FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Einbindung des Sparse Matrix Solvers MUMPS in das Jacobi-Davidson-Verfahren

Nils Gröblinghoff

FZJ-ZAM-IB-2007-04

Mai 2007

(letzte Änderung: 22. 5. 2007)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

In unterschiedlichen Bereichen der Natur- und Ingenieurwissenschaften treten immer wieder Probleme auf, die die Berechnung von Eigenwerten großer und dünnbesetzter Matrizen erfordern. Häufig sind dabei nicht alle Eigenwerte einer Matrix von Interesse, sondern nur die größten oder kleinsten Eigenwerte, beziehungsweise die Eigenwerte um einen bestimmten Wert herum.
Projektionsverfahren, wie zum Beispiel das Jacobi-Davidson-Verfahren, stellen genau diese Möglichkeit zur Verfügung. Dazu wird das gegebene Eigenwertproblem zunächst in einen Unterraum kleinerer Dimension projiziert, in dem daraufhin Eigenwerte und Eigenvektoren bestimmt werden. Diese Eigenwerte und -vektoren sind dann sogenannte Ritzwerte und Ritzvektoren der ursprünglichen Matrix, also Annäherungen an die gesuchte Lösung. Diese Approximationen werden in jedem Durchlauf des Jacobi-Davidson-Algorithmus stetig verbessert. Der Vektor, zur Erweiterung des Unterraums, wird jeweils durch Lösen eines linearen Gleichungssystems, der sogenannten Korrekturgleichung, bestimmt.

Im Rahmen dieser Diplomarbeit wurde zur Lösung der Korrekturgleichung ein weiterer Gleichungssystemlöser zu dem Löser-Modul des bestehenden parallelen Jacobi-Davidson-Programms hinzugefügt. Dieser Löser, MUMPS, ist ein direktes Verfahren zur Lösung linearer Gleichungssysteme und basiert auf der Multifrontal-Methode. Durch die Einbindung der neuen Löser-Routine konnte unter gewissen Voraussetzungen die Berechnung der Eigenwerte beschleunigt werden.

Abstract

In different areas of the natural sciences and engineer's sciences appear over and over again the problems which require the calculation of eigenvalues of large and sparsely populated matrices. Besides, often not all eigenvalues of a matrix, but only the biggest or smallest eigenvalues, or the eigenvalues round a certain value are of interest. Projection procedures, as for example the Jacobi-Davidson-procedure, make available exactly this possibility. First the given eigenvalue problem is projected in a subspace of smaller dimension in which as a result eigenvalues and eigenvectors are determined. These eigenvalues and eigenvectors are so-called ritz values and ritz vectors of the original matrix, so approaches to the solution in request. These approximations are improved in every run of the Jacobi-Davidson-algorithm steadily. The vector, to the extension of the subspace, is determined in each case by solving a linear system of equations, the so-called correction equation.

On the occasion of this dissertation it was added another solver for the solution of the correction equation to the module of solvers of the existing parallel Jacobi-Davidson-program. This solver, MUMPS, is a direct procedure for the solution of systems of linear equations and is based on the multifrontal-method. The calculation of the eigenvalues could be accelerated under certain conditions by the integration of thit new solver.

Kapitel 1

Einleitung

In der Natur- und der Ingenieurwissenschaft müssen an vielen Stellen numerische Berechnungen von Eigenwerten großer, dünnbesetzter Matrizen durchgeführt werden. Sind dabei die Eigenwerte an den Rändern des Spektrums, sprich die größten oder kleinsten von Interesse, so stehen eine Vielzahl von Algorithmen zur Verfügung, um diese zu ermitteln. Schwieriger wird es, wenn die Eigenwerte aus dem Inneren des Spektrums berechnet werden sollen, denn dafür existieren nur wenige geeignete Verfahren. Ein großer Fortschritt auf diesem Gebiet lieferte die Entwicklung der Jacobi-Davidson-Methode mit normalen, beziehungsweise mit harmonischen Ritzwerten. Hiermit lassen sich häufig sehr gute Näherungen für die gewünschten Eigenwerte bestimmen.

Da es wegen der hohen Komplexität der Probleme oftmals nicht möglich ist, die Berechnungen auf einem seriellen Rechner durchzuführen, wurde in den letzten Jahren innerhalb des Forschungszentrums Jülich eine parallele Implementierung des Jacobi-Davidson-Verfahrens vorgenommen, welche daraufhin auf den Supercomputer JUMP portiert wurde.
Um die gesuchten Eigenwerte zu bestimmen, wird innerhalb des Jacobi-Davidson-Algorithmus das Ausgangsproblem zunächst in einen kleineren Unterraum projiziert, da die Eigenpaare des projizierten Problems weitaus einfacher zu berechnen sind. Mit Hilfe dieser berechneten Eigenpaare können anschließend Approximationen für die Eigenpaare des ursprünglichen Problems bestimmt werden. Innerhalb dieses Algorithmus wird der oben beschriebene Unterraum nach und nach erweitert in Abhängigkeit von der Lösung einer sogenannten Korrekturgleichung.
Die Korrekturgleichung entspricht dabei einem linearen Gleichungssystem, zu dessen Lösung bisher übliche, bekannte Verfahren gebraucht wurden, die in Kapitel 2 vorgestellt werden.
Die Lösung der Korrekturgleichung übt dabei einen sehr starken Einfluss auf die Konvergenzgeschwindigkeit des gesamten Jacobi-Davidson-Verfahrens aus, da häufig mit sehr großen, dünnbesetzten Matrizen gerechnet werden muss. Aus diesem Grund ist es überaus wichtig, zur Lösung dieser Korrekturgleichung einen passenden und effektiven Löser zu verwenden.
Im Rahmen dieser Diplomarbeit wurde deshalb zu den bisher auswählbaren Lösern ein weiterer hinzugefügt, der sogenannte Sparse-Matrix-Solver *MUMPS*, der speziell zur Lösung großer Gleichungssysteme mit dünnbesetzten Ausgangsmatrizen entwickelt wurde. Der Name *MUMPS* steht dabei für *MUltifrontal Massively Parallel Solver*, sprich für einen Löser, der auf der Multifrontal-Methode basiert, und speziell für die Berechnung mittels Parallelrechnern ausgelegt ist. In Kapitel 2.3 wird ausführlich auf das Prinzip der Multifrontal-Methode eingegangen.
Im Anschluss daran gibt das Kapitel 3 einen Überblick über das Konzept des Jacobi-Davidson-Algorithmus. Kapitel 4 beschäftigt sich schließlich mit der Einbindung des Multifrontal-Lösers in das bestehende Eigenwertprogramm. Ob sich die Integration des neuen Lösers MUMPS positiv auf die Konvergenzgeschwindigkeit des Jacobi-Davidson-Programms ausgewirkt hat, wird anhand einiger Beispiele im Kapitel 6 diskutiert.

Kapitel 2

Verfahren zur Lösung linearer Gleichungssysteme

In diesem Kapitel werden nun verschiedene Verfahren zur Lösung linearer Gleichungssysteme der Art:

$$Ax = b \quad mit\ A \in \mathbb{R}^{n \times n}\ und\ b \in \mathbb{R}^n \tag{2.1}$$

vorgestellt. Begonnen wird dabei mit klassischen direkten Lösern, wie der LU-Faktorisierung, sowie der Cholesky-Zerlegung. Beide Verfahren beruhen auf der Gaußelimination (siehe auch [16]). Einen Ausblick auf gängige iterative Verfahren gibt der darauffolgende Teil, Kapitel 2.2. Mit diesen Methoden lässt sich die Lösung des oben gezeigten Gleichungssystems zwar nicht direkt bestimmen, aber oftmals gut approximieren. In Kapitel 2.3 wird schließlich die Multifrontal-Methode beschrieben. Im Rahmen dieser Diplomarbeit sollte diese Methode in das Jacobi-Davidson-Programm eingefügt wurde, um dort das Lösen von Gleichungssystemen zu übernehmen.

2.1 Direkte Löser

Im Allgemeinen werden direkte Methoden bei dichtbesetzten oder bandstrukturierten Matrizen A des zu lösenden Gleichungssystems eingesetzt. Ist die Systemmatrix dagegen eher dünnbesetzt und zusätzlich noch unregelmäßig strukturiert, so existieren weitaus effektivere Verfahren. Auf Probleme, die bei direkten Lösern für solche Matrizen auftreten können, wird an späterer Stelle eingegangen. Zwei klassische direkte Verfahren werden nun im Folgenden vorgestellt.

Die klassische LU-Zerlegung
Um mit diesem Verfahren das System $Ax = b$ mit $A \in \mathbb{R}^{n \times n}$ und $b \in \mathbb{R}^n$ zu lösen, muss zunächst für die Matrix A einmal eine Gaußelimination durchgeführt werden. Dadurch erhält man für A folgende Zerlegung:

$$A = LU$$

Es wird dabei vorausgesetzt, dass während der Elimination kein Diagonalelement gleich Null auftritt. Sollte dies dennoch passieren, müssen Zeilen oder Spalten vertauscht werden, dazu aber später. Die Zerlegung der Matrix A wird LU-Zerlegung oder auch LU-Faktorisierung genannt, wobei L eine untere Dreiecksmatrix mit Diagonale 1 und U eine obere Dreiecksmatrix darstellt. Das ursprüngliche Lösungssystem lässt sich nun auch schreiben als:

$$LUx = b$$

Lösungsmethode nach LU-Zerlegung:

- Löse zuerst $Ly = b$ und danach $Ux = y$.
- Dann ist x genau die gesuchte Lösung, denn es gilt:

$$b = Ly = LUx = Ax$$

- Das Lösen von $Ux = y$ erfolgt dabei durch Rückwärtssubstitution, denn U ist obere Dreiecksmatrix.
- Mit L als untere Dreiecksmatrix löst man $Ly = b$ durch Vorwärtssubstitution.

Zum Vergleich ist eine Gegenüberstellung des Aufwands der ursprünglichen Gaußelimination gegenüber dem Aufwand der LU-Zerlegung aufgeführt, wobei der Aufwand gemessen wird anhand der Anzahl von Floating-Point-Operationen:

- $Ax = b$ mittels der gängigen Gaußelimination hat einen Aufwand der Ordnung $\sim \frac{2}{3}n^3$
- L, U zu bestimmen mittels Gaußelimination hat gleiche Ordnung $\sim \frac{2}{3}n^3$
- $LUx = b$ zu bestimmen jedoch nur einen Aufwand der Ordnung $\sim 2n^2$

Anhand dieser Aufstellung ist deutlich zu erkennen, dass man mittels LU-Zerlegung die Lösung für mehrere unterschiedliche rechte Seiten b sehr einfach erhalten kann, da die Matrix A dazu nur einmal in L und U zerlegt werden muss.

Wie oben schon angesprochen gelingt die LU-Zerlegung nur, wenn im Zuge der Elimination keines der Diagonalelemente gleich Null wird. Dies kann jedoch für beliebig reguläre Matrizen nicht vorausgesetzt werden. Abhilfe schafft ein passender Zeilentausch vor der Elimination einer Spalte. Dieser Zeilentausch erfolgt durch Multiplikation mit einer Permutationsmatrix P, wobei PA einen Austausch entsprechender Zeilen und AP einen Austausch entsprechender Spalten hervorruft. Es läßt sich zeigen, dass, wenn $A \in \mathbb{R}^{n \times n}$ regulär ist, es Permutationen P_k mit $k = 1, ..., n-1$ gibt, so dass mit der Permutationsmatix $P = P_{n-1} \cdot ... \cdot P_1$ gilt:

$$PA = LU$$

Dabei ist L wieder eine untere Dreiecksmatrix mit Diagonale 1 und U obere Dreiecksmatrix. Es folgt daraus für die Lösung des linearen Gleichungssystems:

$$\begin{array}{rrcl} & Ax & = & b \\ \Leftrightarrow & PAx & = & Pb \\ \Leftrightarrow & LUx & = & Pb \end{array}$$

Nach der oben beschriebenen Methode hat man nun für beliebig reguläre Matrizen erreicht, dass während der Elimination der Spalten kein Diagonalelement gleich Null ist.
Die sogenannte Spalten-Pivot-Suche geht noch ein wenig weiter. So werden entsprechende Zeilen der Matrix derart getauscht, dass anschließend nicht irgendein Element ungleich Null auf der Diagonalen steht, sondern das betragsgrößte Element der Spalte. Anstelle der Spaltenpivotstrategie mit Zeilentausch kann auch eine Zeilenpivotstrategie mit Spaltentausch durchgeführt werden, beide Strategien erfordern im schlimmsten Fall $\mathcal{O}(n^2)$ zusätzliche Operationen. Ausführlich wird in [15] auf diese Methode eingegangen.

Zu Beginn des Kapitels wurde schon angemerkt, dass direkte Methoden ungeeignet sind, um eine dünnbesetzte Matrix A zu faktorisieren. Der Grund dafür ist der, dass die Verfahren nicht zwischen Null- und Nicht-Null-Einträgen in der Systemmatrix unterscheiden, und somit unabhängig von der Besetzung der Matrix immer die selbe Menge an Operationen durchführen. Das hat zur Folge, dass Berechnungen für Null-Einträge ausgeführt werden, obwohl man solche Berechnungen vermeiden sollte. Schließlich läuft dies darauf hinaus, dass die gefundenen Faktoren L und U in der Regel

nicht dünnbesetzt sind. Dieses Auffüllen der Faktoren bezeichnet man gewöhnlich als „Fill-In"-Effekt. Je größer dabei die zu faktorisierende Matrix A ist, desto schwerwiegender wirkt sich dieser Effekt negativ auf die Effizienz des Verfahrens aus. Je nach Menge des Fill-Ins können zusätzlich Speicherplatzprobleme auftreten, da die Matrizen anschließend nicht mehr dünnbesetzt sind.

Die Cholesky-Zerlegung

Die Cholesky-Zerlegung wird im Folgenden sehr ausführlich beschrieben, da sie die Basis der in Kapitel 2.3 beschriebenen Multifrontal-Methode bildet, die wiederum das Kernstück dieser Diplomarbeit darstellt.

Um mit der Cholesky-Zerlegung ein lineares Gleichungssystem der Form (2.1) lösen zu können, muss A eine symmetrisch positiv definite Matrix sein. Für zusätzliche Informationen bezüglich dieser Methode siehe auch [16].

Die Matrix A kann wegen ihrer Symmetrie wie folgt dargestellt werden:

$$A = A^{(1)} = \begin{pmatrix} a_{1,1} & a_{2,1} & \cdots & a_{n,1} \\ a_{2,1} & & & \\ \vdots & & R^{(1)} & \\ a_{n,1} & & & \end{pmatrix}$$

Da $A^{(1)}$ zusätzlich positiv definit ist, gilt auch $a_{1,1} > 0$, sodass an dieser Stelle auch kein Zeilentausch vorgenommen werden muss. Betrachtet man nun die Gaußelimination für diese Matrix $A^{(1)}$, so wird zunächst durch linksseitiges Multiplizieren einer Matrix L_1 die Elimination der ersten Spalte von $A^{(1)}$ eingeleitet. Daraus ergibt sich:

$$L_1 A^{(1)} = \begin{pmatrix} a_{1,1} & a_{2,1} & \cdots & a_{n,1} \\ 0 & & & \\ \vdots & & R^{(2)} & \\ 0 & & & \end{pmatrix}$$

Wird außerdem mit der Matrix L_1^T von rechts multipliziert, so erhält man aus Symmetriegründen eine Matrix, deren erste Zeile und erste Spalte bis auf das Diagonalelement $a_{1,1}$ eliminiert wurden:

$$A^{(2)} = L_1 A^{(1)} L_1^T = \begin{pmatrix} a_{1,1} & a_{2,1} & \cdots & a_{n,1} \\ 0 & & & \\ \vdots & & R^{(2)} & \\ 0 & & & \end{pmatrix} \begin{pmatrix} 1 & -l_{2,1} & \cdots & -l_{n,1} \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} = \begin{pmatrix} a_{1,1} & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & R^{(2)} & \\ 0 & & & \end{pmatrix}$$

Die Matrix $A^{(2)}$ ist nun wieder eine symmetrische Matrix, denn es gilt:

$$(A^{(2)})^T \;=\; (L_1 A^{(1)} L_1^T)^T \;=\; L_1 {A^{(1)}}^T L_1^T \;=\; L_1 A^{(1)} L_1^T \;=\; A^{(2)}$$

Des Weiteren lässt sich auch zeigen, dass die Matrix $A^{(2)}$ positiv definit ist, denn:

$$< x, A^{(2)} x > = \; x^T L_1 A^{(1)} L_1^T x \;=\; (L_1^T x)^T A^{(1)} (L_1^T x) \;=\; y^T A^{(1)} y \; = < y, A^{(1)} y >$$

Mit $y = L_1^T x$ und L_1^T regulär, folgt aus $x \neq 0$ auch $y \neq 0$ und zusammen mit der Voraussetzung, dass A symmetrisch positiv definit ist, erhält man zuletzt:

$$< x, A^{(2)} x > = < y, A^{(1)} y > > \; 0$$

Mit diesen Erkenntnissen, erfüllt die Matrix $A^{(2)}$ die gleichen Voraussetzungen wie die Matrix $A^{(1)}$, so dass nun in $A^{(2)}$ analog die zweite Zeile und Spalte eliminiert werden können:

$$A^{(3)} \;=\; L_2 A^{(2)} L_2^T = L_2 L_1 A^{(1)} L_1^T L_2^T = \begin{pmatrix} a_{1,1} & 0 & \cdots & \cdots & 0 \\ 0 & a_{2,2}^{(2)} & 0 & \cdots & 0 \\ \vdots & 0 & & & \\ \vdots & \vdots & & R^{(3)} & \\ 0 & 0 & & & \end{pmatrix}$$

Sind gleichermaßen $n-1$ Schritte abgearbeitet, so erhält man folgende Darstellung einer Diagonalmatrix:

$$L_{n-1} \cdots L_1 A^{(1)} L_1^T \cdots L_{n-1}^T = \begin{pmatrix} a_{1,1} & & & \\ & a_{2,2}^{(2)} & & \\ & & \ddots & \\ & & & a_{n,n}^{(n)} \end{pmatrix} := D$$

Die Diagonalmatrix D bildet mit nur positiven Diagonalelementen ebenfalls eine symmetrisch positiv definite Matrix.

Somit lässt sich A nun schreiben als:

$$A = L_1^{-1} \cdots L_{n-1}^{-1} D (L_{n-1}^T)^{-1} \cdots (L_1^T)^{-1}$$

Da für alle regulären Matrizen $M \in \mathbb{R}^{n \times n}$ gilt, dass $(M^T)^{-1} = (M^{-1})^T$, kann man die Matrix rechts von D nun umschreiben zu:

$$(L_{n-1}^T)^{-1} \cdots (L_1^T)^{-1} = (L_{n-1}^{-1})^T \cdots (L_1^{-1})^T = (L_1^{-1} \cdots L_{n-1}^{-1})^T = \tilde{L}^T$$

Zusammengefasst ist nun die Matrix darstellbar durch:

$$A = \tilde{L} D \tilde{L}^T$$

wobei $\tilde{L}$ eine untere Dreiecksmatrix mit 1 auf der Diagonalen repräsentiert. Diese Zerlegung der Matrix A wird auch *rationale Cholesky Zerlegung* genannt.

Eine andere Art der Darstellung wird möglich, wenn die Diagonalmatrix noch mit in die Faktormatrix einfließt. Da die Diagonalmatrix D nur Diagonalelemente größer Null enthält, lässt sie sich folgendermaßen umformen:

$$D = \begin{pmatrix} d_1 & & \\ & \ddots & \\ & & d_n \end{pmatrix} = \begin{pmatrix} \sqrt{d_1} & & \\ & \ddots & \\ & & \sqrt{d_n} \end{pmatrix} \begin{pmatrix} \sqrt{d_1} & & \\ & \ddots & \\ & & \sqrt{d_n} \end{pmatrix} = EE^T$$

Schließlich erhält man für die Matrix A die allgemein bekannte Darstellung der *Cholesky- Faktorisierung* mit L als untere Dreiecksmatrix:

$$A = \tilde{L} E E^T \tilde{L}^T = L L^T \quad , \; L = \tilde{L} E$$

In der **Praxis** wird jedoch meist eine andere Form, die nicht mehr an die LU-Zerlegung erinnert, verwendet. Die Voraussetzung für die Matrix A bleibt identisch, sodass gefordert wird, dass $A \in \mathbb{R}^{n \times n}$ symmetrisch positiv definit mit $A = LL^T$ ist, wobei:

$$L = \begin{pmatrix} l_{1,1} & & & \\ l_{2,1} & l_{2,2} & & \\ \vdots & \vdots & \ddots & \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{pmatrix} = \begin{pmatrix} l_{1,1} & & \\ l^{(2)} & & L^{(2)} \\ & & \end{pmatrix}, \text{ mit}$$

$$l^{(2)} = \begin{pmatrix} l_{2,1} \\ \vdots \\ l_{n,1} \end{pmatrix} \in \mathbb{R}^{n-1} \qquad \text{und} \qquad L^{(2)} = \begin{pmatrix} l_{2,2} & & \\ \vdots & \ddots & \\ l_{n,2} & \cdots & l_{n,n} \end{pmatrix} \in \mathbb{R}^{(n-1)\times(n-1)}$$

Die Matrix A lässt sich also mit der Erkenntnis, dass $A = LL^T$ ist, darstellen als:

$$\begin{pmatrix} a_{1,1} & a_{2,1} & \cdots & a_{n,1} \\ a_{2,1} & & & \\ \vdots & & R^{(1)} & \\ a_{n,1} & & & \end{pmatrix} = LL^T = \begin{pmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & & & \\ \vdots & & L^{(2)} & \\ l_{n,1} & & & \end{pmatrix} \begin{pmatrix} l_{1,1} & l_{2,1} & \cdots & l_{n,1} \\ 0 & & & \\ \vdots & & L^{(2)^T} & \\ 0 & & & \end{pmatrix}$$

$$A = \begin{pmatrix} l_{1,1}^2 & l_{1,1}l_{2,1} & \cdots & l_{1,1}l_{n,1} \\ l_{1,1}l_{2,1} & & & \\ \vdots & & L^{(2)}L^{(2)^T} + l^{(2)}l^{(2)^T} & \\ l_{1,1}l_{n,1} & & & \end{pmatrix}$$

Führt man einen Koeffizientenvergleich der beiden Matrizen durch, lässt sich leicht erkennen, dass für $j = 2, \cdots, n$ Folgendes gilt:

$$\begin{aligned} l_{1,1}^2 = a_{1,1} \quad &\Rightarrow \quad l_{1,1} = \sqrt{a_{1,1}} \\ l_{1,1}l_{j,1} \quad &\Rightarrow \quad l_{j,1} = \frac{a_{j,1}}{\sqrt{a_{1,1}}} \end{aligned}$$

Die erste Spalte der Faktor-Matrix L ist damit bestimmt, jedoch bleibt die Submatrix $L^{(2)}$ noch unbekannt. Da aber

$$\begin{aligned} A^{(2)} &:= L^{(2)}L^{(2)^T} = B^{(2)} - l^{(2)}l^{(2)^T} \\ a_{j,k}^{(2)} &= a_{j,k} - l_{j,1}l_{k,1} \quad \text{für} \quad j,k \geq 2 \end{aligned}$$

gilt, lässt sich nun aus $A^{(2)}$ auf die gleiche Art und Weise die erste Spalte von $L^{(2)}$, also die zweite Spalte von L berechnen. Diese Methode kann schließlich so lange durchgeführt werden, bis die gesamte Matrix L bestimmt ist.

Der p-te Schritt des Verfahren kann nun wie folgt dargestellt werden:
Zur Berechnung der p-ten Spalte von L:

$$\begin{aligned} l_{p,p} &= \sqrt{a_{p,p}^{(p)}} \\ l_{j,p} &= \frac{a_{j,p}^{(p)}}{l_{p,p}} \qquad \text{für} \quad j = p+1, \cdots, n \end{aligned} \tag{2.2}$$

Zur Berechnung der Elemente der noch zu faktorisierenden Matrix $A^{(p+1)}$:

$$\begin{aligned} a_{j,k}^{(p+1)} &= a_{j,k}^{(p)} - l_{j,p}l_{k,p} \qquad \text{für} \quad j = p+1, \cdots, n \\ &\qquad\qquad\quad\ \ k = p+1, \cdots, j \end{aligned} \tag{2.3}$$

Ist die Matrix A schließlich im Ganzen faktorisiert, sodass $A = LL^T$ gilt, kann zur Lösung des Gleichungssystems $Ax = b$ wie gewohnt vorgegangen werden.

$$Ax = LL^T x = b$$

wird wie bei der LU-Zerlegung in zwei Schritten gelöst:

$$Ly = b \qquad \text{durch Vorwärtssubstitution}$$

$$L^T x = y \qquad \text{durch Rückwärtssubstitution}$$

Bezüglich des Aufwands sei hier angemerkt, dass dieser für die Cholesky-Zerlegung in etwa halb so groß ist, wie der Aufwand für die LU-Zerlegung, nämlich $\sim \frac{1}{3}n^3$ Operationen. Der verringerte Aufwand entsteht dadurch, dass nach Voraussetzung die Matrix A symmetrisch positiv definit sein muss. Wegen der Symmetrie sind oberes und unteres Dreieck der Matrix identisch, weshalb nur die Hälfte der Koeffizienten berechnet werden muss.

Zuletzt sei hier noch darauf hingewiesen, dass veschiedene Implementierungen der Cholesky- Faktorisierung existieren. Im Zuge der gerade beschriebenen Methode wurden die Elemente der Faktor-Matrix L spaltenweise bestimmt. Andere Algorithmen erreichen das Ergebnis, indem sie zeilenweise faktorisieren oder aber mit Untermatrizen rechnen. Die Art der Faktorisierung mittels Untermatrizen ist zum Beispiel grundlegend für die in Kapitel 2.3 beschriebene Multifrontal-Methode.

2.2 Iterative Löser

In diesem Kapitel werden nun Methoden vorgestellt, mit denen lineare Gleichungssysteme iterativ gelöst werden können. Diese Verfahren sind, wie anfangs schon erwähnt, besonders geeignet für große, dünnbesetzte Matrizen und werden in technischen Anwendungen oft eingesetzt. Warum dies so ist, soll durch die nachfolgende Gegenüberstellung von direkten und iterativen Lösern verdeutlicht werden.

Wie in dem vorherigen Teil dieses Kapitels zu erkennen war, basierten direkte Verfahren zur Lösung von Gleichungssystemen auf der Faktorisierung der Eingangsmatrix A. Angenommen man hat eine $n \times n$ Matrix zu faktorisieren, so benötigte die LU-Zerlegung $\mathcal{O}(N^3)$ Operationen und einen Speicher von $\mathcal{O}(N^2)$, unabhängig von der Struktur der Eingangsmatrix.

Mittels iterativer Verfahren wird nun keine direkte Lösung mehr für das Problem $Ax = b$ berechnet, sondern lediglich eine angenäherte Lösung $x^{(k)}$. Um eine Folge von Näherungslösungen $\left\{x^{(k)}\right\}$ erzeugen zu können, muss man das jeweilige iterative Verfahren mit einer Anfangsnäherung $x^{(0)}$ von x starten, wobei x die unbekannte Lösung des Problems darstellt. Es werden daraufhin solange Näherungslösungen erzeugt, bis die aktuelle Näherung $x^{(k)}$ nah genug an der exakten Lösung x liegt.

Die Ausführungszeiten von iterativen Verfahren werden dabei dominiert durch den Berechnungsaufwand der benötigten Matrix-Vektor-Multiplikationen. Üblicherweise entstehen bei der Multiplikation von einer Matrix mit einem Vektor $\mathcal{O}(N^2)$ Operationen. Wird jedoch mit einer dünnbesetzten Matrix gerechnet, die zum Beispiel nur $\mathcal{O}(N)$ Nicht-Null-Elemente enthält, so schrumpft die Komplexität der Matrix-Vektor-Produkte auf $\mathcal{O}(N)$. Es lässt sich also erkennen, dass sich der Einsatz von iterativen Verfahren besonders bei der Lösung von Gleichungssystemen mit großen dünnbesetzten Matrizen lohnt. Ein weiterer Vorteil gegenüber direkten Verfahren ist, dass weniger Speicherplatz benötigt wird, um die Matrix abzuspeichern. Da iterative Verfahren lediglich mit den von Null verschiedenen Einträgen rechnen, brauchen auch nur diese abgespeichert werden. Das heißt je schwächer die Matrix besetzt ist, desto weniger Speicherplatz wird auch benötigt.

Zwei grundlegende Parameter ergeben sich bei der Lösung von linearen Gleichungssystemen mittels iterativen Verfahren im k-ten Schritt, und zwar der durch das Verfahren gemachte Fehler e_k und das Residuum r_k:

$$\begin{aligned} e_k &= x - x_k \\ r_k &= b - Ax_k \end{aligned}$$

Zusammengefasst gilt also für das Residuum $r_k = Ae_k$. Da jedoch x zu diesem Zeitpunkt noch unbekannt ist, ist auch e_k nicht berechenbar. Man erwartet jedoch von einem guten iterativen Verfahren, dass der Fehler für steigendes k immer kleiner wird. Ausführlich ist dieser Sachverhalt in [16] diskutiert.

Ein wichtiges Kriterium für die Güte iterativer Methoden zur Lösung von Gleichungssystemen ist deren Konvergenzgeschwindigkeit, das heißt wie schnell die angenäherten Lösungen $x^{(k)}$ gegen die exakte Lösung x konvergieren.

Klassische iterative Verfahren sind das Jacobi- , sowie das Gauß-Seidel-Verfahren, welche spezielle Splitting-Verfahren darstellen und ausführlich in [15] erläutert werden.
In den folgenden Abschnitten sind die grundlegenden Algorithmen zu den CG-artigen Verfahren abgebildet, die bisher Anwendung im Jacobi-Davidson-Algorithmus fanden. Dazu gehören das allgemeine Konjugierte-Gradienten-Verfahren (auch CG-Verfahren genannt), sowie die zwei modifizierten CG-artigen Verfahren, das QMR- (Quasi-Minimal Residual) und das TFQMR-Verfahren (Transpose-Free Quasi-Minimal Residual). Alle drei Methoden gehören dabei der Klasse der Krylov-Unterraum-Verfahren an (vgl. Kapitel A.12).

Das Verfahren der konjugierten Gradienten

Das CG-Verfahren ist ein iteratives Suchverfahren zur Lösung linearer Gleichungssysteme mit symmetrisch positiv definiter Koeffizientenmatrix A. Der eigentliche Algorithmus der CG-Methode basiert auf der Lösung einer quadratischen Optimierungsaufgabe. So wird das ursprüngliche Problem, die Lösung des Gleichungssystems, darauf zurückgeführt, das eindeutige Minimum der Funktion

$$F(x) = \frac{1}{2}x^T Ax - x^T b$$

zu finden. Nach einigen Vorüberlegungen bezüglich Voraussetzungen und Gegebenheiten, die ausführlich in [15] dokumentiert sind, erhält man den in Abbildung 2.1 beschriebenen Algorithmus.
Pro Iterationsschritt werden jeweils eine Näherung des Lösungsvektors $x^{(k)}$, das Residuum $r^{(k)}$, sowie $d^{(k)}$ gebildet, wobei $d^{(k)}$ die Richtung angibt, in der die aktuelle Näherung des Lösungsvektors bestimmt wird.

Es läßt sich zeigen, dass in exakter Arithmetik maximal endlich viele, sprich höchstens n CG-Schritte erforderlich sind, um die exakte Lösung x zu erhalten. Somit kann man das CG-Verfahren bei exakter Arithmetik auch als ein direktes Verfahren ansehen.
In der Praxis wird die exakte Lösung jedoch wegen Rundungsfehlereinflüssen im Allgemeinen nicht erreicht. Man erhält aber schon nach wenigen Iterationsschritten brauchbare Näherungen für die exakte Lösung x.

Wie in Abbildung 2.1 zu erkennen ist, wird die Iteration innerhalb des CG-Algorithmus abgebrochen, sobald ein *Kriterium* erfüllt ist. Was unter diesem Kriterium zu verstehen ist, soll nun verdeutlicht werden. Wünschenswert wäre natürlich ein Abbruchkriterium, wie $\|x - x^{(k)}\|$ „hinreichend klein“. Da dieses Kriterium aufgrund des unbekannten x nicht ausführbar ist, verwendet man in

der Praxis eine vereinfachte Abfrage. Alternativ wird schließlich zu Beginn eines jeden Durchlaufs überprüft, ob

$$\|r^{(k)}\| = \|x - x^{(k)}\|_{A^2} \quad \text{„hinreichend klein“}$$

ist. Eine ausführliche Diskussion bezüglich eines geeigneten Abbruchkriteriums ist auch in [16] nachzulesen. Es sei an dieser Stelle nur kurz darauf hingewiesen, dass die Verwendung der Residuennorm als Abbruchkriterium gewisse Probleme mit sich bringt. So kann es unter Umständen sein, dass die Norm des Residuums recht groß ist, obwohl bereits eine gute Annäherung an die exakte Lösung gefunden wurde. Andererseits kann auch der entgegengesetzte Fall auftreten, dass nämlich die Norm sehr klein wird, obwohl der Iterationsvektor noch weit von der Lösung entfernt ist.

wähle $x^{(0)} \in \mathbb{R}^n$ beliebig
$r^{(0)} = b - Ax^{(0)}$
$d^{(0)} = r^{(0)}$
$k = 0$
Kriterium nicht erfüllt
$\gamma_k = \frac{{r^{(k)}}^T r^{(k)}}{{d^{(k)}}^T A d^{(k)}}$
$x^{(k+1)} = x^{(k)} + \gamma_k d^{(k)}$
$r^{(k+1)} = r^{(k)} + \gamma_k A d^{(k)}$
$\delta_k = \frac{{r^{(k+1)}}^T r^{(k+1)}}{{r^{(k)}}^T r^{(k)}}$
$d^{(k+1)} = -r^{(k+1)} + \delta_k d^{(k)}$
$k = k + 1$

Abbildung 2.1: Algorithmus des CG-Verfahrens

In der Praxis werden im Allgemeinen iterative Lösungsmethoden, wie die CG-artigen Löser, in Verbindung mit einem effizienten Verfahren zur *Vorkonditionierung* verwendet, damit die Konvergenzgeschwindigkeit des Prozesses deutlich postiv beeinflusst wird.

Da unter anderem für das CG-Verfahren Abschätzungen der Konvergenzgeschwindigkeit monoton von der Kondition

$$\kappa_2(A) := \|A\|_2 \, \|A^{-1}\|_2$$

bezüglich der Spektralnorm abhängig sind, muss man einen Weg finden, die Kondition einer Matrix A verringern zu können. Man sucht letztendlich nach einer Möglichkeit das Problem $Ax = b$ so zu transformieren, dass die Matrix A von möglichst kleiner Kondition ist. Wie das funktioniert wird ausführlich in [16] beschrieben. Da diese erwünschte Verbesserung der Kondition von A vor

der Lösung des Gleichungssystems erfolgen soll, ergibt sich der Name für diese Art der Transformation, die sogenannte *Vorkonditionierung*. Das gegebene Problem wird nun derart transformiert, dass nicht mehr $Ax = b$ mit einer symmetrisch positiv definiten Matrix A gelöst wird, sondern ein gleichwertiges Problem, welches mit Hilfe einer beliebigen invertierbaren Matrix B entsteht:

$$\overline{A}\,\overline{x} = b, \quad \overline{x} := B^{-1}x \quad \overline{A} := AB$$

Dabei muss jedoch darauf geachtet werden, dass das Problem seine Symmetrieeigenschaft nicht verliert, damit die iterativen Löser auch weiterhin darauf anwendbar bleiben.

Nach Definition muss die Systemmatrix A eines Gleichungssystems symmterisch positiv definit sein, damit das Verfahren der konjugierten Gradienten anwendbar ist. Sind die Matrizen stattdessen indefinit oder unsymmetrisch, so muss das CG-Verfahren modifiziert werden.
Die beiden CG-artigen Methoden, das QMR-(Quasi-Minimal Residual) und das TFQMR-Verfahren (Transpose-Free Quasi-Minimal Residual), zeichnen sich dadurch aus, dass sie teilweise auch im Fall unsymmetrischer Systeme funktionieren. Garantiert werden kann dies jedoch nicht, da es innerhalb der Verfahren zu Divisionen durch Null, sogenannten Breakdowns, kommen kann, welche zur Folge haben, dass die Berechnung stagniert. Roland W. Freund beschreibt diesen Sachverhalt ausführlich im Bezug auf das QMR-Verfahren in [13].

Auf das QMR- und TFQMR-Verfahren wird nun im Folgenden kurz eingegangen. Beide Methoden beruhen, wie der Name schon verrät, auf dem Ansatz der *quasi-minimalen Residuen*. Ausführliche Beschreibungen der beiden Verfahren, sowie Vergleiche untereinander sind in [13], [14], [20], [21] und [22] zu finden.

Das QMR-Verfahren

Wie schon erwähnt, besitzt das QMR-Verfahren gegenüber dem CG-Verfahren den Vorteil, dass es auch auf unsymmetrische Matrizen anwendbar ist. Dieses erfordert jedoch, dass nicht mehr nur eine Matrix-Vektor-Multiplikation mit der Koeffizientenmatrix A durchgeführt werden muss, sondern zwei. Ein Matrix-Vektor-Produkt wird dabei mit der Ausgangsmatrix A berechnet und eines mit ihrer Transponierten. Die Matrizen, mit denen multipliziert wird, sind dabei fettgedruckt, um diese Operationen im Folgenden ein wenig hervorzuheben.

Die QMR-Methode basiert dabei auf dem klassischen unsymmetrischen Lanczos-Algorithmus (siehe auch [13]), der hier zur Erzeugung der Basisvektoren der Krylov-Räume verwendet wird und auch zum Generieren von oberen Hessenberg-Matrizen dient, die wiederum bei der Berechnung des jeweiligen QMR-Residuums benötigt werden. Kombiniert mit dem Ansatz der quasi-minimalen Residuen ensteht daraus das QMR-Verfahren. Zusätzlich versucht man das Originalsystem $\mathbf{A}x = b$ durch Vorkonditionierung derart zu transformieren, dass ein äquivalentes System, sprich ein System mit derselben Lösung, jedoch mit einem günstigeren Konvergenzverhalten, entsteht. Mit Hilfe einer Matrix $\mathbf{M} = \mathbf{M}_1\mathbf{M}_2$ besitzt das vorkonditionierte System

$$\begin{array}{rlcl} & \mathbf{A}\,x & = & b \\ \Leftrightarrow & \mathbf{A}\,\mathbf{M}_2^{-1}\,(\mathbf{M}_2\,x) & = & b \\ \Leftrightarrow & \mathbf{M}_1^{-1}\mathbf{A}\,\mathbf{M}_2^{-1}\,(\mathbf{M}_2\,x) & = & \mathbf{M}_1^{-1}b \end{array}$$

dieselbe Lösung wie das ursprüngliche System $Ax = b$. Der Algorithmus des QMR-Verfahrens, wie er in Abbildung 2.2 dargestellt ist, enthält dabei die Vorkonditionierung lediglich in Operationen der Form:

$$\begin{array}{lrcl} \text{löse} & \mathbf{M}_1^T\,\tilde{z} & = & z \\ \text{löse} & \mathbf{M}_2\,\tilde{y} & = & y \end{array}$$

wähle $x^{(0)} \in \mathbb{R}^n$ beliebig

$r^{(0)} = \mathbf{A}x^{(0)} - b, \quad \tilde{q}^{(1)} = -r^{(0)}$

Löse $\mathbf{M}_1 y = \tilde{q}^{(1)}$

$\rho_1 = \|y\|_2$

wähle $\tilde{\omega}^{(1)}$ beliebig, z.B. $\tilde{\omega}^{(1)} = -r^{(0)}$

löse $\mathbf{M}_2^T z = \tilde{\omega}^{(1)}$

$\xi_1 = \|z\|_2, \quad \gamma_0 = 1, \quad \eta_0 = -1$

while $x^{(k)}$ nicht konvergiert

$\nu^{(k)} = \frac{\tilde{\nu}^{(k)}}{\rho_k}, \quad y = \frac{y}{\rho_k}, \quad \omega^{(k)} = \frac{\tilde{\omega}(k)}{\xi_k}, \quad z = \frac{z}{\xi_k}, \quad \delta_k = z^T y$

löse $\mathbf{M}_2 \tilde{y} = y$, löse $\mathbf{M}_1^T \tilde{z} = z$

if $k = 1$ **then**

$p^{(k)} = \tilde{y}, \quad q^{(k)} = \tilde{z}$

else if $k > 1$ **then**

$p^{(k)} = \tilde{y} - \frac{\xi_k \delta_k}{\epsilon_{k-1}} p^{(k-1)}, \quad q^{(k)} = \tilde{z} - \frac{\rho_k \delta_k}{\epsilon_{k-1}} \nu^{(k-1)}$

end if

$\tilde{p} = \mathbf{A}p^{(k)}$

$\epsilon_k = {q^{(k)}}^T \tilde{p}, \quad \beta_k = \frac{\epsilon_k}{\delta_k}, \quad \tilde{\nu}^{(k+1)} = \tilde{p} - \beta_k \nu^{(k)}$

löse $\mathbf{M}_1 y = \tilde{\nu}^{(k+1)}$

$\rho_{k+1} = \|y\|_2, \quad \tilde{\omega}^{(k+1)} = \mathbf{A}^T q^{(k)} - \beta_k \omega^{(k)}$

löse $\mathbf{M}_2^T z = \tilde{\omega}^{(k+1)}$

$\xi_{k+1} = \|z\|_2, \quad \vartheta_k = \frac{\rho_{k+1}}{\gamma_{k-1}|\beta_i|}, \quad \gamma_k = \frac{1}{\sqrt{1+\vartheta_k^2}}, \quad \eta_k = \frac{-\eta_{k-1}\rho_k\gamma_k^2}{\beta_k \gamma_{k-1}^2}$

if $k = 1$ **then**

$d^{(k)} = \eta_k p^{(k)}, \quad s^{(k)} = \eta_k \tilde{p}$

else if $k > 1$ **then**

$d^{(k)} = \eta_k p^{(k)} + (\vartheta_{k-1}\gamma_k)^2 d^{(k-1)}, \quad s^{(k)} = \eta_k \tilde{p} + (\vartheta_{k-1}\gamma_k)^2 s^{(k-1)}$

end if

$x^{(k)} = x^{(k-1)} + d^{(k)}, \quad r^{(k)} = r^{(k-1)} + s^{(k)}$

end while

Abbildung 2.2: Der QMR-Algorithmus

Das TFQMR-Verfahren

Problematisch wird die Berechnung mittels QMR-Verfahren auf Parallelrechnern mit verteiltem Speicher, denn dort ist das Transponieren einer Matrix nicht besonders effizient durchführbar. Auf diesen Sachverhalt wird unter anderem in [22] weiter eingegangen.

Das Problem umgeht die TFQMR-Methode, indem keine Matrix-Vektor-Produkte mit der Transponierten der Matrix A berechnet werden müssen. Statt dessen generiert der Algorithmus pro Schritt je zwei Iterierte auf einmal, wodurch alle erreichbaren Suchrichtungen ausgenutzt werden. Wie das genau funktioniert, ist anhand des zugehörigen Algorithmus in Abbildung 2.3 nachvollziehbar.

wähle $x^{(0)} \in \mathbb{R}^n$ beliebig

$r^{(0)} = \mathbf{A}x^{(0)} - b, \quad y^{(1)} = -r^{(0)}$

$\omega^{(1)} = -r^{(0)}, \quad \nu^{(0)} = \mathbf{A}y^{(1)}$

$d^{(0)} = 0, \quad \tau_0 = \|r^{(0)}\|_2$

$\vartheta_0 = 0, \quad \eta_0 = 0$

wähle $\tilde{r}^{(0)}$ so, dass $\rho_0 = -\tilde{r}^{(0)^T} r^{(0)} \neq 0$

while $x^{(s)}$ nicht konvergiert

$\sigma_{k-1} = \tilde{r}^{(0)^T} \nu^{(k-1)}, \quad \alpha_{k-1} = \frac{\rho_{k-1}}{\sigma_{k-1}}$

$y^{(2k)} = y^{(2k-1)} - \alpha_{k-1}\nu^{(k-1)}$

for $s = 2k-1, 2k$

$\omega^{(s+1)} = \omega^{(s)} - \alpha_{k-1}\mathbf{A}y^{(s)}$

$\vartheta_s = \frac{\|\omega^{(s+1)}\|_2}{\tau_{s-1}}, \quad \gamma_s = \frac{1}{\sqrt{1+\vartheta_s^2}}$

$\tau_s = \tau_{s-1}\vartheta_s\gamma_s, \quad \eta_s = \gamma_s^2\alpha_{s-1}$

$d^{(s)} = y^{(s)} + \frac{\vartheta_{s-1}^2\eta_{s-1}}{\alpha_{s-1}} d^{(s-1)}$

$x^{(s)} = x^{(s-1)} + \eta_s d^{(s)}$

end for

$\rho_k = \tilde{r}^{(0)^T} \omega^{(2k+1)}, \quad \beta_k = \frac{\rho_k}{\rho_{k-1}}$

$y^{(2k+1)} = \omega^{(2k+1)} + \beta_k y^{(2k)}$

$\nu^{(k)} = \mathbf{A}y^{(2k+1)} + \beta_k(\mathbf{A}y^{(2k)} + \beta_k\nu^{(k-1)})$

end while

Abbildung 2.3: Der TFQMR-Algorithmus

2.3 Die Multifrontal-Methode

Die Grundidee

Die Entwicklung der Multifrontal-Methode im Jahre 1983 durch Duff und Reid war entscheidend für die Anwendung direkter Methoden zur Lösung großer Gleichungssysteme mit dünnbesetzten Koeffizientenmatrizen. Ursprünglich wurde die Methode zur Faktorisierung großer Maritzen entwickelt, die nicht vollständig in den Hauptspeicher eines Rechners passten. Zur heutigen Zeit wird die Multifrontal-Methode dazu benutzt, die Faktorisierung großer dünnbesetzter Koeffizientenmatrizen effektiv durchzuführen.

Die Idee dabei ist es, die Faktorisierung der Matrix zurückzuführen auf die Faktorisierung vie-

ler kleiner, dafür aber vollbesetzter Matrizen. Für die Faktorisierung solcher sogenannter Frontal-Matrizen kann man dann wiederum Programmiertechniken anwenden, die eigentlich zur Lösung von vollbesetzten Matrizen entwickelt wurden (vgl. Kapitel 2.1). Die Zerlegung solcher vollbesetzter Matrizen ist daraufhin sehr effizient auf Parallelrechnern durchführbar.
Den Begriff „multifrontal“ verwendeten Duff und Reid schließlich für diese Verfahrensart, nachdem sie die Frontal-Methode, die 1970 von Bruce Irons vorgestellt wurde, mittels den oben beschriebenen Überlegungen verallgemeinert hatten.

Die Lösung des linearen Gleichungssystems wird dabei in drei Phasen aufgegliedert:

- Analyse
- Faktorisierung
- Lösung

In der ersten Phase wird zunächst eine *symbolische Faktorisierung* der Eingangsmatrix A durchgeführt mit dem Ziel, die Besetzungsstruktur der Matrix zu ermitteln. Zum einen kann aufgrund der matrixspezifischen Struktur anschließend eine Permutationsmatrix erzeugt werden. Wenn sie auf die Matrix A angewendet wird, so wird das „Fill-In“ in der Faktor-Matrix L verringert. Zum anderen wird in der Analyse-Phase ein sogenannter Eliminationsbaum erstellt, der eine wesentliche Rolle bei der anschließenden Faktorisierungs- und Lösungs-Phase spielt. Die Faktorisierung in dieser ersten Phase wird symbolisch genannt, da die Matrix A lediglich auf der symbolischen Ebene betrachtet wird, im Gegensatz zu der *numerischen Faktorisierung* in der zweiten Phase, wo die eigentliche Berechnung stattfindet.
Die Multifrontal-Methode ist dabei so ausgelegt, dass sie sowohl für symmetrisch positiv definite als auch für unsymmetrische Matrizen geeignet ist. Der entscheidende Unterschied liegt dabei in der zweiten Phase, der Faktorisierung. Hier dient entweder die Cholesky-Zerlegung oder die LU-Zerlegung als Grundlage zur Faktorisierung der Systemmatrix.
Um die symbolische Faktorisierung zu beschleunigen, wird im Falle einer unsymmetrischen Koeffizientenmatrix jedoch nicht A zum Aufbau des Eliminationsbaums verwendet, sondern die stets symmetrische Matrix $M = A + A^T$. Dies ist erlaubt, da der Eliminationsbaum der unsymmetrischen LU-Zerlegung von A identisch ist mit dem der Cholesky-Zerlegung für die symmetrische Matrix M.

Da die Faktorisierung den wichtigsten Teil des Verfahrens darstellt, liegt hier auch der Schwerpunkt dieses Kapitels. Der Einfachheit halber ist jedoch nur das Konzept der Faktorisierung für den symmetrischen Fall dargestellt, so dass als Basis die Cholesky-Faktorisierung dient. Das Verfahren für den symmetrischen, sowie für den unsymmetrischen Fall ist in [5] sehr ausführlich beschrieben.
Zuletzt wird in der dritten Phase das Gleichungssystem mittels Rückwärts- und Vorwärtssubstitution gelöst, vergleiche dazu auch Kapitel 2.1.

Als Grundlage für dieses Kapitel dienten [1], [3], [4], [6] und [7].

Der Bezug zur Cholesky-Faktorisierung

Die Basis für die Multifrontal-Methode beruht auf dem Ansatz der Cholesky-Faktorisierung. Wie in Kapitel 2.1 schon angemerkt, existieren im Grunde drei unterschiedliche Implementierungsarten für die Cholesky-Methode und zwar die zeilenweise Faktorisierung, die spaltenweise und die Faktorisierung mittels Untermatrizen. Wenn man von dieser Einteilung der verschiedenen Implementierungsarten ausgeht, so kann man die Multifrontal-Methode als eine Methode betrachten, die die Cholesky-Faktorisierung mit Untermatrizen durchführt. Die neuartige Eigenschaft der Multifrontal-Methode ist, dass die Update-Auswirkungen einer Faktorspalte bezüglich der verbleibenden Untermatrix zwar berechnet, aber nicht direkt auf die Matrixeinträge angewandt werden. Unter Update-Auswirkungen versteht man dabei die Beiträge aus einem Eliminationsschritt, die für später zu berechnenden Faktorspalten von Bedeutung sind. Man sammelt diese Beiträge zunächst mit de-

nen anderer Faktorspalten in sogenannten Update-Matrizen, bevor das eigentliche aktuelle Update durchgeführt wird.

Da im Folgenden oftmals der Begriff *Vektorprodukt-Update* auftauchen wird, soll nun an dieser Stelle eine kurze Begriffserläuterung stattfinden. Das Vektorprodukt-Update wird hier zunächst an einer vollbesetzten Matrix erklärt. Dafür betrachte man die Cholesky-Zerlegung einer vollbesetzten $n \times n$ Matrix A in LL^T, wobei die oben beschriebene Cholesky-Faktorisierung mittels Untermatrizen verwendet wird. Ein Schritt der Zerlegung kann nun wie folgt dargestellt werden:

$$A = \begin{pmatrix} d & v^T \\ v & C \end{pmatrix} = \begin{pmatrix} \sqrt{d} & 0 \\ \frac{v}{\sqrt{d}} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - \frac{vv^T}{d} \end{pmatrix} \begin{pmatrix} \sqrt{d} & \frac{v^T}{\sqrt{d}} \\ 0 & I \end{pmatrix}$$

Hierbei ist d der erste Diagonaleintrag und v stellt einen $(n-1)$-Vektor dar. Die Submatrix

$$C - \frac{vv^t}{d}$$

bildet den verbleibenden, noch zu faktorisierenden Teil, wobei deren Faktorisierung rekursiv mit derselben Technik durchgeführt wird.

Die nächsten Schritte der Faktorisierung können nun einfach verallgemeinert werden, sodass man eine Blockschreibweise erhält. Angenommen es wurden bereits $j-1$ Faktorisierungsschritte mit $j > 1$ durchgeführt, so ergibt sich folgende Aufteilung der Matrix A, die der Einfachheit halber in einer 2×2 Blockschreibweise beibehalten wurde:

$$A = \begin{pmatrix} B & V^T \\ V & C \end{pmatrix} = \begin{pmatrix} L_B & 0 \\ VL_B^{-T} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - VB^{-1}V^T \end{pmatrix} \begin{pmatrix} L_B^T & L_B^{-T}V^T \\ 0 & I \end{pmatrix}$$

Die Matrix $B = L_B L_B^T$ bildet nun die Cholesky-Faktorisierung der vorherigen $(j-1) \times (j-1)$ Haupt-Untermatrix B. Wie oben auch bildet die Submatrix $C - VB^{-1}V^T$ wieder den noch zu faktorisierenden Teil der Matrix A, wobei hierbei nun deutlich sichtbar ist, dass die $(n-j+1) \times (n-j+1)$ Untermatrix $-VB^{-1}V^T$ die Update-Auswirkungen der ersten $(j-1)$ Zeilen und Spalten bezüglich der Matrix C repräsentiert. Diese Untermatrix lässt sich wie folgt in Termen der ersten $(j-1)$ Spalten des Cholesky-Faktors formulieren :

$$-VB^{-1}V^T = -(VL_B^T)(L_B^{-1}V^T) = \sum_{k=1}^{j-1} \begin{pmatrix} l_{j,k} \\ \vdots \\ l_{n,k} \end{pmatrix} (l_{j,k} \ \cdots \ l_{n,k})$$

Hieran ist nun deutlich erkennbar, dass die Untermatrix $-VB^{-1}V^T$ nun als eine Summe von einfachen Vektorprodukten der zugehörigen Teile der $(j-1)$ Spalten des Cholesky-Faktors geschrieben werden kann. Diese Schreibweise des Vektorprodukt-Updates bildet die Grundlage für die Definitionen der benötigten Frontal- und Update-Matrizen zur Faktorisierung von dünnbesetzten Matrizen.

In der Tat liefert die Multifrontal-Methode eine sehr effektive Möglichkeit, die oben angesprochenen Vektorprodukt-Updates durchzuführen, ganz besonders aber für sehr dünn besetzte Matrizen, wie im Folgenden erkennbar wird.

Die Frontal- und Update-Matrizen

Das Schlüsselkonzept der Multifrontal-Methode beruht auf den Frontal- und Update-Matrizen. Sie werden im weiteren Verlauf definiert durch Summenterme, die durch ausgewählte Teilmengen der Vektorprodukt-Updates gebildet werden. Diese Teilmengen werden des Weiteren beherrscht durch eine Baumstruktur, auf die nun näher eingegangen wird.

Dazu sei A zunächst eine $n \times n$ symmetrisch positiv definite dünnbesetzte Matrix mit einem Cholesky-Faktor L. Der sogenannte *Eliminationsbaum* der Matrix A bildet eine Struktur mit n Knoten derart, dass Knoten p den Vater von j darstellt genau dann, wenn gilt:

$$p = min\{i > j \mid l_{i,j} \neq 0\}$$

Es ist leicht nachvollziehbar, dass dies einen Baum darstellt, wenn die Matrix L in jeder Spalte, ausgenommen in der letzten, einen Eintrag ungleich Null unterhalb der Diagonalen enthält.

Der Index j wird nun im Folgenden für zwei Dinge verwendet, zum einen für die Spalte der Matrix, zum anderen für den entsprechenden Knoten im Eliminationsbaum, der durch die Notation $T(A)$ repräsentiert wird, beziehungsweise T wenn aus dem Kontext der Bezug zur Matrix A eindeutig ist. Diese Baumstruktur wird schließlich bei der Berechnung des Sparse-Matrix Problems von großer Bedeutung sein und dient bei paralleler Verarbeitung dazu die parallelen Berechnungen der darauffolgenden Gauß-Elimination einzuteilen.

Zunächst werden nun zwei Sätze vorgestellt, durch die die Eigenschaften des Eliminationsbaums spezifiziert werden. Mit dem Aufbau einer Faktorspalte ist dabei der Satz der Zeilenindizes von Nicht-Null-Elementen der jeweiligen Spalte gemeint.

Satz 2.3.1. *Falls der Knoten k einen Nachfahren von j im Eliminationsbaum darstellt, dann ist auch der Aufbau des Vektors $(l_{j,k}, \cdots, l_{n,k})$ enthalten in der Struktur $(l_{j,j}, \cdots, l_{n,j})$*

Satz 2.3.2. *Falls $l_{j,k} \neq 0$ und $k < j$ sind, so ist der Knoten k ein Nachkömmling von j im Eliminationsbaum*

Die Notation $T[j]$ soll nun dazu dienen, die Menge aller Nachkommen des Knotens j innerhalb des Eliminationsbaumes T einschließlich j selbst bezeichnen zu können. In Abbildung 2.4 ist nun ein kleines Beispiel einer Sparse-Matrix A abgebildet, sowie deren Cholesky-Faktor L. Dabei stehen die „$\bullet$“ für die Nicht-Null-Einträge der Matrix A und die „$\circ$“ für Einträge, die in der Faktor-Matrix L durch das Faktorisieren dazugekommen sind, die sogenannten „Fill-Ins“.

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{pmatrix} a & & & & & & \bullet & \bullet & \bullet \\ & b & & \bullet & & \bullet & & & \\ & & c & & \bullet & & & \bullet & \\ & \bullet & & d & & & & \bullet & \bullet \\ & & \bullet & & e & \bullet & & \bullet & \\ & \bullet & & & \bullet & f & & & \bullet \\ \bullet & & & & & & g & \bullet & \bullet \\ \bullet & & \bullet & \bullet & \bullet & & \bullet & h & \\ \bullet & & & \bullet & & \bullet & \bullet & & i \end{pmatrix} \qquad L = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{pmatrix} a & & & & & & & & \\ & b & & & & & & & \\ & & c & & & & & & \\ & \bullet & & d & & & & & \\ & & \bullet & & e & & & & \\ & \bullet & & \circ & \bullet & f & & & \\ \bullet & & & & & & g & & \\ \bullet & & \bullet & \bullet & \bullet & \circ & \bullet & h & \\ \bullet & & & \bullet & & \bullet & \bullet & \circ & i \end{pmatrix}$$

Abbildung 2.4: Beispiel für dünnbesetzte Matrix und zugehörige Matrix der Cholesky-Zerlegung

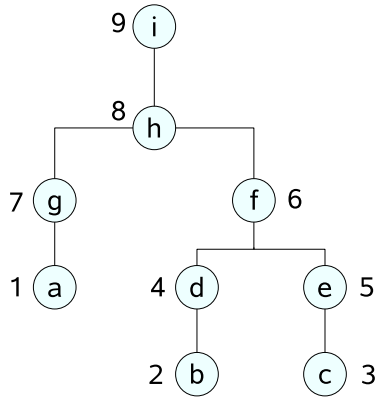


Abbildung 2.5: Eliminationsbaum für die Matrix in Abbildung 2.4

In Abbildung 2.5 ist der zu der Matrix aus Abbildung 2.4 zugehörige Eliminationsbaum dargestellt. Die Notation $T[j]$ soll nun anhand eines Knotens verbildlicht werden. So kann man nun für den in Knoten 6 eingewurzelten Unterbaum auch schreiben:

$$T[6] = \{2, 3, 4, 5, 6\}$$

An dieser Stelle angelangt, ist man nun in der Lage die Frontal- und die Update-Matrizen zu definieren. Man betrachte dazu die j-te Spalte von L und definiere $i_0, ..., i_r$ als die Zeilenindizes der Nicht-Null-Elemente in $L_{\star j}$, wobei $i_0 = j$ ist und die Spalte j genau r Nicht-Null-Elemente unterhalb der Diagonalen besitzt.

Die *Unterbaum-Update-Matrix* der Spalte j für die Matrix A ist dann definiert als die Matrix:

$$\overline{U}_j = - \sum_{k \in T[j] - \{j\}} \begin{pmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} (l_{j,k}\ l_{i_1,k} \cdots\ l_{i_r,k}) \tag{2.4}$$

Das heißt, die Matrix beinhaltet Beiträge von Vektorprodukten aus vorangegangenen Spalten, die im Eliminationsbaum direkte Nachfahren des Knotens j darstellen. Aus Satz 2.3.1 folgt unmittelbar, dass wenn k im Eliminationsbaum ein Nachkömmling von j ist, so ist auch die Menge von Indizes der Nicht-Null-Elemente aus $(l_{j,k}, l_{j+1,k}, \cdots, l_{n,k})^T$ enthalten in der Menge $\{j, i_1, \cdots, i_r\}$. Die Nicht-Null-Beiträge der Vektorprodukt-Updates aller Spalten der Nachkömmlinge von j sind somit enthalten in $\overline{U}_j$.

Anschließend kann man nun mit Hilfe der Unterbaum-Update-Matrix die j-te *Frontal-Matrix* für A bilden:

$$F_j = \begin{pmatrix} a_{j,j} & a_{j,i_1} & \cdots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{j,i_r} & & & \end{pmatrix} + \overline{U}_j \tag{2.5}$$

Die Dimension $(r+1)$ der Frontal-Matrix F_j ist die gleiche wie die der Matrix $\overline{U}_j$ und repräsentiert wieder die Anzahl der Nicht-Null-Elemente der j-ten Spalte des Faktors L.

Da Satz 2.3.2 besagt, dass wenn $k < j$ und $l_{j,k} \neq 0$ ist, k einen Nachkömmling von j darstellt, so kann man dafür auch schreiben $k \in T[j] - \{j\}$, was bedeutet, dass das Vektorprodukt-Update der Spalte k enthalten ist in $\overline{U}_j$. Mit dieser Erkenntnis lässt sich nun $\overline{U}_j$ wie folgt in zwei Komponenten aufspalten:

$$\overline{U}_j = - \sum_{\substack{k<j \\ l_{j,k} \neq 0}} \begin{pmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} (l_{j,k}\ l_{i_1,k} \cdots\ l_{i_r,k}) - \sum_{\substack{k \in T[j] - \{j\} \\ l_{j,k}=0}} \begin{pmatrix} 0 \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} (0\ l_{i_1,k} \cdots\ l_{i_r,k})$$

An dieser Darstellung lässt sich einfach erkennen, dass nur die erste der beiden Komponenten zu der ersten Spalte der Unterbaum-Update-Matrix $\overline{U}_j$ beiträgt. Demzufolge ist nun die erste Spalte von $\overline{U}_j$, die alle Nicht-Null-Update-Einträge für die j-te Spalte enthält, gegeben durch:

$$\overline{U}_j = - \sum_{\substack{k<j \\ l_{j,k} \neq 0}} l_{j,k} \begin{pmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix}$$

Diese Menge bezeichnet man auch als die *komplette Update-Spalte* zur Spalte j.

Wie man anhand der Definition 2.5 für F_j sehen kann, besteht die erste Spalte und die erste Zeile der Frontal-Matrix aus $A_{\star j}$ und der kompletten Update-Spalte zur Spalte j. Das heißt, wenn F_j berechnet ist, wurden die erste Zeile und Spalte von F_j komplett erneuert. Dafür ergibt ein Eliminationsschritt bezüglich F_j die Nicht-Null-Einträge der Faktorspalte $L_{\star j}$. Man erhält damit folgende Darstellung:

$$F_j = \begin{pmatrix} l_{j,j} & 0 \\ l_{i_1,j} & \\ \vdots & \\ l_{i_r,j} & I \end{pmatrix} \begin{pmatrix} 1 & 0 \\ & \\ 0 & U_j \end{pmatrix} \begin{pmatrix} l_{j,j} & l_{i_1,j} & \cdots & l_{i_r,j} \\ & & & \\ 0 & & I & \end{pmatrix} \tag{2.6}$$

Da $j, i_1, \cdots, i_r$ gerade die Indizes der Nicht-Null-Elemente in $L_{\star j}$ darstellen, ist einleuchtend, dass der Vektor $(l_{j,j}, l_{i_1,j}, \cdots, l_{i_r,j})^T$ vollbesetzt sein muss. Dies hat zur Folge, dass auch die erste Zeile und Spalte der Frontal-Matrix F_j vollbesetzt ist. Ferner muss auch die nach diesem einen Eliminierungsschritt noch zur Faktorisierung übrig gebliebende Matrix U_j vollbesetzt sein, die im weiteren Verlauf mit *Update-Matrix* der Spalte j bezeichnet wird.

Satz 2.3.3.

$$U_j = -\sum_{k \in T[j]} \begin{pmatrix} l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} (l_{i_1,k} \cdots l_{i_r,k}) \tag{2.7}$$

Beweis
Betrachtet man noch einmal die letzte Darstellung der Frontal-Matrix F_j, so kann man sehen, dass sie auch wie folgt umgeschrieben werden kann:

$$F_j = \begin{pmatrix} l_{j,j} \\ l_{i_1,j} \\ \vdots \\ l_{i_r,j} \end{pmatrix} (l_{j,j}\ l_{i_1,j}\ \cdots\ l_{i_r,j}) + \begin{pmatrix} 0 & 0 \\ & \\ 0 & U_j \end{pmatrix}$$

Vernachlässigt man nun bei F_j die erste Zeile und Spalte, so erhält man dieselbe Untermatrix wie durch Weglassen der ersten Zeile und Spalte der Matrix $\overline{U}_j$. Zusammengefasst heißt das:

$$-\sum_{k \in T[j]-\{j\}} \begin{pmatrix} l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} (l_{i_1,k} \cdots l_{i_r,k}) = \begin{pmatrix} l_{i_1,j} \\ \vdots \\ l_{i_r,j} \end{pmatrix} (l_{i_1,j}\ \cdots\ l_{i_r,j}) + U_j$$

woraus direkt 2.3.3 folgt.

Es ist überaus wichtig, dass man zwischen U_j und $\overline{U}_j$ unterscheidet. Wie man anhand der Gleichung (2.5) nachvollziehen kann, dient die Unterbaum-Update-Matrix $\overline{U}_j$ dazu, die j-te Frontal-Matrix zu bilden. Die Update-Matrix U_j hingegen entsteht erst aus einem Eliminationsschritt mit der zuvor erstellten Frontal-Matrix F_j. Dies erkennt man anhand der Darstellung von F_j in der Gleichung (2.6). Um den Unterschied noch etwas zu verdeutlichen folgt nun ein kleines Beispiel.

Beispiel zu Frontal- und Update-Matrizen

Im Folgenden wird weiterhin auf die Matrix aus Abbildung 2.4 Bezug genommen, um die Definitionen der Update- und der Frontal-Matrizen etwas zu veranschaulichen. Unschwer erkennbar ist, dass für die ersten drei Spalten $\overline{U}_1 = 0,\ \overline{U}_2 = 0$ und $\overline{U}_3 = 0$ gilt, da 1, 2 und 3 Blätter im Eliminationsbaum darstellen und somit keine Nachfolger haben. Bei Spalte 4 entsteht folgende Menge $T[4] - \{4\} = \{2\}$ und die Unterbaum-Update-Matrix für Spalte 4 ist gegeben durch:

$$\overline{U}_4 = -\begin{pmatrix} l_{4,2} \\ l_{6,2} \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} l_{4,2} & l_{6,2} & 0 & 0 \end{pmatrix} = -\begin{pmatrix} l_{4,2}^2 & l_{4,2}\, l_{6,2} & 0 & 0 \\ l_{6,2}\, l_{4,2} & l_{6,2}^2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Daraus folgt für die Frontal-Matrix:

$$F_4 = \begin{pmatrix} a_{4,4} & 0 & a_{4,8} & a_{4,9} \\ 0 & 0 & 0 & 0 \\ a_{8,4} & 0 & 0 & 0 \\ a_{9,4} & 0 & 0 & 0 \end{pmatrix} + \overline{U}_4 = \begin{pmatrix} a_{4,4} - l_{4,2}^2 & -l_{4,2}\, l_{6,2} & a_{4,8} & a_{4,9} \\ -l_{6,2}\, l_{4,2} & -l_{6,2}^2 & 0 & 0 \\ a_{8,4} & 0 & 0 & 0 \\ a_{9,4} & 0 & 0 & 0 \end{pmatrix}$$

Ein Eliminationsschritt bezüglich F_4 ergibt schließlich die Faktor-Spalte $L_{\star 4}$, sowie die vollbesetzte Update-Matrix

$$U_4 = -\sum_{k \,\in\, \{2,4\}} \begin{pmatrix} l_{6,k} \\ l_{8,k} \\ l_{9,k} \end{pmatrix} \begin{pmatrix} l_{6,k} & l_{8,k} & l_{9,k} \end{pmatrix} = \begin{pmatrix} -l_{6,2}^2 - l_{6,4}^2 & -l_{6,4}\, l_{8,4} & -l_{6,4}\, l_{9,4} \\ -l_{6,4}\, l_{8,4} & -l_{8,4}^2 & -l_{8,4}\, l_{9,4} \\ -l_{6,4}\, l_{9,4} & -l_{8,4}\, l_{9,4} & -l_{9,4}^2 \end{pmatrix}$$

Diese Darstellung lässt sich aufteilen in zwei Komponenten, da der Laufindex k der Summe aus einer zweielementigen Menge gewählt wird. Zusammen mit $l_{8,2} = l_{9,2} = 0$ ergibt sich daraus folgendes Bild:

$$U_4 = -\begin{pmatrix} l_{6,2} \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} l_{6,2} & 0 & 0 \end{pmatrix} - \begin{pmatrix} l_{6,4} \\ l_{8,4} \\ l_{9,4} \end{pmatrix} \begin{pmatrix} l_{6,4} & l_{8,4} & l_{9,4} \end{pmatrix}$$

Etwas aufwändiger wird es für die Update- und Frontal-Matrizen der sechsten Spalte, da die 6 im Eliminationsbaum vier Nachkommen besitzt. Das bedeutet, die Menge $T[6] - \{6\} = \{2, 3, 4, 5\}$ aus der das k der Summe zur Generierung der Unterbaum-Update-Matrix $\overline{U}_6$ gewählt wird, ist nun doppelt so groß wie bei $\overline{U}_4$. Das Auflösen der Summe ergibt schließlich:

$$\begin{aligned} \overline{U}_6 = &-\begin{pmatrix} l_{6,2} \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} l_{6,2} & 0 & 0 \end{pmatrix} - \begin{pmatrix} 0 \\ l_{8,3} \\ 0 \end{pmatrix} \begin{pmatrix} 0 & l_{8,3} & 0 \end{pmatrix} \\ &- \begin{pmatrix} l_{6,4} \\ l_{8,4} \\ l_{9,4} \end{pmatrix} \begin{pmatrix} l_{6,2} & l_{8,4} & l_{9,4} \end{pmatrix} - \begin{pmatrix} l_{6,5} \\ l_{8,5} \\ 0 \end{pmatrix} \begin{pmatrix} l_{6,5} & l_{8,5} & 0 \end{pmatrix} \end{aligned}$$

Die Frontal-Matrix F_6 lässt sich daraufhin bilden aus:

$$F_6 = \begin{pmatrix} a_{6,6} & 0 & a_{6,9} \\ 0 & 0 & 0 \\ a_{9,6} & 0 & 0 \end{pmatrix} + \overline{U}_6 = \begin{pmatrix} a_{6,6} - l_{6,2}^2 - l_{6,4}^2 - l_{6,5}^2 & -l_{6,4}l_{8,4} - l_{6,5}l_{8,5} & a_{6,9} - l_{6,4}l_{9,4} \\ -l_{6,4}l_{8,4} - l_{6,5}l_{8,5} & -l_{8,3}^2 - l_{8,4}^2 - l_{8,5}^2 & -l_{8,4}l_{9,4} \\ a_{9,6} - l_{6,4}l_{9,4} & -l_{8,4}l_{9,4} & -l_{9,4}^2 \end{pmatrix}$$

$\overline{U}_6$ beinhaltet an dieser Stelle alle Vektorprodukt-Beiträge der Spalten, die im Eliminationsbaum direkte Nachkommen darstellen. Da aber für die dritte Spalte $l_{6,3} = 0$ gilt, erhält man für diese Spalte keinen Beitrag bezüglich der ersten Spalte von F_6, gleiches gilt für den Eintrag $l_{9,3} = 0$ bezüglich der letzten Spalte der Frontal-Matrix.

Ein Eliminationsschritt bezüglich F_6 liefert wieder die Faktor-Spalte $L_{\star 6}$ und die Update-Matrix U_6. Wie auch schon für U_4 beschrieben, lässt sich mit Hilfe von Satz 2.3.1 die Update-Matrix wie folgt in mehreren Komponenten darstellen:

$$U_6 = -\begin{pmatrix} l_{8,3} \\ 0 \end{pmatrix}\begin{pmatrix} l_{8,3} & 0 \end{pmatrix} - \begin{pmatrix} l_{8,4} \\ l_{9,4} \end{pmatrix}\begin{pmatrix} l_{8,4} & l_{9,4} \end{pmatrix} - \begin{pmatrix} l_{8,5} \\ 0 \end{pmatrix}\begin{pmatrix} l_{8,5} & 0 \end{pmatrix} - \begin{pmatrix} l_{8,6} \\ l_{9,6} \end{pmatrix}\begin{pmatrix} l_{8,6} & l_{9,6} \end{pmatrix}$$

Die Vektorprodukt-Updates von $L_{\star 2}$ tragen hier nicht zur Berechnung von U_6 bei, da $l_{8,2}$ sowie $l_{9,2}$ beide gleich Null sind.

Entwicklung der Frontal- und der Update-Matrizen mit Hilfe der *erweiterten Addition*

Da die Frontal-Matrizen sowie die Update-Matrizen von zentraler Bedeutung für die Multifrontal-Methode sind, wird in diesem Teil des Kapitels nun auf den speziellen Zusammenhang zwischen den Frontal-Matrizen $\{F_j\}$ und den Update-Matrizen $\{U_j\}$ eingegangen. Damit diese Beschreibung des Zusammenhangs gelingt, benötigt man jedoch noch ein kleines Hilfsmittel.

Es sei nun zunächst V eine $v \times v$ Matrix, wobei v kleiner oder gleich n, der Matrixdimension der Ausgangsmatrix A, sein muss. Zudem sei W eine $w \times w$ Matrix ebenfalls mit $w \leq n$. Die Zeilen und Spalten von V und W werden gebildet aus Zeilen und Spalten der gegebenen Matrix A, wobei lediglich die Nicht-Null-Elemente berücksichtigt werden. Dazu seien $i_1 \leq \cdots \leq i_v$ die entsprechenden Indizes von V in A und $j_1 \leq \cdots \leq j_w$ die von Matrix W. Stellt man sich nun eine Vereinigung dieser beiden Indexmengen vor und definiert diese als $k_1 \leq \cdots \leq k_t$, so erkennt man, dass durch geschickte Erweiterung der Matrix V mittels Null-Zeilen und Null-Spalten, die ursprüngliche Indexmenge der Matrix dann gerade der neuen $k_1 \leq \cdots \leq k_t$ entspricht. Dasselbe gilt auch für die Matrix W und deren Indexmenge.

Definiert man nun $V \boxplus W$ als die Matrix T der Dimension $t \times t$, die durch Addition der beiden Matrizen V und W entsteht, dann kann man den dafür benötigten Operator „$\boxplus$“ als *erweiterte Addition* bezeichnen.

Diese Verallgemeinerung einer Matrixaddition lässt sich an einem kleinen Beispiel leicht veranschaulichen. Dazu seien nun V und W wie folgt definiert:

$$V = \begin{array}{c} \\ 4 \\ 6 \end{array}\begin{array}{c} \begin{array}{cc} 3 & 7 \end{array} \\ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \end{array} \qquad W = \begin{array}{c} \\ 4 \\ 8 \end{array}\begin{array}{c} \begin{array}{cc} 1 & 3 \end{array} \\ \begin{pmatrix} e & f \\ g & h \end{pmatrix} \end{array}$$

Die durch die erweiterte Addition gebildete 3×3 Matrix $V \boxplus W$ nimmt dann folgende Gestalt an:

$$V \boxplus W = \begin{array}{c} \\ 4 \\ 6 \\ 8 \end{array}\begin{array}{c} \begin{array}{ccc} 1 & 3 & 7 \end{array} \\ \begin{pmatrix} 0 & a & b \\ 0 & c & d \\ 0 & 0 & 0 \end{pmatrix} \end{array} \boxplus \begin{array}{c} \\ 4 \\ 6 \\ 8 \end{array}\begin{array}{c} \begin{array}{ccc} 1 & 3 & 7 \end{array} \\ \begin{pmatrix} e & f & 0 \\ 0 & 0 & 0 \\ g & h & 0 \end{pmatrix} \end{array} = \begin{array}{c} \\ 4 \\ 6 \\ 8 \end{array}\begin{array}{c} \begin{array}{ccc} 1 & 3 & 7 \end{array} \\ \begin{pmatrix} e & a+f & b \\ 0 & c & d \\ g & h & 0 \end{pmatrix} \end{array}$$

In diesem Kapitel soll nun der Zusammenhang zwischen $\{F_j\}$ und $\{U_j\}$ beschrieben werden. Wie gleich zu sehen ist, wird dies mit Hilfe des Eliminationsbaumes und der gerade eingeführten erweiterten Addition möglich sein.

Dazu betrachte man zunächst wieder die Spalte j und, wie im vorherigen Abschnitt auch, $j, i_1, \cdots, i_r$, die Zeilenindizes der Nicht-Null-Elemente der Spalte $L_{\star j}$. Der folgende Satz liefert dann die wichtige Beziehung zwischen den Update- und Frontal-Matrizen.

Satz 2.3.4. *Die Knoten $c_1, \cdots, c_s$ seien die Söhne von Knoten j im Eliminationsbaum. Dann gilt für die Frontal-Matrix:*

$$F_j = \begin{pmatrix} a_{j,j} & a_{j,i1} & \cdots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{i_r,j} & & & \end{pmatrix} \Updownarrow U_{c_1} \Updownarrow \cdots \Updownarrow U_{c_s} \tag{2.8}$$

Beweis
F_j ist in (2.5) definiert unter Verwendung der Matrizen $\overline{U}_j$, die wiederum nach (2.4) die Summe über alle Vektorprodukt-Updates der Spalten $T[j]-\{j\}$ abbilden. Da nun $c_1, \cdots, c_r$ die Nachkommen des Knotens j im Eliminationsbaum darstellen, kann $T[j]-\{j\}$ vereinfacht als die *disjunkte Vereinigung* der Knoten in den Unterbäumen $T[c_1], \cdots, T[c_s]$ beschrieben werden. Mit dieser Erkenntnis lässt sich nun $\overline{U}_j$ berechnen als die Summe der Vektorprodukt-Updates der Spalten aus $T[c_1], \cdots, T[c_s]$.
Für jedes der $T[c_t]$ mit $1 \leq t \leq s$ gilt nach Satz 2.3.3, dass dessen Vektorprodukt-Updates bezüglich F_j gegeben sind durch U_{c_t}. Das heißt, alle Updates bezogen auf die Spalten $T[j]-\{j\}$ sind in $U_{c_1}, \cdots, U_{c_s}$ enthalten, woraus das in Satz 2.3.4 behauptete folgt.

Die Untermatrix $U_{c_1} \Updownarrow \cdots \Updownarrow U_{c_s}$ aus (2.8) kann weniger Spalten und Zeilen enthalten als F_j. Ausgleichen kann man dies jedoch durch eine entsprechende Erweiterung, sodass die Indexsätze der beiden Matrizen übereinstimmen. Da die nun erweiterte Untermatrix $U_{c_1} \Updownarrow \cdots \Updownarrow U_{c_s}$ alle Vektorprodukt-Beiträge von $\overline{U}_j$ bezüglich F_j beinhaltet, ist sie nicht anderes als genau die benötigte Matrix $\overline{U}_j$ selbst.
Das Ergebnis ist die Grundlage der von Duff und Reid entwickelten Multifrontal-Methode. Sie bezeichnen dabei den gesamten Prozess zur Bildung der j-ten Frontal-Matrix F_j aus $A_{\star j}$ und der Update-Matrizen bezüglich den Nachkommen im Eliminationsbaum als die Frontal-Matrix *assembly*-Operation. Demzufolge wird die Baumstruktur, auf der die assembly Operationen basieren auch als *assembly tree*, also als „Montage-Baum“ bezeichnet.

Cholesky-Faktorisierung für dünnbesetzte Matrizen

Mit den nun gewonnenen Erkenntnissen ist man in der Lage, den Algorithmus der Cholesky-Faktorisierung, der schon in Kapitel 2.1 vorgestellt wurde, mittels Frontal-Matrizen $\{F_j\}$ und Update-Matrizen $\{U_j\}$ zu beschreiben.

Der Algorithmus aus Abbildung 2.6 schildert dabei die wesentliche Kernidee der Multifrontal-Methode. Um das Ergebnis aus Satz 2.3.4 in Bezug auf den dargestellten Algorithmus etwas zu verdeutlichen, sei nun F_6 wieder die Frontal-Matrix bezüglich der Ausgangsmatrix in Abbildung 2.4.
Auch wenn hier eigentlich vier Spalten $L_{\star 2}, L_{\star 3}, L_{\star 4}$ und $L_{\star 5}$ zu der Unterbaum-Update-Matrix $\overline{U}_6$ beitragen, so hat der Knoten 6 jedoch nur zwei Kinder, nämlich 4 und 5. Das bedeutet aber wiederum nach Satz 2.3.4, dass F_6 lediglich gebildet wird aus $A_{\star 6}$ und den beiden Update-Matrizen U_4 und U_5. Die Matrix U_4 beinhaltet aber die nötigen Update-Beiträge für die Spalten 4 und 2, U_5 beinhaltet die Informationen von Spalte 3 und 5. Im Endeffekt erhält man für die beiden Update-Matrizen folgende Darstellung:

$$U_4 = -\begin{pmatrix} l_{6,2} \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} l_{6,2} & 0 & 0 \end{pmatrix} - \begin{pmatrix} l_{6,4} \\ l_{8,4} \\ l_{9,4} \end{pmatrix} \begin{pmatrix} l_{6,4} & l_{8,4} & l_{9,4} \end{pmatrix}$$

$$U_5 = -\begin{pmatrix} 0 \\ l_{8,3} \end{pmatrix} \begin{pmatrix} 0 & l_{8,3} \end{pmatrix} - \begin{pmatrix} l_{6,5} \\ l_{8,5} \end{pmatrix} \begin{pmatrix} l_{6,5} & l_{8,5} \end{pmatrix}$$

Bestimme den Eliminationsbaum $T[A]$

Für Spalte $j = 1, .., n$

Wähle $j, i_1, \cdots, i_r$ als Zeilenindizes der Nicht-Null-Elemente von $L_{\star j}$

Es seien $c_1, \cdots, c_s$ die Nachkommen von j im Eliminationsbaum

Bilde die Update-Matrix $\overline{U} = U_{c_1} \Updownarrow \cdots \Updownarrow U_{c_s}$

$$F_j = \begin{pmatrix} a_{j,j} & a_{j,i1} & \cdots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{i_r,j} & & & \end{pmatrix} \Updownarrow \overline{U}$$

Faktorisiere F_j in

$$F_j = \begin{pmatrix} l_{j,j} & 0 \\ l_{i_1,j} & \\ \vdots & \\ l_{i_r,j} & I \end{pmatrix} \begin{pmatrix} 1 & 0 \\ & \\ 0 & U_j \end{pmatrix} \begin{pmatrix} l_{j,j} & l_{i_1,j} & \cdots & l_{i_r,j} \\ & & & \\ 0 & & I & \end{pmatrix}$$

Abbildung 2.6: Algorithmus zur Cholesky-Faktorisierung mittels Frontal- und Update-Matrizen

Der Indexsatz von F_6 gleicht dem Indexsatz der Matrix, die durch $U_4 \Updownarrow U_5$ entsteht. Es gilt also: $\overline{U}_6 = U_4 \Updownarrow U_5$.

Allgemeines zu assembly trees

Der Eliminationsbaum an sich dient dazu, den Aufbau der Frontal- und Update-Matrizen durchzuführen, wie es in Satz 2.3.4 erläutert ist. Ein assembly tree hingegen kann bezüglich der Multifrontal -Methode viel allgemeiner angegeben werden. Für die Definition eines assembly trees sei zunächst ein Baum mit n Knoten gegeben. Zudem muss für jeden Knoten j mit dem Vaterknoten v, also für alle $v > j$ gelten, dass die Struktur von $L_{\star j}$ unterhalb der Diagonalen eine Teilmenge der Struktur von $L_{\star v}$ darstellt.

Es ist nun einleuchtend, dass der Eliminationsbaum diese Voraussetzung erfüllt, aber es gibt auch andere Baumstrukturen, die diese Eigenschaft haben. Zum Beispiel kann man sich einen assembly tree derart definieren, dass ein Knoten v nur dann einen Vater von j darstellt, wenn gilt:

$$v = min\{k > j \mid \text{Struktur unterhalb der Diagonalen von } L_{\star j} \subseteq \text{Struktur von} L_{\star k}\}$$

Anhand dieser Definition kann man einen neuen Baum für das Matrixbeispiel aus Abbildung 2.4 erstellen, wobei nun der Vaterknoten von Knoten 6 nicht mehr 8 ist sondern der Knoten 7, wie anhand der Abbildung 2.7 nachzuvollziehen ist.

Mit jedem assembly tree ist man schließlich in der Lage, den Prozess der Multifrontal-Methode durchführen zu können. Offensichtlich müssen dann aber die Unterbaum-Update-Matrix $\overline{U}_j$ sowie die Frontal-Matrix entsprechend der Struktur des assembly trees gebildet werden und nicht mehr bezüglich der Struktur des Eliminationsbaums.
In der Praxis verschafft diese allgemeine Art des assembly trees jedoch häufig keinen sichtbaren

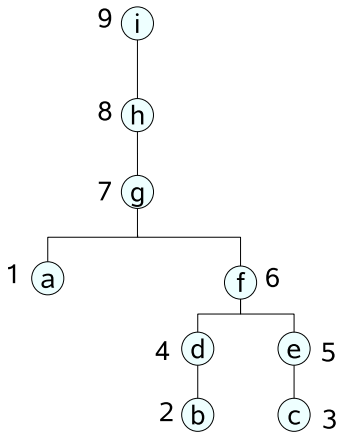


Abbildung 2.7: Ein möglicher assembly tree für die Matrix aus Abbildung 2.4

Vorteil gegenüber dem zuvor definierten Eliminationsbaums, so dass in Zukunft weiterhin der Eliminationsbaum zum Matrixaufbau verwendet wird. Dieser Sachverhalt wird ausführlich in [1] diskutiert.

Graphentheoretische Interpretation

In diesem Unterkapitel wird nun kurz darauf eingegangen, welches graphentheoretische Konzept hinter der Idee der Frontal- und Update-Matrizen steckt. Sei dazu G ein ungerichteter Graph, der zu der gegebenen $n \times n$ Matrix A aus Abbildung 2.4 gehört. Ohne Einschränkung der Allgemeinheit kann man annehmen, dass G zusammenhängend ist. Betrachtet man nun jegliche zusammenhängende Teilgraphen C von G, so definiert man die an C angrenzende Menge von Knoten aus G als die *Front* der Teilmenge C. Um diese Front einer Teilmenge bezüglich des Graphen G bezeichnen zu können wird die Funktion $Adj_G(C)$ verwendet, welche die Adjazenzliste zu der Teilmenge C darstellt. Diese Funktion wird dazu gebraucht um einen Graphen effizient abspeichern zu können. Die Bedeutung wird nun im Folgenden an einem Beispiel verdeutlicht.
Es ist offensichtlich, dass für jeden Knoten j im Eliminationsbaum, der Unterbaum $T[j]$ einen zusammenhängenden Teilgraphen von G darstellt. Die Zeilen- und Spaltenindizes der Frontal-Matrix F_j und der Update-Matrix U_j sind präzise gegeben durch die Mengen $\{j\} \cup Adj_G(T[j])$ und $Adj_G(T[j])$.

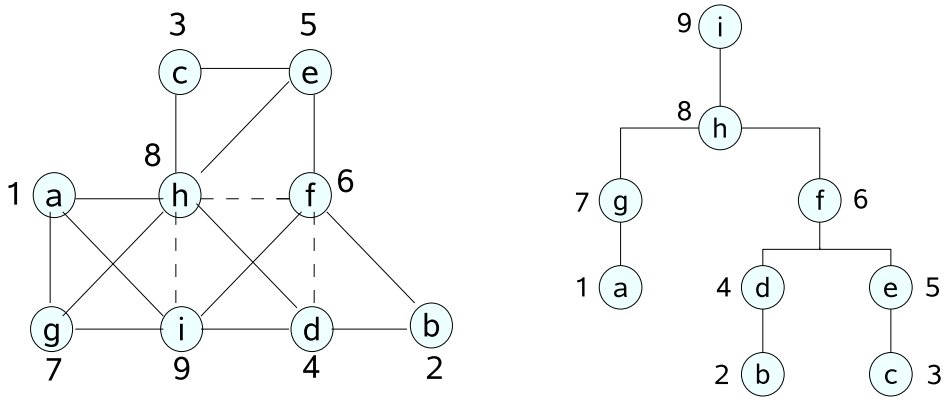


Abbildung 2.8: Ungerichteter Graph und Eliminationsbaum für die Matrix aus Abbildung 2.4

In Abbildung 2.8 ist nun der ungerichtete, zu der Matrix in Abbildung 2.4 passende Graph dargestellt, wobei jede durchgezogene Linie einen Nicht-Null-Eintrag „•“ in der Ursprungsmatrix A

repräsentiert und eine gestrichelte Linie einen Fill-In „○“ im Cholesky-Faktor. Der Übersicht halber ist daneben noch einmal der dazugehörige Eliminationsbaum abgebildet.

Betrachtet man zum Beispiel den Knoten 5, so bildet $Adj_G(T[5]) = Adj_G(\{3,5\}) = \{6,8\}$ genau die Indexmenge der Update-Matrix U_5. Äquivalentes Vorgehen für den Knoten 6 ergibt mit $T[6] = \{2,3,4,5,6\}$, dass $Adj_G(T[6]) = \{8,9\}$ ist, was dem Indexsatz von U_6 entspricht.

Ziel ist es, dass der ganze Eliminationsprozess der Matrix schließlich zurückgeführt werden kann auf ein Verfahren, bei dem lediglich mehrere dieser Fronten gebildet werden.

An einem kleinen Beispiel soll dieser Sachverhalt nun noch einmal verdeutlicht werden. Dazu seien aus dem in Abbildung 2.8 dargestellten Eliminationsbaum die Knoten 1 bis 5 bereits eliminiert worden. Zu diesem Zeitpunkt existieren dann genau drei Teilbäume mit schon eliminierten Knoten und zwar $T[1] = \{1\}$, $T[4] = \{2,4\}$ und $T[5] = \{3,5\}$. Die zugehörigen Adjazenzlisten werden durch die drei Fronten $Adj_G(T[1]) = \{7,8,9\}$, $Adj_G(T[4]) = \{6,8,9\}$ und $Adj_G(T[5]) = \{6,8\}$ abgebildet.
Man betrachte nun den nächsten Schritt, in dem die Frontal-Matrix F_6 mittels $\overline{U}_6$ gebildet wird, wobei nach Satz 2.3.4 die Matrix $\overline{U}_6$ ja gerade aus $U_4 \dotplus U_5$ entsteht. Das heißt also, dass zur Bildung der Frontal-Matrix F_6 keine Update-Informationen bezüglich des Knotens 1 benötigt werden. Vereinigt man nun die beiden Fronten $Adj_G(T[4])$ und $Adj_G(T[5])$, so erhält man die Menge $\{6,8,9\}$, die gerade der Indexmenge der Frontalmatrix F_6 entspricht. Eliminiert man anschließend den Knotens 6 aus dieser Menge, so entsteht eine neue Front mit der Menge $\{8,9\}$, die genau $Adj_G(T[6])$ entspricht und somit auch dem Indexsatz der Update-Matrix U_6.

Ein Eliminationsschritt auf einer Front F_j bezüglich dem Knoten j im Eliminationsbaum führt schließlich in einem von zwei Fällen zu der Bildung einer neuen Front. Falls der Knoten j ein Blatt im Eliminationsbaum darstellt, so wird die neue Front $Adj_G(T[j])$ lediglich zu der Menge der bisher bestimmten Fronten hinzugefügt. Ist der betrachtete Knoten aber kein Blatt, erhält man die neue Front durch Zusammenmischen der Fronten aller Nachkommen von Knoten j.

Die Multifrontal-Methode bildet somit einen neuartigen Entwurf, der die numerische Berechnung mittels der gerade dargestellten graphentheoretischen Sinngebung komplett umgestaltet.

Speicherung von Frontal-/ Update-Matrizen und Baum-Postordering

Der Algorithmus 2.6, der die Grundidee der Multifrontal-Methode beschreibt, zeigt, dass die Spalten in aufsteigender Reihenfolge von 1 bis n abgearbeitet und gleichzeitig die Update-Matrizen $\{U_j\}$ gebildet werden. Im Ablauf der zugrundeliegenden Cholesky-Faktorisierung der Matrix A kommt es jedoch vor, dass die gerade berechnete Update-Matrix U_j nicht sofort zur Generierung von F_{j+1} beiträgt, sondern erst in einem der nächsten Schritte gebraucht wird. Das hat zur Folge, dass temporär Speicherplatz für diese Matrix angelegt werden muss, damit sie für darauffolgende Bildungen von Frontal-Matrizen dienen kann.

Betrachtet man nochmals die Beispielmatrix in Abbildung 2.4, so erkennt man, dass zur Bearbeitung der Frontal-Matrix F_4 lediglich die Update-Matrix U_2 benötigt wird. Das heißt, dass die beiden Matrizen U_1 und U_3 aus den vorherigen Schritten zunächst im Speicher abgelegt werden müssen. Aus der Frontal-Matrix F_4 wird wiederum die Update-Matrix U_4 gebildet, die aber für den nächsten Schritt, in dem die Frontal-Matrix F_5 mit Hilfe der im Speicher abgelegten Update-Matrix U_3 berechnet wird, keine Verwendung hat. Sie muss also auch zunächst temporär im Speicher abgelegt werden, bis sie zur Berechnung der Frontal-Matrix F_6 wieder abgerufen wird.

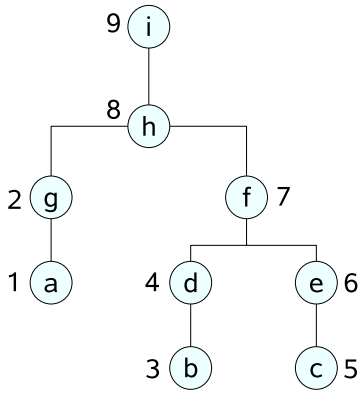
Anhand dieser Überlegungen wird deutlich, dass eine systematische Speicherung und Abfrage der Update-Matrizen erforderlich ist, um mit der Multifrontal-Methode möglichst effektiv arbeiten zu können. Um eine mögliche Verschwendung von Speicherplatz zu vermeiden, wird nun eine Neuanordnung der Matrix vorgestellt, die basierend auf dem Eliminationsbaum definiert ist.

Um den Eliminationsbaum „optimal“ durchlaufen zu können, muss dieser zunächst neu geordnet werden. Dazu wird das sogenannte *Postordering* verwendet, welches eine typische topologische Anordnung bezüglich $T[A]$ darstellt. Laut Definition ist eine Anordnung bezüglich $T[A]$ genau dann ein Postordering, wenn die Knoten in jedem Teilbaum $T[j]$ mit $j = 1, \cdots, n$ konsekutiv nummeriert werden, wobei der Wurzelknoten stets als letztes nummeriert wird. Siehe dazu auch [1].

Durch die aufsteigend nummerierten Knoten in jedem Teilbaum ergibt sich nun der Vorteil, dass die Update-Matrizen in einem Stack abgespeichert und im LIFO-Prinzip (last in first out) abgearbeitet werden können. Ein unnötig langes Abspeichern von Update-Matrizen kann somit also umgangen werden.

Das bedeutet, wenn die aktuelle Frontal-Matrix generiert wird, so speichert man die daraus entstehende Update-Matrix auf dem Stack ab. Wird auf der anderen Seite eine Update-Matrix zur Berechnung einer der nachfolgenden Frontal-Matrizen benötigt, so wird diese unmittelbar aus dem Stack geholt.

Anhand der Matrix aus Abbildung 2.4 und dem dazugehörigen Eliminationsbaum aus Abbildung 2.5 lässt sich das Prinzip des Postorderings und die daraus entstehenden Vorteile leicht nachvollziehen. In Abbildung 2.9 ist der neu sortierte Eliminationsbaum zusammen mit der zugehörigen Matrix und ihrem Cholesky-Faktor abgebildet. Die Abbildung 2.10 zeigt den benötigten Stack, in dem die Update-Matrizen abgelegt werden. Hierbei beschreibt ein Rechteck links von der betrachteten Frontal-Matrix F_j gerade den Inhalt des Stacks vor der Berechnung von F_j. Jeder der Pfeile repräsentiert dabei das Holen einer Update-Matrix aus dem Stack, welche zur Bildung der aktuellen Frontal-Matrix benötigt wird. Man sieht nun deutlich, dass zur Berechnung einiger Frontal-Matrizen gar keine Update-Matrizen benötigt werden, wohingegen zur Bestimmung anderer Frontal-Matrizen sogar mehrere Update-Matrizen beitragen.



$$\overline{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} a & & & & & & & \bullet & \bullet \end{array} \\ \begin{array}{ccccccccc} \bullet & g & & & & & & \bullet & \bullet \end{array} \\ \begin{array}{ccccccccc} & & b & \bullet & & & \bullet & & \end{array} \\ \begin{array}{ccccccccc} & & \bullet & d & & & & \bullet & \bullet \end{array} \\ \begin{array}{ccccccccc} & & & & c & \bullet & & \bullet & \end{array} \\ \begin{array}{ccccccccc} & & & & \bullet & e & \bullet & \bullet & \end{array} \\ \begin{array}{ccccccccc} & & \bullet & & & \bullet & f & & \bullet \end{array} \\ \begin{array}{ccccccccc} \bullet & \bullet & & \bullet & \bullet & \bullet & & h & \end{array} \\ \begin{array}{ccccccccc} \bullet & \bullet & & \bullet & & & \bullet & & i \end{array} \end{array} \qquad \overline{L} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} a & & & & & & & & \end{array} \\ \begin{array}{ccccccccc} \bullet & g & & & & & & & \end{array} \\ \begin{array}{ccccccccc} & & b & & & & & & \end{array} \\ \begin{array}{ccccccccc} & & \bullet & d & & & & & \end{array} \\ \begin{array}{ccccccccc} & & & & c & & & & \end{array} \\ \begin{array}{ccccccccc} & & & & \bullet & e & & & \end{array} \\ \begin{array}{ccccccccc} & & \bullet & \circ & & \bullet & f & & \end{array} \\ \begin{array}{ccccccccc} \bullet & \bullet & & \bullet & \bullet & \bullet & \circ & h & \end{array} \\ \begin{array}{ccccccccc} \bullet & \bullet & & \bullet & & & \bullet & \circ & i \end{array} \end{array}$$

Abbildung 2.9: Postordering und neu geordnete Matrix für das Beispiel aus Abbildung 2.4

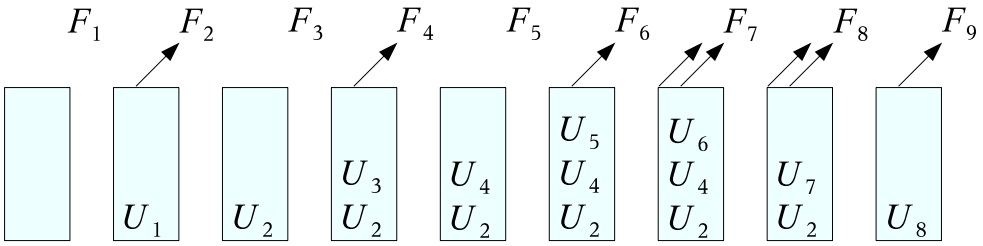


Abbildung 2.10: Die Stackinhalte bezogen auf das Posterdering aus Abbildung 2.9

In [1] wird zudem noch ein Kriterium zum optimalen Postordering bezüglich des Speicherverbrauchs angegeben. In der Praxis reicht jedoch meistens ein einfaches Postordering, wie es oben angegeben ist aus, um den Speicherverbrauch in ausreichendem Maße reduzieren zu können.

Ein ausführliches Beispiel für die Multifrontal-Methode

Um die Arbeitsweise der Multifrontal-Methode nun abschließend noch einmal zu verdeutlichen, sei hier ein ausführliches Beispiel anhand der symmetrisch positiv definiten Matrix

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

gegeben. Um einen nicht zu trivialen Eliminationsbaum und ein aussagekräftiges Beispiel zu erhalten, ist die Matrixdimension absichtlich ein wenig größer gewählt worden. Es wird nun im Folgenden die Zerlegung der Matrix A in

$$A = LL^T$$

gesucht.

In Abbildung 2.11 sind zudem der zugehörige zusammenhängende Graph sowie der Eliminationsbaum abgebildet, die zur Durchführung der Multifrontal-Methode benötigt werden. Im Graphen bilden die gestrichelten Linien auch hier wieder die Verbindungen, die durch „Fill-In“ entstehen, wie im vorherigen Teil des Kapitels schon beschrieben wurde.

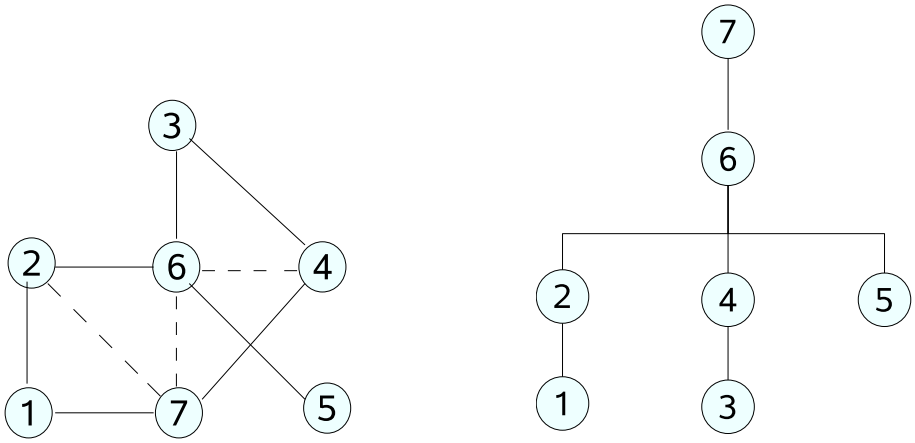


Abbildung 2.11: Graph und Baum zur Matrix A

Des Weiteren benötigt man zur Berechnung der Frontal- und Update-Matrizen die Formeln aus den Gleichungen (2.7) und (2.8), sowie zwei zusätzliche Index-Mengen $\mathcal{L}_j$, $\mathcal{L}_j^\star$. Die Menge $\mathcal{L}_j$ ist dabei der Satz der Zeilenindizes von Nicht-Null-Einträgen einer Spalte j unterhalb der Diagonalen von A und $\mathcal{L}_j\star$ ist dieselbe Menge inklusive des Indexes j selber.

- **Knoten 1:**
  Knoten 1 ist ein Blatt im Eliminationsbaum, was bedeutet, dass keine Update-Matrizen aus vorherigen Schritten mit zur Bildung der Frontal-Matrix F_1 beitragen.

$$\mathcal{L}_1^\star = \{1, 2, 7\} \quad \mathcal{L}_1 = \{2, 7\}$$

$$F_1 = \begin{pmatrix} a_{1,1} & a_{1,\mathcal{L}_1} \\ a_{\mathcal{L}_1,1} & 0 \end{pmatrix} = \begin{matrix} \\ 1 \\ 2 \\ 7 \end{matrix} \begin{matrix} \begin{matrix} 1 & 2 & 7 \end{matrix} \\ \begin{pmatrix} 4 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{matrix}$$

  Der Knoten 1 steht nun zur Elimination bereit. Durch die Elimination aus F_1, mittels der Gleichungen (2.2) aus dem Kapitel 2.1, erhält man dann die erste Spalte der gesuchten Faktor-

Matrix L:

$$L_1 = \begin{matrix} 1 \\ 2 \\ 7 \end{matrix} \begin{pmatrix} 2 \\ \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$$

Die Update-Matrix U_1 hat unter Verwendung der Gleichung (2.7) schließlich folgendes Aussehen:

$$U_1 = \begin{matrix} 2 \\ 7 \end{matrix} \begin{matrix} \begin{matrix} 2 & 7 \end{matrix} \\ \begin{pmatrix} -\frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & -\frac{1}{4} \end{pmatrix} \end{matrix}$$

- **Knoten 2:**
  Knoten 2 besitzt den Knoten 1 als Sohn, was bedeutet, dass die Update-Matrix U_1 aus dem vorherigen Schritt mit zur Bildung der Frontal-Matrix F_2 beiträgt. Da die Indexsätze nicht übereinstimmen, wird mittels der erweiterten Addition je eine Null-Zeile und Null-Spalte in beide Matrizen eingefügt. Es ergibt sich dann Folgendes:

$$\mathcal{L}_2^\star = \{2, 6\} \quad \mathcal{L}_2 = \{6\}$$

$$F_2 = \begin{pmatrix} a_{2,2} & a_{2,\mathcal{L}_2} \\ a_{\mathcal{L}_2,2} & 0 \end{pmatrix} \Updownarrow U_1 = \begin{matrix} 2 \\ 6 \\ 7 \end{matrix} \begin{matrix} \begin{matrix} 2 & 6 & 7 \end{matrix} \\ \begin{pmatrix} \frac{15}{4} & 1 & -\frac{1}{4} \\ 1 & 0 & 0 \\ -\frac{1}{4} & 0 & -\frac{1}{4} \end{pmatrix} \end{matrix}$$

  Der Knoten 2 steht nun zur Elimination aus F_2 bereit und man erhält die zweite Spalte der Faktor-Matrix L:

$$L_2 = \begin{matrix} 2 \\ 6 \\ 7 \end{matrix} \begin{pmatrix} \frac{\sqrt{15}}{2} \\ \frac{2\sqrt{15}}{15} \\ -\frac{\sqrt{15}}{30} \end{pmatrix}$$

  Hier ist nun schön der Effekt des „Fill-In“ zu erkennen. Die zweite Spalte von L hat nun in der siebten Zeile einen Eintrag. Das entsprechende Feld der Systemmatrix A war jedoch unbesetzt.
  Für die Update-Matrix U_2 gilt:

$$U_2 = \begin{matrix} 6 \\ 7 \end{matrix} \begin{matrix} \begin{matrix} 6 & 7 \end{matrix} \\ \begin{pmatrix} -\frac{4}{15} & \frac{1}{15} \\ \frac{1}{15} & -\frac{4}{15} \end{pmatrix} \end{matrix}$$

- **Knoten 3:**
  Knoten 3 ist wieder ein Blatt (keine Beiträge von Update-Matrizen).

$$\mathcal{L}_3^\star = \{3, 4, 6\} \quad \mathcal{L}_3 = \{4, 6\}$$

$$F_2 = \begin{pmatrix} a_{3,3} & a_{3,\mathcal{L}_3} \\ a_{\mathcal{L}_3,3} & 0 \end{pmatrix} = \begin{matrix} 3 \\ 4 \\ 6 \end{matrix} \begin{matrix} \begin{matrix} 3 & 4 & 6 \end{matrix} \\ \begin{pmatrix} 4 & 2 & \frac{1}{2} \\ 2 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \end{pmatrix} \end{matrix}$$

  Der Knoten 3 steht nun zur Elimination aus F_3 bereit und man erhält die dritte Spalte von L:

$$L_3 = \begin{matrix} 3 \\ 4 \\ 6 \end{matrix} \begin{pmatrix} 2 \\ 1 \\ \frac{1}{4} \end{pmatrix}$$

  Für die Update-Matrix U_3 gilt:

$$U_3 = \begin{array}{c} \\ 4 \\ 6 \end{array} \begin{array}{c} \begin{array}{cc} 4 & 6 \end{array} \\ \begin{pmatrix} -1 & -\frac{1}{4} \\ -\frac{1}{4} & -\frac{1}{16} \end{pmatrix} \end{array}$$

- **Knoten 4:**
  Knoten 4 besitzt den Knoten 3 als Sohn und wird somit gleich behandelt wie Knoten 2 bezüglich seinem Sohn, dem Knoten 1.

$$\mathcal{L}_4^\star = \{4, 7\} \quad \mathcal{L}_4 = \{7\}$$

$$F_4 = \begin{pmatrix} a_{4,4} & a_{4,\mathcal{L}_4} \\ a_{\mathcal{L}_4,4} & 0 \end{pmatrix} \updownarrow U_3 = \begin{array}{c} \\ 4 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccc} 4 & 6 & 7 \end{array} \\ \begin{pmatrix} 3 & -\frac{1}{4} & 1 \\ -\frac{1}{4} & -\frac{1}{16} & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{array}$$

  Der Knoten 4 steht nun zur Elimination aus F_4 bereit und man erhält die vierte Spalte von L:

$$L_4 = \begin{array}{c} \\ 4 \\ 6 \end{array} \begin{array}{c} \\ \begin{pmatrix} \sqrt{3} \\ \frac{\sqrt{3}}{4} \end{pmatrix} \end{array}$$

  Auch hier hat wieder ein Fill-In stattgefunden, und zwar in der sechsten Zeile von L_4. Für die Update-Matrix U_4 gilt:

$$U_4 = \begin{array}{c} \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cc} 6 & 7 \end{array} \\ \begin{pmatrix} -\frac{1}{48} & \frac{1}{12} \\ \frac{1}{12} & -\frac{1}{3} \end{pmatrix} \end{array}$$

- **Knoten 5:**
  Knoten 5 ist ein Blatt (keine Beiträge von Update-Matrizen).

$$\mathcal{L}_5^\star = \{5, 6\} \quad \mathcal{L}_5 = \{6\}$$

$$F_5 = \begin{pmatrix} a_{5,5} & a_{5,\mathcal{L}_5} \\ a_{\mathcal{L}_5,5} & 0 \end{pmatrix} = \begin{array}{c} \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cc} 5 & 6 \end{array} \\ \begin{pmatrix} 4 & 1 \\ 1 & 0 \end{pmatrix} \end{array}$$

  Der Knoten 5 steht nun zur Elimination aus F_5 bereit und man erhält die fünfte Spalte von L:

$$L_5 = \begin{array}{c} \\ 5 \\ 6 \end{array} \begin{array}{c} \\ \begin{pmatrix} 2 \\ \frac{1}{2} \end{pmatrix} \end{array}$$

  Für die Update-Matrix U_5 gilt:

$$U_5 = \begin{array}{c} \\ 6 \end{array} \begin{array}{c} \begin{array}{c} 6 \end{array} \\ \begin{pmatrix} -\frac{1}{4} \end{pmatrix} \end{array}$$

- **Knoten 6:**
  Der Knoten 6 ist der interessanteste Knoten, denn er besitzt drei Söhne, die Knoten 2, 4 und 5. Das heißt, es fließen drei Update-Matrizen mit in die Berechnung von F_6 ein. Damit die Matrizen miteinander verrechnet werden können, werden auch hier wieder mittels der erweiterten Addition Leerzeilen und Leerspalten in die Matrizen eingefügt, damit alle den gleichen Indexsatz erhalten.

$$\mathcal{L}_4^\star = \{6\} \quad \mathcal{L}_4 = \{\}$$

$$F_6 = \begin{pmatrix} a_{6,6} & a_{6,\mathcal{L}_6} \\ a_{\mathcal{L}_6,6} & 0 \end{pmatrix} \Updownarrow U_2 \Updownarrow U_4 \Updownarrow U_5$$

$$= \begin{matrix} & 6 & 7 \\ 6 & 4 & 0 \\ 7 & 0 & 0 \end{matrix} + \begin{matrix} & 6 & 7 \\ 6 & -\frac{4}{15} & \frac{1}{15} \\ 7 & \frac{1}{15} & -\frac{4}{15} \end{matrix} + \begin{matrix} & 6 & 7 \\ 6 & -\frac{1}{48} & \frac{1}{12} \\ 7 & \frac{1}{12} & -\frac{1}{3} \end{matrix} + \begin{matrix} & 6 & 7 \\ 6 & -\frac{1}{4} & 0 \\ 7 & 0 & 0 \end{matrix}$$

$$F_6 = \begin{matrix} & 6 & 7 \\ 6 & \frac{17}{5} & \frac{3}{20} \\ 7 & \frac{3}{20} & -\frac{7}{20} \end{matrix}$$

Der Knoten 6 steht nun zur Elimination aus F_6 bereit und man erhält die sechste Spalte der Matrix L. Auch hier tritt wieder der Fill-In Effekt auf:

$$L_6 = \begin{matrix} & 6 \\ 6 & \frac{\sqrt{85}}{5} \\ 7 & \frac{3\sqrt{85}}{340} \end{matrix}$$

Für die Update-Matrix U_6 gilt:

$$U_6 = \begin{matrix} & 7 \\ 7 & (-\frac{165}{272}) \end{matrix}$$

- **Knoten 7:**
  Knoten 7 ist der letzte zu eliminierende Knoten im Eliminationsbaum. Er besitzt den Knoten 6 als Sohn.

$$\mathcal{L}_4^\star = \{7\} \quad \mathcal{L}_4 = \{\}$$

$$F_7 = \begin{matrix} & 7 \\ & (a_{7,7}) \end{matrix} \Updownarrow U_6 = \begin{matrix} & 7 \\ 7 & (\frac{923}{272}) \end{matrix}$$

  Der Knoten 7 steht nun zur Elimination aus F_7 bereit und man erhält die siebte und letzte Spalte von L:

$$L_7 = \begin{matrix} & 7 \\ 7 & \left(\frac{\sqrt{15691}}{68}\right) \end{matrix}$$

Die endgültige Faktorisierung der Matrix A in LL^T ist schließlich mit der Matrix L möglich, die aus den jeweils berechneten Spalten L_j aus den vorherigen Berechnungen zusammengesetzt wird. Die drei Fill-In Werte, die während der Faktorisierung entstanden sind, wurden fettgedruckt, um sie etwas von den anderen Einträgen hervorzuheben:

$$L = \begin{pmatrix} 2 & & & & & & \\ \frac{1}{2} & \frac{\sqrt{15}}{2} & & & & & \\ & & 2 & & & & \\ & & 1 & \sqrt{3} & & & \\ & & & & 2 & & \\ & \frac{2\sqrt{15}}{15} & \frac{1}{4} & \frac{-\sqrt{\mathbf{3}}}{\mathbf{12}} & \frac{1}{2} & \frac{\sqrt{85}}{5} & \\ \frac{1}{2} & -\frac{\sqrt{\mathbf{15}}}{\mathbf{30}} & & \frac{\sqrt{3}}{3} & & \frac{\mathbf{3}\sqrt{\mathbf{85}}}{\mathbf{340}} & \frac{\sqrt{15691}}{68} \end{pmatrix}$$

Parallelisierung der Multifrontal-Methode

Da im Zuge der Diplomarbeit der Multifrontal-Löser MUMPS in ein paralleles Eigenwertprogramm eingebunden werden sollte, wird nun in diesem letzten Abschnitt des Kapitels erläutert, wie es

möglich wird, die Multifrontal-Methode effizient auf einem Parallelrechner auszuführen. Das heißt, die einzelnen Rechenschritte, die unabhängig ablaufen können, werden auf die zur Verfügung stehenden Prozessoren verteilt, die dann parallel an der Lösung des Problems arbeiten. Allgemeines zum Thema Parallelrechner ist desweiteren in [30] zu finden.

Bei der Parallelverarbeitung wird zunächst im Laufe der Analyse-Phase der Eliminationsbaum auf die jeweiligen Prozessoren statisch aufgeteilt. Dieser Vorgang wird in der Literatur häufig auch als Abbildung oder Mapping bezeichnet, vergleiche dazu auch [3].

Das Ziel der Analyse-Phase ist es, eine Kontrolle über die Kommunikationskosten, den Ausgleich des benutzten Speicherplatzes und auch über die Rechenkosten der einzelnen Prozessoren zu erlangen. Die Rechenkosten werden dabei angenähert über die Anzahl an durchzuführenden Floating-Point-Operationen. Es wird nun im Folgenden eine Vorgehensweise dargestellt, die es ermöglicht, den Eliminationsbaum derart auf die Prozessoren abzubilden, dass gleichzeitig die oben genannten Faktoren kontrolliert werden können.

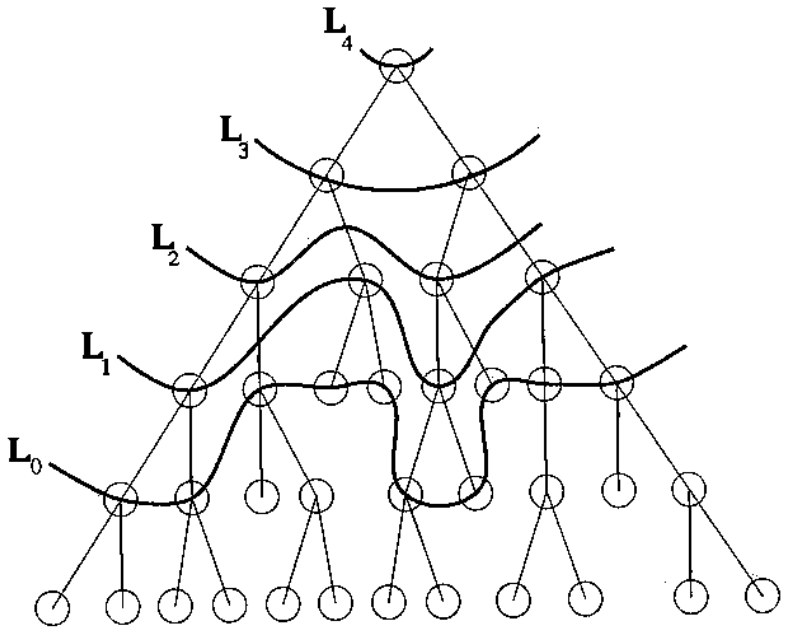


Abbildung 2.12: Einteilung des Eliminationsbaumes in die verschiedenen Level

Es sei dazu als Beispiel ein Eliminationsbaum gegeben, Abbildung 2.12, der von unten nach oben Level für Level bearbeitet wird. Level 0 wird dabei mit Hilfe des nachstehenden Algorithmus 2.13 ermittelt. Die dazugehörigen Abbildung 2.14 soll die Vorgehensweise des Algorithmus noch ein wenig verdeutlichen.

L_0 sei die Menge der Wurzeln des Eliminationsbaums
while Belastung unausgeglichen
  Finde Knoten q aus L_0, der am meisten Rechenkosten verursacht
  Initialisiere L_0 mit $(L_0 \setminus \{q\}) \cup \{\text{Nachkommen von } q\}$
  Verteile die Knoten von L_0 zyklisch auf die Prozessoren
  Schätze die Unausgeglichenheit ab
end while

Abbildung 2.13: Entwicklung und Abbildung des Anfangs-Levels L_0

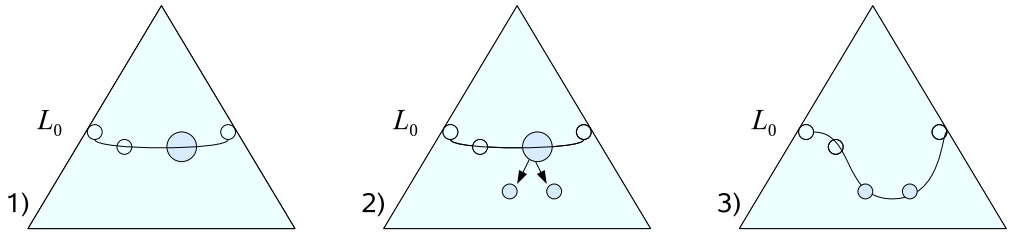


Abbildung 2.14: Ein Schritt bei der Bildung von Level L_0

Ein Knoten gehört zu L_i, $i > 0$, wenn alle seine Söhne zu L_j gehören mit $j < i$. Als erstes werden die Knoten von Level 0 und deren untergeordneten Teilbäume abgebildet. Dieser erste Schritt wird durchgeführt, um den Arbeitsaufwand in den Unterbäumen auszugleichen und die benötigte Kommunikation zu reduzieren, da alle Knoten eines Unterbaums auf denselben Prozessor abgebildet werden. Normalerweise ist es notwendig, viel mehr Knoten als Prozessoren in Level 0 zu haben, um einen guten Ausgleich zu erhalten. Deshalb ist L_0 auch abhängig von der Prozessoranzahl, mit der die Multifrontal-Methode durchgeführt werden soll. Sprich, je mehr Prozessoren beteiligt werden, desto kleiner werden auch die Unterbäume, solange dies mit der Dimension der Matrix vereinbar ist.

Die Abbildung bezüglich höherer Levels im Baum bewirkt lediglich einen Ausgleich von Speicherplatz. Für jeden Prozessor wird als erstes der Speicherbedarf für die Knoten aus Level L_0 berechnet. Für Knoten aus allen anderen Levels L_i, mit $i \geq 1$ gilt, dass jeder noch nicht abgebildete Knoten auf denjenigen Prozessor abgebildet wird, der bisher den wenigsten Speicherplatz benötigt. Nach der Abbildung des Knotens wird der Speicherplatzbedarf des Prozessors dementsprechend korrigiert, und es wird mit dem nächsten Knoten fortgefahren.

Anschließend wird die Koeffizientenmatrix anhand dieser durchgeführten Abbildung explizit auf die jeweiligen Prozessoren verteilt. Zusätzlich ist es mittels der Abbildung nun möglich Aussagen über den Umfang an Rechenarbeit und den Speicherplatzbedarf zu tätigen.

Anhand des ausführlichen Beispiels für die Faktorisierung einer Koeffizientenmatrix im vorherigen Abschnitt des Kapitels konnte man erkennen, dass während dem Prozesses der Faktorisierung in einigen Knoten sehr viel mehr Rechenaufwand anfiel als in anderen, weil immer mehr Update-Beiträge aus Knoten niedrigerer Levels mit einbezogen werden müssen, je weiter man im Baum in Richtung Wurzel vordringt. Amestoy und Duff haben sogar zeigen können, dass in den obersten drei Levels des Eliminationsbaumes mehr als drei Viertel des gesamten Berechnungsaufwandes stattfindet. Dies macht es notwendig, für diese „großen“ Knoten in der Nähe der Wurzel eine Parallelverarbeitung der Berechnungen innerhalb der Knoten zu definieren.
Die Parallelisierung wird dabei in drei verschiedene Typen unterteilt, die Typ1-, Typ2- und Typ3-Parallelisierung.

Typ1-Parallelisierung
Hier liegt der rein serielle Fall vor, sprich die Frontal-Matrix F_j eines Knotens j wird nur auf einem einzigen Prozessor berechnet.

Typ2-Parallelisierung
In diesem zweiten Fall ist die Frontal-Matrix des Knotens so groß, dass die Faktorisierung in mehrere Teile aufgespalten wird. Die Zeilen der Frontal-Matrix werden dabei auf mehrere Prozessoren aufgeteilt, wobei ein Prozessor als „Master“ definiert ist und die restlichen als „Slaves“. Wegen der zeilenweisen Aufteilung der Matrix spricht man bei diesem Typ der Parallelisierung auch von der *1D-Block-Partitionierung*. Der Master-Prozess behält dabei die Zeilen der Frontal-Matrix, die bereit sind zur Faktorisierung. Diese Zeilen werden auch als „fully-summed“ also als vollständig summiert bezeichnet, weil keine Update-Beiträge aus anderen Levels mehr mit einfließen. Alle an-

deren Zeilen werden dann den Slave-Prozessoren zugeordnet.
Der Master erhält schließlich zur Ausführungszeit von seinen Söhnen symbolische Information bezüglich der Struktur der Beitragsblöcke, die er von ihnen zur Bildung seiner Frontal-Matrix geschickt bekommt. Basierend auf dieser Information bestimmt der Master die exakte Struktur seiner Frontal-Matrix und entscheidet dann, welcher Slave-Prozessor an der Faktorisierung des Knotens teilnehmen wird. Um diesen Sachverhalt etwas zu verdeutlichen, ist in Abbildung 2.15 eine grafische Darstellung eines solchen Typ2-Knotens gegeben.

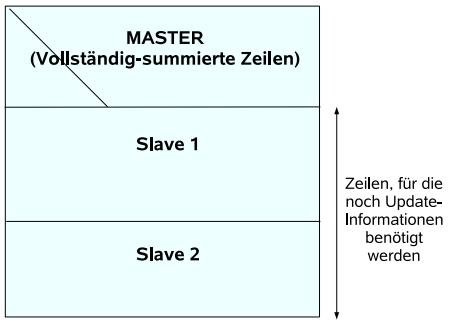


Abbildung 2.15: Typ2-Knoten: Aufteilung der Frontal-Matrix

Typ3-Parallelisierung
Die Frontal-Matrix wird bei Typ3-Parallelisierung nicht wie bei Typ2-Knoten zeilenweise verteilt, sondern zunächst in Blöcke unterteilt. Diese Blöcke bilden nun kleine Teilmatrizen der Frontal-Matrix. Der Master sammelt schließlich wieder symbolische Information aller seiner Söhne bezüglich der noch mit einzubeziehenden Update-Informationen und bildet symbolisch die Struktur seiner Frontal-Matrix.
Diese Information wird daraufhin an alle teilnehmenden Prozessoren versendet. Somit können dann bei der Faktorisierung der Frontal-Matrix die Update-Beiträge der Söhne ganz gezielt an diejenigen Prozessoren gesendet werden, die diese Informationen für ihre Berechnungen auch benötigen. Wegen seiner Eigenschaften ist dieser Prozess auch bekannt unter dem Begriff *Zyklische-2D-Block-Partitionierung*.

Diese drei Typen der Parallelisierung sind in Abbildung 2.16 noch einmal zusammen mit den zugehörigen Frontal-Matrizen dargestellt, um eine deutlichere Vorstellung der verschiedenen Aufteilungen zu erhalten.

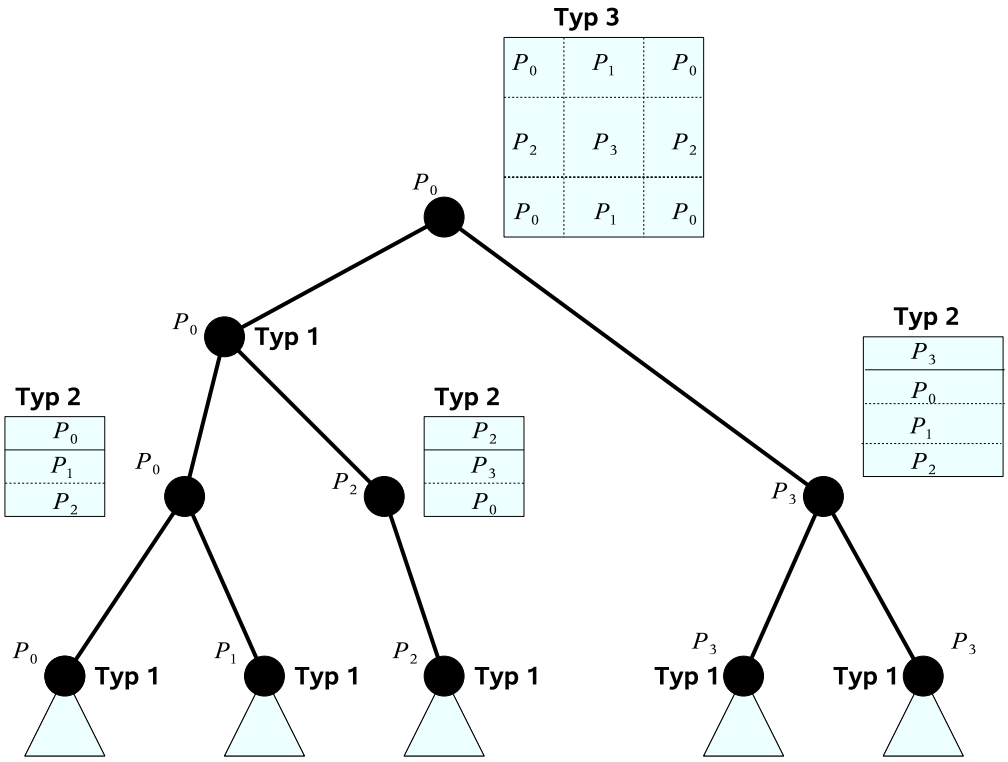


Abbildung 2.16: Aufteilung der Berechnungen eines Eliminationsbaums

Kapitel 3

Das Jacobi-Davidson-Verfahren

In diesem Kapitel wird das Jacobi-Davidson-Verfahren, welches von Sleijpen und Van Der Vorst 1996 (siehe auch [11]) erstmals beschrieben wurde, kurz erläutert mit einem speziellen Augenmerk auf die Lösung der Korrekturgleichung. Im Rahmen der Diplomarbeit sollte an dieser Stelle ein zusätzlicher Löser für lineare Gleichungssysteme mit dünnbesetzter Koeffizientenmatrix, der sogenannte Sparse Matrix Solver MUMPS, herangezogen werden (vgl. Kapitel (2.3)).

Die Grundidee
Das Jacobi-Davidson-Verfahren dient zur Berechnung der größten oder der kleinsten Eigenwerte, beziehungsweise von Eigenwerten aus dem Inneren des Spektrums in der Umgebung eines vorgegebenen Wertes. Der Einfachheit halber beschränkt sich die folgende Erläuterung des Verfahrens auf die Berechnung des größten Eigenwertes einer symmetrischen und reellen Matrix A. Somit ist das folgende Eigenwertproblem gegeben:

$$Au = \lambda u \quad \text{mit} \quad A \in \mathbb{R}^{n \times n} \quad \text{symmetrisch}$$

Mit Hilfe des Jacobi-Davisdon-Algorithmus will man nun den größten Eigenwert λ_{max}, sowie den dazugehörigen Eigenvektor u_{max} von A bestimmen.

Die grundlegende Idee bei diesem Verfahren ist es, die Eingangsmatrix A in einen niederdimensionalen Unterraum $\mathcal{V}$ zu projizieren. Hier wird dann das Eigenwertproblem „direkt gelöst“ und im Anschluss daran der Unterraum geschickt erweitert. Diese Vorgehensweise wird mittels Iteration so oft wiederholt, bis das Residuum schließlich klein genug ist.
Auf die einzelnen Schritte, die dazu nötig sind, wird nun im Folgenden eingegangen.

Der Projektionsschritt
Dazu sei $V = [v_1, ..., v_m] \in \mathbb{R}^{n \times m}$ orthonormal und $\mathcal{V} = span\{v_1, ..., v_m\}$. Die Projektion von A in $\mathcal{V}$ ist gegeben durch:

$$H = V^T A V$$

Im Fall, dass $\mathcal{V}$ einen Eigenvektor u von A zum dazugehörigen Eigenwert λ enthält, besitzt H ebenfalls den Eigenwert λ. Im umgehrten Fall bedeutet dies, falls (θ, s) ein Eigenpaar von H ist, so ist (θ, u) mit $u = Vs$ ein Ritzpaar von A bezüglich $\mathcal{V}$. Das heißt $Au - \theta u$ steht senkrecht zu $\mathcal{V}$ mit $u \in \mathcal{V}$.
Die Ritzwerte bilden dabei Approximationen an die Eigenwerte von A und die Ritzvektoren Näherungen an die Eigenvektoren von A. Je kleiner schließlich der Winkel zwischen dem exakten Eigenvektor und dem Unterraum $\mathcal{V}$ ist, desto besser sind die Approximationen.

Die Unterraumerweiterung
Nach Berechnung der aktuellen Näherungen θ und $u \in \mathcal{V}$ für λ_{max} und u_{max}, zerlegt man den Vektor u_{max} in:

$$u_{max} = u + u^{\perp},$$

wobei $u^{\perp} \in U^{\perp}$ und $U = span\{u\}$ ist. Mit Hilfe des Jacobi-Davidson-Verfahrens wird nun versucht den Vektor $u^{\perp}$ anzunähern und daraufhin den Raum $\mathcal{V}$ um genau diese Näherung zu erweitern. Hierzu projiziert man die Matrix A wie folgt in den Raum $U^{\perp}$:

$$B = (I - uu^T)A(I - uu^T) \tag{3.1}$$

Dabei wird u als normalisiert angenommen.
Mit Hilfe der Definition für das Residuum $r = AU - \theta u$ und der Eigenschaft, dass $r \perp u$, gilt für den Ritzwert θ:

$$\begin{array}{rrcl} & \theta u & = & Au - r \\ \Rightarrow & \theta u^T u & = & u^T Au - u^T r \\ \Rightarrow & \theta & = & u^T Au \end{array} \tag{3.2}$$

Damit kann die Gleichung (3.1) auch geschrieben werden als:

$$\begin{array}{rrcl} & B & = & (I - uu^T)A(I - uu^T) \\ \Leftrightarrow & B & = & (A - uu^T A)(I - uu^T) \\ \Leftrightarrow & B & = & A - uu^T A - Auu^T + u(u^T Au)u^T \\ \Leftrightarrow & A & = & B + uu^T A + Auu^T - \theta uu^T \end{array} \tag{3.3}$$

Weiterhin gilt mit $u_{max} = u + u^{\perp}$ die Bedingung:

$$A(u + u^{\perp}) \quad = \quad \lambda_{max}(u + u^{\perp}) \tag{3.4}$$

Zudem erfüllt die Matrix B die folgende Gleichung:

$$\begin{array}{rcl} Bu & = & (I - uu^T)A(I - uu^T)u \\ & = & (I - uu^T)A(u - u) \\ & = & 0 \end{array} \tag{3.5}$$

Nun folgt aus (3.4) unter Berücksichtigung von (3.2), (3.3), (3.5) und $r = Au - \theta u$:

$$\begin{array}{rl} & (B + uu^T A + Auu^T - \theta uu^T)(u + u^{\perp}) = \lambda_{max}(u + u^{\perp}) \\ \Leftrightarrow & 0 + Bu^{\perp} + u\theta + uu^T Au^{\perp} + Au + 0 - \theta u + 0 = \lambda_{max}u + \lambda_{max}u^{\perp} \\ \Leftrightarrow & Bu^{\perp} - \lambda_{max}u^{\perp} = -(Au - \theta u) + (\lambda_{max} - \theta - u^T Au^{\perp})u \\ \Leftrightarrow & (B - \lambda_{max}I)u^{\perp} = -r + (\lambda_{max} - \theta - u^T Au^{\perp})u \end{array} \tag{3.6}$$

Da $(B - \lambda_{max}I)u^{\perp}$ und r beide senkrecht zu u sind, muss der Vorfaktor von u gleich Null sein. Damit gilt:

$$(B - \lambda_{max}I)u^{\perp} \quad = \quad -r \tag{3.7}$$

Da der Eigenwert λ_{max} nicht bekannt ist, und das Lösen dieser Gleichung ähnlich aufwändig wäre wie das Lösen des Ausgangsproblems, nähert man die Gleichung an, indem man λ_{max} durch θ ersetzt. Dies funktioniert allerdings nur zufriedenstellend gut, wenn θ „nah“ an λ_{max} liegt. Falls dem

nicht so ist, ist es effizienter λ_{max} durch einen konstanten Wert zu ersetzen, der in der Nähe des gesuchten Eigenwertes liegt. Damit kann verhindert werden, dass die Eigenwertnäherung schließlich gegen einen unerwünschten Wert läuft, beziehungsweise, dass eine lokale Stagnation stattfindet.
Angenommen θ ist „nah“ an λ_{max}, so erhalten wir aus (3.7):

$$(B - \theta I)u^{\perp'} = -r$$

Mit (3.1) und der folgenden Gleichung:

$$(I - uu^T)u^{\perp'} \quad = \quad u^{\perp'} \tag{3.8}$$

führt dies zu dem für die Diplomarbeit wichtigsten Abschnitt, zu der Korrekturgleichung:

$$\begin{aligned}
& [(I - uu^T)A(I - uu^T) - \theta I]u^{\perp'} &&= && -r \\
\Leftrightarrow \quad & (I - uu^T)A(I - uu^T)u^{\perp'} - \theta I u^{\perp'} &&= && -r \\
\Leftrightarrow \quad & (I - uu^T)Au^{\perp'} - \theta(I - uu^T)u^{\perp'} &&= && -r \\
\Leftrightarrow \quad & (I - uu^T)(A - \theta I)u^{\perp'} &&= && -r \\
\Leftrightarrow \quad & (I - uu^T)(A - \theta I)(I - uu^T)u^{\perp'} &&= && -r
\end{aligned} \tag{3.9}$$

Ist ein Ritzwert θ gegeben, so wird mit der Gleichung (3.9) ein $u^{\perp'}$ berechnet, mit welchem daraufhin der Unterraum erweitert wird. Anschließend berechnet man wieder ein neues Ritzpaar bezüglich des erweiterten Unterraums. Diese Schritte werden so lange wiederholt, bis die neue Eigenwertnäherung exakt genug ist.
An der Gleichung (3.9) lässt sich jedoch leicht erkennen, dass die Bestimmung von $u^{\perp'}$ bei großen Matrizen sehr aufwändig wird. Aus diesem Grund ist es erforderlich, die erhaltene Gleichung noch zu vereinfachen. Wie das realisiert wird und welche Vorteile daraus entstehen ist explizit im folgenden Kapitel 4 erläutert.

In der Abbildung (3.1) ist das bisher beschriebene Jacobi-Davidson-Verfahren in einem Nassi-Shneidermann-Diagramm dargestellt. Wie auch in der Beschreibung des Verfahrens ist man mit dieser vereinfachten Form des Algorithmus nur in der Lage, den größten Eigenwert λ_{max} von A anzunähern. Zusätzlich wird noch eine Variable m_{max} eingeführt. Mit deren Hilfe wird sichergestellt, dass der Unterraum nicht beliebig oft erweitert wird, da es dann zu Problemen kommen kann. Zum einen kann der Speicherplatz eines Rechners überschritten werden und zum anderen wird das Lösen des projizierten Eigenwertproblems mit wachsendem Unterraum immer aufwändiger. Um diese Ineffektivität zu vermeiden, wird ein Restart durchgeführt, wenn m_{max} überschritten wird. Eine ausführliche Erläuterung zum Thema Restart ist in [11] zu finden.

Die Verallgemeinerung

Die Algorithmen zum Jacobi-Davidson-Verfahren, die man in verschiedener Literatur findet, beschreiben das Verfahren oft allgemeiner. So werden dort die größten oder kleinsten n Eigenwerte einer Matrix bestimmt, wobei die Matrix auch nicht zwangsläufig symmetrischen Ursprungs sein muss. Zudem können mit Hilfe harmonischer Ritzwerte auch Eigenwerte aus dem Inneren des Spektrums ermittelt werden. Auch zur Lösung der Korrekturgleichung werden einige verschiedene Methoden dargestellt.

Wähle einen normierten Startvektor v
Wähle ein m_{max} für die maximale Unterraumerweiterung
Setze $V := [v]$ und $m := 1$
berechne Projektion $H := V^T AV$
berechne größtes Eigenpaar (θ, s) von H
setze den Ritzvektor $u := Vs$
setze den Residuumsvektor $r := Au - \theta u$
löse näherungsweise ein t aus $(I - uu^T)(A - \theta I)(I - uu^T)t = -r$ mit $t \perp u$
orthonormalisiere t gegenüber V und füge Ergebnis zu V hinzu
setze $m := m + 1$
$m > m_{max}$ (Y / N)
führe einen Restart durch mit $V := [u]$
Solange bis Konvergenz eintritt

Abbildung 3.1: Einfache Form des Jacobi-Davidson-Algorithmus zur Berechnung des größten Eigenwerts von A

Kapitel 4

Integration des Sparse Matrix Solvers MUMPS in das Jacobi-Davidson-Programm

4.1 Lösung der Korrekturgleichung

Wie in Kapitel 3 beschrieben, ist es erforderlich die Korrekturgleichung (3.9) in jedem Iterationsschritt des Jacobi-Davidson-Verfahrens zu lösen. Aus diesem Grund sollte versucht werden, dieses Problem so zu vereinfachen, dass die Lösung möglichst einfach und somit effizient zu bestimmen ist. Hierzu wird in Gleichung (3.9) die Vereinfachung (3.8) eingesetzt:

$$\begin{aligned} & (I - uu^T)(A - \theta I)(I - uu^T)u^{\perp'} &=& -r \\ \Leftrightarrow \quad & (I - uu^T)(A - \theta I)u^{\perp'} &=& -r \\ \Leftrightarrow \quad & (A - \theta I)u^{\perp'} &=& -r + \alpha u \end{aligned} \tag{4.1}$$

Da $u^{\perp'}$ zu diesem Zeitpunkt unbekannt ist, kann α nur durch rechenintensive Umformungen bestimmt werden.
Betrachtet man die letzte Gleichung etwas genauer, bemerkt man auf der rechten Seite eine Linearkombination zweier Vektoren r und u. Diese Linearkombination lässt sich mit zwei entsprechend großen Spaltenvektoren, x und y, ersetzen:

$$(A - \theta I)(x, y) \;=\; (u, r)$$

Daraus folgt:

$$(A - \theta I)x \;=\; u \quad \Leftrightarrow \quad x \;=\; (A - \theta I)^{-1}u \tag{4.2}$$

$$(A - \theta I)y \;=\; r \quad \Leftrightarrow \quad y \;=\; (A - \theta I)^{-1}r \tag{4.3}$$

Auf diese Weise hat man die Gleichung (4.1) auf **zwei** einfache lineare Gleichungssysteme zurückgeführt.

Mit (4.2), (4.3) und der Gleichung (4.1) erhält man letztendlich:

$$\begin{aligned} & (A - \theta I)\, u^{\perp'} &=& -r + \alpha u \\ \Leftrightarrow \quad & u^{\perp'} &=& -(A - \theta I)^{-1}r + \alpha(A - \theta I)^{-1}u \\ \Leftrightarrow \quad & u^{\perp'} &=& \alpha x - y \end{aligned} \tag{4.4}$$

Da für das $u^{\perp'}$ gefordert ist, dass es senkrecht auf u steht, kann aus (4.4) schließlich der Wert für α bestimmt werden:

$$\begin{aligned} & u^T u^{\perp'} &=& u^T \alpha x - u^T y \\ \Leftrightarrow \quad & 0 &=& \alpha u^T x - u^T y \\ \Leftrightarrow \quad & \alpha &=& \frac{u^T y}{u^T x} \end{aligned} \tag{4.5}$$

Da die linke Seite der Gleichungen (4.2) und (4.3), sprich $(A - \theta I)$, in der Praxis meist eine große, dünnbesetzte Matrix darstellt, sollte innerhalb der Diplomarbeit getestet werden, ob sich die Effizienz des Jacobi-Davidson-Verfahrens dadurch steigern lässt, dass zur Lösung der beiden Gleichungssysteme statt den bisherigen CG-artigen Lösern, die in Kapitel 2 vorgestellt wurden, der Sparse Matrix Solver MUMPS verwendet wird. Speziell bei der Berechnung von kleinsten und inneren Eigenwerten traten in der Vergangenheit oftmals mit diesen Verfahren Konvergenzprobleme auf, die durch Verwendung des MUMPS-Lösers beseitigt werden sollten.
In den folgenden Abschnitten dieses und des nächsten Kapitels wird nun genauer auf diese Sachverhalte eingegangen. Dazu beschreibt das nächste Unterkapitel 4.2 zunächst die Funktionsweise und die Möglichkeiten des neu eingebundenen Lösers MUMPS. In 4.3 wird danach erläutert, an welcher Stelle der Löser in das bestehende parallele Eigenwertprogramm eingebaut wurde. Abschließend soll in Kapitel 6 eine ausführliche Performance-Analyse zeigen, ob sich mit Hilfe des Lösers MUMPS die Effizienz des Programms hat steigern lassen.

4.2 Die Software-Bibliothek MUMPS

Einleitung
MUMPS (MUltifrontal Massively Parallel Solver), ist eine Software-Bibliothek, die auf der Multifrontal Methode basiert (vergleiche Kapitel 2.3). Dieses Paket dient zum Lösen linearer Gleichungssysteme der Form

$$Ax = b$$

bei denen A eine quadratische, dünnbesetzte Matrix ist. Des Weiteren kann die Koeffizientenmatrix unsymmetrisch, symmetrisch positiv definit, oder generell symmetrisch sein. Die Faktorisierung der Matrix A mit Hilfe des direkten Verfahrens kann auf zwei verschiedene Arten erfolgen, mittels LU- oder LDL^T-Zerlegung, abhängig davon, ob die Originalmatrix unsymmetrisch oder symmetrisch ist.

Der Algorithmus zur Lösung wird, wie auch schon in Kapitel 2.3 beschrieben, in drei Phasen durchgeführt:

1. Analyse
2. Faktorisierung
3. Lösung

MUMPS verteilt dabei die nötigen Berechnungen zur Lösung des Gleichungssystems auf die zur Verfügung stehende Anzahl an Prozessoren. So wird zu Beginn ein Prozessor als „Host“ definiert, der im Anschluß den gesamten Lösungsprozess steuern wird. Die Aufgaben der drei Phasen sind wie folgt unterteilt:

1. Zunächst führt der Host in der ersten Phase ein Ordering der Matrix basierend auf der symmetrischen Matrix $A + A^T$ durch, wie es auch schon im Kapitel 2.3 erläutert wurde. Zudem ermittelt der Host über die symbolische Faktorisierung den anfallenden Rechenaufwand für die in der Faktorisierungs-Phase zu berechnenden Frontal-Matrizen. Nach der Erstellung des Eliminationsbaumes und dem Mapping auf die Prozessoren, wird die entsprechende symbolische Information vom Host an alle anderen Prozessoren versendet. Anhand dieser In-

formation kann nun der von den Prozessoren für die Faktorisierung und Lösung benötige Speicherplatz abgeschätzt werden.

2. Zu Beginn der Faktorisierungs-Phase wird die Originalmatrix auf die teilnehmenden Prozessoren verteilt. Die numerische Faktorisierung einer jeden Frontal-Matrix wird daraufhin von einem Master-Prozessor (während der Analyse-Phase ermittelt durch Host) und einem oder mehreren Slave-Prozessoren (während der Ausführung dynamisch zugeordnet) durchgeführt. Jeder Prozessor legt dazu einen Vektor für die Update- und Faktor-Beiträge an, wobei die Faktor-Beiträge bis zur Lösungs-Phase gespeichert werden.
3. Während der Lösungs-Phase wird die rechte Seite b des Gleichungssystems vom Host an alle anderen Prozessoren übermittelt. Die Prozessoren berechnen daraufhin die gesuchte Lösung x mit Hilfe der in Phase 2 berechneten Faktor-Beiträge. Die Lösung des Problems wird schließlich wieder auf dem Host gesammelt oder kann auch auf den einzelnen Prozessoren verteilt bleiben.

Jede der drei Phasen kann durch Einstellung bestimmter Parameter separat ausgeführt werden, wobei MUMPS dem Host-Prozessor auch erlauben kann, mit an den Berechnungen der Faktorisierung und Lösung Teil zu haben oder nicht.

Mit dem MUMPS-Löser wurde für beide Arten von Matrizen, symmetrische und unsymmetrische, ein Verfahren entwickelt, das auf einer völlig asynchronen Kommunikation mit dynamischer Verteilung der Berechnungsprozesse basiert.
Die asynchrone Kommunikation wird verwendet, damit Berechnungen und Kommunikation parallel stattfinden können. Durch die dynamische Verteilung entsteht der Vorteil, dass sich der Algorithmus sozusagen eigenständig zur Laufzeit berichtigen kann. So können während der numerischen Faktorisierung Berechnungen auf andere, zu der Zeit weniger ausgelastete Prozessoren verteilt werden. Im Endeffekt führt eine Kombination aus statischer und dynamischer Verteilung zum erwünschten Ziel, einer möglichst ausgeglichenen Verteilung von Rechenzeit und Speicherplatzbedarf unter den einzelnen Prozessoren. Dazu werden nur einige der Hauptprozesse anhand der aus der Analyse-Phase ermittelten Erwartungen auf Prozessoren statisch abgebildet. Alle anderen anfallenden Prozesse werden dynamisch zur Laufzeit auf die Prozessoren verteilt.

Des Weiteren besitzt MUMPS eine Vielzahl von Variablen zur Steuerung dieser drei Phasen des Berechnungsprozesses durch den Benutzer. Auf einige der Einstellmöglichkeiten der Software-Bibliothek wir im Folgenden eingegangen. Für einen Gesamtüberblick über das Ausmaß der zur Verfügung stehenden Möglichkeiten steht zum einen der User's Guide [10] zur Verfügung, zum anderen dient der Artikel [5] zur Orientierung auf diesem Gebiet.

Für die Integration eines weiteren Lösers in das parallele Jacobi-Davidson (JD) Programm bot sich die MUMPS-Software-Bibliothek aus mehreren Gründen an. Zum einen ist die Software ebenfalls in Fortran90 implementiert, sodass man nicht auf Schnittstellen zurückgreifen muss, auch wenn diese für C und MATLAB schon existieren. Zum anderen basiert die Nachrichtenübertragung zwischen den Prozessoren bei der parallelen Version von MUMPS auf MPI, so dass auch hier keine Probleme entstehen sollten, da die parallele Version des JD-Programms ebenfalls mittels MPI die Kommunikation zwischen den Prozessoren koordiniert.

Die Datenstruktur von MUMPS

Im Folgenden wird nun zunächst kurz auf die Einstellmöglichkeiten der MUMPS-internen Parameter eingegangen, die für die Einbindung des Solvers in das JD-Programm nötig beziehungsweise hilfreich waren.

Alle von MUMPS zur Ausführungszeit benötigten Variablen sind in einer großen Struktur namens

```
[SDCZ]MUMPS_STRUCT
```

deklariert, wobei jeweils nur einer der in eckigen Klammern aufgeführten Buchstaben vorne angehangen wird, je nachdem, ob MUMPS als REAL-, DOUBLE PRECISION-, COMPLEX- oder DOUBLE COMPLEX-Version ausgeführt werden soll. Auf die einzelnen Komponenten, zum Beispiel auf den Vektor ICNTL, kann dann wie folgt zugegriffen werden:

```
mumps_par_ICNTL
```

Der Einfachheit halber wird im Folgenden dafür aber nur noch der Komponentenname ICNTL verwendet.

Kontrolle über die drei Phasen: Analyse, Faktorisierung, Lösung

In diesem Teil des Kapitels wird nun auf die Komponenten der Variablen mumps_par% vom Datentyp DMUMPS_STRUCT eingegangen, die der Benutzer zur Steuerung von MUMPS setzen muss. Die Variable *mumps_par%job* ist vom Typ Integer und muss vom Benutzer auf allen Prozessoren vor dem Aufruf von MUMPS initialisiert werden. Über die Variable werden die Hauptaufgaben des Solvers gesteuert. Folgende Möglichkeiten bestehen im Umgang mit dieser Variablen.

- *JOB = -1* initialisiert ein Objekt der Struktur MUMPS_STRUCT. Ein Aufruf der MUMPS-Routine mit JOB=-1 ist in jedem Fall einmal zu Beginn durchzuführen. Dadurch werden Default-Werte anderer Komponenten von MUMPS_STRUCT gesetzt, wie zum Beispiel ICNTL, auf die an späterer Stelle noch eingegangen wird. Diese Default-Werte können selbstverständlich im Anschluß wieder verändert werden, bevor die Analyse-Phase von MUMPS stattfindet. Wichtig ist jedoch, dass vor dem Aufruf der Routine MUMPS mit JOB=-1 noch drei weitere Komponenten vom Benutzer initialisiert werden. Diese sind:

  ```
  mumps_par%COMM
  mumps_par%PAR
  mumps_par%SYM
  ```

  Die Integer-Variable *mumps_par%COMM* muss auf einen gültigen MPI-Kommunikator gesetzt werden (z.B. MPI_COMM_WORLD), der zum Nachrichtenaustausch innerhalb von MUMPS gebraucht wird. Der Wert darf während der Berechnung vom Benutzer nicht mehr verändert werden. Der Prozessor in diesem Kommunikator mit Rang 0 wird von MUMPS als Host verwendet und es sollten auch nur Prozessoren aus diesem Kommunikator MUMPS aufrufen.

  Mit Hilfe der Komponente *mumps_par%PAR* besteht die Möglichkeit, dass der Host nicht nur die Analyse-Phase durchführt und die anderen Phasen überwacht. Durch Setzen der Variablen PAR auf 1, statt dem Default-Wert 0, wird der Host mit in die Berechnung der Faktorisierungs- und Lösungs-Phase einbezogen. Ist die Eingangs-Matrix sehr groß und nicht verteilt über die Prozessoren, sondern nur auf dem Host abgelegt, so sollte von einem MUMPS-Aufruf mit PAR=1 abgesehen werden, da es dann schnell zu Unausgeglichenheit im Speicher des Hosts kommen kann, was den Programmablauf wesentlich verlangsamen kann.

  Die Variable *mumps_par%SYM* muss angegeben werden, um MUMPS mitzuteilen welche Art von Matrix verarbeitet werden soll. Dabei stehen folgende Möglichkeiten zur Verfügung:

  SYM = 0 steht für unsymmetrische,
  SYM = 1 für symmetrisch positiv definite, und
  SYM = 2 für symmetrische Matrizen.

  Nach einem Aufruf von MUMPS mit JOB=-1 enthält die interne Komponente mumps_par %MYID jeweils den Rang des aufrufenden Prozessors.

- *JOB=-2* ist das Gegenstück zu JOB=-1 und zerstört somit ein Objekt der MUMPS-Struktur. Alle Variablen, die zu diesem Objekt gehören, bis auf die vom Benutzer angelegten, werden deallozііert. Deshalb sollte MUMPS mit der Option auch nur aufgerufen werden, wenn abzusehen ist, dass keine weiteren Aufrufe mit diesem Objekt mehr stattfinden werden.

- *JOB=1* leitet die Analyse-Phase ein. MUMPS wählt dabei Pivot-Elemente von der Diagonalen und benutzt ein Auswahlverfahren, was dafür sorgt, dass während der anschließenden Faktorisierungs-Phase die Faktor-Matrix nicht unnötig aufgefüllt wird. Zudem verwendet MUMPS die Besetzungs-Struktur von $A + A^T$ dazu, unabhängig von den numerischen Werten den Eliminationsbaum zu bilden und den Aufwand der anfallenden Berechnungen in der Faktorisierungs-Phase abzuschätzen.
  Da die numerischen Werte in der Analyse-Phase noch keine Rolle spielen, reicht es prinzipiell aus, diese MUMPS auch erst nach der ersten Phase zur Verfügung zu stellen. Da in den meisten Fällen die Input-Matrix vor der Analyse schon bekannt ist, werden im Folgenden nur die Unterschiede bei der Eingabe der Matrix beschrieben. So ist es einerseits möglich, die Matrix auf dem Host abzuspeichern, oder aber verteilt auf den einzelnen Prozessoren, was sich besonders bei sehr großen Matrizen anbietet. Abhängig ist die Speicherung der Matrix von dem Parameter *ICNTL(18)*, über den genau diese beiden Möglichkeiten steuerbar sind. Besitzt die Variable den Default-Wert 0, so muss der Benutzer die Komponenten *N, NZ, IRN, JCN* und *A* der MUMPS-Struktur eigenhändig definieren. N und NZ speichern dabei die Dimension und die Anzahl der Nicht-Null-Einträge der Originalmatrix. IRN und JCN sind jeweils Vektoren der Länge NZ, auf denen anschließend die Zeilen- und Spaltenindizes der Matrixeinträge abgelegt werden. Auf dem Vektor A, der gleichen Länge sind die Nicht-Null-Einträge gespeichert.
  Soll die Matrix verteilt abgelegt werden, ordnet man der Komponente *ICNTL(18)* vor der Analyse-Phase den Wert 3 zu. Diese Änderung beinhaltet jedoch auch, dass nun nicht mehr nur auf dem Host wie oben die zusätzlichen Komponenten definiert werden, sondern auf jedem einzelnen der Prozessoren. Die einizige Variable, die unverändert auf dem Host definiert wird, ist die Matrixdimension *N*. Alle anderen Komponenten ändern sich wie folgt. Auf jedem Prozessor müssen *NZ_loc, IRN_loc, JCN_loc* und *A_loc* definiert werden. NZ_loc gibt dabei die Anzahl von Zeilen der Originalmatrix lokal auf dem entsprechenden Prozessor an. IRN_loc, JCN_loc und A_loc sind wieder Vektoren, nun aber der Länge NZ_loc, auf denen jeweils die lokalen Zeilen- und Spaltenindizes, sowie die Matrixeinträge der entsprechenden Nicht-Null-Elemente abgespeichert werden.

- Mit *JOB=2* wird die zweite Phase gestartet, die numerische Faktorisierung der Matrix. Falls für den Aufruf von MUMPS mit JOB=1 nur die Struktur der Matrix, sprich N und NZ, IRN, JCN beziehungsweise NZ_loc, IRN_loc, JCN_loc, übergeben wurde, müssen spätestens jetzt vor dem Aufruf mit JOB=2 auch die numerischen Werte A beziehungsweise A_loc vom Benutzer bereitgestellt werden. Logischerweise muss ein Aufruf von MUMPS mit JOB=1 zuvor stattgefunden haben, damit die Information der symbolischen Faktorisierung bereitsteht.

- Die Lösungs-Phase wird mit *JOB=3* begonnen. Um das Gleichungssystem Ax = b zu lösen, werden die rechte Seite, die zuvor vom Benutzer bereitgestellt werden muss, und die Faktoren, die während der zweiten Phase durch MUMPS berechnet wurden, benötigt. Die rechte Seite des Systems wird definiert durch die Komponente *mumps_par%RHS*. RHS wird dabei auf dem Host definiert als ein Feld der Länge $(NRHS * LRHS)$, wobei NRHS die Anzahl der Vektoren für die rechte Seite angibt und LRHS die Dimension der Originalmatrix. Das heißt, dass MUMPS für mehrere rechte Seiten nur einmal aufgerufen werden muss, wobei der Vektor RHS dann entsprechend verlängert wird. Da nach Lösung des Gleichungssystems für gewöhnlich die rechte Seite nicht weiter von Bedeutung ist, wird default-mäßig (*ICNTL(21)=0*) RHS überschrieben mit der Lösung des Systems. Ist *ICNTL(21)* mit 1 initia-

lisiert, wird die rechte Seite beibehalten. Die Lösung des Gleichungssystems wird dann auf lokalen Vektoren, die auf allen teilnehmenden Prozessoren vom Benutzer manuell angelegt werden müssen, verteilt.

- In dem Fall, dass zu Beginn oder zum Schluß einzelner Phasen von MUMPS bereits alle vom Benutzer bereitzustellenden Daten angegeben wurden, müssen noch folgenden Aufrufe zum Einleiten der nächsten Phasen nicht mehr einzeln erfolgen. Auf folgende Art und Weise können mit der Variablen *JOB* Aufrufe von mehreren Phasen zusammengefasst werden.
  - *JOB=4* kombiniert die Aktionen von JOB=1 und JOB=2.
    Ein Aufruf von MUMPS mit JOB=-1 muss zuvor schon stattgefunden haben.
  - *JOB=5* kombiniert die Aktionen von JOB=2 und JOB=3.
    Ein Aufruf von MUMPS mit JOB=1 muss zuvor schon stattgefunden haben.
  - *JOB=6* kombiniert die Aktionen von JOB=1, JOB=2 und JOB=3.
    Ein Aufruf von MUMPS mit JOB=-1 muss zuvor schon stattgefunden haben.

Zu der MUMPS-Struktur zählen neben den bereits vorgestellten Komponenten noch eine ganze Reihe weiterer, die zum Teil vom Benutzer modifiziert werden können, sodass die Bibliothek abhängig vom zu lösenden Problem individuell einsetzbar ist.
In der MUMPS-Version 4.6.2., die für diese Diplomarbeit eingesetzt wurde, enthält der Vektor *ICNTL* insgesamt 22 Felder mit unterschiedlicher Bedeutung. Zusätzlich zu den oben schon erwähnten Möglichkeiten zur Manipulation der Programmausführung kann zum Beispiel mit Hilfe dieses Vektors auch die Ausgabe von Zwischenschritten und -lösungen erlaubt oder unterdrückt werden. Weiterhin sind die Angabe von erwünschten Skalierungs- und Ordering-Strategien möglich, sowie die Ausgabe der in der Faktorisierungs-Phase berechneten Schur-Komplement-Matrix. Für detaillierte Informationen steht der User's Guide [10] zur Verfügung.

In Abbildung 4.1 ist nun abschließend noch ein kleines Fortran-Beispielprogramm dargestellt. Anhand dessen sollen ein paar der oben beschriebenen Möglichkeiten, die die Software-Bibliothek bereitstellt, erläutert werden.
Mit diesem Programm wird ein kleines symmetrisches lineares Gleichungssystem gelöst, mit Koeffizienten-Matrix $A = diag(12)$ und rechte Seite $RHS = [14]^T$. Wegen der sehr geringen Größe bleiben die Daten selbstverständlich nur lokal auf dem Host und werden nicht auf teilnehmende Prozessoren verteilt. Damit die MUMPS-Struktur benutzt werden kann, sowie auch benötigte MPI-Routinen, werden zu Beginn die beiden Header-Dateien *mpif.h* und *dmumps_struct.h* eingebunden. DMUMPS steht dafür, wie oben schon erläutert, dass MUMPS in der DOUBLE PRECISION Version ausgeführt werden soll.

4.3 Einbindung in das Jacobi-Davidson-Eigenwertprogramm

Dieser Teil des Kapitels beschreibt nun abschließend, wie die Software-Bibliothek MUMPS in das Jacobi-Davidson-Programm eingebunden wird.

Das Jacobi-Davidson-Programm
Das bestehende Eigenwertprogramm wurde bewusst einfach gehalten, sodass an vielen Stellen ohne großen Aufwand Änderungen durchgeführt werden können. Im Zuge der Diplomarbeit [18] wurde das Programm-Paket derart modularisiert, dass es im Wesentlichen in zwei Teile aufgespalten wurde, in ein Hauptprogramm und ein Hauptunterprogramm. Durch das Hauptprogramm wird der gesamte Programmablauf gesteuert und im Hauptunterprogramm der eigentliche Jacobi-Davidson-Algorithmus ausgeführt. Eine ausgelagerte Routine, namens $solcorreq$ übernimmt im Laufe des Hauptunterprogramms die Lösung der Korrekturgleichung. Da gerade die Lösung der Korrektur-

```
                PROGRAM DMUMPS_TEST

      IMPLICIT NONE
      INCLUDE 'mpif.h'
      INCLUDE 'dmumps_struc.h'
      TYPE (DMUMPS_STRUC) mumps_par
      INTEGER IERR, I

      CALL MPI_INIT(IERR)

      mumps_par%COMM = MPI_COMM_WORLD
      mumps_par%SYM =  1
      mumps_par%PAR =  1
      mumps_par%JOB = -1

      CALL DMUMPS(mumps_par)

      IF ( mumps_par%MYID .eq. 0 ) THEN
         mumps_par%N = 2
         mumps_par%NZ = 2
         ALLOCATE( mumps_par%IRN ( mumps_par%NZ ) )
         ALLOCATE( mumps_par%JCN ( mumps_par%NZ ) )
         ALLOCATE( mumps_par%A( mumps_par%NZ ) )
         ALLOCATE( mumps_par%RHS ( mumps_par%N  ) )
         mumps_par%IRN = (/1,2/)
         mumps_par%JCN = (/1,2/)
         mumps_par%A = (/1.0, 2.0/)
         mumps_par%RHS = (/1.0, 4.0/)
      END IF

      mumps_par%JOB = 6
      CALL DMUMPS(mumps_par)

C     Solution has been assembled on the host
      IF ( mumps_par%MYID .eq. 0 ) THEN
         WRITE( * , * ) ' Solution is ',
     1        (mumps_par%RHS(I),I=1,mumps_par%N)
      END IF

C     Deallocate user data
      ...

C     Destroy the instance (deallocate internal data structures)
      mumps_par%JOB = -2
      CALL DMUMPS(mumps_par)
      CALL MPI_FINALIZE(IERR)
      END
```

Abbildung 4.1: Lösung eines Gleichungsystems mit der Software-Bibliothek MUMPS

gleichung einen großen Einfluss auf die Konvergenzgeschwindigkeit des Jacobi-Davidson-Verfahrens ausübt, wurden bisher schon verschiedene Methoden angeboten, die je nach gesuchten Eigenwerten und abhängig von der Art der Input-Matrix A auf verschiedene Art und Weise die Gleichungssysteme lösen. Zunächst wurden die bisher implementierten Verfahren eingeteilt in einfache und komplexe Lösermethoden. Die einfachen Löser haben dabei die Eigenschaft, dass die Iterationen sehr schnell laufen, die Konvergenz jedoch sehr langsam ist. Die komplexen Löser bilden das genaue Gegenteil dazu, sie sind sehr präzise, dadurch aber auch sehr aufwändig. Ergebnis ist, dass die Iterationen sehr viel länger dauern, als die der einfachen Löser, dafür jedoch wesentlich weniger

Iterationen benötigt werden. Es bietet sich daher an, zunächst mit einem schnellen schlechten Löser zu beginnen und nach einer gewissen Anzahl von Iterationen auf den langsamen aber präziseren Löser zu wechseln.
Im bisherigen Jacobi-Davidson-Programm standen diese beiden Möglichkeiten mittels den beiden Prozeduren $simplesolv$ und $complsolv$ zur Verfügung, wobei als einfacher Löser die Multiplikation mit der Inversen der Diagonal-, beziehungsweise mit einer inneren Bandmatrix verwendet wurde. Als komplexe Löser dienten die in Kapitel 2.2 schon vorgestellten CG-artigen Methoden, das QMR- und TFQMR-Verfahren. In $solcorreq$ wird anhand von eingelesenen Parametern aus einer für die Ausführung des Jacobi-Davidson-Programms erforderten Parameterdatei entschieden, welcher Löser letztendlich verwendet werden soll. Weitere Informationen und eine ausführliche Beschreibung bezüglich des gesamten Jacobi-Davidson-Programms und der definierbaren Parameter ist in [18] zu finden.

Während dieser Diplomarbeit wurde nun eine weitere Prozedur namens $mumpssolv$ mit in die Routine $solcorreq$ eingebunden. Wie der Name schon verrät, findet hierbei die Lösung der Gleichungssysteme, mittels der Software-Bibliothek MUMPS statt, die bereits im vorherigen Unterkapitel vorgestellt wurde.

Mit dieser neuen Prozedur $mumpssolv$ sollten letztendlich die Vorüberlegung zur Lösung der Korrekturgleichung aus dem Unterkapitel 4.1 mit den Möglichkeiten des Löser-Pakets MUMPS, die im Teil 4.2 vorgestellt wurden, bestmöglich vereint werden.

Mittels der beiden Gleichungen (4.2) und (4.3) war die Lösung der Korrekturgleichung auf die Berechnung von zwei einfachen Gleichungssystemen zurückgeführt worden. Zur besseren Übersicht sind die beiden Gleichungen hier noch einmal abgebildet:

$$(A - \theta I)x = u \tag{4.6}$$
$$(A - \theta I)y = r \tag{4.7}$$

Da sich die beiden Gleichungssysteme jedoch lediglich in ihrer rechten Seite unterscheiden, muss schließlich nur eine Faktorisierung der Matrix $(A - \theta I)$ stattfinden. Wie zuvor schon in Kapitel 2.1 erläutert wurde, erhält man die Lösung eines zweiten Gleichungssystems, das sich nur in der rechten Seite vom ersten unterscheidet, sehr einfach, da keine zusätzliche Faktorisierung mehr stattfinden muss, sondern lediglich eine zweite Lösungs-Phase.
Im Unterkapitel 4.2 wurde bereits beschrieben, dass die vom Benutzer bereitgestellte rechte Seite *RHS* der MUMPS-Struktur aus mehreren Rechte-Seite-Vektoren bestehen darf. In diesem Fall wird sie aus den beiden Vektoren u und r der Gleichungen (4.6) und (4.7) gebildet. Diese beiden Vektoren liegen jedoch zu Beginn des Aufrufes von $mumpssolv$ noch verteilt auf allen Prozessoren, müssen aber laut Definition der MUMPS-Bibliothek auf dem Host gesammelt werden. Da zusätzlich die einzelnen Prozessoren auch eine verschiedene Anzahl von Zeilen dieser beiden Vektoren in ihrem Speicher bereithalten, jeder nämlich genau nnp Zeilen, muss zum Sammeln der Daten auf die MPI-Routine *mpi_gatherv* zurückgegriffen werden. Diese Routine ermöglicht ein Sammeln von Vektoren unterschiedlicher Länge von den Prozessoren innerhalb eines Kommunikators auf einem festgelegten Prozessor. Eine ausführliche Beschreibung der zur Einbindung des MUMPS-Lösers benötigten MPI-Routinen erhält man in [24]. Die oben erwähnte Variable nnp wird bereits durch das Jacobi-Davidson-Programm definiert, und zwar während der Verteilung der eingelesenen Originalmatrix A zu Beginn des Hauptprogramms.
Aus der Tatsache, dass bei vielen Problemstellungen der Natur- und Ingenieurwissenschaft die Eigenwerte von zum Teil sehr großen Matrizen berechnet werden sollen, folgt unmittelbar, dass auch die linken Seiten der beiden Gleichungssysteme $(A-\theta I)$ ebenfalls sehr groß werden können. In der Software-Bibliothek MUMPS ist jedoch default-mäßig die Koeffizientenmatrix lediglich auf dem

Host abgelegt. Sprich, um das Problem von zu wenig Speicherplatz auf einem Prozessor zum Abspeichern der linken Seite zu umgehen, muss die Input-Matrix auf alle teilnehmenden Prozessoren verteilt werden. Dazu muss zunächst für die benötigten Komponenten *NZ_loc, IRN_loc, JCN_loc* und *A_loc* der MUMPS-Struktur, auf deren Bedeutung in 4.2 bereits eingegangen wurde, Speicherplatz angelegt werden, bevor auf jedem Prozessor genau NZ_loc Elemente der Matrix abgespeichert werden können.
Anschließend sind alle von MUMPS benötigten Daten zur Lösung des Gleichungssystems bekannt. Das heißt, MUMPS kann alle drei Phasen, Analyse, Faktorisierung und Lösung, nacheinander ohne Unterbrechung durchlaufen. Dazu wird die Komponente *JOB* mit 6 initialisiert.

Da die Lösung nun ausschließlich auf dem Host bekannt ist, jedoch alle Prozessoren diese Information benötigen, müssen die Daten im Anschluß wieder verteilt werden. Die geschieht mit dem Gegenstück zu *mpi_gatherv*, nämlich mit *mpi_scatterv*. Diese Routine ermöglicht es, ausgehend von einem Prozessor eine unregelmäßige Anzahl von Elementen an verschiedene Prozessoren zu senden, auch hier werden von jedem der beiden Lösungsvektoren wieder nnp Zeilen an die jeweiligen Prozessoren versendet. Die beiden Lösungsvektoren x und y sind nun berechnet und liegen verteilt auf den teilnehmenden Prozessoren. Das heißt, im Anschluss muss die noch unbekannte Konstante α aus Gleichung (4.5) bestimmt werden, um schließlich das erwünschte $u^{\perp'}$ aus Gleichung (4.4) berechnen zu können.
Bei der Bestimmung von

$$\alpha = \frac{u^T y}{u^T x}$$

kann nun jeder Prozessor die lokal verfügbaren Daten zur Berechnung des Zählers und Nenners benutzen. Im Anschluss daran sendet jeder Prozessor die von ihm lokal errechnete Zähler- und Nenner-Summe an alle Prozessoren im Kommunikator mit der Routine *mpi_allreduce*, die an dieser Stelle bewirkt, dass jeder Prozessor seine und die Daten der anderen Prozessoren aufsummiert. So besitzen anschließend alle Prozessoren die Werte für $u^T y$ und $u^T x$ und können durch einfache Division das gesuchte α bestimmen.
Mit dieser Konstanten kann zuletzt lokal auf jedem der teilnehmenden Prozessoren das erwünschte Ergebnis

$$u^{\perp'} = \alpha x - y$$

berechnet werden. Jeder einzelne Prozessor hat schließlich wieder nnp Zeilen des Ergebnisvektors in seinem Speicher abgelegt.

Weiter oben wurde bereits angemerkt, dass pro Aufruf von MUMPS je zwei lineare Gleichungssysteme gelöst werden müssen, die sich jedoch nur in ihrer rechten Seite unterscheiden. Unabhängig davon unterscheiden sich aber die linken Seite von Aufruf zu Aufruf von MUMPS, denn die jeweilige Eigenwertnäherung wird stets von der Diagonalen der Matrix A abgezogen. Das heißt, um eine exakte Lösung der Korrekturgleichung zu erhalten, müsste bei jedem Aufruf der Routine MUMPS die Matrix $(A - \theta I)$ erneut faktorisiert werden.

Im einführenden Kapitel 2 über die Gleichungssystem-Löser wurde bereits erläutert, dass bei direkten Lösungsmethoden speziell die Faktorisierung der Matrix sehr aufwändig ist.

Während Testläufen, die auch im folgenden Kapitel 6 dokumentiert sind, wurden die Ausführungszeiten für eine Faktorisierung der Systemmatrix $(A - \theta I)$ und für eine Lösung des Gleichungssystems nach erfolgreicher Faktorisierung isoliert betrachtet. Diese Messung ergab eine durchweg gleich bleibende Zeitdifferenz mit einem Faktor von mindestens 1000 zwischen den beiden Phasen. Das hieße also, dass zeitlich gesehen während einer Faktorsierung mehr als 1000 Lösungen-Phasen durchlaufen werden könnten. Des Weitereren ergab diese Analyse, dass die Verbesserung des Er-

gebnis nach einer erneuten Faktorisierung wesentlich schlechter ausfiel als die Verbesserung des Ergebnis nach entsprechend vielen Lösungs-Phasen, die gleich viel Zeit in Anspruch nahmen.

Dieses Ergebnis führte schließlich zu der Entscheidung die Prozedur MUMPS derart zu organisieren, dass lediglich beim ersten Aufruf, die ersten beiden Phasen des MUMPS-Lösers, sprich die Analyse- und Faktorisierungs-Phase, durchlaufen werden. Die berechneten Daten aus der Analyse- und Faktorisierungs-Phase bleiben von da an unverändert auf jedem Prozessor gespeichert. Bei jedem weiteren Aufruf von $mumpssolv$ wird nur noch mit den entsprechenden, neuen, rechten Seiten die Lösungs-Phase durchlaufen. Das bedeutet, es werden nur noch die rechten Seiten auf dem Host gesammelt und anschließend wird MUMPS mit *JOB=3* aufgerufen. Die Berechnungen für α und $u^{\perp'}$ bleiben dabei unverändert.

Kapitel 5

Benutzung des Programms

Die Software-Bibliothek MUMPS

Die Software-Bibliothek MUMPS, Version 4.6.3, auf die bereits in Kapitel 4.2 ausführlich eingegangen wurde, ist ein frei verfügbares Software-Paket und im Internet unter [2] erhältlich. Nach dem Ausfüllen eines Anfrageformulars erhält man per E-Mail das gewünschte Software-Paket.

Die in dem Paket enthaltene Readme-Datei sollte als erstes sorgfältig gelesen werden, denn dort sind alle Schritte ausführlich erläutert, die nötig sind, um die Software auf dem jeweiligen Rechner zu installieren. Ein kurzer Überblick über diese Schritte soll hier nun gegeben werden.
Da die Distribution für verschiedene Rechnerarchitekturen implementiert wurde, muss zunächst ein passendes Makefile generiert werden, um den Quellcode von MUMPS übersetzen zu können. Dazu stehen in dem Verzeichnis *Make.inc* verschiedene vordefinierte Makefiles zur Verfügung, die abhängig von der jeweiligen Architektur ausgewählt werden müssen. Das Makefile benötigt zur Generierung des ausführbaren Programms zusätzlich noch die Angabe über die gewünschte Präzision, mit der später MUMPS ausgeführt werden soll. Zur Auswahl stehen dabei, einfache und doppelte Genauigkeit, jeweils für das Rechnen mit reellen oder auch mit komplexen Zahlen. Entsprechend müssen die Ausdrücke *simple, double, cmplx* oder *cmplx16* dem Aufruf von *make* mit übergeben werden. Wird nur *make* ausgeführt ohne Angabe von einer Präzision, so wird der Quellcode standardmäßig für das Rechnen mit reellen Zahlen bei doppelter Genauigkeit übersetzt, mit *make all* wird der gesamte zum Software-Paket gehörende Code kompiliert.

Nach dem Übersetzen enthält das Verzeichnis *include* jegliche Header-Dateien, die von MUMPS benötigt werden, und im Verzeichnis *lib* sind alle generierten MUMPS-Bibliotheken zu finden. Ein kleines Testbeispiel liegt im Verzeichnis *test*, welches zum einen zeigt, wie MUMPS nun verwendet und aufgerufen werden kann, und zum anderen zur Überprüfung dient, ob das Übersetzen des Quellcodes erfolgreich war.

Auch wenn die parallele MPI-Version von MUMPS auf einem einzelnen Prozessor funktioniert, besteht zudem die Möglichkeit mit einem entsprechenden Makefile (ebenfalls bereitgestellt in dem Verzeichnis *Make.inc*) eine rein sequentielle Version von MUMPS zu generieren. Dies ist dann sinnvoll, wenn man darauf verzichten möchte, alle benötigten parallelen Bibliotheken, wie zum Beispiel MPI und ScaLAPACK, auf dem Einprozessor-System zu installieren. In diesem Fall liegt in dem Verzeichnis *libseq* eine Dummy-MPI-Bibliothek bereit, die alle zur parallelen Bibliothek gehörigen Symbole, die während der Link-Phase benötigt werden, definiert.

Das Jacobi-Davidson-Programm und der MUMPS-Löser

Das Jacobi-Davidson-Programm, welches im Laufe der Diplomarbeit [18] von René Puttin parallelisiert und modularisiert wurde, wird über viele verschiedene Parameter gesteuert. Neben den Default-Einstellungen, die innerhalb des Programms gesetzt werden, können weitere Parameter

auch über eine Parameterdatei *jada15.par* definiert werden. Auf diese Weise kann das Programm so ausgeführt werden, wie es der Benutzer wünscht.

In der nachstehenden Tabelle 5.1 sind einige der Optionen aufgelistet, die in der Parameterdatei angegeben werden können. Es werden hier jedoch nur diejenigen Parameter vorgestellt, die für die Performance-Analyse in Kapitel 6 zum Einsatz gekommen sind. Weitere Informationen zu diesem Thema sind in [18] zu finden. Zusätzlich sei noch angemerkt, dass sich der Aufbau der Parameterdatei im Laufe der Weiterentwicklung des Programms ein wenig verändert hat. So werden nun nicht mehr alle Parameter in der Parameterdatei angegeben, sondern, wie oben schon angemerkt, lediglich solche, die abweichend von den Default-Einstellungen gesetzt werden sollen.

Option	Name	mögliche Werte
Art der gesuchten Eigenwerte	whicheigv	0 = kleinste, 1 = größte, 2 = innere
Anzahl Eigenwerte	numevs	$\leq$ max. Anzahl Iter. vor Restart
Löser für Korrekturgleichung	precon	1 = TFQMR, 5 = MUMPS
Anz. Iterationen vor Restart	itm	≥ 1
Bildschirmausgabe-Level	outlev	0 = keine Ausgabe 1 = nur Endergebnis 4 = alles wird ausgegeben
Inkrementierung der Anz. an Löser-Iterationen	itinc	0 = nein 1 = entspr. Iterationsschritt 2 = entspr. letztem Residuum
Ist Matrix symmetrisch ?	mat_sym	0 = nein, 1 = ja
max. Anz. Jacobi-Davidson-Iterationen	jditmax	≥ 1
Bereich für innere Eigenwerte	tau	$\in \mathbb{R}$
Abbruchgrenze: Residuennorm / Anfangsnorm	reseps	$\in \mathbb{R}^{>0.0}$
Dateiname mit der einzulesenden Matrix	filename	Character-String

Tabelle 5.1: Parameter des Jacobi-Davidson-Programms

Die Tabelle 5.1 ist absichtlich in drei Teile unterteilt, da die Einlese-Routine innerhalb des Jacobi-Davidson-Programms unterscheidet, ob der Wert des einzulesenden Parameters eine ganze Zahl oder eine Gleitkomma-Zahl ist oder aber aus einer Zeichenketten besteht.
So bildet der obere Teil der Tabelle diejenigen Optionen ab, deren Wertebereich die natürlichen Zahlen sind. Die Parameter aus dem mittleren Teil werden als Fließkommazahl abgespeichert und als letztes wird der Dateiname als Zeichenkette angegeben, in doppeltem Hochkomma eingeschlossen.

Die Abbildung 5.1 zeigt ein Beispiel für den Aufbau der Parameterdatei *jada15.par*. Die unterschiedlichen, einzulesenden Werte der Parameter werden durch je eine Zeile mit „ # “ und einer abschließenden „ 0 “ voneinander getrennt. Die Namen der Parameter sind mit einfachen Hochkommata anzugeben.

Ursprünglich war das Jacobi-Davidson-Programm ein rein paralleles Programm. Es wurde jedoch im Laufe der Zeit derart weiterentwickelt, dass es sowohl auf einem Parallelrechner als auch auf einem seriellen Rechner ausgeführt werden kann.

Die Eigenwert-Bibliothek ist seit Kurzem über den Internetauftritt des Zentralinstituts für Angewandte Mathematik des Forschungszentrums in Jülich [27] verfügbar. Entsprechende Hinweise zur Installation dieser Bibliothek sind dort ebenfalls zu finden.

```
'whicheigv'          1
'numevs'             10
'precon'             5
'itm'                60
'outlev'             1
'itinc'              2
'mat_sym'            0
'jditmax'            100000
'#########'    0
'tau'                1.67e9
'reseps'             1.d-4
'#########'    0
'filename'           "bcsstk35"
```

Abbildung 5.1: Beispiel für eine Parameterdatei

Nach der Ausführung des entsprechenden Makefiles, Makefile.Linux.SEQ für die Installation auf einem seriellen Linux-Rechner oder Makefile.AIX.PAR für die Installation auf einem parallelen AIX-System, kann das Jacobi-Davidson-Programm wie folgt gestartet werden.

Im seriellen Fall wird das Programm mit

$$./jada15\_g.exe\ [< Parameterdatei >]$$

aufgerufen.

Für die parallele Version ist ein entsprechender Loader für parallele Programme notwendig. So hat der Aufruf für die parallele Version des Jacobi-Davidson-Programms folgende Gestalt:

$$llrun\ -p < n >\ jada15\_g.exe\ [< Parameterdatei >]$$

wobei n die Anzahl der einzusetzenden Prozessoren darstellt.

Bei beiden Aufrufen kann, wie oben schon beschrieben, eine eigene Parameterdatei mit übergeben werden. Wird hier keine Datei angegeben, so benutzt das Programm standardmäßig die Datei *jada15.par*. Deshalb sollte diese Datei vor jeder Ausführung des Programms überprüft werden, ob die richtigen Paramter eingestellt worden sind.

Die Wahl des Lösers für die Korrekturgleichung

In der Parameterdatei *jada15.par* ist der jeweilige Löser mit Hilfe der Option *precon* anzugeben, der für das Lösen der Korrekturgleichung innerhalb des Jacobi-Davidson-Verfahrens genutzt werden soll. Wie aus der oben abgebildeten Tabelle 5.1 ersichtlich ist, kann mit dem Wert 5 der neu in die Bibliothek eingebundene MUMPS-Löser gewählt werden und über den Wert 1 wird der CG-artige Löser TFQMR verwendet.

Wann schließlich MUMPS vorzugsweise gegenüber dem bisher verwendeten Löser TFQMR zum Einsatz kommen soll, zeigt die Performance-Analyse im anschließenden Kapitel 6. Dort werden mit Hilfe des Jacobi-Davidson-Programms verschiedene Eigenwerte von unterschiedlichen Matrizen bestimmt, stets mit beiden Lösern. So wird ein fairer Vergleich gewährleistet, der als Fazit die Vor- und Nachteile der beiden Löser aufzeigen soll.

Kapitel 6

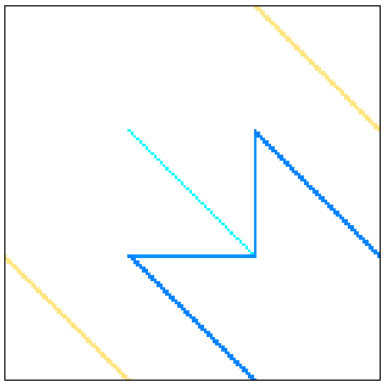
Performance-Analyse

In diesem Kapitel werden zunächst einige dünnbesetzte Matrizen vorgestellt, mit denen im Anschluss das Jacobi-Davidson-Programm getestet wird, um die Performance der beiden Korrekturgleichungslöser, TFQMR und MUMPS, miteinander vergleichen zu können.
Die Testläufe des Programms werden dabei auf dem Supercomputer JUMP des Forschungszentrums Jülich durchgeführt. Nähere Informationen zu der Rechnerarchitektur der pSeries 690 von IBM, zu denen die JUMP gehört, sind unter [26] zu finden.
Als Vergleichskriterium zwischen den beiden Lösern dient die benötigte Rechenzeit für den Jacobi-Davidson-Algorithmus, gemessen mit Hilfe der MPI-Routine *mpi_wtime*. Die Routine und deren Anwendung wird in [24] näher erläutert.

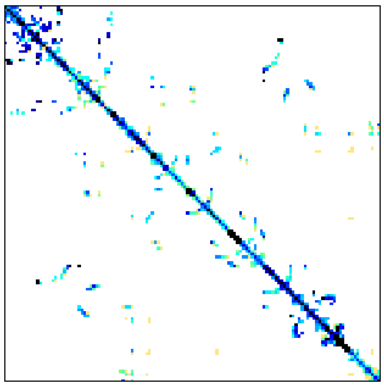
Ziel der Performance-Analyse ist es, herauszufinden, welcher der beiden Löser sich besser und welcher sich weniger gut zur Lösung des jeweiligen Eigenwertproblems mit Matrizen von unterschiedlicher Größe und unterschiedlichem Besetzungsgrad eignet.

Die Testmatrizen
Das Jacobi-Davidson-Programm wird mit sechs verschiedenen, reellen Matrizen getestet. Drei Matrizen sind dabei symmetrisch, zwei symmetrisch positiv definit und eine ist unsymmetrisch.
Bei den Matrizen *bcsstk35*, *cvxbqp1*, *bloweybl*, *thermal2* und *fidapm11* handelt es sich um Matrizen aus der Matrixsammlung der Universität von Florida [28] und von Matrix Market [29]. Alle fünf Matrizen sind dünnbesetzt mit reellen Einträgen und haben jeweils einen eigenen Praxisbezug. Die Matrizen *bcsstk35* und *bloweybl* sind symmetrisch, *cvxbqp1* und *thermal2* sind symmetrisch positiv definit und *fidapm11* ist unsymmetrisch.
Die Abbildung 6.1 zeigt die Strukturplots der einzelnen Matrizen und in der nachstehenden Tabelle sind die Dimensionen der Matrizen, sowie die Anzahl der Nicht-Null-Elemente und die daraus entstehenden Besetzungsgrade aufgeführt.
Die Matrix *kurbel*, die ebenfalls bei den Strukturplots und in der Tabelle zu finden ist, wurde am Zentralinstitut für angewandte Mathematik im Forschungszentrum Jülich erzeugt. Sie beschreibt die Steifigkeit einer Kurbel und ist dünnbesetzt, reell und symmetrisch.

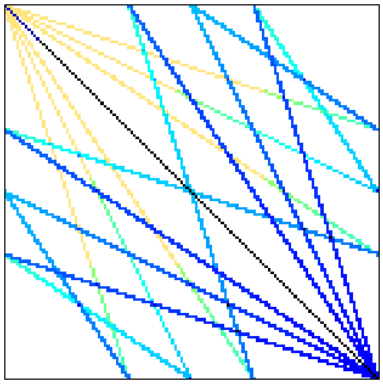
bloweybl



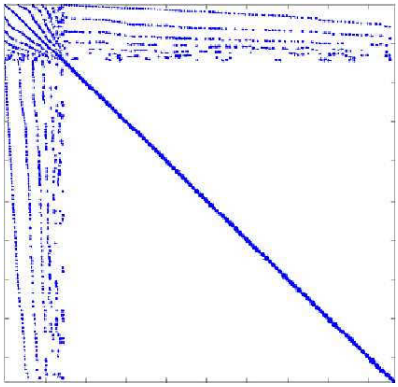
bcsstk35



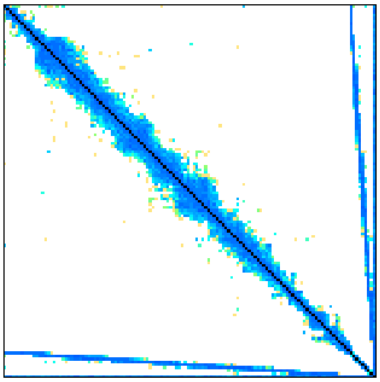
cvxbqp1



kurbel



thermal2



fidapm11

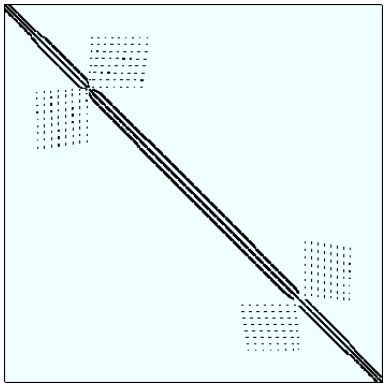


Abbildung 6.1: Strukturplots der Testmatrizen

Matrix	Dimension	Einträge	Besetzungsgrad
bloweybl	30.003	109.999	0,012%
bcsstk35	30.237	1.450.163	0,159%
cvxbqp1	50.000	349.968	0,014%
kurbel	192.858	24.259.520	0,065%
thermal2	1.228.045	8.580.313	0,0006%
fidapm11	22294	623554	0,125%

Tabelle 6.1: Dimensionen und Anzahl von Nicht-Null-Einträgen der Testmatrizen

Probleme bei der Zeitmessung und den Ergebnissen

Zu Beginn des Kapitels wurde bereits erwähnt, dass die gemessene Zeit für den Jacobi-Davidson-Algorithmus als Vergleichskriterium zwischen den beiden Lösern, TFQMR und MUMPS, dienen soll. Es wird also die sogenannte Wall-Clock-Zeit gemessen, sprich die reale Zeit, in der die Prozessoren zur Lösung des Problems belegt wurden. Diese Zeitmessung schließt dabei natürlich Zeiten eventueller Blockierung einzelner Prozessoren während der Ausführungszeit mit ein. Diese Blockierungen kommen zustande, da nicht vorhersehbar ist, welcher Prozessor wie lange für die ihm zugeteilte Aufgabe benötigt. Diese Ausführungszeit kann zum Teil sehr stark schwanken, sodass es zu Blockierungen anderer Prozessoren kommt, die an einer „Schranke“ im Programmcode auf den langsameren Prozessor warten müssen. Es existiert zwar die Möglichkeit nur die sogenannte CPU-Zeit zu messen, also nur die Zeit, in der der Prozessor effektiv arbeitet, jedoch erhält man dadurch keine reale Erkenntnis über die tatsächlich benötigte Zeit, um ein bestimmtes Problem lösen zu können.
Da also die Wall-Clock-Zeit als Vergleichskriterium dienen soll, muss darauf geachtet werden, dass so wenig störende Faktoren wie möglich mit in die gemessene Zeit einfließen, da die Ergebnisse ansonsten massiv verfälscht werden könnten.
Bei kleinen Matrizen, bei denen der eigentliche Rechenaufwand sehr gering ausfällt, kann zum Beispiel das Betriebssystem die Zeitmessung verfälschen. Die oben vorgestellten Testmatrizen sind daher ausreichend groß gewählt, um diesen Störfaktor so gering wie möglich ausfallen zu lassen. Zusätzlich wird mit der Wahl großer Matrizen auch verhindert, dass die Zeit, die durch Kommunikation zwischen den einzelnen Prozessoren anfällt, zu stark ins Gewicht fällt.
Werden neben dem Jacobi-Davidson-Programm auf einem Knoten des Supercomputers gleichzeitig weitere kommunikationsintensive Fremdprogramme ausgeführt, kann dies dazu führen, dass das eigene Programm bei gleich eingestellten Parametern eine zum Teil wesentlich längere Rechenzeit aufweist. Dieses Phänomen lässt sich zwar nicht verhindern, aber die daraus entstehende Verfälschung der Messzeit kann insofern eingedämmt werden, als dass die Testläufe lediglich auf den sogenannten Batch-Knoten des Supercomputers durchgeführt werden. Die Batch-Knoten zeichnen sich dadurch aus, dass auf ihnen wesentlich weniger Benutzer gleichzeitig rechnen, da die dort ausgeführten Programme zum Teil sehr lange Ausführungszeiten aufweisen und viele Prozessoren benötigen.
Zusätzlich wurde das Jacobi-Davidson-Programm für die folgenden Testläufe mit gleich eingestellten Parametern je dreimal unabhängig hintereinander ausgeführt, um anschließend eine Aussage darüber treffen zu können, wie sehr die Ausführungszeiten untereinander schwanken, beziehungsweise, um sehen zu können, ob bei gleich eingestellten Parametern auch die Ergebnisse gleich sind. Allgemein bekannt ist, dass parallele Programme nicht deterministisch sind, sprich, dass sowohl die Ergebnisse als auch die Zustandsänderungen während des Programmablaufs nicht eindeutig sind. So kann es vorkommen, dass ein und derselbe Testlauf zu zwei verschiedenen Ergebnissen führt. Solche Effekte treten zum Beispiel bei Summenbildungen unter Verwendung der MPI-Routine *MPI_Allreduce* auf. Da im MPI Standard nicht festgelegt ist, in welcher Reihenfolge die jeweiligen Operationen durchgeführt werden sollen, können die Ergebnisse von Lauf zu Lauf verschieden ausfallen, bedingt durch Rundungsfehler.
Da das Jacobi-Davidoson-Programm nun auf einem Parallelrechner ausgeführt wird, kann es durchaus während den Ausführungen zu solchen nicht vorhersehbaren Zwischenfällen kommen.
Sind während den Tests alle drei Durchläufe für jeweils gleich eingestellte Parameter terminiert, so wurden zur grafischen Darstellung und zum Vergleich der Löser die jeweils kleinsten Zeiten benutzt, weil die erhöhten Ausführungszeiten auf oben beschriebene Fremdeinflüsse schließen lassen.

Die Vorbereitungen für die Testläufe

Im Folgenden wird erläutert mit welchen Parametern das Jacobi-Davidson-Programm eingestellt wurde, um von den oben bereits vorgestellten Matrizen die gewünschten Eigenwerte zu berechnen.
Ziel war es, für die Matrizen kleinerer Dimensionen, sprich, für *bcsstk35*, *cvxbqp1*, *bloweybl* und

fidapm11, Testläufe mit jeweils einem Prozessor und vier Prozessoren durchzuführen. Dabei sollten je ein größter ('whicheigv' = 1, 'numevs' = 1) und ein kleinster Eigenwert ('whicheigv' = 0, 'numevs' = 1), sowie zehn und 50 größte und kleinste Eigenwerte bestimmt werden. Zudem wurde versucht je zehn innere Eigenwerte ('whicheigv' = 2, 'numevs' = 10) in der Nähe von 10%, 50% und 75% des Spektrums zu finden. Dazu muss der Parameter 'tau' mit dem Wert besetzt werden, in dessen Nähe die Eigenwerte gesucht werden sollen (zum Beispiel: 'tau' = 1.67e9) .
Für die beiden größeren Matrizen *kurbel* und *thermal2* wurde die Untersuchung ein wenig der Dimension der Matrizen angepasst. Die Testläufe wurden sofern möglich auf einem Prozessor, sowie auf vier, acht, 16 und 32 Prozessoren durchgeführt. Auch hier wurden je ein größter und ein kleinster Eigenwert, sowie zehn und 50 größte und kleinste Eigenwerte gesucht, außerdem noch zehn Eigenwerte aus der Mitte des Spektrums.
Alle Testläufe wurden jeweils mit beiden Korrekturgleichungs-Lösern MUMPS ('precon' = 5) und TFQMR ('precon' = 1) durchgeführt.

Da die jeweiligen Unterraum-Erweiterungen an die Anzahl der gesuchten Eigenwerte angepasst werden muss, wurden diese zuvor entsprechend festgelegt. Für die Berechnung eines Eigenwertes wurde eine Unterraum-Erweiterung von sechs ('itm' = 6) gewählt, bei zehn Eigenwerten eine von 50 und bei 50 Eigenwerten eine Unterraum-Erweiterung von 150.

Die Testläufe, die im Folgenden dokumentiert werden, sind stets auf eine von drei Arten terminiert. Im positiven Fall wurde innerhalb einer zuvor erwarteten Zeit das erforderte Residuum von 10^{-4} ('reseps' = 1.d-4) erreicht. Es kam jedoch auch vor, dass entweder keine Konvergenz des Verfahrens stattgefunden hat oder aber die maximale Anzahl an Jacobi-Davidson-Iterarionen, zuvor auf 1 000 000 festgelegt ('jditmax' = 1000000), erreicht wurde. In beiden Fällen wurde versucht die Ursache dafür zu finden, dass die gesuchten Eigenwerte nicht entsprechend berechnet werden konnten.

Bei der Berechnung innerer Eigenwerte bei 10%, 50% und 75% des Spektrums wäre es prinzipiell nötig gewesen alle Eigenwerte zu berechnen, um zu wissen, wie diese verteilt sind. Da dies zum einen nicht im Sinne der Aufgabenstellung ist und zum anderen mit einem sehr großen zusätzlichen Aufwand verbunden wäre, versucht man die Verteilung der Eigenwerte anzunähern. Dafür greift man darauf zurück, dass in der Regel die Matrizen der vorliegenden Problemstellungen in etwa einer 3D-Diskretisierungen des Laplace-Operators entsprechen. Bei solchen Problemen liegen die Eigenwerte in etwa so verteilt vor, wie die Werte einer Summe aus drei Quadratzahlen, die der Größe nach sortiert werden. Jede dieser drei Zahlen, nimmt dabei Werte zwischen eins und der dritten Wurzel aus der Matrixdimension an. Wird diese Verteilung linear auf den Bereich zwischen dem zuvor berechneten größten und kleinsten Eigenwert einer Matrix skaliert, erhält man eine sehr grobe, aber ziemlich zuverlässige Aussage darüber, wie die Eigenwerte im Inneren des Spektrums verteilt sein müssten.
Da diese Berechnung bei sehr großen Matrizen dennoch sehr aufwändig ist, kann man näherungsweise für einen Eigenwert aus der Mitte des Spektrums auf einen Wert zurückgreifen, der bei zehn Prozent des größten Eigenwertes liegt. In der Regel erhält man so einen Wert, der sich ziemlich genau in der Nähe von 50% des Spektrums befindet.
Die Tabelle 6.2 bildet die Näherungen ab, die die oben beschriebenen Berechnungen ergaben. Diese Werte wurden im Anschluss, zur Bestimmung der inneren Eigenwerte, mit Hilfe des Parameters *tau* in der Datei *jada15.par*, dem Jacobi-Davidson-Programm zur Verfugüng gestellt.
Für die *bloweybl*-Matrix wurden keine derartigen Näherungen an die Werte innerhalb des Spektrums berechnet, da bei der Bestimmung kleinster und größter Eigenwerte ganz spezielle Probleme auftraten, die später noch ausführlich dokumentiert werden.

Die Startvektoren für den Jacobi-Davidson-Algorithmus werden standardmäßig durch das Lanczos-

Matrix	Wert bei 10% des Spektrums	Wert bei 50% des Spektrums	Wert bei 75% des Spektrums
bcsstk35	5.6881e8	1.6738e9	2.3165e9
cvxbqp1	5.2440e4	1.6152e5	2.2392e5
fidapm11	-0.2683e0	0.7727e0	1.0670e0
kurbel	-	7.8337e6	-
thermal2	-	0.7805e0	-

Tabelle 6.2: Näherungswerte der Suche für 10%, 50% und 75% des Spektrums

Verfahren, für symmetrische Matrizen, und durch das Arnoldi-Verfahren, für unsymmetrische Matrizen, generiert. Dadurch wird gewährleistet, dass die ersten Eigenwert-Approximationen schon ziemlich in der Nähe derer liegen, die als Ergebnis herauskommen sollen. Diese beiden zur Startvektorerzeugung entwickelten Routinen wurden erst vor kurzem zu der in [18] beschriebenen Version des Jacobi-Davidson-Programms hinzugefügt. Der Grund für die Hinzunahme weiterer Routinen war, dass der Jacobi-Davidson-Algorithmus gerade dann gute Konvergenzeigenschaften aufweist, wenn sich die Eigenwertnäherungen schon in der Nähe der exakten Lösung befinden. Da die beiden Routinen genau dies in angemessener Zeit leisten, erhält man verbesserte Ausführungszeiten gegenüber Laufzeiten des Programms unter Verwendung von Startvektoren, die aus Pseudo-Zufallsvektoren oder Einheitsvektoren bestehen.
Ein weiterer Vorteil, der sich durch den Einsatz des Arnoldi-Verfahrens ergibt, ist der, dass nach Beenden des Verfahrens die berechneten Annäherungen für die Eigenwerte ausgegeben werden und anhand derer eine Aussage darüber getroffen werden kann, ob die Eigenwerte der unsymmetrischen Matrizen ein reelles oder aber ein komplexes Spektrum bilden. Dies ist eine wichtige Information, die zur Auswahl der Testmatrizen herangezogen wurde. Das Jacobi-Davidson-Programm verfügt zwar über die Möglichkeit nach nahezu beliebig vielen extremen oder inneren Eigenwerten zu suchen, jedoch wird ein reelles Spektrum dazu vorausgesetzt.

Aus Gründen der Zeitauslastung wurden für einige extrem rechenintensive Testläufe, speziell für die Bestimmung von kleinsten und inneren Eigenwerten, lediglich Approximationen der jeweils benötigten Laufzeit ermittelt. Wie diese Approximationen im Einzelnen durchgeführt wurden, ist in dem entsprechenden Teil dieses Kapitels näher erläutert.

Die Übersetztung des Programms

Da das Programm ausschließlich auf dem parallelen Rechner *JUMP* getestet wurde, musste der Quellcode, wie in Kapitel 5 bereits erwähnt, mit dem Makefile *Makefile.AIX.PAR* kompiliert werden.

Im Folgenden werden nun kurz einige der Compiler-Optionen vorgestellt, die für die Testläufe verwendet wurden, um Ergebnisse aus zukünftigen Tests besser mit den hiesigen vergleichen zu können.

Das Übersetzten des Programmcodes für die Testläufe hat stets mit dem Compiler *mpxlf* und der zusätzlichen Compiler-Option *-O2*, sprich Optimierungsstufe zwei, stattgefunden. Testläufe, bei denen der Code ohne Optimierung, beziehungsweise nur mit Optimierungsstufe eins kompiliert wurde, ergaben weitaus langsamere Ausführungszeiten des Programms. Testläufe aus der Vergangenheit zur Perfomance-Messung des Jacobi-Davidson-Programms, nachzulesen in [19], zeigten zudem, dass Übersetzungen des Codes mit der nächsthöheren Optimierungsstufe, *-O3*, zu stark schwankenden Iterationszahlen führen können. Deshalb wurde auf eine höhere Optimierungsstufe als *-O2* verzichtet.

Für die 64bit Anwendungen, die der Supercomputer unterstützt, musste die Option *-q64* angegeben werden. Zudem enthielt das Makefile den Zusatz *-bmaxdata:3500000000*. Mit *bmaxdata* wird eine Obergrenze für den maximal nutzbaren Speicherbereich während der Programmausführung festgelegt. Im Laufe der Ausführung selbst muss aber nur der wirklich benutzte Speicherplatz vorhanden sein, weshalb man mit der Angabe für die Speichergröße bei dieser Option relativ großzügig umgehen kann. Die Angabe erfolgt dabei in Bytes, sprich mit der obigen Option von *bmaxdata* wird ein ausführbares Programm erzeugt, welches einen maximalen Speicherplatz von 3,5 Gigabyte belegen kann. Diese Option wurde notwendig bei der Eigenwertberechnung sehr großer Matrizen. Bei Tests ohne eine entsprechende Option war nicht genügend Speicherplatz vorhanden, um alle, für die Programmausführung benötigten Datenobjekte anzulegen.

Zusätzlich notwendige Optionen, die zum Einbinden von Programmbibliotheken, wie PESSLsmp und ScaLAPACK, mit angegeben wurden, sind: *-lesslsmp*, *-lpesslsmp*, *-lessl*, *-lblacspd*, *-lscalapack* und *-llapack*

MUMPS benötigt zudem beim Laden zwei weitere Bibliotheken libdmumps.a und libpord.a, die mit den Optionen *-ldmumps* und *-lpord* eingebunden werden. Diese Bibliotheken befinden sich in dem, bei der Installation von MUMPS erzeugten Verzeichnis *lib*.

Die Testläufe

Ziel der Tests ist es, dem Benutzer des Programms eine Aussage darüber liefern zu können, wann für welche Matrizen, bei welcher Anzahl gesuchter Eigenwerte und unter dem Einsatz wie vieler Prozessoren, es sich lohnt den CG-artigen Löser TFQMR oder aber den Multifrontal-Löser MUMPS zu wählen. Aus diesem Grunde wurden die Testmatrizen in Größe und Struktur auch so unterschiedlich wie möglich ausgewählt.

Zunächst werden nun die Ergebnisse präsentiert, die sich bei der Berechnung der größten Eigenwerte für die gegebenen Testmatrizen ergaben. Anschließend sind die Testläufe für kleinste und innere Eigenwerte dokumentiert, die jedoch nur zum Teil in der vorgegebenen Zeit terminiert sind. Die oben schon erwähnten Approximationen für diese Testläufe werden danach beschrieben.

In den beiden Abbildungen 6.2 und 6.3 sind die Ausführungszeiten der zu berechnenden größten Eigenwerte dargestellt. Bei der Matrix *bcsstk35* zeigt sich schon zu Beginn ein klares Defizit beim Einsatz des MUMPS-Lösers. Ist die Problemstellung sehr leicht und die gesamte Ausführungszeit dementsprechend kurz, wie es bei der Berechnung eines größten Eigenwertes der Fall ist, benötigt das Programm, verglichen mit der Rechenzeit bei Verwendung von TFQMR, deutlich länger. Zu erklären ist diese Tatsache damit, dass zu Beginn beim ersten Aufruf von MUMPS je eine sym-

bolische und eine numerische Faktorisierung der Matrix durchgeführt werden muss. Diese sind jedoch sehr rechenintensiv und zudem abhängig von der entsprechenden Matrixgröße. Da sowohl die Faktorisierungs-, als auch die Lösungs-Phase von MUMPS jeweils parallel durchgeführt werden, minimiert sich dieses Defizit der zeitaufwändigen Faktorisierung, wenn mehr Prozessoren gleichzeitig an dem Problem arbeiten. Deshalb fällt die zeitliche Differenz bei der Berechnung eines größten Eigenwertes unter Einsatz von vier Prozessoren auch wesentlich geringer aus.

Deutlich erkennbar ist, dass die Ausführungszeit zur Bestimmung der zehn größten Eigenwerte bei einem der Läufe nahezu identisch ist mit der Zeit für die Bestimmung eines Eigenwertes. In drei von vier Fällen lag die Laufzeit sogar weit unter der, die zur Berechnung für nur einen Eigenwert gebraucht wurde. Bei näherer Betrachtung der Ergebnisdaten wurde erkennbar, dass bei der Bestimmung der zehn größten Eigenwerte die Lanczos-Approximationen derart nah an der korrekten Lösung lagen, dass keine Korrekturgleichung gelöst werden brauchte und die approximierten Werte direkt als Lösung ausgegeben wurden. Bei der Bestimmung eines Eigenwertes war jedoch die korrekte Lösung nicht bei den Approximationen dabei und es brauchte für beide Löser weitere sechs Jacobi-Davidson-Iterationen und damit knapp 30 Lösungen der Korrekturgleichung, bis das errechnete Residuum genügend klein wurde. Die 30 Lösungen kommen dadurch zu Stande, da während jeder äußeren Iteration des Verfahrens der Unterraum entsprechend der zuvor gemachten Angaben erweitert wird. Da die Lösung der Korrekturgleichung nun eine sehr zeitaufwändige Berechnung verursacht, ist damit erklärt, weshalb die Bestimmung eines größten Eigenwertes zum Teil länger dauert als die von zehn größten Eigenwerten.
Warum dieses Verhalten gerade bei der *bcsstk35*-Matrix auftrat, konnte jedoch aus zeitlichen Gründen nicht weiter untersucht werden.
Ein besonderes Verhalten von MUMPS ist anhand der geringeren Laufzeiten für die 50 größten Eigenwerte beim Einsatz von einem Prozessor zu erkennen. Die Stärke des direkten Lösers scheint sich also bemerkbar zu machen, wenn die Problemgröße steigt, aber nur wenige Prozessoren zum Einsatz kommen. Dieses Verhalten wurde im weiteren Verlauf der Tests noch des Öfteren beobachtet.

Die Abbildung der Laufzeiten für die Berechnung größter Eigenwerte der Matrix *cvxbqp1* spiegelt wieder, was man erwartete hätte, nämlich, dass für mehr zu berechnende Eigenwerte der Aufwand und damit auch die Ausführungszeit des Programms ansteigt. Lediglich die Berechnung von 50 Eigenwerten mit einem Prozessor und MUMPS wurde nicht terminiert. Die Näherungen der Eigenwerte verrieten, dass bei jedem der drei unabhängigen Läufe der erste Eigenwert stetig steigt und gegen keinen Wert konvergiert. Nach einer Wall-Clock-Zeit von 30 Minuten wurden die Läufe abgebrochen. Der Grund für ein solches Verhalten ist vermutlich, der oben schon erwähnte Sachverhalt, dass parallele Programme zum Teil nicht deterministisch und deren Ergebnisse somit vorab auch nicht absehbar sind. Dass dies die Ursache nach aller Wahrscheinlichkeit nach auch ist, unterstreicht die Tatsache, dass lediglich zwei, der insgesamt drei Testläufe für 50 größte Eigenwerte mit TFQMR beim Einsatz von einem Prozessor terminierten. Der dritte Test wies genau das selbe Verhalten auf, welches oben für den Lauf mit MUMPS beschrieben wurde.

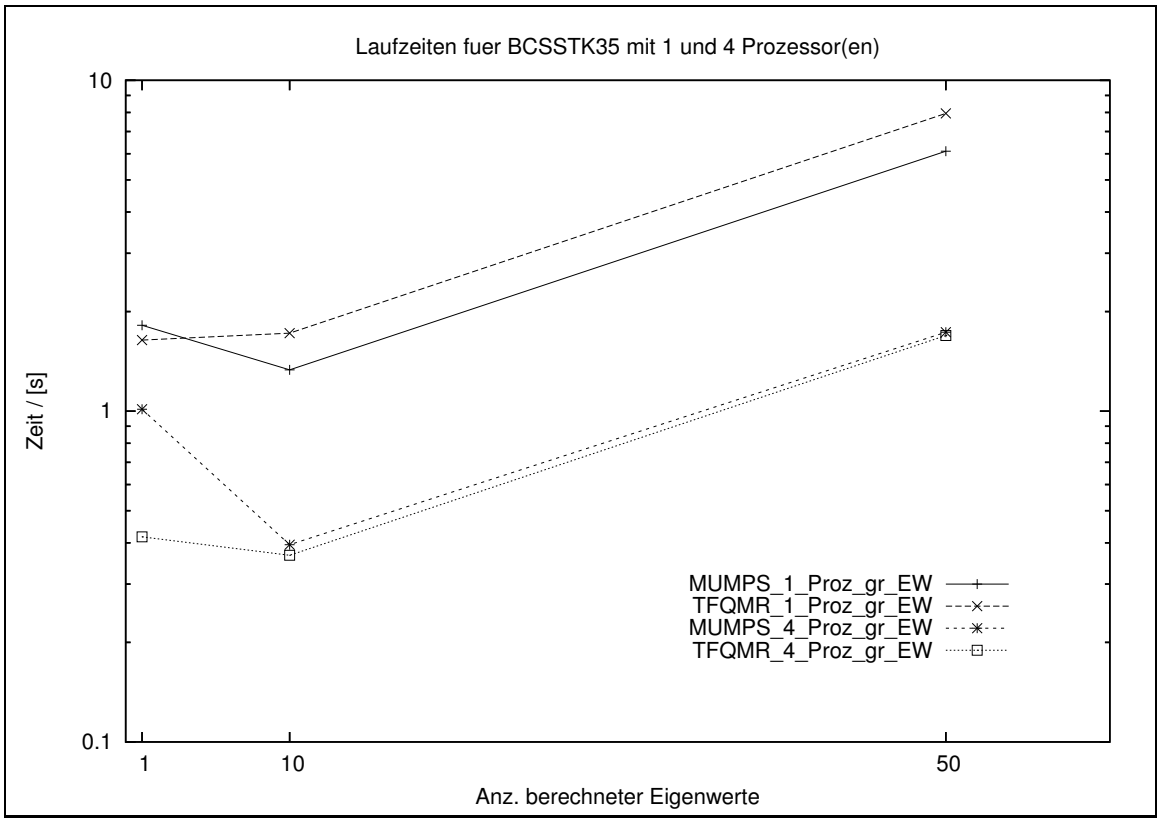


Abbildung 6.2: Zeiten zur Berechnung der Eigenwerte von Matrix *bcsstk35*

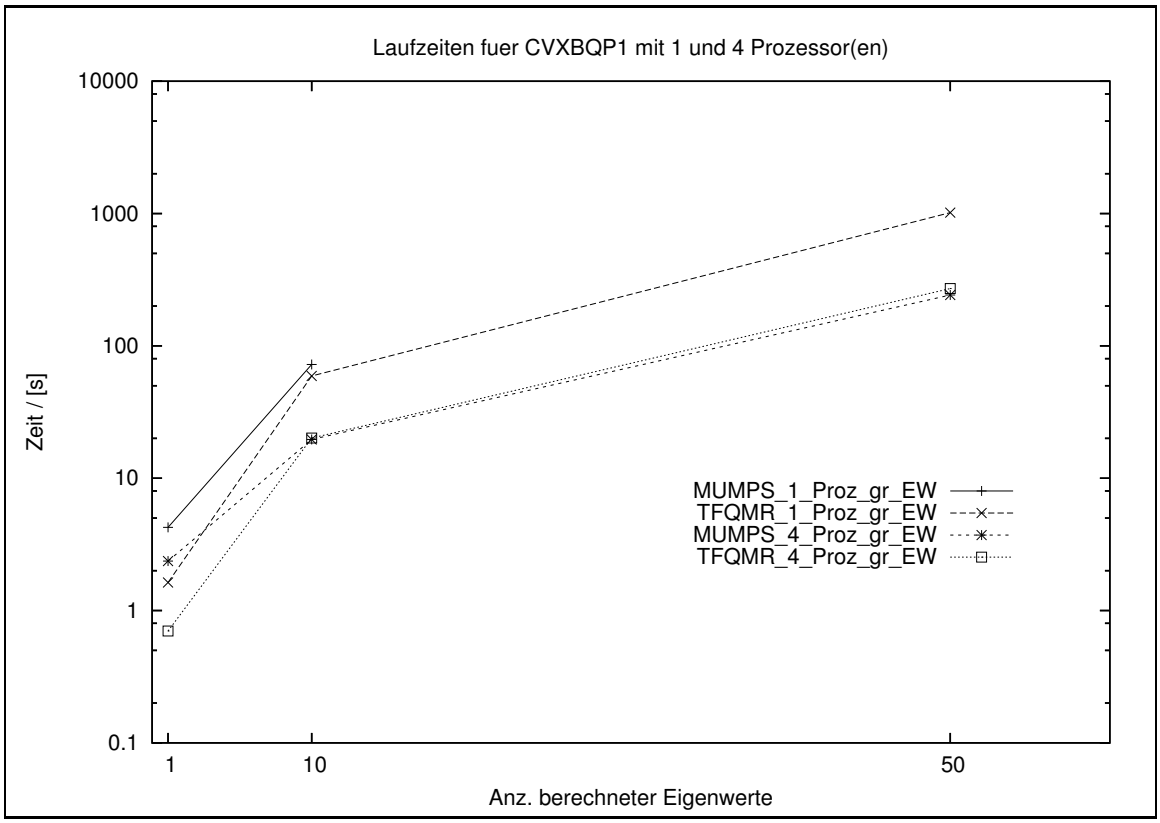


Abbildung 6.3: Zeiten zur Berechnung der Eigenwerte von Matrix *cvxbqp1*

Die Ausführungszeiten des Jacobi-Davidson-Programms zur Berechnung größter Eigenwerte der *kurbel*-Matrix sind in Abbildung 6.4 dargestellt. Man erkennt deutlich, dass mit steigender Matrixdimension die Zeit für die Faktorisierung von MUMPS ins Gewicht fällt. Je größer zudem die Anzahl der eingesetzten Prozessoren ist, desto extremer wird die Zeitdifferenz zwischen den Ergebnissen mit den beiden Lösern. Bei 32 Prozessoren liegt bereits ein Faktor 10 zwischen den Zeiten zur Bestimmung von einem größten Eigenwert.
Betrachtet man die Ausführungszeiten mit MUMPS weiter für zehn größte Eigenwerte, stellt man fest, dass hier kein großer zeitlicher Zuwachs stattfindet. Dieses Verhalten ist auf zwei Sachverhalte zurückzuführen:
Zum einen ist bei der *kurbel*-Matrix ein ähnliches Verhalten, wie bei der *bxsstk35*-Matrix festzustellen, nämlich, dass die Lanczos-Methode bei zehn größten Eigenwerten wieder ziemlich gute Näherungen der exakten Lösung berechnet. Dadurch werden zur Berechnung der zehn größten Eigenwerte lediglich sechs Iterationen mehr vom Jacobi-Davidson-Programm durchgeführt, als die 51 Gesamtiterationen, die zur Berechnung eines größten Eigenwertes nötig waren.
Zum anderen spielt aber auch hier wieder der Sachverhalt eine Rolle, dass, wenn einmal die bei solchen Matrixdimensionen besonders aufwändige Faktorisierung von MUMPS durchgeführt wurde, die weiteren Lösungsschritte sehr schnell berechnet werden können. Genaue Zeitangaben zu den Ausführungszeiten der drei MUMPS-Phasen sind im letzten Teil dieses Kapitels zu finden.

Schließlich ist auch bei der *kurbel*-Matrix wieder festzustellen, dass sich der Einsatz von MUMPS gerade dann lohnt, wenn die Problemstellung ausreichend groß ist. So werden bei der Berechnung von 50 größten Eigenwerten der Matrix mit MUMPS sehr viel bessere Ausführungszeiten erreicht, als es mit TFQMR der Fall ist. Auch hier ist wieder eindeutig die bessere Performance des MUMPS-Lösers beim Einsatz einer kleineren Anzahl an Prozessoren erkennbar.

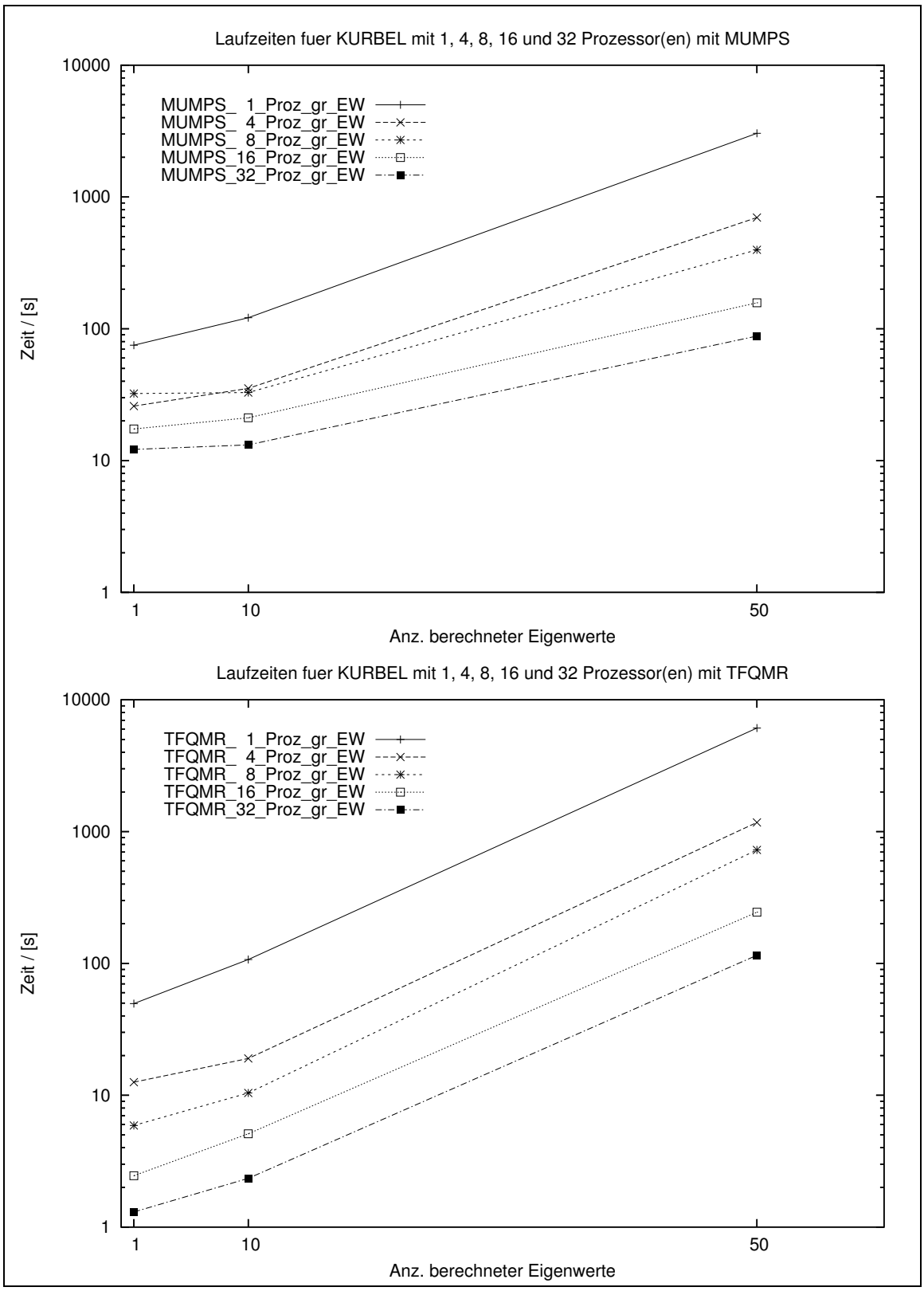


Abbildung 6.4: Laufzeiten zur Berechnung der größten Eigenwerte von Matrix *kurbel*

Zusätzlich zu den reinen Ausführungszeiten mit den beiden Lösern wurde für die *kurbel*-Matrix der Speedup berechnet.

Der Speedup eines Programms wird über

$$S(N) = \frac{T(1)}{T(N)}$$

berechnet. N ist dabei die Anzahl der eingesetzten Prozessoren und $T(1)$ die benötigte Rechenzeit zur Problemlösung auf einem Prozessor. Aufgetragen in einem Diagramm gegen die Anzahl der Prozessoren, kann somit die Effizienz, auch Skalierung genannt, eines Programms festgestellt werden. Das heißt, man kann eine Aussage darüber machen, inwieweit ein Programm parallelisierbar ist und wie viel Zeit beim Einsatz von mehr Prozessoren eingespart werden kann.

Das Amdahlsche Gesetz besagt Folgendes über den Speedup:
Seien f der Anteil des Programmcodes, der ausschließlich seriell ausgeführt werden kann, N die Anzahl der Prozessoren und S der erreichte Speedup, dann gilt nach diesem Gesetz:

$$S(N) \leq \frac{1}{f + \frac{1-f}{N}}$$

Im Idealfall strebt nun mit immer größer werdenden Problemstellungen f gegen 0 und S(N) ist gleich N. Das hieße, bei doppelter Anzahl eingesetzter Prozessoren, wird nur die Hälfte der zuvor benötigten Zeit gebraucht. Dieser Fall wird auch idealer oder linearer Speedup genannt.

Abbildung 6.5 zeigt nun den Speedup, der durch das Jacobi-Davidson-Programm unter Einsatz der beiden Löser erreicht wurde.

Wie man erkennt, liegen alle Speedup-Kurven, die mit TFQMR erreicht wurden weit über dem linearen Speedup. Dies ist speziell bei parallen Programmen und sehr großen Problemstellungen keine Seltenheit. Begründbar ist dieses Verhalten durch *Cache-Effekte*. Jedesmal, wenn Daten der Matrix für die Berechnungen innerhalb eines Programms benötigt werden, schickt der arbeitende Prozessor eine Anfrage zu seinem Cache-Speicher, ob die Daten dort vorrätig sind. Liegen die Daten nicht im Cache tritt ein sogenannter „Cache-Miss“ ein und die benötigten Daten müssen aus dem Arbeitsspeicher in den Cache geladen werden. Je größer nun die Matrizen sind und je weniger Prozessoren eingesetzt werden, desto häufiger kommen solche Cache-Misses vor. Dieses häufige Anfragen der zum Rechnen benötigten Daten erfordert extrem viel Zeit. Werden hingegen mehrere Prozessoren eingesetzt um eine Aufgabe zu lösen, so ist die Wahrscheinlichkeit um einiges höher, dass die gerade benötigten Daten bereits im Cache vorrätig liegen. Dieser Sachverhalt macht einen *superlinearen Speedup* möglich, so wie er hier bei der *kurbel*-Matrix dargestellt ist.

Zudem sieht man deutlich, dass MUMPS erst superlinearen Speedup erreicht, wenn die Problemgröße ausreichend groß ist, sprich bei der Berechnung von 50 Eigenwerten. Insgesamt zeigen die Testläufe für größte Eigenwerte der *kurbel*-Matrix, dass der Speedup des Jacobi-Davidson-Programms mit TFQMR zwar wesentlich höher ausfällt als der mit MUMPS, jedoch sprechen die Laufzeiten aus Abbildung 6.4 für den Einsatz des direkten Lösers, wenn mit weniger als 16 Prozessoren 50 größte Eigenwerte der Matrix bestimmt werden sollen.

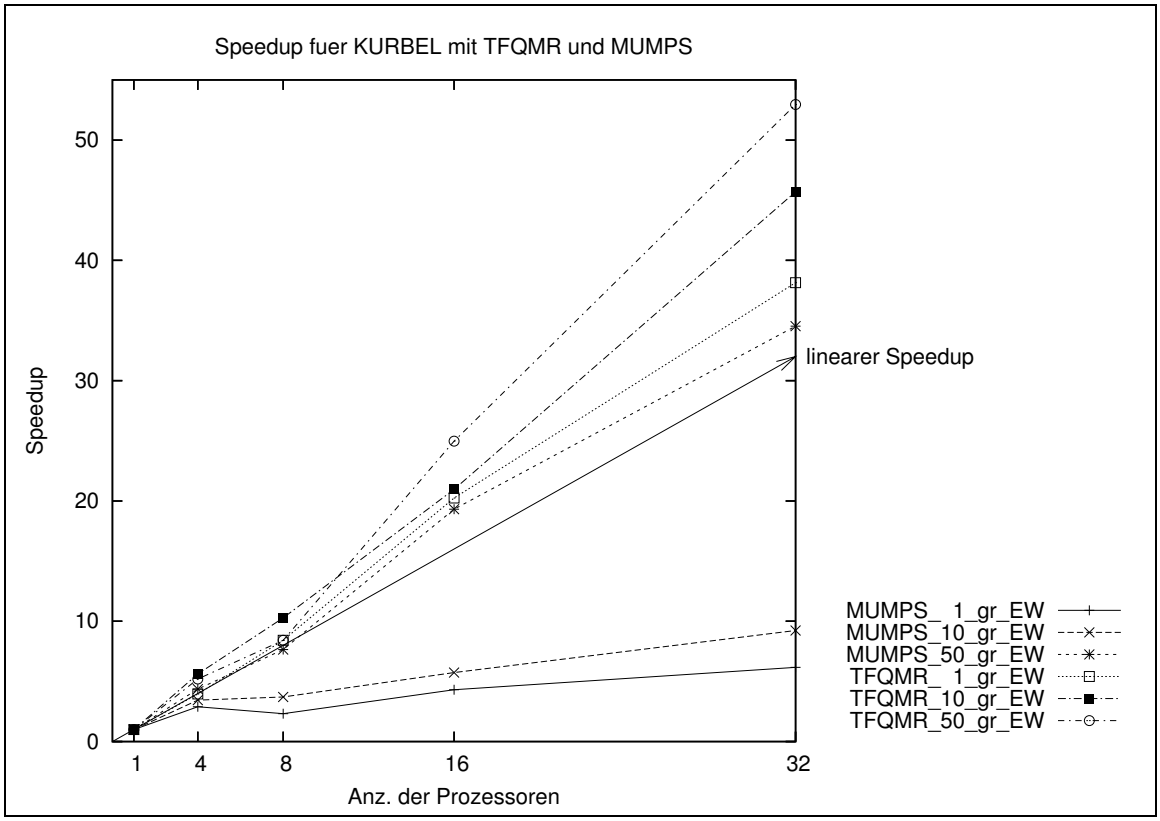


Abbildung 6.5: Speedup der Eigenwertberechnung von Matrix *kurbel*

Die Testläufe mit der *thermal2*-Matrix zur Berechnung der größten Eigenwerte spiegeln in etwa die zuvor dokumentierten Ergebnisse der Eigenwertberechnung für die *kurbel*-Matrix wieder.
Wie anhand der Abbildung 6.6 zudem erkennbar ist sind einige Testläufe nicht terminiert. Dazu gehört zum einen die Bestimmung 50 größter Eigenwerte auf einem Prozessor unter Einsatz von MUMPS. Mit einer Dimension von 1,2 Millionen und einer Anzahl Nicht-Null-Einträge von mehr als 8,5 Millionen übeschreitet die *thermal2*-Matrix die auf einem Prozessor verfügbaren Ressourcen. Bei der Bestimmung der 50 größten Eigenwerte mit TFQMR trat hingegen ein ähnliches Problem auf wie bei der *cvxbqp1*-Matrix. Der jeweils erste Eigenwert wurde mit jeder Jacobi-Davidson-Iteration immer größer, während die restlichen Näherungen der gesuchten Eigenwerte ein sprunghaftes Verhalten aufwiesen, sprich nach jeder Iteration an einer anderen Eigenwertnäherung arbeiteten. Auf diese Weise wurde das Programm selbst nach zwei Stunden auf 32 Prozessoren nicht terminiert und deshalb abgebrochen. Ein weiterer Testlauf hingegen mit 64 Prozessoren erzielte nach nur 15 Minuten eine genügend genaue Näherung der Eigenwerte. Da sich dieses Verhalten nun aber bei einigen, unabhängig voneinander laufenden, Tests gezeigt hat, auch bei der Berechnung von Eigenwerten unterschiedlicher Matrizen, stellt sich die Frage, ob die ursache dafür wirklich nur damit zu erklären ist, dass parallele Programme nicht deterministisch sind. Leider blieb nicht genügend Zeit, die genaue Ursache für dieses Verhaltens näher zu untersuchen.

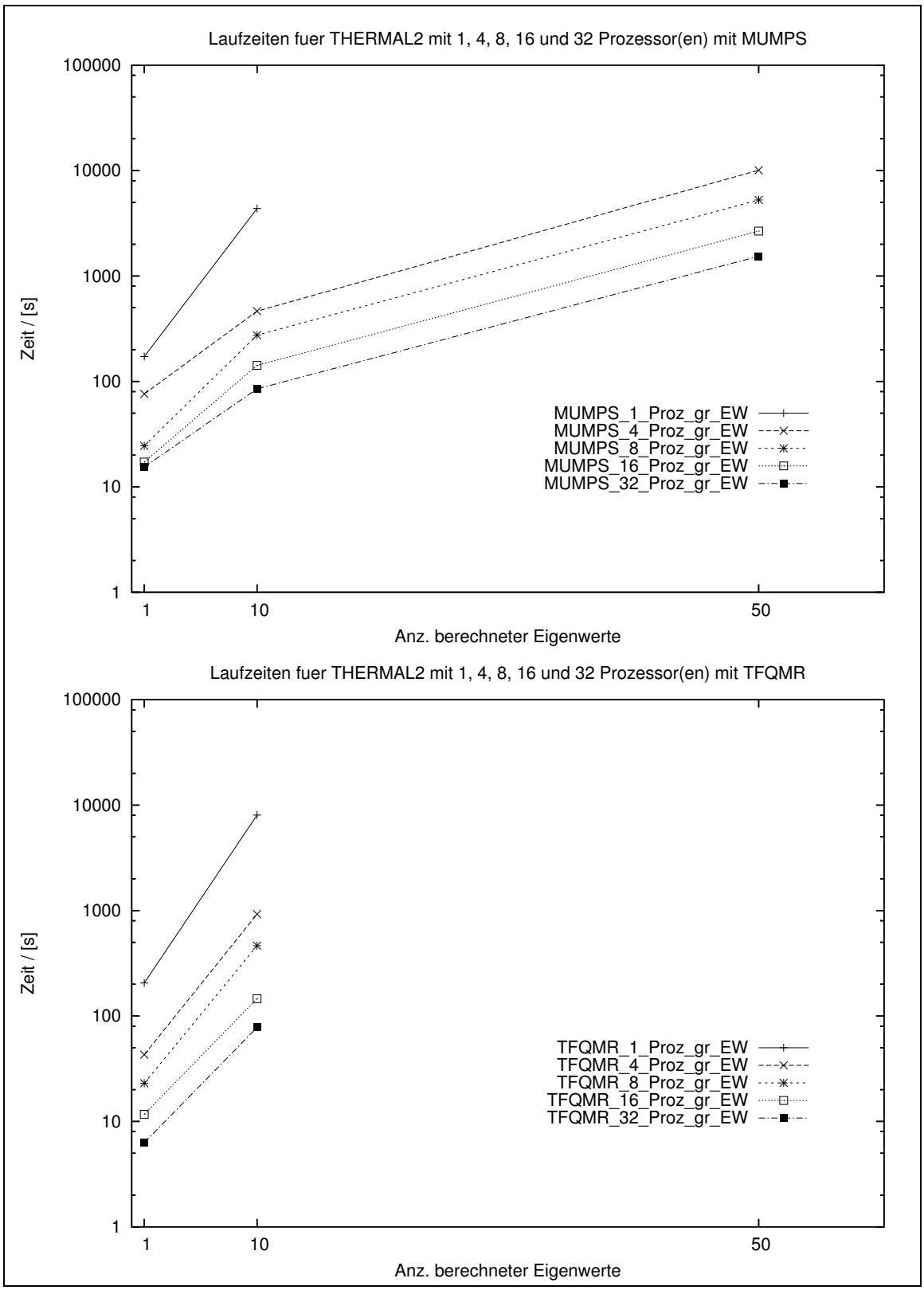


Abbildung 6.6: Laufzeiten für größte Eigenwerte der Matrix *thermal2*

Die folgende Grafik 6.7 stellt den Speedup dar, der bei der Eigenwertberechnung der *thermal2*-Matrix erreicht wurde. Aufgrund der oben schon beschriebenen Probleme bei der Berechnung der 50 größten Eigenwerte fehlt hier jedoch die Kurve für den TFQMR-Löser. Die entsprechende Kurve für MUMPS beginnt erst bei vier Prozessoren. Der Speedup wurde hier nicht bezüglich der Zeit mit einem Prozessor berechnet sondern bezüglich vier Prozessoren. Dabei sind natürlich eine deutlich geringere Anzahl an Cache-Misses als bei der Berechnung auf einem Prozessor aufgetreten, weswegen die Kurve auch weit weniger steil verläuft, verglichen mit der Kurve für 50 Eigenwerte bei der *kurbel*-Matrix.

Die Abbildung zeigt zudem, dass auch bei den Tests mit der *thermal2*-Matrix mit steigender Problemgröße der Speedup für MUMPS weiter ansteigt, jedoch stets unterhalb des Speedups von TFQMR bleibt. Die Ausführungszeiten von MUMPS mit kleiner Anzahl von Prozessoren liegen ab einer Berechnung von zehn und mehr Eigenwerten weit unter denen, die das Programm mit TFQMR als Löser benötigt. Je mehr Prozessoren zum Einsatz kommen, desto geringer wirkt sich der Zeitgewinn aus, was im Grunde an einer schlechten Skalierung der Lösungs-Phase von MUMPS liegen muss, denn diese wird nach einmaliger Ausführung der Faktorisierung nur noch aufgerufen. Auf diesen Sachverhalt wird am Ende dieses Kapitels noch näher eingegangen.

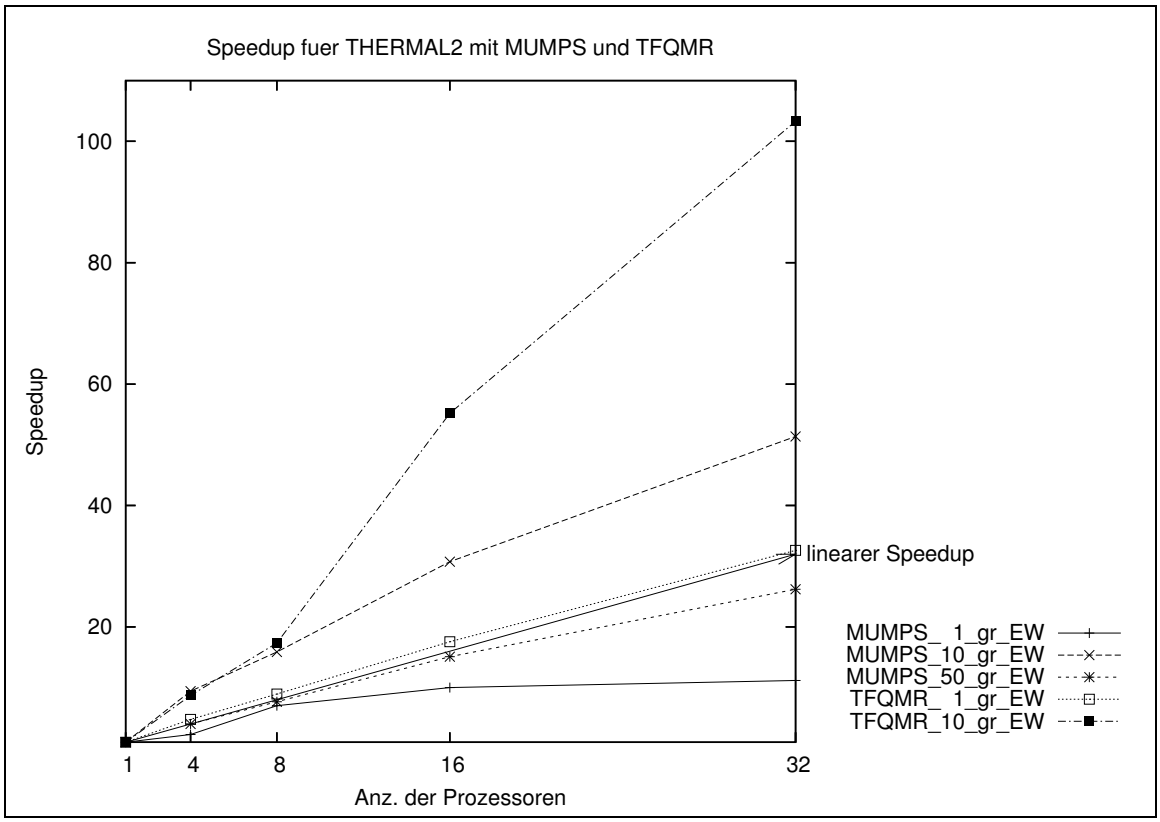


Abbildung 6.7: Speedup der Eigenwertberechnung für Matrix *thermal2*

Die Beschreibung der Testläufe umfasst bis hierhin zunächst nur die Läufe zur Berechnung größter Eigenwerte von vier der sechs vorgestellten Matrizen, da das Programm dabei meist in der vorgegebenen Zeit terminiert wurde. Die bisher gesammelten Informationen lassen an dieser Stelle bereits ein Muster erkennen, wann sich der Einsatz von MUMPS zur Lösung der Korrekturgleichung zu lohnen scheint und wann eher TFQMR benutzt werden sollte.

Als nächstes werden die Ergebnisse der Testläufe mit der unsymmetrischen *fidapm11*-Matrix vorgestellt.

Bei der Eigenwertberechnung unsymmetrischer Matrizen mit Hilfe des Jacobi-Davidson Programms treten häufig Probleme auf, da das Jacobi-Davidson-Programm sich nur dann für unsymmetrische Matrizen eignet, wenn deren Spektrum auch reell ist. Sollte ein Eigenwert dennoch einen imaginäre Anteil besitzen wird dieser ignoriert. Das heißt aber, dass ein Fehler gemacht wird, der anschließend durch eine höhere Genauigkeit im Realteil ausgeglichen werden muss, was die Konvergenz des Verfahrens zum Teil sehr verlangsamt. Damit das Jacobi-Davidson-Programm in diesem Fall überhaupt konvergiert, dürfen die Imaginärteile der Eigenwerte jedoch nur sehr klein sein. Bei größeren Imaginärteilen, bei vielen unsymmetrischen Matrizen häufig um Faktor 10 oder 100 größer als deren Realteil, kann es vorkommen, dass gar keine Konergenz mehr eintritt.

Abbildung 6.8 zeigt die Ausführungszeiten des Jacobi-Davidson-Programms mit beiden Lösern zur Bestimmung der kleinsten Eigenwerte. Bei dieser Berechnung sind keinerlei derartige Probleme aufgetreten.

Hier ist deutlich ein anderes Verhalten als bei bisherigen Testläufen erkennbar. Mit MUMPS wird hier nun jeweils mit einem und mit vier Prozessoren der kleinste Eigenwert schneller berechnet als mit TFQMR. Für alle anderen Läufe lagen die Ausführungszeiten mit MUMPS stets über denen mit TFQMR. Eine mögliche Erklärung für dieses Verhalten ist, dass MUMPS im Gegensatz zu symmetrischen Matrizen, die unsymmetrischen nicht als eine untere Dreiecksmatrix verwalten und faktorisieren kann. Hierbei scheint der MUMPS-Löser zur Lösung der Korrekturgleichung ein klares Defizit in seiner Perfomance aufzuzeigen.

In Tabelle 6.3 sind diejenigen Ausführungszeiten der Tests zur Berechnung größter Eigenwerte aufgelistet, die nach einer vorgegebener Zeit von vier Minuten terminiert sind. Wieder einmal traten sowohl bei der Berechnung eines größten Eigenwertes mit TFQMR, als auch bei 50 größten Eigenwerten mit beiden Lösern Probleme auf, wie sie für die *cvybqp1*-Matrix bereits beschrieben wurden. Da jeweils der Wert des ersten Eigenwertes stetig anstieg, konnte bei diesen Läufen keine Konvergenz des Verfahrens eintreten und somit auch keine Zeit gemessen werden. Die Zeiten der terminierten Läufe zeigen aber auch hier wieder, dass der Einsatz von MUMPS längere Ausführungszeiten bewirkt.

Löser	**Anz. Proz.**	**Laufzeit 1 gr. EW**	**Laufzeit 10 gr. EWe**	**Laufzeit 50 gr. EW**
MUMPS	1	4.05	16.24	-
TFQMR	1	-	15.93	-
MUMPS	4	2.24	4.88	-
TFQMR	4	1.57	2.59	39.03

Tabelle 6.3: Laufzeiten / [s] für größte Eigenwerte der Matrix *fidapm11*

Nur schwer interpretierbar waren die Zeiten, die zur Berechnung innerer Eigenwerte der *fidapm11*-Matrix benötigt wurden. Zwischen dem größten Eigenwert, 1.6984, und dem kleinsten, -0.5498, die bereits als Ergebnisse der schon terminierten Testläufe vorlagen, sollten nun bei -0.2683, 0.7727 und bei 1.067 jeweils 10 Eigenwerte bestimmt werden. Die Tabelle 6.4 bildet die dafür benötigten Zeiten ab.

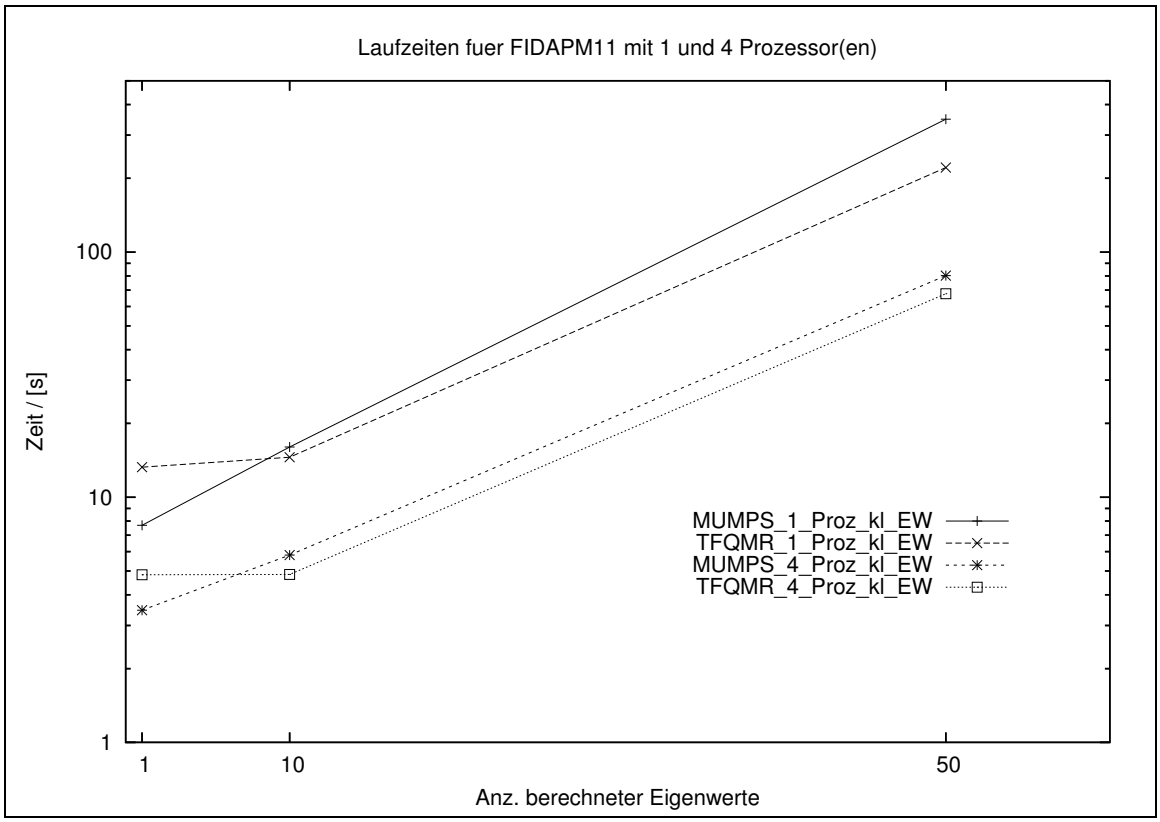


Abbildung 6.8: Zeiten zur Berechnung der kleinsten Eigenwerte von Matrix *fidapm11*

Die Ergebnisse zeigen durchweg, dass sich die Eigenwerte bei 75% des Spektrums ziemlich einfach berechnen lassen, andere bei 50% eher schwerer. Die Berechnung der Eigenwerte bei 10% des Spektrums wurde auf vier Prozessoren nach fünf Stunden mit beiden Lösern nicht terminiert. Ein Grund dafür mag sein, dass die Eigenwerte im Inneren sehr geclustert vorliegen, sprich, sich erst ab der dritten Nachkommastelle stark unterscheiden, sodass über viele Jacobi-Davidson-Iterationen hinweg nur eine sehr langsame Konvergenz der Eigenwerte erzielt wird. Zum anderen kann es auch vorkommen, dass während der Ausführung des Programms berechnete Zwischenmatrizen komplexe Einträge besitzen. In diesem Fall kann die Konvergenz des Verfahrens stark abnehmen oder sogar stagnieren.
Als Fazit der Laufzeitanalyse für diese unsymmetrische Matrix kann lediglich eine Aussage getroffen werden, nämlich, dass MUMPS allem Anschein nach große Schwierigkeiten bei der Lösung der Korrekturgleichung hat. Deutlich sichtbar ist eine große Zeitdifferenz speziell zwischen den Läufen für innere Eigenwerte mit MUMPS und TFQMR. Ob dieses Verhalten generell bei der Berechnung innerer Eigenwerte auftritt, oder vielleicht nur an der Unsymmetrie der Matrix liegt, werden weitere Testläufe zeigen müssen.

Löser	**Anz. Proz.**	**Laufzeit in. EW bei 10%**	**Laufzeit in. EW bei 50%**	**Laufzeit in. EW bei 75%**
MUMPS	1	-	-	43 Min. 56 Sek.
TFQMR	1	-	-	10 Min. 51 Sek.
MUMPS	4	-	3 Std. 6 Min.	11 Min.54 Sek.
TFQMR	4	-	1 Std. 2 Min.	2 Min. 52 Sek.

Tabelle 6.4: Laufzeiten für innere Eigenwerte der Matrix *fidapm11*

Sehr schwierig gestaltete sich die Berechnung von mehr als einem Eigenwert bei der *bloweybl*-Matrix, wobei es hier keinen Unterschied macht, ob kleinste oder größte Eigenwerte zu bestimmen waren. Die Tabelle 6.5 zeigt die Ausführungszeiten des Jacobi-Davidson-Programms zur Berechnung der jeweils extremsten Eigenwerte der Matrix, die sich ohne jegliche Probleme bestimmen ließen.

An beiden Rändern des Spektrums zeigten sich jedoch gleichermaßen bei der Berechnung von mehr als einem Eigenwert, dass je ein Eigenwert bei $+100$ und -100 existiert und alle weiteren bei $+2.8$ und -2.8 ganz dicht beieinander liegen. Innerhalb solcher Cluster überschneiden sich die Eigenwerte derart, dass sie nicht mehr unterscheidbar sind, sodass Projektionsverfahren, zu denen auch das Jacobi-Davidson-Verfahren gehört, keine Konvergenz mehr aufzeigen.

Tabelle 6.6 zeigt die letzten vier Näherungen bei der Suche nach den zehn kleinsten Eigenwerten. Die letzte Näherung steht dabei in der Spalte ganz rechts und wurde nach zehn Stunden beim Einsatz von vier Prozessoren ausgegeben. Das gleiche Verhalten wurde auf der anderen Seite des Spektrums beobachtet, als die zehn größten Eigenwerte der Matrix bestimmt werden sollten, hier änderte sich lediglich das Vorzeichen

Laut Beschreibung der Matrix Collection [28] führten Testläufe mit einer früheren Version eines Gleichungssystem-Lösers für dünnbesetzte Matrizen *MA57*, der in [9] beschrieben wird, mit dieser Matrix zu keinem Ergebnis. Deshalb war zu erwarten, dass an einer zu Beginn nicht vorhersehbaren Stelle auch bei der Eigenwertberechnung Probleme auftreten können.

Löser	**1 Proz. / 1 gr. EW**	**4 Proz. / 1 gr. EW**	**1 Proz. / 1 kl. EW**	**4 Proz. / 1 kl. EW**
MUMPS	0.0365	0.180	0.373	0.183
TFQMR	0.0365	0.184	0.759	0.182

Tabelle 6.5: Laufzeiten / [s] zur Berechnung eines Eigenwertes von Matrix *bloweybl*

1. EW	-99.986251114504	-99.986251116256	-99.986251115968	-99.986251116128
2. EW	-2.812889142039	-2.814493964458	-2.814753021925	-2.812085989682
3. EW	-2.816894537345	-2.812639691088	-2.815275676714	-2.816261643265
4. EW	-2.816348992657	-2.814178848863	-2.813525905756	-2.816878114067
5. EW	-2.818377167161	-2.818311348353	-2.818309707709	-2.818370896389
6. EW	-2.816519971734	-2.816512831593	-2.804767378034	-2.799025727387
7. EW	-2.816302308877	-2.815247748679	-2.807217045748	-2.817293778542
8. EW	-2.812882413598	-2.806817929184	-2.807021759006	-2.814466844875
9. EW	-2.783929464118	-2.795881052306	-2.813215250474	-2.799708139678
10. EW	-2.618139476721	-2.618300235793	-2.618290255459	-2.618093339267

Tabelle 6.6: Die letzten Iterationen bei der Suche nach dem kleinsten Eigenwert von *bloweybl*

Wie bereits zu Beginn der Testlauf-Dokumentation erwähnt wurde, scheint das Jacobi-Davidson-Programm allgemein Schwierigkeiten zu haben für alle zum Testen ausgesuchte Matrizen die kleinsten und ganz besonders die im Inneren des Spektrums liegenden Eigenwerte berechnen zu können. Man muss jedoch auch dazu sagen, dass die ausgewählten Matrizen alle schwere Problemstellungen darstellen und sich derart in ihrer Dimension und Struktur unterscheiden, dass im Voraus nicht absehbar ist, an welchen Stellen bei der Berechnung ihrer Eigenwerte genau Probleme auftreten können.

Die beiden nächsten Tabellen 6.7 und 6.8 zeigen nun noch Laufzeiten, die bei terminierten Testläufen zur Berechnung von kleinsten Eigenwerten der *thermal2*-Matrix, sowie von inneren Eigenwerte der *bcsstk35*-Matrix gemessen worden sind.

Anz. Proz.	**MUMPS 1 kleinster EW**	**TFQMR 1 kleinster EW**	**MUMPS Speedup**	**TFQMR Speedup**
1	660.11	989.39	1	1
4	186.65	288.92	3.56	3.42
8	90.57	141.61	7.29	6.99
16	50.6	64.04	13.05	15.45
32	32.67	41.68	20.21	23.74

Tabelle 6.7: Laufzeiten / [s] und Speedup für einen kleinsten Eigenwert der Matrix *thermal2*

Löser	**Anz. Proz.**	**Laufzeit / [s] in. EW bei 10%**	**Laufzeit / [s] in. EW bei 50%**	**Laufzeit / [s] in. EW bei 75%**
MUMPS	1	62.71	518.31	502.17
TFQMR	1	36.19	47.08	22.64
MUMPS	4	6.49	27.17	198.42
TFQMR	4	7.23	19.58	175.43

Tabelle 6.8: Laufzeiten für innere Eigenwerte der Matrix *bcsstk35*

Die Laufzeiten aus Tabelle 6.8 unterstreichen die Vermutung, auf die bei der *fidapm11*-Matrix bereits hingewiesen wurde, nämlich, dass sich der Einsatz des MUMPS-Lösers beim Berechnen innerer Eigenwerte gegenüber TFQMR nicht lohnt. Die gemessenen Ausführungszeiten des Programms mit MUMPS zeigen bis auf einen einzigen Test massive Zeiteinbußen, wenn MUMPS zum Lösen der Korrekturgleichung eingesetzt wird.

Im Gegensatz dazu scheint der Einsatz des MUMPS-Löser zur Berechnung kleinster Eigenwerten sinnvoll zu sein. Hier wurden, auch wenn der Speedup mit zunehmender Prozessoranzahl sinkt, stets bessere Laufzeiten erzielt als beim Einsatz des TFQMR-Lösers.

Laufzeit-Approximationen

Da sich die Berechnung kleinster und innerer Eigenwerte durch die Testläufe hinweg als sehr aufwändig erwiesen hat, wurde beschlossen nur noch mit einer großen Anzahl an Prozessoren diese Tests durchzuführen, um eine Aussage darüber zu erhalten, wie viele Jacobi-Davidson-Iterationen benötigt werden, um die Eigenwerte bis zu der geforderten Genauigkeit zu berechenen. Die Anzahl der Jacobi-Davidson-Iterationen dient dabei als eine zuverlässige Messgröße, da sie relativ unanbhängig von der Anzahl eingesetzter Prozessoren ist. Im Anschluss werden mit weniger Prozessoren dieselben Testläufe bei gleich eingestellten Parametern durchgeführt, wobei jedoch nach 20 Jacobi-Davidson-Iterationen abgebrochen wird. Von den Zeiten der Iterationen, die während des Programmablaufs ausgegeben werden, wird dann der Mittelwert gebildet. Dieser Wert multipliziert mit der Anzahl der Iterationen, die zum Erreichen der gewünschten Genauigkeit nötig sind, ergibt eine ziemlich gute Approximation für die Laufzeiten, die für weniger Prozessoren nötig gewesen wären.

Es wurden im Folgenden Testläufe mit einer Wall-Clock-Zeit von jeweils 24 Stunden gestartet, die mit je 32 Prozessoren von der *bcsstk35*-, der *cvxbqp1*- und der *kurbel*-Matrix alle kleinsten Eigenwerte berechnen sollten, von der *kurbel*-Matrix zudem auch die inneren Eigenwerte. Mit 64 Prozessoren bei gleicher Zeit sollten zudem kleinste und innere Eigenwerte der *thermal2*-Matrix bestimmt werden.

Selbst beim Einsatz einer derartigen Anzahl von Prozessoren über eine Zeit von einem Tag wurden einige Testläufe immer noch nicht terminiert, woran zu erkennen ist, dass es sich um nur äußerst schwer lösbare Problemstellungen handelt.

Die beiden nächsten Tabellen 6.9 und 6.10 zeigen für die *thermal2*-Matrix die Hochrechnungen der Laufzeiten zur Bestimmung von zehn und 50 kleinsten Eigenwerten. Wieder eindeutig erkennbar ist hier die Stärke des Jacobi-Davidson-Programms, zusammen mit MUMPS als Löser für die Korrekturgleichung zur Bestimmung der kleinsten Eigenwerte, speziell bei der Ausführung auf wenigen Prozessoren. So liegt der berechnete Speedup bis hin zu 32 Prozessoren oberhalb des linearen Speedups, und bei Berechnung mit bis zu acht Prozessoren sogar über dem Speedup, der erreicht wird wenn TFQMR als Löser verwendet wird.

Zur Bestimmung der 50 kleinsten Eigenwerte führte nur noch der Lauf mit MUMPS als Löser zu einer Konvergenz der Eigenwerte innerhalb der maximal möglichen Rechenzeit von 24 Stunden. Beim Einsatz von 64 Prozessoren wurde hier das Ergebnis aber auch erst nach einer Wall-Clock-Zeit von über 20 Stunden ausgegeben. Die approximierte Zeit für vier Prozessoren liegt bei über einer Woche Rechenzeit. Da auf einem Prozessor nicht ausreichend Speicherplatz vorhanden war, um dort von der *thermal2*-Matrix die 50 kleinsten Eigenwerten berechnen zu können, wurde der Speedup bezogen auf die Laufzeit mit vier Prozessoren.

Auch die Testläufe, die die zehn Eigenwerte aus dem Inneren des Spektrums berechnen sollten, sind nach 24 Stunden nicht terminiert. Diese Problemstellungen sind derart aufwändig, dass noch mehr Prozessoren eingesetzt werden müssten, um in der maximal möglichen Rechenzeit von 24 Stunden auf der JUMP ein entsprechendes Ergebnis zu erzielen.

Ein ähnliches Verhalten wurde bei den selben eingestellten Parametern für die *kurbel*-Matrix festgestellt, was auf die Größe der Matrix, die Anzahl an Nicht-Null-Elementen und damit auch auf die Schwierigkeit der Problemstellung zurückzuführen ist.

Anz. Proz.	MUMPS 10 kl. EW (Appr.)	TFQMR 10 kl. EW (Appr.)	MUMPS Speedup	TFQMR Speedup
1	1 Tag 6 Std.	1 Tag 11 Std.	1	1
4	5 Std. 40 Min.	6 Std. 54 Min.	5.31	5.08
8	4 Std. 6 Min	5 Std. 42 Min.	7.32	6.15
16	1 Std. 36 Min.	1 Std. 40 Min.	18.82	21.09
32	50 Min. 36 Sek.	1 Std. 8 Min.	35.66	31.09
64	36 Min. 50 Sek.	32 Min. 2 Sek.	48.99	65.66

Tabelle 6.9: Laufzeit-Approximationen für 10 kleinste Eigenwerte der Matrix *thermal2*

Anz. Proz.	MUMPS 50 kl. EW (Appr.)	MUMPS Speedup
4	7 Tage 13 Std.	4
8	4 Tage 12 Std.	6.69
16	1 Tag 19 Std.	16.99
32	1 Tag 9 Std.	21.8
64	20 Std. 38 Min.	35.2

Tabelle 6.10: Laufzeit-Approximationen für 50 kleinste Eigenwerte der Matrix *thermal2*

Etwas differenzierter gestalten sich die Ergebnisse der Approximationen an die Laufzeiten für kleinste Eigenwerte der Matrizen *bcsstk35* und *cvxbqp1*. Die Tabelle 6.11 zeigt die Ergebnisse der Annäherungen für jeweils einen kleinsten Eigenwert der Matrix *bcsstk35* mit beiden zu testenden Lösern. Wie auch für die Matrix *cvxbqp1* wurde das Jacobi-Davidson-Programm mit 32 Prozessoren ausgeführt bei einer Wall-Clock-Zeit von 24 Stunden. Als Ergebnis der Läufe war festzustellen, dass lediglich die Berechnung eines kleinsten Eigenwertes bei beiden Matrizen in dieser Zeit terminiert wurde und zwar nicht durch Erreichen der geforderten Genauigkeit, sondern nur, weil das Maximum an Iterationen, also 1 000 000, erreicht wurde. Die letzten Näherungen, abgebildet in Tabelle 6.12 sprechen zwar für eine Konvergenz des Verfahrens, jedoch nur eine äußerst langsame.

Anz. Proz.	MUMPS 1 kl. EW (Appr.)	TFQMR 1 kl. EW (Appr.)	MUMPS Speedup	TFQMR Speedup
1	23 Std. 45 Min.	5 Tage 11 Std.	1	1
4	7 Std. 24 Min.	1 Tag 7 Std.	3.21	4.75
8	4 Std. 29 Min	15 Std. 46 Min.	5.3	8.34
16	3 Std. 11 Min.	8 Std. 11 Min.	7.41	16.06
32	4 Std. 29 Min.	4 Std. 48 Min.	5.3	27.3

Tabelle 6.11: Laufzeit-Approximationen für einen kleinsten Eigenwert der Matrix *bcsstk35*

MUMPS 1 kl. EW	Residuum	TFQMR 1 kl. EW	Residuum
3.252404775325	.165170773695D+02	3.562661054434	.329113842023D+02
3.252401832779	.345991488354D+02	3.562658029833	.224523739308D+02
3.252398890192	.165170499730D+02	3.562655005233	.329113316911D+02
3.252395947660	.345990914439D+02	3.562651980634	.224523381222D+02
3.252393005069	.165170225845D+02	3.562648956060	.329112791943D+02
3.252390062522	.345990340619D+02	3.562645931470	.224523023133D+02

Tabelle 6.12: Die letzten Iterationen bei der Suche nach dem kleinsten Eigenwert von *bcsstk35*

Deutlich erkennbar ist auch hier wieder, dass TFQMR wesentlich besser skaliert. Die Gesamtlaufzeit hingegen fällt bei der Berechnung mit wenigen Prozessoren unter Einsatz des MUMPS-Lösers um einiges geringer aus.

Noch deutlicher fällt die Tatsache auf, dass es einen Leistungseinbruch bei der Lösung des Problems mit MUMPS gibt, wenn mehr als 16 Prozessoren eingesetzt werden. Dieses Phänomen ist während den Testläufen des Öfteren aufgetreten und wurde aufgrund dessen auch speziell noch untersucht. Im nächsten Teil dieses Kapitels wird gezeigt, dass die Lösungs-Phase von MUMPS, auch wenn zeitlich nur gering bemerkbar, unter Einsatz von mehr als 8 Prozessoren einen Geschwindigkeitsverlust hinnehmen muss.
Bei den meisten Beispielen ist dieses Verhalten nicht so extrem aufgefallen, da viele der betrachteten Testläufe mit weit weniger als 1000 Jacobi-Davidson-Iterationen auskamen. Bei der Berechnung für einen kleinsten Eigenwert der beiden Matrizen, *bcsstk35* und *cvxbqp1*, wurden jedoch jedesmal die maximal möglichen 1 000 000 Jacobi-Davidson-Iterationen durchgeführt, bis die dann derzeitige Eigenwertnäherungen ausgegeben wurden. Dies ist darauf zurückzuführen, dass eine Iteration im Durchschnitt nur etwas mehr als eine hundertstel Sekunde zur Ausführung benötigte, da die Matrizen von relativ kleiner Dimension und niedrigem Besetzungsgrad sind. Da aber nun in jeder dieser Iterationen auch der Anzahl an Unterraum-Erweiterungen entsprechend oft die Korrekturgleichung gelöst werden muss, wird auch in jeder dieser Iterationen die Lösungs-Phase von MUMPS einmal aufgerufen. Da, wie oben bereits erwähnt, bei hoher Prozessorzahl die Lösungs-Phase von MUMPS an Geschwindigkeit verliert, ist damit zu erklären, warum ein rückläufiger Speedup bei der Suche nach einem kleinsten Eigenwert der beiden Matrizen, *bcsstk35* und *cvxbqp1*, mit MUMPS zu verzeichnen ist, je mehr Prozessoren zum Einsatz kommen.

Die Laufzeiten der drei MUMPS-Phasen

In Kapitel 4.3 wurde bereits darauf hingewiesen, dass es sich als sinnvoll erweist, die Faktorisierung der linken Seite der Korrekturgleichung $(A - \theta I)$ nur einmal zu Beginn des Jacobi-Davidson-Algorithmus mit der ersten Eigenwertnäherung θ durchzuführen, auch wenn man dadurch mit einer fehlerbehafteten linken Seite spätere Lösungs-Phasen durchlaufen muss.

Die Abbildung 6.9 zeigt die Zeiten der einzelnen MUMPS-Phasen bei drei der Testmatrizen mit unterschiedlicher Größe und beim Einsatz von unterschiedlicher Anzahl an Prozessoren. Man sieht hier ganz deutlich, dass im günstigsten Fall ungefähr ein Faktor 1000 zwischen der Zeit für eine Faktorisierungs- und der für eine Lösungs-Phase liegt. Sprich eine einzige Faktorisierung müsste mindestens den selben Gewinn, also eine Verbesserung des Ergebnisses erzielen, wie 1000 Lösungsphasen. Tests haben gezeigt, dass dieser Fall nie eintritt. Aus diesem Grund wurde auf weitere Faktorisierungen des MUMPS-Lösers während der Ausführung des Jacobi-Davidson-Programms verzichtet. Je größer die Matrix wird, desto größer wird dieser Zeitunterschied der MUMPS-Phasen. Während die Zeiten für eine Faktorisierung der Matrix relativ zur Matrixgröße ansteigt, bleiben die Zeiten der Lösungs-Phasen nahezu identisch klein. Auch dieser Aspekt spricht speziell bei der Berechnung großer Matrizen gegen einen weitere Ausführung der Faktorisierungs-Phase von MUMPS während des Programmablaufs.

Während der Laufzeit-Analyse der drei MUMPS-Phasen ist ein Aspekt negativ aufgefallen, der auch während den Testläufen zur Eigenwertberechnung bereits angesprochen wurde, nämlich, dass mit zunehmender Anzahl an Prozessoren die Zeit für die Lösungsphase ansteigt. Dies spricht für eine äußerst schlechte Skalierung dieser MUMPS-Phase. Der entsprechende Speedup der einzelnen Phasen für die drei Matrizen *bcsstk35*, *kurbel* und *thermal2* ist in der Grafik 6.10 gegen die Anzahl der eingesetzten Prozessoren aufgetragen worden.

Während die Analyse- und die Faktorisierungs-Phase bis zum Einsatz von acht Prozessoren zum Teil noch linearen oder gar superlinearen Speedup aufweisen, zeigt sich hier ganz deutlich die schlechte Skalierung der Lösungs-Phase von MUMPS.

Literatur bezüglich Performance-Tests mit dem MUMPS-Löser ist nur wenig vorhanden. Die Beiträge, die man jedoch findet haben stets MUMPS mit allen drei Phasen gleichzeitig getestet, und kamen dann zu Ergebnissen, die besagen, dass bis zu einer Anzahl von vier oder acht Prozessoren der MUMPS-Löser eine gute Skalierung aufweist, bei mehr Prozessoren aber zum Teil sogar langsamer wird. Die Abbildung 6.11 bestätigt genau diese Aussagen. Eine realtiv gute Skalierung ist bis maximal acht Prozessoren zu erkennen für die *bcsstk35*- und die *thermal*-Matrix. Bei der *kurbel*-Matrix wird lediglich bis zu vier Prozessoren ausreichend gut skaliert.

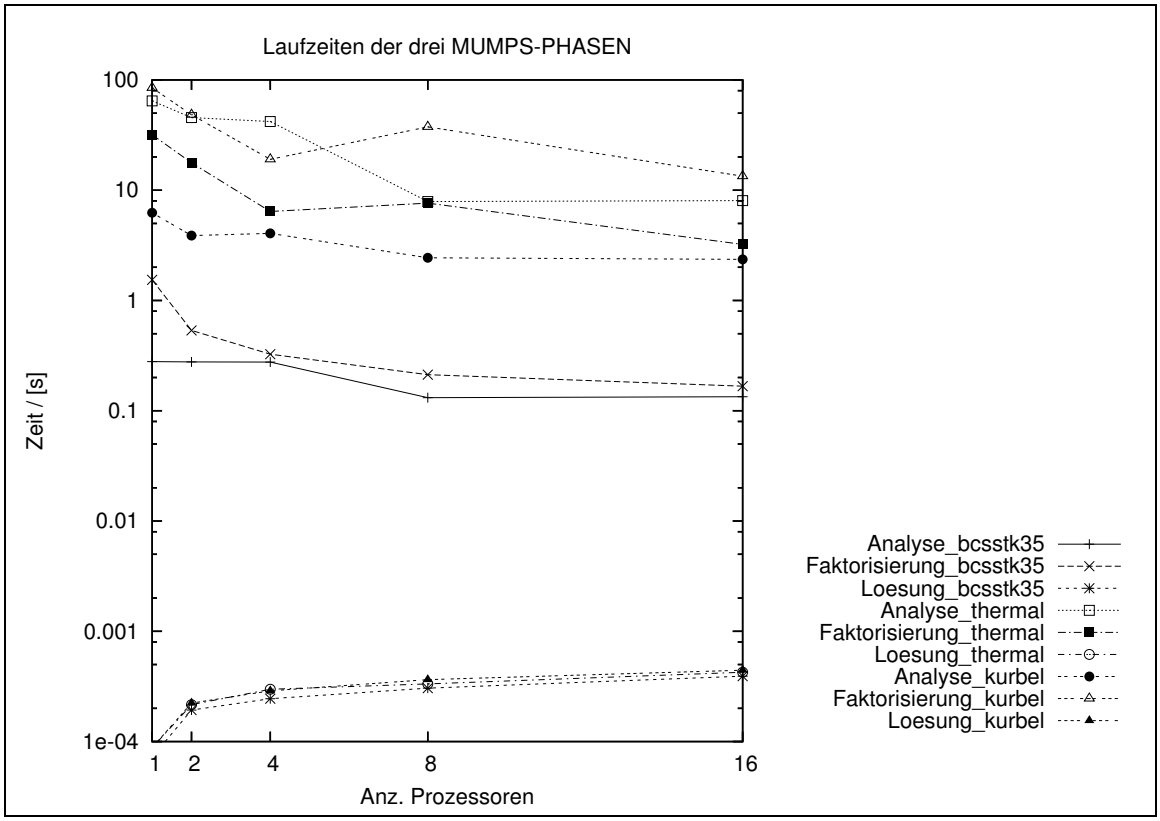


Abbildung 6.9: Laufzeiten der MUMPS-Phasen für die Matrizen *bcsstk35*, *kurbel* und *thermal2*

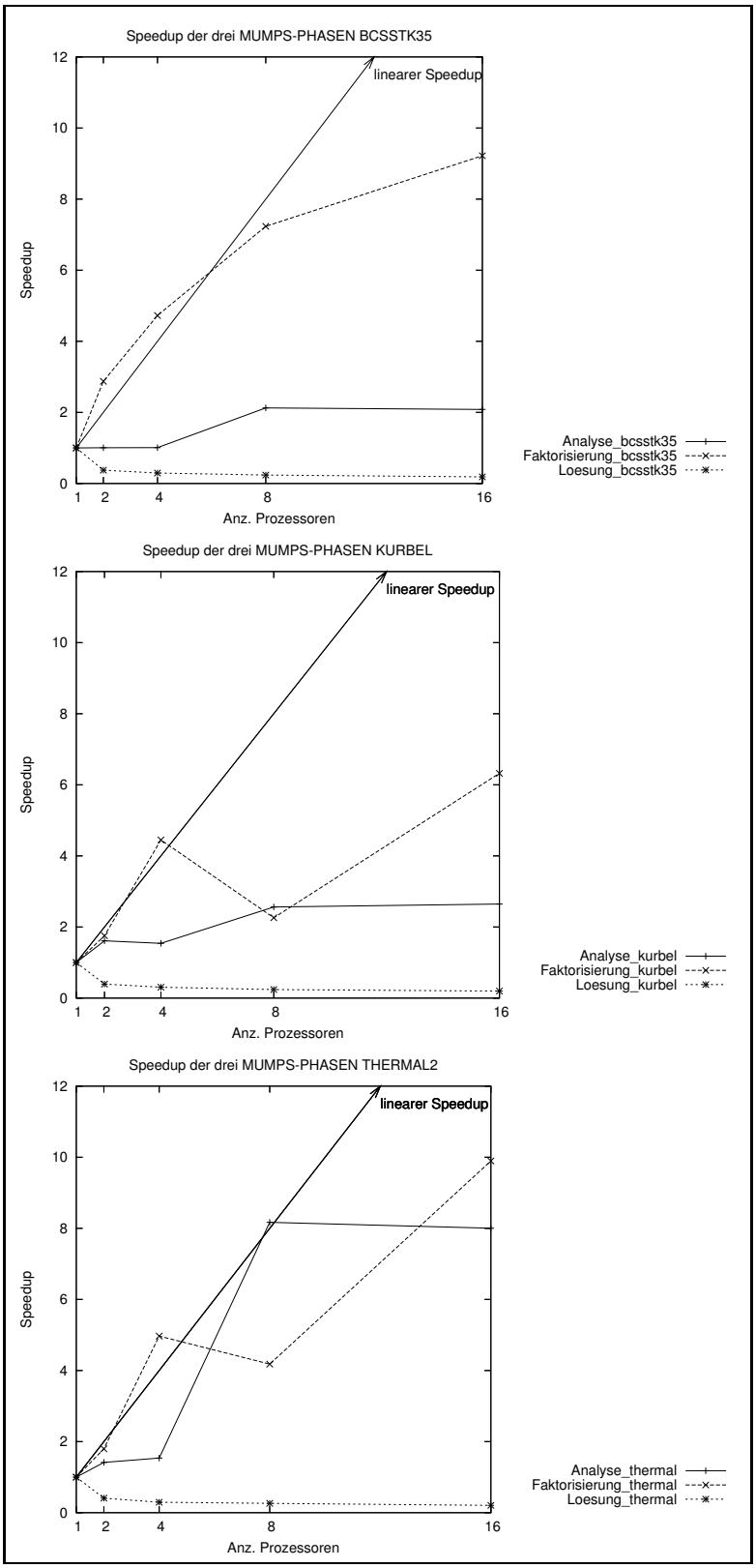


Abbildung 6.10: Speedup der Gesamtlaufzeiten für MUMPS für *bcsstk35*, *kurbel* und *thermal2*

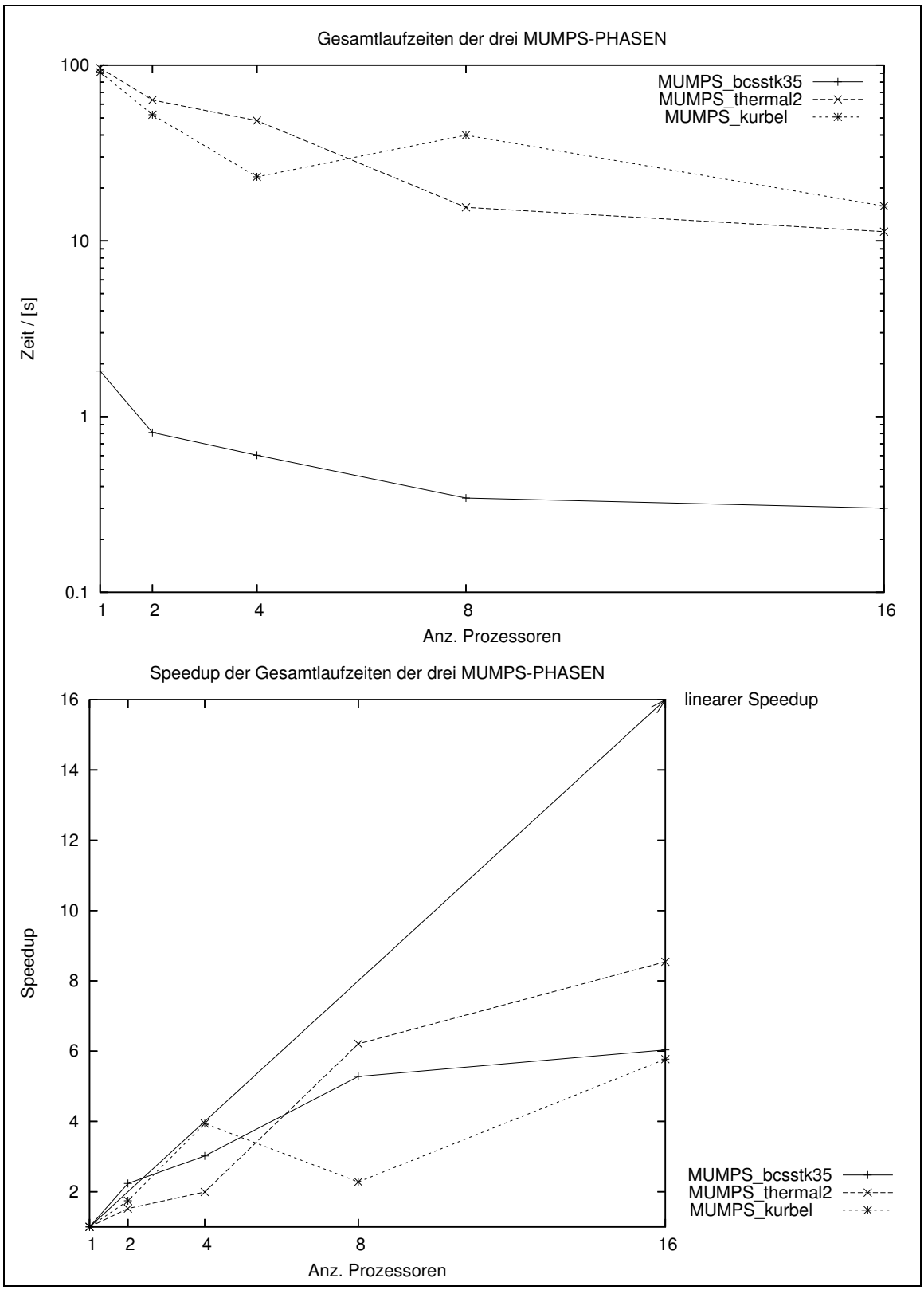


Abbildung 6.11: Gesamtlaufzeiten und Speedup für MUMPS bei Matrix *bcsstk35*, *kurbel* und *thermal2*

Schlusswort zu den Testläufen

Das zu erreichende Ziel der Testläufe mit dem Jacobi-Davidson-Programm unter Einsatz der beiden Gleichungssystemlöser MUMPS und TFQMR war, künftigen Benutzern der Eigenwertlöser-Bibliothek im Voraus sagen zu können für welche Aufgabenstellungen sich die Verwendung des einen oder des anderen Lösers besser eignet.

Über die Testläufe hinweg hat sich mit der Zeit ein klares Muster abgebildet, wann der Löser MUMPS zur Lösung der Korrekturgleichung dem TFQMR-Löser vorgezogen werden sollte.

MUMPS lohnt sich vor allem bei der Berechnung größter Eigenwerte, wenn die Problemgröße ausreichend hoch ist, sprich, bei Matrizen, deren Dimension mindestens 50 000 entspricht und von der mindestens zehn Eigenwerte bestimmt werden sollen. Je größer die Matrix und je mehr Eigenwerte bestimmt werden, desto überlegener wird der MUMPS-Löser.

Dabei sollten bei den Berechnungen der Eigenwerte jedoch nicht mehr als acht Prozessoren für sehr große und nicht mehr als vier Prozessoren für kleinere Matrizen eingesetzt werden, da mit steigender Anzahl von Prozessoren stets ein geringerer Vorteil von MUMPS gegenüber TFQMR zu beobachten war.

Einen weiteren Leistungszuwachs erhält das Jacobi-Davidson-Programm unter Einsatz von MUMPS wenn speziell die kleinsten Eigenwerte einer Matrix berechnet werden sollen.

Die Zeitmessungen zur Berechnung innerer Eigenwerte und Eigenwerte von unsymmetrischen Matrizen ergab ein ganz anderes Ergebnis. Hier lagen die Ausführungszeiten mit TFQMR als Gleichungssystem-Löser immer weit unter denen, die mit MUMPS gemessen wurden.

Die Untersuchungen, die mit dem Programm unter Verwendung der beiden unterschiedlichen Lösern durchgeführt wurden fanden alle an ganz speziellen Matrizen statt, jeweils mit unterschiedlichen Dimensionen und Strukturen. Deshalb können die oben genannten Ergebnisse lediglich als Leitfaden genutzt werden, nicht aber als allgemeingültige Aussagen, besonders die Ergebnisse zu unsymmetrischen Matrizen, denn für diese gab es nur sehr wenige Tests.

Kapitel 7

Zusammenfassung und Ausblick

Ziel dieser Diplomarbeit war die Erweiterung eines Moduls, innerhalb des bestehenden parallelen Jacobi-Davidson-Programms, zur Lösung der Korrekturgleichung um den direkten Multifrontal-Löser MUMPS.
Die durchgeführten Tests des Programms nach der Einbindung des neuen Lösers haben gezeigt, dass unter gewissen Voraussetzungen wesentliche Verbesserungen bezüglich der benötigten Rechenzeit zu erzielen sind, wenn dieser Löser statt dem CG-artigen TFQMR eingesetzt wird. Speziell bei der Berechnung kleinster Eigenwerte und bei der Suche nach vielen größten Eigenwerten hochdimensionierter Matrizen konnten die benötigten Rechenzeiten zum Teil drastisch gesenkt werden, hauptsächlich jedoch, wenn gleichzeitig auch nur eine geringe Anzahl von Prozessoren zur Lösung des Problems eingesetzt wurden.

Die Berechnung von Eigenwerten aus dem Inneren des Spektrums macht trotz Einbindung eines direkten Lösers immer noch Probleme, hier sind weitere Untersuchungen des Jacobi-Davidson-Programms notwenig, gegebenenfalls auch noch die Einbindung weiterer stabiler Löser-Routinen.

Bei der MUMPS-Version, die in das bestehende Eigenwert-Programm eingebunden wurde, handelt es sich um die Version, die mit reellen Werten und doppelter Genauigkeit rechnet. Eine weitere Version, die es zulässt Gleichungssysteme mit komplexen Einträgen zu lösen könnte im Zuge einer Weiterentwicklung des Jacobi-Davidson-Programms, die es ermöglicht auch Eigenwerte allgemeiner unsymmetrischer Matrizen zu berechnen, mit in das Programm einfließen.

Anhang A

Mathematische Grundlagen

In diesem Kapitel werden nun anschließend noch einige mathematische Grundlagen beschrieben, die in den vorangegangenen Kapiteln benötigt wurden. Für einen ausführlichen Überblick zum Thema Lineare Algebra siehe [17] und für weitere Informationen zur numerischen Mathematik siehe [15], [16].

A.1 Allgemeines zu Vektoren

Definition A.2. *Seien $x, y \in \mathbb{R}^m$ zwei reelle Vektoren, dann nennt man:*

$$< x, y >= \sum_{i=1}^{m} x_i y_i$$

Skalarprodukt der Vektoren x und y.

Seien $u, v \in \mathbb{C}^m$ zwei komplexe Vektoren, dann nennt man:

$$< u, v >= \sum_{i=1}^{m} \overline{u}_i v_i$$

Skalarprodukt der Vektoren u und v, wobei $\overline{u}_i$ die komplex konjugierte Komponente von u_i beschreibt.

Definition A.3. *Die 2er-Norm, die auch euklidische Norm genannt wird, hat folgende Gestalt:*

$$\|x\|_2 = \sqrt{< x, x >}$$

Sie beschreibt die Länge eines Vektors x im euklidischen Raum.

A.4 Eigenschaften von Matrizen

Definition A.5. *Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt genau dann symmetrisch, wenn sie gleich ihrer Transponierten ist:*

$$A = A^T$$

Definition A.6. *Sei $A \in \mathbb{R}^{n\times n}$ eine Matrix, dann heißt A genau dann anti- oder schiefsymmetrisch, wenn gilt:*

$$A = -A^T$$

Definition A.7. *Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt orthogonal, falls gilt:*

$$A^T A = E$$

mit der transponierten Matrix A^T und der Einheitsmatrix E.

Bemerkung A.8. *Folgende Aussagen sind äquivalent:*

- *A ist eine orthogonale Matrix*
- *Die Zeilen- und Spaltenvektoren $a^{(i)}$ von A bilden ein Orthonormalsystem bezüglich des Skalarprodukts im $\mathbb{R}^n$, sprich:*

$$< a^{(i)}, a^{(j)} >= \delta_{ij} = \left\{ \begin{array}{cc} 1 & \text{für } i = j \\ 0 & sonst \end{array} \right\} \forall\, i, j = 1, .., n$$

- *$AA^T = E$, das heißt, $AA^T = A^T A$*
- *A ist invertierbar und es gilt: $A^{-1} = A^T$*

Definition A.9. *Eine $n \times n$ Matrix A heißt positiv definit, falls gilt:*

$$< x, Ax > \quad > 0 \quad \forall\, x \neq 0$$

Möchte man die Definition verallgemeinern, sodass man sie auch auf den komplexen Zahlenbereich anwenden kann, so definiert man eine Matrix als positiv definit, wenn gilt:

$$Re(< x, Ax >) \; > \; 0 \quad \forall\, x \neq 0$$

Bemerkung A.10. *Folgende Aussagen sind äquivalent:*

- *A ist positiv definit*
- *A hat nur positive Eigenwerte*
- *A ist invertierbar und A^{-1} ist ebenfalls positiv definit*

Bezeichnung A.11. *Neben den klassischen und geläufigen Besetzungsarten für Matrizen, wie untere und obere Dreiecksmatrix, oder Bandmatrix, will hier noch folgende genannt sein, die sogenannte Sparse-Matrix. Wie der Name Sparse (dt. spärlich) schon besagt, handelt es sich hierbei um dünnbesetzte Matrizen.*

A.12 Eigenwerte und Ritzwerte

Definition A.13. *Seien $A \in \mathbb{R}^{n\times n}$ eine Matrix, $u \in \mathbb{R}^n$, $u \neq 0$ ein Vektor und λ ein Skalar, die die Gleichung*

$$Au = \lambda u$$

erfüllen, so nennt man λ Eigenwert und u Eigenvektor der Matrix A. Zusammengefasst zu (λ, u) heißen sie auch Eigenpaar.

Bemerkung A.14. *Die Eigenwerte von A entsprechen den Nullstellen des charakteristischen Polynoms*

$$det(\lambda I - A) = 0$$

Bezeichnung A.15. *Seien $A \in R^{n\times n}$ eine Matrix, $\theta \in \mathbb{R}$ eine Näherung für den Eigenwert λ und $y \in \mathbb{R}^n$, $y \neq 0$ eine Näherung für den zugehörigen Eigenvektor u der Matrix A, dann bezeichnet man mit*

$$r = Ay - \theta y$$

das Residuum des genäherten Eigenpaars.

Für numerische Berechnungen von Eigenwerten großer Matrizen, werden häufig Unterraumtechniken angewandt, um das betrachtete Problem zu vereinfachen. Dazu folgendes:

Definition A.16. *$U \subset \mathbb{R}^n$ heißt Unterraum des R^n, falls gilt:*

- $U \neq \varnothing$
- $x + y \in U \quad \forall\, x, y \in U$ *(Additivität)*
- $\alpha \cdot x \in U \quad \forall\, x \in U, \alpha \in \mathbb{R}$ *(Homogenität)*

Zudem muss man beachten, dass ein Unterraum durch seine Basis festgelegt wird, die folgendermaßen definiert ist:

Definition A.17. *Seien $U \subset \mathbb{R}^n$ ein Unterraum und $u_i \in \mathbb{R}^n$ mit $i = 1, .., k$ und $k < n$ Vektoren, dann heißt $u_1, .., u_k$ Basis des Unterraums U, falls:*

- *die Vektoren u_i linear unabhängig sind und*
- *die lineare Hülle $L = \left\{ \sum_{i=1}^{k} \alpha_i u_i,\ \alpha_i \in \mathbb{R},\ i = 1, .., k \right\} = U$ ist*

Desweiteren wird mit $dim(U) = k$ die Dimension des Unterraums bezeichnet.

Viele iterative Löser linearer Gleichungssysteme nennt man auch Unterraumverfahren, weil sie das gestellte Problem in Unterräume projizieren. Zu ihnen gehören gewisse lineare Teilräume des $\mathbb{C}^n$, die sogenannten Krylov-Räume.

Definition A.18. *Sei dazu eine quadratische Matrix $A \in \mathbb{C}^{n\times n}$ gegeben, sowie ein Vektor $q \in \mathbb{C}^n$, dann ist der Krylov-Raum von A über q definiert als:*

$$K_m(A, q) = span\left[q, Aq, .., A^{m-1}q\right] \ , \quad m \geq 1 \ , \quad K_0(A, q) = \{0\}$$

Der Vektor q wird dabei als Startvektor der Krylov-Sequenz bezeichnet.

Bemerkung A.19. *Seien $t \in \mathbb{C}^n$ ein Vektor und $U \subset \mathbb{C}^n$ ein Unterraum, und es gelte:*

$$t \perp u \quad \forall\, u \in U$$

so schreibt man

$$t \perp U$$

und definiert

$$U^{\perp} = \{t \in \mathbb{C}^n : \ t \perp U\}$$

als orthogonales Komplement des Unterraums U.

Repräsentiert A_k eine Projektion von einer Matrix A auf einen niederdimensionalen Unterraum U, so nennt man die Eigenwerte von A_k Ritzwerte der Matrix A im Bezug auf den Unterraum U.

Definition A.20. *Seien $U \subset \mathbb{R}^n$ ein Unterraum, $A \in \mathbb{R}^{n \times n}$ eine Matrix, $z \in U, \ z \neq 0$ ein Vektor und ein $\theta \in \mathbb{C}$ ein Skalar, die die Ritz-Galerkin-Bedingung:*

$$Az - \theta z \perp U$$

erfüllen, dann nennt man θ Ritzwert und z Ritzvektor zu A bezüglich des Unterraums U. (θ, z) ist dann das entsprechende Ritzpaar.

Bei Eigenwertberechnungen kommt es zum Teil vor, dass man Eigenwerte aus dem Inneren des Spektrums berechnen muss. Diese werden über sogenannte harmonische Ritzwerte bestimmt.

Definition A.21. *Seien $U \subset \mathbb{R}^n$ ein Unterraum, $A \in \mathbb{R}^{n \times n}$ eine Matrix, $\theta \in \mathbb{C}$ ein Skalar, dann ist θ genau dann ein harmonischer Ritzwert von A bezüglich des Unterraums U, wenn θ^{-1} ein Ritzwert von A^{-1} in Bezug auf den Unterraum U ist.*

Literaturverzeichnis

[1] Joseph W. H. Liu (1992)
The multifrontal method for sparse matrix solution: theory and practice
SIAM Review, Vol. 34, No. 1, pp. 82-109

[2] offizielle MUMPS Homepage
http://graal.ens-lyon.fr/MUMPS/index.html

[3] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent (1998)
Multifrontal Parallel Distributed Symmetric and Unsymmetric Solvers
http://epubs.cclrc.ac.uk/bitstream/154/raltr-1998051.pdf

[4] Patrick R. Amestoy, Chiara Puglisi
An Unsymmetrized Multifrontal LU Factorization
http://citeseer.ist.psu.edu/cache/papers/cs/29457/
http:zSzzSzwww.enseeiht.frzSzapozSzMUMPSzSzampu_revised.pdf/
amestoy00unsymmetrized.pdf

[5] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, Jacko Koster (2001)
A Fully Asynchroneus Multifrontal Solver Using Distributed Dynamic Scheduling
SIAM J. Matrix Anal. Appl., Vol. 23, No.1, pp. 15-41

[6] I.S. Duff, J.K Reid (1983)
The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations
ACM Trans. Math. Software, Vol 9, No. 3, pp. 302-325

[7] I.S. Duff, J.K Reid (1984)
The Multifrontal Solution of Unsymmetric Sets of Linear Equations
Siam J. Sci. Stat. Comput. Vol 5., No. 3, pp. 633-641

[8] I.S. Duff, A.M.Erisman, J.K Reid (1986)
Direct Methods for Sparse Matrices (Monographs on Numerical Analysis)
Oxford University Press (ISBN: 0-19-853421-3)

[9] I.S. Duff (2004)
MA57- A new code for the solution of sparse symmetric definite and indefinite systems
ACM Trans. Math. Software, Vol. 30, No. 2, pp. 118 - 144

[10] User's Guide (2006)
MUltifrontal Massively Parallel Solver (MUMPS Version 4.6.2)
http://www.enseeiht.fr/irit/Numerique/Noail/MUMPS/userguide_4.6.2.pdf

[11] Gerard L. G. Sleijpen, Henk A. van der Vorst (1996)
A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems
SIAM J. Matrix Anal. Appl. Vol. 17, No. 2, pp. 401-425

[12] Z. Bai, J.Demmel, J. Dongarra, A. Ruhe, H. van der Vorst (2000)
Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide
Philadelphia, Pa.: SIAM (ISBN: 0-89871-471-0)

[13] Roland W. Freund und Noël M. Nachtigal (1994)
QMR: A Quasi-Minimal Residual Method for Non-Hermitian Systems
Numerische Mathematik Vol. 60, No. 3

[14] Roland W. Freund (1993)
A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Systems
SIAM J. Sci. Comput. Vol. 14, pp.470-482

[15] Martin Hanke-Bourgeois (2002)
Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens
Teubner (ISBN 3-519-00356-2)

[16] Peter Deuflhard und Andreas Hohmann (1993)
Numerische Mathematik I; Eine algorithmisch orientierte Einführung
Walter de Gruyter (ISBN 3-11-013974-X)

[17] Howard Anton (1995)
Lineare Algebra
Spektrum (ISBN 3-86025-137-6)

[18] René Puttin (2005)
Modulare und parallele Implementierung des Jacobi-Davidson-Verfahrens
Interner Bericht des Forschungszentrums Jülich
(FZJ-ZAM-IB-2005-17)

[19] Frank Schmidt (2006)
Performance-Messungen eines parallelen Jacobi-Davidson Eigenwertlösers
Beitrag zum Wissenschaftlichen Rechnen - Ergebnisse des Gaststudentenprogramms 2006 des John von Neumann-Instituts für Computing
Interner Bericht des Forschungszentrums Jülich
(KFA-ZAM-IB-2006-14)

[20] Martin Bücker, Achim Basermann (1994)
A Comparison of QMR, CGS und TFQMR on Distributed Memory Machines
Interner Bericht des Forschungszentrums Jülich
(KFA-ZAM-IB-9412)

[21] Achim Basermann (1995)
Iterative Verfahren für dünnbesetzte Matrizen zur Lösung technischer Probleme auf massiv-parallelen Systmen
Bericht des Forschungszentrums Jülich
(KFA-ZAM-JÜL-3015)

[22] Martin Bücker (1994)
Parallelisierung der QMR-Methode zur Lösung linearer Gleichungssysteme
Bericht des Forschungszentrums Jülich
(KFA-ZAM-JÜL-2955)

[23] Gerd Groten (1999)
Programmieren in Fortran 90/95 (Vorlesungsskript)
Benutzerhandbuch des Forschungszentrums Jülich
(KFA-ZAM-BHB-0124)

[24] University Tennessee (1995)
MPI: A Message-Passing Interface Standard

[25] Frank Elsner (2000)
Einführung in Gnuplot
http://www.rz.uni-osnabrueck.de/Zum_Nachlesen/Skripte_Tutorials/Gnuplot_Einfuehrung/pdf/gnuplot.pdf

[26] FZJ Dokumentation der Supercomputer
http://jumpdoc.fz-juelich.de/

[27] Download-Seite für die parallele Eigenwertlöser-Bibliothek des
Zentralinstituts für Angewandte Mathematik des Forschungszentrums in Jülich
http://www.fz-juelich.de/zam/appliedmath/eigenlib/download

[28] Sparse-Matrix Sammlung der Universität von Florida
http://www.cise.ufl.edu/research/sparse/matrices/index.html

[29] National Institute of Standards and Technology,*Matrix Market*
http://math.nist.gov/MatrixMarket

[30] Online-Lexikon Wikipedia
http://de.wikipedia.org