodifizierende Adaptions- und Lastverteilungsphasen stets ändern können. Partitionen, die vor einer solchen Phase noch benachbart waren, können danach durch eine oder mehrere weitere Zwischenpartitionen voneinander getrennt sein. Daraus folgt eine sich ständig ändernde Topologie für die Kommunikation während der Laufzeit. Diese kann am Besten durch einfache Punkt-zu-Punkt Kommunikationsoperationen bewerkstelligt werden. Die Problemstellung, die initial gelöst werden muss, lautet demnach, einen deadlock-freien Kommunikationsschedule zu bestimmen, der mit möglichst vielen, gleichzeitigen Punkt-Zu-Punkt Kommunikationsoperationen die Länge des Schedules, d.h. die Länge des *kritischen Kommunikationspfades*, minimiert.

DEFINITION 3.9 (KOMMUNIKATIONSSCHEDULE $\mathcal{C}(\mathcal{P}_\Omega)$)
Gegeben sei eine Partitionierung $\mathcal{P}_\Omega$ einer gültigen Tetraedierung T_Ω. Ein Kommunikationsschedule $\mathcal{C}(\mathcal{P}_\Omega)$ *zur Partitionierung $\mathcal{P}_\Omega$ ist eine Menge von* partitionslokalen Kommunikationsphasen C_i *mit*

$$\mathcal{C}(\mathcal{P}_\Omega) = \{C_i\}_{1 \le i \le n} = \{(s, p) \mid s, p \in \mathbb{N} \wedge P_p \in \text{neighbor}(P_i)\}_{1 \le i \le n} \tag{3.6}$$

für alle n Partitionen der Partitionierung. Ein Tupel $(s, p) \in C_i$ gibt dabei an, dass Partition P_i in Schritt s der parallel laufenden Kommunikation mit Partition P_p Daten mittels einer Punkt-Zu-Punkt Kommunikationsoperation austauscht.

Algorithmus 3.1 : Berechnung eines deadlock-freien Kommunikationsschedules $\mathcal{C}(\mathcal{P}_\Omega)$

Eingabe : Partitionierung $\mathcal{P}_\Omega = \{P_i\}_{1 \le i \le n}$
Ausgabe : Kommunikationsschedule $\mathcal{C}(\mathcal{P}_\Omega) = \{C_i\}_{1 \le i \le n}$

```
1  proc CommTopology begin
2      for i ← 1 to n do
3          C_i ← ∅; npid_i = ∅;
4          foreach P_j ∈ neighbor(P_i) do npid_i ← npid_i ∪ {j} ;
5      end
6      Konstruiere Graph G = (V, E): V = {1, ..., n} und E = {{i, j} | j ∈ npid_{1≤i≤n}};
7      step ← 1;
8      while E ≠ ∅ do
9          V' ← V; V* ← ∅; E* ← ∅;
10         while ∃u : u = deg_max(V') ∧ u ∉ V* do
11             v ← deg_max(u);
12             V* ← V* ∪ {u, v}; E* ← E* ∪ {{u, v}};
13         end
14         foreach (i, j) ∈ E* do
15             C_i ← C_i ∪ {(step, j)};
16             C_j ← C_j ∪ {(step, i)};
17         end
18         E ← E \ E*;
19         step ← step + 1;
20     end
21 end proc
```

Die Berechnung eines Schedules auf der Basis einer Punkt-Zu-Punkt Kommunikation kann auf die Bestimmung maximaler Paarungen (maximales Matching) im Partitionsgraphen zurückgeführt werden, wobei der Graph keine besonderen Eigenschaften aufweist, also z.B. weder bipartit noch planar sein muss[4]. Da der zu bestimmende Schedule möglichst optimal, d.h. die Länge des Schedules möglichst kurz sein soll, bedarf es einer besonderen Strategie bei der Konstruktion.

Es gibt mehrere Möglichkeiten, ein Matching in einem Graphen zu finden. Die in Algorithmus 3.1 für das Objektmodell verwendete Methode basiert auf einem Greedy-Verfahren bzgl. Maximierung der Knotengrade. Sie wurde gewählt, da sie einfach zu konstruieren ist und eine kurze Laufzeit besitzt, dafür allerdings nicht immer den optimalen Schedule berechnet. Andere Verfahren zur Berechnung eines maximalen Matchings sind z.B. die Algorithmen von Edmonds [Edm65], von Micali und Vazirani [MV80, Vaz94, PL88] sowie von Blum [Blu90].

[4] Dies ist dadurch begründet, dass die Partitionierung einer Tetraedierung T_Ω in $\mathbb{R}^3$ einen allgemeinen Graphen induziert, an den keine besonderen Anforderungen gestellt werden.

Algorithmus 3.2 : Generischer Datenaustauschalgorithmus einer Partition P_i

Eingabe : Kommunikationsschedule $\mathcal{C}(\mathcal{P}_\Omega) = \{C_i\}_{1 \leq i \leq n}$ bzw. lokaler Anteil C_i, lokales Datenpaket d_i auf Partition P_i

Ausgabe : Datenmenge $D_i = \{d_j \mid P_i, P_j \text{ sind benachbart}\}$

```
1  parallel proc Exchange begin
2      s_max ← max{s_ji | (s_ji, p_ji) ∈ C_j}_{1≤j≤n};
3      D_i ← ∅;
4      for s ← 1 to s_max do
5          if ∃s_ik : (s_ik, j) ∈ C_i ∧ s_ik = s then
6              if i < j then send_j(d_i); receive_j(d_j);
7              else receive_j(d_j); send_j(d_i);
8              D_i ← D_i ∪ {d_j};
9          end
10     end
11 end proc
```

Algorithmus 3.1 bestimmt einen gültigen, deadlock-freien Kommunikationsschedule zu einer gegebenen Partitionierung $\mathcal{P}_\Omega$. Die optimierende Eigenschaft bei der Konstruktion des Schedules besteht in der Auswahl der Kanten zwischen zwei potentiellen Kommunikationspartnern, die Knoten im Graphen mit maximalen Knotengrad verbindet. Die Funktion $\deg_{max}()$ in Zeile 10 und 11 bestimmt dazu für eine gegebene Menge V den Knoten $u \in V$, der den größten Knotengrad besitzt. Für einen expliziten Knoten $u \in V$ bestimmt $\deg_{max}(u)$ einen zu u adjazenten Knoten v mit maximalen Knotengrad.

Die while-Schleife in Zeile 10 bestimmt nun im Partitionsgraphen, der in den Zeilen 2 bis 6 konstruiert wurde, alle Knotenpaare mit der Eigenschaft, dass die Paare im Ursprungsgraphen zusammen einen maximalen Knotengrad gegenüber den restlichen noch nicht betrachteten Knoten haben. Alle diese Knotenpaare können nun in einem Schritt deadlock-frei miteinander kommunizieren.

Die Paare werden mit ihrem gefundenen Zeitpunkt $step$ in die Mengen der partitionslokalen Kommunikationsphasen C_i bzw. C_j in Zeile 15 bis 18 eingefügt. Anschließend müssen die Paare aus dem Graphen entfernt werden, um die Kommunikationpartner des nächsten Schrittes zu bestimmen. Der Algorithmus terminiert genau dann, wenn alle notwendigen Paare gefunden wurden, d.h. die Kantenmenge des Ursprungsgraphen leer ist. Da eine Partition aber immer mindestens eine Nachbarpartition im parallelen Fall hat (Ω ist zwingend zusammenhängend) terminiert Algorithmus 3.1 für beliebige Partitionierungen $\mathcal{P}_\Omega$.

Ist zu einer Partitionierung $\mathcal{P}_\Omega$ ein deadlock-freier Kommunikationsschedule $\mathcal{C}(\mathcal{P}_\Omega)$ durch Algorithmus 3.1 bestimmt worden, erfolgt der eigentliche Austausch von Daten auf der Basis des Algorithmus 3.2. Dieser Algorithmus benötigt auf einer Partition P_i nur den lokalen Anteil C_i des Schedules. Jede Partition muss diesen Algorithmus zu

definierten gleichen Zeitpunkten ausführen. Das Ziel ist es, einen reihenfolgen-korrekten und kostengünstigen[5] Austausch der partitionslokalen Datenpakete mit den Nachbarpartitionen zu erreichen. Im Algorithmus wird dazu in einer Schleife, die über die maximale Länge des Schedules $\mathcal{C}(\mathcal{P}_\Omega)$ geht (Zeile 2), der Kommunikationspartner im aktuellen Schritt bestimmt (Zeile 5). Um die Deadlock-Freiheit zu garantieren, erfolgt eine Unterscheidung bzgl. der sender-initiierenden[6] Partition in den Zeilen 6 und 7.

Algorithmus 3.2 dient als generische Grundlage für alle folgenden Algorithmen, in denen Partitionen Informationen austauschen müssen. Die Laufzeit des Algorithmus wird maßgeblich von der Länge des Kommunikationsschedules $\mathcal{C}(\mathcal{P}_\Omega)$ und damit von der Qualität der Partitionierung $\mathcal{P}_\Omega$, d.h. von der Anzahl der Nachbarn und der Form bzw. der Anordnung der Partitionen zueinander, beeinflusst.

Halo

Der Halo (vgl. hierzu Abschnitt 2.1) ist für viele Algorithmen bei parallelen, numerischen Anwendungen ein zentrales Konzept zur Verarbeitung und Berechnung verteilter Netzdaten. In dem in dieser Arbeit entwickelten, verteilten Objektmodell spielt der Halo bzw. die Struktur des Halos bei numerischen Gleichungslösern und Abgleichalgorithmen eine wichtige Rolle.

DEFINITION 3.10 (HALOSYSTEM $\mathcal{H}_i$ EINER PARTITION P_i)
Sei durch $\mathcal{P}_\Omega = \{P_i\}_{1\leq i\leq n}$ *eine Partitionierung einer Tetraedierung* T_Ω *gegeben. Das System*

$$\mathcal{H}_{1\leq i\leq n} = \{H_{i,j}\}_{1\leq i\leq n} = \{(H_{i\to j}, H_{j\to i})\}_{1\leq i\leq n} \tag{3.7}$$

definiert das Halosystem $\mathcal{H}_i$ *einer Partition* P_i *zu seinen Nachbarpartitionen* $P_j \in \text{neighbor}(P_i)$. *Dabei beschreibt das Tupel* $(H_{i\to j}, H_{j\to i})$ *den lokalen Anteil* $H_{i\to j}$ *sowie den entfernten bzw. kopierten Anteil* $H_{j\to i}$ *am Gesamthalo von Partition* P_i *mit Nachbar* P_j. *Die Anteile selbst sind Tupel aus Elementmengen* $(N \subset N_\Omega, E \subset E_\Omega, F \subset F_\Omega, V \subset V_\Omega)$. *Die Elementmengen des lokale Anteils* $H_{i\to j}$ *enthalten nur Elemente aus Partition* P_i, *der entfernte Anteil Elemente aus* P_j. *Der Rand* $\partial P_{i,j}$ *selbst ist nicht Teil des Halos* $H_{i,j}$.

Die Elementmengen des lokalen und entfernten Haloanteils enthalten genau die Elemente, die auf dem Partitionsrand zwischen den beiden Partition P_i und P_j genau einen Konstruktionsknoten besitzen. Algorithmus 3.3 zeigt die Konstruktion der beiden Anteile des Halosystems $\mathcal{H}_i$ zu einer Partition P_i. Im Algorithmus wird für jeden Nachbarn P_j zu P_i der Partitionsrand bzgl. der daran "hängenden" Tetraeder analysiert (Zeile 5), um aus diesen Tetraedern die Konstruktionselemente zu extrahieren. Diese

[5] Kostengünstig ist hier im Sinne von sparsamen Gebrauch von Netzwerkressourcen definiert, bspw. durch geeignete Kommunikationsoperationen, die asynchron/puffernd arbeiten.

[6] In einer Implementierung des Algorithmus können, je nach verwendeter Kommunikationsbibliothek, unterschiedliche und deadlock-freie Operationen verwendet werden, z.B. MPI_SendRecv() statt der einzelnen MPI_Send() und MPI_Recv() Operationen.

Elemente werden dann zu den Elementmengen des lokalen Haloanteils $H_{i\to j}$ hinzugefügt. Als Hilfmittel zur Bestimmung der Elemente dienen die Referenzmengen über einem Tetraederelement. Zu beachten ist hier, dass Konstruktionselemente der betrachteten Tetraeder, die auf dem Partitionsrand liegen, nicht zum Halo zählen. Die Menge dieser Elemente ist alleine durch $\partial P_{i,j}(\mathbb{K})$ mit $\mathbb{K} \in \{N_{\Omega_i}, E_{\Omega_i}, F_{\Omega_i}, V_{\Omega_i}\}$ gegeben.

Informationen über den entfernten Anteil $H_{j\to i}$ werden von der Partition P_i nicht selbst berechnet, ihr fehlen zu Beginn die nötigen Daten am Rand. Stattdessen erfolgt ein Datenaustausch über die rein lokalen Anteile mittels des vorher berechneten Kommunikationsschedules $\mathcal{C}(\mathcal{P}_\Omega)$ (Zeile 13, s.a. Algorithmus 3.2). Nach dem Austausch besitzt die Partition P_i alle entfernten Haloanteile ihrer Nachbarpartitionen P_j, die Nachbarn haben ihrerseits den lokalen Anteil von P_i erhalten. Es bleibt als Letztes die Integration der einzelnen Anteile in das Halosystem bzw. der entfernten Anteile in die verteilte Netzdatenstruktur (Zeile 14 und 15). Die Operation $\oplus$ bedeutet hier eine Integration der empfangenen Elementdaten durch Erzeugung von Elementen mit eigenen neuen Namen, d.h. ihre Benennung ist innerhalb der Partition P_i eindeutig.

Durch das Hinzufügen der entfernten Haloanteile $H_{j\to i}$ zu einer Partition P_i mittels Erzeugung neuer, partitionslokaler Objekte, entstehen weitere Anteile der Teiltetraedierung T_{Ω_i}, die sich, neben dem eigentlichen Partitionsrand $\partial P_{i,j}$, noch mit weiteren

Algorithmus 3.3 : Konstruktion der partitionslokalen Halos $H_{i,j}$ auf Partition P_i

Eingabe : Partition $P_i = (N_{\Omega_i}, E_{\Omega_i}, F_{\Omega_i}, V_{\Omega_i}, \mathcal{R}_{\Omega_i}) \in \mathcal{P}_\Omega$,
lokaler Kommunikationsschedule $C_i \in \mathcal{C}(\mathcal{P}_\Omega)$
Ausgabe : Halosystem $\mathcal{H}_i = \{H_{i,j}\} = \{(H_{i\to j}, H_{j\to i})\}$ der Partition P_i mit ihren Nachbarpartitionen $P_j \in \text{neighbor}(P_i)$

```
1  parallel proc ConstructHalo begin
2      H_i ← ∅;
3      foreach j ∈ neighbor*(P_i) do
4          L_V ← ∅; L_F ← ∅; L_E ← ∅; L_N ← ∅;
5          foreach ν ∈ R_V(n) : n ∈ ∂P_{i,j}(N_{Ω_i}) do
6              L_V ← L_V ∪ {ν};
7              L_F ← L_F ∪ {f ∈ F_{Ω_i} | f ∈ R_F(ν) ∧ f ∉ ∂P_{i,j}(F_{Ω_i})};
8              L_E ← L_E ∪ {e ∈ E_{Ω_i} | e ∈ R_E(ν) ∧ e ∉ ∂P_{i,j}(E_{Ω_i})};
9              L_N ← L_N ∪ {n ∈ N_{Ω_i} | n ∈ R_N(ν) ∧ n ∉ ∂P_{i,j}(N_{Ω_i})};
10         end
11         H_{i→j} ← {(L_N, L_E, L_F, L_V)};
12     end
13     Datenaustausch mit allen Nachbarn P_j: {H_{j→i}} ← Exchange ({H_{i→j}}, C_i);
14     foreach j ∈ neighbor*(P_i) do H_i ← H_i ∪ {(H_{i→j}, H_{j→i})};
15     Integration der entfernten Haloanteile: P_i ← P_i ⊕ {H_{j→i}};
16 end proc
```

Teilen aus der Nachbarpartition P_j überlappen. Daraus folgt für diese Objekte, dass sie, global betrachtet, mehrdeutige Namen besitzen und daher für Algorithmen, die das Halokonzept nutzen, eine gesonderte Behandlung bzgl. der Namensauflösung benötigen.

Das Gegenstück zur Konstruktion des Halosystems $\mathcal{H}_i$ zu einer Partition P_i ist ein Algorithmus DestructHalo(), der den umgekehrten Weg geht, d.h. er entfernt die Haloanteile $H_{j\to i}$ aus P_i und setzt $\mathcal{H}_i \leftarrow \emptyset$. Aufgrund seines einfachen Ablaufs wird auf diesen Algorithmus hier jedoch nicht weiter eingegangen.

Verteiltes, konsistentes Tetraedernetz

Um nun ein komplettes, verteiltes Netz im Objektmodell angeben zu können, wird noch eine Abbildung benötigt, die eine eindeutige Auflösung der mehrdeutigen Namen von Partitionsrand- und Haloobjekten gewährleisten kann.

DEFINITION 3.11 ($\omega_{\mathbb{K}_i}$-ABBILDUNG EINER PARTITION P_i)
Sei $\mathbb{K}_i \in \{N_{\Omega_i}, E_{\Omega_i}, F_{\Omega_i}, V_{\Omega_i}\}$ *eine Objektmenge der Partition* $P_i \in \mathcal{P}_\Omega$. *Die Abbildung* $\text{label}_{\mathbb{K}_i}$ *definiere den partitionslokalen Namensraum über* $\mathbb{K}_i$ *von* P_i. *Durch die Abbildungsvorschrift*

$$\omega_{\mathbb{K}_i} : \mathbb{K}_i \times \mathbb{N} \to \mathbb{N}_0; \quad (k, j) \mapsto n \tag{3.8}$$

mit

$$\omega_{\mathbb{K}_i}(k,j) = \begin{cases} \text{id}(\text{label}_{\mathbb{K}_i}(k)) & : \quad i = j \\ \text{id}(\text{label}_{\mathbb{K}_j}(k)) & : \quad i \neq j \wedge (k \in \partial P_{i,j}(\mathbb{K}_j) \vee k \in H_{j\to i}) \\ 0 & : \quad \text{sonst} \end{cases} \tag{3.9}$$

wird die partitionslokale $\omega_{\mathbb{K}_i}$-Abbildung *bzw.* ω_i-*Abbildung definiert, die zu einem Netzelement* k *der Partition* P_i *den eindeutig identifizierenden Namen in* P_i *bzw. zu einer Nachbarpartition* $P_j \in \text{neighbor}(P_i)$ *bestimmt.*

Gilt $\omega_{\mathbb{K}_i}(k,j) = 0$, so ist T_{Ω_i} keine gültige Teiltetraedierung, die durch $P_i \in \mathcal{P}_\Omega$ induziert wird. Der Wert 0 stellt also einen Fehlerwert dar, der eine besondere Behandlung in den parallelen Algorithmen benötigt. Wenn aus dem Kontext des Algorithmus klar ist, aus welcher Objektmenge $\mathbb{K}_i$ das Element k ist, so kann statt $\omega_{\mathbb{K}_i}(k,j)$ auch vereinfacht $\omega_i(k,j)$ geschrieben werden. Werden ferner lokale Objekte einer Partition P_i identifiziert, dann kann stattdessen $\omega_{\mathbb{K}_i}(k)$ oder einfach nur, gemäß vorhergehender Vereinbarung, $\omega_i(k)$ verwendet werden.

Mit der $\omega_{\mathbb{K}_i}$-Abbildung als letztes Element sind nun alle notwendigen Modellstrukturen definiert, um ein verteiltes, konsistentes Tetraedernetz im Objektmodell beschreiben zu können.

DEFINITION 3.12 (VERTEILTES, KONSISTENTES NETZ $\mathcal{M}_\Omega$)
Sei durch $T_\Omega = (N_\Omega, E_\Omega, F_\Omega, V_\Omega)$ *eine gültige Tetraedierung eines nicht-leeren, beschränkten und zusammenhängenden Gebietes* $\Omega \subset \mathbb{R}^3$ *gegeben. Ferner sei durch*

$\mathcal{P}_\Omega = \{P_i\}_{1\leq i\leq n} = \{(N_{\Omega_i}, E_{\Omega_i}, F_{\Omega_i}, V_{\Omega_i})\}_{1\leq i\leq n}$ *eine* n*-Partitionierung von* T_Ω *in Teiltetraedierungen* $T_{\Omega_{1\leq i\leq n}}$ *gegeben. Bezeichne weiter* $\mathcal{H}_\Omega = \{\mathcal{H}_i\}_{1\leq i\leq n}$ *das Halosystem über* $\mathcal{P}_\Omega$ *und* $\omega_\Omega = \{\omega_{\mathbb{K}_i}\}_{1\leq i\leq n}, \mathbb{K}_i \in \{N_{\Omega_i}, E_{\Omega_i}, F_{\Omega_i}, V_{\Omega_i}\}$*, das Abbildungssystem der lokalen Namensräume für die Partitionierung* $\mathcal{P}_\Omega$*. Durch*

$$\mathcal{M}_\Omega(n) = (\mathcal{P}_\Omega, \mathcal{H}_\Omega, \omega_\Omega) \tag{3.10}$$

wird ein verteiltes, *aus* n *Einzelpartitionen bestehendes* Tetraedernetz *mit partitionslokalen Namensräumen der Netzelemente beschrieben, dessen Zusammenhangstruktur über das Halosystem* $\mathcal{H}_\Omega$ *definiert ist. Das verteilte Netz ist* konsistent, *wenn die Gültigkeit der Tetraedierung auch über die Partitionsgrenzen* $\{\partial P_i\}_{1\leq i\leq n}$ *und die Halos* $\{\mathcal{H}_i\}_{1\leq i\leq n}$ *hinweg gilt, d.h* $\omega_{\mathbb{K}_i}(k,j) \neq 0 : \forall k \in \mathbb{K}_{i,j}$ *für* $\omega_{\mathbb{K}_i} \in \omega_\Omega$.

Das Konstrukt $\mathcal{M}_\Omega(n)$ und die Kurzform $\mathcal{M}_\Omega$ bzw. die für eine lokale Partition P_i sichtbaren Anteile $\mathcal{M}_{\Omega_i}(n) = (P_i, \mathcal{H}_i, \omega_{\mathbb{K}_i})$ dienen in den folgenden parallelen Algorithmen als Eingabe zur Beschreibung der Verteilung und Struktur des Tetraedernetzes.

3.1.2 Datenstrukturen

Die Umsetzung des verteilten Objektmodells in reale Datenstrukturen für die Implementierung bedarf besonderer Beachtung, da im Objektmodell zwar endliche Mengen von Objekten behandelt werden, der Namensraum dieser Objekte aber beliebig groß ist, d.h. $\text{label}_{\mathbb{K}_i}(k) = (n, i)$ mit $n \in \mathbb{N}$ ist beliebig aber eindeutig unter $\mathbb{K}_i$ (s. Definition 3.8). Desweiteren ist im Modell die Anzahl der Partitionen nicht beschränkt.

Im Folgenden wird eine beispielhafte Implementierung der zwei wichtigsten Datenstrukturen für das verteilte Objektmodell vorgestellt, die effizient in der numerischen CFD-Anwendung *padfem*^{2} (vgl. hierzu Abschnitt 4.1, [BKM03]) umgesetzt wurde.

Partitionslokaler Namensraum

Für eine reale Implementierung der lokalen Namensraumabbildung $\text{label}_{\mathbb{K}_i}$, mit $\mathbb{K}_i \in \{N_{\Omega_i}, E_{\Omega_i}, F_{\Omega_i}, V_{\Omega_i}\}$, wird eine Datenstruktur benötigt, die speicherplatz-sparend die Objekte des Tetraedernetzes verwaltet. Zudem wird eine möglichst effiziente Umsetzung der Operationen verlangt, da Zugriffe auf die Netzobjekte aufgrund der Häufigkeit die Laufzeit der Anwendung dominieren. Zu den Operationen zählen maßgeblich das Referenzieren sowie das Erzeugen und Löschen von Objekten.

Referenzierung tritt immer dann auf, wenn mit dem Netzelement "gearbeitet" wird. Erzeugung und Löschen eines Objektes bedeutet hier, einen eindeutigen Namen, zur Identifizierung des Objektes zu finden bzw. ihn wieder freizugeben. Das Objekt selbst anzulegen oder freizugeben ist Aufgabe der Speicherverwaltung der Anwendung und soll hier nicht weiter behandelt werden. Die Datenstruktur für die lokale Namensraumabbildung verwaltet somit die Abbildung von Objektname zu Zeiger auf das Netzelement.

Das Grundprinzip der hier entwickelten Datenstruktur ist an die Speicherverwaltung von Betriebssystemen angelehnt (vgl. [SG97, Tan01]). Ein Betriebssystem, welches das Konzept des virtuellen Speichers nutzt, muss eine effiziente Methode zur Umsetzung von Adressen aus dem logischen Adressraum in den physikalischen Adressraum und umgekehrt bieten. Dabei werden für gewöhnlich hardware-unterstützende Funktionen der CPU/MMU[7] genutzt. Adressen aus dem logischen Adressraum werden dabei durch ein mehrstufiges Umsetzungsverfahren in die physikalischen Adressen überführt. Der Aufbau der logischen Adresse entspricht dabei einem aus mehren Teilindizierungen zusammengesetzten Konstrukt. Dieses Prinzip der Teilindizierungen wird auch für die Verwaltung und Konstruktion der Objektnamen verwendet.

Der Name eines Objektes $k \in \mathbb{K}_i$ mit $\mathbb{K}_i \in \{N_{\Omega_i}, E_{\Omega_i}, F_{\Omega_i}, V_{\Omega_i}\}$ ist ein Tupel $(n, p) \in \mathbb{I} \subset \mathbb{N} \times \mathbb{N}$ (vgl. Definition 3.8). Der Partitionsindex $p \in \mathbb{N}$ ist auf einem Rechenknoten implizit[8] gegeben und wird daher in der Datenstruktur für den lokalen Namensraum nicht mitgespeichert. Es bleibt das eigentliche Identifizierungsmerkmal $n \in \mathbb{N}$, der Objektname, übrig, welches in die endliche Zahlendarstellung auf dem Rechensystem überführt werden muss. In der *padfem*^{2}-Simulationsumgebung wurde für den möglichen Darstellungsbereich ein Datentyp mit 32 Bit Breite gewählt, unabhängig davon, ob dem Rechensystem eine 32-Bit oder 64-Bit Architektur zu Grunde liegt. Partitionsnummern sind aus Implementationsgründen auf 16 Bit beschränkt. Eine Erweiterung der Datentypen auf größere Breiten ist ohne weiteres möglich, wenn die entsprechenden Rahmenparameter (s.u.) angepasst werden. Damit ergibt sich eine Auswahl des Identifizierungsmerkmals n aus $2^{32} - 1$ Möglichkeiten, der Wert 0 ist ein besonderer Wert zur Fehlerindikation und muss daher ausgenommen werden. Dieser Darstellungsbereich ist ausreichend für die Namensvergabe *eines Elementtyps*, da jeder Elementtyp einen eigenen Namensraum, d.h. eine eigene Tabelle hat.

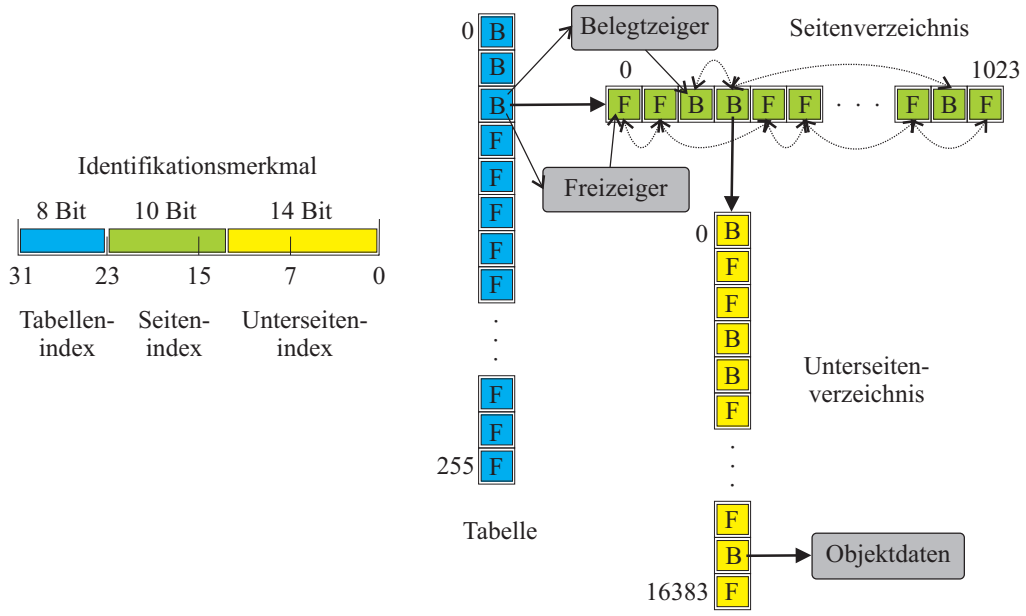
Das 32-bittige Identifizierungsmerkmal n wird unterteilt in drei Teilindizierungen von 8, 10 und 14 Bit Breite, die als Indizes in die sog. Elementtabelle (8 Bit = 256 Einträge), das Seitenverzeichnis (10 Bit = 1024 Einträge) und das Unterseitenverzeichnis (14 Bit = 16384 Einträge) verwendet werden. Ein Tabelleneintrag verweist somit auf 2^{10+14} Netzobjekte. In einem Eintrag im Unterseitenverzeichnis steht dann der Zeiger auf den Speicherplatzverweis des Netzelementes. Tabellen werden als normale Array-Struktur im Rechensystem abgebildet, was eine optimale Zugriffsgeschwindigkeit garantiert. Abbildung 3.1 zeigt schematisch die Datenstruktur. Die Stückelung der Eintragsverweise auf Array-Blöcke mit kleiner konstanter Größe garantiert einen reduzierten Speicherverbrauch mit geringem Verschnitt sowie eine Verminderung häufiger Betriebssystem- bzw. Laufzeitsystemaufrufe der Speicherallokation/ -deallokation[9]. Die Hauptoperationen auf die Datenstruktur, die im weiteren Verlauf der Arbeit als *erweiterte Array-Struktur (advanced array)* bezeichnet werden soll, gestalten sich nun wie folgt.

[7] MMU = memory management unit

[8] bspw. durch den MPI-Prozessrang

[9] Speichersystem-Operationen zählen zu den zeitintensivsten Operationen für die Laufzeit eines Programmes und sollten daher durch geschickte Programmierung vermieden werden.

Abbildung 3.1 Erweiterte Array-Struktur zur Namensraumverwaltung



Elementreferenzierung Die Referenzierung zu einem Identifizierungsmerkmal n besteht in der Zerlegung des 32-Bit Wertes in die drei Teilindizes und der Verfolgung durch die Eintragverzeigerung in der Datenstruktur. Der Aufwand dafür ist konstant und liegt damit in $\mathcal{O}(1)$. Als Ergebnis der Referenzierung wird entweder der Zeiger auf das Netzelement geliefert, oder aber, wenn der Name nicht gültig bzw. nicht belegt ist, ein Nullzeiger zurückgegeben, was gleichbedeutend mit dem 0-Fehlerwert ist. Die Referenzierungsoperation kann somit auch für die Prüfung der Gültigkeit eines Identifizierungsmerkmals genutzt werden.

Elementerzeugung Das Auffinden eines unbenutzten Namens für ein neu erzeugtes Netzobjekt basiert auf einem "Freizeiger"-Mechanismus. Dazu sind die Einträge im Unterseitenverzeichnis in zwei doppelt-verketteten Listen verknüpft, der *Freiliste*, die die nicht belegten Einträge beinhaltet, und der *Belegtliste* für die bereits vergebenen Einträge. Dies bedeutet, dass in einem Unterseiteneintrag Platz für drei Zeiger sein muss, dem Zeiger auf das Netzelement und den beiden Verwaltungszeigern für eine doppelt verkettete Liste. Die Liste wird wahlweise für die Belegt- oder Freiliste verwendet. Der Freizeiger eines Unterseitenverzeichnisses zeigt nun immer auf das erste Element der Freiliste. In Abbildung 3.1 ist dies am Beispiel eines Seitenverzeichnisses dargestellt. Das Dereferenzieren des Freizeigers liefert nun den freien Namen für das neue Objekt, indem aus den einzelnen Eintragindizes der Tabelle, des Seiten- und Unterseitenverzeichnisses das Identifizierungsmerkmal n konstruiert wird. Ist die Unterseite voll, d.h. der Freizeiger zeigt auf keinen freien Platz, dann wird eine Ebene höher in dem Seitenverzeichnis nach dem nächsten Unterseiteneintrag mit freien Plätzen gesucht und ggf. ein neuer Array-

Block für ein Unterseitenverzeichnis vom Laufzeitsystem angefordert. Auch hier findet eine Referenzierung mittels einer Frei- und Belegtliste zur beschleunigten Suche statt. Im Falle eines Tabelleneintrages, der auch nur belegte Plätze für das Seitenverzeichnis aufweist, kann in der Tabelle nach dem nächsten freien Seitenverzeichnis ebenfalls mit Hilfe von Frei- und Belegtlisten für die Tabelleneinträge gesucht werden. Ist ein freier Platz für den Namen gefunden, müssen die Verwaltungsstrukturen der Frei- und Belegtlisten auf allen durchsuchten Hierarchieebenen aktualisiert werden. Da bei dieser Verwaltungsmethode nur Zeigermanipulationen und/oder -referenzierungen bis zu einer Tiefe von drei Hierarchien erfolgen, wird für die Suche nach einem neuen Namen nur konstanter Aufwand benötigt, d.h. der Aufwand für die Erzeugung eines neuen Objektes liegt mit dieser Datenstruktur in $\mathcal{O}(1)$.

Elementfreigabe Das Freigeben eines Namens für ein Netzobjekt gestaltet sich in ähnlicher Weise wie die Erzeugung eines Namens. Ist durch Referenzierung der Unterseiteneintrag gefunden worden und als gültig identifiziert, d.h. der Objektzeiger im Eintrag ist nicht der Nullzeiger (doppelte Löschung), wird der Objektzeiger auf den Nullzeiger gesetzt und die Verwaltungsstruktur der Frei- und Belegtliste im Unterseitenverzeichnis aktualisiert. Ist die Belegtliste leer, also alle Einträge des Unterseitenverzeichnisses sind leer, kann das Verzeichnis freigegeben werden, welches eine Aktualisierung der Verwaltungsstruktur in der übergeordneten Hierarchie des Seitenverzeichnisses nach sich zieht und so fort. Der Objektzeiger des zu entfernenden Elementes kann nun der Speicherverwaltung des Laufzeitsystems bzw. der Anwendung übergeben werden[10]. Auch hier bewegt sich der Aufwand für die Aktualisierung der Verwaltungsstrukturen, wie beim Erzeugen eines neuen Namens, in $\mathcal{O}(1)$.

Die wichtigsten Operationen der Datenstruktur für die Verwaltung der lokalen Namensräume laufen damit alle mit konstantem Zeitaufwand ab. Der Speicheraufwand hält sich mittels der Allokation/Deallokation von ganzen Array-Blöcken bestimmter Größe in vertretbarem Rahmen, da ein großer Verschnitt vermieden wird. Der schlechteste Fall eines Verschnitts ist eine Lücke zwischen zwei Objektnamen, deren Identifikationsmerkmale eine Differenz von genau $2 \cdot (2^{14} + 2^{10}) = 34816$ Einträgen aufweisen, die sich also in zwei verschiedenen Tabelleneinträgen mit maximaler Weite befinden. Dieser Fall tritt aber so gut wie gar nicht ein, da die Objektnamen eines initialen Netzes für gewöhnlich eine zusammenhängende Folge ergeben und Lücken im Namensraum während der Laufzeit der Anwendung nur durch Entfernen von Objekten im Netz, d.h durch Modifikation der Netzgeometrie, entstehen (geometrische Adaption). Diese werden aber sofort durch die neu erzeugten Objekte mittels des Freizeiger-Konzeptes wieder aufgefüllt.

Iteratoren Algorithmen, die auf den Objektdaten arbeiten, benötigen generell eine Aufzählungsmöglichkeit, die immer die gleiche Folge von Elementen liefert, falls keine Netz-

[10] Bei Verwendung von sehr vielen Objekten mit dynamischer Allokation bzw. Deallokation sollte aus Effizienzgründen das Konzept des Speicherpools angewendet werden, um den Zeitaufwand bei der Speicherverwaltung zu reduzieren.

modifikation erfolgt, also Elemente hinzugefügt oder entfernt werden. Dazu kann man in der Datenstruktur für die Namensverwaltung einen Iterator nutzen, der anhand der verketteten Belegtliste eine feste Abfolge aufzählt. Benötigt wird dazu nur ein Zeiger auf das erste Netzobjekt, bei dem in der doppelt verketteten Belegtliste kein Vorgänger existiert. Das Iterieren erfolgt dann durch kontinuierliches Verfolgen des Zeigers zum nächsten Element. Ist ein Unterseitenverzeichnis abgelaufen, erfolgt gemäß der Verkettung in der nächst höheren Hierarchieebene im Seitenverzeichnis das nächste Unterseitenverzeichnis. Dieser Vorgang wiederholt sich bis zum nächsten Tabelleneintrag und dort bis zum letzten gültigen Eintrag. Die Reihenfolge, in der die Netzelemente durch das Iterieren aufgezählt werden, legen eine neue eigenständige, vom lokalen Namensraum der Partition unabhängige und zusammenhängende Zählung fest, die ebenfalls für nur partitionslokal operierende Algorithmen verwendet werden kann.

Um die Zugriffsgeschwindigkeit auf die Netzelemente durch den Iterator weiter zu erhöhen, falls in Algorithmen häufig auf eine *konstante* Netzgeometrie zugegriffen wird, kann der Iterator in ein einzelnes kompaktes Array konstanter Größe überführt werden. Dies entfernt den Verwaltungsaufwand für die Zeigerverfolgung in der Namensraumtabelle und reduziert den Iterator zu einer simplen Index-Inkrementierung beim Array-Zugriff. Sobald allerdings Netzmodifikationen auftreten, wird das kompakte Array ungültig und muss neu aufgebaut werden. Da dies aber nur zu bestimmten Zeitpunkten während eines Simulationslaufes passiert, nämlich in einer Adaptionphase und der nachfolgenden Lastverteilung, ist dieser Aufwand im Verhältnis zur Gesamtlaufzeit, die vom numerischen Laufzeitanteil dominiert wird, verschwindend gering. Ein weiterer Vorteil der Überführung in ein kompaktes Array ist eine effizientere Cache-Ausnutzung beim Zugriff auf die Netzdaten, da sie im Speicher aufeinander folgende Bereiche belegen[11].

Parallelisierung Eine Mehrprozessorunterstützung der Datenstruktur ist mit wenig Aufwand möglich. Die Operationen Erzeugen und Freigeben können durch Semaphoren bzw. Mutex-Verfahren bei Zeigermanipulationen geschützt werden, die Referenzierung eines Netzelementes benötigt keine besondere Schutzmaßname, sofern sie nicht mit Modifikationen der Datenstruktur konkurrieren muss. Aufwändiger ist allerdings die Parallelisierung der Iteratoren. In der *padfem*^{2}-Umgebung wurde dazu der *Slice*-Datentyp eingeführt, der einen definierten Teilbereich der Aufzählungsfolge der Netzelemente darstellt. Jedem Slice wird ein Prozessor innerhalb eines Rechenknotens zugewiesen, der den Teilbereich per Iterator aufzählt. Ein Slice wird durch einen Anfangs- und einen Endzeiger repräsentiert, die in die Unterseitenverzeichnisse der erweiterten Array-Struktur verweisen, wobei die beiden Zeiger eines Slices nicht auf Einträge der gleichen Unterseite zeigen müssen. Der *parallele Iterator* zählt dann nur die Netzelemente zwischen Anfangs- und Endzeiger auf. Da sich die Anzahl der Elemente innerhalb einer Namensraumtabelle durch geometrische Netzmodifikationen an beliebiger Stelle ändern kann, d.h. Lücken beliebiger Größe im Namensbereich entstehen können, ist eine gleichmäßige Aufteilung der Elemente innerhalb der Slices zwingend notwendig, um Warte-

[11] Voraussetzung hierfür ist allerdings eine effektive, anwendungsinterne Speicherverwaltung, bspw. durch Memorypools mit Arrayunterstützung.

zeiten bei parallelen Algorithmen durch Prozessorleerlauf zu minimieren. Dieser Lastausgleich geschieht durch Überprüfung bei jeder Erzeugungs- und Freigabeoperation in der Tabelle. Dazu wird der Slice, in der das einzufügende bzw. löschende Element liegt, ermittelt. Hat sich nun die Anzahl der Elemente nach der auszuführenden Operation gegenüber seinen Nachbarslices verändert, erfolgt eine Verschiebung der Anfangs- und Endzeiger der Slices, um das Gleichgewicht wieder herzustellen. Dazu werden die Verwaltungszeiger der erweiterten Array-Struktur herangezogen. Der Aufwand für diese Art der Lastverteilung liegt hier ebenfalls wie die Hauptoperationen der Namensraumtabelle in der Klasse $\mathcal{O}(1)$.

$\omega_{\mathbb{K}_i}$-Abbildung

Für die Umsetzung der $\omega_{\mathbb{K}_i}$-Abbildung aus Definition 3.11 wird ebenfalls eine effiziente Datenstruktur benötigt, da im parallelen Fall bei Kommunikationsoperationen sehr häufig auf Randelemente und die Halos sowohl der eigenen Partition als auch der Nachbarpartitionen zugegriffen wird. Dies ist z.B. bei Aktualisierungen des Halos einer Partition während der Laufzeit eines Gleichungslösers notwendig oder bei einer geometrischen Adaption, bei der sich eine Adaptionsfront von einer Partition in eine Nachbarpartition ausbreitet.

Da Suchoperationen die Anwendung der Datenstruktur für die $\omega_{\mathbb{K}_i}$-Abbildung dominieren, bietet sich hierfür ein Hashtabellen-Mechanismus an, bei dem der Aufwand für das Suchen von Tabelleneinträgen asymptotisch[12] in der Klasse $\mathcal{O}(1+\alpha)$ mit Füllfaktor α liegt (vgl. hierzu auch [OW90, CLR90, Sed02]). Hashtabellen stellen eine gute Möglichkeit einer Datenstruktur dar, bei der die Hauptanwendung eine verzeichnis-basierte Verwaltung von Daten benötigt. Dies trifft insbesondere auf die $\omega_{\mathbb{K}_i}$-Abbildung zu. In der *padfem*^{2}-Umgebung wird als reale Implementierung für die Datenstruktur der $\omega_{\mathbb{K}_i}$-Abbildung daher eine Kombination aus Array und dynamisch verwalteten Hashtabellen verwendet. Jede Partition besitzt pro Elementtyp eine eigene Datenstruktur diesen Typs. Das Array besitzt für jede Partition im gesamten Netz einen Eintrag, wobei allerdings nur die Einträge gültig sind, für die eine Partition reale Nachbarn hat. D.h. nur $|\text{neighbor}(P_i)| = q$ Einträge einer n-Partitionierung haben einen gültigen Wert, der restliche Anteil $n - q$ hat Null-Einträge. Die gültigen Einträge selbst sind Zeiger auf Hashtabellen, in denen Schlüssel/Wert-Paare gespeichert sind, die die eigentliche Umsetzung der Identifikationsmerkmale einer lokalen Partition in eine andere, benachbarte Partition ermöglichen. Manche parallel arbeitenden Algorithmen benötigen für eine konsistente Datenhaltung bzw. Prüfung eine Möglichkeit zur reversen Suche. Dies bedeutet, dass eine Partition P_i von einer Nachbarpartition P_j ein Datum gesendet bekommt, welches im Namensraum von P_j ein Identifikationsmerkmal besitzt. Um das entsprechende lokale Netzelement zu ermitteln, ist eine reverse Suche nötig, was gleichbedeutend mit der Umkehrfunktion $\omega_{\mathbb{K}_i}^{-1}$ ist. Daher besitzt jede Partition zusätzlich

[12] je nach verwendeter Hashfunktion und Kollisionsauflösungsmethode

noch eine Datenstruktur für die Umkehrabbildung, hat also den doppelten Speicherumfang für die Verwaltung der $\omega_{\mathbb{K}_i}$-Abbildung.

Bei Hashtabellen stellt sich immer die Frage, wie groß die Tabelle sein muss, um die Schlüssel/Wert-Paare zu speichern, und welche Hashfunktion eine möglichst gleichmäßige Verteilung der Schlüssel/Wert-Paare auf die zur Verfügung stehenden Tabelleneinträge bewirkt. Die Wahl der Hashfunktion impliziert auch das Verfahren zur Kollisionsauflösung bei Abbildung zweier verschiedener Schlüssel auf denselben Eintrag der Tabelle. Da die Identifikationsmerkmale von Netzobjekten aus der Menge der natürlichen Zahlen $\mathbb{N}$ stammen, bietet es sich an, als Hashfunktion eine Divisionsmethode (modulo-Methode) zu nutzen. Die Datenstruktur zur Verwaltung der partitionslokalen Namensräume unterstützt zudem durch das Freizeigerkonzept das Lokalitätsprinzip zur Namensvergabe. Dies bedeutet, geometrisch benachbarte Netzelemente haben Identifikationsmerkmale, die aufeinanderfolgend oder zumindest dicht beinander liegen. Daraus folgt, daß räumlich dicht angeordnete Netzobjekte in der Hashtabelle durch eine Divisonsmethode auf benachbarte Tabelleneinträge abgebildet werden und keine Kollisionen erzeugen. Dies führt auf eine Hashfunktion $\mathrm{hash}(n)$ mit n als Identifikationsmerkmal, die folgenden sehr einfachen aber effektiven Aufbau besitzt:

$$\mathrm{hash} : \mathbb{N} \to \{0, 1, 2, \ldots, m-1\}; \quad n \mapsto n \mod m, \tag{3.11}$$

wobei die Tabellengröße m definiert ist durch $m = \mathrm{nextprime}(2^q)$. Durch die Funktion $\mathrm{nextprime}(x)$ wird zu einem gegebenen Wert $x \in \mathbb{N}$ die Primzahl ermittelt, die größer oder gleich x ist. $q \in \mathbb{N}$ ist ein zur speichernden Datenmenge passender Wert, der sich während der Laufzeit der Simulation bzw. der Lebensdauer der Hashtabelle dynamisch anpasst. Beispielsweise kann q in Abhängigkeit des Füllgrades der Tabelle verändert werden[13]. Andere Arten von Hashfunktionen, wie z.B. eine multiplikative Methode oder doppeltes Hashing, funktionieren mit Identifikationsmerkmalen als Suchschlüssel ebenfalls gut, sind aber in der Berechnung wegen zusätzlicher Operationen aufwändiger. Da das Lokalitätsprinzip der Namensraumdatenstruktur auch eine gute Verteilung der Identifikationsmerkmale über die Hashtabelle erzeugt, sind für etwaige Probleme beim Abbilden auf dieselben Eintragsplätze Kollisionslisten gut geeignet. Die Länge der Listen ist somit auch klein, welches den Suchaufwand in der Tabelle weiter reduziert. Degenerierte Tabellen (mit langen Kollisionslisten) treten wegen der dynamischen Größenverwaltung über den Füllgrad nicht auf. Die Aufwandsklasse einer dynamischen Größenanpassung der Tabelle ist $\mathcal{O}(n)$, wenn n die Anzahl der eingetragenen Elemente in der Tabelle ist. Die Häufigkeit, mit der eine Größenanpassung erfolgt, sinkt mit steigender Anzahl der zu verwaltenden Elemente, welches durch die $\mathrm{nextprime}(2^q)$-Funktion bedingt ist (exponentielles Wachstum der Tabellengröße).

Die kombinierte Datenstruktur für die $\omega_{\mathbb{K}_i}$-Abbildung und ihre Umkehrabbildung $\omega_{\mathbb{K}_i}^{-1}$ wird für jede Partition lokal einmal beim Start einer Simulation mit dem initial partitionierten Netz aufgebaut und bleibt bis zu einer netzmodifizierenden Phase im Si-

[13] In der *padfem*^{2}-Umgebung wird q z.B. bei einem Füllgrad der Tabelle von 75% um eins erhöht und bei 15% um eins erniedrigt. Diese Werte haben sich in der Praxis als am effektivesten erwiesen.

mulationsverlauf konstant. Ändert sich der Rand bzw. der Halo einer Partition, so werden die Veränderungen des Netzes in die Datenstruktur kontinuierlich eingepflegt. Neu hinzukommende Nachbarpartitionen bzw. Nachbarn, die vom Rand der Partition wegmigrieren, erfordern das Anlegen in oder Entfernen aus der Arraystruktur. Ein vollständiger Auf- bzw. Abbau der Datenstruktur ist nicht erforderlich, stattdessen dient sie in einer Modifizierungsphase als Hilfestellung für partitionsübergreifende Änderungen in der Netzgeometrie und zur Konsistenzprüfung.

Eine parallele Nutzung der Datenstruktur zur $\omega_{\mathbb{K}_i}$-Abbildung auf einem Mehrprozessorknoten gestaltet sich einfach, da durch simple Semaphoren bzw. Mutex-Variablen die kritischen Verwaltungsoperationen der Hastabelle geschützt werden können. Die Suchoperation ist die am meisten genutzte Operation der Hashtabelle in diesem Objektmodell. Sie besitzt grundsätzlich keine kritischen Stellen, so dass hier der Aufwand für die parallele Verwaltung wegfällt. Dies gilt natürlich nur für eine konstante Datenmenge in der Hashtabelle, was aber in den nachfolgenden Algorithmen bei Benutzung von $\omega_{\mathbb{K}_i}$ bzw. $\omega_{\mathbb{K}_i}^{-1}$ immer gewährleistet ist.

3.2 Numerische Algorithmen

Simulationsumgebungen für strömungsmechanische Problemstellungen[14] (s. hierzu z.B. [Sch90, MGN95, Bra03, BJ93, JL01]) verbringen den größten Teil ihrer Laufzeit mit der numerischen Behandlung von einzelnen Berechnungsschritten. Diese bestehen zum einen aus der Überführung einer Simulationsgeometrie mit problemspezifischen Merkmalen (Randbedingungen, zeitliche Änderungen, Abhängigkeiten, usw.) in ein lineares bzw. nicht-lineares Gleichungssystem. Als nächster Schritt folgt das Lösen dieses Systems. Im abschließenden dritten Schritt erfolgt die Rücktransformation der Lösung in die Geometrie zur visuellen Betrachtung und Interpretation des Ergebnisses. In diesem Abschnitt des Kapitels erfolgt eine Darstellung, wie mittels dem in dieser Arbeit entwickelten verteilten Objektmodell numerische Algorithmen umgesetzt werden können. Dabei soll weniger die strömungsmechanische Problemstellung als viel mehr die Umsetzung numerischer Standardgleichungslöser und Randabgleichalgorithmen untersucht werden.

3.2.1 Gleichungslöser

Numerische Gleichungslöser sind Hauptbestandteil für die Lösung von mathematischen Problemstellungen, die mittels Simulationen berechnet werden. Es existiert eine Vielzahl von Verfahren, um ein Gleichungssystem $Ax = b$ zu lösen. Dabei bedienen sie sich bestimmter Eigenschaften der Koeffizientenmatrix A und dem Umfeld, aus dem diese

[14] In dieser Arbeit liegt der Schwerpunkt für die numerische Behandlung von FEM-Simulationen im strömungsmechanischen Umfeld. Alle Aussagen bzgl. numerischer Anwendungsfälle treffen auch auf andere Problemstellungen zu, die mit der FE-Methode gelöst werden können (wie z.B. strukturmechanische).

Matrix generiert wird. Eine ausführliche Abhandlung über die verschiedenen Verfahren und ihre Anwendungen kann z.B. in [GL96, Saa03, Hac93, Mei05, SK04] gefunden werden.

Im Umfeld strömungsmechanischer Problemstellungen, die in dieser Arbeit die Grundlage für die Untersuchung des entwickelten, verteilten Objektmodells bilden, treten bevorzugt Gleichungssysteme auf, deren Koeffizientenmatrizen schwach besetzt, symmetrisch und positiv definit bzw. semidefinit sind. Für solche Strukturen haben sich Krylov-Unterraum-Verfahren bewährt, die einen iterativen Ansatz zur Lösung eines Gleichungssystem $Ax = b$ verfolgen. Aus dieser Klasse sollen nun beispielhaft einige der Verfahren mittels des Objektmodells für partitionslokale Namensräume umgesetzt werden.

Matrixtransformation einer verteilten Geometrie

Es existieren zwei Möglichkeiten, einen Algorithmus eines Gleichungslösers anzuwenden. Die erste ist, auf der im Speicher des Rechners vorhandenen Geometriedatenstruktur direkt zu arbeiten. Die zweite Möglichkeit sieht eine Transformation der Geometrie im Speicher und der mit dieser verbundenen numerischen Daten in eine spezielle Datenstruktur vor, auf der dann der Gleichungslöser angewendet wird. Das direkte Arbeiten auf der Geometriedatenstruktur hat den Vorteil, dass kein zusätzlicher Speicher benötigt wird. Zudem lassen sich die Algorithmen sehr leicht bzw. mit geringem Aufwand für einen Löser im Programmcode formulieren. Der Nachteil dieser Methode allerdings wiegt wesentlich schwerer. Dadurch, dass die Netzgeometrie eine komplex verkettete Datenstruktur darstellt, ist jeder Zugriff auf Datenelemente, welche die Einträge der Matrix und der rechten Seite im Gleichungssystem repräsentieren, mit einem oder mehreren langsamen Speicherreferenzierungen belastet. Die Zugriffsreihenfolge wirkt sich zusätzlich negativ auf die Prozessorcachenutzung aus, da die Zeileneinträge auf den Nebendiagonalen einer Matrix nicht notwendigerweise Netzelemente in der Geometrie repräsentieren, die im Speicher hintereinander liegen und so den Effekt der Fehlzugriffe auf den Cache begünstigen. Formt man aber die für den Gleichungslöser benötigten Daten, also Haupt- und Nebendiagonalelemente der Koeffizientenmatrix A sowie die rechte Seite b, in eine spezielle Datenstruktur um, so können diese z.B. für eine bestimmte Prozessorarchitektur so günstig im Speicher angeordnet werden, dass durch eine optimale Zugriffsreihenfolge auf die Daten maximale Geschwindigkeit bei der Verarbeitung erreicht wird. Die Lokalität der Daten wird dadurch erhöht. Der Geschwindigkeitsvorteil der zweiten Methode gegenüber der ersten Methode kann sich gravierend auswirken. So haben sich bspw. in der *padfem*^{2}-Umgebung (vgl. Abschnitt 4.1) Faktoren um 60-80 ergeben. Die zweite Methode benötigt aber zusätzlichen Speicher zu der eigentlichen Geometriedatenstruktur, einen gewissen Zeitaufwand für die Konstruktion der Datenstruktur und erschwert die Modellierung der Algorithmen. Dennoch wird in der Praxis meistens die Methode mittels Zusatzdatenstruktur verwendet, da der Vorteil einer höheren Rechengeschwindigkeit den größeren Speicherverbrauch überwiegt.

Es existieren eine Reihe von spezialisierten Datenstrukturen, die auf kompakte Weise und unter Ausnutzung der Matrixstruktur und -eigenschaften, eine schwach besetzte Matrix im Speicher eines Rechners repräsentieren können. Dabei basieren diese Datenstrukturen grundsätzlich auf einem Arraystruktur-Prinzip, um die numerischen Daten möglichst kompakt und cache-effizient zu halten. Zu den bekanntesten Datenstrukturen zählen *Coordinate Storage (CS), Compressed Row Storage (CRS), Compressed Column Storage (CCS), Blocked Compressed Row Storage (BCRS), Compressed Diagonal Storage (CDS), Jagged Diagonal Scheme (JDS)* sowie *Skyline Matrix Storage (SMS)* (vgl. hierzu auch z.B. [BBC+94, SW05]). In dieser Arbeit wird das CRS-Format bevorzugt, da es am meisten Verwendung findet.

Die CRS-Format Darstellung einer $n \times m$ Matrix $A = (a_{ij})$, $1 \leq i \leq n$, $1 \leq j \leq m$ besteht aus drei einzelnen Arrays bzw. Vektoren, wobei eines die Werte $a_{ij} \neq 0$ speichert, und die zwei anderen Verwaltungsinformationen über die Position der Werte aus dem ersten Array.

DEFINITION 3.13 (COMPRESSED ROW STORAGE FORMAT EINER MATRIX A)
Sei $A \in \mathbb{R}^{n \times m}$ *mit* $A = (a_{ij})$, $1 \leq i \leq n$, $1 \leq j \leq m$. *Die Darstellung* $\mathrm{CRS}(A) = (v, c, r, nnz)$ *repräsentiert die kompakte Schreibweise der Matrix* A *im* Compressed Row Storage Format. *Dabei bezeichnet* nnz, *mit* $0 < nnz \leq n \cdot m$, *die Anzahl der von Null verschiedenen Matrixeinträge aus* A *und* $v = (a_k)_{1 \leq k \leq nnz}$ *mit* $a_k = a_{ij} \wedge a_{ij} \neq 0$ *den Vektor der von Null verschiedenen Matrixeinträge in Zeilenaufzählweise.* $c = (c_k)_{1 \leq k \leq nnz}$ *und* $r = (r_k)_{1 \leq k \leq n}$ *sind zwei Indexvektoren, die die Position der Vektoreinträge aus* v *in* A *bzgl. Spalte und Zeile beschreiben, d.h. gilt* $v[k] = a_k = a_{ij}$ *für* $1 \leq k \leq nnz$, *dann ist* $c[k] = c_k = j$ *und* $r[i] = r_i \leq k < r_{i+1} = r[i+1]$.

Der Speicheraufwand[15] der Matrix A reduziert sich somit von $n \cdot m$ bzw. bei quadratischen Matrizen n^2 auf $2 \cdot nnz + n$ in der CRS-Darstellung. Ist die Matrix A symmetrisch, so kann noch mehr Speicher eingespart werden, wenn bspw. nur der obere Dreiecksanteil der Matrix gespeichert wird. Diese Methode benötigt allerdings ein kompliziertes Zugriffsmuster (durch indirekte Addressierung), welches die Laufzeit durch Cacheineffizienz erhöht. Dieser Ansatz wird hier nicht weiter verfolgt. Auf die einzelnen Komponenten der Formatdarstellung wird in den folgenden Algorithmen mittels $\mathrm{CRS}(A).v$, $\mathrm{CRS}(A).c$, $\mathrm{CRS}(A).r$ und $\mathrm{CRS}(A).nnz$ zugegriffen.

Es gilt nun, aus einer verteilten Geometrie $\mathcal{M}_\Omega(n) = (\mathcal{P}_\Omega, \mathcal{H}_\Omega, \omega_\Omega)$, eine Darstellung $\mathrm{CRS}(A)$ für das Matrixsystem A aufzustellen, die mittels den aus $\mathcal{M}_\Omega(n)$ gegebenen Partitions- bzw. Halostrukturen eine verteilte, aber konsistente Struktur besitzt. Auf dieser kann dann ein Gleichungslöser arbeiten. Die Einträge der Matrix A, die sich aus dem Kontext der zu lösenden Simulationsproblemstellung ergeben, seien hier schon als berechnet (assembliert) angenommen. D.h. für ein $k \in \mathbb{K}$ mit $\mathbb{K} \in \{N_\Omega, E_\Omega, F_\Omega, V_\Omega\}$ existiert ein anhängiges Datum $k.a_h \in \mathbb{R}$ bzw. $k.a_n \in \mathbb{R}$, welches den Haupt- bzw.

[15] In manchen Implementierungen der CRS-Datenstruktur wird für den Vektor r, der die Zeilenstartpositionen speichert, eine Länge von $n+1$ angenommen und der Wert $r[n+1] = r_{n+1} = nnz$ gesetzt.

Nebendiagonaleintrag der Matrix A symbolisiert. Das gleiche gilt für die rechte Seite b des Gleichungssystems, im Folgenden durch $k.a_b \in \mathbb{R}$ gekennzeichnet. Desweiteren beschreibt ein Eigenschaftswert $k.p \in \{$"Dirichlet","Neumann", "Robin"$\}$ für ein $k \in \mathbb{K}$, von welchem Typ die Randbedingung für dieses Netzobjekt ist, falls $k \in \overline{\mathbb{K}}$ ein echtes Gebietsrandelement ist (zu Randbedingungen s. z.B. [JL01, BJ93, Bra03]). Eine mögliche Konstruktion von $CRS(A)$ aus $\mathcal{M}_\Omega(n)$ wird nun durch Algorithmus 3.4 verdeutlicht. Der Einfachheit halber seien für den reinen FEM-Ansatz nur Gebietsknoten mit ihren Kanten als Verbindungsobjekte betrachtet. Das Äquivalent über Tetraeder und Dreiecksflächen für z.B. Finite-Volumen-Methoden wird hier nicht weiter betrachtet, kann aber durch Ersetzung der entsprechenden Objektmengen in den jeweiligen Algorithmen leicht hergeleitet werden.

Algorithmus 3.4 benötigt als Eingabe die lokale Partitionsnummer p sowie den partitionslokalen Anteil der Netzgeometrie $\mathcal{M}_{\Omega_p}(n) = (P_p, \mathcal{H}_p, \omega_{\mathbb{K}_p})$. Als Ausgabe erzeugt der Algorithmus die kompakte, *verteilte* Repräsentation der Matrix A im CRS-Format, die rechte Seite b des Gleichungssystems und den Startlösungsvektor x. Alle Ausgaben beziehen sich auf den lokalen Datenanteil der Partition P_p. Um die Verteilung der Daten darstellen zu können, muss das CRS-Standardformat aus Definition 3.13 für partitionslokale Namensräume um zwei zusätzliche Komponenten erweitert werden. Die erste Komponente $CRS(A).lut$ ist eine Abbildung, die den lokalen Namensraum der Netzknoten in eine eigene Indexdarstellung für die Matrixrepräsentation überführt. Bei der zweiten Komponente $CRS(A).H^j_{s,r}$ handelt es sich um zwei Vektoren zur Speicherung der lokalen und entfernten Haloknoten im Indexformat für die Matrixrepräsentation. Diese zwei Vektoren mit Indizes s und r für den lokalen und entfernten Anteil existieren für jede Nachbarpartition $P_j \in neighbor(P_p)$ von P_p. Beide Komponenten ermöglichen in den nachfolgenden Löseralgorithmen auf komfortable Weise eine Aktualisierung der Haloobjekte einer Partition in jeder Iteration.

Algorithmus 3.4 gliedert sich in vier Phasen. In der ersten Phase (Zeile 2-6) erfolgt eine Initialisierung der einzelnen Komponenten der CRS-Struktur für die Matrix A. Hierbei werden auch die beiden Mengen S_1 und S_2 der zu betrachtenden Netzobjekte bestimmt. S_1 beinhaltet sämtliche Gebietsknoten der lokalen Partition P_p. Gebietsrandknoten, deren Randbedingung vom Typ *Dirichlet* sind, werden nicht betrachtet. Zu den Knoten in S_1 zählen auch Partitionsrandknoten und Knoten in den entfernten bzw. kopierten Halogebieten der Partition. S_2 bezeichnet die Menge aller *nicht-Dirichlet* Nachbarknoten für Knoten aus S_1. Die Summe der Elementanzahlen aus S_1 und S_2 ergeben gerade die von Null verschiedenen Elemente nnz in der Matrix A, S_1 beschreibt somit die Hauptdiagonalelemente und S_2 alle Nebendiagonalelemente von A. In Zeile 6 erfolgte die Aufstellung der matrixlokalen Indexrepräsentation in $CRS(A).lut$. Die Initialisierungsphase liegt in der Aufwandsklasse $\mathcal{O}(|S_1|+|S_2|) = \mathcal{O}(n)$, wenn n die Anzahl der Knoten in N_{Ω_p} ist und im schlechtesten Fall keine Gebietsrandknoten vom Typ *Dirichlet* enthält.

Phase zwei (Zeile 7-17) baut gemäß Definition 3.13 die Komponenten ν, c und r der CRS-Struktur auf. Im Algorithmus werden die Daten der zu betrachtenden Knoten

Algorithmus 3.4 : Aufsetzen von $CRS(A)$ aus $\mathcal{M}_\Omega(n)$

Eingabe : $p \in \{1,\dots,n\}$ sei die lokale Partitionsnummer,
$\mathcal{M}_{\Omega_p}(n) = (P_p, \mathcal{H}_p, \omega_{\mathbb{K}_p})$ sei der partitionslokale Anteil der Geometrie
Ausgabe : Matrix $CRS(A)$, rechte Seite b, Startlösungsvektor x

```
1  parallel proc SetupCRSSystem begin
2      S_1 ← {n ∈ N_Ωp | n.p ≠ "Dirichlet"};
3      S_2 ← {n ∈ N_Ωp | n' ∈ S_1 : n ∈ R_N(n') ∧ n.p ≠ "Dirichlet"};
4      CRS(A).nnz ← |S_1| + |S_2|; CRS(A).v ← ∅; CRS(A).c ← ∅; CRS(A).r ← ∅;
5      CRS(A).lut ← ∅; CRS(A).H_{s,r}^{p∈neighbor(P_p)} ← ∅;
6      for i = 1,...,|S_1|, n_i ∈ S_1 do CRS(A).lut ← CRS(A).lut ∪ (n_i, i);
7      k ← 1;
8      for i = 1,...,|S_1|, n_i ∈ S_1 do
9          CRS(A).v_k ← n_i.a_h;
10         CRS(A).r_i ← k; CRS(A).c_k ← j : (n_i, j) ∈ CRS(A).lut;
11         k ← k + 1;
12         foreach n' ∈ S_{n_i} = {n' ∈ R_N(n_i) : n'.p ≠ "Dirichlet"} ⊂ S_2 do
13             CRS(A).v_k ← n'.a_n;
14             CRS(A).c_k ← j : (n', j) ∈ CRS(A).lut;
15             k ← k + 1;
16         end
17     end
18     foreach p' ∈ neighbor*(P_p) do
19         H_l ← {n ∈ H_{p→p'}.L_N | n.p ≠ "Dirichlet"};
20         H_r ← {n ∈ H_{p'→p}.L_N | n.p ≠ "Dirichlet"};
21         for i = 1,...,|H_l|, n_i ∈ H_l do
22             CRS(A).H_{s_i}^{p'} ← j : (n_i, j) ∈ CRS(A).lut;
23         end
24         for i = 1,...,|H_r|, n_i ∈ H_r do
25             Bestimme n' ∈ S_1 : ω_p(n', p) = ω_p^{-1}(n_i, p);
26             CRS(A).H_{r_i}^{p'} ← j : (n', j) ∈ CRS(A).lut;
27         end
28     end
29     for i = 1,...,|S_1|, n_i ∈ S_1 do
30         b_i ← n_i.b;
31         x_i ← 0;
32     end
33 end proc
```

im Vektor ν derart angeordnet, dass das Hauptdiagonalelement $n_i.a_h$ als erstes im Ausschnittsbereich der Zeilendarstellung von A steht und danach die *nicht-Dirichlet* Nebendiagonalelemente $n_i.a_n$ in "zufälliger" Reihenfolge. Aus Performancegründen[16] und als Anforderung bestimmter Lösungsverfahren[17] ist es u.U. wichtig, hier eine Ordnung in der Reihenfolge der Elemente für die Matrixzeile einzubringen. Zugriffe auf $CRS(A).lut$ können durch einen Hashtabellen-Mechanismus in konstanter Zeit realisiert werden. Die Nachbarknotenmenge S_{n_i} kann durch Vorausberechnung und Speicherung, beispielsweise beim Laden der Geometrie, ebenso als konstant angesehen werden. Daher ergibt sich für die Bestimmung der CRS-Struktur in dieser Phase ein Aufwand $\mathcal{O}(|S_1| + 2 \cdot |E_{\Omega_p}|)$ und kann somit in linearer Zeit abgearbeitet werden. Bei einer gewünschten Sortierung der Zeilenelemente bzw. der gesamten Matrix steigt der Aufwand entsprechend an.

Die dritte Phase (Zeile 18-28) berechnet die notwendige Indexvektorstruktur für den verteilten Ansatz im CRS-Format. Die Mengen H_l und H_r beschreiben die *nicht-Dirichlet* Knoten in der Netzgeometrie, die sich im lokalen bzw. entfernten Halo der Partition P_p zur aktuell betrachteten Nachbarpartition $P_{p'}$ befinden. Die erste Schleife (Zeile 21-23) bestimmt die Indexdarstellung für die lokalen Haloknoten und speichert diese für Partition $P_{p'}$ im Vektor $CRS(A).H_s^{p'}$. Die Indexberechnung für den entfernten Haloanteil (Zeile 24-27) benötigt eine zusätzlich Namensraumumsetzung für die Netzknoten, da die Identifikationsmerkmale im Halosystem $\mathcal{H}_p$ für die entfernten Anteile nur im Namensraum der Nachbarpartition bekannt sind. Die Umsetzung (Zeile 25) eines entfernten Haloknotens in einen lokalen Knoten erfolgt unter Benutzung der $\omega_{\mathbb{K}_p}$-Abbildung und deren Umkehrabbildung $\omega_{\mathbb{K}_p}^{-1}$. Wie in Abschnitt 3.1.2 beschrieben, kann hier ebenso durch einen Hashtabellen-Mechanismus die Umsetzung in konstanter Zeit erfolgen. Die Bestimmung der Matrixindexdarstellung für dieses nun ermittelte lokale Knotenobjekt in den Vektor $CRS(A).H_r^{p'}$ erfolgt wie beim lokalen Haloanteil. Der Aufwand für diese Phase ist abhängig von der Zahl der Nachbarpartitionen und der Menge der lokalen und entfernten Haloknoten je Nachbar. Die Umsetzung selbst kann in konstanter Zeit erfolgen. Daher ergibt sich approximativ die Aufwandsklasse $\mathcal{O}(|\text{neighbor}^*(P_p)| \cdot (|H_l| + |H_r|))$ für diesen Abschnitt im Algorithmus.

Die letzte Phase initialisiert nur noch die Vektoren für die rechte Seite b des Gleichungssystems sowie den Startlösungsvektor x. Die Startlösung für die iterativen Gleichungslöser in den folgenden Abschnitten wird mit dem Nullvektor vorbelegt.

Die Gesamtlaufzeit von Algorithmus 3.4 wird im Wesentlichen bestimmt von der Menge der zu betrachtenden Knoten aus S_1 mit den Nachbarknoten, die nicht der Dirichletrandbedingung genügen (Phase zwei) sowie der Anzahl der Partitionshaloknoten über alle Nachbarn (Phase drei), da die hier zu bearbeitende Datenmenge am größten ist. Daher ergibt sich als Gesamtaufwand die Klasse $\mathcal{O}(|S_1| + 2 \cdot |E_{\Omega_p}| + |\text{neighbor}^*(P_p)| \cdot (|H_l| + |H_r|))$. Die Zahl der lokalen Gebietsknoten überwiegt in der Praxis bei weitem die Menge der Haloobjekte, so dass sich der Gesamtaufwand praktisch auf $\mathcal{O}(|S_1|) \leq$

[16] Zugriffsmuster auf die Matrixeinträge und Ausnutzung von Cacheeffizienz
[17] präkonditionierte Gleichungslöser wie z.B. mittels einer LR-Zerlegung

$\mathcal{O}(|N_{\Omega_p}|) = \mathcal{O}(n)$ reduziert. Kommunikation ist für diesen Algorithmus nicht nötig. Das Aufsetzen der verteilten CRS-Struktur für das System $Ax = b$ ist somit in linearer Zeit möglich.

Abbildung 3.2 zeigt die (verteilte) CRS-Matrixstruktur für ein Anwendungsproblem der Temperaturdynamik (ähnlich zu Problemstellung 4.1), welches parallel auf vier Rechenknoten gerechnet wurde. Die Achsen in den Diagrammen entsprechen den Indices i, j eines Matrixeintrages $a_{i,j}$. Auf der linken Seite ist die Struktur der lokalen CRS-Darstellung der Partitionen 1-4 zu sehen. Jede Partition besitzt im Durchschnitt 13000 Knoten in der Geometrie. Die rechte Seite zeigt die Übertragung der lokalen Anteile der Matrizen in einen globalen Namensraum, in dem jeder Knoten ein global eindeutiges Identifikationsmerkmal besitzt. Es ist zu beobachten, dass die Struktur der Matrix im lokalen und im globalen Namensraum im Kernbereich quasi übereinstimmen. Außerhalb des Kernbereiches liegen die Partitionsrandknoten. Die "chaotische" Anordnung der Matrixeinträge in Abbildung 3.2 ergibt sich aus der Auswahl der Nebendiagonalelemente und Überführung in die matrixlokale Indexnummerierung ohne besondere Kriterien aus Algorithmus 3.4. Die Matrixbandbreite ist praktisch unbeschränkt. Solche Strukturen sind in der Praxis aus Effizienzgründen (Cache-Zugriffsmuster) ungünstig; auf das Ergebnis der Rechnung haben sie aber letztendlich keinen Einfluss. Daher erfolgt meistens eine Umsortierung der Einträge durch geeignete Methoden, z.B. mit Hilfe sog. raumfüllender Kurven (space-filling curves, *SFC*, vgl. [Wie03, Zum01, Zum03]) oder Multilevel-Verfahren (vgl. [KK98b]). In Abbildung 3.3 ist ein Beispiel einer Matrixstruktur dargestellt, die aus einer "chaotischen" Anordnung (links) unter Anwendung von raumfüllenden Kurven erzeugt wurde (rechts). Die effiziente Anordnung der Matrixeinträge in einer einfachen Bandstruktur zur Performancesteigerung umfasst einen großen und komplexen Themenbereich. Da die Untersuchung solcher Methoden nicht Gegenstand dieser Arbeit ist, wird auf diesen Aspekt nicht weiter eingegangen.

Hilfsoperationen

Es gibt drei wesentliche Operationen, die bei einem *parallelen* Algorithmus für die Lösung eines Gleichungssystems $Ax = b$ notwendig sind und maßgeblich die Geschwindigkeit des gesamten Verfahrens bestimmen. Diese Kernoperationen sind die Matrix-Vektor Multiplikation, das Skalarprodukt und die Halo-Aktualisierung. Die parallele Umsetzung dieser Operationen im verteilten Objektmodell soll in den folgenden Abschnitten näher erläutert werden.

Parallele Matrix-Vektor Multiplikation Eine Kernoperation eines iterativen Lösers ist die Matrix-Vektor Multiplikation (MVM), die gerade auf schwach besetzten Matrizen, die kompakt im Speicher eines Rechenknotens gehalten werden, einen großen Einfluss auf die Laufzeit des Gesamtalgorithmus hat. Das Problem bei Verwendung der CRS-Struktur bzw. einer generellen, kompakten Speichermethode ist die so genannte indirekte Adressierung beim Zugriff auf die Datenelemente sowohl der Matrix A als auch des Vektors x. Indirekte Adressierung zählt zu den teuersten Operationen bei numerischen

Abbildung 3.2 Verteilte und globale CRS-Indexdarstellung mit vier Partitionen

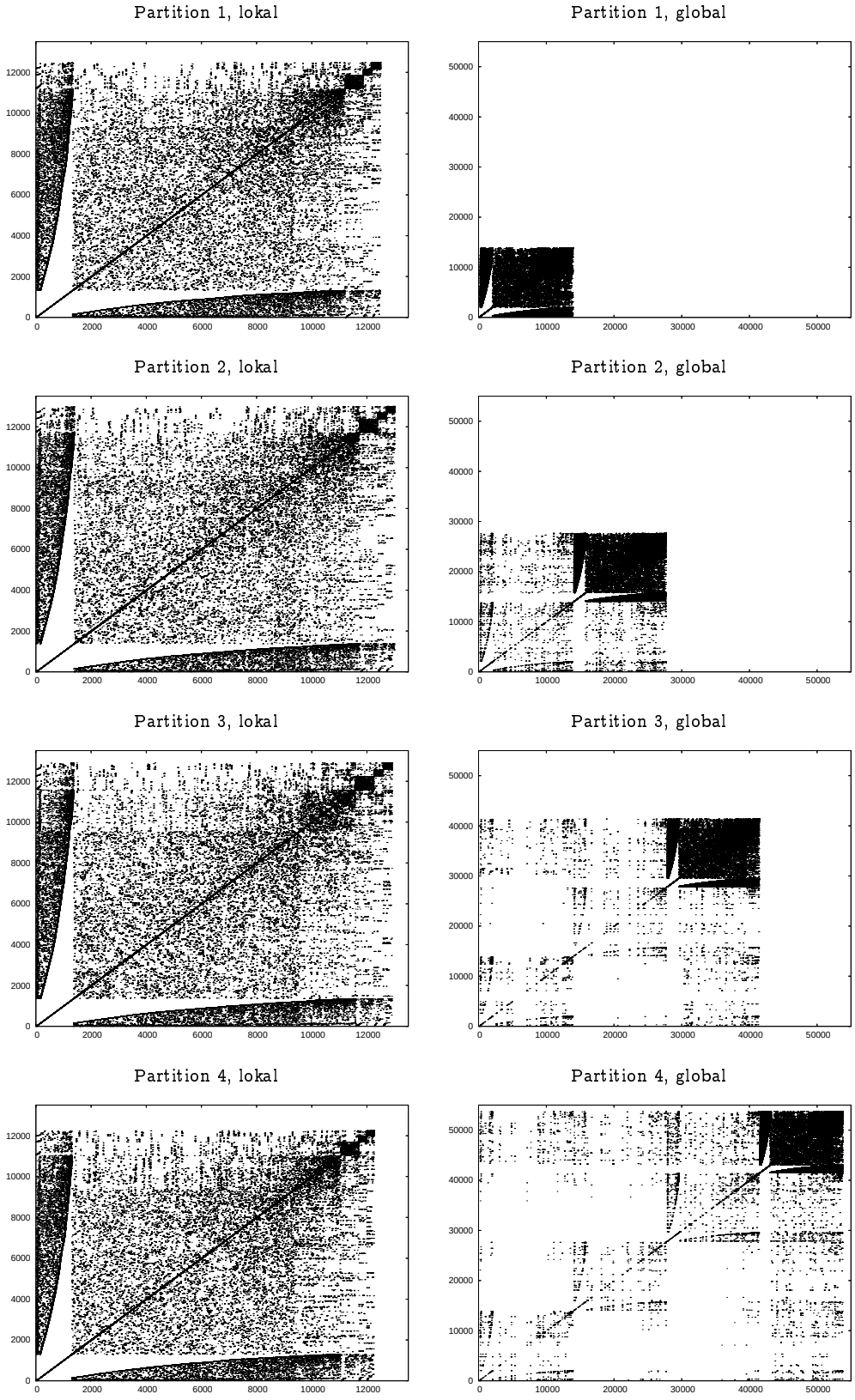


Abbildung 3.3 Matrixumstrukturierung durch eine raumfüllende Kurve (SFC)

Algorithmus 3.5 : Thread-parallele Matrix-Vektor Multiplikation $Ax = b$

Eingabe : partitionslokaler Anteil der globalen Matrix A im CRS-Format,
partitionslokaler Anteil des globalen Vektors x,
$q \in \mathbb{N}$ sei die Anzahl der Threads,
$t \in \{1, \ldots, q\}$ sei die Threadnummer

Ausgabe : Matrix-Vektor Produkt b

```
1  parallel proc MVM begin
2      m ← |{n ∈ N_Ωp | n.p ≠ "Dirichlet"}|;
3      for i = (t − 1) m/q + 1, ..., t m/q do
4          y ← 0;
5          for j = CRS(A).r[i], ..., CRS(A).r[i + 1] − 1 do
6              y ← y + CRS(A).v[j] · x[CRS(A).c[j]];
7          end
8          b[j] ← y;
9      end
10 end proc
```

Anwendungen, da sie sehr häufig die Caches der Prozessoren invalidieren kann. Dies ist auch der Grund, warum bei realen numerischen Anwendungen eine sehr große Diskrepanz zwischen der theoretischen Peak-Performance und der in der Praxis tatsächlich erreichbaren Leistung bei Verwendung kompakter Matrixspeichermethoden herrscht.

Algorithmus 3.5 beschreibt den Algorithmus zur Matrix-Vektor Multiplikation (auf der Basis des spezialisierten Objektmodells) für das CRS-Format in einer thread-parallelen Version für den Einsatz auf Mehrprozessorsystemen. Die Zählschleife in Zeile 3 teilt die einzelnen Zeilen der Matrix A blockweise den zur Verfügung stehenden q Threads zu, da diese unabhängig für die Berechnung des Skalarproduktes (Zeile 5-7) verwendet werden können. Zeile 6 nutzt die indirekte Adressierung beim Zugriff auf den x-Vektor. Der Algorithmus kommt gänzlich ohne Kommunikation zwischen den einzelnen Re-

chenknoten aus. Dies liegt daran, dass durch die Halostruktur jeder Gebietsknoten alle seine Nachbarknoten kennt, d.h. für einen Knoten existiert seine zugehörige Zeile in der Matrix A lokal. Ein Datentransport ist für diese Operation nicht notwendig.

Für die Laufzeitbestimmung von Algorithmus 3.5 wird die durchschnittliche Anzahl der Nebendiagonaleinträge pro Zeile benötigt, die mit $\text{avg}_i(\text{CRS}(A).r_{i+1} - \text{CRS}(A).r_i)$ bezeichnet werden soll. Damit ergibt sich nun als Aufwandsklasse $\mathcal{O}(\frac{m}{q} \cdot \text{avg}_i(\text{CRS}(A).r_{i+1} - \text{CRS}(A).r_i)) \leq \mathcal{O}(\frac{m}{q} \cdot c) \leq \mathcal{O}(m \cdot c) \leq \mathcal{O}(m)$, wobei m die Anzahl der betrachteten nicht-Dirichlet Elemente ist, und q die Anzahl der beteiligten Threads bezeichnet. Die Matrix-Vektor Multiplikation läuft also wie erwartet in linearer Zeit ab. Die Anzahl der Fließkommaoperationen c_{flop}, die benötigt werden, berechnet sich zu $2 \cdot \frac{m}{q}$ pro Thread, d.h. $2 \cdot m = 2 \cdot \text{CRS}(A).nnz$ pro Partition.

Paralleles Skalarprodukt Die zweite wichtige Kernoperation innerhalb eines iterativen, parallelen Gleichungslöseralgorithmus ist das globale Skalarprodukt $\langle a, b \rangle \in \mathbb{R}$ zweier Vektoren $a, b \in \mathbb{R}^n$. Im Unterschied zur Matrix-Vektor Multiplikation im verteilten Objektmodell benötigt das Ergebnis eines Skalarproduktes alle Werte eines verteilten Vektors, d.h. bei jeder Berechnung wird eine globale Kommunikationsoperation notwendig, um die Daten zusammenzutragen und das Ergebnis der Berechnung wieder an alle Rechenknoten bzw. Partitionen zu verteilen. Bei der Bildung des Skalarproduktes im verteilten Objektmodell ist zusätzlich zu beachten, dass Datenwerte von Knoten, die sich am Partitionsrand $\partial P_{i,j}$ oder im entfernten Halo $H_{j \to i}$ einer Partition P_i befinden, also im verteilten Fall mehrfach vorhanden sind, in die Berechnung nur einfach mit eingehen dürfen. Zu diesem Zweck wird ein zusätzliches Hilfsmittel, der sog. Zugehörigkeitsvektor einer Partition, benötigt.

DEFINITION 3.14 (ZUGEHÖRIGKEITSVEKTOR EINER PARTITION)
Sei $P_i \in \mathcal{P}_\Omega$ *eine Partition aus einem verteilten Netz* $\mathcal{M}_\Omega(n) = (\mathcal{P}_\Omega, \mathcal{H}_\Omega, \omega_\Omega)$. *Sei weiter* $\mathbb{K} \in P_i$ *eine Elementmenge der Partition mit* $m = |\mathbb{K}|$. *Ein Vektor* $z \in \{0,1\}^m$ *heißt Zugehörigkeitsvektor der Partition* P_i *bzgl. der Elementmenge* $\mathbb{K}$, *wenn für* $x_k \in \mathbb{K}, 1 \leq k \leq m$, *gilt*

$$z_k = \begin{cases} 1 & : \quad x_k \notin H_{j \to i} \wedge p = \min\{j\} \quad \textit{für } j \in \text{neighbor}^*(P_i) \\ 0 & : \quad \text{sonst} \end{cases} \tag{3.12}$$

Ein Zugehörigkeitsvektor $z \in \{0,1\}^m$ gibt also für jedes Element einer Partition an, ob es zu dieser Partition gehört oder nicht, wobei die Zugehörigkeit dadurch definiert ist, dass im Namensraum das Element derjenigen Partition angehört, die die kleinste Nummer über alle Partitionen besitzt, die dieses Element geometrisch in der Teiltetraedierung enthalten.

Algorithmus 3.6 zeigt (auf der Basis des spezialisierten Objektmodells) eine Realisierung einer thread-parallelen und verteilten Berechnung des Quadrates eines Skalarproduktes $\langle a, b \rangle^2 = c \in \mathbb{R}$, wobei die Vektoren $a, b \in \mathbb{R}^m$ entsprechend der Diskretisierung und Partitionierung verteilt vorliegen. Es ist u.U. sinnvoller, statt des echten Skalarproduktes das Quadrat zu berechnen, da in manchen Algorithmenformulierungen die

Algorithmus 3.6 : Globale, thread-parallele Skalarproduktberechnung $c = \langle a, b \rangle^2$

Eingabe : partitionslokale Vektoren $a, b \in \mathbb{R}^m$,
Zugehörigkeitsvektor $z \in \{0,1\}^m$,
$t \in \{1, \ldots, q\}$ sei die Threadnummer aus $q \in \mathbb{N}$ Threads,
$p \in \{1, \ldots, P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen
Ausgabe : globales Skalarproduktquadrat $c \in \mathbb{R}$

```
1 parallel proc DotSquare begin
2 |   s_t ← 0;
3 |   for i = (t−1)·m/q + 1, ..., t·m/q do  s_t ← s_t + z[i] · a[i] · b[i];
4 |   Berechne lokalen Anteil: s_p ← ⨁^l s_{1≤t≤q};
5 |   Berechne globalen Wert: c ← ⨁^g s_{1≤p≤P};
6 end proc
```

Berechnung der Wurzel eingespart werden kann und somit eine teure Fließkommaoperation vermieden wird.

In Algorithmus 3.6 erfolgt in Zeile 3 zuerst die Berechnung des thread-lokalen Anteils des Skalarproduktes mittels des aus Definition 3.14 eingeführten Zugehörigkeitsvektors $z \in \{0,1\}^m$. Durch diesen Vektor z gehen nur die Berechnungsanteile einer Partition ein, die der Partition *logisch* gehören. Hat jeder Thread seinen Anteil lokal berechnet, so erfolgt in Zeile 4 die partitionslokale Summierung s_p der thread-lokalen Teilsummen s_t über alle Threads $t \in \{1, \ldots, q\}$ der Partition. Dieser Schritt benötigt einen koordinierten Ablauf (atomar und exklusiv), was durch die Operation $\bigoplus^l$ symbolisiert wird. Als letztes bleibt die globale Aufsummierung und Verteilung c über alle Partitionen in Zeile 5, damit jeder Thread und damit jede Partition das Gesamtergebnis der Skalarproduktberechnung propagiert bekommt. Für solche Operationen existieren für gewöhnlich in den Kommunikationsbibliotheken besondere Funktionen, sog. *Reduce- und Gather-operationen*, die dies effizient durchführen. Die globale Kommunikationsoperation wird im Algorithmus mit dem Symbol $\bigoplus^g$ gekennzeichnet.

Die Berechnung des Quadrates des Skalarproduktes läuft in linearer Zeit ab, da thread-parallel $\frac{m}{q}$ Elemente bearbeitet und konstanter Aufwand c für die Kommunikation benötigt werden, d.h. die Laufzeit liegt in $\mathcal{O}(\frac{m}{q}+c) \leq \mathcal{O}(m)$. Die Anzahl der Fließkommaoperationen c_{flop} beträgt für alle Threads und alle Partitionen $q \cdot (3 \cdot \frac{m}{q}) + (q-1) + (n-1) = 3m + (q-1) + (n-1)$, wobei n hier die Anzahl der Partitionen im Gesamtsystem wiedergibt.

Halo-Aktualisierung Die letzte wichtige Kernoperation für einen iterativen, parallelen Gleichungslöser beschäftigt sich weniger mit einem algebraischen Problem, als viel mehr mit der Abbildung des numerischen Verfahrens auf einem Parallelrechner. Es handelt sich hierbei um die sog. *Halo-Aktualisierung* bzw. das *Halo-Update*. Diese Operation sorgt dafür, dass in jeder Iteration des Lösungsalgorithmus die berechneten Daten aus der vorherigen Iteration im entfernten Halo $H_{j \to i}$ bzw. im mit dem Halo assoziierten

Algorithmus 3.7 : Abgleichalgorithmus über das entfernte Halosystem $\{H_{j\to i}\}$

Eingabe : partitionslokaler Anteil von $CRS(A)$ mit m Zeilen,
partitionslokaler Update-Vektor $v \in \mathbb{R}^m$,
$p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega)$

Ausgabe : aktualisierter Update-Vektor $v \in \mathbb{R}^m$

```
1  parallel proc HaloUpdate begin
2      foreach p' ∈ neighbor*(P_p) do
3          for i = 1,...,|CRS(A).H_s^p'| do
4              j ← CRS(A).H_{s_i}^p';
5              s_i^p' ← v_j;
6          end
7      end
8      Datenaustausch mit allen Nachbarn P_p': {r^p'} ← Exchange ({s^p'}, C_p);
9      foreach p' ∈ neighbor*(P_p) do
10         for i = 1,...,|CRS(A).H_r^p'| do
11             j ← CRS(A).H_{r_i}^p';
12             v_j ← r_i^p';
13         end
14     end
15 end proc
```

Vektor einer Partition P_i von seinem Nachbarn P_j vorhanden sind. Die Laufzeit des Halo-Updates hängt im hohen Maße von der aktuellen Partitionierung und damit von der Zusammensetzung des Kommunikationsschedules ab. Als wichtigste Komponenten für den Sende- und Empfangsteil werden die für die CRS-Struktur des Matrixsystems A berechneten Indexvektoren der Halo-Nachbarobjekte $CRS(A).H_s^{p'}$ und $CRS(A).H_r^{p'}$ benötigt.

Algorithmus 3.7 zeigt die Struktur der Halo-Aktualisierung. Der Algorithmus besteht aus drei Phasen. In Phase eins (Zeile 2-7) wird der Sendevektor $s^{p'}$ für jede Nachbarpartition $P_{p'}$ der aktuellen Partition P_p konstruiert. Dazu müssen die Indizes für die Komponenten des Vektors v, der durch die Update-Funktion parallel aktualisiert werden soll, bestimmt werden. Diese liegen im Indexvektor der CRS-Struktur $CRS(A).H_s^{p'}$ vor. Die anschließende Übertragung der jeweiligen v-Komponenten in den Sendevektor bildet dann den Abschluss des Schleifenrumpfes. Phase zwei (Zeile 8) besteht aus dem eigentlichen Datenaustausch mittels des partitionslokalen Anteils $C_p \in \mathcal{C}(\mathcal{P}_\Omega)$ des Kommunikationsschedules unter Verwendung von Algorithmus 3.2. Die letzte Phase (Zeile 9-14) fügt die von allen Nachbarpartitionen $P_{p'}$ empfangenen Daten aus den Vektoren $\{r^{p'}\}$ neu in den zu aktualisierenden Vektor v ein. Dazu wird der entsprechende

Gegenpart $CRS(A).H_r^{p'}$ aus der CRS-Struktur verwendet.

Der Vektor v enthält nach Ausführung von Algorithmus 3.7 Teilkomponenten v_i, die lesend (innerer/lokaler Halo, $\sum_{p'} H_s^{p'}$), schreibend (entfernter/kopierter Halo, $\sum_{p'} H_r^{p'}$) und unverändert ($v \in \mathbb{R}^m, m \gg \sum_{p'}(|H_s^{p'}|+|H_r^{p'}|)$) bearbeitet wurden. Da bei der Konstruktion des Halosystems $\mathcal{H}_p$ einer Partition P_p, ein Netzobjekt (ein Knoten oder eine Kante) Teil mehrerer Nachbarpartitionen sein kann[18], muss eine spezielle Betrachtung dieser Objekte beim Halo-Update erfolgen. Für gewöhnlich werden aber die zu aktualisierenden Daten auf allen Partitionen, in denen das betrachtete Netzobjekt vorhanden ist, auf äquivalente Weise berechnet. Eine Unterscheidung, welche der berechneten Daten nun für die Update-Operation ausgewählt bzw. miteinander gemittelt/verknüpft werden, kann entfallen. Die letzte Schreiboperation über alle Nachbarpartitionen legt somit die Komponente im Vektor fest.

Die Laufzeitklasse von Algorithmus 3.7 wird bestimmt durch die Halosystemgröße (s. Konstruktion in Algorithmus 3.4) jeder Partition $P_{p'} \in \text{neighbor}(P_p)$

$$|\mathcal{H}_p| = \sum_{p'}(|H_{p\to p'}| + |H_{p'\to p}|) = \sum_{p'}(|CRS(A).H_s^{p'}| + |CRS(A).H_r^{p'}) = c_{H^p}$$

und durch dem Aufwand $c_{comm}(C_p)$ für den eigentlichen Datenaustausch durch die Punkt-zu-Punkt Kommunikationsoperationen. Dies ergibt $\mathcal{O}(c_{H^p}+c_{comm}(C_p))$ als Aufwandsklasse. Da aber bei realen Problemstellungen die Gesamtgröße des Halosystems $\mathcal{H}_p$ sehr viel kleiner ist als die Partition P_p selbst, wird der Aufwand für die Halo-Aktualisierung durch die Kosten für die Kommunikation $c_{comm}(C_p)$ dominiert. D.h. die Aufwandsklasse liegt bei realen Anwendungen in $\mathcal{O}(c_{comm}(C_p))$. Der Aufwand für den verwendeten Speicher im Algorithmus beträgt $\mathcal{O}(2 \cdot \text{avg}_{p'\in\text{neighbor}^*(P_p)}(|H_s^{p'}| + |H_r^{p'}|) = \mathcal{O}(\text{avg}_{p'\in\text{neighbor}^*(P_p)}(|H_s^{p'}|+|H_r^{p'}|)$, da zusätzlich zu den Indexstrukturen für die Halodaten $H_{s,r}^{p'}$ noch Speicher für die Sende- und Empfangsvektoren $s^{p'}$ und $r^{p'}$ mit einbezogen werden muss.

Als Letztes bleibt noch zu bemerken, dass die Kopierschleifen im Algorithmus in den Zeilen 3-6 und 10-13 thread-parallel ausgeführt werden können, da die zu bearbeitenden Daten unabhängig sind. Allein der Datenaustausch mit anderen Partitionen darf nur von einem Thread ausgeführt werden, es sei denn, es liegt eine sog. thread-safe Implementierung für die Kommunikationsbibliothek vor, in der mehrere Threads eine Kommunikationsoperation zeitgleich durchführen können. Der Beginn und das Ende des Datenaustausches stellen somit Synchronisationspunkte für alle Threads einer Partition dar. Die Halo-Update-Operation ist ein Synchronisationspunkt für die Partitionen. Durch geeignete Aufteilung der zu bearbeitenden Mengen, also den Vektoren und der Matrix, kann eine Entkopplung von Kommunikation und Rechnung erfolgen. Berechnung und Kommunikation können damit parallel stattfinden und die Gesamtlaufzeit eines Gleichungslösers minimieren (communication hiding).

[18] in den lokalen Namensräumen existieren mehrere Objektnamen für das gleiche geometrische Objekt

Gleichungslöser der CG-Klasse

Mit Hilfe der im vorherigen Abschnitt eingeführten Hilfsoperationen kann nun eine Formulierung von iterativen Gleichungslösern im verteilten Objektmodell erfolgen. Die Modellierungen der Algorithmen sind derart gestaltet, dass sie immer eine thread-parallele und verteilte Version darstellen. Dies bedeutet, dass ein Löseralgorithmus von einem Thread ausgeführt wird, wobei (möglicherweise) mehrere Threads eine Partition bearbeiten, die auf einen Rechenknoten abgebildet ist. Um zu kennzeichnen, welche Operationen (Skalar-, Vektor- oder Matrizenoperationen) durch Threads parallel bearbeitet werden können, wird in den Algorithmenformulierungen der spezielle Zuweisungsoperator $\overset{t}{\leftarrow}$ benutzt. Durch Aufteilung der Daten in disjunkte Mengen, die unabhängig von den Threads bearbeitet werden können, kann die Thread-Parallelität immer erreicht werden. Tritt ein Synchronistationpunkt im Algorithmus auf, der nur von einem Thread der Partition ausgeführt werden darf, so wird hierzu der Zuweisungsoperator $\overset{p}{\leftarrow}$ benutzt.

Algorithmus 3.8 zeigt das *Verfahren der konjugierten Gradienten (conjugate(d) gradient method, CG)* in seiner unmodifizierten Standardform, d.h. ohne Präkonditionierung (vgl. u.a. [Saa03]). Das CG-Verfahren kann immer dann eingesetzt werden, wenn die Matrix A symmetrisch und positiv-definit ist. Bei den für diese Arbeit betrachteten Problemstellungen für instationäre Strömungen treten solche Matrizen bspw. bei Bestimmung von Druck- und Geschwindigkeitsfeldern auf.

Der Algorithmus besteht aus einer Initialisierungsphase (Zeile 2-4), einer iterativen Berechnungsphase (Zeile 5-15) und der für das Objektmodell notwendigen Rücktransformationsphase (Zeile 16-18), in der das berechnete Ergebnis (die Komponenten des Lösungsvektors x) wieder an die anhängigen Daten eines geometrischen Objektes (s. Matrixtransformation, CRS-Konstruktion) zurück übertragen werden. Vektoren und Matrizen, auf die thread-parallel zugegriffen wird, sind in Fettschrift hervorgehoben.

Auf die Eigenschaften des CG-Verfahrens soll hier nicht weiter eingegangen werden, diese können in der umfangreichen Literatur zur Numerik von linearen Gleichungssystemen (z.B. [Hac93, GL96, Saa03, SK04, Mei05]) nachgeschlagen werden. Erwähnt sei hier allerdings der Aufwand bzw. die Zahl der Iterationen, die das Standard-CG-Verfahren benötigt. In der Theorie konvergiert das Verfahren nach genau n Schritten, wenn n die Zahl der Unbekannten ist. In der Praxis, die zudem keine exakte Arithmetik kennt, ist eine Berechnung nur bis zu einer gewissen Genauigkeit $0 < \epsilon \ll 1$ nötig (bspw. $\epsilon \leq 10^{-8}$ oder selten sogar bis zur Maschinengenauigkeit). Man kann dann die nötige Anzahl der Iterationen k für das CG-Verfahren bei Anwendung auf das Gleichungssystem $Ax = b$ bestimmen durch

$$k \leq \frac{1}{2}\sqrt{\xi(A)}\ln\left(\frac{2}{\epsilon}\right) + 1 =: \mathcal{I}_{CG}(A, \epsilon),$$

wenn $\xi(A) = \frac{\lambda_{max}}{\lambda_{min}}$ die Konditionszahl der Matrix A bzgl. der Euklid-Norm bezeichnet, die aus dem größten und den kleinsten Eigenwert von A bestimmt werden kann.

Algorithmus 3.8 : Thread-paralleler und verteilter CG-Algorithmus

Eingabe : partitionslokaler Netzanteil $\mathcal{M}_{\Omega_p}$ mit m Netzknoten,
Zugehörigkeitsvektor $\mathbf{z} \in \{0,1\}^m$,
$t \in \{1,\ldots,q\}$ sei die Threadnummer aus $q \in \mathbb{N}$ Threads,
$p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega)$,
Genauigkeit $\epsilon \in \mathbb{R}$ und max. Iterationszahl $K \in \mathbb{N}$

Ausgabe : aktualisierte Netzgeometrie durch Lösungsvektor $\mathbf{x} \in \mathbb{R}^m$

```
1  parallel proc CGSolver begin
2      (CRS(A), b, x) <-p- SetupCRSSystem(p, M_Ωp);
3      u <-t- b; ρ ← 1; k ← 1;
4      α1 <-t- DotSquare(u, u, z, t, p); μ ← sqrt(α1);
5      while ρ > ε ∧ k < K do
6          u <-p- HaloUpdate(CRS(A), u, p, C_p);
7          w <-t- MVM(CRS(A), u, q, t);
8          α2 <-t,p- DotSquare(u, w, z, t, p);
9          x <-t- x + (α1/α2) u;
10         b <-t- b − (α1/α2) w;
11         α2 <-t,p- DotSquare(b, b, z, t, p);
12         u <-t- b + (α2/α1) u;
13         α1 ← α2; ρ ← sqrt(α1)/μ;
14         k ← k + 1;
15     end
16     for i = (t − 1) m/q + 1, ..., t m/q, n_i ∈ {n ∈ N_Ωp | n.p ≠ "Dirichlet"} do
17         n_i.x ← x_j : (n_i, j) ∈ CRS(A).lut;
18     end
19 end proc
```

Die Anzahl der benötigten Fließkommaoperationen[19] c_{flop}^{CG} für die thread-parallele und verteilte Version des Algorithmus bestimmt sich wie folgt. In der Initialisierungsphase werden *pro Thread* $\frac{3m+(q-1)+1}{q}$ Fließkommaoperationen verwendet (m bezeichne die Anzahl der Unbekannten und q die Zahl der Threads pro Partition). Die Berechnungsphase benötigt bei k Iterationen

$$k \cdot \left(\frac{2s}{q} + 2 \cdot \frac{3m + (q-1) + 1}{q} + 3 \cdot \left(1 + \frac{2m}{q} \right) + 2 \right) = k \cdot \left(\frac{2s + 12m + 7q}{q} \right)$$

[19] Als echte Fließkommaoperationen gelten u.a. Addition und Subtraktion, Multiplikation, Division und Wurzelziehen. Zuweisungsoperationen zählen nicht dazu. Schleifenzählvariablen sind bei keiner Fließkommaoperation beteiligt.

Fließkommaoperationen (s sei die Anzahl der von Null verschiedenen Einträge in der Matrix A). Unter Voraussetzung, dass die Zahl der Unbekannten m auf allen Partitionen P gleich ist, ergibt sich nun für die Gesamtanzahl der Fließkommaoperationen $c_{flop}^{CG} = P \cdot (3m+q+k \cdot (2s+12m+7q))$. Mit dieser Formel kann der Gesamtaufwand an Fließkommaoperationen des Algorithmus gemessen werden, welcher in der Praxis üblicherweise in der Einheit *MFLOP/s (million floating point operations per second)* angegeben wird.

Um die Zahl der notwendigen Iterationen bis zur Konvergenz des CG-Verfahrens zu reduzieren, modifiziert man das Standardverfahren derart, dass die Matrix A wesentlich besser konditioniert ist. Als Beispiel sei hier die sog. *Präkonditionierung mittels Skalierung (scaling PCG)*, auch bekannt als *Diagonal- oder Jakobi-Präkonditionierung*, im verteilten Objektmodell vorgestellt (s. z.B. [GL96, Saa03]). Mit Hilfe der Skalierung werden die Zeileneinträge der Matrix und der rechten Seite bzgl. des Hauptdiagonalelementes in jeder Iteration normiert.

Algorithmus 3.9 zeigt das präkonditionierte CG-Verfahren mit Hilfe der Skalierung. Auch dieser Algorithmus besteht wie das Standard-CG-Verfahren aus drei Phasen. Initialisierung (Zeile 2-7), Berechnungsphase (Zeile 8-19) und Rücktransformation (Zeile 20-22). In der Initialisierungsphase des skalierenden, präkonditionierten CG-Verfahrens (SPCG) wird, im Gegensatz zur Initialisierungphase des CG-Verfahrens, zusätzlich der Diagonal- und Skalierungsvektor (d und s) aus der Matrix A bzw. aus den zu betrachtenden nicht-Dirichlet Netzelementen bestimmt (Zeile 3-5). Die Verwendung dieser neu eingeführten Vektoren findet in der Berechnungsphase in den Zeilen 14-16 statt. Die Division in Zeile 14 des Algorithmus ist komponentenweise ($s_i = b_i/d_i$) zu verstehen. Der Rest des Algorithmus gleicht dem Standard-Verfahren in Algorithmus 3.8.

Die Theorie (vgl. [GL96, Saa03, Mei05]) zeigt, das im schlechtesten Fall die Anzahl der benötigten Iterationen k, um die Fehlernorm des Residuenvektors auf eine bestimmte Genauigkeit ϵ zu drücken, gleich dem des CG-Verfahrens ist, also $\mathcal{I}_{SPCG}(A, \epsilon) = \mathcal{I}_{CG}(A, \epsilon)$. In der Praxis ergibt sich allerdings für das SPCG-Verfahren ein wesentlich besseres Verhalten. Hier kann, je nachdem welche Methode zur Berechnung des Skalierungsvektors benutzt wird (Diagonalelement, Zeilen-/Spaltenskalierung bzgl. Betragssummennorm, euklidischer Norm oder Maximumsnorm), eine Reduzierung der benötigten Iterationszahl auf $\mathcal{I}_{SPCG}^{praxis}(A, \epsilon) \approx \frac{2}{3} \cdot \mathcal{I}_{CG}(A, \epsilon)$ erreicht werden. Eine weitere Reduzierung ist mit komplexeren Präkonditionierungen, wie bspw. durch *unvollständige Cholesky-Zerlegung (IC)* oder *unvollständige LR-Zerlegung mit Fill-in Level k (ILR(k))*, möglich. Eine Reduzierung auf bis zu $\mathcal{O}(\sqrt{\mathcal{I}_{CG}(A, \epsilon)})$ ([Hac93, Saa03]) Iterationen kann durch solche Präkonditionierungsverfahren erreicht werden. Solche Verfahren sollen aber hier nicht weiter behandelt werden, da sie sich in der Struktur, abgesehen von der (parallelen) Berechnung der Präkonditionierungsmatrix, den bereits vorgestellten Standard-CG-Verfahren und SPCG-Verfahren ähneln.

Die Anzahl der Fließkommaoperationen des SPCG-Verfahrens aus Algorithmus 3.9 setzt sich nun wie folgt zusammen. In der Initialisierungsphase werden pro Thread $\frac{m}{q} +$

Algorithmus 3.9 : Thread-paralleler und verteilter, skalierender PCG-Algorithmus

Eingabe : partitionslokaler Netzanteil $\mathcal{M}_{\Omega_p}$ mit m Netzknoten,
Zugehörigkeitsvektor $\mathbf{z} \in \{0,1\}^m$,
$t \in \{1,\ldots,q\}$ sei die Threadnummer aus $q \in \mathbb{N}$ Threads,
$p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega)$,
Genauigkeit $\epsilon \in \mathbb{R}$ und max. Iterationszahl $K \in \mathbb{N}$

Ausgabe : aktualisierte Netzgeometrie durch Lösungsvektor $\mathbf{x} \in \mathbb{R}^m$

1 **parallel proc** `SPCGSolver` **begin**
2 $(\mathrm{CRS}(\mathbf{A}), \mathbf{b}, \mathbf{x}) \overset{p}{\leftarrow}$ `SetupCRSSystem`$(p, \mathcal{M}_{\Omega_p})$;
3 **for** $i = (t-1)\frac{m}{q}+1,\ldots,t\frac{m}{q},\ n_i \in \{n \in N_{\Omega_p} \mid n.p \neq$ *"Dirichlet"*$\}$ **do**
4 $d_i \leftarrow n_i.a_h;\ s_i \leftarrow \frac{b_i}{d_i}$;
5 **end**
6 $\mathbf{u} \overset{t}{\leftarrow} \mathbf{b};\ \rho \leftarrow 1;\ k \leftarrow 1$;
7 $\alpha_1 \overset{t}{\leftarrow}$ `DotSquare`$(\mathbf{u},\mathbf{u},\mathbf{z},t,p);\ \mu \leftarrow \sqrt{\alpha_1}$;
8 **while** $\rho > \epsilon \wedge k < K$ **do**
9 $\mathbf{u} \overset{p}{\leftarrow}$ `HaloUpdate`$(\mathrm{CRS}(\mathbf{A}),\mathbf{u},p,C_p)$;
10 $\mathbf{w} \overset{t}{\leftarrow}$ `MVM`$(\mathrm{CRS}(\mathbf{A}),\mathbf{u},q,t)$;
11 $\alpha_2 \overset{t,p}{\leftarrow}$ `DotSquare`$(\mathbf{u},\mathbf{w},\mathbf{z},t,p)$;
12 $\mathbf{x} \overset{t}{\leftarrow} \mathbf{x} + \frac{\alpha_1}{\alpha_2}\mathbf{u}$;
13 $\mathbf{b} \overset{t}{\leftarrow} \mathbf{b} - \frac{\alpha_1}{\alpha_2}\mathbf{w}$;
14 $\mathbf{s} \overset{t}{\leftarrow} \mathbf{b}/\mathbf{d}$;
15 $\alpha_2 \overset{t,p}{\leftarrow}$ `DotSquare`$(\mathbf{s},\mathbf{b},\mathbf{z},t,p)$;
16 $\mathbf{u} \overset{t}{\leftarrow} \mathbf{s} + \frac{\alpha_2}{\alpha_1}\mathbf{u}$;
17 $\alpha_1 \leftarrow \alpha_2;\ \rho \leftarrow \frac{\sqrt{\alpha_1}}{\mu}$;
18 $k \leftarrow k+1$;
19 **end**
20 **for** $i = (t-1)\frac{m}{q}+1,\ldots,t\frac{m}{q},\ n_i \in \{n \in N_{\Omega_p} \mid n.p \neq$ *"Dirichlet"*$\}$ **do**
21 $n_i.x \leftarrow x_j : (n_i, j) \in \mathrm{CRS}(\mathbf{A}).\mathrm{lut}$;
22 **end**
23 **end proc**

$\frac{3m+(q-1)+1}{q}$ Operationen benötigt. Pro Thread werden in der Berechnungsphase bei k Iterationen

$$k \cdot \left(\frac{2s}{q} + 2 \cdot \frac{3m+(q-1)+1}{q} + 3 \cdot \left(1 + \frac{2m}{q}\right) + \frac{m}{q} + 2\right) = k \cdot \left(\frac{2s+13m+7q}{q}\right)$$

Fließkommaoperationen benötigt. Die Gesamtanzahl auf allen Partitionen P ergibt sich damit zu $c^{\mathrm{SPCG}}_{\mathrm{flops}} = P \cdot (4m + q + k \cdot (2s + 13m + 7q))$.

Das CG-Verfahren sowie SPCG bzw. seine verwandten präkonditionierten Verfahren lösen iterativ Gleichungssysteme $Ax = b$, deren Koeffizientenmatrix A symmetrisch und positiv-definit sind. Im Umfeld der strömungsmechanischen Problemstellungen treten allerdings auch Matrizen auf, die nicht notwendigerweise symmetrisch, nur semi-definit oder einfach regulär sind. Solche Matrizen entstehen z.B. bei der Diskretisierung des konvektiven Terms (Transportschritt) in der Navier-Stokes Gleichung (antisymmetrische Matrix). Um auch solche Gleichungssysteme lösen zu können, erfolgt hier die Vorstellung des *stabilisierenden, bi-konjugierten Gradientenverfahren (bi-conjugate(d) gradient method, stabilized (BiCGStab))* im verteilten Objektmodell. Das BiCGStab-Verfahren ist aus numerischer und programmtechnischer Sicht wesentlich aufwändiger als die weiter oben vorgestellten Verfahren, da es keine besonderen Eigenschaften der Matrix A ausnutzen kann.

Algorithmus 3.10 zeigt das bekannte BiCGStab-Verfahren in einer thread-parallelen, verteilten Version. Auch hier ist der Algorithmus wieder in die drei klassischen Phasen Initialisierung (Zeile 2-6), Berechnungsphase (Zeile 7-25) und Rücktransformation (Zeile 26-28) unterteilt. Es fällt sofort auf, dass der Kommunikationsaufwand in der Berechnungsphase mit sechs globalen Skalarproduktbildungen und zwei Halo Aktualisierungen über zwei verschiedene Vektoren (u und s) deutlich höher ist als beim CG- bzw. SPCG-Verfahren mit je zwei Skalarproduktbildungen und einer Halo Aktualisierung. Hinzu kommen vom Rechenaufwand zwei statt einer Matrix-Vektor Multiplikation.

Das BiCGStab-Verfahren ist dafür bekannt, dass sich das Konvergenzverhalten nichtdeterministisch verhält. Eine Abschätzung für die Iterationsanzahl bei vorgegebener Genauigkeit $\mathcal{I}_{\mathrm{BiCGStab}}(A, \epsilon)$ wie bei $\mathcal{I}_{\mathrm{CG}}(A, \epsilon)$ oder $\mathcal{I}_{\mathrm{SPCG}}(A, \epsilon)$ ist daher nicht möglich. Ein Vorteil des Verfahrens ist jedoch, dass wegen seiner stabilisierenden Eigenschaft die mitunter abschnittsweise, stark oszilierende Residuumsnormfolge wie beim CG- oder SPCG-Verfahren eingeschränkt oder sogar vermieden wird. Eine ausführliche Diskussion des Verfahrens und der Vergleich mit anderen iterativen Gleichungslösern (z.B. *GMRES(k)*, *TFQMR* oder *QMRCGStab*) kann in der umfangreichen Literatur zu numerischen Gleichungslösern gefunden werden (s. z.B. [Saa03, GL96, Mei05]).

Es bleibt noch die Anzahl der Gesamtfließkommaoperationen $c_{\mathrm{flops}}^{\mathrm{BiCGStab}}$ für das BiCGStab-Verfahren zu bestimmen. In der Initialisierungsphase werden pro Thread $2 \cdot \left(\frac{3m+(q-1)+1}{q}\right) + 1$ Operationen benötigt. Die Berechnungsphase des Algorithmus 3.10 besteht bei k Iterationen aus

$$k \cdot \left(2 \cdot \frac{2s}{q} + 6 \cdot \frac{3m + (q-1) + 1}{q} + 6 \cdot \frac{2m}{q} + 10\right) = k \cdot \left(\frac{4s + 30m + 16q}{q}\right)$$

Fließkommaoperationen, die pro Thread ausgeführt werden müssen. Somit ergibt sich schließlich als Gesamtanzahl für P Partitionen $c_{\mathrm{flops}}^{\mathrm{BiCGStab}} = P \cdot (6m + 2q + 1 + k \cdot (4s + 30m + 16q))$ Operationen mit Fließkommaarithmetik.

Eine Leistungsbewertung der hier vorgestellten iterativen Gleichungslöser anhand eines akademischen Testproblems befindet sich in Abschnitt 4.2. Die Löser wurden dafür in

Algorithmus 3.10 : Thread-paralleler und verteilter BiCGStab-Algorithmus

Eingabe : partitionslokaler Netzanteil $\mathcal{M}_{\Omega_p}$ mit m Netzknoten,
Zugehörigkeitsvektor $\mathbf{z} \in \{0,1\}^m$,
$t \in \{1,\ldots,q\}$ sei die Threadnummer aus $q \in \mathbb{N}$ Threads,
$p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega)$,
Genauigkeit $\epsilon \in \mathbb{R}$ und max. Iterationszahl $K \in \mathbb{N}$

Ausgabe : aktualisierte Netzgeometrie durch Lösungsvektor $\mathbf{x} \in \mathbb{R}^m$

parallel proc `BiCGStabSolver` **begin**
  $(\mathrm{CRS}(\mathbf{A}), \mathbf{b}, \mathbf{x}) \overset{p}{\leftarrow}$ `SetupCRSSystem`$(p, \mathcal{M}_{\Omega_p})$;
  $\mathbf{r} \overset{t}{\leftarrow} \mathbf{b}$; $\mathbf{s} \overset{t}{\leftarrow} 0$; $\mathbf{u} \overset{t}{\leftarrow} 0$; $\mathbf{w} \overset{t}{\leftarrow} 0$;
  $\rho_0 \leftarrow 1$; $\alpha \leftarrow 1$; $\omega \leftarrow 1$;
  $\tau \overset{t,p}{\leftarrow}$ `DotSquare`$(\mathbf{b}, \mathbf{b}, \mathbf{z}, t, p)$; $\mu \leftarrow \sqrt{\tau}$;
  $\rho_1 \overset{t,p}{\leftarrow}$ `DotSquare`$(\mathbf{b}, \mathbf{r}, \mathbf{z}, p)$;
  while $\rho > \epsilon \wedge k < K$ **do**
    $\beta \leftarrow \frac{\rho_1 \alpha}{\rho_0 \omega}$;
    $\mathbf{d} \overset{t}{\leftarrow} \mathbf{u} - \omega\mathbf{w}$;
    $\mathbf{u} \overset{t}{\leftarrow} \mathbf{r} + \beta\mathbf{d}$;
    $\mathbf{u} \overset{p}{\leftarrow}$ `HaloUpdate`$(\mathrm{CRS}(\mathbf{A}), \mathbf{u}, p, C_p)$;
    $\mathbf{w} \overset{t}{\leftarrow}$ `MVM`$(\mathrm{CRS}(\mathbf{A}), \mathbf{u}, q, t)$;
    $\gamma \overset{t,p}{\leftarrow}$ `DotSquare`$(\mathbf{b}, \mathbf{w}, \mathbf{z}, p)$;
    $\alpha \leftarrow \frac{\rho_1}{\gamma}$;
    $\mathbf{s} \overset{t}{\leftarrow} \mathbf{r} - \alpha\mathbf{w}$;
    $\mathbf{s} \overset{p}{\leftarrow}$ `HaloUpdate`$(\mathrm{CRS}(\mathbf{A}), \mathbf{s}, p, C_p)$;
    $\mathbf{t} \overset{t}{\leftarrow}$ `MVM`$(\mathrm{CRS}(\mathbf{A}), \mathbf{s}, q, t)$;
    $\omega \overset{t,p}{\leftarrow} \frac{\texttt{DotSquare}(\mathbf{t},\mathbf{s},\mathbf{z},p)}{\texttt{DotSquare}(\mathbf{t},\mathbf{t},\mathbf{z},p)}$;
    $\rho_0 \leftarrow \rho_1$; $\rho_1 \overset{t,p}{\leftarrow} -\omega \cdot$ `DotSquare`$(\mathbf{b}, \mathbf{t}, \mathbf{z}, p)$;
    $\mathbf{d} \overset{t}{\leftarrow} \mathbf{x} + \alpha\mathbf{u}$;
    $\mathbf{x} \overset{t}{\leftarrow} \mathbf{d} + \omega\mathbf{s}$;
    $\mathbf{r} \overset{t}{\leftarrow} \mathbf{s} - \omega\mathbf{t}$;
    $\rho \overset{t,p}{\leftarrow} \frac{\sqrt{\texttt{DotSquare}(\mathbf{r},\mathbf{r},\mathbf{z},p)}}{\mu}$;
    $k \leftarrow k + 1$;
  end
  for $i = (t-1)\frac{m}{q} + 1, \ldots, t\frac{m}{q}$, $n_i \in \{n \in N_{\Omega_p} \mid n.p \neq$ *"Dirichlet"*$\}$ **do**
    $n_i.x \leftarrow x_j : (n_i, j) \in \mathrm{CRS}(\mathbf{A}).\mathrm{lut}$;
  end
end proc

der massiv parallelen FEM-Simulationsumgebung *padfem*^{2} (s. a. [BKM03, BM03]) prototypisch in verschiedenen Varianten (speicher- und thread-optimiert) implementiert. Als Analysekriterien zur Bewertung dienten u.a. die duchschnittliche Rechengeschwindigkeit in MFLOP/s, der Zeitaufwand für einen Simulationszeitschritt, d.h. das Lösen eines kompletten Gleichungssystems sowie der Zeitaufwand für eine einzelne Iteration, den lokalen Rechenaufwand und den Kommunikationsaufwand. Dabei wurden die Löser mit unterschiedlichen Problemgrößen, d.h. durch Adaption verfeinerte Gitter, konfrontiert.

3.2.2 Abgleicher

Unter einem (Rand-)Abgleicher versteht man ein Verfahren bzw. einen Algorithmus, der dafür sorgt, dass im parallelen Fall auf Partitionsrandelementen lokal ausgeführte Berechnungsschritte global einheitlich und eindeutig sind. Ein solches Vorgehen ist u.a. beim Aufstellen der Matrix im verteilten Fall notwendig. Aus Geschwindigkeitsgründen werden die Matrixeinträge auf den Partitionen lokal berechnet[20] anstatt die Matrix von einem Prozessor sequenziell assemblieren zu lassen. Da im verteilten Fall durch z.B. unterschiedliche Reihenfolgen von Fließkommaoperationen bei der Bearbeitung von Netzelementen auf Partitionsrändern kleine numerische Ungenauigkeiten (Auslöschung, Größenordnung, Unterlauf, etc) entstehen können, ist die Matrixzusammensetzung aus globaler Sicht nicht eindeutig. Obwohl die numerischen Ungenauigkeiten normalerweise im Bereich nahe der Maschinengenauigkeit (zwischen 10^{-16} bis 10^{-18}) liegen, reichen diese in der Praxis schon aus, um die Konvergenzeigenschaften eines iterativen Gleichungslösers zu stören.

Die Aufgabe eines Abgleichalgorithmus ist es, alle notwendigen Informationen für ein Randelement einer Partition, welches möglicherweise in zwei oder mehr Partitionen geometrisch vorhanden ist, zusammenzutragen, durch eine geeignete (mathematische) Operation global eindeutig zu machen, und eventuell das Ergebnis dieser Operation wieder an die Nachbarpartitionen zu propagieren. Als "Verknüpfungsoperation" kann z.B. die Mittelwert-, Maximum- oder Minimumbildung dienen. Eine andere Möglichkeit ist es durch eine zwingende Addition die verletzende Eigenschaft (z.B. Quasi-Diagonaldominanz) wieder herzustellen.

Im verteilten Modell der partitionslokalen Namensräume für die Netzobjekte muss eine geeignete Abbildung zwischen den Partitionsrandobjekten umgesetzt werden. Mit Hilfe des Datenaustausches aus Algorithmus 3.2 und den lokalen $\omega_{\mathbb{K}_i}$-Abbildungen aus Definition 3.11 kann ein Abgleichalgorithmus formuliert werden. Es wird dabei angenommen, dass das abzugleichende Datum eines Partitionsrandobjektes $k \in \partial P_p(\mathbb{K}_p)$ mit $\mathbb{K}_p \in \{N^*_{\Omega_p}, E^*_{\Omega_p}, F^*_{\Omega_p}\}$ in einem anhängigen Objekt $k.d \in \mathbb{R}$ gespeichert sei.

Algorithmus 3.11 zeigt eine generische Version eines Abgleichalgorithmus. Als Eingabeparameter bekommt dieser Algorithmus zusätzlich zu den bekannten Parametern noch

[20] Alle notwendigen Informationen sind am Partitionsrand durch den Halo gegeben.

Algorithmus 3.11 : Generischer Partitionsrand-Abgleicher

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
Partitionsrand $\partial P_p(\mathbb{K}_p)$ mit Objektmenge $\mathbb{K}_p \in \{N^*_{\Omega_p}, E^*_{\Omega_p}, F^*_{\Omega_p}\}$,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega), \mathcal{P}_\Omega \in \mathcal{M}_\Omega(n)$

Ausgabe : aktualisierte Daten am Partitionsrand $\partial P_p(\mathbb{K}_p)$

```
1  parallel proc BorderAdjustment begin
2      foreach p' ∈ neighbor*(P_p) do
3          for i = 1,...,|∂P_{p,p'}(K_p)|, k_i ∈ ∂P_{p,p'}(K_p) do
4              s_i^{p'} ← (ω_p(k_i), k_i.d);
5          end
6      end
7      Datenaustausch mit allen Nachbarn P_{p'}: {r^{p'}} ← Exchange ({s^{p'}}, C_p);
8      for i = 1,...,|∂P_p(K_p)|, k_i ∈ ∂P_p(K_p) do
9          d_i ← {d | (q, d) ∈ r^{p'} ∧ ω_p(k_i, p') = q};
10         u_i ← op_1(d_i, P_p);
11     end
12     foreach p' ∈ neighbor*(P_p) do
13         for i = 1,...,|∂P_{p,p'}(K_p)|, k_i ∈ ∂P_{p,p'}(K_p) do
14             s_i^{p'} ← (ω_p(k_i), u_i);
15         end
16     end
17     Datenaustausch mit allen Nachbarn P_{p'}: {r^{p'}} ← Exchange ({s^{p'}}, C_p);
18     for i = 1,...,|∂P_p(K_p)|, k_i ∈ ∂P_p(K_p) do
19         d_i ← {d | (q, d) ∈ r^{p'} ∧ ω_p(k_i, p') = q};
20         k_i.d ← op_2(d_i, P_p);
21     end
22 end proc
```

die Objektmenge über dem gesamten Rand $\partial P_p(\mathbb{K}_p)$ mit $\mathbb{K}_p \in \{N^*_{\Omega_p}, E^*_{\Omega_p}, F^*_{\Omega_p}\}$ (vgl. hierzu die Partitionsranddefinition). Der Abgleichalgorithmus ist in mehrere Phasen unterteilt. Die Initialisierungsphase (Zeile 2-6) setzt für alle Nachbarpartitionen $P_{p'}$ die Sendevektoren $s^{p'}$ über alle gemeinsamen Randobjekte $k \in \partial P_{p,p'}$ auf, wobei anhängige Daten $k.d$ gemeinsam mit dem lokalen Identifizierungsmerkmal $\omega_{\mathbb{K}_p}(k)$ als Tupel im Vektor gespeichert werden.

Nach dem Datenaustausch in Zeile 7 besitzt jede Partition alle notwendigen Informationen zu einem Randobjekt, das möglicherweise in mehr als einer Nachbarpartition geometrisch vorhanden ist. In einer Berechnungsphase (Zeile 8-11) werden alle diese Informationen zu einem Randobjekt aus dem Empfangsvektor extrahiert und durch eine (mathematische) Operation $op_1()$, die u.U. ebenfalls nur auf partitionslokal vorhande-

ne Informationen zugreift, verknüpft. Das Ergebnis dieser Verknüpfungsoperation wird wieder an die Nachbarpartitionen versendet (Zeile 12-17). Aus den nun empfangenen Datenvektoren wird ein aktualisiertes, neues Datum $k.d$ für ein Randobjekt durch eine weitere Verknüpfungsoperation $op_2()$ berechnet (Zeile 18-21).

Die Aufwandsklasse von Algorithmus 3.11 wird einerseits bestimmt durch die Größe des gesamten Partitionsrandes $|\partial P_p(\mathbb{K}_p)|$, wobei die Randelemente in zwei expliziten (Zeile 8 und 18) und zwei impliziten (Zeile 2 und 12) Schleifen bearbeitet werden. Andererseits werden im generischen Abgleichalgorithmus zwei Kommunikationsoperationen ausgeführt, was ebenfalls in die Laufzeit eingeht. Der Extraktionsaufwand (Zeile 9 und 19) sowie der Aufwand für die Verknüpfungsoperationen (Zeile 10 und 20) können als konstant angesehen werden, da die Menge der Daten pro Randelement, d.h. die Anzahl der Nachbarrandobjekte, durch die Anzahl der Nachbarpartitionen nach oben begrenzt ist ($\leq P$). Somit ergibt sich als Aufwandsklasse $\mathcal{O}(4 \cdot (|\partial P_p(\mathbb{K}_p)| + c_1) + 2 \cdot c_{comm}(C_p) + c_2)$. Auch hier gilt wieder für die Praxis, dass die Laufzeit von Algorithmus 3.11 durch die Kosten für die Kommunikation dominiert wird, also der Aufwand letztendlich in $\mathcal{O}(c_{comm}(C_p))$ liegt. Im Algorithmus werden zusätzlich zu den Sende- und Empfangsvektoren noch ein Speichervektor für die Verknüpfungsoperation (Vektor $u \in \mathbb{R}^m$ mit $m = |\partial P_p(\mathbb{K}_p)|$) und ein Vektor für die Zwischenspeicherung der Randobjektdaten benötigt.

Besitzen alle Partitionen $P_{p'} \in \text{neighbor}(P_p)$, in denen ein Randobjekt $k \in \mathbb{K}_p$ gemeinsam vorhanden ist, schon nach dem ersten Datenaustausch alle notwendigen Daten, um $k.d$ *eindeutig* bestimmen zu können, so kann die zweite Austauschphase entfallen. Ein Beispiel für diese Art des Abgleichens ist in Algorithmus 3.12 dargestellt. Das Verfahren basiert hier auf der Zuweisung des Datums aus der Partition, die den kleinsten Index besitzt. Neben der Reduzierung der Gesamtlaufzeit, wird bei diesem Algorithmus auch Speicherplatz gespart, da die Vektoren d und u nicht benötigt werden. Die Aufwandklasse bleibt dennoch $\mathcal{O}(c_{comm}(C_p))$.

Es sei noch erwähnt, dass die globale Matrix, die aus den im parallelen Fall abgeglichenen Daten der Randobjekte berechnet wird, *nicht* mit der Matrix übereinstimmt, die im sequenziellen Fall aufgestellt wird. Diese Gleichheit kann nur dadurch sichergestellt werden, dass die gesamte Konstruktion der Matrix auch im parallelen Fall nur von einem Prozessor durchgeführt wird. Sowohl aus Geschwindigkeitsgründen[21] als auch aus komplexen Implementierungsgründen ist in der Praxis das parallele Aufsetzen aus den nur lokal vorhandenen Partitionsinformationen sinnvoller. Die numerischen Ungenauigkeiten und die mathematischen Probleme, die sich daraus möglicherweise ergeben können, werden bewusst in Kauf genommen.

[21] Die Assemblierung des Matrixsystems aus Diskretisierung und Problemstellung kann u.U. sehr zeitaufwändig sein.

Algorithmus 3.12 : Partitionsrand-Abgleicher bzgl. kleinstem Partitionsindex

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
Partitionsrand $\partial P_p(\mathbb{K}_p)$ mit Objektmenge $\mathbb{K}_p \in \{N^*_{\Omega_p}, E^*_{\Omega_p}, F^*_{\Omega_p}\}$,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega), \mathcal{P}_\Omega \in \mathcal{M}_\Omega(n)$

Ausgabe : aktualisierte Daten am Partitionsrand $\partial P_p(\mathbb{K}_p)$

```
1  parallel proc BorderAdjustmentMinPart begin
2      foreach p' ∈ neighbor*(P_p) do
3          for i = 1,...,|∂P_{p,p'}(K_p)|, k_i ∈ ∂P_{p,p'}(K_p) do
4              s_i^{p'} ← (ω_p(k_i), k_i.d);
5          end
6      end
7      Datenaustausch mit allen Nachbarn P_{p'}: {r^{p'}} ← Exchange ({s^{p'}}, C_p);
8      for i = 1,...,|∂P_p(K_p)|, k_i ∈ ∂P_p(K_p) do
9          k_i.d ← d : (q,d) ∈ r^{p'} ∧ ω_p(k_i,p') = q ∧ p' = min_{j∈neighbor(P_p)} j;
10     end
11 end proc
```

3.3 Irreguläre 3D-Adaption auf Tetraedernetzen

Die Anwendung von geometrischen Adaptionsverfahren auf diskretisierte Gebiete hat einen großen Einfluss auf die Objektverwaltung der Simulation. Durch das Hinzufügen und Entfernen von oftmals sehr vielen Netzelementen entsteht ein großer Verwaltungsaufwand. In diesem Unterkapitel wird eine Realisierung der geometrischen, *irregulären Adaption auf der Basis der Rot-Grün Verfeinerung* (vgl. hierzu u.a. [Bey95, Car97, Bey98, Löh01]) für das verteilte Objektmodell entwickelt. Der Schwerpunkt liegt hierbei auf der Unterstützung für den massiv parallelen Fall. Um dieses Ziel zu erreichen, legt das Verfahren besonderen Wert auf die Reduktion der notwendigen Datenmengen beim Austausch zwischen Partitionen, wenn eine Adaptionsfront Partitionsgrenzen überschreitet. Für das in dieser Arbeit entwickelte *Zwei-Schritt Verfahren* reicht es dabei vollkommen aus, nur Kanteninformationen auszutauschen, um eine konsistente Datenhaltung des Tetraedergitters zu erhalten.

Das irreguläre Adaptionsverfahren beruht auf einem Satz von Verfeinerungsregeln, die eine ausgewählte Menge von Tetraedern des diskretisierten Simulationsgebietes modifizieren und dabei ein konsistentes und nicht-degeneriertes Netz vor und nach jedem Adaptionsschritt sicherstellen. Die Auswahl der Tetraeder erfolgt durch Anwendung eines oder einer Kombination mehrerer Fehlerschätzer bzw. Fehlerindikatoren (vgl. z.B. [RB03, BD03, Aki05]). Auf allen Tetraedern, die durch diese Auswahl bestimmt (markiert) wurden, wird die sog. *rote Verfeinerungsregel*, auch *reguläre Verfeinerungsregel* genannt, angewendet. Diese Regel fügt geometrisch auf jeder der sechs Kanten des Tetraeders einen neuen Knoten *mittig* ein und ersetzt die Kante durch zwei neue. Damit

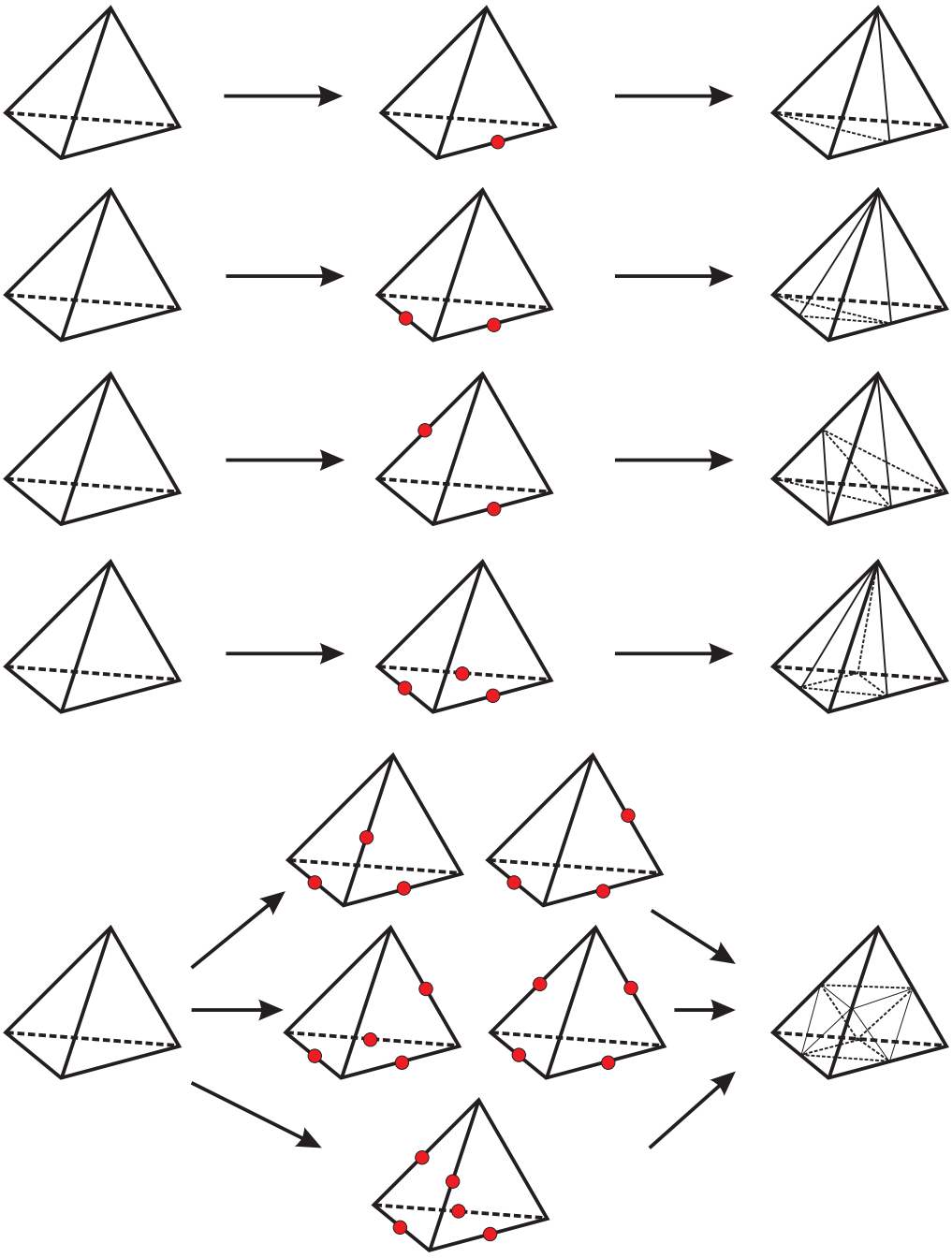
erhöht sich nun die Anzahl der Freiheitsgrade, d.h. die Anzahl der Unbekannten, an der geometrischen Position des Tetraeders bzw. der verfeinerten Kante. Das urspüngliche Tetraeder wird, da es nun kein Teil einer gültigen Tetraedierung nach Definition 2.23 ist, durch acht Teiltetraeder ersetzt, von denen vier zum Ausgangstetraeder kongruent sind und die anderen vier Tetraeder einer oder zwei anderen Kongruenzklassen angehören (s. a. [Zha95, Bey00]). Die acht Teiltetraeder bilden nun für sich betrachtet eine gültige Tetraedierung des ursprünglichen Tetraeders (s. Abbildung 2.4, Abschluss (a)). Da aber nun neue Knoten an den Kanten des Ausgangstetraeders eingefügt wurden, gilt für alle Tetraeder, die an den Kanten anhängen, dass diese Teil *keiner* gültigen Tetraedierung sind, weil sie hängende Netzelemente besitzen. Diese Netzelemente sind die in das Netz neu eingefügten Knoten und Kanten. Dazu gehören auch die vier neu erzeugten Teildreiecke auf den jeweils vier urspünglichen Dreicksfläche des Ausgangstetraeder.

Um die Bereiche, in denen die Tetraedierung hängende Netzobjekte besitzt, aufzulösen und in ein konsistentes Netz zu überführen, dienen die *grünen Abschlussregeln*. In der Abschlussphase werden alle Tetraeder betrachtet, die mindestens eine verfeinerte Kante mit hängenden Objekten besitzen. Da ein Tetraeder sechs Kanten besitzt, können so, wenn die Fälle "keine Kante verfeinert" und "alle Kanten verfeinert" ausgenommen werden, $2^6 - 2 = 62$ Kombinationen als Abschlusstetraeder gebildet werden. Aus Symmetriegründen kann diese Anzahl auf 9 Fälle reduziert werden (1 Kante: 1 Fall, 2 Kanten: 2 Fälle, 3 Kanten: 3 Fälle, 4 Kanten: 2 Fälle, 5 Kanten: 1 Fall). In der Praxis wird die Anzahl aufgrund von Qualitätseigenschaften der Abschlusstetraeder[22] (vgl. hierzu Abschnitt 2.2.2) für gewöhnlich weiter auf nur vier Möglichkeiten reduziert. Tritt ein Fall der verfeinerten Kantenkombinationen auf, der nicht durch die vier grünen Abschlussregeln abgedeckt wird, so wird für das betrachtete Abschlusstetraeder die rote Verfeinerungsregel angewandt, d.h. auf allen Kanten werden neue Knoten erzeugt. In Abbildung 3.4 sind die möglichen Tetraeder bei Anwendung der Rot-Grün Verfeinerungsstrategie dargestellt. Auf der linken Seite stehen die Ausgangstetraeder (vor Anwendung der roten Verfeinerung). In der Mitte sind die möglichen Kombinationen der Kantenverfeinerung dargestellt. Die rechte Seite zeigt schließlich den reduzierten, in der Praxis angewendeten Regelsatz.

Da im reduzierten Regelsatz fünf mögliche Kantenkombinationen im Tetraeder durch eine einzige Regel (Abbildung 3.4, unten) abgedeckt werden, entstehen zwangsläufig neue verfeinerte Kanten im betrachteten Tetraeder, die eigentlich nicht notwendig wären. Dadurch kann eine Welle von neuen Tetraedern mit hängenden Objekten durch das Tetraedernetz transportiert werden, die im Folgenden als *Dominoeffektwelle* bzw. als *Dominoeffekt* bezeichnet werden soll. Die Ausbreitung dieses Effektes lässt sich leider nicht vorherbestimmen, da sie im hohen Maße von der Struktur des Netzes abhängt

[22] Abschlusstetraeder besitzen, wenn sie nicht kongruent zum Ausgangstetraeder sind, viel kleinere Raumwinkel, die z.B. durch Halbierung oder Viertelung entstehen. Diese kleineren Winkel beeinflußen maßgeblich die Konditionszahl der zur Gebietsdiskretisierung assoziierten Matrix und sollten daher vermieden werden.

Abbildung 3.4 Reduzierter Regelsatz der Rot-Grün Verfeinerungsstrategie



und letztendlich sowohl durch die initiale Diskretisierung als auch von der Dynamik der instationären Problemstellung (Fehlerschätzer bzw. Fehlerindikator) vorgegeben wird. Die Auflösung dieses Effektes im parallelen Fall, bei dem von mehreren Partitionen (gleichzeitig) solche Wellen ausgehen und sich in angrenzende Partitionen ausbreiten können, ist ein weiterer Schwerpunkt im Design des Verfeinerungsalgorithmus.

Die geometrische Adaption besteht aus einer Kombination zweier Methoden, der Verfeinerung und der Vergröberung von Netzbereichen. Ein Fehlerschätzer bzw. ein Fehlerindikator bestimmt mit Hilfe von Informationen aus der Netzgeometrie und den numerisch berechneten Daten, wo in einem diskretisierten Simulationsgebiet welche Methode angewendet wird (s. hierzu z.B. [RB03, BD03, Aki05]). Dabei sollte im Idealfall eine stufenlose Verfeinerung bzw. Vergröberung in der Diskretisierung möglich sein. Eine Vergröberung über Teilgebiete der initialen Diskretisierung hinaus ist manchmal wünschenswert, jedoch nur mit hohem Aufwand bspw. durch lokale oder globale Neuvernetzungsmethoden (s. z.B. [Löh90, Löh01]) möglich. Vergröberungen in der Diskretisierung lassen sich mittels Netzhierarchien einfach realisieren, benötigen dafür aber einen Mehraufwand an Verwaltung und Speicherplatz pro Partition. Um diesen Mehraufwand durch die Hierarchien einzusparen, erlaubt der in dieser Arbeit entwickelte Adaptionsalgorithmus nur eine einstufige Rücknahme von Verfeinerungen.

3.3.1 Der Zwei-Schritt Algorithmus zur parallelen Adaption

Die Kernziele beim Design des parallelen Adaptionsalgorithmus sind:

- die Fähigkeit, bei Anwendung im massiv parallelen Fall gut mit der Anzahl der Partitionen zu skalieren,
- ein minimales Datenaufkommen bei der Kommunikation zu benötigen und
- die Konsistenz über Partitionsgrenzen hinweg wahren zu können.

Als Grundidee wird hierbei die Maximierung der Anzahl von partitionslokal durchzuführenden Aufgaben für die Adaption heranzgezogen. Transformationen von Tetraedern durch Anwendung des Adaptionsregelsatzes werden lokal ohne Kommunikationsaufwand durchgeführt. Kommunikation erfolgt nur in definierten Phasen zur Konsistenzerhaltung. Dazu reicht es, Informationen über die Verfeinerung von *Partitionsrandkanten* auszutauschen. Dieser Austausch der Kanteninformationen geschieht auf der Basis der Klassifizierung der Kanten, die bei der Adaptionsregel für die reguläre Verfeinerung eines Tetraeders entstehen. Die Konsistenz der Namensräume während des Adaptionsvorganges bleibt erhalten, indem der Austausch in zwei Kommunikationsschritten erfolgt, um die Klassifizierungen der Randkanten separat behandeln zu können. Die Anzahl der Kommunikationsschritte erklärt somit die Namensgebung des Algorithmus.

Die Idee des Zwei-Schritt Adaptionsalgorithmus

Der Zwei-Schritt Algorithmus ist in fünf Phasen eingeteilt:

1. Initialisierung,

2. lokale Adaption der markierten Tetraeder,
3. parallele Kantenpropagation mit regulärer Auflösung,
4. lokale Abschluss-Detektion,
5. Finalisierung.

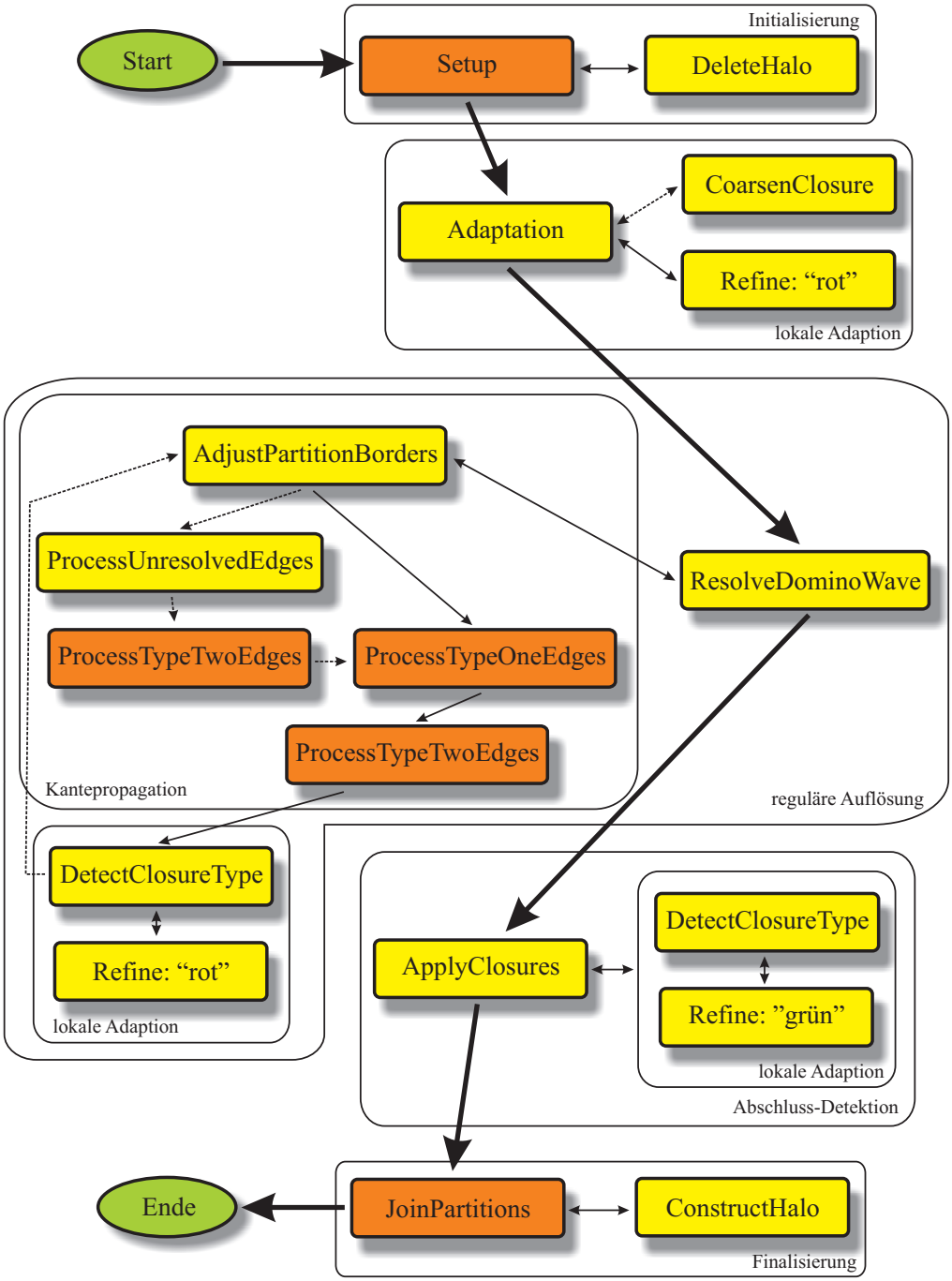
Auf die Arbeitsweise der einzelnen Phase wird im Folgenden kurz eingegangen, um eine Überblick über die Algorithmenstruktur des parallelen Adaptionsverfahrens zu erhalten. Eine wesentlich detailierte Ausführung der Funktionsweise der einzelnen Phasen bzw. der Funktionen sowie wichtiger Datenstrukturen, die in den Phasen benötigt werden, erfolgt in den anschließenden Kapitelabschnitten. Einen grafischen Überblick über die Phasen des Algorithmus zeigt Abbildung 3.5. Funktionen, in denen eine oder mehrere globale bzw. Punkt-zu-Punkt Kommunikationen stattfinden, sind durch orangefarbene Kästen hervorgehoben. Streng lokal arbeitende Funktionen des Algorithmus stehen in gelben Kästen.

Initialisierung Sie dient der Initialisierung von Datenstrukturen für den Algorithmus. In dieser Phase wird die Struktur der lokalen Partitionsränder mit jeder Nachbarpartition analysiert und die daraus gewonnenen Informationen entsprechend für die nachfolgenden Phasen in speziellen Datenstrukturen zwischengespeichert. Es wird zudem geprüft, ob überhaupt Tetraeder im *gesamten* verteilten Netz zur Adaption ausgewählt bzw. markiert wurden. Falls keine Tetraeder dafür vorgesehen wurden, sind die nachfolgenden aufwändigen Rechenschritte in den einzelnen Phasen nicht notwendig, und der Algorithmus kann sofort enden. Außerdem erfolgt in der Initialisierungsphase der komplette Abbau des Halosystems, da es für die nachfolgenden Phasen des Adaptionsalgorithmus nicht benötigt wird und nur zusätzlichen Zeit- und Speicheraufwand kosten würde, ihn im verteilten Netz konsistent zu halten. Es reicht für den Algorithmus aus, wenn der logische Zusammenhang der Partitionen untereinander durch die Partitionsränder beschrieben wird.

Lokale Adaption Diese Phase bestimmt über die Informationen, die der Fehlerschätzer bzw. -indikator liefert, eine Teilmenge der Tetraeder innerhalb einer jeden Partition, auf denen die *rote* bzw. *reguläre* Adaptionsregel (vgl. Abbildung 2.4 (a)) angewendet wird. Sowohl die Bestimmung der Tetraedermenge als auch die Anwendung der Adaptionsregel geschieht ausschließlich lokal in jeder Partition, eine Kommunikation ist hierfür nicht notwendig.

Gemäß den Regeln der *irregulären Adaption* erfolgt vor der eigentlichen Anwendung der roten Verfeinerungsregel eine einstufige Rücknahme von Abschlusstetraedern (Abbildung 2.4 (b)-(e)) zu groben Urspungstetraedern mit hängenden, verfeinerten Kanten. Diese Vorgehensweise wird zur Vermeidung von schlechten Raumwinkeln in Teiltetraedern von Abschlüssen durchgeführt. Eine Verkleinerung der Raumwinkel führt zu einer Verschlechterung der Kondition des Systems, die der Diskretisierung zugrunde

Abbildung 3.5 Ablaufdiagramm des parallelen Zwei-Schritt Adaptionsalgorithmus



liegt, und sollte somit vermieden werden. Es werden grundsätzlich alle Abschlusstetraeder in einer Partition vergröbert. Da zu diesem Zeitpunkt nicht entschieden werden kann, welche der Abschlusstetraeder möglicherweise durch den Dominoeffekt bei der nachfolgenden regulären Auflösung von einer Verfeinerung betroffen sind, ist diese

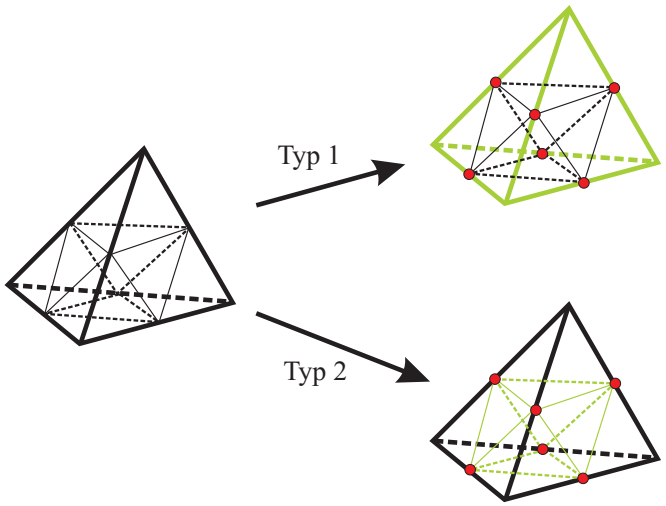
Vorgehensweise notwendig. Selbst wenn innerhalb einer Partition wenige oder sogar gar keine Abschlusstetraeder in reguläre Tetraeder umgeformt werden, kann lokal nicht entschieden werden, ob über eine Nachbarpartition eine Adaptionsfront in die Partition propagiert wird. Die Vergröberung in dieser Phase erfolgt streng lokal, d.h. Nachbarpartitionen brauchen nicht über die Vergröberung von Tetraedern, die Kanten auf den Partitionsrändern besitzen, informiert werden, da für die Auflösung des Dominoeffektes in der nachfolgenden Phase nur Informationen über Kantenverfeinerungen benötigt werden.

Kantenpropagation und reguläre Auflösung Sind alle Tetraeder, die durch den Fehlerschätzer bzw. -indikator markiert wurden, regulär verfeinert, beginnt die Phase der parallelen Kantenpropagation im verteilten Netz. Die Vergröberung von Abschlusstetraedern und die Anwendung der regulären Verfeinerung erzeugen in der Tetraedierung der einzelnen Partitionen neue hängende Kanten und Knoten (Erhöhung der Zahl von Unbekannten, degree of freedom (DOF)). Befindet sich nun eine hängende Kante mit ihrem neuen Mittelpunktknoten auf einem Partitionsrand, so müssen alle Partitionen, die diese Kante in der Geometrie gemeinsam haben, über die Verfeinerung informiert werden. Dies ist deshalb notwendig, weil möglicherweise das Tetraeder in der Nachbarpartition, welches diese Randkante als Konstruktionselement besitzt, nicht durch den Fehlerschätzer bzw. den Fehlerindikator markiert wurde. Der Dominoeffekt überschreitet also an dieser geometrischen Stelle die Partitionsgrenze.

Während hängende Kanten mit neuen Mittelpunktknoten durchaus Teil von mehr als zwei Partitionen sein können, entstehen durch die reguläre Verfeinerungsregel auch Kanten auf Dreicksflächen eines groben Tetraeders, deren Konstruktionsknoten neue hängende Mittelpunktsknoten sind, die nur von exakt zwei benachbarten Partitionen geteilt werden. Auch solche Kanten müssen zur Konsistenzwahrung des verteilten Netzes zwischen den beiden beteiligten Partitionen propagiert werden, da es vorkommen kann, dass so eine Kante im gleichen Adaptionsschritt wiederum verfeinert wird, d.h. in eine neue hängende Kante mit Mittelpunktknoten umgeformt werden muss. Wurde nun das Randtetraeder in der anderen Partition zur Adaption nicht markiert, so fehlt der Nachbarpartition zu diesem Zeitpunkt die Information, welche Kante der Dreiecksfläche verfeinert werden soll bzw. es wird mittels der Namensraumabbildung auf eine Kante zur Verfeinerung Bezug genommen, die in der Nachbarpartition noch nicht vorhanden ist. Die Folge wäre eine Inkonsistenz des verteilten Netzes an dieser geometrischen Stelle.

Um solche Inkonsistenzen zu vermeiden, wird für den parallelen Adaptionsalgorithmus eine Kantenklassifizierung eingeführt und eine getrennte Behandlung der beiden Kantenklassen in der Kommunikationphase bei der Kantenpropagation durchgeführt. Kanten, die durch Verfeinerung einen neuen Mittelpunktknoten bekommen, werden als *Kanten vom Typ 1*, neue Kanten zwischen erzeugten Mittelpunktknoten als *Kanten vom Typ 2* bezeichnet. Abbildung 3.6 zeigt die beiden Kantentypen (grün hervorgehoben), die ausschließlich bei Anwendung der regulären Adaptionsregel auf und an den Dreiecksflächen eines Urspungstetraeders entstehen können.

Abbildung 3.6 Typklassifizierung der Kanten bei der regulären Verfeinerung



Nachdem durch die Kantenpropagation jede Partition ihre eigenen lokal verfeinerten Randkanten an die Nachbarn gesendet und von diesen wiederum (möglicherweise neue) Informationen über verfeinerte Randkanten empfangen hat, befinden sich nun u.U. Tetraeder am Partitionsrand, die im Regelsatz für die irreguläre Adaption eine rote Verfeinerung erfordern. Dadurch entstehen neue hängende Kanten an Tetraedern, die im vorherigen, lokalen Adaptionsschritt nicht notwendig waren. Es entsteht eine neue Dominowelle, die ihren Usprung in der Nachbarpartition hat. Der nächste Schritt ist wieder eine lokale Auflösung durch die reguläre Verfeinerungsregel und einer anschließenden Kantenpropagation mit Informationsaustausch. Diese Vorgehensweise aus lokaler Adaption und paralleler Kantenpropagation wird iterativ fortgesetzt bis kein Tetraeder mehr im gesamten, verteilten Netz durch die rote Adaptionsregel verfeinert werden muss. Wenn dieser Punkt im Ablauf des Algorithmus erreicht wurde, können keine neuen hängenden Kanten mehr entstehen und der Algorithmus kann in die nächste Phase eintreten.

Abschluss-Detektion Da nun keine Dominoeffektwelle durch neu hinzukommende hängende Kanten erzeugt wird, kann in dieser Phase jedes Tetraeder, welches hängende Kanten und Mittelpunktknoten besitzt, durch ein Abschlusstetraeder aus dem Regelsatz der irregulären Adaption (vgl. Abbildung 2.4 (b)-(e)) aufgelöst bzw. geschlossen werden. Dies ist ein lokaler Vorgang und benötigt daher keine Kommunikation. Nach dieser Phase befindet sich die Partition in einem konsistenten Zustand ohne hängende Netzobjekte, d.h. die Partition ist lokal gesehen eine gültige Tetraedierung. Der Partitionsrand ist jedoch bzgl. der lokalen Namensraumabbildung noch nicht vollständig, da im Algorithmus nur Kanteninformationen am Partitionsrand ausgetauscht wurden. Dreiecksflächen und neu hinzugekommene Knoten auf dem Partitionsrand wurden bzgl. der partitionslokalen Namensraumabbildung noch nicht aktualisiert.

Finalisierung Um nun auch global die Konsistenz im verteilten Netz zu erhalten, folgt in der letzten Phase des Adaptionsalgorithmus die Finalisierung. Diese berechnet den logischen Zusammenhang der Partitionen untereinander neu, indem die lokalen Namensräume für die Partitionsrandobjekte mit den Nachbarn abgeglichen und dadurch neu bestimmt werden. Diese Phase benötigt daher eine Kommunikationsoperation für den Abgleich. Als letztes erfolgt noch der Wiederaufbau des Halosystems und eine Neuberechnung des Referenzmengensystems. Mit Abschluss dieser Phase befindet sich das verteilte Netz wieder in einem konsistenten Zustand gemäß Definition 3.12 und der Adaptionsalgorithmus kann enden.

Grundlegende Datenstrukturen und Operationen

Für den Zwei-Schritt Algorithmus zur parallelen Adaption werden spezielle Datenstrukturen und Operationen benötigt, um die verteilte Objektverwaltung während des Algorithmenablaufes realisieren zu können. Zur besseren Übersicht seien die Datenstrukturen in diesem und dem nachfolgenden Abschnitt als Mengen von Tupeln modelliert, die in den Funktionen des Adaptionsalgorithmus benutzt werden sollen. Diese Mengen sind in einem eigenen Namensbereich *ADS (Adaptionsdatenstrukturen)* zusammengefasst. Auf die einzelnen Mengen innerhalb des Namensbereiches wird in den Funktionen mit Hilfe des "."-Operators zugegriffen.

Der gesamte Adaptionsalgorithmus benötigt im Wesentlichen drei Kerndatenstrukturen bzw. Mengen. Zum einen wird eine Datenstruktur benötigt, die Informationen über grobe Ursprungstetraeder und die dazugehörenden Teiltetraeder für den Abschluss speichert. Informationen über hängende Netzobjekte in der Geometrie wie Knoten, Kanten und Dreiecksflächen, müssen in einer zweiten Datenstruktur verwaltet werden. Die dritte benötigte Datenstruktur speichert Informationen über die Zusammenhangsstruktur der Partitionsränder für die Namensräume der Netzobjekte.

Verwaltungsstruktur für Abschlüsse Mit $ADS.V_c$ sei die Menge bezeichnet, die obengenannte erste Datenstruktur modelliert. Sie enthält als Elemente 2-Tupel (V, N), wobei V wiederum eine Menge und N ein Tupel darstellen. V enthält die Teiltetraeder ν_i für die gilt, dass $\bigcup_i \nu_i = \nu$ und ν das Urspungstetraeder mit den hängenden Objekten ist, für dass das Abschlusstetraeder zur Auflösung bestimmt ist. $ADS.V_c$ besitzt für die Abschlussregeln "grün 1:2" zwei, "grün 1:3" drei und für "grün 1:4a" sowie "grün 1:4b" vier Elemente. Das Tupel N beinhaltet die Knoten des Ursprungstetraeders in einer bestimmten Reihenfolge[23]. Mittels dieser Knoten und der Reihenfolgenvorschrift kann im Vergröberungsteil des Adaptionsalgorithmus das Ursprungstetraeder leicht wiederhergestellt und in die Geometrie integriert werden, ohne dass ein passendes Matching für die hängenden Kanten berechnet werden muss. D.h. es ist keine Translation, Rotation bzw. Spiegelung der Konstruktionselemente der Teiltetraeder notwendig.

[23] Die Reihenfolgenvorschrift transformiert das Tetraeder der Diskretisierung in ein Standardtetraeder mit eigenem Koordinatensystem, in dem die Adaptionsregeln zur Modifizierung bzw. Auflösung angewendet werden.

Als reale Implementierung der Datenstruktur bzw. dieser Menge $ADS.V_c$ empfiehlt sich eine Hashtabelle, da die Hauptoperation bei der Vergröberungsphase das Bestimmen der Knoten des Ursprungstetraeders ist (Suchoperation), welche möglichst in konstanter Zeit erfolgen muss. Dazu können bspw. die Identifikationsmerkmale der Teiltetraeder aus der Menge V als Suchschlüssel in der Hashtabelle für das Tupel (V, N) gewählt werden. Ein Tupel wird also durch mehrere Schlüssel aus V identifiziert. Dieses Vorgehen der mehrfachen Referenzierung ist sehr nützlich, da beim Iterieren über die zu vergröbernden Abschlusstetraeder bzw. der Teiltetraeder keine Modifizierung der Bearbeitungsreihenfolge (welches Teiltetraeder des Abschlusstetraeders wurde schon erfasst bzw. bearbeitet?) erfolgen muss. Somit kann mit linearem Laufzeitaufwand die Bestimmung der Grobtetraeder in einer Partition erfolgen. Der Aufwand für Einfüge- und Löschoperationen ergibt sich gemäß der Hashtabellendatenstruktur ebenfalls approximativ zu $\mathcal{O}(1)$ (vgl. [OW90, CLR90]). $ADS.V_c$ ist die einzige Menge bzw. Datenstruktur, die ihre Gültigkeit über die Laufzeit des Adaptionsalgorithmus hinaus behält, da auch in der Lastverteilungsphase der parallelen Simulation Informationen über Abschlusstetraeder benötigt werden (s. hierzu Abschnitt 3.4).

Verwaltungsstruktur für hängende Netzobjekte Die zweite wichtige Kerndatenstruktur bzw. Menge ist $ADS.O_h$. Bei dieser Struktur handelt es sich genau genommen um drei einzelne Mengen $ADS.O_h^N$, $ADS.O_h^E$ und $ADS.O_h^F$, die aus Gründen der Übersicht in den nachfolgenden Funktionen häufig zum Synonym $ADS.O_h$ zusammengefasst werden. $ADS.O_h$ dient zur Speicherung von Informationen über die hängenden Objekte in der Tetraedierung des Gebietes zu jedem Zeitpunkt bzw. in jeder Phase des Adaptionsalgorithmus. Dabei bezeichnen die Superskripte N, E und F, welcher hängende Objekttyp (Knoten, Kante oder Dreicksfläche) in der Struktur mit den notwendigen Zusatzinformationen gespeichert wird. In $ADS.O_h^N$ sind Tupel der Form (v, N_{ref}) mit $N_{ref} = \{n_i\}$ enthalten. v sei dabei das eindeutige Tetraeder mit den hängenden Knoten n_i. Die Menge $ADS.O_h^E$ verwaltet Tupel der Form (v, E_{ref}), wobei $E_{ref} = \{(e_i, e_i^1, e_i^2)\}$ ist. Hierbei bezeichnet ebenfalls v das eindeutige Tetraeder, dessen Kanten e_i in die Teilkanten e_i^1 und e_i^2 durch Mittelpunktsunterteilung verfeinert wurden. Äquivalent gilt für $ADS.O_h^F$, dass Tupel der Form (v, F_{ref}) mit $F_{ref} = \{(f_i, f_i^1, f_i^2, f_i^3, f_i^4)\}$ für die hängenden Dreiecksflächen in der Menge gespeichert werden. Der Index i bewegt sich, je nach Objekttyp und der Anzahl der verfeinerten bzw. hängenden Objekte des Tetraeders v, zwischen eins und vier (Knoten), eins und sechs (Kanten) bzw. eins und vier (Flächen).

Auch für die Mengen aus $ADS.O_h$ bietet es sich an, in einer realen Implementierung jeweils eine Hashtabelle zu benutzen. Als Schlüssel dient hier das Identifikationsmerkmal des Tetraeders v mit seinen hängenden Objekten. Such-, Einfüge- und Löschoperationen liegen damit approximativ in der Aufwandsklasse $\mathcal{O}(1)$.

Verwaltungsstruktur für Zusammenhang der Partitionsränder Die letzte wichtige Datenstruktur ist für die Konsistenzwahrung der Partitionsränder notwendig. Sie wird im Folgenden mit der Menge $ADS.P^p$ idealisiert, wobei p die Nachbarpartition zur gerade betrachteten Partition angibt. Genau so wie $ADS.O_h$ stellt $ADS.P^p$ eine Zusammen-

fassung von drei Mengen über die Objekttypen des Partitionsrandes dar: $ADS.P_N^p$, $ADS.P_E^p$ und $ADS.P_F^p$. In diesen Mengen werden Tupel (o, k, t) verwaltet, bei denen o ein lokales Objekt auf dem Rand zur Partition p, k den partitionslokalen Namen des Objektes o in der Partition p und $t \in \{\text{“Start”}, \text{”Typ 1”}, \text{”Typ 2”}\}$ die Klassifizierung des Objektes darstellt. Da die Klassifizierung nur für Randkanten im Kantenpropagationsalgorithmus der Adaption benötigt wird, ist die Menge der möglichen Werte für t um einen Wert “Start” erweitert, um einerseits initiale Kanten, d.h. im Verlauf des Algorithmus nicht verfeinerte Kanten zu kennzeichnen, andererseits aber auch um Knoten und Dreiecksflächen im selben Tupelschema abbilden zu können.

Die Mengen $ADS.P_N^p$, $ADS.P_E^p$ und $ADS.P_F^p$ können ebenfalls über Hashtabellen für jeden Objekttyp und jede Nachbarpartition implementiert werden. Als Schlüssel für ein Tupel (o, k, t) dient dann das Identifikationsmerkmal des Objektes o. Durch die Hashtabellendatenstruktur kann in den Funktionen des Adaptionsalogrithmus der Aufwand für die Verwaltungsoperationen bei Bearbeitung der Partitionsränder damit approximativ in $\mathcal{O}(1)$ gehalten werden.

Nachdem die drei Kerndatenstrukturen für den parallelen Zwei-Schritt Adaptionsalgorithmus vorgestellt wurden, erfolgt nun eine Darstellung dreier Grundfunktionen. In den folgenden Algorithmen kennzeichnet $\mathcal{M}^*_{\Omega_p}$ eine modifizierte Tetraedierung einer Partition P_p, d.h. sie enthält mindestens ein hängendes Objekt (Knoten, Kante, Fläche), welches noch nicht aufgelöst wurde. Der Übersichtlichkeit halber sind in den Algorithmen an einigen Stellen statt eines komplexen Formalismus basierend auf dem Objektmodel (s. Abschnitt 3.1) nur umgangssprachliche Formulierungen angegeben. Dies betrifft insbesondere Aktualisierungsoperationen der oben eingeführten Kerndatenstrukturen.

Bestimmung des Abschlusstyps Der Adaptionsalgorithmus benötigt an mehreren Stellen im Ablauf eine Möglichkeit, ein Tetraeder, welches hängende Kanten besitzt, danach zu beurteilen, welcher Abschlusstyp (roter bzw. grüner Fall) für dieses gilt. Dazu wird hier eine Funktion DetectClosureType() vorgestellt, deren Struktur in Algorithmus 3.13 für das in dieser Arbeit entwickelte Objektmodell angegeben ist.

DetectClosureType() bestimmt in einem ersten Analyseschritt die Anzahl der verfeinerten Kanten eines Tetraeders v^*. Dazu wird das für v^* assozierte Datum E_{ref} aus $ADS.O_h^E$ ermittelt und die Kardinalität dieser Menge bestimmt. Sind mehr als drei Kanten in E_{ref} vorhanden oder ist mindestens eine der Kanten aus E_{ref} ebenfalls hängend[24], dann wird als Verfeinerungsstrategie die rote Regel ausgewählt (Zeile 3). Handelt es sich nur um eine Kante in E_{ref}, ist die Strategie “grün 1:2” (Zeile 5-6, vgl. Abbildung 2.4 Regel (c)). Bei zwei Kanten in E_{ref} (Zeile 7-13) muss eine Unterscheidung erfolgen, ob die beiden verfeinerten Kanten adjazent sind (Zeile 9), d.h. mindestens einen Konstruktionsknoten gemeinsam haben. Dies entscheidet, ob als Verfeinerungsstrategie die Regel “grün 1:3” oder “grün 1:4b” festgelegt wird (vgl. Abbildung 2.4 Regel (d),(e)). Sind

[24] Ein Teiltetraeder einer regulären Verfeinerung wurde durch den Dominoeffekt (bspw. aus einer Nachbarpartition) für die rote Verfeinerung vorgesehen.

Algorithmus 3.13 : Algorithmus zur Typbestimmung der Abschlusstetraeder

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
$v^* \in V_{\Omega_p}$ sei zu bestimmendes Tetrader,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : Abschlusstyp t des Tetraeders v^*

```
1  parallel proc DetectClosureType begin
2      bestimme Tupel (v, E_ref) ∈ ADS.O_h^E : v* = v;
3      if |E_ref| > 3 ∨ ∃(e, e1, e2) ∈ E_ref mit e1 oder e2 sind "hängende" Kanten then
4          t ← "rot";
5      else if |E_ref| = 1 then
6          t ← "grün 1:2";
7      else if |E_ref| = 2 then
8          sei E_ref = {(e, e1, e2), (e', e1', e2')};
9          if R_N(e) ∩ R_N(e') ≠ ∅ then
10             t ← "grün 1:3";
11         else
12             t ← "grün 1:4b";
13         end
14     else
15         sei E_ref = {(e, e1, e2), (e', e1', e2'), (e'', e1'', e2'')};
16         if cstr(e, e', e'') ∈ R_F(v) then
17             if cstr(e, e', e'') ∈ ∂P_p(F_Ωp) then
18                 t ← "rot";
19             else
20                 f ← cstr(R_N(e1) ∩ R_N(e2), R_N(e1') ∩ R_N(e2'), R_N(e1'') ∩ R_N(e2''));
21                 if R_E(f) hat "hängende" Kanten then
22                     t ← "rot";
23                 else
24                     t ← "grün 1:4a";
25                 end
26             end
27         end
28         t ← "rot";
29     end
30 end proc
```

exakt drei Kanten von v^* verfeinert (Zeile 15), wird die Regel "grün 1:4a" nur dann als Strategie ausgewählt (vgl. Abbildung 2.4 Regel (b)), wenn die drei Kanten eine Dreiecksfläche von v^* bilden (Zeile 20-25), ansonsten muss "rot" verfeinert werden. Handelt es sich bei der Dreiecksfläche um ein Partitionsrandobjekt, so wird als Strategie "rot"

ausgewählt (Zeile 17). Diese Situation kann nur dann auftreten, wenn eine Adaptionsfront aus der Nachbarpartition in die Partition hineinwandert, oder aber drei Tetraeder, die jeweils eine dieser Kanten als Konstruktionselement, aber keine gemeinsame Dreiecksfläche mit v^* besitzen, verfeinert werden. In beiden Fällen ist es sinnvoll, regulär zu verfeinern, da es sich um ein lokal begrenztes Gebiet handelt, in dem spätestens im nächsten Iterationsschritt bei der regulären Auflösung die Umgebung durch weitere Abschlusstetraeder modifiziert wird. Da die Entscheidungsstellen im Algorithmus, die bestimmen, welche Abschlussregel als Strategie ausgewählt werden soll, alle mit konstantem Aufwand berechnet werden können (Referenzmengen sind vorausberechnet), liegt der Gesamtaufwand für DetectClosureType() in $\mathcal{O}(1)$.

Tetraederverfeinerung Die zweite grundlegende Operation, die an verschiedenen Stellen im Adaptionsalgorithmus verwendet wird, ist die Verfeinerungsfunktion, die ein Tetraeder in seine Teiltetraeder zerlegt. Algorithmus 3.14 veranschaulicht die Funktion Refine(), die ein Tetraeder $v \in V_{\Omega_p}$ und eine Verfeinerungsstrategie s als Eingabeparameter annimmt. Als Rückgabewert liefert sie die Menge der neuen Tetraeder V^*, deren "volumetrische" Vereinigung v ergeben. Beim Verfeinern einer Kante entsteht ein neuer Mittelpunktsknoten, der im Algorithmus mit $n_{a,b}$ bezeichnet wird, wenn a und b die beiden Konstruktionsknoten einer Grobkante e sind ($\mathcal{R}_N(e) = \{a, b\}$).

Refine() bestimmt zuerst (Zeile 2) die Menge der Konstruktionsknoten von v. Danach folgt eine Tupelbildung durch Anwenden einer Permutation $\pi : \{0, \ldots, 3\} \to \{0, \ldots, 3\}$ auf die ermittelten Knoten. Durch diese Permutation wird das Tetraeder v in ein Standardtetraeder transformiert, auf dem dann die Verfeinerungsstrategie angewendet wird. Dies geschieht aus zwei Gründen. Zum einen muss die Permutation der Knoten im Falle einer roten Verfeinerung (Zeile 6-16) dafür Sorge tragen, dass bei fortgesetzter regulärer Verfeinerung der Teiltetraeder keine Degeneration der dabei neu entstehenden Tetraeder erfolgt. Dies kann durch eine vom Ursprungstetraeder geerbte und stets mitgeführte Aufzählungsreihenfolge der Konstruktionsknoten sichergestellt werden (vgl. hierzu [Zha95, Bey98]). Eine Beschränkung auf maximal drei Kongruenzklassen der Teiltetraeder bei iterativer Anwendung der roten Strategie ist damit möglich. Im Algorithmus 3.14 sei die Vererbung der Knotenreihenfolge durch die Funktion cstr() für Tetraeder gegeben. Der zweite Grund für die Permutation ist, das Tetraeder durch Drehung und Spiegelung aus der gegebenen Diskretisierung der Geometrie so in das Standardtetraeder zu transformieren, dass der Regelsatz für die grünen Abschlussstrategien ("1:2" bis "1:4b") nur für einen Fall der Knotennummerierung modelliert werden muss.

Wurde die Reihenfolge der Knotennummerierung mittels der Permutation π und der Strategie s festgelegt, erfolgt die Konstruktion der zwei bis acht Teiltetraeder v_i^* in Abhängigkeit von s, die zusammen die Menge V^* bilden. Um bei der Abschlussgenerierung keine mehrfachen, geometrisch gleichen Objekte zu erzeugen (durch die cstr()-Funktion), ist es notwendig, in den Mengen aus $ADS.O_h$ nach bereits hängenden Objekten für die Teiltetraeder zu suchen. Sind solche Objekte vorhanden, werden diese

Algorithmus 3.14 : Algorithmus zur Rot-Grün Verfeinerung

Eingabe : $p \in \{1,\dots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
$v \in V_{\Omega_p}$ sei zu verfeinerndes Tetraeder,
s sei die Verfeinerungsstrategie (rote oder eine der grünen Strategien),
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : Menge V^* von neuen Teiltetraedern v^*_i, die zusammen v bilden

```
1  parallel proc Refine begin
2      {n_0, n_1, n_2, n_3} ← R_N(v);
3      permutiere Tetraederknoten bzgl. Strategie s : (n_π(0), n_π(1), n_π(2), n_π(3)) =: N*;
4      ermittle aus ADS.O_h alle notw. Objekte zur Konstruktion bzgl. Strategie s;
5      switch s do
6          case "rot"
7              v*_1 ← cstr(n_π(0), n_π(0),π(1), n_π(0),π(2), n_π(0),π(3));
8              v*_2 ← cstr(n_π(0),π(1), n_π(1), n_π(1),π(2), n_π(1),π(3));
9              v*_3 ← cstr(n_π(0),π(2), n_π(1),π(2), n_π(2), n_π(2),π(3));
10             v*_4 ← cstr(n_π(0),π(3), n_π(1),π(3), n_π(2),π(3), n_π(3));
11             v*_5 ← cstr(n_π(0),π(1), n_π(0),π(2), n_π(0),π(3), n_π(1),π(3));
12             v*_6 ← cstr(n_π(0),π(1), n_π(0),π(2), n_π(1),π(2), n_π(1),π(3));
13             v*_7 ← cstr(n_π(0),π(2), n_π(0),π(3), n_π(1),π(3), n_π(2),π(3));
14             v*_8 ← cstr(n_π(0),π(2), n_π(1),π(2), n_π(1),π(3), n_π(2),π(3));
15             V* ← {v*_1, v*_2, v*_3, v*_4, v*_5, v*_6, v*_7, v*_8};
16         end
17         case "grün 1:2"
18             v*_1 ← cstr(n_π(0), n_π(0),π(1), n_π(2), n_π(3));
19             v*_2 ← cstr(n_π(0),π(1), n_π(1), n_π(2), n_π(3));
20             V* ← {v*_1, v*_2};
21         end
22         case "grün 1:3"
23             v*_1 ← cstr(n_π(0), n_π(0),π(1), n_π(0),π(2), n_π(3));
24             v*_2 ← cstr(n_π(2), n_π(0),π(2), n_π(0),π(1), n_π(3));
25             v*_3 ← cstr(n_π(1), n_π(2), n_π(0),π(1), n_π(3));
26             V* ← {v*_1, v*_2, v*_3};
27         end
28         ... (fortgesetzt)
29     end
30 end proc
```

verwendet und, falls es nötig ist, d.h. die Objekte wurden vollständig in der Tetraedie-

Algorithmus 3.14 : Algorithmus zur Rot-Grün Verfeinerung (Fortsetzung)

```
1  parallel proc Refine begin
2      switch s do
3          ...
4          case "grün 1:4a"
5              v1* ← cstr(n_π(0), n_π(0),π(1), n_π(0),π(2), n_π(3));
6              v2* ← cstr(n_π(1), n_π(1)π(2), n_π(0),π(1), n_π(3));
7              v3* ← cstr(n_π(2), n_π(0),π(2), n_π(1),π(2), n_π(3));
8              v4* ← cstr(n_π(0),π(1), n_π(0),π(2), n_π(1),π(2), n_π(3));
9              V* ← {v1*, v2*, v3*, v4*};
10         end
11         case "grün 1:4b"
12             v1* ← cstr(n_π(0), n_π(0),π(1), n_π(2),π(3), n_π(3));
13             v2* ← cstr(n_π(1), n_π(2)π(3), n_π(0),π(1), n_π(3));
14             v3* ← cstr(n_π(0), n_π(0),π(1), n_π(2), n_π(2),π(3));
15             v4* ← cstr(n_π(1), n_π(2), n_π(0),π(1), n_π(2),π(3));
16             V* ← {v1*, v2*, v3*, v4*};
17         end
18     end
19     if s ≠ "rot" then ADS.V_c ← ADS.V_c ∪ {(V*, N*)};
20     aktualisiere ADS.O_h über alle neuen "hängenden" Objekte aus V*;
21     füge jede neue hängende Kante e ∈ ∂P_p,p' als Tupel (e, ω_p(e, p'), t) mit Typ
       t ∈ {"Typ 1", "Typ 2"} zu ADS.P_E^p' hinzu;
22 end proc
```

rung aufgelöst[25], entfernt (Zeile 3-4, erster Teil).

Nach der geometrischen Erzeugung der Teiltetraeder müssen die Datenstrukturen für den Adaptionsprozess aktualisiert werden. Handelt es sich bei der Verfeinerungsstrategie s um einen grünen Fall (Zeile 19, zweite Teil), so bildet die Vereinigung der Tetraeder in V^* ein *echtes Abschlusstetraeder*. In diesem Fall muss die Menge $ADS.V_c$ um ein Tupel, bestehend aus V^* und den Konstruktionsknoten des Grobtetraeders N^* ergänzt werden. Diese Informationen werden einerseits für die Vergröberung des Abschlusses, andererseits aber auch für die nachfolgende Lastverteilungsphase bei der Migration benötigt. Bei der Migration von Tetraedern in andere Partitionen zum Zwecke des Lastausgleichs muss darauf geachtet werden, dass keine Teiltetraeder von Abschlüssen in verschiedene Partitionen transportiert werden. Eine Vergröberung wäre dann nicht

[25] Eine Überprüfung, ob eine vollständige Auflösung eines hängenden Objektes erfolgt ist, kann bspw. duch das Konzept des Referenzzählers ermöglicht werden. Beim Erzeugen eines hängenden Objektes wird der Referenzzähler mit der Anzahl der anhängigen Objekte (s. Referenzmengen) initialisiert. Jede Auflösung eines anhängigen Objektes erniedrigt den Zähler. Die letzte Auflösung entfernt dann das Tupel aus der Menge $ADS.O_h$

mehr möglich.

Im Falle einer roten, also einer regulären Verfeinerungsstrategie für s können neue hängende Objekte in der Tetraedierung entstehen. Alle neu entstandenen, hängenden Objekte müssen daher in den Mengen von $ADS.O_h$ (also $ADS.O_h^N$, $ADS.O_h^E$ sowie $ADS.O_h^F$) eingetragen werden. Diese Überprüfung und Aktualisierung der Mengen muss für alle Teiltetraeder aus V^* durchgeführt werden.

Liegt das zu verfeinernde Tetraeder ν mit mindestens einer Grobkante, die in eine hängende Kante transformiert wurde, auf einem Partitionsrand, so müssen die Nachbarpartitionen $P_{p'}$ von P_p, die diese Kante ebenfalls auf dem Partitionsrand haben, über diese Veränderung informiert werden. Dazu dient insbesondere die Datenstruktur bzw. Menge $ADS.P_E^{p'}$. Es wird daher für *jede betroffene* Grobkante $e \in \mathcal{R}_E(\nu)$ ein Tupel der Art $(e, \omega(e, p'), \text{“Typ 1”})$ in die Menge für die Nachbarpartitionen eingetragen. Entsprechendes gilt natürlich auch für Kanten zwischen neu entstandenen Mittelpunktknoten, also Kanten der Klassifizierung “Typ 2” (Zeile 21, erster Teil). Wurden alle Aktualisierungen der Datenstrukturen für die parallele Adaption durchgeführt, endet die Funktion Refine().

Tetraedervergröberung Das Gegenstück zur Funktion Refine() aus Algorithmus 3.14 stellt die Funktion CoarsenClosure() zur einstufigen Vergröberung dar, deren Ablaufstruktur in Algorithmus 3.15 aufgezeigt ist. Die Vergröberung findet vor der eigentlichen regulären Verfeinerung der vom Fehlerschätzer bzw. -indikator markierten Tetraeder statt. Sämtliche in der Partition vorhandenen Abschlusstetraeder sind von der Vergröberung betroffen, d.h. alle Tetraeder, die ein assoziiertes Tupel (V^*, N^*) in $ADS.V_c$ besitzen. Die Knoten im Tupel N^* bilden Konstruktionsknoten des groben Tetraeders ν, welche in die Diskretisierung eingefügt werden (Zeile 3-4). Hierbei ist zu beachten, dass bei der Vergröberung neue hängende Objekte entstehen können. Eine Aktualisierung von $ADS.O_h$ über diese neuen Objekte ist daher notwendig (Zeile 15).

Für den Fall, dass durch den Fehlerschätzer bzw. -indikator ein oder mehrere Teiltetraeder eines Abschlusses markiert wurden, muss gemäß dem Rot-Grün Verfeinerungsverfahren das vergröberte Tetraeder ν in die Menge der markierten Tetraeder V_{FS} mit aufgenommen werden (Zeile 5-8). Dadurch wird garantiert, dass keine Raumwinkelverkleinerung, d.h. keine Verschlechterung der Kondition des Systems (aus der Diskretisierung) entstehen kann (vgl. z.B. [BA76, Kri92]) und trotzdem eine Erhöhung der Zahl der Unbekannten an der geometrischen Position des Abschlusses erfolgt.

Nach der Erzeugung des Grobtetraeders werden die Teiltetraeder $\nu_i^* \in V^*$ aus der Diskretisierung entfernt (Zeile 10). Wie bei der Verfeinerung in Algorithmus 3.14 muss folgendes beachtet werden: wenn Kanten der Teiltetraeder des Abschlusses auf Partitionsrändern liegen, muss die Adaptionsdatenstruktur $ADS.P_E^{p'}$ entsprechend den Nachbarpartitionen $P_{p'}$ modifiziert werden. In der Funktion CoarsenClosure() werden daher die assoziierten Tupeleinträge für die Teiltetraederkanten aus $ADS.P_E^{p'}$ entfernt (Zeile 11-13). Die Grobkante muss nicht in $ADS.P_E^{p'}$ eingetragen werden, da in den Nachbarpartitionen $P_{p'}$ die Verfeinerung schon existiert und in P_p das Grobtetraeder ν in

Algorithmus 3.15 : Algorithmus zur Rot-Grün Vergröberung

Eingabe : $p \in \{1, \ldots, P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
V_{FS} sei Menge der zu verfeinernden Tetraeder,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : modifizierter, partitionslokaler Netzanteil $\mathcal{M}^*_{\Omega_p}(P)$

```
1  parallel proc CoarsenClosures begin
2      foreach (V*, (n_0, n_1, n_2, n_3)) ∈ ADS.V_c do
3          v ← cstr(n_0, n_1, n_2, n_3);
4          M*_{Ω_p} ← M*_{Ω_p} ⊕ v;
5          if ∃v* ∈ V* : v* ∈ V_FS then
6              V_FS ← V_FS \ V*;
7              V_FS ← V_FS ∪ {v};
8          end
9          foreach v* ∈ V* do
10             M*_{Ω_p} ← M*_{Ω_p} ⊖ v*;
11             foreach e' ∈ R_E(v) : e' ∈ ∂P_{p,p'}(E_{Ω_p}) ∧ p' ∈ neighbor*(P_p) do
12                 ADS.P_E^{p'} ← ADS.P_E^{p'} \ {(e, k, t) ∈ ADS.P_E^{p'} | e = e' ∧ k = ω_p(e, p') ∧
                       t ∈ {"Start", "Typ 1", "Typ 2"}};
13             end
14         end
15         aktualisiere ADS.O_h über alle neuen "hängenden" Objekte von v;
16     end
17     bestimme Referenzsystem R_{Ω_p} neu;
18 end proc
```

der nachfolgenden Phase im Adaptionsalgorithmus wieder verfeinert wird (regulär oder wieder als Abschluss). Die Konsistenz der Namensräume ist daher auf allen Partitionsseiten gegeben und ein Informationsaustausch ist zu diesem oder einem nachfolgenden Zeitpunkt nicht notwendig.

Der Zwei-Schritt Adaptionsalgorithmus im Detail

Nachdem die Grundidee sowie die Kerndatenstrukturen und -operationen des parallelen Zwei-Schritt Adaptionsalgorithmus aufgezeigt wurden, folgt nun eine detailiertere Vorstellung des Verfahrens und der Ablaufstruktur, um das Zusammenspiel und die Auswirkungen auf andere Teile der Simulation aufzuzeigen. Im Folgenden werden die Algorithmen der wesentlichen Funktionen beschrieben, die in Abbildung 3.5 zur Verdeutlichung des Adaptionsalgorithmus benutzt wurden. Dabei wird eine Top-Down

Algorithmus 3.16 : Hauptalgorithmus zur parallelen Adaption

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
$V_{FS} \subseteq V_{\Omega_p}$ seien die durch den Fehlerschätzer markierten Tetraeder,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega), \mathcal{P}_\Omega \in \mathcal{M}_\Omega(P)$

Ausgabe : aktualisierter, partitionslokaler Netzanteil $\mathcal{M}_{\Omega_p}(P)$

```
1  parallel proc AdaptationMain begin
2      s_{1≤i≤P} ← 0; s_p ← |V_FS|;
3      Datenaustausch mit allen Nachbarn: s_{1≤i≤P} ← Exchange(s_p, C_p);
4      if Σ_{i=1}^{P} s_i ≠ 0 then
5          Initialisierung der Adaptionsdatenstrukturen ADS für alle Ränder ∂P_{p,p'}
               über alle Nachbarn P_{p'} ∈ neighbor(P_p);
6          Abbau des Halosystems H_p durch DestructHalo(P_p);
7          M_{Ω_p} ← Adaptation(M_{Ω_p}, V_FS, C_p, ADS);
8          Aufbau des Halosystems H_p durch ConstructHalo(P_p, C_p);
9      end
10 end proc
```

Sicht der gesamten Algorithmenstruktur verfolgt.

Startfunktion der Adaption Algorithmus 3.16 stellt die Hauptfunktion des Zwei-Schritt Verfahrens dar. Es wird für den Algorithmus vorausgesetzt, dass ein (globaler) Fehlerschätzer bzw. -indikator eine Teilmenge der Tetraeder des gesamten Netzes zur Verfeinerung ausgewählt hat. Jede Partition besitzt somit eine (möglicherweise leere) Menge V_{FS}, in der der lokale Anteil der markierten Tetraeder eingetragen ist. Eine Adaption ist nur dann sinnvoll, wenn mindestens ein Tetraeder markiert wurde. Daher prüft die Funktion AdaptationMain vor dem eigentlichen Adaptionsprozess (Zeile 2-4), ob überhaupt die nachfolgenden Phasen des Algorithmus durchgeführt werden müssen. Dies geschieht durch eine Aufsummierung der einzelnen Kardinalitäten der partitionslokalen Mengen V_{FS}. Ist die Summe ungleich Null, existieren Partitionen mit Gebieten zur lokalen Adaption, die möglicherweise sogar durch den Dominoeffekt eine Adaptionsfront in andere Partitionen transportieren können, deren lokale Menge V_{FS} leer ist. Arbeitet der Fehlerschätzer bzw. -indikator mit globalem Datenaustausch, so erübrigt sich die Überprüfung, da dann schon vorher entschieden werden kann, ob eine Adaption durchgeführt werden muss, ansonsten erfolgt der Überprüfungsschritt explizit in dieser Funktion.

Steht global fest, dass ein Adaptionsprozess notwendig ist, erfolgt als nächster Schritt die Initialisierung der notwendigen Datenstrukturen (Zeile 5). Dies bedeutet vor allem, dass für jede Nachbarpartition $P_{p'}$ von P_p der Partitionsrand $\partial P_{p,p'}$ bzgl. der Identifikationsmerkmale analysiert wird und entsprechende Tupel in die Mengen $ADS.P^{p'}$ eingetragen werden. Jedes Tupel $(o,k,t) \in ADS.P^{p'}$ erhält als initialen Wert für die

Objektklassifizierung t von o den Wert "Start". Da zu diesem Zeitpunkt im Algorithmus noch keine hängenden Objekte in der Diskretisierung existieren, werden die Mengen von $ADS.O_h$ leer angelegt. Handelt es sich um den ersten Adaptionsvorgang in der Simulation, so wird $ADS.V_c$ ebenfalls mit der leeren Menge initialisiert, ansonsten befinden sich schon Tupel aus der vorherigen Netzmodifikation in $ADS.V_c$[26].

Geometrische Modifikation Bevor nun die eigentliche geometrische Modifikation des Netzes ausgeführt wird, wird das Halosystem $\mathcal{H}_p$ der Partition P_p abgebaut (Zeile 6), da es für den Adaptionsvorgang nicht benötigt wird. Nachdem die geometrische Modifikation abgeschlossen ist, kann das Halosystem $\mathcal{H}_p$ wieder aufgebaut werden (Zeile 8). Das neu aufgebaute Halosystem muss nicht notwendigerweise mit dem übereinstimmen, welches vor der geometrischen Netzmodifikation bestand. Wurde eine Dominowelle über einen Partitionsrand transportiert oder aber Tetraeder am Rand markiert, stimmt die Struktur des Randes zwischen beiden Systemen nicht mehr überein. Eine Zwischenspeicherung des Systems, z.B. zum Zwecke der Laufzeitminimierung, reicht also nicht aus und eine Neuberechnung ist zwingend erforderlich. Nach dem Wiederaufbau von $\mathcal{H}_p$ steht wieder ein verteiltes, konsistentes Netz $\mathcal{M}_\Omega$ gemäß Definition 3.12 zur weiteren Verwendung in der Simulation zur Verfügung. Nach dem Adaptionsprozess folgt normalerweise in einer parallelen Simulation die Lastausgleichsberechnung mit anschließender Datenmigration. Auf die Behandlung dieser Phase im verteilten Objektmodell und den Zusammenhang mit den Abschlusstetraedern wird in Abschnitt 3.4 näher eingegangen.

Die komplette geometrische Modifikation des verteilten Netzes erfolgt in der Funktion Adaptation(), deren Struktur in Algorithmus 3.17 dargestellt ist. Der Algorithmus gliedert sich in vier logische Abschnitte. Zu Beginn von Adaptation() (Zeile 2) wird die Vergröberungsphase der Abschlusstetraeder eingeleitet. Diese erfolgt nur, wenn es *echte Abschlusstetraeder* in $ADS.V_c$ gibt. Der Ablauf der Vergröberung wurde im vorherigen Abschnitt über die Kernoperationen schon erläutert (s. Algorithmus 3.15). Es ist hier anzumerken, dass die Menge der regulär zu verfeinernden Tetraeder V_{FS} durch die Funktion CoarsenClosure() verändert worden sein kann (Markierung eines Abschlusstetraeders). Ebenso können zu diesem Zeitpunkt im Ablauf der Adaption schon hängende Objekte in $ADS.O_h$ vermerkt worden sein.

Die nächste Phase besteht aus der regulären Verfeinerung der Tetraeder, die in V_{FS} enthalten sind (Zeile 3-7). Für die Verwaltung der Grobtetraeder $v \in V_{FS}$, die in diesem Abschnitt des Algorithmus noch nicht aus dem Netz geometrisch entfernt werden, legt die Funktion eine Datenstruktur (Menge) V_{old} an, die diese zwischenspeichert. Der Grund hierfür liegt im Referenzsystem $\mathcal{R}_{\Omega_p}$, dass das urspüngliche, unmodifizierte Netz implizit strukturell repräsentiert. Da in den nachfolgenden Funktionen des Adaptionsalgorithmus mehrfach Referenzmengen von Objekten benötigt werden, können

[26] Diese Annahme gilt nur für eine adaptive Simulation, die mit einem initialen Netz, d.h. mit einem unveränderten Netz, welches ein Tetraedierungsalgorithmus erzeugt hat, gestartet wird. Ein Simulationslauf kann aber auch durch einen Checkpointing-Mechanismus eine vorher abgebrochene Simulation wieder aufsetzen. In diesem Fall benötigt der Adaptionsalgorithmus Informationen über die Abschlusszusammensetzung aus dem vorherigen Lauf, um $ADS.V_c$ initial zu belegen.

Algorithmus 3.17 : Paralleler Adaptionsalgorithmus

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
$V_{FS} \subseteq V_{\Omega_p}$ seien die durch den Fehlerschätzer markierten Tetraeder,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega), \mathcal{P}_\Omega \in \mathcal{M}_\Omega(P)$,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : aktualisierter, partitionslokaler Netzanteil $\mathcal{M}_{\Omega_p}(P)$

```
1  parallel proc Adaptation begin
2      if |ADS.V_c| > 0 then  M*_Ωp ← CoarsenClosures(M_Ωp, V_FS, ADS);
3      V_old ← ∅;
4      foreach v ∈ V_FS do
5          M*_Ωp ← M*_Ωp ⊕ Refine(M*_Ωp, v, "rot", ADS);
6          V_old ← V_old ∪ {v};
7      end
8      M*_Ωp ← ResolveDominoWave(M*_Ωp, V_old, C_p, ADS);
9      M*_Ωp ← ApplyClosures(M*_Ωp, V_old, ADS);
10     M*_Ωp ← M*_Ωp ⊖ V_old;
11     bestimme Referenzsystem R_Ωp neu;
12     M_Ωp ← JoinPartitions(M*_Ωp, C_p, ADS);
13 end proc
```

die Grobtetraeder noch nicht entfernt werden, da ansonsten Inkonsistenzen bei der Analyse von Netzobjektumgebungen entstehen. Der Vorgang der Verfeinerung mittels der Funktion Refine() wurde schon erläutert (s. Algorithmus 3.14). Gemäß der irregulären Verfeinerung werden vom Fehlerschätzer bzw. -indikator markierte Tetraeder grundsätzlich regulär verfeinert, d.h. die rote Regel kommt hierfür zum Einsatz (Zeile 5).

Wurden alle markierten Tetraeder regulär verfeinert, beginnt die nächste Phase in der Adaption. Diese besteht aus der Auflösung der Dominowelle und der Erkennung und dem Hinzufügen der Abschlusstetraeder zur Tetraedierung, um hängende Objekte im verteilten Netz zu beseitigen. Die Dominowelle wird mittels der noch zu beschreibenden, komplexen Funktion ResolveDominoWave() bearbeitet (s. Algorithmus 3.18). Diese Funktion stellt, abgesehen von der Initialisierungsphase, die erste Stelle im Adaptionsalgorithmus dar, in der eine Kommunikation (durch Kantenpropagation) mit den Nachbarpartitionen durchgeführt wird. Die Vergröberungsphase und die Phase der regulären Verfeinerung markierter Tetraeder laufen unabhängig in jeder Partition ab. ResolveDominoWave() modelliert somit den ersten Synchronisationspunkt im Algorithmus. Mittels ApplyClosures() (Zeile 9) werden alle noch hängenden Objekte innerhalb der Partition P_p aufgelöst, so dass nach Beendigung dieser Funktion in den Mengen von

$ADS.O_h$ keine Tupelelemente mehr vorhanden sind. Es bleibt als letzte Aktion (Zeile 10) in dieser Phase die geometrische Entfernung der Grobtetraeder, die in der Menge V_{old} zwischengespeichert wurden (ergänzt um Elemente aus ResolveDominoWave() und ApplyClosures()). Da nun das alte Referenzsystem $\mathcal{R}_{\Omega_p}$ infolge der Strukturänderung auch nicht mehr gültig ist, muss es an dieser Stelle des Algorithmus neu aufgestellt werden (Zeile 11).

Zu diesem Zeitpunkt des Adaptionsalgorithmus stellt jede Partition eine korrekte Tetraedierung im Sinne von Definition 2.23 dar. Global gesehen wurde aber durch die alleinige Konsistenzwahrung der Namensräume von Partitionsrandkanten (Kantenpropagation) die Zusammenhangstruktur der Namensräume von Randknoten und Randdreiecksflächen der Partitionen untereinander gestört. Eine nachträgliche Zusammenführung dieser Namensräume, d.h. ein Datenaustausch mittels Kommunikation, beseitigt diese Konsistenzstörung. Dazu dient die Funktion JoinPartitions() (s. Algorithmus 3.24), die letztendlich die globale Gültigkeit der Tetraedierung wiederherstellt. Nach Ausführung dieser Funktion endet dann auch Algorithmus 3.17.

Auflösung des Dominoeffektes Algorithmus 3.18 beschreibt die Struktur der Funktion ResolveDominoWave(). Sie stellt den zentralen Teil des parallelen Adaptionsalgorithmus dar, da der größte Teil des Laufzeitaufwandes hier verbraucht wird. Die Aufgabe der Funktion ResolveDominoWave() besteht darin, nach der regulären Verfeinerung markierter Tetraeder, zuerst sicher zu stellen, dass Informationen über verfeinerte Partitionsrandkanten in die Nachbarpartitionen bzw. von diesen in die eigene Partition transportiert werden. Dazu dient die Funktion AdjustPartitionBorders() (Zeile 2), deren Struktur in Algorithmus 3.19 beschrieben wird. Nach Ausführung dieser Funktion befindet sich ein aktualisierter Kantenzustand auf dem kompletten Rand der Partition. Zu diesem Zeitpunkt im Algorithmus können Randkanten von Nachbarpartitionen aus verfeinert worden sein, die lokal gesehen ursprünglich nicht dafür vorgesehen waren.

Der nächste Schritt besteht nun darin zu überprüfen, ob Tetraeder mit verfeinerten Kanten nach dem *reduzierten Rot-Grün Regelsatz* eine reguläre Verfeinerungsstrategie benötigen. Dazu wird eine Menge S bestimmt (Zeile 3), in der alle Tupel aus $ADS.O_h^E$ enthalten sind, für die diese Bedingung zutrifft. Mit Hilfe der Funktion DetectClosureType() (s. Algorithmus 3.13) kann diese Menge mit linearem Aufwand (Iteration über die Elementtupel aus $ADS.O_h^E$) konstruiert werden. Ist die Menge S der betroffenen Tetraeder nicht leer (Zeile 4), so muss jedes zu einem Tupel aus S assoziierte Tetraeder ν regulär verfeinert werden (Zeile 6-8). Dadurch werden einerseits hängende Objekte von ν aufgelöst. Andererseits entstehen möglicherweise weitere Tetraeder $\nu' \in \mathcal{R}_V(\nu)$ mit neuen hängenden Objekten. Daher benötigt die Datenstruktur $ADS.O_h$ für jedes bearbeitete Tetraeder ν bzw. ν' eine Aktualisierung. Bei vollständiger Auflösung der hängenden Objekte eines Tetraeders ν, muss das assoziierte Tupel (ν, E_{ref}) zudem aus $ADS.O_h^E$ entfernt werden (Zeile 9).

Entstehen durch den Verfeinerungsvorgang neue hängende Objekte, besteht wiederum die Möglichkeit, dass Tetraeder innerhalb der Partition wegen des reduzierten Adapti-

Algorithmus 3.18 : Algorithmus zur Auflösung des Dominoeffektes

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
$V_{old} \subseteq V_{\Omega_p}$ sei Menge der aus dem Netz zu entfernenden Tetraeder,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega), \mathcal{P}_\Omega \in \mathcal{M}_\Omega(P)$,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : modifizierter, partitionslokaler Netzanteil $\mathcal{M}^*_{\Omega_p}(P)$

```
1  parallel proc ResolveDominoWave begin
2      AdjustPartitionBorders(M*_Ωp, C_p, ADS);
3      S ← {(v, E_ref) ∈ ADS.O_h^E | DetectClosureType(M*_Ωp, v, ADS) = "rot"};
4      while |S| > 0 do
5          foreach (v, E_ref) ∈ S do
6              V* ← Refine(M*_Ωp, v, "rot", ADS);
7              foreach v* ∈ V* do M*_Ωp ← M*_Ωp ⊕ v*;
8              V_old ← V_old ∪ {v};
9              aktualisiere ADS.O_h über alle aufgelößten oder neuen "hängenden"
               Objekte von v und entferne Tupel (v, E_ref) aus ADS.O_h^E, falls möglich;
10         end
11         AdjustPartitionBorders(M*_Ωp, C_p, ADS);
12         S ← {(v, E_ref) ∈ ADS.O_h^E | DetectClosureType(M*_Ωp, v, ADS) = "rot"};
13     end
14 end proc
```

onsregelsatzes regulär verfeinert werden müssen. Zudem könnten auch neue Randkanten von einer Verfeinerung betroffen sein. Dies bedeutet, dass der Vorgang der Kantenpropagation in Nachbarpartitionen inklusive Bestimmung der regulär zu verfeinernden Tetraeder wiederholt werden muss (Zeile 11-12). Ein Dominoeffekt ist damit eingetreten, der partitionslokal nicht notwendigerweise gestoppt werden kann. Eine mehrfach wiederholte Anwendung der genannten Schritte in einer Schleife ist also erforderlich. Diese kann nur dann verlassen werden, wenn kein Tetraeder im globalen Netz mehr regulär verfeinert werden muss, d.h. alle lokalen Mengen S leer sind. Tritt dieser Fall ein, können keine neuen Kanten bzw. Knoten im Netz mehr entstehen und die verbliebenen hängenden Objekte im Netz werden durch Anwendung der grünen Abschlussregeln vollständig aufgelöst (s. Algorithmus 3.23).

Der Laufzeitaufwand von Algorithmus 3.18 wird maßgeblich von der Kardinalität von S bestimmt, d.h. S bestimmt indirekt die Anzahl der Iterationen der while-Schleife. S ändert sich aber stetig wegen der durch die Kantenpropagation evtl. neu hinzukommenden Tetraeder, die durch den reduzierten Adaptionsregelsatz regulär verfeinert werden müssen. Partitionslokal kann somit nicht entschieden werden, wieviele Schleifendurchläufe (Zeile 4-13) notwendig sind. Auch global lässt sich keine Aussage über

die Anzahl der Schleifendurchläufe machen, da die Dynamik dieses Vorganges sowohl von der Vernetzungsstruktur der Diskretisierung als auch von den Auswahlkriterien des Fehlerschätzers bzw. -indikators zur Markierung von Tetraedern abhängt. Im günstigsten Fall ist S gleich zu Beginn der Funktion ResolveDominoWave() leer, d.h. eine Dominowelle tritt nicht auf. Der schlechteste Fall wäre, dass der Fehlerschätzer wenige Tetraeder markiert, aber die Menge S stetig neue Tetraeder durch die Funktion DetectClosureType() hinzugefügt bekommt und zwar derart, dass sämtliche Tetraeder der Partition als Tupelelement in der Menge S landen. Dies würde dann einer Vollverfeinerung entsprechen, die die Menge der Tetraeder verachtfacht (rote Verfeinerungsregel liefert 8 Tetraeder). Eine solche drastische Erhöhung der Datenmenge ist wegen der damit verbundenen starken Laufzeiterhöhung im numerischen Teil der Simulation grundsätzlich unerwünscht, kann aber in der Praxis nicht immer verhindert werden.

Kantenpropagation Die Kantenpropagation in Algorithmus 3.18 wird durch einen Aufruf der Funktion AdjustPartitionBorders() durchgeführt, deren Ablaufstruktur in Algorithmus 3.19 veranschaulicht ist. Sie stellt die wichtigste Stelle im parallelen Adaptionsalgorithmus dar, da sie die Konsistenzwahrung der Namensräume von Partitionsrandkanten sicherstellt. AdjustPartitionBorders() ist in drei Phasen eingeteilt: eine Korrekturphase, eine Selektionsphase und eine Bearbeitungsphase.

Die Korrekturphase (Zeile 2) bereinigt aus der vorherigen Bearbeitungsiteration *nicht aufgelöste Partitionsrandkanten*, deren Kantenklassifizierung vom Typ 2 sind. Dies geschieht in der Funktion ProcessUnresolvedEdges() (s. Algorithmus 3.22). Nicht aufgelöste Kanten entstehen genau dann, wenn eine Nachbarpartition ein Tetraeder regulär verfeinert, welches eine Dreiecksfläche auf dem Partitionsrand besitzt, und in der gleichen Iteration eines der entstandenen Teiltetraeder wiederum verfeinert (induziert vom Abschlusstyp eines Nachbartetraeders). Somit gibt es mindestens eine Kante mit Klassifizierung Typ 2, die wiederum verfeinert ist. Am Rand in der anderen Partition existiert allerdings noch keine Kante vom Typ 2, wenn die Information zur Verfeinerung aus der Nachbarpartition propagiert wird. Werden solche Typ-2-Kanten identifiziert, werden sie in einer Menge $ADS.U_E^{p'}$ zwischengespeichert, bis sie im darauffolgenden Aufruf von AdjustPartitionBorders() in der Korrekturphase bearbeitet werden können, da dann gemäß DetectClosureType() durch die rote Verfeinerungsstrategie am Partitionsrand die dazugehörige Typ-2-Kante erzeugt wurde.

Sind die nicht aufgelösten Kanten abgearbeitet, folgt die nächste Phase, in der die Partitionsrandkanten aus $ADS.P_E^{p'}$ bzgl. ihrer Kantenklassifikation in zwei Mengen $ADS.T_1^{p'}$ und $ADS.T_2^{p'}$ für alle Nachbarpartitionen $P_{p'}$ einsortiert werden (Zeile 3-12). Es werden hierfür nur Kanten betrachtet, die Typ 1 bzw. Typ 2 entsprechen. Kanten mit Klassifizierung "Start" werden nicht beachtet, da sie unverändert bleiben. Es ist für den Kantenabgleichalgorithmus wichtig, dass erst Kanten vom Typ 2 aussortiert werden (Zeile 5) und erst dann Typ-1-Kanten (Zeile 7). Nur so können Kanten, die sich als nicht auflösbar für die Nachbarpartition $P_{p'}$ herausstellen, von Typ-1-Kanten unterschieden werden. Für hängende Partitionsrandkanten gilt zudem, dass das assoziierte

Algorithmus 3.19 : Algorithmus zum Kantenabgleich an Partitionsrändern

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega), \mathcal{P}_\Omega \in \mathcal{M}_\Omega(P)$,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : modifizierter, partitionslokaler Netzanteil $\mathcal{M}^*_{\Omega_p}(P)$

```
1  parallel proc AdjustPartitionBorders begin
2      M*_Ωp ← ProcessUnresolvedEdges(M*_Ωp, C_p, ADS);
3      foreach p' ∈ neighbor*(P_p) do
4          foreach (e, k, t) ∈ ADS.P_E^p' do
5              if t = "Typ 2" then
6                  ADS.T_2^p' ← ADS.T_2^p' ∪ {e};
7              else if e ist hängende Kante then
8                  ADS.T_1^p' ← ADS.T_1^p' ∪ {e};
9                  ADS.P_E^p' ← ADS.P_E^p' \ {(e, k, t)};
10             end
11         end
12     end
13     M*_Ωp ← ProcessTypeOneEdges(M*_Ωp, C_p, ADS);
14     M*_Ωp ← ProcessTypeTwoEdges(M*_Ωp, C_p, ADS);
15 end proc
```

Datentupel aus $ADS.P^{p'}$ entfernt werden kann (Zeile 9), da die Kante in einem nachfolgenden Aufruf von AdjustPartitionBorders() nicht mehr weiter beachtet werden muss.

In der letzten Phase (Zeile 13-14) erfolgt nun der eigentliche Abgleich der Namensräume sowie die geometrische Kantenverfeinerung, welche durch die Kantenpropagation induziert wird. Dazu dienen die beiden Funktionen ProcessTypeOneEdges() und ProcessTypeTwoEdges(). Die Funktionen müssen in dieser Reihenfolge aufgerufen werden, damit die Konsistenz der Namensräume zwischen den Nachbarpartitionen eingehalten und die geometrische Korrektheit gewahrt werden kann.

Verarbeitung der Kanten vom Typ 1 Algorithmus 3.20 zeigt nun die Struktur der Funktion ProcessTypeOneEdges() auf, die den Abgleich von Partitionsrandkanten der Klassifizierung "Typ 1" zwischen Nachbarpartitionen durchführt. Die Funktion besteht aus drei Phasen. Als erstes werden die in Algorithmus 3.19 in die Menge $ADS.T_1^{p'}$ aussortierten, verfeinerten Grobkanten für jede Nachbarpartition $P_{p'}$ analysiert und ihr Identifikationsmerkmal für den Abgleich in einen Sendevektor $s^{p'}$ übertragen (Zeile 2-5). Danach kann $ADS.T_1^{p'}$ geleert werden, da die Grobkanten in dieser Iteration der Kantenpropagation nicht weiter betrachtet werden müssen. Ein Datenaustausch mit allen beteiligten Nachbarpartitionen $P_{p'}$ schließt die Aufsetzphase ab (Zeile 6).

Algorithmus 3.20 : Algorithmus zum Abgleich von Partitionsrandkanten vom Typ 1

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega), \mathcal{P}_\Omega \in \mathcal{M}_\Omega(P)$,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : modifizierter, partitionslokaler Netzanteil $\mathcal{M}^*_{\Omega_p}(P)$

```
1  parallel proc ProcessTypeOneEdges begin
2      foreach p' ∈ neighbor*(P_p) do
3          for i = 1,...,|ADS.T_i^{p'}| do  s_i^{p'} ← ω_p(e, p');
4          ADS.T_i^{p'} ← ∅;
5      end
6      Datenaustausch mit Nachbarn p': {r^{p'}} ← Exchange({s^{p'}}, C_p)
7      E_add ← ∅;
8      foreach p' ∈ neighbor*(P_p) do
9          for i = 1,...,|r^{p'}| do
10             bestimme e ∈ ∂P_{p,p'}(E_{Ω_p}) mit ω_p(e) = r_i^{p'};
11             if e ist keine hängende Kante then
12                 verfeinere Kante e und aktualisiere ADS.O_h;
13                 ADS.P_E^{p'} ← ADS.P_E^{p'} \ {(e, ω_p(e, p'), t) | t ∈ {"Start","Typ 1"}};
14             end
15             foreach p* ∈ neighbor*(P_p) : e ∈ ∂P_{p,p*}(E_{Ω_p}) do
16                 E_add ← E_add ∪ {(ω_p(e), p*)};
17                 ADS.P_E^{p*} ← ADS.P_E^{p*} \ {(e, ω_p(e, p*), t) | t ∈ {"Start","Typ 1"}};
18             end
19         end
20     end
21     foreach p' ∈ neighbor*(P_p) do
22         sammle alle ω_p-Daten über gesendete, empfangene Kanten sowie aus E_add
               in Vektor u^{p'} der Länge |s^{p'}| + |r^{p'}| + |{(k, p*) ∈ E_add | p' = p*}|;
23     end
24     Datenaustausch mit Nachbarn p': {w^{p'}} ← Exchange({u^{p'}}, C_p);
25     foreach p' ∈ neighbor*(P_p) do
26         integriere empfangene ω_{p'}-Daten aus Vektor w^{p'} in lokale ω_p-Abbildung
               und aktualisiere ADS.O_h und ADS.P_E^{p'} über neu erzeugte bzw entfernte
               Objekte;
27     end
28 end proc
```

Die nächste Phase beschäftigt sich mit der geometrischen Modifizierung des Gitters durch empfangene Kanteninformationen, die lokal nicht von einer Verfeinerung betroffen sind. Zudem müssen die lokalen Identifikationsmerkmale der lokal verfeinerten Kanten (zwei Teilkanten und ein Mittelpunktknoten) gesammelt werden, um sie an die Nachbarpartitionen weiterreichen zu können. Hierbei ist allerdings zu beachten, dass es Situationen gibt, in denen mindestens drei Partitionen, z.B. die Partition P_a, P_b und P_c, eine Randkante miteinander teilen und nur von einer Partition, z.B. von P_a, die Verfeinerung durchgeführt wird. Propagiert nun P_a die Verfeinerungsinformation nach P_b und P_c, so verfeinern diese die gemeinsame Kante und senden die Identifikationsmerkmale der neue entstandenen Objekte an P_a zurück. Das Problem besteht nun darin, dass zwar die Geometrie korrekt aufgebaut ist, d.h. die Randkante wurde von allen Partitionen verfeinert, aber P_b und P_c kennen nicht die Identifikationsmerkmale ihrer neuen Teilobjekte der Grobkante untereinander. Nur P_a kennt diese. Eine Inkonsistenz der partitionslokalen Namenräume ist aufgetreten. Um diese Inkonsistenz zu vermeiden, wird eine partitionslokale Menge E_{add} innerhalb der Funktion ProcessTypeOneEdges() eingeführt, die Tupel (k, p^*) mit $k, p^* \in \mathbb{N}$ speichert. Diese Tupel dienen zur zusätzlichen Propagation von benötigten Objektinformationen einer Grobkante e mit $k = \omega_p(e)$ in die Nachbarpartitionen P_{p^*}, für die die beschriebene Inkonsistenz der Namensräume möglich wäre.

Die Modifizierungsphase analysiert nun für jede Nachbarpartition $P_{p'} \in \text{neighbor}(P_p)$ den empfangenen Datenvektor $r^{p'}$ (Zeile 9) und bestimmt die zugehörige lokale Randkante e mit $\omega_p(e) = r_i^{p'}$. Diese Kante muss lokal existieren, da der Rand zu diesem Zeitpunkt konsistent ist (Zeile 10). Handelt es sich bei der Kante e um keine hängende Kante, dann wurde eine Adaptionsfront vom Nachbarn $P_{p'}$ nach P_p transportiert. D.h. die Kante e muss verfeinert werden (inkl. Aktualisierung der neuen hängenden Objekte von e) und das assoziierte Tupel aus $ADS.P_E^{p'}$ entfernt werden, da die Kante als bearbeitet gilt (Zeile 11-14). Zur Konsistenzwahrung wird nun das Datentupel $(\omega_p(e), p^*)$ der Kante e für jede Nachbarpartition P_{p^*}, die e als gemeinsames, geometrisches Randkantenobjekt enthält, in E_{add} eingetragen. Auch hier gilt, dass das zu e assoziierte Tupel aus $ADS.P_E^{p^*}$ entfernt wird (Zeile 15-18).

Die letzte Phase von Algorithmus 3.20 integriert nun die notwendigen Informationen über die Identifikationsmerkmale in die ω_p-Abbildung. Dazu trägt sie alle Daten für eine Nachbarpartition $P_{p'}$ aus dem eigenen gesendeten Vektor $s^{p'}$, dem von $P_{p'}$ empfangenen Vektor $r^{p'}$ und aus der konstruierten Menge E_{add}, für die ein Tupel existiert, in einem neuen Sendevektor $u^{p'}$ zusammen und sendet diesen an den Nachbarn $P_{p'}$ (Zeile 21-24). Nach dem Datenaustausch werden dann die empfangenen Informationen über die Identifikationsmerkmale der Nachbarpartitionen in die eigene ω_p-Abbildung integriert sowie die notwendigen Datenstrukturen über neue bzw. entfernte Objekte in $ADS.O_h$ bzw. $ADS.P_E^{p'}$ aktualisiert. Dies gilt insbesondere für Objekte, die aus der Menge E_{add} von $P_{p'}$ stammen, um Inkosistenzen zu vermeiden. Nach dieser Phase endet die Funktion ProcessTypeOneEdges(). Der Partitionsrand ∂P_p hat nun alle eigenen Kanten der Klassifizierung Typ 1 zu seinen Nachbarn propagiert und von diesen (mög-

licherweise neue) empfangen und in die eigene Struktur integriert.

Verarbeitung der Kanten vom Typ 2 Der nächste Schritt beim Randkantenabgleich zwischen Partitionen ist nun, alle Kanten mit Klassifizierung Typ 2, also Kanten, die auf Dreiecksflächen von Tetraedern bei der regulären Verfeinerung entstehen, an die Nachbarn zu propagieren. Dies geschieht in der Funktion ProcessTypeTwoEdges(), die in Algorithmus 3.21 aufgezeigt wird.

Zunächst werden wie bei der Funktion ProcessTypeOneEdges() die in der Menge $ADS.T_2^{p'}$ gesammelten Kanten vom Klassifizierungstyp zwei (aus Algorithmus 3.19) in einem Sendevektor $s^{p'}$ für jeden Nachbarn $P_{p'}$ mittels der lokalen ω_p-Abbildung vermerkt (Zeile

Algorithmus 3.21 : Algorithmus zum Abgleich von Partitionsrandkanten vom Typ 2

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega), \mathcal{P}_\Omega \in \mathcal{M}_\Omega(P)$,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : modifizierter, partitionslokaler Netzanteil $\mathcal{M}^*_{\Omega_p}(P)$

```
1  parallel proc ProcessTypeTwoEdges begin
2      foreach p' ∈ neighbor*(P_p) do
3          for i = 1,...,|ADS.T_2^p'| do
4              s_i^p' ← ω_p(e);
5              setze Typ t im Tupel (e, ω_p(e,p'), t) ∈ ADS.P_E^p' auf "Typ 1";
6          end
7          ADS.T_2^p' ← ∅;
8      end
9      Datenaustausch mit Nachbarn p': {r^p'} ← Exchange({s^p'}, C_p)
10     foreach p' ∈ neighbor*(P_p) do
11         for i = 1,...,|r^p'| do
12             if ∄e ∈ ∂P_{p,p'}(E_{Ω_p}) : r_i^p' = ω_p(e,p') then
13                 seien n'_1, n'_2 die lokalen Mittelpunktknoten der verfeinerten
                   Tetraederkanten vom Typ 1;
14                 ADS.U^p' ← ADS.U^p' ∪ {(r_i^p', n'_1, n'_2)};
15             else
16                 aktualisiere ω_p-Abbildung bzgl. der Kante e;
17                 setze Typ t im Tupel (e, ω_p(e,p'), t) ∈ ADS.P_E^p' auf "Typ 1";
18             end
19         end
20     end
21 end proc
```

2-8). Zudem müssen für diese Kanten der Klassifizierungstyp im assoziierten Tupel aus $ADS.P_E^{p'}$ *von Typ 2 auf Typ 1 geändert werden* (Zeile 5). Dies erfolgt deshalb, weil Typ-2-Kanten potentiell ebenfalls verfeinert sein können. Infolge der Konsistenzwahrung und der Algorithmenstruktur des Kantenabgleichverfahrens kann aber eine Typ-2-Kante erst verfeinert werden, wenn sie auf der Gegenseite auch geometrisch existiert, d.h. das zugehörige Tetraeder mit der roten Strategie verfeinert wurde. Die Typänderung bewirkt, dass die Kante in der nächsten Iteration des Kantenabgleichs verfeinert werden kann. Sind die einzelnen Sendevektoren $s^{p'}$ konstruiert, erfolgt der Datenaustausch mit den Nachbarn $P_{p'}$ (Zeile 9) und die Aufsetzphase endet.

Nun folgt die Integrationsphase, in der die lokale ω_p-Abbildung mit den empfangenen Daten aus $r^{p'}$ aktualisiert wird. Dazu wird für jedes empfangene Identifikationsmerkmal (Zeile 11) die lokale Kante auf dem Rand bestimmt. Existiert keine geometrische Repräsentation zu dem Identifikationsmerkmal, so handelt es sich um eine nicht-auflösbare Kante vom Typ 2 (das zugehörige Tetraeder wurde noch nicht regulär verfeinert) und muss zwischengespeichert werden (Zeile 12-15). Die Menge $ADS.U_E^{p'}$ nimmt diese Kante für die folgende (in der nächsten Iteration des Abgleiches) Auflösung auf. Hierzu werden die Konstruktionsknoten für die lokale Kante bestimmt, die sich aus den Mittelpunktknoten zweier verfeinerter Tetraederkanten vom Typ 1 ermitteln lassen, da diese ja im Abgleichschritt in ProcessTypeOneEdges() vorher schon konstruiert wurden und somit bekannt sind. Das empfangene Identifikationsmerkmal $r_i^{p'}$ der Kante sowie jene der lokalen Konstruktionsknoten n_1' und n_2' werden als Tupel der Menge $ADS.U_E^{p'}$ für die spätere Verarbeitung hinzugefügt (Zeile 13). Kann jedoch das empfangene Identifikationmerkmal lokal zugeordnet werden, so erfolgt die gleiche Vorgehensweise der ω_p-Aktualisierung wie bei Typ-1-Kanten. Auch hier muss das Klassifizierungsmerkmal im assoziierten Tupel in $ADS.P_E^{p'}$ zu Typ 1, wie oben bereits beschrieben, geändert werden (Zeile 15-18). Wurden alle Elemente aus $r^{p'}$ für alle Nachbarn $P_{p'}$ abgearbeitet, endet die Integrationsphase und damit auch die Funktion ProcessTypeTwoEdges().

Verarbeitung nicht aufgelöster Kanten vom Typ 2 Algorithmus 3.22 verdeutlicht die Struktur der Funktion ProcessUnresolvedEdges(). Das Ziel dieser Funktion ist es, vorhandene, aus dem letzten Iterationschritt des Abgleichs stammende Kanten vom Typ 2, die nicht aufgelöst werden konnten, in das Netz zu integrieren und die Nachbarpartition darüber zu informieren. Dazu wird für jedes Tupel (k, n_1, n_2), das sich in der Menge $ADS.U_E^{p'}$ befindet, die lokale Partitionsrandkante $e \in \partial P_{p,p'}$ mit den Konstruktionsknoten n_1 und n_2 bestimmt. e muss zu diesem Zeitpunkt existieren, da die rote Verfeinerungsstrategie für das Tetraeder in der lokalen Adaptionsphase angewendet wurde, d.h. alle drei Kanten auf dem Partitionsrand sind verfeinert worden (s. auch DetectClosureType()). Mittels dieser Kante e kann nun die ω_p-Abbildung mit dem im Tupel gespeicherten Identifikationsmerkmal k für die Kante in der benachbarten Partition $P_{p'}$ aktualisiert werden (Zeile 4-5).

Um nun die Typ-2-Kante e in die Nachbarpartition zu propagieren, wird sie in die Menge $ADS.T_2^{p'}$ aufgenommen. Dadurch kann ihr lokales Identifikationsmerkmal $\omega_p(e)$

Algorithmus 3.22 : Algorithmus zum Auflösen von Partitionsrandkanten vom Typ 2

Eingabe : $p \in \{1, \ldots, P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega), \mathcal{P}_\Omega \in \mathcal{M}_\Omega(P)$,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : modifizierter, partitionslokaler Netzanteil $\mathcal{M}^*_{\Omega_p}(P)$

```
1  parallel proc ProcessUnresolvedEdges begin
2      foreach p' ∈ neighbor*(P_p) do
3          foreach (k, n_1, n_2) ∈ ADS.U^{p'} do
4              ermittle Kante e ∈ ∂P_{p,p'}(E_{Ω_p}) mit e = cstr(n_1, n_2, );
5              aktualisiere ω_p-Abbildung bzgl. gefundener Kante e mit k;
6              ADS.T_2^{p'} ← ADS.T_2^{p'} ∪ {e};
7              setze Typ t im Tupel (e, ω_p(e, p'), t) ∈ ADS.P_E^{p'} auf "Typ 1";
8          end
9          ADS.U^{p'} ← ∅;
10     end
11     M*_{Ω_p} ← ProcessTypeTwoEdges(M*_{Ω_p}, C_p, ADS);
12 end proc
```

unter Ausnutzung von Algorithmus 3.21 (Zeile 11) ausgetauscht werden. Für die Kante e muss zudem der Klassifizierungstyp wie bei ProcessTypeTwoEdges() in $ADS.P_E^{p'}$ auf Typ 1 geändert werden, um eine mögliche Verfeinerung in der nächsten Iteration des Abgleiches durchführen zu können (Zeile 7). Nachdem alle Kanten aus $ADS.U^{p'}$ verarbeitet sind, kann die Menge geleert werden (Zeile 9). Der Aufruf der Funktion ProcessTypeTwoEdges() zum Abgleich der nun aufgelösten Kanten schließt die Funktion ProcessUnresolvedEdges() ab.

Die Funktionen ProcessTypeOneEdges() und ProcessTypeTwoEdges() sowie die Funktion ProcessUnresolvedEdges() bilden zusammen mit AdjustPartitionBorders() die Kantenpropagationsphase des parallelen Zwei-Schritt Algorithmus (vgl. Abbildung 3.5). Durch den wiederholten Aufruf von AdjustPartitionBorders() wird die Konsistenz der Namensräume zwischen den Partitionen für die Kanten hergestellt. Die Zahl der Iterationen, die dazu notwendig ist, ist abhängig von der Struktur der Diskretisierung, dem Partitionsrandverlauf und der Dynamik der Markierung, die Fehlerschätzer bzw. -indikator liefern. Der Dominoeffekt spielt daher maßgeblich eine Rolle bei der Laufzeit der gesamten Adaption in der Simulation. Die Kantenpropagation stellt somit mit der lokalen Adaption innerhalb der regulären Auflösungsphase den komplexen Hauptteil des Algorithmus zur irregulären Adaption dar, wie in Abbildung 3.5 ersichtlich ist.

Abschlussdetektion und -auflösung Nach der regulären Auflösungphase, die durch die Funktion ResolveDominoWave() in Adaptation() eingeleitet wird, folgt die Abschluss-Detektion, in der die verbliebenen noch hängenden Objekte der partitionslokal ungülti-

Algorithmus 3.23 : Algorithmus zur Anwendung der grünen Abschlusstetraeder

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
$V_{old} \subseteq V_{\Omega_p}$ sei Menge der aus dem Netz zu entfernenden Tetraeder,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : modifizierter, partitionslokaler Netzanteil $\mathcal{M}^*_{\Omega_p}(P)$

```
1  parallel proc ApplyClosures begin
2      foreach (v, E_ref) ∈ ADS.O_h^E do
3          switch DetectClosureType(M_Ωp, v, ADS) do
4              case "grün 1:2":  V* ← Refine(M_Ωp, v, "grün 1:2", ADS);
5              case "grün 1:3":  V* ← Refine(M_Ωp, v, "grün 1:3", ADS);
6              case "grün 1:4a":  V* ← Refine(M_Ωp, v, "grün 1:4a", ADS);
7              case "grün 1:4b":  V* ← Refine(M_Ωp, v, "grün 1:4b", ADS);
8          end
9          foreach v* ∈ V* do  M*_Ωp ← M_Ωp ⊕ v*;
10         V_old ← V_old ∪ {v};
11         aktualisiere ADS.O_h über alle aufgelösten "hängenden" Objekte von v und
               entferne Tupel (v, E_ref) aus ADS.O_h^E falls möglich;
12     end
13 end proc
```

gen Tetraedierung beseitigt werden. In Algorithmus 3.23 ist die Struktur der Funktion ApplyClosures() aufgezeigt, die die letzte geometrische Modifikation im Netz durchführt (s. Algorithmus 3.17). Für jedes vorhandene Tupel $(v, E_{ref}) \in ADS.O_h^E$ wird entschieden, welcher grüne Regelsatz für ein Tetraeder v anzuwenden ist (Zeile 3-8). Wurde die Menge der Teiltetraeder V^* für ein Abschlusstetraeder v zur Diskretisierung hinzugefügt, kann v zur Entfernung markiert, d.h. in die Menge V_{old} aufgenommen werden. Die nachfolgende Aktualisierung der Datenstrukturen für $ADS.O_h$ schließt den Vorgang für ein Tetraeder mit hängenden Objekten ab. Sind alle Tupel verarbeitet und damit aus $ADS.O_h^E$ entfernt, endet die Funktion und die Diskretisierung innerhalb der Partition beschreibt, abgesehen von den noch vorhandenen Grobtetraedern (in V_{old} vermerkt) und dem Referenzsystem (welches die Grobtetraeder berücksichtigt) eine gültige Teiltetraedierung. ApplyClosures() arbeitet lokal ohne Kommunikation, genauso wie die Phase, in der vom Fehlerschätzer bzw. -indikator markierte Tetraeder regulär verfeinert werden. Abbildung 3.5 zeigt in der Ablaufstruktur die Abschluss-Detektion, in der die Funktion ApplyClosures() verwendet wird, als vorletzte Phase im Adaptionsalgorithmus.

Namensraumzusammenführung der Partitionen Zum Abschluss der Adaptionsphase in der Simulation bleibt noch, wie aus Algorithmus 3.17 (Zeile 16-18) ersichtlich ist, die Ent-

fernung der alten Grobtetraeder inklusive ihrer Konstruktionselemente[27] sowie der Neuaufbau des partitionslokalen Referenzsystems $\mathcal{R}_{\Omega_p}$. Da zu diesem Zeitpunkt im Adaptionsalgorithmus die Namensräume für Knoten und Dreiecksflächen nur teilweise bzw. gar nicht konsistent sind, erfolgt als letzter Schritt die Zusammenführung der Objektnamensräume benachbarter Partitionen. Dazu dient die Funktion JoinPartitions(), deren Ablaufstruktur in Algorithmus 3.24 dargestellt ist.

Algorithmus 3.24 besteht aus drei Phasen. In der ersten Phase werden die benötigten Daten für die Zusammenführung bzw. Aktualisierung der partitionslokalen Namensräume aufgesetzt und in einen Sendevektor bzw. ein Tupel $s^{p'}$ von Einzelvektoren für die Partitionsrandobjekttypen gespeichert (Zeile 2-7). Diese Daten werden anschließend an die betreffenden Nachbarpartitionen $P_{p'}$ versendet und deren Informationen empfangen (Zeile 8). Zu den Daten für die Partitionsrandknoten zählen das lokale Identifikationsmerkmal sowie die Koordinaten eines Knotens. Für Kanten und Dreiecksflächen werden ebenfalls die lokalen Identifikationsmerkmale und zusätzlich die Konstruktionselemente (aus den Referenzmengen $\mathcal{R}_N(e)$ bzw. $\mathcal{R}_E(f)$ einer Kante e bzw. Fläche f) ausgetauscht. Mit diesen Daten kann der Namensraum, d.h. die ω_p-Abbildung, für alle Partitionsrandobjekte vollständig rekonstruiert werden.

Die zweite Phase dient der Initialisierung der Suchdatenstrukturen für die lokalen Randobjekte. Es gibt drei Datenstrukturen S_N, S_E und S_F für die Rekonstruktion der Namensräume für Knoten, Kanten und Dreiecksflächen. Im Algorithmus sind die Suchdatenstrukturen aus Übersichtsgründen als Mengen modelliert. Zu diesen Mengen gehören Schlüsselfunktionen h_N, h_E und h_F, die eindeutige Schlüssel zu einem Objekttyp liefern. Für eine reale Implementierung empfielt es sich, *Hashtabellen mit geeigneten Hashfunktionen* zu benutzen, da sie mit linearem Zeitaufwand konstruiert werden können und Suchoperationen somit approximativ in der Aufwandsklasse $\mathcal{O}(1)$ liegen. Die Datenstrukturen werden mit den lokalen Objekten des gesamten Partitionsrandes ∂P_p aufgefüllt. Eine Unterscheidung nach Nachbarpartitionen ist nicht nötig.

Da die Identifikationsmerkmale für die Suche in den Datenstrukturen nicht zur Verfügung stehen (gerade diese sollen ja aktualisiert werden), wird eine andere Berechnungsmethode für die Hashfunktionen benötigt (vgl. Standardfunktion bei Hastabellen für die ω_p-Abbildung (3.11)). Dabei ergeben sich für die drei Objekttypen jeweils unterschiedliche Berechnungsvorschriften, die in Algorithmus 3.24 durch die Funktionen h_N, h_E und h_F verdeutlicht werden. In der *padfem*^{2}-Simulationsumgebung (s. a. [BKM03]) haben sich als Berechnungsvorschriften für die Hashfunktionen folgende Methoden in der Praxis bewährt, da sie einfach zu berechnen sind und dennoch den möglichen Schlüsselraum der Hashtabelle effektiv abdecken.

Ein Knoten $p = (x, y, z)^T \in \mathbb{R}^3$ wird durch die Funktionsvorschrift

$$h_N : \mathbb{R}^3 \to \{0, \ldots, m-1\};\quad (x, y, z)^T \mapsto \left(x \cdot \alpha \exp(1) + y \cdot \beta\sqrt{2} + z \cdot \gamma\pi\right)_{\text{cast}} \bmod m \tag{3.13}$$

[27] Die Konstruktionsknoten der Tetraeder bleiben allerdings erhalten, da diese Teil der neuen, gültigen Tetraedierung sind.

Algorithmus 3.24 : Algorithmus zur Zusammenführung der Partitionen

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega), \mathcal{P}_\Omega \in \mathcal{M}_\Omega(P)$,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : konsistenter, partitionslokaler Netzanteil $\mathcal{M}_{\Omega_p}(P)$

```
1  parallel proc JoinPartitions begin
2      foreach p' ∈ neighbor*(P_p) do
3          for i = 1,...,|ADS.P_N^p'|, n_i ∈ ADS.P_N^p' do  s_i^N ← (ω_p(n_i), n_i.x, n_i.y, n_i.z);
4          for i = 1,...,|ADS.P_E^p'|, e_i ∈ ADS.P_E^p' do  s_i^E ← (ω_p(e_i), R_N(e_i));
5          for i = 1,...,|ADS.P_F^p'|, f_i ∈ ADS.P_F^p' do  s_i^F ← (ω_p(f_i), R_E(f_i));
6          s^p' ← (s^N, s^E, s^F);
7      end
8      Datenaustausch mit Nachbarn p': {r^p'} ← Exchange({s^p'}, C_p);
9      S_N ← ∅;
10     for i = 1...,|∂P_p(N_Ωp)|, n_i ∈ ∂P_p(N_Ωp) do
11         S_N ← S_N ∪ {(h_N(n_i.x, n_i.y, n_i.z), ω_p(n_i), n_i)};
12     end
13     S_E ← ∅;
14     for i = 1...,|∂P_p(E_Ωp)|, e_i ∈ ∂P_p(E_Ωp) do
15         S_E ← S_E ∪ {(h_E(R_N(e_i)), ω_p(e_i), e_i)};
16     end
17     S_F ← ∅;
18     for i = 1...,|∂P_p(F_Ωp)|, f_i ∈ ∂P_p(F_Ωp) do
19         S_F ← S_F ∪ {(h_F(R_E(f_i)), ω_p(f_i), f_i)};
20     end
21     M_Ωp ← PastePartitions(M*_Ωp, S_N, S_E, S_F, {r^p'});
22 end proc
```

auf das Intervall $[0, m-1] \subset \mathbb{N}_0$ abgebildet. Dabei bezeichne $m = \text{nextprime}(2^q)$ die kleinste Primzahl, für die gilt $m \geq 2^q$ für ein geeignet gewähltes $q \in \mathbb{N}$. Desweiteren führt der Operator $(\cdot)_{\text{cast}}$ eine bitweise Konvertierung einer im Rechner dargestellten Fließkommazahl[28] in eine vorzeichenlose Ganzzahl (float-to-uint cast) aus. Die Anwendung des cast-Operators ist notwendig, um das Ergebnis des Operators als Indexwert für eine Tabelle nutzen zu können. Die Hashfunktion h_N nutzt eine Variante der multiplikativen Methode (Divisionsrest-Methode), bei der ein Schlüssel zur Bestimmung des Tabelleneintrages mit einer transzendenten Zahl verknüpft wird (s. z.B. [OW90]). Im Falle eines Knotens aus dem $\mathbb{R}^3$ dienen hier drei unterschiedliche transzendente Zahlen $\exp(1)$, $\sqrt{2}$ und π mit geeignet gewählten Gewichtungen α, β und γ, die mit jeweils ei-

[28] in 2er-Komplementdarstellung

ner einzelnen Raumkoordinate multipliziert und aufsummiert werden. Die Anwendung des cast-Operators zusammen mit der Modulo-Operation ergeben letztendlich den Indexwert für den Tabellenplatz. Empirische Untersuchungen in der *padfem*^{2}-Umgebung zeigen, dass die Gewichte mit $\alpha = \beta = \gamma = \frac{1}{2}$ eine gute, d.h. in Bezug zu m eine möglichst gleichmäßige Verteilung der Schlüssel/Wert-Paare in der Hashtabelle für Knotendaten ergeben, wenn diese auf Partitionsrandoberflächen räumlich verteilt vorliegen.

Die Hashfunktion h_E für Kanten gestaltet sich nach einem ähnlichen Prinzip wie die Hashfunktion h_N in Gleichung 3.13. Allerdings werden hierzu die Identifikationsmerkmale der lokalen Konstruktionsknoten der Kante verwendet. Sei $e = \mathrm{cstr}(n_1, n_2)$ mit $n_1, n_2 \in \partial P_p(N_{\Omega_p})$ eine Partitionsrandkante der Partition P_p, $a = \omega_p(n_1)$ und $b = \omega_p(n_2)$. Dann bildet die Hashfunktion

$$h_E : \mathbb{N} \times \mathbb{N} \to \{0, \ldots, m-1\}; \quad (a,b) \mapsto \left(\alpha \exp(1) \cdot \frac{\min\{a,b\}}{\max\{a,b\}} \right)_{\mathrm{cast}} \bmod m \tag{3.14}$$

die Kante e auf das Intervall $[0, m-1] \subset \mathbb{N}_0$ ab. Auch hier sei wie bei der Hashfunktion h_N der Wert $m = \mathrm{nextprime}(2^q)$ für ein geeignet gewähltes $q \in \mathbb{N}$ festgelegt. Ein Gewichtungsfaktor von $\alpha = \frac{1}{2}$ zeigt auch bei dieser Hashfunktion in der Praxis eine gute Überdeckung des Schlüsselraumes.

Für die Hashfunktion h_F wird das gleiche Abbildungsschema wie für die Hashfunktion h_E benutzt, allerdings liegen hier nun drei Konstruktionsobjekte vor. Sei $f = \mathrm{cstr}(e_1, e_2, e_3)$ mit $e_1, e_2, e_3 \in \partial P_p(E_{\Omega_p})$ eine Dreiecksfläche auf dem Partitionsrand von P_p. Mit diesen Voraussetzungen bildet die Hashfunktion

$$\begin{aligned} h_F \quad : \quad & \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \{0, \ldots, m-1\}; \\ & (a,b,c) \mapsto \left(\alpha \exp(1) \cdot \frac{\min\{a,b,c\}}{\max\{\{a,b,c\} \setminus \max\{a,b,c\}\}} \right)_{\mathrm{cast}} \bmod m \end{aligned} \tag{3.15}$$

die Dreiecksfläche f auf das Intervall $[0, m-1] \subset \mathbb{N}_0$ ab. Wiederum sei $m = \mathrm{nextprime}(2^q)$ für ein geeignet gewähltes $q \in \mathbb{N}$ und α ein Gewichtungsfaktor. Innerhalb der *padfem*^{2}-Umgebung ergibt sich mit $\alpha = \frac{1}{2}$ wie bei den Hashfunktionen h_N und h_E ein guter Parameterwert für die Schlüsselverteilung.

Die Datenstrukturen bzw. Mengen S_N, S_E und S_F verwalten nun für jedes Netzelement des Partitionsrandes ein Tupel bestehend aus dem Element selbst, dem lokalen Identifikationsmerkmal des Elementes sowie dem zugehörigen, eindeutigen Suchschlüssel. Nachdem die lokalen Elemente für die Zusammenführung der Namensräume in ihre Mengen eingefügt wurden (Zeile 9-20), beginnt die letzte Phase, in der die von den Nachbarpartitionen empfangenen Elementdaten verarbeitet werden (Zeile 21). Dazu dient die Funktion PastePartitions(), deren Struktur in Algorithmus 3.25 dargestellt ist.

Die Funktion PastePartitions() analysiert für jede Nachbarpartition $P_{p'}$ (Zeile 2) die Datentupel im empfangenen Vektor $r^{p'} = (r^N, r^E, r^F)$. Da die Daten nicht unabhängig

Algorithmus 3.25 : Algorithmus zur Aktualisierung der ω_p-Abbildung

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
S_N, S_E, S_F seien Lookup-Mengen für die lokalen Randobjekte,
$\{r^{p'}\}$ sei Vektorsystem von entfernten Randobjekte der
Nachbarpartitionen $P_{p'} \in \text{neighbor}(P_p)$

Ausgabe : konsistenter, partitionslokaler Netzanteil $\mathcal{M}_{\Omega_p}(P)$

```
1  parallel proc PastePartitions begin
2      foreach p' ∈ neighbor*(P_p) do
3          foreach (k, x, y, z) ∈ r^{p'}.r^N do
4              bestimme n ∈ N_{Ω_p} mit (h_N(x, y, z), k', n) ∈ S_N;
5              aktualisiere ω_p(n, p') und ω_p^{-1}(n, p') mittels k und k';
6              S_N ← S_N \ {(h_N(x, y, z), k', n)};
7          end
8          foreach (k, S) ∈ r^{p'}.r^E do
9              sei S = {n_1, n_2}, S' = {ω_p^{-1}(n_1, p'), ω_p^{-1}(n_2, p')};
10             bestimme e ∈ E_{Ω_p} mit (h_E(S'), k', e) ∈ S_E;
11             aktualisiere ω_p(e, p') und ω_p^{-1}(e, p') mittels k und k';
12             S_E ← S_E \ {(h_E(S'), k', e)};
13         end
14         foreach (k, S) ∈ r^{p'}.r^F do
15             sei S = {e_1, e_2, e_3}, S' = {ω_p^{-1}(e_1, p'), ω_p^{-1}(e_2, p'), ω_p^{-1}(e_3, p')};
16             bestimme f ∈ F_{Ω_p} mit (h_F(S'), k', f) ∈ S_F;
17             aktualisiere ω_p(f, p') und ω_p^{-1}(f, p') mittels k und k';
18             S_F ← S_F \ {(h_F(S'), k', f)};
19         end
20     end
21 end proc
```

voneinander sind (Kanten nutzen die Knoten als Konstruktionselement und die Flächen entsprechend die Kanten), müssen als erstes die Knotendaten verarbeitet werden. Dazu wird für jeden empfangenen Tupeldatensatz (k, x, y, z) aus dem Knotenanteil des Empfangsvektor $r^{p'}.r^N$ mit Hilfe der Schlüsselfunktion h_N und den empfangenen Koordinaten der eindeutige Schlüssel bestimmt. Dieser dient nun als Suchkriterium in der Menge S_N, um den passenden lokalen Knoten n zu finden (Zeile 4). Dabei ergibt sich auch automatisch aus dem Datentupel aus S_N das lokale Identifikationsmerkmal $k' = \omega_p(n)$, welches zusammen mit dem empfangenen Identifikationsmerkmal k für den gespiegelten Netzknoten in der Nachbarpartition die Aktualisierungsdaten für die ω_p-Abbildung und deren Umkehrabbildung darstellt (Zeile 5). Danach kann das zum Knoten n assoziierte Datentupel aus S_N entfernt werden.

Als nächstes erfolgt die Rekonstruktion des Namensraumes für die Partitionsrandkanten. Da zuvor die Namensräume der Randknoten wieder konsistent aufgebaut wurden, kann nun eine lokale Randkante e durch die empfangenen Konstruktionsknoten einer gespiegelten Kante e' aus der Nachbarpartition gefunden werden. Dazu werden mittels der ω_p-Abbildung die empfangenen Konstruktionsknoten n_1 und n_2 aus dem Datenvektoranteil für die Kanten $r^{p'}.r^E$ auf die lokalen Knoten abgebildet (Zeile 9). Mit diesen Identifikationsmerkmalen kann durch die Schlüsselfunktion h_E die zugehörige, lokale Partitionsrandkante e und ihr Identifikationsmerkmal k' gefunden werden (Zeile 10). Dadurch ist es nun möglich, die ω_p-Abbildung und ihre Umkehrabbildung für die Kanten mit Hilfe von k' und dem empfangenen k zu aktualisieren (Zeile 11). Die Entfernung des Datentupels aus der Menge bzw. Datenstruktur S_E schließt die Bearbeitung der Kante ab (Zeile 12).

Die gleiche Vorgehensweise wie für die Partitionsrandkanten wird auch für die Dreiecksflächen zwischen den Partitionen angewendet (Zeile 14-19). Hier werden allerdings die drei empfangenen Identifikationsmerkmale der Konstruktionskanten aus den Nachbarpartitionen sowie die nun konsistenten Namensräume der lokalen Randkanten verwendet. Nachdem eine Randfläche gefunden und ihre zugehörige ω_p-Abbildung aktualisiert wurde, kann das Datentupel aus S_F entfernt werden.

Sind alle Objekte aus den empfangenen Datenvektoren $r^{p'}$ der Nachbarpartitionen $P_{p'}$ bearbeitet worden, sollten die Suchdatenstrukturen S_N, S_E und S_F alle leer sein, welches gleichbedeutend ist mit der kompletten Rekonstruktion und Konsistenzeinhaltung der Namensräume. In der Praxis zeigt sich jedoch in einigen seltenen Fällen, dass dies nicht so ist. Der Grund hierfür liegt in der beschränkten Darstellungsmöglichkeit von Fließkommazahlen im Rechner. Da die Schlüsselfunktionen, d.h. die realen Hashfunktionen h_N, h_E und h_F, mit transzendenten Zahlen arbeiten und diese numerisch nicht exakt in einem Rechensystem mit nur endlicher Darstellungsmöglichkeit verarbeitet werden können, ergeben sich zwangsläufig Ungenauigkeiten bei der Berechnung von Schlüsseln für die Suchdatenstrukturen. Dies bedeutet, dass zu einem gegebenen Datum ein falscher Schlüssel berechnet werden kann. Bei Untersuchungen in der *padfem*^{2}-Simulationsumgebung hat sich herausgestellt, dass dies für einen geringen Anteil der Daten zutrifft (0.5% - 2%). Als Lösung für dieses Problem muss dann eine iterative Durchsuchung der entsprechenden Datenstruktur durchgeführt werden. Weil die Menge der gespeicherten Daten in den Suchdatenstrukturen bei jedem korrekten Auffinden reduziert wird, eignet sich diese Methode für den seltenen Fall einer fehlerhaften Objektrekonstruktion in der Praxis sehr gut.

Mit dem Abschluss der Funktion PastePartitions() endet auch die letzte der modifizierenden Funktionen für das verteilte Netz $\mathcal{M}_{\Omega_p}$ (Algorithmus 3.17, Zeile 12). Damit befindet sich nun das komplette verteilte Netz in einem global konsistenten Zustand. Die letzte Aktion des Zwei-Schritt-Algorithmus ist der Wiederaufbau des am Anfang des Verfahrens abgebauten, partitionslokalen Halosystem $\mathcal{H}_p$ mit Hilfe der Funktion ConstructHalo(). Hiermit endet auch die in Abbildung 3.5 dargestellte Finalisierungsphase.

Die Praxistauglichkeit des in diesem Kapitel vorgestellten Adaptionsalgorithmus wird anhand der Integration in die *padfem*^{2}-Simulationsumgebung in Abschnitt 4.3 mit künstlichen Testsimulationsdaten sowie in Abschnitt 4.5 an einer realen Simulationsproblemstellung demonstriert. Dabei werden neben der Skalierungseigenschaft des gesamten Verfahrens auch Performanzaspekte einzelner Phasen des Algorithmus untersucht.

3.3.2 Qualitätsgebundene Formadaption

Abschlusstetraeder der irregulären 3D-Adaption (s. Abbildung 2.4, (b)-(e)) haben die Eigenschaft, dass ihre Teiltetraeder nicht kongruent zum groben Ausgangstetraeder sind. Zusätzlich kommt noch hinzu, dass eine Verkleinerung mindestens eines Raumwinkels des Grobtetraeders eintritt, wenn die Teiltetraeder gebildet werden. Da sowohl die Form der Tetraeder als auch die Raumwinkel Einfluss auf die Konditionszahl des Systems haben, welches aus der Diskretisierung gewonnen wird (s. hierzu z.B. [BA76, Kri92]), folgt daraus zwangsweise eine Erhöhung der Konditionszahl bei Anwendung der Adaptionsregeln. Je höher eine Konditionszahl eines Systems ist, umso schwieriger ist es für einen Gleichungslöser, innerhalb einer für den Anwender akzeptablen Anzahl von Iterationen eine Lösung zu bestimmen (vgl. Abschnitt 3.2.1, [Mei05, Saa03, GL96]).

Um nun die Erhöhung der Konditionszahl zu minimieren, ist es eine sinnvolle Erweiterung des Adaptionsverfahrens, die Qualität des gesamten Netzes bzw. der einzelnen Tetraeder mit in die Entscheidung für die Auswahl der Adaptionsregeln einfließen zu lassen. Das Verfahren, das diesen Ansatz verfolgt, soll im weiteren Verlauf dieser Arbeit als *qualitätsgebundene Formadaption* bezeichnet werden, da die Form eines Tetraeders hierbei eine Rolle spielt. Die Grundidee ist, bei jeder Typbestimmung eines Tetraeders, welches hängende Objekte besitzt, die Qualität zu prüfen. Unterschreitet die Qualitätsprüfung einen bestimmten Wert, wird das Tetraeder statt mit einer der vier Abschlussregeln mit der roten Verfeinerungsstrategie regulär verfeinert.

Diese Vorgehensweise besitzt neben dem bedeutenden Vorteil der Verminderung des Qualitätsverlustes leider auch zwei entscheidene Nachteile. Zum einen werden mehr Tetraeder durch den so modifizierten Adaptionsvorgang erzeugt als eigentlich notwendig wären. Der Schwellwert für die Qualitätsmetrik beeinflusst erheblich die Anzahl der zusätzlich Tetraeder. Mehr Tetraeder bedeuten für das resultierende Gleichungssystem mehr Unbekannte. Größere Gleichungssysteme erfordern grundsätzlich mehr Iterationen zur Ermittlung einer Lösung als kleine Gleichungssysteme. Der zweite Nachteil liegt in der Qualitätsmetrik selbst. Leider gibt es nicht *die optimale Metrik*, die eine eindeutige Beurteilung für eine Diskretisierung bzw. ein Tetraeder zulässt (s. Abschnitt 2.2.1). Ein Netz bzw. ein Tetraeder, das durch eine Qualitätsmetrik Q_a einen guten Wert zugeordnet bekommt, kann bei einer Metrik Q_b ein anderes Verhalten zeigen. Geometrische Tendenzen können zwar mittels äquivalenter Qualitätsmaße nach Definition 2.22 bestimmt werden, diese können allerdings ebenfalls starken Schwankungen unterliegen.

Trotz der beschriebenen Nachteile ist es sinnvoll, diesen modifizierten Adaptionsregel-

satz anzuwenden. Da der kleinste Raumwinkel im Netz die Konditionszahl maßgeblich beeinflusst, kann durch geeignete Wahl des Schwellwertes für die Qualitätsmetrik der kleinste Winkel verbessert und die Zahl der zusätzlich erzeugten Tetraeder klein gehalten werden. Die Fragestellung, ob die eindeutige Bestimmung des Schwellwertes für eine gewählte Qualitätsmetrik in Abhängigkeit von der Diskretisierung und der Dynamik der Fehlerschätzermarkierung möglich ist, wird hier nicht weiter theoretisch verfolgt, da diese von der jeweiligen mathematischen Problemstellung abhängt.

Um nun die qualitätsgebundene Formadaption im verteilten Objektmodell umzusetzen, muss eine Modifikation der Funktion DetectClosureType() aus Algorithmus 3.13 durchgeführt werden. In Algorithmus 3.26 wird die gleiche, schon bekannte Struktur der Typbestimmung für ein Abschlusstetraeder verwendet. Der Unterschied besteht in der zusätzlichen Einfügung von Prüfstellen in Form einer Qualitätsbestimmung von Tetraedern. Dazu wird eine Funktion $Q : N_\Omega^4 \to (0, 1]$ benutzt, die vier Knoten der Diskretisierung als Argument bekommt und daraus einen Wert zwischen 0 und 1 einschließlich bestimmt, der einem normierten Qualitätsmaß entspricht (vgl. Abschnitt 2.2.1, Definition 2.12). Die Funktion $Q()$ bekommt als Parameter vier Knoten eines Tetraeders und nicht das Tetraeder selbst übergeben. Der Grund hierfür ist, dass $Q()$ meistens auf Teiltetraedern eines möglichen Abschlusstetraeders angewendet wird. Diese Teiltetraeder liegen aber zum Zeitpunkt der Qualitätsbewertung noch nicht in der Diskretisierung $\mathcal{M}_{\Omega_p}$ vor, sondern werden erst durch $Q()$ bestimmt. Daher werden die Konstruktionsknoten eines *virtuellen Teiltetraeders* für $Q()$ herangezogen. Die Knotenreihenfolge spielt hier für die Metrik keine Rolle.

Es werden im Algorithmus 3.26 an fünf Stellen Qualitätsprüfungen vorgenommen. Die erste Überprüfung erfolgt am Anfang des Algorithmus (Zeile 5). Hier wird für das Grobtetraeder v geprüft, ob dessen Qualität schon unterhalb eines Schwellwertes σ liegt. Hierdurch soll verhindert werden, dass ein Tetraeder aus der Ausgangsdiskretisierung, also schon vor der ersten Adaptionsphase, durch die Adaption eine noch schlechtere Form aufgrund der Raumwinkelverkleinerung erhält. Hält das Tetraeder v dieser Prüfung stand, erfolgt gemäß dem Ursprungsalgorithmus von DetectClosureType() die Entscheidung über die vier grünen Abschlussregeln bzgl. der Anzahl der verfeinerten Kanten. Die Verfeinerungsstrategie t wird allerdings nicht mehr alleine vom Tupel $(v, E_{ref}) \in ADS.O_h^E$ bestimmt. Stattdessen erfolgt die Festlegung von t über eine weitere Funktion CheckQuality(), die als Eingabeparameter neben dem Tupel über die hängenden Kanten von v noch zusätzlich die gewünschte Abschlussregel sowie die Metrik $Q()$ und den Schwellwert σ erhält (Zeile 8, 12, 14 und 26). Die restliche Struktur von Algorithmus 3.26 gleicht der von Algorithmus 3.13.

Algorithmus 3.27 verdeutlicht die Arbeitsweise der Funktion CheckQuality(). Die Funktion ist in zwei Phasen eingeteilt. In der ersten Phase (Zeile 2-5) werden alle notwendigen Knoten für die Qualitätsprüfung bestimmt. Dies sind zum einen die Knoten des übergebenen Grobtetraeders v (Zeile 2) sowie alle hängenden Knoten an den Grobkanten von v. Diese sind eindeutig bestimmt durch die Menge E_{ref} aus dem assoziierten Datentupel $(v, E_{ref}) \in ADS.O_h^E$ (Zeile 3). Die lokale Menge N^* speichert diese Knoten.

Algorithmus 3.26 : Erweiterter Algorithmus zur Tetraedertypbestimmung

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
$v^* \in V_{\Omega_p}$ sei zu bestimmendes Tetrader,
partitionslokale Datenstrukturen für die Adaptionsphase ADS,
$Q : N^4_\Omega \to (0,1]$ sei Qualitätsmetrik für Tetraeder,
$\sigma \in (0,1]$ sei Schwellwert für die Tetraederqualität

Ausgabe : Abschlusstyp t des Tetraeders v^*

```
1  parallel proc DetectClosureType begin
2      bestimme Tupel (v, E_ref) ∈ ADS.O_h^E : v* = v;
3      if |E_ref| > 3 ∨ ∃(e, e_1, e_2) ∈ E_ref mit e_1 oder e_2 sind "hängende" Kanten then
4          t ← "rot";
5      else if Q(R_N(v)) < σ then
6          t ← "rot";
7      else if |E_ref| = 1 then
8          t ← CheckQuality(M*_Ωp, (v, E_ref), Q, σ, "grün 1:2");
9      else if |E_ref| = 2 then
10         sei E_ref = {(e, e_1, e_2), (e', e'_1, e'_2)};
11         if R_N(e) ∩ R_N(e') ≠ ∅ then
12             t ← CheckQuality(M*_Ωp, (v, E_ref), Q, σ, "grün 1:3");
13         else
14             t ← CheckQuality(M*_Ωp, (v, E_ref), Q, σ, "grün 1:4b");
15         end
16     else
17         sei E_ref = {(e, e_1, e_2), (e', e'_1, e'_2), (e'', e''_1, e''_2)};
18         if cstr(e, e', e'') ∈ R_F(v) then
19             if cstr(e, e', e'') ∈ ∂P_p(F_Ωp) then
20                 t ← "rot";
21             else
22                 f ← cstr(R_N(e_1) ∩ R_N(e_2), R_N(e'_1) ∩ R_N(e'_2), R_N(e''_1) ∩ R_N(e''_2));
23                 if R_E(f) hat "hängende" Kanten then
24                     t ← "rot";
25                 else
26                     t ← CheckQuality(M*_Ωp, (v, E_ref), Q, σ, "grün 1:4a");
27                 end
28             end
29         end
30         t ← "rot";
31     end
32 end proc
```

Algorithmus 3.27 : Qualitätsbestimmung von Abschlusstetraedern

Eingabe : $p \in \{1, \ldots, P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}(P)$ sei der partitionslokale Anteil des verteilten Netzes,
$(\nu, E_{ref}) \in ADS.O^E_h$ sei Tupel über hängende Kanten von ν,
$Q : N^4_\Omega \to (0, 1]$ sei Qualitätsmetrik für Tetraeder,
$\sigma \in (0, 1]$ sei Schwellwert für die Tetraederqualität,
$t \in \{$"grün 1:2","grün 1:3","grün 1:4a","grün 1:4b"$\}$ sei gewünschte Verfeinerungsstrategie

Ausgabe : gewählte Verfeinerungsstrategie

```
1  parallel proc CheckQuality begin
2      N* ← R_N(ν);
3      foreach (e, e_1, e_2) ∈ E_ref do
4          N* ← N* ∪ (R_N(e_1) ∩ R_N(e_2));
5      end
6      if ∃{n_i ∈ N* | n_i ≠ n_j, 1 ≤ i, j ≤ 4} : Q(n_1, n_2, n_3, n_4) < σ then
7          return "rot";
8      end
9      return t;
10 end proc
```

Die zweite Phase im Algorithmus (Zeile 6-8) bildet nun vierelementige Teilmengen von N^*, die die Konstruktionsknoten für die Teiltetraeder des grünen Abschlusses enthalten. Auf diese Teilmengen wird die Qualitätsmetrik $Q()$ angewendet und entsprechend dem Schwellwert σ eine Entscheidung für die Verfeinerungsstrategie getroffen. Es reicht aus, wenn mindestens eine Teilmenge der Knoten aus N^* existiert, für die $Q() < \sigma$ (Zeile 6) gilt, um als Strategie für ν die reguläre Verfeinerung auszuwählen. Ansonsten gibt die Funktion CheckQuality() die als Eingabeparameter übergebene, gewünschte Strategie für den Abschluss als Funktionswert zurück (Zeile 9).

Der Zeitaufwand für Algorithmus 3.27 liegt in $\mathcal{O}(1)$. Dies liegt darin begründet, dass die erste Phase des Algorithmus im schlechtesten Fall drei hängende Kanten bearbeiten muss, d.h. die Menge N^* höchstens sieben Knoten enthalten kann. Der Fall, dass mehr als drei hängende Kanten vorliegen, wird von Algorithmus 3.26 schon zu Beginn abgefangen (Zeile 3). Unter der Voraussetzung, dass die Referenzmengenberechnung ebenfalls konstanten Aufwand benötigt (durch Vorausberechnung möglich), liegt der Zeitaufwand für die erste Phase in $\mathcal{O}(1)$. Da N^* höchstens sieben Knoten enthalten kann, existieren auch höchstens vier *sinnvolle* vierelementige Teilmengen von N^* mit Berücksichtigung der Reihenfolgenpermutationen der Knoten. Dies entspricht dem Fall des Abschlusses "grün 1:4a". Somit benötigt auch Phase zwei konstanten Zeitaufwand und die komplette Strategiebestimmung für ein Tetraeder ν mit hängenden Kanten kann durch Algorithmus 3.26 mit konstantem Aufwand durchgeführt werden.

Für die Wahl der Qualitätsmetrik $Q()$ bieten sich nun verschiedene Strategien an. $Q()$ kann eine der in Abschnitt 2.2.1 vorgestellten Standardmetriken sein. Auch eine (gewichtete) Kombination zweier oder mehrerer Standardmetriken ist eine Möglichkeit, $Q()$ zu konstruieren. Dies kann eine Verbesserung der Beurteilung eines Tetraeders ermöglichen, da es ja keine optimale Qualitätsmetrik gibt. Eine Abschluss-spezifische Unterscheidung für die Metriken ist eine weitere Möglichkeit für eine Strategie. D.h. für jeden Abschlusstyp existiert eine eigene (geeignet konstruierte) Metrik $Q_i()$ mit Schwellwert σ_i für $1 \leq i \leq 4$.

In Abschnitt 4.3.1 erfolgt eine Untersuchung der qualitätsgebundenen Formadaption unter Berücksichtigung verschiedener Qualitätsmetriken $Q()$ und (benutzerdefinierbarer) Schwellwerte anhand eines Modellproblems für die Adaption innerhalb der *padfem*^{2}-Simulationsumgebung. Dabei werden die statistischen Verteilungen der Tetraederqualitäten im gesamten Netz betrachtet und eine Analyse der Anzahl der durch den modifizierten Regelsatz zusätzlich erzeugten Tetraeder durchgeführt. Dort wird gezeigt, dass aufgrund der hohen Dynamik der Markierung durch Fehlerschätzer bzw. -indikator, bedingt durch die zu berechnende Problemstellung, sowie durch die initiale Diskretisierung des Gebietes keine analytische Bestimmung eines optimalen Schwellwertes für $Q()$ möglich ist. Bei der Bewertung des gesamten modifizierten Adaptionsalgorithmus zur irregulären Adaption spielt daher die Erfahrung mit der zu berechnenden Problemstellung eine große Rolle.

3.4 Lastverteilung und Migration

Nach einer Adaptionsphase, in der vom Fehlerschätzer bzw. -indikator Tetraeder markiert wurden, existieren Partitionen im verteilten Netz, die mehr oder weniger Tetraeder als vor der Adaption besitzen. Es ist somit ein Ungleichgewicht in der Anzahl der Tetraeder bzgl. der Partitionen entstanden. Dies bedeutet, dass die partitionslokalen Gleichungssysteme unterschiedlich viele Unbekannte besitzen. Dieser Unterschied wirkt sich gravierend auf die Laufzeit der parallelen Gleichungslöser aus, da Partitionen mit mehr Unbekannten länger für eine Iteration rechnen müssen als diejenigen, die weniger Unbekannte haben. Dadurch kann es zu Wartezeiten beim Abgleichen des Halosystems innerhalb jeder Iteration kommen, weil Kommunikationspartner unterschiedlich schnell den Datenaustauschpunkt im Algorithmus erreichen. Als Folge ergibt sich damit eine Verlängerung der Laufzeit für das Lösen des globalen Gleichungssystems und damit auch der Gesamtlaufzeit einer Simulation, da das Lösen der Gleichungssysteme u.U. einem der aufwändigsten Teile einer Simulation entspricht.

Um nun diese Laufzeitverlängerung zu vermeiden, bedarf es eines Austausches von Tetraedern zwischen den Partitionen, damit ein Gleichgewicht in der Anzahl der Tetraeder bzw. der Unbekannten im System wiederhergestellt werden kann. Dazu existieren verschiedene Verfahren (vgl. Abschnitt 2.1.4), die unter Berücksichtigung von definierten Optimierungszielen eine gleichmäßige Lastverteilung berechnen können. Die parallel ar-

beitenden Verfahren bekommen als Eingabeparameter einen verteilten Graphen. Knoten und Kanten des Eingangsgraphen können gewichtet sein. Eine Gewichtung ist u.a. dann sinnvoll, wenn zur Berechnung des Lastausgleichs nicht die Anzahl der Knoten des Teilgraphen je Partition Ausschlag gebend ist, sondern z.B. die Rechenzeit pro Knotenelement, falls diese sich unterscheiden. Eine weitere Anwendungsmöglichkeit wäre die parallele Simulation auf einem heterogenen Parallelrechensystem, bei dem die einzelnen Rechenknoten unterschiedliche Leistungsmerkmale besitzen (z.B. bzgl. Anzahl der CPUs, Speichergröße, Netzanbindung, usw.). Die Ausgabe eines parallelen Verfahrens zur Lastverteilungsberechnung ist wiederum ein Teilgraph je Partition mit der zusätzlichen Information, welcher Knoten des Teilgraphen wohin migriert wird.

Die Einzelaufgaben einer vollständigen, parallelen Lastverteilung mit einem Lastausgleichsverfahren L im spezialisierten, verteilten Objektmodell dieser Arbeit bestehen aus den folgenden drei Phasen:

- **Berechnungsphase**: Transformation des verteilten Tetraedernetzes $\mathcal{M}_{\Omega_p}$ in eine für das Lastausgleichsverfahren L geeignete Eingabe, Ausführung des Verfahrens L, Rücktransformation der Ausgabe des Verfahrens L in für das Objektmodell verwertbare Informationen,
- **Korrekturphase**: Gültigkeitsprüfung und eine eventuelle Modifikation des Ergebnisses von Verfahren L,
- **Migrationsphase**: Identifikation und Migration der betroffenen Netzelemente zu den durch das Verfahren L und der Korrekturphase vorgesehenen Zielpartitionen sowie die Integration der migrierten Netzelemente in diese.

Die notwendigen Datenstrukturen sowie die einzelnen Phasen der Lastverteilung im verteilten Objektmodell werden in den folgenden Abschnitten weiter vertieft.

Datenstrukturen für die Lastverteilung

Wie schon in Abschnitt 2.1.4 erwähnt, existieren verschiedene Verfahren zur Berechnung einer Lastverteilung. Sie unterscheiden sich in der Berechnungsweise und den Optimierungszielen sowie in der Art der Eingabedaten, um ein Gleichgewicht in der Verteilung der Knoten eines Graphen zu bestimmen. Da in dieser Arbeit nicht das Lastverteilungsverfahren selbst untersucht werden soll, beschränkt sich dieser Abschnitt auf die Ein- und Ausgabeformate für ein Lastverteilungsverfahren. Um die möglichen Formate für Ein- und Ausgabe für verschiedene Verfahren beschreiben zu können, wird nun eine Schnittmenge der Möglichkeiten definiert.

DEFINITION 3.15 (ALLG. DATENSTRUKTUR EINES LASTVERTEILUNGSVERFAHRENS)
Sei L *ein Lastverteilungsverfahren, welches durch geeignet definierte Optimierungsziele eine gleichmäßige Aufteilung (Partitionierung) für einen verteilt vorliegende Graphen* $G^P = (V, E)$ *mit* $V = \{1, \ldots, m\}$ *und* $E = \{(u, v) \mid u, v \in V)\}$

berechnet. P *bezeichne hierbei die Gesamtanzahl der Partitionen. Einer Partition* $p \in \{1,\dots,P\}$ *sei ein nichtleerer, nicht zwingend zusammenhängender Teilgraph* $G_p = (V'_p, E'_p) \subset G^P$ *zugeordnet, wobei* $V'_p = V_p \cup V^*_p$ *und* $E'_p = E_p \cup E^*_p$ *gilt mit*

$$\begin{aligned} V_p \subset V \quad & \textit{mit} \quad V_p \cap V_q = \emptyset : \forall q \in \{1,\dots,P\}, q \neq p, \\ E_p \quad &= \quad \{(u,v) \in E \mid u,v \in V_p\} \\ V^*_p \quad &= \quad \{v \in V \mid \exists (u,v) \in E : u \in V_p \wedge v \in V_q \textit{ für } p \neq q\} \\ E^*_p \quad &= \quad \{(u,v) \in E \mid u \in V_p \wedge v \in V_q \textit{ für } p \neq q\}. \end{aligned}$$

Das Tupel $L^p_{in} = L_{in}(G_p) = (c^p, d^p, e^p, w^p_n, w^p_e)$ *wird als Eingabestruktur für das Verfahren* L *bezeichnet, wenn für die Elemente des Tupels gilt*

$$\begin{aligned} c^p \quad &= \quad |V_p| \\ d^p \in \mathbb{N}_0^{c^p} \quad & \textit{mit} \quad d^p_i = \deg(v_i), v_i \in V_p \\ e^p \in \mathbb{N}_0^s, s = \sum_{i=1}^{c^p} d^p_i \quad & \textit{mit} \quad e^p_{i+j} = u_j : (v_k, u_j) \in E'_p \wedge 1 \leq j \leq d^p_k, i = \sum_{m=1}^{k-1} d^p_m, v_k \in V_p \\ w^p_n \in \mathbb{R}^{c^p} \quad & \textit{mit} \quad w^p_{n,i} \textit{ ist reelles Gewicht für Knoten } v_i \in V_p \\ w^p_e \in \mathbb{R}^s, s = \sum_{i=1}^{c^p} d^p_i \quad & \textit{mit} \quad w^p_{e,i} \textit{ ist reelles Gewicht zum Kanteneintrag } e^p_i. \end{aligned}$$

Das Tupel $L^p_{out} = L_{out}(G_p) = (c^p, s^p)$ *wird als Ausgabestruktur für das Verfahren* L *bezeichnet, wenn für die Elemente des Tupels gilt*

$$\begin{aligned} c^p \quad &= \quad |V_p| \\ s^p \in \{1,\dots,P\}^{c^p} \quad & \textit{mit} \quad s^p_i \textit{ ist Migrationsziel für Knoten } v_i \in V_p. \end{aligned}$$

Durch die Tupel L^p_{in} und L^p_{out} werden also in kompakter Weise die lokalen Ein- und Ausgabeinformationen für ein Lastverteilungsverfahren L beschrieben. Diese Tupel dienen in den nachfolgenden Abschnitten als Grundlage zur Formulierung für die durchzuführende Migration.

Da das Lastverteilungsverfahren als Eingabe einen Graphen benötigt, bei dem die Knoten durchgehend nummeriert sind, das verteilte Objektmodell jedoch eine solche Nummerierung der Tetraeder nicht erfüllt, muss eine Umsetzung der lokalen Nummerierung in die für das Verfahren benötigte erfolgen. Hierzu wird, ähnlich wie bei der Adaption, ein partitionslokaler Namensraum LDS für die Datenstrukturen der Lastverteilung und Migration eingeführt. In diesem Namensraum existieren für die notwendige Umnummerierung die Variable LDS.b und die Datenstruktur LDS.V. LDS.b speichert den Basisoffset für die Nummerierung einer Partition. LDS.V bildet eine Menge von Tupeln (v,i), die einem partitionslokalen Tetraeder $v \in V_{\Omega_p}$ eine global eindeutige Zahl $i \in \mathbb{N}$ zuordnet, wobei die Zahlenmenge global zusammenhängend, d.h. ohne Lücken ist.

Berechnungsphase

Der erste Schritt einer Lastverteilung im verteilten Objektmodell für ein Netz $\mathcal{M}_{\Omega_p}$ besteht darin, die Struktur des Netzes, d.h. die Tetraeder und ihren Zusammenhang umzuformen in den sog. dualen Volumengraphen, bei dem die Tetraeder die Knoten und der Zusammenhang der Tetraeder durch Kanten dargestellt werden. Während die Abbildung von Tetraeder auf Knoten mittels der Umnummerierung einfach durchgeführt werden kann, bestehen für den Zusammenhang des Netzes mehrere Möglichkeiten, diese im dualen Graphen abzubilden. Zwei Knoten im dualen Graphen sind genau dann durch eine Kante verbunden, wenn die zu den Knoten assoziierten Tetraeder im Netz benachbart sind. Für die Nachbarschaftsrelation bestehen nun genau drei Möglichkeiten. Wenn ein Tetraeder $v \in V_{\Omega_p}$ gegeben ist, so kann ein Tetraeder $v' \in V_{\Omega_p}$ als Nachbar bezeichnet werden, wenn es in einer der Mengen

- **Knotennachbarschaft:** $\mathcal{N}_N(v) := \mathcal{R}_V(v)$
- **Kantennachbarschaft:** $\mathcal{N}_E(v) := \{v^* \in V_{\Omega_p} \mid \mathcal{R}_E(v^*) \cap \mathcal{R}_E(v) \neq \emptyset\}$
- **Flächennachbarschaft:** $\mathcal{N}_F(v) := \{v^* \in V_{\Omega_p} \mid \mathcal{R}_F(v^*) \cap \mathcal{R}_F(v) \neq \emptyset\}$

vorhanden ist. Es gilt $\mathcal{N}_F(v) \subset \mathcal{N}_E(v) \subset \mathcal{N}_N(v)$. Die Zahl der Kanten im dualen Volumengraphen steigt damit zur größeren Menge hin an, d.h. die Kantendichte steigt. In der Praxis hat sich innerhalb der *padfem*^{2}-Simulationsumgebung gezeigt, dass die Kantendichte des dualen Volumengraphen einen Einfluss auf die Form der Partitionsoberflächen hat, abhängig davon, welches Lastverteilungsverfahren bzw. -bibliothek (z.B. METIS [KK98a], JOSTLE [Wal02], FLUX [Sch06]) eingesetzt wird. Diese Form kann wiederum die numerischen Verfahren[29] beeinflussen, die auf den Partitionsrändern bzw. dem gesamten Netz arbeiten. Aus Gründen der Übersichtlichkeit wird in den folgenden Algorithmen die Flächennachbarschaft $\mathcal{N}_F$ verwendet. Die beiden anderen Nachbarschaftsarten können äquivalent an den betreffenden Stellen angewendet werden.

Algorithmus 3.28 beschreibt die Funktion LBSetup(), die ein vorliegendes verteiltes Netz $\mathcal{M}_{\Omega_p}$ in das benötigte Tupel L_{in}^p transformiert. Der Algorithmus besteht aus mehreren Phasen. In der Initialisierungsphase (Zeile 2-4) wird die grundlegende Menge V_L bestimmt, in der alle lokalen Tetraeder einer Partition P_p enthalten sind, die nicht Teil des kopierten Halos $H_{p' \to p}$ aller Nachbarpartitionen $P_{p'}$ sind. Aus dieser Menge ergibt sich das erste Tupelelement c^p von L_{in}^p. In der folgenden Phase wird der Basisoffset für die Nummerierung der lokalen Partition durch Austauschen der lokalen c^p berechnet und in LDS.b gespeichert (Zeile 5-8). Darauf aufbauend erfolgt für jedes lokale Tetraeder aus V_L die Umnummerierung mittels Tupelbildung für LDS.V (Zeile 10). In der gleichen Schleife über V_L wird zudem das Tupelelement d^p von L_{in}^p berechnet, wobei der Knotengrad eines Tetraeders v über die Flächennachbarschaft aus $\mathcal{R}_F(v)$

[29] Zu solchen Verfahren zählen z.B. Transportberechnungen oder Partikelpositionsbestimmungen bei Charakteristiken-Verfahren. Hierbei kann eine ungünstige Partitionsform die Berechnungs- bzw. Suchdauer verlängern.

Algorithmus 3.28 : Transformation von $\mathcal{M}_{\Omega_p}$ in eine Eingabe für die Lastverteilung

Eingabe : $p \in \{1, \ldots, P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}_{\Omega_p}$ sei der partitionslokale Anteil des verteilten Netzes,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega)$,
partitionslokale Datenstrukturen für die Lastverteilungsphase LDS

Ausgabe : Eingabe $L_{in}^p = (c^p, d^p, e^p, w_n^p, w_e^p)$ für die Lastverteilung

1 **parallel proc** `LBSetup` **begin**
2 sei $L_V' \subset V_{\Omega_p}$ Menge der kopierten Halotetraeder aus $\mathcal{H}_p = \{(H_{p \to p'}, H_{p' \to p})\}$;
3 $V_L \leftarrow \{v \in V_{\Omega_p} \mid v \notin L_V'\}$;
4 $c^p \leftarrow |V_L|$;
5 sei $c \in \mathbb{Z}^P$ mit $c_i = \begin{cases} c^p & : \quad p = i \\ 0 & : \quad sonst \end{cases}$, $1 \leq i \leq P$;
6 $c \leftarrow \texttt{gatherall}_p(c)$;
7 $LDS.b \leftarrow \sum_{i=1}^{p-1} c_i$;
8 $LDS.V \leftarrow \emptyset$; $Q^p \leftarrow \emptyset$;
9 **foreach** $v_i \in V_L, 1 \leq i \leq |V_L|$ **do**
10 $LDS.V \leftarrow LDS.V \cup \{(v_i, LDS.b + i)\}$;
11 $d_i^p \leftarrow |\{f \in \mathcal{R}_F(v_i) \mid f \notin \overline{F_{\Omega_p}}\}|$;
12 $w_{n,i}^p \leftarrow \text{weight}_n(v_i)$;
13 **end**
14 $Q^p \leftarrow \emptyset$;
15 **foreach** $f \in \partial P_p(F_{\Omega_p})$ **do**
16 $Q^p \leftarrow Q^p \cup \{(\omega_{p'}(f), q) \mid f \in \partial P_{p,p'}(F_{\Omega_p}) \wedge (v, q) \in LDS.V \text{ mit } v \in \mathcal{R}_V(f)\}$;
17 **end**
18 Datenaustausch mit allen Nachbarn $P_{p'}$: $\{Q^{p'}\} \leftarrow \texttt{Exchange}(Q^p, C_p)$;
19 $i \leftarrow 1$;
20 **foreach** $v_j \in V_L, 1 \leq j \leq |V_l|$ **do**
21 **foreach** $f \in \mathcal{R}_F(v_j) : f \notin \overline{F_{\Omega_p}}$ **do**
22 **if** $f \in \partial P_p(F_{\Omega_p})$ **then**
23 $e_i^p \leftarrow q : (\omega_p(f), q) \in Q^{p'} \wedge f \in \partial P_{p,p'}(F_{\Omega_p})$;
24 **else**
25 $e_i^p \leftarrow q : (v', q) \in LDS.V \wedge v' \in \mathcal{R}_F(v_j) \wedge v' \neq v_j$;
26 **end**
27 $w_{e,i}^p \leftarrow \text{weight}_e(f)$;
28 $i \leftarrow i + 1$;
29 **end**
30 **end**
31 $L_{in}^p \leftarrow (c^p, d^p, e^p, w_n^p, w_e^p)$;
32 **return** L_{in}^p;
33 **end proc**

ohne Gebietsrand bestimmt wird. Das zugehörige Knotengewicht für w_n^p ermittelt eine Funktion $\text{weight}_n()$, die hier für die weitere Betrachtung nicht weiter erklärt werden muss (Zeile 11-12).

Nun fehlen zur Komplettierung des Tupels L_{in}^p noch die Kanteninformationen des dualen Volumengraphen und deren Gewichte. Die Bestimmung dieser Daten gestaltet sich komplizierter, da hierfür auch Kanten benötgt werden, deren Knoten in zwei verschiedenen Partitionen liegen. Da zu diesem Zeitpunkt eine Partition aber nicht die durch die Umnummerierung erzeugten neuen Bezeichner der Tetraeder auf den Nachbarpartitionsrändern kennt, muss ein Informationsaustausch zwischen Partitionsnachbarn stattfinden. Dazu erfolgt in der nächsten Phase (Zeile 14-17) für alle Partitionsranddreiecke das Aufsetzen eines Sendevektors Q^p für die Nachbarn $P_{p'}$. In diesem Sendevektor werden Tupel gespeichert, die zu einer Fläche auf dem Partitionsrand den neuen Bezeichner des lokalen Tetraeders assoziiert (Zeile 16). Nach Datenaustausch (Zeile 18) mit den Nachbarpartitionen, kann nun die Konstruktion des Tupelelementes e^p von L_{in}^p erfolgen (Zeile 19-30). Hierzu werden über die Dreiecksflächen jedes Tetraeders aus V_L die Kanten sowie das zugehörige Kantengewicht mittels einer Funktion $\text{weight}_e()$ bestimmt. Handelt es sich um eine Fläche auf dem Partitionsrand, wird der Bezeichner für den Knoten aus dem empfangene Datenvektor $Q^{p'}$ ermittelt (Zeile 23). Ansonsten ist die Kante durch zwei lokale Tetraeder bestimmt und der Bezeichner für e_i^p steht im zugehörigen Tupel in $LDS.V$ (Zeile 25). Da nach Bearbeitung der Tetraeder aus V_L alle Tupelelemente von L_{in}^p vorhanden sind, kann der Algorithmus zur Transformation des Netzes enden und das Ergebnis zurückgeben (Zeile 31-32).

Algorithmus 3.29 zeigt die Einbindung der Funktion LBSetup() in die Hauptfunktion LoadBalancing() der parallel arbeitenden Lastverteilung. Die Berechnungsphase (Zeile 2-4) führt die notwendige Transformation des verteilten Netzes (L_{in}^p) für ein Lastverteilungsverfahren durch (hier durch eine Funktion LB() symbolisiert), ruft das Verfahren auf und erhält schließlich die Rückgabedaten des Verfahrens (L_{out}^p) zurück. In der anschließenden Interpretationsphase (Zeile 5-10) muss die Rücktransformation der Ergebnisdaten von L_{out}^p durchgeführt werden, um die zusammenhängende Bezeichnungsweise der Tetraeder, die für LB() notwendig war, wieder in die für das verteilte Objektmodell benötigte lokale Namensraumabbildung zu überführen.

Hierzu dient ein Tupel $S = (S^1, \ldots, S^P)$ (Zeile 10), dessen Tupelelemente S^i eine Menge von Tetraedern bilden, die zu einer Partition P_i für den Migrationsvorgang assoziiert sind. Für jedes Tetraedertupel $(v, q) \in LDS.V$ wird nun aus dem Rückgabevektor s^p von L_{out}^p die zugehörige Migrationspartitionnummer p' bestimmt, indem von der globalen Nummer q von v wieder der Basisoffset $LDS.b$ abgezogen wird. Dadurch ergibt sich die Zielpartition für das Tetraeder v (Zeile 7) und kann dem entsprechenden Tupelelement von S zugeordnet werden. Da $LDS.V$ als Hashtabelle realisiert werden kann, kann die komplette Rücktransformation mit Zuordnung zu den Zielpartitionen in linearer Zeit berechnet werden.

Es bleibt noch die notwendige Korrektur der Migrationsinformationen (Zeile 11) durchzuführen sowie die eigentliche Migration der Tetraeder inklusive aller damit verbunde-

Algorithmus 3.29 : Lastverteilungsberechnung, Korrektur und Datenmigration

Eingabe : $p \in \{1, \ldots, P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}_{\Omega_p}$ sei der partitionslokale Anteil des verteilten Netzes,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega)$,
partitionslokale Datenstrukturen für die Lastverteilungsphase LDS,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : repartitionierter, partitionslokaler Netzanteil $\mathcal{M}_{\Omega_p}$

```
1  parallel proc LoadBalancing begin
2      L_in^p ← LBSetup(M_Ωp, C_p, LDS);
3      L_out^p ← LB(L_in^p);
4      sei L_out^p = (c^p, s^p);
5      sei S^i = ∅ für 1 ≤ i ≤ P;
6      foreach (v, q) ∈ LDS.V do
7          p' ← s^p_{q−LDS.b};
8          S^{p'} ← S^{p'} ∪ {v};
9      end
10     S ← (S^1, ..., S^P);
11     S ← LBCorrection(M_Ωp, S, LDS, ADS);
12     M_Ωp ← LBMigration(M_Ωp, S, C_p, LDS, ADS);
13     return M_Ωp;
14 end proc
```

ner Daten anzustoßen (Zeile 12). Nach der Migration und Integration der Daten ist die komplette Lastverteilung im verteilten Objektmodell abgeschlossen und der Algorithmus 3.29 endet. Die Rückgabe der Funktion LoadBalancing() ergibt ein neues, repartitioniertes und vollständig konsistentes, verteiltes Netz $\mathcal{M}_{\Omega_p}$ für jede Partition P_p. Jede dieser Partitionen P_p *formt* sich gemäß den Optimierungszielen des Lastverteilungsverfahrens L durch LB().

Korrekturphase

Eine Korrektur der von einem Lastverteilungsverfahren L ermittelten Umpartitionierung ist für das verteilte Objektmodell notwendig, da das Verfahren L keine Kenntnisse über die Art und Positionierung der Teiltetraeder von Abschlussvolumen im Netz besitzt. Es ist deshalb möglich, dass die ermittelte Lösung von L, bedingt durch das vorgegebene Optimierungsziel, ein Abschlusstetraeder aufteilt und die Teile in mehrere verschiedene Partitionen migrieren möchte. Solche aufgeteilten Teiltetraeder können allerdings nicht mehr für einen Vergröberungsprozess in einer nachfolgenden Adaptionsphase berücksichtigt werden, da dann Informationen in der Partition fehlen.

Es gibt nun drei Möglichkeiten, das Problem des Auseinanderreißens von Abschlusste-

traedern anzugehen. Die erste Möglichkeit besteht darin, die Knoten- bzw. Kantengewichte für die Teiltetraeder von Abschlüssen derart zu modifizieren, dass diese durch das Optimierungsziel des Lastverteilungsverfahrens L nicht auseinander reißen. Diese Möglichkeit kann allerdings keine Garantie hierfür geben, so dass diese Methode ausscheidet. Die zweite Möglichkeit ist, alle Teiltetraeder eines Abschlusses zu einem Knoten im dualen Volumengraphen zusammenzulegen. Diese Möglichkeit garantiert, dass die Teiltetraeder nicht in verschiedene Partitionen migrieren können. Allerdings ist der Aufwand, um den dualen Volumengraphen zu konstruieren, wesentlich höher als der benötigte Aufwand aus Algorithmus 3.28, da eine zusätzliche Abbildungsstufe von Tetraedern auf die durchgehende Nummerierung für den dualen Graphen hierdurch eingeführt wird. Dafür ist auch zusätzliche Kommunikation notwendig. Um diesen Aufwand zu sparen, wird in dieser Arbeit die dritte Möglichkeit favorisiert. Hierbei wird eine Nachbearbeitungsphase der Ergebnisdaten des Lastverteilungsverfahrens L durchgeführt, die die von einem Auseinanderreißen betroffenen Teiltetraeder eines Abschlusses erkennt und wieder zusammen legt.

Algorithmus 3.30 beschreibt das Verfahren für die Nachbearbeitungsphase, in der die Korrektur der Migrationsvorschläge L^{p}_{out} durchgeführt wird. Der Algorithmus besteht aus zwei Phasen. In der ersten Phase (Zeile 2-14) erfolgt die Detektion von Teiltetraedern aus den Migrationsvorschlägen, die in $S = (S^1, \ldots, S^P)$ gespeichert sind. Hier wird für jedes Tetraeder einer Zielpartition P_i aus S^i analysiert, ob es sich um ein Teiltetraeder eines grünen Abschlusses handelt. Dafür muss sich das Tetraeder innerhalb der Menge $ADS.V_c$ befinden, d.h. es muss ein Tupel (V, N) in $ADS.V_c$ existieren, wobei das Teiltetraeder in V vorhanden ist (Zeile 5). Existiert so ein Tupel nicht, dann handelt es sich um ein gewöhnliches Tetraeder, d.h. ein von der Adaptionsphase unberührtes Tetraeder, oder aber es ist Teil einer roten Verfeinerung, welches dann nicht weiter beachtet werden muss.

Für die Teiltetraeder eines grünen Abschlusses wird eine Hilfsmenge Q benötigt, die über alle Migrationspartitionen die Informationen für den kompletten Abschluss speichert. Q speichert Tupel der Form (q, V, R), wobei q einen eindeutigen Bezeichner für einen Abschluss symbolisiert, V alle Teiltetraeder des Abschlusses beinhaltet und R eine Menge von Migrationszielpartitionen speichert, um jedem Teiltetraeder des Abschlusses seine vom Verfahren L vorgegebene Zielpartition zuordnen zu können. Für ein Teiltetraeder ν wird nun untersucht, ob ein entsprechendes Tupel in Q vorhanden ist. Dazu wird ein eindeutiger Wert q ermittelt, der durch eine Hashfunktion $h()$ bestimmt wird. Als Schlüsselfunktion dient die Knotenmenge des groben Abschlusstetraeders $n_i \in N$ mit $(V, N) \in ADS.V_c$, deren lokale Identifikationsmerkmale $\omega_p(n_i)$ durch $h()$ geeignet miteinander verknüpft werden. Existiert ein Tupel (q, V, R) in Q zu diesem eindeutigen Hashwert $q = h(N)$ (Zeile 6), so wird das Tupelelement R um die gerade in Bearbeitung befindliche (S^i) Zielpartition P_i erweitert. Ansonsten muss ein initiales Tupel $(h(N), V, \{i\})$ angelegt und in Q eingefügt werden (Zeile 9). Das betrachtete Teiltetraeder muss als letztes noch aus der Menge S^i entfernt werden, da es womöglich seine Zielpartition in der anschließenden Analysephase wechseln kann. Die

Algorithmus 3.30 : Algorithmus zur Korrektur der Migrationsvorschläge

Eingabe : $p \in \{1, \ldots, P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}_{\Omega_p}$ sei der partitionslokale Anteil des verteilten Netzes,
$S = (S^1, \ldots, S^P), S^i \subset V_{\Omega_p}$, sei Menge der Migrationsvorschläge,
partitionslokale Datenstrukturen für die Lastverteilungsphase LDS,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : korrigiertes Tupel S der Migrationsvorschläge

```
1  parallel proc LBCorrection begin
2      Q ← ∅;
3      foreach S^i ≠ ∅, 1 ≤ i ≤ P do
4          foreach v ∈ S^i do
5              if ∃(V, N) ∈ ADS.V_c : v ∈ V then
6                  if ∃(q, V, R) ∈ Q : q = h(N) then
7                      R ← R ∪ {i};
8                  else
9                      Q ← Q ∪ {(h(N), V, {i})};
10                 end
11                 S^i ← S^i \ {v};
12             end
13         end
14     end
15     foreach (q, V, R) ∈ Q do
16         sei i ← p* : |{k ∈ R | k = p*}|_{1≤p*≤P} ist maximal;
17         S^i ← S^i ∪ V;
18     end
19     return S;
20 end proc
```

gesamte Detektionsphase kann in linearer Zeit in Abhängigkeit von der Zahl der zu migrierenden Tetraeder unter Verwendung von Hashtabellen und darauf basierenden, effizienten Such- und Einfügeoperationen ablaufen.

Nachdem alle S^i abgearbeitet wurden, erfolgt in der Analysephase die Modifikation der in der vorherigen Phase gesammelten Teiltetraederinformationen (Zeile 15-18). In dieser Phase wird für jedes Tupel (q, V, R) die Menge R betrachtet und der Partitionsindex i ausgewählt, der am häufigsten in R auftaucht (Zeile 16). Diese Strategie gewährleistet, dass die durch das Lastverteilungsverfahren L verfolgten Optimierungsziele nicht großartig verletzt werden. Steht die Zielpartition P_i fest, so werden alle Teiltetraeder, die in V gespeichert wurden, zur Menge S^i hinzugefügt. Damit migriert das gesamte Abschlusstetraeder in dieselbe Zielpartition. Auch diese Phase kann in linearer Zeit ablaufen, so dass der gesamte Korrekturalgorithmus die Migrationsvorschläge in linearer

Zeit bearbeiten kann.

Migrationsphase

Die Migrationsphase führt den eigentlichen Datenaustausch der Tetraeder zwischen den Partitionen aus, die das Lastverteilungsverfahren L bestimmt hat. Für die Migration, welche durch $S = (S^1, \ldots, S^P)$ beschrieben wird, entstehen nun mehrere Teilaufgaben. Diese sind in Algorithmus 3.31 in den einzelnen Phase verdeutlicht.

Zunächst ist ein neuer Kommunikationsschedule $\mathcal{C}^*(S)$ zu bestimmen. Dieser unterscheidet sich vom bisher verwendeten Schedule $\mathcal{C}(\mathcal{P}_\Omega)$, da durch das Verfahren L Partitionsnachbarn voneinander getrennt werden können und umgekehrt. Der lokale Anteil $C_p^* \in \mathcal{C}^*(S)$ kann aus den Daten aus S wie in Algorithmus 3.1 berechnet werden (Zeile 2). Ist C_p^* bestimmt, müssen die Mengen, die die Partitionsränder und den Halo beschreiben neu initialisiert bzw. invalidiert werden. Dies erfolgt als letzter Schritt in der Initialisierungsphase für die Migration (Zeile 3-6). Das Halosystem $\mathcal{H}_p$ der Partition P_p wird für den Migrationsvorgang nicht benötigt und wird daher abgebaut. Es wird als letzter Schritt in diesem Algorithmus, basierend auf den neu verteilten Tetraedern und den neuen Partitionsrändern, wieder aufgebaut.

In der folgenden Tetraedersendephase des Algorithmus werden zuerst Informationen über die benötigten Datenmengen (Vektor c^p), die zwischen den Kommunikationspartner auszutauschen sind, bestimmt und zu allen Partitionen gesendet (Zeile 7-8). Aus den empfangenen Längenvektoren $c^{p'}$ aller Kommunikationspartner $P_{p'}$ und dem eigenen Längenvektor c^p kann nun ein Datensystem $\{s^i\}$ (Sendevektoren) und $\{r^{p'}\}$ (Empfangsvektoren) konstruiert werden, welches die auszutauschenden Daten aufnimmt (Zeile 11-17, 19). Der Sendevektor s^i für einen Kommunikationspartner P_i (gemäß Schedule C_p^*) enthält in jedem Eintrag s_j^i, für $1 \leq j \leq c_i^p$ zu einem Migrationsvektor S^i (Zeile 12), ein (aus Speicherplatzgründen komprimiertes) Datenpaket, das sämtliche für das verteilte Objektmodell rekonstruierbaren Daten eines zu migrierenden Tetraeders enthält. Dazu zählen das Tetraeder mit seinen mathematischen Informationen selbst, die zu allen Konstruktionselementen (Knoten, Kanten, Flächen) gehörenden Informationen sowie die für die Adaption notwendigen Daten zur Vergröberung (Knoten des Grobtetraeders). Im Algorithmus (Zeile 15) wird das Zusammenfügen aller Informationen für das Datenpaket durch eine Funktion Pack() symbolisiert. Sind alle Sendevektoren $\{s^i\}$ aufgesetzt und ist für die Empfangsvektoren $\{r^{p'}\}$ entsprechend $\{c^{p'}\}$ Speicher bereitgestellt, kann der eigentliche Austausch der Tetraederdaten mit Hilfe des speziellen lokalen Kommunikationsschedules C_p^* durchgeführt werden (Zeile 20).

Nach dem Austausch der Daten folgt in der nächsten Phase die Entfernung der versendeten Tetraeder aus dem lokalen Anteil des verteilten Netzes. Dazu werden für jedes Tetraeder aus S^i mit $1 \leq i \leq P$ die kompletten Konstruktionsobjekte (Zeile 23) ermittelt und entsprechend aus $\mathcal{M}_{\Omega_p}$ entfernt. Die Operatoren $\overset{*}{\leftarrow}$ und $\ominus$ in Algorithmus 3.31 führen diese Entfernungsoperation auf allen Objekten zu einem Tetraeder ν aus. Hierbei ist zu beachten, dass auch alle mit dem Tetraeder ν assoziierten numerischen Daten

Algorithmus 3.31 : Algorithmus zur Datenmigration

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}_{\Omega_p}$ sei der partitionslokale Anteil des verteilten Netzes,
$S = (S^1,\ldots,S^P), S^i \subset V_{\Omega_p}$, sei Menge der Migrationsvorschläge,
lokaler Kommunikationsschedule $C_p \in \mathcal{C}(\mathcal{P}_\Omega)$,
partitionslokale Datenstrukturen für die Lastverteilungsphase LDS,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : repartitionierter, partitionslokaler Netzanteil $\mathcal{M}_{\Omega_p}$

1 **parallel proc** `LBMigration` **begin**
2 berechne lokale Kommunikationsstrategie $C^*_p \in \mathcal{C}^*(S)$ (s. Algorithmus 3.1);
3 **foreach** $p' \in \text{neighbor}^*(P_p)$ **do**
4 $N^*_{\Omega_{p,p'}} \leftarrow \emptyset$; $E^*_{\Omega_{p,p'}} \leftarrow \emptyset$; $F^*_{\Omega_{p,p'}} \leftarrow \emptyset$;
5 **end**
6 Abbau des Halosystems $\mathcal{H}_p$ durch `DestructHalo`(P_p);
7 **for** $i \leftarrow 1$ **to** $P, i \neq p$ **do** $c^p_i \leftarrow |S^i|$;
8 Datenaustausch mit allen Partitionen: $\{c^{p'}\} \leftarrow$ `gatherall`$_p(c^p)$;
9 **for** $i \leftarrow 1$ **to** $P, i \neq p$ **do**
10 sei s^i Sendevektor der Länge c^p_i;
11 **if** $c^p_i > 0$ **then**
12 **foreach** $v_j \in S^i, 1 \leq j \leq c^p_i$ **do**
13 $N^* \leftarrow \emptyset$;
14 **if** $\exists (V,N) \in ADS.V_c : v_j \in V$ **then** $N^* \leftarrow N$;
15 $s^i_j \leftarrow$ `Pack`$(v_j, \{\mathcal{R}_{N,E,F}(v_j)\}, N^*)$;
16 **end**
17 **end**
18 **end**
19 seien $r^{p'}$ Empfangsvektoren der Länge $c^{p'}_p, 1 \leq p' \leq P$;
20 Datenaustausch mit Migrationspartnern $P_{p'}$: $\{r^{p'}\} \leftarrow$ `Exchange`$(\{s^{p'}\}, C^*_p)$;
21 **for** $i \leftarrow 1$ **to** $P, i \neq p$ **do**
22 **foreach** $v \in S^i$ **do**
23 $\mathcal{M}^*_{\Omega_p} \overset{*}{\leftarrow} \mathcal{M}^*_{\Omega_p} \ominus (v, \{\mathcal{R}_{N,E,F}(v)\})$;
24 **if** $\exists (V,N) \in ADS.V_c : v \in V$ **then**
25 $ADS.V_c \leftarrow ADS.V_c \setminus \{(V,N)\}$
26 **end**
27 **end**
28 **end**
29 $\mathcal{M}^*_{\Omega_p} \leftarrow$ `Reconstruction`$(\mathcal{M}^*_{\Omega_p}, \{c^{p'}\}, \{r^{p'}\}, ADS)$;
30 berechne neuen lokalen Kommunikationsschedule C_p mittels `CommTopology`();
31 Aufbau des Halosystems $\mathcal{H}_p$ durch `ConstructHalo`(P_p, C_p);
32 **return** $\mathcal{M}_{\Omega_p} \leftarrow \mathcal{M}^*_{\Omega_p}$;
33 **end proc**

für die Migration bearbeitet und möglicherweise entfernt werden müssen. Handelt es sich bei dem aktuell betrachteten Tetraeder ν um ein Teiltetraeder eines Abschlusses, so muss ebenfalls die Datenbank über die grünen Abschlüsse ($ADS.V_c$) aktualisiert werden, d.h. der entsprechende Eintrag für ν muss entfernt werden (Zeile 24-26).

Die letzte Aufgabe für die Funktion LBMigration() besteht nun darin, die von den Kommunikationspartnern empfangenen Tetraederdaten in den eigenen lokalen Anteil des Netzes $\mathcal{M}^*_{\Omega_p}$ zu integrieren und daraus die neuen Partitionsgrenzen zu bestimmen. Aufgrund der Komplexität ist dieser Teil des Migrationsalgorithmus in eine eigene Funktion Reconstruction() ausgelagert (Zeile 29). Auf der Basis der neuen Partitionsgrenzen kann nun der neue Kommunikationsschedule $\mathcal{C}(\mathcal{P}^*_\Omega)$ von $\mathcal{M}^*_{\Omega_p}$ bzw. der lokale Anteil $C_p \in \mathcal{C}(\mathcal{P}^*_\Omega)$ mittels Algorithmus 3.1 ermittelt werden (Zeile 30). Steht der Kommunikationsschedule fest, erfolgt als letzte Aktion der Wiederaufbau des Halosystems $\mathcal{H}_p$ (Zeile 31). Die Rückgabe von Algorithmus 3.31 ist nun das umverteilte Netz $\mathcal{M}_{\Omega_p}$, welches gemäß den Optimierungszielen des Lastverteilungsverfahrens L repartitioniert wurde.

Algorithmus 3.32 verdeutlicht die Arbeitsweise der Funktion Reconstruction() zur Rekonstruktion der empfangenen Tetraederdaten. Die Aufgabe des Algorithmus besteht in der partitionslokalen Erzeugung und Integration der Tetraeder mit ihren Konstruktionsobjekten und numerischen Daten. Zusätzlich müssen die neuen Partitionsränder aus den rekonstruierten und integrierten Tetraedern ermittelt werden.

In der Integrationsphase (Zeile 2-12) wird jeder Empfangsvektor $r^{p^\circ *} \in \{r^{p'}\}$ eines Migrationspartners P_{p^*} analysiert. Dazu entpackt die Funktion Unpack() aus der gerade betrachteten Vektorkomponente das Tetraeder mit seinen zugehörigen Daten (Konstruktionsobjekte, numerische Daten, Abschlussknotenmenge) und integriert diese in den lokalen Anteil des verteilten Netzes (Zeile 4-5). Dies ist die zugehörige Umkehroperation zu der Pack- und Entfernungsoperation aus Algorithmus 3.31. Die extrahierte Knotenmenge N^* für grüne Abschlusstetraeder wird, wenn sie nicht leer ist, zusätzlich für die Rekonstruktion von $ADS.V_c$ verwendet. Existiert schon ein Eintragstupel (V, N^*) in der Datenbank, so wird die Menge der Teiltetraeder V für den Abschluss um das gerade zu integrierende Tetraeder ν erweitert (Zeile 6-7), ansonsten muss das Tupel initial angelegt werden (Zeile 9). Da nun neue Objekte in der Partition vorhanden sind, muss das Referenzsystem $\mathcal{R}_{\Omega_p}$ neu aufgebaut werden, d.h. die Nachbarschaftsrelationen der Netzobjekte müssen neu berechnet werden (Zeile 13).

Als nächstes gilt es, die Partitionsränder wiederherzustellen. Dazu werden die Mengen der Randobjekte $F^*_{\Omega_p}$, $E^*_{\Omega_p}$ und $N^*_{\Omega_p}$ neu aufgebaut (Zeile 14-16). Diese Mengen wurden in der Initialisierungsphase von Algorithmus 3.31 reinitialisiert. Die Vorgehensweise zur Rekonstruktion besteht darin, alle Dreiecksflächen $f \in F_{\Omega_p}$ zu bestimmen, an denen nur ein Tetraeder hängt ($|\mathcal{R}_V(f)| = 1$). Da dies auch für Gebietsrandflächen gilt, müssen diese von der Selektion ausgenommen werden. Die Menge dieser so bestimmten Dreiecksflächen bildet nun den globalen Partitionsrand ∂P_p der Partition P_p. Dementsprechend sind alle Konstruktionsobjekte (Knoten und Kanten) dieser Flächenelemente ebenfalls Partitionsrandnetzobjekte (Zeile 15). Es fehlen jetzt noch die Zuordnungen

Algorithmus 3.32 : Algorithmus zur Rekonstruktion migrierter Tetraeder

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}$ sei der partitionslokale Anteil des verteilten Netzes,
$\{c^{p'}\}$ sei Menge der Empfangsvektorgrößen,
$\{r^{p'}\}$ sei Menge der empfangenen Migrationstetraederdaten,
partitionslokale Datenstrukturen für die Adaptionsphase ADS

Ausgabe : modifizierter, partitionslokaler Netzanteil $\mathcal{M}^*_{\Omega_p}$

```
1  parallel proc Reconstruction begin
2      foreach r^{p*} ∈ {r^{p'}} : P_{p*} ist Migrationspartner von P_p do
3          for i ← 1 to c_p^{p*} do
4              (v, R_{N,E,F}(v), N*) ← Unpack(r_i^{p*});
5              M*_{Ω_p} ←* M*_{Ω_p} ⊕ (v, {R_{N,E,F}(v)});
6              if ∃(V, N) ∈ ADS.V_c : N = N* then
7                  V ← V ∪ {v};
8              else
9                  ADS.V_c ← ADS.V_c ∪ {({v}, N*)};
10             end
11         end
12     end
13     bestimme Referenzsystem R_{Ω_p} neu;
14     foreach f ∈ F_{Ω_p} : |R_V(f)| = 1 ∧ f ∉ F̄_{Ω_p} do
15         F*_{Ω_p} ← F*_{Ω_p} ∪ {f}; E*_{Ω_p} ← E*_{Ω_p} ∪ R_E(f); N*_{Ω_p} ← N*_{Ω_p} ∪ R_N(f);
16     end
17     {∂P_{p,p'}} ← ReconstructBorders(N*_{Ω_p}, E*_{Ω_p}, F*_{Ω_p});
18     return M*_{Ω_p};
19 end proc
```

der Partitionsrandobjekte zu den einzelnen Nachbarpartitionen $P_{p'}$. Dieser komplexe Zuordnungsalgorithmus ist in die Funktion ReconstructBorders() ausgelagert (Zeile 17). Die Funktion bekommt als Eingabeparameter die schon bestimmten partitionsglobalen Randobjekte $F^*_{\Omega_p}$, $E^*_{\Omega_p}$ und $N^*_{\Omega_p}$ und berechnet daraus das System der zu den Nachbarn $P_{p'}$ zugeordneten Partitionsrandobjekte $\{\partial P_{p,p'}\}$ von P_p. Der Rückgabewert von Algorithmus 3.32 ist die partitionszusammenhängende Sicht des lokalen Anteils des verteilten Netzes $\mathcal{M}^*_{\Omega_p}$.

Die Rekonstruktion der Partitionsränder zu jeder Nachbarpartition $P_{p'}$ einer Partition P_p ist in Algorithmus 3.33 dargestellt. Die Bestimmung der einzelnen Mengen $N^*_{p,p'}$, $E^*_{p,p'}$ und $F^*_{p,p'}$ über alle $P_{p'}$ erfolgt nach einem ähnlichen Prinzip wie in Algorithmus 3.24 und Algorithmus 3.25. Es werden auch hier drei Hashfunktionen h_N, h_E und h_F benötigt, die die Eigenschaften bzw. deren Konstruktionsobjekte, in einen eindeuti-

gen Hashwert umformen.

Im Algorithmus werden zuerst die Knoten auf dem Rand für jede Nachbarpartition bestimmt, danach die Kanten und schließlich die Flächen. Eine Hilfsmenge Q speichert als Hashtabelle jeweils alle lokalen Partitionsrandobjekte nebst ihren berechneten Hashwerten. Mit dem Aufbau der Hashtabelle Q erfolgt auch gleichzeitig eine Konstruktion eines Sendevektors s^p, der an alle anderen Partitionen versendet wird und die für Partition P_p lokalen Identifikationsmerkmale sowie die Konstruktionsobjekte des jeweils betrachteten Netzrandobjektes speichert. Anhand dieser Informationen aus den versendeten Vektoren können die Nachbarpartitionen $P_{p'}$, die ein Randobjekt mit P_p gemeinsam haben, ihre eigenen ω_p-Abbildungen sowie die Mengen neighbor(P_p) und neighbor*(P_p) neu aufbauen.

In der Bearbeitungsphase für die Partitionsrandknoten $N^*_{\Omega_p}$ (Zeile 2-17) wird als erster Schritt die Menge der Knoten pro Partition ermittelt und untereinander ausgetauscht (Zeile 2). Danach folgt die Initialisierung der Hashtabelle Q sowie des partitionslokalen Sendevektors s^p. Q wird mit einem Tupel (q, n) pro Knoten $n \in N^*_{\Omega_p}$ erweitert, wobei q der eindeutige Hashwert einer Funktion h_N ist, der aus den Koordinaten $(n.x, n.y, n.z)$ des Knotens n berechnet wird. Das Äquivalent zu diesem Tupel ist eine Vektorkomponente im Sendevektor s^p. Eine Komponente s^p_i besteht aus einem Tupel, deren Tupelelemente die Koordinaten $(n.x, n.y, n.z)$ und das partitionslokale Identifikationsmerkmal $\omega_p(n)$ eines Knotens $n \in N^*_{\Omega_p}$ beinhaltet (Zeile 5-6). Nachdem die lokalen Informationen versendet wurden (Zeile 8) erfolgt die Analyse der empfangenen Knotendaten von jedem Vektor r^{p^*} aus dem System $\{r^{p'}\}$, dessen Länge größer Null ist, d.h. die Partition P_{p^*} teilt mindestens einen gemeinsamen Knoten mit P_p. Kann für die Koordinaten eines Knotens aus dem Empfangsvektor r^{p^*} mittels der Hashfunktion h_N ein passendes Tupel (q, n) in Q gefunden werden (Zeile 11), so ist der gerade betrachtete Knoten n ein Randknoten der Partition P_{p^*} und kann der Menge $N^*_{\Omega_{p,p^*}}$ zugeordnet werden (Zeile 12-14).

Die Bearbeitungsphase für die Partitionsrandkanten (Zeile 18-32) verläuft in ähnlicher Weise wie für die Randknoten. Auch hier werden zunächst die Anzahl der Kantenobjekte auf dem Rand global ausgetauscht sowie die Hashtabelle Q und der Sendevektor s^p initial aufgebaut. Anstatt der Koordinaten eines Randknotens dienen hier allerdings die Konstruktionsknoten selbst als Unterscheidungsmerkmal für die Suche. Die Komponenten des Sendevektors s^p werden daher mit Tupel $(\omega_p(n_1), \omega_p(n_2), \omega_p(e))$ belegt, die die partitionslokalen Identifikationsmerkmale einer Kante $e \in E^*_{\Omega_p}$ und deren Konstruktionsknoten $\mathcal{R}_N(e) = \{n_1, n_2\}$ als Elemente enthalten (Zeile 21). Entsprechend wird die Hashtabelle Q mit Tupeln erweitert, deren Elemente einen mittels einer Hashfunktion h_E ermittelten Wert aus den Konstruktionsknoten der Kante selbst zuordnet (Zeile 22). Nach dem Datenaustausch (Zeile 24) folgt, wie bei den Randknoten, die Untersuchung auf lokal vorhandene Kantenobjekte, indem der Hashwert über die Konstruktionsknoten einer empfangenen Randkante aus dem Vektor r^{p^*} mit denen aus Q verglichen wird (Zeile 27). Dies ist deshalb möglich, weil vorher in der Bearbeitungsphase der Knoten die ω_{N_p}-Abbildung mit den Randknoten auf den aktuellsten Stand gebracht wurde.

Algorithmus 3.33 : Algorithmus zur Rekonstruktion der Partitionsgrenzen

Eingabe : $p \in \{1,\ldots,P\}$ sei die Partitionsnummer aus $P \in \mathbb{N}$ Partitionen, $P \geq 2$
$\mathcal{M}^*_{\Omega_p}$ sei der partitionslokale Anteil des verteilten Netzes,
$N^*_{\Omega_p}, E^*_{\Omega_p}, F^*_{\Omega_p}$ seien Mengen der lokalen Partitionsrandelemente

Ausgabe : Mengensystem der bestimmten Randelemente zu den Nachbarpartitionen

```
1  parallel proc ReconstructBorders begin
2      c^p ← |N*_{Ω_p}|; c ← gatherall_P(c^p);
3      Q ← ∅;
4      foreach n_i ∈ N*_{Ω_p}, 1 ≤ i ≤ c^p do
5          s_i^p ← (n_i.x, n_i.y, n_i.z, ω_p(n_i));
6          Q ← Q ∪ {(h_N(n_i.x, n_i.y, n_i.z), n_i)};
7      end
8      {r^p'} ← gatherall_P(s^p);
9      foreach r^p* ∈ {r^p'} : c_p* > 0 do
10         for i ← 1 to c_p* do
11             if ∃(q, n) ∈ Q ∧ r_i^p* = (x, y, z, k) ∧ h_N(x, y, z) = q then
12                 N*_{Ω_p,p*} ← N*_{Ω_p,p*} ∪ {n};
13                 aktualisiere ω_p(n, p*) mittels k;
14                 Partition P_p* ist nun Nachbarpartition von P_p;
15             end
16         end
17     end
18     c^p ← |E*_{Ω_p}|; c ← gatherall_P(c^p);
19     Q ← ∅;
20     foreach e_i ∈ E*_{Ω_p}, 1 ≤ i ≤ c^p, R_N(e_i) = {n_1, n_2} do
21         s_i^p ← (ω_p(n_1), ω_p(n_2), ω_p(e_i));
22         Q ← Q ∪ {(h_E(ω_p(n_1), ω_p(n_2)), e_i)};
23     end
24     {r^p'} ← gatherall_P(s^p);
25     foreach r^p* ∈ {r^p'} : c_p* > 0 do
26         for i ← 1 to c_p* do
27             if ∃(q, e) ∈ Q ∧ r_i^p* = (k_1, k_2, k) ∧ h_E(k_1, k_2) = q then
28                 E*_{Ω_p,p*} ← E*_{Ω_p,p*} ∪ {e};
29                 aktualisiere ω_p(e, p*) mittels k;
30             end
31         end
32     end
33     ... (fortgesetzt)
34 end proc
```

Algorithmus 3.33 : Algorithmus zur Rekonstruktion der Partitionsgrenzen (Forts.)

```
1  parallel proc ReconstructBorders begin
2      ...
3      c^p ← |F*_{Ω_p}|; c ← gatherall_P(c^p);
4      Q ← ∅;
5      foreach f_i ∈ F*_{Ω_p}, 1 ≤ i ≤ c^p, R_E(f_i) = {e_1, e_2, e_3} do
6          s_i^p ← (ω_p(e_1), ω_p(e_2), ω_p(e_3), ω_p(f_i));
7          Q ← Q ∪ {(h_F(ω_p(e_1), ω_p(e_2)), ω_p(e_3), f_i)};
8      end
9      {r^{p'}} ← gatherall_P(s^p);
10     foreach r^{p*} ∈ {r^{p'}} : c_{p*} > 0 do
11         for i ← 1 to c_{p*} do
12             if ∃(q, f) ∈ Q ∧ r_i^{p*} = (k_1, k_2, k_3, k) ∧ h_F(k_1, k_2, k_3) = q then
13                 F*_{Ω_{p,p*}} ← F*_{Ω_{p,p*}} ∪ {f};
14                 aktualisiere ω_p(f, p*) mittels k;
15             end
16         end
17     end
18     return {∂P_{p,p'}} = {(N*_{Ω_{p,p*}}, E*_{Ω_{p,p*}}.F*_{Ω_{p,p*}})};
19 end proc
```

Bei erfolgreichem Auffinden ist die gerade betrachtete Kante eine gemeinsame Kante zwischen den Partitionen P_p und P_{p^*} und kann der Menge $E^*_{\Omega_{p,p^*}}$ zugeordnet werden. Auch hier wird anschließend die lokale ω_p-Abbildung mit den ermittelten Werten aktualisiert (Zeile 28-29).

Die Bearbeitungsphase für die Partitionsrandflächen (Zeile 3-17, Teil 2) gleicht der der Randkanten mit dem Unterschied, dass hier die Konstruktionskanten der Dreiecksflächen für s^p und Q verwendet werden und die Menge $F^*_{\Omega_{p,p^*}}$ aufgebaut wird. Nachdem alle partitionsbezogenen Randobjektmengen $N^*_{\Omega_{p,p'}}$, $E^*_{\Omega_{p,p'}}$ und $F^*_{\Omega_{p,p'}}$ konstruiert worden sind, steht das System $\{\partial P_{p,p'}\}$ fest und kann am Ende von Algorithmus 3.33 als Rückgabeparameter übergeben werden.

4 Anwendung und Leistungsanalyse

Inhalt

Der Schwerpunkt dieses Kapitels liegt in der Untersuchung und Leistungsbewertung des verteilten Objektmodells mit seinen partitionslokalen Namensräumen in der Anwendungspraxis. Alle in dieser Arbeit vorgestellten Algorithmen und Datenstrukturen wurden in der parallelen Simulationsumgebung für FEM-Problemstellungen *padfem*^{2} praktisch umgesetzt. Die folgenden Abschnitte werden für die drei einzelnen Hauptsäulen einer parallelen Simulation (Numerik, Adaption und Lastverteilung) die zeit- und ressourcenkritischen Problemstellen innerhalb der *padfem*^{2}-Simulationsumgebung darstellen sowie ihre Eigenschaften und ihr Verhalten analysieren und gegenüberstellen. Den Abschluss dieses Kapitels bildet eine Analyse eines kompletten Simulationslaufes einer Problemstellung aus der Strömungsmechanik.

Sämtliche Messungen wurden auf einem massiv parallelen Rechensystem des *Paderborn Center for Parallel Computing (PC*^{2}*)* [www07c] durchgeführt. Dabei handelt es sich um das Cluster-System *ARMINIUS* [www07a], welches sich aus 200 Rechenknoten und 8 spezialisierten Visualisierungsknoten zusammensetzt. Jeder Rechenknoten besteht aus einem Dual Intel XEON 3.2 GHz EM64T System mit 4 GByte Hauptspeicher und 80 GByte Festplattenspeicher. Als Verbindungsnetzwerk dient ein Infiniband 4x PCI-e System. Die Peakperformance des Gesamtsystem beträgt 2.6 TFLOP/s, die LINPACK-Leistung 1.978 TFLOP/s. Die Messungen sind alle in einer Multiuser-Umgebung im *space-sharing* Betrieb durchgeführt worden. Dies bedeutet, dass Messungen, an denen Kommunikationsoperationen beteiligt sind, wegen externer Einflüsse[1] von (sporadischen) Schwankungen betroffen und nicht durch Mittelung gedämpft werden können.

4.1 Simulationsumgebung *padfem*^{2}

Im Rahmen des *Sonderforschungsbereiches SFB-376 "Massive Parallelität"* der Deutschen Forschungsgemeinschaft (DFG) an der Universität Paderborn wurde ein Werkzeug zur Simulation instationärer Navier-Stokes Problemstellungen auf der Basis der Finiten Element Methode (FEM) entwickelt. Dieses Werkzeug mit dem Namen *padfem*^{2} ist eine Weiterentwicklung des Simulationsprogrammes *PadFEM* (vgl. [DDNR96]), welches urspünglich zu Untersuchungszwecke für Lastverteilungsverfahren von dynamischen Netzwerken (Teilprojekt A3 SFB-376) implementiert wurde. Im Folgenden wird die Simulationsumgebung *padfem*^{2} mit ihren Komponenten näher vorgestellt.

4.1.1 Zielsetzung

Die Simulationsumgebung *padfem*^{2} (s. [BKM03, BKM04a, BKM04b, BM07]) ist ein Werkzeug zur Lösung von zeitlich dynamischen Problemstellungen (vgl. [BM03, BM05a, BM06]) auf der Basis des Finiten Element Ansatzes (vgl. z.B. [JL01]). Während der

[1] fremde Kommunikation zwischen Rechenknoten, die nicht Teil der Partition sind, auf denen gemessen wird

Entwicklung wurde besonders Wert auf das Design gelegt, um die Simulationsumgebung auf massiv parallelen Cluster-Systemen mit Mehrprozessorrechenknoten effizient betreiben zu können. Damit unterstützt es softwareseitig den gegenwärtigen Trend bei der Auswahl und Zusammenstellung der Hardware von Clustern hin zu Mehrkerntechnologien.

Eines der Ziele bei der Entwicklung der Simulationsumgebung war die einfache Benutzerschnittstelle für die Anwender aus den naturwissenschaftlichen Bereichen, d.h. Chemiker, Physiker, Ingenieure u.ä. Die doch recht komplexe Verwendung paralleler Systeme, angefangen von einzelnen SMP-Knoten bis hin zu großen Cluster-Systemen, sollte abstrahiert werden. Der Anwender bringt in die Simulationsumgebung lediglich sein eigenes Basiswissen in Form von spezialisierten Algorithmen ein. Diese werden unterstützt von Methoden und Basisalgorithmen aus dem *padfem*^{2}-Kern, die der Anwender nur aufzurufen braucht, um seine Problemstellung zu lösen. Über die Verwendung paralleler Verfahren, wie z.B. parallele Numerik, Partitionierung, geometrische Adaption an Partitionsgrenzen oder Lastausgleichsmechanismen bei ungleicher Datenverteilung, braucht er sich somit keine Gedanken machen.

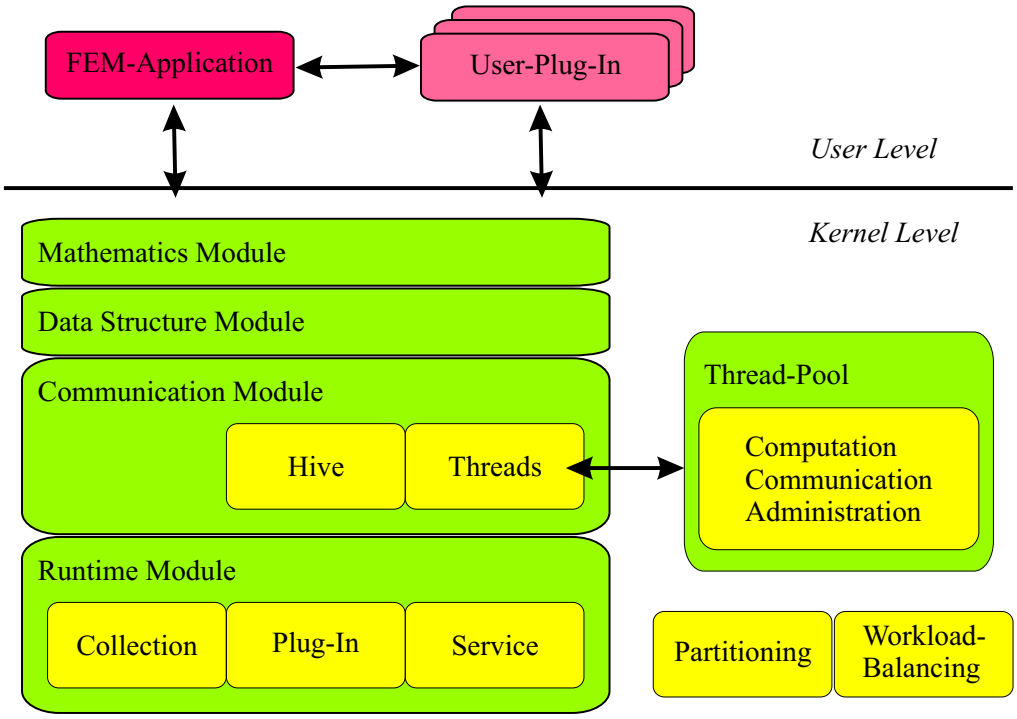
Ein weiteres Ziel beim Design war die explizite Erweiterungsfähigkeit. Parallele FEM-Simulation beinhaltet viele in sich abgeschlossene Kernmethoden, für die es unterschiedliche Verfahren und Algorithmen gibt, z.B. Ansätze zur Lösung der Navier-Stokes Gleichungen, numerische Lösungsverfahren für Gleichungssysteme oder geometrische Adaption. Diese Methoden können in *padfem*^{2} ausgetauscht oder erweitert werden. So ist es z.B. möglich, die geometrische irreguläre Standardadaption (s. [Bey98]) durch eigene Qualitätsmetrik-Funktionen zu erweitern, wie diese Arbeit zeigt (s. Abschnitt 2.1.3 bzw. Abschnitt 3.3), eine eigene geometrische Adaption auf der Basis der Bisektion oder ein spezielles Lastverteilungsverfahren zu integrieren. Die Simulationsumgebung unterstützt auch hier den Programmierer durch Basisfunktionen und Algorithmen um die Entwicklung und Lösung von Problemstellungen aus anderen Kernbereichen zu vereinfachen.

4.1.2 Architektur

Die Architektur der Simulationsumgebung *padfem*^{2} ist modular aufgebaut und besteht aus zwei Ebenen, *Kernel-Level* bzw. *User-Level* (s. Abbildung 4.1). Die Trennung in diese zwei Bereiche ist durch die verschiedenen Anwendungsmöglichkeiten begründet. Auf der einen Seite ist der Anwender, der über eine abstrahiert definierte Schnittstelle alle Bereiche benutzen kann (User-Level), die sich in der Simulation mit Parallelität oder Datenmanagement beschäftigen. Er implementiert sein Basiswissen in Module, sog. *Plug-Ins*, die vom Hauptprogramm geladen werden. Auf der anderen Seite ist der Entwickler von Kernbereichen (Datenstrukturen, parallele Grundalgorithmen), der sich auf seinen Spezialbereich im Simulationskern (Kernel-Level) konzentriert und seine Methoden und Verfahren dem Anwender zur Verfügung stellt.

Der Kernel-Level besteht aus vier Modulen, die aufeinander aufbauen. Das Grundmo-

Abbildung 4.1 Schematischer Architekturaufbau der Simulationsumgebung *padfem*^{2}



dul ist das *Laufzeitmodul (Runtime Module)*. Es stellt für die Simulationsumgebung grundlegende Funktionen und Datenstrukturen bereit. Dazu zählen im Wesentlichen abstrakte Standardtypen zum Datenmanagement *(Collections)*, ein Mechanismus zum Laden und Benutzen von Anwendungsmodulen *(Plug-Ins)* und Servicefunktionen *(Service)*, wie z.B. Speichermanagement oder Messfunktionen. Aufbauend auf diesem Modul implementiert das *Kommunikationsmodul (Communication Module)* sämtliche Funktionen zur Kommunikation, Kooperation und Synchronisation. Das Modul abstrahiert die grundlegenden Funktionen zur Kommunikation zwischen Rechenknoten des Cluster-Systems in einem *Untermodul Hive*. Dadurch kann flexibel der Unterbau der Kommunikation ausgetauscht werden, so dass auf die jeweilige Hardware optimierte Low-Level Kommunikationsbibliotheken (z.B. für Myrinet, Infiniband, Blue Gene) einsetzbar sind. Desweiteren sind in diesem Modul Algorithmen zum Thread-Management untergebracht. Einen detailierten Einblick in die Parallelverarbeitung in *padfem*^{2} liefert der nächste Abschnitt 4.1.3. Das *Datenmodul (Data Structure Module)* stellt die verteilte Grunddatenstruktur und einige Hilfsstrukturen für die Simulationsumgebung bereit. Es sorgt selbständig für eine konsistente Ordnung der verteilten Daten. Anwender nutzen dieses Modul, um auf effiziente Weise auf die einzelnen Objekte, also Knoten, Kanten, Dreiecksflächen und Tetraeder, zugreifen zu können. Dazu bietet das Datenmodul ein Iteratorkonzept, das es ermöglicht, Objekte als aufzählbare Mengen zu behandeln. Dieses Konzept vereinfacht erheblich die Formulierung von numerischen

Algorithmen. Als letztes Modul bietet das *Mathematikmodul (Mathematics Module)* grundlegende Typen, Funktionen und Verfahren für den numerischen Einsatz an. Der wesentliche Hauptteil wird von parallel arbeitenden Lösungsverfahren für Gleichungssysteme basierend auf der verteilten Datenstruktur eingenommen.

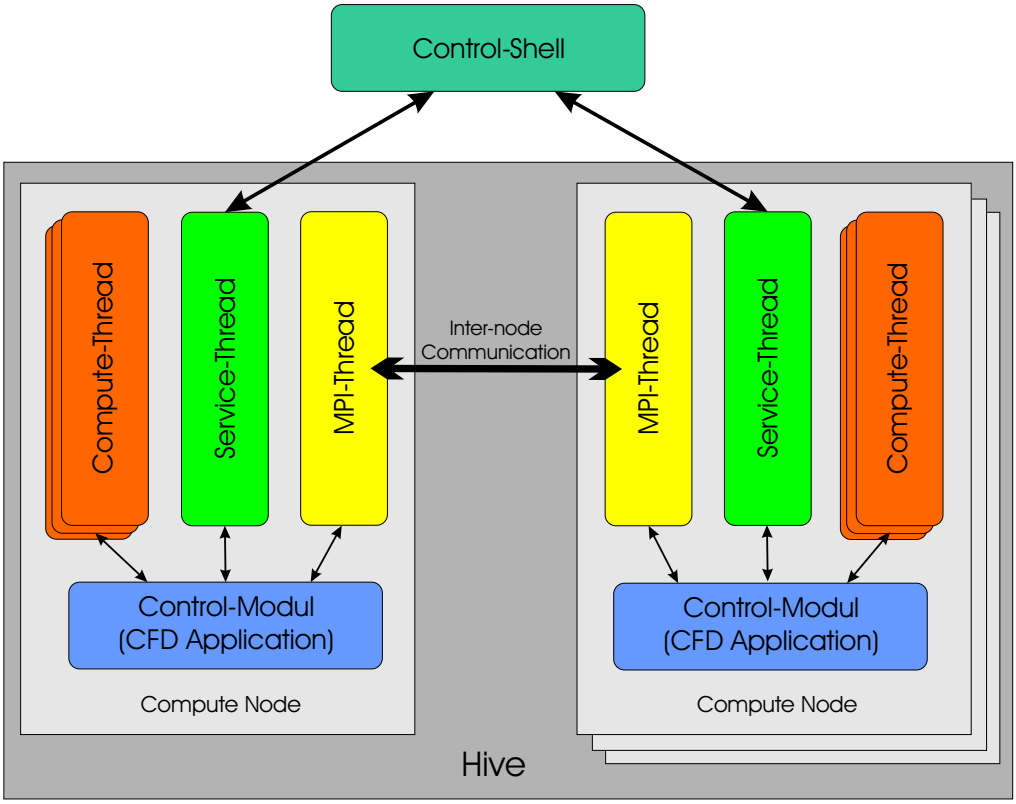
Der User-Level besteht aus einem *Hauptmodul (FEM-Application)*, welches die Anwendung definiert (Problemlöser, Partitionierer, Konsistenzprüfer, Gittermodifizierer, etc.) sowie einer Menge von Plug-Ins, die Kernbereiche für die Anwendung bereitstellen (fachspezifische Numerik, geometrische Adaption, Lastverteilungsverfahren, etc.). Die Hauptanwendung ist die Simulation bzw. Lösung instationärer strömungsmechanischer Probleme. Hierzu wird neben der Definition der Gebietsgeometrie auch eine Vorgabe (Anfangs- und Randbedingungen) für die zu berechnenden Daten benötigt. Der Anwender nutzt die in den Kernmodulen Datenstruktur und Mathematik vorgegebenen Typen und erweitert sie um die spezifischen Daten, die nicht von *padfem*^{2} angeboten werden. Der so neu definierte Typ bzw. Datencontainer wird als *Attachment* bezeichnet und kann an beliebige Objekte in der verteilten Datenstruktur des Netzes angehängt werden. Die vom Anwender gelieferten numerischen Verfahren zur Berechnung seiner Problemstellung können auf diese Attachments zugreifen, darauf arbeiten und entsprechende Lösungsdaten dort speichern. Diese Lösungsdaten werden von der Simulationsumgebung für spezifizierte Zeitschritte auf Massendatenspeichern gespeichert und können mittels geeigneter Postprocessing-Werkzeugen (z.B. [BKM04a]) weiter verarbeitet werden. Entwickler von Kernbereichen liefern ihr Know-How in bereitgestellten Plug-Ins, die vom Anwender bei Bedarf in die Hauptanwendung eingeladen und benutzt werden.

4.1.3 Parallelität

Die *padfem*^{2} Simulationsumgebung wurde unter besonderer Berücksichtigung einer effizienten Parallelverarbeitung entworfen. Massiv parallele Rechensysteme bestehen heutzutage aus einer großen Anzahl einzelner Rechenknoten, die wiederum häufig ein eigenständiges Mehrprozessorsystem darstellen. Typischerweise enthält so ein Rechenknoten zwei bis vier Prozessoren, die zusätzlich noch zwei bis vier CPU-Kerne je Prozessor enthalten. Dies ergibt zusammengefasst bis zu sechzehn physikalisch vorhandene CPU-Kerne je Rechenknoten. Durch dieses Mehrprozessor-/Mehrkernkonzept wird das so genannte *hybride Kommunikationsmodell* in *padfem*^{2} motiviert.

Das Programmiermodell in *padfem*^{2} nutzt für die Parallelverarbeitung auf einem einzelnen Rechenknoten die POSIX-Thread Schnittstelle. Hierdurch ist es möglich, einen optimierten Code für die Parallelisierung zu entwerfen. Durch automatische bzw. halbautomatische Optimierung, wie es z.B. die OpenMP-Schnittstelle (vgl. [Boa07, Cha00]) bietet, ist dies gar nicht oder nur eingeschränkt möglich. Ein höherer Aufwand bei der Entwicklung der Kooperations- und Synchronisationsoperationen ist der Preis, den man für einen handoptimierten Code bezahlen muss. In der Simulationsumgebung wird die Komplexität der Shared-Memory-Programmierung mittels Threads in internen C++-Klassen verborgen, so dass der Anwender bzw. der Entwickler von Kernmodulen mit

Abbildung 4.2 Schematischer Aufbau der Kommunikationsstruktur in *padfem*^{2}



dem Thread-Management so gut wie gar nicht in Berührung kommt.

Abbildung 4.2 zeigt den schematischen Aufbau der Kommunikationsstruktur innerhalb eines einzelnen Rechenknotens. Es existieren drei Arten von Threadtypen, die miteinander über ein Kontrollmodul kooperieren. Der *Servicethread* übernimmt Managementaufgaben innerhalb eines Rechenknotens (z.B. Speicheroperationen, Zustandsabfragen/-änderungen). Dieser kann über eine *globale Kontrollinstanz (Control-Shell)* von außen Daten empfangen oder zu dieser senden. Dies ist u.a. nützlich, wenn zu einem besonderen Zeitpunkt in der laufenden Simulation Daten abgefragt werden sollen oder ein manuelles Checkpointing[2] durchgeführt werden soll. Der zweite Threadtyp ist der *Rechen-Thread (Compute-Thread)*. Von diesem Typ existieren auf einem Rechenknoten im Normalfall[3] exakt so viele, wie Prozessorkerne vorhanden sind. Diese Threadgruppe, die aus dem internen Threadpool von *padfem*^{2} allokiert wird, steht exklusiv der

[2] Als Checkpointing wird ein Vorgang bezeichnet, der das Sichern aller relevanten Daten einer Simulation auf einem dauerhaften Speichermedium durchführt. Mit Hilfe dieser Daten kann ein Neuaufsetzen der Simulation zum abgespeicherten Simulationszeitpunkt durchgeführt werden. Dies kann bei Langzeitsimulationen nützlich sein, wenn z.B. durch äußere Einflüsse ein Rechenknoten ausfällt. Ein kompletter Neustart ist somit nicht nötig.

[3] kann vom Anwender oder automatisch bestimmt werden

numerischen Anwendung zur Verfügung. Die numerischen Gleichungslöser verwenden z.B. diesen Threadtyp. Der Kommunikationsthread ist der dritte Typ. Dieser Thread ist für die Kommunikation zwischen den einzelnen Rechenknoten zuständig. Er nutzt dazu die MPI-Schnittstelle bzw. eine abstrahierte Klassenschnittstelle, die es ermöglicht, die Low-Level-Kommunikationsschicht (z.B. Myrinet: GM, GM-2, Infiniband: VMI, etc.) auszutauschen, um eine für die verwendete Kommunikationshardware optimale Leistungsausbeute zu erzielen.

Datenpartitionen können innerhalb von *padfem*^{2} sowohl von einem einzelnen Rechenknoten des Clusters, mit möglicherweise mehreren CPU-Kernen zusammen, als auch von einem einzelnen CPU-Kern bearbeitet werden. Eine solche partitionsbearbeitende Einheit wird im globalen Parallelitätsmodell in *padfem*^{2} als *Drohne (engl. drone)* bezeichnet, die Gesamtheit aller Drohnen bildet den sog. *Bau bzw. Stock (engl. hive)*. Dieses Bienenstock-Modell wurde bewußt gewählt, da die einzelnen partitionsbearbeitenden Einheiten möglichst unabhängig, d.h. lokal auf ihren Daten arbeiten sollen. Die Struktur der Algorithmen zur Lösung eines Problems muss daher dieses Modell bei Kooperations- und Kommunikationsoperationen berücksichtigen.

4.2 Leistungsanalyse der Numerik

Dieser Abschnitt befasst sich mit der Leistungsbewertung (Benchmarking) von numerischen Aspekten der *padfem*^{2}-Simulationsumgebung. Es werden hierfür verschiedene iterative Löser für lineare Gleichungssysteme sowie der zeitkritische Abgleichalgorithmus des Halosystemes (vgl. Abschnitt 3.2.1) betrachtet[4].

4.2.1 Iterative Gleichungslöser

Problembeschreibung

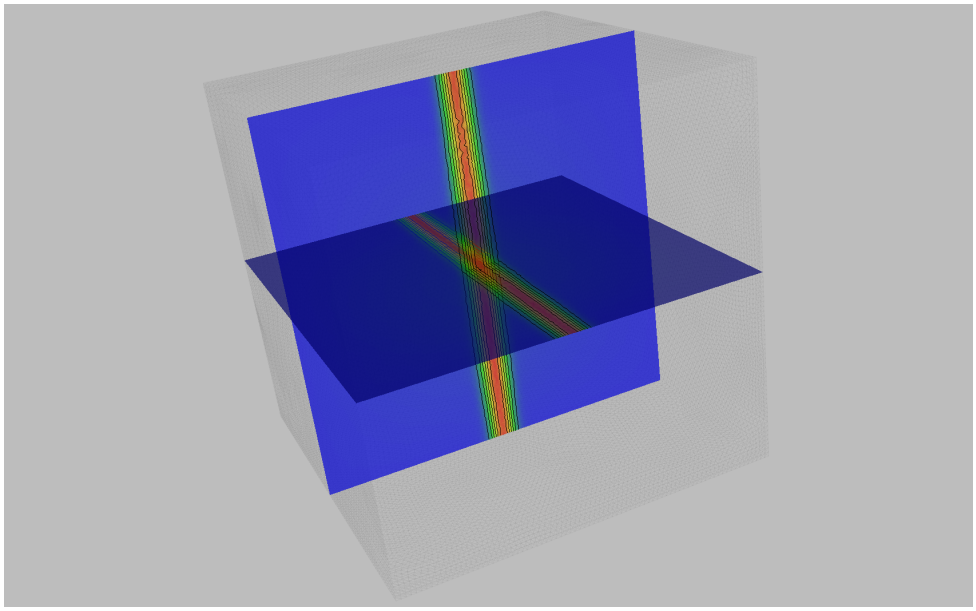
Das Modellproblem, welches als Grundlage für die Leistungsbewertung der iterativen Gleichungslöser im verteilten, spezialisierten Objektmodell herangezogen wird, ist ein Diffusionsproblem im dreidimensionalen Raum, das folgendermaßen definiert ist.

PROBLEMSTELLUNG 4.1
Gegeben sei als Simulationsgebiet der Einheitswürfel $\Omega = [0,1]^3$. *In* Ω *soll folgendes System von partitiellen Differentialgleichungen gelößt werden:*

$$\begin{aligned} -\Delta T_h &= -e^{-a(x-b)^2} \cdot ((x-b)^2 \cdot 4a^2 - 2a) \\ T_h &= T \text{ auf } \partial\Omega \\ \Omega &= [0,1]^3 \subset \mathbb{R}^3 \end{aligned} \tag{4.1}$$

[4] Weitergehende numerische Verfahren, wie z.B. FEM Matrix-Assemblierung oder Transportberechnung in der Strömungsmechanik, sind Teil spezieller numerischer Lösungsmethoden und benötigen ein umfangreiches Fachwissen für die Anwendung. Solche Verfahren sind aber nicht Kern dieser Arbeit und werden hier daher nicht weiter behandelt

Abbildung 4.3 Visualisierung zur analytischen Lösung von Problemstellung 4.1



.

Die analytische Lösung von Problemstellung 4.1 ist $T(x,y,z) = e^{-a(x-b)^2}$. Abbildung 4.3 zeigt eine grafische Darstellungen der Lösung für die Konstanten $a = 1000$ und $b = \frac{1}{2}$. In den Einheitswürfel wurden für die Visualisierung zwei (transparente) Ebenen in xy- und xz-Achsenrichtung jeweils in der Mitte des Intervallbereiches eingebracht. Die Farbgebung gibt die Verteilung der Werte wieder, wobei rot den Maximal- und blau den Minimalwert symbolisiert. Es ist so deutlich die Extremaverteilung der Lösung zu den Intervallmitten hin erkennbar.

Diskretisierung und Gebietscharakteristika

Die Diskretisierung des Simulationsgebietes für Problemstellung 4.1 wurde mit dem Tetraedierungsprogramm *TetGen* [Si04, Si07] durchgeführt. Die Qualität der Diskretisierung, d.h. die Verteilung der Qualitätswerte der Tetraeder, hat indirekt Einfluss auf die iterativen Gleichungslöser, da die Konditionszahl, der aus dem Gebiet assemblierten Matrix, durch die Winkelbedingungen mitbestimmt wird (s. hierzu [BA76, Kri92]). Aus diesem Grund ist es sinnvoll, die Verteilung der Qualitäten statistisch anzugeben (s. a. [DLGC98]).

Für die Messungen wurden, ausgehend von einem initialen Netz, drei verschiedene Größenklassen der Problemstellung verwendet. Es werden deshalb mehrere Größenklassen benutzt, weil die Datenmenge (Matrixdaten) für große Rechenknotenanzahlen sonst zu klein werden und möglicherweise nicht mehr ein reales Bild der Ressourcenausnutzung

Tabelle 4.1 Netzelementdaten für Problemstellung 4.1

Netz	Knoten	Kanten	Flächen	Tetraeder	Halozusatz
Ω_0	3472	22594	36875	17752	15.68 - 481.58 %
Ω_A	199631	1357694	2294192	1136128	3.72 - 77.98 %
Ω_B	1557325	10734092	18265792	9089024	9.23 - 35.24 %
Ω_C	12291417	85354584	145775360	72712192	11.82 - 16.65 %

(z.B. wegen Cacheeffekte, Kommunikationstotzeiten) darstellen. Die für die Rechnungen verwendeten Größenklassen bestehen aus dem Grundnetz Ω_0, welches zweimal (Ω_A), dreimal (Ω_B) bzw. viermal (Ω_C) vollverfeinert wurde, also durch Anwendung der regulären Regel auf jedes Tetraeder.

Tabelle 4.1 zeigt die Anzahl der Netzelemente für das Grundnetz und die drei Größenklassen. Die Werte beziehen sich auf ein Netz, welches im sequenziellen Fall, also unpartitioniert zu betrachten ist. Im parallelen Fall sind noch die Haloelemente und die mehrfach vorhandenen Randobjekte pro Partition hinzuzurechnen. In der vierten Spalte von Tabelle 4.1 ist daher der Prozentsatz an zusätzlichen Tetraedern für die verteilte Darstellung des Netzes in der minimalen und maximalen Rechenknotenkonfiguration angegeben[5].

Abbildung 4.4 zeigt die Verteilung der normierten Q_{ar}-Metrik des Grundnetzes Ω_0 (links) sowie der Größenklasse Ω_A (rechts). Die Verteilungen der beiden Größenklassen Ω_B und Ω_C gleichen der Klasse Ω_A (Begründung: rekursive reguläre Verfeinerung [Zha95, Bey00]) und sind daher hier nicht aufgeführt. Deutlich ist in beiden Verteilungen zu erkennen, dass die Masse der Tetraeder im Qualitätsintervall $[0.45, 0.8]$ liegt, wobei für Ω_A (und somit auch für Ω_B und Ω_C) eine schmalere Ausbreitung und damit

[5] Die minimale Rechenknotenkonfiguration ist die Rechneranzahl, in die die Datenmenge ohne Nutzung von Sekundärspeicher in den Hauptspeicher passt (Ω_0 und Ω_A 1 Knoten, Ω_B 8 Knoten, Ω_C 64 Knoten). Die maximale Rechenknotenkonfiguration beträgt bei allen Gittern 128 Knoten.

Abbildung 4.4 normierte Q_{ar}-Metrik Verteilung zu Problemstellung 4.1

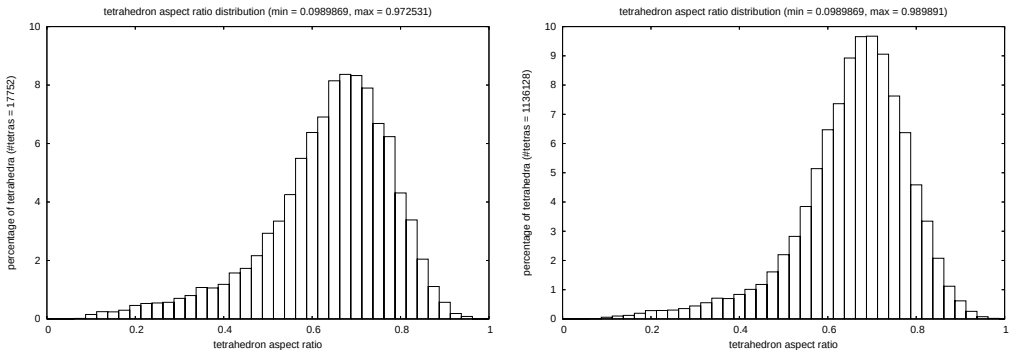


Tabelle 4.2 Normierte Verteilung verschiedener Metriken zu Problemstellung 4.1

	Ω_0				Ω_A			
Metrik	$\overline{\text{min}}$	$\bar{\mu}$	$\overline{\text{max}}$	$\bar{\sigma}$	$\overline{\text{min}}$	$\bar{\mu}$	$\overline{\text{max}}$	$\bar{\sigma}$
Q_{ar}	1.523e-1	1.0	1.496	2.208e-1	1.477e-1	1.0	1.477	1.886e-1
Q_{mr}	2.619e-1	1.0	1.253	1.649e-1	2.559e-1	1.0	1.225	1.411e-1
Q_{er}	5.768e-1	1.0	1.502	1.250e-1	1.739e-1	1.0	1.577	1.614e-1
Q_{rr}	1.363e-1	1.0	1.319	2.065e-1	4.924e-2	1.0	1.288	1.858e-1
Q_{sa}	1.123e-1	1.0	1.868	2.853e-1	1.013e-1	1.0	1.890	2.881e-1
Q_{da}	1.213e-1	1.0	1.580	2.581e-1	1.164e-1	1.0	1.517	2.199e-1
V_T	7.695e-2	1.0	3.083	3.302e-1	7.695e-2	1.0	3.083	3.302e-1

auch höhere Verteilung in diesem Intervall als Ω_0 besitzt. Daraus lässt sich schließen, dass die drei für den Benchmark verwendeten Größenklassen keine signifikanten Störeinflüsse durch die rekursive Verfeinerung erhalten haben. Dies bestätigt sich, wenn man die statistischen Kenndaten über mehrere Qualitätsmetriken betrachtet.

In Tabelle 4.2 sind die *normierten Kennwerte* Minimum ($\overline{\text{min}}$), Mittelwert ($\bar{\mu}$), Maximum ($\overline{\text{max}}$) sowie die Standardabweichung ($\bar{\sigma}$) für die in Abschnitt 2.2.2 definierten Qualitätsmetriken dargestellt. Auch hier sind die Werte nur für die Größenklassen Ω_0 und Ω_A angegeben, da sich Ω_A und Ω_B sowie Ω_A und Ω_C außer in der Anzahl der Tetraeder statistisch gesehen stark gleichen. Es handelt sich in allen Metrikfällen zwar nicht um exakt normalverteilte Datenwerte (s. Abbildung 4.4), dennoch kann hier grob die 68-95-99 Regel aus der Statistik angewendet werden (s. z.B. [OL01]): 68.3 % der Werte liegen im Intervall $[\bar{\mu}-\bar{\sigma}, \bar{\mu}+\bar{\sigma}]$, 95.5 % der Werte liegen im Intervall $[\bar{\mu}-2\bar{\sigma}, \bar{\mu}+2\bar{\sigma}]$ und 99.7 % im Intervall $[\bar{\mu}-3\bar{\sigma}, \bar{\mu}+3\bar{\sigma}]$. Bis auf Q_{er} und Q_{sa} zeigen alle Metriken eine Verkleinerung der Standardabweichung nach der Verfeinerung. Minimum- und Maximumwert verschlechtern sich nur minimal bei den wichtigen Metriken[6] Q_{ar} und Q_{mr} und stellen Ausreißer mit geringer Häufigkeit dar. Insbesondere die Kennwerte der normierten Volumenverteilung zeigen keine Veränderung. Die drei Größenklassen Ω_A, Ω_B und Ω_C bilden somit für die Messungen der iterativen Gleichungslöser eine gute Datenbasis.

Datenauswertung

Für die Messreihen wurden der SPCG-Algorithmus (scaling PCG, Algorithmus 3.9) als einfachster Vertreter der präkonditionierten CG-Verfahren und der BiCGStab-Algorithmus (bi-conjugate(d) gradient method, stabilized, Algorithmus 3.10) als Vertreter

[6] Die Metriken Aspect-Ratio und Mean-Ratio werden in der Praxis am häufigsten zur Qualitätsbestimmung genutzt, weil sie schnell und stabil zu berechnen sind bzw. die Form des Tetraeders am besten beschreiben können.

eines rechentechnisch aufwändigeren Verfahrens gewählt. Für den BiCGStab-Algorithmus gilt zusätzlich, dass, im Gegensatz zum SPCG-Algorithmus, die notwendige Anzahl an Iterationen zur Konvergenz nicht-deterministisch ist (vgl. z.B. [Saa03, GL96]). Ein Vergleich der Messreihen beider Verfahren liefert daher wegen Komplexitätsunterschied pro Iteration und Konvergenzverhalten eine gute Bewertungsgrundlage für das in dieser Arbeit entwickelte verteilte Objektmodell mit partitionslokalen Namensräumen.

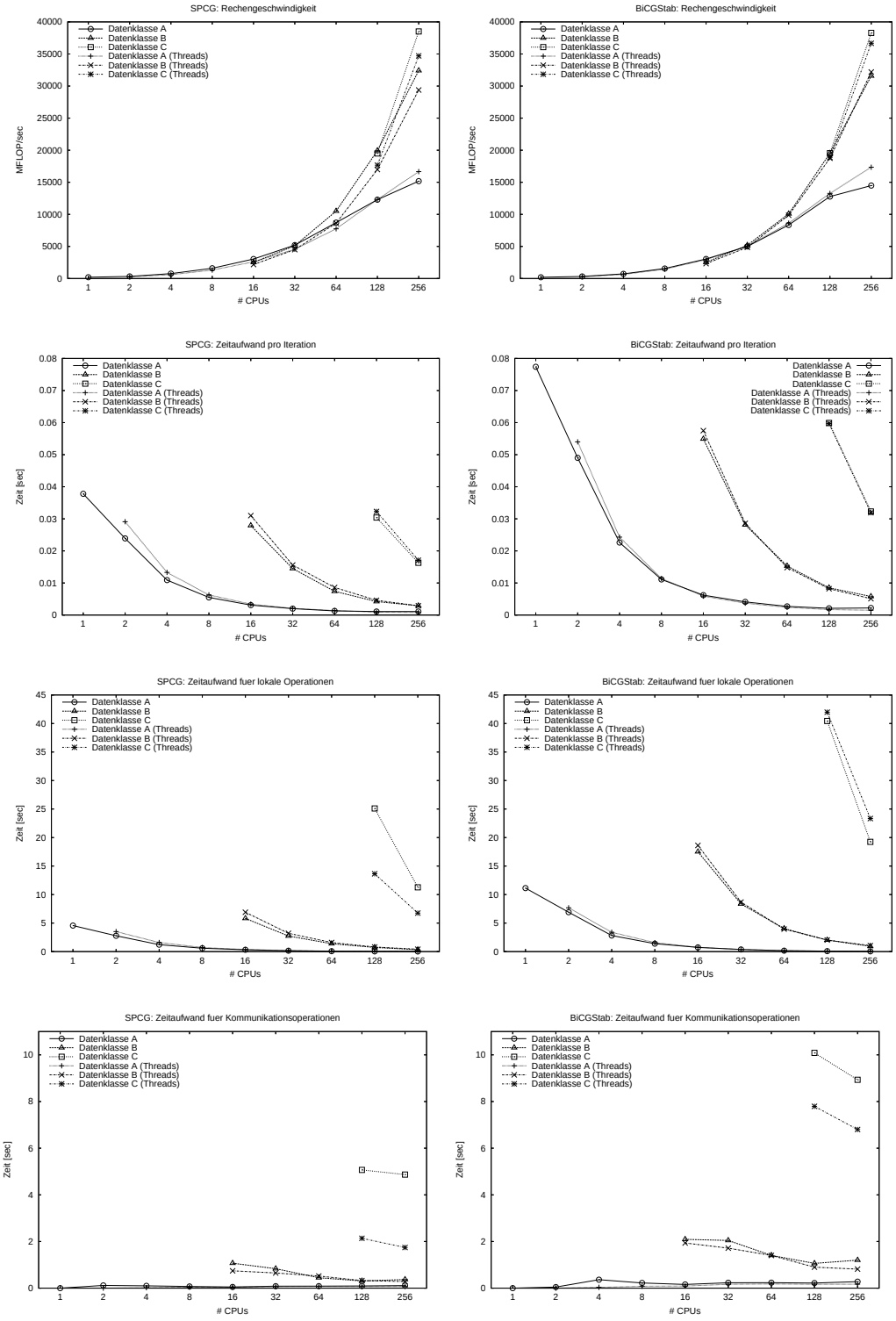
Die Schwerpunkte der Messungen bilden die Rechengeschwindigkeit zum Lösen der Gleichungssysteme, der Zeitaufwand, den eine Iteration des zu untersuchenden Verfahrens benötigt, sowie der Zeitaufwand für die lokalen Rechenoperationen und die globalen Kommunikationsoperationen. Da die Löseralgorithmen das hybride Kommunikationsmodell (MPI + Threads) im verteilten Objektmodell nutzen, wurden die drei Größenklassen Ω_A, Ω_B und Ω_C sowohl in der multi-threaded Umgebung (ein MPI-Prozess und zwei Threads pro Rechenknoten) als auch in der non-threaded Umgebung (zwei MPI-Prozesse pro Rechenknoten) betrachtet. Die Ergebnisse der Messungen sind in den Kurvendiagrammen von Abbildung 4.5 dargestellt. Die linke Spalte zeigt die Daten für den SPCG-Algorithmus, in der rechten Spalte gegenübergestellt befinden sich die Daten für den BiCGStab-Algorithmus. Alle Messwerte bilden Durchschnittswerte über 100 Messungen.

Rechengeschwindigkeit Sowohl die Kurven für den SPCG- als auch für den BiCGStab-Algorithmus zeigen ein fast ähnliches Steigungsverhalten. Wie zu erwarten steigt mit der Größe der verteilten Datenmenge und der Anzahl der verwendeten Prozessoren die erreichte Rechengeschwindigkeit. Es zeigt sich, dass das hybride Kommunikationsmodell dem reinen MPI-Modell bei beiden Algorithmen größtenteils unterlegen ist. Auffällig allerdings ist, dass bei der Verwendung der maximalen CPU-Anzahl von 256 die erreichte Rechengeschwindigkeit bei der kleinsten Größenklasse Ω_A in der multi-threaded Umgebung geringfügig höher liegt als bei der non-threaded Umgebung. Dieser Unterschied fällt beim BiCGStab-Verfahren höher aus als beim SPCG-Verfahren. Weiterhin auffällig ist die Annäherung der Kurven für beide Kommunikationsmodelle bei den beiden anderen Größenklassen Ω_B und Ω_C, wenn die Anzahl der CPUs (ab 16, 8 Rechenknoten) steigt. Auch hier zeigt der BiCGStab-Algorithmus ein besseres Leistungsverhalten gegenüber dem SPCG.

Das Kurvenverhalten lässt sich folgendermaßen interpretieren. Das hybride Kommunikationsmodell, in dem mehrere Threads für Kommunikation und Rechenarbeit zuständig sind, rentiert sich erst bei einer recht hohen Anzahl von Mehrprozessorknoten. Diese Eigenschaft ist unabhängig von der Datengröße. Für kleine Datengrößen (Ω_A) ist dies in den Kurven erkennbar, größere Datenmengen zeigen in den Tests lediglich Tendenzen hin zu diesem Verhalten. Leider standen nicht genug Ressourcen für die Messungen zur Verfügung, um dies empirisch zu zeigen. Untersuchungen ähnlicher hybrider Kommunikationsmodelle (vgl. z.B. [Nak03, Rab03, OLHB02]) bestätigen dieses Verhalten, welches sich erst bei sehr hohen CPU/Knoten-Anzahlen einstellt.

Die Diskrepanz zwischen der theoretischen Peakleistung der verwendeten Parallelrech-

Abbildung 4.5 Messwerte der iterativen Gleichungslöser SPCG und BiCGStab



nerhardware und den tatsächlich erreichten Geschwindigkeiten liegt in der komprimierten Speicherung der Matrizen und der Zugriffsgeschwindigkeiten auf diese Struktur in den Löseralgorithmen begründet. Durch die verwendete Speichermethode erfolgt der Zugriff auf die Matrix- und Vektorelemente über indirekte Adressierung (Cacheineffizienz), was eine enorme Verlustleistung für das Gesamtsystem bedeutet. Die hier erreichten Messwerte liegen bei dieser Hardwarearchitektur (CISC/RISC-Prozessoren) im durchaus üblichen Rahmen von 5-10% der Peakleistung (s. z.B. [DHBW04, Vud03]). Selbst auf Vektorrechensystemen (z.B. Earth-Simulator) wird in realen Applikationen, die hochoptimiert für die Architektur implementiert wurden, höchstens bis zu 50% der Peakleistung erreicht. Da die hier verwendeten Verfahren auf dem entwickelten, verteilten Objektmodell prototypisch umgesetzt wurden, besteht noch weiteres Optimierungspotenzial.

Zeitaufwand pro Iteration Die Zeit, welche pro Iteration benötigt wird, stellt ein gutes Maß für die Rechenkomplexität des verwendeten Verfahrens dar. Im Vergleich zwischen den beiden Algorithmen zeigt sich, dass der BiCGStab-Algorithmus nahezu doppelt so viel Zeit verwendet als der SPCG-Algorithmus. Beide Kurven haben jedoch das gleiche Steigungsverhalten, unabhängig von der zu verarbeitenden Datenmenge. Auch hier ist auffällig, dass im multi-threaded Betrieb minimal mehr Aufwand benötigt wird. Der Abstand der Kurven für beide Kommunikationsmodelle verringert sich allerdings zunehmend bei steigender Zahl von Prozessoren im System.

Zeitaufwand für lokale Rechenoperationen Betrachtet man den Zeitaufwand für das Lösen des Gleichungssystems bezogen auf den rein lokalen Rechenaufwand ohne globale Kommunikationsoperationen, so erkennt man für die Datengrößen Ω_A und Ω_B ähnliche Kurvenverhalten mit nahezu identischen Werten für die Anwendung mit oder ohne Threadbenutzung. Bei der Größenklasse Ω_C zeigt sich aber ein anderes Verhalten. Während im SPCG-Algorithmus die multi-threaded Nutzung einen deutlichen Gewinn gegenüber der non-threaded Nutzung bringt, ist das Verhalten beim BiCGStab-Algorithmus umgekehrt und mit deutlich weniger Differenz zwischen den beiden Kurven. Zudem sind beide Kurven deutlich steiler, d.h. die Skalierung ist hier besser. Die Vertauschung der beiden Kurven liegt begründet in der erhöhten Häufigkeit der lokal arbeitenden numerischen Funktionen beim BiCGStab-Verfahren, die die Ressourcen des Speichersystems auslasten (Matrix- und Vektoroperationen) sowie den vermehrt auftretenden Synchronisationspunkten bei der Threadbenutzung (Skalarprodukt).

Zeitaufwand für Kommunikationsoperationen Die Kurven für den Zeitaufwand der rechenknoten-übergreifenden Kommunikation für das Lösen eines Gleichungssystems sind für die Größenklassen Ω_A und Ω_B bei beiden Algorithmen mit beiden Kommunikationsmodellen ähnlich. Sie zeigen eine leichte Tendenz zu Geraden. Dies bedeutet eine sehr gute Skalierung für die Kommunikation inklusive der Halo-Aktualisierung, die im BiCGStab-Algorithmus sogar zweimal pro Iteration benötigt wird. Einen echten Gewinn zeigt die multi-threaded Umgebung bei der Größenklasse Ω_C. Mehr als eine Halbierung des Kommunikationszeitaufwandes bei Verdoppelung der CPU-Anzahl ist beim SPCG-Verfahren

zu erkennen. Das BiCGStab-Verfahren erreicht hier wegen der erhöhten Häufigkeit der Kommunikationsoperationen (Halo-Aktualisierung, Skalarprodukt) bei der Threadnutzung zwar keine Halbierung aber dennoch eine deutlich messbare Einsparung im Zeitverbrauch.

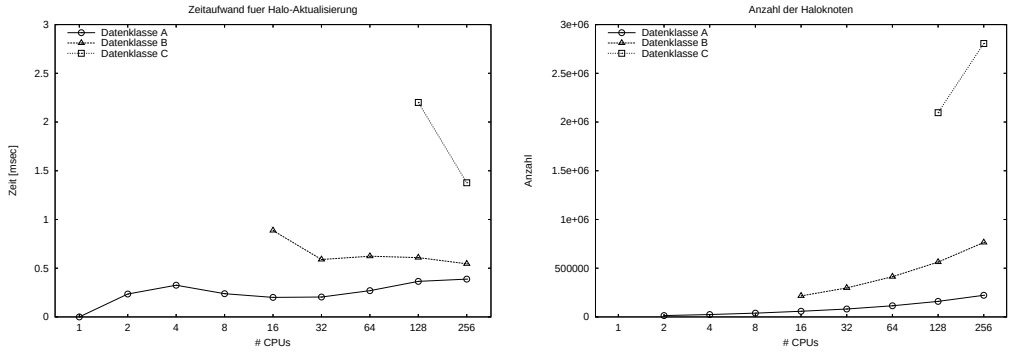
4.2.2 Haloabgleich

Die Aktualisierung des Halos mit Hilfe eines Abgleichers ist ein äußerst zeitkritischer Prozess. Die Aktualisierung erfolgt in jeder Iteration eines Löseralgorithmus mindestens einmal (BiCGStab benötigt z.B. zwei Aktualisierungen). Da Kommunikationsoperationen grundsätzlich langsamer sind als lokale Rechenoperationen, bildet die Halo-Aktualisierung zusammen mit den anderen notwendigen Kommunikationsoperationen für das Skalarprodukt den Leistungsengpass. Eine effiziente Umsetzung des Aktualisierungsvorganges ist daher entscheidend für die Rechengeschwindigkeit der iterativen Gleichungslöser.

Für die in diesem Abschnitt vorgestellten Messungen dienen als Grundlage Algorithmus 3.2 und Algorithmus 3.7, die die für einen numerischen Gleichungslöser notwendigen Daten aus dem spezialisierten, verteilten Objektmodell ermitteln, transformieren und diesen zur Verfügung stellen. Als Problemstellung wird wie in Abschnitt 4.2.1 die Problemstellung 4.1 auf den gleichen Datenklassen Ω_A, Ω_B und Ω_C angewendet. Gemessen wird der Zeitaufwand, der benötigt wird, um *eine komplette Aktualisierung des Halos* innerhalb einer Löseriteration durchzuführen. Dies umfasst die komplexe Transformation der numerischen Haloobjektdaten in die Sendevektoren für die jeweiligen Partitionsnachbarn, den eigentlichen Datenaustausch mittels des vorherbestimmten Kommunikationsschedules sowie die Rücktransformation der empfangenen Haloobjektdaten in die Löserstruktur. Der Aufbau der verteilten Löserstruktur selbst ist nicht Teil der Messung, da diese nur zu Beginn einer Lösungsberechnung (bei der Transformation des verteilten Objektmodells in die komprimierte CRS-Form) durchgeführt wird und von da an als konstant über alle Iterationen zu betrachten ist. Als Grundlage wurden wiederum 100 Messungen durchgeführt und daraus ein Mittelwert berechnet. Eine Unterscheidung nach non-threaded oder multi-threaded Umgebung wie in Abschnitt 4.2.1 muss in diesem Testumfeld nicht gemacht werden, da der Aktualisierungsvorgang des Halos im Gegensatz zu den restlichen numerischen Operationen innerhalb der Löseriteration (Datenabhängigkeit) von einem einzelnen Thread ausgeführt wird.

Abbildung 4.6 zeigt auf der linken Seite die Messkurven für die Halo-Aktualisierung innerhalb einer Löseriteration über mehrere Prozessoranzahlen. Die Laufzeit des Aktualisierungsvorganges ist (linear) abhängig von der Anzahl der Objekte, die im Halo betrachtet werden müssen. Daher ist auf der rechten Seite in Abbildung 4.6 zum Vergleich die Größenmenge der Knoten im Halo bezogen auf alle Partitionen angegeben. Es handelt sich hierbei um die im gesamten Netz mehrfach vorkommenden Knoten, die wegfallen, wenn das Netz sequenziell betrachtet wird. Aus dem Kurvenverlauf wird deutlich, dass die Zahl der Haloknoten mit Zunahme der Prozessoranzahlen stark steigt

Abbildung 4.6 Haloknotenanzahl und Aktualisierungszeiten



(vgl. auch Tabelle 4.1). Trotz dieser Zunahme an Knoten bleibt der Zeitaufwand, der zur Aktualisierung notwendig ist, *quasi-konstant*, d.h. die Kurve bewegt sich um eine Asymptote bei steigender Zahl an verwendeten Prozessoren. Dies ist an den Kurven für die Datengrößenklasse Ω_A und Ω_B deutlich erkennbar (Asymptote bei 0.5 msec). Für Ω_C zeigt sich lediglich eine starke Zeitersparnis, ein asymptotisches Verhalten kann aufgrund der Datengröße und der beschränkten Anzahl an zur Verfügung stehenden Prozessoren leider nicht ermittelt werden.

4.3 Leistungsanalyse der Adaption

Die Adaption gehört neben der Lastverteilung zu den zentralen geometrischen Netzmodifikationsverfahren und hat somit starken Einfluss auf die numerischen Teile einer Simulation (z.B. auf die Anzahl der Unbekannten der zu lösenden Gleichungssysteme). Die Geschwindigkeit der Adaptionsphase wird im verteilten bzw. parallelen Fall durch Kommunikations- und Synchronisationsoperationen sowie der Datenkonsistenzhaltung erheblich beeinflusst, so dass die theoretische Peakleistung (linearer Speedup) nicht erreicht werden kann. D.h. trotz Fokusierung auf eine Maximierung der lokalen Operationsanzahl (vgl. Abschnitt 3.3) bildet die Kommunikation den Flaschenhals für die Gesamtleistung.

Messungen für die Adaption bei realen Anwendungen unterliegen einer Dynamik. Je nachdem, wieviel Tetraeder vom Fehlerschätzer bzw. -indikator markiert werden, d.h. die Genauigkeit der numerischen Lösung bei zeitlicher Dynamik schwankt, hat der Adaptionsalgorithmus eine variable Menge an Daten zu verarbeiten. Hinzu kommt, dass die Häufigkeit der Tetraedermarkierungen nicht gleichmäßig über alle Partitionen verteilt sein müssen, so dass mit einem Ungleichgewicht der Datenmenge gerechnet werden kann (Lokalität der Adaption). Dies alles macht es schwer, einigermaßen aussagekräftige Bewertungen über die Skalierung, der relativen Geschwindigkeit und der Leistungsfähigkeit der parallelen Adaption zu erhalten. Daher wird für die Messung der in dieser

Arbeit entwickelten parallelen Adaption auf Grundlage des verteilten, spezialisierten Objektmodells (s. Abschnitt 3.1) ein Benchmark definiert und angewendet, der sowohl die Algorithmen der Adaption als auch der Lastverteilung (vgl. Abschnitt 4.4) bzgl. der oben genannten Kriterien besonders fordert.

Problembeschreibung

Als Problemstellung soll ein Simulationsbenchmark dienen, der ohne einen numerischen Teil auskommt. Dadurch verkürzt sich die gesamte Laufzeit der Simulation und numerische Ungenauigkeiten, die zwangsläufig beim Lösen von Gleichungssystemen auftreten, können den hier benutzten *geometrischen Fehlerindikators* nicht beeinflussen.

PROBLEMSTELLUNG 4.2
Gegeben sei ein quaderförmiger Kanal (Ω bzw. Ω_h) mit den Ausmaßen $1 \times 1 \times 10$. *Im Kanal bewegt sich eine Kugel* K *mit Radius* $R_K = 7.5 \cdot 10^{-2}$ *von Position* $(0.15, 0.5, 0)^T$ *nach* $(0.15, 0.5, 10)^T$. *Die Schrittweite* S_K, *der den Mittelpunkt* M_K *von* K *in* z*-Richtung bewegt, betrage hierbei* 10^{-1}. *Der Mittelpunkt* M_K *bewegt sich zusätzlich in der* xy*-Ebene in einer Kreisbahn um den Mittelpunkt* $(0.5, 0.5)^T$ *des Querschnittes der* xy*-Ebene.*

Es sollen alle Tetraeder in Ω_h *verfeinert werden, die von der Oberfläche der Kugel* K *im Laufe der Simulation geschnitten werden. Unterschreitet ein so ausgewähltes Tetraeder ein Volumen von* $V_T = 10^{-7}$, *so wird es nicht weiter verfeinert.*

Die Kugel bewegt sich also auf einer Helixbahn durch den Kanal und schiebt in Bewegungsrichtung eine Adaptionsfront vor sich her. Dies bedeutet, das in einem Zeitschritt kontinuierlich Tetraeder für die Verfeinerung markiert werden. Durch die stetige Erhöhung der Anzahl von Tetraedern im Simulationsgebiet, die zudem noch streng lokal an der zu approximierenden Kugeloberfläche positioniert sind, bedeutet dies für die Lastverteilung ebenfalls einen hohen Aufwand in der Berechnung, da sich die Partitionsgrenzen an der Adaptionsfront der Kugel konzentrieren. Mit der Adaptionsfront bewegen sich also auch die Grenzen der Partitionen mit und erzeugen somit ein hohes Datenaufkommen für die Migrationsphase.

Damit sich die Kugeloberfläche in dem zu Beginn groben Gebiet ausprägen kann, wird am Anfang der Simulation ein Zeitraum von 8 Schritten zusätzlich zu den 100 Schritten eingefügt, in dem sich der Mittelpunkt der Kugel nicht bewegt. Diese 8 Schritte reichen aus, um die Oberfläche durch die Adaption approximativ anzunähern, so dass alle Tetraeder, die von der Kugeloberfläche geschnitten werden, ein Volumen $V_T < 10^{-7}$ besitzen, wie es in der Problemstellung 4.2 gefordert ist.

Die Lastverteilungsberechnung für diesen Benchmark, wurde mit der Bibliothek JOSTLE (vgl. [Wal02]) durchgeführt. Die Bibliothek METIS (s. [KK98a]) liefert für den Benchmark vergleichbare Werte. Da für diesen Benchmark auch keine numerischen Algorithmen benötigt werden, ist eine Qualitätsbetrachtung der Diskretisierung wie für Problemstellung 4.1 nicht nötig.

Datenauswertung

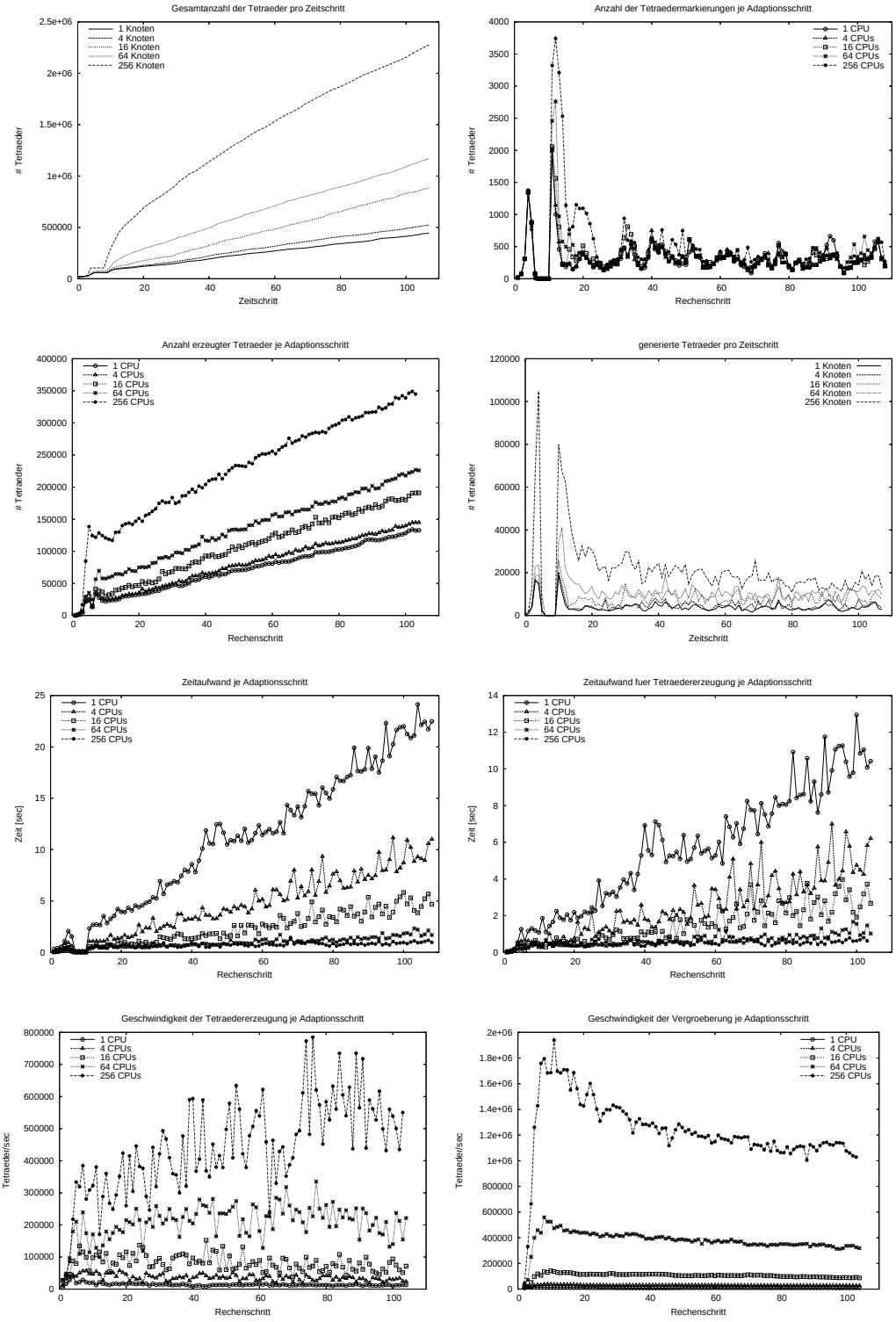
Abbildung 4.7 zeigt die Ergebnisse als Kurvendiagramme über verschiedene Aspekte der Adaptionsphase. Angegeben sind die Messwerte für den sequenziellen Lauf sowie für 4, 16, 64 und 256 Prozessoren. Es ist zu beachten, dass in einigen Diagrammen weniger als die gerechneten 108 Zeitschritte aufgezeigt sind. Dies liegt daran, dass in der "Einschwingphase", also den ersten 8 Schritten der Simulation, teilweise keine Tetraeder durch die Adaption erzeugt werden, weil die Volumengrößenbedingung aus Problemstellung 4.2 früher erfüllt ist (s. z.B. Zeitaufwand für Tetraedererzeugung). Eine Messkurve über den Gesamtzeitaufwand, der pro Simualtionsschritt benötigt wird, ist in Abbildung 4.11 in der linken Spalte oben zu finden.

Gesamtanzahl der Tetraeder Im oberen linken Diagramm von Abbildung 4.7 ist der Verlauf über die Anzahl der Tetraeder im gesamten (sequenziell) betrachteten Netz dargestellt. Hierbei fällt sofort auf, dass die Gesamtanzahl der Tetraeder mit steigender Zahl der verwendeten Prozessoren stark zunimmt. Die Begründung hierfür liegt in der speziellen Anwendung der regulären (roten) Verfeinerungsstrategie an den Partitionsgrenzen (s. Algorithmus 3.13, Zeile 17-26). Je mehr Partitionsgrenzen im Netz vorhanden sind, umso häufiger tritt dieser Fall der grenzüberschreitenden regulären Adaptionsregel ein. Da sich in diesem besonderen Benchmark die Grenzen der Partitionen kontinuierlich mit der Adaptionsfront der Kugeloberfläche bewegen und die Auflösung des Dominoeffekts in diesen Grenzbereichen in der Umgebung Tetraeder zusätzlich regulär verfeinert, erklärt dies die immer stärker werdende Anzahl der Tetraeder im gesamten Netz bei steigender Zahl der Prozessoren.

Anzahl der Tetraedermarkierungen Die Kurven für die Anzahl der vom geometrischen Fehlerindikator markierten Tetraeder (Abbildung 4.7, rechte Spalte, oben) zeigen kurz nach der Einschwingphase ein ähnliches, um eine konstante Zahl schwankendes, Verhalten. Zwischen 200 und 800 markierte Tetraeder sind ab Schritt 20 erkennbar. Die Kurven für 64 und 256 Prozessoren zeigen unmittelbar nach der Einschwingphase eine höhere Markierungszahl als die anderen, gleichen sich aber ebenfalls ab Schritt 20-25 den anderen Kurven an. Zu sehen ist dieses Verhalten auch in den Kurven für die Gesamtanzahl der Tetraeder, wo die Kurven für 64 und 256 Prozessoren nach dem Einschwingen eine starke Steigung zeigen, diese aber ab Schritt 20 abgeschwächt verläuft.

Gesamtanzahl erzeugter Tetraeder Die Gesamtanzahl der erzeugten Tetraeder je Adaptionsschritt (Abbildung 4.7, linke Spalte, 2. Diagramm von oben) korrelliert mit der Gesamtzahl der Tetraeder. Auch hier ist mit steigender Zahl der Prozessoren eine immer höher werdende Zahl an erzeugten Tetraedern sichtbar. Die Werte setzen sich aus der Zahl der zum urspünglichen Netz hinzugekommenen Tetraedern und der Zahl der durch den Vergröberungsprozess (s. Algorithmus 3.15) entstandenen und wieder verfeinerten Tetraedern zusammen. Da die Zahl der Tetraeder, die Abschlüsse bilden und vergröbert werden müssen, im Verlauf der Simulation stetig zunimmt, ist in den Kurven

Abbildung 4.7 Laufzeitcharakteristik der Adaption für Problemstellung 4.2



eine konstante Steigung für alle Anzahlen von Prozessoren nach der Einschwingphase erkennbar.

Anzahl neu erzeugter Tetraeder Betrachtet man nur die Zahl der zum Netz neu hinzukommenden Tetraeder (Abbildung 4.7, rechte Spalte, 2. Diagramm von oben), also nur regulär verfeinerte Tetraeder, so zeigt sich ein völlig anderes Bild. Die Kurven ähneln dem Verhalten der Kurven für die Zahlen der Tetraedermarkierungen. Für große Prozessorzahlen zeigen sich vor und kurz nach der Einschwingphase erhöhte Werte an Tetraedern, diese verringern sich aber im Laufe der Simulation und nähern sich den Kurven für niedrige Prozessorzahlen an. Diese Kurven zeigen, dass der größte Anteil an der Zahl der Tetraedern sowohl im gesamten Netz als auch bei der Gesamterzeugung durch Teiltetraeder von Abschlussvolumen bestimmt wird. D.h. je mehr Partitionen bzw. Prozessoren verwendet werden, umso höher ist der Anteil an Abschlusstetraedern. Da Abschlusstetraeder eine hohe Bedeutung wegen der Winkeleigenschaft (Konditionszahl) für die Numerik besitzen, ist somit die Wahl eines geeigneten Fehlerschätzers bzw. -indikators wichtig.

Gesamtzeitaufwand der Adaptionsphasen Die Kurven für den Gesamtzeitaufwand der Adaption zeigen ein gutes Skalierungsverhalten (Abbildung 4.7, linke Spalte, 3. Diagramm von oben). Wie durch das Design für den parallelen Adaptionsalgorithmus beabsichtigt wurde, verlaufen die Kurven mit steigender Anzahl an beteiligten Prozessoren zunehmend flacher. Im Schnitt halbiert sich der Zeitaufwand bei Vervierfachung der Prozessoranzahlen. Dieses Verhalten zeigt sich über den gesamten Verlauf der Simulation ohne größere Einbrüche, die durch schwankende Zahlen an markierten Tetraedern begründet sind. Es zeigt sich bei größer werdenden Prozessorzahlen (64 und 256 Prozessoren), dass die Kurven gegen Ende der Simulation eher weiter auseinander laufen.

Zeitaufwand der Tetraedererzeugung Aus dem Kurvenverlauf für den Zeitaufwand der Tetraedererzeugung (Abbildung 4.7, rechte Spalte, 3. Diagramm von oben) ist ersichtlich, dass ca. die Hälfte der Gesamtzeit der Adaptionsphase für die eigentliche Erzeugung der Tetraeder benötigt wird. Die restliche Zeit wird für den Auf- und Abbau des Halosystems sowie Konsistenzprüfung und Partitionszusammenfügung (Namensraumabgleich der Randobjekte) verwendet. Die Schwankungen der Werte sind in diesen Kurven deutlicher zu sehen als in denen der Gesamtzeit, was auf die ständig schwankende Zahl der markierten Tetraeder je Partition zurückzuführen ist. Der Kurvenverlauf gleicht ansonsten dem der Gesamtzeit für alle Prozessoranzahlen.

Geschwindigkeit der Tetraedererzeugung Die Geschwindigkeit der Tetraedererzeugung (Abbildung 4.7, linke Spalte, unten) steigt mit Zunahme der Prozessorzahlen. Je mehr Prozessoren beteiligt sind, umso stärker machen sich die Schwankungen bemerkbar. Auch hier zeigt sich das Skalierungsverhalten wie in den anderen Kurvendiagrammen, bei Vervierfachung der Prozessoren verdoppelt sich im Schnitt die Geschwindigkeit. Dieses Verhalten zeigt sich nach der Einschwingphase kontinuierlich über den gesamten Simulationslauf.

Geschwindigkeit der Tetraedervergröberung Die Geschwindigkeit für die Tetraedervergröberung (Abbildung 4.7, rechte Spalte, unten) zeigt ein ähnliches Skalierungsverhalten wie für die Tetraedererzeugung. Die Schwankungen fallen in diesem Fall für die Kurven allerdings deutlich geringer aus. Dies liegt in der Lokalität im Algorithmus der Vergröberung begründet. Die Vergröberung findet nur lokal auf einem Rechenknoten bzw. einer Partition statt. Dadurch erzielt die Vergröberung eine deutlich höhere Geschwindigkeit als die Verfeinerung, weil keine Kommunikation zu Nachbarpartitionen durchgeführt werden muss und die Aktualisierungskomplexität für die Datenstrukturen der Netzdarstellung wesentlich geringer ist (s. Algorithmus 3.15).

4.3.1 Qualitätsgebundene Formadaption

Das Ziel der qualitätsgebundenen Formadaption (s. Abschnitt 3.3.2) ist es, während der Entscheidungsphase für die Art des Abschlusses die Form der Teiltetraeder bzw. des Ausgangstetraeders mit in die Entscheidung einfließen zu lassen. Dazu dient ein geeignet gewähltes Qualitätsmaß, das je nach Wert für die gegebene Form des Abschlusses ein weiteres Hilfsmittel zur Entscheidung darstellt. Unterschreitet die Qualität eine vorgegebene Schranke, so wird das Tetraeder nicht als Abschluss zur Auflösung der hängenden Netzobjekte betrachtet, sondern wiederum mittels der roten Verfeinerungsstrategie regulär verfeinert (vgl. hierzu Algorithmus 3.26). Der Wert für die untere Schranke ist abhängig von der aktuellen Qualitätsverteilung des Netzes und kann wegen der Dynamik, die normalerweise der zu berechnenden (numerischen) Problemstellung zugrunde liegt, nicht vorherbestimmt werden. Er muss daher empirisch bestimmt werden.

Wenn es möglich ist, mittels der qualitätsgebundenen Formadaption die Verteilung der Qualitätswerte zu verändern, dann hat dies indirekt Einfluss auf den numerischen Teil der Simulation. Die Konditionszahl des Matrixsystems, welches aus dem diskretisierten Gebiet berechnet (assembliert) wird, steigt mit der Art und Anzahl der kleinsten Winkel der Tetraeder (s. z.B. [BA76, Kri92]). Werden also die Werte im unteren Bereich für die Verteilung der Qualitäten verbessert, erreicht man dadurch automatisch eine Verbesserung der Konditionszahl. Deshalb verringert sich die notwendige Zahl an Iterationen für die numerischen Gleichungslöser bis zur Konvergenz. Zusätzlich kann durch die Formadaption eine Stabilisierung bei Matrizen mit numerischen Problemen (z.B. Druckkorrektur im Rahmen eines Splittingverfahrens [BM06, BM07]) erreicht werden. Der Nachteil an der Formadaption ist die zwangsläufig gesteigerte Anzahl an Tetraedern, die durch die zusätzliche Anwendung der regulären Verfeinerungsstrategie erzeugt werden. Es gilt also, die untere Schranke für die qualitätsgebundene Formadaption so zu bestimmen, dass die Anzahl der Tetraeder und damit die Zahl der zusätzlich eingeführten Unbekannten in vertretbarem Rahmen (Laufzeit der Gleichungslöser!) bleibt und damit nicht kontra-produktiv zur Gesamtlaufzeit der Simulation ist.

Problembeschreibung

In diesem Kapitelabschnitt soll anhand einer Beispielsimulation eine Untersuchung durchgeführt werden, um eine solche untere Schranke zu bestimmen. Hierbei wird wiederum ein geometrischer Fehlerschätzer wie in Problemstellung 4.2 verwendet, der diesmal in einem Einheitswürfel die Oberfläche einer Kugel aus einem groben Startnetz approximieren soll.

PROBLEMSTELLUNG 4.3
Gegeben sei ein Einheitswürfel $\Omega = [0,1]^3$*. Die Oberfläche einer Kugel* K *mit Mittelpunkt* $M_K = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2})^T$ *und dem Radius* $R_K = \frac{1}{4}$ *soll durch einen geometrischen Fehlerschätzer approximiert werden.*

Es sollen alle Tetraeder in Ω *verfeinert werden, die von der Oberfläche der Kugel* K *im Laufe der Simulation geschnitten werden. Unterschreitet ein so ausgewähltes Tetraeder ein Volumen von* $V_T = 10^{-7}$*, so wird es nicht weiter verfeinert.*

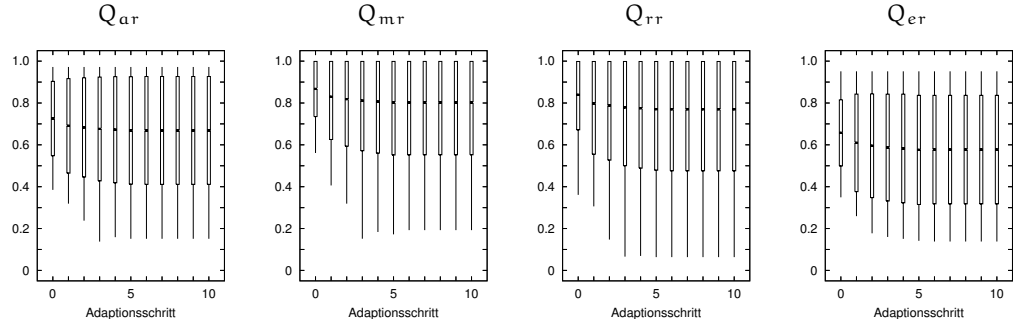
Um die Kugeloberfläche in Ω auszubilden, werden für das diskretisierte Gebiet 10 Adaptionen benötigt, bis der geometrische Fehlerschätzer wegen der Volumenbeschränkung V_T keine weiteren Tetraeder mehr markiert. Es sollen vier normierte Qualitätsmetriken (s. Abschnitt 2.2.2 untersucht werden. Diese sind Q_{ar}, Q_{mr}, Q_{rr} sowie als Beispiel eines *unechten* Qualitätsmaßes Q_{er}. Als Schranke werden die Werte 0.1, 0.2, 0.3 und 0.4 gewählt. Schranken größer als 0.4 bewirken bei allen betrachteten Qualitätsmetriken eine nahezu komplette Auswahl der Tetraeder durch den Bestimmungsalgorithmus 3.26 für die Abschlusswahl. Dies würde einer Vollverfeinerung entsprechen; die Wahl einer Schranke > 0.4 ist somit für die Problemstellung nicht geeignet.

Datenauswertung

Abbildung 4.8 zeigt die Kennwerte der statistischen Verteilung der vier Qualitätsmetriken über alle 10 Adaptionsschritte ohne Formadaptivität in der so genannten Candlestick-Darstellung. Dabei bedeutet der schwarze Punkt innerhalb der kleinen schmalen Rahmen je Adaptionsschritt den Mittelwert μ über alle Tetraederqualitäten der jeweiligen Metrik. Der schmale Rahmen selbst beschreibt den Intervallbereich $[\mu - 2\sigma, \mu + 2\sigma]$ mit Standardabweichung σ (empirische 68-95-99 Regel, s. Abschnitt 4.2.1). Die Linien am oberen und unteren Ende verdeutlichen das jeweilige Maximum und Minimum der Qualitäten.

Für alle Metrikdiagramme gilt, dass ab Schritt 4 die Minima sich nahezu konstant verhalten, nachdem sie sich in einer Einschwingphase in Schritt 1-3 stark nach unten verschoben haben. Die Maxima bleiben konstant über alle Adaptionsschritte. Der Intervallbereich $[\mu-2\sigma, \mu+2\sigma]$ vergrößert sich sowohl nach oben (Q_{ar}, Q_{er}) als auch nach unten (alle vier) drastisch im Vergleich zur Startverteilung in Schritt 0. Die Mittelwerte bewegen sich nach der Einschwingphase ab Schritte 4 nicht mehr, liegen allerdings während der Schritte 0-3 auf einem leicht höheren Niveau.

Abbildung 4.8 Verteilung der Qualitätsmetriken ohne Formadaptivität



In Abbildung 4.9 sind die Candlestick-Diagramme für die betrachteten Qualitätsmetriken dargestellt, wobei als Schranke der größte Wert 0.4 für die jeweilige Testmetrik gesetzt ist. Die Qualitätsverteilung in den Diagramme pro Zeile sind jeweils mit der gleichen Testmetrik berechnet worden. D.h. der Algorithmus 3.26 benutzt in der ersten Zeile die Q_{ar}-Metrik, in der zweiten Zeile die Q_{mr}-Metrik, in der dritten Zeile die Q_{rr}-Metrik und in der letzten Zeile die Q_{er}-Metrik als Auswahlkriterium für die rote Verfeinerungsregel.

Die Diagramme zeigen deutlich, dass sich die Minimagrenze bei fast allen Metrikauswahlen im Vergleich zu den Simulationsläufen ohne Formadaptivitätnutzung (s. Abbildung 4.8) deutlich verbessern, d.h. größere Werte annehmen. Bei einigen Werten, wie z.B. $Q_{rr}\,|_{Q_{mr}<0.4}$ oder $Q_{rr}\,|_{Q_{er}<0.4}$ sind weder Verbesserungen noch Verschlechterungen im gesamten Verlauf erkennbar. Die Einschwingphase in Schritt 1-3 zeigt bei allen größtenteils Verbesserungen in den Minimawerten.

Für $Q_{ar}\,|_{Q_{er}<0.4}$ und $Q_{ar}\,|_{Q_{rr}<0.4}$ werden in den Maximawerten der größtmögliche Wert von 1.0 kurz nach der Einschwingphase erreicht, was im Verteilungsdiagramm für Q_{ar} ohne Formadaptivität nicht sichtbar ist. Dies gilt auch für $Q_{er}\,|_{Q_{rr}<0.4}$ und $Q_{er}\,|_{Q_{er}<0.4}$, wobei gleichzeitig eine starke Verschiebung der Minimgrenze zu größeren Werten hin erkennbar ist. Diese Werte sind aber mit Vorsicht zu betrachten, da die Q_{er}-Metrik keine *echte*, normierte Qualitätsmetrik ist (s. Abschnitt 2.2.2).

Die Mittelwerte in den Diagrammen verschieben sich gegenüber den Diagrammen in Abbildung 4.8 bei fast allen Metriken nicht, lediglich für Q_{rr} ist eine leichte Verschlechterung sichtbar. Die Intervalle um den Mittelwert $[\mu-2\sigma, \mu+2\sigma]$ zeigen selten minimale Veränderungen vor allem an der unteren Grenze (s. z.B. Q_{rr}).

Aus Abbildung 4.9 geht hervor, dass als Auswahlmetrik für den Bestimmungsalgorithmus 3.26 für die Abschlusstypen die Aspect-Ratio-Metrik Q_{ar} am Besten geeignet ist. Sie zeigt die günstigste Verteilung der Qualitäten im Vergleich über alle anderen Metriken. In Abschnitt A.1 ist eine komplette Auflistung über alle Schrankenwerte für die Formadaptivität in den jeweils betrachteten Qualitätsmetriken aufgeführt. Für alle Werte kleiner als die Schranke 0.4 zeigen sich allerdings schlechtere Werte in der Verteilung, so dass der Wert 0.4 für diese Problemstellung als optimal betrachtet werden

Abbildung 4.9 Verteilung der Qualitätsmetriken mit Adaptionsschranke < 0.4

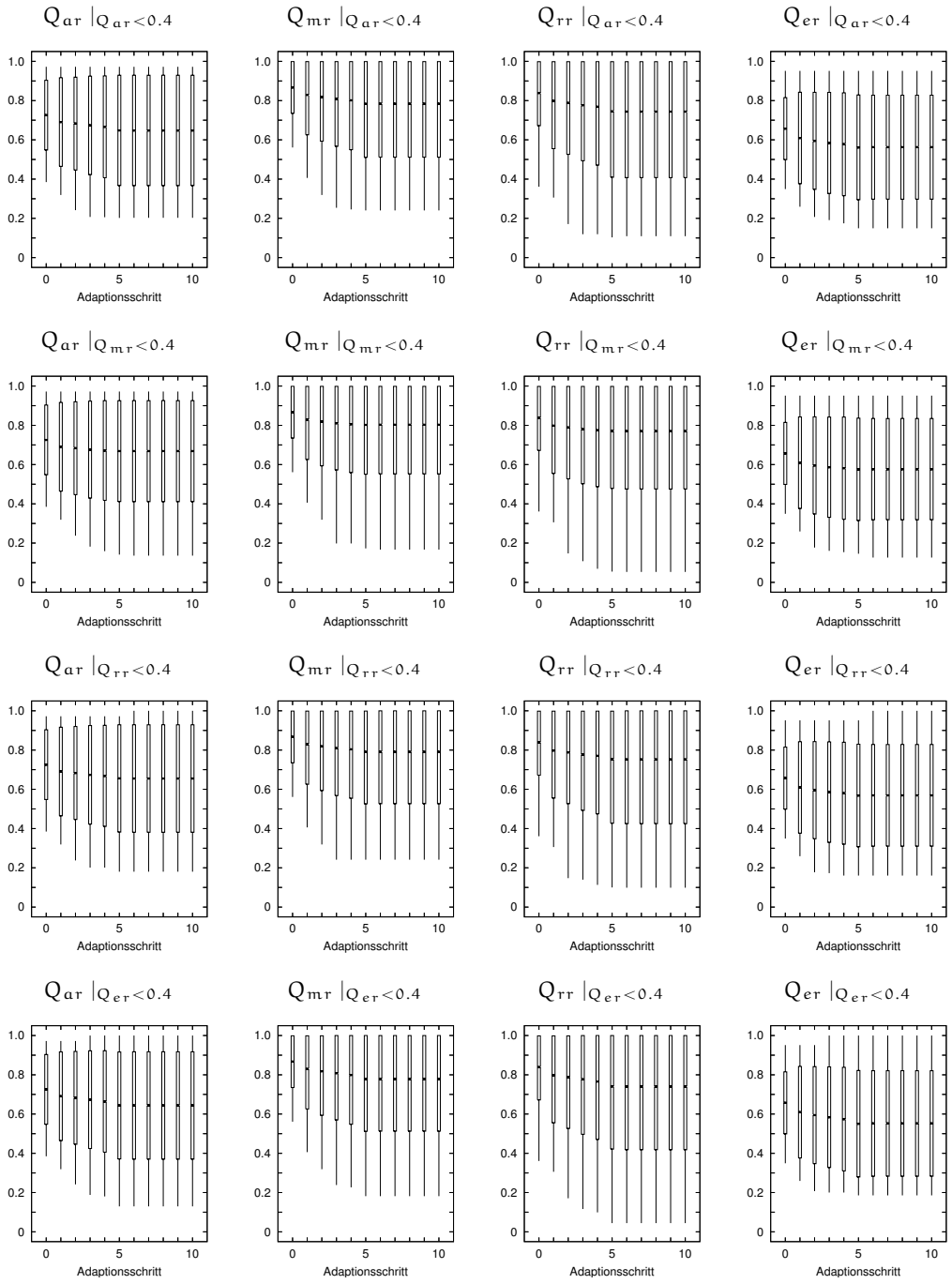
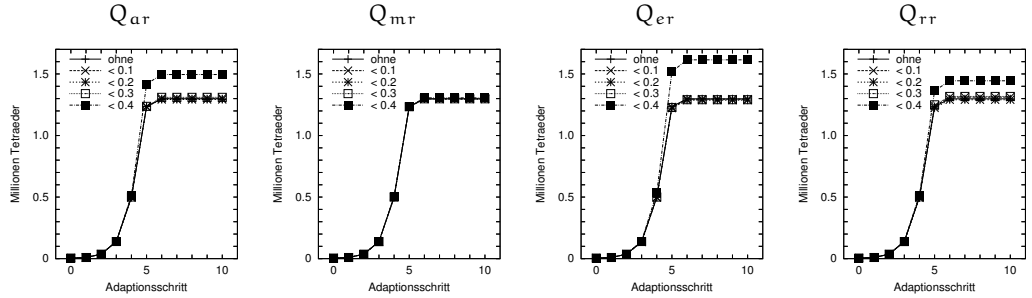


Abbildung 4.10 Tetraederanzahl bei adaptiver Qualitätsbeschränkung



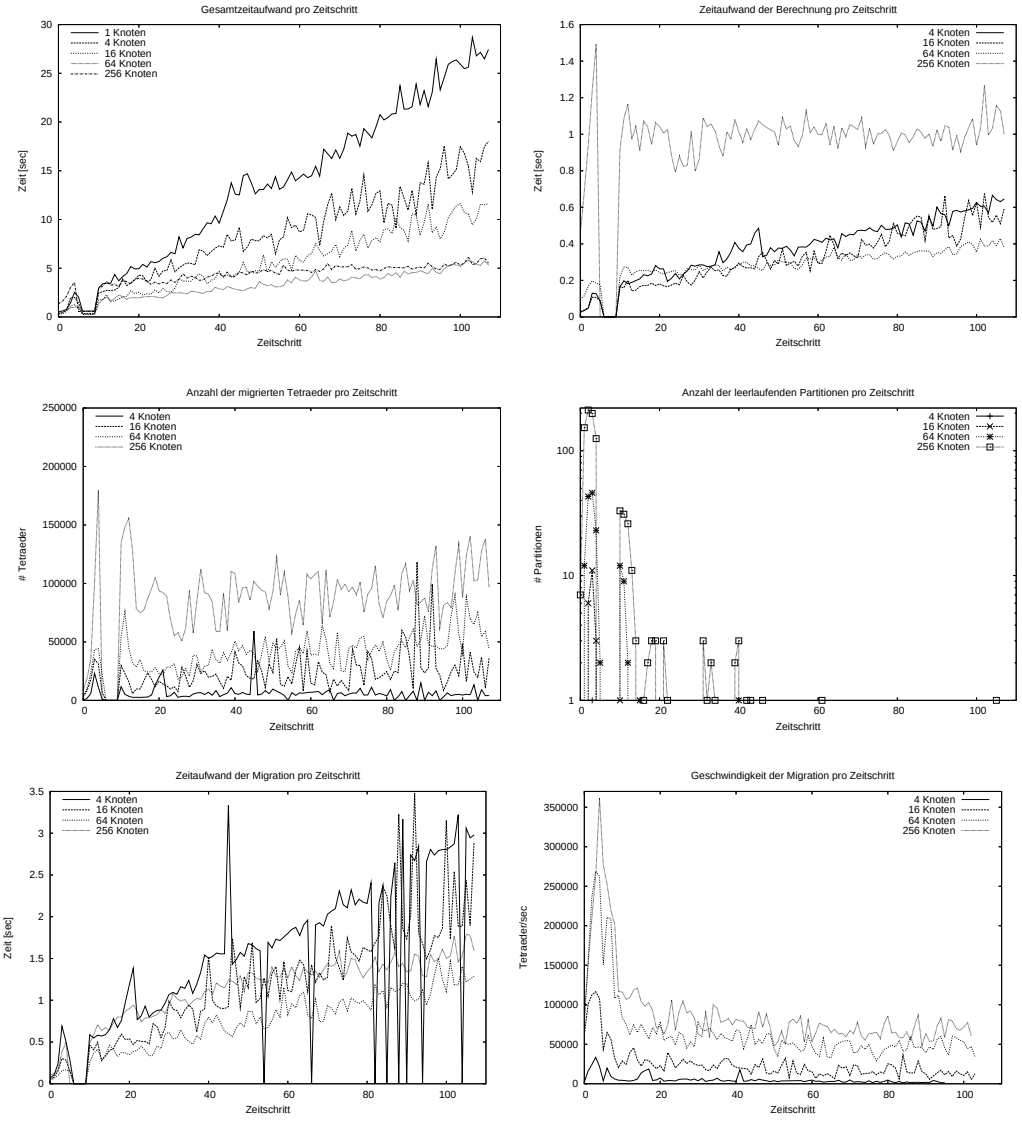
kann.

Betracht man die Zahl der erzeugten Tetraeder jeweils ohne und mit Formadaptivität bei unterschiedlichen Schrankenwerten (s. Abbildung 4.10), so zeigt sich, dass der Schrankenwert 0.4 eine nicht unerhebliche Zahl an Tetraedern zusätzlich erzeugt. Während die Anzahlen für die Schrankenwerte zwischen 0.1 und 0.3 sowie ohne Formadaptivität nahezu gleich bleiben, vergrößert der Wert 0.4 bei allen Metriken bis auf Q_{mr} die Tetraederzahl. Die Metrik Q_{ar} erzeugt rund 15% Tetraeder mehr und liegt gegenüber Q_{rr} und Q_{er} im Mittelfeld. Die eigentlich bessere Wahl von Q_{mr} erzeugt zwar nahezu keine zusätzlichen Tetraeder, verbessert aber die Verteilungen der Tetraederqualitäten kaum (s. Abbildung 4.9, zweite Zeile). Sollte also der Numerikteil der Simulation (vor allem Gleichungslöser) einen Vorteil durch die Qualitätsverbesserungen für die Berechnungen erhalten, so stellt die Metrik Q_{ar} mit einem Schrankenwert von 0.4 die beste Wahl für Problemstellung 4.3 dar.

4.4 Leistungsanalyse der Lastverteilung

Die Lastverteilung gehört wie die Adaption zu den zentralen Phasen einer Simulation, die geometrische Netzmodifikationen durchführt. Die Aufgabe der Lastverteilung ist die möglichst geschickte Verteilung der Elemente eines Netzes auf die Partitionen, um dadurch eine gleichmäßige Auslastung für alle Rechenknoten zu erhalten. Für die Beurteilung der Lastverteilungsphase sind in der Praxis mehrere Kriterien wichtig. Dazu zählen der Zeitaufwand für die Berechnung des Lastausgleiches, die Anzahl der zu migrierenden Tetraeder und die Zeit, die benötigt wird, um die Migration durchzuführen. Zusätzlich ist es wichtig, das die Lastverteilung entscheidet, möglichst keine Partitionen komplett zu migrieren. Diese Entscheidung hängt allerdings sehr von den Optimierungszielen ab, die die Lastverteilungsstrategie verfolgt. Eine Partition komplett zu migrieren bedeutet erhöhten Aufwand sowohl in zeitlicher Hinsicht als auch bzgl. des Speicheraufwandes.

Abbildung 4.11 Laufzeitcharakteristiken der Lastverteilung von Problemstellung 4.2



Datenauswertung

Die Messungen für die Lastverteilungsphase in diesem Kapitel beziehen sich auf die Problemstellung 4.2 in Abschnitt 4.3. Als Lastverteilungsverfahren wurde die Bibliothek JOSTLE [Wal02] eingesetzt. Andere Lastverteilungsverfahren wie z.B. METIS [KK98a] oder FLUX [Sch06] liefern bzgl. ihrer Optimierungsstrategie ein vergleichbares Verhalten mit ähnlichen Messwerten. Als Messgrundlage dienten Startnetze, die mit der sequenziellen Bibliothek PARTY [DP97] initial partitioniert wurden. Da die Optimierungsstrategien bzw. -ziele zwischen den Verfahren für die initiale Partitionierung und den dynamischen Repartitionierungen teilweise unterschiedliche sind, findet zu Beginn

einer Simulation häufig ein erhöhter Austausch von Tetraedern statt, um der Optimierungsstrategie des verwendeten Verfahrens zu genügen. Für die Messungen wurden als Rechnenkonfiguration 4, 16, 64 und 256 Prozessoren verwendet. Die Ergebnisse sind als Kurvendiagramme in Abbildung 4.11 dargestellt.

Gesamtlaufzeit Der Zeitaufwand für einen gesamten Schritt einer Simulation (Abbildung 4.11, oben links) von Problemstellung 4.2 verhält sich nahezu äquivalent zum Gesamtzeitaufwand der Adaptionsphase je Zeitschritt (s. Abbildung 4.7), d.h. die meiste Zeit der Simulation wird für die Adaption benötigt. Auffällig ist, dass die Kurven für 64 und 256 Prozessoren nach der Einschwingphase (Schritt 1-10) verhältnismäßig gleich verlaufen und sich gegen Ende der Simulation sogar angleichen, wobei die Kurve für 256 Prozessoren minimal über der für 64 Prozessoren liegt. Schwankungen innerhalb der Kurven sind durch die unterschiedliche Zahl der zu modifizierenden Tetraeder durch Adaption und Lastverteilung begründet. Der Einbruch der Kurven vor Zeitschritt 10 findet deshalb statt, weil in der Einschwingphase der geometrische Fehlerschätzer für unterschiedliche Prozessoranzahlen unterschiedlich viele Tetraeder markiert. Die Netze unterscheiden sich daher schon zu Beginn der Simulation und verstärken dieses Verhalten im weiteren Verlauf. Dies liegt u.a. an der Volumenbegrenzung V_T des Fehlerschätzers.

Zeitaufwand der Lastausgleichsberechnung Die Zeit, die benötigt wird, um einen Lastausgleich innerhalb eines Schrittes zu bestimmen (Abbildung 4.11, oben rechts), verhält sich atypisch für ein paralleles Programm. Obwohl die Zahl der Tetraeder im Laufe der Simulation stetig steigt (s. Abbildung 4.7), benötigen die Simulationsläufe für 4, 16 und 64 Prozessoren bis auf die obligatorischen Schwankungen nahezu gleich viel Zeit, um einen Ausgleich zu berechnen. Eine leicht steigende Tendenz im Kurvenverlauf ist zudem erkennbar. Einzig die Simulation, die auf 256 Prozessoren gerechnet wurde, benötigt sowohl vor als auch nach der Einschwingphase eindeutig mehr (doppelt soviel) Zeit. Dies ist ein Indiz dafür, dass JOSTLE mit dieser Problemstellung und dem Startnetz bei hohen Prozessorezahlen weniger gut klar kommt, als mit niedrigeren Anzahlen.

Anzahl der migrierten Tetraeder Die Kurven für die Anzahl der migrierten Tetraeder (Abbildung 4.11, mitte links) zeigen deutlich, dass mit steigender Zahl von verwendeten Prozessoren die Menge der zu migrierenden Tetraeder ebenfalls steigt. Es fällt auf, dass sich trotz der starken Schwankungen in den Kurvenverläufen die Anzahl der Tetraeder um ein konstantes Niveau bewegen, d.h. JOSTLE versendet für diese Problemstellung bis auf Schwankungen ungefähr gleiche Datenmengen abhängig von der Zahl der Partitionen bzw. Prozessoren. Zu berücksichtigen hierbei ist, dass während der Simulation die Gesamtzahl der Tetraeder allerdings stetig steigt (s. Abbildung 4.7). Daraus lässt sich schließen, dass die Partitionen sich um die Kugel im Simulationsgebiet Ω konzentrieren und mit ihr mitbewegen.

Anzahl der leerlaufende Partitionen Ein wichtiges Kriterium zur Beurteilung für die Qualität der Lastverteilungsberechnung ist die Anzahl der Partitionen, die das Verfahren

bzgl. ihrer Optimierungsstrategie dafür bestimmt, komplett, d.h. mit allen Netzobjekten und Daten, zu einer anderen Partition bzw. Rechenknoten zu migrieren (leerlaufen einer Partition). Je größer die Anzahl solcher Partitionen ist und je häufiger solche Komplettmigrationen auftreten, umso schlechter ist dies für die Simulation sowohl in zeitlicher Hinsicht (Datentransfer, Zeitschrittdauer) als auch bzgl. des erhöhten Speicheraufkommens, der für die Kommunikationspuffer benötigt wird. In Abbildung 4.11, mitte rechts, liegt dieses Migrationsverhalten als Kurvendiagramm vor. Hier ist deutlich erkennbar, dass vor der Einschwingphase der Simulation ein starkes Leerlaufen der Partitionen bei allen vier Kurven sichtbar ist. Zudem fällt auf, dass je mehr Prozessoren verwendet werden, die Zahl der betroffenen Partition stark ansteigt. Dieses Verhalten ist dadurch erklärbar, dass die Startnetze, die initial mit PARTY partitioniert wurden, dem Optimierungsziel von JOSTLE nicht genügen. Daher entscheidet JOSTLE zum Beginn der Simulationslaufzeit, viele Partitionen komplett zu verschieben. Kurz nach der Einschwingphase treten nur noch für die hohen Prozessoranzahlen (64 und 256) solche Leerlauf-Situationen auf. Für 256 Prozessoren zieht sich dieses Verhalten sogar weit bis zur Mitte der Simulation hin und ist vereinzelt nach am Ende der Simulation sichtbar. Dies ist wiederum ein Indiz dafür, dass JOSTLE in dieser Problemstellung mit großen Prozessorzahlen weniger gut klar kommt als mit niedrigen.

Zeitaufwand der Migration Die Kurven für den Zeitaufwand der Migration (in Abbildung 4.11, unten links) verhalten sich entsprechend den Anzahlen der zu migrierenden Tetraeder. Mit dem Fortschreiten der Simulation wird immer mehr Zeit für die Phasen der eigentliche Datenmigration (vgl. Abschnitt 3.4) aufgewendet. Im Diagramm fällt auf, dass die Kurve für 256 Prozessoren teilweise auf einem höheren Niveau (zwischen der Kurve für 16 und 64 Prozessoren) verläuft. Dies ist verständlich, da für 256 Prozessoren durch JOSTLE entschieden wurde (s. Diagramm mitte links), eine höhere Anzahl an Tetraeder zu migrieren. Das Verhalten korrelliert zudem mit der Gesamtzahl der Tetraeder im Netz. Die Einbrüche in der Kurve für 4 Prozessoren rühren daher, das JOSTLE zu den betrachteten Zeitpunkten keine Tetraeder migrieren möchte, da in diesen Schritten das Optimierungsziel von JOSTLE nach der Adaptionphase noch erfüllt ist, d.h. kein Ungleichgewicht aus Sicht des Ausgleichsverfahrens besteht. Der hohe Peak bei ca. Zeitschritt 45 für die 4-Prozessorenkurve stammt von einer erhöhten Anzahl von zu verschiebenden Tetraedern, welches ebenfalls im Diagramm über die Anzahl der zu migrierenden Tetraeder erkennbar ist.

Geschwindigkeit der Migration Überträgt man die Daten für den Zeitaufwand der Migration in ein Diagramm für die erreichte Migrationsgeschwindigkeit (Abbildung 4.11, unten rechts), so erkennt man, dass während der Einschwingphase der Simulation ein hoher Wert für alle Rechenkonfigurationen erreicht wird. Nach dem Einschwingen fällt jede Kurve stark ab und bewegt sich unter leichten Schwankungen für den Rest der Simulationszeit auf gleichem Niveau. Dieses Verhalten ist dadurch erklärbar, dass die geringen Tetraederanzahlen nach der Einschwingphase nicht das volle Potenzial des Migrationsalgorithmus (vgl. Algorithmus 3.31) ausnutzen können. Die Werte während der

Einschwingphase spiegeln daher besser das Geschwindigkeitspotenzial wieder. Aus den Kurven ist dennoch gut ablesbar, dass die Migration gut skaliert: bei Vervierfachung der Prozessoranzahl, verdoppelt sich die erreichte Geschwindigkeit. Dieses Verhalten bleibt bis auf kleine Schwankungen in den Kurven nahezu konstant.

4.5 Anwendungsbeispiel: Umströmung eines Zylinders

Das folgende Unterkapitel zeigt anhand eines in der Literatur bekannten Anwendungsproblems (vgl. hierzu [MS96]) Leistungswerte des in dieser Arbeit entwickelten Objektmodells und der darauf aufbauenden Algorithmen. Die Durchführung der einzelnen Simulationsläufe erfolgte ebenfalls wie in den vorangegangenen Unterkapitel mittels der Simulationsumgebung *padfem*^{2}. Bei der Analyse der Teilproblemstellungen wird weniger auf die für Navier-Stokes-typischen Fragestellungen eingegangen, als viel mehr die Auswirkungen des entwickelten Objektnamensmodell nebst dessen induzierter Algorithmenstrukturen untersucht.

Das hier behandelte Beispiel einer stationären/instationären Strömungssimulation behandelt die Umströmung eines Zylinders mit angenähertem Kreis- bzw. Quadratquerschnitt in einem quaderförmigen Kanal unter vorgegebenen Rand- und Simulationsparametern. Dieses Simulationsszenario wird u.a. dazu verwendet, sowohl numerische Verfahren als auch komplette Simulationsumgebungen zu testen und zu vergleichen.

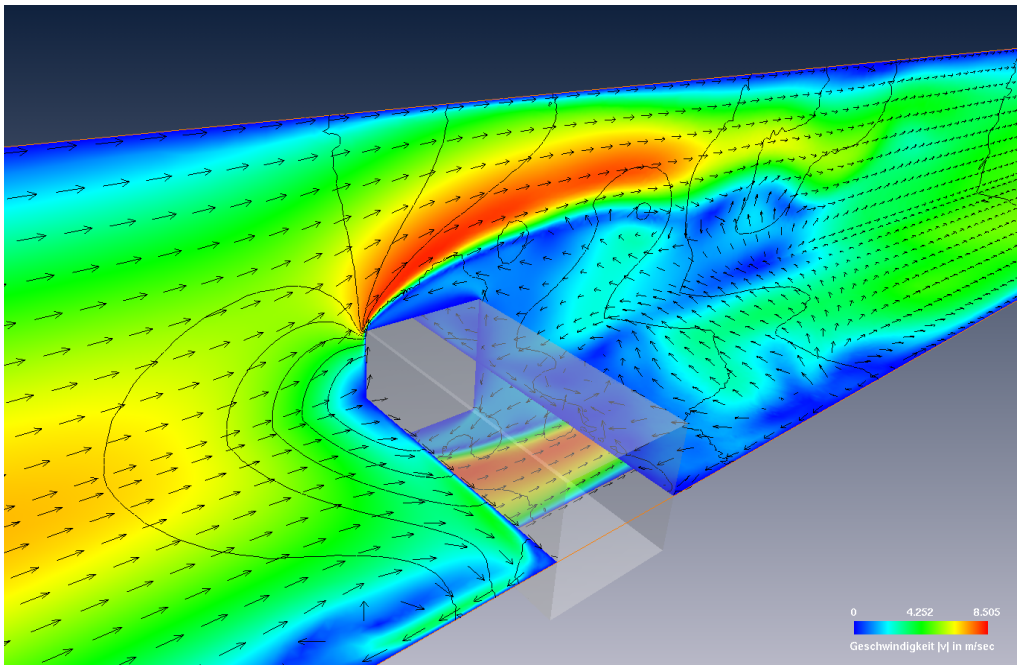
Die Simulation kann mit mehreren unterschiedlichen Reynoldszahlen durchgeführt werden. Referenzwerte für $\mathrm{Re} = 20$ (stationäre Lösung) bzw. $\mathrm{Re} = 100$ (Übergang in instationären Zustand) sind in der Literatur bekannt. Charakteristisch für dieses Simulationsszenario ist, dass je höher die Reynoldszahl ist, desto schneller und deutlicher tritt ein Flackern im Strömungs- und Druckbild auf, welches sich zur sog. *Karmanschen Wirbelstraße* ausprägt. Es wurden reale physikalische Experimente durchgeführt, bei denen dieses Simulationsmodell nachgebaut und exakt vermessen wurde. Die Ergebnisse dieser Experimente dienen als Referenz für die Verhaltensmuster der Strömung bei unterschiedlichen Reynoldszahlen.

Abbildung 4.12 zeigt beispielhaft in einer Nahaufnahme eine mittig positionierte Schnittebene durch den Kanal in z-Achsenrichtung, auf der der Betrag der Geschwindigkeit als Farbverlauf optisch anschaulich dargestellt ist ($\mathrm{t} = 0.94$ Sekunden, $\mathrm{Re} = 3200$). Man kann deutlich den Beginn einer Wellenbewegung erkennen, die sich im weiteren Verlauf der Simulation verstärkt.

Geometriebeschreibung

In [MS96] werden sowohl eine Geometriedefinition für den zwei- als auch den dreidimensionalen Fall vorgegeben. Zudem kann der zu umströmende Zylinder, der im vorderen Teil des Kanals eingebracht ist, sowohl quadratisch als auch (approximativ)

Abbildung 4.12 Strömungsvisualisierung ($Re = 3200, t = 0.94\,sec$)



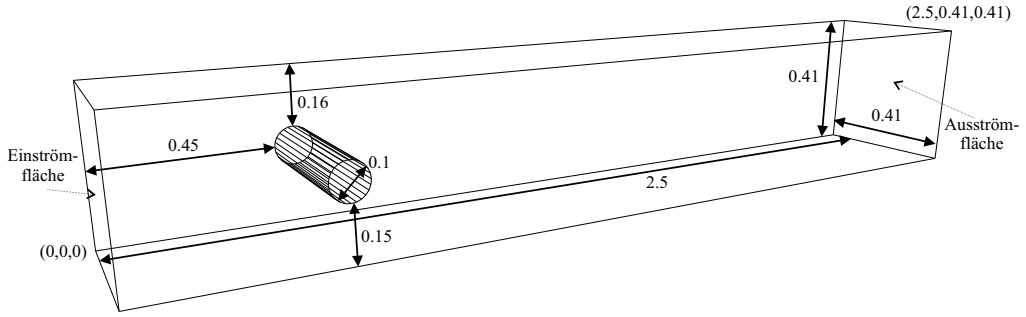
kreisförmig aufgelöst werden. In diesem Anwendungsbeispiel wird der interessante dreidimensionale Fall für den quadratischen Querschnitt des Zylinders betrachtet.

Die Simulation findet in einem quaderförmigen Kanal (Ω bzw. Ω_h) mit den Ausmaßen $2.5 \times 0.41 \times 0.41$ (in Meter) statt. Die Mittelpunktgerade des Zylinders geht durch die Koordinaten $(0.5, 0.16, 0.0)^T$ und $(0.5, 0.16, 0.41)^T$, der Zylinder selbst hat einen Durchmesser von $D_{cyl} = 0.1$ bei kreisförmigem Querschnitt (s. Abbildung 4.13). Für das hier angewandte Szenario mit quadratischem Querschnitt besitzt der Zylinder eine Kantenlänge von 0.1 Meter und ist mit seinen Seitenflächen parallel zu den Flächen des Kanals an der gleichen Position wie beim anderen Szenario eingehängt. Die Einströmfläche Γ_{in} ist die dem Zylinder näher zugewandte kleine, quadratische Mantelfläche des Kanals, die Ausströmfläche Γ_{out} liegt auf der gegenüberliegenden Seite. Der Koordinatenursprung ist die linke untere Ecke auf der Einströmfläche. Alle Raumkoordinaten sind positiv. Man beachte, dass der Zylinder nicht zentriert in der y-Achsenrichtung des Zylinder positioniert ist. Diese Verschiebung um 0.05 Meter begünstigt eine Störung des Geschwindigkeits- bzw. Druckfeldes im Laufe der Simulation, um eine mögliche stationäre Lösung bei niedrigen Reynoldszahlen zu unterbinden.

Problembeschreibung

In den Problemstellungen für die Szenarien wird ein *inkompressibles Fluid* betrachtet, für das die Erhaltungsgleichungen von Impuls und Masse wie folgt definiert sind (vgl.

Abbildung 4.13 Geometriedefinition der Zylinderumströmung



Bezeichnungen und Parameter in [MS96]):

$$\rho\frac{\partial U_i}{\partial t} + \rho\frac{\partial}{\partial x_j}(U_j U_i) = \rho\nu\frac{\partial}{\partial x_j}\left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i}\right) - \frac{\partial P}{\partial x_i}. \tag{4.2}$$

und

$$\frac{\partial U_i}{\partial x_i} = 0, \tag{4.3}$$

Hierbei bezeichnen t die Zeit, $(x_1, x_2, x_3)^T = (x, y, z)^T$ kartesische Raumkoordinaten, P den Druck und $(U_1, U_2, U_3)^T = (U, V, W)$ die Geschwindigkeit. Die kinematische Viskosität beträgt $\nu = 10^{-3}\,m^2/s$, die Dichte des Fluids ist konstant $\rho = 1.0\,kg/m^3$.

Die Randbedingung für den Einströmrand Γ_{in} lautet

$$U(0, y, z) = 16 \cdot U_m \cdot yz\frac{(0.41 - y)(0.41 - z)}{0.41^4}, V = W = 0, \tag{4.4}$$

wobei U_m die Einströmgeschwindigkeit in Meter pro Sekunde bezeichnet und für die Beispielrechnung $72m/s$ beträgt (Reynoldszahl $Re = \frac{\overline{U}_m \cdot D_{cyl}}{\nu} = \frac{\frac{4}{9}72 \cdot 0.1}{10^{-3}} = 3200$).

Da das Anwendungsbeispiel lokal adaptiv gerechnet werden soll, wird ein Fehlerschätzer benötigt. Hierzu wird ein residualer Ansatz (s. z.B. [BM07]) verwendet, der folgendermaßen definiert ist. Berechnet wird für jedes Tetraeder $T \in \Omega_h$ ein Fehlerwert η_T mit

$$\eta_T = \left[h^2 \left\| \frac{U^n - U^{n+1}}{dt} + \nu\Delta U - \nabla P - U \cdot \nabla U \right\|^2_{L^2(T)} + \|\mathrm{div}\, U\|^2_{L^2(T)}\right]^{1/2}. \tag{4.5}$$

Es wird die Maximumstrategie angewendet, d.h. mit $\eta_{max} = \max_{T \in \Omega_h}\{\eta_T\}$ wird jedes Tetraeder für die Adaption markiert, für das $\eta_T > \theta\eta_{max}$ gilt mit einem Skalierungsparameter $\theta \in (0, 1)$.

Hiermit ergibt sich nun die folgende Problemstellung:

PROBLEMSTELLUNG 4.4
Gegeben sei die aus Abschnitt 4.5 definierte Kanalgeometrie als Simulationsgebiet Ω bzw. das diskretisierte Gebiet Ω_h. Zu bestimmen ist der zeitliche Verlauf der

Tabelle 4.3 Objektstatistik der partitionierten Startnetze von Problemstellung 4.4

Partitionen	Knoten	Kanten	Flächen	Tetraeder	Halozusatz
1	34541	221482	365928	178987	0 %
2	36390	231194	380319	185514	3.65 %
4	39251	246104	402350	195494	9.22 %
8	43160	266103	431601	208650	16.58 %
16	48892	295308	474248	227816	27.28 %
32	57161	337016	534961	255075	42.51 %
64	68603	393839	617066	291766	63.01 %

(instationären) Lösung von Gleichung 4.2 und 4.3 mit der Einströmrandbedingung aus Gleichung 4.4 für die Reynoldszahl $\mathrm{Re} = 3200$*. Als Fehlerschätzer für die lokale Adaption dient ein residualer Ansatz auf der Basis der in Gleichung 4.5 vorgestellten Berechnung.*

Diskretisierung und Gebietscharakteristika

Die Diskretisierung des Simulationsgebietes erfolgte mit dem Tetraedierungsprogramm *NETGEN* [SGG07]. NETGEN erzeugt Tetraedernetze für die bekannt ist, dass sie sehr gute Qualitätseigenschaften besitzen. Die Netzobjektstatistik für die Startnetze von Problemstellung 4.4 ist in Tabelle 4.3 für die verschiedenen Partitionsanzahlen dargestellt. Die Werte für die Elemente geben die Gesamtanzahl im verteilten Netz, inklusive mehrfach vorhandener Objekte, auf dem Rand und im Halo an. In der Spalte für den Halozusatz steht der prozentuale Anteil an Tetraedern, die gegenüber dem Startnetz im sequenziellen Fall noch zusätzlich zur Gesamtzahl hinzukommen.

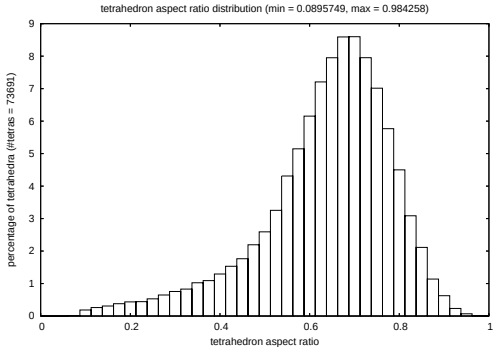
Da sich die Netze nach einer Adaption im parallelen Fall an Partitionsgrenzen unterscheiden und während der Laufzeit durch die Partitionsgrenzverschiebungen, die durch die dynamische Lastverteilung bedingt sind, immer weiter differieren, kann nur für das Startnetz ein einheitliches Bild der Netzcharakteristika angegeben werden. Diese sind für die verschiedenen Qualitätsmetriken als normierte Werte in Tabelle 4.4 angegeben. Abbildung 4.14 zeigt zusätzlich die Verteilung der Standardmetrik $\mathrm{Q_{ar}}$ als Balkendiagramm. Es ist deutlich erkennbar, dass sich die Menge der Tetraeder wie auch schon bei Problemstellung 4.1 zu sehr großen Anteilen im Intervall $[0.45, 0.8]$ anhäufen. Dies ist wiederum ein Kriterium für die gute Qualität des Ausgangsnetzes.

Im Vergleich zu den normierten Qualitätsmetriken von Problemstellung 4.1 zeigt sich, dass sich die Werte größtenteils schlechter darstellen, die zugehörigen Differenzen der beiden Netze unterscheiden sich allerdings nur um sehr kleine Werte. Einige Metriken, wie z.B. $\mathrm{Q_{sa}}$, $\mathrm{Q_{da}}$ oder $\mathrm{Q_{rr}}$, zeigen sogar eine minimale Verbesserung in allen oder einigen Statistikwerten. Wegen dieser geringen Unterschiede kann auch dieses Startnetz

Tabelle 4.4 Normierte Verteilung verschiedener Metriken zu Problemstellung 4.4

Metrik	$\overline{\min}$	$\bar{\mu}$	$\overline{\max}$	$\bar{\sigma}$
Q_{ar}	1.378e-1	1.0	1.515	2.231e-1
Q_{mr}	2.419e-1	1.0	1.253	1.667e-1
Q_{er}	5.544e-1	1.0	1.525	1.226e-1
Q_{rr}	1.422e-1	1.0	1.322	2.088e-1
Q_{sa}	1.131e-1	1.0	1.863	2.837e-1
Q_{da}	1.009e-1	1.0	1.573	2.590e-1
V_T	9.798e-2	1.0	3.379	3.304e-1

Abbildung 4.14 Q_{ar}-Metrik Verteilung von Problemstellung 4.4

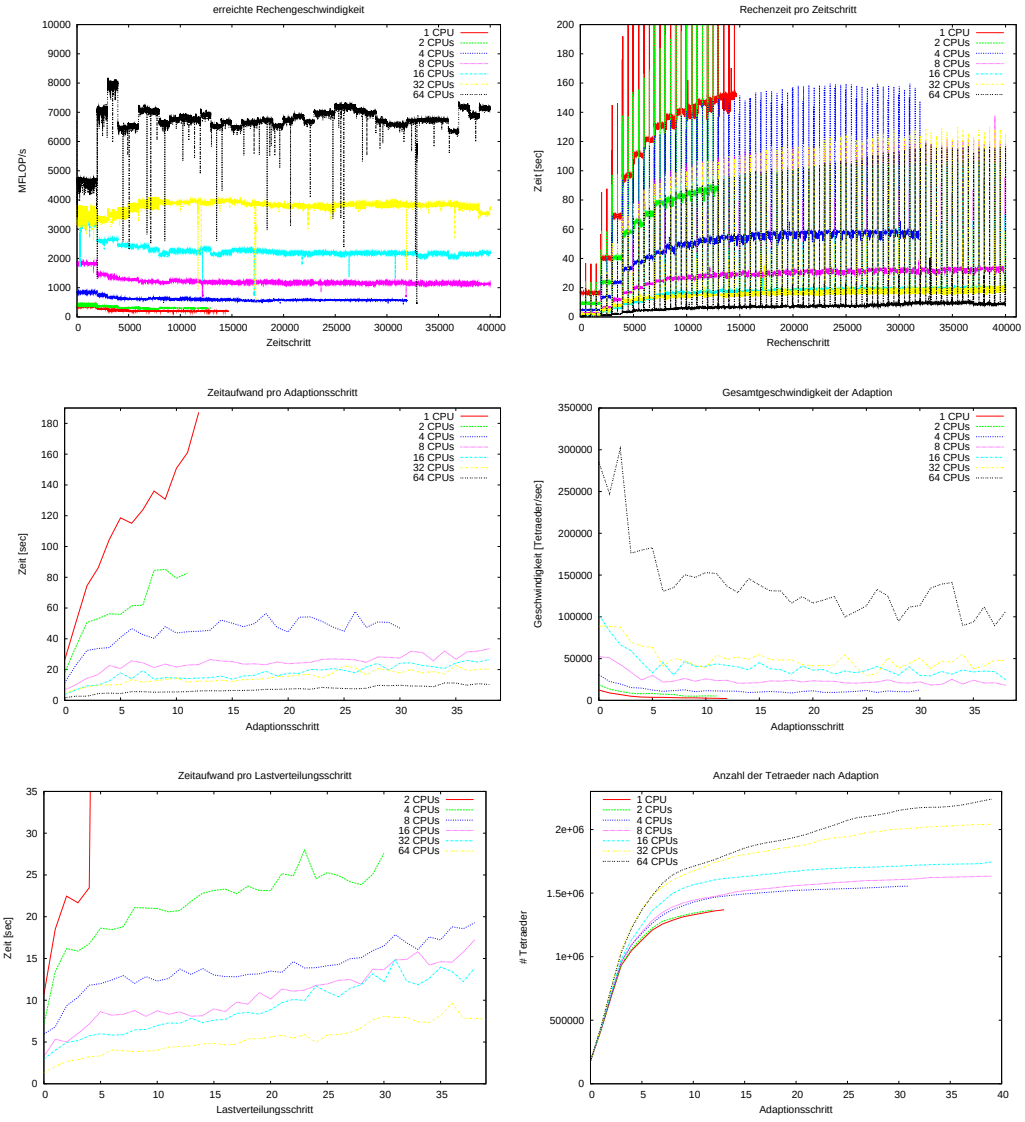


wie das von Problemstellung 4.1 als eine gute Datenbasis betrachtet werden.

Datenauswertung

Für die Simulation von Problemstellung 4.4 wurden 40000 Zeitschritte gerechnet, wobei die Zeitschrittweite $\Delta t = 10^{-4}$ betrug. Dies entspricht somit 4 Sekunden Simulationszeit. Die Adaption wurde zum ersten Mal in Zeitschritt 2000 durchgeführt und von da an alle 1000 Zeitschritte. Der Skalierungsparameter θ für die Maximumsstrategie des Fehlerschätzers wurde auf 10^{-2} festgelegt. Als Lastverteilungsverfahren für den parallelen Fall diente METIS (s. [KK98a]). Als Folge des begrenzten RAM-Speichers für die sequenzielle sowie die parallele Ausführung für 2 und 4 Prozessoren (auf einen bzw. zwei Rechenknoten abgebildet) konnten wegen der steigenden Zahl der Tetraeder während der Adaption nur ca. 14600, 12900 bzw. 32000 Zeitschritte berechnet werden. Die Messkurven gehen daher für diese Fälle nur über einen Teil des Gesamtbereiches der 4 Sekunden Simulationszeit. Es zeigt sich dadurch, dass für eine komplette Simulation mindestens 4 Rechenknoten (8 Prozessoren) wegen des Datenaufkommens benötigt werden.

Abbildung 4.15 Laufzeitcharakteristiken von Problemstellung 4.4



Als Messkriterien für die 4 Sekunden der Simulation von Problemstellung 4.4 dienen die erreichte Rechengeschwindigkeit in MFLOP/s, die aufgewendete Rechenzeit pro Zeitschritt, die benötigte Zeit für einen Adaptionsschritt sowie des nachfolgenden Lastausgleichs inklusive Migration der Tetraederdaten, die Geschwindigkeit der Adaption in Tetraeder pro Sekunde sowie die Anzahl der Tetraeder im verteilten Netz im Verlauf der Simulation. Die farbigen Kurvendiagramme in Abbildung 4.15 zeigen die ermittelten Messwerte auf dem Parallelrechner *ARMINIUS*.

Laufzeitcharakteristik In Tabelle 4.5 sind die ermittelten sowie extrapolierten Laufzeitdaten und Speedup-Werte für die sequenzielle und parallele Simulationsdurchführung

Tabelle 4.5 Laufzeitstatistik der parallelen Simulation von Problemstellung 4.4

P	T_{max}	$\overline{t_s}$	$\overline{t_{s,N}}$	$\overline{t^*_{s,N}}$	T_s	$T^*_{s,N}$	$S^*_N(P)$	$S^*(P)$
1	14634	107.232 s	48.862 s	64.415 s	435.9 h	715.72 h	-	-
2	12950	60.446 s	32.382 s	44.316 s	217.44 h	492.4 h	1.45	1.77
4	31957	48.045 s	25.902 s	27.317 s	426.49 h	303.52 h	2.36	2.23
8	40000	27.312 s	16.418 s	16.418 s	303.48 h	182.42 h	3.92	3.92
16	40000	16.478 s	10.027 s	10.027 s	183.1 h	111.41 h	6.42	6.51
32	40000	15.526 s	11.909 s	11.909 s	172.52 h	132.32 h	5.41	6.91
64	40000	6.947 s	4.747 s	4.747 s	77.19 h	52.74 h	13.57	15.44

aufgeführt. In der Tabelle bedeuten die Spaltenüberschriften folgendes: P ist die Anzahl der verwendeten Prozessoren. T_{max} ist die Anzahl der tatsächlich berechneten Zeitschritte. $\overline{t_s}$ gibt die durchschnittliche Zeit in Sekunden an, die für einen Zeitschritt bezogen auf T_{max} mit P Prozessoren benötigt wurde, $\overline{t_{s,N}}$ ist der durchschnittliche Zeitaufwand pro Zeitschritt, den der numerische Teil der Simulation benötigt hat, wiederum bezogen auf T_{max} Zeitschritte. $\overline{t^*_{s,N}}$ zeigt den extrapolierten Wert für $\overline{t_{s,N}}$ bezogen auf 40000 Zeitschritte, wenn $T_{max} < 40000$ für P Prozessoren ist. Hierzu wird für die Extrapolation angenommen, dass der Zeitaufwand für alle Zeitschritte größer als T_{max} konstant bleibt und somit dem letzten berechneten Zeitwert entspricht. T_s und $T^*_{s,N}$ stehen für die tatsächliche Gesamtlaufzeit der Simulation sowie für die extrapolierte Zeit des numerischen Anteils in Stunden, bezogen auf $T_{max} = 40000$ Zeitschritte. $S^*_N(P)$ und $S^*(P)$ geben schließlich den extrapolierten Speedup-Wert für den numerischen Anteil und für die Gesamtlaufzeit der Simulation an, wiederum für $T_{max} = 40000$ Zeitschritte.

Der gemessene numerische Anteil bei der Simulation besteht aus dem Lösen von 7 Gleichungssystemen (6 für das Geschwindigkeitsfeld, 1 für die Druckkorrektur) sowie der Berechnung des Transportschrittes für das Navier-Stokes System. Zusätzliche numerische Berechnungen, die für einen Zeitschritt durchgeführt wurden (Matrix-Assemblierung, Rotationsbestimmung, etc), wurden nicht gemessen. Der tatsächliche Zeitaufwand für den Numerikanteil in einem Zeitschritt liegt somit höher als in der Tabelle angegeben. Desweiteren ist zu beachten, dass die Simulation ein Einschwingverhalten am Anfang besitzt, welches bedingt durch die Zunahme der Tetraederanzahl in den Adaptionsphasen ist. Dadurch liegen die Durchschnittswerte in der Tabelle zu niedrig. Erst ab ca. 15000 Zeitschritten können die Laufzeitwerte der Simulation als konstant betrachtet werden (vgl. hierzu Abbildung 4.15, oben rechts).

Rechengeschwindigkeit Das Kurvendiagramm für die erreichte Rechengeschwindigkeit (Abbildung 4.15, links oben) zeigt das typische Verhalten der verwendeten iterativen Gleichungslöser (SPCG, nur Durckkorrektur). Es fällt deutlich auf, dass mit steigender Zahl der verwendeten CPUs, die Kurve mehr und mehr ihr "glattes" Verhalten verliert und sogar bei der höchsten Prozessoranzahl große Störungen im MFLOP/s-

Wert besitzt. Diese Störungen sind durch zwei Gründe bedingt. Der Parallelrechner *ARMINIUS* wird zum einen im Mehrbenutzerbetrieb verwendet. Parallele Applikationen anderer Benutzer des Systems, die über das dedizierte Kommunikationsnetzwerk Daten austauschen, belasten das Netzwerk je nach Kommunikationsaufwand zusätzlich. Ein konstanter Kommunikationsaufwand für die Messungen ist so nicht erreichbar gewesen, da das System nicht exklusiv zur Verfügung stand. Zum anderen ist das gesamte parallele Rechensystem in einem experimentellen (Hardware) und zum Teil auch instabilen (Software) Zustand zum Zeitpunkt der sehr zeitaufwändigen Messungen gewesen. Die dadurch bedingten Störungen zeigen sich ebenfalls in den Kurven durch nicht-periodische Einbrüche in der Rechenleistung. Dieses Verhalten zeigt sich auch in Tabelle 4.5 in den Zeilen für 16 bzw. 32 Prozessoren, hier unterscheiden sich wegen "externer" Störungen die Werte für $\overline{t_s}$ und $\overline{t_{s,N}}$ nur gering trotz Verdoppelung der Prozessoranzahl, ein größerer Sprung ist wiederum bei 64 Prozessoren zu sehen. Dennoch ist erkennbar, dass der Verlauf für alle Kurven nach einer Einschwingphase (ca. ab Zeitschritt 15000) auf einem nahezu konstanten Niveau bleibt. Die erreichten Werte deuten an, dass die Simulation für das recht aufwändige Lösen der Druckkorrektur gut mit der Anzahl der Prozessoren skaliert, da die Werte sich von Kurve zu Kurve im Schnitt relativ verdoppeln. Dies ist sehr deutlich an den Kurven für 8, 16, 32 und 64 Prozessoren zu sehen.

Zeitaufwand pro Zeitschritt Die Kurven für die Rechenzeit pro Zeitschritt (in Abbildung 4.15, rechts oben) zeigen einen einheitlichen Verlauf für alle Anzahlen von Prozessoren. Es ist deutlich eine schrittweise Einpendelung der Werte bis ca. Zeitschritt 15000 erkennbar, danach liegen die Zeitwerte relativ auf konstantem Niveau. Die periodisch auftretenden Peaks in den Kurven sind den Adaptionsphasen mit anschließender Lastverteilung (wenn nötig) sowie langsamen (verteilten) Dateioperationen zuzuschreiben. Es ist deutlich erkennbar, dass die benötigte Zeit pro Schritt mit der Zahl der Prozessoren skaliert (nahezu halbiert). Für 16 und 32 Prozessoren ist dieses Verhalten jedoch nicht sichtbar. Der Grund hierfür liegt in den technischen Schwierigkeiten (s. weiter oben) des verwendeten Parallelrechners. Die Kurve für 32 Prozessoren müsste nach der Tendenz der anderen Kurven niedriger sein (vgl. Tabelle 4.5).

Zeitaufwand pro Adaptionsschritt Das Verhalten der Kurven für den Zeitaufwand der Adaptionsphase (Abbildung 4.15, mitte links) ähnelt dem der Kurven für den Gesamtaufwand pro Zeitschritt. Nach einem steilen bis leichten Anstieg der Kurven für die einzelnen Prozessoranzahlen im Bereich Schritt 5 (bei 32 Prozessoren) bis Schritt 10 (1 bis 2 Prozessoren) verlaufen diese relativ flach bzw. nahezu konstant. Kleinere Peaks im Verlauf der Kurven werden durch unterschiedliche Anzahlen von zu adaptierenden Tetraedern pro Rechenpartition hervorgerufen. Anhand der benötigten Zeit lässt sich erkennen, das der Aufwand einer einzelnen Adaptionsphase[7] ohne Lastverteilungsschritt ungefähr dem Zeitaufwand eines einzelnen Zeitschrittes, in dem nur numerische

[7] Hierzu zählt u.a. Haloauf- und -abbau, Referenzmengenbestimmung, Tetraedervergröberung und -verfeinerung sowie die Netzaktualisierung.

Verfahren ohne geometrische Netzmodifikationen durchgeführt werden, entspricht. Da eine Adaption aber nur alle 1000 Zeitschritte erfolgt, ist der hohe Aufwand hierfür aus Anwendersicht vertretbar.

Adaptionsgeschwindigkeit Aus dem Zeitaufwand für die Adaption lässt sich die Geschwindigkeit der Tetraedererzeugung ableiten (Abbildung 4.15, mitte rechts). Hierbei zeigt sich ein ähnliches Kurvenverhalten wie bei der erreichten Rechengeschwindigkeit im Numerikteil. Nach einer Einschwingphase, die durch Cacheeffekte bedingt sind, pendeln sich die Kurven im weiteren Verlauf auf einem relativ konstanten Niveau ein. Mit zunehmender Zahl an verwendeten Prozessoren steigt auch das Schwankungsverhalten der Kurven. Dies ist besonders bei der Kurve für 32 Prozessoren sichtbar. Der Grund dafür liegt wiederum in der variablen Zahl der zu adaptierenden Tetraeder pro Partition sowie pro Zeitschritt, da es sich bei der Simulation um ein dynamisches System handelt.

Zeitaufwand pro Lastverteilungsschritt Der Zeitaufwand für die Lastverteilung (Abbildung 4.15, unten links) ist weit geringer als der Aufwand für die Adaption. Auch der Kurvenverlauf ist anders. Eine Einschwingphase ist zwar am Anfang jeder Kurve erkennbar, jedoch zeigen die Kurven für die verschiedenen Prozessorzahlen keine Tendenz zu einem konstanten Niveau. Stattdessen steigen diese im Laufe der Schritte leicht an. Die Schwankungen innerhalb der Kurven bleiben aber wie bei der Adaption aus den gleichen Gründen vorhanden. Die Lastverteilung wurde mit der parallelen Version von METIS (vgl. [KK98b, KK98a]) berechnet und durchgeführt. Da METIS in dieser Version sporadisch entscheidet, mal wenige bzw. keine und dann ein paar Schritte später sehr viele Tetraeder (Knoten im dualen Volumengraphen) zu migrieren, schlägt sich dieses Verhalten auf den Kurvenverlauf nieder. Zusätzlich steigt die Komplexität der neuen Partitionsgrenzbestimmung und des notwendigen Abgleiches der Namesräume für die Netzobjekte, so dass dies ebenfalls die Laufzeit in dieser Phase mit steigender Schrittzahl kontinuierlich vergrößert. Die drastische Steigerung bei 2 Prozessoren liegt in dem nicht genug vorhandenen Hauptspeicher begründet. Die Simulation benutzte daher Sekundärspeicher (swapping), dass sich in einem stark erhöhten Zeitverbrauch zeigt.

Tetraederanzahl Das letzte Kurvendiagramm (Abbildung 4.15, unten rechts) zeigt die Anzahl der Tetraeder nach jeder Adaption. Man sieht, dass alle Kurven zu Beginn einen gleichen, stark steigenden Verlauf haben. Dies liegt an der großen Zahl durch den Fehlerschätzer markierter Tetraeder zu Beginn der Simulation. Da das Gitter am Anfang sehr grob für die Simulation ist, d.h. der Fehler an fast allen betrachteten Knoten groß ist, werden hier fast identisch für alle Prozessorzahlen die gleichen Tetraeder zur Adaption ausgewählt. Da markierte Tetraeder an Partitionsgrenzen aber nach einer anderen Regel als innerhalb einer Partition verfeinert werden (s. Algorithmus 3.13, Zeile 17-26) entstehen zwangsläufig unterschiedliche Netze, die umso mehr differieren, je mehr Partitionsgrenzen im verteilten Netz existieren, d.h. es wird mit mehr Prozessoren gerechnet. Dieser Umstand ist im Diagramm ab Adaptionsschritt 3-4 deutlich erkennbar

(ca. 1 Million Tetraeder im Netz), wo die Kurven anfangen auseinander zu laufen. Die Differenz benachbarter Kurven steigt daher mit fortschreitener Schrittzahl.

5 Zusammenfassung

In dieser Arbeit wurde ein verteiltes, spezialisiertes Daten- bzw. Objektmodell für numerische FEM-Simulationsumgebungen für den massiv parallelen Einsatz entworfen, evaluiert und Messungen in einer realen (prototypischen) Implementierung durchgeführt. Es stellt das wichtigste und ressourcenkritischste (zeit- und speichertechnisch gesehen) Bindeglied zwischen den einzelnen Modulen einer numerischen Simulationsumgebung dar. Es zeigte sich, dass das Skalierungsverhalten im massiv parallelen Fall für alle drei Hauptsäulen einer numerischen Simulation – numerisches Modul, geometrisches Adaptionsmodul, Lastverteilungs- und Migrationsmodul – die geforderten Ansprüche erfüllt. Die gesteckten Ziele beim Design wurden alle erreicht.

Die numerische Gesamtleistung, die sich aus dem Objektmodell und der zu berechnenden Problemstellung ableitet, ist mit anderen Simulationsumgebungen trotz der prototypischen Umsetzung gut vergleichbar. Die Transformationsoperationen vom Objektmodell in das numerische Datenmodell (spezielles Matrixformat und Vektoren) hat hier eine entscheidende Bedeutung für die Performance des Numerikmoduls. In der Praxis werden selbst auf großen Parallelrechnern mit realen Problemstellungen nur 5-10% der Peakleistung erreicht. Diesen Wertebereich erreicht das Numerik- und Objektmodell in der *padfem*^{2}-Simulationsumgebung ebenfalls (vgl. Abschnitt 4.2.1 und Abschnitt 4.2.2) und bietet darüber hinaus in der praktischen Umsetzung noch weiteres Optimierungspotential, das ausgeschöpft werden kann. Bei der Untersuchung der numerischen Gesamtleistung zeigte sich zudem, dass die Kombination aus verteiltem Objektmodell und thread-paralleler Bearbeitung der numerischen Daten bei hohen Prozessorzahlen (abhängig von der Gesamtdatengröße) der einfachen Partition-Prozessor Zuordnung – ein Bearbeitungsmodell, welches in anderen kommerziellen und akademischen Simulationsumgebungen eingesetzt wird – überlegen ist.

Das Modul für die geometrische Adaption in numerischen Simulationsumgebungen stellt für das Objektmodell wegen seiner zahlreichen Eingriffe in die Verwaltungsstruktur das Modul mit dem stärksten Modifikationsaufkommen dar. Aufgrund dieser vielen modifizierenden Zugriffe müssen alle Operationen des Objektmodells effizient arbeiten und immer einen konsistenten Zustand hinterlassen. Dies gilt insbesondere für die Bereiche an Partitionsgrenzen im parallelen Fall. In dieser Arbeit wurde daher ein effizienter paralleler Algorithmus zur irregulären Adaption (Rot-Grün Verfeinerung) präsentiert, der direkt auf dem Objektmodell operiert und keine Transformation in ein Datenmodell benötigt, wie es in anderen Simulationsumgebungen üblich ist. Um die Forderung nach hoher Skalierbarkeit zu erfüllen, wurde der parallele Algorithmus derart entworfen, dass er mit reduziertem Informationsgehalt bei Kommunikationsoperationen die partitionsübergreifende Konsistenzwahrung im Objektmodell einhalten kann (nur Kanteninformationen werden übertragen, vgl. Abschnitt 3.3.1). In Messungen mit der parallelen FEM-Simulationsumgebung *padfem*^{2} wurde das Algorithmenverhalten untersucht und die geforderten Skalierungseigenschaften anhand eines Modellproblems bestätigt (vgl. Abschnitt 4.3).

Zur qualitativen Verbesserung der geometrischen Adaption wurde in dieser Arbeit die qualitätsgebundene Formadaption vorgestellt. Mit Hilfe dieser Methode, die durch weni-

ge Änderungen des präsentierten parallelen Adaptionsalgorithmus angewendet werden kann, lässt sich die kleinste Raumwinkeleigenschaft in der verwendeten Netzgeometrie verbessern (vgl. Abschnitt 3.3.2). Da die Raumwinkeleigenschaft direkten Einfluss auf die Konditionszahl zur assoziierten Matrix der Diskretisierung hat, erhält man durch die qualitätsgebundene Formadaption eine Verbesserung im Laufzeitverhalten der Gleichungslöseralgorithmen. Die erfolgreiche Verbesserung der kleinsten Winkeleigenschaft wurde anhand statistischer Untersuchungen mittels einer Beispielsanwendung bewertet (vgl. Abschnitt 4.3.1).

Die Lastverteilung und im Speziellen die Migration der Objekte stellt innerhalb einer numerischen Simulationsumgebung ebenfalls ein Modul dar, welches einen massiven Eingriff in die Verwaltungsstruktur des Objektmodells fordert. In dieser Arbeit wurden daher für das entwickelte verteilte Objektmodell alle Phasen, die während einer Lastverteilung in einer parallelen FEM-Simulation auftreten, analysiert und in Algorithmenform präsentiert (vgl. Abschnitt 3.4). Dabei stellte sich heraus, dass die Konsistenzwahrung während der Migrationsphase aus datentechnischer Sicht den aufwändigsten Teil dieses Moduls darstellt. Um das geforderte Skalierungsverhalten im massiv parallelen Fall zu erhalten, ist es daher notwendig – neben einer für diesen Fall effizienten Methode zur Lastausgleichsberechnung – an dieser Stelle ebenfalls ein effizientes Verfahren einzusetzen. Dies wurde im verteilten Objektmodell mittels der Migrations- und Rekonstruktionsfunktionen umgesetzt und das Verhalten in einer Beispielsanwendung demonstriert.

Eine Einzelbewertung der drei Hauptsäulen einer numerischen FEM-Simulationsumgebung ist für sich betrachtet zwar sinnvoll, besitzt aber letztendlich für eine reale Anwendung aufgrund des dynamischen Verhaltens der Problemstellung und des verwendeten Rechensystems keine große Aussagekraft. Daher wurde eine aus der Literatur bekannte Beispielsanwendung (vgl. Abschnitt 4.5) zu einer Gesamtbewertung von Objektmodell, Algorithmen und Datenstrukturen herangezogen. Hierbei zeigte es sich, dass in einer realen Langzeitanwendung die Ergebnisse mit Ergebnissen künstlicher Benchmarks (s.o.) der Einzelkomponenten durchaus vergleichbar sind.

Die FEM-Simulationsumgebung *padfem*^{2} (vgl. Abschnitt 4.1), die das verteilte Objektmodell und sämtliche in dieser Arbeit vorgestellten Algorithmen und Datenstrukturen praktisch umsetzt, stellt ein für den massiv parallelen Einsatz konzipiertes Baukastensystem dar. Es diente als Beitrag zum gelungenen Abschluss des Sonderforschungsbereiches SFB 376 *Massive Parallelität*, Teilprojekt A3, bei der Untersuchung und Bewertung von neuen Lastverteilungsmethoden und -algorithmen. Darüber hinaus bietet *padfem*^{2} durch sein Softwaredesign und seiner parallelen Konzepte eine ideale Entwicklungs- und Testumgebung für numerische Untersuchungen im Bereich der Strömungsmechanik, welches im Paderborn Center for Parallel Computing (PC^2) an der Universität Paderborn erfolgreich eingesetzt wird.

A Anhang

Inhalt

A.1 Verteilung der Qualitätsmetriken mit Formadaption

Die Grafiken in diesem Abschnitt zeigen die Verteilung der Qualitätsmetriken zu Problemstellung 4.3 für verschiedene Adaptionsschranken mit Einschränkungsparametern.

Abbildung A.1 Verteilung der Qualitätsmetriken mit Adaptionsschranke bzgl. Q_{ar}

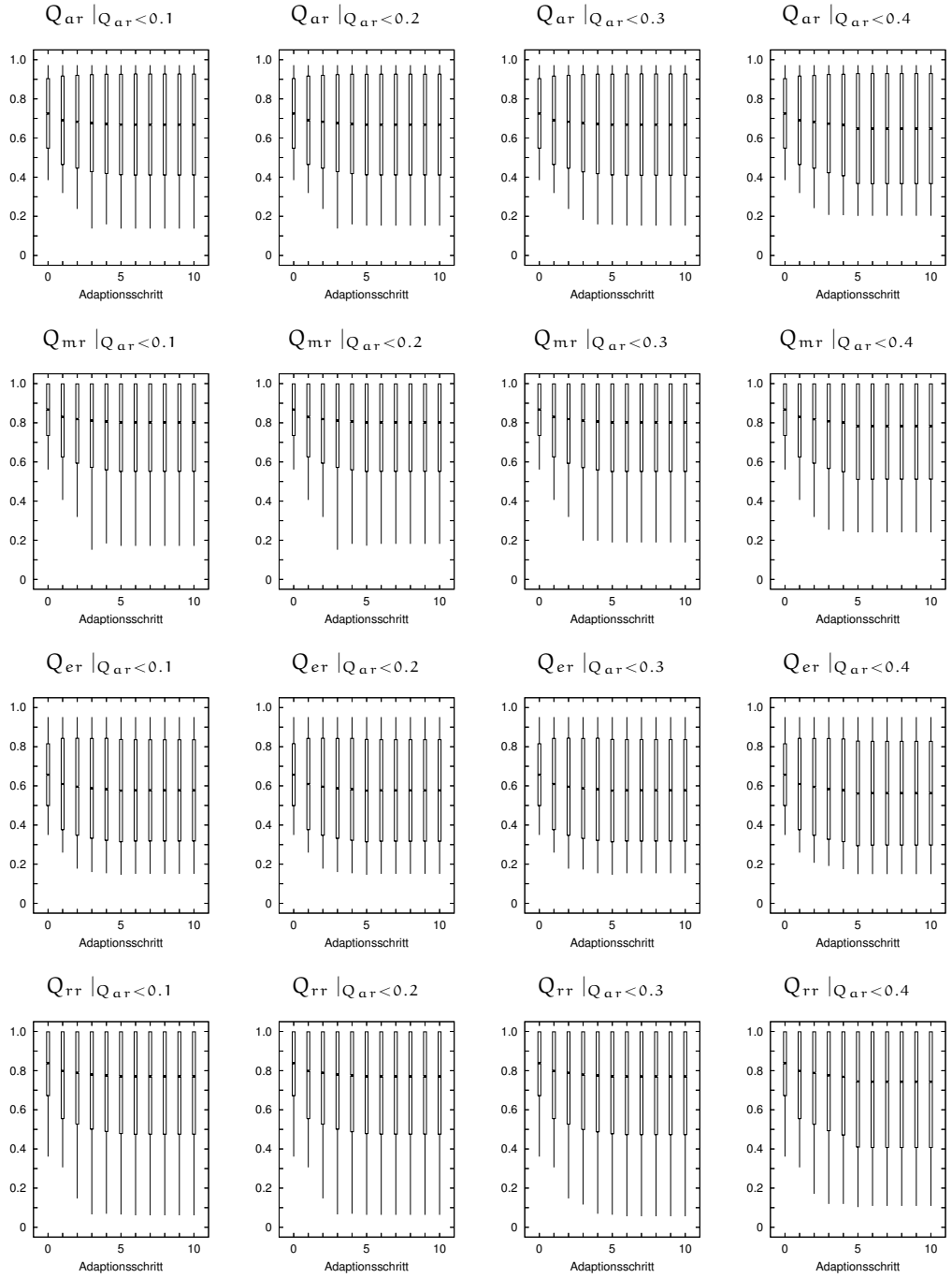


Abbildung A.2 Verteilung der Qualitätsmetriken mit Adaptionsschranke bzgl. Q_{mr}

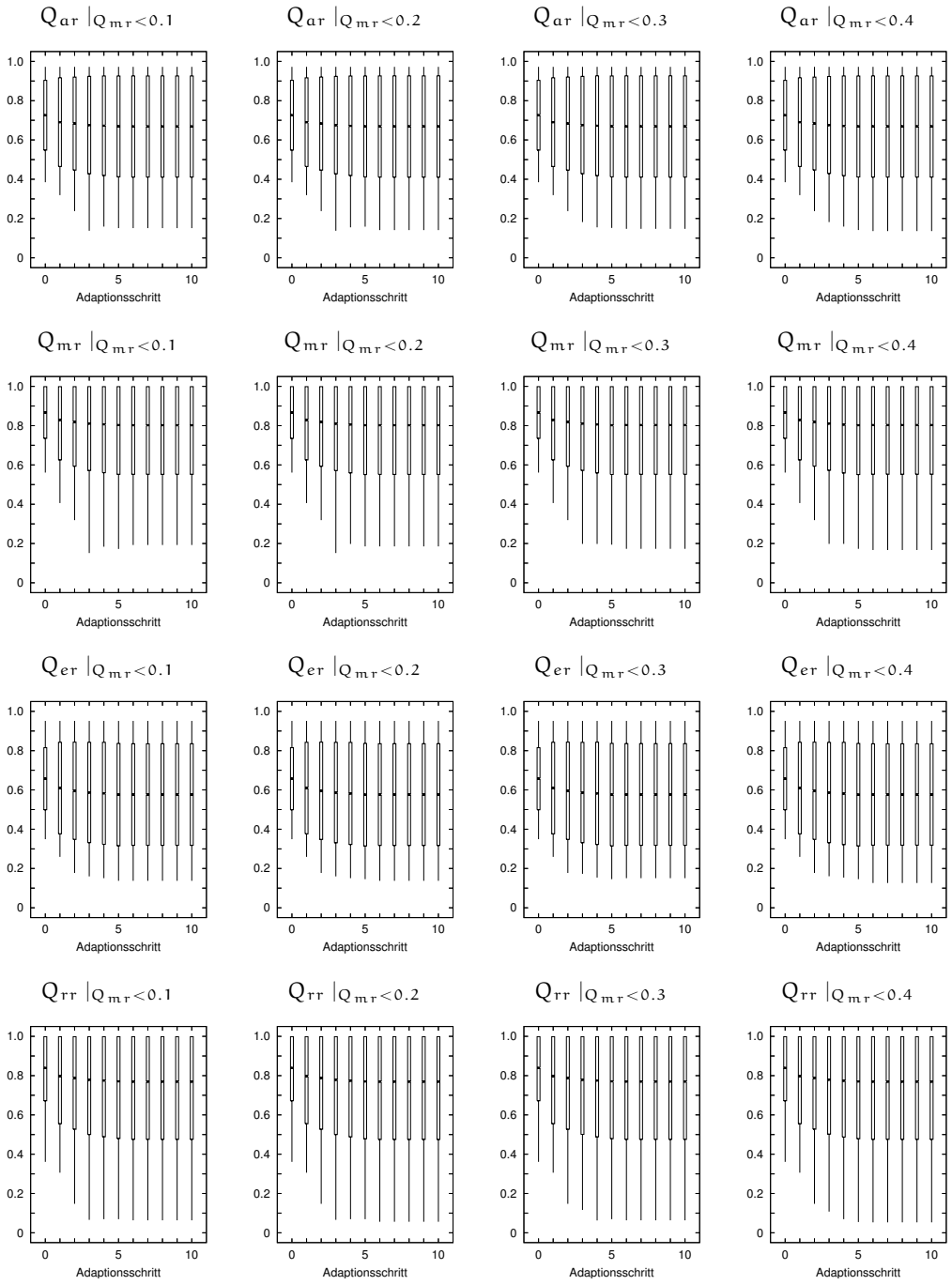


Abbildung A.3 Verteilung der Qualitätsmetriken mit Adaptionsschranke bzgl. Q_{er}

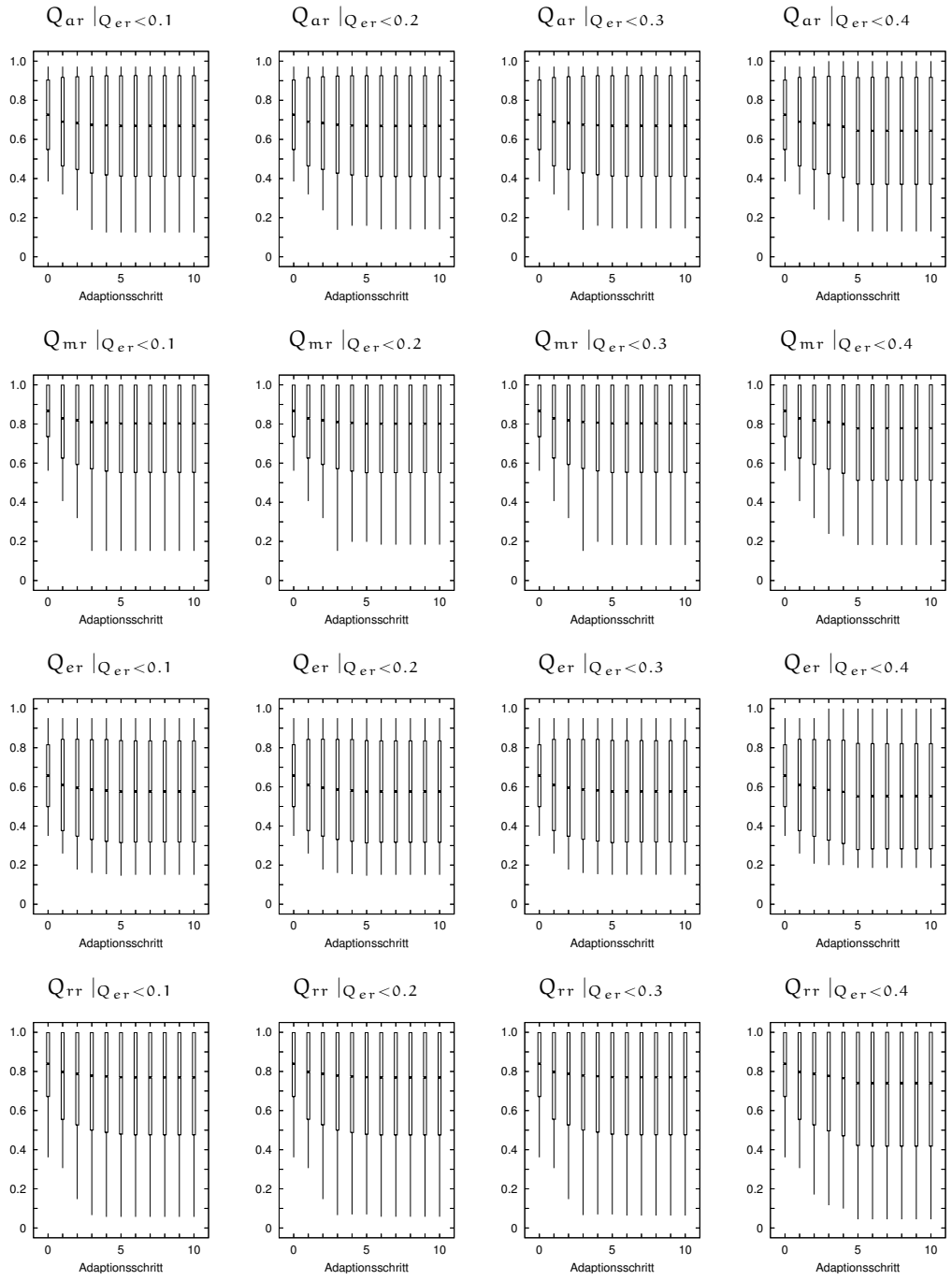
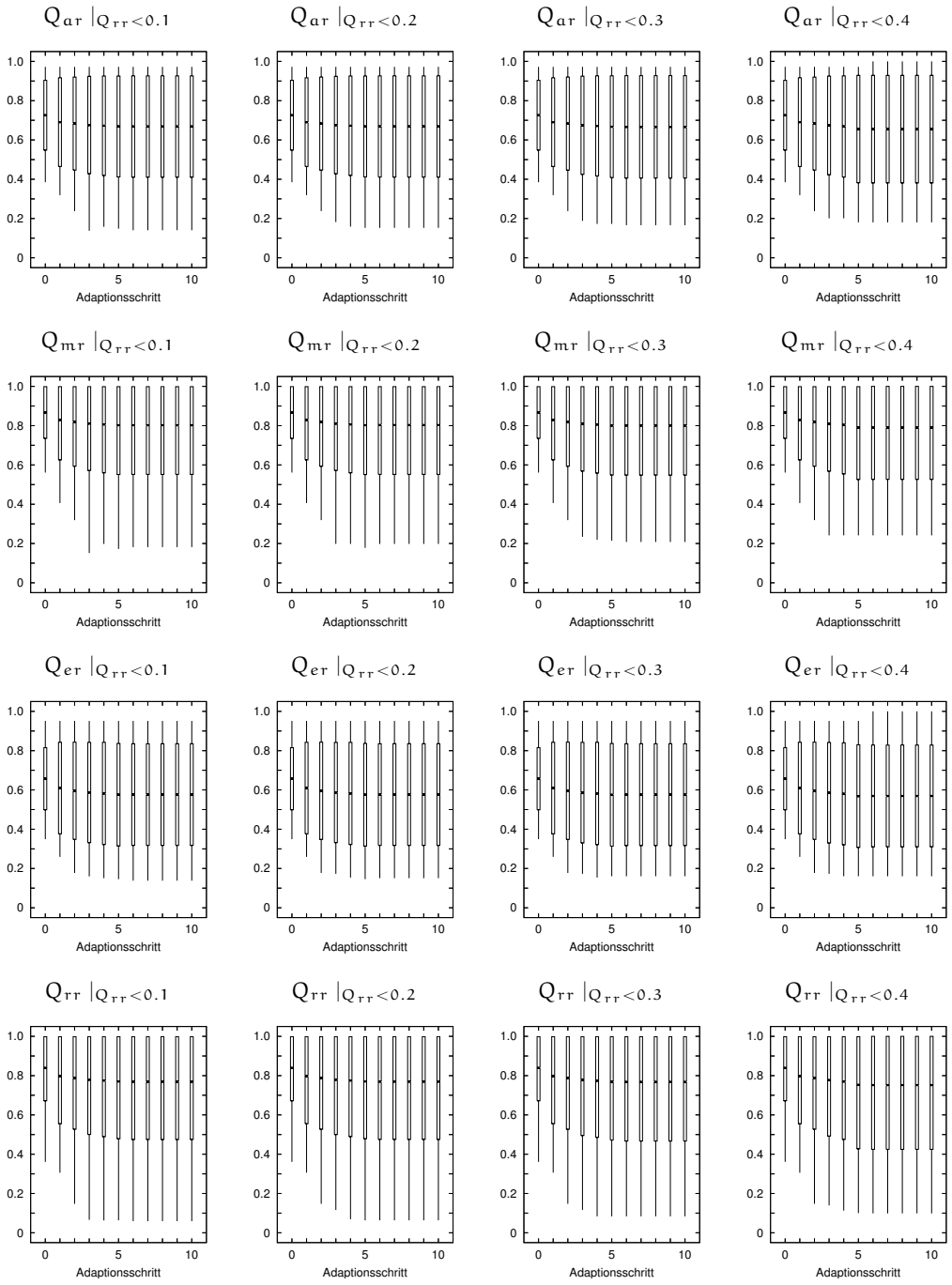


Abbildung A.4 Verteilung der Qualitätsmetriken mit Adaptionsschranke bzgl. Q_{rr}



Algorithmenverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Aki05] J. E. Akin. *Finite element analysis with error estimators: An introduction to the FEM and adaptive error analysis for engineering students.* Oxford Elsevier, 2005.

[ALO02] G. Alefeld, I. Lenhardt, and H. Obermaier. *Parallele numerische Verfahren.* Springer-Verlag Berlin, Heidelberg, New York, 2002.

[ANS07a] Inc ANSYS. Homepage ANSYS: http://www.ansys.com, 2007.

[ANS07b] Inc ANSYS. Homepage CFX: http://www.ansys.com/products/cfx.asp, 2007.

[ANS07c] Inc ANSYS. Homepage Fluent: http://www.fluent.com, 2007.

[BA76] I. Babuska and A. K. Aziz. On the angle condition in the finite element method. *SIAM Journal on Numerical Analysis*, 13(2):214–226, 1976.

[BBB+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[BBD07] P. Bastian, M. Blatt, and A. Dedner. Homepage DUNE: http://www.dune-project.org, 2007.

[BBE+06] P. Bastian, M. Blatt, C. Engwer, A. Dedner, R. Klöfkorn, S. P. Kuttanikkad, and M. Ohlbergerand O. Sander. The distributed and unified numerics environment (DUNE). In *Proc. of the 19th Symposium on Simulation Technique (ASIM)*, 2006.

[BBE+07] S. Balay, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang. *The PETSc Users Manual: Version 2.3.3.* Argonne National Laboratory, 2007.

[BBJ+97] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert, and C. Wieners. UG – a flexible software toolbox for solving partial differential equations, 1997.

[BCER95] M. Bern, L. P. Chew, D. Eppstein, and J. Ruppert. Dihedral bounds for mesh generation in high dimensions. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms).*, 1995.

[BD03] T. J. Barth and H. Deconinck. *Error estimation and adaptive discretization methods in computational fluid dynamics.* Springer-Verlag Berlin, New York, 2003.

[Bey81] W. H. Beyer. *CRC Standard Mathematical Tables.* CRC Press, Boca Raton, Florida, 26th edition, 1981.

[Bey95] J. Bey. Tetrahedral grid refinement. *Computing*, 55:355–378, 1995.

[Bey98] J. Bey. *Finite-Volumen- und Mehrgitter-Verfahren für elliptische Randwertprobleme.* B. G. Teubner Verlag Stuttgart, 1998.

[Bey00] J. Bey. Simplicial grid refinement: On Freudenthal's algorithm and the optimal number of congruence classes, 2000.

[Bir98] K. Birken. *Ein Modell zur effizienten Parallelisierung von Algorithmen auf komplexen, dynamischen Datenstrukturen.* PhD thesis, Universität Stuttgart, 1998.

[BJ93] U. Breitschuh and R. Jurisch. *Die Finite-Element-Methode.* Akademie-Verlag, 1993.

[BKM03] S. Blazy, O. Kao, and O. Marquardt. padfem^2 – An Efficient, Comfortable Framework for Massively Parallel FEM-Applications. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Proc. of the European PVM/MPI User's Group Meeting (EuroPVM/MPI)*, volume 2840 of *LNCS*, pages 681–685. Springer-Verlag, Sep. 2003.

[BKM04a] S. Blazy, O. Kao, and O. Marquardt. padviz – A Prototype for Visualization of Three Dimensional Fluid Flow on Unstructured Grids. In H. R. Arabnia, editor, *International Conference on Imaging Science, Systems, and Technology (CISST)*, pages 237–240. CSREA, Jun. 2004.

[BKM04b] S. Blazy, O. Kao, and O. Marquardt. pmf – A Comfortable, Modular Framework for Parallel Meshing Algorithms. In H. R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 648–651. CSREA, Jun. 2004.

[Blu90] N. Blum. A new approach to maximum matching in general graphs. In *ICALP*, pages 586–597, 1990.

[BM03] S. Blazy and O. Marquardt. A Characteristic Algorithm for the 3D Navier-Stokes Equation using padfem^2. In T. Gonzalez, editor, *Proc. of the 15th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems (PDCS)*, volume 2 of *IASTED*, pages 745–750. ACTA Press, Nov. 2003.

[BM05a] S. Blazy and O. Marquardt. Parallel finite element computations of three-dimensional benchmark problems. In M. Kowarschik F. Hülsemann and U. Rüde, editors, *Proc. of the 18th Symposium on Simulationstechniques (ASIM)*, pages 152–157, 2005.

[BM05b] S. Blazy and O. Marquardt. Parallel refinement of tetrahedral meshes on distributed-memory machines. In T. Fahringer and M.H. Hamza, editors, *Proc. of the 23th IASTED Intl. Conf. on Parallel and Distributed Computing and Networks (PDCN)*, volume 1 of *IASTED*, pages 686–692. ACTA Press, Feb. 2005.

[BM06] S. Blazy and O. Marquardt. Parallel computation of three-dimensional flows on unstructured grids. In F. Meyer auf der Heide and B. Monien, editors, *New Trends in Parallel & Distributed Computing*, volume 181, pages 63–80. HNI-Verlagsschriftenreihe, Jan. 2006.

[BM07] S. Blazy and O. Marquardt. Parallel finite element computations of three-dimensional flow problems using padfem^2. *International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)*, 22:257–274, 2007.

[Boa07] OpenMP Architecture Review Board. Homepage OpenMP specification: http://www.openmp.org, 2007.

[Bra94] A. Brandstädt. *Graphen und Algorithmen.* B.G. Teubner Stuttgart Verlag, 1994.

[Bra03] D. Braess. *Finite Elemente.* Springer-Verlag, 2003.

[Ca07] CD-adapco. Homepage Star-CD: http://www.cd-adapco.com/products/star-cd/index.html, 2007.

[Car97] G. F. Carey. *Computational Grids.* Taylor & Francis, 1997.

[CDE+99] S.-W. Chen, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng. Sliver exudation. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry.*, pages 1–13. ACM Press, 1999.

[Cha00] R. Chandra. *Parallel Programming with OpenMP.* Morgan Kaufmann Publishers, 2000.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[DDNR96] R. Diekmann, U. Dralle, F. Neugebauer, and T. Romke. Padfem: A portable parallel fem-tool. In *Proc. Int. Conf. High-Performance Computing and Networking (HPCN Europe)*, pages 580–585, Apr. 1996.

[DHBW04] F. Deserno, G. Hager, F. Brechtefeld, and G. Wellein. Performance of scientific applications on modern supercomputers. *High Performance Computing in Science and Engineering*, pages 339–348, 2004.

[Die98] R. Diekmann. *Load Balancing Strategies for Data Parallel Applications*. PhD thesis, Universität-GH Paderborn, 1998.

[DLGC98] J. Dompierre, P. Labbé, F. Guibault, and R. Camarero. Benchmarks for 3d unstructured tetrahedral mesh optimization. In *IMR*, pages 459–478, 1998.

[DMP95] R. Diekmann, B. Monien, and R. Preis. Using helpful sets to improve graph bisections. In *Interconnection Networks and Mapping and Scheduling Parallel Computations*, volume 21 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 57–73. AMS, 1995.

[Don87] J. Dongarra. The LINPACK Benchmark: An Explanation. In *ICS*, pages 456–474, 1987.

[DP97] R. Diekmann and R. Preis. Party: A software library for graph partitioning. *Advances in Computational Mechanics with Parallel and Distributed Processing*, pages 63–71, 1997.

[Dut93] S. Dutt. New faster kerninghan-lin-type graph-partitioning algorithms. In *Proc. of IEEE/ACM International Conference on CAD*, pages 370–377, 1993.

[Edm65] J. Edmonds. Paths, trees and flowers. In *Canad. J. Math.*, volume 17, pages 449–467, 1965.

[Els97] Ulrich Elsner. Graph partitioning - a survey, 1997.

[Fie73] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Math. J.*, 23:298–305, 1973.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. *Proc. of the 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[Gad52] J. W. Gaddum. The sum of the dihedral and trihedral angles in a tetrahedron. In *Amer. Math. Monthly*, volume 59, pages 370–371, 1952.

[GB98] P. L. George and H. Borouchaki. *Delaunay Triangulation and Meshing – Application to Finite Elements.* Hermes, Paris, 1998.

[GFB+04] E. G., G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Interactability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[GL96] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The John Hopkins University Press, Baltimore, 3rd edition edition, 1996.

[GO96] G. H. Golub and J. M. Ortega. *Scientific Computing.* Teubner Verlag Stuttgart, 1996.

[GPRR02] S. Groß, J. Peters, V. Reichelt, and A. Reusken. The DROPS package for numerical simulations of incompressible flows using parallel adaptive multigrid techniques. Technical Report IGPM-Report 211, RWTH Aachen, 2002.

[Hac76] W. Hackbusch. A fast iterative method for solving poisson's equation in a general domain. In R Bulirsch, R. D. Grigorieff, and J. Schröder, editors, *Proc. Oberwolfach*, pages 51–62. Springer-Verlag Berlin, Heidelberg, New York, 1976.

[Hac85] W. Hackbusch. *Multi-Grid Methods and Applications.* Springer-Verlag Berlin, Heidelberg, 1985.

[Hac93] W. Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme.* 2.,überarb. Aufl. Teubner Verlag, 1993.

[Hei07] IWR Heidelberg. Homepage UG: http://sit.iwr.uni-heidelberg.de/~ug, 2007.

[Hen98] Bruce Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? (extended abstract). In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 218–225, 1998.

[HM91] J. Hromkovic and B. Monien. The bisection problem for graphs of degree 4 (configuring transputer systems). In *Proc. of the 16th Conf. on Mathematical Foundations of Computer Science (MFCS)*, volume 520 of *LNCS*, pages 211–220. Springer-Verlag, 1991.

[HW07] M. A. Heroux and J. M. Willenbring. *The Trilinos Users Guide*. Sandia National Laboratories, 2007.

[JL01] M. Jung and U. Langer. *Methode der finiten Elemente für Ingenieure*. Teubner Verlag, 2001.

[Kit07] Inc. Kitware. Homepage VTK: http://www.vtk.org, 2007.

[KK97] G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph-partitioning algorithm. In *Proc. of the 8th SIAM Conf. on Parallel Processing for Scientific Computing*, 1997.

[KK98a] G. Karypis and V. Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. University of Minnesota, Army HPC Research Center, 1998.

[KK98b] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[KK99] G. Karypis and V. Kumar. Parallel multilevel series k-way partitioning scheme for irregular graphs. *SIAM Rev.*, 41(2):278–300, 1999.

[KL70] B. W. Kernighan and S. Lin. An effective heuristic procedure for partitioning graphs. *The Bell Systems Technical Journal*, pages 291–308, 1970.

[Knu01] P. M. Knupp. Algebraic mesh quality metrics. *SIAM J. Sci. Comput.*, 23(1):193–218, 2001.

[Kri92] M. Krizek. On the maximum angle condition for linear tetrahedral elements. *SIAM Journal on Numerical Analysis*, 29(2):513–520, 1992.

[Lab07] Sandia National Laboratories. Homepage Trilinos: http://trilinos.sandia.gov, 2007.

[Lis99] V. D. Liseikin. *Grid Generation Methods*. Springer-Verlag, 1999.

[LJ94a] A. Liu and B. Joe. On the shape of tetrahedra from bisection. In *Math. of Comp.*, volume 63, pages 141–154, 1994.

[LJ94b] A. Liu and B. Joe. Relationship between tetrahedra shape measures. In *BIT*, volume 34, pages 268–287, 1994.

[Ltd07] OpenCFD Ltd. Homepage OpenFOAM: http://www.opencfd.co.uk/openfoam, 2007.

[Löh90] R. Löhner. Three-dimensional fluid-structure interaction using a finite element solver and adaptive remeshing. *Computer Systems in Engineering*, 1(2-4):257–272, 1990.

[Löh01] R. Löhner. *Applied CFD Techniques*. John Wiley & Sons Ltd., 2001.

[Lül96] R. Lüling. *Lastverteilungsverfahren zur effizienten Nutzung paralleler Systeme*. PhD thesis, Universität-GH Paderborn, 1996.

[McC07] J. D. McCalpin. Homepage STREAM benchmark: http://www.cs.virginia.edu/stream, 2007.

[McM86] F. H. McMahon. The Livermore Fortran kernels: A computer test of the numerical performance range. Technical Report Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[Mei05] A. Meister. *Numerik linearer Gleichungssysteme, Eine Einführung in moderne Verfahren*. Vieweg Verlag, Braunschweig, Wiesbaden, 2. edition, 2005.

[MGN95] T. Dornseifer M. Griebel and T. Neunhoeffer. *Numerische Simulation in der Strömungsmechanik*. Vieweg Lehrbuch Verlag, 1995.

[MMS06] H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating shape optimizing load balancing for parallel FEM simulations by algebraic multigrid. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, (IPDPS'06)*, page 57 (CD). IEEE Computer Society, 2006.

[MS96] S. Turek M. Schaefer. Benchmark computations of laminar flow around cylinder. In *Flow Simulation with High-Performance Computers II*, volume 52 of *Notes on Numerical Fluid Mechanics*, pages 547–566, Apr. 1996.

[MV80] S. Micali and V. V. Vazirani. An $\mathcal{O}(\sqrt{V} \cdot E)$ algorithm for finding maximum matching in general graphs. In *FOCS*, pages 17–27, 1980.

[Nak03] K. Nakajima. OpenMP/MPI hybrid vs. flat MPI on the earth simulator: Iterative solvers for finite element method. *Proc. of the 5th Intl. Symposium on High Performance Computing, ISHPC*, 2858:486–499, 2003.

[NL07] C. D. Norton and J. Lou. Homepage Pyramid: http://www.openchannelfoundation.org/projects/pyramid, 2007.

[NLPT04] C. D. Norton, G. Lyzenga, J. Parker, and R. E. Tisdale. Developing parallel GeoFEST(P) using the PYRAMID AMR library. In *Proc. Earth Science Technology Conference*. Jet Propulsion Laboratory, NASA, Pasadena, CA, 2004.

[OL01] R. L. Ott and M. Longnecker. *An Introduction to Statistical Methods and Data Analysis*. Thomson Learning, 5th edition, 2001.

[OLHB02] L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Rev.*, 44(3):373–393, 2002.

[OW90] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen.* B.I. Hochschultaschenbücher : Reihe Informatik. Bibliographisches Institut, 1990.

[PL88] P. A. Peterson and M. C. Loui. The general maximum matching algorithm of Micali and Vazirani. *Algorithmica*, 3:511–533, 1988.

[Pre98] R. Preis. *The PARTY Graph Partitioning Library – User Manual.* 1998.

[Pre00] R. Preis. *Analyses and design of efficient graph partitioning methods.* PhD thesis, Universität Paderborn, 2000.

[PSL90] A. Pothen, H. D. Simon, and K. P. Liu. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matr. ANal. Appl.*, 11:430–452, 1990.

[Rab03] R. Rabenseifner. Hybrid parallel programming: Performance problems and chances. *Proc. of the 45th CUG Conference, Columbus, Ohio*, 2003.

[RB03] R. Rannacher and W. Bangerth. *Adaptive Finite Element Methods for Differential Equations.* Birkhäuser Verlag, 2003.

[RR00] T. Rauber and G. Rünger. *Parallele und verteilte Programmierung.* Springer-Verlag Berlin, Heidelberg, New York, 2000.

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[Sag94] H. Sagan. *Space Filling Curves.* Springer-Verlag, 1994.

[SBK+07] B. Smith, S. Balay, M. Knepley, H. Zhang, V. Eijkhout, D. Karpeev, and L. Dalcin. Homepage PETSC: http://www-unix.mcs.anl.gov/petsc/petsc-as, 2007.

[Sch90] B. E. Schönung. *Numerische Strömungsmechanik.* Springer-Verlag, 1990.

[Sch00] J. Schulze. *Faktorisierung dünn besetzter, positiv definiter Matrizen.* PhD thesis, Universität Paderborn, 2000.

[Sch06] S. Schamberger. *Shape Optimized Graph Partitioning.* PhD thesis, Department of Computer Science, University of Paderborn, 2006.

[Sed02] R. Sedgewick. *Algorithmen in C++.* Pearson Studium, 2002.

[SG97] A. Silberschatz and P. B. Galvin. *Operating System Concepts.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[SGG07] J. Schöberl, J. Gerstmayr, and R. Gaisbauer. Homepage NETGEN: http://www.hpfem.jku.at/netgen, 2007.

[She94] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

[She97] J. R. Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1997.

[Si04] H. Si. Tetgen: A quality tetrahedral mesh generator and three-dimensional delaunay triangulator. Technical Report 9, Weierstraß-Institut für Angewandte Analysis und Stochastik (WIAS), 2004.

[Si07] H. Si. Homepage Tetgen: http://tetgen.berlios.de, 2007.

[Sim07] J. Simon. Homepage PC^2 Benchmarking Center: http://wwwcs.upb.de/pc2/about-us/staff/jens-simons-pages/benchmarkingcenter.html, 2007.

[SK04] H. R. Schwarz and N. Köckler. *Numerische Mathematik*. 5.,überarb. Aufl. Teubner Verlag, 2004.

[SML04] W. Schröder, K. Martin, and B. Lorensen. *The Visualization Toolkit – An Object-Oriented Approach to 3D Graphics*. Kitware, Inc., 3rd edition, 2004.

[SO98] M. Snir and S. Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[SS05] A. Schmidt and K. G. Siebert. *Design of Adaptive Finite Element Software, The Finite Element Toolbox ALBERTA*. Lecture Notes in Computational Science and Engineering. Springer, Berlin, Heidelberg, New York, 2005.

[SSK+07] A. Schmidt, K. G. Siebert, D. Köster, O. Kriessl, and C.-J. Heine. Homepage ALBERTA: http://www.alberta-fem.de, 2007.

[Str92] W. A. Strauss. *Partielle Differentialgleichungen – Eine Einführung*. Vieweg Verlag, 1992.

[SW03] S. Schamberger and J. M. Wierum. Graph partitioning in scientific simulations: Multilevel schemes vs. space-filling curves. In *Parallel Computing Technologies, PACT*, volume 2763 of *LNCS*, pages 165–179. Springer-Verlag, 2003.

[SW05] M. Silva and R. Wait. Sparse matrix storage revisited. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 230–235, New York, NY, USA, 2005. ACM Press.

[Tan01] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[TSW99] J. F. Thompson, B. K. Soni, and N. P. Weatherill. *Handbook of Grid Generation.* CRC Press, 1999.

[UPS07] Inc. UGS PLM Software. Homepage I-deas: http://www.ugs.com/products/nx/ideas, 2007.

[Vaz94] V. V. Vazirani. A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V} \cdot E)$ general graph maximum matching algorithm. *Combinatorica*, 14(1):71–109, 1994.

[Vud03] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels.* PhD thesis, University of California, Berkeley, December 2003.

[Wal02] C. Walshaw. *The Parallel JOSTLE Library User Guide: Version 3.0.* University of Greenwich, London, 2002.

[Wei84] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.

[Wie03] J. M. Wierum. *Anwendung diskreter raumfüllender Kurven: Graph-partitionierung und Kontaktsuche in der Finite-Elemente-Simulation.* PhD thesis, Universität Paderborn, 2003.

[www07a] Homepage ARMINIUS/PC2: http://www.upb.de/pc2/services/systems/arminius.html, 2007.

[www07b] Homepage INTEL MPI Benchmark: http://www.intel.com/cd/software/products/asmo-na/eng/307696.htm#mpibenchmarks, 2007.

[www07c] Homepage PC2: http://www.upb.de/pc2, 2007.

[www07d] Homepage Standard Performance Evaluation Corporation SPEC: http://www.spec.org, 2007.

[www07e] Homepage Top 500: http://www.top500.org, 2007.

[www07f] Homepage TPC Benchmarks: http://www.tpc.org, 2007.

[www07g] Homepage Whetstone Benchmark: http://www.cse.scitech.ac.uk/disco/benchmarks/whetstone.shtml, 2007.

[Zha95] S. Zhang. Successive subdivisions of tetrahedra and multigrid methods on tetrahedral meshes. *Houston J. Math.*, 21:541–556, 1995.

[Zum01] G. W. Zumbusch. On the quality of space-filling curve induced partitions. *Z. Angew. Math. Mech.*, 81:25–28, 2001.

[Zum03] G. W. Zumbusch. *Parallel Multilevel Methods*. 1. Auflage. Teubner Verlag, 2003.

Erklärung

Ich versichere hiermit, dass ich diese Arbeit selbständig und ohne fremde Hilfe angefertigt habe. Es wurden von mir keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt.

Paderborn, September 2007
