Ein Programm für die Parallelisierung dynamisch adaptiver Mehrgitterverfahren

Inaugural - Dissertation

zur

Erlangung des Doktorgrades

der Mathematisch-Naturwissenschaftlichen Fakultät

der Universität zu Köln

vorgelegt von

Ekkehart Kiparski

aus Reichshof

Berichterstatter: Prof. Dr. Ewald Speckenmeyer
(Gutachter)
Prof. Dr. Hubert Randerath

Tag der mündlichen Prüfung: 13.01.2016

Danksagungen

Mein besonderer Dank gilt meinem Doktorvater Prof. Dr. E. Speckenmeyer für die Möglichkeit, an dieser Promotion zu arbeiten, für ein anspruchsvolles Thema, bei dem ich viel gelernt habe, für seine Betreuung, seine Hinweise und sein Eingehen auf meine Fragen sowie seine Hilfe bei formalen Hürden.

Ferner danke ich meinem Zweitgutachter Prof. Dr. H. Randerath für die Begutachtung dieser Arbeit.

Daneben gilt mein Dank Dr. F. Meisgen für seine Betreuung der Arbeit in der ersten Phase meiner Arbeit, mit wichtigen Anregungen, sowie dem Vertrautmachen mit dem Konzept PLB, das am Lehrstuhl entwickelt wurde.

Dazu danke ich den Mitarbeitern des RRZK für den Zugang zu CHEOPS.

Schließlich möchte ich meinen Eltern danken, die mir durch ihre Unterstützung die Arbeit an diesem Thema ermöglicht haben.

Inhaltsverzeichnis

Zusammmenfassung

In dieser Arbeit werden dynamisch adaptive Mehrgitterverfahren parallelisiert. Bei dynamisch adaptiven Mehrgitterverfahren wird ein Gebiet mit einem Gitter überdeckt, und auf diesem Gitter wird gerechnet, indem Gitterpunkte in der Umgebung herangezogen werden, um den Wert des nächsten Zeitpunktes zu bestimmen. Dann werden gröbere und feinere Gitter erzeugt und verwendet, wobei die feineren Gitter sich auf Teilgebiete konzentrieren. Diese Teilgebiete ändern sich im Verlauf der Zeit. Durch die Verwendung der zusätzlichen Gitter werden die numerischen Eigenschaften verbessert. Die Parallelisierung solcher Verfahren geschieht in der Regel durch Bisektion. In der vorliegenden Arbeit wird die Umverteilung der Gebiete realisiert, indem Mengen von einzelnen Gitterpunkten verschickt werden. Das ist ein Scheduling-Verfahren.
Die Mehrgitterstrukturen sind so aufgebaut, dass fast beliebige Gitterpunktverteilungen auf den Gitterebenen vorliegen können. Die Strukturen werden einmal erzeugt, und nur bei Bedarf geändert, sodass keine Speicherallokationen während der Iterationen nötig sind. Neben dem Gitter sind zusätzliche Strukturen, wie zum Beispiel die Randstrukturen, erforderlich. Eine Struktur Farbenfeld verzeichnet, auf welchem Kern sich ein Außenrandpunkt befindet.
In der parallelen adaptiven Verfeinerung werden für einzelne durch ein Entscheidungskriterium ausgewählte Gitterpunkte 5 x 5 Punktüberdeckungen vorgenommen. Dazu werden die verfügbaren Entscheidungsinformationen zur Bestimmung von komplexeren Strukturen herangezogen. Damit muss das Verfeinerungsgitter nicht komplett abgebaut und dann wieder aufgebaut werden, sondern nur die Änderungen am Gitter sind vorzunehmen. Das spart viel Berechnungszeit.
Der letzte Schritt besteht darin, den Lastausgleich durchzuführen. Zunächst werden die Lasttransferwerte bestimmt, die angeben, wie viele Gitterpunkte von wo nach wo zu verschicken sind. Das geschieht mit Hilfe einer PLB genannten Methode bzw. einer Variante. PLB wurde bisher vor allem für kombinatorische Probleme eingesetzt. Dann erfolgt eine Auswahl der zu verschickenden Gitterpunkte mit einer Strategie, welche Punkte eines Kerns zu welchen Nachbarkernen transferiert werden sollen. Im letzten Schritt werden schließlich die ausgewählten Punkte migriert, wobei alle Gitterpunktstrukturen umgebaut werden und solche Informationen gepackt werden müssen, sodass ein Umbau seiner Gitterpunktstrukturen bei dem Empfänger möglich wird. Neben den Gitterpunktstrukturen müssen auch Strukturen für die parallele adaptive Verfeinerung verändert werden. Es muss ein Weiterverschicken von Gitterpunkten möglich sein, wenn über die Lastkanten in mehreren Runden Last verschickt wird. Während des Lastausgleichs wird noch Arbeit durch eine Struktur Zwischenkorrektur durchgeführt, die es ermöglicht, das Farbenfeld intakt zu halten, wenn benachbarte Gitterpunkte gleichzeitig verschickt werden.

Abstract

In this thesis dynamic adaptive multi-grid methods are parallelized. In dynamic adaptive multi-grid methods a domain is covered by a grid, and this grid is utilized to calculate objective function values. The objective function at a specific grid point at time t is calculated according to the values of this point and its neighbors at time t-1. The accuracy of this method can be improved by applying coarser and finer grids, where the latter concentrate on subdomains which may change over time.

The parallelization of such methods usually is conducted by bisection of the grid and distributed calculation of the partial objective function values. In this thesis a scheduling approach for the redistribution of partial domains by transferring sets of grid points is presented.

The design of the applied multi-grid structures supports almost any distribution of grid points. The grids' data structures are generated once and changed only when required. Thus, no dynamic memory allocation is necessary during iterations. In addition to the multi-grid further structures are necessary, e.g. a structure called Farbenfeld (color field) records on which CPU core certain boundary points of the grid are located.

During the parallel adaptive refinement phase a decision criterion is applied in order to create 5x5 fine grid points for one coarse grid point fullfilling the criterion. This is performed using complex data structures only necessitating local modifications to the grid, significantly reducing computation time.

Lastly, load balancing is performed by calculating Lasttransferwerte (load transfer values), which specify the number of grid points to be send from one core to another. For this purpose the PLB method and a variant of it are utilized, which usually are applied to complex combinatorial optimization problems. Subsequently, the specific grid points to be transferred from one core to its neighboring cores are chosen and the necessary information for restructuring is send to the affected CPU cores. In addition to the multi-grid structures the structures used by parallel adaptive refinement phase need to be modified. If load is transferred in multiple consecutive load balancing phases, grid points need to be passed on to other neighboring cores. If neighboring grid points are migrated simultaneously during load balancing, a data structure Zwischenkorrektur (intermediate correction) is utilized to keep intact the Farbenfeld.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Die Gitterebenen

h_gesamt	Anzahl aller Gitterebenen
h_gesamt_variante	h_gesamt oder h_cap, je nachdem, ob eine cap existiert
h_cap	Gitterebene der cap (sonst -1)
h_min	Anzahl an Verfeinerungsebenen
h_max	Anzahl an Vergröberungsebenen

Konzepte

cap	Konzept zur Behandlung der Parallelität bei vielen Vergröberungsstufen, Abschnitt 4.5
LBC	Load Balancing Classes
MLAT	Multi-Level-Adaptive-Technics Abschnitt 3.5.3
MPI	Message Passing Interface: Ein paralleles Programmierkonzept Abschnitt 2.1.3
PLB	Precomputation-based Load Balancing Ein Parallelisierungskonzept Abschnitt 6.2

Laufzeitmessungen

Abschnitt 1.1.6

Eff	Effizienz
S	Speedup
T_p	parallele Laufzeit
T_s	sequentielle Laufzeit

Die Bestandteile der Mehrgitterverfahren

Abschnitt 2.2

G_h	Das feine Gitter

G_H	Das grobe Gitter
Grundgitter	Darüber liegen die Vergröberungen, darunter die Verfeinerungen
Prolongation I_H^h	Bringen von Werten des gröberen Gitters auf das feinere Gitter
Residuum r_h	Berechnung von f_h - $L_h u_h$
Restriktion I_h^H	Bringen von Werten des feineren Gitters auf das gröbere Gitter

Die Variablen der Mehrgitterverfahren

Abschnitt 2.2

e	Die Variable der Grobgitterkorrektur
f	Die Funktionswerte beim Mehrgitterverfahren
r	Das Residuum
rS	Die rechte Seite beim FAS
u	Die Lösungsvariable

Himmelsrichtungen

EHR	Erweiterte Himmelsrichtungen (N,NO,O,SO,S,SW,W,NW,Z)
HR	Himmelsrichtungen (N,NO,O,SO,S,SW,W,NW)
HR-Zahl	Eine Zahl zwischen 0 und 255, welche für alle 8 Himmelsrichtungen ein Flag setzt Abschnitt 4.2.3
NHR	Normale Himmelsrichtungen (N,O,S,W) Abschnitt 5.7.1
NNHR	Nicht normale Himmelsrichtungen (NO,NW,SO,SW) Abschnitt 5.7.1
OSWN	Die Himmelsrichtungen Ost, Süd, West und Nord
OW	Die Himmelsrichtungen Ost und West
NS	Die Himmelsrichtungen Nord und Süd

Mengen von Himmelsrichtungen

MEHR	Menge der erweiterten Himmelsrichtungen Abschnitt 5.3.1
MHR	Menge der Himmelsrichtungen Abschnitt 4.2.3

Zur Identifizierung der Kerne

Farbe	Die Identität eines Kerns. Sie nimmt den Wert -1 an, wenn sie ein Feld kennzeichnet, auf dem sich kein Gitterpunkt befindet. Abschnitt 3.4.1
Größe	Anzahl der Kerne, die das Mehrgitterprogramm abarbeiten
Rang	Die Identität des eigenen Kerns. Dieser Wert ist immer im Bereich 0 bis Größe - 1

Strukturen der Klasse *Gitter*

Abschnitt 3.4.1

dynamische Randpunktliste	Die Liste zum Durchlaufen des Gitterrandes
farbenfeld	Die Angabe, auf welchem Kern ein Gitterpunkt liegt.
FeldListe	Kombinationsstruktur zum Speichern von Gitterpunkten
selbstnachbarn	Die Zahl zur Bestimmung der Gitterzugehörigkeit von Punkten auf den umliegenden Punkten. Es geht um die Nachbarpunkte ('Nachbarn') auf dem eigenen Kern ('selbst').

Ränder

Abschnitt 3.4.1

AR	Der Außenrand
AR(farbe)	Der Außenrand zum Kern mit der ID farbe
IR	Der Innenrand
IR(farbe)	Der Innenrand zum Kern mit der ID farbe

Randpunktmengen

Abschnitt 3.4.1

HAR(farbe)	horizontale Vektoren für AR(farbe)
HIR(farbe)	horizontale Vektoren für IR(farbe)
PAR(farbe)	verbliebene Punkte aus AR(farbe)
PIR(Farbe)	verbliebene Punkte aus IR(farbe)
VAR(farbe)	vertikale Vektoren für AR(farbe)
VIR(farbe)	vertikale Vektoren für IR(farbe)

Mengen

$M \cup N$	Vereinigungsmenge
$M \cap N$	Schnittmenge

Eigenschaftspunkte

Abschnitt 4.4.1

feld_aktuell	Die aktuelle Information der Eigenschaftspunkte
feld_vergangen	Die vergangene Information der Eigenschaftspunkte
pfl_feld_aktuell	Wie *feld_aktuell*, nur als *Punktfeldliste* gespeichert
pfl_feld_vergangen	Wie *feld_vergangen*, nur als *Punktfeldliste* gespeichert

Die cap

Abschnitt 4.5

sammeln	Sammeln der Gitterpunkte der Kerne nebst der Werte bei einem Master-Kern
zerstreuen	Verteilen der Gitterwerte des Master-Kerns auf alle Kerne

parallele adaptive Verfeinerung
Kapitel 5

EP^+	Die hinzugekommenen Eigenschaftspunkte
EP^-	Die hinweggekommenen Eigenschaftspunkte
EP-SNB	Ein Feld, dass zu einem Punkt die EP-Information auf den umliegenden Punkten angibt
fgff	Das Feingitterfarbenfeld
FU	Farbenupdate: Die Aktualisierung des Farbenfeldes bei der parallelen adptiven Verfeinerung
FU^+	Farbenupdate hinzufügen: Die Aktualisierung des Farbenfeldes wegen hinzukommender Punkte
FU^-	Farbenupdate entfernen: Die Aktualisierung des Farbenfeldes wegen entfernter Punkte
ggff	Das Grobgitterfarbenfeld
Kommunikation 4er	Eine in 3 Blöcke gespaltene Kommunikation, in die die Daten für FU^+, PU^- und FU^- eingetragen werden
M^a	Eine Punktmenge zur Beschreibung von P_b^-
MaxG	Das maximal verfeinerbare Gitter
$P_{5,5}$(f,g)	Die 5 x 5 Punktüberdeckung
P(EHR,f,g)	Der Teil der 5 x 5 Punktüberdeckung in Himmelsrichtung HR Abschnitt 5.3.1
P_k^+	Die hinzukommenden Punkte des Kerns
P_b^+	Die Berechnungformel zu P_k^+
P_k^-	Die zu entfernenden Punkte des Kerns
P_b^-	Die Berechnungsformel zu P_k^-
PU	Punktupdate: Die Aktualisierung der Punktmenge bei der parallelen adptiven Verfeinerung
PU^-	Punktupdate: Die Aktualisierung der Punktmenge wegen entfernter Punkte
PU^+	Punktupdate: Die Aktualisierung der Punktmenge wegen hinzukommender Punkte
Punktfeldliste	Kombinationsstruktur zum Speichern von Gitterpunkten
P_Z(f,g)	Der zentrale Block zum Punkt (f,g) Abschnitt 5.3.1

Lastausgleich-Vorausberechnungen
Abschnitte 6.1 bis 6.5

KTA	Knotenteilanzahl: Die Größe eines an einem Knoten wurzelnden Teilbaums
LTW	Lasttransferwert: Wie viel Last in einer Lastausgleichsrunde fließen soll
PS	Partialsumme
vL	verschickende Last: Wie viel Last zwischen den Knoten für einen vollständigen Lastausgleich fließen sollte
wT	Der an einem Knoten wurzelnde Teilbaum eines Wurzelbaums

Lastausgleich - Migration - Strategie

Abschnitte 6.8 und 6.7

ev	Die Punktmenge des groben Gitters für einen einfachen Lastausgleich
ev2	Die Punktmenge des groben Gitters für einen reversen Lastausgleich
1-dimensionale Feldliste	Eindimensionale Feldliste
1-dimensionale FeldlisteMF	Feldliste1d mit Merkfeld
listenlängeNS	Länge einer *NS-1-dimensionalenFeldliste*
listenlängeOW	Länge einer *OW-1-dimensionalenFeldliste*
maxNS	Die südlichste vorhandene Koordinate
maxOW	Die östlichste vorhandene Koordinate
minNS	Die nördlichste vorhandene Koordinate
minOW	Die westlichste vorhandene Koordinate
	Folgende Klassen sind erklärt in Abschnitt 6.8.2:
mhf-Buffer	Speichert die bei einer NS-Migration übertragenen Informationen bis auf die Werte
mvf-Buffer	Speichert die bei einer OW-Migration übertragenen Informationen bis auf die Werte
mv-horizontal	Die transferierten Gittervektoren aller Gitterebenen. Es handelt sich dabei um horizontale Vektoren.
mv-vertikal	Die transferierten Gittervektoren aller Gitterebenen. Es handelt sich dabei um vertikale Vektoren.

Kommunikationsgrößen

Abschnitt 7.2

Bandbreite	Die Zeitkosten für eine Speicherzelle - ohne die Latenzzeit
Latenzzeit	Die Zeitkosten für eine leere Kommunikation

Die Balancing-Messung

Abschnitt 7.1.1

MAR(K,L,h)	Migration Außenrand
MARG(K,L,h)	Migration Außenrand Gesamtanzahl
MG(K,L,h)	Migrationsgebiet
MGG(K,L,h)	Migrationsgebietgröße
MV(K,L,h)	Anzahl der gepackten Vektoren
MVG(K,L,h)	Migration Vektoren Gesamtanzahl

Sonstige:

All-Reduce-Kommunikation	Eine all to all Kommunikation, die für die Operation Maximum erläutert wird: Jeder Kern besitzt hinterher

	den Maximalwert aller Kerne bzgl. einer Variablen.
echter Randpunkt	Ein Randpunkt, neben dem eine Stelle ist, auf der sich kein Gitterpunkt befindet. Abschnitt 4.2.5
idle - Zeiten	Warten auf eine Kommunikation, während deren keine Arbeit geleistet werden kann
K_n	Der vollständige Graph
f(n) = O(g(n))	f(n) wächst nicht wesentlich schneller als g(n)

Kapitel 1

Einleitung

Zuerst wird in den Abschnitten 1.1, 1.2 und 1.3 die Arbeit in den größeren Zusammenhang gestellt, und in Abschnitt 1.4 wird die eigene Arbeit kurz vorgestellt.

1.1 Paralleles Programmieren

1.1.1 Bedeutung des parallelen Programmierens

Parallelität spielte schon immer eine Rolle bei der Entwicklung von Computern und Computerprogrammen. Am Anfang ging es dabei vor allem um die Verwendung von größeren Zahlen, und damit der parallelen Behandlung der Bits der Zahlen in Registern und Operatoren. Der Autor hat bereits eine 4-Bit-CPU programmiert, die heutigen CPU's sind 64-Bit-CPU's. Eine weitere Möglichkeit, Parallelität zu nutzen, war die Bildung von Vektoren und entsprechenden Vektorregistern. Ein sehr interessanter Schritt hin zu mehr Parallelität war das Pipelining, d.h. das gemeinsame Bearbeiten von Befehlen, die aufeinander folgen, wobei das auch bei Verzweigungen (Branches) möglich wurde. Der momentan modernste Schritt sind die Multicore-CPU's, bei denen mehrere Kerne mit vollem Befehlssatz, sowie ihren Steuerungseinheiten auf einem Prozessor integriert sind. Ferner gibt es die GPU's mit viel mehr Kernen von einfacherer Bauart. Bei einer GTX 280 hat man 240 Kerne, vgl. [47].

Parallel zu diesen Errungenschaften gab und gibt es auch den Schritt, dass eine Vielzahl von Prozessoren über Kommunikationsverbindungen vernetzt werden, um eine höhere Leistung zu erzielen.

Paralleles Programmieren

Von allen diesen möglichen Parallelitäten bezieht sich die parallele Programmierung primär auf die Verteilung von Last auf verschiedene Prozessoren bzw. Kerne - wie das auch beim hier vorgestellten Programm der Fall ist. Das ist nicht die ausschließliche Möglichkeit zur Verwendung von Parallelität, vgl. dieses Beispiel zu der Nutzung der Parallelität größerer Zahlen: Bei der Schachprogrammierung gibt es sogenannte Bitbords: Dabei wird eine Stellung gespeichert, indem man die Figuren bitweise codiert, und für jedes dieser Bits eine 64-Bit-Speicherung anlegt. Die 64-Bit-Zahlen geben insgesamt an, auf welchen Feldern sich die Figuren befinden. Hierbei ist die Verwendung von 64-Bit-Variablen besonders günstig: Bei 32-Bit-Variablen muss man für ein Bit für

alle Felder schon 2 Variablen anlegen.
Bei 4- oder 8-Bit-Prozessoren macht diese Programmierung wenig Sinn. Also änderte sich die Programmierung auch mit der Größe der Variablen.

Bemerkung: Die Größe der Variablen spielt auch bei diesem Programm nicht nur für die Fließkommaoperationen eine Rolle, sondern auch für die später erklärte Struktur namens *selbstnachbarn* bzw. *EP-SNB*, wo in einem switch-case Befehl einer 8-Bit-Zahl die Situation für gleich 8 Himmelsrichtungen erkannt werden kann. Dazu braucht man zumindestens schon eine 8-Bit-CPU.

Bemerkung: Die rein sequentielle Programmierung hat eine Grenze in der Hardware. Die Größe der Atome und der Lichtgeschwindigkeit und die Anzahl der Schaltungen pro Chip ergeben eine obere Schranke der möglichen Taktfrequenz, vgl. z.B. [74] auf den Seiten 3 und 4, oder [19], weil es möglich sein muss, die in dem Bauteil maximal zurückgelegte Strecke in einem Takt zu durchlaufen. Allerdings: Zur Zeit liegt die minimale Dicke bei ca. 22 nm, vgl. [85] (bald 14nm), während ein Atom einen Durchmesser in der Größenordnung 0.1 nm hat (oder ein Angström). Diese Grenze ist also noch bei weitem nicht erreicht.
Die Verkleinerung der Schaltungen kann man neben der Steigerung der Taktfrequenz auch darauf verwenden, mehr Schaltungen aufzubringen - entsprechend dem Gesetz von Moore ([67]) verdoppelt sich alle 18 bis 24 Monate die Anzahl an Transistoren. Es spielt dabei wohl eine Rolle, dass bei höheren Frequenzen (> 5 GHz) mehr Hitze entsteht. Diese bildet auch einen limitierenden Faktor.

Bemerkung: Aktuell steigt die Taktfrequenz der CPU's wieder an, allerdings ist '> 5 GHz' immer noch ein sehr hoher Wert dafür.

1.1.2 Literatur zur Entwicklung paralleler Programme

Aufgrund der immer stärker anwachsenden *parallelen* Rechenleistung, und vor allem dem Aspekt der Verwendung mehrerer Kerne auf Prozessoren bzw. vieler Kerne auf GPU's, und der zusätzlichen Verwendung mehrerer Prozessoren bzw. GPU's, wurden viele Bücher zur Entwicklung paralleler Programme geschrieben, vgl. [16], [47], [75], [74], [15], [42], [59], [45], [72], [46]

1.1.3 Hardware - spezifisches paralleles Programmieren

Die vorhandene parallele Hardware bestimmt zu einem guten Teil mit, wie die Entwicklung von parallelen Programmen aussieht. Als erster hat Flynn in [30] eine Klassifikation von Hardware angegeben, die jetzt kurz dargestellt wird:

Das SISD Hardware - Modell

Single Instruction, Single Data: Hier liegt eine einzige Befehlsabfolge und die Daten liegen nur auf einer Recheneinheit vor. Man hat einen von Neumann Rechner, vgl. [74], der rein sequentiell programmiert wird.

Das SIMD Hardware - Modell

Single Instruction, Multiple Data: Die Befehle gelten für alle Instanzen gleich, die Daten sind aber verschieden. Das ist z.B. bei Vektorrechnern der Fall. Programmiert man z.B. eine Matrizenmultiplikation, so ist die Formel $\sum_{i=1}^{k} a_i\ b_i$ für jede Recheneinheit dieselbe, nur eben mit verschiedenen Daten.

Der MISD 'Fall'

Multiple Instruction, Single Data: Dieser Fall kommt nur zustande, weil alle 4 Fälle SISD, SIMD, MISD und MIMD aufgezählt werden. Er ist ohne praktische Relevanz. Es ist auch schwer, sich eine Anwendung vorzustellen, bei der dieselben Daten unterschiedlichen Rechenoperationen unterworfen werden.

Das MIMD Hardware - Modell

Multiple Instruction, Multiple Data: Hier arbeitet jede Instanz ihre Befehle ab, auf ihren speziellen Daten: Eine typische Hardware für aktuelle Groß- wie Kleinrechner, sei es CHEOPS (das ist der aktuelle Großrechner der Universität Köln) oder ein moderner PC.

Ähnlich dazu und auch aktuell ist der Fall SPMD - single programm multiple data - von Atallah, vgl. [6]. Alle Instanzen arbeiten dasselbe Programm ab, mit ihren speziellen Daten. Es ist hier charakteristisch, dass auf allen Rechnern dasselbe Programm abläuft, das aber in Abhängigkeit der Identität der Recheneinheit unterschiedlichen Programm-Code ausführen kann. Da das MIMD-Modell dieses auch erlaubt, liegt bei diesem Programm der Fall SPMD bzw. MIMD vor.

Weitere Klassifikationen neben der von Flynn

Nach [65] gibt es zur recht alten Klassifikation von Flynn modernere Varianten: Die von Kuck, vgl. [51], und die von Gurd, vgl. [33].

Eine wichtige Unterscheidung der Hardware für die Software-Entwicklung ist die Unterscheidung nach shared memory und distributet memory:

Shared Memory

In diesem Fall greifen die Recheneinheiten auf denselben Speicher zu, in letzterem Fall erfolgt die Koordination der Recheneinheiten über Kommunikation. Der Fall gemeinsamen Speichers ist leichter zu programmieren, wobei er Einschränkungen hat: Der gemeinsame Speicher ist meistens eingeschränkt in seiner Größe, und nur wenige Recheneinheiten haben gemeinsamen Zugriff. Außerdem gibt es Regeln für gemeinsames Schreiben in dieselbe Speicherzelle, damit immer die 'richtigen Werte' geschrieben werden.

Distributed Memory

Man spricht hierbei von verteiltem Speicher. Bei diesem Modell liegt der Speicher lokal bei den einzelnen Recheneinheiten vor. Aber es kommt zum Datenaustausch über

Kommunikationsvorgänge, bei denen Nachrichten zwischen Recheneinheiten verschickt werden.

Die Konsequenzen davon: Einerseits ist die Programmentwicklung deutlich aufwendiger. Und die Kommunikation läuft auf einer Shared Memory-Hardware schneller ab als über Kommunikationsnetzwerke. Andererseits gilt: Ist das Programm aber erst einmal so geschrieben, dann kann man es, wenn nicht das Programm selber das begrenzt, mit beliebig vielen Kernen laufen lassen.

Distributed Shared Memory

Es gibt dann noch den Fall, vgl. [34], bei dem eine Hardware mit verteiltem Speicher vorliegt, wobei es aber durch die entsprechende Software möglich wird, sie so zu programmieren, als wenn gemeinsamer Speicher vorliegt. Hier ist die Programmierung so einfach wie beim shared memory, obwohl keine Shared Memory-Hardware dafür vorliegt. Aber es muss die entsprechende Software dafür programmiert, und es müssen Effizienzverluste gegenüber einer direkten Distributed Memory-Programmierung in Kauf genommen werden.

Bei dem in dieser Arbeit entwickelten Programm liegt der Fall Distributed Memory vor. Allerdings läuft das Programm auch auf shared memory Systemen, nämlich bei Multi-Core-CPU's und auf gemischten Systemen, wo Multi-Core-CPU's über Kommunikationshardware miteinander verbunden sind.

CPU versus GPU

Eine Unterscheidung soll hier noch angegeben werden: Multicore-CPU's, ggf. verknüpft über ein Kommunikationsnetzwerk, sind anders zu programmieren als moderne GPU's (Graphics Processing Unit, Grafikprozessor). Eine moderne GPU kann nämlich auch programmiert werden, vgl. [47]. Sie hat viele 100 oder 1000 Kerne, die aber von recht einfacher Bauart sind und nur wenig gemeinsamen Speicher haben. Man hat dort strenge Restriktionen bei der Programmierung, vor allem ist der gemeinsame Speicher streng limitiert, normalerweise nur wenige Kilobyte groß. Und für eine effiziente Programmierung spielen die Limitierungen eine wichtige Rolle. Bei der Programmentwicklung muss durchdacht werden, wie ein Maximum an Leistung erreicht werden kann, unter Berücksichtigung dieser starken Beschränkungen.

Genauer: Bei der GPU geht es in der Programmierung darum, unter verschiedenen Einschränkungen auszuwählen, und dann die vorhandenen begrenzten Ressourcen möglichst clever auszunutzen, vgl. [47]. Es geht um limitierende Faktoren, die auf verschiedene Weise optimal ausgenutzt werden können. Das Programm zu dieser Dissertation ist viel komplexer. Damit es parallel läuft, musste sehr viel Know How entwickelt werden. Solch komplexe Operationen sind mit aktuellen GPU's nicht zu realisieren, allein schon wegen des Speicherplatzbedarfes. Beispielsweise kann man nicht 1025 x 1025 Matrizen mit diversen Strukturen auf ihnen und einer flexiblen Kerntopologie mit vielen GPU-Kernen, die nur wenig shared memory besitzen, das zudem nur einige festvorgegebene Kerne verbindet, bearbeiten.

Der Begriff der Granularität

Granularität ist eine Eigenschaft eines Programmes oder Algorithmus, gehört also auf den ersten Blick nicht in eine Hardwareklassifikation. Der 'zweite Blick' erfolgt unten bei dem kleinen Abschnitt 'Folgerungen $\cdots$'. Zuerst wird aber der Begriff geklärt. Man unterscheidet feinkörnige Granularität, wenn bei einem Programm schon nach wenigen Arbeitsschritten kommuniziert wird, und grobkörnige Granularität, wenn bei einem Programm zwischen zwei Kommunikationen viel gearbeitet wird, vgl. [34]. Als Beispiel: Man nehme das Rechnen auf einer Matrix. Wenn nach dem Rechnen auf nur ein paar Gitterpunkten kommuniziert wird, dann hat man einen feinkörnigen Algorithmus. Rechnet man hingegen für sehr viele Gitterpunkte zwischen den Kommunikationen, hat man einen grobkörnigeren Algorithmus.

Folgerung des Konzeptes der Granularität für die Hardware - Überlegungen
Distributed memory und feinkörnige Granularität passen nicht zusammen. Denn um eine Kommunikation durchzuführen über ein Kommunikationsnetz braucht es Zeit. Wenn dazwischen nur wenig Arbeit geleistet wird, lohnt sich das parallele Programm nicht. Wenn hingegen die Hardware entsprechend entwickelt wurde, beispielsweise indem man die Kommunikation in einem Chip integriert hat, ist feinkörnige Granularität möglich. Als Konsequenz: Bei einer Kommunikation über längere Kommunikationsleitungen ist grobkörnige Granularität notwendig. Und im Falle des hier vorliegenen Programmes ist das auch der Fall.

1.1.4 Zur Hardware zu dem Modell shared / distributed memory

Shared memory ist bei einer Multi-Core-CPU direkt über Leiterbahnen auf einem Prozessor realisiert. Es gibt da Probleme bei den Zugriffen. Insbesondere stellt das Schreiben in den Speicher eine schwierige Aufgabe dar, wenn verschiedene Kerne in dieselbe Speicherzelle schreiben wollen. Die Hardware-Hersteller entwickelten dafür ein Konzept mit Semaphoren - das sind Variablen, die angeben, ob ein Zugriff erlaubt wird oder nicht, wie bei einer rot-grün Ampel.

Beim distributed memory Modell geschieht die Verbindung zwischen den einzelnen Prozessoren über Kommunikationen. Die vorhandenen Prozessoren müssen damit über ein Kommunikationsnetzwerk miteinander verbunden werden. Kommunikationsnetzwerke sind unterschiedlich leistungsfähig. Ein Kriterium für die Leistungsfähigkeit eines solchen Netzes ist der Durchmesser des Netzes:

Definition 1.1: Durchmesser eines Kommunikationsnetzwerkes:
Der Durchmesser ist angegeben durch den Maximalwert unter den kürzesten Wegelängen in Bezug auf alle Punktpaare, bzw. ∞ bei einem unzusammenhängenden Netz.

Drei Kommunikationsnetze werden im Folgenden angegeben, vgl. diese Kommunikationsnetze in [34].

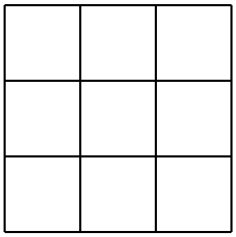


Abbildung 1.1: Das (2 - dimensionale) Gitter

Das Kommunikationsnetz Gitter

Hier sind die Kerne entsprechend dem Namen des Netzwerkes in einem Gitter angeordnet, vgl. Abbildung 1.1. Bis auf die Kerne an den Rändern hat jeder Kern einen nördlichen, östlichen, südlichen und westlichen Partner. Der Knotengrad ist daher maximal 4. Der Durchmesser eines n x m Netzes beträgt n+m-2, ist also relativ hoch. Bei dem Parallelisierungsprogramm dieser Arbeit erfolgt das Verschicken von Last beim Lastausgleich grob betrachtet über so ein Netz. Genauer betrachtet wird dieses Netz in 2 Netze aufgespalten, das mit den Ost-West-Kanten sowie den Nord-Süd Kanten, vgl. die Abbildungen 6.3 und 6.4.

Der Hypercube

Ein Hypercube der Dimension d hat $n = 2^d$ Knoten und entsteht so: Alle n Knoten seien nummeriert mit den Zahlen 0 bis n-1. Nun werden die Knoten miteinander durch Kanten verbunden, wenn sich die Binärdarstellungen ihrer Nummern in genau einem Bit unterscheiden. In Dimension 1 hat man eine Kante, in Dimension 2 ein 2x2 Gitter, und in Dimension 3 einen normalen Würfel.

Leistung des Netzes Der Durchmesser dieses Netzes ist d. Im Prinzip ist es ein gutes Netzwerk aus diesen Gründen:

1. Die Kommunikation geht schnell, da die längste Strecke im Netz, die überwunden werden muss, die Länge d hat.

2. Es entstehen keine Engpässe, da das Routen eines Nachrichtenpaketes durch das Netz auf viele unterschiedliche Weisen erfolgen kann. (Die Anzahl an unterschiedlichen Bits gibt die Länge der kürzesten Wege zwischen zwei Knoten an. Und die Anzahl an Permutationen in Bezug auf die Anzahl unterschiedlicher Bits gibt an, auf wie viele Weisen eine Kommunikation über kürzeste Wege erfolgen kann, da es egal ist, in welcher Reihenfolge der Bitexchange erfolgt.)

Der Nachteil des Netzes besteht darin, dass der Knotengrad mit der Dimension zunimmt, und bei Dimension d beträgt der Knotengrad d. Es wurde bis d = 13 praktisch realisiert, vgl. [34].

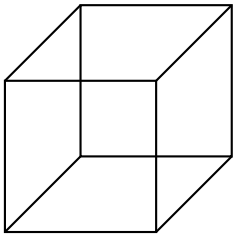


Abbildung 1.2: Der 3 - dimensionale Hypercube

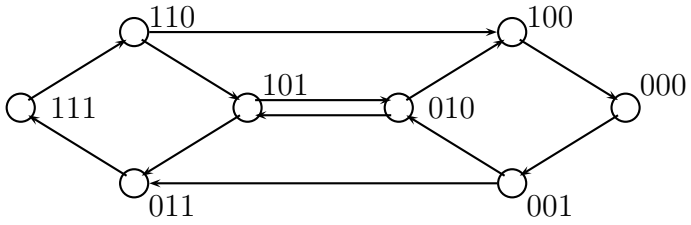


Abbildung 1.3: Das de Bruijn - Netzwerk, d=3, n=8, vgl. [34]

Das DeBuijn Netzwerk

Es basiert auf einem gerichteten Graphen. Man hat bei Dimension d n = 2^d viele Knoten. Diese seien binär nummeriert. Man verbindet nun die Knoten, die aus einer Linksverschiebung der Bits entstehen (Left Shuffle), und einer solchen Linksverschiebung, wo das herausgeschobene Bit negiert wird (Left Shuffle exchange). Die Richtung zeigt immer auf den Knoten mit der durch die Linksverschiebung mit ggf. negiertem herausgeschobenen Bit entstandenen Bitnummer.

Eigenschaften Dieses Netzwerk hat bei 2^d vielen Knoten einen Durchmesser von d, und dabei nur einen Knotengrad von maximal 4. Aber: Es wurde kommerziell noch nicht realisiert, vgl. [34].

1.1.5 Threats versus Prozesse

Bisher ging es bei der Diskussion zur Parallelisierung von Programmen vor allem um die Hardware. Die sich jetzt anschließende Diskussion von Threats versus Prozessen orientiert sich mehr an der Programmierung, wobei hier Hardware-Aspekte auch eine Rolle spielen, vgl. CPU-Threats im Vergleich mit GPU-Threats, siehe unten unter Threats.

Eine Diskussion von Threats versus Prozesse findet sich in [74]. Prozesse sind ausgeführte Programme, die bei einer Parallelisierung erforderlich sind. Threats sind hingegen Teile eines Prozesses, dessen Arbeit aufgeteilt wurde.

Der Unterschied für den Programmierer wird durch folgende Darstellung deutlich:

Prozesse

Ein Prozess ist der Teil eines ausgeführten Programms, der auf einem Kern abgearbeitet wird. Er hat die vollständigen Ressourcen zu seiner Bedienung alleine zur Verfügung,

also Arbeitsspeicher und auch Register. Werden Prozesse gewechselt, muss die komplette Information gesichert werden.

Threats

Sie treten auf als Bestandteile von Prozessen. Die Threats teilen sich untereinander Ressourcen. Sie müssen deswegen vor einem Wechsel keine Informationen speichern oder laden.

Für einen Kern gilt: Während bei Leistungsläufen dort im Wesentlichen nur ein Prozess läuft, können mehrere Threats nebeneinander laufen.

Bemerkung: Der Hauptunterschied zwischen CPU-Threats und GPU-Threats besteht darin: Bei der GPU hat man sehr viele eher kurzlebige Threats, während bei der CPU wenige aufwendigere Threats vorliegen, vgl. [47].

Bei dem hier vorgestellten Programm werden immer Prozesse verwendet; insbesondere wird bei Messungen nur ein Prozess pro Kern gestartet, d.h. auf 16 Kernen laufen 16 Prozesse.

1.1.6 Speedup und Effizienz

Die Leistungsfähigkeit einer Parallelisierung eines Algorithmus soll angegeben werden. Man misst dabei die Zeiten der sequentiellen und parallelen Ausführung, T_s und T_p. Dabei steht s für sequentiell, und p für parallel. Ggf. ist die optimale sequentielle Laufzeit T_s^* zu bestimmen. Sei K die Anzahl der verwendeten Kerne. Es gilt:

Speedup

$S = \frac{T_s}{T_p}$, vgl. [34], oder

$S^* = \frac{T_s^*}{T_p}$, vgl. [74].

Der Speedup gibt an, um welchen Faktor die parallele Bearbeitung bei K Kernen schneller wird.

Bemerkung: Streng genommen gibt es natürlich keine optimalen Algorithmen in komplexeren Fällen. Im Folgenden wird die Definition von [34] genommen. Die vorgenommenen Messungen erfolgten sequentiell und parallel mit demselben Algorithmus, wobei beim sequentiellen Lauf parallele Teile wie der Balancer abgeschaltet wurden.

Effizienz

Man hat als Effizienz:

$\text{Eff} = \frac{T_s}{K * T_p} = \frac{S}{K}$

Man geht davon aus, dass im optimalen Fall - der Speedup wäre K - ein Wert von 100 % erreicht wird. Die Effizienz gibt an, wie weit der parallele Algorithmus von dieser Leistung entfernt ist.

Amdahl's Gesetz

Im folgenden Gesetz - vgl. [4] - wird unter einer einfachen Annahme eine Abschätzung für den Speedup gemacht. Ein unvollständiger Lastausgleich oder Kommunikationszeiten werden dabei nicht berücksichtigt.

Man nimmt an, dass ein Algorithmus einen Anteil s von sequentiellen Anteilen hat, und einen Anteil p = 1 - s von rein parallel auszuführenden Arbeiten. Dann folgt:

$\text{S} = \frac{T_s}{T_p} = \frac{s+p}{s+\frac{p}{K}} = \frac{1}{s+\frac{1-s}{K}} \leq \frac{1}{s}.$

Bei einem sequentiellen Anteil von z.B. 10 % am gesamten Programm (90 % wäre parallelisierbar) ist bei beliebig vielen ausführbaren Einheiten ein maximaler Speedup von 10 zu erreichen.

In Abschnitt 2.1.1 wird dieser Fall auf den Fall von unausgeglichener Last erweitert.

1.1.7 Speicherzugriffszeiten versus Rechenzeiten

Wenn man parallele Programme bzgl. ihrer Leistungsfähigkeit einschätzen will, sind Messungen unumgänglich, vgl. den Abschnitt 8.2. Diese sollen dann auch interpretiert werden. Dafür muss man wissen, was die einzelnen Befehle an Zeit kosten. In der innersten Schleife dieses Programmes wird gerechnet, d.h. relevant sind Speicherzugriffe und Fließkommaoperationen. Relevant sind ferner die Kosten des Lastausgleiches, insbesondere die Kosten der Migration, sowie die der parallelen adaptiven Verfeinerung.

Speicherzugriffszeiten

Der Speicher ist nicht in gleicher Weise schneller geworden, wie es die CPU's geworden sind. In [74] Seite 3 steht, dass von 1990 bis 2006 die Anzahl von Zyklen zur Abfrage des Hauptspeichers von 6 bis 8 auf 220 angestiegen sind. Andere Quellen sagen zur aktuellen Zahl Vergleichbares.

Bemerkung: In [47] ist die Rede von einer Latenz von 200 Zyklen. Man kann ggf. aus dem Speicher Werte in strenger Abfolge schneller lesen. Natürlich muss dann auch so programmiert werden bzw. werden können, dass es Sinn macht, größere Datenmengen

hintereinander aus dem Speicher auszulesen.

Um diese Zeitdauern zu erfassen, hat man folgende 2 Definitionen:

Definition 1.2: Latenz und Bandbreite:
Bei einer Kommunikation ist die Latenz die Zeit, die eine maximal kurze Kommunikation benötigt. Die Bandbreite gibt an, um wie viel sich diese Zeit verlängert, wenn eine längere Kommunikation erfolgt. Also beispielsweise verlängert sich die Kommunikationsdauer um so und so viele Millisekunden, wenn ein kByte mehr transferiert wird. Bei der Latenzzeit geht es um die Dauer des Ereignisses Kommunikation und Speicherzugriff selber, bei der Bandbreite um die Dauer des Lesens von vielen Daten hintereinander.

Rechenzeiten

Fließkommaoperationen waren früher sehr teuer. Heutzutage braucht man für Additionen und Multiplikationen ca. 11 CPU-Zyklen, für Divisionen ca. 30 CPU-Zyklen, vgl. [3].

Diskussion - auch zum Cache

Man sieht, dass Speicherzugriffe relativ zum Rechnen teuer sind. Hier spielt der Cache und seine Wirksamkeit eine wichtige Rolle, weil dadurch die teuren Zugriffe auf den Speicher, und damit die Wartezeiten der CPU, wesentlich verringert werden können. Der Cache-Speicher ist aber limitiert, was beim Rechnen von großen Strukturen wie großen Gittern eine Rolle spielt. Bei der Durchführung und Interpretation von Messungen muss man diese 2 Aspekte berücksichtigen - teurer Speicherzugriff und limitierter Cache-Speicher.

1.2 Adaptives Mehrgitter als Anwendung

1.2.1 Verschiedene Anwendungen von paralleler Programmierung

Die Nutzung paralleler Programmierung erfolgt einmal direkt über das Betriebssystem, indem verschiedene Prozesse auf die Kerne verteilt werden, vgl. [19]. Ansonsten werden parallele Programme in verschiedenen Bereichen entwickelt, wie z.B.:

1. Numerische Probleme wie adaptives Mehrgitter, also z.B. beim hier vorliegenden Programm.

2. NP-vollständige Probleme, insbesondere kombinatorische Suchprobleme wie SAT oder Vertex Cover, vgl. [20] und [21]. Der Autor hat mit Hilfe von PLB, vgl. [19], seinen Vertex Cover Solver parallelisiert. Und hat das dabei gelernte Lastausgleichs-Prinzip, Nachverschicken von Last zwecks Lastausgleich, bei der Parallelisierung von adaptiven Mehrgitterverfahren verwendet.

3. Verkehrssimulation, vgl. [76].

4. Bildverarbeitung, vgl. [54].

5. Schachprogramme, wie z.B. Stockfish [79] oder Toga [81].

   Der Entwickler von [81] hat beispielsweise sein Schachprogramm parallelisiert, nicht indem er den Suchbaum als solchen parallel verteilt hat, sondern indem er die Hashtabelle (für die Bewertung der schon berechneten Stellungen) in den gemeinsamen Speicher platziert hat.

1.2.2 Strömungssimulationen

Bei der Anwendung in diesem Fall geht es in der allgemeinsten Form um Strömungssimulationen. Dabei gibt es eine Reihe von Möglichkeiten zur Nutzung von solchen Simulationen:

1. Luft - oder Wasserströmungen, z.B. in der Meteorologie für die Wettervorhersage, die Windkanalsimulation (z.B. für die CW-Wert-Bestimmung), oder das Blut im Körper.

2. Die Simulation von Elektronen im Halbleiter oder anderen Stromleitern.

Zuständig für die Handhabung von Strömungen ist im ersten Schritt die Wissenschaft Physik, weil es um Teilchenbewegungen geht. Hier gibt es zwei Möglichkeiten:

1. Man könnte die Teilchenbewegungen direkt simulieren, wobei vor allem virtuelle Stöße vorzunehmen wären. Das geht nur für sehr kleine Mengen, weil man schnell enorme Mengen an Teilchen hätte.

2. Oder man verwendet Differentialgleichungen: Physiker haben Differentialgleichungen entwickelt, die im Wesentlichen am selben Ort zu denselben Werten wie Temperatur oder Windgeschwindigkeit führen, als wenn man eine Simulation für die Teilchen selber vornehmen, bzw. physikalische Messungen durchführen würde.

1.2.3 Lösungen von Differentialgleichungen

Differentialgleichungen können auf zwei verschiedene Weisen gelöst werden:

1. Erst einmal können sie analytisch gelöst werden. Zum Beispiel gibt es im Physikstudium Aufgabenstellungen dazu, die der Student exakt löst mithilfe der Theorie der Vorlesung.

2. Eine numerische Lösung ist möglich, wobei Variablen Gitterpunkte zugeordnet werden. Man hat dazu dann Anfangswerte und diskretisierte Differentialgleichungen, abgeleitet von den 'normalen' Differentialgleichungen, mit denen man das (Strömungs-) Problem durch Gitterberechnungen lösen kann.

1.2.4 Numerische Methoden zur Lösung von Differentialgleichungen

Es gibt eine ganze Reihe numerischer Methoden zur Lösung von (partiellen) Differentialgleichungen:

1. Die einfachen Lösungsmethoden wie Jacoby oder Gauss-Seidel, vgl. den Abschnitt 2.2.3.

2. Die für ihre Art sehr effizienten Mehrgitterverfahren, vgl. den Abschnitt 1.2.5.

3. Krylov Unterraum Methoden mit dem Spezialfall konjugierte Gradienten Methode. Zu letzterem vgl. z.B. [2].

4. Für spezielle Differentialgleichungen entwickelte Methoden, wie z.B. in [34] für die Navier-Stokes Gleichungen.

1.2.5 Mehrgitterverfahren

Mehrgitterverfahren sind sehr effiziente Lösungsmethoden im Verhältnis zu einfachen Lösern wie Jacoby oder Gauss-Seidel. Man verwendet zu relativ feinen Gittern mehrere Abstufungen von Vergröberungen, häufig unter Auslassung jedes 2. ten Gitterpunktes je Dimension. Die Werte auf den vergröberten Gittern stellen dabei Korrekturen zu den Werten auf den jeweils feineren Gittern dar.

Die Idee der Methode

Es gibt gewisse Eigenfunktionen zu Differentialgleichungen genannt Frequenzen, vgl. [43] auf Seite 40. Auf den feinen Gittern werden die Unterschiede der Werte von sehr nahe beieinander liegenden Punkten gut wiedergegeben. Das bedeutet, dass sie Schwingungen mit kurzen Wellenlängen oder hohen Frequenzen gut darstellen. Auf groben Gittern werden die Unterschiede in Bezug auf sehr weit voneinander entfernte Punkte gut wiedergegeben. Das bedeutet, dass sie Schwingungen mit langen Wellenlängen oder sehr niedrigen Frequenzen gut darstellen. Durch die Berechnungen auf und zwischen den Gittern werden dann die guten Leistungen für beide Fälle kombiniert.

1.2.6 Adaptive Mehrgitterverfahren

Es tritt der Fall auf, dass über das Gitter verteilt Bereiche entstehen, wo eine höhere Auflösung erforderlich ist, wie bei einem Wirbelsturm bei der Wettersimulation, und Bereiche, wo das nicht erforderlich ist, z.B. bei Windstille bei der Wettersimulation. In diesem Fall macht es Sinn, Verfeinerungen nur in gewissen Gebieten vorzunehmen, um Rechenzeit zu sparen. Man spricht dann von adptiven Mehrgitterverfahren.

Unterscheidung von adaptiven Mehrgitterverfahren

Man unterscheidet predefinierte adaptive und selbstadaptive Mehrgitterverfahren, vgl. [82]. Es macht auch Sinn, von statisch adaptiven und dynamisch adaptiven Mehrgitterverfahren zu sprechen. Im ersten Fall wird einmal vor der eigentlichen Rechnung verfeinert, und diese dann für den Rest des Rechnens beibehalten. Wenn auf einer Struktur die Geometrie konstant bleibt, wie z.B. bei einem Flugzeug im Windkanal, dann kann man so vorgehen. Ändert sich aber der Verfeinerungsbedarf im Laufe der Zeit, muss die Verfeinerung immer wieder neu angepasst werden. Man spricht dann von einem selbstadaptiven Mehrgitterverfahren. Zu Testzwecken, insbesondere für Leistungsmessungen, bei denen vorgegeben wird, wie sich die Verfeinerungsgebiete bewegen, kann man dabei ein solches Mehrgitterverfahren simulieren mit vorgegebenen Änderungen

der Verfeinerungen, vgl. [37]. Dieses sind dann gesteuerte Strukturen, wie z.B. in eine Richtung bewegte Rechtecke.

1.3 Paralleles adaptives Mehrgitterverfahren

Es wurde in die parallele Programmierung eingeleitet, sowie in die adaptiven Mehrgitterverfahren. Nun wird etwas zur Kombination beider gesagt. Allgemein zur Parallelisierung von numerischen Methoden vergleiche man z.B. [2] oder [34].

Möglichkeiten

Hier werden drei Ansätze wiedergegeben, um sich im Laufe der Zeit ändernde Verfeinerungen parallel zu behandeln:

1. In einem Mehrgitterkurs, vgl. [44], wurde diese Methode angegeben: Das gesamte Gebiet wird in viele kleine Teilgebiete unterteilt, deren Zahl viel größer ist als die der für die Parallelisierung verwendeten Kerne. Diese Teilgebiete werden reihum den Kernen zugeordnet. Wird nun ein größeres Gebiet verfeinert, dann wird automatisch dieser Bereich auf die kleinen Teilgebiete verteilt, und damit auf verschiedene Kerne. Man erhält so ein gewisses Maß an Lastausgleich. Nachteil: Man hat keinen vollständigen Lastausgleich, und man hat, wenn diese Teilgebiete eher klein sind, viel Kommunikation zwischen diesen Gebieten.

2. Die Verwendung von Patches, vgl. [60] oder [36]: Der Verfeinerungsbereich wird überlagert durch eine Vielzahl an Rechtecken genannt Patches, wobei die Punkte auf einem Patch jeweils komplett auf einem Kern liegen. Es muss dann immer wieder umverteilt werden, und neue Patches gebildet werden, und es muss eine Kommunikation auf und zwischen den Gittern erreicht werden. Für den Lastausgleich verwendet man z.B. die Bisektionsmethode, d.h. die Zerlegung eines Gebietes in eine vorher angegebene Zahl von Teilgebieten, wie z.B. der Anzahl der verwendeten Kerne.

3. Die hier verwendete Methode: Die Gitter jeder Ebene lassen für jeden Kern fast beliebige Gitterpunktmengen zu. Es muss nur die Bedingung eingehalten werden, dass übereinander liegende Gitterpunkte auf demselben Kern liegen. Dadurch besteht die Möglichkeit, das Gitter flexibel aufzuteilen, was beim Lastausgleich dann genutzt wird.
   Es werden dabei, entsprechend einem Verfeinerungskriterium, die Verfeinerungsgitter parallel bestimmt, vgl. Kapitel 5, und durch Umverteilung von Gitterpunktemengen der Lastausgleich erreicht, wobei sich der Lastausgleich aufteilt in eine Lastumverteilungsberechnung sowie in die Lastmigration, vgl. Kapitel 6.

1.4 Über die eigene Arbeit

1.4.1 Das Ziel der Arbeit

Der Arbeit gingen drei Arbeiten voraus:

1. In der einen Arbeit wurde ein adaptives Mehrgitterverfahren parallelisiert, vgl. [36].

2. Dann gab es das Konzept PLB, bei dem durch das Verschicken von Lasteinheiten ein Lastausgleich erreicht wird, vgl. [19].

3. Und in der dritten Arbeit wurden auf einem Gitter einzelne Gitterpunkte verschickt, vgl. [53].

Entsprechend dieser Arbeiten lautet das Ziel wie folgt: Ein paralleles adaptives Mehrgitterverfahren zu entwickeln, welches das Konzept PLB verwendet, Lasteinheiten zu verschicken, die dann aus einzelnen Gitterpunkten bestehen.

1.4.2 Inhaltliches über die Arbeit

Wichtig ist zu erwähnen, dass dabei über eine einfache Anwendung von PLB weit hinausgegangen werden muss, weil beim Verschicken von Gitterpunkten die Nachbarschaft um die Punkte herum mit beachtet werden muss. Das ist anders als beim Verschicken von Last beim SAT-Problem in [19]. Die Lasteinheiten in [19] sind unabhängig voneinander.
Es stellte sich dann heraus, dass dazu eine besondere Aufgabe hinzukommt, nämlich die adaptive Verfeinerung parallel zu behandeln. Diese in Kapitel 5 behandelte Aufgabe stellt einen zentralen Bestandteil der Arbeit dar.
In Kapitel 6 wird dann der parallele Lastausgleich behandelt, bei dem aber nicht nur die Umgebungen um die verschickten Gitterpunkte behandelt werden, sondern auch die für Kapitel 5 benötigten Strukturen so geändert werden müssen, als wenn die Gitterpunkte schon immer dort gewesen wären. Diese beiden Kapitel bilden den Kern der Arbeit.
Die einfacheren Algorithmen des Programms werden in Kapitel 4 behandelt.
In Kapitel 3 werden die Kapitel 4 bis 6 vorbereitet, wobei schon in wichtige Strukturen eingeführt wird, und die Aufgaben der Kapitel 4 bis 6 schon formuliert, aber noch nicht gelöst werden.
In Kapitel 2 werden Überlegungen aus dieser Einleitung aufgegriffen und erweitert, und es werden für die Arbeit wichtige Konzepte wie das Message Passing Interface bzw. die Mehrgitterschemata vorgestellt.
In Kapitel 7 wird dann der Zeitaufwand des Balancers theoretisch behandelt.
In Kapitel 9 geschieht ein Vergleich mit dem, was andere Autoren geleistet haben. Insbesondere geschieht ein direkter Vergleich mit parallelen dynamisch adaptiven Mehrgitterverfahren.

Kapitel 2

Vorüberlegungen zum Programm

Das Programm wird in den folgenden Kapiteln präsentiert. Einige einfache Überlegungen erleichtern das Verständnis. Die Vorüberlegungen gliedern sich auf in die Abschnitte 'Vorüberlegungen zum parallelen Programmieren', 'Vorüberlegungen zum Mehrgitter' sowie 'Vorüberlegungen zum parallelen Mehrgitter'. Der Schwierigkeitsgrad zum Verstehen dieser Vorüberlegungen ist dabei gering.

2.1 Vorüberlegungen zum parallelen Programmieren

2.1.1 Fortführung von 2 Punkten aus Kapitel 1

Amdahls Gesetz erweitern

Bei Amdahls Gesetz geht es nur um den sequentiellen und parallelen Anteil des Programmes. Wie ausgeglichen die Last wird, oder welchen Beitrag die Kommunikation spielt, wird nicht berücksichtigt. Dem Aspekt unausgeglichener Last wird hier Rechnung getragen.

Angenommen, dass die K Lasten L_i ungleich verteilt seien. Ferner seien $\overline{L}$ und maxL definiert durch $\overline{L} = \frac{1}{K} \sum_{i=1..K} L_i$ und $\text{maxL} = \max_{i=1..K} L_i$. Es sei q der Prozentsatz, den der Kern mit der höchsten Last gegenüber einem idealen Durchschnittswert der Lasten mehr an Laufzeit benötigt. Diese Laufzeit ist dann maxL = (1+q) $\overline{L}$. Mit diesen Bezeichnungen gilt:

Satz 2.1: Die Effizienz beträgt, wenn man den Laufzeitverlust nur aufgrund von Lastimbalancen bestimmt:

$$\text{Eff} = \frac{T_s}{KT_p} = \frac{\sum_{i=1..K} L_i}{K maxL} = \frac{\frac{1}{K}\sum_{i=1..K} L_i}{maxL} = \frac{\overline{L}}{maxL} = \frac{1}{1+q}$$

Im nächsten Schritt kombiniert man die ursprüngliche Formel von Amdahl mit obiger Formel. Dann hat man 2 Gründe für Effizienzverluste: Einmal, dass ein sequentieller Anteil vorliegt, und dann, dass der parallele Anteil der Last nicht wie bei Amdahls Annahme ideal $\frac{p}{K}$ ist, sondern mehr Laufzeit benötigt, nämlich $\frac{p}{K} \frac{maxL}{\frac{\sum_{i=1..K} L_i}{K}} = \text{p} \frac{maxL}{\sum_{i=1..K} L_i}$.

Denn aufgrund von Lastimbalancen ändert sich nicht die Laufzeit des sequentiellen Teils des Algorithmus, aber die parallele Laufzeit nimmt um den Faktor (1+q) zu. Damit folgt Satz 2.2:

Satz 2.2: Die Effizienz beträgt bei Leistungsverlusten aufgrund eines sequentiellen Anteils sowie wegen unvollständigem Lastausgleich

$$\text{Speedup} = \frac{T_s}{T_p} = \frac{s+p}{s+p\frac{maxL}{\sum_{i=1..K} L_i}} = \frac{1}{s+(1-s)\frac{(1+q)}{K}} \Rightarrow$$

$$\text{Eff} = \frac{Speedup}{K} = \frac{1}{sK+(1-s)(1+q)}$$

Es ergeben sich für q = 0 das Amdahl'sche Gesetz, und für s = 0 obiger Satz 2.1 als Spezialfälle:

1. q = 0: $\text{Speedup} = \frac{1}{s+(1-s)\frac{(1+q)}{K}} = \frac{1}{s+(1-s)\frac{1}{K}} = \frac{1}{s+\frac{(1-s)}{K}}$
2. s = 0: $\text{Effizienz} = \frac{1}{K}\ (\frac{1}{s+(1-s)\frac{(1+q)}{K}}) = \frac{1}{K}\ (\frac{1}{0+(1-0)\frac{(1+q)}{K}}) = \frac{1}{1+q}$

Das sind allgemeine Formeln. Man kann auch noch die Kommunikation in die Laufzeitüberlegungen einbeziehen, siehe später in Kapitel 7. Es ergeben sich dann aber spezielle Formeln für die Gitterparallelisierung dieses Programms.

Zu shared / distributed memory, vgl. Abschnitt 1.1.3

An dieser Stelle sind zwei Bemerkungen vorzunehmen:

1. Bei einer shared memory Hardware lässt sich die distributed memory Hardware leicht emulieren, indem die zu verschickenden Nachrichten in den gemeinsamen Speicher kopiert werden.

2. In Bezug auf dieses Programm gilt: Bei einer shared memory Version des Programms würde für eine Hardware mit genügend gemeinsamem Speicher eine enorme Vereinfachung in der Programmierung im Vergleich mit der distributed memory Version möglich werden. Man müsste dazu nur die Datenstruktur *Farbenfeld*, die beim Programm vielfach mühsam zu bestimmen und zu aktualisieren ist, in den gemeinsamen Speicher aufnehmen. Die Vereinfachungen der Algorithmen wären beträchtlich.
   Da die Datenstruktur *Farbenfeld* erwähnt wurde, ist dazu folgende Bemerkung zu machen: Diese Datenstruktur gibt an, auf welchem Kern sich ein Punkt befindet. Und sie nimmt bei nicht existierenden Punkten negative Werte, z.B. die -1, an.

2.1.2 Vorbemerkungen zu Programmteilen

Lastausgleich

Das hier vorgestellte Konzept zum Lastausgleich basiert darauf, dass sich die Gitterpunkte auf die Kerne verteilen und im Gesamten das jeweilige Gitter ergeben. Die kleinste zu verschickende Einheit ist die eines Gitterpunktes, im Unterschied zu einer Überdeckung des Verfeinerungsgebietes mit Patches, wo ganze Rechtecke sich auf

Abbildung 2.1: Aufbaustruktur der parallelen adaptiven Verfeinerung; EP^+/EP^- sind die Datenstrukturen zu den Mengen EP^+/EP^-

einem Kern befinden. Daraus ergibt sich mit den folgenden Schritten ein Konzept für den Lastausgleich, was sich an das Konzept bei der Nutzung von PLB - genannt Precomputation-based Load Balancing - bei einem kombinatorischen Problem anlehnt, vgl. [19]:

1. Die Lastmengen werden bestimmt. Das kann man in den Datenstrukturen *ev* bzw. *ev2* direkt abfragen. Zu diesen Strukturen vgl. man den Abschnitt 6.7. Ein direktes Ablesen ist nicht möglich in dem Fall, wo Randgitterpunkte nicht als Last gesehen werden - das ist der Fall, bei der die Lastwertekorrektur zum Einsatz kommt, vgl. Abschnitt 6.9.

2. Man berechnet daraus, wie viele Gitterpunkte zwischen bezüglich der Kerntopologie benachbarten Kernen zu verschicken sind.

3. Man führt den Transport dieser Gitterpunkte durch.

Parallele adaptive Verfeinerung

Das ist eine schwer zu behandelnde Methode. Die Idee hinter der Lösung der Aufgabe, diese Verfeinerung parallel vorzunehmen, besteht darin, die vorhandenen Informationen zu verwenden, um darauf aufbauende Informationen zu bekommen. Die Entwicklung der aufbauenden Datenstrukturen erforderte viel Entwicklungsarbeit. Grundlage sind einerseits die Verfeinerungsinformationen, aus denen die Mengen EP^+ und EP^- gebildet werden. Diese sind zu einer Datenstruktur *EP-SNB-Update* zu aggregieren. Daraus ergibt sich, ob ein Punkt hinzukommt oder entfernt wird, was in den Datenstrukturen P_k^+ und P_k^- verzeichnet ist.

Diese Schritte müssen hier noch nicht verstanden werden, weil die entsprechenden Datenstrukturen erst in Kapitel 5 in Abschnitt 5.3.1 eingeführt werden. Es geht hier nur darum, die Aufbaustruktur der parallelen adaptiven Verfeinerung schon einmal zu präsentieren, vgl. Abbildung 2.1.

2.1.3 MPI - eine Schnittstelle für die parallele Programmierung

MPI - das Message Passing Interface - ist ein Zusatz zu C/C++ bzw. Fortran zur parallelen Programmierung. Da das Programm darauf basiert, sind zu MPI einige Bemerkungen zu machen - inklusive zu Überlegungen, was MPI leistet.
Für die Dokumentation von MPI vgl. [68] oder [69].

Einordnung unter vergleichbaren Schnittstellen

So wie MPI im Falle von verteiltem Speicher eingesetzt wird, gibt es z.B. OpenMP ([86]) für die Programmierung von shared memory Systemen. Vergleichbar mit MPI

sind Konzepte wie

1. PVM, vgl. [73]
2. LEDA, vgl. die Literatur auf der Internetseite [58]

Bestandteile von MPI

Um MPI unter C/C++ unter Windows nutzen zu können, benötigt man 4 Dateien:

1. Eine C-header Datei mpi.h, wo die Funktionen, die aufgerufen werden, in C-Code definiert sind.
2. Eine Bibliotheksdatei mit der Bezeichnung libmpi.a, oder einer ähnlichen Bezeichnung wie libmpich.a, bei der die aufzurufenden Funktionen in Maschinencode geschrieben sind.
3. MPIrun.exe - das ist eine auszuführende Datei, mithilfe derer man compilierte Programme für mehrere Kerne starten kann.
4. mpi.ps ist eine Postscript-Datei, die zum Lernen der Befehle bzw. der Möglichkeiten von MPI geeignet ist, vgl. [68]. Es gibt davon mehrere Dateien, je nach Umfang der jeweiligen Version von MPI.

Bemerkung: Unter Ubuntu-Linux wird statt mpi.h und libmpi.a ein eigener MPI - Compiler mpic++ verwendet.

Das Prinzip hinter MPI

Das Prinzip hinter MPI besteht darin, diverse MPI-Funktionen bereitzustellen samt verwendeten MPI-Konstanten, um Botschaften verschicken zu können. Das deckt sich mit den Anforderungen an eine Distributed Memory-Hardware.

Diverse Befehle von MPI

1. Gestartet wird MPI mit dem Befehl MPI_Init, bei dem Programmparameter übergeben werden. Mit MPI_Finalize wird MPI beendet.
2. Die Identifikation des eigenen Kerns und die der anderen Kerne, an welche Botschaften geschickt werden können, geschieht über MPI_Comm_size, wo die Anzahl der beteiligten Prozesse angegeben wird, und MPI_Comm_rank, mit denen eine ID angegeben wird, damit das Programm erkennt, welcher Prozess der eigene ist. Diese ID wird auch Rang genannt.
3. Man sendet nicht blockierend mit MPI_Isend und empfängt nicht blockierend mit MPI_Irecv. Nicht blockierend bedeutet hier, dass das Programm an diesen Stellen weiterläuft, auch wenn die Kommunikation nicht abgeschlossen wurde. Der Abschluss der Kommunikation wird dann mit dem Befehl MPI_Test geprüft, bzw. mit MPI_Wait wird auf den Abschluss der Kommunikation gewartet.

   Bei den Sende-/Empfangsbefehlen wird der zu lesende bzw. zu beschreibende

Buffer angegeben, die Identität des Partner-Prozesses/-Kerns, entsprechend seinem **Rang**, und ein **Tag**, um mehrere Botschaften, die zwischen denselben Kernen in dieselbe Richtung gehen, zu unterscheiden, sowie ein **Request** für die spätere Testabfrage, ob die Kommunikation erfolgt ist.

4. Es gibt weitere Befehle, z.B. um die elementaren Strukturen zu definieren, und diese dann zu erweitern durch Befehle von MPI zur Bildung von Strukturen. Die zusammengebauten Strukturen können dann auch verwendet werden wie eigene MPI-Strukturen. Es gibt sogenannte Kommunikatoren, die eine Teilmenge von Kernen festlegen mit eigenen ID's nur für diese Teilmenge, die damit einschränken, mit wem man kommunizieren kann. Und dann sind Kommunikationen zwischen mehr als 2 Partnern möglich, wie z.B. MPI_Bcast, um von einem Kern Werte an alle Mitglieder eines Kommunikators zu schicken, oder MPI_Allreduce, um Nachrichten von allen Prozessen zu allen Prozessen eines Kommunikators zu schicken.

2.2 Vorüberlegungen zum Mehrgitter

2.2.1 Vorbemerkungen

Ziel ist es, das adaptive Schema einzuführen. Dabei ist von einem Schema im Sinne der Mehrgittertheorie zu sprechen, wenn man eine Abfolge von Mehrgitter-Befehlen angibt, nach deren Abarbeitung ein Fortschritt in numerischem Sinne erfolgt, der sich auf die Werte bestimmter Variablen, wie die für das Residuum, bezieht. Nach den anfänglichen Definitionen und dem Eingitterproblem werden drei Schemata mit den jeweils hinzukommenden Funktionen dargestellt: Das (einfache) Mehrgitterschema, dann das komplexere FAS und schließlich das Schema für die adaptiven Verfeinerungen, wobei sich die letzteren beiden Schemata auf vollständigen Gittern nicht unterscheiden.
Dazu werden dann solche Multigrid-Funktionen dargestellt, die die Poisson-Gleichung lösen. Das Programm ist allerdings so aufgebaut, dass es nicht nur für eine spezielle Differentialgleichung verwendbar ist, sondern es ist allgemein verwendbar für solche Fälle, wo die Differentialgleichung mit 5- oder 9-Punkte Sternoperatoren arbeitet, wobei es vom Prinzip her auch auf größere Sternoperatoren erweiterbar ist. Das wird an dieser Stelle aber nicht weiter vertieft.

Bei der parallelen adaptiven Verfeinerung, vor allem in Kapitel 5, geht es weniger um das adaptive Schema selber und um seine Funktionen, sondern vielmehr wird die Gitterumbildung an sich thematisiert. Allerdings kann man streng genommen beide Punkte nicht trennen, und zwar deshalb, weil im adaptiven Schema auch von alten und neuen Gittern die Rede ist, vgl. G_h und G_h' im Abschnitt 2.2.6.

2.2.2 Definitionen

Partielle Differentialgleichungen als Ausgangspunkt

Die Methode beginnt bei den partiellen Differentialgleichungen. Diese kommen aus dem Gebiet der Physik. Wir wählen dazu die Differentialgleichung L u = f auf Ω und u = g auf $\delta\Omega$. Dabei ist Ω das gewählte Gebiet und $\delta\Omega$ der Rand des Gebietes. Ω^0 bezeichnet

das Innere des Gebietes. L ist der Differentialoperator, u die Lösungsvariable, und f bzw. g sind die Werte an der jeweiligen Stelle.

Die eigentliche Diskretisierung

Man überdeckt das Gebiet mit einem Gitter - z.B. mit gleichmäßigen Abständen zwischen den Gitterpunkten - sowohl in vertikaler als auch horizontaler Richtung. Um die Differentialgleichung in Bezug auf dieses Gitter formulieren zu können, braucht man einen diskretisierten Differentialoperator L_h. Das ist eine Formel, die auf Gittern definiert ist und von der Wirkung her einem Differentialoperator auf einem kontinuierlichen Gebiet entspricht. Die Formeln lauten dann $L_h\ u_h = f_h$ auf Ω_h und $u_h = g_h$ auf $\delta\Omega_h$. Der jeweilige diskretisierte Differentialoperator muss explizit hergeleitet werden.

Als Beispiel die Poisson-Gleichung

Hier ist L = - Δ, mit $\Delta\ u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$, vgl. [43]. Dabei lautet die partielle diskretisierte Differentialgleichung $\frac{1}{h^2}$ (-$u_{i-1,j}$ -$u_{i+1,j}$ -$u_{i,j-1}$ -$u_{i,j+1}$ + 4 $u_{i,j}$) = $f_{i,j}$.
Der diskretisierte Differentialoperator L_h lautet in diesem Fall: $\frac{1}{h^2}$ (-$u_{i-1,j}$ -$u_{i+1,j}$ -$u_{i,j-1}$ -$u_{i,j+1}$ + 4 $u_{i,j}$).
Als Sternoperator lautet er, vgl. [43]: $\frac{1}{h^2} \begin{bmatrix} & -1_{-1,0} & \\ -1_{0,-1} & 4_{0,0} & -1_{0,1} \\ & -1_{1,0} & \end{bmatrix} (u_{i,j})$
Der Sternoperator stellt hierbei eine verkürzte Schreibweise für einen diskretisierten Differentialoperator dar.

2.2.3 Die Eingitteraufgabe

Hier hat man nur ein Gitter, auf dem gerechnet wird. Im Folgenden wird das Jacoby- und das Gauss-Seidel-Verfahren zur Lösung vorgestellt, erst allgemein, und dann für die Poisson-Gleichung.

Glätter und Löser

Die allgemeinen Formeln Für die folgenden Formeln hat man als Literaturquellen [2], [52] oder [31]. Für eine iterative Lösung oder Glättung im Fall A x = b hat man das Jacoby-Verfahren, und dafür die Formel:

$$\forall_{k=1,\cdots}\forall_{i=1,\cdots,n}\ u_i^k = \frac{1}{a_{ii}}(b_i - \sum_{j=1;j\neq i}^{n} a_{ij}u_i^{k-1})$$

Für das Gauss-Seidel-Verfahren lautet die Formel:

$$\forall_{k=1,\cdots}\forall_{i=1,\cdots,n}\ u_i^k = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}u_i^k - \sum_{j=i+1}^{n} a_{ij}u_i^{k-1})$$

Dabei ist k die Zahl, die angibt, die wievielte Iteration gemacht wird.

Die Angabe dieser Formeln bei der Poisson Gleichung. Im Falle der Poisson Gleichung vereinfachen sich die Formeln zu den folgenden Formeln, vgl. wieder [2], [52] oder [31]. Im Jacoby-Verfahren hat man:

$\forall$ (i,j) $\in \Omega^0$ $u_{i,j}^{neu} = \frac{1}{4}(u_{i-1,j}^{alt} + u_{i+1,j}^{alt} + u_{i,j-1}^{alt} + u_{i,j+1}^{alt} - h_x\ h_y\ f(x_i,y_j))$

Für das Gauss-Seidel-Verfahren ergibt sich:

$\forall$ (i,j) $\in \Omega^0$ $u_{i,j}^{neu} = \frac{1}{4}(u_{i-1,j}^{neu} + u_{i+1,j}^{alt} + u_{i,j-1}^{neu} + u_{i,j+1}^{alt} - h_x\ h_y\ f(x_i,y_j))$

Bemerkung: Man hat hier 5-Punkte-Sterne, vgl. den Sternoperator in Abschnitt 2.2.2.

2.2.4 Mehrgitter

Gegeben sei ein einfaches Gitter. Man kommt nun jeweils zum gröberen Gitter, indem man sowohl waagerecht als auch senkrecht jeden zweiten Gitterpunkt herauslässt. Die Koordinate (f,g) eines groben Gitters entspricht der Koordinate (2f, 2g) des feineren Gitters. Wenn der Abstand der Gitterpunkte des feineren Gitters h beträgt, dann besteht der Abstand der Gitterpunkte des gröberen Gitters 2 h.
Bemerkung: Es gibt auch Semi-Vergröberungen, bei denen in Richtung einer Achse jeder zweite Gitterpunkt herausgelassen wird, in Richtung der anderen Achse aber die Abstände beibehalten werden, vgl. [43].

Nun wird das (einfache) Mehrgitterschema dargestellt:
(Bei 2 Gittern auch Zweigitterschema.)

1. [Vorglättung] u_h = Glätter(u_h, L_h, f_h) wird ν_1 Mal aufgerufen
2. [Residuum berechnen] $r_h = f_h - L_h\ u_h$
3. [Restriktion Residuum] $r_H = I_h^H\ r_h$
4. [Löser] $L_H\ e_H = r_H$
   oder löse durch das Mehrgitterschema selber
5. [Korrekturtransfer und u_H - Restriktionen] $e_h = I_H^h\ e_H$
6. [Grobgitterkorrektur] $u_h = u_h + e_h$
7. [Nachglättung] u_h = Glätter(u_h, L_h, f_h) wird ν_2 Mal aufgerufen

Nun zur Erläuterung der einzelnen Schritte:

Die Schritte 1 und 4 und 7

Sie werden bestimmt durch die Funktionen aus Abschnitt 2.2.3. Bei einem Mehrgitter wird Schritt 4 auf allen Ebenen außer der gröbsten Ebene durch ein aufgesetztes Mehrgitterschema gelöst.

Die Formel für das Residuum - Schritt 2

Sie lautet im Falle der Poisson-Gleichung für das Residuum $r_{h_{i,j}}$

$\forall$ (i,j) $\in \Omega^0$ $r_{h_{i,j}} = f(x_i,y_j) - \frac{1}{h_x h_y}(-u_{i-1,j} - u_{i+1,j} + 4\ u_{i,j} - u_{i,j-1} - u_{i,j+1})$,
vgl. das Beispielprogramm in [43].

Die Restriktion - Schritt 3

Bei diesem Abschnitt über die Restriktion werden die Matrizen unten entsprechend der Arbeit in [43] präsentiert.
Der Buchstabe h hat in dieser Arbeit eine doppelte Bedeutung: In den Mehrgitterschemata, bzw. den Formeln der Mehrgittertheorie hat er die Bedeutung eines Abstandes. In allen anderen Fällen gibt er die aktuelle Gitterebene an.
Seien die Gitterpunktabstände h und H = 2h gegeben, und sei ohne Einschränkung u die Variable, für die eine Restriktion durchzuführen ist. Die Werte sind dort $u^h_{i,j}$. Es gibt verschiedene Restriktionsformeln, je nach Position des Gitterpunktes. Es werden hier die Formeln aus [43] angegeben.

Die einfachste Form der Restriktion ist die Injektion, bei der der Feingitterwert unter dem Grobgitterwert übernommen wird:

$u^{2h}_{i,j} = u^h_{2i,2j}$

Im Folgenden wird das Full Weighting beschrieben, welches in Schritt 3 Anwendung findet. Gegeben sei dazu eine Matrix A, siehe unten. Damit gilt für alle Punkte (i,j), die nicht am (echten) Rand liegen:

$u^{2h}_{i,j} = \frac{1}{16} \sum_{k=-1}^{1} \sum_{l=-1}^{1} u^h_{2i+k,2j+l}\ a_{k+1,l+1}$ (*)
mit der Matrix

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

Zu den Gitterpunkten $u^h_{i,j}$ am Rand: Im Folgenden gelte die Formel (*) ohne die Terme, bei denen $a_{i,j} = 0$ sind. Dann gilt die Formel in der unteren rechten Ecke mit der

Matrix

$$\mathrm{A} = \begin{pmatrix} 4 & 4 & 0 \\ 4 & 4 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

und am unteren Rand mit der Matrix

$$\mathrm{A} = \begin{pmatrix} 2 & 4 & 2 \\ 2 & 4 & 2 \\ 0 & 0 & 0 \end{pmatrix}$$

und bei einer inneren Ecke, wobei die Koordinate, auf der kein Gitterpunkt existiert, sich rechts unten befindet:

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

Die restlichen Matrizen an den anderen Kanten und Ecken seien entsprechend dargestellt.

Prolongation - Schritt 5

Das ist eine Abbildung von einem groben auf ein feines Gitter. Die abstrakte Bezeichnung lautet: I_H^h. Es gibt dabei wieder verschiedene Möglichkeiten, diese durchzuführen. Hier wird die bi-quadratische Interpolation angegeben.

Sie wird berechnet über eine vierfache Formel, wobei (i,j) die Koordinaten eines Feingitterpunktes ist. Hier wird wieder ohne Einschränkung die Variable u zur Darstellung ausgewählt. Es kommt alternativ z.B. die Variable e infrage für eine Prolongation, wobei ggf. auf der linken und rechten Seite der folgenden Gleichungen verschiedene Variablen stehen können.

$$\mathrm{u}_H\ (\mathrm{i,j}) = \begin{cases} u_h(\frac{i}{2}, \frac{j}{2}) & \text{für i und j gerade} \\ \frac{1}{2}(u_h(\frac{i}{2}, \frac{j}{2}) + u_h(\frac{i}{2}, \frac{j+1}{2})) & \text{für i gerade, j ungerade} \\ \frac{1}{2}(u_h(\frac{i}{2}, \frac{j}{2}) + u_h(\frac{i+1}{2}, \frac{j}{2}) & \text{für i ungerade, j gerade} \\ \frac{1}{4}(u_h(\frac{i}{2}, \frac{j}{2}) + u_h(\frac{i+1}{2}, \frac{j}{2}) + & \\ u_h(\frac{i+1}{2}, \frac{j}{2}) + u_h(\frac{i+1}{2}, \frac{j+1}{2})) & \text{für i und j ungerade} \end{cases}$$

Das gilt, falls die Variable u übertragen wird, und entsprechende Formeln gelten, wenn e übertragen wird.

Die Grobgitterkorrektur - Schritt 6

Hier wird das feine Gitter durchlaufen, und $\mathrm{u}_{i,j}^h = \mathrm{u}_{i,j}^h + \mathrm{e}_{i,j}^h$ gerechnet.

Bemerkung: Bei Integerwerten wird bei der Division durch 2 abgerundet, vgl. die Ko-

ordinaten der Prolongationsformeln von eben.

2.2.5 FAS

Laut [43] liefert das FAS - genannt Full Approximation Scheme - Lösungen im nichtlinearen Fall. Hier wird dieses Schema präsentiert, da es dem Schema bei der adaptiven Verfeinerung sehr nahe kommt, viel näher als das normale Mehrgitterschema. Man will das Rechnen entsprechend diesem Schema als Option behalten. Zu dem Schema vergleiche man [43].

1. [Vorglättung] u_h = Glätter(u_h, L_h, f_h) wird ν_1 Mal aufgerufen
2. [Residuum berechnen] $r_h = f_h$ - L_h u_h
3. [Restriktion Residuum] $r_H = I_h^H$ r_h
4. [Löser] L_H $u_H = r_H + L_H$ I_h^H u_h
   oder aber löse durch das FAS selber
5. [Korrektur berechnen] $e_H = u_H$ - I_h^H u_h
6. [Korrekturtransfer] $e_h = I_H^h$ e_H
7. [Grobgitterkorrektur] $u_h = u_h + e_h$
8. [Nachglättung] u_h = Glätter(u_h, L_h, f_h) wird ν_2 Mal aufgerufen

Hier sind nur die Schritte 4 und 5 anders als beim Mehrgitterschema, weshalb nur diese Schritte erläutert werden müssen.

Schritte 4 und 5

Zuerst wird der Term I_h^H u_h für beide Schritte berechnet. Er wird als Injektion bestimmt und als eigenständige Matrix-Variable Iu gespeichert. Dann wird die Formel L_H I_h^H u_h durch Anwendung des Operators L_H auf die Variable Iu wie in der Residuenberechnung bestimmt, d.h. $\frac{1}{h^2}$ (-$Iu_{i-1,j}$ -$Iu_{i+1,j}$ -$Iu_{i,j-1}$ -$Iu_{i,j+1}$ + 4 $Iu_{i,j}$). Durch einfache Addition mit r_H wird die rechte Seite der Lösungsformel $r_H + L_H$ I_h^H u_h bestimmt. Der Schritt 5 folgt analog unter Verwendung der Matrix-Variablen Iu.

2.2.6 Adaptives Mehrgitter

Hier sind zwei Aufgaben zu lösen: Einmal ist die Feingitteränderung zu bestimmen. Dann ist das adaptive Schema mithilfe der Mehrgitter-Funktionen anzugeben.

Die Feingitteränderung

Durch die folgenden zwei Betrachtungen wird vermittelt, wie die Änderung des feinen Gitters erfolgt:

1. Das Verfeinerungskriterium berechnen: Auf dem groben Gitter wird bestimmt, wo darunter verfeinert wird oder nicht verfeinert wird. Es gibt da verschiedene

Möglichkeiten. An dieser Stelle wird eine Formel zur Berechnung angegeben:

$\tau_H^h = \mathrm{L}_H\ \mathrm{I}_h^H\ \mathrm{u}_h$ - $\mathrm{I}_h^H\ \mathrm{L}_h\ \mathrm{u}_h$

Falls die rechte Seite aus dem FAS berechnet wird, kann diese Formel ohne viel Aufwand mitberechnet werden, vgl. Abschnitt 3.6, nach einer Bemerkung in [43], dass das geht. Allerdings ist diese Vereinfachung nur bei einer einzigen Verfeinerungsstufe anwendbar - vgl. den Abschnitt 3.6.

2. Das abgeleitete Verfeinerungsgitter bestimmen: Ist (f,g) der zu verfeinernde Gitterpunkt, dann wird um den Gitterpunkt (2f,2g) des feinen Gitters ein 5 x 5 Gitter gelegt, wobei (2f,2g) der Mittelpunkt dieses Gitters ist.

Das Mehrgitterschema mit adaptiven Gitterverfeinerungen

Hier seien G_H das grobe Gitter, sowie G_h das feine Gitter vor der adaptiven Verfeinerung, und G_h' das neu gebildete feine Gitter. Dabei werden die neu entstandenen Punkte durch ein Markierungsfeld gekennzeichnet.
Ebenfalls neu hinzu kommt hier die Randinterpolation RI_H^h, die in Abschnitt 4.3.9 ausgeführt wird. Sie basiert auf den Grobgitterwerten, für die eine Polynominterpolation bis zum dritten Grad erfolgt. Diese Interpolationswerte werden dabei auf das feine Gitter aufgebracht, und zwar an den Positionen zwischen den Grobgitterstellen. Damit lautet das adaptive Schema, dass sich von dem Schema in [36] durch die Verwendung von G_h' unterscheidet:

1. [Vorglättung] u_h = Glätter(u_h, L_h, f_h) wird ν_1 Mal auf G_h aufgerufen
2. [Residuum berechnen] $\mathrm{r}_h = \mathrm{f}_h$ - $\mathrm{L}_h\ \mathrm{u}_h$ auf G_h
3. [Restriktion Residuum] $\mathrm{r}_H = \mathrm{I}_h^H\ \mathrm{r}_h$ auf $\mathrm{G}_H \cap \mathrm{G}_h$
4. [Werte der rechten Seite berechnen]
   Berechne F_H:= $\begin{cases} f_H & \text{für } G_H - G_h \\ r_H + L_H I_h^H u_h & \text{für } G_H \cap G_h \end{cases}$
5. [Löser] $\mathrm{L}_H\ \mathrm{u}_H = \mathrm{F}_H$ auf G_H
   oder aber durch dieses adaptive Schema. Bei einem vollständigen Gitter können das auch das Mehrgitterschema oder das FAS sein.
6. [Korrektur berechnen] $\mathrm{e}_H = \mathrm{u}_H$ - $\mathrm{I}_h^H\ \mathrm{u}_h$ auf $\mathrm{G}_H \cap \mathrm{G}_h$
7. [Korrekturtransfer und u_H - Restriktionen] $\mathrm{e}_h = \mathrm{I}_H^h\ \mathrm{e}_H$ auf $\mathrm{G}_h \cap \mathrm{G}_h'$ (alte Punkte)
   $\mathrm{u}_h = \mathrm{I}_H^h\ \mathrm{u}_H$ auf G_h' - G_h (neue Punkte)
8. [Grobgitterkorrektur] $\mathrm{u}_h = \mathrm{u}_h + e_h$ auf $\mathrm{G}_h \cap \mathrm{G}_h'$
9. [Randinterpolation] $\mathrm{u}_h = \mathrm{RI}_H^h\ \mathrm{u}_H$ auf G_h'
10. [Nachglättung] u_h = Glätter(u_h, L_h, f_h) wird ν_2 Mal auf G_h' aufgerufen

Der Terminus Grundgitter

Unter einem Grundgitter sei das Gitter zu verstehen, bei dem einerseits die Vergröberungsebenen darüber liegen, und andererseits die adaptiven Verfeinerungsebenen darunter liegen.

2.3 Vorüberlegungen zum parallelen Einfachgitter

Hier werden Überlegungen zur Parallelisierung auf einem Gitter vorgenommen, auch als Vorüberlegung zum parallelen Mehrgitterverfahren. Was an Parallelisierung darüber hinausgeht, wird in den sich anschließenden Kapiteln ausführlich behandelt.

2.3.1 Das Jacoby- und Gauss-Seidel-Verfahren parallel

Es gibt zwei Ziele für diesen Abschnitt: Einmal aufzeigen, wie man Idle-Zeiten für die Kommunikation der Ränder vermeiden kann, und dann zeigen, wie die Glätter und Löser für das Mehrgitterverfahren, die Jacoby- und die Gauss-Seidel-Methode, realisiert werden können.

2 Varianten zur Vermeidung von Idlezeiten

Die Idee dabei ist, während der Kommunikation des Randes zu rechnen, damit nicht gewartet werden muss. Es sollen dadurch Idlezeiten vermieden werden.
Man kann ganz allgemein festlegen, dass man nach dem folgenden Prinzip vorgeht, welches in 3 Schritten vor sich geht:

1. Die Kommunikation des Randes beginnen.

2. Das Innere des Gitters wird berechnet.

3. Man schließt die Kommunikation hinterher ab. Man hat dann keine oder eine viel kürzere Wartezeit.

Es gibt dazu aber zwei verschiedene Möglichkeiten:

1. Variante 1:
   Erst wird der Rand des Gitters berechnet. Dann werden die Werte dieses Randes gesendet. Dann wird das Innere des Gitters berechnet. Und am Ende wird die Kommunikation abgeschlossen und der empfangene Rand außerhalb des Gitters eingefügt.

2. Variante 2:
   Erst wird der Rand des Gitters verschickt. Dann wird das Innere des Gitters berechnet. Dann wird dabei der Außenrand empfangen. Der besteht aus den Punkten, die das eigene Gitter von außen her begrenzen. Am Ende wird der Rand des Gitters berechnet.

Auswahl zwischen den Varianten

Bei Variante 1 muss auch nach der Durchführung des Rechnens immer zusätzlich zum eigenen Gebiet der Außenrand gültig bleiben. Beim Lastausgleich müssten dann nicht nur die Werte auf dem eigenen Gebiet transferiert werden, sondern auch die auf dem Außenrand. Bei Variante 2 ist dies nicht nötig. Aus diesem Grund wird im Folgenden, wie auch im Programm, Variante 2 gewählt.

Der Ablauf bei der parallelen Jacoby-Methode

1. Starte die Randübertragung
2. Berechne für das innere Gebiet des Gitters die Formel für $u_{i,j}^{neu}$ des Jacoby-Verfahrens
3. Warte auf das Ende der Randübertragung
4. Berechne für den Rand die Formel für $u_{i,j}^{neu}$ des Jacoby-Verfahrens

Die Möglichkeiten der Gauss-Seidel Parallelisierung

Für folgende Überlegungen vergleiche man [52] oder [31]: Allgemein kann man die Gauss-Seidel-Methode nicht parallelisieren. Damit ist ein Durchlauf einer äußeren Schleife von oben nach unten, und einer inneren Schleife von links nach rechts, gemeint. In dieser Reihenfolge müssten die Formeln für die Gauss-Seidel-Methode in Abschnitt 2.2.3 berechnet werden.
Es gibt jedoch diese 2 Möglichkeiten, eine Parallelisierung vorzunehmen:

1. Man macht eine Block Gauss-Seidel-Parallelisierung: Man rechnet mit dem Gauss-Seidel Verfahren für alle Punkte, die auf einem Kern liegen, also von links nach rechts, und von oben nach unten - ohne die Wechselwirkungen der Gauss-Seidel-Formel zwischen diesen Blöcken zu berücksichtigen. Die einzige Ausnahme bildet der normale Randaustausch, der aber vor oder nach der Berechnung der Blöcke erfolgt, vgl. [31]. Das geht allgemein, hat aber andere numerische Eigenschaften als eine strenge Gauss-Seidel-Methode.
2. Bei einem 5 Punkte-Stern, wie er bei der Poisson-Gleichung vorliegt, geht eine Methode mit dem Namen Gauss-Seidel red-black Verfahren: Man rechnet erst auf allen Prozessoren die Punkte (f,g) mit (f+g) mod 2 = 1 (red Punkte), und danach die Punkte mit (f+g) mod 2 = 0 (black Punkte). Hier muss man bei der Berechnung der red Punkte den black Rand übertragen - und umgekehrt. Dass man 'diese Nummerierung' der Punkte wählen kann, kann man z.B. in [2] nachlesen.

Die red-black Variante der Gauss-Seidel-Methode wird im Folgenden zur Parallelisierung gewählt.

Der Ablauf der parallelen Gauss-Seidel-Methode

1. Die Randübertragung für die red Punkte starten
2. Berechnen der black Punkte im inneren Bereich des Gitters

3. Die red Randübertragung abwarten und abschließen
4. Berechnen der black Punkte am Rand
5. Die Randübertragung für die black Punkte starten
6. Berechnen der red Punkte im inneren Bereich des Gitters
7. Die black Randübertragung abwarten und abschließen
8. Berechnen der red Punkte des Randes

Nun noch die Begründung, dass man so korrekt einen parallelen red-black Gauss-Seidel Schritt durchführt: Für black Punkte am Rand müssen die red Punkte am Rand übertragen worden sein, da die black Punkte hiervon abhängig sind. Dasselbe gilt für die red Punkte in Abhängigkeit der black Punkte. Das wird bei der eben dargestellten Reihenfolge erreicht.

Kapitel 3

Der Aufbau des Programms

3.1 Was für ein Programmtyp dieses Programm ist

Diese Betrachtung erscheint in Variation mehrfach in dieser Arbeit, da sie zur Abgrenzung von anderen Mehrgitterparallelisierungen sehr wichtig ist.

Bei diesem Parallelisierungsprogramm für dynamisch adaptive Mehrgitterverfahren ist die Idee die, einzelne Gitterpunkte beliebig auf die Kerne zu verteilen mit der im nächsten Abschnitt beschriebenen Ausnahme. Um dieser Aufgabe gerecht zu werden, liegt insbesondere keine Patch-Struktur vor, bei der komplette Rechtecke auf einem Kern liegen. Die einzelnen Gitterpunkte werden dann bei Bedarf automatisch umverteilt.

Es gibt eine Ausnahme davon, die Gitterpunkte völlig beliebig zu verteilen: Gitterpunkte, die übereinander liegen, befinden sich auf demselben Kern. Deswegen bedarf es wohl einer Randkommunikation auf einem Gitter, nicht aber zwischen den Gittern. Das verringert die Komplexität des Programmes in sinnvoller Weise.

3.2 Was in diesem Kapitel erreicht wird

Es werden viele wichtige Bestandteile des Programms bzw. der Klassen vorgestellt, sowie ihr Zusammenspiel erläutert, was hier 'Verknüpfungen' genannt wird. Diese Verknüpfungen werden in einer Übersicht dargestellt, vgl. Abbildung 3.1. Dann werden die Verknüpfungen in diesem Kapitel erläutert.

Dieses Kapitel ist wichtig, da es eine abstrakte Darstellung der Klassenbeziehungen enthält.

3.3 Vorbemerkung zu den Kapiteln 3 bis 10

Im de Kapiteln 3 bis 6 wird das Programm erläutert. Entsprechend obiger Annahmen werden zuerst die Strukturen und ihre Verknüpfungen dargestellt, die Algorithmen

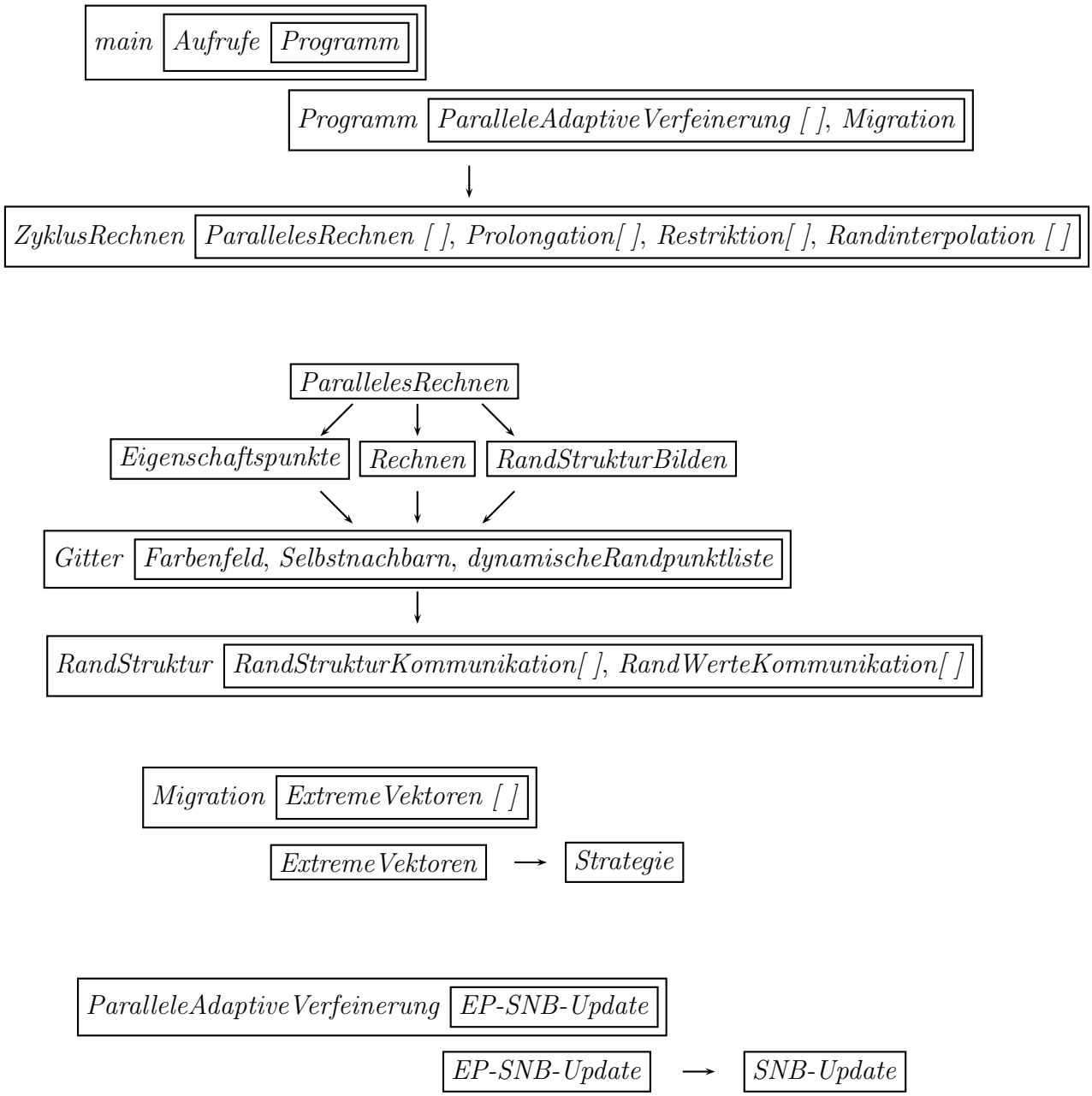


Abbildung 3.1: Die Klassenübersicht über das Programm. Dabei zeigt der Pfeil jeweils von der abgeleiteten Klasse auf die Basisklasse. Innerhalb der Box definiert die Klasse vorne die Strukturen in der inneren Box. Dabei bedeutet *[ ]*, dass von der Struktur ein Feld angelegt wird.

hingegen in den folgenden Kapiteln 4 bis 6.

Bemerkung zu der Sprechweise: Man spricht davon, dass eine Klasse Elternklasse oder Basisklasse oder Oberklasse ist, wenn die davon abgeleitete Klasse, die auch Kindklasse genannt wird, auf alle 'public' oder 'protected' definierten Funktionen zugreifen kann, vgl. [77]. Es sei für diese Arbeit vereinbart, dass die Rede davon ist, dass die Kindklasse 'Zugriff' hat auf ihre Elternklasse.

3.3.1 Vorbemerkung zu diesem Kapitel

In diesem Kapitel werden in den Abschnitten 3.4 bis 3.7 die verschiedenen Programmteile bzw. Klassen eingeführt und ihre Wechselbeziehung dargestellt, also insbesondere wie sie untereinander verknüpft sind, und auch über welche Objekte das geschieht. Algorithmen werden in diesem Kapitel, bis auf eine Ausnahme, nicht dargestellt. Dafür sind die folgenden Kapitel 4 bis 6 da.

Inhaltlich beginnt dieses Kapitel mit dem Abschnitt 3.4, bei dem eine Struktur Mehrgitter vorgestellt wird, auf der jedes einzelne Gitter eine Punktmenge hat, mit Variablen und diversen Datenstrukturen zur sequentiellen und parallelen Behandlung. Es geht dabei sozusagen um den diskreten Raum, der dem Programm zu Grunde liegt.
Dazu werden die Komponenten eines Gitters dargestellt. Aufgrund der Verknüpfung der Gitterteile der verschiedenen Ebenen entsteht das Mehrfachgitter. Dazu: Verknüpft werden die Gitterteile in der Klasse *ParallelesRechnen*. Die Gitter der verschiedenen Ebenen werden zugreifbar in den Klassen *ZyklusRechnen* und *Programm*. Genauere Informationen zu diesen Klassen kommen später.

Im nächsten Abschnitt 3.5 wird dann auf diesem Mehrfachgitter gerechnet. In diesem Abschnitt werden kaum eigene Datenstrukturen verwendet, sondern es werden die Datenstrukturen des Mehrgitters aus Abschnitt 3.4 dazu herangezogen.

In Abschnitt 3.6 geht es darum, das Gitter abzuändern. Dazu sind drei Objekte nötig: Einmal die Bestimmung der Eigenschaftspunkte. Das sind die Punkte, für die eine Verfeinerungsentscheidung getroffen wurde. Dann die *ParalleleAdaptiveVerfeinerung* und der *Lastausgleich*. Die Algorithmen dafür werden in derselben Reihenfolge in den Kapiteln 4, 5 und 6 beschrieben, wobei die *ParalleleAdaptiveVerfeinerung* und der *Lastausgleich* jeweils das ganze Kapitel ausmachen.

Im letzen Abschnitt 3.7 werden alle Bestandteile verbunden, und zwar in den Klassen *Programm* und *ZyklusRechnen*.

Bemerkung: Bei der Darstellung, wie die Klassen verbunden sind, werden auch die verbindenden Elemente zwischen den Klassen eingeführt, wie beispielsweise die Instanz P_k^+ der Klasse *PunktFeldliste* in der Klasse *ParalleleAdaptiveVerfeinerung*. Diese verbindenden Elemente sind Klassen oder andere Datenstrukturen wie Felder, die Klassen verbinden.

3.3.2 Vorbemerkung zu Kapitel 4

Hier werden die Algorithmen zu den Strukturen aus diesem Kapitel vorgestellt, bis auf die parallele adaptive Verfeinerung und den Lastausgleich. Die Zahl der Algorithmen, die in Kapitel 4 präsentiert werden, ist recht groß, sodass an dieser Stelle selbst eine kurze Darstellung derselben nicht sinnvoll ist.

3.3.3 Vorbemerkung zu den Kapiteln 5 und 6

Hier werden die Algorithmen zur parallelen adaptiven Verfeinerung und des Lastausgleiches, bestehend aus Precomputation, Balancing und Migration, dargestellt. Darüber hinaus werden zu Beginn jeweils enthaltene Substrukturen dieser Teile dargestellt, und notwendige Beweise für die Korrektheit der Algorithmen vorgenommen. Unter Substrukturen sind hier Klassen wie z.B. *EP-SNB-Update* in der Klasse *ParalleleAdaptiveVerfeinerung* zu verstehen (die allerdings erst später in Kapitel 5 erklärt wird). Das sind Klassen, die gewisse Aufgaben übernehmen, um die zentralen Klassen wie *ParalleleAdaptiveVerfeinerung* oder *Lastausgleich* zu unterstützen.
Bemerkung: Für den Lastausgleich hat man nicht nur die Klasse *Lastausgleich*, sondern auch noch Klassen für die Vorberechnungen und das Balancing, sowie Funktionen in der Hauptklasse *Programm*.

3.3.4 Vorbemerkung zu den Kapiteln 7 bis 10

Die eigentliche Darstellung des Programms ist mit Kapitel 6 abgeschlossen. In Kapitel 7 wird der Zeitbedarf, den der Balancer hat, theoretisch untersucht. In Kapitel 8 geht es um die Messungen des Programms. Zuvor geschieht noch eine Überlegung zur Dynamik der Gitterverfeinerungen beim Verfeinerungskriterium der Experten für Mehrgittertheorie. In Kapitel 9 wird diese Arbeit mit anderen Arbeiten mit ähnlichen Fragestellungen verglichen, um diese Arbeit einordnen zu können. Und im Kapitel 10 wird darauf eingegangen was geleistet wurde und welche Arbeiten noch gemacht werden könnten.

3.4 Paralleles dynamisch änderbares Multi-Gitter

Hier wird das parallele dynamisch änderbare Gitter eingeführt. Dabei geht es um die auf die Kerne verteilten Punktmengen, die Ränder sowie auch die Wertemengen der Variablen. Und es wird auch schon davon die Rede sein, wie diese Datenstrukturen, die in der Klasse *ParallelesRechnen* alles an Informationen für eine Gitterebene bereitstellen, in der Klasse *ZyklusRechnen* als Mehrgitter zusammengefasst werden.
Worum es in diesem Abschnitt insbesondere nicht geht ist die Änderung des verteilten Gitters durch die parallele adaptive Verfeinerung sowie den sich anschließenden Lastausgleich. Es geht hier auch noch nicht darum, wie alle Bestandteile verbunden werden, was in Abschnitt 3.7 geleistet wird.

3.4.1 Die Bestandteile/Klassen des Programms für eine Gitterebene

In diesem Abschnitt geht es um die Klassen für eine Gitterebene, die es ermöglichen, das Gitter dynamisch zu ändern.
Zuerst eine Auflistung der dafür zu erfüllenden Aufgaben bzw. zu speichernden Informationen:

1. Gebraucht wird eine Liste der Gitterpunkte. Diese muss dynamisch änderbar sein.

2. Eine Besonderheit bei der Programmierung mit einzeln zu verschickenden Gitterpunkten stellt das *Farbenfeld* dar. So weiß jeder Kern, auf welchem Kern sich ein Gitterpunkt befindet - in einem gewissen zulässigen Bereich.

3. *Selbstnachbarn*: Bezüglich aller Himmelsrichtungen wird jeweils durch ein Bit festgelegt, ob in dieser Himmelsrichtung ein eigener Punkt existiert.

4. *DynamischeRandpunkteliste*: Man hat eine Liste mit allen Punkten am Rand. Als Liste können damit alle Randpunkte durchlaufen werden. Und sie ändert sich mit dem Einfügen und Entfernen von Punkten.

5. Die *Außen-* und *Innenrandlisten*: Hier sind die Koordinaten der Punkte angegeben, deren Werte verschickt oder empfangen werden. Die Listen werden dabei in Bezug auf jeden anderen Kern definiert.

6. Der Randaustausch bezüglich Variablen.

7. Die Aktualisierung der *Außen-* und *Innenrandlisten*.

Die Gitterpunkteliste

Die zur Speicherung von Gitterpunkten verwendete Struktur bekommt den Namen *Feldliste*. Sie besteht aus einer Liste und einem 2-dimensionalen Feld. Sie leitet nach der Klasse *Gitter* ab, ist also von dort zugreifbar. Man hat die Deklaration *'class Gitter :: virtual public Feldliste'*.

Auf die Feldliste zugreifen kann man durch die Funktionen *füge-punkt-hinzu(f,g)*, bzw. *entferne-punkt(f,g)*. Ferner kann man sie durchlaufen mittels der Funktionen *setze-auf-anfang* (positioniere die Liste an den Anfang), *bestimme-naechstes* (bestimme den nächsten Gitterpunkt), *ist-durchlaufen* (ist die Liste am Ende angekommen?), *lese-f*, *lese-g* (lese die Koordinaten). Wie diese Klasse funktioniert, insbesondere das Zusammenspiel von Liste und Feld, vgl. 4.2.1.

Man braucht diese Struktur, weil die Gitter sich ändern können sowohl wegen der adaptiven Verfeinerung als auch der Migration von Punkten.

Bemerkung: Es gibt auch noch die *PunktFeldliste*. Sie erfüllt denselben Zweck, wobei aber die doppelt-verkettete Liste ersetzt wird durch ein Array von Koordinaten - vgl. Abschnitt 5.4.3.

Das *Farbenfeld*

Es ist ein zweidimensionales Feld, welches verzeichnet, auf welchem Kern sich ein Gitterpunkt befindet. Es gibt einen Gültigkeitsbereich für das Farbenfeld, nämlich das Gebiet des eigenen Kernes sowie das Gebiet seines umgebenden Randes.

Der Zugriff erfolgt direkt auf das zweidimensionale Feld: Man schreibt hinein, bzw. liest es aus. Es geht darum, was und wo bei den Algorithmen geschrieben und auf welchen Koordinaten gelesen wird, insbesondere bei der parallelen adaptiven Verfeinerung, dort vor allem bei dem Farbenupdate, und auch bei der Migration. Diese Aufgaben zu erfüllen ist komplex - die Datenstruktur selber, bzw. das Lesen und Schreiben in diese Struktur sind es nicht.

Bemerkung 1: In diesem Zusammenhang ist auch der Begriff der *Farbe* zu erläutern. Er kennzeichnet die Identität eines Kerns, insbesondere die Zugehörigkeit eines Punktes auf einem Kern beim *Farbenfeld*. Dabei nimmt die Farbe auf einer Position einen negativen Wert (in der Regel -1) an, wenn sich dort kein Punkt befindet.

Bemerkung 2: Davon abzuheben ist der Begriff des Rangs, der die eigene ID eines Kerns angibt. Er liegt zwischen 0 und Größe-1, wobei die Größe die Anzahl der Kerne angibt, für die das Programm gestartet wird.

Die Datenstruktur *Selbstnachbarn*

Zu jedem Punkt wird eine Zahl zwischen 0 und 255 abgespeichert. Dabei steht für jede Himmelsrichtung ein Bit, vgl. Definition 4.1, sodass man die Randsituation für alle Himmelsrichtungen durch eine einzige switch-case Anweisung, allerdings mit 256 Fällen, erkennen kann. Das wird von vielen Klassen genutzt, wobei diese 256er switch-case Anweisungen auch in Bezug auf die später erklärte Datenstruktur *EP-SNB-Update* zur Anwendung kommen.

Definiert wird die Datenstruktur *Selbstnachbarn* in der Klasse *Gitter*. Die Schnittstelle dort ist so, dass zu einer Koordinate (f,g) entweder ein Gitterpunkt eingefügt oder entfernt wird. Der Zugriff auf das Feld zum Lesen erfolgt durch eine Abfrage direkt in das zweidimensionale Feld *Selbstnachbarn*.

DynamischeRandpunkteliste

Diese Struktur basiert auf einer Feldliste, die mit der Struktur *Selbstnachbarn* wie folgt verbunden ist: Hier sind alle die Punkte eingetragen in eine Feldliste, die am Rand liegen, d.h. für die *Selbstnachbarn* ungleich 255 ist. Wie das geht vgl. Abschnitt 4.2.4.

Zur Schnittstelle gehört das Eintragen oder Entfernen von Punkten. Ob sie dann am Rand liegen, wird automatisch festgestellt über *Selbstnachbarn*, und der Durchlauf durch die Liste erfolgt wie bei der *Feldliste*.

Die *DynamischeRandpunkteliste* ist ebenfalls in der Klasse *Gitter* definiert.

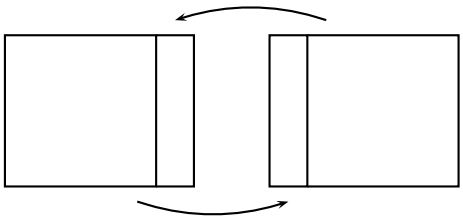


Abbildung 3.2: Die Innen- und Außenrandlisten für die Randkommunikation

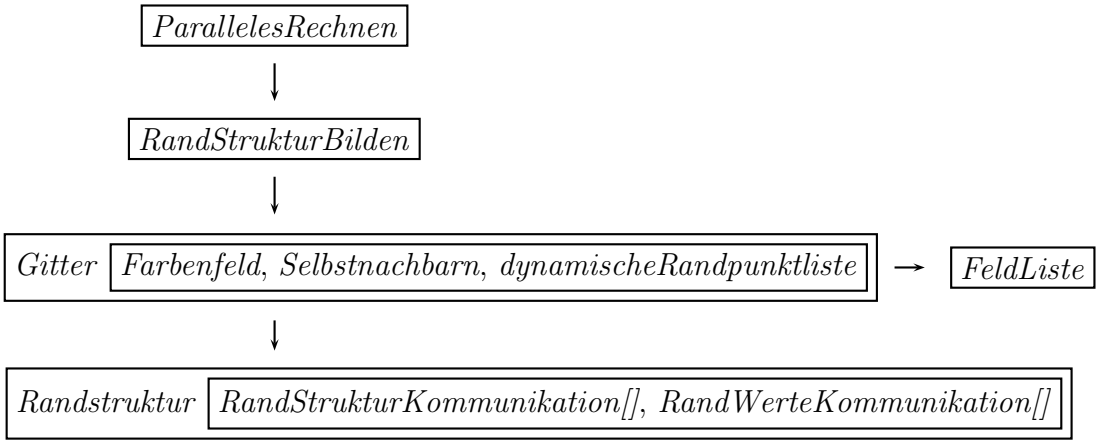


Abbildung 3.3: Die Eingitter-Verknüpfungen des Abschnitts 3.4 aufzeigen. Dabei zeigt der Pfeil jeweils von der abgeleiteten Klasse auf die Basisklasse. Innerhalb der Box definiert die Klasse vorne die Datenstrukturen in der inneren Box. Dabei bedeutet *[ ]*, dass von der Datenstruktur ein Feld angelegt wird.

Die Außen- und Innenrandlisten

Vorbemerkung 1: Für die Kommunikation an den Gitterrändern hat man Außen- und Innenrandlisten, vgl. Abbildung 3.2. Dabei werden die Werte entsprechend den Innenrandlisten verschickt und bzgl. der Außenrandlisten entpackt.

Vorbemerkung 2: Um diesen Abschnitt zu verstehen, ist ein Blick auf Abbildung 3.3 sinnvoll (vgl. z.B. die Klassen *RandstrukturBilden*, *Selbstnachbarn* und *dynamische-Randpunktliste*).

Zuerst ist zu erwähnen, dass hier eine Fallunterscheidung gemacht wird, ob die Listen als Punkte oder als Vektoren zuzüglich von Punkten bestimmt und gespeichert werden. Bei der vektoriellen Behandlung erfolgt eine vektorielle Bestimmung des Randes, sowie sein Verschicken, während bei der 'Sortierung' nur der Innen- und Außenrand punktweise bestimmt, und dann beide sortiert werden, vgl. Abschnitt 4.2.5. Genauer: Die Farbwerte selber müssen auf dem Außenrand, der aus den Punkten besteht, die direkt an das eigene Gitter des Kernes grenzen, bekannt sein, und das ist ohne Kommunikation nicht zu erreichen. Bei der Sortierung ist aber nur diese Farb-Information nötig. Weitergehende Kommunikation ist demnach nicht erforderlich.

Ferner: Diese Randlisten werden geführt in Bezug auf jeden anderen Kern, sind aber genau dann leer, wenn es keinen gemeinsamen Rand gibt, also kein benachbartes Punk-

tepaar existiert zwischen dem eigenen Kern und dem jeweils anderen Kern.

Die Innenrandliste gibt an, für welche Koordinaten man Werte in den Buffer, der verschickt wird, eintragen muss. Die Außenrandliste legt fest, wohin die übertragenen Werte kopiert werden. Beide Listen müssen also die Punktmengen bzw. Vektormengen in derselben Reihenfolge abgespeichert haben.

Die Idee dabei ist: Wurden diese Listen einmal generiert, brauchen die Koordinaten selber nicht mit übertragen werden, wenn die Randwerte der verschiedenen Variablen übermittelt werden sollen. Allerdings müssen diese Listen wieder aktualisiert werden, wenn sich die Ränder ändern, also nach der parallelen adaptiven Verfeinerung bzw. dem Lastausgleich.

Der Zugriff auf diese Listen erfolgt als auf eindimensionale Koordinatenfelder, wobei die Koordinaten Koordinaten von Punkten oder Vektoren sind. Bei der Randbestimmung durch Kommunikation werden vertikale und horizontale Vektoren sowie die verbliebenen Punkte gespeichert. Bei der Randbestimmung durch Sortierung hat man nur eine Punktspeicherung.

Erzeugt bzw. aktualisiert werden sie durch spezielle Algorithmen, vgl. 4.2.5. Erwähnt werden soll hier schon, dass die Randpunkte vor der Vektorisierung nebst Kommunikation oder der Sortierung in der Klasse *Randstruktur* gesammelt werden. Diese wird durch *'class Gitter : virtual public Randstruktur'* zugreifbar für die Klassen, die die Klasse *ParallelesRechnen* verwenden.

Gespeichert werden die Vektoren und Punkte bei der Randbestimmung durch Kommunikation in der Klasse *RandstrukturKommunikation*, während die Punkte bei der Randbestimmung durch Sortierung in der Klasse *RandwerteKommunikation* gespeichert werden, vgl. dazu auch Abbildung 3.3. Dort geschieht dann auch jeweils die Übertragung der Werte des Randes. Beide Klassen sind in der Klasse *Randstruktur* als eindimensionale Felder definiert, damit sie zum Randaustausch in Bezug auf jeden anderen Kern zur Verfügung stehen.

Der Randaustausch der Randwerte der verschiedenen Variablen in den Klassen *RandstrukturKommunikation* und *RandwerteKommunikation*

Beim Randaustausch für Variablen bzw. Variablenkombinationen - bei letzterem Ausdruck ist damit der gemeinsame Austausch von Randwerten zweier Variablen wie u_0 und r_0 gemeint - kann der ganze Rand ausgetauscht werden, oder es werden nur die red bzw. black Punkte ausgetauscht. Um die Randwerte zu übertragen werden die Buffer geschrieben, übermittelt, es wird getestet, ob sie angekommen sind, und sie werden wieder gelesen. Verwendet werden dabei die Außen- und Innenrandlisten.

Zur Schnittstelle für den Randaustausch in der Klasse *Randstruktur*: Der Aufruf geschieht für schreiben, übermitteln, testen aufs Ankommen sowie Lesen in der Klasse *Randstruktur*. Die Algorithmen dazu stehen in der Klasse *Randstruktur* wie vorher, sowie entweder in der Klasse *RandstrukturKommunikation* bzw. der Klasse *RandWer-*

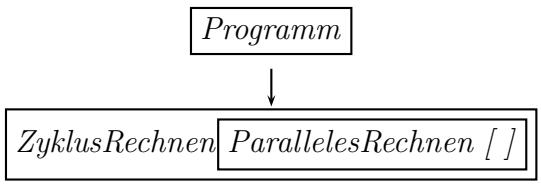


Abbildung 3.4: Die Mehrgitter-Verknüpfungen des Abschnitts 3.4 aufzeigen. Dabei zeigt der Pfeil jeweils von der abgeleiteten Klasse auf die Basisklasse. Innerhalb der Box definiert die Klasse vorne die Strukturen in der inneren Box. Dabei bedeutet *[ ]*, dass von der Struktur ein Feld angelegt wird.

teKommunikation, die in der Klasse *Randstruktur* deklariert worden sind, vgl. Abbildung 3.3.

Bemerkung: Es gibt neben dem normalen Randaustausch, z.B. bei der Berechnung des Residuums, auch noch einen grob-fein oder fein-grob Randaustausch, z.B. bei der Prolongation: Dabei wird der Rand auf dem feinen Gitter durchlaufen, aber Grobgitterwerte werden übertragen, sofern das möglich ist.

Die Aktualisierung der Außen - und Innenrandliste - als Befehl

Die Aktualisierung geschieht per Befehl der Klasse *RandStrukturBilden*. Die aufzurufenden Funktionen lauten: *vorbereiten*, *vorbereiten2* und *vorbereiten3*. Dabei werden Funktionen der Klasse *Gitter* wie *randstruktur_bilden* und *randstruktur_bilden_beide* aufgerufen. *randstruktur_bilden* sammelt die Punkte des Innenrandes, *randstruktur_bilden_beide* die Punkte von Innenrand und Außenrand.
Zu den konkreten Algorithmen der Aktualisierung der Randlisten vgl. Abschnitt 4.2.5.

Der Update der Kerntopologie

Indem geprüft wird, ob die Außenrandliste oder Innenrandliste zu einem anderen Kern leer ist oder nicht, weiß man, ob bzgl. der Punktnachbarschaft eine Verbindung mit diesem Kern da ist oder nicht. Diese Kern-Topologie ist also direkt aus den Randlisten abzulesen.

3.4.2 Die Verknüpfungen bzw. das Wechselspiel der Klassen in Bezug auf das in dem Abschnitt 3.4.1 Behandelte

Man vergleiche die Darstellung dieses Abschnitts 3.4.2 mit den Abbildungen 3.3 und 3.4.

Die Verknüpfungen des parallelen Eingitters, vgl. Abbildung 3.3

Ein Teil der Verknüpfungen wurde schon angegeben.

In der Klasse *Gitter* sind die Klassen *Farbenfeld*, *Selbstnachbarn* und *DynamischeRandpunkteliste* definiert. Sie enthält die Funktionen der Klasse *Feldliste* via Vererbung.

In der Klasse *Randstruktur* sind die Klassen *RandstrukturKommunikation* und *Rand-*

WerteKommunikation definiert. Sie leitet ab zur Klasse *Gitter*, die wiederum zur Klasse *RandStrukturBilden*, und die zur Klasse *ParallelesRechnen*. Mit der Klasse *Parallelles-Rechnen* hat man Zugriff auf alle eben dargestellten Methoden.

Die Verknüpfungen des Eingitters zum Mehrgitter, vgl. Abbildung 3.4

Nicht so ausführlich behandelt, für diesen Abschnitt aber wesentlich, ist die Bildung des Mehrgitters.
Ein eindimensionales Feld von *ParallelesRechnen* wird in der Klasse *ZyklusRechnen* angelegt, womit man ein Mehrgitter etabliert. Diese Klasse leitet nach der Hauptklasse *Programm* ab, die somit einen Zugriff auf das Mehrgitter sowie die Methoden der Klasse *ZyklusRechnen* hat. Was der Zugriff der Klasse *Gitter* auf die Klasse *ZyklusRechnen* bedeutet - siehe Abschnitt 3.5.

3.5 Rechnen auf dem Multi-Gitter

Das Rechnen bezieht sich auf das Spektrum vom einfachen sequentiellen Rechenschritt bis hin zu den parallelen *MLAT-Funktionen*, am Ende dieses Abschnittes.
Vorbemerkung: Die hier aufgezeigten Rechenschritte beziehen sich auf das zuerst implementierte adaptive Mehrgittersystem in Bezug auf die Poisson-Gleichung. Es wird im Sinne der Mehrgittertheorie dabei auf die Literatur von [43] und [82] zurückgegriffen.

Dieser Abschnitt teilt sich auf in 3 Bereiche:

Zuerst werden die Funktionen aufgelistet, die rein sequentiell ausgeführt werden. Diese sind definiert in der Klasse *Rechnen*, die zwischen der Kindklasse *paralleles Rechnen* und der Elternklasse *Gitter* liegt, und zwar über das Mittel Vererbung, vgl. Abbildung 3.5.
In einem zweiten Schritt werden dieselben Funktionen behandelt, die aber parallel ausgeführt werden, indem neben sequentiellen Schritten auch ein Randaustausch erfolgt.
In einem dritten Schritt werden mithilfe dieser Funktionen die *MLAT-Funktionen* gebildet. MLAT steht für Multi-Level-Adaptive-Technics. Die MLAT-Funktionen werden vorgestellt in Abschnitt 3.5.3.

Die eigentlichen Algorithmen dazu finden sich in Kapitel 4 im Abschnitt 4.3.

3.5.1 Die sequentiellen Funktionen

Die Klassen *Rechnen*, *Prolongation*, *Restriktion* und *Randinterpolation*

Von der Klasse *Gitter* wird die Klasse *Rechnen* abgeleitet, nach der die Klasse *ParallelesRechnen* abgeleitet wird, vgl. Abbildung 3.5. Damit hat man in der Klasse *Rechnen* über die Klasse *Gitter* Zugriff auf die Variablen der Klasse *Randstruktur*. Dabei insbesondere auf die Variablen, für die gerechnet wird. Andererseits hat die Klasse *ZyklusRechnen* über die Klasse *ParallelesRechnen* Zugriff auf die in *Rechnen* definierten Rechenfunktionen, vgl. die Abbildungen 3.4 und 3.5.
In dieser Klasse *Rechnen* sind Funktionen zur sequentiellen Berechnung von Glättern -

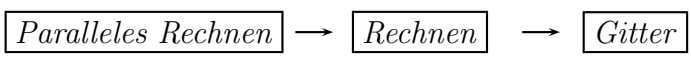


Abbildung 3.5: Die Klasse *Rechnen* im Sinne der Vererbung. Dabei zeigt der Pfeil jeweils von der abgeleiteten Klasse auf die Basisklasse. Innerhalb der Box definiert die Klasse vorne die Strukturen in der inneren Box. Diese Abbildung gehört zu Abschnitt 3.5.1.

Abbildung 3.6: Die Verknüpfungen von *Prolongation* und *Restriktion* und *Randinterpolation*. Innerhalb der Box definiert die Klasse vorne die Strukturen in der inneren Box. Dabei bedeutet *[ ]*, dass von der Struktur ein Feld angelegt wird.

diese explizit oder semi-implizit (red-black Varianten), Residuen, Grobgitterkorrektur usw. vorhanden. Beim Aufruf dieser Funktionen werden Punktmengen übergeben. Die übergebenen Parameter sind (f*,g*,anzahl), wobei f* und g* Koordinatenlisten sind, und Anzahl die Länge dieser Koordinatenlisten angibt.
Deklariert sind diese Funktionen in der Klasse *Rechnen*, und sie können von der Klasse *ParallelesRechnen* aus aufgerufen werden, vgl. Abbildung 3.5.

Die Klassen *Prolongation* und *Restriktion*, vgl. Abbildung 3.6, ermöglichen einen Prolongations- bzw. Restriktionsschritt. Die Variablen können verschieden sein, z.B. die Prolongation bzgl. u oder e sein. Bei der Prolongation hat man Zugriff auf die Feingitterpunktliste, bei der Restriktion auf die Grobgitterpunktliste.

Zur Verknüpfung dieser Strukturen: Die einzelne Klasse von *Prolongation* und *Restriktion* hat immer die benötigten Informationen in Bezug auf das gröbere oder feinere Gitter, also für zwei Ebenen. Beide Klassen sind in *ZyklusRechnen* als eindimensionales Feld deklariert, d.h. als *Prolongation* * und *Restriktion* *, um diese Operationen für alle Ebenenpaare (grobes und feines Gitter) bereitzustellen.

Beendet wird dieser Abschnitt mit der Randinterpolation, bei der Grobgitterwerte übermittelt und dann interpoliert werden an echten Rändern im adaptiven Bereich. Zu den Methoden hier vgl. Abschnitt 4.3.9. Die Randinterpolation ist deklariert als eindimensionales Feld in *ZyklusRechnen* - mit genau denselben Ebenenpaaren, wie bei der *Prolongation* und *Restriktion*.

Zusammenfassend zu den Verknüpfungen der sequentiellen Funktionen

Zur Klasse Rechnen vergleiche man Abbildung 3.5.
Zu den Klassen Prolongation und Restriktion und Randinterpolation vergleiche man die Abbildung 3.6.

3.5.2 Die parallelen Funktionen für das Rechnen

Es sind dieselben Funktionen wie die sequentiellen Funktionen in Abschnitt 3.5.1, also glätten/lösen, Prolongation, Restriktion, Randinterpolation und Residuum berechnen.

Diese Funktionen arbeiten aber parallel, wobei sie nur den Parameter der Gitterebene übergeben bekommen. Dabei werden möglichst die Kommunikationen für die Randübertragungen durchgeführt, wie in Abschnitt 2.3.1 erläutert, wenn währenddessen die Hauptlast des Innenbereiches (nicht Rand) berechnet wird. Die Methoden werden in Abschnitt 4.3 vorgestellt. Diese Funktionen werden in der Klasse *ZyklusRechnen* bereitgestellt. Sie existieren mehrfach, da sie in verschiedenen Programmbereichen durchgeführt werden, und zwar für den adaptiven und nicht adaptiven Bereich (Index: unten und oben), auch rein sequentiell bei einer sogenannten cap (vgl. Abschnitt 4.5), und im nicht adaptiven Bereich als FAS oder als normales Mehrgitterverfahren. Das sind die *MLAT-Funktionen* in allen Variationen.

3.5.3 Die *MLAT-Funktionen*

Bei den *MLAT-Funktionen* handelt es sich um Funktionen, die verschiedene andere Rechenfunktionen aufrufen. Sie greifen z.B. auf die zu Beginn von Abschnitt 3.5.2 erläuterten Funktionen zu, und es gibt die folgenden 5 davon.
Für das Rechnen vom feinsten bis zum gröbsten Gitter hat man *MLAT_1*, dann für das Lösen auf dem gröbsten Gitter *MLAT_2*, dann für die Operationen vom groben Gitter zum feinen Gitter, und diese bis zum Grundgitter *MLAT_3*, und *MLAT_4* für die Nachglättung. Vom Grundgitter bis zum feinsten Gitter geht es weiter mit *MLAT_3*, dann *MLAT_RI* für die Randinterpolation und *MLAT_4*.
Es gibt verschiedene Typen von *MLAT-Funktionen*: Man unterscheidet für diese Funktionen, ob sie im Grobgitterbereich, im adaptiven Bereich, oder für die cap, rein sequentiell arbeiten.

3.5.4 Die Verknüpfungen der Klassen *Rechnen*, *Prolongation*, *Restriktion* und *Randinterpolation*

Die Verknüpfungen sind aufgezeigt: Die Klasse *Rechnen* liegt in Bezug auf Ableitungen zwischen den Klassen *Gitter* und *ParallelesRechnen*, vgl. Abbildung 3.5. *Randinterpolation* und *Prolongation* und *Restriktion* sind als Felder in *ZyklusRechnen* definiert, vgl. Abbildung 3.6. Die Klasse *ZyklusRechnen* leitet nach *Programm* ab, von wo aus auf die Funktionen aus der Überschrift über die *MLAT-Funktionen* zugegriffen wird.

3.6 Klassen für Änderungen des Multi-Gitters

Es kommen hier 3 Strukturen infrage, die für die Änderungen des Gitters nötig sind:

1. Das Bestimmen der Eigenschaftspunkte in der Klasse *EigenschaftsPunkte*, vgl. Abschnitt 3.6.1. Hier wird festgelegt, auf welchen Gitterpunkten verfeinert wird. Aufgrund der Überdeckung dieser Punkte mit 5 x 5 Gittern auf dem feinen Gitter muss das noch berechnet werden. Es wird festgehalten, wie die Eigenschaftspunkte in der aktuellen Iteration und der zurückliegenden Iteration waren. Wenn beides vorliegt, hat man eine Grundlage, um die Gitter nicht neu berechnen zu müssen, sondern nur die **Gitteränderungen** vorzunehmen.

2. Die Änderung in der Struktur *ParalleleAdaptiveVerfeinerung*, vgl. Kapitel 5. An dieser Stelle wird nur die Struktur *EP-SNB* diskutiert, vgl. Abschnitt 3.6.2, die

zu jedem Punkt durch jeweils eine Zahl angibt, in welcher der 8 Himmelsrichtungen Eigenschaftspunkte liegen. Das ist eine Struktur, die der beschleunigten Erfassung der Verfeinerungssituation dient, und die nicht neu berechnet, sondern aktualisiert wird. Sie greift auf die Klasse *SNB-Update* zurück, vgl. die Verknüpfungen in Abbildung 3.9. Alles weitere zu dieser Klasse und den anderen Teilen der *ParallelenAdaptivenVerfeinerung* erfolgt in Kapitel 5.

3. Der Lastausgleich wird durchgeführt. An dieser Stelle geht es insbesondere um die Struktur *Migration*, vgl. Kapitel 6. Davon wird an dieser Stelle die Auswahl unter den Gitterpunkten motiviert, was in der Klasse *'Strategie'* geschieht. Das geht unabhängig von den anderen Bestandteilen der *Migration*, und macht klar, wie der Lastausgleich gesteuert wird. Daher wird sie in diesem Kapitel behandelt, und zwar in Abschnitt 3.6.3.

Die Algorithmen zu den 3 Klassen *EigenschaftsPunkte bestimmen*, *EP-SNB-Update* und *Strategie*, werden im Folgenden vorgestellt und in Kapitel 4 dargestellt.

3.6.1 Eigenschaftspunkte bestimmen

Zu erwähnen ist zuerst, dass die Bestimmung der Eigenschaftspunkte nach 2 verschiedenen Methoden vor sich geht: Der eine Weg ist der, die Verfeinerungsgebiete entsprechend einer Simulation zu bewegen, der andere, aufgrund der berechneten Werte die Verfeinerungsentscheidungen zu treffen.

Bei der ersten Methode, der Simulation, kommt es zu zwei Aufrufen: Einmal in *Programm* zur Bildung der Parameter, die das Verfeinerungsgebiet bestimmen, wie z.B. die Angabe nach nördlichster, südlichster, östlichster und westlichster Koordinate eines Rechtecks von Eigenschaftspunkten, die die Verfeinerungseigenschaft erfüllen. Das bedeutet dann, dass für diese Punkte mit 5 x 5 Gebieten verfeinert wird. Dann kommt es aufgrund dieser Rechtecke in der Klasse *Eigenschaftspunkte* zur Bildung von *EP-aktuell*.

Bei der zweiten Methode zur Bestimmung der Eigenschaftspunkte, wo diese Punkte aus den Werten der Variablen bestimmt werden, müssen via *ZyklusRechnen* und *Rechnen* entweder beim Verfeinerungskriterium der Mehrgittertheoretiker τ oder bei der Verfeinerung entsprechend dem Verfeinerungskriterium des Autors τ_1 und τ_2 berechnet werden, vgl. Abschnitt 4.4.1. In *EigenschaftsPunkte* wird dann in der entsprechenden Funktion die für die Eigenschaftspunkte zuständige Variable auf diesen Koordinaten zwecks Verfeinerung auf true gesetzt. Das geschieht, wo der Diskretisierungsfehler τ oder die gewichtete Summe der Diskretisierungsfehler τ_1 und τ_2 einen Schwellwert übersteigt.

In beiden Fällen merkt man sich die im letzten Zeitschritt erfasste Eigenschaftspunktmarkierung $EP_{vergangen}$, damit man sich bei der Berechnung an den Änderungen orientiert, und auch nur Änderungen des Verfeinerungsgitters, und keine komplette Neuberechnung vornimmt.

Bemerkung: Das Verfeinerungsgebiet entspricht einer 5 x 5 Feingitterpunktüberdeckung

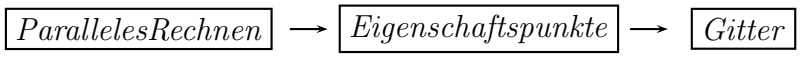


Abbildung 3.7: Die Verknüpfugen der *Eigenschaftspunkteklasse*

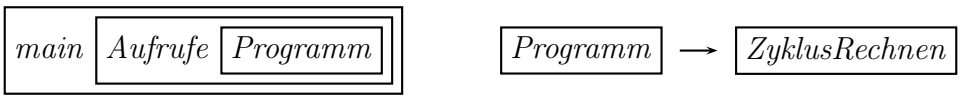


Abbildung 3.8: Die Klasse *Programm*

in Bezug auf die Grobgittereigenschaftspunkte.

3.6.2 *EP-SNB* aktualisieren

Man hat hier die Klasse *SNB-Update*, die nach *EP-SNB-Update* ableitet, vgl. Abbildung 3.9. Die Funktion *tun* der Klasse *EP-SNB-Update* basiert auf den Änderungen von EP, EP^- und EP^+, vgl. Abschnitt 5.4.5, und führt die Aktualisierung der Struktur *EP-SNB* durch. Diese erfasst in einer Zahl die Eigenschaftspunkte in allen 8 Himmelsrichtungen. Die Funktionen *hinzu* und *hinweg* von *SNB-Update* zur Änderung von *EP-SNB* werden dabei benutzt.

3.6.3 Strategie

Der Transfer der Gitterpunkte geschieht wie folgt: Ein bzgl. der Lastkerntopologie östlich liegender Kern bekommt die östlichsten eingefügten Gitterpunkte bei der Punktauswahl. Das geschieht dann analog bei den anderen Himmelsrichtungen. Die gesuchten Punkte werden dann automatisch gelöscht, damit die Datenstruktur *Strategie* immer aktuell gehalten werden kann. Ansonsten gibt es nur das Einfügen von Gitterpunkten. Das Suchen und Entfernen sowie das Einfügen definieren die Schnittstelle der Klasse *Strategie*. Für die Realisation, auch insbesondere der Änderung der hier gespeicherten Punktemenge, vgl. den Abschnitt 6.7.

3.6.4 Weitere Verknüpfungen

Verknüpfungen zur Klasse *EigenschaftsPunkte*

Die Klasse *EigenschaftsPunkte* wird von der Klasse *Gitter* abgeleitet und leitet selber ab nach der Klasse *ParallelesRechnen*, vgl. Abbildung 3.7.

Nun noch die Positionierung der zusätzlichen Programmteile in der Klasse *Programm*

Damit der Fall der Simulation möglich wird, ist die die bewegenden Gitter definierende Funktion in der Klasse *Programm* geschrieben. Das ist die alles umfassende Klasse, bis auf die die Programmparameter festlegenden und die Klasse *Programm* aufrufende Klasse *Aufrufe*, und die diese Klasse definierende Funktion *main*, vgl. Abbildung 3.8.

Die Berechnung der *MLAT-Funktionen MLAT_1-4* und *MLAT_RI* erfolgt in *Zyklus-Rechnen*, *Rechnen*, *Prolongation*, *Restriktion* und *Randinterpolation*, vgl. die Abbildung

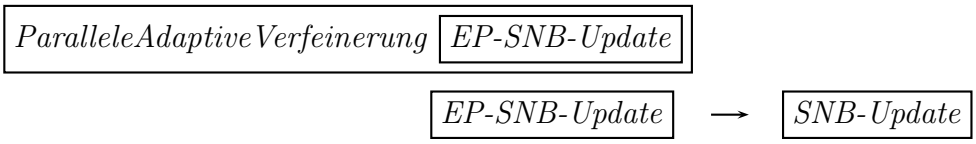


Abbildung 3.9: Die Verknüpfungen von *EP-SNB-Update*

Abbildung 3.10: Die Verknüpfungen der Klasse *Strategie*

3.4 für *ZyklusRechnen*, Abbildung 3.5 für die Klasse *Rechnen* und die Abbildung 3.6 für die Klassen *Prolongation*, *Restriktion* und *Randinterpolation*.

Verknüpfungen der Klasse *EP-SNB-Update*

Sie wird abgeleitet von der Klasse *SNB-Update*. Sie wird einmal definiert in der Klasse *ParalleleAdaptiveVerfeinerung*, welche immer in Bezug auf 2 Gitterebenen definiert ist (grob und fein). Die Klasse *EP-SNB-Update* ist definiert für das grobe Gitter der Klasse *ParalleleAdaptiveVerfeinerung*, vgl. Abbildung 3.9.

Die weiteren Klassen der Klasse *ParalleleAdaptiveVerfeinerung*, wie P_k^+, P_k^-, EP^+ und EP^- werden in Kapitel 5 behandelt.

Verknüpfungen der Klasse *Strategie*

Die Klasse *Strategie* leitet ab nach der Klasse *ExtremeVektoren*, die dafür sorgt, dass nicht der einzelne Gitterpunkt, z.B. der östlichste Gitterpunkt, entfernt wird, sondern gleich ganze Vektoren entfernt werden. Diese Klasse wird dann einfach oder doppelt bei verschiedenen Punktemengen in der Klasse *Migration* als eindimensionales Feld definiert. Deshalb hat man auf die Klasse *ExtremeVektoren* auf jeder Gitterebene, auf der Punkte ausgewählt werden sollen, Zugriff, siehe Abbildung 3.10.

3.7 Zusammenspiel auf der Top Programmierebene

3.7.1 Das Programm auf der Top Programmierebene ausführen

Da diverse Bestandteile des Programmes vorgestellt worden sind, können nun die verbindenden Teile dazu dargestellt werden. Es geht dabei aber nur um die Aufrufe der bereits besprochenen Programmteile auf einer Top Programmierebene, wo z.B. die Migration oder die parallele adaptive Verfeinerung als jeweils ein Arbeitsschritt dargestellt werden.

3.7.2 Die Funktion *tun* in der Klasse *Programm*

Eine sehr grobe Darstellung der Hauptfunktion *'tun'*

Die Funktion *tun* besteht aus vier Schritten:

1. Die Initialisierung. Hier werden die Strukturen angelegt, der Speicher bereitgestellt, die Werte werden bestimmt.

2. Die Iterationen. Für eine Anzahl von Iterationen wird gerechnet, wobei fast alle Klassen zum Einsatz kommen. Dieser Teil wird unten weiter ausgeführt, vgl. Schritt 2 der groben Darstellung.

3. Die Ausgabe von verschiedenen Laufzeiten zu verschiedenen Teilen des Programms.

4. Die Finalisierung: Die Datenstrukturen werden abgebaut, der Speicher wird freigegeben. Damit wird das Programm beendet.

Kommentar zu Schritt 2 der Funktion *tun*: Die Iterationen und das Rechnen dabei

a) Grobgitterzyklus: Von der Grundebene h_{min} werden Vergröberungsschritte gemacht bis zur Ebene h_{gesamt}-1. Dort wird ein Schritt für die gröbste Gitterebene getan, und dann werden Verfeinerungsschritte gemacht, im gröberen Bereich, bis zur Grundgitterebene h_{min}. Eine Erläuterung dazu folgt gleich.

b) Die Schritte im adaptiven Bereich von der Ebene h_{min} bis zur Ebene 0, der feinsten Ebene machen.
Dabei erfolgt am Ende der Behandlung einer Ebene ein Update der Kerntopologie und des Randes für die geänderte Ebene, bzw. für alle Ebenen, wenn das Gitter der Ebene 0 geändert worden ist.

c) Es werden Grobgitterschritte im adaptiven Bereich von der Ebene 0 bis zur Grundgitterebene h_{min} durchgeführt.

Kommentar zu Schritt 2a der Funktion *tun*: Der Grobgitterzyklus

Es gibt mehrere Typen von Grobgitterzyklen. Zuerst ist hier zu erwähnen, dass es davon mehrere Grobgitterzyklen gibt, denn:

i Es gibt die cap oder es gibt sie nicht. Sie wird in Abschnitt 4.5 vorgestellt. Die Befehle sammeln und zerstreuen werden dort auch erläutert.

ii Man wendet das normale Mehrgitterschema an oder das FAS.

Je nachdem sind die verschiedenen MLAT - Funktionen aufzurufen:

i *MLAT*_sequentiell*

ii *MLAT*_oben* im Falle des FAS

iii *MLAT*_oben_nicht-adaptiv* im Falle des normalen Mehrgitterschemas.

Dabei steht das '*' für einen *MLAT*-Index. Die Indizes lauten 1 bis 4 und RI.

Bemerkung: Es werden nun die verschiedenen Grobgitterzyklen für den V-Zyklus erklärt. Der V-Zyklus ist dadurch beschrieben, dass man die Ebenen der Reihe nach von der gröbsten bis zur feinsten Ebene durchgeht, und der Reihe nach wieder zurück.

Realisierungen des Grobgitterzyklus

Vorbemerkung 1: Die Funktionen mit dem Suffix 'oben' sind immer parallele Funktionen.

Vorbemerkung 2: Zum Begriff der cap vergleiche Abschnitt 4.5.

Algorithmus 3.1: Der Grobgitterzyklus mit FAS und ohne cap für den V-Zyklus

1. **for** h = h_min (Grundgitterebene) **to** h_gesamt - 1 (gröbste Gitterebene)
2. *MLAT1_oben*(h)
3. *MLAT2_oben*(h = gröbste Ebene)
4. **end for**
5. **for** h = h_gesamt-1 **to** h_min **step** -1
6. *MLAT3_oben*(h)
7. *MLAT4_oben*(h)
8. **end for**

Algorithmus 3.2: Der Grobgitterzyklus mit FAS und mit cap für den V-Zyklus

1. **for** h = h_min **to** h_cap - 1
2. *MLAT1_oben*(h)
3. *MLAT1a_oben*(h_cap)
4. **end for**
5. Sammle die Punkte auf der Ebene h_cap
6. *MLAT1b_sequentiell*(h_cap)
7. **for** h = h_cap + 1 **to** h_gesamt - 1
8. *MLAT1_sequentiell*(h)
9. *MLAT2_sequentiell*(h = gröbste Ebene)
10. **end for**

11. **for** h = h_gesamt - 1 **to** h_cap + 1 **step** -1
12. *MLAT3_sequentiell*(h)
13. *MLAT4_sequentiell*(h)
14. **end for**
15. *MLAT3a_sequentiell*(h_cap)
16. Zerstreue die Punkte auf der Ebene h_cap
17. *MLAT3b_oben*(h_cap)
18. **for** h = h_cap - 1 **to** h_min **step** -1
19. *MLAT3_oben*(h)
20. *MLAT4_oben*(h)
21. **end for**

Bemerkung 1: *MLAT1a* oder *MLAT3a* reichen immer so weit für eine Ebene, bis man zu der Variablen kommt, wo die Werte gestreut oder gesammelt werden.

Bemerkung 2: Beim Sammeln werden die Koordinaten auch gespeichert, sodass sie beim Zerstreuen nicht mehr übertragen werden müssen.

Bemerkung 3: Beim normalen Mehrgitterschema werden die Befehle mit dem Suffix 'oben' durch das Suffix 'oben_nicht-adaptiv' ersetzt. In dem Fall wird der Grobgitterzyklus mit Mehrgitterschema und mit cap für den V-Zyklus realisiert. Dieser Fall ist aber etwas anders programmiert, als es im Algorithmus 3.2 dargestellt wird.

Bemerkung 4: Die Algorithmen beschreiben einen V-Zyklus. Die Grobgitterzyklen haben allerdings ein frei definierbares Schema, das aber im Sinne dieser Algorithmen funktioniert. Diese Erweiterung des V-Zyklus wurde programmiert.

Vorbemerkung zu den Schritten im adaptiven Bereich, d.h. vom Grundgitter (keine Verfeinerung, keine Vergröberung) bis zum feinsten Gitter

Es gibt einen großen Unterschied zwischen den Schritten vom Grundgitter hin zum feinsten Gitter, und denen vom feinsten Gitter wieder hin zum Grundgitter. Bei den Schritten hin zum feinsten Gitter muss das Gitter umgebaut werden. Bei den sich anschließenden Schritten hin zum Grundgitter (0,1,..., h_{min}) bleibt das Gitter so wie es ist.

Kommentar zu Schritt 2b der Funktion *tun*: Die adaptiven Schritte bis zu Ebene 0

Hier erfolgt ein Gitterumbau. Wir durchlaufen eine Schleife k = h_{min} bis 1 (Step -1). Für jede Ebene geschieht das Folgende, wobei es zwei Möglichkeiten bzw. Varianten gibt:

1. Berechne auf der Ebene k eine neue Verfeinerungspunktmenge EP_aktuell - mit Update von EP_vergangen. Dazu wird erst EP_vergangen gelöscht, dann werden die Zeiger auf EP_vergangen und EP_aktuell getauscht, und am Ende wird EP_aktuell berechnet.
2. Mache parallele adaptive Verfeinerung von k auf k-1.
3. Randupdate(k-1), Update Kerntopologie(k-1)
4. Lastausgleich(k)
5. **if** k>1 **then** Mache Randupdate(k-1) und Kerntopologieupdate für alle Ebenen
6. **if** k = 1 **then** Mache für alle Gitterebenen h:
7. Randupdate(h), Update Kerntopologie(h)
8. Führe hier aus *MLAT_3*, *MLAT_RI*, *MLAT_4*

Bemerkung 1: Die Mehrgitteroperationen erfolgen sämtlich nach dem Lastausgleich. Es muss aber - auch dafür - kommuniziert werden, welche Gitterpunkte neu entstanden sind. Das wird für die Prolongation gebraucht, weil die für neue Punkte anders abläuft.

Bemerkung 2: Während des Lastausgleiches erfolgt ein Kerntopologie-Update. Nach dem Lastausgleich erfolgt im Falle der untersten Verfeinerungsebene das gemeinsame Randupdate und Kerntopologieupdate für alle Ebenen. Auf höheren Verfeinerungsebenen braucht hingegen der Randupdate nur für die geänderte Gitterebene erfolgen, weil man sie auf den anderen Ebenen noch nicht braucht. Sie werden ja nur für die *MLAT-Funktionen* mit dem Suffix 3, 4 und RI auf der neu entstandenen Ebene benötigt.

Bemerkung 3: Da der Lastausgleich auf allen Ebenen erfolgt, muss, damit die Zwischenkorrektur aus Abschnitt 6.8.9 korrekt arbeitet, das Farbenfeld für jede Ebene richtig sein. Dass ggf. nach einem adaptiven Schritt noch 'Gitter übersteht', wenn entfernte Punkte einer Gitterebene noch zu keiner Entfernung der Punkte darunter geführt haben, vgl. Abschnitt 5.9, ändert nichts an dieser Korrektheit auf jeder Ebene.

Zu Schritt c zum 2.ten Schritt der Funktion *tun*: Das sind im adaptiven Bereich die Vergröberungsschritte:

Im Wesentlichen wird auf den Ebenen k = 0 bis h_{min} - 1 (Ebene Grundgitter - 1) MLAT_1_unten (k) ausgeführt. Damit gelangt man von der Ebene 0 aufs Grundgitter.

3.7.3 Die Verknüpfungen oberhalb der Klasse *Programm*

Nachdem die zentrale Funktion *tun* der fast alles zusammenfassenden Klasse *Programm* kurz erläutert wurde, bleiben von den Top-Datenstrukturen nur noch die oberhalb der Klasse Programm vorhandenen Programmteile zu diskutieren.
Es gibt folgende Datenstrukturen in diesem Zusammenspiel: Die Funktion *main*, die Klasse *Aufrufe*, und die Klasse *Programm*. Dabei deklariert die Funktion *main* die Klasse *Aufrufe*, und die Klasse *Aufrufe* deklariert die Klasse *Programm*, vgl. Abbildung 3.1.

Da alle Aufgaben in der Klasse *Programm* abgearbeitet werden, dienen der Programmteil *main* und die Klasse *Aufrufe* der Bereitstellung der Parameter für die Klasse *Programm*:
Ein Teil der Parameter kann in den Argumenten beim Aufruf bestimmt werden. Die Funktion *'main'* ist dafür zuständig, und schreibt die Parameter in Variablen der Klasse *Aufrufe*. Weitere Parameter werden in der Klasse *Aufrufe* festgelegt. Dann schreibt die Klasse *Aufrufe* einen Teil dieser Parameter in die Klasse *Programm*, und der andere Teil der Parameter wird beim Aufruf der Funktion *Programm::tun* übergeben.

3.7.4 Die Verknüpfungen der Top-Programmebene

Nach den obigen Darstellungen der Top-Programmebene werden hier noch einmal wichtige Verknüpfungen dazu präsentiert.

1. Top-Ebene: *main* deklariert *Aufrufe*, *Aufrufe* deklariert *Programm*, vgl. Abbildung 3.1.

2. Zweite Ebene: *ZyklusRechnen* leitet nach *Programm* ab und deklariert ein eindimensionales Feld von *ParallelesRechnen*. Ferner wird ein Feld von *adaptiveVerfeinerung* und die *Migration* definiert, vgl. Abbildung 3.1.

3. *ParallelesRechnen*: In der Klasse *Randstruktur* werden *RandstrukturKommunikation* und *RandWerteKommunikation* deklariert. Dann wird nach *Gitter* abgeleitet. Über *EigenschaftspunkteBestimmen*, *RandStrukturBilden* und *Rechnen*, die nebeneinander stehen (virtual public), wird das dann nach *parallelesRechnen* abgeleitet, vgl. Abbildung 3.1.

Kapitel 4

Die kleineren Programmteile

4.1 Über den Inhalt von Kapitel 4

In diesem Kapitel werden die Methoden zu den Bestandteilen, wie sie in Kapitel 3 diskutiert wurden, dargestellt. Es geht um die Programmierung, also um Klassen, Funktionen und Algorithmen. Es wird möglichst die Reihenfolge von Kapitel 3 eingehalten, damit der Leser leicht die Methoden zu den Darstellungen aus Kapitel 3 nachschlagen kann. Ausnahmen: Die *cap*, die in Kapitel 3 nicht dargestellt worden ist, weil sie sich über mehrere Klassen 'erstreckt'. Was aber in Abschnitt 3.7.2 gemacht worden ist bei der Darstellung des Grobgitterzyklus, ist die Anwendung der *cap*-Programmteile. Offen ist noch die Darstellung der Prolongation, vgl. Abschnitt 4.3.7.

Die größeren Programmteile werden in Kapitel 5 - die parallele adaptive Verfeinerung - und Kapitel 6 - der Lastausgleich - behandelt.

4.2 Paralleles dynamisch änderbares Multi-Gitter

Dieser Abschnitt trägt denselben Namen wie Abschnitt 3.4. Der Unterschied besteht darin, dass dort die Bestandteile des Programms dargestellt wurden. Hier werden die Algorithmen dargestellt.

4.2.1 Die Gitterpunktlisten

Hier werden beide Gitterpunktlisten vorgestellt: Die *Feldliste* und die *Punktfeldliste*. In beiden Fällen hat man zwei Datenstrukturen: Einmal die Liste selber, die entweder wie bei der *Feldliste* eine doppelt verkettete Liste ist, oder wie bei der *Punktfeldliste* eine einfache Liste ist. Und dann ein zweidimensionales Feld, welches über Indizes auf Positionen in der Liste zeigt, vgl. Abbildung 4.1.

Das Einfügen in O(1) ist trivial: Der Eintrag in die jeweilige Liste funktioniert so: Es wird ein Speicherplatz für den neuen Eintrag bestimmt, und die Koordinaten werden dort eingetragen. Der Eintrag in das zweidimensionale Feld erfolgt so, dass das Feld unter den angegebenen Koordinaten auf den neuen Speicherplatz der Liste zeigt.
Das Entfernen geht in beiden Fällen in O(1), wenn man die Koordinaten hat. Und die

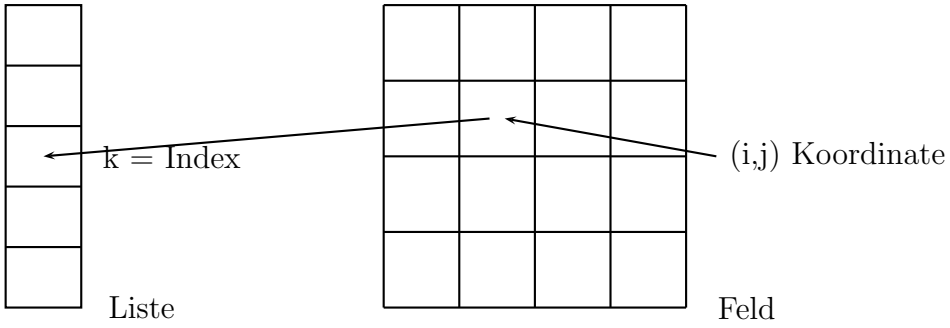


Abbildung 4.1: Die zweidimensionale Feldliste. Die Koordinate (i,j) liest im Feld ab, unter welchem Index k der Punkt in der Liste verzeichnet ist.

hat man, wenn Gitterpunkte entfernt werden. Erstens muss man die Stelle des Eintrages in der Liste kennen, um ein Element zu entfernen. Auf diese Stelle kommt man mit Hilfe des Feldes.
Zweitens muss man den Punkt aus der Liste löschen. Bei einer doppelt verketteten Liste werden wie bekannt die Indizes vor und nach dem Element verbunden. Bei einem einfachen Array wird das letzte Element auf das zu löschende Element kopiert und die Variable Listenlänge dekrementiert, vgl. die ausführliche Darstellung der Punktfeldliste in Abschnitt 5.4.3.

Bemerkung: Zur Cache-Optimierung wurde für die Klasse *Feldliste* eine Klassen-Variante *FeldlisteSortiert* entwickelt. Dafür werden die Punktmengen sortiert, indem die Punkte mit gemeinsamem f-Wert in einer *1-dimensionalenFeldliste* (vgl. Abschnitt 6.7.3) gespeichert werden - wie bei der Klasse *Strategie* in Abschnitt 6.7.4. Dabei muss nur überlegt werden, wie der Durchlauf (Anfangsposition, nächstes_Element, ist_fertig, lesen_f, lesen_g) und die Änderungen (einfügen/entfernen) erfolgen. Beim Durchlaufen der Punktmenge werden dann die *1-dimensionalenFeldlisten* entsprechend ihrem f-Wert nacheinander durchlaufen. Die Position in der Liste ist gekennzeichnet durch den f-Wert, die Position in der geordneten Gesamtpunktmenge entsprechend der Sortierung hier sowie der Nummer in der *1-dimensionalenFeldliste* zum f-Wert.

4.2.2 Die Algorithmen zum Farbenfeld

Diese Struktur wird in der Klasse *ParalleleAdaptiveVerfeinerung* geändert, wie auch in der Klasse *Lastausgleich*, vgl. die Kapitel 5 und 6. Das geschieht durch das Lesen des Feldes und durch das Schreiben in dieses Feld.

4.2.3 Die Datenstruktur *Selbstnachbarn*

Dieser Teil wird formal behandelt, weil diese Definitionen auch später gebraucht werden.

Definition 4.1: Die HR-Zahl
Den Himmelsrichtungen werden folgende Zahlen zugeordnet:
HR-Zahl(NORD) = 1, HR-Zahl(NORD-OST) = 2, HR-Zahl(OST) = 4, HR-Zahl(SÜD-OST) = 8, HR-Zahl(SÜD) = 16, HR-Zahl(SÜD-WEST) = 32, HR-Zahl(WEST) = 64,

HR-Zahl(NORD-WEST) = 128.

Definition 4.2: Koordinaten zu einer Himmelsrichtung, bezeichnet mit HR(f,g):
N: NORD(f,g) = (f-1,g)
NO: NORD-OST(f,g) = (f-1,g+1)
O: OST(f,g) = (f,g+1)
SO: SÜD-OST(f,g) = (f+1,g+1)
S: SÜD(f,g) = (f+1,g)
SW: SÜD-WEST(f,g) = (f+1,g-1)
W: WEST(f,g) = (f,g-1)
NW: NORD-WEST(f,g) = (f-1,g-1)

Definition 4.3: Inverse Himmelsrichtungen:
Zu HR lautet sie $\overline{HR}$:
$NORD^{-1}$ = SÜD
$(NORD - OST)^{-1}$ = SÜD-WEST
OST^{-1} = WEST
$(SÜD - OST)^{-1}$ = NORD-WEST
Mit $(HR^{-1})^{-1}$ = HR folgen die übrigen inversen Himmelsrichtungen.

Definition 4.4:
Die Menge MHR bezeichnet die Menge aller Himmelsrichtungen, also
MHR = {NORD, NORD-OST, OST, SÜD-OST, SÜD, SÜD-WEST, WEST, NORD-WEST}

Definition 4.5: Die Zahl einer Menge von Himmelsrichtungen
Für M $\subseteq$ MHR definieren wir HR-Zahl(M) = $\sum_{HR \in M}$ HR-ZAHL(HR)

Folgerung: Man erhält für M $\subseteq$ MHR eine Zahl zwischen 0 und 255. Diese Zahl zugeordnet zu einer Koordinate kann angeben, in welchen Himmelsrichtungen eine Eigenschaft erfüllt ist, z.B. ob dort Punkte auf dem eigenen Kern liegen, was bei der Struktur Selbstnachbarn gilt, oder ob dort das Verfeinerungskriterium erfüllt ist, vgl. die Klasse *EP-SNB-Update*.

Definition 4.6: Komplementärmenge zu einer Menge von Himmelsrichtungen
Es gibt noch eine komplementäre Menge zu einer Teilmenge M von MHR, definiert durch

$$\overline{M} = \bigcup_{HR \in MHR} \begin{cases} HR & \text{falls } HR \notin M \\ \emptyset & \text{sonst} \end{cases}$$

Beispiel: Die komplementäre Teilmenge von M={OST} besteht aus allen anderen Himmelsrichtungen.

Satz 4.1: Die komplementäre HR-Zahl:
HR-Zahl($\overline{M}$) = 255 - HR-Zahl(M) entsprechend den Definitionen 4.5 und 4.6.

Nun kann die Datenstruktur *Selbstnachbarn* definiert werden:

Definition 4.7: Das zweidimensionale Feld *Selbstnachbarn*.
Sei h die Gitterebene und G_h die eigene Gitterpunktmenge. Dann wird die Datenstruktur *Selbstnachbarn* der Gitterebene h definiert durch:
Selbstnachbarn(f,g) = {HR ∈ MHR | HR(f,g) ∈ G_h}
Damit definieren wir: *Zahl-Selbstnachbarn*(f,g) = HR-Zahl(*Selbstnachbarn*(f,g))

Mit diesen Definitionen können die Algorithmen zur Datenstruktur *Selbstnachbarn* formuliert werden. Man beachte, dass sie das Feld *Selbstnachbarn* nicht bestimmen, sondern ändern bzw. aktualisieren. Deswegen benötigt man zur Verwendung dieses Feldes neben diesen Algorithmen noch die Angabe von *Selbstnachbarn* in der Anfangsaufteilung.
Die Idee der folgenden Algorithmen besteht darin, dass durch das Einfügen und Entfernen von Punkten das *Selbstnachbarn-Feld* auf den umgebenden Punkten verändert wird. Wie das Feld entsprechend zu ändern ist, wird mit den folgenden Algorithmen klar:

Algorithmus 4.1: Ändern von *Zahl-Selbstnachbarn* aufgrund des Einfügens des Punktes (f,g):
for all HR ∈ MHR
    if (feld(HR(f,g))≥ 0) **then**
        Zahl-Selbstnachbarn(HR(f,g)) = *Zahl-Selbstnachbarn*(HR(f,g)) **OR**
        HR-Zahl(HR^{-1})
end for

Die Idee von Algorithmus 4.1: Die Nachbarpunkte HR(f,g) zu (f,g) zeigen ab jetzt den Punkt (f,g) als existent an. Denn auf der Position HR(f,g) steht in Richtung HR^{-1} der Punkt (f,g).

Algorithmus 4.2:
Ändern von *Zahl-Selbstnachbarn* aufgrund des Entfernens des Punktes (f,g):
for all HR ∈ MHR
    if (feld(HR(f,g))≥ 0) **then**
        Zahl-Selbstnachbarn(HR(f,g)) = *Zahl-Selbstnachbarn*(HR(f,g)) **AND**
        HR-Zahl($\overline{HR^{-1}}$)
end for

Die Idee von Algorithmus 4.2: Die Nachbarpunkte HR(f,g) zu (f,g) behalten nur in Richtungen, die von HR^{-1} verschieden sind, die Information bei, dass eigene Punkte dort existieren.

Bemerkung 1: Die **for all**-Befehle stehen nicht für übliche Schleifen, sondern sind Be-

fehl für Befehl fest implementiert.

Bemerkung 2: Die drei Felder, das Feld der *Feldliste*, das *Farbenfeld* sowie *Zahl-Selbstnachbarn* sind so definiert, dass ein Zugriff auf den um das Gitter umliegenden fiktiven Rand zulässig ist, also beispielsweise für die Koordinate (-1,-1). Damit sind sie für HR(f,g) mit (f,g) aus dem Gitter definiert.

4.2.4 Die Datenstruktur *DynamischeRandpunktliste*

Die *DynamischeRandpunktliste* ist eine *Feldliste*. Es wird im Folgenden gezeigt, wie sie geändert wird. Grundlage dafür ist die Überlegung, dass innere Punkte vorliegen, wenn *Zahl-Selbstnachbarn* dort den Wert 255 hat, also die Gitterpunkte in allen Himmelsrichtungen vorliegen.

Wenn beim Einfügen eines Punktes ein Wert von *Zahl-Selbstnachbarn* von < 255 auf 255 steigt, dann wird dieser Punkt, falls er sich in der Randliste befindet, aus dieser Liste gelöscht. Umgekehrt: Werden durch Entfernen von Punkten Werte von *Zahl-Selbstnachbarn* von anfangs 255 verringert, was bei Werten von 255 immer der Fall ist, werden diese Punkte, falls existent, in die Randliste eingefügt.

Die Algorithmen 4.1 und 4.2 seien entsprechend erweitert um den Update der Klasse *DynamischeRandpunktliste*.

Diese Liste ist sehr wichtig, weil damit der Rand in O(Anzahl von Randpunkten) vielen Schritten durchlaufen werden kann. Diese Liste benötigt man bei der Bestimmung der *Randstruktur*, vgl. den nächsten Abschnitt 4.2.5, sowie bei den Randdurchläufen der *ParallelenAdaptivenVerfeinerung* in Abschnitt 5.6.3.

4.2.5 Die Außen- und Innenrandlisten

Definition 4.8: Ein **echter Randpunkt** ist ein Gitterpunkt (f,g), für den gilt: Es gibt einen benachbarten Gitterpunkt (f',g') mit fgff(f',g') < 0. Dort existiert kein Punkt auf keinem Kern.
Ferner liegt ein echter Randpunkt vor, wenn die Koordinaten des Gitterpunkts Extremwerte annehmen, wie im Fall von $g = 0$.

Hier ist die algorithmische Behandlung von Abschnitt 3.4.1. Die Mengen für den Innenrand und Außenrand werden definiert. Danach werden dann die Algorithmen zu deren Bestimmung entwickelt. Dabei müssen die Mengen auch noch eine Reihenfolgebedingung (s.u.) erfüllen, damit ein Randaustausch bzgl. dieser Listen möglich wird.

Definition 4.9: Der Innenrand *IR(farbe)*
Der gemeinsame Innenrand IR^{gem} ist die Menge aller Gitterpunkte mit *Selbstnachbarn* $\neq 255$.
Der Innenrand *IR(farbe)* besteht aus den eigenen Gitterpunkten, die zu dem Kern mit

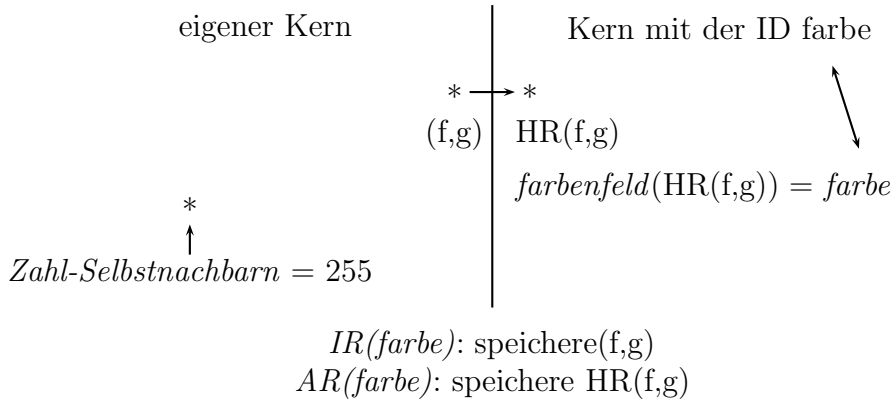


Abbildung 4.2: Die Mengen *IR(farbe)* und *AR(farbe)*

der Farbe bzw. Kern-ID farbe benachbart sind, vgl. Abbildung 4.2.

$farbe \neq Rang \Rightarrow IR(farbe) = \{(f,g) \in G \mid \exists_{HR \in MHR}\ farbenfeld(HR(f,g)) = farbe\}$

Es gilt: IR ist die Vereinigung aller *IR(farbe)* über alle möglichen Farben.

Definition 4.10: Der Außenrand *AR(farbe)*
Der gemeinsame Außenrand AR^{gem} besteht aus den Punkten der anderen Kerne, die direkt neben dem eigenen Gitter liegen. Das sind die Punkte HR(f,g) in Abbildung 4.2 mit irgendeinem Farbwert ≥ 0.
Der Außenrand *AR(farbe)* sind die Punkte auf dem gemeinsamen Außenrand mit der Farbe farbe. Das sind in Abbildung 4.2 die Punkte HR(f,g) mit $farbenfeld(HR(f,g)) = farbe$.

$farbe \neq Rang \Rightarrow$
$AR(farbe) = \{HR(f,g) \mid (f,g) \in G$ und $HR \in MHR$ und $farbenfeld(HR(f,g)) = farbe\}$

Es gilt: AR^{gem} ist die Vereinigung aller *AR(farbe)* über alle möglichen Farben.

Die folgenden Schritte bereiten die Algorithmen zur Bestimmung von *IR(farbe)* und *AR(farbe)* vor, die dabei so anzuordnen sind, dass die Liste von Koordinaten von *IR(P)* auf Kern Q dieselbe Reihenfolge haben wie die in *AR(Q)* auf Kern P.

Nun wird ein Testfeld vorgestellt, welches markiert wird. Es ist eine Hilfe bei der Bestimmung von *IR(farbe)* und *AR(farbe)* ohne mehrfach-Einträge.
Die folgenden Algorithmen 4.3 und 4.4 benötigen ein Testfeld. Dabei wird festgehalten, ob ein Punkt des Innenrandes bzw. Außenrandes schon eingetragen wurde. Trägt man in das Testfeld ein, sagt man, man **markiert** den Punkt mit Hilfe seiner Koordinaten für die *Farbe*. Zur Prüfung, ob auf dem Punkt (f,g) zu der Farbe farbe schon in die Liste IR(farbe) oder AR(farbe) eingetragen worden ist, verwendet man die Funktion **ist-markiert**. Deshalb bestehen die Argumente dieser Funktion aus Punktkoordinaten und einem Farbwert. Beim **Remarkieren** entfernt man eine Markierung. Beim **vollständigen-Remarkieren** werden alle Markierungen entfernt. Die Testfelder sind

definiert für jede Farbe außer der eigenen, und sie existieren aus folgenden Gründen:

1. Um keine doppelten Einträge zu bekommen
2. Beim Vektorisieren (s.u.) wird es gebraucht.

Abspeichern der Innenränder und Außenränder. Als Terminologie für die Algorithmen 4.3 und 4.4: Durch den Befehl *IR(farbe,f,g)* werden in die Kommunikationsliste des Innenrandes zum Kern *farbe* die Koordinaten (f,g) eingetragen. Das gilt analog für den Befehl *AR(farbe,f,g)* und die Kommunikationsliste des Außenrandes.

Es werden zwei unterschiedliche Methoden zur Bestimmung der geordeten Innen- und Außenränder vorgestellt. Sie tragen die Namen 'Randbestimmung durch Vektorisieren und Kommunikation' und 'Randbestimmung durch Sortieren'.

Randbestimmung durch Vektorisieren und Kommunikation. Mit Algorithmus 4.3 werden die Innenränder IR(farbe) bestimmt. Hier basiert die Aufgabe, eine Ordnung herzustellen, auf der Idee, die Innenränder zu verschicken. Durch den Algorithmus 4.5 wird es möglich, dass diese als Vektoren dargestellt werden. Das hat den Zweck, dass die Kommunikation der Rand-Koordinaten kürzer wird und deswegen schneller geht.
In Algorithmus 4.6 wird dann die gesamte Randbestimmung durchgeführt.

Randbestimmung durch Sortieren. Im Gegensatz zur 'Randbestimmung durch Vektorisierung', wo der Innenrand verschickt wird, und dadurch zum Außenrand wird, müssen in diesem Fall Innenrand und Außenrand beide bestimmt werden. Das geschieht in Algorithmus 4.4. Hier besteht die Idee dazu, eine gemeinsame Reihenfolge herzustellen, darin, diese Mengen gemäß einer lexikographischen Ordnung zu sortieren.
Abgeschlossen wird dieser Abschnitt dann durch Algorithmus 4.7, bei dem alle Schritte hin zur geordneten Randbestimmung aufgerufen werden.

Algorithmus 4.3: ***Bestimmung Innenrand***

1. **for all** Punkte (f,g) des Gitterrandes
2. **switch**(255-*Zahl-Selbstnachbarn*(f,g))
3. **case** zahl: zahl = HR-Zahl($\{HR_1, \cdots, HR_k\}$)
4. Es werden nun für die HR_i die folgenden Schritte gemacht
5. **if**(*ist-markiert*(*farbe*(HR_i(f,g)),f,g)) = true **then**
6. *trage_ein_in_IR*(*farbe*(HR_i(f,g)),f,g)
7. *markiere*(*farbe*(HR_i(f,g)),f,g)
8. **end if**
9. **end switch**

10. **end for**

Bemerkung: *trage_ein_in_IR*(*farbe*,f,g) meint, dass der Punkt (f,g) in die Innenrandliste bzgl. der Farbe *farbe* eingetragen wird. Analog ist AR(*farbe*,f,g) zu verstehen.

Vorbemerkung zu Algorithmus 4.4: Es reicht hier ein Testfeld pro anderem Kern aus, da die Innenränder zu den Außenrändern disjunkt sind.

Algorithmus 4.4: Bestimmung Innen- und Außenrand

1. **for all** Punkte (f,g) des Gitterrandes, vgl. 4.2.4
2. **switch**(255-*Zahl-Selbstnachbarn*(f,g))
3. **case** zahl: zahl = Zahl($\{HR_1, \cdots, HR_k\}$)
4. Es werden nun für die HR_i die folgenden Schritte gemacht
5. **if**(*ist-markiert*(*farbe*(HR_i(f,g)),f,g)) = true **then**
6. *trage_ein_in_IR*(*farbe*(HR_i(f,g)),f,g)
7. *markiere*(*farbe*(HR_i(f,g),f,g))
8. *trage_ein_in_AR*(*farbe*(HR_i(f,g)),HR_i(f,g))
9. markiere(*farbe*(HR_i(f,g)),f,g),HR_i(f,g))
10. **end if**
11. **end switch**
12. **end for**
13. *remarkiere* die Testfelder zu allen Kernen

Bemerkung 1: Im Algorithmus 4.3 wird am Ende nicht remarkiert, weil die Markierung noch in Algorithmus 4.5 für die Vektorisierung gebraucht und dort auch remarkiert wird. In Algorithmus 4.4 wird direkt remarkiert, da man keine Vektorisierung macht, und auch keine andere Verwendung für die markierten Testfelder hat.

Bemerkung 2: Zum Verständnis von Algorithmus 4.4: In Algorithmus 4.4 werden für die Befehle der Schritte 6 und 7, die sich auf den Innenrand beziehen, die Koordinaten (f,g) genommen, für die Befehle der Schritte 8 und 9, die sich auf den Außenrand beziehen, die Koordinaten HR_i(f,g).

Zur Speicherung der Koordinatenmengen in Algorithmus 4.5 benötigen wir folgende Definitionen:

Definition 4.11: Die vektorisierten Teile des Innenrandes
Nach der Vektorisierung bzgl. einer Farbe hat man drei Mengen:
VIR(*farbe*) = vertikale Vektoren für *IR(farbe)*

HIR(*farbe*) = horizontale Vektoren für *IR(farbe)*
PIR(*farbe*) = verbliebene Punkte aus *IR(farbe)*

Definition 4.12: Die vektorisierten Teile des Außenrandes
Nach der Vektorisierung bzgl. einer Farbe hat man drei Mengen:
VAR(*farbe*) = vertikale Vektoren für *AR(farbe)*
HAR(*farbe*) = horizontale Vektoren für *AR(farbe)*
PAR(*farbe*) = verbliebene Punkte aus *AR(farbe)*

Diese Punktemengen sind zwar abstrakt definiert, werden aber auch zum Abspeichern in den folgenden Algorithmen gebraucht.

Vorbemerkung 1: Zu Algorithmus 4.5: Hier wird auf die markierten Testfelder zurückgegriffen, weil sie angeben, zu welchen der Mengen *IR(farbe)* ein Innenrandpunkt gehört. Wir wollen ja *IR(farbe)* vektorisieren, und nicht den gemeinsamen Innenrand IR selber.

Vorbemerkung 2: Der Befehl continue springt zum Ende bzw. Anfang der Schleife.

Algorithmus 4.5: Vektorisiere Innenrand für eine Farbe, inklusive der Testfeldremarkierung

1. **for all** Punkte (f,g) $\in$ *IR(farbe)*
2. **if** *ist-markiert(farbe,f,g)* **then**
3. **continue**
4. **if** der Punkt (f,g) ist Nord-Süd verlängerbar auf markierten Punkten **then** (wird via *ist-markiert* erkannt)
5. Verlängere den Punkt (f,g) für markierte Punkte maximal nach Norden und Süden zu einem Vektor und füge ihn *ein* in VIR
und *remarkiere* dort das *testfeld*; **continue**
6. **if** der Punkt (f,g) ist Ost-West verlängerbar auf markierten Punkten **then** (wird via *ist-markiert* erkannt)
7. Verlängere den Punkt (f,g) für markierte Punkte maximal nach Osten und Westen zu einem Vektor und füge ihn *ein* in VIR
und *remarkiere* dort das *testfeld*; **continue**
8. *Füge* den Punkt (f,g) *hinzu* in PIR, und *remarkiere testfeld(farbe,f,g)*
9. **end for**

Mit diesen Vorüberlegungen ist es möglich, die erste Methode zur Randbstimmung zu formulieren:

Algorithmus 4.6: Randbestimmung mit Vektorisierung und Kommunikation

1. *Bestimme IR(farbe)* gemeinsam für alle Farben *farbe* ungleich Rang
2. **for all** Farben *farbe* ungleich Rang
3. *vektorisiere*(*farbe*) mit *speichern*
4. **end for**
5. **for all** Farben farbe mit *IR(farbe)*$\neq \emptyset$
6. *Übermitteln* von VIR, HIR und PIR für die Farbe *farbe*
7. **end for**
8. **for all** Farben *farbe* mit *IR(farbe)*$\neq \emptyset$
9. *Empfange und speichere* in VAR(*farbe*), HAR(*farbe*) und PAR(*farbe*)
10. **end for**

Nun zur zweiten Möglichkeit der Randstruktur-Bestimmung, bei der nach der Randmengen-Bestimmung ohne weitere Kommunikation die geordneten Punktemengen pro Farbe bestimmt werden.

Algorithmus 4.7: *Sortiere* die Ränder

1. *Bestimme IR(farbe)* und *AR(farbe)* gemeinsam für alle Farben *farbe* ungleich Rang, vgl. Algorithmus 4.4.
2. **for all** Farben *farbe* mit *IR(farbe)*$\neq \emptyset$
3. *Sortiere* alle Mengen *IR(farbe)* und *AR(farbe)* gemäß der lexikographischen Ordnung
4. *Speichern* der sortierten Mengen in PIR und PAR
5. **end for**

Bemerkung zur Korrektheit einer Koordinatensortierung: Diese Sortierung führt zu für den Randaustausch verwendbaren Punktmengen, weil offenbar für zwei Kerne der Farben q und r gilt: PIR(q) auf K_r ist gleich und in der selben Reihenfolge geordnet wie PAR(r) auf K_q.
Wenn das nicht gelten würde, würde Algorithmus 4.7 seinen Zweck verfehlen, einen Randaustausch nur mit der Übertragung des Wertebuffers - also ohne Übertragung der Koordinaten - zu ermöglichen.

4.2.6 Der Randaustausch bzgl. Variablen

Involvierte Klassen beim Randaustausch von Variablenwerten

Hier sind folgende Klassen involviert: Die Klasse *Randstruktur*, die verwendet wird für den Randaustausch. Dort wird, je nachdem, ob die Randlisten durch Vektorisierung oder Sortierung entstehen, auf die Klasse *RandstrukturKommunikation* oder die Klasse *RandwerteKommunikation* zugegriffen. Da aber die Option für beide Klassen bleiben soll, müssen die entsprechenden Funktionen (*packen** und *entpacken**) für beide Klassen programmiert werden.

Welche Funktionen dafür programmiert werden

Es geht um eine Kommunikation von Werten, also im Allgemeinen um Packen, Übermitteln bzw. Testen auf Vollendung der Kommunikation und dann das Entpacken. An dieser Stelle wird nur von *packen* und *entpacken* die Rede sein, weil die übrigen Testfunktionen für alle übertragenen Buffer gleich sind. Es handelt sich dabei um rein technische Programmteile wie der Aufruf von MPI-Funktionen.

Zu den übertragenen Variablen

Da man bei vielen Variablen einen Randaustausch vornehmen muss, hat man sehr viele Funktionen.

Man hat aber auch verschiedene Typen davon, was kommuniziert wird. Zum Beispiel wird einmal eine Variable übertragen, ein anderes Mal gleichzeitig zwei Variablen. Außerdem werden einmal Grobgitterwerte übertragen, ein anderes Mal Feingitterwerte. Dazu: Man hat die Ränder einer Ebene, aber die Variablen einer anderen Ebene. Die letzte Variante ist dann der red-black-Randaustausch.

Bemerkung zum Randaustausch von Grobgitterwerten bzw. Feingitterwerten: *IR(i)* ist prinzipiell gültig genau für die Ebene, auf der die Randmengen bestimmt werden. Werden Grobgitterwerte oder Feingitterwerte gepackt, kommen als Mengen *IR(i)* für die gröbere oder feinere Ebene in Betracht. Das ist aber nicht der Fall. Es sollen die Ränder einer Ebene und die Variablenwerte der dazu gröberen oder feineren Ebene genommen werden. Man vgl. z.B. die Verwendung der Grobgitterwerte bei der Prolongation in Abschnitt 4.3.7.

Die Verwendung von Substitutionen

Weil sehr viele *packen*- und *entpacken*-Funktionen programmiert werden müssen, wird das Zeichen * als Platzhalter für Suffixe von Funktionsnamen benutzt. Der Leser braucht dann die Suffixe dort nur einzusetzen, und kommt auf alle Funktionen.

Auflistung der *packen*- und *entpacken*-Funktionen

Zuerst werden die verschiedenen Typen von Funktionen fürs Packen und Entpacken aufgelistet. Diese stehen stellvertretend für eine größere Anzahl an Funktionen fürs Packen und Entpacken, weil der Algorithmus viele Variablen verwendet. Die Typen sind aber aufgelistet.

1. Packen einer Variable: *packen_u0*
2. Packen von zwei Variablen: *packen_u0_r0*
3. Packen von red-Werten: *packen_red_u0*
4. Packen von Feingitterwerten: *packen_fg_u0*
5. Packen von Grobgitterwerten: *packen_gg_u0*
6. Entpacken einer Variable: *entpacken_u0*
7. Entpacken von zwei Variablen: *entpacken_u0_r0*
8. Entpacken von red-Werten: *entpacken_red_u0*
9. Entpacken von Feingitterwerten: *entpacken_fg_u0*
10. Entpacken von Grobgitterwerten: *entpacken_gg_u0*

Die Funktionen zum Packen und Entpacken in der Klasse *Randstruktur*

Zuerst zwei Bezeichnungen: Die Klasse *RandStrukturKommunikation* trägt die Randstrukturen im Falle des Konzeptes 'Vektorisieren', die Klasse *RandWerteKommunikation* trägt die Randstrukturen im Falle des 'Sortierungs'-Konzeptes. Bei der *RandStrukturKommunikation* wird bzgl. Vektoren und Punkten ge- und entpackt, bei der *RandWerteKommunikation* geschieht dieses bzgl. Punktlisten.

Zuerst ist zu bemerken, dass die Klassen *RandStrukturKommunikation* und *RandWerteKommunikation* für alle Farben definiert werden. Deshalb stehen die Farbwerte als Indizes hinter diesen Klassen.

Der Algorithmus fürs Packen lautet also wie im Folgenden dargestellt. Der * hat die Bedeutung, dass statt *packen** eine der oben aufgeführten Funktionen fürs Packen beschrieben wird, vgl. die Punkte 1 bis 5 der Auflistung.

Algorithmus 4.8: *packen**

1. **if**(vektorisieren)
2. **for** i=0 **to** farbenanzahl - 1
3. **if**(i $\neq$ eigene-farbe und *IR(i)* $\neq \emptyset$) **then**
4. *randstrukturkommunikation*[i].*packen**
5. **end for**
6. **else**
7. **for** i=0 **to** farbenanzahl - 1
8. **if**(i $\neq$ eigene-farbe und *IR(i)* $\neq \emptyset$) **then**

9. *randwertekommunikation*[i].*packen**
10. **end for**
11. **end if**

Der Algorithmus fürs Entpacken ist analog zum Algorithmus für Packen geschrieben, mit dem Unterschied, dass statt *packen entpacken* Funktionen aufgerufen werden.

Die Funktionen *packen* und *entpacken* in den Klassen *RandStrukturKommunikation* und *RandWerteKommunikation* - abstrakt

Verwendet wird nun wiederum das Symbol *, einmal, wo ein Suffix der Funktion einzutragen ist, und einmal zur Verwendung für die Fallunterscheidung in den Abbildungen 4.3 und 4.4 zu den Algorithmen.
In der Programmierung selber gibt es die Fallunterscheidungen aber nicht. Dort sind alle Fälle programmiert.
Die Mengen IR, VIR, HIR usw. erscheinen ohne Farbwert, weil die Klassen *RandstrukturKommunikation* und *RandwerteKommunikation* für jede Farbe *farbe* $\neq$ Rang angeleget sind.

Algorithmus 4.9: *packen** in *RandStrukturKommunikation*

1. k = 0, setze den Bufferindex
2. **for all** (f_1,f_2,g) $\in$ VIR
3. **for** f=f_1 **to** f_2
4. *packen**(f,g)
5. **end for**
6. **end for**
7. **for all** (f,g_1,g_2) $\in$ HIR
8. **for** g=g_1 **to** g_2
9. *packen**(f,g)
10. **end for**
11. **end for**
12. **for all** (f,g) $\in$ PIR
13. *packen**(f,g)
14. **end for**

buffer(k) = u0(f,g); k = k+1	für * = u0
buffer(k) = u0(f,g); k = k+1 buffer(k) = r0(f,g); k = k+1	für * = u0,r0
if(f=0(mod2) & g=0(mod2)) buffer(k) = gg_u0($\frac{f}{2}$,$\frac{g}{2}$); k = k+1	für * = gg_u0
buffer(k) = fg_u0(2f, 2g); k = k+1	für * = fg_u0
if((f+g)=0(mod2)) buffer(k) = u0(f,g); k = k+1	für * = red_u0

Abbildung 4.3: Die verschiedenen Weisen, in den Buffer zu packen, bei *packen**

Algorithmus 4.10: *packen in *RandWerteKommunikation***

1. k = 0, setze den Bufferindex
2. **for all** (f,g) ∈ IR
3. *packen**(f,g)
4. **end for**

Zur Formulierung von *packen*(f,g)* in den Algorithmen 4.9 und 4.10, vgl. die Abbildung 4.3. Es erfolgt eine kurze Erläuterung der Fälle:

1. * = u0: Hier wird u0 an der Stelle (f,g) in den Buffer geschrieben.
2. * = u0,r0: Hier wird erst u0 an der Stelle (f,g) in den Buffer geschrieben, danach r0.
3. * = gg_u0: Hier wird, falls ein Grobgitterwert an dieser Stelle zur Verfügung steht (f = 0 mod 2, g = 0 mod 2), dieser in den Buffer geschrieben.
4. * = fg_u0: Hier wird der Feingitterwert an der Stelle (2f,2g) in den Buffer geschrieben.
5. * = red_u0: Falls die Koordinate (f,g) eine Red - Koordinate ist, wird der Wert in den Buffer geschrieben.

Bemerkung: Die Variable gg_u0 wird in der Klasse *Programm* erzeugt, und zwar durch *paralleles_rechnen[i].gg_u0 = paralleles_rechnen[i+1].u0*. Die Variable *fg_u0* wird erzeugt in der Klasse *Programm*, und zwar durch *paralleles_rechnen[i].fg_u0 = paralleles_rechnen[i-1].u0*.

Der Algorithmus fürs Entpacken in der Klasse *RandStrukturKommunikation* ist ähnlich zu dem Algorithmus 4.9, nur mit der Funktion *entpacken** statt *packen**, den Mengen VAR statt VIR, HAR statt HIR und PAR statt PIR, sowie der Verwendung von Abbildung 4.4 statt der Abbildung 4.3.

Der Algorithmus fürs Entpacken in der Klasse *RandWerteKommunikation* ist ähnlich zu dem Algorithmus 4.10, nur mit der Funktion *entpacken** statt *packen**, den Mengen AR statt IR, HAR statt HIR und PAR statt PIR, sowie der Verwendung von Abbildung 4.4 statt der Abbildung 4.3.

u0(f,g)=buffer(k); k = k+1	für * = u0
u0(f,g)=buffer(k); k = k+1 r0(f,g)=buffer(k); k = k+1	für * = u0,r0
if(f=0(mod2) & g=0(mod2)) gg_u0($\frac{f}{2}$,$\frac{g}{2}$) = buffer(k); k = k+1	für * = gg_u0
fg_u0(2f, 2g) = buffer(k); k = k+1	für * = fg_u0
if((f+g)=0(mod2)) u0(f,g) = buffer(k); k = k+1	für * = red_u0

Abbildung 4.4: Die verschiedenen Weisen, aus dem Buffer zu entpacken, bei *entpacken**

4.2.7 Die 'kleine' Randstruktur

Im adaptiven Bereich gibt es folgende Beobachtung: Die Gitterpunkte über dem feinen Gitter belegen nicht das ganze Gebiet. Daher können für gewisse Funktionen, wie z.B. bei der Berechnung der rechten Seite, die Innen- und Außenrandmengen des gröberen Gitters eingeschränkt werden. Man verschickt zu diesem Zweck eine **Signatur**: Jeder Randpunkt in seiner Reihenfolge bekommt ein Bit zugeordnet, ob darunter ein Punkt existiert. Der daraus folgende Buffer mit einem Bit pro Gitterpunkt wird verschickt. Danach brauchen Sender und Empfänger nur ihre Listen durchgehen, und je nachdem, ob dann das Bit gesetzt ist oder nicht, wird der Randwert geschrieben oder nicht bzw. gelesen oder nicht.

4.2.8 Die Aktualisierung der Außen- und Innenrandliste

Diese geschieht durch Neuberechnung, vgl. Abschnitt 4.2.5.

4.2.9 Der Update der Kerntopologie

Man kann die Kerntopologie bestimmen, indem man prüft, ob zu einer Variablen die Innenrandliste leer ist. Auch bei der Vektorisierung liegt diese Information als Vorform der Vektorlisten vor in *IR(farbe)*, sodass für eine mögliche Kernverbindung eine Variable - nämlich die Länge von *IR(farbe)* - geprüft werden muss.

Bemerkung 1: Insgesamt müssen für die Kerntopologie für alle Ebenen und Farben diese Prüfungen erfolgen.

Bemerkung 2: Wenn man mithilfe der Kerntopologie in Bezug auf eine Farbe *farbe* für alle Ebenen prüft, ob eine Punktenachbarschaft existiert, erhält man die Multi-Level-Kerntopologie für diese Farbe.

Bemerkung 3: Zwischen mehreren Migrationen macht es Sinn, nur die Kerntopologie zu bestimmen, weil die Ränder nicht gebraucht werden. Mit der dynamischen Randliste geht das schnell, indem man diese durchläuft, danach *Zahl-Selbstnachbarn* mit einer switch-case-Anweisung überprüft, und dann für die nicht eigenen Felder testet, ob das Farbenfeld ≥ 0 ist. In dem Fall existiert die Verbindung zu dem Kern mit der ID des Farbenfeldes. Bei Bedarf können dabei für die Lastwertekorrektur sogar die echten Randpunkte markiert werden, wenn das Farbenfeld < 0 ist, vgl. auch die Lastwerte-

korrektur in Kapitel 6.

Wichtig ist hier je nach Position im Zyklus zu wissen, welche der beiden Kerntopologien man verwendet. Sie können je nachdem, ob sie aktualisiert wurden oder nicht, voneinander abweichen. Nach dem Gitterumbau bis zum nächsten Gitterumbau wird die durch Randlisten bestimmte Topologie verwendet, während beim Gitterumbau mit Ausnahme der Ebene, auf der gerechnet wird, die schneller bestimmbare Topologie verwendet wird.

4.3 Rechnen auf dem Multi-Gitter

Zu den Bedeutungen der Variablen u, e, r, Iu, usw. vergleiche man Abschnitt 2.2.

Der Unterschied zu Abschnitt 3.5, obwohl dieselbe Überschrift verwendet wird

Dort wurden Klassen und ihre Verknüpfungen dargestellt. Hier aber werden die konkreten Algorithmen ausgeführt.

Was hier dargestellt wird.

In diesem Abschnitt werden in den Teilen 4.3.1 bis 4.3.9 überwiegend die recht einfachen Algorithmen der *MLAT*-Funktionen beschrieben. Beschrieben werden die einzelnen Schritte der *MLAT*-Funktionen. Das sind sequentielle, parallele und adaptiv parallele Funktionen. Die *MLAT*-Funktionen selber werden hier nicht mehr besprochen, sondern nur die im Folgenden dargestellten Teilschritte der *MLAT*-Funktionen. Eine Ausnahme hierzu bildet die *MLAT-Randinterpolation*.

Ein Beispiel dafür, dass auf diese Weise, mit einfacheren Algorithmen, auch komplexere Formeln berechnet werden, ist die Berechnung der rechten Seite in Abschnitt 4.3.5. Dem geht z.B. die Berechnung der Restriktion in Abschnitt 4.3.4 voraus.

Der Aufbau der Abschnitte 4.3.1 bis 4.3.9.

Der Aufbau folgt in diesen Abschnitten immer in dieser Reihenfolge: 1. Zuerst erfolgt die sequentielle Darstellung in der Klasse *Rechnen*. 2. Dann wird als Vorbereitung zur parallelen Darstellung angegeben, welcher Randaustausch zum Einsatz kommt. Danach werden, in den Punkten 3 bis 5, die Teilfunktionen der Klasse *ZyklusRechnen* angegeben. 3. Da kommt zuerst der sequentielle Algorithmus der Klasse *ZyklusRechnen*, damit die '*cap*' realisiert werden kann. 4. Es wird dann die parallele nicht adaptive Darstellung angegeben. 5. Schließlich wird die adaptive Darstellung vorgenommen.

1. Die sequentielle Darstellung. Bei diesen Algorithmen wird, wenn sie in der Klasse *Rechnen* erfolgen, üblicherweise eine Punktemenge mit übergeben. Bei Restriktion und Prolongation wird hingegen mithilfe von Zeigern ein Zugriff auf die entsprechenden Mengen des Gitters erfolgen. Deshalb werden jeweils zwei Algorithmen geschrieben, getrennt nach der Berechnung im inneren Gebiet und auf dem Rand.

2. Die Auswahl des Randaustausches wird erläutert. Das basiert im Allgemeinen auf der Variablen, die die Quelle der Rechenoperation ist. Wird beispielsweise der Wert für einen red-Punkt berechnet, werden black-Punktwerte ausgetauscht, da diese Werte für die Berechnung gebraucht werden, und damit die **Quelle für die Berechnung** darstellen.

3. Die sequentiellen *ZyklusRechnen MLAT*-Teile. Nach dem Randaustausch wird die sequentielle *MLAT*-Funktion ausgeführt, die im Allgemeinen aus einem Aufruf der sequentiellen Funktion mit den Punktmengen für das Innere sowie für den Rand besteht. Sie trägt den Suffix **sequentiell**.

4. Die parallelen *ZyklusRechnen MLAT*-Teile. Nach der sequentiellen *MLAT*-Funktion wird die nicht adaptive *MLAT*-Funktion für den jeweiligen Rechenschritt dargestellt. Sie trägt den Suffix **oben**.

5. Die parallelen adaptiven *ZyklusRechnen MLAT*-Teile. Am Ende der jeweiligen Teilabschnitte werden die Abweichungen der MLAT-Schritte im adaptiven Fall gegenüber dem nicht adaptiven Fall vorgestellt, falls Abweichungen vorliegen. Sie trägt den Suffix **unten**.

Drei Bemerkungen zu den Punkten 1 bis 5

Bemerkung 1: Es gibt zwei Varianten das Innere des Gitters zu berechnen, während die Randkommunikation stattfindet, vgl. den Abschnitt 2.3.1. Hier wird die Variante gewählt (wegen der Fülle an Funktionen muss man wählen), wo zuerst die Randübermittlung gestartet wird, dann das Innere gerechnet wird, dann die Randübermittlung vollendet und am Ende dann der eigene Rand berechnet wird.

Bemerkung 2: Diese Fälle werden für die MLAT-Funktionen im Falle der Poisson-Gleichung angegeben. Sonst erfolgen ggf. Abweichungen.

Bemerkung 3: Die Struktur der 'cap' benötigt die sequentiellen Routinen, vgl. diesen Abschnitt.

4.3.1 Der Glätter oder Löser: A. Die Jacoby-Methode

Die Glätter bzw. Löser, die hier vorgestellt werden, stammen von der Jacoby- oder Gauss-Seidel-Methode. Bei anderen Mehrgitterverfahren müssen sie ggf. ersetzt werden durch entsprechende Funktionen.

Vorgestellt wird hier zuerst die Jacoby-Methode, und dann die Gauss-Seidel Methode. Gewählt wurde hier eine red-black Gauss-Seidel Methode.

1. Das sequentielle Rechnen in der Klasse *Rechnen* - der Jacoby-Glätter bzw. -Löser

Algorithmus 4.11: Jacoby-Glätter bzw. -Löser in der Klasse Rechnen
Input: Eine Liste von Gitterpunkten: L (x- und y-Koordinate, Länge der Liste); dabei gilt, dass echte Randpunkte in L nicht vorkommen
Output: Das aktualisierte Feld zu $u_{f,g}$

1. **for all** (f,g) $\in$ L
2. $u_{f,g} = \frac{1}{4}(h_x h_y \, rS_{f,g} + u_{f-1,g} + u_{f+1,g} + u_{f,g-1} + u_{f,g+1})$
3. **end for**

Die Variablen und Konstanten: u ist die Hauptvariable der diskretisierten Differentialgleichung, h_x ist der Gitterabstand in x-Richtung, h_y ist der Gitterabstand in y-Richtung, und rS die rechten Seite. Wie die rechte Seite berechnet wird, vgl. später in 4.3.5. Auf der feinsten Ebene ist rS = f, dem Funktionswert, und auf den anderen Ebenen wird rS berechnet bzw. übernimmt rS den Funktionswert f.

2. Die Wahl des Randaustausches

Die Angabe des Randaustausches ist deswegen wichtig, weil er die sequentiellen Routinen der Klasse ergänzt. Die Kombination aus sequentiellem Rechnen und dem Randaustausch ergibt dann das parallele Rechnen der Klasse *ZyklusRechnen* (nicht zu verwechseln mit der Klasse *ParallelesRechnen*).

Hier gilt: Die Wahl des Randaustausches ist *randaustausch_u0.*

3. Die Funktion Jacoby-Glättung bzw. -Lösung der *MLAT-sequentiell-Funktionen*

Algorithmus 4.12: Jacoby-Löser sequentiell in der Klasse *ZyklusRechnen*
Input: Gitterebene h
Output: Durchgeführte Berechnung von u.

1. *Glättung* (Alg. 4.11) auf Ebene h(für die Innenpunkte der Ebene h)
2. *Glättung* (Alg. 4.11) auf Ebene h(für die unechten Randpunkte der Ebene h)

4. Die Funktion Jacoby-Glättung bzw. -Lösung der *MLAT-parallel-nicht-adaptiv-Funktionen*

Algorithmus 4.13: Jacoby-Löser parallel und nicht adaptiv in der Klasse *ZyklusRechnen*
Input: Gitterebene h
Output: Durchgeführte Berechnung von u.

1. *Starte Randübermittlung* auf Ebene h
2. *Glättung* auf Ebene h(für die Innenpunkte der Ebene h)
3. *Beende Randübermittlung* auf Ebene h
4. *Glättung* auf Ebene h(für die unechten Randpunkte der Ebene h)

5. Die Funktion Jacoby-Glättung bzw. -Lösung der *MLAT-parallel-adaptiv-Funktionen*

Dazu kann man den Algorithmus 4.13 verwenden.

4.3.2 Der Glätter oder Löser: B. Die Gauss-Seidel Methode

1. Das sequentielle Rechnen in der Klasse *Rechnen* - der *red_Gauss-Seidel-Glätter* bzw. *-Löser*

Der black Gauss-Seidel-Glätter bzw. -Löser wird analog zum red_Gauss-Seidel-Glätter bzw. -Löser formuliert. Deswegen wird hier nur der red-Glättungsschritt aufgeführt:

Algorithmus 4.14: Der red_Gauss-Seidel-Glätter bzw. -Löser in der Klasse *Rechnen*
Input: Eine Liste von Gitterpunkten: L (x- und y-Koordinate, Länge der Liste), ohne echte Randpunkte
Output: Das aktualisierte Feld zu $u_{f,g}$

1. **for all** (f,g) $\in$ L
2. **if**(f+g=0(mod 2)) **then**
3. $u_{f,g} = \frac{1}{4}(h_x h_y \, rS_{f,g} + u_{f-1,g} + u_{f+1,g} + u_{f,g-1} + u_{f,g+1})$
4. **end for**

Zu den Variablen und Konstanten: Vergleiche die Kommentare zur Jacoby-Glättungsmethode

2. Die Wahl des Randaustausches

Hier gilt: Die Wahl des Randaustausches ist *randaustausch_u0_black*.

3. Die Funktion *red_Gauss-Seidel-Glättung* der *MLAT-sequentiell-Funktionen*

Algorithmus 4.15: Der *red_Gauss-Seidel-Glätter* bzw. *-Löser* sequentiell in der Klasse *ZyklusRechnen*
Input: Gitterebene h
Output: Durchgeführte Berechnung von u.

1. *red_Glättung* bzw. *red_Lösung* (Alg. 4.14) auf Ebene h(für die Innenpunkte der Ebene h)
2. *red_Glättung* bzw. *red_Lösung* (Alg. 4.14) auf Ebene h(für die unechten Randpunkte der Ebene h)

4. Die Funktion *red_Gauss-Seidel-Glättung* bzw. *-Lösung* der *MLAT-parallel-nicht-adaptiv-Funktionen*

Algorithmus 4.16: *red_Gauss-Seidel-Glätter* bzw. *-Löser* parallel - nicht adaptiv - in der Klasse *ZyklusRechnen*
Input: Gitterebene h
Output: Durchgeführte Berechnung von u.

1. *starte_black_Randübermittlung* auf Ebene h

2. *red_Glättung* bzw. -*Lösung* auf Ebene h(für die Innenpunkte der Ebene h)

3. *beende_black_Randübermittlung* auf Ebene h

4. *red_Glättung* bzw. -*Lösung* auf Ebene h(für die unechten Randpunkte der Ebene h)

Bemerkung: Man übermittelt die black Randpunkte, weil sie für die red Randpunkte Berechnung gebraucht werden.

5. Die Funktion *red_Gauss-Seidel-Glättung* bzw. -*Lösung* der *MLAT-parallel-adaptiv-Funktionen*

Dazu kann man den Algorithmus 4.16 verwenden.

4.3.3 Residuum berechnen

1. Das sequentielle Rechnen in der Klasse *Rechnen* - die Berechnung des Residuums

Algorithmus 4.17: *Residuum_berechnen* in der Klasse *Rechnen*
Input: Eine Liste von Gitterpunkten: L (x- und y-Koordinate, Länge der Liste), ohne echte Randpunkte
Output: Das aktualisierte Feld zu $r_{f,g}$

1. **for all** (f,g) $\in$ L

2. $r_{f,g} = rS_{f,g} - \frac{1}{h_x h_y} (4\ u_{f,g} - u_{f-1,g} - u_{f+1,g} - u_{f,g-1} - u_{f,g+1})$

3. **end for**

2. Die Wahl des Randaustausches

Zur Berechnung der Formeln wird nur der *randaustausch_u0* benötigt. Die Variable rS wird nämlich nur auf der Koordinate (f,g) benötigt und nicht daneben.

3. Die Funktion *Residuum_berechnen* der *MLAT-sequentiell-Funktionen*

Algorithmus 4.18: *Residuum_berechnen_sequentiell* in der Klasse *ZyklusRechnen*:
Input: Die Ebene h
Output: Der Wert r des Residuums ist aktualisiert.

1. *Residuum_berechnen* auf Ebene h(für die Innenpunkte der Ebene h)

2. *Residuum_berechnen* auf Ebene h(für die unechten Randpunkte der Ebene h)

4. Die Funktion *Residuum_berechnen* der *MLAT-parallel-nicht-adaptiv-Funktionen*

Algorithmus 4.19: *Residuum_berechnen_parallel_nicht_adaptiv* in der Klasse *Zyklus-Rechnen*
Input: Gitterebene h
Output: Durchgeführte Berechnung von u.

1. *randaustausch_u0* auf Ebene h starten
2. *Residuum_berechnen* auf Ebene h(für die Innenpunkte der Ebene h)
3. *randaustausch_u0* auf Ebene h vollenden
4. *Residuum_berechnen* auf Ebene h(für die unechten Randpunkte der Ebene h)

5. Die Funktion *Residuum berechnen* der *MLAT-parallel-adaptiv-Funktionen*

Dazu kann man den Algorithmus 4.19 verwenden.

4.3.4 Die (parallele) Restriktion

1. Die Berechnung der Restriktionen - nicht in der Klasse *Rechnen*, sondern in einer eigenen Klasse

Vorbemerkung 1: Hier wird die Berechnung der Restriktionen von u und r vorgestellt.

Vorbemerkung 2: Hier werden für die Algorithmen keine Mengen übergeben. Die Klasse Restriktion verfügt über die Mengen. Es gibt Zeiger auf die Grobgitterpunktemenge; auf die Punkteanzahl wird auch zugegriffen.

Vorbemerkung 3: Bei der Restriktion der Variablen r wird die Vorausberechnung nicht für die inneren Punkte der Grobgitterpunktmenge genommen, sondern die inneren Punkte der Feingitterpunktmenge. Deshalb wurden zwei Algorithmen für die Restriktion geschrieben: Einmal für die inneren Punkte auf dem feinen Gitter, und dann für die verbliebenen Punkte.

Algorithmus 4.20: *Restriktion_der_Variablen_u* berechnen in der Klasse *Restriktion* - eine Injektion
Input: Aufruf der Funktion *tun_u* in der Klasse *Restriktion* der Ebenen h und h+1.
Output: Das Feld zu $u_{f,g}$ wird aktualisiert.

1. **for all** (f,g) $\in G_{h+1}$
2. ff = 2 f und gg = 2 g
3. $u^{h+1}_{f,g} = u^{h}_{ff,gg}$
4. **end for**

Mit $u^h_{f,g}$ ist u an der Stelle (f,g) auf der Ebene h gemeint.

Zuerst werden die Formeln für die Restriktionsmethode **Full Weighting** dargestellt. Es gibt da 3 typische Fälle:

1. Für einen Punkt, der nicht am echten Rand liegt, lautet die Formel:
   $r_{f,g} = \frac{1}{16}$ (4 $r_{ff,gg}$ + 2 $r_{ff-1,gg}$ + 2 $r_{ff+1,gg}$ + 2 $r_{ff,gg-1}$ + 2 $r_{ff,gg+1}$ + $r_{ff-1,gg-1}$ + $r_{ff+1,gg-1}$ + $r_{ff-1,gg+1}$ + $r_{ff+1,gg+1}$)
2. Für einen Punkt, der an einem echten oberen Rand liegt, lautet sie:
   $r_{f,g} = \frac{1}{8}$ (2 $r_{ff,gg}$ + 2 $r_{ff+1,gg}$ + $r_{ff,gg-1}$ + $r_{ff,gg+1}$ + $r_{ff+1,gg-1}$ + $r_{ff+1,gg+1}$)
3. Und für einen Punkt in der linken, oberen Ecke eines Gitters, also wenn links und oben ein echter Rand vorliegen:
   $r_{f,g} = \frac{1}{4}$ ($r_{ff,gg}$ + $r_{ff+1,gg}$ + $r_{ff,gg+1}$ + $r_{ff+1,gg+1}$)
4. Für eine Innere Ecke, beim dem in Südost-Richtung kein Gitterpunkt existiert:
   $r_{f,g} = \frac{1}{16}$ (4 $r_{ff,gg}$ + 3 $r_{ff-1,gg}$ + $r_{ff+1,gg}$ + 3 $r_{ff,gg-1}$ + $r_{ff,gg+1}$ + 2 $r_{ff-1,gg-1}$ + $r_{ff+1,gg-1}$ + $r_{ff-1,gg+1}$)

Für die anderen Fälle seien analoge Formeln anzunehmen.

Algorithmus 4.21: *Restriktion_der_Variablen_r* berechnen in der Klasse *Restriktion* (Full Weighting) für Feingitter-Innenpunkte
Input: Aufruf der Funktion *tun_r* in der Klasse *Restriktion* der Ebenen h und h+1.
Output: Das Feld zu $r_{f,g}$ wird aktualisiert.

1. **for all** (f,g) $\in G_{h+1}$
2. ff = 2 f und gg = 2 g
3. **if**(*Zahl-Selbstnachbarn*(ff,gg)(der feinen Ebene)=255) **then**
4. Die passende Full Weighting Formel für r - in einem der vier Fälle, s.o..
5. **else**
6. Aufnahme des Punktes (f,g) in die Restpunktemenge
7. **endif**
8. **end for**

Algorithmus 4.22: *Restriktion_der_Variablen_r* berechnen in der Klasse *Restriktion* (Full Weighting) für Restpunkte
Input: Aufruf der Funktion *tun_r* in der Klasse *Restriktion* der Ebenen h und h+1.
Output: Das Feld zu $r_{f,g}$ wird aktualisiert.

1. **for all** (f,g) $\in$ Restpunktemenge
2. ff = 2 f und gg = 2 g
3. Die passende Full Weighting Formel - in einem der vier Fälle, s.o..
4. **end for**

2. Die Wahl des Randaustausches

Die Wahl des Randaustausches ist *randaustausch_r* auf dem feinen Gitter als Quelle der Werte zur Berechnung der Restriktion von r, da die Injektion von u keinen Randaustausch benötigt.

3. Die Funktion *Restriktion_für_u_und_r* der *MLAT-sequentiell-Funktionen*

Algorithmus 4.23: *Restriktion_der_Variablen_r_und_u* sequentiell in der Klasse *ZyklusRechnen*
Input: Ebene h
Output: Durchführung der Restriktion

1. *Restriktion_u* zwischen h und h+1 für den ganzen Rand(h+1) (Injektion)
2. *Restriktion_u* zwischen h und h+1 für das Innere(h+1) (Injektion)
3. *Restriktion_r* zwischen h und h+1 für den ganzen Rand(h+1) (Full Weighting)
4. *Restriktion_r* zwischen h und h+1 für das Innere(h+1) (Full Weighting)

Bemerkung: Hier wird auch der echte Rand durchlaufen.

4. Die Funktion *Restriktion* r und u der *MLAT-parallel-nicht-adaptiv--Funktionen*

Algorithmus 4.24: *Restriktion_der_Variablen_r_und_u_parallel_und_nicht_adaptiv* in der Klasse *ZyklusRechnen*
Input: Ebene h
Output: Durchführung der Restriktion

1. *randaustausch_r* auf der Ebene h starten
2. *Restriktion_u* zwischen h und h+1 für den ganzen Rand(h+1) (Injektion)
3. *Restriktion_u* zwischen h und h+1 für das Innere(h+1) (Injektion)
4. *Restriktion_r* zwischen h und h+1 (Full Weighting) für Feingitter-Innenpunkte d.h. Algorithmus 4.21
5. *randaustausch_r* auf der Ebene h vollenden
6. *Restriktion_r* zwischen h und h+1 (Full Weighting) für die Restpunktmenge, d.h. Algorithmus 4.22

5. Die Funktion *Restriktion* der *MLAT-parallel-adaptiv-Funktionen*

Dazu kann man den Algorithmus 4.24 mit einer Abweichung verwenden:

Die Grobgitterpunktmenge, die herangezogen wird für das, was gerechnet wird, wird eingeschränkt darauf, ob Punkte darunter existieren. Das hat Folgen für die Funktionen, die auf diese Restriktionswerte zugreifen. Sie müssen unterscheiden, ob Restriktionswerte verfügbar sind, oder nicht. Diese Änderung gilt aber nur für den adaptiven Fall, weil ansonsten die Gitterpunkte vorhanden sind.

4.3.5 Berechnung der rechten Seite

Vorbemerkungen: Die Werte der Variablen **rechte Seite** (rS) wird als Inputwerte der Glätter genommen. Deswegen wird bei der Initialisierung auf allen möglichen Punkten rS(f,g) = f(f,g) genommen.

Dann geht es so weiter : Beim FAS bzw. dem adaptiven Schema wird rS, wie danach dargestellt, berechnet, vgl. [43]. Beim normalen Mehrgitterschema hingegen wird keine rechte Seite berechnet, und es bleibt rS = f.

1. Das sequentielle Rechnen in der Klasse *Rechnen* - die Berechnung der rechten Seite

Algorithmus 4.25:
Input: Die Punktemenge L, ohne echten Rand
Output: Aktualisierung der Werte von rS

1. **for all** (f,g) $\in$ L
2. **if**(es existiert ein feinerer Punkt unter (f,g)) **then**
3. $rS_{f,g}$ = $Iu_{f,g}$ - $Iu_{f-1,g}$ - $Iu_{f+1,g}$ - $Iu_{f,g-1}$ - $Iu_{f,g+1}$ + $r_{f,g}$
4. **else**
5. $rs_{f,g}$ = $f_{f,g}$
6. **end if**
7. **end for**

2. Die Wahl des Randaustausches

Es erfolgt ein Randaustausch nur für die Iu-Werte, weil der Wert für r nur auf dem Punkt (f,g), und nicht auf seinem Außenrand, zur Verfügung stehen muss.

3. Die Funktion *berechne_rechte_Seite* der *MLAT-sequentiell-Funktionen*

Algorithmus 4.26: *berechne_rechte_Seite_sequentiell* in der Klasse *ZyklusRechnen*:
Input: Die Ebene h
Output: Der Wert rS ist aktualisiert.

1. *berechne_rechte_Seite* auf Ebene h(für unechte Randpunkte der Ebene h)
2. *berechne_rechte_Seite* auf Ebene h(für Innenpunkte der Ebene h)

4. Die Funktion *berechne_rechte_Seite* der *MLAT-parallel-nicht-adaptiv-Funktionen*

Algorithmus 4.27: *berechne_rechte_Seite_parallel_nicht_adaptiv* in der Klasse *Zyklus-Rechnen*
Input: Gitterebene h
Durchgeführte Berechnung von u.

1. *randaustausch_Iu0* auf Ebene h starten

2. *rechte_Seite_berechnen* auf Ebene h(für Innenpunkte der Ebene h)

3. *randaustausch_Iu0* auf Ebene h vollenden

4. *rechte_Seite_berechnen* auf Ebene h(für unechte Randpunkte der Ebene h)

5. Die Funktion *berechne rechte Seite* der MLAT-parallel-adaptiv-Funktionen

Dazu kann man den Algorithmus 4.27 verwenden.
Aber man kann Kommunikationszeit sparen, weil es ausreicht, einen Randaustausch nur für die Punkte zu machen, unter denen Feingitterpunkte liegen, vgl. Abschnitt 4.2.7 bzw. Abschnitt 4.3.4.

Damit sind die Funktionen für *MLAT_1* fertig. *MLAT_2* besteht aus einem Löser, wurde also schon diskutiert. Bleiben also *MLAT_3* und *MLAT_4* und die *MLAT_Randinterpolation* übrig.

4.3.6 Berechne die Variable e des Mehrgitterschemas

1. Das sequentielle Rechnen in der Klasse *Rechnen* - die Berechnung von e

Algorithmus 4.28: *berechne_e* in der Klasse *Rechnen*
Input: Eine Liste von Gitterpunkten: L
Output: Das Feld zu $e_{f,g}$ wird aktualisiert.

1. **for all** (f,g) $\in$ L

2. $e_{f,g} = u_{f,g}$ - $Iu_{f,g}$

3. **end for**

3. Die Funktion *berechne_e* der *MLAT-sequentiell-Funktionen*

Algorithmus 4.29: *berechne_e_sequentiell* in der Klasse *ZyklusRechnen*
Input: Gitterebene h
Output: Durchgeführte Berechnung von e.

1. *berechne_e* auf Ebene h(Innenpunkte der Ebene h)

2. *berechne_e* auf Ebene h(unechte Randpunkte der Ebene h)

4. und 5. Die Funktion *berechne_e* der *MLAT-parallel-Funktionen-(nicht)-adaptiv*

Man verwendet in beiden Fällen den Algorithmus 4.29. Denn: Da hier nur auf dem Punkt gerechnet wird, und nicht in seiner Umgebung, entfällt jegliche Kommunikation.

4.3.7 Die (parallele) Prolongation

1. Die sequentielle Prolongation in der Klasse *Prolongation*

Definiert wird die Prolongation in Bezug auf die Ebenen h und h+1.

Vorbemerkung 1: Die Punktmenge G_h sei in der Klasse *Prolongation* über Zeiger direkt zugreifbar. Für die Variablen u und e sind für beide Ebenen Zeiger da, sodass der Zugriff von der Klasse *Prolongation* auf alle nötigen Strukturen möglich ist.

Vorbemerkung 2: Da der Zugriff auf die Feingitterpunktemenge erfolgt, ist zwischen dem adaptiven und dem nicht adaptiven Fall keine Unterscheidung nötig.

Vorbemerkung 3: Hier wird die Prolongation für die Variable u gemacht. Sie kann aber auch für die Variable e nötig sein, ggf. für beide Variablen gemischt. Im adaptiven Fall ist für neu hinzugekommene Punkte ggf. eine andere Prolongation nötig als für schon vorhandene Punkte. Es gibt dann für jeden Gitterpunkt die Unterscheidung, ob ein Punkt neu hinzugekommen ist oder nicht, und gegebenenfalls kommt es zu verschiedenen Prolongationen.

Algorithmus 4.30: Die Bestimmung eines Prolongationswertes für einen Punkt
Input: Die Koordinate (f,g) auf dem feinen Gitter, das Feld u0-grob
Output: Abspeichern des Prolongationswertes u_0-fein(f,g)

1. ff = $\lfloor \frac{f}{2} \rfloor$, gg = $\lfloor \frac{g}{2} \rfloor$
2. **if** f=0(mod 2) **AND** g=0(mod 2) **then**
   u0-fein (f,g) = u0-grob (ff,gg)

   if f=1(mod 2) **AND** g=0(mod 2) **then**
   u0-fein (f,g) = $\frac{1}{2}$ (u0-grob (ff,gg) + u0-grob (ff+1,gg))

   if f=0(mod 2) **AND** g=1(mod 2) **then**
   u0-fein (f,g) = $\frac{1}{2}$ (u0-grob(ff,gg) + u0-grob(ff,gg+1))

   if f=1(mod 2) **AND** g=1(mod 2) **then**
   u0-fein (f,g) = $\frac{1}{4}$ (u0-grob(ff,gg) + u0-grob(ff+1,gg) + u0-grob(ff,gg+1) + u0-grob(ff+1,gg+1))

Algorithmus 4.31: Die Bestimmung der rein sequentiellen Prolongation in der Klasse *Prolongation* auf dem Rand oder im inneren Gebiet
Input: keiner
Output: Abspeichern aller sequentiell berechneten Prolongationswerte u_0(f,g) für den Rand oder die Innenpunktmengen

1. Algorithmus: Berechnung für die Randpunktmenge

   1. **for all** (f,g) $\in$ Randpunkte(G_h)
   2. *Bestimme den Prolongationswert* für (f,g): Algorithmus 4.30

3. **end for**

2. Algorithmus: Berechnung für die Innenpunktmenge

1. **for all** (f,g) $\in$ Innenpunkte(G_h)

2. *Bestimme den Prolongationswert* für (f,g): Algorithmus 4.30

3. **end for**

Bemerkung 1: Man hat hier nicht einen Algorithmus mit 2 Punktmengen, sondern zwei separate Algorithmen.

Bemerkung 2: Die Unterscheidung nach Rand- und Innenpunkten ist **wesentlich** in der Klasse *Zyklus Rechnen*.

2. Der Randaustausch für die Prolongation

Ein Kandidat für den Randaustausch. Der Ursprung bzw. die Quelle der Prolongationsberechnung sind die Grobgitterwerte. Denn die Prolongation benötigt Grobgitterdaten für ihre Berechnung. Deswegen wäre ein Grobgitter-Randaustausch der Kandidat für die Art des Randaustausches.

Der normale Randaustausch auf dem groben Gitter reicht nicht aus. Das ist leicht begründet: Man nehme z.B. einen Punkt mit f = 1 (mod 2) und g = 1 (mod 2), also einen 'mittleren Punkt' bzgl. der Grobgitterwerte. Dieser Punkt sei umgeben von Punkten, die auf anderen Gittern liegen. Dann verfügt dieser Punkt selber über keine Grobgitterpunkte in der Nähe. Ein Grobgitter-Randaustausch würde an dieser Stelle ins Leere laufen.

Die Lösung. Man löst dieses Problem, indem man einen Randaustausch auf dem feinen Gitter macht, nur für gemeinsame Punkte, d.h. Punkte mit f = 0 (mod 2) und g = 0 (mod 2), aber nicht die Feingitterwerte überträgt, sondern die Grobgitterwerte an dieser Stelle, vgl. den Fall * = gg_u0 in Abschnitt 4.2.6. In diesem Fall wird der isolierte Punkt mit allen nötigen Grobgitterinformationen versorgt.

3. Die Funktion *Prolongation* der *MLAT-sequentiell-Funktionen*

Algorithmus 4.32: Die Prolongation sequentiell berechnen in der Klasse *ZyklusRechnen*:
Input: Zugriff auf eine Klasse *Prolongation* (Feingitterebene h)
Output: Der Wert u ist aktualisiert.

1. *Prolongation* in Bezug auf die Ebenen h und h+1 berechnen im Inneren

2. *Prolongation* in Bezug auf die Ebenen h und h+1 am Rand berechnen

4. Die Funktion *Prolongation* der *MLAT-parallel-nicht-adaptiv-Funktionen*

Algorithmus 4.33: *Prolongation_parallel_berechnen_nicht_adaptiv* in der Klasse *ZyklusRechnen*
Input: Die Gitterebene
Output: Durchgeführte Berechnung von u.

1. *randaustausch_gg_u0 starten* auf Ebene h
2. *Prolongation_Innen* für die Ebenen h und h+1 auf dem feinen Gitter
3. *randaustausch_gg_u0 vollenden* auf Ebene h
4. *Prolongation_am_Rand* für die Ebenen h und h+1 auf dem feinen Gitter

Die Funktion *Prolongation* der *MLAT-parallel-adaptiv-Funktionen*

Dazu kann man den Algorithmus 4.33 verwenden. Aufgrund der Verwendung der Feingitterpunktmenge werden mit demselben Algorithmus in beiden Fällen genau die benötigten Werte berechnet.

4.3.8 Grobgitter-Korrektur auf feinem Gitter

1. Die Grobgitter-Korrktur in der Klasse *Rechnen*, also eine rein sequentielle Berechnung

Algorithmus 4.34: *Grobgitterkorrektur_berechnen* in der Klasse *Rechnen*
Input: Eine Liste L von Gitterpunkten, ohne echte Randpunkte
Output: Das Feld zu $u_{f,g}$ wird berechnet.

1. **for all** (f,g) $\in$ L
2. $u_{f,g} = u_{f,g} + e_{f,g}$
3. **end for**

2. Der Randaustausch

Hier ist kein Randaustausch nötig.

3. Die Funktion *Grobgitterkorrektur* der *MLAT-sequentiell-Funktionen*

Algorithmus 4.35: *Grobgitterkorrektur_berechnen_sequentiell* in der Klasse *Zyklus-Rechnen*:
Input: Die Ebene h
Output: Der Wert u ist aktualisiert.

1. *Grobgitterkorrektur_berechnen* auf Ebene h(für Innenpunkte der Ebene h)
2. *Grobgitterkorrektur_berechnen* auf Ebene h(für unechte Randpunkte der Ebene h)

4. 5. Die Funktion *Grobgitterkorrektur* der *MLAT-parallel-Funktion-(nicht)-adaptiv*

Man verwendet dafür den Algorithmus 4.35, da kein Randaustausch nötig ist.

Damit ist *MLAT_3* fertig; *MLAT_4* ist auch fertig, da der Glätter schon behandelt worden ist. Bleibt die Randinterpolation übrig.

4.3.9 Die (parallele) Randinterpolation

Bei der Randinterpolation wird kein Algorithmus im Sinne der Abarbeitung von Schritten durchgeführt, sondern die Methode erläutert, wie man eine parallele Randinterpolation erhält. Zuerst wird gesagt, in welchen Schritten dieses erfolgt.

Ein kurzer Überblick über die Teile dieser Darstellung

Am Anfang wird die Randinterpolation vorgestellt. Dann wird aufgezeigt, wie sie im Prinzip funktioniert. Dann wird dargestellt, wie ihre Parallelisierung funktioniert, indem gesagt wird, welche Werte an andere Kerne verschickt werden. Am Ende kommt die Ausführung der Interpolation, basierend auf den dann vorhandenen Werten.

Was ist die Randinterpolation?

Die Randinterpolation erfolgt am echten Rand. Quelle der Informationen ist das grobe Gitter. In das feine Gitter wird eingefügt. Dabei werden die Grobgitterwerte interpoliert. Als Feingitterpunkte werden dabei solche genommen, die zwischen zwei Grobgitterpunkten stehen.

Wie die Randinterpolation im Prinzip funktioniert.

Gegeben sei ein Punkt (f,g) auf einem feinen Gitter, und rechts daneben seien keine Gitterpunkte mehr. Ferner sind auf der Position (f-1,g) und (f+1,g) gemeinsame Gitterpunkte, d.h. dort liegen Grobgitterpunkte und Feingitterpunkte übereinander. Dann kann man die Grobgitterwerte auf den Koordinaten zu den Feingitterkoordinaten (f-3,g), (f-1,g), (f+1,g) und (f+3,g) heranziehen, um durch eine kubische Interpolation den Feingitterwert auf den Koordinaten (f,g) zu bekommen.

Man bekommt die Formel dafür, indem man einem Polynom 3.ten Grades an den Stellen -3, -1, 1 und 3 Werte vorgibt. Man erhält als Wert einer kubischen Interpolation die folgende Formel, wobei man die u-Werte auf der rechten Seite vom groben Gitter auf den vorgegebenen Positionen bekommt:
u(f,g) = -0.0625 u(f-3,g) + 0.5625 u(f-1,g) + 0.5625 u(f+1,g) - 0.0625 u(f+3,g)
Hat man andere 4 Stützstellen, dann ergeben sich durch Anwendung der kubischen Polynome entsprechende Formeln. Hat man weniger als 4 Stützwerte, muss man Polynome geringeren Grades verwenden.

Es werden im Folgenden 2 Aufgaben bearbeitet: 1. Wie werden die Grobgitterwerte kommuniziert, um Stützstellen zu bekommen? 2. Wie erfolgen mit diesen Werten optimal die nötigen Randinterpolationen?

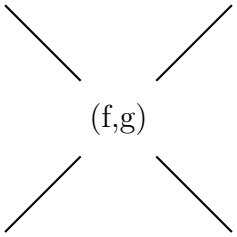


Abbildung 4.5: Der Check auf den Diagonalen

Markierung der Werte

Zuerst ist zu sagen, dass nach der Kommunikation klar sein muss, welche Grobgitterwerte vorhanden sind. Man zieht dazu die Struktur Feld der Feldliste (in der Klasse Gitter) heran. Die Idee besteht dann darin, dass man Werten in diesem Feld - sie sind ganzzahlig - die ≥ 0 sind, die Bedeutung gibt, dass ein Eintrag vorliegt.
Im Sinne dieser Idee wird für einen kommunizierten Wert während der Randinterpolation eine 2 in sein Feld der Grobgitterfeldliste eingetragen. Es sind dann genug Werte ≥ 0 vorhanden, und man kann mit den auf diese Weise als bekannt identifizierten Werten die Interpolation durchführen.

Bemerkung: Am Ende der Randinterpolation müssen die aufgrund der Kommunikation geänderten Einträge in der Feldliste wieder rückgängig gemacht werden.

Durchführung der Kommunikation

Der folgende Algorithmus 4.36 hat in den Schritten der Fallunterscheidung und der Behandlung der Fälle, was in den Schritten 4 und 5 von Algorithmus 4.36 geschieht, ausführliche Erklärungen.
Zuvor eine Definition: Sei Richtung (k = 0) NORD_OST, Richtung (k = 1) SÜD_OST, Richtung (k = 2) SÜD_WEST und Richtung (k = 3) NORD_WEST. Damit bekommt eine Menge $\subseteq$ {NO,SO,SW und NW} eine Zahl zwischen 0 und 15 zugeordnet, indem man der Richtung i das Bit i in der Zahl zugeordnet.

Algorithmus 4.36:

1. Mache eine Randinterpolation auf der groben Gitterebene für die Variable u.
   Bemerkung: Dadurch erhöht sich am Rand die vorhandene Information um 1.

2. Durchlaufe den Rand des feinen Gitters in Zeit O(Anzahl Randpunkte). Man hat dann jeweils den Feingitterpunkt (f,g).

3. Behandele nur die Punkte, die unter einem Grobgitterpunkt (ff,gg) liegen (gemeinsame Punkte).

4. Checke das Farbenfeld der 4 zu (f,g) diagonal liegenden Punkte, ob sie < 0 sind, vgl. Abbildung 4.5. Für k zwischen 0 und 3: setze das Bit k der 4-Bit-Variablen v genau dann, wenn das Feingitterfarbenfeld von (f,g) in Richtung(k) < 0 ist, d.h. dort kein Gitterpunkt existiert.
   Die Variable v hat 16 Ausprägungen, und man unterscheidet hier insgesamt 3 Typen, vgl. Abbildung 4.6: In Fall A sind die Bits bis auf SÜD_OST gesetzt, in

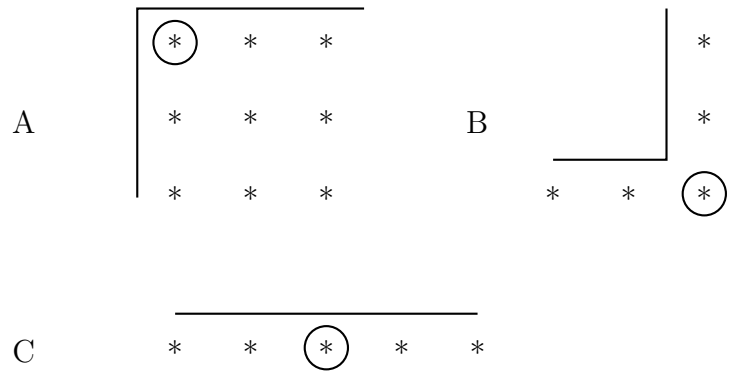


Abbildung 4.6: Die Fälle A, B und C bei der Kommunikation der Randinterpolation. Die mit * markierten Punkte sind Grobgitterpunkte

Fall B ist nur das Bit für NORD_WEST gesetzt, und in Fall C sind die Bits für NORD_WEST und NORD_OST gesetzt. Damit gilt: In A liegen nach links und nach oben keine Punkte, in B liegen oberhalb auf der linken Seite keine Punkte, und in C liegen oberhalb keine Punkte.

5. Nun wird gesagt, was in den Fällen A, B und C zu kommunizieren ist. Es wird dabei für die Kommunikation der Terminus 'wird bedient' verwendet: Dass ein rechter Punkt von links her bedient wird, bedeutet, dass die Grobgitterwerte links ab (ff,gg) $\in G_H$ zu dem Kern der Farbe *farbenfeld(f,g+1)* mit (f,g+1) $\in G_h$ geschickt bzw. kommuniziert werden, soweit möglich, aber maximal 3 davon, d.h. maximal bis (ff,gg-2). Dabei ist die mögliche Reichweite wegen des Grobgitter-Randaustausches zu Beginn ein Feld weiter, als das eigene Gitter reicht, also bis zum Außenrand; maximal bis zum echten Rand des groben Gitters. Sie können also insbesondere auch jenseits des feinen Gitters vorhanden sein. Die entsprechende Funktion heißt *scan_west_für_ost*, falls der rechte Punkt 'bedient' wird.

   Im Fall A wird der rechte Punkt (f,g+1) von links her 'bedient' (falls möglich), und der untere Punkt (f+1,g) von oben her. Zu den Koordinaten: Der Punkt (f,g) ist der Feingitterpunkt unter dem 'umrandeten' Grobgitterpunkt.
   Im Fall B wird der linke Punkt von rechts her bedient und der obere Punkt von unten her.
   Im Fall C wird der linke Punkt von rechts her bedient und der rechte Punkt mit den links liegenden Grobgitterwerten.

Begründung: Auf diese Weise werden die Punkte am echten Rand mit Grobgitterinformationen versorgt.

Bemerkung: Bei der Programmierung wird die Situation alternierender Punktzugehörigkeit ausgenutzt. Das bedeutet: Wenn der Kern die ID 0 hat, und in einer Randlinie die Grobgitterpunkte die ID's 0 1 0 haben, dann werden, falls das erwünscht ist, alle 3 Werte übertragen, weil der mittlere Wert durch den Grobgitterrandaustausch bestimmt wurde, und die vorhandene Information für alle Einträge reicht.

Die Durchführung der Randinterpolation nach der Kommunikation

Algorithmus 4.37: Die *Randinterpolation* nach der Kommunikation dafür.

1. **for all** (f,g) $\in$ Rand des feinen Gitters
2. **if** fgff(f,g-1) < 0 **OR** fgff(f,g+1) < 0 **then**
   (echter Rand nach Westen oder Osten)
3. Führe eine Nord-Süd Interpolation aus, und gehe zum nächsten Rand-Gitterpunkt
4. **if** fgff(f-1,g) < 0 **OR** fgff(f+1,g) < 0 **then**
   (echter Rand nach Norden oder Süden)
5. Führe eine Ost-West Interpolation aus.
6. **end for**

Bleibt noch die Nord-Süd-Interpolation zu beschreiben. (Die Ost-West-Interpolation erfolgt analog.) Bei einem gemeinsamen Punkt wird der Grobgitterwert aufs feine Gitter kopiert. Ansonsten hat man einen Zwischenpunkt, der entweder horizontal oder vertikal zwischen Grobgitterrandpunkten liegt.

Man interpoliert möglichst kubisch und symetrisch nach Norden und Süden. Inwieweit das möglich ist, hängt von den Markierungen ab. Das sind Einträge im Feld der Grobgitterfeldliste, die ≥ 0 sind. Es werden dazu die Werte von maximal 3 Grobgitterpunkten nördlich bzw. südlich des Feingitterpunktes herangezogen.

Bemerkung: An den Rändern, wo daneben keine Gitterpunkte sein können, wie z.B. im Fall g=0, muss immer eine Rand-Interpolation gemacht werden.

Fertig stellen durch Remarkierung

Nach Beenden dieser Prozedur werden die markierten Eintragungen in der Datenstruktur Feld des groben Gitters wieder entfernt.

4.4 Die Verfeinerungsentscheidungen des Multi-Level-Gitters

An dieser Stelle wird die Bestimmung der Verfeinerungsinformation besprochen.

4.4.1 Die Eigenschaftspunkte bestimmen

Vorbemerkungen

Hier behandeln wir den Fall dynamisch veränderbarer Verfeinerungsgitter. Deswegen ist die Bestimmung von Verfeinerungsgebieten dauerhafter Teil des Programms. Es kann der Fall statischer Gitter natürlich als Unterfall auch behandelt werden.
Das Programm arbeitet mit lokaler Verfeinerungsinformation. Damit ist gemeint, dass

jeder Kern die Verfeinerungsinformation nur auf seinem Gebiet zu kennen braucht, und zwar in dem Teil des Programms, wo die Verfeinerungsinformation bestimmt wird, nämlich in diesem Abschnitt 4.4.1. Alle weiteren Aufgaben übernimmt das Programm selber.
Die Konsequenz ist, dass beides realisiert werden kann: Man kann Verfeinerungsbewegungen simulieren, indem auf allen Kernen bekannte Verfeinerungsinformationen in lokale Verfeinerungsinformationen umgewandelt werden. Und man kann selbstadaptive Verfeinerung machen, wo die Verfeinerungsstellen selber durch Berechungen ermittelt werden.

Im Sinne dieser Situation wurden einerseits Simulationen etabliert - mit verschiedenen Verfeinerungsbewegungen, und es wurden für die Poisson-Gleichung selbstadaptive Verfeinerungen realisiert. Die Darstellung für beides, die Simulation wie die selbstadaptive Verfeinerung, erfolgt in zwei Schritten: Zuerst wird angegeben, wie die Gitterpunkte festgelegt werden. In einem zweiten Schritt wird das Feld der Eigenschaftspunkte beschrieben. Es wird dabei in eine Punktfeldliste geschrieben, wodurch die Verfeinerungsinformation nicht nur als zweidimensionales Feld vorliegt, sondern auch als Liste. Insbesondere kann das Feld der Eigenschaftspunkte daher nach jedem Zeitschritt wieder zurückgesetzt werden.

Die Eigenschaftspunkte werden in *pfl_feld_aktuell* und *pfl_feld_vergangen* gespeichert. Sie sind als *PunktFeldlisten* gemäß Abschnitt 5.4.3 definiert. Im Folgenden wird auch nur *feld_aktuell* und *feld_vergangen* geschrieben. Dass man hier nicht nur ein Feld, sondern auch die Punktliste benötigt, wird klar in Abschnitt 5.4.5.

Die Multi-Level-Simulation von Verfeinerungsgittern

Vorbemerkung zur Simulation von Verfeinerungsgittern
Es gibt an dieser Stelle eine Auswahl an Szenarien und Verfeinerungsgittergrößen über die Parameter für die Simulationsnummer und, je nach Simulationsnummer, für die Verfeinerungsgittergröße.
Außerdem ergibt sich eine Variation der Einstellungen der Simulation durch die Angabe der Verfeinerungsebenenanzahl und die Gittermaße n und m.
Und das konkrete Verfeinerungsgitter ist abhängig von der Zeitschrittvariablen t, weil es sich ja im Allgemeinen bewegt.

Schritt 1: Die Festlegung der Parameter. Die Berechnung der Verfeinerungsgitter geschieht in folgendem Algorithmus:

Algorithmus 4.38: Bestimmung der Koordinaten der Verfeinerungsrechtecke bei der Simulation
Input: Simulationsnummer sim-nummer, Verfeinerungsgittergrößenindex vgitter (legt auch die Größe des Verfeinerungsgitters fest), Gittergröße (n,m), Verfeinerungsebenenanzahl h_{min}sowie Zeitpunkt t
Output: l_anzahl ist die Anzahl von Verfeinerungsgittern, und (oben, unten, links, rechts) sind die Parameter für jedes Verfeinerungsrechteck sowie jede Gitterebene, für das Verfeinerungsinformationen bereitgestellt werden.

1. **switch**(sim-nummer)
2. **case** i:(i zwischen 0 und maximale Variantenanzahl - 1)
3. l_anzahl = Anzahl(Verfeinerungsgittern)(i)
4. **switch**(vgitter)
5. **case** j:(j zwischen 0 und Gittergrößenauswahl - 1)
6. **for** l = 1 **to** l_anzahl
7. oben(l) = Formel_für_oben $_{i,j,l}$ (n,m,t);
   unten(l) = Formel_für_unten $_{i,j,l}$ (n,m,t);
   links(l) = Formel_für_links $_{i,j,l}$ (n,m,t);
   rechts(l) = Formel_für_rechts $_{i,j,l}$ (n,m,t);
8. **end for**
9. **end switch**
10. **end switch**
11. **for** l = 1 **to** l_anzahl
12. **for** i = 2 **to** h_{min}
13. eigenschaftspunkte[i].oben(l) = oben(l),
    eigenschaftspunkte[i].unten(l) = unten(l),
    eigenschaftspunkte[i].rechts(l) = rechts(l),
    eigenschaftspunkte[i].links(l) = links(l)
14. oben(l) = oben(l)/2-1
    unten(l) = unten(l)/2+1
    links(l) = links(l)/2-1
    rechts(l) = rechts(l)/2+1
15. **end for**
16. **end for**

Diese Funktion wird aufgerufen, bevor die Eigenschaftspunkte bestimmt werden.

Bemerkung 1: Die maximale Variantenanzahl gibt an, wie viele verschiedene Simulationen es gibt. Die Gittergrößenauswahl mit der Variablen vgitter ermöglicht es, verschieden große Gitter pro Simulationsvariante zu wählen.

Bemerkung 2: Die Formeln Formel_für_oben $_{i,j,l}$ usw. für die Rechteckkoordinaten legen, in Abhängigkeit der angegebenen Parameter fest, wie die jeweiligen Rechtecke aussehen sollten.

Bemerkung 3: In den Zeilen 1 bis 7 werden die Verfeinerungsentscheidungen für die

Ebene 1 getroffen.

Bemerkung 4: Die Zeilen 8 bis 11 haben den Zweck, die Verfeinerungsgitter auf den anderen Gitterebenen mit Verfeinerungsentscheidungen zu definieren.

Bemerkung 5: Die Rechteckkoordinaten werden in der Klasse *Eigenschaftspunkte* gespeichert.

Schritt 2: Die Verfeinerungsberechnung. Der Algorithmus hierfür im Falle der Simulation einer Rechtecksbewegung:

Algorithmus 4.39: *Verfeinerungspunkte_berechnen* in der Klasse *Eigenschaftspunkte*
Input: Der Output von Algorithmus 4.38
Output: Die *PunktFeldliste pfl_feld_aktuell* zur Information *feld_aktuell* (Die Verfeinerungsinformation)

1. *pfl_feld_aktuell.reset*: Reset der Verfeinerungspunkte
2. **for all** (f,g)$\in$ Feldliste von Gitter
3. **for** i = 1 **to** l_anzahl
4. **if**(links(i) $\leq$ f $\leq$ rechts(i)) **then**
5. **if**(oben(i) $\leq$ g $\leq$ unten(i)) **then**
6. *pfl_feld_aktuell.hinzufügen*(f,g)
   (ggf. feld_aktuell(f,g) = true)
7. **end for**
8. **end for**

Bemerkung: In gewissen Fällen wäre es schneller, die Blöcke zu durchlaufen, statt alles zu durchlaufen, und dann über if-Abfragen zu testen. Das ist aber nicht mehr unbedingt der Fall, wenn sich die Gitter über die Gebiete mehrerer Kerne erstrecken.
In jedem Fall ist die Programmierung hier näher dran an den Varianten zur Selbstadaptivität.

Die Verfeinerungsberechnung der Mehrgittertheorie

Schritt 1: Die Festlegung der Entscheidungswerte Es wird die Variable τ berechnet, und zwar auf der Ebene über der Entscheidungsebene, weil Restriktionen von der Entscheidungsebene aus erfolgen. Die Formel der Theorie lautet ([36] bzw. [43]):

$\tau_H^h = \mathrm{L}_H\ \mathrm{I}_h^H\ \mathrm{u}_h$ - $\mathrm{I}_h^H\ \mathrm{L}_h\ \mathrm{u}_h$.

Die Idee dabei ist, die u Werte eingesetzt in den Differentialoperator auf Gittern unterschiedlicher Gitterebenen miteinander zu vergleichen, um zu sehen, wie viel besser

die Werte auf der feineren Ebene als auf der gröberen Ebene sind. Dabei ist die feinere Ebene die, wo die Verfeinerungsentscheidungen getroffen werden.

Im Falle des FAS wird berechnet:

rS = L_H I_h^H u_h + r_h, und mit r_h = I_h^H (f_h - L_h u_h) folgt

τ_H^h = rS - I_h^H f_h.

Die Idee zu dieser 'Ableitung' entstammt [43].
Die Variable τ_H^h ist also sehr leicht zu berechnen. I_h^H f_h wird als Full Weighting entsprechend, wie im Abschnitt 4.3 berechnet, nur mit f statt r. Damit folgt:

Algorithmus 4.40: Verfeinerungskriterium berechnen
Input: Mehrgitter-Variablen-Informationen(f_H, rS_H, usw.), Gitterebene h, H ist die Maschenweite zur Ebene h+1
Output: Die τ_H^h auf dem groben Gitter der Ebene h+1.

1. I_h^H f_h berechnen, vgl. die Restriktionsberechung in Abschnitt 4.3.4.
2. τ_H^h berechnen auf den Ebenen h+1 und H+1, für die Innenpunkte(h+1)
3. τ_H^h berechnen auf den Ebenen h+1 und H+1, für die Randpunkte(h+1)

Bemerkung: Bei der τ - Berechnung erfolgt kein Randaustausch, weil die obige Differenz zwischen rS ud I_h^H f_h jeweils nur auf dem Gitterpunkt erfolgt, für den der Wert des Verfeinerungskriteriums berechnet wird.

Schritt 2: Die Bestimmung der Eigenschaftsinformation.

Algorithmus 4.41: Bestimmung der Punkte, in Bezug auf die verfeinert wird - in diesem Fall der Verfeinerungsberechnung der Mehrgittertheorie
Input: Ebene h, wo die Verfeinerungsentscheidung getroffen wird. Das ist eine Ebene über der Ebene, wo sich das zu ändernde Gitter befindet.
Output: Berechnung der Punkte, für die das Verfeinerungskriterium erfüllt ist (auf dem groben Gitter, falls das feine Gitter geändert wird), in der Datenstruktur *pfl_feld_aktuell*.

1. *pfl_feld_aktuell.reset*: Reset der Verfeinerungspunkte
2. **for all** (f,g)$\in$ Feldliste von Gitter
3. **if**((f,g) hat Grobgitterpunkt darüber) **then**
4. **if**(| τ_{oben} ($\frac{f}{2}$,$\frac{g}{2}$) | > const) **then**
5. *pfl_feld_aktuell.einfügen*(f,g)
6. **end for**

Bemerkung 1: Die Variable τ_{oben} ist τ_H^h in Bezug auf die Entscheidungsebene, und damit zwei Ebenen über dem zu ändernden Gitter.

Bemerkung 2: Das Verfeinerungskriterium wird nur auf den gemeinsamen Punkten der Entscheidungsgitterebene berechnet. Daher gilt:
Ggf. muss man dann zu einem dieser Punkte aus *pfl_feld_aktuell* auch noch seine Nachbarpunkte zu *pfl_feld_aktuell* hinzunehmen.

Bemerkung 3: Die pfl-Felder *pfl_feld_aktuell* und *pfl_feld_vergangen* werden genommen, damit man die Eigenschaftspunkte bei Bedarf löschen kann.

Eine eigene Verfeinerungsberechnung

Schritt 1: Die Festlegung der Entscheidungswerte. Das Kriterium basiert darauf, wie sehr ein auf 4 Stützstellen basiertes Polynom an der Stelle (f,g) mit dem tatsächlichen Wert übereinstimmt. Die Idee ist ähnlich der Verwendung von Gradienten, weil prognostizierte Werte von Stützstellen verglichen werden mit den tatsächlichen Werten. Denn beim Gradienten wird ein Zwischenwert verglichen mit dem Wert der Strecke in der Mitte von 2 Stützstellen. Man vergleiche dazu auch die Verfeinerungskriterien in [38].

Um auf vier Stützstellen zu kommen, kam es nun zu dieser Idee: Normalerweise hat man die Werte direkt daneben, aber nicht auf den weiter entfernten Punkten verfügbar. Wenn man aber einen gemeinsamen Punkt hat (f(mod 2) = g(mod 2) = 0), kann man per geeignetem Randaustausch, vgl. 4.2.6 mit * = fg_u0, über das grobe Gitter den Feingitterwert im Abstand 2 nehmen.

Damit lässt sich dann berechnen für gerades (f,g):

τ_1(f,g) = $\frac{2}{3}$ ($u_{f-1,g}$ + $u_{f+1,g}$) + $\frac{1}{6}$ ($u_{f-2,g}$ + $u_{f+2,g}$) - $u_{f,g}$
τ_2(f,g) = $\frac{2}{3}$ ($u_{f,g-1}$ + $u_{f,g+1}$) + $\frac{1}{6}$ ($u_{f,g-2}$ + $u_{f,g+2}$) - $u_{f,g}$

Daraus folgt diese sequentielle Berechnung:

Algorithmus 4.42: τ_1 und τ_2 sequentiell berechnen
Input: Ebene h, für die die Verfeinerungsinformationen beschafft werden müssen.
Output: τ_1 und τ_2 für diese Ebene

1. Berechne τ_1 und τ_2 auf Ebene h für gerade (gemeinsame) Innenpunkte(Ebene h)

2. Berechne τ_1 und τ_2 auf Ebene h für gerade (gemeinsame) Randpunkte(Ebene h) (nicht für echte Randpunkte)

Dabei werden ggf. für eine Vergleichbarkeit mit der parallelen Version nur die Werte für die gemeinsamen Punkte berechnet, obschon im sequentiellen Fall die Werte auch

für Punkte, die keine gemeinsamen Punkte sind, berechnet werden könnten.

Die parallele Berechnung erfolgt so:

Algorithmus 4.43: τ_1 und τ_2 parallel berechnen
Input: Ebene h
Output: τ_1 und τ_2 für diese Ebene

1. *Randaustausch_u* auf der Ebene h (für die Punkte direkt daneben)

2. *Randaustausch_fg_u* (also mit * = fg_u) auf der Ebene h+1,vgl. Abschnitt 4.2.6

3. Anwendung von Algorithmus 4.42

Bemerkung: Es wird nur für gemeinsame, nicht echte Randpunkte gerechnet. Diese Punkte haben einen Mindestabstand 2 zum echten Rand, denn gemeinsame Punkte können keinen Abstand 1 zum echten Rand haben. Daher existieren die Punkte (f-2,g), (f+2,g), (f,g-2) und (f,g+2).

Schritt 2: Die Bestimmung der Eigenschaftsinformation
Algorithmus 4.44: Verfeinerungspunkte berechnen
Input: Ebene h, wo die Verfeinerungsentscheidung getroffen wird. Das ist eine Ebene darüber, wo sich das zu ändernde Gitter befindet.
Output: Die Berechnung, für welche Punkte das Verfeinerungskriterium erfüllt ist, geschieht in der Datenstruktur *pfl_feld_aktuell*

1. *pfl_feld_aktuell.reset*: Reset der Verfeinerungspunkte

2. **for all** (f,g)$\in$ Feldliste von Gitter

3. **if**((f,g) hat Grobgitterpunkt darüber) **then**

4. **if**(| τ_1 (f,g) | const_1 + | τ_2 (f,g) | $\text{const}_2 > 1$) **then**

5. *pfl_feld_aktuell.hinzufügen*(f,g)

6. **end for**

Bemerkung 1: Hierbei erfolgt kein Zugriff auf die τ's eine Ebene höher, da die Variable auf der Ebene selber bestimmt wird.

Bemerkung 2: Allerdings liegt trotzdem die Information nur auf den gemeinsamen Punkten vor, weil die entfernten Werte nur dort sicher bestimmt werden können. Daher gilt, wie beim anderen Verfeinerungskriterium: Ggf. muss man dann zu jedem der ausgewählten Punkte auch noch seine Nachbarpunkte hinzunehmen.

Bemerkung 3: Zur Kombination von zwei Verfeinerungswerten in Zeile 4 von Algorithmus 4.44, vgl. [36].

4.4.2 Durch eine Veränderung der Funktionswerte die Eigenschaftspunkte beeinflussen

Zuerst wird eine spezielle Differentialgleichung für die Poisson-Gleichung formuliert.

Mit $x_i = i * h_i$ und $y_j = j * h_y$, wobei h_x und h_y die Gitterbreiten sind auf einem Gitter der Größe [0,1] x [0,1], folgt:
$f(i,j) = -6 * (x_i^4 \, y_j + 2 * x_i^2 \, y_j^3)$ sowie
$u(i,j) = x_i^4 \, y_j^3$ auf dem Rand des Einheitsquadrates.
Diese Formeln sind [43] entnommen.

Die Änderungen von f in der Zeit (vom Autor)

Im Fall, wo f(i,j) in der Zeit geändert werden soll, wird nur die Formel für y_j geändert: Statt $y_j = j * h_y$ wird ausgegangen von $y_j^* = j * h_y + t_d$, wobei t_d eine lineare Funktion der Zeit t und der Gitterbreite h_y ist. Bei $y_j^* \leq 1$ wird $y_j = y_j^*$ gesetzt. Im Fall von $y_j^* \geq 1$ wird ferner $y_j = 2 - y_j^*$ gesetzt, sodass immer gewährleistet ist, dass $y_j \leq 1$ ist.

4.5 Die cap

4.5.1 Was ist eine cap? Sinn und grundsätzliche Methode.

Der Sinn der cap: Wenn man viele Vergröberungsebenen hat, wird die Anzahl der zu berechnenden Gitterpunkte pro Gitterebene immer kleiner, obschon die Randkommunikation immer weiter durchgeführt werden muss. So kann auf entsprechend kleinen Gittern das sequentielle Rechnen schneller sein als das parallele. Dann kann man die Last auf wenige Kerne oder nur einen Kern legen, vgl. dazu auch [44]. Dafür muss dann die eingesparte Zeit allerdings auch noch den Aufwand rechtfertigen, Variablenwerte zu sammeln und wieder durch Zerstreuen zurückzutransportieren.
Diese Idee hat der Autor bei einem Vortrag (vgl. [70]) zum ersten Mal gehört.

Zur Methode: Es wird ab einer gewissen Ebene sequentiell gerechnet und auf den feineren Ebenen parallel. Dabei muss es eine Ebene geben auf der anfänglich parallel und danach sequentiell gerechnet wird, wenn man von den feineren zu den gröberen Gittern geht, und im umgekehrten Fall erst sequentiell und dann parallel rechnet. Diese Gitterebene nenne ich h_{cap}.

Bemerkung: Man kann die Verwendung von rechnenden Kernen auch stufenweise reduzieren, indem zwischen rein parallelem und sequentiellem Rechnen das gesamte Gitter nur auf einer Teilzahl von Kernen gerechnet wird. In diesem Programm wird aber nur parallel oder sequentiell gerechnet.

4.5.2 Die Methoden zur Etablierung der cap

Die folgenden Methoden werden nur verwendet, falls die cap 'eingeschaltet ist'. Andernfalls wird sie komplett vom Programm 'ignoriert'.

Vorbemerkung zu den nötigen Schritten: Zuerst wird erläutert, wie die Übertragung der Werte der Gitter der Kerne funktioniert. Dann geht es um die verschiedenen Punktmengen des Kerns mit der ID 0, bzgl. derer gerechnet wird: Die Punktmenge für das parallele Rechnen für $h < h_{cap}$, die Punktmengen für das sequentielle bzw. parallele Rechnen für $h = h_{cap}$, und die Punktmenge für das rein sequentielle Rechnen für $h > h_{cap}$. Dann entstehen durch Aufspaltung zusätzliche *MLAT*-Funktionen, die auf der Ebene $h = h_{cap}$ nötig werden, um teils sequentiell, und teils parallel zu arbeiten. Und als letztes ist auf den Grobgitterzyklus mit cap zu verweisen, vgl. Algorithmus 3.2, bei dem auf der Ebene h_{cap} sequentiell und parallel gearbeitet wird, und die Werte gesammelt und verstreut werden.

Sammeln und verstreuen der Werte auf der Ebene h_{cap}

Wenn nur auf einem Kern gerechnet werden soll, müssen die Werte von allen anderen Kernen gesammelt werden, und zum parallelen Rechnen hin wieder verstreut werden. Das macht die Klasse *ÜbertrageGitterwerte*.
Man muss allerdings bei deren Initialisierung festlegen, bzgl. welcher Variablen gesammelt bzw. zerstreut wird, wobei dieses beim Sammeln auch eine andere Variable sein kann als beim Zerstreuen. Je nachdem, ob ein normaler Mehrgittergrobgitterzyklus erfolgt oder ein FAS, vgl. 2.2.5, sind das verschiedene Variablen:

Fall	normales Mehrgitter	FAS
sammeln	u0	rS0
zerstreuen	e0	u0

Bemerkung 1: Welche Variablen dies sind, liegt beim Sammeln daran, wo die *MLAT_1*-Funktion zwischen sequentiell und parallel gesplittet wird, und beim Zerstreuen daran, wo die *MLAT_3*-Funktion zwischen sequentiell und parallel gesplittet wird. Beim normalen Mehrgitter sind das andere Positionen als beim FAS.

Bemerkung 2: Sammeln und zerstreuen erfolgt auf derselben Gitterebene h. Nur so ist gewährleistet, dass die Bestimmung der Punktmengen beim Sammeln auch beim Zerstreuen benutzt werden kann, was weniger Kommunikationskosten verursacht, als wenn man das anders machen würde.

Die verschiedenen Gitterpunktmengen auf dem Kern mit dem Rang 0 - je nach Gitterebene.

Die Gitterpunktmengen auf den feineren Ebenen als die der Ebene der cap bleiben erhalten. Die Gitterpunktmengen auf den gröberen Ebenen als die der Ebene der cap konzentrieren sich komplett auf den Kern mit dem Rang 0. Auf diesen gröberen Ebenen

erfolgt dann auch kein Punkttransport bei der Migration.

Auf der Gitterebene h_{cap} selber müssen beide Mengen vorliegen, da bis zum Sammeln parallel gerechnet wird, nach dem Sammeln aber sequentiell, und bis zum Zerstreuen sequentiell, nach dem Zerstreuen aber wieder parallel gerechnet wird. Dabei geschieht das Sammeln im Bereich des Zyklus vom Feinen zum Groben hin, und das Zerstreuen vom Groben zum Feinen hin.

Die Funktionen auf den verschiedenen Ebenen, insbesondere auf der Ebene h_{cap}

Bei den feineren Ebenen als die cap werden nur parallele Funktionen aufgerufen, bei den gröberen Ebenen die sequentiellen Funktionen, vgl. 4.3. Auf der Ebene selber geschieht folgendes: Beim Sammeln wird die Funktion *MLAT_1* aufgesplittet in *MLAT_1a* und *MLAT_1b*, wobei auch nach FAS oder normalem Mehrgitter unterschieden wird. Denn je nach Schema wird die Funktion *MLAT_1* unterschiedlich gesplittet. *MLAT_1a* berechnet parallel bis zu der Stelle, bei der das Sammeln erfolgt. *MLAT_1b* vollendet dann auf dieser Ebene rein sequentiell, weshalb hier auch die entsprechende Punktmenge gebraucht wird, vgl. den Abschnitt über die Punktmengen. Umgekehrt wird in *MLAT_3a* sequentiell gerechnet, dann kommt das Zerstreuen, und in *MLAT_3b* wird wieder parallel gerechnet, wobei je nach FAS oder normalem Mehrgitter die Schnitte durch die Funktionen *MLAT_1* und *MLAT_3* verschieden sind.
Mit Schnitte durch die Funktionen ist hiermit gemeint, dass der Anfang der Funktionen zu einer neuen Teilfunktion führt, und das Ende der Funktionen zu einer zweiten neuen Teilfunktion.

Bemerkung 1: Beim normalen Mehrgitterschema erfolgt der Übergang vom sequentiellen zum parallelen Rechnen und umgekehrt an anderer Stelle als beim FAS.

Bemerkung 2: Die cap wurde nur in Bezug auf die Poissongleichung realisiert, aber nicht in Bezug auf die Shallow Water Equations entsprechend Abschnitt 8.2.1.

4.6 Die Struktur oberhalb der Klasse *Programm*

In Kapitel 3 ist die Datenstruktur selber angegeben, vgl. Abbildung 3.8. Die Parameter werden dabei teilweise von *main* an *Aufrufe*, nämlich bei den Übergabeparametern des Programmstarts, und von *Aufrufe* vollständig ans *Programm* übergeben, wobei auch Parameter übergeben werden, die nicht in *main* durch Programmargumente bestimmt wurden. Anzugeben bleibt, um welche Parameter es sich handelt. Die wichtigsten werden hier nochmal aufgeführt:

Die Gitterparameter

1. m und n: Die Gitterpunkteanzahl zur Definition des feinstmöglichen Gitters

2. h_max und h_min: Die Anzahl an Vergröberungs- und Verfeinerungsebenen. Die Gesamtzahl an Gitterebenen berägt dann $h_{gesamt} = h_{min} + 1 + h_{max}$

3. h_cap: Ist keine cap aktiviert, trägt der Parameter h_cap den Wert -1. Ansonsten hat h_cap den Wert, auf welcher Ebene Sammeln und Zerstreuen geschehen.

Die Parameter zur Festlegung des Mehrgitterschemas

1. auf-ab: Die Reihenfolge, in der in dem nicht adaptiven Bereich Vergröberungs- und Verfeinerungsschritte durchführt werden. Typisch ist hier der V-Zyklus.

2. oben-nicht-adaptiv: Auswahl im nicht adaptiven Bereich: Normales Mehrgitterverfahren oder FAS? Standardmäßig hat man hier das FAS.

3. μ_1/μ_2: Anzahl an Vor- bzw. Nachglättungsschritten

Die Parameter für den Lastausgleich

1. ev-typ: Hier ist auf den Abschnitt 6.7.6 zu verweisen. Kurz gesagt werden bei ev-typ = ev alle Punkte für die Strategie gespeichert, während bei ev-typ = ev2 nur die Punkte gespeichert werden, wo sich keine Punkte darunter befinden.

2. lwk-typ: Man hat hier die Auswahl zwischen den zwei Bewertungen, ob nur Innenpunkten eine Last zugeschrieben werden soll, oder ob das auch für echte Randpunkte gelten soll.

3. PLB im eigentlichen Sinne, oder die PLB-Variante: Was von beidem wird gewählt? Bei der PLB-Variante gibt es dann noch zusätzliche Parameter.

Die restlichen Parameter

1. Der Parameter *verfeinerungstyp* gibt an, ob simuliert wird, oder eines der beiden Verfeinerungskriterien berechnet wird. Falls simuliert wird, wird noch die *simulationsnummer* für den Typ und ggf. der Parameter *vgitter* für die Gittergrößen übergeben.

2. Der Parameter *vektorisieren*: Eine boolsche Variable, die angibt, wie die Randbestimmung funktioniert. Bei true werden Vektoren bestimmt und diese kommuniziert, bei false wird die Randbestimmung mit Hilfe der Sortierung erreicht.

3. Der Parameter *iterationsgesamtzahl*: Wie viele Iterationen durchgeführt werden.

Bemerkung: Die Entscheidung für und gegen Korrektheitstests erfolgen nicht über Parameter, sondern über Präprozessorvariablen.

Kapitel 5

Die parallele adaptive Verfeinerung

5.1 Bezeichnungen

Zuerst werden Abkürzungen für das gesamte Kapitel aufgeführt:
Man hat hier ein grobes und ein feines Gitter, bezeichnet mit G_H (grobes Gitter) und G_h(feines Gitter). Ferner ist ggff das Grobgitterfarbenfeld und fgff das Feingitterfarbenfeld.

Nun werden die Bezeichnungen für Abschnitt 5.2 vorgestellt:

Abstrakte Bezeichnungen für wichtige Variablen des Abschnitts 5.2

Man hat die folgenden Informationen nur lokal auf den Kernen. Von dieser Einschränkung wird im Folgenden abstrahiert. Diese abstrakten Informationen geben an, wie die Situation wäre, wenn man alle verteilten Informationen hätte. Der Index 'ab' steht für abstrakt.

Obwohl das nur als Arbeitsgrundlage gedacht ist, könnte man diese Informationen bekommen, wenn man die Eigenschaftspunkte in einen shared memory Speicher bringen würde und damit alle anderen 'abstrakten' Variablen berechnen würde.

Mit $EP_{ab}(t)$ sind die zum Zeitpunkt t durch das Verfeinerungskriterium bestimmten Punkte gemeint. Es sei als Punktmenge und Feld aufgefasst. $EP_{ab}(t)$ ist auf dem groben Gitter definiert.
Mit $P^+_{ab}(t)$ sind die zum Zeitschritt t hinzukommenden Punkte gemeint. $P^-_{ab}(t)$ sind die entfernten Punkte. Diese Mengen sind auf dem feinen Gitter definiert.

Konkrete Bezeichungen für wichtige Variablen des Abschnitts 5.2

Mit EP^a sind die aktuell auf dem Kern verfügbaren Informationen des Verfeinerungskriteriums gemeint. Sie sind als Punktmenge und markiertes Feld verfügbar. EP^v sind die entsprechenden Informationen des letzten Zeitschritts.

P^+_k ist die Menge der aktuell auf dem Kern verfügbaren Information von $P^+_{ab}(t)$, vgl. Definition 5.12. Mit P^-_k ist die Menge der entfernten Punkte auf dem Kern zum Zeitpunkt t gemeint (d.h. was der Kern über die Löschung von Punkten auf dem eigenen

feinen Gitter weiß, vgl. Definition 5.13).

Mit EP^+ und EP^- sind die Punktmengen gemeint, die angeben, wo das Verfeinerungskriterium nun erfüllt ist, bzw. nicht mehr erfüllt ist. Genauere Angaben dazu erfolgen in Abschnitt 5.3 in Definition 5.2.

5.2 Die Teilschritte zur parallelen adaptiven Verfeinerung

Dieser Abschnitt stellt eine Einleitung in dieses Kapitel dar.

5.2.1 Das abstrakte Prinzip dazu

Rein abstrakt betrachtet läuft die parallele adaptive Verfeinerung so ab: Man hat als Vorgabe $EP_{ab}(t)$ und $EP_{ab}(t\text{-}1)$. Daraus werden $P^+_{ab}(t)$ und $P^-_{ab}(t)$ berechnet, welche Punkte dazukommen bzw. zu entfernen sind. Im letzten Schritt werden mithilfe der Information $P^+_{ab}(t)$ und $P^-_{ab}(t)$ einerseits Punkte aus dem Verfeinerungsgitter entfernt bzw. dort hinzugefügt, und andererseits wird das Farbenfeld fgff aktualisiert. Damit ist dann das neue verteilte Verfeinerungsgitter korrekt.

5.2.2 Die konkreten Schritte hin zur parallelen adaptiven Verfeinerung

1. Man hat auf Kern k schon die EP^a-Information auf dem Gitter vorliegen. Durch einen Randaustausch liegt EP^a dann auch auf dem Außenrand vor.

2. Durch Verwendung von 2 Zeigern wird zu EP^a auch noch EP-vergangen gespeichert, bezeichnet mit EP^v. Nach jeder Ausführung der adaptiven Verfeinerung des Zeitschrittes t wird dann der Zeiger auf EP^a mit dem auf EP^v vertauscht, und umgekehrt, und EP^a dann jeweils zur Vorbereitung der nächsten Iteration gelöscht. Dieses Vertauschen wird hier *schieben* genannt, und es erfolgt sowohl in der Klasse *Eigenschaftspunkte* als auch in der Klasse *parallele adaptive Verfeinerung.*

   Dieses wird gemacht, um die **Änderungen** der EP-Strukturen zu erfassen, weil der Aufbau der kompletten Struktur viel zu viel Zeit in Anspruch nehmen würde.

3. Aus EP^a und EP^v werden EP^+ und EP^- bestimmt.

4. Mittels EP^+ und EP^- erhält man über die im Folgenden definierte Struktur *EP-SNB* die Mengen P^+_k und P^-_k. Diese Strukturen geben auf dem Gebiet zuzüglich dem Aussenrand an, wo Punkte entfernt werden bzw. dazukommen - entsprechend der verfügbaren EP-Informationen.

5. Am Ende erfolgen mit den gewonnenen Informationen der Punkteupdate und der Farbenupdate. Wie beides organisiert wird, wird in den Abschnitten 5.5 und 5.6 behandelt. Außerdem wird gegen Ende dieses Kapitels die Gesamtfunktion *tun* dargestellt, die alle nötigen Schritte in der richtigen Reihenfolge durchführt.

Abbildung 5.1: Nicht verfügbare EP^a-Informationen bei Kern P. Auf den umrandeten Kernen liegen gemeinsame Punkte vor.

AABB

Abbildung 5.2: Die Information EP^a reicht im Allgemeinen aus.

Es folgen Bemerkungen zu den einzelnen Schritten:

Bemerkung 1: Der Randaustausch von Schritt 1 erfolgt unter Übermittlung einzelner Bits, um die Kommunikationsbuffer des Randaustausches möglichst kurz zu haben. Dazu wurden die Bit-Pack-Klassen *bool-short-int* oder *bool-int* erstellt, die auch in der Klasse Lastausgleich verwendet werden.

Bemerkung 2: (zu Schritt 4 dieses Abschnitts) Die Strukturen P_k^+ und P_k^- sind, wenn sie auf EP^a und EP^v basieren, nicht für alle Kerne immer vollständig vorhanden, was man im folgenden Fall sieht, vgl. Abbildung 5.1: Das ist der Fall, wenn man einen echten Randpunkt auf Kern P hat, der keinen Grobgitterpunkt über sich hat, und der - bis auf die nicht existierenden Punkte - von Punkten auf anderen Kernen umgeben ist. Die Idee: Für den Kern liegt die Information EP^a dann dort nicht vor, obschon die Entwicklung des Aussenrandes von Interesse ist. Dieser Fall wird aber in den Strukturen zum stellvertretenden Einfügen beim Farbenupdate gehandhabt.

Bemerkung 3: (zu Schritt 4 dieses Abschnitts) Im Allgemeinen reicht die Information aber aus. Man betrachte diesen Fall: Vier Grobgitterpunkte liegen nebeneinander, die linken beiden Punkte auf Kern A, und die anderen 2 jeweils rechtsliegenden Punkte auf Kern B, vgl. Abbildung 5.2. Zunächst gilt: Nur der Punkt links erfüllt das Verfeinerungskriterium. Damit reicht das Gitter bis zum zweiten Punkt, der auf Kern A liegt. Jetzt kommt es zu einer Änderung: Wenn nun der am weitesten rechtsliegende Punkt das Kriterium erfüllt, ändert das den Außenrand des von Kern A nicht, weil nur der umgebende 3x3-Grobgitterbereich um den rechten Punkt geändert wird, vgl. den Abschitt 5.6.6. Die Verfeinerungsinformation auf dem Außenrand zu haben bringt also schon viel.

Bemerkung 4: Ansonsten werden alle Änderungen, bei denen P_k^+ und P_k^- wegen fehlender EP^a- bzw. EP^v-Informationen einer besonderen Behandlung bedürfen, im Abschnitt 5.6.1 über den Farbenupdate behandelt.

Bemerkung 5: Die Mengen P_k^+ und P_k^- werden nicht streng nach den Definitionen im Abschnitt 5.3 bestimmt, sondern auf viel effizientere Weise. Die Berechnungformeln werden mit den Symbolen P_b^+ und P_b^- verbunden, mit b für Berechnung. Es muss dann gezeigt werden, dass $P_k^+ = P_b^+$ und $P_k^- = P_b^-$ ist.

NW	N	NO
W	Z	O
SW	S	SO

Abbildung 5.3: Die erweiterten Himmelsrichtungen mit dem Zentrum in der Mitte

5.3 Vorbereitende Definitionen und Sätze

5.3.1 Definitionen

Zu den EP-Strukturen EP^a, EP^v, EP^+ und EP^-

Definition 5.1: Die Strukturen EP^a und EP^v wurden eben behandelt. Sie enthalten die aktuelle und vergangene Verfeinerungs-Informationen auf dem Gebiet und dem Außenrand.
Sie werden wie folgt bestimmt: Zuerst wird die Verfeinerungsinformation EP ermittelt. Dann erfolgt ein Randaustausch dieser Verfeinerungsinformation, wonach man EP^a bekommt. Dann erfolgt die parallele adaptive Verfeinerung. Mithilfe eines Zeigertausches zwischen EP^a und EP^v bekommt man die Information EP^v , die die jetzt vergangene EP-Information enthält. Vergangen nennt man diese Information, weil die parallele adaptive Verfeinerung schon durchgeführt wurde. Am Ende wird nur noch ein Reset der EP^a Information gemacht.
In der nächsten Iteration geht diese Vorgehensweise dann wieder von vorne los.

Definition 5.2: Die Änderungsmengen für die Verfeinerungsinformation EP lauten:

$EP^+ = EP^a$ - EP^v

und

$EP^- = EP^v$ - EP^a

Berechnet wird EP^+, indem die Punktliste zu EP^a durchlaufen wird, und geprüft wird, ob das Feld zu EP^v dort false ist. Analog wird EP^-berechnet, indem die Punktliste zu EP^v durchlaufen wird, und geprüft wird, ob das Feld zu EP^a dort false ist.

Als Vorbereitung zu EP-SNB: Die Strukturen bzgl. Himmelsrichtungen

Definition 5.3: Die Himmelsrichtungen aus Definition 4.2 werden erweitert durch

Zentrum(f,g) = (f,g), oder auch Z(f,g) = (f,g)

zu den erweiterten Himmelsrichtungen EHR.
Die Menge der erweiterten Himmelsrichtungen ist MEHR = MHR $\cup$ {Z}.

Die Abbildung 5.3 stellt alle erweiterten Himmelsrichtungen dar.

Partitionierung der 5 x 5 Überdeckungen mithilfe der Himmelsrichtungen

Vorbemerkung: 5 x 5 Überdeckungen sind sinnvoll:

1. Die Bildung von kleineren Überdeckungen wie 3 x 3 Gebieten machen numerisch keinen Sinn.
2. Man hat so an den äußeren Grenzen des Gebietes Linien, wo jeder 2.te Gitterpunkt einen Grobgitterpunkt über sich hat. Das ist z.B. für die Randinterpolation notwendig.

Definition 5.4: Ist (f,g) eine Koordinate auf dem groben Gitter, dann ist (f*,g*) = (2f,2g) die darunter liegende Koordinate.

Definition 5.5: Sei (f*,g*) der Wert zu (f,g) gemäß Definition 5.4. Es gibt zur Koordinate (f,g) vier Arten von Blöcken darunter:

1. $B_1(f,g) = \{(f^*,g^*)\}$
2. $B_2(f,g) = \{(f^*,g^*),(f^*+1,g^*)\}$
3. $B_3(f,g) = \{(f^*,g^*),(f^*,g^*+1)\}$
4. $B_4(f,g) = \{(f^*,g^*),(f^*+1,g^*),(f^*,g^*+1),(f^*+1,g^*+1)\}$

Damit kommen wir zur folgenden Menge, die eine 5 x 5 Überdeckung darstellt, und mit Himmelsrichtungen HR aufgeteilt wird.

Definition 5.6: Die Idee hier: Die 5 x 5 Überdeckung mithilfe der Himmelsrichtungen aufteilen wie in Abbildung 5.4a und 5.4b. P(EHR,f,g) nimmt die folgenden Werte an:

$P(EHR,f,g) = B_4(EHR(f,g))$ für $EHR \in \{NW, N, W, Z\}$
$P(EHR,f,g) = B_3(EHR(f,g))$ für $EHR \in \{SW, S\}$
$P(EHR,f,g) = B_2(EHR(f,g))$ für $EHR \in \{NO, O\}$
$P(EHR,f,g) = B_1(EHR(f,g))$ für $EHR \in \{SO\}$

In der Abbildung 5.4b wird die Aufteilung der Punkte nach Abbildung 5.4a entsprechend der in Definition 5.6 definierten Punktmengen dargestellt.

Definition 5.7: $P_{5,5}(f,g)$

$P_{5,5}(f,g)$ ist die Menge aller Punkte in Abbildung 5.4, d.h. offenbar gilt:
$P_{5,5}(f,g) = \bigcup_{EHR \in MEHR} P(EHR,f,g)$.

Das ist die Menge aller Verfeinerungspunkte unter einer Koordinate, wobei am echten Gitterrand hiervon Gitterpunkte wegfallen, wenn sie das grobe Gitter überragen, vgl. die Definition 5.10 und den Abschnitt 5.4.7. Denn offenbar können von den 5 x 5 Punkten am Gitterrand Punkte wegfallen.

a.

*	*	*	*	*	
*	*	*	*	*	
*	*	*	*	*	
*	*	*	*	*	
*	*	*	*	*	

b.

B_4	B_4	B_2
B_4	B_4	B_2
B_3	B_3	B_1

Abbildung 5.4: a. Die Mengen P(EHR,f,g) der erweiterten Himmelsrichtungen in Bezug auf den mittleren Punkt (f,g). b. Die Wahl der Mengen B_i in der jeweiligen erweiterten Himmelsrichtung.

Die EP-SNB-Strukturen *EP-SNB*, *EP-SNB-Zahl* und *EP-SNB-Punkte*

Nach den Vorarbeiten zu den Himmelsrichtungen werden nun die EP-SNB-Strukturen definiert. *EP-SNB* hat folgende Bedeutung: Zu einer Koordinate wird mit einer Menge, die durch eine Zahl abgebildet wird, die EP-Information aller Himmelsrichtungen $\in$ MHR erfasst. Es folgt:

Definition 5.8 *EP-SNB* und *EP-SNB-Zahl*:

$$\forall_{(f,g)\in G_h} \text{ EP-SNB (f,g)} = \bigcup_{HR\in MHR} \begin{cases} HR & \text{falls } EP^a(HR(f,g)) = true \text{ ist} \\ \emptyset & \text{sonst} \end{cases}$$

Damit ist gemeint, dass alle die Himmelsrichtungen in einer Menge zusammengefasst werden, für die die Punkte in diesen Richtungen die Verfeinerungseigenschaft erfüllen.

Mit Hilfe der den Himmensrichtungen zugeordneten Zahlen aus Definition 4.1 erhalten wir:

$$\forall_{(f,g)\in G_h} \text{ EP-SNB-Zahl (f,g)} = \sum_{HR\in EP-SNB(f,g)} HR - Zahl(HR)$$

Dabei wird jedes Bit gesetzt, zu dem der Punkt in der Himmelsrichtung die Verfeinerungseigenschaft erfüllt.

Definition 5.9 P_Z und *EP-SNB-Punkte*:

Der zentrale 2 x 2 Block P_Z ist definiert durch P_Z(f,g) = P(Z,f,g). Dann hat die Menge *EP-SNB-Punkte* den Zweck, dass ihre Überdeckung mit P_Z Mengen alle möglichen Punkte, die verschwinden könnten, einschließt. Sie wird bei der Bestimmung von *EP-SNB* gleich mitbestimmt. Es ist eine Grobgittermenge, und sie lautet:

$$\text{EP-SNB-Punkte} = \bigcup_{(f,g)\in EP^-} \bigcup_{HR\in EMHR(f,g)} \text{HR(f,g)}$$

Die Idee der Bildung dieser Menge ist die, dass man zur Bestimmung von P_k^- nur alle Punkte (f,g) aus EP-SNB-Punkte durchlaufen und die jeweils vier Punkte von P_Z(f,g) so reduzieren muss, dass mit der Vereinigung aller dieser Punkte genau P_k^- übrig bleibt.

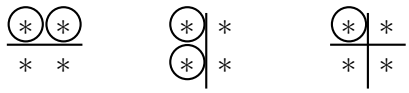


Abbildung 5.5: Die feinen Gebiete im Fall von echtem Osten und/oder Süden

Wie diese vier Punkte jeweils reduziert werden, wird in Abschnitt 5.3.2 gezeigt.

Um diese Idee zu realisieren, wird diese Eigenschaft von *EP-SNB-Punkte* genutzt:

Satz 5.1:
$$\bigcup_{(f,g)\in EP^-} \mathrm{P}_{5,5}(\mathrm{f,g}) \subseteq \bigcup_{(f,g)\in EP-SNB-Punkte} \mathrm{P}_Z(\mathrm{f,g})$$

Das Maximalgitter als notwendige Information zur Definition von $\mathbf{P}_k^+$ und $\mathbf{P}_k^-$

Bevor die Definitionen erfolgen, muss das maximale Verfeinerungsgitter definiert werden. Das ist das feine Gitter, das entsteht, wenn überall verfeinert würde. Man kann deswegen nicht einfach P_Z(f,g) über alle Grobgitterpunkte vereinigen, weil am rechten und unteren Rand dann etwas 'übersteht'. Die Grobgitterrandlinien begrenzen auch das feine Gitter.
Also: Das grobe Gitter definiert die maximalen Ausmaße des feinen Gitters. Wir definieren

Definition 5.10 Das maximale Gitter MaxG (auf einem Kern):

Vorbereitende Definition a:
echter_Osten(f,g) = true $\Leftrightarrow$ g = g_{max} oder sonst ggff(f,g+1) < 0
echter_Süden(f,g) = true $\Leftrightarrow$ f = f_{max} oder sonst ggff(f+1,g) < 0
Damit wird erkannt, ob das grobe Gitter rechts oder unten endet.

Vorbereitende Definition b:
$$\mathrm{B(f,g)} = \begin{cases} B_2(f,g) & \text{falls echter\_Osten(f,g) = true, echter\_Süden(f,g)=false} \\ B_3(f,g) & \text{falls echter\_Osten(f,g) = false, echter\_Süden(f,g)=true} \\ B_1(f,g) & \text{falls echter\_Osten(f,g) = true, echter\_Süden(f,g)=true} \\ B_4(f,g) & \text{sonst} \end{cases}$$
Die Idee dieser Definition: Im echten_Osten und/oder echten_Süden wird das feine Gebiet nicht durch eine Menge P_Z(f,g), sondern durch eine Teilmenge davon bestimmt, vgl. Abbildung 5.5.
Damit werden im Falle eines rechts endenden groben Gitters von P_Z die rechten zwei Punkte weggelassen. Endet das grobe Gitter unten, werden die unteren beiden Punkte von P_Z weggelassen. Und gilt beides, bleibt nur der gemeinsame Gitterpunkt von P_Z

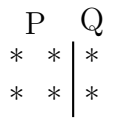


Abbildung 5.6: Die Ostgrenze von P

übrig.

Damit definieren wir das lokale Maximalgitter:

MaxG = $\bigcup_{(f,g)\in G_H}$ B(f,g)

Bemerkung: Da mit den lokalen Informationen (G_H und ggff) MaxG erkannt werden kann, ist

MaxG-global = $\bigcup_{k\in Kerne}$ MaxG(k), d.h. die globale Situation wird lokal erkannt.

$\mathbf{P_k^+}$und $\mathbf{P_k^-}$und $\mathbf{P_b^+}$und $\mathbf{P_b^-}$

Sei P_{ab}(t) die Punktemenge auf dem feinen Gitter, und es ist P_{ab}^+ = P_{ab}(t) - P_{ab}(t-1) (mit P_{ab}(-1) = leere Menge), und P_k^-= P_{ab}(t-1) - P_{ab}(t). Diese Definitionen sind aber global, und damit für die einzelnen Kerne nicht verfügbar. Daher definieren wir erst einmal lokal mit k als ID des Kernes:

Definition 5.11
P_k(t) = ($\bigcup_{(f,g)\in EP^a}$ $P_{5,5}$(f,g)) ∩ MaxG(t) und
P_k(t-1) = ($\bigcup_{(f,g)\in EP^v}$ $P_{5,5}$(f,g)) ∩ MaxG(t-1)

Das sind die auf die Kerne bezogenen Punktmengen. Sie ergänzen sich exakt, d.h. es entstehen keine Übergangs-Lücken, was man sich an einer Gittergrenze im Osten klar machen kann, vgl. Abbildung 5.6. Ist dort ein Übergang zu einem anderen Kern, dann ist echter Osten(f,g) = false, und es werden von P_Z keine östlichen Punkte entfernt, vgl. Definition 5.10.
Also: Sowohl bei der Lastmigration als auch bei der parallelen adaptiven Verfeinerung sind die obigen Punktmengen P_k(t) und P_k(t-1) korrekt.
Damit können wir die Punktmengen, die lokal hinzukommen bzw. zu entfernen sind, definieren durch

Definition 5.12 P_k^+:
$P_k^+ = ((\bigcup_{(f,g)\in EP^a} P_{5,5}(f,g)) - (\bigcup_{(f,g)\in EP^v} P_{5,5}(f,g))) \cap MaxG\ (t)$

Idee: Man hat die durch Überdeckungen bestimmte aktuelle Punktmenge ∩ MaxG.

Definition 5.13 P_k^-:
$P_k^- = ((\bigcup_{(f,g)\in EP^v} P_{5,5}(f,g)) - (\bigcup_{(f,g)\in EP^a} P_{5,5}(f,g))) \cap MaxG\ (t\text{-}1)$

Das stimmt wegen $P_k^+ = P_k(t) - P_k(t\text{-}1)$, und $P_k^- = P_k(t\text{-}1) - P_k(t)$.
Außerdem gilt $P_k(t) = P_k(t\text{-}1) \cup P_k^+ - P_k^-$.

Das sind die Mengen, die als Grundlage für den Punkte- und Farbenupdate dienen. Sie wären aufwendig zu berechnen, und sie würden insbesondere auch die sequentielle Laufzeit erhöhen. Berechnet werden an Stelle dessen P_b^+ und P_b^-, wie im Folgenden definiert. Natürlich muss dann, wie es in Abschnitt 5.3.2 geschieht, die Identität von P_k^+ und P_b^+ sowie P_k^- und P_b^- bewiesen werden. Deswegen können sie hier einfach definiert werden:

Definition 5.14 Die Punktmenge P_b^+ auf dem feinen Gitter:
$P_b^+ = (\bigcup_{(f,g)\in EP^+} (P_Z(f,g) \bigcup_{HR\in MHR-EP-SNB(f,g)} P(HR,f,g)) - (\bigcup_{(f,g)\in EP^v} P_{5,5}(f,g)))) \cap MaxG$

Für die Definition von P_b^- braucht man noch diese Definition:

Definition 5.15 Die Punktmenge M^a auf dem groben Gitter:

M^a = EP-SNB-Punkte - EP^a

Definition 5.16 Die Punktmenge P_b^- auf dem feinen Gitter:
$P_b^- = (\bigcup_{(f,g)\in M^a} (P_Z(f,g) - \bigcup_{HR\in EP-SNB(f,g)} P_{5,5}(HR(f,g))) \cap G_h^{alt}$
Dabei ist G_h^{alt} das Gitter des vorherigen Zeitschrittes t-1.

5.3.2 Vorbereitende Sätze

Satz 5.2

Es gilt stets: (A ∩ B) - C = (A - C) ∩ B

Beweis: x ∈ rechter Seite oder linker Seite ⇔

x ∈ A und x ∈ B und x ∉ C

Satz 5.3 Vorbereiten zum Satz 5.10

$(G_h^{alt}$ - $\bigcup_{(f,g)\in EP^a} P_{5,5}(f,g)) = P_k^-$

Beweis:
$G_h^{alt} = (\bigcup_{(f,g)\in EP^v} P_{5,5}(f,g)) \cap MaxG \Rightarrow$
G_h^{alt} - $\bigcup_{(f,g)\in EP^a} P_{5,5}(f,g) = (\bigcup_{(f,g)\in EP^v} P_{5,5}(f,g)) \cap MaxG$ - $\bigcup_{(f,g)\in EP^a} P_{5,5}(f,g)$, und weil wegen Satz 5.2 '∩ MaxG' und '- $\bigcup_{(f,g)\in EP^a} P_{5,5}(f,g)$' vertauscht werden können, folgt die Behauptung aus der Definition von P_k^-.

Satz 5.4: Direkte Umformung von P_k^+und P_k^-

$P_k^+ = ((\bigcup_{(f,g)\in EP^+} P_{5,5}(f,g))$ - $(\bigcup_{(f,g)\in EP^v} P_{5,5}(f,g))) \cap MaxG$ (t)

$P_k^- = ((\bigcup_{(f,g)\in EP^-} P_{5,5}(f,g))$ - $(\bigcup_{(f,g)\in EP^a} P_{5,5}(f,g))) \cap MaxG$ (t-1)

Beweis: Beide Formeln entsprechen einander, nur mit t und t-1 vertauscht, was auch EP^+ mit EP^-, EP^a mit EP^v und P_k^+ mit P_k^- vertauscht. Deswegen ist nur ein Beweis nötig.
Ohne Einschränkung bringen wir den Beweis für P_k^+. In der Formel zu P_k^+ fallen die gemeinsamen $P_{5,5}(f,g)$ für (f,g) in $EP^v \cap EP^a$ weg, weil die Mengendifferenz diese Punkte entfernt. Deswegen kann man statt EP^a EP^+ nehmen.

Satz 5.5: Vorbereiten von $P_k^+ = P_b^+$

$$\bigcup_{(f,g)\in EP^+} P_{5,5}(f,g) - \bigcup_{(f,g)\in EP^v} P_{5,5}(f,g) =$$
$$\bigcup_{(f,g)\in EP^+} (P_Z(f,g) \bigcup_{HR\in MHR-EP-SNB(f,g)} P(HR,f,g)) - \bigcup_{(f,g)\in EP^v} P_{5,5}(f,g)$$

Beweis: Die Idee bei der Darstellung der rechten Seite ist die, dass die in den Himmelsrichtungen aus *EP-SNB* liegenden Blöcke durch P_Z von geeigneten Koordinaten aus EP^+überdeckt werden. Deswegen müssen nur die nicht in den Himmelsrichtungen aus *EP-SNB* liegenden Blöcke wirklich in die Vereinigungsmenge der rechten Seite einbezogen werden.
Zu $\supseteq$: Da sich die 5 x 5 Überdeckungen aus den 9 Blöcken zusammensetzen, folgt aus den Definitionen:

$$\bigcup_{(f,g)\in EP^+} P_{5,5}(f,g) = \bigcup_{(f,g)\in EP^+} (P_Z(f,g) \bigcup_{HR\in MHR} P(HR,f,g)) \Rightarrow$$

$$\bigcup_{(f,g)\in EP^+} P_{5,5}(f,g) - \bigcup_{(f,g)\in EP^v} P_{5,5}(f,g) =$$

$$\bigcup_{(f,g)\in EP^+} (P_Z(f,g) \bigcup_{HR\in MHR} P(HR,f,g)) - \bigcup_{(f,g)\in EP^v} P_{5,5}(f,g)(*)$$

Wegen MHR - EP-SNB(f,g) $\subseteqq$ MHR ist

$$\bigcup_{HR\in MHR-EP-SNB(f,g)} P(HR,f,g) \subseteqq \bigcup_{HR\in MHR} P(HR,f,g)$$

Und damit ist die rechte Seite in der Gleichung aus der Behauptung enthalten in der rechten Seite von (*), die wegen (*) gleich der linken Seite der Gleichung ist.

Zu $\subseteqq$: Sei nun (k,l) aus der Menge auf der linken Seite:
Dann ist (k,l) $\in P_{5,5}(f,g)$ mit (f,g) $\in EP^+$.

Es folgt aus der Definition von $P_{5,5}$ in Definition 5.7:
$\exists$ EHR $\in$ MEHR mit (k,l) $\in$ P(EHR,f,g).

Ist nun EHR = Z oder EHR $\notin$ *EP-SNB*(f,g) $\Rightarrow$ (k,l) ist in der Menge der rechten Seite der Formel.

Daher sei EHR $\in$ *EP-SNB*(f,g). Damit folgt per Definition von P(EHR,f,g) und P_Z(f,g), dass

(k,l) $\in$ P(EHR,f,g) $\subseteq P_Z$ (EHR(f,g)).

Wegen der Definition von EP-SNB ist P_Z (EHR(f,g)) = P_Z (p,q) mit (p,q) $\in EP^a$, also zusammen ist (k,l) $\in P_Z$(p,q) mit (p,q) $\in EP^a$.

Nun ist (p,q) $\in EP^+$, denn sonst wäre wegen $EP^+ = EP^a$ - EP^v und (p,q) $\in EP^a$ (p,q) $\in EP^v$, weshalb (k,l) nicht in der Formel der linken Seite der Behauptung sein kann, was ein Widerspruch ist zu der Anfangsannahme, dass (k,l) aus der Menge der linken Seite ist. Wegen (p,q) $\in EP^+$ und (k,l) $\in P_Z$(p,q) $\subseteqq \bigcup_{(f,g)\in EP^+} P_Z$(f,g) und wegen (k,l) $\notin \bigcup_{(f,g)\in EP^v} P_{5,5}$(f,g) , da es in der Menge auf der linken Seite der Behauptung liegt,

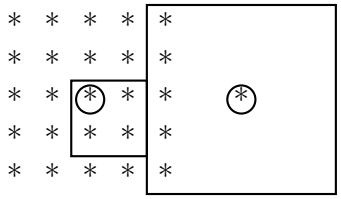


Abbildung 5.7: Der Schnitt von P_Z(f,g) mit $P_{5,5}$(p,q) mit $|(f,g)-(p,q)|_\infty$=2. Der linke umrandete Punkt ist (f,g), der rechte umrandete Punkt (p,q). Das kleine Quadrat markiert P_Z(f,g), das große Quadrat $P_{5,5}$(p,q).

liegt (k,l) in der Formel der rechten Seite der Behauptung, was zu zeigen war.

Definition 5.17 Unendlich Norm: $|(f,g)|_\infty = |\max(f,g)|$

Satz 5.6 Erstes Vorbereiten zu $P_k^- = P_b^-$:

Es gilt:

$$\forall_{(f,g)\in G_H} \quad P_Z(f,g) \text{ - } \bigcup_{HR\in EP-SNB(f,g)} P_{5,5}(HR(f,g)) =$$

$$P_Z(f,g) \text{ - } \bigcup_{(f',g')\in EP^a-\{(f,g)\}} P_{5,5}(f',g')$$

Insbesondere ist die rechte Seite der 2.ten Zeile noch abhängig von (f,g).

Beweis: Sei (f,g) $\in G_H$ ausgewählt. Es seien definiert $M^k = \{(r,s)||(f,g)-(r,s)|_\infty = k\}$ und $N^k = M^k \cap (EP^a\text{-}(f,g))$. N^k ist die Menge der Punkte aus EP^a-(f,g) mit dem Abstand k zu (f,g) bezüglich der 'unendlich Norm'. Dann ist $N^0 = \emptyset$, und nach Definition 5.8 ist N^1 = (EP-SNB(f,g))(f,g)(*). Denn es gilt:
$HR \in$ *EP-SNB(f,g)* $\Leftrightarrow$ HR(f,g) $\in M^1$ und EP(HR(f,g)) = true $\Leftrightarrow$ HR(f,g) $\in N^1$.

Nun gilt: P_Z(f,g) - $\bigcup_{(p,q)\in N^{\geq 2}} P_{5,5}(p,q) = P_Z(f,g)$ (**). Denn bei einem Grobgitterabstand von (p,q) von 2 reicht P_Z nicht weit genug, um $P_{5,5}$(p,q) zu schneiden, vgl. auch Abbildung 5.7.

Damit folgt:
$P_Z(f,g)$ - $\bigcup_{(f',g')\in EP^a-\{(f,g)\}} P_{5,5}(f',g') =$

$P_Z(f,g)$ - $\bigcup_{(p,q)\in N^1} P_{5,5}(p,q)$ - $\bigcup_{(p,q)\in N^{\geq 2}} P_{5,5}(p,q)$
Diese Gleichung gilt wegen EP^a-(f,g) = $N^0 \cap N^1 \cap N^{\geq 2} = N^1 \cap N^{\geq 2}$

$= P_Z(f,g)$ - $\bigcup_{HR\in EP-SNB(f,g)} P_{5,5}(HR(f,g))$ - $\bigcup_{(p,q)\in N^{\geq 2}} P_{5,5}(p,q)$ (wegen *)

$= P_Z(f,g)$ - $\bigcup_{(p,q)\in N^{\geq 2}} P_{5,5}(p,q)$ - $\bigcup_{HR\in EP-SNB(f,g)} P_{5,5}(HR(f,g))$

$= P_Z(f,g)$ - $\bigcup_{HR \in EP-SNB(f,g)}$ $P_{5,5}$(HR(f,g)) (wegen **)

Satz 5.7 Zweites Vorbereiten zu $P_k^- = P_b^-$:

Es gilt:

$$\forall_{(f,g)\in G_H,(f,g)\notin EP^a} \quad P_Z(f,g) - \bigcup_{HR\in EP-SNB(f,g)} P_{5,5}\ (\mathrm{HR(f,g)}) =$$

$$P_Z(f,g) - \bigcup_{(f',g')\in EP^a} P_{5,5}\ (\mathrm{f',g'})$$

Insbesondere ist die rechte Seite der 2.ten Zeile unabhängig von (f,g).

Beweis: Es gilt die Behauptung von Satz 5.6.

Wegen (f,g)$\notin$ EP^a ist aber EP^a - {(f,g)} = EP^a, woraus durch Ersetzen von EP^a durch EP^a - {(f,g)} in der rechten Seite der Behauptung des Satzes 5.6 diese Behauptung folgt.

Satz 5.8 Drittes Vorbereiten zu $P_k^- = P_b^-$:

$P_k^- \cap \bigcup_{(f,g)\in M^a} P_Z(\mathrm{f,g}) = P_k^-$

Beweis: Zu zeigen ist:

$P_k^- \subseteq \bigcup_{(f,g)\in M^a} P_Z(\mathrm{f,g})$

Wegen Satz 5.4 reicht zu zeigen:
$((\bigcup_{(f,g)\in EP^-} P_{5,5}(\mathrm{f,g})) - (\bigcup_{(f,g)\in EP^a} P_{5,5}(\mathrm{f,g}))) \subseteq \bigcup_{(f,g)\in M^a} P_Z(\mathrm{f,g})$
Sei nun (k,l) aus der Menge der linken Seite, d.h. nach Definition 5.7 existiert (r,s) $\in$ EP^-und EHR $\in$ MEHR mit (k,l) $\in$ P(EHR,r,s), und (k,l) $\notin \bigcup_{(f,g)\in EP^a} P_{5,5}(\mathrm{f,g})$

Sei nun (p,q) = EHR(r,s). Dann ist wegen Definition 5.8 (p,q) $\in$ *EP-SNB-Punkte*, und es ist (k,l) $\in$ P(EHR,r,s) $\subseteqq$ P_Z(p,q).

Wegen (k,l) $\notin \bigcup_{(f,g)\in EP^a} P_{5,5}(\mathrm{f,g})$ist (p,q)$\notin$ EP^a.(*)
Zu (*): (k,l) $\in$ P_Z(p,q) - $\bigcup_{(f,g)\in EP^a} P_{5,5}(\mathrm{f,g})$. Wäre (p,q)$\in$ EP^a, dann wäre diese Menge leer, und (k,l) würde nicht existieren. Ende zum Beweis zu (*).

Damit ist (k,l) $\in \bigcup_{(f,g)\in M^a} P_Z(\mathrm{f,g})$ wegen (k,l) $\in$ P_Z(p,q) und (p,q)$\in$ M^a nach Definition 5.15 wegen (p,q) $\in$ *EP-SNB-Punkte* und (p,q) $\notin$ EP^a. qed.

5.3.3 Die 2 Hauptsätze

Die Beweise von Satz 5.9 und 5.10 folgen direkt aus vorstehenden Definitionen und Sätzen:

Satz 5.9 $P_k^+ = P_b^+$

$P_k^+ = ((\bigcup_{(f,g)\in EP^+} P_{5,5}(f,g)) - (\bigcup_{(f,g)\in EP^v} P_{5,5}(f,g))) \cap MaxG(t)$ (wegen Satz 5.4)

$= (\bigcup_{(f,g)\in EP^+} (P_Z(f,g) \bigcup_{HR\notin EP-SNB(f,g)} P(HR,f,g)) - \bigcup_{(f,g)\in EP^v} P_{5,5}(f,g)) \cap MaxG(t)$
(nach Satz 5.5)

$= P_b^+$ (per Definition 5.14)

Satz 5.10 $P_k^- = P_b^-$

$P_b^- =$ (nach Definition 5.16)
$(\bigcup_{(f,g)\in M^a} (P_Z(f,g) - \bigcup_{HR\in EP-SNB(f,g)} P_{5,5}(HR(f,g)))) \cap G_h^{alt} =$

(nach Satz 5.7, dessen Voraussetzung erfüllt ist, weil nach Definition 5.15
$(f,g) \in M^a \notin EP^a$ist)

$(\bigcup_{(f,g)\in M^a} (P_Z(f,g) - \bigcup_{(f,g)\in EP^a} P_{5,5}(f,g))) \cap G_h^{alt} =$

(wegen Satz 5.2)
$\bigcup_{(f,g)\in M^a} P_Z(f,g) \cap (G_h^{alt} - \bigcup_{(f,g)\in EP^a} P_{5,5}(f,g)) =$ (wegen Satz 5.3)

$\bigcup_{(f,g)\in M^a} P_Z(f,g) \cap P_k^-$

$= P_k^-$ (wegen Satz 5.8)

5.4 Die Algorithmen für die Größen aus Teil 5.3

5.4.1 Was in diesem Abschnitt 5.4 gemacht wird

Nach den Definitionen und Sätzen ist nun zu zeigen, wie die in Abschnitt 5.3 behandelten Punktmengen algorithmisch berechnet werden sollen. Es muss auch begründet werden, dass das Berechnete übereinstimmt mit den Definitionen. Das gilt insbesondere bei der Berechnung von P_k^+ und P_k^-.

5.4.2 Die Reihenfolge der Abarbeitung der Algorithmen

Zuerst geht es um die Listen EP^a und EP^v. Damit werden EP^+ und EP^- bestimmt. Es schließt sich die Berechnung von *EP-SNB* und *EP-SNB-Punkte* an. Und am Ende

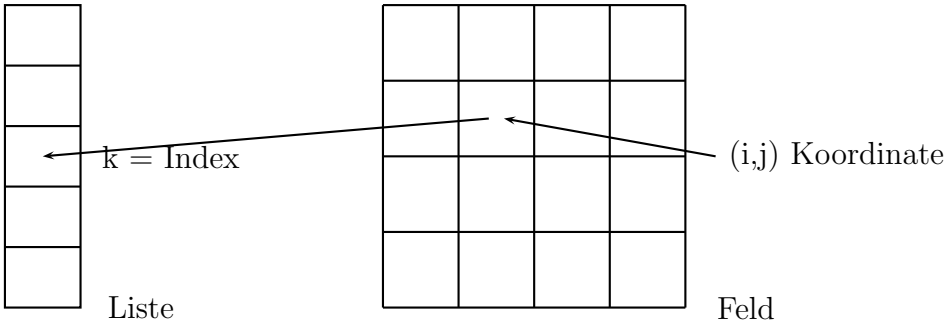


Abbildung 5.8: Die zweidimensionale *PunktFeldliste*. Auf der Koordinate (i,j) wird im Feld abgelesen, unter welchem Index k der Punkt in der Liste verzeichnet ist.

werden P_k^+ und P_k^- berechnet.

Insgesamt werden die einzelnen Schritte im Algorithmus *tun* in Abschnitt 5.8 dargestellt. Es geht dabei vor allem um Aufrufe von Funktionen mit großer Wirkung. An dieser Stelle kann dieser Algorithmus aber noch nicht präsentiert werden, weil die einzelnen Schritte dafür noch nicht ausgeführt worden sind.

5.4.3 Die *PunktFeldliste*

Die Klasse *PunktFeldliste* wird bei der Bildung der Strukturen dieses Abschnitts vielfach gebraucht, und wird daher zu Beginn vorgestellt.

Zur Leistung dieser Liste:

Die *Punktfeldliste* ist eine Klasse, bei der man Gitterpunkte in Zeit O(1) einfügen und entfernen kann, und alle Punkte in Zeit O(Anzahl Punkte) löschen kann, bzw. in Zeit O(Anzahl Punkte) durchlaufen kann.

Der Aufbau dieser Liste

Man hat dazu zwei Strukturen: Einmal eine eindimensionale Liste für die Koordinatenpaare (f,g), mit der Angabe der Anzahl der eingetragenen Punkte.
Und dann ein zweidimensionales Feld, mit Verweisen, unter welchem Index die Koordinaten in der eindimensionalen Liste eingetragen sind.
Für eine Darstellung der Liste vergleiche man Abbildung 5.8.

Die Funktionsweise

Einfügen eines Elementes in O(1): Die Koordinaten werden hinten in die Liste eingefügt. Der Index zu der Liste wird in das zweidimensionale Feld unter den Koordinaten eingetragen.

Löschen eines Elementes in O(1): Zuerst wird der Index auf das eindimensionale Feld im zweidimensionalen Feld gelesen. Dort wird standardmäßig gelöscht, indem das letzte Listenelement dahin kopiert und die Listenlänge um eines verringert wird. Dann wird unter der Koordinate im zweidimensionalen Feld gelöscht. Außerdem wird der Eintrag

des umkopierten Elementes im Feld auf die neue Position abgeändert.

Für das komplette Löschen in O(Anzahl Punkte) durchläuft man die Liste und löscht die Einträge unter den Koordinaten im zweidimensionalen Feld. Anschließend setzt man die Anzahl für die eindimensionale Liste in einem Schritt auf 0.

Bemerkung: Diese Liste erfüllt denselben Zweck wie die Feldliste, braucht aber weniger Speicherplatz, und die Zugriffe gehen schneller.

5.4.4 Die Algorithmen zu EP^a und EP^v

Die ganze Abarbeitung bis hin zu der Berechnung von P_k^+ und P_k^- beginnt mit den Strukturen EP^a und EP^v. Wie EP^a und EP^v bestimmt werden, einschließlich des Randaustausches, wurde im Abschnitt 5.2 besprochen.

Damit diese Informationen als Ursprung der nachfolgenden Schritte gebraucht werden können, müssen sie korrekt sein. Daher gilt:
Zwischen adaptiver Verfeinerung und dem Lastausgleich wird EP^v gelöscht und EP^v und EP^a getauscht ('schieben'). Die vergangene Information muss bei der Migration von Last übermittelt werden. Es muss nur diese Information kommuniziert werden, weil in der nächsten Iteration EP^a neu berechnet wird.
Alles Weitere zur Migration von EP^v wird im Kapitel 6 ausgeführt.

5.4.5 Zur Bestimmung der Größen EP^+ und EP^-

Diese beiden Objekte werden als *PunktFeldliste* gespeichert. Die Bestimmung von EP^+ und EP^- erfolgt so: Zuerst werden die Punkte aus *pfl_feld_aktuell* durchlaufen und geprüft, welche nicht in *pfl_feld_vergangen* vorhanden sind. Diese werden in EP^+ aufgenommen. Durch dieselben Schritte mit vertauschten pfl-Strukturen wird EP^- bestimmt. Wichtig hierbei: Da die Punkte als Liste vorliegen, werden auch schon gelöschte Punkte noch durchlaufen. Dadurch wird erreicht, dass das rekursive Entfernen aus Abschnitt 5.9 automatisch geschieht, weil die in der Gitterhierarchie tiefer liegenden Punkte noch in *pfl_feld_vergangen* vorhanden sind, wenn sie die Eigenschaft erfüllt haben. Da sie nicht mehr in *pfl_feld_aktuell* liegen, werden sie in EP^- eingefügt.

5.4.6 Die Algorithmen zu *EP-SNB* und *EP-SNB-Punkte*

Vorbemerkung zur Änderung dieser Strukturen bei der Migration: Es muss in den Ebenen, wo die Eigenschaftspunkte bestimmt werden, bei dem Lasttransport die *EP-SNB* Information geändert werden: Das geschieht in Algorithmus 5.2, wenn ein neuer Punkt hinzu kommt und nicht weiterverschickt wird, und in Algorithmus 5.3, falls ein Punkt entfernt wird. Alles weitere dazu in Kapitel 6.

Bemerkung: Man muss eine Veränderung der *EP-SNB*-Informationen auch im Zusammenhang mit dem Löschen von Punkten bei der parallelen adaptiven Verfeinerung eine Ebene höher durchführen. Das zu ändernde Gitter ist damit auf der Ebene der *EP-SNB* Information. Diese Veränderung geschieht automatisch wegen dem rekursiven

Entfernen, beschrieben in Abschnitt 5.9, in Kombination mit Algorithmus 5.1, weil die gelöschten Punkte in EP^- eingefügt werden.

Die Datenstrukturen

Die Datenstruktur für *EP-SNB* ist ein einfaches zweidimensionales Feld, wo in jedem Koordinatenpaar die *EP-SNB-Zahl* steht. Die Datenstruktur für *EP-SNB-Punkte* ist eine einfache Liste (ein Array mit der Anzahl Einträge). Wegen einem Reset der Punktmenge *EP-SNB-Punkte* zu Beginn von *EP-SNB-Update.tun*, also Algorithmus 5.1, ist ein Entfernen einzelner Koordinaten bei dieser Struktur nicht notwendig.

Die Aufruffunktion *tun* in *EP_SNB_Update*

Die Updatefunktion für EP^+ und EP^- lautet *tun*, und ist in der Klasse *EP_SNB_Update* definiert. Sie verwendet die in der Klasse *SNBUpdate erklärten* und nach dort vererbten Funktionen '*hinzu* für *EP-SNB*' und '*hinweg* für *EP-SNB*' sowie die Punktmenge *EP-SNB-Punkte*:

Algorithmus 5.1: *tun* (der Klasse *EP-SNB-Update*)

1. reset von *EP-SNB-Punkte*
2. **for all** (f,g) $\in EP^+$
3. *hinzu_für*_EP-SNB(f,g) (Alg. 5.2)
4. **end for**
5. **for all** (f,g) $\in EP^-$
6. *hinweg_für*_EP-SNB_*und_einfügen_für*_EP-SNB-Punkte(f,g) (Alg. 5.3)
7. **end for**

Die Funktion *hinzu* der Klasse *SNB-Update*

Diese Funktion ist von der Vaterklasse *SNB-Update* vererbt und lautet wie folgt:

Algorithmus 5.2: hinzu_*für*_EP-SNB(f,g)

1. **for all** HR $\in$ MHR
2. *EP-SNB-Zahl*(HR(f,g)) = *EP-SNB-Zahl*(HR(f,g)) **OR** HR-Zahl(HR^{-1})
3. **end for**

Erläuterung: Wird auf der Koordinate (f,g) ein Punkt mit der Verfeinerungseigenschaft hinzugefügt, ändern sich in allen Himmelsrichtungen die *EP-SNB*-Strukturen, indem sie ihr Bit in diese Richtung auf *true* ändern. Die *EP-SNB*-Information für (f,g) ändert sich hingegen nicht, weil dieser Punkt (f,g) durch *EP-SNB*(f,g) nicht erfasst wird.

Hinweg der Klasse *SNB-Update*

Diese Funktion ist von der Vaterklasse *SNB-Update* vererbt und lautet wie folgt:

Algorithmus 5.3: *hinweg_für*_EP-SNB_*und_einfügen_für*_EP-SNB-Punkte(f,g)

1. *EP-SNB-Punkte.hinzu*(f,g)
2. **for all** HR $\in$ MHR
3. *EP-SNB-Zahl*(HR(f,g)) = *EP-SNB-Zahl*(HR(f,g)) **AND** HR-Zahl($\overline{HR^{-1}}$)
4. *EP-SNB-Punkte.hinzu*(HR(f,g))
5. **end for**

Erläuterung: Wird auf der Koordinate (f,g) ein Punkt mit der Verfeinerungseigenschaft entfernt, ändern sich in allen Himmelsrichtungen die *EP-SNB*-Strukturen, indem sie ihr Bit in diese Richtung auf *false* ändern. Die *EP-SNB*-Information für (f,g) ändert sich hingegen nicht, weil dieser Punkt (f,g) durch *EP-SNB*(f,g) nicht erfasst wird.

Bemerkung: In den Schritten 1 und 4 wird *EP-SNB-Punkte.hinzu* ausgeführt, weil hier diese Punktmenge aufgebaut wird.

Korrektheit

Die Datenstruktur *EP-SNB* wird durch Algorithmus 5.1 immer aktuell gehalten. Wie dabei die Funktionen '*hinzu* für *EP-SNB*' und '*hinweg* für *EP-SNB* und einfügen für *EP-SNB-Punkte*' funktionieren, wurde erläutert. Für die Menge *EP-SNB-Punkte* wird genau die Menge EP^-durchlaufen, und die Punkte in allen erweiterten Himmelsrichtungen eingefügt, wie es nach Definition 5.9 sein soll.

Ein Aktualisierungsschritt von *EP-SNB* via EP^+ und EP^- korrigiert die Strukturen *EP-SNB* und *EP-SNB-Punkte*. Nur die Migrationsänderungen müssen noch berücksichtigt werden. Dafür werden auch die Funktionen '*hinzu* für *EP-SNB*' und '*hinweg* für *EP-SNB* und *EP-SNB-Punkte*' verwendet. Bei letzterer Funktion werden die Schritte '*hinweg* für *EP-SNB*' und '*EP-SNB-Punkte*' in einer Funktion kombiniert.

Bemerkung: *EP-SNB* ist korrekt auf G_H. Auf dem Außenrand ist dies nicht immer der Fall, weil EP(f,g) = true Einträge im Abstand ≥ 2 zu G_H nicht vorliegen. Das muss bei Beweisen des Feingitterfarbenfeldupdates unter Verwendung von P_k^+ und P_k^- berücksichtigt werden, vgl. den Abschnitt 5.6.6.

5.4.7 Verwendung der Menge MaxG als Vorbereitung zur P_k^+-Berechnung

Es gibt 2 Möglichkeiten zu verhindern, dass Punkte außerhalb von MaxG eingefügt werden. Sie werden zuerst vorgestellt, und dann wird diskutiert, welche Wahl die bessere ist.

Wie erreicht wird, dass nur MaxG-Punkte bei der P_k^+-Berechnung aufgenommen werden - Möglichkeit 1

Die erste Idee besteht darin, gemäß Definition 5.10 die Punkte nicht aufzunehmen, die nicht zu MaxG gehören. Dazu wird dieser Algorithmus *P^+-hinzu-erfüllt-MaxG* von dem Algorithmus *P^+-hinzu* aufgerufen.

Der Algorithmus erfolgt vom Prinzip her, indem die 5 x 5 Überdeckungen an den Rändern abgeschnitten werden, vgl. den Algorithmus C.1 im Anhang.

Wie erreicht wird, dass nur MaxG-Punkte bei der P_k^+-Berechnung aufgenommen werden - Möglichkeit 2

Die Idee: Es wird die Menge EP^a geändert. Randpunkte werden ins Innere gezogen. Ist z.B. (f,g) $\in EP^a$ und *echter_Osten*(f,g) = *true*, dann wird (f,g) aus EP^a entfernt, und (f,g-1) in EP^a eingefügt. Ist z.B. *echter_Osten*(f,g) = *echter_Süden*(f,g) = *true*, dann wird (f,g) aus EP^a entfernt, und (f-1,g-1) in EP^a eingefügt. Das gilt für alle 8 Fälle in der unmittelbaren Nachbarschaft.

Bemerkung 1: Um es exakt zu formulieren, muss auch *echter_Norden* und *echter_Westen* definiert sein.
echter_Westen(f,g) = *true* $\Leftrightarrow$ f = 0 **OR** ggff(f-1,g) < 0.
echter_Norden(f,g) = *true* $\Leftrightarrow$ g = 0 **OR** ggff(f,g-1) < 0.

Bemerkung 2: Der genaue Algorithmus dazu wird nicht angegeben. Er ist aus der Idee abzuleiten.
Aber: Bei der Selbstadaptivität werden zu jedem Punkt aus EP^a auch die benachbarten Punkte gewählt. In diesem Fall muss EP^a anschließend nur für echte Randpunkte gelöscht werden. Um die echten Randpunkte zu erkennen wird der Rand durchlaufen, und für den jeweiligen Punkt (f,g) wird geprüft, ob er am absoluten Gitterrand liegt oder für ein HR $\in$ *Selbstnachbarn*(f,g) ggff(HR(f,g)) = -1 ist.

So wird diese Methode realisiert bei der Simulation

Es reicht hier aus, die Einträge in *feld-aktuell* für echte Randpunkte auf *false* zu setzen. Bei Rechtecken hat man immer bei Randpunkten auch die Punkte zum Inneren hin eingefügt. Eine Ausnahme tritt auf, wenn isolierte Eigenschaftspunkte ganz am Rand liegen, weil vom Rechteck nur die Randlinie übriggeblieben ist, vgl. Abbildung 5.9. Dieser Fall tritt nur für eine Iteration auf. Außerdem spricht bei einer Simulation nichts dagegen, den Fall mit einer isolierten Randlinie einfach zu überspringen. Bei großen Gittern tritt dieser Fall erst nach sehr vielen Iterationen auf. Da außerdem eine Randlinie kaum Gitterpunkte hat, spielt das für Messungen, um die es bei der Simulation geht, auch gar keine Rolle.

So wird diese Methode realisiert bei der selbstadaptiven Verfeinerung

Bei der selbstadaptiven Verfeinerung treten Löcher auf im Feld *feld-aktuell*, weil das Kriterium nur für gemeinsame Gitterpunkte berechnet werden kann. Um das zu kompensieren, werden für jeden Gitterpunkt, der die Verfeinerungseigenschaft erfüllt, auch

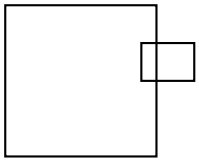


Abbildung 5.9: Ein Beispiel bei der Simulation: Vom Verfeinerungsrechteck bleibt nur die Randlinie übrig. Das Quadrat zeigt dabei den maximal verfeinerbaren Bereich auf.

die 8 umgebenden Gitterpunkte mit dieser Eigenschaft versehen. Damit gilt für die echten Randpunkte, dass die nach innen liegenden Randpunkte die Verfeinerungseigenschaft erfüllen. Deswegen reicht es auch in diesem Falle aus, nach dem Auffüllen der Löcher für die echten Randpunkte *feld-aktuell* auf *false* zu setzen.

Welche Methode ist besser?

Die zweite Methode ist aus einem Grund besser: Pro Gitterpunkt, für den die 5 x 5 Überdeckung gebildet wird, bleibt ein 5 x 5 Feld. Wird der Rand abgespalten wie in Methode 1, dann können in den Eckpunkten 3 x 3 Feingitterfelder übrigbleiben. Die numerischen Eigenschaften von solchen Kleingittern sind natürlich schlechter als von den sonst verbleibenden 5 x 5 Gittern. Implementiert wurde daher Letzteres.

Bemerkung: An dieser Stelle ist auf den Abschnitt 5.9 zu verweisen, weil MaxG auch dann Ränder abschneiden muss, wenn daneben Punkte aus EP^- liegen.

5.4.8 Die Algorithmen zu P_b^+

Wir wiederholen zuerst die Definition von P_b^+(Definition 14):

$$P_b^+ = \Big(\bigcup_{(f,g)\in EP^+} (P_Z(f,g) \bigcup_{HR\notin EP-SNB(f,g)} P(HR,f,g)) \text{ - } \Big(\bigcup_{(f,g)\in EP^v} P_{5,5}(f,g)\Big)\Big) \cap MaxG(t)$$

Der letzte Teil 'MaxG(t)' wurde im vorigen Abschnitt 5.4.7 behandelt. Der Teil
' - ($\bigcup_{(f,g)\in EP^v} P_{5,5}(f,g)$) ' wird geprüft, indem man testet, ob das Feld der Feingitterfeldliste an der Stelle (f,g) ≥ 0 ist, d.h. ob an der Stelle (f,g) schon ein Feingitterpunkt existiert. In dem Fall darf (f,g) nicht in P_k^+ eingefügt werden.

Der Gesamtalgorithmus kommt so zustande: In Algorithmus 5.5 wird je nach Entscheidung der MaxG-Möglichkeiten einer der folgenden 2 Algorithmen mit dem Namen P_k^+-hinzu, also Algorithmus C.2 bzw. 5.4, aufgerufen. Damit wird abgesichert, dass keine Punkte aus $\bigcup_{(f,g)\in EP^v} P_{5,5}(f,g)$ eingefügt, und außerdem nur Punkte aus MaxG(t) verwendet werden. Deshalb müssen je nach Lösung der MaxG-Aufgabe in 5.4.7, 2 Algorithmen definiert werden.

Der erste Algorithmus P_k^+-hinzu, der sich auf MaxG Version 1 bezieht, steht im Anhang, vgl. Algorithmus C.2.

Algorithmus 5.4:P_k^+-hinzu im Fall der MaxG Version 2
Input: (f,g)
Output: (f,g) wird zu P_k^+hinzugefügt oder nicht

1. **if** fgff(f,g) < 0 **then**
2. **return**
3. $P_k^+ = P_k^+ \cup \{(f,g)\}$

Damit muss nach der Definition von P_b^+ nur noch gezeigt werden, wie für die Menge

$$(\bigcup_{(f,g)\in EP^+}(P_Z(f,g) \bigcup_{HR\notin EP-SNB(f,g)} P(HR,f,g))$$

die Funktion P_k^+-hinzu aufgerufen wird. Das wird wie folgt erreicht:

Algorithmus 5.5:P_b^+ *berechnen*

1. **for all** (f,g) $\in EP^+$
2. P_k^+-hinzu(P_Z(f,g))
3. **switch**(255-EP-SNB-Zahl(f,g))
4. **case** 0: Keine Punkte dazu
5. **case** 1: P_k^+-hinzu (N(f,g)) - der Fall für Nord
   ⋮
6. **case** k: P_k^+-hinzu (P(HR,f,g)) für alle HR entsprechend der Zahl k = 255 - *EP-SNB-Zahl*(f,g)
   ⋮
7. **case** 255: P_k^+-hinzu (HR(f,g)) für alle HR $\in$ MHR
8. **end switch**
9. **end for**

Jetzt kommt der Beweis, dass die Menge
$(\bigcup_{(f,g)\in EP^+}(P_Z(f,g) \bigcup_{HR\notin EP-SNB(f,g)} P(HR,f,g))$
berechnet wird.

Zeile 1 des Algorithmus 5.5 sorgt dafür, dass alle Punkte der Menge $(f,g) \in EP^+$ durchlaufen werden, d.h. die Vereinigungsmenge $\bigcup_{(f,g)\in EP^+}$ abgearbeitet wird. Zeile 2 sorgt dafür, dass P_Z(f,g) an Punkte-hinzu übergeben werden. Und die Zeilen 3 bis 7 gewährleisten, dass $\bigcup_{HR\notin EP-SNB(f,g)}$ P(HR,f,g) an Punkte-hinzu übergeben werden.
Bemerkung dazu: Für HR $\notin$ *EP-SNB*(f,g) muss also switch(255 - *EP-SNB-Zahl*(f,g)) genommen werden.
qed.

5.4.9 Die Algorithmen zu P_b^-

Wir wiederholen zuerst die Definition von P_b^-(Definition 5.16):

$$P_b^- = \Big(\bigcup_{(f,g)\in M^a} (P_Z(f,g) - \bigcup_{HR\in EP-SNB(f,g)} P_{5,5}(HR(f,g))) \cap G_h^{alt}$$

Zuerst geben wir die Funktion P_k^--hinzu an. Durch sie wird erreicht, dass nur Punkte aus G_h^{alt} eingefügt werden:

Algorithmus 5.6: P_k^--*hinzu*
Input: (f,g)
Output: Eintragen in P_k^-, falls (f,g) *in* G_h^{alt} ist.

1. **if**(fgff(f,g) ≥ 0)
2. $P_k^- = P_k^- \cup \{(f,g)\}$

Im Fall von fgff(f,g) ≥ 0 ist (f,g) $\in G_h^{alt}$. Gebraucht werden die Werte auf dem Gitter und dem Außenrand, sodass die Prüfung des Feldes des feinen Gitters auf feld-fg(f,g) ≥ 0 nicht hinreichend ist, weil dann Punkte aus dem Außenrand nicht in P_k^- aufgenommen werden können. Diese brauchen wir aber essentiell, weshalb die Prüfung fgff(f,g) ≥ 0 die richtige ist.
Die Funktion P_k^--hinzu wird aufgerufen im Algorithmus 5.7.

Damit bleibt für die Menge $\Big(\bigcup_{(f,g)\in M^a} (P_Z(f,g) - \bigcup_{HR\in EP-SNB(f,g)} P_{5,5}(HR(f,g))))$
P_k^--hinzu aufzurufen.

Vorbemerkung zur Funktionsweise von Algorithmus 5.7:
Es geht in Algorithmus 5.7 um das Hinzufügen einer Auswahl der 4 Punkte von P_Z zu P_k^-. Welche von diesen 4 Punkten genommen werden hängt nur ab von *EP-SNB*(f,g). Man hat also wieder eine **switch-case**-Anweisung mit 256 Fällen.

Verwendet wird die Zahl *EP-SNB-Zahl*, und in dem jeweiligen Fall wird dann die eindeutig bestimmte Teilmenge von Punkten von P_Z mit P_k^--*hinzu* aufgerufen.

Algorithmus 5.7:P_b^-berechnen

1. **for all** (f,g) $\in$ EP-SNB-Punkte
2. **if**(EP^a(f,g)=true) **continue** (d.h. man springt zum nächsten Schleifenelement)
3. **switch**(*EP-SNB-Zahl*(f,g))
4. **case** 0: P_k^--*hinzu* für die Punkte aus P_Z (EP-SNB ist leer)
5. **case** 1: P_k^--*hinzu*({(f+1,g),(f+1,g+1)}), vgl. Beispiel unten in Abbildung 5.11

⋮

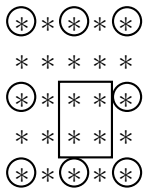


Abbildung 5.10: Die Bestimmung der P_k^-*-hinzu* Punkte im allgemeinen Fall. Das Kästchen gibt an, welche Punkte hinzugefügt werden können, wenn sie nicht durch die 5 x 5 Quadrate um die gekennzeichneten Punkte überdeckt werden, wobei die Punkte gemäß *EP-SNB-Zahl* ausgewählt werden.

6. **case** k: P_k^-*-hinzu*($(\mathrm{P}_Z(\mathrm{f,g})$ - $\bigcup_{HR\in \text{EP-SNB(f,g) entsprechend der Zahl k}} \mathrm{P}_{5,5}(\mathrm{HR(f,g)}))$

   ⋮

7. **case** 255: Nichts tun, da *EP-SNB*(f,g) die maximale Menge MHR ist.

8. **end switch**

9. **end for**

Zum Beweis der Korrektheit des Algorithmus 5.7:
Die Zeilen 1 und 2 stehen für den Teil $\bigcup_{(f,g)\in M^a}$ in der Definition von P_b^-, vgl. die Definition von M^a (Definition 5.15).
Die Zeilen 3 bis 7 erledigen den Ausdruck P_Z(f,g) - $\bigcup_{HR\in \text{EP-SNB(f,g)}} \mathrm{P}_{5,5}(\mathrm{HR(f,g)})$). qed.

Damit bleibt noch zu erläutern, welche Punkte von P_Z in Abhängigkeit der *EP-SNB-Zahl* genommen werden. Es wird zuerst anhand von Abbildung 5.10 das Prinzip gezeigt, das danach an zwei Beispielen erläutert wird.
Man betrachte Abbildung 5.10: Gemäß *EP-SNB-Zahl* werden die umrandeten Punkte ausgewählt. Es ist dabei klar, welcher Punkt zu welcher Himmelsrichtung gehört. Um die ausgewählten Punkte macht man 5 x 5 Umgebungen. Die Punkte des mittleren Kästchens, die davon nicht überdeckt werden, werden an P_k^-*-hinzug* übergeben.

Zum Fall *EP-SNB*(f,g) ={N}: Vergleiche Abbildung 5.11.

Zum Fall *EP-SNB*(f,g) ={NO}: Vergleiche Abbildung 5.12.

5.5 Punktupdate

5.5.1 Punkte hinzufügen

Hier werden alle Punkte (f,g) aus P_b^+bzw. P_k^+durchlaufen.

Dann wird getestet, ob das Grobgitterfeld(f/2,g/2) vorhanden ist. Also wird geprüft,

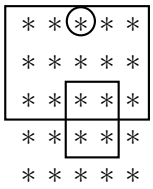


Abbildung 5.11: Die Bestimmung der P_k^--*hinzu* Punkte im Fall *EP-SNB*(f,g) = {Nord}. Das kleine Kästchen gibt P_Z an, das große Kästchen die 5 x 5 Umgebung um den umrandeten Punkt im Norden. Die davon nicht überdeckten P_Z-Punkte werden an P_k^--*hinzu* übergeben. In diesem Fall sind das die unteren 2 Punkte des P_Z-Kästchens.

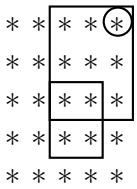


Abbildung 5.12: Die Bestimmung der P_k^--*hinzu* Punkte im Fall *EP-SNB(f,g)* = {NordOst}. Das kleine Kästchen gibt P_Z an, das große Kästchen die 5 x 5 Umgebung um den umrandeten Punkt im Nordosten. Die davon nicht überdeckten P_Z-Punkte werden an P_k^--*hinzu* übergeben.

ob der Punkt auf diesem Kern entstehen kann. Denn nur in diesem Falle ist dieser Kern für die Erzeugung des Punktes zuständig.

In dem Fall wird der Punkt (f,g) in *Gitter* und in *Extreme Vektoren* (wird in Abschnitt 6.7.5 vorgestellt) auf der jeweiligen Gitterebene erzeugt.
Kurz würde das so lauten:

Algorithmus 5.8: Punkte hinzufügen
Input: Gitterebene h, zugehöriges P_k^+.

1. **for all** (f,g) $\in P_k^+$
2. **if**(feld-gg(f/2,g/2) ≥ 0
3. *löschen_einfügen* der Ebene h.einfügen(f,g)
4. **end for**

Dabei bedeutet *löschen_einfügen* der Ebene h.einfügen(f,g), dass auf Ebene h der Punkt (f,g) eingefügt wird. Das hat folgende Konsequenzen:

1. *paralleles_rechnen[h].hinzufügen(f,g)*
2. *migration.ev[h].hinzufügen(f,g)* im Falle der Wahl von ev, oder

   migration.ev2[h].hinzufügen(f,g), und falls (f,g) gerade ist, dann führe noch *migration.ev2[h+1].entfernen*($\frac{f}{2}$,$\frac{g}{2}$) aus.

Bemerkung: ev2[h] kann man immer hinzufügen. Denn da der Punkt noch nicht existierte, existierte insbesondere nichts darunter.

Das wird schon hier gebracht, obwohl die Konzepte von *ev* und *ev2* (Instanzen der Klasse *Extreme Vektoren*) sowie die Klasse *Migration* erst in Kapitel 6 vorgestellt werden.

5.5.2 Punkte entfernen

Entsprechend werden hier alle Punkte (f,g) aus P_b^- durchlaufen.

Dann wird ebenfalls getestet, ob der Grobgitterpunkt(f/2,g/2) vorhanden ist. Denn genau in diesem Fall ist dieser Kern für die Löschung des Punktes zuständig. Und der Kern hat dann auch die nötige EP^a und EP^v Information zur Berechnung von P_k^+ und P_k^- gehabt.

Wenn das der Fall ist, teilt sich diese Aufgabe, den Punkt zu entfernen, auf:

1. Falls fgff(f,g) gleich -1 ist, braucht dieser Punkt nicht entfernt zu werden.

2. Falls fgff(f,g) = ggff(f/2,g/2) ist, d.h. der Punkt liegt auf dem eigenen feinen Gitter, wird der Punkt (f,g) in *Gitter* und in *ExtremeVektoren* entfernt.

3. Andernfalls wird die Information (f,g) an den Kern mit der ID fgff(f,g) verschickt. Dieser löscht dann den Punkt (f,g) in *Gitter* und *ExtremeVektoren*.

   In diesem Fall teilen sich die zwei Kerne die Arbeit: Der Kern mit dem Grobgitterpunkt darüber identifiziert die Stelle, wo gelöscht wird. Und der Kern mit der ID fgff(f,g) nimmt die Löschung vor, weil dort der Punkt eingetragen ist.

Bemerkung 1a: Über die Kommunikation: Sie erfolgt gemeinsam mit der Kommunikation der Farbwerte in der Klasse *Kommunikation4er*. Deshalb ist der vorher genannte Schritt 3 auch nicht nur Teil eines Algorithmus, sondern von drei 'Algorithmen': Zuerst erfolgen Schritte bis zum Packen der Information 'Lösche den Punkt (f,g)' für den Kern fgff(f,g). Dann wird bei der Kommunikation gearbeitet. Und als dritten 'Algorithmus' hat man das Löschen auf den empfangenen Punkten.

Bemerkung 1b: Die Kern-/Kommunikations-Topologie von *Kommunikation4er* ist die des Feingitterfarbenfeldes vereinigt mit der des Grobgitterfarbenfeldes. Denn es wird beim Punkte- und Farbenupdate sowohl die Grobgitter- als auch die Feingitterinformation verwendet, vgl. Abschnitt 5.7.8. Es werden z.B. beim Farbenupdate sowohl Grobgitter- wie Feingitter-Farbeninformationen verschickt.

Bemerkung 2: Man beachte, dass Einfügen und Entfernen nicht gleich ablaufen. Das liegt daran, dass das Einfügen immer auf dem Kern mit der Farbe des Grobgitterfarbwertes erfolgt. Das ist beim Löschen nicht der Fall, wenn sich der zu entfernende Punkt nicht auf dem Kern, den das Grobgitterfarbenfeld angibt, befindet.

5.6 Farbenupdate

5.6.1 Vorbereitende Definitionen für die Farbänderungen

Die Definition von ep-fg

Im Folgenden wird eine Struktur an verschiedenen Stellen benötigt, die der Autor deshalb hier einführt:

Definition 5.18: Gegeben seien die Punktmengen P_k^+ und P_k^-. Dann definieren wir ep-fg durch:

$$\text{ep-fg(f,g)} = \begin{cases} 1 & \text{für } (f,g) \in P_k^- \\ 2 & \text{für } (f,g) \in P_k^+ \\ 0 & \text{für } (f,g) \notin P_k^- \cup P_k^+ \end{cases}$$

Bemerkung 1: Zur Bezeichnung: ep-fg steht für Eigenschaftspunkte (ep) auf dem feinen Gitter (fg).

Bemerkung 2: Dieses Feld erhält man so: Anfangs ist es mit 0 initialisiert. Bevor es verwendet wird, durchläuft man P_k^+ und P_k^- und trägt 2en und 1sen ein (Set). Nach Verwendung durchläuft man wieder P_k^+ und P_k^- und trägt dort 0en ein (Reset). Wichtig ist hierbei, dass Reset noch vor Ablauf der *ParallelenAdaptivenVerfeinerung* erfolgt.

Bei den Änderungsalgorithmen benötigt man die ep-fg Information für Punktepaare. Diese Information ep-fg-pp wird hier definiert:

Definition 5.19: Seien (f,g) und (f',g') ein Punktepaar. Wir definieren ep-fg-pp(f,g,f',g') = (ep-fg(f,g),ep-fg(f',g'))

Bemerkung 1: Verwendet wird ep-fg-pp bei einem direkten Punkt (f,g) mit dem Farbfeldänderungspunkt (f',g'), vgl. die folgenden Definitionen dieser beiden Punkte.

Bemerkung 2: Offenbar gibt es 9 Ausprägungen dieser Variablen ep-fg-pp, und noch 3 Ausprägungen, wenn man einen ep-fg - Wert festlegt.

Definitionen in Bezug auf Punkteeigenschaften

Definition 5.20: Auf einem Gitter sei (f,g) ein Punkt. Der Punkt zP(f,g) = $(2\lfloor\frac{f}{2}\rfloor,2\lfloor\frac{g}{2}\rfloor)$ heißt zentraler Punkt zum Punkt (f,g).

Definition 5.21: Ein β-Punkt oder gemeinsamer Punkt eines Gitters G (bzw. des Gitters der Gitterebene h) ist ein Punkt (f,g) der gleich seinem zentralen Punkt ist, d.h. $(2\lfloor\frac{f}{2}\rfloor,2\lfloor\frac{g}{2}\rfloor)$ = (f,g), d.h. es ist ein Gitterpunkt mit Grobgitterpunkt darüber. Bedingung ist, dass darüber ein Gitter existiert, was bei einer Verfeinerungssituation aber immer der Fall ist.

Bemerkung 1: Eigentlich sollte hier der echte Randpunkt definiert werden. Diese Defini-

tion musste vorgezogen werden, weil der Begriff vorher gebraucht wurde, vgl. Definition 4.8.

Bemerkung 2: Die Eigenschaft, Randpunkt bzw. echter Randpunkt zu sein, ist temporär. In diesem Abschnitt werden auch ehemalige echte Randpunkte verwendet.

Definition 5.22: Zu einem Punkt (f,g) heißt ein Punkt (f',g') owns-daneben, falls für eine Himmelsrichtung HR aus OSWN = {Ost, West, Süd, Nord} HR(f,g)=(f',g') ist.

Definition 5.23: Ein Punkt P ist ein direkter Punkt $\Leftrightarrow$ P ist ein gemeinsamer Punkt oder $\exists_{HR \in MEHR}$ HR(P) ist ein gemeinsamer Punkt und fgff(HR(P))=fgff(P).
Ansonsten heißt er indirekter Punkt.
In Worten: Ein Punkt ist ein direkter Punkt, wenn er benachbart zu einem gleichfarbigen gemeinsamen Punkt ist, oder er selber ein gemeinsamer Punkt ist.

Bemerkung 1: Ein neu hinzukommender Punkt ist immer ein direkter Punkt.

Bemerkung 2: Ein Kern verfügt im Bereich direkter Punkte über die EP-Informationen EP^a und EP^v auf seinem groben Gitter.

Definition 5.24: Stellvertretende Farbänderung
Idee: Bei indirekten Punkten kann folgendes Problem auftreten: Direkte Punkte behandelt der Prozessor selber in Bezug auf Farbänderungen. Bei indirekten Punkten fehlt die Nachbarschaft zu einem gleichfarbigen zentralen Punkt. Deswegen fehlt dem Kern in dem Falle an dieser Stelle ggf. die EP und *EP-SNB* Information.
Als Folgerung ergibt sich: Da seine P_k^+- und P_k^--Informationen unvollständig sind, muss ein Kern, der einen Nachbarpunkt hat, der ein direkter Punkt ist, diese Aufgabe für ihn übernehmen, wobei Kommunikation erforderlich wird. Diese Aktion wird stellvertretende Farbänderung genannt.

Definition 5.25: Wenn zu einem direkten Punkt auf einem Nachbarpunkt eine Farbfeldänderung vorgenommen wird, dann heißt dieser Punkt Farbfeldänderungspunkt.

5.6.2 Wie der Farbenfeldupdate funktioniert

Eine Erläuterung des Farbenupdates

Hier wird die prinzipielle Vorgehensweise erläutert. Wie die Algorithmen dann im Detail korrekt formuliert werden, wird dann in den folgenden Abschnitten ausführlich behandelt.

Die Idee ist die: Man durchläuft den Innenrand, wobei sich zeigen wird, dass es 5 verschiedene Innenränder gibt, die jeweils zu gewissen Zeitpunkten existieren. Sei (f,g) dabei der aktuell betrachtete Punkt des Innenrandes, und der zu behandelnde benachbarte Farbfeldänderungspunkt sei (p,q). Wir diskutieren nun die nötige Farbänderung. Die Situation beider Punkte ist durch ep-fg-pp(f,g,p,q) erfasst. Dann sind es 9 Fälle. Wir diskutieren hier ein paar davon.

Ist ep-fg(f,g) = 1, dann wird der Punkt entfernt, und es muss deswegen keine Farbfeldänderung gemacht werden. Diese drei Fälle in Bezug auf ep-fg-pp seien damit behandelt, vgl. die Bemerkung 2 zu Definition 5.19.
Es sei jetzt angenommen, dass ep-fg(p,q) = 2 ist, d.h. auf der Farbfeldänderungskoordinate entsteht ein neuer Punkt. Über den Farbwert seines zP-Punktes, den wir vom korrekten Grobgitterfarbenfeld ablesen können, bestimmen wir, wie die Farbe zu ändern ist.
Ferner sei als Letztes der Fall zu untersuchen, dass ep-fg-pp(f,g,p,q) = (2,0) ist. Dann muss der Kern, bei dem der Punkt (p,q) liegt, mitteilen, welche Farbe (p,q) hat. Diese wird dann eingetragen.
Alle weiteren Fälle, und insbesondere die Handhabung indirekter Punkte, werden in Abschnitt 5.6.8 behandelt.

Die 3 Teile der Algorithmen

Beim Farbenupdate werden separat die Änderungen aufgrund von P_k^+ zu denen von P_k^- behandelt. In beiden Fällen geschieht das durch folgende Teile, die das Prinzip für den Farbenupdate verdeutlichen.

Farbenupdate für direkte Punkte:

1. Man durchlaufe die Randpunktliste und wählt alle direkten Punkte aus.
2. Man bestimmt dazu die Farbenfeldänderungspunkte mithilfe von Selbstnachbarn.
3. Man bestimmt die Farbwerte für diese Punkte und trägt sie ein.

Farbenupdate für indirekte Punkte:

1. Man durchlaufe die Randpunktliste und wählt alle direkten Punkte aus.
2. Man wählt dazu indirekte Punkte aus (man erreicht so alle, vgl. den späteren Beweis dazu) sowie die Farbfeldänderungspunkte für das Punktepaar direkter/indirekter Punkt(vgl. später).
3. Man schickt die Farbenfeldänderungsinformation an den Kern, der den indirekten Punkt hat, der dann die Farbenfeldänderung durchführt.
4. Man schickt ggf. die Information in Bezug auf den indirekten Punkt an den Kern der Farbenfeldänderungskoordinaten.

Nach diesem Prinzip funktioniert fast der ganze Farbenfeldupdate, bis auf zusätzliche Korrekturen. Wir geben hier noch einen kleinen Überblick über die sich anschließenden Abschnitte:

1. In Abschnitt 5.6.3 wird der Schritt 1 der oben dargestellten Algorithmen zur Bestimmung der Punktmengen durchgeführt.
2. Wie Schritt 2 vom Verfahren 'Farbenupdate für direkte Punkte' zu erfüllen ist, wird in Schritt 2 behandelt. Die Realisierung von Schritt 2 für die indirekten Punkte wird in den Abschnitten 5.6.4 und 5.6.5 behandelt.

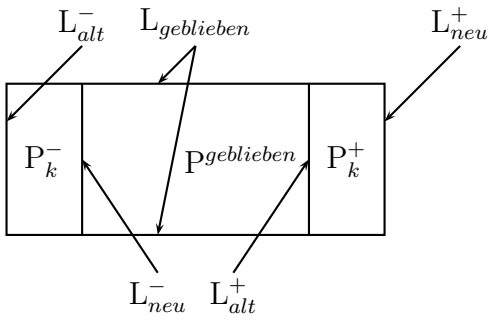


Abbildung 5.13: Die verschiedenen Randpunktmengen beim Punktupdate, immer einschließlich der Verzweigungspunkte. Damit sind die gemeinsamen Punkte verschiedener Linien gemeint.

3. Der jeweilige Schritt 3 zur Bestimmung der Farbwerte findet sich in Abschnitt 5.6.8, mit einer Ausnahme, die in Abschnitt 5.6.7 behandelt wird.

Wenn diese drei Teile korrekt sind, ist auch der Farbenupdate richtig.

5.6.3 Die Randpunktmengen

Die Identifikation der hier verwendeten Randpunktmengen geschieht in 3 Schritten:

1. Die 5 Randpunktmengen darstellen.
2. Erläutern, für welche dieser Mengen hinzugefügte oder entfernte Punkte untersucht werden müssen.
3. In einem Ablauf der Funktion *ParalleleAdaptiveVerfeinerung::tun*, die später erklärt wird, wird diskutiert, welche dieser Punktmengen wann zur Verfügung stehen.
   Durch Bildung und Differenzbildung von Randpunktmengen verschiedener Zeitpunkte verfügt man über die Randpunktmengen, wie sie gebraucht werden.

Die möglichen Randpunktmengen wegen der Punktupdates bei der parallelen adaptiven Verfeinerung

Wir besprechen zuerst die 5 verschiedenen Punktmengen anhand der Linien aus Abbildung 5.13:

1. $L_{geblieben}$: Das sind die Punkte, die vor und nach dem kompletten Punktupdate Randpunkte sind.
2. L^-_{neu}: Das sind die Punkte, die aufgrund des Entfernens von Gitterpunkten Randpunkte geworden sind.
3. L^-_{alt}: Das sind die Punkte, die vor dem Entfernen Randpunkte waren und entfernt werden.
4. L^+_{neu}: Das sind die Punkte, die neu hinzugekommen sind, und zu Randpunkten werden.
5. L^+_{alt}: Das waren ehemalige Randpunkte, die aufgrund neuer Randpunkte vom Rand ins Innere 'gewandert' sind.

Was auf welcher dieser Mengen gemacht werden muss.

Die Schnittpunkte dieser Randmengen müssen für jede dieser Randmengen behandelt werden. Diese Mehrfachbehandlung der Schnittpunkte ermöglicht eine Konzentration der Arbeit auf die jeweilige Randpunktmenge.

1. L^-_{alt}: Hier muss nichts gemacht werden. Nach der Durchführung der Punktupdates gibt es diese Punkte nicht mehr. Deshalb spielt es für den Außenbereich in Bezug auf P^-_k keine Rolle, ob Punkte dazukommen oder entfernt werden. Eine Ausnahme bilden die Schnittpunkte mit anderen Randpunktmengen, die aber dann dort behandelt werden.

2. L^-_{neu}: Für die eigenen wie auch die stellvertretenden Punkte auf dieser Menge ist in das Farbenfeld für das Gebiet P^-_k jeweils -1 einzutragen, was bedeutet, dass dort keine Punkte existieren. Beachtet werden muss dort also nur das Entfernen von Gitterpunkten.

3. $L_{geblieben}$: Die Punkte außerhalb, an deren Position eine Farbänderung infrage kommt, können eingefügt oder entfernt werden. Hier muss das fgff geändert werden, fürs Einfügen und Entfernen von Gitterpunkten in Bezug auf Punkte außerhalb des eigenen Gebietes.

4. L^+_{alt}: An dieser Stelle kommen nur Gitterpunkte hinzu. Deswegen muss nur das Hinzufügen behandelt werden - insbesondere für indirekte Punkte, denn in diesem Fall ist der Farbenupdate für direkte Punkte trivial, muss aber gemacht werden.

5. L^+_{neu}: Es geht hier um die Verbindung zu existierenden Punkten, zu denen durch die hinzugekommenen Punkte eine Nachbarschaft entstanden ist, als auch zu neuen Punkten, wo sich Verfeinerungsgitter aufeinander zu bewegen. Die Entfernung alter Punkte bezieht das nicht ein bis auf Schnittpunkte mit $L_{geblieben}$.

Im Folgenden unterscheiden wir den Farbenupdate danach, ob er aufgrund des Einfügens eines Gitterpunktes oder des Entfernens eines Gitterpunktes zustande kommt.

Das Einfügen von Gitterpunkten erfolgt für $L_{geblieben}$, L^+_{alt} und L^+_{neu} (*), das Entfernen von Gitterpunkten erfolgt für L^-_{neu} und $L_{geblieben}$(**).

Der Ablauf vom Punkt- und Farbenupdate wird angegeben, um zu zeigen, dass die richtigen Punktmengen fürs Einfügen und Entfernen von Gitterpunkten entsprechend (*) und (**) zur Verfügung gestellt werden.

Bevor gezeigt werden kann, dass alle nötigen Randlinien verwendet werden gemäß Abschnitt 5.6.3, muss gesagt werden, wie man an die Randmengen des Programmes kommt:

Der Ablauf von Punkt- und Farbenupdate:

1. Speichere die Randliste in die Liste $\{(f_1,g_1)\}$

2. Punktupdate P^+_k

3. **for all** (f,g) $\in \{(f_1,g_1)\}$
   if(f,g) ist nicht mehr Randpunkt $\Rightarrow$ trage (f,g) unter $\{(f_2,g_2)\}$ ein
4. *Farbenupdate 'hinzufügen'* für die Punktmenge $\{(f_2,g_2)\}$
5. *Farbenupdate 'einfügen'* für den aktuellen Rand
6. *Punktupdate* P_k^-
7. *Farbenupdate 'entfernen'* für den aktuellen Rand

Die initiale Randpunktmenge ist $L_{alt}^- \cup L_{alt}^+ \cup L^{geblieben}$. Nach dem der Punktupdate für P_k^+ durchgeführt wurde ist die Randpunktmenge $L_{alt}^- \cup L_{neu}^+ \cup L^{geblieben}$. Die Differenzmenge $\{(f_2,g_2)\}$ wird dabei durch die Linie L_{alt}^+ gekennzeichnet.

Zum Farbenupdate in Bezug auf das Hinzufügen von Punkten: Vor dem Farbenupdate in Schritt 5 ist die Randpunktmenge $L_{alt}^- \cup L_{neu}^+ \cup L^{geblieben}$. Prüft man die Punkte auf ep-fg = 2, fällt L_{alt}^- weg. Zusammen mit der Menge $\{(f_2,g_2)\}$, also L_{alt}^+, führt man in den Schritten 4 und 5 den Farbenupdate für eingefügte Punkte für die Menge $L_{geblieben}$, L_{alt}^+ und L_{neu}^+ durch - wie zuvor in (*) angegeben.

Zum Farbenupdate wegen dem Entfernen von Punkten: Nach dem Punktupdate aufgrund von P_k^- liegt die Punktmenge L_{neu}^-, $L_{geblieben}$ und L_{neu}^+ vor. Auch hier wird die benötigte Randpunktmenge fürs Entfernen durchlaufen, vgl. zuvor in (**). Das zusätzlich L_{neu}^+ durchlaufen wird, ändert nichts an der Korrektheit.

Bemerkung 1: 'Einfügen' und 'Entfernen' müssen getrennt durchgeführt werden. Die Kommunikationen für stellvertretendes Einfügen können so getrennt in einen Buffer geschrieben werden, was den Kommunikationsaufwand (Bufferlänge und Kommunikationszeit) begrenzt.
Allerdings gibt dann der 'Ablauf von Punkt- und Farbenupdate' nicht genau einen Teil aus der Funktion *tun* der *ParallelenAdaptivenVerfeinerung* wieder, ist also noch kein fertiger Algorithmus, weil der Wechsel in der Kommunikation vom Einfügen zum Entfernen vor dem Entfernen in der Klasse *Farbenupdate* erfolgen muss.
Mit Wechsel ist gemeint, dass die Klasse *Kommunikation 4er* drei verschiedene Informationen packt, nämlich zu entfernende Punkte, Farbwert eintragen und entfernen. Deswegen muss angegeben werden, wann man von einem Block an Informationen zum nächsten geht. Speichert man sie als Blöcke, ist die Information kürzer, als wenn die Zugehörigkeit der Information immer markiert werden müsste (als PU^-, FU^+ oder FU^-, PU steht dabei für Punktupdate, FU für Farbenupdate).
Zu *Kommunikation 4er*: Die Klasse war ursprünglich dazu gedacht, 4 Informationen zu übermitteln, nämlich einzufügende Punkte, zu entfernende Punkte, Farbwerte einzutragen und zu entfernen. Der erste Punkt, einzufügende Punkte, tritt aber nicht auf.

Bemerkung 2: Die Änderungen von PU^-, FU^+ oder FU^- im Anschluss an die Kommunikation erfolgen nach den Schritten des 'Ablaufes von Punkt- und Farbenupdate'. Daher könnte das Timing nicht mehr stimmen. Der wichtigere Punkt für die Kommunikation ist der, wann die Entscheidung für eine Änderung getroffen wird, und nicht,

wann sie durchgeführt wird. Und die Entscheidungen werden stets zum rechten Zeitpunkt getroffen.
Es ist anzumerken, dass der Abschluss der Änderungen vor dem Ende der parallelen adaptiven Verfeinerung erfolgt, also bevor diese Informationen fehlen würden.

5.6.4 Die Punktpaare direkter Punkt und indirekter Punkt finden

Alle direkten Punkte liegen auf den eben erwähnten Randpunktmengen. Nun müssen alle indirekten Punkte behandelt werden.

Sei also ein indirekter Punkt gegeben (wie man ihn eingrenzt: vgl. Satz 5.12). Er liegt also am echten Rand, und als nicht direkter Punkt ist es kein gemeinsamer Punkt.
Weil das Verfeinerungsgebiet durch 5 x 5 Blöcke überdeckt wird, und der Punkt am echten Rand liegt, folgt aus der davon abgeleiteten Beschaffenheit des Randes: Er liegt als mittlerer Punkt in einem der beiden folgenden möglichen Fällen vor:

Fall 1: Die Punkte liegen so vor : • - • - •. Die äußeren Punkte sind dabei gemeinsame Randpunkte, und es sind daher direkte Punkte.

Fall 2: Die Punkte liegen so vor:

•
|
•
|
•

Und auch in diesem Fall sind die äußeren Punkte gemeinsame und damit direkte Randpunkte. Dann gilt:

Satz 5.11:

1. Indirekte Punkte sind immer OSWN-benachbart zu gemeinsamen Randpunkten.

2. Wenn für alle gemeinsamen Randpunkte in den Richtungen Ost/Süd/Nord und West nach indirekten Punkten gesucht wird, werden alle indirekten Punkte gefunden.
   Genauer: Sie werden vertikal oder horizontal eingeschlossen von gemeinsamen Randpunkten. Die Kerne dieser direkten Punkte teilen sich die Arbeit des Farbenupdates für den stellvertretenden Punkt, vgl. Abschnitt 5.6.5.
   Erkennt genau ein Kern den Punkt als indirekten Punkt, so erkennt der andere Kern ihn als direkten Punkt, also ist der Punkt insgesamt ein direkter Punkt.

Bemerkung: Die indirekten Punkte werden mithilfe der direkten Punkte am Rand komplett abgehandelt. Direkte und indirekte Punkte bilden Punktpaare. Das wird in den folgenden Abschnitten 5.6.5 und 5.6.8 erläutert.

Vorbemerkung zum folgenden Satz: Die Menge an Kandidaten für indirekte Punkte kann man weiter einschränken, um die dann anfallende Kommunikation zu verringern.

Das geht mithilfe der nun folgenden vier Bedingungen für indirekte Punkte:

Satz 5.12: Sei zu einem echten Randpunkt (f,g) ein Kandidat (f',g') für einen indirekten Nachbarpunkt gegeben. Um für diesen die stellvertretenden Farbbestimmungen durchzuführen, müssen folgende vier Bedingungen erfüllt sein:

1. (f',g') ist kein gemeinsamer Punkt
2. ep-fg(f',g') = 0
3. fgff(f',g') $\neq$ rang
4. fgff(f',g') $\geq$ 0

Beweis, dass diese 4 Bedingungen erfüllt sein müssen:

Zur Bedingung, dass (f',g') kein gemeinsamer Punkt ist: Andernfalls liegt dort ein direkter Punkt vor, für den die Farbbestimmungen ohne Kommunikation korrekt erfolgen. Ohne diese Bedingung können die Punkte die Rollen tauschen: Der Kern des Punktes, der eigentlich informiert werden muss, verschickt an den Kern des Punktes, der die richtigen Informationen direkt bestimmt, die Umgebungsinformationen bzgl. (f',g'). Ist (f,g) ein echter Randpunkt, und gilt diese Bedingung, dann ist er auch ein gemeinsamer Punkt, und damit ein direkter Punkt.

Zur Bedingung ep-fg(f',g') = 0: Bei ep-fg(f',g') = 2 hätte man einen neuen Punkt, weshalb ein direkter Punkt vorliegt. Das schließt eine stellvertretende Farbänderung aus. Bei ep-fg(f',g') = 1 verschwindet der Punkt, weshalb für ihn keine Farbbestimmungen nötig werden.

Zur Bedingung fgff(f',g') $\neq$ rang: Falls das Feingitterfarbenfeld den Wert des Randes hat, haben (f',g') und (f,g) denselben Farbwert, und es liegt in (f',g') ein direkter Punkt vor, also keine stellvertretende Behandlung.

Zur Bedingung fgff(f',g') $\geq$ 0: Andernfalls gibt es in (f',g') bisher keinen Punkt. Es liegen 2 Fälle vor: Entweder entsteht dort ein Punkt. Dann wäre es ein direkter Punkt, und für diesen macht man keine stellvertretende Farbänderung. Oder es existiert in (f',g') anschließend kein Punkt. Dann ist keine Farbbehandlung für diesen Punkt (f',g') nötig.

Beispiel 5.1: Ein Beispiel für die Wirkung der Bedingungen 2 bis 4 von Satz 5.12.

Sei eine Punktlinie gegeben, die auf k Kernen aufgeteilt ist, und wo darüber keine Punkte liegen, vgl. Abbildung 5.14. Wegen Bedingung fgff(f',g') $\neq$ rang kommen auf der Linie nur solche Punkte für stellvertretende Farbänderungen infrage, die an den Kerngrenzen liegen. Es könnten noch die Punkte stellvertretend sein, die neu dazukommen. Aber die Bedingung ep-fg(f',g') = 0 schließt das aus. De facto sind bei der ganzen Verfeinerung nur jeweils die Punkte auf der Linie, d.h. Punkte mit echtem Rand, Kandidaten für stellvertretende Farbänderung (1-dimensional), und dort nur diese, wo es

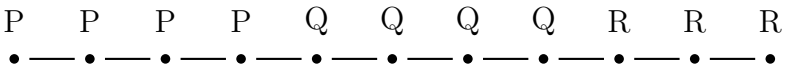


Abbildung 5.14: Abbildung zu Beispiel 5.1: Eine Randlinie auf mehreren Kernen

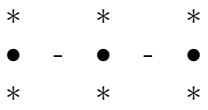


Abbildung 5.15: Die Änderungsmengen für indirekte Punkte im horizontalen Fall: '*' charakterisiert die Änderungspunkte. Der mittlere Punkt ist der indirekte Punkt.

zu Farbänderungen kommt (P ↔Q und Q↔R), und nur für solche Punkte, die es schon gab.

5.6.5 Die zu untersuchenden Außenrandpunkte

Im Fall direkter Punkte (f,g) ist folgendes klar: Über *Selbstnachbarn* bekommt man die Kandidaten, für die das Farbenfeld ggf. geändert werden muss. Damit ist die Bestimmung der Farbenfeldänderungspunkte für direkte Punkte auch behandelt.
Bei indirekten Punkten kann *Selbstnachbarn* nicht verwendet werden: Selbst wenn man für einen möglichen Farbfeldänderungspunkt des anderen Kerns einen Eintrag in *Selbstnachbarn* hat, so kann man diesen nicht verwerten, weil die Eigenfarbe von dem Kern des gemeinsamen Punktes verschieden ist von der Eigenfarbe des Kerns des indirekten Punktes, und daher die Eigenschaft *Selbstnachbar zu sein* in Bezug auf P und Q verschieden ist. Fremde Punkte für P sind nicht notwendig fremde Punkte für Q.

Man geht zur Bestimmung der Farbenfeldänderungspunkte dann wie folgt vor:
Sei also mit (f',g') ein indirekter Punkt gegeben. Zunächst kommen als Punktkandidaten für die Farbänderung die 8 benachbarten Punkte infrage. Die gemeinsamen Punkte, die den indirekten Punkt einschließen, brauchen nicht berücksichtigt werden, weil wegen ep-fg(f',g')=0 der indirekte Punkt ohne Einschränkung schon existierte, und damit auch seine beiden zugehörigen gemeinsamen Punkte. Damit stehen diese Farbwerte dem mittleren indirekten Punkt zur Verfügung.

Dann liegt wiederum einer dieser 2 Fälle vor.

Fall 1: Die Punkte liegen vor entsprechend Abbildung 5.15.

Fall 2: Die Punkte liegen vor entsprechend Abbildung 5.16.

Zu den Abbildungen: Auch hierbei sind die äußeren Punkte gemeinsame Punkte und damit direkte Punkte.
Man sieht, dass jeweils der Kern eines gemeinsamen Punktes diese Farbenfeldänderungen für den indirekten Punkt alleine nicht machen kann. Sein Rand reicht nicht weit

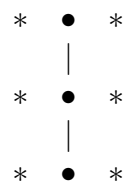


Abbildung 5.16: Die Änderungsmengen für indirekte Punkte im vertikalen Fall: '*' charakterisiert die Änderungspunkte. Der mittlere Punkt ist der indirekte Punkt.

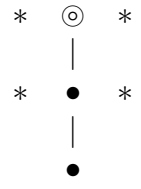


Abbildung 5.17: Die seitliche Änderungsmenge in Bezug auf den gemeinsamen Punkt ◎ oben

genug. Beide Kerne müssen sich diese Arbeit wie folgt teilen:

In Fall 2 sieht das aus wie in den Abbildungen 5.17 und 5.18 dargestellt:

Der Kern des oberen gemeinsamen Punktes übernimmt die Farbenfeldänderungspunkte wie in Abbildung 5.17 mit $*$ gekennzeichnet, und der Kern des unteren gemeinsamen Punktes übernimmt die Farbenfeldänderungspunkte wie in Abbildung 5.18 mit $*$ gekennzeichnet. Damit sind für den Fall 2 die zu untersuchenden Farbenfeldänderungspunkte bestimmt.

Für den Fall 1:
Für den linken gemeinsamen Punkt hat man die Farbenfeldänderungspunkte gemäß Abbildung 5.19 zu untersuchen. Für den rechten gemeinsamen Punkt hat man die Farbenfeldänderungspunkte gemäß Abbildung 5.20 zu untersuchen.

Satz 5.13: Wenn man die Punkte nach Satz 5.11 durchläuft, und für die daraus abgeleiteten Kandidaten der indirekten Punkte die Einschränkungen nach Satz 5.12 prüft, und dann die Farbenfeldänderungspunkte wie oben auswählt, reichen die Punkte in ihrer jeweiligen Funktion, als indirekte Punkte oder Farbenfeldänderungspunkte, für

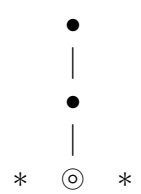


Abbildung 5.18: Die seitliche Änderungsmenge in Bezug auf den gemeinsamen Punkt ◎ unten

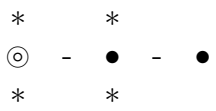


Abbildung 5.19: Die Änderungsmengen oben und unten in Bezug auf den gemeinsamen Punkt ◎ links

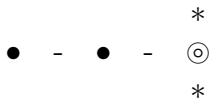


Abbildung 5.20: Die Änderungsmengen oben und unten in Bezug auf den gemeinsamen Punkt ◎ rechts

die Behandlung des Farbenupdates aus.

Beweis: Klar nach den angegebenen Sätzen 5.11 und 5.12.

Bemerkung zur Begrenzung der Kommunikation: Wenn man mit diesen Farbenfeldänderungspunkten arbeitet, können zu demselben Kern dieselben Änderungsinformationen mehrfach übermittelt werden. Man verhindert das durch die Verwendung von Merkfeldern, welche vermerken, wenn einem Punkt eine Farbfeldänderung (löschen oder eintragen) zugeordnet wird, damit das mehrfache Eintragen der Farbinformation eines Punkts vermieden wird. Damit haben die Kommunikationsbuffer nur die wirklich nötigen Informationen übertragen. Für direkte und indirekte Punkte braucht man aber verschiedene Markierungsfelder.

5.6.6 Der Beweis, dass P_k^+ und P_k^- hinreichend genau sind

In den vorangegangenen Abschnitten wurden sowohl die Randpunkte für die Aufgaben als auch die Farbenfeldänderungspunkte zu den Randpunkten bestimmt. Damit hat nun die Bestimmung der Farbwerte zu erfolgen.

Bevor die Farbwerte festgelegt werden können, muss die Korrektheit von P_k^+ und P_k^- und damit die von ep-fg überprüft werden. Denn aufgrund dieser Werte werden die Farbänderungen vorgenommen.

Korrektheit von ep-fg

Definition 26 :

Sei IPM die Innenpunktmenge, also Punkte des Gebietes ohne den Innenrand. Sei IR die Innenrandmenge, AR die Außenrandmenge. Das sind Punkte außerhalb des Gebietes, die zu mindestens einem Punkt aus IR benachbart sind. Dann bezeichnen wir mit AR-2 Punkte, die einen Abstand 2 zu einem und einen Mindestabstand von 2 zu allen Punkten haben. Man vergleiche dazu Abbildung 5.21.

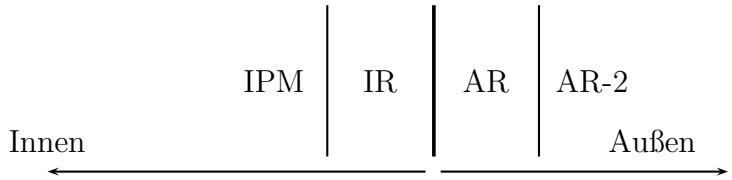


Abbildung 5.21: Die Punktmengen am Rand - für das grobe oder feine Gitter.

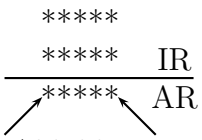


Abbildung 5.22: Die erste Abbildung zu Satz 5.14: Die Feingittersituation

Satz 5.14: Die fehlende Information von EP jenseits von Gebiet und Außenrand ist irrelevant für die Benutzung von P_k^+und P_k^-, mit einer Ausnahme, die erläutert wird, und im Anschluss noch ausgeführt wird.

Beweis:

Man betrachte Abbildung 5.21 für das grobe, die Verfeinerungsentscheidung treffende Gitter.
Vorbemerkung: Die Eigenschaftspunkte sind bekannt bis zu AR. Zu zeigen bleibt, dass es keine Probleme gibt, wenn AR-2 Eigenschaftspunkte nicht erkannt werden.
Wir betrachten dabei zuerst die Situation für Verfeinerungspunkte auf dem groben Gitter, die auf einer Linie liegen, also den eindimensionalen Fall. Wenn ein Eigenschaftspunkt nur auf IPM existiert, also insbesondere nicht auf IR oder AR, dann ist eine Eigenschaftsinformation auf AR-2 irrelevant, weil sich die 5 x 5 Gitter um die Eigenschaftspunkte nicht berühren, also keine Farbänderung erforderlich machen. Bleibt also der Fall, dass ein Entscheidungspunkt auf IR oder AR liegt. Dann sieht die Feingitter Verfeinerungssituation aus wie in Abbildung 5.22. Offenbar haben die Punkte des Kernes, für den die Farbenänderungen gemacht werden, schon an den Stellen im Bereich von AR und IR alle ihre Farbwerte bekommen, weil sie als alte oder neu hinzugekommene Punkte erkannt worden sind. Deswegen ist ein Eigenschaftspunkt auf AR-2 irrelevant für die Farbenänderungen. Man braucht sie nicht, weil die Farben schon aufgrund der vorhandenen Eigenschaftspunkte an den Stellen im Bereich von IR oder AR festgelegt sind oder wurden.

Betrachten wir die Situation jetzt zweidimensional. Es gilt weiterhin, dass ein Eigenschaftspunkt für die IPM in Bezug zu einem Eigenschaftspunkt aus AR-2 zu keinen Berührungen bzgl. 5 x 5 Überdeckungen führen kann. Bleibt damit der Fall von Grobgitterpunkten aus AR oder IR. Man hat damit ohne den AR-2-Eigenschaftspunkt wieder die Situation von Abbildung 5.22. Aber: Ein AR-2-Punkt rechts oder links von dem IR- oder AR- Grobgitterpunkt, führt zu einer Situation wie in Abbildung 5.23 (der Feingitterpunkt zum Grobgitter AR-2-Punkt ist in 5.23 dargestellt) :
Konsequenz: Man betrachte die Punkte links oder rechts auf dem Außenrand AR in Abbildung 5.22, markiert durch Pfeile in dieser Abbildung. Der Kern mit diesen Punkten verfügt über die AR-2 Information. Deshalb muss er dem anderen Kern, der die IR-Punkte darüber hat, mitteilen, ob es links bzw. rechts von ihnen zu neuen Punk-

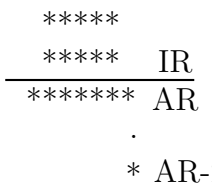


Abbildung 5.23: Die zweite Abbildung zu Satz 5.14: Die Feingittersituation

Abbildung 5.24: Szenario innerer Ecken

ten kommt, wie es in Abbildung 5.23 der Fall ist. Wenn schrägliegende AR-2-Punkte entfernt wurden, muss auch das mitgeteilt werden. Mit dieser Korrektur nach dem eigentlichen Farbenupdate wird die vollständige Korrektheit der Farbenfeldstruktur erreicht.

5.6.7 Das Szenario 'nach Innen gerichtete Ecken' behandeln

Es wird erläutert, wie das folgende Szenario behandelt wird. Alle anderen analogen Szenarien nur mit gespiegelten oder gedrehten Szenarien werden analog behandelt. Es gibt dabei keine Ausnahmen.

Das jetzt behandelte Szenario entsteht z.B. bei diesen EP-Konstellationen, auf dem groben Gitter, vgl. Abbildung 5.24:

Dabei liegen die AR-2-Punkte diagonal vor oder nicht, vgl. den Beweis von Satz 5.14. Die oberen Punkte liegen auf Kern 1, die unteren Punkte auf Kern 3. Auf dem feinen Gitter entsteht dann das folgende Verfeinerungsgebilde von Abbildung 5.25, wobei die innere Ecke mit den Koordinaten (f,g) eingezeichnet ist. Das ist die Stelle, wo in Bezug auf den Überrgang von IR nach AR in Abbildung 5.24 der Pfeil hinzeigt. Liegen die AR-2-Punkte diagonal nicht vor, dann fehlen die linken beiden Punkte des Verfeinerungsgebildes.

Die 'innere Ecke' in Abbildung 5.25 hat die Koordinaten (f,g). Dann muss in dem Fall, falls der Punkt (f,g-1) existiert, der Farbwert von (f,g-1) an den Inhaber des Punktes (f-1,g) geschickt werden, also in diesem Fall zu Kern 1. Und eine -1 wird verschickt, wenn der Punkt nicht existiert, weil die AR-2-Punkte fehlen.

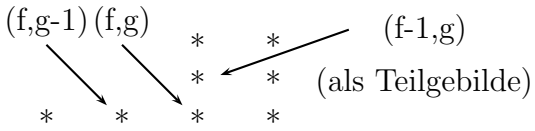


Abbildung 5.25: Ein Verfeinerungsgebilde

Prüfung, ob der Farbwert im Westen zu übermitteln ist

Zuerst ein paar Vorbemerkungen, um die Ausdrücke besser zu verstehen. ep-fg (f,g) = 2 bedeutet, dass an der Stelle (f,g) ein Punkt hinzugekommen ist. ep-fg (f,g) = 1 bedeutet, dass der Punkt an der Stelle (f,g) entfernt wird. ep-fg (f,g) = 0 bedeutet, dass an der Stelle (f,g) keine Änderung erfolgt ist.

Das Folgende gilt immer: Ist ep-fg(f,g) = 2, so entsteht der Punkt, und da der Grobgitterpunkt darüber an der Stelle (f/2,g/2) für das Entstehen zuständig ist, wird sein Farbwert ggff(f/2,g/2) sein. Ist hingegen ep-fg(f,g) = 0, dann ist der Farbwert direkt vom Feingitterfarbenfeld abzulesen, also ist der Farbwert fgff(f,g).

Im obigen Fall ist der Kern des Punktes (f,g) zuständig für die Farbbestimmung. Der zu untersuchende Farbwert hat die Koordinate (f,g-1), ist also der Farbenfeldänderungspunkt, und der Empfänger ist der Kern, der den Punkt (f-1,g) hat.

Genau nach diesen Prinzipien sind die Farbwerte unten gewählt worden.

Nun werden Einschränkungen gemacht an die Entscheidung zur Informationsübermittlung, damit möglichst wenig kommuniziert wird.

1. In Richtung Nord erfolgt die Prüfung, ob dort ein Punkt existiert bzw. entsteht. Nur dann erfolgt ein stellvertretendes Einfügen des Kernes des Punktes (f,g). Im Falle ep-fg(f-1,g) = 0 wird fgff(f-1,g) geprüft (ob es ≥ 0 ist), im Falle ep-fg(f-1,g) = 2 wird ggff($\frac{f-1}{2}$, $\frac{g}{2}$) geprüft, ob es ≥ 0 ist.

2. In Richtung Nordwest soll hinterher kein Punkt liegen. Also gilt: ep-fg(f-1,g-1) = 0 und fgff(f-1,g-1) < 0 oder ep-fg(f-1,g-1) = 1. Bemerkung: Sonst ist die Stellvertretung nicht nötig, da man den Fall der eindimensionalen Linie hat im Beweis zu Satz 5.14.

3. In Richtung West soll ein Punkt liegen. Ist ep-fg(f,g-1) = 2, dann ist das erfüllt. Bei ep-fg(f,g-1) = 0 muss noch fgff(f,g-1) ≥ 0 geprüft werden.
   Wird auf Position (f,g-1) ein Punkt entfernt, weil AR-2-Punkte entfernt werden, wird eine -1 übermittelt, also im Fall ep-fg(f,g-1) = 1.

Bestimmung der ID des Empfängers und der zu übermittelnden Werte

Bei ep-fg(f-1,g) = 0 wird an den Kern mit der Farb-Information fgff(f-1,g) übermittelt. Bei ep-fg(f-1,g) = 2 wird an den Kern ggff((f-1)/2,g/2) übermittelt.

Bei ep-fg(f,g-1) = 0 wird der Farbwert fgff(f,g-1) übermittelt. Bei ep-fg(f,g-1) = 2 (im Westen) wird der Farbwert ggff(f/2,(g-1)/2) übermittelt. Bei ep-fg(f,g-1) = 1 wird eine -1 übermittelt.
Insgesamt wird diese Prozedur für alle entsprechenden Fälle implementiert.

Bemerkung: Es geschieht während des 'Farbenupdate hinzufügen' als Ausnahme die Übermittlung des Farbwertes '-1' bei nach 'innen gerichteten Ecken'. Korrekt ist dieses, da auch während des 'Farbenupdate entfernen' eine -1 eingefügt würde. Nur der

Buffer wird hier um ein Byte länger pro Eintrag. Andernfalls müssten die 'inneren Ecken' doppelt behandelt werden, und zwar einmal beim 'Farbenupdate hinzufügen', und einmal beim 'Farbenupdate entfernen', was auch Nachteile hätte.

5.6.8 Die Bestimmung der Farbwerte

Aus Bemerkungen der Begründung von Abschnitt 5.6.7 rekapitulieren wir:

Es sei (f,g) der direkte Punkt und (f',g') sein Farbenfeldänderungspunkt. Dann ist kurz definiert ep-fg-pp = ep-fg-pp(f,g,f',g').

ep-fg (f,g) = 2 bedeutet, dass an der Stelle (f,g) ein Punkt hinzugekommen ist. ep-fg (f,g) = 1 bedeutet, dass der Punkt an der Stelle (f,g) entfernt wird. ep-fg (f,g) = 0 bedeutet, dass an der Stelle (f,g) keine Änderung erfolgt ist.

Ist ep-fg(f,g) = 2, so entsteht der Punkt mit der Koordinate (f,g), und da der Grobgitterpunkt darüber an der Stelle (f/2,g/2) für das Entstehen zuständig ist, wird sein Farbwert ggff(f/2,g/2) sein. Ist hingegen ep-fg(f,g) = 0, dann ist der Farbwert direkt vom Feingitterfarbenfeld abzulesen, also ist der Farbwert fgff(f,g).

Die Änderungsmengen beim Hinzukommen von Punkten

An dieser Stelle werden die Fälle untersucht, wo Punkte hinzukommen, ohne das welche verschwinden.
Das wird getrennt vom Fall verschwindender Punkte behandelt, weil man erreicht, dass die Kommunikationen fürs Hinzufügen von denen fürs Entfernen (mit Farbwert -1) strikt getrennt werden können, um Speicherplatz in den Kommunikationsbuffern zu sparen. (Die Farbenfeldinformation -1 muss nicht mitübertragen werden.)

Bemerkung 1: Man muss hier beachten, dass entsprechend der Situation mit (f,g) als direktem Punkt und (f',g') als Änderungskoordinate auch die Situation mit vertauschten Rollen auftreten kann.

Bemerkung 2: Im indirekten Fall sind die vertauschten Rollen so, dass die Farbenfeldänderung für den Kern mit den Farbenfeldänderungskoordinaten (f',g') an der Stelle des indirekten Punktes (f",g") zu machen ist.

1. Zum Fall direkter Punkte: (f,g) sei ein direkter Punkt und (f',g') seine Änderungskoordinate.
   Falls ep-fg-pp = (2,2) ist, wird fgff(f',g') = ggff($\frac{1}{2}$f', $\frac{1}{2}$g') gesetzt.
   Falls ep-fg-pp = (0,2) ist, wird fgff(f',g') = ggff($\frac{1}{2}$f', $\frac{1}{2}$g') gesetzt, und es wird der eigene Farbwert unter der Koordinate (f,g) an den Kern mit der ID ggff($\frac{1}{2}$f', $\frac{1}{2}$g') geschickt.
   Falls ep-fg-pp = (2,0) ist, wird an dieser Stelle nichts getan. Es wird entsprechend dem Fall (0,2) aber der Farbwert an der Stelle (f',g') durch Kommunikation bestimmt, die mit der Kommunikationklasse *Kommunikation 4er* übertragen wird.

2. Die Begründung der obigen 3 Fälle.
   Zum Fall ep-fg-pp = (2,2). Der Algorithmus durchläuft beide Stellen (f,g) und

(f',g') spiegelbildlich, und trägt jeweils den korrekten Farbwert ein, der abgeleitet ist vom Grobgitterwert.
Zum Fall ep-fg-pp = (0,2). Erst einmal wird für den Punkt (f,g) der korrekte Farbwert an der Stelle (f',g') eingetragen. Es kommt aber auch zur Situation, wo die Rolle der Punkte vertauscht wird. Weil der Punkt (f',g') die Farbinformation an der Stelle (f,g) nicht selber bestimmen kann, wird ihm diese zugeschickt.
Zum Fall ep-fg-pp = (2,0): Hier wird später für den Kern auf der Position (f,g) der Wert empfangen, der im Fall ep-fg-pp = (0,2) mit vertauschten Rollen verschickt wurde.

3. Zum Fall indirekter Punkte: Mit (f,g) als direktem Punkt und (f',g') als Farbenfeldänderungspunkt ist (f",g") der indirekte Punkt. Da nach Satz 5.12 ep-fg(f",g") = 0 ist, und ferner der Wert ep-fg(f,g) beliebig ist, weil es nur darum geht, die Farbwerte für (f',g') und (f",g") gegeneinander auszutauschen, braucht nur nach ep-fg(f',g') unterschieden werden.
Zum Fall ep-fg(f',g') = 0: Dann ist ep-fg(f',g',f",g") = (0,0), und die benötigten Farbwerte sind beiden Kernen bekannt.
Zum Fall ep-fg(f',g') = 2: Man schickt dem Kern K(f",g") den Wert ggff($\frac{1}{2}$f', $\frac{1}{2}$g') auf der Position (f',g'), und dem Kern (f',g') den Farbwert fgff(f",g") für die Position (f",g").
Zum Fall ep-fg(f',g') = 1: Siehe unten in 'übrige Fälle' Teil 3.

4. Begründung für den Fall ep-fg(f',g') = 2: Da ein neuer Punkt entstanden ist, müssen die beteiligten Kerne ihr Farbenfeld ändern. Auf der Position (f',g') ist wegen ep-fg(f',g') = 2 der neue Farbwert ggff($\frac{1}{2}$f', $\frac{1}{2}$g'), der an den Kern K(f",g") mit der ID fgff(f",g") nebst Koordinaten (f',g') geschickt wird. Auf der Position (f",g") ist wegen ep-fg(f",g") = 0 der Farbwert fgff(f",g"), der dem Kern K(f',g') mit der ID ggff($\frac{1}{2}$f',$\frac{1}{2}$g') zusammen mit den Koordinaten (f",g") übermittelt werden muss.

Die übrigen Fälle

1. Sei (f,g) ein direkter Punkt und (f',g') seine Farbenfeldänderungskoordinaten.
Falls ep-ff-pp = (1,1) oder (1,0) oder (1,2) ist, so ist nichts zu tun.
Falls ep-ff-pp = (0,1) oder (2,1) ist, so wird fgff(f',g') = -1 .

2. Begründungen:
Zum ersten Fall: Falls ep-ff-pp = (1,1) oder (1,0) oder (1,2) ist, verschwindet der Punkt (f,g), weshalb das umgebende Farbenfeld für diesen Punkt irrelevant ist, weil es den Punkt hinterher nicht mehr gibt.
Falls ep-ff-pp = (0,1) oder (2,1) ist, verschwindet der Änderungspunkt, und wir setzen das Farbenfeld dort auf -1.

3. Zum Fall indirekter Punkte: Sei (f,g) der direkte Punkt, (f',g') seine Farbenfeldänderungskoordinate und (f",g") der indirekte Punkt.
Es ist nur noch der Fall ep-fg(f',g') = 1 zu untersuchen.
In diesem Fall schickt (f,g) die Koordinaten (f',g') an K(f",g") in dem *Kommunikation 4er*-Entfernungsblock, damit dieser nach Erhalt am Ende der Prozedur fgff(f",g") = -1 setzt.

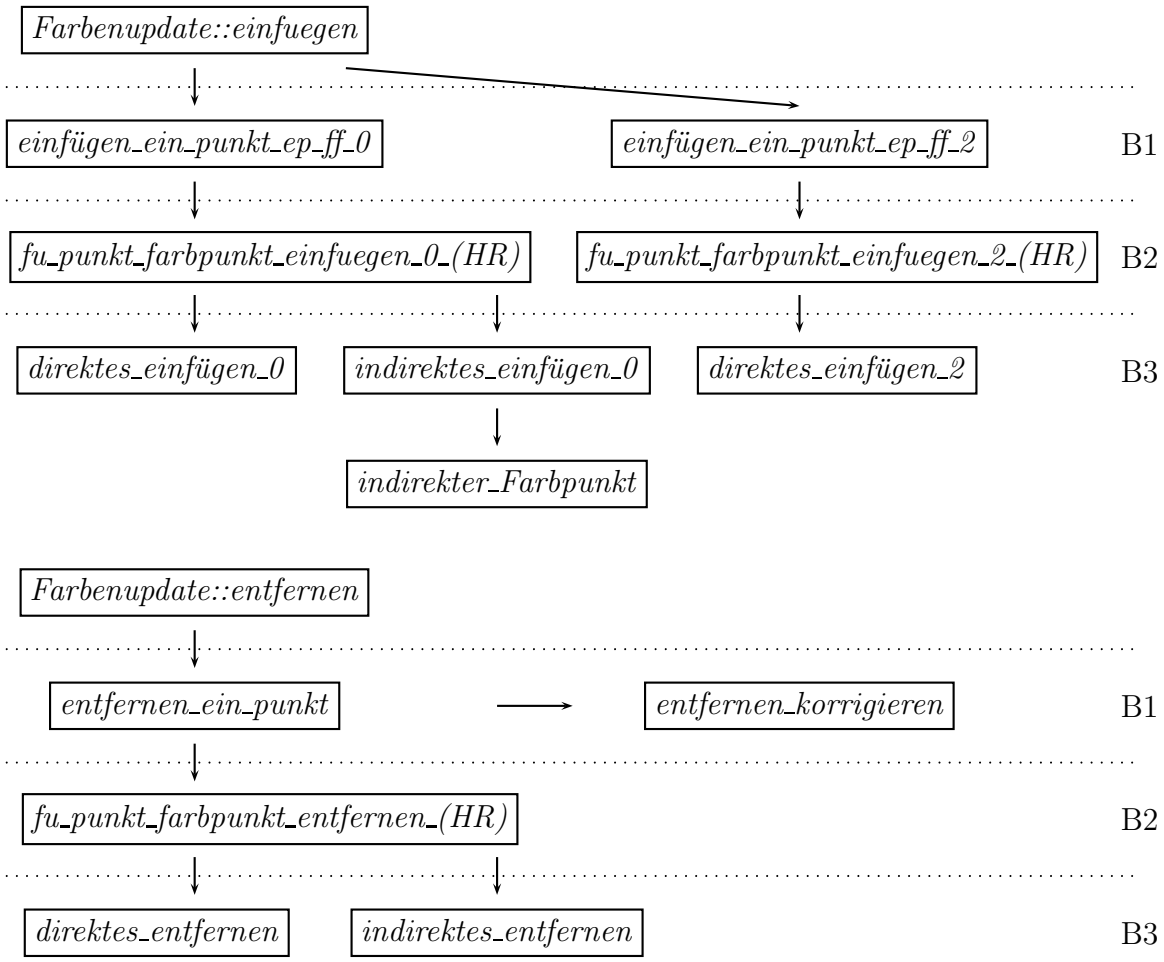


Abbildung 5.26: Die Übersicht über die Algorithmen zu *Farbenupdate::einfuegen* und *Farbenupdate::entfernen*. B1, B2 und B3 sind die Bearbeitungsstufen

4. Begründung: Der Kern K(f",g") muss vermerken, dass der Punkt (f',g') gelöscht worden ist. Daher wird ihm die -1 übermittelt, die er auch in sein fgff-Feld schreibt. An K(f',g') ist nichts zu schicken, weil der Punkt (f',g') nicht mehr existiert.

5.7 Die Algorithmen für den Farbupdate

In Abbildung 5.26 wird eine Übersicht über den Zusammenhang der Algorithmen für den Farbenupdate gegeben. Der genaue Zusammenhalt dieser Algorithmen wird im Folgenden angegeben. Hiermit wird es erreicht, die vorher formulierte Theorie umzusetzen.

5.7.1 Vorbemerkungen zu den Algorithmen

Zunächst sei angemerkt, dass man die Informationen zu *feld-aktuell*, *feld-vergangen*, *EP-SNB*, P_k^+, P_k^- sowie *ep-fg* zur Verfügung hat. Die entsprechenden Algorithmen wur-

den in Abschnitt 5.4 behandelt.

Als nächstes wird der Algorithmus 'ist-unwissend' erläutert. Mit diesem Algorithmus werden direkte und indirekte Punkte unterschieden. Das ist deswegen wichtig, weil nur die Kerne mit direkten Punkten beim Farbupdate Entscheidungen treffen können.

Algorithmus 5.9: *ist-unwissend*

Input: Koordinaten (f,g). Zugriff auf die eigene ID (rang) sowie das Farbenfeld (fgff).

Output: boolsche Variable, die angibt, ob der Punkt 'unwissend' ist oder nicht. Der Punkt ist genau dann unwissend (return *true*), wenn ein indirekter Punkt vorliegt.

1. **switch**((f **AND** 1)*2+(g **AND** 1))

2. **case** 0:[gemeinsamer Punkt] return(*false*) (f=0(mod 2), g=0(mod 2))
   Begründung: ist 'wissend' gilt hier immer, da gemeinsame Punkte immer wissend sind

3. **case** 1:[horizontal zwischen gemeinsamen Punkten]
   (f=0(mod 2), g=1(mod 2))
   if fgff(f,g-1) = rang **OR** fgff(f,g+1) = rang **then**
   return (*false*)
   else
   return(*true*)
   Begründung: Genau dann, wenn eine der beiden if-Bedingungen true ist, 'ist der Punkt wissend', da er auf demselben Kern wie horizontal benachbarte gemeinsame Punkte liegt und daher wie diese korrekte EP und *EP-SNB*-Information hat. Ansonsten ist er nicht wissend.

4. **case** 2:[vertikal zwischen gemeinsamen Punkten]
   (f=1(mod 2), g=0(mod 2))
   if fgff(f-1,g) = rang **OR** fgff(f+1,g) = rang **then**
   return (*false*)
   else
   return(*true*)
   Analoge Begründung wie in Fall 1, nur diesmal vertikal statt horizontal

5. **case** 3:[in allen Diagonalrichtungen sind gemeinsame Punkte]
   (f=1(mod 2), g=1(mod 2))
   if fgff(f-1,g-1) = rang **OR** fgff(f-1,g+1) = rang **OR**
   fgff(f+1,g-1) = rang **OR** fgff(f+1,g+1) = rang **then**
   return(*false*)
   else
   return(*true*)
   Begründung: Genau dann, wenn eine der if-Bedingungen erfüllt ist, ist der Punkt wissend, da mindestens ein diagonal benachbarter gemeinsamer Punkt auf demselben Kern liegt, und daher dieser Kern dort die korrekte EP und *EP-SNB*-Informationen hat.

Bemerkung: Indirekte Punkte sind OSWN-benachbart zu direkten Punkten. Der Algorithmus *ist-unwissend* wird daher nicht im Fall 3 aufgerufen.

Im Folgenden wird (auf Bearbeitungsstufe 1, vgl. Abschnitt 5.7.4) die 256-switch-case Fallunterscheidung in Bezug auf *Selbstnachbarn* vorgenommen. Für die *EP-SNB-Zahl*, die in den Algorithmen 5.21 und 5.27 vorkommt, sei an Definition 5.8 erinnert.

Mit derselben switch-case Fallunterscheidung werden sowohl für die direkten Punkte die Farbenfeldänderungspunkte wie auch die Kandidaten für die indirekten Punkte bestimmt. Bei den indirekten Punkten zählen aber nach Satz 5.11 nur die Richtungen Nord, Ost, Süd und West. Dieser Unterscheidung dienen die folgenden Definitionen:

Definition 5.27: Normale Himmelsrichtungen NHR

Eine Himmelsrichtung HR ist eine normale Himmelsrichtung, bzw. HR $\in$ NHR, wenn sie Nord, Ost, Süd oder West ist.

Definition 5.28: Nicht normale Himmelsrichtungen NNHR

NNHR = MHR - NHR

5.7.2 Die Hauptfunktionen für den Farbenupdate für das Einfügen und Entfernen.

Das sind die Hauptalgorithmen, die die Hauptunter-Algorithmen 5.12 bis 5.14 aufrufen.

Algorithmus 5.10: *Farbenupdate::einfuegen*
Bewirkt die Farbenänderung wie theoretisch beschrieben.

1. **for all** (f,g) $\in \{(f_2,g_2)\}$
2. **if**((ep-fg(f,g) = 0) **AND** (ist-unwissend(f,g)=false)) **then**
3. *einfuegen_ein_punkt_ep_fg_0(f, g)* (Alg. 5.12)
4. **else**
5. Einfügen in Liste unwissender Punkte(f,g)
6. **end if**
7. **end for**
8. **for all** (f,g) $\in$ aktueller Rand
9. **if**((ep-fg(f,g) = 0) **AND** (ist-unwissend(f,g)=false)) **then**
10. *einfuegen_ein_punkt_ep_fg_0(f, g)* (Alg. 5.12)

11. **else**
12. Einfügen in Liste unwissender Punkte(f,g)
13. **end if**
14. **end for**
15. **for all** (f,g) ∈ aktueller Rand
16. **if**(ep-fg(f,g) = 2) **then**
17. *einfuegen_ein_punkt_ep_fg_2(f, g)* (Alg. 5.13)
18. **for all**

Bemerkung: Bei der letzten Schleife braucht nicht auf 'unwissende Punkte' geprüft werden, weil neu hinzukommende Punkte immer 'wissend' sind.

Algorithmus 5.11:*Farbenupdate::entfernen*
Bewirkt die Farbenänderung wie theoretisch beschrieben

1. **for all** (f,g) ∈ aktueller Rand
2. **if**(ist-unwissend(f,g)=false) **then**
3. *entfernen_ein_punkt(f,g)* (Alg. 5.14)
4. **end for**

5.7.3 Die 3 Bearbeitungsstufen

Die eben aufgerufenen 3 Funktionen *einfuegen_ein_punkt_ep_fg_0*, *einfuegen_ein_punkt_ep_fg_2* und *entfernen_ein_punkt* werden in 3 Bearbeitungsstufen behandelt, in deren erster Stufe sie stehen. Auf diese Weise werden für alle Himmelsrichtungen alle Farbenupdates ausgeführt.
Dabei rufen die Funktionen der ersten Stufe die der zweiten Stufe, und die der zweiten Stufe die der dritten Stufe auf.

Diese Bearbeitungsstufen sind deswegen vorhanden, um die 256er Fallunterscheidungen der Bearbeitungsstufe 1 zu ermöglichen. Die Fallunterscheidungen werden vorgenommen, um die Farbenfeldänderungskoordinaten bei direkten Punkten, sowie die Koordinaten der indirekten Punkte bei dem Fall indirekter Punkte zu bestimmen. Damit stimmt (f',g') bei den direkten Punkten mit dem (f',g') der Theorie überein, bei indirekten Punkten aber mit den Variablen (f",g") der Theorie aus Abschnitt 5.6.8. Das ist algorithmisch nur so zu lösen, weil man zwei Aufgaben (direkte und indirekte Punkte zu behandeln) in demselben Algorithmus abhandeln kann.

In Bearbeitungsstufe 3 werden konkrete Werte von ep-fg undd fgff abgefragt und das Farbenfeld wird umgeschrieben.

5.7.4 Die 'Bearbeitungsstufe 1'-Algorithmen

Der folgende Algorithmus dient letztlich dem Farbenupdate für direkte Punkte durch Algorithmus 5.19 und dem Farbenupdate für indirekte Punkte mithilfe von Algorithmus 5.22.

Algorithmus 5.12: *einfügen_ein_punkt_ep_ff_0*
Aufgerufen von dem Algorithmus 5.10.
Input: (f,g)

1. **switch**(255-*Zahl-Selbstnachbarn*(f,g))
   (a) **case** 0: nichts tun
   (b) ⋮
   (c) **case** i: Für alle Himmelsrichtungen HR entsprechend der Zahl i:

   fu_punkt_farbpunkt_einfuegen_0_(HR) (Alg. 5.16)

   (d) ⋮
   (e) **case** 255: *fu_punkt_farbpunkt_einfuegen_0_(HR)* für alle Himmelsrichtungen aus MHR. (Alg. 5.16)
2. **end switch**

Bemerkung 1: **switch**(255-*Zahl-Selbstnachbarn*) wird aufgerufen, weil es um die Punkte geht, die **nicht** auf dem eigenen Kern liegen. Für die Punkte auf dem eigenen Kern hätte man **switch**(*selbstnachbarn*) aufgerufen. Diese Bemerkung gilt auch für die weiteren 'Bearbeitungsstufe 1'-Algorithmen.

Bemerkung 2: '*_(HR)' bedeutet in den 'Bearbeitungsstufe 1 bis 3 Algorithmen', dass die Himmelsrichtung dem Namen der Funktion angefügt wird. Es ist kein(!) Funktionsparameter.

Der folgende Algorithmus dient letztlich dem Farbenupdate für direkte Punkte durch Algorithmus 5.20.

Algorithmus 5.13: *einfügen_ein_punkt_ep_ff_2*
Aufgerufen von dem Algorithmus 5.10.
Input: (f,g)

1. **switch**(255-*Zahl-Selbstnachbarn*(f,g))
   (a) **case** 0: nichts tun
   (b) ⋮
   (c) **case** i: Für alle Himmelsrichtungen HR entsprechend der Zahl i:

   fu_punkt_farbpunkt_einfuegen_2_(HR) (Alg. 5.17)

(d) ⋮

(e) **case** 255: *fu_punkt_farbpunkt_einfuegen_2_(HR)* für alle Himmelsrichtungen aus MHR. (Alg. 5.17)

end switch

Der folgende Algorithmus dient letztlich dem direkten Entfernen in Algorithmus 5.24, sowie dem indirekten Entfernen in Algorithmus 5.25.

Algorithmus 5.14: *entfernen_ein_punkt*
Aufgerufen von dem Algorithmus 5.11.
Input: (f,g)

1. *entfernen_korrigieren*(f,g) (Alg. 5.15)
2. **switch**(255-*Zahl-selbstnachbarn*(f,g))
   (a) **case** 0: nichts tun
   (b) ⋮
   (c) **case** i: Für alle Himmelsrichtungen HR entsprechend der Zahl i:

   fu_punkt_farbpunkt_entfernen_(HR) (Alg. 5.18)

   (d) ⋮
   (e) **case** 255: *fu_punkt_farbpunkt_entfernen_(HR)* für alle Himmelsrichtungen aus MHR. (Alg. 5.18)
3. **end switch**

Vorbemerkung zu Algorithmus 5.15: Entfernen korrigieren bewirkt in gewissen Fällen das Setzen von Farbwerten auf -1, siehe unten, falls dies möglich ist. Es hat damit einen Einfluss auf die Bedingung 3 von Satz 5.12. Deswegen geschieht *entfernen_korrigieren* vor dem eigentlichen *Farbenupdate-entfernen* Algorithmus.

Algorithmus 5.15: *entfernen_korrigieren*
Aufruf vom Algorithmus 5.14 aus.
Input: (f,g) - diese Koordinaten werden abgelesen, da der Algorithmus in derselben Klasse wie Algorithmus 5.14 ist

1. **if**((f,g) ist kein gemeinsamer Punkt **OR** (ep-fg(f,g)$\neq$ 2)) **then return**
2. bestimme f_grob und g_grob ($\frac{1}{2}$f, $\frac{1}{2}$g)
3. **switch**(255-*Zahl-Selbstnachbarn*(f,g))
   (a) **case** 0: nichts tun
   (b) ⋮

(c) **case** i: Für alle Himmelsrichtungen HR entsprechend der Zahl i:

entfernen_korrigieren_(HR) (Alg. 5.27)

(d) ⋮

(e) **case** 255: *entfernen_korrigieren_(HR)* für alle Himmelsrichtungen aus MHR. (Alg. 5.27)

4. **end switch**

Bemerkung: *entfernen_korrigieren_(HR)* wird den 'Bearbeitungsstufe 3'-Algorithmen zugeordnet. Denn diese Algorithmen bewirken die konkreten Änderungen.

5.7.5 Die 'Bearbeitungsstufe 2'-Algorithmen

Algorithmus 5.16: *fu_punkt_farbpunkt_einfuegen_0_(HR)*
Mit HR als Teil des Funktionsnamens.
Dieses geschieht für schon existierende direkte Punkte, vgl. Algorithmus 5.17.
Input: (f,g)

1. (f',g') = HR(f,g) // fest programmiert, da HR jeweils fest ist
2. *direktes_einfuegen0()*
3. *indirektes_einfuegen0_(HR)()* (Alg. 5.22) für den Fall, dass HR ∈ NHR ist, vgl. Abschnitt 5.7.3.
4. *innere_ecken_behandeln_(HR)()*, vgl. Abschnitt 5.6.7

Bemerkung: Das HR in Schritt 1 für Schritt 2 hat einen anderen Zweck als in Schritt 3. In Schritt 1 bestimmt es die Farbenfeldänderungskoordinaten, während es in Schritt 3 der Bestimmung der Koordinaten der indirekten Punkte dient, vgl. Abschnitt 5.7.3.

Algorithmus 5.17: *fu_punkt_farbpunkt_einfuegen_2_(HR)*
Mit HR als Teil des Funktionsnamens.
Dieses geschieht für neu hinzukommende direkte Punkte. Man vergleiche damit den Algorithmus 5.16.
Input: (f,g)

1. (f',g') = HR(f,g) // fest programmiert, da HR jeweils fest ist
2. *direktes_einfuegen2()*

Bemerkung 1: innere Ecken nur einmal behandeln, d.h. sie tauchen nur einmal auf, also nicht hier, nur in Algorithmus 5.16.

Bemerkung 2: Indirekte Punkte werden im Fall von ep-fg(f,g) = 2 nicht behandelt. Denn dann ist ep-fg(f",g") = 2, mit Widerspruch zu den Einschränkungen von Satz 5.12.

Algorithmus 5.18: *fu_punkt_farbpunkt_entfernen_(HR)* für HR ∈ NHR
Mit HR als Teil des Funktionsnamens
Input: (f,g)

1. (f',g') = HR(f,g) // fest programmiert, da HR jeweils fest ist
2. *direktes_entfernen()*
3. *indirektes_entfernen_(HR)()* für den Fall, dass HR ∈ NHR ist

Bemerkung 1: *indirektes_einfuegen_0* und *indirektes_entfernen* geschieht nur für die Himmelsrichtungen HR ∈ NHR, vgl. Satz 5.11.

Bemerkung 2: Hier erfolgt wiederum die Farbbehandlung gemeinsam für direkte und indirekte Punkte, vgl. Abschnitt 5.7.3.

5.7.6 Die 'Bearbeitungsstufe 3'-Algorithmen

Im Prinzip wurde in Abschnitt 5.6.8 festgelegt, wie das zu erfolgen hat. Hier die Algorithmen:

Das Hinzufügen von Punkten

Algorithmus 5.19: *direktes_einfügen_0*

Input: Zugriff auf (f,g) und (f',g')

1. **if** ep-fg(f',g') = 2 **then**
2. Übernehme Grobgitterpunktfarbe für fgff(f',g')
3. Schicke an diesen Kern die eigene Farbinformation von (f,g)
4. **end if**

Bemerkung: Das ist der Fall ep-fg-pp = (0,2) in Abschnitt 5.6.8 in Unterabschnitt 'Die Änderungsmengen beim Hinzukommen von Punkten' Punkt 1.

Algorithmus 5.20: *direktes_einfügen_2*

Input: Zugriff auf (f,g) und (f',g')

1. *Lösche_Farbenfeld_vor_neuen_Punkten_EP-Information_dort_false*
   Das ist Algorithmus 5.21.
2. **if** ep-fg(f',g') = 2 **then**
3. Übernehme Grobgitterpunktfarbe für fgff(f',g')

Bemerkung 1: Das ist der Fall ep-fg-pp(f,g) = (2,2) in Abschnitt 5.6.8 in Unterabschnitt 'Die Änderungsmengen beim Hinzukommen von Punkten' Punkt 1.

Bemerkung 2: Der Fall ep-fg-pp(f,g) = (2,0) in Abschnitt 5.6.8 steht hier in keinem Algorithmus. Das ist richtig, weil der Fall mithilfe der Kommunikation des Kerns erfolgt,

der den Wert ep-fg-pp = (0,2) hat.

Algorithmus 5.21: *Lösche_Farbenfeld_vor_neuen_Punkten_EP-Information_dort_false*
Aufgerufen von Algorithmus 5.20.

Input: Zugriff auf (f,g) und (f',g'). Seien (p',q') die zu (f',g') gehörenden Grobgitterkoordinaten (durch 2 teilen).

1. **if** ep-fg(f,g) = 2 **AND** ep-fg(f',g') = 0 **AND** feld-aktuell(p',q') = false **AND** EP-SNB-Zahl(p',q') = 0 **then**

2. Schreibe fgff(f',g') = -1

Bemerkung 1: Das geschieht vor dem indirekten Entfernen, einem Unter-Algorithmus eines später aufgerufenen Haupt-Algorithmus, weil die Bedingung fgff ≥ 0 an indirekte Punkte hierdurch beeinflusst wird. fgff wird damit aktualisiert.

Bemerkung 2: Zur Korrektheit von Algorithmus 5.21. Er arbeitet korrekt, weil im Fall von EP-SNB(p',q') = 0 und feld-aktuell(p',q') = false der Punkt (f',g') = (2p',2q') danach nicht mehr existiert, weil Grobgitterpunkte mit der Verfeinerungseigenschaft einen Grobgitterabstand von mindestens 2 haben.

Algorithmus 5.22: *indirektes_einfügen_0* für HR $\in$ NHR

1. **if** ((f' **AND** 1) $\neq$ 0) **OR** (g' **AND** 1) $\neq$ 0)) **AND** ep-fg(f',g') = 0 **AND** fgff(f',g') $\neq$ rang **AND** fgff(f',g') $\geq$ 0 **then**

2. Rufe *indirekter_farbpunkt* (Alg. 5.23) auf für die Punkte entsprechend der Abbildungen 5.17 bis 5.20.
(Das sind die Farbenfeldänderungspunkte der indirekten Punkte.)

Bemerkung: Es muss geprüft werden, ob ein indirekter Punkt möglich ist nach Satz 5.12. Falls ja, dann wird für die Punkte nach den Abbildungen 5.17 bis 5.20 der Algorithmus 5.23 aufgerufen.

Algorithmus 5.23: *indirekter_farbpunkt*
Dieser Algorithmus wird von Algorithmus 5.22 aufgerufen. (f",g") sind hier - im Gegensatz zur Bezeichnung in der Theorie aus Abschnitt 5.6.8 - die Farbenfeldänderungskoordinaten.

1. **if** ep-fg(f",g") = 2 **then**

2. Schicke ggff(f"/2,g"/2) und (f",g") zu Kern fgff(f',g')
(Schicke Farbwert und Koordinate zum Kern.)

3. Schicke falls ggff $\neq$ rang: fgff(f',g') und (f',g') zu Kern ggff(f"/2,g"/2)
(Schicke Farbwert und Koordinate zum Kern.)

4. **end if**

Bemerkung 1: Das ist der Fall ep-fg(f',g') = 2 in Abschnitt 5.6.8 in Unterabschnitt 'Die Änderungsmengen beim Hinzukommen von Punkten' Punkt 3. (f',g') verstanden in der Terminologie von Abschnitt 5.6.8 - das entspricht hier der Abfrage ep-fg(f",g") = 2.

Bemerkung 2: Wichtig: Das erfolgt hier getrennt vom Entfernen in Algorithmus 5.26, vgl. die Argumentation in Abschnitt 5.6.8.

Das Entfernen von Punkten

Algorithmus 5.24 : *direktes_entfernen*

1. **if** ep-fg(f',g') = 1 **then** fgff(f',g') = -1

Bemerkung: Das ist der Fall ep-fg-pp = (0,1),(1,1) oder (2,1) in Abschnitt 5.6.8 in Unterabschnitt 'Die übrigen Fälle' Punkt 1.

Algorithmus 5.25: *indirektes_entfernen* für HR $\in$ NHR

1. **if** ((f' **AND** 1) $\neq$ 0) **OR** (g' **AND** 1) $\neq$ 0)) **AND** ep-fg(f',g') = 0 **AND** fgff(f',g') $\neq$ rang **AND** fgff(f',g') $\geq$ 0 **then**

2. *indirekter_farbpunkt_entfernen* (also Algorithmus 5.26) für die Punkte entsprechend der Abbildungen 5.17 bis 5.20.

Algorithmus 5.26: *indirekter_farbpunkt_entfernen*
Aufruf dieses Algorithmus durch den Algorithmus 5.25. (f",g") sind die Farbenfeldänderungskoordinaten nach den Abbildungen 5.17 bis 5.20.

1. **if** ep-fg(f",g") = 1 **then**

2. Schicke an fgff(f',g') die Information, dass an der Stelle (f",g") eine -1 ins Feingitter-Farbenfeld zu schreiben ist.
((f",g") werden mitgeschickt, die -1 nicht, weil diese Informationen in den letzten Block von *Kommunikation 4er* gepackt werden.)

Bemerkung: Das ist der Fall ep-fg(f',g') = 1 in Abschnitt 5.6.8 in Unterabschnitt 'Die übrigen Fälle' Punkt 3, in der Terminologie des Abschnitts 5.6.8, vgl. Abschnitt 5.7.3.

Den Algorithmus *entfernen korrigieren*

Dieser Algorithmus wird aufgerufen in Algorithmus 5.15. Bevor es zum Entfernen in Bezug auf direkte oder indirekte Punkte kommt, werden in das Farbenfeld korrekte '-1' Werte eingetragen, damit *entfernen_korrigieren* funktioniert. Die Korrektheit dieser Funktion wird im Anschluss bewiesen. Der Name *entfernen_korrigieren* kommt daher, weil eine Korrektur des Farbenfeldes auf '-1' beim Entfernen aufgrund von EP-Informationen erfolgt.

Algorithmus 5.27: *entfernen_korrigieren_(HR)*
Input: Es wird dabei f_grob und g_grob mitübergeben.

Die Algorithmen für HR $\in$ NHR und HR $\in$ NNHR sind verschieden. Daher behandeln

wir den Algorithmus für Nord und NordOst.

Algorithmus *entfernen_korrigieren_Nord*

1. **if**((*EP-SNB*(f_{grob}-1,g_{grob}) **AND** 124 = 0) **AND** (ep-aktuell(f_{grob}-1,g_{grob})=false) **then**
2. fgff(f-1,g) = -1

Es ist (f_{grob},g_{grob}) = ($\frac{1}{2}$f, $\frac{1}{2}$g), und (f,g) ein gemeinsamer Punkt.
Dabei steht die erste if-Bedingung für ep-aktuell an den Stellen {(f_{grob}-1,g_{grob}-1), (f_{grob}-1,g_{grob}+1),(f_{grob},g_{grob}-1),(f_{grob},g_{grob}) und (f_{grob},g_{grob}+1)}. Die zugehörigen Zahlen der HR's in Bezug auf die Koordinaten (f_{grob}-1,g_{grob}) und die eben genannten Stellen sind 64, 32, 16, 8 und 4, und ihre Summe ist 124, vgl. Zeile 1. Ist *feld-aktuell* an allen diesen Stellen *false* und gilt dazu auch die 2. if-Bedingung, so exisiert an der Feingitterposition (f-1,g) zwingend kein Punkt, weil diese 6 Grobgitterpunkte den Punkt (f-1,g) vor EP-Einträgen 'abschirmen', und es kann fgff(f-1,g) = -1 geschrieben werden.
Denn alle anderen Grobgitterpunkte reichen in Bezug auf eine 5 x 5 Feingitterüberdeckung nicht mehr bis (f-1,g).

Algorithmus *entfernen_korrigieren_NordOst*

1. **if**((*EP-SNB*(f_{grob}-1,g_{grob}) **AND** 28 = 0) **AND** (*feld-aktuell*(f_{grob}-1,g_{grob})=*false*) **then**
2. fgff(f-1,g+1) = -1

Hier werden nur die zu (f-1,g+1) diagonal liegenden Grobgitterpunkte auf *feld-aktuell* = *false* getestet. Denn die *feld_aktuell* - Werte der Punkte (f_{grob}-1,g_{grob}+1), (f_{grob},g_{grob}+1) und (f_{grob},g_{grob}) entsprechen den *EP-SNB*(f_{grob}-1,g_{grob})-Werten von 4, 8 und 16, in der Summe also 28, und der Punkt(f_{grob}-1,g_{grob}) wird direkt überprüft. Falls aber die diagonal liegenden Grobgitterpunkte in Bezug auf *feld-aktuell false* sind, existiert auf (f-1,g+1) kein Gitterpunkt.
Denn alle anderen Grobgitterpunkte reichen in Bezug auf eine 5 x 5 Feingitterüberdeckung nicht mehr bis (f-1,g+1) - nur bis zu den diagonal liegenden Grobgitterpunkten.

5.7.7 Der Beweis der Algorithmen

De facto sind die Algorithmen direkt entsprechend Abschnitt 5.6.8 entwickelt. Um alle Punktpaare zu durchlaufen für alle Himmelsrichtungen und die Farbänderungen durchzuführen, werden die Hauptfunktionen sowie alle 3 Bearbeitungsstufen gebraucht. Die Änderungen selber entsprechen denen der vorher abgeleiteten Theorie von Abschnitt 5.6.8.

5.7.8 Die Kommunikation von Punkteupdate und Farbenupdate, insbesondere mit der Kerntopologie dazu.

1. Es werden Buffer bereitgestellt für alle Farben. Denn die Punktnachbarschaften können für alle Punktpaare nötig werden.

```
0  0 -1  1  1                0  0  0  1  1
0  0 -1  1  1      ⟶        0  0  0  1  1
2  2 -1  3  3                2  2  2  3  3
2  2 -1  3  3                2  2  2  3  3
```

Abbildung 5.27: Das erste Beispiel zur Kerntopologie für die Kommunikation der parallelen adaptiven Verfeinerung

```
-1  0      ⟶    2  0        2  0      ⟶    2  0
-1  1           2  1        2  0           2  0
```

Feingitter-Kerntopologie Grobgitter-Kerntopologie

Abbildung 5.28: Das zweite Beispiel zur Kerntopologie für die Kommunikation der parallelen adaptiven Verfeinerung. Links von $\rightarrow$ ist die Situation vor dem Verfeinerungsupdate, rechts von $\rightarrow$ die Situation nach dem Verfeinerungsupdate dargestellt.

2. Die Kommunikation erfolgt zu allen Kernen, welche entweder auf dem groben oder dem feinen Gitter benachbarte Punkte haben. Nur für die Kerntopologie KT des feinen Gitters alleine können Kommunikationsverbindungen fehlen.

   Beispiel a, vgl. Abbildung 5.27: Man habe $4 = 2$ x 2 Kerne. Man habe zwei Verfeinerungsgitter gleicher Größe: Die Punkte des linken Gitters liegen bei Kern 0 und 2, auf dem rechten Gitter bei Kern 1 und 3. Werden die Gitter nun verbunden, erhält man eine andere Feingitter-Kerntopologie als vorher. Die Grobgittertopologie bestimmt dabei die neuen Verbindungen mit.

   Beispiel b: Die Vereinigungstopologie von grobem und feinem Gitter reicht aber aus, vgl. Abbildung 5.28: Man nehme einen gemeinsamen Punkt auf Kern 0, und südlich daneben einen indirekten Punkt auf Kern 1. Man bilde daneben 2 westliche Punkte auf Kern 2. Die Kerne 1 und 2 sind vorher weder in der Grobgittertopologie benachbart, dort sind es Kerne 0 und 2, noch in der Feingittertopologie benachbart. Dort sind die Kerne 0 und 1 benachbart und Kern 2 kommt neu dazu. Nachher sind sie es doch, weil Kern 0 stellvertretend für Kern 1 den Farbenfeldwert 2 zu Kern 1 schickt.
   Um das zu erreichen, muss keine 1 zu 2 Kerntopologieverbindung existieren!

3. Es werden ferner bei der Klasse *Kommunikation_4er* 3 verschiedene Kommunikationen hintereinander gepackt, nämlich für *Punkteupdate_hinweg* und *Farbenupdate_hinzu* oder *Farbenupdate_hinweg*. Die Kommunikation für *Punkteupdate_hinzu* fällt dabei weg, weil hinzugefügte Punkte immer direkte Punkte sind. Das Einfügen macht deshalb nie eine Kommunikation erforderlich.

   Das Entpacken erfolgt für die Punkte und Farben jeweils hintereinander in der Reihenfolge, in der auch gepackt wird, nämlich erst Einträge für FU^+, dann Einträge für P_k^-, und dann Einträge für FU^-, wobei FU für Farbenupdate steht.

5.8 Die Gesamtfunktion

Hier wird die Haupt-Funktion der parallelen adaptiven Verfeinerung angegeben. Diese besteht daraus, die entsprechenden oben genannten Algorithmen in der richtigen Reihenfolge aufzurufen.

Vorbemerkung 1: Mit dem Zeichen '&' wird ein Zeigerzugriff beschrieben. Auf diese Weise werden hier Klassen übergeben.

Vorbemerkung 2: Es werden in diesem Algorithmus die Funktionen *Farbenupdate_vervollständigen* und *Punktupdate.vervollständigen* genannt. Das sind einfache Funktionen: Beim *Farbenupdate.vervollständigen* werden die aus dem Buffer gelesenen Koordinaten und Farbwerte beim Hinzufügen in das Farbenfeld geschrieben. Beim Entfernen wird an der Stelle der gelesenen Koordinaten eine -1 ins Farbenfeld geschrieben. Beim *Punktupdate.vervollständigen* wird der Punkt an den jeweils übermittelten Koordinaten gelöscht.

Vorbemerkung 3: Kommunikation vorbereiten und beenden enthält selber keine Kommunikation, sondern legt den jeweiligen Kommunikationsblock fest.

Algorithmus 5.28: Hauptalgorithmus der parallelen adaptiven Verfeinerung
Input: EP^a und EP^v liegen vor.

1. EP's bestimmen (EP^+ und EP^-)
   (Mit Zugriff auf *PunktFeldlisten* für *feld-aktuell* und *feld-vergangen.*)
2. *EP-SNB* updaten(&EP^+,&EP^-)
   (Mit Zugriff auf *PunktFeldlisten* für *feld-aktuell* und *feld-vergangen.*)
3. P_k^+und P_k^-bestimmen
   (Mit Zugriff auf *PunktFeldlisten* für *feld-aktuell* und EP^+und EP^-.)
   Mit Reset von P_k^+und P_k^-.
4. *ep-fg* daraus bestimmen(&P_k^+,&P_k^-)
5. (f_1,g_1) speichern
6. *Punktupdate_hinzu*(&P_k^+)
7. (f_2,g_2) bestimmen
8. *Kommunikation4er.vorbereiten* FU^+
9. *Farbenupdate_hinzu*
   (Mit Zugriff auf ep-fg, fgff, ggff)
10. *Kommunikation4er.beenden* FU^+
11. *Kommunikation4er.vorbereiten* P_k^-
12. *Punktupdate_hinweg*(&P_k^-)

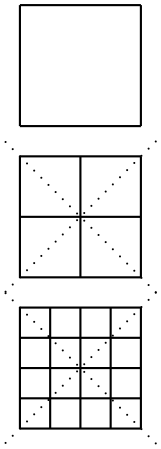


Abbildung 5.29: Das Prinzip hinter dem rekursiven Entfernen. Die gestrichelten Linien zeigen, welche Gitter entfernt wurden.

13. *Kommunikation4er.beenden* P_k^-
14. *Kommunikation4er.vorbereiten* FU^-
15. *Farbenupdate hinweg*
    (Mit Zugriff auf ep-fg, fgff, ggff)
16. *Kommunikation4er.beenden* FU^-
17. *Kommunikation4er.kommunikation-durchführen*
18. *Farbenupdate.vervollständigen* (aufgrund der Kommunikation)
19. *Punkteupdate.vervollständigen* (aufgrund der Kommunikation)
20. Reset *ep-fg* (P_k^+,P_k^-)
21. Reset der unwissenden Punkte
22. Reset der EP's
23. Reset der *EP-SNB-Punkte*

Bemerkung 1: Ein Reset von EP^v oder EP^a kommt auch vor, aber außerhalb dieser Funktion. Es wird genau eine dieser Mengen gelöscht. Welche das ist, hängt von der Position des Resets im Programm ab.

5.9 Rekursives Entfernen

Bisher wird die jeweils nächste Verfeinerungsebene korrekt bestimmt. Dort jedoch, wo Punkte entfernt wurden, existieren eine Ebene tiefer die Punkte potenziell immer noch. Das wäre falsch, weil dann Punkte existieren würden, über denen es keine Grobgitterpunkte gibt, was unzulässig ist.
Dieses Problem wird so gelöst: Die Punkte (f,g) des feinen Gitters, die entfernt werden, werden eingetragen in die EP^--Struktur derjenigen Gitterebene, aus der die Punkte entfernt wurden, was in Abschnitt 5.4.5 erklärt wurde. Man betrachte nun Abbildung

5.29. Denn die parallele adaptive Verfeinerung bezieht sich immer auf zwei Gitterebenen, hat also selber eine Gitterebenenzuordnung. Es werden die feinen Punkte der höheren Ebenen der parallelen adaptiven Verfeinerung zu groben Punkten der nächsttieferen Ebenen der parallelen adaptiven Verfeinerung. Das ist die Ebene h in Abbildung 5.29. Das gilt auch für mittels Kommunikation entfernte Punkte eines anderen Kerns, vgl. Abschnitt 5.5.2.

Wird dann die parallele adaptive Verfeinerung in Bezug auf die nächsten beiden Gitterebenen (h und h-1 entsprechend der Abbildung) gemacht, werden die Punkte der feineren Gitterebene dieser parallelen adaptiven Verfeinerung (Ebene h-1) entfernt.

5.9.1 Rekursives Entfernen in Kombination mit MaxG-Rändern

Es seien 3 Gitterebenen betrachtet: h+1, h, und h-1. Damit gilt die folgende Regel: Wenn Einträge in EP^- auf Ebene h geschehen, müssen diese Punkte für MaxG in Bezug auf die Ebenen h und h-1 berücksichtigt werden. Die Idee ist die, dass der Eintrag entfernter Punkte in EP^- perfekt zusammenwirkt mit MaxG, ohne dass Zusätzliches getan werden muss, vgl. den Beweis unten.

Im folgenden Beweis behandeln wir die zweite Methode für MaxG, bei der die Information '*feld-aktuell* = *true*' vom echten Rand nach innen transportiert wird.

Beweis:

Sei A ein Punkt, der die Verfeinerungseigenschaft erfüllt, und M die Menge seiner benachbarten existierenden Gitterpunkte. Diese seien auf der Ebene h definiert. Punkt A soll nun aufgrund einer Entscheidung der Ebene h+1 entfernt werden. Punkt A wird dann in EP^- der Ebene h direkt eingetragen und aus dem Gitter entfernt. Weil Punkt A entfernt wurde, wirkt MaxG so, dass die Eigenschaftsinformation von den Punkten aus M entfernt und zu den nach innen liegenden Punkten wieder eingetragen wird. Insgesamt werden bei der adaptiven Verfeinerung von h auf h-1 sowohl der Punkt A als auch die Punkte aus M als nicht mehr die Verfeinerungseigenschaft erfüllenden Punkte angesehen. Damit werden der Punkt unter A und seine 8 umliegenden Punkte entfernt. Und genau das soll mit dem rekursiven Entfernen erreicht werden.

Bemerkung: Um einen Gitterpunkt zu entfernen, entweder direkt oder per Kommunikation, muss der Grobgitterpunkt darüber auf dem eigenen Kern liegen. Ansonsten wird diese Entscheidung ggf. auf einem Kern getroffen, der nicht über genug EP-Informationen verfügt. An dieser Stelle aber würde damit die rekursive Verfeinerung blockiert, weil die entfernten Gitterpunkte Grobgitterpunkte der rekursiv zu entfernenden Gitterpunkte darstellen. Deshalb muss die Menge der adaptiv entfernten Gitterpunkte als Punktfeldliste an die nächst tiefere parallele adaptive Verfeinerung weitergegeben werden. Dann werden diese mittels des adaptiven Schemas entfernten Punkte bei der Entscheidung, die darunter liegenden Gitterpunkte zu entfernen, wie vorhandene Grobgitterpunkte behandelt, und die rekursive Entfernung ist erfolgreich.

Kapitel 6

Programmierung des Lastausgleiches

Zuerst gibt es eine kurze Übersicht. In den Abschnitten 6.1 bis 6.6 wird die Berechnung der zu verschickenden Lastmengen erläutert, während es in den Abschnitten 6.7 bis 6.9 um den Transport der Last in Form von Gitterpunkten geht. Der Abschnitt 6.8 ist der eigentliche Abschnitt zur Migration, während in den anderen Abschnitten unterstützende Klassen zur Migration behandelt werden.

6.1 Die Grundprinzipien dieses Lastausgleiches

6.1.1 Die Kerntopologie des Lastausgleiches

Die Durchführung des Lastausgleiches beginnt bei der Wahl der Kerntopologie für den Lastausgleich. Daher wird das als erstes diskutiert.
Bei $n_p = n_x\ n_y$ vielen Kernen ($\{0,\cdots,n_x\text{-}1\}$ x $\{0,\cdots,n_y\text{-}1\} \cong \{0,\cdots,n_p\text{-}1\}$) wird das Grundgitter (die Verfeinerungsgitter beginnen eine Ebene darunter) in n_p viele Teilgitter aufgeteilt, vgl. Abbildung 6.1. Von der Gitteraufteilung kommt man eigentlich auf eine Kerntopologie für den Lastausgleich, entsprechend der Abbildung 6.2. Nur wird mit dieser Topologie selber nicht gearbeitet. Man macht einen West-Ost- und einen Nord-Süd-Lastausgeich, und hat dafür die Kerntopologien der Abbildungen 6.3 und 6.4. Für die West-Ost-Kerntopologie in Abbildung 6.3 nimmt man die Verbindungskanten von Abbildung 6.2 in Ost-West-Richtung, für die Nord-Süd-Kerntopologie in Abbildung 6.4 nimmt man die Verbindungskanten der Abbildung 6.2 in Nord-Süd-Richtung.

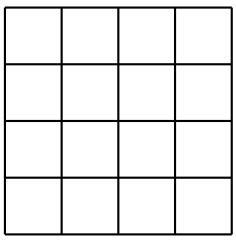


Abbildung 6.1: Die Anfangs-Gitteraufteilung des Grundgitters bei 16 Kernen

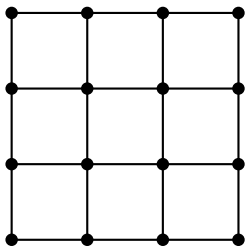


Abbildung 6.2: Die Lastausgleichs-Kerntopologie

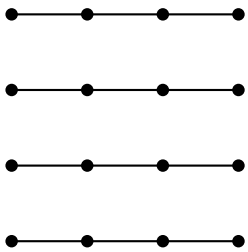


Abbildung 6.3: Die Ost-West-Kerntopologie

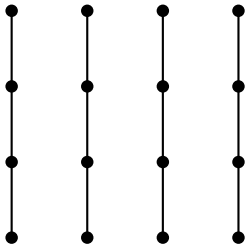


Abbildung 6.4: Die Nord-Süd-Kerntopologie

a	a	a	a
b	b	b	b
c	c	c	c
d	d	d	d

Abbildung 6.5: Die Lastverteilung nach einem vollständigen OW-Lastausgleich

6.1.2 Wie wird der Lastausgleich erreicht?

Man hat einen West-Ost-Lastausgleich und einen Nord-Süd-Lastausgleich. Für einen kompletten Lastausgleich müssen beide erfolgen. Das Grundprinzip dazu erläutern wir für den West-Ost-Lastausgleich in diesem Abschnitt 6.1. Der Nord-Süd-Lastausgleich erfolgt in gleicher Weise, wird aber in diesem Abschnitt nicht behandelt. Aber in den Abschnitten 6.5 und 6.6 kommen beide Lastausgleiche in jedem vorgestellten Algorithmus vor.

Der Ost-West-Lastausgleich

Das Grundprinzip des Lastausgleichs besteht darin, erst die Lastmengen zu bestimmen, die verschickt werden sollen, gefolgt vom Verschicken dieser Lasten.
Ein effizientes Verfahren dieser Vorgehensweise ist das in [19] entwickelte, in [63] weiterentwickelte und PLB (Precomputation-based Load Balancing) bzw. LBC (Load Balancing Classes in [63]) genannte Verfahren, mit dem ein vollständiger Lastausgleich erzielt werden kann.
Wie in Abschnitt 2.1.1 erläutert, führt eine unausgeglichene Last zu Ineffizienzen.

Global versus local decision und global versus local migration

In [19] wird die Klassifizierung von Lastausgleichsalgorithmen durch local oder global decision bzw. local oder global migration benannt, die auf [62] zurückgeht.
Zu global versus local decision: Bei global decision hat man die komplette Information, kann also prinzipiell einen kompletten Lastausgleich herbeiführen. Bei local decision wird die Lastentscheidung unter begrenzter Information getroffen, wonach der Lastausgleich nicht vollständig sein kann, und damit nach Satz 2.2 zu wesentlichen Effizienzverlusten führt. In diesem Sinne ist PLB ein global decision Verfahren, während die Dimension Exchange Methode, bei der ein Austausch der Lastinformationen nur mit dem Nachbarkern erfolgt, ein Beispiel ist für ein local decision Verfahren. Es sei denn man verwendet als Kerntopologie die des Hypercubes. Man muss hier aber noch anmerken, dass durch den Ost-West-Lastausgleich allein kein vollständiger Lastausgleich hergestellt werden kann, sondern nur bezüglich der Kerne, die untereinander in der Ost-West Kerntopologie verbunden sind. Insgesamt erfolgt aber durch einen vollständigen Ost-West-Lastausgleich, gefolgt von einem vollständigen Nord-Süd-Lastausgleich, ein global vollständiger Lastausgleich. Man vergleiche dazu Abbildung 6.5, die die Lastverteilung nach einem vollständigen Ost-West-Lastausgleich angibt.
Global migration versus local migration: Es stellt sich die Frage, mit welchem Kern ein Kern Last austauschen kann: Entweder erfolgt das nur zu Nachbarkernen bzgl. der jeweiligen Kerntopologie, was local migration genannt wird, oder zu weiter entfernten Kernen, was dann mit global migration bezeichnet wird. Bei den hier vorgestellten Verfahren liegt, wie bei PLB, local migration vor. Last weiter zu transportieren wäre

wünschenswert, würde aber weitere Implementationen erfordern.

Man geht für den Ost-West-Lastausgleich in folgenden Schritten vor:

1. Bestimme die Durchschnittslast der Kerne bzgl. der Ost-West-Kerntopologie.

2. Daraus leite man ab, wie viel Last zu den Nachbarn bzgl. der Ost-West-Kerntopologie fließen muss, bzw. von ihnen wegfließen muss, um einen vollständigen Lastausgleich zu erreichen, und zwar mittels der Partialsummenbildung. Das geschieht in der Precomputations-Phase.

3. Man durchlaufe eine oder mehrere Lastausgleichsrunden. In einer Lastausgleichsrunde passiert folgendes: Es wird von der in der Precomputation-Phase berechneten, zu verschickenden Last, immer so viel Last verschickt, wie möglich ist. Das macht man solange, bis für den jeweiligen Kern genauso viel Last verschickt wie berechnet wurde. Diese Schritte nennt man die Balancing-Phase.
   Das Verschicken der Last nennt man Migration. Dabei werden entsprechend der zu verschickenden Lastmenge Gitterpunkte ausgewählt (Klasse Strategie, Abschnitt 6.7).

Bemerkung 1: Nach Abschnitt 6.7.6 erfolgt der Lastausgleich entweder nur für eine Gitterebene, oder man führt mithilfe der Datenstruktur *ev2* gleichzeitig für zwei Gitterebenen einen Lastausgleich durch. In letzterem Fall erreicht man einen Lastausgleich auf **allen** Gitterebenen. Im letzteren Falle müssen auch die PLB-Berechnungen für zwei Gitterebenen erfolgen.

Bemerkung 2: Der oben dargestellte Fall ist nicht der allgemeinste Fall, weil gegebenenfalls noch eine Entscheidung dazukommt, welcher der Lastausgleiche in Nord-Süd- oder Ost-West-Richtung als erster erfolgen soll, vgl. die Algorithmen 6.12 und 6.13.

Bemerkung 3: Die Vorgehensweise folgt hier der Arbeit in [19], wobei dort ein kombinatorisches Problem parallelisiert wurde.
a) Diese Strategie ist ungewöhnlich für ein adaptives Mehrgitterverfahren, wo die Lastaufteilung typischerweise mithilfe eines Bisektionsverfahrens erfolgt, vgl. z.B. [36].
b) Es gibt bei dieser Arbeit Abweichungen zur Arbeit in [19], was vor allem daran liegt, dass neben PLB eine PLB-Variante dargestellt wird. Die Darstellung weicht von der in [19] ab, damit beide Methoden gemeinsam dargestellt werden können, indem die Precomputation Methode aufgegliedert wurde in allgemeine Überlegungen (Sätze 6.1 und 6.2) und die konkrete Partialsummenberechnung.

Bemerkung 4: Die Balancing-Phase kann aufgeteilt werden in eine Phase, wo die Größen der real zu verschickenden Lastmengen (genannt Lasttransfermengen) über alle Runden bestimmt werden, gefolgt von der eigentlichen Migration entsprechend dieser Zahlen, vgl. dazu z.B. den Algorithmus 6.2.

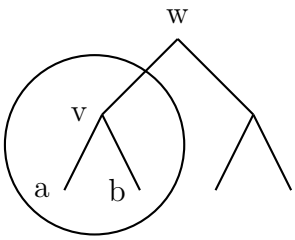


Abbildung 6.6: Der an v wurzelnde Teilbaum von B

6.2 Precomputation und Balancing

6.2.1 Die allgemeinen Vorüberlegungen zur Precomputations-Phase

Definition 6.1: Ein wurzelnder Teilbaum wT(v) und die Partialsumme PS(v) und die Knotenteilanzahl KTA(v) eines Knoten v, vgl. Abbildung 6.6.

Gegeben sei ein Wurzelbaum B in einem Graphen G mit der Wurzel r. Es sei eine Lastverteilung auf dem Graphen gegeben, d.h. jedem Knoten v sei eine Last L(v) zugeordnet. Nun sei v ein beliebiger Knoten des Graphen.

Dann ist mit wT(v) der an v wurzelnde Teilbaum von B gemeint.
Dann ist mit der Partialsumme des Knoten v, sie sei bezeichnet mit PS(v), die Summe aller Lasten der Knoten von wT(v) gemeint, d.h. es gilt: PS(v) = $\sum_{w \in wT(v)}$ L(w).
KTA(v) ist die Anzahl der Knoten von wT(v). Es ist: KTA(v) = $\sum_{w \in wT(v)}$ 1.

Damit können wir den perfekten Lastausgleich angeben, wenn man außer Acht lässt, dass Knoten maximal ihre vorhandene Last verschicken können:

Satz 6.1: Man kann berechnen, wie viel Last jeweils verschickt werden muss für einen vollständigen Lastausgleich. Diese Berechnung wird im Folgenden angegeben, als erster Schritt der Precomputation-Methode.

Gegeben sei ein Graph G, ein Wurzelbaum B mit einer Wurzel r.
Dann kann zu jeder gerichteten Kante e = (v,w) des Baumes die zu verschickende Last vL(e) angegeben werden, sodass ein Lasttransfer entsprechend dieser vL's zu einem perfekten Lastausgleich führen würde. Bei vL > 0 wird die Last von v nach w, bei vL < 0 von w nach v geschickt. Es gilt ferner vL(v,w) = -vL(w,v). Dabei wird entsprechend den Werten von vL ggf. mehr Last verschickt als ein Knoten hat, sodass man ggf. mehrere Migrationsrunden benötigt, um wirklich einen perfekten Lastausgleich herzustellen.
Liegt w näher an der Wurzel, dann werden für die Berechnung von vL(e) die Werte PS(v) und PS(r) benötigt.

Beweis: Man hat die Gesamtlast des Graphen GL(G) = PS(r). Damit hat man auch die durchschnittliche Last eines Knoten v ∈ V als $\phi = \frac{PS(r)}{|B|}$ mit |B| = Anzahl der Knoten des Baumes. Sei nun ein Knoten v ≠ W ausgewählt, und sei e seine Kante in

Richtung der Wurzel und w der von v über e erreichte Knoten. Man berechne vL über die Formel vL(e) = PS(v) - ϕ(v) * KTA(v). Links in der Formel zu vL(e) steht die tatsächliche Last des an v wurzelnden Teilbaums von B, und rechts abgezogen wird die ideale Last des an v wurzelnden Teilbaums von B.
Benötigt werden hier nur PS(v) und PS(r), da KTA(v) und |B| fest liegende Informationen des sich nicht ändernden Graphen sind.

Bemerkung zum Vorzeichen von vL(e):
Ist vL(e) $>$ 0, dann ist die tatsächliche Last PS(v) des an v wurzelnden Teilbaums größer als die ideale Last. Die Lastmenge vL(v) muss über die Kante e von v nach w fließen, um einen Lastausgleich herzustellen.
Ist vL(e) $<$ 0, dann ist die tatsächliche Last PS(v) des an v wurzelnden Teilbaums kleiner als die ideale Last. Die Lastmenge -vL(v) muss über die Kante e von w nach v fließen, um einen Lastausgleich herzustellen.

Satz 6.2: Lokal bestimmter vollständiger Lastausgleich:

Seien wieder G,|B| und r wie oben gegeben. Wenn man zu jedem Knoten v seine Last L(v), sowie für seine direkt benachbarten Knoten bis auf den Knoten in Richtung der Wurzel $v_1, \cdots, v_l$ die PS(v_i) kennt, und außerdem noch PS(r), dann kann der Kern des Knotens v die zu verschickende Last vL für seine eigenen Kanten berechnen. Damit wird ein vollständiger Lastausgleich berechnet, mit der Einschränkung, dass die bzgl. der vL aller Kanten zu verschickende Last größer sein kann als die vorhandene Last.

Beweis: Wegen der Kenntnis der Werte PS(v_i) sowie PS(r) kann wie in Satz 6.1 angegeben vL für die Kanten zwischen v und v_i berechnet werden. Falls v die Wurzel ist, ist man damit fertig. Ansonsten bleibt es übrig, für die Kante Richtung Wurzel vL zu berechnen. Das ist möglich wegen der Kenntnis von PS(v) = $\sum_{i=1\cdots l}$PS(v_i) + L(v) sowie von PS(r) durch Anwendung von Satz 6.1.

Folgerung: Man muss nur die in der Bedingung von Satz 6.2 nötigen Partialsummen kennen, und kann damit das für den Lastausgleich nötige vL bestimmen. Nun gibt es dazu die ursprüngliche PLB-Precomputation-Methode, diese Partialsummen zu berechnen, und eine vom Autor - basierend auf seinen Kenntnissen über PLB - entwickelte PLB-Precomputation-Variante, diese Partialsummen zu berechnen.

Es werden beide Methoden im Folgenden vorgestellt, allerdings nur in Bezug auf die Struktur des linearen Arrays, weil nur diese Struktur, vgl. die Abbildungen 6.3 und 6.4, hier benötigt wird. Deswegen wird für eine Anwendung von Satz 6.2 nur für jeden Knoten v der Wert der Partialsummen für einen Vorgängerknoten, dass ist sein direkt benachbarter Wegknoten, und PS(r) sowie seine Last L(v) benötigt.

6.2.2 Die Partialsummenberechnung beim PLB-Verfahren für das lineare Array.

Die Auswahl der Wurzel: Bei n Knoten 0, $\cdots$, n-1 liege die Wurzel beim Knoten (n-1)/2, wobei diese Zahl abgerundet oder aufgerundet sein kann. Bei einer geraden

Zahl n wäre das $\frac{n}{2}$-1 oder $\frac{n}{2}$. So wird die Kommunikation stärker nebenläufig, als wenn man die Wurzel mehr in Richtung eines Blattes verschiebt. Denn im Prinzip gibt es zwei Kommunikationsrichtungen, einmal von dem einen Blatt bis zur Wurzel, und parallel dazu vom anderen Blatt bis zur Wurzel. Diese Abfolgen von Kommunikationen werden gleich vorgestellt.

Die Bestimmung von PS(v) für den Vorgängerknoten: Es wird nun beschrieben, wie man die Partialsummen für die Knoten 0 bis zur Wurzel bestimmt. Vom Knoten n-1 bis zur Wurzel geht es in derselben Weise.
Dabei hat jeder Knoten maximal zwei Nachbarknoten, einen links, und einen rechts. Die Partialsumme PS_l ist bezogen auf den linken Knoten, PS_r ist dieser Wert zuzüglich der eigenen Last, die an den rechten Knoten weitergegeben werden muss.

1. Für Knoten 0 ist PS_l 0, und PS_r seine eigene Last. Dieser Wert wird an Knoten 1 verschickt.

2. Der Knoten i ≥ 1 empfängt den Wert PS_l von Knoten i-1 und addiert seine Last dazu, erhält damit PS_r, und schickt das an Knoten i+1 $\leq$ (n-1)/2.

Für die Knoten von n-1 bis zur Wurzel geht es spiegelbildlich, d.h. PS_l ist die Partialsumme mit der eigenen Last, und PS_r die Partialsumme ohne die eigene Last.

Die Berechnung von PS(r): Nachdem diese Schritte durchgeführt wurden, kommen zwei Partialsummen bei der Wurzel W an. Diese berechnet den Wert PS(r) = PS_l (von rechts) + PS_r (von links) + L(W). Dieser Wert wird an alle anderen Knoten des linearen Arrays verschickt mittels einer Broadcast-Kommunikation. Mit PS_l, PS_r und PS(r) kann dann jeder Knoten, ob er links oder rechts der Wurzel liegt, oder die Wurzel ist, gemäß Satz 6.2 seine vL-Werte berechnen.

6.2.3 Die Partialsummenberechnung bei der PLB-Variante

Die PLB-Variante wurde vom Autor entwickelt, basierend auf seinen PLB-Kenntnissen. Die Theorie aus Abschnitt 6.2.1 bleibt erhalten. Der Unterschied ist der, dass bei der PLB-Variante im Voraus Lasten anderer Kerne kommuniziert werden. Man geht in 2 Schritten vor:

1. Man bestimmt für jeden Kern die Lasten aller Kerne in Ost-West-Richtung. In einer Variante der Variante bekommt jeder Knoten sogar die Lastwerte aller anderen Kerne mitgeteilt.

2. Man berechnet damit die Partialsummen ohne weitere Kommunikation.

Die Bestimmung der Lasten der anderen Kerne

Zuerst wird das lineare Array behandelt.

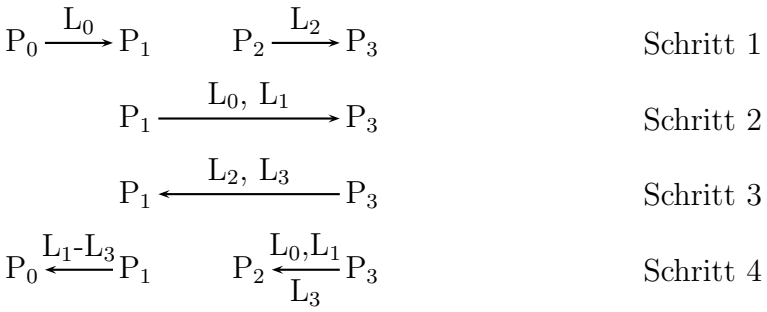


Abbildung 6.7: Lastwerte kommunizieren

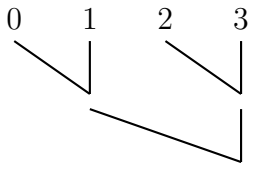


Abbildung 6.8: Der Baum zu Abbildung 6.7

Die Kommunikation über einen vollständigen Graphen. Hier schickt jeder Kern seinen Lastwert an jeden anderen Kern - in der Ost-West- oder Nord-Süd-Kerntopologie. Für bis zu insgesamt 16 Kerne ist diese Kommunikation sinnvoll, wenngleich es auch bei einer höheren Anzahl von Kernen eine mögliche Methode darstellt. In Ost-West oder Nord-Süd Richtung liegen dann 4 Kerne nebeneinander, und der Aufwand wäre 3 Mal senden und 3 Mal empfangen pro Kern.

Die Kommunikation der Lastwerte über einen Baum. Es gibt jedoch auch eine Alternative, die in dieser Arbeit Kommunikation über einen Baum genannt wird, die für n = 2^k Kerne die Werte der anderen Lastmengen ermittelt. Zuerst zwei Abbildungen (Abbildung 6.7 und Abbildung 6.8) dazu bei 4 Kernen:

Zuerst die Beschreibung: Bei 2^k Kernen hat man 2 * k Kommunikationsrunden. Bei der ersten Kommunikationsrunde schickt der erste Kern seinen Lastwert an den zweiten, der dritte an den vierten, usw.. Für die nächste Linie werden die Kerne ausgewählt, die Lastwerte empfangen haben. Unter diesen schickt wieder der erste seinen Lastwert an den zweiten, der dritte an den vierten, usw.. Das macht man, bis es auf Linie k zu einer einzigen Kommunikation kommt. Dann hat der Empfänger dieser Information die Lastwerte aller Kerne. Indem man nun in umgekehrter Reihenfolge noch nicht vorhandene Lastwerte zurück kommuniziert, haben am Ende alle Kerne alle Informationen. Man braucht dazu $\log_2$(Anzahl Kerne) * 2 viele Schritte.

Bemerkung 1: Bei den Kommunikationsrunden k+1 bis 2k des Kommunikationsschemas wie in Abbildung 6.7 tritt die folgende Schwierigkeit auf: Man muss sich getrennt für jeden Fall überlegen, welche Lastwerte der jeweilige Kern schon hat, um ihm nicht schon bekannte Lastinformationen mitzuschicken.

Bemerkung 2: Sollen die Lasten von allen Kernen für alle Kerne bestimmt werden, macht man diese Schritte erst für die Ost-West-Linien und dann für die Nord-Süd-

Linien, wobei im zweiten Schritt die Buffer jeweils statt einem Lastwert alle Lastwerte der jeweiligen Ost-West-Linie übertragen.

Bemerkung 3: Welche der beiden Kommunikationen ist besser? Bei kleinen Kernanzahlen pro Kerntopologie-Linie sind beide Verfahren schnell. Ggf. ist dann die Kommunikation über einen vollständigen Graphen besser, weil in dem Fall alle Kommunikationen im Wesentlichen gleichzeitig ablaufen. Bei größeren Kernanzahlen ist hingegen die *all to all*-Kommunikation unpraktisch, weil sie dann richtig aufwendig wird und auch das Kommunikationsnetz erheblich belastet.

Die Berechnung der Partialsummen bei der PLB-Variante

Gegeben sei ein lineares Array zu den Knoten 0 bis n-1. n-1 wird zur Wurzel bestimmt. Dann legt man ein Feld a der Länge n an. a(0) = L(Kern 0), und es wird gerechnet:
for i = 0 **to** n-2
    a(i+1) = a(i) + L(Kern i+1).
Dann kann jeder Knoten k die benötigten Partialsummenwerte direkt ablesen:
PS(ohne die eigene Last) = a(k-1) für k $\geq$ 1, für k=0 ist PS(ohne eigene Last) = 0.
PS(mit der eigenen Last) = a(k)
PS(r) = a(n-1)
Die restlichen Schritte werden nach Satz 2 durchgeführt. Es erfolgt hierbei kein Broadcasting, weil jeder Kern a(n-1) direkt ablesen kann.

Bemerkung: Man kann hier auf jedem Kern vL für alle Kanten bzgl. der Kerntopologie berechnen, d.h. man hat diese Information nicht nur lokal für den eigenen Kern und den linken und rechten Nachbarn.

6.2.4 Die Balancing-Phase - beim normalen PLB

Das Precomputation-Verfahren reicht zur Lastberechnung aber noch nicht aus, weil ggf. mehr Last über eine Kante fließen müsste, als der zu verschickende Knoten hat. Zuerst aber der Fall, dass man nur eine Lastausgleichsrunde macht:

Algorithmus 6.1: Balancing - eine Runde, keine Simulation
Input: Zum Knoten v die Werte von vL auf seinen Kanten

1. **for all** Kanten e von v:
2. **if** vL(e) $>$ 0 **then**
3. a. Schicke eine Last von min(L(v),vL(e)) an über die Kante e, und
   b. Verringere vL um diese Lastmenge
4. **end if**
5. **if** vL(e) $<$ 0 **then**
6. a. Empfange die Last
   b. Erhöhe vL(e) um den Wert der angekommenen Last.

Kern	0	1	2	3	4
Last A	0	0	0	0	100
Last B	0	0	100	0	0

Abbildung 6.9: Lastsituation für Kern 0: Ausgleich zu verschiedenen Zeitpunkten

7. **end if**

8. **end for**

Bemerkung 1: Zu Schritt 6b: Dabei bleibt vL(e) negativ, verringert aber seinen Absolutbetrag, wenn Last ankommt.

Bemerkung 2: Die Schritte 3b/6b werden bei einer einzigen Lastausgleichsrunde nicht gebraucht.

Macht man aber mehrere Migrations-Runden, so wie sie im Folgenden erklärt werden, dann kommt es zum vollständigen Lastausgleich. Dafür führt man Algorithmus 6.1 so lange aus, bis vL(e) = 0 ist für alle Kanten. Das ist die Balancing Methode in [19]. Dabei kommt es zu einer 'all to all'-Kommunikation in Bezug auf die Überprüfung vL(e) = 0 für alle Kerne, damit alle Kerne dieselbe Anzahl an Lastausgleichsrunden durchführen. Das bezieht sich nicht nur auf die Kerne einer Kerntopologie-Linie. Also müssen ggf. auch noch Leerrunden eingefügt werden. Damit wird erreicht, dass alle Kerne die Zwischenkorrektur gemeinsam ausführen.
Dass der Algorithmus konvergiert wurde in [19] bewiesen. Es ist auch klar, weil alle Überlast irgendwann auf 0 reduziert wird.

Bemerkung 1: In Schritt 6 bekommt man die Information, um wie viel vL zu erhöhen ist, entweder über den Balancer, oder eine separat auszuführende Kommunikation.

Bemerkung 2: Man weiß bei mehreren Runden nicht, wann die entsprechenden Lasten ankommen, vgl. Abbildung 6.9. Für Kern 0 muss in den Fällen A und B dieselbe Lastmenge von 20 über seine Kante fließen. Er bekommt diese aber zu unterschiedlichen Zeitpunkten. Das hat zur Folge, dass es für den jeweiligen Kern mit nur lokaler Information unbestimmt ist, wie viele Lastausgleichsrunden erfolgen müssen (höchstens n-1). Das wiederum ist nötig zu wissen wegen der Zwischenkorrektur bei der Migration, vgl. Abschnitt 6.8.9, weil für die Zwischenkorrektur alle Kerne dieselbe Anzahl von Runden Lastausgleich machen müssen. Denn die Zwischenkorrektur muss immer für alle Kerne gemacht werden, selbst wenn dabei keine Last übertragen wird. Bei der ursprünglichen Balancing Methode von 6.1 müsste nach jeder Balancing-Runde eine All-Reduce-Kommunikation, vgl. Abkürzungsverzeichnis, aller Kerne (von allen Ost-West-Linien) erfolgen, um zu ermitteln, ob weitere Runden erfolgen müssen.
Deswegen hat man bei einer Vorausberechnung, vgl. Algorithmus 6.2, den Vorteil, erst die Anzahl an Runden zu bestimmen, und dann diese Zahl in einem Schritt zu übermitteln, als das nach jeder Runde zu machen.

Bemerkung 3: Der Wert min(L(v),vL(e)), der über die Kante geschickt wird, soll Last-

transferwert (LTW) genannt werden.

Man kann die Berechnung des LTW's und die Migration getrennt voneinander durchführen:

Algorithmus 6.2: Balancing für eine Runde mit Simulation
Input: Für Knoten v die Werte von vL auf seinen Kanten

1. **for all** mit v inzidenten Kanten e = (v,w)
2. **if** vL(e) > 0 **then**
3. a. Schicke Lasttransferwert LTW(e) = min(L(v),vL(e)) an w
   b. Verringere vL um diese Lastmenge
   c. Speichere LTW(e)
4. **end if**
5. **if** vL(e) < 0 **then**
6. a. Empfange LTW(e) von w
   b. Erhöhe vL(e) um den Wert der angekommenen Last.
   c. Speichere LTW(e)
7. **end if**
8. **end for**
9. **for all** mit v inzidenten Kanten e = (v,w)
10. **if** LTW(e) > 0 **then** Verschicke die Last LTW(e) an den Kern w.
11. **else** Empfange die Last von Kern w.
12. **end if**
13. **end for**

Die Berechnung erfolgt in den Schritten 1-5, die Migration in den Schritten 6 und 7.

Die Schritte 1-5 werden im Folgenden Balancing-Simulation oder Balancing-Vorberechnung genannt, weil nur die LTW-Werte bestimmt werden, ohne eine Migration durchzuführen.

Bemerkung: Bei einer Lastausgleichsrunde können die Schritte 3b/3c/6b/6c entfallen.

Eine Erweiterung des Algorithmus 6.2 um den Fall, dass für einen vollständigen Lastausgleich genügend viele Lastausgleichsrunden erfolgen, ist wie folgt möglich:

Algorithmus 6.3: Balancing für mehrere Runden mit Simulation

Man iteriert die Schritte 1 bis 5 solange, bis überall vL = 0 ist, wobei man die LTW-Werte zu jeder Runde abspeichert. Dann führt man **eine** all to all Kommunikation der

Anzahl der Lastausgleichsrunden aus. Danach migriert man entsprechend den Schritten 6 und 7 für alle für den eigenen Kern notwendigen Migrationsrunden. Man macht am Ende so viele Leerrunden, d.h. Migration ohne Lasttransfer, sodass alle Kerne dieselbe Anzahl an Lastausgleichsrunden haben, was für die Zwischenkorrektur nötig ist.

6.2.5 Die Balancing-Phase - bei der PLB-Variante

Algorithmus 6.4: Balancing für eine Runde der PLB-Variante

Die Schritte erfolgen genauso wie bei Algorithmus 6.2, nur dass die LTW-Werte nicht verschickt, sondern abgelesen werden.

Algorithmus 6.5: Balancing für mehrere Runden bei der PLB-Variante

Sollen mehrere Lastausgleichsrunden gemacht werden, so werden die Schritte 1 bis 5 des Algorithmus 6.2 rundenweise durchgeführt, bis für alle Knoten der eigenen Ost-West-Linie vL = 0 gilt. Dabei ist die Lastsituation nach jeder Runde für alle Knoten der Ost-West-Linie bekannt. Man braucht dann keine weiteren Kommunikationen mehr, um an alle LTW-Werte jeder Runde zu kommen.
Man beachte, dass man bei dieser Methode alle Lasttransferwerte für den Knoten zum eigenen Kern für alle Runden speichern muss.
Am Ende jeder Runde wird vL = 0 auf allen Kanten getestet. Wenn man so viele Balancing-Runden macht, erfolgt ein vollständiger Lastausgleich. Dabei muss auch die maximale Anzahl der Lastausgleichsrunden aller Kerne bestimmt werden. Also werden dann so viele Migrationsrunden entsprechend den abgespeicherten LTW-Werten durchgeführt, bzw. es erfolgen zusätzliche Leerrunden.

Bemerkung 1: Für die Zwischenkorrektur: Bei der Kommunikation am Ende der Berechnungen über die Zahl der Lastausgleichsrunden ist keine All-Reduce-Kommunikation in Bezug auf alle Kerne mehr nötig, da jeder Kern der Linie weiß, nach wie vielen Runden die Last bzgl. dieser Linie ausgeglichen ist. Deshalb muss für beliebig viele Runden nur eine einzige All-Reduce-Kommunikation für die verschiedenen Linien erfolgen, unter Angabe der Anzahl der Lastausgleichsrunden. Man kann das hier als eine All-Reduce-Kommunikation bzgl. eines Graphen auffassen, für den jede Ost-West-Linie einem Punkt entspricht. Wenn man beispielsweise 16 Kerne hat und 4 Ost-West bzw. 4 Nord-Süd-Linien, dann muss statt einer Kommunikation zu 15 Kernen nur noch eine zu 3 Kernen erfolgen, und das nur einmal für einen vollständigen Ost-West Lastausgleich. Implementiert ist allerdings bei der PLB-Variante für mehrere Lastausgleichsrunden nur die Version, bei der vorher die Lastwerte aller Kerne gesammelt werden (Matrix).

Bemerkung 2: Wenn man die Lastwerte aller anderen Kerne kennt, kann man das Balancing aller Ost-West und Nord-Süd-Linien simulieren, und braucht dann zur Bestimmung der Lastausgleichsrunden der anderen Kerne gar nicht mehr zu kommunizieren.

Bemerkung 3: Man kann die Anzahl der Lastausgleichsrunden aus der Kenntnis der vL-Werte bestimmen, vgl. [29]: Sie beträgt $\lceil \frac{max\{vL\}}{\overline{vL}} \rceil$ mit $\overline{vL}$ ist der Durchschnitt der vL.

6.2.6 Ein Vergleich zwischen PLB und der PLB-Variante

Zuerst der Vergleich von PLB zur PLB-Variante, wo die Bestimmung der Lastwerte der anderen Kerne durch eine all-to-all-Kommunikation erfolgt, bei geringen Knotenzahlen.

1. Der Aspekt 'Anzahl an Kommunikationen':
   Bei der PLB-Methode hat man weniger Kommunikationen, die aber nacheinander von den Blättern zur Wurzel erfolgen, wo ein Kern auf den anderen warten muss. Wegen Synchronisationen im Anschluss an jede Lastausgleichsrunde kann während des Lastausgleiches dieses Warten nicht kompensiert werden. Denn während der Precomputations-Phase kann keine andere Arbeit geleistet werden. Bei der PLB-Variante sind es mehr Kommunikationen, die aber alle (fast) gleichzeitig erfolgen können. Das Kommunikationsnetz wird aber stärker ausgelastet.

2. Der Aspekt 'Länge der Kommunikation':
   In beiden Fällen werden während der Precomputations-Phase jeweils nur eine (einfacher Lastausgleich) oder zwei (reverser Lastausgleich) Lastwerte verschickt, vgl. den Abschnitt 6.7.6 zum reversen Lastausgleich.

Als nächstes der Vergleich von PLB zur PLB-Variante mit der Kommunikation der Lastwerte der anderen Kerne über einen Baum.

1. Der Aspekt 'Länge der Kommunikation': Bei der PLB-Methode wird jeweils nur ein Lastwert (bzw. 2 bei auf 2 Gitterebenen gepackten Lasten) übermittelt. Bei der PLB-Variante sind die Buffer größer. Die genaue Buffergröße ist aber schwierig zu bestimmen, wenn man in diesem Fall nur Last-Informationen verschickt, die dem Empfänger noch nicht bekannt sind.

2. Der Aspekt 'Anzahl an Kommunikationen': Als Kompensation für den Nachteil beim ersten Aspekt hat man bei der PLB Methode n/2 viele Kommunikationen nacheinander, während das in diesem Fall nur 2 $\log_2$ n viele Kommunikationen nacheinander sind. Es läuft hier bei der PLB-Variante sehr viel mehr parallel ab. Da es hinterher zu Synchronisationen kommt, können die Warte-Schritte nacheinander bei der PLB-Methode nicht für andere Arbeiten genutzt werden.

Bemerkung zum ersten Aspekt: Die Buffergröße spielt bei kleinen Werten für die Kommunikation kaum eine Rolle. Erst bei sehr viel größeren Buffern wird das relevant, und dieser Fall tritt erst ein, wenn viele Kerne Arbeit verrichten. Man muss dabei die Latenzzeit, das ist die Kommunikationszeit für eine Kommunikation ohne Übertragung, in Beziehung setzen zur Bandbreite mal Buffergröße, was angibt, wie viel zusätzliche Zeit die Übertragung eines Buffers einer gewissen Länge braucht. Bei einer kleinen Buffergröße spielt letzterer Wert im Allgemeinen eine untergeordnete Rolle.

Weitere Vorteile der PLB-Variante: Bei beiden PLB-Varianten entfällt noch die Broadcasting Kommunikation am Ende.

Außerdem fallen Kommunikationen beim Balancing weg, bzw. sie werden reduziert, die beim normalen PLB nötig sind, damit alle Kerne dieselbe Anzahl an Balancing-Runden ausführen, was wegen der Zwischenkorrektur nötig ist. Diese Zeitersparnis

entfällt jedoch, wenn von vornherein nur eine Balancing-Runde durchgeführt werden soll.

Betreibt man den Aufwand, alle Lastmengen allen Kernen zur Verfügung zu stellen, gibt es noch weitere Möglichkeiten der Programmierung. Man kann Entscheidungen treffen, die sich positiv auf das Laufzeitverhalten auswirken können, vgl. den Abschnitt 6.6.

6.3 Die unterschiedlichen Möglichkeiten PLB zu nutzen

Im Folgenden werden einige Varianten von PLB und der PLB-Variante skizziert.
Man wird bei den anschließenden Algorithmen der Abschnitte 6.5 und 6.6 sehen, dass die Überlegungen dieses Abschnittes 6.3 hilfreich sind.

Normales PLB gegenüber der PLB-Variante

Bei der normalen PLB-Methode werden die Partialsummen direkt errechnet und jeweils durch Kommunikation weitergegeben, und dann wird für die Gesamtsumme ein Broadcast durchgeführt. Die Algorithmen dazu, die aus dem Aufruf einzelner Methoden bestehen, werden in Abschnitt 6.5 gebracht. Bei der Variante von PLB werden hingegen die Lasten der beteiligten Kerne bestimmt, und mit diesen Informationen die nötigen Partialsummen berechnet. Die Algorithmen dazu werden in Abschnitt 6.6 dargestellt. Vor- und Nachteile der Methoden wurden in Abschnitt 6.2.6 und werden in Abschnitt 6.6.2 für den Algorithmus 6.13 diskutiert.

Mit oder ohne Lastausgleich auf der oberen Gitterebene

Wie man im Abschnitt 6.7.6 sehen wird, kann man zum Lastausgleich zur Ebene h geschickt einen Lastausgleich für die Ebene darüber machen, der durch Schicken von Last in umgekehrter Richtung auch auf gröberen Gitterebenen einen Lastausgleich erreicht. Die normale Einstellung hier ist **mit** diesem Lastausgleich, die Alternative hat die Bezeichnung **ohne**. Eine Verwendung **mit** diesem Lastausgleich wird auch reverser oder umgekehrter Lastausgleich genannt. Beide Fälle treten bei allen Algorithmen der Kapitel 6.5 und 6.6 auf.

Eine Lastausgleichsrunde im Vergleich mit den zum vollständigen Lastausgleich benötigten Lastausgleichsrunden

Hier wird herausgestellt, dass es diese Unterscheidung gibt. Nun wird ausgeführt, was dafür und dagegen spricht, die eine oder andere Entscheidung zu treffen.
Wenn sich die Last auf einen Kern konzentiert, kann ein direkter Lastausgleich nur zu seinen Gitternachbarn erfolgen, also ohne die Last auf alle Kerne zu verteilen bei hinreichend vielen Kernen. In dem Fall ist es sinnvoll, mehrfach die Migrationen durchzuführen, wodurch die Last auch zu entfernteren Kernen gebracht werden kann.

Wenn sich die Lastsituation nicht zu sehr ändert, reicht hingegen eine Lastausgleichs-

runde aus. Zu Beginn, oder bei komplexen Situationen, oder neu entstehenden entfernten Gebieten kann es sein, dass mehrfache Balancing-Runden erforderlich sind.

Option: all to all ($\mathbf{K}_n$) versus baumartige Informationsbestimmung bei der PLB-Variante

Bei der Variante des Autors gibt es 2 verschiedene Wege, die Lastinformation zu bekommen. Man kann z.B. in einer Linie eine all to all Kommunikation anstoßen, oder die Informationen auch über einen asymetrischen Baum sammeln, vgl. Abbildung 6.8. Auch hier wurde schon überlegt was besser ist, und wie aufwendig sich das im Vergleich mit dem normalen PLB gestaltet, vgl. die Abschnitte 6.2.6 und 6.2.3.
Diese Einstellung ist möglich für alle Algorithmen von Abschnitt 6.6.

Unterscheidung bei der PLB-Variante: Matrix versus Vektoren

Bei der PLB-Variante werden entweder die Lasten der Linien bzw. Reihen bestimmt (Vektoren), oder aber es werden die Lasten aller Kerne bestimmt (Matrix). Im letzteren Fall kann man zusätzliche Entscheidungen fällen, wie z.B. die, ob erst der Ost-West-Lastausgleich erfolgen soll, oder erst der Nord-Süd-Lastausgleich, oder auch nur einen der beiden Lastausgleiche durchzuführen. Diese Unterscheidung ist wesentlich zwischen den Algorithmen aus Abschnitt 6.6, und wird in den Algorithmen 6.12 und 6.13 genutzt.

Option: Gegebenenfalls keinen Lastausgleich durchführen

Ob man diese Option zur Verfügung stellen kann hängt davon ab, welche Informationen vorhanden sind. Bei der normalen Variante bzw. der PLB-Variante 'mit Vektoren' ist dieses nicht gut möglich: Die Kerne der Ost-West Lastausgleiche verschiedener Zeilen wissen nichts voneinander. Dass hier aber eigentlich eine gemeinsame Entscheidung zu treffen ist, ist wie folgt zu begründen.
Zum Beispiel kann sich der Lastausgleich für eine Zeile lohnen, für eine andere aber nicht. Es lohnt sich meistens aber nur, für alle Ost-West-Linien Lastausgleich zu machen oder aber für keine Ost-West-Linie. Der Lastausgleich nur für eine Ost-West-Linie kostet auch die Kerne der übrigen Linien Zeit, weil sie wegen Synchronisationen auf die Vollendung dieses Lastausgleiches warten müssen.
Es könnte sich natürlich lohnen, nur für eine Zeile oder Spalte von Kernen einen Lastausgleich durchzuführen, wenn dadurch eine sehr unausgeglichene Lastsituation bereinigt wird.

Hingegen sind bei der Matrix-Variante der PLB-Variante alle möglichen Informationen auch für solche Entscheidungen vorhanden. Natürlich müssen dann aber ggf. weitere Berechnungen erfolgen, z.B. darüber, mit welchem Effizienzverlust aufgrund des Aussetzens des Lastausgleiches zu rechnen ist. Für den Effizienzverlust in Bezug auf alle Kerne muss die Laufzeit aller Kerne berechnet werden.

6.4 Eine Übersicht über die Algorithmen aus 6.5 und 6.6.

Zuerst wird in Abschnitt 6.5 PLB wie in [19] präsentiert. Es beginnt mit dem Algorithmus 6.6, wo nur eine Lastausgleichsrunde durchgeführt wird. Dabei wird auch die Verwendung des simulierten Balancings erwähnt. In den übrigen 2 Algorithmen 6.7 und 6.8 wird PLB mit mehreren Lastausgleichsrunden durchgeführt, und zwar im ersten Fall ohne, im zweiten Fall mit einer Balancing-Vorberechnung.

Danach werden in Abschnitt 6.6 die Algorithmen der PLB-Variante dargestellt. Zuerst werden die Fälle mit einer Lastbestimmung nur von einzelnen Linien (Vektor) gebracht. Das geschieht bei dem Algorithmus 6.9 im Fall nur mit einer Lastausgleichsrunde.

Im Anschluss erfolgt die Lastbestimmung für alle Lastwerte (Matrix). Bei den Algorithmen 6.10 und 6.11 wird sie ohne Option vorgestellt, die Reihenfolge von Ost-West- und Nord-Süd-Lastausgleich aufgrund einer Beurteilung zu verändern, während bei den Algorithmen 6.12 und 6.13 diese Option besteht. In diesen 2 mal 2 Fällen wird dann im ersten Fall jeweils eine Lastausgleichsrunde, beim zweiten Fall werden hinreichend viele Lastausgleichsrunden durchgeführt.

Die Variable mit-ohne bezieht sich dabei darauf, den reversen Lastausgleich durchzuführen oder nicht. Dieser wird in Abschnitt 6.7.6 eingeführt.

6.5 Die vollständigen Algorithmen zum PLB-Verfahren

Hier werden die Algorithmen in Gänze dargestellt, einschließlich Ost-West- und Nord-Süd-Lastausgleich. Es wird aber nur gesagt, was zu tun ist, d.h. es wird hier nicht mehr auf die Details eingegangen.

Zunächst zwei Bemerkungen zu den in [19] vorgestellten Verfahren. Sie gelten für diesen Abschnitt als auch für Abschnitt 6.6.

1. Ob ein reverser Lastausgleich gemacht wird, wird durch die Variable mit-ohne gekennzeichnet.
   In dem Fall des reversen Lastausgleichs müssen die Partialsummen für beide Ebenen berechnet werden, und die Migration muss auch jeweils für zwei Ebenen erfolgen, weshalb auch die Balancing-Routine angepasst werden muss. Es muss dabei allerdings nicht unbedingt ein vollständiger Lastausgleich beider Ebenen angestrebt werden.

2. Der 'Algorithmus 2.3, Balancing(T, λ, δ) ' in [19], ist im Falle mehrerer Lastausgleichsrunden so nicht anwendbar, weil wegen der Zwischenkorrektur die Lastausgleichsrundenanzahl für alle Kerne gleich sein muss.

6.5.1 PLB mit nur einer Lastausgleichsrunde

Der Algorithmus gestaltet sich in diesem Fall wie folgt:

Algorithmus 6.6: PLB mit nur einer Lastausgleichsrunde
Input: mit-ohne: Das ist eine bool'sche Variable, die angibt, ob ein reverser Lastausgleich erfolgen soll, vgl. Abschnitt 6.7.6.

1. Führe einen Precomputations-Schritt durch für den Ost-West Lastausgleich(mit-ohne)
2. Führe die Migration gemäß den LTW-Werten aus Algorithmus 6.2 aus. (Dieser Schritt ist auch mit Algorithmus 6.1 möglich.)
3. Führe die Schritte 1 und 2 für den Nord-Süd Lastausgleich durch

Bemerkung 1: Man kann hier nicht entscheiden, welcher Lastausgleich zuerst durchgeführt wird, weil man die sich ergebende Lastsituation sowie die diesbezügliche maximale Last nicht kennt.

Bemerkung 2: Dieser Algorithmus stellt eine Grundlage für beide Algorithmen dar, Algorithmus 6.7 und Algorithmus 6.8.

Bemerkung 3: Dieser Algorithmus kann mit und ohne Balancing-Simulation erfolgen, also wie Algorithmus 6.1 oder wie Algorithmus 6.2. Was unter einer Balancing-Simulation zu verstehen ist, vgl. die Bemerkung zu Algorithmus 6.2.

6.5.2 PLB ohne Balancing-Vorausberechnung mit mehrfachen Lastausgleichsrunden

Hier wird die Information, wie viel Last jeweils transportiert wird, von der Migrationsfunktion bezogen. Das wurde nicht implementiert, weil ein Eingriff in den Balancer notwendig geworden wäre.
Damit gibt es kein PLB ohne die Simulation des Balancings.

Algorithmus 6.7: PLB mit hinreichend vielen Lastausgleichsrunden und ohne Vorberechnung
Input: mit-ohne

1. Führe einen Precomputations-Schritt durch für den Ost-West Lastausgleich (mit-ohne)
2. **while** die Last der Hauptebene ist nicht perfekt ausgeglichen **do**
3. Mache die Migration entsprechend der vL - Information(mit-ohne), vgl. Algorithmus 6.1 Schritte 3a oder 6a
4. Korrigiere die vL - Informationen entsprechend der Migration, vgl. Algorithmus 6.1 Schritte 3b oder 6b
5. Führe die Schritte 1 bis 4 für den Nord-Süd Lastausgleich durch

6. **end do**

Zur Prüfung der Bedingung der **while**-Schleife: Man prüft die Bedingung für alle Kerne gemeinsam, vgl. die Bemerkung 2 direkt zu Beginn von Abschnitt 6.5, d.h. es wird dafür eine All-Reduce-Kommunikation gebraucht.

6.5.3 Mehrfache Lastausgleichsrunden mit Vorberechnung für die Migration

Algorithmus 6.8: PLB mit hinreichend vielen Lastausgleichsrunden und mit Vorberechnung
Input: mit-ohne

1. Führe einen Precomputations-Schritt durch für den Ost-West Lastausgleich(mit-ohne)

2. Bestimme daraus durch Simulation von Lasttransfer (vgl. den Algorithmus 6.2) für jeden Kern die Anzahl der Migrationsrunden, sowie die in jeder Runde zu verschickende Last über seine Kanten. Speichere diese Zahlen für die zu verschickenden Lasten ggf. auch für 2 Ebenen (mit-ohne).

3. Bestimme durch eine All-Reduce-Kommunikation das Maximum an Migrationsrunden für alle Kerne, das mit *max-anzahl(OW)* bezeichnet wird.

4. Mache die Migration (mit-ohne) für alle *max-anzahl(OW)* vielen Runden. Dabei sind wenn nötig Leerrunden einzufügen.

5. Führe die Schritte 1 bis 4 für den Nord-Süd Lastausgleich durch

Man sieht in allen drei Fällen, dass man hier keine Optionen hat, die Reihenfolge von OW- und NS-Lastausgleich geschickt zu wählen, aus Mangel an Informationen.
Ein wesentlicher Schritt dieses Algorithmus besteht darin, die Migrationsrundenanzahl vorauszuberechnen, vgl. den Algorithmus 6.2, im Unterschied zu Algorithmus 6.7.

6.6 Die vollständigen Algorithmen der PLB-Variante

6.6.1 Die Variante 'Vektor'

Vorbemerkung: Das Sammeln der Lastwerte geschieht hier für eine Ost-West- oder Nord-Süd-Linie.
Es sei dabei schon gewählt, ob das über eine 'all to all' Kommunikation (K_n) oder die baumartige Kommunikation geschieht.

Die Variante 'Vektor' mit einer Lastausgleichsrunde

Der Algorithmus gestaltet sich in diesem Fall wie folgt:

Algorithmus 6.9: Die PLB-Variante im Fall 'Vektor' mit nur einer Lastausgleichsrunde
Input: mit-ohne

1. Sammle die Lasten der Zeile des Kerns für den Ost-West Lastausgleich (mit-ohne)
2. Mache Precomputation für diese Ost-West-Zeile entsprechend der PLB-Variante (mit-ohne)
3. Berechne den Lasttransferwert (LTW-Wert) gemäß vL und eigener Last; (mit-ohne)
4. Mache eine Migration nach Ost und West entsprechend dieses LTW's (mit-ohne)
5. Führe die Schritte 1 bis 4 für den Nord-Süd Lastausgleich durch

Kommentar: Zu den ersten beiden Zeilen: Vergleiche den Abschnitt 6.2.3. Für die Zeilen 3 und 4 vergleiche man den Algorithmus 6.4.

Bemerkung: Man kann hier nicht entscheiden, welcher Lastausgleich zuerst duchgeführt werden soll, weil man die entstehende Lastsituation nicht vorausberechnen kann.

6.6.2 Die Variante 'Matrix'

Vorbemerkung 1: Das Sammeln der Lastwerte geschieht hier erst für eine Ost-West-Linie. Dann geschieht das Sammeln über eine Nord-Süd-Linie, dieses aber nicht mit einem Wert, sondern mit allen Werten der jeweiligen Ost-West-Linie.
Es sei dabei schon gewählt, ob das über zwei all to all Kommunikationen (K_n x K_n) oder zwei baumartige Kommunikationen (Baum x Baum) geschieht.

Vorbemerkung 2: Das Sammeln der Lastwerte erfolgt komplett, noch bevor irgendwelche Lasten verschickt werden.

Vorbemerkung 3: Mit Option ist im Folgenden gemeint, zu entscheiden, welcher Lastausgleich als erster durchgeführt wird.

Die Variante Matrix mit einer Lastausgleichsrunde und ohne Option

Algorithmus 6.10: Die PLB-Variante im Fall 'Matrix' mit nur einer Lastausgleichsrunde und ohne Option
Input: mit-ohne

1. Sammle die Lasten aller Kerne für alle Kerne in 2 Schritten (erst in Ost-West-Richtung, dann in Nord-Süd-Richtung) (mit-ohne)
2. Führe den Precomputations-Schritt aus in Ost-West-Richtung für alle Zeilen (mit-ohne)
3. Berechne in Ost-West-Richtung die zu verschickenden Lasttransfermengen entsprechend dem Balancing für eine Runde für die PLB-Variante (mit-ohne)
4. Führe die Ost-West Migration aus(mit-ohne)
5. Für alle Zeilen wird die neue Lastmatrix berechnet für eine Runde(mit-ohne); damit vermeidet man die Kommunikation, um diese Matrix zu bestimmen;

6. Führe die Schritte 2 bis 4 für den Nord-Süd Lastausgleich durch

Kommentar: Für Schritte 1 bis 4 des Algorithmus 6.10 folgt man dem Algorithmus 6.9. Neu ist hier Schritt 5, wo die Nord-Süd-Informationen nicht gesammelt, d.h. durch Kommunikation bestimmt, sondern aus der ursprünglichen Matrix durch Berechnung erlangt werden.

Bemerkung: Je nach 'Grad von Lastausgleich', und der ist in diesem Fall vorausberechenbar, kann die jeweilige Migration unterdrückt werden. Unter 'Grad von Lastausgleich' sei hier der Quotient des Lastmaximums zu dem Lastdurchschnitt zu verstehen.

Die Variante Matrix mit mehreren Lastausgleichsrunden und ohne Option

Algorithmus 6.11: Die PLB-Variante im Fall 'Matrix' mit hinreichend vielen Lastausgleichsrunden und ohne Option
Input: mit-ohne

1. Sammle die Lasten aller Kerne für alle Kerne in 2 Schritten (erst in Ost-West-Richtung, dann in Nord-Süd-Richtung) (mit-ohne)

2. Wende die Precomputation-Methode in Ost-West-Richtung für alle Zeilen an (mit-ohne)

3. Berechne in Ost-West-Richtung die zu verschickenden Lasttransfermengen für alle Zeilen und Runden, basierend auf dem Balancing für die PLB-Variante, wo die zu verschickenden Lasttransfermengen gespeichert werden. Mithilfe der Berechnung der vL-Werte für alle Kerne wird auch die Lastmatrix geändert, vgl. Algorithmus 6.10, sodass man auf allen Kernen die Situation der Lastmatrix nach dem Ost-West-Lastausgleich kennt. Das Speichern der Lasttransferwerte erfolgt nur für die Ost-West Linie in Bezug auf den jeweiligen Kern, aber in Abhängigkeit der Variablen 'mit-ohne'.
   Der Wert *anzahl(OW)* bezieht sich im Falle des reversen Lastausgleiches - weshalb auf 2 Ebenen gerechnet wird - aber nur auf die untere Gitterebene. (mit-ohne)

4. Bestimme daraus das Maximum an Rundenanzahlen *max_anzahl(OW)*, ohne weitere Kommunikation. Jeder Kern hat ja dafür die *anzahl(OW)* Werte der anderen Kerne verfügbar.

5. Mache eine Migration in Ost-West-Richtung für *max_anzahl(OW)* viele Runden (mit-ohne), ggf. mit Leerrunden. Bei Leerrunden macht der Kern dann nur die Zwischenkorrektur mit.

6. Führe die Schritte 2 bis 5 für den Nord-Süd Lastausgleich durch.

Kommentar: Die Schritte 1 und 2 sind wie in Algorithmus 6.10. Zentral ist Schritt 3, wo die Lasttransfermengen berechnet werden für alle Lastausgleichsrunden, und gleichzeitig die Lastmatrizen als Ganzes einen Gesamtschritt (für alle Runden) weiter gebracht werden. In Schritt 4 kann das Maximum der Lastausgleichsrundenanzahl direkt abgelesen werden, weil die Anzahl aller Lastausgleichsrunden für alle OW-Zeilen bekannt sind. Dafür müssen die Werte von *anzahl(OW)* bei Schritt 3 bestimmt und festgehalten werden. Vor dem Precomputations-Schritt in Zeile 6 müssen die Lastwerte nicht

gesammelt werden, weil die Lastmatrix schon aktuell ist. Der Rest folgt wie in den Schritten 2-5.

Bemerkung: Je nach Grad von Lastausgleich kann die jeweilige Migration unterdrückt werden.

Die Variante Matrix mit nur einer Lastausgleichsrunde und der Option, die Reihenfolge der Lastausgleiche zu ändern

Der folgende Algorithmus 6.12 beruht auf dieser Idee: Nach dem Sammeln wird die Lastmatrix gespeichert und fortentwickelt und die Lasttransferwerte dazu gespeichert, erst für den Ost-West-, und dann für den Nord-Süd-Lastausgleich. Dann wird die Lastmatrix wieder geladen und fortentwickelt und die Lasttransferwerte dazu gespeichert. Das geschieht erst für den Nord-Süd-, und dann für den West-Ost-Lastausgleich. Am Ende wird das bessere Ergebnis dann durch Migrationen umgesetzt.

Algorithmus 6.12: Die PLB-Variante mit Matrix und mit nur einer Lastausgleichsrunde und mit Option
Input: mit-ohne

1. Sammle die Lasten aller Kerne für alle Kerne in 2 Schritten (erst in Ost-West Richtung, dann in Nord-Süd-Richtung) (mit-ohne)
2. Speichere die entsprechende Lastmatrix (mit-ohne)
3. Mache die Schritte 2 - 6 von Algorithmus 6.10 - aber ohne die Migrationsschritte
4. Bewerte das Ergebnis, vgl. die Bemerkung
5. Reproduziere die Lastmatrix (mit-ohne)
6. Mache die Schritte 2 - 6 von Algorithmus 6.10, aber ohne die Migrationsschritte, und dabei zuerst für den Nord-Süd-, gefolgt vom Ost-West-Lastausgleich.
7. Bewerte das Ergebnis, vgl. die Bemerkung
8. Führe nun die Migrationsschritte von Algorithmus 6.10 aus, entweder die, wo erst der Ost-West-Lastausgleich erfolgt, oder die, wo erst der Nord-Süd-Lastausgleich erfolgt, je nachdem, was besser ist.

Bemerkung: Die Bewertung erfolgt hier durch den größten Lastwert eines Kerns, um mögliche Effizienz-Verluste zu minimieren, vgl. die Funktion *teilweise_entscheidung_matrix_mit* der Klasse *LastAusgleichsSteuerung* im Programm zur Dissertation.

Die Variante 'Matrix' mit mehreren Lastausgleichsrunden und der Option Reihenfolge der Lastausgleiche ändern

Die Idee ist im Prinzip dieselbe wie von Algorithmus 6.12, nur mit mehreren Lastausgleichsrunden, statt einer Lastausgleichsrunde, weshalb man sich statt auf Algorithmus 6.10 auf Algorithmus 6.11 bezieht.

Algorithmus 6.13: PLB-Variante, Matrix, hinreichend viele Lastausgleichsrunden, mit Option
Input: mit-ohne

1. Sammle die Lasten aller Kerne für alle Kerne in 2 Schritten (erst in Ost-West-Richtung, dann in Nord-Süd-Richtung) (mit-ohne)

2. Bewerte die Lastmatrix danach, ob die Maximallast nach dem West-Ost-Lastausgleich niedriger ist als nach dem Nord-Süd-Lastausgleich, was mittels einer Durchschnittsbildung der Zeilen bzw. Spalten zu prüfen ist.

3. **if** WO-Lastausgleich ist besser **then**
   Mache die Schritte 2 - 6 von Algorithmus 6.11

4. **else**
   Mache die Schritte 2 - 6 von Algorithmus 6.11
   und dabei zuerst für den Nord-Süd-, gefolgt vom Ost-West-Lastausgleich;

Bemerkung 1: Zur Bewertung vgl. die Klasse *GrosseLastAusgleichsSteuerung*, vgl. das Programm zur Dissertation.

Bemerkung 2: Als abschließende Diskussion: Die folgende Diskussion bezieht sich nur auf diesen Algorithmus 6.13, und stellt damit eine Ergänzung zu Abschnitt 6.2.6 sowie einen vollständigen Vergleich der Algorithmen 6.8 und 6.13 dar.
Die Kommunikation zur Berechnung der Partialsummen in der Nähe des Knotens ist bei der ursprünglichen PLB-Methode weniger umfangreich (bei ≥ 16 Kernen insgesamt). Bei der all-to-all Kommunikation für das Sammeln der Lasten bei der Precomputations-Methode hat man mehr Kommunikationsverbindungen, und bei der baumartigen Kommunikation sind die Buffer länger. Und es muss auch mehr gerechnet werden. Dem stehen bei diesem Algorithmus aber folgende Vorteile gegenüber:

1. Man spart bei der Kommunikation:

   (a) Bei der Precomputations-Methode:

      i. Bei der Partialsummenberechnung wird bei PLB nacheinander kommuniziert. Dadurch entstehen Wartezeiten.

      ii. Man braucht keine Kommunikation, um den Wert der Gesamtlast von der Wurzel an alle Kerne weiterzugeben. (Broadcast)

   (b) Bei der Balancing-Methode:

      i. Die Kommunikation bei den Schritten 3a und 6a in Algorithmus 6.2 fallen weg.

      ii. Es ist keine Kommunikation nötig, um die Anzahl an Lastausgleichsrunden zu bestimmen.

2. Bei PLB geschieht der Lastausgleich immer, und zwar ausschließlich erst der Ost-West, und dann der Nord-Süd Lastausgleich. Bei diesem Algorithmus ist man wegen zusätzlicher Kenntnisse in diesem Punkt flexibler.

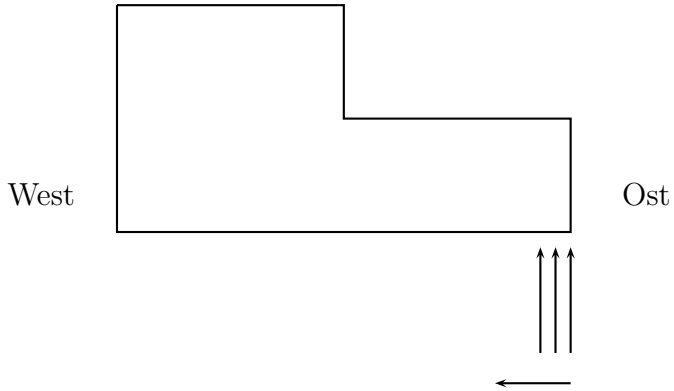


Abbildung 6.10: Auswahl von Punkten bei östlichem Kernnachbarn: Immer zuerst die östlichsten Punkte wählen

6.7 Die Strategie bei der Punktauswahl

In diesem Abschnitt wird die Auswahl der Gitterpunkte besprochen. Wie die entsprechenden Klassen dieses Abschnittes 6.7 (*1-dimensionaleFeldliste*, *Strategie*, *Extreme-Vektoren*) aufgerufen werden, ist Teil der Migration (Abschnitt 6.8) bzw. der parallelen adaptiven Verfeinerung (Kapitel 5), wobei in diesen beiden Abschnitten von den obigen drei Klassen vor allem ein Zugriff auf die Klasse *ExtremeVektoren* erfolgt. Die Gemeinsamkeit von der *Migration* und der *ParallelenAdaptivenVerfeinerung* ist die Änderung der Gitterpunktmengen der Klasse Strategie, bei der *ParallelenAdaptivenVerfeinerung* aufgrund der Verfeinerungen, und bei der *Migration* aufgrund der Umverteilungen. Die Klasse Strategie dient der Auswahl der Gitterpunkte.

Zuerst wird die Idee der Punkteauswahl angegeben. Danach wird die *Lastwertekorrektur* kurz erläutert, weil zum Teil darauf zugegriffen werden soll. Daraufhin wird die *1-dimensionaleFeldliste* (nebst der Klasse *1-dimensionaleFeldlisteMf*) eingeführt, die man für die Klasse *Strategie* essenziell benötigt. Im Anschluss wird die Klasse *Strategie* beschrieben, die mittels Vererbung zu der bei der Klasse *Migration* benutzten Klasse *ExtremeVektoren* erweitert wird. Am Ende wird festgelegt, mit welchen Punkten die Klasse *Strategie* jeweils beladen wird, vgl. Abschnitt 6.7.6.

6.7.1 Die Idee zur Punktauswahl

In der Literatur gibt es folgenden Ansatz: Man wählt Gitterpunkte so aus, damit sie möglichst nahe am Mittelpunkt des die Last empfangenden Kerns liegen bzgl. der Norm $\sqrt{x^2 + y^2}$. Das ist aber eine ungünstige Wahl, denn es werden die Punkte auf Kreislinien um den Mittelpunkt ausgewählt. Das zieht ein entsprechend problematisches Lastausgleichsverhalten nach sich.

Vom Autor stammt die folgende Idee: Wenn Punkte zu einem östlichen Kern geschickt werden sollen, dann werden die östlichsten Punkte ausgewählt. Das hat zwei günstige Effekte:

1. Es erfolgt eine plausible Auswahl.

2. Die Koordinaten bleiben sortiert in dem Sinne, dass im Allgemeinen ein östlich

liegender Kern bzgl. der Kerntopologie der Abbildung 6.2 auch östlich liegende Punkte hat. Dasselbe soll auch für die anderen Himmelsrichtungen gelten.
In einem konkreten Fall gibt es später Abweichungen von dieser Regel, insbesondere beim reversen Lastausgleich, vgl. Abschnitt 6.7.6.

6.7.2 *Die LastWerteKorrektur* - Teil 1

Alle Klassen und Funktionen zur *LWK* (*LastWerteKorrektur*) haben folgenden Zweck:

Es geht hier um folgende Option: Normalerweise hat jeder Gitterpunkt den Lastwert 1. Hier wird die Möglichkeit eröffnet, Gitterpunkten am echten Rand den Lastwert 0 zuzuteilen, aus folgendem Grund: Wenn nur auf inneren Punkten gerechnet wird, macht es Sinn, bei der Lastbewertung beim Lastausgleich nur inneren Punkten eine Last zuzuordnen.

Hier hat man beide Optionen, sodass der Anwender sich entscheiden kann, ob die LWK gemacht wird, oder nicht.

Die Arbeit an dieser Aufgabe teilt sich auf in:

1. Bei der Bestimmung der zu transferierenden Vektoren in Abschnitt 6.7.3 kann mithilfe eines **Merkfeldes**, welches verzeichnet, ob ein Gitterpunkt ein echter Randpunkt ist, erreicht werden, dass entweder echte Randpunkte gar nicht verschickt werden, oder die echten Randpunkte werden mitverschickt, aber nicht in die Lastzählung einbezogen. Man kann auch beide Methoden kombinieren, indem sie abwechselnd durchgeführt werden.

2. In Abschnitt 6.9 wird erläutert, wie man die Lastmenge eines Gitters so bewertet, dass nur Punkte, die keine echten Randpunkte sind, zur Last gezählt werden. Außerdem wird dort die Bestimmung des **Merkfeldes für echte Randpunkte** präsentiert.

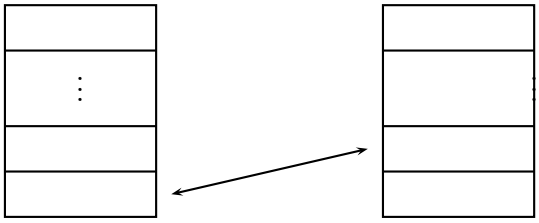
6.7.3 Die Klasse *1-dimensionaleFeldliste*

Die Klasse *Strategie* verwendet die Klasse *1-dimensionaleFeldliste* als Unterstruktur. Deshalb wird zuerst diese Struktur beschrieben:

Die Daten dieser Klasse *1-dimensionaleFeldliste*

Der Zweck dieser Klasse besteht darin, eine Liste von einfachen Koordinaten zu haben, die man in O(1) einfügen und entfernen und in O(Anzahl Einträge) durchlaufen bzw. sie komplett löschen kann.
Sie ist analog zur *2-dimensionalenFeldliste* aufgebaut, vgl. Abschnitt 4.2.1, nur mit einem eindimensionlen Feld statt dem 2-dimensionalen Feld. Deswegen hat sie auch die Eigenschaften der *2-dimensionalenFeldliste*, und erfüllt also die oben genannten Leistungsmerkmale.

Sie besteht aus 2 Datenstrukturen, die aufeinander verweisen:



Doppel-verkettete-Liste Koordinaten-Index-Liste

Abbildung 6.11: Die *1-dimensionaleFeldliste*

1. Die doppelt verkettete Liste mit Vorgänger, Nachfolger sowie der Koordinate als Inhalt.

2. Die Koordinaten-Index-Liste: Ein 1-dimensionales Koordinaten-Feld, welches für eine Koordinate auf das zugehörige Feld der doppelt verketteten Liste für diese Koordinate zeigt.

Bemerkung: Man kann statt einer doppelt verketteten Liste auch eine einfache Liste nehmen, braucht aber auf jeden Fall die Koordinaten-Index-Liste. Das würde genau wie die *Punktefeldliste* der *ParallelenAdaptivenVerfeinerung* funktionieren, ist aber nicht implementiert worden.

Die Funktion Vektorbildung

Neben den Funktionen fürs Einfügen und Entfernen von einfachen Koordinaten, sowie dem Durchlaufen und dem kompletten Löschen, gibt es die folgende weitere Zusatz-Funktion, die jetzt beschrieben wird:
Man kann bei einer eindimensionalen Feldliste gleich einen ganzen Vektor auswählen. Dafür wird ein Lastwert übertragen, der die Maximallänge dieses Vektors angibt.

Algorithmus 6.14: *Vektorbildung_der_eindimensionalen_Feldliste*
Input: Vektormaximallänge vml
Output: Ein 1-dimensionaler Vektor. Er ist entweder nicht verlängerbar oder er hat die Maximallänge vml

1. koord = erster Eintrag der Feldliste als doppelt verkettete Liste
   (Wähle 'zufällig' ein Element der 1-dim. Feldliste aus.)

2. **if**(vml=1) **then return**(koord, koord)
   (Soll der Vektor nur 1 Punkt lang sein, dann gebe (koord, koord) zurück.)

3. **for** i von koord-1 **to** max(0, koord - vml + 1) **step** -1
   (Scanne die *1-dimensionaleFeldliste* vom ausgewählten Element -1 bis an ihren Anfang bzw. bis zur maximalen Vektorlänge. Nach Durchlauf ist der Wert i = -1 oder i = koord - vml, und der Vektor beginnt bei (i+1).)

4. **if** i kein Eintrag in der Feldliste ist **then** Beende die Schleife
   (Falls i kein Eintrag mehr ist, beende den Scan Richtung Anfang.)

5. **end for**

6. **for** j von koord+1 **to** max(Feldlistenlänge - 1, (i+1) + vml - 1) **step** 1
   (Scanne die 1 - dim. Feldliste vom ausgewählten Element (koord) +1 bis an ihr Ende, oder bis zur maximalen Vektorlänge. Bei einem kompletten Schleifendurchlauf hat die Variable j den Wert Feldlistenlänge oder (i+1) + vml, d.h. j-1 ist in dem Fall die untere Koordinate des zurückgegebenen Vektors.)

7. **if** j kein Eintrag in der Feldliste ist ⇒ return (i+1,j-1)
   (Falls j kein Eintrag mehr ist, beende den Scan Richtung Ende.)

8. **end for**

9. **return**(i+1,j-1)
   (Wurde die 2. Schleife komplett durchlaufen, gebe den resultierenden Vektor zurück.)

Die Erweiterung zur *1-dimensionalenFeldlisteMf*

Wir erweitern die *1-dimensionaleFeldliste* um die Möglichkeit einer Lastwertekorrektur (LWK).
Bevor der dafür zuständige Algorithmus beschrieben wird, muss das Merkfeld erläutert werden: Es ist 2-dimensional, die *1-dimensionaleFeldliste* ist aber eindimensional. Dennoch haben wir in der *1-dimensionalenFeldliste* eine 1-dimensionale Funktion *ist-randpunkt*, mit der abgefragt werden kann, ob ein Punkt ein echter Randpunkt ist oder nicht.

Zunächst eine kurze Beschreibung der *OW-* und *NS-1-dimensionale*Feldlisten, damit die Funktion *ist-randpunkt*(koordinate) erklärt werden kann mit Vorgriff auf Abschnitt 6.7.4. Zu den *1-dimensionaleFeldlistenMF*: Sie wird eine k. *OW-1-dimensionaleFeldliste-MF* werden, wenn sie die k.te Spalte des Gitters darstellt, oder eine k. *NS-1-dimensionale-FeldlisteMF*, wenn sie die k.te Zeile zu dem Gitter darstellt. Die Idee bei dieser Zuordnung: Die *OW-1-dimensionaleFeldliste* dient dem Ost-West-Lastausgleich, arbeitet also mit vertikalen Vektoren. Analoges gilt für die *NS-1-dimensionaleFeldliste* zum Nord-Süd-Lastausgleich, und damit zur Übertragung von horizontalen Vektoren. Damit besteht folgende Lösung für obiges Problem:

Falls die *1-dimensionaleFeldlisteMF* die k.te *NS-1-dimensionaleFeldlisteMF* ist, dann gilt: ist-randpunkt(i) = merkfeld[k][i].

Falls die *1-dimensionaleFeldlisteMF* die k.te *OW-1-dimensionaleFeldlisteMF* ist, dann gilt: ist-randpunkt(i) = merkfeld[i][k].

Die Funktion Vektorbildung-LWK

Mit der Funktion *ist-randpunkt* lautet nun diese Funktion wie in Algorithmus 6.15 angegeben. Die Funktion *ist-randpunkt* liegt aber doppelt vor (für *OW-* und *NS-1-dimensionaleFeldlisten Mf*), sodass der folgende Algorithmus auch doppelt implementiert werden muss (für *OW-* und *NS-1-dimensionaleFeldlisten Mf*):

Algorithmus 6.15 : *Vektorbildung-LWK_der_1-dimensionalenFeldlisteMf*
Input: Vektormaximallänge vml
Output: Zurückgegeben wird ein 3-Tupel: Von dem bestimmten Vektor wird die linke und rechte oder obere und untere Koordinate zurückgegeben. Mit der dritten Koordinate des Vektors ist die *Feldliste MF* indiziert. Die aufrufende Instanz kennt diesen Index, der deswegen nicht benötigt wird. Die Last des bestimmten Vektors wird als dritte Komponente des 3-Tupels zurückgegeben.
Es gilt, dass der bestimmte Vektor entweder nicht verlängerbar ist oder die Maximallast vml hat.
Damit haben wir 3 Begriffe in dem Algorithmus: 'vml' ist die Maximallast, 'last-gefunden' der Lastwert des vorübergehend bestimmten Vektors, und 'ist-randpunkt' gibt an, ob ein echter Randpunkt vorliegt.

1. koord = erster Eintrag der *1-dimensionalenFeldliste* als doppelt verkettete Liste (Finde eine beliebige existierende Koordinate.)

2. **if**(vml=1 **AND** *ist-randpunkt*(koord)=*false*) **then return**(koord, koord, 1) (Soll der Vektor die Last 1 haben, und ist der gefundene Punkt kein echter Randpunkt, terminiert der Algorithmus erfolgreich.)

3. **if** *ist-randpunkt*(koord)=true **then** last-gefunden = 0
   else last-gefunden = 1
   (Setze die Variable 'last-gefunden' entsprechend der Last der Koordinate.)
   end if

4. **for** i von koord-1 **to** 0 **step -1**
   Dabei sei i=-1 bei komplettem Durchlauf
   (Scanne die *1-dimensionaleFeldliste* vom ausgewählten Element (-1) bis an ihren Anfang.)

5. **if** i kein Eintrag in der Feldliste ist **then** Beende Schleife
   (Falls i kein Eintrag mehr ist, beende den Scan Richtung Anfang.)

6. **if** ist-randpunkte(i) = false **then** last-gefunden ++
   (Wurde ein Punkt, der kein echter Randpunkt ist, gefunden, so inkrementiere den Lastwert.)

7. **if** gefunden = vml **then** return(i, koord, last-gefunden)
   (Ist ein Vektor der vorgegebenen Lastmenge gefunden, beende den Algorithmus.)

8. **end for**

9. **for** j von koord+1 **to** Feldlistenlänge-1 **step 1**
   (Scanne die 1 - dim. Feldliste vom ausgewählten Element (+1) bis an ihr Ende.)

10. **if** j kein Eintrag in der Feldliste ist **then return** (i+1,j-1,last-gefunden)
    (Falls j kein Eintrag mehr ist, beende den Scan Richtung Ende.)

11. **if** *ist-randpunkt*(j) = false **then** last-gefunden ++
    (Wurde ein Punkt, der kein echter Randpunkt ist, gefunden, so inkrementiere den Lastwert.)

12. **if** gefunden = vml **then return**(i+1,j,last-gefunden)
    (Hat der Vektor die vorgegebene Last, beende den Algorithmus.)

13. **end for**

14. **return**(i+1,Feldlistenlänge-1,last-gefunden)
    (Ist die Schleife ohne vorzeitiges Beenden durchgelaufen, nehme den Vektor bis zum Feldlistenende.)

Bemerkung 1: Die Schleifenzielwerte sind hier nicht wie in Algorithmus 6.14 an die Vektorlistenlänge anpassbar, weil echte Randpunkte nicht gezählt werden und daher eine vorausberechnete Länge übertroffen werden könnte, vgl. die Schritte 4 und 8.

Bemerkung 2: Es gibt einen zweiten Algorithmus zum Finden von Vektoren, deren Randpunkte nicht als Last zählen. Bei diesem Algorithmus werden echte Randpunkte gar nicht verschickt. Er entsteht, wenn man in Algorithmus 6.14 die Abfrage 'ist kein Eintrag' ersetzt durch 'ist kein Eintrag oder ist echter Randpunkt', mithilfe der Funktion *ist-randpunkt*.

Dieser Algorithmus wird hier nicht weiter ausgeführt, weil dabei Gebiete auftreten können mit langen Randlinien echter Randpunkte, da die inneren Punkte übertragen werden und nur die echten Randpunkte verbleiben. Deshalb wird Algorithmus 6.15 zur Lösung der LWK-Aufgabe bevorzugt. Implementiert wurden aber beide Algorithmen.

6.7.4 Die Klasse *Strategie*

In dieser Klasse wird die Gitterstruktur und alle Informationen zum Abruf der östlichsten, westlichsten, nördlichsten und südlichsten Gitterpunkte gespeichert.
Die Darstellung der Klasse erfolgt in 3 Schritten:

1. Angabe der Strukturen, die darin enthalten sind
2. Erläutern des Speicherns von Gitterpunkten anhand einer Abbildung (Abbildung 6.12).
3. Darlegen wie die Strukturen benutzt werden zum Suchen und Entfernen sowie zum Einfügen.

Angabe der Datenstrukturen *OW-1-dimensionaleFeldliste*, *NS-1-dimensionaleFeldlisten*, *maxOW*, *minOW*, *maxNS* und *minNS*

Gegeben sei ein Gitter mit n Zeilen und m Spalten. Dann werden 2 Felder von *1-dimensionalenFeldlisten* angelegt: die m *OW-1-dimensionalen-FeldlistenMf* der Länge n und die n *NS-1-dimensionalen-FeldlistenMf* der Länge m.

Die i.te *OW-1-dimensionale-FeldlisteMf* gibt dabei an, welche Punkte die Ost-West-Koordinate i haben, und die j.te *NS-1-dimensionale-FeldlisteMf* gibt an, welche Punkte die Nord-Süd-Koordinate j haben. Oder anders ausgedrückt: Die i. *OW-1-dimensionale-FeldlisteMf* speichert die i. Spalte des Gitters, die j. *NS-1-dimensionale-FeldlisteMf* die

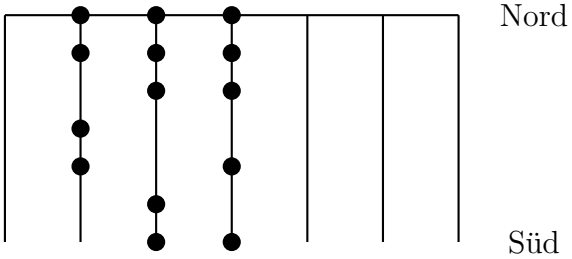


Abbildung 6.12: Die *OW-1-dimensionaleFeldlistenMf* der Klasse *Strategie*

j. Zeile des Gitters.

Dann braucht man noch 4 Variablen *maxOW*, *maxNS*, *minOW* und *minNS*, die folgendes angeben:

1. *maxOW* : Die östlichste vorhandene Koordinate
2. *minOW* : Die westlichste vorhandene Koordinate
3. *maxNS* : Die südlichste vorhandene Koordinate
4. *minNS* : Die nördlichste vorhandene Koordinate

Diese Variablen müssen nach jeder Änderung aktualisiert werden.

Dafür muss man im OW-Fall ein eindimensionales Feld genannt *listenlängeOW* der Länge m anlegen, welches an der Stelle k verzeichnet, wie viele Punkte die k. Spalte hat. Analog muss man ein eindimensionales Feld genannt *listenlängeNS* der Länge n im NS-Fall anlegen, welches an der Stelle l verzeichnet, wie viele Punkte die l. Zeile hat.

Bemerkung: Ferner müssen bei einer noch leeren Liste die Werte maxOW = maxNS = -1 sowie minOW = m + 1 und minNS = n + 1 gesetzt werden, um bei dem ersten Eintrag dessen Koordinaten automatisch als Minimalwerte bzw. Maximalwerte zu übernehmen.

Erläutern des Speicherns von Gitterpunkten

Durch die Benutzung der m *OW-1-dimensionalen-FeldlistenMf* der Länge n kann man das gesamte lokale Gitter abbilden, vgl. Abbildung 6.12. Dasselbe gilt für die n *NS-1-dimensionalen-FeldlistenMf*.

Benutzung der Strukturen

Das Finden der östlichsten, westlichsten, nördlichsten und südlichsten Gitterpunkte ist relativ einfach. Zum Beispiel finden sich die östlichsten Gitterpunkte in der *maxOW*. *OW-1-dimensionaleFeldlisteMf*. Analoges gilt für die anderen Listen.

Das Finden östlichster, westlichster, nördlichster und südlichster Gitterpunkte steht in Verbindung mit den Funktionen *Vektorbildung* bzw. *Vektorbildung-LWK*, vgl. die Algorithmen 6.14 und 6.15. Die Funktionen werden aufgerufen in der Feldliste mit einem

extremen Wert eines Indexes (z.B. *maxOW* bei *OW-1-dimensionalenFeldlisten*).
Der Befehl lautet dann: *feldlisteOW(maxOW).vektorbildung(lastmenge).*
Für das Finden der Vektoren werden Namen verwendet, wie z.B. *finde_östlichsten_Vektor(Lastmenge).* Zusammen mit dem Suchen des Vektors geschieht das Entfernen desselben.

Zum Einfügen und Entfernen: Es bleibt zu zeigen, wie die anderen Strukturen beim Einfügen und Entfernen zu handhaben sind.

1. Entfernen und Einfügen eines Punktes in die *OW-* und *NS-1-dimensionaleFeldliste-Mf*: Soll (f,g) eingefügt werden, wird in die g. *OW-1-dimensionaleFeldlisteMf* an der Koordinate f eingefügt, sowie die f. *NS-1-dimensionaleFeldlisteMf* an der Koordinate g. Entsprechendes gilt fürs Entfernen.

2. Die Änderung der Listen *listenlängeOW* und *listenlängeNS* beim Einfügen bzw. Entfernen eines Punktes: Soll (f,g) eingefügt werden, wird *listenlängeOW*[g] inkrementiert, und *listenlängeNS*[f] inkrementiert.

   Soll (f,g) entfernt werden, wird *listenlängeOW*[g] dekrementiert, und *listenlängeNS*[f] dekrementiert.

3. Die Änderung von *maxOW*, *minOW*, *maxNS*, *minNS* beim Einfügen eines Punktes: Es wird $maxOW = \max(maxOW,g)$, $minOW = \min(minOW,g)$, $maxNS = \max(maxNS,f)$ und $minNS = \min(minNS,f)$ gesetzt.

   Die Änderung von *maxOW*, *minOW*, *maxNS*, *minNS* beim Entfernen eines Punktes: Nach dem Entfernen eines Punktes aus der Spalte g wird *maxOW* wie folgt geändert:

   Algorithmus 6.16: *Update_maxOW*
   Input: Die Koordinate g, wobei der Punkt (f,g) entfernt wurde

   (a) **if**(Gitter komplett leer) **then** *maxOW* = -1, beende den Algorithmus
   (b) *listenlängeOW*(g)– (dekrementieren, da ein Punkt weniger ist in der g. Spalte)
   (c) **if** *listenlängeOW*(g)>0 **then** Beende den Algorithmus
   (d) **for** i = g zähle herunter bis 0
   (e) **if** *listenlängeOW*(i)>0 **then** Beende die Schleife
   (f) **end for**
   (g) *maxOW* = i

6.7.5 Die Klasse *ExtremeVektoren*

Bei der Klasse *ExtremeVektoren* wird zu einer vorgegebenen Lastmenge eine Liste von Vektoren mit dieser Last bestimmt. Es wird hier nur ein Algorithmus angegeben für die Himmelsrichtung Osten, vgl. Algorithmus 6.17. Für die anderen Himmelsrichtungen

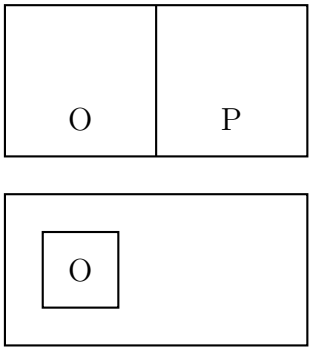


Abbildung 6.13: Beispiel: Zwei Gitterebenen Lastverteilung vor dem Lastausgleich

folgt das analog.

Man beachte: Die gefundenen und gespeicherten Vektoren müssen immer gleich gelöscht werden. Nur durchs Löschen bleiben die Indizes minOW, maxOW, usw. aktuell, damit auch der nächste Vektor gefunden werden kann.

Algorithmus 6.17: Extreme Vektoren Osten(ggf. mit LWK)
Input: lastmenge
Output: Liste von Vektoren mit der Gesamtlast

1. **while** lastmenge $\neq 0$ **do**
2. Bestimme östlichsten Vektor(lastmenge) (f_1, f_2, g) mithilfe der Klasse *Strategie* - ggf. mit *LWK*, vgl. die Algorithmen 6.14 und 6.15.
3. Entferne den Vektor aus der Klasse *Strategie*
4. Reduziere den Wert der Variable lastmenge um (f_2-f_1+1) im Fall ohne LWK bzw. um den Wert der Variablen last-gefunden beim Algorithmus im Fall mit der LWK
5. Speichere den Vektor
6. **end do**

6.7.6 Festlegung, mit welchen Gitterpunkten die Klasse Extreme Vektoren belegt wird.

Das folgende Beispiel erläutert die Problematik, wenn das gesamte lokale Gitter eingetragen wird in die Klasse *ExtremeVektoren* bzw. die Klasse *Strategie*:
Gegegeben seien die Ebenen 1 und 2, wobei 2 der Index für die gröbere Ebene sei. Und es seien 2 Kerne O und P gegeben. Ferner gelte für Ebene 2, dass sie komplett gefüllt ist, auf Ebene 1 sei aber nur das gesondert umrandete Gebiet vorhanden. Diese Situation ist dargestellt in Abbildung 6.13.

Nun erfolgt ein Lastausgleich, wobei auf Ebene 1 die östliche Hälfte des Verfeinerungsgebietes von O an P geschickt wird. Da übereinander liegende Punkte auf demselben

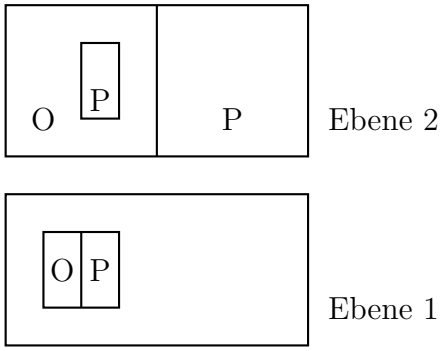


Abbildung 6.14: Beispiel: Zwei Ebenen Lastverteilung nach dem Lastausgleich

Kern liegen, ändert sich auch die Lastverteilung auf Ebene 2, und es entsteht die Situation von Abbildung 6.14.

Nun ist die Last auf Ebene 2 unausgeglichen. Wenn nun ein zusätzlicher Lastausgleich auf Ebene 2 gemacht würde, dann würde der zuletzt auf Ebene 1 gemachte Lastausgleich wieder umgekehrt, und man erhält die Ausgangssituation.

Man hat nun zwei Möglichkeiten. Entweder verzichtet man auf den Lastausgleich auf Ebene 2, und nimmt den unvollständigen Lastausgleich der gröberen und damit kleineren Gitter hin, oder man löst dieses Problem wie folgt:

Auf Ebene 2 wird eine Instanz *ev2* der Klasse *ExtremeVektoren* eingeführt, mit den folgenden Gitterpunkten: Man nimmt alle Gitterpunkte von Ebene 2, ohne darunterliegende Gitterpunkte von Ebene 1. Macht man jetzt neben dem Lastausgleich auf Ebene 1 einen Lastausgleich auf Ebene 2 bzgl. *ev2*, so wird der Lastausgleich auf Ebene 1 durch den Lasttransfer auf Ebene 2 nicht tangiert, und die Last ist auf beiden Ebenen (fast) ausgeglichen. Auf Ebene 2 ist die Last nach beiden Lastausgleichen aus folgendem Grund auf beiden Ebenen ausgeglichen: Das gilt für die Punktmenge **mit** darunter liegenden Gitterpunkten wegen des Lastausgleichs bzgl. der Punkte auf Ebene 1. Weiterhin gilt dies für die Punktmenge **ohne** darunter liegende Gitterpunkte wegen des Lastausgleichs bzgl. der Punkte auf Ebene 2.

Bemerkung 1: Alle Ebenen darüber sind auch (fast) ausgeglichen.

Bemerkung 2: Nun wird erläutert, dass die Last **fast** ausgeglichen wird. Das bezieht sich darauf, dass bei einer Änderung der Last der Menge m auf der Ebene h sich nicht automatisch die Last auf Ebene h+1 um m/4 ändert. Man weiß nämlich nicht, wie viele gemeinsame Punkte es bei den Punkten gibt, die die Last der Menge m bilden. Da aber fast jeder 4. Punkt bzgl. den Gitterpunkten auf Ebene h ein gemeinsamer Punkt ist, stimmt diese 'Näherung' fast.

Bemerkung 3: Man kann beide Lastausgleiche gemeinsam durchführen, weil die verwendeten Gitterpunkte disjunkt sind.
Man führt das aus, was in Abbildung 6.15 beschrieben ist. Wird dann das Farbenfeld außen gepackt, werden die richtigen Randwerte übermittelt, auch wenn die nach We-

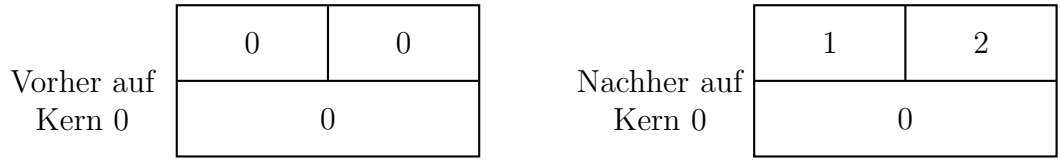


Abbildung 6.15: Berührung von Punkten, die in 2 verschiedene Richtungen geschickt werden. Vor dem Verschicken der Farbenfeldwerte wird das eigene *Farbenfeld* mit der ID des Zielkerns beschrieben.

sten und Osten verschickten Punkte sich berühren. Außerdem stimmen die Farbwerte auf dem Kern, der die Punkte verschickt.

Bemerkung 4: Wird die Option 'mit ev2' genommen, muss man bei allen Punktänderungen auf allen Ebenen die Struktur *ev2* so ändern, dass sie alle Punkte enthält, welche die gröbere Ebene hat und die feinere Ebene nicht. Also muss diese Struktur geführt werden, insbesondere was das Einfügen und Entfernen von Gitterpunkten betrifft, und zwar bei der *Migration*, vgl. Abschnitt 6.8, und bei der *ParallelenAdaptivenVerfeinerung* aus Kapitel 5.

1. Es wird die Änderung bei der *ParallelenAdaptivenVerfeinerung* diskutiert, weil diese Problematik in Kapitel 5 noch nicht behandelt werden konnte:
Fügt man einen Gitterpunkt ein, dann wird er, weil zu dem Zeitpunkt kein feinerer Gitterpunkt darunter existiert, in *ev2* eingefügt. Außerdem wird, wenn der Gitterpunkt darüber in *ev2* liegt, dieser aus *ev2* entfernt.
Entfernt man einen Gitterpunkt, dann wird der Gitterpunkt darüber, falls er existiert, in *ev2* eingefügt. Außerdem wird der Gitterpunkt selber entfernt, falls er in *ev2* enthalten war.

2. Migration: Wird ein Punkt (f,g) verschickt, und er wurde mithilfe der Klasse *ExtremeVektoren* gefunden, so brauchen die Datenstrukturen von *ev2* nicht geändert zu werden, weil er beim Suchen entfernt wurde und darüber und darunter kein Punkt in *ev2* liegen kann. Beim Einfügen von Punkten (f,g) muss geprüft werden, ob Gitterpunkte darunter existieren. Dafür muss die Änderung des Multi-Gitters vor der Änderung der *ev2*-Strukturen erfolgen.

6.8 Migration

Hier wird die Migration in Ost-West-Richtung dargestellt. Die Migration in Nord-Süd-Richtung erfolgt in genau derselben Weise und wird deswegen hier nicht dargestellt. Ferner werden, wo dies möglich ist, die Routinen nur für die Ost-Richtung dargestellt, wenn in West-Richtung dasselbe implementiert wird.

6.8.1 Übersicht über die Migrationsklasse

Zuerst werden in Abschnitt 6.8.2 wichtige Klassen definiert, die die Arbeit an der Migration wesentlich erleichtern. Es folgt in Abschnitt 6.8.3 eine Erläuterung, inwieweit der reverse Lastausgleich aus Abschnitt 6.7.6 realisiert wird.

Danach wird in Abschnitt 6.8.4 die Hauptfunktion mit ihrer Unterfunktion *testen_und_entpacken* vorgestellt. Diese Unterfunktion hat denselben Stellenwert wie die Hauptfunktion selber.
Danach wird der Umgang mit den Strukturen aus Abschnitt 6.8.2 erläutert, sowie die Packen- und Entpacken-Routinen und die Zwischenkorrektur dargestellt.

6.8.2 Unterstützende Grundklassen und die Flags

Die Klasse *mv-vertikal*

Diese Klasse ist enorm wichtig für eine übersichtliche Klasse *Migration*, und sie wird vielfach in dieser Klasse benötigt (z.B. für das Beschreiben der *mvf-Buffer).*
Hier werden die Koordinaten der vertikalen Vektoren für alle Gitterebenen hineingeschrieben sowohl beim Senden, als auch beim Empfangen. Es ist eine Multi-Level-Vektorenliste, auf der aufbauend die Gitterstrukturen beim Packen abgebaut und beim Entpacken aufgebaut werden.
Definiert wird sie für Ost und West, weil nach dort verschickt und von dort empfangen wird. Beim Ost-West-Lastausgleich werden vertikale Vektoren verschickt.

Bemerkung: Bei den *mv-horizontal*-Datenstrukturen beim Nord-Süd-Lastausgleich wird der Speicherplatz der *mv-vertikal*-Datenstrukturen weiterverwendet, um Speicherplatz zu sparen.

Die Buffer *mvf-buffer*

Diese Buffer werden definiert für Osten und Westen und für die Migration oben und unten, weil das die möglichen Buffer für die Migrationen sind (sie werden auch für die Migration 'oben' definiert, aber nur beim reversen Lastausgleich). Dort werden die Informationen des sendenden Kernes gepackt, und aus diesen Buffern werden die Informationen der empfangenden Kerne entpackt. Dazwischen werden sie verschickt.
In die *mvf-Buffer* werden Multi-Level Informationen gepackt, wie die Vektorlisten der Klasse *mv-vertikal* oder der anderen Strukturen aus Abschnitt 6.8.10. In Bezug auf die Klasse *mv-vertikal* gibt es die Klassen, die *mv-vertikal*-Informationen zu *mvf-Buffer*-Informationen machen, und umgekehrt. Die entsprechenden Klassen werden *mv-vertikal-zu-mvf-Buffer* oder *mvf-Buffer-zu-mv-vertikal genannt.*
Es werden alle Informationen in den *mvf-Buffern* verschickt bis auf die Werte auf den verschickten Punkten, die in separaten Buffern verschickt werden.

Die Flags

Sie geben an, welche Schritte jeweils gemacht werden müssen, um entsprechend der Lastsituation die notwendigen Arbeiten durchzuführen.
Man hat zuerst die Flags für *senden* und *empfangen* nach *Osten* und nach *Westen*, d.h. ob das auf der Grundebene (unten) jeweils gemacht wird oder nicht. Dann gibt es diese Flags für die *untere* und die *obere* Ebene beim reversen Lastausgleich. Die Flags halten ferner fest, ob die *mvf-Buffer*-Information noch zu kommunizieren ist oder ob das noch für die Werte durchzuführen ist.
Das sind die Fälle {west,ost} x {senden, empfangen} x {werte, mvf-buffer} x {unten,

oben}, d.h. 16 Kommunikationsflags nur für den Ost-West-Lastausgleich. Es wird damit festgehalten, ob die jeweilige Aktion noch zu tun ist, oder nicht.

6.8.3 Überlegungen zum reversen Lastausgleich

Es geht hierbei um Konsequenzen zu den Vorüberlegungen dazu, auf der Ebene oben eine Kompensation für den Lastausgleich aufgrund der unten verschickten Punkte zu erreichen, vgl. Abschnitt 6.7.6. Studiert man diesen Abschnitt, kommt man zu der Überlegung, dass oben dann gesendet wird, wenn unten empfangen wird, und umgekehrt. Die Logik dahinter: Wird durch die parallele adaptive Verfeinerung auf dem unteren Gitter mehr Gebiet auf einem Kern als bei seinem Nachbarn erzeugt, dann werden oben mehr Punkte aus *ev2* ausgetragen. Das hat zur Folge, dass auf der oberen Ebene der Nachbarkern mehr Last hat. Die Konsequenz daraus: Die Lasten unten und oben werden in entgegengesetzer Richtung verschickt.

Andererseits zeigt die Praxis (aufgrund von Läufen) insbesondere bei mehreren Lastausgleichsrunden, dass es Sinn macht, oben Last zu verschicken, auch wenn unten keine Last verschickt wird. Die Hauptfunktion wurde deswegen entsprechend umgeschrieben, um das zuzulassen, vgl. Abschnitt 6.8.4 Schritt 3.

Im Prinzip wäre der Fall möglich, dass sowohl unten als auch oben an denselben Kern Last geschickt wird. Das ist kein wichtiger Punkt bei der Entwicklung des entgegengesetzten Lastausgleichs gewesen, und ist es immer noch nicht.
Es wurde dafür aber das Folgende implementiert. Man benötigt die Projektionsfunktionen aus Abschnitt 6.8.7. Damit funktioniert das so: Für die auf dem feineren Gitter bestimmte Punktmenge werden rekursiv die Punktmengen auf allen feineren Gittern bestimmt. Man nutzt hier den Umstand aus, dass die auf dem gröberen Gitter definierte Punktmenge keine feineren Gitterpunkte unter sich hat. Dann wird die feinere Punktmenge auf die gröbere Ebene projeziert. Dort werden die gröberen Punkte hinzukopiert. Rekursiv bestimmt man dann von der gröberen Ebene aus die Vektoren aller gröberen Ebenen.

6.8.4 Die Hauptfunktion der Migration

Hier wird die Hauptfunktion des Ost-West-Lastausgleiches mit dem reversen Lastausgleich ('oben') dargestellt. Wir definieren dazu OW = {Ost, West}. Die Angabe, wie viel Last in welche Richtung zu schicken ist, sei mit Lastdifferenz-HR(-oben) bezeichnet, als Differenz zum angestrebten Wert.

Beim nicht reversen Lastausgleich kommt es dementsprechend zu Vereinfachungen.

In dem Hauptalgorithmus werden verschiedene Packen- und Entpacken-Routinen verwendet, einige Flags und auch die Zwischenkorrektur. Das wird in den folgenden Abschnitten erläutert. Die Strukturen *mv-vertikal* und *mvf-Buffer*, die vor allem fürs Packen und Entpacken verwendet werden, sind Teil der Algorithmen der folgenden Abschnitte. Sie werden also in der Hauptfunktion selber nur beim Reset zu Beginn mit

aufgeführt.

Vorbemerkung 1: packen(HR) und empfangen(HR) seien Flags für HR ∈ {Ost,West}.

Vorbemerkung 2: Teile wie *Zwischenkorrektur*, *packen_entfernen_punkte*, *packen_ieS* und *packen_mv* werden in den Abschnitten ab 6.8.6 behandelt.

Algorithmus 6.18: Hauptfunktion der Klasse *Migration*

Input: Für HR ∈ OW Lastdifferenz-HR und Lastdifferenz-HR-oben (Die Lastdifferenz ist positiv wenn verschickt werden soll.), die Gitterebene h (oben ist die Ebene h+1)

1. **for all** HR ∈ OW
   Setzte *mv-vertikal*(HR) zurück und bestimme die Flags zum Senden und Empfangen nach Ost und West.
   end for

2. **for all** HR ∈ OW
   if empfangen(HR) = *true* **then**
   Setze empfangen-HR-mvf = *true* und empfangen-HR-werte = *true* und initiiere die dazugehörigen Kommunikationen
   end for

3. **for all** HR ∈ OW
   if Lastdifferenz-HR ≤ 0 und Lastdifferenz-HR-oben > 0 **then**
   Flags für sende oben HR (Werte und Info) auf *true* setzen und *packen_mv*(HR)-oben (Lastdifferenz-HR-oben, h+1) durchführen. (*packen-mv* wird in Abschnitt 6.8.6 erläutert)
   else
   Setze Flags für sende HR oben auf *false*
   end if
   end for

4. **for all** HR ∈ OW
   if Lastdifferenz-HR ≥ 0 **AND** Lastdifferenz-HR-oben < 0 **then**
   Flags für empfange oben HR (Werte und Info) auf *true* setzen und *entpacken-mv* für HR-oben initiieren.
   else
   Setze Flags für empfange HR oben auf *false*
   end if
   end for

5. **for all** HR ∈ OW
   if packen(HR) = *true* **then**
   packen_mv(HR) (Lastdifferenz - HR, h)
   end for

6. *zwischenkorrektur_packen_und_verschicken_für_west_und_ost*

7. **for all** HR ∈ OW
   if(flag_senden(HR)=*true*) **then** *packen_entfernen_punkte*(HR)
   end for

8. **for all** HR ∈ OW
   if(flag_senden_oben(HR)=*true*) **then** *packen_oben_entfernen_punkte*(HR)
   end for

9. *zwischenkorrektur_test_und_entpacken_für_west_und_ost* (Kommunikation abwarten und dann die Informationen auspacken)

10. **for all** HR ∈ OW
    if(flag_senden(HR)=*true*) **then** *packen_ieS*(HR)
    end for

11. **for all** HR ∈ OW
    if(flag_senden_oben(HR)=*true*) **then** *packen_ieS*(HR)
    end for

12. **while** Flags zu {west,ost} x {senden, empfangen} x {werte, mvf-buffer} x {unten, oben} sind nicht alle *false* **do**
    test_und_entpacken: Das ist Algorithmus 6.19
    end do

Der folgende Algorithmus 6.19 stellt das Ende von Algorithmus 6.18 dar.

Vorbemerkung 1: Die Funktionen *'ist_abgeschickt'* und *'ist_angekommen'* basieren auf MPI-Funktionen, die testen, ob eine Nachricht abgeschickt bzw. angekommen ist.

Vorbemerkung 2: Bei diesem Algorithmus werden die zu übertragenen Informationen geteilt: Der Hauptteil der Nachricht ist die **Info**, mit allen Koordinaten, Farbenfeldwerten usw., während die 'Werte' nur die Werte der zu übertragenen Variablen speichern. Die 'Info' muss daher vor den Werten entpackt werden. Das leistet dieser Algorithmus mit den Flags flag_entpacken_Info(HR) und flag_entpacken_Werte(HR), sodass die Werte erst dann gelesen und in die Variablen geschrieben werden können, nachdem das Flag flag_entpacken_Info(HR) auf *false* gesetzt worden ist.

Vorbemerkung 3: Teile wie *entpacken_Info* werden in den Abschnitten ab 6.8.12 gebracht.

Algorithmus 6.19: *test_und_entpacken*

Input: Dieselben Informationen wie für die Hauptfunktion

1. **for all** HR ∈ OW
   if(flag_packen_Info(HR)=true und Info_ist_abgeschickt)
   then flag_packen_Info(HR) auf
   false setzen
   end for

2. Dasselbe wie in Zeile 1 machen für packen_oben_Info und packen_Werte und packen_oben_Werte, indem man in der Zeile 1 wie folgt substituiert: Info → Info-oben, Info → Werte und Info → Werte-oben.

3. **for all** HR ∈ OW
   if(flag_entpacken_Info(HR)=*true* **AND** Info(HR)_ist_angekommen)
   then *entpacken_Info*(HR) und
   setze flag_entpacken_Info(HR) auf *false*
   end for

4. **for all** HR ∈ OW
   if(flag_entpacken_Info(HR)=*false* **AND** flag_entpacken_Werte(HR)=*true*
   AND Werte(HR)_ist_angekommen)
   then *entpacken_Werte*(HR) und setze flag_entpacken_Werte(HR)
   auf *false*
   end for

5. Dasselbe wie die Zeilen 3 und 4 für 'oben' machen, d.h. alle Flags und entpacken-Befehle bekommen den Ausdruck 'oben' dazu.

6.8.5 Die Zeile 1 der Hauptfunktion der Migration.

Der Reset der *mv-vertikal*-Klassen erfolgt durch Setzen der Indizes für jede Gitterebene auf 0.

Flags setzen bei der Ost-West-Migration

Mit △Last_Osten sei die nach Osten zu schickende Lastmenge (falls > 0) bzw. von dort zu empfangende Lastmenge (falls < 0) gemeint.

△Last_Osten < 0: senden_osten = *false*, empfangen_osten = *true*.

△Last_Osten $= 0$, oder es gibt keinen Kern im Osten: senden_osten = *false*, empfangen_osten = *false*.

△Last_Osten > 0: senden_osten = *true*, empfangen_osten = *false*.

Dasselbe für den Westen analog.

6.8.6 *packen_mv*, Zeile 3 und 4 des Hauptalgorithmus

Algorithmus 6.20 berechnet *packen_mv*, je nach Himmelsrichtung HR sowie ev-Typ *ev* oder *ev2*, und je nachdem ob das für die obere oder untere Ebene gemacht werden soll.

Algorithmus 6.20: *packen_mv_HR_(oben)*
Input: Lastmenge gesamtlast, Gitterebene h

1. Bilde die Vektoren aus *ExtremeVektoren* und überführe sie aus *ExtremeVektoren* nach *mv-vertikal*(h) (h+1 bei 'oben')
2. Bilde daraus *mv-vertikal* in den anderen Ebenen(h) (h+1 bei 'oben')
3. Korrigiere *ExtremeVektoren* in den anderen Ebenen(h) (h+1 bei 'oben')

Zu Zeile 1:
Für *packen_west_mv* hat man
ev bzw. *ev2*[h].*bilde_extreme_vektoren*(gesamtlast, *mv-vertikal-west*, h)
Das gilt für *packen_ost_mv* analog.
Für *packen_oben_west_mv* wird ausgeführt:
ev bzw. *ev2*[h+1].*bilde_extreme_vektoren*(gesamtlast, *mv-vertikal-west*, h+1)
Das gilt für *packen_oben_ost_mv* analog.

Zu Zeile 2: Vgl. den nächsten Abschnitt 6.8.7

Zu Zeile 3:
Falls ev-Typ = *ev*: Entferne aus ev alle Punkte auf allen Ebenen außer h bzw. h+1 bei oben, denn für h oder h+1 bei oben sind die Punkte mit Schritt 1 entfernt.
Falls ev-Typ = *ev2*: Für Schritt 3 geschieht in diesem Fall nichts. Denn auf Ebene h bzw. h+1 im Fall oben sind die Punkte entfernt, da in *ExtremeVektoren* suchen und entfernen gemeinsam ausgeführt werden. Und außerdem gilt, dass Punkte, die übereinander liegen, nur auf einer Ebene *ev2*-Einträge haben, und daher nur für eine Ebene entfernt werden müssen.

Bemerkung: Zum gemeinsamen Packen von Punkten auf dem feineren und gröberen Gitter vergleiche man Abschnitt 6.8.3.

6.8.7 Bilde aus Vektoren einer Ebene ganz mv-vertikal, vgl. Schritt 2 von Algorithmus 6.20

Die Schritte hin zur Bildung von mv-vertikal werden hin zum Gröberen gemacht von Gitterebene h zu Gitterebene h+1, womit man rekursiv alle gröberen Gitterebenen behandelt. Auf diese Weise geht es auch von Gitterebene h zu Gitterebene h-1, also zum feineren Gitter hin, womit man mit diesem Schritt rekursiv alle feineren Gitterebenen behandelt.

Gegeben seien nun die Vektoren auf der Ebene h. Die folgenden Schritte werden Vektor für Vektor abgearbeitet.
Demnach: Gegeben sei ein Nord-Süd Vektor(f_1,f_2,g). (f_1 nördliche Koordinate, f_2 südliche Koordinate, g ost-west Koordinate)

Bestimmung der jeweils gröberen Ebene der Klasse *mv-vertikal* für eine Ebene h. Das ist der einfache Fall.

Ist g ungerade $\Rightarrow$ kein Eintrag.

Ansonsten: a = f_1, b = f_2. Ist a ungerade $\Rightarrow$ a = a + 1. Ist b ungerade $\Rightarrow$ b = b - 1.

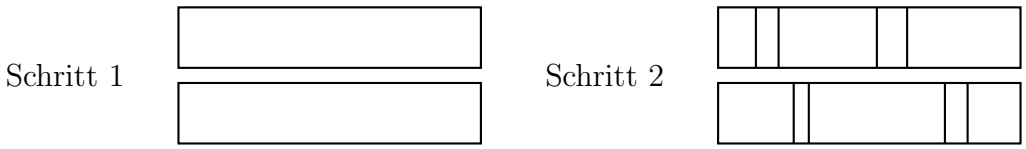


Abbildung 6.16: Die Bildung der feineren zu übertragenen Vektoren. Links sieht man die Bildung der maximal möglichen Vektoren, rechts ihre Unterbrechung.

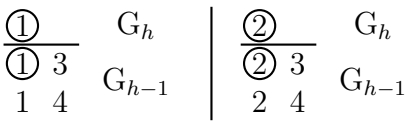


Abbildung 6.17: Lasttransfer von Kern 1 zu Kern 2, links zeigt den Fall vorher, rechts den Fall nachher. Oben liegt G(h) unten G(h-1). Die umrandeten Punkte liegen jeweils übereinander.

ist nun a $\leq$ b $\Rightarrow$ Trage Vektor (a/2,b/2,g/2) ein.

Bestimmung der jeweils feineren Ebene der Klasse *mv-vertikal* für eine Ebene h. Das ist der komplizierte Fall.

Hier sind 2 Aspekte zu berücksichtigen:

1. Welche Punkte werden gepackt?
2. Wie wird die Vektorisierung dabei erreicht?

Welche Punkte werden gepackt? Wenn (f,g)$\in$ G(h) auf Kern 1 liegt, dann befindet sich per Definitionem (2f,2g) - falls es in G(h-1) vorhanden ist - ebenfalls auf Kern 1. Insgesamt ist in diesem Fall Kern 1 zuständig für alle 4 Punkte M = {(2f,2g),(2f+1,2g),(2f,2g+1),(2f+1,2g+1)}, sofern diese Punkte die Gittergrenzen (n(h), m(h)) nicht erreichen. (Ansonsten die entsprechende Teilmenge von M.)
Bei der adaptiven Verfeinerung liegt diese Zuständigkeit vor, und beim Lastausgleich ebenfalls.

Schickt Kern 1 nun (f,g) an Kern 2, so werden alle Punkte von M, die auf Kern 1 liegen, mitgeschickt an Kern 2. Die übrigen Punkte bleiben wo sie sind. So übernimmt Kern 2 an dieser Stelle die Aufgaben von Kern 1! Natürlich müssen dann Kern 1 und 2 diese Änderungen vornehmen, und die anderen Kerne müssen über diese Änderung informiert werden. Letzteres gehört aber nicht in diesen Abschnitt 6.8.7 sondern in Abschnitt 6.8.9. Hier geht es nur um die Punktauswahl im feinen Bereich.
Man vergleiche zu den Änderungen auch das Beispiel 'vorher - nachher' in Abbildung 6.17.

Wie wird die Vektorisierung dabei erreicht? Das geschieht in zwei Schritten. Zuerst werden die Vektoren bestimmt, die alle diejenigen Punkte abdecken, die gewählt würden, wenn die Kernzugehörigkeit nicht berücksichtigt wird, vgl. Abbildung 6.16 links. Für den Vektor (f_1, f_2,g) auf Ebene h hat man den Vektor ($2f_1$, $2f_2$+1,2g) auf

Ebene h-1, sowie den Vektor ($2f_1$, $2f_2$+1,2g+1) auf Ebene h-1, falls 2g+1 im zulässigen Bereich liegt, d.h. bei 2g+1 < m(h-1). Dabei ist $2f_2$+1 durch $2f_2$ zu ersetzen, wenn $2f_2$+1 über den zulässigen Bereich n(h-1)-1 hinausgeht.

Im zweiten Schritt ruft man für diese 1 bis 2 Vektoren pro zu bearbeitendem Vektor den unteren Algorithmus 6.21 aus folgendem Grund auf, vgl. Abbildung 6.16 rechts: Nach dem Abschnitt 'Welche Punkte werden gepackt?' können auf diesem Vektor auch Punkte auf anderen Kernen liegen. In diesem Fall müssen diese beiden Vektoren an diesen Stellen unterbrochen werden. Das geht nach folgendem Prinzip:

Man durchläuft die Vektoren auf dem Kern alternierend mit dem Durchlauf auf den Zwischenräumen.

Algorithmus 6.21: Bestimmung der feinen Vektoren in mv_vertikal

1. Man sucht die erste Koordinate mit eigenem Punkt.
2. Man verlängert den dort beginnenden Vektor, soweit eigene Punkte existieren, und speichert den Vektor ab.
3. Man verlängert den nun beginnenden Zwischenraum, soweit er existiert.
4. Mit dem dort beginnenden Vektor wiederholt man dessen Verlängerung, soweit es geht, und speichert den Vektor ab.
5. Die Schritte 3 und 4 wiederholt man, bis man an die letzte Position kommt.
6. Dieser Algorithmus terminiert dann dort mit dem letzten Vektor oder Zwischenraum.

6.8.8 *packen_entfernen_Punkte*

Sei h_gesamt_variante die Gitterebene, die entweder das gröbste Gitter indiziert oder aber bei h_cap ≠ -1 h_cap ist. Das ist die gröbste Ebene, für die beim Lastausgleich Punkte übertragen werden.
Die Idee von *packen_entfernen_Punkte* besteht darin, aus der Klasse *Gitter* für alle Ebenen bis h_gesamt_variante alle Punkte zu entfernen.

Algorithmus 6.22: *packen_entfernen_Punkte* (in der Klasse *Migration*)

1. **for** h=0 **to** h_gesamt_variante
2. *entferne_ebene_dynamisch*(mv-vertikal) in der Klasse *ParallelesRechnen* der Ebene h
3. **end for**

Algorithmus 6.23: *entferne_ebene dynamisch* (in der Klasse *Gitter*)
Input: *mv-vertikal*; 'gitterebene' wird nicht übergeben, siehe unten

1. **for all** (f_1,f_2,g) ∈ *mv-vertikal*(gitterebene)

*B: Q → S

*A: P → R

Abbildung 6.18: Das Beispiel für Problemstellung und Lösung der Zwischenkorrektur. Der '*' bei A steht für die Stelle des Punktes A, der '*' bei B steht für die Stelle des Punktes B.

2. **for** i=f_1 **to** f_2

3. *hinweg-dynamisch*(i,g)

4. **end for**

5. **end for**

Bemerkung 1 zu Algorithmus 6.23: Die Funktion *entferne_ebene_dynamisch* ist definiert in der Klasse *Gitter* und hat damit Zugriff auf den Wert gitterebene, der für jedes Gitter gespeichert ist. Dadurch hat man Zugriff auf die Vektoren von *mv-vertikal*(gitterebene). Die Funktion *'hinweg-dynamisch'* ruft die Funktion *'hinweg'* als Teil der *Feldliste* der Klasse *Gitter* auf. Ferner wird die dynamische Randliste aufgrund des Entfernens des Punktes aktualisiert. Dazu wird der jeweilige Punkt aus der Klasse *DynamischeRandliste* gelöscht. Und die Nachbarpunkte, die in der Klasse *DynamischeRandliste* noch nicht enthalten sind, werden dort eingefügt, da sie jetzt am Rand sind.

Bemerkung 2: In der Klasse Gitter wird bei der Änderung der Punktmenge immer die dynamische Randliste mit aktualisiert.

6.8.9 Zwischenkorrektur

Zuerst wird die Problemstellung angegeben, dann die Lösung für eine Himmelsrichtung, einen Nachbarkern und eine Gitterebene. Danach werden die Algorithmen für den allgemeinen Fall dargestellt, vgl. den Unterabschnitt mit der Bezeichnung 'So wird diese einfache Methode zu einer komplexeren Methode'. Die Hauptalgorithmen der Zwischenkorrektur, Algorithmus 6.26 und 6.27, mussten für West und Ost gemeinsam entwickelt werden, während die Unteralgorithmen 6.24, 6.25 und 6.28 für HR geschrieben werden mit HR ∈ OW.

Die Problemstellung

Seien zwei Punkte A und B gegeben, die zueinander benachbart sind. A liege auf Kern P und B auf Kern Q. Nun soll A nach R und B nach S geschickt werden, vgl. Abbildung 6.18. Ohne weitere Arbeitsschritte würde S auf der Position A P identifizieren, und R auf der Position B den Kern Q. Das wäre aber falsch.

West	Ost

Abbildung 6.19: Der geteilte Buffer bei der Zwischenkorrektur - um Farbwerte nicht kommunizieren zu müssen.

Die Lösung des Problems

P informiert Q, bevor Q die Farbenfeldwerte um den Punkt B packt, dass A nun auf Kern R liegt, und umgekehrt Q P, dass B nun auf dem Kern S liegt, bevor P die um den Punkt A liegenden Farbenfeldwerte packt. Dann werden die Farbwerte entsprechend dieser Information gespeichert. Damit steht bei Kern Q auf der Position A der Wert R, und bei P auf der Position B der Wert S. Damit sind nach dem Verschicken die Farbwerte auf den Kernen S und R korrekt, vorausgesetzt, dass vor dem Packen die Zwischenkorrektur abgeschlossen ist.

Bemerkung 1: Wenn Q und R denselben Kern bezeichnen, muss Q informiert werden, obwohl diese Zwischenkorrektur-Information von P an R und damit an Q verschickt wird und man so Kommunikation sparen könnte. Das mögliche Problem dabei wird an einem Beispiel mit eigenen Bezeichnungen erklärt: Sei ein Punkt A auf Kern Q und ein Punkt B auf Kern R. Nun soll A von Q an R und B von R an S geschickt werden. Zwar bekommt R von A auch ohne die Zwischenkorrektur die Information, dass A nun auf R liegt. Allerdings erst, nachdem B an S geschickt wurde. So wird von Kern S fälschlich angenommen, dass A auf Q liegt. Damit ist das Beispiel beendet. Aus dem mit dem Beispiel erläuterten Grunde muss die Zwischenkorrektur auch für den Fall, dass Q und R dieselben Kerne bezeichnen, durchgeführt werden. In dem Fall wird R rechtzeitig darüber informiert, dass Q nun auf R liegt.

Bemerkung 2: Wichtig ist auch diese Idee: Die Farben der Zielkerne brauchen nicht mitverschickt werden, weil man weiß, wenn P etwas nach Westen oder Osten schickt, an welchen Kern das geht. Dafür muss aber der Buffer, der bei der Zwischenkorrektur verschickt wird, nach Westen und Osten geteilt sein, vgl. Abbildung 6.19, damit der Empfänger der Buffer zuordnen kann, wohin der verschickende Kern seine Punkte migriert hat.

So wird diese einfache Methode zu einer komplexeren Methode

Mit Multi-Level-Kerntopologie - kurz ML-Kerntopologie - ist im Folgenden in diesem Abschnitt die Topologie bzgl. Nachbarpunkten gemeint, allerdings für alle beteiligten Gitterebenen. Zwei Kerne sind genau dann topologische Nachbarn, wenn auf einer Gitterebene bis *h_gesamt_variante* eine Punktnachbarschaft existiert.
(*h_gesamt_variante* ist *h_gesamt*, falls keine cap existiert, und andernfalls *h_cap*.)

Oben wurde der Fall für eine Gitterebene und einen Nachbarkern erläutert. Es wird komplizierter, weil man mehrere Nachbarn bzgl. der ML-Kerntopologie hat, weil die Migration nach Osten und Westen erfolgen kann, und weil das Ganze auf vielen Ebenen, also Multi-Level erfolgt. Denn auf allen Gitterebenen bis *h_gesamt_variante* muss diese Korrektur erfolgen.

Die Algorithmen entsprechend der erläuterten Vorgehensweise

Bevor die Hauptidee und die Algorithmen vorgestellt werden, eine wichtige Vorüberlegung: Je nachdem, ob man nach Osten oder Westen verschickt, werden für jeden anderen Kern ein oder zwei Bufferheader benötigt, um die Anzahlen der übertragenen Punkte abzuspeichern. Die Variable, die festhält, ob nur nach Westen, nur nach Osten, nach Westen und Osten oder gar nicht geschickt wird, heißt *sow* (senden-ost-west) und wird mit dem Buffer mitgeschickt. Entsprechend kann der Sender die Bufferheader anlegen und der Empfänger sie ablesen.

Idee von Algorithmus 6.24: Gehe die Punkte (f,g) aus *mv-vertikal* durch und bestimme dann mithilfe von *selbstnachbarn* deren Nachbarpunkte (k,l). Teile dann dem Kern des Punkts (k,l) die Koordinaten (f,g) mit, sowie durch eine Aufteilung des Buffers in Blöcke, an wen der Farbwert und die Koordinaten dieses Gitterpunktes geschickt werden.

Algorithmus 6.24: *zwischenkorrektur_für_eine_ebene_packen*
Input: Angabe der Ebene h, Himmelsrichtung HR $\in$ OW, *mv-vertikal* für HR. Ferner muss der Buffer für alle Farben für die Himmelsrichtung HR vorbereitet sein, d.h. es muss dem entpackenden Kern mitgeteilt werden, dass für diese Himmelsrichtung HR gepackt wird. Für die Ebene h geschieht das durch einen Aufruf von Algorithmus 6.25.
Output: Es wird in die vorbereiteten Buffer bzgl. Kerne der Kerntopologie geschrieben

1. **for all** (f1,f2,g)$\in$ *mv-vertikal*-HR(h)
2. **for** j = f_1 **to** f_2
3. **switch**(255 - *Zahl-Selbstnachbarn*(j,g))
4. **case** $HR_1 \cdots HR_k$:
5. **for** i=1 **to** k
6. **if**(*farbenfeld*(h)(HR_i(j,g)) $\geq$ 0)
7. Packe (f,g) in Buffer der Farbe *farbenfeld*(h)(HR_i(j,g))
8. **end for**
9. **end switch**
10. **end for**
11. **end for**

Bemerkung 1: *Selbstnachbarn*(j,g) gibt die Nachbarschaft zu eigenen Punkten an, und 255 - *Selbstnachbarn* die Nachbarschaft zu nicht eigenen Punkten oder Koordinaten, auf denen keine Punkte existieren.

Bemerkung 2: Ein doppeltes Packen von Punkten ist durch ein Testfeld vermeidbar.

Algorithmus 6.25: *zwischenkorrektur_für_alle_ebenen_packen*
Input: Himmelsrichtung HR, *mv-vertikal* für HR, sow. Auch hier müssen die Buffer für alle Farben für die Himmelsrichtung HR vorbereitet sein, vgl. den Input von Algorithmus 6.24.

1. **for** h=0 **to** h_gesamt_variante-1
2. *zwischenkorrektur_für_eine_ebene_packen*(h,HR,*mv-vertikal*)
3. Schreibe für alle Farben der ML-Kerntopologie die Punktanzahl zu der Farbe und Himmelsrichtung in den Bufferheader in Abhängigkeit von (h,HR, *sow*).
4. **end for**

Der folgende Algorithmus 6.26 entspricht dem Schritt 6 des Hauptalgorithmus 6.18 der Klasse *Migration*.

Algorithmus 6.26: *zwischenkorrektur_packen_und_verschicken_für_west_und_ost*

1. *sow* bestimmen
2. Bufferheader entsprechend *sow* vorbereiten
3. **if** Buffer wird nach Westen geschickt (wenn *sow* entsprechend bestimmt ist)
4. Buffer für West vorbereiten für alle Kerne der Kerntopologie (alle Indizes im Westen auf 0; dabei sind die Buffer entsprechend *sow* vorbereitet). Damit ist der Buffer für Algorithmus 6.25 bereit.
5. *zwischenkorrektur_für_alle_ebenen_packen*(West, *mv-vertikal-west*, *sow*) (Algorithmus 6.25)
6. **end if**
7. Analog für den Osten zu den Schritten 3 bis 6.
8. Verschicke alle Buffer bzgl. der ML-Kerntopologie

Bemerkung: Zu *sow*: Z.B.: **if** *sow* = nur_Westen **then** schreibe nur für Westen.

Damit ist der erste Teil der Zwischenkorrektur fertig. In einem zweiten Schritt müssen die verschickten Buffer nach dem Erhalt entpackt werden.

Der folgende Algorithmus 6.27 entspricht dem Schritt 9 des Hauptalgorithmus 6.18 der Klasse *Migration*:

Algorithmus 6.27: *zwischenkorrektur_test_und_entpacken_für_west_und_ost*

1. Setze die Kerne der ML-Kerntopologie als unmarkiert bzgl. des Entpackens;
2. **while** nicht alle Kerne der ML-Kerntopologie sind als entpackt markiert **do**

3. **for all** Kern k ∈ Kerntopologie
4. **if** Nachricht für k angekommen, und
   k noch nicht als entpackt markiert **then**
5. **if**(*sow*(k) für Westen = *true*) **then**
6. *zwischenkorrektur_entpacke_buffer_bzgl._des_kerns_k_nach_west* (*sow*)(Algorithmus 6.28)
7. Führe die Schritte 5 und 6 analog für den Osten durch
8. Markiere Kern k als entpackt
9. **end if**
10. **end for**
11. **end do**

Die Kerntopologie wurde oben im Unterabschnitt mit der Überschrift 'So wird diese einfache Methode zu einer komplexeren Methode' als Multi-Level-Kerntopologie angegeben. ZK steht für Zwischenkorrektur.

Algorithmus 6.28: *zwischenkorrektur_entpacke_buffer_bzgl._des_kerns_k_nach_HR*
Input: Farbe k, HR ∈ OW, sow

1. **for** h=0 **to** h_gesamt_variante-1
2. **for all** (f,g) ∈ Buffer des Kerns k(Ebene h, HR, sow)
3. *farbenfeld*(Ebene h)(f,g) = HR(K)
4. **end for**
5. **end for**

Der Ausdruck '**for all** (f,g) ∈ Buffer des Kerns k(Ebene h, HR, sow)' ist dabei so zu verstehen: Der Buffer ist in Blöcke aufgeteilt je nach Himmelsrichtung HR, nach *sow* und Gitterebene h. Im Bufferheader kann man zu jedem Block seine Größe nachlesen.

Bemerkung 1: Aufgrund von *sow* gilt: Ist *sow* = nur_Westen, dann wird nur dieser Header kommuniziert. Bei *sow* = nur_Osten analog. Bei *sow* = Westen_und_Osten werden die 2 Header aufeinanderfolgend abgespeichert.

Bemerkung 2: Dieser Algorithmus ist angelegt für alle Gitterebenen bis $h_{gesamt\_variante}$.

Bemerkung 3 zur letzten Zeile: Man kennt die Zielfarbe HR(k), da die Migrationskerntopologie für alle Kerne auf allen Kernen bekannt ist.

Bemerkung 4 zu den Algorithmen insgesamt: Für jeden Kern außer dem eigenen werden die Buffer angelegt. Dass in Algorithmus 6.28 Buffer k(Ebenen h,HR,*sow*) steht

bedeutet: Für alle Ebenen und die 2 möglichen HR'en wird in denselben entsprechend *sow* organisierten Buffer gepackt bzw. aus demselben Buffer entpackt.

6.8.10 *packen_ieS*

Hier wird mit *packen_ieS* wieder ein Packen nach Westen bzw. Osten erläutert. Verschickt werden dabei Vektoren aus *mv_vertikal*, die von Norden nach Süden verlaufen. Für Norden und Süden verläuft alles analog mit Vektoren, die von Osten nach Westen verlaufen.

Bei packen_ieS (ieS = im eigentlichen Sinne) geschieht das eigentliche Packen, nämlich das Beschreiben der Buffer, sodass der Empfänger daraus seine Strukturänderungen vornehmen kann. Die Werte werden separat verschickt. Es geht dabei um drei Aspekte:

1. Koordinaten packen
2. Farben packen auf den die verschickten Vektoren umgebenden Punkten.
3. Die Eigenschaftspunktinformationen (*feld_vergangen*) werden auf den Vektoren, die verschickt werden, gepackt. Das geschieht zusammen mit der Durchführung der Änderungen beim Packen in Bezug auf *EP-SNB-Update*.

1. Koordinaten packen

Man speichert die Anzahl-Indizes für alle Hierarchieebenen, d.h. wie viele Vektoren jeweils gespeichert werden. Danach werden die Koordinaten der Vektoren gespeichert. Das 'danach' bezieht sich dabei auf die Buffer.

Algorithmus 6.29: *koordinaten_packen*

1. Schreibe die Indizes von *mv-vertikal* in den Buffer - für alle Ebenen.
2. Setze i = *h_gesamt_variante*, also auf die Position nach den Indizes.
3. **for** h = 0 **to** *h_gesamt_variante*
4. **for all** (f1,f2,g)$\in$ *mv-vertikal-HR*(h)
5. **for** j = f_1 **to** f_2
6. buffer(i) = f1; i=i+1; buffer(i) = f2; i=i+1; buffer(i) = g; i=i+1;
7. **end for**
8. buffer(h) = i
9. **end for**
10. **end for**

2. Farben in der umgebenden Punktmenge packen

Zuerst ist zu erwähnen, dass diese Aktion für alle Gitterebenen bis h_gesamt_variante erfolgen muss. In der Klasse *FarbenfelderVerschicken* wird das erledigt: Die Funktion *packen_HR* in dieser Klasse durchläuft alle Gitterebenen, und übergibt der Klasse *FarbenfeldVerschicken* die Vektoren und das *Farbenfeld*. Es wird der Bufferindex in der Klasse *FarbenfeldVerschicken* gesetzt.

Algorithmus 6.30: *packen_HR* (Himmelsrichtung) in der Klasse *FarbenfelderVerschicken*
Input: *mv-vertikal-HR*, das Farbenfeld aller Ebenen wird in der Struktur *farbenfelder* gespeichert

1. index in *FarbenfeldVerschicken* = 0
2. **for all** h = 0 **to** h_gesamt_variante
3. index = *farbenfeldinformation_packen*(*mv-vertikal-HR*(h),*farbenfelder*(h), n(h), m(h), index)
4. **end for**

Algorithmus 6.31: *farbenfeldinformation_packen* in der Klasse *FarbenfeldVerschicken*
Input: *mv-vertikal-HR-einer-Ebene*, *Farbenfeld*, n und m (für Ränder), Index index
(Input: (dauerhaft) Buffer buffer)
Output: Index zurück

1. Markiere alle Punkte der Vektoren *mv-vertikal-HR-einer-Ebene* der Ebene in einem Testfeld
2. **for all** (f_1, f_2, g) $\in$ *mv-vertikal-HR-einer-Ebene*
3. Der Vektor (f_1, f_2, g) wird so umlaufen:
   + + +
   + o +
   + o +
   + o +
   + o +
   + + +
   Dabei werden zwei Nord-Süd-Schleifen gebildet plus die Punkte oben und unten → Koordinaten (f,g) auf den mit '+' markierten Punkten
4. **if**(f,g) nicht markiert
5. buffer(index) = *Farbenfeld*(f,g); index=index+1;
6. **end for**
7. Lösche Markierungen des Testfeldes

Die Idee dabei: Die umlaufenden Farbwerte werden gebraucht, weil der Empfänger diese Farbmarkierung herstellen muss. Zu den Schritten 4 und 5: Auf den Vektoren wird beim Sender und Empfänger die Farbe des neuen Kernes eingetragen. Diese Farbwerte müssen nicht mitgepackt werden. Da Sender und Empfänger dieselben Vektoren kennen, ist bei beiden das Testfeld gleich. Deshalb wird nur dann geschrieben bzw. gelesen, wenn die Stelle im Testfeld nicht markiert wurde. Nur dann ist der Farbwert beim Empfänger nicht bekannt.

Bemerkung 1: Also muss beim Empfangen dasselbe Testfeld generiert werden.

Bemerkung 2: Die 'übergebene Ebene' von *mv-vertikal* besteht aus Zeigern auf f_1, f_2 und g sowie der Anzahl der Vektoren

Bemerkung 3: Man könnte auch die Koordinaten geschriebener Farbwerte markieren, damit kein Farbwert doppelt gepackt wird.

Bemerkung 4: In der aktuellen Version des Programms wird keine EP-Information auf dem umgebenden Rand geschrieben, da die Randinformation mithilfe der EP-Randkommunikation ermittelt wird.

Bemerkung 5: Man braucht hier Anzahlen der Punkte nicht zu speichern, weil alle Indizes durch die Vektoren aus *mv-vertikal* bestimmt sind.

3. Eigenschaftspunkte packen, und auch ggf. das Markierungsfeld für neue Gitterpunkte behandeln

Hauptidee fürs Packen und Entpacken:

1. Es werden nur Bits kommuniziert. Das geschieht mithilfe der Klassen *bool_int* und *bool_short_int*. Dabei bekommen diese Klassen zum Lesen und Schreiben der Bits die Bufferzeiger der aktuellen Stelle übergeben.

2. Es wird *mv-vertikal* komplett durchlaufen, also alle Vektoren für alle Ebenen.

3. Wenn man beides kombiniert, dann reicht es als Darstellung aus, in Bezug auf eine Koordinate (f,g) mitzuteilen, was für Bits dort zu schreiben sind, bzw. was dort zu lesen ist.

Vor der Erläuterung, welche Bit's pro übertragenem Gitterpunkt gespeichert werden, werden nun 2 Punkte vorausgeschickt:

a. Es erfolgt eine Weitergabe des Markierungsfeldes für neu entstandene Gitterpunkte. Dabei wird ein Bit pro möglichem neuen Gitterpunkt benötigt.

b. Die Informationen von *feld_vergangen* müssen übermittelt werden. Auch muss die Datenstruktur *EP-SNB-Update* entsprechend geändert werden. Dafür braucht auch nur ein Bit verschickt zu werden. Diese Strukturen müssen auch beim Weiterverschicken korrekt geführt werden. Um das geschickt zu machen, erfolgt ein Eingriff in die Strukturen EP^+ und EP^- der Klasse *ParalleleAdaptiveVerfeinerung*.

Das Bit für das Markierungsfeld wird auf der Gitterebene gebraucht, auf der der Lastausgleich erfolgen soll, weil die parallele adaptive Verfeinerung dafür geschieht. Das Bit für *feld_vergangen* wird auf den Gitterebenen transferiert, die die Eigenschaftspunktinformationen haben. Werden beide Informationen auf einer Ebene gebraucht, werden die Bits hintereinander gepackt.

Die Weitergabe des Markierungsfeldes. Es werden alle Mehrgitter-Schritte nach dem Lastausgleich gemacht. Das ist auch unter Lastverteilungsaspekten genau richtig. Das Programm muss dafür wissen, wo neue Gitterpunkte dazukommen, weil diese bei der biquadratischen Interpolation anders behandelt werden. Deswegen wird auf der entsprechenden Gitterebene pro migriertem Gitterpunkt ein Bit zur Identifikation neuer Punkte mitgepackt.

Der Eingriff in die Strukturen EP^+ und EP^-. Die Programmierung ist wie folgt: Nach (!) der adaptiven Verfeinerung erfolgt ein Reset von EP^+ und EP^- . Damit hat die Migration und damit der Lastausgleich die Möglichkeit, diese Strukturen vor der nächsten adaptiven Verfeinerung zu beschreiben.

Das Schreiben des Bits: Das Packen in Bezug auf die Eigenschaftspunkte, bzw. für *feld_vergangen*. Sei ein Gitterpunkt (f,g) $\in$ *mv-vertikal* gegeben. Nun zu dem Beschreiben des einen Bits für diesen Gitterpunkt für die Eigenschaftspunkte: Falls durch das Entpacken (s.u.) einer vorausgegangenen Lastausgleichsrunde bereits in EP^+_h (für EP^+auf Ebene h) bzw. EP^-_h (für EP^-auf Ebene h) ein Eintrag an der Stelle (f,g) erfolgt ist, wird bei einem Eintrag in EP^+ das Bit *true* gepackt, bei einem Eintrag in EP^- das Bit *false* gepackt. Entsprechend diesen Bits wird beim Entpacken *feld_vergangen* beschrieben.

Falls kein Eintrag erfolgt ist in EP^+_h bzw. EP^-_h, wird das Bit entsprechend dem Eintrag in das boolsche Feld *feld_vergangen* gepackt, und zwar *true* bei *feld_vergangen*(f,g) = *true*, und *false* andernfalls. Beim Verschicken wird dann aber *feld_vergangen*(f,g) auf *false* gesetzt.

Die Handhabung der Datenstruktur *EP-SNB-Update* wegen der Bit-Übermittlung von *feld_vergangen*. In dem Fall, wenn *feld_vergangen* von *true* auf *false* gesetzt wird, muss allerdings die Funktion *hinweg(f,g)* der Klasse *EP-SNB-Update* auf dem sendenden Kern aufgerufen werden, weil es den Gitterpunkt mit der Verfeinerungseigenschaft auf diesem Kern gab, jetzt aber nicht mehr gibt. Wird der Gitterpunkt entsprechend dem ersteren Fall von packen aufgrund eines EP^+ bzw. EP^- Eintrages nur weitergeschickt, wurde *EP-SNB-Update* noch nicht verändert, und es erübrigt sich eine Änderung von *EP-SNB-Update*.

Der Abschluss der letzten Lastausgleichsrunde - Aktualisierung von *EP-SNB-UPdate*. Es wird im Anschluss an die letzte Lastausgleichsrunde auf der groben Gitterebene der *parallelen adaptiven Verfeinerung* das Folgende durchgeführt: Für alle Punkte aus EP^+_h wird die Funktion *hinzu* der Klasse *EP-SNB-Update* aufgerufen, sowie für alle Punkte aus EP^-_h wird die Funktion *hinweg* derselben Klasse aufgerufen. Danach werden die Punkte aus EP^+_h und EP^-_h entfernt. So geschieht

die Änderung der Datenstruktur *EP-SNB-Update* aufgrund der verschickten Punkte erst am Ende des Lastausgleiches, sodass die *EP-SNB-Update*-Änderungen nur dort erfolgen, wo der Gitterpunkt endgültig entfernt und eingefügt worden ist - bzgl. des gesamten Ost-West-Lastausgleiches.

Zur Änderung von *feld_vergangen* beim Entpacken vergleiche man Abschnitt 6.8.12.

6.8.11 Das Packen der Werte

Es werden alle Gitterebenen durchlaufen, und auf jeder Ebene die *mv-vertikal* Vektoren. Für alle betroffenen Variablen werden dann die double-Werte in die Buffer geschrieben, zum Beispiel bei *packen_werte_west* in den Buffer zu dem westlichen Kern:

Algorithmus 6.32: *packen_werte-HR* (HR Himmelsrichtung)

1. i = 0
2. **for** h = 0 **to** *h_gesamt_variante*
3. **for all** (f1,f2,g)∈ *mv-vertikal-HR*(h)
4. **for** j = f_1 **to** f_2
5. buffer-HR(i) = variable$_1$(j,g), i=i+1, $\cdots$,
   buffer-HR(i) = variable$_k$(j,g), i=i+1
6. **end for**
7. **end for**
8. **end for**

6.8.12 Entpacken Information

Beim Packen gliedert sich die Aufgabe der Migration bis auf den Wertetransport in die 3 Teile *packen_mv*, *packen_abschliessen* und *packen_ieS*. Diese Teile werden in der Hauptfunktion der Migration separat aufgerufen. Für das Entpacken gibt es keine derartige Gliederung, aber die folgenden 5 Aufgaben:

1. Aus dem Buffer lesen und damit *mv-vertikal* bilden
2. Farbenfeldinformation entpacken
3. Entpacken *Eigenschaftspunkte ggf. mit Markierungsfeld*
4. Einfügen der transferierten Vektoren in die Struktur *Gitter*
5. Einfügen in *ev2*

Bemerkung: Die ersten 3 Aufgaben basieren auf der Auswertung der Buffer, und die Teile 4 und 5 können dann schon mit den gewonnenen Informationen, die in *mv-vertikal* stehen, realisiert werden.

Den Buffer lesen, und damit *mv-vertikal* bilden

Man durchläuft die Gitterebenen. Zuerst wird die Anzahl der übertragenen Gitterpunkte pro Gitterebene aus dem Buffer gelesen und in *mv-vertikal* eingetragen. Danach werden die Vektoren aus dem Buffer gelesen und in *mv-vertikal* eingetragen.

Farbenfeldinformation entpacken

Man verwendet dieselben Algorithmen wie beim Packen, also Algorithmus 6.30 und 6.31, mit diesen Änderungen:

1. Die Algorithmen heißen hier *entpacken_HR* und *farbenfeldinformation_entpacken*, statt *packen_HR* und *farbenfeldinformation_packen*.
2. *entpacken_HR* (Himmelsrichtung) ruft *farbenfeldinformation_entpacken* auf.
3. In *farbenfeldinformation_entpacken* lautet die 5. Zeile:
   farbenfeld(f,g) = buffer(index); index=index+1

Wichtig ist, dass sowohl beim Packen wie Entpacken dasselbe Testfeld verwendet wird.

Entpacke Eigenschaftspunkte, und behandele das Markierungsfeld für neu entstandene Gitterpunkte auf denselben Ebenen wie beim Packen.

Man hat hier dieselbe Situation wie beim Packen: Da für alle Gitterebenen die Vektoren mit den Punkten (f,g) durchlaufen werden, und jeweils die Punktinformation von (f,g) bitweise entpackt werden, braucht nur erläutert werden, wie man mit dem einzelnen Bit zu der Koordinate (f,g) umgeht. Das bezieht sich auf die EP-Strukturen, *feld_vergangen* sowie das *Markierungsfeld*.

Zum *Markierungsfeld*. Es liegt dieselbe Situation wie beim Packen vor, nur dass hier das *Markierungsfeld* für neue Gitterpunkte an der Gitterposition in der jeweiligen Gitterebene h auf den Wert des gelesenen Bits gesetzt wird.

Zu den Eigenschaftspunkten. Es seien die Gitterebene h und die Koordinaten (f,g) vorgegeben. Und es sei ein Bit ausgelesen worden.
Dieses Bit wird in *feld-vergangen*(f,g) eingefügt. Falls es *true* ist, wird *ep_plus_h.hinzu*(f,g) und *ep_minus_h.hinweg*(f,g) aufgerufen, bzw. bei *false ep_plus_h.hinweg*(f,g) und *ep_minus_h.hinzu(f,g)* ausgeführt. Das hat in dem Fall, wenn der Punkt nicht mehr weiterverschickt wird, auch eine Auswirkung auf *EP-SNB-Update*, vgl. den schon behandelten Abschnitt 'Der Abschluss der letzten Lastausgleichsrunde - Aktualisierung von *EP-SNB-Update*'. Das Handhaben der EP-Strukturen beim Entpacken hat Einfluss auf die Bearbeitung dieser Strukturen beim nächsten Packen, wenn es ums Weiterverschicken geht, siehe den behandelten Abschnitt 'Das Schreiben des Bits: Das Packen in Bezug auf die Eigenschaftspunkte, bzw. für *feld_vergangen*'.

Bemerkung 1: Beim Weiterverschicken eines Punktes, also beim nächsten Packen, wird, wie oben erläutert, ein gesetztes Flag für *feld-vergangen* wieder zurückgesetzt.

Bemerkung 2: Es wird, statt nur ein Feld *feld_vergangen* zu führen, immer eine *Punktfeldliste* für *feld_vergangen* und auch für *feld_aktuell* geführt.

Einfügen der transferierten Vektoren in die Struktur Gitter

Beim Entpacken werden Punkte ins Gitter eingefügt. Das funktioniert analog zu Algorithmus 6.22, nur mit dem Befehl *einfügen_ebene_dynamisch*(*mv-vertikal*(h)) statt des Befehls *entferne_ebene_dynamisch*(*mv-vertikal*(h)), die sich beide auf die Klasse *ParallelesRechnen* der Ebene h beziehen.

Der Befehl *einfügen_ebene_dynamisch* bewirkt zweierlei:

1. Es wird *mv-vertikal*(h) durchlaufen und dabei werden die Gitterpunkte der Klasse *Gitter* der Ebene h eingefügt.

2. Es wird die Klasse *DynamischerRand* der Ebene h aktualisiert. Das ist eine Option, die aber immer gewählt wird, damit eine Randpunktliste verfügbar ist. Zum Aktualisieren prüft man, ob der neue Punkt am Rand liegt. In dem Fall wird er eingefügt. Man prüft, ob die umliegenden Punkte nun nicht mehr am Rand liegen. Solche Punkte werden dann aus der Klasse *DynamischerRand* der Ebene h entfernt.

Einfügen in *ev* bzw. *ev2*

Es gibt die Option, die Strukturen *ev* oder *ev2* zu verwenden.

Zuerst zum Einfügen in *ev*: Man fügt für alle Ebenen, auf denen *ev* gebraucht wird, alle Gitterpunkte entsprechend den Vektoren von *mv-vertikal* ein.

Zum Einfügen in *ev2*: Für die Ebene 0 werden alle Punkte entsprechend *mv-vertikal*(0) eingefügt. Für die anderen Ebenen h werden nur die Punkte von *mv-vertikal*(h) eingefügt, die darunter keinen Gitterpunkt haben.
Deswegen ist es unverzichtbar, diese Funktion aufzurufen, nachdem die Punkte in die Klasse *Gitter* eingefügt wurden! Dann kann man in der Klasse *Gitter* nachsehen, ob eine Ebene tiefer Gitterpunkte existieren.

6.8.13 Die Funktion *entpacken_werte-HR*

Diese Funktion arbeitet analog zu *packen_werte-HR*. Man hat nur statt der Befehle buffer-HR(i) = variable$_l$(j,g) die Befehle variable$_l$(j,g) = buffer-HR(i) auszuführen.

6.9 Die Lastwertekorrektur - Teil 2

6.9.1 Worum es geht

Bei der Lastwertekorrektur geht es darum, die Last von Punkten am echten Rand mit 0 zu bewerten und für alle anderen Punkte mit 1. Das macht dann Sinn, wenn sich aufgrund des Mehrgitterverfahrens die Hauptrechenlast im Inneren des Gitters befindet, und auf dem echten Rand selber nur wenig gerechnet wird, was z.B. bei der Randinterpolation geschieht.

6.9.2 Die 2 Möglichkeiten

Es gibt hier 2 Möglichkeiten, die Lastwertekorrektur zu realisieren. Beide werden kurz beschrieben, realisiert wird aber nur die zweite Methode.

1. Möglichkeit: Änderungen erfassen

Man könnte das so lösen: Man hält 2 Punktmengen, die Menge der echten Randpunkte und die Menge der inneren Punkte. Jedes Mal, wenn sich Einträge im *Farbenfeld* ändern, von Werten < 0 auf ≥ 0, und umgekehrt, werden diese 'Datenstrukturen' auf Änderungen überprüft. Man hat einen Punkttransfer dieser Datenstrukturen: Menge der echten Randpunkte $\leftrightarrow$ innere Punktmenge. Das Problem: Es gibt im Programm sehr viele Stellen mit Änderungen im *Farbenfeld*. Damit würde diese Lösung das Programm insgesamt komplexer machen. Deswegen wurde dieser Ansatz nicht realisiert.

2. Möglichkeit: Verwendung eines Merkfeldes

Der erste Schritt hierbei besteht darin, dass, basierend auf der Randstruktur sowie der dynamischen Randfeldliste, die die Randpunkte enthält, ein Merkfeld für Randpunkte gebildet wird, vgl. Abschnitt 6.9.3. In einem zweiten Schritt wird daraus die Anzahl der echten Randpunkte bestimmt, vgl. Abschnitt 6.9.4. Daraus wird dann die Gesamtanzahl an Gitterpunkten mit Last 1 festgestellt. In einem dritten Schritt wird ein Algorithmus der Klasse *ExtremeVektoren* bestimmt, der Vektorenlisten packt, die eine gewisse vorgegebene Anzahl an Punkten mit Last 1 enthält. Dieses wurde realisiert in Abschnitt 6.7.5 mit Algorithmus 6.17, der auf Algorithmus 6.15 für die *1-dimensionaleFeldlisteMf* basiert.

6.9.3 Schritt 1: Die Bestimmung des *Merkfeldes* für Randpunkte

Der folgende Algorithmus muss für jede Gitterebene, für die die Klasse *ExtremeVektoren* gebraucht wird, aufgerufen werden. Entsprechend viele *Merkfelder* gibt es. Dabei wird das *Merkfeld* als *Punktfeldliste* gespeichert.
Die Idee dieses Algorithmus besteht darin, den Rand zu durchlaufen (was in O(Randpunktzahl) möglich ist), und zu prüfen, ob der Randpunkt eine Nachbarkoordinate ohne existierenden Punkt auf dem Gitter hat.

Algorithmus 6.33: *bestimme_merkfeld_für_randpunkte*

1. Reset *Merkfeld*
2. **for** (f,g) $\in$ Rand
3. **if** (f=0 **OR** f=n-1 **OR** g=0 **OR** g=m-1)
4. Einfügen ins *Merkfeld*(f,g)
5. **continue** (springe in die Schleife zurück)
6. **end if**

7. **switch**(255-*Zahl-Selbstnachbarn*(f,g))
8. **case** Zahl(HR_1, .., HR_k):
9. **for** i von 1 bis k:
10. **if**(*farbenfeld*(HR_i(f,g)))< 0
11. Einfügen ins *Merkfeld*(f,g)
12. **continue** (springe in die Schleife zurück, also zu Zeile 2)
13. **end if**
14. **end for**
15. **end switch**
16. **end for**

Bemerkung: Die Zeilen 9 - 14 sind nicht als Schleife programmiert.

6.9.4 Schritt 2: Zählen der Punkte

Das Zählen der Gitterpunkte, die die Eigenschaft der Lastwerte Korrektur erfüllen, geschieht für eine Ebene h wie folgt: Das *Merkfeld* wird durchlaufen, und falls der jeweilige Punkt existiert, wird der Zähler inkrementiert.

Kapitel 7

Die theoretische Laufzeit des Balancers

7.1 Die Punktlasten des Balancers

Wegen der sich anschließenden Synchronisation muss die Laufzeitabschätzung alle Kerne einbeziehen. Es ist das Maximum an Laufzeiten des Balancers über alle Kerne in Bezug auf das Balancing zu bestimmen.

7.1.1 Definitionen

Zuerst werden die für diese Untersuchung notwendigen Migrationslasten definiert. Dabei seien die zu migrierenden Punktmengen schon ausgewählt.

Die Punkte sind nach *ev* oder *ev2* ausgewählt. Unterschieden wird beides hier nur danach, wie die Strukturen *ev* bzw. *ev2* geändert werden. *ev* wird anders geändert als *ev2*.

Definition 7.1: Punktmengen

Sei K ein Kern und L der westliche oder östliche oder nördliche oder südliche Nachbarkern. Dann verschickt er auf der Ebene h die Punkte des Migrationsgebietes MG(K,L,h). Ferner muss der Außenrand der Migrationsgebiete definiert werden, weil für diese Punkte auch Arbeit geleistet werden muss. MAR(K,L,h) ist der Außenrand der Migration von K nach L auf der Ebene h. Die Menge der gepackten Vektoren sei MV(K,L,h) (Migration Vektoren).

Für einen Balancing-Schritt auf einer Gitterebene h müssen Punkte aller Gitterebenen gepackt, übermittelt und entpackt werden - bis auf Einschränkungen im Grobgitterbereich bei der Kappe.

Definition 7.2: Gesamtanzahl der Punktmengen

Sei K ein Kern und L ein Nachbarkern. Dann wird die Anzahl an Punkten von MG(K,L,h) mit MGG(K,L,h) bezeichnet (Migrationsgebietgröße). Die Anzahl der Punkte des Au-

ßenrandes MAR(K,L,h) wird mit MARG(K,L,h) bezeichnet (Migration Außenrand Gesamtanzahl). Und die Anzahl der gepackten Vektoren MV(K,L,h) wird als MVG(K,L,h) (Migration Vektoren Gesamtanzahl) definiert.

Definition 7.3: Die Gesamtanzahlen für Multi-Level-Punktmengen (ML-Punktmengen)

Sei wiederum K der Kern und L ein Nachbarkern bzgl. des Lastausgleiches.

1. Dann ist GMGG(K,L) die gesamte Anzahl an verschickten Gitterpunkten für alle Ebenen, GARG(K,L) die Anzahl aller Gitterpunkte der Außenränder der Migration aller Gitterebenen, und EMGG(K,L) die Menge aller auf Verfeinerungsentscheidungsebenen gelegenen verschickten Gitterpunkte.

2. Zu den Punktmengen für die Behandlung der Punkte aus *ExtremeVektoren*: Die Anzahl der Punkte der Migration der Gitterpunkte aller an der Auswahl von Gitterpunkten beteiligten Ebenen sei MMGG(K,L). Die folgende ML-Punktmenge wird fürs Entfernen von Gitterpunkten in *ExtremeVektoren* verwendet. Für alle diese Ebenen, mit Ausnahme einer bestimmten Ebene h, sei die Anzahl eMMGG(K,L,h). Gebraucht wird diese Zahl für die Behandlung aller Ebenen außer der Vektorenauswahlebene, weil Suchen und Löschen immer gemeinsam erfolgen. Wird *ev* genommen, wird eMMGG benötigt. Bei *ev2* allerdings nicht, weil in dem Fall nur die auswählende Gitterebene zu löschen ist.

3. Zu den Lasten der Vektoren bei der Projektion zur Bildung von *mv-vertikal*: Die Last für alle Vektoren auf gröberen Gittern sei durch den Wert hEMVG(K,L,h) bestimmt(höhere Ebenen für MVG). Die Last aller Vektoren auf allen Ebenen ist GMVG(K,L). Die Last der Gebiete für die niedrigeren Ebenen sei nEMGG(K,L,h) (niedrigere Ebenen MGG).

Bemerkung: Die Größen, bis auf die Durchführung der Projektion sowie fürs Suchen und Entfernen in *ExtremeVektoren*, beziehen sich sowohl auf das Senden, wie auf das Empfangen von Last, d.h. sie werden gebraucht sowohl für den Sender wie für den Empfänger.

7.1.2 Die Ableitung der Größen aus Definition 7.3 aus den Größen von Definition 7.2

Nach diesen Definitionen können durch folgende Summen die in Definition 7.3 eingeführten Größen durch die Einzelkosten aus Definition 7.2 bestimmt werden:

Satz 7.1: Sei K ein Kern und L ein östlicher oder westlicher Nachbar. Dann gelten:

1.
$$GMGG(K,L) = \sum_{\text{h=0 bis h}_{gesamt}\text{-1}} MGG(K,L,h)$$

(alle Punkte, alle Ebenen)

$$GARG(K,L) = \sum_{\text{h=0 bis h}_{gesamt}\text{-1}} MARG(K,L,h)$$

(alle Randpunkte, alle Ebenen)

$$EMGG(K,L) = \sum_{\text{h=1 bis h}_{min}} MGG(K,L,h)$$

(alle Entscheidungspunkte)

2.
$$MMGG(K,L) = \sum_{\text{h=0 bis h}_{min}\text{-1}} MGG(K,L,h)$$

(Auswahl aller Punkte der Klasse *ExtremeVektoren*)

$$eMMGG(K,L,h) = \sum_{\text{k=0 bis h}_{min}\text{-1, k} \neq \text{h}} MGG(K,L,k)$$

(Auswahl derselben Punkte bis auf die die Vektoren bestimmende Ebene)

3.
$$hEMVG(K,L,h) = \sum_{\text{k=h bis h}_{gesamt}\text{-2}} MVG(K,L,k)$$

(Die Vektorenanzahl auf den gröberen Ebenen bei der Bestimmung von *mv-vertikal)*

$$GMVG(K,L) = \sum_{\text{k=0 bis h}_{gesamt}\text{-1}} MVG(K,L,k)$$

(Wird gebraucht für das Schreiben des Koordinatenbuffers auf allen Ebenen)

$$nEMGG(K,L,h) = \sum_{\text{k=0 bis h-1}} MGG(K,L,k)$$

(Die Vektorenanzahl auf den feineren Ebenen bei der Bestimmung von *mv-vertikal)*

7.2 Die Balanceraufgaben und ihre Zeitabschätzungen

Hier werden nur die Zeiten, die zur Migration notwendig sind, erfasst. Zusätzliche Kosten des Balancings wegen der PLB-Arbeiten beim Balancing sind von der ausgewählten Methode abhängig, je nachdem, welche Variante von PLB genommen wird. Diese

werden hier nicht berücksichtigt.

Die gemessenen Zeiten beziehen sich aufs Packen, Kommunizieren und Entpacken. Dabei wird unterschieden zwischen den Laufzeiten für die Informationen dieser 3 Größen ohne die Werte, sowie die Zeiten für die Übertragung der Werte auf den Gitterpunkten.

Die einzelnen Zeitdauern sind in Abschnitt 7.2.2 bis 7.2.7 angegeben. In Abschnitt 7.3 werden die 6 berechneten Terme verwendet, um für gewisse Szenarien die Gesamtlaufzeit pro Kern zu berechnen.

Noch eine Bemerkung zu den in Abschnitt 7.2.2 bis 7.2.7 angegebenen Zeiten: Für alle Zeitkosten bis auf die Kommunikationen werden direkte Zeitangaben gemacht. Bei der Kommunikation geht es nicht direkt um Zeiten, sondern um eine Abschätzung der Länge des Buffers, weil die Kommunikationszeit genau davon abhängig ist. Folgende Definition wird für diesen Fall gebraucht:

Definition 7.4: Die Latenzzeit t_{Latenz} einer Kommunikation ist die Zeit, die benötigt wird, um die Kommunikation ohne Last durchzuführen. Die Bandbreite gibt an, wie viel Zeit es kostet, eine Speicherzelle (z.B. ein Byte) zu übertragen, ohne die Latenzzeit zu berücksichtigen. Die gesamten Zeitkosten der Kommunikation betragen dann:
$t_{gesamt} = t_{Latenz}$ + Bandbreite * Bufferlänge

7.2.1 Der Aufwand der 6 zu untersuchenden Migrationsaktionen

Im Folgenden werden die 6 Größen, die die Laufzeit ausmachen, aufgelistet.

1. Packen Info
2. Packen Werte
3. Kommunikation Info
4. Kommunikation Werte
5. Entpacken Info
6. Entpacken Werte

7.2.2 Die einzelnen Kosten vom Packen der Informationen

Tabelle 7.1: Die Zeit-Kosten für das Packen der Informationen

Punkteauswahl und -entfernung aus der Klasse *Strategie*	O(MGG(K,L,h))
bestimmen von *mv-vertikal*	O(hEMVG(K,L,h) + nEMGG(K,L,h))
Punkte aus *Gitter* entfernen	O(GMGG(K,L))

Punkte aus *ExtremeVektoren* entfernen	O(eMMGG(K,L,h))
Vektoren in den Buffer schreiben	O(GMVG(K,L))
Farbenfeld umlaufen und in die Buffer schreiben	O(GMGG(K,L)) (Gebiete füllen) O(GARG(K,L)) (Umlaufen)
Werte für *feld-vergangen* in den Buffer eintragen	O(GMGG(K,L))
Zwischenkorrektur	entfällt, da sie nebenläufig erfolgt

Die Gesamtkosten sind die Summe der eben angegebenen Einzelkosten, bezeichnet mit 'packen-Info(K,L,h)'.
packen-Info(K,L,h) = O(MGG(K,L,h) + hEMVG(K,L,h) + nEMGG(K,L,h) + GMGG(K,L) + eMMGG(K,L,h) + GMVG(K,L) + GARG(K,L)).

7.2.3 Die Einzelkosten vom Packen der Werte

Das Packen der Werte benötigt an Zeit O(#Variablen GMGG(K,L)). Diese Zeit wird bezeichnet mit 'packen-Werte(K,L)'.

7.2.4 Die Einzelkosten für die Kommunikation der Informationen

Diese Kosten hängen von der Länge des Buffers ab.

Tabelle 7.2: Der Speicherplatzbedarf bei der Kommunikation

Schreiben der Vektoren in die Buffer	O(GMVG(K,L))
- umlaufen des übertragenen Gebietes - Farbfeldwerte in den Buffer kopieren	O(GARG(K,L))
Werte für *feld-vergangen* eintragen	O(GMGG(K,L))

Die gesamten Kommunikationszeitkosten betragen:
Kommunikation-Info(K,L) = t_{Latenz} + Bandbreite * O(GMVG(K,L) + GARG(K,L) + GMGG(K,L))

7.2.5 Die Einzelkosten der Kommunikation der Werte

Es gilt für die Zeitkosten der Kommunikation der Werte:
Kommunikation-Werte(K,L) = t_{Latenz} + Bandbreite * O(#Variablen GMGG(K,L))

7.2.6 Die Einzelkosten des Entpackens der Informationen

Folgende Tabelle gibt die Zeitkosten des Entpackens der Informationen an:

Tabelle 7.3: Zeitkosten des Entpackens der Informationen

Vektorkoordinaten aus Buffer lesen und in *mv-vertikal* schreiben	O(GMVG(K,L))
einfügen der Punkte in die Klasse *Strategie*	O(EMGG(K,L))
prüfen auf Hinzufügen zu *ev2*	O(EMGG(K,L))
Punkte in das *Gitter* einfügen	O(GMGG(K,L))
ändern von *Selbstnachbarn*, *Feldliste*, *dynamische Randlisten* und dem *Farbenfeld*	O(GMGG(K,L))
umlaufen der Datenstruktur *Farbenfeld* nebst schreiben aus den Buffern	O(GARG(K,L))
Werte für *feld-vergangen* und Merkfeld für neue Gitterpunkte	O(GMGG(K,L))

Die Gesamtkosten sind die Summe der eben angegebenen Einzelkosten, bezeichnet mit entpacken-Info(K,L):
entpacken-Info(K,L) = O(GMVG(K,L) + EMGG(K,L) + GMGG(K,L) + GARG(K,L)).

7.2.7 Die Einzelkosten des Entpackens der Werte

Die Zeitkosten für das Entpacken der Werte betragen O(#Variablen GMGG(K,L)). Sie werden bezeichnet mit entpacken-Werte(K,L).

7.3 Die Formeln für die Gesamtlaufzeit

7.3.1 Zu der Komplexität der Analyse

Der allgemeine Fall - bei beliebiger Lastentwicklung - ist schwierig zu analysieren, was an drei Tatsachen liegt, die wie folgt benannt werden:

1. Viele Kommunikations-Szenarien
2. Die Nebenläufigkeit der Migrationsaufgaben
3. Die Synchronisation am Ende der Migration

Im Folgenden analysieren wir nur den einfachen Lastausgleich. Der reverse Lastausgleich ist noch wesentlich schwieriger zu analysieren, weil es mehr Kommunikations-Szenarien gibt, sowie auch mehr Möglichkeiten bei der Nebenläufigkeit von Migrationsaufgaben. Allerdings wird im Allgemeinen beim reversen Lastausgleich die Migration für eine Ebene höher durch die Migration in Bezug auf die untere Ebene dominiert.

Kommunikations-Szenarien

Gegeben sei eine Ost-West Migrationssituation. Es seien 5 in Ost-West-Richtung aufeinanderfolgende Kerne gegeben:

Zuerst wird ein Szenarium angegeben, bei dem nur die Nachbarschaften festgelegt wer-

den, aber ohne Angabe, in welche Richtung geschickt wird: P - Q - R - S - T

Es gibt folgende Beobachtung: Wenn die Zeitbelastung des mittleren Kerns R bestimmt werden soll, dann müssen die Kerne T und P grundsätzlich einbezogen werden, weil die Migrationen mit ihnen die Kerne Q und S in Anspruch nehmen können, bevor sie mit R kommunizieren. Durch die Festlegung der Richtung der Migrationen kommt man auf $2^4 = 16$ Fälle. Allerdings geht es auch noch darum, ob S erst mit T oder R kommuniziert. Dasselbe gilt für Q in Bezug auf die Kerne P und R, sodass man insgesamt 128 Fälle hat, die hier nicht alle analysiert werden können.

Im Folgenden wird das Best-Case Szenarium

$S1 : P \rightarrow Q \rightarrow R \rightarrow S \rightarrow T$

sowie das Worst-Case Szenarium

$S2 : P \leftarrow Q \rightarrow R \leftarrow S \rightarrow T$

erläutert, wobei im Fall von S2 S erst mit T und dann mit R kommuniziert, und Q zuerst mit P und dann mit R, d.h. es liegt hier wirklich der Worst-Case vor.
Als Einleitung in die Analysen wird der 2-Kern-Fall besprochen. Dabei meint man 2 Kerne in Ost-West Richtung und nicht insgesamt 2 Kerne.

Nebenläufigkeit

Die obigen Migrationsaktionen von Abschnitt 7.2 laufen nicht strikt nacheinander, sondern auch nebeneinander ab. Die Gesamtzeit ist nicht die Summe aller 6 Einzellaufzeiten, sondern sie ist niedriger, weil die Migrationsaktionen zum Teil nebeneinander ablaufen. Beispielsweise kommt es vor, dass die Werte kommuniziert werden, während die Informationen schon entpackt werden. Das macht das Programm schneller, aber die Analyse komplizierter. In Abschnitt 7.3.2 wird herausgearbeitet, wie man die Nebenläufigkeit mit Hilfe von Formeln erfassen kann.

Synchronisation

Es kommt bei der Zwischenkorrektur, aber auch zum Abschluss des Balancing, beim sich anschließenden Rechnen zur Synchronisation, die alle Kerne einschließt. Mithilfe einer Formel dargestellt, entspricht dieses einer Maximumsbildung über die Formeln für jeden Kern. Da die einzelnen Kerne allerdings verschiedenen Kommunikations-Szenarien unterliegen, müssen bei der Gesamtformel verschiedene Einzelformeln kombiniert werden.
Hier werden nur die obigen 3 Szenarien für jeweils einen Kern analysiert.

Grundsätzliche Annahmen

Immer wird zuerst die Information gepackt und danach die Werte. Also geschieht auch zuerst die Kommunikation der Info und später die der Werte. Es muss auch zuerst die

Info entpackt werden und erst dann die Werte, denn man braucht die Koordinaten, um die Werte zu entpacken.

Eine grundsätzliche Tatsache

Im Prinzip läuft eine Migration so ab: Erst wird gepackt, dann kommuniziert, und dann entpackt. Damit folgt, dass der Sender stets vor dem Empfänger fertig wird in Bezug auf die jeweilige Migrationsaktion. Der Sender kann natürlich noch andere Migrationsaufgaben haben.

7.3.2 Die allgemeinen Analysen

Bevor wir zu den eigentlichen Analysen gehen können, muss ein Modell für die Kommunikation entwickelt werden, denn sie ist wesentlicher Bestandteil der Analysen.

Das Kommunikationsmodell

Ein Kommunikationsmodell muss 2 Eigenschaften erfüllen:

1. Wenn eine große Kommunikation in zwei kleinere aufgeteilt wird, dürfen diese nicht schneller ablaufen. Nimmt man nur einen Zeitwert für eine Kommunikation an, würde man, wenn man 2 Kommunikationen gleichzeitig losschickt annehmen, dass beide vollendet sind nach Ende der längeren Kommunikation, d.h. man würde einen Maximalwert annehmen. De Facto muss man aber die Summe der Einzellaufzeiten annehmen.

2. Bei der Kommunikation einer längeren Nachricht wird die gesamte Nachricht nicht zu einem Zeitpunkt losgeschickt, sondern nach und nach, und sie wird auch nicht zu einem Zeitpunkt empfangen, sondern die einzelnen Bestandteile gehen nach und nach ein.

Folgendes Kommunikationsmodell erfüllt beide Eigenschaften: Bei einer Kommunikation von einem Kern K zu einem Kern L hat man 2 Zeiten: Man hat die Zeit vom Verschicken der ersten Information bis zur letzten Information, die mit 'a-Kommunikationszeit', bzw. $t_{Kommunikation,a}$ oder $t_{K,a}$ bezeichnet wird. Und dann die Übertragungszeit $t_{ü}$, vom Abschicken der ersten Information bis zum Ankommen dieser Information auf dem empfangenden Kern. Man nimmt an, dass diese Übertragungszeit für jede Informationseinheit gleich ist.
Indirekt wird damit angenommen, dass das Packen aus dem Sende-Buffer in die Kommunikationshardware pro Speicherzelle in etwa dieselbe Zeit in Anspruch nimmt, wie das Entpacken dieser Informationen aus der Kommunikationshardware in den Empfangs-Buffer.
Ferner wird angenommen, dass der Kern während der Zeit $t_{K,a}$ keine anderen Aufgaben übernehmen kann. Man könnte hier auch die umgekehrte Situation annehmen, dass dazu gleichzeitig Aufgaben bearbeitet werden. Nur sollen hier nicht beide Fälle gleichzeitig analysiert werden.
Die tatsächliche Situation ist hingegen wahrscheinlich eine Kombination aus beiden Fällen. Einige Aufgaben können parallel erfolgen, andere, vor allem mit Zugriff auf

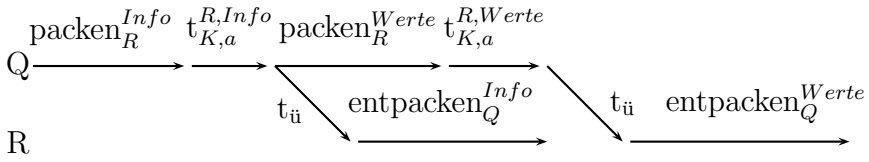


Abbildung 7.1: Das Q → R Kommunikationsszenarium

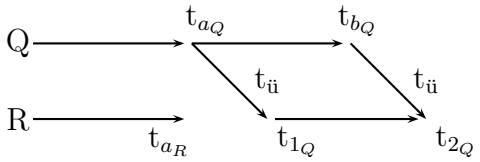


Abbildung 7.2: Das S1 - Szenarium: t_{1_Q} und t_{2_Q} sind die Ankunftszeiten der Nachrichten auf Kern R. Bei t_{1_Q} kommen die Informationen an, bei t_{2_Q} die Werte.

den Arbeitsspeicher, müssen warten, bis in den Buffer geschrieben bzw. er ausgelesen worden ist.

Der 2-Kerne-Fall

Wir haben das Kommunikationsszenarium Q → R, vgl. Abbildung 7.1.

Zuerst werden die einzelnen Größen diskutiert. Auf Kern Q liegen vor: packen_R^{Info} als Packen des Informationsbuffers für den Kern R. packen_R^{Werte} für das Packen des Wertebuffers für den Kern R. $t_{K,a}^{R,Info}$ als $t_{K,a}$-Wert für den Transport des Informationbuffers nach Kern R, $t_{K,a}^{R,Werte}$ ist analog für den Wertebuffer definiert.

Auf Kern R liegt vor: $\text{entpacken}_Q^{Info}$ als Entpacken des Informationsbuffers von Kern Q, und $\text{entpacken}_Q^{Werte}$ analog für den Wertebuffer.
Man kann hier die Gesamtlaufzeit direkt anhand der Abbildung ablesen: Sie beträgt:
$t_{balancing,R} = \text{packen}_R^{Info} + t_{K,a}^{R,Info} + \max\{\text{packen}_R^{Werte} + t_{K,a}^{R,Werte} + t_ü, t_ü + \text{entpacken}_R^{Info}\} + \text{entpacken}_R^{Werte}$
$= \text{packen}_R^{Info} + t_{K,a}^{R,Info} + \max\{\text{packen}_R^{Werte} + t_{K,a}^{R,Werte}, \text{entpacken}_R^{Info}\} + t_ü + \text{entpacken}_R^{Werte}$

Das S1-Szenarium behandeln

Man betrachte die Abbildung 7.2:
Es sind zu den 5 Zeiten aus Abbildung 7.2, die nicht $t_ü$ sind, die Zeitdauern anzugeben:

1. $t_{a_Q} = \text{packen}_R^{Info} + t_{K,a}^{R,Info}$
2. $t_{b_Q} = t_{a_Q} + \text{packen}_R^{Werte} + t_{K,a}^{R,Werte}$
3. $t_{a_R} = \text{packen}_S^{Info} + t_{K,a}^{S,Info} + \text{packen}_S^{Werte} + t_{K,a}^{S,Werte}$
   Denn bis dahin treten keine Wartezeiten auf Kern R auf.
4. $t_{1_Q} = t_{a_Q} + t_ü$

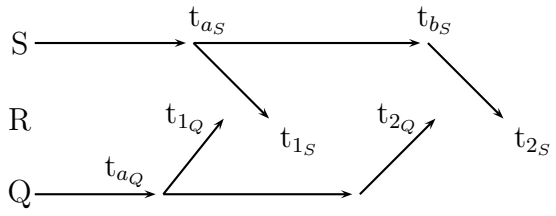


Abbildung 7.3: Das S2 - Szenarium: t_{1_S} und t_{2_S} sind die Ankunftszeiten der Nachrichten von Kern S auf Kern R, t_{1_Q} und t_{2_Q} sind ebenfalls Ankunftszeiten, aber von Kern Q auf Kern R. Die Zeiten $t_{ü}$ seien bei jedem diagonalen Pfeil mit dazu gedacht.

5. $t_{2_Q} = t_{b_Q} + t_{ü}$

Es sind die folgenden 3 Fälle zu unterscheiden:

Fall 1: $t_{a_R} < t_{1_Q} < t_{2_Q}$
$t_{balancing,R} = \max\{t_{1_Q} + \text{entpacken}_R^{Info}, t_{2_Q}\} + \text{entpacken}_R^{Werte}$

Fall 2: $t_{1_Q} < t_{a_R} < t_{2_Q}$
$t_{balancing,R} = \max\{t_{a_R} + \text{entpacken}_R^{Info}, t_{2_Q}\} + \text{entpacken}_R^{Werte}$

Fall 3: $t_{1_Q} < t_{2_Q} < t_{a_R}$
$t_{balancing,R} = t_{a_R} + \text{entpacken}_R^{Info} + \text{entpacken}_R^{Werte}$

Bemerkung: Man kommt auf die einzelnen Terme, indem man sich anschaut, in welcher Reihenfolge die Nachrichten ankommen, und in welcher die einzelnen Migrationsaufgaben abgearbeitet werden.
Im Fall 3 kommt es zu keiner Fallunterscheidung, weil beide Nachrichten schon angekommen sind, wenn Kern R alle Kommunikationsoperationen in Bezug auf Kern S durchgeführt hat. Danach muss R allerdings noch die angekommenen Nachrichten entpacken.
In Fall 2 konkurrieren das Packen von R für S mit dem Werteverschicken von Q an R. Zum Packen in der Zeit t_{a_R} kommt noch das Entpacken der Information von Q hinzu, sodass die Gesamtlaufzeit $\max\{t_{a_R} + \text{entpacken}_R^{Info}, t_{2_Q}\} + \text{entpacken}_R^{Werte}$ folgt. Denn nach der konkurrierenden Nachricht muss noch entpackt werden.
In Fall 1 wird analog diskutiert, nur dass hier das Informationsverschicken von Q an R mit dem Werteverschicken von Q an R konkurriert, weil das Packen von R für S durch das Informationsverschicken dominiert wird. Beim Informationsverschicken kommt noch das Informationsentpacken dazu, und ganz am Ende müssen die Werte noch entpackt werden.

Das S2-Szenarium behandeln

Man betrachte die Abbildung 7.3:
Zur Definition der Zeiten: t_{1_Q}, t_{2_Q}, t_{a_Q} und t_{b_Q} seien wie oben definiert. Und t_{1_S}, t_{2_S}, t_{a_S} und t_{b_S} seien analog definiert, nur für den Kern S, statt des Kerns Q.
Ohne Einschränkung sei $t_{1_Q} < t_{1_S}$, d.h. bzgl. der Informationsnachrichten kommt die

von Kern Q eher an als die von Kern S. Damit haben wir auch hier 3 Fälle:

Fall 1: $t_{1_Q} < t_{2_Q} < t_{1_S} < t_{2_S}$
$t_{balancing,R} = \text{entpacken}_S^{Werte} + \max\{t_{2_S}, \text{entpacken}_S^{Info} + \max\{t_{1_S}, \text{entpacken}_Q^{Werte} + \max\{t_{2_Q}, t_{1_Q} + \text{entpacken}_Q^{Info}\}\}\}$

Fall 2: $t_{1_Q} < t_{1_S} < t_{2_Q} < t_{2_S}$
$t_{balancing,R} = \text{entpacken}_S^{Werte} + \max\{t_{2_S}, \text{entpacken}_Q^{Werte} + \max\{t_{2_Q}, \text{entpacken}_S^{Info} + \max\{t_{1_S}, t_{1_Q} + \text{entpacken}_Q^{Info}\}\}\}$

Fall 3: $t_{1_Q} < t_{1_S} < t_{2_S} < t_{2_Q}$
$t_{balancing,R} = \text{entpacken}_Q^{Werte} + \max\{t_{2_Q}, \text{entpacken}_S^{Werte} + \max\{t_{2_S}, \text{entpacken}_S^{Info} + \max\{t_{1_S}, t_{1_Q} + \text{entpacken}_Q^{Info}\}\}\}$

Die 3 Fälle kommen ganz analog zustande. Man hat dieselbe Struktur $t_{balancing,R} =$ entpacken(letzte-Zeit)+max{$t_{letzte-Zeit}$,entpacken(vorletzte-Zeit),max{$t_{vorletzte-Zeit} \cdots$ }}.
Erläutert wird deshalb die Formel für Fall 1:
Schritt 1: Zuerst werden die Zeitdauern t_{1_Q} und t_{2_Q} in Beziehung gesetzt, weil die dazu gehörenden Ereignisse als erstes einsetzen. Da beim Ereignis zu t_{1_Q} noch das Entpacken der Information von Kern Q hinzukommt, dauert das längere beider Ereignisse $t_{1_A} = \max\{t_{2_Q}, t_{1_Q} + \text{entpacken}_Q^{Info}\}$. Dieses Ereignis wird nun mit dem Ereignis zur Zeit t_{1_S} in Beziehung gesetzt, wobei zum Ereignis bzgl. t_{1_A} noch das Werteentpacken von Kern Q dazukommt. Damit folgt für dieses Ereignis $t_{2_A} = \max\{t_{1_S}, t_{1_A} + \text{entpacken}_Q^{Werte}\}$. Nun wird dieses Ereignis in Beziehung gesetzt zum Eintreffen der Werte von Kern S. Dafür muss zu t_{2_A} noch die Zeit für das Entpacken der Information von Kern S addiert werden. Als Ergebnis hat man: $t_{3_A} = \max\{t_{2_S}, t_{2_A} + \text{entpacken}_S^{Info}\}$. Wenn man hier noch das abschließende Entpacken der Werte von Kern S hinzufügt, und die Formeln ineinander einsetzt, kommt man auf die oben angegebene Formel.

Bemerkung: Zu t_{2_A}: Es könnte auch sein, dass $t_{1_Q} + \text{entpacken}_Q^{Info} > t_{1_S}$ ist. Dann gibt es 2 Möglichkeiten des wartefreien Entpackens: Entweder wird nach $\text{entpacken}_Q^{Info}$ $\text{entpacken}_Q^{Werte}$durchgeführt, was zur Formel führt. Oder es kann $\text{entpacken}_S^{Info}$vor $\text{entpacken}_Q^{Werte}$durchgeführt werden, was zu einer anderen Formel führt. Welche der beiden Formeln gültig ist liegt dann an der Implementierung.

7.4 Formeln in einem Spezialfall

7.4.1 Der Spezialfall wird für *ev* und *ev2* analysiert

Der Spezialfall besteht darin, ein rechteckiges Verfeinerungsgebiet von links nach rechts zu bewegen, vgl. Abbildung 7.4. Wenn man *ev* zu Grunde legt, dann ändert sich das Gebiet, unter der es keine Verfeinerung gibt, nicht. Die groben Gitter ändern sich nur dort, wo sich das Verfeinerungsgebiet jeweils befindet. Dieser Fall wird in diesem Abschnitt 7.4 behandelt. In Abschnitt 7.5 wird *ev2* zu Grunde gelegt, womit sich, um die Last auszugleichen, das grobe Gitter weit über das Verfeinerungsgebiet hinaus verändert. Das kann nicht vollständig analysiert werden. Soweit es geht, wird es behandelt. Und es wird diskutiert, worin die Schwierigkeiten bestehen.

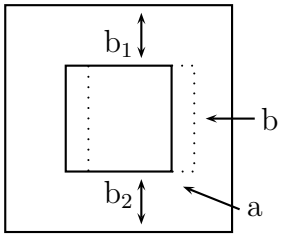


Abbildung 7.4: Ein Rechteck bewegt sich nach rechts

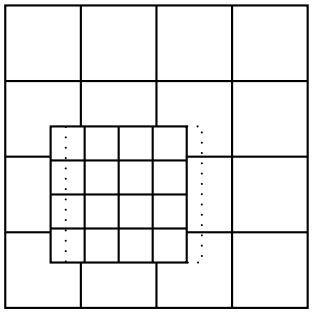


Abbildung 7.5: Ein Rechteck bewegt sich nach rechts

7.4.2 Die Formulierung des Spezialfalls

Es wird ein rechteckiges Verfeinerungsgebiet von links nach rechts bewegt, ohne zusätzliche Bewegungen auf den groben Gittern, vgl. Abschnitt 7.4.1 sowie Abbildung 7.4. Es bewegt sich um die Strecke a nach rechts und hat die Höhe b. Die Anzahl an Punkten darüber beträgt b_1, darunter beträgt sie b_2. Das Gebiet habe auf allen Verfeinerungsebenen dieselbe Größe.
Der Lastausgleich erfolgt in zwei Schritten

1. In Abbildung 7.5 sieht man, dass die Aufteilung des groben Gitters anders ist als die des feinen Gitters. Durch den Nord-Süd-Lastausgleich wird das neu entstandene Gebiet gleichmäßig unter den Kernen aufgeteilt, die das vom Verfeinerungsgebiet rechts liegende Gebiet haben, vgl. Abbildung 7.5.
   In dieser Abbildung ist nach dem Nord-Süd-Lastausgleich das neu entstandene Gebiet gleichmäßig auf die vier im Osten liegenden Kerne aufgeteilt.
   Bemerkung: Eine Grobgitterrestrukturierung für das entfernte Gebiet entsprechend der Anfangs-Aufteilung des groben Gitters geschieht dabei nicht. Damit ist das Gebiet links in Abbildung 7.5 gemeint, welches das neue Verfeinerungsgitter 'zurücklässt'. Dort geschieht keine weitere Lastumverteilung.

2. Durch den sich anschließenden Ost-West Lastausgleich werden Streifen der Breite a von den Kernen, deren Gebiete rechts auf dem Verfeinerungsgitter liegen, zu ihrem linken Nachbarn weitergegeben, vgl. Abbildung 7.9.
   Insgesamt hat man dann eine Verschiebung des Gitters um die Breite a, vgl. Abbildung 7.4 und Abbildung 7.9.

Bevor wir die Lasten für die einzelnen Kerne angeben: Es werden die Werte der Größen aus den Definitionen 7.2 und 7.3 für den Fall bestimmt, dass ein Rechteck von dem Kern K zu dem Kern L mit den Größen c und d, entsprechend der Abbildung 7.6, transportiert wird. Dann brauchen wir nur noch die Größen c und d im speziellen Fall

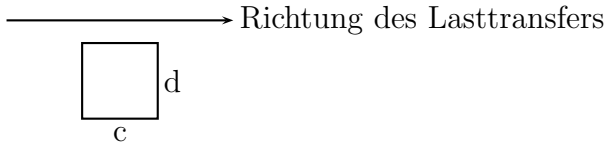


Abbildung 7.6: Die Berechnung der Größen eines transferierten Rechteckes

zu bestimmen, also für den Nord-Süd- und Ost-West-Lastausgleich mit den Daten a, b, b_1 und b_2 nach Abbildung 7.4.

7.4.3 Die Werte der Definitionen 7.2 und 7.3 im Falle des Transports eines Rechteckes

Zuerst wird die Abbildung 7.6 betrachtet:

Dieses Rechteck sei auf dem feinsten Gitter definiert, mit der Breite c in Ost-West-Richtung und der Höhe d in Nord-Süd-Richtung. Damit haben wir approximativ folgende Punktzahlen auf der Ebene h, wobei h=0 die Ebene des feinsten Gitters ist:

1. Das Gebiet hat die Größe c * d * 2^{-2*h}
2. Der Rand hat die Länge 2 * (c+d) * 2^{-h} = (c+d) * 2^{-h+1}
3. Die Anzahl an Vektoren ist c * 2^{-h}.

Natürlich haben wir eigentlich diskrete Zahlen, weshalb diese Formeln nicht exakt sein können. Damit folgt:

Satz 7.2:

1. MGG(K,L,h) ≈ c * d * 2^{-2*h}
2. MARG(K,L,h) ≈ (c+d) * 2^{-h+1}
3. MVG(K,L,h) ≈ c * 2^{-h}

Beweis: Klar nach obigen Formeln für den Transfer eines Gebietes.

Wegen

$$\sum_{i=0}^{k} p^i = \frac{1-p^{k+1}}{1-p}$$

ist

$$\sum_{i=l}^{k} p^i = \frac{p^l-p^{k+1}}{1-p}$$

Daraus folgt:

Satz 7.3: Es gilt:

1. GMGG(K,L) $\approx$ c * d * $\frac{1-0.25^{h_{gesamt}}}{0.75}$
2. GARG(K,L) $\approx$ 2 * (c+d) * $\frac{1-0.5^{h_{gesamt}}}{0.5}$
3. EMGG(K,L) $\approx$ c * d * $\frac{0.25-0.25^{h_{min}+1}}{0.75}$ (für die Ebenen 1 bis h_{min})
4. MMGG(K,L) $\approx$ c * d * $\frac{1-0.25^{h_{min}}}{0.75}$ (für die Ebenen 0 bis h_{min}-1)
5. eMMGG(K,L,h) $\approx$ c * d * ($\frac{1-0.25^{h_{gesamt}}}{0.75}$ - 2^{-2*h})
6. GMVG(K,L) $\approx$ c * $\frac{1-0.5^{h_{gesamt}}}{0.5}$
7. hEMVG(K,L,h) $\approx$ c * $\frac{0.5^{(h+1)}-0.5^{h_{gesamt}}}{0.5}$ (für die Ebenen h bis h_{gesamt}-1)
8. nEMGG(K,L,h) $\approx$ c * $\frac{1-0.5^{h}}{0.5}$ (für die Ebenen 0 bis h - 1)

Beweis: Nach Satz 7.2, den Definitionen 7.3 und den Formeln für die geometrische Reihe folgen die Näherungsformeln.
In den letzten drei Formeln taucht nur der Parameter c auf, weil jeweils c Vektoren übertragen werden.

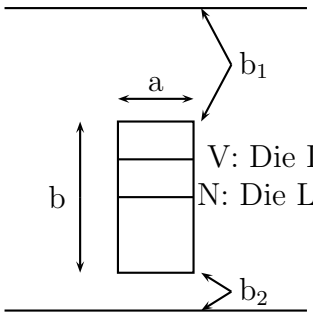
Bemerkung: In der Realität werden keine Rechtecke der Längen c und d vorliegen, sondern abweichende Gebilde aus Punkten. Die Ränder werden dadurch länger, und die Anzahl an übertragenen Vektoren steigt ebenfalls.
Dieses gilt insbesondere dann, wenn der reverse Lastausgleich mit der Struktur *ev2* gewählt wird, vgl. auch Abschnitt 7.5, weil die Gebietsaufteilung dadurch weniger geordnet wird.

7.4.4 Der Nord-Süd-Lastausgleich bei *ev*, also ohne Grobgitterneuaufteilungen

Vorbemerkung 1: Die folgenden Modelle sind vereinfachend, weil in der Realität die Verfeinerungsgebiete keine idealen Rechtecke sind. Das gilt auch für den sich anschließenden Ost-West-Lastausgleich.

Vorbemerkung 2: Die Maße wie a, b, b_1 und b_2 beziehen sich auf das feinste Gitter. In Bezug auf x- und y-Koordinaten gelten dieselben Maße für alle Verfeinerungsgitter.

Wir behandeln diesen Fall für 2 oder 4 Kerne in Nord-Süd-Richtung, mit z.B. gleicher Anzahl an Kernen in beiden Richtungen (# OW-Kerne = # NS-Kerne).
Der allgemeine Fall für beliebig viele Kerne in Nord-Süd-Richtung ist auf diese Weise nicht zu bestimmen, würde aber ähnlich dem Fall mit 4 Kernen in Nord-Süd-Richtung behandelt werden.



V: Die Lastverteilung vor dem Lastausgleich
N: Die Lastverteilung nach dem Lastausgleich

Abbildung 7.7: Der 2 Kerne Nord-Süd-Lastausgleich

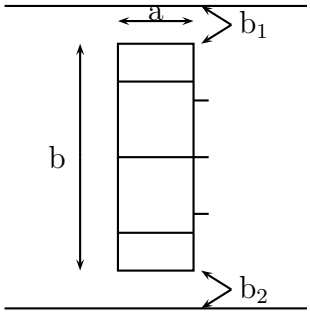


Abbildung 7.8: Der 4 Kerne Nord-Süd-Lastausgleich, mit Angabe der Lastverteilung vorher (Linien im Rechteck) und nachher (Linien rechts neben dem Rechteck).

Der Nord-Süd-Lastausgleich bei 2 Kernen

Wenn man den Streifen rechts in Abbildung 7.5 für zwei Kerne betrachtet, so kommt man auf folgende Abbildung 7.7: Die ursprüngliche Grobgitteraufteilung wird durch die horizontale Linie neben dem V gekennzeichnet, wenn das grobe Gitter durch die horizontale Mittellinie geteilt wird, wie es bei dem Grundgitter bei 2 Kernen in Nord-Süd-Richtung zu Beginn der Fall ist. Die horizontale Linie mit dem N charakterisiert die Gebiete, die nach dem vollständigen Lastausgleich entstehen.
Es wird nun gerechnet. Dabei werden die folgenden Entfernungen in y-Richtung von der oberen Linie des ganzen Gitters an genommen. Der Punkt V ist in Höhe von $\frac{n}{2}$ mit $n = b_1 + b + b_2$. Der Punkt N ist in Höhe von $b_1 + \frac{b}{2}$. Damit ist der Abstand N zu V gleich $| 0.5 * (b_1 + b + b_2) - (b_1 + \frac{b}{2}) | = 0.5 | b_1 - b_2 |$. Es werden die Formeln in Satz 7.3 mit $c = 0.5 | b_1 - b_2 |$ und $d = a$ angewendet.

Der Nord-Süd-Lastausgleich bei 4 Kernen

Wenn man für den Fall, dass das grobe Gitter das Grundgitter ist, die gepunkteten Streifen rechts in Abbildung 7.5 für vier Kerne betrachtet, so kommt man auf die Entfernungen entsprechend Abbildung 7.8. Diese Abbildung bezieht sich auf große Verfeinerungsgebiete, die alle 4 Ost-West-Streifen teilweise bedecken. Hier wird dieser Fall behandelt. Die anderen Fälle folgen entsprechend.

Dabei charakterisieren die Linien in dem Rechteck die Lastverteilung vor dem Nord-Süd-Lastausgleich, und die kurzen Linien rechts neben dem Rechteck die Lastverteilung nach dem Nord-Süd-Lastausgleich.
Nun wird wiederum gerechnet, wobei Kern 1 den Kern mit dem obersten Gebiet dar-

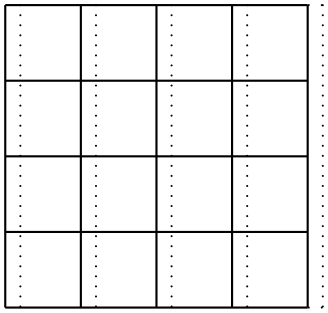


Abbildung 7.9: Der 4 Kerne Ost-West-Lastausgleich. Die gepunkteten Linien charakterisieren die neue Lastaufteilung. Dabei werden links Punkte entfernt, die rechts hinzugefügt werden. Beim Lastausgleich schicken dann die östlichen Nachbarn ihre westlichsten Gitterpunkte an ihren westlichen Nachbarn.

stellt, bis hin zu Kern 4 als Kern mit dem untersten Gebiet.
Kern 2 hat eine Last in Bezug auf eine Höhe von $\frac{1}{4}$ n, und Kern 1 eine Last in Bezug auf eine Höhe von $\frac{1}{4}$ n - b_1. Die Hälfte des Lastunterschiedes wird verschickt, also ein Gebiet, das $\frac{1}{2}$ b_1 hoch und a breit ist. Analog verschickt Kern 3 an Kern 4 ein Gebiet, das $\frac{1}{2}$ b_2 hoch und a breit ist. Damit folgt, dass beim Nord-Süd-Lastausgleich maximal ein Gebiet der Größe c Mal d mit c = $\frac{1}{2}$ max(b_1, b_2) sowie d = a verschickt wird.

Bemerkung 1: Auf den anderen Kernen, d.h. die keine Gebiete im Osten haben, geschieht kein Nord-Süd-Lastausgleich. Das spielt für die Formeln keine Rolle, weil immer der maximale Lastausgleich die gesamte Laufzeit aller Kerne bestimmt.

Bemerkung 2: Hier kann man die verschiedenen Theorien kombinieren. Zu dem durch die Größen c und d festgelegten Gebiet werden die in Satz 7.3 verwendeten Punktanzahlen bestimmt. Mit denen werden die in Abschnitt 7.2 bestimmten Größen fürs Packen, Kommunizieren und Entpacken von Informationen und Werten bestimmt. Mit deren Hilfe können nach Abschnitt 7.3 die Balancing-Zeiten berechnet werden, bis auf den Aspekt, dass nicht alle Szenarien behandelt werden.

7.4.5 Der Ost-West-Lastausgleich

Betrachte die Abbildung 7.9:

Das Rechteck in 7.9 ist vorher und - durch die gestrichelte Linie gekennzeichnet - hinterher gleichmäßig in gleich große Rechtecke aufgeteilt.
Die verschobene Last ist zwischen allen Ost-West-Paaren a breit und $\frac{1}{4}$ b hoch. Mit n_p^{NS} als Anzahl von Kernen in Nord-Süd-Richtung hat man im allgemeinen Fall die verschickten Gebiete mit c = a und d = $\frac{1}{n_p^{NS}}$ b, womit genauso wie in der zweiten Bemerkung zm Nord-Süd-Lastausgleich in Abschnitt 7.4.4 die Balancing-Zeiten berechnet werden könnten, wenn man alle 128 Kommunikation-Szenarien bestimmt hätte. Hier ist d = $\frac{1}{n_p^{NS}}$ b, weil bei der Höhe in Nord-Süd-Richtung durch n_p^{NS} geteilt werden muss.

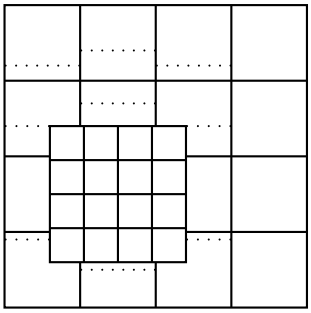


Abbildung 7.10: Wie ein Verfeinerungsrechteck sein grobes Gitter verändert - qualitativ gesehen. Hier wird nur illustriert, wie es sich in der Nord-Süd-Richtung verhält.

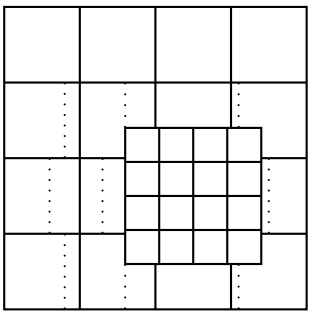


Abbildung 7.11: Wie ein Verfeinerungsrechteck sein grobes Gitter verändert - qualitativ gesehen. Hier wird nur illustriert, wie es sich in der Ost-West-Richtung verhält.

7.5 Der Spezialfall bei der *ev2*-Struktur

Der Spezialfall des sich nach rechts bewegenden Rechtecks wird hier für den Fall des reversen Lastausgleichs untersucht. Insbesondere ändert sich mit der Feingitteraufteilung die Grobgitteraufteilung in von der Verfeinerung nicht betroffenen Gebieten. Es wird der Spezialfall mit 16 Kernen untersucht. Der 4-Kerne-Fall folgt daraus auf einfache Weise.

7.5.1 Illustration, wie sich das Grobgitter bei einem Verfeinerungsrechteck auf 16 Kernen verändert.

Es wird anhand der Abbildungen 7.10 und 7.11 gezeigt, welche Auswirkung ein Verfeinerungsgitter auf die gröberen Gitter hat. Dabei entsteht durch eine Kombination des Ost-West- und Nord-Süd-Lastausgleiches eine Punktverteilung, die sich nicht mehr ohne weiteres in einer Abbildung darstellen lässt. Da beginnt die Grenze der Analysierbarkeit. Doch zuerst wird dieser Spezialfall bei der *ev2*-Struktur soweit möglich analysiert.

7.5.2 Die Bestimmung der maximalen Lastmenge eines einfachen Nord-Süd-Lastausgleichs

Zuerst wird der Fall behandelt, wo das Verfeinerungsgebiet auf anderen Kernen liegt als denen der neu hinzukommenden Punkte. Dann sind in dem neu hinzukommenden Bereich dieselben Grobgitterlinien wie zum Programmstart vorhanden, und die Positi-

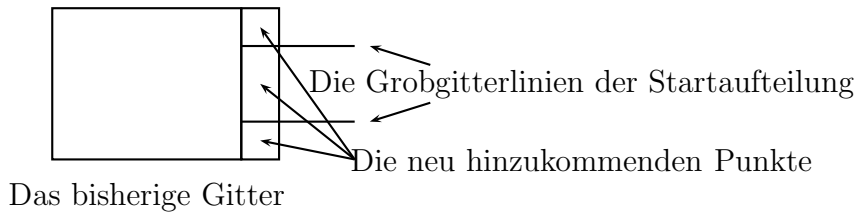


Abbildung 7.12: Die Bestimmung der Zeiten des Lastausgleichs bei der Verwendung von *ev2*, wobei das neue Gebiet auf einem anderen Kern liegt

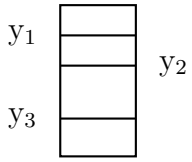


Abbildung 7.13: Es liegt ein vertikal aufgeteiltes Rechteck vor, bevor der NS-Lastausgleich erfolgt.

on der neu hinzukommenden Punkte ist auch bekannt, vgl. Abbildung 7.12. Damit ist die Aufteilung dieser Punkte bekannt, vgl. Abbildung 7.13.

Die Kosten der Umverteilung dieses Gebietes sind mit PLB auszurechnen. Im Falle nur einer benötigten Lastausgleichsrunde geht das mit dem folgenden Satz:

Satz 7.4: Wenn ein Rechteck durch drei Linien wie in Abbildung 7.13 unterteilt ist, dann sind die Kosten des Lastausgleichs für dieses Rechteck in dem Fall, dass er mit **einer** Lastausgleichsrunde erfolgt, direkt zu berechnen. Dabei sei h die Höhe und b die Breite des Gitters.

Beweis: Der vollständige Lastausgleich hat die 3 Linien h/4, h/2 und 3/4 h. Damit betragen die zu transferierenden Lasten im einfachen Fall $|y_1\text{-}h/4|$ b, $|y_2\text{-}h/2|$ b und $|y_3\text{-}3/4\ h|$ b. Die Kosten des Lastausgleichs werden dann durch das Maximum dieser 3 Werte bestimmt.

Wenn das Verfeinerungsgebiet bereits in das Gebiet der Kerne hereinragt, wo die neuen Gitterpunkte dazukommen, sind die Kosten des Lastausgleichs schwieriger zu berechnen. Einmal verschieben sich die Grobgitterlinien entsprechend Satz 7.5. Und dann sind die Lastmengen, für die eine Umverteilung erfolgt, einerseits abhängig von den Grobgitterlinien. Andererseits sind sie davon abhängig, wo und wie weit das Verfeinerungsgitter in das Gebiet der Kerne hereinragt, die die neuen Punkte bereitstellen, vgl. Satz 7.6 zu beiden Punkten.

7.5.3 Der nächste Schritt: Die Änderung der Grobgittersituation im oben beschriebenen Fall

Man betrachte die Abbildung 7.10 und darin die Spalte, wo die rechte Seite des Rechtecks liegt. Dann kann man die Aufteilung des groben Gitters in diesem Fall berechnen.

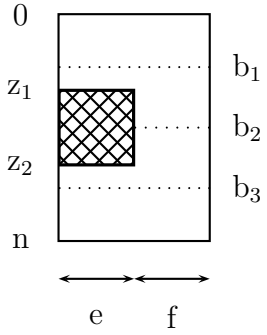


Abbildung 7.14: Ein hineinragendes Verfeinerungsgitter

Satz 7.5:

Gegeben sei die Situation von Abbildung 7.14: Es habe ein Gitter die Höhe n, und es rage ein Verfeinerungsgitter von der Höhe z_1 bis zur Höhe z_2 mit der Breite e hinein, wobei die Gesamtbreite e+f beträgt. Man kann dann b_1, b_2 und b_3 wie in der Abbildung so bestimmen, dass die nicht durch das Verfeinerungsgebiet belegten 4 Flächen denselben Flächeninhalt haben, also einen vollständigen Lastausgleich darstellen.

Beweis: Es wird der Wert der Variablen b_1 berechnet. Die restlichen Variablen werden entsprechend berechnet. Der Flächeninhalt des verbliebenen Gitters beträgt F = n (e+f) - e (z_2 - z1). Die Last, die der oberste Kern nach dem Lastausgleich hat, ist F/4. Folgende Fälle ergeben sich daher:

1. $F/4 \leq z_1$ (e+f).
   Dann ist $b_1 = F/4 \frac{1}{e+f}$. Denn b_1 liegt oberhalb des Verfeinerungsgitters.

2. z_1 (e+f) $\leq F/4 \leq z_1$ (e+f) + (z_2 - z_1) f.
   Es gilt b_1 = (F/4 - z_1 (e+f)) $\frac{1}{f}$. Denn b_1 liegt in Höhe des Verfeinerungsgitters.

3. z_1 (e+f) + (z_2 - z_1) f $\leq F/4$
   Dann ist b_1 = (F/4 - z_1 (e+f) - (z_2-z1) f) $\frac{1}{e+f}$. Denn es gilt: b_1 liegt unterhalb des Verfeinerungsgitters.

Auf entsprechende Weise werden b_2 und b_3 berechnet. qed

7.5.4 Berechnung der Migrationslast

Es kann nun die Migrationslast berechnet werden.

Satz 7.6:
Man kann die Migrationslast berechnen, wenn ein Gebiet entsprechend 7.14 vorliegt.
Beweis: Man hat hier wieder Fälle zu unterscheiden:

1. $b_1 \geq z_2$. Dann liegt die Last auf dem oberen Kern, und es kann nur eine Last von F/2 verschickt werden - in einer Lastausgleichsrunde.

2. $z_1 \leq b_1$ und $b_3 \leq z_2$, und es reicht eine Lastausgleichsrunde zum vollständigen Lastausgleich aus. Dann wende man Satz 7.4 an mit $y_1 = \frac{b_1 - z_1}{z_2 - z_1}$, $y_2 = \frac{b_2 - z_1}{z_2 - z_1}$ und $y_3 = \frac{b_3 - z_1}{z_2 - z_1}$.

3. $b_1 \leq z_1 \leq b_2 \leq z_2 \leq b_3$. Es wird der Fall behandelt, dass eine Lastausgleichsrunde zum vollständigen Lastausgleich ausreicht. Hierbei ist das Verfeinerungsgitter zweigeteilt, und es kommt zur Anwendung von Satz 7.4 mit $y_1 = 0$, $y_2 = h \frac{b_2 - z_1}{z_2 - z_1}$ und $y_3 = h$. Dabei ist h wie in Satz 7.4 gewählt, also $h = z_2 - z_1$, und damit $y_2 = b_2 - z_1$ und $y_3 = z_2 - z_1$.

Die anderen Fälle werden auch per Fallunterscheidung behandelt. Wobei diese nicht trivial sind, wenn die Last nicht in einer Runde ausgleichbar ist.

Es ist mit der Formel aus Abschnitt 6.2.5 zur Berechnung der Anzahl der Lastausgleichsrunden zu überlegen, wann in Abhängigkeit von y_1, y_2 und y_3 eine Lastausgleichsrunde zum vollständigen Lastausgleich ausreicht. Es ist ferner auch Satz 7.4 zu erweitern auf die Fälle mit mehreren Lastausgleichsrunden.

7.5.5 Verschieden breite Streifen

Bei den bisherigen Überlegungen muss man davon ausgehen, dass die Breiten '(e+f)' vorgegeben sind. De facto ist es aber so, vgl. Abbildung 7.11, dass das grobe Gitter nicht in Streifen verschiedener Breiten zerfällt, sondern dass auch Punkte verschiedener Kerne in dem Verfeinerungsgitter übereinander liegen. (Damit ist nicht das Übereinander liegen von Punkten zweier Gitterebenen gemeint.) Wenn man aber eine Aufteilung in Nord-Süd-Streifen annimmt, kann man die Breite dieser Streifen mit folgender Überlegung bestimmen: Wenn man das Verfeinerungsgitter gedanklich an den unteren Rand verschiebt, und dann den Satz 7.5 mit einem um 90 Grad gedrehten Gitter anwendet, bekommt man mit b_1 bis b_3 eine Gitteraufteilung, welche der Lastsituation entspricht. Man stelle sich dafür die Abbildung 7.14 um 90 Grad gedreht vor.

Fazit: Wenn man den reversen Lastausgleich behandelt, wird die Punktverteilung schon bei einem Verfeinerungsrechteck kompliziert, und es sind vereinfachende Annahmen nötig.

Kapitel 8

Test-Läufe

8.1 Schlussfolgerungen zu Beobachtungen

8.1.1 Sehr dynamische Verfeinerungsgebiete

Es geht hierbei um die folgenden zwei Beobachtungen:

1. Wenn das Mehrgitterverfahren mit der Poisson-Gleichung läuft sowie dem Verfeinerungskriterium '$I_h^H L_h u_h$ - $L_H I_h^H u_h$', dann kommt es von Iteration zu Iteration zu größeren neuen Gebieten, und alte größere Gebiete fallen weg. Man hat keine flüssige Bewegung der Verfeinerungsgitter. Statt dessen 'springen' die Verfeinerungsgebiete. (Getestet mit einem 33 x 33 Gitter.)

2. Ändert man das Mehrgitterverfahren nur insoweit, als dass die Funktionswerte f_h der diskretisierten Differentialgleichung $L_h u_h = f_h$ geändert werden von Iteration zu Iteration, so verschwindet dieser Effekt weitgehend, d.h. die Ergebnisse sind dann ähnlich flüssig wie bei dem vom Autor entwickelten Verfeinerungskriterium. Das Verfeinerungsgitter ändert sich nicht so stark in der Zeit.

Das soll jetzt erklärt werden. Zuerst werden die möglichen Gründe für diese zwei verschiedenen Änderungen der Verfeinerungsgitter genannt, und diese in Bezug auf Beobachtung 1 einem Ausschlussprinzip unterzogen, um an die Ursache für die übertriebene Dynamik im ersten Fall heranzukommen.
Mögliche Gründe für die Änderungen beim Diskretisierungsfehler sind:

a. Das normale Rechnen ändert die Gitterwerte.

b. Durch neu entstehende Verfeinerungsbereiche ändern sich die Gitterwerte.

c. Die Änderungen der Funktionswerte f_h bewirken solche Änderungen.

Nun zu Beobachtung 1, d.h. man geht erst einmal von einem festen f_h aus. Nun ist es so, dass bei der Konvergenz von u in den Gitterpunkten sich die Werte auf den Gittern immer weiter approximieren, also '$I_h^H L_h u_h$ - $L_H I_h^H u_h$' (das Mehrgittertheoretiker-Verfeinerungskriterium, vgl. z.B. [43]) stationär werden sollte, weil sowohl $I_h^H L_h u_h$ als auch $L_H I_h^H u_h$ konvergieren. Das Ergebnis dieses Effektes wären stationäre Gebiete, und die 'Sprünge' können damit nicht erklärt werden. Da Grund c bei festem f_h nicht zur Diskussion steht, bleibt Ursache b übrig: Das Bilden von neuen Verfeinerungsgebieten bewirkt diese Dynamik.

Insbesondere die Ränder zu neuen Verfeinerungsgebieten ändern sich. Dort ändern sich dann auch die steuernden Werte des Verfeinerungskriteriums.

Nun zu Beobachtung 2: Warum ist dieser Effekt nicht da, wenn die Funktionswerte f_h während der Iterationen geändert werden? Als Deutung kommt nur infrage, dass die Änderungen von f_h (Grund c) die üblichen Änderungen (Gründe a + b) dominieren, also die Änderungen der Verfeinerungsgebiete vor allem auf den Änderungen von f_h basieren. Bei moderaten Änderungen von f_h kommt es deshalb zu flüssigen Bewegungen.

8.2 Die Messungen der Laufzeiten

8.2.1 Die möglichen Einstellungen für die Messungen

Diese Einstellungen beziehen sich vor allem auf die Festlegung des Mehrgittersystems sowie die Varianten beim Lastausgleich, insbesondere die Konfigurierung von PLB mit den für PLB einstellbaren Parametern. Die parallele adaptive Verfeinerung ist im Wesentlichen bestimmt, d.h. nicht durch Parametereinstellungen beeinflussbar. Eine umfassende Darstellung, die aber für Tests nicht unbedingt gebraucht wird, findet sich in Anhang A.2.

Die Festlegung des Mehrgittersystems

Folgende Parameter definieren das Mehrgittersystem, eignen sich aber nicht zum Tuning, da sie die Aufgabe definieren und nicht deren unterschiedliche Behandlung.

1. Die Parameter, die das Mehrgitter selber bestimmen: Die Gittermaße n und m, und die Anzahl an Verfeinerung- und Vergröberungsebenen h_{min} und h_{max}. An dieser Stelle nennen wir auch die Variable h_{cap}, die angibt, ob eine 'cap' existiert, und wenn ja, auf welcher Gitterebene.

2. Die implementierten Differentialgleichungen:

   (a) Wir nennen hier zuerst die Poisson-Gleichung. Bei der Poisson-Gleichung hat man zusätzlich noch die Variablen μ_1 und $\mu 2$, die angeben, wie viel Mal vor- und nachgeglättet wird. Ferner gibt es bei der Poisson-Gleichung einen Parameter Punktlast oder PL, mit dem man in jeden Gitterpunkt mehr Last legen kann. Genauer: Eine Punktlast von 1 rechnet pro Gitterpunkt und Glättungsschritt einmal mehr den 5-Punkte-Stern des Differentialoperators aus.

   (b) Gemessen wurden die Laufzeiten der Shallow Water Equations, basierend auf den Operatoren von [36] und den Erfahrungen mit der Poisson-Gleichung, wobei dabei nicht die Werte abgebildet wurden, sondern die Laufzeiten. Es wurden Operatoren wie solve-pre und solve-post durch ähnliche Operatoren abgebildet, die dann pro Mehrgitterzyklus und Gitterebene einfach aufgerufen werden, wobei für entsprechend viele Variablen ein Randaustausch durchgeführt wurde.

3. Die Wahl verschiedener Adaptivitätsentscheidungen.

(a) Die selbstadaptive Verfeinerung: Hier wird nur die Poisson-Gleichung behandelt. Gewählt wird das Kriterium '$I_h^H L_h u_h$ - $L_H I_h^H u_h$', vgl. z.B. [43], oder das eigens entwickelte Kriterium, welches inspiriert wurde durch das Kriterium in [36] oder durch Gradienten-basierte Kriterien.
Dabei kann man als Option auch die Funktionswerte in der Zeit ändern. Bei der Selbstadaptivität hat man ferner als Parameter den oder die Grenzwerte beim Verfeinerungskriterium.

(b) Vorgaben verschiedener bewegter Gitter. Diverse Szenarien tauchen hier auf.

i. KOMPLETT: Das ganze Verfeinerungsgitter wird gewählt
ii. EINGITTER: Ein bewegtes Verfeinerungsrechteck wird gewählt. Die Gittergröße wird durch den Parameter 'vgitter' mit den Werten 0 bis 2 ausgewählt.
iii. EINGITTER_UNBEWEGT: Ein stationäres Verfeinerungsrechteck entsteht.
iv. AUFSCHLAG: Ein Verfeinerungsrechteck stößt auf ein stationäres Verfeinerungsrechteck
v. ZUSAMMENSTOß: Zwei Verfeinerungsrechtecke laufen gegeneinander.

4. Es besteht die Option, auf den nicht adaptiven Ebenen ein FAS oder ein normales Mehrgitterschema zu berechnen.

Die Flexibilität beim Lastausgleich - verschiedene Möglichkeiten

1. Zu den PLB-Einstellungen:

(a) Zuerst die Wahl zwischen PLB und der PLB-Variante. Bei letzterem die Unterscheidung nach 'K_n' - oder 'Baumlastbestimmung', sowie nach 'Matrix' und 'Vektoren'. Bei der 'Matrix' hat man noch Steuerungsmöglichkeiten.

(b) Beim Balancer kann man folgendes unterscheiden: Man kann pro Zeitschritt eine Lastausgleichsrunde durchführen, und dann optional am Anfang einen vollständigen Ausgleich durchführen über so viele Runden wie nötig. Die Idee dabei: Am Anfang muss das ganze neue Verfeinerungsgitter verteilt werden. Dagegen müssen in jedem Zeitschritt nur die begrenzten Änderungen zum Ausgleich gebracht werden.
Bemerkung: Bei dem Entstehen entfernter Gebiete kann allerdings auch nicht nur am Anfang eine Entscheidung für mehrere Lastausgleichsrunden sinnvoll sein, wenn der Ausgleich zwischen entfernten Kernen erfolgen soll.

2. Man kann den Balancer ein- oder ausschalten.

3. Weitere Einstellungen für die Migration:

(a) Die Wahl der Strategien *ev* versus *ev2* - entsprechend einem einfachen oder reversen Lastausgleich. Also macht man den Lastausgleich nur auf einer Ebene oder gleichzeitig noch auf der Ebene darüber.

(b) Lastwertekorrektur-Typ (LWK-Typ). Man unterscheidet dabei, ohne Lastwertekorrektur zu arbeiten oder aber mit ihr zu arbeiten. Dabei unterscheidet man, ob Randpunkte nicht gepackt werden, das ist der LWK-Typ 'Vollpunkt', oder ob sie gepackt werden, aber ohne Lastinkrementierung, die mit

dem LWK-Typ 'auch Nullpunkt' bezeichnet wird.
Genaueres hierzu kann in A.2.3 nachgeschlagen werden.

Sonstige Parameter

1. Ein Kriterium ist die Anzahl an Iterationen.

Bemerkung 1: Es gibt diverse Debug-Möglichkeiten, auch unter Verwendung von Dateien (man kann hier z.B. einen Dateinamen mit AD in der Kommandozeile angeben), die aber an den Effizienzen nichts ändern, da sie bei Leistungsläufen stets abgeschaltet sein müssen.

Bemerkung 2: Die Messungen erfolgten auf dem Großrechner CHEOPS des Rechenzentrums der Universität zu Köln, allerdings mit maximal 16 Kernen. Zur Information über diesen Rechner: vgl. [26].

Dieser Rechner ist ausgestattet mit dual socket Knoten von Xeon X5550 Quad-Core- bzw. Xeon X5650 Hexa-Core-Prozessoren mit 2.66 GHz. Hat man besondere Rechte für CHEOPS, kann man daneben Zugriff auf quad socket Knoten von Xeon 7560 2.27 GHz-Octo-Core-Prozessoren mit besonders viel Arbeitsspeicher bekommen.

8.2.2 Die Auswertung der Laufzeiten

Es werden 6 Läufe gemacht, und für die 3 kürzesten Zeiten wird der Durchschnitt bestimmt. Daraus wird dann der Speedup bestimmt.

8.3 Die Messwerte

8.3.1 Die Grundannahmen der Messungen

4 Annahmen werden getroffen:

1. Es werden die (parallelen) Rechenzeiten gemessen. Die Initialisierung der Gitter sowie ihre Finalisierung werden nicht mitgemessen.

2. Die Poisson-Gleichung hat pro Gitterpunkt sehr wenig Rechenlast. Deswegen wurden bei den Glättungen zum Teil zusätzlich dummy-Glättungen vorgenommen. Die Variable Punktlast gibt an, wie viele dummy-Glättungen pro Glättung vorgenommen werden. Bei den dummy-Glättungen kommt es zu keiner Randkommunikation.

3. Es wird mit einem Kern pro Knoten gerechnet. Bei max. 48 GB pro Knoten und einem 10241x10241 Gitter kann nicht mit mehreren Kernen auf einem Knoten gerechnet werden. Bei 5121x5121 Gittern zeigt sich, dass das Rechnen mit einem Kern pro Knoten bessere Laufzeiten ergibt. Das wurde für ein eher großes asymmetrisches Verfeinerungsgitter (vgitter 8) gemessen, vgl. Tabelle 8.10. Das kann man darauf zurückführen, dass die Kerne auf einem Chip sich den Daten- und Adressbus beim Zugriff auf den Arbeitsspeicher teilen müssen. Da 2 sockets auf einem Knoten sind, kann aus Effizienzgründen mit 2 Kernen pro Knoten gerechnet werden.

4. Bei einem parallelen Testlauf wird der Maximalwert der Laufzeiten aller Kerne zu Grunde gelegt.

8.3.2 Die Grundeinstellungen

Gewählte Einstellungen, die den anschließenden Variationen zu Grunde liegen.

Die Festlegung des Mehrgittersystems

1. n = 10241, m = 10241, h_{min}= 1, h_{max}= 2 ($\Rightarrow$ h_{gesamt}= 4), h_{cap}= -1
2. $\mu_1 = 2$, $\mu_2 = 2$
3. Poisson-Gleichung mit Punktlast 10, Simulation mit relativ großem Verfeinerungsgitter (vgitter 8)
4. 10 Iterationen

Die Flexibilität des Lastausgleichs

1. Wahl der PLB-Variante
   Datenbeschaffung: K_n
   Vektor oder Matrix: Vektor
2. Strategie: *ev2*, d.h. der reverse Lastausgleich wird aktiviert
3. Balancer: aktiviert
4. LWK ausgeschaltet

Sonstige Kriterien

1. 10 Iterationen
2. DEBUG = off

8.3.3 Die Laufzeitwerte auf CHEOPS

Im Folgenden sei n = 10241 und m = 10241. Nur in Tabelle 8.10 wird n = 5121 und m = 5121 gewählt.

Mit der folgenden Tabellenstruktur werden die Laufzeiten und Speedups angegeben:

Tabelle 8.1: Der Tabellenaufbau für Messwerte:

1	6 Messwerte für einen Kern	Durchschnitt der 3 besten Werte
4	6 Messwerte für 4 Kerne	Durchschnitt der 3 besten Werte
8	6 Messwerte für 8 Kerne	Durchschnitt der 3 besten Werte
16	6 Messwerte für 16 Kerne	Durchschnitt der 3 besten Werte
Speedup 4 Kerne, Effizienz 4 Kerne		
Speedup 8 Kerne, Effizienz 8 Kerne		
Speedup 16 Kerne, Effizienz 16 Kerne		

Tabelle 8.2: Die Messwerte für ein relativ großes simuliertes Gitter (vgitter 8):

1	728.151, 699.414, 727.483, 695.895, 720.005, 725.654	705.105
2	398.387, 397.787, 397.913, 398.044, 397.81, 398.009	397.837
4	203.298, 202.717, 202.759, 202.098, 203.172, 203.121	202.525
8	119.709, 119.346, 117.14, 119.523, 119.572, 119.552	118.67
8 gL	116.765, 116.24, 116.288, 116.621, 116.283, 116.093	116.205
16	62.7531, 59.8004, 59.6502, 59.4765, 59.5819, 59.6326	59.5637
16 gL	58.6501, 58.3899, 60.2866, 58.5388, 58.4272, 58.6112	58.452
2 Kerne: Speedup 1.77235, Effizienz 0.88617		
4 Kerne: Speedup 3.48157, Effizienz 0.87039		
8 Kerne: Speedup 5.94173, Effizienz 0.74272		
8 Kerne, gL: Speedup 6.06777, Effizienz 0.75847		
16 Kerne: Speedup 11.8378, Effizienz 0.73986		
16 Kerne, gL: Speedup 12.06297, Effizienz 0.75394		

Tabelle 8.3: Die Messwerte für ein großes simuliertes Verfeinerungsgitter (vgitter 2), ohne Punktlast:

1	146.16, 145.313, 145.358, 145.514, 147.394, 145.676	145.395
2	92.0501, 90.5517, 92.2802, 90.3467, 90.1448, 90.0518	90.1811
4	50.8905, 50.1854, 50.1901, 50.1319, 50.4302, 51.2824	50.169
8	24.7479, 24.435, 24.6001, 24.5263, 24.3681, 24.3592	24.387
8 gL	24.8955, 24.2748, 24.266, 24.3337, 24.2897, 24.3285	24.277
16	13.8851, 13.7647, 13.6372, 13.8185, 13.6364, 13.643	13.639
16 gL	13.8714, 13.7396, 13.6868, 13.6912, 13.7302, 13.7184	13.699
2 Kerne: Speedup 1.61226, Effizienz 0.80613		
4 Kerne: Speedup 2.8981, Effizienz 0.72453		
8 Kerne: Speedup 5.96199, Effizienz 0.74525		
8 Kerne, gL: Speedup 5.989, Effizienz 0.74863		
16 Kerne: Speedup 10.66024, Effizienz 0.66626		
16 Kerne, gL: Speedup 10.61355, Effizienz 0.66335		

Tabelle 8.4: Die Messwerte für ein großes simuliertes Verfeinerungsgitter (vgitter 2), Zeitsimulation der Shallow Water Equations:

1	468.558, 472.049, 471.374, 471.81, 471.556, 464.429	468.12
2	262.623, 260.332, 259.767, 261.878, 261.133, 261.555	260.411
4	136.768, 136.633, 136.545, 136.291, 136.319, 136.384	136.331
8	68.5677, 68.418, 68.5181, 68.6231, 68.7957, 68.8333	68.501
8 gL	68.2178, 68.4659, 68.6675, 68.0024, 68.5339, 68.0702	68.097
16	36.0332, 35.5209, 36.0481, 35.9217, 36.0744, 35.9276	35.79
16 gL	35.7847, 36.0361, 36.1381, 36.0839, 35.8481, 35.9993	35.877
2 Kerne: Speedup 1.79762, Effizienz 0.89881		
4 Kerne: Speedup 3.4337, Effizienz 0.85843		
8 Kerne: Speedup 6.83377, Effizienz 0.85422		
8 Kerne, gL: Speedup 6.87431, Effizienz 0.85929		
16 Kerne: Speedup 13.07963, Effizienz 0.81748		
16 Kerne, gL: Speedup 13.04791, Effizienz 0.81549		

Tabelle 8.5: Die Messwerte für ein mittelgroßes selbstadaptives Verfeinerungsgitter, entsprechend dem eigenen Verfeinerungskriterium (verfeinerungstyp 4):

1	535.406, 556.955, 550.987, 557.856, 557.252, 555.188	547.194
2	306.724, 310.971, 309.802, 309.662, 310.143, 310.157	308.729
4	175.589, 174.914, 175.999, 175.067, 175.339, 174.98	174.987
8	107.241, 106.808, 109.147, 106.73, 106.662, 106.891	106.733
8 gL	105.441, 104.672, 105.186, 105.051, 104.983, 105.18	104.902
16	61.5977, 60.6357, 60.7816, 60.8973, 60.7632, 60.6732	60.691
16 gL	53.8268, 53.8717, 53.4097, 53.3478, 53.5542, 53.5833	53.437
2 Kerne: Speedup 1.77241, Effizienz 0.8862		
4 Kerne: Speedup 3.12706, Effizienz 0.78176		
8 Kerne: Speedup 5.12676, Effizienz 0.64084		
8 Kerne, gL: Speedup 5.21624, Effizienz 0.65203		
16 Kerne: Speedup 9.01606, Effizienz 0.5635		
16 Kerne, gL: Speedup 10.23998, Effizienz 0.64		

Tabelle 8.6: Die Messwerte für ein asymetrisches mittelgroßes simuliertes Verfeinerungsgitter (vgitter 1):

1	432.086, 431.459, 431.484, 431.611, 431.083, 431.413	431.318
2	263.324, 263.674, 264.403, 263.776, 263.372, 262.778	263.158
4	130.503, 129.491, 129.527, 129.427, 129.611, 129.816	129.482
8 gL	79.0114, 78.6017, 78.7826, 78.622, 78.5457, 78.6638	78.59
16 gL	43.1591, 42.7136, 42.6769, 42.8106, 42.6282, 42.7168	42.673
2 Kerne: Speedup 1.63901, Effizienz 0.8195		
4 Kerne: Speedup 3.3311, Effizienz 0.83278		

8 Kerne, gL: Speedup 5.4882, Effizienz 0.68603
16 Kerne, gL: Speedup 10.13095, Effizienz 0.63318

Tabelle 8.7: Dieselben Messparameter wie bei der letzten Tabelle, nur mit ausgeschaltetem Balancer:

1	(Werte übernehmen, da bei einem Kern Balancer ausgeschaltet)	431.318
2	322.262, 316.781, 323.07, 321.873, 322.065, 321.663	320.106
4	162.427, 162.829, 162.636, 163.295, 163.074, 162.762	162.608
8 gL	81.038, 81.1385, 81.985, 82.4199, 81.0537, 81.1137	81.068
16 gL	66.6504, 63.6893, 63.2896, 65.3908, 66.2734, 66.3447	64.123
2 Kerne: Speedup 1.34742, Effizienz 0.67371		
4 Kerne: Speedup 2.6525, Effizienz 0.66313		
8 Kerne, gL: Speedup 5.32045, Effizienz 0.66506		
16 Kerne, gL: Speedup 6.72642, Effizienz 0.42040		

Tabelle 8.8: Die Messwerte für ein relativ großes simuliertes Verfeinerungsgitter (vgitter 8), ohne Punktlast:

1	118.268, 120.478, 117.718, 117.503, 117.429, 117.613	117.515
2	76.1025, 75.7429, 76.7604, 75.806, 75.7684, 75.8071	75.772
4	39.4394, 38.6745, 38.784, 38.7181, 38.8468, 38.533	38.642
8 gL	26.3292, 25.9461, 25.9228, 25.9023, 25.8946, 26.285	25.907
16 gL	15.535, 14.3538, 14.298, 14.4894, 14.3114, 14.3476	14.319
2 Kerne: Speedup 1.5509, Effizienz 0.77545		
4 Kerne: Speedup 3.04112, Effizienz 0.76028		
8 Kerne, gL: Speedup 4.53603, Effizienz 0.567		
16 Kerne, gL: Speedup 8.20693, Effizienz 0.51293		

Tabelle 8.9: Die Messwerte für ein relativ großes simuliertes Verfeinerungsgitter (vgitter 8), Zeitsimulation der Shallow Water Equations:

1	379.236, 379.443, 379.347, 379.082, 379.065, 379.408	379.128
2	209.146, 207.946, 208.491, 210.301, 208.337, 209.473	208.258
4	106.774, 106.774, 106.455, 106.594, 106.441, 106.515	106.47
8 gL	63.4862, 63.4357, 63.2879, 63.2679, 63.2886, 63.2424	63.266
16 gL	33.4734, 33.3787, 33.298, 33.2475, 33.2504, 33.3273	33.265
2 Kerne: Speedup 1.82047, Effizienz 0.91024		
4 Kerne: Speedup 3.56089, Effizienz 0.89022		
8 Kerne, gL: Speedup 5.9926, Effizienz 0.74908		
16 Kerne, gL: Speedup 11.3972, Effizienz 0.71233		

Tabelle 8.10: Berechnung eines relativ großen Verfeinerungsgitters (vgitter 8) mit m = 5121 und n = 5121

1 Kern	167.31, 166.299, 166.052, 166.182, 166.265, 166.116
4 Kerne auf 4 Knoten	46.1479, 51.9905, 45.8735, 45.8713, 45.8694, 45.9348
4 Kerne auf 1 Knoten	63.5577, 54.895, 55.4971, 55.0813, 55.5828, 55.1997

Die folgenden Abbildungen stellen Bezierkurven der eben berechneten Effizienzen mit der Wahl von 'gL' dar. Bezierkurven sind nicht völlig wertetreu, haben dafür aber für Effizienzen ein 'natürlicheres Aussehen' als Splines.

Abbildung 8.1: Die Effizienzkurve zu Tabelle 8.2.: Simulation, Punktlast 10, vgitter 8

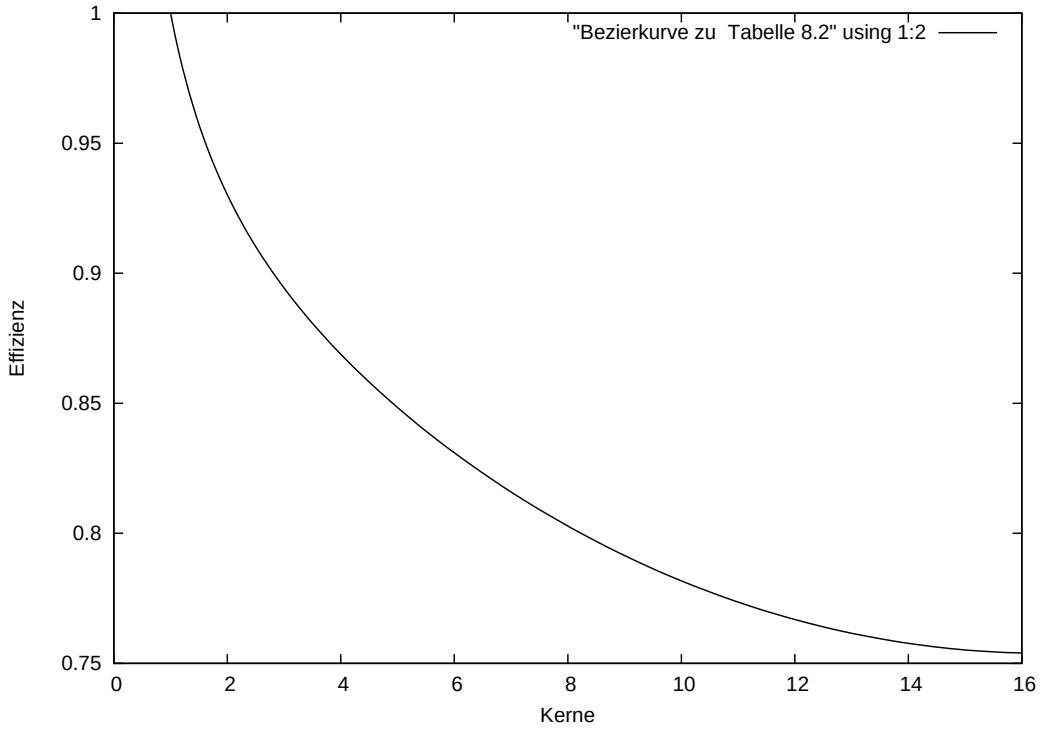


Abbildung 8.2: Die Effizienzkurve zu Tabelle 8.3.: Simulation, Punktlast 0, vgitter 2

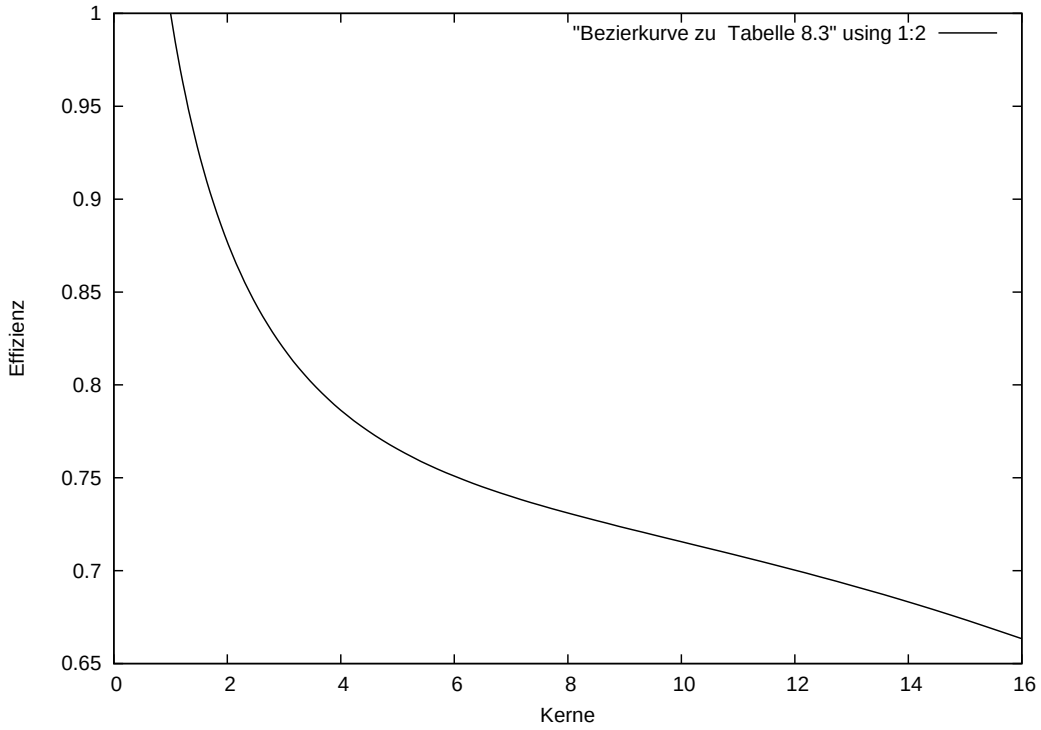


Abbildung 8.3: Die Effizienzkurve zu Tabelle 8.4.: Simulation, SWE, vgitter 2

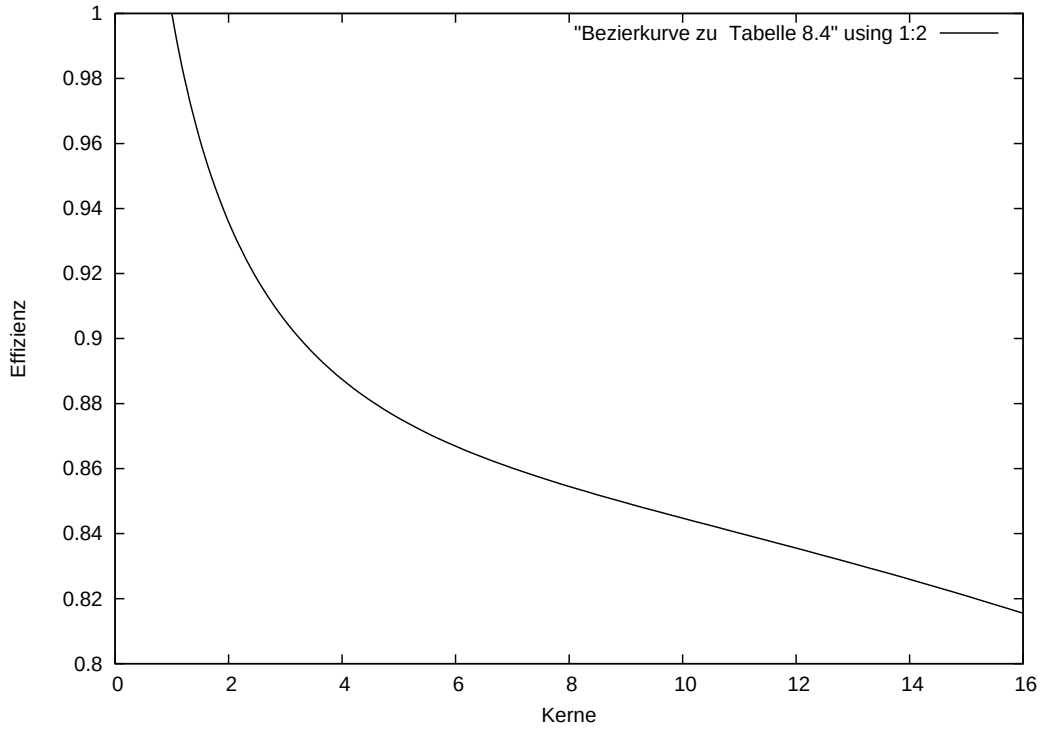


Abbildung 8.4: Die Effizienzkurve zu Tabelle 8.5.: Selbstadaptiv, verfeinerungstyp 4

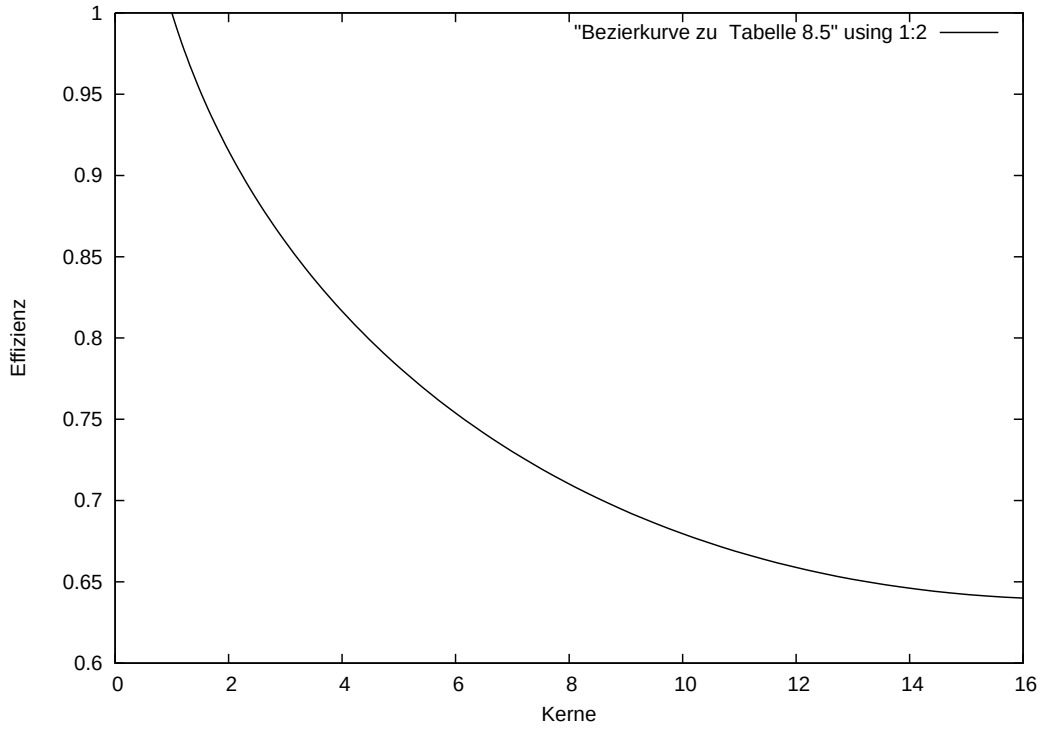


Abbildung 8.5: Die Effizienzkurve zu Tabelle 8.6.: Simulation, vgitter 1, balancing

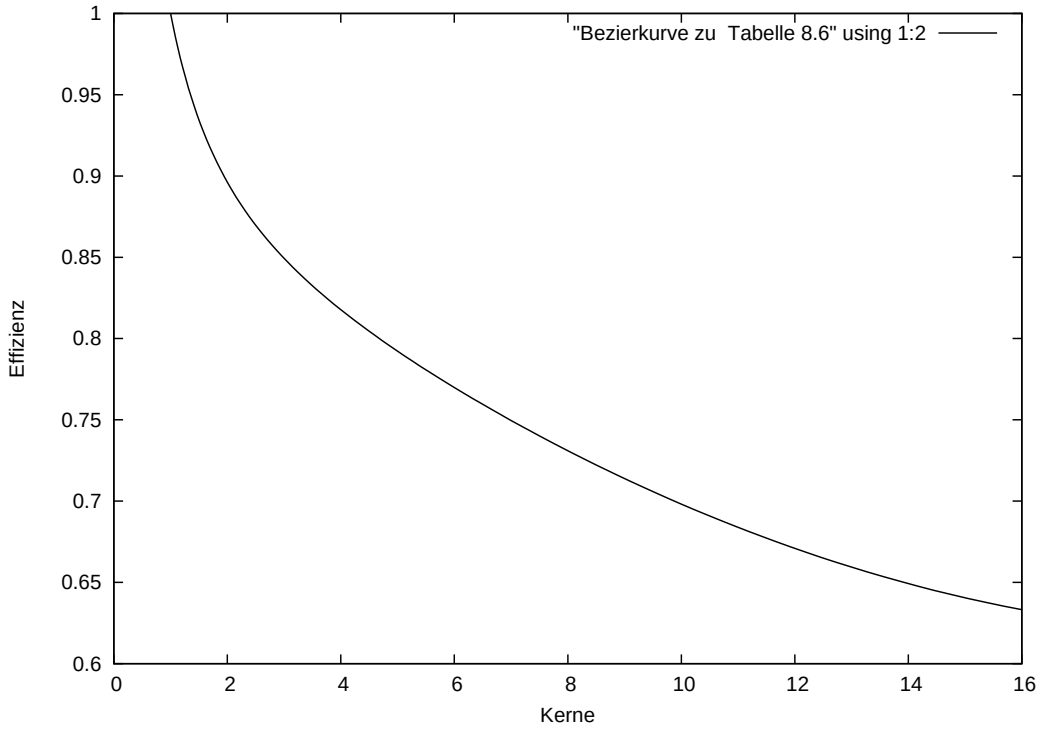


Abbildung 8.6: Die Effizienzkurve zu Tabelle 8.7.: Simulation, vgitter 1, ohne balancing

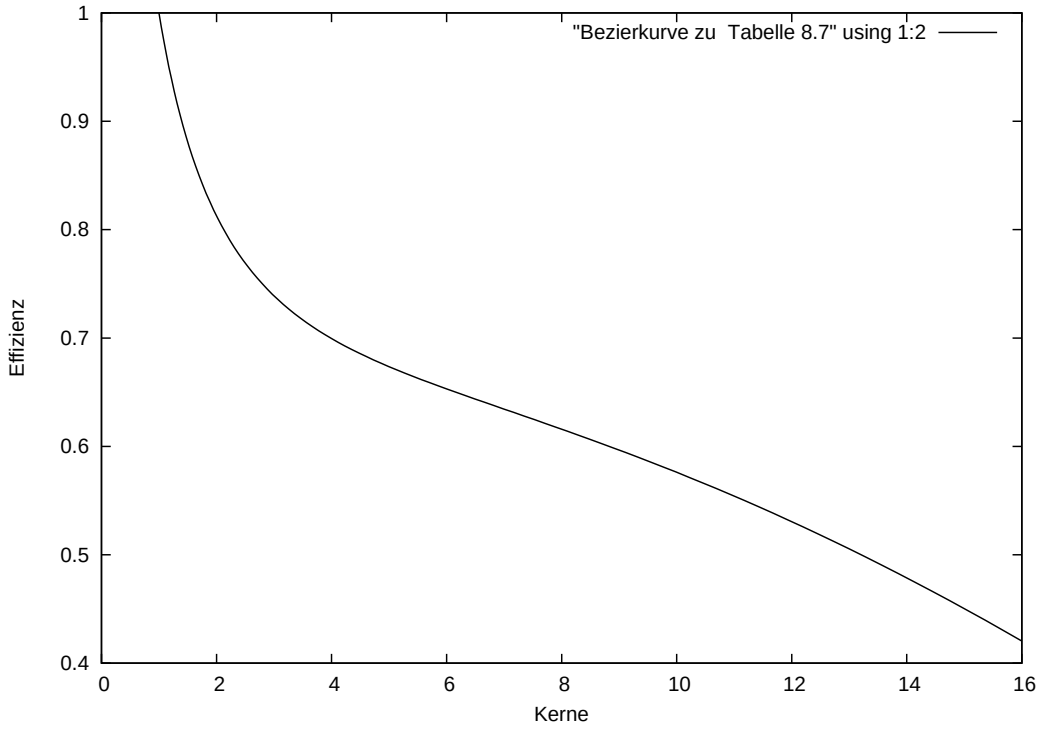


Abbildung 8.7: Die Effizienzkurve zu Tabelle 8.8.: Simulation, Punktlast 0, vgitter 8

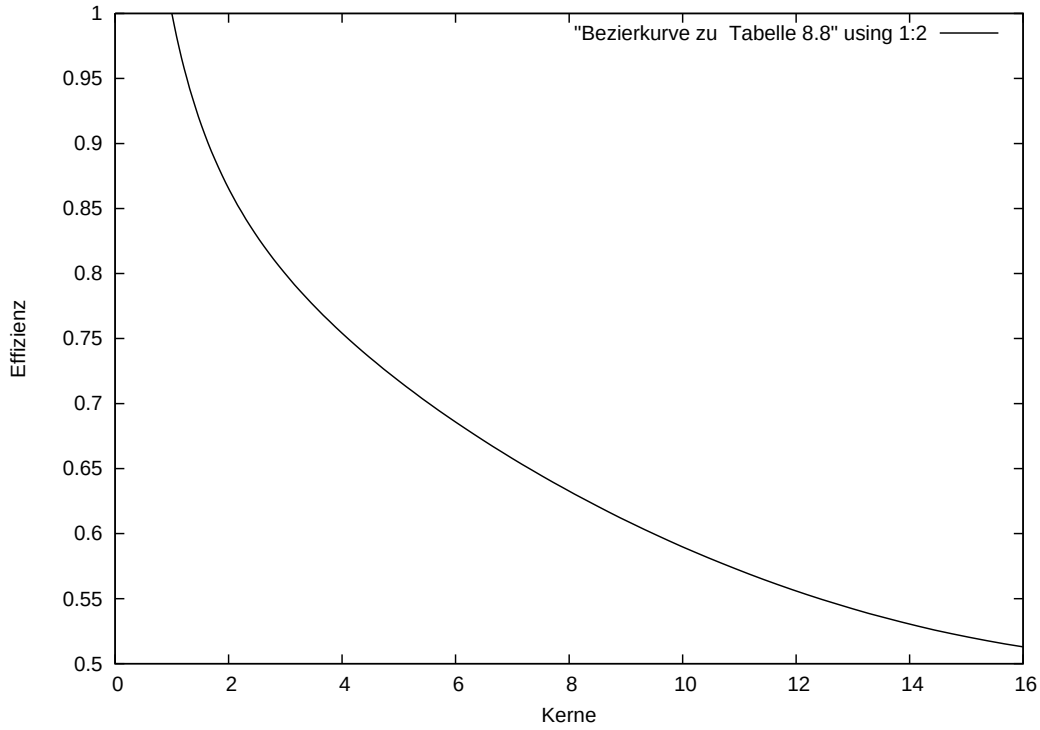
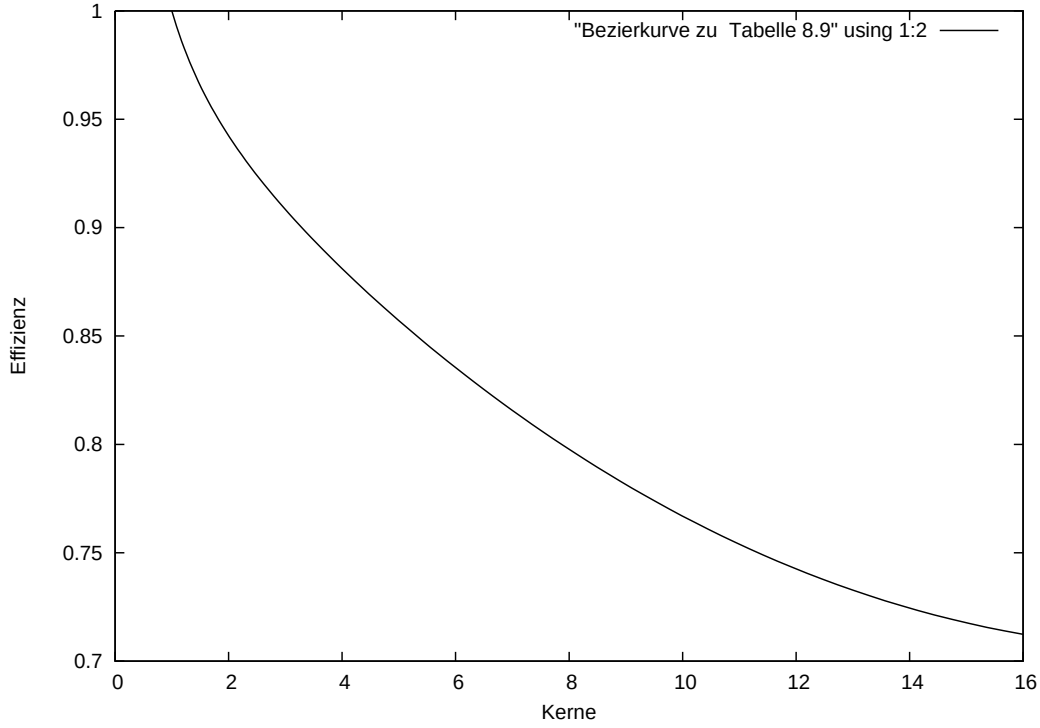


Abbildung 8.8: Die Effizienzkurve zu Tabelle 8.9.: Simulation, SWE, vgitter 8



8.3.4 Abbildung zur Punkteverteilung

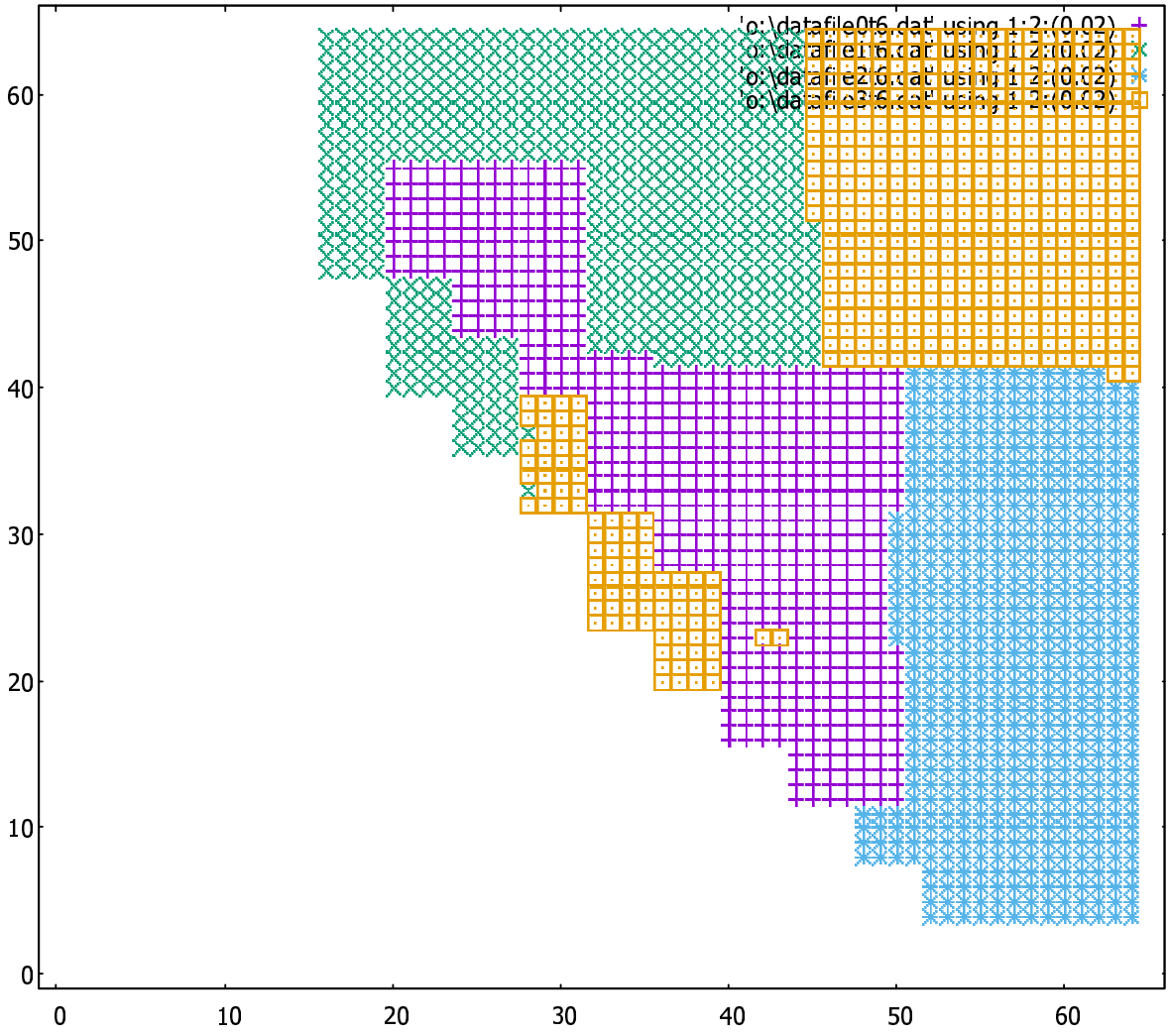


Abbildung 8.9: Die Punkteverteilung auf 4 Kerne - Beispiel

8.3.5 Kommentare zu den Messungen

1. Beobachtungen bei den Tests mit CHEOPS:

   (a) Es gibt auf CHEOPS verschieden leistungsfähige Prozessoren. Allerdings führt die Wahl des Arbeitsspeichers von 44 GB sowie der Aufruf '-C inca12' dazu, dass auf denselben Prozessoren gerechnet wird.

   (b) CHEOPS ist am besten nachts und am Wochenende zu benutzen. Dann ist er am wenigsten belastet. Messungen unter Belastung sind nicht so aussagefähig, wie solche unter keiner Last. Allerdings geschahen alle Messungen mit '–exclusive', damit der jeweilige Knoten exclusiv genutzt wird.

2. Zur Parameterwahl

   (a) Das Ausschalten des Balancers reduziert den Speedup, vgl. die Abbildungen 8.5 und 8.6. Nur bei 8 Knoten sind die Werte vergleichbar. Dafür muss das

Gitter schon initial gut auf die 8 Kerne verteilt worden sein.

(b) Zu den Shallow Water Equations: Die hier gemessenen Effizienzen liegen recht hoch, deutlich über denen mit Punktlast 0. Es wurde das gleiche große Verfeinerungsgitter genommen. Selbstadaptiv kann man sie nicht wählen wegen der fehlenden Werte der Gleichungen. Es wird nur das Laufzeitverhalten simuliert.

(c) Aus dem Vergleich von Abbildung 8.3 (größtes Gitter) und Abbildung 8.1 (mittleres Gitter) und Abbildung 8.5 (kleinstes Gitter) kann man schließen, dass größere Verfeinerungsgitter zu einer höheren Effizienz führen.

(d) Die Effizienzkurven von Abbildung 8.4 und Abbildung 8.5 sind vergleichbar. Es geht also bei den Effizienzen primär um die Größe der Verfeinerungsgitter, und weniger darum, ob simuliert oder selbstadaptiv verfeinert wurde.

(e) Zur Punktlast: Man kann sogar mit Punktlast 0 Ergebnisse erzielen, obschon die Rechenlast pro Gitterpunkt dabei sehr gering ist. Allerdings ist in Abbildung 8.2 (Punktlast 0) das Verfeinerungsgitter entsprechend groß. Eine höhere Punktlast führt zu einer höheren Effizienz.

(f) Die den Abbildungen 8.2 und 8.3 zu Grunde liegenden Messungen wurden mit einem kleineren Verfeinerungsgitter (vgitter 8) durchgeführt. Die resultierenden Kurven werden in den Abbildungen 8.8 und 8.9 wiedergegeben. Die Effizienzen nehmen bei den kleineren Verfeinerungsgittern ab, allerdings bei der Punktlast 0 stärker als bei der SWE-Zeitsimulation. Immerhin werden selbst in diesem Extremfall mit nur geringer Gitterpunktlast noch deutliche Speedups erreicht.

3. Übrige Beobachtung: Es ist eine gute Wahl, 6 Läufe zu machen und die drei besten Laufzeiten auszuwählen und davon den Durchschnittswert zu berechnen. Man nimmt die besten Werte, weil das die Leistung des Algorithmus widerspiegelt. Es zeigt sich aber, dass alle Werte einer Messung nahe beieinanderliegen. Das liegt an diesen Einstellungen: Wahl des Arbeitsspeichers von 44 GB, der Aufruf '-C inca12' sowie der Aufruf '–exclusive'.

Kapitel 9

Literaturvergleich

9.1 Einführung in das Kapitel

9.1.1 Mehrgitter-Tutorials

Eine Einleitung in die Theorie der Mehrgitterverfahren in recht einfacher Darstellung erfolgt in [22].
Eine andere einführende Arbeit in die Mehrgittertheorie ist das Tutorial von Briggs, [24].
Die Arbeit von Bastian, [11], gibt einen Überblick über numerische Methoden zur Lösung von partiellen Differentialgleichungen.

9.1.2 Vorstellung eines Konzeptes

Strukturierte/unstrukturierte Gitter

Strukturierte Gitter basieren auf Gittern, deren Punkte untereinander und nebeneinander stehen. Bei unstrukturierten Gittern liegt eine andere Ordnung der Gitterpunkte vor: Zum Beispiel bilden immer 3 Punkte ein Dreieck, und das Gebiet wird dann mit Dreiecken überdeckt.
Die Verfahren ähneln sich, sind aber ohne Umprogammierung inkompatibel.

9.2 Alternative sequentielle Algorithmen

9.2.1 Bestandteile des Mehrgitterprogramms

Die folgenden Literaturangaben beziehen sich hier nur auf Teilbereiche des Mehrgitteralgorithmus.

Glätter

Für die Glätter der Poisson-Gleichung in [80] wurde ein red-black Gauss-Seidel Verfahren vorgestellt.
In [48] werden nur Glätter von Mehrgitter-Verfahren verglichen.
Aus einer anderen wissenschaftlichen Disziplin, nämlich der Finanzmathematik, kommt ein weiteres Glättungsverfahren, siehe [41].

Diskretisierungsfehler

Eine kurze Abhandlung des Diskretisierungsfehlers findet sich in [84]. Im Prinzip geht es dabei um Steigungen von Kurven. Eine auf einer Geometrie beruhende Verfeinerungsentscheidung liegt auch bei dem vom Autor entwickelten Verfeinerungskriterium vor. Eine weitere Abhandlung zu dem Thema Diskretisierungsfehler findet sich in [17], wo ebenfalls Steigungen für die Bestimmung des Diskretisierungsfehlers herangezogen werden.
Eine Fehlerschätzung für die Finite Elemente Diskretisierung wird in [61] dargestellt. Eine Anwendung auf die parallelen adaptiven Mehrgitterverfahren wäre zu klären. Aber diese Verfahren sind inkompatibel.

9.2.2 Sequentielle Mehrgitterverfahren

Nicht adaptive sequentielle Verfahren

Hier erfolgt eine Aufzählung von nicht adaptiven sequentiellen Mehrgitterverfahren. Diese sind wie diskutiert in Abschnitt 9.3.2 als Anwendungen des hier vorgestellten Programms im Prinzip ungeeignet, weil dieses Programm nur bei dynamisch adaptiven Mehrgitterverfahren sein Potenzial voll nutzen kann.

Die Arbeiten in [1],[18] und [27] gehören in diesen Bereich.

Sequentielle statisch und/oder dynamisch adaptive Verfahren

Die Unterscheidung zwischen statischen und dynamischen adaptiven Mehrgitterverfahren erfolgt in Abschnitt 9.3.2.

Von Bastian, der an Entwicklungen zu adaptiven parallelen Mehrgitterverfahren, vgl. [8], gearbeitet hat, stammen auch Anwendungen, vgl. [13] und [14].

Zum Bereich adaptiver Verfahren zählt auch die Arbeit [83], in der Strömungen in 3 Dimensionen bearbeitet werden. In der Arbeit [87] werden keine parallelen Algorithmen behandelt, aber es liegt ein adaptives Mehrgitterverfeinerungskriterium vor. Da der Ausdruck 'die Singularität' verwendet wird mit entsprechenden Abbildungen, ist ein statisch adaptives Verfahren anzunehmen. Verwendet werden aber auch zeitlich abhängige Differentialgleichungen. Interessant ist die Angabe eines Verfeinerungskriteriums. Dort wird die Adaptivität sehr kurz abgehandelt.

Die Arbeit in [71] stellt wegen ihres Kapitels 8 (instationäre Erweiterung) ein dynamisch adaptives Mehrgitterverfahren dar, das aber mit unstrukturierten Gittern arbeitet.

Als mögliche Anwendung eines Parallelisierungsprogramms für dynamisch adaptive Mehrgitterverfahren sind die Shallow Water Equations zu nennen, siehe [36] und [64].

Ein weiteres sequentielles adaptives Verfahren findet sich in [39], welches auch die in dieser Arbeit verwendete Poisson-Gleichung behandelt, allerdings auf unstrukturierten Gittern.

Als klassischen Vertreter dieser Arbeiten, die insbesondere aus der Anfangszeit der Mehrgitterverfahren stammen, ist die Arbeit von Brand [23] zu nennen.
Auch in diesen Bereich adaptiver Mehrgitterverfahren fällt die Arbeit [78].
Die Arbeit [49] ist den statischen adaptiven Mehrgitterverfahren zuzuordnen; in 'Figure 5.16' spricht er von einem stationären Fall. Es ist eine umfangreiche Abhandlung.
Von besonderem Interesse für die Anwendung wäre die Arbeit [40], weil sie sich mit dem Wetter beschäftigt. Sie behandelt sowohl den stationären Fall (Kap. 4) als auch den dynamischen Fall (Kap. 5).

9.3 Vergleiche des Programms mit parallelen Algorithmen

Hier stehen die Arbeiten zur Diskussion, die eine Alternative zum hier vorgestellten Programm darstellen können. Deswegen wird vor der eigentlichen Angabe von kommentierten Literaturstellen eine Diskussion erfolgen mit dem Ziel, herauszuarbeiten, wie der Vergleich mit dem hier entwickelten Programm aussieht.

9.3.1 Drei parallele Methoden

Das hier vorgestellte Verfahren basiert auf folgenden strukturellen Feststellungen

Ausgangspunkt dieser Arbeit waren zwei Punkte: Einen Gitter-Lastausgleich durchzuführen, basierend auf dem Transfer von einzelnen Gitterpunkten, vgl. [53], sowie das am Lehrstuhl entwickelte PLB-Verfahren einzusetzen, bei dem bei jedem Lastausgleich jeweils berechnet wird, wie viel Last zwischen zwei Kernen verschickt werden soll, und von wo nach wo sie transportiert werden sollen, um einen Lastausgleich herzustellen. Man geht dabei aus von einer existierenden Lastverteilung. Nach der adaptiven Verfeinerung erfolgt durch den Transfer von Gitterpunkten ein optional vollständiger Lastausgleich. Mit Hilfe von reversem Lastausgleich, vgl. Abschnitt 6.7.6, wird sogar ein fast vollständiger gleichzeitiger Lastausgleich auf allen Gitterebenen erreicht.

Lastausgleich durch Bisektion

Die Idee ist folgende: Man hat ein Gebiet und teilt dieses auf, z.B. durch Kanten, die ein Gebiet zerlegen. Bei einer rekursiven Bisektion in Bezug auf 16 Kerne teilt man erst das Gebiet durch zwei, die resultierenden Gebiete wieder durch 2, und zwar solange, bis 16 Teile entstanden sind. Dieser Ansatz wird bei dieser Arbeit eben nicht verfolgt.

Statisch versus dynamisch. Man kann diese Methode 'Bisektion' statisch und dynamisch verwenden. Die Anwendungen in [8] sind statischer Art, weil adaptive Verfeinerungsgebiete vorgegeben sind, während sie in [36] dynamisch verwendet wurden, wie man bei einem sich ändernden Verfeinerungsgebiet bei der dort vorgestellten Visualisierung gesehen hat.

Vergleich mit der hier angewendeten Methode. An dieser Stelle werden die Probleme bei der Bisektion aufgezeigt, und zwar getrennt für den allgemeinen Fall, siehe

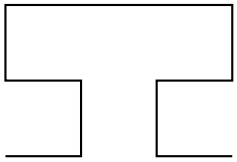


Abbildung 9.1: Eine Peano-Kurve, vgl. Abbildung 2.15 in [57].

den nächsten Absatz, sowie den dynamischen Fall im Besonderen, vgl. den übernächsten Abschnitt.

Der allgemeine Fall eines Bisektionsverfahrens: Bei dem hier entwickelten Programm wurde ein reverser Lastausgleich gemacht, um einen Multi-Level-Lastausgleich zu realisieren. Dieselbe Problematik liegt auch bei einer Bisektion vor, wenn übereinander liegende Gitterpunkte auf demselben Kern liegen. Selbst wenn die Punkte nicht strikt übereinander liegen, ist eine ähnliche Ebenen übergreifende Aufteilung anzustreben. Es führt aber eine perfekte Aufteilung des Gebietes auf einer Ebene zu einer durchaus nicht perfekten Aufteilung auf den anderen Ebenen. Man kann das korrigieren, indem man eine weitere Aufteilung auf den gröberen Ebenen macht, wie beim reversen Lastausgleich. Allerdings kostet das Zeit bei der Programmausführung. Es entstehen auch weitere Kommunikationskosten. Außerdem muss das schon bei der Programmentwicklung berücksichtigt werden. Insgesamt ist eine Multi-Level-Bisektion nicht optimal.

Bei der dynamischen Anwendung in [36] ist zu beachten: Ein Problem besteht darin, dass eine geometrische Aufteilung nicht genau passt zu einer Überdeckung mit Rechtecken, die in diesem Fall gegeben ist. Außerdem müssen diese Rechtecke berechnet und gebildet werden. Im dynamischen Fall muss laufend Speicher allokiert und wieder deallokiert werden.
Dann müssen alte Informationen, wie die bisherigen Mehrgitter-Werte, auf neue Rechtecke gebracht werden. Es muss eine neue Aufteilung zwischen den Kernen durchgeführt, sowie neue Ränder müssen wegen der neuen Aufteilung erkannt werden. Diese notwendigen Arbeiten, die der Computer durchführen muss, wirken sich nachteilig auf die parallele Laufzeit aus, wobei natürlich auch bei der hier vorgestellten Parallelisierung einige dieser Aufgaben erfüllt werden müssen.

Die Methode Peanokurvenverfahren für den Lastausgleich

Eigentlich wird diese Methode nicht bei adaptiven Mehrgitterverfahren verwendet, weil sie nur auf einem unterschiedlich verfeinerten Gitter arbeitet, und deswegen bei finite Elemente Methoden angewendet werden kann. Sonst müssten auf allen Gitterebenen raumfüllende Kurven gebildet, und diese untereinander abgestimmt werden, was nicht gemacht wird. Es wird immer nur eine raumfüllende Kurve, hier die Peano-Kurve, vgl. Abbildung 9.1 verwendet, und das geht nur mit einem unterschiedlich feinen Gitter.

Lässt man diesen Punkt beiseite, so kann man davon ausgehen, dass im Fall der Peano-Kurven Verwendung nur eine statische Parallelisierung erfolgen kann, und zwar aus

folgendem Grund: Schon bei zwei Kernen P und Q und einer auf P beginnenden raumfüllenden Kurve, weiß Q nicht, wo auf P verfeinert wird oder nicht, und kann daher den Anfangspunkt seines Teiles der Kurve nicht selber bestimmen. Die raumfüllende Kurve wird also quasi sequentiell berechnet (zumindestens auf den Kernen nacheinander), was für einen dynamischen Lastausgleich sehr ungünstig ist, abgesehen von den Problemen Ränder zu aktualisieren und schon vorhandene Werte zu verschicken.

Konsequenz: Solche Methoden stellen keine Alternative zum hier entwickelten Programm dar.

9.3.2 Die Diskussion der verschiedenen Mehrgitterverfahren

Zunächst sei angemerkt, dass die Entwicklung nicht adaptiver paralleler Mehrgitterverfahren extrem einfach ist. Ein Lastausgleich braucht nicht zu erfolgen, weil die folgende triviale Aufteilung in jeweils ein Rechteck pro Kern optimal ist. Bei p Mal q Kernen teilt man das Gitter zum Beispiel horizontal durch q-1, und vertikal durch p-1 Linien auf.

Da hier keine Adaptivität stattfindet, wäre für einen Vergleich zu diesem Programm sowohl die parallele adaptive Verfeinerung als auch der Balancer auszuschalten - die eigentliche Hauptarbeit würde überbrückt. Deswegen macht ein solcher Vergleich eigentlich keinen Sinn. Die Folgerung: Nur Parallelisierungsprogramme für adaptive Problemstellungen stellen eine potenzielle Alternative dar.

Statische versus dynamisch adaptive Mehrgitterverfahren

Den Unterschied mit Beispielen erkären. Bei statischer Adaptivität wird das Mehrgittersystem einmal vorab adaptiv verfeinert und dann beibehalten. Eine typische Anwendung wäre hier die Simulation eines Körpers in einem Windkanal. Die adaptive Verfeinerung bleibt erhalten.
Bei dynamischer Adaptivität ändert sich die adaptive Situation. Ein Beispiel dazu wäre eine Wettersimulation bei einem sich bewegenden Sturm. Die adaptive Verfeinerung muss immer wieder angepasst werden, und ein Lastausgleich muss erfolgen. Diese Situation ist der eigentliche Vergleichsfall zu dem hier dargestellten Programm, weil seine Funktionsfähigkeit maximal ausgenutzt wird, da sowohl die parallele adaptive Verfeinerung als auch der Lastausgleich in jeder Iteration genutzt werden.

Lastausgleich in beiden Fällen. Bei statischer Verfeinerung würde die Lastumverteilung nicht genutzt, sondern nur die parallele adaptive Verfeinerung. Man könnte das deshalb immer noch vergleichen, auch die Effizienzen. Aber der eigentliche Vergleich ist die dynamische Verfeinerung. Speziell dafür sind die vielen komplexen Schritte gemacht worden.
Da der Lastausgleich bei statischem Mehrgitterverfahren nur einmal zu Beginn erfolgt, ist es hier wichtig, einen möglichst guten Lastausgleich unter möglichst wenig Kommunikation an Rändern zu erhalten, während bei dynamischem Lastausgleich auch die Geschwindigkeit des Lastumverteilens eine wichtige Rolle spielt.

Wie findet man heraus, ob ein Verfahren nur statisch verfeinert oder auch dynamisch? Normalerweise schreibt ein Autor nicht, dass seine Methode nur bei statischem Mehrgitter eine Anwendung findet. Wenn die Anwendung aber eine feste Struktur hat, wie die 'Nordsee' in [8], oder wenn der Lastausgleich so aufwendig ist, wie dies bei der Anwendung von Peano-Kurven ist, vgl. im Abschnitt 9.3.1 den Teil über Peanokurven, dann kann man darauf schließen, dass das Verfahren statisch adaptiv ist.

Literaturvergleich: Zum Peanokurvenverfahren

Die Diplomarbeit in [35] gibt eine Lösung für Finite Elemente Verfahren an mit Peanokurven. Diese Arbeit basiert auf anderen Arbeiten zu dem Thema, und verwendet ein Programm namens DYNASTY. Für Mehrgitterverfahren ist es nicht geeignet. Für statische adaptive FEM wurde eine Parallelisierung der Initialisierung geschrieben, während für einen Lastausgleich - also dynamisch adaptive FEM - noch Arbeiten ausstanden. Insbesondere geschahen dafür keine Messungen. Die Arbeit [57] gehört ebenfalls in diesen Bereich.
Eine weitere Arbeit über die Verwendung von raumfüllenden Kurven, der Autor nennt sie Hilbert Kurve, stammt von [88]. Erwähnt wird insbesondere der Vergleich mit Bisektionsverfahren: 'Graph partitioning can be expensive'.

9.3.3 Literaturvergleich: Nicht adaptive parallele Mehrgitterverfahren

Hier ist die Arbeit [66] zu nennen, in der für die kompletten Gitterebenen eine Gebietszerlegung gemacht wird zur Aufteilung auf verschiedene Prozesse, bei der aber Einschränkungen an die numerische Genauigkeit gemacht werden - gerade auch deshalb, weil während des V-Zyklus nicht kommuniziert wird. Die Zerlegung vollständiger Gitterebenen (Abschnitt 3.1.2 auf Seite 61) lässt darauf schließen, dass Adaptivität in dieser Arbeit nicht behandelt wird.
In diesen Bereich fällt die Arbeit [7], in der nur sehr wenig über Parallelisierung enthalten ist, und in deren Abbildung 5.3 eine nicht adaptive Parallelisierung beschrieben wird.
Auch die Arbeit [50] behandelt Parallelisierungen, die ganz einfache Aufteilungen verwenden, vgl. die Abbildung dort auf Seite 7.
In der Arbeit [25] basiert die Parallelisierung auf einfach strukturierten Partitionen. Anwendung ist die Simulation des Wetters.
In [10] werden in Kapitel 4 (Seite 31/32) fest vorgegebene Gitter Ω_l definiert, die auf ein nichtadaptives Mehrgitterverfahren schließen lassen. Sonst benötigt man ein Verfeinerungskriterium, welches die Gitter festlegt.

9.3.4 Literaturvergleich: Über parallele adaptive Mehrgitterverfahren

Ein allgemeiner Vergleich paralleler adaptiver Mehrgitterverfahren

Zuerst zu [56]: Dort wird insbesondere geschrieben, dass die Programmierung von parallelen adaptiven Mehrgitterverfahren eine sehr aufwendige Tätigkeit ist. Hierher gehört

auch die Arbeit [12], mit den typischen adaptiven Strukturen eines statisch adaptiven Mehrgitterverfahrens auf unstrukturierten Gittern. Unter typisch statisch adaptiven Strukturen ist dabei folgendes zu verstehen: Die Gitter werden immer feiner, je näher sie gewissen Grenzpositionen kommen.
In [28] wird zu Beginn eine Übersicht über traditionelle Verfahren gegeben, in der bisektionsähnliche Verfahren und die Verwendung von raumfüllenden Kurven angesprochen werden.

Die Verwendung von Bisektionsverfahren

Bei der Arbeit [60] ist von einem statischen Mehrgitterverfahren auszugehen, weil nur stationäre Probleme behandelt werden können, vgl. die Seite 84 im Ausblick dieser Arbeit. Eine Erweiterung auf zeitabhängige Probleme wird als möglich angegeben. Interessant dabei sind die das Gitter partitionierenden Patches (Abschnitt 3.2: Die Patch-Gitterstruktur), die sich aber im Allgemeinen über große Gebiete erstrecken, im Unterschied zu der Punktauflösung in dieser Arbeit. Im Gegensatz zum reversen Lastausgleich geschieht keine Aufteilung der Gitter, die gemeinsam ist für alle Gitterebenen. An Stelle dessen gibt es in dieser Arbeit neben horizontaler auch vertikale Kommunikation, vgl. Seite 24, d.h. Kommunikation zwischen den Gitterebenen. Es wird aber die Notwendigkeit zur Minimierung beider Kommunikationen gesehen. Wie auch in der vorliegenden Arbeit werden Grobgitter-Interpolationsinformationen über das feine Gitter hinweg verwendet, vgl. den Abschnitt über die Randinterpolation mit 'schräger' Kommunikation, vgl. Seite 40.

Die Arbeiten [8] und [9] stellen ein statisch adaptives Mehrgitterverfahren vor.

Die Arbeit [36] stellt ein dynamisch adaptives Mehrgitterverfahren dar. Die Problematiken dieses Verfahrens sind oben in Abschnitt 9.3.1 beschrieben worden.

In der Arbeit [55] über dynamisch adaptive Mehrgitterverfahren werden die Konzepte verteilte adaptive Verfeinerung, Lastverteilung und Lastmigration behandelt, die für ein dynamisches Verfahren sprechen. Diese drei Methoden kommen auch in der vorliegenden Arbeit vor. Als Unterscheidung zu der hier vorgestellten Arbeit basiert der Lastausgleich auf Clusterung (7.4), d.h. es werden größere Gebiete migriert.

Bemerkung: Bei der parallelen Verfeinerung in [55] werden Kleinstelemente unterteilt, wodurch beispielsweise ein 2 x 2 Gitter zu einem 3 x 3 Gitter wird. Auf einem 3 x 3 Gitter zu rechnen macht numerisch gesehen nicht viel Sinn. Es müssen sich dann mehrere 3 x 3 Gitter gegenseitig ergänzen.

Ein Bisektionsverfahren wird auch in [32] präsentiert. Es ist dynamisch adaptiv. Denn es wird auf Seite 102 erklärt, dass dynamische und parallele Lastbalancierung erforderlich ist.
Bei der Verfeinerung werden unstrukturierte Gitter in Form von Tetraedern verwendet. Die Verfeinerung ist damit prinzipiell anders als im Programm in dieser Arbeit. Es wird eine Multi-Level-Graphpartitionierung (5.3.5) verwendet, eine Lösung zu einem oben angesprochenen Problem, welches in dieser Arbeit mit dem reversen Lastausgleich

gelöst wurde.

In [5] wird ein Partitionierungsverfahren vorgestellt, vgl. Abbildung 4 dort. Es geht um ein Mapping zwischen Gebieten und Prozessoren, hat also dieselbe Funktion wie ein Bisektionsverfahren. Es wird nur auf einem Gitter durchgeführt, denn es liegt ein Finite Elemente Verfahren vor. Ein weiterer Vergleich mit dem hier entwickelten Programm ist nicht da, wohl aber zu parallelen Bisektionsverfahren.

Kapitel 10

Schlusswort/Schlussbetrachtung

10.1 Was erarbeitet worden ist:

10.1.1 Strukturell wurde erreicht:

Unterstützende Strukturen wurden geschaffen.

Dazu gehören das *Farbenfeld*, *Selbstnachbarn*, die *dynamischeRandpunktliste*, *mv-vertikal*, die *Punktfeldliste* und *EP-SNB-Update*. Auch die Bestimmung der Randstruktur durch die 2 Methoden, die Vektorübertragung und die Sortierung, wurde realisiert.

Bei der parallelen adaptiven Verfeinerung wurde erreicht:

Das Erkennen der parallelen Situation durch die Bestimmung der Punktmengen P_k^+ und P_k^-, und die Änderungen in den Strukturen *Farbenfeld* und *Feldliste* sowie *ExtremeVektoren* durchzuführen, war eine große Herausforderung. Das Finden der Strukturen EP^+ und EP^-, sowie der Struktur *EP-SNB*, die Ableitung von P_k^+ und P_k^- sowie die Durchführung der entsprechenden Punkt- und Farbenupdates, ist gut durchdacht. Zudem wird bewiesen, dass die Verwendung dieser Strukturen völlig korrekt ist.

Diese zusammenpassenden Strukturen zur effizienten Behandlung, die nur auf Änderungen der betroffenden Strukturen basieren, wurden nicht sofort gefunden, sondern nur durch stetige Weiterentwicklung. Hervorzuheben ist dabei die beschleunigte Bestimmung von P_k^+ und P_k^- mittels *EP-SNB* - eine Struktur, die wichtig, aber aus der Problemstellung so nicht abzuleiten ist.

Beim Lastausgleich wurde erreicht:

Mithilfe der Struktur *mv-vertikal* kann bei den verschiedenen Schritten beim Packen und Entpacken auf die gesamte Information über verschickte Punkte auf allen Gitterebenen zugegriffen werden.

In der Arbeit wurde die Idee präsentiert, Farbfeldinformation beim Packen und Entpacken des Buffers zu reduzieren mittels einer Markierung der übertragenen Punkte.

Entwickelt, aber hier nicht dargestellt, wurde das Konzept der Nachkorrektur, um Farbenfeldfehler zu verhindern, zusammen mit der späteren Verbesserung zur Zwischenkor-

rektur, bei der durch ein verbessertes Timing eine Reihe von Kommunikationsschritten obsolet werden.

Der reverse Lastausgleich, zuzüglich der *ev2*-Strategie für einen vollständigen Lastausgleich auf allen Gitterebenen, bildet einen weiteren wichtigen Schritt hin zu einem effizienten Programm.

10.1.2 Konzeptionell wurde erreicht:

Das Scheduling bei adaptiven Mehrgitterverfahren

Dieses Konzept, einzelne Gitterpunkte verschicken zu können, in Kombination mit dem die Lastwerte zu bestimmenden Verfahren PLB, wurde mit dieser Arbeit gründlich untersucht. Es wurden nur wenige Annahmen gemacht wie die, dass gemeinsame Gitterpunkte auf demselben Kern liegen müssen.

Abgesehen von den möglichen alternativen Lösungen zu Problemen wurde ein allgemeines Konzept für die Behandlung des Scheduling-Problems bei adaptiven Mehrgitterverfahren entwickelt, womit die Frage entschieden wird, wie man so etwas realisieren kann. Das ist eine echte Behandlung dieser Aufgabe, die allerdings auch sehr aufwendig geworden ist.

10.2 Vor- und Nachteile

10.2.1 Vorteile: Die numerische Leistungsfähigkeit

Das dynamisch adaptive Mehrgitterverfahren wurde realisiert. Dabei hat man pro zu verfeinernder Gitterstelle eine 5 x 5 Überdeckung mit Feingitterpunkten. Insgesamt wird das jeweilige Gebiet eines Kernes eingeschlossen in einen Außenrand bekannter Zuordnung von Punkten zu Kernen, was 9-Punktsterne ermöglicht.

10.2.2 Vorteile: Guter und schneller Lastausgleich

Es liegt ein guter Lastausgleich vor:

1. Mit der PLB-Methode ist ein vollständiger Lastausgleich auf einer Ebene möglich.
2. Mit dem reversen Lastausgleich und der *ev2* Strategie wird daraus ein Multi-Level-vollständiger Lastausgleich.

Ferner liegt ein schneller Lastausgleich vor:

1. Das Konzept des Scheduling-Algorithmus erfordert nur eine Lastmengenberechnung zusammen mit einem Punkttransfer, dessen ausgewählte Punkte schnell bestimmt werden können.

   Diese Methode steht dem sonst üblichen Bisektionsverfahren oder anderen Gebietszerlegungsmethoden gegenüber.

   Aufgrund des Verschickens von Gitterpunkten eines allokierten Gitters sind währ-

end der parallelen Iterationen keine weiteren Allokationen und de-Allokationen nötig.

2. Durch die Auswahl der Gitterpunkte bei der Migration wird erreicht, dass die Aufteilung des Gitters zu der Kerntopologie passt. Diese Auswahl ist plausibel, gut und schnell.

3. Bei der Zwischenkorrektur wird durch geschicktes Timing Kommunikation gespart.

4. Die parallele adaptive Verfeinerung berechnet mithilfe von Änderungsinformation sowie der Klasse *EP-SNB-Update* schnell die Punktänderungsmengen P_k^+ und P_k^-.

5. Auch die Kommunikation des Farbenfeldes erfolgt durch das Konzet der Markierung transferierter Punkte besonders schnell, weil der Kommunikationsbuffer dadurch verkürzt wird.

6. Ferner ermöglicht das nach Änderungen zu erfolgende Bestimmen und Speichern der Ränder, den Randaustausch nur auf die Werte zu beschränken, sodass die Koordinaten dabei nicht mehr übertragen werden müssen.

10.2.3 Nachteile

Man hat die Informationen nur auf einem äußeren Rand der Breite von einem Punkt. Informationen auf weiter entfernten Punkten werden nur erreicht bei der Randinterpolation sowie der Bestimmung des Verfeinerungskriteriums. Allerdings kann das prinzipiell erweitert werden.

Ansonsten benötigt der Algorithmus entsprechend viel Speicherplatz, damit er auf jedem Kern überall Punkte entstehen lassen kann. Dadurch kann fast jede Lastsituation abgebildet werden. Dementsprechend ist der Algorithmus sehr flexibel.

Zur Kompensation dieser Nachteile könnte man das Programm weiterentwickeln, vgl. den Abschnitt 10.3.

Bisher läuft das Programm auf einem, zwei, vier, acht und sechzehn Kernen. Für eine Anwendung mit sehr vielen Kernen müsste vermieden werden, dass aufgrund der Punktmigrationen viele Punktnachbarschaften entstehen können, vgl. den Abschnitt 10.3.

10.3 Ausblick

10.3.1 Speicherplatz sparen

Man kann das Programm modifizieren, sodass es weniger Speicherplatz benötigt. Eine Konsequenz ist dann die, dass deswegen auf einem Gitter nicht mehr jede Punktverteilung möglich ist. Man vergleiche diese Möglichkeiten:

Speicherplatz sparen: Möglichkeit 1

Man weiß, welchen Gitterbereich ein Kern in der Startaufteilung zugewiesen bekommt. Wenn man diesen nach allen Seiten hinreichend erweitert, kann der Kern einerseits auch woanders arbeiten, müßte aber nicht das gesamte Gitter allokieren. Diese Möglichkeit ist im Prinzip nicht so schwierig zu realisieren. Allerdings kann mancher Kern weit entfernte Gebiete dann nicht behandeln.

Speicherplatz sparen: Möglichkeit 2

Um z.B. im Bereich der Wettervorhersage das Entstehen eines Hurrikans prognostizieren zu können, ist ein erhöhter Rechenaufwand erforderlich. Dabei lässt sich die Beobachtung nutzen, dass Hurrikans nur in bestimmten Regionen entstehen, insbesondere vor der Ostküste Nordamerikas. So könnte man jedem Kern einen erweiterten eigenen Bereich zuweisen - z.B. die Arktis - und einen gemeinsamen Bereich, wie den vor der Ostküste Nordamerikas.

Dabei wäre Folgendes zu realisieren: Man legt Zeiger für jede Zeile an. Diese unterteilt man in 2^k Bereiche. Unter denen bekommen diejenigen Speicherplatz zugewiesen, die dem eigenen erweiterten Bereich sowie dem potenziell gemeinsamen Bereich entsprechen. Auf diese Weise würde alles völlig parallel abgearbeitet, der eigene wie der gemeinsame Bereich.

Die einzige Einschränkung wäre, vorher gemeinsame Bereiche festzulegen, d.h. die Problemstellung muss so eine Einschränkung erlauben.

Ferner muss dann beim Lastausgleich schnell entschieden werden, ob ein Gitterpunkt zu einem Kern geschickt werden kann. Das geht nur, wenn dazu Speicherplatz zur Verfügung steht.

10.3.2 Farbenfeld erweitern

Bisher ist das Farbenfeld auf dem Außenrand um das Gebiet herum im Einpunktabstand gültig. Das könnte man erweitern auf einen größeren Abstand vom eigenen Gebiet mit mehr numerischen Möglichkeiten. Allerdings hat das dann Einfluss auf andere Strukturen, wie z.B. die Koordinaten der Randstrukturen. Diese müssten dann auch erweitert werden.

De facto müsste geprüft werden, welche Strukturen in welcher Weise zu modifizieren wären.

10.3.3 Größere Probleme behandeln und mehr Kerne verwenden

Dazu ist das Sparen von Speicherplatz erforderlich, vgl. Abschnitt 10.3.1. Ein anderes Problem besteht darin, dass bei sehr vielen Kernen im Laufe der Iterationen mehr Punktnachbarschaften entstehen könnten. Ein Ansatz, das zu behandeln, wäre ein Remapping, vgl. [29]:

'Remapping'

Wenn man sehr viele Iterationen rechnen wollte, könnte man eine Verbesserung erreichen, indem man einem Gitter seine Anfangsaufteilung zuteilt. Da dann jeder Punkt weiß, zu welchem Kern er gehört, müssten die Gitterwerte entsprechend auf den Weg gebracht werden. Man könnte die Werte entlang dem Kerngitter des Lastausgleiches verschicken, sodass keine all to all Kommunikation anfallen würde. Aufwendig zu programmieren ist das im Prinzip nicht. Allerdings ist am Ende der Prozedur die Last im Allgemeinen unausgeglichen, d.h. es müsste sich ein Lastausgleich anschließen, der wiederum zu einer anderen Aufteilung als der Anfangsaufteilung führt.

10.3.4 Andere Anwendungen

Eine interessante Anwendung ist die Wettervorhersage. Wenn man z.B. das Wetter global darstellen wollte, könnte man als Grundgebiet kein Rechteckgitter nehmen, sondern beispielsweise einen Globus darstellen. Dafür müsste man aber auch in der parallelen adaptiven Verfeinerung Änderungen vornehmen. Es wären beispielsweise Datenstrukturen wie *EP-SNB-Update* anzupassen.

Literaturverzeichnis

[1] S.N.S. Acharya, Multigrid Conjugate Gradient Method, Master Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2006

[2] G. Alefeld, I. Lenhardt, H. Obermaier, Parallele numerische Verfahren, Springer 2002

[3] AMD Software Optimization Guide for AMD Family 15h Processors, Publication No. 47414, Revision 3.08, Date january 2014

[4] G. Amdahl, Validity of the Single Prozessor Approach to Achieving Long-Scale Computing Capabilities. In: AFIPS Conference Proceedings 30, 1967, S. 483 - 485

[5] T. Apel, U. Reichel, Partitioning of finite element meshes for parallel computing: A case study, Preprint-Reihe des Chemnitzer SFB 393, SFB393/96-18a, 1997

[6] M.J. Atallah, Algorithms and theory of computation handbook, Boca Raton, Fl: CRC Press, 1998

[7] M. Bader, Robuste, parallele Mehrgitterverfahren für die Konvektions-Diffusions-Gleichung, Dissertation der Technischen Universität München, 2001

[8] P. Bastian, Parallele adaptive Mehrgitterverfahren, Stuttgart: Teubner, 1996

[9] P. Bastian, Load balancing for adaptive multigrid methods, 21 Seiten

[10] P. Bastian, Die Frequenzzerlegungsmethode als robustes Mehrgitterverfahren: Implementierung und Parallelisierung, Diplomarbeit (die übertragene Datei heißt diplom.pdf)

[11] P. Bastian, Numerische Lösung partieller Differentialgleichungen, Universität Stuttgart, Institut für Parallele und Verteilte Systeme, 2008

[12] P. Bastian, R. Helmig, Efficient Fully-Coupled Solution Techniques for Two-Phase Flow in Porous Media, eine 35 Seiten Ausarbeitung. Parallel multigrid solution and large scale computations

[13] P. Bastian, V. Reichenberger, Multigrid for Higher Order Discontinuous Galerkin Finite Elements Applied to Groundwater Flow, Interdisziplinäres Zentrum für Wissenschaftliches Rechnen, Universität Heidelberg

[14] P. Bastian, G. Wittum, On Robust and Adaptive Multi-Grid Methods

[15] M. Ben-Ari, Principles of Concurrent and Distributed Programming, Prentice Hall International (UK) Ltd., 1990

[16] G. Bengel, C. Baun, M. Kunze u.a., Masterkurs Parallele und Verteilte Systeme, Vieweg und Teubner, 2008

[17] P. Benner, Vorlesung Numerische Mathematik, Technische Universität Chemnitz, SS 2006

[18] S. Bergler, Mehrgitterverfahren für beliebige Gitterweiten mit Anwendung in der Quantenchemie, Diplomarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2007

[19] M. Böhm, Verteilte Lösung harter Probleme: Schneller Lastausgleich PhD thesis, Köln 1996

[20] M. Böhm and E. Speckenmeyer. A dynamic processor tree for solving game trees in parallel. In Methods of Operations Research, volume 63, pages 479 - 489, 1989

[21] M. Böhm and E. Speckenmeyer. A fast parallel SAT-Solver - efficient workload balancing. Annals of Mathematics and Artificial Intelligence 17 (1996) 381-400

[22] A. Borzi, Introduction to multigrid methods, Institut für Mathematik und Wissenschaftliches Rechnen Karl-Franzens-Universität Graz, Austria

[23] A. Brandt, Multi-Level Adaptive Solutions to Boundary-Value Problems. Mathematics of computation, Volume 31, Number 138, 1977, Seiten 333-390

[24] W. L. Briggs, A Multigrid Tutorial, Presented by Van Emden Henson Center for Applied Scientific Computing Lawrence Livermore National Laboratory

[25] S. Buckeridge, Numerical Solution of Weather and Climate Systems, for the degree of Doctor of Philosophy of the University of Bath Department of Mathematical Sciences November 2010

[26] V. Achter, S. Borowski, L. Nieroda, L. Packschies, V. Winkelmann, CHEOPS Brief Instructions, Stand 07.10.2013

[27] I. Christadler, Mehrgitterverfahren für die Berechnung des Optischen Flusses mit Nicht-Standardregularisierungen, Diplomarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2004

[28] K. D. Devine, E. G. Boman, G. Karypis Partitioning and Load Balancing for Emerging Parallel Applications and Architectures, in Parallel Processing for Scientific Computing, herausgegeben von M. A. Heroux, P. Raghaven and H. D. Simon, 2006

[29] E. Speckenmeyer, persönliche Mitteilung

[30] M. Flynn, Some computer organisations and their effectiveness, IEEE Trans. Comput., (9), 948-960, 1972

[31] T.L. Freeman and C. Phillips, Parallel Numerical Algorithms, Prentice Hall International 1992

[32] S. Groß, Parallelisierung eines adaptiven Verfahrens zur numerischen Lösung partieller Differentialgleichungen, Diplomarbeit der Technischen Hochschule Aachen, 2002

[33] J. R. Gurd, A taxonomie of parallel computer architectures. Proceedings of the international conference on the design and application of parallel digital processors. Lisbon, Portugal. IEE 1988

[34] G. Haase, Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen, B.G. Teubner Stuttgart Leizig, 1999

[35] W. Herder, Lastverteilung und parallelisierte Erzeugung von Eingabedaten für ein paralleles Cache-optimales Finite-Element-Verfahren, Diplomarbeit, Technische Universität München, 2005

[36] R. Hess, Dynamisch adaptives Mehrgitter auf Parallelrechnern für eine semiimplizite Diskretisierung der Flachwassergleichungen, Dissertation, 1998

[37] R. Hess, persönliche Mitteilung

[38] R. Hess, Programmcode zu seinen Flachwassergleichungen

[39] O. Iliev, D. Stoyanov, Multigrid - adaptive local refinement solver for incompressible flows, Berichte des Fraunhofer ITWM, Nr. 54 (2003)

[40] C. Jablonowski, Adaptive grids in weather and climate modeling, A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Atmospheric and Space Sciences and Scientific Computing) in The University of Michigan 2004

[41] T. Jahnke, Vorlesung Numerische Methoden in der Finanzmathematik II Sommersemester 2011, Karlsruher Institut für Technologie

[42] J. Jaja, An Introduction to Parallel Algorithms, Addison-Wesley Publishing Company Inc. 1992

[43] W. Joppich, Grundlagen der Mehrgittermethode, Einführung in Standardverfahren, Shaker Verlag 2011

[44] W. Joppich, Mehrgitterkurs, mehrfach durchgeführt

[45] H. F. Jordan, G. Alaghband, Fundamentals of Parallel Processing, Pearson Education Inc. 2003

[46] A. Kaminsky, Building Parallel Programs, Course Technology 2010

[47] David B. Kirk und Wen-mei W. Hwu, Programming Massively Parallel Processors, Elsevier Inc. 2010

[48] R. Konrath, Vergleich verschiedener Glätter bei Multigrid-Verfahren mittels Toeplitz-Techniken, Diplomarbeit, Technische Universität München, 2000

[49] R. Kornhuber, Adaptive Monotone Multigrid Methods for Nonlinear Variational Problems, Stuttgart 1996

[50] J. Krupka, I. S. Simecek, Parallel solvers of Poisson's equation, Department of Computer Systems, Faculty of Information Technology, Czech Technical University, Prague, MEMICS 2010

[51] D. J. Kuck, High-Speed machines and their compilers. In Parallel processing systems (ed. D. Evans), Cambridge University Press 1982

[52] V. Kumar, A. Grana, A. Gurten, G. Karypsis Introduction to parallel computing, Benjamin Kummings Publishing Company 1994

[53] V. Kutzschebauch, Dynamischer Lastausgleich im Parallelen Meteorologischen Modell MPMM, Dissertation Köln 1998

[54] B. Lang, Digitale Bildverarbeitung mit kombinierter Pipeline- und Parallel-Architektur, PhD thesis, Hamburg 1992

[55] S. Lang, P. Bastian, Parallele Numerische Simulation instationärer Probleme mit adaptiven Methoden auf unstrukturierten Gittern, Universität Stuttgart, Dissertation 2001

[56] S. Lang, P. Bastian, Parallele adaptive Mehrgitterlöser, http://www.rus.uni-stuttgart.de/bi/1995/1/File12.html
'rus' steht für Rechenzentrum Uni Stuttgart

[57] M. Langlotz, Parallelisierung eines Cache-optimalen 3D Finite-Element-Verfahrens, Diplomarbeit, Technische Universität München, 2004

[58] LEDA: Internetseite mit Literaturdownload, www.algorithmic-solutions.com/leda/index.htm

[59] F. T. Leighton, Introduction to Parallel Algorithms and Architektures: Arrays-Trees-Hypercubes. Morgan Kaufman Publishers, San Mateo, CA, 1992

[60] H. Lötzbeyer, Parallele adaptive Mehrgitterverfahren, ein objektorientierter Ansatz auf semistrukturierten Gittern, Diplomarbeit, Technische Universität München, 1996

[61] M. Lonsing, Zwei a posteriori Fehlerschätzer für die Wärmeleitungsgleichung, Diplomarbeit, Hattingen 1998

[62] R. Lüling, B. Monien, und F. Ramme. A study of dynamic load balancing algorithms. In Proceedings of the 3rd IEEE SPDP, Seiten 686-689, 1991

[63] F. Meisgen, Dynamische Lastausgleichsverfahren in heterogenen Netzwerken PhD thesis, Köln 1998

[64] B. L. Mitchell, S. R. Fulton, Adaptive Multigrid Solution of the Shallow Water Equations, Technical Report No. 2000-02, Department of Mathematics and Computer Science, Clarkson University, Potsdam, New York

[65] J. J. Modi, Parallel Algorithms and Matrix Computation, Oxford University Press, 1988

[66] M. Mohr, Kommunikationsarme parallele Mehrgitteralgorithmen, Diplomarbeit, Technische Universität München, 1997

[67] G. E. Moore, Cramming more components onto integrated circuits, in: Electronics 38, Nr. 8, 1965, Seite 114 - 117

[68] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 1.0. www.mpi-forum.org, 1994

[69] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 1.3. www.mpi-forum.org, 2008

[70] Oosterlee, Vortrag am Lehrstuhl Speckenmeyer

[71] A. Papastavrou, Adaptive Finite Element Methoden für Konvektions-Diffusionsprobleme, Dissertation, Ruhr-Universität Bochum, 1998

[72] W.P. Petersen and P. Arbenz, Introduction to Parallel Computing, Oxford University Press 2004

[73] PVM : parallel virtual machine - a users guide and tutorial for networked parallel computing / A. Geist et al, Massachusetts Institute of Technology, 1994

[74] T. Rauber und G. Rünger, Multicore: Parallele Programmierung, Springer Verlag 2008

[75] T. Rauber und G. Rünger, Parallel Programming for Multicore and Cluster Systems, Springer Verlag 2010

[76] M. Rickert, Traffic Simulations on Distributed Memory Computers, PhD thesis, Center for Parallel Computing, University of Cologne, Germany, 1998

[77] Regionales Rechenzentrum der Universität Hannover, C++ für C - Programmierer, 12., veränderte Auflage März 2002

[78] U. Rüde, Fully adaptive multigrid methods, Technische Universität München

[79] Stockfish Chess entwickelt von T. Romstad, M. Costalba und J. Kiiski, auf stockfishchess.org

[80] M. Stürmer, Optimierung des Red-Black-Gauss-Seidel-Verfahrens auf ausgewählten x86-Prozessoren, Studienarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2005

[81] Toga Chess entwicklt von T. Gaksch basierend auf Fruit entwickelt von F. Letouzey

[82] U. Trottenberg, C.W. Oosterlee and A. Schüller, Multigrid Methods: Basics, Parallelism and Adaptivity, 2001

[83] E. Wagner, Adaptive Gitterverfeinerung für eine Strömungssimulation in drei Dimensionen, Diplomarbeit, Technische Universität München, 1996

[84] Wikipedia, Stichwort Diskretisierungsfehler, Online-Lexikon

[85] Wikipedia, Stichwort Intel-Core-i-Serie, Online-Lexikon, 16.5.2015

[86] Wikipedia, Stichwort Open MP, Online-Lexikon, 19.5.2015, auf de.wikipedia.org/wiki/OpenMP

[87] H. Wörndl-Aichriedler, Adaptive Mehrgitterverfahren in Raum und Zeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1999

[88] G. Zumbusch, A parallel adaptive multigrid method, Institut für Angewandte Mathematik, Bonn

Anhang A

Die Benutzung des Programms

A.1 Fehlertester

Dieser Abschnitt ist gedacht für den Programmentwickler, insbesondere für den, der Änderungen am Programm vornehmen möchte, z.B. für eine andere Anwendung.

Die Fehlertester dienten bei der Programmentwicklung dem Aufspüren von Fehlern, die dann entfernt wurden. Die Tests sind, da sie Laufzeit kosten, während der Messungen deaktiviert.

Ferner: Man hat zu jedem Test 5 Informationen: Die Wichtigkeit des Tests, den Namen des Tests, die Aufgabe des Tests, d.h. was er bringt, wie er funktioniert sowie die Zugehörigkeit zu einem gewissen Programmteil. Letzteres wird aufgeteilt in die unterstützenden Strukturen zum Anfang, die parallele adaptive Verfeinerung sowie den Lastausgleich, und am Ende den allgemeinen Fall, z.B. in Bezug auf die Klasse *Programm*. In dieser Reihenfolge werden die Tests auch präsentiert. Der Name der Tests wird nur angegeben, wenn er mit dem Namen des Abschnittes nicht übereinstimmt.

A.1.1 Farbenfeldtester

Die Wichtigkeit des Tests

Das ist der wichtigste Test. Der Grund dafür: Die meiste Arbeit bei diesem Programm bestand darin, die Farbfeldwerte in ihrem Gültigkeitsbereich - eigenes Gebiet plus Außenrand - korrekt und schnell zu bestimmen. Denn sie werden bei der parallelen Mehrgitterberechnung wegen des Randaustauschs immer gebraucht, wenn das Programm mit mehr als einem Kern läuft. Man sieht vor allem bei dem Farbenupdate der parallelen adaptiven Verfeinerung, wie viel Arbeit erforderlich ist, um diese Aufgabe zu erfüllen. Das Programm in dieser Hinsicht auf Korrektheit zu testen ist also sehr wichtig gewesen.

Die Aufgabe des Testes

Es wird für jede Gitterebene überprüft, ob die Farbwerte auf dem Gebiet plus Außenrand korrekt sind. Dabei wird mitgetestet, dass derselbe Punkt sich nicht auf zwei (verschiedenen) Kernen befindet.

Die Realisierung des Testes

Algorithmus A.1: Farbenfeldtester

Input: Gitterebene h

1. Sammeln der Koordinaten auf Ebene h in einen Buffer

2. Kommunikation dieser Buffer mit jedem anderen Kern

3. Vorbereiten der Vergleichsfelder, beschrieben mit dem Wert -1 (steht für eine Stelle ohne Gitterpunkt).

4. Empfangen aller Nachrichten, die an den eigenen Kern geschickt wurden

5. Entpacken, indem beim Empfang von einem Kern der Farbe a für alle gelesenen Koordinaten (f_i,g_i) die Farbe a ins Farbenfeld an die Stellen (f_i,g_i) geschrieben wird. Sollte an eine bereits geschriebene Stelle geschrieben werden (Farbenfeld dort nicht -1), dann wird ein Fehler gemeldet.

6. Vergleiche dieses *Farbenfeld* mit dem *Farbenfeld* des *Gitters* dieser Ebene für alle eigenen Gitterpunkte sowie den Außenrand. Bei einem Unterschied in den Farbwerten der Felder kommt es zur Fehlermeldung.

Die Zugehörigkeit zu einem Programmteil

Formal gesehen, und deswegen auch am Anfang präsentiert, gehört dieser Tester zum Farbenfeld, und damit zu den unterstützenden Klassen. Er ist programmiert in der Klasse *Gitter*. De facto war die Hauptanwendung dieses Testers die Kontrolle des Farbenupdates der *ParallelenAdaptivenVerfeinerung*. In der Tat konnten damit auch Fehler in der Datenstruktur *EP-SNB-Update* der *ParallelenAdaptivenVerfeinerung* gefunden werden. Dieser Test hat ebenso eine Bedeutung für die Migration der Gitterpunkte.

A.1.2 Test der dynamischen Randliste

Die Wichtigkeit des Testes

Da die Bildung dieser Struktur nur in der Klasse *Gitter* erfolgt und für einen überschaubaren Bereich, ist die Korrektheit der Struktur *DynamischeRandliste* schnell verifiziert.

Die Aufgabe des Testes

Hier wird getestet, ob diese Liste den Rand anzeigt. Dabei geht es sowohl um den echten Rand, wie auch den Rand zu einem anderen Kern. Es handelt sich hierbei um die Innenliste.

Die Realisierung des Testes

Algorithmus A.2: Test der dynamischen Randliste
Input: Den Wert h der Gitterebene

1. **for all** (f,g) ∈ Gitter
2. **if**(*selbstnachbarn*(f,g)!=255)
3. **if** (*dynamische_randliste.feld*(f,g) = *false*) **then**
4. Fehlermeldung Typ 1
5. **endif**
6. **endif**
7. **end for**
8. **for all** (f,g) ∈ Rand
9. **if** (*selbstnachbarn*(f,g) = 255) **then** Fehlermeldung Typ 2 **endif**
10. **end for**

Bemerkung: In Zeile 2 wird geprüft, ob ein Randpunkt vorliegt. In Zeile 3 wird geprüft, ob der Punkt (f,g) in der Randliste verzeichnet ist. In den Zeilen 8+10 wird die dynamische Randliste durchlaufen. Und in Zeile 9 wird geprüft, ob ein Punkt vorliegt, der nicht am Rand liegt.

Zugehörigkeit zu einem Programmteil

Hier liegt der Test einer unterstützenden Klasse vor, die ein Bestandteil von Kapitel 4 ist.

A.1.3 Test der Werte von *Selbstnachbarn*

Die Realisierung des Tests

Man durchläuft das Gitter und prüft entsprechend dem Farbenfeld jedes Bit von *Selbstnachbarn*.

Die Wichtigkeit des Tests

Bemerkung: Der Aufbau und die Aktualisierung dieser Struktur ist einfach, indem für alle Himmelsrichtungen die tatsächlichen Bits mit denen entsprechend dem Farbenfeld verglichen werden.

A.1.4 Test der Werte von *EP-SNB-Update*

Die Zugehörigkeit zu einem Programmteil

Die Funktion dazu steht in der Klasse *ParalleleAdaptiveVerfeinerung* und lautet:

test_EP_SNB_beschraenkt

Die Realisierung des Tests

Algorithmus A.3: Test von *EP-SNB-Update*
Wird gemacht für eine Instanz der *ParallelenAdaptivenVerfeinerung*, welche sich im Wesentlichen immer auf ein gröberes und feineres Gitter bezieht. Bei *EP-SNB-Update* geht es aber nur um das gröbere Gitter. Der Test für diese Datenstruktur geschieht auf die folgende Weise:

1. Markierung aller Koordinaten des Gebietes plus des Außenrandes
2. Für diese Punkte (f,g) wird (*EP-SNB-Zahl*(f,g) **AND** Zahl der Himmelsrichtung HR) verglichen mit *feld_aktuell*(HR(f,g)) für alle Himmelsrichtungen HR (*feld_aktuell* = *true* entspricht *EP-SNB-Zahl* **AND** HR-Zahl $\neq 0$, *feld_aktuell* = *false* entspricht *EP-SNB-Zahl* **AND** HR-Zahl = 0)
3. Falls eine Diskrepanz auftritt:

   Fehlermeldung mit Ausgabe der Koordinate, wo der Fehler auftritt, der Himmelsrichtung HR, für die er auftritt, und ob *EP-SNB-Zahl* in dieser Himmelsrichtung fälschlich ein Bit '$\neq 0$' (return 1000 + HR-Zahl (> 1000)) oder ein Bit '$= 0$' (return HR - Zahl (<1000)) gesetzt hat. Die Programmierung ist so, dass dieses alles erkannt und ausgegeben wird.

Die Wichtigkeit des Tests

Zu Beginn der Verwendung von *EP-SNB-Update*, und für das Wechselspiel von der *ParallelenAdaptivenVerfeinerung* und der *Migration* war der Test durchaus wichtig. Insbesondere führten Fehler von *EP-SNB-Update* zu Fehlern im Farbenfeld, d.h. es gab daher ein Wechselspiel zwischen diesem Test und dem Farbenfeldtest, vgl. Abschnitt A.1.1.

A.1.5 Test von ev2

Die Aufgabe des Tests

Diese besteht darin, für alle Koordinaten (f,g) $\in$ *ev2* zu prüfen, ob diese auf der Ebene vorhanden sind, aber auf der Ebene darunter nicht. Genau das ist die Punktmenge von *ev2*.

Die Realisierung des Tests

Ohne Einschränkung wird er für eine Gitterebene h gemacht. Insgesamt wird das dann für jede Gitterebene gemacht, für die *ev2* definiert ist.

Algorithmus A.4: Test auf Korrektheit von *ev2*

1. **for all** (f,g) $\in$ *ev2* (Ebene h)
2. **if**((feld(G_h)(f,g)<0) **OR** (feld(G_{h-1}(2f,2g)≥ 0)) **then**
3. Fehlermeldung

4. **endif**

5. **endfor**

Bemerkung: Das Feld hier ist nicht das *Farbenfeld*, sondern das Feld der *Feldliste* des *Gitters* der jeweiligen Gitterebene h oder h-1. Es wird gesehen, dass auf Ebene h ein Punkt und darunter kein Punkt vorliegt.

Die Wichtigkeit des Tests

Wichtig war dieser Test bei der Migration der Gitterpunkte, also beim Lastausgleich, insbesondere beim Packen und Entpacken der Struktur *ev2*.

Ein zweiter Test für ev2

Tiefste Gitterpunkte sind genau die Punkte, unter denen sich kein feinerer Gitterpunkt befindet. Bei dem zweiten Test für ev2 wird für alle tiefsten Gitterpunkte geprüft, ob sie in ev2 liegen.

A.1.6 Der Vergleich der Werte

Die Aufgabe des Tests

Es geht bei diesem Test um die Kontrolle der Werte der Variablen wie u, r, Iu usw.. Es soll der jeweils erste abweichende Wert gefunden werden. Dieser Vergleich ist kein strenger Test, da wegen Randinterpolation in Ecken sowie ggf. bei selbstadaptiver Verfeinerung, wenn EP-Einträge zur Überbrückung von Lücken hinzugefügt werden, sequentielle und parallele Werte nicht zwingend exakt übereinstimmen müssen.

Die Realisation des Tests

An dieser Stelle wird das Prinzip dazu erklärt: Man durchläuft den Zyklus und gibt nach jeder Operation die Gitterwerte auf der berechneten Ebene in eine Datei aus. Das wird in Zyklus Rechnen gemacht. Neben der Angabe der jeweiligen Werte in Form einer Matrix wird durch Zahlen verzeichnet, welcher Schritt des Mehrgitterschemas jeweils gemacht wird. Ferner wird angegeben, welche Variable u, r usw. jeweils untersucht wird, ob man sich im Zyklus auf oder ab bewegt, in welcher Iteration und auf welcher Gitterebene.
Auf diese Weise werden alle relevanten Berechnungen durch Angabe der Stelle im Programm sowie der Werte der Matrix zu diesen Werten und der ausgewählten Variablen kontrollierbar.

Dazu wurde dann ein kleines Auswertungsprogramm geschrieben, mit dem diese Protokolldaten für den sequentiellen Fall mit dem parallelen Fall verglichen werden. Es wird im Falle einer Abweichung direkt die Iterationsnummer, die Gitterebene, der Typ von Operation sowie die Variable und die Stelle der Abweichung in der Matrix ausgegeben.

Die Wichtigkeit des Tests

Am Anfang war dieser Test durchaus wichtig, um das Programm korrekt ablaufen zu lassen. Ohne einen solchen Test muss man nämlich 'von Hand' die ersten abweichenden Werte finden.
Bei dieser Version der Shallow Water Equations wurde dieser Test nicht angewendet, da die Werte dort sowieso nicht genutzt werden können, und der Mehrgitter-Zyklus selber ja schon mit der Poisson-Gleichung geprüft worden war.

A.1.7 Test gemeinsamer Punkte auf gleiche Farbe

Ein weniger wichtiger Test: Hier können kaum Fehler auftreten.

Der Name des Tests

Farbtest gemeinsamer Koordinaten

Die Aufgabe des Tests

Überprüfung, ob die Grundannahme, dass Punkte mit denselben reellen Koordinaten, d.h. die Koordinaten (f,g) auf dem gröberen Gitter im Vergleich mit (2f,2g) auf dem feineren Gitter, auf demselben Kern liegen.

Die Realisierung des Tests

Algorithmus 5: Farbtest gemeinsamer Koordinaten

1. **for** h=0 **to** h_{gesamt}-2
2. **for all** (f,g)$\in G_h$
3. **if**(f,g)=(2k,2l) **then**
4. **if**((k,l) $\notin G_{(h+1)}$) **then** Fehlermeldung **endif**
5. **endif**
6. **end for**
7. **end for**

Bemerkung: Der Test '**if** ((k,l) $\notin G_{(h+1)}$)' geschieht über die Feldliste der Struktur Gitter.

A.2 Aufrufe des Programms

Dieser Abschnitt ist gedacht für den Anwender des Programms.

A.2.1 Compilierungen von Präprozessor-Direktiven

Mit der Compilierung '#ifdef SWE' werden Shallow Water Equations des Mehrgittersystems aktiviert. Wird das auskommentiert, dann wird die Poissongleichung gewählt.

A.2.2 Compilierungen von Parametern

Ein Teil der Parameter ist nicht durch die Kommandozeile änderbar. Für Parameter muss dann, nachdem sie im Quelltext gesetzt wurden, compiliert werden.

A.2.3 Der Aufruf von Parametern über die Kommandozeile

Das sind die Parameter, die dem Programm beim Aufruf mitgegeben werden. Sie werden hier der Reihe nach aufgeführt.
Zuerst wird angegeben, wie der Aufruf in der Kommandozeile aussieht. Dann wird angegeben, welche Variable im Programm in der Klasse *Aufrufe* dafür steht. Dann wird der Zweck der Variablen angegeben. Am Ende wird angegeben, welche Werte die jeweilige Variable annehmen kann.

Die Parameter m und n

1. Der Aufruf erfolgt durch eine Angabe wie 'n 513 m 513'.

2. Die Variablen lauten 'm' und 'n'.

3. Sie geben an, wie groß die Matrizen der Variablen und Klassen (z.B. *Farbenfeld*) der größten Verfeinerungsstufe sein sollen.

4. Zu den Bedingungen an die Werte: Über die Formeln n(h+1) = $\frac{1}{2}$ (n(h)+1) und m(h+1) = $\frac{1}{2}$ (m(h)+1) werden die Größen der Gitter der Ebenen darüber bestimmt. Diese Werte müssen immer ganzzahlig sein, weshalb man für n und m bei h_{gesamt} vielen Gittern nur Zahlen der Form a $2^{h_{gesamt}} + 1$ wählen kann.

Die Iterationszahlen

1. Der Aufruf erfolgt z.B. durch 'iterationen 10'.

2. Die Variable dazu lautet 'iterationsgesamtzahl'.

3. Dieser Parameter gibt an, wie viele Iterationen das Programm durchläuft.

4. Es gibt hier keine Einschränkung, aber höhere Zahlen sind hier eher dann nötig, wenn man bei der Effizienz auch die Anfangs- und Endzeit des Algorithmus misst. Natürlich muss in diese Überlegung einbezogen werden, dass der Lastausgleich der ersten Iteration für die Anfangsaufteilung auch relativ viel Zeit in Anspruch nehmen kann. Das wird nämlich immer mitgemessen.

Die Iterationstypen

1. Der Aufruf erfolgt z.B. durch 'verfeinerungstyp 1'

2. Die Variable dazu lautet verfeinerungstyp.

3. Der Parameter gibt an, ob eine Simulation durchgeführt wird oder ob und welche selbstadaptive Verfeinerung erfolgt.

4. Zulässig sind hier die Zahlen 1, 3 und 4. Dabei wird die Zahl 1 gewählt, wenn die Verfeinerungsgitter einer vorgegebenen Bewegung eines oder zweier Gitter folgen sollen. Man muss dann als Unterauswahl die Simulationsnummer - siehe unten - sowie den Wert für 'vgitter' angeben für die Größe des Gitters bei der Simulation 'Eingitter'. Die Größe 'vgitter' wird nicht bei allen Simulationen verwendet.
   Der Wert 'iterationstyp 3' ist aufzurufen, wenn das selbstadaptive Kriterium der Mehrgitterexperten gewählt wird. Simulationsnummer und Gittergröße des Verfeinerungsgitters brauchen hier nicht angegeben zu werden.
   Der Wert 'iterationstyp 4' wird bei dem vom Autor entwickelten Kriterium gewählt.

Bemerkung 1: Bei den Iterationstypen 3 und 4 werden standardmäßig die Funktionswerte neu geschrieben, wobei dabei der Parameter t_konst angibt, wie stark sich die Funktionswerte im Laufe der Zeit ändern.

Bemerkung 2: Mit 'verf-krit' werden dem Programm die Gewichte der Verfeinerungsungleichungen übergeben.

Die Simulationsnummer

1. Der Aufruf erfolgt z.B. durch 'simulationsnummer 1'.
2. Die Variable dazu lautet simulationsnummer.
3. Beim Verfeinerungstyp 1 (Simulation) gibt diese Variable an, welche Art Simulation durchgeführt werden soll.
4. Es werden folgende Werte gewählt:
   (a) EINGITTER (0): Ein bewegtes Verfeinerungsrechteck wird gewählt. Die Gittergröße wird durch den Parameter 'vgitter' mit den Werten 0 bis 10 ausgewählt.
   (b) AUFSCHLAG (1): Ein Verfeinerungsrechteck stößt auf ein stationäres Verfeinerungsrechteck.
   (c) ZUSAMMENSTOß (2): Zwei Verfeinerungsrechtecke laufen gegeneinander.
   (d) EINGITTER_GROß (3): Ein großes Verfeinerungsgitter wird bewegt.
   (e) KOMPLETT (4): Das ganze Verfeinerungsgitter wird gewählt.
   (f) EINGITTER_BESCHRÄNKT (5): Wie Eingitter, nur dass die Bewegung ab einer gewissen Iterationszahl aufhört.
   (g) EINGITTER_UNBEWEGT (6): Ein stationäres Verfeinerungsrechteck liegt vor.

Der Parameter 'vgitter'

1. Der Aufruf lautet z.B. 'vgitter 0'.
2. Die Variable lautet uebergebe_vgitter.
3. Die Bedeutung: Bei einer Simulation mit der Auswahl EINGITTER oder EINGITTER_BESCHRÄNKT wird die Größe des einen Gitters hiermit festgelegt.

4. Es werden die Zahlen 0 bis 10 angegeben; unter den Zahlen 0-2 steht 0 dabei für das kleinste, 2 für das größte Verfeinerungsgitter.

Die Parameter h_{min} und h_{max} und h_{cap}

1. Die Aufrufe lauten 'h_min 1', 'h_max 2' sowie 'h_cap -1'.

2. Die Variablen lauten h_min, h_max und h_cap

3. Sie geben die Anzahl der Verfeinerungsebenen, der Vergröberungsebenen sowie der 'cap-Ebene' an.

4. Getestet sind die Parameter bis $h_{min} \leq 2$, und $h_{max} \leq 3$. Der Parameter h_{cap} muss so gewählt werden, dass er im Bereich h_{min+1} bis h_{gesamt}-1 liegt. Es gibt ferner noch die Bedingung an das 'Experten-Kriterium' (iterationstyp 3): Hier darf h_{min} nicht > 1 sein, weil die Berechunng des Kriteriums über die Werte des FAS geschieht. Warum das nicht geht:
   Nehmen wir den Fall $h_{min} = 2$. Dann ist das Verfeinerungsgitter auf Ebene 1 vor der adaptiven Verfeinerung allgemein anders als nach der Verfeinerung. Aber die zugehörigen FAS-Werte auf Ebene 2 sind nur für das Gitter vor der Verfeinerung gegeben. Für die neuen Gitterpunkte existieren die FAS-Werte nicht. Dann werden zwar Verfeinerungsgebiete berechnet, sie basieren aber nicht auf den FAS-Werten!

Der Parameter t_konst

1. Der Aufruf erfolgt z.B. mit t_konst 1.0

2. Die Variable lautet t_konst

3. Das Ausmaß der Funktionswerteänderung bei den Verfeinerungstypen 3 und 4 wird festgelegt.

4. Der Parameter ist frei wählbar.
   Bemerkung: Soll keine Funktionswerteänderung erfolgen, so ist in der Klasse *Gitter* in der Funktion *schreibe_f_werte* die Variable t_d auf 0 zu setzen, oder man definiert die Präprozessorvariable T_D_GLEICH_0

Die Parameter μ_1 und μ_2

1. Der Aufruf sieht so aus: 'mu_1 2' 'mu_2 2'.

2. Die Variablen lauten mue1 und mue2.

3. Diese Parameter stehen für die Anzahl an Vor- bzw. Nachglättungen.

4. Diese Parameter können frei gewählt werden. Zu hohe Werte machen aber aus numerischer Sicht keinen Sinn, denn sie kosten Zeit und können die Konvergenz nur bedingt bewirken.

Der Parameter Punktlast

1. Der Aufruf lautet z.B. 'PL 1'.

2. Die Variable dazu lautet wie_viel_punktlast.

3. Man kann zum Ausgleich, dass die Poisson-Gleichung selber kaum Last bereitstellt, eine Punktlast bei den Glättern hinzufügen.

4. Der Parameter Punktlast kann frei gewählt werden. Im Fall PL 0 wird keine Punktlast verwendet.

Der Ausdruck 'adapt'

1. Der Aufruf lautet z.B. 'adapt'.

2. Die Variable dazu lautet oben_nicht_adaptiv

3. Standardmäßig wird nur im adaptiven Bereich ein FAS basiertes Verfahren berechnet, hingegen auf den gröberen Ebenen ein 'normales' Mehrgitterschema.

4. Soll das adaptive Schema auf allen Gitterebenen gemacht werden, gibt man in der Kommandozeile den Parameter 'adapt' an. Sonst macht man keine Angaben in der Kommandozeile.

Ausschalten des Balancers

1. Das Kommando zur Deaktivierung des Balancers in der Kommandozeile nennt sich 'baloff'.

2. Die zugehörige Variable nennt sich balancen.

3. Zu Testzwecken wird der Lastausgleich komplett deaktiviert.

4. Man fügt 'baloff' in die Kommandozeile ein oder tut das nicht.

Aktivieren der Lastwertekorrektur

1. Die Aufrufe nennen sich LwK bzw. lwk_typ 2.

2. Die Variablen dazu lauten lastwerte_korrektur bzw. lwk_typ.

3. Man kann die Lastwertekorrektur aktivieren oder deaktivieren. Falls sie aktiviert wird gilt es, unter 4 Fällen auszuwählen.

4. Die Lastwertekorrektur, vgl. die Abschnitte 6.7.2 und 6.9, wird mit dem Kommando 'LwK' aktiviert. Man hat dann noch die Variable lwk_typ, für die die Zahlenwerte im Folgenden angegeben werden. Man unterscheidet zwischen IMMER_NUR_VOLLPUNKT (Zahl 0), wenn Punkte ohne Last nicht aufgenommen

werden in die Liste der zu verschickenden Punkte, und IMMER_AUCH_NULLPUNKT(2), wenn auch Punkte der Last 0 verschickt werden sollen, was die bessere Strategie ist. Wenn beides gemischt werden soll, dann erfolgt entweder eine häufigere Verwendung der VOLLPUNKT-Variante oder der NULLPUNKT-Variante. Über eine Variable lwk_intervall_laenge wird angegeben, wie stark gewichtet wird. Je größer die Zahl, desto stärker die Gewichtung. IMMER_AUCH_NULLPUNKT ist dabei der am geeignetste Wert und ist damit der default-Wert. Es werden weitere Zahlenzuordnungen für den lwk_typ angegeben: HAUPTSÄCHLICH_NUR_VOLLPUNKT(1) und HAUPTSÄCHLICH_AUCH_NULLPUNKT(3).

Lastausgleich festlegen

PLB oder PLB-Variante

1. Der Aufruf lautet 'echtes-PLB'.
2. Die Variable dazu lautet echtes_plb.
3. Es geht darum, ob PLB oder die PLB-Variante durchgeführt werden.
4. Wird 'echtes-PLB' in der Kommandozeile aufgerufen, dann wird PLB umgesetzt, ansonsten die PLB-Variante. PLB wird aber erst ab einer Gesamtanzahl von über 4 Kernen gemacht, weil bis dahin die Methoden im Wesentlichen gleich sind, da nur maximal 2 Kerne in Ost-West- oder Nord-Süd-Richtung liegen.

Anzahl der Lastausgleichsrunden

1. Der Aufruf lautet 'gL'.
2. Die Variable lautet gL.
3. Es geht darum, in der ersten Iteration einen vollständigen Lastausgleich zu garantieren.
4. Wird 'gL' aufgerufen, so wird die Variable gL auf *true* gesetzt. Dann geschieht in der ersten Iteration ein Lastausgleich über so viele Lastausgleichsrunden wie nötig ist für einen vollständigen Lastausgleich. Ansonsten geschieht immer nur eine Lastausgleichsrunde.

Reverser Lastausgleich

1. Der Aufruf erfolgt mit 'mit' oder 'ohne'.
2. Die Variable lautet mit_ohne.
3. Es wird festgelegt, ob ein reverser Lastausgleich erfolgt.
4. Bei 'mit' erfolgt ein reverser Lastausgleich, bei 'ohne' nicht, d.h. bei 'ohne' geschieht der Lastausgleich nur auf einer Ebene. Daher ist 'mit' der default-Wert.

Die Lastinformationsbeschaffung bei der PLB-Variante

1. Der Aufruf erfolgt durch 'plb_kn 0/1' sowie 'matrix_vektor 0/1'.

2. Die Variablen lauten plb_kn und matrix_vektor.

3. Es wird festgelegt, ob der Lastausgleich über einen vollständigen Graphen oder entsprechend Abschnitt 6.2.3 erfolgt. Ferner wird festgelegt, ob die Lastinformationen getrennt für Ost-West und Nord-Süd bestimmt werden (Vektor) oder für alle Kerne (Matrix).

4. plb_kn = 1 steht für den vollständigen Graphen, plb_kn = 0 für den anderen Fall der Lastübertragung über einen asymetrischen Baum.
'matrix_vektor = 0' steht für Vektor, 'matrix_vektor = 1' für Matrix.

Den Strategie-Typ festlegen

1. Der Aufruf erfolgt mit 'ev_typen 0' bis 'ev_typen 4'.

2. Die Variable lautet ev_typen.

3. Hier wird festgelegt, wo ev bzw. ev2 verwendet werden, und auf wie vielen Ebenen.

4. Man setzt ev_typen, falls nur *ev* verwendet wird, auf TYP_EV (0) oder TYP_EVEV (2). Wenn nur *ev2* verwendet werden soll, verwendet man TYP_EV2 (1) oder TYP_EV2EV2 (4).
Folgende Restriktion gilt für die Parameter: Man bemerke, wenn *ev2* verwendet wird, dass dann die Lastausgleichsvariable mit_ohne auf *true* gesetzt werden muss, damit ein doppelter Lastausgleich erfolgt.

   Die doppelte Verwendung von EV/EV2 nach 'TYP_' geschieht dann, wenn der Lastausgleich bzgl. 2 Gitterebenen erfolgt. Daher ist der Standard TYP_EV für den normalen und TYP_EV2EV2 für den reversen Lastausgleich.

Der Parameter 'Vektorisieren'

1. Der Aufruf ist 'vektorisieren'.

2. Die Variable lautet vektorisieren.

3. Es wird festgelegt, ob der Rand des Partnerkerns durch eine Übertragung von Vektoren erfolgt (vektorisieren = *true*), oder durch eine Sortierung der Ränder (vektorisieren = *false*).

4. Der default-Wert ist *false*. Bei Angabe von vektorisieren in der Kommandozeile wird der Wert auf *true* gesetzt.

Ausgabe von gnuplot-Darstellungen

Hier werden 4 Variablen verwendet:

1. Die Aufrufe sind z.B. 'gp_ausgabe, gp_ausgabe_mod 3, gp_mit_io, gp_mit_aufteilung'
2. Die Variablen lauten gp_ausgabe, gp_ausgabe_mod, gp_mit_io, gp_mit_aufteilung
3. Die Variablen werden so gewählt, dass eine Ausgabe der Werte und der Gitteraufteilung möglich wird. Die Werte können auch teilweise unterdrückt werden, vgl. Punkt 4.
4. Für eine Ausgabe der Werte werden gp_ausgabe und gp_mit_io auf *true* gesetzt, bzw. in der Kommandozeile angegeben. gp_ausgabe_mod gibt an, nach wie vielen Schritten die nächste Ausgabe erfolgt. Und gp_mit_aufteilung = *true*, bzw. der Aufruf von gp_mit_aufteilung in der Kommandozeile bewirkt eine Ausgabe der Gitteraufteilung im Anschluss an den Programmdurchlauf.

   Bemerkung 1: Das alles muss bei Messungen deaktiviert sein!

   Bemerkung 2: Wird in der Kommandozeile der Ausdruck 'visualisieren' angegeben, dann werden obige Variablen so gesetzt, dass die Visualisierungsdateien erstellt werden und die Visualisierung durchgeführt wird.

   Bemerkung 3: Wird auf CHEOPS gerechnet, dann werden nur die für die Visualisierung nötigen Dateien geschrieben. Für die eigentliche Visualisierung muss dann noch 'gnuplot commandfile' für die Darstellung der Werte bzw. 'gnuplot commandfile2' für die Darstellung der Lastverteilung aufgerufen werden.

A.2.4 Unmögliche Parameterkombinationen

Sie werden hier der Reihe nach aufgelistet:

1. Bei den iterationstyp = 3 darf h_min höchstens 1 sein.
2. (a) Bei der Compilierung von #define SWE dürfen die iterationstypen 3 und 4 nicht gewählt werden.
   (b) Bei dieser Compilierung ist eine Visualisierung und eine cap ausgeschlossen.

Anhang B

Theoretische Fakten

B.1 Sortierung

Die hier vorliegende Sortierung basiert auf der Sortierung sortierter Listen und benötigt dafür O(n⌈(log(n))⌉) viel Zeit.
Man nennt das Merge-Sortierung.

B.1.1 Funktionsweise

Zwei aufsteigend sortierte Mengen der Längen k und l werden zu einer gemeinsam aufsteigenden Menge der Länge k+l sortiert. Das geht einfach: Man hält zwei Zeiger auf die kleinsten Elemente und trägt das jeweils kleinere in die neue Liste ein, wobei deren Index inkrementiert wird. Man macht dieses, bis der Zeiger auf das Ende einer Liste zeigt. Dann werden die verbliebenen Elemente der anderen Liste hinten angehängt.

Auf diese Weise werden erst zweielementige, dann 4-, dann 8-, dann 16- usw. elementige Teilmengen entstehen. Jede dieser Sortierungen dauert n Schritte, wenn n die Gesamtanzahl zu sortierender Werte ist. Nach n ⌈(log(n))⌉ vielen Schritten hat man die sortierte Liste als Ergebnis.

B.2 Graphen

Die Struktur (V,E) mit den Knoten V und Kanten E $\subseteq$ V x V wird Graph genannt.

B.3 Geometrische Reihe

Die trivialerweise zu berechnende Formel lautet:

$$\sum_{i=0}^{k} q^i = \frac{1-q^{k+1}}{1-q}$$

B.4 Die O - Notation

Für eine Funktion g:$\mathbb{N} \to \mathbb{R}$ gilt g = O(f(n)), wenn zwei Konstanten c und n_0 existieren mit $\forall$ n $\geq$ n_0 g(n) $\leq$ c f(n). Diese Notation wird zur Angabe des Laufzeitverhaltens eines Algorithmus benötigt.

Anhang C

Algorithmen

Algorithmus C.1: *P^+-hinzu-erfüllt-MaxG*
Input: Ein Punkt (f,g)
Output: Der Punkt wird hinzugefügt mit Hilfe von *P^+-hinzu* oder ignoriert.

1. ff = $\frac{f}{2}$ und gg = $\frac{g}{2}$
2. **if**(ggff(ff,gg) < 0) **then**
3. **return**
4. **if**(g **AND** 1 = 1) **then**
5. **if**(gg = gg_{max}) **then**
6. **return**
7. **if**(ggff(ff,gg+1) < 0) **then**
8. **return**
9. **end if**
10. **if**(f **AND** 1 = 1) **then**
11. **if**(ff = ff_{max}) **then**
12. **return**
13. **if**(ggff(ff+1,gg) < 0) **then**
14. **return**
15. **end if**
16. $P_k^+ = P_k^+ \cup \{(f,g)\}$

Bemerkung: Zu den Bezeichnungen: Mit ggff ist das *Grobgitterfarbenfeld* gemeint. Die Funktion *return* beendet den Algorithmus. Und: gg_{max} ist die maximal mögliche Grobgitter-x-Koordinate, ff_{max} ist die maximal mögliche Grobgitter-y-Koordinate.

(f,g-1) (f,g)(f,g+1)

Abbildung C.1: Zur ersten MaxG-Methode: Wenn der rechte Punkt (f,g+1) nicht existiert, dann kann der mittlere Punkt (f,g) auch nicht existieren. Die umrandeten Punkte sind gemeinsame Punkte.

Warum diese Funktion funktioniert: Durch die ggff(ff,gg)-Abfrage aus Zeile 2 wird geprüft, dass das Gitter nicht im Norden oder Westen um einen ganzen Block übersteht. Denn dann ist dort ggff(ff,gg) < 0. Die restlichen Abfragen bewirken, dass im Osten und Süden die Randlinie nicht um einen Feingitterpunkt übersteht. Wenn nämlich (g AND 1=1) und ggff(ff,gg+1) < 0 sind, dann ist (f,g+1) nicht mehr existent wegen der ggff Bedingung, und (f,g) darf es daher auch nicht sein wegen der (g AND 1=1) - Bedingung, vgl. Abbildung C.1. Denn ggff(ff,gg) ≥ 0 reicht nur für die Existenz von (f,g-1), weil (f,g+1) nicht existiert.

Algorithmus C.2: P_k^+-hinzu für MaxG Version 1
Input: (f,g)
Output: (f,g) wird zu P_k^+ hinzugefügt oder nicht

1. **if** fgff(f,g) < 0 **then**
2. **return**
3. *P^+-hinzu-erfüllt-MaxG(f,g)*, also Algorithmus C.1 anwenden

Ich versichere, dass ich die von mir vorgelegte Dissertation selbstständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; dass diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; dass sie - abgesehen von unten angegebenen Teilpublikationen - noch nicht veröffentlicht worden ist sowie, dass ich eine solche Veröffentlichung vor Abschluss des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen der Promotionsordnung sind mir bekannt.
Die von mir vorgelegte Dissertation ist von Herrn Prof. Dr. Ewald Speckenmeyer betreut worden.

Lebenslauf Ekkehart Kiparski

Geburtsdatum:	20. Juni 1970
Geburtsort:	Waldbröl
Eltern:	Reinhold Kiparski und Brunhilde Kiparski, geb. Hesse
Staatsangehörigkeit:	deutsch

Schulausbildung

1989	Wüllenweber-Gynasium Bergneustadt, Abschluss: Abitur

Studium

WS 89/90 - WS 95/96	Studium der Mathematik mit Nebenfach Physik an der Universität Köln
Januar 96	verliehen Diplom Mathematik
SS 96	Eingeschrieben für Promotion Mathematik an der Universität Köln
WS 96/97 - WS 04/05	Studium der Betriebswirtschaftslehre an der Universität zu Köln
Mai 05	verliehen Diplom Kaufmann
ab 97	Promotionsstudium in Informatik an der Universität zu Köln bei Herrn Prof. Dr. E. Speckenmeyer
97 - 00	Stipendiat des Graduiertenkollegs 'Scientific Computing' der Universität zu Köln gefördert von der DFG
WS 05/06	umgeschrieben von Promotion Mathematik auf Promotion Informatik