Eine cache-optimale Implementierung der Finite-Elemente-Methode

Frank Günther

Institut für Informatik
der Technischen Universität München

Lehrstuhl für numerische Programmierung
und Ingenieuranwendungen in der Informatik

Eine cache-optimale Implementierung der Finite-Elemente-Methode

Frank Günther

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Hans Michael Gerndt
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Christoph Zenger
	2. Univ.-Prof. Dr. Arndt Bode

Die Dissertation wurde am 09.03.2004 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 14.05.2004 angenommen.

Inhaltsverzeichnis

1 Einführung und Überblick

Moderne numerische Algorithmen für die Lösung Partieller Differenzialgleichungen müssen mit den effizientesten numerischen Methoden wie etwa Mehrgitterverfahren und adaptiver Gitterverfeinerung und damit mit hierarchischen Datenstrukturen arbeiten können. Gleichzeitig sollte ein numerisches Simulationsprogramm die Möglichkeiten moderner Rechnerarchitekturen möglichst effizient nutzen, um anwendungsrelevante Szenarien mit vertretbarem Aufwand behandeln zu können. In diesem Zusammenhang ist die effiziente Speicherung und Organisation der Daten von großer Bedeutung. Sie manifestiert sich in einem sehr schnellen Datenzugriff durch eine optimale Ausnutzung der Speicherhierarchien der Prozessoren. Eine weitere unerlässliche Anforderung bei der Simulation relevanter Probleme ist die Möglichkeit zur Parallelisierung.

Unglücklicherweise lassen sich diese beiden Forderungen – numerische Effizienz und Effizienz in der Hardwarenutzung – nicht auf ganz triviale Weise vereinbaren. So erzeugen – zumindest in den meisten existierenden Implementierungen – hierarchische Datenstrukturen, die üblicherweise in Bäumen gespeichert werden, einen nicht zu vernachlässigenden Overhead beim Zugriff auf die gespeicherten Daten. Die im Rahmen dieser Arbeit entwickelte Methodik benutzt, um dieser Zwickmühle zu entkommen, das Konzept der Raumfüllenden Kurven, um auch für adaptive Gitter und hierarchische Ansätze Datenstrukturen aufzubauen, die streng linear bearbeitet werden. Tatsächlich werden sogar nur Keller als einzige Datenstruktur benutzt. Daher ist der Zugriff auf die benötigten Daten in der Cache-Hierarchie moderner Prozessoren mit hoher Performance möglich. Er ist sogar schneller als der sonst übliche Zugriff auf nicht-hierarchische Daten, die in Matrizen gespeichert sind. Die Verwendung von Kellern führt zu einer substanziellen Minimierung der Anzahl von cache misses während des Programmlaufs, die messbare hit rate beispielsweise auf dem L2-Cache moderner Prozessoren liegt konstant jenseits 98,0%. Die im Rahmen dieser Arbeit entstandenen Programme lösen Modellgleichungen wie die Poisson-Gleichung oder die Stokes-Gleichung mit einem Mehrgitteransatz auf Basis der Finite-Elemente-Verfahren in im Prinzip beliebigen in das Einheitsquadrat eingebetteten Geometrien. Die zugehörigen Rechengitter können dabei a-priori adaptiv verfeinert sein.

Die Details der entwickelten Methodik, ihre Implementierung sowie einige numerische Ergebnisse werden in den folgenden Kapiteln dargestellt.

In **Kapitel 2** beschreiben wir zunächst die Grundlagen von Geometriebeschreibungen und Baumkonzepten zu ihrer Organisation und Speicherung. Außerdem geben wir einen Überblick über die Theorie der Raumfüllenden Kurven und stellen dar, wie sie gewinnbringend mit Geometriebäumen verknüpft und so für die Ablauf- und Datenorganisation nutzbar gemacht werden können.

Kapitel 3 ist der Darstellung der von uns verwendeten mathematischen und numerischen Konzepte gewidmet. Wir gehen dabei auf die Theorie der Finite-Elemente-Verfahren ein und beschreiben Ideen und Ansätze von Mehrgitterverfahren. Von besonderer Bedeutung ist hier die Wahl des Ansatzraumes für das Finite-Elemente-Verfahren, weil dadurch bereits die Weichen für die hohe Effizienz unserer Verfahren gestellt werden. Wir zeigen, dass durch die Entscheidung für ein hierarchisches Erzeugendensystem zusammen mit einem additiven Mehrgitterverfahren eine Kopplung der Numerik an die aus der Theorie der Raumfüllenden Kurven abgeleiteten Ideen zur Programm- und Datenorganisation auf natürliche Weise möglich ist. Ein weiterer Schwerpunkt dieses Abschnitts sind einige Überlegungen zur Lokalität von Daten und Berechnungen, insbesondere der Finite-Elemente-Sterne und der Mehrgitter-Interpolationen und -Restriktionen, die ebenfalls für die erzielbare Effizienz des Algorithmus im Speicher eines realen Rechners von entscheidender Bedeutung sind. Abschließend stellen wir zwei Modellgleichungen, die Poisson- und die Stokes-Gleichung vor, die wir für unsere numerischen Experimente verwendet haben.

Aus dem Konzept, Raumfüllende Kurven zur Organisation von Daten in Rechengittern zu verwenden, ergibt sich auf natürliche Weise der Ansatz, Gitterpunkte in speziellen Datenstrukturen – den Kellern – zu speichern. In **Kapitel 4** motivieren wir diesen Ansatz und zeigen auf, wie sich aus der durch die Raumfüllende Kurve vermittelten Linearisierung der Durchlaufreihenfolge durch die Zellen unserer Rechengitter Zugriffsregeln auf in Kellern gespeicherte Daten formulieren lassen. Dies wird insbesondere anhand eines Mehrgitterverfahrens auf einem hierarchischen Erzeugendensystems gezeigt. Wir begründen, warum die Speicherung der Daten in Kellern bezüglich der Speichereffizienz sinnvoll erscheint, und stellen exemplarisch dar, wie sich deterministische und effizient programmierbare „Regelsätze“ für Kellerzugriffe ableiten und implementieren lassen.

Nach den Vorüberlegungen in den Kapiteln 2 bis 4 stellen wir in **Kapitel 5** den neuartigen Algorithmus vor, der die vorgestellten Ideen zu einem hocheffizienten und zugleich flexiblen Programm zur Lösung Partieller Differenzialgleichungen verknüpft. Wir zeigen das Zusammenspiel der eingegangenen Konzepte und gehen auf die konkrete Implementierung zur Lösung der Poisson-Gleichung und der Stokes-Gleichung ein.

Kapitel 6 stellt schließlich die Ergebnisse der numerischen Experimente dar, die wir mit unserem Programm erzielen konnten. Wir beschreiben zunächst einige Grundlagen zu den Speicherhierarchien moderner Prozessoren und den dort implementier-

ten Algorithmen für Speicherzugriffe. Danach stellen wir Werkzeuge zur Simulation und Messung des Speicherzugriffsverhaltens von Programmen vor und gehen kurz auf plattformspezifische Compiler und Compileroptionen zur Verbesserung des Laufzeitverhaltens ein. Abschließend präsentieren wir die Ergebnisse unserer Beispielrechnungen für die Poisson- und die Stokes-Gleichung und zeigen dabei, dass sowohl die an die numerische Effizienz – charakterisiert durch Mehrgittereffizienz und Konvergenzgeschwindigkeit – als auch die an die Speichereffizienz – ausgedrückt durch extrem hohe Werte für erfolgreiche L2-Cache-Zugriffe – gestellten Forderungen mehr als erfüllt sind, ohne dass dabei auf die Flexibilität adaptiver Rechengitter und komplizierterer Geometrien des Rechengebiets verzichtet werden muss.

Schließlich geben wir in **Kapitel 7** eine kurze Zusammenfassung der bisher realisierten Implementierung und zeigen Erweiterungsmöglichkeiten auf, die zum Teil bereits in der Umsetzung begriffen sind.

*

An dieser Stelle möchte ich mich bei allen bedanken, die direkt und indirekt zum Gelingen dieser Arbeit und der ihr zugrunde liegenden Forschung und Entwicklung beigetragen haben:

Zu aller erst gilt mein großer Dank meinem Doktorvater und Betreuer Prof. Dr. Christoph Zenger, der es verstanden hat, meine Arbeit geschickt und zielführend immer in die richtige Richtung zu lenken, und der stets ein offenes Ohr sowohl für Bedenken und kritische Fragen als auch für überschwängliche Erfolgsmeldungen und Vorschläge hatte.

Weiterhin gilt mein Dank allen Kollegen in unserer Gruppe und an anderen Lehrstühlen unseres Instituts, die durch gemeinsame Forschung und Projektarbeit ihren Beitrag zu den erzielten Ergebnissen geleistet haben. Hierbei gilt mein besonderer Dank meiner Kollegin Miriam Mehl. Sie konnte durch ihre kritische Durchsicht meiner Arbeit viele wertvolle und konstruktive Hinweise geben, die wesentlich zur Qualität der Darstellung unserer Ergebnisse beigetragen haben. Besonders auch während der Entwicklung der Methodik und der Programme selbst war sie stets ein wertvoller Ratgeber und hat durch viele „richtige Fragen zur richtigen Zeit“ entscheidend zum Erfolg dieses Projekts beigetragen. Weiterhin bedanke ich mich bei meinen Kollegen Markus Pögl und Andreas Krahnke, die in unserer Gruppe am gleichen Projekt mit anderen Schwerpunktsetzungen arbeiten. In unseren Diskussionen konnten wir viele Zusammenhänge und Hintergründe beleuchten und gemeinsam verstehen. Michael Bader konnte alle meine Fragen zu den Details der Dokumentenerstellung mit LaTeX fundiert beantworten, was die optische Qualität dieser Arbeit entschieden positiv beeinflusst hat. Ebenso möchte ich mich bei den Kollegen Jürgen

Jeitner, Martin Mairandres und Josef Weidendorfer vom Lehrstuhl für Rechnertechnik und Rechnerorganisation (Prof. Dr. Arndt Bode) bedanken, die entscheidend zu meinem Verständnis der Details moderner Cache-Hierarchien beigetragen haben und die uns den Zugang zu Hochleistungsrechnern mit entsprechenden Werkzeugen zur Analyse unserer Programme zur Verfügung stellen.

Ich bedanke mich bei den Mitarbeitern unseres Lehrstuhls, die alle durch Offenheit, Freundlichkeit und Fairness zu einem äußerst angenehmen Arbeitsklima beigetragen haben, in dem es mir möglich war, motiviert und in den wichtigen Momenten unbelastet von anderen Dingen diese Arbeit zu erstellen. Außerdem bedanke ich mich bei den Studenten Reinhard Stein, Mario Gleirscher und Sebastian Kienzl, die im Rahmen ihrer Arbeiten an unserem Lehrstuhl wichtige Unterstützung für die Erzielung und die Bewertung der Ergebnisse dieser Arbeit geleistet haben.

Zuletzt und zugleich am allermeisten gilt mein Dank meiner Frau, die mir durch Geduld und Großzügigkeit vor allem den zeitlichen Spielraum gegeben hat, diese Arbeit zu erstellen. Stets hat sie mich in dieser Zeit bestärkt und unterstützt, hat sich geduldig viele in der jeweiligen Situation unverständliche Details angehört und die teilweise nicht familienfreundlichen Arbeitszeiten ertragen. Nicht zuletzt ihre abschließende und strenge Durchsicht der Arbeit hat noch einige Stellen mangelnd präziser Darstellung aufgedeckt, auch dafür gilt ihr mein besonderer Dank.

2 Raumfüllende Kurven und Geometrie

Ausgangspunkt für die Entwicklung der Methodik dieser Arbeit ist eine diskrete Geometriebeschreibung. Darunter verstehen wir eine möglichst effiziente Repräsentation des Rechengebiets, die neben der reinen Geometrieinformation auch die physikalischen Eigenschaften der Gebietsränder beinhaltet.

2.1 Geometriebeschreibung

In Literatur, Forschung und existierenden Programmen werden viele verschiedene Modelle zur Beschreibung von Gebietsgeometrien verwendet. Für die numerische Simulation sind diejenigen Modelle von Interesse, die eine effiziente Darstellung der geometrischen und topologischen Struktur des Simulationsgebietes auf dem Rechner ermöglichen. Darüber hinaus müssen die relevanten Merkmale der physikalischen Wechselwirkungen der einzelnen Objekte im Simulationsgebiet untereinander sowie des Simulationsgebietes mit seiner Umgebung erfassbar sein. Auch Änderungen der Geometrie und Topologie sollten ohne großen Aufwand darstellbar sein. Als Beispiel denke man etwa an Fluid- Struktur-Wechselwirkungen wie in [18] beschrieben, bei denen der zeitliche Verlauf der Strömungssimulation die Geometrie der beteiligten Objekte verändert und sich damit die für die Simulation der Strömung relevante Geometrie im Simulationsverlauf ändert.

In [9] werden hierzu verschiedene Möglichkeiten wie etwa Volumen-Modelle (Normzellen-Aufzählungsschema, Zellzerlegungsschema, CSG) oder Oberflächenmodelle (Drahtgittermodell, Dreiecksnetze) dargestellt. Dabei setzt man sinnvollerweise voraus, dass sich die zu rechnenden Geometrien aus einer (endlichen) Anzahl von Objekten (zum Beispiel geometrischen Primitiven) zusammensetzen, deren Gesamtheit eine Einteilung des Gebiets in Bereiche erlaubt, auf denen bestimmte physikalische Gesetze gelten, die beispielsweise durch Partielle Differenzialgleichungen beschreibbar sind.

Im einfachsten – und in dieser Arbeit betrachteten – Fall wird der gesamte Raum in „Innen“ und „Außen“ eingeteilt, im Inneren ist ein Satz von Gleichungen zu

erfüllen, etwa die Navier-Stokes-Gleichungen zur Strömungssimulation, im Äußeren ist nichts zu tun. Diese Geometrien lassen sich sehr gut mit Hilfe so genannter raumpartitionierender Strukturen beschreiben [9]. Dabei wird das Gesamtgebiet in Zellen zerlegt, die solange rekursiv unterteilt werden, bis sie ganz zu „Innen“ oder „Außen“ gehören. Dieser Prozess wird bis zur gewünschten Approximationsgüte beziehungsweise einer vorgegebenen Minimalgröße der kleinsten Zellen fortgesetzt (vgl. Abb. 2.1). Die Zerlegung lässt sich durch hierarchische Strukturen wie etwa Oktalbäume darstellen.

Oktalbäume lassen sich sehr sparsam speichern, Grundoperationen wie Vereinigung und Schnitt von Teilbäumen sind effizient implementierbar, Ansätze für Adaption und Parallelisierung sind inhärent vorhanden. Ebenso lassen sich weitere notwendige Eigenschaften leicht erzielen, zum Beispiel Beschränkungen im Größenunterschied benachbarter Zellen. Dies kann aus numerischer Sicht sinnvoll sein und lässt sich durch Balancierungsvorschriften an den zu erzeugenden Baum leicht erzielen. Wie in [9] gezeigt, ist die hohe Effizienz auf die Organisationsprinzipien dieser Strukturen zurückzuführen, von besonderer Bedeutung sind dabei iterative Formulierungen, Rekursion und Hierarchie.

Abbildung 2.1 zeigt den Weg von einer einfachen Geometrie zu einem adaptiven Quadtree (rekursiv aufgebauter Zellbaum, bei dem in jedem Schritt in beide Dimensionen durch Teilung durch zwei verfeinert wird), bei dem jede Zelle durch die Eigenschaft „innen/außen/Randzelle“ attribuiert ist. Dabei wird eine Zelle als Randzelle markiert, wenn sie nach Abbruch der Rekursion weder ganz zu „Innen“ noch zu „Außen“ gehört, das heißt wenn der Rand durch sie hindurch läuft. Natürlich lassen sich die Zellen mit weiteren Eigenschaften versehen, zum Beispiel Materialeigenschaften, zur Zelle gehörige numerische Freiheitsgrade und vieles mehr.

In dieser Arbeit verwenden wir als Zellen Quadrate auf einem kartesischen Gitter, die hierarchisch in Baumstrukturen angeordnet werden. In diversen Arbeiten zu diesem Thema werden weitere Eigenschaften und Möglichkeiten solcher Baumkonzepte dargestellt, wir werden uns in dieser Arbeit auf die für unsere Implementierungen tatsächlich notwendigen beschränken.

Zu erwähnen ist hier noch, dass die Möglichkeit besteht, für die Rechnung andere Zellen respektive Gebietszerlegungen zu verwenden. Zum Beispiel wird manchmal der Rand bezogen auf seine Zellen vergröbert und das Innere stärker verfeinert oder es wird zum Rechnen eine andere Struktur als Bäume verwendet. In unserem Fall ist jedoch sowohl die Geometrieeingabe als auch das Rechengitter ein Oktalbaum. In den meisten Fällen wird auch der Geometriebaum direkt als Rechenbaum verwendet, nur in einigen Ausnahmefällen werden im Inneren zusätzliche Zellen eingefügt.

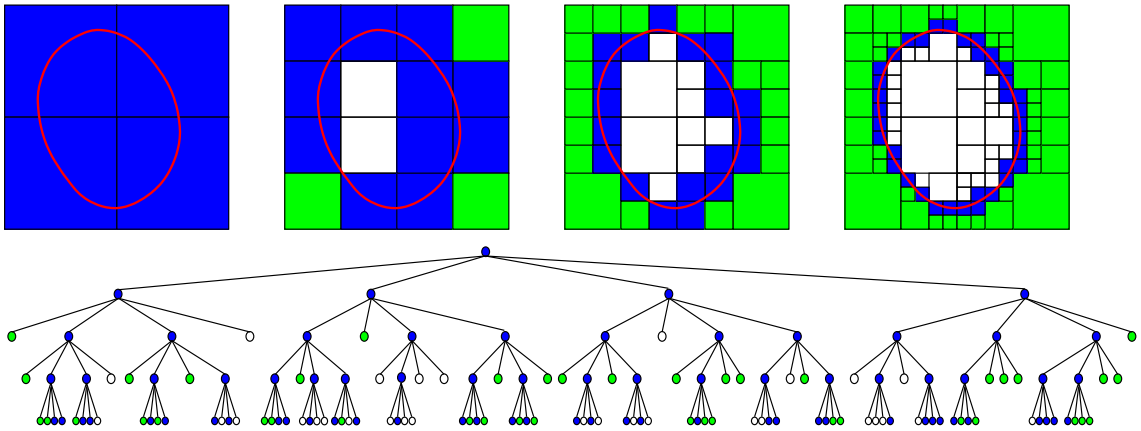

Abbildung 2.1: Rekursive adaptive Partitionierung in quadratische Zellen. Die Attribuierung der Zellen ist mit Farben realisiert: weiß=innen, grün=außen, blau=Randzelle.

2.2 Raumfüllende Kurven

Raumfüllende Kurven spielen in unserem Ansatz zur numerischen Simulation insofern eine entscheidende Rolle, als sie gewissermaßen als „Wegbeschreibung“ durch das Rechengebiet respektive die Zellen des Oktalbaums verwendet werden. Im folgenden Abschnitt wird nach der Darstellung einiger Grundlagen gezeigt, warum das im Zusammenhang mit raumpartitionierenden Strukturen und Baumdarstellungen der Rechengeometrie interessant und sinnvoll ist. Weiter wird die Auswahl der Peano-Kurve für unsere Algorithmen begründet.

2.2.1 Geschichte und Definitionen

Die folgende Darstellung lehnt sich an die von Sagan in [20] an, dort wird allerdings der mathematischen Theorie und exakten Beweisen mehr Würdigung zuteil, während hier nur die für unsere Zwecke wichtigsten Definitionen und Eigenschaften erwähnt werden.

Einige Jahre vor und nach der vorletzten Jahrhundertwende erlebte die Theorie der raumfüllenden Kurven ihre Geburtsstunde und ihre Blütezeit. Ausgehend von Cantors Erkenntnis, dass zwei endlich-dimensionale Mannigfaltigkeiten unabhängig von ihrer Dimensionalität dieselbe Kardinalität besitzen (1878) (was beispielsweise bedeutet, dass sich das Einheitsintervall $[0,1]$ bijektiv auf das Einheitsquadrat $[0,1]^2$ oder den Einheitswürfel $[0,1]^3$ abbilden lässt), begann man Funktionen zu suchen, die eine solche 1-zu-1-Beziehung zwischen zwei geeigneten Mengen herstellen. Zur Beschreibung derartiger Abbildungen benötigen wir die folgende Definitionen.

Definition 2.1 (Direktes Bild) *Sei $\mathbb{E}^n$ der n-dimensionale Euklidische Raum. Wenn f eine Abbildung einer Untermenge von $\mathbb{E}^m$ nach $\mathbb{E}^n$ ist, dann heißt*

$$f_*(A) = \{f(x) \in R(f) \mid x \in A \cap D(f)\}, \tag{2.1}$$

wobei $A \subseteq \mathbb{E}^m$, das direkte Bild von A unter f. $D(f)$ bezeichnet den Definitions- und $R(f)$ den Wertebereich von f.

Definition 2.2 (Kurve) *Wenn $f : I = [0,1] \mapsto \mathbb{E}^n$ stetig ist, dann heißt das direkte Bild $f_*(I)$ Kurve. $f(0)$ wird als Anfangs- und $f(1)$ als Endpunkt der Kurve bezeichnet.*

Definition 2.3 (Raumfüllende Kurve) *Wenn $f : I \mapsto \mathbb{E}^n$, $n \geq 2$, stetig ist und sein direktes Bild positiven Jordan-Inhalt hat, dann heißt das direkte Bild $f_*(I)$ raumfüllende Kurve.*

Ernst Netto hat 1879 bewiesen, dass eine Abbildung f, die eine raumfüllende Kurve erzeugt, nicht bijektiv sein kann. Die Frage war jetzt also: „Gibt es raumfüllende Kurven, also Kurven die jeden Punkt einer Fläche durchlaufen und somit positiven Jordan-Inhalt besitzen?“.

In den folgenden Jahren wurden verschiedene raumfüllende Kurven entdeckt. Die erste wurde 1890 von Guiseppe Peano vorgeschlagen, von besonderer Bedeutung für unsere Anwendung war jedoch die von David Hilbert 1891 vorgeschlagene Kurve, weil er für den Beweis, dass sie stetig und raumfüllend ist, geometrische Argumente verwendet hat, die im Kontext von Raumpartitionierung durch Zellen äußerst hilfreich ist. Die bekanntesten weiteren Kurven stammen von Henri Leon Lebesgue (1904) und Waclaw Sierpinski (1912).

2.2.2 Die Hilbert-Kurve und ihre geometrische Erzeugung

Zur Konstruktion einer raumfüllenden Kurve schlägt Hilbert folgendes Vorgehen vor:
Wenn das Intervall $I = [0,1]$ stetig auf das Quadrat $Q = [0,1]^2$ abgebildet werden kann, dann kann, nachdem man I in vier kongruente Teilintervalle und Q in vier kongruente Teilquadrate aufgeteilt hat, jedes Teilintervalle stetig auf eines der vier Teilquadrate abgebildet werden. Das selbe Argument gilt, wenn man jedes der vier Teilintervalle und jedes der vier Teilquadrate wieder kongruent in vier neue Subintervalle beziehungsweise -quadrate aufteilt und immer so fort. Wenn man das bis ins Unendliche fortsetzt, dann sind I und Q in 2^{2n} für $n = 1, 2, 3, \ldots$ kongruente Kopien aufgeteilt worden. Hilbert hat gezeigt, dass die Teilquadrate sowie die Anfangs- und Endpunkte der Kurven auf den Teilquadraten so angeordnet werden können,

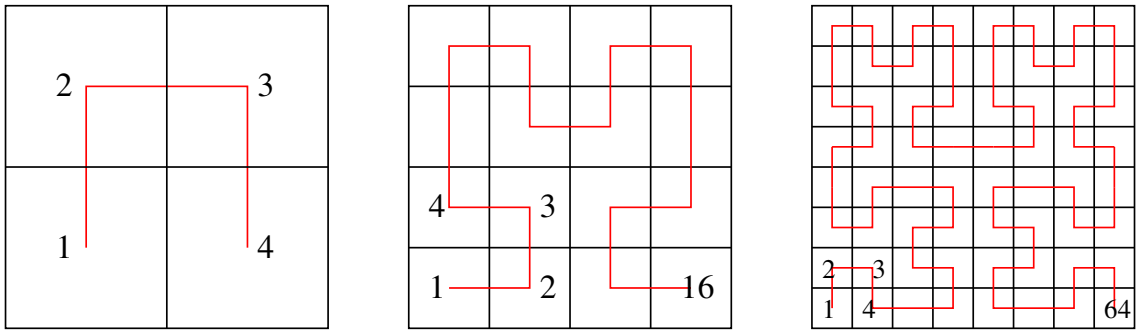

Abbildung 2.2: Geometrische Erzeugung der Hilbert-Kurve.

dass aneinander hängende Teilintervalle aufeinander folgenden Teilquadraten entsprechen und die Kurven auf den Teilquadraten zu einer stetigen Kurve auf der gesamten Fläche zusammengesetzt werden können. Die Hilbert-Kurve entsteht also als Grenzwert eines rekursiven Unterteilungsprozesses: Eine festgelegte Anfangsunterteilung des Intervalls sowie des Quadrats mit definierter Zuordnung aufeinander folgender Intervalle zu aneinander hängenden Quadraten wird rekursiv auf den entstehenden Teilquadraten wiederholt, wobei bei der Zuordnung von Intervallen zu Quadraten beim Übergang zur nächsten Verfeinerungsstufe Drehungen und Spiegelungen erlaubt sind, um das Zusammentreffen von End- und Anfangspunkten der Teilkurven zu gewährleisten. In Abbildung 2.2 ist das Ergebnis von Hilberts Bemühungen für die ersten drei rekursiven Unterteilungen dargestellt. Mathematisch lassen sich die Hilbert-Kurve und ihre erzeugende Abbildung wie folgt definieren. Wegen Cantors Erkenntnis ist es unerheblich, welches Intervall und welche Fläche $Q \subset \mathbb{E}^n$ man betrachtet, lediglich die Darstellung wird komplizierter.

Definition 2.4 (Hilbert-Kurve) *Jedes $t \in I$ ist eindeutig bestimmt durch eine Folge von geschachtelten abgeschlossenen Intervallen, die durch die schrittweise Partitionierung von I wie oben beschrieben entstehen. Die Länge dieser Intervalle schrumpft auf 0. Zu einer solchen Folge gehört eine eindeutige Folge von geschachtelten abgeschlossenen Quadraten, deren Diagonalen zu einem Punkt schrumpfen, der einen eindeutigen Ort in Q darstellt, das Bild $f_h(t)$ von t. Dann nennen wir das direkte Bild $f_{h*}(I)$ die Hilbert-Kurve.*

Hilbert bedient sich in dem Beweis, dass die vorgeschlagene Kurve – also das der Grenzwert der Intervall- und Quadratschachtelungen für $n \to \infty$ – eine raumfüllende Kurve ist, der Eigenschaften dieser Einschließungen. Zum Beweis, dass die zugrunde liegende Abbildung f_h *surjektiv* ist, führt er an, dass jeder Punkt in Q in einer Folge von ineinander geschachtelten Quadraten liegt, deren Diagonalen zu einem Punkt schrumpfen. Zu dieser Folge von Quadraten gehört eine Folge von geschachtelten Subintervallen, die zu einem Punkt schrumpfen, dessen Bild unter f_h

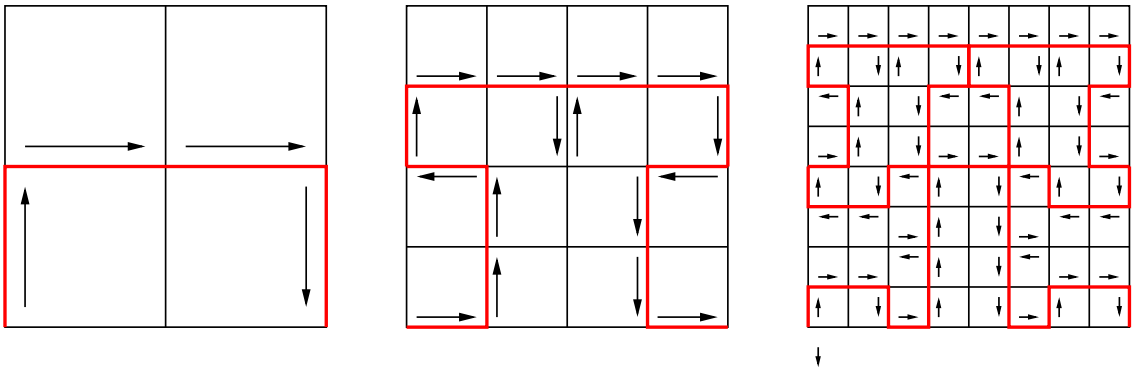

Abbildung 2.3: Approximierende Polygone der Hilbert-Kurve.

gerade der betrachtete Punkt in Q ist. Es sei darauf hingewiesen, dass Punkte, die auf Kanten oder Ecken der Quadrate liegen, kein eindeutiges Urbild besitzen, da sie zu mehr als einer Quadratschachtelung gehören. Dies ist vor dem Hintergrund von Nettos Satz über die Nicht-Bijektivität solcher Abbildungen auch nicht weiter überraschend. Die *Stetigkeit* der Abbildung f_h folgt aus Betrachtungen über Intervalllängen und Größen der zugehörigen Quadrate. Da das direkte Bild von I unter f_h gerade Q ist und der Jordan-Inhalt $J_2(Q) = 1$ ist, folgt zusammen mit Stetigkeit und Surjektivität, dass die Hilbert-Kurve eine raumfüllende Kurve ist. Ebenso leicht lässt sich zeigen, dass die Hilbert-Kurve nirgends differenzierbar ist [20].

2.2.3 Approximierende Polygone

Identifiziert man jedes der Quadrate, die in obiger Schachtelung entstehen, mit einem Punkt, so ergibt die Verbindung dieser Punkte ein polygonale Näherung für die Hilbert-Kurve.

Definition 2.5 (Approximierendes Polygon) *: Die polygonale Linie, die die Punkte*

$$f_h(0), f_h(1/2^{2n}), f_h(2/2^{2n}), f_h(3/2^{2n}, \ldots, f_h((2^{2n}-1)/2^{2n}), f_h(1) \tag{2.2}$$

verbindet, heißt das n-te approximierende Polygon der Hilbert-Kurve.

Abbildung 2.3 zeigt die ersten drei approximierenden Polygone gemäß obiger Definition. Für die Folge dieser Polygone lässt sich zeigen, dass sie gleichmäßig gegen die Hilbert-Kurve konvergieren. Q wird in 2^{2n} kongruente Teilquadrate partitioniert, von denen jedes die Seitenlänge $1/2^n$ hat. Jedes Quadrat enthält genau eine Segment des approximierenden Polygons, jedes Segment hat die Länge $1/2^n$ und somit ergibt sich die Länge des approximierenden Polygons zu 2^n.

Das oben definierte Polygon lässt sich rekursiv definieren, indem man der gerichteten Linie von $(0,0)$ nach $(1,0)$ bei einer Verfeinerung des Quadrats das erste approximierende Polygon als Rekursionsvorschrift zuordnet. Durch Drehung des Polygons lassen sich entsprechende Vorschriften für die neu entstehenden gerichteten Linien darstellen. Insgesamt ergeben sich vier Beziehungen, mit denen das approximierende Polygon für beliebig große n rekursiv deterministisch erzeugt werden kann. Somit kann man die Hilbert-Kurve respektive ihre approximierenden Polygone als Fraktale ansehen mit der gerichteten Linie von $(0,0)$ nach $(1,0)$ als Leitmotiv.

2.2.4 Bezug zu kartesischen Simulationsgittern

Nach obigen Betrachtungen lässt sich sofort ein sehr nahe liegender Bezug zu kartesischen Gittern herstellen, wie sie in der numerischen Simulation verwendet werden:

Identifiziert man die bei den Schachtelungen entstehend Quadrate mit quadratischen Zellen eines quadratischen Rechengebiets, so vermittelt das „passende“ n-te approximierende Polygon der Hilbert-Kurve eine lineare Aufzählung der Zellen, die als Definition einer Durchlaufreihenfolge für numerische Lösungsalgorithmen verwendet werden kann.[1]

Die Durchlaufreihenfolge entlang der Kurve bedingt, dass immer benachbarte Zellen nacheinander besucht werden. Dies ist wie später noch dargestellt von großer Bedeutung für die Lokalität von Daten, die auf den Zellen oder deren Eckpunkten gespeichert werden (siehe Kapitel 3 und 4). Die rekursive Konstruktion der Kurve lässt sich direkt kombinieren mit einer rekursiven Verfeinerungsstrategie der Zellen des Rechengebiets. In diesem Sinne ist die Kurve „oktalbaumkompatibel“ , das heißt wird ein Teilquadrat betreten, werden alle in ihm erzeugten Teilquadrate abgearbeitet, was bezüglich des Oktalbaums bedeutet, dass nach einer Depth-First-Strategie Unterbäume jeweils komplett bearbeitet werden, bevor ein neuer Unterbaum gleichen Levels an der Reihe ist. In Abbildung 2.4 ist diese Tatsache dargestellt, die Blätter des Baumes sind mit den Zellnummern der Hilbert-Kurve attribuiert. Auch für adaptive Gitter, auf die beispielsweise zeilenweise Nummerierungen nicht mehr anwendbar sind, kann so eine lineare Reihenfolge der Zellen festgelegt werden. Im Oktalbaum entsteht ein adaptiv verfeinertes Gebiet durch Einhängen eines neuen Unterbaums, der die neuen feineren Zellen enthält. Dies überträgt sich unmittelbar

[1] Ab jetzt werden die Begriffe „approximierendes Polygon der Hilbert-Kurve “ und „Hilbert-Kurve“ beziehungsweise „Kurve“ gleichberechtigt verwendet. Wenn also von einer Hilbert-Kurve oder Kurve gesprochen wird, ist immer die endliche Repräsentation durch ein approximierendes Polygon gemeint.

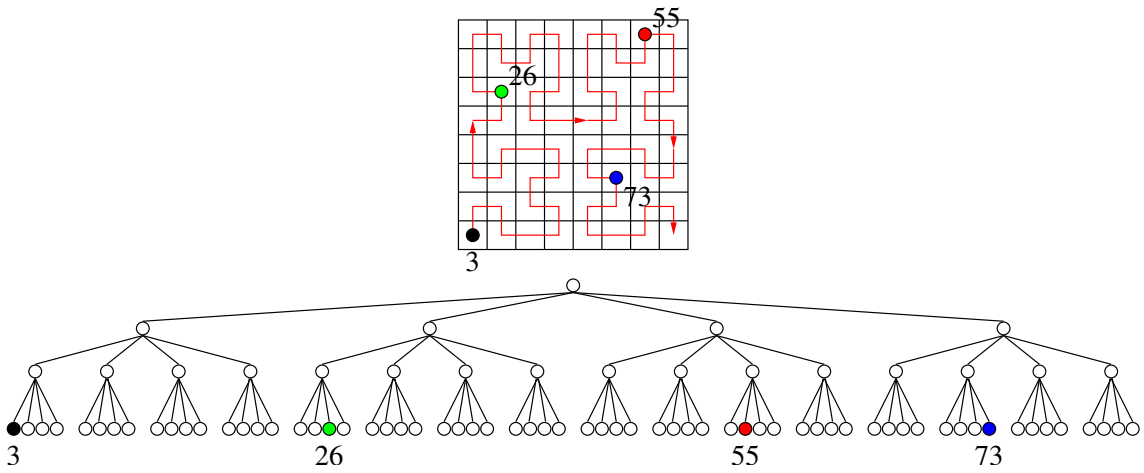

Abbildung 2.4: Reguläres Rechengebiet mit Hilbert-Kurve und zugehöriger Oktalbaum.

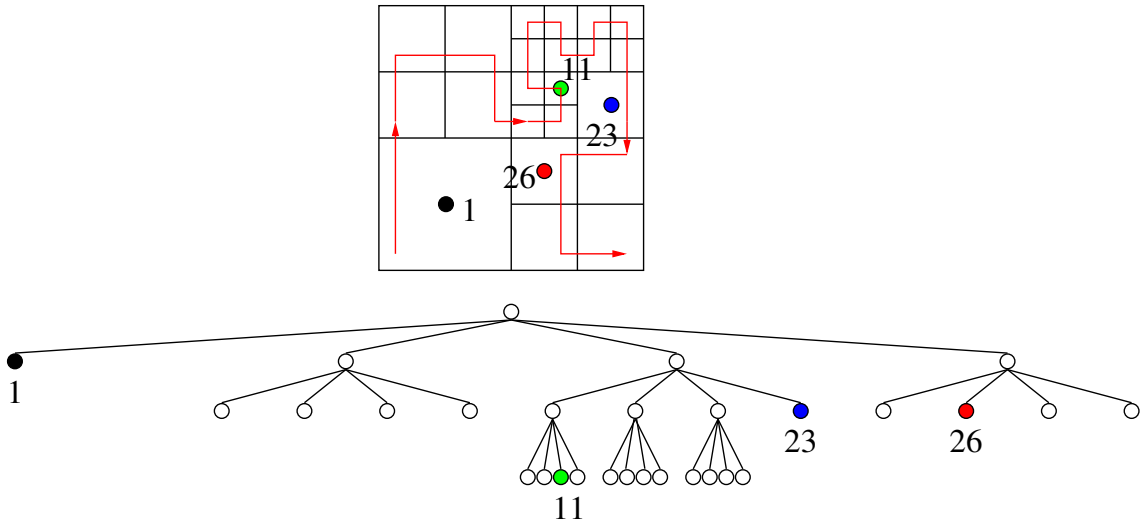

Abbildung 2.5: Adaptives Rechengebiet mit Hilbert-Kurve und zugehöriger Oktalbaum.

auf die Kurve durch rekursives Einfügen von Teilquadraten an der entsprechenden Stelle. Abbildung 2.5 zeigt dazu ein Beispiel.[2]

Neben adaptiven Gittern werden für effiziente numerische Simulationsprogramme Multi-Level-Ansätze verwendet. Auch hier zeigt sich die Stärke der Kombination von Oktalbaumrepräsentation und raumfüllender Kurve: Hält man etwa physikalische Information nicht nur auf den Blättern des Baumes – also dem feinsten Level – sondern auch auf den inneren Knoten des Baumes, so lassen sich zusammen mit einer geeignet gespeicherten rekursiven Version der Kurve auch Wege über die Zellen nur eines bestimmten Levels definieren und damit beispielsweise Mehrgitterverfahren realisieren. Die dazu nötige Repräsentation der Daten in einem hierarchischen Erzeugendensystem wird in Kapitel 3 erläutert.

In allen Fällen vermittelt die Kurve eine Linearisierung des Zellbaums über alle Level, was nützlich für serielle Rechnungen ist (siehe Kapitel 4), aber auch klare Strategie zur Parallelisierung liefert: Der linearisierte Baum kann auf einfache Weise (im wesentlichen) gleichmäßig auf die zur Verfügung stehende Zahl von Prozessoren verteilt werden. Insbesondere ist eine Neuverteilung der Arbeit auf die Prozessoren etwa nach einer adaptiven Veränderung des Rechengebiets durch die Strukturiertheit der Kurve leicht möglich. Aufgrund der guten Lokalitätseigenschaften der Kurve (komplette Abarbeitung eines Teilquadrats, bevor das nächste betreten wird) ist die so gewonnene Partitionierung bezüglich der Schnittflächen – also des Kommunikationsaufwands – sogar quasi-optimal [28].

2.2.5 Die Peano-Kurve

Guiseppe Peano (1858-1932) war im Jahr 1878 der erste, der eine Abbildung angeben konnte, die I stetig und surjektiv auf Q abbildet und die im obigen Sinne eine raumfüllende Kurve ist:

Satz 2.1 (Peano-Kurve) *Seien t_j ternäre Ziffern, also $t_j \in \{0,1,2\}$, k mit $kt_j := 2 - t_j$ ein Operator auf diesen Ziffern und k^{t_i} die t_i-te Iterierte dieses Operators. Dann ist das direkte Bild von*

$$f_p(0_{\dot{3}} t_1 t_2 t_3 t_4 \ldots) = \begin{pmatrix} 0_{\dot{3}}(k^{t_2} t_3)(k^{t_2+t_4} t_5)\ldots \\ 0_{\dot{3}}(k^{t_1} t_2)(k^{t_1+t_3} t_4)\ldots \end{pmatrix} \tag{2.3}$$

stetig, surjektiv und eine raumfüllende Kurve. Das Bild $f_{p*}(I)$ heißt Peano-Kurve.

[2] Genau genommen werden im Falle der Adaption Teile von unterschiedlichen approximierenden Polygonen aneinander gesetzt. Es ist aber leicht einzusehen, dass dies auch problemlos möglich ist, da jedes Teilstück des n-ten Polygons rekursiv und vor allem stetig ersetzt werden kann durch vier Teilstücke des $(n+1)$-ten Polygons. Auch die Vorstellung, dass die rekursive Erzeugung des $(n+1)$-ten Polygons aus dem n-ten unvollständig, das heißt nicht für alle Segmente des n-ten Polygons durchgeführt wird, ist hier hilfreich.

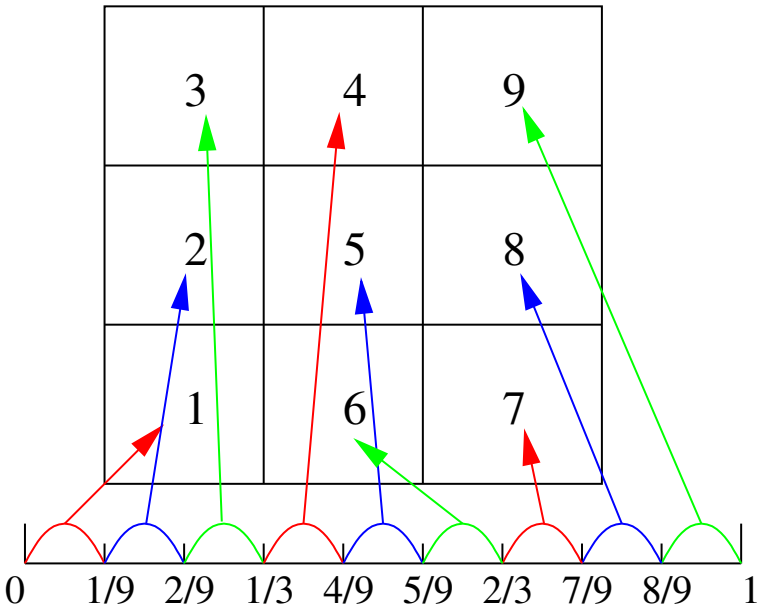

Abbildung 2.6: Abbildung auf das Quadrat mit der Peano-Kurve.

Der Beweis wird aus Gründen der Übersichtlichkeit hier weggelassen. Obwohl Peano offensichtlich nie über eine geometrische Erzeugung seiner Kurve nachgedacht hat, lässt sich jedoch das von Hilbert vorgeschlagene Vorgehen unmittelbar übertragen. Gemäß der Definition von f_p gilt

$$f_p(0_{\dot{3}}00t_3t_4\ldots) = \begin{pmatrix} 0_{\dot{3}}0\xi_2\xi_3\xi_4\ldots \\ 0_{\dot{3}}0\eta_2\eta_3\eta_4\ldots \end{pmatrix}, \tag{2.4}$$

was nichts anderes bedeutet, als dass das Intervall $[0, 1/9]$ abgebildet wird auf das Quadrat 1 in Abbildung 2.6 mit den vier Ecken $(0,0), (0,1/3), (1/3,0), (1/3,1/3)$. Allgemein: Das Intervall $[\frac{j-1}{9}, \frac{j}{9}]$ mit $j = 1,2,3,\ldots,9$ wird abgebildet auf das Quadrat Nummer j in Abbildung 2.6. Dies führt zu einer geometrischen Konstruktion mit Hilfe approximierender Polygone wie bei der Hilbert-Kurve. Das Intervall I wird rekursiv in 3^{2n}, $n = 1,2,3,\ldots$ kongruente Teilintervalle partitioniert, die auf 3^{2n} kongruente Teilquadrate von Q abgebildet werden. Die ersten drei Schritte (also die ersten drei approximierenden Polygone) sind in Abbildung 2.7 dargestellt. Hier wird ein Quadrat für das approximierende Polygon über seinen Mittelpunkt identifiziert.[3]

[3] Abbildungen 2.6 und 2.7 zeigen unterschiedliche Orientierungen der Peano-Kurve. Dies ist lediglich bedingt durch ein unterschiedliches Leitmotiv. Eine Umformulierung der Definition von f_p ist möglich, aus historischen Gründen wird die Definition von Peano verwendet. Die Programme und die weitere Darstellung dieser Arbeit gehen jedoch von dem in 2.7 verwendeten Leitmotiv aus.

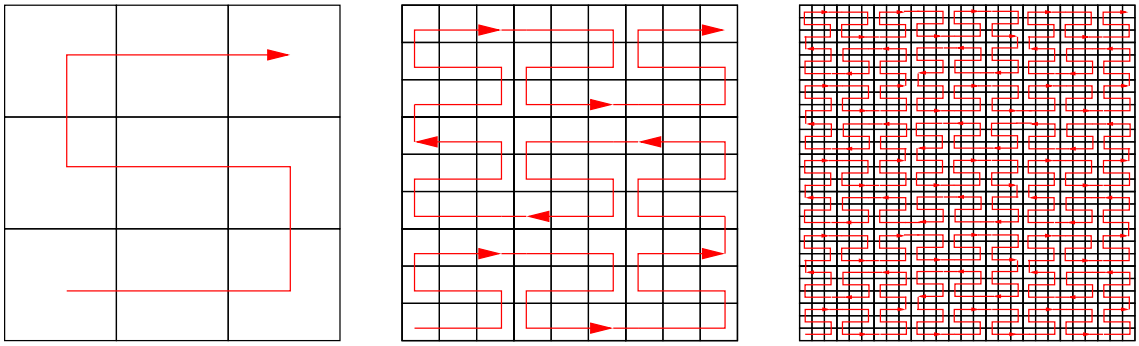

Abbildung 2.7: Geometrische Konstruktion der Peano-Kurve.

Analog zur Hilbert-Kurve kann auch die Peano-Kurve verwendet werden, um eine Reihenfolge eines Gebietsdurchlaufs zu vermitteln. Die im Zusammenhang mit der Hilbert-Kurve angeführten Vorteile lassen sich unmittelbar auf die Peano-Kurve übertragen. Statt eines Oktalbaums, dessen Anzahl von Söhnen durch eine Teilung der Quadrate/Würfel/Hyperwürfel durch zwei je Raumrichtung bestimmt wird, entsteht hier eine raumpartitionierende Struktur, die auf einer Teilung durch drei je Raumrichtung basiert. Solche Strukturen bezeichnen wir unabhängig von dem gewählten Teilungsparameter allgemein als Spacetrees. Auch hier entsteht eine „spacetree-kompatible“ Struktur und die Lokalitätseigenschaften der Kurve sind qualitativ vergleichbar.

In dieser Arbeit wie auch in den angeschlossenen Arbeiten unserer Gruppe zu dieser Thematik wird immer die Peano-Kurve verwendet. Als Grund dafür sei hier nur erwähnt, dass die Peano-Kurve im Dreidimensionalen die für uns angenehme Eigenschaft hat, dass jede Projektion einer in einen Würfel einbeschriebenen Peano-Kurve auf die sechs Flächen dieses Würfels wieder eine Peano-Kurve ist. Dies hat für uns bezüglich der Effizienz der verwendeten Datenstrukturen große Bedeutung (siehe Kapitel 4). Ein weiterer Vorteil der Peano-Kurve ist, dass sie beziehungsweise ihre approximierenden Polygone gewissermaßen dimensionsiterativ formuliert werden können. Dies lässt eine Entwicklung von Algorithmen für mehr als drei Dimensionen zumindest als machbar erscheinen. Es ist uns nicht gelungen, eine Version der dreidimensionalen Hilbert-Kurve zu finden, die diese Eigenschaften hat. Für eine detailliertere Darstellung dieser Tatsachen sei auf die entstehenden Arbeiten zu diesem Thema in unserer Gruppe wie zum Beispiel [19] verwiesen, Details zur dreidimensionalen Hilbert-Kurve finden sich in [1].

Im Rahmen eines Interdisziplinären Projekts an unserem Lehrstuhl wurde von Mario Gleirscher und Sebastian Kienzl ein Gittergenerator programmiert, der den oben dargestellten Prinzipien folgt [10]. Dort werden `tiff`-Bilder, die farbcodiert Informationen wie Innen, Außen und Randtyp je Zelle enthalten, eingelesen und danach ein vollständiger Baum der Zellen über alle Level mit einbeschriebener raumfüllen-

der Kurve erzeugt. Dann wird unter Einhaltung gewisser Vorgaben wie beispielsweise Feinheit des Randes, maximaler Levelunterschied benachbarter Zellen usw. versucht, den Baum „auszudünnen“ das heißt ein adaptives Gitter zu erzeugen. Dieses wird dann sequentialisiert entlang der Kurve zur Weiterverarbeitung in ein File ausgegeben. Dabei trägt jede Zelle alle Informationen, die für eine effiziente Verarbeitung in einem numerischen Löser, wie er im Rahmen dieser Arbeit entstanden ist, notwendig sind.

3 Finite-Elemente- und Mehrgitterverfahren

Nach der Darstellung der wichtigsten Grundlagen von Spacetree-Methoden und raumfüllenden Kurven, die in dieser Arbeit zur Erzeugung der für die Simulationsrechnung benötigten Datenstrukturen verwendet werden, soll nun auf die unserem Verfahren zugrunde liegende numerische Mathematik eingegangen werden. Dafür werden zuerst einige Grundlagen zur Methode der Finiten Elemente angeführt. Danach wird begründet, warum der Einsatz eines hierarchischen Erzeugendensystems einer nodalen Basis vorzuziehen ist, um unser Ziel eines effizienten additiven Mehrgitterverfahrens für beliebige und insbesondere adaptive Gitter zu erreichen. Schließlich wird aufgezeigt, wie in diesem Kontext ein Verfahren für zwei einfache Modellgleichungen – die Poisson-Gleichung und die Stokes-Gleichung – mit Hilfe lokaler zellorientiert distributierter 9-Punkte-Sterne zu formulieren ist. Dieser zellorientierte Ansatz ist von zentraler Bedeutung für FE-Diskretisierungen auf adaptiven Gittern im Allgemeinen und für unseren Algorithmus, der das Simulationsgebiet zellenweise durchläuft, im Speziellen.

3.1 Grundlagen der Finite-Elemente-Methode für partielle Differentialgleichungen

Zur Vereinfachung der Darstellung beschränken wir uns auf zweidimensionale elliptische partielle Differentialgleichungen auf $\Omega \subset [0,1]^2$ mit homogenen Dirichlet-Randbedingungen, die seit langem bekannten Beweise der Sätze werden im Allgemeinen weggelassen und es wird auf [5] verwiesen. Sei also

$$Lu = f \text{ auf } \Omega \tag{3.1}$$

$$u_0 = 0 \text{ auf } \partial\Omega \tag{3.2}$$

mit geeigneten Randbedingungen u_0 und exakter Lösung u. Mit einer geeigneten Bilinearform kommt man zur Variationsformulierung des elliptischen Randwertproblems: Die Suche nach einer Funktion u, die die Differentialgleichung erfüllt wird zurückgeführt auf die Suche nach dem Minimum einer geeigneten Größe.

Satz 3.1 *Sei V eine linearer Raum und*

$$a \ : \ V \times V \to \mathbb{R} \tag{3.3}$$

eine symmetrische, positive Bilinearform, also $a(u,u) > 0 \ \forall u \in V, \ u \neq 0$. Ferner sei

$$l \ : \ V \to \mathbb{R} \tag{3.4}$$

ein lineares Funktional. Die Größe

$$J(v) := \frac{1}{2}a(v,v) - (l,v) \tag{3.5}$$

nimmt in V genau dann ihr Minimum bei u an, wenn

$$a(u,v) = (l,v) \ \forall v \in V \tag{3.6}$$

gilt. Außerdem gibt es höchstens eine Minimallösung.

Dieser Satz wird benötigt, um die folgende Aussage zu beweisen und damit den Zusammenhang zwischen der Minimierungsaufgabe und der Differentialgleichung herzustellen.

Satz 3.2 *Jede klassische Lösung der Randwertaufgabe*

$$-\sum_{i,k} \partial_i(a_{ik}\partial_k) + a_0 u \ = \ f \ in \ \Omega, \tag{3.7}$$

$$u \ = \ 0 \ auf \ \partial\Omega, \tag{3.8}$$

also eine Funktion, die (3.1) erfüllt, die Randwerte annimmt und für Dirichlet-Randwerte in $C^2(\Omega) \cap C^0(\overline{\Omega})$ enthalten ist, ist Lösung des Variationsproblems

$$J(v) := \int_\Omega \left[\frac{1}{2} \sum_{i,k} a_{ik}\partial_i v \partial_k v + \frac{1}{2} a_0 v^2 - fv \right] dx \ \rightarrow \ min \ ! \tag{3.9}$$

unter allen Funktionen in $C^2(\Omega) \cap C^0(\overline{\Omega})$ mit Nullrandwerten.

Im Beweis folgt mit der Greenschen Formel und den Identifikationen

$$\begin{aligned} a(u,v) \ &:= \ \int_\Omega \left[\sum_{i,k} a_{ik}\partial_i u \partial_k v + a_0 uv \right] dx \\ (l,v) \ &:= \ \int_\Omega fv dx, \end{aligned} \tag{3.10}$$

dass $a(u,v) - (l,v) = 0$ ist, wenn $Lu = f$, also u eine klassische Lösung ist. Mit Satz 3.1 folgt die Minimaleigenschaft.

Der bekannte Satz von Lax-Milgram garantiert nun die Existenz von genau einer Lösung für das Variationsproblem (3.5). Damit ist jede Lösung des Variationsproblems, die in $C^2(\Omega) \cap C^0(\Omega)$ liegt, gleichzeitig auch eine klassische Lösung.

Führt man nun noch die so genannten Sobolev-Räume H_0^m ein, die nichts anderes sind als Vervollständigungen von $C_0^\infty(\Omega)$ bezüglich einer speziellen Norm $||\cdot||_m$, deren Definition auf dem Begriff der schwachen Ableitung in $L_2(\Omega)$ beruht, so erhält man eine Existenzaussage für so genannte schwache Lösungen des Problems (3.1) in einem Sobolev-Raum. Im hier vorliegenden Fall homogener Dirichlet-Randbedingungen ist dieser Raum $H_0^1(\Omega)$.

Satz 3.3 *Sei L ein gleichmäßig elliptischer Differentialoperator 2. Ordnung. Dann hat das Dirichlet-Problem 3.1 stets eine schwache Lösung in $H_0^1(\Omega)$. Diese ist das Minimum des Variationsproblems*

$$\frac{1}{2}a(u,v) - (f,v)_0 \;\rightarrow\; min\ ! \tag{3.11}$$

in $H_0^1(\Omega)$.

Hier ist $a(u,v)$ die gemäß (3.10) definierte „zum Problem gehörige“ Bilinearform,

$$(u,v)_0 := \int_\Omega u(x)v(x)dx \tag{3.12}$$

ist das Skalarprodukt in $H_0^1(\Omega)$. Bis zu diesem Punkt hat noch keinerlei Approximation oder Diskretisierung stattgefunden.

Durch die Einführung der Sobolevräume und des Begriffs der schwachen Lösung hat sich lediglich die Menge der behandelbaren Probleme vergrößert, da nun auch Lösungen mit schwächeren Differenzierbarkeitseigenschaften zugelassen sind.

Das hier Dargestellte ist lediglich eine „Kurzfassung“ der Theorie der Variationsformulierungen. Es sei vor allem darauf hingewiesen, dass wir uns nur mit homogenen Randbedingungen vom Dirichlet-Typ befassen. Diese werden häufig auch als „wesentliche“ Randbedingungen bezeichnet, da sie unmittelbar als Forderung in die Konstruktion der Lösungsräume eingegangen sind. Darüber hinaus existieren noch zahlreiche weitere Randbedingungen wie zum Beispiel die „natürlichen“ oder Neumann-Randbedingungen, die Bedingungen an die Ableitung der Lösung auf dem Rand darstellen, oder gemischte Randbedingungen, für die jeweils Charakterisierungssätze, variationelle Formulierungen und Existenz- und Eindeutigkeitssätze für Lösungen in geeigneten Räumen nachgewiesen werden konnten. Es sei dazu auf die einschlägige Literatur wie etwa [5] oder [23] verwiesen.

Ausgehend von der variationellen Formulierung lassen sich nun die Finite Elemente Verfahren aufbauen. Beim so genannten Ritz-Galerkin-Verfahren werden dazu sowohl der Raum der Lösungen u als auch der Raum der „Testfunktionen“ v auf einen endlichdimensionalen Unterraum $S_h \subset H_0^1(\Omega)$ eingeschränkt. Nach Satz 3.1 ist damit u_h eine Lösung in einem endlichdimensionalen Unterraum $S_h \subset H_0^1(\Omega)$, wenn

$$a(u_h, v) = (l, v) \text{ für alle } v \in S_h. \tag{3.13}$$

Wenn $\psi_1, \ldots, \psi_n$ eine Basis von S_h ist, dann ergibt sich aus (3.13)

$$a(u_h, \psi_i) = (l, \psi_i),\ i = 1, \ldots, n. \tag{3.14}$$

Stellt man nun noch die Lösung u_h als Linearkombination der Basisfunktionen von S_h dar, ergibt sich

$$u_h = \sum_{k=1}^{n} z_k \psi_k \tag{3.15}$$

$$\Rightarrow \sum_{k=1}^{n} a(\psi_k, \psi_i) z_k = (l, \psi_i),\ i = 1, \ldots, n \tag{3.16}$$

oder in Matrix-Vektor-Form

$$Az = b. \tag{3.17}$$

Je nach Wahl des Ansatzraums (Unterraum, in dem u_h gesucht wird, beziehungsweise dessen Basis) und des Testraums (also des Raumes der Funktionen für das zweite Argument von a) in Gleichung (3.13) ergeben sich unterschiedliche Verfahrenstypen. Bei den *Petrov-Galerkin-Verfahren* dürfen Ansatz- und Testraum als Unterräume gleicher Dimension des entsprechenden Sobolevraums unterschiedlich gewählt werden, was beispielsweise bei Problemen mit Singularitäten sinnvoll erscheint. Bei *Rayleigh-Ritz-Verfahren* wird ebenfalls ein Minimum in S_h bestimmt, in der Herleitung wird aber auf die basisfreie Darstellung wie in (3.13) verzichtet.

Mit dem Übergang auf die Suche eines Minimums in einem endlichdimensionalen Unterraum ist nun eine Näherung eingeführt. Zu den verschiedenen Verfahrenstypen existiert eine ausgereifte Approximationstheorie, die Aussagen über Konsistenz, Stabilität und Konvergenz der Verfahren trifft. Zentrale Bedeutung hat dabei das Lemma von Céa, das zeigt, dass die Genauigkeit der Näherungslösung wesentlich davon abhängt, wie gut u in S_h approximiert werden kann.

Außerdem können Aussagen darüber getroffen werden, wie gut Approximationen bestenfalls werden, wenn man als Basisfunktionen von S_h Polynome verwendet. Dabei und in der Praxis stellt sich heraus, dass es nur sehr begrenzt Sinn macht, zur Steigerung der Genauigkeit den Polynomgrad zu erhöhen, da dann Oszillationsprobleme auftreten. Statt dessen wird üblicherweise mehr in eine hinreichend feine

Zerlegung von Ω bei vergleichsweise niedrigem Polynomgrad der Basisfunktionen investiert.

Um in (3.17) eine dünn besetzte Matrix A zu erhalten, werden die Basisfunktionen in der Regel als stückweise Polynome mit kompaktem auf wenige Zellen eines Rechengitters beschränktem Träger erklärt. Die Räume, in denen man das Variationsproblem dann in der Praxis löst, heißen typischerweise *Finite-Elemente-Räume*. Sie entstehen durch Zerlegung von Ω in endlich viele Rechenzellen, auf denen Basisfunktionen jeweils Polynome sind. Durch diese Zerlegung entsteht ein so genanntes Rechengitter, dessen überschneidungsfreie Zellen ganz Ω abdecken. Die Zellen heißen dann *Elemente*, die *Kombination* aus Zelle und „darauf lebender Funktion“ nennt man *Finites Element*.

Die darauf basierenden Diskretisierungsmethoden lassen sich durch folgende Eigenschaften charakterisieren:

1. Art des Gitters: beispielsweise auf Basis von Dreiecken, Quadraten, Würfeln, Tetraedern, ...

2. Polynomgrad der Basisfunktionen als lokale Eigenschaft der Elemente.

3. Stetigkeits- und Differenzierbarkeitseigenschaften an den Elementgrenzen.

Man spricht von konformen Finiten Elementen, wenn die Funktionen in dem Sobolevraum enthalten sind, in dem das Variationsproblem gestellt ist, also in unserem Fall $S_h \subset H_0^1(\Omega)$. Zu den verschiedenen Gitterarten existieren jeweils formale Charakterisierungseigenschaften, zu nicht rechtwinkligen Zerlegungen von Ω können zusätzliche Bedingungen, zum Beispiel über die Innenwinkel bei Dreiecks- oder Tetraederelementen, an die Zerlegung gestellt werden. Da wir uns in dieser Arbeit jedoch ausschließlich mit bilinearen quadratischen Elementen beschäftigen, sei zu diesen Themen erneut auf die Literatur (zum Beispiel [5]) verwiesen.

Bilineare quadratische Elemente besitzen auf jeder der vier Ecken des Quadrats einen Freiheitsgrad, durch Vorgabe je eines Funktionswertes ist die auf dem Element lebende Funktion eindeutig bestimmt. Die Näherungslösung u ist dann auf jedem Element ein Polynom zweiten Grades, die Restriktion auf die Kanten des Quadrats ist jedoch eine lineare Funktion, die nur zwei Freiheitsgrade besitzt. Somit ist über die zwei an einer Kante anliegenden Freiheitsgrade die lineare Funktion auf der Kante eindeutig bestimmt und ein stetiger Anschluss an den Elementgrenzen gewährleistet. Insgesamt ist die Lösung also in $C^0(\Omega)$. Für diese Elemente kann man zeigen, dass die Approximationsgüte im Inneren des Rechengebiets von der Ordnung $O(h^2)$ ist, wenn h die feinste Gitterweite respektive Kantenlänge eines Elements ist, am Rand ist sie in unserem Fall $O(h)$.

3.2 Nodale Basis und hierarchisches Erzeugendensystem

In diesem Abschnitt werden die Begriffe nodale Basis, hierarchische Basis und hierarchisches Erzeugendensystem erläutert. Dabei wird bewusst auf detaillierte Beschreibungen der Eigenschaften und der Vor- und Nachteile verzichtet, da diese in vielen Arbeiten unserer Gruppe nachgelesen werden können. Die nodale Basis wird in dieser Arbeit lediglich als Ausgangspunkt eines natürlichen Weges zu Multilevelansätzen für Finite-Elemente-Verfahren erwähnt. Einige weiter gehende Informationen zu den unterschiedlichen Basen finden sich zum Beispiel in [2] in Kapitel 2.

Wie in 3.1 beschrieben muss für den Übergang von Gleichung (3.13) zum diskreten Gleichungssystem (3.16) eine geeignete Basis des Teilraums S_h gewählt werden. In unserem Fall des bilinearen quadratischen Elements sind also Funktionen gesucht, die stückweise bilinear auf jedem Element und linear auf den Rändern beziehungsweise Kanten der Elemente sind, das heißt stückweise die Gestalt

$$u_h(x,y) = axy + bx + cy + d, \quad \text{mit } a,b,c,d \text{ konstant} \tag{3.18}$$

besitzen. Als Basisfunktionen wählt man nun stückweise definierte Funktionen mit Funktionswert 1 am Mittelpunkt ihres Trägers und 0 an den Kanten. Die Träger sind dabei jeweils Quadrate. Diese Funktionen werden wegen ihres Aussehens auch Pagodenfunktionen genannt, ein Beispiel findet sich in Abbildung 3.1, die dort dargestellte Funktion ist auf dem Quadrat $[0,1]^2$ stückweise definiert als

$$u_1(x,y) := \begin{cases} 4xy & \text{für} \quad 0 \le x \le \frac{1}{2} \text{ und } 0 \le y \le \frac{1}{2} \\ -4xy + 4y & \text{für} \quad \frac{1}{2} < x \le 1 \text{ und } 0 \le y \le \frac{1}{2} \\ -4xy + 4x & \text{für} \quad 0 \le x \le \frac{1}{2} \text{ und } \frac{1}{2} < y \le 1 \\ 4xy - 4x - 4y + 4 & \text{für} \quad \frac{1}{2} < x \le 1 \text{ und } \frac{1}{2} < y \le 1 \end{cases} \tag{3.19}$$

3.2.1 Nodale Basis

Obige Funktionen lassen sich auf jedem Quadrat erzeugen. Wählt man für eine feste Gitterweite h alle Funktionen, die als Träger ein Quadrat mit Seitenlänge $2h$ haben, dessen Eckpunkte die diskreten Punkte eines in beide Raumrichtungen äquidistanten Gitters sind, so erhält man die so genannte nodale Basis

$$\{\Psi_i; i = 1, \ldots, n\}. \tag{3.20}$$

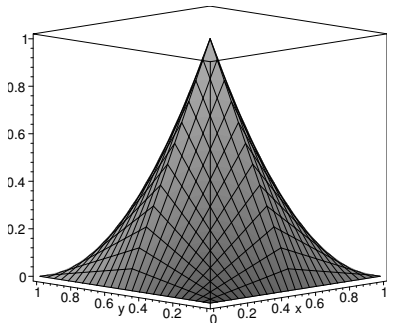

Abbildung 3.1: Pagodenfunktion für das Einheitsquadrat.

Wegen

$$\Psi_i(x_i) = 1 \quad \text{und} \quad \Psi_i(x_j) = 0 \quad \text{für alle} \quad i \neq j \tag{3.21}$$

gilt

$$u_h(x_i) = z_i. \tag{3.22}$$

Die Funktionswerte der diskreten Lösung u_h lassen sich somit unmittelbar aus den Koeffizienten der Basisdarstellung (3.15) ablesen. Berechnet man nun für eine Basisfunktion Ψ_i die Werte der Integrale durch die in Gleichung (3.10) definierten Bilinearform mit allen anderen Basisfunktionen, also die Matrixeinträge $A_{ik}, k = 1, \ldots, n$ der Systemmatrix aus (3.17), so ergibt sich aus den eng beschränkten Trägern der Basisfunktionen, dass jede Basisfunktion nur von ihren unmittelbaren Nachbarn beeinflusst wird, die Integrale mit allen anderen Basisfunktionen sind 0. Somit lässt sich der Einfluss der Nachbarn auf eine Basisfunktion im Falle der Poisson-Gleichung durch den bekannten so genannten 9-Punkte-FEM-Stern darstellen:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}_* \tag{3.23}$$

Die in Gleichung (3.17) verwendete Matrix A ist also dünn besetzt und hat Bandstruktur. Trotzdem wird in der Praxis diese Matrix nie komplett aufgestellt und gelöst. Wie bereits in [5] ab Seite 90 dargestellt führt obige *knoten-orientierte* Betrachtung der Freiheitsgrade und ihrer Einflussbereiche zu unnötigen Mehrfachauswertungen. Daher wird stattdessen in einer *element-orientierte* Betrachtungsweise lediglich der additive Beitrag eines Elements zur Steifigkeitsmatrix A ermittelt. Diese Vorgehensweise wird in den Abschnitten 3.4 und 3.5 für unsere Beispiele noch genauer erläutert. Somit ist pro Element eine $s \times s$-Untermatrix zu berechnen, wenn s die Anzahl der Knoten beziehungsweise Freiheitsgrade je Element ist. Da wir nun die Elemente mit den Gitterzellen identifizieren können, ist sofort klar, dass eine element-orientierte Sichtweise methodisch sehr gut zur Beschreibung eines Weges durch die Zellen

des Gebietes mittels einer raumfüllenden Kurve passt. Zur Berechnung einer Finite-Elemente-Lösung eines gegebenen Problems ist also in jeder Löser-Iteration das Gebiet entlang der raumfüllenden Kurve zu durchlaufen und auf jeder Zelle ein additiver Beitrag zur Lösung von Gleichung (3.17) zu berechnen. Wie später noch dargestellt wird, erfolgt die Berechnung der Lösung dieser Gleichung durch iterative Verfahren, die sich wiederum sehr natürlich in das oben dargestellte Gefüge von Zell- und Element-Orientierung in der Geometriebeschreibung und im Ansatz zur Berechnung der Lösung einer Partiellen Differenzialgleichung einfügen.

3.2.2 Hierarchische Erzeugendensysteme

Moderne Löser für partielle Differenzialgleichungen müssen Multi-Level-Ansätze wie Mehrgitterverfahren unterstützen, um bezüglich Aufwand und Rechenzeit konkurrenzfähig sein zu können. Voraussetzung für Multilevelverfahren sind sicherlich geeignete Ansatzräume je Level, die sich leicht erzeugen lassen und den Informationstransport zwischen den Leveln auf einfache Art und Weise ermöglichen. Kombiniert man die Gewinnung von Basen wie oben dargestellt mit den Ideen der Raumpartitionierung aus Kapitel 2, so gelangt man zu hierarchischen Erzeugendensystemen, wie sie im Folgenden dargestellt werden.

Multilevelverfahren zeichnen sich dadurch aus, dass Informationen über die Lösung, die auf verschieden fein aufgelösten Gittern gewonnen werden, geschickt kombiniert werden, wodurch die Konvergenzgeschwindigkeit deutlich erhöht wird und sogar unabhängig von der Auflösung des feinsten Gitters wird.[1]

Denkt man zurück an die rekursive Raumpartitionierung, wie sie in Kapitel 2 vorgeschlagen ist, so ergibt sich die Wahl der Gitter unterschiedlicher Feinheit für den Multilevelansatz sehr natürlich aus der Abfolge der Zellen im Partitionierungsprozess. Jede Ebene des durch Raumpartitionierung entlang der raumfüllenden Kurve entstandenen Spacetrees ist eine vollständige Unterteilung des Grundgebietes Ω in Quadrate und ist somit geeignet als Zerlegung des Gebietes für einen Finite-Elemente-Ansatz. Die zugehörigen Basisfunktionen lassen sich leicht wie oben dargestellt erzeugen. Damit kann für jedes Level des Baumes ein Satz von Basisfunktionen angegeben werden, auf denen eine Finite-Element-Lösung berechnet werden kann.

Die Gesamtheit dieser Funktionen über alle Level ist nun aber keine Basis mehr, da es auf Punkten, die in mehr als einem Level vorkommen, auch mehr als eine

[1] Hierbei wird implizit vorausgesetzt, dass die Lösung mit Hilfe von iterativen Verfahren berechnet wird und die Konvergenzgeschwindigkeit durch die Anzahl von Iterationen bis zum Erreichen einer gegebenen Genauigkeit der Näherungslösung charakterisiert ist. Für einfache iterative Verfahren gilt, dass bei gleicher Genauigkeitsschranke die Anzahl der Iterationen mit zunehmender Feinheit des Gitters (unter Umständen dramatisch) steigt, weil das Gleichungssystem $Az = b$ im Falle der nodalen Basis meist schlecht konditioniert ist.

zugehörige Basisfunktion gibt. Das so erzielte System von Ansatzfunktionen stellt lediglich noch ein hierarchisches Erzeugendensystem dar, das entsprechende Gleichungssystem $Az = b$ ist durch die lineare Abhängigkeit der Ansatz-Funktionen nur semidefinit. Wir bezeichnen das hierarchische Erzeugendensystem mit E.

Eine hierarchische Basis B, die zu einem definiten System und einer eindeutigen Darstellung von Funktionen als Linearkombinationen der Basisfunktionen führt, ist jedoch als Teilmenge in der Menge der Funktionen des hierarchischen Erzeugendensystems E enthalten. Griebel zeigt dazu in [11] in Kapitel 2:

1. Es existiert eine Transformation, die aus einer gegebenen Darstellung einer Funktion u_E in E eine eindeutige Darstellung u_B in B erzeugt.
2. Zwei verschiedene Darstellungen $u_{E,1}$ und $u_{E,2}$, die das selbe Bild unter der Transformation haben, also die gleiche Darstellung bezüglich B, unterscheiden sich um einen Vektor, der im Kern der Transformation enthalten ist.
3. Mit einem verallgemeinerten Konditionsbegriff lässt sich zeigen, dass die semidefiniten Systeme immer lösbar sind (weil entsprechende Forderungen an den Rang immer erfüllt sind). Es existieren jeweils viele Lösungen, für die aber gezeigt werden kann, dass ihre Überführung in eine Darstellung unter B immer auf eine eindeutige Lösung führt.

Dies sind Voraussetzungen, unter denen eine weitere Betrachtung der aus hierarchischen Erzeugendensystemen entstanden semidefiniten Systeme sinnvoll erscheint.

3.3 Additive Mehrgitterverfahren

Mehrgitterverfahren sind inzwischen weitläufig bekannt als hochperformante numerische Verfahren zur Lösung von Gleichungssystemen. Die Basisidee von Mehrgitterverfahren ergibt sich (zumindest für die geometrischen Verfahren) aus der Beobachtung, dass eine Frequenzanalyse der Fehler, die bei der Berechnung von Näherungslösungen entstehen, zeigt, dass die Größenordnung dieser Fehler oft stark von ihrer Frequenz abhängen. Ein iterativer Löser wirkt gewissermaßen als Glätter für den Fehler, nach Anwendung einiger Gauß-Seidel- oder Jacobi-Schritte ist der Fehler zwar im Betrag nicht notwendig deutlich reduziert, der Verlauf ist jedoch glatter geworden. Genauere Analysen zeigen nun, dass es bei gegebener Gitterweite h eine Bereich von Fehlerfrequenzen gibt, für die ein Glätter sehr gut funktioniert, also sehr schnell (mit wenigen Iteration) die entsprechenden Fehlerfrequenzen dämpft. Dieser optimale Bereich liegt im Bereich der Wellenlänge, die die gleiche Ordnung hat wie die Gitterweite. Auf Basis dieser Erkenntnisse ist nun das Vorgehen für ein Zweigitterverfahren nahe liegend:

1. Auf einem feinen Gitter wird eine Näherungslösung u_h durch einige Glättungsschritte berechnet. Diese Näherungslösung enthält kaum noch hochfrequente Fehleranteile.

2. Durch Restriktion auf eine geeignete Teilmenge von Gitterpunkten (Grobgitter der Weite H) gewinnt man ein Gleichungssystem auf dem groben Gitter, mit dessen Hilfe niederfrequente Fehler reduziert werden sollen.

3. Die Grobgittergleichung wird gelöst. Der Aufwand hierfür ist wegen der reduzierten Anzahl an Freiheitsgraden sehr viel geringer als der Aufwand zur Lösung des Feingittergleichungssystems.

4. Die Näherungslösung u_H auf dem groben Gitter wird durch Interpolation auf das feine Gitter transportiert.

5. Neu entstandene Feingitterfehler werden durch einige Nachglättungsschritte korrigiert.

Setzt man diesen Vorgang rekursiv auf noch gröbere Gitter fort, indem man zum Beispiel von Gitter zu Gitter je Raumrichtung die Anzahl der Freiheitsgrade halbiert, kommt man zu dem bekannten V-Zyklus. Zur Theorie der Mehrgitterverfahren ist bereits sehr viel geschrieben worden, von entscheidender Bedeutung sind folgende Punkte:

- Die Lösung muss auf jedem Gitter repräsentierbar sein. Insbesondere die physikalischen Phänomene der zugrunde liegenden Gleichungen müssen auf den groben Gittern noch wiedergegeben (aufgelöst) werden können.

- Man kann zeigen, dass zwar der Aufwand an Punkten durch die Hinzunahme der Grobgitter wächst, jedoch nur um einen konstanten Faktor, da die Anzahl der Freiheitsgrade der Level mit einer geometrischen Reihe abnimmt. Gleiches gilt für den Rechenaufwand inklusive der Restriktions- und Interpolationsoperatoren.

- Der entscheidende Vorteil ergibt sich in der Anzahl von Iterationsschritten, die durchgeführt werden müssen, bis die Lösung eine gegebene Fehlerschranke unterschreitet. Für viele Verfahren und Beispiele kann man zeigen und nachmessen, dass der Aufwand an Iterationsschritten konstant ist, egal welche Auflösung das feinste Gitter hat. Dies ist ein gravierender Unterschied zu Eingitterverfahren, deren Iterationszahl mit feinerer Gitterauflösung wächst.

- Als Interpolationsoperatoren werden häufig lineare oder bilineare Operatoren verwendet, die Restriktion ist entweder ein Kopieren der Feingitterwerte (Injektion) oder meist die Transponierte des Interpolationsoperators.

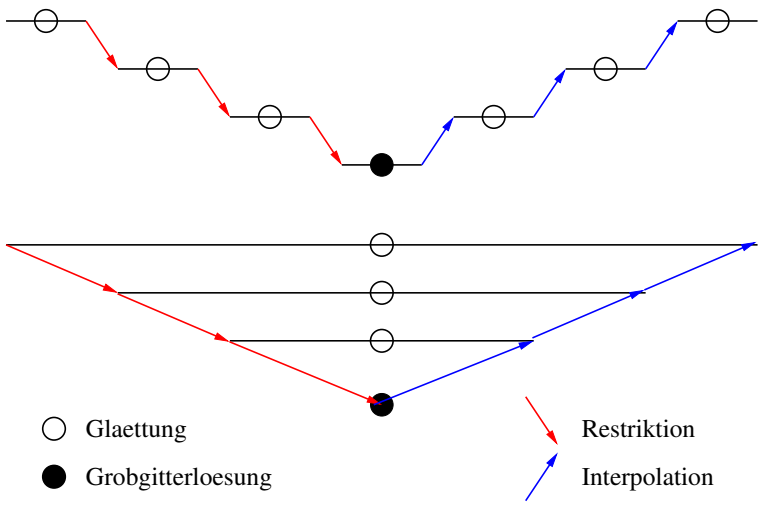

Abbildung 3.2: Ablaufstruktur von multiplikativem (oben) und additivem (unten) Mehrgitterverfahren.

- Im Zusammenhang mit Finite-Elemente-Verfahren ist durch die Auswahl des Elementtyps bekannt, welchen Typs die Basisfunktionen auf dem Rand und im Inneren eines Elements sind. Es daher naheliegend und ratsam, die Interpolationen und Restriktionen so zu wählen, dass diese den Eigenschaften der Basisfunktionen entsprechen. Im unserem Kontext der quadratischen Elemente mit bilinearen Basisfunktionen, deren Randeinschränkungen linear sind, sollte also eine Interpolation gewählt werden, die die Grobgitterwerte eines Elements auf dessen Rand linear und im Inneren bilinear auf die Feingitterpunkte transportiert. Entsprechendes gilt für den Restriktionsoperator.

In der algorithmischen Umsetzung dieser Grundidee gibt es mindestens zwei Klassen, die unterschieden werden müssen. Es sind dies multiplikative und additive Mehrgitterverfahren. In [3] werden die Unterschiede und Gemeinsamkeiten ausführlich dargestellt. Der entscheidende Unterschied ergibt sich dadurch, dass im Falle des *multiplikativen* Mehrgitterverfahrens die Gitter nacheinander besucht werden, dort geglättet wird und die Information (oder die Korrektur) dann mit Interpolation (von grob zu fein) oder Restriktion (von fein zu grob) auf das nächste Gitter transportiert wird. Klassische V-Zyklen sind vom multiplikativen Typ. Bei *additiven* Verfahren kann die Glättung auf allen Gittern parallel ablaufen, Interpolation und Restriktion laufen jedoch weiterhin sequenziell ab. Abbildung 3.2 zeigt dieses unterschiedliche Verhalten.

Nach [3] werden additive Mehrgitterverfahren typischerweise als Vorkonditionierer und multiplikative Verfahren eher als Löser eingesetzt. Theoretisch ist nachweisbar, dass beide Verfahren je Zyklus auf einem sequenziellen Computer die gleiche Arbeit zu verrichten haben. Bastian, Hackbusch und Wittum [3] weisen außerdem

nach, dass sich mit multiplikativen Mehrgitterverfahren bessere Laufzeiten erzielen lassen als mit additiven Verfahren. In dieser Arbeit wird ein additives Mehrgitterverfahren programmiert, bei dem nur auf dem feinsten Level geglättet wird. In Kapitel 5 wird begründet und motiviert, warum in unserem Fall trotz der Performancenachteile zunächst ein additives Verfahren sinnvoll erscheint. Wichtig ist hier noch zu erwähnen, dass Griebel in [11] vor allem zeigt, dass iterative Methoden wie das Gauß-Seidel- oder das Jacobi-Verfahren geeignet formuliert auf den hierarchischen Erzeugendensystemen zu Mehrgittermethoden führen, für die die übliche qualitative Konvergenztheorie wie in [27] angewendet werden kann. Die Formulierung der Mehrgitter-Algorithmen in Kapitel 5 beruht auf den Ideen von Griebel [11].

3.4 Die Poisson-Gleichung

In diesem und dem folgenden Abschnitt werden nun die beiden Modellprobleme dargestellt, die für die numerischen Experimente verwendet wurden. Dabei wird vor allem auf die Realisierung der element-orientierten Sichtweise und eine Lokalisierung der Finite-Element-Sterne eingegangen.

Die erste Gleichung, die verwendet wurde, ist die Poisson-Gleichung, die als einfaches Beispiel für eine elliptische partielle Differenzialgleichung häufig herangezogen wird. Gesucht ist eine Lösung der Gleichung

$$\begin{aligned} \triangle u(\mathbf{x}) &= -2\pi^2 \sin(\pi x)\sin(\pi y) && (3.24)\\ &\quad \text{für alle} \quad \mathbf{x} = (x,y)^T \in \Omega = [0,1]\times[0,1] && (3.25)\\ u(\mathbf{x}) &= 0 \quad \text{für alle} \quad \mathbf{x} \in \partial\Omega. && (3.26) \end{aligned}$$

Die exakte Lösung ist gegeben durch $u(\mathbf{x}) = \sin(\pi x)\cdot\sin(\pi y)$. Weiterhin wurde diese Gleichung auf Kreisgebieten wie in Abbildung 3.3 gelöst. Dafür ist zu formulieren:

$$\begin{aligned} \triangle u(\mathbf{x}) &= -2\pi^2 \sin(\pi x)\sin(\pi y) && (3.27)\\ &\quad \text{für alle} \quad \mathbf{x} = (x,y)^T \in \Omega = \{\mathbf{x}\in[0,1]\times[0,1] : ||\mathbf{x}|| \le r\} && (3.28)\\ u(\mathbf{x}) &= 0 \quad \text{für alle} \quad \mathbf{x} \in \partial\Omega. && (3.29) \end{aligned}$$

mit einem positiven Radius $r \le 1$. Diese Kreisgebiete zeigen vor allem die Möglichkeit auf, kompliziertere Geometrien in das Einheitsquadrat einzubetten. In Kapitel 5 wird erklärt, warum dies in unserem Kontext ein sinnvolles Vorgehen ist.

Die Diskretisierung dieser Gleichung auf einer nodalen Basis eines Finite-Elemente-

Raums mit quadratischen Elementen ergibt den bekannten 9-Punkte-Stern

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}_*, \tag{3.30}$$

wobei vorausgesetzt ist, dass auf jeder Ecke einer Zelle ein Freiheitsgrad existiert. Dieser Stern „lebt“ auf vier benachbarten Zellen, die das gesamte Einflußgebiet des in der Mitte liegenden Freiheitsgrades darstellen (Abbildung 3.4). Eine Verteilung des Einflusses auf jede der vier Zellen ist ebenfalls in Abbildung 3.4 zu sehen. Jede Zelle rechnet gewissermaßen „ihren“ Anteil an der Korrektur des Freiheitsgrades aus und addiert ihn auf. Nachdem alle vier Zellen ihren Beitrag geleistet haben, ist ein Residuum berechnet, das mit einer direkten Auswertung des Sterns unter Zu-Hilfe-Nahme aller Daten der umliegenden Zellen übereinstimmt.

Direkte Auswertung:

$$-8 \cdot u_* + \sum_{i=1}^{8} u_i. \tag{3.31}$$

Zellorientierte Auswertung:

$$\text{Zelle I:} \quad -2 \cdot u_* \quad + \quad u_1 + \frac{1}{2} \cdot u_2 + \frac{1}{2} \cdot u_8 \tag{3.32}$$

$$\text{Zelle II:} \quad -2 \cdot u_* \quad + \quad \frac{1}{2} \cdot u_2 + u_3 + + \frac{1}{2} \cdot u_4 \tag{3.33}$$

$$\text{Zelle III:} \quad -2 \cdot u_* \quad + \quad \frac{1}{2} \cdot u_4 + u_5 + + \frac{1}{2} \cdot u_6 \tag{3.34}$$

$$\text{Zelle IV:} \quad -2 \cdot u_* \quad + \quad \frac{1}{2} \cdot u_6 + u_7 + + \frac{1}{2} \cdot u_8 \tag{3.35}$$

$$\text{Summe:} \quad -8 \cdot u_* \quad + \quad u_1 + u_2 + u_3 + \ldots + u_8 \tag{3.36}$$

Die detaillierte algorithmische Realisierung (Relaxationsparameter, Residuum, rechte Seite, usw.) findet sich in Kapitel 5. Notwendig ist eine zell-orientierte Bereitstellung der Daten auf den Ecken einer Zelle. Die Umsetzung dieser Forderung ist ebenfalls in Kapitel 5 dargestellt und wird in Kapitel 4 motiviert.

3.5 Die Stokes-Gleichung

Die Stokes-Gleichung ist eine erste Möglichkeit, Strömungen zu beschreiben. Im Unterschied zu den Navier-Stokes-Gleichungen, die als vollständige Beschreibung von Strömungen angesehen werden, fehlen hier die Terme, die die Konvektion und damit die innere Reibung eines Fluids beschreiben. Für uns ist sie aber interessant, weil sie bezüglich Speicherkomplexität und notwendigen Daten pro Zelle bereits

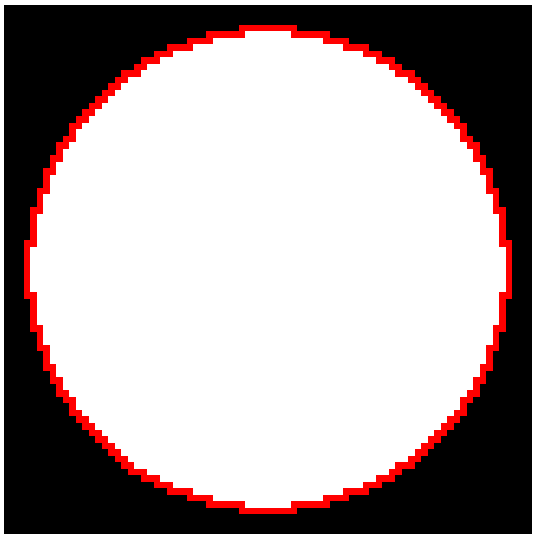

Abbildung 3.3: Kreisgebiet eingebettet in das Einheitsquadrat. Im weißen Bereich ist die Gleichung zu lösen, im schwarzen Bereich ist Nichts zu tun. Die Trennlinie ist rot markiert, was vom Programm als Dirichlet-Rand interpretiert wird.

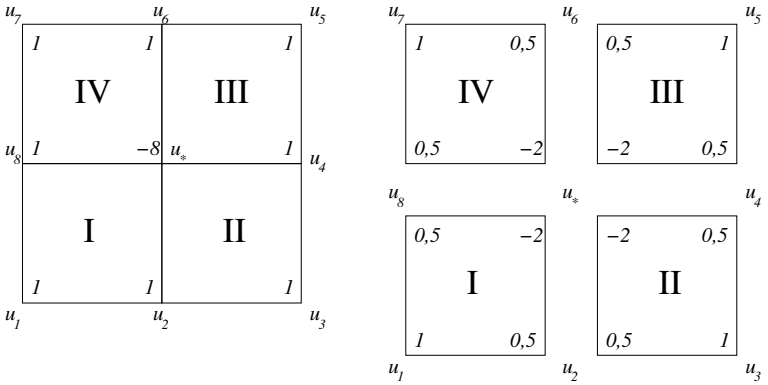

Abbildung 3.4: Element-orientierte Verteilung des 9-Punkte-Sterns.

die gleichen Anforderungen stellt wie die Navier-Stokes-Gleichungen, algorithmisch und numerisch jedoch eine relativ leicht durchzuführende Erweiterung des Lösers für die Poisson-Gleichung ist. Gesucht ist ein Lösung $\mathbf{u} = (u, v)^T : \Omega \mapsto \Omega$ der Gleichung

$$\begin{aligned} \triangle \mathbf{u} - \text{grad} p &= 0 && (3.37)\\ \text{div}\mathbf{u} &= 0 && (3.38)\\ u(x,y) = 1,\ v(x,y) &= 0 \quad \text{für} \quad y = 1 && (3.39)\\ u(x,y) = v(x,y) &= 0 \quad \text{für} \quad (x,y)^T \in \partial\Omega \backslash \{(x,y)^T \in \partial\Omega : y = 1\} && (3.40) \end{aligned}$$

wobei Ω das Einheitsquadrat oder ein konvexes Teilgebiet des Einheitsquadrats ist, das die Kante von $(0,1)$ bis $(1,1)$ vollständig enthält. Der Druck p ist dabei nur bis auf eine Konstante bestimmt.

Es muss jeweils eine Poisson-Gleichung für u und v gelöst werden, zusätzlich werden die Residuen von u und v mit einer „Druckkorrektur“ versehen. Zu diesem Zweck wird zusätzlich auf jeder Zelle eine weitere Variable eingeführt, die den Druckwert der Zelle enthält. Hierbei ist zu beachten, dass der Druck eine zellbasierte Information darstellt, wohingegen die Werte für u beziehungsweise v zur Lösung der Poisson-Gleichungen knotenbasierte Werte sind. In Kapitel 5 wird aufgezeigt, wie die Bereitstellung der zellbasierten Information auf effiziente Weise erfolgen kann und wie die Druckkorrektur realisiert ist.

4 Keller und Datenströme

Dieses Kapitel behandelt den eigentlichen Kern der Arbeit, die Konstruktion von cache-effizienten Datenstrukturen zur Realisierung von Baumdurchläufen entlang der Peano-Kurve mit Zugriff auf die den Zellecken zugeordneten Daten. Dabei ergibt sich aus der Reihenfolge des Datenzugriffs auf sehr natürliche Weise die Verwendung der einfachen und seit langem bekannten Keller. Es wird exemplarisch dargelegt, wie man mit Hilfe der raumfüllenden Kurve deterministische „Regelsätze“ zum Datentransfer von und zu Kellern realisieren kann, was entscheidend zu der sehr hohen Performance der Programme in der Speicherhierarchie moderner Prozessoren führt.

4.1 Keller als Datenspeicher und ihre Zugriffsmechanismen

Keller oder Kellerautomaten sind ein seit langem bekanntes und bestechend einfaches Konzept. Ende der 50er Jahre des vorigen Jahrhunderts von K. Samelson und F. L. Bauer [21] zur Syntaxanalyse von Programmiersprachen entwickelt, stellt der so genannte Kellerautomat zunächst ein theoretisches Maschinenkonzept dar. Solche Maschinenkonzepte werden in Verbindung mit Sprachen in der theoretischen Informatik in den Bereichen der Berechenbarkeits- und Entscheidbarkeitstheorie und der Theorie der formalen Sprachen eingesetzt. Die dabei interessanten Fragestellungen sind die nach dem Wortproblem („Man gebe für eine Grammatik G einen (terminierenden) Algorithmus an, der entscheidet, ob ein gegebenes Wort dem Sprachschatz der durch G erzeugten Sprache angehört oder nicht“) und dem Zerteilungsproblem („Man gebe einen Algorithmus an, der für jedes Wort des Sprachschatzes von G einen Reduktionsweg zum Axiom der Sprache liefert.“)[4]. Die mechanischen Realisierungen der eventuell angebbaren Algorithmen für das Wort- und das Zerteilungsproblem einer Sprache sind entsprechend geeigneter Charakterisierungen (zum Beispiel deterministisch oder nicht-deterministisch) gerade die theoretischen Maschinen wie die Turing-Maschine oder der Kellerautomat. Ein Kellerautomat ist ein um den Keller erweiterter endlicher Automat und ist somit ein 8-Tupel $A = (I, O, Q, \delta, q_0, F, \Gamma, Z_0)$ [8].

Dabei ist

- I das Eingabealphabet,
- O das Ausgabealphabet,
- Q die Zustandsmenge,
- δ die Zustandsübergangsrelation,
- q_0 der Anfangszustand,
- F die Menge der Endzustände,
- Γ das Kelleralphabet und
- Z_0 das Grundsymbol des Kellers mit $Z_0 \in \Gamma$.

Ein (nicht-deterministischer) Kellerautomat erkennt eine kontextfreie Sprache, die Arbeitsweise eines Kellerautomaten kann durch eine kontextfreie Grammatik beschrieben werden. Solche Zusammenhänge zwischen Automaten und formalen Sprachen sind seit einiger Zeit bekannt. Da sie für den Kern dieser Arbeit jedoch nicht von Bedeutung sind, sei auf die Literatur, zum Beispiel [12], verwiesen.

Wir nutzen vom Konzept des Kellerautomaten lediglich die Eigenschaften eines Datenspeichers. In unserem Sinne ist ein Keller ein Stapel von Datenobjekten, für den es nur zwei Zugriffsmöglichkeiten gibt:

- `pop`: Hole das oberste Element vom Stapel.
- `push`: Lege ein Element oben auf den Stapel.

Auf den ersten Blick erscheint das als eine nicht besonders intelligente Wahl zur Speicherung von Daten für numerische Berechnungen, weil jede Flexibilität zum Datenzugriff fehlt: Man kann lesend und schreibend nur das oberste Element eines Kellers verändern. Gelingt es jedoch, die Daten passend zu einer bestimmten Durchlaufreihenfolge geeignet in wenigen Kellern anzuordnen, so dass also die zu einem bestimmten Zeitpunkt benötigten Daten tatsächlich die obersten Elemente der verwendeten Keller sind, so kann man eine hohe Performance in der Speicherhierarchie moderner Prozessoren erhoffen. Diese Hoffnung ergibt sich aus der Tatsache, dass bei korrekter Implementierung der Keller ihre Daten streng sequenziell abgearbeitet werden. Liegen nun die Nutzdaten ebenfalls sequenziell im Hauptspeicher des Rechners, was durch die geeignete Wahl eines Sprachmittels für die Realisierung wie etwa Felddatentypen erzielt werden kann, werden sie mit hoher Wahrscheinlichkeit immer „passend“ in die Caches des Prozessors geladen. Dies sollte zu einer niedrigen Zahl an cache misses (besonders im Level-2-Cache) führen, was einen Engpass,

der die Effizienz der meisten modernen Löser für partielle Differenzialgleichungen erheblich beeinträchtigt, per Konzept gar nicht erst entstehen lässt.

In Kapitel 6 werden diese hier nur angedeuteten Zusammenhänge genauer erläutert und mit Beispielrechnungen bestätigt. Das in diesem Abschnitt dargestellte soll lediglich die Verwendung der Keller, die in ihrer Implementierung häufig auch als Stacks bezeichnet werden, als Datenstruktur motivieren. Entscheidend für eine erfolgreiche Realisierung erscheinen – neben der Notwendigkeit, dass die in Kapitel 1 dargestellten Anforderungen an moderne Löser für große Gleichungssysteme oder Differenzialgleichungen erfüllt werden können – folgende Forderungen:

- **Begrenzte Zahl der Keller**. In der Kellerrealisierung darf die Anzahl der benötigten Keller nur von der Anzahl der (Raum-) Dimensionen des gestellten Problems abhängen, nicht aber von der Anzahl der Zellen oder Freiheitsgrade oder der Tiefe des Geometriebaums. Diese Abhängigkeit würde zu einer nachteiligen Komplexität der Programme führen, weil dann die Zugriffsroutinen auf die Stacks tiefenabhängig programmiert werden müssten. Dies ist bei der angestrebten rekursiven Programmierung des Lösers äußerst unbefriedigend und würde darüber hinaus eine auflösungsabhängige Cache-Effizienz bedingen.

- **Geeigneter Datentyp**. Der Datentyp für die Realisierung der Keller muss dem Compiler ermöglichen, die im Keller enthaltenen Daten linear im realen Hauptspeicher abzulegen. Für Hochsprachen wie etwa `C` scheint ausschließlich der Typ `array` geeignet. Eine Realisierung über durch Zeigergeflechte aufgebaute Listen brächte zwar bezüglich der Höhe der Keller die Möglichkeit einer Dynamisierung, eine lineare Anordnung der Daten im Arbeitsspeicher ist jedoch in diesem Fall sehr unwahrscheinlich.

- **Kapselung der Keller**. Die konkrete Realisierung der Keller sollte vor den Löserroutinen verborgen werden. Diese kennen lediglich einen Datentyp, der ein Kellerelement repräsentiert, sowie die Routinen `push` und `pop`. Dadurch hat man die Möglichkeit für unterschiedliche Zwecke unterschiedliche Keller zu implementieren. Wie in Kapitel 5 aufgezeigt wird, ist es beispielsweise sehr hilfreich, die Keller für die Repräsentation der Lösung vor und nach einer Iteration anders beziehungsweise mit weniger Daten pro Kellerelement zu realisieren als die Keller für lokale Zwischenspeicherung von Daten innerhalb einer Iteration.

Nach diesen Vorüberlegungen kommen wir im nächsten Abschnitt zu einem konkreten Konzept für die Verwendung der Keller im Kontext unserer Geometrie- und Datenrepräsentation.

4.2 Färbung der Gitterpunkte – Keller für eine nodale Basis

Bevor wir darstellen, welche und wie viele Keller in unseren Algorithmen benutzt werden, wollen wir motivieren, warum die Wahl dieser Speicherstruktur in unserem Fall naheliegend und natürlich ist. Dazu setzen wir voraus, dass ein Algorithmus zur iterativen Lösung der aus einem FE-Ansatz entstehenden Gleichungssysteme alle Zellen des gesamten Rechengebiets oder zumindest des eingebetteten Gebiets, auf denen die zugrunde liegende Differenzialgleichung erfüllt sein muss, pro Iteration mindestens einmal betreten muss. Weiter setzen wir voraus, dass wie in Kapitel 2 beschrieben die Abarbeitung der Zellen eines Gebietes innerhalb einer Iteration entlang einer einbeschriebenen diskreten raumfüllenden Kurve (in unserem Fall der Peano-Kurve) erfolgt.

Zur Veranschaulichung betrachte man zunächst den in Abbildung 4.1 dargestellten Gitterausschnitt mit der zugehörigen Peano-Kurve. Da in diesem Abschnitt zunächst nur nodale Daten betrachtet werden, ist jedem Gitterknoten genau eine Dateneinheit (für die Poisson-Gleichung beispielsweise bestehend aus Wert der rechten Seite, Wert der Unbekannten und Wert des Residuums) zugeordnet. Diese Daten gehören jeweils zu den Blättern des Spacetrees, werden also bei Betreten einer nicht mehr weiter verfeinerten Zelle des Gitters benötigt. Betrachtet man nun die Punkte 1 bis 9 auf der horizontalen Linie in der Mitte von Abbildung 4.1, so werden sie entlang der Kurve zuerst in aufsteigender und sofort danach in absteigender Reihenfolge von den Zellen benötigt. Zumindest für die Punkte 2 bis 9 kann man sogar festhalten, dass sie nach Abarbeitung der dargestellten Zellen innerhalb einer Iteration sicher nicht mehr benötigt werden, egal in welches Gebiet diese Zellen eingebettet sind. Es ist sofort ersichtlich, dass es geschickt wäre, die Punkte auf dieser Linie in einem Stack (zwischen) zu speichern, da sie streng linear aufsteigend und absteigend bearbeitet werden.

Betrachtet man die ersten drei Iterierten der Peano-Kurve (Abbildung 4.2), erkennt man, dass im rekursiven Verfeinerungsprozess viele solche Linien entstehen, auf denen Punkte in der oben beschriebenen Weise bearbeitet werden, und dass einmal entstandene Linien von Rekursion zu Rekursion immer mehr Punkte beinhalten, aber in ihrer Struktur nicht verändert werden. Ebenso kann man sich leicht überlegen, dass auch bei adaptiver Verfeinerung dieser Effekt nicht zerstört werden kann. Auf einer Linie werden höchstens neue Punkte eingefügt, aber keine vorhandenen verändert oder zerstört. Dies liegt daran, dass die Peano-Kurve überschneidungsfrei und stetig ist und daher die Durchlaufreihenfolge entlang einer Linie durch die Verfeinerung nicht geändert werden kann und außerdem die Kurve lokal verfeinert wird, das heißt keine Kreuzung mit einer Grobgitterlinie vorkommt, wo vor der Verfeinerung auch keine aufgetreten ist.

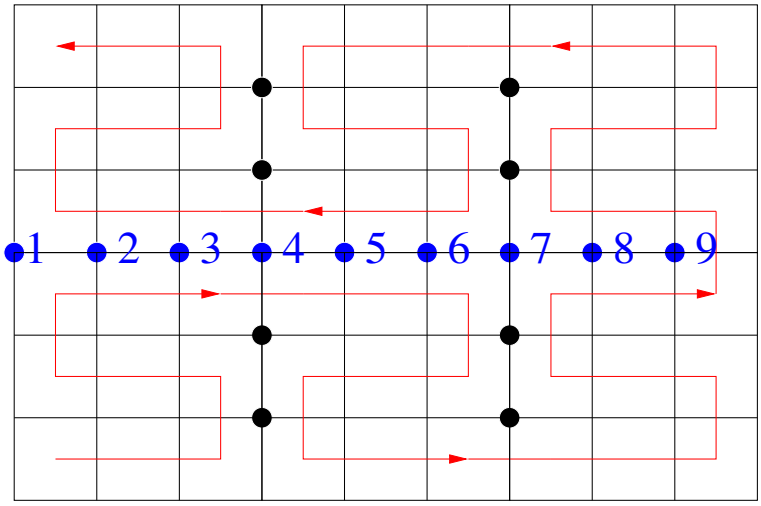

Abbildung 4.1: Äquidistante Zellen mit einbeschriebener Peano-Kurve.

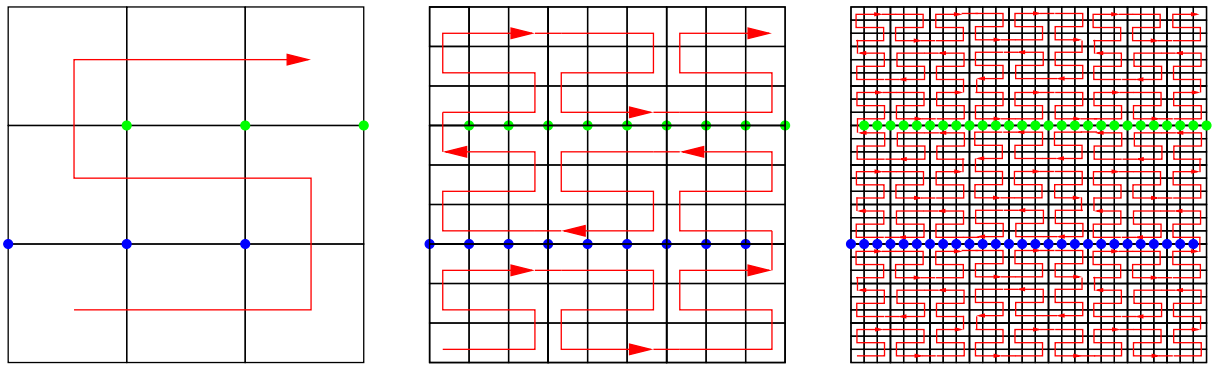

Abbildung 4.2: Die ersten drei Iterierten der Peano-Kurve.

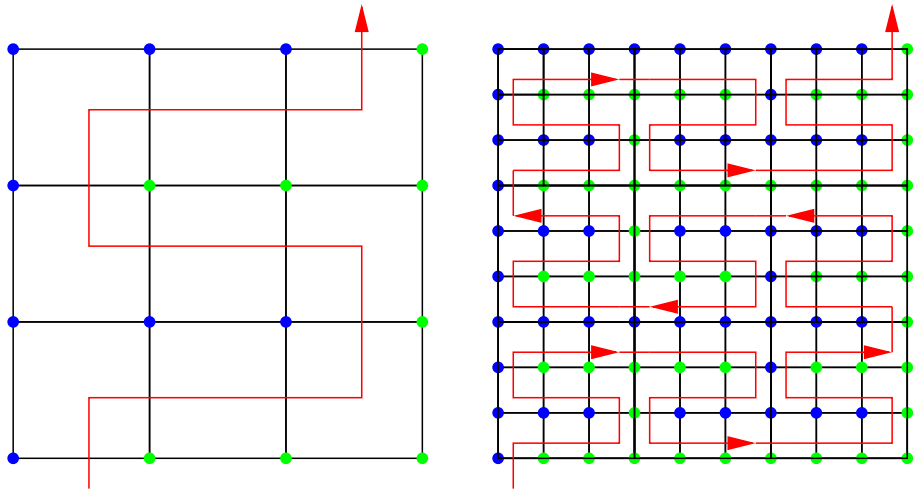

Abbildung 4.3: Färbung der Gitterpunkte im Falle der nodalen Basis.

Nun muss man sich überlegen, ob sich alle Gitterpunkte auf solchen Linien anordnen lassen und wie groß die Zahl dieser Linien und der daraus abgeleiteten Stacks ist. Im Falle der nodalen Basis (ein Freiheitsgrad pro Gitterpunkt auf dem feinsten Gitterlevel) stellt man fest, dass zur Zwischenspeicherung der Daten innerhalb einer Iteration zwei Stacks genügen. Dabei enthält der eine Stack die Punkte, die topologisch „links“ von der Kurve liegen, der anderen entsprechend die Punkte, die „rechts“ von der Kurve liegen. Die Festlegung erfolgt rekursiv zusammen mit der Erzeugung der Kurve durch „Einfärben“ der Punkte wie in Abbildung 4.3 dargestellt. Dies funktioniert immer für reguläre Gitter, die inklusive der einbeschriebenen Peano-Kurve rekursiv erzeugt werden. Für den Fall adaptiver Gitter soll ein hierarchisches Erzeugendensystem verwendet werden (siehe 3.2). Dies erfordert eine Erweiterung des oben dargestellten Konzepts, die im nächsten Abschnitt behandelt wird. Offen ist nun noch die Frage nach einer geeigneten Speicherform für die Daten zwischen den Gebietsdurchläufen (zum Beispiel zur Durchführung einer Glätteriteration). Diese Datenstruktur muss die Daten in geeigneter Reihenfolge zur Verfügung stellen, so dass sie in die oben beschriebenen Stacks geholt werden können. Näheres dazu wird in Kapitel 5 besprochen.

4.3 Erweiterung für ein hierarchisches Erzeugendensystem

Im Zusammenhang mit der Konstruktion der Stacks und der Zugriffe auf diese ist der entscheidende Unterschied zwischen der nodalen Basis und einem hierarchischen Erzeugendensystem, dass es Orte im Gitter gibt, an denen mehrere „Basis-

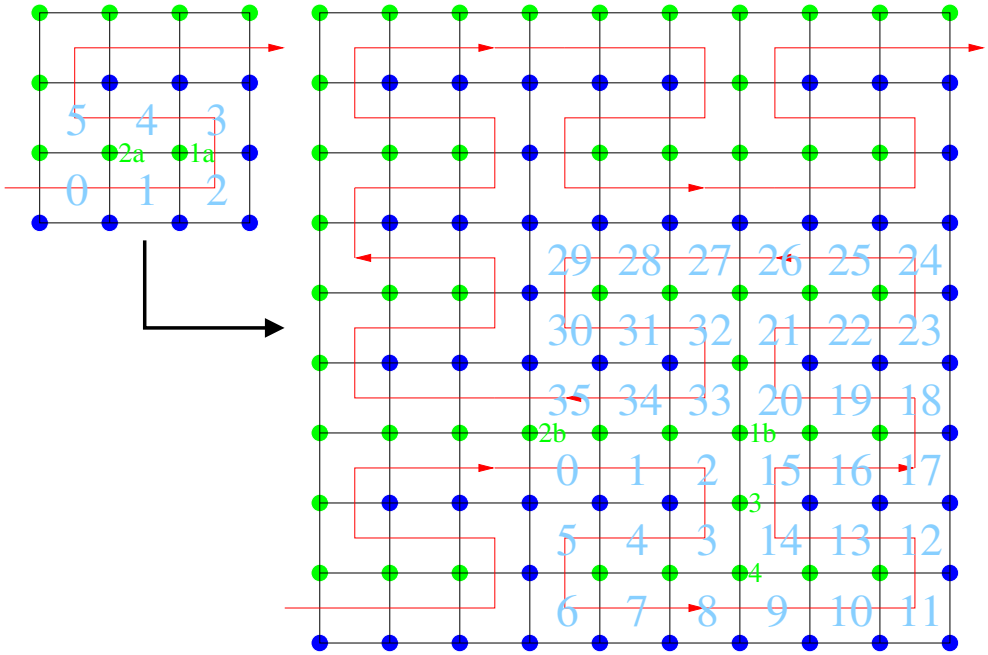

Abbildung 4.4: Rekursive Färbung der Gitterpunkte für den Fall eines hierarchischen Erzeugendensystems nach der Methode für die nodale Basis.

funktionen“ und in unserem Sinn Datenpunkte existieren. Wir setzen hier voraus, dass in diesem Fall jede Zelle auf jedem Level ihre Punktdaten von Stacks holt, sobald sie zur Bearbeitung ansteht, und ihre Punkte auf Stacks schreibt, sobald ihre Bearbeitung abgeschlossen ist. Dazu muss nun die durch die Peano-Kurve festgelegte Reihenfolge *aller* Zellen (nicht nur der Zellen des feinsten Gitterlevels wie im obigen Abschnitt) betrachtet werden. Entsprechend der rekursiven Konstruktion der diskreten Peano-Kurve ergibt sich eine Top-Down-Depth-First-Anordnung: Angefangen von der gröbsten Zelle – dem Gesamtgebiet – werden in jeder Zelle zunächst alle Sohnzellen entlang der Verfeinerung der Peano-Kurve besucht bevor die (entsprechend der gröberen Peano-Kurve) nächste Zelle desselben Levels betreten wird.

Insbesondere holt eine Zelle eines groben Levels zuerst ihre Punkte, startet dann eine rekursive Routine, die ihre neun Kindzellen auf einem feineren Level bearbeitet, um nach Beendigung dieses Programms ihre Gitterpunkte auf Stacks zu schreiben. Diese Voraussetzung ist sinnvoll, um Zellen unabhängig von ihrem Level programmieren zu können, wie wir in Kapitel 5 noch sehen werden. Untersucht man das obige Konzept mit zwei so genannten Linien- oder 1D-Stacks als Zwischenspeicher, so stellt man fest, dass dies nicht mehr ausreichend ist. Es treten zwei Probleme auf, die nun anhand des Beispiels aus Abbildung 4.4 erklärt werden. Man sieht einen rekursiven Verfeinerungs- und Färbeschritt. Punkt 1 und 2 existieren im groben Gitter als Punkte $1a$ und $2a$, im feinen Gitter als Punkte $1b$ und $2b$.

1. Bei Anwendung der Färbemethode für die nodale Basis haben die Feingitterpunkte 3 und 4 dieselbe Farbe wie der Grobgitterpunkt 2*a*. Die Zellen 3 und 8 auf dem feinen Level legen die Punkte 3 und 4 auf den „grünen“Stack, *danach* legt die Zelle 1 des groben Levels Punkt 2*a* ebenfalls auf den grünen Stack, dieser Punkt liegt nun oberhalb von den Punkten 3 und 4. Die Feinlevelzellen 9 und 14 wären jetzt nicht in der Lage, die Punkte 3 und 4 vom grünen Stack zu lesen, weil dieser durch Punkt 2*a* blockiert ist. Dieses Problem träte übrigens auch dann auf, wenn man statt des (an einigen Punkten) mehrdeutigen hierarchischen Erzeugendensystems eine hierarchische Basis verwendete, die an jedem Ort im Gitter nur eine Basisfunktion und damit nur einen Punkt in unserem Sinne besäße (der allerdings im Unterschied zur nodalen Basis „seinem“ Level zugeordnet wäre).

   Abhilfe: Es werden auf beiden Seiten der Kurve waagrechte und senkrechte Linien unterschiedlich gefärbt, es werden also zwei weitere Linien- oder 1D-Stacks eingeführt. Anhand von Abbildung 4.5 kann man sich davon überzeugen, dass das Problem damit behoben ist.

2. Das zweite Problem ist spezifisch für das hierarchische Erzeugendensystem. Wir nehmen an, dass Problem 1 bereits beseitigt ist, die Färbung also wie in Abbildung 4.5 vorgenommen wurde. Die Feingitterzelle 20 legt nun Punkt 1*b* auf den grünen Stack, dort liegt er jetzt in jedem Fall oberhalb von Punkt 2*a*. Diese Situation ist unverändert, wenn die Groblevelzelle 3 verlassen und 4 betreten wird. Es ist für Zelle 4 jetzt unmöglich den Punkt 2*a* zu holen, da der entsprechende grüne Stack von Punkt 1*b* blockiert wird.

   Abhilfe: Zusätzlich zu den Linien-Stacks werden Punkt- oder 0D-Stacks eingeführt. Dort werden in gewissen Situationen Feingitterpunkte einer Farbe zwischengespeichert, um den oben dargestellten Konflikt zu vermeiden. Hier wird also die Feingitterzelle 20 angewiesen, den Punkt 1*b* auf den grünen 0D-Stack zu legen. Der Punkt 2*a* liegt auf dem grünen 1D-Stack und ist jetzt für die Grobgitterzelle 4 dort als oberstes Element verfügbar.

Für das hierarchische Erzeugendensystem müssen an den Kellern also zwei Modifikationen vorgenommen werden. Die Anzahl der Farben wird von zwei auf vier erhöht, die Punkte auf einer Seite der Kurve werden entlang der Koordinatenachsen unterschiedlichen Stacks zugeordnet. Zusätzlich muss eine neue Klasse von Kellern eingeführt werden, die nicht mehr Linien im Gitter repräsentieren sondern gewissermaßen dazu orthogonale Linien von „übereinander“ liegenden Punkten auf einem Ort im Gitter speichern. Wie man in Abbildung 4.5 andeutungsweise erkennen kann, ist der Vorgang des Einfärbens der Punkte respektive der Zuordnung zu einer Stackgruppe rekursiv formulierbar und kann im selben Top-Down-Depth-First-Prozess wie die Erzeugung des Geometriebaums und der raumfüllenden Kurve durchge-

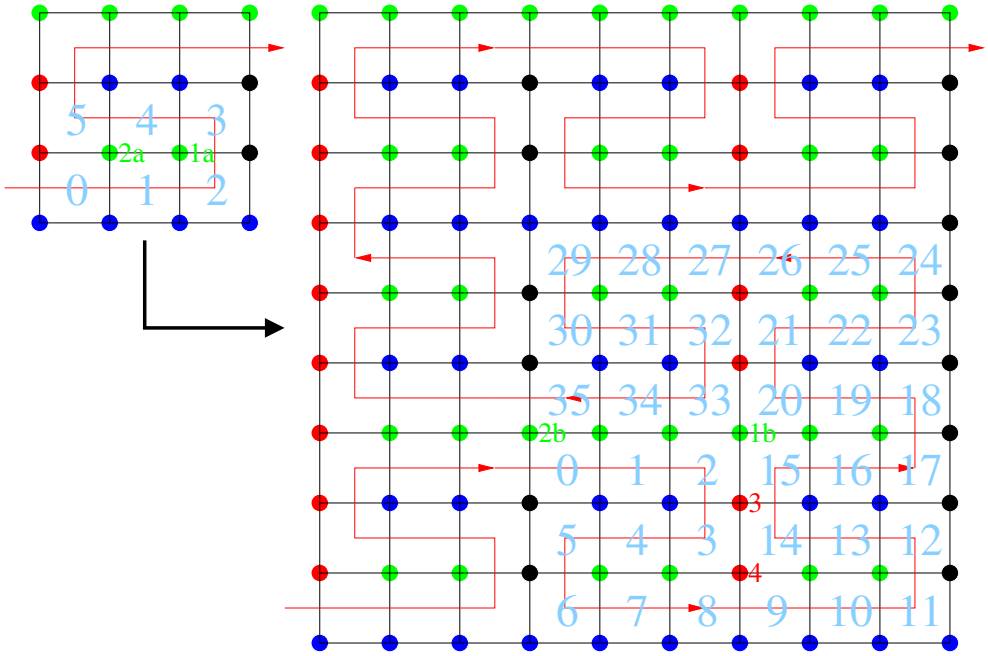

Abbildung 4.5: Rekursive Färbung der Gitterpunkte für den Fall eines hierarchischen Erzeugendensystems.

führt werden. In der konkreten Realisierung, die in Kapitel 5 erörtert ist, wird diese Färbung im Prinzip bei jedem Baumdurchlauf vorgenommen. Aus der vorgegebenen Peano-Kurve in einer Zelle ergeben sich hierbei die Farben in allen Fällen auf sehr einfache Art und Weise. Meist kann die Farbe eines Punktes direkt – insbesondere ohne Auswertung von `if`-Abfragen oder anderen Berechnungen – angegeben werden, in einigen wenigen Fällen muss ein dann aber auch sehr geringer Aufwand von wenigen Codezeilen investiert werden, um die Farbe eines Punktes zu ermitteln. Insgesamt ist die „Neufärbung“ der Punkte bei jedem Baum- oder Gebietsdurchlauf billiger, als zum Beispiel die Farbe in jedem Datenpunkt zu speichern.

Neben der damit abgeschlossenen Zuordnung der Gitterpunkte zu den Stacks muss für einen effizienten Algorithmus ein einfaches Regelwerk gefunden werden, das allein aufgrund der lokal in der aktuellen Zelle verfügbaren Informationen entscheidet, woher die aktuelle Zelle ihre vier Datenpunkte holen soll und wo sie beim Verlassen der Zelle gespeichert werden sollen. Dieser Punkt ist von großer Bedeutung, weil zum Beispiel ein Vorgehen der Art „Hole von allen Stacks das oberste Element und prüfe, welche vier Elemente die passenden sind!“ eine Vielzahl an unnötigen Vergleichsoperationen verursacht. Außerdem müsste dann eine Zelle zwangsläufig Informationen über ihre Koordinaten besitzen. Mit dem im folgenden Abschnitt dargestellten Regelwerk kommen dagegen sowohl die Zellen als auch die Punkte ohne solche Geometrieinformationen aus. Eine Überprüfung der Koordinaten auf Konsistenz wird lediglich für Debug-Zwecke eingeschaltet (siehe Kapitel 5).

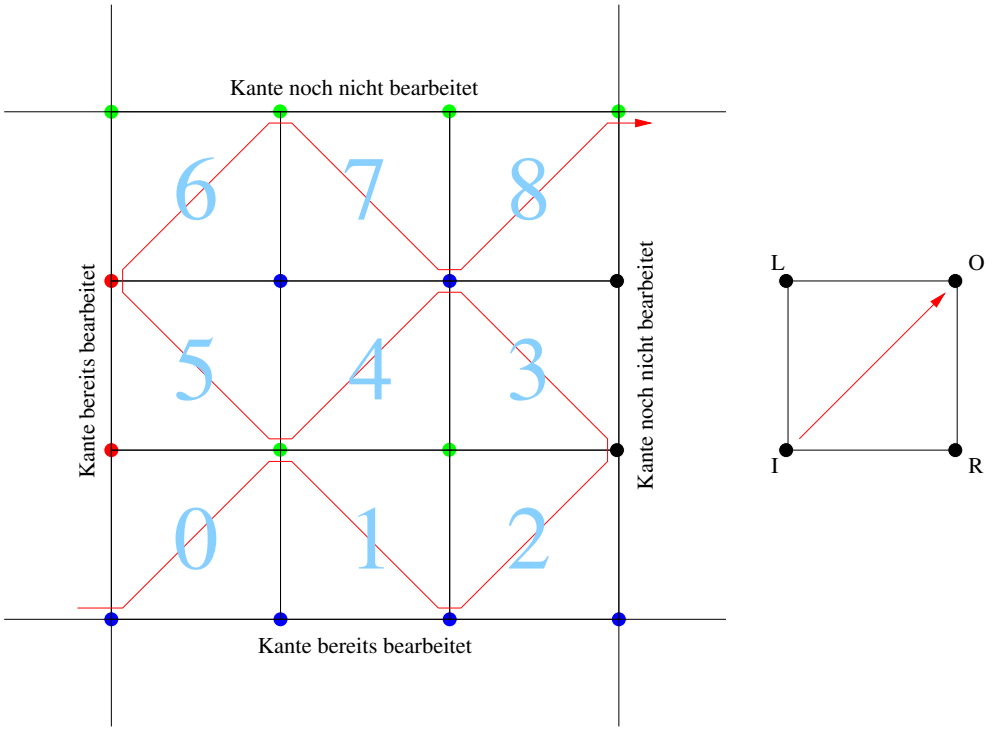

Abbildung 4.6: 3×3-Beispiel für die Erzeugung von Regelsätzen für die Kellerzugriffe.

4.4 Regelsätze für Kellerzugriffe

In diesem Abschnitt soll für einige einfache Fälle demonstriert werden, wie man aus einer einfachen Grundregel unter Zuhilfenahme von lokal zur Verfügung stehenden Eigenschaften von Zellen, die man aus dem Verlauf der Peano-Kurve ableiten kann, Regelsätze für bestimmte Typen von Zellen ableiten kann. Kompliziertere Zelltypen, die sich bei adaptiven Gittern sowie komplizierteren Geometrien ergeben (siehe Kapitel 5) werden dabei um einer kompakten Darstellung willen zunächst vernachlässigt. Wir beschränken uns auf den Fall der „inneren“ Zelle. Innere Zellen sind Zellen, in denen alle vier Eckpunkte Freiheitsgrade im Sinne der Finite-Elemente-Methode darstellen. Innere Punkte haben in unserem Kontext die Eigenschaft, dass sie von genau vier Zellen benutzt werden. Wir setzen die Existenz geeigneter Datenstrukturen zum Einlesen der Startlösung und zum Speichern der in einer Iteration erzielten Lösung voraus.[1]

Zur Definition der Grundregel für die Datenbewegungen von und zu den Stacks

[1] In unseren Programmen sind diese Strukturen auch wieder Keller, in Kapitel 5 wird näher beschrieben, warum wir sie auf diese Weise realisieren. Die entsprechenden Strukturen werden ab jetzt als Flächen- oder 2D-Stacks bezeichnet.

betrachten wir nun also einen beliebigen inneren Punkt und geben jeweils die Operationen an, die nacheinander in den vier beteiligten Zellen ausgeführt werden. Die Zellen sind dabei in der Reihenfolge des Betretens entlang der raumfüllenden Kurve mit 1 bis 4 durchnummeriert. Damit ergibt sich folgender Ablauf:

Zelle	**Lesen**	**Schreiben**
1	INPUT	0D-Stack oder Nächste Zelle
2	0D-Stack oder Vorherige Zelle	1D-Stack oder Nächste Zelle
3	1D-Stack oder Vorherige Zelle	0D-Stack oder Nächste Zelle
4	0D-Stack oder Vorherige Zelle	OUTPUT

Dabei bedeutet „Schreiben zur nächsten Zelle“ dass der Punkt über lokale Variablen direkt an die nächste Zelle übergeben wird, und dementsprechend „Lesen von der vorherigen Zelle“ dass der Punkt von der vorherigen Zelle lokal weitergegeben wird. Dies wird aus Effizienzgründen in den realen Programmen so gemacht, deshalb soll es auch Bestandteil der Darstellung an dieser Stelle sein.

Obige Grundregel ergibt sich auf kanonische Weise aus den „Abhilfen“ der in Abschnitt 4.3 beschriebenen Probleme, die das hierarchische Erzeugendensystem mit sich bringt. Es werden keinerlei globale Informationen benötigt, jeder Punkt muss lediglich wissen, wie oft er bereits gebraucht wurde. In obiger Tabelle sind noch Nicht-Determinismen enthalten, die sich aber unter Einbeziehung von zusätzlichen Informationen beheben lassen. Wichtig ist dabei, dass die Auflösung dieser Unsicherheiten ausschließlich durch Informationen möglich ist, die sich entweder in der aktuellen Zelle billig errechnen lassen oder die von der Vaterzelle im rekursiven Abarbeitungsprozess „geerbt“ werden können. Der Zugriff auf levelabhängige Informationen oder auf Informationen des Großvaters, des Urgroßvaters und so weiter muss unbedingt vermieden werden, da die daraus entstehenden Ketten von Fallunterscheidungen zu algorithmischen Komplikationen führen.

Wir zeigen die Auflösungen der Nicht-Determinismen an einem Beispiel, das in Abbildung 4.6 illustriert ist. Die Bedingungen werden je Zelle formuliert, da auch das algorithmische Vorgehen entlang der raumfüllenden Kurve zellorientiert ist. Dazu führen wir zuerst noch koordinaten-unabhängige Bezeichnungen für die Gitterpunkte einer Zelle ein (Abbildung 4.6). Zunächst ist dazu eine modifizierte graphische Darstellung der Peano-Kurve notwendig, wie sie ebenfalls in Abbildung 4.6 zu sehen ist. Sie ist motiviert durch die Tatsache, dass die eigentliche Peano-Kurve, also der Grenzwert der diskreten Kurven die Zellen jeweils an Eckpunkten betritt und verlässt. Gleichzeitig können die pro Zelle entstehenden Diagonalen als Leitmotiv angesehen werden, wie es auch in 2.2 für die Hilbert-Kurve beschrieben wurde.

Allein aus der Richtung der Diagonalen lässt sich eindeutig die nächste Verfeinerungsstufe der Peano-Kurve für die entsprechende Zelle ableiten.[2]

Entlang des Verlaufs Peano-Kurve wird nun definiert

- **I**: Eintrittspunkt der Kurve in die Zelle.
- **O**: Austrittspunkt der Kurve aus der Zelle.
- **L**: Linker Punkt, also der Punkt der beiden verbliebenen, der links von der Kurve liegt.
- **R**: Rechter Punkt entsprechend.

Für einen 3×3-Block, der ausschließlich aus inneren Zellen besteht, lässt sich allgemein folgendes Schema der Zugriffe angeben:

Zelle	**Lesen**				**Schreiben**			
	I	**L**	**R**	**O**	**I**	**L**	**R**	**O**
0	0D	1D/2D	1D/2D	2D	1D/2D	0D	NZ	NZ
1	VZ	2D	VZ	1D/2D	1D	NZ	1D/2D	NZ
2	VZ	VZ	0D/1D/2D	1D/2D	1D/2D	NZ	0D/1D	NZ
3	VZ	VZ	1D/2D	2D	1D/2D	NZ	0D	NZ
4	VZ	VZ	2D	1D	1D	2D	NZ	NZ
5	VZ	0D	VZ	1D/2D	2D	1D/2D	NZ	NZ
6	VZ	0D/1D/2D	VZ	1D/2D	1D/2D	0D/1D/2D	NZ	NZ
7	VZ	1D/2D	VZ	1D	1D/2D	NZ	2D	NZ
8	VZ	VZ	0D	1D/2D	2D	1D/2D	1D/2D	0D

Hier bedeutet „VZ“, dass der Punkt von der Vorgängerzelle übernommen wird, und „NZ“ dass der Punkt an die nächste Zelle übermittelt wird. Zusätzlich muss noch die Farbe (Zugehörigkeit des Punktes zu einer Kellergruppe) bekannt sein, die lässt sich jedoch wie oben bereits erwähnt leicht algorithmisch bestimmen.

Die in dieser Tabelle enthaltenen Nicht-Determinismen werden aufgelöst durch Informationen darüber, ob eine Außenkante des 3×3-Blocks bereits vom benachbarten Block (den es im Falle innerer Zellen immer gibt) bearbeitet wurde oder nicht. Im Einzelnen ergibt sich für diesen Fall:

- Zelle 0: linke und untere Kante bereits bearbeitet $\Rightarrow$
  Lesen L vom 1D-Stack, Lesen R vom 1D-Stack, Schreiben I auf 2D-Stack

[2] Man kann diese Zusammenhänge auch mit den Konzepten der Theorie der formalen Sprachen beschreiben. Mein Kollege Markus Pögl wird in [19] darauf näher eingehen und zeigen, dass sich die approximierenden Polygone der Peano-Kurve durch Grammatiken beschreiben lassen.

- Zelle 1: untere Kante bereits bearbeitet ⇒
  Lesen O vom 1D-Stack, Schreiben R auf 2D-Stack
- Zelle 2: untere Kante bereits bearbeitet und rechte Kante noch nicht bearbeitet ⇒
  Lesen R vom 0D-Stack, Lesen O vom 2D-Stack, Schreiben I auf 2D-Stack, Schreiben R auf 1D-Stack
- Zelle 3: rechte Kante noch nicht bearbeitet ⇒
  Lesen R vom 2D-Stack, Schreiben I auf 1D-Stack
- Zelle 4: ist bereits deterministisch
- Zelle 5: linke Kante bereits bearbeitet ⇒
  Lesen O vom 1D-Stack, Schreiben L auf 2D-Stack
- Zelle 6: linke Kante bereits bearbeitet und obere Kante noch nicht bearbeitet ⇒
  Lesen L vom 0D-Stack, Lesen O vom 2D-Stack, Schreiben I auf 2D-Stack, Schreiben L auf 1D-Stack
- Zelle 7: obere Kante noch nicht bearbeitet ⇒
  Lesen L vom 2D-Stack, Schreiben I auf 1D-Stack
- Zelle 8: obere Kante und rechte Kante noch nicht bearbeitet ⇒
  Lesen O vom 2D-Stack, Schreiben L auf 1D-Stack, Schreiben R auf 1D-Stack

Dieser Vorgang der Regeldefinition muss nun noch für alle anderen Zell- und Punkttypen und jeweils für alle Situationen bezüglich der Bearbeitungsstände der Vaterzellenkanten wiederholt werden. Zum jetzigen Zeitpunkt implementieren wir im zweidimensionalen die Zelltypen

- Innere Zelle
- Äußere Zelle
- Zelle mit mindestens einem hängenden Punkt im Falle adaptiver Gitter
- Zelle mit mindestens einem Randpunkt vom homogenen Dirichlet-Typ.

Der Zelltyp für Zellen mit Dirichlet-Randpunkten ist dabei so ausgelegt, dass auch alle anderen Zell- und Punkttypen in seinen Verfeinerungen oder auf seinen Zellen vorkommen können. Die übrigen Zelltypen sind aus Performancegründen getrennt implementiert, um Ketten von Abfragen für andere Punkttypen zu verhindern. In Kapitel 5 wird diese Vorgehensweise genauer erläutert.

Die genaue Untersuchung aller vorstellbaren Fälle (deren Zahl auf Grund der starken Strukturiertheit durch die Verwendung der Peano-Kurve durchaus beschränkt ist) führt immer auf ein dem obigen ähnliches Regelwerk, das dann implementiert werden kann. Auf Basis der Erkenntnisse für Zellen mit ausschließlich inneren Punkten ist dies zumindest für zweidimensionale Gitter auf noch überschaubare Weise möglich. Markus Pögl beschreibt in [19] einen etwas anderen Weg, der gekennzeichnet ist durch zusätzliche dimensionsiterative Rekursionen, die eine deutlich kompaktere Darstellung der unterschiedlichen Fälle erlauben. Dies ist für die dreidimensionale Variante der hier beschrieben Methodik auch notwendig, da die Anzahl der zu betrachtenden Fallunterscheidungen hier deutlich größer ist und vermutlich auf die oben beschriebene Weise realistisch nicht zu bewältigen ist.

Im folgenden Kapitel werden wir nun beschreiben, wie die in den bisherigen Kapiteln dargestellten algorithmischen und methodischen „Zutaten“ zu einem Programm kombiniert werden, das insbesondere die erstrebte Cache-Effizienz erreicht.

5 Algorithmische Realisierung

In diesem Kapitel wollen wir zeigen, wie die in den vorangegangenen Kapiteln dargestellten Ingredienzien, von denen wir an einigen Stellen gezeigt haben, dass sie sich aufs Beste kombinieren lassen, zu einem Programm zusammengefügt werden können, das zur effizienten Lösung von Partiellen Differenzialgleichungen geeignet ist. Grundsätzlich sei aber darauf hingewiesen, dass dies nicht die vollständige Dokumentation zu den im Rahmen dieses Projekts entstandenen Programmen sein kann. Allein der Löser, das zentrale Ergebnis der Programmierungen für diese Arbeit, umfasst ca. 28.000 Programmzeilen, eine vollständige Darstellung aller Details der Realisierung ist an dieser Stelle nicht möglich. Wir möchten uns deshalb darauf beschränken, die entscheidenden Stellen darzustellen, an denen das Zusammenwirken der „Zutaten“ stattfindet und die Performance-Gewinne erzielt werden.

Das im Rahmen dieser Arbeit entwickelte Programm zur Lösung der Poisson- und der Stokes-Gleichung ist Bestandteil eines Projekts, dass derzeit in unserer Gruppe realisiert wird. Neben den Lösern für zwei- und dreidimensionale Testfälle ist bereits ein Gittergenerator für den 2D-Fall im Rahmen eines interdisziplinären Projekts entstanden. Weitere Programme oder Programmteile sind in Vorbereitung, mehr dazu ist in Kapitel 7 nachzulesen.

Das hier vorgestellte Programm `stream` für den zweidimensionalen Fall ist komplett in `ANSI-C` implementiert [16],[13]. Diese Designentscheidung wurde vor allem aus Gründen der Performance und wegen des experimentellen Charakters des Programms so getroffen. Es erschien uns insbesondere nicht sinnvoll, eine objektorientierte Sprache wie `C++` einzusetzen, weil der dafür nötige Top-Down-Entwicklungsprozess in unserem Fall nicht gegeben war. Dies hat sich insbesondere vor dem Hintergrund vieler grundsätzlicher Änderungen etwa in den Datenstrukturen während der Entwicklung dieses Programms als richtig erwiesen. Diese Änderungen wären in einem objektorientierten Ansatz nur mit wesentlich höherem Aufwand zu bewerkstelligen gewesen. Ein Redesign mit einer Sprache wie `C++` erscheint aber mittelfristig sinnvoll und notwendig, mehr dazu ebenfalls in Kapitel 7.

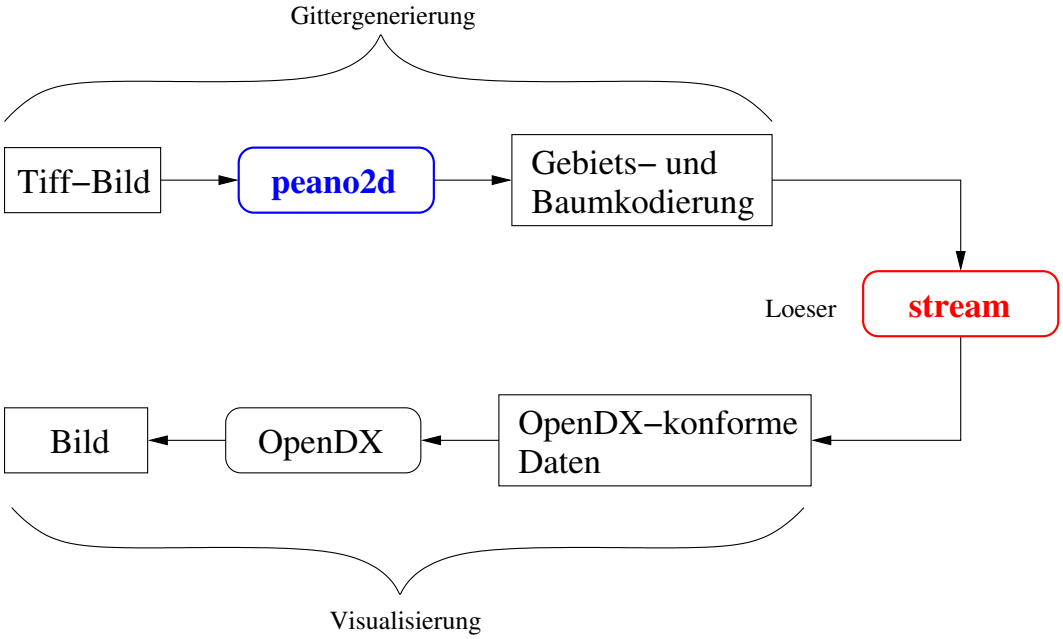

Abbildung 5.1: Grundsätzlicher Ablauf der Simulation von den Eingabedaten bis zur Visualisierung. Die im Rahmen dieses Projekts neu entwickelten Programme sind blau beziehungsweise rot gefärbt.

5.1 Grundsätzliche Struktur und Einbettung des Programms

Zunächst wollen wir einen Überblick über die grundsätzliche Struktur des Programms `stream` geben, das die Umsetzung der Ideen der vorangegangenen Kapitel und den wesentlichen Teil der praktischen Arbeit dieser Dissertation ausmacht.

Das Programm ist eingebunden in einen „workflow“ wie er in Abbildung 5.1 dargestellt ist. Ausgehend von einem `tiff`-Bild [30], das die Geometriebeschreibung enthält, wird ein Rechenbaum beziehungsweise ein Rechengitter erzeugt. Das Bild enthält genau so viele Pixel, wie als Auflösung auf dem feinsten Level vom Benutzer gewünscht ist. Soll die Gitterweite auf dem feinsten Level $1/3^n$ betragen, so muss das `tiff`-Bild $3^n \times 3^n$ Pixel haben. Die Pixel können dabei die Farben schwarz (Außenbereich), weiß (Innenbereich), rot (Dirichlet-Rand) oder grün (nur im Inneren, hier muss die feinste Auflösung erhalten bleiben) annehmen. In dem im Rahmen des Interdisziplinären Projekts [10] entwickelten Programms `peano2D` werden zunächst die Pixel des Eingabebildes als Zellen interpretiert und ein Peano-Kurve einbeschrieben. Danach wird vom Benutzer steuerbar versucht, in einem Bottom-Up-Prozess, bei dem die Generierung der Peano-Kurve gewissermaßen umgekehrt wird, gleichartige Zellen feiner Level zu Groblevelzellen zusammenzufassen. Dadurch entsteht

ein adaptives Gitter, der Rand eines eingebetteten Rechengebiets bleibt jedoch per Definition immer in maximaler Auflösung erhalten. Die Feinheit des Rechengitters ist vor allem wegen der Möglichkeit, durch grüne Pixel eine Vergröberung zu verhindern, dabei nicht zwangsläufig korreliert mit der Feinheit der Geometrieauflösung (siehe Abschnitt 2.1). Das gesamte Gitter wird von `peano2D` auf einfache Weise über Integerzahlen codiert und in einer wiederverwendbaren Datei abgelegt.[1] Diese Datei wird von `stream` eingelesen. Außerdem werden als Eingabe eine Steuerdatei, die Informationen über die Anzahl der Zellen und die Länge der Codierung des Rechengitters enthält, und einige Kommandozeilenoptionen benötigt. Innerhalb von `stream` wird dann mit einem iterativen Verfahren eine Lösung entweder der Poisson- oder der Stokes-Gleichung auf dem gegebenen Gitter berechnet. Diese kann in einem Datenformat ausgegeben werden, das kompatibel mit OpenDX von IBM ist. Somit kann die Lösung schließlich direkt mit den entsprechenden OpenDX-Funktionen visualisiert werden [29].

Grundsätzliche Dinge wie zum Beispiel Erzeugen der Ausgabedaten für OpenDX, Konsistenzcheck auf den Eingabedaten, Lösung der Poisson- oder Stokes-Gleichung können über Optionen im `Makefile` eingestellt werden. Für spätere Performanceanalysen ist insbesondere die Möglichkeit, die Konsistenzchecks und die OpenDX-Ausgabe abschalten zu können, wichtig, weil dadurch zusätzliche Unterprogramme und I/O-Operationen verhindert werden, die auf die Qualität der Lösung keinen Einfluss haben, aber unnötige CPU- und Laufzeit kosten.

Nun wollen wir die innere Struktur von `stream` genauer beschreiben. Das folgende Codebeispiel ist eine um der kompakten Darstellung willen gekürzte Version des Hauptprogramms für die Lösung der Poisson-Gleichung. Daran lässt sich die grundsätzliche Funktionsweise des Programms beschreiben.

```
int main (int argc, char *argv[])
{
   /* Variablen deklarieren und initialisieren*/
   ...
   /* Kommandozeilenparameter o.k.? */
   ...
   /*Parameter einlesen*/
   readParam(argv[2]);
   /*Maximale Iterationszahl einlesen*/
   sscanf (argv[3],"%d", &max_iter);
```

[1] Eine detaillierte Beschreibung des Programms findet sich in [10], hier haben wir nur die Kernfunktionalität dargestellt. Insbesondere bietet das Programm große Flexibilität in Bezug auf die erzeugten Rechengitter. So kann zum Beispiel die Vergröberung vollständig verhindert werden und man erhält ein reguläres Gitter oder es kann die Ausbreitung von Vergröberungen in Randnähe begrenzt werden, zum Beispiel durch die Bedingung, dass benachbarte Zellen sich nur um maximal ein Level unterscheiden dürfen.

```
   sscanf (argv[4],"%d", &resolution);
   /*Einige globale Variablen initialisieren*/
   ...
   RESID_EPS = 1.0e-5;
   ...
#ifdef _CHECKS
   /*CELLLIST initialisieren*/
   CL_IND = 0;
   CL_INDOD = 0;
   initCellList(NOC);
#endif
   /*STACK initialisieren*/
   initStacks(NOC,resolution);
    /*TREE initialisieren*/
   initTrees(TREECODELENGTH);
   /*Bitkodierung des Baumes einlesen*/
   readTree(DIRECTION, argv[1]);
   /*Erste Zelle initialisieren*/
   setRootCellData(localCellData,treeElem);
   /*Status der Eckpunkte der ersten Zelle initialisieren*/
   for (i=0; i<_PPC; i++) {
      src[i] = _TOUCHED;
   }
   /*initialer Baumdurchlauf, Zellliste aufbauen*/
   initBuildCellList(localCellData,src);
#ifdef _ODX
   /*OpenDX-Ausgabe*/
   prepareForIteration(localCellData,src,treeElem);
   dxOutputIteration(localCellData,src,0);
#endif
   /*Iterieren*/
   /* Residuen kuenstlich gross machen ;-)*/
   MAX_RESIDUUM_U = 1.0;
   i = 1;
   while (i <= max_iter && !(fabs(MAX_RESIDUUM_U) < RESID_EPS)) {
      /* Residuen zu null machen */
      MAX_RESIDUUM_U = 0.0;
      /* Iterationsschritt vorbereiten und durchfuehren */
      prepareForIteration(localCellData, src, treeElem);
      iteration(localCellData,src,treeElem);
      i = i+1;
```

```
   }
#ifdef _ODX
   /*OpenDX-Ausgabe*/
   prepareForIteration(localCellData,src,treeElem);
   dxOutputIteration(localCellData,src,i-1);
#endif
   exit(0);
}
```

Nach einigen Deklarationen, Initialisierungen globaler Variablen und Überprüfungen der Startparameter wird, falls ein Konsistenzcheck für die Eingabedaten (Geometrie- und Gitterbeschreibung) gewünscht ist, mit Hilfe der `#ifdef`-Anweisung von `C` die Datenstruktur `CELLLIST` initialisiert. Dies erfolgt nur, wenn im `Makefile` die Option `D_CHECKS` aktiviert wurde. Die Struktur `CELLLIST` enthält Speicherplatz und Zugriffsmethoden, um jede Zelle des Rechengebiets mit topologischen Informationen wie zum Beispiel ihren vier Eckpunkten zu speichern. Diese können dann verwendet werden, um etwa beim Lesen der Eckpunkte von den Stacks zu überprüfen, ob die richtigen Punkte geholt wurden. Dies hat sich zur Verifizierung der Programmierung der Stackzugriffe mit Hilfe der in Abschnitt 4.4 beschriebenen Regelsätze als sehr nützlich erwiesen. Ebenso lassen sich Fehler im `tiff`-Bild, dass von `peano2d` verwendet wird, relativ leicht finden. Abbildung 5.2 zeigt einen Fehler in der Färbung der Randzellen.

Danach werden die Speicherstrukturen `STACK` und `TREE` initialisiert. Pro benötigtem Keller (siehe Abschnitte 4.2 und 4.3) enthält `STACK` ein `C`-Array mit den Zugriffsroutinen `push` und `pop`. Die Details der Implementierung von `STACK` sind gegenüber den Routinen, die sie verwenden, verborgen, die Routinen kennen nur die Anzahl und die Namen der Stacks sowie die durch `push` und `pop` definierten Schnittstellen zu den Kellern. Die Elemente der Stacks sind `structs`, die je nach Verwendungszweck einige `int`- und `float`- oder `double`-Werte enthalten. `TREE` ist ein Array von `int`-Werten, das die Codierung des Baumes beziehungsweise seiner Zellen und gegebenenfalls die Typen der Eckpunkte einer Zelle enthält.

Während in Kapitel 4 lediglich die innerhalb eines Baumdurchlaufs benötigten 0D- und 1D-Stacks beschrieben wurden, benötigt man nun zusätzliche so genannte 2D-Ein- und Ausgabestacks, die vor Beginn einer Löseriteration sämtliche Datenpunkte (in der richtigen Reihenfolge, in der sie benötigt werden) enthalten sowie die während einer Iteration anfallenden „fertigen“ Punkte (also solche Punkte, die bereits von allen benachbarten Zellen besucht wurden) aufsammeln. Man kann sich leicht davon überzeugen, dass das „rückwärts“ Abarbeiten aller Punkte des 2D-Ausgabestacks mit Hilfe von `pop` gerade einem gegenüber der Reihenfolge des Eingabestacks „umgekehrten“ Baumdurchlauf entspricht. Der Ausgabestack wird daher direkt als Eingabestack für die nächste Iteration verwendet. Es ergibt sich eine von Iteration

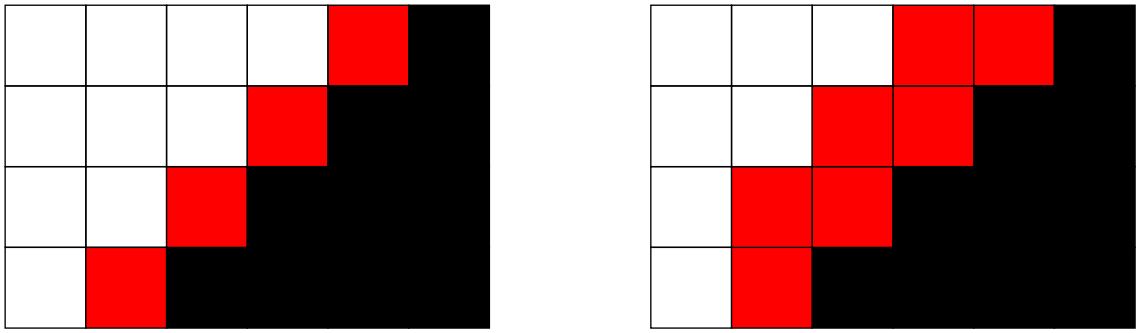

Abbildung 5.2: Fehler in der Farbcodierung des Rechengebiets.

zu Iteration alternierende Durchlaufreihenfolge über den Zellbaum.

Zur Vorbelegung der „physikalischen“ Werte wird nach der Initialisierung der ersten Zelle, die per Definition das Einheitsquadrat[2] ist, wird das Unterprogramm `initBuildCellList(...)` gestartet. Dies ist ein rekursives Programm, das das gesamte Gebiet, also alle Zellen auf allen Leveln, einmal durchläuft. Dabei werden die Stackelemente (siehe oben) für alle Gitterpunkte erzeugt, im Falle der Poisson-Gleichung die Startlösung $u^0 \equiv 0$ gesetzt und die Stackelemente entsprechend der Regelsätze wie in Abschnitt 4.4 dargestellt über die Keller bewegt. Nach dem Durchlauf dieses Programms sind für alle Gitterpunkte oder genauer gesagt für alle Freiheitsgrade Stackelemente auf dem 2D-Ausgabestack abgelegt. Die Rekursion erfolgt gemäß der rekursiven Definition der Peano-Kurve. Die neun in die „Vaterzelle“ eingebetteten Kindzellen werden in einer Schleife abgearbeitet. Wenn eine dieser Zellen verfeinert ist, wird das Programm `initBuildCellList` mit dieser Zelle als Vaterzelle erneut – und damit rekursiv – aufgerufen und immer so fort. Für den genauen Ablauf von `initBuildCellList` sei auf Abschnitt 5.2 verwiesen. Dort wird das Unterprogramm `iteration` genauer erklärt, `initBuildCellList` läuft nach genau dem gleichen Schema ab.

Die Startlösung wird zur Visualisierung mit OpenDX ausgegeben, wenn über den `Makefile`-Schalter `D_ODX` die Erzeugung der Ausgabedateien aktiviert ist. Das Programm `dxOutputIteration` bewerkstelligt dies, indem es erneut auf dieselbe rekursive Weise wie `initBuildCellList` das Gebiet durchläuft und die gelesenen Daten in Ausgabedateien schreibt. Die prinzipielle interne Struktur dieses Programms ist wieder die gleiche wie `iteration`.

Nun startet die Schleife für die Löseriterationen. Die relevanten Gitterpunkte lie-

[2] Jedes beliebige beschränkte Simulationsgebiet lässt sich in eine quadratische Umgebung einbetten, welche wiederum auf das Einheitsquadrat abgebildet werden kann. Beispiele dazu finden sich in Kapitel 6.

gen auf dem 2D-Ausgabestack so angeordnet, dass entlang der Peano-Kurve die zuerst verwendeten Punkt unten und die zuletzt verwendeten Punkte oben liegen. Deshalb wird für den nächsten Gebietsdurchlauf – also die erste Löseriteration – der 2D-Ausgabestack zum 2D-Eingabestack. Somit liegen die Punkte genau in der benötigten Reihenfolge auf dem Eingabestack und jede Notwendigkeit für Umspeicherung oder Umindizierung entfällt. Das Gebiet wird nun in umgekehrter Richtung entlang der Kurve durchlaufen. An einigen wenigen Stellen in den Unterprogrammen für die Stackzugriffe (siehe 5.2) muss bekannt sein, welche Durchlaufrichtung aktuell vorliegt. Dies wird über eine globale Variable gesteuert, die neben einigen anderen organisatorischen Daten vom Unterprogramm `prepareForIteration` vor Beginn einer Iteration passend gesetzt wird. Die Schleife für die Iterationen terminiert entweder dann, wenn die maximale Iterationszahl `max_iter` erreicht ist (wird vom Benutzer als Kommandozeilenoption übergeben) oder wenn das betragsmäßig größte Residuum die vorgegebene Schranke `RESID_EPS` unterschreitet.

Falls die Erzeugung von Visualisierungsdaten gewünscht ist, wird schließlich die Näherungslösung, die in der letzten Iteration erzielt wurde, in neue Ausgabedateien geschrieben.

5.2 Stacks und Zugriffe

In diesem Abschnitt wollen wir nun detaillierter auf die Struktur der rekursiven Routinen, die den Gebietsdurchlauf realisieren, und die Implementierung der Stacks sowie ihrer Zugriffsschnittstellen eingehen. In diesen Routinen stecken sowohl der wesentliche Teil des Programmieraufwands als auch die entscheidenden Maßnahmen zur Sicherstellung der für das Stack-Konzept erhofften hohen Performance im L2-Cache moderner Prozessoren.

Betrachten wir dazu ein gekürztes und vereinfachtes Codebeispiel. Die oben bereits erwähnte rekursive Routine `iteration` realisiert den rekursiven Zelldurchlauf entsprechend der durch die Peano-Kurve definierten Top-Down-Depth-First-Strategie. Es ist auf genau die gleiche Weise organisiert wie die Programme `initBuildCellList` und `dxOutputIteration`, die lediglich mit anderen Aufgaben versehen ebenfalls alle Zellen einmal durchlaufen.

```
void iteration(sData fatherCellData[_PPC], int src[_PPC],
               int fcTreeElem[_NOTREEELEM]) {
   /*Deklarartionen*/
   ...
   /*Schleife ueber 9 Zellen*/
   for (cellCount=0; cellCount<_NOCREF; cellCount++) {
      /*Naechstes Element von TREE waehlen*/
```

```
treeElem[4] = TREE[DIRECTION].tree[TREE_IND].elem;
/*Status der Randpunkte festlegen*/
cellSetSrc(cellCount, localSrc, src);
/*Zelltyp?*/
switch (treeElem[4]) {
   case _INNERCELL_R:
      /*Daten von Stacks und Vorgaengerzellen holen*/
      innerCellGetData(cellCount,src,xOffSet,yOffSet,
            localCellData,fatherCellData,transferCellData);
      /*Tiefe erhoehen*/
      DEPTH = DEPTH + 1.0;
      /*einiges Organisatorisches*/
      ...
      /*Rekursiver Aufruf von iteration*/
      iteration(localCellData, localSrc, treeElem);
      /*Restringieren*/
      restrict(cellCount,xOffSet,yOffSet,localCellData,
               fatherCellData,fcTreeElem);
      /*Daten auf Stacks schreiben*/
      innerCellDataToStacks(cellCount,src,xOffSet,yOffSet,
            localCellData,fatherCellData,transferCellData);
      break;
   case _INNERCELL_NR:
      /*Daten von Stacks und Vorgaengerzellen holen*/
      innerCellGetData(cellCount,src,xOffSet,yOffSet,
            localCellData,fatherCellData,transferCellData);
      /*einiges Organisatorisches*/
      ...
      /*Stern auf Zelle berechnen*/
      innerCellApplyStencil(cellCount,xOffSet,yOffSet,localCellData,
                            fatherCellData);
      /*Restringieren*/
      restrict(cellCount,xOffSet,yOffSet,localCellData,
               fatherCellData,fcTreeElem);
      /*Daten auf Stacks schreiben*/
      innerCellDataToStacks(cellCount,src,xOffSet,yOffSet,
            localCellData,fatherCellData,transferCellData);
      /*einiges Organisatorisches*/
      ...
      break;
   case _INNERCELL_HP_R:
```

```
            ...
            break;
         case _INNERCELL_HP_NR:
            ...
            break;
         case _HOMDIR_R:
           ...
           break;
         case _HOMDIR_NR:
           ...
           break;
         case _OUTERCELL_R:
            ...
           break;
         case _OUTERCELL_NR:
            ...
           break;
         default:
            sprintf(abortstr,"iteration: Typ %d unbekannt!",treeElem[4]);
            criticalAbort(abortstr);
            break;
      }
   }
   DEPTH = DEPTH - 1.0;
}
```

Die Routine `iteration` wird nur in den Knoten des Gitterbaums, niemals in den Blättern aufgerufen und besteht deshalb hauptsächlich aus einer Schleife über die neun Kindzellen der aufrufenden Zelle.[3] Die Kindzellen werden dabei in der durch die Peano-Kurve vorgegebenen Reihenfolge besucht. Die Routine erhält von ihrem Aufrufer – das sind entweder das Hauptprogramm oder `iteration` selbst – als Parameter die Stackelemente, die die vier Eckpunkte der Zelle, in der es gestartet wird, enthalten (`sData fatherCellData[...]`), sowie die Informationen über den Bearbeitungsstatus der anliegenden Kanten (`int src[...]`) und die Punkttypen der Zelle (`int fcTreeElem[...]`). Nach dem Holen der Zelltypinformation (siehe 4.4) der aktuellen Kindzelle (`treeElem[4]= ...`) und dem Festlegen der lokalen Situa-

[3] Dieses Abbrechen der Rekursion vor Erreichen der Blattebene des Baumes ergibt sich aus der Notwendigkeit, in den Kindzellen Informationen der Vaterzelle zur Verfügung zu haben, um das in 4.4 vorgestellte Regelwerk zum Lesen und Schreiben der Punktdaten anwenden zu können.

tion des Bearbeitungsstatus der Kanten dieser Zelle[4] (`cellSetSrc(...)`) wird der Zelltyp in einer `switch`-Anweisung ausgewertet und je nach Typ der entsprechende Programmteil gestartet. Die Unterscheidung nach Zelltypen wird hier vor allem aus Performancegründen getroffen. Betrachtet man die in Abschnitt 4.4 beschriebenen Regeln für die Kellerzugriffe für unterschiedliche Zelltypen wie zum Beispiel innere Zellen oder Randzellen, so stellt man fest, dass sich insgesamt eine große Zahl von Fallunterscheidungen ergibt. Betrachtet man jedoch einzelne Zelltypen isoliert, so ist die Zahl der Fälle oft recht überschaubar. Daher bekommt jeder Zelltyp also „seine eigenen" Steuerprogramme für die Zugriffe auf die Keller (siehe unten). Ein weiterer wichtiger Grund für die Trennung der Zelltypen ergibt sich aus der Betrachtung der Häufigkeit, in der die Zelltypen Verwendung finden. Es ist leicht einzusehen, dass es im Allgemeinen größenordnungsmäßig $O(n^2)$ innere beziehungsweise äußere Zellen jedoch nur $O(n)$ Randzellen geben wird. Deshalb sollten die Routinen, die innere beziehungsweise äußere Zellen behandeln, nicht mit dem unnötigen Ballast der Fallunterscheidungen für Randzellen belastet sein.

Exemplarisch für die Zelltypen betrachten wir im Folgenden innere weiter verfeinerte Zellen (`case _INNERCELL_R`) und innere nicht weiter verfeinerte Zellen (`case _INNERCELL_NR`). Die beiden Fälle unterscheiden sich im wesentlichen nur dadurch, dass in nicht weiter verfeinerten Zellen der rekursive Aufruf der Routine durch andere Operationen (im Falle von `iteration` ein Glätterschritt auf der Basis des lokalen FEM-Sterns) ersetzt wird. Für beide Fälle werden jedoch immer die Daten der Eckpunkte der aktuellen Zelle von den Stacks geholt (`innerCellGetData(...)`). Wird ein Punkt dabei vom 2D-Stack gelesen – also in der aktuellen Iteration zum ersten Mal verwendet –, dann wird (im Baumabstieg) jetzt die zur Berechnung der dehierarchischen Funktionswerte benötigte Mehrgitterinterpolation des Punktes aus seiner Vaterzelle durchgeführt. Anschließend erfolgt der rekursive Aufruf von `iteration` beziehungsweise der Glättungsschritt. Nach der Abarbeitung des an dieser Stelle eingehängten Unterbaums, also im Baumaufstieg, werden die jetzt vorliegenden relevanten Korrekturen auf die Vaterzelle dieser Zelle übertragen (`restrict(...)`), also die für das additive Mehrgitterverfahren notwendigen Operationen durchgeführt.[5] Schließlich werden die Punkte der Zelle auf die entsprechenden Stacks geschrieben oder an die nächste Zelle weitergereicht (`innerCellDataToStacks(...)`).

Die anderen Zelltypen, die im Codebeispiel aus Platzgründen ohne die spezifischen Routinen angegeben sind, werden auf genau die gleiche Weise behandelt, es finden lediglich spezialisierte Versionen der `...GetData`- und `...DataToStacks`-Routinen Anwendung.

[4] Diese Information kann algorithmisch aus dem Bearbeitungsstatus der Vaterzelle zusammen mit der lokalen Richtung der Peano-Kurve (siehe Abschnitt 4.3 und Abbildung 4.6) im Grunde einfach berechnet werden, wir verzichten aber aus Platzgründen auf eine detaillierte Darstellung.

[5] Die notwendigen Details zum Verständnis der Operationen, die durch das Mehrgitterverfahren bedingt sind, werden in Abschnitt 5.3 genauer erläutert.

Die oben erwähnten Stackzugriffsroutinen sind nun recht direkte Umsetzungen von den in Abschnitt 4.4 exemplarisch dargestellten Regelsätzen für Stackzugriffe. Im Wesentlichen enthalten die Routinen abhängig von der lokalen Kurvenrichtung jeweils neun Fallunterscheidungen für jede der neun Kindzellen einer Zelle. Jeder dieser Fälle beschreibt dann eine Folge von maximal vier Anwendungen von `pop` oder `push`, mit denen die Eckpunktdaten von Stacks geholt oder auf Stacks geschrieben werden. In den meisten Fällen sind das jeweils zwei Aufrufe von `push` oder `pop`, die beiden übrigen Punkte werden entsprechend von der Vorgängerzelle übernommen oder an die Nachfolgerzelle übergeben. Diese Stackzugriffsroutinen sind zwar in ihrer Struktur sehr einfach, werden jedoch durch das Ziel, für den einzelnen Fall möglichst wenige Operationen zu erzeugen und die dadurch entstehende Notwendigkeit, jeden Fall getrennt auszuprogrammieren, im Programmtext relativ lang. Sie machen einen großen Teil der insgesamt etwa 28.000 Programmzeilen aus.

Zur konkreten Implementierung der Stacks sei noch erwähnt, dass jeder der zehn Stacks[6] ein `array` von Stackelementen ist. Das Handling der Indizierung dieser `arrays` steckt vollständig maskiert in den unmittelbaren Zugriffsprogrammen `push` und `pop`, die übrigen Programme wissen davon nichts. Falls über das `Makefile` die Konsistenzchecks eingeschaltet sind, findet bei jedem Kellerzugriff auch eine Überprüfung auf Unter- beziehungsweise Überlauf der Feldgrenzen statt.

Es gibt derzeit zwei Typen von Stackelementen. Diese beinhalten für die 2D-Stacks, die zum Transport der Näherungslösung von einer Iteration zur nächsten benötigt werden, nur eine Gleitpunktzahl pro Gitterpunkt respektive Freiheitsgrad, den Koeffizienten der Darstellung der numerischen Lösung auf dem hierarchischen Erzeugendensystem. Die Elemente der 0D- und 1D-Stacks tragen neben den ganzzahligen Informationen „Farbe“und „Tiefe“ je eine Gleitpunktzahl für die Koordinaten x, y im Einheitsquadrat, den hierarchischen Überschuss u, die im Mehrgittersinn dehierarchisierte Lösung u_{ip} sowie zwei Speicherplätze r_u und k für zu errechnende und zu akkumulierende Überschüsse.[7] In Abschnitt 5.3 wird dargestellt, warum hier zwei Werte benötigt werden. Die Ergänzung respektive Löschung dieser zusätzlichen Werte beim Lesen von oder Schreiben auf einen 2D-Stack wird von den hierfür spezialisierten `push`- und `pop`-Routinen gewährleistet.

Die Anzahl von Elementen, mit denen die Keller zum Programmstart initialisiert

[6] Dies sind je ein 0D- und ein 1D-Stack pro Farbe, also insgesamt acht, sowie je ein 2D-Eingabe und 2D-Ausgabe-Stack, also in Summe zehn Stacks.

[7] Die Tiefe sowie die Koordinaten x und y werden vor allem für Debugging-Zwecke und für die Konsistenzchecks eingesetzt, ihr Vorhandensein bringt aber auch an einigen Stellen mehr Bequemlichkeit bei der Gestaltung von Routinen. Eine Einsparung dieser Daten in den 0D- und 1D-Stacks ist nicht nötig, da die 0D- und 1D-Stacks in ihrer Höhe linear von der Anzahl der Gitterpunkte in *einer* Raumrichtung beziehungsweise sogar nur von der Tiefe des Rechenbaums abhängen (siehe Ende dieses Abschnitts). Die mögliche Einsparung an Daten im Arbeitsspeicher wäre bei weitem von dem dann nötigen Mehraufwand in der Programmierung und damit der Laufzeit des Programms überwogen.

werden, wird abhängig von der maximalen Tiefe t_{max} des vorliegenden Baumes, der Anzahl n der Gitterpunkte in einer Raumrichtung auf dem feinsten Gitterlevel und der Anzahl N der Zellen abgeschätzt. Diese Schätzungen beziehen sich bei adaptiven Gittern immer auf das entsprechende volle Gitter der selben maximalen Tiefe. Dies ist nicht übermäßig zu teuer, weil die Höhe der 2D-Stacks mit $O(N)$, die 1D-Stacks mit $O(n)$ und die 0D-Stacks mit $O(t_{max})$ estimiert werden. Dadurch sind die 2D-Stacks, die den größten Teil des benötigten Speichers ausmachen, immer direkt von der Größe des Problems abhängig initialisiert, die weniger ins Gewicht fallenden 0D- und 1D-Stacks sind für adaptive Gitter immer etwas zu groß geschätzt, die Abweichung vom tatsächlichen Bedarf beträgt jedoch in der Regel nur wenige Prozentpunkte.

5.3 Implementierung des Mehrgitterverfahrens

Nach der detaillierteren Darstellung des Programms `stream` bezüglich seiner Organisation, Datenhaltung und seines Ablaufs wollen wir nun auf die Realisierung der numerischen Verfahren, insbesondere des Mehrgitterverfahrens und des Glätters eingehen. Dies geschieht im Detail anhand des Programms für die Lösung der Poisson-Gleichung, im nächsten Abschnitts gehen wir noch auf die Modifikationen für die Stokes-Gleichung ein.

Betrachtet man allgemein klassische Gauß-Seidel- und Jacobi-Verfahren auf Knotenbasen, so ergibt sich als Unterschied, dass beim Jacobi-Verfahren die Verbesserung einer Variablen während einer Iteration erst in der nächsten Iteration verwendet wird, wohingegen beim Gauß-Seidel-Verfahren die Verbesserung einer Variablen innerhalb einer Iteration sofort verwendet wird und damit in die Verbesserung aller noch folgenden Variablen eingeht. Für beide Verfahren erhält man dieselbe Konvergenzordnung, im Falle des Gauß-Seidel-Verfahrens jedoch mit einem besseren Vorfaktor.

Griebel hat in seiner Arbeit [11] gezeigt, dass sich hier für Mehrgitterverfahren eine Analogie zeigt. Er betrachtet beide Verfahren auf hierarchischen Erzeugendensystemen und ordnet die Reihenfolge der Auswertungen auch levelweise an. Es zeigt sich, dass sich das Jacobi-Verfahren als additive Teilraumkorrekturmethode und das Gauß-Seidel-Verfahren als multiplikative Teilraumkorrekturmethode interpretieren lässt, wobei jeder der Teilräume aus genau einem Punkt besteht. Fasst man nun jeweils die Punkte eines Gitterlevels zu einem Block zusammen, so lässt sich das entsprechende Block-Jacobi-Verfahren (in Kombination mit geeigneten Glätteriterationen innerhalb der Blockstruktur) auf jedem Level als additives Mehrgitterverfahren interpretieren, während sich aus dem Block-Gauß-Seidel-Verfahren ein multiplikatives Mehrgitterverfahren ergibt. Auch hier ist der Vorfaktor der Konvergenzordnung für die Gauß-Seidel-Version besser, die Ordnung selbst jedoch identisch. In beiden

Fällen bedeutet aber die Implementierung einer Teilraumkorrekturmethode oder eines Mehrgitterverfahrens eine deutliche Verbesserung gegenüber einem Eingitterverfahren auf dem feinsten Level. Da in unserem Fall die Koeffizienten der Lösung bezüglich des Erzeugendensystems in einer Depth-First-Reihenfolge besucht werden, können nicht ohne weiteres alle Variablen eines Gitterlevels aktualisiert werden, bevor das nächste Level besucht wird. Da dies jedoch Voraussetzung für die Implementierung eines Block-Gauß-Seidel-Verfahrens auf den levelweise angeordneten Koeffizienten und damit eines multiplikativen Mehrgitterverfahrens ist, ist hier die Implementierung eines additiven Mehrgitterverfahrens naheliegend. Wie in Kapitel 6 gezeigt wird, kann damit bereits echte Mehrgitterkonvergenz erreicht werden. Unser Verfahren führt nur auf dem feinsten Level die Jacobi-Schritte durch. Somit genügt eine Diskretisierung für das feinste Level, auf den Grobleveln werden nur von den feineren Leveln transportierte Korrekturen aufgebracht. Da die Lösung im Erzeugendensystem dargestellt ist, müssen vor der Anwendung des diskreten Operators auf dem feinsten Level zunächst die Knotenwerte der Lösung u auf den Feingitterpunkten berechnet werden. Damit ergibt sich folgende rekursive Programmstruktur:

```
für alle neun Kindzellen der aktuellen Zelle {
     Addition der interpolierten Grobgitterwerte
          zu den Punktwerten der Lösung
     falls Kindzelle nicht weiter verfeinert: {
          Addition des Zellbeitrags des diskreten Operators
          zu den Residuen an allen vier Eckpunkten der Zelle
     }
     sonst {
          rekursiver Aufruf
     }
     Weitergabe des Zellbeitrags zu den Residuen der Eckpunkte
          der Zelle an Vaterzelle mit geeigneter Restriktionsvorschrift
     falls einer der Eckpunkte "fertig" ist: {
          Addition des Residuums mit geeignetem Relaxationsparameter
          zur bisherigen Lösung
     }
}
```

Dabei bedeutet „fertig“ dass der Punkt in der aktuellen Iteration nicht mehr benötigt wird, das heißt dass er von allen seinen Nachbarzellen verwendet wurde und sein Residuum vollständig akkumuliert ist.

Im Anschluss an diese grundsätzlichen Bemerkungen wollen wir nun die Details von Interpolation, Glättung und Restriktion, wie sie in unserem Programm verwirklicht wurden, aufzeigen. Da es dazu genügt, zwei aufeinander folgende Gitterlevel

zu betrachten, gehen wir ohne Einschränkung der Allgemeinheit von der in Abbildung 5.3 gezeigten Modellzelle mit den Eckpunkten $V0, V1, V2$ und $V3$ aus. Sie wird verfeinert zu neun Zellen mit den Knotenpunkten 0 bis 15.

Beim rekursiven Abstieg im Zellbaum[8] wird die vollständige Lösung $\mathbf{u_{ip}} = (u_{ip}^0, \ldots, u_{ip}^{15})^T$ auf den Feingitterpunkten mit Hilfe der Grobgitterpunkte nach der Vorschrift

$$\mathbf{u_{ip}} = \mathbf{u} + \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 2/3 & 0 & 1/3 & 0 \\ 1/3 & 0 & 2/3 & 0 \\ 0 & 0 & 2/3 & 1/3 \\ 0 & 0 & 1/3 & 2/3 \\ 2/3 & 1/3 & 0 & 0 \\ 1/3 & 2/3 & 0 & 0 \\ 0 & 2/3 & 0 & 1/3 \\ 0 & 1/3 & 0 & 2/3 \\ 4/9 & 2/9 & 2/9 & 1/9 \\ 2/9 & 1/9 & 4/9 & 2/9 \\ 2/9 & 4/9 & 1/9 & 2/9 \\ 1/9 & 2/9 & 2/9 & 4/9 \end{pmatrix} \cdot \begin{pmatrix} u_{ip}^{V0} \\ u_{ip}^{V1} \\ u_{ip}^{V2} \\ u_{ip}^{V3} \end{pmatrix} \tag{5.1}$$

interpoliert. Dies erfolgt im Programm allerdings nicht in dieser geschlossenen Form für alle 16 Punkte gleichzeitig, sondern zellweise entlang der Peano-Kurve jeweils bei der ersten Verwendung eines Punktes. Auch hier wird grundsätzlich die koordinatenunabhängige Bezeichnung der Punkte einer Zelle entlang der Kurve wie in Abschnitt 4.4 dargestellt verwendet. Obige Interpolation findet im Top-Down-Prozess des Programms auf jedem Level statt. Ein Punkt trägt somit auf dem Speicherplatz u_{ip} immer den bis zum aktuellen Level akkumulierten Wert, also den tatsächlichen Wert der Lösung am entsprechenden Knoten, der sich aus den Beiträgen aller Funktionen des hierarchischen Erzeugendensystems ergibt, deren Definitionsbereich den Punkt enthält. Der Speicherplatz u ist dagegen für den Koeffizienten im Erzeugendensystem reserviert.

Ist das Programm bei einer Zelle angelangt, die auf dem (lokal) feinsten Level liegt, wird auf dieser Zelle mit einem Jacobi-Schritt geglättet. In kurvenabhängiger Nummerierung der Lösungswerte $\mathbf{u_{ip}^{Zelle}} := (u_{ip}^0, \ldots, u_{ip}^3)^T$ auf den Zellknoten wird hier

[8] Im Gegensatz zur allgemein für Mehrgitterverfahren verwendeten Ausdrucksweise bedeutet Abstieg hier nicht den Übergang zum nächst gröberen Gitterlevel sondern bezieht sich auf die Baumstruktur, in der die groben Zellen „oberhalb“ der feinen angeordnet sind.

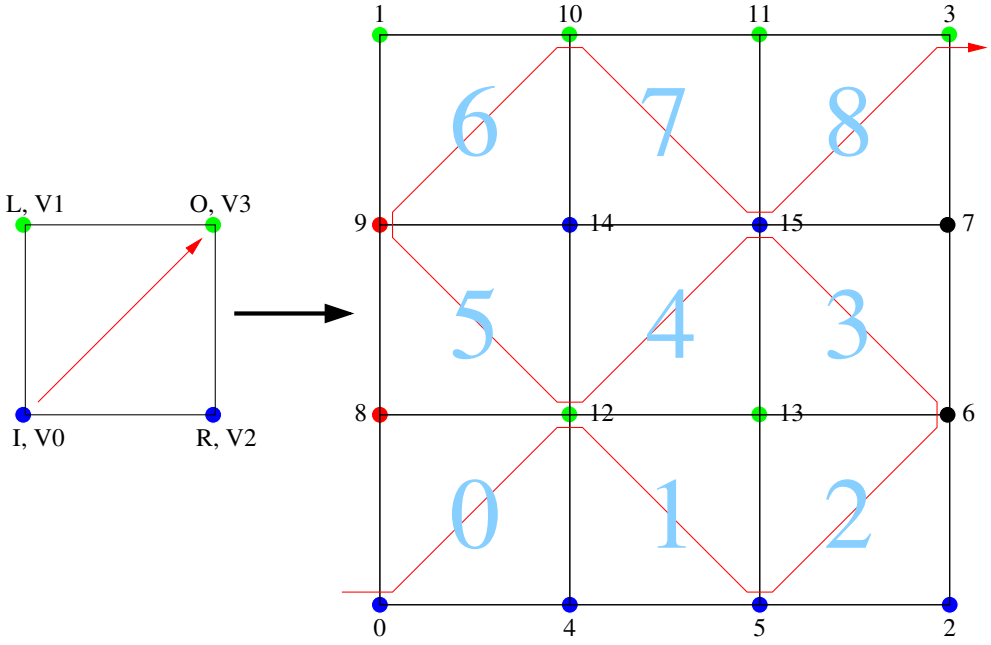

Abbildung 5.3: Nummerierungen für Interpolation und Restriktion.

mit

$$r_i^{lokal} = -h^2 \cdot b_i + \frac{1}{3} \cdot \underbrace{\begin{pmatrix} -2 & 1/2 & 1/2 & 1 \\ 1/2 & -2 & 1 & 1/2 \\ 1/2 & 1 & -2 & 1/2 \\ 1 & 1/2 & 1/2 & -2 \end{pmatrix}}_{=:M} \cdot \mathbf{u}_{\mathbf{ip}}^{\mathbf{Zelle}} \tag{5.2}$$

je Punkt ein lokales Residuum r_i^{lokal} erzeugt. Dabei ist b_i die rechte Seite der Differenzialgleichung ausgewertet für den i-ten Punkt und h die Gitterweite der Zelle. Die Matrix M ergibt sich aus der elementweisen Finite-Elemente-Diskretisierung des Laplace-Operators.

Dieses lokale Residuum wird zum einen für die spätere Korrektur des hierarchischen Wertes des Punktes akkumuliert, also

$$r_i = r_i + r_i^{lokal} \quad \text{für} \quad i = 0, \ldots, 3. \tag{5.3}$$

Sobald das Residuum r_i eines Punktes vollständig akkumuliert ist – für einen inneren Punkt ist das nach viermaliger Verwendung des Punktes der Fall –, wird der auf dem Punkt lebende Freiheitsgrad, also der entsprechende Koeffizient der Lösung im Erzeugendensystem nach

$$u_i = u_i + \lambda \cdot r_i \tag{5.4}$$

mit einem geeigneten Relaxationsparameter λ korrigiert. Im Programm findet dieser Vorgang dann statt, wenn ein Punkt auf den 2D-Ausgabestack geschrieben wird, weil dann feststeht, dass er in der laufenden Iteration nicht mehr benötigt wird und somit auch das Residuum nicht mehr verändert wird.

Außerdem wird mit

$$k_i = r_i^{lokal} \tag{5.5}$$

und der Restriktion

$$\begin{aligned} r_{Vj} &= r_{Vj} + A \cdot \mathbf{k} \quad \text{für} \quad j = 0, \ldots, 3 && (5.6)\\ k_{Vj} &= k_{Vj} + A \cdot \mathbf{k} \quad \text{für} \quad j = 0, \ldots, 3 && (5.7)\\ \text{mit} \quad \mathbf{k} &:= (k_0, \ldots, k_3)^T && (5.8) \end{aligned}$$

der lokale Einfluss der Zelle auf die Vaterzelle transportiert. Dabei ist A eine 4×4-Matrix, die durch Auswahl von vier Spalten aus der Transponierten der in 5.1 angegebenen Interpolationsmatrix entsteht, die die vier Punkte der aktuellen Feinzelle repräsentieren. Dies findet, wie man auch dem Codebeispiel von `iteration` entnehmen kann, auf allen Leveln vor dem Ablegen der Punkte einer Zelle auf die Stacks statt. Jede Vaterzelle „sammelt“ somit gewissermaßen die additiven Beiträge ihrer Kindzellen sofort auf. Auf den ersten Blick erscheint das Vorgehen, die lokalen Korrekturen mit r_i und k_i getrennt zu akkumulieren und zu transportieren, vielleicht etwas aufwändig, in der tatsächlichen Realisierung erweist sich diese Trennung jedoch als äußerst nützlich, weil dadurch die `restrict`-Routine einheitlich für alle Punkte auf allen Leveln und sogar unabhängig vom Punkttyp programmiert werden kann.[9]

Zu erwähnen ist an dieser Stelle noch, dass die Anpassung dieser Schritte auf andere Punkttypen recht einfach möglich ist. In unserem Programm geschieht dies dadurch, dass etwa ein homogener Dirichlet-Punkt zwar prinzipiell ein r_i^{lokal} erzeugen kann, danach aber algorithmisch verhindert wird, dass der Funktionswert dieses Punktes verändert werden kann, da er kein Freiheitsgrad ist. Bei der Interpolation wird für solche Punkte einfach $u_{ip} \equiv 0$ gesetzt, in der Restriktion wird für den Fall eines homogenen Dirichlet-Punktes die korrespondierende Spalte mit 0 besetzt.

[9] Würde man in jeder Zelle die akkumulierten Residuen r_0 bis r_3 jeweils an die Vaterzelle weitergeben, würde man nach dem oben beschriebenen einheitlichen Restriktionsverfahren für innere Zellen die Beiträge der entlang der Peano-Kurve ersten Nachbarzelle eines Punktes viermal, die der zweiten dreimal, der dritten zweimal und der vierten einmal an den Vater weitergeben. Dies würde zu einer asymmetrischen Gewichtung der Zellen und damit zu einem falschen Residuum in der Vaterzelle führen.

5.4 Implementierung eines Löser für die Stokes-Gleichung

Für die Lösung der Stokes-Gleichung (siehe Abschnitt 3.5) wird im Prinzip je eine Poisson-Gleichung für zwei Geschwindigkeitskomponenten u und v gelöst, bei der der zellbasierte Druck p in die rechte Seite eingeht. Es muss deshalb zuerst eine Speicherstruktur für zellbasierte Informationen (eben den Druck als eine Gleitpunktzahl pro Zelle) geschaffen werden. Dies ist wiederum bei der Verwendung je eines zusätzlichen Stacks für die Eingabe- und Ausgabewerte des Drucks je Zelle leicht möglich. Bei Verfeinerung einer Vaterzelle wird ihr Druck mit

$$p_{ip}^{i} = p^{i} + p_{ip}^{Vater} \quad \text{für} \quad i = 0, \ldots, 8 \tag{5.9}$$

konstant auf die neun Kindzellen interpoliert. Bei fester Nummerierung der Punkte wie in Abbildung 5.4 wird vermöge

$$dp = -\frac{1}{2 \cdot h}((u_{ip,1} + u_{ip,3} - u_{ip,0} - u_{ip,2}) + (v_{ip,0} + v_{ip,1} - v_{ip,2} - v_{ip,3})) \tag{5.10}$$

auf dem feinsten Level eine Druckkorrektur berechnet, die zum einen durch

$$p = p + dp \tag{5.11}$$

den Druck der Zelle selbst korrigiert und zum anderen durch

$$b_i := \frac{dp}{dx} \quad \text{für} \quad i = 0, \ldots, 3 \tag{5.12}$$

entsprechend (5.2) in die Korrektur der Geschwindigkeitswerte eingeht und somit auch auf alle höheren Level transportiert wird. Dieses Verfahren zur Druckkorrektur bei der Stokes-Gleichung wird auch als Uzawa-Verfahren bezeichnet (siehe zum Beispiel [6] und [7]).

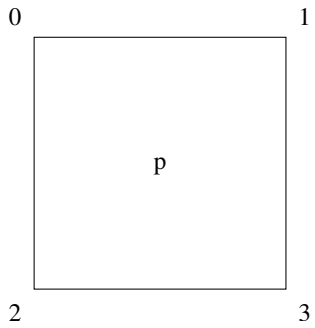

Abbildung 5.4: Feste Punktnummerierung für die Druckkorrektur der Stokes-Gleichung. Auf jedem der Punkte $0, \ldots, 3$ existieren zwei Geschwindigkeitskoeffizienten u_i und v_i sowie die interpolierten Werte $u_{ip,i}$ und $v_{ip,i}$.

6 Numerische Experimente und Cache-Effizienz

In diesem Kapitel stellen wir anhand einiger Beispielrechnungen das numerische Verhalten sowie die Cache-Effizienz unseres Programms dar. Für alle gerechneten Beispiele lässt sich eine gute auflösungsunabhängige Mehrgitterkonvergenz beobachten. Darüber hinaus können wir zeigen, dass die von uns verwendete neuartige Datenorganisation tatsächlich zu einem Programm führt, das allein aufgrund der Methodik ohne weitere Optimierungsstrategien (siehe zum Beispiel [15]) eine extrem hohe Effizienz im L2-Cache moderner Prozessoren erreicht. Dazu stellen wir zunächst einige grundlegende Informationen über die Cache-Hierarchien und Zugriffsalgorithmen moderner PC- und Serverprozessoren sowie einige Tools für Cache-Simulation und -analyse dar. Anschließend stellen wir eine zusätzliche Möglichkeit zur Einsparung von Speicherbedarf durch eine sparsame Repräsentation der Ergebnisse im 2D-Ein- und Ausgabestack vor. Danach zeigen wir anhand der Poisson- und der Stokes-Gleichung auf verschiedenen Geometrien die konstant hohe Effizienz sowohl für reguläre als auch für adaptive Gitter unseres Programms. Der Begriff Effizienz gliedert sich dabei in numerische Effizienz (im Wesentlichen Konvergenzgeschwindigkeit) und Cache-Effizienz bezüglich des L2-Caches.

6.1 Caches und Analysewerkzeuge

In diesem Abschnitt geben wir zuerst einen Überblick über die Cache-Hierarchien moderner Prozessoren und die Auswirkungen des Speicherzugriffs auf die Performance numerischer Programme. Danach stellen wir Werkzeuge zur Simulation und Analyse des Verhaltens von Programmen im Bezug auf ihre Speicherzugriffe vor, um schließlich einige Hinweise auf Performanceunterschiede zu geben, die sich allein durch die Verwendung unterschiedlicher Compiler und Compileroptionen ergeben.

6.1.1 Cache-Hierarchien

Moderne Prozessoren implementieren hierarchische Speicherstrukturen, um den immer größer werdenden Unterschied zwischen der CPU-Geschwindigkeit und der

Performance des Hauptspeichers – beschrieben durch Latenz und Bandbreite der Zugriffsschnittstelle – auszugleichen. Dafür gibt es typischerweise an der Spitze dieser Hierarchie einen kleinen und sehr teuren Speicherbereich auf dem Chip des Prozessors, der Daten mit sehr niedriger Latenz und hoher Bandbreite an die CPU übergibt, zum Beispiel die Prozessorregister. Daran anschließende Speicherhierarchien sind gekennzeichnet durch steigende Latenz, sinkende Bandbreite aber größer werdende Speichermenge, je weiter man sich vom Prozessor entfernt. Die größte Speichermenge bietet der Hauptspeicher – zum Beispiel bis zu 4 GByte RAM bei Standard-Workstation-Architekturen–, der zugleich aber auch im Zugriff am langsamsten ist. Zwischen den Prozessorregistern und dem Hauptspeicher sind üblicherweise zwei (zum Beispiel bei Intel Xeon oder Pentium IV) oder drei (zum Beispiel bei Intel Itanium2) Speicherhierarchien angeordnet, die als Level-1-, Level-2- und Level-3-Caches oder kurz L1-, L2- und L3-Caches bezeichnet werden. Diese Caches tragen ebenso wie die Prozessorregister Kopien von Daten des Hauptspeichers, ihre Existenz ist aber vor der Logik der Maschinensprache verborgen. Das heißt, dass ein Programm auf eine Speicheradresse des Hauptspeichers zugreift, ohne vorher zu wissen, ob eine Kopie der unter dieser Adresse gespeicherten Daten in einem der Caches liegt oder nicht. Das Programm ist also insbesondere nicht in der Lage, gezielt benötigte Daten im Voraus aus dem Hauptspeicher in den Cache zu laden, um eine Verlangsamung des Programms aufgrund der relativ langen Latenzzeit des Hauptspeichers zu vermeiden. Es ist deshalb vor allem Aufgabe des Programmentwicklers, durch optimierte Gestaltung seiner Programme dafür zu sorgen, dass möglichst oft innerhalb eines Programmlaufs die benötigten Daten bereits in einem der Caches liegen. Die Notwendigkeit dieser Optimierung wird sofort aus den im Folgenden dargestellten gravierenden Performanceunterschieden der Cache-Level untereinander sowie des Caches und des Hauptspeichers klar.[1]

Der L1-Cache wird üblicherweise in zwei getrennte Bereiche aufgeteilt, den Daten- und den Instruktionsbereich. Die Latenz der On-Chip-L1-Caches ist typischerweise ein bis zwei CPU-Zyklen, das heißt die CPU muss nach einer Anforderung von Daten, die im L1-Cache liegen, ein bis zwei Zyklen warten, bevor die Daten in die Register geladen sind und verarbeitet werden können. Auf Grund von physikalischen Beschränkungen bezüglich der Laufzeit von Signalen in sehr hoch getakteten modernen Prozessoren muss die Größe des L1-Caches zur Zeit auf 64 KByte oder sogar weniger begrenzt werden.

Unter dem L1-Cache liegt der L2-Cache, der gewissermaßen als Zugriffsbackup für den L1-Cache dient. Er wird heutzutage in der Regel auch als On-Chip-Cache im-

[1] Genau genommen muss man den Hintergrundspeicher, also zum Beispiel Festplatten oder Netzwerkdatenspeicher, auch als Bestandteil der Speicherhierarchie ansehen. Dessen Zugriffsperformance ist jedoch im Vergleich zu den oberen Leveln der Speicherhierarchie so niedrig (zum Beispiel ist der Zugriff auf ein einzelnes Datum auf einer Festplatte bis zu 1000 Mal langsamer als der Zugriff auf den Hauptspeicher), dass er beim Ablauf numerischer Programme außer bei Start und Initialisierung im Prinzip keine Rolle spielen darf.

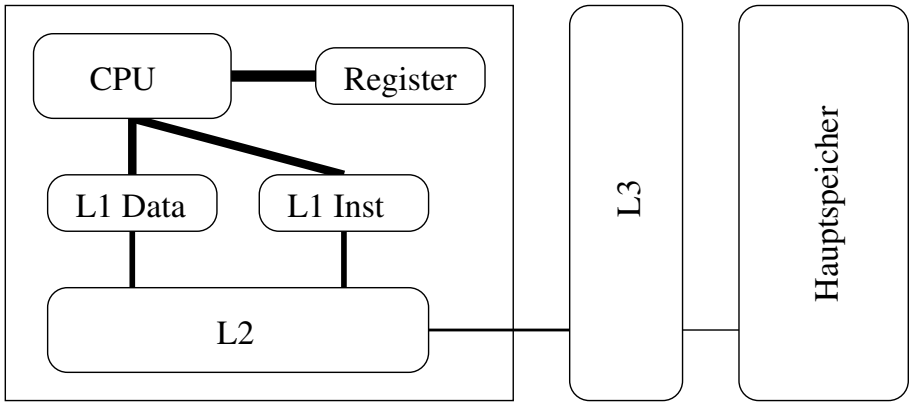

Abbildung 6.1: Speicherhierarchie moderner Mikroprozessoren. Die Strichstärke der Verbindungslinien der Hierarchielevel beschreibt die Bandbreite des Speicherzugriffs der Level untereinander.

plementiert und ist 256 KByte bis 512 KByte groß. Er stellt Daten typischerweise innerhalb von fünf bis zehn Zyklen zur Verfügung. Bei sehr teuren Hochleistungsprozessoren wie etwa dem Intel Itanium2 wird noch ein L3-Cache mit einer Größe zwischen 1 MByte und 16 MByte – in der Regel als Off-Chip-Cache – realisiert, der bei Zugriff circa 10 bis 20 Wartezyklen verursacht. Der darunter liegende Hauptspeicher des Systems besitzt nochmals mindestens um einen Faktor 10 bis 50 höhere Latenzen als der L3-Cache. In Abbildung 6.1 ist die Speicherhierarchie zusammenfassend dargestellt, die Dicke der Verbindungslinien der Level soll einen Hinweis auf die Bandbreite und die Latenz der Hierarchiestufen untereinander geben.

Aus obiger Darstellung der Zugriffsgeschwindigkeiten wird schnell deutlich, dass ein Speicherzugriff immer teurer – im Sinne von zu erwartenden Latenzzyklen des Prozessors – wird, je tiefer in der Hierarchie er erfolgt. Die Aufgabe des Programmierers ist es also, die Anzahl von so genannten cache misses auf den oberen Cache-Leveln – das sind Zugriffe auf den L1- oder L2-Cache, bei denen die gerade benötigten Daten nicht als Speicherkopie im Cache vorliegen – zu minimieren beziehungsweise das Verhältnis von erfolgreichen Cachezugriffen – so genannten cache hits – zu den cache misses möglichst günstig zu gestalten.[2]

[2]In der Realität moderner Prozessoren ist dieser Zusammenhang zwar richtig, es ist jedoch auf Grund anderer Designkonzepte wie Pipelining, Prefetching und Umordnung der Ausführung von Instruktionen im Allgemeinen sehr schwierig respektive unmöglich, genau zu quantifizieren, wie teuer es ist, wenn die gerade benötigten Daten nicht im L1- oder L2-Cache liegen, da der Prozessor oft Operationen vorziehen kann, die diese Daten nicht benötigen. Insbesondere sind deshalb die Kosten eines L1- oder L2-Cache-Misses programm- und situationsspezifisch, das heißt sie variieren zwischen verschiedenen Programmen und sogar innerhalb eines Programms je nach Zustand und weiterem Fortgang.

6.1.2 Optimalitätsforderungen

Innerhalb der Caches sind die Daten in so genannten cache lines gespeichert. Eine cache line enthält die Kopie eines Speicherblocks des Hauptspeichers, sie stellt gewissermaßen die Containergröße für den Transfer von Daten zwischen dem Hauptspeicher und den Cache-Leveln dar. Wird auf ein Datum eines Speicherblocks zugegriffen, der in einer cache line liegt, spricht man von einem cache hit. Findet sich das benötigte Datum jedoch nicht in einer der cache lines, so ereignet sich ein cache miss. Daraufhin wird der entsprechende Speicherblock des Hauptspeichers in eine cache line geladen und der Prozessor kann zugreifen. Dafür muss aber zwangsläufig – da man allein schon wegen des auf einem Rechner laufenden Betriebssystem mit einem zum Beispiel bei Linux mehrere Megabyte großen Betriebssystemkerns davon ausgehen muss, dass die cache lines immer gefüllt sind – ein anderer Speicherblock aus einer cache line verdrängt werden, um Platz für den neuen Speicherblock zu schaffen. Da man möglichst vermeiden möchte, dass ein Speicherblock verdrängt wird, der in Kürze wieder vom Programm benötigt wird, ergibt sich die Notwendigkeit, eine Strategie für solche Verdrängungen zu entwickeln. Diese Arbeit konzentriert sich dabei auf die optimale Implementierung eines numerischen Algorithmus, also auf die Gestaltung der Software. Dazu muss jedoch zunächst die Implementierung der Verdrängung von Speicherblöcken auf Seiten der Hardware bekannt sein.

Hier spielt zum Einen die Anzahl möglicher cache lines, in die ein Speicherblock transferiert werden kann, eine Rolle, man spricht hier von der *Assoziativität* des Caches. Bei einer n-fach assoziativen Cache-Organisation stehen für einen Speicherblock aus dem Hauptspeicher n verschiedene cache lines zur Auswahl. Im Fall $n = 1$ spricht man von einem *direct mapped cache*, wenn n gleich der Anzahl der cache lines ist von einem *voll-assoziativen* Cache. Je größer n für einen Cache ist, desto komplizierter und damit auch teurer ist seine Hardwareimplementierung, desto größer werden aber auch die so genannten *hit rates*, also das Verhältnis zwischen cache hits und cache misses.

Zum Anderen wird eine Cache-Implementierung – bei mindestens zweifach-assoziativen Caches – von der Strategie bestimmt, nach der entschieden wird, welcher in einer der zur Auswahl stehenden cache lines residierende Speicherblock verdrängt wird. Am häufigsten werden hier die Strategien „zufällig“ oder „letzte Verwendung am weitesten zurückliegend“ (least recently used, LRU) verwendet. Bei Ersterer wird eine cache line zufällig ausgewählt, deren Speicherblock verdrängt wird. Bei Zweiterer wird der Block verdrängt, dessen letzte Benutzung am weitesten zurückliegt, der also bezüglich seiner Benutzung durch das Programm „am ältesten“ ist. Weniger häufig finden Strategien wie „first in, first out (FIFO)“ oder „least frequently used (LFU)“ Verwendung.
Die *optimale* Verdrängungsstrategie würde den Speicherblock ersetzen, der im weiteren Programmablauf als letzter wiederverwendet wird. Diese Strategie (von nur

theoretischem Interesse) lässt sich aber für reale Caches nicht implementieren, da man dazu Informationen über die Zukunft des Programmablaufs zur Verfügung haben müsste. Man kann aber zeigen, dass ein voll-assoziativer Cache mit dieser optimalen Verdrängungsstrategie die geringste Zahl an cache misses unter allen Caches gleicher Gesamtgröße produzieren würde [25].

Vor diesem Hintergrund der Unmöglichkeit einer Hardwareimplementierung der optimalen Verdrängungsstrategie wird das Optimierungspotenzial auf Seiten der Software deutlich. Es ergeben sich recht natürlich folgende Forderungen an die Abfolge der Zugriffe, die ein Algorithmus auf seine Nutzdaten im Arbeitsspeicher durchführt:

- *Zeitliche Lokalität*
- *Örtliche Lokalität*

Dabei bedeutet *zeitliche Lokalität*, dass für ein Datum, das mehrfach benötigt wird, alle Zugriffe möglichst zeitnah erfolgen sollten. Dies führt mit hoher Wahrscheinlichkeit dazu, dass der Speicherblock, der das Datum trägt, zwischen den Verwendungen nicht aus „seiner“ cache line verdrängt wird. *Örtliche Lokalität* ist die Forderung nach der zeitnahen Verwendung von im Hauptspeicher zum aktuellen Datum benachbarter Daten. Der Sinn dieser Maßnahme liegt in der Tatsache, dass immer ganze Speicherblöcke in cache lines geladen werden. Befindet sich ein zeitnah benötigtes Datum in der örtlichen Umgebung des aktuell verwendeten Datums, so ist es mit hoher Wahrscheinlichkeit bereits im Speicherblock des aktuellen Datums enthalten und somit bereits in eine cache line geladen worden. Der Zugriff verursacht also keinen zusätzlichen cache miss.

Die oben beschriebenen Lokalitätsforderungen beeinflussen im Wesentlichen nur die Häufigkeit von cache misses bei der mehrfachen Verwendung eines Datums. Die Cache-Effizienz bei der ersten Verwendung eines Datums wird dagegen kaum verbessert. Hier hilft eine Technik, die als *prefetching* bezeichnet wird. Vereinfacht ausgedrückt bedeutet prefetching, dass ein Datum (mit seinem Speicherblock) in eine cache line geladen wird, obwohl es im Moment (noch) nicht benötigt wird (und damit auch nicht zum Prozessor zur Verarbeitung weitergegeben wird). Dies kann wiederum entweder durch die Software geschehen (Einfügen von Prefetch-Anweisungen in den Code durch den Programmierer oder den Compiler), wobei entweder der Programmierer oder der Compiler wissen muss, wann ein prefetching Sinn macht, oder durch so genanntes *hardware-based prefetching*. Im zweiten Fall werden zum Beispiel zusätzlich zum aktuellen Speicherblock n die Speicherblöcke $n \pm 1, n \pm 2, \ldots, n \pm i$ geladen. Man spricht hier von *sequential prefetching*. Typischerweise sind in modernen Prozessoren aber nur 1-Schritt-Prefetching-Strategien mit Richtungserkennung implementiert, das heißt es wird in der Richtung, in der die Speicherblöcke gerade

abgearbeitet werden, der nächste Block zusätzlich (spekulativ) geladen. Wenn man nun durch geeignete Softwareimplementierung des auszuführenden Algorithmus erreichen kann, dass der spekulativ geladene Speicherblock gerade die als nächstes benötigten Daten mit hoher Wahrscheinlichkeit enthält, so lässt sich das hardware-based prefetching sehr gezielt zur Reduzierung der durch das erste Laden der Daten verursachten cache misses ausnutzen [15].

6.1.3 Werkzeuge für Simulation und Hardwareanalyse

Wie oben bereits angedeutet, ist es durchaus kompliziert, das Verhalten eines Programms bezogen auf seine Speichereffizienz vorherzusagen, unter anderem weil die tatsächlichen Kosten von cache misses schwer zu quantifizieren sind. Es existieren jedoch Werkzeuge, die das Cacheverhalten eines Programms simulieren oder über prozessorintegrierte Zähler die tatsächliche Performance messen können.

Im Entwicklungsprozess der Programme dieser Arbeit wurden aus der Vielzahl von Werkzeugen, die für Simulation und Analyse der Speichereffizienz von Algorithmen zur Verfügung stehen, die Tools *cachegrind* [22] und *perfmon* [31] beziehungsweise *hpcmon* ausgewählt.

6.1.3.1 Detaillierte Speichersimulation mit cachegrind

Mit Hilfe von cachegrind wird der Ablauf eines Programms mit seinen Speicherzugriffen simuliert. Dabei wird der Programmlauf vollständig auf einem Software-Emulator durchgeführt. Dieser Emulator lässt sich bezüglich relevanter Parameter wie beispielsweise der Größe der cache lines oder der Assoziativität des Cache-Modells individuell einstellen. Es werden alle Arten von Speicherzugriffen, also auch cache hits und cache misses, auf alle Level der Speicherhierarchie aufgezeichnet. Zusätzlich werden die Kosten des Programmlaufs abgeschätzt und in CPU-Zyklen ausgedrückt. Dazu wird die Näherungsformel

$$\sharp\text{CPU-Zyklen} = \text{Instruktionen} + 10 \cdot \text{L1-Misses} + 100 \cdot \text{L2-Misses}$$

verwendet. Dies ist eine sehr grobe Schätzung, für die sich gezeigt hat, dass sie für reale Programmläufe zu negativ ist [24]. Es ist im Allgemeinen nicht zu erwarten, dass der Prozessor beim Auftreten eines L2-Misses tatsächlich 100 Zyklen „steht“, deshalb werden die benötigten CPU-Zyklen von cachegrind zu hoch angesetzt. Mit dem Programm Kcachegrind [26] von Josef Weidendorfer lassen sich die umfangreichen Daten, die von einer Cachegrind-Simulation erzeugt werden, griffig visualisieren und analysieren. Es besteht zum Beispiel die Möglichkeit, die cache misses graphisch in den Aufrufbaum eines Programms zu integrieren oder quellcodebezogen zu quantifizieren, das heißt die cache misses den Zeilen des Quelltextes des

Programms zuzuordnen. Dies ist äußerst praktisch für die detaillierte Performance-Analyse, weil dadurch deutlich wird, wo im Programm mögliche Optimierungspotenziale stecken respektive welche Programmteile bezüglich der Speicherzugriffe besonders teuer sind. Die Simulation des Cache-Verhaltens durch cachegrind hat sich im Rahmen unserer Untersuchungen als sehr zuverlässig herausgestellt, die durch die Simulation vorhergesagten cache hit rates stimmen sehr gut mit den gemessenen Werten überein [24]. Die Simulation des Cache-Verhaltens bietet den Vorteil einer sehr feingranularen Zuordnung der Ereignisse (zum Beispiel misses oder hits) auf den Speicherstrukturen zur Programmierung der Algorithmen und der damit verbundenen Möglichkeit zur Optimierung. Als Nachteil ist jedoch anzuführen, dass die Laufzeit eines Programms erheblich verlängert wird. Ein Faktor 50 gegenüber einem „normalen“ Durchlauf ist unser Erfahrungswert.

6.1.3.2 Messung der Gesamtperformance mit perfmon und hpcmon

Versucht man die tatsächliche Speicherperformance eines Programms auf einem realen Rechner allein an der Laufzeit festzumachen, ist dies insofern problematisch, als ein Laufzeitgewinn etwa nach einer Änderung im Programm nur einen *relativen* Hinweis auf Verbesserungen liefert. Es ist auf diese Weise nicht möglich, eine *absolute* Aussage über die Cache-Performance des Programms zu treffen. Eine zusätzliche Schwierigkeit ergibt sich aus der Tatsache, dass die Messergebnisse durch den momentanen Zustand und die Auslastung des Systems, auf dem das Programm läuft, „verschmutzt“ werden. Moderne Prozessoren enthalten jedoch spezielle Register, um verschiedene Ereignisse wie etwa cache misses, cache hits oder Prozessor-Zyklen zu zählen. Diese Register heißen *hardware performance counter*, die Informationen, die sie tragen, und der Zugriff darauf ist von Architektur zu Architektur verschieden. Mit Werkzeugen wie *perfmon* [31] von Hewlett Packard und dem darauf aufsetzenden *hpcmon* von Intel lassen sich diese Zähler auswerten, was sogar prozess-spezifisch möglich ist. Somit lassen sich zuverlässige Zahlen (zum Beispiel der cache misses) im Wesentlichen unabhängig von der momentanen Gesamtlast des Rechners gewinnen. Abbildung 6.2 zeigt die Bildschirmausgabe des Programms *hpcmon* für Intel Itanium Architekturen. Vergleichbare Werkzeuge existieren auch für IA32-Architekturen (Intel Pentium III oder IV, Intel Xeon), zum Beispiel *perfex*.

Der Vorteil dieser Tools gegenüber Simulatoren wie cachegrind ist die schnelle Verfügbarkeit von Ergebnissen. Es genügen drei (perfex) beziehungsweise sechs (hpcmon) Durchläufe des zu analysierenden Programms, um Ergebnisse wie in Abbildung 6.2 zu erhalten. Es ist jedoch nicht möglich, einen Bezug zum Programmtext herzustellen. Somit sind diese Hilfsprogramme sehr gut geeignet, die tatsächliche und absolute Performance des Programms bezüglich einiger Parameter zu messen, sie sind jedoch kaum geeignet, Flaschenhälse aufzuspüren und Optimierungspotenziale aufzuzeigen.

```
================================================================================
HPCmon Ver. 1.2 - (c) 2003 Gernot Hoyler, Intel GmbH
[pfmon Ver. 1.2 - (c) 2001-2003 Hewlett-Packard Company]
================================================================================
CPU speed             :   1296555004 Hz
Elapsed time          :   4978117272 cycles (3.84 sec)
Monitored clockticks  :   4964659508 cycles (3.83 CPU sec)
Instructions retired  :  11105760514 events (2892.51 Mev/sec, 2.24 IPC)
Nops retired          :   3149522217 events (820.30 Mev/sec, 0.63 NPC)
Off predicated instr. :   1278935059 events (333.10 Mev/sec, 0.26 NPC)
FP instr. retired     :   1782067294 events (464.14 Mev/sec or MFLOPS)
FP results FTZ'ed     :            0 events (0.00 Mev/sec)
L2 references         :   2059609339 events (536.43 Mev/sec)
L2 misses             :      1371249 events (0.36 Mev/sec, 99 % hit rate)
L3 references         :      1757720 events (0.46 Mev/sec)
L3 misses             :        69806 events (0.02 Mev/sec, 96 % hit rate)
Bus data cycles       :       678114 events (0.18 Mev/sec, 0 % bus load)
L2 DTLB misses        :         6435 events (0.00 Mev/sec)
L2 ITLB misses        :          526 events (0.00 Mev/sec)
L2 bank conflicts     :     92305425 events (24.04 Mev/sec)
Misaligned loads      :            0 events (0.00 Mev/sec)
Stores to shared line :         1550 events (0.00 Mev/sec)
Branches retired      :    489015654 events (127.36 Mev/sec)
Mispredicted branches :     25657401 events (0.00 Mev/sec, 94 % hit rate)
Back end bubbles      :   2049542824 events (533.81 Mev/sec)
FR load bubbles       :    706983930 events (184.13 Mev/sec)
GR load bubbles       :    225579186 events (58.75 Mev/sec)
FPU bubbles (except.) :            3 events (0.00 Mev/sec)
```

Abbildung 6.2: Ausgabe von *hpcmon* für einen Testlauf unseres Programms auf einem Intel Itanium basierten System.

Wir haben beide oben aufgeführte Klassen von Werkzeugen erfolgreich benutzt, wie in den folgenden Abschnitten dargestellt wird. Dabei ergibt sich zum einen, dass die absolute Speicherperformance unserer Programme im L2-Cache herausragend gut ist, zum anderen stellt sich heraus, dass die L2-Cache-Misses in ihrer Zahl auf das unserer Ansicht nach unabdingbar Notwendige beschränkt sind und genau dort passieren, wo sie unvermeidlich sind.

6.1.4 Vergleich von Compilern und Optionen

Eine weitere interessante Fragestellung im Zusammenhang mit der erzielbaren Speicher- und Gesamt-Performance eines Programms ist der Einfluss verschiedener Compiler und deren Optionen auf einer Testarchitektur. So können unterschiedlich aggressive Optimierungsoptionen zu deutlich unterschiedlichen Laufzeitergebnissen eines Programms führen, leider in beide Richtungen. Eine zu aggressive Optimierung kann in seltenen Fällen die Laufzeit verlängern oder sogar die numerischen Ergebnisse (zum Beispiel durch Umordnung der Reihenfolge der Ausführung von Gleitpunktoperationen) verändern.

Wir haben diese Einflüsse unter anderem im Rahmen der Arbeit von Reinhard Stein [24] untersucht. Dabei stellt sich heraus, dass im Falle unserer Programme zwar die Laufzeit verkürzt werden kann, jedoch ohne dass sich die Cache-Performance ändert. So hat zum Beispiel der Vergleich des GNU `C`-Compilers mit dem Intel Native-IA32-Compiler für das Programm zur Lösung der Poisson-Gleichung ergeben, dass in beiden Fällen die L2-Hit-Rate bei über 99,0% lag, mit dem Intel Compiler aber die Laufzeit niedriger und die MFLOPS-Rate (Millionen Gleitpunktoperationen pro Sekunde) höher war. Der Grund für die Performance-Steigerung ist in der von uns verwendeten Compiler-Option `-xW` zu suchen. Sie sorgt unter anderem dafür, dass möglichst viele Gleitpunktoperationen in den so genannten SSE-Einheiten der IA32-Prozessoren ausgeführt werden können. Dies sind eigentlich für Multimedia-Anwendungen optimierte Bereiche der IA32-Architektur, die bestimmte Gleitpunktoperationen besonders effizient ausführen können [32].

Die Tatsache, dass die Cache-Performance unserer Programme also sehr robust gegen unterschiedliche Compiler (auch auf unterschiedlichen Plattformen) ist, betrachten wir als ein ermutigendes Ergebnis, insbesondere da das Performance-Niveau absolut gesehen sehr hoch ist (Details dazu finden sich in den Abschnitten 6.3 und 6.4). Es hat sich aber auch gezeigt, dass ein gewisses Fein-Tuning an den Optionen und die Verwendung architektur-spezifischer Compiler große Vorteile bezüglich der Gesamtleistung unserer Programme – gemessen in MFLOPS und Programmlaufzeit – bringt.

6.2 Ergebnisrepräsentation im 2D-Stack

In diesem Abschnitt zeigen wir anhand einiger grundsätzlicher Betrachtungen über erzielbare Rechengenauigkeiten bei der iterativen Lösung der betrachteten Partiellen Differenzialgleichungen, dass es im Falle der von uns verwendeten Ansätze genügt, die Lösung beziehungsweise die Koeffizienten ihrer Darstellung auf dem hierarchischen Erzeugendensystem mit einfach genauen Gleitpunktzahlen zu repräsentieren. Dies bringt deutliche Vorteile hinsichtlich des Speicherbedarfs, den das Programm auf einem Rechner zur Laufzeit benötigt.

Aus der Finite-Elemente-Diskretisierung ergibt sich letztlich die Aufgabe, ein Gleichungssystem der Form

$$Au = b \tag{6.1}$$

zu lösen. In einem iterativen Verfahren, wie es von uns verwendet wird, ist in jeder Iteration das Residuum

$$r := A(\hat{u} - u) \tag{6.2}$$

zu verkleinern, das als Maßstab für die Differenz zwischen der Näherungslösung $\hat{u}$ und der tatsächlichen Lösung u verwendet wird. Mit geeigneten Normen lässt sich die Genauigkeit der Näherungslösung durch

$$\|\hat{u} - u\| \leq \|A^{-1}\| \cdot \|r\| \tag{6.3}$$

abschätzen. Wenn A symmetrisch ist, was bei unseren Beispielrechnungen für die aus der FE-Diskretisierung entstehende Systemmatrix A gegeben ist, lässt sich $\|A^{-1}\|$ durch den inversen Wert des kleinsten Eigenwertes $\lambda_{min}(A)$ von A abschätzen und obige Gleichung wird zu

$$\|\hat{u} - u\| \leq \frac{1}{\lambda_{min}(A)} \cdot \|r\|. \tag{6.4}$$

Für nodale Basen ist der kleinste Eigenwert von A proportional zum Quadrat der Gitterweite h, also

$$\lambda_{min}(A) \sim c \cdot h^2 \tag{6.5}$$

mit einer Konstante c. Die Abschätzung der Genauigkeit der Lösung ergibt sich dann zu

$$\|\hat{u} - u\| \leq \frac{1}{c \cdot h^2} \cdot \|r\|. \tag{6.6}$$

Dies hat zur Konsequenz, dass die erzielbare Genauigkeit der Lösung von der gewählten Gitterweite h und der Genauigkeit des Residuums r abhängt. Ein Beispiel macht dies deutlich: Sei $c \approx 1$. Soll die Genauigkeit der Lösung sechs Stellen sein, also $\|\hat{u} - u\| \leq 10^{-6}$, so muss bei einer Gitterweite $h = 10^{-3}$ das Residuum auf mindestens zwölf Stellen genau berechnet werden, weil nur dann die rechte Seite der Abschätzung (6.6) mindestens von der Ordnung $10^6 \cdot 10^{-12} = 10^{-6}$ ist. Dar-

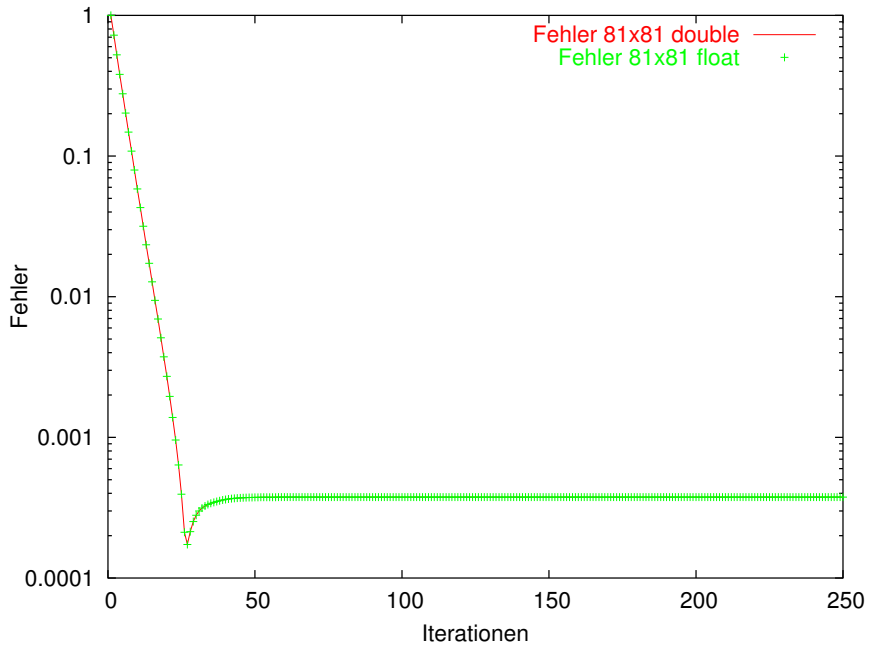

Abbildung 6.3: Entwicklung des Fehlers in der Lösung der Poisson-Gleichung über die Löseriterationen bei einem Gitter mit 81x81 Zellen.

aus ergibt sich die Forderung, alle Operationen für die Berechnung des Residuums mit wesentlich höherer Genauigkeit durchzuführen, wobei doppelte Genauigkeit im Allgemeinen ausreicht. Anschaulich gesagt bedeutet das aber auch, dass von einem hoch genau berechneten Residuum nur die Stellen für die Korrektur der Werte von $\hat{u}$ verwendet werden können, die nicht durch die Beschränkung vernichtet werden, die durch den Einfluss von h^2 entsteht. In unserem Beispiel kann man also davon ausgehen, dass vermutlich nur noch die fünfte und sechste Stelle des Residuums eine brauchbare Korrektur auf $\hat{u}$ erzeugen.

Griebel hat in [11] gezeigt, dass für den Fall der Verwendung eines hierarchischen Erzeugendensystems für den kleinsten Eigenwert der (jetzt singulären) Systemmatrix A

$$\lambda_{min}(A) \geq \hat{c} \tag{6.7}$$

mit einer Konstante $\hat{c}$ gilt. Der kleinste Eigenwert ist also nicht mehr abhängig von der Gitterweite h. Berechnet man jetzt wieder das Residuum mit doppelter Genauigkeit, so gehen von den hinteren Stellen des Residuums deutlich weniger im Sinne der erzielbaren Genauigkeit durch Auslöschung verloren, da der Einfluss von h^2 fehlt. Trotzdem ist es nicht sinnvoll, die Lösung genauer als in der Größenordnung der lokal feinsten Gitterweite zu korrigieren, weil bereits der Diskretisierungsfehler der verwendeten FE-Methode Gitterfehlers nicht unter die Größenordnung $O(h^2)$ gedrückt werden kann. Bei unseren Programmen ist die feinste Gitterweite ein ne-

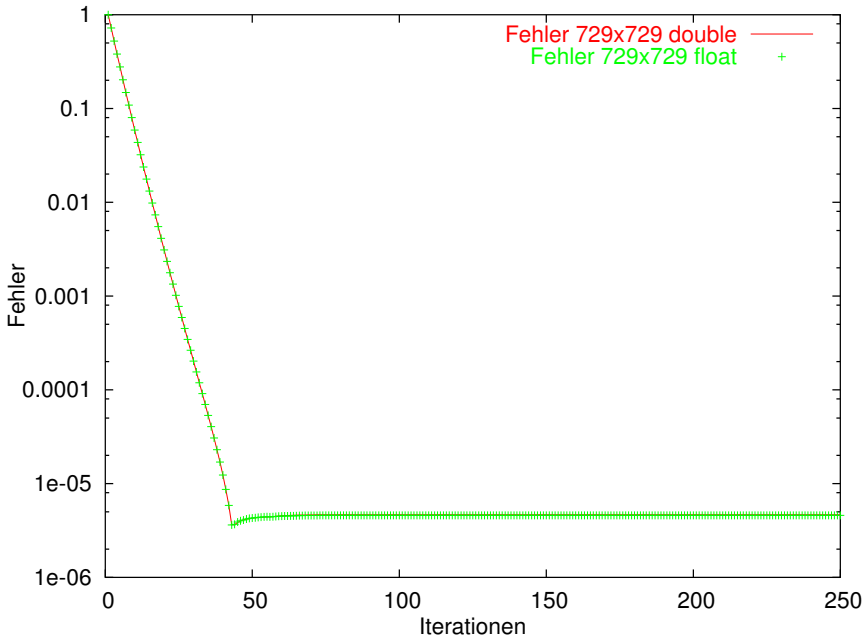

Abbildung 6.4: Entwicklung des Fehlers in der Lösung der Poisson-Gleichung über die Löseriterationen bei einem Gitter mit 729x729 Zellen.

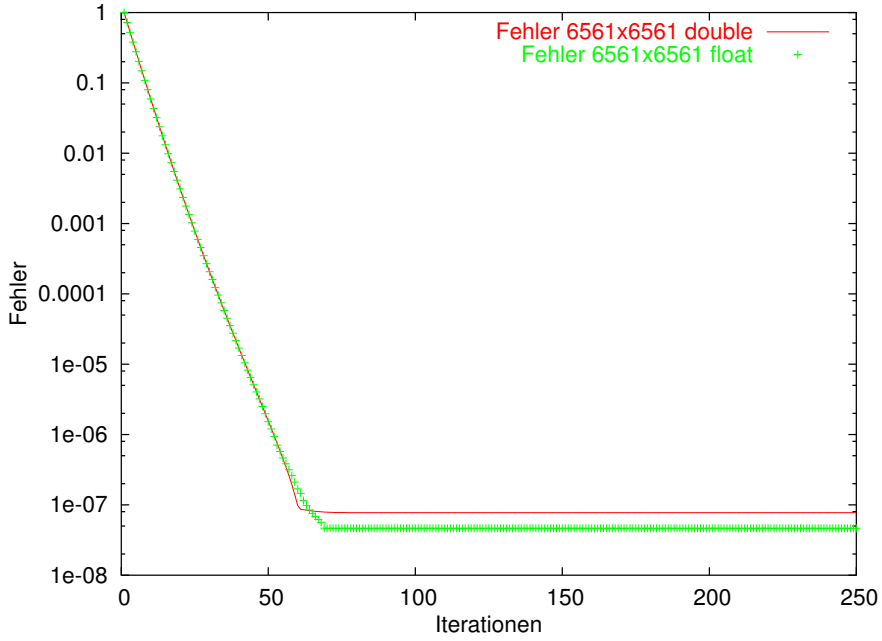

Abbildung 6.5: Entwicklung des Fehlers in der Lösung der Poisson-Gleichung über die Löseriterationen bei einem Gitter mit 6561x6561 Zellen.

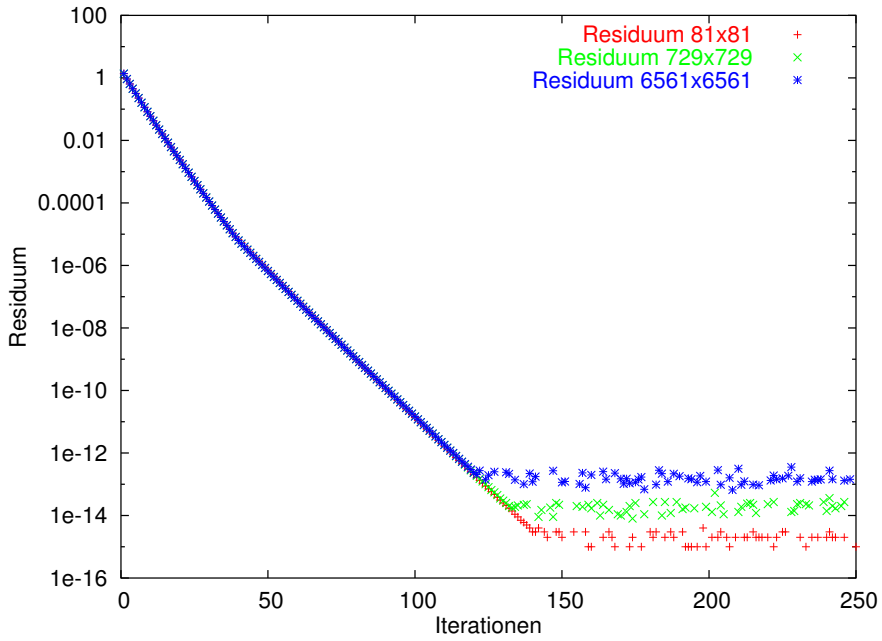

Abbildung 6.6: Entwicklung des Residuums der Poisson-Gleichung über die Löseriterationen für unterschiedliche Gitterauflösungen bei doppelt genauer Repräsentation der Lösung.

gative Potenz von 3. Da erst für $3^{-11} = 5.6 \cdot 10^{-6}$ der Diskretisierungsfehler kleiner als die Genauigkeit einer einfach genauen Gleitpunktzahl werden kann, ist es naheliegend, zumindest auf den 2D-Stacks die Koeffizienten der Lösung nur mit `float`-Zahlen zu speichern.

Im Inneren der numerischen Routinen und auf den 0D- und 1D-Stacks wird dagegen alles doppelt genau gespeichert und gerechnet. Dies ist notwendig, weil das Residuum zum Beispiel eines inneren Punktes in vier Schritten aus vergleichsweise großen Zahlen akkumuliert wird und dabei Auslöschungseffekte auftreten. Mit doppelt genauer Rechnung gelingt es aber, mindestens die fünfte und sechste Stelle des Residuums genau zu bekommen und damit die Näherungslösung zu korrigieren.

Wir haben diese Idee auf einigen regulären Gittern für die Lösung der Poisson-Gleichung auf dem Einheitsquadrat (siehe Abschnitt 6.3) getestet. In diesem Fall ist die exakte Lösung bekannt (siehe Abschnitt 3.4) und man kann den tatsächlichen Fehler der Näherungslösung berechnen. In den Abbildungen 6.3, 6.4 und 6.5 ist für unterschiedliche Gitterweiten für jede Iteration der Fehler in der Maximumsnorm bei einfach genauer und doppelt genauer Repräsentation der Lösung aufgetragen. Es wurden jeweils 250 Iterationen gerechnet. Für alle diese Fälle lässt sich mit der doppelt genauen Repräsentation der Lösungskoeffizienten keine quantitativ bessere Lösung erzielen.

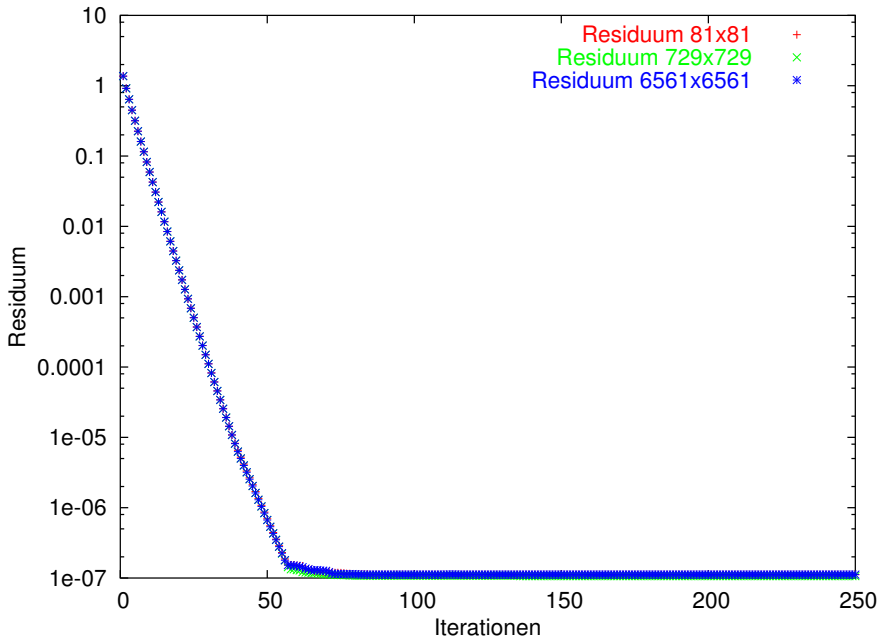

Abbildung 6.7: Entwicklung des Residuums der Poisson-Gleichung über die Löseriterationen für unterschiedliche Gitterauflösungen bei einfach genauer Repräsentation der Lösung.

Betrachtet man nun noch den Verlauf der Residuen, ebenfalls gemessen mit der Maximumsnorm, so stellt man fest, dass für den Fall doppelt genauer Repräsentation die Residuen bis in den Bereich der Maschinengenauigkeit einer doppelt genauen Gleitpunktzahl klein werden (Abbildung 6.6) und dass für den Fall einfach genauer Lösungsrepräsentation die Residuen im Bereich von 10^{-7} „stehen bleiben“ (Abbildung 6.7). Somit sehen wir die oben beschriebene Vermutung, dass in unserem Fall des hierarchischen Erzeugendensystems der durch Gleichung (6.6) ausgedrückte Einfluss von h^2 auf den Zusammenhang von Lösungsgenauigkeit und Größe des Residuums entfällt, trotzdem aber die Speicherung der Lösungswerte mit einfacher Genauigkeit ausreicht, für unseren Fall als bestätigt an, alle im Weiteren angegebenen numerischen Ergebnisse beziehen sich auf Rechnungen mit einfach genauer Repräsentation der Lösung in den 2D-Stacks.

6.3 Ergebnisse für die Poisson-Gleichung

In diesem und dem nächsten Abschnitt werden wir die Ergebnisse der numerischen Experimente für die beiden in 3.4 und 3.5 beschriebenen Modellgleichungen darstellen. Wir zeigen dabei, dass sich mit der in den Kapiteln 4 und 5 dargestellten

Methodik ein Programm realisieren lässt, das sowohl die mathematisch orientierten Forderungen nach Mehrgitterfähigkeit und Adaptivität der Rechengitter als auch die Hoffnung auf eine hohe Effizienz im L2-Cache der verwendeten Prozessoren erfüllt.

Die dargestellten Ergebnisse sind alle mit einer weit entwickelten Version erzielt worden, die die in den vorigen Kapiteln beschriebenen Techniken und Methodiken vollständig umsetzt. Insbesondere implementiert das benutzte Programm ein additives Mehrgitterverfahren, die Rechengitter können a-priori adaptiv gestaltet sein und es können für homogene Dirichlet-Randbedingungen kompliziertere Geometrien wie zum Beispiel Kreis- oder parabelförmige Gebiete in das Einheitsquadrat eingebettet werden. Wir zeigen auf, dass sich das gewünschte Mehrgitterverhalten bezüglich der Zahl benötigter Iterationen bei veränderlichen Auflösungen ergibt. Danach wird dargestellt, dass eine sehr hohe Performance im L2-Cache erreicht wird, dass damit eine streng lineare Skalierung der Kosten bezogen auf die Freiheitsgrade des Rechengitters realisiert ist und es wird bezugnehmend auf die in Abschnitt 6.1.1 dargestellten Optimalitätskriterien begründet, warum diese Effizienz und Skalierbarkeit möglich ist. Für die Grenzen der momentanen Implementierung und mögliche Erweiterungen sei auf Kapitel 7 verwiesen.

6.3.1 Die Poisson-Gleichung auf dem Einheitsquadrat

In einem ersten Schritt zeigen wir jetzt die numerischen Ergebnisse von Testrechnungen für die Poisson-Gleichung auf dem Einheitsquadrat mit homogenen Dirichlet-Randbedingungen:

$$\begin{aligned} \triangle u(\mathbf{x}) &= -2\pi^2 \sin(\pi x)\sin(\pi y) && (6.8)\\ &\quad \text{für alle} \quad \mathbf{x} = (x,y)^T \in \Omega = [0,1]\times[0,1] && (6.9)\\ u(\mathbf{x}) &= 0 \quad \text{für alle} \quad \mathbf{x} \in \partial\Omega. && (6.10) \end{aligned}$$

Dieses Problem eignet sich aufgrund seiner Einfachheit und der bekannten Lösung sehr gut für eine erste Evaluierung der numerischen Richtigkeit und Leistungsfähigkeit unseres Programms. Wir berechnen die Lösung für reguläre und adaptive Gitter verschiedener Auflösungen. Aus Tabelle 6.1 wird ersichtlich, dass für alle gerechneten Auflösungen bis zum Erreichen des Abbruchkriteriums $\varepsilon = 1.0 \cdot 10^{-5}$ (siehe Abschnitt 5.1) die exakt gleiche Anzahl von Iterationen benötigt wurde, also das gewünschte Mehrgitterverhalten vorliegt. In dieser Tabelle beschreibt die Spalte *Auflösung* die Anzahl von Zellen auf dem feinsten Level. Die Anzahl der Freiheitsgrade wird im Programmlauf gemessen, sie lässt sich an der maximalen Höhe der 2D-Stacks erkennen. Als Freiheitsgrade werden in unserem Programm auf allen Leveln nur solche Punkte betrachtet, bei denen der Träger der zugehörigen Ansatzfunktion

des hierarchischen Erzeugendensystems ganz im Inneren des Rechengebiets liegt.[3] Für ein reguläres Gitter, dessen zugehöriger Spacetree die Tiefe t hat, lässt sich die Anzahl N der Freiheitsgrade auf dem Einheitsquadrat bei Dirichlet-Randbedingungen gemäß

$$N = \sum_{j=1}^{t} (3^j + 1)^2 - \sum_{j=1}^{t} 4 \cdot 3^j \tag{6.11}$$

berechnen. Dabei wird in der linken Summe die Anzahl aller Gitterpunkte der einzelnen Level aufaddiert und von ihr für jedes Level die Zahl der Punkte abgezogen, auf denen homogene Dirichlet-Randbedingungen gelten (rechte Summe). Für adaptive Gitter und kompliziertere Geometrien (siehe unten) lässt sich naturgemäß keine solche allgemeine Formel angeben, daher werden die Freiheitsgrade durch das Programm gezählt. Die Anzahl der Iterationen ist die der echten Löseriterationen,

Auflösung	**Baumtiefe**	**Freiheitsgrade**	**Iterationen**
27×27	3	744	39
81×81	4	7.144	39
243×243	5	65.708	39
729×729	6	595.692	39
2187×2187	7	5.374.288	39
6561×6561	8	48.407.888	39

Tabelle 6.1: Mehrgitterverhalten für die Poisson-Gleichung auf dem Einheitsquadrat bei regulären Gittern.

Initialisierungsdurchläufe usw. (siehe Kapitel 5) wurden nicht gezählt.

Auflösung	**Baumtiefe**	**Freiheitsgrade**	**% reg. Gitter**	**Iterationen**
27×27	3	298	40,05	38
81×81	4	1.140	15,95	38
243×243	5	3.800	5,78	38
729×729	6	13.840	2,3	38
2187×2187	7	36.369	0,67	38
6561×6561	8	109.952	0,23	38

Tabelle 6.2: Mehrgitterverhalten für die Poisson-Gleichung auf dem Einheitsquadrat bei adaptiven Gittern.

Tabelle 6.2 zeigt die entsprechenden Ergebnisse für adaptive Gitter. Dabei wird der Rand, auf dem die Dirichlet-Bedingungen gelten, immer mit der maximalen Fein-

[3] Zu den verwendeten Begriffen wie „Inneres“, Erzeugendensystem usw. sei auf die entsprechenden Abschnitte in den Kapiteln 2, 3 und 5 verwiesen.

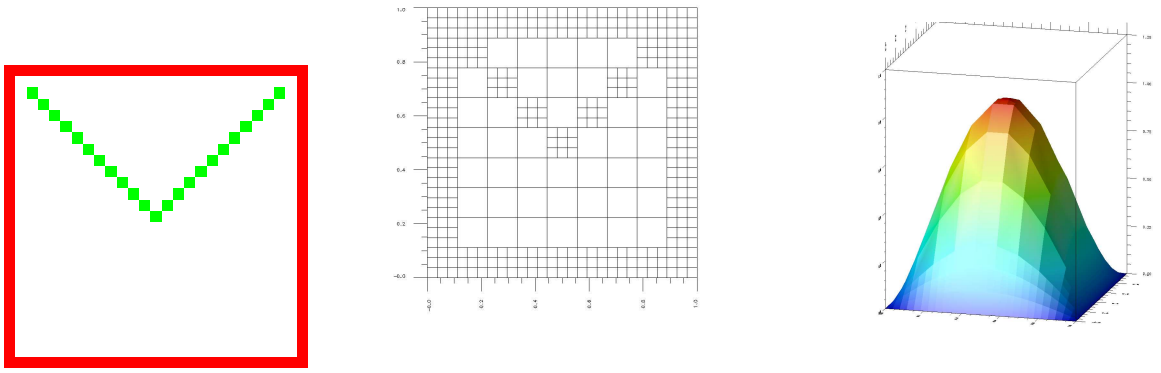

Abbildung 6.8: Eingabebild, adaptives Gitter und erzielte Lösung für eine Randauflösung von 27×27 Zellen.

heit des entsprechenden regulären Gitters (Spalte *Auflösung*) aufgelöst. Nach innen wird das Gitter vergröbert. Dabei werden exemplarisch Bereiche von der Vergröberung ausgenommen, also entsprechend der Darstellung in 5.1 im Eingabebild für `peano2d` grün gefärbt. So lässt sich die Richtigkeit der Diskretisierung und ihrer zellorientierten Implementierung auch für Sprünge in der lokal maximalen Baumtiefe sowohl am Rand als auch im Inneren zeigen. Abbildung 6.8 zeigt das Eingabebild, das resultierende Gitter und die Lösung nach 38 Iterationen für die Randauflösung 27×27 . Auch hier ergibt sich ein perfektes Mehrgitterverhalten, die Anzahl benötigter Iterationen ist exakt gleich für alle Randauflösungen. Bemerkenswert ist auch die massive Reduktion der Freiheitsgrade. So kann insbesondere für große Auflösungen wie 2187×2187 oder 6561×6561 die Zahl der benötigten Freiheitsgrade unter ein Prozent des regulären Gitters gedrückt werden, ohne die Lösung substanziell zu verschlechtern. Dies ist in diesem speziellen Fall deshalb möglich, weil der Rand maximal aufgelöst bleibt und der Mittelpunkt des Quadrats ebenfalls nicht vergröbert wird. Dies sind die Bereiche, in denen aus numerischer Sicht eine feine Gitterauflösung notwendig ist. Um für komplizierte Beispiele eine ähnlich effiziente Gitterverfeinerung beziehungsweise -vergröberung durchführen zu können, soll das Programm in Zukunft um Strategien zur Lösungsadaption ergänzt werden.

Eine detaillierte Untersuchung des Konvergenzverhaltens (bezogen auf das Abbruchkriterium „betragsgrößtes Residuum“ aller Punkte) unterschiedlicher Auflösungen und Gitter zeigt, dass diese ebenfalls unabhängig von der gewählten Gitterfeinheit und von der Art des Gitters (regulär oder adaptiv) ist. Dies ist auf Grund der oben dargestellten Iterationszahlen mit perfektem Mehrgitterverhalten auch nicht überraschend. Abbildung 6.9 gibt dazu für zwei Gitter der Randauflösung 81×81 ein Beispiel für die sehr gute Übereinstimmung der Konvergenz bei adaptiven mit

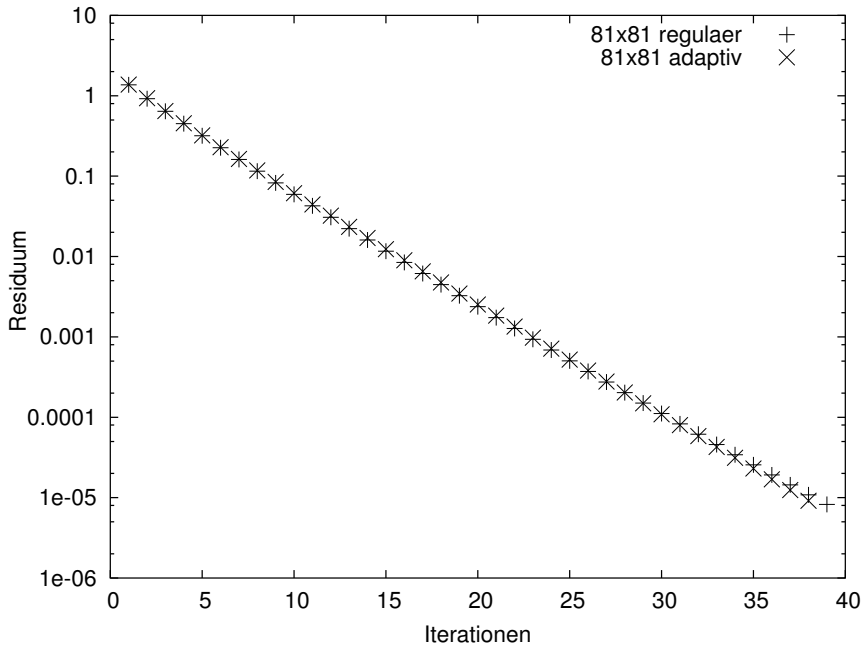

Abbildung 6.9: Konvergenzgeschwindigkeit für ein reguläres und ein adaptives Gitter mit Randauflösung 81×81.

regulären Gittern.

6.3.2 Die Poisson-Gleichung auf Kreisgebieten

In diesem Abschnitt zeigen wir am Beispiel von Kreisgebieten Ergebnisse für einfache in das Einheitsquadrat eingebettete Geometrien. Wir lösen die Poisson-Gleichung im Inneren der Kreisgebiete, auf dem Rand der Kreisgebiete gelten wieder homogene Dirichlet-Randbedingungen, außerhalb der Kreise ist keine Differenzialgleichung zu lösen, also nichts zu tun:

$$\begin{aligned} \triangle u(\mathbf{x}) &= -2\pi^2 \sin(\pi x)\sin(\pi y) && (6.12)\\ &\quad \text{für alle} \quad \mathbf{x} = (x,y)^T \in \Omega = \{\mathbf{x} \in [0,1]\times[0,1] : ||\mathbf{x}|| \leq r\} && (6.13)\\ u(\mathbf{x}) &= 0 \quad \text{für alle} \quad \mathbf{x} \in \partial\Omega. && (6.14) \end{aligned}$$

mit einem positiven Radius $r \leq 1$. Das Programm durchläuft dabei trotzdem das vollständige Rechengitter, auf den äußeren Zellen (Zelltyp „Äußere Zelle“ beziehungsweise `_OUTERCELL_*`, siehe Abschnitte 4.4 und 5.3) existieren keine Freiheitsgrade und somit wird dort nichts gerechnet. Abbildung 6.10 zeigt das verwendete adaptive Gitter und die berechnete Lösung für ein Kreisgebiet mit 81×81 Zellen auf dem feinsten Level.

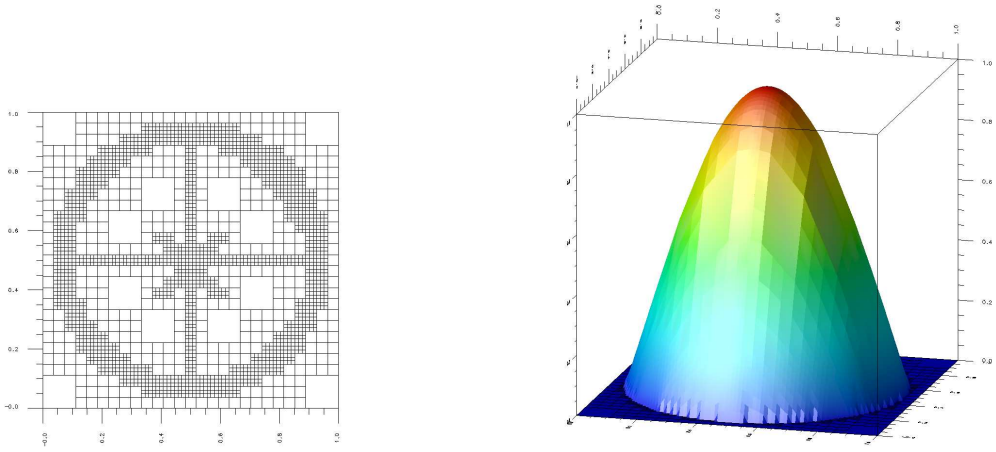

Abbildung 6.10: Adaptives Gitter und Lösung nach 210 Iterationen auf einem Kreisgebiet mit Randauflösung 81×81.

Da die Geometrie der Kreisgebiete durch ein kartesisches Gitter approximiert wird, dessen Zellen entweder ganz dem Inneren oder ganz dem Äußeren des Rechengebiets zugeordnet sind, ergeben sich abhängig von der gewählten Auflösung des feinsten Levels unterschiedliche Geometrien, was sich in der Praxis in unterschiedlichen Eingabebildern für den Baumerzeuger `peano2d` ausdrückt (vgl. Abbildung 6.11). Daher lassen sich bei diesem Beispiel die Ergebnisse von Gittern mit unterschiedlichen Randauflösungen nur schwer vergleichen. Unmittelbar vergleichbare Iterationszahlen sind nicht zu erwarten. Interessanter ist hier der Vergleich von adaptiven und regulären Gittern der gleichen Randauflösung. Die dabei von `peano2d` erzeugten Rechengitter repräsentieren in diesem Fall dieselbe Geometrie und somit sollten vergleichbare Iterationszahlen zur Untersuchung des Mehrgitterverhaltens erzielbar sein.

Die in Tabelle 6.3 aufgeführten Zahlen ergeben das erwartete Bild. Bei gleicher Geometrie ergibt sich jeweils für Gitter gleicher Randauflösung ein vergleichbares Konvergenzverhalten, die Iterationszahlen liegen entsprechend dicht zusammen. Die bei differierenden Geometrien (also hier unterschiedlichen Randauflösungen) erzielten Iterationszahlen zeigen jedoch Unterschiede auf. Für die genaueren Gitter 243×243 und 729×729 liegen sie zwar noch in einem vergleichbaren Bereich, die gröberen Gitter 27×27 und 81×81 bringen jedoch deutlich geringere Iterationszahlen, was sich durch die auf diesen Gittern stark vereinfachte und verkleinerte Geometrie (vgl. Abbildung 6.11) erklären lässt.

Weiterhin fällt auf, dass sich für die Kreisgebiete bei gleichem Abbruchkriterium die Konvergenzrate gegenüber dem Einheitsquadrat deutlich verschlechtert. Dies ist da-

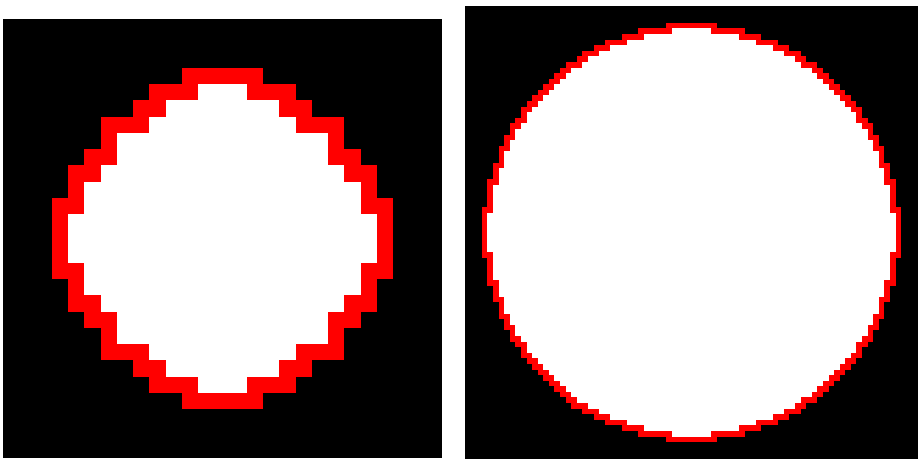

Abbildung 6.11: Eingabebilder für `peano2d`: bei unterschiedlicher Basisauflösung, in diesem Beispiel 27×27 und 81×81, ergeben sich unterschiedliche Geometrien, die Differenzialgleichung wird auf differierenden Gebieten gelöst.

mit zu begründen, dass durch die Einbettung Grobgitter-Basisfunktionen „verloren gehen“. In unseren Programmen trägt wie oben bereits erwähnt ein Gitterpunkt nur dann einen Freiheitsgrad, wenn der Träger seiner Basisfunktion ganz im Inneren des Rechengebiets liegt. Im Falle der Kreisgebiete kommt es jedoch nicht selten vor, dass die Träger von Grobgitter-Basisfunktionen in das Äußere der Geometrie greifen (Abbildung 6.12), die zugehörigen Gitterpunkte keine Freiheitsgrade darstellen und somit keinen Beitrag zur Lösung liefern können. Dies führt in Randnähe zwangsläufig zu einer Verschlechterung der Mehrgittereffizienz, damit zu einer Verlangsamung der Konvergenzgeschwindigkeit und somit zu höheren Iterationszahlen. Dieses Problem lässt sich durch das Wiedereinfügen der betroffenen Freiheitsgrade in Randnä-

Auflösung	**Baumtiefe**	**Freiheitsgrade**	**% reg. Gitter**	**Iterationen**
27×27	3	308	100,0	114
27×27	3	92	29,87	110
81×81	4	4.689	100,0	204
81×81	4	1.753	37,85	214
243×243	5	48.740	100,0	269
243×243	5	15.554	31,91	264
729×729	6	438.719	100,0	281
729×729	6	156.316	35,63	286

Tabelle 6.3: Mehrgitterverhalten für die Poisson-Gleichung auf Kreisgebieten für adaptive und reguläre Gitter.

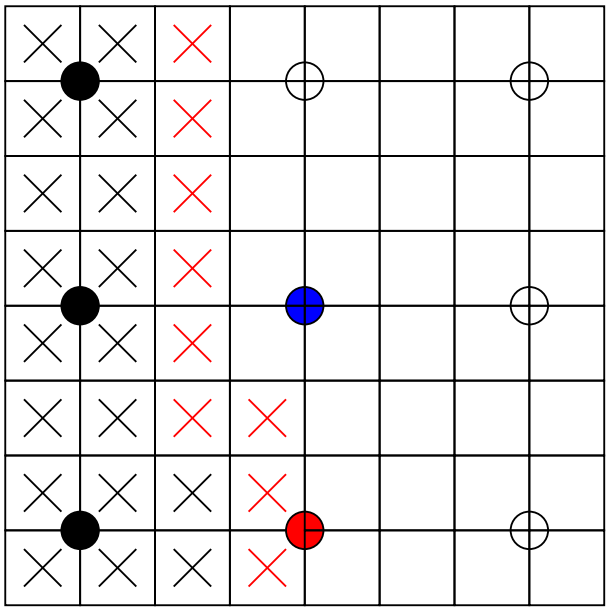

Abbildung 6.12: „Verlorener“ Grobgitterpunkt. Die zu dem blau markierten Grobgitterpunkt gehörige Basisfunktion greift mit ihrem Träger über den Rand in das Gebietsäußere und ist damit kein Freiheitsgrad.

he in Kombination mit einer geeigneten an die vorliegende Randform angepassten, lokal neu definierten Operatordiskretisierung vermeiden (siehe [9] und [17]). Eine entsprechende Erweiterung unseres Programms sollte ohne Auswirkungen auf die verwendete Methodik zur Datenorganisation, das heißt insbesondere ohne Verluste bei der Cache-Effizienz, durchführbar sein (siehe auch Kapitel 7).

Zusammengefasst lässt sich festhalten, dass für die Poisson-Gleichung sowohl für das Einheitsquadrat als auch für die komplizierteren Kreisgebiete als Definitionsbereich die numerisch orientierten Ziele erreicht sind. Das Programm realisiert Mehrgittereffizienz in beiden Fällen, die a-priori Adaption sowie die Behandlung eingebetteter Geometrien wirkt sich nicht störend auf das Mehrgitterverhalten aus. Lediglich die „Ausdünnung“ der Freiheitsgrade im Falle der Kreisgebiete führt zu einer verschlechterten Mehrgittereffizienz bezogen auf jeweils eine spezielle Geometrie. Wird dieselbe Geometrie mit regulären oder adaptiven Gittern gerechnet, sind die Iterationszahlen wieder nahezu identisch, das Programm zeigt sich hier also als durchaus robust. Die Verschlechterung der Mehrgittereffizienz in diesen Fällen stellt aber ein prinzipielles Problem dar, das nicht in unserer speziellen Implementierung begründet ist.

6.3.3 Verhalten im L2-Cache

In diesem Abschnitt stellen wir dar, wie die Effizienz unseres Programms bezogen auf die Ausnutzung der Leistungsfähigkeit modernen Prozessoren zu bewerten ist. Dazu betrachten wir zunächst die Ergebnisse von Simulationen der Cache-Effizienz mit Hilfe von *cachegrind*, zeigen dann einige Performance-Messwerte, die mit *perfex* beziehungsweise *hpcmon* aus den hardware performance counters ermittelt wurden, und versuchen schließlich die erzielten Performance-Werte anhand der in Abschnitt 6.1.1 erläuterten Optimalitätsforderungen zu begründen.

6.3.3.1 Simulation mit cachegrind

Die Simulation des Verhaltens unseres Programms zur Lösung der Poisson-Gleichung in Bezug auf cache misses im L2-Cache bringt als wesentliches Ergebnis, dass vor allem für praxisrelevante Auflösungen ab 81×81 Zellen auf dem feinsten Level die cache misses im Wesentlichen genau dort stattfinden, wo sie zu erwarten und unvermeidlich sind.

Die Simulation mit cachegrind erlaubt – wie oben bereits erwähnt – die auftretenden Ereignisse bei Speicherzugriffen zum Einen zu quantifizieren und zum Anderen im Programm zu lokalisieren. Wie man der Abbildung 6.13 entnehmen kann, sind bei höheren Auflösungen weniger als fünf Prozent der cache misses in der Routine `iteration` selbst zu lokalisieren, mehr als 95 Prozent der Gesamtzahl verteilen sich gleichmäßig auf die Stack-Zugriffsroutinen `push` und `pop`. Diese beiden Routinen haben die Aufgabe, Daten von den Stacks zu holen oder sie auf den Stacks abzulegen (siehe Abschnitt 5.2). Dass dabei cache misses auftreten, ist aufgrund der Tatsache unvermeidlich, dass der gesamte Speicherbedarf eines Programmlaufs durch zum Beispiel 256 KByte L2-Cache beim Intel Pentium III nicht mehr abgedeckt werden kann. Dass nun aber die Mehrzahl der insgesamt auftretenden cache misses in diesen Routinen zu finden ist, ist äußerst positiv zu bewerten, da es zum Einen zeigt, dass das Programm insgesamt bezogen auf die Speicherzugriffe wenig Overhead erzeugt, und zum Anderen darlegt, dass qualitativ cache misses nur dort auftreten, wo sie unvermeidbar sind.

Zur endgültigen Feststellung der Unvermeidbarkeit der auftretenden cache misses muss natürlich ergänzen die Gesamtzahl der cache misses analysiert werden. Jeder denkbare iterative Gleichungslöser muss in jeder Iteration jeden Gitterpunkt beziehungsweise Freiheitsgrad mindestens einmal „anfassen“ um die numerische Lösung verbessern zu können. Deshalb schätzen wir mit der Anzahl N der Freiheitsgrade, der Größe s eines Stackelements[4] und der Größe cl einer cache line die minimale

[4] Ein Stackelement ist in unserem Fall ein `C struct`, das verschiedene Zahlenwerte wie Koeffizienten der Basisdarstellung, interpolierte Gesamtlösung usw. trägt (siehe auch Abschnitt 5.2).

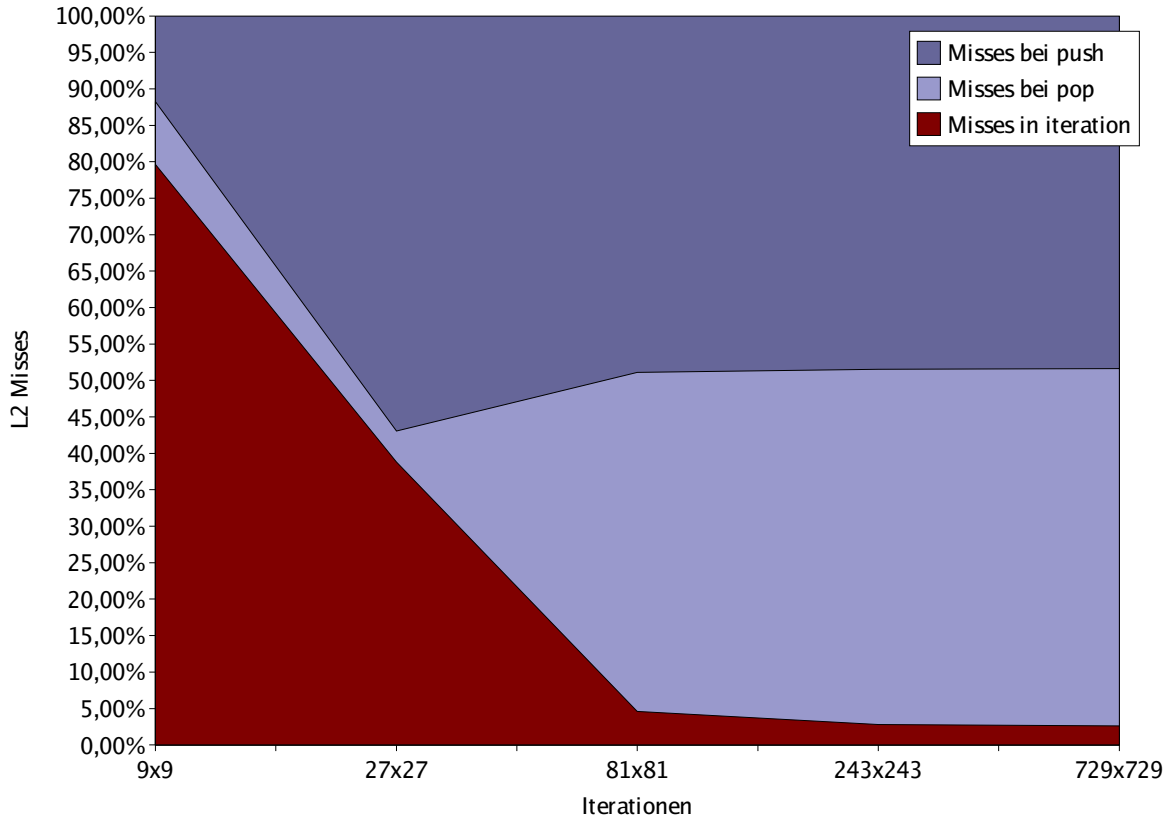

Abbildung 6.13: Verteilung der L2 cache misses auf `push`, `pop` und `iteration` für verschiedene Auflösungen bei jeweils 25 Iterationen. Der Einfluss der Initialisierungsroutine wurde herausgerechnet [24].

Zahl

$$cm_{min} := \frac{N \cdot s}{cl} \tag{6.15}$$

der notwendigen cache misses pro Löseriteration ab. Dabei setzten wir voraus, dass jeweils ein zusammenhängender Abschnitt der Größe cl aus dem im Hauptspeicher befindlichen 2D-Eingabestack in den L2-Cache geladen wird. Wir gehen also davon aus, dass die Reihenfolge und der Zusammenhang der Array-Elemente in der Adressierung im Hauptspeicher beibehalten werden. Diese Annahme könnte in der Praxis eher etwas zu optimistisch sein. Außerdem nehmen wir an, dass die Daten eines Speicherblocks bei der ersten Benutzung innerhalb einer Iteration nicht mehr im L2-Cache zur Verfügung stehen und somit jeweils cache misses verursachen. Aufgrund der geringen Größe des L2-Caches verglichen mit dem gesamten Speicherbedarf ist dies zumindest für höhere Auflösungen eine realistische Annahme. Der Quotient

$$Rate := \frac{cm_{real}}{cm_{min}} \tag{6.16}$$

setzt diesen Minimalwert in Beziehung mit den tatsächlich je Iteration gemessenen cache misses cm_{real}. In Tabelle 6.4 ist dieser Wert für verschiedene Gitter zur Lösung der Poisson-Gleichung auf dem Einheitsquadrat berechnet. Die Werte für 27×27-

Auflösung	**Baumtiefe**	**Freiheitsgrade**	**Rate**
27×27	3	744	0,11
81×81	4	7.144	0,88
243×243	5	65.708	1,09
729×729	6	595.692	1,12
2187×2187	7	5.374.288	1,12

Tabelle 6.4: Verhältnis tatsächlicher cache misses zu geschätztem Minimum für verschiedene Gitter.

und 81×81-Gitter sind kleiner als eins, es finden also weniger cache misses statt als die Abschätzung für diese Gitter ergibt. Dies lässt sich auf die kleine Gesamtgröße der Probleme zurückführen. Ihr Speicherbedarf ist nur ein kleines Vielfaches der oder sogar kleiner als die Größe des L2-Caches. Bei einem Speicherbedarf von 60 Byte pro Freiheitsgrad werden für die zwei 2D-Stacks im Falle des 27×27-Gitters $2 \cdot 60 \cdot 744$ Byte $= 89280$ Byte $= 87,1875$ KByte und im Falle des 81×81-Gitters $837,1875$ KByte benötigt. Daher ist die Wahrscheinlichkeit, dass benötigte Daten von einer Iteration zur nächsten oder von einer Verwendung der Daten zur nächsten innerhalb einer Iteration nicht aus dem Cache verdrängt werden, hier noch nicht zu vernachlässigen. Für größere Gitter scheint sich die Rate um einen Wert von 1,12 zu stabilisieren. Dies ist auch für noch größere Gitter als die hier gezeigten zu erwarten, da keine uns bekannten Störeffekte hinzukommen, die schlechtere Raten zulas-

sen würden. Zu erwähnen ist an dieser Stelle auch noch, dass die realen Zahlen an cache misses nur um den Einfluss der Initialisierungsschleife bereinigt sind. Insbesondere wurde weder in der Abschätzung für cm_{min} noch durch Manipulation von cm_{real} berücksichtigt, dass bei unserem Finite-Elemente-Ansatz jeder Freiheitsgrad im Inneren viermal statt nur einmal verwendet wird. Die in die Rate eingegangenen Zahlen cm_{real} sind also die echten Messwerte für das FE-Verfahren bei voller Verwendung aller Freiheitsgrade. Daher liefern die erzielten Raten einen deutlichen Hinweis darauf, dass tatsächlich keine unnötigen cache misses auftreten.

Im Rahmen der Arbeit von Reinhard Stein [24] wurden noch einige weitere Effekte mit Hilfe des Simulationswerkzeugs cachegrind untersucht, die wir hier nur stichpunktartig zusammenfassen wollen. Dabei gehen wir von ausreichend großen Gittern ab 81×81 Zellen aus.

- Die Gesamtzahl der cache misses skaliert linear mit der Anzahl der Freiheitsgrade und der Iterationen. Dadurch lässt sich der Aufwand für Berechnungen auf größeren Gittern (zusammen mit der unten dargestellten Skalierung der Laufzeiten) sehr gut vorhersagen. Insbesondere reicht für die Analyse unserer Programme mit cachegrind die Betrachtung sehr weniger Iterationen aus, was wegen der oben besprochenen drastischen Laufzeitverlängerung bei der Simulation mit cachegrind viel Zeit spart. Man muss eben nicht viele Iterationen rechnen, um Störeinflüsse quasi „auszumitteln“, die Simulation ist ausreichend präzise bereits bei sehr wenigen Iterationen.

- Die Zahl der cache misses skaliert linear mit der Größe des Stackelements. Dies ist besonders deshalb interessant, weil dadurch der Einfluss größerer Stackelemente als bisher, die man bei komplizierteren Differenzialgleichungen oder algorithmischen Erweiterungen (siehe Kapitel 7) einführen müsste, vorhersagbar ist. Insbesondere hat sich dabei gezeigt, dass es in unserem Fall nicht notwendig ist, auf die Größe einer cache line Rücksicht zu nehmen. Oft wird versucht, die Größe von Speicherstrukturelementen als Vielfaches der Cache-Line-Größe zu implementieren. Dies hat bei uns keinerlei Auswirkung, weder positiv noch negativ. Daher ergibt sich für die Entwickler von geplanten Erweiterungen sehr viel Flexibilität im Bezug auf die Gestaltung der Stackelemente.

- Die Zahl der cache misses skaliert linear mit der Größe der cache lines. Mit cachegrind ist es möglich, den Cache, auf dem simuliert werden soll, in gewissen Grenzen selbst zu definieren. Es lassen sich die Gesamtgröße, die Cache-Line-Größe und die Assoziativität einstellen. Dabei stellen wir fest, dass sich bei Line-Größen von 64 Byte über 128 Byte zu 256 Byte jeweils die Zahl der cache misses halbiert. Dies trifft unabhängig von der Gesamtgröße des Caches und der Assoziativität zu.

- Die Gesamtgröße des Caches und die Assoziativität spielen eine untergeordnete Rolle. Hält man die Cache-Line-Größe fest und variiert darauf die Assoziativität (2-fach, 4-fach, 8-fach) oder die Gesamtgröße des Caches (256 KByte, 512 KByte, 1024 KByte), sind für ausreichend hohe Gitterauflösungen keine signifikanten Änderungen der Cache-Miss-Zahlen zu beobachten.

Insgesamt ergibt sich, dass das Programm bezüglich aller relevanten Parameter linear skaliert. Der optimale Rechner für unser Programm hat einen mindestens 2-fach assoziativen L2-Cache, dessen Größe mit 256 KByte ausreichend groß ist (trifft auf alle modernen Prozessoren wie IA32- oder IA64-Architekturen zu), und möglichst große cache lines , zum Beispiel 256 Byte (dies trifft für die meisten Architekturen nicht zu, in der Regel sind 128 Byte große cache lines implementiert).

6.3.3.2 Performance-Messungen mit perfex und hpcmon

Nun wenden wir uns der Messung der realen Cache-Performance auf konkreten Rechnern mit den Werkzeugen perfex und hpcmon zu. Diese Programme werten prozessindividuell die Event-Monitoring-Register der Prozessoren aus und stellen die gemessenen Werte übersichtlich dar (siehe Abbildung 6.2).

In Tabelle 6.5 sind die Performance-Werte Laufzeit, L2-Hit-Rate und MFLOPS (Millionen Gleitpunktoperationen pro Sekunde) angegeben. Die Ergebnisse wurden auf einem Dual Xeon 2,4 GHz mit vier GByte Hauptspeicher unter Redhat Linux berechnet.[5] Berechnet wurde die Lösung der Poisson-Gleichung auf dem Einheitsquadrat, das Programm wurde mit maximal aggressiven Compileroptionen (-O3 -xW) mit dem nativen Intel C-Compiler Version 8.0 übersetzt. Die Laufzeiten sind der tatsächliche (also nicht um die Initialisierungsschleife bereinigte) Zeitbedarf für 39 Iterationen. Die L2-Hit-Rate ist extrem hoch, die MFLOPS-Rate ist abgesehen von den kleinen Gittern nahezu konstant bei etwa 360 MFLOPS. An den Laufzeiten insbesondere für die größeren Gitter ab 81×81 lässt sich die lineare Skalierung des Programms bezüglich der Anzahl der Freiheitsgrade erkennen: von einer Gittertiefe zur nächsten vergrößert sich die Anzahl der Freiheitsgrade ungefähr um den Faktor neun, die Laufzeiten sind jeweils ebenfalls recht genau um den Faktor neun höher. Für die drei größten Gitter ist die Skalierung auch im Speicherbedarf erkennbar, er steigt auch etwa um einen Faktor neun von Gitter zu Gitter. Für die kleineren Gitter ist diese Skalierung nicht sichtbar, weil dort der Anteil der Nutzdaten (also der Freiheitsgrade) gegenüber anderem Speicherbedarf, wie er beispielsweise beim Laden des Programms selbst in den Arbeitsspeicher entsteht, zu klein ist.

[5] Im Prinzip würde für unser sequenzielles Programm auch ein Prozessor reichen, die vorhandene Hardware am Lehrstuhl für Rechnertechnik und Rechnerorganisation (Prof. A. Bode) in unserem Institut hat jedoch zwei Prozessoren je Rechner. Daher können alle anderen Prozesse des Rechners auf dem zweiten Prozessor bearbeitet werden, die Antwortzeiten werden für uns etwas kürzer.

Tabelle 6.6 zeigt die entsprechenden Werte für adaptive Gitter. Die Laufzeitangaben beziehen sich auf den gesamten Programmdurchlauf mit 38 Löseriterationen. Im Vergleich mit den regulären Gittern bleibt die L2-Hit-Rate in etwa auf dem gleichen sehr hohen Niveau, die MFLOPS-Rate sinkt um circa 9% ab. Dies lässt sich mit dem durch die Verteilung der Verfeinerungsgebiete schlechteren Verhältnis zwischen Organisationsaufwand und tatsächlichen Berechnungen begründen. Da zum Beispiel der Rand weiterhin mit der Auflösung des korrespondierenden regulären Gitters berechnet wird, muss der Spacetree hier trotzdem immer wieder bis zur maximalen Tiefe durchlaufen werden, dies passiert zu Lasten der Floating-Point-Performance, da in den Randzellen deutlich weniger Berechnungen durchgeführt werden als in den inneren Zellen.

Die bisher präsentierten Ergebnisse sind alle auf Rechnern mit IA32-Xeon-Prozessoren berechnet. Interessant ist nun die Frage, ob die hohe Performance erhalten bleibt, wenn man die Architektur wechselt. Wir haben das auf einem Vier-Wege-Itanium2-Rechner und einem AMD Athlon getestet. In beiden Fällen zeigt sich die gleiche extrem hohe L2-Cache-Performance, die Tabelle 6.7 zeigt dies für die Beispielrechnungen auf einem Quad-Itanium2-Rechner, die vier Prozessoren sind mit 1,3 GHz getaktet, der L2-Cache besitzt 128 Byte große cache lines mit insgesamt

Auflösung	**Freiheitsgrade**	**Speicher**	**Laufzeit**	**L2-Hit-Rate**	**MFLOPS**
27×27	744	1,2 MB	0,08 s	99,87%	319,58
81×81	7.144	1,3 MB	0,61 s	99,94%	340,77
243×243	65.708	2,3 MB	5,28 s	99,92%	354,55
729×729	595.692	10 MB	46,88 s	99,91%	360,20
2187×2187	5.374.288	84 MB	424,25 s	99,91%	359,65
6561×6561	48.407.888	742 MB	3.819,41 s	99,91%	358,64

Tabelle 6.5: Performance-Werte für die Poisson-Gleichung auf dem Einheitsquadrat für reguläre Gitter auf einem Dual Xeon 2,4 GHz und 4 GByte Hauptspeicher.

Randauflösung	**Freiheitsgrade**	**Speicher**	**Laufzeit**	**L2-Hit-Rate**	**MFLOPS**
27×27	298	1,2 MB	0,05 s	99,79%	284,21
81×81	1.140	1,3 MB	0,15 s	99,89%	310,43
243×243	3.800	1,5 MB	0,47 s	99,92%	321,53
729×729	13.840	2 MB	1,61 s	99,89%	328,25
2187×2187	36.369	3,1 MB	4,35 s	99,88%	328,10
6561×6561	109.952	7 MB	13,15 s	99,88%	328,93

Tabelle 6.6: Performance-Werte für die Poisson-Gleichung auf dem Einheitsquadrat für adaptive Gitter, Xeon-Architektur.

256 KByte Cache-Größe, es sind 8 GByte Hauptspeicher installiert. Der Itanium2 besitzt im Gegensatz zum Xeon einen L3-Cache mit 3 MByte Größe, auf den aber um vier Größenordnungen weniger oft zugegriffen wird als auf den L2-Cache, er spielt deshalb für die Beurteilung der Performance unseres Programms keine Rolle.[6] Tabelle 6.7 zeigt die Performance-Werte für reguläre Gitter für die Lösung der Poisson-Gleichung auf dem Einheitsquadrat mit 39 Iterationen, also bis zum Erreichen des Abbruchkriteriums von $\varepsilon = 1.0 \cdot 10^{-5}$. Der Vergleich der Werte für den Itanium2

Auflösung	**Freiheitsgrade**	**Laufzeit**	**L2-Hit-Rate**	**MFLOPS**
27×27	744	0,06 s	99,34%	339,14
81×81	7.144	0,44 s	99,81%	440,56
243×243	65.708	3,84 s	99,93%	464,14
729×729	595.692	34,46 s	99,97%	469,63
2187×2187	5.374.288	308,87 s	99,97%	470,41
6561×6561	48.407.888	2801,08 s	99,98%	468,33

Tabelle 6.7: Performance-Werte für die Poisson-Gleichung auf dem Einheitsquadrat für reguläre Gitter, gerechnet auf Itanium2.

mit denen des Xeon in Tabelle 6.5 zeigt, dass die L2-Cache-Performance praktisch gleich ist. Der Itanium erzielt (bei niedrigerer Taktung) höhere MFLOPS-Raten, für große Gitter ist die Laufzeit um etwa 25% kürzer. Führt man sich jedoch den gravierenden Unterschied im Anschaffungspreis beider Systeme vor Augen, so lohnt sich im Moment der Einsatz der Itanium-Architektur für dieses Programm kaum.[7] Portierbarkeit des Programms auf andere Architekturen bei gleich bleibender L2-Performance ist ein weiterer Hinweis auf die Qualität der Methodik. Alle weiteren Performance-Tabellen dieser Arbeit beziehen sich wieder auf die oben beschriebene Xeon-Architektur.

Für die Lösung der Poisson-Gleichung auf Kreisgebieten ergeben sich ebenfalls vergleichbare Performance-Werte, wie man der Tabelle 6.8 entnehmen kann. Hier wurden die Werte für Laufzeiten und Speicherbedarf weggelassen, weil sich wie oben dargestellt aus den unterschiedlichen Eingabebildern für die Gittergenerierung substanziell unterschiedliche Geometrien und Iterationszahlen ergeben. Bemerkenswert ist hier vor allem, dass sich das Vorhandensein der äußeren Zellen, die zwar durch-

[6] Die hit rate auf dem L3-Cache kann sich aber trotzdem sehen lassen, sie liegt zwischen 50% und 96%.

[7] Es wäre allerdings eine interessante Aufgabe, das Programm so zu verändern, dass es die 64Bit-Architektur des Itanium2 besser nutzt und dadurch deutlich bessere Laufzeiten erzielt. Insbesondere einige Besonderheiten in der FP-Einheit des Itanium wären hier von Interesse. So kann beispielsweise in einem Zyklus auf zwei Zahlen gleichzeitig eine Addition und eine Multiplikation ausgeführt werden. Eine gezielte Ausnutzung solcher Features könnte eventuell die Performance unseres Programms auf dieser Architektur bezüglich MFLOPS und Laufzeit deutlich steigern.

laufen werden müssen, auf denen aber im Sinne der Lösungsberechnung nichts gerechnet wird, nicht negativ auf die L2-Hit-Rate und die MFLOPS-Rate auswirkt. Die erzielten Werte sind (abgesehen vom zu kleinen 27×27 Gitter, siehe oben) im selben Bereich wie die Werte für Rechnungen auf dem Einheitsquadrat. Dies spricht wiederum für die Robustheit und die Effizienz sowohl der Methodik als auch der Implementierung, die dieser Arbeit zugrunde liegen.

Abbildung 6.14 zeigt nochmal die bereits in der Simulation mit cachegrind festgestellte hervorragende Effizienz des Programms im Speicher, insbesondere die gleichmäßige lineare Skalierbarkeit bezüglich der Anzahl der Freiheitsgrade. In diesem Diagramm sind alle oben verwendeten Gitter für die Lösung der Poisson-Gleichung, egal ob auf dem Einheitsquadrat oder in einem Kreisgebiet, egal ob regulär oder adaptiv, mit der Zahl ihrer Freiheitsgrade (logarithmisch skalierte x-Achse) gegen die Kosten in ms, die ein Freiheitsgrad in einer Iteration beim Rechnen auf der Xeon-Architektur erzeugt, aufgetragen. Abgesehen von dem adaptiven 27×27-Gitter für das Kreisgebiet, das nur 92 Freiheitsgrade besitzt und damit wie oben beschrieben zu klein für den realistischen Vergleich mit den anderen Gittern ist, liegen die Kosten pro Freiheitsgrad sehr dicht im Bereich von 0,002 ms bis 0,0045 ms zusammen. Das ist umso bemerkenswerter, als die in dieses Diagramm eingegangenen Gitter in ihrer Charakteristik bezüglich Anzahl von Freiheitsgraden in unterschiedlichen Leveln, Regularität und Adaptivität sowie Vorhandensein von äußeren Punkten deutlich differieren. Es bestätigt neben der Methodik und der gewählten Implementierung vor allem auch das Konzept, komplizierte Geometrien auf die beschriebene Weise in das Einheitsquadrat einzubetten, weil sich das „leere“ Durchlaufen der Außenzellenbereiche der so entstehenden Geometrie kaum negativ auf die Kosten pro Freiheitsgrad auswirkt. Für andere Beispiele, an denen dieser Zusammenhang in Zukunft noch verifiziert werden sollte, ist aus jetziger Sicht nicht zu erwarten, dass diese Feststellung substanziell verändert werden müsste.

Auflösung	**Freiheitsgrade**	**% reg. Gitter**	**L2-Hit-Rate**	**MFLOPS**
27×27	308	100,0	99,92%	241,92
27×27	92	29,87	99,87%	221,39
81×81	4.689	100,0	99,95%	304,55
81×81	1.753	37,85	99,93%	291,66
243×243	48.740	100,0	99,90%	329,49
243×243	15.554	31,91	99,89%	330,59
729×729	438.719	100,0	99,89%	333,91
729×729	156.316	35,63	99,87%	352,05

Tabelle 6.8: Performance-Werte für die Poisson-Gleichung auf Kreisgebieten für adaptive und reguläre Gitter, gerechnet auf Xeon.

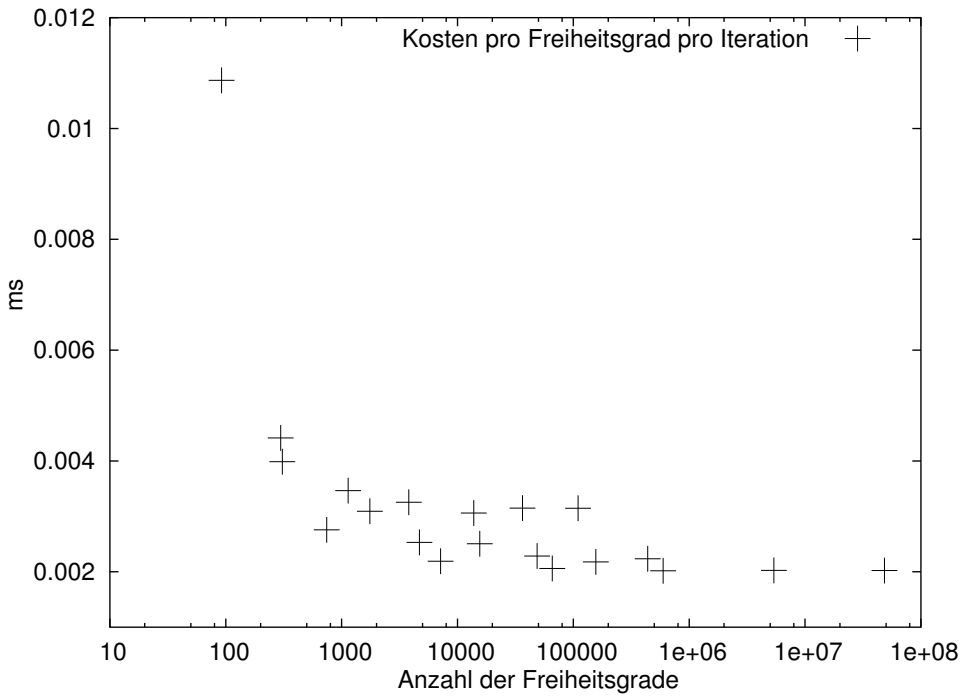

Abbildung 6.14: Kosten in ms pro Freiheitsgrad pro Iteration für alle Gitter.

6.3.3.3 Analyse des Zusammenhangs zwischen erzielter Cache-Performance und verwendeter Methodik

Zum Abschluss dieses Abschnitts wollen wir noch darstellen, wie die gezeigte hervorragende Effizienz unserer Programme im L2-Cache zu erklären ist. Betrachten wir also die in Abschnitt 6.1.1 beschriebenen Anforderungen für hohe Cache-Effizienz:

1. **Zeitliche Lokalität**
   Durch die element-orientierte Implementierung der Finite-Elemente-Methode entlang der raumfüllenden Kurve werden innerhalb einer Löseriteration innere Punkte in den meisten Fällen entweder als Block vier Mal hintereinander bearbeitet und sind danach „fertig“ oder in zwei Blöcken zu je zwei Bearbeitungen verwendet. Dies gilt auf allen Leveln auch im Sinne der Mehrgitterkorrekturen. Somit ist eine große zeitliche Lokalität für alle inneren Punkte gegeben. Randpunkte, hängende Punkte und Punkte außerhalb des Rechengebiets spielen in diesem Zusammenhang keine Rolle, da sie keine Freiheitsgrade im eigentlichen Sinn darstellen und somit nie als Punkte in den 2D-Stacks existieren, also auch nie aus dem Hauptspeicher geholt werden müssen (siehe auch Kapitel 5).

2. **Örtliche Lokalität**
   Alle inneren Punkte sind zu jedem Zeitpunkt während einer Löseriteration auf Stacks angeordnet. Die Stacks werden streng linear bearbeitet und sind in ihrer Anzahl auflösungsunabhängig begrenzt. Wird ein Punkt von einem Stack gelesen oder auf einen Stack geschrieben, so wird mit großer Wahrscheinlichkeit zeitnah erneut auf diesen Stack lesend oder schreibend zugegriffen, es wird also in den meisten Fällen zeitnah auf das im Stack benachbarte Element zugegriffen. Bezogen auf die Stacks kann man somit zu Recht von örtlicher Lokalität sprechen. Die Stacks sind als Vektoren durch das Sprachmittel `array` realisiert. Man kann deshalb mit hoher Wahrscheinlichkeit davon ausgehen, dass die Elemente der Stacks auch im Arbeitsspeicher sequenziell in aufeinander folgenden Speicherblöcken angeordnet sind, die dann aus oben beschriebenen Gründen zeitnah aufeinander folgend geladen oder geschrieben werden. Deshalb ist mit hoher Wahrscheinlichkeit für alle inneren Punkte auch das Kriterium *örtliche Lokalität* erfüllt. Alle anderen Punkte müssen mit dem gleichen Argument wie oben nicht betrachtet werden.

3. **Hardware-Prefetching**
   Für das erste Laden eines Punktes innerhalb einer Iteration wird lesend auf den 2D-Eingabestack zugegriffen. Dieser wird sequenziell und ausschließlich lesend abgebaut. Könnte der Prozessor auf der Ebene der Stack-Elemente ein Ein-Schritt-Hardware-Prefetching realisieren, wären die spekulativ geladenen Daten in den meisten Fällen genau die, die sehr zeitnah (zum Beispiel in der nächsten Zelle) benötigt würden. Der 2D-Stack ist als `array` realisiert, dessen Daten sehr wahrscheinlich sequenziell im Speicher angeordnet sind. Deshalb führt das reale speicherblock-basierte Prefetching entsprechend obiger Argumentation mit hoher Wahrscheinlichkeit beim jeweils ersten Laden der inneren Punkte in einer Iteration zum Erfolg. Die durch das initiale Laden der Daten verursachten cache misses werden substanziell verringert.

Somit sind die drei oben angeführten Optimalitätsanforderungen mit hoher Wahrscheinlichkeit zu jedem Zeitpunkt der Programmausführung erfüllt, was die für ein Programm, das reale Berechnungen auf großen Datenmengen durchführt, extrem hohe L2-Cache-Performance angemessen erklärt. Es sind noch weitere Effekte vorstellbar, die im Zusammenhang mit der Cache-Effizienz unserer Programme eine Rolle spielen könnten wie zum Beispiel die Möglichkeit der so genannten out-of-order-execution, also der Ausführung einer Operation außerhalb der vom Programmtext vorgegebenen Reihenfolge. Diese Effekte sind jedoch sehr aufwändig zu untersuchen und in ihrer Wirkung schwer zu quantifizieren.

6.4 Ergebnisse für die Stokes-Gleichung

Zum Abschluss dieses Kapitels stellen wir nun die Ergebnisse der numerischen Experimente für die Stokes-Gleichung dar. Dabei ergeben bezogen auf die Cache-Effizienz sich im Wesentlichen die gleichen Performance-Werte wie bei der Poisson-Gleichung. Wir verzichten deshalb an einigen Stellen auf die eingehende Interpretation der Ergebnisse, da diese keine neuen Erkenntnisse im Vergleich zu den Ergebnissen bei der Lösung der Poisson-Gleichung bringen würden.

6.4.1 Die Stokes-Gleichung auf dem Einheitsquadrat

Zunächst betrachten wir die Aufgabe, die Stokes-Gleichung auf dem Einheitsquadrat zu lösen. Nach Abschnitt 3.5 ist also eine Lösung $\mathbf{u} = (u, v)^T : \Omega \mapsto \Omega$ der Gleichung

$$\begin{aligned} \triangle \mathbf{u} - \text{grad}p &= 0 && (6.17)\\ \text{div}\mathbf{u} &= 0 && (6.18)\\ u(x,y) = 1,\ v(x,y) &= 0 \quad \text{für} \quad y = 1 && (6.19)\\ u(x,y) = v(x,y) &= 0 \quad \text{für} \quad (x,y)^T \in \partial\Omega \backslash \{(x,y)^T \in \partial\Omega : y = 1\} && (6.20) \end{aligned}$$

gesucht, wobei Ω das Einheitsquadrats ist. Der Druck p ist dabei nur bis auf eine Konstante bestimmt. Dieses Problem ist auch als *driven cavity* bekannt, die Strömung wird gewissermaßen von einer Platte, die mit konstanter Geschwindigkeit entlang der oberen Kante des Einheitsquadrats gezogen wird, angetrieben.

Abbildung 6.15 zeigt die Lösung, repräsentiert durch Geschwindigkeitsvektoren, für ein reguläres 27×27-Gitter nach 168 Iterationen, dem Zeitpunkt des Erreichens der Abbruchbedingung. Man sieht deutlich den charakteristischen Wirbel, der sich etwas oberhalb des Mittelpunkts des Einheitsquadrats ausbildet. Am oberen Rand ist die Geschwindigkeit am größten, dort wird die Strömung angetrieben.

Tabelle 6.9 zeigt die erzielten Performance-Werte für reguläre Gitter auf dem Xeon-System. Auch hier zeigt sich das typische Mehrgitterverhalten, die Konvergenzgeschwindigkeit ist jedoch niedriger als bei der Poisson-Gleichung. Dies lässt sich vor allem auf die verwendete Uzawa-Druckkorrektur zurückführen (siehe auch [6] und [7]). Die Performance im L2-Cache ist immer noch sehr hoch, insgesamt aber um einen Prozentpunkt schlechter als bei der Poisson-Gleichung. Der vermutliche Grund dafür sind die zusätzlichen Stacks für die zellbasierte Information „Druck“ (siehe Abschnitt 5.4). Da sich hier cache misses (ähnlich wie bei den 2D-Stacks) für das erstmalige Lesen der Daten je Iteration nicht vollständig vermeiden lassen, gehen die dabei entstehenden cache misses negativ in die Bilanz der L2-Cache-Events ein. Da aber jeder Druckwert nur einmal pro Iteration verwendet wird, erzeugt er

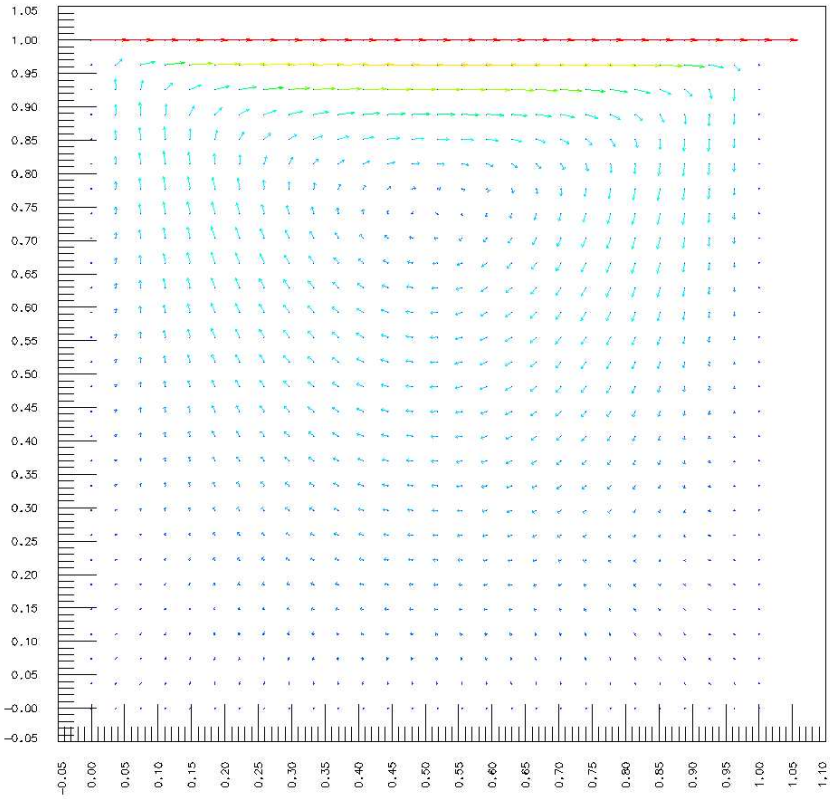

Abbildung 6.15: Lösung der Stokes-Gleichung auf dem Einheitsquadrat.

im Gegensatz zu den öfter verwendeten Freiheitsgraden für die Geschwindigkeiten u und v auf den anderen Stacks weniger cache hits, die positiv in die Cache-Miss-Bilanz eingehen würden. Die niedrigere MFLOPS-Rate lässt sich mit der rechten Seite der hier zu lösenden Poisson-Gleichungen erklären. Diese ist bis auf die Uzawa-Druckkorrektur 0, es sind also auf dem feinsten Level wesentlich weniger Gleitpunktoperationen durchzuführen als im Falle der Poisson-Gleichung mit der trigonometrischen rechten Seite (siehe Abschnitt 3.4). Wir haben diesen Zusammenhang durch entsprechende Tests für die Poisson-Gleichung mit verschiedenen rechten Seiten verifiziert. Setzt man dort eine einfachere rechte Seite an, zum Beispiel

$$\triangle u(\mathbf{x}) = 1 \tag{6.21}$$

mit gleichen Randbedingungen wie oben, so sinkt die MFLOPS-Rate auf Werte um 150 ab.

Tabelle 6.10 zeigt die Performance-Werte für die Stokes-Gleichung auf adaptiven Gittern. Die Interpretation dieser Werte ist die gleiche wie für reguläre Gitter, das nicht mehr ganz perfekte Mehrgitterverhalten lässt sich vermutlich auch auf die Verwendung der Uzawa-Druckkorrektur zurückführen.

Auflösung	Freiheitsgrade	Laufzeit	Iterationen	L2-Hit-Rate	MFLOPS
27×27	744	0,34 s	168	99,92%	142,80
81×81	7.144	2,9 s	168	98,87%	146,10
243×243	65.708	25,95 s	169	98,87%	148,47
729×729	595.692	228,91 s	169	98,80%	151,49

Tabelle 6.9: Performance-Werte für die Stokes-Gleichung auf dem Einheitsquadrat für reguläre Gitter.

Auflösung	Freiheitsgrade	% reg. Gitter	Zeit	Iter	L2-Hit-Rate	MFLOPS
27×27	298	40,05	0,30 s	249	99,74%	134,40
81×81	1.140	15,95	1,25 s	286	99,42%	137,88
243×243	3.800	5,78	4,18 s	287	99,20%	139,46
729×729	13.840	2,3	14,89 s	296	99,12%	141,59

Tabelle 6.10: Performance-Werte für die Stokes-Gleichung auf dem Einheitsquadrat für adaptive Gitter.

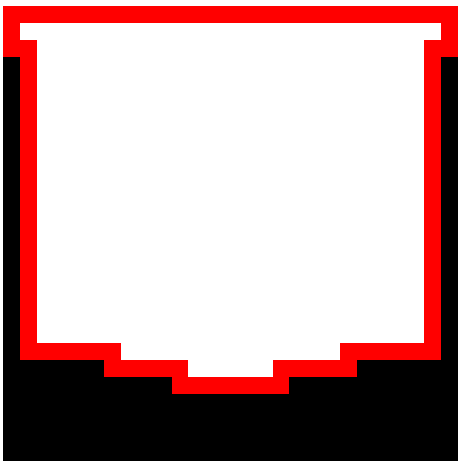

Abbildung 6.16: Eingabebild für `peano2d` zur Erzeugung eines Rechengitters für eine eingebettete Geometrie.

6.4.2 Die Stokes-Gleichung auf eingebetteten Geometrien

Weiterhin haben wir die Stokes-Gleichung auf komplizierteren Geometrien gerechnet. Wir haben dazu an die „Antriebslinie“ von $(0,1)$ nach $(1,1)$ einfach zusammenhängende, in das Einheitsquadrat eingebettete, Gebiete Ω angehängt. Abbildung 6.16 zeigt das `peano2d`-Eingabebild für ein 27×27-Gitter. In Abbildung 6.17 ist die Lösung für diese Geometrie auf einem adaptiven 27×27-Gitter nach 343 Iterationen dargestellt.

Tabelle 6.11 zeigt die relevanten Performance-Werte für eingebettete Geometrien für die Stokes-Gleichung. Auch in diesem Fall werden sehr gute Werte für die Performance im L2-Cache erzielt. Bei gleicher Auflösung des Randes und damit gleicher Geometrie zeigt sich wieder das gewünschte Mehrgitterverhalten in Form von nahezu gleichen Löseriterationen für adaptive und reguläre Gitter.

Auflösung	Freiheitsgrade	% reg. Gitter	Zeit	Iter	L2-Hit-Rate	MFLOPS
27×27	542	100,0	0,75 s	346	99,74%	135,23
27×27	510	94,09	0,57 s	343	99,57%	134,66
81×81	5.374	100,0	4,48 s	300	98,92%	137,96
81×81	2.711	50,44	2,73 s	302	99,03%	134,84
243×243	48.870	100,0	39,09 s	301	98,85%	140,24
243×243	24.912	50,97	22,14 s	302	98,92%	139,28

Tabelle 6.11: Performance-Werte für die Stokes-Gleichung in eingebetteten Geometrien für reguläre und adaptive Gitter.

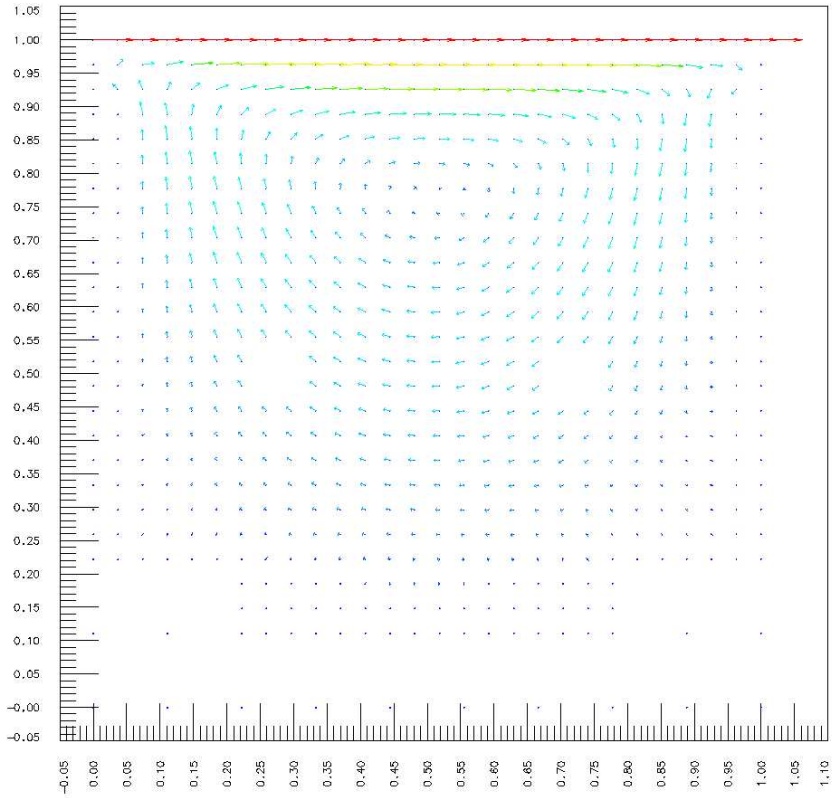

Abbildung 6.17: Lösung der Stokes-Gleichung in einer eingebetteten Geometrie auf einem 27×27-Gitter.

Insgesamt ist somit festzustellen, dass sich mit den erzielten Lösungen für die Stokes-Gleichung die Sinnhaftigkeit und Flexibilität der von uns gewählten Methodik bestätigt hat. Die Erweiterung des Programms zur Lösung der Poisson-Gleichung auf die Stokes-Gleichung hat nur wenige Tage Zeitaufwand erfordert, obwohl zusätzliche Speicher- und Zugriffsstrukturen für zellbasierte Daten (hier ist dies der Druck) implementiert werden mussten. Dies ist vor allem auf die starke Kapselung der numerisch relevanten und damit gleichungsbezogenen Programmteile und Routinen zurückzuführen. Es hat sich dabei auch erwiesen, dass zellbasierte Daten bei richtiger und konsequenter Organisation keine negativen Auswirkungen auf die Performance des Programms haben, da wir zeigen konnten, dass die niedrigeren MFLOPS-Raten vor allem in der im Vergleich zur Poisson-Gleichung einfacheren rechten Seite begründet sind. Somit hat man für künftige Entwicklungen die Flexibilität, Daten sowohl punktbasiert auf den Ecken der Zellen als auch zellbasiert zu verwenden, ohne dabei relevante Performance-Einbußen in der Speicherhierarchie befürchten zu müssen.

7 Zusammenfassung und Ausblick

Zum Schluss dieser Arbeit wollen wir nun neben einer Zusammenfassung der bisher erreichten Ergebnisse auf vorstellbare Erweiterungen eingehen. Viele Arbeiten von Kollegen und Studenten unserer Arbeitsgruppe beschäftigen sich zur Zeit mit diesen Erweiterungen und es zeigt sich bereits, dass das Potenzial unseres Ansatzes bei weitem noch nicht ausgeschöpft ist. Neben den rein wissenschaftlichen Zielen bei der Weiterentwicklung der vorgestellten Methodik beschäftigen wir uns auch mit einer möglichen Kommerzialisierung dieser Ideen. Eine Firmengründung mit dem Ziel, eine konkurrenzfähige Softwarelösung zur Strömungssimulation und zur Simulation von Fluid-Struktur-Wechselwirkungen auf den Markt zu bringen, ist durchaus denkbar.

7.1 Erfüllung der gesteckten Ziele

In der Einführung und in den Kapiteln über die informatischen und mathematischen Grundlagen wurden Ziele definiert, die im Rahmen dieser Arbeit erreicht werden sollten. Wir wollen diese nochmals darstellen und zusammenfassen:

1. **Adaptivität und Mehrgitterfähigkeit**
   Die Implementierung gestattet die Verwendung von a-priori lokal verfeinerten adaptiven Gittern. Ein additives Mehrgitterverfahren auf einem hierarchischen Erzeugendensystem konnte realisiert werden. Es wurde gezeigt, dass sich die Verwendung adaptiver Gitter nicht negativ auf die Kosten pro Freiheitsgrad (gemessen in Programmlaufzeit) auswirkt und dass bei allen verwendeten Gittern eine gute bis sehr gute Mehrgittereffizienz erzielt werden kann.

2. **Flexibilität bei den zugrunde liegenden Geometrien**
   Das Einbetten komplizierterer Geometrien in das Einheitsquadrat hat sich als sinnvolle Maßnahme erwiesen. Die Kosten des Verfahrens skalieren auch in diesem Fall im Wesentlichen linear mit der Anzahl der Freiheitsgrade. Die „leer“ durchlaufenen Zellen außerhalb des eigentlichen Rechengebiets verursachen also kaum Kosten, insbesondere da ihr Anzahl durch eine geeignete Gittervergröberung sehr gering gehalten werden kann.

3. **Effizienz im L2-Cache**
   Die Messwerte für L2-Hit-Rates bestätigen auf beeindruckende Weise die Stärke der im Rahmen dieser Arbeit entwickelten und realisierten Methodik. Allein die konsequente Umsetzung der Ideen führt zu einer extrem hohen Cache-Effizienz, weitere Optimierungsstrategien sind nicht notwendig. Die Anzahl der cache misses bleibt trotz großer Flexibilität des Programms bezüglich der Rechengitter und der Geometrien stets auf ein Minimum beschränkt.

4. **Portierbarkeit und Möglichkeiten für Erweiterungen**
   Die messbare extrem hohe Cache-Effizienz bleibt auf allen getesteten Plattformen erhalten, die Verwendung von auf die jeweilige Architektur spezialisierten Compilern führt zu Laufzeitgewinnen durch bessere Nutzung der Floating-Point-Einheiten der Prozessoren, die Cache-Effizienz ist aber schon bei der Verwendung von nicht spezialisierten Compilern wie zum Beispiel dem praktisch überall verfügbaren GNU C-Compiler gegeben. Die leicht durchführbare Erweiterung des Programms für die Poisson-Gleichung um die Fähigkeit zur Lösung der Stokes-Gleichung gibt einen ersten Eindruck von der Flexibilität der Methodik auch im Hinblick auf weitere Ziele und Ergänzungen.

Es konnte somit gezeigt werden, dass es möglich ist, den im Einführungskapitel dargestellten Zielkonflikt zwischen numerischer Effizienz und speichereffizienter Implementierung zu überwinden, ohne dabei auf andere relevante und gewünschte Eigenschaften verzichten zu müssen. Die für viele numerische Verfahren typische Begrenzung der Leistungsfähigkeit auf realen Rechnern durch den Arbeitsspeicher („memory boundedness“) ist in unserer Implementierung zumindest bezogen auf die Bandbreite und Busauslastung des Arbeitsspeichers aufgehoben.

7.2 Erweiterbarkeit und Grenzen der Implementierung

In dieser Arbeit ist der Fokus bezüglich möglicher Anwendungen stark auf die Lösung Partieller Differenzialgleichungen gesetzt. In dieser Problemklasse sollte mit der vorgestellten Methodik alles behandelbar sein, was die notwendigen Lokalitätseigenschaften erfüllt. Dazu müssen lediglich die FE-Diskretisierungen sowie die Mehrgitter-Interpolationen und -Restriktionen so formulierbar beziehungsweise zerlegbar sein, dass in einer elementorientierten Auswertung nur die Daten einer Zelle benötigt werden, egal ob es sich um punktbasierte oder zellbasierte Daten dieser Zelle handelt. Es muss bei allen Erweiterungen jedoch unbedingt vermieden werden, dass diese zellorientierte Lokalität durchbrochen wird. So würde beispielsweise ein durch ungeeignete Operatordiskretisierung notwendiger Zugriff auf Informationen von Nachbarzellen vermutlich sofort zu einem Zusammenbruch der Cache-Effizienz führen, da im Allgemeinen nicht zu erwarten ist, dass die nun benötigten Daten

„passend“ auf den Daten-Stacks platziert sind. Somit müssten im Speicher verstreute Daten gesucht werden und die strenge Linearisierung aller Zugriffe wäre verloren.

Neben der Lösung Partieller Differenzialgleichungen gibt es bereits Überlegungen für andere Anwendungsmöglichkeiten. Auf dem Gebiet der numerischen linearen Algebra wäre zum Beispiel eine vergleichbar cacheeffiziente Implementierung von Matrix-Vektor- oder Matrix-Matrix-Produkten auf Basis der Peano-Kurve und der Verwendung von Stacks als Datenstrukturen realisierbar, erste Versuche in diese Richtung finden bereits in unserer Gruppe statt.

7.3 Realisierte, laufende und geplante Weiterentwicklungen

Wie bereits mehrfach angedeutet, ist diese Arbeit eingebettet in ein derzeit laufendes Projekt an unserem Lehrstuhl, bei dem die Ideen der dargestellten Methodik auf verschiedene Aspekte des wissenschaftlichen Rechnens übertragen werden sollen.

Mein Kollege Markus Pögl beschäftigt sich derzeit mit der Realisierung vergleichbarer Programme für drei Raumdimensionen. Die bisher von ihm erzielten Ergebnisse geben zu der Hoffnung Anlass, dass sich die im Rahmen dieser Arbeit erzielten Ergebnisse auf den ungleich komplizierteren 3D-Fall vollständig übertragen lassen [19].

Andreas Krahnke versucht im Rahmen seiner Dissertation ein multiplikatives Mehrgitterverfahren zu implementieren. Wie in Kapitel 5 angedeutet, ist die bei unserer Datenorganisation gewissermaßen natürliche Wahl das additive Mehrgitterverfahren. Im Falle des multiplikativen Mehrgitterverfahrens steckt die Schwierigkeit zu einem großen Teil im Erhalt der Cache-Effizienz bei der jetzt notwendigen komplizierteren Datenbereitstellung. Ein weiterer Aspekt seiner Arbeit wird zudem noch die Realisierung von Lösungsadaption sein, also der Fähigkeit, das Rechengitter in seiner lokalen Feinheit zwischen zwei Löseriterationen auf effiziente Weise an numerische oder physikalische Erfordernisse anzupassen [14].

Im Rahmen von studentischen Arbeiten, vor allem in Form von Diplomarbeiten, sind geplant oder laufen bereits folgende Erweiterungen:

- Parallelisierung des Programms für drei Raumdimensionen. Hier lässt sich die Anordnung der Gitterzellen entlang der Raumfüllenden Kurve sehr effizient zur Generierung einer balancierten Gitterpartitionierung nutzen [28].
- Implementierung eines Lösers für die Navier-Stokes-Gleichungen.
- Verbesserte Randapproximation durch automatische zellindividuelle Operatordiskretisierungen.

- Höhere Ordnung der Operatordiskretisierungen, *hp*-Adaptivität.
- Objektorientiertes Redesign und dabei Integration aller entwickelten Features in ein Programmpaket.

Außerdem gibt es bereits Ideen zur Nutzung der vorgestellten Methodik im Zusammenhang mit der Simulation von Fluid-Struktur-Wechselwirkungen, die in der Forschung und der industriellen Praxis immer mehr an Bedeutung gewinnen. Auch hier werden voraussichtlich zukünftige Aktivitäten unserer Gruppe angesiedelt sein.

Bei allen angedeuteten Erweiterungen ist es prinzipiell nicht besonders schwierig, jeweils ein Konzept zu entwickeln, wie das entsprechende Ziel unter Beibehaltung von Flexibilität und Effizienz erreicht werden kann. Dies bestätigt erneut die Stärke der vorgestellten Kombination der Einzelteile zu einer leistungsfähigen Methodik und lässt auf weitere spannende und neue Erweiterungen, Ergänzungen und Ergebnisse hoffen.

Literaturverzeichnis

[1] Alber J., Niedermeier R.: *On Multidimensional Curves with Hilbert Property*; Theory of Computing Systems 33, 295-312 (2000);

[2] Bader, Michael: *Robuste, parallele Mehrgitterverfahren für die Konvektions-Diffusions-Gleichung*; Herbert Utz Verlag (2001);

[3] Bastian P., Hackbusch W., Wittum G.: *Additive an Multiplicative Multi-Grid – a Comparison*; Computing, 60:345-368 (1998);

[4] Bauer F.L., Goos G.: *Informatik: eine einführende Übersicht – Zweiter Teil*; Springer-Verlag Berlin - Heidelberg - New York - Tokyo, 3. Auflage (1984);

[5] Braess, Dietrich: *Finite Elemente: Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*; Springer Verlag (1997);

[6] Bramble J.H., Pasciak J.E., Vassilev A.T.: *Analysis of the inexact Uzawa algorithm for saddle point problems*; SIAM Numer. Anal. 34 (1997) pp. 1072-1092;

[7] Bramble J.H., Pasciak J.E., Vassilev A.T.: *Inexact Uzawa Algorithms for Nonsymmetric Saddle Point Problems*; Math. Comp. 69 (2000) pp.667-689;

[8] Engesser, Hermann (Hrsg.): *Duden Informatik*; 2., vollst. überarb. und erw. Auflage, Dudenverlag Mannheim, Leipzig, Wien, Zürich (1993);

[9] Frank, Anton: *Organisationsprinzipien zur Integration von geometrischer Modellierung, numerischer Simulation und Visualisierung*; Dissertationsschrift TU München (2000);

[10] Gleirscher M., Kienzl S.: *Anwendung von Peano-Kurven zur Parallelisierung von Berechnungen in der Strömungsmechanik*; IDP im Nebenfach Mathematik, Fakultät für Informatik der TU München (2003);

[11] Griebel, Michael: *Multilevelmethoden als Iterationsverfahren über Erzeugendensystemen*; Teubner (1994);

[12] Hopcroft J.E., Ullman J.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*; 3., korrigierte Auflage, Addison-Wesley (1994);

[13] Kernighan B., Ritchie D.: *Programmieren in C*; 2. Ausg. ANSI C, Hanser Verlag München, Wien (1990);

[14] Krahnke, Andreas: *Dissertation*; Technische Universität München, Fertigstellung voraussichtlich 2004;

[15] Kowarschik M., Weiß C.: *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*; Proceedings of the GI-Dagstuhl Forschungsseminar: Algorithms for Memory Hierarchies, Lecture Notes in Computer Science (LNCS), Vol. 2625, Springer (2003);

[16] Louis, Dirk: *ANSI C/C++ Kompendium*; Markt & Technik - Verlag (2000);

[17] Mehl, Miriam: *Ein interdisziplinärer Ansatz zur dreidimensionalen numerischen Simulation von Strömung, Stofftransport und Wachstum in Biofilmsystemen auf der Mikroskala*; Dissertationsschrift TU München (2001);

[18] Meier, Florian: *Effiziente numerische Behandlung von Strömungen in veränderlichen Geometrien auf kartesischen Gittern*; Dissertationsschrift TU München (2000);

[19] Pögl, Markus: *Dissertation*; Technische Universität München, Fertigstellung voraussichtlich 2004;

[20] Sagan, Hans: *Space-Filling Curves*; Springer Verlag (1994);

[21] Samelson K., Bauer F.L.: *Sequentielle Formelübersetzung*; Elektronische Rechenanlagen 1(4): 176-182 (1959);

[22] Seward J., Nethercote N., Fitzhardinge J.: *cachegrind: a cache-miss profiler*; `http://valgrind.kde.org/docs.html`;

[23] Strang G., Fix G.: *An Analysis of the finite element method*; Wellesley-Cambridge Press (1997);

[24] Stein, Reinhard: *Untersuchung und Optimierung des Cache-Verhaltens eines neuartigen Algorithmus für numerische Simulation*; System-Entwicklungs-Projekt, Fakultät für Informatik der TU München (2004);

[25] Temam, Olivier: *Investigating Optimal Local Memory Performance*; in Proc. ACM int. Conference on Architectural Support for Programming Languages and Operating Systems, San Diego, California, USA (1998);

[26] Josef Weidendorfer: *KCachegrind - Profiling Visualization*;
`http://kcachegrind.sourceforge.net`;

[27] Xu, Jinchao: *Theory of multilevel methods*; Report No. AM48, Departement of Mathematics, Pennsylvania State University (1989);

[28] Zumbusch, Gerhard: *Adaptive Multilevel Methods for Partial Differential Equations*; Habilitationsschift Universität Bonn (2001);

[29] *OpenDX: User Manual*;
`http://www.opendx.org`;

[30] *libtiff: Bibliothek für das TIFF Dateiformat*;
`http://remotesensing.org/libtiff/`;

[31] *perfmon: create powerful performance analysis tools which use the IA-64 Performance Monitoring Unit (PMU)*;
`http://www.hpl.hp.com/research/linux/perfmon/index.php4`;

[32] *IA-32 Intel(R) Architecture Optimization Reference Manual*;
`http://developer.intel.com/design/pentium4/manuals/248966.htm`;

Abbildungsverzeichnis