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

Interner Bericht

Performance-Analysen von seriellen und parallelen Algorithmen zur Bestimmung von Eigenwerten

Angela Etzbach

FZJ-ZAM-IB-2004-04

April 2004

(letzte Änderung: 19.04.2004)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Performance-Analysen von seriellen und parallelen Algorithmen zur Bestimmung von Eigenwerten

In dieser Arbeit werden die in den Paketen LAPACK und ScaLAPACK gegenwärtig verfügbaren Eigenwertproblemlöser untersucht. Ziel ist neben einem Performance-Vergleich die Entwicklung und Bewertung von Modellfunktionen, die es erlauben, die zu erwartende serielle und parallele Rechenzeit in Abhängigkeit von Parametern des Computersystems und der Eigenwertproblemlöser abzuschätzen.
Die zugrunde liegenden Algorithmen werden kurz dargestellt, ebenso die Rechnerarchitekturen von ZAMpano und IBM Regatta. Die Zielsetzung, Durchführung und Auswertung der seriellen und parallelen Performance-Untersuchungen werden erläutert. So wird dem Nutzer eine sinnvolle Auswahl des am besten geeigneten Verfahrens und die Anforderung der erforderlichen Computerressourcen ermöglicht.

Performance Analysis of Serial and Parallel Algorithms for Solving Eigenvalue Problems

In this diploma thesis, the actually available solvers for eigenvalue problems in the LAPACK and ScaLAPACK packages are tested. The intention is to compare the corresponding performances and to generate functions which allow to pre-estimate the expected serial and parallel computation time. This time is dependent on parameters as the computer system and the used eigensolver.
The algorithms of the eigenvalue solvers are displayed, as well as the architecture of the computer systems ZAMpano and IBM Regatta. The purpose, execution and analysis of the serial and parallel tests are described. Thus the user is enabled to take a reasonable choice of the most adequate method and to request the necessary computer ressources.

1 Einleitung

Viele Optimierungs- und Randwertprobleme in den Ingenieur- und Naturwissenschaften lassen sich auf Eigenwertprobleme zurückführen. Eigenwertprobleme lassen sich unter anderem dort finden, wo Schwingungen und Zyklen auftreten wie zum Beispiel Eigenschwingungen von Gebäuden und Brücken oder Schall und Resonanz im Konzertsaal.

In der Quantenchemie tauchen speziell die algebraischen Eigenwertprobleme in einigen zentralen Verfahren auf. Hier muss die Lösung der analytisch nicht zugänglichen elektronischen zeitunabhängigen Schrödingergleichung (eine lineare partielle Differentialgleichung 2. Ordnung mit 3N unabhängigen Variablen) approximiert werden. Hierzu wird die von den Koordinaten der N Elektronen abhängige Lösung in Linearkombinationen von Produkten von n Einteilchenfunktionen, dem sogenannten Basissatz, approximiert.
Im Rahmen der Hartree-Fock-Näherung [1, 2] wird die Wellenfunktion in einer Slater-Determinante approximiert. Die freien Parameter der Wellenfunktion werden so bestimmt, dass die Gesamtenergie minimiert wird und die Wellenfunktion normiert ist. Dies entspricht der Lösung eines Eigenwertproblems der Größe des Basissatzes (typischerweise $n > 5N$), so dass für Moleküle mit 100 leichten Atomen und ca. 400 Elektronen bereits eine Dimension des Eigenwertproblems von mindestens 2000 erreicht wird. Im Rahmen des iterativen Lösungsalgorithmus' muss ein Eigenwertproblem gelöst werden, wobei alle Eigenwerte und alle Eigenvektoren benötigt werden.
Da die Rechenzeit zur Lösung eines Eigenwertproblems (Eigenwerte und Eigenvektoren) eine Komplexität von N^3 aufweist, wird dieser Teil des Hartree-Fock-Verfahrens die Rechenzeit für große Molekülgrößen dominieren, so dass es sinnvoll und notwendig ist, die verfügbaren seriellen und parallelen Eigenwertproblemlöser zu bewerten.

In dieser Arbeit werden die in den Paketen LAPACK[3] und ScaLAPACK[4] gegenwärtig verfügbaren Eigenwertproblemlöser untersucht. Ziel ist neben einem relativen seriellen und parallelen Performancevergleich die Entwicklung und Bewertung von Modellfunktionen, die es erlauben, die zu erwartende serielle und parallele Rechenzeit in Abhängigkeit von Parametern des Computersystems und der Eigenwertproblemlöser abzuschätzen. Dies erlaubt dem Nutzer solcher "Black-Box"-Verfahren die sinnvolle Auswahl des geeignetsten Verfahrens und die Anforderung der erforderlichen Computerresourcen ohne zu den häufig praktizierten teueren und frustierenden "trial-and-error" Vorgehensweisen greifen zu müssen.

2 Theorie und Grundlagen

In diesem Kapitel werden die allgemeinen Strategien zur Lösung des Eigenwert-Problems vorgestellt.

2.1 Mathematische Grundlagen zur Eigenwert-Problematik

Eigenwerte und Eigenvektoren: Sei $A \in \mathbb{R}^{m \times m}$ eine quadratische Matrix. Dann ist $x \in \mathbb{R}^m$, mit $x \neq 0$, ein *Eigenvektor* von A, und $\lambda \in \mathbb{R}$ sein entsprechender *Eigenwert*, wenn

$$Ax = \lambda x. \tag{2.1}$$

Charakteristisches Polynom: Diese Matrizengleichung ist äquivalent zu $(A - \lambda E)x = 0$, und dieses homogene Gleichungssystem hat nichttriviale Lösungen genau dann, wenn

$$\det(A - \lambda E) = 0. \tag{2.2}$$

Die Lösungen dieser charakteristischen Gleichung sind die Eigenwerte λ. Die linke Seite dieser Gleichung ist das *charakteristische Polynom* p_A der Matrix A:

$$p_A(\lambda) = \det(A - \lambda E) \tag{2.3}$$

Eigenwerte: Die Eigenwerte λ_i der quadratischen Matrix A sind die Nullstellen ihres charakteristischen Polynoms.

$$p_A(\lambda) = 0. \tag{2.4}$$

Die Eigenwerte sind alle reell, wenn A eine symmetrische Matrix ist.

Eigenvektoren: Die zu einem Eigenwert λ gehörigen Eigenvektoren ergeben sich aus dem Gleichungssystem

$$(A - \lambda E) \cdot x = 0. \tag{2.5}$$

Eigenraum: Die Lösungsmenge dieses Gleichungssystems bildet für ein festes λ einen Vektorraum, den man als den zu λ gehörigen Eigenraum von A bezeichnet.

Eigenwert-Zerlegung: Eine *Eigenwert-Zerlegung* einer symmetrischen Matrix A ist eine Faktorisierung der Form

$$A = X \Lambda X^{-1} \tag{2.6}$$

mit

$$\Lambda = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_m \end{bmatrix}. \tag{2.7}$$

X ist eine reguläre Matrix, so dass A und $X \Lambda X^{-1}$ die selben Eigenwerte haben. Diese Definition kann umgeschrieben werden als

$$AX = X\Lambda, \tag{2.8}$$

und das ist

$$\left[\begin{array}{c} \\ A \\ \\ \end{array}\right] \left[\begin{array}{c|c|c|c} & & & \\ x_1 & x_2 & \dots & x_m \\ & & & \end{array}\right] = \left[\begin{array}{c|c|c|c} & & & \\ x_1 & x_2 & \dots & x_m \\ & & & \end{array}\right] \left[\begin{array}{cccc} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_m \end{array}\right]. \tag{2.9}$$

Wenn x_j die j-te Spalte von X, und λ_j der j-te Diagonaleintrag von Λ ist, dann gilt $Ax_j = \lambda_j x_j$. Die j-te Spalte von X ist ein Eigenvektor von A, und der j-te Eintrag von Λ ist der entsprechende Eigenwert.

Diagonalisierbarkeit: Besitzt A m linear unabhängige Eigenvektoren x_i mit zugehörigen Eigenwerten λ_i, dann ist $X = (x_1, ..., x_m)$ regulär und es gilt:

$$A = X\Lambda X^{-1}, \tag{2.10}$$

mit $\Lambda = diag(\lambda_1, ..., \lambda_m), X = (x_1, ..., x_m)$. Dann heißt A diagonalisierbar.

Mehrfacher Eigenwert: Ist $\lambda_i = \lambda_j$, $i \neq j$, so heißt λ_i *mehrfacher Eigenwert*.

Algebraische und geometrische Vielfachheit: Liegt ein mehrfacher Eigenwert vor, unterscheidet man die *algebraische Vielfachheit* und die *geometrische Vielfachheit*. Mit der algebraischen Vielfachheit bezeichnet man die Vielfachheit der Nullstellen des charakteristischen Polynoms p_a, während die geometrische Vielfachheit die Anzahl der linear unabhängigen zum Eigenwert gehörenden Eigenvektoren angibt.
Hierbei gilt folgende Relation:

$$1 \leq geometrischeVielfachheit \leq algebraischeVielfachheit$$

Symmetrisches Eigenwertproblem: Wenn A eine symmetrische Matrix ist, können die Eigenvektoren q_i so gewählt werden, dass sie eine Orthonormalbasis bilden. So ist $Q = (q_1, q_2, ..., q_n)$ eine Orthonormalmatrix (mit $Q^{-1} = Q^T$) und es gilt:

$$A = Q\Lambda Q^{-1} = Q\Lambda Q^T \quad mit \quad \Lambda = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_m \end{bmatrix} \tag{2.11}$$

Für Λ gilt:

$$\Lambda = Q^T AQ \tag{2.12}$$

Schur-Faktorisierung: Die meisten der heutzutage verwendeten Eigenwert-Algorithmen führen die Berechnung mittels einer approximierten Schur-Faktorisierung durch. Dazu muss die zu behandelnde Matrix A symmetrisch sein. Um die Schur-Faktorisierung $A = Q\Lambda Q^T$ zu berechnen, zerlegen wir die Matrix A durch eine Reihe elementarer unitärer Ähnlichkeitstransformationen, so dass das Produkt

$$\Lambda_j = Q_j^T ... Q_2^T Q_1^T A Q_1 Q_2 ... Q_j \tag{2.13}$$

für $j \to \infty$ gegen eine Diagonalmatrix Λ konvergiert.

Verallgemeinertes Eigenwertproblem: Um ein verallgemeinertes symmetrisches Eigenwertproblem $\tilde{A}\tilde{x} = \lambda B\tilde{x}$ zu lösen, wird es in ein spezielles Eigenwertproblem überführt. Dies kann mit einer Cholesky-Faktorisierung der Matrix B durchgeführt werden. Dazu muss B eine symmetrische positiv definite Matrix sein. Die Transformation des verallgemeinerten Eigenwertproblems in ein spezielles Eigenwertproblem geschieht durch folgende Umformungen:
Durch die Ersetzung von $B = LL^T$ ergibt sich für $\tilde{A}\tilde{x} = \lambda B\tilde{x}$:

$$\tilde{A}\tilde{x} = \lambda LL^T\tilde{x} \tag{2.14}$$

$$\tilde{A}L^{-T}L^T\tilde{x} = \lambda LL^T\tilde{x} \tag{2.15}$$

$$\underbrace{L^{-1}\tilde{A}L^{-T}}_{A}\underbrace{L^T\tilde{x}}_{x} = \lambda\underbrace{L^T\tilde{x}}_{x} \tag{2.16}$$

Die Eigenwerte des speziellen Eigenwertproblems stimmen mit den Eigenwerten des verallgemeinerten Eigenwertproblems überein. Zur Berechnung der Eigenvektoren x muss die Rücktransformation $L^T\tilde{x} = x$ berechnet werden.

In der vorliegenden Arbeit wird nur das spezielle Eigenwertproblem reeller symmetrischer Matrizen betrachtet: $Ax = \lambda x$.

2.2 Algorithmen zur Lösung des symmetrischen Eigenwertproblems

Das Eigenwertproblem kann als Nullstellenproblem eines Polynoms betrachtet werden. Sei $p_A(\lambda)$ ein normiertes Polynom mit

$$p(\lambda) = \lambda^m + a_{m-1}\lambda^{m-1} + ... + a_1\lambda + a_0. \tag{2.17}$$

So sind die Nullstellen von $p_A(\lambda)$ gleich den Eigenwerten der zu diesem Polynom gehörenden Matrix A.
Für die Nullstellenberechnung eines gewöhnlichen Polynoms mit einem Grad von fünf oder mehr existiert jedoch kein endlicher Algorithmus, gemäß folgendem Theorem:

Theorem 1 *Für jedes $5 \leq m$ gibt es ein Polynom $p(z)$ vom Grad m, mit rationalen Koeffizienten, das eine reelle Nullstelle $p(r) = 0$ hat, wobei r durch keinen Ausdruck mit rationalen Zahlen, Addition, Subtraktion, Multiplikation, Division oder Wurzelberechnung ausgedrückt werden kann.*

Es ist nicht möglich, exakte Nullstellen eines Polynoms in endlicher Anzahl von Elementaroperationen zu bestimmen. Numerische Algorithmen zur Eigenwertberechnung können daher nur iterative Verfahren sein. Ein Eigenwertproblemlöser muss versuchen, Sequenzen von Werten zu erzeugen, welche möglichst schnell gegen die Eigenwerte konvergieren.

Zwei Phasen: Die Schur-Faktorisierung (2.13) wird bei den meisten Verfahren zur Eigenwertberechnung in zwei Phasen aufgeteilt: Phase 1 transformiert die symmetrische Matrix A in eine tridiagonale Matrix T. Phase 2 transformiert anschließend die Matrix

T in eine Diagonalmatrix D.

$$\underbrace{\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}}_{A=A^T} \xrightarrow{Phase1} \underbrace{\begin{bmatrix} x & x & & & \\ x & x & x & & \\ & x & x & x & \\ & & x & x & x \\ & & & x & x \end{bmatrix}}_{T} \xrightarrow{Phase2} \underbrace{\begin{bmatrix} x & & & & \\ & x & & & \\ & & x & & \\ & & & x & \\ & & & & x \end{bmatrix}}_{D} \tag{2.18}$$

2.2.1 Reduktion auf Tridiagonalform

Um die Schur-Faktorisierung $A = Q\Lambda Q^T$ zu berechnen, werden unitäre Ähnlichkeitstransformationen auf die Matrix A angewandt, so dass alle Matrix-Elemente unterhalb der ersten Nebendiagonalen anschließend Null sind. Dazu werden geeignete Householder-Matrizen $Q_1^T, ..., Q_{n-2}^T$ konstruiert [5], um die Matrix A sukzessive zu transformieren. Durch Multiplikation der Matrix A mit Q_1^T werden alle Elemente unterhalb der Subdiagonalen der ersten Zeile annulliert. Wird Q_1^T von links an A multipliziert, entstehen Nullen in der ersten Spalte der dritten bis m-ten Zeile. Wird dann von rechts mit Q_1 multipliziert, entstehen Nullen in der ersten Zeile ab dem dritten Element.

$$\underbrace{\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}}_{A} \xrightarrow{Q_1^T \cdot} \underbrace{\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ 0 & x & x & x & x \\ 0 & x & x & x & x \\ 0 & x & x & x & x \end{bmatrix}}_{Q_1^T A} \xrightarrow{\cdot Q_1} \underbrace{\begin{bmatrix} x & x & 0 & 0 & 0 \\ x & x & x & x & x \\ 0 & x & x & x & x \\ 0 & x & x & x & x \\ 0 & x & x & x & x \end{bmatrix}}_{Q_1^T A Q_1} \tag{2.19}$$

Dieses Vorgehen wird wiederholt, um in den nachfolgenden Spalten Nullen zu erzeugen. So lässt der zweite Householder-Reflektor Q_2 die erste Spalte und Zeile unverändert, und erzeugt Nullen in der zweiten Spalte ab dem vierten Element.

$$\underbrace{\begin{bmatrix} x & x & 0 & 0 & 0 \\ x & x & x & x & x \\ & x & x & x & x \\ & x & x & x & x \\ & x & x & x & x \end{bmatrix}}_{Q_1^T A Q_1} \xrightarrow{Q_2^T \cdot} \underbrace{\begin{bmatrix} x & x & 0 & 0 & 0 \\ x & x & x & x & x \\ & x & x & x & x \\ & 0 & x & x & x \\ & 0 & x & x & x \end{bmatrix}}_{Q_2^T Q_1^T A Q_1} \xrightarrow{\cdot Q_2} \underbrace{\begin{bmatrix} x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 \\ & x & x & x & x \\ & & x & x & x \\ & & x & x & x \end{bmatrix}}_{Q_2^T Q_1^T A Q_1 Q_2} \tag{2.20}$$

Nach $m-2$ Wiederholungen erhalten wir wie gewünscht ein Produkt in Tridiagonalform:

$$\begin{bmatrix} x & x & & & \\ x & x & x & & \\ & x & x & x & \\ & & x & x & x \\ & & & x & x \end{bmatrix} \tag{2.21}$$

$$\underbrace{Q_{m-2}^T \cdots Q_2^T Q_1^T}_{Q^T} A \underbrace{Q_1 Q_2 \cdots Q_{m-2}}_{Q} = T \tag{2.22}$$

Der zugehörige Algorithmus sieht wie folgt aus:

for k	$= 1, 2, ..., m-2$
	$x = A_{k+1:m,k}$
	$v_k = sign(x_1)\|x\|_2 e_1 + x$
	$v_k = v_k / \|v_k\|_2$
	$A_{k+1:m,k:m} = A_{k+1:m,k:m} - 2v_k(v_k^* A_{k+1:m,k:m})$
	$A_{1:m,k+1:m} = A_{1:m,k+1:m} - 2(A_{1:m,k+1:m} v_k)v_k^*$

Abbildung 1: Algorithmus zur Tridiagonalisierung

Dabei wird die Matrix $Q = \prod_{k=1}^{m-2} Q_k$ nie explizit gebildet.

2.2.2 Vektoriteration, Rayleigh-Quotient, Inverse Iteration

Die folgenden Ideen betreffen meist Phase 2. Dies bedeutet, dass A nicht nur reell und symmetrisch, sondern auch tridiagonal ist.

Rayleigh-Quotient: Der Rayleigh-Quotient eines Vektors $x \in \mathbb{R}^m$ zur Matrix A ist der Skalar

$$r(x) = \frac{x^T Ax}{x^T x}. \tag{2.23}$$

Falls $x = q_i$, Eigenvektor zum Eigenwert λ_i, so lässt sich die Gleichung (2.23) wie folgt umschreiben:

$$r(q_i) \quad = \quad \frac{q_i^T A q_i}{q_i^T q_i} = \frac{q_i^T \lambda_i q_i}{q_i^T q_i} = \lambda_i \frac{q_i^T q_i}{q_i^T q_i} = \lambda_i \tag{2.24}$$

Somit ist $r(q_i) = \lambda_i$ der zum Eigenvektor q_i gehörige Eigenwert λ_i.
Falls ein gegebener Vektor x nah genug an einem Eigenvektor q liegt, so ist der Skalar gesucht, welcher der zum angenäherten Eigenvektor x gehörende Eigenwert ist. Dies lässt sich mittels der Methode der kleinsten Fehlerquadrate bestimmen: Da $Ax \neq \alpha x$, gilt $Ax - \alpha x \neq 0$. $||Ax - \alpha x||_2$ muss bezüglich α minimiert werden. Die Form $x\alpha = Ax$ lässt sich umschreiben mit $\alpha x^T x = x^T Ax$, daraus ergibt sich $\alpha = \frac{x^T Ax}{x^T x}$. Dies ist der Rayleigh-Quotient. So gilt $\alpha = r(x)$, und das besagt, dass der Rayleigh-Quotient als eine Schätzung für den Eigenwert λ betrachtet werden kann, falls x nah genug bei einem Eigenvektor liegt.

Vektoriteration: Die Matrix A sei wiederum symmetrisch und daher diagonalisierbar, so dass eine Orthonormalbasis von Eigenvektoren $q_1, ..., q_n$ existiert. Sei $x^{(0)}$ ein Vektor mit $\|x^{(0)}\| = 1$. Jedes $x \in \mathbb{R}^n$ kann umgeschrieben werden als

$$x = \sum_i c_i q_i, \tag{2.25}$$

dann gilt:

$$x^{(1)} = Ax^{(0)} = \sum_i c_i A q_i = \sum_i c_i \lambda_i q_i. \tag{2.26}$$

Mit

$$A^k q_i = A^{(k-1)} A q_i = A^{(k-1)} \lambda_i q_i = \lambda_i^k q_i \tag{2.27}$$

lassen sich folgende Umformungen machen:

$$x^{(k)} = A^k x^{(0)} = \sum_i c_i A^k q_i = \sum_i c_i \lambda_i^k q_i = \lambda_1^k \sum_i c_i \left(\frac{\lambda_i}{\lambda_1}\right)^k q_i \tag{2.28}$$

Der Term $\left(\frac{\lambda_i}{\lambda_1}\right)^k$ geht für $k \to \infty$ gegen Null für $i \neq 1$, wenn $|\lambda_1| > |\lambda_i| \forall i$. Daraus folgt für $x^{(k)}$:

$$x^{(k)} \approx c_1 \lambda_1^k q_1, \tag{2.29}$$

für k hinreichend groß. Da immer wieder mit A multipliziert wird, bleibt im Wesentlichen noch ein Vielfaches von q_1 übrig. Ist zusätzlich $|\lambda_1| > 1$, so wird $||A^k x||_2$ immer größer. Um damit verbundene numerische Probleme zu vermeiden, wird der neu berechnete Vektor nach jeder Multiplikation mit A normiert. Dadurch konvergiert x_k, falls $\lambda_1 > 0$, gegen ein Vielfaches von q_1, dem Eigenvektor zum betragsmäßig größten Eigenwert. Für $\lambda_1 < 0$ konvergiert $(-1)^k x_k$ und nicht x^k gegen q_1. Die Richtung des Eigenvektors bleibt immer erhalten.
Der Algorithmus der Vektoriteration, auch 'Von Mises-Potenzmethode' [5] genannt, ist in Abbildung 2 aufgezeigt. $\lambda^{(k)}$ ist eine Näherung des betragsgrößten Eigenwerts von

$x^{(0)}$	ein Vektor mit $\|x^{(0)}\| = 1$	
for k	$= 1, 2, ...$	
	$w = Ax^{(k-1)}$	A auf den Vektor anwenden
	$x^{(k)} = w/\|w\|$	normieren
	$\lambda^{(k)} = (x^{(k)})^T A x^{(k)}$	Rayleigh-Quotient einsetzen

Abbildung 2: Algorithmus der Vektoriteration

A, w ist bei jedem Schritt die Approximation an einen zugehörigen Eigenvektor.
Entsprechend gilt:

$$\lambda^{(k)} = (x^{(k)})^T \cdot A x^{(k)} \tag{2.30}$$

Bemerkung: Falls nur der betragsgrößte Eigenwert gesucht ist, und dieser Eigenwert weiten Abstand zum betragsmäßig zweitgrößten Eigenwert hat, ist die Vektoriteration sehr vorteilhaft. Der Nachteil ist entsprechend, dass die Konvergenz durch den Quotienten der beiden betragsgrößten Eigenwerte bestimmt wird. Vektoriteration kann nur den zum betragsgrößten Eigenwert passenden Eigenvektor finden. Die Konvergenz des Eigenwertes ist linear, der Fehler wird bei jedem Iterationsschritt nur durch einen konstanten Faktor $\approx |\lambda_2/\lambda_1|$ reduziert.

Inverse Iteration: Für jedes $\mu \in \mathbb{R}$, wobei μ kein Eigenwert der Matrix A ist, hat $(A - \mu I)^{-1}$ die selben Eigenvektoren wie A, und die zugehörigen Eigenwerte sind $(\lambda_j - \mu)^{-1}$, wobei λ_j die Eigenwerte von A sind. Sei μ sehr nah an einem Eigenwert λ_J der Matrix A. Dann ist $|(\lambda_J - \mu)|^{-1}$ viel größer als $|(\lambda_j - \mu)|^{-1}$ für alle $j \neq J$. Wenn wir die Vektoriteration auf $(A - \mu I)^{-1}$ anwenden, konvergiert dieser Prozess schnell gegen den Eigenvektor q_J. Diese Idee wird Inverse Iteration genannt. Der Algorithmus der Inversen Iteration sieht wie folgt aus:

$v^{(0)}$	ein Vektor mit $\|v^{(0)}\| = 1$	
for k	$= 1, 2, ...$	
	löse $(A - \mu I)w = v^{(k-1)}$ für w	$(A - \mu I)^{-1}$ anwenden
	$v^{(k)} = w/\|w\|$	normieren
	$\lambda^{(k)} = (v^{(k)})^T A v^{(k)}$	Rayleigh-Quotient

Abbildung 3: Algorithmus der Inversen Iteration

Sei A eine symmetrische relle Matrix, q_i die Menge der Eigenvektoren und λ_i die Menge der Eigenwerte. So ergibt sich für $\mu \neq \lambda_i$:

$$\begin{aligned} v^{(0)} &= \sum_i c_i q_i \\ v^{(1)} = (A - \mu I)^{-1} v^{(0)} &= \sum_i c_i q_i (\lambda_i - \mu)^{-1} \\ v^{(k)} = (A - \mu I)^{-k} v^{(0)} &= \sum_i c_i q_i (\lambda_i - \mu)^{-k} \end{aligned} \tag{2.31}$$

Konvergenz: Wird μ in der Nähe eines Eigenwertes gewählt, konvergiert $\lambda^{(k)}$ gegen diesen Eigenwert. Bei ungeeigneter Wahl von μ divergiert das Verfahren.
Ebenso wie die Vektoriteration konvergiert die Inverse Iteration auch nur linear. Bei der Inversen Iteration können wir allerdings den gesuchten Eigenvektor durch einen Schätzwert μ des zugehörigen Eigenwerts auswählen. Die lineare Konvergenz hängt von der Qualität dieses Wertes μ ab, somit ist sie kontrollierbar.

Rayleigh-Quotienten-Iteration: Bisher wurde eine Methode vorgestellt, um einen Eigenwert-Schätzwert aus einem geschätzten Eigenvektor zu erlangen (Rayleigh-Quotient), und eine andere Methode, um einen Eigenvektor-Schätzwert aus einem geschätzten Eigenwert zu erlangen (Inverse Iteration). Diese beiden Methoden werden kombiniert:

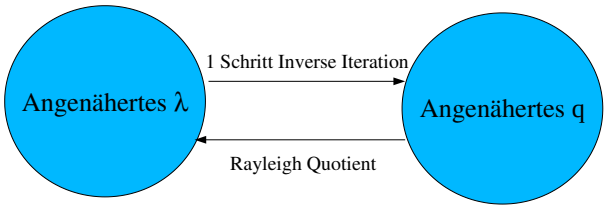

Abbildung 4: Zusammenhang von Inverser Iteration und Rayleigh-Quotienten-Iteration

Die Darstellung in Abbildung 4 ist sehr vereinfacht, denn um von einem angenäherten λ_J zu einem angenäherten q_J zu kommen, braucht man nicht nur einen Schritt der Inversen Iteration, sondern auch eine vorausgehende Annäherung an q_J.
Die Idee ist nun, kontinuierlich sich bessernde Eigenwert-Schätzwerte zu nutzen, um die Konvergenz der Inversen Iteration bei jedem Schritt zu erhöhen.
Diesen Algorithmus nennt man Rayleigh-Quotienten-Iteration:

Konvergenz: Die Konvergenz dieses Algorithmus' ist kubisch [6].

$v^{(0)}$	ein Vektor mit $\|v^{(0)}\| = 1$	
$\lambda^{(0)} = (v^{(0)})^T A v^{(0)}$	entsprechender Rayleigh-Quotient	
for k = 1, 2, ...		
	löse $(A - \lambda^{(k-1)} I)w = v^{(k-1)}$ für w	$(A - \lambda^{(k-1)} I)^{-1}$ anwenden
	$v^{(k)} = w/\|w\|$	normieren
	$\lambda^{(k)} = (v^{(k)})^T A v^{(k)}$	Rayleigh-Quotient

Abbildung 5: Algorithmus der Rayleigh-Quotienten-Iteration

Aufwand: Betrachtet man die drei beschriebenen Iterationsverfahren, so lässt sich folgendes zum Arbeitsaufwand pro Iterationsschritt bemerken: Ist $A \in \mathbb{R}^{m \times m}$ eine vollbesetzte Matrix, so beinhaltet jeder Schritt der Vektoriteration eine Matrix-Vektor-Multiplikation, welche $O(m^2)$ *Flops* [1] benötigt. Jeder Schritt der Inversen Iteration beinhaltet die Lösung eines linearen Gleichungssystems, das sind im ersten Schritt $O(m^3)$ Flops, in jedem weiteren Schritt nur noch $O(m^2)$ Flops, wenn die Matrix vorher durch LU- oder QR-Faktorisierung umgeformt wird. Bei der Rayleigh-Quotienten-Iteration ändert sich die zu invertierende Matrix bei jedem Schritt, daher kann nicht auf vorherige Faktorisierung zurückgegriffen werden.
Der Aufwand bessert sich stark, wenn A eine tridiagonale Matrix ist, da dann alle drei Iterationen nur noch $O(m)$ Flops pro Schritt benötigen.

2.2.3 QR-Algorithmus

Das QR-Verfahren wird in der Praxis am häufigsten zur Berechnung aller Eigenwerte einer quadratischen Matrix eingesetzt. Es beinhaltet die globalen Konvergenzeigenschaften der Vektoriteration und die schnelle lokale Konvergenz der Rayleigh-Quotienten-Iteration.

QR-Algorithmus (ohne Shift): Das Verfahren lässt sich sehr einfach formulieren: Die Matrix A sei reell und symmetrisch. Damit besitzt sie nur relle Eigenwerte $\lambda_1, ..., \lambda_n \in \mathbb{R}$ und es existiert eine Orthonormalbasis der Eigenvektoren $q_1, ..., q_n$.
Mit Givens- oder Householder-Transformationen kann die reguläre Matrix A zerlegt werden in $A = QR$. Q ist eine orthonormale Matrix und R eine obere Dreiecksmatrix. Da $Q^T = Q^{-1}$ ist, gilt

$$RQ = Q^{-1} A Q, \tag{2.32}$$

dies bedeutet, dass RQ und $A = QR$ die selben Eigenwerte haben.
Daraus ergibt sich der QR-Algorithums ohne Shift wie in Abbildung 6 beschrieben.
Die QR-Faktorisierung wird immer wieder mit den berechneten Faktoren Q und R in umgekehrter Reihenfolge RQ multipliziert. Im Verlauf der Iteration sollen die Matrizen $A^{(k)}$ allmählich obere Dreiecksmatrizen werden, von deren Diagonalen schließlich die Eigenwerte λ_i, $i = 1, ..., n$, von A abgelesen werden können.
Vergleicht man speziell die erste Spalte der Matrix aus der Gleichung

$$A^{(k+1)} = Q^{(k)} R^{(k)} = \hat{Q}_k \hat{R}_k \tag{2.33}$$

[1] Flops (Floating Point Operations) = Additionen, Subtraktionen, Multiplikationen, Divisionen, Wurzelziehen

$A^{(0)}$	$= A$	
for k	$= 1, 2, ...$	
	$Q^{(k)}R^{(k)} = A^{(k)}$	QR-Faktorisierung von $A^{(k-1)}$
	$A^{(k+1)} = R^{(k)}Q^{(k)}$	Faktoren in umgekehrter Reihenfolge rekombinieren

Abbildung 6: QR-Algorithmus ohne Shift

so ergibt sich

$$A^{(k+1)}e_1 = \hat{Q}_k \hat{r}_{11}^{(k)} e_1 = \hat{r}_{11}^{(k)} \hat{q}_1^{(k)}, \tag{2.34}$$

wobei e_1 der erste Einheitsvektor ist, $\hat{r}_{11}$ das (1,1)-Element von $\hat{R}_k$ ist und $\hat{q}_1^{(k)}$ die erste Spalte von $\hat{Q}_k$ ist. Man kann nun erwarten, laut Vektoriteration, dass $\hat{q}_1^{(k)}$ für hinreichend große k eine gute Näherung an einen Eigenvektor zum dominanten Eigenwert λ_1 von A ist.
Da $A^{(k+1)} = \hat{Q}_k^T A_k \hat{Q}_k$, gilt:

$$A^{(k+1)}e_1 = \hat{Q}_k^T A \hat{q}_1^{(k)} \approx \lambda_1 \hat{Q}_k^T \hat{q}_1^{(k)} = \lambda_1 e_1 \tag{2.35}$$

und somit bekommt $A^{(k)}$ die Form:

$$A^{(k+1)} \approx \left(\begin{array}{c|c} \lambda_1 & \cdots \\ 0 & \\ \vdots & \\ 0 & \cdots \end{array}\right) \tag{2.36}$$

Wegen der Orthogonalität der Matrix $\hat{Q}_k$ folgt, falls die Matrix A invertierbar ist:

$$\hat{Q}_k^T = \hat{R}_k A^{-(k+1)}.$$

Eine Multiplikation von links mit e_n^T ergibt

$$\hat{q}_n^{(k)T} = e_n^T \hat{Q}_k^T = e_n^T \hat{R}_k A^{-(k+1)} = \hat{r}_{nn}^{(k)} e_n^T A^{-(k+1)}. \tag{2.37}$$

Der Vektor $\hat{q}_n^{(k)}$, die letzte Spalte von $\hat{Q}_k$, ist somit das Ergebnis von k+1 Schritten der Inversen Iteration mit A^T. Er bildet eine Näherung für einen linken Eigenvektor zu dem betragskleinsten Eigenwert λ_n von A. Falls dieser $\neq 0$ ist, folgt

$$e_n^T A^{(k+1)} = e_n^T Q_k^T A Q_k = {q_n^{(k)}}^T A Q_k \approx \lambda_n {q_n^{(k)}}^T Q_k = \lambda_n e_n^T \tag{2.38}$$

So zeigt sich, dass die letzte Zeile von A_{k+1} näherungsweise ein Vielfaches von e_n^T ist. So erhalten die Matrizen A_k für $k \to \infty$ die gewünschte Form einer rechten oberen Dreiecksmatrix:

$$A^{(k)} \xrightarrow{k\to\infty} \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix} \tag{2.39}$$

QR-Algorithmus mit Shift: Der QR-Algorithmus mit Shift ermöglicht ein besseres Konvergenzverhalten. Dazu führt man eine Folge reeller Zahlen $\mu_{k\geq 0}$ als sogenannte Shifts ein. Dann kann im k-ten Iterationsschritt mit Hilfe der QR-Zerlegung folgende Berechnung angestellt werden:

$$A_k - \mu_k I = Q_k R_k \quad \Rightarrow \quad A_{k+1} = R_k Q_k + \mu_k I \tag{2.40}$$

Um die Konvergenz der Inversen Iteration zum kleinsten Eigenwert λ_n von A in (2.37) zu beschleunigen, und die lokal schnelle Konvergenz der Rayleigh-Quotienten-Iteration auszunutzen, bildet man den Rayleigh-Quotienten $\mu_k = e_n^T A_k e_n$ (das rechte untere Eckelement von A_k) und führt einen Schritt der Inversen Iteration bezüglich des linken Eigenvektors aus.
Wegen (2.40) ergibt sich

$$e_n^T (A_k - \mu_k I)^{-1} = e_n^T R_k^{-1} Q_k^T = \frac{1}{r_{nn}^{(k)}} e_n^T Q_k^T = \frac{1}{r_{nn}^{(k)}} {q_n^{(k)}}^T, \tag{2.41}$$

wobei $r_{nn}^{(k)}$ das rechte untere Eckelement von R_k und $q_n^{(k)}$ die letzte Spalte von Q_k darstellt. R_k^{-1} ist wieder eine obere Dreiecksmatrix, deren Diagonaleinträge die Kehrwerte der entsprechenden Diagonaleinträge von R_k sind.
Ein Schritt der Rayleigh-Quotienten-Iteration liefert die letzte Spalte von Q_k als neue Näherung an den linken Eigenvektor von A_k zu λ_n. Das rechte untere Eckelement von A_{k+1} ist jedoch der zugehörige Rayleigh-Quotient, also der nächste Shift μ_{k+1}:

$$\mu_{k+1} = {q_n^{(k)}}^T A_k q_n^{(k)} = e_n^T Q_k^T A_k Q_k e_n = e_n^T A_{k+1} e_n \tag{2.42}$$

Der QR-Algorithmus, der eben genannte Modifikationen beinhaltet, ist wie folgt gegliedert:

for k	$= 1, 2, ...$	
	wähle $\mu^{(k)}$	z.B. $\mu^{(k)} = A_{mm}^{(k-1)}$
	$Q^{(k)} R^{(k)} = A^{(k-1)} - \mu^{(k)} I$	QR-Faktorisierung von $A^{(k-1)} - \mu^{(k)} I$
	$A^{(k)} = R^{(k)} Q^{(k)} + \mu^{(k)} I$	Faktoren in umgekehrter Reihenfolge rekombinieren

Abbildung 7: QR-Algorithmus mit Shift

Deflation: Eben wurde als Shift μ_k das (n,n)-Element ${a_{nn}}^{(k)}$ von $A^{(k)}$ gewählt. Daneben gibt es noch die Strategie der Deflation. Das (n,n)-Element von $A^{(k)}$ konvergiert sehr schnell gegen den Eigenwert, und das (n,n-1)-Element nähert sich der Null auf Maschinengenauigkeit ϵ, so dass gilt:

$$A^{(k)} \xrightarrow{k\to\infty} \left[\begin{array}{cccc|c} * & \dots & \dots & * & * \\ * & \ddots & & & \vdots \\ & \ddots & \ddots & \vdots & \vdots \\ 0 & & * & * & * \\ \hline 0 & \dots & \dots & \epsilon & \lambda_n \end{array}\right] \approx \left[\begin{array}{ccc|c} & B_{k-1} & & * \\ & & & \vdots \\ & & & * \\ \hline 0 & \dots & 0 & \lambda_n \end{array}\right] \tag{2.43}$$

Nun reicht es aus, das kleinere Teilproblem mit der Hessenberg-Matrix B_{k-1} zu betrachten. Diese Reduktion nennt man Deflation. Ebenso zerfällt das Problem in Teilprobleme,

sobald ein Nebendiagonalelement $a_{j,j+1}^{(k)}$, $j = 1, ..., n-2$, Null wird oder ausreichend nah an Null im Bereich der Maschinengenauigkeit liegt.

Konvergenz: Wählt man als Shift μ_k jeweils das (n,n)-Element von $A^{(k)}$, dann darf man bei der Rayleigh-Quotienten-Iteration sehr schnelle, kubische, Konvergenz dieser Eckelemente gegen den kleinsten Eigenwert λ_n von A erwarten.

2.2.4 Bisektion

Da die Eigenwerte einer reellen symmetrischen Matrix auch reell sind, können wir sie durch Berechnung der Nullstellen des charakteristischen Polynoms $p(x) = \det(A - xI)$ finden. Die Idee dabei ist, das Polynom $p(x)$ an einer bestimmten Stelle x auszuwerten, ohne die Koeffizienten zu betrachten, und das gewöhnliche Bisektion-Verfahren für nicht-lineare Funktionen anzuwenden. Der daraus resultierende Algorithmus ist sehr stabil. Sein besonderer Reiz wird durch weitere Eigenschaften der Eigenwerte und Determinanten ausgelöst:
Sei $A \in \mathbb{R}^{m \times m}$ eine symmetrische Matrix, wobei $A^{(1)}, ..., A^{(m)}$ die quadratischen Untermatrizen mit den Dimensionen $1, ..., m$ bezeichnen. $A^{(k)}$ hat folgende Form:

$$A^{(k)} = \begin{pmatrix} a_{11} & \dots & a_{1k} \\ \vdots & \dots & \vdots \\ a_{k1} & \dots & a_{kk} \end{pmatrix} \tag{2.44}$$

Die Matrix A ist tridiagonal und nicht reduzierbar:

$$A = \begin{pmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & b_2 & a_3 & \ddots & \\ & & \ddots & \ddots & b_{m-1} \\ & & & b_{m-1} & a_m \end{pmatrix}, \qquad b_j \neq 0. \tag{2.45}$$

Die Eigenwerte von $A^{(k)}$ bezeichnen wir mit

$$\lambda_1{}^{(k)} < \lambda_2{}^{(k)} < ... < \lambda_k{}^{(k)},$$

und $\det(A) = \Pi\lambda_i$. Die entscheidende Eigenschaft ist, dass diese Eigenwerte immer gemäß folgender Ungleichung verschachtelt sind:

$$\lambda_j{}^{(k+1)} < \lambda_j{}^{(k)} < \lambda_{j+1}{}^{(k+1)} \tag{2.46}$$

für $k = 1, 2, ..., m-1$ und $j = 1, 2, ..., k-1$.
Durch diese Verschachtelungs-Eigenschaft ist es möglich, die exakte Anzahl der Eigenwerte einer Matrix in einem bestimmten Intervall zu bestimmen.
Sturm'sche Kette

Definition 1 *Die Folge $p(x) = p_0(x), p_1(x), ..., p_m(x)$ reeller Polynome heißt Sturmsche Kette, falls gilt:*
1. $p_o(x)$ besitzt nur einfache Nullstellen.
2. sign $p_1(\psi) = -sign\ p_0{}'(\psi)$ für alle reellen Nullstellen ψ von $p_0(x)$.
3. Für $i = 1, 2, ..., m-1$ gilt: $p_{i+1}(\psi)p_{i-1}(\psi) < 0$, falls ψ reelle Nullstelle von $p(x)$ ist.
4.Das letzte Polynom $p_m(x)$ ändert sein Vorzeichen nicht.

Dann gilt der Satz:

Satz 1 *Die Anzahl der rellen Nullstellen von $p(x) = p_0(x)$, die kleiner sind als a, ist gegeben durch $w(a)$. Die Anzahl im Intervall $a \leq x < b$ ist gleich $w(b) - w(a)$.*

$w(a)$ gibt die Anzahl der Vorzeichenwechsel an.
Es gilt: Für jede nicht zerfallende symmetrische tridiagonale Matrix $A \in \mathbb{R}^{m \times m}$ bilden die Polynome $p_0(\lambda) = p(\lambda) = det(A - \lambda I) = det(A^m - \lambda I), p_1(\lambda) = det(A^{m-1} - \lambda I), ..., p_m(\lambda) = 1$, eine Sturm'sche Kette. Damit kann die Anzahl der Eigenwerte innerhalb eines beliebigen Intervalls $[a, b)$ bestimmt werden.

Aufwand: Die Kosten sind $\mathcal{O}(m)$ Flops für jede Auswertung einer Sequenz. In Folge dessen werden $\mathcal{O}(m \log(\epsilon_{machine}))$ Flops insgesamt benötigt, um einen Eigenwert relativ exakt zu $\epsilon_{machine}$ zu finden. Falls nur wenige Eigenwerte gesucht sind, ist dies eine deutliche Verbesserung zu den $\mathcal{O}(m^2)$ Operationen des QR-Algorithmus'.

Anwendung: Die Methode der Bisektion hat einen großen praktischen Wert. Nachdem eine Matrix auf Tridiagonalform gebracht wurde, wird standardmäßig Bisektion angewendet, sofern nur eine Auswahl der Eigenwerte berechnet werden soll. Bisektion kann alle Eigenwerte innerhalb eines gewählten Intervalls finden. Dabei kann das Intervall entweder definiert werden mit $\lambda_i \in [u, o]$ oder durch die Angabe, es solle der u-te bis o-te Eigenwert bei aufsteigender Sortierung berechnet werden. Sobald die gewünschten Eigenwerte gefunden sind, können die zugehörigen Eigenvektoren durch inverse Iteration erhalten werden.

2.2.5 Divide-and-Conquer

Der Divide-and-Conquer-Algorithmus basiert auf einer rekursiven Unterteilung eines symmetrischen tridiagonalen Eigenwertproblems in Probleme kleinerer Dimension.
Sei $T \in \mathbb{R}^{m \times m}$, mit $m \geq 2$, eine symmetrische tridiagonale und nicht reduzierbare Matrix. Dann kann T für jedes n, mit $1 \leq n \leq m$, in Untermatrizen aufgeteilt werden:

$$\mathrm{T} = \begin{bmatrix} T_1 & \beta \\ \beta & T_2 \end{bmatrix} = \begin{bmatrix} \hat{T}_1 & \\ & \hat{T}_2 \end{bmatrix} + \begin{bmatrix} & \\ \beta & \beta \\ \beta & \beta \end{bmatrix} \tag{2.47}$$

Hier ist T_1 die obere linke $n \times n$ Untermatrix auf der Hauptdiagonalen von T, und T_2 ist die untere rechte $(m - n) \times (m - n)$ Untermatrix auf der Hauptachse. Dabei ist $\beta = t_{n+1,n} = t_{n,n+1} \neq 0$. Der einzige Unterschied zwischen T_1 und $\hat{T}_1$ besteht in dem unteren rechten Eintrag t_{nn}, welcher durch $t_{nn} - \beta$ ersetzt wurde. Der Unterschied zwischen T_2 und $\hat{T}_2$ besteht in dem oberen linken Eintrag $t_{n+1,n+1}$, welcher durch $t_{n+1,n+1} - \beta$ ersetzt wurde. Durch diese beiden Modifikationen erhält die rechte Matrix in (2.47) den Rang 1.
Man kann nun sagen: Eine tridiagonale Matrix kann als Summe einer 2×2 blockdiagonalen Matrix mit tridiagonalen Blöcken und einer einrangigen Korrektur geschrieben werden.
Der Divide-and-Conquer-Algorithmus arbeitet wie folgt: Die Matrix T wird, wie in

(2.47) beschrieben, aufgeteilt mit $n \approx m/2$. Dabei seien die Eigenwerte von $\hat{T}_1$ und $\hat{T}_2$ bekannt. Da die Korrektur-Matrix den Rang 1 hat, gelangt man mittels einer nichtlinearen Berechnung von den Eigenwerten von $\hat{T}_1$ und $\hat{T}_2$ zu den Eigenwerten von T selbst. Ebenso lassen sich die Eigenwerte von $\hat{T}_1$ und $\hat{T}_2$ durch weitere Unterteilungen mit einrangigen Korrekturen finden, und so weiter. So wird also ein $m \times m$ Eigenwertproblem auf eine Menge von 1×1 Eigenwertproblemen zusammen mit einrangigen Korrekturen reduziert. In der Praxis ist es dabei üblich, um eine größere Effizienz zu erzielen, auf den QR-Algorithmus zu wechseln, sobald die Untermatrizen genügend kleine Dimensionen haben, anstatt alle Rekursionen bis zum Ende auszuführen.
Der mathematische Kernpunkt in diesem Ablauf liegt in der Frage, wie man von den bekannten Eigenwerten der Matrizen $\hat{T}_1$ und $\hat{T}_2$ zu den Eigenwerten der Matrix T gelangt. Dazu seien folgende Diagonalisierungen vorgenommen:

$$\hat{T}_1 = Q_1 D_1 {Q_1}^T, \qquad \hat{T}_2 = Q_2 D_2 {Q_2}^T$$

Dann folgt aus der in (2.47) beschriebenen Aufteilung:

$$T = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \left(\begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix} + \beta z z^T \right) \begin{bmatrix} {Q_1}^T & \\ & {Q_2}^T \end{bmatrix} \tag{2.48}$$

Dabei ist $z^T = ({q_1}^T, {q_2}^T)$, wobei ${q_1}^T$ die letzte Zeile der Matrix Q_1 ist und ${q_2}^T$ die erste Zeile von Q_2 ist.
So haben wir das mathematische Problem auf das Problem reduziert, die Eigenwerte einer diagonalen Matrix plus einer einrangigen Korrekturmatrix zu finden. Um zu zeigen, wie dies funktioniert, vereinfachen wir die Notation wie folgt:
Angenommen, wir möchten die Eigenwerte von $D + ww^T$ finden, wobei $D \in \mathbb{R}^{m \times m}$ eine diagonale Matrix mit unterschiedlichen Diagonalelementeinträgen d_j ist, und $w \in \mathbb{R}^m$ ein Vektor ist. Wir setzen voraus, dass $w_j \neq 0$ für alle j, denn nur dann ist die Tridiagonalmatrix nicht zerfallend. Dann sind die Eigenwerte von $D + ww^T$ die Nullstellen der gebrochen-rationalen Funktion

$$f(\lambda) = 1 + \sum_{j=1}^{m} \frac{{w_j}^2}{d_j - \lambda} \tag{2.49}$$

Wenn $(D + ww^T)q = \lambda q$ für ein $q \neq 0$, dann ist $(D - \lambda I)q + w(w^T q) = 0$. Das bedeutet $q + (D - \lambda I)^{-1} w(w^T q) = 0$. Wenn dies gilt, gilt auch $w^T q + w^T (D - \lambda I)^{-1} w(w^T q) = 0$. Dies führt zu der Gleichung $f(\lambda)(w^T q) = 0$, wobei $w^T q$ ungleich Null sein muss, da sonst, wegen $(D + ww^T)q = Dq + ww^T q = \lambda q$, q Eigenvektor von D wäre.

Säkulare Gleichung: Zusammengefasst kann man sagen, dass, falls q ein Eigenvektor von $D + ww^T$ mit dem Eigenwert λ ist, $f(\lambda) = 0$ sein muss. Der umgekehrte Fall folgt daraus, da die Form von $f(\lambda)$ garantiert, dass sie genau m Nullstellen hat. Die Gleichung $f(\lambda) = 0$ ist bekannt als die *Säkulare Gleichung*.

Operationsanzahl: Bei jedem rekursiven Schritt des Divide-and-Conquer-Algorithmus' wird die Gleichung (2.49) mittels eines schnellen iterativen Prozesses (*Verfahren von Hebden* [5]), der mit Newton's Methode zusammenhängt, gelöst. Dazu werden nur $\mathcal{O}(1)$ Iterationen pro Lösung benötigt (oder $\mathcal{O}(log(|\, log(\epsilon_{machine})\,|))$ Iterationen, wenn $\epsilon_{machine}$ als Variable betrachtet wird). Die Lösung dieser Gleichung benötigt also $\mathcal{O}(m)$ Flops für eine $m \times m$ Matrix, oder $\mathcal{O}(m^2)$ Flops insgesamt. Bei einer Rekursion, die

eine m-dimensionale Matrix bei jedem Schritt genau in der Hälfte teilt, wird die Gesamtanzahl der Operationen, die benötigt werden, um die Eigenwerte einer tridiagonalen Matrix mittels des Divide-and-Conquer-Algorithmus' zu bestimmen, zu:

$$\mathcal{O}(m^2 + 2(\frac{m}{2})^2 + 4(\frac{m}{4})^2 + 8(\frac{m}{8})^2 + ... + m(\frac{m}{m})^2) \tag{2.50}$$

Dies bezeichnet die Ordnung der Summe:

$$\sum_{i=0}^{m} \frac{1}{2^i} \cdot m^2 = m^2 \cdot \sum_{i=0}^{m} \left(\frac{1}{2}\right)^i = 2 \tag{2.51}$$

Diese harmonische Reihe konvergiert für $m \to \infty$ gegen 2, und daher beträgt der Aufwand $\mathcal{O}(m^2)$ (nicht $\mathcal{O}(m^2\ log\ m)$). Die Operationenanzahl der Divide-and-Conquer-Methode scheint gleicher Größenordnung zu sein wie bei dem QR-Algorithmus.

Nutzen: Der Divide-and-Conquer-Algorithmus ist mehr als doppelt so schnell wie der QR-Algorithmus, wenn sowohl Eigenwerte als auch Eigenvektoren gesucht sind [7]. Er repräsentiert den wichtigsten Fortschritt bei Matrix-Eigenwert-Algorithmen seit 1960, erstmals eingeführt von Cuppen im Jahre 1981.

2.2.6 Relativ robuste Repräsentation (RRR)

Mittels des RRR-Algorithmus'[8] lassen sich die Eigenwerte und Eigenvektoren einer reellen symmetrischen tridiagonalen Matrix bestimmen. Diese Methode ist eine Variation der Inversen Iteration und kann in den meisten Fällen die benötigte Zeit, um orthogonale Eigenvektoren zu bestimmen, wesentlich verringern.
Die schnellste Methode, um an Eigenwerte und Eigenvektoren zu kommen, ist mittels Bisektion (für Eigenwerte) und Inverser Iteration (für Eigenvektoren) zu erreichen. Dabei ist die Zeit, die zur Eigenwertbestimmung nötig ist recht klein im Vergleich zu der Zeit, die benötigt wird, um orthogonale Eigenvektoren zu bestimmen. Ein altes Problem bei der Inversen Iteration ist das Auftreten von Eigenwerten, die sehr nah zusammen liegen. Ebenso ist die geeignete Wahl des Startvektors wichtig. Wenn die Eigenwerte sehr nah zusammen liegen und der Startvektor für die Inverse Iteration ungünstig gewählt wurde, werden die daraus resultierenden Eigenvektoren nicht orthogonal sein und wichtige Komponenten werden die selbe Richtung haben.
Die Methode der RRR verwendet einen Algorithmus, den Berkeley-Algorithmus, zur Bestimmung von orthonormalen Eigenvektoren für reelle symmetrische tridiagonale Matrizen. Der Berkeley-Algorithmus reduziert die Menge der erforderlichen Orthonormalisierungen. Die wesentliche Idee liegt darin, die Tridiagonalmatrix T durch stabilere bidiagonale Faktoren, von denen die Eigenvektoren berechnet werden können, gemäß $T = LL^T$ zu ersetzen.
Um den Algorithmus beschreiben zu können, vorab zwei Definitionen:

Absoluter Abstand: Der absolute Abstand zwischen den zwei Eigenwerten λ_i und λ_{i+1} wird definiert durch:

$$absgap(\lambda_i, \lambda_{i+1}) = |\lambda_{i+1} - \lambda_i| \tag{2.52}$$

Relativer Abstand: Der relative Abstand zwischen den zwei Eigenwerten λ_i und λ_{i+1} wird definiert durch:

$$relgap(\lambda_i, \lambda_{i+1}) = \frac{|\lambda_{i+1} - \lambda_i|}{max(|\lambda_i|, |\lambda_{i+1}|)} \tag{2.53}$$

Algorithmus: Der Berkeley-Algorithmus besteht aus zwei Teilen. Algorithmus 1 berechnet die Eigenwerte von T, und Algorithmus 2 berechnet einen Eigenvektor z von LL^T bei einer gegebenen Eigenwertschätzung $\hat{\lambda}$. Die wesentliche Schritte werden in Abbildung 8 und Abbildung 9 zusammengefasst.

1. wähle μ so, dass $T + \mu I$ positiv definit ist

2. berechne die Cholesky-Zerlegung: $T + \mu I = LL^T$

3. berechne die Singulärwerte von L, gekennzeichnet durch: $0 \leq \hat{\sigma}_1 \leq \hat{\sigma}_2 \leq ... \leq \hat{\sigma}_n$ mit hoher relativer Genauigkeit (z.B. mit Bisektion)

4. gruppiere die Eigenwerte von LL^T, $\hat{\lambda}_i = \hat{\sigma}_i^2$ ein nach:

4.a) isoliert: $\hat{\lambda}_m$ ist isoliert wenn:
$min(relgap(\lambda_m, \lambda_{m+1}), relgap(\lambda_m, \lambda_{m-1})) > min(10^{-3}, 1/n)$

4.b) freie Cluster: $\hat{\lambda_m}, \hat{\lambda}_{m+1}, ..., \hat{\lambda}_{m+k}$ bilden ein freies Cluster, wenn:
$max(100, n)\epsilon \leq min(relgap(\lambda_i, \lambda_{i+1}), relgap(\lambda_i, \lambda_{i-1})) \leq min(10^{-3}, 1/n)$

4.c) dichte Cluster: $\hat{\lambda}_m, \hat{\lambda}_{m+1}, ..., \hat{\lambda}_{m+k}$ sind eng geclustert wenn es Eigenwerte λ_j, λ_{j+1} gibt mit $m \leq j < m + k$, so dass gilt:
$relgap(\lambda_j, \lambda_{j+1}) < max(100, n)\epsilon$
und
$min(relgap(\lambda_i, \lambda_{i+1}), relgap(\lambda_i, \lambda_{i-1})) \leq min(10^{-3}, 1/n)$

5. für alle isolierten und freien Cluster von Eigenwerten: entsprechende Eigenvektoren mit Algorithmus 2 berechnen

6. für dichte Cluster: Inverse Iteration mit Reorthogonalisierung in der inneren Schleife, um Eigenvektoren zu berechnen

Abbildung 8: Teil 1 des RRR-Algorithmus'

Relativ robuste Repräsentation: Eine relativ robuste Repräsentation ist ein Menge von Zahlen die eine Matrix A so definieren, dass kleine komponentenweise Störungen in diesen Zahlen zu einer kleinen relativen Änderung in den Eigenwerten führen. Die Änderung in den Eigenvektoren ist invers proportional zu den relativen Abständen in den Eigenwerten.

Als Beispiel: eine bidiagonale Matrix L und eine positive Diagonalmatrix D sind die tridiagonalen Faktoren der tridiagonalen Matrix LDL^T. Sie bilden eine relativ robuste Repräsentation.

Definition 1 *Eine Relativ Robuste Repräsentation (RRR) ist eine Menge von Zahlen x_j die eine Matrix A so definieren, dass, wenn x_j gestört ist zu $x_j(1 + \epsilon_j)$, für alle*

1. berechne $LL^T - \hat{\lambda}I = L_+D_+U_+ = U_-D_-L_-$

2. berechne $\phi_k = D_+(k) + D_-(k) - (LL^T)_{kk}$ für $k = 1, 2, ..., n$, und wähle Index r dort wo $|\phi_k|$ minimal ist

3. löse $(LL^T - \hat{\lambda}I)z = \phi_r e_r$

Abbildung 9: Teil 2 des RRR-Algorithmus'

$j = 1, 2, ..., n$, *gilt:*

$$\begin{aligned} \frac{\delta\lambda_j}{\lambda_j} &= O\left(\sum_i \epsilon_i\right), \\ |\sin L(v_j, v_j + \delta\lambda_j)| &= O\left(\frac{\sum_i \epsilon_i}{relgap(\lambda_j, \lambda_k | k \neq j)}\right). \end{aligned} \tag{2.54}$$

Hierbei ist *relgap* der relative Abstand nach Definition (2.53).
Normalerweise sind die RRRs tridiagonale Faktoren L und D der geshifteten Matrix $T - \mu I$. Dabei bezeichnen wir oft solch eine Faktorisierung als eine Repräsentation von T mit dem Shift μ. Ebenso wird häufig auf die zugrunde liegende Matrix LDL^T als RRR verwiesen (anstelle der einzelnen Faktoren L und D). Kann eine Repräsentation nur einige, aber nicht alle, Eigenwerte mit hoher Genauigkeit bestimmen, wird sie partielle RRR genannt und mit $RRR(j, j+1, ..., k)$ gekennzeichnet.

Vergleich der Algorithmen: (QR $\leftrightarrow$ Divide-and-Conquer) Die Reduktion einer vollbesetzten Matrix auf eine tridiagonale Form (Phase 1) verlangt $4m^3/3$ Flops. Jede Verbesserung in den $\mathcal{O}(m^2)$ Operationen für die Diagonalisierung der tridiagonalen Matrix ist sehr wichtig. Anders verhält es sich, wenn sowohl Eigenwerte als auch Eigenvektoren berechnet werden. Hier benötigt Phase 1 $8m^3/3$ Flops, und Phase 2 $\mathcal{O}(m^3)$ Flops. Beim QR-Algorithmus wäre die Operationsanzahl bei Phase 2 bei etwa $6m^3$. Der Divide-and-Conquer-Algorithmus reduziert dies, da sich die einzigen nicht linearen Iterationen auf die skalare Gleichung (2.49) beschränken, und nicht die orthogonalen Matrizen Q_j betreffen, wo hingegen der QR-Algorithmus bei jedem Iterationsschritt die Matrizen Q_j verändern muss.
Die Operationsanzahl bemisst sich dann wie folgt: Der Operationsteil des Divide-and-Conquer-Algorithmus' berechnet die Multiplikationen von Q_j und ${Q_j}^T$ in (2.48). Die Gesamtanzahl an Operationen, über alle Rekursionsschritte aufsummiert, beträgt $4m^3/3$ Flops, eine Verbesserung gegenüber $6m^3$ Flops um mehr als $4m^3$ Flops. Hinzu kommen die $8m^3/3$ Flops für Phase 1, welche eine Verbesserung von etwa $9m^3$ auf $4m^3$ bewirken. In der Praxis ist der Divide-and-Conquer-Algorithmus sogar noch besser. Für die meisten Matrizen A haben viele der Vektoren z und Matrizen Q_j aus der Gleichung (2.48) Werte, die kleiner als die Maschinengenauigkeit sind. Diese dünne Besetzung der Matrizen ermöglicht eine *Numerische Deflation*, wobei aufeinander folgende tridiagonale Eigenwertprobleme auf einzelne Probleme kleinerer Dimensionen reduziert werden. Dies reduziert im typischen Fall die Operationenanzahl von Phase 2 auf eine Ordnung von weniger als m^3 Flops, insgesamt wird dadurch die Operationenanzahl von Phase 1 und 2 auf $8m^3/3$ reduziert.

Sollen nur die Eigenwerte berechnet werden, gilt (2.50) als oberste Schranke und die Operationsanzahl bei Phase 2 wird auf eine Ordnung von weniger als m^2 Flops reduziert.
In der Praxis kann man nicht von einem einzigen Divide-and-Conquer-Algorithmus sprechen, da es viele Varianten dazu gibt. Kompliziertere einrangige Verbesserungen werden häufig für mehr Stabilität eingesetzt, und auch zweirangige Verbesserungen werden manchmal benutzt.

2.3 Serielle Eigenwertproblemlöser in LAPACK

LAPACK[3] (Linear Algebra PACKage) ist eine Programmbibliothek, die hocheffiziente Routinen für Probleme aus der Linearen Algebra wie lineare Gleichungssysteme, Eigenwertprobleme und Singulärwertberechnungen bereitstellt. Sie basiert auf der BLAS-Bibliothek (Basic Linear Algebra Subprograms), die Routinen für Matrix-Matrix- (BLAS3-Level), Matrix-Vektor- (BLAS2-Level) und Vektor-Vektor-Operationen (BLAS1-Level) beinhaltet.

2.3.1 Eigenwertproblemlöser-Treiber

Es gibt vier Arten von Treiber-Routinen in LAPACK, um symmetrische Eigenwertprobleme seriell zu lösen.

Erläuterung der Namensgebung: Die Namen der Routinen setzen sich wie folgt zusammen: D=double precision, SY=symmetric, EV=eigenvalue, worauf ein weiterer Buchstabe zur Kennzeichnung des Algorithmus' folgt.

- **Standard-Treiber:**
  Der Standard-Treiber, DSYEV, berechnet alle Eigenwerte und optional die Eigenvektoren mittels des geshifteten QR-Algorithmus'.

- **Experten-Treiber:**
  Der Experten-Treiber, DSYEVX, berechnet alle oder eine Auswahl der Eigenwerte mit Hilfe der Bisektion und optional die Eigenvektoren mit Inverser Iteration.
  Falls nur wenige Eigenwerte oder Eigenvektoren zu berechnen sind, ist der Experten-Treiber schneller als der Standard-Treiber. Falls alle Eigenwerte zu berechnen sind wird standardmäßig der QR-Algorithmus verwendet. Durch den Parameter $ABSTOL = 2 * SLAMCH('S')$ kann hier Bisektion erzwungen werden.

- **Divide-and-Conquer-Treiber:**
  Der Divide-and-Conquer-Treiber, DSYEVD, berechnet alle Eigenwerte und optional die Eigenvektoren mittels des Divide-and-Conquer-Algorithmus'.

- **relativ robuster Repräsentations (RRR)-Treiber:**
  Der relativ robuste Repräsentations-Treiber, DSYEVR, berechnet alle oder eine Auswahl der Eigenwerte, und optional die Eigenvektoren. Er ist - bis auf wenige Fälle - der schnellste von allen, und benötigt am wenigsten Speicherplatz. Zur Zeit ist er nur implementiert für den Fall, dass alle Eigenwerte gesucht sind. Soll nur ein Teil der Eigenwerte bestimmt werden, wird analog zu DSYEVX vorgegangen.

2.3.2 Eigenwertproblemlöser-Routinen

Die vier Eigenwertproblemlöser DSYEV, DSYEVX, DSYEVR und DSYEVD basieren auf einer Reihe anderer LAPACK-Routinen. Die Aufrufhierarchie ist in Abbildung 10 dargestellt.

Beschreibung der Routinen:

- **DSYTRD:** Transformation der symmetrischen Matrix in Tridiagonalform
  (wird aufgerufen von: DSYEV, DSYEVX, DSYEVD, DSYEVR)
- **DSTEQR:** Berechnung aller Eigenwerte und Eigenvektoren unter Anwendung des geshifteten QR-Algorithmus'; wenn Eigenvektoren gewünscht sind, zuerst Aufruf von DORGTR
  (wird aufgerufen von: DSYEV)
- **DSTERF:** Berechnung aller Eigenwerte mittels wurzelfreier Version des QR-Algorithmus' [9], es werden keine Eigenvektoren berechnet
  (wird aufgerufen von: DSYEV, DSYEVD, DSYEVR)
- **DSTEIN:** Bestimmung der Eigenvektoren der Tridiagonalmatrix bei gegebenem Eigenwert mit Hilfe der Inversen Iteration, anschließend Aufruf von DORMTR
  (wird aufgerufen von: DSYEVX, DSYEVR)
- **DSTEBZ:** Berechnung der gewünschten Eigenwerte in einem gegebenen Bereich mit Bisektion
  (wird aufgerufen von: DSYEVX, DSYEVR)
- **DSTEDC:** Berechnung der gewünschten Eigenwerte und Eigenvektoren der Tridiagonalmatrix mit Hilfe des Divide-and-Conquer-Algorithmus', anschließend Aufruf von DORMTR
  (wird aufgerufen von: DSYEVD)
- **DSTEGR:** Berechnung aller Eigenwerte und Eigenvektoren der Tridiagonalmatrix mittels des RRR-Algorithmus'
  (wird aufgerufen von: DSYEVR)
- **DORGTR:** Erzeugung von Q aus der Reduktionsphase, für QR
- **DORMTR:** Rücktransformation der Eigenvektoren bei EVX, EVR und EDC
  (wird aufgerufen von: DSYEVR, DSYEVX, STEDC)

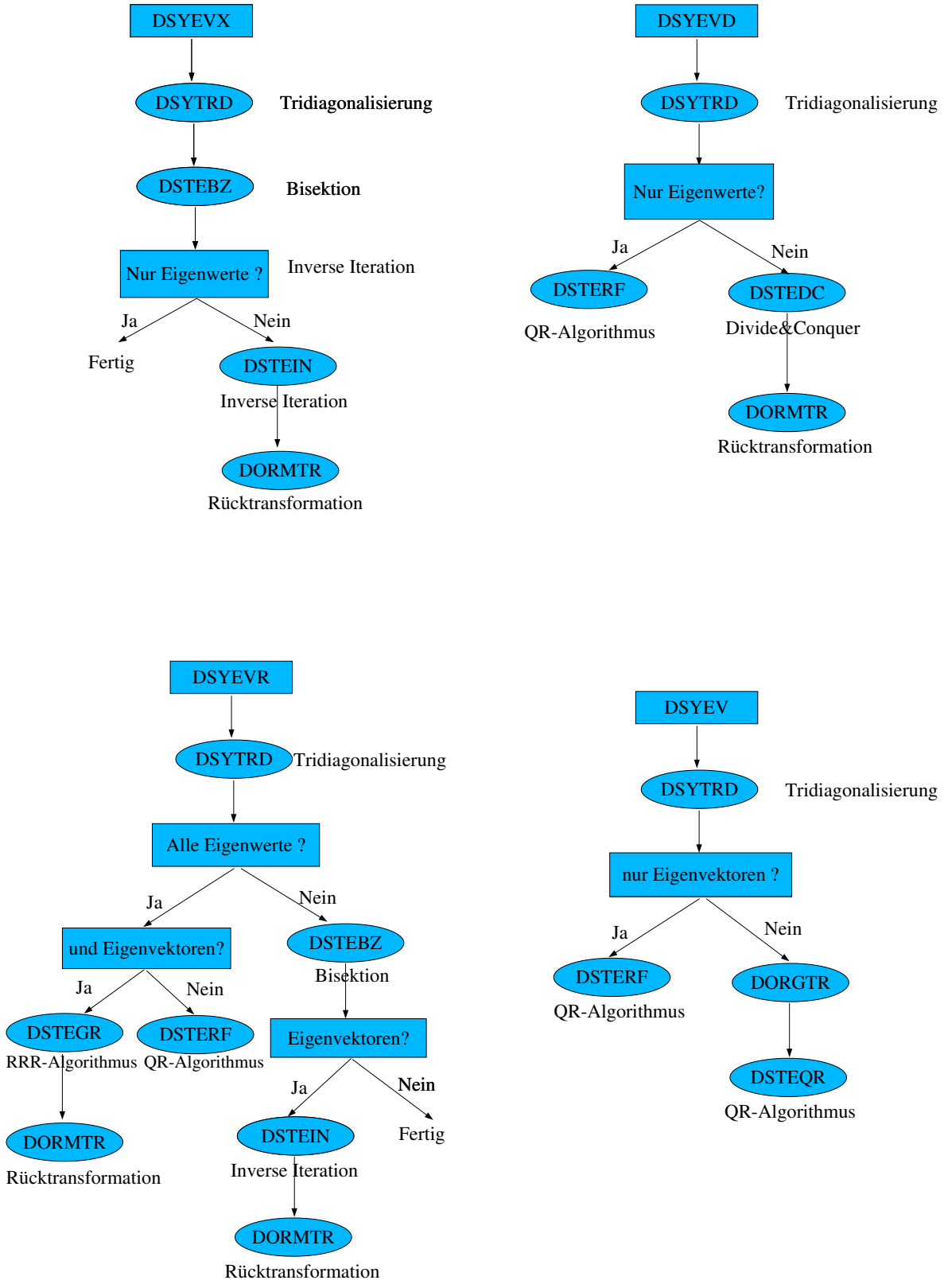

Abbildung 10: Aufrufhierarchie der seriellen Treiber-Routinen

2.4 Parallele Eigenwertproblemlöser in ScaLAPACK

ScaLAPACK[4] (Scalable Linear Algebra PACKage) ist eine Programmbibliothek für Parallelrechner mit verteiltem Speicher, die hocheffiziente Routinen für Probleme aus der Linearen Algebra wie lineare Gleichungssysteme, Eigenwertprobleme und Singulärwertberechnungen bereitstellt. ScaLAPACK stellt eine Weiterentwicklung der LAPACK-Bibliothek dar und basiert auf der parallelisierten Variante, der Parallel BLAS- bzw. PBLAS-Library und den Basic Linear Algebra Communication Subroutines (BLACS).

2.4.1 Eigenwertproblemlöser-Treiber

Es gibt drei verschiedene Treiber-Routinen in ScaLAPACK, um symmetrische Eigenwertprobleme parallel zu lösen: PDSYEV, PDSYEVD und PDSYEVX.

Erläuterung der Namensgebung: Die Namen der Routinen setzen sich ebenso wie bei den seriellen Routinen zusammen. Es wird lediglich ein 'P' vorangestellt, um zu kennzeichnen, dass es sich um eine parallele Routine handelt. Die Treiber und Routinen arbeiten nach dem gelichen Verfahren. Ihre Beschreibungen sind dem seriellen Abschnitt aus 2.3 zu entnehmen.

Speicherplatz-Anforderungen der Eigenwertproblemlöser-Treiber: In ScaLAPACK werden für die Routinen zur Eigenwertproblemlösung Speicherplatz-Anforderungen gemäß Tabelle 1 beschrieben.

Routine	Speicherplatz pro Prozessor	
	(ohne Eigenvektoren)	(mit Eigenvektoren)
PDSYEV	$\approx N^2/P + (NB+8)\cdot N$	$\approx 3N^2/P + (NB+8)\cdot N$
PDSYEVX	$\approx N^2 + (6+NB)\cdot N$	$\approx N^2/P \cdot (2+\alpha) + 6N$
PDSYEVD	(nicht möglich)	$\approx 4N^2/P + 9N$

Tabelle 1: Speicheranforderungen

NB = Blockgröße (ab S.44 genauer erklärt); N = Dimension der tridiagonalen Matrix; $\alpha = \frac{Anzahl\ generierter\ Eigenvektoren}{N}$, Vorfaktor abhängig von der Anzahl der Eigenwerte (25%, 50% oder 100%); $P = NPROW \cdot NPCOL$; $NPROW$ = Zahl der Prozessorzeilen; $NPCOL$ = Zahl der Prozessorspalten im Prozessorgitter

Folgerung: PDSYEVD braucht am meisten Speicherplatz: $\mathcal{O}(4N^2/P)$. PDSYEV braucht $\mathcal{O}(3N^2/P)$ und bei PDSYEVX mit Berechnung von 100% der Eigenwerte und Eigenvektoren wird auch $\mathcal{O}(3N^2/P)$ benötigt, bei kleiner Anzahl kann Speicherplatz gespart werden. Allerdings kann zur Reorthogonalisierung bis zu $\mathcal{O}(N^2)$ auf einem Prozessor gebraucht werden.

2.4.2 Eigenwertproblemlöser-Routinen

Die in den folgenden Abbildungen aufgezeigten Routinen arbeiten analog zu den seriellen Routinen, beschrieben in 2.3.2. Die Routine DSTEQR2 wird nicht parallel, sondern repliziert auf jeden Prozessor aufgerufen. Hinzu kommt folgende Routine:

- **PDGEMR2D:** Erzeugung einer Kopie der Matrix, als Voraussetzung zur Rücktransformation
  (wird aufgerufen von: PDSYEV)

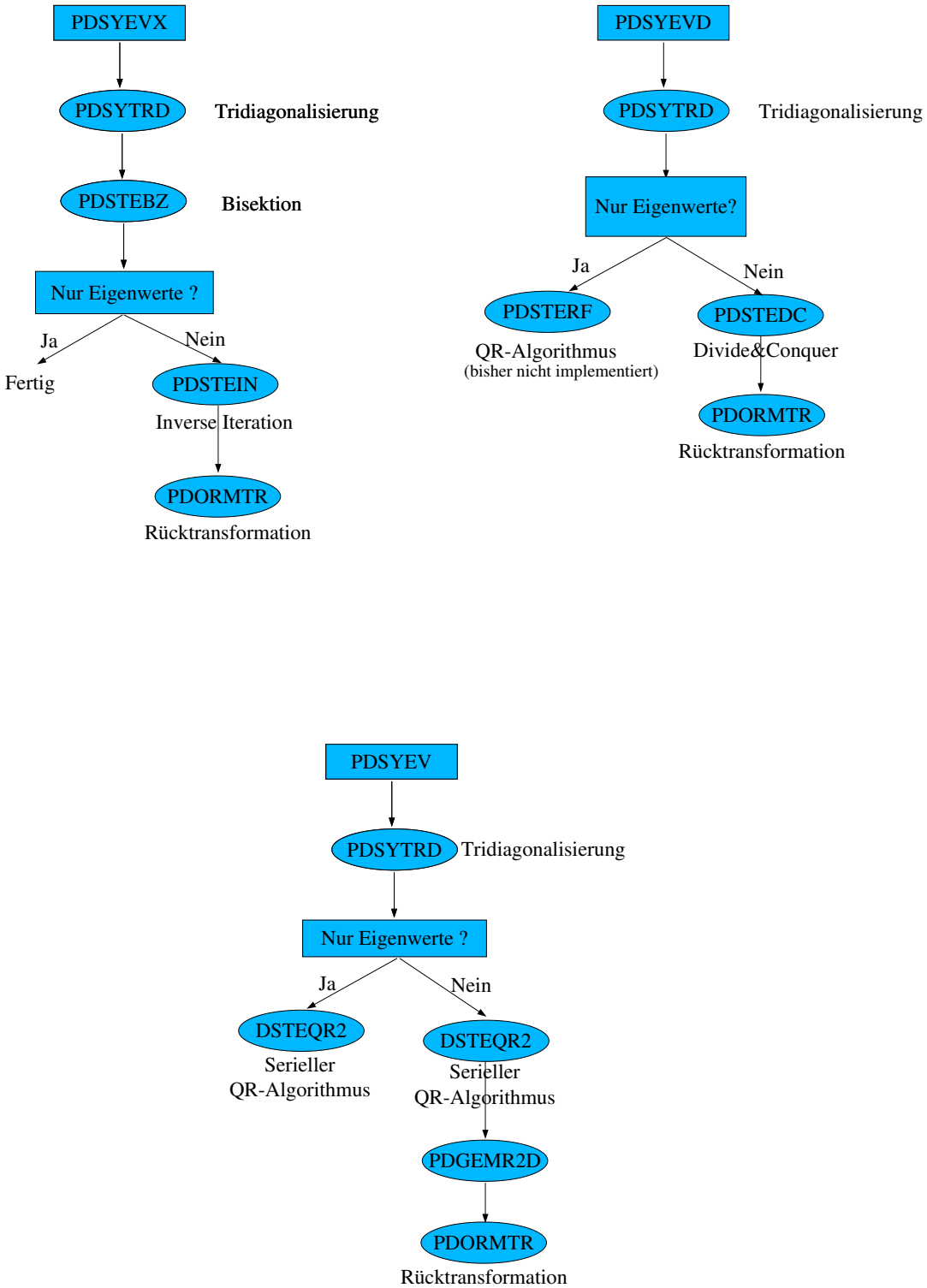

Abbildung 11: Aufrufhierarchie der parallelen Treiber-Routinen

3 Rechnerarchitekturen - Software und Hardware

3.1 Der Linux-SMP-Cluster ZAMPANO [10]

- **Konfiguration:**
  Zampano ist ein Cluster aus acht Knoten mit je vier Intel Pentium III Xeon Prozessoren mit je 550 MHz und einem Server-Knoten. Es besitzt 512 KB Zwischenspeicher pro Prozessor und 2 GB ECC-RAM pro Knoten. Die Peak-Performance beträgt 19.8 GFLOPS und der Gesamtarbeitsspeicher (einschließlich Server) 18 GB.

- **Betriebssystem:**
  Das Betriebssystem ist SUSE Linux 7.2 mit Kernel 2.4.4-4GB-SMP (SuSE).

- **Kommunikation:**
  Das SMP Cluster ist mit dem ZAM LAN (zamnet2) durch das Cluster front-end (zam008) verbunden. Diese Verbindung bietet ein Gateway für Fast-Ethernet. Intern sind die Cluster-Knoten mit Fast-Ethernet und Myrinet verbunden. Für parallele Programme gibt es MPI/MPICH-GM 1.2.1.7b über Myrinet. Bei der intra-node Kommunikation liegt die maximale Übertragungsrate ohne shared memory bei etwa 65 MB/s. Die Kommunikation mit Hilfe von shared memory wird durch die Option *–gm use-shmem* beim Programmaufruf angefordert. Für shared-memory intra-node Kommunikation beträgt dann die maximale Übertragungsrate 200 MB/s bei einer Wartezeit von etwa 15 μ Sekunden. Diese 200 MB/s stellen jedoch nur die Spitzenleistung dar, die bei Nachrichten (Messages) von etwa 64 KB auftritt. Bei Nachrichten von etwa 200 KB fällt er auf etwa 100 MB/s ab. Nachrichten werden hier nicht über das Netzwerk ausgetauscht, sondern vom sendenden Prozess auf dem shared memory abgelegt und dort vom empfangenden Prozess abgeholt. Die maximale Übertragungsrate über Myrinet liegt bei 120 MB/s zwischen den Knoten.

- **Anwendung:**
  Parallele Anwendungen können mittels qrun oder smprun (bei Reservierung) gestartet werden. Pro Knoten können vier Prozesse gestartet werden. Bestimmte Knoten können für interaktive oder batch-Programme online oder mittels *xsched* reserviert werden.

- **Bibliotheken:**

  BLAS: BLAS (Basic Linear Algebra Subprograms) definiert ein Set von Unterfunktionen, um einfache immer wieder auftretende Aufgaben mittels Rechnungen der Linearen Algebra durchzuführen. Es verfügt über drei Level: Level 1 BLAS führt Flops der Ordnung n auf Daten der Ordnung n aus (Vektor-Vektor), Level 2 BLAS führt Flops der Ordnung n^2 auf Daten der Ordnung n^2 aus (Matrix-Vektor) und Level 3 BLAS führt Flops der Ordnung n^3 auf Daten der Ordnung n^2 aus (Matrix-Matrix).

  Zur Verfügung steht die Version 3.2.1 der BLAS-Bibliothek des ATLAS-Projekts (Automatically Tuned Linear Algebra Software).

  Um die Fortran Bibliothek einzubinden müssen folgende Compiler-Optionen angegeben werden: *-g77libs -lf77blas -latlas*

Die Performance eines Algorithmus' wird im wesentlichen bestimmt durch den *Data Re-Use Faktor*. Er beschreibt die Anzahl der durchgeführten Operationen dividiert durch die Datenmenge, die zwischen Hauptspeicher und Prozessor (bzw. Cache-Speicher) ausgetauscht wird.
Für die Performance auf einem 226 MHz PentiumII gilt für Operationen bei Vektoren der Länge $n = 1000$ und $n \times n$-Matrizen [11]:

Operation	BLAS-Routine	Re-Use Faktor	MFlops/s
$\alpha = x^T y$	BLAS 1	1	36.1
$y = \alpha Ax + \beta y$	BLAS 2	2	61.1
$C = \alpha AB + \beta C$	BLAS 3	$n/2$	195.2

Tabelle 2: Überblick über BLAS-Routinen (226 MHz PentiumII)

Wie man in Tabelle 2 sieht, kann ein hoher Re-Use Faktor die Performance deutlich verbessern. Nur Level 3 BLAS-Routinen erreichen ca. 86% der Peak-Performance, Level 2 ca. 27%, Level 1 sogar nur 16%. Ähnliche Werte kann man auch bei anderen Rechnern mit Cache-Speicher erwarten.

LAPACK: LAPACK 3.0 wurde mit einigen optimierten Routinen des ATLAS-Projekts installiert. Es muss mit
-llapack -lf77blas -latlas -g77libs
eingebunden werden.

ScaLAPACK: ScaLAPACK Version 1.7 und BLACS Version 1.1 wurden installiert. Sie können mit
mpif90 name.f -lscalapack -lblacsF77init -lblacs -lblacsF77init -lf77blas -latlas -g77libs
eingebunden werden.

Myrinet: Myrinet ist ein kostengünstiges hochleistungsfähiges Paketkommunikations und Switching-Werkzeug, das weitgehend zur Zusammenschaltung von Clustern von Workstations, PCs, Servers oder Einplatinenrechnern benutzt wird. Cluster bieten auf ökonomische Weise

- **Hohe Leistung:** Sie wird dadurch erreicht, dass anspruchsvolle Berechnungen über ein Feld von kostengünstigen Hosts verteilt werden. Für eng gekoppelte verteilte Berechnungen muss die Verbindung eine Kommunikation zwischen den Hosts bieten, die über schnelle Datenübertragung und kurze Wartezeit verfügt.
- **Hohe Verfügbarkeit:** Sie wird dadurch erzielt, dass einer Berechnung erlaubt wird, mit einem Teil der Hosts fortzufahren. Die Verbindung sollte geeignet sein, Fehler zu entdecken und zu beseitigen, und sie sollte alternative Kommunikationspfade nutzen können.

Übliche Netzwerke wie Ethernet können dazu genutzt werden, Cluster aufzubauen. Sie bieten aber keine Funktionen oder Fähigkeiten, die für Hochleistungs- und Hochverfügbarkeits-Clusterung erforderlich sind. Mehr zu den Eigenschaften und Unterschieden von Myrinet zu anderen Netzwerken ist unter [12] zu finden.

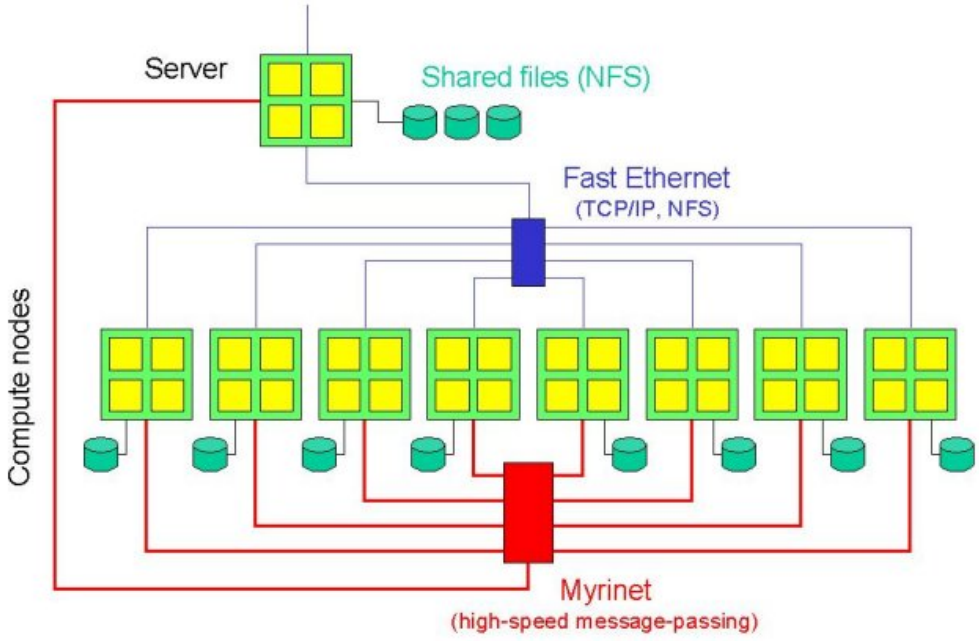

Abbildung 12: Konfiguration auf Zampano

GM: GM ist das low-level message-passing System für Myrinet-Netzwerke [13]. Das GM-System beinhaltet einen Treiber, ein Myrinet-Schnittstellen Kontrollprogramm, ein Netzwerk Abbildungsprogramm und die GM API Bibliothek. Die Eigenschaften von GM umfassen:

- gleichzeitiger gesicherter Zugang auf Benutzerebene zur Myrinet-Schnittstelle
- verlässliche Überlieferung der Nachrichten
- automatische Mapping- und Strecken-Berechnung
- automatisches Nachverfolgen von vorübergehenden Netzwerkproblemen
- Skalierbarkeit zu Tausenden von Knoten
- niedrige Wartezeit, hohe Datenrate und sehr geringe Host-CPU Belastung
- erweiterbare Software um simultanen direkten Support von GM API, IP (TCP, UDP), MPI, und anderen APIs zu ermöglichen

GM erreicht seine außergewöhnliche Performance mit einer Technik, die als Operating-System bypass (OS-bypass) bekannt ist. Nach anfänglichen Operating-System Aufrufen um Speicher für Kommunikation zu allokieren und zu registrieren, können Anwendungsprogramme Nachrichten ohne System-Aufrufe senden und empfangen. Stattdessen kommunizieren die GM API Funktionen über gewöhnlichen Speicher mit Hilfe des Myrinet-Kontrollprogramms, welches auf dem Prozessor der Myrinet-Schnittstelle kontinuierlich ausgeführt wird.

MPI - Message Passing Interface: Message Passing ist eine Möglichkeit, Daten zwischen Prozessen einer verteilten Anwendung auszutauschen, das insbesondere auf Maschinen mit verteiltem Speicher genutzt wird. Es gibt viele Variationen, doch das Grundkonzept, wie Prozesse über Nachrichten miteinander kommunizieren, ist überall das gleiche. Der größte Vorteil eines standardisierten Message-Passing-Modells ist die Portabilität und leichte Handhabbarkeit.
Das Ziel von MPI ist, ein weit genutztes Standard-Modell zu entwickeln, um Message-Passing-Programme zu schreiben. Als solches sollte die Schnittstelle ein praktisches, portables, effizientes und flexibles Modell für Message-Passing darstellen. Mehr Informationen dazu sind unter [14] zu finden.
BLACS baut auf MPI auf. Es enthält Message-Passing-Routinen speziell für Lineare Algebra. BLACS wurde entwickelt als MPI noch nicht existierte und war auch ein Standardisierungsversuch.

3.1.1 Tools für Messungen auf Zampano

RTC - Real Time Clock: Auf Zampano wurde die Zeitmessung mit Hilfe des RTC-Tools durchgeführt [15]. RTC liefert die Wall Clock Time, gemessen in Prozessor-Ticks. Um diese Ticks in Sekunden zu konvertieren werden sie mit 1.82119850d-9 multipliziert.

PCL - Performance Counter Library: Die Performance Counter Library [16] bietet eine Programmierschnittstelle, über die es möglich ist, die Werte bestimmter Performance-Counter zu bestimmten zählbaren Ereignissen zu erhalten. Im Rahmen der vorliegenden Arbeit sollte die Anzahl der Floating-Point-Operationen (Flop) mit Hilfe des PCL-Tools ermittelt werden.
Da dieses Tool nicht auf Zampano installierbar ist, wurde versucht, es auf einem lokalen Linux-PC zu installieren. Hier sollte die Anzahl der Floating-Point-Operationen gemessen werden, um daraus anschließend die Mega-FLOP-Rate zu errechnen. Zur Installation wurde ein kernel patch nötig, doch auch dies scheiterte. Somit musste an dieser Stelle leider auf die Messung der Anzahl an Floating-Point-Operationen und der Mega-Flop-Rate verzichtet werden.
Da die Anzahl der Flops in erster Näherung rechnerunabhängig sein sollte, und die Ermittlung der Flops auf IBM Regatta erfolgreich waren, können die Ergebnisse von dort auf Zampano und generell auf alle anderen Rechnertypen übertragen werden.

VAMPIRTRACE: Das VAMPIRTRACE-Profiling-Tool [17] für MPI-Anwendungen produziert sogenannte Trace-Dateien, die mit dem Vampir-Performance-Analysis-Tool analysiert werden können. Es zeichnet alle Anfragen an die MPI-Bibliothek und alle überbrachten Nachrichten auf, und erlaubt beliebige benutzerdefinierte Ereignisse zu definieren und aufzuzeichnen. Die Instrumentierung kann zur Laufzeit ein- oder ausgeschaltet werden. Ein Filtermechanismus hilft, die Menge der entstehenden Trace-Daten zu limitieren. Vampirtrace ist eine Erweiterung zu existierenden MPI-Implementationen. Lediglich die Verlinkung der Anwendung mit der Vampirtrace- Profiling- Bibliothek ermöglicht das Aufzeichnen aller Aufrufe von MPI-Routinen.

3.2 Der IBM Server p690 (Regatta)[18]

- **Konfiguration:**
  Ein IBM Regatta-Knoten verfügt über 32 Prozessoren des Typs Power4+,1.7 GHz. Jeder Prozessor verfügt über 2 FPUs. Der Hauptspeicher beträgt 128 GB, mit einer Taktrate von 567 MHz. Diese Taktrate könnte der Prozessor erreichen, vom Memory wird jedoch nur etwa ein Drittel der Taktrate erreicht. Die Latenz beträgt 252 Zyklen. Die Peak-Performance liegt bei 218 GFLOPS pro Knoten, das Gesamtsystem besteht derzeit aus sechs Knoten, insgesamt ergibt sich damit eine Maximalleistung von 1.3 TFLOPS. Es gibt 8 TB global verfügbaren Plattenspeicherplatz.

  Der Cache-Speicher ist in 3 Level aufgeteilt:

  - Interner L1 Cache: 64 KB Instruction Cache, 32 KB Data Cache (pro CPU)
  - Gemeinsamer L2 Cache: 1.5 MB pro Chip (= 2 CPUs)
  - Gemeinsamer L3 Cache: 512 MB pro Frame (= 32 CPUs)

  Vor allem L2-Cache ist wichtig, da die Verbindung zum Hauptspeicher zu langsam ist. Durch die Multiplikation-Addition-Anweisung und 2 FPUs können pro Takt theoretisch 4 Flop erreicht werden $\Rightarrow$ Peak pro Knoten $= 4 \cdot 1.7 \cdot 32 = 218$ GFlop pro Frame ($4 \cdot 1.7$ GFlop pro Prozessor)

  Knoten ist die Betriebssystemsicht, *Frame* die Hardwaresicht. Hier gibt es einen Knoten pro Frame, möglich sind zum Beispiel auch 4 Knoten zu je 8 Prozessoren je Frame.

- **Betriebssystem:**
  Das Basis-Betriebssystem für AIX ist ein UNIX-Derivat. Die aktuelle Version ist AIX 5.1.

- **Kommunikation:**
  Die Knoten sind verbunden mit Gigabit Ethernet. Für parallele Anwendungen kann zur Zeit jeweils nur ein Knoten verwendet werden, da die Kommunikation mit MPI über Knoten hinweg derzeit, bis zum Einbau des Hochleistungsswitches, (noch) nicht zulässig ist.

- **Bibliotheken:**
  Zur Verfügung stehen die Basis-Bibliotheken zur Linearen Algebra ESSL (Engineering and Scientific Subroutine Library), PESSL (Parallel ESSL) [19], BLAS, LAPACK, und ScaLAPACK im 32-Bit and 64-Bit-Modus. Sie werden mit den Optionen *-lscalapack_r -lessl -lblacspd_r -lblacsF77initpd_r* eingebunden.

- **Anwendung:**
  Mittels *llrun* können die parallelen Anwendungen interaktiv gestartet werden. Bei einem Aufruf kann die Anzahl der Knoten (zur Zeit maximal ein Knoten) und Prozessoren (maximal 32 Prozessoren pro Knoten) in folgender Form angegeben werden: *llrun -n 1 -p 32 maintime*
  Für Batchverarbeitung wird ein Job mit *llsubmit* an den Load Leveler gegeben. Dieser setzt ihn auf die Warteschleife und lässt ihn laufen, sobald genügend Prozessoren verfügbar sind.

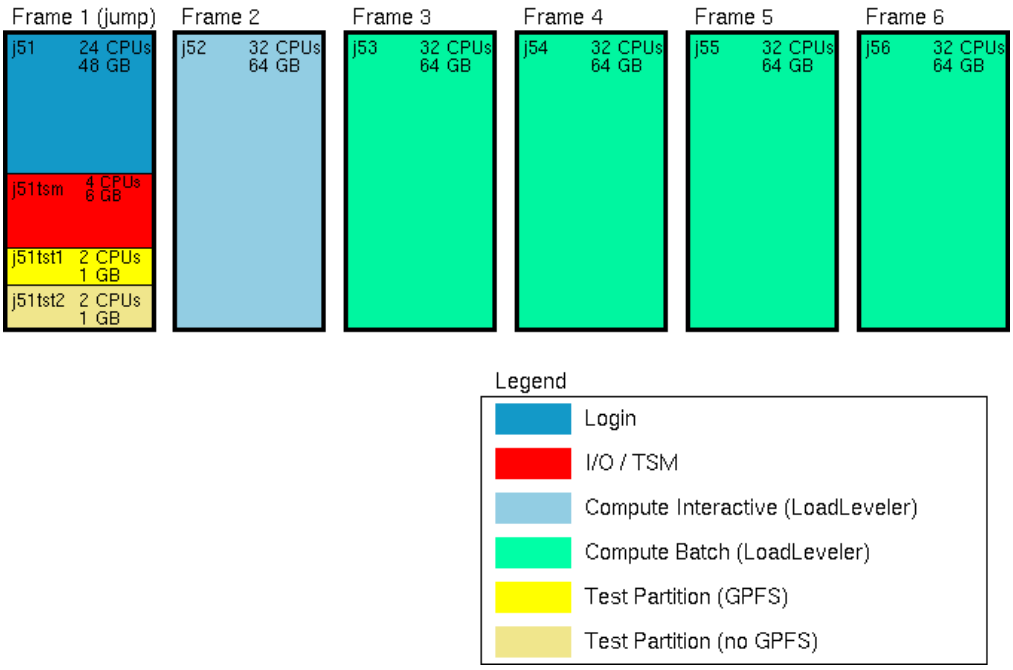

Abbildung 13: Konfiguration der IBM Regatta

3.2.1 Tools für Messungen auf IBM Regatta

HPM-Toolkit: Alle Messungen wurden mittels des HPMCOUNT-Tools gemessen [20]. Das HPM-Toolkit wurde entwickelt, um Performance-Messungen von Anwendungen auf IBM Systemen durchzuführen. Es unterstützt die Prozessoren Power3 und Power4 mit den Betriebssystemen AIX 5L und AIX 4.3.3. Das Toolkit besteht aus:

- **hpmcount:** Hpmcount startet eine Anwendung und liefert nach der Ausführung die benötigte Zeit, Informationen zur Ausführung, *abgeleitete Hardware Metrik* [2] und Statistiken zur Auslastung.
- **libhpm:** Libhpm ist eine *Instrumentierungs-Bibliothek* [3], welche instrumentierte Programme liefert, mit einer Inhaltsausgabe, welche die oben erwähnten Informationen für jeden instrumentierten Bereich in einem Programm enthält. Diese Bibliothek unterstützt sowohl serielle als auch parallele Anwendungen, in Fortran, C und C++.
- **hpmviz:** Hpmviz ist eine graphische Benutzerschnittstelle, welche die von libhpm generierte Performance-Datei graphisch visualisert.

Ausgabeparameter von HpmCount:

- **FDiv:**
  Anzahl aller ausgeführten Divisions-Anweisungen

[2] Hardware Metrik bezeichnet die Zähler von Laden, Speichern, Cachezugriffen und Anweisungen
[3] Eine Instrumentierungs-Bibliothek veranlasst, dass die Ausführung eines Programms protokolliert wird. Dabei können Start und Ende des zu protokollierenden Teils des Programms durch Statements im Quelltext angegeben werden.

- **FMA:**
  Anzahl aller ausgeführten Multiplikation-Addition-Anweisungen, dabei wird eine Addition kombiniert mit einer Multiplikation als eine Multiplikation-Addition-Operation gezählt.

- **FPU 0** und **FPU 1:**
  Hardware Floating Point Anweisungen, die auf den beiden Floating-Point-Units, Unit 0 und Unit 1, gezählt werden (ganzzahlige Werte)

- ***Flip*** [4]**:**
  Floating-Point-Anweisungen plus FMA (ganzzahliger Wert, oder in MFlips angegeben)
  FMA wird dazugezählt, da sonst solch eine Anweisung nur als eine einfache Operation gezählt werden würde. Die Speicherungen auf den FPU-Units 0 und 1 werden auf Power4 mitgezählt (FPU 0 und FPU 1) und müssen daher für die Rate wieder abgezogen werden.
  Power4: Flip = FPU 0 + FPU 1 + FMA - FPU-Speicherungen

- **MFlip-Rate:**
  Rate von Floating Point Anweisungen plus FMA (in MFlips/sec)
  0.000001 * Flip / Wall clock time

Verwendung von libhpm: Um die Bibliothek libhpm zu verwendet muss sie im Makefile mit *-lhpm_r -lpmapi -lm* verlinkt werden. Es wurde jeweils genau die zu untersuchende LAPACK- bzw. ScaLAPACK-Routine instrumentiert.

[4] Flip werden von IBM abweichend von Flop definiert. In dieser Arbeit wird jedoch einheitlich der Begriff Flop verwendet.

4 Serielle Performance

4.1 Zielsetzung

In diesem Kapitel werden die Untersuchungen zur seriellen Performance von vier verschiedenen Eigenwertproblemlösern auf zwei Rechnern unterschiedlicher Charakteristik (SMP versus Cluster von SMPs) beschrieben. Ziel ist, eine Modellfunktion zur Performance von seriellen Eigenwertproblemlösern zu parametrisieren und die Parameter anhand von Messungen zu testen. Die Parameter des Modells sind die Dimension der Matrix, deren Eigenwerte und Eigenvektoren berechnet werden sollen, der Eigenwertproblemlöser und die Charakteristiken der Rechner, auf denen die Messungen durchgeführt werden.

4.2 Performance-Modelle

Flop-Modell: Die erforderliche Anzahl der Floating-Point-Operationen (Flop) zur Lösung der Eigenwertprobleme der Dimension N wird mittels folgender Modellfunktion berechnet:

$$Flop \approx \alpha' \cdot N^{\beta'} \tag{4.1}$$

Der Vorfaktor α' ist ein methodenabhängiger und weitgehend maschinenunabhängiger Parameter, β' stellt die formale maschinenunabhängige Komplexität der Methode dar. Da die Komplexität aller Verfahren insgesamt -wie ab Seite 28 beschrieben- $O(N^3)$ beträgt, wird für β ein Wert von 3 erwartet.

Zeitabschätzung: Die Gesamtrechenzeit wird in erster Näherung durch die Zahl der Flop bestimmt. Die Rechenzeit t wird durch eine analoge Modellfunktion approximiert:

$$t = \alpha \cdot N^{\beta} \tag{4.2}$$

Wie in Gleichung 4.2 stellt β ein Maß für die Komplexität der Methode dar. Der Vorfaktor α ist methoden- und maschinenabhängig. α ist proportional zu α' und hängt von der Peak-Performance ab. Idealerweise wird erwartet, dass $\alpha = \frac{\alpha'}{Peak-Performance}$ ist. Bei BLAS3-Routinen kann laut Tabelle 2 nur etwa 90% der Peak-Performance erreicht werden. Aufgrund des schlechten Cache-Re-Use-Faktors fällt die maximale Performance für BLAS2 und BLAS 1 deutlich ab. Laut [21] verwenden die Eigenwertproblemlöser DSYEVX und DSYEV bei Matrizen großer Dimensionen ($N \geq 2000$) bis zu 27% bzw. 9% BLAS2 und bis zu 30% bzw. 9% BLAS3. So kann α als nur recht klein klein erwartet werden. Daher wird α' in der Praxis bei $\alpha = \frac{\alpha'}{0.4 \cdot Peak-Performance}$ erwartet, da nur etwa ein Drittel 50%, ein weiteres Drittel 30% und der Rest weniger als 15% der Peak-Performance erreicht. Die Komplexität sollte $\beta' = \beta$ ergeben.

Anpassung der Parameter mittels GNUPLOT: GNUPLOT ist ein Plot-Programm zur graphischen Visualisierung von Funktionen und tabellierten Daten [22]. Es verfügt über die Möglichkeit, für gegebene Messwerte einen nichtlinearen least-squares-Fit mit Hilfe des Marquardt-Levenberg-Algorithmus' durchzuführen. Das Minimum ist jedoch nicht leicht zu finden, die Approximation hängt von dem Startwert ab, eventuell wird ein anderes lokales Minimum gefunden.
Gegebene Startwerte für α und β lassen sich aus den Messzeiten ermitteln. Bei der Berechnung werden die Rechenzeiten für eine Matrix der Dimension 1000 (t_1), und der größten Matrix (t_2) (siehe Tabelle der gemessenen Zeiten im Anhang A.3) benötigt.

α und β ergeben sich zu:

$$\beta = \frac{\log\left(\frac{t_1}{t_2}\right)}{\log\left(\frac{N_1}{N_2}\right)} \qquad \alpha = \frac{t_1}{N_1^\beta} \tag{4.3}$$

Logarithmierter Fit: Um einen sicheren Fit zu erhalten, wurde ein linearer Fit durchgeführt. Durch Logarithmieren der Fit-Funktion ergibt sich ein linearer Fit für α' und β' (Flop-Fit), bzw. für α und β (Zeit-Fit). Die Gleichungen (4.1) und (4.2) werden umgeformt zu:

$$\log Flop = \log \alpha' + \beta' \cdot \log N \tag{4.4}$$

$$\log Flop = \log \alpha + \beta \cdot \log N \tag{4.5}$$

Im Vergleich zur Verwertung der Modellfunktion (4.1) gewährleistet (4.4) eine angemessenere Wichtung des Fehlers über die gesamten Wertebereiche.

4.3 Durchführung der Messungen

Zur Durchführung der Messungen mit den LAPACK-Eigenwertproblemlösern DSYEV, DSYEVX, DSYEVD und DSYEVR wird die Dimension der Matrix, deren Eigenwerte und Eigenvektoren berechnet werden, eingelesen. Die Eigenwerte werden mit Zufallszahlen erzeugt und in aufsteigender Reihenfolge angegeben. Sie werden in einem Intervall zwischen 10 und 2000 gesucht. Dabei wird der Wertebereich, in dem Eigenwerte erzeugt werden, beachtet.
Mittels dieses vorgegebenen Eigenwertspektrums und einer aus einem normierten Zufallsvektor W generierten Unitärmatrix $U = E - 2WW^T$ wird die zu diagonalisierende symmetrische Matrix $A = U^T DU$ erzeugt. Die Diagonalmatrix D enthält das vorgegebene Eigenwertspektrum.
Die Eigenvektoren zu den Eigenwerten, die sich um weniger als $10^{-5} \cdot ||(A)||_2$ unterscheiden, sollen reorthogonalisiert werden.
Für jeden Eigenwertproblemlöser wurde der Bereich der Matrixdimension von N=500 bis N=10000 gewählt. Er umfasst bei allen Eigenwertproblemlösern zehn gleichmäßig über den Messbereich verteilte Messpunkte, wobei jeder der Messpunkte drei Einzelmessungen umfasst. Der Bereich ist rechnerabhängig so gewählt, dass die Rechenzeiten den Zeitrahmen von einer Stunde weitgehend nicht überschreiten.
Für jeden Eigenwertproblemlöser wurden alle Eigenwerte berechnet. DSYEVX und DSYEVR berechnen ebenso die Eigenwerte eines Spektrums, hier 25% und 50% der Eigenwerte. Der DSYEVR ruft intern die gleichen Funktionen auf wie der DSYEVX und wird daher in dieser Funktion nicht gesondert betrachtet.

Tests zur Verifizierung der Ergebnisse: Die Ergebnisse werden anhand folgender Kriterien verifiziert.

- Ermittlung des größten Fehlers zwischen berechneten und erwarteten Eigenwerten
- Überprüfung der euklidischen Norm $||A \cdot EV - EW \cdot EV||_2$, hier wird $\frac{||A \cdot EV - EW \cdot EV||_2}{Matrixdimension \cdot \epsilon_{machine}} < Toleranzgrenze$ als erfolgreich gewertet (Toleranzgrenze ist 20)
- Überprüfung der Orthogonalität der Eigenwertproblemlöser, auch hier wird $\frac{||EV^T EV - I||_2}{Skalierung} < Toleranzgrenze$ als erfolgreich gewertet

4.4 Auswertung der Messungen

Die Reproduzierbarkeit der erhaltenen Messwerte wird dadurch gewährleistet, dass die Messungen mehrfach und bei unterschiedlicher Auslastung wiederholt wurden. Die Messungen der Zahl der Flops konnte nur auf IBM Regatta durchgeführt werden (Abbildung 14).

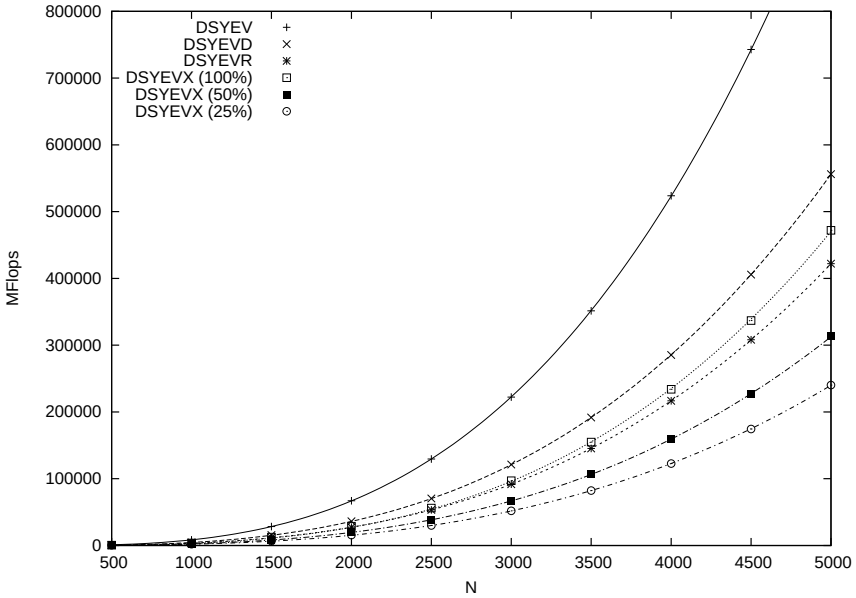

Abbildung 14: Flop-Fit aller Eigenwertproblemlöser (IBM Regatta)

- In allen Fällen konnte die Modellfunktion die Messdaten in hinreichender Genauigkeit reproduzieren. Alle Eigenwerteproblemlöser liefern ein praktisch identisches $\beta' \approx 3$, wie zu erwarten ist. Der Vorfaktor α' zeigt, dass der DSYEV mit Abstand am aufwendigsten ist, während DSYEVD und DSYEVX ca. 40% weniger Flop benötigen. Der DSYEVR bietet eine weitere Verringerung der Flop um ca. 15%.

- Der Rechenaufwand beim DSYEVX skaliert abhängig von der Anzahl der berechneten Eigenwerte und Eigenvektoren. Die Flops bei DSYEVX lassen sich mittels $Flop \approx (2 + 0.04 \cdot Eigenwert - Anteil) \cdot N^{\beta}$ berechnen. Dieser Befund ist allerdings abhängig von der Clusterung des Eigenwert-Spektrums und einer eventuell erforderlichen Reorthogonalisierung.

Verfahren	α' [Flop]	β'
dsyev (100%)	10.91 +/- 0.007907	2.97 +/- 0.002347
dsyevd (100%)	6.49 +/- 0.01424	2.95 +/- 0.01424
dsyevr (100%)	4.58 +/- 0.01446	2.96 +/- 0.004294
dsyevx (100%)	5.99 +/- 0.06241	2.94 +/- 0.01853
dsyevx (50%)	4.16 +/- 0.04041	2.94 +/- 0.012
dsyevx (25%)	3.03 +/- 0.02787	2.94 +/- 0.008273

Tabelle 3: Flop-Fit von α', β' und σ (IBM Regatta)

Erwartet wurde β' im Bereich von 3. Die ermittelten Werte für β' befinden sich sogar alle und teils sehr deutlich unterhalb von 3. Ihre Genauigkeit ist auf zwei Nachkomma-stellen begrenzt. Es handelt sich um einen Fit, und jeder erhaltene Wert kann eine Abweichung ab der dritten Nachkommastelle beinhalten. Der DSYEVX erhält unabhängig von der Anzahl der Eigenwerte und Eigenvektoren, die berechnet wurden (25%, 50% oder 100%) den gleichen Wert für β'.
Die gefitteten Werte für α' liegen alle im erwarteten Rahmen. Der α'-Wert bei DSYEV ist etwas höher, bei DSYEVX sinkt der Wert von α' mit der Anzahl der Eigenwerte und Eigenvektoren.

Fit der Messzeiten: Abbildung 15 zeigt den Fit von α und β bei dem Eigenwertproblemlöser DSYEVX (Zampano).

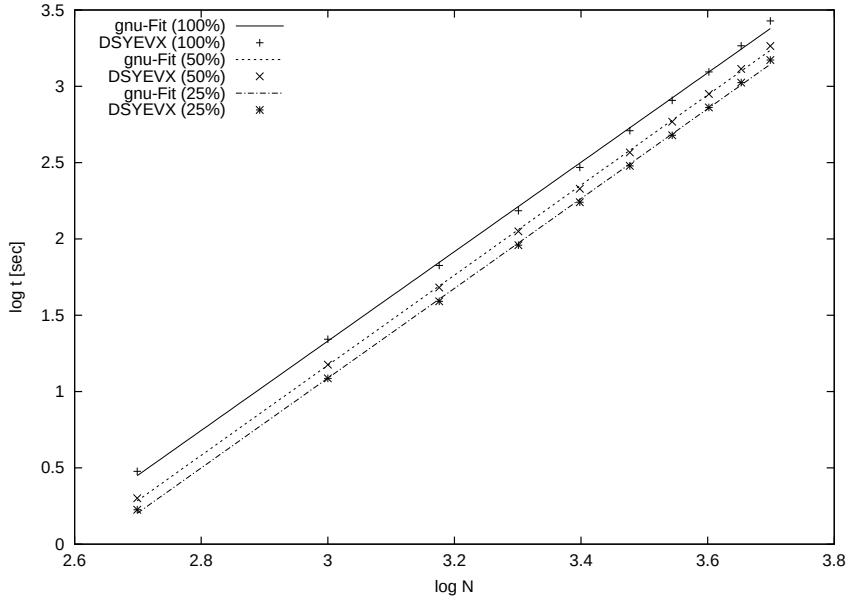

Abbildung 15: Fit der Zeitmessung mit DSYEVX (Zampano)

Die gefitteten Werte für α und β werden in Tabelle 4 (Zampano) aufgezeigt.

Verfahren	α [Sek.]	β
dsyev (100%)	$1.42 \cdot 10^{-7}$ +/- 0.2734	2.96 +/- 0.08118
dsyevd (100%)	$3.38 \cdot 10^{-8}$ +/- 0.03494	2.92 +/- 0.01037
dsyevr (100%)	$1.87 \cdot 10^{-8}$ +/- 0.06677	2.97 +/- 0.01982
dsyevx (25%)	$1.52 \cdot 10^{-8}$ +/- 0.06299	2.94 +/- 0.0187
dsyevx (50%)	$3.57 \cdot 10^{-8}$ +/- 0.05849	2.95 +/- 0.01736
dsyevx (100%)	$3.43 \cdot 10^{-8}$ +/- 0.1007	2.93 +/- 0.02989

Tabelle 4: Werte für α und β (Zampano)

In Abbildung 16 werden die gefitteten Werte von α und β bei DSYEVX (25%, 50% und 100% der Eigenwerte wurden bestimmt) graphisch veranschaulicht (IBM Regatta). Die explizit ermittelten Werte sind in Tabelle 5 (IBM Regatta) aufgeführt.

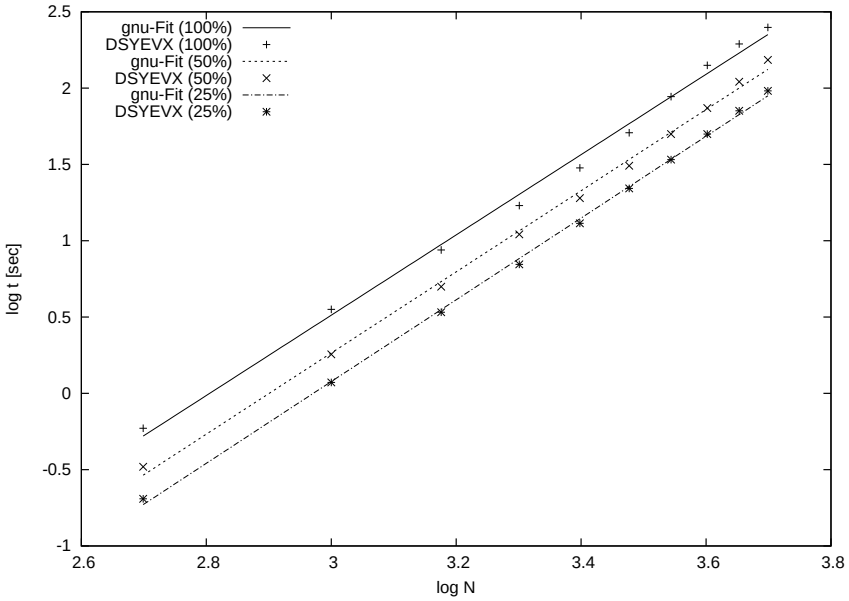

Abbildung 16: Fit der Zeitmessung mit DSYEVX (IBM Regatta)

Verfahren	α [Sek.]	β
dsyev (100%)	$2.66 \cdot 10^{-10}$ +/- 0.1355	3.41 +/- 0.04023
dsyevd (100%)	$4.98 \cdot 10^{-9}$ +/- 0.1378	2.87 +/- 0.0409
dsyevr (100%)	$1.8 \cdot 10^{-9}$ +/- 0.2024	2.98 +/- 0.0601
dsyevx (25%)	$2.32 \cdot 10^{-9}$ +/- 0.2141	2.87 +/- 0.06357
dsyevx (50%)	$3.88 \cdot 10^{-9}$ +/- 0.1474	3.09 +/- 0.04376
dsyevx (100%)	$4.61 \cdot 10^{-9}$ +/- 0.09683	2.9 +/- 0.02875

Tabelle 5: Werte für α und β (IBM Regatta)

Die gefitteten Werte für β sind, außer bei DSYEV und DSYEVX, auf beiden Rechnern immer kleiner als der erwartete Wert 3.
Die Werte für α sind extrem klein, da die gemessenen Zeiten für t_1 im Verhältnis zu N^β sehr klein sind, und somit der Bruch $\frac{t_1}{N_1^\beta}$ sehr klein ausfällt. N^β ist die Größenordnung der Anzahl der Operationen. Bei einem Rechner, der $O(10^8)$ Operationen pro Sekunde leistet, muss $\alpha = O(10^{-8})$ sein.
Die Erwartung für α ergibt sich aus

$$\alpha = \frac{\alpha'}{f \cdot Peak - Performance}. \tag{4.6}$$

α hat die Einheit Sekunden, α' hat die Einheit Flop. Somit hat $\frac{\alpha'}{\alpha}$ die Einheit $\frac{Flop}{Sekunden}$ und ist die Performance-Rate. Die Peak-Performance beträgt auf Zampano $5.5 \cdot 10^8$

und bei IBM Regatta $6.8 \cdot 10^9$ Flop/Sek., falls β und β' eines Eigenwertproblemlösers übereinstimmen, müsste Gleichung (4.6) gelten. Bei dem Beispiel des DSYEVD ($\beta = 2.87$ und $\beta' = 2.95$) erhalten wir $f = 19\%$. Es werden nur 19% der Peak-Performance erreicht.

Güte der Fits: Der Fit des DSYEVX auf Zampano sowie auf IBM Regatta ist sehr ausgeglichen. Interessant ist der systematische Fehler, bei dem beim DSYEVX mit der Berechnung von 100% der Eigenwerte die Messwerte bei kleinen Matrizen unterhalb der Fit-Kurve und bei größeren Matrizen etwas oberhalb der Fit-Kurve liegen.

Um den abhängig von der Matrixdimension optimalen Eigenwertproblemlöser zu finden, werden die Schnittpunkte zwischen en gefitteten Modellfunktionen berechnet. Diese Schnittpunkte werden wie folgt berechnet:

$$\alpha_1 \cdot N^{\beta_1} = \alpha_2 \cdot N^{\beta_2} \tag{4.7}$$

Es ergibt sich:

$$N = \left(\frac{\alpha_1}{\alpha_2}\right)^{\frac{1}{\beta_2 - \beta_1}} \tag{4.8}$$

Tabelle 6 listet die Schnittpunkte auf Zampano auf. In der Tabelle linksstehend ist die schnellere Routine, sie wird ab dem Schnitt bei der benannten Matrixdimension N von der rechtsstehenden Routine abgelöst.

schneidende Routinen	N (Zampano)
DSYEVX - DSYEVR	178*
DSYEVX - DSYEVD	3281
DSYEVR - DSYEVD	45177*
DSYEVX - DSYEV	400785*

Tabelle 6: Schnittpunkte auf Zampano

Abbildung 17 (Zampano) stellt den Verlauf und die Schnittpunkte der Eigenwertproblemlöser dar (logarithmische Skalierung).

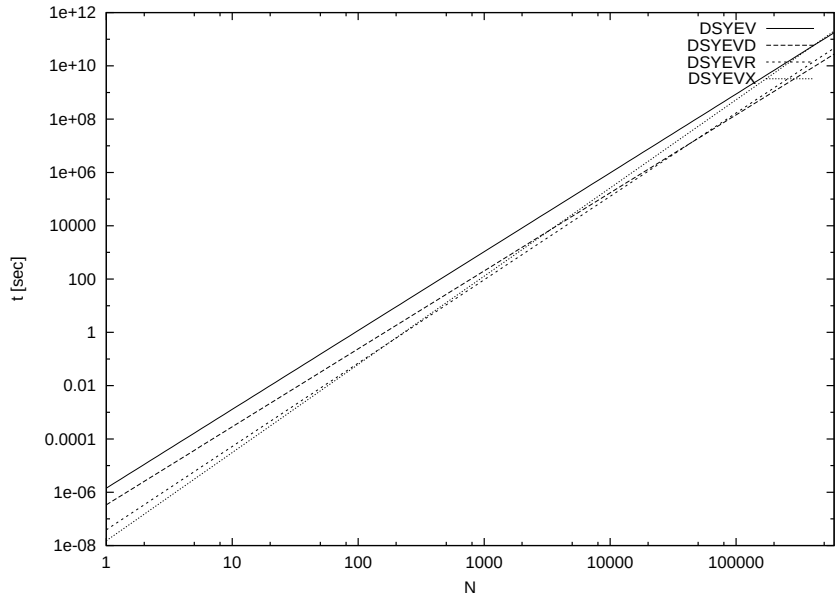

Abbildung 17: Schnittpunkte der Graphen (Zampano)

Auf IBM Regatta werden die in Tabelle 7 (IBM Regatta) aufgeführten Schnittpunkte ermittelt.

schneidende Routinen	N (IBM Regatta)
DSYEV - DSYEVX	261*
DSYEV - DSYEVR	692
DSYEVD - DSYEVR	1121
DSYEVX - DSYEVR	2095
DSYEVD - DSYEVX	7054

Tabelle 7: Schnittpunkte auf IBM Regatta

Abbildung 18 (IBM Regatta) zeigt die Schnittpunkte graphisch.

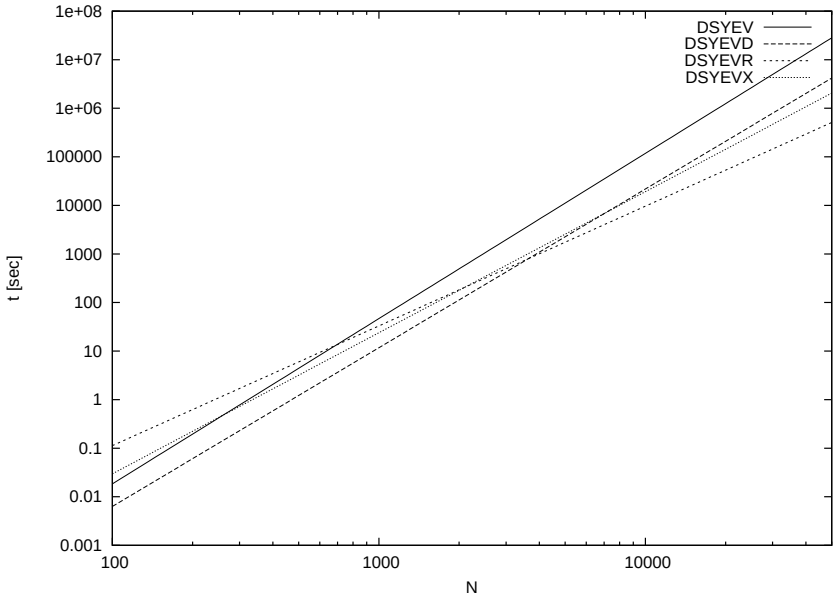

Abbildung 18: Schnittpunkte der Graphen (IBM Regatta)

Es gibt hier immer einen Schnittpunkt, wenn für $\beta_1 > \beta_2$ gilt $\alpha_1 < \alpha_2$. Dann ist $\frac{\alpha_1}{\alpha_2} > 1$ und $\frac{\alpha_1}{\alpha_2}^{\frac{1}{\beta_2 - \beta_1}}$. DSYEV und DSYEVD haben keinen Schnittpunkt, da sowohl $\alpha_1 > \alpha_2$ als auch $\beta_1 > \beta_2$ ist, Kurve 1 also immer größer als Kurve 2 ist.
Da der Messwertbereich auf Zampano und Regatta zwischen $N = 500$ und $N = 10000$ liegt, sind die mit Sternchen versehenen Werte extrapoliert und gegebenenfalls mit größerer Unsicherheit behaftet.

4.5 Bewertung der Ergebnisse

Vergleich von α und β: Die Werte von α und β bestimmen die Kosten des Verfahrens, abhängig von der Dimension der Matrix:

- N $\rightarrow \infty$: β bestimmt die Kosten, der Eigenwertproblemlöser mit dem kleinsten β wird am schnellsten
- N const: α bestimmt die Kosten, wenn die β ungefähr gleich sind

β gibt also an, wie teuer das Verfahren ist, falls die Matrix genügend groß ist. Daher ist im vorliegenden Fall β von großer Wichtigkeit.

Zeiten: Die kürzesten Rechenzeiten erhielten die Eigenwertproblemlöser auf den beiden Rechnern in Abhängigkeit von der Matrixdimension in folgender Aufstellung:

- **Zampano:** Hier ist der DSYEVR für die gesamte auf diesem Rechner praktikable Problemgröße die beste Wahl.
- **IBM Regatta:** Für Matrizen bis zu einer Dimension von etwa $N = 1121$ erzielt der DSYEVD eindeutig die kürzesten Zeiten zur Eigenwertproblemlösung. Für Matrizen, die eine Matrix größerer Dimension haben ist der DSYEVR der schnellste.

Speicherbedarf: Auf Zampano ist DSYEVX bis $N = 178$ zwar am schnellsten, er benötigt aber den meisten Speicherplatz, ist daher sehr teuer. Dies ist jedoch bei solch kleinen Matrizen unerheblich. Der DSYEVD benötigt etwas weniger Speicher und ist daher auf beiden Rechnern zu empfehlen. DSYEVR ist auf Regatta immer, auf Zampano bis zu einer Matrixdimension von $N \approx 45177$ am besten. Was größer ist als 45177, ist nur geschätzt, würde aber ohnehin nicht in den Speicher eines Prozessors passen.

5 Parallele Eigenwertlöser - Skalierbarkeit

5.1 Zielsetzung

In diesem Kapitel werden die Untersuchungen der parallelen Performance der drei Eigenwertproblemlöser PDSYEV, PDSYEVX und PDSYEVD auf den beiden Rechnern Zampano und IBM Regatta beschrieben. Ziel ist, die Skalierbarkeit der Peformance der verschiedenen Eigenwertproblemlöser anhand von *Amdahl's Gesetz* zu parametrisieren.

5.2 Amdahl's Gesetz

Die maximal erreichbare Beschleunigung durch Nutzung eines Parallelrechners ist durch den relativen seriellen Anteil eines Programms an der Gesamtrechenzeit begrenzt. Unter der Annahme, dass der parallele Anteil ideal skaliert ergibt sich die Rechenzeit auf p Prozessoren zu (**Amdahl's Gesetz**):

$$T(p) = T(1) \cdot \frac{1-a}{p} + T(1) \cdot a + t_{com} \tag{5.1}$$

a stellt den Teil des Programms dar, der nur seriell bearbeitet werden kann. $T(1)$ ist die Zeit, die das Programm bei Ausführung auf einem Prozessor benötigt. Diese Zeit bleibt für den nur seriell ausführbaren Teil a immer gleich. Für den Anteil $(1-a)$ wird sie im besten Fall durch p geteilt. Die Gesamtausführungszeit des parallelen Programms errechnet sich also aus der Summe der Ausführungszeit $T(1)$ multipliziert mit dem nur seriell ausführbaren Programmanteil a, und der Ausführungszeit $T(1)$ multipliziert mit dem parallel ausführbaren Programmanteil $(1-a)$. Die minimale Rechenzeit für beliebig viele Prozessoren ergibt sich zu $T(1) \cdot a$, der maximale Speedup ist somit $\frac{1}{a}$.
Amdahl's Gesetz vernachlässt Synchronisations- und Kommunikationszeiten. In der Praxis fällt die Kommunikation jedoch teils stark ins Gewicht, so dass ein zusätzlicher Term t_{com} für die Kommunikation eingeführt wird. Dieser Term hängt vom algorithmenabhängigen Datenvolumen und der verfügbaren effektiven Bandbreite ab.
Es ist wichtig, vor der Durchführung einer Messung genau abzuschätzen, wie viele Prozessoren nötig sind, und wie effektiv der Einsatz einer bestimmten Anzahl an Prozessoren sein kann. Es muss eine praktikable Obergrenze geben, welche angibt, wie viel schneller Multiprozessorsysteme im Vergleich zu Einprozessorsystemen sein können, denn ansonsten würden sich die Entwicklungskosten für Multiprozessorsysteme kaum lohnen.
Abbildung 19 zeigt den Verlauf der Speedup-Kurve nach Amdahl's Gesetz bei unterschiedlichen seriellen Anteilen.

5.3 Durchführung der Messungen

Für die ScaLAPACK-Eigenwertproblemlöser PDSYEV, PDSYEVX und PDSYEVD wurden Messungen zur Ermittlung der Performance durchgeführt. Dabei wurden für jeden Eigenwertproblemlöser Matrixdimensionen aus dem Bereich von $N = 500$ bis $N = 10000$ gewählt. Da hierbei im Vergleich zur seriellen Performance auch die Prozessoranzahl zu variieren ist, sind die Matrixdimensionen sinnvoll zu wählen: Wird kein weiterer signifikanter Speedup mehr für ein gegebenes N festgestellt, so braucht die Zahl der Prozessoren nicht weiter erhöht zu werden. Dabei gilt die Faustregel [4]: $n_{proc} < (int(N/750) + 1)^2$. Für eine Matrix der Dimension $N = 1000$ braucht man also maximal $2^2 = 4$ Prozessoren. Für jeden Eigenwertproblemlöser wurden 100% der

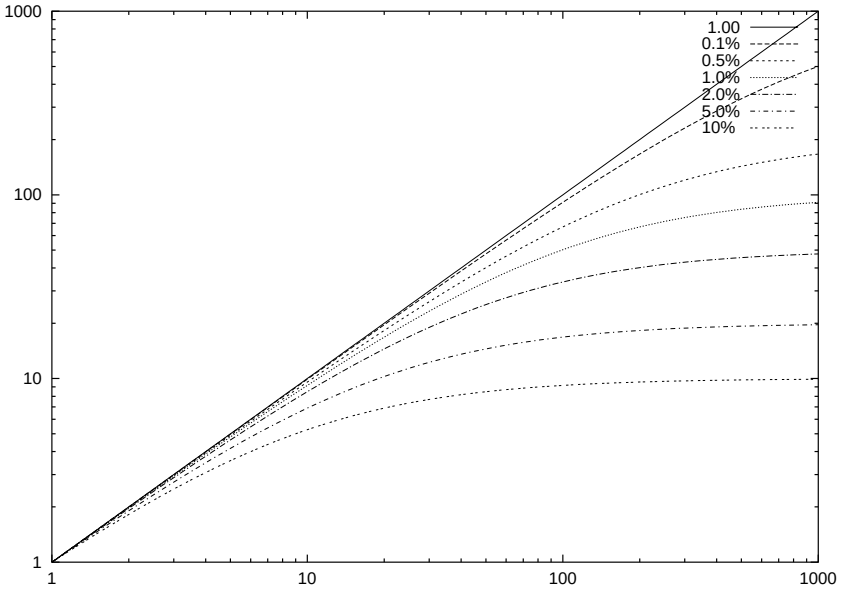

Abbildung 19: Speedup mit unterschiedlichen Amdahl-Werten

Eigenwerte berechnet. Bei dem Eigenwertproblemlöser PDSYEVX wurden die Messungen ebenso für 25% und 50% der Eigenwerte vorgenommen. Die Zahl der Prozessoren variiert je nach Rechner zwischen 1,2,4,8,12,16,20,24,28 und 32. Auf Zampano wurden Messungen mit 1, 2, 3 oder 4 Prozessoren auf einem Knoten durchgeführt. Dabei wurden große Schwankungen der Zeiten festgestellt, so dass außerdem Messungen mit einem Prozessor auf je einem Knoten, und Messungen mit 4 Prozessoren auf je einem Knoten durchgeführt wurden. Auf IBM Regatta stehen 32 Prozessoren auf einem Knoten zur Verfügung, so dass eine Aufteilung auf verschiedene Knoten nicht nötig ist (und außerdem zur Zeit nicht erlaubt ist). Es werden Zeiten, Flop-Anzahl und Mega-Flop-Rate gemessen.
Die Eingaben sind analog zur seriellen Berechnung (siehe Abschnitt 4.3). Hinzu kommt bei der parallelen Durchführung die Angabe der ScaLAPACK-Blockgröße und des Prozessorgitters für die Reduktion und Rücktransformation. Es ist wichtig, ein möglichst quadratisches Gitter zu wählen, also bei 4 Prozessoren 2×2. Dies ergibt eine bessere Lastverteilung und niedrigere Kommunikation als 1×4 oder 4×1.
Die Daten werden in ScaLAPACK block-zyklisch-2dimensional verteilt. Dabei wird die N x N-Matrix in Blöcke der Dimension NB x NB-Blöcke aufgeteilt. Diese werden dann 2d-zyklisch auf P = NPROW x NPCOL Prozessoren verteilt ([4], Kap.4).
Als Testrechner dienen wiederum der Zampano und der Regatta-Knoten. Da die Performance von der ScaLAPACK-Blockgröße abhängt, ist zunächst für jeden Rechner anhand eines mittelgroßen Beispiels eine geeignete Blockgröße zu ermitteln.

Blockgröße ermitteln: Um die für jeden Rechner geeignete Blockgröße zu ermitteln werden Testmessungen mit allen Eigenwertproblemlösern bei 4 Prozessoren, unterschiedlichen Matrixdimensionen und verschiedenen Blockgrößen (im Bereich von 10 bis 200) ausgeführt. Die Matrixdimension wird dabei so gewählt, dass die gemessene Zeit in einem Bereich von etwa zwei Minuten liegt. Auf Zampano entspricht dies etwa

der Matrixdimension $N = 1500$, auf IBM Regatta etwa $N = 2000, 4000, 5000$. Jede Messung wird zu unterschiedlicher Auslastung des Rechners wiederholt, um die Reproduzierbarkeit der Ergebnisse zu gewähren.
Abbildung 20 zeigt graphisch die Ermittlung der optimalen Blockgröße am Beispiel des PDSYEV. Tabelle 8 listet die konkret ermittelten Werte auf. Das deutliche Minimum ist bei allen drei Eigenwertproblemlösern ähnlich.

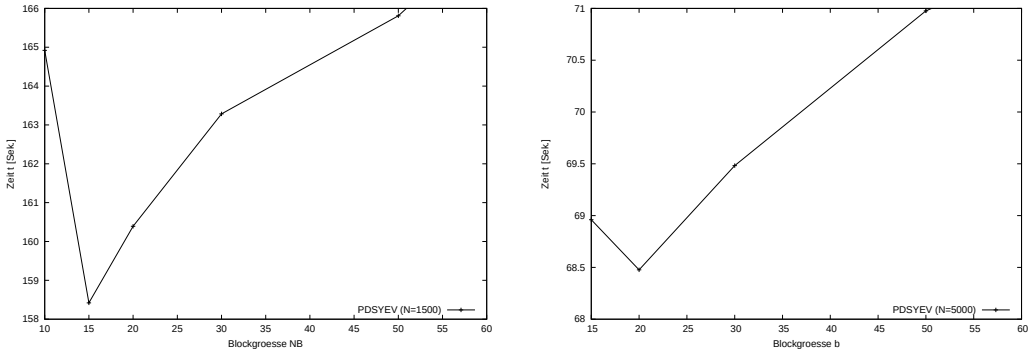

Abbildung 20: Blockgrößenermittlung (Zampano und IBM Regatta)

Eigenwertproblemlöser	Zampano	IBM Regatta
PDSYEV	15	20
PDSYEVD	20	30
PDSYEVX	30	30

Tabelle 8: Werte für Blockgröße (Zampano und IBM Regatta)

5.4 Auswertung der Messungen

Speedup der Eigenwertproblemlöser: Zu jedem Eigenwertproblemlöser lässt sich mittels folgender Formel der Speedup, das Verhältnis von $T(1)$ zu $T(p)$, bestimmen:

$$S(p) = \frac{T(1)}{T(p)} \approx \frac{2 \cdot T(2)}{T(p)} \tag{5.2}$$

$T(1)$ ist die Zeit bei Ausführung des parallelen Programms auf einem Prozessor, und $T(p)$ die Zeit bei der Ausführung auf p Prozessoren, also ist $T(2)$ die Zeit bei zwei Prozessoren. Falls $T(1)$ nicht gemessen werden kann, kann $2 \cdot T(2)$ als Abschätzung dienen.
Für den Speedup ergeben sich unter Berücksichtigung von (5.1) folgende Umformungen:

$$S(p) = \frac{T(1)}{T(1)\frac{1-a}{p} + T(1)a} = \frac{1}{\frac{1-a}{p} + a} = \frac{p}{(1-a) + p \cdot a} \xrightarrow{p \to \infty} \frac{1}{a} \tag{5.3}$$

Der Speedup $S(p)$ konvergiert bei steigender Prozessoranzahl gegen die obere Schranke $\frac{1}{a}$.

Der Speedup für PDSYEVD auf Zampano, bei den Matrixdimensionen $N = 1500, N = 3000$ und $N = 4500$, mit der Knoten-Prozessor-Anordnung (Knoten x Prozessoren): 1x1, 1x2, 2x2, 2x4, 3x4, 4x4 und 5x4, ist in Tabelle 9 aufgelistet. Abbildung 21 zeigt die zugehörige Graphik.

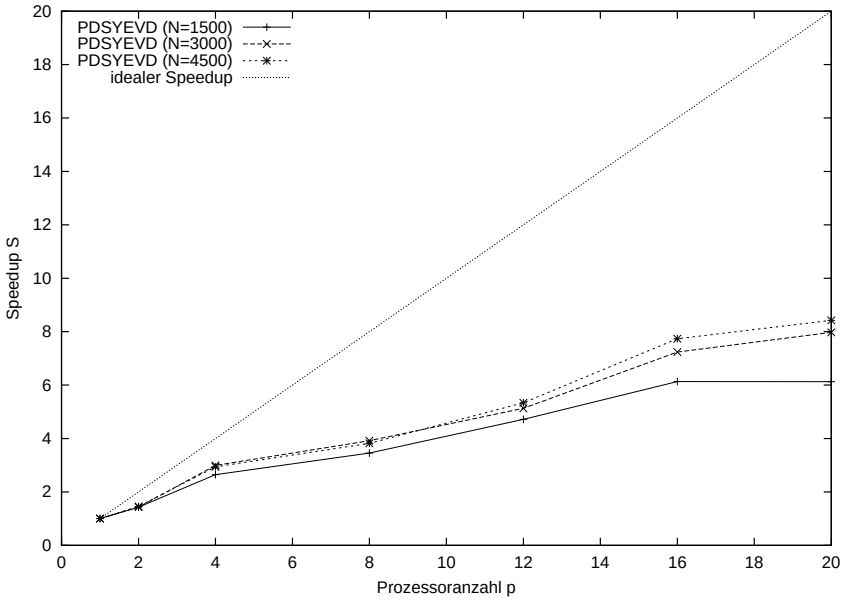

Abbildung 21: Speedup bei PDSYEVD (Zampano)

Prozessoranzahl	Speedup (N=1500)	Speedup (N=3000)	Speedup (N=4500)
1	1	1	1
2	1.427031864	1.433147729	1.456337226
4	2.645306034	2.990461374	2.939221225
8	3.453867008	3.917171275	3.820195328
12	4.717771477	5.131710207	5.336665202
16	6.13319986	7.239018958	7.734857601
20	6.127534086	7.972000329	8.421076325

Tabelle 9: Speedup PDSYEVD (Zampano)

Der Speedup steigt bis zu einer Prozessorzahl von 4 relativ stark an, bei höherer Prozessoranzahl, zum Beispiel 8, wird die Steigung flacher. Dies liegt an der Kommunikation zwischen den Knoten: Auf Zampano können auf einem Knoten 4 Prozessoren über den gemeinsamen Speicher recht schnell untereinander kommunizieren. Sobald mehr als 4 Prozessoren benötigt werden, die Kommunikation also über einen Knoten hinaus geht, beeinträchtigt die damit verbundene Kommunikation über Myrinet den Speedup sehr.

Auf IBM Regatta ergeben sich für den Speedup bei PDSYEVX die in Tabelle 10 aufgelisteten Werte, Abbildung 22 ist die zugehörige Abbildung.

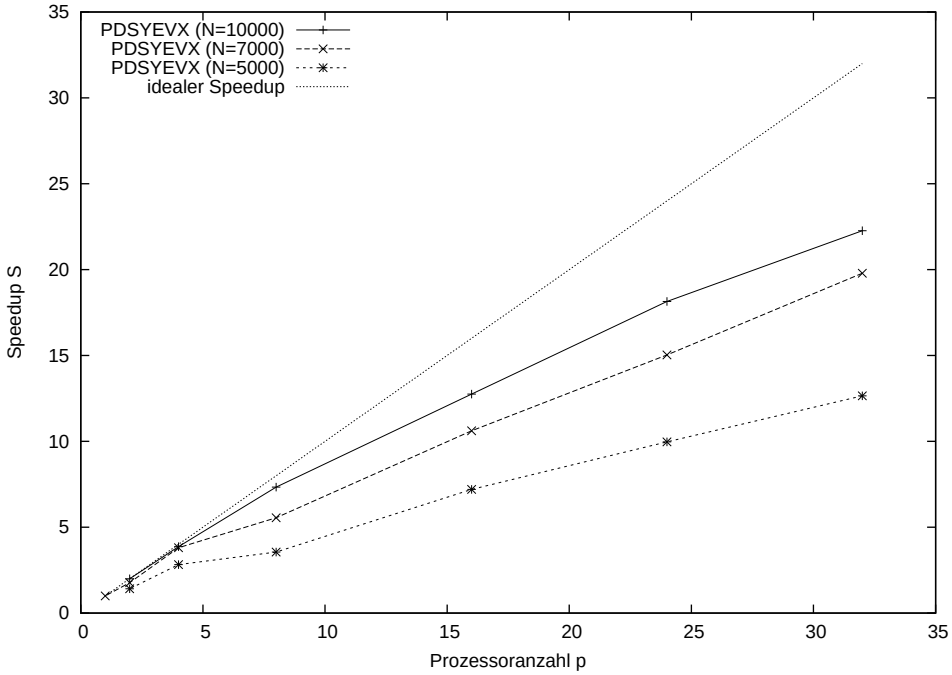

Abbildung 22: Speedup bei PDSYEVX (IBM Regatta)

Prozessoranzahl	Speedup (N=5000)	Speedup (N=7000)	Speedup (N=10000)
2	2	1.772560027	
4	3.980000527	3.806348342	4
8	5.013878475	5.551288241	5.065983562
16	10.15168238	10.61575221	12.24745422
24	14.04862901	15.0344019	14.96289398
32	17.82994418	19.7915796	19.1452106

Tabelle 10: Speedup PDSYEVX (IBM Regatta)

Bei steigender Prozessoranzahl ist ein klar steigender Speedup zu bemerken. Je größer die Matrix, deren Eigenwerte berechnet wurden, desto näher kommt der Speedup dem idealen Speedup. Da bis zu 32 Prozessoren auf einem einzigen Knoten arbeiten, entstehen auf der IBM Regatta keine starken Kommunikationsschwankungen, es kann durchgehend auf einem Knoten kommuniziert werden. Dies geschieht über shared memory, was sehr schnell ist.

Bestimmung des seriellen Anteils nach Amdahl: Um den seriellen Anteil eines Eigenwertproblemlösers zu bestimmen, kann der jeweils berechnete Speedup anhand der Formel

$$S(p) = \frac{p}{(1-a) + p \cdot a} \tag{5.4}$$

nach a gefittet werden. Löst man Gleichung (5.1) nach a auf, so ergibt sich folgende Gleichung für den seriellen Anteil:

$$a = \frac{p - S(p)}{S(p) \cdot (p-1)} \tag{5.5}$$

mit $S(p) = \frac{T(1)}{T(p)} \approx \frac{2 \cdot T(2\ Prozessoren)}{T(p\ Prozessoren)}$.

Für die Speedup-Kurven auf Zampano war ein Fit leider nicht sinnvoll. Dies liegt an der bei Amdahl's Gesetz vernachlässigten Kommunikation zwischen den Prozessoren. Auf Zampano nimmt die Kommunikation, abhängig von der Eigenwertproblemlösungs-Routine, eine beachtliche Zeit ein.

Um dies genauer zu untersuchen wurde VAMPIRTRACE eingesetzt. VAMPIRTRACE gibt für die Aufzeichnung der Kommunikationszeiten während der Eigenwertberechnung des PDSYEVX, hier bei den Matrixdimensionen $N = 2000$ und $N = 3000$ beispielhaft gezeigt, folgende Zeiten für MPI und CPU aus:

N	Prozessoren	MPI-Zeit	CPU-Zeit
2000	4 (2×2)	6.693	59.293
2000	8 (2×4)	40.573	48.873
2000	12 (3×4)	16.068	30.027
3000	4 (2×2)	16.416	192.919
3000	8 (2×4)	120.10	158.449
3000	12 (3×4)	47.405	101.152

Tabelle 11: Kommunikationszeiten bei PDSYEVX (Zampano)

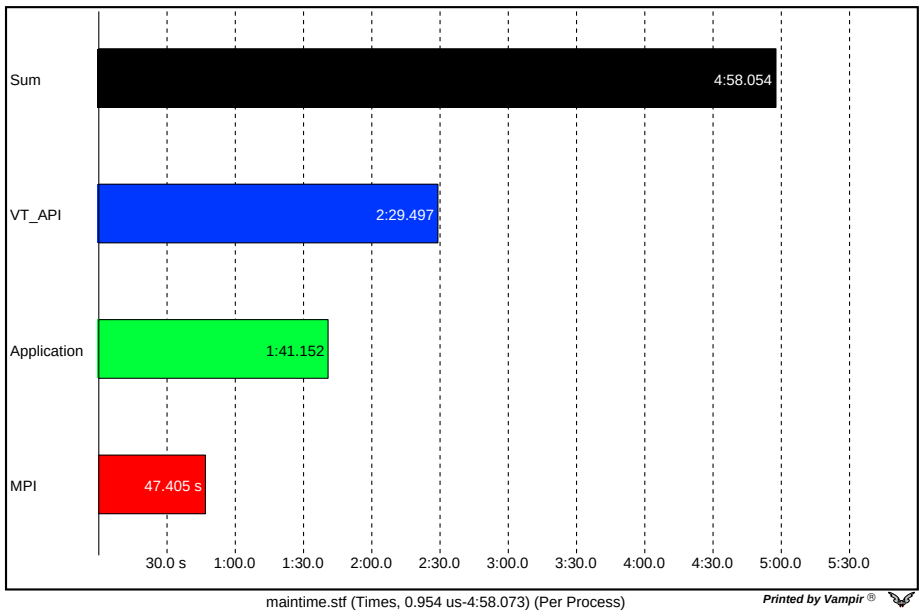

Abbildung 23: Plot zu VAMPIRTRACE, PDSYEVX, N=3000, P=12 (Zampano)

Lässt man die MPI-Zeiten nun einmal außen vor, so lassen sich die reinen CPU-Zeiten fitten. Der Fit der CPU-Zeiten ergibt die in Tabelle 12 aufgelisteten Werte.

Matrixdimension N	Serieller Anteil a	max. Speedup
2000	0.124562	8.0281357
3000	0.124708	8.0187317

Tabelle 12: Serieller Anteil am Beispiel von PDSYEVX (Zampano)

Der maximale Speedup berechnet sich mittels $1/a$.
Bei IBM Regatta ist für alle drei Eigenwertproblemlöser ein Fit des Speedups nach der Formel (5.4) durchführbar. Die Kommunikation ist auf IBM Regatta auf einen einzelnen Knoten beschränkt, so dass bei den Messungen keine starken Schwankungen entstehen. Der serielle Anteil ist insgesamt recht klein.
Abbildung 24 zeigt den graphischen Verlauf der Fit-Kurven. Die expliziten Werte für den seriellen Anteil a sind Tabelle 13 zu entnehmen.

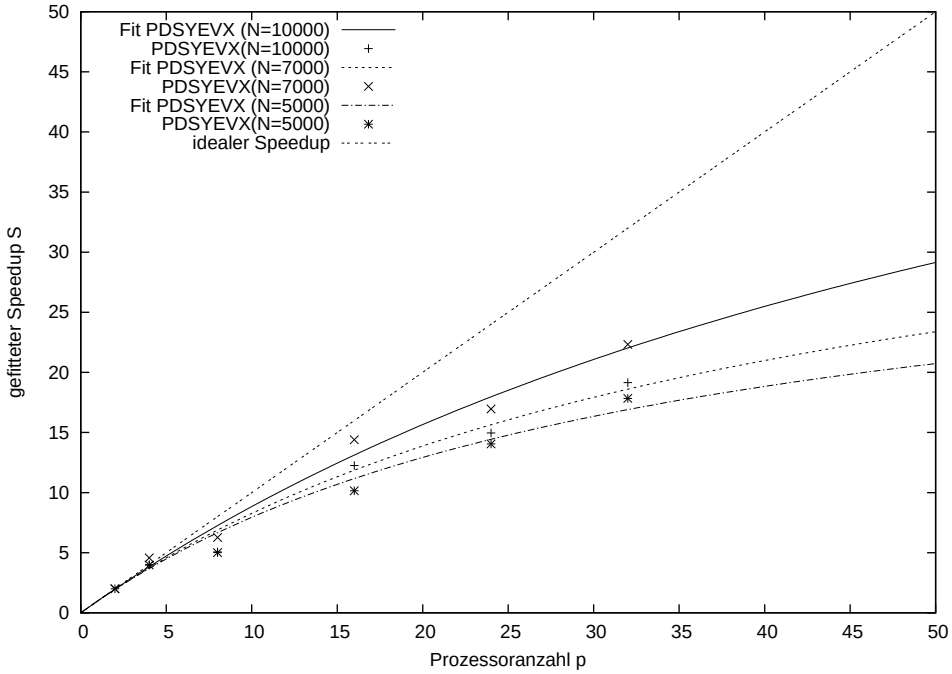

Abbildung 24: Gnu-Fit des Speedups zur Bestimmung des seriellen Anteils (IBM Regatta)

Verfahren	Matrixdimension	Serieller Anteil a	max. Speedup
PDSYEV	10000	0.0160618	93.793
PDSYEV	7000	0.0268118	37.297
PDSYEV	5000	0.032159	31.095
PDSYEVX	10000	0.0146028	68.480
PDSYEVX	7000	0.0232284	43.051
PDSYEVX	5000	0.0288151	34.704
PDSYEVD	10000	0.0127997	78.127
PDSYEVD	7000	0.0199204	50.200
PDSYEVD	5000	0.0247529	40.399

Tabelle 13: Serieller Anteil (IBM Regatta)

5.5 Bewertung der Ergebnisse

Auf IBM Regatta sinkt der der serielle Anteil mit steigender Matrixdimension bei allen drei Eigenwertproblemlösern. Der PDSYEV weist bei $N = 10000$ 1.61% seriellen Anteil auf, der PDSYEVX bei gleicher Matrixdimension 1.46% und der PDSYEVD nur 1.28%. Auf Zampano liegen die Werte für den seriellen Anteil deutlich höher. Bei PDSYEVX beträgt der serielle Anteil bei einer Matrix der Dimension $N = 2000$ oder $N = 3000$ 12.5%.

Kommunikation: Die großen Unterschiede zwischen den Ergebnissen auf den beiden Rechnern Zampano und IBM Regatta liegen an der Kommunikation. Auf IBM Regatta läuft die Kommunikation sehr homogen ab, alle Berechnungen werden auf einem Knoten ausgeführt. Auf Zampano stehen 8 Knoten zur Verfügung, und die Kommunikation innerhalb eines Knotens ist schneller als die Kommunikation zwischen den Knoten. Außerdem ist auf IBM Regatta die Bandbreite höher und die Latenz geringer, verglichen mit Zampano.
Die Peak-Performance der Rechner ist unterschiedlich, was bei gleichem β zu größeren α auf Zampano führen sollte.
Die Speicherzugriffe sind unterschiedlich schnell im Vergleich zu den Rechenoperationen. Hinzu kommt, dass andere BLAS verwendet werden, so dass die Peak-Performance nur unzureichend erreicht werden kann.

6 Parallele Eigenwertlöser - Performance Modell

6.1 Zielsetzung

Ziel ist es, ein Modell zu finden, welches die Performance eines Eigenwertproblemlösers in Abhängigkeit von Rechnertyp, Methode, Matrixgröße und Parametern der vorhergegangenen Kapitel bestimmen kann.

6.2 Performance-Modell

Das Performance-Modell der seriellen Performance

$$t = \alpha \cdot N^{\beta} \tag{6.1}$$

lässt sich mit Amdahl's Gesetz

$$T(p,N) = T(1,N) \cdot \frac{1-a(N)}{p} + T(1,N) \cdot a(N) \tag{6.2}$$

recht einfach zusammensetzen. Da t aus dem Modell der seriellen Performance der Zeit $T(1,N)$ aus Amdahl's Gesetz entspricht, entsteht dabei folgende Gleichung:

$$T(p,N) = \alpha \cdot N^{\beta} \cdot \frac{1-a(N)}{p} + \alpha \cdot N^{\beta} \cdot a(N) \tag{6.3}$$

Die Parameter, die in diesem Modell variieren können lauten:

- α ist ein methoden- und maschinenabhängiger Parameter, der den Vorfaktor von N^{β} bei dem Modell der seriellen Performance bildet.
- β ist methodenspezifischer Parameter, der die formale maschinenunabhängige Skalierung der Methode als Funktion der Problemgröße darstellt. Da die Komplexität aller Verfahren insgesamt -wie zu Beginn in der Theorie beschrieben- $O(N^3)$ beträgt, wird β in der Größenordnung von 3 erwartet.
- N ist die Größe der Matrix, deren Eigenwerte und Eigenvektoren berechnet werden sollen.
- p ist Anzahl der Prozessoren, die bei der parallelen Berechnung eingesetzt werden.
- $a(N)$ stellt den Teil eines Programms dar, der nur seriell bearbeitet werden kann. Es handelt sich um eine methodenabhängige Funktion der Matrixdimension.

Um den seriellen Anteil zu bestimmen, wurde a methodenabhängig und als eine Funktion von N betrachtet:

$$a(N) = \epsilon_0 \cdot e^{-\epsilon_1 \cdot N} \tag{6.4}$$

Durch einen Fit für ϵ_0 und ϵ_1 anhand der Funktion (6.4) mittels GNUPLOT ergibt sich für die drei Eigenwertproblemlöser auf IBM Regatta:

Verfahren	Funktion $a(N)$
PDSYEVX	$a(N) = 0.058571327 \cdot e^{-0.00013742 \cdot N}$
PDSYEVD	$a(N) = 0.049057608 \cdot e^{-0.000133131 \cdot N}$
PDSYEV	$a(N) = 0.111211665 \cdot e^{-0.000227641 \cdot N}$

Tabelle 14: Gleichungen für $a(N)$ (IBM Regatta)

6.3 Test des Modells

Es wurde getestet, welche Zeiten das Modell für $T(p)$ aus (6.3) für Matrizen von bisher nicht bearbeiteter Dimension vorschlägt (=erwartete Zeiten). Die Messungen wurden auf IBM Regatta ausgeführt, da dort der Parameter a für den seriellen Anteil gefittet werden konnte. Als Eigenwertproblemlöser wurde der PDSYEVD gewählt, da er auf IBM Regatta am besten skalierte. Die Ergebnisse sind in Tabelle 15 aufgelistet (alle Zeiten in Sekunden).

PDSYEVD		erwartete Zeit	gemessene Zeit
N=4000	p=4	29.4492742	27.863190
N=4000	p=8	16.28615029	17.440222
N=8000	p=8	110.8141101	126.962120
N=8000	p=16	62.10944218	74.898376
N=11000	p=16	144.5916153	243.639451
N=11000	p=32	83.50834947	128.022976

Tabelle 15: Ergebnisse der Testmessungen, PDSYEVD (IBM Regatta)

PDSYEVX		erwartete Zeit	gemessene Zeit
N=4000	p=4	35.44574905	35.856804
N=4000	p=8	19.85861443	29.531244
N=8000	p=8	254.2752334	279.332031

Tabelle 16: Ergebnisse der Testmessungen, PDSYEVX (IBM Regatta)

6.4 Bewertung der Ergebnisse

Die anhand der entwickelten Funktion $T(p)$ berechneten Zeiten für die Berechnung der Eigenwerte und Eigenvektoren mittels PDSYEVD auf IBM Regatta bei Matrizen der Dimensionen $N = 4000$ und $N = 8000$ trifft die tatsächlich gemessenen Werte gut. Bei einer Matrix der Dimension $N = 11000$ werden die Rechenzeiten um etwa ein Drittel niedriger als die tatsächlich gemessenen Zeiten erwartet.
Die erwarteten Zeiten für PDSYEVX treffen auch gut zu. Auch hier steigen die Abweichungen der berechneten erwarteten Rechenzeit im Vergleich zur gemessenen Zeit mit der Matrixdimension und Prozessoranzahl an.
Diese Abweichungen sind darauf zurück zu führen, dass zur Berechnung der erwarteten Zeiten zwar die Funktion (6.3) verwendet wird, in diesem Modell jedoch die Ergebnisse der seriellen Untersuchungen mit den der parallelen Messungen vermischt werden. Um beispielsweise $T(1, 4000)$ zu berechnen, wird $\alpha \cdot N^{\beta}$ aus dem seriellen Modell von DSYEVD berechnet. Dann stimmt jedoch der Fit von $a(N)$ nicht mehr, denn es kann nicht davon ausgegangen werden, dass $S(2) = 2$ für $N = 5000$ beziehungsweise $S(4) = 4$ für $N = 10000$.
Es ist daher eingeschränkt möglich, mit den untersuchten Modellen für serielle und parallele Verfahren ein gemeinsames Modell zu erstellen.

7 Zusammenfassung

Das erste Ziel dieser Diplomarbeit war, die serielle Performance von vier verschiedenen Eigenwertproblemlösern auf zwei Rechnern unterschiedlicher Charakteristik zu untersuchen, und im Hinblick auf eine Modellfunktion zur Performance von seriellen Eigenwertproblemlösern zu parametrisieren und die Parameter anhand von Messungen zu testen. Die Parameter des Modells waren dabei die Dimension der Matrix, deren Eigenwerte und Eigenvektoren berechnet werden sollten, die Eigenwertproblemlöser und die Charakteristiken der Rechner, auf denen die Messungen durchgeführt wurden.
Das Ergebnis bei den Untersuchungen lieferte Werte für die Parameter α und β der Modellfunktion, welche Aussagen zu den Kosten des Verfahrens, abhängig von der Dimension der Matrix zuließen. Die kürzesten Rechenzeiten erhielten die Eigenwertproblemlöser in Abhängigkeit von der Matrixdimension in folgender Reihenfolge: Auf Zampano erhielt der DSYEVR über den gesamten auf diesem Rechner untersuchten Bereich der Matrixdimensionen die kürzesten Rechenzeiten. Für Matrizen der Dimension von etwa $N \leq 178$ ist der DSYEVX der schnellste, Auf IBM Regatta erzielte der DSYEVD bis zu einer Matrixdimension von etwa $N = 1121$ die besten Zeiten zur Eigenwertproblemlösung. Für Matrizen, die eine Matrix größerer Dimension haben ist der DSYEVR der schnellste.

Das zweite Ziel war, die Untersuchungen der parallelen Performance der drei Eigenwertproblemlöser PDSYEV, PDSYEVX und PDSYEVD auf den beiden Rechnern Zampano und IBM Regatta zu beschreiben und die Skalierbarkeit der Peformance der verschiedenen Eigenwertproblemlöser anhand von *Amdahl's Gesetz* zu parametrisieren.
Hierbei waren die Ergebnisse auf Zampano (Cluster von acht SMP-Knoten) und IBM Regatta (ein SMP-Knoten) sehr unterschiedlich, da die Kommunikation auf IBM Regatta sehr homogen ablaufen kann, alle Berechnungen werden auf einem Knoten ausgeführt, auf Zampano jedoch acht Knoten zur Verfügung stehen und die die Kommunikation innerhalb eines Knotens schneller ist als die Kommunikation zwischen den Knoten, somit die Kommunikation inhomogen abläuft.
Insgesamt wächst der serielle Anteil bei Reduktion der Matrixgröße. Auf IBM Regatta weist der serielle Anteil für PDSYEV bei $N = 10000$ einen Wert von 1.61% auf, PDSYEVX erhält bei gleicher Matrixdimension 1.46% und der PDSYEVD nur 1.28%. Um den seriellen Anteil auf Zampano zu bestimmen, musste der große Anteil an Zeit, die zur Kommunikazion verwendet wurde, abgezogen werden, so dass die reine CPU-Zeit gefittet werden konnte und ein serieller Anteil von 12.5% bei einer Matrixdimension von $N = 2000$ für PDSYEVX erhalten wurde.

Es wurde versucht, die untersuchten Modelle der seriellen und parallelen Verfahren in ein gemeinsam gültiges Modell zusammen zu fassen, so dass die Performance eines Eigenwertproblemlösers in Abhängigkeit von Rechnertyp, Methode, Matrixdimension und Parametern der vorangegangenen Untersuchungen bestimmt werden kann.
Tests des konstruierten Modells ergaben, dass die mit Hilfe des Modells berechneten erwarteten Rechenzeiten, mit methodenabhängigen Abweichungen, annähernd mit den tatsächlich gemessenen Werten übereinstimmen.

A Anhang

A.1 Liste der Bezeichnungen

Name	Bedeutung
A	reelle symmetrische Matrix
E	Einheitsmatrix
Q	Orthonormalmatrix ($Q^{-1} = Q^T$)
X	reguläre Matrix
Λ	Matrix der Eigenwerte
R	obere Dreiecksmatrix
λ_i	Eigenwerte der Matrix
x	Variable für Eigenvektor
q_i	Eigenvektoren der Matrix
q_J	Eigenvektor der Matrix
r(x)	Rayleigh Quotient eines Vektors $x \in \mathbb{R}^m$
v_k	Reflektionsvektoren
c_i	skalarer Faktor
p	Anzahl der Prozessoren
a	Amdahl-Anteil
$\epsilon_{machine}$	Maschinengenauigkeit
$\mathcal{O}$	Operationsanzahl, Größenordnung für den Aufwand
L^{-T}	$= (L^{-1})^T$, transponierte Inverse der Matrix L

A.2 Begriffserklärungen

- **singuläre Matrix:** quadratische Matrix deren Determinante gleich Null ist
- **reguläre Matrix:** quadratische Matrix mit nicht verschwindendender Determinante: $det(A) \neq 0$, nicht singuläre Matrix
- **unitäre Matrix:** Falls $\bar{A}^T = A^{-1}$ mit $\bar{A} = (\bar{a}_{ik})$, wobei $(\bar{a}_{ik})$ die konjugiert Komplexen zu a_{ik} sind.
- **orthogonale Matrix:** Zeilen- sowie Spaltenvektoren einer orthogonalen Matrix bilden je ein Orthonormalsystem, manchmal wird sie mit Orthonormalmatrix bezeichnet
- **nicht reduzierbare Tridiagonalmatrix:** Matrix deren Nebendiagonalelemente alle ungleich Null sind

A.3 Tabellen der Messergebnisse

Alle Zeiten in Sekunden.

A.3.1 Serielle Messergebnisse auf Zampano

N	DSYEV	DSYEVX			DSYEVD	DSYEVR	
		25%	50%	100%		50%	100%
500	13.4826	1.6893	2.425	3.6259	2.8884	2.0444	2.3993
1000	107.3035	12.2310	15.5789	22.5106	20.6157	14.8231	17.5257
1250	205.5397	-	-	-	-	-	-
1500	354.4851	39.3777	48.7074	67.8187	65.8484	46.9518	56.1577
2000	825.9020	91.0207	112.4818	153.7896	152.9740	109.4583	130.4843
2500	1597.1748	174.3786	213.5534	294.6371	294.6957	212.8572	252.5930
2750	2124.3700	-	-	-	-	-	-
3000	2735.6764	301.4664	369.1816	512.0734	505.4644	368.8392	435.9348
3500	4329.2091	478.1415	586.8354	809.5613	803.6923	585.9923	692.1281
4000	6434.8228	727.7269	891.6285	1243.4766	1179.1873	890.6840	1019.4048
4500	9087.5078	1058.8608	1300.4566	1843.1106	1647.9999	1327.8553	1427.4830
5000	12431.6825	1487.2740	1837.2951	2682.5333	2264.2730	1843.0852	1955.5865

Tabelle 17: Serielle Messergebnisse auf Zampano

A.3.2 Serielle Messergebnisse auf IBM Regatta

N	Zeit	FDiv	MFlips	MFlip-Rate
500	0.798	505883	1108.890	1389.790
1000	7.633	1912715	8501.735	1113.808
1500	25.554	4244009	28343.934	1109.162
2000	56.659	7529269	66973.609	1182.056
2500	109.712	11623025	129608.539	1181.351
3000	203.959	16560443	222197.500	1089.425
3500	295.444	22432807	351514.875	1189.784
4000	498.352	29242923	523757.906	1050.980
4500	838.577	36820869	742937.750	885.951
5000	1092.663	45193553	1014858.188	928.794

Tabelle 18: Serielle Messergebnisse für DSYEV (100%)

N	Zeit	FDiv	MFlips	MFlip-Rate
500	0.223	1813635	463.692	2074.852
1000	1.464	7020441	3522.870	2406.792
1500	5.257	15410186	11683.009	2222.224
2000	11.712	25219234	27432.838	2342.267
2500	20.631	42252979	53333.332	2585.150
3000	44.510	59919361	91819.648	2062.877
3500	76.097	80925679	145453.484	1911.431
4000	114.688	98839690	216643.188	1888.979
4500	142.880	133158167	308099.938	2156.360
5000	163.614	153410816	422046.313	2579.523

Tabelle 19: Serielle Messergebnisse für DSYEVR (100%)

N	Zeit	FDiv	MFlips	MFlip-Rate
500	0.260	3667429	621.681	2396.731
1000	2.419	14214230	4705.434	1945.058
1500	6.591	31396709	15582.962	2364.325
2000	13.916	54657589	36326.137	2610.375
2500	26.604	84723059	70511.734	2650.433
3000	45.054	120883011	121251.860	2691.273
3500	69.395	164297513	191817.594	2764.134
4000	108.663	212932934	285411.656	2626.567
4500	152.269	268379828	405733.563	2664.584
5000	230.995	331054682	556173.000	2407.726

Tabelle 20: Serielle Messergebnisse für DSYEVD (100%)

N	Zeit	FDiv	MFlips	MFlip-Rate
500	0.204	3444384	278.664	1368.414
1000	1.182	13471568	2037.475	1723.856
1500	3.428	29916675	6662.833	1943.842
2000	7.283	52718490	15547.901	2134.845
2500	13.380	81722197	30091.539	2248.932
3000	22.129	117211539	52018.160	2350.682
3500	34.605	158815870	82231.078	2376.243
4000	50.176	206572644	122679.031	2444.979
4500	71.095	260350338	174479.328	2454.187
5000	96.405	320355381	240299.906	2492.611

Tabelle 21: Serielle Messergebnisse für DSYEVX (25%)

N	Zeit	FDiv	MFlips	MFlip-Rate
500	0.334	6775551	367.990	1100.700
1000	1.827	26552378	2644.564	1447.272
1500	5.062	59148044	8605.566	1700.0120
2000	11.376	104386165	20048.234	1762.331
2500	19.602	161871605	38804.930	1979.684
3000	31.011	232215752	67062.984	2162.552
3500	50.672	314457992	106611.578	2103.972
4000	74.412	408812401	158963.563	2136.256
4500	110.788	515800046	226488.266	2044.346
5000	153.440	634596711	313300.250	2041.829

Tabelle 22: Serielle Messergebnisse für DSYEVX (50%)

N	Zeit	FDiv	MFlips	MFlip-Rate
500	0.599	13329399	546.545	912.384
1000	3.556	52165416	3861.747	1086.064
1500	8.704	115979581	12488.871	1434.831
2000	17.160	204746892	29011.672	1690.645
2500	30.709	317736764	56129.273	1827.798
3000	51.156	455141487	97053.164	1897.212
3500	88.092	616508817	154866.172	1758.014
4000	141.886	801736733	233885.031	1648.400
4500	194.553	1010845801	336990.625	1732.131
5000	250.518	1243440300	472131.500	1884.623

Tabelle 23: Serielle Messergebnisse für DSYEVX (100%)

A.3.3 Parallele Messergebnisse auf Zampano

```
# Ausgabe der Messwerte
# zum Eigensolver: PDSYEV
# bei fester Matrixgroesse: N=1500
#
# Prozessoranzahl       Zeit (in Sek.)
            1              223.345993
            2              135.091003
            4              68.133003
            8              77.942703
            12              54.887798
            16              41.495098
            20              32.897202
```

Tabelle 24: Parallele Messergebnisse für PDSYEV (100%) bei N=1500

```
# Ausgabe der Messwerte
# zum Eigensolver: PDSYEV
# bei fester Matrixgroesse: N=3000
#
# Prozessoranzahl       Zeit (in Sek.)
            1              1754.352051
            2              1060.375977
            4              532.168518
            8              607.106628
            12              419.264191
            16              307.836487
            20              257.357910
```

Tabelle 25: Parallele Messergebnisse für PDSYEV (100%) bei N=3000

```
# Ausgabe der Messwerte
# zum Eigensolver: PDSYEV
# bei fester Matrixgroesse: N=4500
#
# Prozessoranzahl       Zeit (in Sek.)
            1              6856.724121
            2              4410.296875
            4              2016.929443
            8              2417.310791
            12              1626.000000
            16              1206.806763
            20              986.810303
```

Tabelle 26: Parallele Messergebnisse für PDSYEV (100%) bei N=4500

```
# Ausgabe der Messwerte
# zum Eigensolver: PDSYEVX
# bei fester Matrixgroesse: N=1500
#
# Prozessoranzahl       Zeit (in Sek.)
            1              62.282700
            2              49.211601
            4              28.466999
            8              36.456699
            12              19.362301
            16              11.889000
            20              10.832700
```

Tabelle 27: Parallele Messergebnisse für PDSYEVX (100%) bei N=1500

```
# Ausgabe der Messwerte
# zum Eigensolver: PDSYEVX
# bei fester Matrixgroesse: N=3000
#
# Prozessoranzahl       Zeit (in Sek.)
            1              454.818604
            2              359.214691
            4              201.410797
            8              289.861511
            12              140.647507
            16              87.608902
            20              83.168999
```

Tabelle 28: Parallele Messergebnisse für PDSYEVX (100%) bei N=3000

```
# Ausgabe der Messwerte
# zum Eigensolver: PDSYEVX
# bei fester Matrixgroesse: N=4500
#
# Prozessoranzahl       Zeit (in Sek.)
            1              1467.843140
            2              1161.671143
            4              652.796082
            8              963.528870
            12              458.101501
            16              283.575500
            20              272.481689
```

Tabelle 29: Parallele Messergebnisse für PDSYEVX (100%) bei N=4500

```
# Ausgabe der Messwerte
# zum Eigensolver: PDSYEVD
# bei fester Matrixgroesse: N=1500
#
# Prozessoranzahl       Zeit (in Sek.)
            1              71.637001
            2              50.200001
            4              27.080799
            8              20.741100
            12              15.184500
            16              11.680200
            20              11.691000
```

Tabelle 30: Parallele Messergebnisse für PDSYEVD (100%) bei N=1500

```
# Ausgabe der Messwerte
# zum Eigensolver: PDSYEVD
# bei fester Matrixgroesse: N=3000
#
# Prozessoranzahl       Zeit (in Sek.)
            1              568.984009
            2              397.016998
            4              190.266296
            8              145.253799
            12              110.876099
            16              78.599602
            20              71.372803
```

Tabelle 31: Parallele Messergebnisse für PDSYEVD (100%) bei N=3000

```
# Ausgabe der Messwerte
# zum Eigensolver: PDSYEVD
# bei fester Matrixgroesse: N=4500
#
# Prozessoranzahl       Zeit (in Sek.)
            1              1791.557007
            2              1230.180054
            4              609.511780
            8              468.970001
            12              335.707214
            16              231.621201
            20              212.746796
```

Tabelle 32: Parallele Messergebnisse für PDSYEVD (100%) bei N=4500

A.3.4 Parallele Messergebnisse auf IBM Regatta

N	Prozessoranzahl	Zeit
14000	32	234.206635
14000	24	287.296448
14000	16	390.991028
14000	8	692.330750

Tabelle 33: Parallele Messergebnisse für PDSYEVD (100%) bei N=14000

N	Prozessoranzahl	Zeit
12000	32	140.313126
12000	24	172.233994
12000	16	235.592316
12000	8	402.469849

Tabelle 34: Parallele Messergebnisse für PDSYEVD (100%) bei N=12000

N	Prozessoranzahl	Zeit
10000	32	79.127159
10000	24	97.063736
10000	16	138.099945
10000	8	240.196854
10000	4	455.103027
10000	2	880.732788

Tabelle 35: Parallele Messergebnisse für PDSYEVD (100%) bei N=10000

N	Prozessoranzahl	Zeit
7000	32	28.571255
7000	24	34.807629
7000	16	51.184460
7000	8	74.652229
7000	4	142.470413
7000	2	287.020721

Tabelle 36: Parallele Messergebnisse für PDSYEVD (100%) bei N=7000

N	Prozessoranzahl	Zeit
5000	32	11.862277
5000	24	14.426264
5000	16	20.808386
5000	8	30.046595
5000	4	54.931942
5000	2	97.199859

Tabelle 37: Parallele Messergebnisse für PDSYEVD (100%) bei N=5000

N	Prozessoranzahl	Zeit
10000	32	89.973381
10000	24	115.122070
10000	16	140.646317
10000	8	340.024658
10000	4	430.639832
10000	2	880.732788

Tabelle 38: Parallele Messergebnisse für PDSYEVX (100%) bei N=10000

N	Prozessoranzahl	Zeit
7000	32	31.928726
7000	24	42.031597
7000	16	49.526627
7000	8	123.833023
7000	4	156.017365
7000	2	356.501282

Tabelle 39: Parallele Messergebnisse für PDSYEVX (100%) bei N=7000

N	Prozessoranzahl	Zeit
5000	32	13.624303
5000	24	17.291407
5000	16	19.929094
5000	8	48.449631
5000	4	61.035309
5000	2	121.460281

Tabelle 40: Parallele Messergebnisse für PDSYEVX (100%) bei N=5000

N	Prozessoranzahl	Zeit
10000	32	410.291260
10000	24	459.287323
10000	16	582.266785

Tabelle 41: Parallele Messergebnisse für PDSYEV (100%) bei N=10000

N	Prozessoranzahl	Zeit
7000	32	125.598442
7000	24	137.831528
7000	16	181.124084
7000	8	278.226196
7000	4	484.970947

Tabelle 42: Parallele Messergebnisse für PDSYEV (100%) bei N=7000

N	Prozessoranzahl	Zeit
5000	32	43.722858
5000	24	48.771183
5000	16	65.223457
5000	8	95.894905
5000	4	172.313751

Tabelle 43: Parallele Messergebnisse für PDSYEV (100%) bei N=5000

Literatur

[1] F.L. Pilar
Elementary Quantum Chemistry
Dover, New York 2001

[2] C.C.J. Roothaan
Rev. Mod. Phys. Vol. 32, p 179, 1960

[3] E. Anderson, Z. Bai und andere
LAPACK User's Guide,
SIAM Philadelphia, 1999
ebenso im Internet unter:
http://www.netlib.org/lapack/lug/lapack_lug.html

[4] L.S.Blackford, J.Choi, A.Cleary und andere
ScaLAPACK Users' Guide
SIAM Philadelphia, 1997
ebenso im Internet unter:
http://www.netlib.org/scalapack/slug/scalapack_slug.html

[5] M. Hanke-Bourgeois
Grundlagen der Numerischen Mathematik und des wissenschaftlichen Rechnens
Verlag Teubner

[6] W.Gellert, H.Köstner, Dr.S.Neuber
Fachlexikon Mathematik
Verlag harri deutsch
Thun und FrankfurtMain

[7] Lloyd N. Trefethen, David Bau, III
Numerical Linear Algebra
SIAM Philadelphia

[8] Inderjit Singh Dhillon
A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem
University of California, Berkeley, 1997

[9] A.Greenbaum, J.Dongarra
Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem
UT, CS-89-92, November 1989
http://www.netlib.org/lapack/lawns

[10] Zampano
http://http://zampano.zam.kfa-juelich.de/

[11] Bruno Lang
Direct Solvers for symmetric Eigenvalue Problems
Aachen University of Technology
Computing Center

[12] Myrinet
http://www.myri.com/myrinet/overview/index.html

[13] GM
http://www.myri.com/scs/

[14] MPI (Message Passing Interface)
http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[15] RTC-Real Time Clock
http://zampano/software/libzam.htm#rtc

[16] PCL-Performance Counter Library
http://www.fz-juelich.de/zam/PCL/doc/pcl/pcl.html

[17] Vampirtrace 2.0
Installation and User's Guide
Pallas GmbH
http://www.pallas.com

[18] IBM Regatta
http://juppdoc.fz-juelich.de/

[19] ESSL, PESSL
http://www-1.ibm.com/servers/eserver/pseries/library/sp_books/essl.html

[20] HPMCOUNT
http://juppdoc.fz-juelich.de/ibmsc/software/HPM_2_4_4.html#Header_hpmcount

[21] Inge Gutheil
Comparison of Some Parallel Solvers for the Real Full Symmetric Eigenproblem on CRAY T3E

[22] M.Busch, Th.Williams, C.Kelley
GNUPLOT - An Interactive Plotting Program
FZJ-ZAM-BHB-0114

[23] Jack J.Dongarra, R.Clint Whaley
A User's Guide to the BLACS v1.1,
http://www.netlib.org/blacs/lawn94.ps

[24] Stoer, Bulirsch
Numerische Mathematik I und II
Springer

[25] Deuflhard, Hohmann
Numerische Mathematik
Walter de Gruyter

[26] Martin Reißel
Skript Numerik I

[27] Theo Ungerer
Parallelrechner und parallele Programmierung
Spektrum-Verlag
Heidelberg 1997