INSTITUT FÜR INFORMATIK
DER
TECHNISCHEN UNIVERSITÄT MÜNCHEN

Funktionale Beschreibung und Parallelisierung von Algorithmen auf dünnen Gittern

Alexander Pfaffinger

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. Jürgen Eickel

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Christoph Zenger
2. Univ.-Prof. Dr. Arndt Bode

Die Dissertation wurde am 18. 12. 1996 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18. 3. 1997 angenommen.

Funktionale Beschreibung und Parallelisierung von Algorithmen auf dünnen Gittern

Zusammenfassung

Die vorliegende Schrift gibt eine detaillierte Studie über die Parallelisierungsaspekte numerischer Algorithmen, die auf sogenannten *dünnen Gittern* arbeiten. Diese haben sich als sehr effizientes Instrument für die Darstellung von Funktionen, die Interpolation, die numerische Quadratur und die numerische Lösung partieller Differentialgleichungen erwiesen.

Als wesentliches Hilfsmittel zur Analyse und parallelen Implementierung wird zunächst die funktionale Datenflußsprache FASAN eingeführt. Mit dieser Sprache wird dann ein wichtiges Anwendungsgebiet dünner Gitter, die Finite-Elemente-Lösung partieller Differentialgleichungen, für eine und zwei Dimensionen funktional spezifiziert und die Verallgemeinerung für höhere Dimensionalität angegeben. Ziel ist dabei, die zunächst stark miteinander verwobenen Programmteile zu entzerren und die Kommunikations– und Ablaufstruktur offenzulegen.

Anhand der funktionalen Spezifikation werden die Parallelisierungsmöglichkeiten analysiert und verschiedene Lastverteilungsstrategien diskutiert. Laufzeitmessungen auf einem Netz von 16 Workstations belegen die praktische Durchführbarkeit der theoretischen Überlegungen.

Inhaltsverzeichnis

1 Einführung 1

2 Hierarchische Basen und dünne Gitter 7

2.1 Hierarchische Basen 7

2.2 Dünne Gitter 10

3 Die Helmholtzgleichung auf dünnen Gittern 15

3.1 Die Helmholtzgleichung 15

3.2 Die eindimensionalen Massenmatrizen 18

3.3 Die D-dimensionale Matrixmultiplikation 20

3.3.1 Der unidirektionale Algorithmus 21

3.3.2 Die D-dimensionale Überschußschleuder 22

3.4 Probleme der imperativen Beschreibung 24

4 FASAN 29

4.1 Zur Entwicklung funktionaler Sprachen 29

4.2 Konzepte und Syntax von FASAN 31

4.3 Kabelbäume 34

4.3.1 Divide&Conquer: das Problem der lokalen Daten 35

4.3.2 Maximale Parallelität und minimale Kommunikation 37

5 Funktionale Beschreibung des Helmholtzlösers 39

5.1 Die eindimensionale Helmholtzgleichung 39

5.2 Die zweidimensionale Helmholtzgleichung 43

5.3 Verallgemeinerung auf beliebige Dimensionalität 47

5.3.1 Die D-dimensionale Up-Funktion 48

5.3.2 Die D-dimensionale Down-Funktion 52

5.3.3 Optimierung der Kommunikation 55

6 Lastverteilung **59**

6.1 Kodierung dünner Gitter 59

6.2 Verteilung auf 2^r Prozessoren 62

6.3 Der unidirektionale Algorithmus auf De-Bruijn-Netzen 66

6.4 Zweidimensionale dünne Gitter auf Kantenhypercuben 71

6.5 Dualisierung des Hypercubes 75

6.6 Hypercube–Topologien für D-dimensionale Gitter 77

6.7 Lastverteilungsstrategien für adaptive dünne Gitter 78

7 Leistungsmessungen **81**

7.1 Kosten der funktionalen Beschreibung 81

7.2 Kosten der D-dimensionalen Beschreibung 82

7.3 Paralleler Speedup 83

7.4 Zusammenfassung 86

8 Schlußbetrachtung **87**

Literatur **91**

1 Einführung

Die vorliegende Arbeit handelt von Algorithmen auf sogenannten *dünnen Gittern*, die hier stellvertretend für die wichtige Klasse der *Divide-et-Impera*-Algorithmen stehen. Diese Algorithmen sollen *parallelisiert* werden, d.h. sie sollen in Teile zerlegt werden, die gleichzeitig auf mehreren Prozessoren zur Ausführung kommen. Als zentrales Mittel zur Beschreibung und Implementierung der parallelen Algorithmen dient uns eine neue *funktionale* Sprache namens FASAN. Warum lohnt es sich, Dünngitteralgorithmen zu untersuchen, weshalb müssen diese parallelisiert werden, welche Probleme treten dabei auf, und wie helfen uns dabei funktionale Sprachen? Diese Fragen wollen wir im folgenden beantworten.

Partielle Differentialgleichungen bilden die mathematische Grundlage zur Beschreibung zahlreicher technischer und natürlicher Phänomene, etwa im Bereich der Statik, der Wärmeleitung oder der Strömungsmechanik. Bei der numerischen Behandlung partieller Differentialgleichungen wird die Lösungsfunktion an diskreten Stützstellen (Gitterpunkten) des betreffenden Gebietes approximiert. Die Zahl der Punkte ist ein Maß für den Rechen- und Speicheraufwand der numerischen Simulation.

Oft benötigt man zur realistischen und detailgetreuen Modellierung eine sehr feine Auflösung des Gitters, wodurch die Zahl der Punkte in die Höhe schnellt. Die Lage verschärft sich bei der Untersuchung drei- oder höherdimensionaler Probleme, die, wie etwa im Bereich der Strömungsmechanik, auch eine zeitliche Dynamik aufweisen können. Sollen dann die Berechnungen noch in Echtzeit modelliert werden, so werden unerfüllbare Anforderungen an die vorhandenen Rechenkapazitäten gestellt.

Die in [Zen91] entworfene Methode der *dünnen Gitter* stellt nun ein mächtiges Instrument zur Reduktion der Gitterpunkte dar. Erster Schritt ist dabei der Übergang zu sogenannten *hierarchischen Basen*. Zenger entdeckte, daß bei der multilinearen Approximation von Funktionen auf rechteckigen Gittern ein Großteil der hierarchischen Basisfunktionen nur wenig zur Approximationsgenauigkeit beiträgt, wenn einfache Glattheitsbedingungen an die zu approximierende Funktion erfüllt sind. Durch Weglassen dieser Basisfunktionen gelangt man zu dünnen Gittern.

Bei einer multilinearen Stützpunktbasis auf einem D-dimensionalen Würfel der Maschenweite $h = 2^{-n}$ ergeben sich $O(2^{nD})$ Basisfunktionen. Wie in [Bun92] gezeigt wurde, kann nach einer Basistransformation auf eine hierarchische Basis die Zahl der Basisfunktionen auf $O(2^n \cdot n^{D-1})$ dramatisch reduziert werden, ohne daß die erzielte Genauigkeit der Lösungsdarstellung signifikant beeinträchtigt würde. Der Approximationsfehler in der L_2- bzw. L_∞-Norm verschlechtert sich nämlich lediglich von $O(4^{-n})$ auf $O(4^{-n} \cdot n^{D-1})$.

Dünne Gitter können wegen ihres rekursiven hierarchischen Aufbaus überdies hervorragend adaptiv erzeugt werden, so daß die hohe Auflösung an Problemstellen des untersuchten Gebiets nur lokal durchgeführt wird. Dadurch läßt sich die Zahl der Gitterpunkte noch einmal deutlich reduzieren.

Trotz dieser starken Reduktion der Gitterpunkte durch (adaptive) dünne Gitter ist der Berechnungs- und Speicheraufwand gerade bei realistischen Problemstellungen noch außerordentlich hoch. Abhilfe verspricht hier die *Parallelisierung* der Algorithmen, d.h. die verteilte Bearbeitung auf mehreren Prozessoren. Als Zielarchitektur bieten sich v.a. Workstationnetze an, die im normalen Gebrauch ohnehin nur zu einem Bruchteil ihrer Kapazitäten ausgelastet sind. Da sie in vielen Bereichen oft schon zahlreich vorhanden sind, bieten sie eine kostengünstige Alternative zu dedizierten Parallelrechnern.

Unabhängig von der jeweiligen Zielarchitektur ist das Erreichen eines optimalen Speedups, also der k-fachen Beschleunigung eines Programms beim Einsatz von k Rechnern, aus verschiedenen Gründen schwierig bis unmöglich. So müßte zunächst einmal der Algorithmus ein genügend großes Parallelisierungspotential bieten, also idealerweise in k völlig unabhängige Berechnungsteile zerfallen. Normalerweise existieren aber komplexe Abhängigkeitsbeziehungen zwischen diesen Teilen, die immer wieder zu einem gegenseitigen Warten führen und zudem Datenaustausch über Rechnergrenzen bewirken. Gerade diese externe Kommunikation ist aber in der Regel sehr teuer. Im Extremfall dominieren sogar die Kommunikationszeiten auf einem leistungsschwachen Netz die Laufzeit. Deswegen muß man bei der Parallelisierung stets darauf bedacht sein, die Abhängigkeitsstruktur möglichst stark zu vereinfachen, damit die Rechenknoten weitgehend unabhängig und mit wenig Datenaustausch arbeiten können.

Für die Parallelisierung eines Algorithmus' ist es entscheidend, inwieweit er in Teile zerlegt werden kann, die unabhängig voneinander berechnet werden können. Mit herkömmlichen imperativen Programmiersprachen (C, FORTRAN, Pascal, usw.), wie sie fast ausschließlich für numerische Aufgabenstellungen verwendet werden, ist dies oft nur schwierig zu bewältigen. Denn hier wird eine feste Abarbeitungsreihenfolge von Befehlen festgelegt, die einen globalen Speicherzustand verändern. So kann es durch globale Variablen oder bedingte Sprünge zu komplexen Abhängigkeiten zwischen den Programmteilen kommen. Eine effiziente *automatische* Parallelisierung – etwa durch Compiler – ist daher nahezu unmöglich.

Deshalb werden häufig parallele Erweiterungen der Standardsprachen benutzt, mit denen der Programmierer die parallele Struktur beschreiben muß. Am effizientesten sind dabei sogenannte *Message-Passing*-Modelle (PVM [BDG$^+$94], MPI [CGH94, DOSW95]), in denen der Benutzer exakt kodieren muß, welche Module zu welchen anderen Modulen Daten schicken müssen. Doch der Zwang zur *expliziten* Festlegung der parallelen Ablaufreihenfolge steht in vielen Fällen quer zur logischen Programmgliederung. Außerdem ist dieses Vorgehen fehleranfällig, so daß es leicht zu gegenseitigem Warten auf Daten (*deadlock*) kommen kann.

Hier bietet der funktionale Ansatz große Vorteile gegenüber imperativer Programmierung. Die Funktionen haben klare Schnittstellen, denn sie beeinflussen sich nur über ihre Ein- und Ausgabeparameter. Damit können sequentielle Abhängigkeiten und mögliche parallele Abläufe in funktionalen Programmen automatisch erkannt werden. Funktionale Sprachen zeichnen sich dadurch aus, daß sie allein auf Ausdrücken aufbauen. Ausdrücke sind Daten und Funktionsanwendungen auf Daten. Eine Funktionsanwendung kann semantisch äquivalent durch ihre Ergebnisdaten ersetzt werden. Die Berechnung eines Ausdrucks geschieht ohne Seiteneffekte und ohne Verwendung eines globalen Zustands. Die Reihenfolge der Teilberechnungen ist nicht festgelegt, sie wird allein durch die Datenabhängigkeit beschrieben.

Ein einfaches Beispiel aus dem Gebiet der partiellen Differentialgleichungen soll die Problematik verdeutlichen. Wenn wir die zweidimensionale Laplacegleichung $\Delta u = 0$ auf dem Einheitsquadrat durch ein lexikographisches Gauß-Seidel-Verfahren auf einem äquidistanten Gitter lösen, so werden in jeder Iteration die Näherungswerte jedes Gitterpunktes p durch folgende Zuweisung berechnet (vgl. Abb. 1.1):

$$p_{neu} := \frac{1}{4}(w_{neu} + n_{neu} + e_{alt} + s_{alt}) \tag{1.1}$$

Dabei beschreiben w, n, e, s die Funktionswerte der Nachbarpunkte in den vier Himmelsrichtungen. Im Gegensatz zu anderen Relaxationsverfahren werden hier vom Westen und Norden bereits die aktuellen Funktionswerte benutzt, aus Osten und Süden die alten aus der vergangenen Iterierten. Durch die mathematische Vorschrift (1.1) ist die optimale Abarbeitungsreihenfolge und die mögliche Parallelisierung bereits vorgegeben. Abbildung 1.1 (links) zeigt, welche Punkte in welchem Zeitschritt berechnet werden können. Dabei ist zu beachten, daß die (nicht eingezeichneten) Randpunkte stets zur Verfügung stehen. Die Pfeile deuten an, wie sich die Gitterpunkte gegenseitig beeinflussen.

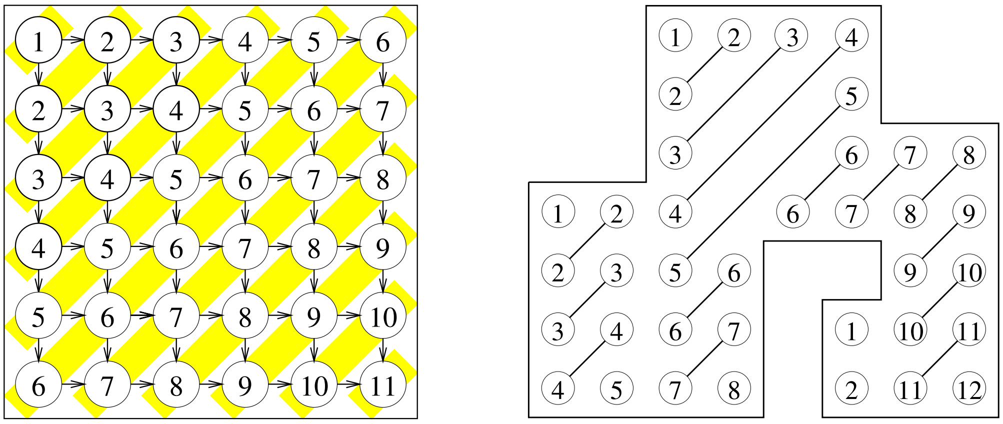


Abbildung 1.1: Links: *Optimale Abarbeitungsreihenfolge bei einer Gauß-Seidel Iteration. Die Pfeile beschreiben den Datenfluß zwischen den Gitterpunkten.* Rechts: *Ein Gebiet mit komplexer Geometrie. Punkte auf den Diagonalen können gleichzeitig bearbeitet werden.*

In einer imperativen Programmiersprache würde man nun herkömmlicherweise die Iteration durch zwei ineinandergeschachtelte Schleifen über die Zeilen und über die Spalten beschreiben. Aber weder die Zeilen noch die Spalten sind als ganzes betrachtet voneinander unabhängig, so daß man sie sequentiell berechnen muß und auf diese Weise das gesamte Parallelisierungspotential verschenkt. Die parallele Komplexität sinkt von $O(n)$ auf $O(n^2)$, wenn n die Anzahl der Gitterpunkte in einer Dimension angibt.

Die explizite Beschreibung eines Algorithmus', der die voneinander unabhängigen Gitterpunkte in den Diagonalen parallel aktualisiert, ist dagegen schon merklich aufwendiger.

Wenn wir statt eines Quadrats ein komplizierter gestaltetes Gebiet behandeln wollen (vgl. Abb. 1.1 rechts), so ist die explizite Formulierung der optimalen parallelen Berechnungsreihenfolge nahezu unmöglich.

Wünschenswert wäre dagegen, wenn der Programmierer nur die Abhängigkeiten zwischen den Daten angibt (wie es in der mathematischen Beschreibung der Fall ist) und die Berechnungsreihenfolge sich automatisch nach diesen Abhängigkeiten richtet. Diese Nähe zur mathematischen Spezifizierung numerischer Programme ist jedoch nur bei funktionalen Sprachen zu finden.

In dieser Arbeit wird daher eine funktionale Datenflußsprache namens FASAN eingeführt und ihre Eignung zur Parallelisierung von Dünngitteralgorithmen unter Beweis gestellt. Dies geschieht exemplarisch für die D-dimensionale Helmholtzgleichung. Im einzelnen gliedert sich die Arbeit wie folgt.

Im anschließenden Kapitel 2 geben wir zunächst eine kurze Einführung in die Theorie hierarchischer Basen und dünner Gitter. Dabei wird die für die späteren Teile der Arbeit notwendige Terminologie bereitgestellt.

Kapitel 3 widmet sich der Diskretisierung und algorithmischen Behandlung der D-dimensionalen Helmholtzgleichung auf dünnen Gittern mit Hilfe der Finite-Elemente-Methode. Es wird gezeigt, daß die optimale parallele Komplexität eines Iterationsschritts, der im wesentlichen aus der Multiplikation mit der Steifigkeitsmatrix der Helmholtzgleichung besteht, bei fester Dimension nur linear von der Tiefe T des dünnen Gitters abhängt. Bei dem Entwurf eines Programms, das die maximale Parallelität beschreiben kann, stößt man aber in herkömmlichen imperativen Sprachen auf Schwierigkeiten.

Deshalb wird in Kapitel 4 die im Rahmen dieser Dissertation entworfene funktionale Datenflußsprache FASAN vorgestellt. Sie dient im folgenden einerseits als effektives Beschreibungsmittel der parallelen Struktur numerischer Algorithmen, zum anderen erlaubt sie auch die effiziente Implementierung der parallelen Programme auf einem Workstationnetz. Wesentliches Merkmal von FASAN ist das sogenannte Kabelbaumprinzip, mit dem es gelingt, Daten ohne Umwege und unnötige Synchronisationen vom Sender zum Endempfänger zu leiten, obwohl syntaktisch die Daten über diverse Zwischenknoten des Datenflußgraphen gehen können.

FASAN dient uns in Kapitel 5 zur funktionalen Beschreibung der Multiplikation mit der Steifigkeitsmatrix der Finite-Elemente-Diskretisierung, dem Kern eines Iterationsschritts zur Behandlung der Helmholtzgleichung. Zunächst werden spezielle Programme für ein- und zweidimensionale Probleme vorgestellt. Daraus wird dann ein universeller lgorithmus für beliebig dimensionale Probleme abgeleitet, bei dem die Dimensionalität als Parameter übergeben wird. Bei allen Programmen sorgen die Kabelbaum- und Datenflußsemantik von FASAN dafür, daß die maximale Parallelität erzielt werden kann.

Neben der sprachlichen Beschreibung sind auch Fragen der Lastverteilung und der Kommunikation wichtige Aspekte bei der Parallelisierung. Deshalb sind graphentheoretische Untersuchungen von parallelen Dünngitteralgorithmen und deren Kommunikations- bzw.

Aufrufstruktur Gegenstand des 6. Kapitels. Für den zweidimensionalen Fall wird eine in der Zahl der Rechner skalierbare Netztopologie vorgestellt, in der sich der Kommunikationsgraph mit konstanter Dilation 2 einbetten läßt.

Kapitel 7 präsentiert abschließend Laufzeitmessungen der parallelen FASAN-Implementierung auf einem Netz aus 16 Workstations.

Alle wesentlichen Ergebnisse dieser Arbeit werden zwar im Kontext dünner Gitter erläutert, als Voraussetzung wird aber lediglich benutzt, daß der der Parallelisierung zugrundeliegende Abhängigkeitsgraph aus einem D-dimensionalen Produkt von Binärbäumen besteht. In diesem Sinne sind die Resultate auf alle Divide-et-Impera-Algorithmen übertragbar, denen eine solche Struktur zugrundeliegt.

Ich möchte mich an dieser Stelle ausdrücklich bei meinem Doktorvater Prof. Christoph Zenger bedanken, der mir die Aufgabenstellung überlassen und mir mit seinen Anregungen und Ideen immer wieder geholfen hat. Außerdem bedanke ich mich bei Thomas Dornseifer für die Bereitstellung der sequentiellen Vergleichsprogramme und der zahlreichen kleinen IDL-Skripten. Ralf Ebner hat nicht nur für die Implementierung von FASAN gesorgt, sondern war auch immer wieder eine große Hilfe bei der Beseitigung kleinerer und größerer Tücken des Compilers. Bei beiden Herren bedanke ich mich zudem für die Durchsicht des Manuskripts. Hans Bungartz hat wesentlich zur Verbesserung der Darstellung beigetragen und mich vor so manchem Lapsus bewahrt. Bei Stefan Zimmer, Andreas Paul, Thomas Grauschopf und Thomas Bonk möchte ich mich für die Hilfe bei soft- und hardwaretechnischen Problemen bedanken. Letzterer fungierte in der Entwicklungsphase von FASAN außerdem als ein wichtiger Ratgeber. Insgesamt hat mir das außerordentlich gute Klima am Lehrstuhl, das von großer Hilfsbereitschaft und Diskussionsfreudigkeit der Kollegen geprägt ist, über so manche Durststrecke hinweggeholfen. Schließlich möchte ich mich bei meiner Freundin Martina Kappeller und bei Frau Uscha Dietrich bedanken, die mir in der Schlußphase dieser Arbeit eine wertvolle moralische Unterstützung waren. Für die materielle Unterstützung in den letzten drei Jahren bedanke ich mich bei der Siemens AG und bei FORTWIHR.

2 Hierarchische Basen und dünne Gitter

In diesem Kapitel wollen wir die für die späteren Kapitel notwendige Theorie und Terminologie im Zusammenhang mit der Approximation von Funktionen mehrerer unabhängiger Veränderlicher auf dünnen Gittern in groben Zügen zusammenfassen. Zunächst erklären wir den Unterschied zwischen nodaler und hierarchischer Basis bei der Funktionsapproximation im eindimensionalen Fall. Die Verallgemeinerung auf D Dimensionen erreichen wir durch Produkte der eindimensionalen Basisfunktionen. Im darauffolgenden Abschnitt zeigen wir dann den Übergang von vollen zu dünnen Gittern und fassen deren wichtigste Eigenschaften zusammen. Für eine ausführlichere Beschreibung der in diesem Kapitel präsentierten Ergebnisse verweisen wir auf die Arbeiten [Bun92] und [Bal94].

2.1 Hierarchische Basen

Bei der Darstellung einer D-dimensionalen reellen Funktion $u : [0,1]^D \to I\!R$ im Rechner werden üblicherweise die Werte an den Punkten eines regelmäßigen Gitters der Maschenweite $h_n = 2^{-n}$ betrachtet und dadurch ein stetiger stückweise D-linearer Interpolant $u_n^{(D)}$ festgelegt. Wir gehen im folgenden davon aus, daß u auf dem Rand verschwindet. Dies stellt keine wesentliche Einschränkung dar, erhöht aber die Übersichtlichkeit der Darstellung beträchtlich. Der Raum $\mathcal{S}_n^{(D)}$ der stückweise stetigen D-linearen Interpolanten $u_n^{(D)}$ besitzt demnach die Dimension $(2^n - 1)^D$.

Im eindimensionalen Fall wird als Standardbasis des $\mathcal{S}_n^{(1)}$ üblicherweise die sogenannte Stützpunkt- oder nodale Basis $\mathcal{B}_n$ gewählt. Sie besteht aus stückweise linearen Funktionen, die jeweils an einem Gitterpunkt den Wert 1 annehmen und an allen anderen Gitterpunkten verschwinden. Ein Beispiel zeigt Abbildung 2.1.

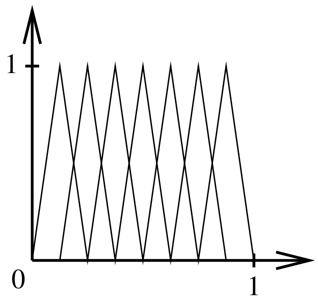


Abbildung 2.1: *Stützpunktbasis* $\mathcal{B}_3$

Die hierarchische Basis $\mathcal{H}_n$ wird hingegen in der folgenden Weise rekursiv definiert:

$$\mathcal{H}_1 \;:=\; \mathcal{T}_1 := \mathcal{B}_1,$$

$$\mathcal{H}_i \;:=\; \mathcal{H}_{i-1} \cup \mathcal{T}_i = \bigcup_{j=1}^{i} \mathcal{T}_j \quad (i > 1). \tag{2.1}$$

wobei $\mathcal{T}_i$, $i > 1$, aus denjenigen Basisfunktionen von $\mathcal{B}_i$ besteht, die auf dem zu $\mathcal{B}_{i-1}$ gehörigen Gitterpunkten verschwinden. Abbildung 2.2 zeigt die Basisfunktionen aus $\mathcal{H}_3$.

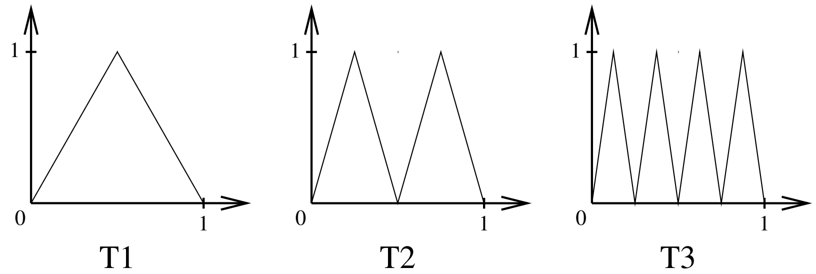


Abbildung 2.2: *Hierarchische Basis* $\mathcal{H}_3$

Im D-dimensionalen Fall stützen wir uns auf Produkte eindimensionaler Basisfunktionen ab. Die Stützpunktbasen $\mathcal{B}_n^{(D)}$ und die hierarchische Basen $\mathcal{H}_n^{(D)}$ werden also folgendermaßen konstruiert:

$$\mathcal{B}_n^{(D)} \;:= \Big\{ \varphi_{k_1,\ldots,k_D}(x_1,\ldots,x_D) \;:= \prod_{j=1}^{D} \varphi_{k_j}(x_j) : \varphi_{k_j} \in \mathcal{B}_n \Big\}, \tag{2.2}$$

$$\mathcal{H}_n^{(D)} \;:= \Big\{ \tilde{\varphi}_{k_1,\ldots,k_D}(x_1,\ldots,x_D) \;:= \prod_{j=1}^{D} \tilde{\varphi}_{k_j}(x_j) : \tilde{\varphi}_{k_j} \in \mathcal{H}_n \Big\}, \tag{2.3}$$

Analog zum eindimensionalen Fall läßt sich $\mathcal{H}_n^{(D)}$ in disjunkte Teilmengen $\mathcal{T}_{i_1,\ldots,i_D}$ zerlegen (vgl. (2.1)):

$$\mathcal{H}_n^{(D)} = \bigcup_{i_1=1}^{n} \cdots \bigcup_{i_D=1}^{n} \mathcal{T}_{i_1,\ldots,i_D} \tag{2.4}$$

mit

$$\mathcal{T}_{i_1,\ldots,i_D} := \Big\{ \tilde{\varphi}_{k_1,\ldots,k_D}(x_1,\ldots,x_D) \in \mathcal{H}_n^{(D)} : \tilde{\varphi}_{k_j}(x_j) \in \mathcal{T}_{i_j}, \quad \forall j \in \{1,\ldots,D\} \Big\} \quad .$$

Für die Zahl der Basisfunktionen folgt aus $|\mathcal{T}_i| = 2^{i-1}, i \geq 1$, nach dieser Definition sofort

$$|\mathcal{T}_{i_1,\ldots,i_D}| = 2^{i_1+\ldots+i_D-D}, \qquad i_1,\ldots,i_D \geq 1 \quad . \tag{2.5}$$

Den von $\mathcal{T}_{i_1,\ldots,i_D}$ aufgespannten Teilraum von $\mathcal{S}_n^{(D)}$ bezeichnen wir mit $T_{i_1,\ldots,i_D}$. Es gilt

$$\mathcal{S}_n^{(D)} = \bigoplus_{i_1=1}^{n} \cdots \bigoplus_{i_D=1}^{n} T_{i_1,\ldots,i_D} \tag{2.6}$$

Zur Veranschaulichung zeigen wir für den zweidimensionalen Fall die Träger der Basisfunktionen $\mathcal{T}_{i,j}$ aus $\mathcal{H}_4^{(2)}$ in Abbildung 2.3 in Form eines quadratischen Schemas. Oben links ist die einzige Basisfunktion des Teilraumes $T_{1,1}$ dargestellt, deren Träger das gesamte Gebiet überdeckt. Die Träger der Basisfunktionen werden beim Übergang von $T_{i,j}$ auf $T_{i+1,j}$ in x_1-Richtung halbiert, beim Übergang auf $T_{i,j+1}$ in x_2-Richtung.

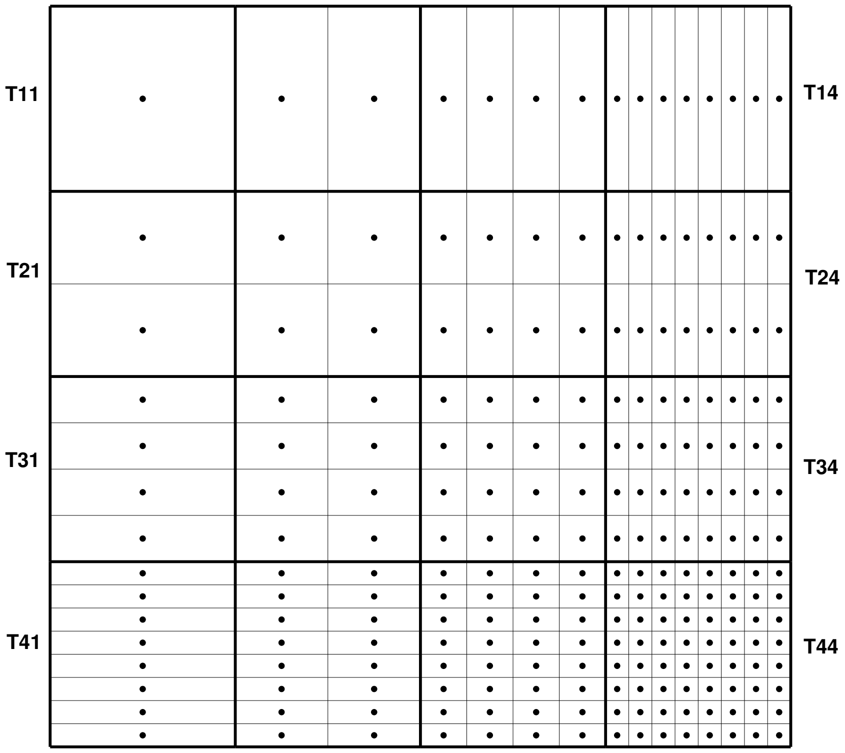


Abbildung 2.3: *Träger der hierarchischen Basisfunktionen in* $\mathcal{H}_4^{(2)}$.

In der Graphik ist zudem der Mittelpunkt des Trägers jeder Basisfunktion φ, der sogenannte *Basispunkt* $bp(\varphi)$ eingezeichnet. Durch Überlagerung aller Basispunkte in $\mathcal{H}_n^{(D)}$ erhält man ein Gitter der Maschenweite h_n.

Motiviert durch (2.5) nennen wir

$$t := t(i_1, \ldots, i_D) := i_1 + \ldots + i_D - D \tag{2.7}$$

die *Tiefe* von $\mathcal{T}_{i_1,\ldots,i_D}$ und

$$t_j := i_j - 1 \tag{2.8}$$

die *Tiefe* von $\mathcal{T}_{i_1,\ldots,i_D}$ in Richtung j.

Die Tiefe einer Basisfunktion $\tilde{\varphi}_{k_1,\ldots,k_D} \in \mathcal{T}_{i_1,\ldots,i_D}$ und die Tiefe des aufgespannten Teilraumes $T_{i_1,\ldots,i_D}$ soll der Tiefe von $\mathcal{T}_{i_1,\ldots,i_D}$ entsprechen. Wir fassen die Basisfunktionen einer festen Tiefe t zusammen:

$$\mathcal{T}_t^{(D)} := \bigcup_{i_1+\ldots+i_D-D=t} \mathcal{T}_{i_1,\ldots,i_D}$$

In Abbildung 2.3 liegen die $\mathcal{T}_{i_1,\ldots,i_D}$ gleicher Tiefe auf den Diagonalen des quadratischen Schemas.

Wir wollen an dieser Stelle noch einige Bezeichnungen einführen. Zunächst vereinfachen wir die Schreibweise durch Multiindizes: $\tilde{\varphi}_K$ bezeichne im folgenden $\tilde{\varphi}_{k_1,\ldots,k_D}$. Sei nun dieses $\tilde{\varphi}_K$ eine Basisfunktion aus $\mathcal{T}_{i_1,\ldots,i_D}$. Wir nennen $\tilde{\varphi}_L$ aus $\mathcal{T}_{j_1,\ldots,j_D}$ einen *hierarchischen Sohn* von $\tilde{\varphi}_K$, wenn der Träger von $\tilde{\varphi}_L$ durch Halbierung des Trägers von $\tilde{\varphi}_K$ entsteht. In diesem Fall gilt insbesondere:

$$\exists m \in \{1, \ldots, D\} : \quad (j_1, \ldots, j_D) = (i_1, \ldots, i_{m-1}, i_m + 1, i_{m+1}, \ldots, i_D).$$

$\tilde{\varphi}_L$ heißt dann genauer *hierarchischer Sohn (in Richtung m)* von $\tilde{\varphi}_K$. Diese Relationen induzieren partielle Ordnungen $<$ ($<_m$) auf der Menge der Basisfunktionen. Wir nennen $\tilde{\varphi}_K$ *hierarchisch höher (in Richtung m)* als $\tilde{\varphi}_L$, wenn $\tilde{\varphi}_K > \tilde{\varphi}_L$ (bzw. $\tilde{\varphi}_K >_m \tilde{\varphi}_L$) gilt.

Im folgenden gehen wir davon aus, daß die eindimensionalen Basisfunktionen konsistent zu ihrer hierarchischen Ordnung numeriert sind, d.h.:

$$\tilde{\varphi}_i < \tilde{\varphi}_k \Longrightarrow i > k. \tag{2.9}$$

Der Graph der partiellen Ordnungsrelation ist ein Binärbaum, dessen Wurzel die einzige Basisfunktion aus $\mathcal{T}_1$ bildet. Bei Dünngitteralgorithmen erweist sich der Binärbaum auch als eine vorteilhafte Datenstruktur, während bei Algorithmen, die mit der Standardbasis rechnen, in der Regel flache Felder zur Abspeicherung der Knotenwerte dienen.

Für ein festes $\tilde{\varphi}_i$ seien weiterhin folgende Indexmengen definiert:

$$\{k \supset i\} := \{k : \tilde{\varphi}_k > \tilde{\varphi}_i\}$$

und

$$\{k \subset i\} := \{k : \tilde{\varphi}_k < \tilde{\varphi}_i\}.$$

Letztere gehört zu dem Teilbaum, dessen Wurzel $\tilde{\varphi}_i$ bildet. Mit $\{i\}$ sei die Menge der Indizes aller Funktionen der betrachteten Basis bezeichnet.

Eine Basisfunktion $\tilde{\varphi}_{k_1,\ldots,k'_m,\ldots,k_D}$ heißt *hierarchischer Nachbar in Richtung m* von $\tilde{\varphi}_{k_1,\ldots,k_m,\ldots,k_D}$, wenn der Basispunkt von $\tilde{\varphi}_{k'_m}$ auf einem der beiden Randpunkte des Trägers von $\tilde{\varphi}_{k_m}$ liegt.

Zu jedem $\tilde{\varphi}_K$ gibt es in jeder Richtung m je zwei hierarchische Söhne, wenn die Tiefe von $\tilde{\varphi}_K$ in Richtung m nicht maximal in der betrachteten Basis ist, und zwei hierarchische Nachbarn, außer die Randpunkte des Trägers von $\tilde{\varphi}_{k_m}$ sind 0 oder 1. Einer dieser beiden hierarchischen Nachbarn ist der hierarchische Vater von $\tilde{\varphi}_K$. Der andere ist ebenfalls hierarchisch höher als $\tilde{\varphi}_K$.

2.2 Dünne Gitter

Im vorigen Abschnitt haben wir zwei Basen für den Raum $\mathcal{S}_n^{(D)}$ kennengelernt. Ihr Hauptunterschied besteht darin, daß im Gegensatz zu $\mathcal{B}_n^{(D)}$ die Träger der hierarchischen Basisfunktionen in $\mathcal{H}_n^{(D)}$ unterschiedliches Volumen besitzen. Die Basen $\mathcal{T}_{i_1,\ldots,i_D}$ der verschiedenen Teilräume $T_{i_1,\ldots,i_D}$ sind außerdem unterschiedlich mächtig.

Die Mächtigkeit einer Basis, d.h. die Zahl der in ihr enthaltenen Basisfunktionen, ist ein Maß für den damit verbundenen Speicheraufwand. Das Volumen des Trägers einer Basisfunktion steht dagegen in engem Zusammenhang zu ihrem Informationsgehalt bei der Funktionsapproximation. Durch Weglassen teurer aber unwichtiger Teilräume aus $\mathcal{S}_n^{(D)}$ gelangen wir zu den dünnen Gittern.

Wie wir in (2.5) gesehen haben, ist die Mächtigkeit einer Basis $\mathcal{T}_{i_1,\ldots,i_D}$ der Tiefe t gleich 2^t. Der Beitrag einer Basisfunktion zur Approximationsgenauigkeit hängt ebenfalls von deren Tiefe ab. Seien nämlich die Gewichte $\tilde{u}_I$ der Basisfunktionen $\tilde{\varphi}_I$ durch

$$u_n^{(D)} = \sum_{\tilde{\varphi}_I \in \mathcal{H}_n^{(D)}} \tilde{u}_I \tilde{\varphi}_I \in \mathcal{S}_n^{(D)} \tag{2.10}$$

festgelegt und sei außerdem die Seminorm

$$|u|_\infty := \left\| \frac{\partial^{2D}}{\partial x_1^2 \, \partial x_2^2 \ldots \partial x_D^2} u \right\|_\infty \tag{2.11}$$

existent und stetig, so gilt nach [Bun92]:

$$|\tilde{u}_I| \leq 2^D \cdot |u|_\infty \cdot 4^{-t}. \tag{2.12}$$

Folglich ist bei Basisfunktionen gleicher Tiefe ein annähernd gleiches Gewicht zu erwarten. Außerdem fallen die Gewichte mit zunehmender Tiefe exponentiell ab.

Die Berücksichtigung von Teilräumen großer Tiefe ist also in doppelter Hinsicht ungeschickt. Einerseits erhöhen sie die Zahl der Basisfunktionen beträchtlich, und andererseits tragen diese einen sehr niedrigen Informationsgehalt. Das Weglassen solcher Teilräume führt uns auf die sogenannte *Dünngitterbasis*:

$$\tilde{\mathcal{H}}_n^{(D)} := \bigcup_{i_1+\ldots+i_D-D<n} \mathcal{T}_{i_1,\ldots,i_D} = \bigcup_{t=0}^{n-1} \mathcal{T}_t^{(D)} \tag{2.13}$$

Für den von ihr aufgespannten Approximationsraum $\tilde{\mathcal{S}}_n^{(D)}$ gilt demnach:

$$\tilde{\mathcal{S}}_n^{(D)} := \langle \tilde{\mathcal{H}}_n^{(D)} \rangle = \bigoplus_{i_1+\ldots+i_D-D<n} T_{i_1,\ldots,i_D} \tag{2.14}$$

In Abbildung 2.4 sind zur Verdeutlichung die Träger der Basisfunktionen in $\tilde{\mathcal{H}}_4^{(2)}$ weiß unterlegt. Die gegenüber der Definition von $\mathcal{H}_4^{(2)}$ weggefallenen Basisfunktionen sind dagegen grau unterlegt.

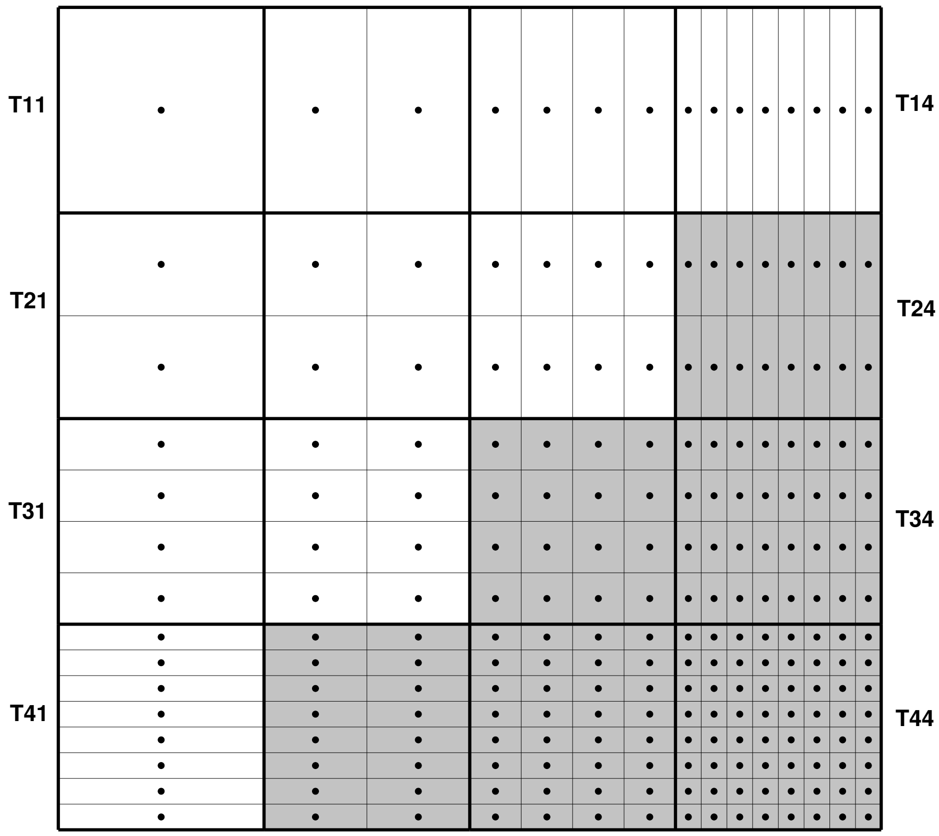


Abbildung 2.4: *Träger der hierarchischen Basisfunktionen in $\tilde{\mathcal{H}}_4^{(2)}$.*

Wie bereits oben angedeutet wurde, kann den jeweiligen Basen ein Gitter zugeordnet werden, indem man die Basispunkte $bp(\varphi)$ für alle Basisfunktionen φ auf dem Lösungsgebiet markiert. Die Tiefe T des dünnen Gitters sei dann definiert durch die maximale Tiefe seiner Teilräume, also gilt: $T = n - 1$. In Abbildung 2.5 sind die zu $\mathcal{H}_4^{(2)}$ und $\tilde{\mathcal{H}}_4^{(2)}$ gehörigen Gitter $\mathcal{G}_4^{(2)}$ und $\tilde{\mathcal{G}}_4^{(2)}$ (mit den entsprechenden Randpunkten) dargestellt.

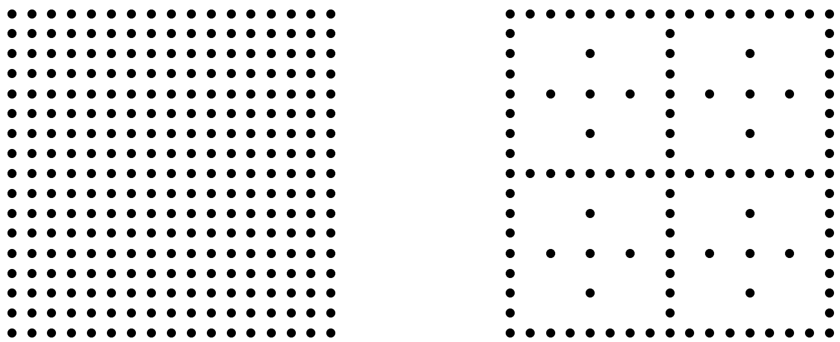


Abbildung 2.5: *Vergleich zwischen einem vollen und einem dünnen Gitter.*

Man kann hier bereits einen optischen Eindruck gewinnen, wie stark die Reduktion der Basispunkte durch den Übergang zu dünnen Gittern ist. Um diese Reduktion zu quantifizieren, wollen wir nun eine Formel für die Zahl $\tilde{N}_n^{(D)}$ der Basisfunktionen in $\tilde{\mathcal{H}}_n^{(D)}$ geben.

Wie wir bereits festgestellt haben, besteht jede Basis eines Teilraumes der Tiefe t aus 2^t Funktionen. Nach elementarer Kombinatorik gibt es $\binom{D+t-1}{t}$ Teilräume $T_{i_1,\ldots,i_D}$ einer festen Tiefe t. Somit haben wir ingesamt

$$\tilde{L}_n^{(D)} := |\mathcal{T}_t^{(D)}| = 2^t \binom{D+t-1}{t} \tag{2.15}$$

Basisfunktionen der Tiefe t und damit insgesamt

$$\tilde{N}_n^{(D)} = \sum_{t=0}^{n-1} \tilde{L}_n^{(D)} = \sum_{t=0}^{T} 2^t \binom{D+t-1}{t} \tag{2.16}$$

Basisfunktionen in $\tilde{\mathcal{H}}_n^{(D)}$.

Wie in [Bun92] gezeigt wird, ist $\tilde{N}_n^{(D)}$ von der Ordnung $O(2^n n^{D-1}) = O(h_n^{-1} \log^{D-1}(h_n^{-1}))$. Dies bedeutet im Vergleich zur Ordnung $O(2^{nD}) = O(h_n^{-D})$ der vollen Basis eine drastische Reduktion. Die Ordnungen bleiben auch dann erhalten, wenn die Ränder des Gitters hinzugenommen werden. Die Approximationsgenauigkeit des Interpolanten in der L_2- oder der L_∞-Norm sinkt lediglich von $O(4^{-n})$ beim vollen Gitter auf $O(4^{-n} n^{D-1})$ beim dünnen Gitter.

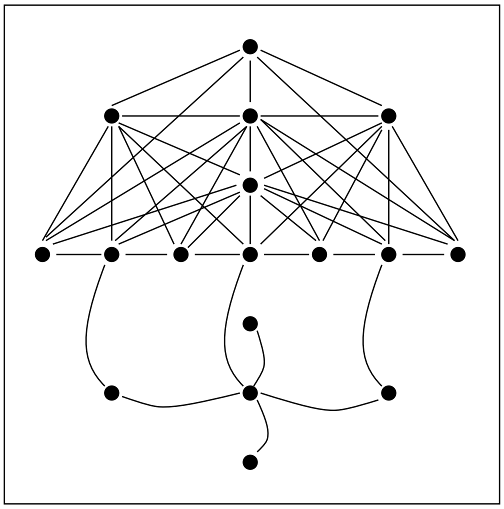


Abbildung 2.6: *Im oberen Teil des Gitters sind diejenigen Punkte miteinander verbunden, deren Träger einen nichtleeren Schnitt besitzen. Die komplexen Abhängigkeiten zwischen den Dünngitterpunkten werden entflochten, indem man die Kommunikationen stets über die hierarchischen Söhne leitet (unterer Teil).*

Der Übergang zu hierarchischen Basen birgt allerdings einen Nachteil. Die Träger der hierarchischen Basisfunktionen überlappen sich in einem wesentlich stärkeren Ausmaß als bei der Standardbasis, in der jede Funktion nur mit ihren direkten Nachbarn überlappt. So hat z.B. der Träger der hierarchisch höchsten Basisfunktion aus $\tilde{\mathcal{H}}_n^{(D)}$ einen nichtleeren Schnitt mit dem Träger jeder anderen Basisfunktion. In den entsprechenden Algorithmen müssen die Werte an den Gitterpunkten zweier überlappender Basisfunktionen gegenseitig ausgetauscht werden. Gerade bei der Parallelisierung kommt es aber darauf an, stark verwobene Strukturen zu entflechten und große voneinander unabhängige Teile zu finden.

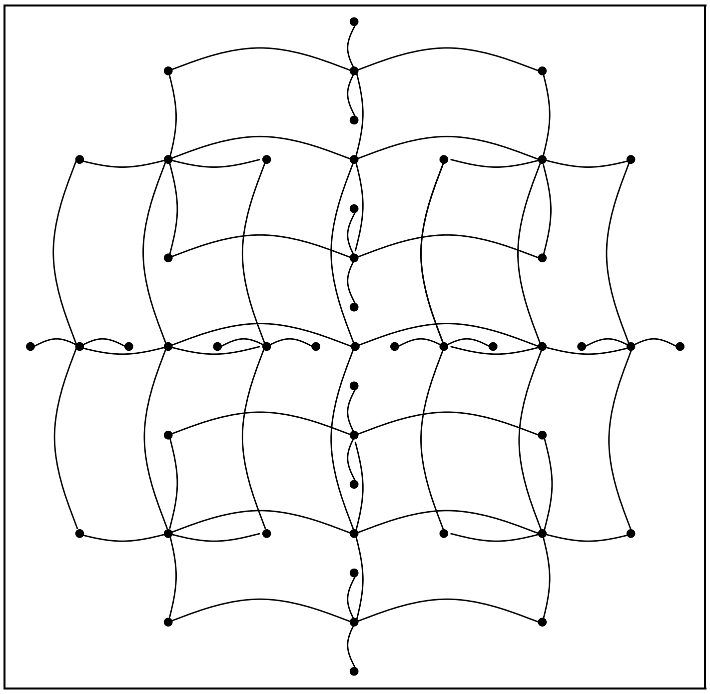


Abbildung 2.7: *Der Kommunikationsgraph eines dünnen Gitters ist ein Produktgraph von Binärbäumen.*

In den folgenden Kapiteln werden wir sehen, daß die wechselseitigen Einflüsse der Basisfunktionen nicht jedesmal direkt ausgetauscht werden müssen, sondern über die hierarchischen Söhne bzw. Väter gereicht werden können. Wie in den Abbildungen 2.6 und 2.7 angedeutet ist, vereinfacht sich dadurch der Abhängigkeitsgraph zu einem D-dimensionalen Produkt von Binärbäumen.

3 Die Helmholtzgleichung auf dünnen Gittern

Abgesehen von den Unterschieden in den numerischen Berechnungen auf den einzelnen Punkten eines dünnen Gitters, ist der strukturelle Aufbau der Programme, d.h. der Durchlauf durch das Gitter und die Abhängigkeiten zwischen den Punkten, sehr ähnlich. Eines der interessantesten und aufwendigsten Anwendungsgebiete stellt hierbei zweifellos die numerische Lösung partieller Differentialgleichungen dar, mit der wir uns in diesem Kapitel am Beispiel der Helmholtzgleichung beschäftigen werden.

Im nächsten Abschnitt beschreiben wir die Diskretisierung der Helmholtzgleichung im Rahmen der Finite-Elemente-Methode auf einem dünnen Gitter. Unser Ziel ist es im folgenden, die mit der Berechnung der Massen- bzw. Steifigkeitsmatrix einhergehenden D-dimensionalen Integrationen auf eindimensionale zurückzuführen. Danach werden zwei algorithmische Varianten zur Berechnung der Massenmatrix analysiert. Zum Abschluß werden die Schwierigkeiten bei der parallelen Beschreibung mit herkömmlichen imperativen Programmiersprachen offengelegt.

3.1 Die Helmholtzgleichung

Wir betrachten ein homogenes Dirichlet-Randwertproblem der folgenden Gestalt:

$$\begin{aligned} -\Delta u + c \cdot u = f \quad &\text{in} \quad \Omega =]0,1[^D \quad \text{mit} \\ u(x) = 0 \quad &\text{auf} \quad \partial\Omega. \end{aligned}$$

Eine Diskretisierung nach der Finite-Elemente-Methode liefert:

$$\int\limits_{\Omega} \nabla\tilde{u}\nabla\tilde{\varphi}_K + c\,\tilde{u}\tilde{\varphi}_K\,dx = \int\limits_{\Omega} \tilde{f}\tilde{\varphi}_K\,dx \qquad \forall K \in \{J\}, \tag{3.1}$$

mit dem Dünngitterinterpolanten $\tilde{u}(x) = \sum\limits_{\{J\}} \tilde{u}_I \cdot \tilde{\varphi}_I(x)$ von u und $\tilde{f}(x) = \sum\limits_{\{J\}} \tilde{f}_I \cdot \tilde{\varphi}_I(x)$ von f. Die Menge $\{J\}$ stehe für die Menge der Indizes aller möglichen Basisfunktionen in $\tilde{\mathcal{H}}_n^{(D)}$. Mit der durch

$$A_{IK}^{(D)} := (\int_{\Omega} \nabla\tilde{\varphi}_I\nabla\tilde{\varphi}_K + c\,\tilde{\varphi}_I\tilde{\varphi}_K\,dx), \qquad \forall I, K \in \{J\}$$

definierten Steifigkeitsmatrix $A^{(D)}$ und der durch

$$M_{IK}^{(D)} := (\int_{\Omega} \tilde{\varphi}_I\tilde{\varphi}_K\,dx), \qquad \forall I, K \in \{J\}$$

definierten Massenmatrix $M^{(D)}$ läßt sich (3.1) als lineares Gleichungssystem formulieren:

$$A^{(D)}(\tilde{u}_I) = M^{(D)}(\tilde{f}_I).$$

Unser Ziel ist es nun, die beiden Matrixmultiplikationen auf einfache eindimensionale Integrationen zurückzuführen. Betrachten wir dazu zunächst die Massenmatrix und formen die Einträge $M_{IK}^{(D)}$ mit der Produktdarstellung der hierarchischen Basisfunktionen um:

$$M_{IK}^{(D)} = \int\limits_{\Omega} \prod_{l=1}^{D} \tilde{\varphi}_{i_l}(x_l) \cdot \prod_{l=1}^{D} \tilde{\varphi}_{k_l}(x_l)\, dx = \prod_{l=1}^{D} \int\limits_{0}^{1} \tilde{\varphi}_{i_l}(x_l) \cdot \tilde{\varphi}_{k_l}(x_l)\, dx_l. \tag{3.2}$$

Zur Vereinfachung der Schreibweise setzen wir

$$S(i,k) := \int_0^1 \tilde{\varphi}_i(x) \cdot \tilde{\varphi}_k(x)\, dx$$

und erhalten

$$M_{IK}^{(D)} = \prod_{l=1}^{D} S(i_l, k_l). \tag{3.3}$$

Wir definieren nun den *l-direktionalen Anteil der Massenmatrix* durch:

$$M_{IK}^{l} := \begin{cases} S(i_l, k_l) & \text{falls } (\forall l' \neq l) \quad i_{l'} = k_{l'} \\ 0 & \text{sonst} \end{cases} \tag{3.4}$$

und für festes D die *l-Massenmatrix* durch

$$M_{IK}^{(l)} := \begin{cases} \prod\limits_{l'=1}^{l} S(i_{l'}, k_{l'}) & \text{falls } (\forall l'' \in \{l+1, \ldots, D\}) \quad i_{l''} = k_{l''} \\ 0 & \text{sonst} \end{cases} \tag{3.5}$$

Die Definition ist nach (3.3) für $l = D$ konsistent.

Lemma 3.1 *Mit obigen Bezeichnungen gilt für alle $l \in \{1, \ldots, D\}$:*

$$M^{(l)} = \prod_{l'=1}^{l} M^{l'}. \tag{3.6}$$

Beweis (durch vollständige Induktion):

Für $l = 1$ ist die Behauptung nach Definition richtig. Für $l > 1$ gilt:

$$(M^{(l-1)} \cdot M^{l})_{IK} = \sum_{X \in \{J\}} M_{IX}^{(l-1)} \cdot M_{XK}^{l}$$

Nach Induktionsvoraussetzung gilt:

$$\begin{aligned} M_{IX}^{(l-1)} \neq 0 &\implies (\forall l' \geq l) \quad x_{l'} = i_{l'} \\ M_{XK}^{l} \neq 0 &\implies (\forall l' \neq l) \quad x_{l'} = k_{l'} \end{aligned}$$

Dabei beide Bedingungen den Bereich $l' > l$ abdecken, ist für $(i_{l+1}, \ldots, i_D) \neq (k_{l+1}, \ldots, k_D)$ somit die Summe Null. Im Falle der Gleichheit liefert höchstens $X = (k_1, \ldots, k_{l-1}, i_l, \ldots, i_D) = (k_1, \ldots, k_{l-1}, i_l, k_{l+1}, \ldots, k_D)$ einen nichtverschwindenden Beitrag zu obiger Summe und es gilt:

$$(M^{(l-1)} M^l)_{IK} = \prod_{l'=1}^{l-1} S(i_{l'}, k_{l'}) \cdot S(i_l, k_l) = \prod_{l'=1}^{l} S(i_{l'}, k_{l'}).$$

Zusammengenommen erhalten wir:

$$(M^{(l-1)} M^l)_{IK} = \begin{cases} \prod\limits_{l'=1}^{l} S(i_{l'}, k_{l'}) & \text{falls } (\forall l' > l) \quad i_{l'} = k_{l'} \\ 0 & \text{sonst} \end{cases}$$

Damit ist die Behauptung bewiesen. ◊

Hieraus folgt insbesondere für $l = D$:

$$M^{(D)} = \prod_{l'=1}^{D} M^{l'}. \tag{3.7}$$

Da die Reihenfolge der Integrationen keine Rolle spielt, sind die Matrixmultiplikationen in (3.7) kommutativ.

Mit (3.7) ist es uns also geglückt, die D-dimensionale Massenmatrix durch das Produkt ihrer l-direktionalen Anteile zu ersetzen, in denen jeweils nur eindimensionale Integrationen vorkommen. Wenden wir uns nun der Steifigkeitsmatrix zu. Der Helmholtzteil $(\int_\Omega c\, \tilde{\varphi}_I \tilde{\varphi}_K \, dx)$ kann natürlich analog zur Massenmatrix behandelt werden. Der Laplaceteil $(\int_\Omega \nabla \tilde{\varphi}_I \nabla \tilde{\varphi}_K dx)$ bedarf hingegen einer näheren Betrachtung.

Die eindimensionalen hierarchischen Basisfunktionen sind bzgl. der Bilinearform

$$< \varphi, \psi > := \int_0^1 \frac{\partial}{\partial x} \varphi \cdot \frac{\partial}{\partial x} \psi \, dx$$

orthogonal, und es gilt:

$$\int_0^1 \frac{\partial}{\partial x} \tilde{\varphi}_i(x) \cdot \frac{\partial}{\partial x} \tilde{\varphi}_k(x) \, dx = \frac{2}{h_k} \cdot \delta_{ik}. \tag{3.8}$$

Dabei bezeichnet h_k wieder die halbe Trägerlänge der Basisfunktion $\tilde{\varphi}_k$ und δ_{ik} das bekannte Kroneckersymbol. Der Laplaceteil der Steifigkeitsmatrix läßt sich nun wie folgt umformen:

$$\int_\Omega \nabla \tilde{\varphi}_I \cdot \nabla \tilde{\varphi}_K \, dx = \int_\Omega \Big(\nabla \prod_{l=1}^{D} \tilde{\varphi}_{i_l}(x_l) \Big) \cdot \Big(\nabla \prod_{l=1}^{D} \tilde{\varphi}_{k_l}(x_l) \Big) \, dx =$$

$$= \sum_{j=1}^{D} \int_\Omega \frac{\partial}{\partial x_j} \prod_{l=1}^{D} \tilde{\varphi}_{i_l}(x_l) \cdot \frac{\partial}{\partial x_j} \prod_{l=1}^{D} \tilde{\varphi}_{k_l}(x_l) \, dx = \sum_{j=1}^{D} \delta_{i_j k_j} \frac{2}{h_{k_j}} \prod_{l \neq j} S(i_l, k_l). \tag{3.9}$$

Wie ein Vergleich mit (3.3) deutlich macht, können wir damit den Laplaceteil durch eine gewichtete Summe $(D-1)$-dimensionaler Massenmatrizen ausdrücken.

3.2 Die eindimensionalen Massenmatrizen

Im vorigen Abschnitt haben wir die Berechnung der D-dimensionalen Massen- bzw. Steifigkeitsmatrix auf die Berechnung eindimensionaler Integrale $S(i,k)$ zurückgeführt. Diese wollen wir nun genauer betrachten.

Elementare Integration ergibt für $i = k$:

$$S(i,i) = \frac{2}{3} \cdot h_i$$

und für $support(\tilde{\varphi}_i) \cap support(\tilde{\varphi}_k) = \emptyset$ natürlich:

$$S(i,k) = 0.$$

Ansonsten sei o.B.d.A. $support(\tilde{\varphi}_i) \subset support(\tilde{\varphi}_k)$. Bezeichnet h_i bzw. h_k die halbe Trägerlänge von $\tilde{\varphi}_i$ bzw. $\tilde{\varphi}_k$, so gilt:

$$S(i,k) = h_i \cdot \tilde{\varphi}_k(bp(\tilde{\varphi}_i)) \tag{3.10}$$

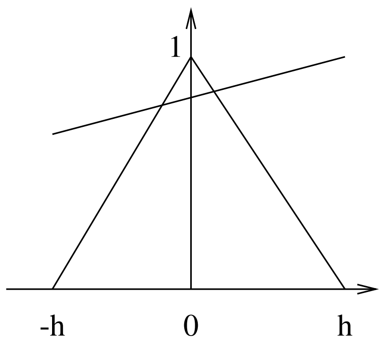
Die letzten Beziehung ergibt sich sofort aus

Lemma 3.2 *Sei $\lambda(x) := \lambda x + \mu$ linear auf $[-h,h]$ und φ definiert durch*

$$\varphi(x) := \begin{cases} \frac{1}{h}(x+h) & x \in [-h,0] \\ -\frac{1}{h}(x-h) & x \in [0,h] \end{cases}$$

Dann gilt:

$$\int_{-h}^{h} \lambda(x)\varphi(x)dx = h\lambda(0) = h\mu. \tag{3.11}$$



Beweis (mit partieller Integration):

$$\begin{aligned}
&\int_{-h}^{h} \lambda(x)\varphi(x)dx = \\
&= \lambda(x)\frac{(x+h)^2}{2h}\Big|_{-h}^{0} + \lambda(x)\frac{(x-h)^2}{-2h}\Big|_{0}^{h} - \int_{-h}^{0} \frac{\lambda}{2h}(x+h)^2dx + \int_{0}^{h} \frac{\lambda}{2h}(x-h)^2dx \\
&= \lambda(0)\frac{h^2}{2h} - 0 + 0 - \lambda(0)\frac{-h^2}{2h} - \frac{\lambda}{2h}\Big(\int_0^h x^2dx + \int_0^h x^2 - 2hx - h^2dx\Big) \\
&= h\lambda(0) - \frac{\lambda}{2h}(h^2x - hx^2)\Big|_0^h \\
&= h\lambda(0)
\end{aligned}$$

$\diamondsuit$

Aus dem Kontext der hierarchischen Basisdarstellung ist bekannt, daß die Anzahl der nichtverschwindenden Einträge in der D-dimensionalen Massenmatrix eine schlechtere Ordnung als die Zahl der Punkte des zugehörigen Gitters besitzt. In [Bal94] wird dies speziell für dünne Gitter noch einmal gezeigt. Demnach verbietet sich das explizite Aufstellen der Matrix. Stattdessen müssen wir die Multiplikation mit der Matrix implizit, d.h. lokal in jedem Gitterpunkt, berechnen.

Wir wollen nun zunächst mit dem eindimensionalen Fall fortfahren und genauer untersuchen, wie sich die Matrixvektormultiplikation

$$g = M \cdot (\tilde{u}_l)$$

für jeden Knoten i durchführen läßt. Wir unterteilen gemäß obiger Darstellung:

$$g_i = M_i \cdot (\tilde{u}_l) = low_i + diag_i + high_i, \tag{3.12}$$

wobei M_i die i-te Zeile von M bezeichne und

$$\begin{aligned} diag_i &= \frac{2}{3} \cdot h_i\, \tilde{u}_i, & (3.13)\\ high_i &= \sum_{\{k \supset i\}} S(i,k) \cdot \tilde{u}_k = \sum_{\{k \supset i\}} h_i \cdot \tilde{\varphi}_k(bp(\tilde{\varphi}_i)) \cdot \tilde{u}_k, & (3.14)\\ low_i &= \sum_{\{k \subset i\}} S(i,k) \cdot \tilde{u}_k = \sum_{\{k \subset i\}} h_k \cdot \tilde{\varphi}_i(bp(\tilde{\varphi}_k)) \cdot \tilde{u}_k. & (3.15) \end{aligned}$$

In (3.14) spiegelt sich nach Definition von $\{k \supset i\}$ der Einfluß der bzgl. des Punktes i hierarchisch höheren Gitterpunkte wider. Die Summe der gewichteten Basisfunktionen $\sum_{\{k \subset i\}} \tilde{u}_k \tilde{\varphi}_k$ bildet gerade den linearen Interpolanten von u auf dem Träger von $\tilde{\varphi}_i$, der durch die beiden Randpunkte des Trägers definiert ist. Deshalb kann (3.14) nach Lemma 3.2 leicht aus den (dehierarchisierten) Funktionswerten $u_{l(i)}$ und $u_{r(i)}$ an den hierarchischen Nachbarn $\tilde{\varphi}_{l(i)}$ und $\tilde{\varphi}_{r(i)}$ berechnet werden:

$$high_i = h_i \cdot \frac{u_{l(i)} + u_{r(i)}}{2}. \tag{3.16}$$

Ordnet man die Gitterpunkte gemäß ihrer hierarchischen Ordnung binärbaumartig an, so kann dieser Beitrag in einem Top-Down-Durchlauf berechnet werden.

Anders verhält es sich mit dem Einfluß der hierarchisch tiefer gelegenen Basisfunktionen in (3.15). In [BZ92] wird ein Verfahren beschrieben, mit dem der Beitrag $h_k \cdot \tilde{u}_k$ in der richtigen Gewichtung $S(i,k)$ im Knoten $\tilde{\varphi}_i$ ankommt. Dabei wird dieser Beitrag ausgehend vom Knoten $\tilde{\varphi}_k$ jeweils mit dem Faktor $\frac{1}{2}$ zu den beiden hierarchischen Nachbarn $\tilde{\varphi}_{l(k)}$ und $\tilde{\varphi}_{r(k)}$ geworfen, die ihrerseits den erhaltenen Wert halbieren und zu ihren Nachbarn werfen. Die Summe der in $\tilde{\varphi}_i$ angekommenen Werte bildet gerade den gewünschten Beitrag. Die Korrektheit des Verfahrens folgt aus der Beziehung

$$\tilde{\varphi}_i(bp(\tilde{\varphi}_k)) = \frac{1}{2}\, \tilde{\varphi}_i(bp(\tilde{\varphi}_{l(k)})) + \frac{1}{2}\, \tilde{\varphi}_i(bp(\tilde{\varphi}_{r(k)})).$$

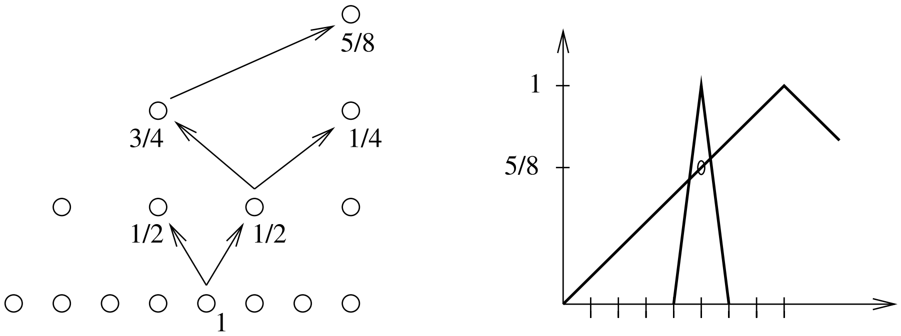


Abbildung 3.1: *Die eindimensionale Überschußschleuder: Der Beitrag 5/8 der kleineren Basisfunktion zu der größeren (rechts) entsteht durch das sukzessive Werfen (links) des halben Knotenbeitrags auf die hierarchischen Nachbarn.*

Dieser Algorithmus wird als (eindimensionale) *Überschußschleuder* bezeichnet, weil die Koeffizienten $\tilde{u}_i$ der hierarchischen Basisfunktionen oft auch *Überschüsse* genannt werden. Abbildung 3.1 soll das Verfahren an einem Beispiel demonstrieren.

Da wir die Numerierung der Basisfunktionen als mit ihrer hierarchischen Ordnung konform vorausgesetzt haben, entspricht die Aufteilung in (3.12) gerade einer Zerlegung der Matrix M in

$$M = L + U, \tag{3.17}$$

wobei L bzw. U eine untere bzw. obere Dreiecksmatrix ist. Den Diagonalanteil schlagen wir der unteren Dreiecksmatrix L zu.

Mit demselben Argument lassen sich auch die l-direktionalen Anteile M^l der D-dimensionalen Massenmatrix $M^{(D)}$ in L^l und U^l zerlegen. Wir zerteilen dazu in Gedanken das zugrundeliegende dünne Gitter in Linien der Richtung l. Für jede dieser Linien ist die entsprechende Restriktion von M^l eine eindimensionale Massenmatrix. Numeriert man die Punkte so, daß nacheinander die Punkte jeder Linie zusammenhängend gezählt werden, so wird M^l demnach blockdiagonal, wobei die Blöcke aus eindimensionalen Massenmatrizen unterschiedlicher Größe bestehen.

3.3 Die D-dimensionale Matrixmultiplikation

Nachdem wir gesehen haben, wie die Multiplikation mit der eindimensionalen Massenmatrix für jeden Knoten lokal berechnet werden kann, wollen wir nun zum D-dimensionalen Fall zurückkehren. Nach (3.7) kann die D-dimensionale Massenmatrix $M^{(D)}$ als Produkt ihrer l-direktionalen Anteile M^l geschrieben werden. Diese Matrizen können nach (3.17) weiter in die Matrizen L^l und U^l aufgespaltet werden. Damit läßt sich (3.7) wie folgt umformen:

$$M^{(D)} = \prod_{l=1}^{D} M^l = \prod_{l=1}^{D} (L^l + U^l) \tag{3.18}$$

Das Produkt auf der rechten Seite kann auf verschiedene Arten berechnet werden. Die Algorithmen zur Berechnung des Matrixvektorprodukts $M^{(D)}(\tilde{u}_I)$ unterscheiden sich darin, wie das Produkt (3.18) aufgelöst wird. Die Multiplikationen der L^l und U^l in (3.18) sind nämlich im Prinzip kommutativ, da die Reihenfolge der damit verbundenen Integrationen unerheblich ist. Allerdings erzwingt die Struktur des dünnen Gitters, daß in einem Summanden jeweils alle L-Multiplikationen *nach* den U-Multiplikationen durchgeführt werden, weil sonst die Zwischenknoten zur Ablage von Teilergebnissen fehlen. Für eine genauere Erklärung dieser Problematik verweisen wir auf die Seiten 29f in [Bal94].

Im folgenden werden zwei Algorithmen zur Berechnung der Matrixmultiplikation diskutiert. Während die erste Variante auf geringen Speicherverbrauch optimiert ist, werden bei der zweiten die Zahl der Baumdurchläufe minimiert.

3.3.1 Der unidirektionale Algorithmus

Jede Multiplikation einer Matrix L^l (bzw. U^l) mit dem Vektor $(\tilde{u}_I)$ kann nach den Ausführungen unter Abschnitt 3.2 in einem einzigen Top-Down- (bzw. Bottom-Up-) Durchlauf durch den kompletten Dünngitterbaum berechnet werden. Wir bezeichnen deshalb diesen Durchlauf im folgenden mit *Down-Zyklus (bzw. Up-Zyklus) in l-Richtung*.

Denkt man sich den D-dimensionalen Baum in Linien (d.h. Binärbäumen) der Richtung l zerlegt, so sind die einzelnen Linien während eines Down-/Up-Zyklus in l-Richtung völlig voneinander unabhängig. Sie können daher auch parallel berechnet werden.

Das erste Verfahren basiert nun auf der folgenden Zerlegung:

$$\prod_{l=1}^{D}(L^l + U^l) = L^1 \cdot \prod_{l=2}^{D}(L^l + U^l) + \prod_{l=2}^{D}(L^l + U^l) \cdot U^1. \tag{3.19}$$

Bei der Implementierung wird eine Kopie der Knotenwerte $\tilde{u}_I$ erstellt. Auf dem Original wird ein Up-Zyklus in 1-Richtung gemacht und auf die so gebildeten Werte das $(D-1)$-dimensionale Produkt angewandt. Die Kopie des Vektors $(\tilde{u}_I)$ wird zuerst mit dem $(D-1)$-dimensionalen Produkt multipliziert und ein Down-Zyklus in 1-Richtung nachgeschaltet. Zum Schluß müssen die beiden so modifizierten Vektoren noch addiert werden.

Die Matrixmultiplikation mit dem $(D-1)$-dimensionalen Produkt wird analog rekursiv aufgelöst, so daß insgesamt nur unidirektionale Baumdurchläufe in den verschiedenen Richtungen auf Kopien der Knotenwerte ausgeführt werden. Für $D = 3$ ergibt sich das in Abbildung 3.2 dargestellte Ablaufschema.

Wie man sich leicht aus der Graphik klar macht, benötigt der unidirektionale Algorithmus $O(2^D)$ Zyklen (Baumdurchläufe). In jedem Punkt werden pro Baumdurchlauf eine konstante Anzahl von Operationen durchgeführt. Durch die Kopien müssen bis zu $O(D)$ Werte *pro Punkt* gespeichert werden. In der sequentiellen Version des Algorithmus' ist

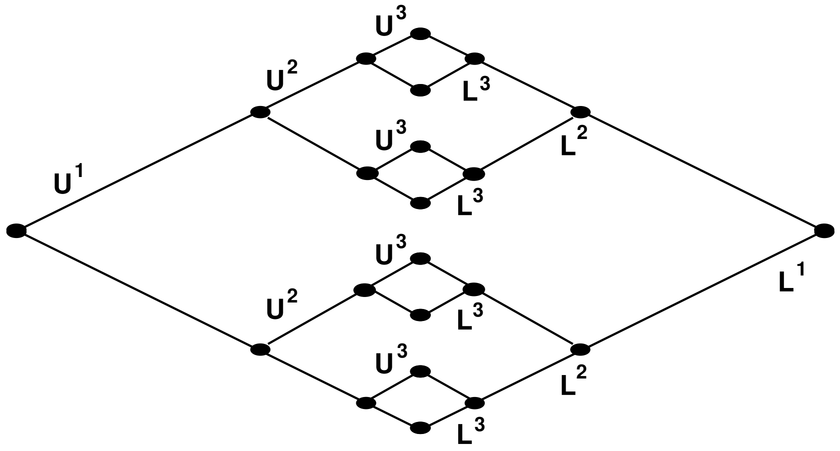


Abbildung 3.2: *Berechnung der Matrixmultiplikation im dreidimensionalen Fall.*

es sogar möglich, mit nur $O(\tilde{N}_T^{(D)})$ Speicherplätzen *insgesamt* auszukommen, weil diese $O(D)$ Werte nicht gleichzeitig für alle Punkte benötigt werden.

Wie sieht es mit dem Parallelisierungspotential aus? Bei jedem unidirektionalen Baumdurchlauf können alle Linien des Gitters in der entsprechenden Richtung voneinander unabhängig berechnet werden. Die Komplexität wird durch die Linie mit der feinsten Auflösung bestimmt. Bei Tiefe T des dünnen Gitters besitzt diese Linie $O(2^T)$ Punkte und kann in $O(T)$ parallelen Schritten berechnet werden. Führt man die $O(2^D)$ Baumdurchläufe nacheinander aus, so ergibt sich eine parallele (Zeit-)Komplexität von $O(2^D \cdot T)$, d.h. der Algorithmus kann in $O(2^D \cdot T)$ Zeitschritten berechnet werden, wenn genügend viele Prozessoren zur Verfügung stehen.

Wir haben damit aber noch nicht das gesamte Parallelisierungspotentials des unidirektionalen Algorithmus' ausgereizt, denn es können auch ganze Zyklen unabhängig voneinander stattfinden. So können beispielsweise die beiden Summanden auf der rechten Seite von (3.19) gleichzeitig berechnet werden. Führt man dieses Prinzip bei den $(D-1)$-dimensionalen Produkten innerhalb der beiden Summanden fort, so erhält man bei einem entsprechenden Mehreinsatz von Prozessoren eine parallele Komplexität von $O(DT)$.

3.3.2 Die D-dimensionale Überschußschleuder

Wir wenden uns nun einem zweiten Verfahren zur Matrixmultiplikation zu, das auf einer ganz anderen Zerlegung von (3.18) basiert:

$$\prod_{l=1}^{D}(L^l + U^l) = \sum_{\mathcal{P}\subset\{1,\ldots,D\}} \prod_{l\notin\mathcal{P}} L^l \prod_{l\in\mathcal{P}} U^l. \tag{3.20}$$

$\mathcal{P}$ durchläuft hier alle Teilmengen von $\{1,\ldots,D\}$. Im Gegensatz zu der bereits in [BZ92] vorgestellten Implementierung, die $O(D)$ Baumdurchläufe benötigt, werden wir in der vorliegenden Arbeit einen Algorithmus entwickeln, der mit nur zwei Baumtraversionen durch den Dünngitterbaum auskommt.

Wie wir sehen werden, können zur Berechnung von $M^{(D)} \cdot (\tilde{u}_I)$ zunächst die 2^D Größen $(g_I^{\mathcal{P}}) := \prod_{l \in \mathcal{P}} U^l\,(\tilde{u}_I)$ für alle $\mathcal{P}$ in einem einzigen Bottom-Up-Durchlauf auf einen Schlag berechnet werden. Diesen Teil des Algorithmus' bezeichnen wir im folgenden als *Up*-Teil der D-dimensionalen Überschußschleuder. In einem anschließenden Top-Down-Durchlauf, dem sogenannten *Down*-Teil, werden dann die Vektoren $\prod_{l \notin \mathcal{P}} L^l\,(g_I^{\mathcal{P}})$ und deren Summe berechnet.

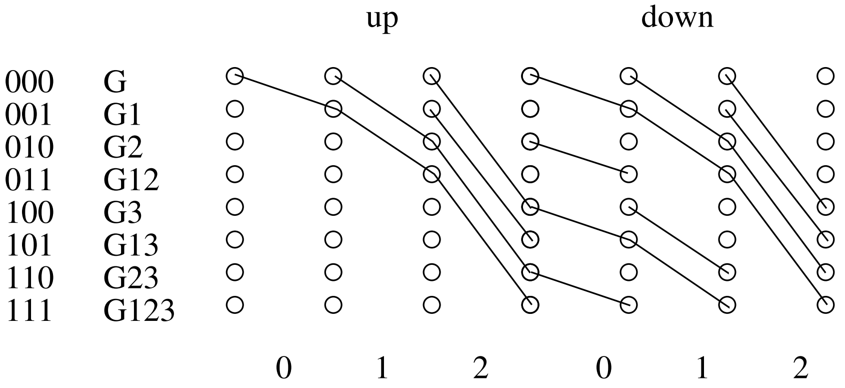


Abbildung 3.3: *Einflüsse der verschiedenen g-Werte untereinander im Up- und Down-Teil der dreidimensionalen Überschußschleuder. Die Graphik zeigt, welche Anteile in welchem Schritt geändert werden.*

Die D-dimensionale Überschußschleuder kann realisiert werden, indem man in jedem Knoten ein Feld G der Länge 2^D bereitstellt, dessen Einträge gerade die Größen $g^{\mathcal{P}}$ repräsentieren. $G[\sum_{l=1}^{j} 2^{k_l - 1}]$ enthält dabei den Wert $g^{k_1,\ldots,k_j}$. $G[0]$ wird zu Beginn auf $\tilde{u}_I$ und der Rest des Feldes auf Null gesetzt. Abbildung 3.3 veranschaulicht für ein dreidimensionales Problem den Ablauf des Algorithmus'. Zunächst werden nacheinander die Multiplikationen mit U^l für $l = 1, \ldots, D$ durchgeführt. Im l-ten Schritt wird hierzu die im vorigen Abschnitt beschriebene eindimensionale Überschußschleuder in Richtung l für alle $g^{k_1,\ldots,k_j}$ mit $j \in \{0, \ldots, l-1\}$ und $k_1 < \ldots < k_j < l$ (d.h. $g^{\mathcal{P}}$ mit $\mathcal{P} \subset \{1, \ldots, l-1\}$) angewandt. Dabei werden die Werte $g^{k_1,\ldots,k_j,l}$ verändert. Am Ende des Up-Teils stehen dann die 2^D Größen $(g_I^{\mathcal{P}}) := \prod_{l \in \mathcal{P}} U^l\,(\tilde{u}_I)$ für alle $\mathcal{P}$ zur Verfügung.

Der nachfolgende Down-Teil durchläuft ebenfalls D Schritte. Im Gegensatz zum Up-Teil werden aber im l-ten Schritt *alle* Größen $g^{k_1,\ldots,k_j}$ mit $j \in \{0, \ldots, D-1\}$ und $k_1, \ldots, k_j \neq l$ (d.h. $g^{\mathcal{P}}$ mit $\mathcal{P} \subset \{1, \ldots, D\} \backslash \{l\}$) gemäß (3.16) behandelt und auf $g^{k_1,\ldots,k_j,l}$ addiert. Auf diese Weise wird zusätzlich zur Matrixmultiplikation mit L^l eine Teilsummation der Vektoren erledigt, so daß nach dem kompletten Down-Teil bereits das vollständige Produkt mit der Massenmatrix in $g^{1,\ldots,D}$ vorliegt.

Der Unterschied zu der in [BZ92] beschriebenen Variante der D-dimensionalen Überschußschleuder besteht nun darin, daß dort jeder der oben aufgeführten Schritte einen eigenen Baumdurchlauf kostete, während in unserer Variante alle Schritte des Up- (bzw. Down-)Teils in einem Baumdurchlauf stattfinden.

Im Zuge der obigen Berechnungen fallen auch die Größen $\prod_{n \neq l \notin \mathcal{P}} L^l\,(g_I^{\mathcal{P}}), n \in \{1, \ldots, D\}$

an. Mit ihnen kann der Laplaceteil der Steifigkeitsmatrix berechnet werden, wie ein Vergleich mit (3.9) deutlich macht. Da die Überlegungen für die Massenmatrix auf den Helmholtzteil der Steifigkeitsmatrix übertragbar sind, kann somit auch die Multiplikation $(A_{IK}^{(D)})(\tilde{u}_I)$ in nur zwei Baumdurchläufen erledigt werden.

Wie wir oben gesehen haben, finden in jedem Punkt pro Baumdurchlauf $O(2^D)$ Operationen statt. Sie können, wie in Abbildung 3.3 deutlich wird, in $O(D)$ parallelen Schritten durchgeführt werden. Da bei diesem Algorithmus alle Punkte einer Tiefe t unabhängig voneinander berechnet werden können und nur von ihren Vätern bzw. Söhnen abhängen, erhalten wir eine parallele Komplexität von $O(DT)$.

Die Überschußschleuder besitzt also im wesentlichen die gleichen Parallelisierungseigenschaften wie der unidirektionale Algorithmus. Für die Praxis weist sie allerdings zusätzliche Vorteile auf. Denn dort brauchen Berechnungen an einem Punkt nicht mehr weiter parallelisiert werden, da das Potential der gleichzeitigen Behandlung aller Punkte einer festen Tiefe bereits völlig ausreichend ist. Dann aber ist es besser, die pro Knoten nötigen $O(2^D)$ Operationen in nur zwei Baumdurchläufen zu erledigen, anstatt sie über $O(2^D)$ Baumtraversionen zu verteilen. So kommt man zu einem günstigeren Verhältnis zwischen Berechnungs- und Kommunikationskosten. In realer Hardware ist die Kommunikation stets um Größenordnungen teurer als die Rechenleistung.

Sogar im sequentiellen Fall hat die Überschußschleuder bei hohen Dimensionen den Vorteil, daß die Zahl der rekursiven Funktionsaufrufe auf den Gitterpunkten mit der Zahl der Baumtraversionen im Vergleich zu dem unidirektionalen Algorithmus stark reduziert ist. Diesen Effekt werden wir in Kapitel 7 anhand konkreter Laufzeitmessungen belegen können.

Wir greifen die D-dimensionale Überschußschleuder in Kapitel 5 noch einmal auf und zeigen die praktische Realisierung in einer funktionalen Sprache. Vorher möchten wir aber diskutieren, weshalb sich die parallele Beschreibung dieses Dünngitteralgorithmus' in imperativen Sprachen wie C oder FORTRAN schwierig gestaltet.

3.4 Probleme der imperativen Beschreibung

Ein gängiges Parallelisierungskonzept bei numerischen Algorithmen ist die Gebietszerlegung. Dabei wird versucht, das Problemgebiet so in kleinere Stücke zu zerteilen, daß die erforderlichen Berechnungen dort unabhängig voneinander durchgeführt werden können. Das Prinzip der Unterteilung kann rekursiv auf die Teilgebiete angewandt werden. In einem allgemeineren Kontext spricht man auch von *Divide-et-Impera-Algorithmen*. Sie erlauben eine übersichtliche Beschreibung der Parallelisierung.

Betrachten wir z.B. die zweidimensionale Helmholtzgleichung, so können die erforderlichen Berechnungen auf dem Einheitsquadrat parallel in der nördlichen und südlichen Hälfte durchgeführt werden, da sich die zugehörigen Basisfunktionen gegenseitig nicht beeinflussen. Allerdings besteht eine Wechselwirkung beider Hälften zu ihrem Separator, der Mittellinie des Quadrats. Da die Basisfunktionen auf dem Separator hierarchisch über denen der Gebietshälften liegen, muß die Mittellinie im Up-Teil des Algorithmus' *nach*, im Down-Teil dagegen *vor* den Hälften aktualisiert werden. Die Mittellinie kann ebenfalls in zwei Teillinien links und rechts des Mittelpunktes geteilt werden. Auf diese Weise wird der Parallelisierungsgrad weiter erhöht.

Das Verfahren läßt sich analog auf den D-dimensionalen Fall fortsetzen. So wird etwa im Down-Teil des Algorithmus' als erstes der $(D-1)$-dimensionale Separator $\{x_D = \frac{1}{2}\}$ mit Hilfe der Werte auf den entsprechenden Rändern berechnet. Der aktualisierte Separator dient dann als Rand für die D-dimensionalen Würfelhälften $\{x_D > \frac{1}{2}\}$ und $\{x_D < \frac{1}{2}\}$, die unabhängig voneinander berechnet werden können. Das Prinzip der Aufteilung in Separator und zwei Teilgebiete wird rekursiv sowohl auf die beiden Hälften, als auch auf den $(D-1)$-dimensionalen Trennhyperwürfel angewandt. Die Rekursion terminiert für $D = 0$ oder $T = 0$. In beiden Fällen besteht das zu behandelnde Gitter aus einem Punkt.[1]

Das Grobgerüst dieses Algorithmus' (ohne Terminierungsbedingungen) sieht wie folgt aus:

```
down(D,Gebiet) {
    down(D-1,Separator);
    PARDO {
        down(D,erste_Haelfte);
        down(D,zweite_Haelfte);
    }
}
```

Uns interessiert nun die Komplexität dieses parallelen Algorithmus'. Auskunft darüber gibt

Lemma 3.3 *Sei D eine beliebige, aber konstante, natürliche Zahl. Sei weiterhin die Funktion*

$$pt : I\!N \times I\!N \longrightarrow I\!N, (D,T) \mapsto pt(D,T)$$

definiert als die Anzahl der Zeitschritte, die der oben skizzierte Down-Algorithmus auf einem D-dimensionalen dünnen Gitter der Tiefe T benötigt. Dann gilt:

$$pt(D,T) = \binom{D+T}{D} = O(T^D)$$

Beweis (durch vollständige Induktion über D und T):

[1]Ein nulldimensionales dünnes Gitter wurde zwar in Kapitel 2 nicht eingeführt, weil es in der Praxis keine Bedeutung besitzt. Für die algorithmische Formulierung ist diese Erweiterung aber nützlich.

Für beliebige $D > 0$ besteht $\tilde{\mathcal{G}}_0^{(D)}$ nur aus einem Punkt. Deshalb gilt

$$pt(D,0) = 1 = \binom{D+0}{D} \quad \forall D \in I\!N \tag{3.21}$$

Für beliebige $T > 0$ besteht $\tilde{\mathcal{G}}_T^{(0)}$ ebenfalls nur aus einem Punkt. Deshalb gilt

$$pt(0,T) = 1 = \binom{0+T}{0} \quad \forall T \in I\!N \tag{3.22}$$

Sei nun allgemein $D > 0$ und $T > 0$. Aus dem angegebenen Algorithmus resultiert folgende Rekursionsbeziehung für $pt(D,T)$:

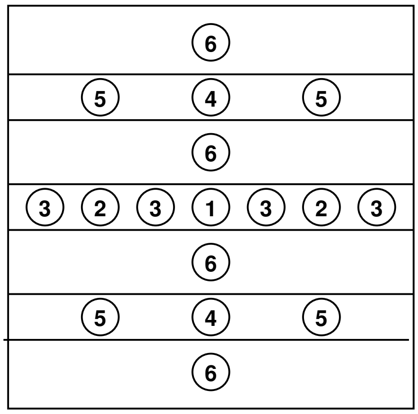
$$pt(D,T) = pt(D-1,T) + pt(D,T-1)$$

Mit der Induktionsvoraussetzung gilt:

$$pt(D,T) = \binom{D-1+T}{D-1} + \binom{D+T-1}{D} = \binom{D+T}{D} = O(T^D) \tag{3.23}$$

Aus (3.21), (3.22) und (3.23) folgt unmittelbar die Behauptung. ♢

Betrachten wir allerdings die Abhängigkeitsstruktur zwischen den einzelnen Gitterpunkten näher, so sehen wir, daß die maximale Parallelität bei weitem nicht ausgeschöpft ist. Wie schon in [Zen92] dargelegt wurde, könnten aufgrund der Symmetrie bzgl. der Dimension prinzipiell alle $2D$ hierarchischen Söhne eines Gitterpunktes nach dessen Aktualisierung *gleichzeitig* im nächsten Schritt parallel berechnet werden (vgl. Abbildung 3.4 rechts). Vom Mittelpunkt des Gebiets startend, könnten so alle Punkte in spätestens T Schritten erreicht werden. Statt $O(T^D)$ erhalten wir so eine mögliche parallele Komplexität von nur $O(T)$.



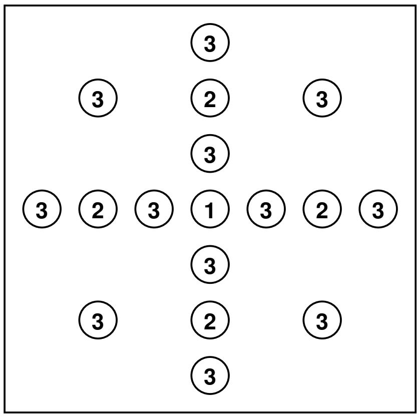


Abbildung 3.4: *Berechnungsreihenfolge der Gitterpunkte im zweidimensionalen Fall. Die linke Graphik zeigt, welche Punkte zu welchem Zeitschritt des zweidimensionalen Gebietszerlegungsalgorithmus' besucht werden. Im rechten Bild ist die optimale Reihenfolge dargestellt.*

Der Nachteil des obigen Down-Algorithmus' besteht darin, daß die beiden Hälften und der Separator *als ganzes* gesehen werden. Deshalb muß der Separator komplett vor den

Teilgebieten berechnet werden (vgl. Abbildung 3.4 links), obwohl einzelne *Punkte* der Teilgebiete jeweils nur von einem Punkt des Separators abhängen und somit bereits vor dessen völligen Fertigstellung aktualisiert werden könnten.

Ein weiteres Problem ist die resultierende Kommunikationsstruktur. Wenn wir annehmen, daß alle Punkte des Gitters auf eigenen Prozessoren liegen, so müssen etwa in der zweidimensionalen Variante des obigen Algorithmus' alle Daten einer Linie zum Mittelpunkt geschickt werden, bei dem der Binärbaum aufgebaut wird. Dieser Baum wird dann an die Mittelpunkte der beiden Teilhälften gesandt, von wo aus der Baum wieder sukzessive aufgespalten wird und dessen Knoten auf die entsprechenden Gitterpunkte verteilt werden. So marschiert der Wert eines Punktes u.U. über viele Zwischenstationen, die den Wert eigentlich gar nicht benötigen. Wesentlich besser wäre es, wenn die Kommunikation direkt zwischen Sender und Empfänger erfolgen würde, statt die Baumhierarchie hinauf und wieder hinunter zu laufen.

Es ist allerdings in der Tat äußerst schwierig und aufwendig, mit herkömmlichen imperativen Programmiersprachen einen Algorithmus für die D-dimensionale Überschußschleuder anzugeben, der die optimale Komplexität auszudrücken vermag. Man stelle sich vor, wie kompliziert eine Anreicherung des Gebietszerlegungsprogramms mit Message-Passing-Direktiven aussähe, mit deren Hilfe voneinander abhängige Punkte direkt Daten austauschen könnten.

Wie sehen die Alternativen zur Gebietszerlegung aus? Es ist z.B. keine gute Lösung, im Mittelpunkt zu starten und $2D$ rekursive Aufrufe auf den hierarchischen Söhnen parallel zu tätigen. Die Crux dabei ist, daß ein und derselbe Punkt normalerweise in jeder Dimension einen Vater besitzt. Er würde also von bis zu D Punkten aus aufgerufen. Statt $O(2^T \cdot T^{D-1})$ Punkten würden demnach $O(\sum_{i=0}^{T} 2^i D^i) = O(2^{T+1} D^{T+1})$ Punkte gerechnet werden.

Einen Ausweg böte ein Algorithmus, der in einer Schleife alle Tiefen durchliefe und dort jeweils alle Punkte aktualisierte. Dies könnte dann innerhalb einer Tiefe parallel geschehen. Allerdings wäre es hierbei sehr schwierig zu beschreiben, welche Punkte der aktuellen Tiefe von welchen Punkten höherer (bzw. niedrigerer) Tiefe abhängen. Konkret bedeutet dies, daß sich der Zugriff auf die hierarchischen Söhne, Väter und Nachbarn kompliziert gestalten würde. Da diese Verwandtschaftsbeziehungen zwischen den Punkten nun stark verschleiert würden, wäre es auch mühsam, eine kommunikationsoptimierte parallele Verteilung der Punkte auf die Rechenknoten zu finden.

Die bisherige Diskussion führt uns zu einer häufig gemachten Erkenntnis: Der Zwang zur expliziten Formulierung der parallelen Berechnungsreihenfolge steht oft quer zur logischen Gliederung eines Programms. Diese Schwierigkeit motiviert die Untersuchung funktionaler Konzepte zur Beschreibung paralleler numerischer Algorithmen. Mit der im folgenden Kapitel vorgestellten Sprache FASAN ist es möglich, an der bequemen und übersichtlichen Gebietszerlegungsstrategie festzuhalten und trotzdem das maximale Parallelisierungspotential auszunutzen.

4 FASAN

FASAN ist eine funktionale Datenflußsprache, die aus einer Reihe von Diplomarbeiten ([Ebn94, Rau92, Vei90]) hervorgegangen ist und auf den in [Zen92] vorgestellten Ideen beruht. Der Name steht für „Functions Applied to Streams in Algorithms of Numerical computing“. FASAN ist insbesondere für hierarchische numerische Algorithmen gedacht, die auf einem Workstationcluster automatisch parallelisiert werden sollen. Mit FASAN soll es dem Anwender möglich sein, sich bei der Programmierung im wesentlichen nur auf die Formulierung der numerischen Abhängigkeitsbeziehungen zu beschränken, während die Parallelisierung und die nötige Kommunikation automatisch erledigt werden.

Für die Numerische Mathematik stehen hohe Geschwindigkeit und niedriger Speicherbedarf der Programme im Vordergrund. Es ist klar, daß die Umsetzung der funktionalen Ebene zusätzliche Kosten verursacht. In FASAN wurden aber eine Reihe von Strategien entwickelt, die den Mehraufwand in Grenzen halten, bzw. ihn auf die Startphase beschränken.

Außerdem sollen große Programmteile aus bereits bestehendem Code weiterverwendet werden können, da der Anwender meist nicht bereit ist, auf ausgeklügelte Unterroutinen zu verzichten. Deshalb dient die funktionale Ebene von FASAN lediglich der parallelen Komposition bereits vorhandener sequentieller Module.

Im nächsten Abschnitt geben wir einen kurzen Abriß über die Geschichte funktionaler Sprachen. Abschnitt 4.2 bietet einen Überblick über die wesentlichen Konzepte und die Syntax von FASAN. Eine wichtige Rolle bei der effizienten Implementierung von FASAN spielen die sogenannten *Kabelbäume*, denen der letzte Teil dieses Kapitels gewidmet ist.

4.1 Zur Entwicklung funktionaler Sprachen

Bevor wir näher auf die Sprache FASAN eingehen, möchten wir einen knappen Abriß über die bisherige Geschichte der funktionalen Programmiersprachen und ihrem Umfeld geben. Für eine ausführlichere Einführung empfehlen wir [Sch93] und [Mar92].

Die Initialzündung ging im Jahre 1978 von Backus’ Turing-Award-Vortrag [Bac78] aus, in der Backus, der Erfinder von FORTRAN, den „Von-Neumann–Flaschenhals“ zwischen Prozessor und Speicher herkömmlicher Maschinen anprangerte und die Vorzüge funktionaler Sprachen gegenüber traditionellen Ansätzen darstellte. Seitdem unternahmen Wissenschaftler auf der ganzen Welt große Anstrengungen, um diese Ideen in die Praxis umzusetzen.

Da man lange Zeit davon ausging, daß prinzipiell neue Rechnertypen notwendig seien, um hohe Leistung mit funktionalen Sprachen zu erzielen, wurden eine Reihe von (zumeist parallelen) Architekturen entworfen. Eine Klasse dieser Architekturen bilden die sogenannten *Reduktionsmaschinen*, die den nachfragegetriebenen Sprachen (s. unten) entsprechen. Zu ihnen gehören u.a. das von Darlington und Reeve entwickelte transputerbasierte Modell ALICE (Applicative Language Idealised Computing Engine) [DR81] und das auf einem Netzwerk konventioneller Prozessoren aufbauende GRIP (Graph Reduction in Parallel) [PJCSH87].

Eine andere Klasse von Architekturen sind *Datenflußmaschinen*, deren prinzipielle Idee in zwei Punkten besteht. Erstens fließen Daten direkt zwischen Instruktionen anstatt sie über gemeinsame Variablen auszutauschen. Zweitens wird der Programmablauf nicht über einen Programmzähler oder einen zentralen Kontrollmechanismus gesteuert, sondern allein über die Verfügbarkeit der Daten. Beispiele für solche Maschinen sind die MDM (Manchester Dataflow Machine) [GKW85] und Monsoon [Pap94].

Auf der Software-Seite wurden zahlreiche funktionale Sprachen entworfen. Im sequentiellen Bereich erzielte man dabei mit Lazy ML [Aug84] und SML [MTH90], dem Orbit-Compiler für Scheme [Kra86] und Haskell [HPJWe92], einem Quasi-Standard, große Fortschritte. Im parallelen Bereich konnten jedoch die theoretisch zu erwartenden großen Vorteile gegenüber imperativen Programmiersprachen noch nicht erreicht werden.

Die Grundidee funktionaler Sprachen ist die Darstellung eines Programms als eine *Funktionsabstraktion* f, d.h. als ein parametrisierter Ausdruck, der sich rekursiv aus Teilausdrücken zusammengesetzen kann. Die Ausführung des Programms besteht in der Auswertung von f für bestimmte Eingabeparameter. Diese Sichtweise ist für die parallele Programmierung sehr vorteilhaft. Denn aus dem *Church-Rosser-Theorem* (s. etwa [Bar84]) folgt, daß die Auswertungsreihenfolge der einzelnen Teilausdrücke für das Gesamtergebnis irrelevant ist. Sie können demnach auch parallel erfolgen.

Die verschiedenen funktionalen Ansätze unterscheiden sich daher im wesentlichen in der Strategie, in welcher Reihenfolge sie die Teilberechnungen durchführen (*scheduling*). Sie lassen sich grob gesprochen in zwei Klassen einteilen. In *nachfragegetriebenen (demand-driven)* Sprachen werden Ausdrücke erst dann ausgewertet, wenn ihre Ergebnisse tatsächlich gebraucht werden (*lazy evaluation*). Implementiert werden diese Sprachen meistens durch sogenannte *Graphreduktionssysteme* und weniger häufig durch *Textreduktionssysteme*. Der Unterschied besteht darin, daß bei Textreduktion jede Nachfrage nach dem Ergebnis eines Ausdrucks durch eine vollständige Kopie des Ausdrucks beantwortet wird, so daß das Resultat jedesmal neu berechnet werden muß. Bei Graphreduktion wird der Ausdruck hingegen nur das erste Mal berechnet und danach ein Zeiger auf das Ergebnis übergeben. Zu den bekannten Vertretern paralleler Reduktionssprachen zählen HOPE für die ALICE-Architektur und Concurrent Clean auf ZAPP (Zero Assignment Parallel Prozessor) [GMS93].

Die ursprünglich für Datenflußmaschinen entworfenen Datenflußsprachen sind im Gegensatz zu Reduktionssprachen *datengetrieben (data-driven)*. Hier bestimmt allein die Verfügbarkeit der Daten den Zeitpunkt der Funktionsauswertung: Ein Task ist rechenbereit, wenn alle Eingabedaten vorliegen. Dies führt im allgemeinen zu *eager evaluation*, d.h. alle Argumente einer Funktion werden vor dem Funktionsaufruf selbst ausgewertet.

Die Programme werden standardgemäß in einen Datenflußgraphen übersetzt, bei dem jede Instruktion einem Knoten entspricht. Neben der in [AN90] beschriebenen Sprache Id (Irvine dataflow) gewinnt hier gerade im numerischen Bereich die Sprache SISAL (Streams and Iterations in a Single-Assignment Language) [FMS+95, McG85] an Bedeutung. SISAL-Programme konnten sowohl auf Shared-Memory-Prozessoren als auch auf Vektorrechnern die Geschwindigkeit von FORTRAN-Programmen erzielen.

Diese Erfolge sind darauf zurückzuführen, daß SISAL speziell auf die Bedürfnisse herkömmlicher Numerikprogramme eingeht, v.a. durch eine effiziente Implementierung von Array-Strukturen. Moderne, hierarchisch aufgebaute Algorithmen bergen aber, wie wir im letzten Kapitel verdeutlicht haben, neue Probleme, denen wir hier durch einen eigenen Ansatz Rechnung tragen wollen.

4.2 Konzepte und Syntax von FASAN

In diesem Abschnitt sollen die wesentlichen Komponenten der Sprache FASAN vorgestellt werden. Dabei wird auch die Syntax informell eingeführt, so daß der Leser ein Gefühl für die Sprache bekommt und die Programme im nächsten Kapitel verstehen kann. Für eine formalere und umfassendere Beschreibung der Syntax verweisen wir auf [Ebn94]. Das Laufzeitsystem wird in [EP96] näher beschrieben.

Die Grundbausteine von FASAN sind Funktionen, funktionale Einheiten mit einer festen Zahl von Ein- und Ausgängen. Sie werden durch *Ströme* zu einem Netz verbunden. Diese Ströme übertragen unidirektional Sequenzen von Daten. Eine Funktion hat also die folgende allgemeine Definition:

function `A(in1,...,inN): out1,...,outM ;`

wobei `in1,...,inN` die N Eingangsströme und `out1,...,outM` die M Ausgangsströme von `A` sind. Eine Zuweisung, bei der diese Funktion ausgewertet wird, erscheint dann im Programmtext wie folgt:

```
o1,...,oM := A(i1,...,iN);
```

Eine Funktion kann auch sogenannte *Zwischenströme* besitzen, die nur innerhalb des Funktionenkontexts bekannt sind und am ehesten mit lokalen Variablen imperativer Programmiersprachen vergleichbar sind. Auch für Zwischenströme gilt, daß sie nur einen Erzeuger, eventuell aber mehrere Verbraucher besitzen (*single assignment*).

Wie die nachfolgenden Beispielprogramme zeigen, erscheinen die Ströme dem Programmierer als normale Ein- bzw. Ausgabeparameter, da die Funktionen bei jedem Aufruf in der Regel[1] genau ein Stromelement von jedem Eingabestrom lesen und nach der Berechnung genau ein Stromelement auf jeden Ausgabestrom schreiben. Die Stromsemantik ist

[1] Die Ausnahme bilden sog. *asynchrone* Funktionen, auf die wir hier nicht näher eingehen wollen. Für eine detailliertere Auseinandersetzung mit dieser Problematik verweisen wir auf [Ebn94].

aber auf Implementierungsebene wichtig. Einerseits kann ein bereits erzeugter komplexer Datenflußgraph immer wieder für verschiedene Daten verwendet werden, etwa ein Gleichungssystemlöser mit verschiedenen rechten Seiten. So können Kosten für das wiederholte Erzeugen des Netzes gespart werden. Andererseits ist dadurch pipeline-artige Parallelisierung möglich.

FASAN unterscheidet zwei Arten von Funktionen: *Elementarfunktionen* und *Netzfunktionen*. Elementarfunktionen sind Funktionen über Strömen und werden nicht weiter untergliedert. In ihnen finden die eigentlichen Berechnungen statt. Sie stützen sich auf sequentielle Funktionen aus einer externen Objektdatei oder Programmbibliothek, die in einer beliebigen funktionalen oder imperativen Programmiersprache (C, C++, FORTRAN, etc.) geschrieben worden sind. Von ihnen wird lediglich gefordert, daß sie nach außen ein funktionales Verhalten zeigen. Ist beispielsweise die Summation zweier Zahlen in einer C-Funktion `sum_c` kodiert, so läßt sich diese folgendermaßen in FASAN integrieren:

```
function sum(x,y): z
is function sum_c(x,y,z);
```

Der Anwender kann auf die Weise große Teile seines alten Codes unverändert übernehmen. Auf FASAN-Ebene gibt es keine Datentypen, sie treten nur in den sequentiellen Funktionen auf. Ein einzelnes Stromelement kann sehr komplexe Datenstrukturen beinhalten, z.B. große Felder. FASAN legt es in die Verantwortung der sequentiellen Funktionen, daß zwischen Sender und Empfänger (beides sequentielle Funktionen) keine Inkonsistenzen auftreten. Über die Größe der eingebundenen sequentiellen Funktionen läßt sich auch die Granularität der Parallelisierung steuern.

Netzfunktionen (*komponierte Funktionen*) hingegen repräsentieren ein Netz von Funktionen und stellen die Verbindung der Ein- und Ausgangsströme von Elementarfunktionen her. Diese können somit hinter- und nebeneinander geschaltet werden. Dabei sind Rekursionen und bedingte Zuweisungen (**if ... then ... else**) erlaubt. Das eigentliche Hauptprogramm ist als ein initiales Funktionsnetz vorgegeben, das zur Laufzeit *entfaltet* wird, indem die darin enthaltenen komponierten Funktionen schrittweise durch ihr Funktionennetz ersetzt werden. Um die maximale Parallelität gewährleisten zu können, werden Netzfunktionen bereits *entfaltet*, wenn einer der Eingangsströme das erste Datum liefert. Allerdings ist eine Elementarfunktion nur dann *rechenbereit*, wenn sie auf allen Eingangsströmen Daten erhält. Die Auswertungsstrategie ist also in diesem Sinne „eager“. Lediglich bei bedingten Anweisungen wird erst die Bedingung ausgewertet und dann der entsprechende Zweig der Fallunterscheidung. Diese Einschränkung bezeichnet man mit *lenient evaluation*.

Netzfunktionen können auch als (rekursiver) Graph dargestellt werden, der den Datenfluß zwischen den inneren Funktionen beschreibt. Ein Beispiel ist in Abbildung 4.1 auf Seite 36 dargestellt. Man kann sich vorstellen, daß in dem Graphen die komponierten Funktionen schrittweise durch ihren Netzgraphen ersetzt werden, bis nur noch Elementarfunktionen enthalten sind. Dieses Entfalten ist jedoch nicht statisch zur Compilezeit möglich, da die Rekursionstiefe normalerweise durch Laufzeitbedingungen gesteuert wird.

Nach diesen grundsätzlichen Bemerkungen zu Syntax und Konzepten von FASAN beschreiben wir im folgenden einige wichtige Spezialitäten der Sprache, die für das Verständnis der folgenden Kapitel wesentlich sind.

Um den Aufwand für die Verwaltung des Datenflußgraphen (Funktionennetzes) in Grenzen zu halten, können Funktionennetze in FASAN „recycelt" werden. Dies ist v.a. für iterative Algorithmen gedacht. Hier ist es möglich, die Ergebnisse einer Iteration als Startwerte für die neue Iteration auf dem gleichen Funktionennetz zu verwenden. Wenn beispielsweise bereits einmal ein komplexes rekursives Netz namens `iterate` entfaltet worden ist, so sorgt

`NewResult :=` **recycle** `iterate(ResultOfLastIteration);`

dafür, daß das bestehende Netz mit den neuen Startdaten gefüttert wird. In dem Datenflußgraphen kommt es hierbei zur zyklischen Umleitung von Ausgangsströmen auf die Eingänge. Fehlt das Schlüsselwort **recycle**, wird dagegen das Netz vollständig neu aufgebaut. Dies ist dann vorteilhaft, wenn man eine Neuplazierung der parallelen Funktionen wünscht.

Rein syntaktisch gesehen ist Recycling nur möglich bei sogenannten *endrekursiven (tail-recursive)* Funktionen, d.h. der rekursive Aufruf der Funktion ist zugleich die äußerste Funktionsanwendung (vgl. hierzu [Thi94], S. 210f). Solche Funktionen werden oft auch repetitiv oder iterativ genannt, weil sie unmittelbar in eine iterative Vorschrift abgebildet und ohne die Verwendung eines Funktionsstapels implementiert werden können. Bei endrekursiven Funktionen kann es durch das Recycling zu keiner Verklemmung kommen, obwohl die Zyklenfreiheit des Datenflußgraphen aufgehoben wurde.

Eine Besonderheit von FASAN stellen die sogenannten *Kabelbäume* dar, denen der nächste Abschnitt gewidmet ist. Sie sorgen dafür, daß unnötige Kommunikationen und Synchronisationspunkte, die allein durch eine bequemere textuelle Beschreibung zustande gekommen sind, eliminiert werden. Wesentliche Idee ist dabei, daß Ströme *direkt* ohne Zwischenstationen vom Sender zu den jeweiligen Empfängern gezogen werden, auch wenn es im Programmtext so erscheint, als wenn sie über verschiedene Zwischenfunktionen liefen. Wie wir später sehen werden, können damit Funktionen aus operationeller Sicht auch langlebige lokale Daten besitzen, die sie am Platz verändern können.

Die Lastverteilung geschieht über benutzerdefinierte Lokationen, also Lastverteilungsfunktionen, die als Ergebnis einen „Ort" im Rechnernetz liefern. So bewirkt z.B.

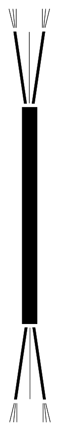
`x := A(y)` **on** `next_proc;`

daß die Funktion `A` auf demjenigen Rechner gestartet wird, den die Lokation `next_proc()` zur Laufzeit berechnet. `A` kann dabei Elementar- oder Netzfunktion sein. Lokationen sind aus der Sicht des Programmierers spezielle Elementarfunktionen mit einer definierten Schnittstelle, in der z.B. der aktuelle Rechner und die Gesamtzahl aller Rechner automatisch übergeben werden. Weil diese Schnittstelle auch statistische Laufzeitinformationen vorsieht, kann der Programmierer mit Lokationen auch dynamische Lastverteilungsstrategien implementieren.

Durch eine Aufrufoption kann die Entfaltungstiefe begrenzt werden. Dadurch wird verhindert, daß die Granularität zu klein bzw. die Zahl der entstehenden Tasks zu groß wird. Ab dieser Tiefe wird die Breitenrekursion (weitere Entfaltung des Funktionennetzes) durch Tiefenrekursion (einfache sequentielle Auswertung) ersetzt. Für jede Netzfunktion wird nämlich zur Compilezeit eine sequentialisierte (rekursive) C-Funktion erzeugt, die in diesem Fall verwendet wird. Eine mögliche Strategie besteht etwa darin, das Netz solange aufzufalten, solange noch freie Prozessoren zur Verfügung stehen und danach auf die sequentiellen C-Funktionen überzugehen. Allerdings kann es ähnlich wie bei *thread*-basierten Ansätzen vorkommen, daß auch innerhalb eines Prozessors eine weitere Auffaltung in unabhängige Module ein größeres Parallelisierungspotential eröffnet.

Der FASAN-Compiler übersetzt FASAN-Programme nach C, die mit einem Laufzeitsystem auf Basis von PVM zusammengebunden werden. Dieses Laufzeitsystem sorgt für die automatische Plazierung, Migration und Kommunikation der Funktionen.

4.3 Kabelbäume



Kabelbäume sind eine Besonderheit von FASAN. Sie sind Ströme, die wieder Ströme (und Kabelbäume) enthalten können. Damit lassen sich Teilströme syntaktisch zu Listen und Bäumen zusammenfassen, die in einem einzigen Strom gebündelt sind. Dies erhöht die Übersichtlichkeit und bessere Handhabung rekursiver Algorithmen. Die Zusammenfassung der Ströme `s1,...,sN` in einen Kabelbaum `S` kann durch eine der beiden gleichwertigen Zuweisungen geschehen:

```
S := ⟨s1,...,sN⟩ ;
S := compose(s1,...,sN);
```

Die Teilströme in `S` können dann ebenfalls auf zwei notationelle Arten extrahiert werden:

```
⟨s1,...,sN⟩ := S;
s1,...,sN := decompose(S);
```

Wie wir in diesem Abschnitt aufzeigen, werden in den einzelnen Inkarnationen der Funktionen oft nur wenige Ströme des Kabelbaumes gelesen bzw. erzeugt. Die anderen werden dagegen nur durchgeleitet. Wie bei ihren Vorbildern aus der Technik macht es aber für den Fluß durch die einzelnen Teilkabel eines Kabelbaums keinen Unterschied, wie diese verlegt oder zusammengefaßt werden. Der Strom fließt stets direkt vom Erzeuger zum Verbraucher. Die Implementierung der Kabelbäume hat deswegen die Aufgabe, das Senden von Daten

- ohne unnötige Synchronisation mit anderen Daten und
- ohne Umwege über Zwischenrechner *direkt* vom Erzeuger zu den Empfängern

zu gewährleisten.

Im folgenden werden wir an zwei Beispielen die Bedeutung der Kabelbäume illustrieren. Das erste Beispiel, ein iterativer Divide&Conquer–Algorithmus, zeigt, daß es mit Hilfe von Kabelbäumen möglich ist, trotz rein funktionaler Syntax das Arbeiten auf lokalen Daten am Platz zu ermöglichen. Dies ist für große numerische Anwendungen unabdingbar. Das zweite Beispiel veranschaulicht die Notwendigkeit der Kabelbäume, um die maximale Parallelität auszunutzen.

4.3.1 Divide&Conquer: das Problem der lokalen Daten

Die Grundidee von Divide & Conquer ist die rekursive Aufspaltung eines großen Problems in (zwei oder) mehrere kleinere Teilprobleme, die unabhängig voneinander gelöst werden können. Als Aufrufgraph ergibt sich somit ein (Binär-)Baum, dessen Knoten die jeweiligen Teilprobleme repräsentieren.

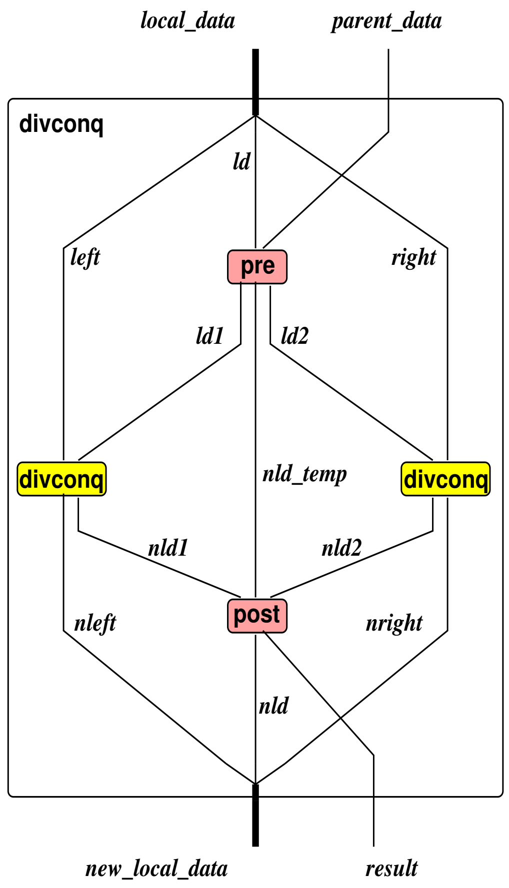
In der parallelen Numerik tritt dieses Verfahren häufig in Form von Gebietszerlegungsalgorithmen auf, die wir im vorigen Kapitel kennengelernt haben. Ein weiteres Beispiel dafür ist die Berechnung der statischen Kräfte, die in einem technischen Gebilde, etwa einem Kran, auftreten. Der Kran kann dann z.B. in den Turm und den Ausleger aufgeteilt werden, und die auftretenden Kräfte in beiden Gebieten unabhängig gerechnet werden. Dabei ist lediglich die Kopplung über den Separator zu berücksichtigen. Die beiden Teilgebiete können rekursiv weiter untergliedert werden.

Im allgemeinen gehören zu den einzelnen Teilproblemen lokale Datenstrukturen (z.B. große Matrizen), die der zugehörige Rechner lokal verwaltet und modifiziert. Bei vielen numerischen Anwendungen genügt ein Durchlauf nicht. In mehreren Iterationen wird der Aufrufbaum immer wieder durchlaufen und dabei die lokalen Daten so lange modifiziert, bis die erforderliche Genauigkeit erreicht worden ist.

In Abbildung 4.1 ist ein allgemeiner iterativer Divide&Conquer–Algorithmus in FASAN angegeben. Daneben steht das zugehörige Funktionennetz, das den Datenfluß im `else`-Zweig von `divconq` veranschaulichen soll. Wir wollen nun verfolgen, wie dabei die lokalen Daten modifiziert werden.

Zu Beginn wird geprüft, ob man bereits an einem Blatt des Aufrufbaumes angekommen ist, und ggf. die sequentielle Elementarfunktion `work` angestoßen. Ansonsten findet zunächst ein Preprocessing (`pre`) auf den eigenen, lokalen Daten `ld` statt. Hier fließen Informationen des Vaters (`parentdata`), in obigem Beispiel etwa die Kräfte auf den Rändern des Teilproblems, mit ein. Dabei werden unter anderem Daten für die Söhne (`ld1` und `ld2`) bereitgestellt, etwa die Kräfte auf den Separatoren. Danach erfolgt die Aufspaltung in zwei Teilprobleme, die parallel in den Sohnknoten berechnet werden (`divconq`). Die Ergebnisse der Söhne (`nld1` und `nld2`) werden dann in einem Postprocessing (`post`) in die lokalen Daten eingearbeitet.

Nun passen aber lokale Daten, die iterativ verändert werden, nicht in das Bild funktionaler Sprachen, das wir oben entworfen haben. Sie stellen ja quasi einen globalen Zustand dar.




```
function divconq(datatree,parentdata): newtree,result;
begin
   if isLeave(datatree) then
      ld := val(datatree);
      newtree, result := work(ld,parentdata);
   else
      ⟨left,ld,right⟩ := datatree;
      ld1,ld2,nldtemp := pre(ld,parentdata);
      nleft, nld1 := divconq(left,ld1) on Son1;
      nright,nld2 := divconq(right,ld2) on Son2;
      nld,result := post(nld1,nldtemp,nld2);
      newtree := ⟨nleft,nld,nright⟩
   fi
end

function iterate(datatree, eps): newtree,result;
begin
   nld,r := divconq(datatree, nodata());
   if error2big(r,eps) then
      newtree,result := recycle iterate(nld,eps);
   else
      result := postprocess(r);
      newtree := nil;
   fi
end
```

Abbildung 4.1: *FASAN-Programmtext (rechts) eines allgemeinen Divide&Conquer-Algorithmus'. Der Datenflußgraph (links) zeigt den `else`-Zweig der Funktion `divconq`.*

FASAN soll dagegen seiteneffektfrei sein und Variablen nur einmal zuweisen. Wir wollen zeigen, wie es in FASAN möglich ist, trotz rein funktionaler Beschreibung die effiziente Umsetzung der Lokalität der Daten zu gewährleisten.

Wenn wir dem funktionalen Charakter von FASAN treu bleiben wollen („kein globaler Zustand"), so müssen die alten lokalen Daten eines Knotens aus der vergangenen Iteration als Eingabestrom in die Funktionen fließen und die modifizierten Daten als Ausgabestrom erscheinen. Konsequenterweise müssen dann für jeden Knoten des Aufrufbaumes die lokalen Daten bis zur Wurzel gereicht werden und von dort wieder beim Absteigen auf die Knoten verteilt werden. Es entsteht so ein Baum von lokalen Daten. Dieses Verhalten wäre nicht nur unsinnig, es würde auch bei großen Anwendungen die Kapazitäten der oberen Knoten und des Verbindungsnetzwerkes sprengen. Hier helfen uns die Kabelbäume. Sie sorgen dafür, daß die lokalen Daten am Platz bleiben und keine überflüssigen Informationen über das Netz gehen.

In unserem Beispiel werden tatsächlich aus syntaktischer Sicht die lokalen Daten als Teilströme gekapselt bis zur Wurzel geleitet (`datatree`) und beim Abstieg wieder in den einzelnen Funktionen entflochten. Die jeweiligen Knoten haben dabei nur eine lokale Sicht. Wenn sie die äußere Hülle des Kabelbaums entfernen, so können sie lediglich ihren eigenen Datenstrom (`ld`) lesen. Die beiden Teilkabelbäume (`left, right`) sind dagegen gekapselt, d.h. ihre Teilströme sind an dieser Stelle nicht sichtbar.

Durch die Kabelbaumsemantik fließen die Daten direkt vom Sender zum Empfänger. Für unsere lokalen Daten bedeutet dies, daß sie stets am gleichen Ort bleiben, wenn Sender und Empfänger im gleichen Knoten sitzen. Genau dies garantiert die **recycle**-Anweisung.

Obwohl die Daten auf dem gleichen Rechner bleiben, müssen sie doch jedesmal neu kopiert werden. Aber auch dieser unerfreuliche Umstand, ein Hauptkritikpunkt gegen funktionale Ansätze, läßt sich in FASAN beheben, indem man lediglich die *Referenzen (Zeiger)* auf die Datenstrukturen in den Kabelbaum steckt. Da die Elementarfunktionen in Benutzerhand stehen und FASAN keine Datentypen kennt, ist dies ohne weiteres möglich. Trotz reiner funktionaler Syntax auf FASAN-Ebene ist es gelungen, langlebige lokale Daten effizient zu implementieren.

4.3.2 Maximale Parallelität und minimale Kommunikation

Die direkten Verbindungsströme, die durch die Kabelbaumentflechtung entstehen, sorgen auch für optimale Kommunikationswege in einem komplexen Netz paralleler Tasks. Unnötige Zwischenstationen oder Synchronisationspunkte, die nur wegen einer übersichtlicheren textuellen Beschreibung entstanden sind, können so aufgelöst werden.

Wir erinnern uns an die in Abschnitt 3.4 aufgeworfene Problematik, nämlich der sprachlichen Beschreibung der maximalen Parallelität bei der D-dimensionalen Überschußschleuder. Dort haben wir festgestellt, daß die gebietszerlegungsorientierte Formulierung in herkömmlicher Semantik nicht nur zu ungünstigen Kommunikationsstrukturen mit vielen unnötigen Zwischenstationen führte, sondern auch zu einer drastischen Komplexitätsverschlechterung von $O(T)$ auf $O(T^D)$.

Dies lag z.B. im zweidimensionalen Fall daran, daß die Punkte auf einer horizontalen Linie des Gitters ihre Werte über die Väter an den Mittelpunkt schicken mußten. Dieser setzte daraus einen Linienbaum zusammen und schickte ihn komplett an den nördlichen und südlichen Sohn, von wo aus der Baum wieder sukzessive auf die Punkte der jeweiligen horizontalen Linie verteilt wurde.

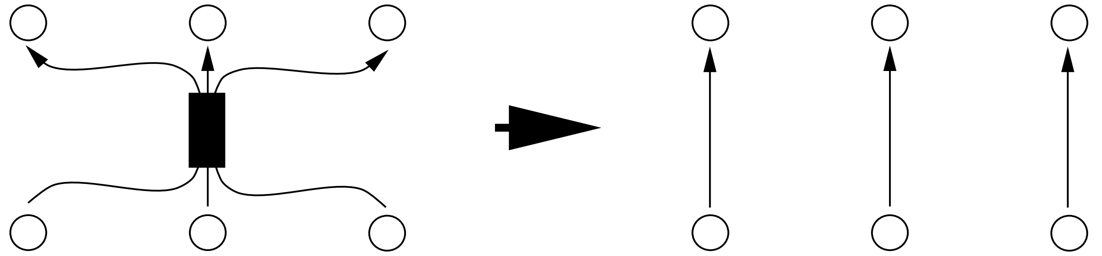


Abbildung 4.2: *Auflösung eines Kabelbaums: Syntaktisch gesehen fließen die Daten über die Zeilenmittelpunkte. Löst man die Kabelbaumhülle, so strömen die Daten direkt vom Erzeuger zum Verbraucher.*

Verwenden wir aber statt der herkömmlichen Binärbäume nun Kabelbäume zum Zusammenfassen und Aufteilen der Einflüsse der Gitterpunkte, so werden die Daten *direkt* vom Erzeugerknoten zum Empfängerknoten gesandt (vgl. Abb. 4.2). Somit wird also die Kommunikation beträchtlich vermindert.

Kombinieren wir das Kabelbaumprinzip nun noch mit der *Datenflußsemantik* von FASAN, so erreichen wir sogar die optimale Komplexität. Denn durch das unmittelbare Versenden der Daten eines Gitterpunktes auch auf die nördlichen und südlichen hierarchischen Söhne können diese bereits im darauffolgenden Zeitschritt weiterrechnen. Die unnötige Synchronisierung über die Zeilenmittelpunkte entfällt.

Im nächsten Kapitel werden wir die Thematik vertiefen und schrittweise die FASAN-Programme zur Lösung der D-dimensionalen Helmholtzgleichung erarbeiten. Dabei werden wir sehen, daß auch hier die Kombination aus Kabelbaum- und Datenflußsemantik für die optimale parallele Komplexität sorgt.

5 Funktionale Beschreibung des Helmholtzlösers

Nachdem wir in den vorigen Kapiteln die Problemstellung und die sprachlichen Werkzeuge zur Formulierung eines Lösungsalgorithmus' kennengelernt haben, wollen wir nun die Eignung von FASAN zur parallelen Beschreibung von Dünngitteralgorithmen untersuchen. Als konkretes Beispiel betrachten wir wieder die Matrixmultiplikation mit der Steifigkeitsmatrix der Helmholtzgleichung und verwenden dazu die in Abschnitt 3.3.2 beschriebene D-dimensionale Überschußschleuder.

Im folgenden Abschnitt beginnen wir mit dem eindimensionalen Fall und entwickeln schrittweise die wesentlichen Teile des FASAN-Programms zur Lösung der Helmholtzgleichung. Der darauffolgende Abschnitt behandelt die Matrixmultiplikation für den zweidimensionalen Fall.

Im letzten Teil des Kapitels verallgemeinern wir auf beliebige Dimensionalität D, d.h. hier ist D ein Parameter in dem Algorithmus. Diese Verallgemeinerung führte schon in bisherigen sequentiellen Versionen durch das Zusammenspiel von Gebietszerlegung und Rekursion in der Dimensionalität zu Programmen, die zwar erstaunlich kurz aber relativ schwierig zu verstehen waren. Jetzt verallgemeinern wir auch noch von einem Rechner auf mehrere, d.h. von einer sequentiellen Version des Programms auf eine parallele, die zudem noch die maximale Parallelität auszuschöpfen vermag.

Wie wir sehen werden, bleiben die Programme dennoch außerordentlich kurz und prägnant. Dies gelingt v.a. durch die in FASAN vorgegebene Trennung zwischen den numerischen Berechnungen (in den Elementarfunktionen) und den Abhängigkeitsbeziehungen (in den Netzfunktionen). Allerdings genügt auch hier für das Verständnis der Algorithmen eine oberflächliche Betrachtung nicht. Wir haben aber versucht, den interessierten Lesern durch entsprechende Verweise auf vorhergehende Kapitel und zahlreiche Abbildungen im Text eine entsprechende Unterstützung zu bieten, ohne die Darstellung übermäßig in die Länge zu ziehen.

5.1 Die eindimensionale Helmholtzgleichung

Wie wir in Kapitel 3 gesehen haben, müssen zur Berechnung der Matrixmultiplikation für jeden Knoten die Einflüsse der hierarchisch tieferen und der hierarchisch höheren Knoten berücksichtigt werden. Im Gegensatz zu bisherigen sequentiellen Varianten des Algorithmus' stellt in der parallelen Version der direkte Zugriff auf die entsprechenden Knoten ein Problem dar. Jetzt gilt es, die Abhängigkeits- und damit die Kommunikationsstruktur zwischen den Knoten möglichst stark zu vereinfachen, damit nur eine minimale Anzahl von Zugriffen nötig ist. Deshalb war es das Ziel, den Algorithmus so zu gestalten, daß alle Datentransporte nur noch von hierarchischen Vätern auf deren Söhne und umgekehrt stattfinden.

Erinnern wir uns an die eindimensionale Überschußschleuder aus Abschnitt 3.2, die die Beiträge der hierarchisch tieferen Knoten mit korrekter Gewichtung an die darüberliegenden Knoten liefern soll. Wenn wir uns die Basispunkte wie üblich binärbaumartig angeordnet denken, so werden die hierarchischen Überschüsse von den Blättern des Baumes beginnend mit dem Faktor $\frac{1}{2}$ an die jeweiligen hierarchischen Nachbarn geworfen. Während einer der beiden Nachbarn eines Knotens gleichzeitig der hierarchische Vater ist, kann sich der andere an einer beliebigen Stelle weiter oben im Baum befinden.

In einem sequentiellen Algorithmus würde nun einfach dieser zweite Nachbar durch Zeiger oder geeignete Berechnungen ausfindig gemacht und der Wert direkt an ihn zugewiesen. In der parallelen Version ist diese Methode unvorteilhaft, da ja zudem auch noch der Rechner, auf dem der zweite Nachbar liegt, bestimmt werden müßte. Außerdem würde die oben beschriebene Vorgehensweise in einem komplizierten Kommunikationsmuster resultieren (vgl. dazu die Bemerkungen am Ende von Kapitel 2).

In FASAN stehen uns ohnehin keine Mittel zum aktiven Senden von Nachrichten zur Verfügung. Daten können lediglich als Aufrufparameter einem Funktionsaufruf übergeben oder als Ausgabeparameter von einer aufgerufenen Funktion zurückgegeben werden. Der Algorithmus kann folglich nur so aussehen, daß der hierarchische Vater die Beiträge seiner Söhne als Ausgabeparameter zweier Funktionsaufrufe auf den beiden Sohnknoten erhält. Um nun auch deren Einflüsse zu dem zweiten hierarchischen Nachbarn weiterleiten zu können, bedienen wir uns eines einfachen Tricks: jeder Funktionsaufruf auf den Söhnen liefert *zwei* Zahlen `l`, `r`, die die Beiträge auf den linken bzw. rechten hierarchischen Nachbarn des jeweiligen Sohnknotens beinhalten. Einer dieser Nachbarn ist der aufrufende Knoten selbst, so daß die entsprechende Zahl für ihn bestimmt ist. Der andere Wert wird einfach an den eigenen Vater weitergegeben, wobei zuvor der eigene Beitrag aufaddiert werden muß.

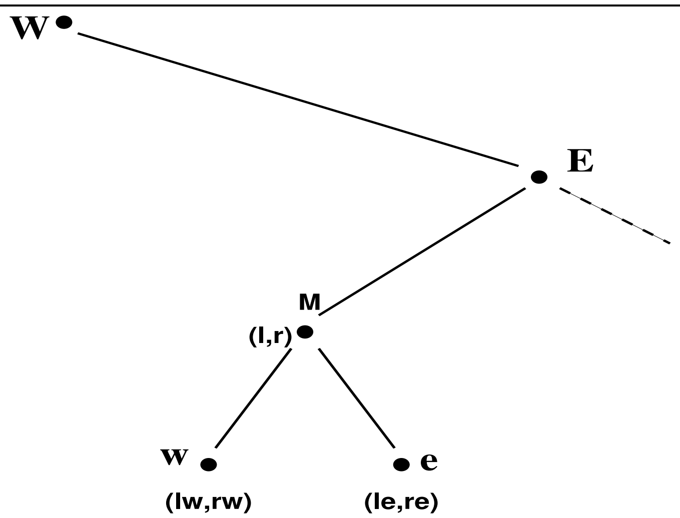


Abbildung 5.1: *Weiterleitung der Beiträge zu den hierarchischen Nachbarn.*

Diese für das Verständnis aller folgenden Up-Programme wichtige Technik soll durch Abbildung 5.1 verdeutlicht werden. Das Blatt `w` des Binärbaumes bestimmt seine Beiträge `lw, rw` für seine beiden hierarchischen Nachbarn `W` und `M`. Das gleiche macht Blatt `e` für seine hierarchischen Nachbarn `M` und `E`. Diese Beiträge werden dem aufrufenden Vater `M` übergeben, der für beide Söhne zugleich einer der hierarchischen Väter ist. So sind die Werte `rw` und `le` für ihn bestimmt. Die beiden anderen Werte muß er dagegen weiterleiten. Da sein linker hierarchischer Nachbar `W` auch der linke hierarchische Nachbar von `w` ist, muß er zu seinem eigenen Beitrag den Wert `lw` hinzurechnen. Das Ergebnis steht dann in `l`. Das gleiche gilt auch für den Beitrag `re`, der in `r` einfließt und für `E` bestimmt ist.

Der Up-Teil der Matrixmultiplikation lautet demnach:

```
function up(G,h): G1,l,r;
begin
  <Gw,g,Ge>:= G;
  if isNil(Gw) then          // g ist ein Blatt
    G1w := nil;
    G1e := nil;
    g1,l,r := up_leaf(g,h);
      // l = r = 1/2*h*g;
      // g1 = 2/3*h*g;
  else
    h2   :=  half(h);              //      h2:= h/2
    G1w,lw,rw   := up(Gw,h2) on west;
    G1e,le,re   := up(Ge,h2) on east;
    g1,l,r :=   up_calc(g,h,lw,rw,le,re);
      // g1 = rw + le + 2/3 h*g
      // l = 0.5(rw+le+h*g) + lw
      // r = 0.5(rw+le+h*g) + re
  fi
  G1 :=   <G1w,g1,G1e>;
end;
```

Der Binärbaum `G` beinhaltet die Knotenwerte und entspricht den in Abschnitt 4.3.1 beschriebenen lokalen Daten. Er wird zunächst aufgeteilt in einen westlichen und östlichen Teilbaum (`Gw, Ge`) und den Wert `g` an dem eigenen Knoten. Wenn der Knoten ein Blatt ist, müssen wir lediglich seine Beiträge `l` und `r` an den Vater berechnen und den eigenen Knotenwert mit der Elementarfunktion `up_leaf` zu `g1` verändern. Ansonsten benötigen wir erst die Einflüsse der Sohnknoten, die wir durch rekursiven Aufruf von `up` auf den beiden Teilbäumen erhalten. Die beiden Teilberechnungen finden parallel auf den von `west` bzw. `east` vorgeschlagenen Prozessoren statt. Die Frage der Lastverteilung, d.h. der konkreten Implementierung der Lokationen, überlassen wir Kapitel 6.

Als Abbruchkriterium dient uns hier und im folgenden der Test auf leere Teilbäume. In

FASAN wird ein leerer Strom mit `nil` bezeichnet, der Test auf die Leerheit eines Stroms geschieht mit der vordefinierten Funktion `isNil`. Wegen der Regularität des dünnen Gitters genügt die Überprüfung des westlichen Teilbaumes.

Die Funktionen `up_leaf` und `up_calc` sind Elementarfunktionen, die in einer Standardsprache kodiert sind. Die dort ausgeführten numerischen Berechnungen sind der Vollständigkeit halber in den Kommentaren im Text beschrieben (Kommentare werden in FASAN wie in C++ durch ein „ // “ eingeleitet). Wir verweisen in diesem Zusammenhang auf die Ausführungen in Kapitel 3.2.

Um die Matrixmultiplikation zu vervollständigen, müssen wir in einem anschließenden `down`-Zyklus den Einfluß der hierarchisch höheren Knoten auf die tieferen Knoten verarbeiten. Dazu benötigen wir, wie in Abschnitt 3.2 erläutert wurde, als zusätzliche Eingabeparameter die dehierarchisierten Werte `w,e` auf den beiden Randpunkten.

```
function   down(G,G1,w,e,h,c): Q;
begin
   <G1w,g1,G1e> := G1;
   <Gw,g,Ge> := G;
   m,q    := down_calc(g,g1,w,e,h,c);
          //    m     =    g + 1/2*(w+e)     =   dehier. g
          //    g1' =   g1 + 1/2*h*(w+e)
          //    q      = -2*g/h + c*g1'
   if isNil(Gw) then
      Qw := nil;
      Qe := nil;
   else
      h2 :=  half(h);                 //    h2:= h/2
      Qw := down(G1w,Gw,w,m,h2,c) on west;
      Qe := down(G1e,Ge,m,e,h2,c) on east;
   fi
   Q := <Qw,q,Qe>;
end;
```

Die eigentlichen Berechnungen finden wieder in einer Elementarfunktion statt: In `down_calc` wird zunächst der aktuelle Knotenwert mit Hilfe der zugehörigen Randwerte `w` und `e` dehierarchisiert. Anschließend wird die Matrixmultiplikation vervollständigt und der Wert in `q` abgelegt. Dieser setzt sich zusammen aus dem Laplaceteil $-2gh$ und dem Helmholtzteil cg_1, wobei `c` für die Helmholtzkonstante steht.

Die Matrixmultiplikation bildet den Kern des iterativen Helmholtzlösers. In einem Schritt eines Iterationsverfahrens (z.B. des CG-Verfahrens) werden die berechneten Knotenwerte benutzt, um eine bessere Approximation für die Lösung zu gewinnen. Außerdem wird ein Maß für den Iterationsfehler `error` bestimmt. Ist dieser größer als eine vorgegebene Schranke `eps`, so wird ein neuer Iterationsschritt angehängt. Dank des in Kapitel 4 vorgestellten Recycling-Prinzips kann dabei der alte Datenflußgraph, bestehend aus einem Up-Down-Solve-Block, weiterverwendet werden. Der Verwaltungsaufwand für das Erzeugen des Graphen entfällt. Die Iteration terminiert in einem Postprocessing-Schritt, wenn die gewünschte Genauigkeit erzielt worden ist. Das Postprocessing kann z.B. ein Maß für die Güte der Lösung liefern. Wir erhalten somit folgenden (vereinfachten) Iterationsalgorithmus:

```
function   iteration (G,w,e,eps,h0,c): result;
begin
    G1,l,r   := up(G,h0);
    Q    := down(G,G1,w,e,h0,c);
    G_,error    := solve(G,Q,h0);
    if error2big(error,eps) then
        result := recycle iteration(G_,w,e,eps,h0,c);
    else
        result := postprocess(G_,w,e,h0);
    fi;
end;
```

Der Parameter `h0` steht für die Anfangsschrittweite $h_0 = 0.5$.

5.2 Die zweidimensionale Helmholtzgleichung

Dieses Kapitel dient in erste Linie als Vorbereitung des D-dimensionalen Falls. Der Leser soll ein intuitives Verständnis für die Problematik und Lösungsstrategien bei höherer Dimensionalität gewinnen.

Beim Übergang auf den zweidimensionalen Fall benötigen wir zur Repräsentation der verschiedenen Werte auf dem dünnen Gitter Bäume, deren Knoten wiederum Bäume sind (vgl. Abbildung 2.7). In den folgenden Programmen werden zur Verdeutlichung einfache Bäume mit einem, Bäume von Bäumen dagegen mit zwei Großbuchstaben bezeichnet. Zur Durchführung der Überschußschleuder müssen wir zu den bisherigen Werten g und g^1 nun auch noch g^2 und g^{12} in jedem Gitterpunkt verwalten (vgl. Abschnitt 3.3). Deshalb sind die Knotenwerte des Baumes nun keine Zahlen, sondern Felder der Länge 4.

Außerdem werden die Funktionen zur Beschreibung des Up- bzw. Down-Zyklus' komplexer. So müssen wir zwischen dem 2D-Up/Down, das auf einem zweidimensionalen Gebiet operiert, und einem linienorientierten 1D-Up/Down, das von ihm aufgerufen wird, unterscheiden. Während die zweidimensionalen Vertreter nur Einflüsse in Nord-Süd-Richtung verwalten, müssen sich die eindimensionalen Funktionen auch noch um die West-Ost-Richtung kümmern, wodurch die Parameterliste im Vergleich zur Version im vorigen Kapitel länger wird. Dieses auf den ersten Blick sonderbare Gebaren rührt daher, daß ein

Punkt erst dann gerechnet werden kann, wenn die Einflüsse aus allen Dimensionen vorliegen. Beim rekursiven Zerlegen des Gebiets in Nord-/Süd-Richtung (durch die zweidimensionalen Up- und Down-Funktionen) kann aber nur die 2. Dimension berücksichtigt werden. Erst bei der Rekursion in der Dimension (durch Aufruf der eindimensionalen Up- und Down-Funktionen) kommen noch die Einflüsse in der 1. Dimension hinzu.

Betrachten wir zunächst die flächenorientierte

```
function    up2D(GG,h1,h2):  GG_,O,U;
begin
    <GGn,G,GGs>:= GG;
    if isNil(GGn) then
        GG_n,On,Un := nil,nil,nil;
        GG_s,Os,Us := nil,nil,nil;
    else
        h22 :=    half(h2);
        GG_n,On,Un :=    up2D(GGn,h1,h22)  on north;
        GG_s,Os,Us :=    up2D(GGs,h1,h22)  on south;
    fi
    G_,ignore_l,ignore_r,O,U := up1D(G,On,Un,Os,Us,h1,h2);
    GG_ := <GG_n,G_,GG_s>;
end;
```

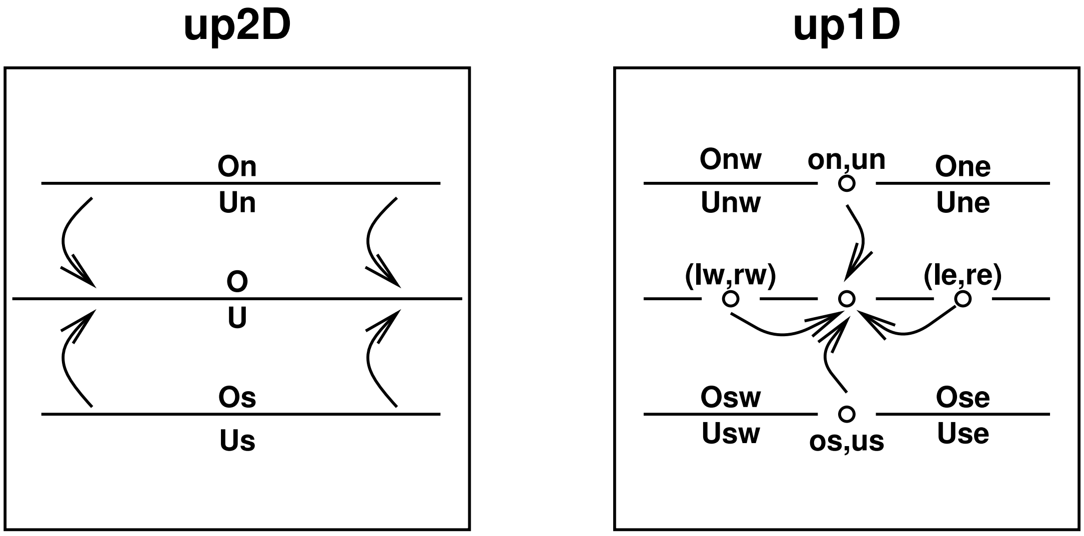


Abbildung 5.2: *Schematische Darstellung der Funktionen* `up2D` *und* `up1D`.

Das Gebiet wird in Nord/Süd-Richtung halbiert und die Einflüsse der Teilgebiete (bzw. ihrer Mittellinien) auf die Separatorlinie bestimmt. Statt zweier Werte `l,r` wie im eindimensionalen Fall müssen nun die Einflüsse kompletter Linien `O,U` in den Ausgangsströmen von `up2D` gesammelt und ausgegeben werden (vgl. Abbildung 5.2).

Die eigentlichen Berechungen finden in dem linienorientieren `up1D` statt, genauer gesagt, in den Elementarfunktionen, die darin aufgerufen werden:

```
function    up1D(G,On,Un,Os,Us,h1,h2):  G_,l,r,O,U;
begin
    <Gw,g,Ge>  :=  G;
    if  isNil(Gw)  then          // Blatt erreicht
        g_,l,r  :=  up_leaf(g,h1);     // g -> g1
        g__,o,u  :=  up_leaf(g_,h2); // g -> g2 , g1 -> g12

        G_w,Ow,Uw  :=  nil,nil,nil;
        G_e,Oe,Ue  :=  nil,nil,nil;
    else
        <Onw,on,One>  :=  On;
        <Osw,os,Ose>  :=  Os;
        <Unw,un,Une>  :=  Un;
        <Usw,us,Use>  :=  Us;

        h12  :=  half(h1);
        G_w,lw,rw,Ow,Uw  :=  up1D(Gw,Onw,Unw,Osw,Usw,h12,h2)    on  west;
        G_e,le,re,Oe,Ue  :=  up1D(Ge,One,Une,Ose,Use,h12,h2)    on  east;

        g_,l,r          :=  up_calc(g,h1,lw,rw,le,re);
        g__,o,u  :=  up_calc(g_,h2,on,un,os,us);
    fi
    G_  :=  <G_w,g_,G_e>;
    O  :=  <Ow,o,Oe>;
    U  :=  <Uw,u,Ue>;
end;
```

Die Funktion operiert dabei wie ihr eindimensionales Pendant aus dem vorhergehenden Abschnitt und bearbeitet rekursiv die linke und rechte Teillinie. Zudem müssen aber auch die Einflüsse aus Nord/Süd-Richtung, die von `up2D` mitgegeben worden sind, einbezogen werden.

Bei dem zweidimensionalen Down-Algorithmus ist die Aufgabenverteilung zwischen `down2D` und `down1D` analog zum Up-Algorithmus. Während `down2D` die Einflüsse in Nord/Süd-Richtung verwaltet, kümmert sich `down1D` auch um die West/Ost-Richtung:

```
function    down2D(GG,W,E,N,S,h1,h2,c):  QQ;
begin
    <GGn,G,GGs>  :=  GG;
    <Wn,w,Ws>  :=  W;
    <En,e,Es>  :=  E;
    Q,M  :=  down1D(G,w,e,N,S,h1,h2);
        // M: in NS-Richtung deh. Mittellinie bzgl. GG,GG1
    if  isNil(GGn)  then
        QQn  :=  nil;
```

```
        QQs := nil;
    else
        h22   :=  half(h2);
        QQn := down2D(GGn,Wn,En,N,M,h1,h22)  on north;
        QQs := down2D(GGs,Ws,Es,M,S,h1,h22)  on south;
    fi
    QQ := <QQn,Q,QQs>;
end;

function   down1D(G,w,e,N,S,h1,h2,c): Q,M;
begin
    <Gw,g,Ge>:= G;
    <Nw,n,Ne>:= N;
    <Sw,s,Se>:= S;
    g_,m1 := down_calc(g,w,e,h1); // g -> g1 , g2 -> g12
    g__,m := down_calc(g_,n,s,h2);  // g -> g2 , g1 -> g12
    q:= build_q(g__,h1,h2,c);       // q:= c*g12 - 2*(g1/h2 + g2/h1)
    if isNil(Gw) then
        Qw,Mw := nil,nil;
        Qe,Me := nil,nil;
    else
        h12 := half(h1);
        Qw,Mw := down1D(Gw,w,m1,Nw,Sw,h12,h2) on west;
        Qe,Me := down1D(Ge,m1,e,Ne,Se,h12,h2) on east;
    fi;
    M := <Mw,m,Me>;
    Q := <Qw,q,Qe>;
end;
```

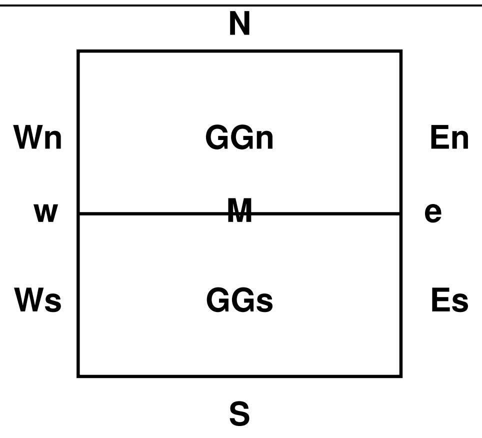


Abbildung 5.3: *Rekursive Gebietszerlegung im `down2D`.*

Die rekursive Zerlegung des zweidimensionalen Gebiets durch `down2D` wird in Abbildung 5.3 veranschaulicht.

In den Randlinien `N` und `S` sind die bzgl. der 1-Richtung dehierarchisierten Größen g und g^2 enthalten, in `W` und `E` die bzgl. der 2-Richtung dehierarchisierten Größen g und g^1 (vgl. Abbildung 3.3 in Abschnitt 3.3.2).

An dieser Stelle sei noch einmal darauf hingewiesen, daß wir trotz der einfachen Gebietszerlegungsstrategie optimale Parallelisierung auf Punktebene erzielt haben. Obwohl z.B. in `down2D` die Mittellinie `M` rein textuell betrachtet *vor* den Gebietshälften berechnet wird und obwohl `M` in der Parameterliste der darauffolgenden `down2D`-Routinen auftaucht, sorgen die Kabelbaum- und die Datenflußsemantik von FASAN für eine sofortige Bereitstellung der Knotenwerte des Separators an die abhängigen Punkte der Teilgebiete. Diese können somit parallel zu `M` berechnet werden.

5.3 Verallgemeinerung auf beliebige Dimensionalität

Wir abstrahieren nun von Speziallösungen für feste Dimensionalität und entwerfen einen FASAN-Algorithmus, bei dem die Dimensionalität als Parameter übergeben wird.

Das D-dimensionale dünne Gitter wird konsequenterweise nun durch einen D-dimensionalen Binärbaum repräsentiert, in dessen Knoten $(D-1)$-dimensionale Binärbäume stehen. Diese Baumstruktur beinhaltet für jeden Punkt des Gitters eine Referenz auf einen 2^D-dimensionalen Vektor, der die Werte $g, g^1, g^2, \ldots, g^D, g^{12}, \ldots, g^{1\ldots D}$ faßt. In Abschnitt 3.3.2 wurde ausführlich dargelegt, wie die Komponenten des Vektors modifiziert werden müssen. Wie wir sehen werden, kann auf diesem Vektor am Platz gearbeitet werden, d.h. der alte Speicher wird in jeder Iteration wiederverwendet. Dies ist für die Effizienz der Programme von wesentlicher Bedeutung. In diesem Zusammenhang möchten wir nochmal an die in Abschnitt 4.3.1 diskutierte Problematik der lokalen Daten erinnern.

Bei der allgemeinen Beschreibung der D-dimensionalen Up- und Down-Routinen stoßen wir auf Schwierigkeiten. Denn wie wir beim Übergang vom eindimensionalen auf den zweidimensionalen Fall gesehen haben, werden die Parameterlisten der Funktionen immer länger und für jede Dimension, die bei dem rekursiven Durchlauf auftritt, mußte eine eigene Up- bzw. Down-Funktion entworfen werden.

Der Ausweg besteht nun darin, Teile der komplexen Parameterlisten (wie etwa die Ränder in `down`) in eine einzige Liste zu packen, die dann als *ein* Parameter in die Funktionen eingeht. Die gleiche Technik wird auch auf die Ausgabeparameter angewandt. Rekursive Listen können in FASAN bislang nur durch Kabelbäume implementiert werden. Durch diese zweite Bedeutung der Kabelbäume sollte sich der Leser beim Studium der folgenden Programme nicht verwirren lassen.

Die Up- und Down-Funktionen rufen sich rekursiv selbst auf, wobei die Parameter-Kabelbäume geeignet aufbereitet werden müssen. Zudem werden stets zwei Zahlen D und d übergeben, die für die Dimensionalität des Gesamtproblems bzw. des gerade zu berechnenden Teilgebietes stehen. Die Rekursion terminiert, wenn $d = 0$ (ein Punkt) erreicht ist. Dann müssen die Beiträge aus allen Richtungen für den jeweiligen Punkt als Kabelbaum vorhanden sein. Eigene Funktionen `down0` bzw. `up0` arbeiten den Kabelbaum rekursiv ab und stoßen die eigentlichen Berechnungen an.

Um zu sehen, wie die einzelnen Kabelbäume modifiziert werden müssen, analysieren wir im folgenden die Veränderung der Parameterlisten im Zuge der rekursiven Abarbeitung. Zunächst widmen wir uns dem D-dimensionalen Up-Zyklus.

5.3.1 Die D-dimensionale Up-Funktion

Im Up-Durchlauf müssen in jedem Punkt des Gitters die Beiträge aller hierarchischen Söhne berücksichtigt und die Einflüsse auf die hierarchischen Väter berechnet werden. Genauer gesagt, werden in Richtung d die Größen $g^{i_1,\ldots,i_j}$ mit $j \in \{0,\ldots,d-1\}$ und $i_1 < \ldots < i_j < d$ benötigt (s. wieder Abschnitt 3.3.2).

Diese Beiträge werden im Laufe der rekursiven Bearbeitung aufgesammelt, wobei in jeder Dimension die Einflüsse der höheren Dimensionen bekannt sind. Ausgehend von dem kompletten D-dimensionalen Hyperwürfel wird nämlich die Berechnung analog zum 2-dimensionalen Fall sukzessiv auf niederdimensionale Probleme zurückgeführt, bis die Rekursion für $d = 0$ terminiert. Ein d-dimensionales Gebiet wird dabei stets in zwei Hälften gleicher Dimensionalität und deren $(d-1)$-dimensionalen Separator aufgeteilt.

Der folgende Programmtext beschreibt die D-dimensionale Up-Funktion:

```
function  up(D,d,G,h,P): G_,ThrowList,L,R;
begin
  <Gl,g,Gr>:= G;
  Pl,p,Pr := decomp_rek_llMrr(P);

  if isNil(Gl) then
    G_l,TLl,Ll,Rl := nil,nil,nil,nil;
    G_r,TLr,Lr,Rr := nil,nil,nil,nil;
  else
    h2 := half(h,d);
    G_l,TLl,Ll,Rl    := up(D,d,Gl,h2,Pl) on new_proc_l;
    G_r,TLr,Lr,Rr    := up(D,d,Gr,h2,Pr) on new_proc_r;
  fi;
  d_1 := decr(d);
  if isNull(d_1) then
    g_,Tlist_1:= up0(D,d_1,g,h,<Ll,Rl,p,Lr,Rr>);
  else
    g_,Tlist_1,ignore1,ignore2 := up(D,d_1,g,h,<Ll,Rl,p,Lr,Rr>);
  fi;
```

```
    if isNil(Tlist_1) then
        L,tlist,R := nil,nil,nil;
    else
        <L,tlist,R> := Tlist_1;
    fi
    ThrowList := comp_rek_lMr(TLl,tlist,TLr);
    G_ := <G_l,g_,G_r>;
end;
```

Abbildung 5.4 illustriert die Veränderung der Ein- und Ausgabelisten für den dreidimensionalen Fall. Die Anzahl der Großbuchstaben in den Parameternamen entspricht ihrer Dimensionalität.

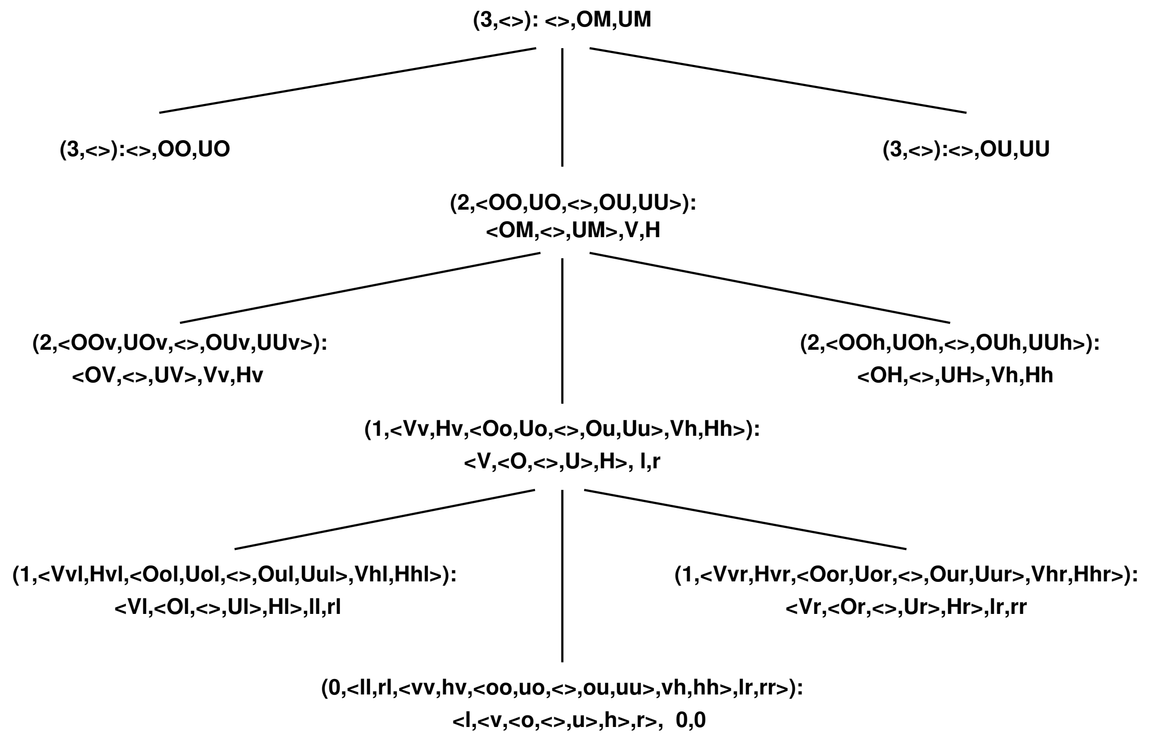


Abbildung 5.4: *Beispiel einer rekursiven Abarbeitung des Up-Zyklus' für $D = 3$. Dargestellt werden up$(d, P) : Throwlist, L, R$.*

Die Überschüsse in der aktuellen Dimension d werden in zwei Kabelbäumen `L` und `R` ausgegeben, die von der Dimensionalität $d-1$ sind (vgl. `up2D`). Sie werden von den hierarchisch höher liegenden Hyperebenen derselben Dimensionalität benötigt. Andererseits müssen auch die Überschüsse des zugeordneten Gebiets in allen höheren Dimensionen bereitgestellt werden. Diese werden in einer rekursiven Liste `ThrowList` verwaltet. Die Liste wird natürlich umso länger, je niedriger die Dimension ist.

Mit den Eingabeparametern verhält es sich analog: mit sinkender Dimension kommen immer mehr Einflüsse aus höheren Dimensionen hinzu. Wenn die Rekursion schließlich in einem Punkt terminiert, so stehen alle D Beiträge der hierarchischen Söhne zur

Verfügung. Diese werden in dem Punkt verarbeitet und dessen Einfluß auf die $2D$ hierarchischen Nachbarn ausgegeben.

In Abbildung 5.4 ist ebenfalls erkennbar, daß die Liste `P` gemäß der rekursiven Aufteilung in zwei Hälften und Separator gespalten werden muß – allerdings nicht in ihre Teilströme, sondern jeder ihrer Teilströme für sich. Man nehme als Beispiel etwa Kabelbaum `P=<OO,UO,<>,OU,UU>` aus der mittleren Spalte (2. Zeile) der Graphik. Jeder der Teilkabelbäume aus `P` wird nun *für sich* zerteilt, z.B. `OO` in `OOv`, `Oo` und `OOh`. Diese Teile werden dann wieder zusammengesetzt in `<OOv,UOv,<>,OUv,UUv>`, `<Oo,Uo,<>,Ou,Uu>` und `<OOh,UOh,<>,OUh,UUh>`, und an die entsprechenden Gebietshälften bzw. an den Separator übergeben.

Für den Separator muß `P` um vier Kabelbäume für die nächste Dimension erweitert werden. Diese stammen von den Aufrufen auf den beiden Hälften.

Wie gerade angedeutet wurde, müssen die Kabelbäume in `P` bzw. `ThrowList` „quer" zu ihrer Zerlegungsstruktur gespalten bzw. zusammengesetzt werden, das heißt, der Aufbau des Kabelbaumes bleibt der gleiche, nur seine Bestandteile werden verändert. Dies kann nicht mit einem einzigen Aufruf der Standardfunktionen `compose` bzw. `decompose` bewerkstelligt werden, sondern muß durch rekursives Abarbeiten der (durch den Kabelbaum repräsentierten) Liste kodiert werden. Wir geben exemplarisch eine dieser Funktionen an. Wichtig ist in diesem Zusammenhang, daß während der rekursiven Umstrukturierung des Kabelbaums *kein Lesezugriff* erfolgt. Bei seiner späteren Entflechtung entsteht also kein Zwischenknoten durch diese Verwaltungsfunktion. Die Daten fließen nach wie vor direkt vom Erzeuger zum Empfänger.

```
function    comp_rek_lMr(Bo,b,Bu):B;
   // <Bol,Bom,Bor>,<bl,Bm,br>,<Bul,Bum,Bur>   ->
   //        ->         <<Bol,bl,Bul>,<Bom,Bm,Bum>,<Bor,br,Bur>>
begin
   if isNil(b) then
      B := nil;
   else
      <Bol,Bom,Bor>:= Bo;
      <bl,Bm,br>:= b;
      <Bul,Bum,Bur>:= Bu;
      BM := comp_rek_lMr(Bom,Bm,Bum);
      Bl := <Bol,bl,Bul>;
      Br := <Bor,br,Bur>;
      B  := <Bl,BM,Br>;
   fi;
end;
```

Ist die Rekursion an einem Punkt angelangt, so liegen die vollständigen Einflußgrößen aus allen Richtungen in der rekursiven Liste `P` vor. Diese wird in der nun folgenden Funktion `up0` abgearbeitet und für jede Richtung die eigentliche Berechnung angestoßen:

```
function   up0(D,d,g,h,P): g__,T;
begin
  if isNil(P) then
    g__ := g; T := nil;
  else                    // g[index] -> g[index,d] werfen
    <ll,rl,PRest,lr,rr> := P;
    if isNil(ll) then
      g_,l,r := up_leaf(D,d,g,h);
    else
      g_,l,r := up_calc(D,d,g,h,ll,rl,lr,rr);
    fi;
    g__, TRest := up0(D,incr(d),g_,h,PRest);   // die nächste Richtung
    T := <l,TRest,r>;
  fi;
end;
```

Verbleiben nur noch die Funktionen `up_calc` und `up_leaf`, von denen wir hier die erste in C-Notation angeben:

```
void up_calc_c(int D,int d,double* g,int glen, double* h,int hlen,double* ll,
   int lllen, double* rl,int rllen,double* lr,int lrlen,double* rr,int rrlen,
  /* OUT */ double **g_,int *g_len,double **l,int *llen, double ** r,int *rlen)
{
  double x;
  int i;

  *llen = lllen;                        /*  = pow2(d) */
  *l = (double*) malloc(sizeof(double)*(*llen));
  *rlen = rllen;
  *r = (double*) malloc(sizeof(double)*(*rlen));
  for (i=0; i<pow2(d); i++) {
    x =  h[d] * g[i] + rl[i] + lr[i];
    (*l)[i] = 0.5 *x + ll[i];
    (*r)[i] = 0.5 *x + rr[i];
    g[i+pow2(d)] = x - (h[d] * g[i])/3.0;   /*  = 2/3 hn*g + rl + lr */
  }
  *g_len = glen;
  *g_ = g;
}
```

Nach Konvention muß in der Parameterliste der C-Funktion für jedes Array zusätzlich noch eine Integer-Variable stehen, die dessen Länge übergibt. Die Funktion `up_leaf_c` unterscheidet sich von `up_calc_c` lediglich darin, daß die Vektoren `ll,lr,rl` und `rr` wegfallen.

Wie der Programmtext verdeutlicht, wird der Vektor g am Platz modifiziert. Die funktionale Sichtweise der aufrufenden FASAN-Funktion up bedingt allerdings, daß ein zweiter Zeiger g_ angelegt werden muß, der auf den alten Vektor zeigt.

5.3.2 Die D-dimensionale Down-Funktion

Im Down-Durchlauf müssen im Gegensatz zum Up-Teil an jedem Punkt p des Gitters die Einflüsse der hierarchisch höheren Punkte berücksichtigt werden. Wie in Abschnitt 3.3.2 erklärt wurde, müssen dazu die dehierarchisierten Größen der $2D$ hierarchischen Nachbarn von p bekannt sein, genauer gesagt, für die Dimension d die in d-Richtung dehierarchisierten Werte $g^{i_1,\ldots,i_j}$ mit $j \in \{0, \ldots, D-1\}$ und $i_1, \ldots, i_j \neq d$. Nach Definition liegen die hierarchischen Nachbarn auf dem Rand der zu p gehörenden Basisfunktion. Bei der rekursiven Abarbeitung wird deshalb erst der $(d-1)$-dimensionale Separator in der Mitte des d-dimensionalen Hyperwürfels berechnet und dieser dann als Rand für die beiden d-dimensionalen Hälften verwendet.

```
function    down(D,d,G,h,Done,Todo): G_,M,Mlist;
begin
    <Gl,g,Gr>:= G;
        // teile Done rek. in Dim. d auf
    Dl,done,Dr := decomp_rek_lMr(Done);
    <NowTodoL,YetTodo,NowTodoR> := Todo;
        // teile YetTodo rek. in Dim. d auf
    YTl,yettodo,YTr := decomp_rek_lMr(YetTodo);
    d_1 := decr(d);
    if isNull(d_1) then
        g_,M_1,Mlist_1 := down0(D,d_1,g,h,<NowTodoL,done,NowTodoR>);
    else
        g_,M_1,Mlist_1 := down(D,d_1,g,h,<NowTodoL,done,NowTodoR>,yettodo);
    fi;
    if isNil(Gl) then
        G_l := nil; Ml := nil; Mlistl := nil;
        G_r := nil; Mr := nil; Mlistr := nil;
    else
        h2 := half(h,d);
        G_l,Ml,Mlistl := down(D,d,Gl,h2,Dl,<NowTodoL,YTl,M_1>) on new_proc_l;
        G_r,Mr,Mlistr := down(D,d,Gr,h2,Dr,<M_1,YTr,NowTodoR>) on new_proc_r;
    fi;
    if isNil(Mlist_1) then
        m,mlist := nil,nil;
    else
        <m,mlist> := Mlist_1;
    fi;
```

M := <Ml,m,Mr>;
Mlist := comp_rek_mM(Mlistl,mlist,Mlistr);
G_ := <G_l,g_,G_r>;
end;

In Done und Todo werden die Einflüsse von den Rändern des aktuellen Gebiets verwaltet. Done ist dabei der schon erledigte Teil, während Todo noch einbezogen werden muß. Abbildung 5.5 zeigt für ein dreidimensionales Problem die Entwicklung der Ausgabeliste und der Parameter Done und Todo. Abbildung 5.6 soll die Gebietszerlegung geometrisch veranschaulichen.

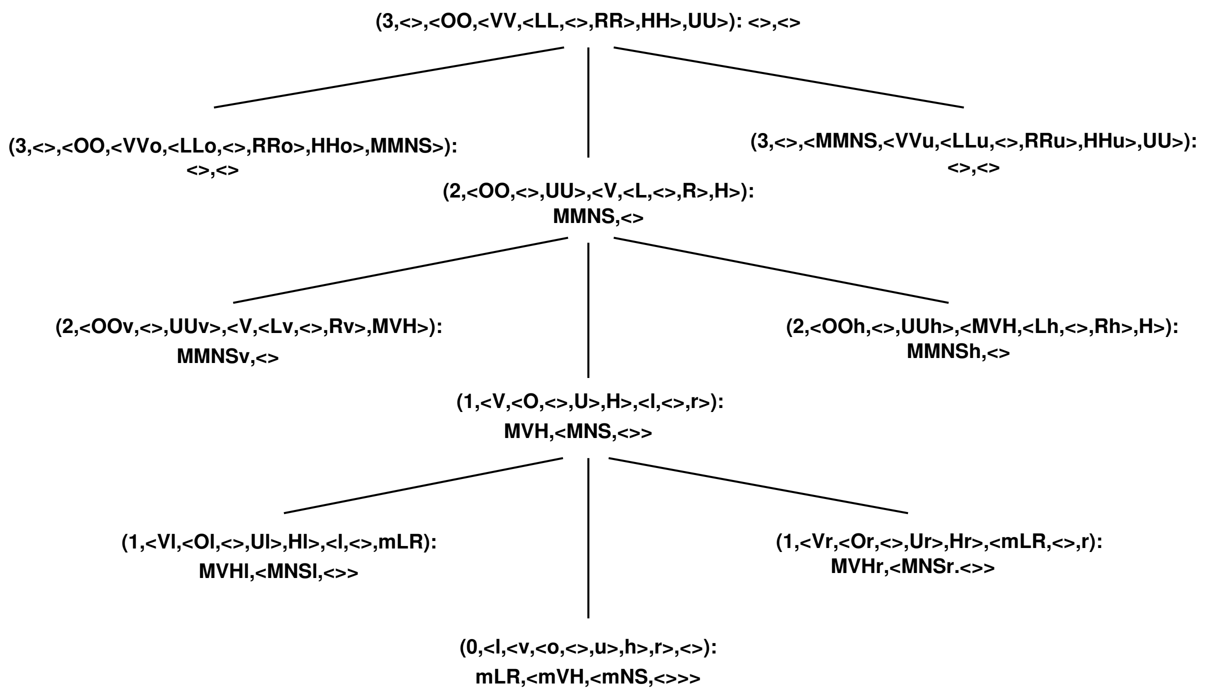


Abbildung 5.5: *Beispiel für die rekursive Abarbeitung des Down-Zyklus' ($D = 3$). Dargestellt werden down$(d, Done, Todo) : M, Mlist$.*

Zuerst müssen die in Done und Todo übergebenen Ränder gemäß der rekursiven Gebietsaufteilung unterteilt werden. Danach wird der rekursive Aufruf auf dem Separator getätigt. Dessen dehierarchisierte Werte in M_1 – im folgenden als „Mitte" bezeichnet – dienen als Rand für die nachfolgenden Aufrufe auf den Hälften des zu behandelnden Gebiets. Da diese dehierarchisierten Werte für jede Richtung (Dimension) aufgehoben werden müssen, werden neben der aktuellen Mitte M auch die anderen Richtungen in der Liste Mlist mitgezogen. Wie schon im Up-Teil tauchen auch hier Funktionen zur rekursiven Zerlegung bzw. Zusammensetzung von Kabelbäumen auf.

Die Rekursion endet wieder mit $d = 0$. Die Einflüsse der $2D$ hierarchischen Nachbarn liegen dann als rekursive Liste P vor. Mit ihnen werden der aktuelle Punkt g modifiziert und die Einflüsse auf die hierarchischen Söhne berechnet. Diese erscheinen dann in M.

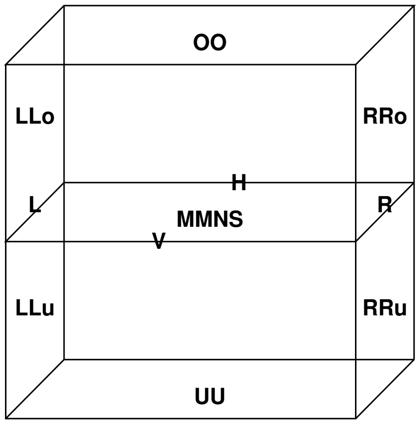


Abbildung 5.6: *Rekursive Gebietszerlegung eines dreidimensionalen Gebiets im Down-Algorithmus.*

```
function    down0(D,d,g,h,P): g__,m,M;
begin
    if isNil(P) then
        g__ := g;
        m,M := nil,nil;
    else
        <l,PRest,r> := P;
        g_,m := down_calc(D,d,g,h,l,r);     // g -> gd, g1 -> g1d, ... werfen
        g__,n,N := down0(D,incr(d),g_,h,PRest);
        M := <n,N>;
    fi;
end;
```

Analog zu `up_calc` geschieht dies nacheinander in D Schritten. Wie in Abschnitt 3.3 dargestellt, werden im d-ten Schritt alle $g[b_D, \ldots, b_{d+1}, 1, b_{d-1}, \ldots, b_1]$ verändert.

```
void down_calc_c(int D,int d,double* g,int glen,double* h,int hlen,
    double* l,int llen,double* r,int rlen,
    /* OUT */ double **g_,int* g_len,double **m,int* mlen)
{
    int i,x,y,low,high;

    *mlen = pow2(D-1);
    *m = (double*) malloc(sizeof(double)*(*mlen));

    for (low=0; low < pow2(d); low++) {
        for (high=0; high < pow2(D-d-1); high++) {
            i= high*pow2(d) + low;
            x= high*pow2(d+1) + low;
            y= x + pow2(d);
            g[y]  +=  0.5*(l[i]+r[i])*h[d];
```

```
            (*m)[i]  = g[x] + 0.5*(l[i]+r[i]); /* dehier. g[x] */
        }
    }
    *g_len = glen;
    *g_ = g;
}
```

Auch hier wird, wie schon zuvor bei den Berechnungsfunktionen für den Up-Teil, am Platz gearbeitet und der Ausgabevektor `g_` durch Überschreiben des Eingabevektors `g` gebildet.

5.3.3 Optimierung der Kommunikation

Wenn wir uns die funktionalen Programme aus diesem Abschnitt für den Down-Teil noch einmal genauer ansehen, stellen wir fest, daß die Kabelbaumsemantik eine zu große Verflechtung der Kommunikationsbeziehungen zwischen den Gitterpunkten bewirkt. Eigentlich sollten ja die Daten nur über hierarchische Väter zu den Söhnen fließen. Tatsächlich werden aber die Kabelbäume stets direkt vom Erzeuger zum Verbraucher gezogen, was dazu führt, daß für jeden Gitterpunkt Ströme von all seinen hierarchischen Nachbarn gezogen werden (vgl. die Diskussion am Ende von Kapitel 2). Hier bringen die direkte Verbindung und die damit verbundene sofortige Verfügbarkeit der Daten auch keinen Zeitgewinn, weil die Söhne ohnehin noch auf die Daten ihres Vaters warten müssen.

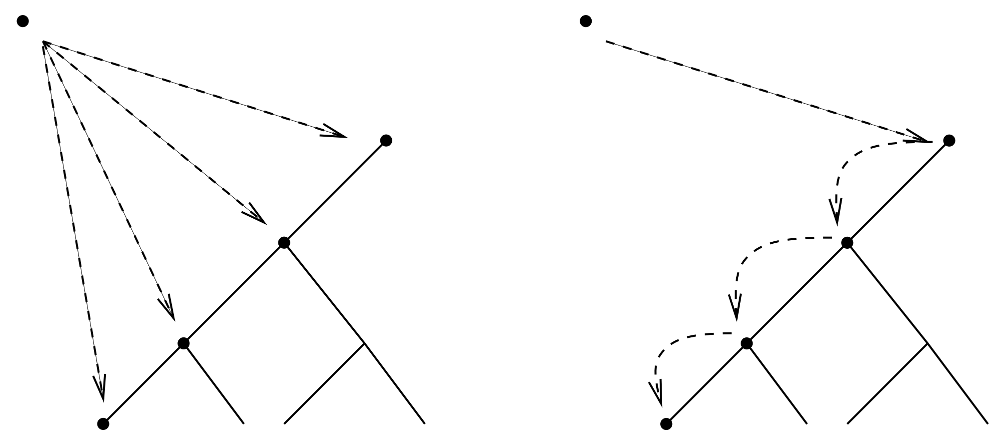


Abbildung 5.7: *Direktes Ziehen der Kabelbäume führt zu starker Verflechtung der Kommunikationsbeziehungen von den Rändern eines Gebiets zu den inneren Knoten (links). Durch Ziehen der Kabel über die Zwischenknoten wird die Kommunikation vereinfacht (rechts).*

Dieses unerwünschte Phänomen kann aber leicht beseitigt werden. Man muß nur dafür sorgen, daß die Kabel von den jeweiligen Zwischenknoten absorbiert und auf einen neuen Strom kopiert werden, der dann weiter zu den Söhnen geht. Da der neue Strom jetzt vom Vater *erzeugt* worden ist, gehen die Daten den gewünschten Weg. Abbildung 5.7 soll dieses Vorgehen veranschaulichen.

Um dies zu bewerkstelligen, muß die D-dimensionale Down-Funktion folgendermaßen geändert werden:

```
function    down(D,d,G,h,Done,Todo): G_,M,Mlist,Done_;
begin
    <Gl,g,Gr>:= G;
    Dl,done,Dr := decomp_rek_lMr(Done);
    <NowTodoL,YetTodo,NowTodoR> := Todo;
    YTl,yettodo,YTr := decomp_rek_lMr(YetTodo);
    d_1 := decr(d);
    if isNull(d_1) then
        g_,M_1,Mlist_1 := down0(D,d_1,g,h,<NowTodoL,done,NowTodoR>);
        NowTodo_L := NowTodoL;
        NowTodo_R := NowTodoR;
        done_     := done;
    else
        g_,M_1,Mlist_1,DONE := down(DIM,d_1,g,h,<NowTodoL,done,NowTodoR>,yettodo);
        <NowTodo_L,done_,NowTodo_R> := DONE;
    fi;
    if isNil(Gl) then
        G_l,Ml,Mlistl,D_l := nil,nil,nil,nil;
        G_r,Mr,Mlistr,D_r := nil,nil,nil,nil;
    else
        h2 := half(h,d);
        G_l,Ml,Mlistl,D_l :=
          down(DIM,d,Gl,h2,Dl,<NowTodo_L,YTl,M_1>) on new_proc_l;
        G_r,Mr,Mlistr,D_r :=          down(DIM,d,Gr,h2,Dr,<M_1,YTr,NowTodo_R>)
 on new_proc_r;
    fi;
    if isNil(Mlist_1) then
        m,mlist := nil,nil;
    else
        <m,mlist> := Mlist_1;
    fi;
    M := <Ml,m,Mr>;
    Mlist := comp_rek_mM(Mlistl,mlist,Mlistr);
    Done_ := comp_rek_lMr(D_l,done_,D_r);
    G_ := <G_l,g_,G_r>;
end;
```

Hier kommt im wesentlichen der neue Ausgabestrom `Done_` hinzu, der den Eingabestrom `Done` kopiert. Wir wir in Kapitel 7 sehen werden, werden durch die Ersetzung der alten durch die neue `down`-Funktion zwar die Programme auf *einem Rechner* geringfügig langsamer, da ein erhöhter Verwaltungsaufwand durch die Zusatzströme entsteht. Die parallelen Laufzeiten werden allerdings deutlich durch die erzielte Entflechtung der Kommunikationsbeziehungen gesenkt. Man beachte, daß das oben geschilderte Problem nicht im Up-Zyklus auftritt, weil dort die Daten ohnehin stets von den direkten Söhnen neu erzeugt werden.

In diesem Kapitel wurde anhand eines komplexen numerischen Algorithmus' gezeigt, daß FASAN in der Lage ist, maximale Parallelität und übersichtliche kurze Beschreibung miteinander zu verbinden. Die beiden wesentlichen Eigenschaften von FASAN sind in diesem Zusammenhang die Datenflußsemantik und das Kabelbaumprinzip. Trotz relativ einfacher sprachlicher Konstrukte und fehlender Higher-Order-Semantik ist FASAN expressiv genug, um Dünngitterprogramme, in denen die Dimensionalität als Parameter übergeben wird und Ein- und Ausgabelisten variabler Länge verlangt werden, elegant zu beschreiben.

6 Lastverteilung

Ein wichtiger Punkt der Parallelisierung wurde bisher nicht berührt – die Frage der möglichst ausgewogenen Verteilung des Berechnungsaufwandes auf die zur Verfügung stehenden Prozessoren. Dies wollen wir jetzt nachholen. Dazu beschäftigen wir uns zunächst mit der systematischen Numerierung der Punkte eines dünnen Gitters. Ausgehend von dieser Kodierung lassen sich in übersichtlicher Weise verschiedene Lastverteilungsstrategien beschreiben und bewerten. Dabei werden wir feststellen, daß die komplexe Abhängigkeitsstruktur der Gitterpunkte bei einfachen Verteilungsschemata zu einem beträchtlichen Kommunikationsaufwand führt. Dies ist besonders brisant bei busbasierten Workstationclustern. Das gemeinsam genutzte Transfermedium ist dann schnell überlastet und stellt einen Flaschenhals dar.

Aus diesem Grunde werden verschiedene Lastverteilungsstrategien auf zum Teil neuartigen skalierbaren Netztopologien betrachtet. Für zweidimensionale Dünngitteralgorithmen wird eine Dualisierung des Hypercubes vorgeschlagen, in der sich der Kommunikationsgraph mit konstanter Dilation 2 einbetten läßt, d.h. zwischen je zwei kommunizierenden Knoten liegt höchstens ein weiterer Rechenknoten. Am Ende des Kapitels stehen Überlegungen zur Lastverteilung adaptiver dünner Gitter.

6.1 Kodierung dünner Gitter

Lastverteilung bedeutet, weitgehend unabhängige Teilprobleme eines Algorithmus' auf die zur Verfügung stehenden Prozessoren zu verteilen. In unserem Modell bestehen die Teilprobleme aus Berechnungen an jeweils einem Gitterpunkt. Wir suchen also eine Lastverteilungsfunktion χ, die die Gitterpunkte auf die Menge der Prozessoren abbildet. Als ersten Schritt wollen wir in diesem Abschnitt untersuchen, wie die Punkte eines dünnen Gitters möglichst systematisch und kompakt beschrieben werden können. Damit lassen sich dann geeignete Funktionen χ leichter finden und darstellen.

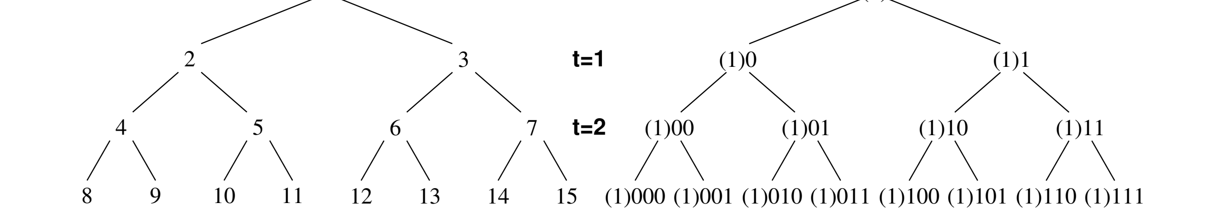


Abbildung 6.1: *Numerierung der binärbaumartig angeordneten Punkte eines eindimensionalen dünnen Gitters. In der Binärdarstellung kann die führende 1 in einer kompakten Kodierung entfallen.*

Sehen wir uns zunächst wieder ein eindimensionales Gitter an. Dieses sei gemäß der hierarchischen Ordnung binärbaumartig angeordnet. Eine natürliche Numerierung der Punkte ist in Abb. 6.1 links angegeben. Daneben sind die Zahlen im Binärsystem dargestellt. Die führende 1 können wir in einer kompakteren Darstellung weglassen. Dann wird ein Gitterpunkt p der Tiefe t eindeutig durch eine Bitsequenz $B(p)$ der Länge t beschrieben. Sind die Bitsequenzen innerhalb einer festen Tiefe t fortlaufend nach der x-Koordinate numeriert, so gilt:

$$B(p) := \frac{1}{2} \cdot (2^{t+1} \cdot x(p) - 1) \tag{6.1}$$
$$x(p) = (2B(p) + 1) \cdot 2^{-t-1} \tag{6.2}$$
$$h(p) = 2^{-t-1} \tag{6.3}$$

Im D-dimensionalen Fall ist ein Gitterpunkt durch seine Koordinaten $x_1, \ldots, x_D$ festgelegt. Jedem x_i kann gemäß (6.1) ein $B_i(p)$ zugeordnet werden. Somit ist p eindeutig durch das Tupel $B(p) := (B_1(p), \ldots, B_D(p))$ bestimmt. In Abbildung 6.2 sind für $D = 2$ die Teilräume $\mathcal{T}_{23}$ und $\mathcal{T}_{14}$ mit den entsprechenden Numerierungen dargestellt. $B_1(p)$ und $B_2(p)$ sind durch ein „x" getrennt. Diese Trennung ist wichtig, da das bloße Aneinanderhängen der Bitsequenzen zu $B_1(p) \ldots B_D(p)$ mehrdeutig ist. Die Sequenz legt nämlich nur die Basisfunktion innerhalb eines Teilraumes fest, nicht jedoch den Teilraum selbst.

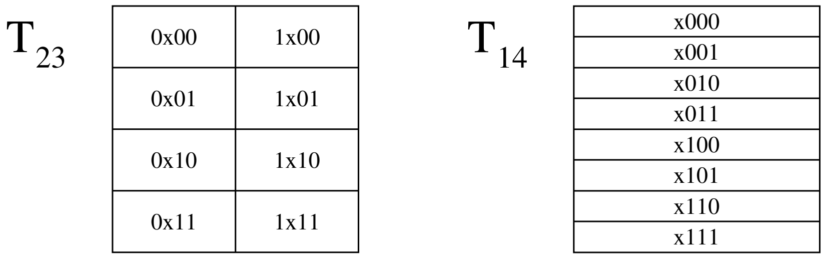


Abbildung 6.2: *Adressierung der Basisfunktionen in $\mathcal{T}_{23}$ und $\mathcal{T}_{14}$* $(D = 2, T = 3)$.

In der obigen Darstellung entspricht ein Punkt $p \in T_{i_1,\ldots,i_D}$ der Tiefe t einer Adresse $A = a_1 \ldots a_{t+D-1} \in \{0, 1, x\}^{t+D-1}$ mit

$$a_j = x \iff j \in \{i_1, i_1 + i_2, \ldots, i_1 + \ldots + i_{D-1}\}. \tag{6.4}$$

Wie man sich leicht überlegt, werden verschiedene Gitterpunkte auf verschiedene Adressen A abgebildet. Umgekehrt läßt sich jeder Adresse $A \in \{0, 1, x\}^*$ eindeutig ein Gitterpunkt p eines dünnen Gitters zuordnen. Dazu setzen wir zunächst:

$$D := |A|_x + 1 \tag{6.5}$$
$$t := |A| - D + 1, \tag{6.6}$$

wobei $|A|$ die Länge der Bitfolge A und $|A|_x$ die Zahl der x in A angibt. Wegen (6.5) besitzt A die folgende Form:

$$A = B_1 x B_2 x \ldots x B_D, \quad \text{mit} \quad B_i \in \{0, 1\}^*. \tag{6.7}$$

Diese Darstellung ist eindeutig. Seien $s_1, \ldots, s_{D-1}$ die aufsteigend geordneten Positionen der x in A. Mit $s_0 := 0$ und $s_D := t + D$ definieren wir

$$i_j := s_j - s_{j-1}, \quad \text{für} \quad j \in \{1, \ldots, D\}.$$

Der Punkt p liegt dann im Teilraum $T_{i_1,\ldots,i_D}$, und es gilt

$$\begin{aligned} x_j &:= (2B_j+1)\cdot 2^{-i_j} & (6.8)\\ h_j &:= 2^{-i_j}. & (6.9) \end{aligned}$$

Zur Vereinfachung der Schreibweise führen wir noch zwei Bezeichnungen ein:

$$\begin{aligned} \mathbb{Z}_n &:= \{0,\ldots,n-1\} \\ \binom{M}{n} &:= \Big\{\{m_1,\ldots,m_n\}\subset M : |\{m_1,\ldots,m_n\}| = n\Big\} \end{aligned}$$

Dabei sei n eine natürliche Zahl und M eine beliebige Menge. $\binom{M}{n}$ ist die Menge der *n-Teilmengen von M*.

Wir haben oben eine kompakte und anschauliche Adressierung für die Punkte eines dünnen Gitters entworfen. Im folgenden bilden wir nun diese Adressen auf natürliche Zahlen ab. Wir beschränken uns dazu vorerst auf die Punkte einer festen Tiefe $t \leq T$.

Zunächst stellen wir fest, daß die Menge der Adressen für Punkte der Tiefe t formal als Kreuzprodukt der Menge $\binom{\mathbb{Z}_{D+t-1}}{D-1}$ (für die möglichen Positionen $s_1,\ldots,s_{D-1}$) und der Menge $\mathbb{Z}_2^t$ (für die möglichen Bitfolgen $B_1(p)\ldots B_D(p)$) geschrieben werden kann. Wir erinnern daran, daß der erste Teil den Teilraum angibt, wohingegen der zweite Teil die Basisfunktion innerhalb des Teilraums festlegt. Wir suchen nun nach einer bijektiven Abbildung

$$\phi_{t,D} : \binom{\mathbb{Z}_{D+t-1}}{D-1} \times \mathbb{Z}_2^t \to \{0,\ldots,\binom{D+t-1}{D-1}\cdot 2^t - 1\} \tag{6.10}$$

Die Menge $\mathbb{Z}_2^t$ kann in natürlicher Weise auf $\mathbb{Z}_{2^t}$ abgebildet werden – die Bitfolge der Länge t wird einfach als Binärzahl B interpretiert. Bezeichne nun $\psi_{t,D}$ die Position des Tupels $(s_1,\ldots,s_{D-1})$ in einer lexikographischen Aufzählung von $\binom{\mathbb{Z}_{D+t-1}}{D-1}$, so haben wir mit

$$\phi_{t,D}(s_1,\ldots,s_{D-1}|B) := \psi_{t,D}(s_1,\ldots,s_{D-1})\cdot 2^t + B \tag{6.11}$$

unsere gewünschte Bijektion gefunden. Für die Funktion ψ läßt sich folgende Rekursionsbeziehung aufstellen:

$$\psi_{t,D}(i_1,\ldots,i_{D-1}) = \binom{D+t-1}{D-1} - \binom{D+t-i_1}{D-1} + \psi_{t-i_1,D-1}(i_2,\ldots,i_D) \tag{6.12}$$

Der erste Binomialkoeffizient gibt die Zahl aller möglichen Kombinationen an, die $D-1$ Zeichen „x“ auf $D+t-1$ Stellen zu verteilen. Vermindert man diesen Wert um 1, so erhält man die Nummer der lexikographisch letzten Kombination, nämlich $00\ldots0x\ldots xx$. Mit dem zweiten Binomialkoeffizienten ziehen wir davon diejenigen Fälle ab, bei denen alle „x“ hinter der (i_1-1)-ten Stelle verteilt sind. Wendet man dieses Verfahren rekursiv auf die restlichen Positionen an, kommen wir auf die Nummer unserer gewünschten Kombination. Die Rekursion in (6.12) läßt sich mit der bekannten Beziehung

$$\binom{n}{k+1} + \binom{n}{k} = \binom{n+1}{k+1}$$

auflösen. Es ergibt sich die folgende explizite Darstellung der Funktion ψ:

$$\psi_{t,D}(s_1, \ldots, s_{D-1}) = \binom{D+t-1}{D-1} - \sum_{k=1}^{D-1} \binom{D+t-1-s_k}{D-k} - 1 \tag{6.13}$$

Wir lassen nun die Beschränkung einer festen Tiefe t fallen und kodieren jetzt beliebige Gitterpunkte. Um alle Punkte des dünnen Gitters $\tilde{\mathcal{G}}_T^{(D)}$ bijektiv auf die Menge $\{0, \ldots, -1 + \sum_{t=0}^{T} 2^t \binom{D+t-1}{t}\}$ abzubilden, müssen wir lediglich zu $\phi_{t,D}(s_1, \ldots, s_{D-1}|B)$ den Offset $\sum_{k=0}^{t-1} 2^k \binom{D+k-1}{k}$ addieren:

$$\begin{aligned} \phi_D \quad &: \quad \tilde{\mathcal{G}}_T^{(D)} \to \{0, \ldots, -1 + \sum_{t=0}^{T} 2^t \binom{D+t-1}{t}\}, \\ \phi_D(s_1, \ldots, s_{D-1}|B) \quad &:= \quad \sum_{k=0}^{t-1} 2^k \binom{D+k-1}{k} + \phi_{t,D}(s_1, \ldots, s_{D-1}|B) \quad \forall t \le T \end{aligned}$$

Abschließend wollen wir kurz diskutieren, wie sich die Adresse eines Punktes $p \in T_{i_1,\ldots,i_D}$ beim Übergang auf die hierarchischen Söhne ändert. Seien also p_1 und p_2 die hierarchischen Söhne von p in Richtung d, liegen also in $T_{i_1,\ldots,i_{d-1},i_d+1,i_{d+1},\ldots,i_D}$. Sei weiterhin p durch $(s_1, \ldots, s_{D-1}|B_1 \ldots B_D)$ beschrieben, so gilt für $p_{1,2} = (s_1^{1,2}, \ldots, s_{D-1}^{1,2}|B_1^{1,2} \ldots B_D^{1,2})$:

$$\begin{aligned} j \neq d \quad &: \quad B_j^{1,2} = B_j \\ j = d \quad &: \quad B_j^1 = 2 \cdot B_j, \quad B_j^2 = 2 \cdot B_j + 1 \\ j < d \quad &: \quad s_j^{1,2} = s_j \\ j \ge d \quad &: \quad s_j^{1,2} = s_j + 1 \end{aligned}$$

So sind z.B. die hierarchischen Söhne von $p = 01x11xx$ in Richtung 2 gleich $p_1 = 01x110xx$ und $p_2 = 01x111xx$ und in Richtung 3 gleich $p_1 = 01x11x0x$ und $p_2 = 01x11x1x$.

6.2 Verteilung auf 2^r Prozessoren

Nach dieser kurzen Einführung in die Kodierung dünner Gitter kommen wir jetzt auf unser ursprüngliches Problem zurück, nämlich der Verteilung der Dünngitterpunkte auf die vorhandenen Rechenknoten. Die Zahl dieser Knoten setzen wir zunächst als Zweierpotenz voraus. Denn erstens gibt es zahlreiche Parallelrechnerarchitekturen, in denen die

Knotenzahl eine Zweierpotenz ist. Zweitens lassen sich beliebige andere Anzahlen für die Menge der Knoten dadurch berücksichtigen, daß wir die nächstkleinere Zweierpotenz wählen und die Rechner mit höheren Nummern leer stehen lassen. So verschenken wir höchstens einen konstanten Faktor 2. Die 2^r Prozessoren kodieren wir standardgemäß mit Bitfolgen der Länge r.

Die Vernetzungstopologie der Knoten untereinander sei an dieser Stelle noch nicht näher spezifiziert. Allerdings ist aus der Praxis bekannt, daß Kommunikation zwischen den Rechnern deutlich teurer ist als der Zugriff auf Daten im lokalen Speicher. Deshalb kommt es uns neben der Ausgewogenheit der Verteilung auch darauf an, voneinander abhängende Punkte möglichst lokal auf einen Rechner zu legen. Dadurch können die Rechner zudem weitgehend unabhängig voneinander arbeiten, weil sie in geringerem Ausmaß auf die Ergebnisse anderer Rechner warten müssen.

Wir haben oben diverse Darstellungen für einen Dünngitterpunkt p gesehen. Eine davon war das Tupel $(s_1, \ldots, s_{D-1}|B) \in \binom{\mathbb{Z}_{D+t-1}}{D-1} \times \mathbb{Z}_2^t$, wobei der erste Teil den zu p gehörenden Teilraum beschreibt, wohingegen der zweite Teil die Basisfunktion selbst innerhalb dieses Teilraums festlegt. Da wir die Zahl der Prozessoren als eine Zweierpotenz angenommen haben, liegt es nahe, zur Abbildung von p auf einen Rechenknoten den zweiten Teil des Tupels heranzuziehen, der ja wie die Prozessoren durch eine Folge von Bits beschrieben ist. Für $t \geq r$ könnte man z.B. einfach die ersten r Bits von B nehmen und sie als Adresse eines Prozessors interpretieren. Das heißt, ein Punkt mit $B = b_1 \ldots b_t$ landet auf Prozessor $b_1 \ldots b_r$. Ist die Tiefe t eines Punktes kleiner als r, so legen wir den Punkt auf den Rechner $b_1 \ldots b_t 0 \ldots 0$. Die so definierte Zuordnung bezeichnen wir fortan mit χ.

Lemma 6.1 *Die Punkte des Gitters $\tilde{\mathcal{G}}_T^{(D)}$ seien mit Hilfe der soeben eingeführten Lastverteilungsfunktion χ auf 2^r Prozessoren verteilt. Dann gilt:*

1. *Die Punkte der Tiefen $t \geq r$ werden gleichmäßig auf alle Rechner verteilt. Die Zahl der Punkte der Tiefe t pro Rechner beträgt*
$$2^{t-r} \cdot \binom{D+t-1}{D-1}.$$

2. *Die Punkte der Tiefen $t < r$ werden gleichmäßig auf 2^t Rechner verteilt. Die Zahl der Punkte der Tiefe t pro Rechner beträgt*
$$\binom{D+t-1}{D-1}.$$

3. *Die maximale Zahl der Punkte auf einem Rechner beträgt*
$$\binom{D+r-1}{D} + \sum_{t=r}^{T} 2^{t-r} \binom{D+t-1}{D-1} = O(r^D + 2^{T-r} T^{D-1}). \tag{6.14}$$

Beweis:

1. Bei einer festen Tiefe $t \geq r$ gibt es zu einer fester Rechneradresse $b_1 \dots b_r$ genau 2^{t-r} mögliche Bitsequenzen, die $b_1 \dots b_r$ als Präfix haben. Jede dieser Bitsequenzen gibt jeweils eine Basisfunktion aus jedem der $\binom{D+t-1}{D-1}$ Teilräume der Tiefe t an. Deswegen besitzt jeder Rechner genau $2^{t-r}\binom{D+t-1}{D-1}$ Punkte der Tiefe t.

2. Die Bitfolge $b_1 \dots b_t$ bestimmt wieder genau eine Basisfunktion aus jedem der $\binom{D+t-1}{D-1}$ Teilräume der Tiefe t. Da diese Bitfolge eindeutig dem Rechner $b_1 \dots b_t 0^{r-t}$ zugeordnet ist, folgt die Behauptung.

3. Nach Konstruktion von χ und dem Beweis zu 2. ist klar, daß Rechner 0 die meisten Punkte zugewiesen bekommt, da er von jeder Tiefe mindestens soviele Punkte erhält wie jeder andere Rechner.

   Nach 2. erhält Rechner 0 bei Tiefe $t < r$ genau $\binom{D+t-1}{D-1}$ Punkte, also insgesamt

   $$\sum_{t=0}^{r-1} \binom{D+t-1}{D-1} = \binom{D+r-1}{D} = O(r^D)$$

   Die ihm zugeordeneten Punkte der Tiefen $t \geq r$ summieren sich zu:

   $$\sum_{t=r}^{T} 2^{t-r} \binom{D+t-1}{D-1} = 2^{-r} \cdot \sum_{t=r}^{T} 2^t \binom{D+t-1}{D-1} = O(2^{T-r} T^{D-1}) \qquad (6.15)$$

   Damit ist die Behauptung bewiesen. ◊

Bemerkung 6.2 *Die in Lemma 6.1 gemachten Aussagen bleiben, wie die Beweise zeigen, auch noch für andere Kodierungen gültig, die durch Permutation der Bits in $B = B_1 \dots B_D$ entstehen. Die Permutationen können sich für verschiedene Tiefen sogar unterscheiden. Wichtig ist aber, daß die Permutation π innerhalb einer bestimmten Tiefe t fest ist, damit für alle Punkte der Tiefe die einzelnen Bitfolgen $B_1, \dots, B_D$ eindeutig aus der Kodierung bestimmt werden können.*

Da die Zahl der Punkte auf einem Rechner gleichzeitig ein Maß für die Zeitkomplexität des parallelen Algorithmus' ist, liefert Lemma 6.1 auch einen Beweis dafür, daß für kleine r nahezu optimale Beschleunigung der Laufzeit möglich ist. Selbst für $r = T$ gilt noch:

Korollar 6.3 *Werden 2^T Rechner eingesetzt, so liegen maximal $O(T^D)$ Punkte auf einem Rechner.*

Beweis: folgt sofort aus (6.14). ◊

Bei einer wie auch immer gearteten optimal ausgewogenen Verteilung lägen dagegen $O(\tilde{N}_T^{(D)}/2^T) = O(T^{D-1})$ Punkte auf jedem Rechner, wobei $\tilde{N}_T^{(D)}$ wieder die Anzahl aller

Punkte des dünnen Gitters bezeichnet. Die oben eingehandelte Verschlechterung beträgt also nur einen logarithmischen Faktor T.

Die Ausgewogenheit der Verteilung ist nur *ein* wichtiges Argument bei der Untersuchung einer Lastverteilungsstrategie. Ein zweites ist die Lokalität von abhängigen Teilen auf einem Rechner. Wie sieht es also hier mit der Lokalität der voneinander abhängigen Punkte aus?

Die gewählte Verteilungsfunktion χ hat den Vorteil, daß sich zumindest einer der $2D$ hierarchischen Söhne auf demselben Rechner wie der Vater befindet. Für $D = 1$ bewirkt die gewählte Strategie sogar, daß ab Tiefe $t = r$ komplette Unterbäume auf einem Rechner sind (s. Abbildung 6.3), so daß in diesem Bereich völlig unabhängig gerechnet werden kann. Lediglich bei den Knoten niedrigerer Tiefe ist Kommunikation nötig. Ist T aber deutlich größer als r, so fällt dies kaum ins Gewicht.

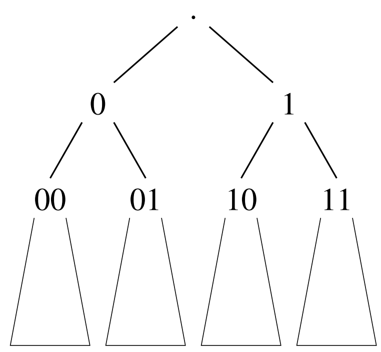


Abbildung 6.3: *Aufteilung eines eindimensionalen dünnen Gitters auf 4 Prozessoren.*

Bei $D > 1$ wird die Situation durch die Auszeichnung der niedrigen Dimensionen schwieriger. Um ein Gefühl für die resultierende Verteilung zu bekommen, ist in Abbildung 6.4 schematisch dargestellt, welche Bereiche eines zweidimensionalen dünnen Gitters auf welche der 2^2 bzw. 2^3 Prozessoren zu liegen kommen. (Ähnliche Verteilungsmuster für zweidimensionale Gitter wurden bereits in [Hah90] vorgestellt.) Hier sieht man schon plastisch, daß die gewählte Lastverteilungsstrategie die y-Richtung gegenüber der x-Richtung bevorzugt. Das Gebiet wird im wesentlichen durch horizontale Streifen geteilt, so daß häufig die hierarchischen Söhne in y-Richtung auf demselben Rechner wie der Vater liegen. In x-Richtung sind die Söhne dagegen stärker verstreut.

Allgemein gilt im D-dimensionalen Fall, daß die Söhne in Richtung $l \in \{1, \ldots, D\}$ bei steigendem l mit geringerer Wahrscheinlichkeit auf anderen Knoten liegen als der Vater. Dies ist leicht einzusehen, da ja nur die ersten r Positionen von B zur Differenzierung verwendet werden. Für $l = D$ gilt:

Lemma 6.4 *Bei obigem Lastverteilungsschema liegen die beiden hierarchischen Söhne eines Punktes p in Richtung D genau dann auf demselben Rechner wie p, wenn die Tiefe von p größer oder gleich r ist.*

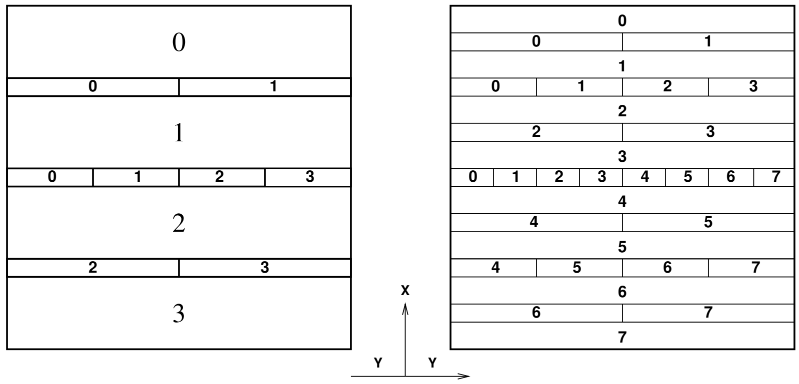


Abbildung 6.4: *Aufteilung eines zweidim. dünnen Gitters auf 4 bzw. 8 Prozessoren.*

Beweis:
Sei p die Bitfolge, die durch Konkatenation der Bitfolgen $B_1, \ldots, B_D$ entsteht, zugeordnet. Den Söhnen p_1 und p_2 entsprechen dann die Bitfolgen $B_1 \ldots B_D 0$ und $B_1 \ldots B_D 1$.

Die Funktion prefix(r, B) gebe die ersten r Bits in B als Zeichenfolge zurück. Dann folgt aus

$$\text{prefix}(r, B_1 \ldots B_D 00 \ldots 0) = \text{prefix}(r, B_1 \ldots B_D 10 \ldots 0) \quad \Longleftrightarrow \quad |B_1 \ldots B_D| \geq r$$

die Behauptung. $\diamondsuit$

Mit den gleichen Überlegungen kommt man zu der allgemeineren Aussage:

Lemma 6.5 *Die beiden Söhne eines Punktes p in Richtung l liegen genau dann auf demselben Rechner wie p, wenn gilt:*

$$\sum_{i=1}^{l} t_i \geq r \tag{6.16}$$

Dabei seien t_i die nach (2.8) definierten Tiefen der Punktes p in Richtung i.

Beweis: folgt sofort aus $|B_i| = t_i$ nach dem Beweisschema aus Lemma 6.4. $\diamondsuit$

6.3 Der unidirektionale Algorithmus auf De-Bruijn-Netzen

Wir kommen jetzt auf die Frage der Kommunikation zurück und untersuchen in diesem Abschnitt zunächst den unidirektionalen Algorithmus aus Abschnitt 3.3. Verteilen wir die Punkte des dünnen Gitters nach dem im vorigen Kapitel angegebenen Schema, so zeigt Lemma 6.5, daß die Verteilung gerade für die Richtung D am besten geeignet ist. Hier liegen nach Lemma 6.4 alle Söhne eines Knotens ab Tiefe r auf demselben Rechner. Das heißt, alle Berechnungen unterhalb Tiefe r können für die Richtung D lokal auf einem Rechner geschehen.

Bei Up- bzw. Down-Durchläufen in anderen Richtungen l wird nach Lemma 6.5 die Punkteverteilung immer ungünstiger, je niedriger l ist. Deshalb ist es eine naheliegende Idee, die Punkte vor jedem Richtungswechsel neu zu verteilen, so daß sie für die nächste Richtung optimal liegen. Bei der neuen Verteilung benutzen wir wieder die alte Strategie – die ersten r Bits der Binäradresse von p geben die Rechnernummer an. Allerdings sortieren wir die Bitmuster $B_1, \ldots, B_D$ in B so, daß nun B_l *als letztes* in B steht. Da bei dem unidirektionalen Algorithmus nur Richtungswechsel von l nach $l+1$ oder $l-1$ vorkommen, muß bei diesem Umsortieren entweder die führende Bitsequenz an die letzte Stelle oder die letzte Bitsequenz an die erste Stelle geschoben werden.

Wir wollen im folgenden untersuchen, welche Kommunikationskosten bei diesem Algorithmus entstehen. Sie werden bestimmt durch

1. die Anzahl der Richtungswechsel $RW(D)$
2. die Kosten einer Neuverteilung $NV(D, T, r)$ und
3. die Kommunikationskosten $C(D, r)$ in den oberen Leveln $0, \ldots, r-1$.

Die beiden letzten Punkte hängen natürlich von den Eigenschaften des zugrundeliegenden Netzwerks ab. Da wir bei der Neuverteilung Bitadressen verschieben müssen, bieten sich sogenannte *De-Bruijn-Netze* an, die wir kurz definieren:

Definition 6.1 *Ein (ungerichteter) r-dimensionaler De-Bruijn-Graph besteht aus 2^r Knoten. Ein Knoten $u_1 \ldots u_r$ besitzt Kanten zu den Knoten $u_2 \ldots u_r 0$, $u_2 \ldots u_r 1$, $0 u_1 \ldots u_{r-1}$ und $1 u_1 \ldots u_{r-1}$ (und nur zu diesen).*

Man beachte, daß diese vier Knoten nicht notwendigerweise untereinander und von $u_1 \ldots u_r$ verschieden sein müssen. Abbildung 6.5 zeigt einen dreidimensionalen De-Bruijn-Graphen.

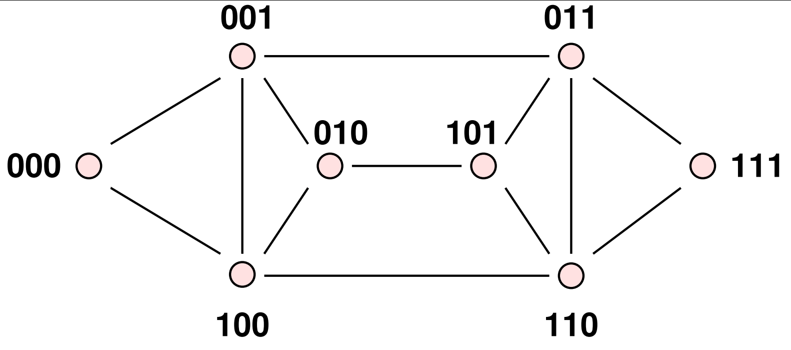


Abbildung 6.5: *Dreidimensionaler De-Bruijn-Graph.*

Aus der Definition folgt sofort

Lemma 6.6 *Jeder Knoten eines r-dimensionalen De-Bruijn-Graphen besitzt höchstens vier Kanten.*

und

Lemma 6.7 *Der Durchmesser des r-dimensionalen De-Bruijn-Graphen ist r, d.h. jeder Knoten kann in höchstens r Schritten jeden anderen Knoten erreichen.*

Sehen wir uns nun die Kosten einer Neuverteilung $NV(D, T, r)$ der Dünngitterpunkte in einem als De-Bruijn-Graph angelegten Rechnernetz (kurz: *De-Bruijn-Netz*) an. Eine obere Schranke liefert

Lemma 6.8 *In einem r-dimensionalen De-Bruijn-Netz gilt:*

$$NV(D, T, r) = O(r^{D+1} + 2^{T-r} \cdot T^D) = O(r^{D+1} + T \cdot \tilde{N}_T^{(D)}/2^r). \tag{6.17}$$

Beweis:
Die Behauptung ist wegen Lemma 6.1 und der Bemerkung 6.2 bewiesen, wenn wir zeigen können, daß alle Punkte einer festen Tiefe in $O(r)$ für $t < r$ bzw. $O(T)$ für $t \geq r$ Kommunikationsschritten zu ihrem Zielrechner gelangen können und nach jedem Kommunikationsschritt höchstens $\binom{D+t-1}{t}$ Punkte für $t < r$ bzw. $2^{t-r}\binom{D+t-1}{t}$ Punkte für $t \geq r$ auf jedem Rechner liegen, die weiterverschickt werden müssen. Die letzte Bedingung über die gleichmäßige Verteilung der Punkte auf die Rechner auch während der Umverteilung ist wesentlich, da es bei ungeschickter Vorgehensweise passieren kann, daß in irgendeinem Schritt plötzlich alle Knoten auf einem Rechner liegen und weitergesandt werden müssen. Dann wäre die Komplexität aber $O(\tilde{N}_T^{(D)})$!

O.B.d.A seien die Bitsequenzen $B_1, \ldots, B_D$ eines Punktes p der Tiefe t so angeordnet, daß der Rechner durch prefix$(r, B_1 \ldots B_D 0^{T-r})$ bestimmt ist. Wie wir oben dargelegt haben, wird der Zielrechner von p entweder durch prefix$(r, B_D B_1 \ldots B_{D-1} 0^{T-r})$ oder prefix$(r, B_2 \ldots B_D B_1 0^{T-r})$ bestimmt. Die Argumentation ist für beide Fälle analog, weswegen wir uns im folgenden auf den ersten Fall beschränken.

Sie zunächst $t < r$ fest. Der Rechner des Punktes p ist demnach gleich $B_1 \ldots B_D 0^{r-t}$, der Zielrechner gleich $B_D B_1 \ldots B_{D-1} 0^{r-t}$. Außerdem sei $B_D =: b_1 \ldots b_{t_D} \in \mathbb{Z}_2^{t_D}$.

Nach Konstruktion des De-Bruijn-Graphen sind Rechner, die durch einen linken bzw. rechten Shift ihrer Bitadresse ineinander übergehen, direkt benachbart. Außerdem darf dabei das letzte bzw. erste Bit neu gesetzt werden. Um uns die Argumentation zu vereinfachen, sprechen wir im folgenden oft nur davon, daß Bitmuster in einer Adresse der Länge r verschoben und einzelne Bits gesetzt werden. Jeder solchen Veränderung der Adresse entspricht natürlich ein Kommunikationsschritt, bei dem alle durch $B_1, \ldots, B_D$ bestimmten Punkte von dem Rechner mit der alten Adresse auf den Rechner mit der neuen Rechner geschoben werden.

Wir teilen die Kommunikation in zwei Phasen ein. Die erste Phase besteht aus r Schritten. Im Schritt k wird die Rechneradresse um eine Position nach rechts geschoben. Ist $k < t_D$, so wird gleichzeitig vorn das Bit b_{t_D-k+1}, andernfalls eine 0 gesetzt.

$$\begin{array}{lcll} b_{t_D-k+2}\dots b_{t_D}B_1B_2\dots & \longrightarrow & b_{t_D-k+1}b_{t_D-k+2}\dots b_{t_D}B_1B_2\dots & \text{bzw.} \\ b_{t_D-k+2}\dots b_{t_D}B_1B_2\dots & \longrightarrow & 0b_{t_D-k+2}\dots b_{t_D}B_1B_2\dots & \end{array}$$

Je nach Länge von B_D kann es vorkommen, daß Bits aus B_D doppelt erscheinen (vor und hinter $B_1\dots B_{D-1}$). Dies stört nicht, da für unsere Argumentation nur wichtig ist, daß aus einer gegebenen Binärsequenz die Teilsequenzen $B_1,\dots,B_D$ eindeutig bestimmt werden können. Bei unserer Vorgehensweise ist dies aber nach jedem Schritt möglich.

Am Ende der ersten Phase liegen die Punkte p mit $B = B_1\dots B_D$ auf dem Rechner $0^{r-t}B_DB_1\dots B_{D-1}$. In der zweiten Phase werden die $r-t$ Nullbits durch zyklische Linksverschiebung wieder ans Ende der Adresse gebracht.

Da in beiden Phasen nach jedem Schritt alle Bits aus $B_1\dots B_D$ in der neuen Rechneradresse auftauchen, ist mit Lemma 6.1 und Bemerkung 6.2 klar, daß im nächsten Schritt nicht mehr als $\binom{D+t-1}{t}$ Knoten pro Rechner weiterversandt werden müssen. Da beide Phasen in $O(r)$ Schritten vollendet sind, erhalten wir insgesamt eine Komplexität von $O(r^{D+1})$ Zeitschritten für alle Punkte der Tiefe $t < r$.

Sei nun $t \geq r$ eine fest gewählte Tiefe. Wie im vorigen Fall schieben wir auch hier sukzessive die Bits b_{t_D} bis b_1 von B_D an die Front der Adresse, d.h. ein Punkt mit $B = B_1\dots B_D$ liegt nach dem k-ten Schritt auf dem Rechner prefix$(r, b_{t_D-k+1}\dots b_{t_D}B_1\dots B_D)$. Punkte mit $t_D < k$ bleiben danach auf demselben Rechner liegen und werden nicht weiter berücksichtigt.

Auch hier sind die Zahl der weiterzuverschickenden Punkte pro Rechner nach jedem Schritt höchstens von der Ordnung $O(2^{t-r}t^{D-1})$. Allerdings benötigen wir jetzt insgesamt $O(t)$ Schritte. Über alle Tiefen $t \geq r$ summiert, ergibt sich demnach:

$$\sum_{t=r}^{T} t\cdot 2^{t-r}t^{D-1} = O(2^{T-r}T^D)$$

Damit ist die Behauptung bewiesen. ♢

In diesem Teil des Algorithmus', der Neuverteilung der Punkte, verschlechtern wir uns für kleine r also gegenüber der möglichen parallelen Komplexität von $O(\tilde{N}_T^{(D)}/2^r)$ nur um einen logarithmischen Faktor T. Die zusätzlichen Kosten für den Datenaustausch nichtlokaler Punkte in den oberen Leveln sind auch relativ gering:

Lemma 6.9 *In einem r-dimensionalen De-Bruijn-Netz gilt:*

$$C(D,r) = O(r^{D+1}). \tag{6.18}$$

Beweis:
Wir schränken uns hier zur Vereinfachung der Darstellung auf die Betrachtung eines Down-Schrittes ein. Die Argumentation ist für den Up-Schritt im wesentlichen dieselbe.

Nach Lemma 6.4 entstehen unterhalb der Tiefe r keine Kosten durch externe Kommunikation, da alle Söhne dann lokal auf dem jeweiligen Rechner liegen. Beim Übergang von Tiefe $t < r$ nach $t + 1$ müssen alle Rechner mit Adressen $u_1 \dots u_t 00 \dots 0$ mit den Rechnern $u_1 \dots u_t 10 \dots 0$, die ja die Söhne ihrer Punkte in D-Richtung besitzen, kommunizieren.

Um das entsprechende Bit in der Rechneradresse zu setzen, müssen wir erst die $r - t$ Nullbits am Ende der ersten Adresse durch zyklische Rechtsverschiebungen eliminieren, dann die 1 setzen und anschließend durch $r - t - 1$ zyklische Linksverschiebungen die Nullen wieder hinten anhängen. Alternativ kann man auch die $u_1 \dots u_t$ durch zyklische Linksverschiebungen ans Ende der Adresse bringen, die 1 am Anfang setzen und die $u_1 \dots u_t$ wieder zurückschieben. Also sind insgesamt $2 \cdot min(t, r - t)$ Verschiebungen nötig, die jeweils einem Kommunikationsschritt in dem De-Bruijn-Netz entspricht. Da auf jedem Rechner $\binom{D+t-1}{D-1} = O(t^{D-1})$ Knoten der Tiefe t liegen, folgt mit

$$2 \cdot \sum_{t=0}^{r/2} t \cdot t^{D-1} \quad + \quad 2 \cdot \sum_{t=r/2}^{r-1} (r - t) \cdot t^{D-1} \quad = \quad O(r^{D+1})$$

die Behauptung. ◊

Somit fehlt nur noch eine Aussage über die Zahl der Richtungswechsel $RW(D)$:

Lemma 6.10 *Bei dem D-dimensionalen unidirektionalen Algorithmus sind*

$$RW(D) = 2^D - 2 = O(2^D) \tag{6.19}$$

Richtungswechsel nötig.

Beweis:
Für $D = 1$ gilt: $RW(1) = 0$. Aus Gleichung (3.19) in Abschnit 3.3.1 resultiert die Rekursionsbeziehung

$$RW(D) = 2 \cdot RW(D - 1) + 2,$$

woraus unmittelbar die Aussage folgt. ◊

Wir erhalten nun insgesamt die folgende Abschätzung

Satz 6.11 *Die Kommunikationskosten des oben angegebenen Algorithmus' in einem r-dimensionalen De-Bruijn-Netzwerks sind von der Ordnung* $O(2^D \cdot (r^{D+1} + 2^{T-r} \cdot T^D)) = O(2^D \cdot (r^{D+1} + T \cdot \tilde{N}_T^{(D)})/2^r)$.

Für kleine r ist der erste Summand zu vernachlässigen und die Kommunikation verschlechtert die parallele Komplexität von bestenfalls $O(2^D \cdot \tilde{N}_T^{(D)})/2^r)$ nur um einen logarithmischen Faktor T. Für $r \approx T$ gilt dagegen:

$$O(2^D \cdot r^{D+1} + 2^{T-r} \cdot T^D) = O(2^D \cdot T^{D+1}) = O(2^D \cdot T^2 \cdot \tilde{N}_T^{(D)}/2^T)$$

Jetzt kommt also noch der Faktor $T^2 \approx r^2$ hinzu, der aber wiederum nur logarithmische Verschlechterung bedeutet.

6.4 Zweidimensionale dünne Gitter auf Kantenhypercuben

In den vorigen beiden Abschnitten sind wir von einer gegebenen Anzahl von Rechnern bzw. einer festen Topologie ausgegangen. Jetzt wollen wir die Fragestellung umkehren und untersuchen, wie eine Rechnertopologie aussehen müßte, damit wir die optimale parallele Komplexität erreichen können.

Zunächst ist mit (2.16) klar, daß wir dazu $O(2^T \binom{D+T-1}{T})$ Rechner brauchen, um die Punkte der untersten Tiefe T parallel bearbeiten zu können. Gehen wir also der Einfachheit halber davon aus, wir hätten genau $\tilde{L}_T^D = 2^T \binom{D+T-1}{T}$ Knoten zur Verfügung. Dann können wir jedem Blatt des Dünngittergraphen genau einen Knoten zuordnen. Dies läßt sich z.B. mit der im ersten Abschnitt dieses Kapitels entworfenen Abbildung $\phi_{T,D}$ bewerkstelligen. Einem Knoten der Tiefe $t < T$ ordnen wir den Rechner zu, auf dem der erste hierarchische Sohn in Richtung 1 liegt. In unserer Adreßnotation wird demnach $p = B_1 x \dots x B_D$ mit dem Punkt $p' = B_1 x \dots x B_D 0 \dots 0$ der Tiefe T gleichgesetzt.

Das Hauptproblem liegt nun darin, eine Verbindungstopologie für die Rechner zu entwerfen, die folgenden Punkten gerecht wird:

1. regelmäßige Struktur
2. geringe Anzahl von I/O-Ports pro Rechenknoten
3. die nötigen Kommunikationen zwischen hierarchischen Vätern und Söhnen sollen weitgehend parallel und
4. über möglichst wenige Zwischenstationen verlaufen

Der ersten beiden Punkte sind für die Realisierbarkeit des Netzwerks wichtig. Die beiden folgenden Bedingungen sollen verhindern, daß die Kommunikation zum Flaschenhals wird.

Für $D = 1$ ist die zu einem Punkt gehörende Adresse A eine reine Bitsequenz ohne „x“. Zu einer festen Tiefe $t \leq T$ gehören genau 2^t Gitterpunkte, denen gemäß (6.1) eine eindeutige Bitsequenz $b_1 \dots b_t$ zugeordnet ist. Die beiden hierarchischen Söhne sind $b_1 \dots b_t 0$ und $b_1 \dots b_t 1$.

Eine naheliegende Architektur für 2^T Rechner ist der T-dimensionale Hypercube. Darin ist jeder Rechner $\mathcal{R}$ mit T anderen Rechnern verbunden, deren Binäradresse sich in genau einem Bit von Rechner $\mathcal{R}$ unterscheiden. Formal können wir einen N-dimensionalen binären Hypercubegraphen $H(N)$ durch die Menge seiner Knoten $V(N)$ und der Verbindungskanten $E(N)$ definieren:

$$H(N) := (V(N), E(N))$$

mit

$$\begin{aligned} V(N) &:= \mathbb{Z}_2^N \\ E(N) &:= \left\{ \{v, w\} \in \binom{V(N)}{2} : d(v, w) = 1 \right\} \end{aligned}$$

Dabei bezeichne $d(v, w)$ den *Hamming*-Abstand zwischen den Knoten v and w, d.h. die Zahl der unterschiedlichen Bits in $v = v_1 \dots v_T$ und $w = w_1 \dots w_T$. Wir verallgemeinern diese Metrik in kanonischer Weise auf mehr als zwei Knoten:

$$d(v^1, \dots, v^n) := \max_{i \neq j} d(v^i, v^j)$$

Kehren wir wieder zu unserem Ausgangsproblem zurück und analysieren die vorgeschlagene Lastverteilungsstrategie, nämlich die Gitterpunkte $b_1 \dots b_t$ auf Prozessor $b_1 \dots b_t 0^{T-t}$ zu legen:

Lemma 6.12 *Für die beschriebene Lastverteilung gilt:*

1. *Je zwei Gitterpunkte p und p' derselben Tiefe t liegen auf verschiedenen Rechnern.*
2. *Alle Kommunikationen zwischen zwei Leveln können direkt (ohne Zwischenknoten) und vollständig parallel erfolgen.*

Beweis: trivial. $\diamondsuit$

Für den eindimensionalen Fall haben wir also mit dem Hypercube unser Ziel erreicht. Die nötigen Kommunikationen beeinträchtigen die optimale Komplexität des parallelen Algorithmus' nicht. Versuchen wir nun unser Glück für den zweidimensionalen Fall.

Wenn wir uns die Zahl der Punkte eines zweidimensionalen dünnen Gitters der Tiefe T ansehen, so stellen wir fest, daß sie genau der Zahl der *Kanten* von $H(T + 1)$ entspricht. Deshalb identifizieren wir nun die Kanten mit den Rechnern einer neuen Topologie, d.h. wir stellen uns vor, daß auf jeder Kante von $H(T + 1)$ ein Rechner sitzt.

Bei dem „normalen“ Hypercube wurden zur Beschreibung der Verbindungstopologie zwischen den Ecken (Dimension 0) die Kanten (Dimension 1) benutzt. Es liegt nun nahe, bei unserem neuen „Kantenhypercube“ die Flächen (Dimension 2) zur Definition der Verbindungsstruktur heranzuziehen.

Da eine Fläche mit vier Kanten inzidiert, können wir nicht mehr mit direkten Links zwischen zwei Rechnern operieren. Vielmehr müssen wir uns auf jeder Fläche einen Kommunikationsbus vorstellen, auf den die vier Kanten (= Rechner) gemeinsamen Zugriff haben. In Abbildung 6.6 ist ein dreidimensionaler Kantenhypercube neben dem entsprechenden Hypercube dargestellt.

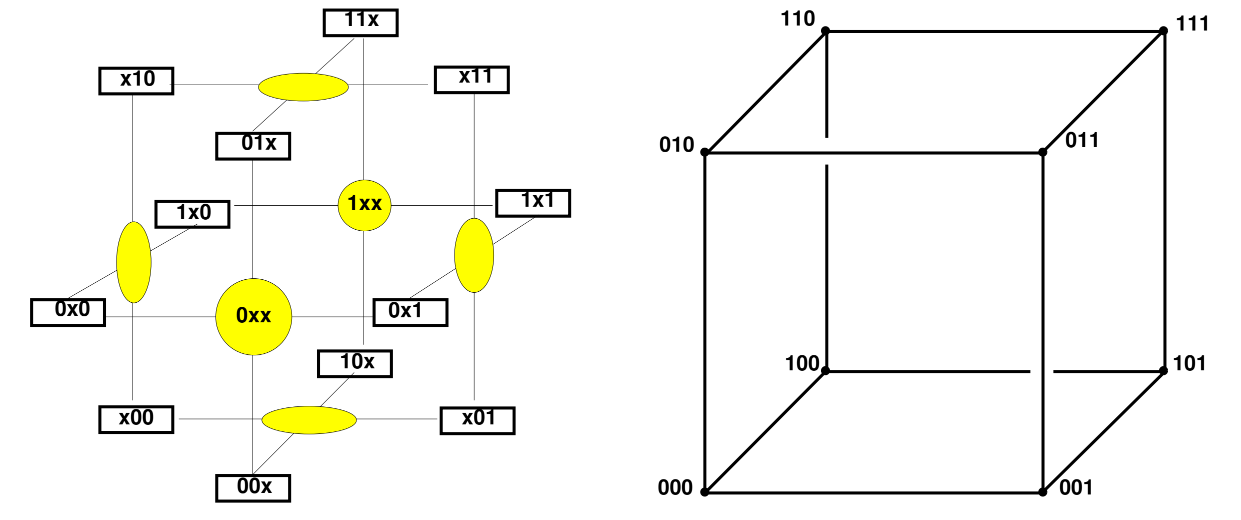


Abbildung 6.6: *Der dreidimensionale Kantenhypercube (links) und der dreidimensionale Hypercube (rechts).*

Für eine formalere Beschreibung ordnen wir den Kanten und Flächen wiederum Adressen zu. Da sich die beiden Ecken, an denen eine Kante sitzt, in genau einem Bit unterscheiden, bekommt sie die Adresse, bei der das fragliche Bit durch ein x ersetzt wurde. Umgekehrt kann jede Adresse $A \in \{0, 1, x\}^*$ mit $|A|_x = 1$ eindeutig einer Kante zugeordnet werden.

Flächen im Hypercube sind durch ihre vier begrenzenden Ecken gekennzeichnet. Die Adressen dieser Ecken unterscheiden sich in genau zwei Bitpositionen. Markieren wir diese mit einem x, so erhalten wir eine eindeutige Adresse für die Fläche. Umgekehrt können wir jeder Adresse $A \in \{0, 1, x\}^*$ mit $|A|_x = 2$ durch Ersetzen der beiden x mit 0 bzw. 1 die vier begrenzenden Ecken zuordnen und damit eine eindeutige Fläche bestimmen. Ersetzen wir dagegen nur ein x in A durch 0 oder 1, so erhalten wir die vier Kanten, die die Fläche umfassen.

Betrachten wir nun wieder die Punkte unseres dünnen Gitters. Wir verwenden zunächst das gleiche Verfahren wie im eindimensionalen Fall und setzen einen Punkt B_1xB_2 der Tiefe t auf den Rechner $B_1xB_20^{T-t}$. Die Punkte $p \in \mathcal{T}_T^{(2)}$, also die Punkte maximaler Tiefe, tragen dann dieselbe Adresse wie der Rechner, dem sie zugewiesen wurden. Wir analysieren im folgenden die Kommunikationsstruktur.

Die Söhne in 2-Richtung machen geringe Probleme. Der erste Sohn $p_1 = B_1xB_20$ liegt sogar auf demselben Rechner wie p. Der zweite Sohn $p_2 = B_1xB_21$ liegt auf $B_1xB_210^{T-t-1}$ und kann folglich über den Bus $B_1xB_2x0^{T-t-1}$ erreicht werden.

Schwierigkeiten machen dagegen die Söhne in 1-Richtung, nämlich $p_1 = B_10xB_2$ und $p_2 = B_11xB_2$. Vergleicht man die Adresse ihrer Rechner mit denen von p, so muß hier das Bitmuster B_2 um eine Stelle nach rechts verschoben werden. Im allgemeinen sind dafür $O(t)$ Schritte, d.h. Zwischenrechner, nötig.

Diese Einbettungsstrategie ist also ungünstig. Durch einen kleinen Kniff läßt sich aber der obige Nachteil umgehen. Statt den Gitterpunkt $p = B_1xB_2$ auf den Rechner $B_1xB_20^{T-t}$ zu legen, weisen wir ihm den Rechner $B_1x0^{T-t}B_2'$ zu. Dabei soll B_2' die zu B_2 spiegelbildliche Binärsequenz sein.

Die Rechner der Söhne in 1-Richtung lauten nun $B_10x0^{T-t-1}B_2'$ und $B_11x0^{T-t-1}B_2'$. Diese sind direkt benachbart zum Rechner von p, denn sie sind über den Bus $B_1xx0^{T-t-1}B_2'$ erreichbar. Bei den Söhnen in 2-Richtung ändert sich gegenüber der ersten Einbettungsstrategie wenig. Wieder liegt $p_1 = B_1xB_20$ lokal und der Rechner $B_1x0^{T-t-1}1B_2'$ des Punktes p_2 kann direkt über $B_1x0^{T-t-1}xB_2'$ erreicht werden. Wir fassen nun unsere bisherigen Überlegungen zusammen:

Lemma 6.13 *Für die gerade beschriebene Lastverteilung des zweidimensionalen dünnen Gitters der Tiefe T auf einen Kantenhypercube der Dimension $T+1$ gilt:*

1. *Je zwei Gitterpunkte p und p' derselben Tiefe t liegen auf verschiedenen Rechnern.*
2. *Alle Kommunikationen zwischen zwei Leveln können direkt (ohne Zwischenknoten) erfolgen.*

Da wir jetzt statt direkter Links zwischen je zwei Rechnern Kommunikationsbusse verwenden, auf die mehrere Rechner u.U. gleichzeitig zugreifen wollen, müssen wir noch untersuchen, ob es hierbei zu Verzögerungen kommen kann. Prinzipiell gilt, daß wegen der konstanten Zahl der Rechner pro Bus die Komplexität nicht schlechter werden kann. In unserem Fall können wir eine präzise Aussage machen:

Lemma 6.14 *Im Down-Teil des zweidimensionalen Dünngitteralgorithmus' wird beim Datenaustausch von einer Tiefe auf die nächste jeder Bus des Kantenhypercubes von höchstens einem Rechner aus benutzt.*

Beweis:
Bei fester Tiefe t ist jeder Gitterpunkt p eindeutig durch $B_1x0^{T-t}B_2'$, d.h. durch den Rechner, auf dem er liegt, beschrieben. Nach obigen Ausführungen werden für die Kommunikation die Busse $B_1x0^{T-t-1}xB_2'$ und $B_1xx0^{T-t-1}B_2'$ benötigt.

Folglich werden nur Busse, deren Adresse einer der beiden Formen entspricht, verwendet. Da t und T feststehen, sind B_1 und B_2' eindeutig bestimmt, und damit auch der Punkt p, von dem der Kommunikationswunsch ausgeht. Deshalb wird der Bus höchstens von einem einzigen Punkt p benutzt. $\diamondsuit$

Allerdings benötigt p den Bus $B_1xx0^{T-t-1}B_2'$ für alle t zweimal, nämlich für die Söhne $B_10x0^{T-t-1}B_2'$ und $B_11x0^{T-t-1}B_2'$. Für $t = T-1$ liegt sogar der dritte nichtlokale Sohn am selben Bus, so daß dieser gleich dreifach benutzt wird.

6.5 Dualisierung des Hypercubes

Sowohl der N-dimensionale Hypercube als auch der Kantenhypercube weisen eine mit N steigende Zahl an I/O-Ports pro Rechenknoten auf. Beim Hypercube sind es gerade N Ports pro Rechner, beim Kantenhypercube $N - 1$ Ports. Obwohl diese Zahlen nur logarithmisch mit der Gesamtzahl der Rechnerknoten ansteigen, sind die Topologien für große N nicht mehr realisierbar.

Für den Hypercube werden deshalb in der Literatur zahlreiche Alternativ-Topologien vorgeschlagen, die den Hypercube mehr oder weniger gut simulieren können, bei denen die Rechner aber jeweils nur eine konstante Zahl von Verbindungen besitzen. Prominente Vertreter sind hierbei die Cube-Connected-Cycles, Shuffle-Exchange- und die de-Bruijn-Graphen, mit denen wir uns bereits in Abschnitt 6.3 beschäftigt haben. Zur Vertiefung empfehlen wir [Lei92].

Für den Kantenhypercube schlagen wir hier eine andere Strategie vor. Die Busse werden statt auf die Flächen nun auf die *Ecken* des Hypercubes gelegt. Jeder Rechner (=Kante) von $H(N)$ benötigt deshalb nur zwei Ausgänge. Wegen des Rollentauschs zwischen Ecken und Kanten nennen wir diese Topologie *Dualhypercube*. In Abbildung 6.7 ist zur Veranschaulichung ein Dualhypercube der Dimension 3 mit Numerierung dargestellt. Zum Vergleich ist auf der linken Seite noch einmal der entsprechende Kantenhypercube angegeben.

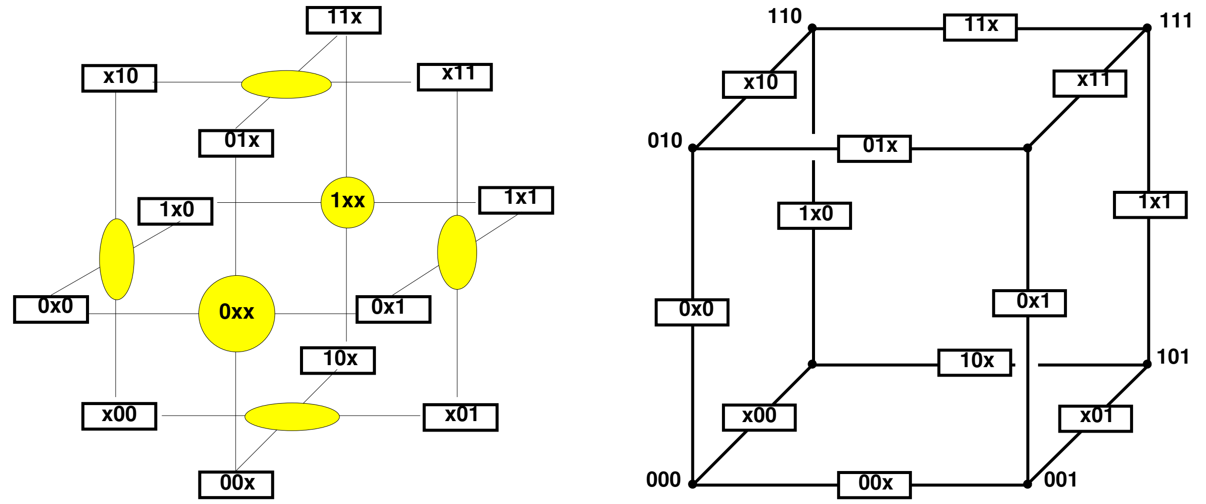


Abbildung 6.7: *Der dreidimensionale Kantenhypercube (links) und Dualhypercube (rechts).*

Die Adjazenzstruktur ist leicht an den Adressen abzuleiten: Ersetzt man das x in der Rechneradresse durch 0 und 1, so erhält man die beiden Busse, an denen der Rechner partizipiert. An einem Bus sitzen die Rechner mit denjenigen Adressen, die entstehen, wenn ein Bit der Busadresse durch x ersetzt wird. Zwei verschiedene Rechner sind genau dann benachbart, wenn ihre Adressen bis auf die beiden x-Positionen übereinstimmen.

Wie bewährt sich dabei die im vorigen Abschnitt entworfene Einbettungsstrategie, nämlich den Gitterpunkt $p = B_1 x B_2$ auf den Rechner $B_1 x 0^{T-t} B_2'$ zu legen?

Die Söhne in 1-Richtung auf den Rechnern $B_1 0 x 0^{T-t-1} B_2'$ und $B_1 1 x 0^{T-t-1} B_2'$ sind direkt benachbart zu p, denn sie sind über die Busse $B_1 000^{T-t-1} B_2'$ und $B_1 100^{T-t-1} B_2'$ zu erreichen.

Der Sohn in 2-Richtung $p_1 = B_1 x B_2 0$ verbleibt auf demselben Rechner. Der zweite Sohn $p_2 = B_1 x B_2 1$ liegt dagegen nicht einmal auf einem Nachbarrechner. Zum Glück kann er aber über nur einen Zwischenrechner erreicht werden. Dafür gibt es zwei Möglichkeiten:

$$\begin{array}{cc}
1. & 2. \\
B_1 x 0^{T-t} B_2' & B_1 x 0^{T-t} B_2' \\
(B_1 00^{T-t} B_2') & (B_1 10^{T-t} B_2') \\
B_1 00^{T-t-1} x B_2' & B_1 10^{T-t-1} x B_2' \\
(B_1 00^{T-t-1} 1 B_2') & (B_1 10^{T-t-1} 1 B_2') \\
B_1 x 0^{T-t-1} 1 B_2' & B_1 x 0^{T-t-1} 1 B_2'
\end{array}$$

Die Adressen sind dabei von oben nach unten zu lesen, wobei die benutzten Busse (= Ecken) in runden Klammern angegeben sind. Damit ist folgendes Lemma bewiesen:

Lemma 6.15 *Der Abhängigkeitsgraph eines zweidimensionalen dünnen Gitters der Tiefe T läßt sich mit Dilation 2 in einen Dualhypercube der Dimension $T+1$ einbetten.*

Den Nachteil, den wir uns gegenüber dem Kantenhypercube eingehandelt haben, liegt darin, daß jeder Bus eines Dualhypercubes der Dimension N mit N Rechnern inzidiert. Es besteht also die Möglichkeit, daß mehrere Rechner ein und denselben Bus gleichzeitig beanspruchen wollen. Leider tritt dieser Fall tatsächlich ein:

Lemma 6.16 *Beim Datenaustausch zwischen Gitterpunkten zweier aufeinanderfolgenden Tiefen t und $t+1$ wird der Bus 0^{T+1} gleichzeitig von $t+1$ Rechnern beansprucht.*

Beweis (durch vollständige Induktion):
Für $t = 0$ werden Daten zwischen Punkt $p = x$ und $p = 0x$ ausgetauscht. Auf die Rechner übertragen, heißt das:

$$x0^T \longrightarrow (0^{T+1}) \longrightarrow 0x0^{T-1}$$

Für $T > t > 0$ liegen die Punkte $p_i := 0^i x 0^{t-i} \quad (i = 0, \ldots, t)$ in $\mathcal{T}_t^{(2)}$. Sie befinden sich auf den Rechnern $R_i = 0^i x 0^{T-i}$. Ein Sohn von p_i ist $p_i' := 0^{i+1} x 0^{t-i-1}$. Dieser ist auf dem Rechner $R_i' = 0^{i+1} x 0^{T-i-1}$ plaziert. Somit muß für jedes $i \in \mathbb{Z}_t$ der Bus 0^{T+1} zum Datenaustausch zwischen R_i und R_i' benutzt werden. $\Diamond$.

Die $t+1$ Kommunikationswünsche auf dem gemeinsamen Bus müssen sequentiell abgearbeitet werden. Wegen $\sum_{t=0}^{T} t = O(T^2)$ ergibt sich sofort das folgende

Korollar 6.17 *Der Datenaustausch innerhalb einer kompletten Iteration eines zweidimensionalen Dünngitteralgorithmus' der Tiefe T benötigt $O(T^2)$ Zeitschritte.*

Zum Schluß dieses Abschnitts sei noch auf [Pfa95] verwiesen. Dort werden einige theoretische Eigenschaften des Dualhypercubes zusammengestellt und außerdem über die praktischen Erfahrungen mit einem dreidimensionalen Dualhypercube berichtet.

6.6 Hypercube–Topologien für D-dimensionale Gitter

Die in den vorigen Abschnitten vorgestellten Topologien lassen sich natürlich auf den D-dimensionalen Fall verallgemeinern. Dabei identifizieren wir nun statt der *Kanten* des Hypercubes allgemein die Subcubes der Dimension $D-1$ mit den Rechenknoten. Bei einem dreidimensionalen dünnen Gitter würden wir beispielsweise die (zweidimensionalen) Flächen des Hypercubes als Knoten verwenden.[1] Formal definieren wir die k-Subcubes eines N-dimensionalen Hypercubes wie folgt:

$$C_k(N) := \left\{ \{v^1, \ldots, v^{2^k}\} \in \binom{V(N)}{2^k} : d(v^1, \ldots, v^{2^k}) = k \right\}$$

Die Menge der Ecken von $H(N)$ ist also gerade $C_0(N)$, die Menge der Kanten $C_1(N)$. $C_2(N)$ bezeichnet die Flächen und $C_3(N)$ die Würfel von $H(N)$. Jedes Element $S = \{v^1, \ldots, v^{2^k}\}$ der Menge $C_k(N)$ kann eindeutig durch eine N-stellige Adresse $s_1 \ldots s_T \in \{0, 1, x\}^N$ repräsentiert werden:

$$s_i = \begin{cases} 0 & \text{falls} \quad \forall j \in \{1, \ldots, 2^k\} : \quad v_i^j = 0 \\ 1 & \text{falls} \quad \forall j \in \{1, \ldots, 2^k\} : \quad v_i^j = 1 \\ x & \text{sonst} \end{cases}$$

Aus dieser Darstellung folgt sofort:

$$|C_k(N)| = \binom{N}{k} \cdot 2^{N-k}$$

Deshalb entspricht die Zahl der Gitterpunkte maximaler Tiefe T eines D-dimensionalen dünnen Gitters genau der Zahl der $(D-1)$-Subcubes eines $(T+D-1)$-dimensionalen Hypercubes.

Anschaulich gesprochen erhalten wir die begrenzenden Ecken eines k-Subcube dadurch, daß wir alle x-Positionen seiner Adresse durch 0 oder 1 ersetzen. Das gilt auch für die inzidierenden Kanten, Flächen, etc. Hier müssen alle bis auf eine, zwei, etc. x-Positionen mit 0 oder 1 besetzt werden.

Zwei k-Subcubes S_1 und S_2 teilen sich einen gemeinsamen $(k-1)$-Subcube genau dann, wenn ihre Adressen sich in einer x-Position unterscheiden und alle anderen Positionen identisch sind. Deshalb besitzt jeder k-Subcube eines N-dimensionalen Hypercubes genau $2k(N-k)$ benachbarte k-Subcubes.

Beim Dualhypercube wurden die Rechner mit den Kanten, d.h. den 1-Subcubes, identifiziert und die Verbindungstopologie über die Ecken, d.h. die 0-Subcubes, definiert. Greift man diese Idee auf, so könnte man für D-dimensionale dünne Gitter der Tiefe T eine Architektur entwerfen, bei denen die $(D-1)$-Subcubes von $H(D+T-1)$ mit den

[1] Man beachte die unterschiedlichen Dimensionsbegriffe!

Rechnern gleichgesetzt werden. Diese wären verbunden über Busse, die den $(D-2)$-Subcubes entsprächen. Jeder Rechner hinge dann an $2(D-1)$ Bussen und an jedem Bus säßen $T+1$ Rechner. Insgesamt hätte man dann gerade soviele Rechner zur Verfügung wie Gitterpunkte der Tiefe T.

Genauso könnte man den Kantenhypercube verallgemeinern. Statt auf die $(D-2)$-Subcubes von $H(D+T-1)$ müßte man die Busse auf die D-Subcubes legen. Jeder Rechner bräuchte dann T Ports und an jedem Bus lägen $2D$ Rechner.

Leider läßt sich aber das Einbettungsschema für zweidimensionale dünne Gitter aus dem vorigen Abschnitt nicht so leicht auf höhere Dimensionen verallgemeinern. Der Trick bestand ja darin, bei der Abbildung von Gitterpunktadressen auf Rechneradressen die zweite Binärsequenz zu spiegeln und „Puffer-Nullen“ zwischen die beiden Adreßteile zu schalten. Dadurch konnte verhindert werden, daß beim Übergang zu den hierarchischen Söhnen lange Bitmuster verschoben werden mußten. Bei den D Binärsequenzen in $B_1 x \ldots x B_D$ kann dieser Trick natürlich nicht mehr angewandt werden.

6.7 Lastverteilungsstrategien für adaptive dünne Gitter

Wir kehren jetzt von der Theorie zurück und wenden uns in diesem Abschnitt wieder einer sehr praxisorientierten Frage zu, nämlich der Lastverteilung sogenannter adaptiver dünner Gitter. Wir erinnern uns daran, daß uns bisher als Entscheidungskriterium, welche Funktionen zur Dünngitterbasis gehören sollen, allein deren Tiefe diente. Dadurch ergab sich eine regelmäßige Struktur – Verfeinerungen betrafen stets das gesamte Definitionsgebiet. Die zugehörigen dünnen Gitter werden daher als *regulär* bezeichnet.

Um nun die Punktezahl noch weiter zu reduzieren bzw. die Genauigkeit weiter zu erhöhen, empfiehlt es sich, nur dort zu verfeinern, wo die zu approximierende Funktion u „rauher“ ist. Ein Maß dafür ist der Betrag der Ableitung $\frac{\partial^{2D}}{\partial x_1^2 \partial x_2^2 \ldots \partial x_D^2} u$, die wiederum eng mit dem hierarchischen Überschuß $\tilde{u}_I$ an der betreffenden Stelle gekoppelt ist. Denn $\tilde{u}_I$ entspricht bis auf den Faktor $(-1)^D 2^{-D} h_{i_1}^2 \ldots h_{i_D}^2$ dem üblichen Differenzenquotienten zur Approximation von $\frac{\partial^{2D}}{\partial x_1^2 \partial x_2^2 \ldots \partial x_D^2} u$.

Eine vernünftige und einfach zu implementierende Adaptivitätsstrategie sieht demnach so aus, daß ein Punkt genau dann weiter durch seine $2D$ hierarchischen Söhne verfeinert wird, wenn der hierarchische Überschuß größer als eine vorgegebene Schranke ϵ ist. In dieser Sichtweise ist das reguläre dünne Gitter lediglich ein Spezialfall eines adaptiven dünnen Gitters. Abbildung 6.8 zeigt ein Beispiel eines so erzeugten adaptiven dünnen Gitters mit zugehöriger Funktion u. Neben dieser Strategie gibt es weitere Alternativen, auf die wir hier nicht näher eingehen werden. Wir verweisen in diesem Zusammenhang auf die Arbeiten [Bun92, Bon95].

Wie verteilen wir nun die Punkte eines adaptiven dünnen Gitters möglichst gleichmäßig auf die uns zur Verfügung stehenden Prozessoren? Wir stoßen hier im Gegensatz zum

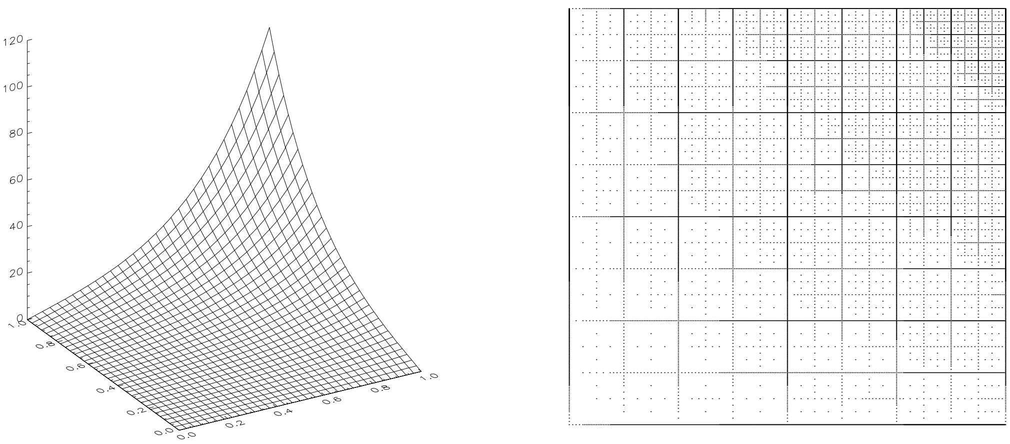


Abbildung 6.8: *Ein adaptives dünnes Gitter für die Funktion* $u = sinh(3x_1) \cdot sinh(3x_2)$.

regulären Fall auf das Problem, daß wir nicht mehr a priori wissen, welche Punkte wo entstehen. Da wir bei der Lösung partieller Differentialgleichungen die darzustellende Lösungsfunktion u nicht kennen, entwickelt sich die Gestalt des adaptiven Gitters erst im Laufe der Iterationen.

Nun ist es aber häufig so, daß sich die grobe Struktur des adaptiven Gitters schon nach wenigen Iterationen abzeichnet und sich im folgenden die Teilgebiete mehr oder weniger *regelmäßig* verfeinern. Legen wir diese Annahme zugrunde, so empfiehlt es sich, diese wenigen Schritte zunächst auf einem Rechner durchzuführen, die Struktur zu analysieren und daraus eine günstige Verteilung abzuleiten.

Um diese Verteilung automatisieren zu können, kehren wir kurz zu regulären Gittern zurück und untersuchen, wie sich die Zahl der Blätter eines regulären dünnen Gitters mit zunehmender Tiefe entwickelt. Wegen

$$\frac{\tilde{L}_{T+1}^{(D)}}{\tilde{L}_T^{(D)}} = \frac{2^{T+1}\binom{D+T}{D-1}}{2^T\binom{D+T-1}{D-1}} = \frac{2(T+D)}{T+1} \approx 2 \quad \text{für } T \gg D$$

folgt für ganzzahlige k sofort

$$\frac{\tilde{L}_{T+k}^{(D)}}{\tilde{L}_T^{(D)}} \approx 2^k. \tag{6.20}$$

Die Zahl der Blätter zweier regulärer Gitter der Tiefen T und $T + k$ unterscheiden sich also etwa um den Faktor 2^k. Der Faktor bleibt auch erhalten, wenn die beiden regulären Gitter gleichmäßig verfeinert werden. Auch bei adaptiven Gittern bildet sich dieser Faktor heraus, wenn ab einer gewissen Tiefe regulär verfeinert wird. Dies motiviert den nachfolgenden Algorithmus zur Verteilung adaptiver dünner Gitter:

Algorithmus ADAPTLAST:

1. Führe n_0 Iterationsschritte durch.
2. Unterteile das D-dimensionale Gebiet in $M := m^D$ Teile $C_1, \ldots, C_M$ gleichen Volumens auf.
3. Bestimme für $k \leq M$:

$$\begin{aligned} t_k &:= \text{maximale Tiefe der Punkte in } C_k \\ t_{\min} &:= \min_k(t_k) \\ t'_k &:= t_k - t_{\min} \end{aligned}$$

4. Zerlege alle C_k in $2^{t'_k}$ Teile T_i gleichen Volumens.
5. Verteile die $\sum_{k=1}^{M} 2^{t'_k}$ Teilgebiete T_i möglichst gleichmäßig auf die zur Verfügung stehenden Prozessoren.

Die Parameter n_0 und m sind dabei geeignet festzulegen. n_0 sollte klein gegenüber der Gesamtzahl der Iterationsschritte sein, aber so groß, daß sich die adaptive Struktur bereits abzeichnet. Damit die nachfolgende Zerlegung problemlos vollzogen werden kann, sollte m aufgrund der Konstruktion des dünnen Gitters günstigerweise eine (kleine) Zweierpotenz sein. Die anschließende weitere Aufteilung der C_k in $2^{t'_k}$ Teilgebiete ist dann ebenfalls leicht zu bewerkstelligen. Wegen (6.20) ist zu erwarten, daß bei den folgenden Verfeinerungsschritten alle Teilgebiete T_i in etwa die gleiche Zahl von Punkten enthalten. Zur Verdeutlichung ist in Abbildung 6.9 ein Beispiel für $D = 2, m = 4$ dargestellt.

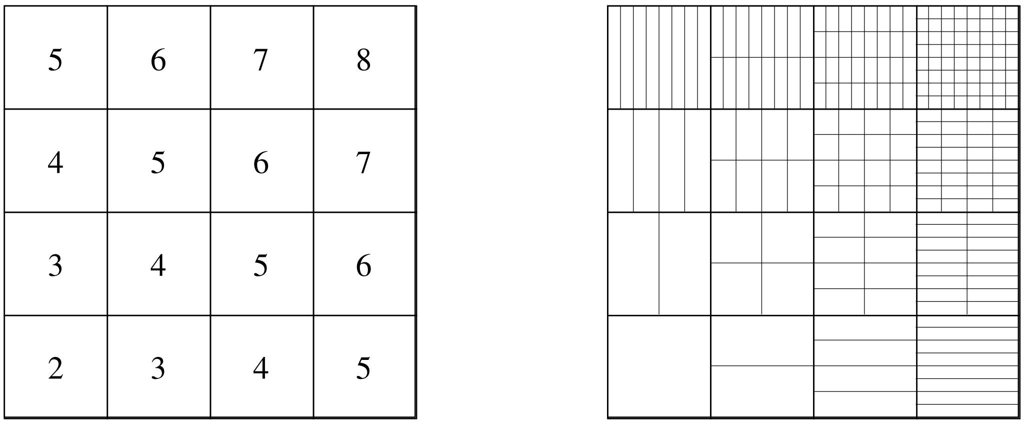


Abbildung 6.9: *Beispiel für die Aufteilung eines adaptiven dünnen Gitters nach Algorithmus ADAPTLAST. a) Das zweidimensionale Gitter wird in $M = 4^2$ Quadrate aufgeteilt und deren maximale Tiefe t_k bestimmt. b) Jedes der Quadrate wird in $2^{t'_k} = 2^{t_k - 2}$ Teilgebiete zerlegt, die dann möglichst gleichmäßig unter den zur Verfügung stehenden Prozessoren aufgeteit werden müssen.*

Ein Nachteil der so erhaltenen Verteilung besteht in der zu erwartenden großen Streuung der Punkte auf die Prozessoren. Deshalb bietet sich der Algorithmus nur in Verbindung mit leistungsfähigen Kommunikationsnetzen an.

7 Leistungsmessungen

In diesem Kapitel wollen wir nun die bisher erarbeiteten Konzepte auf ihre Praxistauglichkeit untersuchen. Die in Kapitel 5 entworfenen FASAN-Algorithmen werden auf einem Netz von 16 Workstations des Typs HP 9000/720 gestartet und die Laufzeiten verglichen. Dabei interessiert uns zunächst, wie das FASAN-System gegenüber herkömmlichen C-Programmen auf einem Rechner abschneidet. Dann wollen wir vergleichen, welcher zusätzliche Aufwand durch die allgemeine D-dimensionale FASAN-Beschreibung im Vergleich zu Spezialversionen des FASAN-Programms für feste Dimensionalität entsteht. Außerdem möchten wir natürlich wissen, wie gut das Parallelisierungspotential durch das FASAN-Programm genutzt wird.

7.1 Kosten der funktionalen Beschreibung

Der FASAN-Compiler und das FASAN-Laufzeitsystem existieren bisher nur in einer prototypischen Implementierung. Die Programme werden in C-Code übersetzt, der mit einem Laufzeitsystem auf PVM-Basis zusammenarbeitet. Es ist daher zu erwarten, daß wir im Vergleich zu einer expliziten parallelen oder sequentiellen Implementierung in einer Standardsprache mit Performanzeinbußen rechnen müssen. Außerdem haben wir in Kapitel 5 einen eigenen Algorithmus zur Multiplikation mit der Steifigkeitsmatrix speziell für den parallelen Fall entworfen, der auch im sequentiellen eine grundsätzlich andere Struktur aufweist als die bisherigen Algorithmen.

Bevor wir also mit den parallelen Tests beginnen, wollen wir kurz die *sequentiellen* Laufzeiten unseres FASAN-Programms mit der C++ -Implementierung des sich im sequentiellen Bereich mehr und mehr durchsetzenden unidirektionalen Algorithmus', wie er in [Bal94] vorgestellt worden ist, vergleichen. Da die betrachteten Algorithmen doch sehr verschieden sind, kommt es uns nur darauf an, einen ungefähren Eindruck von der Größenordnung der tatsächlichen Laufzeit zu gewinnen.

Wie in Tabelle 7.1 zu sehen ist, sind die Laufzeiten des FASAN-Programms bei niedriger Dimensionalität deutlich langsamer als das C++ -Programm. Je höher die Dimensionalität des Problems, um so günstiger wird allerdings der in FASAN kodierte Algorithmus, bis er schließlich die C++ -Version überholt. Dies ist wohl darauf zurückzuführen, daß die hohe Zahl der Baumtraversionen und die damit verbundenen rekursiven Funktionsaufrufe den unidirektionalen Algorithmus stark bremsen, wohingegen der FASAN-Algorithmus viele Berechnungen an einem Knoten innerhalb eines Funktionsaufrufs bündelt. Hier zeigt sich außerdem, daß der Mehraufwand, der programmtechnisch aufgrund der funktionalen Beschreibung nötig war, bei hoher Dimensionalität in den Hintergrund tritt.

In der letzten Spalte sind die parallelen Laufzeiten des FASAN-Programms auf 16 Workstations angegeben. Diese sind – bis auf eine Ausnahme – bei niedriger Dimensionalität wenigstens noch schneller als der C++ -Algorithmus.

D	T	C++	FASAN (1)	FASAN (16)
1	15	7.77	98.63	6.32
2	12	35.91	355.52	31.85
3	9	47.94	280.21	45.36
4	4	1.87	7.81	3.28
5	4	8.75	24.21	6.60
6	4	37.72	45.36	9.96
7	4	171.40	97.21	18.43

Tabelle 7.1: *Vergleich der Laufzeiten des FASAN-Algorithmus' und einer C++ -Implementierung. Bei hoher Dimensionalität wird das C++ -Programm durch das FASAN-Programm überholt. Bei kleiner Dimensionalität ist zumindest die parallele Version schneller als das sequentielle C++ -Programm.*

Insgesamt sind die Ergebnisse doch sehr ermutigend, zumal die Implementierung von FASAN noch ein großes Optimierungspotential (etwa hinsichtlich der Verwaltung des Datenflußgraphen sowie der damit zusammenhängenden Speicherverwaltung und -verbrauch) offeriert. Dann ist auch bei niedriger Dimensionalität mit akzeptablen Laufzeiten zu rechnen. Weiterhin ist zu beachten, daß gerade bei niedriger Dimensionalität ein gewisser Zusatzaufwand aus der allgemeinen D-dimensionalen Formulierung des Algorithmus' resultiert. Dies wollen wir im folgenden etwas näher betrachten.

7.2 Kosten der D-dimensionalen Beschreibung

In Kapitel 5 ist klar geworden, daß die allgemeine Formulierung eines D-dimensionalen Dünngitteralgorithmus' in FASAN sehr aufwendig ist. Wir wollen in diesem Abschnitt kurz untersuchen, welche zusätzlichen Kosten dies für die Laufzeiten bedeutet. Dazu vergleichen wir das speziell für Dimensionalität 1 entworfene FASAN-Programm `P1` mit dem allgemeinen D-dimensionalen Programm `PD`, das ebenfalls für $D = 1$ untersucht wird. Beide Programme wurden in ihren wesentlichen Zügen in Kapitel 5 vorgestellt.

Wie Tabelle 7.2 zeigt, werden die Laufzeiten durch den D-dimensionalen Overhead in etwa verdoppelt. Dies ist eigentlich recht gering, wenn man bedenkt, welch komplexe Kabelbäume in der allgemeinen Version aufgebaut und verwaltet werden müssen. Gerade der häufige rekursive Umbau schlägt erstaunlich niedrig zu Buche.

Tiefe	10	11	12	13	14	15
P1	2.06	3.45	5.87	11.43	21.69	43.95
PD	3.87	6.62	12.79	25.64	51.16	98.64

Tabelle 7.2: *Laufzeiten zur funktionalen Berechnung der Matrixmultiplikation für $D = 1$. Verglichen wurden die Zeiten der eindimensionalen Spezialversion und des allgemein gehaltenen D-dimensionalen Algorithmus'.*

T	ITER	t_1	t_{16}	Speedup
11	1	6.615	1.632	4.05
11	2	12.239	2.145	5.71
11	4	22.468	3.875	5.80
11	8	45.585	7.255	6.28
12	1	12.792	1.640	7.80
12	2	25.055	2.860	8.76
12	4	45.308	5.005	9.05
12	8	89.513	9.729	9.20
13	1	25.642	2.296	11.17
13	2	46.791	4.177	11.20
13	4	92.446	7.705	12.00
13	8	185.840	14.886	12.48
14	1	51.163	3.662	13.97
14	2	95.375	6.754	14.12
14	4	179.382	12.781	14.03
14	8	361.412	24.676	14.65
15	1	98.636	6.322	15.60
15	2	187.312	11.923	15.71
15	4	360.717	22.778	15.83
15	8	702.825	44.350	15.84

Tabelle 7.3: *Laufzeiten für $D = 1$ und verschiedene Tiefen und Iterationszahlen.*

Bei Spezialversionen für höhere Dimensionalität dürfte der Geschwindigkeitsvorteil schnell verspielt sein und den ungleich höheren Programmieraufwand nicht lohnen.

7.3 Paralleler Speedup

Wir kommen jetzt zur Kernfrage dieses Kapitels, nämlich wie stark der FASAN-Algorithmus sich durch parallelen Einsatz mehrerer Rechner beschleunigen läßt. Dazu wurde für die folgenden Tests das in Kapitel 5 beschriebene D-dimensionale Programm `PD` zur Matrixmultiplikation verwendet. Zur Lastverteilung wurde das in Abschnitt 6.2 entworfene Verfahren benutzt. Wie wir sehen werden, hängt der erzielte Speedup, also die durch den Einsatz mehrerer Rechner erzielte Beschleunigung, stark von den Parametern D (Dimensionalität), T (Tiefe des Gitters) und $ITER$ (Zahl der Iterationsschritte) ab.

Betrachten wir zunächst wieder den eindimensionalen Fall ($D = 1$). Hier sind besonders gute Speedups zu erwarten, da die Kommunikations- und die Abhängigkeitsstruktur zwischen den parallelen Teilen sehr einfach sind. Die Rechner können also weitgehend unabhängig voneinander rechnen.

D	T	t_1	t_{16}	Speedup
2	9	32.828	6.998	4.69
2	10	72.916	10.140	7.19
2	11	160.931	17.943	8.97
2	12	355.521	31.853	11.16
3	8	112.951	23.196	4.87
3	9	280.217	45.360	6.18
3	10	675.404	90.143	7.49

D	T	t_1	t_{16}	Speedup
2	9	30.695	9.225	3.33
2	10	66.964	14.336	4.67
2	11	156.829	28.090	5.58
2	12	317.552	49.687	6.39
3	8	101.476	45.955	2.21
3	9	249.016	83.510	2.98
3	10	617.773	171.106	3.61

Tabelle 7.4: *Parallele Laufzeitmessungen für eine Iteration im Falle $D = 2$ und $D = 3$. Die Ergebnisse in der rechten Tabelle sind deutlich schlechter, weil hier die Verbindungsströme direkt gezogen werden. Bei den Messungen auf der linken Seite gehen die Daten über die hierarchischen Väter.*

Tabelle 7.3 zeigt Laufzeitmessungen mit verschiedenen Tiefen und Iterationszahlen auf einem und auf 16 Rechnern. Daraus geht deutlich hervor, daß sich höhere Speedups erst bei größeren Gittertiefen oder Iterationszahlen bilden. Der Grund dafür ist klar. Werden die Gitter tiefer, so ist der Anteil der Rechenzeit gegenüber der Kommunikationszeit höher. Bei mehreren Iterationen kürzt sich außerdem der Aufwand für die Initialisierung des Gitters und die Auffaltung des Funktionennetzes dank des in Abschnitt 4.2 vorgestellten „Recycling-Prinzips“ des FASAN-Laufzeitsystems nach und nach heraus. Die Messungen belegen im übrigen, daß es mit dem FASAN-System grundsätzlich möglich ist, das gesamte Parallelisierungspotential einer Anwendung auszuschöpfen.

Wie sieht nun die Situation bei höherer Dimensionalität aus? Prinzipiell gilt, daß mit steigender Dimensionalität des Problems die Kommunikationsstruktur komplexer wird. Da die Kommunikationsleistung des Workstationnetzwerks um Größenordnungen langsamer als die Rechenleistung ist, wird das Erzielen hoher Speedups erschwert.

Die Laufzeitmessungen in Tabelle 7.4 (links) zeigen tatsächlich etwas schlechtere Speedups für $D = 2$ und $D = 3$. Sie werden allerdings mit zunehmender Tiefe besser. Dieses Verhalten war zu erwarten, da in einem Punkt des Gitters bei Dimensionalität 2 noch wenig gerechnet werden muß. Erst wenn die Tiefe des Gitters steigt, ist der kumulierte Rechenaufwand für alle Punkte eines Prozessors groß genug, um den hohen Kommunikationsaufwand zu rechtfertigen. Leider konnten größere Tiefen nicht gemessen werden, da sonst der zur Verfügung stehende Hauptspeicher nicht mehr ausgereicht hätte und die Messungen durch Swap-Effekte verfälscht worden wären.

Die rechte Seite der Tabelle 7.4 zeigt übrigens die Meßergebnisse der bzgl. der Kommunikation nicht optimierten Version, in der die Kabelbäume stets direkt gezogen werden (vgl. Abschnitt 5.3.3). Hier sind zwar die sequentiellen Laufzeiten erwartungsgemäß etwas schneller, da weniger Ströme verwaltet werden müssen, die Laufzeiten auf 16 Rechner sind dagegen deutlich langsamer.

An dieser Stelle wird klar, daß der feingranulare Parallelisierungsansatz funktionaler Sprachen und die Betonung der sofortigen Verfügbarkeit der Daten speziell in Datenflußsprachen bei realistischen Anwendungen im Bereich der numerischen Mathematik

D	T	ITER	t_1	t_{16}	Speedup
5	4	1	24.210	6.599	3.67
5	4	2	43.960	7.268	6.05
5	4	4	80.670	9.807	8.23
5	4	8	155.543	15.097	10.30
5	4	16	304.189	27.979	10.87
6	4	1	45.362	9.956	4.56
6	4	2	89.682	12.487	7.18
6	4	4	175.739	18.190	9.66
6	4	8	366.702	30.259	12.12
6	4	16	703.162	54.677	12.86
7	4	1	97.211	18.435	5.27
7	4	2	194.919	24.353	8.00
7	4	4	388.690	37.656	10.32
7	4	8	754.935	64.288	11.74
7	4	16	1551.631	115.808	13.40

Tabelle 7.5: *Bei höherer Dimensionalität sind aufgrund des exponentiell steigenden Berechnungsaufwands pro Punkt wieder gute Speedups zu erzielen.*

oft einen Schritt zu weit gehen. Es kommt dann nämlich zu einer starken Verflechtung der Kommunikationsbeziehungen zwischen den parallel arbeitenden Rechenknoten, wodurch die Kommunikationszeit die Laufzeit dominieren kann. Diese Problematik wird in parallelen funktionalen Sprachen bisher noch relativ wenig beachtet.

Da der Rechenaufwand pro Knoten exponentiell mit der Dimensionalität steigt, ist das Verhältnis zwischen Kommunikationaufwand und Rechenaufwand bei höherer Dimensionalität wieder günstiger. Dies bestätigen auch die folgenden Messungen. Hier wurden verschiedene Dimensionalitäten bei gleicher Tiefe 4 untersucht. Tiefen kleiner als 4 sind auf 16 Rechnern uninteressant, da in der gewählten Lastverteilungsstrategie stets nur 2^T Rechner zum Zuge kommen. Größere Werte für T konnten aber wegen der hohen Dimensionalität aus Speicherplatzgründen nicht untersucht werden.

Interessanterweise steigen die Speedups in Tabelle 7.5 bei gleicher Dimensionalität und Tiefe mit der Iterationszahl stark an. Dies liegt daran, daß gerade in (bzw. vor) der ersten Iteration der komplette D-dimensionale Binärbaum für die Knotenwerte aufgebaut und im Speicher alloziiert werden muß, so daß hier ein entsprechender Aufwand entsteht. Dieser Aufwand kürzt sich in den nächsten Iterationen wieder heraus. Außerdem werden die Speedups erwartungsgemäß mit zunehmender Dimensionalität besser.

7.4 Zusammenfassung

Wir wollen die Ergebnisse dieses Kapitels zusammenfassen. Insgesamt haben die Laufzeitmessungen exemplarisch für die Lösung der Helmholtzgleichung gezeigt, daß es sich bei komplex strukturierten Algorithmen durchaus lohnen kann, auf höhersprachliche Konzepte wie funktionale Programmiersprachen überzugehen.

Zwar ist zu erwarten, daß auch in einer effizienteren Implementierung von FASAN noch zusätzliche Kosten gegenüber einem optimal kodierten C-Programm zu erwarten sind. Diese halten sich aber schon jetzt bei einfachen Algorithmen in Grenzen und treten bei steigender Komplexität der Programme hinsichtlich ihrer Struktur und ihres Rechenaufwands in den Hintergrund.

Speziell im Falle der hier betrachteten Problemstellung zeigt sich, daß der in Kapitel 5 entworfene Algorithmus zu Matrixvektormultiplikation mit der Steifigkeitsmatrix der Helmholtzgleichung besonders gut für höhere Dimensionalität und Gittertiefen geeignet ist. Die derzeitige FASAN-Implementierung bedingt, daß die Effizienz des parallelen Algorithmus' mit steigenden Iterationszahlen zunimmt.

Niedrige Iterationszahlen sind praktisch uninteressant, da bei einfachen Lösungsverfahren wie CG oder Gauß-Seidel ohnehin sehr viele Iterationsschritte nötig sind (in der Größenordnung von 1000 und mehr). Nutzt man dagegen aufwendigere Verfahren wie etwa die Mehrgittertechnik, so kommt man zwar mit sehr wenigen Iterationsschritten (Größenordnung kleiner 10) aus. Diese sind aber dafür wieder wesentlich rechenintensiver und benötigen u.U. mehrere Matrixmultiplikationen innerhalb eines Schrittes.

8 Schlußbetrachtung

Die bisher untersuchten numerischen Probleme auf dünnen Gittern führten zu Algorithmen derselben Struktur, was die Abhängigkeit der Gitterpunkte betrifft. In der vorliegenden Arbeit haben wir exemplarisch die Parallelisierungsaspekte eines typischen Dünngitteralgorithmus', der Matrixvektormultiplikation mit der Steifigkeitsmatrix der Helmholtzgleichung, untersucht.

Wir haben dabei thematisch einen großen Bogen geschlagen. Zu Beginn stand der mathematische Entwurf und die theoretische Analyse des Parallelisierungspotentials zweier algorithmischer Varianten, des unidirektionalen Algorithmus' und der D-dimensionalen Überschußschleuder. Bei letzterer haben wir in dieser Arbeit eine gegenüber der in [BZ92] beschriebenen Implementierung stark verbesserte Variante entworfen, die nunmehr mit nur zwei statt bisher $O(D)$ Baumdurchläufen auskommt. Dies ist vor allem für die praktische Parallelisierung entscheidend, weil mit jedem Baumdurchlauf neue Synchronisationsstellen und Kommunikationszeitpunkte entstehen.

Bei dem anschließenden Versuch, den neuen Algorithmus parallel zu implementieren, mußten wir feststellen, daß dieses Unterfangen innerhalb des Kontexts herkömmlicher imperativer Programmiersprachen auf große Schwierigkeiten stößt. Das mag auch der Grund dafür gewesen sein, daß die optimierte Version der D-dimensionale Überschußschleuder erst im Zuge der Untersuchung funktionaler Sprachmittel entdeckt wurde.

Wir haben hier nun einen eigenen sprachlichen Ansatz eingeführt, der sich besonders für die Parallelisierung komplexer, hierarchisch strukturierter, rekursiver Iterationsalgorithmen eignet. Die Sprache FASAN wurde dann benutzt, um sukzessiv die Algorithmen für die ein-, zwei- und D-dimensionale Variante zu beschreiben. Dank der Datenflußsemantik von FASAN und des neuartigen Kabelbaumprinzips war es dabei möglich, die optimale parallele Komplexität zu beschreiben, ohne auf die übersichtliche Gebietszerlegungssyntax zu verzichten.

Die Programme für den D-dimensionalen Fall muten auf den ersten Blick recht kompliziert an. Wir geben aber zu bedenken, daß wir mit jeweils nur einer rekursiven Funktion zur Beschreibung des Up- und des Down-Zyklus' für beliebige Dimensionen und Tiefen des dünnen Gitters eine außerordentliche Kürze und Prägnanz gewonnen haben. Außerdem ist derselbe Algorithmus sowohl für den sequentiellen als auch für den parallelen Bereich verwendbar, wodurch sich etwa das Debugging auf den wesentlich leichter zu überblickenden sequentiellen Fall beschränken läßt.

FASAN definiert zwar komfortable Schnittstellen für die Lastverteilung, aber eine allgemeine Strategie hierfür wird nicht vorgegeben. Der Entwurf und die Analyse verschiedener Verteilungsstrategien war deshalb Thema des Kapitels 6. Ideen für solche Verteilungsstrategien lieferte die vorab entworfene Kodierung der Punkte eines dünnen Gitters. Sie führte uns auf eine universelle Strategie, die unabhängig von der Dimension leicht zu implementieren ist und in einer selbst für eine hohe Zahl von Rechenknoten nahezu optimal ausgewogenen Verteilung resultiert. Sie wurde in Kapitel 7 auch für diverse Parameter praktisch untersucht.

Außerdem haben wir in Kapitel 6 die Rolle der Kommunikation für verschiedene skalierbare Rechnertopologien untersucht. Für zweidimensionale Probleme konnten wir eine neuartige hypercubeähnliche Architektur finden, in die der Kommunikationsgraph mit optimaler konstanter Dilation eingebettet werden kann.

Die Messungen aus Kapitel 7 haben schließlich gezeigt, daß sich der Umstieg auf funktionale Konzepte auch im Numerikbereich durchaus bezahlt machen kann. Die Implementierung fordert zwar (v.a. im sequentiellen Bereich) ihren Tribut. Dies wird aber durch die leichter zu beschreibende und z.T. sogar effizientere Parallelisierung schnell wieder wettgemacht. Vor allem braucht sich der Programmierer keine Mühe mit der expliziten Kodierung der Plazierung paralleler Module auf fremde Rechnerknoten und deren wechselseitige Kommunikation zu machen, da dies völlig transparent und automatisch von dem FASAN-Laufzeitsystem erledigt wird. Dadurch werden die Programme auch äußerst kurz.

Der zweischichtige Ansatz von FASAN, die in der Unterscheidung von Netzfunktionen und Elementarfunktionen ihren Ausdruck findet, verbessert die Übersichtlichkeit der Programme, da die numerischen Berechnungen klar von der parallelen Strukturierung getrennt werden. Durch das flexible Lokationskonzept zur Beschreibung der Lastverteilung lassen sich verschiedene Verteilungsstrategien durch Modifikation der verwendeten Lokationsfunktionen untersuchen, ohne den eigentlichen FASAN-Programmtext zu ändern. Auch dieser Teil ist streng gekapselt, was die Code-Entwicklung und die Fehlersuche erleichtert.

Leider mangelt es noch an geeigneten Debugging- und Visualisierungstools für FASAN, die eine große Hilfe darstellen würden. Außerdem könnte die Syntax vereinfacht werden, wenn auf FASAN-Ebene zumindest Zahlen und elementare Rechenoperatoren wie $+, -, *$ und $/$ erlaubt wären. Momentan müssen selbst für einfache Operationen wie das Halbieren einer Zahl noch eigene Elementarfunktionen geschrieben werden.

Im Grunde genommen könnte der FASAN-Compiler parallel ausführbare Funktionen automatisch erkennen, ohne daß der Programmierer dies im Text kenntlich macht. Würde man dann noch über einen vernünftigen, universell einsetzbaren Lastverteilungsmechanismus verfügen, so könnte die Parallelisierung völlig automatisch durchgeführt werden. Inwieweit dies sinnvoll ist, ist eine grundsätzliche Frage. Wie schon in Kapitel 7 angedeutet wurde, ist es unter Umständen in der Praxis gar nicht wünschenswert, die maximale Parallelität eines Algorithmus' auszuschöpfen. Zu viele parallele Tasks auf einem Rechenknoten erhöhen den Verwaltungsaufwand, zu starke Verflechtung der Kommunikationsbeziehungen zwischen den Rechenknoten überlastet das Kommunikationsmedium. Unserer Meinung nach wäre eine halbautomatische Lösung am günstigsten, in der der Compiler Vorschläge zur Parallelisierung unterbreitet und der Programmierer die Entscheidung trifft.

Einige Punkte mußten trotz oder gerade wegen des breiten Spektrums der Darstellung offenbleiben. So haben wir beispielsweise wenig über die Behandlung des Randes ausgesagt. Dies geschah aber vor allem aus Gründen der Übersichtlichkeit. Die entsprechende Theorie und die algorithmische Behandlung des Randes ist für die D-dimensionale Überschußschleuder bereits in [BZ92] beschrieben worden. Die Algorithmen können ohne weiteres auch in die funktionale FASAN-Beschreibung übernommen werden.

Ein weiterer wichtiger Punkt ist die ausführlichere Beschäftigung mit adaptiven Dünngitterprogrammen, für die wir hier nur Anstöße geben konnten. Auch hier ist die Beschreibung in FASAN prinzipiell möglich und wurde für zweidimensionale Quadraturprobleme schon implementiert (vgl. [Ebn94]). Für höhere Dimensionalität wird die Darstellung aber schon einigermaßen kompliziert. Die Frage der Lastverteilung adaptiver dünner Gitter müßte ebenfalls noch eingehender und systematischer behandelt werden.

Literaturverzeichnis

[AN90] Arvind und R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In *IEEE Transactions on Computers*, volume 39 (3), pages 143–152, March 1990.

[Aug84] L. Auguston. A compiler for lazy ML. In *Symposium on Lisp and Functional Programming*, pages 218–227, August 1984.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[BPZ96] M. Backschat, A. Pfaffinger und C. Zenger. Economic-Based Dynamic Load Distribution In Large Workstation Networks In P. Fraigniaud, L. Bouge, Y. Robert, A. Mignotte (Eds.): *Euro-Par '96 - Parallel Processing, Second International Euro-Par Conference, Lyon, France, 26. - 29.8. 1997, Proceedings Volume II*, Springer.

[Bal94] R. Balder. *Adaptive Verfahren für elliptische und parabolische Differentialgleichungen auf dünnen Gittern*. Dissertation, Institut für Informatik, TU München, Dezember 1994.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984.

[BDG$^+$94] A. Beguelin, J. Dongarra, A. Geist, J. Weicheng, R. Manchek und V. Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambrigde (MA) et. al., 1994.

[Bon95] T. Bonk. *Ein rekursiver Algorithmus zur adaptiven numerischen Quadratur mehrdimensionaler Funktionen*. Dissertation, Institut für Informatik, TU München, März 1995.

[Bun92] H. Bungartz. Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung. Dissertation, Institut für Informatik, Technische Universität München, 1992.

[BZ92] R. Balder und C. Zenger. The d-dimensional helmholtz equation on sparse grids. SFB Bericht 342/21/92 A, Institut für Informatik, TU München, 1992.

[CGH94] L. Clarke, I. Glendinning und R. Hempel. The MPI Message Passing Interface Standard. Technical report, March 94.

[DOSW95] J. J. Dongarra, St. W. Otto, M. Snir und D. Walker. An introduction to the MPI standard. Technical Report CS-95-274, University of Tennessee, January 1995.

[DR81] J. Darlington und M. Reeve. ALICE: A multiple-processor reduction machine for the parallel evaluation of applicative languages. In *Functional Programming and Computer Architecture*, pages 65–76, October 1981.

[Ebn94] R. Ebner. FASAN — Neuimplementierung einer funktionalen Sprache zur Parallelisierung numerischer Algorithmen. Diplomarbeit, Technische Universität München, 1994.

[EP96] R. Ebner und A. Pfaffinger. Transformation of Functional Programs into Data Flow Graphs Implemented with PVM. In *Proceedings of the Third Euro PVM Users' Group Meeting, 7–9 October 1996, Munich*, Berlin, 1996. Springer.

[FMS+95] J. T. Feo, P. J. Miller, S. Skedzielewski, S. M. Denton und C. J. Solomon. Sisal 90. Technical report, Lawrence Livermore National Laboratory, Livermore, California, 1995.

[GKW85] J. R. Gurd, C. C. Kirkham und I. Watson. The manchester prototype dataflow computer. In *Comm. ACM*, 28 (1), pages 34–52, January 1985.

[GMS93] R. G. Goldsmith, D. L. McBurney und M. R. Sleep. Parallel execution of Concurrent Clean on ZAPP. In Ronan Sleep, Rinus Plasmeijer und Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practise*, chapter 21. John Wiley and Sons Lts, 1993.

[Hah90] W. Hahn. Parallelisierung eines adaptiven hierarchischen Dünn-Gitter-Verfahrens. Diplomarbeit, Technische Universität München, 1990.

[HPJWe92] P. Hudak, S. L. Peyton Jones und P. Wadler (editors). Report on the programming language haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, Mar, 1992.

[Kra86] D. Kranz. Orbit: An optimizing compiler for Scheme. In *Proc. SIGPLAN '86 Symp. on Compiler Construction*, pages 219–233, June 1986.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures, Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, San Mateo, 1992.

[Mar92] W. S. Martins. Parallel Implementations of Functional Languages. In *Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages, Aachen*, 1992.

[McG85] J. McGraw. *SISAL: Streams and Iterations in a Single-Assignment Language: Reference Manual version 1,2*. Lawrence Livermore National Laboratory, Livermore, California., 1985. Manual M-146, Rev. 1.

[MTH90] R. Milner, M .Tofte und R .Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Pap94] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. PhD thesis, Labratory for Computer Science, August 1994.

[Pfa95] A. Pfaffinger. Parallel Communication On Workstation Networks with Complex Topologies. *Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen und Rechenstrukturen*, 1995.

[PJCSH87] S. L. Peyton Jones, C. Clack, J. Salkild und M. Hardie. GRIP - a high-performance architecture for parallel graph reduction. In *Functional Programming Languages and Computer Architecture*, number 274 in LNCS, pages 98–112, 1987.

[Rau92] H. Rauschmayer. Werkzeuge zur Bearbeitung funktionaler Programme auf Parallelrechnern. Diplomarbeit, Technische Universität München, 1992.

[Sch93] W. Schreiner. Parallel functional programming — an annotated bibliography. Technical Report 93-24, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1993.

[Thi94] P. Thiemann. *Grundlagen der funktionalen Programmierung*. B. G. Teubner, Stuttgart, 1994.

[Vei90] J. Veit. Entwurf und Implementierung eines Sprachkonzepts zur parallelen Konfiguration numerischer Aufgabenstellungen. Diplomarbeit, Technische Universität München, 1990.

[Zen91] C. Zenger. Sparse Grids. *Notes on Numerical Fluid Mechanics*, 31, 1991.

[Zen92] C. Zenger. Funktionale Programmierung paralleler Algorithmen für numerische und technisch-wissenschaftliche Anwendungen. In G. Bader, G. Wittum und R. Rannacher, editors, *GAMM-Seminar über numerische Algorithmen auf Transputer-Systemen, Heidelberg, 31.5.–1.6. 1991, Teubner-Skripten zur Numerik*. Teubner, Stuttgart, 1992.