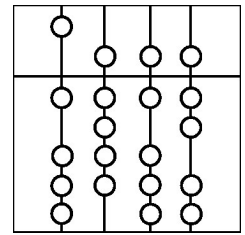


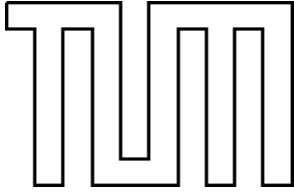
Technische Universität München
Fakultät für Informatik



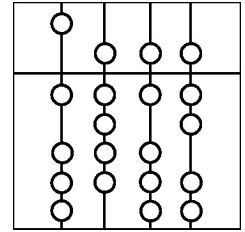
Parallelisierung eines Cache-optimalen 3D Finite-Element-Verfahrens

Diplomarbeit

Markus Langlotz



Technische Universität München
Fakultät für Informatik



Parallelisierung eines Cache-optimalen 3D Finite-Element-Verfahrens

Diplomarbeit

Markus Langlotz

Aufgabensteller : Univ.-Prof. Dr. Christoph Zenger
Betreuer : Markus Pögl

Abgabedatum : 15. Dezember 2004

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Dezember 2004

Markus Langlotz

Inhaltsverzeichnis

1	Einleitung	3
1.1	Aufgabenstellung	4
1.2	Danksagung	5
1.3	Überblick über das Dokument	5
2	Mathematische Grundlagen	6
2.1	Numerisches Lösen von Differentialgleichungen	6
2.2	Finite-Element-Methode	8
2.2.1	Schwache Formulierung	10
2.2.2	Lösungsapproximation	10
2.2.3	Assemblierung der Steifigkeitsmatrix	11
2.3	Lösung linearer Gleichungssysteme	13
2.4	Hierarchische Basen	15
2.5	Mehrgitterverfahren	21
2.6	Raumfüllende Kurven	24
2.6.1	Konstruktion von raumfüllenden Kurven	25
2.6.2	Hölder-Stetigkeit	26
2.6.3	Bestimmung der Oberfläche von Peanokurvenabschnitten	28
3	Ausgangssituation	30
3.1	Der sequentielle Algorithmus	30
3.1.1	Numerik	30
3.1.2	Geometrie und Gitterbeschreibung	32
3.1.3	Datenzugriff und -verwaltung	33
3.1.4	Gebietstraversierung	36
3.2	Grundlagen von MPI	38
4	Parallelisierung	41
4.1	Effizienz paralleler Programme	41
4.2	Partitionierung	42
4.2.1	Theoretischer Ansatz	42
4.2.2	Praktische Umsetzung	46

4.3	Kommunikation	55
4.3.1	Theoretischer Ansatz	55
4.3.2	Praktische Umsetzung	56
4.4	Berechnung des Platzbedarfs	65
4.5	Initialisierung	67
5	Ergebnisse	69
5.1	Laufzeit	69
5.1.1	Sequentielle Effizienz	70
5.1.2	Parallele Effizienz	72
5.2	Speicherverbrauch	75
5.3	Cache-Effizienz	77
6	Zusammenfassung und Ausblick	79
6.1	Ergebnisbewertung	79
6.2	Ausblick	79
	Literatur	81
A	Abschätzung der Oberfläche eines diskreten Peanokurven-	
	abschnittes	83

Kapitel 1

Einleitung

Wir leben in einer Zeit, in der sich Großprojekte ständig in ihrem Umfang übertreffen. Seien es die höchsten Wolkenkratzer, Baumaßnahmen für olympische Spiele oder gigantische Kraftwerke, ständig werden größere Geldmengen und mehr menschliche Arbeitskraft benötigt, um diese Projekte umzusetzen. Einen ganz wesentlichen Kostenfaktor stellt die Einhaltung von sicherheitsrelevanten Anforderungen dar. Dabei müssen sehr grundsätzliche, die Bausweise betreffende Fragen gestellt werden. Wie groß sind die Kräfte, die bei einem Jahrhundertsturm auf das Bauwerk einwirken und an welcher Stelle setzen diese Kräfte an?

Für die Beantwortung dieser Fragen gibt es prinzipiell zwei Möglichkeiten. Entweder mit Hilfe eines Prototyps, der den entsprechenden Umgebungsbedingungen ausgesetzt wird, oder mit Hilfe von mathematischen Berechnungen. Betrachtet man die eingangs gewählten Beispiele, so wird schnell klar, dass die Erstellung eines Prototyps sehr aufwändig und die Simulation eines Jahrhundertsturms, sowie die Messung der auftretenden Kräfte äußerst kompliziert bzw. unmöglich wäre. Deshalb weicht man zunehmend darauf aus, die aufgeworfenen Fragen mit Hilfe von numerischen Simulationen zu beantworten. Dies gilt sowohl für große Bauvorhaben, als auch beispielsweise im Maschinen- und Fahrzeugbau sowie der Chemie.

Die notwendigen numerischen Berechnungen werden unter Einsatz von zunehmend mächtigeren Rechnern durchgeführt. Dabei zeigt sich, dass die Effizienz der verwendeten Programme (bzw. Algorithmen) von vielen verschiedenen Faktoren abhängt. Neben der reinen Rechenleistung der CPU, ist die Größe des Arbeitsspeichers und der Caches von Bedeutung. Hinzu kommt, dass die gute Ausnutzung der Caches außerdem von der Struktur abhängt, mit der die Algorithmen auf den Speicher zugreifen. Besonders ungünstig ist es, in schneller Abfolge auf weit voneinander entfernte Speicherzellen zuzugreifen [Lan04]. In einem solchen Fall ist es sehr unwahrscheinlich, dass die benötigten Speicherzellen im schnellen Cache liegen, weswegen sie aus dem mindestens zehnfach langsameren Arbeitsspeicher geholt werden müssen. In

einem solchen Fall spricht man von einem *Cache Miss*. Das Verhältnis von Cache Misses zu Speicherzugriffen nennt man *Cache Miss Rate* und entsprechend das Verhältnis der erfolgreichen Cachezugriffe *Cache Hit Rate*. Da die gesamte Ausführungszeit einer Berechnung stark von der Cache Hit Rate abhängt, werden numerische Löser oftmals nach ihrer eigentlichen Fertigstellung hinsichtlich effizienter Nutzung der Caches optimiert.

Caches arbeiten genau dann effizient, wenn der Zugriff auf die Daten im Hauptspeicher so stattfindet, dass die selben Daten häufig benutzt (zeitliche Lokalität) und räumlich nah beieinander liegende Daten kurz aufeinander (räumliche Lokalität) verwendet werden. Die Hauptidee der Arbeiten von Markus Pögl [Pög04] und Frank Günther [Gün04] lag darin, einen Löser für numerische Probleme zu entwerfen, der den Datenzugriff und die Datenspeicherung so organisiert, dass die Eigenschaften eines Caches gut ausgenutzt werden. Dabei sollte schon die Natur des Algorithmus' an sich verantwortlich für die hohe Cacheeffizienz sein. Durch die ausschließliche Speicherung der Daten auf Kellern und die dadurch induzierte lineare Abarbeitung erreichten beide unabhängig voneinander Cache Hit Rates von $> 99\%$. Dieses Verhalten beziehungsweise die Herangehensweise, den Algorithmus hinsichtlich Cachenutzung zu optimieren, wird Cache Awareness genannt. Wegen der erreichten sehr hohen Cache Hit Rates werden solche Algorithmen auch als Cache-optimal bezeichnet.

1.1 Aufgabenstellung

Selbst ein numerischer Algorithmus, der mit sehr geringem Organisationsaufwand und optimaler Ausnutzung der Rechnerhardware implementiert wurde, ist durch die Rechenleistung der aktuell schnellsten CPUs und durch die Größe üblicher Arbeitsspeicher in seiner Leistungsfähigkeit limitiert. Die einzige Möglichkeit, die Berechnungen zu beschleunigen, besteht in der Parallelisierung der Algorithmen. Auf diese Art und Weise kann nicht nur die Rechenleistung von mehreren hundert CPUs sondern auch der Arbeitsspeicher von mehreren hundert Rechnern genutzt werden. Deshalb war es das Ziel der vorliegenden Arbeit, mittels Parallelisierung diese Möglichkeiten für den von Markus Pögl entwickelten dreidimensionalen Löser zu eröffnen.

Aufgabe war, den von ihm implementierten Algorithmus so zu erweitern, dass er auf einem Linux-Cluster lauffähig ist. Wegen der großen Verbreitung und der guten Performance sollte für die Datenkommunikation MPI verwendet werden. Aufgrund der Tatsache, dass der zu parallelisierende Algorithmus auf neuesten Konzepten basiert, war zunächst nicht klar, ob er für eine Parallelisierung geeignet ist. Aus diesem Grund diente die vorliegende Arbeit als Proof of Concept. Es sollte also die Machbarkeit der Parallelisierung gezeigt werden, verbunden mit der Anforderung, die hohe Cacheeffizienz zu erhalten. Darüberhinaus war es natürlich wünschenswert, eine hohe paral-

lele Effizienz, also eine deutliche Verringerung der Rechenzeit beim Einsatz mehrerer Prozessoren zu erreichen. Um die Plattformunabhängigkeit von Markus Pögl's Implementierung zu erhalten, sollten nur verbreitete Bibliotheken verwendet und in ANSI C programmiert werden.

1.2 Danksagung

Die Erstellung einer erfolgreichen Diplomarbeit hängt nicht in erster Linie von der Motivation des Erstellers oder dessen Leistungsfähigkeit ab. Vielmehr ist das gesamte Umfeld, das zur Erstellung beigetragen hat, von sehr großer Bedeutung. Besonderen Dank für das interessante Thema und die Möglichkeit, ohne Einschränkungen meine eigenen Vorstellungen und Ideen einzubringen, gebührt meinem Betreuer Markus Pögl. Desweiteren möchte ich den Mitarbeitern des Lehrstuhls von Professor Zenger meinen Dank aussprechen, die mir neben einer Konferenzteilnahme und vieler Nachhilfestunden auch eine äußerst angenehme Arbeitsatmosphäre boten. Für seine Offenheit und Hilfsbereitschaft bei Fragen aller Art bedanke ich mich dabei besonders bei Andreas Krahnke. Die zahlreichen Ideen und Anregungen von Professor Dr. Zenger haben einen großen Beitrag zur Umsetzung der Parallelisierung geleistet. Seiner Motivationsfähigkeit verdanke ich es, dass ich die verbrachte Zeit nie als Pflicht sondern vielmehr als interessante Projektarbeit aufgefasst habe. Nicht zuletzt möchte ich mich bei Veronika Ortner für ihr gewissenhaftes Probelesen und für die Hilfe bei sämtlichen Formulierungsproblemen bedanken. Hinzu kommt, dass sie sowie meine Familie immer Verständnis für meinen Arbeitseifer aufbrachten und mir auf diese Weise viele Dinge erleichterten.

1.3 Überblick über das Dokument

Das vorliegende Dokument beschäftigt sich, bevor das Vorgehen bei der Parallelisierung beschrieben wird, mit den dazu notwendigen Grundlagen. In Kapitel 2 werden dazu die mathematischen Konzepte zusammengefasst. Anschließend geht Kapitel 3 auf die Ausgangssituation, also den Aufbau und die verwendeten Techniken des zu parallelisierenden Lösers von Markus Pögl ein. Daran anschließend werden die zur Parallelisierung notwendigen Schritte in Kapitel 4 motiviert und erläutert. Die erzielten Ergebnisse werden in Kapitel 5 dargestellt, bevor Kapitel 6 einen Ausblick über die sich anschließenden Projekte gibt.

Kapitel 2

Mathematische Grundlagen

Die Parallelisierung eines numerischen Löser hat zum Ziel, einen vorhandenen sequentiellen Löser durch die Verteilung auf mehrere Prozessoren zu beschleunigen. Entscheidend dabei ist, dass die numerischen Ergebnisse des parallelen Programms bis auf Maschinengenauigkeit mit denen des ursprünglichen Löser übereinstimmen müssen. Da dazu Kenntnisse der grundlegenden mathematischen Ideen nötig sind, werden diese in den folgenden Abschnitten kurz zusammengefasst. Abschnitt 2.1 beschreibt ganz allgemein das Vorgehen beim Lösen einer Differentialgleichung, was eine der häufigsten Aufgaben der Numerik darstellt. Abschnitt 2.2 beschreibt mit der Finite-Element-Methode ein sehr flexibles Verfahren zur Diskretisierung einer Differentialgleichung. Der darauf folgende Abschnitt 2.3 beschäftigt sich mit dem eigentlichen Lösungsprozess, bei dem so genannte iterative Gleichungssystemlöser zum Einsatz kommen. Die beiden anschließenden Kapitel 2.4 und 2.5 zeigen kurz Verfahren auf, mit deren Hilfe man die Effizienz solcher Löser deutlich steigern kann. Der Teil Abschnitt 2.6 behandelt die Grundlagen von raumfüllenden Kurven, die im Rahmen der vorliegenden Arbeit sowohl in der Traversierung des Rechengebiets als auch bei der Partitionierung in Teilgebiete Verwendung finden.

2.1 Numerisches Lösen von Differentialgleichungen

Viele naturwissenschaftliche Zusammenhänge lassen sich mit Hilfe von Differentialgleichungen beschreiben. Diese stellen eine Beziehung zwischen einer Größe und ihren Ableitungen her. Eine häufig auftretende Form sind die partiellen Differentialgleichungen (PDE), bei denen Ableitungen nach mehreren Variablen existieren. Zusammen mit den Differentialgleichungen werden Anfangsbedingungen bzw. Randwerte spezifiziert. Diese wählt man gewöhnlicherweise so, dass nur noch eine eindeutige Lösung der Gleichung existiert.

Das Lösen von Differentialgleichungen ist in den meisten Fällen analytisch nicht möglich, weshalb auf numerische Näherungsverfahren ausgewichen werden muss. Dabei wird das ursprünglich kontinuierliche Problem diskretisiert und die Lösung des durch die Diskretisierung entstandenen linearen Gleichungssystems angenähert. Die Genauigkeit der Lösung und die Geschwindigkeit der Lösungsermittlung hängt dabei von den verwendeten Diskretisierungsverfahren und vom verwendeten Löser ab. An einem einfachen Beispiel sollen diese Schritte verdeutlicht werden.

Eine in vielen physikalischen Zusammenhängen auftretende Differentialgleichung ist durch

$$-u''(x) + \sigma u(x) = b(x), \quad 0 < x < 1, \quad \sigma \geq 0, \quad (2.1)$$

$$u(0) = u(1) = 0 \quad (2.2)$$

gegeben. Gleichungen (2.1) und (2.2) beschreiben die stationäre Wärmeverteilung in einem gleichmäßigen Stab. Die Temperatur an den beiden Enden des Stabes wird gleich 0 gesetzt. Obwohl dieses Randwertproblem eine analytische Lösung besitzt, wird es hier seiner Einfachheit wegen verwendet, um numerische Verfahren zu erläutern. Der erste Schritt besteht in der Diskretisierung der Gleichung (2.1). Dazu lässt sich neben der in Abschnitt 2.2 beschriebenen Finite-Element-Methode die wesentlich anschaulichere Finite-Differenz-Methode verwenden. Diese zerlegt das Gebiet $\Omega : 0 \leq x \leq 1$, wie in Abbildung 2.1 gezeigt, in n Teilintervalle der Länge $h = \frac{1}{n}$ und definiert auf dem entstandenen Gitter Ω^h an den Teilintervallgrenzen die Punkte x_j , $0 \leq j \leq n$.

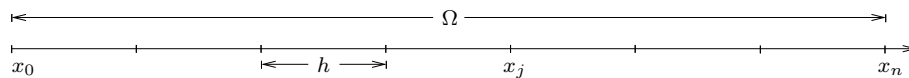


Abbildung 2.1: Diskretisierung des Gebiets Ω in $n = \frac{1}{h}$ Intervalle

An jedem dieser Punkte wird nun die ursprüngliche Differentialgleichung (2.1) durch eine Finite-Differenzen-Näherung zweiter Ordnung ersetzt, welche man mit Hilfe der Taylorentwicklung um x_j erhält. Da es sich bei den Unbekannten nicht mehr um die exakte Lösung handelt, schreiben wir v_j statt $u(x_j)$. Die gesamte Lösung wird nun durch den Vektor $\mathbf{v} = (v_1, \dots, v_n)$ repräsentiert, dessen Einträge man auch als Freiheitsgrade bezeichnet. Mit Hilfe der aus der Taylorentwicklung hergeleiteten zweiten Ableitung gilt

$$\frac{-v_{j-1} + 2v_j - v_{j+1}}{h^2} + \sigma v_j = b(x_j), \quad 1 \leq j \leq n-1, \quad (2.3)$$

$$v_0 = v_n = 0. \quad (2.4)$$

Schreibt man zusätzlich die Werte der rechten Seite an den Gitterpunkten als Vektor $\mathbf{b} = (b(x_1), \dots, b(x_n))^T = (b_1, \dots, b_n)^T$ und setzt $\alpha v_j = 2v_j + \sigma v_j h^2$,

so lassen sich diese $n - 1$ Gleichungen zu einem linearen Gleichungssystem (LGS) der Form

$$\frac{1}{h^2} \begin{pmatrix} \alpha & -1 & 0 & \cdots & 0 \\ -1 & \alpha & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & \alpha \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \cdot \\ \cdot \\ v_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \cdot \\ \cdot \\ b_n \end{pmatrix}$$

zusammenfassen. Weiter verkürzt schreibt man $A\mathbf{v} = \mathbf{b}$. Die bis hierher verwendeten Variablen, u für die gesuchte Temperaturverteilung, v für deren diskrete Näherung, A für die Matrix des zu lösenden Gleichungssystems und b für die rechte Seite der Gleichung, werden in den folgenden Abschnitten ohne erneute Einführung weiter verwendet.

Die einfachste Möglichkeit, um das so erhaltene LGS zu lösen, ist die Gauß-Elimination. Diese liefert in einer festen Anzahl von Schritten eine exakte Lösung. Geht man von einer in der Praxis häufig sehr großen Anzahl von Unbekannten aus, so stellt sich die asymptotische Laufzeit von $O(n^3)$ als sehr ungünstig heraus. Darüber hinaus zerstört die Gauß-Elimination die dünne Besetztheitsstruktur der Matrix A , was eine effiziente Speicherung unmöglich macht.

Als Alternative zur expliziten Lösung bieten sich iterative Verfahren (siehe Abschnitt 2.3) an, die für ein gegebenes LGS eine Näherungslösung bestimmen. Bei der Lösung eines Gleichungssystems $A\mathbf{u} = \mathbf{b}$ gehen diese schrittweise vor, d.h. ausgehend von einer ersten Schätzung der Lösung \mathbf{v}_0 wird diese iterativ verbessert. Die Geschwindigkeit, mit der sich eine gute Lösung einstellt, hängt dabei wesentlich vom verwendeten Iterationsverfahren ab.

2.2 Finite-Element-Methode

Die Finite-Element-Methode (FEM) wird verwendet, um bestimmte Klassen von partiellen Differentialgleichungen zu lösen. Diese so genannten elliptischen Differentialgleichungen kommen häufig bei der Beschreibung von physikalischen Gesetzmäßigkeiten vor. Da ihre exakte Lösung in den seltensten Fällen analytisch ermittelbar ist, ist die numerische Approximation von großem praktischen Nutzen.

Eine Methode zur Diskretisierung genannter Differentialgleichungen stellt die in Abschnitt 2.1 vorgestellte Methode der Finiten Differenzen (FD) dar. Ungünstigerweise treten bei dieser einfachen Herangehensweise schnell wesentliche Nachteile auf:

- Die Differenzgleichungen werden bei Verwendung von nicht äquidistant verteilten Stützstellen sehr kompliziert.

- Wir benötigen Lösungen, deren dritte Ableitung existiert, um die Differenzengleichungen herzuleiten.
- Die Anwendung der Finiten Differenzen Methode ist problematisch, wenn \mathbf{b} unstetig ist.

Die Finite-Element-Methode vermeidet diese Nachteile, bedarf dagegen aber wesentlich umfangreicherer theoretischer Grundlagen. Zweck der vorliegenden Arbeit ist es, den Leser in die Lage zu versetzen, die mathematischen Hintergründe der Parallelisierung zu verstehen. Zuerst erläutern wir die Vorgehensweise bei der Diskretisierung und motivieren anschließend die Berechnung eines zellweisen Operators auf v_j . Dazu halten wir uns im Folgenden eng an [Jün01].

In den folgenden Abschnitten wird eine einfache Differentialgleichung verwendet: die Poissongleichung. Sie ist auf dem Gebiet $\bar{\Omega} = \Omega \cup \partial\Omega$ definiert als

$$-\Delta u = f \quad \text{in } \Omega \tag{2.5}$$

$$u = 0 \quad \text{auf } \partial\Omega \tag{2.6}$$

und stellt ein Standardbeispiel aus der Numerik dar. Sie wurde von Markus Pögl [Pög04] als Referenzbeispiel implementiert und dient auch der vorliegenden Arbeit als solches.

Die Schreibweise in Gleichung (2.5) ist eine häufig verwendete Abkürzung und äquivalent zu

$$-\operatorname{div}(\nabla u) = f.$$

Für den dreidimensionalen Fall, also für $u \equiv u(x_1, x_2, x_3)$, kann man mit Hilfe von $\nabla u = \left(\frac{\partial u}{\partial x_1} \frac{\partial u}{\partial x_2} \frac{\partial u}{\partial x_3}\right)^T$ und $\operatorname{div}(v_1 \ v_2 \ v_3)^T = \frac{\partial v_1}{\partial x_1} + \frac{\partial v_2}{\partial x_2} + \frac{\partial v_3}{\partial x_3}$ die Gleichung (2.5) schreiben als

$$-\left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \frac{\partial^2 u}{\partial x_3^2}\right) = f \quad \text{in } \Omega. \tag{2.7}$$

Die Grundidee der Finite-Element-Methode ist es, die Gleichung (2.5) in die so genannte schwache Formulierung zu transformieren. Diese entspricht einem unendlichdimensionalen Gleichungssystem. Da dieses praktisch nicht lösbar ist, wird es approximiert, indem man einen geeigneten endlichdimensionalen Ansatzraum einführt. Wählt man diesen geschickt, so ist das resultierende Gleichungssystem möglichst dünn besetzt und somit effizient lösbar. Anschließend werden so genannte elementweise Operatoren berechnet, die von einem Löser verwendet werden können. Diese Vorgehensweise erscheint zunächst weit vom ursprünglichen Problem entfernt, führt aber zu einer Reihe von Möglichkeiten, die die Finite-Element-Methode sehr flexibel machen.

2.2.1 Schwache Formulierung

Wie zuvor beschrieben, besteht die erste Aufgabe in der Umformulierung von Gleichung (2.5), die dazu mit einer Testfunktion $v \in C^1(\bar{\Omega})$ mit der Eigenschaft $v = 0$ auf $\partial\Omega$ multipliziert und anschließend über Ω integriert wird. Dabei ist $C^1(\bar{\Omega})$ die Menge aller innerhalb des Gebiets einfach stetig differenzierbaren Funktionen. Folgende Umformungen lassen sich am besten nachvollziehen, wenn man die Darstellung aus (2.7) verwendet. Die Hauptidee dabei besteht in der Verwendung des Divergenzsatzes und der Tatsache, dass das Randintegral zu 0 wird (wegen $v = 0$ auf $\partial\Omega$). Damit ergibt sich

$$\begin{aligned} \int_{\Omega} f v \, dx &= - \int_{\Omega} \operatorname{div}(\nabla u) v \, dx \\ &= \int_{\Omega} \nabla v \cdot \nabla u \, dx - \int_{\Omega} \operatorname{div}(v \nabla u) \, dx \\ &= \int_{\Omega} \nabla v \cdot \nabla u \, dx - \int_{\partial\Omega} v \nabla u \cdot \nu \, dx \\ &= \int_{\Omega} \nabla v \cdot \nabla u \, dx. \end{aligned}$$

Gesucht wird eine Funktion $u \in X$, so dass $\forall v \in X$ gilt:

$$\int_{\Omega} \nabla v \cdot \nabla u \, dx = \int_{\Omega} f v \, dx. \quad (2.8)$$

Diese Gleichung wird schwache Formulierung (oder auch Variationsgleichung) genannt, da die gesuchte Lösung nunmehr nur noch einmal differenzierbar sein muss. Daher beinhaltet der Raum X zumindest alle einfach differenzierbaren Funktionen (also $C^1(\bar{\Omega})$), die zusätzlich auf dem Gebietsrand verschwinden.

Um einfacher mit der Gleichung (2.8) arbeiten zu können, wird in der Literatur oft die abkürzende Schreibweise

$$a(u, v) = \langle f, v \rangle$$

für die Variationsgleichung eingeführt. Da X unendlich viele Funktionen enthält, die wir für v verwenden können, entsteht ein unendlichdimensionales Gleichungssystem.

2.2.2 Lösungsapproximation

Da wir das durch Gleichung (2.8) definierte unendlichdimensionale Gleichungssystem nicht lösen können, geben wir uns mit einer Approximation zufrieden. Dazu wählen wir einen endlichen Teilraum $X^h \subset X$ und $u^h, v \in X^h$ und formulieren $\forall v \in X^h$

$$\int_{\Omega} \nabla v \cdot \nabla u^h \, dx = \int_{\Omega} f v \, dx. \quad (2.9)$$

Nun haben wir ein endlichdimensionales Gleichungssystem, das praktisch lösbar und in gewohnter Matrixschreibweise darstellbar ist. Dazu definieren wir eine Basis $B_{X^h} = \phi_1, \dots, \phi_n$ von X^h und erhalten damit eine alternative Darstellung für $u^h = \sum_{i=1}^n u_i \phi_i$, wobei n der Anzahl der Basisfunktionen entspricht. Wählt man als die Testfunktion v eine der Basisfunktionen ϕ_j (sog. Galerkin-Ansatz), kann man Gleichung (2.9) schreiben als

$$\sum_{i=1}^n u_i \int_{\Omega} \nabla \phi_i \cdot \nabla \phi_j \, dx = \int_{\Omega} f \phi_j \, dx. \quad (2.10)$$

Diese n Gleichungen (man hat n verschiedene Basisfunktionen für v zur Auswahl) kann man mit Hilfe von $A_{ij} = \int_{\Omega} \nabla \phi_i \cdot \nabla \phi_j \, dx$ und $F_j = \int_{\Omega} f \phi_j \, dx$ schreiben als

$$Au^h = F. \quad (2.11)$$

Damit erhalten wir ein Gleichungssystem, das wir mit jedem iterativen Gleichungssystemlöser lösen können. Dazu müssen allerdings zuerst die Integrale ausgewertet und die Matrix A (auch Steifigkeitsmatrix) und F aufgestellt (assembliert) werden. Der Aufbau und die Besetztheit der Matrix hängt dabei von der Wahl der Basisfunktionen ab. Wählt man zum Beispiel $\phi_j = \sin(j\pi x)$, ergeben sich für A nur Einträge in der Hauptdiagonalen, da

$$\int_{\Omega} \nabla \sin(i\pi x) \cdot \nabla \sin(j\pi x) \, dx = 0 \quad \forall i, j \quad i \neq j.$$

2.2.3 Assemblierung der Steifigkeitsmatrix

In der Praxis werden für die Basisfunktionen typischerweise Funktionen mit einem kleinen Träger gewählt, d.h. der Bereich, an dem $\phi_j(x) \neq 0$, ist sehr klein. Markus Pögl [Pög04] verwendet die auch in Abschnitt 2.4 benutzten Hutfunktionen. Da sich deren Träger nur mit dem der direkt benachbarten Funktionen überschneidet, also das Integral $\int_{\Omega} \nabla \phi_i \cdot \nabla \phi_j \, dx$ für die meisten j zu 0 wird, ist die resultierende Matrix sehr dünn besetzt. Im zweidimensionalen Fall entsteht dabei eine wie in Abbildung 2.2 gezeigte Matrixstruktur.

Dabei sind in jeder Zeile nur die Einträge besetzt, die in der zweidimensionalen Gitterdarstellung benachbart sind (deren Trägerfunktionen sich überschneiden).

Das so aufgestellte lineare Gleichungssystem kann man mit einem iterativen Gleichungslöser lösen. Dieser berechnet dazu für jeden Eintrag $v_{i,j}$ des Lösungsvektors angefangen bei einer Startbelegung schrittweise eine bessere Näherungslösung. Im hier zugrunde liegenden zweidimensionalen Fall besteht der dazugehörige Lösungsvektor aus den linear angeordneten Knoten des Gitters und besitzt daher zwei Indizes. Verwendet man das relaxierte Jacobiverfahren (siehe Abschnitt 2.3) als iterativen Löser, so beschreibt

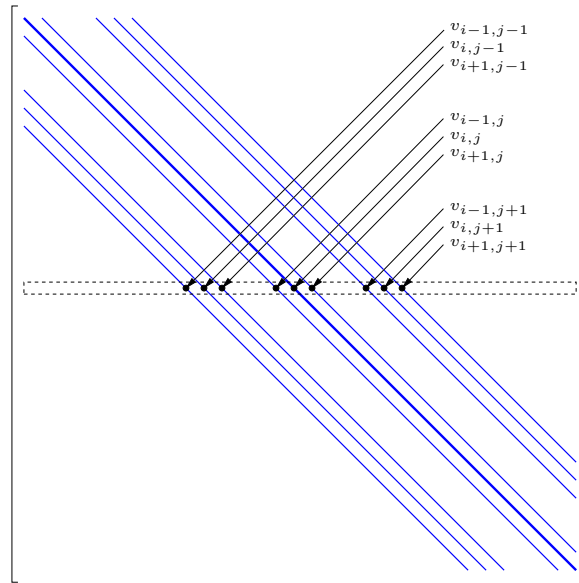


Abbildung 2.2: Aufbau der FEM-Matrix für die zweidimensionale Poisson-Gleichung.

folgende Gleichung einen Schritt der Lösungsverbesserung:

$$v_{i,j}^{k+1} = v_{i,j}^k - \frac{1}{3}\omega D_{i,j}^{-1} \left(\sum_{l=i-1}^{i+1} \sum_{\substack{n=j-1 \\ -(l=i \wedge n=j)}}^{j+1} v_{l,n}^k - 8v_{i,j}^k - b_j \right). \quad (2.12)$$

Ohne zu diesem Zeitpunkt die Herkunft der Gleichung zu verstehen, sieht man dennoch, dass für die Berechnung von $v_{i,j}$ (egal in welcher Iteration also unabhängig von k), nur $v_{i,j}$ und dessen Nachbareinträge addiert werden. Deshalb schreibt man abkürzend statt der Matrix A auch einen so genannten Finite-Element-Stern:

$$A^* = \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Dieser stellt dar, wie man einen Eintrag des Lösungsvektors der nächsten Iteration mit Hilfe seiner Nachbareinträge berechnet. Anschaulich wird dies in [Abbildung 2.3\(a\)](#) gezeigt. Darin kennzeichnen die Pfeile jene Knoten, die in die Berechnung von $v_{i,j}$ eingehen. Wir benötigen demnach Daten aus allen an diesen Knoten angrenzenden Zellen. Damit wird klar, dass eine solche Berechnungsweise von Markus Pögl [[Pög04](#)] nicht umgesetzt werden konnte. Da die Cache-Optimalität seines Algorithmus' auf einer zellbasierten Gebietstraversierung aufbaut, können wir immer nur auf die Eckwerte einer Zelle zugreifen. Wir suchen also einen "Stern", der so aufgebaut ist, dass

er nur die Ecken genau einer Zelle benötigt. In dieser Zelle (auch Element genannt) können dann Operationen ausgeführt werden, die eine wie in Abbildung 2.3(b) gezeigte Zugriffstruktur besitzen.

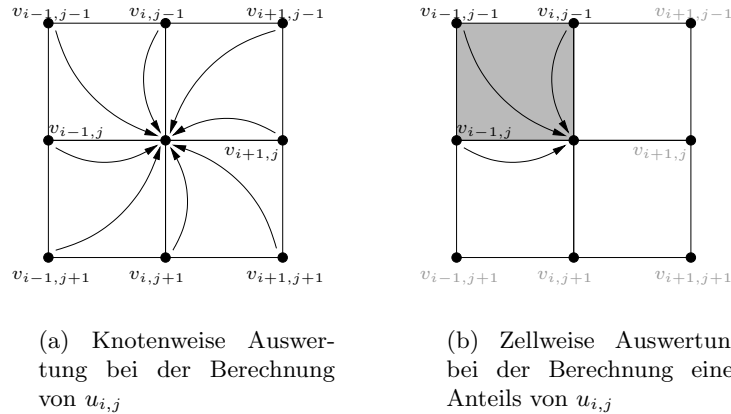


Abbildung 2.3: Verschiedene Auswertungsmuster bei der FEM

Den gesuchten Stern erhalten wir, indem wir Gleichung (2.10) nicht mehr knotenweise, d.h. über den gesamten Träger der Testfunktion, sondern zellweise integrieren. Dazu wird das Integral nur noch über dem Teil der Testfunktionen ausgewertet, die sich innerhalb des gerade betrachteten Elements befinden. Die so erhaltene Steifigkeitsmatrix A und die für die rechte Seite ermittelte so genannte Massenmatrix finden sich für den dreidimensionalen Fall in Markus Pögl's Dissertation [Pög04]. Wie man an Abbildung 2.3(b) erkennen kann, ist es zur Berechnung der gesamten Knotenkorrektur nur notwendig, Zelle für Zelle die Korrekturanteile aller vier angrenzender Knoten zu addieren. Diese Möglichkeit, alle Anteile unabhängige voneinander berechnen und anschließend addieren zu können, führt zu dem in Abschnitt 4.2 vorgestellten Parallelisierungsansatz.

Eine ausführlichere Einführung in die Finite-Element-Methoden, inklusive der Definition der verwendeten Funktionenräume, findet sich im eingangs erwähnten "Kleinen FEM-Skript" [Jün01].

2.3 Lösung linearer Gleichungssysteme

Gesucht wird die Lösung \mathbf{u} eines linearen Gleichungssystems $\mathbf{A}\mathbf{u} = \mathbf{b}$. Ein mögliches Lösungsverfahren ist die explizite Berechnung der Lösung mittels Gauß-Elimination. Dieses Verfahren ist aber wegen seiner hoher asymptotischer Laufzeit in der Praxis nicht sinnvoll einsetzbar. Zudem zerstört es sämtliche günstigen Eigenschaften einer Matrix. Z.B. wird in einer dünn besetzten Matrix im Verlaufe des Verfahrens eine große Menge an Einträgen

erzeugt, die nicht Null sind, und damit der Speicherplatzverbrauch von $O(n)$ auf $O(n^2)$ erhöht. Um dies zu vermeiden, wurden unter anderem sehr effiziente direkte Verfahren entwickelt, die auf der schnellen Fourier-Transformation basieren [BHM00]. Diese beschränken sich jedoch auf das Lösen einiger weniger diskretisierter Differentialgleichungen.

Die Gruppe der iterativen Verfahren hingegen eignet sich zum Lösen sämtlicher linearer Gleichungssysteme. Mit Hilfe von weiteren Techniken erreichen sie zudem die selbe lineare Laufzeit, wie die erwähnten optimierten expliziten Löser. Die Vorgehensweise der iterativen Löser ist dabei immer ähnlich. Ausgehend von einer Anfangsbelegung \mathbf{v}_0 wird die Lösung mit jedem Schritt verbessert. Allgemein beschreibt

$$\mathbf{v}_k = \mathbf{v}_{k-1} + M(\mathbf{b} - A\mathbf{v}_{k-1}), \quad k \in N_0 \quad (2.13)$$

den k -ten Iterationsschritt. Dabei stellt $\mathbf{v}_k = (v_{k,0}, \dots, v_{k,n})$ die Näherungslösung nach der k -ten Iteration dar. Der Term $\mathbf{b} - A\mathbf{v}_{k-1}$ wird als Residuum der k -ten Iteration bezeichnet und mit \mathbf{r}_{k-1} abgekürzt. Man spricht beim Schritt von \mathbf{v}_{k-1} auf \mathbf{v}_k von einem Korrekturschritt und nennt den Term $M\mathbf{r}_{k-1}$ auch Lösungskorrektur.

Offensichtlich stellt $\mathbf{v}_k = \mathbf{u}$ einen Fixpunkt jedes Iterationsverfahrens dieses Typs dar. Die Geschwindigkeit, mit der das Verfahren gegen den Fixpunkt konvergiert, hängt von der Wahl der Matrix M ab. Durch die Entscheidung für eine Matrix M wird zudem das verwendete Verfahren bestimmt. Ein häufig verwendetes Verfahren ist das Jacobi-Verfahren. Man erhält dieses durch eine Aufteilung der Matrix $A = D - L - U$. Dabei stellt D die Diagonalmatrix, L die strikte untere und U die strikte obere Dreiecksmatrix dar. Damit lässt sich das Gleichungssystem $A\mathbf{u} = \mathbf{b}$ als

$$\begin{aligned} (D - L - U)\mathbf{u} &= \mathbf{b} \Leftrightarrow \\ \mathbf{u} &= D^{-1}(L + U)\mathbf{u} + D^{-1}\mathbf{b} \end{aligned}$$

schreiben. Setzt man $R = D^{-1}(L + U)$ und betrachtet die erhaltene Gleichung als Iterationsvorschrift mit Näherungslösung \mathbf{v}_k , so erhält man

$$\mathbf{v}_k = R\mathbf{v}_{k-1} + D^{-1}\mathbf{b}. \quad (2.14)$$

Erneut als Iterationsverfahren mit \mathbf{v}_k als Näherungslösung dargestellt und nach der Addition eines Nullterms ($D^{-1}A\mathbf{u} - D^{-1}A\mathbf{u}$) erhält man aus (2.14) die Gleichung

$$\mathbf{v}_k = \mathbf{v}_{k-1} + D^{-1}(\mathbf{b} - A\mathbf{v}_{k-1}). \quad (2.15)$$

Möchte man also das Jacobi-Verfahren verwenden, so setzt man an die Stelle von M die inverse Diagonal-Matrix D^{-1} in die allgemeine Iterationsgleichung (2.13).

Verwendet man anstatt der Matrix D^{-1} die Matrix $(L+D^{-1})$, so erhält man das Gauß-Seidel-Verfahren und ohne erhöhten Rechenaufwand eine verbesserte Konvergenzrate. Dieses Vorgehen wird auch als Präkonditionieren der Matrix A bezeichnet.

Die Konvergenzgeschwindigkeit eines Iterationsverfahrens hängt dabei direkt von der Kondition der Matrix A ab. Durch die Präkonditionierung von A wird deren Kondition verbessert und damit die Konvergenzgeschwindigkeit erhöht. Eine weitere Möglichkeit, um diese zu verbessern, stellt die Einführung von hierarchischen Basen dar, die in Abschnitt 2.4 vorgestellt werden.

Einen positiven Einfluss auf die Konvergenzgeschwindigkeit eines Iterationsverfahrens hat zudem die Einführung eines Relaxationsparameters ω . Statt den nächsten Iterationswert wie in Gleichung (2.14) zu ermitteln, addiert man den derart berechneten Wert gewichtet zur vorherigen Iterierten. Damit schreibt man

$$\mathbf{v}_k = (1 - \omega)\mathbf{v}_{k-1} + \omega(R\mathbf{v}_{k-1} + D^{-1}\mathbf{b})$$

und erhält, mit Hilfe der Definition der Iterationsmatrix $R_\omega = (1-\omega)I + \omega R$, die Formel

$$\mathbf{v}_k = R_\omega\mathbf{v}_{k-1} + \omega D^{-1}\mathbf{b}. \quad (2.16)$$

Diese wird als ω -relaxiertes Jacobi-Verfahren bezeichnet. Erneut durch Addition eines Nullterms leitet man

$$\mathbf{v}_k = \mathbf{v}_{k-1} + \omega D^{-1}\mathbf{r}_{k-1} \quad (2.17)$$

her. Interessant für die weiteren Betrachtungen ist, dass sich alle vorgestellten Verfahren auf zwei verschiedene Art und Weisen beschreiben lassen. Dies geschieht entweder in der „Korrektur-orientierten“ Form wie die Gleichungen (2.13) (Iterationsverfahren allgemein), (2.15) (Jacobi) und (2.17) (relaxiertes Jacobi-Verfahren) oder in einer „unabhängigen“ Form wie die Gleichungen (2.14) (Jacobi) und (2.16) (relaxiertes Jacobi-Verfahren).

2.4 Hierarchische Basen

Die im Abschnitt 2.2 vorgestellte Diskretisierung führt dazu, dass die gesuchte Funktion $u(x)$ durch einen Vektor \mathbf{v}^h approximiert wird. Dabei zeigt der Index h an, dass der Vektor \mathbf{v} Daten des Gitters Ω^h beinhaltet. Um mit Funktionswerten rechnen zu können, die zwischen den Gitterpunkten liegen, ist es nötig, diese zu interpolieren. Eine einfache Möglichkeit stellt die stückweise lineare Interpolation zwischen den Gitterpunkten dar, die in Abbildung 2.4 gezeigt wird.

Wahlweise kann man diesen linearen Interpolanten \mathbf{v}^h auf dem Gitter Ω^h auch als Linearkombination von so genannten „Hutbasisfunktionen“ $\Phi_j^h(x)$

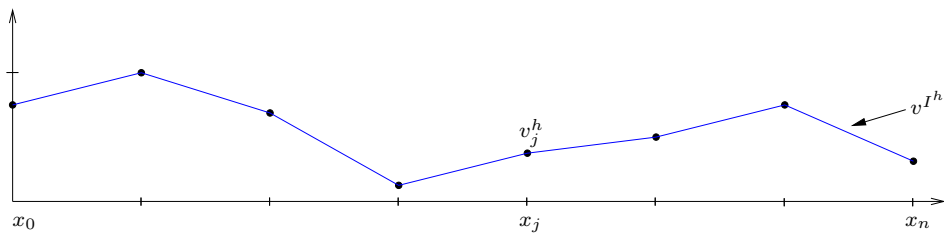


Abbildung 2.4: lineare Interpolation an den Stützstellen

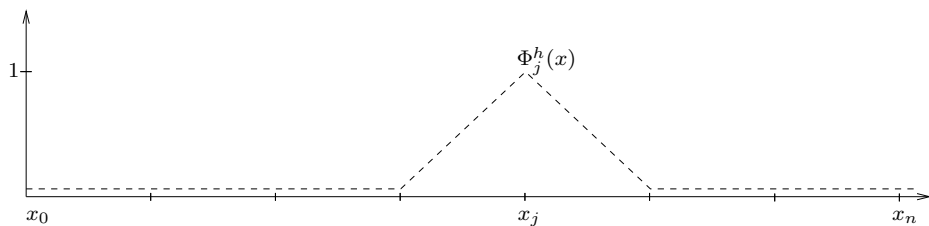


Abbildung 2.5: Hutfunktion an der Stelle x_j auf dem Gitter Ω^h

darstellen. Diese werden, da eine Basisfunktion für jede Stützstelle (Knoten) definiert wird, auch nodale Basis genannt. Eine eindimensionale Hutfunktion auf dem Gitter Ω^h , wie in Abbildung 2.5, ist dabei wie folgt definiert:

$$\Phi_j^h(x) = \begin{cases} 1 - \frac{x_j - x}{h} & x_{j-1} \leq x \leq x_j \\ 1 - \frac{x - x_j}{h} & x_j < x \leq x_{j+1} \\ 0 & \text{sonst} \end{cases}$$

Man sieht leicht ein, dass der von den Hutfunktionen $\Phi_j^h(x)$ aufgespannte Vektorraum

$$V^h = \text{span}\{\Phi_j^h(x), \quad 0 \leq j \leq n\}$$

identisch zu der Menge aller möglichen linearen Interpolanten auf dem selben Gitter ist. Dementsprechend lassen sich alle Interpolanten in der Form

$$\mathbf{v}^{I^h} = \sum_{j=0}^n c_j \cdot \Phi_j^h(x) \quad (2.18)$$

darstellen. Abbildung 2.6 verdeutlicht an einem Beispiel, wie dafür die Basisfunktionen zu kombinieren sind. Die Höhen der einzelnen Hüte stellen die Gewichte $c_j = v_j$ in der Linearkombination dar.

Bei der Implementierung von effizienten numerischen Methoden mit Hilfe von Mehrgitterverfahren (siehe Abschnitt 2.5) werden die diskretisierten Daten häufig auf unterschiedlich fein aufgelösten Gittern benötigt. Dazu ist es nötig, die Darstellung der Daten bzgl. einer Basis in die Darstellung bzgl. einer anderen Basis umzuwandeln. Um z.B. Daten von Gitter Ω^h auf das

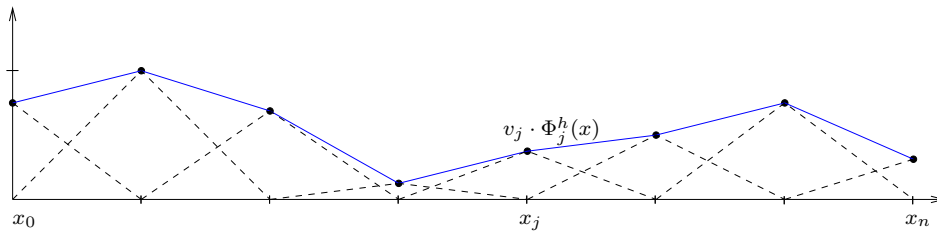


Abbildung 2.6: Darstellung von \mathbf{v}^{I^h} mit Hilfe der $\Phi_j^h(x)$

doppelt so hoch aufgelöste Gitter $\Omega^{\frac{h}{2}}$ zu transportieren, benötigt man die Basisfunktionen und den damit definierten Vektorraum des feineren Gitters. Dieser Vektorraum $V^{\frac{h}{2}}$ konstruiert sich analog zu V^h mit Hilfe von Hutfunktionen $\Phi_j^{\frac{h}{2}}(x)$, deren Breite (genannt Trägerweite) aber nur halb so groß ist.

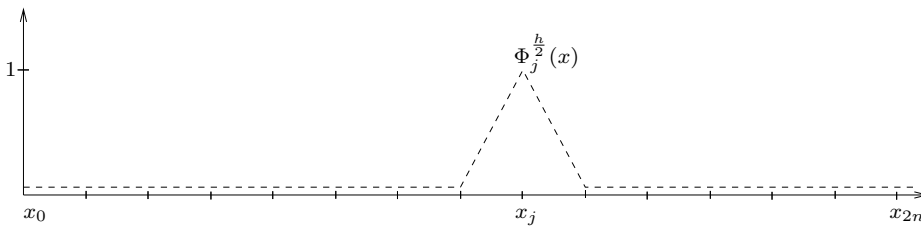


Abbildung 2.7: Hutfunktion des Feinlevels an der Stelle x_j

Abbildung 2.7 stellt eine dieser Basisfunktionen auf dem feineren Gitter $\Omega^{\frac{h}{2}}$ (auch Feinlevel genannt) dar. Entscheidend dabei ist, dass in diesem feineren Gitter jetzt $2n + 1$ statt $n + 1$ Stützstellen x_j und damit dementsprechend viele Basisfunktionen existieren. Mit Hilfe dieser Informationen möchte man

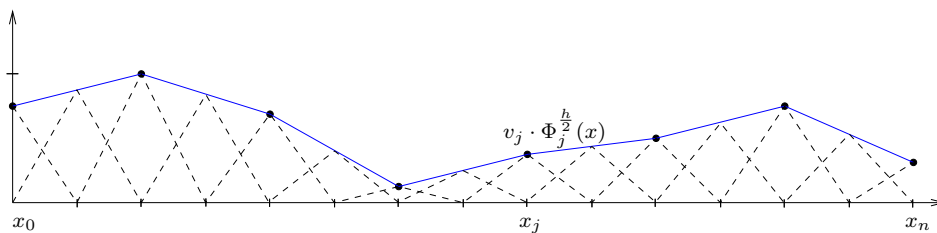


Abbildung 2.8: Darstellung von \mathbf{v} mit Hilfe der $\Phi_j^{\frac{h}{2}}(x)$

nun Daten aus dem Gitter Ω^h in das feine Gitter $\Omega^{\frac{h}{2}}$ transportieren. Offensichtlich lässt sich ein Vektor aus V^h auch in $V^{\frac{h}{2}}$ darstellen, was in Abbildung

2.8 veranschaulicht wird.

Da sich aber alle Basisfunktionen $\Phi_j^h(x)$ (in ihrer Trägerweite) von $\Phi_j^{\frac{h}{2}}(x)$ unterscheiden, ist es nicht möglich, die Gewichte einfach zu übernehmen und zusätzliche Feinleveldaten hinzuzufügen. Vielmehr ist eine Transformation von der Grobblevelbasis in die Feinlevelbasis nötig. Um dies zu vermeiden, bietet sich die Nutzung so genannter hierarchischer Basen an.

Die Haupteigenschaft der hierarchischen Basen liegt in der Tatsache, dass die Menge der Basisfunktionen auf dem Feinlevel eine Übermenge der Basisfunktionen des Grobblevels ist. Dadurch wird es möglich, bei einem Transport der Daten auf ein feineres Level die Grobbleveldaten beizubehalten und nur um Feinlevelinformationen zu ergänzen.

Die hierarchischen Basen auf dem größten Level konstruieren sich wie die nodale Basis. In jedem weiteren Level kommen anschließend weitere Basisfunktionen hinzu, deren Träger exakt zwischen den Maxima zweier benachbarter Grobblevelbasen liegen. Abbildung 2.9 veranschaulicht die Basiskonstruktion im Vergleich zu den zuvor beschriebenen nodalen Basen.

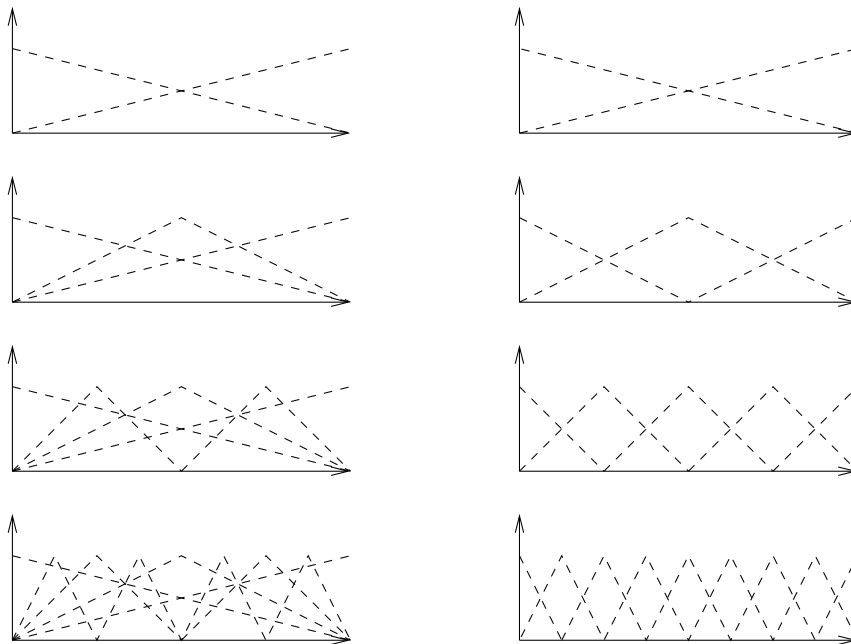


Abbildung 2.9: Konstruktion der hierarchischen (links) und nodalen Basen

Die Verwendung der hierarchischen Basen wird im nachfolgenden Beispiel klar, bei dem mit Hilfe einer ansteigenden Zahl von Gitterpunkten eine Funktion $u(x)$ möglichst genau angenähert wird. Existiert für ein Gitter Ω^h schon eine Annäherung in der Form

$$\mathbf{v}^{I^h} = \sum_{j=0}^n c_j \cdot \Phi_j^h(x),$$

so lässt sich durch die Hinzunahme von weiteren Gitterpunkten deren Genauigkeit erhöhen. Dies geschieht durch Addition weiterer gewichteter Basisfunktionen

$$\mathbf{v}^{I^{\frac{h}{2}}} = \mathbf{v}^{I^h} + \sum_{j=n+1}^{2n} c_j \cdot \Phi_j^{\frac{h}{2}}(x).$$

Für die Gewichte der auf einem feineren Level hinzukommenden Basen wird die Differenz zwischen dem groben Interpolanten und der darzustellenden Funktion $u(x)$ gewählt. Die nachfolgende Abbildung 2.10 macht die Vorgehensweise deutlich. Dabei zeigen die Abbildungen auf der linken Seite, die gesamte Approximation, während die Abbildungen rechts nur die auf dem jeweiligen Level hinzukommenden Basisfunktionen einschließlich ihrer Gewichtung veranschaulichen.

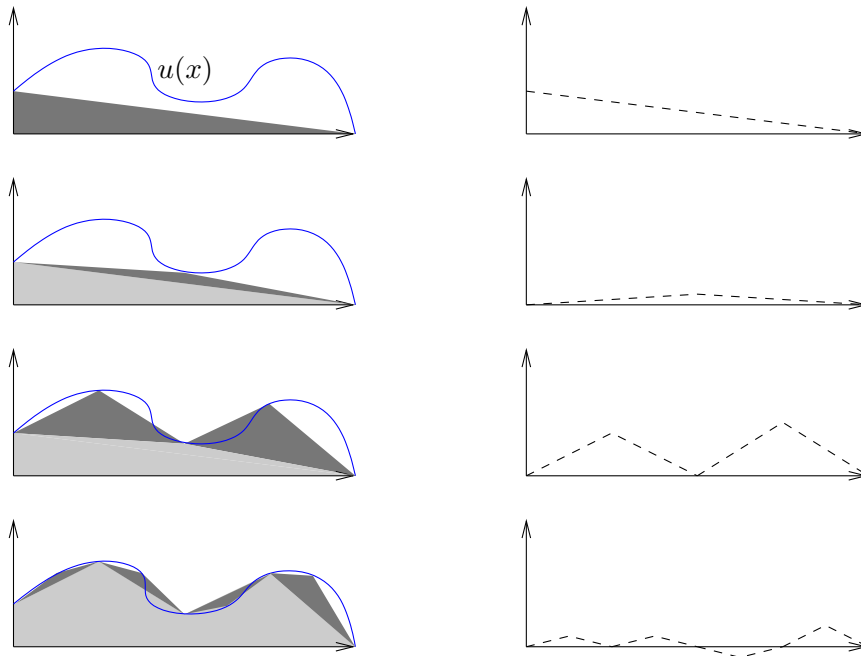


Abbildung 2.10: Approximation von $u(x)$ mit Hilfe von hierarchischen Basisfunktionen.

In Abbildung 2.10 wird deutlich, dass die auf jedem Level hinzukommenden Basen mit zunehmend kleineren Gewichten (genannt Überschüssen) eingehen. Bungartz zeigt in [Bun92], dass für glatte Funktionen der Interpolationsfehler, also auch die Größenordnung der Überschüsse im nächsten Level, von der Größenordnung $O(h^2)$ ist. Dies führt zum ersten Vorteil, den die Verwendung von hierarchischen Basen mit sich bringt. Dieser und die weiteren Vorteile aus der folgenden Auflistung verdeutlichen, wie wichtig das Konzept der hierarchischen Basen in der Numerik ist.

1. Kompakte Speicherung der levelweisen Überschüsse, da diese mit einer Größenordnung von $O(h^2)$ abfallen
2. Transformation der Gleichung $Au = f$ in hierarchische Darstellung führt zu wesentlich besserer Kondition der Iterationsmatrix
3. Durch die Möglichkeit der levelweisen Aufsplittung der Daten lassen sich Multigridverfahren effizient implementieren

Eine Erweiterung dieses Konzeptes stellen die hierarchischen Erzeugendensysteme dar. Bei diesen werden, im Gegensatz zur Darstellung in hierarchischen Basen, nicht nur Funktionen für die neuen Gitterpunkte hinzugefügt, sondern für alle auf dem feineren Gitter vorhandenen Punkte. Abbildung 2.11 stellt diese Idee im Vergleich zu der bekannten Konstruktion der hierarchischen Basen dar. Dabei werden, um in der Darstellung der hierarchischen Erzeugendensysteme die Übersichtlichkeit zu wahren, die Funktionen der jeweils groben Gitter grau eingefärbt. Tatsächlich existieren auf einem Gitter dann sowohl die grauen als auch die schwarzen Hutfunktionen.

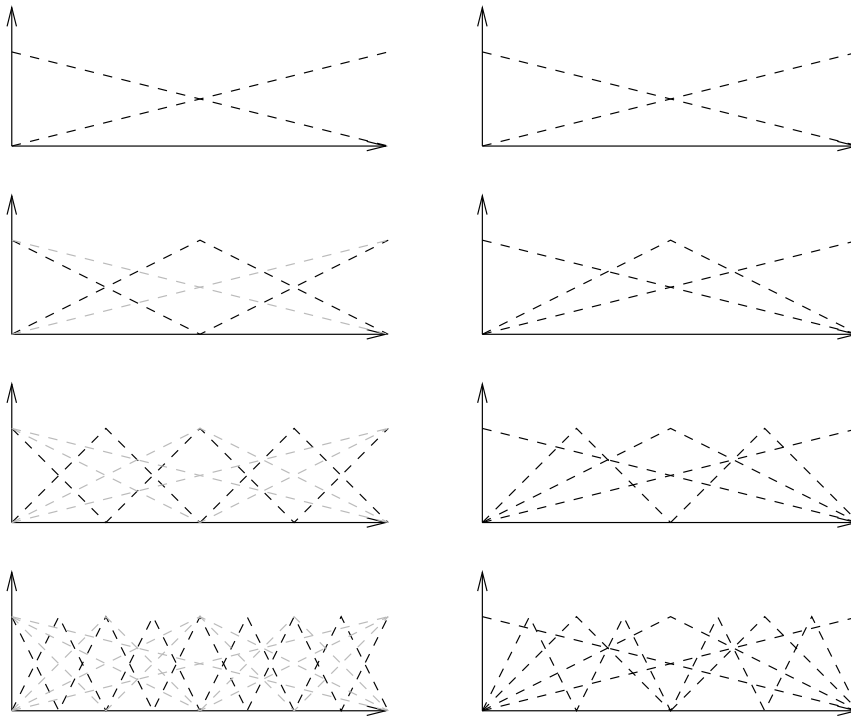


Abbildung 2.11: Konstruktion des hierarchischen Erzeugendensystems (links) und der hierarchischen Basen

Offensichtlich ist die Darstellung im hierarchischen Erzeugendensystem nicht eindeutig. Dennoch bringt sie eine erneute Verbesserung der Kondition der

Iterationsmatrix mit sich, die, wie von Griebel [Gri94] gezeigt wird, unabhängig von der Gitterweite h ist.

Eine sehr schöne Einführung in die Idee der hierarchischen Basen wird in [Kov03] gegeben. Kovacs stellt dort neben den hier gezeigten Konzepten ebenso mehrdimensionale und auf Polynomen basierende hierarchische Basen vor.

2.5 Mehrgitterverfahren

Die in Abschnitt 2.3 beschriebenen iterativen Löser arbeiten alle nach dem selben Prinzip. Um ein gegebenes Gleichungssystem zu lösen, beginnen sie mit einer Startbelegung und versuchen diese iterativ zu verbessern. Dabei stellen sich zwei wesentliche Fragen. Was ist als initiale Schätzung zu wählen und wie entwickelt sich die Näherungslösung im Verlaufe der Iterationen?

Um die Konzepte der Mehrgitterverfahren einzuführen, gehen wir in Anlehnung an [BHM00] zuerst genauer auf die zweite Fragestellung ein. Dazu benötigt man eine Messgröße, die in der Lage ist, zu beschreiben, wie gut eine Näherungslösung \mathbf{v}_k die gesuchte exakte Lösung \mathbf{u} annähert. Da die exakte Lösung nicht bekannt ist, also der Fehler nicht berechnet werden kann, wird zu diesem Zwecke das schon in Abschnitt 2.3 verwendete Residuum benutzt. Dabei steht

$$\mathbf{r}_k = \mathbf{b} - A\mathbf{v}_k$$

für das Residuum der k -ten Iteration. Ist der damit beschriebene Fehler glatt, stellt es eine sehr gute Abschätzung dar. Dadurch erlaubt die Betrachtung des Residuums über mehrere Iterationen hinweg eine Aussage über die Konvergenzgeschwindigkeit des Verfahrens. Wie in [BHM00] beschrieben, führen die ersten Iterationen meist zu einer deutlichen Verringerung des Residuums. Den Grund für die sich nach wenigen Iterationen verschlechternde Konvergenzgeschwindigkeit findet man mit Hilfe folgender Überlegungen.

Löst man anstatt der Ausgangsgleichung $Au = b$ die homogene Gleichung $Au = 0$, so ist die Lösung, nämlich $u = 0$, bekannt. Damit ist man in der Lage, mit $e_k = -v_k$ den exakt Fehler anzugeben und zu beobachten, in welcher Form die Iterationsverfahren den Fehler verringern. Für diesen Sonderfall ist also die Verwendung von r_k nicht mehr nötig. Die entscheidende Idee liegt in der Wahl der Startbelegung für v_0 . Eine interessante Beobachtung lässt sich machen, wenn man für

$$v_{0,j} = \sin\left(\frac{j l \pi}{n}\right), \quad 0 \leq j \leq n, \quad 1 \leq l \leq n-1,$$

also eine diskrete Sinusschwingung der Frequenz $\frac{l}{2}$, wählt. Für große l zeigt sich, dass nach wenigen Iterationen der Fehler e_k fast verschwunden ist. Wählt man dagegen eine Startbelegung mit kleinem l , so stellt man fest, dass diese niedrige Frequenzen im Fehler wesentlich länger erhalten bleiben.

Dies legt nahe, dass die iterativen Löser als Fehlerglätter fungieren, wobei hochfrequente Schwingungen sehr schnell und niederfrequente deutlich langsamer gedämpft werden.

Wie schon im Abschnitt über die hierarchischen Basen 2.4 angedeutet, liegt die Idee der Mehrgitterverfahren darin, die Fehler, die auf einem feinen Gitter nicht bzw. sehr langsam gedämpft werden, auf gröberen Gittern zu verringern. Entscheidend dafür ist die Beobachtung, dass niederfrequente Schwingungen auf einem groben Gitter höherfrequent erscheinen und sich dementsprechend dort wesentlich besser dämpfen lassen. Abbildung 2.12 verdeutlicht dies.

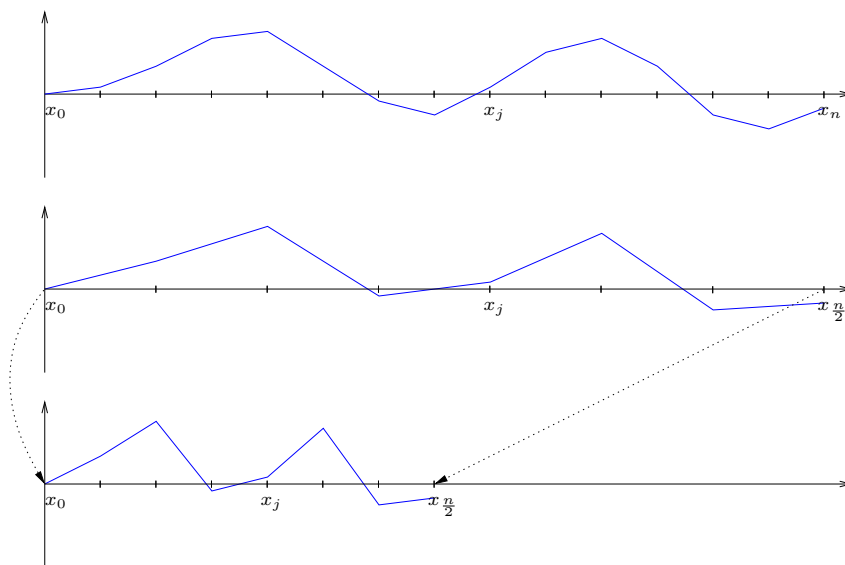


Abbildung 2.12: Betrachtung von Fehlern auf unterschiedlich feinen Gittern.

Damit bietet sich folgende Vorgehensweise an. Nach einigen Iterationen auf einem feinen Gitter wechselt man, sobald sich das Residuum kaum noch ändert, auf ein gröberes Gitter. Dort wird solange iteriert, bis die hochfrequenten Fehleranteile gedämpft wurden und sich das Residuum erneut nicht mehr stark ändert. Anschließend kehrt man zum feinen Gitter zurück und iteriert erneut, bis die durch den Gitterwechsel eingeführten hochfrequenten Fehler erneut verschwunden sind. Dieses Schema lässt sich rekursiv fortsetzen; statt auf dem groben Gitter sämtliche Fehler auszuiterieren, wechselt man, sobald das Residuum keine deutlichen Veränderungen mehr zeigt, erneut auf ein noch gröberes Gitter. Diese Idee führt zu dem bekannten V-Zyklus, der in Abbildung 2.13(a) dargestellt ist.

Neben der Idee, niederfrequente Schwingungen auf groben Leveln zu dämpfen, ist es offensichtlich sinnvoll, das Iterationsverfahren gleich mit einer sehr guten Schätzung zu beginnen. Liegt die Schätzung nahe an der gesuchten Lösung und besitzt nur hochfrequente Fehler, so erreicht man schnell eine

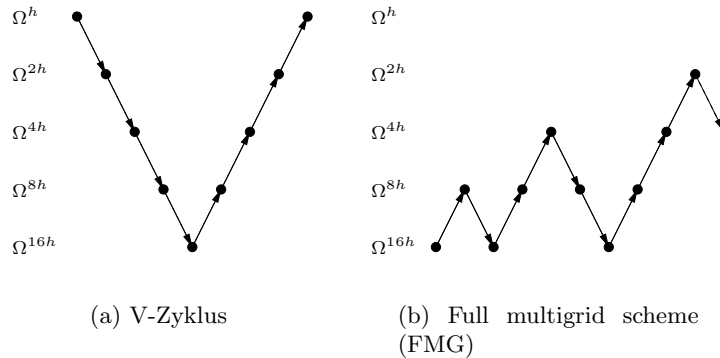


Abbildung 2.13: Überblick über gängige Multigridzyklen

gute Näherungslösung. Diese Überlegungen führen zu der Idee des Full Multigrid Scheme. Dieses gibt vor, statt einer zufälligen Anfangsschätzung für das Gitter Ω^h , die ermittelte Näherungslösung eines gröberen Gitters zu verwenden. Da die Ermittlung der Lösung auf einem groben Gitter wesentlich schneller möglich ist, lohnt sich dieser Aufwand. Erneut bietet es sich an, diese Idee rekursiv umzusetzen (sog. nested iterations), was in Verbindung mit dem V-Zyklus zu dem in Abbildung 2.13(b) gezeigten Vorgehen führt. Diese grundsätzlichen Überlegungen lassen einen wesentlichen Punkt offen. Wie werden die Daten zwischen den Gittern transportiert? Zu diesem Zweck wird die Residuums Gleichung

$$Ae_k = r_k \tag{2.19}$$

eingeführt, die man schnell aus der Definition des Residuums $r_k = b - Av_k$, der Definition des Fehlers $e_k = u - v_k$ und dem zu lösenden Gleichungssystem $Au = b$ erhält. Es lässt sich zeigen, dass diese äquivalent zu dem Ausgangsproblem ist. Damit ist es möglich, statt auf v_k nun auf e_k zu iterieren und mit Hilfe des ausiterierten e_k die Näherungslösung v_k zu korrigieren. Als Anfangsschätzung wählt man $e_0 = 0$.

Aus den Vorüberlegungen zur Berechnung der Lösung von $Av_k = b$ ist bekannt, dass die Iterationen auf dem Gitter Ω^h nur die hochfrequenten Fehler in v_k beseitigen. Deshalb wird nach ein paar Iterationen auf Ω^h nun die äquivalente Residuums Gleichung (2.19) auf dem gröberen Gitter Ω^{2h} gelöst. Dazu wird das Residuum vom feinen auf das grobe Gitter transportiert. Diesen Vorgang nennt man Restriktion und wird wie folgt notiert:

$$r^{2h} = I_{2h}^h r^h \tag{2.20}$$

Für die Restriktion gibt es mehrere Möglichkeiten. Die einfachste stellt die Injektion dar, die einfach die Werte aus dem feinen Gitter übernimmt, die darin an den Positionen des groben Gitters liegen. Meist jedoch werden für

die Restriktion die gewichteten Mittel aus den anliegenden Feingitterpunkten übernommen. Der dazu gehörende Restriktionsoperator hat im Eindimensionalen für den Fall $n = 8$ die Form

$$I_h^{2h} v^h = \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 & & & & & \\ & & 1 & 2 & 1 & & & \\ & & & & 1 & 2 & 1 & \\ & & & & & & 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix}_h = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_{2h} = v^{2h}.$$

Damit ist es möglich, die Residuumswerte aus dem Feinlevel in das Groblevel zu übertragen. Nachdem auf diesem einige Iterationen durchgeführt wurden, ist es nun nötig, den ermittelten Fehler aus Gleichung (2.19) erneut in das feine Level zurückzuführen. Für einen glatten Fehler (ein solcher liegt jetzt vor) ist dies sehr gut mit einer linearen Interpolation möglich und man schreibt

$$e^h = I_{2h}^h e^{2h}. \quad (2.21)$$

Für den Fall einer linearen Interpolation und der gewichteten Restriktion gilt der Zusammenhang

$$I_{2h}^h = c(I_h^{2h})^T.$$

Diese Eigenschaft ist in der Konvergenzanalyse von Mehrgitteralgorithmen von Bedeutung. Dabei stellt man fest, dass mit Hilfe der vorgestellten Algorithmen, eine von der Gitterweite unabhängige Konvergenzrate erreicht werden kann. Diese Eigenschaft ist für die praktische Anwendung von Iterationsalgorithmen von großer Bedeutung, da damit die Beschränkung auf relativ grobe Gitter wegfällt und technische Probleme in wesentlich höherem Detailgrad gelöst werden können.

An dieser Stelle sind alle für den Rest der Arbeit wichtigen Schreibweisen definiert und ihre Verwendung kurz motiviert. Für weitergehende Einblicke wird auf die entsprechende Literatur [BHM00], [TOS01] verwiesen.

2.6 Raumfüllende Kurven

Anschaulich versteht man unter einer raumfüllenden Kurve eine Linie, mit deren Hilfe man ein Blatt Papier zur Gänze füllen kann, ohne dass sich zwei Teilstücke überschneiden. Mathematiker sind schon seit Ende des 19. Jahrhunderts von den Konzepten der raumfüllenden Kurven fasziniert. Neben der Formalisierung von vielen verschiedenen dieser Kurven gelang es ihnen, eine Vielzahl ihrer Eigenschaften zu beweisen. Dieses aus der reinen Mathematik kommende Konzept gewinnt inzwischen in der Praxis zunehmend an Bedeutung. Immer dann, wenn man für unstrukturiert im Raum liegende

Elemente eine lineare Ordnung benötigt, finden raumfüllende Kurven eine Verwendung. Auch die vorliegende Arbeit nutzt einige ihrer Vorteile aus. Bevor jedoch in den Abschnitten 3.1 und 4.2 auf ihren praktischen Nutzen eingegangen wird, zeigt der folgende Teil der Arbeit die Konstruktionsweise und einige für uns interessante Eigenschaften auf. Die verwendeten Schreibweisen und die genannten Beweisideen orientieren sich stark an [Sag94].

2.6.1 Konstruktion von raumfüllenden Kurven

Cantor zeigte 1878, dass die Kardinalität eines Intervalls $I = [0, 1]$ gleich der Kardinalität einer Ebene $Q = [0, 1]^2$ ist. Damit war klar, dass es eine Abbildung geben musste, die das Einheitsintervall surjektiv auf das Einheitsquadrat abbildet. Peano und Hilbert waren die ersten, die eine solche zur Erzeugung einer raumfüllenden Kurve geeignete Abbildung angeben konnten. Um den Begriff raumfüllende Kurve exakt beschreiben zu können, benötigt man folgende Definitionen.

Ist f eine Funktion von \mathbb{E}^m nach \mathbb{E}^n , definiert auf $D(f)$ und $A \subset \mathbb{E}^m$, so wird

$$f_*(A) = \{f(x) | x \in A \cap D(f)\}$$

das Bild von A unter f genannt. Für den Fall, dass $f : I \rightarrow \mathbb{E}^n$ stetig ist, bezeichnet man $f_*(I)$ als Kurve. Ist zudem noch

$$n \geq 2 \wedge J_n(f_*(I)) > 0,$$

handelt es sich bei $f_*(I)$ um eine raumfüllende Kurve. $J_n(f_*)$ steht hier für den Jordaninhalt (Fläche, Volumen, ...) von f_* .

Peano formalisierte die nach ihm benannte Peanokurve mit Hilfe von

$$\begin{aligned} k(t) &= 2 - t \text{ für } t = 0, 1, 2 \quad \text{und} \\ k^\nu(t) &= \underbrace{k \circ k \circ \dots \circ k}_\nu(t). \end{aligned}$$

Damit konnte er eine Abbildung eines in Ternärdarstellung $0_3t_1t_2t_3 \dots$ gegebenen Punktes aus dem Einheitsintervall auf einen Punkt im Einheitsquadrat wie folgt angeben:

$$f_{Peano}(0_3t_1t_2t_3 \dots) = \begin{pmatrix} 0_3t_1(k^{t_2}(t_3))(k^{t_2+t_4}(t_5)) \dots \\ 0_3(k^{t_1}(t_2))(k^{t_1+t_3}(t_4)) \dots \end{pmatrix} = \begin{pmatrix} 0_3x_1x_2 \dots \\ 0_3y_1y_2 \dots \end{pmatrix} \quad (2.22)$$

Mit Hilfe der Tatsache, dass $k \circ k(t) = t$, also k die inverse Funktion zu sich selbst ist, lässt sich die Surjektivität von $f_{Peano}(t)$ leicht zeigen [Sag94]. Wie die Abbildung $f_{Peano}(t)$ anschaulich zu interpretieren ist, zeigt folgende Überlegung.

Die Zahlen, die sich im Intervall $I_1 = [0, \frac{1}{9}] = [0, 0_301]$ befinden, lassen sich

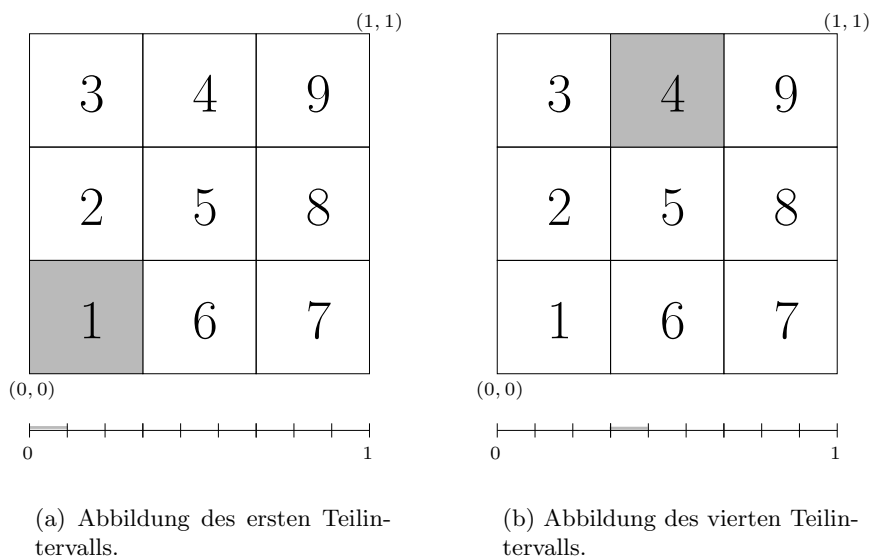


Abbildung 2.14: Abbildung des Einheitsintervalls auf das Einheitsquadrat.

als $t = 0_300t_3t_4t_5$ darstellen. Diese werden auf

$$f_{\text{Peano}}(0_300t_3t_4t_5 \dots) = \begin{pmatrix} 0_30(k^{t_2}(t_3))(k^{t_2+t_4}(t_5)) \dots \\ 0_30(k^{t_1+t_3}(t_4)) \dots \end{pmatrix}$$

abgebildet, liegen also im in Abbildung 2.14(a) gezeigten Teilquadrat 1. Die Zahlen aus dem Intervall $I_4 = [\frac{1}{3}, \frac{4}{9}] = [0_31, 0_311]$ liegen dementsprechend im Teilquadrat 4 (siehe Abbildung 2.14(b)). Betrachtet man allgemein die Teilintervalle $I_i = [\frac{i-1}{9}, \frac{i}{9}]$, so ergibt sich der in Abbildung 2.15(a) gezeigte Durchlauf durch 9 Teilquadrate. Teilt man die Teilintervalle I_i rekursiv jeweils in weitere 9 Teilintervalle, so erhält man den in Abbildung 2.15(b) dargestellten Gebietsdurchlauf auf einer diskreten Peanokurve. Führt man diese Intervallschachtelung unendlich oft fort, so erhält man die kontinuierliche Peanokurve.

2.6.2 Hölder-Stetigkeit

Für die (kontinuierliche) Peanokurve lässt sich zeigen, dass sie zwar nicht differenzierbar, aber stetig auf I ist. Im Bezug auf die Stetigkeit lässt sich noch eine stärkere Aussage machen.

Gilt

$$\|f(x) - f(y)\|_2 \leq C_H \cdot |x - y|^\alpha, \quad (2.23)$$

so ist f Hölder-stetig mit dem Exponent α . Anschaulich begrenzt diese Gleichung den Abstand zweier Kurvenpunkte in Abhängigkeit vom Abstand ih-

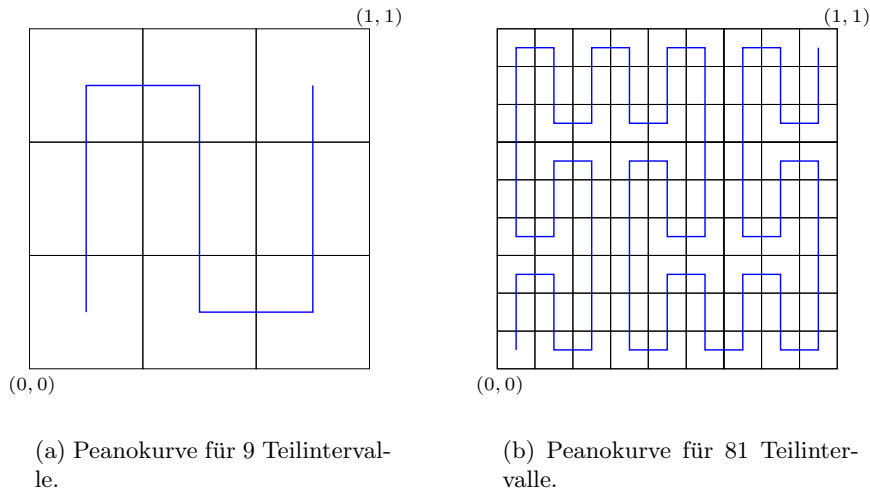


Abbildung 2.15: Gebietsdurchlauf auf einer diskreten Peanokurve.

rer Urbilder, gibt also Aufschluss über die Kompaktheit der Kurve f_* . Es lässt sich zeigen, dass die Peanokurve für Dimension d Hölder-stetig mit dem Exponenten $\frac{1}{d}$ ist. Dies sieht man mit Hilfe der folgenden Überlegung ein. Man wählt analog zu [Zum01] zwei beliebige Punkte $t_1, t_2 \in I$ mit einem Abstand $k = |t_2 - t_1|$. Zusätzlich wählt man eine Intervallschachtelung, bei der die Fläche (das Volumen) v_Q eines Quadrates (Quaders) gerade noch größer ist als k , also $3^{-d(n+1)} < k \leq 3^{-dn} = v_Q$. Da die Abbildung f_{Peano} ein Teilquadrat erst komplett durchläuft, bevor sie in das nächste (benachbarte) eintritt, liegt $[t_1, t_2]$ höchstens in zwei benachbarten Quadraten mit der Fläche v_Q . Der maximale Abstand zwischen t_1 und t_2 ist demnach durch die Diagonale durch diese beiden Quadrate gegeben, also $3^{-n}\sqrt{3+d}$. Mit $k > 3^{-d(n+1)} \Rightarrow 3^{\frac{d}{d}}k > 3^{-n}$ gilt

$$\|f(t_2) - f(t_1)\|_2 \leq 3^{-n}\sqrt{3+d} \quad (2.24)$$

$$\leq 3\sqrt{3+d} \sqrt[d]{|t_2 - t_1|} \quad (2.25)$$

und damit ist die Peanokurve Hölder-stetig mit dem Exponenten $\frac{1}{d}$. Wir wissen also, dass die Entfernung zweier Punkte auf der Peanokurve durch die Entfernung ihrer Urbilder beschränkt ist.

Dabei fällt auf, dass eine der beiden Abschätzungen sehr ungenau ist, da wir entweder (2.24) viel zu große Quadrate (Quader) um die Kurve herum legen (falls $k \approx 3^{-d(n+1)}$) oder (falls $k \approx 3^{-dn}$) die zweite Abschätzung (2.25) sehr ungenau ist. Insgesamt verlieren wir für die Abschätzung der Fläche (des Volumens) auf jeden Fall etwa eine Gittergrößenordnung (3^{-d}). Dies sollten wir im Hinterkopf haben, um im Folgenden brauchbare Konstanten C_H wählen zu können.

2.6.3 Bestimmung der Oberfläche von Peanokurvenabschnitten

Ein weiterer interessanter Punkt ist die Abschätzung der Oberfläche von Teilabschnitten der Peanokurve. Dabei ist das Verhältnis von Volumen v zu Oberfläche s von Interesse. Die Kugel stellt ein theoretisches Optimum für dieses Verhältnis dar. Für sie gilt im Dreidimensionalen

$$s_{Kugel} = 3 \cdot v_{Kugel}^{\frac{2}{3}}.$$

Etwas aufwändiger gestaltet sich dieser Zusammenhang im n-Dimensionalen:

$$s_{Kugel} = \sqrt[d]{2d^{d-1} \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2})} v_{Kugel}^{\frac{d-1}{d}}}.$$

Der erste Teil der Formel ist nur dimensionsabhängig, also konstant bzgl. des Volumens. Allgemein schreiben wir:

$$s \leq C_{Part} \cdot v^{\frac{d-1}{d}} \quad (2.26)$$

mit geeignet gewähltem C_{Part} . Oberflächen, die bis auf einen konstanten Faktor dem Ausdruck $v^{\frac{d-1}{d}}$ entsprechen, nennen wir *quasi-optimal*.

Es lässt sich zeigen, dass das Oberflächen-Volumenverhältnis eines Teilabschnittes $[t, t+v]$ der Peanokurven ebenso quasi-optimal ist. Dazu konstruiert man einen Würfel mit $f(t)$ als Zentrum und der aus der Hölder-Stetigkeit bekannten Seitenlänge

$$l_{Würfel} = 2 \cdot 3\sqrt{3 + d} \sqrt[d]{v}.$$

Durch die Wahl der Seitenlänge ist sichergestellt, dass die Kurve immer innerhalb dieses Würfels verläuft. Da die Oberfläche der Peanokurve nicht fraktal ausgeprägt ist, kann sie durch ein Vielfaches von

$$s_{Würfel} = 2d(l_{Würfel})^{d-1}$$

nach oben abgeschätzt werden. Für den dreidimensionalen Fall erhält man damit:

$$\begin{aligned} s_{Peano} \leq c_{Würfel} \cdot s_{Würfel} &= c_{Würfel} \cdot 2d(6\sqrt{3 + d} \sqrt[3]{v})^{d-1} = \\ c_{Würfel} \cdot 6 \cdot 36(\sqrt{6} \sqrt[3]{v})^2 &= c_{Würfel} \cdot 1296 \cdot v^{\frac{2}{3}} \end{aligned}$$

Verbessert man diese Abschätzung, indem man direkt die Oberfläche der zwei benachbarten Quadranten aus dem Beweis der Hölder-Stetigkeit berechnet, erhält man

$$s_{Peano} \leq c_{Würfel} \cdot 90 \cdot v^{\frac{2}{3}}, \quad (2.27)$$

also eine wesentlich bessere Grenze. Im Anhang [A](#) wird gezeigt, dass der bis hierhin noch unbekannte Faktor $c_{\text{Würfel}}$, der die Oberfläche der Peanokurve zum umschreibenden Würfel nach oben hin abschätzt, im Dreidimensionalen nahe 1 liegt. Die damit konkretisierte Abschätzung [2.27](#) wird im Abschnitt [4.3](#) für die Praxis angepasst und mit empirischen Werten verglichen.

Kapitel 3

Ausgangssituation

Die vorliegende Arbeit baut auf den Ergebnissen der Dissertation von Markus Pögl [Pög04] auf. Der Inhalt seiner Arbeit war die Entwicklung eines Cache-optimalen sequentiellen Gleichungslösers auf Basis der Finite-Element-Methode. Den Aufbau und die wichtigsten Konzepte seiner Arbeit fassen wir im Abschnitt 3.1 zusammen. Darin beschreiben wir sowohl die verwendeten numerischen Verfahren als auch die implementierten Algorithmen, um den Leser in die Lage zu versetzen, die Hintergründe der darauf aufbauenden Parallelisierung zu verstehen.

Abschnitt 3.2 gibt grundlegende Informationen über das zur Parallelisierung verwendete Message Passing Interface (MPI).

3.1 Der sequentielle Algorithmus

3.1.1 Numerik

Moderne Programme zum Lösen von Differentialgleichungen benutzen eine Vielzahl von numerischen Konzepten. Nur deren effiziente Kombination ermöglicht eine schnelle Lösung aktueller praktischer Probleme. Der zugrundeliegende zu parallelisierende Löser verwendet sämtliche in Kapitel 2 vorgestellten Konzepte. Aufbauend auf einer Finite-Element-Diskretisierung der Poissongleichung und einer Datendarstellung bezüglich hierarchischer Basen, wird ein relaxiertes Jacobiverfahren zur iterativen Lösung des entstandenen linearen Gleichungssystems verwendet. Um die Konvergenzgeschwindigkeit zu erhöhen, kommt ein additives Mehrgitterschema zum Einsatz. Der Einsatz all dieser Konzepte hat wesentlichen Einfluss auf den Aufbau des im Folgenden beschriebenen Programms.

Die Diskretisierung des Rechengebietes, wie sie in Abbildung 2.1 für den eindimensionalen Fall vorgestellt wird, führt im Mehrdimensionalen dazu, dass ein Gitter entsteht. Auf dessen Eckpunkten (bzw. Knoten) liegen die Freiheitsgrade und damit auch die zu speichernden Daten. Ein Beispiel für ein mögliches reguläres zweidimensionales Gitter wird in Abbildung 3.1 gezeigt.

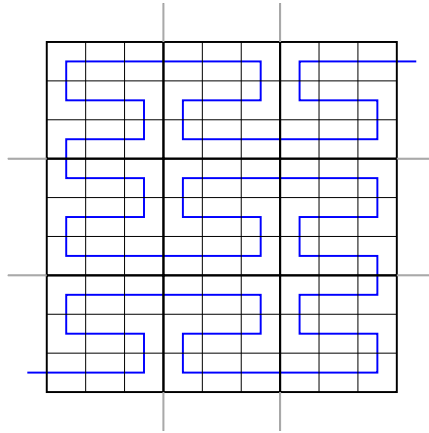


Abbildung 3.1: Mögliches zweidimensionales Gitter.

Die darin zu erkennende wiederkehrende Dreiteilung des Rechengebiets ist durch den Gebietsdurchlauf entlang der diskreten Peanokurve bedingt. In jeder Iteration des iterativen Gleichungslösers wird ein solcher Durchlauf durchgeführt, in welchem man die elementweise Knotenkorrekturen für jeden Knoten berechnet. Entscheidend dabei ist die Tatsache, dass wir Zelle für Zelle und nicht Knoten für Knoten durch das diskrete Gebiet laufen. Damit ergeben sich zwei wesentliche Eigenschaften des Lösers, die wir noch einmal zusammenfassen.

- Speicherung der Daten in den Zellecken,
- aber zellweiser Durchlauf durch das Gebiet.

Dies hat einen ganz wesentlichen Einfluss auf die numerische Verarbeitung der Daten. Wie in Abschnitt 2.2.3 beschrieben, benötigen wir also einen Operator, der nicht knotenweise eine Näherungsverbesserung vornimmt sondern zellweise (siehe Abbildung 2.3). Dies wird erst durch die Flexibilität der Finite-Element-Methode ermöglicht.

Ein weiteres wichtiges Merkmal des Lösers ist der zum Einsatz kommende Gleichungslöser. Wie schon in Abschnitt 2.3 beschrieben, ist die Verwendung eines iterativen Gleichungslösers im Prinzip gängiger Standard. Im vorliegenden Fall wird daher das relaxierte Jacobiverfahren verwendet. Damit ergibt sich für den gesamten Lösungsalgorithmus der für iterative Löser typische und in Algorithmus (1) vorgestellte Aufbau.

Jeder Aufruf der Steuerungsmethode veranlasst einen kompletten Gebietsdurchlauf, nach dessen Abschluss jedes Datenfeld neue Näherungswerte beinhaltet. Dies wird solange wiederholt, bis das Maximum der bei der Lösungskorrektur verwendeten Residuen kleiner als ein vorgegebener Abbruchwert wird. Diese Grundstruktur wird im Abschnitt 4 erweitert, um den Anforderungen der Parallelisierung gerecht zu werden.

Algorithmus 1 Iterativer Löser

```
1: procedure MAIN
2:   Initialisierung();           ▷ Aufbau der Datenstrukturen
3:   while (maxResiduum < abbruchResiduum) do
4:     Steuerung();
5:   end while
6:   Ausgabe();                 ▷ Aufbereitung der Ergebnisse
7: end procedure
```

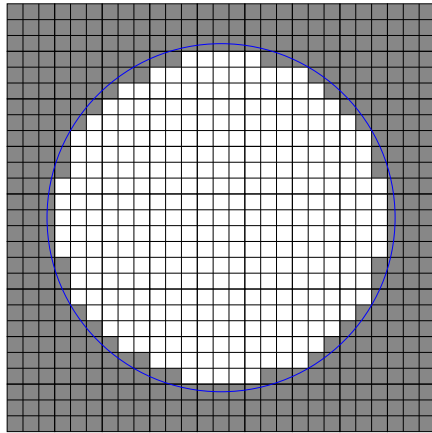
3.1.2 Geometrie und Gitterbeschreibung

Bei der Beschreibung der Geometrie und des Diskretisierungsgitters werden eine Vielzahl von Begriffen verwendet. Grundsätzlich versteht man unter dem Begriff “Geometrie” die Beschreibung der Hindernisstruktur und unter dem Begriff “Gitter” die Gesamtheit der existierenden Zellen.

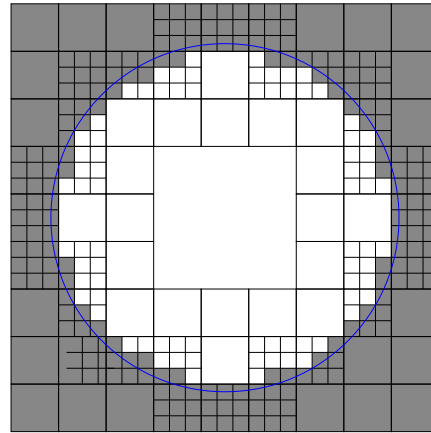
In der Praxis ist es notwendig, verschiedenste Körper zu modellieren, innerhalb derer die Simulation abläuft. Dazu werden die Zellen eines Gitters in *außen* (grau) und *innen* (weiß) eingeteilt. Innerhalb eines Gebietes (z.B. eines Rohres) gelten die gegebenen Differentialgleichungen, außerhalb entsprechend nicht. Der blaue kontinuierliche Kreis in den Abbildungen 3.2 stellt eine Rohrwand dar und legt fest, ob eine Zelle innen oder aussen liegt. Dadurch bestimmt er die Geometrie des Hindernisses.

Hinzu kommt, dass man einen gegebenen Körper möglichst exakt auflösen möchte. Dazu ist es wie in Abbildung 3.2(a) notwendig, sehr fein aufgelöste Gitter zu verwenden. Dadurch steigt allerdings sowohl der Speicherplatzbedarf als auch die benötigte Rechenzeit. Deshalb weicht man auf so genannte nicht reguläre Gitter aus. Ein Beispiel für ein solches Gitter ist in der Abbildungen 3.2(b) dargestellt. Man sieht, dass für die Modellierung eines Rohres ein nicht reguläres Gitter offensichtlich sehr viele Zellen und entsprechend auch Freiheitsgrade spart.

Zuletzt gilt es, die Knoten in Typen einzuteilen. Nachdem wir Zellen in innen und außen eingeteilt haben, gilt diese Unterscheidung entsprechend auch für die Ecken. Hinzu kommt ein weiterer Knotentyp, der an der Grenze zwischen Innen- und Außenzellen liegt. Diese werden Randknoten und entsprechend der implementierten Randbedingung Dirichletknoten genannt. Diese Unterscheidung der Knoten wäre ausreichend, wenn wir nur reguläre Gitter, wie in Abbildung 3.2(a), verwenden würden. Da dies nicht der Fall ist, wird weiterhin zwischen Groblevel- und Feinlevelknoten unterschieden. Betrachtet man zwei unterschiedliche Gitter, wie es bei der Verwendung von Mehrgitteralgorithmen notwendig ist (siehe 2.5), so werden Knoten auf dem größeren Gitter Groblevelknoten und Knoten auf dem feineren Gitter Feinlevelknoten genannt. Ist das feinere Gitter das an der betrachteten Stelle am höchsten auflösende (d.h. es gibt keine noch feineren Gitter), so nennt



(a) Reguläres Gitter.



(b) Nicht reguläres Gitter.

Abbildung 3.2: Gitter für geometrische Approximation eines Rohrquerschnitts

man dessen Zellen Blattzellen und dessen Knoten Blattlevelknoten. Knoten, die am Rand eines Verfeinerungslevels liegen und damit nicht ausschließlich von gleichgroßen Zellen umgeben sind (also keinen definierten Träger für eine Basisfunktion besitzen), werden hängende Knoten genannt. Ebenfalls zu den hängenden Knoten zählen jene, deren Träger sich mit Hinderniszellen überschneidet. Abbildung 3.3 stellt alle auftretenden Knotentypen an einem Beispiel dar.

Eine Variante der Geometriebeschreibung muss an dieser Stelle noch angesprochen werden. Das Konzept der rekursiven Teilung von Zellen (geometrischen Objekten) in kleinere einbeschriebene Zellen, lässt sich sehr gut und effizient mit Hilfe von Bäumen repräsentieren. Wird eine Grobblevelzelle durch einen Vaterknoten in einem Baum repräsentiert, so stellen dessen Kinder die innerhalb der Grobblevelzelle liegenden Feinlevelzellen dar. Die Idee wird am einfachsten an einem Beispiel wie in Abbildung 3.4 klar.

Der so entstandene Baum wird als “spacetree” bezeichnet. Seine Nummerierungsreihenfolge ergibt sich aus der Durchlaufreihenfolge auf der diskreten Peanokurve.

3.1.3 Datenzugriff und -verwaltung

Um den Durchlauf durch das Rechengebiet möglichst effizient zu gestalten, ist es wichtig, sehr schnell auf die Daten der aktuell bearbeiteten Zelle zugreifen zu können. Die angesprochene Repräsentation des Gitters mit Hilfe von Bäumen stellt eine Möglichkeit dar, dies zu erreichen. Allerdings erfordern solche Darstellungen immer den Einsatz von Pointern. Diese haben im

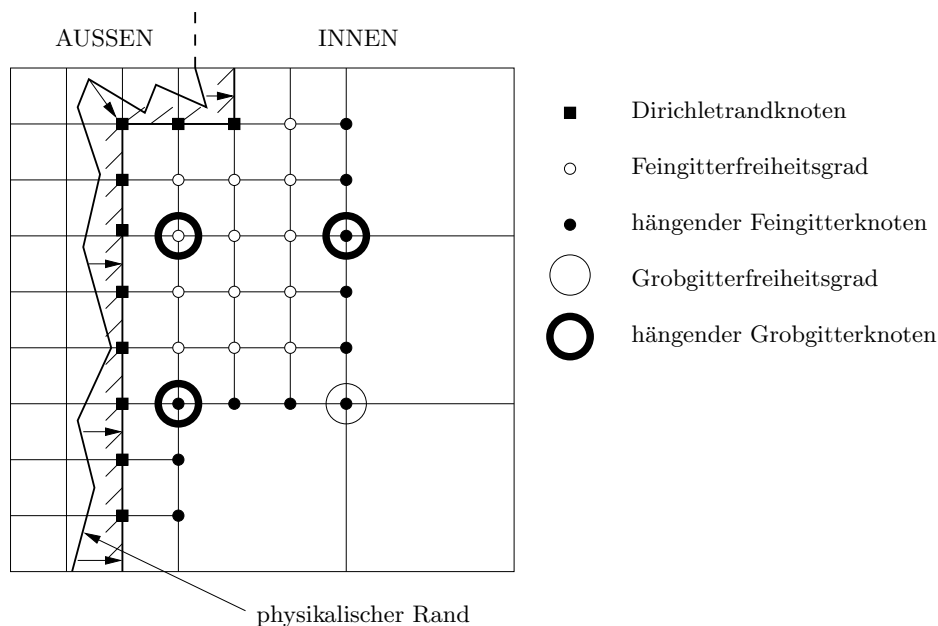


Abbildung 3.3: Spezifizierung der auftretenden Knotentypen [Pög04]

Prinzip als einzige Aufgabe, die Struktur der Gitterdaten zu speichern. Der damit verbundene Speicherbedarf erscheint gering, ist aber in der Praxis ein nicht zu vernachlässigender Faktor, wenn man bedenkt, dass Beispiele mit 10^7 Gitterknoten und mehr berechnet werden. Allein die Speicherung eines einzigen Pointers pro Knoten benötigt in diesem Fall etwa 40MByte Arbeitsspeicher in einer 32Bit Architektur. Zusätzlich hat die Verknüpfung der Daten mittels Pointern den entscheidenden Nachteil, dass geometrisch dicht beieinander liegende Knoten im Speicher weit von einander entfernt sein können. Diese Tatsache und die sternbasierte Auswertung der numerischen Berechnungen führen zu einer schlechten Cacheeffizienz [Lan04].

Um den beiden genannten Nachteilen (schlechte Cacheeffizienz und hoher Speicherverbrauch) zu begegnen, entwickelte Markus Pögl [Pög04] einen Löser, der einen komplett anderen Ansatz zur Speicherung der Daten wählt. Sein Konzept basiert auf der Idee, das diskretisierte Gebiet entlang einer raumfüllenden Kurve zu durchlaufen. Da dies allein nicht der Schlüssel zu hoher Cacheperformance und niedrigem Speicherverbrauch ist, werden die Daten (Knoten) nun statt in d -dimensionalen Feldern auf einer Vielzahl ($= 3^d$) von Kellern gespeichert. Dies erlaubt einen extrem einfachen Zugriff auf die Knoten mittels `push()` und `pop()` und induziert, dass die Daten nur noch linear abgearbeitet werden. Dieser Tatsache ist es zu verdanken, dass die Speicherhierarchie sehr gut ausgenutzt werden kann ([Pög04], [Lan04]). Ungünstigerweise schränkt diese Speicherung den Zugriff auf die Daten sehr ein, da nur noch die oben auf den Kellern liegenden Knoten erreicht werden

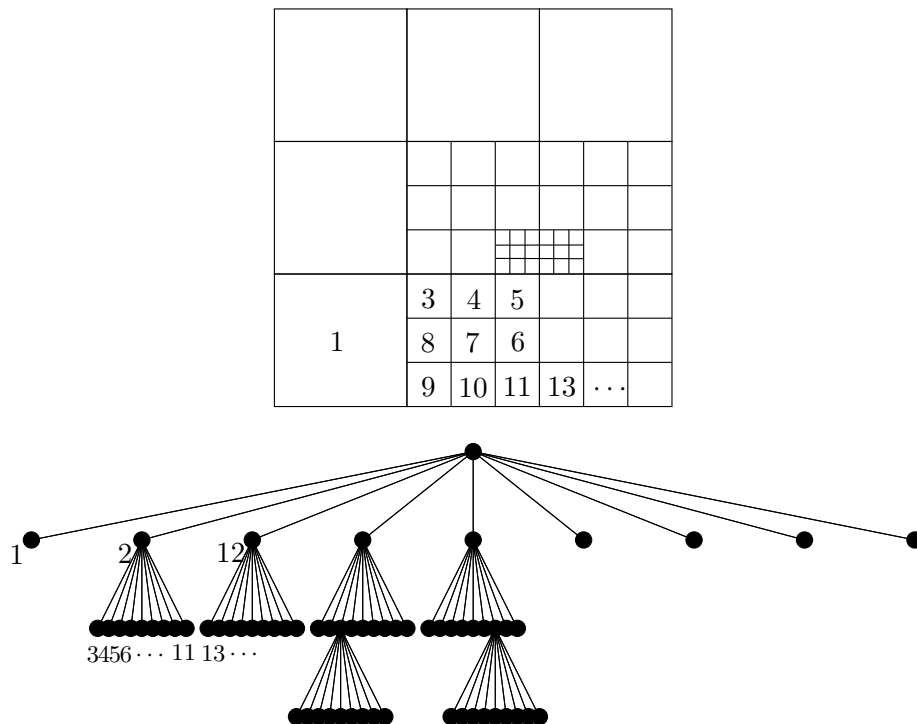


Abbildung 3.4: Gitter mit entsprechender Repräsentation als spacetree

können. Im Speziellen kann nur auf die Eckknoten der aktuell bearbeiteten Zelle zugegriffen werden. Diese Art und Weise des Datenzugriffs muss also, um die guten Performanceergebnisse des sequentiellen Algorithmus erreichen zu können, erhalten bleiben.

Um die Menge der zu speichernden Daten klein zu halten, speichert der sequentielle Algorithmus möglichst keine Werte, die er aus dem aktuellen Zustand σ ermitteln kann. Zu σ zählen vor allem die Information über die aktuelle Position der Traversierung und die Daten der Grobzellen, die auf höheren Leveln liegen. Die Daten der Groblevelzellen liegen dem Algorithmus deshalb vor, weil er zur Traversierung des Gebietes eine Tiefensuche auf dem spacetree durchführt und damit vor jeder Feinlevelzelle die darüberliegende Vater- bzw. Groblevelzelle besucht hat. Das Konzept der Tiefensuche dient hier alleine der Veranschaulichung, wie der Gebietsdurchlauf im Falle einer Speicherung in einem spacetree aussehen würde.

Numerisch gesehen werden während einer Iteration die Korrekturanteile jeder Zelle berechnet. Diese Korrekturanteile werden dann auf deren Eckknoten verteilt. Da jeder Knoten im Dreidimensionalen Eckknoten von 8 (im d -dimensionalen 2^d) Zellen ist, wird also jeder Knoten 8 mal verändert bzw. auf ihn zugegriffen. Auf Grund des für den Kellerzugriff notwendigen Zugriffsdeterminismus [Pög04] muss jeder Knoten exakt die erwähnten 8 mal

gelesen und gespeichert werden. Knoten, die seltener verarbeitet werden würden, blieben, vereinfacht gesprochen, im Weg liegen und würden den Zugriff auf die später benötigten blockieren. Aus diesem Grund werden die Knoten, die wir nicht 8 mal verarbeiten können (weil sie nicht an 8 Zellen angrenzen), auch nicht gespeichert. Im sequentiellen Fall trifft dies auf die Menge der hängenden Knoten zu.

Diese und weitere genannte Eigenschaften wollen wir zusammen mit den wichtigsten Konsequenzen festhalten. Um die Anschaulichkeit zu wahren, werden wir die Datenstrukturen nur noch als abstrakte Container, also ohne Kenntnis des eigentlichen Aufbaus, betrachten:

- Das Gebiet Ω wird entlang der Peanokurve durchlaufen. D.h. die Reihenfolge der Zellen ist fest vorgegeben.
- Beim Durchlaufen einer Zelle stehen nur die Informationen aus deren Eckknoten zur Verfügung. Dies gilt weiterhin auch, wenn eine Feinlevelzelle durchlaufen wird. D.h., dass einer Feinlevelzelle auch die Informationen der Vaterzellen (aufsteigend bis zur Wurzel des spacetrees) zur Verfügung stehen.
- Die Zellen werden in der Reihenfolge einer Tiefensuche im spacetree durchlaufen. Damit müssen wir, um eine Blattzelle durchlaufen zu können, erst in alle darüber liegenden Grobblevelzellen eintreten.
- Jeder Knoten wird innerhalb einer Traversierung 8 mal verarbeitet.
- Vor dem Lesen der Daten von der Datenstruktur muss bekannt sein, welche Knoten im Moment erreichbar sind (oben auf den Kellern liegen). Das liegt daran, dass wir aus Effizienzgründen nicht erst einen Knoten lesen können, um anschließend festzustellen, dass wir für ihn keine Verwendung haben. Also benötigen wir vor dem Lesen die Kenntnis, ob ein Datenelement überhaupt auf der Datenstruktur liegt und wenn dem so ist, auf welchem Keller es sich befindet. Dies ist z.B. von Bedeutung, wenn Knoten einer Zelle gelesen werden, bei der nicht alle Ecken Freiheitsgrade sind und somit auch nicht auf der Datenstruktur gespeichert werden.

Gerade letztere trivial erscheinende Aussage wird für die Parallelisierung einen bedeutenden Aufwand verursachen und entsprechend im Abschnitt 4 weiter ausgeführt. Eine wesentlich ausführlichere Beschreibung vor allem der verwendeten Datenstrukturen und des Zugriffsdeterminismus' findet sich in [Pög04].

3.1.4 Gebietstraversierung

Eine der Hauptaufgaben des sequentiellen Algorithmus' ist es, den Gebietsdurchlauf entlang der Peanokurve zu ermöglichen. Dies ist für den dreidi-

mensionalen Fall eine sehr komplexe Aufgabe. Um die dabei auftretende Vielzahl von verschiedenen Fällen (Orientierung der Teilabschnitte, etc.) zu kontrollieren, wurde von Markus Pögl ein dimensionsrekursiver Ansatz gewählt. Dabei teilt er rekursiv den gerade traversierten Körper entlang der Raumrichtungen in drei Teile. Angefangen bei einem Würfel wird dieser in z -Richtung also entlang dreier Platten durchlaufen. In jeder von diesen läuft der Algorithmus in y -Richtung drei Balken ab, um schließlich jeden Balken in x -Richtung in drei Würfel zu teilen. Die Tatsache, dass diese Fälle bis auf Drehungen und Spiegelungen identisch sind, ermöglicht eine deutlich kompaktere Darstellung im Programm. In Abbildung 3.5 wird der Zerlegungsvorgang anschaulich dargestellt.

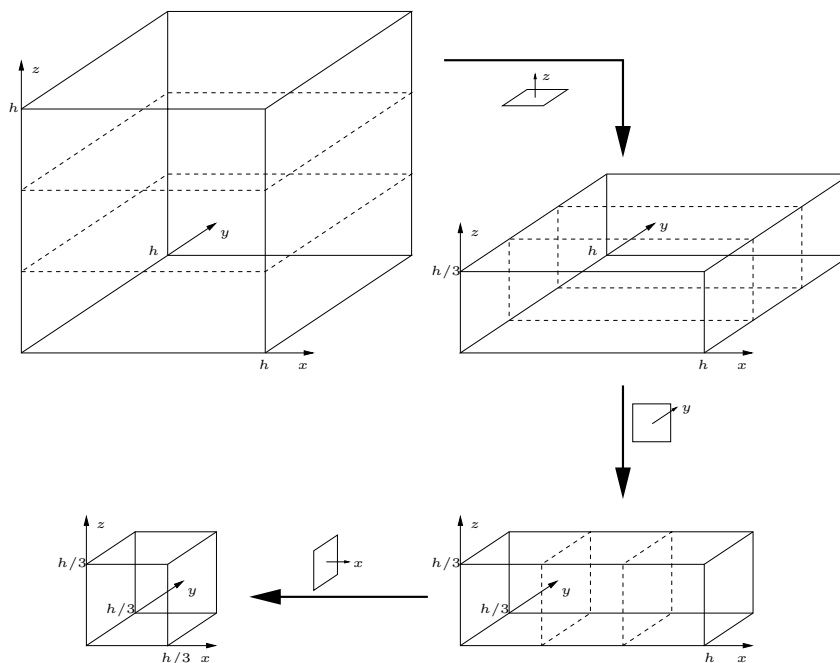


Abbildung 3.5: Dimensionsrekursive Zerlegung eines 3D-Würfels [Pög04]

Interessant hinsichtlich der anstehenden Parallelisierung ist der Steuerungsalgorithmus, der eine derartige Traversierung leistet. Der in (2) vorgestellte Algorithmus ist wesentlich vereinfacht (vereinfachte Numerik und ohne Verwaltung), stellt aber alle wesentlichen oben beschriebenen Bestandteile dar.

Der Algorithmus läuft durch die drei Teilabschnitte des Elternelements (Zeile 3) und ruft rekursiv auf jedem erneut eine Dreiteilung auf (Zeile 5 bzw. Zeile 11). Das Residuum wird in jeder Blattzelle (Zeile 9) berechnet. Mehrgitterberechnungen (Zeile 13) finden jeweils nach vollständiger Traversierung einer Zelle statt.

Diese Vorgehensweise, jeden geometrischen Körper rekursiv in drei Teile zu

Algorithmus 2 Verallgemeinerte Steuerungsmethode

```
1: procedure STEUERUNG(...)
2:   Init();
3:   for (i=0; i<3; i++) do
4:     if ( $\neg$  Würfel()) then  $\triangleright$  Falls innerhalb von Platte oder Balken
5:       Steuerung();  $\triangleright$  Durchlauf der nächst feineren Struktur
6:     else  $\triangleright$  Falls innerhalb eines Würfels
7:       LeseDaten();
8:       if (blattLevel()) then  $\triangleright$  Falls auf feinstem Level
9:         BerechneResiduum();
10:      else  $\triangleright$  Falls nicht auf feinstem Level
11:        Steuerung();  $\triangleright$  Durchlauf durch die nächsten 3 Platten
12:      end if
13:      BerechneMehrgitter();
14:      SchreibeDaten();
15:    end if
16:  end for
17: end procedure
```

zerlegen, wirkt sich auch auf unsere in Abbildung 3.4 vorgestellte Repräsentation als spacetree aus. Jeder Knoten besitzt nun nur noch drei statt neun Kindknoten, die sich jeweils erneut in drei weitere Kindknoten aufsplitten. Die Nummerierung wird allerdings weiterhin nur auf Würfelebene durchgeführt. D.h. dass die Balken (und Platten) keine eigenen Nummern erhalten. In den folgenden Kapiteln wird auf diesem Steuerungsalgorithmus aufgebaut und dieser so erweitert, dass er nicht mehr durch den gesamten Würfel sondern nur noch Teilgebiete durchläuft.

3.2 Grundlagen von MPI

Einen Algorithmus zu parallelisieren bedeutet, seine Ausführung auf mehrere Prozessoren zu verteilen. Dies kann auf unterschiedlichste Art geschehen. Der einfachste denkbare Ansatz ist es, mehrere verschiedene oder ein und das selbe Programm, auf mehreren Rechnern zu starten und zwischen diesen Prozessen zu kommunizieren. Da die dabei auftretenden Aufgaben immer gleicher Art sind, gibt es verschiedene Frameworks, mit deren Hilfe man die Parallelisierung bewerkstelligen kann. Diese unterscheiden sich vor allem in ihrem zu Grunde liegenden Programmiermodell.

Eine Möglichkeit, Prozesse zu parallelisieren, stellt das “shared memory” Konzept dar. Bei diesem können gemeinsame Speicherbereiche definiert und unter gegenseitigem Ausschluss darauf zugegriffen werden. Ein weit verbreiteter Vertreter ist OpenMP [ope]. Dieser stellt einen speziellen Compiler bereit, der den sequentielle Code compiliert und selbständig die Paralle-

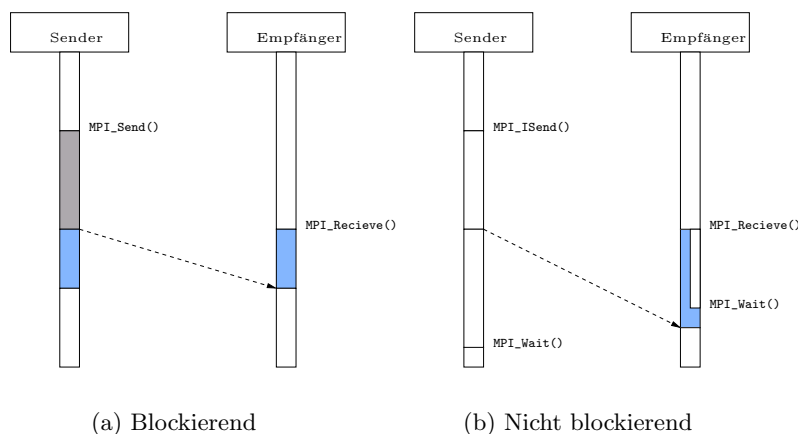


Abbildung 3.6: Kommunikationsvarianten im MPI (blau - Kommunikation; grau - Wartezeit; weiß - Rechenzeit).

lisierung übernimmt. Der Programmierer kann darauf mittels Direktiven Einfluss nehmen. Ein Hauptnachteil dieses Programmiermodells stellt die geringe Kontrolle darüber, wie der Compiler Kommunikation und Datenaufteilung vornimmt, dar. Daraus resultieren zumeist schlechte Performanceergebnisse. Als Vorteil ist die Tatsache zu nennen, dass der mit der Parallelisierung verbundene Aufwand vergleichsweise gering ausfällt.

Ein alternativer Ansatz ist das so genannte “message passing”. Darin stehen dem Programmierer Operationen zur Verfügung, die eine nachrichtenbasierte Punkt-zu-Punkt Kommunikation ermöglichen. Zusätzlich gibt es eine Vielzahl kollektiver Operationen wie Broadcasts oder Barriers. Um einen einheitlichen plattformübergreifenden Standard zu schaffen, wurde das “message passing interface” (MPI) eingeführt, von dem es eine Vielzahl freier und kommerzieller Implementierungen gibt. In der vorliegenden Arbeit wurde die MPI-Implementierung MPICH vom Argonne National Laboratory [mpi] verwendet. Der Hauptvorteil von MPI liegt in den damit erreichbaren sehr guten Performanceergebnissen, denen aber ein hoher Implementierungsaufwand gegenüber steht.

Um dem Leser eine Idee von der Nutzung von MPI zu geben, werden im Folgenden ein paar grundsätzliche Konzepte vorgestellt. Wie der Name schon sagt, basiert MPI auf nachrichtenbasierter Kommunikation. Es existieren also die zwei einfachen Primitiven `receive` und `send`, mit denen Nachrichten empfangen bzw. versendet werden können. Dabei werden zwei Varianten unterschieden, blockierend und nicht blockierend.

Für unsere Zwecke ist dieser Unterschied zwischen blockierenden Primitiven und nicht blockierenden sehr interessant. Verwendet man blockierende send- und receive-Aufrufe, so kehrt der Aufruf erst nach Beendigung

der Kommunikation zurück (3.6(a)). Man spricht in einem solchen Fall von nicht lokaler Beendigungssemantik, da das Verlassen der Funktionen nicht ausschließlich vom lokalen Prozess abhängt. Bei nicht blockierenden send- und receive-Aufrufen hingegen kehrt der Aufruf zurück, sobald die gesamte die Kommunikation betreffende Information (Sender, Empfänger, Identifikator) zwischengespeichert wurde. Von diesem Zeitpunkt an können beide Prozesse mit weiteren Berechnungen fortfahren. Um den Speicherbereich, in dem die zu sendenden Daten liegen, überschreiben bzw. auf die empfangenen Daten zugreifen zu können, wird noch eine weitere Funktion benötigt. Diese so genannten complete-send- sowie complete-recv-Methoden kehren erst zurück, wenn die Kommunikation komplett abgeschlossen ist. Für beide Varianten, das blockierende Senden und Empfangen sowie das nicht blockierende, wird in Abbildung 3.6 ein Beispiel gegeben. Dabei sind `MPI_Send()` und `MPI_Recv()` blockierende Funktionen, `MPI_Isend()` und `MPI_Irecv()` hingegen stellen die nicht blockierenden Entsprechungen dar. Mit Hilfe von `MPI_Wait()` wird auf das Ende einer nicht blockierenden Kommunikation gewartet.

Die Vorteile der nicht blockierenden Kommunikation sind klar erkennbar. Der Sender hat die Möglichkeit, nachdem er die Sendefunktion aufgerufen hat, beliebige weitere Aktionen (auch weitere send-Aufrufe) durchzuführen. Genauso kann der Empfänger verfahren, nachdem er das Empfangen initiiert hat. Im Hintergrund führt MPI parallel die Kommunikation durch und kann so die Effizienz des Programms erhöhen. Mit Hilfe des Aufrufs von `MPI_Wait()` können beide Prozesse kontrollieren, dass die Ausführung eines Programms erst dann fortfährt, wenn die Kommunikation abgeschlossen ist. Dies kann unter Umständen schon beim Aufruf von `MPI_Wait()` der Fall sein, so dass es wie für den Sender in 3.6(b) praktisch zu keiner Verzögerung kommt.

Für die beiden existierenden Varianten (blockierend und nicht blockierend) gibt es in MPI eine Vielzahl an Modi, die es erlauben, die Art und Weise der Kommunikation zu beeinflussen und damit auf die jeweilige Applikation anzupassen. Dazu zählt neben einem robusten Standardverfahren (basic) die Möglichkeit der Zwischenspeicherung der zu sendenden Daten (buffered send/recv), die synchrone Datenübertragung (synchronous) und die Übertragung mittels eines speziell für kurze Nachrichten optimierten Protokolls (ready).

Kapitel 4

Parallelisierung

Die Parallelisierung eines numerischen Lösers besteht typischerweise aus zwei Aufgaben.

1. Es muss ein Weg gefunden werden, den Rechenaufwand auf p Prozessoren (Prozesse) zu verteilen, wofür es zwei mögliche Ansätze gibt. Entweder werden die Daten, auf denen Berechnungen ausgeführt werden, möglichst gleichmäßig auf alle Prozessoren verteilt (Datenparallelität), oder alle Prozessoren führen unterschiedliche Berechnungen auf den gesamten Daten aus (funktionale Parallelität). Die vorliegende Implementierung verwendet einen datenparallelen Ansatz, der in der Numerik sehr häufig eingesetzt wird. D.h. mittels Gebietspartitionierung wird das Rechengebiet in p Teile zerlegt und jeder Prozessor führt ausschließlich auf seinem Teilgebiet Berechnungen durch.

2. Nachdem das Berechnungsgebiet in mehrere Teile unterteilt wurde, existieren Daten, die von mehreren Prozessoren gelesen und auch verändert werden. Um sicherzustellen, dass alle Prozessoren immer mit den aktuellen Werten rechnen, teilt man jede Änderung allen daran interessierten Prozessoren mit. Dies wird innerhalb eines sich an jede Iteration anschließenden Kommunikationszyklus erledigt. Da sich mit zunehmender Anzahl an Prozessoren die reine Berechnungszeit verringert und der Aufwand für die Kommunikation steigt, ist es wichtig, darauf zu achten, dass die Kommunikation sehr effizient gestaltet wird.

Abschnitt 4.1 beschreibt, wie sich die Effizienz und damit die Güte eines parallelen Programms in Zahlen ausdrücken lässt. Die beiden darauf folgenden Abschnitte stellen die Umsetzung der Aufteilung des Rechengebietes (4.2) und die Gestaltung des Datenaustauschs (4.3) dar.

4.1 Effizienz paralleler Programme

Lässt man ein paralleles Programm auf p Prozessoren laufen, so erhofft man sich eine Laufzeit, die etwa dem p -ten Teil der Laufzeit auf einem Prozes-

sor entspricht. Dies lässt sich mit Hilfe des Speedups $s(p)$ überprüfen. Man rechnet

$$s(p) = \frac{T(1)}{T(p)}.$$

Der Speedup ist also das Verhältnis der Laufzeit auf einem Prozessor $T(1)$ zur Laufzeit auf p Prozessoren $T(p)$. Je höher der Speedup $s(p)$ für gegebenes p ist, desto effizienter arbeitet ein paralleles Programm. Interessanter ist allerdings das Verhalten des Speedups bei steigender Prozessorenanzahl. Ist $T(p)$ linear im Verhältnis zu p , so spricht man von einem *linearen Speedup*. In einem solchen Fall bedeutet eine Verdoppelung der Prozessorenanzahl auch eine Verdoppelung der Ausführungsgeschwindigkeit.

Die so gewonnenen Aussagen beziehen sich allerdings ausschließlich darauf, wie gut ein paralleler Algorithmus mit zunehmender Zahl an Prozessoren skaliert, und nicht wie gut der parallele Löser im Vergleich zu anderen ist. Dazu wird der sequentielle Speedup eingeführt, der einen parallelen mit dem schnellsten bekannten sequentiellen Algorithmus vergleicht.

$$s_{sequ}(p) = \frac{T_{sequ}}{T(p)}$$

Um den Begriff Speedup in eine anschaulichere Darstellung zu bringen, wird die parallele Effizienz eingeführt.

$$e(p) = \frac{s(p)}{p}$$

Sie macht eine Aussage darüber, wie viel einer der an der Berechnung beteiligten p Prozessoren im Vergleich zur Berechnung auf einem Prozessor leistet. D.h. bei einer parallelen Effizienz von $e(p) = 0,5$ führt jeder der beteiligten Prozessoren die Berechnungen halb so schnell durch wie im Falle einer Berechnung mit einem Prozessor. Ziel ist es also, eine Effizienz $e(p)$ zu erreichen, die nahe an 1 liegt. Entsprechend strebt man einen Speedup an, der in etwa der Anzahl p der beteiligten Prozessoren entspricht. Auch bei der Effizienz unterscheidet man die genannte parallele Effizienz und die sequentielle Effizienz. Bei zweiterer liegt einfach statt dem parallelen Speedup der sequentielle Speedup zugrunde.

4.2 Partitionierung

4.2.1 Theoretischer Ansatz

Anforderungen: Um einen möglichst hohen Speedup zu erreichen, gilt es, die Ausführungszeit auf p Prozessoren, also die parallele Laufzeit, zu minimieren. Dazu muss man den durch die Parallelisierung eingeführten Mehraufwand so gering wie möglich halten. Zu diesem Zweck betrachten

wir, wovon die parallele Laufzeit $T(p)$ beeinflusst wird. Sie setzt sich aus der maximalen Zeit $\max\{T_i(p)\}$ zusammen, die ein Prozessor i benötigt, um seine Berechnungen durchzuführen, und der Zeit T_{comm} , die für den Datenaustausch nötig ist.

$$T(p) = \max\{T_i(p)\} + T_{comm} \quad (4.1)$$

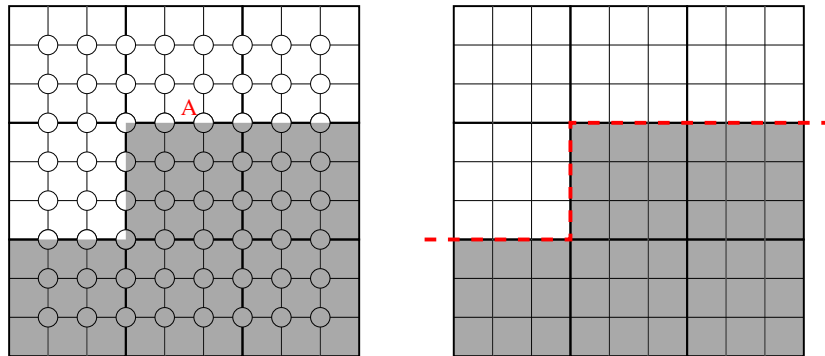
Man erkennt, dass es, um den ersten Summanden zu minimieren, notwendig ist, alle Prozessoren möglichst gleichmäßig auszulasten. Diesen Vorgang nennt man auch Lastausgleich (load balancing). In dem von uns gewählten datenparallelen Ansatz weisen wir also allen Prozessoren einen etwa gleich großen Abschnitt aus dem gesamten Rechengebiet zu. Da der sequentielle Algorithmus das Rechengebiet Ω zellweise durchläuft, ist es sinnvoll, dieses an den Zellgrenzen zu zerteilen, so dass jeder Prozessor eine bestimmte Menge an kompletten Zellen zu bearbeiten hat. Der erste Teil der Gleichung (4.1) wird also minimal, wenn es gelingt, jedem Prozessor etwa die gleiche Anzahl von Zellen zuzuweisen.

Jeder Prozessor führt nun eine Iteration auf seinem Teilgebiet durch und tauscht anschließend jene Daten aus, die von den übrigen Prozessoren benötigt werden. Die Menge der dabei ausgetauschten Daten bestimmt, wie viel Zeit T_{comm} man für die Kommunikation zwischen den Prozessoren aufwenden muss. Dementsprechend sind wir daran interessiert, die Menge der auszutauschenden Daten zu minimieren.

Dazu ist es erforderlich, sich klar zu machen, welche Daten notwendig sind, damit ein Prozessor seine Berechnungen durchführen kann. Legt man den in Abschnitt 2.2 beschriebenen elementweisen Stern zugrunde, erhält man ein Auswertungsschema wie in Abbildung 2.3(b). Darin erkennt man, dass für die Berechnung des Korrekturanteils einer Zelle alle anliegenden Eckknoten benötigt werden. Dementsprechend benötigt jeder Prozessor die Eckdaten aller in seinem Teilgebiet liegenden Zellen. Damit ergibt sich das in Abbildung 4.1(a) gezeigte Schema für die Datenverteilung. Da die Zellecken auf dem äußersten Rand nicht als Freiheitsgrade behandelt werden, sind hier auch keine Datenknoten eingezeichnet.

Interessant sind die auf dem Rand liegenden (zweifarbigen) Knoten, da diese von beiden Prozessoren benötigt werden. Grundsätzlich stellen genau diese Knoten das Hauptproblem dar, welches bei einer Datenaufteilung auf mehrere Prozessoren auftritt. Da sie auf dem Rand zwischen Teilgebieten liegen, kann ein Prozessor allein nicht alle Korrekturanteile dieser Knoten berechnen. Wie schon am Ende von Abschnitt 2.2 beschrieben, birgt die elementweise Betrachtung des Rechengebietes genau die Eigenschaft, dass Knotenkorrekturanteile einzeln berechnet und anschließend addiert werden können. D.h. für den mit "A" markierten Knoten berechnet Prozessor P1 die unteren beiden Elementanteile, P2 die beiden oberen. Anschließend werden die berechneten Anteile ausgetauscht und addiert.

Die Tatsache, dass die Korrekturanteile aller Knoten auf den Prozessrändern zwischen den benachbarten Prozessoren ausgetauscht werden müssen, macht deutlich, dass man bei der Partitionierung darauf achten muss, die Gebietsränder möglichst klein zu halten. Nur so ist es möglich, den Kommunikationsaufwand niedrig zu halten.



(a) Darstellung der benötigten Daten.

(b) Darstellung des Prozessrandes.

Abbildung 4.1: Möglichkeiten zur Darstellung einer Gebietsaufteilung.

Zur Vereinfachung stellen wir ab sofort die Gebietsaufteilung aus Abbildung 4.1(a) wie in Abbildung 4.1(b) ohne explizite Darstellung der Daten dar. Die eingefärbten Teilgebiete werden *Berechnungspartition* (kurz: BP) und die Grenze zwischen den Gebieten *Prozessrand* (kurz: PR) genannt. P_i steht für die Berechnungspartition von Prozessor i und ∂P_i für den gesamten Prozessrand von P_i .

Zusammenfassend suchen wir also einen Algorithmus, der ein diskretes Gebiet in gleich große Berechnungspartitionen zerlegt und dabei eine Zerlegung derart wählt, dass die Summe der Ränder (bzw. Oberflächen) der Teilstücke minimal ist. Es ist bekannt, dass dieses Problem NP-vollständig ist [TC92], weswegen man sich in der Praxis mit guten Näherungen zufrieden gibt.

Partitionierungskonzept: Neben einer Vielzahl von graphenbasierten Algorithmen werden raumfüllende Kurven zunehmend häufiger eingesetzt, um dieses Problem anzugehen. Ihr Vorteil liegt in der Einfachheit der Algorithmen, die zudem sehr gute Ergebnisse liefern. Raumfüllende Kurven stellen, wie in 2.6 beschrieben, eine Möglichkeit dar, das Einheitsintervall $[0, 1]$ surjektiv auf einen mehrdimensionalen Raum abzubilden. Beschränkt man sich bei der Diskretisierung des Rechengebiets auf Drittelungen, so gibt es eine Intervallschachtelung des Einheitsintervalls, bei der sich jedes Teilintervall mittels f_{Peano} (2.22) eindeutig auf ein Element der Diskretisierung abbilden lässt. Damit erhalten wir eine eindeutige Ordnung der Elemente

in unserem Rechengebiet. Eine daraus erzeugte Zellordnung wird im oberen Teil der Abbildung 3.4 gezeigt. Diese Darstellung entspricht allerdings nicht der, die wir aus der Intervallschachtelung erwarten würden. Warum z.B. hat die erste Feinzelle nicht die Nummer 2?

Da der vorliegende Löser Mehrgitterverfahren verwendet, wird auch eine Nummerierung für die Groblevelzellen benötigt. Dazu nummerieren wir diese entsprechend der Besuchsreihenfolge, die bei einer Tiefensuche durch den spacetree entsteht. Anschaulich bedeutet dies, dass wir beim Übergang von einer Feinlevelzelle in die nächste für den Fall, dass beide in unterschiedlichen Groblevelzellen liegen, erst die aktuelle Groblevelzelle verlassen, anschließend in die benachbarte Groblevelzelle eingetreten und zuletzt die neue Feinlevelzelle betreten wird. Entsprechend der Nummerierung aus Abbildung 3.4 bedeutet dies, dass beim Übergang von 11 nach 13 erst 11 und dann 2 verlassen und anschließend erst 12 und dann 13 betreten wird.

Dadurch erhalten wir eine Linearisierung der Zellen entsprechend dem Verlauf der Peanokurve und eine anschauliche Einordnung der Groblevelzellen in dieses Schema. Dadurch ist es ein leichtes, diese Zellkette gleichmäßig auf p Prozessoren zu verteilen. Wir ordnen den Prozessoren einfach gleichlange Teilabschnitte der Peanokurve zu. Eine sich daraus ergebende auf vier Prozessoren lastbalancierte Verteilung eines diskreten Rechengebietes wird in Abbildung 4.2 dargestellt.

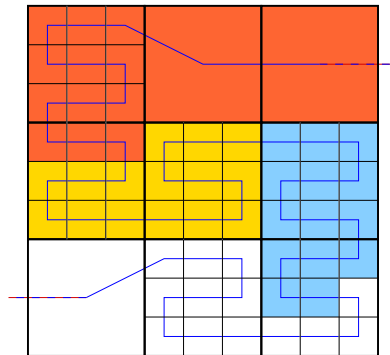


Abbildung 4.2: Gebietszerlegung in vier ausgeglichen große Berechnungspartitionen.

Da der vorliegende sequentielle Algorithmus von Markus Pögl das diskretisierte Rechengebiet entlang der Peanokurve durchläuft, ist eine derartige Partitionierung sehr einfach umzusetzen. Im Prinzip muss jeder Prozessor ausschließlich die Zellnummern der Anfangs- und der Endzelle seiner Berechnungspartition speichern. Diese werden ab sofort als *Seperatoren* bezeichnet. Die Speicherung der Gebietsaufteilung mittels Seperatoren führt folglich zu einem denkbar kleinen Speicheraufwand.

Neben der optimalen Lastbalancierung bietet die verwendete Partitionie-

rungsstrategie einen weiteren Vorteil. Wie in Abschnitt 2.6 beschrieben, ist die Oberfläche eines Peanokurvenabschnittes quasi-optimal. Das heißt, dass wir für ein gegebenes Volumen v (d.h. für eine feste Größe eines Teilgebiets) die Oberfläche durch Gleichung 2.26 nach oben abschätzen können. Zusammenfassend führt die Partitionierung mittels Peanokurve zu

- optimalem Lastausgleich
- und quasi-optimaler Oberflächengröße.

Dadurch erreichen wir, dass die parallele Ausführungszeit $T(p)$ minimal ist und erwarten für den Fall einer effizienten Umsetzung einen beinahe linearen Speedup.

4.2.2 Praktische Umsetzung

Wie zuvor erwähnt bietet sich eine Partitionierung mit Hilfe der durch die Peanokurve induzierten Ordnung an. Dazu sind einige Eingriffe in den sequentiellen Algorithmus notwendig. Die folgenden Abschnitte beschreiben die dazu notwendigen grundsätzlichen Aufgaben und deren Umsetzung.

Anpassung des Steuerungsalgorithmus'

Um während einer Iteration statt durch das gesamte diskretisierte Gebiet nur durch einen Abschnitt laufen zu können, muss offensichtlich der dafür verantwortliche Steuerungsalgorithmus angepasst werden. Dazu sind ein paar Vorüberlegungen notwendig. Was muss getan werden, damit der Algorithmus die Elementanteile nur innerhalb seiner Berechnungspartition berechnet? Die einfachste Möglichkeit wäre es, nach wie vor das gesamte Gebiet zu traversieren, aber nur im inneren der eigenen Berechnungspartition die numerischen Berechnungen durchzuführen. Ungünstigerweise verursacht alleine die Traversierung des Gebiets einen derart großen Rechenaufwand, dass die Parallelisierung kaum zu einem Geschwindigkeitsgewinn führen würde. Aus diesem Grund muss dafür gesorgt werden, dass der Algorithmus nach Möglichkeit nur die Berechnungspartition des lokalen Prozessors traversieren muss.

Abbildung 4.3(a) stellt diese Idee anhand der Berechnungspartition des zweiten Prozessors dar. Diese Idealvorstellung lässt sich leider mit den vorhandenen Datenstrukturen und der rekursiven Implementierung der Steuerungsfunktion nicht vereinbaren. Ein Grund dafür liegt in der Art und Weise der Gitterbeschreibung. Die Entscheidung, ob man eine Zelle verfeinert, wird in Form eines Flags in den Knoten eben dieser Zelle gespeichert. Dabei wird nur unterschieden, ob eine Zelle verfeinert ist oder nicht, d.h. es existieren nur entweder komplett oder überhaupt nicht verfeinerte Zellen. Damit erhalten wir die in Abbildung 4.3(b) dargestellte Traversierung.

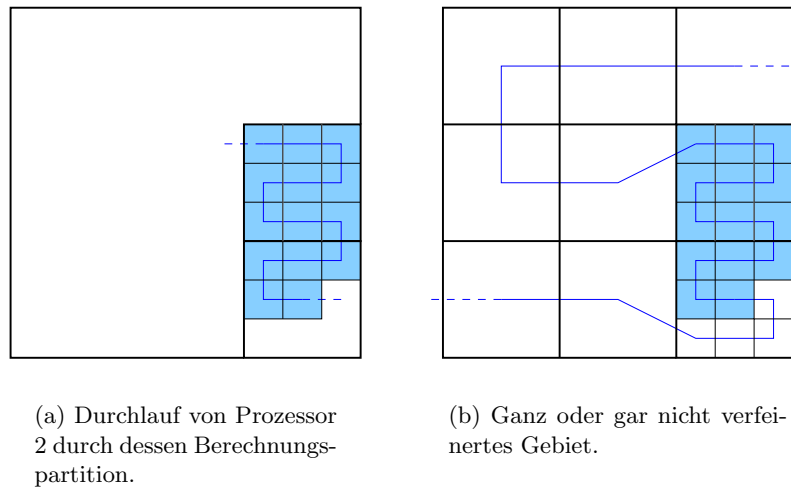


Abbildung 4.3: Teilweiser Gebietsdurchlauf von Prozessor 2.

Zieht man nun noch die Einschränkung in Betracht, dass jeder Knoten in der Berechnungspartition und auf dem Prozessrand vier mal (bzw. acht mal im 3D) verarbeitet werden muss, so stellt man für einige Randknoten eine Verletzung der Regel fest. Um dies zu vermeiden, muss man die Grobblevelzellen, die dem Prozessrand benachbart sind, auch auf dem feinsten Level durchlaufen. Nur dadurch erreicht man eine Verarbeitungsfolge der Randknoten, die der im sequentiellen Fall entspricht. Damit ergibt sich die in Abbildung 4.4(a) dargestellte Traversierung. Die rechte obere Grobblevelzelle wird darin nicht verfeinert, da das Gitter dort schon in der feinsten verlangten Verfeinerung vorliegt (vgl. Abbildung 4.2).

An diesem Punkt stellt sich die Frage, warum die Regel, jeden Knoten 4 mal verarbeiten zu müssen, nicht für die nun hinzugekommenen Blattlevelknoten außerhalb der lokalen Berechnungspartition gilt? Unter diesen gibt es nämlich jetzt ebenso einige, die nicht von 4 benachbarte Zellen umgeben sind und somit auch nicht 4 mal verarbeitet werden. Die Lösung ist einfach, führt allerdings dazu, dass weiterer Aufwand betrieben werden muss. Man speichert die außerhalb des eigenen Prozessgebiets liegenden Blattlevelknoten einfach nicht ab. Dies ist möglich, da man sie zum einen für numerische Berechnungen nicht braucht, und zudem nötig, da sie ansonsten die beschriebenen Probleme aufwerfen. Damit ergibt sich die in Abbildung 4.4(b) dargestellte Situation, in der die Traversierung samt der zu speichernden Knoten abgebildet ist. Die Gesamtheit der von einem Prozessor zu durchlaufenden Zellen, also jene innerhalb seiner Berechnungspartition aber auch jene außerhalb, werden im Folgenden mit *Durchlaufpartition* bezeichnet.

Wie wirken sich die gewonnenen Erkenntnisse auf die Anpassung des sequentiellen Steuerungsalgorithmus' aus? Dieser durchläuft das zu traversierende

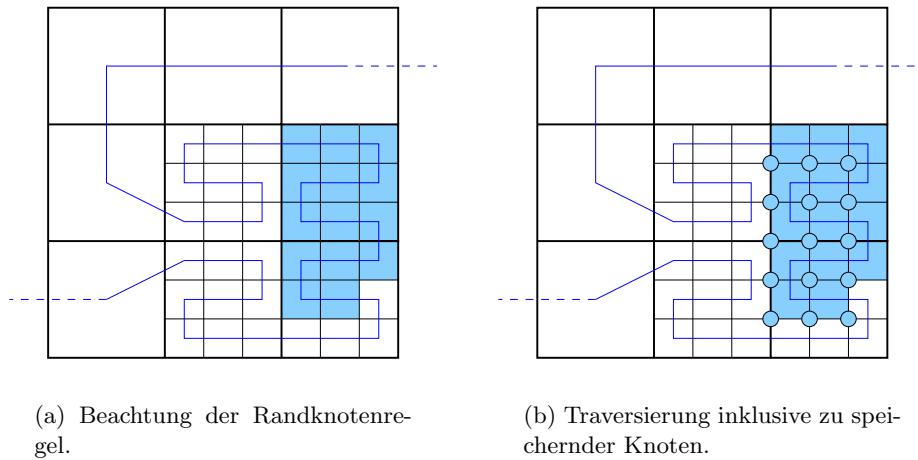


Abbildung 4.4: Teilweiser Gebietsdurchlauf von Prozessor 2.

Gebiet zellweise. In jeder Zelle liest er die Knoten ein, rechnet auf ihnen und speichert sie anschließend ab. Der parallele Steuerungsalgorithmus hingegen liest nicht mehr in jeder Zelle alle Knoten ein und führt auch nicht in jeder besuchten Zelle Berechnungen aus. Befindet sich die Traversierung gerade innerhalb der Berechnungspartition, so liest er alle Knoten und führt die Auswertung des finiten Elementsterns durch. Außerhalb führt der Algorithmus keine Berechnungen durch, da für diese Zellen andere Prozessoren zuständig sind, und liest dort auch nur die Randknoten ein. Es scheint überflüssig zu sein, keine Rechnungen durchzuführen aber dennoch die Randknoten zu lesen. Der Grund hierfür liegt in dem Aufbau der Datenstrukturen, die darauf angewiesen sind, dass die Knoten “weitertransportiert” werden. Anschaulich stellt dies Markus Pögl [Pög04] in Abbildung 4.7 dar.

Der folgende Steuerungsalgorithmus (3) erweitert die in (2) vorgestellte sequentielle Steuerung.

Die Änderungen des Algorithmus’ betreffen nur den Würzelfall, da nur dort tatsächlich Berechnungen durchgeführt und Daten gelesen werden. Der Algorithmus entscheidet in Zeile 8, ob er sich innerhalb der eigenen Berechnungspartition befindet oder nicht. Im ersten der beiden Fälle verfährt er wie im sequentiellen Algorithmus, liest also die Eckdaten ein und berechnet die Korrekturanteile. Falls sich der parallele Algorithmus jedoch außerhalb seines Gebietes befindet, wird versucht, das Restgebiet möglichst grob zu durchlaufen. D.h. nur für den Fall, dass ein Prozessrand in der Nähe existiert, wird weiter verfeinert (siehe Zeile 18). Diese Implementierung gewährleistet, dass alle notwendigen Berechnungen im Prozessgebiet durchgeführt werden. Die Zahl der durchlaufenen Zellen ist dabei so gering wie möglich, da wir nur dort verfeinern, wo dies wegen der Datenstruktur unbedingt nötig ist. Zusätzlich wird durch die Verfeinerung am Rand sichergestellt, dass alle

Algorithmus 3 Parallele Steuerungsmethode

```
1: procedure P_STEUERUNG(...)
2:   Init();
3:   for (i=0; i<3; i++) do
4:     if ( $\neg$  Würfel()) then
5:       P_Steuerung();
6:     else
7:       LeseDaten();
8:       if (inBP()) then ▷ Falls in prozesslokalem Gebiet
9:         if (blattLevel()) then
10:          BerechneResiduum();
11:        else
12:          P_Steuerung();
13:        end if
14:        BerechneMehrgitter();
15:       else ▷ Falls außerhalb der BP
16:         if ( $\neg$  blattLevel() && amPR()) then
17:           ▷ Falls Verfeinerung möglich und in der Nähe des PR
18:           P_Steuerung();
19:         end if
20:       end if
21:       SchreibeDaten();
22:     end if
23:   end for
24: end procedure
```

gespeicherten Knoten korrekt auf der Datenstruktur abgelegt werden. Damit benötigen wir noch ein Konzept für die Umsetzung der zwei in Algorithmus (3) verwendeten Methoden.

- `inBP()` - Bestimmung, ob Gebietstraversierung innerhalb des lokalen Prozesses ist oder nicht.
- `amPR()` - Ermittlung, ob aktuelle Zelle am Prozessrand liegt oder nicht.

Bestimmung der aktuellen Traversierungsposition

Wie in der Vorstellung der Partitionierung mittels raumfüllender Kurven beschrieben, reicht es vollkommen aus, die Grenzen zwischen den Prozessgebieten zu speichern. Auf den ersten Blick genügt es, die Nummer der ersten Zelle pos_{first} im eigenen Gebiet und die Anzahl der zu durchlaufenden Zellen n_i zu speichern. Der parallele Steuerungsalgorithmus müsste nur die schon traversierten Zellen pos_{akt} zählen. Solange $pos_{first} \leq pos_{akt} < (pos_{first} + n_i)$ befindet sich der Algorithmus im eigenen Gebiet. Dieser Ansatz lässt sich

allerdings nicht durchführen, da der parallele Algorithmus nicht alle Zellen durchläuft und dementsprechend auch nicht entscheiden kann, wieviele Zellen des gesamten Gebiets er schon durchlaufen hat. Wir benötigen demnach eine Beschreibung der Traversierungspositionen, die aus den Traversierungsinformationen ermittelbar ist.

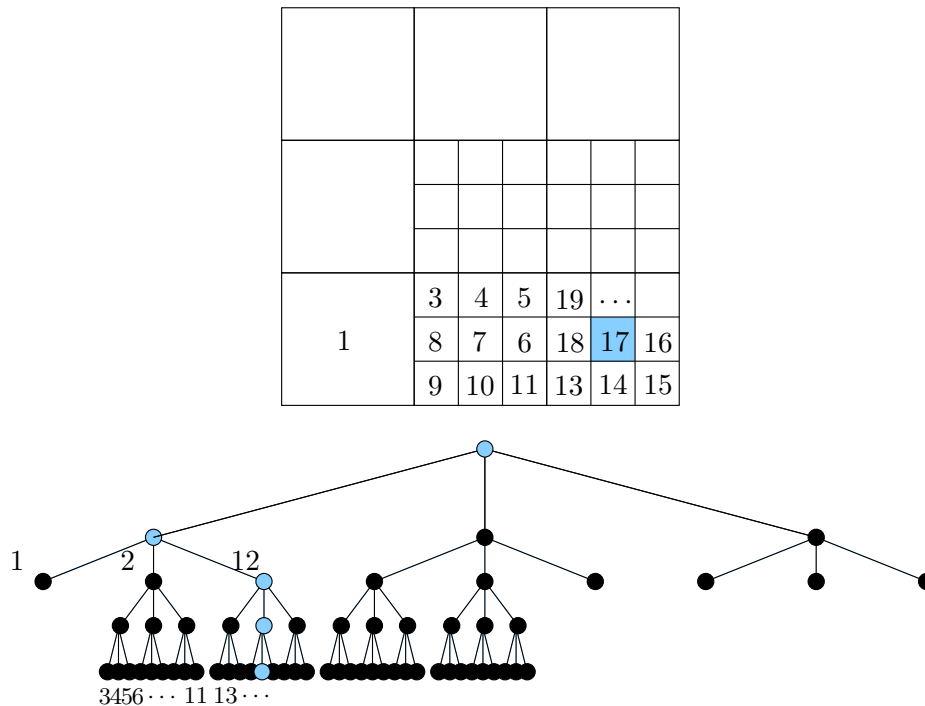


Abbildung 4.5: Bestimmung der Traversierungsposition.

Betrachten wir dazu Abbildung 4.5, so stellen wir fest, dass uns die Tiefensuche die nötigen Informationen liefert. Im vorliegenden Fall liegt die eingefärbte Feinlevelzelle (mit Nummer 17) im ersten (von 3) Balken des gesamten Gebiets. Innerhalb des Balkens liegt sie in der dritten Groblevelzelle, darin im zweiten Balken und ist dort schließlich selbst die zweite Zelle. Diese Informationen besitzt auch der Steuerungsalgorithmus, da die einzelnen Positionen genau den aktuellen Zählvariablen der `for()`-Schleifen in den jeweiligen rekursiven Aufrufen entsprechen. Also stellt die Folge $f_p(17) = 1322$ exakt die Position der eingefärbten Zelle dar und kann einfach aus den Zählerständen der Schleifen ermittelt werden. Zusätzlich ist es für diese Positionsangabe jetzt nicht mehr notwendig, sämtliche davor liegenden Zellen durchlaufen zu müssen. Die einzelnen Ziffern der Positionsangabe beziehen sich auf die Durchlaufreihenfolge bzgl. der Peanokurvenrichtung, weswegen die Funktion und die entstandene Ziffernfolge als Index ein kleines p bekommen. Diese Form der Positionsangabe können wir nun zum Abspeichern der Grenzzellen nutzen. Weiterhin wird im Folgenden eine Zelle durch einen

kleinen Buchstaben (z.B. g) gekennzeichnet, die Nummer der Zelle durch n_g und die Position auf der Peanokurve durch $f_p(n_g)$. Falls von konkreten Zahlen gesprochen wird, bekommt die Peanokurvenposition, um sie von der Zellnummer unterscheiden zu können, den Index p .

Betrachten wir die peanokurvenbezogene Positionsangabe noch einmal. Dabei stellen wir fest, dass die Reihenfolge der Ziffern der Tiefe der damit bezeichneten Gitterelemente (Platte, Balken, Würfel) im spacetree entspricht. D.h. je weiter vorne eine Ziffer in der Positionsangabe, desto weiter oben liegt das damit durchgezählte Gitterelemente im spacetree. Damit bedeutet das Fortschreiten entlang der Peanokurve, also das Erhöhen einer der Schleifenzählvariablen, automatisch eine Erhöhung von f_p . Die beiden auf die Zelle 17 folgenden Zellen haben die Bezeichnungen $f_p(18) = 1323_p$ bzw. $f_p(19) = 1331_p$ und es ist durch einfachen Größenvergleich der Peanokurvenpositionen möglich, ihre Reihenfolge zu ermitteln.

Unser ursprüngliches Problem war, dass wir während eines Durchlaufes die Information benötigten, ob man sich innerhalb oder außerhalb der lokalen Berechnungspartition befindet. Obige Darstellung ermöglicht es uns nun schon, Zellen auf dem feinsten Level zu vergleichen, ohne alle vorher durchlaufenen Zellen zählen zu müssen.

Da der Algorithmus ein additives Mehrgitterschema implementiert, ist es notwendig, in allen Groblevelzellen Berechnungen durchzuführen. Für die parallele Umsetzung bedeutet dies, dass in jeder Groblevelzelle Berechnungen durchzuführen sind, in der wenigstens eine Blattzelle zur eigenen Berechnungspartition gehört. Um nun auch für Groblevelzellen bestimmen zu können, ob sie sich innerhalb der Berechnungspartition befinden, benötigen wir zuerst eine eindeutige Zahlenfolge, um deren Position zu beschreiben. Anschließend muss eine Ordnung definiert werden, die sowohl für Grob- als auch für Feinlevelzellen gilt.

Die Positionszahlenfolgen der Groblevelzellen werden analog zu denen der Blattzellen gebildet. Damit bekommt die Groblevelzelle 12 die Position $f_p(12) = 13_p$. Diese Positionsangabe ermöglicht jetzt zwar einen Vergleich zwischen Groblevelzellen auf einem Level, allerdings ist ein Vergleich z.B. mit der Positionsangabe $f_p(5) = 1213_p$ nicht sinnvoll. Um dies zu ermöglichen, wird die Positionsangabe der Groblevelzellen bis auf die Länge der Feinlevelzellen mit Nullen ergänzt. Wir erhalten $f_p(12) = 1300_p$ und damit liegt Zelle 12 auch hinsichtlich ihrer Positionsangabe eindeutig hinter Zelle 5.

Nun sind wir fast in der Lage zu bestimmen, ob sich die aktuelle Position der Traversierung innerhalb oder außerhalb der lokalen Berechnungspartition befindet. Wir müssen nur testen, ob die Position der aktuellen Zelle innerhalb der beiden Partitionsgrenzzellen liegt. Einzig die Groblevelzellen, die diese Grenzzellen beinhalten, sind noch nicht eindeutig einzuordnen.

Betrachten wir Abbildung 4.2, so stellen wir fest, dass sich die erste (hellblaue) Blattzelle (entspricht Zelle 17) von P_2 innerhalb einer Groblevelzelle befindet, die verschieden gefärbte Zellen enthält. In dieser Groblevelzelle

müssen demnach von Prozessor 2 und von Prozessor 1 Berechnungen durchgeführt werden. Die Positionen der genannten Fein- und Groblevelzelle sind $f_p(17) = 1322_p > 1300_p = f_p(12)$. Speichert man nun $f_p(17) = 1322_p$ als Beginn von P_2 , so bedeutet dies, dass die Groblevelzelle 12 nicht zur lokalen Berechnungspartition von P_2 gehört und auf dieser auch keine Mehrgitterberechnungen durchgeführt werden. Dies legt nahe, dass die Ordnung, welche festlegt, ob eine Zelle innerhalb der Partitions Grenzen liegt, nicht über einen reinen Größenvergleich der eingeführten Peanokurvenpositionen definierbar ist. Wir halten die Anforderungen an diese Ordnung zuerst einmal informell fest.

- Blattzellen liegen innerhalb von P_i , wenn ihre peanokurvenbezogenen Positionsangaben zwischen denen der Separatorzellen von P_i liegen.
- Eine Groblevelzelle g liegt dann innerhalb von P_i , wenn mindestens eine der beiden Separatorzellen innerhalb von g liegt oder wenn g eindeutig, d.h. in der Darstellung $f_p(g)$, zwischen den Separatorzellen liegt.
- Als Separatorzellen werden nur Blattzellen zugelassen.

Die letzte Festlegung wurde getroffen, um die Zahl der Fallunterscheidungen klein zu halten, und stellt keine wesentliche Einschränkung bei der Partitions-grenzwahl dar.

Um den Sachverhalt zu formalisieren, führen wir eine Notation ein, die die Positionsangaben verallgemeinert. Dabei werden Positionsangaben der Groblevelzellen ab sofort mit G aufgefüllt, um deutlich zu machen, dass sie nicht direkt mit Blattzellenpositionen zu vergleichen sind. D.h. $f_p(12) = 13GG$. Zusätzlich wollen wir Positionsangaben hinsichtlich ihres Levels im space-tree einordnen können. Wir schreiben $f_p^{i,j}$ für alle Positionen mit i Stellen, bei der die ersten j Stellen ungleich G sind. Demzufolge beinhaltet $f_p^{4,4}$ die Nummern aller Blattzellen auf einem maximal zweifach verfeinerten Gebiet. Zusätzlich liefert $p(f_p, k)$ die ersten k Stellen einer Positionsdarstellung. Damit definieren wir eine Ordnung $<_p$ auf der zuvor eingeführten Zellpositionsdarstellung:

$$\begin{aligned}
f_p(n_x) <_p f_p(n_y) &\Leftrightarrow f_p(n_x) \in f_p^{i_1, j_1} \wedge f_p(n_y) \in f_p^{i_2, j_2} \\
&\wedge j = \min\{j_1, j_2\} \wedge i_1 = i_2 \\
&\wedge p(f_p(n_x), j) < p(f_p(n_y), j)
\end{aligned} \tag{4.2}$$

$$\begin{aligned}
f_p(n_x) =_p f_p(n_y) &\Leftrightarrow f_p(n_x) \in f_p^{i_1, j_1} \wedge f_p(n_y) \in f_p^{i_2, j_2} \\
&\wedge j = \min\{j_1, j_2\} \wedge i_1 = i_2 \\
&\wedge p(f_p(n_x), j) = p(f_p(n_y), j)
\end{aligned} \tag{4.3}$$

Die Gleichungen (4.2) und (4.3) vergleichen zur Bestimmung der Ordnung ausschließlich die ersten j Ziffern der Positionsangabe. Wenn sich aus diesen

keine eindeutige Reihenfolge (4.2) ableiten lässt, also beide identisch sind, oder eine der Zellen, deren Positionsangaben verglichen werden, in der anderen liegt, werden die beiden Positionsangaben als gleich interpretiert (4.3). Damit kann man mit

$$f_p(\textit{first}) \leq_p f_p(\textit{akt}) \leq_p f_p(\textit{first} + n_i)$$

bestimmen, ob die Positionsangabe der aktuellen Zelle $f_p(\textit{akt})$ innerhalb der Berechnungspartition P_i liegt.

Speicherung des Gebietsrandes

Nachdem wir mit dem zuvor vorgestellten Ansatz in der Lage sind `inBP()` zu implementieren, fehlt noch die Implementierung von `amPR()`. Diese Funktion soll zurückgeben, ob die aktuelle Zelle den Rand einer Partition berührt. Dazu benötigen wir in jeder Zelle nur die Information, ob einer der Eckknoten auf dem Rand liegt. Dementsprechend sollte es ausreichen, ein Bit (Randbit) für jeden Knoten zu speichern, welches anzeigt, ob ein Knoten auf dem Rand liegt oder nicht. Mit dessen Hilfe kann nach dem Lesen der Eckknoten einer Zelle einfach bestimmt werden, ob diese Zelle den Prozessrand berührt.

Ungünstigerweise werden Randknoten und solche, die innerhalb der Berechnungspartition liegen, aus Effizienzgründen auf unterschiedlichen Datenstrukturen gespeichert. Deshalb ist es notwendig zu wissen, ob ein Knoten auf dem Rand liegt oder nicht, bevor dieser gelesen wird. Es bleibt uns nichts weiteres übrig, als die Randbits schon in den Knoten der Groblevelzellen zu speichern. Diese Informationen liegen dann beim Zugriff auf deren Kinder schon vor und können ausgewertet werden, bevor die Knoten der Feinlevelzellen gelesen werden. Da jede Zelle im ungünstigsten Fall nur einen Eckknoten besitzt, welcher auf der Datenstruktur gespeichert wird, muss dieser alle Randbits der Zelle speichern. Für das Zweidimensionale ist in Abbildung 4.6 ein solcher Fall skizziert, bei dem bis auf einen alle Eckknoten hängende Knoten sind. Der mit dem Pfeil markierte Knoten ist im dargestellten Fall der einzige Eckknoten der oberen linken Grobzelle, der nicht hängend und damit in der Datenstruktur gespeichert ist. Dieser muss demnach den Grenzverlauf innerhalb dieser Zelle komplett wiedergeben. Aus der Implementierung von Markus Pögl [Pög04] lässt sich eine wesentliche Aussage gewinnen: Die Feinlevelknoten auf einer Kante einer verfeinerten Zelle sind genau dann keine hängenden Knoten, wenn die zur Kante inzidenten Groblevelknoten ein Tiefenflag besitzen (d.h. eine Verfeinerung anordnen) und demnach auf dem Keller gespeichert sind. Im Umkehrschluss heißt dies, dass jeder Groblevelknoten die gesamte Grenzverlaufsinformation einer Zelle speichern muss, mit Ausnahme der Randbits der an den gegenüberliegenden Kanten liegenden Feinlevelknoten. Die Randbits dieser Knoten können nämlich (nach vorheriger Aussage), wenn diese keine hängenden Knoten sind,

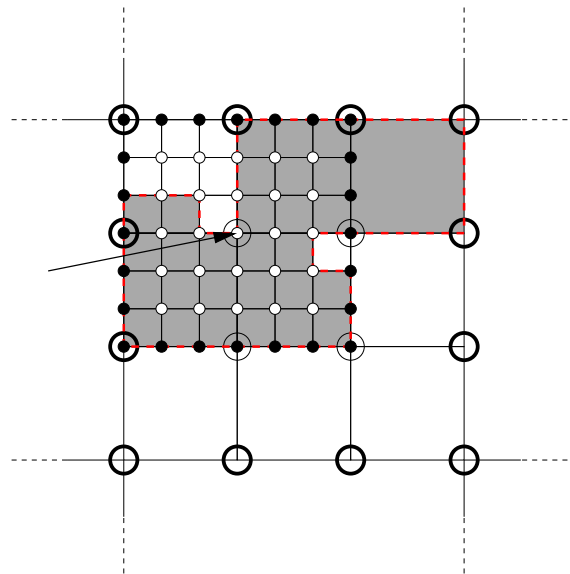


Abbildung 4.6: Mögliche Knotenverteilung.

von einem dann sicher existierenden näher liegenden Groblevelknoten gespeichert werden. Damit ergeben sich für eine Zelle die in Abbildung 4.7(a) dargestellten Bereiche, über die die einzelnen Knoten, wenn sie existieren, Randinformationen speichern müssen.

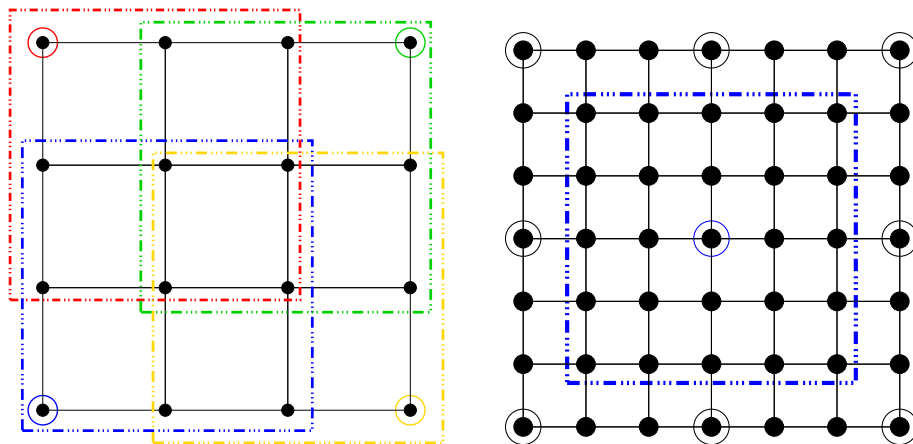
Der markierte Knoten aus Abbildung 4.6 muss also im dargestellten zweidimensionalen Fall 9 Randbits für die obere linke Grobzeile speichern. Die Belegung sieht dabei wie folgt aus.

0	0	1
1	1	1
0	1	1

Dementsprechend liegen die drei mit Nullen versehenen Knoten nicht auf dem Rand, was mit dem tatsächlichen Randverlauf übereinstimmt.

Abbildung 4.7(b) zeigt, welchen Bereich ein Knoten im Zweidimensionalen abdecken muss, um sämtliche Randinformationen aller benachbarter Zellen speichern zu können. Insgesamt ergibt sich ein Speicherbedarf von 125Bit (also 16Byte) im Dreidimensionalen. Dies erscheint im ersten Moment eine beträchtliche Menge, relativiert sich allerdings, da diese Informationen nur in Groblevelknoten, die in der Nähe des Prozessrandes liegen, gespeichert werden müssen. Abschnitt 5.2 gibt einen Überblick, wie sich der für den parallelen Algorithmus nötige Platzbedarf im Vergleich zu dem des sequentiellen Algorithmus' verhält.

Damit können wir den exakten Verlauf des Prozessrandes auf dem feinsten Level speichern. Aufgrund des verwendeten additiven Mehrgitterschemas benötigt man zusätzlich aber auch noch eine exakte Beschreibung des Randes



(a) Bereiche, über die die Knoten einer Zelle Randinformationen speichern.

(b) Bereich, über den ein Knoten Randinformationen speichert.

Abbildung 4.7: Knotenbezogene Randinformationen.

auf den größeren Leveln. Der Grund dafür liegt in dem rekursiven Aufbau des Traversierungsalgorithmus'. Um außerhalb des Berechnungsgebiets den Prozessrand auf feinstem Level durchlaufen zu können, benötigen wir die Information, dass an dieser Stelle verfeinert werden muss, schon in allen Leveln darüber. Dies erreichen wir, indem wir alle Knoten als Randknoten auffassen, die von mindestens zwei verschiedenen Prozessoren benötigt werden. Dadurch besitzen Grobknoten in Randnähe ein Randbit und der parallele Algorithmus verfeinert die angrenzenden Grobzellen. Diese Randbits werden erneut im jeweils darüber liegenden Level gespeichert. Da es in jedem Level wesentlich weniger Knoten gibt als im jeweiligen darunter liegenden, ist der gesamte Speicherplatzverbrauch für die Speicherung der Prozessränder auf allen Leveln zusammen kaum größer als der Aufwand für das feinste Level. Da dieser schon sehr klein war, kann man von einer sehr effizienten Speicherung der Randinformationen sprechen.

4.3 Kommunikation

4.3.1 Theoretischer Ansatz

Im vorherigen Abschnitt wurde gezeigt, wie man durch die Einführung einer geeigneten Nummerierung und der Speicherung von Randinformationen in der Lage ist, effizient durch Teilstücke eines Berechnungsgebiets zu traversieren. Der folgende Abschnitt beschreibt die notwendige Kommunikation zwischen den Prozessoren. Um einen effizienten parallelen Algorithmus zu

erhalten, ist es besonders wichtig, die dafür aufgewendete Zeit so klein wie möglich zu halten. Dazu benötigen wir detaillierte Kenntnisse über die auszutauschenden Daten.

Im Abschnitt 4.2.1 wurden schon Überlegungen darüber angestellt, welche Daten zwischen den Prozessoren ausgetauscht werden müssen. Aus der verwendeten elementweisen Finite-Element-Diskretisierung erhält man lineare Gleichungssysteme, die in jedem Element (jeder Zelle) gelöst werden müssen. Mit Hilfe des verwendeten Jacobi-Verfahrens löst man diese Gleichungssysteme näherungsweise, indem in jeder Iteration ein Korrekturterm wie in Gleichung (2.15) zu jedem Eckknoten addiert wird. Die Aufgabe einer Gebietstraversierung besteht also darin, in jeder Zelle die Korrekturbeiträge für deren Eckknoten zu bestimmen und, nachdem ein Knoten alle Korrekturterme besitzt, diese Korrektur zur Lösung zu addieren.

Abbildung 4.8(a) stellt diesen Sachverhalt bei einer Gebietsaufteilung in zwei Berechnungspartitionen dar. Dabei markieren die beiden unterschiedlichen Farben die Elemente, in denen der jeweilige Prozessor seine Korrekturanteile berechnet. Damit besitzen die auf dem Rand liegenden Knoten am Ende einer Iteration noch nicht alle Korrekturanteile, da jeder Prozessor nur die in seinem Gebiet liegenden Anteile berechnet hat.

Ganz ähnlich ist dies bei den Grobzellen. Wegen des verwendeten additiven Mehrgitterschemas berechnet jeder Prozessor für die Grobzellen Korrekturanteile, in denen er auch Blattzellenkorrekturen berechnet. Diese Grobzellenkorrekturen werden erneut auf die Knoten der Grobzellen verteilt. Damit ergibt sich der in Abbildung 4.8(b) dargestellte Sachverhalt, der die selbe Verteilung der Feinzellen zugrunde legt, wie Abbildung 4.8(a). Die zweifarbig gezeichnete Zelle wird von beiden Prozessoren teilweise durchlaufen. Dadurch liegen auf beiden Prozessoren auch Korrekturanteile in den Eckknoten dieser Zelle. Also müssen wir wie im Blattlevelfall sowohl Korrekturanteile von Eckknoten zwischen einfarbigen, also nur von einem Prozessor durchlaufenen, Grobzellen austauschen, als auch die Korrekturanteile von mehrfarbigen Grobzellen kommunizieren.

4.3.2 Praktische Umsetzung

Die Kommunikationsphase umfasst zwei wesentliche Abschnitte. Zuerst muss die Knotenmenge ermittelt werden, die ein Prozessor an einen anderen verschickt. Anschließend findet die tatsächliche Kommunikation statt, in der die Pakete verschickt werden. Wie diese beiden Aufgaben möglichst effizient umgesetzt werden können, beschreiben die beiden folgenden Abschnitte.

Dazu definieren wir zuerst all jene Knoten als Randknoten, deren Korrekturanteile auf verschiedenen Prozessoren berechnet werden. Dies entspricht exakt der in Abschnitt 4.2.2 vorgestellten Definition von Randknoten. Jeder Prozessor i besitzt also eine bestimmte Menge R_i an Randknoten, wobei verschiedene andere (entfernte) Prozessoren an einer Teilmenge davon inter-

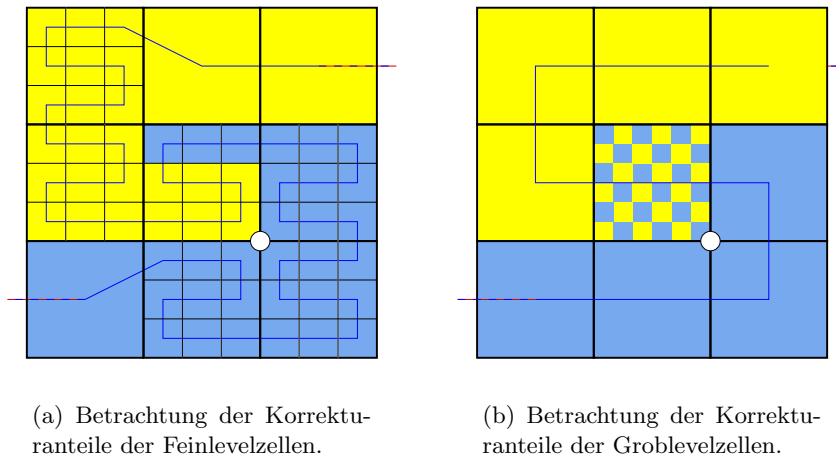


Abbildung 4.8: Verarbeitung der Korrekturanteile.

essiert sind. Wir bezeichnen die Teilmenge von R_i , an der der Prozessor j interessiert ist, mit $R_{i \rightarrow j}$.

Damit bezeichnet $\bigcup_{i=1}^p R_{i \rightarrow j}$ die Menge an Knoten, an der Prozessor j interessiert ist. Verschickt also jeder Prozessor i nur $R_{i \rightarrow j}$ an Prozessor j , so stellt man sicher, dass keine überflüssige Kommunikation stattfindet.

Entsprechend sieht der Ablauf eines Kommunikationszykluses für Prozessor i wie folgt aus:

1. Lokale Randknoten R_i in Teilmengen $R_{i \rightarrow j}$ aufteilen.
2. $R_{i \rightarrow j}$ an j verschicken.
3. $R_{j \rightarrow i}$ von j empfangen.
4. Addition der Korrekturanteile von $R_{j \rightarrow i}$ zu lokalen Korrekturanteilen.

Zuerst beschreiben wir die Punkte 1 und 4, da diese sich mit dem Aufbau der Datenstrukturen beschäftigen. Bei den Punkten 2 und 3, auf die wir anschließend eingehen, handelt es sich um den eigentlichen Datenaustausch, also um den physischen Transfer der Daten.

Ermittlung der zu versendenden Datenpakete

Die Ausgangssituation nach dem einmaligen Durchlauf durch das Rechengebiet wird in Abbildung 4.9 dargestellt. Bei diesem Szenario erkennen wir, dass jeder Prozessor seine Daten auf zwei Keller aufgesplittet hat. Der erste Keller beinhaltet die Daten der Knoten, die komplett innerhalb des Berechnungsgebiets liegen. Diese Knoten besitzen alle Residuumsanteile, da der jeweilige lokale Prozessor alle Anteile selbstständig berechnen konnte. Auf

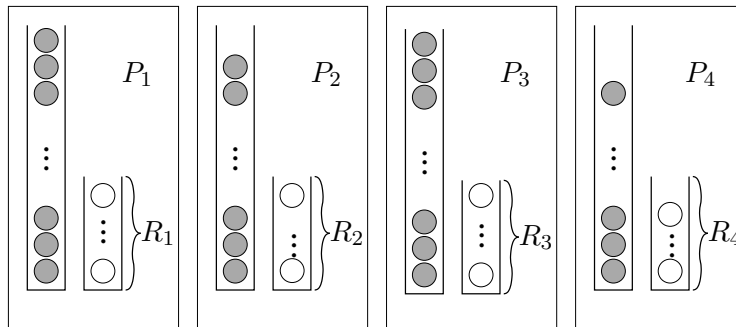


Abbildung 4.9: Ausgangssituation vor der Kommunikation.

dem zweiten Keller liegen sämtliche, sich auf dem Prozessrand befindenden Knoten. Den dort liegenden (weißen) Knoten fehlt mindestens ein Korrekturanteil von mindestens einem anderen Prozessor. Anders gesagt besitzt jeder Prozessor Korrekturanteile, die ein anderer benötigt. Um seine berechneten Anteile verteilen zu können, erstellt also jeder Prozessor i die Teilmengen $R_{i \rightarrow j}$, die er an die anderen Prozessoren verschicken muss. Diese Teilmengen müssen nicht paarweise disjunkt sein, da Knoten auf dem Rand mehrerer Prozessoren liegen können (vgl. Abbildung 4.2).

Entscheidend ist nun die Frage, wie ein Prozessor entscheiden kann, an welche Prozessoren er seine Randknoten schicken soll. Dazu muss er zuerst ermitteln, welchen Elementen (Zellen) dieser Knoten angehört, und anschließend herausfinden, welcher Prozessor diese Zellen bearbeitet. Dies lässt sich erneut anhand der in Abschnitt 4.2.2 eingeführten Peanokurvennummerierung ermitteln. Diese Nummer identifiziert eindeutig die Traversierungsposition einer Zelle innerhalb eines Gitterlevels. Sofern die Anfangs- und Endzellpositionen aller Prozessoren bekannt sind, muss nur bestimmt werden, ob die entsprechenden dem Knoten benachbarten Elemente zwischen diesen Grenzen liegen. Dieser Vergleich ist mit Hilfe der in Abschnitt 4.2.2 eingeführten Ordnung einfach möglich. Einzig die Ermittlung der Elemente, denen der betreffende zu verschickende Knoten angehört, ist nicht einfach umzusetzen. Abbildung 4.10 zeigt ein einfaches Beispiel, welches die Problematik verdeutlicht. Der weiße eingezeichnete Feinknoten liegt auf den Randkellern der ersten drei Prozessoren. Im Folgenden betrachten wir den Fall, dass Prozessor 1, der den Anteil in Zelle 5 berechnet hat, die Prozessoren ermitteln möchte, an die er eben diesen Feinknoten schicken muss. D.h., dass er ausgehend von der Positionsangabe der Zelle 5 auf die Positionsangabe der Zellen 19, 25 und 39 schließen muss.

Halten wir erst einmal die Positionsangaben der 4 beteiligten Zellen fest. Dazu wechseln wir von der bekannten Darstellungsweise in eine der Ternär-darstellung der Peanokurve ähnlicheren. Jede Ziffer z wird einfach durch $z - 1$ ersetzt, wodurch wir eine äquivalente Darstellung in etwas anderer

Dabei wird ausgenutzt, dass die ungeraden Stellen der Positionsangabe gerade die Informationen über die Lage in y-Richtung besitzen, wohingegen die gerade Stellen die Lageinformation in x-Richtung beinhalten. Diese Darstellung entspricht im Wesentlichen der diskreten Ternärdarstellung der Zelle 5. Deren Nachbarzellen lassen sich nun denkbar einfach finden. Zählt man um 1 in x-Richtung weiter, so erhält man unter Beachtung des Übertrags die koordinatenbezogene Positionsangabe von Zelle 19

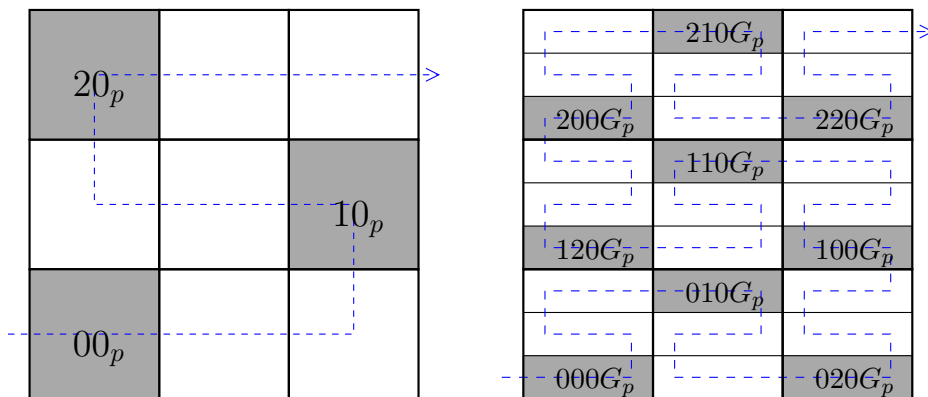
$$(02, 12)_k + (00, 01)_k = (02, 20)_k = f_k(19)$$

Wäre also eine Funktion $t : pos_p \rightarrow pos_k$ bekannt, die die Peanokurvenposition in eine koordinatenbezogene Darstellung transformiert, so könnten die Koordinaten der Nachbarzellen durch einfache Addition leicht ermittelt werden. Anschließend müsste man nur noch die Koordinatenpositionen der Nachbarzellen mit der Umkehrfunktion t^{-1} zurücktransformieren und prüfen, ob sie innerhalb der Grenzen eines Prozessors liegen.

Eine solche Funktion t lässt sich angeben. Betrachten wir dazu die peanokurvenbezogenen Positionsangaben. Wie oben schon für die koordinatenbezogene Angabe der Position festgestellt, beinhalten die erste und die dritte Stelle (also die ungeraden Stellen) Informationen über die Lage der Zellen in y-Richtung. Die geraden Stellen beschreiben hingegen die Lage in x-Richtung. Dabei stellen wir fest, dass die Information “im zweiten Element” egal für welche Position immer das mittlere der drei möglichen Elemente meint. Die Angaben “das erste” bzw. “das dritte” hingegen sagt ohne die Kenntnis des Peanokurvenverlaufs nichts darüber aus, wo das entsprechende Element liegt. Abbildung 4.11(a) stellt dar, wo die jeweilige Zelle im Raum liegt, wenn die x-Position konstant bei 0 belassen wird und die y-Position sich ändert. Dabei stellt man fest, dass die eingefärbte Zelle immer dann in x-Richtung an erster Position liegt, also mit der peanokurvenbezogenen Angabe übereinstimmt, wenn wir in y-Richtung zuvor das erste oder das letzte Element gewählt haben. Dieser Sachverhalt erschließt sich auch aus der von Markus Pögl [Pög04] vorgenommenen Formalisierung der Peanokure. Entscheidend dabei ist die Tatsache, dass die Durchlaufrichtung im ersten und im dritten Teilelement gleich und zudem auch noch identisch zum Elternelement ist. Nur im mittleren Teil kehrt sich die Durchlaufrichtung um.

Betrachtet man nun das selbe Problem statt mit konstanten x-Richtungen mit konstanten y-Richtung, so ergibt sich das in Abbildung 4.11(b) dargestellte Bild. Erneut wählen wir dazu das erste Element in y-Richtung (das erste auf dem feinsten Level) und beobachten, wann dieses tatsächlich auch in der koordinatenbezogenen Darstellung an erster Stelle liegt. Wie zuvor stellen wir fest, dass dies immer der Fall ist, es sei denn, man hat in x-Richtung das mittlere Element gewählt. Wieder lässt sich dies mit der Umkehrung der Durchlaufreihenfolge erklären.

Führt man diese Beobachtungen fort, so erkennt man, dass sich durch eine zweifache Wahl des mittleren Elements auch die Durchlaufrichtung zweimal



(a) Veränderungen in y-Richtung.

(b) Veränderungen in x-Richtung.

Abbildung 4.11: Zuordnung Peanokurvenposition und tatsächlicher Lage im Raum.

ändert, also in ihrer Ausgangsrichtung bleibt. Damit ist für die Position der Elemente in x-Richtung nur noch interessant, ob in y-Richtung das mittlere Element ungerade oft oder gerade oft ausgewählt wurde. Entsprechend gilt dies für die y-Richtung umgekehrt.

Führt man diese Beobachtung nun im Dreidimensionalen fort, so stellt man fest, dass sich die Durchlaufrichtung in x-Richtung jedesmal ändert, wenn in y- oder in z-Richtung das mittlere Element gewählt wurde. Analoges gilt für die zwei verbleibenden Raumrichtungen. Fasst man alle Erkenntnisse zusammen, erhält man den auf Seite 62 dargestellten Algorithmus (4).

Algorithmus (4) wandelt ein mit peanokurvenbezogenen Positionsangaben gefülltes Array `srcKoord` in koordinatenbezogene Positionsangaben um und speichert diese in `destKoord`. Der Name des Algorithmus' ist bewusst allgemein gewählt, da die Funktion zu sich selbst invers ist und demnach sowohl die Hin- als auch die Rücktransformation zwischen den beiden Darstellungsmöglichkeiten durchführt.

Mit der nun bekannten Transformationsmethode haben wir alle benötigten Werkzeuge in der Hand, um ermitteln zu können, an welche Prozessoren ein Randknoten verschickt werden muss. Zuerst ermitteln wir die peanokurvenbezogene Positionsangabe $f_p(g)$ einer im eigenen Berechnungsgebiet liegenden Zelle g , die einen Eckknoten e besitzt, welcher auf dem Prozessrand liegt. Diese Position wird in die koordinatenbezogene Darstellung umgewandelt und wir erhalten $T(f_p(g)) = f_k(g)$. Durch einfache Addition können daraus die Positionen $n[7]_k$ der (im Dreidimensionalen) sieben Nachbarzellen bestimmt werden, denen e angehört. Diese Positionen können nun erneut in peanokurvenbezogene Darstellung $n[7]_p$ umgewandelt werden. Für diese

Algorithmus 4 Transformationsmethode zwischen koordinatenbezogener und peanokurvenbezogener Positionsangabe

```
1: procedure T(int srcKoord[], int destKoord[], int length)
2:   int xCount = yCount = zCount = 0;
3:   for (int i = 0; i < length; i++) do
4:     posID = i % 3;    ▷ Ermitteln, ob i-te Stelle x,y oder z-Angabe
5:     if (srcKoord[i] == 1) then
6:       destKoord[i] = 1;
7:       if (posID == _X_POS) then
8:         xCount++;
9:       end if
10:      if (posID == _Y_POS) then
11:        yCount++;
12:      end if
13:      if (posID == _Z_POS) then
14:        zCount++;
15:      end if
16:    else
17:      if (posID == _X_POS) then
18:        if ((yCount + zCount)%2) then
19:          destKoord[i] = 2 - srcKoord[i];
20:        else
21:          destKoord[i] = srcKoord[i];
22:        end if
23:      end if
24:      if (posID == _Y_POS) then
25:        if ((xCount + zCount)%2) then
26:          destKoord[i] = 2 - srcKoord[i];
27:        else
28:          destKoord[i] = srcKoord[i];
29:        end if
30:      end if
31:      if (posID == _Z_POS) then
32:        if ((xCount + yCount)%2) then
33:          destKoord[i] = 2 - srcKoord[i];
34:        else
35:          destKoord[i] = srcKoord[i];
36:        end if
37:      end if
38:    end if
39:  end for
40: end procedure
```

Darstellungsweise haben wir zuvor eine Ordnung angegeben, mit deren Hilfe sich die Prozessoren bestimmen lassen, denen die Nachbarzellen angehören. Beachten muss man dabei noch, dass Grobzellen zu den Berechnungspartitionen mehrerer Prozessoren gehören können. Alle so ermittelten Prozessoren sind also am Eckknoten e interessiert, der deshalb an sie verschickt werden muss.

Nachdem jeder Prozessor seine Randknoten so sortiert hat, dass er nur noch solche an seine Nachbarprozessoren schickt, die auch von diesen benötigt werden, muss nun noch geklärt werden, wie er die an ihn adressierten Knoten weiterverarbeitet. Wir können zu diesem Zeitpunkt also davon ausgehen, dass er sämtliche noch benötigten Randknotendaten von seinen Nachbarprozessoren geschickt bekommen hat. Nun muss er diese Daten noch seinen eigenen Randknotendaten zuordnen. D.h. für jeden empfangenen Knoten aus $\bigcup_{i=1}^p R_{j \rightarrow i}$ muss Prozessor i ermitteln, auf welchen Knoten aus R_i er den empfangenen Anteil kumulieren muss. Um dies zu ermöglichen, speichern wir die peanokurvenbezogene Zellposition der Zelle für jeden Knoten ab, in der er das letzte mal bearbeitet wird. Diese Zelle ist für einen beliebigen Knoten auf allen Prozessoren die selbe, da alle Prozessoren den selben Weg entlang der Peanokurve laufen. Damit stellt sie einen eindeutigen Identifikator für den Knoten dar. Da es aber Zellen gibt, in denen mehrere Knoten das letzte mal verarbeitet werden, reicht die Zellnummer alleine noch nicht aus. Daher wird zusätzlich noch eine Nummer gespeichert, die den Knoten innerhalb dieser Zelle eindeutig identifiziert. Dieser Identifikator wird so kompakt wie möglich gespeichert und belegt dennoch im Speicher letztendlich 13 Byte, was neben dem Korrekturanteil (Residuum) den größten Anteil bei der Speicherung der Randknoten ausmacht.

Versenden der Datenpakete

Im vorangegangenen Abschnitt wurde beschrieben, wie ein Prozessor i die Knotenmenge R_i , die auf dessen Randkeller liegt, in Pakete $R_{i \rightarrow j}$ aufteilt, die er an seine Nachbarprozessoren j schicken muss. Diese Pakete müssen nun möglichst effizient über das vorhandene Netzwerk an die jeweiligen Prozessoren verschickt werden. Dabei sind wir auf die von MPI definierten und in Abschnitt 3.2 beschriebenen Direktiven festgelegt. Die einfachste Herangehensweise ist es, der Reihe nach immer einen der Prozessoren als Empfänger und sämtliche anderen Prozessoren als Sender fungieren zu lassen. Damit ergibt sich die in Algorithmus (5) dargestellte vereinfachte Implementierung, die davon ausgeht, dass an sämtliche Prozessoren Daten verschickt werden. Der Verlauf der Kommunikation ist denkbar einfach. Zuerst wird Prozessor P_p als Empfänger ausgesucht und alle Prozessoren senden diesem ihre Daten. Anschließend fungiert Prozessor P_{p-1} als Empfänger und empfängt die Daten aller Prozessoren. Einzig und allein Prozessor P_p ist noch nicht in der Lage seine Daten zu versenden, da er zu diesem Zeitpunkt noch Daten

Algorithmus 5 Paketversende- und Empfangsalgorithmus

```
1: procedure TAUSCHDATENAUS
2:   for (empfNum = p; empfNum >= 1; empfNum--) do
3:     if (empfNum ==  $P_i$ ) thenSumme
4:       for (sendNum = p; sendNum >= 1; empfNum--) do
5:         if (empfNum !=  $P_i$ ) then
6:           MPIRecv( $R_{sendNum \rightarrow P_i}$ , sendNum);
7:         end if
8:       end for
9:     else
10:      MPISend( $R_{P_i \rightarrow empfNum}$ , empfNum);
11:    end if
12:  end for
13: end procedure
```

empfängt. Damit wartet P_{p-1} erst auf die Vervollständigung des Empfangszyklus von P_p . Damit muss P_{p-2} , der als nächster an der Reihe ist, Pakete zu empfangen, auf beide vorher genannte Prozessoren warten, usw. Diese Beobachtung legt es nahe, die Kommunikationsreihenfolge umzustellen. Leider ist es nicht ohne weiteres möglich, eine Reihenfolge für ein beliebiges Kommunikationsschema zu ermitteln, die die Wartezeiten beim Empfangen der Pakete minimiert. Da das Schema je nach Prozessorenanzahl und Gitteraufbau variiert, ist es auch unmöglich, gute fest implementierte Kommunikationsabfolgen zu verwenden. Eine alternative Möglichkeit bieten die nicht blockierenden Direktiven von MPI. Mit diesen ist es möglich, erst sämtliche Sende- und Empfangsaufrufe zu tätigen und anschließend das Scheduling MPI zu überlassen. Jeder der MPI-Aufrufe liefert ein Statushandle zurück, mit dessen Hilfe alle Prozessoren darauf warten können, dass MPI die Kommunikation fertiggestellt hat. Dies geschieht typischerweise durch einen MPI_Wait()-Aufruf (vgl. Abschnitt 3.2). Damit ergibt sich der in Algorithmus (6) dargestellte Aufbau.

Erstaunlicherweise stellte sich heraus, dass der zweite auf nicht blockierende Kommunikation basierende Algorithmus nur unwesentlich schneller ist, als der Algorithmus (5). Deshalb untersuchten wir ersteren (5) bezüglich der entstandenen Wartezeiten genauer. Zu diesem Zweck wählten wir das folgende vereinfachte Model:

- Sendeaufrufe sind nicht blockierend und benötigen die kleinste (grafisch) darstellbare Zeiteinheit.
- Jeder Prozessor sendet an jeden anderen ein Paket der selben Größe.
- Das versendete Paket besteht (wie in der Realität) aus einem kleinen Informations- und einem sich anschließenden großen Datenpaket.

Algorithmus 6 Paketversende- und Empfangsalgorithmus

```
1: procedure TAUSCHDATENAUSNB MPI_Status status[2*p];
2:   for (i = p; i >= 1; i--) do
3:     if (i != Pi) then
4:       MPI_Recv(Ri→Pi, sendNum, &(status[i]));
5:     end if
6:   end for
7:   for (i = p; i >= 1; i--) do
8:     if (i != Pi) then
9:       MPI_Send(RPi→i, empfNum, &(status[p+i]));
10:    end if
11:  end for
12:  for (i = 2*p; i >= 1; i--) do
13:    if (i != Pi) then
14:      MPI_Wait(status[i]);
15:    end if
16:  end for
17: end procedure
```

- Die Reihenfolge des Paketversendens orientiert sich an Algorithmus (5).

Mit Hilfe dieser Annahmen simulierten wir grafisch den Verlauf der Kommunikation (siehe Abbildung 4.12).

Die eingezeichneten Wartezeiten zeigen deutlich, warum der zweite Algorithmus (6) nicht wesentlich schneller sein konnte. Die Wartezeiten belaufen sich für das verwendete Modell maximal auf die Zeitdauer, die benötigt wird, um ein Datenpaket zu versenden. Führt man die Überlegung mit diesem Modell fort und nimmt an, dass manche Prozessorenpaare keine Daten austauschen, so erkennt man, dass die gesamte Kommunikationsdauer nicht länger ist als die Zeit, die ein Prozessor maximal für das Empfangen aller eigener und eines zusätzlichen Datenpakets benötigt. Selbst wenn man Pakete unterschiedlicher Größe verwendet, ergibt sich ein effizientes Kommunikationsmuster.

Für die in Kapitel 5 getätigten Messungen verwendeten wir, wegen seiner geringfügig günstigeren Laufzeit, den als zweites eingeführten Algorithmus.

4.4 Berechnung des Platzbedarfs

Abbildung 4.9 stellt die Situation vor einem Datenaustauschzyklus dar. Die Knoten verteilen sich dabei innerhalb eines Prozessors auf zwei Keller. Die Knoten auf den Prozessrändern liegen in einem Randkeller, während die übrigen Knoten im Gebietskeller gespeichert sind. Da es sich bei den so im Arbeitsspeicher gehaltenen Daten um mehrere Hundert Megabyte han-

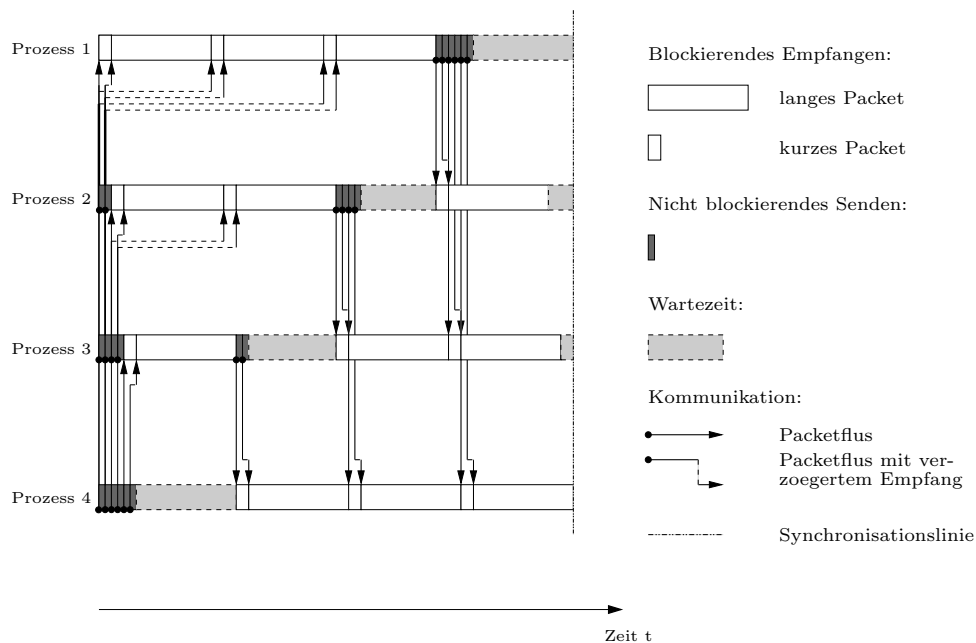


Abbildung 4.12: Vereinfachter Kommunikationsverlauf.

delt, ist es notwendig, die Größe der Datenstrukturen schon zu Beginn gut abschätzen zu können. Markus Pögl [Pög04] konnte für den sequentiellen Algorithmus den Speicherbedarf für den Gebietskeller einfach abschätzen, da bei ihm sowohl die Anzahl der Knoten also auch der pro Knoten benötigte Platz bekannt war. Außerdem gab es natürlich keine Knoten auf dem Randkeller.

Der parallele Algorithmus hingegen kennt zum Zeitpunkt der Speicherreservierung zwar die Gesamtzahl an Knoten, aber nicht die Aufteilung zwischen Rand- und Gebietskeller. Zudem ist der Speicherplatzverbrauch der Randknoten variabel. Er hängt davon ab, ob ein Knoten Grob- oder Feinlevelknoten ist, ob er Randbits speichern muss oder nicht und an wieviele Prozessoren er verschickt werden muss.

Hat man eine gute Schätzung für die Aufteilung der Knoten auf Prozessrand und Prozessgebiet, so kann man leicht die Größe des Gebietskellers angeben, da die dort gespeicherten Knoten alle die selbe Größe haben. Für die auf dem Randkeller gespeicherten Knoten ist nur eine etwas gröbere Abschätzung des Platzbedarfs möglich. Man benötigt zuerst die Information, wie viel Speicherplatz ein einzelner Knoten eines Knotentyps belegt. Kennt man den Platzbedarf, so muss man noch abschätzen, wie oft welcher Knotentyp auf dem Randkeller vorkommt. Beginnen wir mit der Aufschlüsselung des Speicherverbrauchs, die sich in Aufzählung 5.2 auf Seite 75 findet. Addiert man die Elemente der Auflistung auf, so ergibt sich daraus für einen Knoten

folgender Speicherplatzbedarf:

- Feinlevelknoten am Rand: $< 27 + \hat{p}$ Byte
- Groblevelknoten am Rand: $< 43 + \hat{p}$ Byte

\hat{p} steht dabei für die Anzahl der Prozessoren, an die ein Knoten geschickt werden muss. Wir gehen davon aus, dass praktisch alle Feinlevelknoten nur an einen Prozessor geschickt werden. Zusätzlich weiß man, dass es weniger Knoten auf dem jeweiligen Level gibt, je weiter oben man sich im space-tree befindet. Die Zahl der Groblevelknoten ist also vernachlässigbar klein ($\approx 3\%$). Dagegen werden diese Knoten mit jedem Level, welches sie weiter oben im spacetree liegen, von mehr verschiedenen Prozessoren benötigt, besitzen also ein größeres \hat{p} . Selbst unter Berücksichtigung dieser Tatsache ist der Mehrverbrauch der Groblevelknoten für die Abschätzung nicht von Bedeutung. Wir wählen als durchschnittlichen Platzverbrauch eines Randknotens also 28 Byte.

Nun benötigen wir nur noch eine Möglichkeit, die Zahl der Knoten auf dem Rand abzuschätzen. In Abschnitt 2.6.3 wird die Abschätzung (2.27) gezeigt. Zusätzlich gehen wir nach den Aussagen von Abschnitt A davon aus, dass $c_{Würfel} = \frac{9}{8}$ ist und erinnert uns daran, dass bei der Abschätzung des Quader Volumens für (2.27) in 2.6.3 etwa eine Größenordnung (3^{-d}) verloren ging. Im dreidimensionalen Fall heißt das, dass das Volumen etwa um den Faktor 27 und damit auch die Oberfläche um den Faktor 9 zu hoch geschätzt ist. Rechnet man diesen Faktor noch mit ein und schätzt die Zahl der Knoten durch die Zahl der Oberflächenelemente bzw. Zellen ab, so erhält man

$$knoten_{Oberfläche} \approx 10 \cdot knoten_{Berechnungspartition}^{\frac{2}{3}} \quad (4.4)$$

Wählt man in der Implementierung diese Schätzung zur Bestimmung der Kellergröße, so zeigt sich, dass die Wahl sehr gut den Bedarf an Speicherplatz abschätzt. Da mit zunehmender Komplexität des knotenbezogenen Speicherverbrauchs die Schätzung des Gesamtbedarfs schwieriger wird, sollte in Zukunft dazu übergegangen werden, den Platz für die Datenstrukturen dynamisch zu reservieren. Gerade im Hinblick auf lösungsadaptive Verfeinerungen erscheint dies sinnvoll.

4.5 Initialisierung

Die vorgestellten Konzepte der Randspeicherung, des teilweisen Gebietsdurchlaufs und der Kommunikation gehen alle davon aus, dass die Daten in der dazu benötigten Form in den Datenstrukturen vorliegen. Diese Daten müssen natürlich erst in einer Initialisierungsphase erzeugt und mit den zusätzlich nötigen Informationen versorgt werden. Bei Markus Pögl [Pög04] geschieht dies mit Hilfe zweier Initialisierungsiterationen.

Algorithmus 7 Parallelisierter iterativer Löser

```
1: procedure MAIN
2:   Zellzählung();
3:   Initialisierung();
4:   while (maxResiduum < abbruchResiduum) do
5:     P_Steuerung();
6:     TauscheDatenAus();
7:   end while
8:   Ausgabe();
9: end procedure
```

1. Iteration: Ermittlung des Speicherplatzbedarfs aller Datenstrukturen.
2. Iteration: Erzeugung und anschließende Speicherung der Knoten.

Um die Daten des parallelen Algorithmus' entsprechend der Partitionierungsvorschriften zu erzeugen, sind drei Iterationen durch das gesamte Gebiet (aller Prozessoren!) nötig. Diese beinhalten folgende Aufgaben:

1. Iteration: Ermittlung der Gesamtzahl der Zellen in Ω . Danach ist es möglich auszurechnen, welche der Zellen in der Berechnungspartition welches Prozessors liegen.
2. Iteration: Ermittlung des Bedarfs an Speicherplatz für sämtliche Datenstrukturen. (wie Iteration 1 von Markus Pögl)
3. Iteration: Erzeugung und Speicherung der Knoten, die zu einer Berechnungspartition gehören.

Im Anschluss an diese Initialisierung ist es möglich, nur noch die Durchlaufpartitionen der einzelnen Prozessoren zu durchlaufen. Die Notwendigkeit, drei Iterationen durch das gesamte Berechnungsgebiet aller Prozessoren vornehmen zu müssen, stellt für die Berechnung sehr großer Probleme einen deutlichen Nachteil dar, da die Initialisierung in einem solchen Fall den Hauptteil der Rechenzeit ausmacht. Algorithmus (7) stellt den gesamten angepassten Ablauf des parallelen Programms dar. Die Erweiterungen bezüglich Algorithmus (1) betreffen vor allem die zusätzliche Initialisierung und den Datenaustausch.

Kapitel 5

Ergebnisse

Das folgende Kapitel versucht anhand von Messergebnissen die wichtigsten Charakteristika des parallelen Algorithmus' zu beschreiben. Dazu analysieren wir in 5.1 die Laufzeit, in 5.2 den Speicherverbrauch und in 5.3 die Cache-Effizienz. Da die numerischen Ergebnisse identisch zu denen des sequentiellen Algorithmus' sind, wird dafür auf Markus Pögl [Pög04] verwiesen.

5.1 Laufzeit

Wie in Abschnitt 4.1 über die Effizienz paralleler Programme beschrieben, gibt es im Wesentlichen zwei Möglichkeiten, die Laufzeit eines parallelen Algorithmus' zu beurteilen. Bei ersterer vergleicht man den parallelen mit dem schnellsten bekannten sequentiellen Löser. Dadurch gewinnt man eine Aussage darüber wie viel Performanz durch den mit der Parallelisierung einhergehenden Verwaltungsaufwand verloren gegangen ist. Ein solcher Verlust läßt sich im Prinzip kaum vermeiden und wird durch die sequentielle Effizienz beschrieben:

$$e_{sequ}(1) = \frac{s_{sequ}(1)}{1} = \frac{T_{sequ}}{T(1)}$$

Hat man sichergestellt, dass sich der durch die Parallelisierung eingeführte Geschwindigkeitsverlust in einer vertretbaren Größenordnung bewegt, so ist der parallele Speedup von wesentlich größerer Bedeutung. Dieser gibt Aufschluss darüber, wie sich eine Ver- n -fachung der Prozessorenzahl auf die Laufzeit des parallelen Programms auswirkt. Falls man mit jeder Ver- n -fachung erreicht, dass sich die Laufzeit auf ein n -tel der Zeit verkürzt, so spricht man von linearem Speedup. Da es sich dabei um ein theoretisches Optimum handelt, ist in der Praxis wünschenswert, eine möglichst gute Annäherung daran zu erreichen. Erst dann ermöglicht der Einsatz sehr großer Cluster eine fast beliebige Beschleunigung der Rechenzeit, sofern der sequentielle Anteil klein genug ist (Amdahl's Law) [amd].

In den folgenden Absätzen werden im Wesentlichen drei verschiedene Beispiele untersucht:

- **Würfel:** Reguläres Gitter, bei dem sämtliche Zellen innerhalb des Berechnungsgebiets liegen. D.h. es existieren keine Außenzellen.
- **Kugel:** Nicht reguläres Gitter, bei dem an der Kugeloberfläche die Auflösung am höchsten ist. Alle Zellen außerhalb des Kugelgebiets sind Außenzellen. Das Verhältnis von Außenzellen zu Innenzellen ist niedrig.
- **Stern:** Nicht reguläres Gitter, bei dem die Auflösung am Rand und innerhalb des Sterns am höchsten ist. Alle Zellen außerhalb des Sterngebiets sind Außenzellen. Das Verhältnis von Außenzellen zu Innenzellen ist hoch.

Die Unterschiede liegen demnach vor allem im Aufbau des Gitters und in dem Verhältnis von Innen- zu Außenzellen. Wir geben im Zusammenhang mit den Beispielen immer Informationen über die Auflösung des Gitters an. Dabei unterscheiden wir die Auflösung innerhalb, am Rand und außerhalb des Rechengebietes. Für den Rand geben wir immer eine exakte Auflösung an, während für den äußeren und inneren Bereich immer nur minimale Auflösungen angegeben werden. Minimale Auflösung bedeutet dabei, dass jede Zelle, die nicht verfeinert sein muss, um den Rand in der angegebenen Randverfeinerung aufzulösen, auch nicht verfeinert ist. Die Angaben werden dabei von innen nach Aussen genannt. "81:243:27" bedeutet, dass im Inneren mindestens 81er, am Rand genau 243er und außerhalb mindestens 27er Auflösung vorherrscht.

5.1.1 Sequentielle Effizienz

Zuerst wollen wir anhand einiger Messergebnisse die sequentielle Effizienz ermitteln. Dazu vergleichen wir die in Tabelle 5.1 genannten Einprozessorlaufzeiten des parallelen Programms mit den Laufzeiten des sequentiellen Löser von Markus Pögl [Pög04]. Die Einträge stellen die durchschnittliche Laufzeit einer Berechnungssiteration dar. Damit wird die Initialisierungs- und Ausgabezeit nicht berücksichtigt, was es uns ermöglicht, eine Aussage darüber zu machen, wie sich die für die eigentliche Berechnung nötige Zeit verändert hat.

Die Ergebnisse zeigen, dass durch die Parallelisierung 18% - 37% der Performanz verloren gehen. Die großen relativen Unterschiede zwischen den verschiedenen Beispielen haben ihren Ursprung im unterschiedlichen Verhältnis zwischen Aussen- und Innenzellen. Liegen wie im Sternbeispiel sehr viele Außenzellen vor, so verlieren wir durch die Parallelisierung relativ viel Geschwindigkeit. Die Erklärung liegt in der unterschiedlichen Verarbeitung

Würfel (243:243:243)	
Zellen	$14,9 \cdot 10^6$
Freiheitsgrade	$14,7 \cdot 10^6$
Laufzeit sequentiell in s	236,9
Laufzeit parallel (p = 1) in s	279,3
Mehrlaufzeit	17,9%
$e_{sequ}(1)$	84,8%
Kugel (81:243:1)	
Zellen	$1,2 \cdot 10^6$
Freiheitsgrade	$0,9 \cdot 10^6$
Laufzeit sequentiell in s	17,2
Laufzeit parallel (p = 1) in s	20,6
Mehrlaufzeit	19,3%
$e_{sequ}(1)$	83,5%
Stern (243:243:1)	
Zellen	$6,7 \cdot 10^6$
Freiheitsgrade	$0,44 \cdot 10^6$
Laufzeit sequentiell in s	54,4
Laufzeit parallel (p = 1) in s	74,6
Mehrlaufzeit	37,2%
$e_{sequ}(1)$	72,9%

Tabelle 5.1: Einprozessorlaufzeiten im Vergleich zum sequentiellen Programm - System: Pentium III 800Mhz - 1GByte Ram

der beiden Zelltypen. Der sequentielle Code muss in den Aussenzellen wesentlich weniger Berechnungen durchführen als in den Innenzellen. Da die Parallelisierung jedoch für Zellen jedes Zelltyps einen gleichmäßigen Mehraufwand mit sich bringt, fällt der relative Mehraufwand für Beispiele mit vielen Aussenzellen größer aus. Weil man aber in der Praxis versucht, die Zahl der Außenzellen möglichst gering zu halten, sind die ansonsten auftretenden Geschwindigkeitsverluste von etwa 20% durchaus akzeptabel.

Um zu untersuchen, wodurch der Mehraufwand verursacht wird, haben wir mit Hilfe des GNU-Profilers das sequentielle mit dem parallelen Programm verglichen. Dabei stellte sich heraus, dass eine große Zahl von zusätzlich implementierten Methoden für den erhöhten Aufwand verantwortlich sind. Da sich der Rechenaufwand sehr gleichmäßig auf diese verteilt und für jede einzelne verschwindend gering ist, kann man davon ausgehen, dass die Implementierung sehr effizient gelungen ist. Auffallend ist außerdem, dass ein großer Teil des eingeführten Mehraufwandes auf Konstruktions- und Destruktionsmethoden zurückzuführen ist. Einen großen Teil davon machen jene Datenstrukturen aus, die beim Betreten einer Zelle mit Daten aus den Kellern gefüllt und beim Verlassen der Zellen wieder gelöscht werden. Um

diesen Aufwand zu verringern, bietet es sich als zukünftige Optimierung an, diese Datenstrukturen global zu speichern und nicht in jeder Zelle neu anzulegen bzw. zu löschen. Der übrige Aufwand verteilt sich schließlich auf Methoden zur Verwaltung der Randbits und zur Berechnung und Transformation der Positionsangaben.

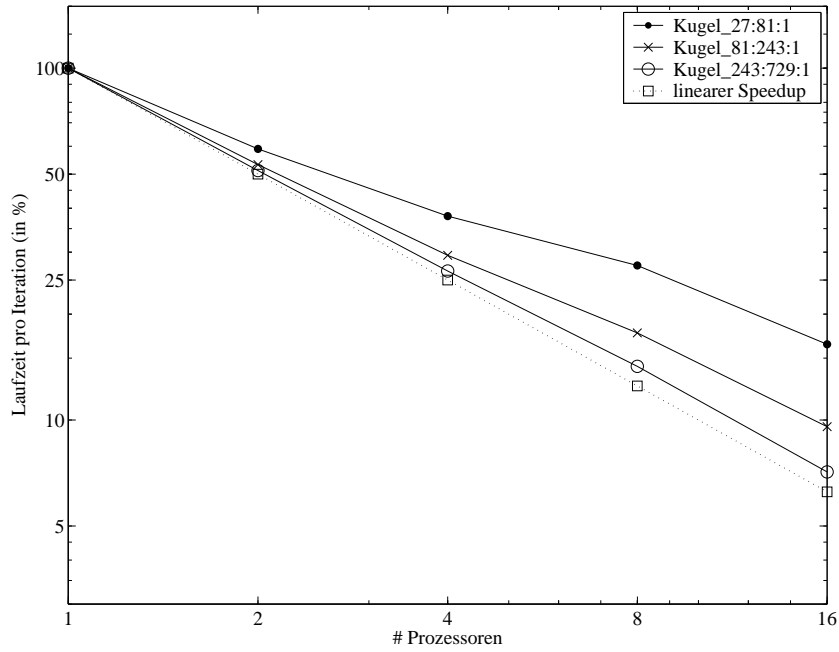
5.1.2 Parallele Effizienz

Nachdem wir gezeigt haben, dass der durch die Parallelisierung eingeführte Mehraufwand in einem vertretbaren Rahmen liegt, wollen wir die parallele Effizienz untersuchen. Dazu führen wir mit den gleichen Beispielen wie zuvor Laufzeitmessungen durch und berechnen den erreichten parallelen Speedup. Da der maximale Speedup durch Amdahl's Law nach oben beschränkt ist, vergleichen wir den Speedup der selben Beispiele mit unterschiedlichen Auflösungen. Dadurch erhalten wir eine Aussage darüber, ob fein aufgelöste Beispiele durch den Einsatz einer größeren Zahl von Prozessoren in vergleichbarer Zeit gelöst werden können wie grob aufgelöste Beispiele. Im dreidimensionalen Fall bedeutet ein Auflösungsschritt etwa eine Verachtundzwanzigfachung der Gesamtzellenzahl, da dabei in jeder Blattzelle 27 neue Zellen entstehen. Wir müssten also die Laufzeiten eines kleinen Beispiels auf 1, 2 bzw. 4 Prozessoren mit der Laufzeit des nächst größeren Beispiels auf 28, 56 bzw. 112 Prozessoren vergleichen. Da zur Zeit der Performanzmessungen nur ein Cluster mit 16 Prozessoren zur Verfügung stand, konnten derartige Messungen nicht durchgeführt werden.

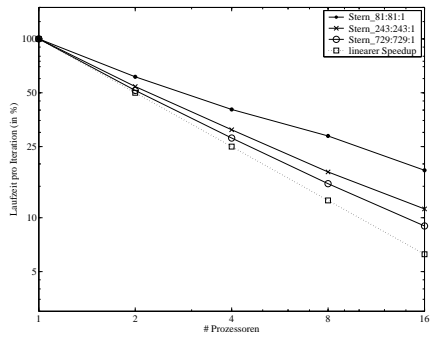
Mit Ausnahme der Cachemessungen wurden sämtliche Beispiele auf dem lehrstuhleigenen Cluster gerechnet. Bei diesem handelt es sich um ein System mit 8 Dual Pentium III Rechnern, denen pro Knoten 2 GByte RAM zur Verfügung stehen. Die Knoten sind untereinander mittels Myrinet verbunden.

Aus diesem Grund haben wir jedes der Beispiele mit unterschiedlich fein aufgelösten Gittern auf 1, 2, 4, 8 und 16 Prozessoren berechnet. Die Ergebnisse sind in den folgenden Diagrammen für das Würfel- (5.1(c)), das Kugel- (5.1(a)) und das Sternbeispiel (5.1(b)) grafisch dargestellt. Zusätzlich listet Tabelle 5.2 die Ergebnisse der jeweils am höchsten aufgelösten Beispiele auf.

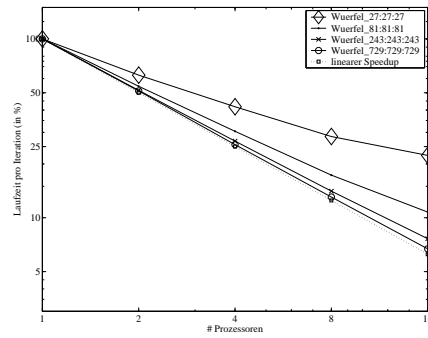
Da das Kugelbeispiel vom Verhältnis Innen- zu Außenzellen zwischen den beiden anderen liegt, wählen wir dieses als Musterbeispiel. Betrachtet man die zugehörige Abbildung, so sieht man, dass je höher das Gitter aufgelöst ist, desto höher ist der Geschwindigkeitsgewinn durch die Parallelisierung. Die Berechnungen auf den am höchsten aufgelösten Gittern des Kugel- und Würfelbeispiels, ergeben einen Speedup von 14-15 auf 16 Prozessoren und liegen damit sehr nahe am linearen Speedup. Dieses Ergebnis lässt die Schlussfolgerung zu, dass der Parallelisierung von großen Problemen mit mehr als 10^7 Freiheitsgraden praktisch keine Grenzen gesetzt sind. Es ist zu erwarten,



(a) Kugelbeispiel.



(b) Sternbeispiel.



(c) Würfelbeispiel.

Abbildung 5.1: Laufzeitvergleich verschiedener Beispiele mit unterschiedlichen Auflösungen.

Beispiel	Würfel	Kugel	Stern
FG	$4,01 \cdot 10^8$	$2,59 \cdot 10^7$	$1,67 \cdot 10^8$
T(1)	1	1	1
T(2)	1,95	1,95	1,94
T(4)	3,9	3,77	3,58
T(8)	7,66	7,04	6,45
T(16)	14,92	13,65	11,11

Tabelle 5.2: Paralleler Speedup $T(p)$ und Freiheitsgradzahl (FG) der größten Beispiele in der jeweiligen Geometrie.

	Würfel	Kugel	Stern
1	0,00%	0,00%	0,00%
2	1,27%	2,61%	2,13%
4	2,45%	5,77%	4,05%
8	4,03%	10,09%	6,32%
16	5,53%	13,47%	8,44%

Tabelle 5.3: Prozentsatz der Zellen, die in der Durchlaufs- aber nicht in der Berechnungspartition liegen. Als Gitter wurde das jeweils am höchsten aufgelöste verwendet.

dass die Berechnung des Würfelbeispiels auf 128 Prozessoren gut 110 mal so schnell erfolgt, wie auf einem Prozessor.

Dass die Parallelisierung für das Sternbeispiel erneut nicht ganz so gute Ergebnisse liefert, liegt vor allem daran, dass die Zahl der Randzellen sehr hoch ist und damit die Kommunikation sehr viel Zeit beansprucht. Ließe man die Kommunikationszeit außer acht, so ergäben sich die selben sehr guten Ergebnisse wie im Würfelbeispiel. Als weiteren Grund für die relativ niedrigen Speedups des Sternbeispiels lässt sich der Anteil der Zellen nennen, die beim Sternbeispiel in der Durchlauf- aber nicht in der Berechnungspartition liegen. Dieser ist, wie in Tabelle 5.3 dargestellt, relativ groß. Allerdings ist der Anteil im Kugelbeispiel noch höher, was darauf hindeutet, dass eine Mehrzellenanzahl in dieser Größenordnung noch keinen wesentlichen Einfluss auf die Laufzeit hat.

Zusammenfassend halten wir fest, dass sich all jene Geometrien sehr gut parallelisieren lassen, deren Außenzellenanteil klein ist. Da dies in der Praxis häufig der Fall sein wird, kann man die sehr guten Zahlen des Kugel- und Würfelbeispiels getrost als Referenzwerte und die Ergebnisse des Sternbeispiels als absolut unterste Grenze betrachten.

5.2 Speicherverbrauch

Bei der numerischen Lösung von Differentialgleichungen im Dreidimensionalen zeigt sich, dass wegen der hohen Zahl von Unbekannten sehr schnell die Größe des Arbeitsspeichers ein limitierender Faktor werden kann. Die Parallelisierung trägt neben der Verringerung der Rechenzeit auch dazu bei, diese Einschränkung aufzuheben. Bei der Verwendung eines datenparallelen Ansatzes verteilt man die Datenmenge gleichmäßig auf alle Prozessoren, wodurch der gesamte verfügbare Speicherplatz mit der Anzahl der zur Verfügung stehenden Prozessoren steigt. Im Falle des für die Messungen verwendeten Clusters mit einem Gigabyte Arbeitsspeicher je Prozessor haben wir insgesamt 16 Gigabyte zur Verfügung. Damit diese enorme Menge es ermöglicht, auch entsprechend größere Probleme zu berechnen, muss der zusätzliche Speicheraufwand, den die Parallelisierung mit sich bringt, gering gehalten werden. Um zu zeigen, dass die vorliegende Implementierung speichereffizient arbeitet, wollen wir zuerst kurz zusammenfassen, wofür das parallele Programm zusätzlichen Speicherplatzverbrauch benötigt.

Wie in Abschnitt 4.4 beschrieben, hängt der benötigte Speicherplatz vom Typ eines Knotens ab. Im Vergleich zum sequentiellen Fall belegen ausschließlich jene Knoten zusätzlichen Speicherplatz, die von mehreren Prozessoren benötigt werden. Der genutzte Speicherplatz der Randknoten setzt sich dabei wie folgt zusammen.

Im sequentiellen Code wurden benötigt:

- 1 Byte: Flags (wie sequentiell)
- 4 Byte: Lösung (wie sequentiell - nur Innenknoten)

Durch die Parallelisierung zusätzlich eingeführter Speicherbedarf:

- 8 Byte: Residuumskorrekturanteil
- 13 Byte: Knotenposition
- 1 Byte : Anzahl der interessierten Prozessoren = p_{id}
- p_{id} Byte: p_{id} ProzessIDs der interessierten Prozessoren
- 16 Byte: Speicherung der Randbits (nur Groblevelknoten)

Damit können wir den durch die Parallelisierung eingeführten Speichermehraufwand mit $22 + p_{id}$ Byte für Feinlevelknoten bzw. $38 + p_{id}$ Byte für Groblevelknoten angeben. Wie in Abschnitt 4.4 kann man erneut die Groblevelknoten wegen ihrer geringen Anzahl unberücksichtigt lassen. Da fast alle Knoten nur von einem weiteren benachbarten Prozessor benötigt werden, setzen wir $p_{id} = 1$ und erhalten einen Mehraufwand pro Randknoten von 23 Byte gegenüber dem sequentiellen Fall. Schätzen wir die Zahl der Randknoten wie in Gleichung (4.4) ab, so erhalten wir in Abhängigkeit von der

Knotenanzahl n und der Anzahl der Prozessoren p folgende Abschätzung für den zusätzlich benötigten Speicher $extraMem$ pro Prozessor:

$$extraMem(n, p) = 230 \cdot \left(\frac{n}{p}\right)^{\frac{2}{3}}$$

Dabei stellt $\frac{n}{p}$ die Knotenanzahl dar, die ein Prozessor in seiner Berechnungspartition zu bearbeiten hat. Um den gesamten relativen Speichermehraufwand auf allen Prozessoren zu erhalten, teilen wir die Gleichung noch durch den Speicherplatzbedarf, den sämtliche Knoten im sequentiellen Fall hatten und multiplizieren $extraMem$ mit p . Dabei nehmen wir an, dass alle Knoten innerhalb des Gebiets liegen, also 5 Byte benötigen:

$$relExtraMem = \frac{extraMem \cdot p}{5 \cdot n}.$$

Damit erhalten wir die in der Tabelle 5.4 aufgelisteten geschätzten relativen Mehraufwände:

n	p	relExtraMem
10^5	4	157.3%
10^5	8	198.2%
10^5	16	249.7%
10^5	64	396.4%
10^7	4	33.9%
10^7	8	42.7%
10^7	16	53.8%
10^7	64	85.4%
10^9	4	7.3%
10^9	8	9.2%
10^9	16	11.6%
10^9	64	18.4%

Tabelle 5.4: Geschätzter relativer Speichermehraufwand

Dabei zeigt sich, dass mit steigender Freiheitsgradanzahl der relative Mehraufwand deutlich sinkt. Da erst Berechnungen mit $> 10^8$ Freiheitsgraden eine Herausforderung in Sachen Speicherverbrauch darstellen und deren Speichermehrverbrauch nur sehr klein ausfällt, kann die Implementierung als speichereffizient betrachtet werden. Dabei muss zusätzlich bedacht werden, dass zum Lösen der Poissongleichung im sequentiellen Fall nur 5 Byte pro Innenknoten gespeichert werden, was einen Mehraufwand von 20% also gerade 1 Byte pro Knoten ausmacht. Zukünftige Strömungsgleichungslöser werden für das Flag (1 Byte), den Druck (4 Byte) und die Geschwindigkeiten (je 4 Byte) alleine schon mindestens 17 Byte pro Knoten speichern, wogegen sich der Parallelisierungsmehraufwand als verschwindend gering ausnimmt. Der

in Tabelle 5.4 genannte große Mehraufwand für die kleinen Beispiele resultiert aus dem in diesen Fällen sehr hohen Anteil an Randknoten und der nicht optimalen Schätzung für die Knotenanzahl auf der Oberfläche. Diese sind also nur der Vollständigkeit wegen genannt und stellen wegen ihres geringen Rechenaufwandes keine realistischen Beispiele dar. Im Vergleich zu den abgeschätzten Werten gibt die Tabelle 5.5 den tatsächlich benötigten relativen Speichermehraufwand des parallelen Programms an. Die beiden

n	p	mem (parallel)	mem (sequ)	relExtraMem
$5,3 \cdot 10^5$	8	6,8MB	3MB	128%
$1,47 \cdot 10^7$	8	110,9MB	76MB	46%

Tabelle 5.5: Tatsächlicher Mehraufwand für Würfelbeispiel (keine Außenknoten)

Beispiele zeigen, dass sich die obige Abschätzung in der Praxis sehr gut bewährt. Wählt man statt dem Würfelbeispiel andere Beispiele, bei denen ein Teil der Zellen außerhalb des Rechengebietes liegt, so steigt der relative Mehrverbrauch etwas. Dies liegt daran, dass die außerhalb liegenden Knoten im sequentiellen Programm nur 1 Byte, die Innenknoten dagegen 5 Byte benötigen. Dadurch ist der relative Mehrverbrauch, für die Randaussenknoten, deutlich höher als der der Randinnenknoten.

5.3 Cache-Effizienz

Eine der Haupteigenschaften des sequentiellen Löser ist dessen Cacheeffizienz. Diese galt es in der vorliegenden Arbeit zu bewahren. Da wir die grundlegende Datenstruktur und deren Zugriffsmechanismen nur in geringem Maße verändert haben, erwarten wir für den parallelen Algorithmus ähnliche Cache Hit Rates wie für das sequentielle Programm. Um diese Vermutung zu überprüfen wurde ein Teil der Beispiele, die auch von Markus Pögl in [Pög04] berechnet wurden, mit dem Tool perfex [per] überprüft. Die Ergebnisse werden in Tabelle 5.6 dargestellt.

Die Ergebnisse bestätigen die Vermutung, dass die optimalen Cache Hit Rates weiterhin bestehen bleiben. Die vorletzte Spalte stellt dabei den Mittelwert der Cache Hit Rates der vier zur Messung verwendeten Prozessoren dar. Die Abweichungen der einzelnen Prozessoren von diesem Wert sind unbedeutend. Damit kann auch der vorgestellte parallele Algorithmus als Cache optimal bezeichnet werden.

Gebiet	Auflösung	Freiheitsgrade	L2 Hit Rate	L2 Hit Rate [Pög04]
Würfel	27	18.096	99,94%	99,99%
	81	530.096	99,95%	99,97%
	243	14.702.584	99,96%	99,95%
Kugel	81:81:81	60.032	99,95%	99,99%
	243:243:243	1.851.784	99,98%	99,97%
	81:243:1	855.816	99,95%	99,94%

Tabelle 5.6: Ergebnisse der Messungen der Cache Hit Rates auf einem Dual Xeon 2,4 GHz mit 4 GByte Hauptspeicher im Vergleich zu Markus Pögls Messwerten

Kapitel 6

Zusammenfassung und Ausblick

6.1 Ergebnisbewertung

Die vorliegende Arbeit stellt eine effiziente Parallelisierung des von Markus Pögl implementierten Cache optimalen Finite Element Lösers vor. Nachdem wir als ersten Schritt gezeigt haben, dass das verteilte Rechnen auf Teilabschnitten der Peanokurve möglich ist, verbesserten wir die zugrunde liegenden Algorithmen hinsichtlich Speicherverbrauch und Laufzeit. Die in Kapitel 5 vorgestellten Ergebnisse bestätigen, dass die parallele Implementierung ohne Einschränkungen einsetzbar ist, um große Probleme mit Hilfe von entsprechend großen Clustern schnell zu berechnen. Wir haben gezeigt, dass die parallele Umsetzung praktisch sehr gut skaliert und für gut gewählte Gitter einen nahezu linearen Speedup erreicht. Darüber hinaus zeigt sich, dass der Speichermehrverbrauch für große Probleme niedrig ist.

6.2 Ausblick

Die in Abschnitt 4.5 erwähnten Nachteile bei der Initialisierung der Datenstrukturen stellen einen großen Nachteil dar. Allein die Initialisierungsphase benötigt etwa die $3p$ -fache Zeit einer Iteration auf p Prozessoren. Da damit Berechnungen auf großen Clustern nicht mehr effizient durchgeführt werden können, ist der nächste Schritt, die Initialisierung der Datenstrukturen zu optimieren. Die dazu nötigen Algorithmen werden zur Zeit von Wolfgang Herder implementiert [Her04]. Dabei werden in Form einer ersten Initialisierungsphase die Daten wie bisher auf p Prozessoren verteilt, wobei das zugrunde liegende Gitter allerdings wesentlich gröber ist. Erst durch weitere parallele Verfeinerungen der Teilgebiete wird dann die eigentlich angestrebte Gitterstruktur erreicht. Zu diesem Zwecke werden die von Nadine Dieminger [Die05] und Andreas Krahnke [Kra04] implementierten Algorith-

men zur Gitteradaption in den parallelen Code übernommen. Außerdem besteht die Möglichkeit die Implementierung, basierend auf den von Judith Hartmann [Har04] gewonnenen Ergebnissen, zu optimieren. Abschließend soll die schon im Zweidimensionalen umgesetzte Lösung der Navier-Stokes Gleichung ([Wei05], [Nec05]) in den parallelen dreidimensionalen Löser integriert werden.

Literaturverzeichnis

- [amd] Das Gesetz von Amdahl. URL <http://iamlasun8.mathematik.uni-karlsruhe.de/parallel/skript/node13.html>.
- [Bad04] Michael Bader. Raumfüllende Kurven - Begleitendes Skriptum zum entsprechenden Kapitel der Vorlesung Algorithmen des Wissenschaftlichen Rechnens. URL <http://www5.in.tum.de/lehre/vorlesungen/algowiss/ss04/vorlesung/rfk.pdf>, 2004.
- [BHM00] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial Second Edition*. 2000.
- [Bun92] Hans Joachim Bungartz. *Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung*. PhD thesis, Technische Universität München, 1992.
- [Die03] Nadine Dieminger. Ferienakademievortrag - Multigrid Methods for Solving Partial Differential Equations, 2003.
- [Die05] Nadine Dieminger. Kriterien für die Selbstadaption cache-effizienter Mehrgitteralgorithmen. Master's thesis, Technische Universität München, 2005.
- [Gri94] Michael Griebel. *Multilevelmethoden als Iterationsverfahren über Erzeugendensystemen*. Teubner Skripten zur Numerik. B.G. Teubner Stuttgart, 1994.
- [Gün04] Frank Günther. *Eine cache-optimale Implementierung der Finite-Elemente-Methode*. PhD thesis, Technische Universität München, 2004.
- [Har04] Judith Hartmann. Entwicklung eines cache-optimalen Finite-Element-Verfahrens zur Lösung d-dimensionaler Probleme. Master's thesis, Technische Universität München, 2004.
- [Her04] Wolfgang Herder. Lastausgleich und parallele Eingabedaten-generierung für ein cache-optimales paralleles Finite-Element-Verfahren. Master's thesis, Technische Universität München, 2004.

- [Jün01] Prof. Dr. Ansgar Jüngel. Das kleine FEM-Skript, 2001.
- [Kov03] Denis Kovacs. Ferienakademievortrag - Hierarchical Bases. URL <http://home.in.tum.de/~kovacsd/math.html>, 2003.
- [Kra04] Andreas Krahnke. *Adaptive Verfahren höherer Ordnung auf cache-optimalen Datenstrukturen für dreidimensionale Probleme*. PhD thesis, Technische Universität München, 2004.
- [Lan04] Markus Langlotz. Cache efficiency and parallelization of numerical algorithms - JASS 2004. URL <http://www.markus.langlotz.info>, 2004.
- [mpi] MPICH. URL <http://www-unix.mcs.anl.gov/mpi/mpich/>.
- [Nec05] Tobias Neckel. Anwendungen einer cache-optimalen Finite-Element-Methode auf 2D-Fluid-Struktur-Wechselwirkungen. Master's thesis, Technische Universität München, 2005.
- [ope] OpenMP. URL <http://openmp.org>.
- [per] perfex: Analysewerkzeug für IA-32 auf Basis des SGI IRIX perfex. URL <http://www.osc.edu/~troy/lperfex/>.
- [Pög04] Markus Pögl. *Entwicklung eines cache-optimalen 3D Finite-Element-Verfahrens für große Probleme*. PhD thesis, Technische Universität München, 2004.
- [Sag94] Hans Sagan. *Space-Filling Curves*. Springer-Verlag New York, 1994.
- [SJ97] Gilbert Strang and George J.Fix. *Multigrid*. Wellesley-Cambridge Press, 1997.
- [TC92] T.N.Bui and C.Jones. Finding good approximate vertex and edge partitions is NP-hard. In *Inf. Process. Lett.*, pages 153–159, 1992.
- [TOS01] Ulrich Torttenberg, Cornelis Oosterlee, , and Anton Schüller. *Multigrid*. Academic Press, London, 2001.
- [Wei05] Tobias Weinzierl. Master's thesis, Technische Universität München, 2005.
- [Zum01] Gerhard Zumbusch. *Adaptive Parallel Multilevel Methods for Partial Differential Equations*. PhD thesis, Universität Bonn, 2001.

Anhang A

Abschätzung der Oberfläche eines diskreten Peanokurvenabschnittes

Allgemein kann davon ausgegangen werden, dass jeder Peanokurvenabschnitt innerhalb eines ihn umgebenden Würfels liegt. Betrachtet man nur die erste auf den umgebenden Würfel folgende Verfeinerungsstufe (siehe eingefärbter Bereich in Abbildung A.1), so erkennt man, dass jeder beliebige zusammenhängende Peanokurvenabschnitt sicher eine kleinere Oberfläche hat, als der ihn umgebende Würfel.

Wir halten fest, dass die Oberfläche der Peanokurve s_j auf der j -ten Verfeinerungsstufe höchstens so groß ist, wie die Oberfläche c_j des sie umgebenden Würfels.

$$s_j \leq c_j \tag{A.1}$$

Nimmt man nun an, dass die Peanokurve bis zu einem Level m verfeinert ist, so kommen weitere Oberflächenanteile zu s_j hinzu. Wie in Abbildung A.2 dargestellt, addieren sich also weitere Oberflächenanteile, die wieder durch die Würfeloberfläche der entsprechenden umschreibenden Würfel nach oben abgeschätzt werden können.

Damit erhält man mit $c_j = 2d \cdot \left(\frac{1}{3}\right)^{j(d-1)}$

$$\begin{aligned} s_{j\dots m} &\leq s_j + \sum_{i=j+1}^m s_i = \sum_{i=j}^m s_i \leq \sum_{i=j}^m c_i \\ &= \sum_{i=j}^m 2d \cdot \left(\frac{1}{3}\right)^{i(d-1)} = 2d \cdot \left(\sum_{i=0}^m \left(\left(\frac{1}{3}\right)^{d-1}\right)^i - \sum_{i=0}^{j-1} \left(\left(\frac{1}{3}\right)^{d-1}\right)^i \right) \\ &\leq 2d \cdot \left(\frac{1}{1 - \left(\frac{1}{3}\right)^{d-1}} - \frac{\left(\frac{1}{3}\right)^{(d-1)j} - 1}{\left(\frac{1}{3}\right)^{d-1} - 1} \right) = 2d \cdot \left(\frac{\left(\frac{1}{3}\right)^{(d-1)j}}{1 - \left(\frac{1}{3}\right)^{d-1}} \right). \end{aligned}$$

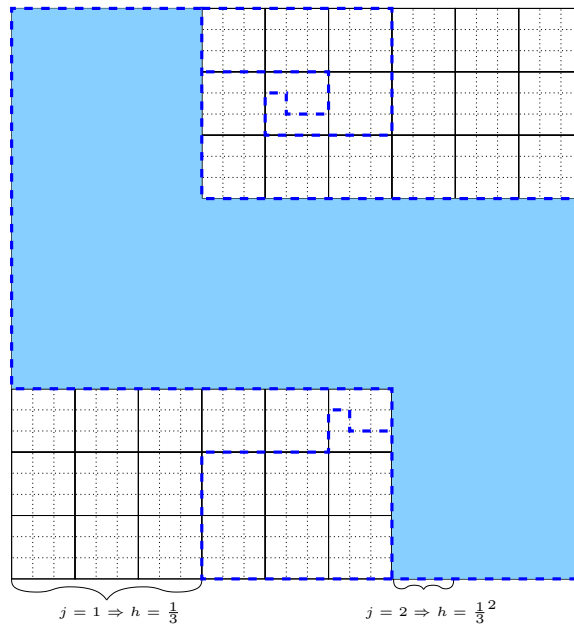


Abbildung A.1: Fläche/Oberfläche einer diskreten Peanokurve im 2D (bei Level 1 eingeraut).

Setzt man diese Oberfläche ins Verhältnis zu c_j erhält man

$$\frac{s_{j \dots m}}{c_j} \leq \frac{2d \cdot \left(\frac{\left(\frac{1}{3}\right)^{(d-1)j}}{1 - \left(\frac{1}{3}\right)^{d-1}} \right)}{2d \cdot \left(\frac{1}{3}\right)^{j(d-1)}} = \frac{1}{1 - \left(\frac{1}{3}\right)^{d-1}} \quad (\text{A.2})$$

$$= 1 + \frac{1}{3^{d-1} - 1}. \quad (\text{A.3})$$

Durch das Einsetzen der Dimension d in die Gleichung (A.3) wird deutlich, dass die Oberfläche der Peanokurve selbst im schlechtesten Fall nur unwesentlich größer ist als die des sie umschreibenden Würfels. Im dreidimensionalen Fall erhalten wir

$$\frac{s_{j \dots m}}{c_j} \leq \frac{9}{8}.$$

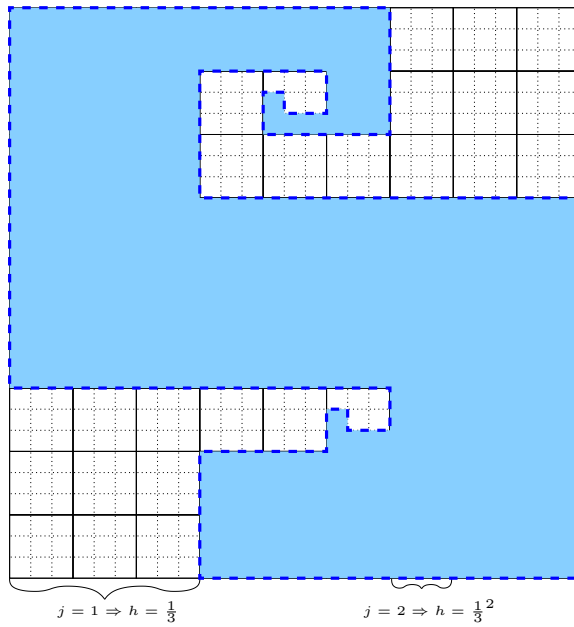


Abbildung A.2: Fläche/Oberfläche einer diskreten Peanokurve im 2D (bis Level 3 (max.) eingeraut).