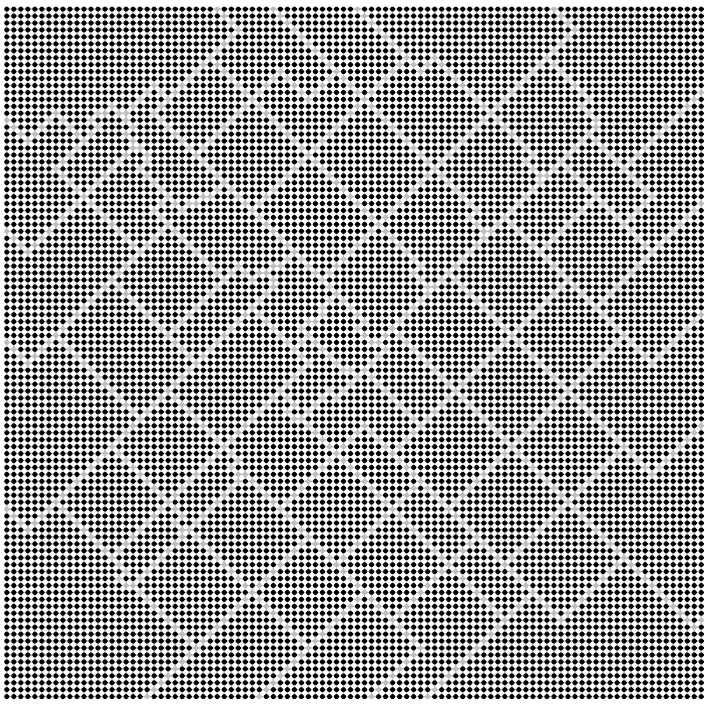
Faktorisierung dünn besetzter, positiv definiter Matrizen



Jürgen Schulze

Faktorisierung dünn besetzter, positiv definiter Matrizen

Dissertationsschrift

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)

vorgelegt im Fachbereich Mathematik/Informatik
der Universität Paderborn

von
Jürgen Schulze

Vorwort

Viele Freunde und Kollegen haben zum Gelingen dieser Arbeit beigetragen. An dieser Stelle möchte ich mich bei ihnen allen für ihre Hilfe und Unterstützung bedanken.

Mein besonderer Dank gilt Prof. Dr. Burkhard Monien für die Betreuung dieser Arbeit. Die Tätigkeit an seinem Lehrstuhl ermöglichte mir die Teilnahme an internationalen Konferenzen und die Beteiligung am aktuellen Forschungsgeschehen. Das hervorragende Arbeitsklima sowie Diskussionen mit Prof. Dr. Burkhard Monien trieben meine Arbeit voran und inspirierten mich.

Bei meinem ehemaligen Kollegen Dr. Ralf Diekmann bedanke ich mich für die vorbildliche Zusammenarbeit, konstruktive Kritik und für so manchen verschafften Durchblick. Vielen Dank auch an Cleve Ashcraft und Patrick Amestoy, die mit großem Eifer meine zahllosen E-Mails beantwortet haben. Insbesondere die enge Zusammenarbeit mit Cleve war sehr fruchtbar und hat mir viel Freude bereitet.

Des weiteren bedanke ich mich bei meinem lieben Kollegen Torsten Fahle für aufmerksames Korrekturlesen und das eine oder andere Dutzend hilfreiche Hinweise. Weiterer Dank gilt Stefan Blazy, Axel Keller, Robert Preis, Dr. Markus Röttger, Ulf-Peter Schroeder und Stefan Tschöke für Anregungen und Hilfestellungen.

Schließlich möchte ich mich bei meinen Eltern bedanken, die mich immer unterstützten und die meine Ausbildung ermöglichten. Herzlichen Dank!

Paderborn, im September 2000 Jürgen Schulze

Inhaltsverzeichnis

Kapitel 1

Einleitung

Mit dem Aufkommen elektronischer Rechenanlagen haben sich die an die numerische Mathematik gestellten Anforderungen entscheidend verändert. Ganz selbstverständlich verbindet man heute mit einem numerischen Verfahren auch einen Algorithmus, der über ein entsprechendes Programm auf einem Computer ausgeführt werden kann. Dieser enge Zusammenhang zwischen der Entwicklung numerischer Methoden und der Umsetzung in effiziente Algorithmen hat im Laufe der vergangenen Jahre zur Bildung einer neuen Disziplin innerhalb der numerischen Mathematik geführt, dem *Wissenschaftlichen Rechnen (scientific computing)*. Ein wichtiges Ziel des Wissenschaftlichen Rechnens besteht darin, für einen Computer „konstruktive Verfahren zu entwickeln und bereitzustellen, mit denen Aufgaben der angewandten Mathematik aus allen Bereichen der Naturwissenschaften und Technik erfolgreich und möglichst effizient bearbeitet und zahlenmäßig zu einer Lösung geführt werden können“ (vgl. [143], S. 1094). In der numerischen Praxis lassen sich viele dieser Aufgaben auf die folgenden drei Standardprobleme der linearen Algebra zurückführen:

Lösung linearer Gleichungssysteme Gesucht ist die Lösung eines linearen Gleichungssystems $Ax = b$, wobei A eine nichtsinguläre $n \times n$-Matrix darstellt und b einen aus n Einträgen bestehenden Vektor. Der Vektor b heißt auch *rechte Seite*. Die (eindeutige) Lösung des Gleichungssystems ist ein Vektor x mit n Einträgen. Dieser Vektor heißt *Lösungsvektor*.

Lösung linearer Ausgleichsprobleme Gesucht ist ein Vektor x, für den $\|Ax - b\|_2$ das Minimum annimmt. Dabei ist A eine $m \times n$-Matrix und b ein Vektor mit m Einträgen. Der Lösungsvektor x besteht aus n Einträgen. $\|\cdot\|_2$ bezeichnet die *euklidische Norm*. Im Falle $m > n$ heißt $Ax = b$ *überbestimmtes Gleichungssystem*. In der Regel existiert für ein solches Gleichungssystem keine Lösung.

Lösung des Eigenwertproblems Zu einer gegebenen $n \times n$-Matrix A ist ein aus n Einträgen bestehender Vektor $x \neq 0$ und ein Skalar λ gesucht, so daß $Ax = \lambda x$ gilt. Der Vektor x heißt *Eigenvektor* zum *Eigenwert* λ.

Die Lösung linearer Gleichungssysteme bildet den Kern vieler numerischer Anwendungen. Große Gleichungssysteme treten z. B. in den Ingenieurswissenschaften bei der Simulation von Flüssigkeits- oder Gasströmungen oder bei der Simulation des Verhaltens von Materialien wie Stahl, Beton, Holz usw. unter Belastung auf. Ganz grob unterscheidet man zwischen *direkten* und *iterativen* Lösungsverfahren (vgl. Abschnitt 1.1.1). Auf lineare Ausgleichsprobleme stößt man, wenn durch wissenschaftliche Experimente der Wert von Konstanten in empirischen Formeln bestimmt werden soll. Eigenwertprobleme findet man z. B. in der Physik bei der Analyse von Schwingungen. Eigenwerte spielen auch bei der Untersuchung des Konvergenzverhaltens iterativer Lösungsverfahren eine große Rolle.

In dieser Arbeit betrachten wir das erste Standardproblem der linearen Algebra. Dabei konzentrieren wir uns auf die Untersuchung und Entwicklung von sequentiellen und parallelen Algorithmen zur schnellen *Faktorisierung* großer, dünn besetzter, *positiv definiter* Matrizen mit reellen Koeffizienten. Eine symmetrische $n \times n$-Matrix A heißt positiv definit, falls für alle n-elementigen Vektoren $x \neq 0$ gilt, daß $x^T Ax$ echt größer als null ist. Positiv definite Matrizen besitzen eine Reihe interessanter Eigenschaften (siehe z. B. Stoer [137]). Insbesondere existiert für jede positiv definite Matrix A genau eine untere Dreiecksmatrix L mit positiven Diagonalelementen, so daß gilt $A = LL^T$. Die Matrix L heißt *Faktormatrix*. Ihre Berechnung stellt den aufwendigsten Schritt bei der direkten Lösung eines Gleichungssystems dar. Ist L bekannt, so erhält man x durch Lösen der *gestaffelten Gleichungssysteme* $Ly = b$ und $L^T x = y$.

Die Faktormatrix L kann mit Hilfe des *Cholesky-Verfahrens* berechnet werden. Wie alle direkten Verfahren zur Lösung eines linearen Gleichungssystems basiert auch die Methode von Cholesky auf einer sukzessiven Elimination der Unbekannten. Ist A dünn besetzt, so entstehen im Laufe des Eliminationsprozesses oftmals viele zusätzliche von null verschiedene Elemente. Diese Elemente heißen *Fill-in* oder *Auffüllung* von A. Ein wesentlicher Nachteil direkter Verfahren besteht nun gerade darin, daß durch die Auffüllung von A der zur Lösung des Gleichungssystems benötigte Rechen- und Speicheraufwand stark ansteigen kann. Durch eine „geschickte“ *Pivotwahl* ist es jedoch möglich, diesen Mehraufwand signifikant zu reduzieren.

Im Falle einer indefiniten Koeffizientenmatrix muß die Pivotwahl darüber hinaus die numerische Stabilität und die Durchführbarkeit des Verfahrens sicherstellen. In der Regel lassen sich nicht alle Ziele gleichermaßen erreichen, so daß die Aufrechterhaltung der numerischen Stabilität durch einen höheren Rechenaufwand erkauft wird. Ist die Matrix jedoch positiv definit, so ist das Verfahren ohne Beeinträchtigung der numerischen Stabilität für jede beliebige Pivotreihenfolge durchführbar. Man wählt deshalb die Reihenfolge so, daß die Auffüllung von A möglichst gering ist. Die Entwicklung verbesserter Algorithmen zur Bestimmung einer möglichst optimalen Pivotreihenfolge bildet einen Schwerpunkt dieser Arbeit.

Gelingt es die Auffüllung von A in Grenzen zu halten, so stellen direkte Lösungsverfahren eine interessante Alternative zu den in der numerischen Praxis oftmals bevorzugten iterativen Verfahren dar. Ob nun ein direktes Verfahren einem iterativen vorgezogen werden sollte, hängt

von vielen Faktoren ab und kann nicht eindeutig beantwortet werden. In Abschnitt 1.1 gehen wir auf diese viel diskutierte Frage näher ein und motivieren, warum wir uns auf Lösungsverfahren für positiv definite Gleichungssysteme konzentrieren. Daran anschließend wird in Abschnitt 1.2 der Aufbau dieser Arbeit beschrieben.

1.1 Motivation

Im Bereich des Wissenschaftlichen Rechnens spielt die Simulation komplexer physikalischer Systeme eine zentrale Rolle. Ein solches System ist beispielsweise durch einen Tragflächenflügel gegeben, der in einem Windkanal umströmt wird. Ziel der Simulation ist die Berechnung der dabei entstehenden Wirbel und Auftriebskräfte sowie die Bestimmung der Druckverteilung und Strömungsgeschwindigkeit in der Nähe der Flügeloberfläche.

Oftmals wird das physikalische System durch *partielle Differentialgleichungen* beschrieben, wobei zusätzliche Nebenbedingungen den Zustand des Systems am Rand und zu Beginn der Simulation festlegen. Aufgabe ist dann die Bestimmung einer Funktion des Ortes und der Zeit, die alle Differentialgleichungen und alle Nebenbedingungen erfüllt. Ist diese Funktion bekannt, so können die gesuchten physikalischen Größen für jeden beliebigen Ort des Grundgebiets berechnet werden. In der Regel ist es jedoch nicht möglich, eine solche geschlossene Funktion zu finden. Das kontinuierliche physikalische System muß dann in ein diskretes mathematisches Modell transformiert werden. Dies kann beispielsweise mit Hilfe eines *Differenzenverfahrens* geschehen.

Differenzenverfahren beruhen auf der Idee, die in den Differentialgleichungen auftretenden Ableitungen durch Differenzenquotienten zu approximieren. Hierfür ist ein Gitter notwendig, das vor Beginn der eigentlichen Simulation auf das Grundgebiet des physikalischen Systems projiziert wird. Die gesuchten physikalischen Größen können dann für jeden Ort des Grundgebiets, der mit einem Gitterknoten zusammenfällt, näherungsweise berechnet werden. In der Regel wird der Wert einer physikalischen Größe nur von den Werten an benachbarten Gitterknoten unmittelbar beeinflußt, so daß die Koeffizientenmatrix A sehr dünn besetzt ist. Des weiteren enthalten viele partielle Differentialgleichungen einen *Operator*, der zu einem Gleichungssystem mit positiv definiter Koeffizientenmatrix führt. Die positive Definitheit ermöglicht den Einsatz effizienter, iterativer Gleichungslöser. Aber auch direkte Lösungsverfahren profitieren von dieser Eigenschaft, da keine Pivotsuche zur Aufrechterhaltung der numerischen Stabilität durchgeführt werden muß.

Im folgenden werden wir die Vor- und Nachteile direkter und iterativer Lösungsverfahren diskutieren. Insbesondere stellen wir die vier grundlegenden Schritte zur direkten Lösung eines positiv definiten Gleichungssystems vor. Daran anschließend werden wir den Einsatz paralleler Rechner zur Lösung dünn besetzter Gleichungssysteme motivieren.

1.1.1 Direkte vs. iterative Lösungsverfahren

Iterative Verfahren erzeugen ausgehend von einem Startvektor x^0 eine Folge $x^1, x^2, \ldots$ von Vektoren, die gegen die gesuchte Lösung x eines Gleichungssystems konvergiert. In vielen Fällen ist der Rechenaufwand pro Iterationsschritt vergleichbar mit dem Aufwand zur Multiplikation von A mit einem Vektor. Klassische Iterationsverfahren wie z. B. das *Jacobi-Verfahren (Gesamtschrittverfahren)* oder das *Gauß-Seidel-Verfahren (Einzelschrittverfahren)* benutzen die Fixpunktiteration $x^{k+1} = Tx^k + c$, um ausgehend von x^0 den Lösungsvektor x zu approximieren. Beide Verfahren unterscheiden sich lediglich in der aus A abgeleiteten Iterationsmatrix T.

Für Gleichungssysteme mit positiv definiter Koeffizientenmatrix existieren weitaus effizientere Iterationsverfahren. An dieser Stelle sei lediglich das *cg-Verfahren (conjugate gradient method)* erwähnt. Es beruht darauf, daß die Lösung x des Gleichungssystems $Ax = b$ das Funktional $F(v) = \frac{1}{2}v^T Av + b^T v$ minimiert. Im Gegensatz zu den klassischen Iterationsverfahren, die zur exakten Berechnung des Lösungsvektors x unendlich viele arithmetische Operationen durchführen müssen, stoppt das cg-Verfahren bei rundungsfehlerfreier Rechnung nach höchstens n Schritten mit der exakten Lösung. In der numerischen Praxis kann jedoch von einer exakten Rechnung nicht ausgegangen werden, so daß eventuell zusätzliche Iterationen notwendig sind.

Die Geschwindigkeit, mit der die Fixpunktiteration gegen den Lösungsvektor x konvergiert, wird durch den betragsmäßig größten Eigenwert $|\lambda_{\max}(T)|$ der Iterationsmatrix T bestimmt. Dieser Wert heißt *Spektralradius* und wird mit $\rho(T)$ bezeichnet. Gilt $\rho(T) < 1$, so erzeugt die Fixpunktiteration für jeden beliebigen Startvektor x^0 eine zum Lösungsvektor x konvergente Folge (vgl. Schwarz [134] oder Stoer und Bulirsch [138]). Die entscheidende Größe für die Konvergenzgüte des cg-Verfahrens ist die *Kondition* der Matrix A. Diese wird mit $\kappa(A)$ bezeichnet und entspricht dem Quotienten $|\lambda_{\max}(A)|/|\lambda_{\min}(A)|$. Der enge Zusammenhang zwischen der Konvergenzgeschwindigkeit und den numerischen Eigenschaften der Matrix A bzw. T ist charakteristisch für alle Iterationsverfahren. Im Gegensatz dazu hängt die Effizienz eines direkten Lösungsverfahrens von der *Nichtnullstruktur* der Matrix A ab. Es sind also völlig unterschiedliche Eigenschaften der Koeffizientenmatrix, die die Effizienz eines iterativen und die eines direkten Lösungsverfahrens beeinflussen.

Der wesentliche Vorteil eines iterativen Lösungsverfahrens liegt in dem geringen Speicherplatzbedarf. Darüber hinaus ist es oftmals möglich, durch spezielle Techniken die Konvergenzgeschwindigkeit signifikant zu erhöhen. Beispielsweise kann in vielen Fällen die Kondition der Matrix A bzw. der Spektralradius der Iterationsmatrix T durch Multiplikation mit einer geeignet gewählten Matrix verbessert werden. In diesem Fall spricht man von einer *Vorkonditionierung*. Ferner kann die Korrektur der Vektorkomponenten beim Übergang von x^k nach x^{k+1} durch Einführung eines *Relaxationsparameters* ω verstärkt werden. Im Falle $\omega < 0$ spricht man von *Unter-* und im Falle $\omega > 0$ von *Überrelaxation*. Es ist jedoch zu bedenken, daß die optimale Wahl eines Relaxationsparameters bzw. einer Vorkonditionierungsmatrix selbst wieder von den numerischen Eigenschaften des zu lösenden Gleichungssystems abhängt. Zudem tauchen in der

numerischen Praxis immer wieder Matrizen auf, deren schlechte Kondition sich überhaupt nicht oder nur unwesentlich verbessern läßt.

Der Vorteil eines direkten Verfahrens besteht nun gerade darin, daß seine Effizienz nicht von den numerischen Eigenschaften der Matrix A abhängt. Gelingt es, den während der Faktorisierung entstehenden Fill-in zu begrenzen, so erreicht man nicht nur eine Beschleunigung des Verfahrens, sondern auch eine signifikante Reduzierung des Speicherbedarfs. Ist A positiv definit, so kann die Pivotreihenfolge und die Nichtnullstruktur von L vor Beginn der eigentlichen Faktorisierung berechnet werden. Die Faktorisierung von A wird dann nicht mehr durch Zeilenvertauschungen oder Speicherverwaltungsoperationen unterbrochen. Dies führt zu einer höheren *Cache-Effizienz* und damit zu einer weiteren Beschleunigung des direkten Verfahrens. Zur Lösung eines positiv definiten Gleichungssystems sind dann die folgenden Schritte notwendig (vgl. George und Liu [53]):

Ordering Berechne eine Pivotreihenfolge, so daß der Grad der Auffüllung von A möglichst gering ist. Vertausche die Zeilen und Spalten von A entsprechend der berechneten Reihenfolge. Bestimme dazu eine geeignete Permutationsmatrix P und bilde PAP^T.

Symbolische Faktorisierung Berechne die Nichtnullstruktur der Faktormatrix L von $PAP^T = LL^T$ und allokiere Speicherplatz für ihre Nichtnullelemente.

Numerische Faktorisierung Berechne die Nichtnullelemente von L.

Bestimmung des Lösungsvektors Löse die gestaffelten Gleichungssysteme $Lz = Pb$, $L^T u = z$ und $x = P^T u$.

Sind mehrere Gleichungssysteme mit identischer Koeffizientenmatrix aber verschiedenen rechten Seiten zu lösen, so müssen die ersten drei Schritte nur einmal durchgeführt werden. Demgegenüber ist bei einem iterativen Verfahren eine Wiederholung des gesamten Lösungsprozesses erforderlich. Der modulare Aufbau des direkten Lösungsverfahrens hat einen weiteren Vorteil: Im Bereich der Strukturmechanik besitzen viele Differentialgleichungen einen Operator, der innerhalb des Differenzenverfahrens zu Koeffizientenmatrizen führt, die sich lediglich in ihren numerischen Einträgen, nicht jedoch in ihrer Besetzungsstruktur unterscheiden. In diesem Fall muß nur einmal – nämlich zu Beginn der Simulation – ein Ordering berechnet werden.

Zusammenfassend läßt sich feststellen, daß sowohl iterative als auch direkte Lösungsverfahren ihre Vorteile besitzen. Der in der Literatur oftmals beklagte hohe Speicherbedarf direkter Verfahren kann durch moderne Ordering-Strategien zum Teil erheblich reduziert werden. Gelingt darüber hinaus eine auf die speziellen Eigenschaften der Rechnerarchitektur zugeschnittene Implementierung des Faktorisierungsalgorithmus, so erhält man ein effizientes, universell einsetzbares Lösungsverfahren.

1.1.2 Sequentielle vs. parallele Lösungsverfahren

In den vergangenen Jahren hat sich die Leistung der klassischen Einprozessorrechner im Durchschnitt alle 18 Monate verdoppelt (Gesetz von Moore). Die steigende Rechnerkapazität ermöglicht es, immer komplexere Aufgaben in Angriff zu nehmen. Dies gilt insbesondere für den Bereich des Wissenschaftlichen Rechnens wo heute Simulationen hochkomplexer physikalischer Systeme durchgeführt werden. Als Beispiel seien an dieser Stelle virtuelle Crash-Tests genannt, durch die sich erhebliche Einsparungspotentiale bei der Entwicklung eines neuen Automobils ergeben. Damit die Simulationen die Realität möglichst exakt widerspiegeln, müssen auch kleinste Details eines physikalischen Systems erfaßt werden. Dazu ist eine Diskretisierung mit einem sehr feinen Gitter erforderlich. Trotz Anwendung *adaptiver Diskretisierungsmethoden* entsteht so ein Bedarf an immer mehr Rechenleistung.

Die potentiell unbeschränkte Nachfrage nach Rechenkapazität kann von den klassischen Einprozessorrechnern nicht erfüllt werden. Daher hat der Einsatz paralleler Rechnersysteme in den letzten Jahren immer mehr an Bedeutung gewonnen. Verstärkt wurde diese Entwicklung durch den stetigen Preisverfall bei Halbleitern und Speicherelementen. Grundsätzlich können moderne Parallelrechner in drei Klassen eingeteilt werden: SMP-Systeme, verteilte Systeme und Vektorrechner. Ein SMP-System (symmetric multi-processor system) besteht aus mehreren Prozessoren, die über einen Bus oder Kreuzschienenschalter (crossbar switch) auf einen *gemeinsamen Speicher* zugreifen. Viele ursprünglich für Einprozessorrechner entwickelte Anwendungen lassen sich relativ einfach auf ein SMP-System portieren. Der Nachteil eines solchen Systems besteht jedoch darin, daß das Konzept des gemeinsamen Speichers physikalisch nicht beliebig skalierbar ist. In großen Parallelrechnern besitzt daher jeder Prozessor seinen eigenen lokalen Speicher. Der Datenaustausch in einem solchen verteilten System (distributed memory system) findet über ein *Verbindungsnetzwerk* statt. Entscheidend für die Kommunikationseffizienz ist dabei weniger die Anzahl der Kanten über die eine Nachricht gesendet wird *(Dilation)*, als vielmehr die Anzahl der Nachrichten, die zur selben Zeit über eine gemeinsame Kante geroutet werden *(Kongestion)*. In einem *Vektorrechner* spielt sich die Parallelität auf einer sehr viel niedrigeren Stufe ab. Diese Rechner besitzen spezielle Prozessoren, die dem *Pipelining-Prinzip* folgend verschiedene Phasen einer Operation mit verschiedenen Operanden gleichzeitig ausführen können. Heute findet man kaum noch reine Vektorrechner. Vielmehr werden verteilte oder SMP-Systeme mit Vektorprozessoren ausgestattet, um so eine weitere Leistungssteigerung zu erreichen. Darüber hinaus existieren Hybrid-Architekturen, bei denen mehrere kleinere SMP-Systeme über ein Netzwerk miteinander verbunden sind.

Die wachsende Bedeutung des parallelen Rechnens hat die Entwicklung skalierbarer Algorithmen zur Lösung dünn besetzter Gleichungssysteme stark forciert. In dieser Arbeit stellen wir einen parallelen Faktorisierungsalgorithmus vor, der auf dem Prinzip des *Nachrichtenaustausches (message passing)* beruht. Durch die Definition standardisierter Kommunikationsschnittstellen wie z. B. *MPI (message passing interface)* ist es prinzipiell möglich, diesen Algorithmus

auf verschiedenen Parallelrechner auszuführen. Dabei ist jedoch zu beachten, daß die von einem parallelen System zur Verfügung gestellte Rechenleistung nur dann voll genutzt werden kann, wenn ein Algorithmus bis ins Detail der jeweiligen Architektur angepaßt ist. Bei der Entwicklung des hier vorgestellten Faktorisierungsalgorithmus wurde ein verteiltes System zugrunde gelegt, wobei die Minimierung des *Kommunikations-Overheads* im Vordergrund stand.

Obwohl die Faktorisierung einer dünn besetzten Matrix ein größeres Parallelisierungspotential besitzt als die einer voll besetzten (vgl. Kapitel 5), ist die Entwicklung eines skalierbaren Faktorisierungsalgorithmus ungleich schwieriger. Bereits im sequentiellen Fall sind komplexere Datenstrukturen und Algorithmen erforderlich, um die dünne Struktur der Koeffizientenmatrix zu erhalten und auszunutzen. Darüber hinaus muß die Faktorisierung so organisiert werden, daß ein effektiver Einsatz der von modernen Computern bereitgestellten Caching-Mechanismen möglich ist. In einem verteilten System stellt sich darüber hinaus die Frage, welche Berechnungsschritte von welchen Prozessoren ausgeführt werden sollen. Die Zuordnung der Berechnungsschritte auf die Prozessoren und die damit verbundene Verteilung der Nichtnullelemente von A bzw. L hat einen entscheidenden Einfluß auf die Skalierbarkeit des Faktorisierungsalgorithmus.

Zusammenfassend läßt sich feststellen, daß bereits die Entwicklung eines effizienten sequentiellen Algorithmus zur Lösung dünn besetzter Gleichungssysteme hohe Anforderungen stellt. Da in der numerischen Praxis immer größere Gleichungssysteme gelöst werden müssen, ist eine Parallelisierung der komplexen sequentiellen Algorithmen unumgänglich. Die Lösung der hierbei zusätzlich entstehenden Optimierungsaufgaben stellt eine weitere Herausforderung dar.

1.2 Aufbau der Arbeit

In dieser Arbeit stellen wir Algorithmen zur Berechnung einer Pivotreihenfolge und Algorithmen zur Durchführung der symbolischen und numerischen Faktorisierung vor. Wir befassen uns also mit den ersten drei Schritten zur Lösung eines positiv definiten Gleichungssystems. Aufgrund ihrer hohen praktischen Bedeutung stehen Algorithmen zur Berechnung einer Pivotreihenfolge im Mittelpunkt dieser Arbeit.

In Kapitel 2 stellen wir zunächst zwei elementare Verfahren zur Berechnung der Faktormatrix L vor. Beide Verfahren sind numerisch äquivalent, d. h. sie führen die gleiche Anzahl von Multiplikations- und Additionsoperationen aus. Die Verfahren unterscheiden sich lediglich in der Reihenfolge der Operationen. Anschließend zeigen wir, wie der bei der Faktorisierung entstehende Fill-in graphentheoretisch beschrieben werden kann. Die grundlegende Idee besteht darin, die Auffüllung von A durch eine Folge von Eliminationsgraphen zu modelieren. Nahezu alle aus der Literatur bekannten Methoden zur Minimierung des Fill-in basieren auf dieser graphentheoretischen Beschreibung. Schließlich stellen wir die drei grundlegenden Ordering-Methoden vor. Es handelt sich um die Profil-, die Bottom-up- und die Top-down-Methode.

In Kapitel 3 untersuchen wir Ordering-Verfahren für Matrizen, deren Nichtnullstruktur einen

gitterförmigen Graphen induziert. Diese Matrizen spielen in der numerischen Praxis eine große Rolle. Wir präsentieren eine leichte Modifikation des bekannten Nested-Dissection-Verfahrens von George [48]. Durch diese Modifikation kann der bei der Faktorisierung eines $n \times n$-Gitters entstehende Fill-in um fast die Hälfte reduziert werden. Gleiches gilt für die Anzahl der Multiplikations- und Additionsoperationen, die zur Berechnung der Faktormatrix benötigt werden. Anhand einer genauen Analyse des modifizierten Nested-Dissection-Verfahrens zeigen wir, daß die Güte eines Orderings ganz entscheidend von der „Form“ der Gebiete abhängt, die im Laufe des Eliminationsprozesses entstehen.

In Kapitel 4 präsentieren wir ein neues Ordering-Verfahren für beliebige Graphen. Charakteristisch für das Verfahren ist eine enge Koppelung zwischen Bottom-up- und Top-down-Methoden. Dabei werden die im Rahmen eines Top-down-Verfahrens konstruierten Knotenseparatoren als Ränder der von einem unvollständigen Bottom-up-Ordering gebildeten Gebiete interpretiert. Die Motivation besteht darin, die Schwächen der einen Methode durch die Stärken der anderen auszuräumen. Dies geschieht in zwei Schritten: Zum einen benutzen wir Bottom-up-Techniken zur Konstruktion der Knotenseparatoren. Dazu entwickeln wir ein neuartiges Multilevel-Verfahren, bei dem spezielle Knotenauswahlstrategien zur Schrumpfung eines Graphen eingesetzt werden. Zum anderen benutzen wir die Knotenseparatoren als ein „Gerüst“ zur Generierung und Evaluierung eines weiten Spektrums von Bottom-up-Orderings. Aus diesem Spektrum kann dann das beste Ordering ausgewählt werden. Im Vergleich zu einem reinen Bottom-up-Algorithmus reduziert sich so die Anzahl der zur Berechnung von L benötigten Multiplikations- und Additionsoperationen um durchschnittlich 42 %. Da der Aufwand zur Berechnung von L den Aufwand zur Lösung eines Gleichungssystems dominiert, führt dies zu einer signifikanten Beschleunigung des direkten Lösungsverfahrens.

In Kapitel 5 beschreiben wir sequentielle und parallele Algorithmen zur Durchführung der symbolischen und numerischen Faktorisierung. Im sequentiellen Fall gehen wir insbesondere auf Techniken zur Steigerung der Cache- und Registereffizienz ein. Der von uns implementierte Faktorisierungsalgorithmus basiert auf der von Duff und Reid [35, 36] entwickelten Multifrontal-Methode. Um die von modernen Hochleistungsrechner bereitgestellte Floating-Point-Leistung voll nutzen zu können, benutzt der Faktorisierungsalgorithmus einen auf BLAS 3 Routinen basierenden numerischen Kern. Zur Parallelisierung unseres Faktorisierungsalgorithmus verwenden wir das von Gupta et al. [63, 66] vorgeschlagene zweidimensionale Mapping-Schema. Im parallelen Fall steht die Minimierung des Kommunikations-Overheads im Vordergrund. Anhand zweier Beispiele zeigen wir, daß die mit Hilfe des neuen Verfahrens berechneten Orderings auch für die parallele Faktorisierung geeignet sind und sehr gute Ergebnisse liefern.

In Kapitel 6 fassen wir die in dieser Arbeit vorgestellten Methoden zur effizienten Lösung dünn besetzter, positiv definiter Gleichungssysteme zusammen und diskutieren einige ungelöste Probleme.

Kapitel 2

Grundlagen

Wie bereits in der Einleitung erwähnt, beruhen alle direkten Verfahren zur Lösung eines linearen Gleichungssystems auf einer sukzessiven Elimination der Unbekannten. In seiner allgemeinsten Form wird dieser Eliminationsprozeß vom *Gaußschen Algorithmus* beschrieben. Angewandt auf eine nichtsinguläre Matrix A liefert der Algorithmus eine Dreieckszerlegung der Form $PA = LR$. Dabei ist P eine Permutationsmatrix, L eine untere Dreiecksmatrix mit Einsen in der Diagonale und R eine obere Dreiecksmatrix. Die Matrix P beschreibt die bei der Pivotwahl vorzunehmenden Zeilenvertauschungen. Ist A dünn besetzt, so muß die Pivotwahl nicht nur die Durchführbarkeit und numerische Stabilität des Verfahrens garantieren, sondern auch die Auffüllung von A in Grenzen halten. Diese Aufgaben lassen sich nur schwer vereinen, so daß die Aufrechterhaltung der numerischen Stabilität oftmals durch einen größeren Rechenaufwand erkauft werden muß.

Ist A jedoch symmetrisch und positiv definit, so existiert eine eindeutig bestimmte untere Dreiecksmatrix L mit positiven Diagonaleinträgen, für die gilt $A = LL^T$. Bei der Berechnung von L werden die Diagonalelemente einfach in ihrer natürlichen Reihenfolge als Pivots gewählt. Die positive Definitheit der Matrix A garantiert dabei die Durchführbarkeit und numerische Stabilität des Verfahrens. Im Falle einer dünn besetzten Koeffizientenmatrix steht jetzt „nur“ noch die Minimierung der Auffüllung von A im Vordergrund. Da für jede Permutationsmatrix P gilt, daß mit A auch PAP^T positiv definit ist, kann wie folgt vorgegangen werden: Bestimme eine Permutationsmatrix P, so daß die Auffüllung von PAP^T möglichst gering ist und berechne anschließend die Faktormatrix L von PAP^T.

Dieses Kapitel ist in drei Abschnitte gegliedert. In 2.1 werden zwei elementare Verfahren zur Berechnung der Faktormatrix L vorgestellt. Es handelt sich um das Cholesky-Verfahren und um eine Variante des Cholesky-Verfahrens. Angelehnt an die Ausführungen in Stoer [137] wird die enge Verwandtschaft beider Verfahren zum Gaußschen Algorithmus und zur direkten LR-Zerlegung nach Crout dargestellt. Abschnitt 2.2 zeigt, wie der bei der Faktorisierung entstehende Fill-in graphentheoretisch beschrieben werden kann. Viele Heuristiken zur Bestimmung einer

„guten“ Permutationsmatrix P benutzen diese graphentheoretische Beschreibung der Auffüllung von A. In der Regel basieren die Heuristiken auf einer der in Abschnitt 2.3 vorgestellten Methoden. Es handelt sich dabei um die *Profil-*, die *Bottom-up-* und die *Top-down-Methode*.

2.1 Gauß-Elimination und Dreieckszerlegung von Matrizen

Im folgenden stellen wir den Gauß-Algorithmus, die direkte LR-Zerlegung nach Crout, das Cholesky-Verfahren und eine Variante des Cholesky-Verfahrens vor. Zwischen allen vier Verfahren gibt es zahlreiche verwandtschaftliche Beziehungen. So unterscheiden sich beispielsweise die ersten und die letzten beiden lediglich in der Reihenfolge der Multiplikations- und Additionsoperationen, nicht jedoch in ihrer Anzahl. Des weiteren kann die Cholesky-Zerlegung als eine auf symmetrische, positiv definite Matrizen zugeschnittene LR-Zerlegung interpretiert werden.

Im folgenden sei angenommen, daß A eine reelle, nichtsinguläre $n \times n$-Matrix ist und $b \in I\!R^n$ ein Vektor. Mit Hilfe des Gaußschen Eliminationsverfahrens kann ein lineares Gleichungssystem

$$Ax = b, \qquad A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \tag{2.1}$$

gelöst werden. Dazu wird (2.1) durch geeignete Zeilenvertauschungen und Linearkombinationen von Zeilen schrittweise in ein gestaffeltes Gleichungssystem der Form

$$Rx = b', \qquad R = \begin{pmatrix} r_{11} & \dots & r_{1n} \\ & \ddots & \vdots \\ 0 & & r_{nn} \end{pmatrix}$$

transformiert, welches dieselben Lösungen besitzt. Da R eine obere Dreiecksmatrix ist mit $r_{ii} \neq 0$ für $i = 1, \dots, n$, kann das gestaffelte Gleichungssystem leicht gelöst werden.

Der Einfachheit halber werden beim Gaußschen Eliminationsverfahren die Zeilenvertauschungen und Linearkombinationen von Zeilen nicht an den Gleichungen (2.1) durchgeführt, sondern an der um den Vektor b erweiterten Koeffizientenmatrix

$$(A, b) = \begin{pmatrix} a_{11} & \dots & a_{1n} & b_1 \\ \vdots & & \vdots & \vdots \\ a_{n1} & \dots & a_{nn} & b_n \end{pmatrix}.$$

Man erhält dann eine Kette von Matrizen

$$(A, b) := (A^{(0)}, b^{(0)}) \to (A^{(1)}, b^{(1)}) \to \dots \to (A^{(n-1)}, b^{(n-1)}) =: (R, b'),$$

wobei der Übergang $(A^{(k-1)}, b^{(k-1)}) \to (A^{(k)}, b^{(k)})$ für $k = 1, \dots, n-1$ formal wie folgt beschrieben werden kann:

(1) Bestimme ein Element $a_{rk}^{(k-1)} \neq 0$, $r \in \{k, \ldots, n\}$, und vertausche die Zeilen r und k von $(A^{(k-1)}, b^{(k-1)})$. Ein solches Element existiert, da A nichtsingulär ist. Sei $(\tilde{A}^{(k-1)}, \tilde{b}^{(k-1)})$ die resultierende Matrix.

(2) Für $i = k+1, \ldots, n$ setze $l_{ik} := \frac{\tilde{a}_{ik}^{(k-1)}}{\tilde{a}_{kk}^{(k-1)}}$ und subtrahiere das l_{ik}-fache der k-ten Zeile von Zeile i. Die resultierende Matrix ist $(A^{(k)}, b^{(k)})$.

Das in Schritt (1) bestimmte Element heißt *Pivotelement* und sollte aus Gründen der numerischen Stabilität sorgfältig gewählt werden (vgl. Stoer [137]). Die Subtraktionen in Schritt (2) bewirken, daß in Spalte k von $A^{(k)}$ alle Elemente unterhalb der Diagonale zu null werden. Die Variable x_k tritt also in den entsprechenden Gleichungen nicht mehr auf; daher der Name Eliminationsverfahren. Man kann den Übergang von $(A^{(k-1)}, b^{(k-1)})$ nach $(A^{(k)}, b^{(k)})$ auch mit Hilfe von Matrizenmultiplikationen beschreiben. Sei $P_k = (e_1, \ldots, e_{k-1}, e_r, e_{k+1}, \ldots, e_{r-1}, e_k, e_{r+1}, \ldots, e_n)$ wobei e_i den i-ten *Einheitsvektor* darstellt. Weiter sei L_k eine untere Dreiecksmatrix der Form

$$L_k = \begin{pmatrix} 1 & & & & & 0 \\ & \ddots & & & & \\ & & 1 & & & \\ & & -l_{k+1,k} & 1 & & \\ & & \vdots & & \ddots & \\ 0 & & -l_{nk} & & & 1 \end{pmatrix}.$$

Dann gilt $(A^{(k)}, b^{(k)}) = L_k P_k (A^{(k-1)}, b^{(k-1)})$. Unter der vereinfachenden Annahme, daß keine Zeilenvertauschungen im Laufe des Eliminationsprozesses vorgenommen werden, erhält man

$$R = L_{n-1} \cdots L_1 A.$$

Da L_k nichtsingulär ist, kann die Gleichung umgeformt werden zu

$$L_1^{-1} \cdots L_{n-1}^{-1} R = A$$

mit

$$L_1^{-1} \cdots L_{n-1}^{-1} = \begin{pmatrix} 1 & & & & 0 \\ l_{21} & 1 & & & \\ \vdots & & \ddots & & \\ l_{n-1,1} & & & 1 & \\ l_{n1} & l_{n2} & \ldots & l_{n,n-1} & 1 \end{pmatrix} =: L.$$

Als Ergebnis erhält man eine Dreieckszerlegung $A = LR$. Diese Dreieckszerlegung läßt sich auch direkt, ohne Bildung der Matrizen $A^{(1)}, \ldots, A^{(n-1)}$, berechnen. Dazu betrachtet man die

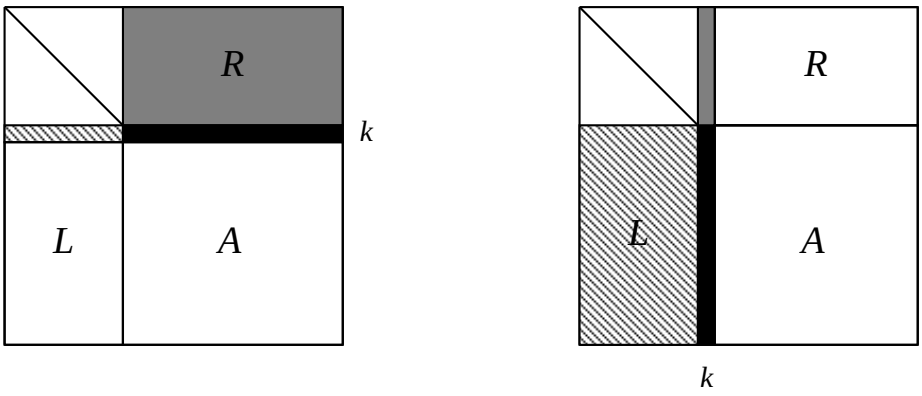


Abb. 2.1: Bei der Berechnung der k-ten Zeile von R (links) bzw. der k-ten Spalte von L (rechts) wird auf die grau dargestellten Elemente aus R und die schraffiert eingezeichneten Elemente aus L zugegriffen. Die zu berechnenden Elemente sind schwarz dargestellt.

Variablen l_{ij}, $i \geq j$ $(l_{ii} = 1)$, und r_{ij}, $j \geq i$, als Unbekannte, die mit Hilfe der n^2 Gleichungen

$$a_{ij} = \sum_{s=1}^{\min(i,j)} l_{is} r_{sj} \qquad (l_{ii} = 1)$$

bestimmt werden. Die Elemente l_{ij} können spaltenweise und die Elemente r_{ij} zeilenweise berechnet werden. Man setzt dazu nacheinander für $k = 1, \ldots, n$

$$\begin{aligned} r_{ki} &= a_{ki} - \sum_{j=1}^{k-1} l_{kj} r_{ji}, & i &= k, \ldots, n, \\ l_{ik} &= (a_{ik} - \sum_{j=1}^{k-1} l_{ij} r_{jk}) / r_{kk}, & i &= k+1, \ldots, n. \end{aligned} \tag{2.2}$$

In Schritt k wird also zuerst die k-te Zeile von R und anschließend die k-te Spalte von L (ohne l_{kk}) berechnet. Führt man die Dreieckszerlegung auf einem Computer durch, so werden die Elemente aus A nach und nach durch die Elemente r_{ki} und l_{ik} überschrieben. Abbildung 2.1 zeigt, auf welche bereits berechneten Elemente von L und R bei der Berechnung der k-ten Zeile von R (links) und der k-ten Spalte von L (rechts) zugegriffen wird. Die oben beschriebene Reihenfolge, in der die Elemente aus L und R berechnet werden, wurde ursprünglich von Crout vorgeschlagen. Nach Banachiewicz ist auch eine zeilenweise Berechnung von L und R möglich (vgl. Wilkinson [141]).

Die direkte Dreieckszerlegung nach Crout (2.2) unterscheidet sich von der Gauß-Elimination lediglich in der Reihenfolge – nicht jedoch in der Anzahl – der benötigten Operationen. Nach der Berechnung der k-ten Zeile von R und der k-ten Spalte von L können die Elemente von $A^{(k)}$ wie folgt berechnet werden:

$$a_{ij}^{(k)} = a_{ij} - \sum_{s=1}^{k} l_{is} r_{sj}, \qquad i, j = k+1, \ldots, n.$$

Die Matrix $A^{(k)}$ dient also als Speicher von stückweise berechneten Skalarprodukten. Bei der direkten Dreieckszerlegung verzichtet man auf einen solchen Zwischenspeicher und berechnet die Skalarprodukte in einem Schritt.

Nicht jede Matrix besitzt eine Dreieckszerlegung der Form $A = LR$ wie das Beispiel

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

zeigt. In der Regel müssen zunächst die Zeilen und Spalten von A in geeigneter Weise permutiert werden. Man erhält dann eine Zerlegung der Gestalt $PA = LR$. Es gibt jedoch eine wichtige Klasse von Matrizen, bei denen man ohne Permutationsmatrix auskommt:

Satz 2.1 *Sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische, positiv definite Matrix. Dann gibt es genau eine (reelle) untere $n \times n$-Dreiecksmatrix L, $l_{ik} = 0$ für $i < k$, mit $l_{kk} > 0$, $k = 1, \ldots, n$, so daß gilt $A = LL^T$.*

Die Dreiecksmatrix L heißt Faktormatrix von A. Der Beweis des Satzes liefert zugleich ein Verfahren zur Berechnung von L. Aus diesem Grund werden wir den Beweis an dieser Stelle noch einmal wiederholen. Wir orientieren uns dabei an den Ausführungen von George und Liu [53].

Beweis: Der Beweis wird durch Induktion nach n geführt. Im Falle $n = 1$ besteht A aus einer (reellen) Zahl $d > 0$, die eindeutig in der Form $d = l_{11}l_{11}$, $l_{11} = +\sqrt{d}$, geschrieben werden kann. Sei nun angenommen, daß der Satz für alle symmetrischen, positiv definiten $(n-1)$-reihigen Matrizen gültig ist. Wir partitionieren A wie folgt:

$$A = \begin{pmatrix} d & x \\ x^T & B \end{pmatrix}.$$

Dabei ist B eine $(n-1)$-reihige Matrix, $x \in \mathbb{R}^{n-1}$ ein Vektor der Länge $n-1$ und $d \in \mathbb{R}$ ein Skalar. Wegen $e_1^T A e_1 = d$ gilt $d > 0$. Sei I_{n-1} die $(n-1)$-reihige *Einheitsmatrix*, dann läßt sich A darstellen als

$$A = \begin{pmatrix} \sqrt{d} & 0 \\ \frac{x}{\sqrt{d}} & I_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & B - \frac{xx^T}{d} \end{pmatrix} \begin{pmatrix} \sqrt{d} & \frac{x^T}{\sqrt{d}} \\ 0 & I_{n-1} \end{pmatrix}. \tag{2.3}$$

Die Matrix $B - \frac{xx^T}{d}$ ist nicht nur symmetrisch, sondern selbst wieder positiv definit, denn für einen Vektor $y \in \mathbb{R}^{n-1}$ der Länge $n-1$ gilt

$$(-\frac{y^T x}{d}, y^T) \begin{pmatrix} d & x \\ x^T & B \end{pmatrix} \begin{pmatrix} -\frac{y^T x}{d} \\ y^T \end{pmatrix} = y^T (B - \frac{xx^T}{d}) y$$

und damit $y^T(B - \frac{xx^T}{d})y > 0$. Nach Induktionsvoraussetzung besitzt die Matrix $B - \frac{xx^T}{d}$ eine eindeutige Faktormatrix $\bar{L}$. Die Matrix A kann daher geschrieben werden als

$$A = \begin{pmatrix} \sqrt{d} & 0 \\ \frac{x}{\sqrt{d}} & \bar{L} \end{pmatrix} \begin{pmatrix} \sqrt{d} & \frac{x^T}{\sqrt{d}} \\ 0 & \bar{L}^T \end{pmatrix} = LL^T.$$

Die Eindeutigkeit von L folgt aus der Eindeutigkeit von $\bar{L}$ und aus der Tatsache, daß d eindeutig als Quadrat von $+\sqrt{d}$ dargestellt werden kann (vgl. Induktionsanfang). ■

Der Vektor $(\sqrt{d}, \frac{x}{\sqrt{d}})^T$ entspricht der ersten Spalte der Faktormatrix L. Die zweite Spalte erhält man, indem (2.3) auf die Restmatrix $B - \frac{xx^T}{d}$ angewandt wird. Dies ist möglich, da $B - \frac{xx^T}{d}$ wieder symmetrisch und positiv definit ist. Die Matrix $B - \frac{xx^T}{d}$ übernimmt dabei – wie die Matrix $A^{(1)}$ bei der Gauß-Elimination – die Rolle des oben angesprochenen Zwischenspeichers für die Skalarprodukte.

Angelehnt an die LR-Zerlegung nach Crout (2.2) können auch hier die Variablen l_{kk} und l_{ik}, $i > k$, der Faktormatrix L direkt berechnet werden. Dazu benutzt man die Bestimmungsgleichungen ($k = 1, \ldots, n$)

$$\begin{aligned} a_{kk} &= \sum_{j=1}^{k} l_{kj}^2, \\ a_{ik} &= \sum_{j=1}^{k} l_{ij} l_{kj}, \qquad i = k+1, \ldots, n. \end{aligned}$$

Aufgelößt nach l_{kk} bzw. l_{ik} erhält man dann die Formeln

$$\begin{aligned} l_{kk} &= \sqrt{a_{kk} - \sum_{j=1}^{k-1} l_{kj}^2}, \\ l_{ik} &= (a_{ik} - \sum_{j=1}^{k-1} l_{ij} l_{kj})/l_{kk}, \qquad i = k+1, \ldots, n. \end{aligned} \tag{2.4}$$

Das Verfahren (2.4) geht zurück auf Cholesky [20]. Die Faktormatrix L wird deswegen auch *Cholesky-Faktor* genannt. In der Literatur bezeichnet man (2.4) als *Inner-Product-* und (2.3) als *Outer-Product-Form* der Cholesky-Zerlegung. Abbildung 2.2 zeigt für beide Verfahren, auf welche Elemente bei der Berechnung der k-ten Spalte von L jeweils zugegriffen wird. Dabei nehmen wir wieder an, daß die Koeffizienten von A nach und nach durch die Koeffizienten von L überschrieben werden. Beide Verfahren benötigen die gleiche Anzahl von Multiplikations- und Additionsoperationen. Bezeichne $\eta(L_{*,k})$ die Anzahl der Subdiagonalelemente $\neq 0$ in Spalte k von L. In L gibt es

$$\eta(L) = \sum_{k=1}^{n-1} \eta(L_{*,k}) \tag{2.5}$$

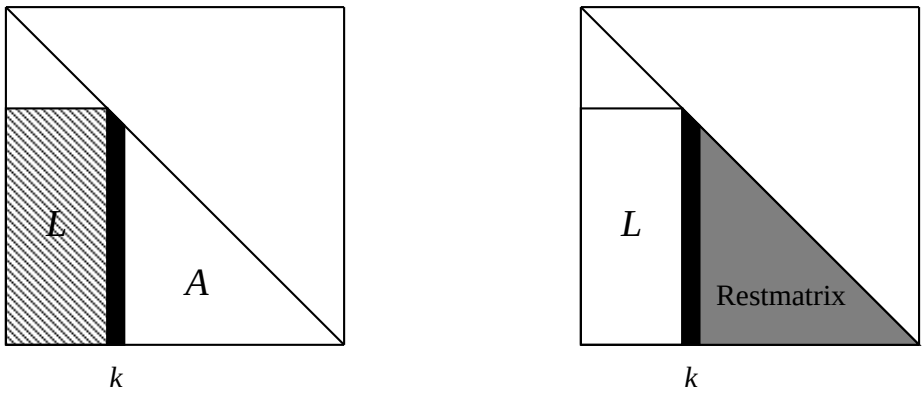


Abb. 2.2: Beim Cholesky-Verfahren (links) wird zur Berechnung der k-ten Spalte von L auf die schraffiert eingezeichneten Elemente von L zugegriffen. Beim Outer-Product-Verfahren (rechts) kann die k-te Spalte sofort berechnet werden. Anschließend werden die grau dargestellten Elemente der Restmatrix wie in (2.3) beschrieben modifiziert.

von null verschiedene Elemente unterhalb der Diagonale. Man kann leicht zeigen (vgl. George und Liu [53]), daß für die Berechnung des Cholesky-Faktors L

$$\frac{1}{2}\sum_{k=1}^{n-1}\eta(L_{*,k})(\eta(L_{*,k})+3) \tag{2.6}$$

Multiplikationsoperationen und

$$\frac{1}{2}\sum_{k=1}^{n-1}\eta(L_{*,k})(\eta(L_{*,k})+1) \tag{2.7}$$

Additionsoperationen durchgeführt werden müssen. Die Summe der für die Berechnung von L benötigten Multiplikations- und Additionsoperationen wird in der Literatur oft mit $\theta(L)$ bezeichnet. Ist die $n\times n$-Matrix A voll besetzt, so gilt $\eta(L_{*,k}) = n-k$. Für die Berechnung von L werden dann $\theta(L) = \frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n$ Multiplikations- und Additionsoperationen benötigt.

An dieser Stelle sei noch einmal darauf hingewiesen, daß bei der Pivotwahl auf die numerische Stabilität keine Rücksicht genommen werden muß. Nach (2.4) gilt nämlich:

$$|l_{kj}| \le \sqrt{a_{kk}}, \qquad j = 1,\ldots,k, \quad k = 1,\ldots,n.$$

Im Gegensatz zur Gauß-Elimination können also die bei der Cholesky-Zerlegung auftretenden Zahlen nicht zu groß werden (vgl. Wilkinson [141]).

2.2 Graphentheoretische Beschreibung der Cholesky-Zerlegung

Ist A eine dünn besetzte, positiv definite Matrix, so enthält der Cholesky-Faktor L in der Regel sehr viel mehr Elemente $\neq 0$ als A selbst. Der Grad der Auffüllung von A, und damit die Anzahl der für die Berechnung von L benötigten Operationen, hängt entscheidend von der Reihenfolge ab, in der die Diagonalelemente als Pivots gewählt werden. In diesem Abschnitt zeigen wir, wie die Entstehung von Fill-in bei der spaltenweisen Berechnung des Cholesky-Faktors nach dem Outer-Product-Verfahren durch die Konstruktion von *Eliminationsgraphen* modelliert werden kann. Viele Heuristiken zur Bestimmung einer „guten“ Permutationsmatrix P basieren auf dieser graphentheoretischen Beschreibung der Auffüllung von A.

Im folgenden benötigen wir einige Begriffe aus der Graphentheorie: Ein Graph G ist ein Tupel (V, E). Die Menge V beschreibt die Knoten und die Menge $E \subset V \times V$ die Kanten des Graphen. G heißt *ungerichtet*, falls mit $(u, v) \in E$, $u \neq v$, auch immer die Kante (v, u) in E enthalten ist. Man unterscheidet dann nicht mehr zwischen den Kanten (u, v) und (v, u). Ein ungerichteter Graph enthält keine *self-loops*, d. h. Kanten der Form (u, u).

Ist jedem Knoten $v \in V$ ein Gewicht $|v|$ zugeordnet, so heißt G *gewichtet*. Das Gewicht einer Knotenmenge $U \subset V$ ist $|U| = \sum_{u \in U} |u|$. In der Literatur bezeichnet man das Gewicht einer Menge U oftmals mit $\mathrm{weight}(U)$. Da wir in dieser Arbeit sowohl ungewichtete als auch gewichtete Graphen betrachten, benutzen wir zur Vereinfachung der Darstellung die Notation $|U|$. Gewichtete Graphen entstehen immer dann, wenn mehrere Knoten eines ungewichteten Graphen zu einem logischen Knoten verschmolzen werden. Standardmäßig betrachten wir alle Graphen als ungewichtet, d. h. $|v| = 1$ und $|U|$ entspricht der Kardinalität von U. In einem gewichteten Graphen bezeichnen wir die Kardinalität von U mit $\mathrm{card}(U)$.

Ist (u, v) eine Kante des Graphen, so heißen die Knoten u, v *adjazent* oder *benachbart* in G. Die Menge aller zu v adjazenten Knoten wird mit $\mathrm{adj}_G(v)$ bezeichnet und $\deg_G(v) = |\mathrm{adj}_G(v)|$ heißt *Grad* des Knotens v. Für $U \subset V$ ist die Menge der zu U adjazenten Knoten definiert als $\mathrm{adj}_G(U) = \bigcup_{u \in U} \mathrm{adj}_G(u) - U$.

Ein *(einfacher) Weg der Länge* k zwischen zwei Knoten $v, v' \in V$ ist eine Folge $v = u_0, u_1, \ldots, u_k = v'$ mit $(u_{i-1}, u_i) \in E$ für $i = 1, \ldots, k$, wobei jeder Knoten nur einmal angetroffen wird. Gilt $(u_k, u_0) \in E$, $k \geq 2$, so erhält man einen *Kreis der Länge* $k + 1$. Der Graph G heißt *zusammenhängend*, falls zwischen je zwei Knoten immer ein Weg existiert.

Der durch die Knotenmenge $U \subset V$ *induzierte Teilgraph* von G ist der Graph $G(U) = (U, E(U))$ mit $E(U) = E \cap (U \times U))$. Die Menge U heißt *(maximale) Zusammenhangskomponente*, falls $G(U)$ zusammenhängend ist und $\mathrm{adj}_G(U) = \emptyset$. Schließlich heißt eine Knotenmenge $C \subset V$ eines ungerichteten Graphen *Clique*, falls die Knoten in C paarweise durch eine Kante miteinander verbunden sind. Durch C wird der *vollständige* Teilgraph $G(C)$ induziert. Der Einfachheit halber schreiben wir $E(C) = C \times C$, wobei $E(C)$ jedoch keine self-loops enthält.

In dieser Arbeit betrachten wir nur ungerichtete Graphen. Jede symmetrische $n \times n$-Matrix A definiert einen ungerichteten Graphen $G = (V, E)$ mit n Knoten. Sei $V = \{v_1, \ldots, v_n\}$, dann ist $(v_i, v_j) \in E$ genau dann, wenn $a_{ij} \neq 0$. Jeder Spalte j, und damit auch jedem Diagonalelement a_{jj}, ist so ein Knoten v_j des Graphen G zugeordnet. Ist G zusammenhängend, so heißt A *irreduzibel*. Durch (2.3) wird die Matrix A transformiert in die Matrix $B - \frac{xx^T}{d} =: A_1$. Nach Parter [108] und Rose [118] erhält man den zu A_1 gehörenden Graphen G_1, indem G wie folgt modifiziert wird:

(1) Entferne Knoten v_1 zusammen mit allen inzidenten Kanten.

(2) Füge Kanten hinzu, so daß alle Knoten aus $\mathrm{adj}_G(v_1)$ eine Clique bilden.

Man sagt, daß G_1 durch Elimination des Knotens v_1 aus G entstanden ist. G_1 heißt deswegen auch Eliminationsgraph von G. Wir wollen die Konstruktion von G_1 an einem Beispiel veranschaulichen. Dazu betrachten wir die symmetrische Matrix

$$A = \begin{pmatrix} a_{11} & 0 & a_{31} & a_{41} & a_{51} & 0 \\ 0 & a_{22} & a_{32} & 0 & 0 & a_{62} \\ a_{31} & a_{32} & a_{33} & \mathbf{0} & \mathbf{0} & 0 \\ a_{41} & 0 & \mathbf{0} & a_{44} & a_{54} & 0 \\ a_{51} & 0 & \mathbf{0} & a_{54} & a_{55} & 0 \\ 0 & a_{62} & 0 & 0 & 0 & a_{66} \end{pmatrix}. \tag{2.8}$$

Wird A wie in (2.3) beschrieben partitioniert, so gilt $d = a_{11}$ und $x = (0, a_{31}, a_{41}, a_{51}, 0)^T$. Nach einer Iteration des Outer-Product-Verfahrens erhält man die Matrix

$$B - \frac{xx^T}{d} = \begin{pmatrix} a_{22} & a_{32} & 0 & 0 & a_{62} \\ a_{32} & a_{33} - \frac{a_{31}a_{31}}{d} & -\frac{\mathbf{a_{41}a_{31}}}{\mathbf{d}} & -\frac{\mathbf{a_{51}a_{31}}}{\mathbf{d}} & 0 \\ 0 & -\frac{\mathbf{a_{41}a_{31}}}{\mathbf{d}} & a_{44} - \frac{a_{41}a_{41}}{d} & a_{54} - \frac{a_{51}a_{41}}{d} & 0 \\ 0 & -\frac{\mathbf{a_{51}a_{31}}}{\mathbf{d}} & a_{54} - \frac{a_{51}a_{41}}{d} & a_{55} - \frac{a_{51}a_{51}}{d} & 0 \\ a_{62} & 0 & 0 & 0 & a_{66} \end{pmatrix} =: A_1.$$

Die in (2.8) in Fettdruck dargestellten Nullelemente von A werden in A_1 ersetzt durch die von null verschiedenen Elemente $-\frac{a_{41}a_{31}}{d}$ und $-\frac{a_{51}a_{31}}{d}$. Ganz allgemein gilt, daß nach der ersten Iteration des Outer-Product-Verfahrens ein Element a_{ij}, $i, j \in \{2, \ldots, n\}$, ersetzt wird durch $a_{ij} - \frac{a_{i1}a_{j1}}{d}$. Dabei entsteht Fill-in, falls $a_{ij} = 0$ und $a_{i1}, a_{j1} \neq 0$. Sieht man von dem entarteten Fall ab, daß $a_{ij} - \frac{a_{i1}a_{j1}}{d} = 0$ für $a_{ij} \neq 0$ (Fill-out), so gilt also für die Kantenmenge E_1 des Eliminationsgraphen G_1:

$$(v_i, v_j) \in E_1 \Leftrightarrow (v_i, v_j) \in E \quad \text{oder} \quad (v_i, v_1), (v_j, v_1) \in E.$$

Dies entspricht genau der Regel (2) von Parter und Rose. Abbildung 2.3 zeigt für A und A_1 die zugehörigen Graphen G und G_1.

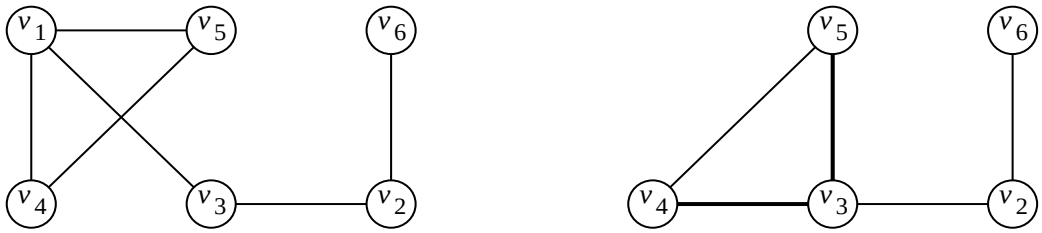


Abb. 2.3: Der aus der Matrix (2.8) abgeleitete Graph G (links) und der Eliminationsgraph G_1 (rechts). Die durch Elimination des Knotens v_1 entstandenen Fill-Kanten sind in Fettdruck dargestellt.

Durch die spaltenweise Berechnung von L nach dem Outer-Product-Verfahren, erhält man so eine Sequenz von Eliminationsgraphen G_k, $k = 1, \ldots, n$, $G_n = \emptyset$, wobei G_k aus G_{k-1} ($G_0 = G$) durch Elimination des Knotens v_k entsteht. Die dabei eingefügten Kanten entsprechen exakt dem Fill-in von A. Man nennt die eingefügten Kanten deswegen auch *Fill-Kanten*. Aufgrund der positiven Definitheit von A ist es möglich, die Diagonalelemente in jeder beliebigen Reihenfolge als Pivots zu wählen. Daher können auch die Knoten in G in jeder beliebigen Reihenfolge eliminiert werden. Eine solche Eliminationsreihenfolge wird mathematisch durch eine Permutation $\pi : V \mapsto \{1, \ldots, n\}$ beschrieben. Ist ein Knoten v_i mit der Zahl k numeriert, d. h. $\pi(v_i) = k$, so wird v_i beim Übergang von G_{k-1} nach G_k eliminiert. Die Permutation π heißt *Ordering*. Man ist nun bestrebt π so zu wählen, daß die Anzahl der entstehenden Fill-Kanten minimiert wird. Aus π kann die Permutationsmatrix P leicht abgeleitet werden. Man setzt dazu $P = (e_{\pi(v_1)}, \ldots, e_{\pi(v_n)})$.

Wir wollen das oben Gesagte an dem aus der Matrix (2.8) abgeleiteten Graphen G veranschaulichen (siehe auch Abbildung 2.3). Werden die Knoten in ihrer natürlichen Reihenfolge eliminiert, so entstehen die Fill-Kanten (v_3, v_4), (v_3, v_5) (Elimination von v_1), (v_3, v_6) (Elimination von v_2) und (v_4, v_6), (v_5, v_6) (Elimination von v_3). Setzt man $\pi(v_1) = 6$, $\pi(v_6) = 1$ und $\pi(v_k) = k$ für $k = 2, \ldots, 5$, so erhält man die Eliminationsreihenfolge $v_6, v_2, v_3, v_4, v_5, v_1$. Man vergewissert sich leicht, daß durch diese Reihenfolge keine Fill-Kante entsteht.

Durch π wird ein weiterer Graph definiert. Dieser heißt *aufgefüllter (filled) Graph* und wird mit $G_\pi = (V, E_\pi)$ bezeichnet. Die Menge E_π enthält neben den Kanten aus E alle Fill-Kanten die entstehen, wenn die Knoten des Graphen G in der durch π beschriebenen Reihenfolge eliminiert werden. Nach Rose et al. [119] stehen die Kantenmengen E und E_π wie folgt in Beziehung zueinander.

Lemma 2.1 ***(Rose et al. [119])***
Sei π eine Numerierung der Knoten von G. Dann ist $(v, v') \in E_\pi$ genau dann, wenn $(v, v') \in E$, oder es gibt einen Weg $v, u_1, \ldots, u_k, v'$ in G mit $\pi(u_i) < \min\{\pi(v), \pi(v')\}$ für alle $i = 1, \ldots, k$.

Zwei Knoten v, v' mit $(v, v') \notin E$ sind also in dem aufgefüllten Graphen G_π benachbart, falls es in dem ursprünglichen Graphen G einen Weg zwischen v und v' gibt, der ausschließlich über

Knoten führt, die vor v und v' eliminiert werden. Lemma 2.1 ist von grundlegender Bedeutung, da hierdurch die Fill-Kanten anhand von G und π bestimmt werden können. Eine explizite Bildung der Eliminationsgraphen ist nicht erforderlich.

Kennt man die Kantenmenge E_π, so kennt man auch die Nichtnullstruktur des Cholesky-Faktors L von PAP^T. Bezeichne $\mathrm{madj}_{G_\pi}(v)$ die Menge aller zu v adjazenten Knoten in G_π, die nach v eliminiert werden, also $\mathrm{madj}_{G_\pi}(v) = \{w \in V;\ w \in \mathrm{adj}_{G_\pi}(v) \text{ und } \pi(w) > \pi(v)\}$. Weiter sei $\mathrm{Struct}(L_{*,k}) = \{i > k;\ l_{ik} \neq 0\}$. Die Menge $\mathrm{Struct}(L_{*,k})$ enthält also die Zeilenindizes der von null verschiedenen Subdiagonalelemente in Spalte k von L. Dann gilt:

$$w \in \mathrm{madj}_{G_\pi}(v) \Leftrightarrow \pi(w) \in \mathrm{Struct}(L_{*,\pi(v)}). \tag{2.9}$$

Die Menge $\mathrm{madj}_{G_\pi}(v)$ heißt *monotone Adjazenz* des Knotens v in dem aufgefüllten Graphen G_π. Aus (2.9) folgt sofort

$$|\,\mathrm{madj}_{G_\pi}(v)| = \eta(L_{*,\pi(v)}). \tag{2.10}$$

Man kann daher $|\,\mathrm{madj}_{G_\pi}(\pi^{-1}(k))|$ anstelle von $\eta(L_{*,k})$ in die Formeln (2.5), (2.6) und (2.7) einsetzen und erhält so die Anzahl der Subdiagonalelemente $\neq 0$ im Cholesky-Faktor L von PAP^T und die Anzahl der zur Berechnung von L benötigten Operationen.

Der Graph G_π besitzt einige sehr interessante Eigenschaften, auf die wir in den nachfolgenden Kapiteln Bezug nehmen. Zur Beschreibung dieser Eigenschaften benötigen wir die folgenden Definitionen aus der Graphentheorie: Ein ungerichteter Graph $G = (V, E)$ heißt *chordal*, falls es in jedem Kreis der Länge > 3 zwei nicht aufeinanderfolgende Knoten u, v gibt mit $(u, v) \in E$. Die Kante (u, v) wird auch *Sehne* genannt. Eine Menge $S \subset V$ heißt *Knotenseparator*, falls der Teilgraph $G(V - S)$ nicht zusammenhängend ist. Seien a, b zwei nicht adjazente Knoten aus V. Ein Knotenseparator S heißt *a, b-Separator*, falls die Knoten a, b in $G(V - S)$ zu verschiedenen Zusammenhangskomponenten gehören. Falls keine echte Teilmenge von S ein a, b-Separator ist, so heißt S *minimaler a, b-Separator*. Ein *minimaler Knotenseparator* ist eine Knotenmenge $S \subset V$, die bezüglich zweier nicht adjazenter Knoten a, b einen minimalen a, b-Separator darstellt. Schließlich heißt ein Ordering π für die Knoten aus V *perfekt*, falls bei der Bildung der Eliminationsgraphen keine zusätzliche Kanten entstehen.

Nach Dirac [29] und Fulkerson, Gross [44] gilt:

Satz 2.2 ***(Dirac [29] und Fulkerson, Gross [44])***
Sei G ein ungerichteter Graph. Dann sind die folgenden Aussagen äquivalent:

(1) G ist chordal.

(2) G besitzt ein perfektes Ordering.

(3) Jeder minimale Knotenseparator in G bildet eine Clique.

Aufgrund der Konstruktion von G_π ist π ein perfektes Ordering für diesen Graphen. Nach Satz 2.2 ist G_π chordal, und jeder minimale Knotenseparator in G_π bildet eine Clique.

2.3 Klassische Ordering-Verfahren

Nach Yannakakis [142] ist die Bestimmung einer Permutationsmatrix P, so daß die Auffüllung von PAP^T minimal ist, ein $\mathcal{NP}$-vollständiges Problem. Zur Bestimmung einer möglichst guten Permutationsmatrix werden Heuristiken benutzt, deren Eingabe in der Regel aus dem ungerichteten Graphen G besteht. Die Heuristiken konstruieren dann für die Knoten des Graphen ein Ordering π, so daß der aufgefüllte Graph G_π möglichst wenige zusätzliche Kanten enthält. In diesem Abschnitt stellen wir kurz drei klassische Ordering-Methoden vor, auf denen viele Heuristiken basieren. Es handelt sich um die *Profil-*, die *Bottom-up-* und die *Top-down-Methode*.

2.3.1 Die Profil-Methode

Die Profil-Methode gehört zu den ältesten und am weitesten verbreiteten Ordering-Methoden. Ziel ist dabei, die Zeilen und Spalten der Matrix A so zu permutieren, daß die von null verschiedenen Elemente möglichst nah an der Diagonalen liegen. Auch für dieses Ordering-Problem ist die Bestimmung einer optimalen Permutationsmatrix wieder $\mathcal{NP}$-vollständig [107]. Zur Erläuterung der Profil-Methode benötigen wir einige zusätzliche Definitionen: Bezeichne $f_i(A)$ den Spaltenindex des ersten Elementes $\neq 0$ in Zeile i, also $f_i(A) = \min\{j;\ a_{ij} \neq 0\}$. Dann ist die *Bandweite* der i-ten Zeile von A, kurz $\beta_i(A)$, definiert durch $\beta_i(A) = i - f_i(A)$. Wegen $a_{ii} \neq 0$ gilt $f_i(A) \leq i$ und damit $\beta_i(A) \geq 0$. Das *Profil* von A besteht nun aus der Indexmenge $\mathrm{Profil}(A) = \{(i,j);\ f_i(A) \leq j < i\}$. Nach George und Liu [53] gilt $\mathrm{Profil}(A) = \mathrm{Profil}(L+L^T)$, so daß der Aufwand für die Berechnung von L entscheidend von der Größe des Profils, d. h. von $\sum_{i=1}^{n} \beta_i(A)$ abhängt.

Einer der bekanntesten Algorithmen zur Profilminimierung ist der Algorithmus von Cuthill und McKee [26]. Hierbei werden die Knoten des Graphen G mit Hilfe einer speziellen Breitensuche durchlaufen. Vor dem Start des Algorithmus wird die innerhalb der Breitensuche benutzte *Schlange (queue)* Q mit einem *pseudo-peripheren* Graphknoten initialisiert. Der Algorithmus arbeitet dann wie folgt: Sei u der erste Knoten in Q. Der Knoten u wird aus Q entfernt und als nächsten numeriert. Anschließend werden alle Nachbarn von u, die weder numeriert noch in Q gespeichert sind, aufsteigend sortiert nach ihrem Knotengrad in Q eingefügt. Der gesamte Prozeß wird so lange wiederholt bis Q leer ist.

Sei wieder π die Numerierung und P die aus π abgeleitete Permutationsmatrix. Durch die Vorgehensweise wird für benachbarte Knoten u, v mit $\pi(u) < \pi(v)$ die Differenz $\pi(v) - \pi(u)$ minimiert. Dies führt wiederum zu einer Minimierung der Bandweite von Zeile $\pi(v)$ in PAP^T, denn es gilt $\beta_{\pi(v)}(PAP^T) = \pi(v) - \min\{\pi(u);\ u \in \mathrm{adj}_G(v) \cup \{v\}\}$. In [99] beweisen Liu und Sherman, daß durch Umdrehen des Cuthill-McKee-Orderings das Profil niemals vergrößert, sondern in vielen Fällen sogar weiter verkleinert wird. Dieser Effekt wurde erstmals von George [47] beobachtet. Das so modifizierte Ordering heißt *Reverse-Cuthill-McKee-Ordering*.

Die Profil-Methode hat jedoch einen entscheidenden Nachteil. Gilt nämlich $f_i(PAP^T) < i$ für alle Zeilen $i = 2, \ldots, n$, so kann man leicht zeigen (vgl. George und Liu [53]), daß unabhängig von der Anzahl der Elemente $\neq 0$ im Profil von PAP^T das Profil von $L + L^T$ voll besetzt ist. Die Bedingung ist beispielsweise erfüllt, wenn A irreduzibel ist und P mit Hilfe des Cuthill-McKee-Algorithmus berechnet wird.

Obwohl die im nächsten Abschnitt vorgestellten Bottom-up- und Top-down-Methoden signifikant bessere Orderings produzieren, werden auch heute noch in vielen kommerziellen Anwendungen Algorithmen zur Profilminimierung eingesetzt. Dies liegt hauptsächlich daran, daß viele der in der numerischen Praxis auftretenden Matrizen bereits eine Bandstruktur besitzen. Da bei Verwendung eines Profil-Orderings die Bandstruktur während der Faktorisierung erhalten bleibt, können einfachere Datenstrukturen benutzt werden. Hierdurch reduziert sich der Aufwand bei der Implementierung eines numerischen Algorithmus erheblich.

Der Entwicklung verbesserter Algorithmen zur Profilminimierung wird daher auch heute noch viel Aufmerksamkeit geschenkt. Neben den klassischen Minimierungsverfahren basieren auch viele neuere Algorithmen auf einer Breitensuche. Zu den klassischen Verfahren gehören der oben beschriebenen Cuthill-McKee-Algorithmus sowie die Algorithmen von King [82], Gibbs-King [57] und Gibbs-Poole-Stockmeyer [58]. Zu den auf Breitensuche basierenden neueren Verfahren gehören der Sloan-Algorithmus [135] sowie die von Duff et al. [38] und Kumfert und Pothen [85] vorgestellten Erweiterungen des Sloan-Algorithmus. Weitere state-of-the-art Algorithmen basieren auf Spektral-Verfahren (vgl. Barnard et al. [18] und Paulino et al. [106]) oder Multilevel-Verfahren (vgl. Boman und Hendrickson [22]). In dieser Arbeit werden wir nicht näher auf die Profil-Methode eingehen.

2.3.2 Die Bottom-up- und die Top-down-Methode

Die Bottom-up-Methode benutzt die in Abschnitt 2.2 vorgestellten Eliminationsgraphen, um ein Ordering zu berechnen. Basierend auf dem aus A abgeleiteten Graphen G wird nach den Regeln von Parter und Rose eine Sequenz von Eliminationsgraphen G_k, $k = 1, \ldots, n$, $G_n = \emptyset$, generiert. Dabei ist entscheidend, nach welcher Vorschrift der beim Übergang von G_{k-1} nach G_k zu eliminierende Knoten ausgewählt wird. Zu den bekanntesten Ausprägungen der Bottom-up-Methode gehört der von Tinney und Walker [139] vorgestellte *Minimum-Degree-Algorithmus*. Hierbei wird aus G_{k-1} ein Knoten v mit minimalem Grad entfernt, damit die in G_k entstehende Clique möglichst klein ist. Der Minimum-Degree-Algorithmus geht zurück auf eine von Markowitz [102] vorgeschlagene Pivotsuche zur Lösung linearer Gleichungssysteme mit unsymmetrischer Koeffizientenmatrix.

Genaugenommen ist die Anzahl der in G_k einzufügenden Kanten nicht von der Größe der entstehenden Clique abhängig, sondern von der Anzahl der nicht durch eine Kante verbundenen Knoten in der Nachbarschaft des eliminierten Knotens. Man nennt diesen Wert die *Unzuläng-*

lichkeit (deficiency) des eliminierten Knotens. Formal ist die Unzulänglichkeit eines Knotens v definiert durch $\mathrm{def}_G(v) = |\{\{u,w\};\ u,w \in \mathrm{adj}_G(v), u \notin \mathrm{adj}_G(w)\}|$. Bei dem von Rose [118] vorgeschlagenen *Minimum-Deficiency-* oder *Minimum-Local-Fill-Algorithmus* wird aus G_{k-1} ein Knoten v entfernt, für den $\mathrm{def}_{G_{k-1}}(v)$ minimal ist. In Kapitel 4 werden wir noch einmal näher auf den Minimum-Degree- und den Minimum-Deficiency-Algorithmus eingehen und weitere Ausprägungen der Bottom-up-Methode vorstellen.

Bei der Bottom-up-Methode wird das Ordering π von „unten nach oben“ aufgebaut. Der beim Übergang von G_{k-1} nach G_k zu eliminierende Knoten wird dabei nach einem *lokalen Knotenauswahlverfahren* bestimmt. Im Gegensatz dazu baut die Top-down-Methode das Ordering π von „oben nach unten“ auf. Man bestimmt also zuerst diejenigen Knoten, die ganz zum Schluß eliminiert werden. Eine weit verbreitete Ausprägung der Top-down-Methode ist der Nested-Dissection-Algorithmus von George und Liu [51]. Der Algorithmus geht zurück auf ein von George [48] vorgestelltes Verfahren zur Numerierung spezieller quadratischer Gitter. Diese Gitter heißen *Gitter mit 9-Punkte-Stern* und werden in Kapitel 3 genauer betrachtet.

Der Nested-Dissection-Algorithmus von George und Liu ist ein rekursiver Algorithmus mit Parametern n_0 und α. Auf Eingabe eines ungerichteten Graphen $G = (V, E)$ und einer Zahl p arbeitet der Algorithmus wie folgt: Gilt $|V| < n_0$, so werden die Knoten aus V in beliebiger Reihenfolge von $p - |V| + 1$ bis p numeriert. Anschließend wird $p := p - |V|$ gesetzt. Im Falle $|V| \geq n_0$ wird ein Knotenseparator S bestimmt, durch dessen Entnahme G in zwei Teilgraphen $G(B)$ und $G(W)$ zerfällt mit $V = S \cup B \cup W$ und $|B|, |W| \leq \alpha|V|$, $0 < \alpha < 1$. Die Knoten aus S werden dann in beliebiger Reihenfolge von $p - |S| + 1$ bis p numeriert. Anschließend wird $p := p - |S|$ gesetzt und es erfolgt ein rekursiver Aufruf für jeden zusammenhängenden Teilgraphen von $G(B)$ und $G(W)$. Initial gilt $p = n$. Der Parameter n_0 steuert die Termination, und der Parameter α beeinflußt die Balance der generierten Partitionen.*

Wegen der rekursiven Struktur des Algorithmus können die Separatoren auf natürliche Art und Weise in *Ebenen* eingeteilt werden. Ein Separator S gehört zur Ebene i, wenn S in Rekursionsstufe i konstruiert wurde. Die Ebene null besteht aus dem initialen Separator. Eine Ebene i heißt *höhere* Ebene bezüglich einer Ebene j, falls $i < j$. Die Ebene null ist die *oberste* Ebene.

Abbildung 2.4 zeigt ein nach George [48] berechnetes Ordering für ein 7×7-Gitter mit 9-Punkte-Stern. Das Ordering kann auch mit Hilfe des Nested-Dissection-Algorithmus generiert werden. Dazu wird beim ersten Aufruf der aus den Knoten v_4, v_{11}, v_{18}, v_{25}, v_{32}, v_{39}, v_{46} bestehende Separator konstruiert. Die Knoten werden von 43 bis 49 numeriert. Nach Entnahme des Separators zerfällt der Graph in zwei Teile. Im linken Teil wählt man den Separator v_{22}, v_{23}, v_{24} und im rechten Teil den Separator v_{26}, v_{27}, v_{28}. Beide Separatoren gehören zur Ebene eins. Die Knoten des ersten Separators werden von 19 bis 22 numeriert, die des zweiten von 40 bis 42. Das Verfahren wird rekursiv fortgesetzt bis nur noch ein einzelner Knoten übrig bleibt

*In dieser Arbeit werden wir oft Partitionierungen durch eine *Färbung* der Knoten darstellen. In diesem Zusammenhang enthält B alle schwarz (black) und W alle weiß (white) gefärbten Knoten.

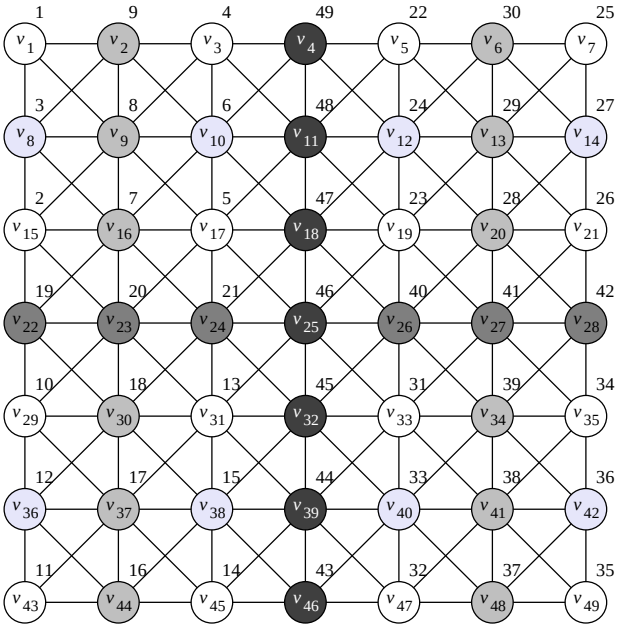


Abb. 2.4: Georges Nested-Dissection-Ordering für ein 7×7-Gitter mit 9-Punkte-Stern. Jeder Knoten v_i ist mit $\pi(v_i)$ beschriftet. Die Knoten aller zu einer Ebene gehörenden Separatoren sind im gleichen Grauton dargestellt. Je höher die Ebene, desto dunkler der Grauton.

($n_0 = 1$). Georges Ordering ist also ein spezielles, auf quadratische Gitter zugeschnittenes Nested-Dissection-Ordering. Man spricht daher auch von Georges Nested-Dissection-Ordering für quadratische Gitter.

Lipton et al. stellen in [90] eine leicht modifizierte Version des Nested-Dissection-Algorithmus von George und Liu vor. In ihrem *verallgemeinerten (generalized)* Nested-Dissection-Algorithmus werden aus $G(B)$ und $G(W)$ die Graphen $\tilde{G}(B) = (B \cup S, E(B \cup S) - E(S))$ und $\tilde{G}(W) = (W \cup S, E(W \cup S) - E(S))$ konstruiert. Der Algorithmus wird dann rekursiv für $\tilde{G}(B)$ und $\tilde{G}(W)$ aufgerufen. Es gibt hier also immer genau zwei rekursive Aufrufe, wobei die Separatorknoten in jeden Aufruf mit einbezogen werden. Sie werden jedoch kein zweites Mal numeriert. Erfüllt der aus einer $n \times n$-Matrix A abgeleitete Graph G ein $n^{1/2}$-*Separator-Theorem* [89], und wird der Graph nach dem verallgemeinerten Nested-Dissection-Ordering numeriert, so gilt für den entsprechenden Cholesky-Faktor L nach Lipton et al.: $\eta(L) = O(n \log n)$ und $\theta(L) = O(n^{3/2})$. Benutzt man hingegen zur Numerierung von G den Nested-Dissection-Algorithmus von George und Liu, so reicht die Existenz eines $n^{1/2}$-Separator-Theorems allein nicht aus, um einen Fill-in von höchstens $O(n \log n)$ zu garantieren. Ist jedoch zusätzlich G *planar* oder von *begrenztem Grad*, so gelten auch hier die obigen Schranken (vgl. Gilbert und Tarjan [61]). Der Nested-Dissection-Algorithmus von Lipton et al. spielt in der praktischen Anwendung kaum eine Rolle. Dies liegt hauptsächlich daran, daß der Algorithmus von George und Liu für die wichtige Klasse der planaren Graphen die gleichen Ergebnisse liefert. Hinzu kommt,

daß dieser Algorithmus sehr viel einfacher implementiert werden kann. Wenn wir im folgenden von einem Nested-Dissection-Ordering sprechen, so meinen wir immer ein nach dem Algorithmus von George und Liu konstruiertes Ordering.

In der Literatur wird das Nested-Dissection-Verfahren wie folgt motiviert: Da die Knoten der Teilgraphen $G(B)$ und $G(W)$ vor den Knoten des Separators S eliminiert werden, kann es nach Lemma 2.1 in G_π keine Kante geben, die einen Knoten aus $G(B)$ mit einem Knoten aus $G(W)$ verbindet. In dem entsprechenden Cholesky-Faktor L gibt es daher Blöcke, die nur aus Nullelementen bestehen. Dies wird durch den Umstand erkauft, daß die Knoten aus S in den meisten Fällen eine Clique in G_π bilden, was wiederum zu einem vollbesetzten Block in L führt. Dazu muß S noch nicht einmal ein minimaler Knotenseparator sein wie in Satz 2.2 verlangt. In der Tat sind die von einer Heuristik konstruierten Knotenseparatoren nur in wenigen Fällen minimale a, b-Separatoren. Nach Lemma 2.1 bildet jedoch S bereits dann eine Clique, wenn in $G(B)$ oder $G(W)$ eine Zusammenhangskomponente U existiert, so daß jeder Knoten aus S zu mindestens einem Knoten aus U adjazent ist. Diese Bedingung wird in der Regel von allen heuristisch berechneten Knotenseparatoren erfüllt.

Vor einer Implementierung der hier kurz beschriebenen Ausprägungen der Bottom-up- und der Top-down-Methode, müssen noch eine Reihe offener Fragen beantwortet werden. So stellt sich beispielsweise bei der Implementierung des Minimum-Degree-Algorithmus die Frage, welcher Knoten eliminiert werden soll, wenn mehrere Knoten den gleichen minimalen Grad besitzen. Die Güte eines Minimum-Degree-Orderings kann ganz entscheidend von einer solchen *Tie-Breaking-Strategie* abhängen (vgl. Abschnitt 3.3). Bei der Implementierung des Nested-Dissection-Algorithmus steht die Entwicklung eines effizienten Verfahrens zur Bestimmung möglichst kleiner Knotenseparatoren im Vordergrund. In den zwei folgenden Kapiteln werden wir im Detail auf diese offenen Fragen eingehen. Insbesondere werden wir, angeregt durch die Ergebnisse aus Kapitel 3, ein Ordering-Verfahren entwickeln, in dem die Bottom-up-Methode und das Nested-Dissection-Verfahren auf eine neue Art und Weise miteinander verknüpft sind.

Kapitel 3

Ordering-Verfahren für gitterförmige Graphen

Die in der numerischen Praxis auftretenden Gleichungssysteme besitzen oftmals eine Koeffizientenmatrix A, deren Nichtnullstruktur einen gitterförmigen Graphen G induziert. Dies ist beispielsweise bei der Lösung des *Dirichletschen Randwertproblems* auf einem offenen, quadratischen Gebiet Ω mittels finiter Differenzen der Fall (vgl. Frommer [43] oder Schwarz [134]). Zur Lösung des Randwertproblems ist eine Funktion ϱ gesucht, so daß für einen Punkt u aus Ω mit Koordinaten (x, y) gilt:

$$\begin{aligned} -\Delta\varrho(x,y) &= f(x,y), && \text{falls } u \text{ im Inneren von } \Omega \text{ liegt,} \\ \varrho(x,y) &= 0, && \text{falls } u \text{ auf dem Rand von } \Omega \text{ liegt.} \end{aligned} \tag{3.1}$$

Dabei bezeichnet das Symbol Δ den *Laplace-Operator*. Um ϱ numerisch zu approximieren, ersetzt man (3.1) mittels einer geeigneten Diskretisierung durch ein lineares Gleichungssystem. Beim Differenzenverfahren wird dazu ein $n \times n$-Gitter auf das quadratische Gebiet Ω projiziert (vgl. Abschnitt 1.1). Man löst jetzt (3.1) nur noch für solche Punkte aus Ω, die mit einem Knoten des Gitters zusammenfallen. Bezeichne (x_i, y_j) die Koordinaten desjenigen Punktes aus Ω, der mit dem Gitterknoten (i, j), $1 \leq i, j \leq n$, zusammenfällt. Bei einer Diskretisierung mit einem *5-Punkte-Stern* ist der Funktionswert $-\Delta\varrho(x_i, y_j)$ eines inneren Punktes (x_i, y_j), $1 < i, j, < n$, abhängig von den Werten $\varrho(x_{i-1}, y_j)$, $\varrho(x_{i+1}, y_j)$, $\varrho(x_i, x_{j-1})$ und $\varrho(x_i, y_{j+1})$. Die entsprechenden Gitterknoten liegen $+$-förmig um (i, j) und bilden zusammen mit (i, j) einen 5-Punkte-Stern. Wird zur Diskretisierung ein *9-Punkte-Stern* benutzt, so ist $-\Delta\varrho(x_i, y_j)$ zusätzlich abhängig von $\varrho(x_{i-1}, y_{j-1})$, $\varrho(x_{i+1}, y_{j-1})$, $\varrho(x_{i-1}, y_{j+1})$ und $\varrho(x_{i+1}, y_{j+1})$. Die zusätzlichen Gitterknoten liegen $\times$-förmig um (i, j). Alle Knoten zusammen bilden mit (i, j) einen 9-Punkte-Stern. In beiden Fällen kann basierend auf den Abhängigkeiten ein lineares Gleichungssystem aufgestellt werden. Dieses Gleichnungsystem enthält für jeden Knoten des Diskretisierungsgitters genau eine Gleichung (die sogenannte *Laplace-Gleichung*). Die Koeffizientenmatrix A ist also eine $n^2 \times n^2$-Matrix. Im ersten Fall induziert A ein zu dem Diskretisierungsgitter isomorphes $n \times n$-Gitter G

mit $V = \{(i,j);\ 1 \le i,j \le n\}$ und $E = \{((i_1,j_1),(i_2,j_2));\ (i_1,j_1),(i_2,j_2) \in V,\ |i_1-i_2|+|j_1-j_2| = 1\}$. Im zweiten Fall enthält G zusätzlich die Kanten $\{((i_1,j_1),(i_2,j_2));\ (i_1,j_1),(i_2,j_2) \in V,\ |i_1-i_2| = 1 \text{ und } |j_1-j_2| = 1\}$. Das erste Gitter heißt *Gitter mit 5-Punkte-Stern*, das zweite *Gitter mit 9-Punkte-Stern*.

In diesem Kapitel betrachten wir Ordering-Verfahren für $h \times n$-Gitter mit 5-Punkte- bzw. 9-Punkte-Stern. Das Kapitel ist wie folgt gegliedert: In Abschnitt 3.1 stellen wir einige wichtige, aus der Literatur bekannte Ergebnisse bezüglich der Numerierung gitterförmiger Graphen vor. Alle Aussagen gelten dabei sowohl für Gitter mit 5-Punkte-Stern als auch für Gitter mit 9-Punkte-Stern. In Abschnitt 3.2 präsentieren wir ein verbessertes Nested-Dissection-Ordering für quadratische Gitter mit 5-Punkte-Stern. Basierend auf einer genauen Analyse des verbesserten Orderings geben wir ein allgemeines Kriterium zur Charakterisierung eines guten Orderings an. In Abschnitt 3.3 zeigen wir, daß die gleichen Verbesserungen auch mit Hilfe eines Bottom-up-Orderings erreicht werden können. Wir benutzen dazu ein Minimum-Degree-Ordering mit einer speziellen Tie-Breaking-Strategie.

3.1 Literaturübersicht

Wird für die Numerierung eines $h \times n$-Gitters mit $h \le n$ ein Algorithmus zur Profilminimierung benutzt, so benötigt man für die Berechnung des Choleky-Faktors mindestens $h^3n + O(h^2n)$ Multiplikations- und Additionsoperationen. Der Choleky-Faktor enthält dabei mindestens $h^2n + O(hn)$ von null verschiedene Subdiagonalelemente (vgl. George und Liu [53]). Eine weitere Reduzierung des Fill-in bzw. der Zahl der benötigten Operationen kann – wenn überhaupt – nur mit komplexeren Ordering-Verfahren erreicht werden. Bei dem von George [49] vorgestellten *One-Way-Dissection-Verfahren* wird das $h \times n$-Gitter zunächst durch vertikale Separatoren in etwa gleich große Blöcke zerteilt. Danach werden die Knoten eines jeden Blocks zeilenweise numeriert. Zum Schluß werden die Separatorknoten numeriert. Man beginnt dazu mit den Knoten des am weitesten links stehenden Separators. Sind alle Knoten dieses Separators numeriert (die Reihenfolge spielt dabei keine Rolle), so fährt man mit den Knoten des rechts davon stehenden Separators fort. Auf diese Weise werden die Separatoren von links nach rechts wie bei einem Profil-Ordering durchnumeriert. George zeigt, daß in Abhängigkeit von den Dimensionen h und n die Zahl der Blöcke so gewählt werden kann, daß für die Berechnung des Cholesky-Faktors nur noch $\sqrt{56/3}\,h^{5/2}n + O(h^2n)$ Operationen notwendig sind.

Mit Hilfe des Nested-Dissection-Verfahrens kann die Zahl der benötigten Multiplikations- und Additionsoperationen weiter reduziert werden. Wir betrachten dazu zunächst ein quadratisches $n \times n$-Gitter. Wird dieses Gitter nach Georges Nested-Dissection-Verfahren numeriert, so benötigt man für die Berechnung des Cholesky-Faktors $829/42\,n^3 + O(n^2 \log n)$ Operationen. Der Cholesky-Faktor enthält dabei $31/4\,n^2 \log n + O(n^2)$ von null verschiedene Elemente (vgl. George und Liu [53]). Die Numerierung eines $h \times n$-Gitters mit $h < n$ kann nun auf die Nume-

rierung mehrerer quadratischer Gitter zurückgeführt werden. Im einfachsten Fall wird das $h \times n$-Gitter ($h \leq n/2$) zunächst rekursiv durch vertikale Separatoren in n/h quadratische Gitter mit Seitenlänge h geteilt. Anschließend werden in den quadratischen Gittern die Separatoren wie von George beschrieben konstruiert. Für die Berechnung des Cholesky-Faktors sind dann insgesamt $124/3\, h^2 n + O(h^3)$ Operationen notwendig. Rose und Whitten [120] beobachteten, daß sich der Aufwand zur Berechnung des Cholesky-Faktors weiter verringern läßt, wenn die obersten $n/h - 1$ vertikalen Sepratoren nicht entsprechend ihrer Rekursionstiefe, sondern einfach von links nach rechts wie beim One-Way-Dissection-Ordering durchnumeriert werden. Zur Berechnung des Cholesky-Faktors werden dann nur noch $112/3\, h^2 n + O(h^3)$ Operationen benötigt. Im Fall $h \leq n/2$ liefert also die Kombination mit einem Profil-Ordering bessere Ergebnisse als ein reines Nested-Dissection-Ordering. Wir werden auf diesen Effekt in Abschnitt 4.1.4 noch einmal genauer eingehen.

Bhat et al. zeigen in [19], daß durch eine geschickte Auslegung des $h \times n$-Gitters, $h < n$, mit quadratischen Gittern die Zahl der benötigten Operationen weiter reduziert werden kann. Auch sie benutzen einen Hybrid-Ansatz. Für die Numerierung der quadratischen Gitter wird Georges Nested-Dissection-Verfahren benutzt. Anschließend werden die verbleibenden Separatorknoten mit Hilfe eines Profil-Verfahrens numeriert. Bhat et al. zeigen, daß bei Verwendung ihres *Local-Nested-Dissection-Verfahrens* nur noch $829/30\, h^2 n - 829/105\, h^3 + O(hn \log h)$ Operationen für die Berechnung des Cholesky-Faktors notwendig sind. Setzt man in der Formel $h := n$, so erhält man interessanterweise wieder $829/42\, n^3 + O(n^2 \log n)$, also den gleichen Aufwand wie bei Georges Nested-Dissection-Verfahren für quadratische Gitter.

Nach Hoffman et al. [74] ist Georges Nested-Dissection-Ordering für quadratische Gitter asymptotisch optimal. Eliminiert man nämlich nacheinander die Knoten eines $n \times n$-Gitters, so tritt unweigerlich der Fall ein, daß zum ersten Mal eine Zeile oder Spalte des Gitters nur noch einen nicht eliminierten Knoten v enthält. Man kann nun zeigen, daß in dem entsprechenden Eliminationsgraphen der Knoten v zu mindestens $n - 1$ Knoten benachbart ist. Diese $n - 1$ Knoten bilden zusammen mit v eine Clique der Größe n in dem aufgefüllten Graphen. Für die Berechnung des Cholesky-Faktors sind demnach mindestens $O(n^3)$ Multiplikations- und Additionsoperationen notwendig.

Eine untere Schranke für den Fill-in kann wie folgt abgeleitet werden (vgl. wieder Hoffman et al. [74]): Jedes $n \times n$-Gitter enthält $(n-k+1)^2$ viele Teilgitter der Größe $k \times k$, $k = 2, \ldots, n$. Da der aufgefüllte Graph nach Satz 2.2 chordal ist, muß in jedem $k \times k$-Teilgitter eine Kante vorhanden sein, die zwei gegenüberliegende Seiten des Teilgitters verbindet. Man sagt, daß diese Kante das Teilgitter zerstört. Da eine solche Kante maximal k Teilgitter der Größe $k \times k$ zerstören kann, enthält der aufgefüllte Graph mindestens $\sum_{k=2}^{n} \frac{(n-k+1)^2}{k} \geq n^2 H_n - n^2$ Kanten, wobei H_n die n-te Harmonische Zahl ist. Bezeichne γ die Euler-Konstante ($\gamma = 0.57721\ldots$), dann gilt (vgl. z. B. Knuth [83]): $H_n = \ln n + \gamma + O(1/n)$. Damit gibt es in dem Cholesky-Faktor mindestens $O(n^2 \log n)$ von null verschiedene Elemente.

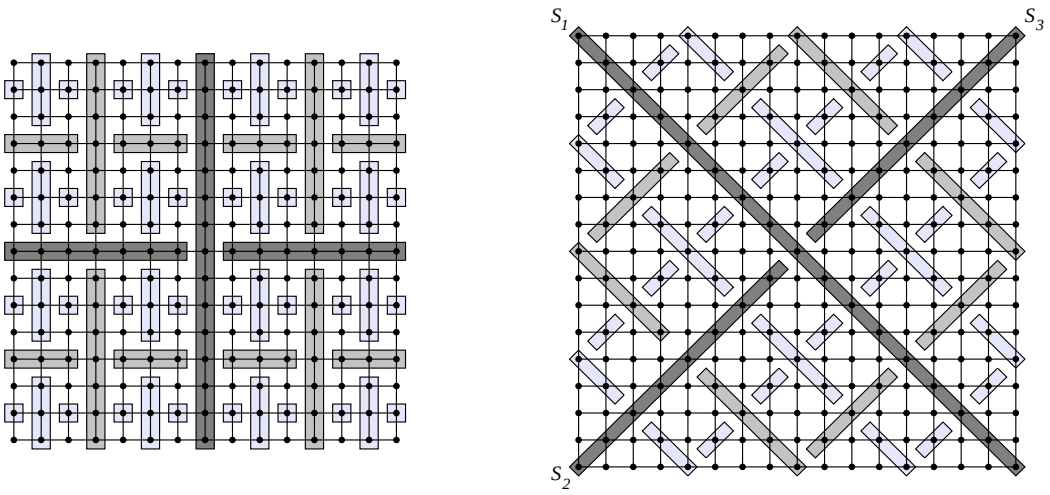


Abb. 3.1: Links die $+$-förmige Anordnung der Separatoren bei dem von George vorgeschlagenen Nested-Dissection-Ordering für ein quadratisches Gitter mit Seitenlänge $n = 2^l - 1$, $l = 4$. Rechts die $\times$-förmige Anordnung der Separatoren in dem modifizierten Nested-Dissection-Ordering. Die Seitenlänge des quadratischen Gitters beträgt hier $n = 2^l + 1$.

Benutzt man für die Numerierung eines $n \times n$-Gitters ein Minimum-Degree-Verfahren, so hängt die Güte des Orderings ganz entscheidend von der verwendeten Tie-Breaking-Strategie ab. Berman und Schnitger geben in [21] ein Minimum-Degree-Ordering an, durch das in dem Cholesky-Faktor ein Fill-in von $O(n^{2\log_3 4})$ erzeugt wird. Für die Berechnung des Cholesky-Faktors werden dabei $O(n^{3\log_3 4})$ Multiplikations- und Additionsoperationen benötigt. Bei einem Minimum-Degree-Ordering mit schlechter Tie-Breaking-Strategie kann also der Fill-in und die Zahl der benötigten Operationen asymptotisch höher sein als bei Georges Nested-Dissection-Ordering.

3.2 Ein verbessertes Nested-Dissection-Verfahren für quadratische Gitter

Die in Abbildung 3.1 (links) dargestellte $+$-förmige Anordnung der Separatoren in einem quadratischen Gitter ist charakteristisch für Georges Nested-Dissection-Ordering. Handelt es sich um ein Gitter mit 5-Punkte-Stern, so kann diese durch eine 45 Grad Drehung überführt werden in eine $\times$-förmige Anordnung (siehe Abbildung 3.1 (rechts)). Im folgenden zeigen wir, daß dadurch der Fill-in und die Zahl der benötigten Operationen in etwa halbiert wird.

Sei also G ein $n \times n$-Gitter mit 5-Punkte-Stern und $n = 2^l + 1$. Weiter sei P die aus dem modifizierten Nested-Dissection-Ordering abgeleitete Permutationsmatrix. Im folgenden wollen wir die Anzahl der Subdiagonalelemente $\neq 0$ im Cholesky-Faktor L von PAP^T und die Anzahl

der zur Faktorisierung benötigten Operationen genau berechnen. Als Hilfsmittel dient uns dabei das folgende Lemma (vgl. auch George [48]).

Lemma 3.1 *Sei π ein Nested-Dissection-Ordering der Knoten V des Graphen G und $S \subset V$ ein minimaler Knotenseparator. Sei weiter angenommen, daß eine Menge $R_S \subset V - S$ existiert mit $\mathrm{madj}_{G_\pi}(v) - S = R_S$ für alle $v \in S$. Es gelte $|S| = s$ und $|R_S| = r$. Dann beträgt die Anzahl der Subdiagonalelemente $\neq 0$ in den zu S gehörenden Spalten des Cholesky-Faktors*

$$f(s,r) = \frac{1}{2}s(s-1) + sr.$$

Zur Faktorisierung dieser Spalten werden

$$g(s,r) = \frac{1}{3}s^3 + \frac{1}{2}s^2 - \frac{5}{6}s + rs^2 + rs + r^2 s$$

Multiplikations- und Additionsoperationen benötigt.

Beweis: Sei $S = \{v_1, \ldots, v_s\}$. Da π ein Nested-Dissection-Ordering ist, können wir annehmen, daß $\pi(v_{i+1}) = \pi(v_i) + 1$ für $i = 1, \ldots, s-1$. Aufgrund der Minimalität von S bilden die Knoten $v_1, \ldots, v_s$ eine Clique in G_π (vgl. Satz 2.2) und es gilt: $\mathrm{madj}_{G_\pi}(v_i) = \{v_{i+1}, \ldots, v_s\} \cup R_S$ für $i = 1, \ldots, s$. Aus (2.10) folgt dann $\eta(L_{*,\pi(v_i)}) = s - i + r$. Die Anzahl der Subdiagonalelemente $\neq 0$ in den Spalten $\pi(v_1), \ldots, \pi(v_s)$ berechnet sich daher zu

$$\sum_{i=1}^{s} \eta(L_{*,\pi(v_i)}) = \sum_{i=1}^{s} s - i + r = f(s,r).$$

Nach (2.6) und (2.7) verursacht die Faktorisierung der Spalten $\pi(v_1), \ldots, \pi(v_s)$

$$\frac{1}{2}\sum_{i=1}^{s}(s-i+r)(s-i+r+3) + \frac{1}{2}\sum_{i=1}^{s}(s-i+r)(s-i+r+1) = g(s,r)$$

Multiplikations- und Additionsoperationen. ∎

Für das neue Nested-Dissection-Ordering werden zunächst die in Abbildung 3.1 (rechts) dargestellten Separatoren S_1, S_2 und S_3 konstruiert. S_1 ist der initiale Seprator (Ebene 0) und besteht aus $2^l + 1$ Knoten. Die Separatoren S_2 und S_3 (Ebene 1) bestehen aus jeweils 2^{l-1} Knoten. Alle drei Separatoren sind minimal. Da alle Knoten, die nicht zu S_1, S_2 oder S_3 gehören, vor den Knoten aus S_1, S_2, S_3 eliminiert werden, kann mit Hilfe von Lemma 2.1 leicht gezeigt werden, daß für einen Knoten $v \in S_2$ (bzw. $v \in S_3$) gilt $\mathrm{madj}_{G_\pi}(v) - S_2 = S_1$ ($\mathrm{madj}_{G_\pi}(v) - S_3 = S_1$). Des weiteren gilt $\mathrm{madj}_{G_\pi}(v) - S_1 = \emptyset$ für alle $v \in S_1$. Es folgt $R_{S_2} = R_{S_3} = S_1$ und $R_{S_1} = \emptyset$. Damit beträgt die Anzahl der Subdiagonalelemente $\neq 0$ in den letzten $2^{l+1} + 1$ Spalten nach Lemma 3.1:

$$\begin{aligned} &f(|S_2|, |S_1|) + f(|S_3|, |S_1|) + f(|S_1|, 0) \\ &= 2 \cdot f(2^{l-1}, 2^l + 1) + f(2^l + 1, 0) = \frac{7}{4}2^{2l} + 2^l. \end{aligned} \tag{3.2}$$

Zur Faktorisierung dieser Spalten werden

$$\frac{23}{12}2^{3l} + \frac{21}{4}2^{2l} + \frac{7}{3}2^{l} \tag{3.3}$$

Multiplikations- und Additionsoperationen benötigt. Nach Entnahme der Separatoren S_1, S_2 und S_3 entstehen vier zusammenhängende Teilgraphen in Form eines rechtwinkligen Dreiecks. Auf den Katheten eines solchen Dreiecks liegen jeweils 2^{l-1} und auf der Hypotenuse $2^l - 1$ Knoten. Jedes Dreieck wird durch die in den Ebenen zwei und drei konstruierten Separatoren in zwei kleinere Dreiecke und eine Raute zerteilt. Auf den Katheten der Dreiecke und den Seiten der Raute liegen jeweils 2^{l-2} Knoten. In den darauf folgenden Rekursionsstufen werden Dreiecke und Rauten durch Entnahme der $\times$-förmig angeordneten Separatoren in immer kleinere Dreiecke und Rauten zerteilt (vgl. Abbildung 3.1 (rechts)). Die Rekursion stoppt, wenn ein Dreieck aus nur noch vier und eine Raute aus nur noch fünf Knoten besteht.

Bezeichne $\mathfrak{D}_i$ ein Dreieck, auf dessen Katheten jeweils 2^i Knoten liegen. Weiter sei $\mathrm{sep}(\mathfrak{D}_i)$ die Menge der Knoten, durch dessen Entnahme $\mathfrak{D}_i$ in zwei kleinere Dreiecke und eine Raute zerfällt, wobei auf jeder Kathete eines kleineren Dreiecks und auf jeder Seite der Raute 2^{i-1} Knoten liegen. Wir wollen jetzt die Anzahl der Subdiagonalelemente $\neq 0$ in den zu $\mathrm{sep}(\mathfrak{D}_i)$ gehörenden Spalten und die Anzahl der zur Faktorisierung dieser Spalten benötigten Operationen berechnen. Abbildung 3.2 (links) zeigt ein Dreieck $\mathfrak{D}_i$ für $i = 3$. Die grau eingezeichneten Separatoren T_1 und T_2 bilden die Menge $\mathrm{sep}(\mathfrak{D}_i)$. Das Dreieck wird von den schraffiert eingezeichneten Knotenmengen U_1, U_2 und dem Knoten u umrandet. Diese Knoten gehören zu Separatoren, die in einer höheren Ebene konstruiert wurden. In $\mathfrak{D}_i$ werden zuerst die Knoten des Separators T_1 und anschließend die des Separators T_2 eliminiert. Beide Separatoren sind minimal. Mit $R_{T_1} = T_2 \cup U_1 \cup \{u\}$ und $R_{T_2} = U_1 \cup U_2 \cup \{u\}$ beträgt die Anzahl der von null verschiedenen Subdiagonalelemente in den zu $\mathrm{sep}(\mathfrak{D}_i)$ gehörenden Spalten nach Lemma 3.1:

$$f_{\triangle}(i) = f(|T_1|, |T_2| + |U_1| + 1) + f(|T_2|, |U_1| + |U_2| + 1).$$

In $\mathfrak{D}_i$ gilt nun: $|T_1| = 2^{i-1} - 1$, $|T_2| = 2^{i-1}$, $|U_1| = 3 \cdot 2^{i-1}$ und $|U_2| = 2^{i-1}$. Daraus folgt:

$$f_{\triangle}(i) = f(2^{i-1} - 1, 4 \cdot 2^{i-1} + 1) + f(2^{i-1}, 4 \cdot 2^{i-1} + 1) = \frac{9}{4}2^{2i} - 2 \cdot 2^{i}. \tag{3.4}$$

Zur Faktorisierung dieser Spalten werden

$$g_{\triangle}(i) = \frac{61}{12}2^{3i} + \frac{1}{2}2^{2i} - \frac{23}{6}2^{i} \tag{3.5}$$

Multiplikations- und Additionsoperationen benötigt. Bezeichne jetzt $\mathfrak{R}_i$ eine Raute, auf dessen Seiten jeweils 2^i Knoten liegen. Weiter sei $\mathrm{sep}(\mathfrak{R}_i)$ die Menge der Knoten, durch dessen Entnahme $\mathfrak{R}_i$ in vier kleinere Rauten zerfällt, wobei auf jeder Seite einer kleineren Raute 2^{i-1} Knoten liegen. Abbildung 3.2 (rechts) zeigt eine Raute $\mathfrak{R}_i$ für $i = 3$. Die grau eingezeichneten Separatoren T_3, T_4 und T_5 bilden die Menge $\mathrm{sep}(\mathfrak{R}_i)$. In der Raute werden zuerst die Knoten aus T_3, T_4

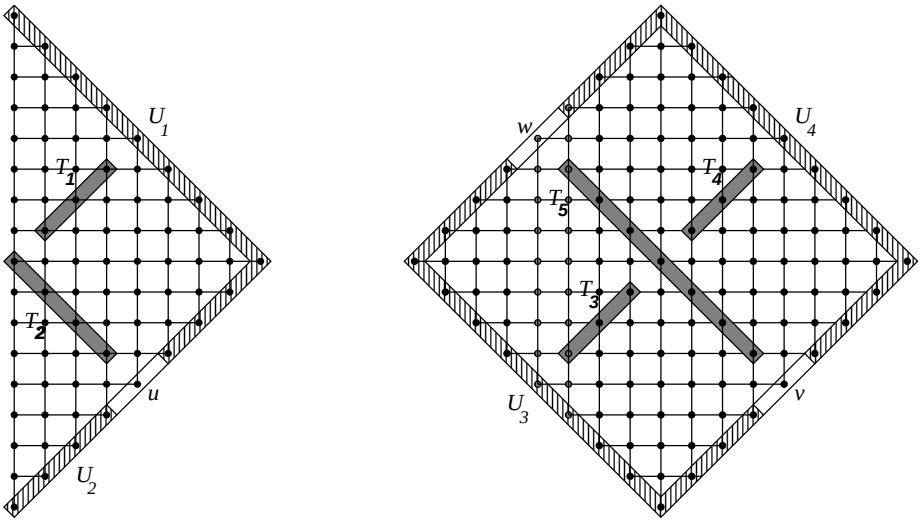


Abb. 3.2: Links ein Dreieck $\mathfrak{D}_i$, rechts eine Raute $\mathfrak{R}_i$, $i = 3$. Die Knotenmengen $\mathrm{sep}(\mathfrak{D}_i)$ bzw. $\mathrm{sep}(\mathfrak{R}_i)$ sind grau dargestellt. Dreieck und Raute werden von den schraffiert eingezeichneten Knotenmengen und dem Knoten u, bzw. den Knoten v, w umrandet.

und anschließend die Knoten aus T_5 eliminiert. Alle Separatoren sind minimal. Nach Lemma 3.1 beträgt die Anzahl der Subdiagonalelemente $\neq 0$ in den zu $\mathrm{sep}(\mathfrak{R}_i)$ gehörenden Spalten

$$f_\diamond(i) = f(|T_3|, |T_5| + |U_3| + 2) + f(|T_4|, |T_5| + |U_4| + 2) + f(|T_5|, |U_3| + |U_4| + 2).$$

In $\mathfrak{R}_i$ gilt: $|T_3| = |T_4| = 2^{i-1} - 1$, $|T_5| = 2^i - 1$ und $|U_3| = |U_4| = 4 \cdot 2^{i-1} - 1$. Daraus folgt:

$$f_\diamond(i) = 2 \cdot f(2^{i-1} - 1, 6 \cdot 2^{i-1}) + f(2^i - 1, 8 \cdot 2^{i-1}) = \frac{31}{4} 2^{2i} - 13 \cdot 2^i + 3. \tag{3.6}$$

Zur Faktorisierung dieser Spalten werden

$$g_\diamond(i) = \frac{371}{12} 2^{3i} - \frac{167}{4} 2^{2i} - \frac{5}{3} 2^i + 3 \tag{3.7}$$

Multiplikations- und Additionsoperationen benötigt. Im folgenden bezeichne a_i die Anzahl der Dreiecke, auf deren Katheten jeweils 2^{l-i} Knoten liegen und b_i die Anzahl der Rauten, auf deren Seiten jeweils 2^{l-i} Knoten liegen. Es gilt $a_1 = 4$ und $b_1 = 0$. Jedes Dreieck zerfällt durch Entnahme der in Abbildung 3.2 dargestellten Separatoren in zwei kleinere Dreiecke und eine Raute. Jede Raute wiederum zerfällt in vier kleinere Rauten. Daher gilt:

$$a_{i+1} = 2 \cdot a_i \quad \text{und} \quad b_{i+1} = 4 \cdot b_i + a_i.$$

Durch Induktion zeigt man leicht:

$$a_i = 2^{i+1} \quad \text{und} \quad b_i = \frac{1}{2} 2^{2i} - 2^i. \tag{3.8}$$

Damit gibt es $a_{l-1} = 2^l$ Dreiecke, die nur aus vier Knoten und $b_{l-1} = \frac{1}{8}2^{2l} - \frac{1}{2}2^l$ Rauten, die nur aus fünf Knoten bestehen. In einem solchen Dreieck bilden die drei Knoten auf den Katheten eine *unabhängige Menge*, d. h. sie sind nicht durch eine Kante miteinander verbunden. Gleiches gilt für die vier Knoten auf den Seiten einer Raute. In dem modifizierten Nested-Dissection-Ordering werden zuerst die Knoten der unabhängigen Menge eliminiert und danach der verbleibende Knoten. In den entsprechenden Spalten des Cholesky-Faktors gibt es

$$15 \cdot 2^l + 24 \cdot \left(\frac{1}{8}2^{2l} - \frac{1}{2}2^l\right) = 3 \cdot 2^{2l} + 3 \cdot 2^l \tag{3.9}$$

von null verschiedene Subdiagonalelemente. Zur Faktorisierung dieser Spalten werden

$$89 \cdot 2^l + 176 \cdot \left(\frac{1}{8}2^{2l} - \frac{1}{2}2^l\right) = 22 \cdot 2^{2l} + 2^l \tag{3.10}$$

Multiplikations- und Additionsoperationen benötigt. Mit Hilfe von (3.2), (3.4), (3.6), (3.8) und (3.9) kann nun die Gesamtzahl der von null verschiedenen Subdiagonalelemente in L berechnet werden. Es gilt:

$$\begin{aligned}\eta(L) &= \frac{7}{4}2^{2l} + 2^l + \sum_{i=1}^{l-2}\left(2^{i+1} \cdot f_\triangle(l-i) + \left(\frac{1}{2}2^{2i} - 2^i\right) \cdot f_\diamond(l-i)\right) + 3 \cdot 2^{2l} + 3 \cdot 2^l \\ &= \frac{31}{8}2^{2l}l - \frac{93}{8}2^{2l} + 11 \cdot 2^l l + \frac{31}{2}2^l + 4.\end{aligned}$$

Die Gesamtzahl der zur Berechnung von L benötigten Operationen beträgt nach (3.3), (3.5), (3.7), (3.8) und (3.10):

$$\begin{aligned}\theta(L) &= \frac{23}{12}2^{3l} + \frac{21}{4}2^{2l} + \frac{7}{3}2^l + \sum_{i=1}^{l-2}\left(2^{i+1} \cdot g_\triangle(l-i) + \left(\frac{1}{2}2^{2i} - 2^i\right) \cdot g_\diamond(l-i)\right) + 22 \cdot 2^{2l} + 2^l \\ &= \frac{631}{72}2^{3l} - \frac{167}{8}2^{2l}l + \frac{393}{8}2^{2l} - \frac{13}{6}2^l l - \frac{421}{18}2^l + 4.\end{aligned}$$

Wir haben damit gezeigt:

Satz 3.1 *Wird zur Numerierung eines $n \times n$-Gitter mit 5-Punkte-Stern das modifizierte Nested-Dissection-Verfahren benutzt, so benötigt man für die Berechnung des entsprechenden Cholesky-Faktors $631/72\, n^3 + O(n^2 \log n)$ Multiplikations- und Additionsoperationen. Der Cholesky-Faktor enthält dabei $31/8\, n^2 \log n + O(n^2)$ von null verschiedene Elemente.*

Tabelle 3.1 zeigt noch einmal einen Vergleich zwischen Georges Nested-Dissection-Ordering und dem modifizierten Nested-Dissection-Ordering. Werden die drei quadratischen Gitter mit Seitenlänge $n = 127$, $n = 255$ und $n = 511$ wie von George beschrieben numeriert, so ergeben sich für $\eta(L)$ und $\theta(L)$ die in den Spalten 2 und 3 angegebenen Werte. Demgegenüber zeigen die

$+$-förmige Separatoren			$\times$-förmige Separatoren		
n	$\eta(L)/10^3$	$\theta(L)/10^6$	n	$\eta(L)/10^3$	$\theta(L)/10^6$
127	439	33	129	266	17
255	2235	291	257	1296	139
511	10900	2461	513	6153	1140

Tab. 3.1: Vergleich von $\eta(L)$ (in Tsd.) und $\theta(L)$ (in Mio.) bei $+$-förmiger und bei $\times$-förmiger Anordnung der Separatoren. Im ersten Fall beträgt die Seitenlänge n der Gitter 127, 255 und 511, im zweiten Fall 129, 257 und 513.

Spalten 5 und 6 die Werte für $\eta(L)$ und $\theta(L)$, wenn zur Numerierung der quadratischen Gitter mit Seitenlänge $n = 129$, $n = 257$ und $n = 513$ das modifizierte Nested-Dissection-Verfahren benutzt wird. Die Werte für $\eta(L)$ sind in Tausend und die Werte für $\theta(L)$ in Millionen angegeben. Obwohl bei $\times$-förmiger Anordnung der Separatoren die Dimension der Koeffizientenmatrix größer ist, werden für die Berechnung von L nur etwa halb so viele Operationen benötigt. Auch die Zahl der von null verschiedenen Subdiagonalelemente in L ist nur noch etwa halb so groß.

Tabelle 3.2 zeigt, daß sich bei einer Halbierung von $\theta(L)$ auch die auf einem Computer benötigte Zeit zur Berechnung von L stark reduziert. Die in den Spalten 3 und 6 angegebene Gesamtzeit (in Sek.) beinhaltet die Zeit zur Durchführung der symbolischen und numerischen Faktorisierung. Für die numerische Berechnung von L wurde ein von uns entwickelter, auf der *Multifrontal-Methode* (vgl. Duff und Reid [35, 36] oder Liu [97]) basierender Algorithmus verwendet. Auf die symbolische und numerische Faktorisierung werden wir in Kapitel 5 noch einmal näher eingehen. Die Spalten 2 und 5 zeigen, daß die numerische Faktorisierung den größten Aufwand verursacht. Alle Zeitangaben wurden auf einer SUN Ultra mit 296 MHz UltraSPARC-II Prozessor und zwei GByte Hauptspeicher ermittelt.

Im folgenden wollen wir untersuchen, wieso es bei einer $\times$-förmigen Anordnung der Separatoren zu einer Reduzierung von $\eta(L)$ und $\theta(L)$ kommt. Wir betrachten dazu eine Raute $\mathfrak{R}_i$ mit Seitenlänge 2^i. $\mathfrak{R}_i$ besteht aus $2^i 2^i + (2^i - 1)(2^i - 1)$ Knoten und wird von $4 \cdot 2^i$ Knoten umrandet. Bei $+$-förmiger Anordnung der Separatoren entsteht analog zu $\mathfrak{R}_i$ ein Quadrat $\mathfrak{Q}_i$ mit Seitenlänge $2^i - 1$. $\mathfrak{Q}_i$ wird ebenfalls von $4 \cdot 2^i$ Knoten umrandet, besteht jedoch aus nur $(2^i - 1)(2^i - 1)$ Knoten. Daher gibt es ungefähr doppelt soviele Quadrate $\mathfrak{Q}_i$ wie Rauten $\mathfrak{R}_i$. Bezeichne $\mathrm{sep}(\mathfrak{Q}_i)$ die Menge der Knoten, durch dessen Entnahme $\mathfrak{Q}_i$ in vier kleinere Quadrate mit Seitenlänge $2^{i-1} - 1$ zerfällt. Es gilt $|\mathrm{sep}(\mathfrak{Q}_i)| = |\mathrm{sep}(\mathfrak{R}_i)|$. Weiter bezeichne $f_\square(i)$ die Anzahl der Subdiagonalelemente $\neq 0$ in den zu $\mathrm{sep}(\mathfrak{Q}_i)$ gehörenden Spalten des Cholesky-Faktors und $g_\square(i)$ die Anzahl der zur Faktorisierung dieser Spalten benötigten Operationen. Man überlegt sich leicht, daß gilt:

$$f_\square(i) = f_\diamond(i) \quad \text{und} \quad g_\square(i) = g_\diamond(i). \tag{3.11}$$

+-förmige Separatoren			×-förmige Separatoren		
n	num. Fakt.	total	n	num. Fakt.	total
127	0.70	0.86	129	0.44	0.57
255	3.12	3.86	257	1.85	2.28
511	22.95	26.24	513	13.60	16.58

Tab. 3.2: Vergleich der zur Berechnung von L benötigten Zeit (in Sek.) bei $+$-förmiger und bei $\times$-förmiger Anordnung der Separatoren. Die Zeit zur Durchführung der numerischen Faktorisierung ist separat angegeben. Sie beträgt mehr als 95 % der Gesamtzeit. Alle Zeitangaben wurden auf einer SUN Ultra mit 296 MHz UltraSPARC-II Prozessor und zwei GByte Hauptspeicher ermittelt.

Für die Berechnung von $\eta(L)$ und $\theta(L)$ bei $+$-förmiger Anordnung der Separatoren können wegen (3.11) weiterhin die Formeln (3.6) und (3.7) verwendet werden. Sie gehen jetzt jedoch mit einem zusätzlichen Faktor von ungefähr zwei in die Gesamtrechnung ein.

Aus der obigen Beobachtung kann ein allgemeines Kriterium zur Charakterisierung eines guten Orderings abgeleitet werden. Sei wieder π ein Ordering des Graphen $G = (V, E)$, $|V| = n$. Des weiteren sei k eine Zahl mit $1 \leq k < n$ und $U_k = \{v \in V;\ \pi(v) \leq k\}$. Die Menge U_k enthält also alle bis zu einem Zeitpunkt k eliminierte Knoten.

Eine Menge $D \subset U_k$ heißt *Gebiet (domain)* bezüglich U_k, falls $G(D)$ ein zusammenhängender Teilgraph von G ist und $\mathrm{adj}_G(D) \subset V - U_k$. Die Menge $\mathrm{adj}_G(D)$ heißt *Rand* des Gebietes D. Wegen $\mathrm{adj}_G(D) \subset V - U_k$ werden alle Knoten auf dem Rand von D nach den Knoten aus D numeriert. Da $G(D)$ ein zusammenhängender Teilgraph von G ist, folgt aus Lemma 2.1 unmittelbar, daß $\mathrm{adj}_G(D)$ eine Clique in dem Eliminationsgraphen G_k und damit auch in dem aufgefüllten Graphen G_π bildet.

Von besonderem Interesse ist nun das Verhältnis von $|D|$ zu $|\,\mathrm{adj}_G(D)|$. Ist der Quotient $|D|/|\,\mathrm{adj}_G(D)|$ groß, so bedeutet dies, daß trotz Elimination vieler Knoten nur eine kleine Clique entstanden ist. Der Eliminationsprozeß hat also einen geringen Fill-in verursacht. Daher ist die Güte eines Orderings ganz entscheidend von der Form der im Laufe des Eliminationsprozesses entstehenden Gebiete abhängig (vgl. auch Rothberg und Eisenstat [125]).

Sind in einem quadratischen Gitter mit 5-Punkte-Stern die Separatoren $+$-förmig angeordnet, so treten im Verlauf des Eliminationsprozesses überwiegend Gebiete in Form eines Quadrates auf. Für ein Quadrat $\mathfrak{Q}_i$ mit Seitenlänge $2^i - 1$ gilt:

$$\frac{|\mathfrak{Q}_i|}{|\,\mathrm{adj}_G(\mathfrak{Q}_i)|} = \frac{(2^i-1)(2^i-1)}{4 \cdot 2^i} \approx \frac{1}{4}(2^i - 2).$$

Demgegenüber treten bei einer $\times$-förmigen Anordnung der Separatoren überwiegend Gebiete in Form einer Raute auf. Für eine Raute $\mathfrak{R}_i$ mit Seitenlänge 2^i gilt:

$$\frac{|\mathfrak{R}_i|}{|\,\mathrm{adj}_G(\mathfrak{R}_i)|} = \frac{2^i 2^i + (2^i-1)(2^i-1)}{4 \cdot 2^i} \approx \frac{1}{2}(2^i - 1).$$

Der Quotient ist hier also – bedingt durch die speziellen *isoperimetrischen* Eigenschaften des Gitters – in etwa doppelt so groß.

3.3 Ein verbessertes Bottom-up-Verfahren für quadratische Gitter

In diesem Abschnitt zeigen wir, daß die rautenförmigen Gebiete auch mit Hilfe eines Bottom-up-Verfahrens generiert werden können. Wir benutzen dazu ein Minimum-Degree-Verfahren mit einer Minimum-Deficiency-Tie-Breaking-Strategie. Besitzen also mehrere Knoten in einem Eliminationsgraphen G_k den gleichen minimalen Grad, so wird derjenige Knoten eliminiert, dessen Unzulänglichkeit am geringsten ist. Um die Form der Gebiete schon während der Berechnung des Orderings erkennen zu können, wird jeder eliminierte Knoten im Gitter markiert. Ein Gebiet wird dann von einer zusammenhängenden Menge markierter Knoten gebildet. Bereits im vorherigen Abschnitt haben wir gesehen, daß die Knoten auf dem Rand eines Gebietes eine Clique bilden. Diese Cliquen entsprechen genau den im Laufe des Eliminationsprozesses entstehenden Cliquen. Daher ist es möglich, anhand der im Gitter eingezeichneten Gebiete den Grad eines Knotens v im Eliminationsgraphen zu bestimmen.

Abbildung 3.3 zeigt in vier Schnappschüssen die durch das spezielle Minimum-Degree-Ordering geformten Gebiete in einem 16×16-*Torus**. Die Gebiete werden dabei von den grau unterlegten und bereits eliminierten Knoten gebildet. Wir haben einen Torus gewählt, da es in einem Gitter aufgrund der Randknoten zu kleineren Verwerfungen bei der Bildung der Gebiete kommt. Diese verursachen zwar nur eine geringe Verschlechterung des Orderings, erschweren jedoch die Analyse erheblich. Perfekte rautenförmige Gebiete werden in einem $n \times n$-Torus mit $n = 2^l$ gebildet.

Wir wollen nun untersuchen, wie die Minimum-Deficiency-Tie-Breaking-Strategie die Bildung der Gebiete beeinflußt. Da in einem Torus alle Knoten den Grad vier besitzen, können wir ohne Einschränkung annehmen, daß der in Abbildung 3.3 (a) eingezeichnete Knoten v als erstes eliminiert wird. Hierdurch entsteht ein Gebiet, das nur v enthält. Abgesehen von den vier zu v benachbarten Knoten, die eine Clique bilden, besitzen alle restlichen Knoten weiterhin den Grad vier. Durch die entstandene Clique ist jedoch die Unzulänglichkeit der Knoten u_1, u_2, u_3 und u_4 um eins geringer. Durch die Tie-Breaking-Strategie wird so erzwungen, daß als nächstes ein Knoten aus $\{u_1, \ldots, u_4\}$ eliminiert wird. Nach $n^2/2$ Eliminationsschritten entsteht so die in Abbildung 3.3 (a) dargestellte Anordnung der Gebiete. In dieser Anordnung liegen alle noch nicht eliminierten Knoten auf den Rändern von vier Gebieten. Jedes Gebiet besteht aus genau einem (eliminierten) Knoten. Man überlegt sich leicht, daß nach weiteren $n^2/8$ Eliminationsschritten die in Abbildung 3.3 (b) dargestellte Anordnung der Gebiete entsteht. Hier besitzen alle

*Ein Torus ist ein Gitter mit zusätzlichen *Wrap-Around-Kanten*.

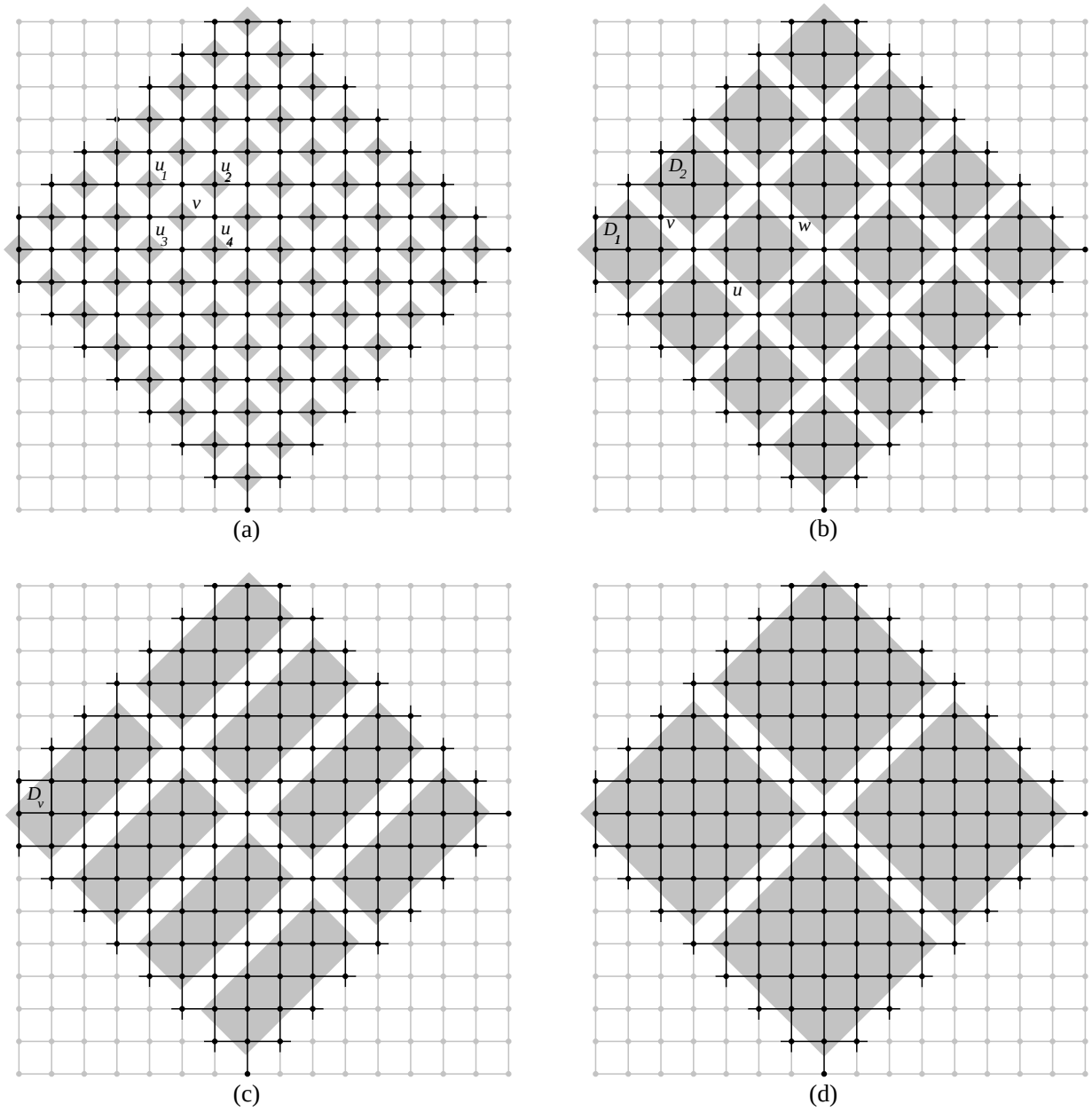


Abb. 3.3: Die durch das spezielle Minimum-Degree-Ordering geformten Gebiete in einem 16×16-Torus. Der Übersichtlichkeit halber ist die Bildung der Gebiete nur für den mittleren Teil des Torus dargestellt. Darüber hinaus wurde auf ein Einzeichnen der Wrap-Around-Kanten verzichtet.

	Nested-Dissection $\times$-förmige Separatoren		Minimum-Degree mit Min.-Def.-Tie-Breaking		Approximate-Minimum-Degree	
n	$\eta(L)/10^3$	$\theta(L)/10^6$	$\eta(L)/10^3$	$\theta(L)/10^6$	$\eta(L)/10^3$	$\theta(L)/10^6$
129	266	17	287	18	346	27
257	1296	139	1386	146	1863	269
513	6153	1140	6547	1205	9834	2776

Tab. 3.3: Vergleich zwischen dem verbesserten Nested-Dissection-, dem speziellen Minimum-Degree- und dem Approximate-Minimum-Degree-Ordering. Die Seitenlänge n der Gitter beträgt jeweils 129, 257 und 513.

Knoten, die auf den Rändern von zwei Gebieten liegen, den gleichen minimalen Grad. Dies gilt insbesondere für die Knoten u, v und w. Ohne Einschränkung sei angenommen, daß v als erster Knoten eliminiert wird. Hierdurch werden die Gebiete D_1 und D_2 verschmolzen, und es entsteht das in Abbildung 3.3 (c) dargestellte rechteckige Gebiet D_v. Wegen der Clique $\mathrm{adj}_G(D_v)$ ist jetzt die Unzulänglichkeit von u geringer als die eines jeden anderen Knotens mit minimalem Grad. Insbesondere wird daher nicht w, sondern u als nächstes eliminiert. Durch die Tie-Breaking-Strategie wird so erzwungen, daß die rechteckigen Gebiete mit ihrer langen Seite einander gegenüberliegen. In den folgenden Eliminationsschritten wird dann aus je zwei rechteckigen Gebieten erneut ein rautenförmiges Gebiet gebildet (vgl. Abbildung 3.3 (d)). Mit Hilfe einer Analyse ähnlich der in Abschnitt 3.2 zeigt man:

Satz 3.2 *Wird zur Numerierung eines $n \times n$-Torus ein Minimum-Degree-Verfahren mit einer Minimum-Deficiency-Tie-Breaking-Strategie benutzt, so benötigt man für die Berechnung des Cholesky-Faktors $145/8\, n^3 + O(n^2 \log n)$ Multiplikations- und Additionsoperationen. Der Cholesky-Faktor enthält dabei $31/8\, n^2 \log n + O(n^2)$ von null verschiedene Elemente.*

Tabelle 3.3 zeigt noch einmal, daß bei dem speziellen Minimum-Degree-Verfahren der Fill-in bzw. die Zahl der benötigten Operationen etwas höher ist als beim verbesserten Nested-Dissection-Verfahren. Ursache hierfür sind die kleineren Verwerfungen bei der Bildung der Gebiete am Rand des Gitters. Mit Hilfe der Minimum-Deficiency-Tie-Breaking-Strategie werden jedoch im Vergleich zum Approximate-Minimum-Degree-Algorithmus von Amestoy et al. [1] sehr viel bessere Orderings erzeugt. Der Approximate-Minimum-Degree-Algorithmus von Amestoy et al. gehört zu den effektivsten Ordering-Verfahren, die derzeit bekannt sind. Mit seiner Hilfe können qualitativ hochwertige Orderings in sehr kurzer Zeit berechnet werden. Der Algorithmus wird in Abschnitt 4.1.2 kurz beschrieben.

Kapitel 4

Ordering-Verfahren für beliebige Graphen

Im vorherigen Kapitel haben wir gesehen, daß Knotenseparatoren bei der Numerierung gitterförmiger Graphen eine wichtige Rolle spielen. Viele der speziell für diese Graphen entwickelten Ordering-Verfahren benutzen in irgendeiner Form Georges Nested-Dissection-Algorithmus. Demgegenüber werden Ordering-Verfahren, die auf der Bottom-up-Methode basieren, kaum betrachtet. Dies ist jedoch nicht verwunderlich, denn aufgrund der homogenen Struktur der Graphen (fast alle Knoten besitzen den gleichen Grad) und der lokalen Struktur eines Bottom-up-Algorithmus (der als nächstes zu eliminierende Knoten wird nach einem lokalen Knotenauswahlverfahren bestimmt) hängt die Güte der generierten Orderings ganz entscheidend von der verwendeten Tie-Breaking-Strategie ab. Eine Random-Tie-Breaking-Strategie allein reicht nicht aus, um mit den speziellen Dissection-Verfahren konkurrieren zu können. Im ungünstigsten Fall erhält man sogar ein Ordering, durch das ein asymptotisch höherer Fill-in erzeugt wird (vgl. Berman und Schnitger [21]).

Die Situation ändert sich jedoch völlig, wenn Numerierungen für beliebige Graphen berechnet werden müssen. In diesen Graphen sind die „guten“ Knotenseparatoren nicht a priori bekannt, sondern müssen erst mit Hilfe eines geeigneten Verfahrens konstruiert werden. Da die Güte eines Nested-Dissection-Orderings ganz entscheidend von der Größe der Knotenseparatoren abhängt, kommt dem Verfahren zur Bestimmung der Separatoren eine herausragende Bedeutung zu. Bis Mitte der neunziger Jahre gab es keinen Nested-Dissection-Algorithmus, der für die in der Praxis auftretenden, nicht gitterförmigen Graphen konsistent bessere Orderings produzierte als ein Minimum-Degree-Algorithmus. Erst in den letzten Jahren gelang es mit dem Aufkommen leistungsfähiger Algorithmen zur Bestimmung von Knotenseparatoren, die Vorherrschaft der Bottom-up-Verfahren zu brechen.

Im Mittelpunkt dieses Kapitels steht die Entwicklung eines neuen Ordering-Verfahrens für beliebige Graphen. Charakteristisch für das Verfahren ist eine enge Koppelung zwischen Bottom-up- und Top-down-Methoden. Dabei werden die im Rahmen eines Top-down-Verfahrens konstruierten Knotenseparatoren als Ränder der von einem unvollständigen Bottom-up-Ordering ge-

bildeten Gebiete interpretiert. Im Umkehrschluß können also Knotenseparatoren als eine Aneinanderreihung von Randsegmenten aufgefaßt werden, die zu den Gebieten eines unvollständigen Eliminationsprozesses gehören. Das neue Ordering-Verfahren zeichnet sich durch die folgenden Besonderheiten aus:

- Die Knotenseparatoren werden mit Hilfe eines speziellen Multilevel-Verfahrens konstruiert. Dabei wird zur Schrumpfung der Graphen ein Eliminationsprozeß benutzt, der dem zur Berechnung eines Bottom-up-Orderings sehr ähnlich ist.
- Basierend auf den Knotenseparatoren wird nicht nur ein Bottom-up-Ordering generiert, sondern mehrere. Das beste Ordering wird schließlich von dem Algorithmus ausgegeben.

Auf der einen Seite benutzen wir also Bottom-up-Techniken zur Konstruktion der Knotenseparatoren, auf der anderen Seite dienen die Knotenseparatoren als ein Gerüst zur Generierung mehrerer Bottom-up-Orderings. Es kommt so zu einer wechselseitigen Bereicherung der Methoden, wodurch sich die Qualität der Orderings erheblich verbessert.

Dieses Kapitel ist wie folgt aufgebaut: Zunächst beschreiben wir in Abschnitt 4.1 die wichtigsten, aus der Literatur bekannten Algorithmen zur Numerierung beliebiger Graphen. Anschließend stellen wir in Abschnitt 4.2 das neue Ordering-Verfahren vor. Im Vergleich zum Approximate-Minimum-Degree-Algorithmus von Amestoy et al. reduziert sich bei Verwendung des neuen Verfahrens die Anzahl der zur Berechnung des Cholesky-Faktors benötigten Operationen um durchschnittlich 42 %. Schließlich stellen wir in Abschnitt 4.3 die Ordering-Bibliothek PORD [131] vor und zeigen, wie das neue Verfahren in die Bibliothek eingebunden ist. Darüber hinaus vergleichen wir PORD mit einigen der zur Zeit mächtigsten Ordering-Codes. Hierzu zählen das an der Universität von Minnesota entwickelte Programm METIS [79] (Karypis und Kumar), die an der Universität von Bordeaux entwickelte Bibliothek SCOTCH [109] (Pellegrini) sowie das bei Harwell-Boeing Information and Support Services entwickelte Programmpaket SPOOLES [8] (Ashcraft und Grimes). Für unseren Vergleich benutzen wir einen weit verbreiteten und frei verfügbaren Satz von Benchmark-Matrizen. Die meisten dieser Matrizen stammen aus der bekannten *Harwell-Boeing-Collection* [34], sowie aus der *Sparse-Matrix-Collection* [28] von Tim Davis, Universität von Florida.

4.1 Literaturübersicht

Dieses Unterkapitel ist in fünf Teilabschnitte gegliedert. Zuerst stellen wir in 4.1.1 die Klasse der Quotientengraphen vor. Diese Graphen spielen eine wichtige Rolle in unserem Ordering-Algorithmus. In Abschnitt 4.1.2 beschreiben wir Techniken zur effizienten Implementierung des Minimum-Degree-Algorithmus. Außerdem gehen wir näher auf den Minimum-Deficiency-Algorithmus ein und präsentieren einige interessante Knotenauswahlstrategien, die auf einer approximativen Berechnung der Unzulänglichkeit eines Knotens beruhen. In Abschnitt 4.1.3 beschreiben

wir kurz die wichtigsten Techniken zur Konstruktion der Knotenseparatoren in einem Nested-Dissection-Algorithmus. Schließlich stellen wir in Abschnitt 4.1.4 bereits bekannte Ansätze zur Kombination von Bottom-up- und Top-down-Methoden vor.

4.1.1 Quotientengraphen

Wie bereits in Abschnitt 2.3.2 erwähnt, werden für die Berechnung eines Bottom-up-Orderings die Eliminationsgraphen G_k, $k = 1, \ldots, n$ benötigt. Aus Effizienzgründen ist es jedoch nicht ratsam, die einzelnen Eliminationsgraphen explizit zu bilden. Mit Hilfe sogenannter *Quotientengraphen (quotient graphs, generalized element model)* (vgl. Duff und Reid [35], Goerge und Liu [53] oder Speelpenning [136]) können die Graphen G_k sehr elegant dargestellt werden. Zur Bezeichnung der Quotientengraphen verwenden wir kaligraphische Buchstaben.

Es sei angenommen, daß $G_k = (V_k, E_k)$ aus G durch Elimination der Knoten $U_k \subset V$ entstanden ist, d. h. $V = V_k \cup U_k$ und $V_k \cap U_k = \emptyset$. Der zu G_k gehörende Quotientengraph $\mathcal{G}_k = (\mathcal{X}_k, \mathcal{E}_k)$ enthält die Knoten $\mathcal{X}_k = \mathcal{D}_k \cup \mathcal{V}_k$ mit

$$\begin{aligned} \mathcal{D}_k &= \{D;\ D \text{ ist ein Gebiet bezüglich } U_k\}, \\ \mathcal{V}_k &= \{\{v\};\ v \in V_k\}. \end{aligned} \tag{4.1}$$

In diesem Kontext heißt eine Menge $D \in \mathcal{D}_k$ *Cliquenelement* (oder kurz *Element*) und eine Menge $\{v\} \in \mathcal{V}_k$ *Variable*. Die Knoten $\mathcal{X}_k = \mathcal{D}_k \cup \mathcal{V}_k$ des Quotientengraphen $\mathcal{G}_k$ liefern eine Partitionierung der Knoten von G. Dabei enthält $\mathcal{D}_k$ disjunkte Gebiete, die aus bereits eliminierten Knoten bestehen, und $\mathcal{V}_k$ eine Variable für jeden nicht eliminierten Knoten. Im folgenden werden wir nicht immer explizit zwischen der Variablen $\{v\}$ und dem Knoten v unterscheiden.

Sei D ein Element und seien u, v zwei Variablen des Quotientengraphen. In $\mathcal{G}_k$ gibt es zwei unterschiedliche Kantentypen: *Domain-Vertex-Kanten* der Form (D, v), falls $v \in \mathrm{adj}_G(D)$ gilt, und *Vertex-Vertex-Kanten* der Form (u, v), falls $(u, v) \in E$ gilt und es kein Element D gibt mit $u \in \mathrm{adj}_G(D)$ und $v \in \mathrm{adj}_G(D)$. In Abschnitt 4.2 betrachten wir Quotientengraphen, in denen es keine Eins-zu-Eins-Beziehung zwischen den Variablen und den nicht eliminierten Knoten gibt. Vielmehr repräsentiert dort jede Variable eine Menge von nicht eliminierten Knoten. Die folgende formale Definition von $\mathcal{E}_k$ deckt auch diesen allgemeineren Fall ab:

$$\begin{aligned} \mathcal{E}_k = &\{(D, V_i);\ D \in \mathcal{D}_k, V_i \in \mathcal{V}_k \text{ und } V_i \cap \mathrm{adj}_G(D) \neq \emptyset\} \\ &\cup \{(V_i, V_j);\ V_i, V_j \in \mathcal{V}_k,\ V_i \cap \mathrm{adj}_G(V_j) \neq \emptyset \text{ und } \nexists D \in \mathcal{D}_k \text{ mit } V_i, V_j \cap \mathrm{adj}_G(D) \neq \emptyset\}. \end{aligned} \tag{4.2}$$

Es gibt eine enge Beziehung zwischen den Elementen von $\mathcal{G}_k$ und den Cliquen in G_k. Die Kantenmenge E_k des Eliminationsgraphen G_k kann nämlich geschrieben werden als

$$E_k = (E \cap (V_k \times V_k)) \cup \bigcup_{D \in \mathcal{D}_k} (\mathrm{adj}_G(D) \times \mathrm{adj}_G(D)). \tag{4.3}$$

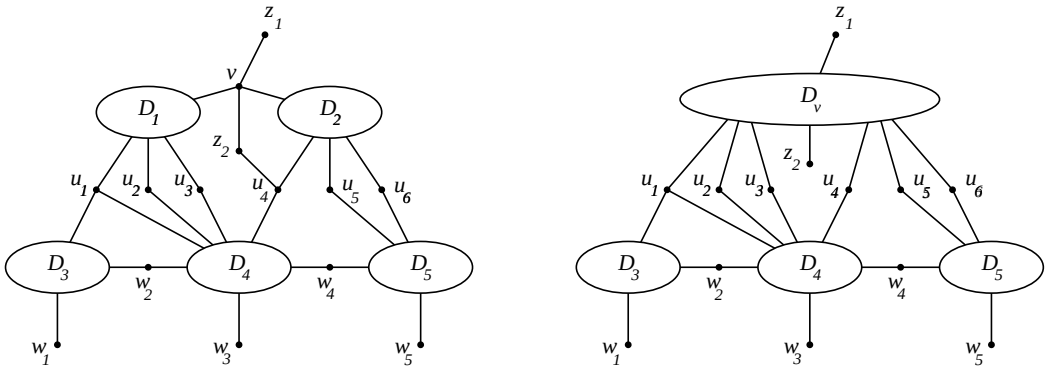


Abb. 4.1: Der Quotientengraph $\mathcal{G}_{k-1}$ (links) und der durch Elimination des Knotens v entstandene Quotientengraph $\mathcal{G}_k$ (rechts). Jedes Cliquenelement ist durch ein Oval dargestellt. Durch die Elimination von v entsteht in $\mathcal{G}_k$ ein neues Cliqenelement D_v. Dieses Cliquenelement ersetzt v und die ursprünglich zu v benachbarten Cliquenelemente D_1 und D_2. Man beachte, daß die Vertex-Vertex-Kante (z_2, u_4) nicht mehr in $\mathcal{G}_k$ vorhanden ist, da beide Knoten zu D_v adjazent sind.

Die erste Menge enthält alle Kanten aus G, die zu nicht eliminierten Knoten inzident sind. Die zweite Menge enthält die Kanten aller Cliquen, die im Laufe des Eliminationsprozesses entstanden sind. Diese Kanten sind jedoch nicht explizit in $\mathcal{G}_k$ abgespeichert. Vielmehr wird eine Clique C durch ein Element $D \in \mathcal{D}_k$ repräsentiert. Die Clique C enthält dabei alle Knoten/Variablen, die von D über eine Domain-Vertex-Kante erreichbar sind.

In dem Eliminationsgraphen G_k besteht die Menge $\mathrm{adj}_{G_k}(v)$ aus allen Knoten u, die in $\mathcal{G}_k$ über eine Vertex-Vertex-Kante (v, u) oder über zwei aufeinanderfolgende Domain-Vertex-Kanten (v, D) und (D, u) erreicht werden können. Damit wird auch klar, warum eine Vertex-Vertex-Kante (u, v) nur dann in $\mathcal{G}_k$ enthalten ist, wenn kein Element $D \in \mathcal{D}_k$ existiert mit $u \in \mathrm{adj}_G(D)$ und $v \in \mathrm{adj}_G(D)$. Gibt es nämlich ein solches D, so gehören u, v zu einer Clique und eine explizite Speicherung der Kante (u, v) ist überflüssig.

Bereits in Abbildung 3.3 haben wir am Beispiel des Torus gesehen, daß durch die Elimination eines Knotens v alle Gebiete D mit $v \in \mathrm{adj}_G(D)$ zu einem neuen Gebiet D_v verschmolzen werden, das v enthält. Dementsprechend entsteht der Quotientengraph $\mathcal{G}_k$ aus $\mathcal{G}_{k-1}$ ($\mathcal{G}_0$ ist *isomorph* zu G) durch Verschmelzen der zu v benachbarten Cliquenelemente. Abbildung 4.1 verdeutlicht die Vorgehensweise an einem Beispiel.

Durch die beim Übergang von $\mathcal{G}_{k-1}$ nach $\mathcal{G}_k$ durchgeführten Verschmelzungsoperationen wird ein Baum auf den Knoten von G definiert. Seien dazu D_{u_i}, $i = 1, \ldots, t$, die zu v benachbarten Cliquenelemente in $\mathcal{G}_{k-1}$, wobei wir annehmen, daß D_{u_i} durch die Elimination des Knotens u_i entstanden ist. Beim Übergang von $\mathcal{G}_{k-1}$ nach $\mathcal{G}_k$ werden die Elemente D_{u_i} von D_v absorbiert. Deswegen definieren wir v als *Vater* von $u_1, \ldots, u_t$. Fährt man auf diese Art und Weise fort, so erhält man einen Baum, der alle Knoten aus G enthält. Die Blätter des Baumes werden dabei von denjenigen Knoten gebildet, die zum Zeitpunkt ihrer Elimination zu keinem Element

benachbart waren. Der Baum heißt *Eliminationsbaum (elimination tree)* und spielt eine wichtige Rolle bei der direkten Lösung dünn besetzter Gleichungssysteme (vgl. Duff und Reid [35], Eisenstat et al. [41], Liu [96], Schreiber [127] oder Speelpenning [136]).

4.1.2 Bottom-up-Verfahren

Bottom-up-Verfahren bauen den Eliminationsbaum von den Blättern zur Wurzel auf. In jeder Iteration wird basierend auf einer lokalen Knotenauswahlstrategie der nächste zu eliminierende Knoten bestimmt. In diesem Abschnitt stellen wir zwei der bekanntesten Bottom-up-Verfahren genauer vor, den Minimum-Degree- und den Minimum-Deficiency-Algorithmus.

4.1.2.1 Der Minimum-Degree-Algorithmus

Das am weitesten verbreitete Bottom-up-Verfahren ist der Minimum-Degree-Algorithmus (MD). Im Laufe der letzten Jahrzehnte wurden verschiedene Techniken entwickelt, die zu einer erheblichen Reduzierung der Laufzeit des Algorithmus führten. Hierzu zählt u. a. die oben beschriebene Darstellung der Eliminationsgraphen als Quotientengraphen. George und Liu zeigen in [52], daß bei Verwendung geeigneter Datenstrukturen zur Speicherung eines Quotientengraphen $\mathcal{G}_k$ nie mehr Platz benötigt wird als zur Speicherung des Graphen $\mathcal{G}_0$. Eine dynamische Allokierung zusätzlichen Speichers ist daher nicht nötig. Im folgenden wollen wir einige weitere Verbesserungen kurz vorstellen. Für einen vollständigen Überblick sei auf Liu [54] verwiesen.

Superknoten Die vielleicht wichtigste Verbesserung des Minimum-Degree-Algorithmus besteht darin, die Knoten aus G_k zu sogenannten *Superknoten* zusammenzufassen.* Zwei Knoten u, v aus G_k gehören zu dem gleichen Superknoten, falls gilt

$$\mathrm{adj}_{G_k}(u) \cup \{u\} = \mathrm{adj}_{G_k}(v) \cup \{v\}. \tag{4.4}$$

In diesem Fall heißen u und v *nicht unterscheidbar (indistinguishable)*. Durch (4.4) wird eine Äquivalenzrelation auf den Knoten aus G_k definiert. Jede Äquivalenzklasse bildet dabei einen Superknoten. In einem Minimum-Degree-Ordering können die Knoten eines Superknotens I aufeinanderfolgend numeriert werden. Dies liegt daran, daß alle Knoten aus I den gleichen Grad besitzen, und daß sich nach Elimination eines Knotens $v \in I$ der Grad aller in I verbleibenden Knoten um eins verringert. War also der Knotengrad von v minimal, so ist anschließend der Knotengrad aller verbleibenden Knoten minimal. Darüber hinaus gilt, daß zwei nicht unterscheidbare Knoten $u, v \in I$ bis zu ihrer Elimination nicht unterscheidbar bleiben. Innerhalb des Minimum-Degree-Algorithmus können daher alle Knoten eines Superknotens I durch einen einzigen Knoten aus I repräsentiert werden. Hierdurch verringert sich die Anzahl der Knoten in

*Im Zusammenhang mit Quotientengraphen spricht man auch von *Supervariablen*.

G_k und damit auch die Anzahl der Iterationen des Minimum-Degree-Algorithmus. Um weiterhin die Knotengrade exakt berechnen zu können, erhält der Repräsentant das Gewicht $|I|$.

Wir wollen die Bildung der Superknoten an den Quotientengraphen $\mathcal{G}_{k-1}$ und $\mathcal{G}_k$ aus Abbildung 4.1 veranschaulichen. In dem durch $\mathcal{G}_{k-1}$ dargestellten Eliminationsgraphen G_{k-1} gilt $\mathrm{adj}_{G_{k-1}}(u_2) \cup \{u_2\} = \mathrm{adj}_{G_{k-1}}(u_3) \cup \{u_3\} = \{v, u_1, u_2, u_3, u_4, w_2, w_3, w_4\}$. Die Knoten u_2 und u_3 sind also nicht unterscheidbar. Sei u_2 Repräsentant des Superknotens $\{u_2, u_3\}$. Dann kann u_3 aus G_{k-1} bzw. $\mathcal{G}_{k-1}$ entfernt werden und es gilt $|u_2| = 2$. Man darf u_3 jedoch nicht bei der späteren Numerierung vergessen. Durch die Verschmelzung von D_1 und D_2 wächst der Superknoten um u_4. Daher kann auch u_4 entfernt werden und es gilt $|u_2| = 3$.

Externer Knotengrad Sehr eng verbunden mit dem Konzept der Superknoten ist das Konzept des *externen Knotengrades* [91]. Sei v der Repräsentant eines Superknotens I. Zur besseren Veranschaulichung sei angenommen, daß die Knoten aus $I - \{v\}$ nicht wie oben beschrieben aus G_k entfernt wurden (G_k ist also ungewichtet). Der externe Knotengrad von v ist dann $\deg_{G_k}(I)$. Die zu eliminierenden Repräsentanten werden jetzt nicht mehr anhand ihres exakten Knotengrades, sondern anhand ihres externen Knotengrades ausgewählt. Dies kann wie folgt motiviert werden: Aus (4.4) folgt unmittelbar, daß jeder Knoten $u \in I - \{v\}$ bereits zu allen Knoten aus $\mathrm{adj}_{G_k}(v)$ benachbart ist. Daher kann durch die Elimination des Knotens v keine Fill-Kante enstehen, die inzident zu einem Knoten aus $I - \{v\}$ ist. Vielmehr können nur Fill-Kanten zwischen den Knoten aus $\mathrm{adj}_{G_k}(I)$ erzeugt werden. Durch die Verwendung des externen Knotengrades erhält man so eine genauere obere Schranke für die Zahl der einzufügenden Fill-Kanten, wodurch sich die Qualität der Orderings leicht verbessert (vgl. George und Liu [54] oder Liu [91]).

Multiple-Minimum-Degree Eine zentrale Eigenschaft des Minimum-Degree-Algorithmus besteht darin, daß in jeder Iteration k genau *ein* Knoten (bzw. Repräsentant) v aus G_{k-1} eliminiert wird. Anschließend wird der Eliminationsgraph G_k konstruiert und der Grad eines jeden Knoten $u \in \mathrm{adj}_{G_{k-1}}(v)$ neu berechnet. Genau dieser Degree-Update-Schritt ist sehr zeitaufwendig. In Lius *Multiple-Minimum-Degree-Algorithmus* (MMD) [91] wird deshalb in jeder Iteration eine *maximale unabhängige* Menge von Knoten mit minimalem Grad aus G_{k-1} entfernt. Diese Multiple-Elimination-Technik reduziert die Anzahl der Iterationen und führt zu einer signifikanten Beschleunigung des Algorithmus. In gewissem Sinne stellt die Multiple-Elimination-Technik eine Art Tie-Breaking-Strategie dar. Im Vergleich zu einem normalen Minimum-Degree-Algorithmus mit einer Random-Tie-Breaking-Strategie werden jedoch keine besseren Orderings generiert.

Approximative Berechnung der Knotengrade Eine weitere Reduzierung der Laufzeit läßt sich dadurch erreichen, daß nach Elimination eines Knotens v der neue Grad aller Knoten $u \in \mathrm{adj}_{G_{k-1}}(v)$ nur approximativ berechnet wird (vgl. Amestoy et al. [1] oder Gilbert et al. [59]). Sei dazu wieder $\mathcal{G}_k$ der Quotientengraph von G_k und D_v das neu entstandene Cliquenelement. In dem *Approximate-Minimum-Degree-Algorithmus* (AMD) von Amestoy et al. [1] wird für u

zunächst der Wert

$$|\operatorname{adj}_{\mathcal{G}_k}(D_v)| + \sum_{\substack{D \neq D_v \\ (D,u) \in \mathcal{E}_k}} |\operatorname{adj}_{\mathcal{G}_k}(D) - \operatorname{adj}_{\mathcal{G}_k}(D_v)| \tag{4.5}$$

berechnet. Dieser Wert wird anschließend um die Zahl der Knoten u' erhöht, die mit u über eine Vertex-Vertex-Kante verbunden sind. Man erhält so eine obere Schranke $\operatorname{approxdeg}_{G_k}(u) \geq \deg_{G_k}(u)$. Dabei kann es durch die Summenbildung in (4.5) zu Mehrfachzählungen kommen.

In Abbildung 4.1 (rechts) ist dies beispielsweise bei der Berechnung von $\operatorname{approxdeg}_{G_k}(u_1)$ der Fall. Der Knoten u_1 ist über eine Domain-Vertex-Kante mit den Elementen D_3 und D_4 verbunden. In die Summenbildung gehen daher $\operatorname{adj}_{\mathcal{G}_k}(D_3) - \operatorname{adj}_{\mathcal{G}_k}(D_v)$ und $\operatorname{adj}_{\mathcal{G}_k}(D_4) - \operatorname{adj}_{\mathcal{G}_k}(D_v)$ ein. Beide Mengen sind jedoch nicht disjunkt. Der Knoten w_2 ist in beiden Mengen enthalten. Daher ist der Wert $\operatorname{approxdeg}_{G_k}(u_1)$ ist um eins höher als der exakte Knotengrad.

Man beachte, daß es durch die Einbeziehung aller Knoten u', die mit u über eine Vertex-Vertex-Kante verbunden sind, nicht zu Mehrfachzählungen kommen kann. Nach Konstruktion der Quotientengraphen existiert die Kante (u, u') nämlich nur dann, wenn es kein Cliquenelement D gibt mit $(D, u) \in \mathcal{E}_k$ und $(D, u') \in \mathcal{E}_k$. Der Knoten u' kann also nicht in (4.5) auftreten.

Unter der Voraussetzung, daß $|\operatorname{adj}_{\mathcal{G}_k}(D)|$ für alle Cliquenelemente D bekannt ist, benötigt man für den Degree-Update-Schritt insgesamt $O(\sum_{u \in \operatorname{adj}_{\mathcal{G}_k}(D_v)} \deg_G(u))$ Zeiteinheiten. Im Gegensatz dazu kostet die Berechnung der exakten Knotengrade Zeit $O(\sum_{u \in \operatorname{adj}_{\mathcal{G}_k}(D_v)} \deg_{G_k}(u))$. In der Regel ist $\deg_G(u)$ sehr viel kleiner als $\deg_{G_k}(u)$, so daß die approximative Berechnung zu einer erheblichen Beschleunigung des Algorithmus führt. Die Qualität der Orderings leidet darunter nicht.†

4.1.2.2 Der Minimum-Deficiency-Algorithmus

Im Gegensatz zum Minimum-Degree- wurde dem Minimum-Deficiency- oder Minimum-Local-Fill-Algorithmus (MF) weit weniger Aufmerksamkeit geschenkt. Dies liegt hauptsächlich daran, daß die Berechnung von $\operatorname{def}(v)$ sehr viel aufwendiger ist als die Berechnung von $\deg(v)$. Darüber hinaus beeinflußt die Elimination eines Knotens v nicht nur die Unzulänglichkeit aller Nachbarn u von v, sondern auch die aller Nachbarn w von u. Nach Elimination von v muß daher die Unzulänglichkeit aller Knoten im Abstand zwei von v neu berechnet werden.

Historisch gesehen wurde der Minimum-Deficiency-Algorithmus unterschätzt. Man glaubte viele Jahre, daß im Vergleich zum Minimum-Degree-Algorithmus die Güte eines Orderings nur marginal verbessert wird (vgl. z. B. Duff et al. [33]). Neuere Untersuchungen belegen jedoch (siehe Meszaros [103] sowie Rothberg und Eisenstat [125]), daß der Minimum-Deficiency-Al-

†Werden die approximativen Knotengrade nach Gilbert et al. berechnet, so verschlechtert sich die Qualität der Orderings zum Teil erheblich.

gorithmus für viele Graphen erheblich bessere Orderings produziert. Aufgrund der extrem hohen Laufzeit ist der Algorithmus jedoch nicht in der Praxis einsetzbar.

Die Unzulänglichkeit eines Knotens kann aber als Tie-Breaker in einem Minimum-Degree-Algorithmus benutzt werden (vgl. z. B. Cavers [25] oder Meszaros [103]). Dann ist die Berechnung von $\mathrm{def}(v)$ nur für Knoten mit minimalem Grad erforderlich. Wie bereits am Beispiel des Torus gesehen, läßt sich mit Hilfe dieser Tie-Breaking-Strategie die Güte eines Minimum-Degree-Orderings erheblich verbessern. Leider ist dies eher die Ausnahme als die Regel.

Der Erfolg des Minimum-Deficiency-Verfahrens beruht darauf, daß neu entstehende Gebiete so positioniert werden, daß sie mit bereits existierenden Gebieten ein großes Randsegment teilen. Die Gebiete heißen dann *wohl positioniert (well aligned)*. Wohl positionierten Gebiete ermöglichen in späteren Eliminationsschritten die Bildung von Gebieten mit einem großen Verhältnis von Inhalt zu Umfang. In einem Minimum-Degree-Algorithmus mit einer Minimum-Deficiency-Tie-Breaking-Strategie können jedoch nur solche Elemente wohl positioniert werden, die durch Elimination von Knoten mit minimalem Grad entstehen.

Eine Möglichkeit, die Laufzeit des Minimum-Deficiency-Algorithmus zu reduzieren, besteht in der approximativen Berechnung der Unzulänglichkeit eines Knotens. Sei wieder v der aus G_{k-1} eliminierte Knoten. Wie beim Minimum-Degree-Algorithmus wird zunächst für jeden Knoten $u \in \mathrm{adj}_{G_{k-1}}(v)$ der neue Knotengrad $\deg_{G_k}(u)$ berechnet. Bezeichne I_u den Superknoten, dessen Repräsentant u ist. Dann ist $d = \deg_{G_k}(I_u)$ der externe Knotengrad von u und $\frac{1}{2}d(d-1)$ eine obere Schranke für $\mathrm{def}_{G_k}(u)$. Da $\mathrm{adj}_{G_{k-1}}(v)$ eine Clique in G_k bildet mit $I_u \subset \mathrm{adj}_{G_{k-1}}(v)$, kann die obere Schranke reduziert werden auf $\frac{1}{2}d(d-1) - \frac{1}{2}c(c-1)$ mit $c = |\mathrm{adj}_{G_{k-1}}(v) - I_u|$.

Rothberg und Eisenstat [125] benutzen die verbesserte obere Schranke zur Formulierung einiger sehr effektiver Knotenauswahlstrategien. Jede Auswahlstrategie ist durch eine Funktion $\mathrm{score} : V \to \mathbb{R}$ beschrieben. Beim Übergang von G_{k-1} nach G_k wird dann immer ein Knoten v mit minimalem Score-Wert eliminiert. In Abhängigkeit von der Auswahlstrategie berechnet sich der neue Score-Wert eines Knotens $u \in \mathrm{adj}_{G_{k-1}}(v)$ wie folgt:

1. *Approximate-Minimum-Local-Fill (AMF)*

$$\mathrm{score}_{\mathrm{AMF}}(u) = \frac{d(d-1) - c(c-1)}{2}. \tag{4.6}$$

2. *Approximate-Minimum-Mean-Local-Fill (AMMF)*
   Wird der Knoten u zu einem späteren Zeitpunkt eliminiert, so ist die Unzulänglichkeit der in I_u verbleibenden Knoten gleich null. Im Schnitt verursacht also die Elimination aller Knoten aus I_u nur $\frac{\mathrm{def}_{G_k}(u)}{|I_u|}$ zusätzliche Kanten. Dies motiviert die Score-Funktion

$$\mathrm{score}_{\mathrm{AMMF}}(u) = \frac{\mathrm{score}_{\mathrm{AMF}}(u)}{|I_u|}. \tag{4.7}$$

3. *Approximate-Minimum-Increase-in-Neighbor-Degree (AMIND)*
   Durch die Elimination aller Knoten aus I_u entstehen $\mathrm{def}_{G_k}(u)$ neue Kanten, die zu einem Knoten aus $\mathrm{adj}_{G_k}(I_u)$ inzident sind. Zugleich werden jedoch auch $d \cdot |I_u|$ Kanten gelöscht, die zu einem Knoten aus $\mathrm{adj}_{G_k}(I_u)$ inzident sind. Daher kann $\mathrm{score}_{\mathrm{AMF}}(u)$ modifiziert werden zu

$$\mathrm{score}_{\mathrm{AMIND}}(u) = \mathrm{score}_{\mathrm{AMF}}(u) - d \cdot |I_u|. \tag{4.8}$$

Alle drei Auswahlstrategien begünstigen ein weiteres Anwachsen bereits großer Gebiete. Vergleichbar einem Graph-Growing-Verfahren wächst ein Gebiet in mehreren aufeinanderfolgenden Eliminationsschritten durch Einverleiben aller benachbarten kleineren Gebiete. Der Prozeß stoppt, sobald das Gebiet eine gewisse Größe erreicht hat und wird dann an einer anderen Stelle des Graphen erneut gestartet.

Sowohl durch einen Minimum-Deficiency-, als auch durch einen auf den Auswahlstrategien AMF, AMMF oder AMIND beruhenden Algorithmus wird die Entstehung von Gebieten mit einem großen Verhältnis von Inhalt zu Umfang begünstigt. Im ersten Fall geschieht dies durch Bildung wohl positionierter Gebiete, im zweiten Fall durch kurzfristige Konzentration des Wachstums auf ein bestimmtes Gebiet.

4.1.3 Top-down-Verfahren

Das am weitesten verbreitete Top-down-Verfahrenen ist der Nested-Dissection-Algorithmus (ND) von George und Liu (vgl. George [48] sowie George und Liu [51]). Wie bereits in Abschnitt 2.3.2 beschrieben besteht die grundsätzliche Idee des Verfahrens darin, eine Knotenmenge $S \subset V$ zu finden, durch deren Entnahme G in zwei Teilgraphen $G(B)$ und $G(W)$ zerfällt mit $V = S \cup B \cup W$ und $|B|, |W| \leq \alpha|V|$, $0 < \alpha < 1$. Im folgenden bezeichnen wir eine solche Partitionierung von G mit (S, B, W). Der Algorithmus wird dann rekursiv für jeden zusammenhängenden Teilgraphen von $G(B)$ und $G(W)$ aufgerufen bis ein Teilgraph weniger als n_0 Knoten enthält. Da die Knoten in S vor den Knoten in $B \cup W$ numeriert werden, baut der Nested-Dissection-Algorithmus den Eliminationsbaum von der Wurzel zu den Blättern auf.

Im Gegensatz zum Minimum-Degree- oder Minimum-Deficiency-Algorithmus stellt das Nested-Dissection-Verfahren mehr ein *Ordering-Framework* dar als einen exakt spezifizierten Algorithmus (vgl. auch Hendrickson und Rothberg [72]). Vor einer Implementierung müssen eine Reihe von Fragen beantwortet werden. Insbesondere muß geklärt werden, wie die Knotenseparatoren konstruiert werden sollen. Weitere, nicht minder wichtige Fragen sind:

- Wie groß soll n_0 gewählt werden, d. h. wieviele Knoten muß ein Teilgraph enthalten, damit für ihn ein Knotenseparator konstruiert wird?
- Wie soll α gewählt werden, d. h. wie wichtig ist eine balancierte Aufteilung der einzelnen Teilgraphen.

Im folgenden beschreiben wir kurz die wichtigsten, aus der Literatur bekannten Verfahren zur Konstruktion und Minimierung von Knotenseparatoren.

4.1.3.1 Konstruktion eines initialen Knotenseparators

Verfahren zur Graph-Partitionierung werden üblicherweise in zwei Klassen eingeteilt: *konstruktive Verfahren* und *iterative Verfahren*. Konstruktive Verfahren nehmen den Graphen G als Eingabe und bestimmen einen initialen Knotenseparator „from scratch“. Dieser Knotenseparator kann dann mit Hilfe eines iterativen Verfahrens weiter minimiert werden. Bevor wir uns im nächsten Abschnitt mit iterativen Verfahren beschäftigen, stellen wir zunächst zwei leistungsstarke Methoden zur Konstruktion von Knotenseparatoren genauer vor. Es handelt sich dabei um die Multilevel- und die Gebietszerlegungsmethode. Andere, in der Literatur erwähnte, konstruktive Verfahren beruhen auf der Spektral- (vgl. Barnard und Simon [17], Hendrickson und Leland [69] sowie Pothen et al. [113]) oder der Graph-Growing-Methode (vgl. Goehring und Saad [62]). Wir werden auf diese Methoden jedoch nicht näher eingehen.

Die Multilevel-Methode In den vergangenen Jahren wurden Multilevel-Verfahren sehr erfolgreich zur Konstruktion von *Kantenseparatoren* eingesetzt (vgl. z. B. Bui und Jones [24], Hendrickson und Leland [70], Karypis und Kumar [78] oder Monien et al. [104]). Ein Multilevel-Verfahren besteht aus drei Phasen. In der ersten Phase wird ausgehend von G eine Folge immer kleinerer Graphen konstruiert. Ziel ist dabei, die strukturellen Eigenschaften von G so weit wie möglich zu erhalten. Ist G beispielsweise ein quadratisches Gitter, so sollten auch die kleineren Graphen ein quadratisches Gitter darstellen. Üblicherweise wird zur Schrumpfung der Graphen eine Technik angwandt, die auf einer Kontraktion der Kanten beruht. Dabei werden die zwei zu einer Kante inzidente Knoten zu einem neuen Knoten zusammengefaßt. Der Vergröberungsprozeß stoppt, wenn in einem Graphen nur noch wenige Knoten vorhanden sind. Anschließend wird in Phase zwei für den kleinsten Graphen der Folge ein initialer Kantenseparator bestimmt. Im darauf folgenden Verfeinerungsprozeß (Phase drei) wird der Kantenseparator in den jeweils größeren Graphen übertragen und mit Hilfe eines iterativen Verfahrens wie z. B. der *Kernighan-Lin*- oder der *Fiduccia-Mattheyses-Heuristik* [42, 81] optimiert. Die Software-Pakete CHACO [68], METIS [79], PARTY [115] und WGPP [64] stellen eine Reihe unterschiedlicher Verfahren für die Schrumpfungs-, initiale Partitionierungs- und die Verfeinerungsphase bereit.

Zur Berechnung eines Nested-Dissection-Orderings benötigen wir jedoch Knoten- und keine Kantenseparatoren. Viele ältere Nested-Dissection-Verfahren bestimmen zunächst mit Hilfe der Multilevel-Methode einen Kantenseparator E' und leiten dann hieraus einen Knotenseparator S ab (vgl. z. B. Bui und Jones [24], Karypis und Kumar [78], Raghavan [116] sowie Schulze et al. [129]). S besteht dabei aus einer Teilmenge der zu E' inzidenten Knoten. Da die Größe eines Kantenseparators E' und die Größe eines aus E' abgeleiteten Knotenseparators S in keiner direkten Beziehung zueinander stehen, produzieren diese Verfahren oftmals sehr viel mehr Fill-

in als ein Minimum-Degree-Algorithmus. Dies liegt auch daran, daß Knoten mit hohem Grad in der Regel nicht inzident zu einer Kante aus E' sind. Die Knoten gehören damit auch nicht zu einem aus E' abgeleiteten Knotenseparator. Es gibt jedoch keinen Grund, warum Knoten mit hohem Grad bei der Konstruktion eines Knotenseparators unberücksichtigt bleiben sollten. In der Tat kann es bei der Berechnung eines Orderings durchaus sinnvoll sein, diese Knoten einem Separator zuzuordnen, da ihre Elimination eine große Clique erzeugt (vgl. z. B. Ashcraft und Liu [12] oder Hendrickson und Rothberg [72]).

In neueren Nested-Dissection-Verfahren werden daher die Knotenseparatoren direkt konstruiert. Dazu wird ein Multilevel-Ansatz benutzt, der dem zur Konstruktion von Kantenseparatoren sehr ähnlich ist (vgl. Gupta [65], Hendrickson und Rothberg [72] sowie Karypis und Kumar [79]). Auch hier wird der Graph mit Hilfe eines Kantenkontraktionsverfahrens geschrumpft. In dem so verkleinerten Graphen bestimmt man dann jedoch sofort einen Knotenseparator. Dieser initiale Knotenseparator wird in den darauf folgenden Verfeinerungsschritten mit Hilfe einer geeigneten Heuristik optimiert. Ashcraft und Liu [10] schlagen dazu eine leicht modifizierte Version des Fiduccia-Mattheyses-Algorithmus vor. Der Algorithmus wird im nächsten Abschnitt genauer vorgestellt.

Die Gebietszerlegungsmethode Im Gegensatz zu dem mehrstufigen Multilevel-Verfahren benutzen Ashcraft und Liu [12] einen zweistufigen Ansatz zur Konstruktion der Knotenseparatoren. In einem ersten Schritt wird die Knotenmenge V des Graphen G partitioniert in $V = \widehat{V} \cup D_1 \cup \ldots \cup D_r$ mit $\mathrm{adj}_G(D_i) \subset \widehat{V}$ für alle $1 \leq i \leq r$. Die Menge $\widehat{V}$ heißt *Multisektor*. Durch Entnahme der Knoten aus $\widehat{V}$ zerfällt G in die zusammenhängenden Teilgraphen $G(D_1), \ldots, G(D_r)$.

Nachdem eine Gebietszerlegung $(\widehat{V}, D_1, \ldots, D_r)$ bestimmt wurde, wird in einem zweiten Schritt jeder Menge D_i eine Farbe aus $\{\mathrm{BLACK}, \mathrm{WHITE}\}$ zugeordnet. Hierauf basierend werden die Knoten $v \in \widehat{V}$ des Multisektors wie folgt gefärbt:

$$\mathrm{color}(v) = \begin{cases} \mathrm{BLACK}, \text{ falls für alle } D \text{ mit } v \in \mathrm{adj}_G(D) \text{ gilt } \mathrm{color}(D) = \mathrm{BLACK} \\ \mathrm{WHITE}, \text{ falls für alle } D \text{ mit } v \in \mathrm{adj}_G(D) \text{ gilt } \mathrm{color}(D) = \mathrm{WHITE} \\ \mathrm{GRAY}, \text{ sonst.} \end{cases} \tag{4.9}$$

Sind also alle Mengen D mit $v \in \mathrm{adj}_G(D)$ schwarz (weiß) gefärbt, so erhält auch v die Farbe schwarz (weiß). Gibt es jedoch zwei Mengen D_i, D_j mit $v \in \mathrm{adj}_G(D_i), v \in \mathrm{adj}_G(D_j)$ und $\mathrm{color}(D_i) \neq \mathrm{color}(D_j)$, so erhält v die Farbe grau. Auf diese Weise induziert jede Färbung von $D_1, \ldots, D_r$ eine Menge $S \subset \widehat{V}$ von grau gefärbten Knoten.

Um zu garantieren, daß die grau gefärbten Knoten einen gültigen Knotenseparator darstellen, müssen die Knoten aus $\widehat{V}$ in geeigneter Form zu *Segmenten* zusammengefaßt werden. Wir werden auf die Segmentbildung in Abschnitt 4.2.1 noch einmal genauer eingehen und ihre Notwendigkeit an einem Beispiel illustrieren. Als Ergebnis erhält man eine Partitionierung $\widehat{V} =$

$V_1 \cup \ldots \cup V_s$, $V_i \cap V_j = \emptyset$, $i \neq j$, des Multisektors. Analog zu (4.9) kann nun jedem Segment $V_i \subset \widehat{V}$ die Farbe

$$\mathrm{color}(V_i) = \begin{cases} \mathrm{BLACK}, \text{ falls für alle } D \text{ mit } V_i \cap \mathrm{adj}_G(D) \neq \emptyset \text{ gilt } \mathrm{color}(D) = \mathrm{BLACK} \\ \mathrm{WHITE}, \text{ falls für alle } D \text{ mit } V_i \cap \mathrm{adj}_G(D) \neq \emptyset \text{ gilt } \mathrm{color}(D) = \mathrm{WHITE} \\ \mathrm{GRAY}, \text{ sonst} \end{cases} \tag{4.10}$$

zugeordnet werden. Jetzt stellt die Menge $S = \{v \in \widehat{V};\ \exists\, V_i \subset \widehat{V} \text{ mit } v \in V_i \text{ und } \mathrm{color}(V_i) = \mathrm{GRAY}\}$ einen gültigen Knotenseparator für alle Färbungen von $D_1, \ldots, D_r$ dar.

Zur Konstruktion einer Gebietszerlegung $(\widehat{V}, D_1, \ldots, D_r)$ benutzen Ashcraft und Liu eine randomisierte, auf Breitensuche basierende Greedy-Methode. Anschließend werden die Mengen $D_1, \ldots, D_r$ mit Hilfe einer speziellen Heuristik gefärbt. Ziel ist dabei eine Färbung zu finden, die die Größe des induzierten Knotenseparators S minimiert. Zum Schluß wird S mit Hilfe der weiter unten beschriebenen *Netzwerk-Fluß-Technik* geglättet. Dazu werden Netzwerke konstruiert, die aus bis zu sieben Schichten bestehen (vgl. Ashcraft und Liu [12, 13]).

4.1.3.2 Minimierung eines Knotenseparators

Genaugenommen besteht das Ziel einer iterativen Verbesserungsheuristik nicht in der Minimierung eines Knotenseparators S, sondern in der Optimierung einer Partitionierung (S, B, W). In die Bewertung einer Partitionierung gehen normalerweise die Größe des Separators und die *Balance* ein. Um zu entscheiden, ob eine Partitionierung (S, B, W) besser oder schlechter ist als eine zweite Partitionierung (S', B', W'), wird eine Bewertungsfunktion F benötigt. Diese Funktion gewichtet die in der Regel in Konflikt stehenden Zielkriterien „Minimierung des Knotenseparators“ und „Maximierung der Balance“. Die Wahl einer geeigneten Bewertungsfunktion ist nicht einfach und muß im Kontext des übergeordneten Ordering-Prozesses gesehen werden (vgl. auch Rothberg [123]).

Die Optimierung einer Partitionierung (S, B, W) geschieht i. allg. durch die Berechnung eines neuen Knotenseparators S' mit der Eigenschaft $|S'| \leq |S|$. Man hofft, daß die Minimierung des Knotenseparators S auch zu einer Verbesserung der Partitionierung (S, B, W) führt. Grundsätzlich unterscheided man zwischen zwei Minimierungstechniken (vgl. Ashcraft [4]): *Heuristische Methoden*, die in mehreren Schritten versuchen durch Verschieben einzelner Knoten einen besseren Separator S' zu finden und *direkte Methoden*, die in einem Schritt durch Verschieben einer ganzer Knotenmenge den besten Separator in der Nachbarschaft von S bestimmen.

Heuristische Methoden Die Algorithmen von Kernighan, Lin [81] und Fiduccia, Mattheyses [42] gehören zu den am häufigsten verwendeten Verfahren zur Minimierung von Kantenseparatoren. Beide Algorithmen bestehen aus zwei ineinander geschachtelten Schleifen. Der

Kernighan--Lin-Algorithmus wählt in der inneren Schleife Paare von zu verschiedenen Komponenten gehörenden Knoten aus, vertauscht ihre Position logisch und sperrt sie. Diese logischen Vertauschungen werden so lange wiederholt bis alle Knoten gesperrt sind. Die Besonderheit des Algorithmus besteht darin, während der logischen Vertauschungen eine Verschlechterung der Bisektionsweite zuzulassen, in der Hoffnung zu einem späteren Zeitpunkt einen viel besseren Kantenseparator zu finden. Nachdem alle Knoten gesperrt sind, wird diejenige Sequenz von logischen Vertauschungen berechnet, die zu dem kleinsten Kantenseparator führt. Ist dieser Kantenseparator besser als der ursprüngliche, so werden die entsprechenden Knoten physisch vertauscht und die innere Schleife wird erneut gestartet.

Zur Beschleunigung des Algorithmus schlagen Fiduccia und Mattheyses vor, in der inneren Schleife nur Verschiebungen von Knoten statt paarweise Vertauschungen zu betrachten. Die innere Schleife kann dann in Zeit $O(e)$ ausgeführt werden, wobei e die Anzahl der Kanten in G bezeichnet. In state-of-the-art Implementierungen werden viele zusätzliche „Tricks“ angewandt, um die Laufzeit und Effektivität des Kernighan-Lin-Algorithmus weiter zu verbessern (vgl. CHACO [68], METIS [79], PARTY [115] und WGPP [64]).

Die Vorgehensweise des Fiduccia-Mattheyses-Algorithmus kann sehr einfach auf Knotenseparatoren übertragen werden. Der *Vertex-Fiduccia-Mattheyses-Algorithmus* wählt in jeder Iteration der inneren Schleife einen Knoten $v \in S$ aus, verschiebt ihn nach B oder W und sperrt ihn. Bei einer Verschiebung nach B müssen alle Knoten $u \in \mathrm{adj}_G(v) \cap W$ in den Separator S eingefügt werden. Daher ändert sich das Gewicht des Separators S um den Wert (wir nehmen an, daß G gewichtet ist)

$$\mathrm{gain}_{S \to B}(v) = |v| - \sum_{u \in \mathrm{adj}_G(v) \cap W} |u|.$$

Analog berechnet man $\mathrm{gain}_{S \to W}(v)$, so daß jedem Knoten $v \in S$ zwei Gain-Werte zugeordnet sind. Basierend auf diesen Werten kann nun eine Sequenz von Knotenverschiebungen bestimmt werden, die zu einem minimalen Separator führt.

Wurde v nach B verschoben, so müssen die Gain-Werte aller Knoten $u \in \mathrm{adj}_G(v) \cap W$ berechnet werden. Darüber hinaus müssen die Gain-Werte aller Knoten $w \in S$ neu berechnet werden, die zu einem Knoten $u \in \mathrm{adj}_G(v) \cap W$ benachbart sind. Da jeder Knoten in der inneren Schleife nur einmal verschoben wird, verursacht die Berechnung der Gain-Werte insgesamt einen Aufwand von $O(e)$ (vgl. Hendrickson und Rothberg [72]). Benutzt man zur Sortierung der Gain-Werte einen *Heap*, so kann die innere Schleife des Vertex-Fiduccia-Mattheyses-Algorithmus in Zeit $O(e \log n)$ ausgeführt werden. Dabei bezeichnet n die Anzahl der Knoten in G. Man beachte, daß die höhere Laufzeit des Vertex-Fiduccia-Mattheyses-Algorithmus einzig und allein auf die Knotengewichte zurückzuführen ist, da in diesem Fall die Gain-Werte nicht wie von Fiduccia und Mattheyses vorgeschlagen mit Hilfe von *Buckets* sortiert werden können.

Direkte Methoden Ein völlig anderer Ansatz zur Minimierung von Knotenseparatoren wurde von Liu [94] vorgeschlagen. Zur Beschreibung des Verfahrens benötigen wir einige zusätzliche Definitionen aus der Graphentheorie: Ein ungerichteter Graph $G = (V, E)$ heißt *bipartit*, falls es eine Partitionierung $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, gibt, so daß für jede Kante aus E einer der Knoten in V_1 und der andere in V_2 liegt. Eine Menge $M \subset E$ heißt *Matching*, wenn in dem Graphen (V, M) jeder Knoten höchstens den Grad eins besitzt. Ein *maximum Matching* ist ein Matching maximaler Kardinalität. Ein Knoten $v \in V$ heißt *frei*, wenn er nicht inzident zu einer Kante aus M ist. Ein *alternierender Weg* ist ein einfacher Weg in G, der an einem freien Knoten startet und dessen Kanten abwechselnd zu $E - M$ und zu M gehören. Ein alternierender Weg, der auch an einem freien Knoten endet, heißt *erweiternder Weg*. Schließlich heißt eine Menge $U \subset V$ *Vertex-Cover*, falls jede Kante aus E zu mindestens einem Knoten aus U inzident ist. Ein *minimum Vertex-Cover* ist ein Vertex-Cover minimaler Kardinalität.

Sei G ein ungewichteter Graph und (S, B, W) eine Partitionierung von G. Es gelte $|B| \geq |W|$. Wir betrachten den durch S und $B' = \mathrm{adj}_G(S) \cap B$ induzierten bipartiten Graphen H. Jedes Vertex-Cover in H definiert einen Knotenseparator in G. Ziel des von Liu beschriebenen Algorithmus ist deshalb, ein minimum Vertex-Cover in H zu finden. Dazu wird ausgehend von dem Vertex-Cover S nach einer Menge $Z \subset S$ mit $|Z| - |\mathrm{adj}_H(Z)| > 0$ gesucht. Es gilt nämlich: $\widetilde{S} := (S - Z) \cup \mathrm{adj}_H(Z)$ ist wieder ein Vertex-Cover und $|\widetilde{S}| < |S|$. Zur Bestimmung einer solchen Menge Z wird ein maximum Matching M in H berechnet. Die Knoten aus S können dann in drei disjunkte Mengen S_I, S_X, S_R aufgeteilt werden mit

$$
\begin{aligned}
S_I &= \{s \in S;\ s \text{ ist über einen altern. Weg von einem freien Knoten aus } S \text{ erreichbar}\},\\
S_X &= \{s \in S;\ s \text{ ist über einen altern. Weg von einem freien Knoten aus } B' \text{ erreichbar}\},\\
S_R &= S - (S_I \cup S_X).
\end{aligned}
$$

Die Aufteilung $S = S_I \cup S_X \cup S_R$ heißt *Dulmage-Mendelsohn-Dekomposition*. Nach [39] ist die Aufteilung unabhängig von der Wahl des maximum Matchings. Darüber hinaus gilt (vgl. Ashcraft und Liu [13] sowie Pothen und Fan [112])

$$
\begin{aligned}
&|S_I| - |\mathrm{adj}_H(S_I)| > 0, \text{ falls } S_I \neq \emptyset,\\
&|S_I| - |\mathrm{adj}_H(S_I)| = |S_I \cup S_R| - |\mathrm{adj}_H(S_I \cup S_R)|.
\end{aligned}
$$

Falls es also in S einen freien Knoten gibt ($S_I \neq \emptyset$), so erhält man ein kleineres Vertex-Cover $\widetilde{S}$, indem S_I durch $\mathrm{adj}_H(S_I)$ oder $S_I \cup S_R$ durch $\mathrm{adj}_H(S_I \cup S_R)$ ersetzt wird. Abbildung 4.2 verdeutlicht dies an einem Beispiel. Nach der Substitution sind alle Knoten des Vertex-Covers $\widetilde{S}$ zu einer Kante des Matchings inzident. Die Kardinalität von $\widetilde{S}$ entspricht also der Kardinalität des maximum Matchings. Nach König [86] ist daher $\widetilde{S}$ bereits ein minimum Vertex-Cover.

Die Idee, Knotenseparatoren mittels eines Vertex-Covers zu bestimmen, geht eigentlich auf Leiserson und Lewis [88] zurück. Im Gegensatz zu Liu berechnen sie jedoch ein *minimales* Vertex-Cover in H. Nach Hopcroft und Karp [75] kostet die Berechnung eines maximum Matchings und damit die Berechnung eines *minimum* Vertex-Covers nur $O(e_H\sqrt{n_H})$ Zeiteinheiten. Dabei

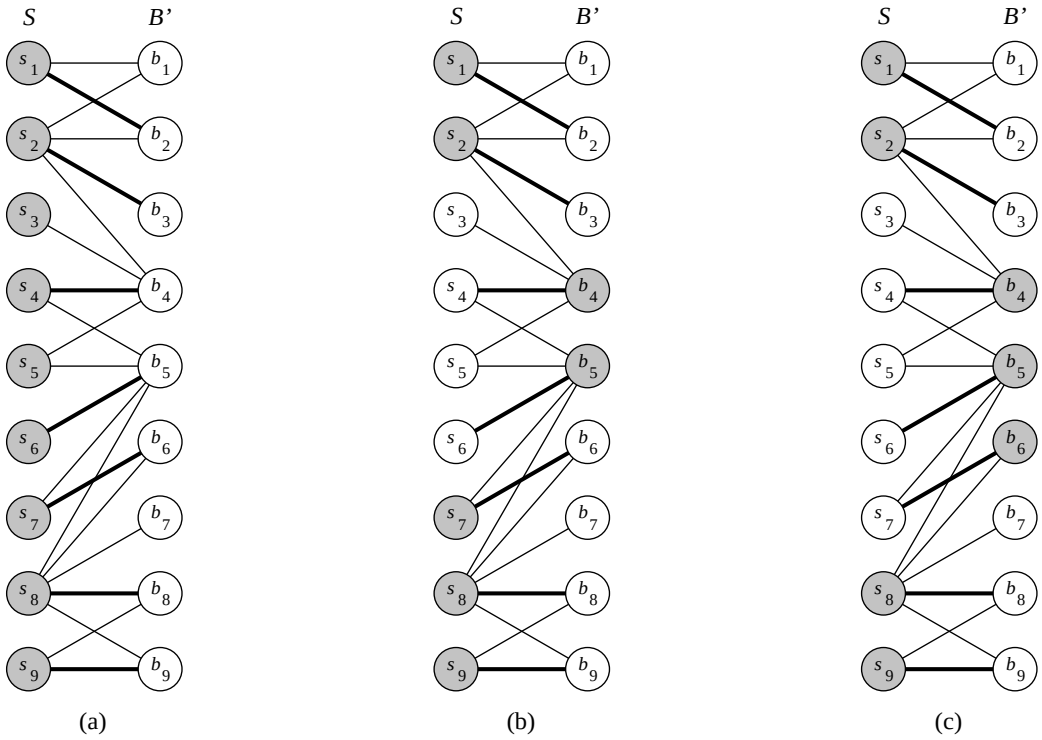


Abb. 4.2: Bipartiter Graph H mit maximum Matching und Vertex-Cover. Die Matching-Kanten sind in Fettdruck dargestellt, und die Knoten des Vertex-Covers sind grau gefärbt. (a) zeigt das initiale Vertex-Cover. Es gilt $S_I = \{s_3, s_5, s_4, s_6\}$, $S_X = \{s_1, s_2, s_8, s_9\}$ und $S_R = \{s_7\}$. (b) und (c) zeigen das Vertex-Cover $\widetilde{S}$ nach Substitution von S_I mit $\mathrm{adj}_H(S_I)$, bzw. nach Substitution von $S_I \cup S_R$ mit $\mathrm{adj}_H(S_I \cup S_R)$.

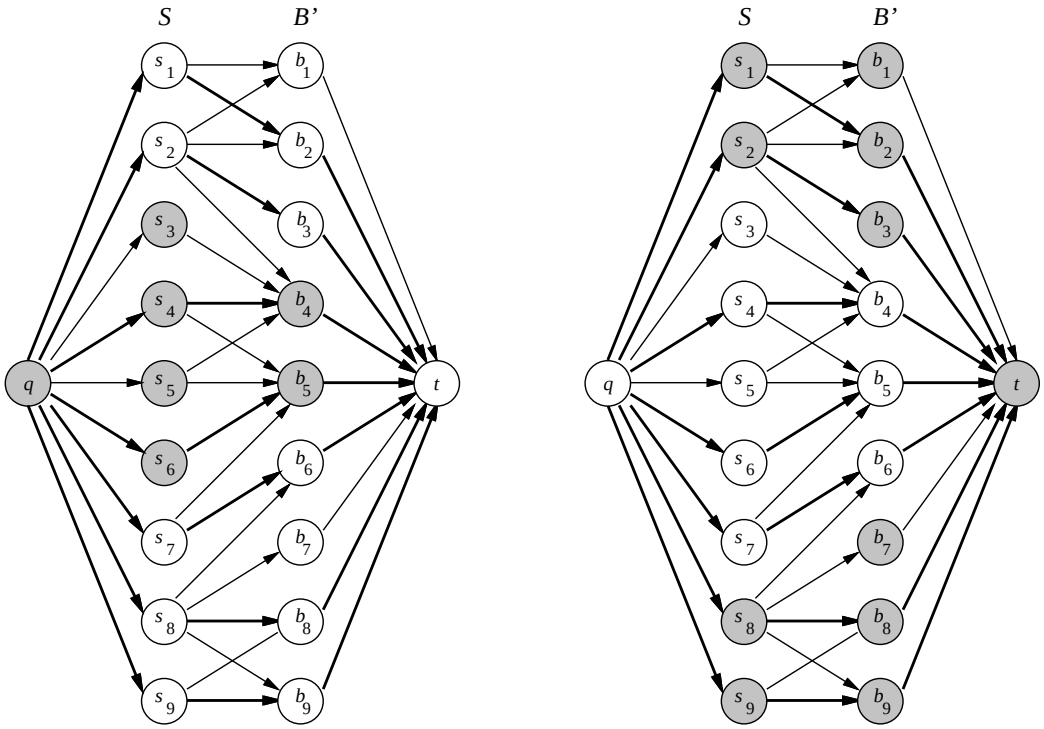


Abb. 4.3: Zwei Kopien des aus dem bipartiten Graphen H konstruierten Netzwerks N. Der Einfachheit halber sei angenommen, daß die Kanten (q, s), $s \in S$, und (b, t), $b \in B'$, die Kapazität eins haben. Alle Kanten über die der berechnete maximale Fluß fließt sind in Fettdruck dargestellt. In der linken Kopie sind alle Knoten, die ausgehend von q über benutzbare Kanten erreicht werden können, grau gefärbt. Zu dem minimal gewichteten Vertex-Cover gehören alle nicht grau gefärbten Knoten aus S und alle grau gefärbten Knoten aus B'. Die rechte Kopie zeigt alle Knoten, die ausgehend von t über benutzbare Kanten erreichbar sind. Zu dem minimal gewichteten Vertex-Cover gehören jetzt alle grau gefärbten Knoten aus S und alle nicht grau gefärbten Knoten aus B'.

bezeichnet e_H die Anzahl der Kanten und n_H die Anzahl der Knoten in H. Deswegen wird in der Praxis Lius Variante der Vorzug gegeben.

Ist G gewichtet, so muß ein minimal gewichtetes Vertex-Cover bestimmt werden. Dies geschieht mit Hilfe eines *Netzwerk-Fluß-Algorithmus*. Dazu wird aus dem bipartiten Graphen H wie folgt ein Netzwerk N konstruiert:

1. Füge zwei ausgezeichnete Knoten q *(Quelle)* und t *(Senke)* mit den Kanten (q, s), $s \in S$, und (b, t), $b \in B'$ ein. Die Kante (q, s) erhält die *Kapazität* $c(q, s) = |s|$ und die Kante (b, t) die Kapazität $c(b, t) = |b|$.
2. Alle Kanten (s, b) zwischen S und B' erhalten die Kapazität $c(s, x) = \infty$ und alle Kanten (b, s) zwischen B' und S werden gelöscht.

In N wird zunächst ein *maximaler Fluß* bestimmt. Anschließend wird mit Hilfe einer Breitensuche, die an dem Knoten q (oder t) startet und über *benutzbare* Kanten[‡] führt, ein *minimaler Schnitt* bestimmt. Zu dem optimalen, d. h. minimal gewichteten Vertex-Cover gehören dann alle Knoten aus $S \cup B'$, die inzident zu einer über den Schnitt führenden Kante sind. Abbildung 4.3 verdeutlicht die Vorgehensweise an einem Beispiel.

Mit Hilfe des *Dinic-Algorithmus* kann ein maximaler Fluß und damit ein minimal gewichtetes Vertex-Cover in Zeit $O(e_H n_H^2)$ berechnet werden (vgl. Ahuja et al. [16]). Im gewichteten Fall verursacht die Berechnung eines optimalen Vertex-Covers also sehr viel mehr Aufwand als im ungewichteten Fall. Die Netzwerk-Fluß-Technik hat jedoch gegenüber der Dulmage-Mendelsohn-Dekomposition einen entscheidenden Vorteil: sie kann auch zur Bestimmung eines optimalen Vertex-Covers in einem aus mehr als zwei Schichten bestehenden Netzwerk verwendet werden (vgl. Ashcraft und Liu [13]). Damit ist es möglich eine größere Nachbarschaft von S zu durchsuchen. Die oben vorgestellte Gebietszerlegungsmethode macht hiervon Gebrauch.

Abschließend sei angemerkt, daß mit der Vertex-Cover-Technik auch ein Knotenseparator S aus einem Kantenseparator E' abgeleitet werden kann. Dazu betrachtet man den durch E' induzierten bipartiten Graphen.

4.1.4 Multisection-Verfahren

In einem Bottom-up-Verfahren wie Minimum-Degree wird der als nächstes zu eliminierende Knoten mit Hilfe einer *lokalen* Knotenauswahlstrategie bestimmt. Der Algorithmus kann nicht voraussehen, welchen Einfluß die Elimination eines Knotens auf die Bildung zukünftiger Gebiete hat. Diese Schwäche nutzen Berman und Schnittger [21] bei der Konstruktion ihres suboptimalen Minimum-Degree-Orderings für quadratische Gitter aus. Ihr Ordering ist so konzipiert, daß Gebiete mit einem „fraktalen“ (also einem sehr großen) Rand entstehen. Demgegenüber ist Georges

[‡]Bezeichne $f(e)$ den Fluß über die Kante e. Dann heißt $e = (u, v)$ benutzbar, falls e eine Kante aus N ist mit $c(u, v) - f(u, v) > 0$, oder falls (v, u) eine Kante aus N ist mit $f(v, u) > 0$.

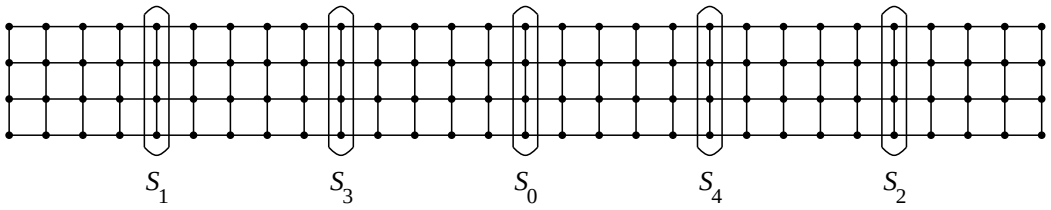


Abb. 4.4: Die obersten Separatoren in einem rechteckigen Gitter. Zuerst wurde der Separator S_0 konstruiert (Ebene 0), danach die Separatoren S_1, S_2 (Ebene 1) und (S_3, S_4) (Ebene 2).

Nested-Dissection-Ordering für $n \times n$-Gitter bis auf einen konstanten Faktor optimal. Werden die Separatoren entsprechend ihrer Rekursionstiefe eliminiert, so entstehen Gebiete mit einem „glatten“ Rand.

Die Schwäche eines Top-down-Verfahrens wie Nested-Dissection kann am besten anhand eines $h \times n$-Gitters mit großem *Aspekt-Ratio* illustriert werden. Für diese Gitter produziert ein Minimum-Degree-Ordering weniger Fill-in als ein Nested-Dissection-Ordering. Im Extremfall $h = 1$ liefert der Minimum-Degree-Algorithmus sogar ein perfektes Ordering, wohingegen Nested-Dissection einen Fill-in von $O(n)$ erzeugt. Da auch hier die besten Knotenseparatoren in die Berechnung des Orderings eingehen (die besten in Bezug auf Größe des Separators und Balance der Partitionierung), kann der hohe Fill-in nicht der Qualität der Separatoren angelastet werden. Vielmehr ist die *Numerierung* der Separatoren für den hohen Fill-in verantwortlich. Abbildung 4.4 zeigt die obersten Separatoren eines rechteckigen Gitters. Werden die Separatoren entsprechend ihrer Rekursionstiefe numeriert, so wird S_3 vor S_1 und S_0 aus dem Gitter entfernt. Dadurch entsteht ein Element, auf dessen Rand $2h$ Knoten liegen. Werden die Separatoren jedoch wie bei einem Profil-Ordering von links nach rechts oder wie bei einem Minimum-Degree-Ordering von beiden Seiten zur Mitte numeriert, so besitzen alle Elemente einen Rand der Größe h.[§]

In Multisection-Verfahren [14] werden die Knotenseparatoren nur noch zur *Aufspaltung* des Graphen G in mehrere Teilgraphen benutzt. Als Ergebnis dieser Aufspaltung erhält man eine Gebietszerlegung $(\Phi, \Omega_1, \ldots, \Omega_q)$, wobei der Multisektor Φ alle Knoten enthält, die zu einem Separator gehören. Wir benutzen griechische Buchstaben, um die Gebietszerlegung $(\Phi, \Omega_1, \ldots, \Omega_q)$ von der in Abschnitt 4.1.3 vorgestellten Gebietszerlegung $(\widehat{V}, D_1, \ldots, D_r)$ zu unterscheiden. Wie von Ashcraft und Liu [12] beschrieben, kann die zweite Gebietszerlegung benutzt werden, um die Knotenseparatoren der ersten zu konstruieren.

Die fundamentale Eigenschaft eines Multisection-Orderings besteht darin, daß die Knoten in den Mengen $\Omega_1, \ldots, \Omega_q$ vor den Knoten aus Φ numeriert werden. Die Eliminationssequenz ist also beschränkt, d. h. einige Knoten müssen vor anderen Knoten eliminiert werden. Zur Nu-

[§] An dieser Stelle sei angemerkt, daß der nur aus den vertikalen Separatoren bestehende Eliminationsgraph bereits chordal ist. Ein Profil- oder ein Minimum-Degree-Algorithmus findet ein perfektes Ordering für diesen Graphen.

merierung der Knoten aus Φ betrachtet man den Eliminationsgraphen $G_\Phi = (\Phi, E_\Phi)$ mit (vgl. Formel (4.3))

$$E_\Phi = (E \cap (\Phi \times \Phi)) \cup \bigcup_{i=1}^{q} (\mathrm{adj}_G(\Omega_i) \times \mathrm{adj}_G(\Omega_i)).$$

Für die Numerierung der Knoten in $\Omega_1, \ldots, \Omega_q$ und für die Numerierung der Knoten des *Schur-Komplement Graphen* G_Φ können unterschiedliche Ordering-Verfahren benutzt werden. Ein Multisection-Ordering ist deswegen durch die folgenden Angaben spezifiziert (vgl. auch Ashcraft und Liu [14]):

1. Verfahren zur Konstruktion der Gebietszerlegung $(\Phi, \Omega_1, \ldots, \Omega_q)$.
2. Knotenauswahlverfahren zur Elimination der Knoten in $\Omega_1, \ldots, \Omega_q$.
3. Knotenauswahlverfahren zur Elimination der Knoten in Φ.

Ein Multisection-Ordering wird mit $\mathrm{MS}(\mathrm{ord}_1, \mathrm{ord}_2)$ bezeichnet. Dabei gibt ord_1 das Knotenauswahlverfahren zur Elimination der Knoten in $\Omega_1, \ldots, \Omega_q$ an und ord_2 das Knotenauswahlverfahren zur Elimination der Knoten in Φ. Viele der in der Literatur beschriebenen Ordering-Verfahren sind Multisection-Verfahren. Beispielsweise ist das in 3.1 beschriebene One-Way-Dissection-Verfahren von George [49] ein Multisection-Verfahren vom Typ $\mathrm{MS}(\mathrm{Profil}, \mathrm{Profil})$. Das ebenfalls in 3.1 beschriebene Local-Nested-Dissection-Verfahren von Bhat et al. [19] ist ein Multisection-Verfahren vom Typ $\mathrm{MS}(\mathrm{ND}, \mathrm{Profil})$. Weitere wichtige Multisection-Verfahren sind:

- $\mathrm{MS}(\mathrm{MD}, \mathrm{ND})$

  In den meisten Fällen ist die Konstruktion der für ein Nested-Dissection-Ordering benötigten Knotenseparatoren sehr viel aufwendiger als die Berechnung eines Bottom-up-Orderings. In der Praxis wird daher die Konstruktion der Knotenseparatoren bereits nach wenigen Rekursionsstufen gestoppt, d. h. der Parameter n_0 wird auf einen relativ hohen Wert gesetzt. Ein solches Ordering bezeichnet man deswegen auch als *unvollständiges (incomplete)* Nested-Dissection-Ordering (vgl. George et al. [56]). Da in diesem Fall die Teilgraphen sehr groß sind, werden die Knoten nicht mehr in beliebiger Reihenfolge, sondern mit Hilfe eines Minimum-Degree-Verfahrens numeriert. Dabei ist das Minimum-Degree-Verfahren so modifiziert, daß die Knoten aus Φ zwar in die Berechnung der Knotengrade eingehen, so jedoch nicht eliminiert werden (vgl. auch Liu [93]). Man erhält so ein Multisection-Ordering vom Typ $\mathrm{MS}(\mathrm{MD}, \mathrm{ND})$.

- $\mathrm{MS}(\mathrm{MD}, \mathrm{MD})$

  Wie an dem $h \times n$-Gitter aus Abbildung 4.4 gesehen, kann die Güte eines unvollständigen Nested-Dissection-Orderings weiter verbessert werden, wenn die Knotenseparatoren nicht entsprechend ihrer Rekursionstiefe, sondern mit Hilfe eines Minimum-Degree-Algorithmus numeriert werden. Man erhält so ein Multisection-Ordering vom Typ $\mathrm{MS}(\mathrm{MD}, \mathrm{MD})$.

Mit Hilfe eines $\mathrm{MS}(\mathrm{MD}, \mathrm{MD})$-Orderings können die Schwächen der Bottom-up- und der Top-down-Methode am besten ausgeglichen werden. $\mathrm{MS}(\mathrm{MD}, \mathrm{MD})$-Orderings gehen zurück auf die unabhängigen Arbeiten von Ashcraft und Liu [11] und Rothberg [122].

Einen völlig anderen Multisection-Ansatz beschreiben Bornstein et al. [23]. In ihrem *Less-Parallel-Nested-Dissection-Algorithmus* (LPND) sind die Konstruktion und die Numerierung der Separatoren nicht mehr klar voneinander getrennt. In jedem Rekursionsschritt wird nach Berechnung eines minimalen Knotenseparators S entschieden, in welcher Reihenfolge die durch Entnahme von S entstehenden zusammenhängenden Teilgraphen $G_1, \ldots, G_r$ abgearbeitet werden. Dazu wird für jede Komponente G_i berechnet, wieviele Knoten aus bereits konstruierten Separatoren zu Knoten aus G_i benachbart sind. Gibt es mehrere Komponenten, für die dieser Wert maximal ist, so werden $G_1, \ldots, G_r$ wie bei einem normalen Nested-Dissection-Schritt in beliebiger Reihenfolge abgearbeitet. Gibt es jedoch genau eine Komponente G_{i^*}, für die der Wert maximal ist, so wird diese als letztes abgearbeitet (die anderen Komponenten kann man wieder in beliebiger Reihenfolge abarbeiten). Dabei wird G_{i^*} wie beim Generalized-Nested-Dissection-Algorithmus vor dem rekursiven Aufruf um die Knotenmenge S erweitert. Im Gegensatz zum Generalized-Nested-Dissection-Algorithmus werden die Knoten aus S jedoch erst in den nun folgenden Rekursionsstufen numeriert. Daher müssen alle Fill-Kanten in S eingefügt werden, die durch Elimination der in den anderen Komponenten enthaltenen Knoten entstehen. Genau dieser Schritt verursacht einen erheblichen Aufwand, weswegen das Verfahren in der Praxis nur beschränkt einsetzbar ist. Der LPND-Algorithmus besitzt eine weitere Besonderheit: Die Knoten des in einem normalen Nested-Dissection-Schritt konstruierten Separators S werden nicht in beliebiger Reihenfolge, sondern mit Hilfe eines Minimum-Degree-Algorithmus numeriert. Hierdurch und durch die Festlegung einer Reihenfolge bei der Abarbeitung der Komponenten $G_1, \ldots, G_r$ ist garantiert, daß in einem chordalen Graphen immer eine perfekte Eliminationssequenz gefunden wird (die Minimalität der Separatoren vorausgesetzt).

4.2 Ein verbessertes Multisection-Verfahren

Im letzten Abschnitt haben wir zwei Kombinationsstufen von Minimum-Degree und Nested-Dissection vorgestellt: $\mathrm{MS}(\mathrm{MD}, \mathrm{ND})$ und $\mathrm{MS}(\mathrm{MD}, \mathrm{MD})$. Insbesondere Multisection-Orderings vom Typ $\mathrm{MS}(\mathrm{MD}, \mathrm{MD})$ sind sehr *robust* und können die Schwächen der Bottom-up- und der Top-down-Methode am besten ausgleichen. Unser Ziel ist es, eine noch engere Kopplung zwischen beiden Methoden zu erreichen. Dazu zeigen wir, wie Bottom-up-Techniken zur Konstruktion besserer Knotenseparatoren und wie Knotenseparatoren zur Konstruktion besserer Bottom-up-Orderings genutzt werden können (vgl. auch Schulze [132]).

Widmen wir uns zunächst dem ersten Vorhaben. Abbildung 4.5 zeigt einen Multisection-Algorithmus, der dem ursprünglich von Ashcraft und Liu [14] beschriebenen Algorithmus sehr

MULTISECTION$(\mathrm{ord}_\Phi, \mathrm{ord}_1, \mathrm{ord}_2)$
01: Determine a domain decomposition $(\Phi, \Omega_1, \ldots, \Omega_q)$ of G by a recursive bisection process. Use node selection strategy ord_Φ to construct the vertex separators.
02: **for** each set Ω_i **do**
03: Eliminate all vertices in Ω_i using node selection strategy ord_1.
04: Construct the elimination graph G_Φ.
05: Number the vertices in G_Φ using node selection strategy ord_2.

Abb. 4.5: Funktion MULTISECTION.

ähnlich ist. Auch hier wird der Multisektor Φ mit Hilfe eines rekursiven Bisektionsprozesses konstruiert. Dabei kommt jedoch eine neuartige Multilevel-Methode zum Einsatz. Im Gegensatz zu herkömmlichen Multilevel-Methoden, die auf einem Kantenkontraktionsverfahren basieren, generiert die neue Methode eine Folge von Quotientengraphen $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_t$. Die Knotenauswahlstrategie ord_Φ dient zur Bestimmung der beim Übergang von $\mathcal{G}_{k-1}$ nach $\mathcal{G}_k$, $1 \leq k \leq t$ zu eliminierenden Variablen. Nachdem so eine Gebietszerlegung $(\Phi, \Omega_1, \ldots, \Omega_q)$ berechnet wurde, werden die Knoten in den Mengen $\Omega_1, \ldots, \Omega_q$ entsprechend der Knotenauswahlstrategie ord_1 und die Knoten des Schur-Komplement Graphen G_Φ entsprechend der Auswahlstrategie ord_2 eliminiert.

In Anlehnung an das Klassifizierungsschema von Ashcraft und Liu, ist unser Multisection-Ordering durch die folgenden Angaben spezifiziert:

1. Knotenauswahlverfahren zur Schrumpfung des Graphen G.
2. Knotenauswahlverfahren zur Elimination der Knoten in $\Omega_1, \ldots, \Omega_q$.
3. Knotenauswahlverfahren zur Elimination der Knoten in Φ.

Unsere Multisection-Orderings werden daher mit $\mathrm{MS}(\mathrm{ord}_\Phi, \mathrm{ord}_1, \mathrm{ord}_2)$ bezeichnet.

Zur Realisierung des zweiten Vorhabens verallgemeinern wir das in Abbildung 4.5 präsentierte Multisection-Verfahren. In dem verallgemeinerten Verfahren dienen die Knotenseparatoren als ein Gerüst zur Generierung mehrerer Bottom-up-Orderings. Die Idee ist, einige Separatoren aus Φ entsprechend ihrer Rekursionstiefe und einige mit Hilfe eines Bottom-up-Verfahrens zu numerieren. Das verallgemeinerte Verfahren heißt deswegen *dreistufiges Multisection*.

Dieser Abschnitt ist wie folgt aufgebaut: In 4.2.1 stellen wir die neue Multilevel-Methode zur Konstruktion der Knotenseparatoren vor. Des weiteren präsentieren wir einige Knotenauswahlstrategien zur Schrumpfung eines Graphen. In 4.2.2 beschreiben wir die iterative Verbesserungsheuristik, die in der Verfeinerungsphase unserer Multilevel-Methode benutzt wird. Wir zeigen, daß die Laufzeit der Heuristik asymptotisch der des Vertex-Fiduccia-Mattheyses-Algorithmus entspricht. Schließlich präsentieren wir in 4.2.3 das dreistufige Multisection-Verfahren.

4.2.1 Konstruktion der Knotenseparatoren

Betrachten wir noch einmal die in Abschnitt 4.1.3 vorgestellte Gebietszerlegungsmethode. Im Gegensatz zur Multilevel-Methode handelt es sich um einen zweistufigen Ansatz zur Konstruktion von Knotenseparatoren. Dabei wird in der ersten Stufe die Knotenmenge V des Graphen G partitioniert in $V = \widehat{V} \cup D_1 \cup \ldots \cup D_r$. Es gibt einen engen Zusammenhang zwischen den Mengen $D_1, \ldots, D_r$ und den innerhalb eines Bottom-up-Verfahrens gebildeten Gebieten. Sei $\widehat{U} = \bigcup_{i=1}^{r} D_i$. Da alle Graphen $G(D_i)$ zusammenhängende Teilgraphen von G sind mit $\mathrm{adj}_G(D_i) \subset \widehat{V}$, stellt jede Menge D_i ein Gebiet bezüglich der eliminierten Knoten $\widehat{U}$ dar. Der Multisektor $\widehat{V}$ enthält dabei alle nicht eliminierten Knoten, und es gilt $V = \widehat{V} \cup \widehat{U}$. Eine Gebietszerlegung $(\widehat{V}, D_1, \ldots, D_r)$ kann daher durch einen Eliminationsprozeß konstruiert werden, der dem zur Berechnung eines Bottom-up-Orderings sehr ähnlich ist. In diesem Fall erhält man eine Folge von Quotientengraphen $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_t$, wobei der letzte Graph der Folge gerade die gesuchte Gebietszerlegung repräsentiert. Aber auch alle vorherigen Graphen stellen eine Gebietszerlegung dar und können deshalb zur Verfeinerung des mit Hilfe von $\mathcal{G}_t$ bestimmten Knotenseparators benutzt werden. Man erhält so einen Multilevel-Algorithmus, der im folgenden genauer beschrieben wird (vgl. auch Schulze [133]).

4.2.1.1 Beschreibung der neuen Multilevel-Methode

Unsere Multilevel-Methode beginnt mit der Konstruktion eines initialen Quotientengraphen $\mathcal{G}_0$. Basierend auf $\mathcal{G}_0$ wird dann eine Folge von Quotientengraphen $\mathcal{G}_1, \ldots, \mathcal{G}_t$ generiert, wobei $\mathcal{G}_k$ aus $\mathcal{G}_{k-1}$, $1 \leq k \leq t$, durch die Elimination von Variablen entsteht. Alle Quotientengraphen $\mathcal{G}_k = (\mathcal{X}_k, \mathcal{E}_k)$ besitzen die folgenden zwei Eigenschaften (dies schließt $\mathcal{G}_0$ ein, d. h. $0 \leq k \leq t$):

(1) Sei $\mathcal{V}_k = \{V_1, \ldots, V_s\}$ die Menge der Variablen von $\mathcal{G}_k$, und sei $\mathcal{S}_k \subset \mathcal{V}_k$ eine Menge von Variablen, durch deren Entnahme $\mathcal{G}_k$ in zwei oder mehrere Komponenten zerfällt. Falls $\mathcal{S}_k$ die Variablen $V_{p_1}, \ldots, V_{p_t}$ enthält, so ist $S = V_{p_1} \cup \ldots \cup V_{p_t}$ ein Separator in G.

(2) $\mathcal{G}_k$ ist bipartit, d. h. es gibt nur Domain-Vertex-Kanten.

Die Eigenschaft (1) ist entscheidend für die Effektivität unserer Multilevel-Methode. Durch sie wird die Minimierung eines Separators $\mathcal{S}_k$ für $\mathcal{G}_k$ in Verbindung gebracht zur Minimierung eines Separators S für G. Zur Konstruktion eines Separators $\mathcal{S}_k$ benutzen wir die von Ashcraft und Liu vorgestellte Färbungstechnik. Dazu erhält jedes Element $D \in \mathcal{D}_k$ eine Farbe $\mathrm{color}(D) \in \{\mathrm{BLACK}, \mathrm{WHITE}\}$. Hieraus leitet sich analog zu (4.10) eine Färbung der Variablen $\{V_1, \ldots, V_s\}$ ab. Wie wir gleich sehen werden garantiert die Eigenschaft (2), daß die grau gefärbten Variablen einen Separator $\mathcal{S}_k$ für $\mathcal{G}_k$ induzieren. Abbildung 4.6 faßt die Schritte unserer Multilevel-Methode zusammen. Im folgenden werden wir jeden Schritt genauer beschreiben.

SEPARATOR(ord_Φ)
01: Construct initial quotient graph $\mathcal{G}_0$ from G.
02: $k := 0$;
03: **while** $\mathcal{G}_k$ not small enough **do**
04: Construct coarser quotient graph $\mathcal{G}_{k+1}$ by eliminating some variables from $\mathcal{G}_k$.
The variables are chosen according to node selection strategy ord_Φ.
05: $k := k + 1$;
06: **end while**
07: Determine a coloring for $\mathcal{G}_k$ that induces a small separator $\mathcal{S}_k$.
08: **while** $k > 0$ **do**
09: Extend the coloring of $\mathcal{G}_k$ to the nodes of $\mathcal{G}_{k-1}$. This induces a separator $\mathcal{S}_{k-1}$ of $\mathcal{G}_{k-1}$.
10: Improve the coloring of $\mathcal{G}_{k-1}$ so that $\mathcal{S}_{k-1}$ is minimized.
11: $k := k - 1$;
12: **end while**
13: Extend the coloring of $\mathcal{G}_0$ to the vertices of G. This induces a separator S of G.
14: Improve S by applying a vertex cover technique.

Abb. 4.6: Funktion SEPARATOR.

Konstruktion des initialen Quotientengraphen Zunächst bestimmen wir in G eine maximale unabhängige Menge U von Knoten. Diese Knoten werden dann aus G eliminiert. Man erhält so $r = \mathrm{card}(U)$ initiale Gebiete. Anschließend werden alle Knoten, die nur auf dem Rand eines Gebietes liegen, mit eben diesem Gebiet verschmolzen. Seien $D_1, \ldots, D_r$ die so erhaltenen Gebiete, und sei $\widehat{V}$ die Menge aller nicht eliminierten Knoten. Nach Ashcraft und Liu generiert die Färbungsregel (4.9) nur dann einen gültigen Knotenseparartor für jede Färbung von $D_1, \ldots, D_r$ wenn gilt

$$\forall\, u, v \in \widehat{V} : (u, v) \in E \Rightarrow \exists\, D \text{ mit } u, v \in \mathrm{adj}_G(D). \tag{4.11}$$

Abbildung 4.7 (a) zeigt, daß die Färbungstechnik versagt, falls (4.11) nicht erfüllt ist. Sei dazu angenommen, daß die drei zu u benachbarten Gebiete schwarz und die zwei zu v benachbarten Gebiete weiß gefärbt sind. Entsprechend der Färbungsregel 4.9 erhält u die Farbe Schwarz und v die Farbe Weiß. Da beide Knoten durch eine Kante verbunden sind, kann die Färbung nicht zu einer Färbung erweitert werden, die einen Knotenseparator in G induziert.

Eine solche Situation tritt nicht auf, falls für alle Knoten $u, v \in \widehat{V}$ mit $(u, v) \in E$ ein Gebiet D existiert mit $u \in \mathrm{adj}_G(D)$ und $v \in \mathrm{adj}_G(D)$. In der Regel erfüllen die Knoten aus $\widehat{V}$ nicht die Bedingung. Sie werden deshalb zu Segmenten $V_1, \ldots, V_s$ zusammengefaßt, so daß gilt

$$\forall\, V_i, V_j \subset \widehat{V} : V_i \cap \mathrm{adj}_G(V_j) \neq \emptyset \Rightarrow \exists\, D \text{ mit } V_i, V_j \cap \mathrm{adj}_G(D) \neq \emptyset. \tag{4.12}$$

Die Segmente $V_1, \ldots, V_s$ sind die kleinsten, die 4.12 erfüllen. In Abbildung 4.7 bilden die Knoten u, v ein Segment. Alle anderen Segmente enthalten nur einen Knoten.

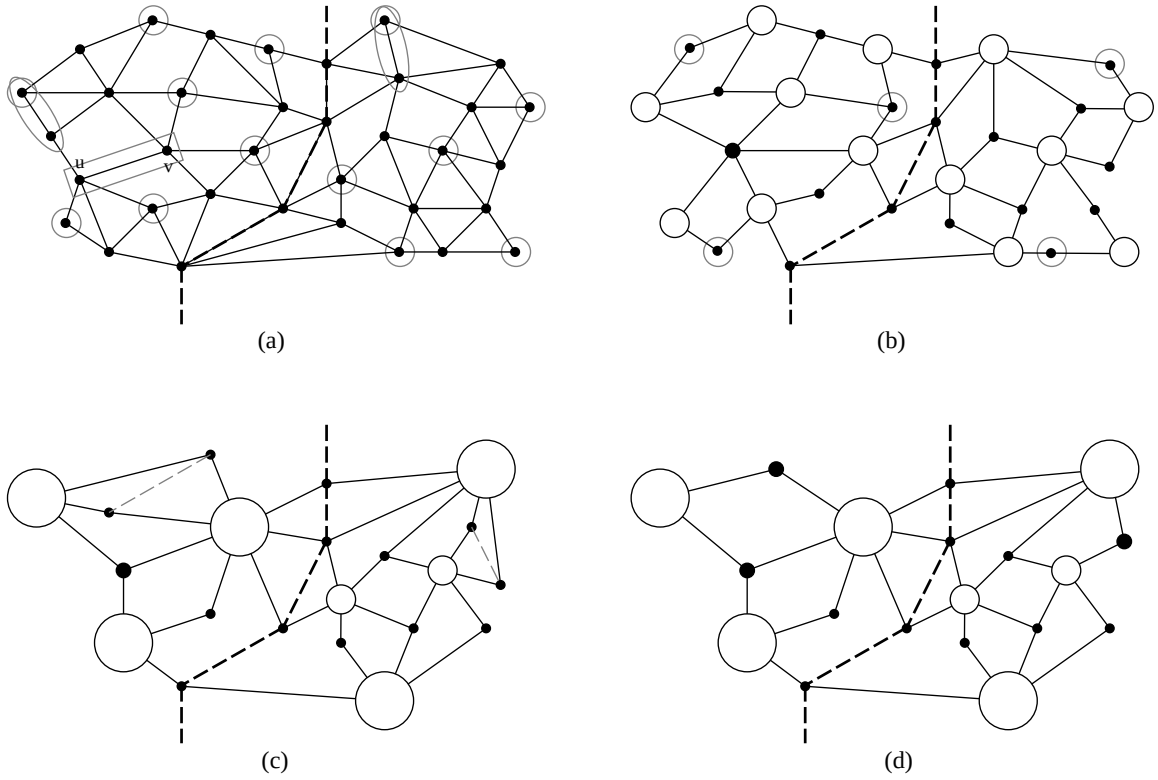


Abb. 4.7: Die Generierung der Quotientengraphen. In (a) ist jeder Knoten des Graphen G, der zur unabhängigen Menge U gehört, durch einen Kreis markiert. Jeder dieser Knoten bildet ein Gebiet. Alle verbleibenden Knoten, die nur zu einem Gebiet benachbart sind, werden mit diesem Gebiet verschmolzen. Dies ist durch ein Oval angedeutet. Die durch Kreise und Ovale umschlossenen Knoten bilden jetzt die initialen Gebiete. Anschließend werden alle durch eine Kante miteinander verbundene Knoten, die zu keinem gemeinsamen Gebiet benachbart sind, zu einem Segment zusammengefaßt. Dies ist durch ein Rechteck angedeutet. Der resultierende initiale Quotientengraph $\mathcal{G}_0$ ist in (b) dargestellt. Dabei entspricht jedem schwarzen Kreis eine Variable und jedem weißen Kreis ein Gebiet. Je mehr Knoten aus G durch ein Element bzw. durch eine Variable repräsentiert werden, desto größer ist der Kreis. (c) zeigt den Quotientengraphen $\mathcal{G}_1$, der aus $\mathcal{G}_0$ durch die Elimination der in (b) markierten Variablen entsteht. Werden in $\mathcal{G}_1$ alle nicht unterscheidbaren Variablen – sie sind in (c) durch eine grau gestrichelte Linie miteinander verbunden – zu einer Supervariablen zusammengefaßt, so erhält man den in (d) dargestellten Quotientengraphen $\mathcal{G}_1'$. Durch die fett gestrichelte Linie wird ein Separator dargestellt.

Wir sind jetzt in der Lage, den initialen Quotientengraphen $\mathcal{G}_0$ zu definieren. Die Elemente von $\mathcal{G}_0$ sind die Gebiete $D_1, \ldots, D_r$ und die Variablen die Segmente $V_1, \ldots, V_s$. Die Kanten von $\mathcal{G}_0$ erhält man durch die Formel (4.2). Offensichtlich besitzt $\mathcal{G}_0$ die Eigenschaft (1). Wegen (4.12) gibt es in $\mathcal{G}_0$ nur Domain-Vertex-Kanten. Daher besitzt $\mathcal{G}_0$ auch die Eigenschaft (2). Die Abbildungen 4.7 (a) und (b) illustrieren die Konstruktion von $\mathcal{G}_0$.

Schrumpfung der Quotientengraphen Sei $\mathcal{G}_k = (\mathcal{D}_k \cup \mathcal{V}_k, \mathcal{E}_k)$ ein Quotientengraph, der die Eigenschaften (1) und (2) besitzt. Zur Konstruktion von $\mathcal{G}_{k+1}$ wird zunächst mit Hilfe der Knotenauswahlstrategie ord_Φ eine unabhängige Menge $\mathcal{U} \subset \mathcal{V}_k$ von Variablen bestimmt. In diesem Zusammenhang heißen zwei Variablen V_i, V_j unabhängig, falls kein Element $D \in \mathcal{D}_k$ existiert mit $(D, V_i) \in \mathcal{E}_k$ und $(D, V_j) \in \mathcal{E}_k$. Anschließend wird jede Variable $V_i \in \mathcal{U}$ mit allen adjazenten Elementen zu einem neuen Element D_{V_i} verschmolzen. Die Verschmelzungsoperationen entsprechen den Eliminationsschritten eines Bottom-up-Algorithmus. Es gibt eine interessante Analogie zu Lius Multiple-Minimum-Degree-Algorithmus. Auch dort wird in jeder Iteration eine unabhängige Menge von Knoten eliminiert. Allerdings ist die Auswahl der Knoten auf solche mit minimalem Grad beschränkt. Um eine schnelle Schrumpfung der Quotientengraphen zu erreichen, ist die Auswahl der Variablen in unserem Vergröberungsprozeß nicht so restriktiv.

Schließlich werden alle Variablen, die nur zu einem Element benachbart sind, mit eben diesem Element verschmolzen. Man erhält so einen Quotientengraphen G_{k+1}, der wieder bipartit ist. Wegen $\mathcal{V}_{k+1} \subset \mathcal{V}_k$ ist jeder Separator in $\mathcal{G}_{k+1}$ auch ein Separator in $\mathcal{G}_k$, und da $\mathcal{G}_k$ die Eigenschaft (1) besitzt, besitzt sie auch $\mathcal{G}_{k+1}$.

Es gibt eine weitere Analogie zu einem Bottom-up-Algorithmus: In $\mathcal{G}_{k+1}$ werden alle Variablen, die zu der gleichen Menge von Elementen benachbart sind, zu einer Supervariablen zusammengefaßt. Hierdurch reduziert sich nochmals die Anzahl der Knoten in $\mathcal{G}_{k+1}$. Die Abbildungen 4.7 (b)-(d) illustrieren die Schrumpfung eines Quotientengraphen.

Färbung der Quotientengraphen Sei wieder $\mathcal{G}_k = (\mathcal{D}_k \cup \mathcal{V}_k, \mathcal{E}_k)$ ein Quotientengraph, der die Eigenschaften (1) und (2) besitzt. Zur Konstruktion eines Separators $\mathcal{S}_k \subset \mathcal{V}_k$ benutzen wir die Färbungstechnik von Ashcraft und Liu [12]. Dabei werden die Variablen nach der folgenden Regel gefärbt:

$$\mathrm{color}(V_i) = \begin{cases} \mathrm{BLACK}, \text{ falls für alle } D \in \mathrm{adj}_{\mathcal{G}_k}(V_i) \text{ gilt } \mathrm{color}(D) = \mathrm{BLACK} \\ \mathrm{WHITE}, \text{ falls für alle } D \in \mathrm{adj}_{\mathcal{G}_k}(V_i) \text{ gilt } \mathrm{color}(D) = \mathrm{WHITE} \\ \mathrm{GRAY}, \text{ sonst.} \end{cases} \tag{4.13}$$

Die Regel (4.13) ist der Regel (4.9) sehr ähnlich. Da es jedoch in $\mathcal{G}_k$ keine Vertex-Vertex Kanten gibt, ist sichergestellt, daß die Menge $\mathcal{S}_k = \{V_i \in \mathcal{V}_k;\ \mathrm{color}(V_i) = \mathrm{GRAY}\}$ für jede Färbung der Elemente in $\mathcal{D}_k$ einen gültigen Knotenseparator darstellt. Eine weitere Segmentbildung bezüglich der Variablen in $\mathcal{V}_k$ ist nicht notwendig.

Zur Bestimmung einer Färbung, die einen möglichst kleinen Separator $\mathcal{S}_k$ induziert, benutzen wir die in Abschnitt 4.2.2 vorgestellte Verbesserungsheuristik. Eingabe der Heuristik ist eine initiale Färbung der Elemente. Im Falle $k = t$ (d. h. $\mathcal{G}_k$ ist der letzte Quotientengraph der Folge) ist die initiale Färbung definiert durch $\mathrm{color}(D) = \mathrm{BLACK}$ für alle $D \in \mathcal{D}_k$. Gilt $k < t$, so erhält man die initiale Färbung durch eine Erweiterung der Färbung von G_{k+1}.

Glättung des Separators Der Separator $\mathcal{S}_0 = \{V_{p_1}, \ldots, V_{p_t}\}$ induziert in G den Separator $S = V_{p_1} \cup \ldots \cup V_{p_t}$. S enthält also nur solche Knoten aus G, die zu einem Segment gehören. In der Regel kann durch Einbeziehung der in den Gebieten von $\mathcal{G}_0$ enthaltenen Knoten das Gewicht des Separators weiter reduziert werden. Diese Glättung des Separators geschieht in einem iterativen Prozeß, der auf der in Abschnitt 4.1.3 beschriebenen Vertex-Cover-Technik beruht. Sei wieder (S, B, W) die Partitionierung von G. Es gelte $|B| \geq |W|$. Mit Hilfe der Dulmage-Mendelsohn-Dekomposition (G ungewichtet) oder der Netzwerk-Fluß-Technik (G gewichtet) wird zunächst in dem durch S und $B' = \mathrm{adj}_G(S) \cap B$ induzierten bipartiten Graphen H ein optimales Vertex-Cover bestimmt. Wie in den Abbildungen 4.2 und 4.3 gezeigt, werden durch beide Verfahren in der Regel zwei Lösungen generiert. In den Lösungen ist das Gewicht des Vertex-Covers – und damit das Gewicht des neuen Separators – gleich, die entsprechenden Partitionierungen können sich jedoch erheblich in ihrer Balance unterscheiden. Da in die Bewertung einer Partitionierung Gewicht und Balance einfließen, wird die besser balancierte Lösung gewählt.

Die Balance ist auch der Grund, warum S zuerst mit der schwereren Menge B gepaart wurde. Nur wenn dies zu keiner Verbesserung führt, wird ein optimales Vertex-Cover in dem durch S und $W' = \mathrm{adj}_G(S) \cap W$ induzierten Graphen berechnet. Schlägt die Optimierung der Partitionierung auch hier fehl, so terminiert der Glättungsprozeß. Anderenfalls wird der gesamte Prozeß mit der verbesserten Partitionierung erneut durchlaufen.

4.2.1.2 Knotenauswahlstrategien zur Schrumpfung eines Graphen

In unserem Multilevel-Verfahren besteht der Separator $\mathcal{S}_k$ eines Quotientengraphen $\mathcal{G}_k = (\mathcal{D}_k \cup \mathcal{V}_k, \mathcal{E}_k)$ aus den in $\mathcal{V}_k$ enthaltenen Variablen. Nach Konstruktion entspricht jeder Variablen aus $\mathcal{V}_k$ ein Segment und jedem Element aus $\mathcal{D}_k$ ein Gebiet in G. Der durch $\mathcal{S}_k$ induzierte Separator S von G setzt sich also aus Randsegmenten der in G gebildeten Gebiete zusammen. Hier zeigt sich ein entscheidender Vorteil unseres Multilevel-Verfahrens. Wird nämlich G mit Hilfe eines Kantenkontraktionsverfahrens zu einem Graphen G' geschrumpft, so induziert jeder Separator S' in G' einen Separator S in G, der aus mehreren nebeneinander liegenden Knotenschichten besteht. Ein großer Teil der in S enthaltenen Knoten ist also redundant. Demgegnüber ist der Anteil redundanter Knoten in einem aus Randsegmenten zusammengesetzten Separator S sehr viel geringer.

Es verbleibt die Frage, nach welchen Kriterien die beim Übergang von $\mathcal{G}_k$ nach $\mathcal{G}_{k+1}$ zu eliminierenden Variablen ausgewählt werden sollen. Unser primäres Ziel ist die Konstruktion

eines leichten Separators S in G. Da sich S aus Randsegmenten zusammensetzt, sollten bei der Bildung von $\mathcal{G}_{k+1}$ die Elemente so verschmolzen werden, daß die Ränder der neuen Elemente aus möglichst wenigen, leichten Segmenten bestehen. Darüber hinaus muß ein unbalanciertes Anwachsen der Elemente vermieden werden. Schwere Elemente entsprechen großen Gebieten in G. Berührt ein Separator ein großes Gebiet so kann der Fall eintreten, daß er eine weite Strecke um dieses Gebiet „herumlaufen“ muß. Der Graph G sollte daher durch ungefähr gleich große Gebiete abgedeckt werden, die einen kleinen Rand besitzen.

Die Generierung von Gebieten mit kleinem Rand entspricht exakt dem Ziel des Minimum-Degree-Algorithmus. Die Knoten auf dem Rand bilden nämlich eine Clique im Eliminationsgraphen. Dies motiviert die folgende Knotenauswahlstrategie: Berechne für jede Variable $V_i \in \mathcal{V}_k$ ihren Knotengrad

$$\deg(V_i) = \sum_{V_j \in \mathcal{M}_{V_i}} |V_j|. \tag{4.14}$$

Dabei enthält $\mathcal{M}_{V_i}$ alle Variablen, die von V_i über ein gemeinsames Element D erreichbar sind, d. h. $V_j \in \mathcal{M}_{V_i} \Leftrightarrow \exists\ D \in \mathcal{D}_k$ mit $V_i \in \mathrm{adj}_{\mathcal{G}_k}(D)$ und $\mathcal{V}_j \in \mathrm{adj}_{\mathcal{G}_k}(D)$. Sortiere anschließend die Variablen aufsteigend nach ihrem Knotengrad und fülle die unabhängige Menge $\mathcal{U}$ beginnend mit der ersten Variablen in der Sortierung auf. Die Knotenauswahlstrategie heißt *Minimum-Degree-In-Quotient-Graph* (QMD).

Unsere zweite Knotenauswahlstrategie basiert auf der *Heavy-Edge-Matching-Heuristik* von Karypis und Kumar [78] Die Idee ist, beim Übergang von $\mathcal{G}_k$ nach $\mathcal{G}_{k+1}$ eine unabhängige Menge möglichst schwerer Variablen zu eliminieren. Diese Auswahlstrategie hat jedoch einen entscheidenden Nachteil: Eine schwere Variable in $\mathcal{G}_k$ entspricht einem großen Randsegment in G. Typischerweise trennt ein solches Randsegment zwei große Gebiete. Durch die bevorzugte Entnahme schwerer Variablen wird so das Wachstum großer Gebiete begünstigt. Um ein solches unbalanciertes Anwachsen der Gebiete zu vermeiden, wird das Gewicht einer Variablen V_i in Relation zu dem Gewicht des neu geformten Elementes D_{V_i} gesetzt. Man erhält dann eine Knotenauswahlstrategie, bei der zur Sortierung der Variablen die Werte

$$\mathrm{score}(V_i) = \frac{1}{|V_i|} \cdot \sum_{D \in \mathrm{adj}_{\mathcal{G}_k}(V_i)} |D| \tag{4.15}$$

benutzt werden. Die Auswahlstrategie heißt *Maximal-Relative-Decrease-Of-Variables-In-Quotient-Graph* (QMRDV). Man beachte, daß durch (4.15) die Generierung von Elementen mit kleinem Aspekt-Ratio begünstigt wird. Dazu betrachten wir zwei Variablen V_i und V_j. Es sei angenommen, daß V_i vier Knoten enthält und zu zwei Elementen adjazent ist, die jeweils ein 4×4-Gitter darstellen. Dann gilt $\mathrm{score}(V_i) = (16+16)/4 = 8$. Durch die Elimination von V_i entsteht ein 4×9-Gitter. Bezüglich V_j sei angenommen, daß die Variable aus zwei Knoten besteht und zu zwei 2×8-Gittern adjazent ist. Für V_j ergibt sich dann ein Score-Wert von $(16+16)/2 = 16$.

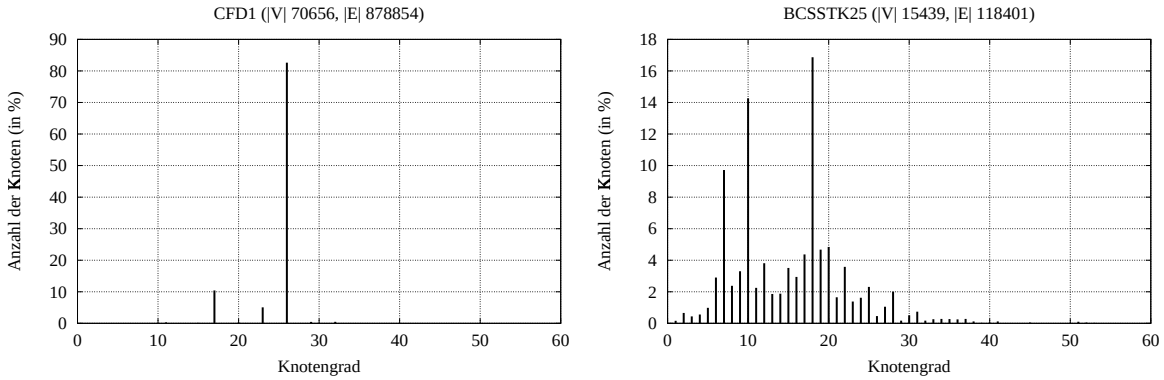


Abb. 4.8: Verteilung der Knotengrade für CFD1 und BCSSTK25.

Wird V_j eliminiert, so entsteht ein 2×17-Gitter. In beiden Fällen entstehen also ungefähr gleich große Gitter. Die Elimination von V_i wird jedoch bevorzugt, so daß ein Gitter mit kleinerem Aspekt-Ratio entsteht.

Wir wollen die Effektivität der Auswahlstrategien exemplarisch an zwei Matrizen darstellen. Die erste Matrix (CFD1) wurde aus einem Programm zur Strömungssimulation extrahiert. Der Graph dieser Matrix ist sehr homogen. Abbildung 4.8 (links) zeigt, daß 83 % der Knoten einen Grad von 26 besitzen. Die zweite Matrix (BCSSTK25) stammt aus der bekannten Harwell-Boeing-Collection [34] und stellt das Modell eines Hochhauses dar. Im Vergleich zu CFD1 ist der Graph von BCSSTK25 sehr viel heterogener. Abbildung 4.8 (rechts) zeigt, daß 14 % der Knoten einen Grad von zehn und 17 % einen Grad von 18 besitzten. Die verbleibenden 69 % der Knoten besitzen einen Grad zwischen eins und 58.

Einfluß der Knotenauswahlstrategie auf die Bildung der Gebiete Wir untersuchen zunächst, welchen Einfluß die Knotenauswahlstrategie auf die Bildung der Gebiete im Vergröberungsprozeß hat. Neben den oben vorgestellten Auswahlstrategien QMD und QMRDV betrachten wir eine dritte Strategie, bei der die zu eliminierenden Variablen nach dem Zufallsprinzip ausgewählt werden. Wir nennen diese Strategie QRAND. Abbildung 4.9 enthält sechs Graphiken, die für jede Matrix und für jede Auswahlstrategie zeigen, wie sich die Größe der im Laufe des Vergröberungsprozesses gebildeten Gebiete gegenüber der Größe ihrer Ränder verhält. Dazu ist jedes Gebiet D durch einen Punkt mit der x-Koordinate $|D|$ und der y-Koordinate $|\operatorname{adj}_G(D)|$ dargestellt. Da wir die „Geometrie“ eines Graphen durch etwa gleich große Gebiete mit einem glatten Rand überdecken möchten, sollten die Punkte möglichst nah am Ursprung des Koordinatensystems liegen.

Die Graphiken auf der linken Seite zeigen die Überlegenheit von QMD und QMRDV im Vergleich zu QRAND. Werden die Quotientengraphen für CFD1 mit Hilfe von QRAND gebildet, so entstehen viele Gebiete, die mehr als 1000 Knoten enthalten. Darüber hinaus besitzten viele

Gebiete D mit $|D| < 1000$ einen größeren Rand als Gebiete, die mit Hilfe der Auswahlstrategien QMD und QMRDV gebildet werden. Man beachte, daß im Falle von QMD und QMRDV kein Gebiet entsteht, das mehr als 1000 Knoten enthält.

Betrachtet man die Graphiken für BCSSTK25, so scheint die Effektivität der Auswahlstrategien QMD und QMRDV nachzulassen. Approximiert man jedoch die Einträge der Graphiken durch eine Kurve, so verläuft diese Kurve sehr viel flacher, wenn die Quotientengraphen mit Hilfe der Auswahlstrategien QMD oder QMRDV gebildet werden. Im Vergleich zu QRAND produzieren also QMD und QMRDV wieder Gebiete mit einem kleineren Rand. Ein Vergleich zwischen den Graphiken für QMD und QMRDV zeigt außerdem, daß durch die Minimum-Degree-Auswahlstrategie ein gleichmäßigeres Anwachsen der Gebiete erreicht wird. Wir haben diese Tendenz für eine Reihe weiterer heterogener Graphen beobachtet.

Zusammengefaßt läßt sich feststellen, daß bei Verwendung der Auswahlstrategien QMD und QMRDV mehr „wohlgeformte“ Gebiete entstehen als bei Verwendung einer auf dem Zufallsprinzip beruhenden Strategie. Im Falle eines heterogenen Graphen, ist die Auswahlstrategie QMD der Auswahlstrategie QMRDV überlegen, da durch sie ein gleichmäßigeres Anwachsen der Gebiete erreicht wird.

Einfluß der Knotenauswahlstrategie auf die Konstruktion der Separatoren Wir widmen uns jetzt der Frage, welchen Einfluß die Knotenauswahlstrategie auf die Konstruktion der Separatoren hat. Dazu haben wir für BCSSTK25 und CFD1 die Quotientengraphen $\mathcal{G}_0, \ldots \mathcal{G}_6$ konstruiert. Die zwei oberen Graphiken in Abbildung 4.10 zeigen die Anzahl der Elemente in $\mathcal{G}_0, \ldots \mathcal{G}_6$ in Abhängigkeit von der verwendeten Knotenauswahlstrategie. Man beachte, daß wir für die y-Achse eine logarithmische Skalierung gewählt haben. Im Falle von CFD1 halbiert sich die Anzahl der Elemente bei jedem Übergang von G_k nach $\mathcal{G}_{k+1}$. Dabei können wir keinen Unterschied zwischen den Auswahlstrategien QMD, QMRDV und QRAND feststellen. Abbildung 4.9 hat jedoch gezeigt, daß die Qualität der Elemente sehr viel besser ist, wenn zur Schrumpfung die Auswahlstrategien QMD oder QMRDV verwendet werden.

Die Graphik für BCSSTK25 illustriert, daß bei Schrumpfung eines heterogenen Graphen die Anzahl der Elemente nicht immer linear abnimmt. Bedingt durch den stark variierenden Knotengrad kommt es schon sehr früh zur Bildung großer Elemente, die zu einer Lähmung des Vergröberungsprozesses führen. Wie in Abbildung 4.9 gesehen, wird ein unbalanciertes Anwachsen der Gebiete am effektivsten von der Auswahlstrategie QMD verhindert. Daher ist es nicht verwunderlich, daß die Anzahl der Elemente in $\mathcal{G}_6$ am geringsten ist, wenn zur Schrumpfung der Quotientengraphen die Auswahlstrategie QMD benutzt wird. Man beachte, daß nur im Falle von QMD das Terminationskriterium (weniger als 200 Elemente im Quotientengraphen) nach sechs Vergröberungsschritten erfüllt ist.

Die beiden Graphiken in der Mitte von Abbildung 4.10 zeigen die Minimierung eines Separators im Laufe des Verfeinerungsprozesses. Dazu haben wir für jeden Quotientengraphen $\mathcal{G}_6$

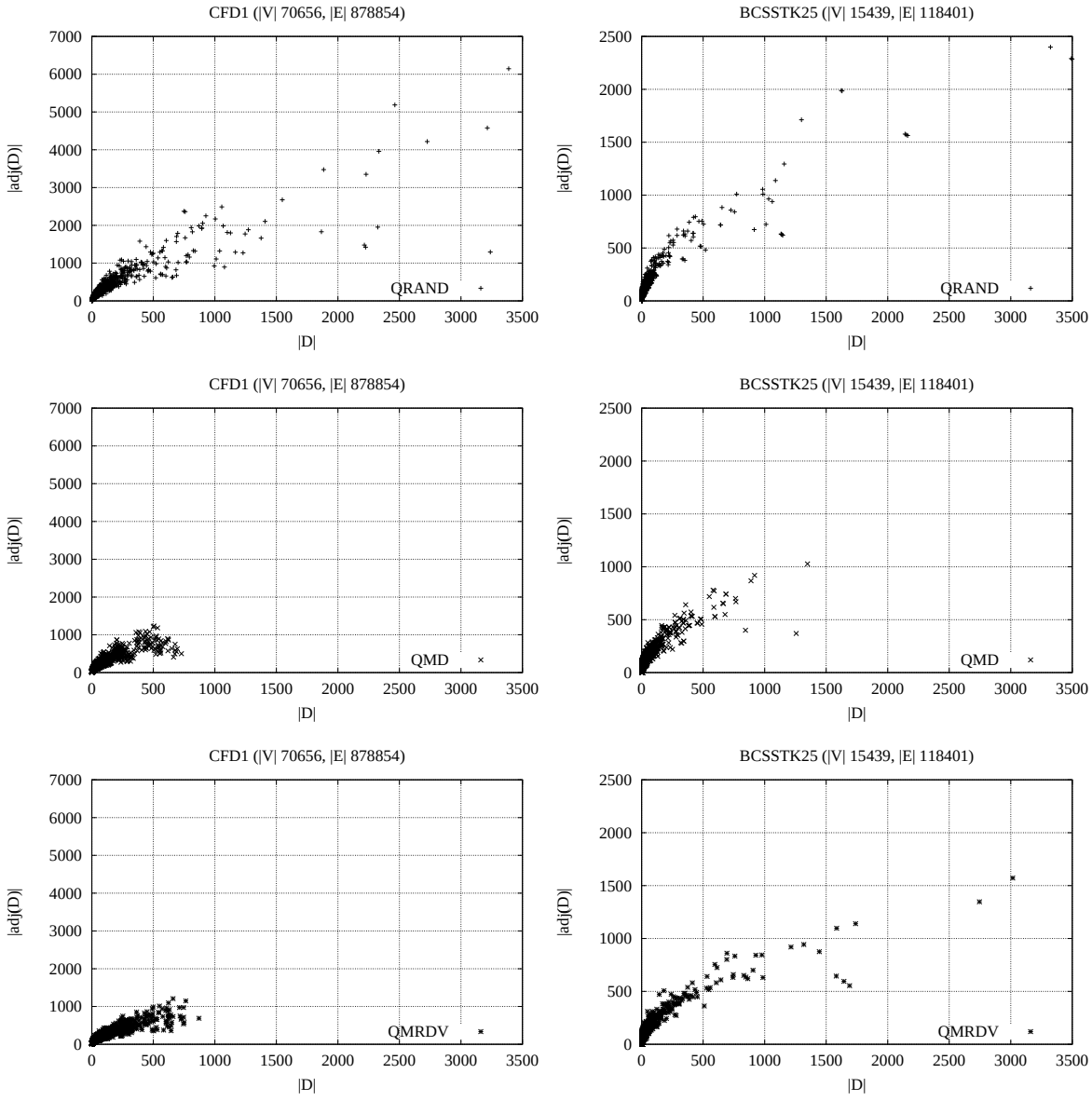


Abb. 4.9: Einfluß der Auswahlstrategien QRAND, QMD und QMRDV auf die Bildung der Gebiete.

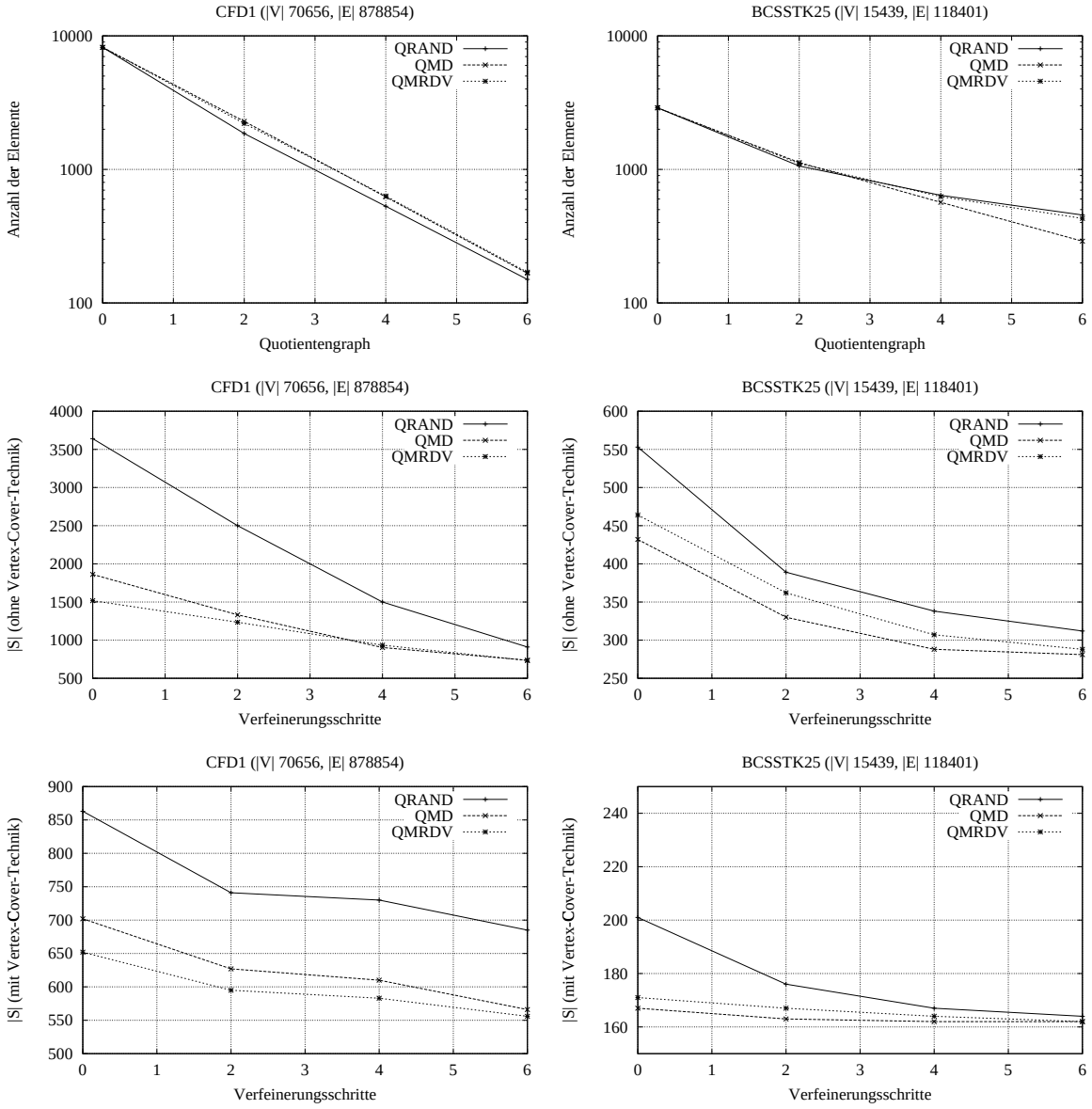


Abb. 4.10: Einfluß der Auswahlstrategien QRAND, QMD und QMRDV auf die Konstruktion der Separatoren.

mit Hilfe der im nächsten Abschnitt vorgestellten Verbesserungsheuristik einen Separator $\mathcal{S}_6$ konstruiert und diesen Separator – wieder mit Hilfe der Verbesserungsheuristik – in i Schritten verfeinert, $i = 0, \ldots, 6$, bis wir einen Separator $\mathcal{S}_{6-i}$ für $\mathcal{G}_{6-i}$ erhalten haben. Die beiden Graphiken zeigen das Gewicht des durch $\mathcal{S}_{6-i}$ induzierten Separators S als eine Funktion von i. Alle Angaben stellen die Durchschnittswerte von elf Ausführungen unseres Multilevel-Verfahrens dar. Vor jeder Ausführung wurden die Adjazenzlisten der Graphen CFD1 und BCSSTK25 zufällig permutiert. Im Falle von QRAND hat der für $\mathcal{G}_6$ konstruierte Knotenseparator ein sehr hohes Gewicht. Dies liegt daran, daß die mit Hilfe von QRAND produzierten Gebietszerlegungen unsere Anforderung „Überdeckung des Graphen G durch ungefähr gleich große Gebiete mit glattem Rand“ am wenigsten erfüllen. Die beiden Graphiken zeigen jedoch auch, daß die negativen Auswirkungen einer schlechten Gebietszerlegung bis zu einem gewissen Grad durch die im Verfeinerungsprozeß verwendete Heuristik kompensiert werden können.

In unserem Experiment haben wir bislang auf den Einsatz der Vertex-Cover-Technik zur Glättung eines Separators verzichtet. Es stellt sich die Frage, wie wichtig der Einsatz dieser Technik zur Konstruktion eines guten Knotenseparators ist. Vielleicht ist diese Technik so mächtig, daß auf den gesamten Verfeinerungsprozeß verzichtet werden kann. Dieser Gedanke ist nicht abwegig, da in dem zweistufigen Ansatz von Ashcraft und Liu [12] die Quotientengraphen $\mathcal{G}_6$ und $\mathcal{G}_{6-i}$ identisch sind. Die letzten beiden Grafiken in Abbildung 4.10 geben eine Antwort auf diese Frage. Die Graphiken zeigen das Gewicht des durch $\mathcal{S}_{6-i}$ induzierten Separators S nach Anwendung der Vertex-Cover-Technik. Die Grafik für CFD1 verdeutlicht, daß trotz der hohen Effektivität der Vertex-Cover-Technik auf den Verfeinerungsprozeß nicht verzichtet werden kann. Insbesondere wenn $\mathcal{G}_6$ eine schlechte Gebietszerlegung darstellt, wie dies bei Verwendung der Auswahlstrategie QRAND der Fall ist, wird der Verfeinerungsprozeß zur Konstruktion eines guten Knotenseparators benötigt. Dies erklärt, warum Ashcraft und Liu einen sehr viel komplexeren Glättungsalgorithmus benutzen.

Die Graphik für BCSSTK25 scheint auf den ersten Blick anzudeuten, daß eine gute Gebietszerlegung bereits ausreicht, um nur mit Hilfe der Vertex-Cover-Technik einen kleinen Knotenseparator bestimmen zu können. In der Tat sind im Falle von QMD und QMRDV die Verfeinerungsschritte vier, fünf und sechs überflüssig. Man muß jedoch bedenken, daß BCSSTK25 das Modell eines Hochhauses darstellt. Daher enthält BCSSTK25 ein weites Spektrum von vielen kleinen Separatoren, die sich lediglich hinsichtlich der Balance ihrer Partitionierungen unterscheiden. Die Effektivität der Vertex-Cover-Technik ist somit auf die spezielle Geometrie von BCSSTK25 zurückzuführen.

4.2.2 Optimierung der Knotenseparatoren

Die Effizienz unserer Multilevel-Methode hängt ganz entscheidend von der Laufzeit des in der Verfeinerungsphase benutzten Optimierungsverfahrens ab. Wie Ashcraft und Liu [12] benutzen

```
IMPROVECOLORING(G = (D ∪ V, E), color)
01: repeat
02:     color* := color;
03:     unmark all D ∈ D;
04:     while there are unmarked elements do
05:         select an unmarked element D;
06:         if color(D) = BLACK then
07:             color(D) := WHITE;
08:             for each variable V_i ∈ adj_G(D) do
09:                 UPDATE_{B→W}(V_i, D);
10:         else
11:             color(D) := BLACK;
12:             for each variable V_i ∈ adj_G(D) do
13:                 UPDATE_{W→B}(V_i, D);
14:         end else
15:         mark D;
16:         let (S*, B*, W*) denote the partition induced by color*;
17:         let (S, B, W) denote the partition induced by color;
18:         if F(S, B, W) < F(S*, B*, W*) then
19:             color* := color;
20:     end while
21:     color := color*;
22: until color has not been improved;
```

Abb. 4.11: Funktion IMPROVECOLORING.

auch wir zur Optimierung einer gegebenen Färbung eine Variante des Fiduccia-Mattheyes-Algorithmus. Im Gegensatz zu ihrem Algorithmus benötigt unsere Variante jedoch nur eine Laufzeit, die asymptotisch der des Vertex-Fiduccia-Mattheyses-Algorithmus entspricht. Im folgenden werden wir zunächst alle Funktionen unseres Optimierungsalgorithmus vorstellen und anschließend seine Laufzeit analysieren.

4.2.2.1 Beschreibung der iterativen Verbesserungsheuristik

Abbildung 4.11 zeigt die Struktur unseres Optimierungsalgorithmus. Eingabeparameter sind ein Quotientengraph $\mathcal{G}$ und eine initiale Färbung color. Der Algorithmus besteht aus zwei ineinander geschachtelten Schleifen. In der inneren while-Schleife werden die Farben ausgewählter Elemente geändert und die neu entstandenen Partitionierungen bewertet. Vor Eintritt der Schleife wird die aktuelle, d. h. die beste bislang gefundene Färbung in color^* gespeichert. Da der Algorithmus die Verschlechterung einer Partitionierung zuläßt, muß sichergestellt werden, daß die

while-Schleife terminiert. Daher darf jedes Element nur einmal seine Farbe wechseln. Nach dem Farbwechsel wird das Element markiert und kann dann nicht mehr in Zeile 05 gewählt werden. Diese Einschränkung motiviert die äußere Schleife, die den Optimierungsprozeß so oft startet bis eine Verbesserung der aktuellen Färbung nicht mehr möglich ist.

Die Auswahl eines Elementes in Zeile 05 geschieht wie folgt: Aus allen unmarkierten schwarzen und aus allen unmarkierten weißen Elementen wird jeweils dasjenige Element ausgewählt, durch dessen Farbwechsel das Gewicht des Separators am meisten reduziert bzw. am geringsten erhöht wird. Seien D_b und D_w die entsprechenden Elemente. In Abhänigigkeit von der Bewertungsfunktion F setzt man dann $D := D_b$ oder $D := D_w$. Gibt es von einer Farbe keine unmarkierten Elemente mehr, so erübrigt sich die Benutzung von F.

Nachdem ein Element D bestimmt und dessen Farbe geändert wurde, wird die Färbungsregel 4.13 auf jede Variable $V_i \in \mathrm{adj}_{\mathcal{G}}(D)$ angewandt. Dies geschieht durch Aufruf der Funktionen UPDATE$_{B\to W}$ und UPDATE$_{W\to B}$. Falls die neue Färbung eine Partitionierung (S, B, W) induziert, die besser ist als die durch color^* induzierte Partitionierung (S^*, B^*, W^*), so wird color nach color^* kopiert. Zur Bewertung einer Partitionierung benuzten wir die Funktion

$$\begin{aligned} F(S,B,W) = |S| + \rho \cdot \max(0, \tau \cdot \max(|B|,|W|) - \min(|B|,|W|)) \\ + \frac{\max(|B|,|W|) - \min(|B|,|W|)}{\max(|B|,|W|)}. \end{aligned} \tag{4.16}$$

In F ist das Gewicht des Separators S die bestimmende Größe. Erst wenn die Differenz zwischen dem Gewicht von B und dem Gewicht von W einen gewissen Toleranzwert übersteigt, wird ein zusätzlicher Strafterm aufaddiert. Die Toleranz wird mit Hilfe des Parameters τ, $0 \leq \tau \leq 1$, eingestellt und die Höhe des Strafterms mit Hilfe des Parameters $\rho > 0$. Der dritte Term fungiert als Tie-Breaker bei der Wahl zwischen mehreren gleich schweren Separatoren, wenn die Balance innerhalb des tolerierten Bereichs liegt.

Ein sehr wichtiges Detail bei der Implementierung der Funktion IMPROVECOLORING ist die effiziente Berechnung der Gewichte $|S|$, $|B|$ und $|W|$ nach einem Farbwechsel. Die Gewichte werden zur Bewertung der neuen Partitionierung benötigt. Um eine effiziente Berechnung zu ermöglichen, speichern wir in $\Delta_S(D)$, $\Delta_B(D)$ und $\Delta_W(D)$, um welchen Wert sich die Gewichte der Mengen ändern, wenn D seine Farbe wechselt. Die Werte $\Delta_S(D)$, $\Delta_B(D)$ und $\Delta_W(D)$ können positiv oder negativ sein. Sind sie bekannt, so erhält man die Bewertung der neuen Partitionierung durch Einsetzen von $|S| + \Delta_S(D)$, $|B| + \Delta_B(D)$ und $|W| + \Delta_W(D)$ in F.

Nachdem die Farbe eines Elementes D geändert wurde, müssen die Δ-Werte aller Elemente D', die mit D über eine Variable verbunden sind, überprüft und gegebenenfalls neu berechnet werden. Auch dies geschieht innerhalb der Funktionen UPDATE$_{B\to W}$ und UPDATE$_{W\to B}$. In beiden Funktionen werden vier Fälle unterschieden. Die ersten beiden Fälle beziehen sich auf die Situation vor, die letzten beiden auf die Situation nach dem Farbwechsel. Im folgenden betrachten wir nur die in Abbildung 4.12 dargestellte Funktion UPDATE$_{B\to W}$. Die Funktion UPDATE$_{W\to B}$ kann analog formuliert werden.

```
UPDATE_{B→W}(V_i, D)
01: /* Case 1: Before flipping D to WHITE there was only one other WHITE element.
02:     Search it and update its Δ_B and Δ_S values. */
03: if there is only one D' ∈ adj_G(V_i), D' ≠ D, with color(D') = WHITE then
04:     Δ_S(D') := Δ_S(D') + |V_i|;
05:     Δ_B(D') := Δ_B(D') − |V_i|;
06: end if
07: /* Case 2: Before flipping D to WHITE all elements were colored BLACK.
08:     Move V_i into the separator and update Δ_B and Δ_S of all BLACK elements. */
09: if color(D') = BLACK for all D' ∈ adj_G(V_i), D' ≠ D then
10:     color(V_i) := GRAY;
11:     for each D' ∈ adj_G(V_i), D' ≠ D do
12:         Δ_S(D') := Δ_S(D') − |V_i|;
13:         Δ_B(D') := Δ_B(D') + |V_i|;
14:     end for
15: end if
16: /* Case 3: After flipping D to WHITE there is only one remaining BLACK element.
17:     Search it and update its Δ_W and Δ_S values. */
18: if there is only one D' ∈ adj_G(V_i), D' ≠ D, with color(D') = BLACK then
19:     Δ_S(D') := Δ_S(D') − |V_i|;
20:     Δ_W(D') := Δ_W(D') + |V_i|;
21: end if
22: /* Case 4: After flipping D to WHITE all elements are colored WHITE.
23:     Remove V_i from the separator and update Δ_W and Δ_S of all WHITE elements. */
24: if color(D') = WHITE for all D' ∈ adj_G(V_i), D' ≠ D then
25:     color(V_i) := WHITE;
26:     for each D' ∈ adj_G(V_i), D' ≠ D do
27:         Δ_S(D') := Δ_S(D') + |V_i|;
28:         Δ_W(D') := Δ_W(D') − |V_i|;
29:     end for
30: end if
```

Abb. 4.12: Funktion UPDATE$_{B\to W}$.

Sei V_i die Variable, für die die Funktion UPDATE$_{B\to W}$ aufgerufen wird. Wir nehmen zunächst an, daß D noch immer schwarz gefärbt ist. In den Zeilen 01–06 betrachten wir den Fall, daß es in der Nachbarschaft von V_i nur ein Element D' gibt mit $\mathrm{color}(D') = \mathrm{WHITE}$. Wäre D' schwarz gefärbt worden, so wäre V_i nicht länger Teil des Separators. Dadurch hätte sich das Gewicht des Separators um den Wert $|V_i|$ reduziert. In $\Delta_S(D')$ ist deshalb der Wert $-|V_i|$ und in $\Delta_B(D')$ der Wert $+|V_i|$ enthalten. Nach dem Farbwechsel von D gibt es nun aber zwei weiße Elemente in der Nachbarschaft von V_i. Erhält jetzt D' die Farbe schwarz, so verbleibt V_i im Separator. Daher müssen $\Delta_B(D')$ und $\Delta_S(D')$ korrigiert werden.

Betrachten wir nun den Fall, daß es kein weiß gefärbtes Element D' in der Nachbarschaft von V_i gibt (Zeilen 07–15). Dann ist D das erste Element in $\mathrm{adj}_{\mathcal{G}}(V_i)$ mit $\mathrm{color}(D) = \mathrm{WHITE}$. Dadurch wird V_i zu einem Teil des Separators und ein Farbwechsel der anderen schwarzen Elemente in $\mathrm{adj}_{\mathcal{G}}(V_i)$ verbilligt sich um den Wert $|V_i|$.

Sei nun angenommen daß D seine Farbe geändert hat und weiß gefärbt ist. In den Zeilen 16–21 betrachten wir den Fall, daß es nur noch ein schwarzes Element D' in der Nachbarschaft von V_i gibt. Nach einem Farbwechsel von D' würde auch V_i weiß gefärbt. Damit wäre V_i nicht länger Teil des Separators. Deshalb wird $\Delta_S(D')$ um den Wert $|V_i|$ reduziert und $\Delta_W(D')$ um den gleichen Wert erhöht.

Betrachten wir abschließend den Fall, daß alle Elemente in der Nachbarschaft von V_i weiß gefärbt sind (Zeilen 22–30). Dann ist V_i nicht länger Teil des Separators. V_i wird jedoch wieder Bestandteil des Separators, wenn ein Element $D' \in \mathrm{adj}_{\mathcal{G}}(V_i)$ schwarz gefärbt wird. Der Farbwechsel eines solchen Elements verteuert sich daher um den Wert $|V_i|$.

Man überlegt sich leicht, daß weitere Fallunterscheidungen nicht notwendig sind. Wie bereits oben erwähnt kann die Funktion UPDATE$_{W\to B}$ analog formuliert werden. Wir sind jetzt in der Lage, die Laufzeit unserer Verbesserungsheuristik zu analysieren.

4.2.2.2 Aufwandsanalyse

Die Effizienz unseres Optimierungsalgorithmus wird ganz entscheidend von der Ausführungszeit der Funktion IMPROVECOLORING bestimmt. Natürlich können wir nicht a-priori berechnen wie oft die äußere Schleife durchlaufen wird. Wir können jedoch analysieren wie teuer eine Ausführung der inneren while-Schleife ist.

Kritisch in IMPROVECOLORING sind die Bestimmung von D (Zeile 05) und die for-Schleifen in den Zeilen 08–09 sowie 12–13. In den Schleifen werden die Funktionen UPDATE$_{B\to W}(V_i, D)$ und UPDATE$_{W\to B}(V_i, D)$ aufgerufen. Trifft mindestens einer der vier Fälle zu, so kostet ein Aufruf $O(\deg_{\mathcal{G}}(V_i))$ Zeiteinheiten. Wir sprechen dann von einem *aktiven* Aufruf. Man beachte, daß alle vier if-Abfragen in konstanter Zeit ausgewertet werden können. Dies geschieht mit Hilfe zweier Zähler $\#_B(V_i)$ und $\#_W(V_i)$, die für eine Variable V_i angeben, wieviele schwarz und wieviele weiß gefärbte Elemente in der Nachbarschaft von V_i vorhanden sind. Jeder nicht aktive

Aufruf kann also in konstanter Zeit abgearbeitet werden. Das folgende Lemma zeigt, daß von den insgesamt $\deg_{\mathcal{G}}(V_i)$ Aufrufen (jedes benachbarte Element wechselt genau einmal seine Farbe) nur maximal vier aktiv sein können. Daher verursachen alle Aufrufe von UPDATE$_{B\to W}$ und UPDATE$_{W\to B}$ zusammen einen Aufwand von $O(|\mathcal{E}|)$.

Lemma 4.1 *Sei V_i eine Variable des Quotientengraphen $\mathcal{G}$. Dann gibt es in* IMPROVECOLORING *maximal vier aktive Aufrufe von* UPDATE$_{B\to W}(V_i, D)$ *und* UPDATE$_{W\to B}(V_i, D)$.

Beweis: Die Aussage ist klar für alle Variablen vom Grad ≤ 4. Sei also $d := \deg_{\mathcal{G}}(V_i)$ mit $d > 4$. Weiter sei a die initiale Anzahl von weiß gefärbten Elementen in $\mathrm{adj}_{\mathcal{G}}(V_i)$. Es gilt $0 \leq a \leq d$. Durch einen Aufruf von UPDATE$_{B\to W}(V_i, D)$ wird $\#_W(V_i)$ um eins erhöht. Dieser Aufruf ist aktiv, falls vor dem Aufruf $\#_W(V_i) \in \{0, 1\}$ gilt, oder falls nach dem Aufruf $\#_W(V_i) \in \{n_v - 1, n_v\}$ gilt. Umgekehrt wird durch einen Aufruf UPDATE$_{W\to B}(V_i, D)$ der Wert $\#_W(V_i)$ um eins reduziert. Dieser Aufruf ist aktiv, falls vor dem Aufruf $\#_W(V_i) \in \{d-1, d\}$ gilt (Fälle 1 und 2 in Abbildung 4.12), oder falls nach dem Aufruf $\#_W(V_i) \in \{0, 1\}$ gilt (Fälle 3 und 4 in Abbildung 4.12). Im folgenden nennen wir einen Aufruf UPDATE$_{B\to W}(V_i, D)$ *Aufwärts-* und einen Aufruf UPDATE$_{W\to B}(V_i, D)$ *Abwärtsschritt*. Da jedes Element in $\mathrm{adj}_{\mathcal{G}}(V_i)$ genau einmal seine Farbe wechselt, gibt es $d - a$ Aufwärts- und a Abwärtsschritte. In Abhängigkeit von a können fünf Fälle unterschieden werden:

Fall 1: $a = 0$

Dann gibt es d Aufwärtsschritte, von denen genau vier aktiv sind.

Fall 2: $a = 1$

Dann gibt es $d - 1$ Aufwärtsschritte, von denen maximal drei aktiv sind. Zusätzlich gibt es einen Abwärtsschritt. Auch dieser kann aktiv sein.

Fall 3: $a = d$ (analog zu Fall 1)

Fall 4: $a = d - 1$ (analog zu Fall 2)

Fall 5: $2 \leq a \leq d - 2$

Wir zeigen im folgenden, daß es maximal zwei aktive Aufwärtsschritte geben kann. Dabei bezeichnen wir einen Aufwärtsschritt durch ein Tupel $(i, i+1)$, wobei i dem Wert $\#_W(V_i)$ vor dem Aufwärtsschritt entspricht. Wir unterscheiden drei Fälle:

Fall 5.1: Der erste aktive Aufwärtsschritt ist $(0, 1)$.

Dann sind alle a Abwärtsschritte bereits ausgeführt worden und von den verbleibenden Aufwärtsschritten (falls vorhanden) kann nur noch $(1, 2)$ aktiv sein.

Fall 5.2: Der erste aktive Aufwärtsschritt ist $(1, 2)$.

Dann ist von den initial a Abwärtsschritten noch einer übrig. Darüber hinaus gibt es noch $\leq d - 3$ weitere Aufwärtsschritte. Daher kann es noch einmal einen aktiven Aufwärtsschritt $(1, 2)$, oder einen aktiven Aufwärtsschritt $(d - 2, d - 1)$ geben.

Beides zusammen ist jedoch nicht möglich, da hierfür die Anzahl der zur Verfügung stehenden Aufwärtsschritte nicht ausreicht.

Fall 5.3: Der erste aktive Aufwärtsschritt ist $(d-2, d-1)$.
Dann gibt es nur noch einen Aufwärtsschritt.

Analog zeigt man, daß es für $2 \leq a \leq d-2$ maximal zwei aktive Abwärtsschritte geben kann. Insgesamt folgt damit die Behauptung. ■

Es verbleibt zu untersuchen, welcher zusätzliche Zeitaufwand durch die Auswahl der Elemente in Zeile 05 verursacht wird. Alle unmarkierten schwarzen Elemente sind sortiert nach ihren Δ_S-Werten in einem Heap abgespeichert. Ein zweiter Heap enthält die unmarkierten weißen Elemente. Zur Auswahl eines Elementes werden daher $O(\log|\mathcal{D}|)$ Zeiteinheiten benötigt. Das Heap-Management verursacht einen zusätzlichen Aufwand bei jedem aktiven Aufruf von UPDATE$_{B\to W}(V_i, D)$ und UPDATE$_{W\to B}(V_i, D)$. Ändert sich nämlich für ein zu V_i benachbartes Element D' der Wert $\Delta_S(D')$, so muß D' neu in den entsprechenden Heap eingeordnet werden. Jeder aktive Aufruf kann daher $O(\deg_{\mathcal{G}}(V_i)\log|\mathcal{D}|)$ Zeiteinheiten kosten. Da es jedoch nur vier aktive Aufrufe für V_i gibt, erhält man:

Satz 4.1 *Sei $\mathcal{G} = (\mathcal{X}, \mathcal{E})$ ein Quotientengraph. Ein Durchlauf der inneren while-Schleife in* IMPROVECOLORING *kostet $O(|\mathcal{E}|\log|\mathcal{X}|)$ Zeiteinheiten.*

Damit besitzt unser Optimierungsalgorithmus die gleiche Effizienz wie der von Ashcraft und Liu [10] vorgeschlagene Vertex-Fiduccia-Mattheyses-Algorithmus.

4.2.3 Dreistufiges Multisection

Das dreistufige Multisection-Verfahren basiert auf einer Verallgemeinerung des in Abbildung 4.5 präsentierten Algorithmus. Die Idee ist, einige Separatoren aus Φ entsprechend ihrer Rekursionstiefe und einige mit Hilfe eines Bottom-up-Verfahrens wie z. B. Minimum-Degree zu numerieren. Das dreistufige Multisection-Verfahren stellt eine Kombination aus unvollständigem Nested-Dissection $\mathrm{MS}(\mathrm{MD}, \mathrm{ND})$, und Local-Nested-Dissection $\mathrm{MS}(\mathrm{ND}, \mathrm{Profil})$ dar. Ersetzt man in dem Local-Nested-Dissection-Algorithmus das Profil- durch ein Minimum-Degree-Ordering, so erhält man durch die Kombination beider Algorithmen ein dreistufige Multisection-Verfahren vom Typ $\mathrm{MS}(\mathrm{MD}, \mathrm{ND}, \mathrm{MD})$. Es verbleibt die Frage, welche Separatoren aus Φ mit Hilfe von Nested-Dissection und welche mit Hilfe von Minimum-Degree numeriert werden sollen.

4.2.3.1 Das generische dreistufige Multisection-Verfahren

Ein Bottom-up-Algorithmus kann aufgrund seiner lokalen Struktur nicht voraussehen, wie die Elimination eines Knotens den weiteren Eliminationsprozeß – und damit die Bildung zukünftiger Gebiete – beeinflußt. Um diese Schwäche zu überwinden, zerteilt man den Graphen mit

Hilfe von Knotenseparatoren in mehrere zusammenhängende Teilgraphen. Die Teilgraphen bilden dann die Gebiete eines unvollständigen Bottom-up-Orderings. Normalerweise werden die Separatoren in einem rekursiven Prozesses konstruiert. Ziel ist dabei die Konstruktion möglichst kleiner Separatoren. Man hofft, daß der rekursive Prozeß die Separatoren so anordnet, daß Gebiete mit einem kleinen Rand entstehen. Genau hier zeigt sich die Schwäche eines Top-down-Verfahrens wie Nested-Dissection: Bei der Bestimmung eines Knotenseparators für einen Graphen G' bleibt der aus bereits konstruierten Separatoren bestehende Rand von G' unberücksichtigt. Folglich wird der Rand eines Gebietes auch bei der Numerierung der Knoten außer Acht gelassen. Im Gegensatz dazu, fließt der Rand eines Gebietes auf natürliche Art und Weise in die Berechnung eines Bottom-up-Orderings ein.

Es gibt eine wichtige Klasse von Graphen, für die das Nested-Dissection-Verfahren eine asymptotisch optimale Eliminationssequenz erzeugt, nämlich die Klasse der $n \times n$-Gitter. Hier werden die Knotenseparatoren so angeordnet, daß immer wieder quadratische Gebiete entstehen. Dies motiviert folgende Strategie zur Verbesserung eines Multisection-Orderings: Falls die Knotenseparatoren des Multisektors Φ derart angeordnet sind, daß durch eine Elimination entsprechend der Rekursionstiefe viele quadratische Gebiete entstehen, so behalte die Nested-Dissection-Numerierung bei. Werden jedoch viele Gebiete mit einem großen Aspekt-Ratio erzeugt, so verwerfe die Nested-Dissection-Numerierung und eliminiere die Separatoren mit Hilfe eines Bottom-up-Verfahrens wie z. B. Minimum-Degree.

Theoretisch wird diese Vorgehensweise unterstützt durch die Arbeiten von Bhat et al. [19]. In ihrem Local-Nested-Dissection-Algorithmus, dem erfolgreichsten Ordering-Verfahren für $h \times n$-Gitter, werden die quadratischen Gebiete mit Hilfe eines Nested-Dissection- und die Knotenseparatoren mit Hilfe eines Profil-Verfahrens numeriert. Benutzt man in dem Local-Nested-Dissection-Algorithmus anstelle eines Profil- ein Minimum-Degree-Verfahren, so ändert sich der Grad der Auffüllung kaum.

Um die Vorgehensweise realisieren zu können, muß der Aspekt-Ratio der entstehenden Gebiete bekannt sein. Aspekt-Ratio ist jedoch nur dann wohldefiniert, wenn für die Knoten des Graphen eine Einbettung in die Ebene existiert. Eine einfache Lösung dieses Problems sieht wie folgt aus: Da die Knotenseparatoren des Multisektors Φ in einem rekursiven Bisektionsprozeß berechnet werden, kann man sie als Binärbaum darstellen. Sei T dieser Binärbaum. Berechne nun für jeden Teilbaum T_S von T ein Minimum-Degree-Ordering, und vergleiche es mit dem gegebenen Nested-Dissection-Ordering. Bestimme anschließend die maximalen Teilbäume, für die Nested-Dissection besser ist als Minimum-Degree, und eliminiere die Separatoren in diesen Teilbäumen entsprechend ihrer Rekursionstiefe. Bezeichne $\Phi' \subset \Phi$ den verbleibenden Multisektor. Bilde den Schur-Komplement Graphen $G_{\Phi'}$, und eliminiere die Knoten in $G_{\Phi'}$ mit Hilfe eines Minimum-Degree-Verfahrens.

Besitzt der Separatorbaum $k = O(\log n)$ Ebenen, so benötigt man für die Berechnung des dreistufigen Multisection-Orderings einen Aufwand, der nach oben begrenzt ist durch den $k+1$-

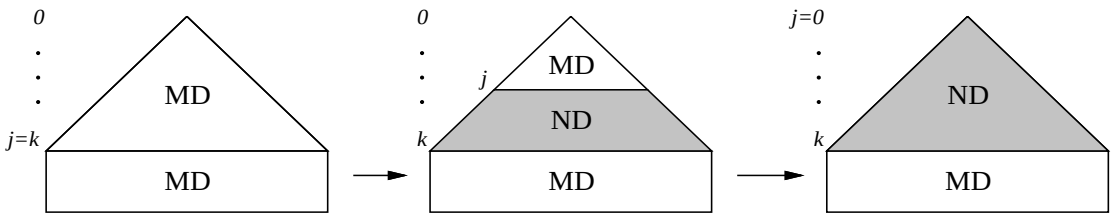


Abb. 4.13: Spektrum von Multisection-Orderings.

fachen Aufwand zur Berechnung eines Minimum-Degree-Orderings. Dies sieht man wie folgt: Für jeden Teilbaum T_S von T muß ein Minimum-Degree-Ordering berechnet werden. Geht man in T eine Ebene tiefer, so verdoppelt sich zwar die Anzahl der Teilbäume, ihre Größe halbiert sich jedoch auch. Für die Berechnung aller Minimum-Degree-Orderings benötigt man deshalb einen Aufwand, der nach oben begrenzt ist durch den Aufwand zur Berechnung von k Minimum-Degree-Orderings. Berücksichtigt man jetzt noch den Aufwand zur Berechnung eines Minimum-Degree-Orderings für die Knoten in $\Omega_1, \ldots, \Omega_q$ und Φ', so erhält man das gewünschte Resultat.

4.2.3.2 Das dreistufige Multisection-Verfahren von Ashcraft, Liu und Eisenstat

Eine effizientere Methode zur Berechnung eines dreistufigen Multisection-Verfahrens wurde von Ashcraft, Liu und Eisenstat vorgestellt [4, 15]. Sei k die tiefste Ebene des Separatorbaumes und $j \in \{0, \ldots, k\}$. Ihr dreistufiges Multisection-Verfahren evaluiert alle Orderings, die nach der folgenden Vorschrift generiert werden können:

(1) Eliminiere die Knotenseparatoren in den unteren Ebenen $k, \ldots, j-1$ entsprechend ihrer Rekursionstiefe.

(2) Eliminiere die Knotenseparatoren in den oberen Ebenen $j, \ldots, 0$ mit Hilfe eines Minimum-Degree-Algorithmus.

Von diesen $k+1$ Orderings wird anschließend das Beste ausgewählt. Abbildung 4.13 zeigt, daß durch die Vorschrift ein weites Spektrum von Orderings generiert wird. Jedes Ordering wird durch ein Dreieck und ein darunter liegendes Rechteck symbolisiert. Das Dreieck repräsentiert den Separatorbaum und enthält alle Knoten aus Φ. Das Rechteck steht für die Knoten aus $\Omega_1, \ldots, \Omega_q$. Auf der linken Seite des Spektrums werden sowohl die Knoten aus $\Omega_1, \ldots, \Omega_q$ als auch die Separatoren aus Φ mit Hilfe eines Minimum-Degree-Verfahrens numeriert. Man erhält so ein Multisection-Ordering vom Typ $\mathrm{MS}(\mathrm{MD}, \mathrm{MD})$. Auf der gegenüberliegenden Seite des Spektrums werden die Separatoren aus Φ entsprechend ihrer Rekursionstiefe eliminiert, und man erhält ein unvollständiges Nested-Dissection-Ordering, d. h. ein Multisection-Ordering vom Typ $\mathrm{MS}(\mathrm{MD}, \mathrm{ND})$.

Die nach der Vorschrift von Ashcraft et al. generierten Orderings überlappen sich stark. Für $j \in \{0, \ldots, k\}$ bezeichne $\Phi_j \subset \Phi$ den Multisektor, der die Separatoren der Ebenen $0, \ldots, j$

TRISTAGEMULTISECTION$(\mathrm{ord}_\Phi, \mathrm{ord}_1, \mathrm{ord}_2)$
01: Determine a domain decomposition $(\Phi_k, \Omega_1, \ldots, \Omega_q)$ of G by a recursive bisection process. Use node selection strategy ord_Φ to construct the vertex separators.
02: **for** each set Ω_i **do**
03: Eliminate all vertices in Ω_i using node selection strategy ord_1.
04: Store operation count in ops_0 and construct elimination graph G_{Φ_k}.
03: $\mathrm{ops}_1 := 0$; $\mathrm{ops}^* := \infty$;
04: **for** $j := k$ **downto** 0 **do**
05: **if** $j < k$ **then**
06: Eliminate from $G_{\Phi_{j+1}}$ all separators in level $j+1$ to obtain the actual elimination graph G_{Φ_j}. Add operation count to ops_1.
07: **end if**
08: Order the vertices in G_{Φ_j} using node selection strategy ord_2.
Store operation count in ops_2.
09: **if** $\mathrm{ops}_0 + \mathrm{ops}_1 + \mathrm{ops}_2 < \mathrm{ops}^*$ **then**
10: $j^* := j$;
11: $\mathrm{ops}^* := \mathrm{ops}_0 + \mathrm{ops}_1 + \mathrm{ops}_2$;
12: **end if**
13: **end for**
14: Splice together the bottom-up orderings on $\Omega_1, \ldots, \Omega_q$, the nested dissection ordering on $\Phi_k - \Phi_{j^*}$, and the bottom-up ordering on Φ_{j^*}.

Abb. 4.14: Funktion TRISTAGEMULTISECTION.

enthält. (d. h. $\Phi_k = \Phi$). Berechnet man die Orderings wie in Abbildung 4.13 angedeutet von $j = k$ bis $j = 0$, so kann beim Übergang von j nach $j-1$ die Nested-Dissection-Numerierung der Separatoren $\Phi - \Phi_j$ zu einer Nested-Dissection-Numerierung der Separatoren $\Phi - \Phi_{j-1}$ erweitert werden. Man muß dann nur noch ein Minimum-Degree-Ordering für die Knoten aus Φ_j berechnen.

Abbildung 4.14 zeigt die Berechnung und Evaluierung der $k+1$ Orderings in dem verallgemeinerten Multisection-Verfahren. Nach Konstruktion des Multisektors Φ_k, werden zunächst die Knoten in den Mengen $\Omega_1, \ldots, \Omega_q$ eliminiert. Die Variable ops_0 speichert die Anzahl der zur Faktorisierung der entsprechenden Spalten benötigten Multiplikations- und Additionsoperationen. Diese läßt sich sehr einfach bestimmen. Wird nämlich ein Knoten v im Schritt k eliminiert (also $\pi(v) = k$), so gilt $\eta(L_{*,k}) = \deg_{G_{k-1}}(v)$. Mit $\eta(L_{*,k})$ ist nach (2.6) und (2.7) auch die Anzahl der zur Faktorisierung von k benötigten Operationen bekannt. Als Ergebnis des Eliminationsprozesses erhält man den Schur-Komplement Graphen G_{Φ_k}. Die Berechnung der verschiedenen Orderings für G_{Φ_k} geschieht in der nun folgenden for-Schleife (Zeilen 04–13).

In jeder Iteration $j < k$ der for-Schleife wird als erstes der aktuelle Eliminationsgraph G_{Φ_j} konstruiert (im Falle $j = k$ entspricht G_{Φ_j} dem Schur-Komplement Graphen G_{Φ_k}). Dazu werden aus dem Eliminationsgraphen $G_{\Phi_{j+1}}$ der Iteration $j+1$ die Knotenseparatoren der Ebene $j+1$ eliminiert. Die Variable ops_1 speichert die Summe der Multiplikations- und Additionsoperationen, die durch die Faktorisierung der in diesen Nested-Dissection-Schritten eliminierten Knoten entstehen. Anschließend wird für die Knoten des Graphen G_{Φ_j} ein Bottom-up-Ordering berechnet. Da G_{Φ_j} in der nächsten Iteration zur Konstruktion von $G_{\Phi_{j-1}}$ benötigt wird, darf der Graph bei der Berechnung des Bottom-up-Orderings nicht zerstört werden. Die Numerierung der Knoten in G_{Φ_j} komplettiert ein neues dreistufiges Multisection-Ordering. Falls es das Beste bisher gefundene Ordering ist, wird j in j^* gespeichert.

Nach Durchlauf der for-Schleife steht in j^* die Ebene, ab der die Separatoren nicht mehr entsprechend ihrer Rekursionstiefe, sondern mit Hilfe eines Bottom-up-Verfahrens eliminiert werden sollten. Das beste dreistufige Multisection-Ordering erhält man daher wie in Zeile 14 beschrieben.

Der Aufwand zur Bestimmung des besten dreistufigen Multisection-Orderings ist nach oben begrenzt durch den Aufwand zur Berechnung von drei Bottom-up-Orderings. Dies sieht man wie folgt: In jeder Iteration halbiert sich die Anzahl der Knoten im aktuellen Eliminationsgraphen. Zeile 08 verursacht daher einen Berechnungsaufwand von höchstens zwei Bottom-up-Orderings. Zusammen mit dem Aufwand zur Berechnung der Bottom-up-Orderings für die Knoten in $\Omega_1, \ldots, \Omega_q$ (Zeilen 02–03) und dem Aufwand zur Konstruktion der aktuellen Eliminationsgraphen (Zeile 06) erhält man das gewünschte Resultat.

4.3 Die Ordering-Bibliothek PORD

Dieser Abschnitt ist in zwei Teilabschnitte gegliedert. Zunächst stellen wir in 4.3.1 die in der Ordering-Bibliothek PORD (**P**aderborn **ORD**ering Tools) enthaltenen Programme **pord** und **multipord** vor. Alle Funktionen, die in diesen Programmen aufgerufen werden, sind Teil der Bibliothek und können damit auch von jedem anderen Programm benutzt werden. Eine vollständige Beschreibung der Funktionen und ihrer Parameter findet man in [131]. Den Schwerpunkt dieses Abschnitts bilden die experimentellen Ergebnissen in 4.3.2. Durch sie wird eindrucksvoll die Leistungsfähigkeit der in PORD enthaltenen Ordering-Algorithmen demonstriert.

4.3.1 Die Programme pord und multipord

Im wesentlichen realisieren die Programme **pord** und **multipord** die in den Abbildungen 4.5 und 4.14 dargestellten Funktionen MULTISECTION und TRISTAGEMULTISECTION. Die Parameter ord_1 und ord_2 können dabei die Werte AMD, AMF, AMMF, AMIND und MF annehmen. Dabei ist zu beachten, daß der in die Berechnung von $\mathrm{score}_{\mathrm{AMF}}$, $\mathrm{score}_{\mathrm{AMMF}}$ und $\mathrm{score}_{\mathrm{AMIND}}$ ein-

gehende externe Knotengrad d nur approximativ berechnet wird. Dies hat jedoch keinen Einfluß auf die Effektivität der Auswahlstrategien (vgl. auch Rothberg und Eisenstat [125]). Im Programm **pord** ist für ord_2 zusätzlich der Wert ND erlaubt. In diesem Fall werden die Separatoren entsprechend ihrer Rekursionstiefe eliminiert. Der Parameter ord_Φ kann die Werte QMD, QMRDV und QRAND annehmen. Darüber hinaus führen die Programme **pord** und **multipord** ein Preprocessing und ein Postprocessing durch. Innerhalb des Preprocessings wird untersucht, ob der Graph G nicht unterscheidbare Knoten enthält. Dies erscheint zunächst recht unwahrscheinlich zu sein. In der numerischen Praxis tauchen jedoch immer wieder Matrizen auf, in denen aufeinanderfolgende Spalten die gleiche Nichtnullstruktur besitzen. Dies gilt insbesondere für Matrizen, die bei einer Diskretisierung mittels finiter Differenzen entstehen, wenn an jedem Punkt des Diskretisierungsgitters mehrere physikalische Größen wie z. B. Druck, Temperatur oder Geschwindigkeit zu berechnen sind. Durch Zusammenfassen der nicht unterscheidbaren Knoten kann der Graph in der Regel erheblich verkleinert werden.

Sei $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$. Zur Identifizierung der nicht unterscheidbaren Knoten wird zunächst für alle $v_i \in V$ der Wert

$$\mathrm{hash}(v_i) = i + \sum_{v_j \in \mathrm{adj}_G(v_i)} j$$

berechnet. Ein bezüglich v_i nicht unterscheidbarer Knoten v_j erfüllt dann die Bedingungen

$$v_j \in \mathrm{adj}_G(v_i) \text{ und } \deg_G(v_j) = \deg_G(v_i) \text{ und } \mathrm{hash}(v_j) = \mathrm{hash}(v_i).$$

Um alle nicht von v_i unterscheidbaren Knoten zu finden, muß man also nur die Adjazenzliste von v_i durchlaufen und für jeden Knoten v_j mit $\deg_G(v_j) = \deg_G(v_i)$ und $\mathrm{hash}(v_j) = \mathrm{hash}(v_i)$ testen, ob $\mathrm{adj}_G(v_i) \cup \{v_i\} = \mathrm{adj}_G(v_j) \cup \{v_j\}$ gilt (vgl. auch Ashcraft [3], Damhaug [27] sowie Hendrickson und Rothberg [72]). Alle Knoten v_j, die den Test bestehen, werden mit v_i zu einem Superknoten verschmolzen. Ganz ähnlich werden die Superknoten in den Bottom-Up-Algorithmen und die Supervariablen in dem neuen Multilevel-Verfahren bestimmt.

Mit Hilfe der Superknoten kann ein komprimierter Graph G_c konstruiert werden. Dabei wird jeder Superknoten I durch einen logischen Knoten mit Gewicht $|I|$ ersetzt. Der komprimierte Graph G_c dient dann als Eingabe für die Funktionen MULTISECTION und TRISTAGEMULTISECTION. Dies ist problemlos möglich, da sowohl die Bottom-Up-Algorithmen als auch die Funktion SEPARATOR knotengewichtete Graphen als Eingabe akzeptieren. Man erhält so ein Ordering π_c für die Knoten des komprimierten Graphen G_c.

In dem Postprocessing Schritt wird das Ordering π_c auf die Knotenmenge V des ursprünglichen Graphen G erweitert. Seien dazu u_c und v_c zwei Knoten des komprimierten Graphen mit $\pi_c(u_c) < \pi_c(v_c)$. Es sei angenommen, daß u_c die nicht unterscheidbaren Knoten $\{u_1, \ldots, u_r\} \subset V$ und v_c die nicht unterscheidbaren Knoten $\{v_1, \ldots, v_s\} \subset V$ repräsentiert. In dem erweiterten Ordering π werden die nicht unterscheidbaren Knoten aufeinanderfolgend numeriert, also

$\pi(u_i) = \pi(u_1) + i - 1, 1 < i \leq r$ und analog $\pi(v_j) = \pi(v_1) + j - 1, 1 < j \leq s$. Des weiteren gilt in dem erweiterten Ordering $\pi(u_r) < \pi(v_1)$.

4.3.2 Experimentelle Ergebnisse

In diesem Abschnitt stellen wir einige experimentelle Ergebnisse vor, die die Leistungsfähigkeit der in PORD enthaltenen Ordering-Algorithmen unter Beweis stellen. Wir benutzen dazu einen weit verbreiteten und frei verfügbaren Satz von Benchmark-Matrizen. Alle Experimente wurden auf einer SUN Ultra mit 296 MHz UltraSPARC-II Prozessor und zwei GByte Hauptspeicher durchgeführt. Im folgenden werden wir zur Evaluierung eines Orderings immer die Anzahl der zur Faktorisierung benötigten Multiplikations- und Additionsoperationen heranziehen. Wie bereits in Kapitel 3 am Beispiel des quadratischen Gitters gesehen, wird die Laufzeit eines Faktorisierungsalgorithmus im wesentlichen durch die Anzahl der arithmetischen Operationen bestimmt.

4.3.2.1 Vorstellung der Testmatrizen

Für unsere Experimente haben wir einen Satz großer, praxisnaher Testmatrizen ausgewählt. Die ersten zwei Matrizen wurden von uns selbst generiert. Es handelt sich um die Laplace-Matrix eines 127×127-Gitters mit 5-Punkte- (GRID127x127) bzw. 9-Punkte-Stern (MESH127x127). Die 15 BCSSTK-Matrizen stammen aus der bekannten Harwell-Boeing-Collection. Eine detailierte Beschreibung dieser Matrizen findet man in [34]. MAT02HBF und MAT03HBF wurden uns von einer Consulting-Agentur der deutschen Automobilindustrie zugeschickt. Es handelt sich hierbei um Matrizen, die aus einem Simulationsprogramm für Crash-Tests extrahiert wurden. Bei den Matrizen BRACK2, WAVE, HERMES, CYL3 und DIME20 handelt es sich um dreidimensionale FEM-Gitter. Die Matrix CRACK stellt ein zweidimensionales FEM-Gitter dar. Die Matrizen BRACK2, CRACK, WAVE stammen von der Carnegie-Mellon-University. Die Matrix HERMES ist von der Michigan-State-University und stellt ein dreidimensionales Modell des europäischen Raumgleiters dar. Die letzten zwei Matrizen CYL3 und DIME20 wurden uns von C. Walshaw, Universität von Southampton, zur Verfügung gestellt. Alle restlichen Matrizen stammen aus der Sparse-Matrix-Collection [28] von Tim Davis. Hierbei handelt es sich um Matrizen, die aus kommerziellen Anwendungen zur Lösung von Problemen aus den Bereichen der Struktur- und Strömungsmechanik extrahiert wurden.

Tabelle 4.1 zeigt einige wesentliche Eigenschaften der Testmatrizen. Die ersten zwei Spalten geben an, wieviele Knoten und Kanten der aus der Matrix A abgeleitete Graph G enthält. Die nächsten zwei Spalten zeigen die Größe des komprimierten Graphen G_c. In Spalte 5 ist angegeben, wieviele von null verschiedene Einträge der Cholesky-Faktor L von A enthält, wenn zur Bestimmung des Orderings der Approximate-Minimum-Degree-Algorithmus von Amestoy et al. [1] benutzt wird. Spalte 6 zeigt wieviele Multiplikations- und Additionsoperationen in die-

	G		G_c		AMD	
Matrix	$\|V\|$	$\|E\|$	$\|V_c\|$	$\|E_c\|$	$\eta(L)/10^3$	$\theta(L)/10^6$
GRID127x127	16129	32004	16129	32004	346	27
MESH127x127	16129	63756	16129	63756	527	43
BCSSTK15	3948	56934	3948	56934	624	155
BCSSTK16	4884	142747	1778	18251	763	162
BCSSTK17	10974	208838	5219	40531	985	138
BCSSTK18	11948	68571	10926	61086	625	127
BCSSTK23	3134	42044	2930	17628	428	125
BCSSTK24	3562	78174	892	6378	270	31
BCSSTK25	15439	118401	13183	80982	1464	316
BCSSTK29	13992	302748	10202	156923	1760	467
BCSSTK30	28924	1007284	9289	111442	3786	947
BCSSTK31	35588	572914	17403	144403	5281	2593
BCSSTK32	44609	985046	14821	113487	5002	989
BCSSTK33	8738	291583	4344	82142	2480	1140
BCSSTK35	30237	709963	6611	32967	2725	399
BCSSTK36	23052	560044	4351	18583	2719	616
BCSSTK37	25503	557737	7093	44462	2755	535
BCSSTK38	8032	173714	3456	40656	718	115
MAT02HBF	46949	1117809	6707	19938	5057	1344
MAT03HBF	73752	1761718	10536	31438	10061	4184
STRUCT3	53570	560062	41644	340543	5040	1096
STRUCT4	4350	116724	4350	116724	2357	2004
PWT	36519	144794	36515	144774	1556	173
BRACK2	62631	366559	62631	366559	7275	3085
CRACK	10240	30380	10240	30380	163	8
3DTUBE	45330	1584144	15909	181865	26310	30053
CFD1	70656	878854	70656	878854	37663	44556
CFD2	123440	1482229	123440	1482229	74884	136477
CYL3	232362	457853	232362	457853	77440	208480
DIME20	224843	336024	224843	336024	3430	330
GEARBOX	153746	4463329	56175	693142	46325	41121
NASASRB	54870	1311227	24954	275813	11624	4538
WAVE	156317	1059331	156316	1059325	114930	372458
PWTK	217918	5708253	41531	221130	60305	49086
HERMES	320194	3722641	320194	3722641	323055	1434744

Tab. 4.1: Eigenschaften der Benchmark-Matrizen.

sem Fall zur Berechnung von L notwendig sind. Die Werte in Spalte 5 sind in Tausend und die Werte in Spalte 6 in Millionen angegeben. Im folgenden werden wir alle unsere Ergebnisse auch in Relation zu den Werten aus den Spalten 5 und 6 angeben. Da der Algorithmus von Amestoy et al. zu den effektivsten Bottom-up-Algorithmen gehört, ist so eine leichte Einordnung unserer Ergebnisse möglich.

4.3.2.2 Ergebnisse für die Programme pord und multipord

Wie bereits eingangs erwähnt realisieren die Programme **pord** und **multipord** im wesentlichen die Funktionen MULTISECTION und TRISTAGEMULTISECTION. Die algorithmischen Komponenten dieser Funktionen haben wir in Abschnitt 4.2 vorgestellt. Die folgende Liste faßt die wichtigsten Parametereinstellungen der Komponenten zusammen.

- In allen Tests wurden die Funktionen MULTISECTION und TRISTAGEMULTISECTION mit $\mathrm{ord}_\Phi = \mathrm{QMRDV}$ aufgerufen. Obwohl in heterogenen Graphen mit Hilfe der Auswahlstrategie QMD bessere Gebietszerlegungen generiert werden können, haben wir ord_Φ auf QMRDV gesetzt, da für eine Variable V_i die Berechnung von $\mathrm{score}_{\mathrm{QMRDV}}(V_i)$ sehr viel billiger ist als die Berechnung von $\deg(V_i)$.
- In den Funktionen MULTISECTION und TRISTAGEMULTISECTION wird die rekursive Konstruktion der Knotenseparatoren solange fortgesetzt bis alle Teilgraphen weniger als 100 Knoten enthalten. Es werden jedoch höchstens 255 Separatoren berechnet.
- In der Funktion SEPARATOR wird der Vergröberungsprozeß abgebrochen sobald ein Quotientengraph weniger als 200 Elemente besitzt. Es werden jedoch höchstens 15 Quotientengraphen berechnet.
- In der Funktion IMPROVECOLORING erfolgt ein vorzeitiger Abbruch der inneren while-Schleife, falls in 100 aufeinanderfolgenden Iterationen die Partitionierung nicht verbessert werden kann.
- In der Bewertungsfunktion F sind die Parameter τ und ρ auf die Werte 0.5 und 100 gesetzt. Solange also das Gewicht der kleineren Partition mindestens 50 % vom Gewicht der größeren Partition ausmacht, geht in die Bewertung einer Partitionierung nur das Gewicht des Separators ein.

Tabelle 4.2 zeigt die Zahl der zur Berechnung des Cholesky-Faktors benötigten Multiplikations- und Additionsoperationen für das Programm **pord**. Dabei wurde die Funktion MULTISECTION mit den Parametern $\mathrm{ord}_1 = \mathrm{ord}_2 = \mathrm{AMD}$ (Spalte 1), $\mathrm{ord}_1 = \mathrm{ord}_2 = \mathrm{AMMF}$ (Spalte 2), $\mathrm{ord}_1 = \mathrm{AMMF}$, $\mathrm{ord}_2 = \mathrm{MF}$ (Spalte 3) und $\mathrm{ord}_1 = \mathrm{AMMF}$, $\mathrm{ord}_2 = \mathrm{ND}$ (Spalte 4) aufgerufen. Obwohl weitere Kombinationen von Knotenauswahlstrategien möglich sind, haben wir uns auf diese vier beschränkt. In Klammern ist jeweils der Wert in Relation zum Approximate-Minimum-Degree-Verfahren angegeben.

Matrix	(AMD, AMD)	(AMMF, AMMF)	(AMMF, MF)	(AMMF, ND)
GRID127x127	16 (0.59)	16 (0.59)	16 (0.59)	17 (0.63)
MESH127x127	36 (0.84)	38 (0.88)	35 (0.81)	39 (0.91)
BCSSTK15	93 (0.60)	79 (0.51)	86 (0.55)	82 (0.53)
BCSSTK16	112 (0.69)	117 (0.72)	115 (0.71)	135 (0.83)
BCSSTK17	124 (0.90)	123 (0.89)	120 (0.87)	172 (1.25)
BCSSTK18	87 (0.68)	85 (0.67)	82 (0.65)	96 (0.76)
BCSSTK23	98 (0.78)	85 (0.68)	98 (0.78)	90 (0.72)
BCSSTK24	30 (0.97)	30 (0.97)	30 (0.97)	31 (1.00)
BCSSTK25	256 (0.81)	207 (0.65)	230 (0.73)	355 (1.12)
BCSSTK29	360 (0.77)	326 (0.70)	277 (0.59)	336 (0.72)
BCSSTK30	722 (0.76)	702 (0.74)	707 (0.74)	982 (1.04)
BCSSTK31	1226 (0.47)	1215 (0.47)	1184 (0.46)	1291 (0.50)
BCSSTK32	827 (0.84)	774 (0.78)	767 (0.77)	969 (0.98)
BCSSTK33	740 (0.65)	626 (0.55)	619 (0.54)	644 (0.56)
BCSSTK35	374 (0.94)	367 (0.92)	369 (0.92)	384 (0.96)
BCSSTK36	461 (0.75)	458 (0.74)	460 (0.75)	500 (0.81)
BCSSTK37	404 (0.75)	403 (0.75)	388 (0.72)	445 (0.83)
BCSSTK38	91 (0.79)	91 (0.79)	89 (0.77)	106 (0.92)
MAT02HBF	1091 (0.81)	1088 (0.80)	1093 (0.81)	1222 (0.91)
MAT03HBF	2654 (0.63)	2723 (0.65)	2473 (0.59)	2666 (0.64)
STRUCT3	717 (0.65)	730 (0.67)	664 (0.61)	731 (0.67)
STRUCT4	574 (0.29)	541 (0.27)	617 (0.31)	504 (0.25)
PWT	108 (0.62)	109 (0.63)	107 (0.62)	110 (0.64)
BRACK2	1923 (0.62)	1610 (0.52)	1661 (0.53)	1982 (0.64)
CRACK	7 (0.87)	7 (0.87)	6 (0.75)	7 (0.87)
3DTUBE	13235 (0.44)	14839 (0.49)	11437 (0.38)	12303 (0.41)
CFD1	9885 (0.22)	8814 (0.20)	8799 (0.20)	11302 (0.25)
CFD2	34421 (0.25)	27978 (0.20)	27902 (0.20)	28636 (0.21)
CYL3	57971 (0.29)	45386 (0.22)	41372 (0.20)	39791 (0.19)
DIME20	175 (0.53)	175 (0.53)	167 (0.51)	191 (0.58)
GEARBOX	18034 (0.44)	17404 (0.42)	17505 (0.43)	17987 (0.44)
NASASRB	2839 (0.63)	2613 (0.58)	2582 (0.56)	3437 (0.76)
WAVE	160843 (0.43)	119694 (0.32)	108804 (0.29)	97480 (0.26)
PWTK	23019 (0.47)	22658 (0.46)	22323 (0.45)	23119 (0.47)
HERMES	326902 (0.23)	266255 (0.19)	268303 (0.19)	265133 (0.18)
Durchschnitt	(0.63)	(0.60)	(0.59)	(0.67)

Tab. 4.2: Anzahl der zur Faktorisierung benötigten Multiplikations- und Additionsoperationen (in Mio.) in Abhängigkeit von den verwendeten Knotenauswahlstrategien. Alle Orderings wurden mit Hilfe des Programms **pord** bestimmt.

	Iteration j							
Matrix	7	6	5	4	3	2	1	0
GRID127x127	–	**16**	**16**	**16**	**16**	17	17	17
MESH127x127	–	38	45	42	**35**	36	39	39
BCSSTK15	–	–	–	79	**78**	**78**	82	82
BCSSTK16	–	–	–	**117**	120	130	135	135
BCSSTK17	–	–	**123**	137	156	171	171	172
BCSSTK18	–	85	87	**81**	86	91	96	96
BCSSTK23	–	–	–	–	85	**84**	91	90
BCSSTK24	–	–	–	–	**30**	31	31	31
BCSSTK25	–	**207**	216	244	309	326	355	355
BCSSTK29	–	326	**309**	321	330	316	336	336
BCSSTK30	–	**702**	725	809	913	973	982	986
BCSSTK31	**1215**	1266	1317	1261	1289	1285	1291	1291
BCSSTK32	**774**	**774**	794	833	929	968	969	969
BCSSTK33	–	–	626	626	**615**	644	644	644
BCSSTK35	–	**367**	376	378	383	385	384	384
BCSSTK36	–	–	**458**	476	475	497	500	500
BCSSTK37	–	403	**393**	403	413	433	445	445
BCSSTK38	–	–	**91**	93	97	103	106	106
MAT02HBF	–	**1088**	1107	1128	1140	1192	1222	1222
MAT03HBF	2723	2726	**2542**	2790	2619	2659	2663	2666
STRUCT3	730	**691**	712	707	725	731	731	731
STRUCT4	–	–	–	541	542	523	**503**	504
PWT	109	111	**108**	**108**	109	110	110	110
BRACK2	**1610**	1646	1633	1655	1817	1981	1982	1982
CRACK	–	7	7	**6**	7	7	7	7
3DTUBE	14839	14953	12907	12366	13392	**12303**	**12303**	**12303**
CFD1	**8814**	8961	10011	10533	10798	10947	11302	11302
CFD2	27978	26379	27616	**26141**	26554	28371	28636	28636
CYL3	45386	40806	41324	41525	40623	40190	**39791**	**39791**
DIME20	175	**174**	182	183	187	190	191	191
GEARBOX	**17404**	17643	17658	17748	17791	17987	17987	17987
NASASRB	**2613**	2706	2767	2955	3201	3280	3437	3437
WAVE	119694	98274	100509	101306	99559	97463	**97472**	97480
PWTK	**22658**	22680	22868	23068	23088	23042	23119	23119
HERMES	266255	260469	260060	**250740**	252313	255478	265133	265133

Tab. 4.3: Anzahl der Multiplikations- und Additionsoperationen (in Mio.) bezüglich aller innerhalb von TRISTAGEMULTISECTION generierten Orderings. Dabei gilt immer $\mathrm{ord}_1 = \mathrm{ord}_2 = \mathrm{AMMF}$. Die besten Ergebnisse sind in Fettdruck dargestellt.

Bereits ein erster Blick auf die Tabelle 4.2 zeigt, daß das neue Multisection-Verfahren für alle vier Parameterkombinationen sehr gute Orderings generiert. Im Vergleich zum Verfahren von Amestoy et al. kann die Zahl der zur Berechnung des Cholesky-Faktors benötigten Operationen im Schnitt um bis zu 41 % reduziert werden. Bemerkenswert ist insbesondere, daß alle „echten" Multisection-Orderings (hier gilt $\mathrm{ord}_2 \neq \mathrm{ND}$) besser sind als die entsprechenden AMD-Orderings. Von unserem Algorithmus werden also *konsistent* bessere Orderings generiert.

Rothberg und Eisenstat [125] haben gezeigt, daß ein Bottom-up-Algorithmus, der auf den Knotenauswahlstrategien AMMF oder MF basiert, sehr viel bessere Orderings erzeugt als ein Minimum-Degree-Algorithmus. Daher liegt es nahe, die Auswahlstrategien auch in unserem Multisection-Verfahren zu benutzen. In der Tat zeigt ein Vergleich zwischen den Spalten 1 und 2 der Tabelle 4.2, daß durch den Übergang von $\mathrm{ord}_1 = \mathrm{ord}_2 = \mathrm{AMD}$ auf $\mathrm{ord}_1 = \mathrm{ord}_2 = \mathrm{AMMF}$ die Qualität der generierten Orderings verbessert wird. Eine weitere Verbesserung erreicht man, wenn zur Elimination der Knotenseparatoren die Auswahlstrategie MF benutzt wird. In der Regel enthält der Schur-Komplement Graph G_Φ nur noch wenige Superknoten. Daher ist die Kombination $\mathrm{ord}_1 = \mathrm{AMMF}$, $\mathrm{ord}_2 = \mathrm{MF}$ auch in der Praxis einsetzbar. Mit Hilfe dieser Kombination kann die Anzahl der zur Berechnung von L benötigten Multiplikations- und Additionsoperationen am stärksten reduziert werden.

Interessant ist auch ein Vergleich der Spalten 2 und 4. Werden die Separatoren nicht mit Hilfe der Auswahlstrategie AMMF, sondern wie bei einem Nested-Dissection-Ordering entsprechend ihrer Rekursionstiefe numeriert, so verschlechtert sich die Qualität der generierten Orderings zum Teil erheblich. Auffällig sind insbesondere die Ergebnisse für BCSSTK17, BCSSTK25 und BCSSTK30. Hier sind zum ersten Mal die von **pord** berechneten Orderings schlechter als die entsprechenden AMD-Orderings. Dies kann jedoch sehr einfach erklärt werden. In den Graphen sind die obersten Separator wie bei einem $h \times n$-Gitter mit $h \ll n$ nebeneinanderliegend angeordnet. Werden dann die Separatoren entsprechend ihrer Rekursionstiefe eliminiert, so entsteht in dem bereits chordalen Eliminationsgraphen zusätzlicher Fill-in (vgl. auch Abbildung 4.4).

Die durch das Nested-Dissection-Ordering vorgegebene Eliminationssequenz kann jedoch auch zu besseren Ergebnissen führen. Dies ist beispielsweise bei den Matrizen CYL3 und WAVE der Fall. Überhaupt zeigt ein Vergleich der letzten drei Spalten von Tabelle 4.2, daß die Ordering-Ergebnisse trotz identischer Separatoren und Auswahlstrategie ord_1 stark variieren können. Da die meisten Multiplikations- und Additionsoperationen bei der Faktorisierung der letzten Spalten einer Matrix anfallen, spielt die Reihenfolge, in der die Knotenseparatoren eliminiert werden eine wichtige Rolle. Mit Hilfe des Programms **multipord** ist es möglich, mehrere Varianten sehr effizient durchzurechnen. Tabelle 4.3 zeigt für jede Matrix die Anzahl der Multiplikations- und Additionsoperationen aller innerhalb von TRISTAGEMULTISECTION generierten Orderings. Dabei gilt immer $\mathrm{ord}_1 = \mathrm{ord}_2 = \mathrm{AMMF}$. Da maximal 255 Separatoren konstruiert werden, hat die Variable k höchstens den Wert sieben. Daher werden in der for-Schleife (vgl. Zeilen 04–13) der Funktion TRISTAGEMULTISECTION maximal acht Orderings berechnet und evaluiert. Die

Anzahl der Multiplikations- und Additionsoperationen bezüglich des in Iteration $j = 7, \ldots, 0$ generierten Orderings findet man in der entsprechenden Spalte der Tabelle. Die Einträge der letzten Spalte ($j = 0$) entsprechen dabei den Einträgen der Spalte $(\mathrm{AMMF}, \mathrm{ND})$ von Tabelle 4.2. Faßt man in jeder Zeile die am weitesten links stehenden Einträge zusammen, so erhält man die Spalte $(\mathrm{AMMF}, \mathrm{AMMF})$ von Tabelle 4.2.

4.3.2.3 Vergleich mit anderen Ordering-Codes

In diesem Abschnitt vergleichen wir die von **pord** und **multipord** generierten Orderings mit denen der Programme, METIS [79], SCOTCH [109] und SPOOLES [8]. Genaugenommen handelt es sich nicht um Programme, sondern um Programmbibliotheken, die eine Reihe zusätzlicher Funktionen bereitstellen. METIS und SCOTCH enthalten Heuristiken zur Graph-Partitionierung und SPOOLES Funktionen zur Lösung dünn besetzter Gleichungssysteme. In jeder Bibliothek sind sogenannte *stand-alone* Programme enthalten, mit deren Hilfe die in der Tabelle 4.4 angegebenen Werte produziert wurden. Bevor wir auf diese Tabelle eingehen, wollen wir kurz die charakteristischen Merkmale der stand-alone Programme beschreiben.

METIS Die Bibliothek METIS wurde von Karypis und Kumar an der Universität von Minnesota entwickelt. Die Quelldateien sind frei verfügbar. METIS enthält zwei stand-alone Programme zur Berechnung eines Orderings. In beiden Programmen werden die Knotenseparatoren mit Hilfe eines Multilevel-Verfahrens bestimmt, das auf einer Matching-Technik basiert. Im Programm **onmetis** wird der Knotenseparator direkt, im Programm **oemetis** anhand eines Kantenseparators bestimmt. In der Regel liefert **onmetis** sehr viel bessere Orderings. Deswegen wird in der Literatur METIS mit dem Programm **onmetis** gleichgesetzt. Zur Elimination der Knoten in den Teilgraphen benutzt METIS einen Multiple-Minimum-Degree-Algorithmus. Die Separatorknoten werden entsprechend ihrer Rekursionstiefe eliminiert. Damit stellt METIS einen state-of-the-art Nested-Dissection-Algorithmus dar.

SCOTCH Die Bibliothek SCOTCH wurde von Pellegrini an der Universität von Bordeaux entwickelt. Auch hier sind die Quelldateien frei verfügbar. Zur Berechnung eines Orderings enthält SCOTCH das stand-alone Programm **ord**. Über eine Vielzahl von Parametern kann die genaue Vorgehensweise bei der Berechnung des Orderings spezifiziert werden. Die in der Tabelle 4.4 angegebenen Werte wurden mit Hilfe der von Pellegrini vorgeschlagenen Einstellungen ermittelt. Wie METIS, so stellt auch SCOTCH einen state-of-the-art Nested-Dissection-Algorithmus dar. Zur Konstruktion der Knotenseparatoren wird wieder ein auf einer Matching-Technik basierender Multilevel-Ansatz benutzt. Im Gegensatz zu METIS werden die Knoten in den Teilgraphen mit Hilfe des Approximate-Minimum-Degree-Algorithmus von Amestoy et al. eliminiert (vg. auch Pellegrini et al. [111]).

SPOOLES Die von Ashcraft und Grimes entwickelte SPOOLES-Bibliothek enthält eine Vielzahl von Algorithmen zur Lösung dünn besetzter Gleichungssysteme. Die gesamte Bibliothek

basiert auf einem objektorientierten Programmieransatz. Zur Vermeidung von Performanzverlusten sind jedoch alle Objekte und Methoden in der Programmiersprache C implementiert. Die Quelldateien sind frei verfügbar. SPOOLES enthält mehrere stand-alone Programme. Die in der Tabelle 4.4 angegebenen Werte wurden mit Hilfe des Programms **ddsep** ermittelt. Dieses Programm wurde bereits in einer früheren Arbeit von Ashcraft und Liu [14] separat vorgestellt. In **ddsep** werden die Knotenseparatoren nach dem in Abschnitt 4.1.3 beschriebenen zweistufigen Verfahren konstruiert. Die Konstruktion der Gebietszerlegung $(\widehat{V}, D_1, \ldots, D_r)$ basiert auf einer randomisierten Greedy-Methode. Deshalb haben wir das Programm für jeden Graphen elfmal gestartet. Die in den Tabellen 4.4 und 4.5 angegebenen Werte stellen jeweils den Durchschnitt der elf Läufe dar. Zur Elimination der Knoten in den Teilgraphen und in dem Multisektor wird Lius Multiple-Minimum-Degree-Algorithmus benutzt.

Tabelle 4.4 zeigt die Überlegenheit unseres Multisection-Verfahrens. Die Werte für **pord** entsprechen den Werten in Spalte 2 von Tabelle 4.2 und die Werte für **multipord** den hervorgehobenen Zahlen in Tabelle 4.3. Während die von METIS (Version 4.0), SCOTCH (Version 3.3) und SPOOLES (Version 2.2) generierten Orderings im Durchschnitt 25 %, 28 % und 31 % weniger Operationen verursachen als der AMD-Algorithmus von Amestoy et al., erzielt **multipord** eine Verbesserung von 42 %. Insbesondere sei darauf hingewiesen, daß alle von **pord** und **multipord** generierten Orderings besser sind als die entsprechenden AMD-Orderings. Dies ist für kein anderes Ordering-Verfahren der Fall.

Interessant ist in diesem Zusammenhang auch ein Vergleich der Nested-Dissection-Variante von **pord** (Spalte 4 in Tabelle 4.2) mit METIS und SCOTCH. Alle drei können als state-of-the-art Nested-Dissection-Algorithmen bezeichnet werden. Während METIS und SCOTCH die Anzahl der arithmetischen Operationen im Schnitt um 25 % bzw. 28 % reduzieren, erreicht die Nested-Dissection-Variante von **pord** eine Verbesserung um 33 %. Lediglich für drei Matrizen ist das von **pord** generierte Nested-Dissection-Ordering schlechter als das entsprechende AMD-Ordering. Bei METIS ist dies für acht und bei SCOTCH für sieben Matrizen der Fall.

Tabelle 4.5 stellt die Laufzeiten der Ordering-Algorithmen einander gegenüber. In Klammern ist wieder der Wert in Relation zu dem Approximate-Minimum-Degree-Algorithmus von Amestoy et al. angegeben. Ein Vergleich zwischen **pord** und **multipord** zeigt, daß der durch die Generierung und Evaluierung von bis zu acht Bottom-Up-Orderings entstehende Mehraufwand sehr gering ist. Insgesamt läßt sich feststellen, daß von den fünf Ordering-Algorithmen METIS die geringste Laufzeit benötigt. Bedingt durch den komplexeren Schrumpfungs- und Optimierungsprozeß können die Laufzeiten von **pord** bzw. **multipord** im Vergleich zu METIS um den Faktor zwei (z. B. für BRACK2 und HERMES) oder mehr (z. B. für CYL3) anwachsen. Im Durchschnitt liegen die Laufzeiten von **pord** und **multipord** jedoch nur leicht über denen von METIS und sind deutlich geringer als die von SCOTCH und SPOOLES.

Auffallend ist, daß die Laufzeiten der fünf Ordering-Algorithmen um ein vielfaches höher sind als die des Approximate-Minimum-Degree-Algorithmus. Im Falle von SCOTCH können sich die Laufzeiten um bis zu einem Faktor von 30 erhöhen. Es stellt sich die Frage, ob die Zeitersparnis bei der numerischen Faktorisierung groß genug ist, um den Mehraufwand zur Berechnung eines besseren Orderings zu rechtfertigen. Die Antwort ist ein klares Ja. Während sich die Laufzeiten zur Berechnung eines Orderings im Bereich von Sekunden bewegen, benötigt die numerische Faktorisierung Laufzeiten im Bereich von mehreren Minuten. Bei sehr großen Matrizen kann die Laufzeit auf mehr als eine Stunde anwachsen.

Wir wollen das Verhältnis der Laufzeiten an der Matrix CFD1 veranschaulichen. Bei Verwendung des von **multipord** generierten Orderings müssen zur Berechnung des Cholesky-Faktors $8814 \cdot 10^6$ arithmetische Operationen durchgeführt werden. Der von uns implementierte Multifrontal-Algorithmus (vgl. Abschnitt 5.1.3) benötigt dazu 87.09 Sekunden. Zusammen mit dem Ordering ergibt sich eine Laufzeit von 104.52 Sekunden. Benutzt man das von METIS generierte Ordering, so ergeben sich die folgenden Werte:

- Anzahl der Operationen: $16345 \cdot 10^6$
- numerische Faktorisierung: 184.47 Sek.
- numerische Faktorisierung plus Ordering: 197.25 Sek.

Der Approximate-Minimum-Degree-Algorithmus von Amestoy et al. benötigt zur Berechnung eines Orderings für CFD1 nur drei Sekunden. Auf Basis dieses Orderings ergeben sich die Werte:

- Anzahl der Operationen: $44556 \cdot 10^6$
- numerische Faktorisierung: 514.90 Sek.
- numerische Faktorisierung plus Ordering: 517.74 Sek.

Ein Vergleich der Laufzeiten zeigt, daß es sich auf jeden Fall lohnt, mehr Zeit in die Berechnung eines guten Orderings zu investieren. Darüber hinaus unterstreicht der Vergleich noch einmal die hohe praktische Relevanz des neuen Ordering-Verfahrens.

Name	METIS-4.0	SCOTCH-3.3	SPOOLES-2.2	**pord**	**multipord**
GRID127x127	22 (0.81)	25 (0.93)	20 (0.74)	16 (0.59)	16 (0.59)
MESH127x127	37 (0.86)	40 (0.93)	43 (1.00)	38 (0.88)	35 (0.81)
BCSSTK15	87 (0.56)	93 (0.60)	95 (0.61)	79 (0.51)	78 (0.50)
BCSSTK16	140 (0.86)	140 (0.86)	129 (0.79)	117 (0.72)	117 (0.72)
BCSSTK17	184 (1.33)	161 (1.16)	135 (0.98)	123 (0.89)	123 (0.89)
BCSSTK18	101 (0.80)	77 (0.60)	84 (0.66)	85 (0.67)	81 (0.64)
BCSSTK23	98 (0.78)	94 (0.75)	91 (0.72)	85 (0.68)	84 (0.67)
BCSSTK24	34 (1.09)	35 (1.13)	38 (1.22)	31 (0.97)	30 (0.97)
BCSSTK25	380 (1.20)	348 (1.10)	235 (0.74)	207 (0.65)	207 (0.65)
BCSSTK29	345 (0.74)	327 (0.70)	341 (0.73)	326 (0.70)	309 (0.66)
BCSSTK30	1203 (1.27)	1114 (1.18)	833 (0.88)	702 (0.74)	702 (0.74)
BCSSTK31	1165 (0.45)	1219 (0.47)	1530 (0.59)	1215 (0.47)	1215 (0.47)
BCSSTK32	1213 (1.23)	1175 (1.19)	866 (0.88)	774 (0.78)	774 (0.78)
BCSSTK33	909 (0.78)	674 (0.59)	739 (0.65)	626 (0.55)	615 (0.54)
BCSSTK35	523 (1.31)	422 (1.06)	393 (0.98)	367 (0.92)	367 (0.92)
BCSSTK36	615 (1.00)	583 (0.95)	496 (0.81)	458 (0.74)	458 (0.74)
BCSSTK37	694 (1.30)	653 (1.22)	433 (0.81)	403 (0.75)	397 (0.74)
BCSSTK38	135 (1.17)	108 (0.94)	103 (0.90)	91 (0.79)	91 (0.79)
MAT02HBF	1192 (0.87)	1099 (0.82)	1156 (0.86)	1088 (0.81)	1088 (0.81)
MAT03HBF	2724 (0.65)	2924 (0.70)	3607 (0.86)	2723 (0.65)	2542 (0.61)
STRUCT3	826 (0.75)	857 (0.78)	773 (0.70)	730 (0.67)	691 (0.63)
STRUCT4	535 (0.27)	541 (0.27)	691 (0.34)	541 (0.27)	503 (0.25)
PWT	110 (0.64)	101 (0.58)	108 (0.62)	109 (0.63)	108 (0.62)
BRACK2	1908 (0.62)	1821 (0.59)	1900 (0.62)	1610 (0.52)	1610 (0.52)
CRACK	7 (0.87)	7 (0.87)	7 (0.87)	7 (0.87)	6 (0.75)
3DTUBE	12071 (0.40)	15834 (0.53)	15523 (0.52)	14839 (0.49)	12303 (0.41)
CFD1	16345 (0.37)	15027 (0.34)	10509 (0.24)	8814 (0.20)	8814 (0.20)
CFD2	31024 (0.23)	35659 (0.26)	35798 (0.26)	27978 (0.20)	26141 (0.19)
CYL3	32164 (0.15)	31670 (0.15)	80818 (0.39)	45386 (0.22)	39791 (0.19)
DIME20	196 (0.59)	181 (0.55)	235 (0.71)	175 (0.53)	175 (0.53)
GEARBOX	20390 (0.50)	24755 (0.60)	21516 (0.52)	17404 (0.42)	17404 (0.42)
NASASRB	3494 (0.77)	3748 (0.83)	2801 (0.62)	2613 (0.58)	2613 (0.58)
WAVE	120180 (0.32)	98547 (0.26)	188316 (0.50)	119694 (0.32)	97463 (0.26)
PWTK	22039 (0.45)	23275 (0.47)	28313 (0.58)	22658 (0.46)	22658 (0.46)
HERMES	258970 (0.18)	368863 (0.26)	518549 (0.36)	266255 (0.19)	250740 (0.17)
Durchschnitt	(0.75)	(0.72)	(0.69)	(0.60)	(0.58)

Tab. 4.4: Vergleich der zur Faktorisierung benötigten Operationen (in Mio.).

Matrix	METIS-4.0	SCOTCH-3.3	SPOOLES-2.2	**pord**	**multipord**
GRID127x127	0.81 (4.1)	2.22 (11.1)	1.58 (7.9)	1.27 (6.4)	1.35 (6.7)
MESH127x127	1.04 (4.7)	2.90 (13.2)	2.36 (10.1)	1.45 (6.6)	1.58 (7.2)
BCSSTK15	0.52 (4.3)	1.96 (16.3)	1.13 (9.4)	0.52 (4.3)	0.60 (5.0)
BCSSTK16	0.21 (2.6)	0.52 (6.5)	0.48 (6.0)	0.19 (2.4)	0.25 (3.1)
BCSSTK17	0.76 (5.1)	1.50 (10.0)	1.13 (7.5)	0.61 (4.1)	0.71 (4.7)
BCSSTK18	1.02 (4.1)	3.70 (14.8)	2.84 (11.4)	1.25 (5.0)	1.57 (6.3)
BCSSTK23	0.25 (2.5)	1.23 (12.3)	0.58 (5.8)	0.29 (2.9)	0.37 (3.7)
BCSSTK24	0.09 (3.0)	0.20 (6.7)	0.16 (5.3)	0.07 (2.3)	0.11 (3.7)
BCSSTK25	1.63 (4.9)	5.81 (17.6)	3.71 (11.2)	1.69 (5.1)	1.90 (5.8)
BCSSTK29	1.65 (5.5)	9.09 (30.3)	3.29 (11.0)	1.35 (4.5)	1.66 (5.5)
BCSSTK30	2.26 (4.2)	4.73 (8.9)	4.30 (8.1)	1.69 (3.2)	1.85 (3.5)
BCSSTK31	3.35 (5.2)	6.61 (10.2)	5.60 (8.6)	2.96 (4.6)	3.44 (5.3)
BCSSTK32	2.86 (4.8)	4.95 (8.2)	4.30 (7.2)	2.28 (3.8)	2.90 (4.8)
BCSSTK33	0.77 (3.7)	2.56 (12.2)	2.61 (12.4)	0.78 (3.7)	0.91 (4.3)
BCSSTK35	0.91 (2.9)	1.57 (5.1)	1.70 (5.5)	0.81 (2.6)	0.95 (3.1)
BCSSTK36	0.56 (2.4)	0.95 (4.1)	1.17 (5.1)	0.48 (2.1)	0.61 (2.7)
BCSSTK37	1.03 (3.7)	1.90 (6.8)	1.71 (6.1)	0.87 (3.1)	0.97 (3.5)
BCSSTK38	0.85 (5.7)	1.63 (10.8)	1.00 (6.7)	0.48 (3.2)	0.61 (4.1)
MAT02HBF	0.90 (2.2)	1.37 (3.3)	1.73 (4.2)	0.87 (2.2)	0.97 (2.4)
MAT03HBF	1.15 (1.9)	2.35 (3.9)	2.70 (4.4)	1.46 (2.4)	1.66 (2.7)
STRUCT3	5.18 (4.9)	18.40 (17.5)	14.00 (13.3)	7.52 (7.2)	8.81 (8.4)
STRUCT4	1.37 (5.1)	3.58 (13.3)	5.43 (20.1)	1.06 (3.9)	1.33 (4.9)
PWT	0.96 (1.7)	6.30 (11.1)	6.30 (11.1)	3.87 (6.8)	4.03 (7.1)
BRACK2	7.14 (3.5)	23.95 (11.7)	17.32 (8.4)	12.48 (6.1)	14.11 (6.9)
CRACK	0.59 (3.3)	1.69 (9.4)	1.06 (5.9)	0.87 (4.8)	0.98 (5.4)
3DTUBE	3.73 (3.9)	7.57 (8.0)	6.70 (7.1)	2.81 (3.0)	3.40 (3.6)
CFD1	12.78 (4.3)	37.57 (12.5)	36.73 (12.2)	15.71 (5.2)	17.43 (5.8)
CFD2	22.34 (5.0)	65.29 (14.5)	64.11 (14.2)	27.61 (6.1)	31.64 (7.0)
CYL3	21.99 (1.3)	77.81 (4.6)	71.53 (4.2)	68.59 (4.0)	71.99 (4.2)
DIME20	13.70 (3.4)	38.00 (9.5)	37.50 (9.4)	32.78 (8.2)	33.07 (8.3)
GEARBOX	18.61 (6.5)	27.83 (9.6)	21.36 (7.4)	13.85 (4.8)	14.86 (5.2)
NASASRB	6.30 (6.3)	9.54 (9.5)	7.95 (7.9)	4.68 (4.7)	5.77 (5.8)
WAVE	21.32 (3.6)	72.51 (12.2)	67.81 (11.4)	35.79 (6.0)	38.59 (6.5)
PWTK	16.74 (7.4)	11.53 (5.1)	10.50 (4.7)	6.80 (3.0)	8.01 (3.6)
HERMES	61.78 (3.9)	181.79 (11.4)	220.68 (13.9)	104.81 (6.6)	110.91 (7.0)
Durchschnitt	(4.0)	(10.6)	(8.7)	(4.4)	(5.1)

Tab. 4.5: Vergleich der Laufzeiten (in Sek.).

Kapitel 5

Symbolische und numerische Faktorisierung

Die numerische Faktorisierung stellt den aufwendigsten Schritt zur Lösung eines dünn besetzten, linearen Gleichungssystems dar. Um die Faktorisierung möglichst effizient durchführen zu können, wird zunächst eine geeignete Datenstruktur zur Speicherung der Faktormatrix L bestimmt. Dies ist Aufgabe der symbolischen Faktorisierung. Wichtigste Eingabe der symbolischen Faktorisierung ist der in Abschnitt 4.1.1 beschriebene Eliminationsbaum. Der Eliminationsbaum spielt auch bei der numerischen Faktorisierung eine entscheidene Rolle.

Fast alle aus der Literatur bekannten Faktorisierungsalgorithmen sind spaltenbasiert, d. h. der Cholesky-Faktor L wird Spalte für Spalte berechnet. Die bekanntesten spaltenbasierten Verfahren sind die *Fan-in-* und die *Fan-out-Methode*. Die Fan-in-Methode [6] realisiert die in Abschnitt 2.1 beschriebene Inner-Product-Form der Cholesky-Zerlegung. Demgegenüber basiert die Fan-out-Methode [50] auf der ebenfalls in 2.1 beschriebenen Outer-Product-Form. Beide Methoden sind numerisch äquivalent, d. h. sie benötigen die gleiche Anzahl von Multiplikations- und Additionsoperationen. Sie unterscheiden sich lediglich in der Reihenfolge, in der die Operationen ausgeführt werden. Man erhält so zwei unterschiedliche Ansätze zur Parallelisierung der Cholesky-Zerlegung (vgl. z. B. Ashcraft et al. [6, 5], George et al. [50] oder Heath et al. [67]).

Eine Variante der Fan-out-Methode ist die Multifrontal-Methode [35, 36]. Ursprünglich wurde die Multifrontal-Methode zur Faktorisierung sehr großer Matrizen entwickelt, die nicht vollständig in den Hauptspeicher eines Rechners passen (vgl. z. B. Liu [95] oder Reid [117]). Heute benutzt man die Multifrontal-Methode hauptsächlich zur Faktorisierung dünn besetzter Matrizen. Mit ihrer Hilfe ist es möglich die Faktorisierung einer dünn besetzten Matrix auf die teilweise Faktorisierung mehrerer kleinerer, voll besetzter Matrizen zurückzuführen. Zur Faktorisierung einer dünn besetzten Matrix können dann Programmiertechniken angewandt werden, die eigentlich zur Lösung voll besetzter Systeme entwickelt wurden. Des weiteren läßt sich die Zerlegung einer voll besetzten Matrix sehr effizient auf einem Vektor- oder Parallelrechner durchführen

(vgl. Gallivan et al. [45] oder Kumar et al. [84]). Daher basieren viele parallele Algorithmen zur Faktorisierung dünn besetzter Matrizen auf der Multifrontal-Methode (vgl. z. B. Ashcraft et al. [9], Dongarra und Eisenstat [31], Gilbert und Schreiber [60], Gupta et al. [63, 66], Lucas et al. [100] oder Schulze [130]).

Dieses Kapitel ist wie folgt aufgebaut. In Abschnitt 5.1 stellen wir einen sequentiellen Algorithmus zur symbolischen und numerischen Faktorisierung vor. Dabei gehen wir nochmals auf den Eliminationsbaum ein und zeigen, wie dieser Baum die symbolische und numerische Faktorisierung steuert. Unser Algorithmus basiert auf der Multifrontal-Methode und dient als Ausgangspunkt für die in 5.2 beschriebene Parallelisierung. Bei der Implementierung des parallelen Algorithmus sind wir von einem verteilten System ausgegangen dessen Verbindungsnetzwerk einem *Hypercube* [87] entspricht.

5.1 Der sequentielle Fall

Entscheidend für die Effizienz der numerischen Faktorisierung ist ein effektiver Einsatz der von modernen Computern bereitgestellten Caching-Mechanismen. Dazu muß in kurzen Zeitabständen *(zeitliche Lokalität)* sehr oft auf einen kleinen, begrenzten Speicherbereich *(räumliche Lokalität)* zugegriffen werden. Sowohl die Fan-in- als auch die Fan-out-Methode basieren auf der einfachen BLAS 1 [101] Operation `daxpy`. Daher ist die Anzahl der auf einem Speicherbereich durchgeführten Floating-Point-Operationen sehr gering. Wird die Matrix jedoch in quadratische Blöcke partitioniert, so kann die Operation durch sehr viel effizientere BLAS 3 [30] Routinen wie z. B. `dgemm` ersetzt werden. Man spricht in diesem Fall von einer *blockweisen* Faktorisierung (vgl. auch Ashcraft et al. [9], Ng und Peyton [105] oder Rothberg [121]). Bei den BLAS 3 Routinen ist die Anzahl der auf einem Speicherbereich durchgeführten Floating-Point-Operationen sehr viel höher, so daß der Prozessor-Cache besser ausgelastet wird.

Die Register eines Prozessors bilden den schnellsten Speicher. Um die Verwendung dieses Speichers zu optimieren (register re-use), benutzt man die Technik des *Loop-Unrolling* [31]. Hierbei werden nochmals quadratische Blöcke gebildet, die jetzt jedoch so klein sind, daß die Operationen auf diesen Blöcken „ausprogrammiert“ werden können. Es entstehen also keine zusätzlichen Schleifen.

Ist A dünn besetzt, so werden die Blöcke mit Hilfe der bei der Berechnung eines Orderings entstehenden Superknoten gebildet. Eine besondere Stellung nimmt in diesem Zusammenhang die Multifrontal-Methode ein. Hier ist mit jedem Superknoten eine voll besetzte untere Dreiecksmatrix verbunden. Durch eine teilweise Faktorisierung dieser Matrix erhält man die zu dem Superknoten gehörenden Spalten des Cholesky-Faktors. Der entscheidende Vorteil der Methode besteht darin, daß die voll besetzte Matrix in der Regel so klein ist, daß sie vollständig in den Cache paßt. Obwohl die Verwaltung der Matrizen einen gewissen Overhead erzeugt, gehört die Multifrontal-Methode zu den schnellsten Faktorisierungsverfahren für dünn besetzte Matrizen.

Dieser Abschnitt ist wie folgt aufgebaut. Zunächst beschäftigen wir uns in 5.1.1 und 5.1.2 mit der symbolischen Faktorisierung. Im Mittelpunkt steht dabei der bereits in Abschnitt 4.1.1 vorgestellte Eliminationsbaum. Schließlich stellen wir in 5.1.3 die Multifrontal-Methode genauer vor. Dabei gehen wir noch einmal auf die oben beschriebenen Techniken zur Steigerung der Cache- und Registereffizienz ein. Um die Leistungsfähigkeit unseres sequentiellen Faktorisierungsalgorithmus unter Beweis zu stellen, vergleichen wir ihn in 5.1.4 mit einem Programm aus der SPOOLES-Bibliothek.

5.1.1 Die symbolische Faktorisierung und der Eliminationsbaum

Ziel der symbolischen Faktorisierung ist die Bestimmung der Nichtnullstruktur des Cholesky-Faktors L von PAP^T. Dazu wird für jede Spalte k von L die Indexmenge $\mathrm{Struct}(L_{*,k})$ berechnet (zur Erinnerung: $\mathrm{Struct}(L_{*,k}) = \{i > k;\ l_{ik} \neq 0\}$). Basierend auf den Indexmengen kann dann eine geeignete Datenstruktur zur Speicherung der von null verschiedenen Subdiagonalelemente von L aufgebaut werden. Für eine genaue Beschreibung der Datenstruktur sei auf Eisenstat et al. [40] verwiesen. In diesem Abschnitt konzentrieren wir uns auf die Berechnung der Mengen $\mathrm{Struct}(L_{*,k})$, $k = 1, \ldots, n$. Dabei spielt der in Abschnitt 4.1.1 vorgestellte Eliminationsbaum eine entscheidende Rolle.

Betrachten wir noch einmal die bei der Berechnung eines Bottom-up-Orderings entstehenden Gebiete. Sei wieder D_v das Gebiet, das durch die Elimination von v entsteht. Es gilt $\mathrm{adj}_G(D_v) = \mathrm{madj}_{G_\pi}(v)$ und damit (vgl. Formel (2.9))

$$w \in \mathrm{adj}_G(D_v) \Leftrightarrow \pi(w) \in \mathrm{Struct}(L_{*,\pi(v)}).$$

Ist also $\mathrm{adj}_G(D_v)$ bekannt, so kann die Nichtnullstruktur der Spalte $\pi(v)$ von L sehr einfach konstruiert werden. Dabei ist jedoch zu beachten, daß zum Zeitpunkt der Elimination von v die Knoten in $\mathrm{adj}_G(D_v)$ noch gar nicht numeriert sind. Daher kann die Berechnung eines Orderings und die Durchführung der symbolischen Faktorisierung nicht in einem Schritt erfolgen.

Hier kommt der Eliminationsbaum ins Spiel. Mit Hilfe dieses Baumes kann die Menge $\mathrm{adj}_G(D_v)$ nach Abschluß des Eliminationsprozesses sehr einfach rekonstruiert werden. Sind $u_1, \ldots, u_t$ die Söhne von v im Eliminationsbaum, so ist D_v durch Verschmelzen der Gebiete D_{u_i}, $i = 1, \ldots, t$, entstanden. Unter der Annahme, daß die Mengen $\mathrm{adj}_G(D_{u_i})$ bekannt sind folgt dann:

$$\mathrm{adj}_G(D_v) = \mathrm{madj}_G(v) \cup \bigcup_{i=1}^{t} (\mathrm{adj}_G(D_{u_i}) - \{v\}). \tag{5.1}$$

Der Rand von D_v setzt sich also aus den noch nicht eliminierten Knoten aus $\mathrm{adj}_G(v)$ und den Rändern der absorbierten Gebiete (ohne v) zusammen. Man beachte, daß wir den Begriff der

```
SYMBFACELIMTREE(T)
01: v := firstPostorder(T);
02: while v ≠ root(T) do
03:     k := π(v);
04:     Struct(L_{*,k}) := Struct(PAP^T_{*,k});
05:     Let u_1, ... , u_t denote the children of v in T.
06:     for each vertex u_i do
07:         Struct(L_{*,k}) := Struct(L_{*,k}) ∪ (Struct(L_{*,π(u_i)}) − {k});
08:     v := nextPostorder(T, v);
09: end while
```

Abb. 5.1: Funktion SYMBFACELIMTREE.

monotonen Adjazenz auf die Knoten des ursprünglichen Graphen G angewandt haben. Basierend auf (5.1) berechnet sich die Indexmenge $\mathrm{Struct}(L_{*,k})$, $k = \pi(v)$, zu

$$\mathrm{Struct}(L_{*,k}) = \mathrm{Struct}(PAP^T_{*,k}) \cup \bigcup_{i=1}^{t} (\mathrm{Struct}(L_{*,\pi(u_i)}) - \{k\}). \tag{5.2}$$

Dabei enthält $\mathrm{Struct}(PAP^T_{*,k})$ die Zeilenindizes der von null verschiedenen Subdiagonalelemente in Spalte k von PAP^T.

Abbildung 5.1 zeigt einen einfachen Algorithmus zur Bestimmung der Nichtnullstruktur von L. Der Aufwand des Algorithmus ist $O(\eta(L))$. Dabei werden die Knoten des Eliminationsbaumes in Postorder-Reihenfolge durchlaufen. Die Postorder-Reihenfolge garantiert, daß die in der for-Schleife (Zeilen 06–07) benötigten Mengen bekannt sind. Die Wurzel $\mathrm{root}(T)$ des Eliminationsbaumes nimmt eine besondere Stellung ein. In der numerischen Praxis tauchen immer wieder Matrizen auf, die einen nicht zusammenhängenden Graphen G induzieren. In diesem Fall ist T ein aus mehreren Bäumen bestehender *Wald*. Der Knoten $\mathrm{root}(T)$ ist ein Hilfsknoten, der die einzelnen Bäume zu einem Baum zusammenfaßt.

Der Eliminationsbaum T besitzt eine Reihe interessanter Eigenschaften. Für einen ausführlichen Überblick sei auf die Zusammenfassung von Liu [96] verwiesen. Für die Herleitung der Multifrontal-Methode in Abschnitt 5.1.3 benötigen wir das folgende Lemma:

Lemma 5.1 *Gilt $l_{i,j} \neq 0$, $j < i$, dann ist in dem Eliminationsbaum T der Knoten $u = \pi^{-1}(j)$ ein Vorgänger des Knotens $w = \pi^{-1}(i)$.*

In dieser Arbeit gehen wir immer davon aus, daß T – quasi als „Nebenprodukt“ – von einem Ordering-Algorithmus konstruiert wurde. Der Eliminationsbaum kann jedoch auch nachträglich berechnet werden. Liu stellt in [96] einen *Union-Find-Algorithmus* vor, der auf Eingabe von G und π den Eliminationsbaum T in Zeit $O(e\,\alpha(e,n))$ konstruiert. Dabei bezeichnet e die Anzahl

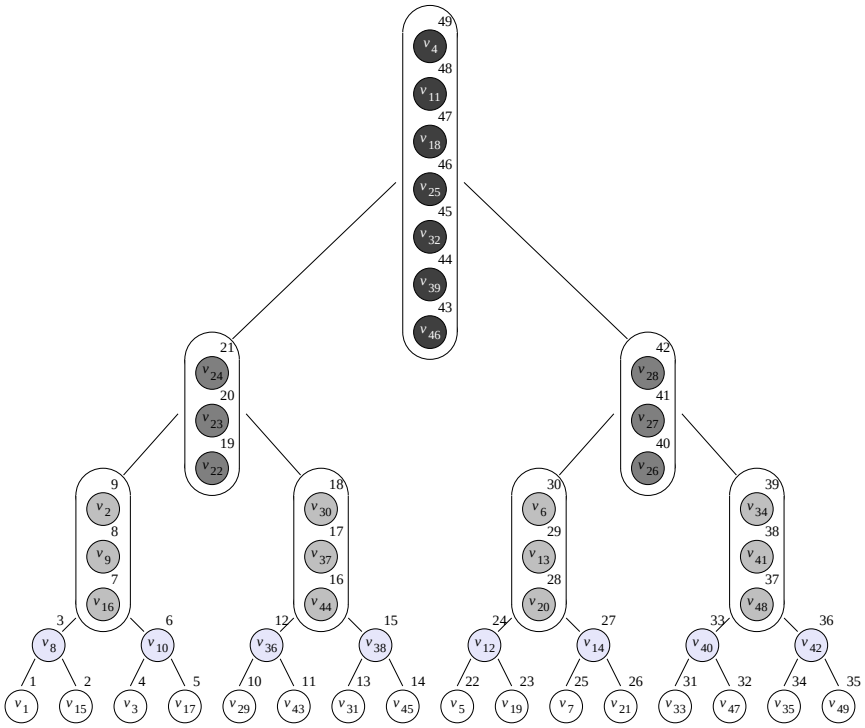


Abb. 5.2: Frontbaum eines 7×7-Gitters mit 9-Punkte-Stern wenn die Knoten in der durch Georges Nested-Dissection-Ordering beschriebenen Reihenfolge eliminiert werden.

der Kanten und n die Anzahl der Knoten in G. Die Funktion α stellt die Inverse der *Ackerman-Funktion* dar. Der Aufwand zur Konstruktion des Eliminationsbaumes wird also von der Anzahl der Kanten in G und nicht von der Anzahl der Kanten im aufgefüllten Graphen G_π bestimmt.

5.1.2 Vom Eliminationsbaum zum Frontbaum

In der Regel bestehen die höheren Ebenen eines Eliminationsbaumes T aus langen Ketten von Knoten. Viele dieser Ketten sind Teil der im Rahmen des Eliminationsprozesses gebildeten Superknoten. In dem zu T gehörenden *Frontbaum* werden solche Ketten durch logische Knoten ersetzt. Jeder logische Knoten repräsentiert eine Menge von aufeinanderfolgend numerierten Graphknoten, die zum Zeitpunkt ihrer Elimination nicht unterscheidbar sind. Ein solcher logischer Knoten heißt *fundamentaler Superknoten* [7] oder kurz *Front*. Im folgenden bezeichnen wir den zu T gehörenden Frontbaum mit dem kaligraphischen Buchstaben $\mathcal{T}$.

Abbildung 5.2 zeigt den Frontbaum, der entsteht wenn die Knoten des in Abbildung 2.4 dargestellten 7×7-Gitters in der durch Georges Nested-Dissection-Ordering beschriebenen Reihenfolge eliminiert werden. Die zu einer Front zusammengefaßten Graphknoten sind durch ein Oval umrandet. In den untersten zwei Ebenen des Baumes besteht jede Front aus nur einem Knoten. Der Frontbaum $\mathcal{T}$ ist hier identisch mit dem Eliminationsbaum T.

Die Fronten bzw. fundamentalen Superknoten stimmen nicht zwangsläufig mit den im Rahmen des Eliminationsprozesses gebildeten Superknoten überein (vgl. Ashcraft und Grimes [7]). Betrachten wir dazu noch einmal das 7×7-Gitter. Werden die Gitterknoten in der durch Georges Nested-Dissection-Ordering beschriebenen Reihenfolge eliminiert, so entsteht zum Schluß des Eliminationsprozesses der Superknoten $\{v_4, v_{11}, v_{18}, v_{25}, v_{32}, v_{39}, v_{46}, v_{26}, v_{27}, v_{28}\}$. In dem Frontbaum aus Abbildung 5.2 ist der Superknoten in zwei Ketten – und damit in zwei fundamentale Superknoten – aufgespaltet, nämlich $\{v_{26}, v_{27}, v_{28}\}$ und $\{v_4, v_{11}, v_{18}, v_{25}, v_{32}, v_{39}, v_{46}\}$.

Der Frontbaum $\mathcal{T}$ kann wie der Eliminationsbaum T während der Berechnung eines Bottom-up-Orderings konstruiert werden. Es ist jedoch auch eine nachträgliche Konstruktion möglich. Dabei ist zu beachten, daß eine Kette in T nicht automatisch einen fundamentalen Superknoten darstellt. Die fundamentalen Superknoten können jedoch mit Hilfe des von Liu et al. [98] vorgeschlagenen Algorithmus nachträglich in T bestimmt werden. Dazu sind lediglich $O(n + e)$ Zeiteinheiten notwendig.

Der Frontbaum $\mathcal{T}$ enthält mehr Informationen als der entsprechende Eliminationsbaum T. Diese zusätzlichen Informationen ermöglichen eine signifikante Beschleunigung der symbolischen Faktorisierung. Betrachten wir dazu die Nichtnullstruktur der zu einer Front F gehörenden Spalten. Seien $v, v' \in F$ mit $\pi(v) = k$ und $\pi(v') = k + 1$. Da die Knoten v, v' zum Zeitpunkt ihrer Elimination nicht unterscheidbar sind, gilt

$$\mathrm{madj}_{G_\pi}(v') = \mathrm{madj}_{G_\pi}(v) - \{v'\}$$

und damit

$$\mathrm{Struct}(L_{*,k+1}) = \mathrm{Struct}(L_{*,k}) - \{k+1\}.$$

Man muß also nur die Nichtnullstruktur der ersten Spalte einer Front berechnen. Die Nichtnullstrukturen der restlichen Spalten ergeben sich durch sukzessives Entfernen des jeweils größten Zeilenindizes. Abbildung 5.3 zeigt den auf einem Frontbaum basierenden Algorithmus zur symbolischen Faktorisierung.

In Zeile 03 wird der erste Knoten der Front F, d. h. der Knoten mit der kleinsten Nummer, ermittelt. Anschließend erfolgt die Initialisierung der Nichtnullstruktur der entsprechenden Spalte. Da die Knoten aus F eine Clique bilden, enthält $\mathrm{Struct}(L_{*,k})$ neben $\mathrm{Struct}(PAP^T_{*,k})$ die Indizes $\{k+1, \ldots, k+s-1\}$. In der for-Schleife (Zeilen 07–10) wird $\mathrm{Struct}(L_{*,k})$ vervollständigt. Sei dazu U_i ein Sohn von F in $\mathcal{T}$. Sei weiter u_i der Knoten mit der größten Nummer in U_i (u_i heißt in diesem Fall letzter Knoten in U_i). Dann wird durch die Elimination des Knotens v das Gebiet D_{u_i} von D_v absorbiert. Daher wird in Zeile 09 die Menge $\mathrm{Struct}(L_{*,\pi(u_i)}) - \{k\}$ der Menge $\mathrm{Struct}(L_{*,k})$ hinzugefügt. Nachdem $\mathrm{Struct}(L_{*,k})$ bekannt ist, können die Indexmengen der restlichen zu F gehörenden Spalten ganz einfach abgeleitet werden (Zeilen 11-12).

Die Fronten spielen auch bei der numerischen Faktorisierung eine entscheidende Rolle. Mit ihrer Hilfe kann der Cholesky-Faktor in quadratische Blöcke partitioniert werden. Jeder Block

```
SymbfacFronttree(T)
01: F := firstPostorder(T);
02: while F ≠ root(T) do
03:     Let s = |F| and let v be the first vertex in F.
04:     k := π(v);
05:     Struct(L_{*,k}) := {k+1, ..., k+s-1} ∪ Struct(PAP^T_{*,k});
06:     Let U_1, ..., U_t denote the children of F in T.
07:     for each front U_i do
08:         Let u_i be the last vertex in U_i.
09:         Struct(L_{*,k}) := Struct(L_{*,k}) ∪ (Struct(L_{*,π(u_i)}) − {k});
10:     end for
11:     for j := k+1 to k+s-1 do
12:         Struct(L_{*,j}) := Struct(L_{*,j-1}) − {j};
13:     F := nextPostorder(T, F);
14: end while
```

Abb. 5.3: Funktion SymbfacFronttree.

wird dann innerhalb der Fan-in- bzw. Fan-out-Methode wie ein Matrixelement behandelt. Durch eine solche blockweise Berechnung von L erhöht sich die Cache-Effizienz des Faktorisierungsalgorithmus erheblich.

Insbesondere in den tieferen Ebenen von $\mathcal{T}$ bestehen die Fronten aus einem oder nur wenigen Knoten (vgl. auch Abbildung 5.2), so daß eine blockweise Berechnung nicht möglich ist. Mit Hilfe des von Ashcraft und Grimes [7] entwickelten Verfahrens können die kleineren Fronten zu einer Front zusammengefaßt werden. Dies geschieht durch kontrolliertes Einfügen von Nullelementen in L. Übertragen auf den Eliminationsprozeß entspricht dies dem Einfügen zusätzlicher Kanten zur Generierung größerer Superknoten. Die neuen Fronten heißen deswegen auch *relaxierte Superknoten*. Da das Verfahren sehr technisch ist, verzichten wir an dieser Stelle auf eine genauere Beschreibung.

5.1.3 Die numerische Faktorisierung nach der Multifrontal-Methode

Die Multifrontal-Methode wurde 1983 von Duff und Reid [35, 36] entwickelt. Die grundlegende Idee der Methode besteht darin, die Faktorisierung einer dünn besetzten Matrix auf die teilweise Faktorisierung mehrerer voll besetzter Matrizen zurückzuführen. Durch diese Vorgehensweise wird die Datenlokalität und damit die Cache-Effizienz des Faktorisierungsalgorithmus signifikant erhöht. Darüber hinaus können Techniken zur Lösung voll besetzter Systeme angewandt werden (vgl. z. B. Duff [32]). Hierzu zählt insbesondere die Technik des Loop-Unrolling, die zu einer besseren Auslastung der Prozessorregister führt.

DENSEFACTOR(A)
01: Initialize L with the lower triangular part of A;
02: **for** $j := 1$ **to** n **do**
03: $l_{j,j} := \sqrt{l_{j,j}}$;

$$\begin{pmatrix} l_{j+1,j} \\ \vdots \\ l_{n,j} \end{pmatrix} := \frac{1}{l_{j,j}} \cdot \begin{pmatrix} l_{j+1,j} \\ \vdots \\ l_{n,j} \end{pmatrix};$$

04: **for** $i := j + 1$ **to** n **do**

$$\begin{pmatrix} l_{i,i} \\ \vdots \\ l_{n,i} \end{pmatrix} := \begin{pmatrix} l_{i,i} \\ \vdots \\ l_{n,i} \end{pmatrix} - l_{i,j} \cdot \begin{pmatrix} l_{i,j} \\ \vdots \\ l_{n,j} \end{pmatrix};$$

05: **end for**
06: **end for**

Abb. 5.4: Funktion DENSEFACTOR.

Zur Beschreibung der Multifrontal-Methode betrachten wir noch einmal die in Abschnitt 2.1 vorgestellte Outer-Product-Variante des Cholesky-Verfahrens. Aus (2.3) läßt sich leicht der in Abbildung 5.4 dargestellte Fan-out-Algorithmus ableiten. Charakteristisch für den Fan-out-Algorithmus ist, daß nach Faktorisierung der Spalte j (Zeile 03) die Einträge der Spalte zur Aktualisierung der nachfolgenden Spalten verwendet werden (Zeile 04). Bei der Multifrontal-Methode werden die Aktualisierungen $-l_{i,j} \cdot (l_{i,j} \cdots l_{n,j})^T$ nicht sofort mit den Spalten $i = j + 1, \ldots, n$ verrechnet, sondern zunächst in einer sogenannten *Update-Matrix* zwischengespeichert. Die Update-Matrix ist auch im Falle einer dünnen Struktur von L immer voll besetzt.

Eng verbunden mit der Multifrontal-Methode ist – wie der Name schon andeutet – der Frontbaum. Zu jeder Front F des Baumes $\mathcal{T}$ gehört eine Update-Matrix und eine sogenannte *Frontal-Matrix*. Auch bei der Frontal-Matrix handelt es sich um eine voll besetzte untere Dreiecksmatrix. Im folgenden bezeichnen wir die zu einer Front F gehörende Frontal-Matrix mit $\mathfrak{F}_F$ und die zu F gehörende Update-Matrix mit $\mathfrak{U}_F$. Die Frontal-Matrix $\mathfrak{F}_F$ setzt sich aus den Update-Matrizen der Söhne $U_1, \ldots, U_t$ von F und aus Einträgen der Matrix PAP^T zusammen. Die Update-Matrix $\mathfrak{U}_F$ enthält alle Aktualisierungen, die aus der Faktorisierung von Spalten resultieren, die zu einer Front im Teilbaum $\mathcal{T}_F$ gehören (dies schließt die Front F ein).

Im folgenden zeigen wir, wie aus der Frontal-Matrix $\mathfrak{F}_F$ die Update-Matrix $\mathfrak{U}_F$ entsteht. Dazu sei angenommen, daß bezüglich der ersten zu F gehörenden Spalte k gilt $\mathrm{Struct}(L_{*,k}) = \{k + 1, \ldots, k+s-1, i_1, \ldots, i_r\}$. Unser Ziel ist die Faktorisierung der Spalten $k, k+1, \ldots, k+s-1$.

Wir konstruieren zunächst die Frontal-Matrix

$$\mathfrak{F}_F = \begin{pmatrix} a_{k,k} & & & & & & & \\ a_{k+1,k} & a_{k+1,k+1} & & & & & & \\ \vdots & \vdots & \ddots & & & & & \\ a_{k+s-1,k} & a_{k+s-1,k+1} & \cdots & a_{k+s-1,k+s-1} & & & & \\ a_{i_1,k} & a_{i_1,k+1} & \cdots & a_{i_1,k+s-1} & 0 & & & \\ \vdots & \vdots & & \vdots & \vdots & \ddots & & \\ a_{i_r,k} & a_{i_r,k+1} & \cdots & a_{i_r,k+s-1} & 0 & \cdots & 0 \end{pmatrix}.$$

Die ersten s Spalten entsprechen dabei den Spalten $k, k+1, \ldots, k+s-1$ von PAP^T. Um die Spalten faktorisieren zu können, ist die Einbeziehung aller Aktualisierungen notwendig, die aus der Faktorisierung vorangegangener Spalten resultieren. Wie Abbildung 5.4 zeigt, trägt eine vorangegangene Spalte j nur dann zur Aktualisierung einer Spalte $i > j$ bei, wenn $l_{ij} \neq 0$ gilt. Nach Lemma 5.1 ist in diesem Fall der zu j gehörende Knoten ein Vorgänger des zu i gehörenden Knotens im Eliminationsbaum T. Daher müssen in $\mathfrak{F}_F$ nur solche Aktualisierungen einbezogen werden, die aus der Faktorisierung von Spalten resultieren, die zu einer Front in einem Teilbaum $\mathcal{T}_{U_i}$, $i = 1, \ldots, t$ gehören. Nach Voraussetzung werden diese Aktualisierungen in den Update-Matrizen $\mathfrak{U}_{U_i}$ gespeichert. Deshalb können wir nach Addition der Update-Matrizen $\mathfrak{U}_{U_1}, \ldots, \mathfrak{U}_{U_t}$ die ersten s Spalten von $\mathfrak{F}_F$ faktorisieren. Wir benuzten dazu einen Algorithmus ähnlich dem in Abbildung 5.4. Sei

$$\mathfrak{F}_F = \begin{pmatrix} l_{k,k} & & & & & & & \\ l_{k+1,k} & l_{k+1,k+1} & & & & & & \\ \vdots & \vdots & \ddots & & & & & \\ l_{k+s-1,k} & l_{k+s-1,k+1} & \cdots & l_{k+s-1,k+s-1} & & & & \\ l_{i_1,k} & l_{i_1,k+1} & \cdots & l_{i_1,k+s-1} & u_{i_1,i_1} & & & \\ \vdots & \vdots & & \vdots & \vdots & \ddots & & \\ l_{i_r,k} & l_{i_r,k+1} & \cdots & l_{i_r,k+s-1} & u_{i_r,i_1} & \cdots & u_{i_r,i_r} \end{pmatrix}$$

die resultierende Frontal-Matrix. Die ersten s Spalten beinhalten jetzt die von null verschiedenen Einträge der Spalten $k, k+1, \ldots, k+s-1$ des Cholesky-Faktors L. Die verbleibende Dreiecksmatrix enthält alle Aktualisierungen, die aus der Faktorisierung von Spalten resultieren, die zu einer Front in einem Teilbaum $\mathcal{T}_{U_i}$ oder zur Front F gehören. Die Dreiecksmatrix stellt daher die Update-Matrix $\mathfrak{U}_F$ dar.

Bei der Addition einer Update-Matrix $\mathfrak{U}_{U_i}$ ist zu berücksichtigen, daß die Elemente aus $\mathfrak{U}_{U_i}$ bezüglich L die Indizes (r, c) mit $r \geq c$ und $r, c \in \mathrm{Struct}(L_{*,\pi(u_i)})$ besitzen (u_i ist wieder der letzte Knoten in U_i) und die Elemente aus $\mathfrak{F}_F$ die Indizes (r, c) mit $r \geq c$ und $r, c \in \mathrm{Struct}(L_{*,k}) \cup \{k\}$. Es dürfen nur Elemente mit „passenden“ Indizes addiert werden.

Gilt beispielsweise

$$\mathfrak{U}_{U_i} = \begin{pmatrix} u_{3,3} & \\ u_{5,3} & u_{7,7} \end{pmatrix} \qquad \text{und} \qquad \mathfrak{F}_F = \begin{pmatrix} f_{3,3} & & \\ f_{4,3} & f_{4,4} & \\ f_{5,3} & f_{5,4} & f_{7,7} \end{pmatrix},$$

so ergibt sich aus der Addition beider Matrizen die Matrix

$$\begin{pmatrix} f_{3,3} + u_{3,3} & & \\ f_{4,3} & f_{4,4} & \\ f_{5,3} + u_{5,3} & f_{5,4} & f_{7,7} + u_{7,7} \end{pmatrix}.$$

Diese *erweiterte Addition (extended add)* wird in der Literatur mit dem Symbol $\oplus$ bezeichnet. Aus der symbolischen Faktorisierung folgt sofort $\mathrm{Struct}(L_{*,\pi(u_i)}) \subset \mathrm{Struct}(L_{*,k}) \cup \{k\}$ (vgl. Abbildung 5.3). Daher ist jedes Indexpaar (r, c) aus $\mathfrak{U}_{U_i}$ auch in $\mathfrak{F}_F$ enthalten. Die erweiterte Addition ist also wohldefiniert.

Abbildung 5.5 zeigt den vollständigen Multifrontal-Algorithmus. Für eine detailierte Herleitung sei auf das Tutorial von Liu [97] verwiesen. Der Frontbaum $\mathcal{T}$ wird wieder in Postorder-Reihenfolge durchlaufen. Hierdurch ist garantiert, daß die zur Bildung von $\mathfrak{F}_F$ benötigten Update-Matrizen $\mathfrak{U}_{U_i}$ bekannt sind.

Wir wollen die Vorgehensweise an einem Beispiel veranschaulichen. Dazu bestrachten wir noch einmal den in Abbildung 5.2 dargestellten Frontbaum. Da der Baum in Postorder-Reihenfolge durchlaufen wird, werden zuerst die Fronten $\{v_1\}$ und $\{v_{15}\}$ bearbeitet. Die erste (und einzige) zu $\{v_1\}$ bzw. $\{v_{15}\}$ gehörende Spalte ist die Spalte eins bzw. die Spalte zwei. Es gilt $\mathrm{Struct}(L_{*,1}) = \{3, 8, 9\}$ und $\mathrm{Struct}(L_{*,2}) = \{3, 7, 8, 19, 20\}$. Beide Fronten sind Blätter des Baumes. Die Frontal-Matrizen $\mathfrak{F}_{\{v_1\}}$ und $\mathfrak{F}_{\{v_{15}\}}$ können daher sofort faktorisiert werden. Man erhält die Spalten eins und zwei des Cholesky-Faktors sowie die Update-Matrizen

$$\mathfrak{U}_{\{v_1\}} = \begin{pmatrix} u_{3,3} & & \\ u_{8,3} & u_{8,8} & \\ u_{9,3} & u_{9,8} & u_{9,9} \end{pmatrix} \qquad \text{und} \qquad \mathfrak{U}_{\{v_{15}\}} = \begin{pmatrix} u'_{3,3} & & & & \\ u'_{7,3} & u'_{7,7} & & & \\ u'_{8,3} & u'_{8,7} & u'_{8,8} & & \\ u'_{19,3} & u'_{19,7} & u'_{19,8} & u'_{19,19} & \\ u'_{20,3} & u'_{20,7} & u'_{20,8} & u'_{20,19} & u'_{20,20} \end{pmatrix}.$$

Als nächstes bearbeitet der Algorithmus die Front $\{v_8\}$. Zu dieser Front gehört die Spalte drei. Es gilt $\mathrm{Struct}(L_{*,3}) = \{7, 8, 9, 19, 20\}$. Nach Initialisierung der Frontal-Matrix $\mathfrak{F}_{\{v_8\}}$ werden die Update-Matrizen $\mathfrak{U}_{\{v_1\}}$ und $\mathfrak{U}_{\{v_{15}\}}$ aufaddiert. Dann gilt:

$$\mathfrak{F}_{\{v_8\}} = \begin{pmatrix} a_{3,3} + u_{3,3} + u'_{3,3} & & & & & \\ a_{7,3} + u'_{7,3} & u'_{7,7} & & & & \\ a_{8,3} + u_{8,3} + u'_{8,3} & u'_{8,7} & u_{8,8} + u'_{8,8} & & & \\ a_{9,3} + u_{9,3} & 0 & u_{9,8} & u_{9,9} & & \\ a_{19,3} + u'_{19,3} & u'_{19,7} & u'_{19,8} & 0 & u'_{19,19} & \\ a_{20,3} + u'_{20,3} & u'_{20,7} & u'_{20,8} & 0 & u'_{20,19} & u'_{20,20} \end{pmatrix}.$$

MULTIFRONAL($\mathcal{T}$)
01: $F := \text{firstPostorder}(\mathcal{T})$;
02: **while** $F \neq \text{root}(\mathcal{T})$ **do**
03: Let $s = |F|$ and let v be the first vertex in F.
04: $k := \pi(v)$;
05: Let $k+1, \ldots, k+s-1, i_1, \ldots, i_r$ be the subscripts in $\text{Struct}(L_{*,k})$.
06: Set up frontal matrix

$$\mathfrak{F}_F = \begin{pmatrix} a_{k,k} & & & & & & \\ a_{k+1,k} & a_{k+1,k+1} & & & & & \\ \vdots & \vdots & \ddots & & & & \\ a_{k+s-1,k} & a_{k+s-1,k+1} & \cdots & a_{k+s-1,k+s-1} & & & \\ a_{i_1,k} & a_{i_1,k+1} & \cdots & a_{i_1,k+s-1} & 0 & & \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \\ a_{i_r,k} & a_{i_r,k+1} & \cdots & a_{i_r,k+s-1} & 0 & \cdots & 0 \end{pmatrix}$$

where the first s columns correspond to columns $k, \ldots, k+s-1$ of PAP^T.
07: Let $U_1, \ldots, U_t$ denote the children of F in $\mathcal{T}$.
08: **for** each front U_i **do**
09: $\mathfrak{F}_F := \mathfrak{F}_F \oplus \mathfrak{U}_{U_i}$.
10: Perform s steps of elimination on $\mathfrak{F}_F$ to obtain the columns $k, k+1, \ldots, k+s-1$ of L and the update matrix $\mathfrak{U}_F$.
11: $F := \text{nextPostorder}(\mathcal{T}, F)$;
12: **end while**

Abb. 5.5: Funktion MULTIFRONTAL.

Die Frontal-Matrix kann jetzt faktorisiert werden. Man erhält so die dritte Spalte von L und die Update-Matrix $\mathfrak{U}_{\{v_8\}}$. Anschließend fährt der Algorithmus mit der Front $\{v_3\}$ fort.

Um die Vorteile der Multifrontal-Methode voll ausnutzen zu können, ist eine sorgfälltige Implementierung notwendig. Dabei muß insbesondere auf die folgenden Punkte geachtet werden:

Verwaltung der Update-Matrizen Da der Algorithmus den Frontbaum in Postorder-Reihenfolge durchläuft, können die bereits generierten, aber noch nicht mit einer Frontal-Matrix verrechneten Update-Matrizen in einem *Stack* gespeichert werden. Der Stack wächst dynamisch mit der Größe der abgelegten Update-Matrizen. Der hierfür bereitzustellende Speicherplatz stellt einen zusätzlichen Overhead dar. Mit Hilfe des von Liu [92] vorgestellten Algorithmus kann die maximale Ausdehnung des Stacks minimiert werden. Dazu werden die Teilbäume $\mathcal{T}_{U_1}, \ldots, \mathcal{T}_{U_t}$ unter einer Front F in aufsteigender Reihenfolge nach der Größe ihrer Updaten-Matrizen $\mathfrak{U}_{U_1}, \ldots, \mathfrak{U}_{U_t}$ angeordnet. In dem Postorder-Durchlauf werden dann diejenigen Bäume zuerst abgearbeitet, die eine kleine Update-Matrix liefern.

Erweiterte Addition Die Addition der Update-Matrizen $\mathfrak{U}_{U_1}, \ldots, \mathfrak{U}_{U_t}$ zu $\mathfrak{F}_F$ stellt einen numerischen Overhead dar. Es entstehen zusätzliche Operationen, die bei Verwendung eines Fan-in- oder eines Fan-out-Verfahrens vermeidbar wären. Zur Beschleunigung der erweiterten Addition benutzt man *lokale Indizes* (vgl. auch Schreiber [127]). Dabei erhalten die Elemente einer Update-Matrix vor der eigentlichen Addition die passenden Indizes relativ zur Frontal-Matrix. Beipielsweise erhält das Element $u_{9,3}$ aus $\mathfrak{U}_{\{v_1\}}$ die Indizes $(4, 1)$.

Faktorisierung der Frontal-Matrix In vielen Fällen paßt die gesamte Frontal-Matrix in den Prozessor-Cache, so daß die Faktorisierung der ersten s Spalten sehr effizient durchgeführt werden kann. Hierin liegt der eigentliche Vorteil der Multifrontal-Methode. Die Effizienz läßt sich durch Anwendung der Loop-Unrolling-Technik noch einmal signifikant steigern. Dazu wird $\mathfrak{F}_F$ in quadratische Blöcke partitioniert. Die Blöcke sind so klein gewählt, daß die Operationen auf ihnen ausprogrammiert werden können. Abbildung 5.6 zeigt die Blockversion des Algorithmus DENSEFACTOR aus Abbildung 5.4. Die Variablen I, J laufen dabei über Zeilen und Spalten bestehend aus einzelnen Blöcken. Besitzen die Blöcke die Größe $q \times q$, so gilt $N = n/q$ (wir nehmen der Einfachheit halber an, daß n ohne Rest durch q teilbar ist). Die Funktion FACTOR (Zeile 03) berechnet dann den Cholesky-Faktor einer $q \times q$-Matrix. Wir weisen nochmals darauf hin, daß die Funktion FACTOR sowie alle anderen Operationen auf den $q \times q$ Blöcken ohne zusätzliche Schleifen auskommen.

Die Loop-Unrolling-Technik entfaltet ihre volle Wirkung nur dann, wenn die Front F eine gewisse Anzahl von Knoten enthält ($s > 1$). Mit Hilfe des von Ashcraft und Grimes [7] vorgestellten Verfahrens können wieder kleinere Fronten zu relaxierten Superknoten zusammengefaßt werden. Hierdurch entstehen zwar zusätzliche Nullelemente in den Frontal- und Update-Matrizen, es erhöht sich jedoch auch die Cache- und Registereffizienz des Faktorisierungsalgorithmus.

5.1.4 Experimentelle Ergebnisse

Wir haben ein sequentielles Programm zur Faktorisierung dünn besetzter, positiv definiter Matrizen entwickelt, das als Ausgangspunkt der im nächsten Abschnitt beschriebenen Parallelisierung dient. Das Programm trägt den Namen **space** (**SPA**rse **C**holesky **E**limination) und stellt eine um die symbolische und numerische Faktorisierung erweiterte Version des Programms **multipord** dar. Die symbolische Faktorisierung orientiert sich an der Funktion SYMBFACFRONTTREE, die numerische an der Funktion MULTIFRONTAL. Die folgende Liste faßt die wichtigsten Merkmale unseres Faktorisierungsalgorithmus zusammen.

- Vor Beginn der symbolischen und numerischen Faktorisierung wird der im Rahmen des Ordering-Prozesses konstruierte Frontbaum $\mathcal{T}$ wie folgt modifiziert:
  - In einem Postorder-Durchlauf werden die Fronten an den Blättern von $\mathcal{T}$ mit den darüberliegenden Fronten zu neuen Blatt-Fronten verschmolzen. Der Prozeß stoppt,

DENSEBLOCKFACTOR(A)
01: Initialize L with the lower triangular part of A;
02: **for** $J := 1$ **to** N **do**
03: $L_{J,J} :=$ FACTOR$(L_{J,J})$;

$$\begin{pmatrix} L_{J+1,J} \\ \vdots \\ L_{N,J} \end{pmatrix} := L_{J,J}^{-1} \cdot \begin{pmatrix} L_{J+1,J} \\ \vdots \\ L_{N,J} \end{pmatrix};$$

04: **for** $I := J + 1$ **to** N **do**

$$\begin{pmatrix} L_{I,I} \\ \vdots \\ L_{N,I} \end{pmatrix} := \begin{pmatrix} L_{I,I} \\ \vdots \\ L_{N,I} \end{pmatrix} - L_{I,J}^{T} \cdot \begin{pmatrix} L_{I,J} \\ \vdots \\ L_{N,J} \end{pmatrix};$$

05: **end for**
06: **end for**

Abb. 5.6: Funktion DENSEBLOCKFACTOR.

sobald eine neue Blatt-Front mehr als 200 zusätzliche Nulleinträge enthält.

– In einem zweiten Postorder-Durchlauf werden die Teilbäume $\mathcal{T}_{U_1}, \ldots, \mathcal{T}_{U_t}$ unter einer Front F aufsteigend nach der Größe ihrer Update-Matrizen angeordnet.

- Zur Faktorisierung der Frontal-Matrizen verwenden wir einen numerischen Kern, der auf Blöcken der Größe 3×3 arbeitet. Größere Kerne führen nur noch zu marginalen Effizienzsteigerungen. Sie erhöhen den Programmieraufwand jedoch erheblich.

Um die Leistungsfähigkeit unseres sequentiellen Faktorisierungsalgorithmus unter Beweis zu stellen, haben wir **space** mit einem Programm aus der SPOOLES-Bibliothek verglichen. Das Programm basiert auf der Fan-in-Methode und benutzt wie **space** einen numerischen Kern der Größe 3×3. Eingabe des Programms ist – abgesehen von der Matrix A – der von **space** konstruierte Frontbaum. Damit sind die Startbedingungen für beide Programme gleich.

Tabelle 5.1 zeigt die Laufzeiten der symbolischen und numerischen Faktorisierung für 15 Matrizen unserer Benchmark-Suite. Wir haben diejenigen Matrizen ausgewählt, deren Faktorisierung die meisten Operationen verursacht. Alle Zeitangaben wurden auf einer SUN Ultra mit 296 MHz UltraSPARC-II Prozessor und zwei GByte Hauptspeicher ermittelt. In den Spalten 3 und 5 sind zusätzlich die jeweils erzielten Megaflops angegeben. Ihre Berechnung basiert auf der Zahl der zur Faktorisierung benötigten Multiplikations- und Additionsoperationen. Diese Zahl ist durch das verwendete Ordering eindeutig bestimmt.* Die von der Multifrontal-Methode zusätzlich durchgeführten Additionsoperationen (extended add) bleiben also unberücksichtigt.

*Die Zahlen sind in der letzten Spalte von Tabelle 4.4 angegeben.

	SPOOLES-2.2		**space**	
Matrix	symb. Fakt.	num. Fakt.	symb. Fakt.	num. Fakt.
BCSSTK30	2.34	8.50 (84.57)	1.31	6.64 (106.95)
BCSSTK31	1.35	13.95 (89.09)	1.01	10.83 (113.61)
BCSSTk32	2.38	9.75 (82.61)	1.43	7.61 (104.01)
MAT02HBF	2.66	12.92 (86.18)	1.56	10.19 (107.02)
MAT03HBF	4.36	29.04 (89.05)	2.58	22.74 (111.85)
BRACK2	1.11	19.40 (86.70)	1.12	15.60 (106.28)
3DTUBE	3.84	159.44 (77.41)	3.40	131.09 (94.08)
CFD1	2.42	96.56 (92.05)	2.86	87.09 (101.66)
CFD2	4.19	333.78 (78.64)	5.37	289.59 (90.49)
CYL3	2.65	616.94 (64.85)	5.14	494.06 (81.20)
GEARBOX	11.52	207.41 (84.60)	7.97	181.54 (96.35)
NASASRB	3.07	29.32 (90.37)	2.27	23.30 (113.09)
WAVE	3.59	1638.23 (59.66)	8.04	1179.09 (82.80)
PWTK	15.09	294.40 (77.33)	10.14	279.81 (81.10)
HERMES	11.57	4158.60 (60.41)	19.64	3273.68 (76.68)

Tab. 5.1: Vergleich der Laufzeiten zur Durchführung der symbolischen und numerischen Faktorisierung (in Sek.). In Klammern sind zusätzlich die erzielten Megaflops angegeben.

Hierdurch ist sichergestellt, daß sich im Falle von **space** der numerische Overhead nicht positiv auf die erzielten Megaflops auswirkt.

Am Beispiel der Matrix CFD1 wollen wir den Einfluß des numerischen Kerns auf die Laufzeit des Faktorisierungsalgorithmus verdeutlichen: Bei Verwendung eines einfachen 1×1-Kerns werden zur Faktorisierung von CFD1 ungefähr 232 Sekunden benötigt. Die Laufzeit reduziert sich auf 109 Sekunden im Falle eines 2×2-Kerns. Bei Verwendung eines 3×3-Kerns ergibt sich eine weitere – diesmal jedoch nicht mehr so starke – Beschleunigung auf 87 Sekunden.

5.2 Der parallele Fall

Interessanterweise besitzt die Faktorisierung dünn besetzter Matrizen ein höheres Parallelisierungspotential als die voll besetzter. Trotzdem gab es bis Anfang der neunziger Jahre keinen parallelen Algorithmus, der die von großen, verteilten Systemen bereitgestellte Rechenleistung effektiv nutzen konnte. Schon im sequentiellen Fall ist das Design eines Faktorisierungalgorithmus sehr viel schwieriger, wenn die Matrizen eine dünne Struktur besitzen. Da zur Parallelisierung dieser komplexen Algorithmen relativ einfache Techniken angewandt wurden, entstand ein gewaltiger Kommunikations-Overhead, der eine schlechte Skalierbarkeit der parallelen Verfahren zur Folge hatte (vgl. Schreiber [128]). Die Skalierbarkeit eines Algorithmus beschreibt

die Fähigkeit, eine vorgegebene Effizienz bei gleichzeitiger Erhöhung der Prozessorzahl und der Problemgröße zu halten. Die *Isoeffizienzfunktion* gibt dabei an, wie stark die Problemgröße in Abhängigkeit von der Prozessorzahl erhöht werden muß (vgl. Kumar et al. [84]).

Das höhere Parallelisierungspotential bei der Zerlegung dünn besetzter Matrizen resultiert aus dem Umstand, daß die Spalten des Cholesky-Faktors nicht zwangsläufig nacheinander berechnet werden müssen. Vielmehr erlaubt die durch den Eliminationsbaum beschriebene partielle Ordnung, Spalten in unterschiedlichen Teilbäumen gleichzeitig zu faktorisieren. In den ersten parallelen Algorithmen wurden die Knoten des Eliminationsbaumes mit Hilfe eines sehr einfachen *Wrap-Mapping-Verfahrens* [50] auf die Prozessoren abgebildet. Das Wrap-Mapping-Verfahren arbeitet wie folgt: Zunächst werden die Blatt-Knoten in zyklischer Form auf die Prozessoren verteilt. Die Knoten werden dann aus dem Eliminationsbaum entfernt, und das Verfahren fährt mit den neuen Blatt-Knoten fort. Auf diese Weise werden die Knoten des Eliminationsbaumes ebenenweise von den Blättern bis zur Wurzel auf die Prozessoren verteilt. Das Wrap-Mapping-Verfahren hat zwei Vorteile: Zum einen nutzt es das durch den Eliminationsbaum beschriebene Parallelisierungspotential voll aus, zum anderen garantiert es eine gleichmäßige Verteilung der Rechenlast.

Trotzdem ist diese Verteilung der Spalten nicht praktikabel wie das folgende Beispiel zeigt. Dazu sei angenommen, daß die Nichtnullstruktur der Matrix A ein $n \times n$-Gitter induziert. Dann sind zur Berechnung des Cholesky-Faktors $O(n^3)$ Multiplikations- und Additionsoperationen notwendig. Der Cholesky-Faktor enthält dabei $O(n^2 \log n)$ von null verschiedene Elemente (vgl. Kapitel 3). Ein paralleler Fan-in- [6] oder Fan-out-Algorithmus [50], der zur Verteilung der Spalten das Wrap-Mapping-Verfahren benutzt, produziert auf p Prozessoren ein Kommunikationsvolumen von $O(n^2 p \log n)$ [55]. Man beachte, daß das Kommunikationsvolumen, also die Summe aller verschickten Daten, lediglich eine untere Schranke für den Kommunikations-Overhead darstellt. Damit steht der Kommunikations-Overhead in keinem akzeptablen Verhältnis zur Anzahl der durchzuführenden arithmetischen Operationen.

Historisch gesehen wurden diese ersten parallelen Algorithmen in zwei Richtungen weiterentwickelt. Zum einen konnte das Kommunikationsvolumen durch geschicktere Mapping-Strategien reduziert werden. In dem von Ashcraft et al. [5] sowie Geist und Ng [46] vorgeschlagenen *Domain-Mapping-Verfahren* werden ganze Teilbäume exklusiv einem Prozessor zugeordnet. Die restlichen Knoten in den obersten Ebenen des Eliminationsbaumes werden wie beim Wrap-Mapping-Verfahren ebenenweise auf die Prozessoren verteilt. Ein Fan-out-Algorithmus, der dieses Mapping-Verfahren benutzt, produziert ein Kommunikationsvolumen von $O(n^2 p \log p)$ (vgl. Hulbert und Zmijewski [76]). Bei dem von George et al. [55] vorgeschlagenen *Subtree-to-Subcube-Mapping* werden auch die restlichen Knoten nicht mehr zyklisch auf alle Prozessoren verteilt, sondern auf einzelne Prozessorgruppen (vgl. auch Abschnitt 5.2.1). Mit Hilfe des Subtree-to-Subcube-Mappings verringert sich das Kommunikationsvolumen eines Fan-in- oder Fan-out-Algorithmus auf $O(n^2 p)$ [55]. Auch bei dem von Pothen und Sun [114] vorgeschlagenen *Pro-*

portinal-Mapping-Verfahren werden ganze Teilbäume rekursiv auf Prozessorgruppen abgebildet. Die Anzahl der Prozessoren in einer Gruppe richtet sich dabei nach der Größe des Teilbaumes.

Nach George et al. [55] ist $O(n^2p)$ eine untere Schranke für das Kommunikationsvolumen eines jeden parallelen Faktorisierungsalgorithmus, der auf einer spaltenweisen Verteilung von A beruht. In einem zweiten Ansatz wurde daher die dünn besetzte Matrix A nicht nur spalten-, sondern auch zeilenweise auf die Prozessoren verteilt. Aus der Literatur sind mehrere Algorithmen bekannt, die auf einer solchen zweidimensionalen Mapping-Strategie basieren. Beispielhaft seien hier die Arbeiten von Gilbert und Schreiber [60], Rothberg [124], Rothberg und Gupta [126] sowie Venugopal und Naik [140] genannt. Der beste Algorithmus reduziert das Kommunikationsvolumen auf $O(n^2\sqrt{p}\log p)$ [126]. In [2] stellt Ashcraft einen *Fan-both-Algorithmus* vor (ein Zwitter aus Fan-in- und Fan-out-Verfahren), der ein Kommunikationsvolumen von $O(n^2\sqrt{p}\log n)$ erzeugt.

In den Arbeiten von Gupta et al. [63, 66] werden beide Ansätze miteinander kombiniert. Ihr paralleler Algorithmus beruht auf der Multifrontal-Methode. Die Knoten des Eliminationsbaumes werden mit Hilfe eines Subcube-to-Subtree-Mappings auf die Prozessoren verteilt. Aus dem Mapping wird dann eine zeilen- und spaltenweise Verteilung der Matrixelemente abgeleitet. Hierdurch reduziert sich nicht nur das Kommunikationsvolumen, sondern auch der Kommunikations-Overhead auf $O(n^2\sqrt{p})$ [63, 66].

Dieser Abschnitt ist wie folgt aufgebaut. Zunächst stellen wir in 5.2.1 unseren Mapping-Algorithmus vor. Der Algorithmus basiert auf dem Subtree-to-Subcube-Mapping von George et al. [55]. Anschließend beschreiben wir in 5.2.2 und 5.2.3 Algorithmen zur parallelen symbolischen und parallelen numerischen Faktorisierung. Die parallele numerische Faktorisierung basiert auf dem von Gupta et al. [63, 66] entwickelten zweidimensionalen Multifrontal-Algorithmus. Schließlich präsentieren wir in 5.2.4 einige experimentelle Ergebnisse. Im Mittelpunkt steht dabei der Einfluß des Orderings auf die parallele numerische Faktorisierung.

5.2.1 Mapping des Frontbaumes

Ursprünglich wurde das Subtree-to-Subcube-Mapping entwickelt, um die Knoten eines $n \times n$-Gitters auf die Prozessoren eines *r-dimensionalen Hypercubes* zu verteilen. Der r-dimensionale Hypercube ist ein Graph bestehend aus 2^r Knoten und $r2^{r-1}$ Kanten. Jeder Knoten des Graphen entspricht einem Prozessor und wird durch eine binäre Zeichenkette der Länge r dargestellt. Zwei Knoten sind genau dann durch eine Kante verbunden, wenn sich ihre binären Zeichenketten in genau einem Bit unterscheiden. Eine Kante heißt *Kante der Dimension* d, $0 \leq d < r$, falls sie zwei Knoten $u, v \in \{0,1\}^r$ verbindet, die sich im d-ten Bit unterscheiden.

Werden die Knoten des Gitters in der durch Georges Nested-Dissection-Ordering beschriebenen Reihenfolge eliminiert, so ergibt sich ein balancierter Frontbaum. Aufgrund der rekursiven Struktur des Hypercubes (siehe [87], Seite 308 ff) können die Teilbäume des Frontbaumes sehr

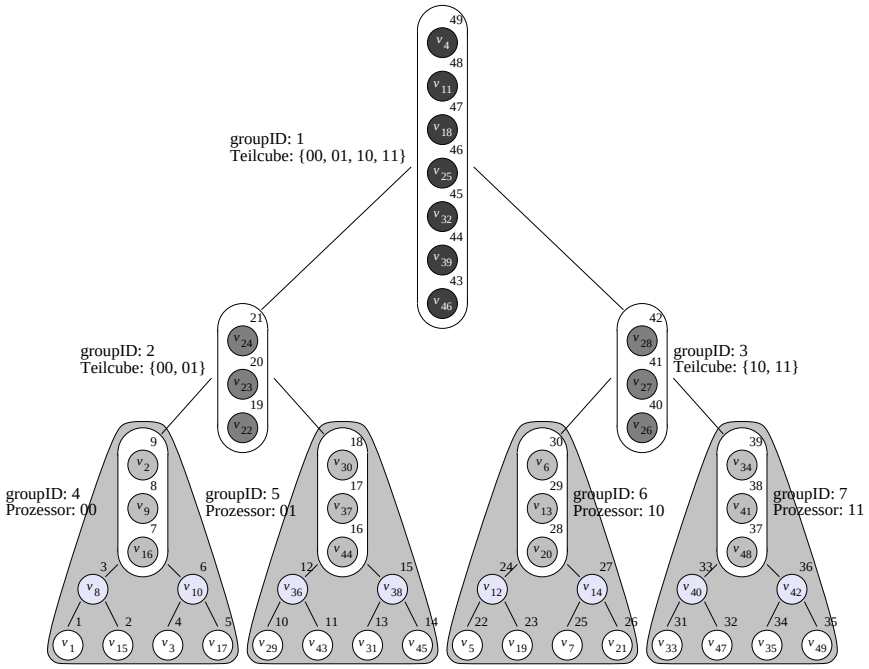


Abb. 5.7: Subtree-to-Subcube-Mapping des Frontbaumes aus Abbildung 5.2.

einfach einzelnen Teilcubes zugeordnet werden. Abbildung 5.7 illustriert das Subtree-to-Subcube-Mapping am Beispiel des 7×7-Gitters und des zweidimensionalen Hypercubes. Der Teilbaum $\mathcal{T}_{\{v_{22},v_{23},v_{24}\}}$ ist dem Teilcube $\{00, 01\}$ zugeordnet. Der Baum teilt sich auf in $\mathcal{T}_{\{v_2,v_9,v_{16}\}}$ und $\mathcal{T}_{\{v_{30},v_{37},v_{44}\}}$. Die Bäume werden auf die Teilcubes $\{00\}$ bzw. $\{01\}$ abgebildet. Alle Fronten in den grau unterlegten Teilbäumen sind so exklusiv einem Prozessor zugeordnet. Die Fronten $\{v_{22}, v_{23}, v_{24}\}$ und $\{v_{26}, v_{27}, v_{28}\}$ werden von den zwei Prozessoren 00 und 01 bzw. 10 und 11 bearbeitet und die Wurzel-Front von allen vier Prozessoren gemeinsam.

Im allgemeinen Fall ist der Frontbaum $\mathcal{T}$ weder binär noch balanciert. Um die Prozessoren eines r-dimensionalen Hypercubes gleichmäßig auszulasten, müssen daher kompliziertere Algorithmen wie z. B. das Domain-Mapping- [5, 46] oder das Proportional-Mapping-Verfahren [114] angewandt werden. In unserem parallelen Algorithmus benutzen wir eine einfache Erweiterung des Subtree-to-Subcube-Mappings (vgl. auch Gupta et al. [66]). Die Idee besteht darin, den Frontbaum $\mathcal{T}$ von „oben nach unten“ zu durchsuchen bis $\mathcal{T}$ in zwei Gruppen von Teilbäumen aufgeteilt werden kann, deren Faktorisierung in etwa den gleichen Aufwand erfordert. Hat man eine solche Aufteilung gefunden, so müssen die zwei Teilbaumgruppen einem $(r-1)$-dimensionalen Hypercube zugeordnet werden. Man erhält so einen rekursiven Mapping-Algorithmus wie er in Abbildung 5.8 beschrieben ist.

Eingabe des Algorithmus ist eine Menge $\mathcal{M}$ von Teilbäumen, die einem Hypercube der Dimension d zugeordnet werden soll. Genaugenommen besteht das Ziel des Mapping-Algorithmus nicht in einer Verteilung der Teilbäume, sondern in einer Verteilung der in den Teilbäumen ent-

```
SPLIT(M, d, groupID)
01:  if d > 0 then
02:      ratio := 0;
03:      while ratio < β do
04:          M' := M;
05:          M_1 := ∅; ops(M_1) := 0;
06:          M_2 := ∅; ops(M_2) := 0;
07:          while M' ≠ ∅ do
08:              Find T_F ∈ M' with ops(T_F) maximal. Set M' := M' − {T_F};
09:              if ops(M_1) < ops(M_2) then
10:                  M_1 := M_1 ∪ {T}; ops(M_1) := ops(M_1) + ops(T_F);
11:              else
12:                  M_2 := M_2 ∪ {T}; ops(M_2) := ops(M_2) + ops(T_F);
13:              end if
14:          end while
15:          ratio := min(ops(M_1), ops(M_2)) / max(ops(M_1), ops(M_2));
16:          if ratio < β then
17:              Find T_F ∈ M with ops(T_F) maximal. Set M := M − {T_F};
18:              frontgroup(F) := groupID;
19:              Let U_1, ..., U_t denote the children of F.
20:              M := M ∪ {T_U1, ..., T_Ut};
21:          end if
22:      end while
23:      SPLIT(M_1, d − 1, 2 · groupID);
24:      SPLIT(M_2, d − 1, 2 · groupID +1);
25:  else
26:      for each front tree T_F in M do
27:          for each front U in T_F do
28:              frontgroup(U) := groupID;
29:  end if
```

Abb. 5.8: Funktion SPLIT.

haltenen Fronten auf die Prozessoren. Dazu erhalten die Fronten eine sogenannte *Gruppenidentifikationsnummer*, kurz $\mathrm{groupID}$. Abbildung 5.7 zeigt die Verteilung der Identifikationsnummern am Beispiel des 7×7-Gitters und des zweidimensionalen Hypercubes. Initial gilt $\mathcal{M} = \{\mathcal{T}\}$, $d = r$ und $\mathrm{groupID} = 1$.

Der Algorithmus arbeitet wie folgt: Gilt $d > 0$, so wird mittels einer einfachen *Bin-Packing-Heuristik* versucht, die in $\mathcal{M}$ enthaltenen Teilbäume in zwei möglichst gleich schwere Mengen $\mathcal{M}_1, \mathcal{M}_2$ aufzuteilen (Zeilen 04–14). Um das Gewicht der Mengen $\mathcal{M}_1, \mathcal{M}_2$ genauer zu spezifizieren, benötigen wir einige zusätzliche Definitionen: Sei wieder F eine Front mit $|F| = s$. Weiter sei v der erste Knoten in F. Es gelte $\pi(v) = k$. Dann bezeichnet $\mathrm{ops}(F)$ die Anzahl der durchzuführenden Multiplikations- und Additionsoperationen, um aus der Frontal-Matrix $\mathfrak{F}_F$ die Update-Matrix $\mathfrak{U}_F$ und die Spalten $k, k+1, \ldots, k+s-1$ des Cholesky-Faktors L zu erhalten (vgl. Funktion MULTIFRONTAL aus Abbildung 5.5). Der Wert $\mathrm{ops}(F)$ kann wieder mit Hilfe der Formeln (2.6) und (2.7) berechnet werden. Dazu muß bei der Berechnung des Orderings nur $\deg_{G_{k-1}}(v)$ gespeichert werden. Wir erweitern die Funktion ops auf Frontbäume und auf Mengen von Frontbäumen. Dazu setzen wir

$$\mathrm{ops}(\mathcal{T}) = \sum_{F \text{ ist Front in } \mathcal{T}} \mathrm{ops}(F) \qquad \text{und} \qquad \mathrm{ops}(\mathcal{M}) = \sum_{\mathcal{T} \in \mathcal{M}} \mathrm{ops}(\mathcal{T}).$$

Zwei Mengen $\mathcal{M}_1, \mathcal{M}_2$ von Frontbäumen heißen dann gleich schwer oder balanciert, wenn gilt

$$\frac{\min(\mathrm{ops}(\mathcal{M}_1), \mathrm{ops}(\mathcal{M}_2))}{\max(\mathrm{ops}(\mathcal{M}_1), \mathrm{ops}(\mathcal{M}_2))} \geq \beta, \qquad 0 < \beta < 1.$$

Gelingt die Aufteilung in zwei balancierte Mengen nicht, so wird der schwerste Teilbaum $\mathcal{T}_F$ aus $\mathcal{M}$ extrahiert und durch die Teilbäume $\mathcal{T}_{U_1}, \ldots, \mathcal{T}_{U_t}$ unter der Wurzel F von $\mathcal{T}_F$ ersetzt. Die Wurzel F erhält dabei die aktuelle $\mathrm{groupID}$. Diese wird in $\mathrm{frontgroup}(F)$ gespeichert (Zeilen 16–21). Anschließend wird der gesamte Prozeß mit der erweiterten Menge $\mathcal{M}$ wiederholt.

Nach erfolgreicher Aufspaltung müssen die Teilbäume in den Mengen $\mathcal{M}_1$ und $\mathcal{M}_2$ einem Hypercube der Dimension $d-1$ zugeordnet werden. Dies geschieht durch die rekursiven Aufrufe in den Zeilen 23 und 24.

Gilt $d = 0$, so ist ein weiteres Aufspalten der Menge $\mathcal{M}$ nicht erforderlich. Alle Fronten, die zu einem Teilbaum in $\mathcal{M}$ gehören, können exklusiv einem Prozessor zugeordnet werden. Dies geschieht wieder über die aktuelle $\mathrm{groupID}$ (Zeilen 26–28).

Nach Termination der Funktion SPLIT ist jeder Front eine Gruppenidentifikationsnummer zugeordnet. Anhand dieser Nummer kann ein Prozessor entscheiden, ob er an der Faktorisierung der Front beteiligt ist oder nicht. Bezeichne $\mathrm{myprocID} \in \{0, \ldots, 2^r - 1\}$ die Nummer eines Prozessors im Dezimalsystem. Jeder Prozessor berechnet $\mathrm{mygroupID} := 2^r + \mathrm{myprocID}$. Der Prozessor $\mathrm{myprocID}$ ist dann an der Faktorisierung einer Front F beteiligt, wenn gilt

$$\mathrm{frontgroup}(F) \in \{\mathrm{mygroupID}, \lfloor \mathrm{mygroupID}/2 \rfloor, \lfloor \lfloor \mathrm{mygroupID}/2 \rfloor / 2 \rfloor, \ldots, 1\}.$$

Beispielsweise ist in Abbildung 5.7 der Prozessor 10 an der Faktorisierung einer Front F beteiligt, wenn gilt $\mathrm{frontgroup}(F) \in \{6, 3, 1\}$.

Je kleiner die Gruppenidentifikationsnummer ist, desto mehr Prozessoren sind an der Faktorisierung einer Front beteiligt. Um den hierbei entstehenden Kommunikations-Overhead möglichst gering zu halten, sollte insbesondere in den obersten Rekursionsstufen der Funktion SPLIT die äußere while-Schleife (Zeilen 03–22) nur wenige Male durchlaufen werden. Zwar erreicht man dies ganz einfach durch Wahl eines kleineren Wertes für β, dabei ist jedoch zu beachten, daß aufgrund von höheren Lastunterschieden die *Idle-Zeiten* der Prozessoren stark anwachsen können. Strenggenommen muß bereits bei der Berechnung des Orderings auf die Balance des Frontbaumes geachtet werden. Dies kann z. B. dadurch geschehen, daß in einem Nested-Dissection-Verfahren das Gewicht der Teilgraphen stärker in die Bewertung einer Partitionierung eingeht. Hierdurch kann sich jedoch das Gewicht der Separatoren und damit der Grad der Auffüllung von A stark erhöhen. Wir werden auf diesen Zielkonflikt in Abschnitt 5.2.4 näher eingehen.

5.2.2 Die parallele symbolische Faktorisierung

Im Vordergrund der parallelen symbolischen Faktorisierung steht weniger der *Speedup*, als vielmehr die effektive Nutzung des gesamten verteilten Speichers. Nur wenn der Cholesky-Faktor L von Anfang an verteilt gespeichert wird, ist die Faktorisierung von großen Matrizen möglich.

In unserem parallelen Algorithmus geschieht die symbolische Faktorisierung in zwei Schritten. Der Prozessor $\mathrm{myprocID}$ berechnet zunächst für jede Front F, an deren Faktorisierung er beteiligt ist, die Nichtnullstruktur der ersten zu F gehörenden Spalte. Die entsprechende Indexmenge wird in $\mathrm{Indices}(F)$ abgelegt. Ist also wieder v der erste Knoten der Front F mit $\pi(v) = k$, so gilt $\mathrm{Indices}(F) = \mathrm{Struct}(L_{*,k})$. In einem zweiten Schritt bestimmt er, welche Spalten der Front F in seinem Speicher abgelegt werden und – mit Hilfe von $\mathrm{Indices}(F)$ – die Nichtnullstruktur dieser Spalten.

Um den zweistufigen Prozeß vollständig verstehen zu können, müssen wir auf die parallele numerische Faktorisierung vorgreifen. Die grundsätzliche Idee des Faktorisierungsalgorithmus von Gupta et al. [63, 66] besteht darin, die zu der Front F gehörende Frontal-Matrix $\mathfrak{F}_F$ spalten- und zeilenweise auf die Prozessoren zu verteilen. Im Vergleich zu einer spaltenweisen Aufteilung reduziert sich so der Kommunikations-Overhead bei der Faktorisierung der ersten s Spalten von $\mathfrak{F}_F$ um den Faktor $O(\sqrt{p'})$ [84]. Dabei bezeichnet p' die Anzahl der beteiligten Prozessoren. Die ersten s Spalten von $\mathfrak{F}_F$, und damit auch die Spalten $k, \ldots, k+s-1$ des Cholesky-Faktors L, sind also verteilt auf p' Prozessoren abgelegt. Wie diese Verteilung genau aussieht wird später gezeigt. Wir widmen uns zunächst der Berechnung der Indexmengen.

Abbildung 5.9 zeigt das parallele Programm zur Berechnung der Mengen $\mathrm{Indices}(F)$. Als erstes berechnet jeder Prozessor die Gruppenidentifikationsnummer $\mathrm{mygroupID}$. Wie bereits oben gesehen, dient die Nummer zur Ermittlung derjenigen Fronten, an deren Faktorisierung der

```
SETUPINDICESPAR(myprocID, T)
01: mygroupID := 2^r + myprocID; L := ∅;
02: for d := 0 to r do
03:     F := firstPostorder(T);
04:     while F ≠ root(T) do
05:         if frontgroup(F) = mygroupID then
06:             Let U_1, ..., U_t denote the children of F in T.
07:             Determine Indices(F) using PAP^T and Indices(U_1), ..., Indices(U_t).
08:             L := (L ∪ {F}) − {U_1, ..., U_t};
09:         end if
10:         F := nextPostorder(T, F);
11:     end while
12:     if d < r then
13:         Send all Indices(F), F ∈ L, to processor myprocID(d) and receive from
            myprocID(d) all Indices(U). Add each U to L.
14:     mygroupID := ⌊mygroupID /2⌋;
15: end for
```

Abb. 5.9: Funktion SETUPINDICESPAR.

Prozessor myprocID beteiligt ist. In einer for-Schleife, die über alle Dimensionen des Hypercubes läuft, werden die Fronten nach abfallenden Gruppenidentifikationsnummern abgearbeitet. In der ersten Iteration der for-Schleife ($d = 0$) erfolgt die Berechnung von $\mathrm{Indices}(F)$ für alle Fronten F, die exklusiv dem Prozessor myprocID zugeordnet sind. Die eigentliche Berechnung von $\mathrm{Indices}(F)$ geschieht in der Zeile 07. Dazu müssen die Mengen $\mathrm{Indices}(U_i)$, $1 \leq i \leq t$, der Söhne U_i von F bekannt sein. Zeile 07 faßt die Anweisungen 05–10 der Funktion SYMBFACFRONTTREE zusammen. Dabei ist jedoch zu beachten, daß $\mathrm{Indices}(U_i)$ nicht die Struktur der letzten zu U_i gehörenden Spalte enthält, sondern die Struktur der ersten. Dies stellt kein Problem dar, weil die Struktur der letzten Spalte ganz einfach aus der Struktur der ersten Spalte abgeleitet werden kann (vgl. wieder Funktion SYMBFACFRONTTREE).

Ist die Front F exklusiv dem Prozessor myprocID zugeordnet, so sind es auch die Söhne $U_1, \ldots, U_t$ von F. Da der Frontbaum in Postorder-Reihenfolge durchlaufen wird, ist sichergestellt, daß die Mengen $\mathrm{Indices}(U_1), \ldots, \mathrm{Indices}(U_t)$ bekannt sind. Die Situation ändert sich, wenn die Front F zwei oder mehr Prozessoren zugordnet ist. In diesem Fall kennt der Prozessor myprocID nicht immer alle Indexmengen der Söhne. Vor Beginn einer neuen Iteration der for-Schleife erhält myprocID alle benötigten Indexmengen von dem in Dimension d benachbarten Prozessor $\mathrm{myprocID}(d)$. Im Gegenzug verschickt myprocID alle Indexmengen, die der Prozessor $\mathrm{myprocID}(d)$ benötigt. Die Fronten, deren Indexmengen verschickt werden müssen, sind in $\mathcal{L}$ gespeichert. $\mathcal{L}$ enhält alle Fronten, die noch nicht mit ihrer Vater-Front verrechnet wurden.

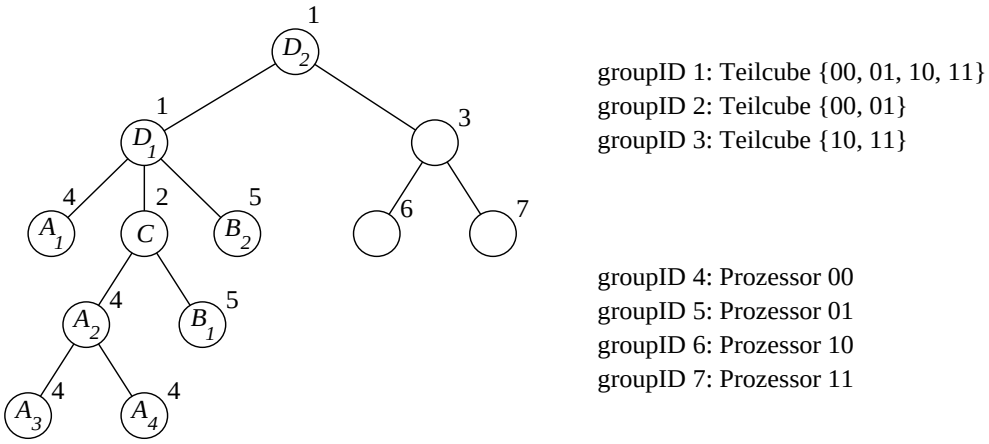


Abb. 5.10: Mapping eines unbalancierten Frontbaumes auf einen zweidimensionalen Hypercube. Die Fronten sind mit ihrer Gruppenidentifikationsnummer beschriftet.

Wir wollen den Datenaustausch an dem Frontbaum aus Abbildung 5.10 veranschaulichen. Die Fronten des Baumes sind mit Hilfe der Funktion SPLIT den Prozessoren eines zweidimensionalen Hypercubes zugeordnet worden. Jede Front ist mit ihrer Gruppenidentifikationsnummer beschriftet. Wir betrachten den Prozessor 00. Für den Prozessor 00 gilt initial $\mathrm{mygroupID} = 2^2 + 0 = 4$. Die Fronten A_i, $i = 1, \ldots, 4$ sind exklusiv dem Prozessor 00 zugeordnet. In der ersten Iteration der for-Schleife berechnet Prozessor 00 die Mengen $\mathrm{Indices}(A_i)$. Da die Fronten A_1 und A_2 in dieser Iteration nicht mit ihrer Vater-Front verrechnet werden, verbleiben sie in $\mathcal{L}$. Durch den Datenaustausch in Zeile 13 erhält Prozessor 01 die Mengen $\mathrm{Indices}(A_1), \mathrm{Indices}(A_2)$ und Prozessor 00 die Mengen $\mathrm{Indices}(B_1), \mathrm{Indices}(B_2)$. Damit gilt für beide Prozessoren $\mathcal{L} = \{A_1, A_2, B_1, B_2\}$

Wir befinden uns jetzt in der zweiten Iteration der for-Schleife ($d = 1$). Für die Prozessoren 00 und 01 gilt $\mathrm{mygroupID} = 2$. Beide Prozessoren berechnen $\mathrm{Indices}(C)$ und fügen C in ihre Menge $\mathcal{L}$ ein. Darüber hinaus entfernen beide Prozessoren die Fronten A_2 und B_1 aus ihrer Menge $\mathcal{L}$. Für beide Prozessoren gilt jetzt $\mathcal{L} = \{A_1, B_2, C\}$. Prozessor 00 verschickt die entsprechenden Indexmengen zu Prozessor 10 und Prozessor 01 zu Prozessor 11. Damit sind alle vier Prozessoren in der Lage, in Iteration $d = 2$ die Indexmenge $\mathrm{Indices}(D_1)$ zu berechnen.

Wir zeigen jetzt, wie die zu einer Front F gehörenden Spalten auf die Prozessoren eines r-dimensionalen Hypercubes verteilt werden. Dazu benötigen wir die folgenden Funktionen: Sei $z \in \{0, \ldots, 2^r - 1\}$ und $0 \leq d \leq r$. Die Funktion $\mathrm{last}_d(z)$ extrahiert die letzten d Bits aus der Binärdarstellung von z. Enthält die Binärdarstellung von z weniger als d Bits, so wird mit Nullen aufgefüllt. Per Definition gilt $\mathrm{last}_0(z) = \epsilon$ für alle $z \in \{0, \ldots, 2^r - 1\}$. Beispiel: $\mathrm{last}_4(5) = \mathrm{last}_4((101)_2) = 0101$, $\mathrm{last}_3(9) = \mathrm{last}_3((1001)_2) = 001$.

Die Funktionen $\mathrm{even}_d(z)$ und $\mathrm{odd}_d(z)$ extrahieren aus den letzten d Bits der Binärdarstellung von z die Bits an gerader bzw. ungerader Position. Wir nehmen an, daß das niederwertigste Bit

```
SETUPSTRUCTUREPAR(myprocID, 𝒯, Indices)
01: mygroupID := 2^r + myprocID;
02: Set Struct(L_{*,k}) := ∅ for all k = 1, ... , n;
03: for d := 0 to r do
04:     F := firstPostorder(𝒯);
05:     while F ≠ root(𝒯) do
06:         if frontgroup(F) = mygroupID then
07:             Let s = |F| and let v be the first vertex in F.
08:             k := π(v);
09:             for each j ∈ {k, ... , k + s − 1} with last_{⌈d/2⌉}(j) = even_d(myprocID) do
10:                 Struct(L_{*,j}) := {i > j; i ∈ Indices(F) and last_{⌊d/2⌋}(i) = odd_d(myprocID)};
11:         end if
12:         F := nextPostorder(𝒯, F);
13:     end while
14:     mygroupID := ⌊mygroupID /2⌋;
15: end for
```

Abb. 5.11: Funktion SETUPSTRUCTUREPAR.

an der Position null steht. Daher hat die von $\mathrm{even}_d(z)$ zurückgegebene binäre Zeichenkette die Länge $\lceil d/2 \rceil$ und die von $\mathrm{odd}_d(z)$ zurückgegebene Zeichenkette die Länge $\lfloor d/2 \rfloor$. Per Definition gilt $\mathrm{even}_0(z) = \epsilon$ und $\mathrm{odd}_0(z) = \mathrm{odd}_1(z) = \epsilon$. Beispiel: $\mathrm{even}_3(9) = \mathrm{even}_3((1\underline{\mathbf{00}}\underline{\mathbf{1}})_2) = 01$, $\mathrm{odd}_4(5) = \mathrm{odd}_4((\underline{\mathbf{0}}1\underline{\mathbf{0}}\underline{\mathbf{1}})_2) = 00$.

Abbildung 5.11 zeigt, wie die zu einer Front gehörenden Spalten auf die Prozessoren des r-dimensionalen Hypercubes verteilt werden. Die Verteilung wird von der binären Zeichenkette des Prozessors bestimmt. Die Bits in $\mathrm{even}_d(\mathrm{myprocID})$ entscheiden welche Spalten und die Bits in $\mathrm{odd}_d(\mathrm{myprocID})$ welche Elemente einer Spalte dem Prozessor $\mathrm{myprocID}$ zugeordnet werden. Von besonderem Interesse ist die for-Schleife in den Zeilen 09–10. Sei F eine dem Prozessor $\mathrm{myprocID}$ zugeordnete Front. Eine Spalte j der Front F ist genau dann dem Prozessor $\mathrm{myprocID}$ zugeteilt, wenn gilt $\mathrm{last}_{\lceil d/2 \rceil}(j) = \mathrm{even}_d(\mathrm{myprocID})$. Ist die Bedingung erfüllt, so sind alle Elemente in Zeile i mit $\mathrm{last}_{\lfloor d/2 \rfloor}(i) = \mathrm{odd}_d(\mathrm{myprocID})$ auf dem Prozessor gespeichert. Bei der Aufteilung der Spalten einer Front spielt also die Dimension d, in der die Front bearbeitet wird, eine entscheidende Rolle.

Gilt $d = 0$, so ist jede Front F mit $\mathrm{frontgroup}(F) = \mathrm{mygroupID}$ exklusiv dem Prozessor $\mathrm{myprocID}$ zugeordnet. Wegen $\mathrm{last}_{\lceil d/2 \rceil}(j) = \mathrm{even}_d(\mathrm{myprocID}) = \epsilon$ für alle $j \in \{k, \ldots, k + s - 1\}$ und $\mathrm{last}_{\lfloor d/2 \rfloor}(i) = \mathrm{odd}_d(\mathrm{myprocID}) = \epsilon$ für alle $i \in \mathrm{Indices}(F)$ sind die s Spalten der Front F vollständig auf dem Prozessor $\mathrm{myprocID}$ gespeichert. Im Falle $d = 1$ ist jede Front F mit $\mathrm{frontgroup}(F) = \mathrm{mygroupID}$ den Prozessoren $\mathrm{myprocID}$ und $\mathrm{myprocID}(0)$ zugeordnet ($\mathrm{myprocID}(0)$ bezeichnet wieder den zu $\mathrm{myprocID}$ benachbarten Prozessor in Dimension 0).

Durch die for-Schleife werden die geraden Spalten von F auf den Prozessor mit der geraden Nummer und die ungeraden Spalten auf den Prozessor mit der ungeraden Nummer abgebildet. Auch hier gilt $\text{last}_{\lfloor d/2 \rfloor}(i) = \epsilon$ für alle $i \in \text{Indices}(F)$. Da die Funktion odd_d im Falle $d = 1$ immer den Wert ϵ zurückliefert, erfolgt keine zeilenweise Aufteilung der Spalten von F.

Dies ändert sich in Iteration $d = 2$. Die Spalten der Front F werden jetzt auf die Prozessoren myprocID und $\text{myprocID}(0)$ sowie auf die beiden zu myprocID und $\text{myprocID}(0)$ benachbarten Prozessoren in Dimension eins verteilt. Seien $\alpha 00$, $\alpha 01$, $\alpha 10$ und $\alpha 11$, $\alpha \in \{0,1\}^{r-2}$ die binären Zeichenketten der vier Prozessoren. Alle geraden Spalten der Front F werden auf die Prozessoren $\alpha 00$, $\alpha 10$ und alle ungeraden Spalten auf die Prozessoren $\alpha 01$, $\alpha 11$ abgebildet. In Zeile 10 erfolgt darüber hinaus eine zeilenweise Aufteilung. Wegen $\text{odd}_2(\alpha 00) = 0$ und $\text{odd}_2(\alpha 10) = 1$ werden die Elemente einer geraden Spalte mit geradem Zeilenindex Prozessor $\alpha 00$ und die mit einem ungeraden Zeilenindex Prozessor $\alpha 10$ zugeordnet. Gleiches gilt für die ungeraden Spalten und die Prozessoren $\alpha 01$ und $\alpha 11$. Man erhält so auch für höhere Dimensionen eine gleichmäßige, zweidimensionale Aufteilung der zu einer Front gehörenden Spalten auf die beteiligten Prozessoren.

Abschließend sei angemerkt, daß innerhalb der Funktion SETUPINDICESPAR kein Datenaustausch unter den Prozessoren stattfindet.

5.2.3 Die parallele numerische Faktorisierung

Bei der parallelen numerischen Faktorisierung wird die zweidimensionale Aufteilung der Spalten einer Front F auf die Spalten der Frontal-Matrix $\mathfrak{F}_F$ erweitert. Wird also $\mathfrak{F}_F$ in Iteration d faktorisiert, so speichert der Prozessor myprocID die Einträge $f_{i,j}$ mit $i,j \in \text{Indices}(F)$, $i \geq j$ und $\text{last}_{\lfloor d/2 \rfloor}(i) = \text{odd}_d(\text{myprocID})$, $\text{last}_{\lceil d/2 \rceil}(j) = \text{even}_d(\text{myprocID})$. Der parallele Multifrontal-Algorithmus basiert auf dieser zweidimensionalen Verteilung der Frontal-Matrizen. Als Ergebnis erhält man die in Abbildung 5.12 beschriebene Funktion MULTIFRONTALPAR.

Die Funktion MULTIFRONTALPAR setzt sich aus Elementen der Funktionen SETUPINDICESPAR und MULTIFRONTAL zusammen. Die Elemente der Funktion SETUPINDICESPAR bilden dabei den Rahmen des parallelen Multifrontal-Algorithmus. Wir konzentrieren uns zunächst auf diesen Rahmen. Dazu blenden wir in Abbildung 5.12 die Zeilen 06–12 aus.

Wie in SETUPINDICESPAR so berechnet auch hier jeder Prozessor als erstes die Gruppenidentifikationsnummer mygroupID. In einer for-Schleife, die über alle Dimensionen des Hypercubes läuft, werden zuerst die Frontal-Matrizen betrachtet, die exklusiv einem Prozessor zugeordnet sind. Danach die Frontal-Matrizen, die auf $2, 4, \ldots, 2^r$ Prozessoren verteilt sind. Die Menge $\mathcal{L}$ enthält alle Fronten, deren Update-Matrix noch nicht mit einer Frontal-Matrix verrechnet wurde. Die Menge $\mathcal{L}$ dient wieder dem Datenaustausch. Vor Beginn einer neuen Iteration der for-Schleife sendet der Prozessor myprocID die Update-Matrizen $\mathfrak{U}_F$, $F \in \mathcal{L}$, zu dem in Dimension d benachbarten Prozessor $\text{myprocID}(d)$. Im Gegenzug erhält er die von $\text{myprocID}(d)$

```
MULTIFRONTALPAR(myprocID, 𝒯, Indices)
01: mygroupID := 2^r + myprocID; ℒ := ∅;
02: for d := 0 to r do
03:     F := firstPostorder(𝒯);
04:     while F ≠ root(𝒯) do
05:         if frontgroup(F) = mygroupID then
06:             Set up (distributed) frontal matrix 𝔉_F using Indices(F).
07:             Let k_1, ..., k_{s_p} be the columns that are associated with F and mapped to myprocID.
08:             Fill columns k_1, ..., k_{s_p} with columns k_1, ..., k_{s_p} of PAP^T.
09:             Let U_1, ..., U_t denote the children of F in 𝒯.
10:             for each front U_i do
11:                 𝔉_F := 𝔉_k ⊕ 𝔘_{U_i}.
12:             Perform s = |F| steps of parallel elimination on 𝔉_F to obtain the (distributed)
                columns k_1, ..., k_{s_p} of L and the (distributed) update matrix 𝔘_F.
13:             ℒ := (ℒ ∪ {F}) − {U_1, ..., U_t};
14:         end if
15:         F := nextPostorder(𝒯, F);
16:     end while
17:     if d < r then
18:         Send all 𝔘_F, F ∈ ℒ, to processor myprocID(d) and receive from
            myprocID(d) all 𝔘_U. Add each U to ℒ.
19:         if d mod 2 = 0 then
20:             Split each update matrix 𝔘_F, F ∈ ℒ, column-wise.
21:         if d mod 2 = 1 then
22:             Split each update matrix 𝔘_F, F ∈ ℒ, row-wise.
23:     end if
24:     mygroupID := ⌊mygroupID /2⌋;
25: end for
```

Abb. 5.12: Funktion MULTIFRONTALPAR.

noch nicht verrechneten Update-Matrizen $\mathfrak{U}_U$. Beide Prozessoren sind jetzt in der Lage, die in Iteration $d+1$ zugeordneten Frontal-Matrizen zu faktorisieren.

Da sich die Binärdarstellungen der Prozessoren $\mathrm{myprocID}$ und $\mathrm{myprocID}(d)$ in den ersten d Bits (also in den Bits an den Positionen $0,\ldots,d-1$) nicht unterscheiden, ist die zweidimensionale Aufteilung der verschickten und erhaltenen Update-Matrizen gleich. Hierdurch ist garantiert, daß nach Addition einer auf Prozessor $\mathrm{myprocID}$ gespeicherten Update-Matrix $\mathfrak{U}_F$, $F \in \mathcal{L}$, mit einer von $\mathrm{myprocID}(d)$ erhaltenen Update-Matrix $\mathfrak{U}_U$ die neue Matrix $\mathfrak{U}_F \oplus \mathfrak{U}_U$ nur solche Spalten und Zeilen enthält, die auch $\mathrm{myprocID}$ zugeordnet sind.

In der Iteration $d+1$ tritt zum ersten Mal der Fall ein, daß die Prozessoren $\mathrm{myprocID}$ und $\mathrm{myprocID}(d)$ gemeinsam an der Faktorisierung einer Frontal-Matrix arbeiten. Bislang war die zweidimensionale Aufteilung der Frontal- und Update-Matrizen auf beiden Prozessoren gleich. Dies gilt jedoch nicht mehr in Iteration $d+1$. Ist d beispielsweise gerade, so erhöht sich die Länge der Zeichenkette $\mathrm{even}_{d+1}(\mathrm{myprocID})$ im Vergleich zu $\mathrm{even}_d(\mathrm{myprocID})$ um eins. Die Zeichenketten $\mathrm{even}_{d+1}(\mathrm{myprocID})$ und $\mathrm{even}_{d+1}(\mathrm{myprocID}(d))$ unterscheiden sich genau in dem zusätzlichen Bit an Position d. Dies führt dazu, daß die von $\mathrm{myprocID}$ und $\mathrm{myprocID}(d)$ gemeinsam bearbeiteten Frontal-Matrizen noch einmal spaltenweise unter den beiden Prozessoren aufgeteilt werden. Dementsprechend ist eine spaltenweise Aufteilung der Update-Matrizen in $\mathcal{L}$ notwendig. Analog müssen im Falle d ungerade die Update-Matrizen zeilenweise aufgeteilt werden.

Wenden wir uns nun dem Inneren der while-Schleife zu (Zeilen 06–12). Im Falle $d=0$ ist die Frontal-Matrix $\mathfrak{F}_F$ exklusiv dem Prozessor $\mathrm{myprocID}$ zugeordnet. Daher gilt $\{k_1,\ldots,k_{s_p}\} = \{k,k+1,\ldots,k+s-1\}$. Die Anweisungen in den Zeilen 06–12 entsprechen den Anweisungen des sequentiellen Multifrontal-Algorithmus (vgl. Abbildung 5.5). Im Falle $d>0$ sind die Einträge der Frontal-Matrix $\mathfrak{F}_F$ auf mehrere Prozessoren verteilt. Es gilt jetzt $\{k_1,\ldots,k_{s_p}\} \subset \{k,k+1,\ldots,k+s-1\}$. Ist $d>1$, so muß bei der Auffüllung der Spalten $k_1,\ldots,k_{s_p}$ mit Elementen aus PAP^T beachtet werden, daß die Spalten nicht vollständig auf $\mathrm{myprocID}$ abgelegt sind. Nach Addition der Update-Mtrizen $\mathfrak{U}_{U_1},\ldots,\mathfrak{U}_{U_t}$ kann die verteilte Frontal-Matrix faktorisiert werden. Wir verwenden dazu eine parallele Version des Algorithmus DENSEBLOCKFACTOR aus Abbildung 5.6. Bevor wir genauer auf diesen Algorithmus eingehen, wollen wir die generelle Vorgehensweise der Funktion MULTIFRONTALPAR an einem Beispiel illustrieren.

Dazu betrachten wir noch einmal den in Abbildung 5.7 dargestellten Frontbaum. Nach Abschluß der while-Schleife in Iteration $d=0$ speichert Prozessor 00 die Update-Matrix $\mathfrak{U}_{\{v_2,v_9,v_{16}\}}$ und Prozessor 01 die Update-Matrix $\mathfrak{U}_{\{v_{30},v_{37},v_{44}\}}$. Die Update-Matrizen sind jeweils exklusiv den Prozessoren zugeordnet. In Zeile 18 kommt es zum Austausch der Update-Matrizen. Beide Prozessoren speichern jetzt $\mathfrak{U}_{\{v_2,v_9,v_{16}\}}$ und $\mathfrak{U}_{\{v_{30},v_{37},v_{44}\}}$.

Es gilt

$$\mathfrak{U}_{\{v_2,v_9,v_{16}\}} = \begin{pmatrix} u_{19,19} & & & & & & \\ u_{20,19} & \mathbf{u_{20,20}} & & & & & \\ u_{21,19} & \mathbf{u_{21,20}} & u_{21,21} & & & & \\ u_{46,19} & \mathbf{u_{46,20}} & u_{46,21} & \mathbf{u_{46,46}} & & & \\ u_{47,19} & \mathbf{u_{47,20}} & u_{47,21} & \mathbf{u_{47,46}} & u_{47,47} & & \\ u_{48,19} & \mathbf{u_{48,20}} & u_{48,21} & \mathbf{u_{48,46}} & u_{48,47} & \mathbf{u_{48,48}} & \\ u_{49,19} & \mathbf{u_{49,20}} & u_{49,21} & \mathbf{u_{49,46}} & u_{49,47} & \mathbf{u_{49,48}} & u_{49,49} \end{pmatrix}$$

und

$$\mathfrak{U}_{\{v_{30},v_{37},v_{44}\}} = \begin{pmatrix} u'_{19,19} & & & & & & \\ u'_{20,19} & \mathbf{u'_{20,20}} & & & & & \\ u'_{21,19} & \mathbf{u'_{21,20}} & u'_{21,21} & & & & \\ u'_{43,19} & \mathbf{u'_{43,20}} & u'_{43,21} & u'_{43,43} & & & \\ u'_{44,19} & \mathbf{u'_{44,20}} & u'_{44,21} & u'_{44,43} & \mathbf{u'_{44,44}} & & \\ u'_{45,19} & \mathbf{u'_{45,20}} & u'_{45,21} & u'_{45,43} & \mathbf{u'_{45,44}} & u'_{45,45} & \\ u'_{46,19} & \mathbf{u'_{46,20}} & u'_{46,21} & u'_{46,43} & \mathbf{u'_{46,44}} & u'_{46,45} & \mathbf{u'_{46,46}} \end{pmatrix}.$$

Nach der spaltenweisen Aufteilung der Update-Matrizen in Zeile 20 speichert Prozessor 00 nur noch die in Fettdruck dargestellten geraden Spalten und Prozessor 01 die in normaler Schrift dargestellten ungeraden Spalten. Beide Prozessoren bilden die verteilte Frontal-Matrix $\mathfrak{F}_{\{v_{22},v_{23},v_{24}\}}$. Bezüglich Prozessor 00 gilt

$$\mathfrak{F}_{\{v_{22},v_{23},v_{24}\}} = \begin{pmatrix} a_{20,20} + u_{20,20} + u'_{20,20} & & & \\ a_{21,20} + u_{21,20} + u'_{21,20} & & & \\ a_{43,20} + u'_{43,20} & & & \\ a_{44,20} + u'_{44,20} & u'_{44,44} & & \\ a_{45,20} + u'_{45,20} & u'_{45,44} & & \\ a_{46,20} + u_{46,20} + u'_{46,20} & u'_{46,44} & u_{46,46} + u'_{46,46} & \\ a_{47,20} + u_{47,20} & 0 & u_{47,46} & \\ a_{48,20} + u_{48,20} & 0 & u_{48,46} & u_{48,48} \\ a_{49,20} + u_{49,20} & 0 & u_{49,46} & u_{49,48} \end{pmatrix}.$$

Bezeichne

$$\mathfrak{U}_{\{v_{22},v_{23},v_{24}\}} = \begin{pmatrix} \mathbf{u''_{44,44}} & \\ u''_{45,44} & \\ \mathbf{u''_{46,44}} & \mathbf{u''_{46,46}} \\ u''_{47,44} & u''_{47,46} \\ \mathbf{u''_{48,44}} & \mathbf{u''_{48,46}} \quad \mathbf{u''_{48,48}} \\ u''_{49,44} & u''_{49,46} \quad u''_{49,48} \end{pmatrix}$$

	0	1	2	3	4	5	6	7	8	9	10	11
0	0											
1	2	3										
2	8	9	12									
3	10	11	14	15								
4	0	1	4	5	0							
5	2	3	6	7	2	3						
6	8	9	12	13	8	9	12					
7	10	11	14	15	10	11	14	15				
8	0	1	4	5	0	1	4	5	0			
9	2	3	6	7	2	3	6	7	2	3		
10	8	9	12	13	8	9	12	13	8	9	12	
11	10	11	14	15	10	11	14	15	10	11	14	15

	0	1	2	3	4	5	6	7	8	9	10	11
0	0											
1	0	0										
2	2	2	3									
3	2	2	3	3								
4	8	8	9	9	12							
5	8	8	9	9	12	12						
6	10	10	11	11	14	14	15					
7	10	10	11	11	14	14	15	15				
8	0	0	1	1	4	4	5	5	0			
9	0	0	1	1	4	4	5	5	0	0		
10	2	2	3	3	6	6	7	7	2	2	3	
11	2	2	3	3	6	6	7	7	2	2	3	3

Abb. 5.13: Verteilung der Frontal-Matrix $\mathfrak{F}_F$ mit $\text{Indices}(F) = \{0, 1, \ldots, 11\}$ auf 16 Prozessoren. Die linke Abbildung zeigt eine einfache zyklische, die rechte eine 2×2 block-zyklische Verteilung.

die nach der Faktorisierung von $\mathfrak{F}_{\{v_{22},v_{23},v_{24}\}}$ auf Prozessor 00 gespeicherte Update-Matrix. Prozessor 00 schickt die Update-Matrix zu dem in Dimension eins benachbarten Prozessor 10. Auch Prozessor 10 sind die geraden Spalten einer Update-Matrix zugeordnet. Anschließend wird $\mathfrak{U}_{\{v_{22},v_{23},v_{24}\}}$ zeilenweise aufgeteilt (vgl. Zeile 22). Prozessor 00 behält die in Fettdruck dargestellten geraden Zeilen von $\mathfrak{U}_{\{v_{22},v_{23},v_{24}\}}$ und Prozessor 10 die in normaler Schrift dargestellten ungeraden Zeilen.

Wie bereits oben erwähnt, basiert der parallele numerische Kern zur Faktorisierung der Frontal-Matrizen auf der in Abbildung 5.6 dargestellten Funktion DENSEBLOCKFACTOR. Wir verzichten an dieser Stelle auf die explizite Angabe eines Algorithmus, da dieser sehr technisch ist. Vielmehr beschreiben wir die parallele Faktorisierung der Frontal-Matrizen an einem Beispiel.

Dazu betrachten wir eine Frontal-Matrix $\mathfrak{F}_F$ mit $\text{Indices}(F) = \{0, 1, \ldots, 11\}$. Es sei angenommen, daß $2^4 = 16$ Prozessoren an der Faktorisierung der Frontal-Matrix beteiligt sind. Wir befinden uns also in Iteration $d = 4$ der Funktion MULTIFRONTALPAR. Die Elemente $f_{i,j}$ der Frontal-Matrix sind wie in Abbildung 5.13 (links) angegeben auf die 16 Prozessoren verteilt. Dabei bezeichnet die in Zeile i und Spalte j stehende Zahl den Prozessor, auf dem $f_{i,j}$ abgebildet ist. Zum Beispiel ist $f_{2,1}$ dem Prozessor $\text{myprocID} = 9$ zugeordnet. Die Verteilung der Elemente auf die Prozessoren entspricht genau dem oben vorgestellten Schema. Für den Prozessor $\text{myprocID} = 9$ gilt beispielsweise $\text{even}_4(9) = \text{even}_4((1\underline{0}0\underline{1})_2) = 01$ und $\text{odd}_4(9) = \text{odd}_4((\underline{1}0\underline{0}1)_2) = 10$. Damit speichert der Prozessor $\text{myprocID} = 9$ alle Elemente $f_{i,j}$ mit $\text{last}_2(i) = 10$ und $\text{last}_2(j) = 01$. Der Prozessor „sieht“ also die Frontal-Matrix

$$\begin{pmatrix} f_{2,1} & & \\ f_{6,1} & f_{6,5} & \\ f_{10,1} & f_{10,5} & f_{10,9} \end{pmatrix}.$$

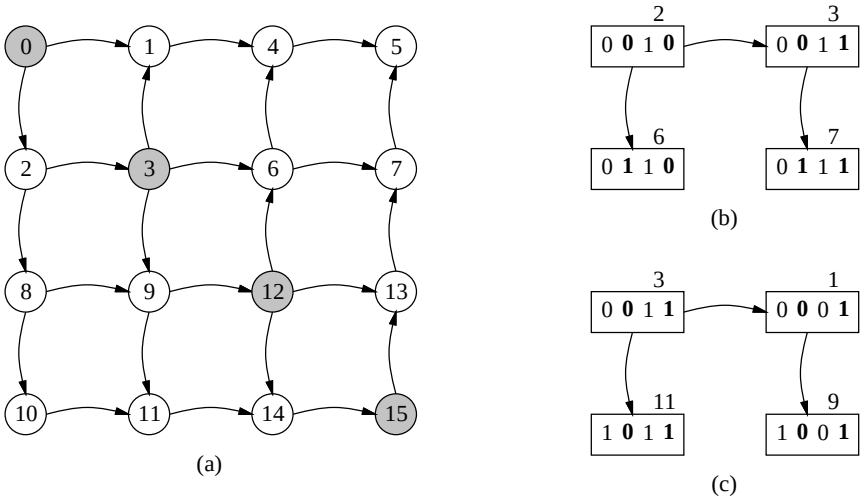


Abb. 5.14: Kommunikationsschema des parallelen numerischen Kerns für 16 Prozessoren. (a) zeigt die horizontale und vertikale Verteilung der auf den Prozessoren $\mathrm{myprocID} = 0, 2, 8, 10$ faktorisierten Spalte. Die Diagonalprozessoren sind grau gefärbt. (b) zeigt wie die von Prozessor $\mathrm{myprocID} = 2$ initiierte horizontale Verteilung als Broadcast implementiert werden kann. Die geraden Bits sind in Fettdruck dargestellt. (c) zeigt den Broadcast für die vertikale Verteilung am Diagonalprozessor $\mathrm{myprocID} = 3$.

Wir nehmen an, daß die erste Spalte von $\mathfrak{F}_F$ faktorisiert wurde. Prozessor $\mathrm{myprocID} = 9$ benötigt zur Aktualisierung seines Teils der Frontal-Matrix $\mathfrak{F}_F$ den Vektor $(l_{2,0}, l_{6,0}, l_{10,0})^T$ und die Skalare $l_{1,0}, l_{5,0}, l_{9,0}$ (vgl. auch Funktion DENSEFACTOR aus Abbildung 5.4). Der Vektor $(l_{2,0}, l_{6,0}, l_{10,0})^T$ ist auf dem Prozessor $\mathrm{myprocID} = 8$ gespeichert. Man überlegt sich leicht, daß auch die Prozessoren $\mathrm{myprocID} = 12$ und $\mathrm{myprocID} = 13$ den Vektor zur Durchführung ihrer Aktualisierungen benötigen. Alle diese vier Prozessoren liegen in Abbildung 5.13 (links) auf einer horizontalen Linie. Die Skalare $l_{1,0}, l_{5,0}, l_{9,0}$ sind auf Prozessor $\mathrm{myprocID} = 1$ gespeichert. Die Skalare werden außerdem von den Prozessoren $\mathrm{myprocID} = 3$ und $\mathrm{myprocID} = 11$ benötigt. Die vier Prozessoren liegen in Abbildung 5.13 (links) auf einer vertikalen Linie.

Es ergibt sich so ein Kommunikationsschema wie es in Abbildung 5.14 (a) dargestellt ist. Nachdem die Prozessoren $\mathrm{myprocID} = 0, 2, 8, 10$ die erste Spalte faktorisiert haben, werden die Elemente der Faktorspalte horizontal an die anderen Prozessoren verteilt. Jeder *Diagonalprozessor* – d. h. jeder Prozessor mit $\mathrm{even}_d(\mathrm{myprocID}) = \mathrm{odd}_d(\mathrm{myprocID})$ – initiiert darüber hinaus ein vertikales Verschicken der erhaltenen Elemente. In unserem Beispiel mit 16 Prozessoren gibt es $\sqrt{16} = 4$ Diagonalprozessoren. Dabei handelt es sich um die Prozessoren $0\mathbf{00}\mathbf{0}$, $0\mathbf{01}\mathbf{1}$, $0\mathbf{11}\mathbf{0}$ und $1\mathbf{11}\mathbf{1}$ (um zu verdeutlichen, daß die binären Zeichenketten aus geraden bzw. ungeraden Bits gleich sind, haben wir die geraden Bits hervorgehoben). Der Prozessor $\mathrm{myprocID} = 9$ erhält also den Vektor $(l_{2,0}, l_{6,0}, l_{10,0})^T$ direkt von Prozessor $\mathrm{myprocID} = 8$ und die Skalare $l_{1,0}, l_{5,0}, l_{9,0}$ von Prozessor $\mathrm{myprocID} = 1$ über den Diagonalprozessor $\mathrm{myprocID} = 3$.

In unserem parallelen numerischen Kern ist das horizontale und das vertikale Verschicken als *Broadcast-Operation* implementiert. Der horizontale Broadcast wird initiiert von den an der Faktorisierung der Spalte beteiligten Prozessoren, der vertikale Broadcast von den Diagonalprozessoren. Dies ist möglich, da die geraden Bits der horizontal auf einer Linie liegenden Prozessoren einen Teilcube des r-dimensionalen Hypercubes bilden. Gleiches gilt für die ungeraden Bits der vertikal auf einer Linie liegenden Prozessoren. Abbildung 5.14 (b) zeigt den von Prozessor $\mathrm{myprocID} = 2$ initiierten horizontalen Broadcast. Es werden ausschließlich Kanten benutzt, die in einer geraden Dimension verlaufen. Der von dem Diagonalprozessor $\mathrm{myprocID} = 3$ initiierte vertikale Broadcast ist in Abbildung 5.14 (c) dargestellt. Der vertikale Broadcast benutzt ausschließlich Kanten, die in einer ungeraden Dimension verlaufen.

Auch die Faktorisierung der ersten Spalte erfordert einen vertikalen Broadcast. Dieser wird von dem *Pivotprozessor* $\mathrm{myprocID} = 0$ initiiert. Der Broadcast dient zur Verteilung des Diagonalelementes $l_{0,0}$ an die Prozessoren $\mathrm{myprocID} = 2, 8, 10$.

Ist d ungerade, so ergibt sich ein ähnliches Kommunikationsschema. In diesem Fall bilden die 2^d Prozessoren kein Quadrat wie in Abbildung 5.14 (a) dargestellt, sondern ein Rechteck. Eine horizontale Linie des Rechtecks enthält $2^{\lceil d/2 \rceil}$ Prozessoren, eine vertikale $2^{\lfloor d/2 \rfloor}$. Da die horizontale Linie doppelt so viele Prozessoren enthält wie die vertikale, gibt es in ihr zwei Diagonalprozessoren. Für einen Diagonalprozessor gilt jetzt $\mathrm{even}_{d-1}(\mathrm{myprocID}) = \mathrm{odd}_d(\mathrm{myprocID})$.

Abschließend sei angemerkt, daß zur Reduzierung des Kommunikationsvolumens die Elemente der Frontal-Matrix $\mathfrak{F}_F$ nicht wie in Abbildung 5.13 (links) angegeben in einfacher zyklischer From, sondern in block-zyklischer Form auf die Prozessoren verteilt werden (vgl. auch Gupta et al. [63, 66]). Um Blöcke der Größe $2^l \times 2^l$ zu erhalten, speichert Prozessor $\mathrm{myprocID}$ alle Elemente $f_{i,j}$ mit $\mathrm{last}_{\lfloor d/2 \rfloor}(i/2^l) = \mathrm{odd}_d(\mathrm{myprocID})$ und $\mathrm{last}_{\lceil d/2 \rceil}(j/2^l) = \mathrm{even}_d(\mathrm{myprocID})$. Abbildung 5.13 (rechts) zeigt eine 2×2 block-zyklische Verteilung der Frontal-Matrix.

5.2.4 Experimentelle Ergebnisse

Bei dem von uns implementierten parallelen Faktorisierungsalgorithmus **pspace** (**P**arallel **SPA**rse **C**holesky **E**limination) handelt es sich um eine Weiterentwicklung des sequentiellen Programms **space**. Wie in **space** wird zunächst der Frontbaum $\mathcal{T}$ modifiziert. Diese Aufgabe übernimmt der Prozessor $\mathrm{myprocID} = 0$. Anschließend berechnet der Prozessor mit Hilfe der Funktion SPLIT eine Verteilung der Fronten auf die Prozessoren des Hypercubes. Der Aufwand für dieses sequentielle Preprocessing beträgt selbst für sehr große Matrizen lediglich ein bis zwei Sekunden. Schließlich wird der modifizierte Frontbaum sowie das in $\mathrm{frontgroup}$ gespeicherte Mapping an die restlichen Prozessoren des Hypercubes geschickt. Danach können alle Prozessoren gemeinsam die symbolische und numerische Faktorisierung durchführen. Die Prozessoren rufen dazu die Funktionen SETUPINDICESPAR, SETUPSTRUCTUREPAR und MULTIFRONTALPAR auf. Die folgende Liste faßt die wichtigsten Merkmale unseres parallelen Algorithmus zusammen.

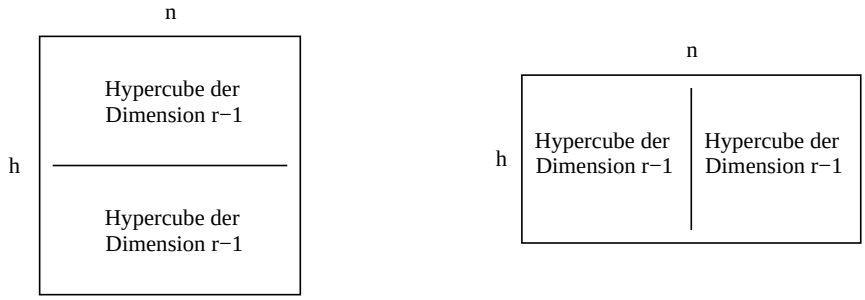


Abb. 5.15: Einbettung des r-dimensionalen Hypercubes in ein $h \times n$-Gitter, $hn = 2^r$. Gilt $h = n$ (links), so wird das quadratische Gitter horizontal in zwei rechteckige Gitter aufgespalten. In jedes rechteckige Gitter wird ein $r-1$-dimensionaler Hypercube eingebettet. Im Falle $h < n$ (rechts) wird das Gitter vertikal entlang der längeren Seite aufgespalten.

- In der Funktion SPLIT ist der Parameter β auf den Wert 0.95 gesetzt. Durch den Austausch der Update-Matrizen in MULTIFRONTALPAR (Zeile 18) synchronisieren sich die Prozessoren $\mathrm{myprocID}$ und $\mathrm{myprocID}(d)$. Um hierbei größere Idle-Zeiten zu vermeiden, sollte β immer auf einen Wert nahe eins gesetzt werden.
- Zur Reduzierung des Kommunikations-Overheads benutzen wir ein $2^l \times 2^l$ block-zyklisches Mapping (vgl. auch Abbildung 5.13). Die optimale Wahl des Parameters l hängt von der *Latenzzeit* des parallelen Systems ab. In unseren Experimenten haben wir l auf den Wert drei gesetzt.
- Zur Faktorisierung der exklusiv einem Prozessor zugewiesenen Frontal-Matrizen benutzten wir wieder einen numerischen Kern, der auf Blöcken der Größe 3×3 arbeitet. Der Kern „verträgt“ sich jedoch nicht mit dem block-zyklischen Mapping. Daher benutzen wir zur Faktorisierung der auf mehrere Prozessoren verteilten Frontal-Matrizen einen einfachen 1×1-Kern.

Alle unsere experimentellen Ergebnisse wurden auf einem Parsytec CC (Cognitive Computing) System mit 16 Knoten ermittelt. Jeder Knoten des Parallelrechners besteht aus einem 133 MHz Motorola PowerPC 604 Prozessor und 64 MByte Hauptspeicher. Die Knoten sind zu einem quadratischen Gitter vernetzt. Der von unserem Faktorisierungsalgorithmus benötigte Hypercube ist wie in Abbildung 5.15 gezeigt in das Gitter eingebettet. Die einfache rekursive Einbettung hat den Vorteil, daß Teilcubes auf disjunkte Teilgitter abgebildet werden. Hierdurch wird vermieden, daß sich die Kommunikation innerhalb einer Prozessorgruppe mit der Kommunikation innerhalb einer anderen überlagert. Zum Nachrichtenaustausch benutzen wir die von dem Betriebssystem PARIX[†] bereitgestellten synchronen Kommunikationsroutinen. Die Latenzzeit beträgt in diesem Fall lediglich 90 Mikrosekunden.

[†]PARIX (**PAR**allel Extensions to Un**IX**) ist ein kommerzielles Produkt der Firma Parsytec.

+-förmige Separatoren			Anzahl Prozessoren			
n	$\eta(L)/10^3$	$\theta(L)/10^6$	2	4	8	16
255	2235	291	5.48	3.14 (1.75)	1.95 (1.61)	1.24 (1.57)
400	6230	1161	22.27	12.27 (1.82)	7.20 (1.70)	4.40 (1.63)
511	10900	2461	—	26.57 —	15.05 (1.76)	8.86 (1.70)

Tab. 5.2: Faktorisierung eines quadratischen Gitters mit Seitenlänge $n = 255, 400, 511$ auf $p = 2, 4, 8, 16$ Prozessoren. Zur Numerierung der Gitterknoten wurde Georges Nested-Dissection-Verfahren benutzt.

Tabelle 5.2 zeigt die Laufzeiten zur Berechnung des Cholesky-Faktors dreier Laplace-Matrizen, die ein quadratisches Gitter mit Seitenlänge $n = 255$, $n = 400$ und $n = 511$ induzieren. Diese Matrizen stellen einen Idealfall dar, weil das asymptotisch optimale Nested-Dissection-Ordering zugleich einen balancierten Frontbaum erzeugt. Darüber hinaus entsteht bei der Faktorisierung dieser Matrizen lediglich ein Kommunikations-Overhead von $O(n^2\sqrt{p})$ [63, 66]. Der linke Teil der Tabelle 5.2 zeigt die Anzahl der subdiagonalen Nichtnullelemente in L (in Tsd.) und die Anzahl der zur Berechnung von L benötigten Operationen (in Mio.). Der rechte Teil zeigt die Laufzeiten auf $p = 2, 4, 8, 16$ Prozessoren. In Klammern ist jeweils der Speedup angegeben, der sich beim Übergang zu einem höherdimensionalen Hypercube ergibt. Im optimalen Fall beträgt der Speedup zwei.

Der Speedup ist am geringsten bei der Faktorisierung des 255×255-Gitters. Dies ist nicht verwunderlich, da bereits auf acht Prozessoren zur Berechnung des Cholesky-Faktors weniger als zwei Sekunden benötigt werden. Da jeder Knoten des CC Systems über lediglich 64 MByte Speicher verfügt, ist im Falle $n = 511$ eine Faktorisierung auf zwei Prozessoren nicht möglich.

Wenden wir uns nun der Faktorisierung beliebiger Matrizen zu. In Abschnitt 4.3 haben wir gesehen, daß sich bei Verwendung des von uns entwickelten Ordering-Verfahrens der Aufwand zur Berechnung des Cholesky-Faktors erheblich reduzieren läßt. Charakteristisch für unser Verfahren ist, daß Knotenseparatoren als Ränder der von einem unvollständigen Bottom-up-Ordering gebildeten Gebiete interpretiert werden. Deshalb spielt die Balance der Teilgraphen bei der Bewertung einer Partitionierung nur eine untergeordnete Rolle. Dies hat unmittelbar zur Folge, daß der durch das Ordering induzierte Frontbaum unbalanciert ist. Im Vergleich zu einem Nested-Dissection-Ordering kann sich so die Anzahl der Fronten, die einer großen Prozessorgruppe zugeteilt werden, drastisch erhöhen. Dies legt die Vermutung nahe, daß die von unserem Verfahren generierten Orderings nicht für die parallele Faktorisierung geeignet sind.

Die folgenden Tabellen zeigen anhand zweier Beispiele, daß diese Annahme nicht zutrifft. Dazu betrachten wir die Matrizen BCSSTK30 und MAT02HBF. Die Matrix BCSSTK30 besitzt wie die in Abschnitt 4.2.1 näher betrachtete Matrix BCSSTK25 einen großen Aspekt-Ratio. Daher ist das von **pord** generierte Ordering vom Typ $(\mathrm{AMMF}, \mathrm{AMMF})$ sehr viel besser als das

Nested-Dissection-Ordering vom Typ $(\mathrm{AMMF}, \mathrm{ND})$ (vgl. Tabelle 4.2). Im Gegensatz dazu sind die Unterschiede bei der Matrix MAT02HBF nicht so groß. Es stellt sich nun die Frage, wie die Orderings die Effizienz der parallelen numerischen Faktorisierung beeinflussen.

Betrachten wir zunächst die Faktorisierung der Matrix BCSSTK30. Wir haben die Faktorisierung nacheinander auf $p = 2, 4, 8, 16$ Prozessoren durchgeführt. Die Tabellen 5.3 und 5.4 zeigen für jeden der vier Läufe wie die arithmetischen Operationen in Abhängigkeit von dem gewählten Ordering auf die Prozessoren des Hypercubes verteilt sind. Dazu ist jede Tabelle in vier horizontale Abschnitte unterteilt. Jeder Abschnitt enthält einen vollständigen Binärbaum über die an der Faktorisierung beteiligten Prozessoren. Die Zahlen an den Blättern des Binärbaumes geben an, wieviele arithmetische Operationen exklusiv einem Prozessor zugeordnet sind. Die Zahlen darunter zeigen die Anzahl der arithmetischen Operationen, die von einer Prozessorgruppe der Größe zwei ausgeführt werden usw.

Die Verteilungen in Tabelle 5.3 basieren auf dem Frontbaum des Orderings $(\mathrm{AMMF}, \mathrm{ND})$ und die in Tabelle 5.4 auf dem Frontbaum des Orderings $(\mathrm{AMMF}, \mathrm{AMMF})$. Um im Falle von $(\mathrm{AMMF}, \mathrm{ND})$ einen möglichst gut balancierten Frontbaum zu erhalten, wurde der Parameter τ in der Zielfunktion (4.16) von 0.50 auf 0.05 abgesenkt. Damit wird das Gewicht eines Separators bereits dann durch einen Strafterm erhöht, wenn die Gewichte der Teilgraphen um mehr als fünf Prozent differieren. Bei Verwendung des Orderings $(\mathrm{AMMF}, \mathrm{ND})$ sind zur Berechnung des Cholesky-Faktors von BCSSTK30 $1183 \cdot 10^6$ arithmetische Operationen notwendig, bei Verwendung des Orderings $(\mathrm{AMMF}, \mathrm{AMMF})$ lediglich $702 \cdot 10^6$.

Betrachten wir zunächst die Faktorisierung der Matrix BCSSTK30 auf einem Hypercube der Dimension eins. Ein Vergleich des ersten Abschnitts der Tabelle 5.3 mit dem ersten Abschnitt der Tabelle 5.4 zeigt, daß bei Verwendung des Orderings $(\mathrm{AMMF}, \mathrm{AMMF})$ wesentlich mehr Operationen den Prozessoren $\mathrm{myprocID} = 0$ und $\mathrm{myprocID} = 1$ zur gemeinsamen Bearbeitung zugeordnet werden als bei Verwendung des Nested-Dissection-Orderings $(\mathrm{AMMF}, \mathrm{ND})$. Während im Falle von $(\mathrm{AMMF}, \mathrm{ND})$ lediglich $22 \cdot 10^6$ Operationen von beiden Prozessoren gemeinsam ausgeführt werden müssen, sind es im Falle von $(\mathrm{AMMF}, \mathrm{AMMF})$ $79 \cdot 10^6$. Bei der Faktorisierung der Matrix auf einem höherdimensionalen Hypercube müssen die $79 \cdot 10^6$ Operationen auf mehr Prozessoren verteilt werden, wodurch sich der Kommunikations-Overhead stärker erhöht als im Falle von $(\mathrm{AMMF}, \mathrm{ND})$.

Die nächsten Abschnitte der Tabellen 5.3 und 5.4 zeigen jedoch, daß hierfür ein hoher Preis gezahlt werden muß. Bereits an dem Beispiel des $h \times n$-Gitter aus Abbildung 4.4 haben wir gesehen, daß in einem Graphen mit großem Aspekt-Ratio Elemente mit einem großen Rand entstehen, wenn die Separatoren entsprechend ihrer Rekursionstiefe eliminiert werden. Jedes dieser Elemente stellt eine große Clique dar, deren Faktorisierung sehr aufwendig ist. Bei der Faktorisierung der Matrix BCSSTK30 auf einem zweidimensionalen Hypercube kommt es daher zu einer starken Belastung der zweielementigen Prozessorgruppen. Ein Vergleich des zweiten Abschnitts der Tabellen 5.3 und 5.4 zeigt, daß im Falle von $(\mathrm{AMMF}, \mathrm{ND})$ die zweielementigen Pro-

zessorgruppen $128 \cdot 10^6$ bzw. $145 \cdot 10^6$ arithmetische Operationen durchführen müssen, während bei Verwendung des Orderings $(\mathrm{AMMF}, \mathrm{AMMF})$ den Prozessorgruppen nur $113 \cdot 10^6$ bzw. $68 \cdot 10^6$ Operationen zugewiesen werden. Auch in den nächsten Abschnitten ist die einer zweielementigen Prozessorgruppe zugewiesene Anzahl arithmetischer Operationen immer höher, wenn das Ordering $(\mathrm{AMMF}, \mathrm{ND})$ benutzt wird. Man beachte, daß es im Falle von $(\mathrm{AMMF}, \mathrm{AMMF})$ sogar Prozessorgruppen gibt, denen gar keine Operationen zugewiesen werden. In diesem Fall erhöht sich die Anzahl der einer kleineren Prozessorgruppe zugewiesenen Operation, was zu einer Reduzierung des Kommunikations-Overheads führt.

Insgesamt läßt sich also feststellen, daß bei der Faktorisierung von Graphen mit großem Aspekt-Ratio die Verwendung eines Nested-Dissection-Orderings allein aus Balancegründen nicht ratsam ist. Zwar kann der Frontbaum an der Wurzel leichter aufgespalten werden, bedingt durch die Numerierung der Separatoren erhöht sich jedoch der Aufwand zur Berechnung von L erheblich. Dies verdeutlichen noch einmal die Tabellen 5.5 (a) und 5.5 (b). Die Tabellen zeigen die Laufzeiten zur Durchführung der numerischen Faktorisierung in Abhängigkeit von dem gewählten Ordering. Jede Tabelle ist wieder in vier Abschnitte unterteilt. Die Binärbäume in den Abschnitten zeigen die von den Prozessorgruppen benötigte Zeit zur Durchführung der zugeteilten arithmetischen Operationen. In Klammern ist jeweils der von einer Prozessorgruppe produzierte Kommunikations-Overhead angegeben. Die letzte Spalte einer Tabelle zeigt die insgesamt zur Berechnung des Cholesky-Faktors benötigte Zeit. Die Gesamtzeit erhält man durch Aufsummieren der auf dem *kritischen Pfad* liegenden Zeiten. In den Binärbäumen sind die Zeitangaben auf den kritischen Pfaden durch Unterstreichen hervorgehoben.

Ein Vergleich der Gesamtzeiten zeigt, daß bei Verwendung des Orderings $(\mathrm{AMMF}, \mathrm{AMMF})$ trotz eines unbalancierten Frontbaumes auf einem $r-1$-dimensionalen Hypercube die gleiche Performanz erzielt werden kann wie im Falle von $(\mathrm{AMMF}, \mathrm{ND})$ auf einem r-dimensionalen Hypercube.

Betrachten wir nun die Faktorisierung der Matrix MAT02HBF. Wir haben diese Matrix ausgewählt, um auf eine weitere Unzulänglichkeit im Zusammenspiel zwischen Ordering- und Mapping-Verfahren hinzuweisen. Auch hier wurde im Falle von $(\mathrm{AMMF}, \mathrm{ND})$ der Parameter τ von 0.50 auf 0.05 abgesenkt. Ein Vergleich des ersten Abschnitts der Tabellen 5.6 und 5.7 zeigt jedoch, daß bei Verwendung des Nested-Dissection-Orderings $(\mathrm{AMMF}, \mathrm{ND})$ die Aufspaltung des Frontbaumes $\mathcal{T}$ an der Wurzel sehr viel schwieriger ist als im Falle von $(\mathrm{AMMF}, \mathrm{AMMF})$. Dies liegt daran, daß die Balance einer Partitionierung anhand der Gewichte der Teilgraphen bewertet wird. Das Gewicht eines Teilgraphen sagt jedoch gar nichts darüber aus, wieviele Operationen zur Faktorisierung der Knoten notwendig sind, und genau diese Zahl ist entscheidend für die Aufspaltung des Frontbaumes. Zum Zeitpunkt der Konstruktion eines Knotenseparators existiert noch kein Ordering für die Knoten in den Teilgraphen. Daher ist auch die zur Faktorisierung eines Teilgraphen benötigte Anzahl an arithmetischen Operationen nicht bekannt. Die Benutzung der Knotengewichte stellt also nur eine heuristische Schätzung dar, die nicht sehr genau ist.

Ein wesentlich besseres Zusammenspiel zwischen Ordering- und Mapping-Verfahren ermöglicht das in Abschnitt 4.2.3 vorgestellte dreistufige Multisection-Verfahren. Innerhalb der for-Schleife in Funktion TRISTAGEMULTISECTION (Zeilen 04–13 in Abbildung 4.14) werden jeweils vollständige Orderings generiert. Als Nebenprodukt liefert jedes dieser Orderings einen Frontbaum $\mathcal{T}$. Da der Aufwand zur Berechnung eines Mappings sehr gering ist, kann auf jeden Frontbaum $\mathcal{T}$ die Funktion SPLIT angewandt werden. Innerhalb der Funktion TRISTAGEMULTISEKTION kann dann dasjenige Ordering ausgewählt werden, das sowohl die Auffüllung von A als auch den bei der parallelen Faktorisierung entstehenden Kommunikations-Overhead minimiert. Hierzu ist lediglich die Formulierung einer geeigneten Zielfunktion notwendig.

Die Tabellen 5.8 (a) und 5.8 (b) zeigen die Laufzeiten zur Berechnung des Cholesky-Faktors von MAT02HBF in Abhängigkeit von dem gewählten Ordering. Bedingt durch einen größeren Fill-in und durch die Schwierigkeiten beim Aufspalten des Frontbaumes an der Wurzel, sind die Laufzeiten bei Verwendung des Nested-Dissection-Orderings in allen vier Fällen höher als die Laufzeiten bei Verwendung des $(\mathrm{AMMF}, \mathrm{AMMF})$-Orderings.

Abschließend wollen wir auf eine Schwäche unseres parallelen Faktorisierungsalgorithmus hinweisen. Dazu betrachten wir den ersten Abschnitt der Tabelle 5.6 und den ersten Abschnitt der Tabelle 5.8 (a). Bei der Faktorisierung von MAT02HBF auf einem Hypercube der Dimension eins werden im Falle von $(\mathrm{AMMF}, \mathrm{ND})$ den Prozessoren $\mathrm{myprocID} = 0$ und $\mathrm{myprocID} = 1$ zur gemeinsamen Abarbeitung $426 \cdot 10^6$ Operationen zugewiesen. Exklusiv werden den Prozessoren ungefähr $500 \cdot 10^6$ Operationen zugewiesen. Zur Ausführung der exklusiv zugewiesenen Operationen benötigen die Prozessoren ca. 17 Sekunden. Für die gemeinsame Abarbeitung der $426 \cdot 10^6$ Operationen sollten daher ungefähr 8.5 Sekunden und nicht 14.13 Sekunden benötigt werden. Dieser „Slowdown“ kann nicht dem Kommunikations-Overhead angelastet werden. Er beträgt lediglich 0.90 Sekunden. Vielmehr ist hierfür der Umstand verantwortlich, daß aufgrund von Kommunikationsoperationen und aufgrund des einfachen 1×1-Kerns die Cache- und Registereffizienz im parallelen Fall nicht so hoch ist wie in sequentiellen.

In einer zukünftigen Implementierung sollte daher der 1×1-Kern durch einen 2×2-Kern ersetzt werden. Dieser Kern ist auch mit dem block-zyklischen Mapping verträglich. Darüber hinaus haben wir in Abschnitt 5.1.4 am Beispiel der Matrix CFD1 gesehen, daß sich die größte Beschleunigung beim Übergang von einem 1×1-Kern zu einem 2×2-Kern ergibt.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
588	573														
22															
229	231	215	213												
128		145													
	22														
90	90	70	66	70	74	80	83								
49		95		71		49									
	128				145										
			22												
22	22	26	26	15	15	24	23	19	20	28	29	27	27	18	18
46		38		39		19		32		17		26		48	
	49				95				71				49		
			128								145				
							22								

Tab. 5.3: Verteilung der zur Faktorisierung von BCSSTK30 durchzuführenden Operationen (in Mio.) auf die Prozessoren des Hypercubes. Die Verteilung basiert auf dem Ordering $(\mathrm{AMMF}, \mathrm{ND})$. Insgesamt müssen $1183 \cdot 10^6$ Operationen verteilt werden.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
311	312														
79															
99	99	122	122												
113		68													
	79														
30	29	49	50	44	44	61	61								
40		0		34		0									
	113				68										
			79												
11	11	15	14	17	17	25	25	18	18	15	14	23	22	31	30
7		0		14		0		8		15		16		0	
	40				0				34				0		
			113								68				
							79								

Tab. 5.4: Verteilung der zur Faktorisierung von BCSSTK30 durchzuführenden Operationen (in Mio.) auf die Prozessoren des Hypercubes. Die Verteilung basiert auf dem Ordering $(\mathrm{AMMF}, \mathrm{AMMF})$. Insgesamt müssen $702 \cdot 10^6$ Operationen verteilt werden.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	total
20.00	19.47															20.76
0.76 (0.10)																
7.61	7.70	7.50	7.35													12.83
4.38 (0.23)		4.90 (0.33)														
	0.43 (0.11)															
2.99	3.01	2.38	2.32	2.55	2.60	2.85	2.80									7.7
1.61 (0.14)		2.69 (0.19)		2.31 (0.15)		1.63 (0.13)										
	2.29 (0.19)				2.50 (0.28)											
			0.29 (0.12)													
0.83	0.82	0.90	0.93	0.55	0.57	0.78	0.76	0.74	0.75	1.01	1.05	0.99	0.96	0.69	0.69	4.8
1.57 (0.21)		1.25 (0.10)		1.25 (0.10)		0.63 (0.06)		1.04 (0.07)		0.56 (0.05)		0.83 (0.07)		1.58 (0.16)		
	0.87 (0.12)				1.50 (0.16)				1.21 (0.15)				0.86 (0.13)			
			1.30 (0.31)								1.50 (0.34)					
							0.18 (0.15)									

Ordering: (AMMF,ND), $\theta(L) = 1183 \times 10^6$.

(a)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	total
9.89	10.03															12.66
2.63 (0.15)																
3.14	3.18	3.84	3.87													7.7
3.08 (0.29)		2.20 (0.18)														
	1.44 (0.16)															
1.11	1.14	1.80	1.88	1.69	1.69	2.27	2.25									4.75
1.04 (0.10)		0.00 (0.00)		0.88 (0.13)		0.00 (0.00)										
	1.77 (0.25)				1.22 (0.19)											
			0.80 (0.20)													
0.40	0.39	0.56	0.58	0.63	0.62	0.92	0.96	0.70	0.73	0.57	0.56	0.75	0.74	1.26	1.21	2.75
0.22 (0.05)		0.00 (0.00)		0.48 (0.04)		0.00 (0.00)		0.26 (0.04)		0.50 (0.04)		0.52 (0.05)		0.00 (0.00)		
	0.55 (0.10)				0.00 (0.00)				0.48 (0.11)				0.00 (0.00)			
			1.09 (0.36)								0.79 (0.25)					
							0.49 (0.20)									

Ordering: (AMMF,AMMF), $\theta(L) = 702 \times 10^6$.

(b)

Tab. 5.5: Laufzeiten (in Sek.) zur Berechnung des Cholesky-Faktors von BCSSTK30 in Abhängigkeit von dem gewählten Ordering.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
497	506														
426															
253	244	258	248												
0		0													
426															
61	63	95	97	100	102	67	68								
129		52		56		113									
0				0											
426															
31	30	28	27	23	22	36	37	38	38	30	31	20	20	34	34
0		8		50		23		24		41		27		0	
129				52				56				113			
0										0					
426															

Tab. 5.6: Verteilung der zur Faktorisierung von MAT02HBF durchzuführenden Operationen (in Mio.) auf die Prozessoren des Hypercubes. Die Verteilung basiert auf dem Ordering $(\mathrm{AMMF}, \mathrm{ND})$. Insgesamt müssen $1429 \cdot 10^6$ Operationen verteilt werden.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
439	446														
203															
180	180	172	173												
79		100													
203															
78	78	82	82	71	69	74	74								
24		16		32		25									
79				100											
203															
22	22	26	27	25	24	32	33	22	22	34	35	26	26	15	15
34		25		32		17		27		0		22		44	
24				16				32					25		
79											100				
203															

Tab. 5.7: Verteilung der zur Faktorisierung von MAT02HBF durchzuführenden Operationen (in Mio.) auf die Prozessoren des Hypercubes. Die Verteilung basiert auf dem Ordering $(\mathrm{AMMF}, \mathrm{AMMF})$. Insgesamt müssen $1088 \cdot 10^6$ Operationen verteilt werden.

(a)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	total
17.12	17.14															31.27
14.13 (0.90)																
8.91	8.21	8.98	8.17													16.22
0.00 (0.00)		0.00 (0.00)														
7.24 (0.76)																
2.25	2.30	3.17	3.20	3.45	3.50	2.35	2.40									10.53
4.28 (0.23)		1.70 (0.11)		1.94 (0.14)		3.70 (0.30)										
0.00 (0.00)				0.00 (0.00)												
3.95 (0.95)																
1.25	1.25	0.98	0.96	0.75	0.74	1.35	1.37	1.40	1.41	1.23	1.21	0.76	0.74	1.33	1.35	5.86
0.00 (0.00)		0.27 (0.03)		1.63 (0.14)		0.75 (0.05)		0.78 (0.06)		1.27 (0.12)		0.89 (0.06)		0.00 (0.00)		
2.20 (0.25)				0.87 (0.11)				0.93 (0.11)				1.92 (0.28)				
0.00 (0.00)											0.00 (0.00)					
2.29 (0.74)																

Ordering: (AMMF,ND), $\theta(L) = 1429 \times 10^6$.

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	total
14.92	14.83															21.94
7.02 (0.49)																
6.12	6.15	5.71	5.81													12.60
2.59 (0.20)		3.28 (0.26)														
3.51 (0.44)																
2.62	2.68	2.85	2.80	2.42	2.37	2.50	2.55									7.02
0.78 (0.07)		0.54 (0.04)		1.06 (0.09)		0.82 (0.07)										
1.35 (0.20)				1.64 (0.22)												
1.90 (0.58)																
0.78	0.75	1.00	1.00	1.00	0.98	1.32	1.30	0.78	0.73	1.37	1.40	0.94	0.97	0.52	0.51	4.53
1.13 (0.09)		0.85 (0.11)		1.07 (0.09)		0.55 (0.04)		0.89 (0.07)		0.00 (0.00)		0.72 (0.06)		1.46 (0.13)		
0.42 (0.08)				0.29 (0.05)				0.57 (0.08)						0.44 (0.05)		
0.82 (0.23)											0.94 (0.30)					
1.17 (0.41)																

Ordering: (AMMF,AMMF), $\theta(L) = 1088 \times 10^6$.

Tab. 5.8: Laufzeiten (in Sek.) zur Berechnung des Cholesky-Faktors von MAT02HBF in Abhängigkeit von dem gewählten Ordering.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurden Algorithmen zur schnellen Faktorisierung großer, dünn besetzter, positiv definiter Matrizen vorgestellt. Dabei haben wir uns zunächst auf die Berechnung einer guten Pivotreihenfolge konzentriert. Wir entwickelten ein Ordering-Verfahren, in dem die klassischen Bottom-up- und Top-down-Methoden auf eine neue Art und Weise miteinander verknüpft sind. Anschließend wurde ein effizienter Algorithmus zur numerischen Faktorisierung vorgestellt. Der Algorithmus basiert auf der Multifrontal-Methode und benutzt einen numerischen Kern, der auf Blöcken der Größe 3×3 arbeitet. Bei der Parallelisierung des Algorithmus sind wir von einem verteilten System ausgegangen, dessen Verbindungsnetzwerk einem Hypercube entspricht.

Zusammenfassung In Kapitel 3 haben wir am Beispiel des $n \times n$-Gitters gezeigt, daß die Güte eines Orderings ganz entscheidend von der Form der Gebiete abhängt, die im Rahmen des Eliminationsprozesses entstehen. Der Aufwand zur Faktorisierung einer Laplace-Matrix mit 5-Punkte-Operator kann im Vergleich zu Georges Nested-Dissection-Ordering durch eine einfache 45 Grad Drehung der Gitterseparatoren um fast die Hälfte reduziert werden. Die exakte Analyse des verbesserten Nested-Dissection-Verfahrens hat gezeigt, daß hierfür die speziellen isoperimetrischen Eigenschaften des Gitters verantwortlich sind. Werden nämlich die Separatoren um 45 Grad gedreht, so entstehen im Laufe des Eliminationsprozesses keine quadratischen, sondern rautenförmige Gebiete. In einem Gitter mit 5-Punkte-Stern ist das Verhältnis von Inhalt zu Umfang für rautenförmige Gebiete etwa doppelt so groß wie für quadratische Gebiete. Interessanterweise entstehen die rautenförmigen Gebiete auch bei einem Minimum-Degree-Ordering mit Minimum-Deficiency-Tie-Breaking-Strategie.

Im Mittelpunkt des Kapitels 4 stand die Entwickelung eines verbesserten Ordering-Verfahrens für beliebige Graphen. Characteristisch für das Verfahren ist, daß Knotenseparatoren als Ränder eines unvollständigen Bottom-up-Orderings interpretiert werden. Im Umkehrschluß setzen sich also Knotenseparatoren aus Randsegmenten einzelner Gebiete zusammen. Diese Beobachtung motivierte die Entwicklung eines neuartigen Multilevel-Verfahrens. In dem neuen Verfahren wird der ursprüngliche Graph durch eine Folge von Quotientengraphen approximiert. Die

Quotientengraphen entstehen durch einen Eliminationsprozeß, der dem zur Berechnung eines Bottom-up-Orderings sehr ähnlich ist. Jeder Quotientengraph stellt eine Gebietszerlegung dar. Ziel ist es, eine Überdeckung des Graphen mit ungefähr gleich großen Gebieten zu finden, deren Aspekt-Ratio klein ist. Wir haben gezeigt, daß sich dieses Ziel am besten erreichen läßt, wenn zur Bildung der Quotientengraphen eine Minimum-Degree-Heuristik benutzt wird. Jeder Separator eines Quotientengraphen induziert einen Separator des ursprünglichen Graphen. Zur Minimierung der Separatoren benutzen wir die von Ashcraft und Liu vorgestellte Färbungstechnik. Im Gegensatz zum Vertex-Fiduccia-Mattheyses-Algorithmus können mit Hilfe der Färbungstechnik ganze Knotenmengen in einem Austauschschritt verschoben werden. Die Färbungstechnik ist damit sehr viel mächtiger als der Vertex-Fiduccia-Mattheyses-Algorithmus. Trotzdem besitzt die von uns entwickelte iterative Verbesserungsheuristik die gleiche niedrige Laufzeit. Eine zweite Besonderheit unseres Ordering-Verfahrens besteht darin, daß die Knotenseparatoren als ein Gerüst zur Bestimmung mehrerer Bottom-up-Orderings benutzt werden. Diese Vorgehensweise ist motiviert durch den Umstand, daß die Elimination der Knotenseparatoren entsprechend ihrer Rekursionstiefe genau dann vorteilhaft ist, wenn durch die Anordnung der Separatoren Gebiete mit einem kleinen Aspekt-Ratio entstehen. Experimentelle Ergebnisse haben gezeigt, daß mit Hilfe des neuen Ordering-Verfahrens der Aufwand zur Berechnung des Cholesky-Faktors im Vergleich zum besten derzeit bekannten Bottom-up-Algorithmus um durchschnittlich 42 % reduziert werden kann. Eine derartige Verbesserung ist mit state-of-the-art Algorithmen wie METIS, SCOTCH oder SPOOLES nicht zu erreichen.

In Kapitel 5 beschäftigten wir uns mit der symbolischen und numerischen Faktorisierung. Bei dem Design des sequentiellen Algorithmus standen Techniken zur Steigerung der Cache- und Registereffizienz im Vordergrund. Wir haben gezeigt, daß durch die Multifrontal-Methode und die Verwendung eines numerischen Kerns der Größe 3×3 die von modernen Computern bereitgestellte Floating-Point-Leistung sehr effektiv genutzt werden kann. Die Parallelisierung des sequentiellen Faktorisierungsalgorithmus basiert auf dem 2-dimensionalen Mapping-Schema von Gupta et al. [63, 66]. Wir haben exemplarisch an zwei Beispielen gezeigt, daß die von unserem Verfahren berechneten Orderings auch im parallelen Fall zu einer signifikanten Beschleunigung der numerischen Faktorisierung führen.

Ausblick Aufgrund ihrer enormen Bedeutung für die Faktorisierung dünn besetzter, positiv definiter Matrizen steht die Entwicklung verbesserter Ordering-Verfahren im Vordergrund zahlreicher Forschungsaktivitäten. Wir wollen abschließend auf drei offene Fragen näher eingehen:

- Anordnung der Knotenseparatoren

  Wie bereits in Abschnitt 4.2.3 dargelegt, besteht die wesentliche Schwäche eines Top-down-Verfahrens wie Nested-Dissection darin, daß bei der Bestimmung eines Knotenseparators S für einen Graphen G' der aus bereits konstruierten Separatoren bestehende Rand von G' unberücksichtigt bleibt. Am Beispiel des $n \times n$-Gitters mit 5-Punkte-Stern haben wir gesehen, daß die „Geometrie“ eines Graphen durch Gebiete mit kleinem Aspekt-Ratio

überdeckt werden sollte. Ein Nested-Dissection-Ordering erreicht dieses Ziel – wenn überhaupt – nur indirekt über eine Minimierung der Separatoren. Die Lösung des Problems scheint auf den ersten Blick recht einfach zu sein: man muß nur den Rand der Teilgraphen $G(B)$ und $G(B)$ in die Bewertung einer Partitionierung (S, B, W) mit einfließen lassen. Wie dieses effektiv geschehen kann, ist jedoch bis heute ein offenes Problem [4].

- Zusammenspiel zwischen Ordering- und Mapping-Verfahren

  Die parallele Faktorisierung stellt eine weitere Anforderung an den Ordering-Prozeß: Die Struktur eines Frontbaumes sollte so beschaffen sein, daß eine effiziente Verteilung der Fronten auf die Prozessoren möglich ist. Die parallele Ausführungszeit kann mit Hilfe eines geeignet definierten *Task-Graphen* abgeschätzt werden. Ein gutes Ordering sollte dann zusätzlich die Eigenschaft besitzen, daß der aus dem Frontbaum abgeleitete Task-Graph einen möglichst kurzen kritischen Pfad besitzt. Die Frage ist nun, wie diese Anforderung bereits bei der Berechnung eines Orderings berücksichtigt werden kann. Einen ersten Lösungsansatz liefert des dreistufige Multisection-Verfahren. Es besitzt die Eigenschaft, ein Ordering aus mehreren Ordering-Modulen zusammenzusetzen. Bei diesem Zusammensetzen können die spezifischen Anforderungen einer parallelen Faktorisierung berücksichtigt werden. Eine Arbeit zu diesem Thema ist in Vorbereitung. Sie wird im Rahmen eines Minisymposiums auf der zehnten SIAM Konferenz *Parallel Processing for Scientific Computing* vorgestellt.

- Parallele Berechnung eines Orderings

  Eine der größten Herausforderungen im Bereich der direkten Lösungsverfahren besteht in der Fromulierung eines skalierbaren, parallelen Algorithmus, der alle vier Schritte zur Lösung dünn besetzter Gleichungssysteme umfaßt. An dieser Stelle sei insbesondere auf die Arbeiten an den Universitäten von Bordeaux [73] und Minnesota [77] hingewiesen. Während das Programm PASTIX von Henon et al. [73] einen sequentiellen Ordering-Algorithmus benutzt, ist in dem Programm PSPACES von Joshi et al. [77] ein paralleles Nested-Dissection-Verfahren integriert. Kern des Verfahrens ist ein paralleler Multilevel-Algorithmus zur Bestimmung von Kantenseparatoren (vgl. Karypis und Kumar [80]). Typischerweise generieren Nested-Dissection-Verfahren, deren Knotenseparatoren aus Kantenseparatoren abgeleitet werden, einen höheren Fill-in als moderne Bottom-up-Verfahren. Es stellt sich hier die Frage, ob zur effizienten parallelen Berechnung eines Orderings nicht ganz neue Ansätze entwickelt werden müssen.

Alle diese Fragestellungen zeigen, daß die Entwicklung von neuen Ordering-Verfahren auch zukünftig ein interessantes Forschungsgebiet darstellt. In dieser Arbeit haben wir gezeigt, wie bestehende Verfahren durch eine engere Koppelung von Bottom-up und Top-down-Methoden verbessert werden können. Dies gilt sowohl hinsichtlich der sequentiellen als auch hinsichtlich der parallelen Faktorisierung.

Literaturverzeichnis

[1] P.R. Amestoy, T.A. Davis, I.S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., Vol. 17, 886–905, 1996.

[2] C. Ashcraft, *The fan-both family of column-based distributed Cholesky factorization algorithms* in *Graph Theory and Sparse Matrix Computations*, A. George, J.R. Gilbert, J.W.H. Liu (Eds.), The IMA Volumes in Mathematics and its Applications, Vol. 56, 1993.

[3] C. Ashcraft, *Compressed graphs and the minimum degree algorithm*, SIAM J. Sci. Comput., Vol. 16, No. 6, 1404–1411, 1995.

[4] C. Ashcraft, *Sparse Direct Methods, Volume 1: Orderings for Matrices with Symmetric Structure*, Preprint, February, 2000.

[5] C. Ashcraft, S.C. Eisenstat, J.W.H. Liu, A. Sherman, *A comparison of three column based distributed sparse factorization schemes*, Proc. 5th SIAM Conf. on Parallel Processing for Scientific Computing, 1991.

[6] C. Ashcraft, S.C. Eisenstat, J.W.H. Liu, *A fan-in algorithm for distributed sparse numerical factorization*, SIAM J. Sci. Stat. Comput., Vol. 11, No. 3, 593–599, 1990.

[7] C. Ashcraft, R. Grimes, *The influence of relaxed supernode partitions on the multifrontal method*, ACM Trans. Math. Software, Vol. 15, No. 4, 291–309, 1989.

[8] C. Ashcraft, R. Grimes, *SPOOLES: an object-oriented sparse matrix library*, 9th SIAM Conference on Parallel Processing for Scientific Computing, March 1999, San Antonio, Texas.

[9] C. Ashcraft, R. Grimes, J.G. Lewis, B.W. Peyton, H.D. Simon, *Progress in sparse matrix methods for large linear systems on vector supercomputers*, Internat. J. Supercomputer Appl., Vol. 1, 10–29, 1987.

[10] C. Ashcraft, J.W.H. Liu, *A partition improvement algorithm for generalized nested dissection*, Tech. Rep. BCSTECH-94-020, Boeing Computer Services, Seattle, 1994.

[11] C. Ashcraft, J.W.H. Liu, *Generalized nested dissection: Some recent progress*, Mini Symposium 5th SIAM Conference on Applied Linear Algebra, Snowbird, Utah, 1994.

[12] C. Ashcraft, J.W.H. Liu, *Using domain decomposition to find graph bisectors*, BIT J. of Numerical Mathematics 37, 506–534, 1997.

[13] C. Ashcraft, J.W.H. Liu, *Applications of the Dulmage-Mendelsohn decomposition and network flow to graph bisection improvement*, SIAM J. Matrix Anal. Appl., Vol. 19, 325–354, 1998.

[14] C. Ashcraft, J.W.H. Liu, *Robust ordering of sparse matrices using multisection*, SIAM. J. Matrix Anal. Appl., Vol. 19, No. 3, 816–832, 1998.

[15] C. Ashcraft, J.W.H. Liu, S.C. Eisenstat, *Practical extensions of the multisection ordering for sparse matrices*, 6th SIAM Conference on Applied Linear Algebra, Snowbird, Utah, October 29, 1997.

[16] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, Upper Saddle River, NJ, 1993.

[17] S.T. Barnard, H.D. Simon, *A fast multilevel implementation of recursive spectral bisection*, Proc. of 6th SIAM Conf. Parallel Processing for Scientific Computing, 711–718, 1993.

[18] S.T. Barnard, A. Pothen, H.D. Simon, *A spectral algorithm for envelope reduction of sparse matrices*, Num. Lin. Algebra Appl., Vol 2, No. 4, 317–334, 1995.

[19] M.V. Bhat, W.G. Habashi, J.W.H. Liu, V.N. Nguyen, M.F. Peeters, *A note on nested dissection for rectangular grids*, SIAM J. Matrix Anal. Appl., Vol. 14, No. 1, 253–258, 1993.

[20] Benoit, *Note sur une méthode de résolution des équations normales etc. (Procédé du commandant Cholesky)*, Bull. géodésique 3, 67–77, 1924.

[21] P. Berman, G. Schnitger, *On the performance of the minimum degree ordering for Gaussian elimination*, SIAM J. Matrix Anal. Appl., Vol. 11, No. 1, 83–88, 1990.

[22] E.G. Boman, B. Hendrickson, *A multilevel algorithm for reducing the envelope of sparse matrices*, Tech. Rep. SCCM-96-14, Stanford University, 1996.

[23] C.F. Bornstein, B.M. Maggs, G.L. Miller, *Tradeoffs between parallelism and fill in nested dissection*, Proc. of 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 191–200, 1999.

[24] T. Bui, C. Jones, *A heuristic for reducing fill-in in sparse matrix factorization*, Proc. 6th SIAM Conference on Parallel Processing for Scientific Computing, 445–452, 1993.

[25] I.A. Cavers, *Using deficiency measure for tiebreaking the minimum degree algorithm*, Tech. Rep. 89-2, Dept. of Computer Science, Univ. of British Columbia, Vancouver, 1989.

[26] E. Cuthill, J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, Proc. of 24th Nat. Conf. of the ACM, 157–172, 1969.

[27] A.C. Damhaug, *Sparse solution of finite element equations*, PhD Thesis, Department of Structual Engeneering, The Norwegian Institute of Technology, Trondheim, Norway, 1992.

[28] T. Davis, *University of Florida Sparse Matrix Collection*, http://www.cise.ufl.edu/~davis/sparse/, ftp://ftp.cise.ufl.edu/pub/faculty/davis/matrices, NA Digest, Vol. 92, No. 42, October 16, 1994, NA Digest, Vol. 96, No. 28, July 23, 1996, and NA Digest, Vol. 97, No. 23, June 7, 1997.

[29] G.A. Dirac, *On rigid circuit graphs*, Abh. Math. Sem. Univ. Hamburg, 25, 71–76, 1961.

[30] J.J. Dongarra, J. Du Croz, S. Hammarling, I.S. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Software, Vol. 16, No. 1, 1–17, 1990.

[31] J.J. Dongarra, S.C. Eisenstat, *Squeezing the most out of an algorithm in Cray Fortran*, ACM Trans. Math. Software, Vol. 10, 219–230, 1984.

[32] I.S. Duff, *Full matrix techniques in sparse Gaussian elimination* in *Lecture Notes in Math. (912)* G.A. Watson (Ed.), Springer Verlag, New York, 71–84, 1982.

[33] I.S. Duff, A.M. Erisman, J.K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, 1987.

[34] I.S. Duff, R.G. Grimes, J.G. Lewis, *Users' guide for the Harwell-Boeing sparse matrix collection,* Tech. Rep. TR/PA/92/86, Res. and Techn. Division, Boeing Computer Services, Seattle, 1992.

[35] I.S. Duff, J.K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations,* ACM Trans. Math. Software, Vol. 9, 302–325, 1983.

[36] I.S. Duff, J.K. Reid, *The multifrontal solution of unsymmetric sets of linear equations,* SIAM J. Sci. Statist. Comput., Vol. 5, 633–641, 1984.

[37] I.S. Duff, J.K. Reid, *Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems,* ACM Trans. Math. Softw., 22, 227–257, 1996.

[38] I.S. Duff, J.K. Reid, J.A. Scott, *The use of profile reduction algorithms with a frontal code,* Int. J. Numer. Meth. Engin., Vol. 28, 2555–2568, 1989.

[39] A. Dulmage, N. Mendelsohn, *Coverings of bipartite graphs,* Can. J. Math., Vol. 10, 517–534, 1958.

[40] S.C. Eisenstat, M.H. Schultz, A.H. Sherman, *Algorithms and data structures for sparse symmetric Gaussian elimination,* SIAM J. Sci. Stat. Comput., Vol. 2, No. 2, 225–237, 1981.

[41] S.C. Eisenstat, M.H. Schultz, A.H. Sherman, *Applications of an element model for Gaussian elimination,* in *Sparse Matrix Computations,* J. Bunch, D. Rose (Eds), Academic Press, New York, 85–96, 1976.

[42] C.M. Fiduccia, R.M. Mattheyses, *A linear-time heuristic for improving network partitions,* 19th IEEE Design Automation Conference, 175–181, 1982.

[43] A. Frommer, *Lösung linearer Gleichungssysteme auf Parallelrechnern,* Vieweg Verlag, 1990.

[44] D. Fulkerson, O. Gross, *Incidence matrices and interval graphs,* Pacific J. Math., 15, 835–855, 1965.

[45] K.A. Gallivan, R.J. Plemmons, A.H. Sameh, *Parallel algorithms for dense linear algebra computations,* SIAM Review Vol. 32, No. 1, 54–135, 1990.

[46] G.A. Geist, E. Ng, *Task scheduling for parallel sparse Cholesky factorization,* International Journal of Parallel Programming, Vol. 18, No. 4, 291–314, 1989.

[47] A. George, *Computer implementation of the finite element method,* Tech. Rep. STAN-CS-71-208, Dept. of Computer Science, Stanford University, 1971.

[48] A. George, *Nested dissection of a regular finite element mesh,* SIAM J. Numer. Anal., Vol. 10, No. 2, 345–363, 1973.

[49] A. George, *An automatic one-way dissection algorithm for irregular finite element problems,* SIAM J. Numer. Anal., Vol. 17, No. 6, 740–751, 1980.

[50] A. George, M.T. Heath, J.W.H. Liu, E. Ng, *Sparse Cholesky factorization on a local-memory multiprocessor,* SIAM J. Sci. Stat. Comput., Vol. 9, No. 2, 327–340, 1988.

[51] A. George, J.W.H. Liu, *An automatic nested dissection algorithm for irregular finite element problems,* SIAM J. Numer. Anal., Vol. 15, No. 5, 1053–1069, 1978.

[52] A. George, J.W.H. Liu, *A fast implementation of the minimum degree algorithm using quotient graphs,* ACM Trans. Math. Software, Vol. 6, 337–358, 1980.

[53] A. George, J.W.H. Liu, *Computer Solution of Large Sparse Positive Definite Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[54] A. George, J.W.H. Liu, *The evolution of the minimum degree ordering algorithm,* SIAM Review, Vol. 31, No. 1, 1–19, 1989.

[55] A. George, J.W.H. Liu, E. Ng, *Communication results for parallel sparse Cholesky factorization on a hypercube,* Parallel Computing 10, 287–298, 1989.

[56] A. George, J.W. Poole, R. Voigt, *Incomplete nested dissection for solving n by n grid problems,* SIAM J. Numer. Anal., Vol. 15, 663–673, 1978.

[57] N.E. Gibbs, *Algorithm 509: A hybrid profile reduction algorithm,* ACM Trans. Math. Software, Vol. 2, 378–387, 1976.

[58] N.E. Gibbs, W.G. Poole, P.K. Stockmeyer, *An algorithm for reducing the bandwidth and profile of a sparse matrix,* SIAM J. Numer. Anal., Vol. 13, No. 2, 236–250, 1976.

[59] J.R. Gilbert, C. Moler, R. Schreiber, *Sparse matrices in MATLAB: design and implementation,* SIAM J. Matrix Anal. Appl., Vol. 13, 333–356, 1992.

[60] J.R. Gilbert, R. Schreiber, *Highly parallel sparse Cholesky factorization,* SIAM J. Sci. Stat. Comput., Vol. 13, No. 5, 1151–1172, 1992.

[61] J.R. Gilbert, R.E. Tarjan, *The analysis of a nested dissection algorithm,* Numerische Mathematik, Vol. 50, 377–404, 1987.

[62] T. Goehring, Y. Saad, *Heuristic algorithms for automatic graph partitioning,* Tech. Rep., Dept. of Computer Science, Univ. of Minnesota, 1994.

[63] A. Gupta, *Analysis and design of scalable parallel algorithms for scientific computing,* Ph.D. Thesis, Dept. of Computer Science, Univ. of Minnesota, 1995.

[64] A. Gupta, *WGPP: Watson graph partitioning (and sparse matrix ordering) package, users manual,* IBM T.J. Watson Research Center, Research Report RC 20453, New York, 1996.

[65] A. Gupta, *Fast and effective algorithms for graph partitioning and sparse matrix ordering,* IBM T.J. Watson Research Center, Research Report RC 20496, New York, 1996.

[66] A. Gupta, G. Karypis, V. Kumar, *Highly scalable parallel algorithms for sparse matrix factorization,* IEEE Trans. on Parallel and Distributed Systems, Vol. 8, No. 5, 502–520, 1997,

[67] M.T. Heath, E. Ng, B.W. Peyton, *Parallel algorithms for sparse linear systems,* SIAM Review, Vol. 33, No. 3, 420–460, 1991.

[68] B. Hendrickson, R. Leland, *The CHACO user's guide,* Tech. Rep. SAND94-2692, Sandia Nat. Lab., 1994.

[69] B. Hendrickson, R. Leland, *An improved spectral graph partitioning algorithm for mapping parallel computations,* SIAM J. Sci. Comput., Vol. 16, 1995.

[70] B. Hendrickson, R. Leland, *A multilevel algorithm for partitioning graphs,* Proc. of 7th Supercomputing Conf., 1995.

[71] B. Hendrickson, E. Rothberg, *Effective sparse matrix ordering: just around the BEND,* Proc. of 8th SIAM Conf. Parallel Processing for Scientific Computing, 1997.

[72] B. Hendrickson, E. Rothberg, *Improving the runtime and quality of nested dissection ordering,* SIAM J. Sci. Comput., Vol. 20, No. 2, 468–489, 1998.

[73] P. Henon, P. Ramet, J. Roman, *PaStiX: A parallel sparse direct solver based on a static scheduling for mixed (1D/2D) block distributions*, Proc. Irregular 2000, LNCS 1800, 519–525, 2000.

[74] A.J. Hoffman, M.S. Martin, D.J. Rose, *Complexity bounds for regular finite difference and finite element grids*, SIAM J. Numer. Anal., Vol. 10, No. 2, 364–369, 1973.

[75] J.E. Hopcroft, R.M. Karp, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comp., Vol. 2, 225–231, 1973.

[76] L. Hulbert, E. Zmijewski, *Limiting communication in parallel sparse Cholesky factorization*, SIAM J. Sci. Stat. Comput., Vol. 12, No. 5, 1184–1197, 1991.

[77] M. Joshi, G. Karypis, V. Kumar, A. Gupta, F. Gustavson, *PSPACES: Scalable parallel direct solver library for sparse symmetric positive definite linear systems*, Technical Report, University of Minnesota and IBM Thomas J. Watson Research Center, 1999.

[78] G. Karypis, V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., Vol. 20, No. 1, 1999.

[79] G. Karypis, V. Kumar, *METIS: a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices (Version 4.0)*, Tech. Rep., Dept. of Computer Science, Univ. of Minnesota, 1998.

[80] G. Karypis, V. Kumar, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, J. of Parallel and Distributed Computing, Vol. 48, 71–95, 1998.

[81] B.W. Kernighan, S. Lin, *An effective heuristic procedure for partitioning graphs*, The Bell Systems Technical Journal, 291–308, 1970.

[82] I.P. King, *An automatic reordering scheme for simultaneous equations derived from network problems*, Int. J. Numer. Meth. Engin., Vol. 2, 523–533, 1970.

[83] D.E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison Wesley, 1973.

[84] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin Cummings Publishing Company, Redwood City, CA, 1994.

[85] G. Kumfert, A. Pothen, *Two improved algorithms for envelope and wavefront reduction*, to appear in the BIT J. of Numerical Mathematics, 1996.

[86] D. König, *Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre*, Math. Ann., 77, 453–465, 1916.

[87] F.T. Leighton, *Einführung in Parallele Algorithmen und Architekturen: Gitter, Bäume und Hypercubes*, Übers. aus dem Amerikan. von B. Monien, M. Röttger und U.-P. Schroeder, Internat. Thomson Publ., 1997.

[88] C.E. Leiserson, J.G. Lewis, *Ordering for parallel sparse symmetric factorization*, in *Parallel Processing for Scientific Computing*, SIAM, Philadelphia, 27–31, 1989.

[89] R.J. Lipton, R.E. Tarjan, *A separator theorem for planar graphs*, SIAM J. Appl. Math., Vol. 36, 177–189, 1979.

[90] R.J. Lipton, D.J. Rose, R.E. Tarjan, *Generalized nested dissection*, SIAM J. Numer. Anal., Vol. 16, No. 2, 346–358, 1979.

[91] J.W.H. Liu, *Modification of the minimum-degree algorithm by multiple elimination,* ACM Trans. Math. Software, Vol. 11, No. 2, 141–153, 1985.

[92] J.W.H. Liu, *On the storage requirement in the out-of-core multifrontal method for sparse factorization,* ACM Trans. Math. Software, Vol. 12, 249–264, 1986.

[93] J.W.H. Liu, *The minimum degree ordering with constraints,* SIAM J. Sci. Stat. Comput., Vol. 10, No. 6, 1136–1145, 1989.

[94] J.W.H. Liu, *A graph partitioning algorithm by node separators,* ACM Trans. Math. Software, Vol. 15, No. 3, 198–219, 1989.

[95] J.W.H. Liu, *The multifrontal method and paging in sparse Cholesky factorization,* ACM Trans. Math. Software, Vol. 15, 310–325, 1989.

[96] J.W.H. Liu, *The role of elimination trees in sparse factorization,* SIAM J. Matrix Anal. Appl., Vol. 11, No. 1, 134–172, 1990.

[97] J.W.H. Liu, *The multifrontal method for sparse matrix solutions: theory and practice,* SIAM Review, Vol. 34, No. 1, 82–109, 1992.

[98] J.W.H. Liu, E.G. Ng, B.W. Peyton, *On finding supernodes for sparse matrix computations,* SIAM J. Matrix Anal. Appl., Vol. 14, No. 1, 242–252, 1993.

[99] J.W.H. Liu, A.H. Sherman, *Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices,* SIAM J. Numer. Anal., Vol. 13, 198–213, 1976.

[100] R. Lucas, T. Blank, J. Tiemann, *A parallel solution method for large sparse systems of equations,* IEEE Trans. Computer-Aided Design, CAD-6, 981–991, 1987.

[101] C.L. Lawson, R.J. Hanson, D.R. Kincaid, F.T. Krogh, *Basic linear algebra subprograms for FORTRAN usage,* ACM Trans. Math. Software, Vol. 5, No. 3, 308–323, 1979.

[102] H.M. Markowitz, *The elimination form of the inverse and its application to linear programming,* Management Science, Vol. 3, 255–269, 1957.

[103] C. Meszaros, *The inexact minimum local fill-in ordering algorithm,* Tech. Rep. WP 95 7, Computer and Automation Research Institute, Hungarian Academy of Sciences, Budapest, 1995.

[104] B. Monien, R. Preis, R. Diekmann, *Quality matching and local improvement for multilevel graph-partitioning,* to appear in Special Issue of 'Parallel Computing', 2000.

[105] E. Ng, B.W. Peyton, *Block sparse Cholesky algorithms on advanced uniprocessor computers,* SIAM J. Sci. Comput., Vol. 14, 1034–1056, 1993.

[106] G.H. Paulino, I.F.M. Menezes, M. Gattass, S. Mukherjee, *Node and element resequencing using the laplacian of a finite element graph,* Int. J. Num. Meth. Engin., Vol. 37, 1511–1530, 1994.

[107] C.H. Papadimitriou, *The NP-completeness of the bandwidth minimization problem,* Computing 16, 263–270, 1976.

[108] S.V. Parter, *The use of linear graphs in Gauss elimination,* SIAM Review, Vol. 3, 119–130, 1961.

[109] F. Pellegrini, *SCOTCH and LibSCOTCH 3.3 User's Guide,* Tech. Rep., LaBRI, UMR CNRS 5800, Universite Bordeaux I, 1999.

[110] F. Pellegrini, J. Roman, *SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs,* Proc. of HPCN'96, Brussels, LNCS 1067, 493–498, 1996.

[111] F. Pellegrini, J. Roman, P. Amestoy, *Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering,* Proc. Irregular'99, LNCS 1586, 986–995, 1999.

[112] A. Pothen, C.-J. Fan, *Computing the block triangular form of a sparse matrix,* ACM Trans. Math. Software, Vol. 16, No. 4, 303–324, 1990.

[113] A. Pothen, H.D. Simon, K.-P. Liou, *Partitioning sparse matrices with eigenvectors of graphs,* SIAM J. Matrix Anal. Appl., Vol. 11, No. 3, 430–452, 1990.

[114] A. Pothen, C. Sun, *A mapping algorithm for parallel sparse Cholesky factorization,* SIAM J. Sci. Comput., Vol. 14, No. 5, 1253–1257, 1993.

[115] R. Preis, R. Diekmann, *The PARTY partitioning library user guide – version 1.1,* Tech. Rep., Dept. of Computer Science, Univ. of Paderborn, 1996.

[116] P. Raghavan, *Parallel ordering using edge contraction,* Tech. Rep. CS-95-293, Dept. of Computer Science, Univ. of Tennessee, 1995.

[117] J.K. Reid, *TREESOLVE, a Fortran package for solving large sets of linear finite element equations,* Tech. Rep. CSS 155, Computer Science and Systems Division, AFRE Harwell, 1984

[118] D.J. Rose, *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations,* in *Graph-Teory and Computing,* R. Read (Ed.), Academic Press, New York, 183–217, 1972.

[119] D.J. Rose, R.E. Tarjan, G.S. Luecker, *Algorithmic aspects of vertex elimination on graphs,* SIAM J. Comput., Vol. 5, No. 2, 266–283, 1976.

[120] D.J. Rose, G.F. Whitten, *A recursive analysis of dissection strategies,* in *Sparse Matrix Computations,* J. Bunch, D. Rose (Eds.), Academic Press, New York, 59–84, 1976.

[121] E. Rothberg, *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization,* Ph. D. thesis, Stanford University, 1993.

[122] E. Rothberg, *Robust ordering of sparse matrices: a minimum degree, nested dissection hybrid,* Silicon Graphics manuscript, 1995.

[123] E. Rothberg, *Exploring the tradeoff between imbalance and separator size in nested dissection ordering,* Silicon Graphics manuscript, 1996.

[124] E. Rothberg, *Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and PARAGON multicomputers,* SIAM J. Sci. Comput., Vol. 17, No. 3, 699–713, 1996.

[125] E. Rothberg, S.C. Eisenstat, *Node selection strategies for bottom-up sparse matrix ordering,* SIAM J. Matrix Anal. Appl., Vol. 19, No. 3, 682–695, 1998.

[126] E. Rothberg, A. Gupta, *An efficient block-oriented approach to parallel sparse Cholesky factorization,* SIAM J. Sci. Comput., Vol. 15, No. 6, 1413–1439, 1994.

[127] R. Schreiber, *A new implementation of sparse Gaussian elimination,* ACM Trans. Math. Software, Vol. 8, 256–276, 1982.

[128] R. Schreiber, *Scalability of sparse direct solvers*, in *Sparse Matrix Computations: Graph Theory Issues and Algorithms*, J.R. Gilbert, J.W.H. Liu (Eds.), Springer Verlag, 1992.

[129] J. Schulze, R. Diekmann, R. Preis, *Comparing nested dissection orderings for parallel sparse matrix factorization*, Int. Conf. on Par. and Distr. Processing Techn. and Appl. (PDPTA'95), H.R. Arabnia (ed.), CSREA-Press, 280-289, 1995.

[130] J. Schulze, *Parallel sparse Cholesky factorization*, in *Multiscale Phenomena and Their Simulation*, F. Karsch, B. Monien, H. Satz (Eds.), World Scientific, 1997.

[131] J. Schulze, *PORD: A software library for computing fill-reducing orderings of sparse positive definite matrices (Version 1.2)*, User's manual, Dep. of Computer Science, Univ. of Paderborn, 1999.

[132] J. Schulze, *Towards a tighter coupling of bottom-up and top-down sparse matrix ordering schemes*, to appear in the BIT J. of Numerical Mathematics, 2000.

[133] J. Schulze, *A new multilevel scheme for the construction of vertex separators*, Mini Symposium 7th SIAM Conference on Applied Linear Algebra, Raleigh, North Carolina, 2000.

[134] H.R. Schwarz, *Numerische Mathematik*, Teubner Verlag, 4. Aufl., 1997.

[135] S.W. Sloan, *An algorithm for profile and wavefront reduction of sparse matrices*, Intl. J. Num. Meth. Eng., Vol. 23, 239–251, 1986.

[136] B. Speelpenning, *The generalized element model*, Tech. Rep. UIUCDCS-R-78-946, Dept. of Computer Science, Univ. of Illinois, 1978.

[137] J. Stoer, *Numerische Mathematik 1*, Springer Verlag, 7. Aufl., 1994.

[138] J. Stoer, R. Bulirsch, *Numerische Mathematik 2*, Springer Verlag, 3. Aufl., 1990.

[139] W.F. Tinney, J.W. Walker, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, Proc. of the IEEE, Vol. 55, 1801–1809, 1967.

[140] S. Venugopal, V.K. Naik, *Effects of partitioning and scheduling sparse matrix factorization on communication and load balance*, Proc. of 3rd Supercomputing Conf., 866–875, 1991.

[141] J.H. Wilkinson, *The algebraic eigenvalue problem*, Monographs on Numerical Analysis, Oxford, Clarendon Press, 1965.

[142] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM J. Alg. Disc. Meth., Vol. 2, No. 1, 77–79, 1981.

[143] E. Zeidler (Hrsg.), *Teubner-Taschenbuch der Mathematik*, begr. von I.N. Bronstein und K.A. Semendjajew. Weitergeführt von G. Grosche, V. Ziegler und D. Ziegler, Teubner, 1996.