FORSCHUNGSZENTRUM JÜLICH GmbH
Jülich Supercomputing Centre
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Performance-Analyse und -Optimierung paralleler Eigenwertlöser auf Blue Gene-Architekturen

Tommy Berg

FZJ-JSC-IB-2011-04

August 2011

(letzte Änderung: 23. Aug. 2011)

Zusammenfassung

Viele Anwendungen aus den Natur- und Ingenieurwissenschaften erfordern die Berechnung der Eigenwerte und -vektoren von symmetrischen, dichtbesetzten Matrizen. Aufgrund der Größe der auftretenden Matrizen ist häufig eine parallele Berechnung der Eigenwerte und -vektoren unumgänglich. In dieser Arbeit werden verschiedene Routinen zur parallelen Lösung symmetrischer Eigenwertprobleme aus den Bibliotheken ScaLAPACK und Elemental mit ihren jeweils genutzten Algorithmen vorgestellt. Für die Verteilung der Matrizen auf die Prozessoren verwenden beide Bibliotheken unterschiedliche Ansätze. Während ScaLAPACK eine zweidimensionale blockzyklische Verteilung der Matrizen nutzt, verfolgt die noch in der Entwicklung befindliche Bibliothek Elemental einen elementweisen zweidimensionalen zyklischen Ansatz zur Verteilung der Matrizen. Parameter wie die algorithmische Blockgröße beeinflussen die Performance der Eigenwertlöser. Es werden daher zunächst für die Rechnerarchitekturen Blue Gene/P und Blue Gene/Q (Prototyp) die optimalen Parameterwerte bestimmt. Die Routinen zur Eigenwertberechnung werden dann auf den beiden Blue Gene-Systemen getestet und bezüglich ihres Laufzeit- und Skalierungsverhaltens sowohl untereinander, als auch hinsichtlich der verschiedenen Architekturen verglichen.

Abstract

Many applications in the natural sciences and engineering require to compute eigenvalues and vectors of dense symmetric matrices. Due to the size of matrices a parallel computation of the eigenvalues and vectors is essential. In this thesis different routines of the ScaLAPACK and Elemental libraries for parallel solution of the symmetric eigenvalue problem and their respective algorithms are presented. Both libraries use different approaches for distribution of the matrices to processors. While ScaLAPACK uses a two-dimensional block-cyclic distribution of matrices, the Elemental library, which is still under development, pursues a two-dimensional element-wise cyclic approach for distribution. Parameters such as the algorithmic block size affect the performance of the eigenvalue solvers. Therefore the optimal parameter values for the computer architectures Blue Gene/P and Blue Gene/Q (prototype) are first determined. The routines for the eigenvalue computation are tested on both Blue Gene systems to compare the runtime and scaling behavior of the different routines on one architecture and of the same routine on different architectures.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In Naturwissenschaft und Technik ist die numerische Simulation ein wichtiges Werkzeug, um komplexe Vorgänge zu verstehen. Simulationen werden häufig als kostengünstigere Alternativen zu Experimenten oder als zusätzliche Methode zur Interpretation der Experimentergebnisse genutzt. Mathematische Modelle in Form von Diffentialgleichungen, welche von Theoretikern durch Beobachtung der Realität entwickelt werden und für unendlich viele Punkte in Raum und Zeit gültig sind, sollten die Realität so gut wie möglich beschreiben. Numerische Simulationen, welche die Realität nachbilden, nutzen diese Modelle, können jedoch nur endlich viele Punkte betrachten, weshalb die mathematischen Modelle diskretisiert werden müssen. Je mehr diskrete Punkte betrachtet werden, desto besser wird die Realität abgebildet, allerdings steigt der Rechenaufwand dadurch erheblich.
Beispiele aus der Naturwissenschaft finden sich in allen Bereichen, z. B. die Simulation einer Bakterienvermehrung aus der Biologie oder Simulationen für Studien über die Entstehung des Universums aus der Physik. Anwendungen der numerischen Simulation in der Technik finden sich unter anderem in der Fahrzeugentwicklung, in der z. B. die Finite-Elemente-Methode für virtuelle Crashtests genutzt wird.
Als wesentlicher Bestandteil vieler Simulationsprogramme müssen Eigenwertprobleme gelöst werden. In vielen Anwendungen, wie der Finite-Elemente-Methode, treten dünnbesetzte Matrizen auf, deren Eigenwerte bestimmt werden müssen. Diese spezielle Struktur der Matrizen kann von effizienten Algorithmen, wie dem Jacobi-Davidson-Verfahren, ausgenutzt werden.
Im Institute for Advanced Simulation (IAS) des Forschungszentrum Jülichs wird die Dichtefunktional-Theorie (DFT) genutzt, um den quantenmechanischen Grundzustand eines Vielelektronen-Systems z. B. in Atom- oder Molekül-Systemen zu bestimmen. In der DFT müssen typischerweise 10 - 30 % der Eigenwerte und -vektoren von großen, dichtbesetzten Matrizen berechnet werden, was häufig einen *Bottleneck* darstellt [1]. Die Matrixdimension hängt von der Größe des Elektronen-Systems ab und kann sehr groß werden. Für viele Anwendungen im IAS sind Matrixdimensionen ab 50.000 von Interesse. Da die hier auftretenden dichtbesetzten Matrizen einen wesentlich größeren Rechenaufwand als dünnbesetzte Matrizen verursachen, ist eine parallele Berechnung dieser Eigenwertprobleme unabdingbar. Eine effizien-

te Nutzung auf einem hochparallelen Rechner setzt allerdings auch hochskalierende Algorithmen voraus. Dazu existieren Bibliotheken, welche die parallele Lösung von Standardproblemen der Linearen Algebra wie dem Eigenwertproblem ermöglichen. Diese Arbeit beschäftigt sich mit zwei solcher Bibliotheken und den darin implementierten parallelen Eigenwertlösern. Diese werden auf Systemen mit verteiltem Speicher genutzt. Für diese Arbeit werden die Löser auf zwei Systemen der Blue Gene Reihe untersucht, dem in Jülich unter dem Namen JUGENE installierten Blue Gene/P-Rechner und einem Prototyp der Blue Gene/Q Architektur. Die Eigenwertlöser werden auf ihre Performance und ihr Skalierungsverhalten untersucht und verglichen. Die Performance der Routinen wird durch Anpassung der Blockgrößen bei der Datenverteilung auf die Prozessoren und bei der geblockten Berechnung optimiert.
Die mathematische Beschreibung des Eigenwertproblems findet sich in Kapitel 2. Dort werden auch die Grundlagen zur Lösung des Eigenwertproblems erläutert.
In Kapitel 3 werden die Algorithmen erklärt, welche von den getesteten Routinen genutzt werden. Alle Algorithmen zur numerischen Lösung werden hier als sequentielle Algorithmen vorgestellt und im Hinblick auf ihre theoretische Komplexität bewertet.
Die beiden genutzten Rechnerarchitekturen und die Bibliotheken werden in Kapitel 4 vorgestellt, wobei insbesondere auf die Unterschiede in der Datenaufteilung für die Systeme mit verteiltem Speicher eingegangen wird.
Das anschließende Kapitel 5 beschäftigt sich mit der Analyse der Parameter, mit dem Ziel, die für die Performance optimalen Einstellungen zu identifizieren, welche für die nachfolgend vorgestellten Messungen genutzt werden.
Die Ergebnisse der Messreihe werden schließlich in Kapitel 6 vorgestellt und im Hinblick auf Laufzeit und Skalierungsverhalten verglichen.
Das letzte Kapitel gibt eine Schlussbetrachtung mit einer Zusammenfassung und Diskussion der vorgestellten Ergebnisse und schließt mit einem Ausblick auf zukünftige Untersuchungen.

Kapitel 2

Mathematische Grundlagen

In diesem Kapitel werden die benötigten mathematischen Definitionen eingeführt und eine Zusammenfassung einiger für diese Arbeit wichtiger Begriffe der Linearen Algebra gegeben. Diese Definitionen und Sätze findet man z. B. in [2] oder [3].

2.1 Grundlegende Definitionen

In der Linearen Algebra ist es üblich mit Matrizen und Vektoren zu arbeiten. So kann man zum Beispiel ein Lineares Gleichungssystem

$$\begin{array}{ccccccccc} a_{1,1}x_1 & + & a_{1,2}x_2 & + & \cdots & + & a_{1,n}x_n & = & b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & + & \cdots & + & a_{2,n}x_n & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{m,1}x_1 & + & a_{m,2}x_2 & + & \cdots & + & a_{m,n}x_n & = & b_m \end{array}$$

wesentlich kompakter mit Hilfe von $Ax = b$ darstellen, wobei das zweidimensionale Anordnungssytem A als $(m \times n)$-Matrix und x bzw. b als n- bzw. m-dimensionale Vektoren bezeichnet werden.
Handelt es sich bei den Matrix- bzw. Vektorelementen um reelle Werte schreibt man $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$ und $b \in \mathbb{R}^m$.
Die Addition von Matrizen und Vektoren erfolgt komponentenweise. Eine Multiplikation von Vektoren ist durch das Skalarprodukt gegeben.

Definition 2.1.1. Seien $u, v \in \mathbb{C}^n$ zwei komplexe Spaltenvektoren, dann wird

$$< u, v > := \sum_{j=1}^{n} \bar{u}_j v_j \quad \in \mathbb{C} \tag{2.1}$$

als **Skalarprodukt** der beiden Vektoren bezeichnet. Dabei ist $\bar{u}_j$ der **konjugiert komplexe Wert** von u_j, d. h. für $u_j = a + bi$ gilt $\bar{u}_j = a - bi$ mit $a, b \in \mathbb{R}$.

Bemerkung 2.1.2. Gilt für das Skalarprodukt zweier Vektoren $u, v \in \mathbb{C}^n$

$$< u, v > = 0,$$

so bezeichnet man sie als **orthogonal** zueinander und schreibt $u \perp v$. Sind $u, v \in \mathbb{R}^n$, so stehen sie im euklidischen Raum senkrecht aufeinander.

Definition 2.1.3. Sei $x \in \mathbb{C}^n$ ein Vektor, dann ist durch

$$\|x\|_2 := \sqrt{< x, x >} = \sqrt{\sum_{j=1}^{n} \bar{x}_j x_j}$$

die **euklidische Norm** definiert. Sie beschreibt für $x \in \mathbb{R}^n$ die Länge des Vektors x im euklidischen Raum.
Im Folgenden ist bei einer Norm $\|\cdot\|$ immer die euklidische Norm $\|\cdot\|_2$ gemeint, falls sie nicht genauer bezeichnet wird.

Definition 2.1.4. Eine Matrix $B \in \mathbb{C}^{m \times n}$ heißt **konjugiert komplex transponierte** oder auch **adjungierte** Matrix von $A \in \mathbb{C}^{n \times m}$, falls für alle Elemente der Matrizen

$$b_{j,i} = \bar{a}_{i,j}$$

gilt. Man schreibt die Matrix B dann als $\bar{A}^{\mathsf{T}}$ oder A^*. Bei einer reellen Matrix spricht man von der **transponierten** Matrix A^{T} von A.

Definition 2.1.5. Seien $A \in \mathbb{C}^{m \times k}$ und $B \in \mathbb{C}^{k \times n}$ zwei komplexe Matrizen, dann wird das Produkt $AB = C \in \mathbb{C}^{m \times n}$ über die **Matrix-Matrix-Multiplikation** berechnet. Für jedes Element der Produktmatrix C gilt:

$$\bigvee_{\substack{1 \le i \le m \\ 1 \le j \le n}} : c_{i,j} = \sum_{l=1}^{k} a_{i,l} b_{l,j}$$

Bemerkung 2.1.6. Das Skalarprodukt zweier Vektoren, wie in (2.1) definiert wurde, lässt sich auch über eine Matrizenmultiplikation zwischen den Vektoren darstellen, wobei der vordere Vektor adjungiert werden muss.

$$< u, v > = u^* v$$

Definition 2.1.7. Eine Matrix $A \in \mathbb{C}^{n\times n}$ heißt **reduzibel** oder **zerfallend**, falls eine Permutationsmatrix $P \in \mathbb{C}^{n\times n}$, also eine Einheitsmatrix mit permutierten Spalten, existiert, so dass

$$PAP^* = \begin{bmatrix} B & C \\ 0 & D \end{bmatrix}$$

gilt, wobei B und D quadratische Matrizen sind. Die Matrix A heißt **irreduzibel** oder **nicht zerfallend**, falls sie nicht reduzibel ist.

Definition 2.1.8. Eine quadratische Matrix $A \in \mathbb{C}^{n\times n}$ heißt **hermitesch**, falls $A = A^*$ gilt. Im reellen Fall, also $A = A^{\mathsf{T}} \in \mathbb{R}^{n\times n}$, bezeichnet man sie als **symmetrische** Matrix.

Definition 2.1.9. Eine Matrix $A \in \mathbb{C}^{n\times n}$ heißt **regulär** oder **invertierbar**, falls eine Matrix $A^{-1} \in \mathbb{C}^{n\times n}$ existiert, sodass

$$AA^{-1} = I$$

gilt, wobei $I \in \mathbb{R}^{n\times n}$ die Einheitsmatrix darstellt. Die Matrix A^{-1} wird als **Inverse** Matrix von A bezeichnet.

Definition 2.1.10. Gilt für eine Matrix $A \in \mathbb{C}^{n\times n}$ die Gleichung

$$A^*A = I,$$

wobei I die Einheitsmatrix ist, so wird die Matrix A als **unitäre** Matrix bezeichnet. Dann gilt für die Inverse $A^{-1} = A^*$. Die Spalten sind bei solchen Matrizen alle orthogonal zueinander und haben die Norm 1. Sie bilden ein Orthogonalsystem.
Im reellen Fall, also wenn

$$A^{-1} = A^{\mathsf{T}}$$

für $A \in \mathbb{R}^{n\times n}$ gilt, wird A als **Orthogonal**- oder **Orthonormalmatrix** bezeichnet.

Definition 2.1.11. Zwei Matrizen $A, \tilde{A} \in \mathbb{C}^{n\times n}$ heißen **ähnlich** zueinander falls

$$\tilde{A} = C^{-1}AC \tag{2.2}$$

gilt, wobei $C \in \mathbb{C}^{n\times n}$ eine beliebige reguläre Transformationsmatrix ist. Die Transformation (2.2) der Matrix A wird als **Ähnlichkeitstransformation** bezeichnet.

Bemerkung 2.1.12. Wenn es sich um eine **orthogonale Ähnlichkeitstransformation**

$$\tilde{A} = Q^* A Q \tag{2.3}$$

handelt, also die Transformationsmatrix $Q \in \mathbb{C}^{n\times n}$ unitär ist, so bleibt für eine hermitesche Matrix $A \in \mathbb{C}^{n\times n}$ die dazu ähnliche Matrix $\tilde{A} \in \mathbb{C}^{n\times n}$ hermitesch:

$$\tilde{A}^* \;=\; (Q^* A Q)^* \;=\; Q^* A^* Q^{*^*} \;=\; Q^* A Q \;=\; \tilde{A}$$

2.2 Das Eigenwertproblem

Definition 2.2.1. Es sei eine quadratische Matrix $A \in \mathbb{C}^{n\times n}$ gegeben. Dann bezeichnet man das Auffinden einer Lösung des Gleichungssystems

$$Av = \lambda v, \quad \|v\| = 1 \tag{2.4}$$

als **Eigenwertproblem**. Dabei wird der Skalar $\lambda \in \mathbb{C}$ als **Eigenwert** und der Vektor $v \in \mathbb{C}^n$ als zugehöriger **Eigenvektor** bezeichnet. Zusammen nennt man Eigenwert λ und Eigenvektor v ein **Eigenpaar** (λ, v).

Definition 2.2.2. Die Gleichung (2.4) ist äquivalent zu

$$Av - \lambda v = 0 \Leftrightarrow (A - \lambda I)v = 0,$$

und damit ein homogenes Gleichungssystem. Falls die Determinante der Koeffizientenmatrix des Gleichungssystems ungleich 0 ist, hat dieses nur die triviale Lösung $v = \vec{0}$. Ist dies nicht der Fall, also

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

so existieren unendlich viele Lösungen für v in Abhängigkeit von λ. Daher sind die Nullstellen des **charakteristischen Polynoms** der Matrix A, wie $\chi_A(\lambda)$ bezeichnet wird, die Eigenwerte der Matrix A.

In vielen Anwendungen treten Matrizen A auf, welche hermitesch bzw. symmetrisch sind. Durch diese Eigenschaft lassen sich effizientere Algorithmen zur Berechnung von Eigenpaaren anwenden. Dabei werden die beiden folgenden Sätze angewendet.

Satz 2.2.3. *Eine hermitesche Matrix besitzt ausschließlich reelle Eigenwerte.*

Beweis. Es seien Eigenwert $\lambda \in \mathbb{C}$ und der zugehörige Eigenvektor $v \in \mathbb{C}^n$ mit $\|v\| = 1$ der hermiteschen Matrix $A \in \mathbb{C}^{n\times n}$ gegeben. Da der Eigenvektor v die Norm 1 besitzt, gilt für das Skalarprodukt $< v, v > = v^*v = 1^2 = 1$. Daher folgt

$$\lambda = v^*\lambda v = v^*Av = v^*A^*v = (Av)^*v = (\lambda v)^*v = v^*\bar{\lambda}v = \bar{\lambda}$$

und insgesamt gilt $\lambda = \bar{\lambda}$. Daraus folgt, dass $\lambda \in \mathbb{R}$, also reell, ist.

□

Satz 2.2.4. *Die Eigenvektoren zu verschiedenen Eigenwerten einer hermiteschen Matrix sind linear unabhängig und orthogonal zueinander und bilden daher ein Orthogonalsystem.*

Beweis. Es seien zwei beliebig gewählte verschiedene Eigenwerte $\lambda_1, \lambda_2 \in \mathbb{R}$, $\lambda_1 \neq \lambda_2$ der hermiteschen Matrix $A \in \mathbb{C}^{n\times n}$ gegeben. Die jeweils dazu gehörenden Eigenvektoren seien $v_1, v_2 \in \mathbb{C}^n$. Für das Skalarprodukt gilt

$$\begin{array}{ccccccccc} & \lambda_2 < v_1, v_2 > & = & \lambda_2(v_1^*v_2) & = & v_1^*\lambda_2 v_2 & = & v_1^*Av_2 \\ = & (A^*v_1)^*v_2 & = & (Av_1)^*v_2 & = & (\lambda_1 v_1)^*v_2 & = & v_1^*\bar{\lambda}_1 v_2 \\ = & \bar{\lambda_1}(v_1^*v_2) & = & \bar{\lambda_1} < v_1, v_2 > & & & & \end{array}$$

Wie in Satz 2.2.3 schon gezeigt wurde, ist der Eigenwert λ_1 reell, woraus schließlich $\lambda_2 < v_1, v_2 > = \lambda_1 < v_1, v_2 >$ folgt. Dies ist allerdings nur erfüllt, wenn $\lambda_1 = \lambda_2$ oder $< v_1, v_2 > = 0$ gilt. Da $\lambda_1 \neq \lambda_2$ Voraussetzung war, gilt für das Skalarprodukt $< v_1, v_2 > = 0$ und damit die Orthogonalität der beiden Vektoren. Orthogonale Vektoren ungleich dem Nullvektor sind immer linear unabhängig und bilden daher ein Orthogonalsystem.

□

Der folgende Satz führt zu der Definition der Schurzerlegung, welche benötigt wird, um das Eigenwertproblem in eine andere Form zu überführen, welche dann numerisch gelöst werden kann.

Satz 2.2.5. *Zu einer Matrix $A \in \mathbb{C}^{n\times n}$ lässt sich eine unitäre Matrix $Q \in \mathbb{C}^{n\times n}$ so finden, dass*

$$R = Q^*AQ, \tag{2.5}$$

gilt, wobei $R \in \mathbb{C}^{n\times n}$ eine obere Dreiecksmatrix ist.

Beweis. Zur Konstruktion der Zerlegung muss zunächst ein Eigenwert $\lambda_1 \in \mathbb{C}$ und der entsprechende normierte Eigenvektor $v_1 \in \mathbb{C}^n$ von A berechnet werden. Diesen kann man mittels des Orthogonalisierungsverfahrens von Gram-Schmidt [4] zu einer

orthonormalen Basis $v_1, y_2, \ldots, y_n$ von $\mathbb{C}^n$ erweitern. Die Vektoren werden dann als Spalten zu einer unitären Matrix Q_1 zusammengefasst, welche die Gleichung

$$Q_1^* A Q_1 = \left[\begin{array}{c|ccc} \lambda_1 & *^1 & \cdots & * \\ \hline 0 & & & \\ \vdots & & A_1 & \\ 0 & & & \end{array}\right]$$

erfüllt, wobei $A_1 \in \mathbb{C}^{(n-1)\times(n-1)}$ eine modifizierte Untermatrix von A ist.
Nun wird wiederum ein Eigenpaar der Matrix A_1 berechnet und wie im ersten Schritt eine unitäre Matrix $Q_2 \in \mathbb{C}^{(n-1)\times(n-1)}$ bestimmt. Die Matrix Q_2 wird auf eine Matrix $\tilde{Q}_2$ der Dimension n erweitert, sodass die Unitarität erhalten bleibt:

$$\tilde{Q}_2 = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & Q_2 & \\ 0 & & & \end{array}\right]$$

Dann gilt für die Matrix A:

$$\tilde{Q_2}^* Q_1^* A Q_1 \tilde{Q}_2 = \left[\begin{array}{cc|ccc} \lambda_1 & * & * & \cdots & * \\ 0 & \lambda_2 & * & \cdots & * \\ \hline 0 & 0 & & & \\ \vdots & \vdots & & A_2 & \\ 0 & 0 & & & \end{array}\right]$$

Dabei ist A_2 eine $(n-2)\times(n-2)$-Matrix. Diese Schritte kann man induktiv fortführen, und durch das Produkt die gewünschte unitäre Matrix

$$Q = Q_1 \tilde{Q}_2 \ldots \tilde{Q}_{n-1}$$

erhalten.

□

Definition 2.2.6. Die Zerlegung in (2.5) wird als **Schur-Zerlegung** von A bezeichnet.
Dabei handelt es sich um eine orthogonale Ähnlichkeitstransformation. Falls die Matrix $A \in \mathbb{C}^{n\times n}$ hermitesch ist, ist die obere Dreiecksmatrix R wegen Bemerkung 2.1.12 ebenfalls hermitesch und damit eine Diagonalmatrix, welche wir mit Λ bezeichnen.
Die Zerlegung

$$\Lambda = Q^* A Q \tag{2.6}$$

wird dann als **Spektralzerlegung** von A bezeichnet.

[1] Als Eintrag in Matrizen bedeutet $*$ eine beliebige Zahl.

Die in Definition 2.1.11 eingeführte Ähnlichkeitstransformation hat im Zusammenhang mit dem Eigenwertproblem eine besondere Eigenschaft, welche von vielen numerischen Algorithmen genutzt wird und mit dem folgenden Satz beschrieben wird.

Satz 2.2.7. *Die Eigenwerte sind bezüglich einer Ähnlichkeitstransformation invariant. Das Eigenpaar einer zu $A \in \mathbb{C}^{n\times n}$ ähnlichen Matrix $\tilde{A} \in \mathbb{C}^{n\times n}$ ist*

$$(\lambda, C^{-1}v),$$

wenn (λ, v) ein Eigenpaar von A ist und $C \in \mathbb{C}^{n\times n}$ als Transformationsmatrix genutzt wurde.

Beweis. Gegeben sei ein Eigenpaar (λ, v) der Matrix A und eine Transformationsmatrix C. Zu zeigen ist die Aussage

$$\tilde{A}(C^{-1}v) = \lambda(C^{-1}v). \tag{2.7}$$

Daraus folgt:

$$\begin{aligned} & (C^{-1}A\underbrace{C)(C^{-1}}_{I}v) &&= \lambda(C^{-1}v) \\ \Leftrightarrow \quad & C^{-1}(Av) &&= C^{-1}(\lambda v) \\ \Leftrightarrow \quad & Av &&= \lambda v \end{aligned} \tag{2.8}$$

Die letzte Aussage der Rechnung (2.8) ist wahr, da (λ, v) als Eigenpaar von A vorgegeben war. Da nur Äquivalenzumformungen genutzt wurden, gilt auch (2.7), womit bewiesen ist, dass $(\lambda, C^{-1}v)$ ein Eigenpaar von $\tilde{A}$ ist.

□

Bemerkung 2.2.8. Aus der Spektralzerlegung (2.6) folgt $AQ = Q\Lambda$, was die Matrixschreibweise für das Eigenwertproblem in (2.4) darstellt. Dabei gelten für die Matrizen

$$\Lambda = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix} \quad \text{und} \quad Q = \begin{bmatrix} & & & \\ v_1 & v_2 & \cdots & v_n \\ & & & \end{bmatrix}, \tag{2.9}$$

dass sie die Eigenwerte $\lambda_i \in \mathbb{R}$ und Eigenvektoren $v_i \in \mathbb{C}^n$ beinhalten. Dies liegt daran, dass es sich bei der Transformation Q^*AQ der Matrix A um eine Ähnlichkeitstransformation handelt, wodurch die Eigenwerte beim Übergang der Matrix A auf die Diagonalmatrix Λ nicht verändert werden, wie im Satz 2.2.7 gezeigt wurde. Daher kann man zur Lösung des hermiteschen bzw. symmetrischen Eigenwertproblems die Spektralzerlegung (2.6) berechnen.

Satz 2.2.9. *Die Eigenvektoren sind bezüglich eines Shifts invariant. Das Eigenpaar einer um den Shift $\mu \in \mathbb{C}$ verschoben Matrix $(A + \mu I) \in \mathbb{C}^{n \times n}$ ist*

$$(\lambda + \mu, v),$$

wenn (λ, v) ein Eigenpaar von $A \in \mathbb{C}^{n \times n}$ ist.

Beweis. Gegeben sei ein Eigenpaar (λ, v) der Matrix A. Dann gilt:

$$\begin{array}{rlcl} & Av & = & \lambda v \\ \Leftrightarrow & Av + \mu v & = & \lambda v + \mu v \\ \Leftrightarrow & (A + \mu I)v & = & (\lambda + \mu)v \end{array}$$

Die letzte Zeile zeigt, dass v der zugehörige Eigenvektor zum Eigenwert $(\lambda + \mu)$ der Matrix $(A + \mu I)$ ist.

□

Kapitel 3

Numerische Verfahren

In diesem Kapitel wird die Vorgehensweise zur numerischen Lösung des Eigenwertproblems beschrieben. Dabei wird die Matrix A aus dem Eigenwertproblem $Av = \lambda v$ als symmetrisch und reell vorausgesetzt, um die speziellen Eigenschaften symmetrischer Matrizen, wie sie im vorherigen Kapitel beschrieben wurden, nutzen zu können. Dazu wird bei A von einer dichtbesetzten Matrix ausgegangen, welche keine oder nur wenige Nulleinträge hat.
Es wird zunächst die für dichtbesetzte symmetrische Matrizen übliche Vorgehensweise erläutert, und im Folgenden auf die einzelnen Algorithmen eingegangen.

3.1 Allgemeine Vorgehensweise

Die numerische Berechnung der Eigenwerte bzw. -vektoren einer dichtbesetzten symmetrischen Matrix A erfolgt mittels eines Verfahrens, welches aus drei Schritten besteht.

1. Zu Beginn wird die dichtbesetzte Matrix $A \in \mathbb{R}^{n\times n}$ auf eine symmetrische Tridiagonalmatrix $T \in \mathbb{R}^{n\times n}$ reduziert. Dies geschieht mit Hilfe von orthogonalen Ähnlichkeitstransformationen, wie in (2.3) definiert, zum Beispiel der Householdertransformation, welche im nächsten Abschnitt beschrieben wird. Eine weitere bekannte Möglichkeit zur Konstruktion einer entsprechenden Orthogonalmatrix bietet die Givenstransformation [5].

2. Im zweiten Schritt muss nun das symmetrische tridiagonale Eigenwertproblem

$$Tz = \lambda z, \quad z \in \mathbb{R}^n \tag{3.1}$$

gelöst werden. Hierfür existieren mehrere Algorithmen, welche in Abschnitt 3.3 erläutert werden.
Die im zweiten Schritt berechneten Skalare λ sind sowohl Eigenwerte der Matrix T, als auch von A. Dies liegt daran, dass im ersten Schritt für die Reduktion nur Ähnlichkeitstransformationen verwendet wurden, welche die Eigenwerte nicht verändern.

3. Da die Eigenvektoren z der transformierten Matrix aber durchaus verändert wurden, müssen diese im dritten Schritt rücktransformiert werden, falls die Eigenvektoren v der Matrix A von Interesse sind. Die Rücktransformation wird in Abschnitt 3.4 genauer beschrieben.

3.2 Tridiagonalisierung

Zu einer symmetrischen Matrix $A \in \mathbb{R}^{n\times n}$ lässt sich eine Orthogonalmatrix $Q \in \mathbb{R}^{n\times n}$ finden, so dass

$$Q^{\mathsf{T}} A Q = T \tag{3.2}$$

eine symmetrische Tridiagonalmatrix ist. Zur Konstruktion der Orthogonalmatrix Q können die Householdertransformationen bzw. Givenstransformationen genutzt werden. Daher spricht man bei den Reduktionsverfahren von Householderreduktion bzw. Givensreduktion.
Beide Reduktionsverfahren bringen eine dichtbesetzte Matrix auf die Hessenbergform [6]. Es handelt sich dabei um orthogonale Ähnlichkeitstransformationen, so dass die Eigenschaft der Symmetrie nicht verloren geht. Bei einer Matrix in Hessenbergform, welche darüber hinaus symmetrisch ist, handelt es sich um eine Tridiagonalmatrix.
Die Givenstransformation ist eine lineare Abbildung, welche eine Rotation eines Vektors im euklidische Raum darstellt. Q ergibt sich aus dem Produkt der orthogonalen Rotationsmatrizen. Die Givensrotation wird häufig verwendet um einzelne Einträge in Matrizen gezielt zu eliminieren oder dünnbesetzte Matrizen zu reduzieren, kann aber als vollständiges Reduktionsverfahren für dichte Matrizen nicht mit dem 1958 von Householder eingeführten Verfahren mithalten [5]. Daher wird hier nicht weiter auf die Givensreduktion eingegangen.
Die Householdertransformation ist ebenfalls eine lineare Abbildung, stellt aber, statt einer Rotation, eine Spiegelung eines Vektors im euklidischen Raum dar. Zu zwei Vektoren $x, z \in \mathbb{R}^n$ ist die Transformationsmatrix (Householdermatrix) $H \in \mathbb{R}^{n\times n}$ definiert durch

$$H := I - 2yy^{\mathsf{T}} \quad \text{mit} \quad y := \frac{x - z}{\|x - z\|},$$

wobei I die Einheitsmatrix ist. Die Matrix H ist dann eine symmetrische Orthogonalmatrix und es gilt $z = Hx$ [7].

Für alle $k \in \{1, \ldots, n-2\}$ lassen sich nun Vektoren $y_k \in \mathbb{R}^n$ so konstruieren, dass für die Householdermatrizen H_k gilt [8]:

$$H_k x = H_k \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ x_{k+2} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ * \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Wählt man nun

$$\tilde{a}_k := \begin{bmatrix} 0 \\ \vdots \\ 0 \\ a_{k,k+1} \\ a_{k,k+2} \\ \vdots \\ a_{k,n} \end{bmatrix} \quad \text{und} \quad y_k := \frac{\tilde{a}_k - \|\tilde{a}_k\| e_{k+1}}{\|\tilde{a}_k - \|\tilde{a}_k\| e_{k+1}\|},$$

wobei $e_{k+1} \in \mathbb{R}^n$ den $(k+1)$-ten Einheitsvektor bezeichnet, so ist

$$H_k := I - 2 y_k y_k^{\mathsf{T}}$$

eine Householdermatrix und es gilt:

$$H_k a_k = H_k \begin{bmatrix} a_{k,1} \\ \vdots \\ a_{k,k} \\ a_{k,k+1} \\ a_{k,k+2} \\ \vdots \\ a_{k,n} \end{bmatrix} = \begin{bmatrix} a_{k,1} \\ \vdots \\ a_{k,k} \\ * \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.3}$$

Mit Hilfe dieser Konstruktion der Householdermatrizen, kann man die voll besetzte Matrix A iterativ mit $(n-2)$ Ähnlichkeitstransformationen auf eine Tridiagonalgestalt bringen. Im Folgenden bezeichne $A^{(k)}$ die k-dimensionale Untermatrix von A, wobei die ersten $(n-k)$ Zeilen und Spalten jeweils abgeschnitten sind.

Für die erste Ähnlichkeitstransformation mit einer entsprechend konstruierten Householdermatrix H_1 gilt:

$$H_1^\mathsf{T} A H_1 = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & H_1^{(n-1)} & \\ 0 & & & \end{array}\right] \left[\begin{array}{c|ccc} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ \hline a_{2,1} & & & \\ \vdots & & A^{(n-1)} & \\ a_{n,1} & & & \end{array}\right] \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & H_1^{(n-1)} & \\ 0 & & & \end{array}\right]$$

$$= \left[\begin{array}{c|ccc} a_{1,1} & \beta_2 & 0 & \cdots & 0 \\ \hline \beta_2 & & & & \\ 0 & & & & \\ \vdots & & \tilde{A}_1^{(n-1)} & & \\ 0 & & & & \end{array}\right] = \tilde{A}_1$$

Dabei sorgt die Multiplikation mit der Householdermatrix von links wie in (3.3) für Eliminierung in der ersten Spalte von A. Die Multiplikation von rechts ist für die Nulleinträge in der ersten Zeile der Ergebnismatrix verantwortlich.
Die zweite Householdermatrix H_2 wird nun auf die zu A ähnliche Matrix $\tilde{A}_1$ angewandt, um in der zweiten Spalte bzw. Zeile Nulleinträge zu erzeugen.

$$H_2 \tilde{A}_1 H_2 = \left[\begin{array}{cc|ccccc} a_{1,1} & \beta_2 & 0 & 0 & \cdots & 0 \\ \beta_2 & \alpha_2 & \beta_3 & 0 & \cdots & 0 \\ \hline 0 & \beta_3 & & & & \\ 0 & 0 & & & & \\ \vdots & \vdots & & \tilde{A}_2^{(n-2)} & & \\ 0 & 0 & & & & \end{array}\right] = \tilde{A}_2$$

Durch die weiteren Ähnlichkeitstransformationen mit den Householdermatrizen entsteht die zu A ähnliche Tridiagonalmatrix T. Die $(n-2)$ Ähnlichkeitstransformationen können durch ihr Produkt als Matrix zusammengefasst werden.

$$\underbrace{H_{n-2} \dots H_1}_{Q^\mathsf{T}} A \underbrace{H_1 \dots H_{n-2}}_{Q} = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-1} & \alpha_{n-1} & \beta_n \\ & & & \beta_n & \alpha_n \end{bmatrix} = T \qquad (3.4)$$

Das Produkt Q der Orthogonalmatrizen bleibt dabei eine Orthogonalmatrix und erfüllt somit die Gleichung (3.2).

Die Berechnung der Householdertransformation wird dabei mit zwei Subtraktionen ausgeführt. Es gilt für die Multiplikation von links und rechts folgende Gleichung:

$$\begin{aligned} H^\mathsf{T}AH = HAH &= (I - 2yy^\mathsf{T})A(I - 2yy^\mathsf{T}) \\ &= (A - 2yy^\mathsf{T}A)(I - 2yy^\mathsf{T}) \\ &= A - 2yy^\mathsf{T}A - 2Ayy^\mathsf{T} + 4yy^\mathsf{T}Ayy^\mathsf{T} \\ &= A - 2yy^\mathsf{T}A + 2yy^\mathsf{T}Ayy^\mathsf{T} - 2Ayy^\mathsf{T} + 2yy^\mathsf{T}Ayy^\mathsf{T} \\ &= A - y\underbrace{(2y^\mathsf{T}A - 2y^\mathsf{T}Ayy^\mathsf{T})}_{:=v^\mathsf{T}} - \underbrace{(2Ay - 2yy^\mathsf{T}Ay)}_{:=v}y^\mathsf{T} \end{aligned}$$

Nun muss noch gezeigt werden, dass es sich bei v^T um den transponierten Vektor von v handelt:

$$(2Ay - 2yy^\mathsf{T}Ay)^\mathsf{T} = 2y^\mathsf{T}A - 2(Ay)^\mathsf{T}(yy^\mathsf{T})^\mathsf{T} = 2y^\mathsf{T}A - 2y^\mathsf{T}Ayy^\mathsf{T}$$

Damit gilt nun für die Berechnung der Householdertransformation:

$$H^\mathsf{T}AH = A - yv^\mathsf{T} - vy^\mathsf{T} \quad \text{mit} \quad v := 2Ay - 2yy^\mathsf{T}Ay$$

Input : A
for $(k = 1, \ldots, (n-2))$ **do**
  $\mu := \|\tilde{a}_k\|$
  $y := \frac{\tilde{a}_k - \mu e_{k+1}}{\|\tilde{a}_k - \mu e_{k+1}\|}$
  $v := 2Ay - 2yy^\mathsf{T}Ay$
  $A_{k+1,k} := \mu$
  $A_{k,k+1} := \mu$
  $A_{k+2:n,k} := \vec{0}$
  $A_{k,k+2:n} := \vec{0}^\mathsf{T}$
  $A_{k+1:n,k+1:n} := A_{k+1:n,k+1:n} - (yv^\mathsf{T})_{k+1:n,k+1:n} - (vy^\mathsf{T})_{k+1:n,k+1:n}$
end
return A

Algorithmus 3.2.1: Householderreduktion der symmetrischen Matrix $A \in \mathbb{R}^{n\times n}$. Die Matrix A wird mit der Tridiagonalmatrix $T \in \mathbb{R}^{n\times n}$ aus (3.4) überschrieben.

Die Tridiagonalisierung einer symmetrischen Matrix mit Hilfe der Householderre-

duktion benötigt $\frac{4n^3}{3}$ Flops[2], welche größtenteils aus Level 2 BLAS[3] bestehen [9]. Sollen insgesamt die Eigenvektoren von A ebenfalls bestimmt werden, wird für die Rücktransformation (siehe Abschnitt 3.4) die Orthogonalmatrix Q benötigt. Q wird allerdings im Algorithmus 3.2.1 gar nicht bestimmt, wodurch sich der Aufwand bei der zusätzlichen Berechnung von Q auf $\frac{8n^3}{3}$ Flops erhöht [8].
Da die Level 2 BLAS ineffizienter als Level 3 BLAS[4] sind [10], versucht man in jedem Algorithmus so viel wie möglich mit den Level 3 BLAS zu berechnen.
Bischof und Van Loan haben sich in diesem Kontext mit der Householderreduktion beschäftigt und die WY-Darstellung [11] für eine Sequenz von Produkten von Householdermatrizen eingeführt.
Bezeichne nun Q_k das Produkt der ersten k Householdermatrizen, dann gilt:

$$Q_k = H_1 \dots H_k = I - W_k Y_k^\mathsf{T} \quad \text{mit} \quad W_k, Y_k \in \mathbb{R}^{n \times k}$$

Die Konstruktion der Matrizen W_K und Y_k lässt sich induktiv zeigen, wobei für $k = 1$

$$Q_1 = H_1 = I - \underbrace{2y_1}_{W_1} \underbrace{y_1^\mathsf{T}}_{Y_1^\mathsf{T}} = I - W_1 Y_1^\mathsf{T}$$

gilt. Beim Übergang von k zu $k+1$ sind die Matrizen W_k und Y_k bekannt und es gilt:

$$\begin{aligned} Q_{k+1} = \underbrace{H_1 \dots H_k}_{Q_k} H_{k+1} &= (\underbrace{I - W_k Y_k^\mathsf{T}}_{Q_k})(\underbrace{I - 2y_{k+1} y_{k+1}^\mathsf{T}}_{H_{k+1}}) \\ &= I - W_k Y_k^\mathsf{T} - Q_k 2 y_{k+1} y_{k+1}^\mathsf{T} \\ &= I - \underbrace{(W_k \mid Q_k 2y_{k+1})}_{W_{k+1}} \underbrace{(Y_k \mid y_{k+1})^\mathsf{T}}_{Y_{k+1}^\mathsf{T}} \end{aligned}$$

Dabei bezeichnet $(A \mid b)$ die Matrix die aus A entsteht, wenn man den Spaltenvektor b als Spalte rechts hinzufügt.
Es kann eine geblockte Householdertransformation konstruiert werden, welche Produkte von Householdermatrizen in der WY-Darstellung verwendet. Diese kann vollbesetzte Matrizen aber nicht direkt auf Tridiagonalgestalt bringen. Hierfür existieren mehrstufige, meist aus zwei Stufen bestehende Algorithmen, welche eine vollbesetzte Matrix mit Level 3 BLAS Operation auf eine Bandmatrix reduzieren. Im zweiten Schritt werden diese Matrizen dann auf die Tridiagonalgestalt gebracht, wobei wieder einfache Householder- oder wegen der Dünnbesetztheit Givenstransformationen genutzt werden.
Da dieses Verfahren in den verwendeten Bibliotheken nicht eingesetzt wird, wird es an dieser Stelle nicht weiter beschrieben.

[2]Flops (floating point operations): In [8] wird ein Flop als eine einzelne Operation einer Fließkommazahl definiert. Dies kann eine Addition, Subtraktion, Multiplikation oder Division sein.

[3]Mit Level 2 BLAS sind Matrix-Vektor-Operationen gemeint. Genauere Informationen zu BLAS sind in Abschnitt 4.2.1.1 über den Aufbau von ScaLAPACK zu finden.

[4]Mit Level 3 BLAS sind in diesem Kontext Matrix-Matrix-Operationen gemeint.

3.3 Lösung des tridiagonalen Eigenwertproblems

In diesem Abschnitt werden die vier gebräuchlichsten Algorithmen zur Lösung des tridiagonalen Eigenwertproblems (3.1) vorgestellt, welche in den Routinen verwendet werden, die in dieser Arbeit getestet wurden.

3.3.1 QR-Algorithmus

Beim QR-Algorithmus werden nach der Reduktion weitere Ähnlichkeitstransformationen auf die Tridiagonalmatrix T aus (3.4) angewandt, so dass diese gegen eine Diagonalmatrix konvergiert. Dabei handelt es sich um die Diagonalmatrix $\Lambda \in \mathbb{R}^{n\times n}$ aus der Schur- bzw. Spektralzerlegung von T, wie sie in (2.9) beschrieben wurde. Diese beinhaltet die Eigenwerte der Matrix T als Diagonaleinträge.
Zur Konstruktion der Ähnlichkeitstransformationen wird die QR-Zerlegung von T_k

$$T_k = Q_k R_k \tag{3.5}$$

berechnet. Dabei ist $Q_k \in \mathbb{R}^{n\times n}$ eine Orthonormalmatrix und $R_k \in \mathbb{R}^{n\times n}$ eine obere Dreiecksmatrix. Zur Konstruktion der QR-Zerlegung sind in [12] drei Verfahren nach Householder, Givens und Gram-Schmidt beschrieben.
Nach der QR-Zerlegung wird die QR-Transformation

$$T_{k+1} := Q_k^{\mathsf{T}} T_k Q_k = R_k Q_k \tag{3.6}$$

berechnet [13].
Beginnend mit $T_0 = T$ konvergiert T_k mit diesem Verfahren gegen die Matrix Λ aus der Spektralzerlegung von T [14]:

$$\lim_{k\to\infty} T_k = \Lambda$$

Das QR-Verfahren kann auch auf eine voll besetzte symmetrische Matrix angewandt werden, aber die vorherige Reduktion auf eine Tridiagonalmatrix verringert den Aufwand.

Der Aufwand für jeden Iterationsschritt von Algorithmus 3.3.1 liegt bei $\mathcal{O}(n^3)$ Flops, sodass man insgesamt bei $\mathcal{O}(n^4)$ Flops liegt, was sehr schlecht ist. Wird der QR-Algorithmus direkt auf eine Tridiagonalmatrix T angewandt, kann der Aufwand pro Schritt auf $\mathcal{O}(n^2)$ Flops verringert werden [8]. Dies liegt daran, dass für die QR-Zerlegung einer dichtbesetzten Matrix normalerweise Householdermatrizen genutzt werden. Durch die Struktur der Tridiagonalmatrix bietet es sich allerdings an, die QR-Zerlegung effizienter über Givensmatrizen zu berechnen, da nur noch wenige Einträge gezielt eliminiert werden müssen.
Zur Beschleunigung der Konvergenz des QR-Algorithmus existieren mehrere modifizierte QR-Verfahren. Dabei werden meistens passende Verschiebungen, sogenannte *Shifts*, in der Matrix genutzt. So wird statt der in (3.5) angegebenen QR-Faktorisierung

$$T_k + \mu_k I = Q_k R_k$$

```
    Input : A, ε
1   while (max{a_{i,j} : 1 ≤ i ≤ n, 1 ≤ j < i} > ε) do
2       [Q, R] :=QR_Faktorisierung(A)
3       A := RQ
4   end
5   return A
```

Algorithmus 3.3.1: QR-Algorithmus mit voll besetzter symmetrischer Matrix $A \in \mathbb{R}^{n \times n}$. Die Matrix A wird dabei mit einer näherungsweisen Diagonalmatrix überschrieben, wobei der Parameter ϵ angibt, wie groß die Nulleinträge der Diagonalmatrix sein dürfen.

genutzt.
Daher muss dann auch die QR-Transformation aus (3.6) durch

$$T_{k+1} := Q_k^{\mathsf{T}} T_k Q_k = R_k Q_k - \mu_k I$$

angepasst werden. Es existieren verschiedene Shiftstrategien für den Parameter μ_k. Häufig wird der Wilkinson-Shift verwendet. Genauere Beschreibungen und Konvergenzuntersuchungen verschiedener Strategien sind in [14], [15] und [16] zu finden.
Ist man neben den Eigenwerten λ_i, mit welchen Λ die Diagonale besetzt hat, auch an den Eigenvektoren z_i von T interessiert, lassen sich diese leicht über die Orthogonalmatrizen Q_k bestimmen. Die Matrix

$$\hat{Q}_K := \prod_{k=0}^{K} Q_k$$

konvergiert für $K \to \infty$ gegen die in (2.9) bezeichnete Matrix Q aus der Spektralzerlegung von T, welche die Eigenvektoren als Spalten beinhaltet. Anders als bei den Eigenwerten, welche sich durch Ähnlichkeitstransformationen nicht verändern, handelt es sich hierbei nicht um die Eigenvektoren von A. Diese müssen noch rücktransformiert werden, was in Abschnitt 3.4 näher beschrieben wird.
Durch das Aufmultiplizieren der Matrix Q erhöht sich der Aufwand nur um eine Konstante. Damit bleibt es bei einem totalen Aufwand von $\mathcal{O}(n^3)$ für die Berechnung aller Eigenwerte und -vektoren einer tridiagonalen Matrix mittels des QR-Algorithmus.

3.3.2 Bisektion und Inverse Iteration

Bei diesem Ansatz zur Lösung des tridiagonalen Eigenwertproblems wird der Zusammenhang der Eigenwerte zum charakteristischen Polynom $\chi_T(\lambda)$ der Matrix T aus (3.4) genutzt, welcher in Definition 2.2.2 beschrieben wurde.

Die Matrix T muss für das Bisektionsverfahren irreduzibel sein.
Bezeichne

$$\chi_T^{(i)}(\lambda) = \det \begin{bmatrix} \alpha_1 - \lambda & \beta_2 & & & & \\ \beta_2 & \alpha_2 - \lambda & \beta_3 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \beta_{i-1} & \alpha_{i-1} - \lambda & \beta_i & \\ & & & \beta_i & \alpha_i - \lambda & \end{bmatrix}$$

das charakteristische Polynom der i-dimensionalen Untermatrix von T, wobei die rechten Spalten bzw. die unteren Zeilen abgeschnitten sind.
Diese Polynome lassen sich für jedes λ schnell auswerten, da sie folgende Rekursionsformel erfüllen:

$$\chi_T^{(0)}(\lambda) := 1, \quad \chi_T^{(1)}(\lambda) := \alpha_1 - \lambda$$

$$\underset{i \in \{2,\dots,n\}}{\forall} : \quad \chi_T^{(i)}(\lambda) = (\alpha_i - \lambda)\chi_T^{(i-1)}(\lambda) - \beta_i^2 \chi_T^{(i-2)}(\lambda)$$

Diese Rekursionsformel lässt sich induktiv über den Laplaceschen Entwicklungssatz [17], welcher die tridiagonale Struktur der Matrix ausnutzt, beweisen.
Mit Hilfe des Verschachtelungssatzes von Cauchy (Cauchys Interlace Theorem [18]) lässt sich zeigen, dass es sich bei der Polynomfolge

$$\chi_T^{(n)}(\lambda) \ , \ \chi_T^{(n-1)}(\lambda) \ , \ \dots \ , \ \chi_T^{(0)}(\lambda)$$

um eine Sturmsche Kette handelt [5].
Die Sturmsche Kette ist eine endliche Folge von Polynomen mit absteigendem Polynomgrad und einer speziellen Verschachtelungseigenschaft der Nullstellen der Polynome.
Man kann mit Hilfe eines Satzes aus der Sturm Theorie [19] Aussagen über die Nullstellen des ersten Polynoms, also des Polynoms mit höchstem Grad, machen. Da in diesem Fall das Polynom mit höchstem Grad $\chi_T^{(n)}(\lambda)$ identisch mit dem charakteristischen Polynom $\chi_T(\lambda)$ der Matrix T ist, können die folgenden Aussagen über die Nullstellen direkt auf die Eigenwerte von T übertragen werden:
Die Anzahl der Nullstellen des Polynoms $\chi_T^{(n)}(\lambda)$ im halboffenen Intervall $]a, b]$ ist gleich $\sigma(b) - \sigma(a)$, wobei $\sigma(\mu)$ für ein festes $\mu \in \mathbb{R}$ die Anzahl der Vorzeichenwechsel in der Zahlenfolge

$$\chi_T^{(n)}(\mu), \chi_T^{(n-1)}(\mu), \dots, \chi_T^{(0)}(\mu)$$

bezeichnet [20]. Die Anzahl der Nullstellen des Polynoms, die kleiner als μ sind, kann daher mit $\sigma(\mu)$ bestimmt werden.
Für die Bisektion wird nun ein Intervall vorgegeben, in welchem nach einem Eigenwert gesucht wird. Dieser kann gefunden werden, indem man das Intervall, in

dem man sucht, halbiert und dann überprüft, in welcher der beiden Hälften sich der gesuchte Eigenwert befindet. Mit dieser Intervallhälfte als neues Intervall, wird die Suche dann sukzessiv fortgeführt, bis die Intervallbreite kleiner als eine vorgegebene Fehlerschranke ist.
Um die Suche zu Beginn schon einzugrenzen und passende Startwerte zu finden, kann man für tridiagonale Matrizen leicht ein Intervall angeben, in welchem alle Eigenwerte liegen müssen. Zur Abschätzung werden Gerschgorin Kreise [21] genutzt:

$$a := \min_{1\leq i\leq n} \alpha_i - \mid \beta_i \mid - \mid \beta_{i+1} \mid, \quad b := \max_{1\leq i\leq n} \alpha_i + \mid \beta_i \mid + \mid \beta_{i+1} \mid \tag{3.7}$$

Dabei werden die nicht in der Matrix T vorhandenen Koeffizienten β_1 und β_{n+1} auf 0 gesetzt.
Werden die Intervallgrenzen a und b wie in (3.7) gewählt, liegen alle n Eigenwerte der Tridiagonalmatrix T innerhalb des Intervalls.

```
     Input : T, k, ε
1    a := min{α_i − | β_i | − | β_{i+1} | : 1 ≤ i ≤ n}
2    b := max{α_i + | β_i | + | β_{i+1} | : 1 ≤ i ≤ n}
3    while (| a − b |> ε) do
4        μ := a + (b−a)/2
5        if (σ(μ) < k) then
6            a := μ
7        else
8            b := μ
9        end
10   end
11    λ_k := a + (b−a)/2
12   return λ_k
```

Algorithmus 3.3.2: Bisektion zur Berechnung des k-ten Eigenwertes $\lambda_k \in \mathbb{R}$ von der Tridiagonalmatrix $T \in \mathbb{R}^{n\times n}$ aus (3.4). Der Parameter ϵ beschreibt dabei die Genauigkeit für die Berechnung des Eigenwertes.

Die Bisektion ist besonders nützlich, wenn nicht alle, sondern nur bestimmte Eigenwerte von Interesse sind, was in der Praxis häufig gefordert ist. Der Aufwand für Algorithmus 3.3.2 beträgt $\mathcal{O}(n)$ Flops, also zur Berechnung von insgesamt k Eigenwerten $\mathcal{O}(kn)$ Flops.
Sollen neben den Eigenwerten zusätzlich die entsprechenden Eigenvektoren berechnet werden, wird dies über die Inverse Iteration gemacht, welche 1944 von Wielandt eingeführt wurde [22]. Die Basis des Verfahrens beruht auf einer Überlegung von

Richard von Mises, welcher herausgefunden hat, dass die folgende Vektoriteration gegen den Eigenvektor des betragsmäßig größten Eigenwerts von A konvergiert [23]:

$$v_{i+1} = Av_i \tag{3.8}$$

Dabei kann mit einem beliebigen Startvektor v_0 gestartet werden.
Bei der Inversen Iteration wird ausgenutzt, dass der mit der Bisektion berechnete Eigenwert, im folgenden mit $\hat{\lambda}_k$ bezeichnet, nah an dem wahren Eigenwert λ_k von T liegt, da

$$| \hat{\lambda}_k - \lambda_k | \leq \frac{\epsilon}{2} \tag{3.9}$$

gilt, was direkt aus dem Algorithmus 3.3.2 folgt. Des Weiteren ist der Zusammenhang zwischen der Matrix

$$T - \hat{\lambda}_k I \tag{3.10}$$

und der Matrix T sehr wichtig für dieses Verfahren. Denn falls λ_k der Eigenwert der Matrix T zum Eigenvektor z_k ist, dann ist $\lambda_k - \hat{\lambda}_k$ der entsprechende Eigenwert der Matrix (3.10) zum gleichen Eigenvektor z_k. Dies folgt aus Satz 2.2.9, da es sich bei (3.10) um eine geshiftete Matrix handelt.
Da die Abschätzung in (3.9) gilt, handelt es sich bei $\lambda_k - \hat{\lambda}_k$ um den betragsmäßig kleinsten Eigenwert. Die inverse Matrix $(T - \hat{\lambda}_k)^{-1}$ hat daher mit $(\lambda_k - \hat{\lambda}_k)^{-1}$ den betragsmäßig größten Eigenwert zum Eigenvektor z_k, sodass dieser mit Hilfe der Vektoriteration aus (3.8) angenähert werden kann.

Input : $T, \hat{\lambda}_k, \epsilon$
1 **repeat**
2 $\tilde{z} := (T - \hat{\lambda}_k)^{-1} z$
3 $z := \frac{\tilde{z}}{\|\tilde{z}\|}$
4 **until** $\|(T - \hat{\lambda}_k)z\| > \epsilon$
5 **return** z

Algorithmus 3.3.3: Inverse Iteration zur Berechnung des Eigenvektors z_k der Matrix T zum gegebenen Eigenwert $\hat{\lambda}_k$ welcher mittels Bisektion bestimmt wurde. Der Parameter ϵ beschreibt die gewünschte Genauigkeit des Eigenvektors.

Der Aufwand der Inversen Iteration hängt stark von der Verteilung der Eigenwerte ab. Wenn die Eigenwerte in Clustern auftreten, also

$$| \lambda_i - \lambda_j | < \epsilon$$

für viele, oder im Worst Case für alle $1 \leq i, j \leq n$ gilt, so ist das Verfahren der Inversen Iteration nicht effektiv. Der Aufwand liegt im Worst Case Szenario bei

$\mathcal{O}(nk^2)$ Flops für die Berechnung der Eigenvektoren zu den k Eigenwerten. Sind die Eigenwerte nicht geclustert, liegt der Aufwand bei $\mathcal{O}(nk)$ Flops [24].
Ein weiteres Problem der Inversen Iteration bei geclusterten Eigenwerten ist, dass die Orthogonalität der Eigenvektoren nicht gewährleistet wird [9]. Daher müssen die Eigenvektoren nach der Berechnung nachorthogonalisiert werden, was zum Beispiel mit Hilfe des Gram-Schmidt Verfahrens [4] realisiert werden kann. Die Nachorthogonalisierung kann sogar fehlschlagen, so dass die Inverse Iteration bei stark geclusterten Eigenwerten gar nicht zu gebrauchen ist [25].

3.3.3 Divide-and-Conquer

Ein Divide-and-Conquer Verfahren wurde 1981 von Cuppen eingeführt [26]. Es berechnet die Schur- bzw. die Spektralzerlegung (2.6) für ein symmetrisches tridiagonales Eigenwertproblem.
Die Matrix T wird in zwei Hälften unterteilt, für welche dann jeweils die Spektralzerlegung berechnet wird. Diese werden dann wieder zur Spektralzerlegung von T zusammengeführt. Die Berechnungen der beiden Spektralzerlegungen der Hälften erfolgen ebenfalls über dieses Verfahren, wodurch eine Rekursion definiert ist.
Die Zerlegung der Matrix T in zwei Hälften sieht wie folgt aus:

$$T = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \beta_{n-1} & \alpha_{n-1} & \beta_n \\ & & & \beta_n & \alpha_n \end{bmatrix} = \left[\begin{array}{cc} T_1 & \\ & \beta_{m+1} \\ \hline \beta_{m+1} & \\ & T_2 \end{array}\right] \tag{3.11}$$

$$= \underbrace{\left[\begin{array}{cc} \check{T}_1 & \\ & \\ \hline & \check{T}_2 \end{array}\right]}_{:=\check{T}} + \underbrace{\left[\begin{array}{cc} & \\ & \beta_{m+1} \\ \hline \beta_{m+1} & \end{array}\right]}_{:=K_{m+1}} \tag{3.12}$$

Die symmetrische Tridiagonalmatrix $T \in \mathbb{R}^{n\times n}$ lässt sich so für jedes $m < n$ in zwei symmetrische tridiagonale Matrizen $\check{T}_1 \in \mathbb{R}^{m\times m}$, $\check{T}_2 \in \mathbb{R}^{(n-m)\times(n-m)}$ und eine einrangige Korrekturmatrix $K_{m+1} \in \mathbb{R}^{n\times n}$ zerlegen. Diese lässt sich durch

$$K_{m+1} = \beta_{m+1} v v^{\mathsf{T}} = \beta_{m+1} \begin{bmatrix} e_m \\ e_1 \end{bmatrix} \begin{bmatrix} e_m^{\mathsf{T}} & e_1^{\mathsf{T}} \end{bmatrix}$$

angeben, wobei $e_m \in \mathbb{R}^m$ den m-ten und $e_1 \in \mathbb{R}^{n-m}$ den ersten Einheitsvektor darstellen.
Wenn man nun davon ausgeht, dass die Spektralzerlegungen der beiden Matrizen $\check{T}_1$ und $\check{T}_2$ schon berechnet wurden, also die Orthogonalmatrizen Q_1, Q_2 und die

Diagonalmatrizen Λ_1, Λ_2 schon bekannt sind, lassen sich diese Teilergebnisse zusammenführen, um an die Spektralzerlegung der Matrix T zu gelangen.
Dies geschieht über

$$\begin{aligned} T &= \begin{bmatrix} Q_1\Lambda_1 Q_1^\mathsf{T} & \\ & Q_2\Lambda_2 Q_2^\mathsf{T} \end{bmatrix} + K_{m+1} \\ &= \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix} \begin{bmatrix} Q_1^\mathsf{T} & \\ & Q_2^\mathsf{T} \end{bmatrix} + \beta_{m+1} v v^\mathsf{T} \\ &= \underbrace{\begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix}}_{:=Q_{1,2}} \left(\underbrace{\begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix}}_{:=\Lambda_{1,2}} + \beta_{m+1} \check{v}\check{v}^\mathsf{T} \right) \underbrace{\begin{bmatrix} Q_1^\mathsf{T} & \\ & Q_2^\mathsf{T} \end{bmatrix}}_{:=Q_{1,2}^\mathsf{T}} \end{aligned}$$

wobei

$$\beta_{m+1}\check{v}\check{v}^\mathsf{T} = \beta_{m+1} \begin{bmatrix} Q_1^\mathsf{T} e_m \\ Q_2^\mathsf{T} e_1 \end{bmatrix} \begin{bmatrix} e_m^\mathsf{T} Q_1 & e_1^\mathsf{T} Q_2 \end{bmatrix}$$

gilt, da Q_1 und Q_2 Orthogonalmatrizen sind.
Nun wird die Spektralzerlegung

$$\Lambda_{1,2} + \beta_{m+1}\check{v}\check{v}^\mathsf{T} = Z\Lambda Z^\mathsf{T} \tag{3.13}$$

berechnet, um so die zwei Spektralzerlegungen der Teilmatrizen auf die Matrix T zurückzuführen:

$$T = Q_{1,2}(\Lambda_{1,2} + \beta_{m+1}\check{v}\check{v}^\mathsf{T})Q_{1,2}^\mathsf{T} = (Q_{1,2}Z)\Lambda(Q_{1,2}Z)^\mathsf{T} \tag{3.14}$$

Dabei handelt es sich schließlich um die vollständige Spektralzerlegung von T, wobei Λ die Diagonalmatrix mit den Eigenwerten und $(Q_{1,2}Z)$ die Orthogonalmatrix mit den Eigenvektoren von T ist.
Die Spektralzerlegung in (3.13) kann dabei sehr leicht über die charakteristische Gleichung

$$f(\lambda) := 1 + \beta_{m+1} \sum_{i=1}^{n} \frac{\check{v}_i^2}{\check{\lambda}_i - \lambda} = 0, \tag{3.15}$$

welche als *secular equation* bezeichnet wird, bestimmt werden [9]. Bei $\check{\lambda}_i$ handelt es sich um die Einträge der Diagonalmatrix $\Lambda_{1,2}$. Die Lösungen λ_i der Gleichung sind die Eigenwerte der Matrix $\Lambda_{1,2} + \beta_{m+1}\check{v}\check{v}^\mathsf{T}$. Die Eigenvektoren dazu werden über

$$z_i = (\Lambda_{1,2} - \lambda_i I)^{-1}\check{v}$$

bestimmt. Durch die Eigenwerte und -vektoren ist die Spektralzerlegung aus (3.13) und damit auch die Spektralzerlegung von T (3.14) bestimmt.

Die als bekannt vorausgesetzten Matrizen $\Lambda_1, \Lambda_2, Q_1, Q_2$ werden ebenfalls über diesen Divide-and-Conquer Algorithmus berechnet, indem die Matrizen $\check{T}_1$ und $\check{T}_2$ selber jeweils in zwei Hälften zerlegt werden.
So überführt man ein n-dimensionales Eigenwertproblem in n 1-dimensionale Probleme, oder löst die Probleme niedrigerer Dimensionen mit Hilfe anderer Algorithmen.

```
     Input : T
1    if (T ∈ ℝ^{1×1}) then
2        Λ := T
3        Q := 1
4    else
5        [Ť_1, Ť_2, β_{m+1}, v] :=divide(T)
6        [Λ_1, Q_1] := divide_conquer(Ť_1)
7        [Λ_2, Q_2] := divide_conquer(Ť_2)
8        v̌ := Q_{1,2}^T v
9        [Λ, Z] :=solve_secular(β_{m+1}, Λ_{1,2}, v̌)
10       Q := Q_{1,2} Z
11   end
12   return [Λ, Q]
```

Algorithmus 3.3.4: Die rekursive Funktion `divide_conquer` wird mit der Tridiagonalmatrix T aufgerufen. Sie berechnet die Spektralzerlegung, also die Matrizen Λ und Q, mit Hilfe von Cuppens Divide-and-Conquer Algorithmus. Die Funktion `divide` berechnet die Zerlegung der Matrix T, wie in (3.11) und (3.12) beschrieben. Die Funktion `solve_secular` löst die in (3.15) angegebene Gleichung.

Der Aufwand für dieses Verfahren ist wie auch bei der Inversen Iteration abhängig von der Lage der Eigenwerte. Im Durchschnitt liegt der Aufwand bei $\mathcal{O}(n^{2.3})$ Flops, kann aber in besonders günstigen Fällen bei $\mathcal{O}(n^2)$ Flops liegen. Im Worst Case Szenario würde das Divide-and-Conquer Verfahren $\mathcal{O}(n^3)$ Flops benötigen [24].

3.3.4 MRRR-Algorithmus

Der MRRR- oder MR^3-Algorithmus (Algorithm of Multiple Relatively Robust Representations) wurde 2004 von Dhillon und Parlett veröffentlicht [27]. Er ermöglicht es, wie auch schon die Bisektion, nur einen Teil der Eigenwerte zu berechnen. Der MR^3-Algorithmus basiert auf der Berechnung relativ robuster Repräsentationen (RRRs), welche Dhillon 1997 in seiner Doktorarbeit [28] wie in Definition 3.3.2 eingeführt hat.
Zunächst ist dafür aber eine weitere Definition notwendig.

Definition 3.3.1. Die Funktion **relgap** [28] bestimmt den kleinsten relativen Abstand zwischen einem Skalar $y \in \mathbb{R}$ und einer endlichen Menge $\mathbb{X}$ von Skalaren $x_i \in \mathbb{R}$:

$$\text{relgap}(y, \mathbb{X}) := \min_{x \in \mathbb{X}} \frac{\mid y - x \mid}{\mid y \mid}$$

Definition 3.3.2. Eine endliche Menge von reellen Zahlen $\mathbb{X} = \{x_i\}, x_i \in \mathbb{R}$ nennt man **relativ robuste Repräsentation** von A, wenn sie eine Matrix $A \in \mathbb{R}^{n \times n}$ vollständig so definiert, dass, falls die Zahlen x_i durch $x_i(1 + \epsilon_i)$ gestört werden, folgende Gleichungen für alle $1 \leq j \leq n$ gelten:

$$\frac{\mid \delta\lambda_j \mid}{\mid \lambda_j \mid} = \mathcal{O}\left(\sum_i \epsilon_i\right)$$

$$\mid \sin \angle(z_j, z_j + \delta z_j) \mid = \mathcal{O}\left(\frac{\sum_i \epsilon_i}{\text{relgap}(\lambda_j, \{\lambda_k \mid k \neq j\})}\right)$$

Dabei bezeichnet $\lambda_j \in \mathbb{R}$ den j-ten Eigenwert von A, wobei $\lambda_j + \delta\lambda_j \in \mathbb{R}$ den dazu gestörten Eigenwert darstellt. Der zugehörige Eigenvektor, bzw. gestörte Eigenvektor, wird mit $z_j \in \mathbb{R}^n$ bzw. $z_j + \delta z_j \in \mathbb{R}^n$ bezeichnet. Die Funktion $\text{relgap}(y, \mathbb{X})$ wurde in Definition 3.3.1 eingeführt.
Eine relativ robuste Repräsentation ist also eine Darstellung der Matrix, so dass alle Eigenwerte und -vektoren bei einer kleinen Störung der Matrixdarstellung ebenfalls nur kleine relative Störungen aufweisen. Ist dies nur für einen Teil der Eigenwerte und -vektoren gegeben, bezeichnet man die Darstellung als **partielle relativ robuste Repräsentation**.

Für den MR^3-Algorithmus werden Zerlegungen der um einen Shift $\mu \in \mathbb{R}$ verschobenen Tridiagonalmatrix genutzt:

$$LDL^\mathsf{T} = T - I\mu \tag{3.16}$$

Wenn der Wert μ so gewählt wird, dass die Matrix $T - I\mu$ positiv definit ist, also

$$\forall_{x \in \mathbb{R}^n \setminus \{\vec{0}\}} : x^\mathsf{T}(T - I\mu)x > 0$$

gilt, so handelt es sich bei der Zerlegung LDL^T um eine RRR [29]. Die Matrix $D \in \mathbb{R}^{n \times n}$ ist eine Diagonalmatrix und bei $L \in \mathbb{R}^{n \times n}$ handelt es sich um eine untere Bidiagonalmatrix, welche auf ihrer Diagonalen Einsen beinhaltet. Bei dieser Faktorisierung handelt es sich um eine Dreieckszerlegung, zum Beispiel die Cholesky Zerlegung [8]. Durch die Anwendung dieser Zerlegung auf eine Tridiagonalmatrix, bekommt die Matrix L statt einer Dreiecksgestalt sogar Bidiagonalform.

Die Eigenwerte der relativ robusten Repräsentation LDL^T lassen sich mittels Bisektion berechnen, welche eine Differential QD-Transformation (Quotienten-Differenzen-Transformation) nutzt [29]. Da es sich um eine positiv definite Matrix handelt, lassen sich die Eigenwerte auch über den dqds-Algorithmus (differential qd-algorithm) [30] berechnen.
Bezeichne nun $\hat{\mathbb{L}}$ die Menge aller berechneten Eigenwerte $\{\hat{\lambda}_i\}$ der RRR. Wenn man nicht alle Eigenwerte benötigt, reicht in diesem Schritt auch schon eine partielle RRR für die entsprechenden Eigenwerte aus. Sind die Eigenwerte in $\hat{\mathbb{L}}$ nicht geclustert, also der relative Abstand

$$\underset{1\leq i\leq n}{\forall} : \mathrm{relgap}(\hat{\lambda}_i, \hat{\mathbb{L}}\backslash\{\hat{\lambda}_i\}) > tol, \tag{3.17}$$

wobei *tol* eine bestimmte Toleranz, z. B. 10^{-3} [29], darstellt, dann können zu den Eigenwerten $\hat{\lambda}_i$, die Eigenvektoren $\hat{z}_i$ bestimmt werden, so dass sie ohne Nutzung des Gram-Schmidt-Verfahrens numerisch orthogonal zueinander stehen. Dies wird über die in [29] als $\texttt{getvec}(L, D, \hat{\lambda}_i)$ definierte Methode erreicht, welche für jedes $\hat{\lambda}_i \in \hat{\mathbb{L}}$ aufgerufen werden muss. Sie berechnet den zugehörigen Eigenvektor über weitere modifizierte QD-Transformationen und über die sogenannte verdrehte Faktorisierung (*twisted factorization*) [31].
Probleme treten nur dann auf, wenn die Eigenwerte in Clustern vorliegen, also die Bedingung (3.17) nicht erfüllt ist. Wie schon die Inverse Iteration, kann auch die Funktion `getvec` die Orthogonalität der Eigenvektoren dann nicht mehr sicherstellen. Daher ist vor der Berechnung der Eigenvektoren mittels `getvec` eine Gruppierung der Eigenwerte notwendig.
Die Indizes der berechneten Eigenwerte $\hat{\lambda}_i$ werden in $m \leq n$ Mengen Γ_c eingeteilt, welche die Cluster definieren. Dabei werden die isolierten Eigenwerte, welche einen großen relativen Abstand zu allen anderen haben, als eigene Cluster behandelt, wie es in Abbildung 3.3.1 für $n = 5$ und $m = 3$ exemplarisch dargestellt wird.
Nun wird jeder Cluster bzw. jede Menge Γ_c für $c = 1, \ldots, m$ einzeln behandelt. Falls der Cluster nur einen isolierten Eigenwert beschreibt, also

$$| \Gamma_c | = 1,$$

so kann über `getvec` direkt der zugehörige Eigenvektor berechnet werden.
In dem Beispiel aus Abbildung 3.3.1 wäre dies bei dem ersten Cluster der Fall. Die Menge $\Gamma_1 = \{1\}$ definiert den Cluster mit dem isolierten Wert $\hat{\lambda}_1$, wobei sich der zugehörige Eigenvektor $\hat{z}_1$ über $\texttt{getvec}(L, D, \hat{\lambda}_1)$ ergibt.
Handelt es sich bei Γ_c um die Beschreibung eines echten Clusters mit mehr als einem Wert, so kann mit der aktuellen relativ robusten Repräsentation LDL^T nicht weiter gerechnet werden. Es wird nun ein weiterer Verschiebungswert μ_c in der Nähe des Clusters ausgewählt und über die dstqds-Transformation [29] eine neue partielle relativ robuste Repräsentation

$$LDL^\mathsf{T} - \mu_c I = L_c D_c L_c^\mathsf{T}$$

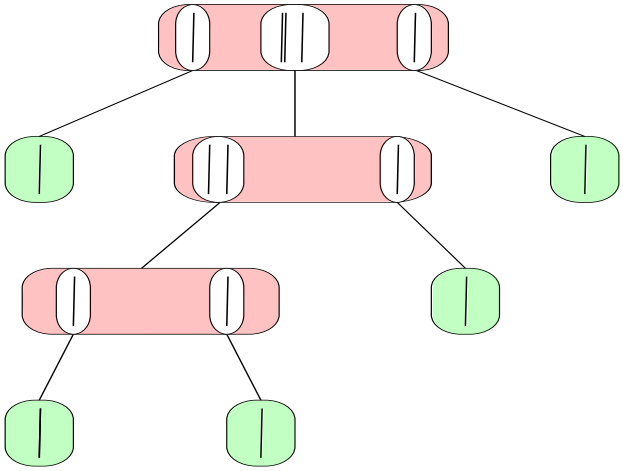


Abbildung 3.3.1: Die Eigenwerte einer Beispielmatrix $T \in \mathbb{R}^{5\times 5}$ wurden zu Beginn in drei Cluster eingeteilt. Der kleinste Eigenwert $\hat{\lambda}_1$ bildet einen eigenen Cluster, da es sich um einen isolierten Wert handelt. Die mittleren drei Eigenwerte $\hat{\lambda}_2, \hat{\lambda}_2$ und $\hat{\lambda}_3$ bilden einen Cluster, da der relative Abstand zwischen ihnen zu klein ist. Der Wert $\hat{\lambda}_5$ ist wiederum isoliert und bildet den dritten Cluster.

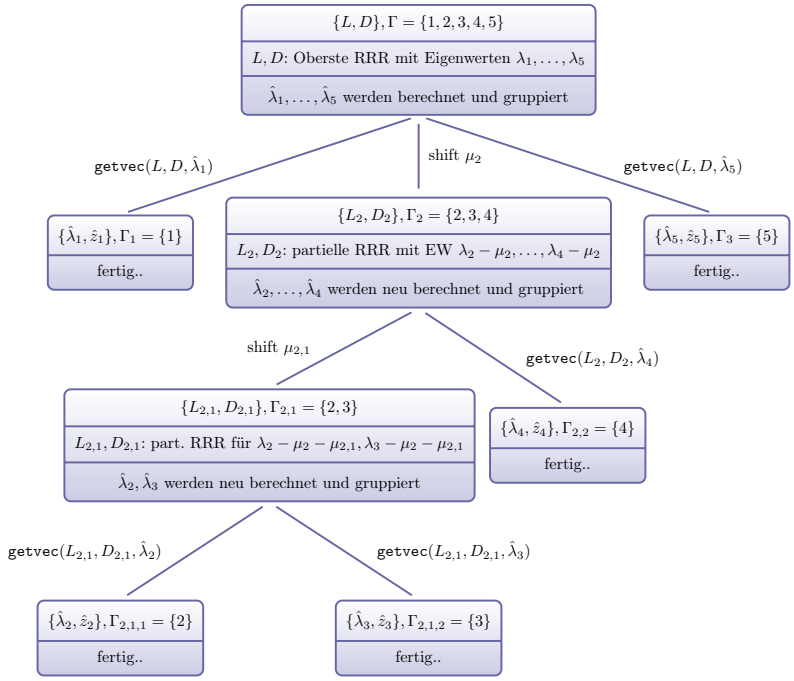


Abbildung 3.3.2: Repräsentationsbaum für eine Beispielmatrix $T \in \mathbb{R}^{5\times 5}$ mit insgesamt drei Cluster, wobei die beiden äußeren nur aus jeweils einem isolierten Eigenwert bestehen. Der mittlere Cluster enthält drei Eigenwerte, von den zwei eine Ebene tiefer immer noch in einem Cluster liegen, weshalb dieser Baum insgesamt aus vier Ebenen besteht.

berechnet. Nun hat $L_c D_c L_c^\mathsf{T}$ die gleichen Eigenvektoren zu den Werten $\hat{\lambda}_i - \mu_c$ wie die RRR LDL^T zu den Werten $\hat{\lambda}_i$ hat. Dies gilt allerdings nur für alle $i \in \Gamma_c$ bei exakter Arithmetik.
Die geforderten Eigenwerte der neuen partiellen RRR $L_c D_c L_c^\mathsf{T}$ werden allerdings neu berechnet, und haben durch die Verschiebung größere relative Abstände zueinander, wodurch die Eigenwerte aus dem Cluster gezogen werden. Die Struktur der Cluster und wie sie isoliert werden, wird mit den sogenannten Repräsentationsbäumen [28] dargestellt. Dabei stellt ein Knoten des Baumes immer einen echten Cluster dar, welcher weiter zerlegt werden muss. Die isolierten Eigenwerte werden, wie auch in Abbildung 3.3.1, als Blätter dargestellt.
Die Abbildung 3.3.2 zeigt passend zum vorherigen Beispiel einen Repräsentationsbaum der (5×5)-Matrix. Eine passende Matrix für beide Beispiele, die auch zur entsprechend dargestellten Baumtiefe führt, hätte zum Beispiel folgende Eigenwerte:

$$\lambda_1 = 0, \quad \lambda_2 = 1, \quad \lambda_3 = 1 + 10^{-15}, \quad \lambda_4 = 1 + 10^{-4}, \quad \lambda_5 = 2$$

Die Werte λ_2, λ_3 und λ_4 würden hierbei bei einer gewählten Toleranz von 10^{-3} einen Cluster bilden. Durch die Verschiebung mit z. B. $\mu = 1$ würde man über die neue partielle relativ robuste Repräsentation die Eigenwerte λ_2 und λ_3 zunächst wieder gemeinsam in einem Cluster behandeln.

Input : T, Γ, tol
1 Wähle μ so, dass $T - \mu I$ positiv definit ist
2 $L, D :=$`Cholesky_Zerlegung`$(T - \mu I)$
3 Berechne die gewünschten Eigenwerte $\hat{\lambda}_i$ für $i \in \Gamma$ von LDL^T mittels Bisektion oder dqds-Algorithmus und füge sie der Menge $\hat{\mathbb{L}}$ zu
4 $[\Lambda, Z] :=$`getvec_cluster`$(L, D, \Gamma, \hat{\mathbb{L}}, tol)$
5 $\Lambda = \Lambda + \mu I$
6 **return** $[\Lambda, Z]$

Algorithmus 3.3.5: Der MR^3-Algorithmus startet mit einer Tridiagonalmatrix T, einer Menge Γ von Indizes und einer Toleranzschranke *tol* und berechnet die gewünschte Menge Λ von Eigenwerten und die Menge Z mit den dazu passenden Eigenvektoren. Welche Eigenwerte bzw. Vektoren berechnet werden sollen, wird über Γ angegeben. Mit der berechneten RRR wird eine rekursive Funktion aufgerufen, welche in Algorithmus 3.3.6 dargestellt ist. Die Rückgabewerte Λ, Z stellen hier nur die Mengen dar, welche die gewünschten Eigenwerte und -vektoren beinhalten. Sie stellen nicht die Matrizen aus der Spektralzerlegung dar.

Die zum Schluss berechneten Eigenpaare der obersten RRR in der Abbildung 3.3.2 setzen sich wie folgt zusammen:

$$\begin{aligned}
(\lambda_1, z_1) &\approx (\hat{\lambda}_1, z_1) \\
(\lambda_2, z_2) &\approx (\hat{\lambda}_2 + \mu_{2,1} + \mu_2, z_2) \\
(\lambda_3, z_3) &\approx (\hat{\lambda}_3 + \mu_{2,1} + \mu_2, z_3) \\
(\lambda_4, z_4) &\approx (\hat{\lambda}_4 + \mu_2, z_4) \\
(\lambda_5, z_5) &\approx (\hat{\lambda}_5, z_5)
\end{aligned}$$

Um an die Eigenwerte der Matrix T zu gelangen, müsste nun der zu Beginn bei der Berechnung der obersten RRR ausgeführte Shift μ aus (3.16) auf die berechneten Eigenwerte der RRR aufaddiert werden. Bei den orthogonal bestimmten Eigenvektoren der RRR handelt es sich um die Eigenvektoren der Matrix T. Die folgt aus dem Satz 2.2.9, da es sich hier um eine geshiftete Matrix handelt.

Input : $L_p, D_p, \Gamma_p, \hat{\mathbb{L}}_p, tol$

1 Gruppiere die Eigenwerte $\hat{\lambda}_{low} \dots \hat{\lambda}_{up} \in \hat{\mathbb{L}}_p$ in m Cluster, wobei zwei aufeinander folgende Werte in einen gemeinsamen Cluster gesetzt werden, falls $\text{relgap}(\hat{\lambda}_i, \{\hat{\lambda}_{i+1}\}) < tol$ gilt. Besetze die m Mengen Γ_c jeweils mit den Indizes der Eigenwerte im c-ten Cluster

2 **for** $(c = 1, \dots, m)$ **do**

3 **if** $(\mid \Gamma_c \mid = 1)$ **then**

4 $j :=$ Index in Γ_c

5 $z_j :=$`getvec`$(L, D, \hat{\lambda}_j)$

6 **else**

7 Wähle μ_c in der Nähe des Clusters

8 $[L_c, D_c] :=$`dstqds`$(L_p D_p L_p^\mathsf{T} - \mu_c I)$

9 Berechne die Eigenwerte $\hat{\lambda}_i$ von $L_c D_c L_c^\mathsf{T}$ für alle $i \in \Gamma_c$ neu und füge sie der Menge $\hat{\mathbb{L}}_c$ zu

10 $[\hat{\Lambda}_c, Z_c] :=$`getvec_cluster`$(L_c, D_c, \Gamma_c, \hat{\mathbb{L}}_c, tol)$

11 $\hat{\lambda}_i := \hat{\lambda}_i + \mu_c$ für alle $i \in \Gamma_c$

12 **end**

13 **end**

14 **return** $[\Lambda_p, Z_p]$

Algorithmus 3.3.6: Die Funktion `getvec_cluster` berechnet die Eigenvektoren zu den schon angenäherten Eigenwerten in der Menge $\hat{\mathbb{L}}_p$. Dabei werden die in Clustern liegenden Eigenwerte rekursiv voneinander getrennt, um die Orthogonalität der über die Funktion `getvec` berechneten Eigenvektoren zu gewährleisten. Die Rückgabewerte Λ_p, Z_p stellen hier wieder Mengen dar, welche die Eigenwerte und Eigenvektoren zu den Indizes aus Γ_p beinhalten.

Für die Berechnung aller Eigenwerte und Vektoren benötigt der MR^3-Algorithmus $\mathcal{O}(n^2)$ Flops [32]. Die Komplexität sinkt auf $\mathcal{O}(nk)$ Flops für die Berechnung von k Eigenpaaren, was an die Bisektion mit der Inversen Iteration erinnert. Ein wichtiger Vorteil des MR^3-Algorithmus ist allerdings, dass die Berechnung im Worst Case, bei stark geclusterten Eigenwerten, weiterhin $\mathcal{O}(nk)$ Flops benötigt [27], wogegen der Worst Case bei der Inversen Iteration bei $\mathcal{O}(nk^2)$ Flops liegt.
Ein weiterer Vorteil gegenüber der Inversen Iteration wird durch die Orthogonalität der Eigenvektoren gegeben, welche vom MR^3-Algorithmus problemlos gewährleistet wird.

3.4 Rücktransformation

Nachdem die symmetrische Matrix $A \in \mathbb{R}^{n\times n}$ durch

$$Q^\mathsf{T} A Q = T$$

im ersten Schritt des Gesamtverfahrens auf die Tridiagonalgestalt reduziert wurde, und im zweiten Schritt alle bzw. nur ausgewählte Eigenpaare (λ_i, z_i) der Tridiagonalmatrix T berechnet wurden, müssen die Eigenvektoren z_i im dritten Schritt rücktransformiert werden, da sie im Allgemeinen nicht mit den Eigenvektoren v_i von A übereinstimmen.
Falls nur die Eigenwerte gewünscht sind, ist man an dieser Stelle schon fertig, da sich die Eigenwerte einer Matrix durch Ähnlichkeitstransformationen, welche zur Reduktion auf T genutzt wurden, nicht verändern.
Die Rücktransformation eines Eigenvektors ist definiert durch

$$v = Qz,$$

was direkt aus dem satz (2.2.7) für ähnliche Matrizen folgt. Für alle Eigenvektoren kann dies auch als Matrixmultiplikation

$$\Upsilon = QZ$$

dargestellt werden, wobei $Z \in \mathbb{R}^{n\times n}$ die Eigenvektoren $z_i \in \mathbb{R}^n$ der Tridiagonalmatrix T als Spalten und $\Upsilon \in \mathbb{R}^{n\times n}$ die Eigenvektoren $v_i \in \mathbb{R}^n$ der Matrix A als Spalten enthält. Die Transformationsmatrix Q ergibt sich aus dem Produkt der einzelnen Householdertransformationen, wie in (3.4) beschrieben wurde.

```
     Input : Z, H_1, ..., H_{n-2}
1    for (i = (n-2)..., 1) do
2        Z = H_i Z
3    end
4    return Z
```

Algorithmus 3.4.1: Rücktransformation der Eigenvektoren z, welche als Spalten der Matrix Z übergeben werden. Die Matrix Z wird mit den transformierten Eigenvektoren $v = H_1 \dots H_{n-2} z$ überschrieben und zurückgegeben.

Der Aufwand der Rücktransformation liegt bei $2n^2m$ Flops für insgesamt m zu transformierende Eigenvektoren. Diese Operationen können fast vollständig mit Level 3 BLAS durchgeführt werden, so dass die Rücktransformation effizient ausgeführt werden kann [9].

Kapitel 4

Rechnerarchitekturen und Bibliotheken

Dieses Kapitel beschäftigt sich mit den Rechnerarchitekturen, auf welchen die Eigenwertlöser zum Einsatz gekommen sind. Dabei handelt es sich um zwei Blue Gene Architekturen [33], welche von IBM [34] entwickelt wurden und in Abschnitt 4.1 beschrieben werden.
Des Weiteren werden in diesem Kapitel in Abschnitt 4.2 die Bibliotheken vorgestellt, welche die getesteten Eigenwertlöser beinhalten. Die dafür verwendeten Compiler und notwendige Software werden in Anhang A aufgelistet.

4.1 Blue Gene

Das Blue Gene Projekt wurde im Dezember 1999 von IBM mit der Vision initiiert, in der Proteinforschung auf PetaFLOPS[5]-Systemen rechnen zu können [35].
Der erste Rechner, welcher im Rahmen dieses Projekts entwickelt wurde, war der Blue Gene/L. Dieser sich noch in der Entwicklung befindende Superrechner wurde 2004 in der November Ausgabe der TOP500-Liste mit 16 Racks, 16.384 Knoten und insgesamt 32.768 Prozessorkernen auf Platz Eins eingestuft [36]. Die letzte Ausbaustufe umfasste 2008 104 Racks und somit insgesamt 212.992 Rechenkerne mit einer Taktfrequenz von jeweils 700 MHz.
Das Blue Gene/P System gilt als zweite Generation, welche im Rahmen dieses Projekts entstanden ist. Ein solches System steht unter dem Namen JUGENE [37] im Forschungszentrum Jülich [38] und wurde unter anderem für die Messungen in dieser Arbeit verwendet. Der JUGENE wird in Abschnitt 4.1.1 näher beschrieben.
Die dritte Generation in der Blue Gene Serie wird als Blue Gene/Q bezeichnet. Dieses System soll bis 2011 eine Peak-Performance von 20 PetaFLOPS erreichen [39]. Das Forschungszentrum Jülich hat Zugang zu einem Prototyp des Blue Gene/Q Sys-

[5]FLOPS steht für Floating Point Operations Per Second und ist daher nicht mit dem Plural von Flop (Floating Point Operation) zu verwechseln, was in dieser Arbeit mit Flops bezeichnet wird.

tems [40], so dass Messungen für diese Arbeit auf diesem System möglich waren. Die Blue Gene/Q Architektur wird in Abschnitt 4.1.2 beschrieben.

4.1.1 Blue Gene/P (JUGENE)

Mit dem JUGENE wurde ein IBM Blue Gene/P System im Jülich Supercomputing Centre (JSC) des Forschungszentrums Jülich installiert. Dieser belegte im November 2007 den zweiten Platz in der TOP500-Liste [41]. Schon ein Jahr später war er nicht mehr unter den zehn schnellsten Rechnern der Welt, so dass im Mai 2009 eine verbesserte Version des JUGENE-Systems eingeweiht wurde, welcher in der kommenden TOP500 den dritten Platz erreichte.
Die einzelnen Hardware-Komponenten des JUGENE-Systems sind in Abbildung 4.1.1

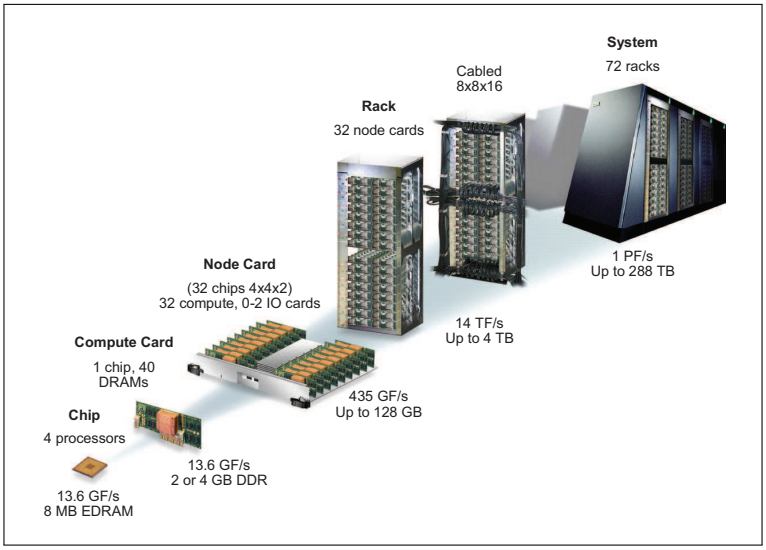


Abbildung 4.1.1: Schematischer Aufbau des Blue Gene/P Systems JUGENE, entnommen aus [42]. Die Einheit FLOPS wird hier mit F/s bezeichnet.

schematisch dargestellt und werden im Folgenden mit anderen charakteristischen Eigenschaften zusammengefasst:

Aufbau:

- 72 Racks
- 1024 Rechnerknoten pro Rack (insgesamt 73728 Knoten)
- 4 Rechnenkerne pro Knoten (insgesamt 294912 Kerne)

Prozessortyp:

- Power PC 450, 32-bit, 4-way SMP
- 850 MHz Taktfrequenz

- 2 GB Hauptspeicher (insgesamt 144 TB)
- 13, 6 GigaFLOPS Peak-Performance

Gesamte Rechenleistung:

- 1 PetaFLOPS Peak-Performance
- 825, 5 TeraFLOPS beim LINPACK-Benchmark [43]

Netzwerke:

- Dreidimensionaler Torus (für Punkt-zu-Punkt Kommunikation zwischen Knoten)
- vollständiger Binärbaum (für kollektive Operationen)
- Low Latency Global Barrier and Interrupt Netzwerk
- 10 Gigabit Ethernet (für I/O Kommunikation)
- 1 Gigabit Ethernet (als Kontrollnetzwerk)

Der Blue Gene/P hat, im Vergleich zum Blue Gene/L System mit zwei Kernen pro Knoten, durch den 4-way SMP vier Kerne auf jedem Rechnerknoten. Die Anzahl der Knoten pro Rack hat sich nicht verändert.
Bei der gesamten Peak-Performance von 1 PetaFLOPS handelt es sich wie bei jedem Superrechner um einen theoretischen Wert, welcher bei perfekter Ausnutzung der Leistung maximal erreicht werden kann [44]. Die Peak-Performance von 825, 5 TeraFLOPS gibt dabei die tatsächlich mit dem LINPACK-Benchmark erreichte Peak-Performance an, so dass der JUGENE dabei eine Effizienz von etwa 80% aufweist.

4.1.2 Blue Gene/Q

Das Blue Gene/Q System wird mit dem Ziel entwickelt bis zu 20 PetaFLOPS zu erreichen. Eine Referenzinstallation des Blue Gene/Q Systems wird am Lawrence Livermore National Laboratory [45] im Jahre 2011 im Rahmen des Advanced Simulation and Computing Program unter dem Namen IBM Sequoia eingerichtet. Es soll 96 Racks umfassen und insgesamt aus 98.304 Knoten mit 1, 6 Millionen Prozessorkernen bestehen [39].
Die Blue Gene/Q zielt insbesondere darauf ab eine höhere Leistung pro Watt zu erreichen. Die meisten zur Zeit neu entwickelten Systeme zielen inzwischen darauf ab, den Stromverbrauch zu verringern. In der Green500 Liste [46] werden die Superrechner eingestuft, welche die beste Peak-Performance pro Watt erreichen. In der Liste vom Juni 2011 belegen zwei Blue Gene/Q Prototypen vom IBM Thomas J. Watson Research Center [47] die ersten beiden Plätze.
Ein Rack wird dabei wie bei beiden Vorgängern 1024 Rechnerknoten umfassen, welche allerdings mehr Kerne mit jeweils höheren Taktraten enthalten. Im Fall von IBM Sequoia werden dies voraussichtlich 16 Kerne sein, wobei über die genaue Taktfrequenz bisher noch nichts bekannt ist.
Während das Hauptnetzwerk zwischen den einzelnen Knoten bei beiden Vorgängern

mittels eines dreidimensionalen Torus realisiert wurde, wird für das Blue Gene/Q System ein fünfdimensionaler Torus mit einer erhöhten Bandbreite genutzt [48].
Der Prototyp der Blue Gene/Q [40], zu dem das JSC einen Zugang hat, umfasst bisher nur 256 Rechnerknoten. Jeder Knoten beinhaltet 16 Kerne, die jeweils 4-way hyper-threaded (SMT) sind, sodass insgesamt bis zu 64 Prozesse pro Knoten gestartet werden können.
Dabei hat jeder Knoten einen Hauptspeicher von 4 GB, wobei dieser in der endgültigen Version des Blue Gene/Q Systems 8 oder sogar 16 GB betragen soll. Eine weitere Einschränkung beim Prototyp ist die Bandbreite des I/O Netzwerks, welche bisher nur $\frac{1}{16}$ der endgültigen Bandbreite beträgt. Die globale Kommunikation ist auf diesem System ebenfalls noch nicht optimiert, da die derzeitige Implementierung Punkt-zu-Punkt Kommunikation nutzt.

4.2 Bibliotheken

Die genutzten Bibliotheken sind auf parallelen Systemen mit verteiltem Speicher nutzbar. Sie stellen Routinen zur Lösung verschiedener Probleme der Linearen Algebra bereit. Bei der Beschreibung dieser, wird insbesondere auf den wesentlichen Unterschied zwischen den Bibliotheken, die Verteilung der Daten auf die verschiedenen Prozessoren, eingegangen.
Getestet wurden vier Routinen aus der Bibliothek ScaLAPACK [49], welche in Abschnitt 4.2.1 beschrieben wird.
In Abschnitt 4.2.2 wird die Bibliothek Elemental [50] vorgestellt, aus welcher eine Routine getestet wurde.

4.2.1 ScaLAPACK

ScaLAPACK (Scalable Linear Algebra Package) ist eine parallele Programmbibliothek, welche in Fortran 77 implementiert wurde. Sie stellt Routinen zur parallelen Lösung einiger Probleme der Linearen Algebra bereit. Für diese Arbeit wurde mit der ScaLAPACK Version 1.8 gearbeitet.

4.2.1.1 Aufbau

ScaLAPACK ist eine Weiterentwicklung der Bibliothek LAPACK (Linear Algebra Package) [51], welche sequentielle Routinen bereitstellt.
Ziel dieser Projekte war die schnellstmögliche Lösung von Standardproblemen der Linearen Algebra, wie die Lösung eines linearen Gleichungssystems oder eines Eigenwertproblems.
Dabei werden für die meisten auszuführenden Rechnungen die optimierten BLAS (Basic Linear Algebra Subprograms) [52] genutzt. Diese sind teilweise in Assembler implementiert und durch diese Nähe zur Hardware meist hochoptimiert. Es existieren zur Zeit mehrere Realisierungen der BLAS. Für die durchgeführten Rechnungen auf dem JUGENE bzw. dem Blue Gene/Q System wurde die ESSL (Engineering and Scientific Subroutine Library) [53] von IBM genutzt. Auf dem Prototyp des

Blue Gene/Q Systems ist die ESSL Bibliothek allerdings nur ein einer vorläufigen Version vorhanden, und daher noch nicht hochoptimiert.
Die BLAS beinhalten drei verschiedene Stufen von elementaren Operationen der Linearen Algebra. Die Level 1 BLAS enthalten Vektor-Rechenoperationen wie Normen oder das Skalarprodukt. Die zweite Stufe, die Level 2 BLAS, realisieren Vektor-Matrix-Rechenoperationen. Level 3 Blas beinhalten Matrix-Matrix-Rechenoperationen, zum Beispiel ein Matrizenprodukt.
Da ScaLAPCK auf parallelen Systemen mit verteiltem Speicher genutzt wird, muss eine Kommunikation zwischen den einzelnen Prozessoren gewährleistet werden, da ein Prozessor keinen direkten Zugriff auf den Hauptspeicher eines anderen Prozessors hat.
Für diese Art der Kommunikation existieren sogenannte Message Passing Bibliotheken. Als eine solche Bibliothek hat sich MPI (Message Passing Interface) [54] als Standard etabliert. Es existieren weitere Bibliotheken wie PVM (Parallel Virtual Machine) [55]. Solche Bibliotheken beinhalten sowohl Punkt-zu-Punkt, als auch kollektive Kommunikationen.
Die Kommunikation in ScaLAPACK wird allerdings über BLACS (Basic Linear Algebra Communication Subprograms) [56] realisiert, was speziell für die Anwendung in der Linearen Algebra zugeschnitten ist. Dabei werden die Prozessoren in ein virtuelles zweidimensionales Prozessorgitter angeordnet, worauf die Matrizen dann aufgeteilt werden, was in Abschnitt 4.2.1.3 näher beschrieben wird.

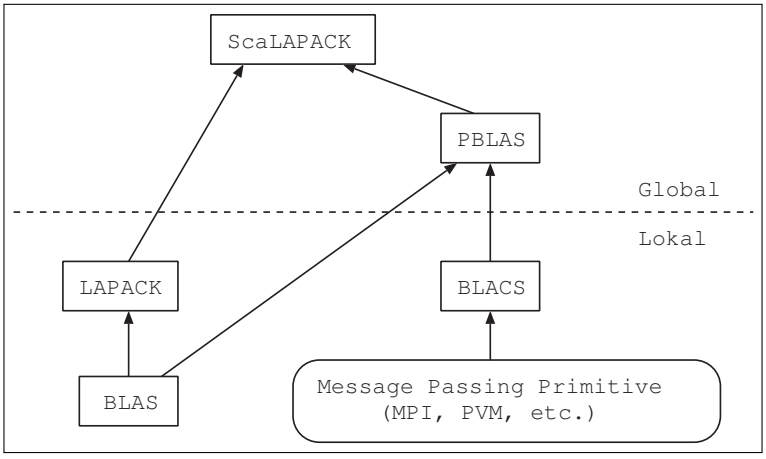


Abbildung 4.2.1: Software Hierarchie ScaLAPACKs, entnommen aus [49]. Die unteren Bibliotheken werden alle lokal auf einem Prozessor aufgerufen, wobei die Daten lokal vorhanden sein müssen. Nur ScaLAPACK und PBLAS (Parallel BLAS) werden global aufgerufen. PBLAS [57] ist dabei eine Weiterentwicklung von BLAS, welche die elementaren Rechenoperationen parallel ausführt.

4.2.1.2 Eigenwertlöser

ScaLAPACK stellt insgesamt vier verschiedene Routinen zur Lösung des symmetrischen Eigenwertproblems bereit. Diese Routinen, welche ein ganzes Problem mit dem Gesamtverfahren lösen, werden in ScaLAPACK als *driver routines* bezeichnet. Diese rufen wiederum andere Routinen aus ScaLAPACK bzw. LAPACK auf, die sogenannten *computational routines* [49].
Die Namensgebung aller Routinen ist in LAPACK/ScaLAPACK einheitlich geregelt, wie es am Beispiel `PDSYTRD` gezeigt wird.
Die parallelen Routinen aus ScaLAPACK beginnen immer mit `P`, während die sequentiellen direkt mit dem Datentyp beginnen. Im Beispiel ist dies ein `D`, welches für *Double Precision* steht.
Die folgenden zwei Buchstaben geben immer den Typ der Matrix an, mit welcher gerechnet wird. So steht z. B. `SY` für eine symmetrische Matrix. Die Art der Berechnung wird schließlich mit den letzten Buchstaben gekennzeichnet. Im Beispiel steht `TRD` für die Tridiagonalisierung.
Die wichtigsten genutzten *computational routines* und die vier *driver* Routinen zur Lösung des symmetrischen Eigenwertproblems, welche für diese Arbeit getestet wurden, werden im Folgenden kurz beschrieben.

Computational routines:

- `PDSYTRD` - Reduktion einer symmetrischen Matrix zu einer Tridiagonalmatrix.
- `PDSYNTRD` - Reduktion einer symmetrischen Matrix zu einer Tridiagonalmatrix. Dies ist eine Weiterentwicklung der Routine `PDSYTRD`. Bei dieser Routine wird die Tridiagonalisierung auf einem quadratischen Gitter ausgeführt, wodurch sich die Laufzeit reduzieren lässt. Dieser Ansatz beruht auf Hendricksons, Jessups und Smiths symmetrischem Eigenwertlöser HJS [58], welcher aber nicht offiziell veröffentlicht wurde und nur auf Intels Paragon [59] lief. Einige Information zur Reduktion der Laufzeit über Verringerung der Lastinbalancen, der Kommunikation und des Software Overheads sind in der Doktorarbeit [60] von Stanley zu finden.
- `PDORMTR` - Multiplikation der orthogonalen Matrix aus der Tridiagonalisierung mit einer gegeben Matrix. Diese Routine wird für die Rücktransformation der Eigenvektoren genutzt.
- `DSTEQR2` - Berechnung aller Eigenwerte und optional der Eigenvektoren einer symmetrischen tridiagonalen Matrix mittels des QR-Algorithmus. Diese Routine ist eine modifizierte Version von `DSTEQR` aus LAPACK, welche sequentiell auf den Prozessoren aufgerufen wird. Dafür sind die Prozessoren in einer eindimensionalen logischen Prozessorspalte statt in einem zweidimensionalen Gitter angeordnet. Die Matrix, welche nach der Berechnung die Eigenvektoren als Spalten beinhaltet, wird spaltenweise auf die Prozessoren verteilt. Die Tridiagonalmatrix, welche nur aus $2n-1$ Werten (Haupt- und Nebendiagonale) besteht, ist auf allen Prozessoren

bekannt, so dass jeder Prozessor seine lokalen Spalten, also Eigenvektoren, berechnen kann.

- `PDSTEDC` - Berechnung aller Eigenwerte und optional der Eigenvektoren einer symmetrischen tridiagonalen Matrix mittels des Divide-and-Conquer-Algorithmus von Cuppen.
- `PDSTEBZ` - Berechnung der gewünschten Eigenwerte einer symmetrischen Tridiagonalmatrix mittels der Bisektion.
- `PDSTEIN` - Berechnung der Eigenvektoren einer symmetrischen Tridiagonalmatrix zu gegebenen Eigenwerten mittels Inverser Iteration.
- `PDSYEGR` - Berechnung der gewünschten Eigenwerte und optional auch Eigenvektoren einer symmetrischen tridiagonalen Matrix durch Verwendung des MR^3-Algorithmus. Die Routine ist nicht im offiziellen Release von ScaLAPACK 1.8 vorhanden. Eine Dokumentation ist unter [32] zu finden.

***Driver routines* zur Lösung des symmetrischen Eigenwertproblems:**

- `PDSYEV`:
  - `PDSYTRD` (Reduktion)
  - Umverteilung der Tridiagonalmatrix auf eindimensionale Spaltenblöcke für `DSTEQR2`
  - `DSTEQR2` (QR-Algorithmus)
  - `PDORMTR` (Rücktransformation)
- `PDSYEVD`:
  - `PDSYTRD` (Reduktion)
  - `PDSTEDC` (Divide-and-Conquer-Algorithmus)
  - `PDORMTR` (Rücktransformation)
- `PDSYEVX`:
  - `PDSYNTRD` (Reduktion)
  - `PDSTEBZ` (Bisektion)
  - `PDSTEIN` (Inverse Iteration)
  - `PDORMTR` (Rücktransformation)
- `PDSYEVR` [32]:
  - `PDSYNTRD` (Reduktion)
  - `PDSYEGR` (MR^3-Algorithmus)
  - `PDORMTR` (Rücktransformation)

4.2.1.3 Datenverteilung

Die Daten müssen für die Nutzung von ScaLAPACK verteilt vorliegen. Jeder Prozessor hat bei einem parallelen System mit verteiltem Speicher seinen eigenen Hauptspeicher, auf den kein anderer Prozessor zugreifen kann. Bei MPI geht man im Allgemeinen davon aus, dass die logische Anordnung aller Prozessoren eindimensional ist und np Prozessoren mit einem Array

P_0	$\dots$	P_{np-1}

dargestellt werden können.
Speziell für die Lineare Algebra, wegen der Rechnungen mit Matrizen, eignet sich eine zweidimensionale logische Anordnung der Prozessoren. Dies wird in ScaLAPACK mit BLACS realisiert. Für die np Prozessoren gilt dann

$P_{0,0}$	$\dots$	$P_{0,nc-1}$
$\vdots$	$\ddots$	$\vdots$
$P_{nr-1,0}$	$\dots$	$P_{nr-1,nc-1}$

,

wobei $np = nr * nc$ gilt.
Eine Matrix $A \in \mathbb{R}^{n \times n}$ muss für die Nutzung von ScaLAPACK in Blöcke aufgeteilt und auf die Prozessoren verteilt werden. Man spricht von einer zweidimensionalen blockzyklischen Verteilung. Die Blockgrößen nb_{row}, nb_{col} für beide Dimensionen kann

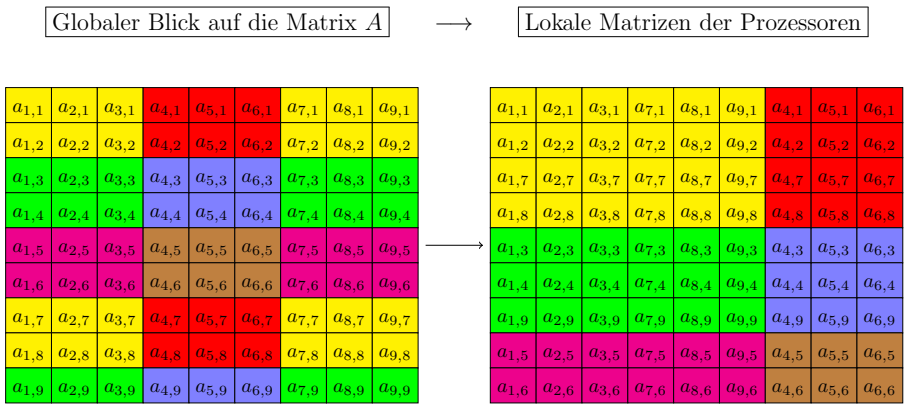


Abbildung 4.2.2: Beispiel der zweidimensionalen blockzyklischen Verteilung einer Matrix $A \in \mathbb{R}^{9 \times 9}$ auf insgesamt sechs Prozessoren (mit verschiedenen Farben dargestellt), welche in einem (3×2)-Gitter angeordnet sind. Die bei ScaLAPACK anzugebenden Blockgrößen nb_{row} bzw. nb_{col} wurden auf 2 bzw. 3 gesetzt.

man als Nutzer selbst bestimmen und bei der Datenverteilung angeben. Da es sich bei Eigenwertproblemen immer um quadratische Matrizen handelt und das Prozessorgitter, wenn möglich ebenfalls quadratisch gewählt wird, macht eine Unterscheidung

bei der Blockgröße zwischen beiden Dimension nicht viel Sinn. Daher ist eine Unterscheidung der beiden Blockgrößen bei den Eigenwertlösern in ScaLAPACK nicht implementiert [49]. Es gilt

$$nb_{row} = nb_{col}$$

für die Eigenwertlöser von ScaLAPACK. Die optimale Blockgröße kann auf verschiedenen Systemen variieren.
Im Allgemeinen unterscheidet man zwischen zwei Arten von Blockgrößen. In dieser Arbeit bezeichnet im Folgenden nb_{dist} die physikalische Blockgröße, die angibt wie die Matrix auf die Prozessoren verteilt wird. Die andere Blockgröße ist für die Rechnung mit Blöcken zuständig. Sie wird häufig als *computational block size* oder als algorithmische Blockgröße bezeichnet. Im Folgenden wird sie mit nb_{algo} gekennzeichnet.
Die Blockgröße nb_{algo} wird zum Beispiel für die Aufteilung einer Matrix in Blöcke genutzt, wenn mit ihr Blockalgorithmen ausgeführt werden. Dies wird häufig gemacht, da man, falls nb_{algo} entsprechend gewählt wird, ganze Blöcke in den Cache laden und mit ihnen die Operationen ausführen kann. Wenn nur Blöcke statt der vollständigen Matrix in den Cache passen, ist die blockweise Berechnung oft effizienter, da die Daten nicht immer wieder neu geladen werden müssen. Für die blockweise Berechnung werden die BLAS genutzt.
Die algorithmische Blockgröße nb_{algo} ist in ScaLAPACK abhängig von der physikalischen Blockgröße der Matrixverteilung nb_{dist} [61], und es gilt:

$$nb_{dist} = nb_{algo}.$$

Je größer die gewählte Blockgröße ist, desto größere Blöcke werden mit Level 3 BLAS Operationen ausgewertet. Für manche Algorithmen, z. B. für Zerlegungen, wird zusätzlich der Kommunikationsaufwand zwischen den Prozessoren durch die Nutzung größerer Blöcke reduziert.
Eine zu große Blockgröße hat den Nachteil, dass es zu einer schlechten Lastverteilung kommen kann, wie das kleine Beispiel aus Abbildung 4.2.2 schon verdeutlicht. Der Prozessor $P_{0,0}$ (gelb) hat lokal eine (4×6)-Matrix, also 24 Elemente, gespeichert, wohingegen der Prozessor $P_{2,1}$ (braun) nur eine (2×3)-Matrix und damit 6 Elemente im Speicher hat.
Eine Blockgröße von $nb_{dist} = 1$ würde zu einer perfekten Lastverteilung, aber auch zu viel Kommunikation führen. Außerdem würden wegen der Abhängigkeit zu nb_{algo} für die Updates der Matrixelemente während eines Verfahrens keine Level 3 BLAS genutzt, da jedes Element einzeln aktualisiert wird.

4.2.2 Elemental

Elemental ist eine parallele Bibliothek, welche Routinen zur parallelen Lösung vieler Probleme aus der Linearen Algebra bereitstellt. Es handelt sich bei Elemental um eine in C++ implementierte Bibliothek, welche sich allerdings noch in der Entwicklung befindet. Für diese Arbeit wurde mit der Elemental Version 0.66 gearbeitet.

4.2.2.1 Aufbau

Elemental nutzt die Objektorientierung, welche durch die Implementierung in C++ möglich ist. So werden alle Vektoren und Matrizen mittels Objekten realisiert.
Die Erzeugung und Verteilung einer Matrix auf die Prozessoren geschieht über Konstruktoraufrufe. Der Aufruf `DistMatrix<double,MC,MR> A(`n, m`,grid)` würde z. B. eine $(n \times m)$-Matrix des Datentyps `double` erzeugen und auf dem Prozessorgitter, welches durch das Objekt `grid` beschrieben wird, verteilen. `[MC,MR]` ist eine Angabe zur Art der Verteilung der Matrix, welche in Abschnitt 4.2.2.3 beschrieben wird.
Neben der Verteilung sind Algorithmen zur Lösung einiger Probleme aus der Linearen Algebra implementiert. Diese werden mit `elemental:advanced:function(...)` aufgerufen und sind Vergleichbar mit den Routinen aus ScaLAPACK oder LAPACK. Mit `elemental:basic:function(...)` werden Routinen zur Rechnung mit Matrizen aufgerufen. Diese rufen im wesentlichen die jeweiligen BLAS Routinen für diese Berechnung auf, da diese optimiert sind. Daher muss für die Rechnungen auf dem JUGENE bzw. dem Blue Gene/Q System ebenfalls die ESSL Bibliothek vorhanden sein.
Die Kommunikation zwischen den einzelnen Prozessoren wird über eigene Objekte mit Funktionen realisiert, welche aber im wesentlich auch MPI Funktionen aufrufen. Die Objekte regeln die logische Anordnung der Prozessoren, welche für die Lineare Algebra sinnvollerweise zweidimensional ist. Diese Objekte sind vergleichbar mit den in ScaLAPACK genutzten BLACS.

4.2.2.2 Eigenwertlöser

Elemental stellt eine Routine zur Lösung des symmetrischen Eigenwertproblems bereit, welche auf dem MR^3-Algorithmus basiert und damit mit ScaLAPACKs `PDSYEVR` vergleichbar ist. Diese und die anderen wichtigsten genutzten Routinen werden nachfolgend kurz aufgelistet und beschrieben. Sie sind, außer in `basic` (BLAS) und `advanced` (LAPACK), nicht weiter unterteilt, werden hier aber passend zu den *computational* und *driver* Routinen aus LAPACK und ScaLAPACK aufgeführt.

Computational routines:

- `elemental:advanced:Tridiag` - Reduktion einer symmetrischen Matrix zu einer Tridiagonalmatrix.
- `elemental:advanced:UT` - Multiplikation der orthogonalen Matrix aus der Tridiagonalisierung mit einer gegeben Matrix. Diese Routine wird für die Rücktransformation der Eigenvektoren genutzt.
- `pmrrr` - Berechnung der gewünschten Eigenwerte und Eigenvektoren einer symmetrischen tridiagonalen Matrix mittels des MR^3-Algorithmus. Diese Routine wurde unabhängig von Elemental in C entwickelt und ist online auf [62] zu finden. Elemental nutzt sie als externe Funktion bei der Lösung des symmetrischen Eigenwertproblems.

***Driver routine* zur Lösung des symmetrischen Eigenwertproblems:**

- `elemental:advanced:HermitianEig`:
  - `elemental:advanced:Tridiag` (Reduktion)
  - `pmrrr` (MR^3-Algorithmus)
  - `elemental:advanced:UT` (Rücktransformation)

4.2.2.3 Datenverteilung

In [63] wurde 1999 folgendes Fazit aufgeführt:

> „Block storage is not necessary for block algorithms and level 3 performance. Indeed, the use of block storage leads to a significant load imbalance when the blocksize is large. This is not a concern on the Paragon, but may be problematic for machines requiring larger blocksizes for optimal BLAS performance."

Darüber hinaus wurde in [64] experimentell gezeigt, dass eine kleine Blockgröße für die Verteilung von Matrizen vorteilhaft ist, falls sie unabhängig von der algorithmischen Blockgröße gewählt werden kann. Dies wird von Elemental als Ansatz zur Datenverteilung auf die Prozessoren genutzt [65].
Elemental nutzt eine zweidimensionale zyklische Verteilung, allerdings werden keine Blöcke, sondern einzelne Elemente aufgeteilt. Die Aufteilung ist vergleichbar mit ScaLAPACKs Aufteilung, falls man die Blockgröße $nb_{dist} = 1$ wählen würde.
Dies würde bei ScaLAPACK zu einer ineffizienten Berechnung führen, da die Level 3 BLAS nur sehr ineffizient genutzt würden, weil die algorithmische Blockgröße nb_{algo} wegen der Abhängigkeit ebenfalls 1 betragen würde. In Elemental gilt stattdessen immer

$$1 = nb_{dist} \neq nb_{algo}.$$

Die Abbildung 4.2.3 zeigt die Aufteilung der (9×9)-Matrix aus dem Beispiel der ScaLAPACK Aufteilung in Abbildung 4.2.2. Bei der elementweisen Verteilung hat man eine bessere Lastverteilung auf die Prozessoren, da jeder Prozessor eine annähernd gleich große Matrix abspeichert, was bei der Blockverteilung nicht der Fall war. Dazu kann man meistens davon ausgehen, dass man die algorithmische Blockgröße größer wählen kann, als dies bei ScaLAPACK der Fall ist, da sie bei Elemental nicht zu einer schlechten Lastverteilung führt.
Die algorithmische Blockgröße wird in Elemental als Parameter behandelt, welche sogar während eines Programms auf den jeweiligen Schritt angepasst werden kann, wofür in ScaLAPACK eine teure Umverteilung der Daten notwendig wäre.
In Elemental wird das Prozessorgitter über Objekte realisiert, welche über Konstruktoren erzeugt werden. Man kann zwischen verschiedenen Arten der Verteilung auswählen, und diese dem Konstruktor als Parameter übergeben. Beim Aufruf von `DistMatrix<double,MC,MR> A(`n, m`,grid)` sorgt `[MC,MR]` für die Verteilung, welche in Abbildung 4.2.3 dargestellt ist. Das erste Argument spricht die Spalten der

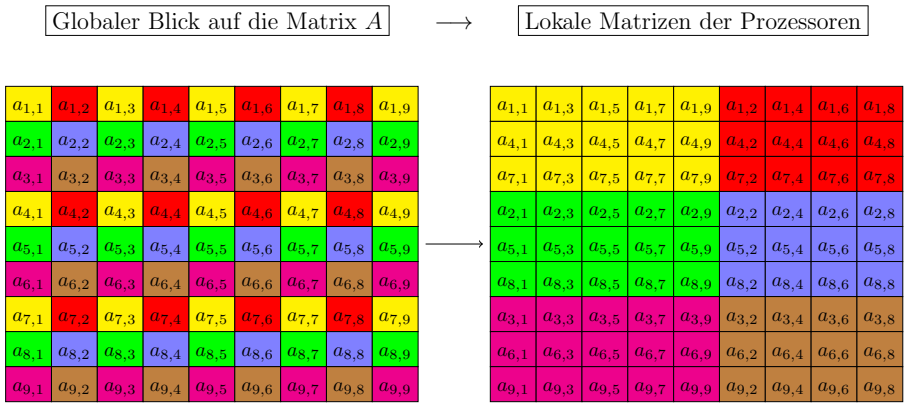


Abbildung 4.2.3: Beispiel der von Elemental genutzten zweidimensionalen zyklischen Verteilung einer Matrix $A \in \mathbb{R}^{9\times 9}$ auf insgesamt sechs Prozessoren (mit verschiedenen Farben dargestellt), welche in einem (3×2)-Gitter angeordnet sind.

Matrix an, und verteilt sie hier wegen `MC` als Spalten über die Prozessorspalten. Das zweite Argument spricht die Zeilen an, und verteilt sie hier als Zeilen. Bei der Angabe von `[MC,MR]` würde die Matrix transponiert verteilt werden.
Weitere Möglichkeiten zur Angabe sind `VC, VR` und `Star`. Die beiden ersten verteilen die Matrix wie einen Vektor, wobei `VC` spaltenweise und `VR` zeilenweise über die Prozessoren im zweidimensionalen Gitter läuft. Der Parameter `Star` gibt an, dass die Daten auf jedem Prozessor vorliegen sollen und gar nicht verteilt werden.
Einige Beispiele zu den verschiedenen Verteilungen sind im Anhang von [65] zu finden.

Kapitel 5

Performance-Analyse und -Optimierung

Im ersten Abschnitt dieses Kapitels wird erklärt, welche Testmatrizen für die Eigenwertlöser erzeugt werden, und wie dies geschieht. In Abschnitt 5.2 wird beschrieben, wie die Ergebnisse der Routinen auf ihre Korrektheit überprüft werden. Der Abschnitt 5.3 beschäftigt sich mit der Performance-Analyse und Optimierung der untersuchten Eigenwertlöser auf den jeweiligen Systemen, insbesondere durch Anpassung der verschiedenen Blockgrößen.

5.1 Testmatrizen

Wie in Kapitel 3 erwähnt, reagieren viele Algorithmen unterschiedlich auf die Lage der Eigenwerte. Insbesondere treten häufig Probleme auf, wenn die Eigenwerte sehr nah beieinander liegen. Man spricht in diesem Fall von geclusterten Eigenwerten.
Um diesen Fall gesondert zu betrachten, werden grundsätzlich zwei Arten von Matrizen $A \in \mathbb{R}^{n \times n}$ verwendet, solche mit einem großen Cluster von Eigenwerten und solche mit isolierten Eigenwerten.

- ***Geclustert:*** $\lambda_1, \ldots, \lambda_{n-1} \in [-10^{-5}, 10^{-5}]$ (gleichverteilt) und $\lambda_n = 1$
- ***Isoliert:*** $\lambda_1, \ldots, \lambda_n \in [0, 1]$ (gleichverteilt)

Zur Generierung dieser Testmatrizen werden zunächst die Eigenwerte von einem Zufallsgenerator erzeugt. Die Diagonalmatrix mit den generierten Eigenwerten auf der Diagonalen wird anschließend mit einer zufälligen Householdertransformation auf eine voll besetzte Matrix gebracht. Da es sich hierbei wieder um Ähnlichkeitstransformationen handelt, werden die Eigenwerte dadurch nicht verändert.
Der Quellcode zur Generierung der Matrix aus einem Vektor mit gegebenen Eigenwerten ist als Beispiel für Elemental (C++) im Anhang B als Funktion `dsyemagsc` zu finden.
Die Routinen, welche es ermöglichen nur einen Teil der Eigenwerte bzw. -vektoren zu berechnen, wurden zusätzlich zur Berechnung aller Eigenwerte und -vektoren

darauf getestet, nur 10% der Eigenwerte und Eigenvektoren zu berechnen. Tests, bei denen nur die Eigenwerte berechnet werden, wurden nicht durchgeführt, da die Eigenvektoren für die meisten Anwendungen ebenfalls von Interesse sind.

5.2 Korrektheit der Ergebnisse

Bei allen durchgeführten Tests wurden die Ergebnisse nach der Berechnung auf ihre Korrektheit überprüft. Dazu wurden insgesamt drei verschiedene Normen berechnet, welche Aufschluss darüber geben. Die drei Normen werden im Folgenden kurz vorgestellt.

$$\|\Lambda_{input} - \Lambda\|_\infty \tag{5.1}$$

$$\|A_{input}Q - Q\Lambda\|_\infty \tag{5.2}$$

$$\|Q^\mathsf{T}Q - I\|_\infty \tag{5.3}$$

Bei allen drei Normen handelt es dich um die Maximumsnorm $\|\cdot\|_\infty$, welche für eine Matrix $A \in \mathbb{R}^{n\times n}$ wie folgt definiert ist:

$$\|A\|_\infty = \max_{1\leq i,j\leq n} \mid a_{i,j} \mid$$

Die erste Norm (5.1) testet nur die berechneten Eigenwerte auf Korrektheit, indem sie mit den vorgegebenen Eigenwerten, aus denen die Matrix erzeugt wurde, verglichen werden. Diese könnte insbesondere genutzt werden, wenn nur die Eigenwerte berechnet werden, und die Eigenvektoren nicht zur Verfügung stehen.
Falls die Vektoren ebenfalls berechnet werden, kann auch die zweite Norm (5.2) genutzt werden. Diese überprüft, ob die Gleichung der Spektralzerlegung eingehalten wird.
Die dritte in (5.3) dargestellte Norm überprüft die Eigenvektoren auf Orthogonalität.

5.3 Parameteroptimierung

Parameter wie die Blockgröße *nb* (siehe Abschnitte 4.2.1.3 und 4.2.2.3) beeinflussen die Performance der Routinen, weshalb es zu Beginn einer Messreihe notwendig ist, diese so gut wie möglich einzustellen. Bei ScaLAPACK ist die algorithmische Blockgröße gleich der Blockgröße der gewählten Aufteilung einer Matrix auf die Prozessoren, wodurch die algorithmische Blockgröße fixiert ist. Eine Umverteilung zur Anpassung der algorithmischen Blockgröße würde einen zu großen Aufwand bedeuten. Daher gilt es für die Routinen aus ScaLAPACK die Blockgröße zu Beginn richtig einzustellen. Die Parameterstudie dazu wird in Abschnitt 5.3.1 beschrieben. Speziell für die Routine `PDSYEVX` existiert ein Parameter `ORFAC`, welcher der Routine

übergeben werden kann. Da die hier genutzte Inverse Iteration bei einem mehrfachen Eigenwert, was einem Cluster entspricht, keine orthogonalen Eigenvektoren liefert, muss zusätzlich eine Nachorthogonalisierung für die entsprechenden Eigenvektoren ausgeführt werden, wie in Abschnitt 3.3.2 beschrieben wurde.
Der Parameter `ORFAC` bestimmt welche Eigenwerte als Cluster definiert sind und damit, für welche Eigenvektoren die Nachorthogonalisierung ausgeführt wird. Gilt für zwei Eigenwerte λ_i, λ_j der Matrix A

$$| \lambda_i - \lambda_j | \leq \texttt{ORFAC} \|A\|_\infty$$

so werden sie als Cluster interpretiert. Die entsprechenden Vektoren v_i und v_j werden nachorthogonalisiert. Der Defaultwert für `ORFAC` liegt bei 10^{-3}. `PDSYEVX` führt keine Nachorthogonalisierung aus, falls der Wert auf 0 gesetzt wird. Alle folgenden Messungen wurden mit `ORFAC` $= 10^{-4}$ ausgeführt, da die numerische Orthogonalität der Eigenvektoren mit diesem Wert noch in einem guten Bereich liegt. Die Ausführung mit `ORFAC` $= 10^{-3}$ liefert keine wesentlich besseren Ergebnisse, ist aber deutlich langsamer.
In Elemental ist die Blockgröße der Aufteilung fest gleich 1 gewählt. Dafür kann man die algorithmische Blockgröße zu jeder Zeit des Programms durch die Funktion `elemental::SetBlocksize(int blocksize)` anpassen.
Die lokale Blockgröße für die Level 2 BLAS Funktion zur Berechnung des Matrix-Vektor-Produkts mit einer symmetrischen Matrix (`SYMV`) kann zu jeder Zeit durch `elemental::basic::SetLocalSymvBlocksize<double>(int blocksize)` eingestellt werden. Tatsächlich wird die Blockgröße hier für andere Level 2 BLAS Operationen (`TRMV`, `GEMV`) gesetzt, da diese Operationen für die Tridiagonalisierung genutzt werden.
Für die Tridiagonalisierung der dichtbesetzten symmetrischen Matrix kann zwischen zwei verschiedene Algorithmen gewählt werden, welche in Elemental als Tuningparameter behandelt werden. Dieser Parameter lässt sich über die Funktion `elemental::advanced::internal::SetTridiagApproach(...)` einstellen. Eins der beiden folgenden Argumente muss der Funktion dazu übergeben werden:

- `elemental::advanced::internal::TRIDIAG_NORMAL`
- `elemental::advanced::internal::TRIDIAG_SQUARE`

Der Algorithmus aus `TRIDIAG_NORMAL` - im Folgenden nur mit `TRI_NORMAL` bezeichnet - kann auf einem beliebigen zweidimensionalen Prozessorgitter laufen, während die zweite Variante immer ein quadratisches Prozessorgitter verwendet.
Die zweite Auswahl muss weiter spezialisiert werden. Es existiert eine Funktion `elemental::advanced::internal::SetTridiagSquareGridOrder(...)` in der man angibt, wie die Prozessoren für das Tridiagonalisierungsverfahren in einem zweidimensionalen quadratischen Gitter angeordnet werden. Es gibt zwei mögliche Argumente:

- `elemental::advanced::internal::COL_MAJOR`

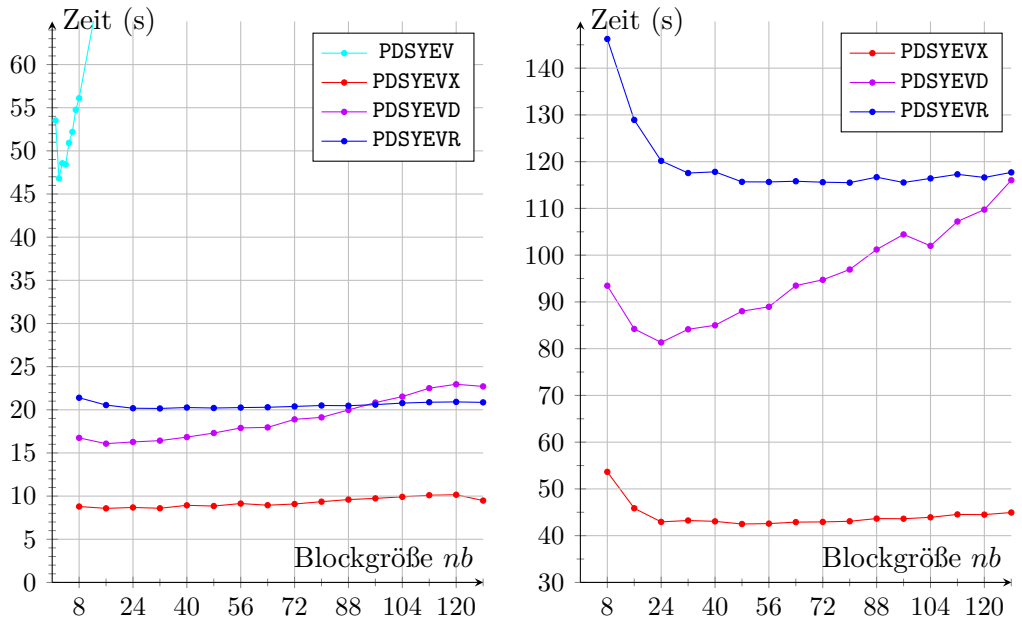
- `elemental::advanced::internal::ROW_MAJOR`

Man kann also ein spalten- oder zeilenweise angeordnetes quadratisches Gitter erzeugen. Im Folgenden werden die Algorithmen abkürzend mit `TRI_SQ_COL` bzw. `TRI_SQ_ROW` bezeichnet. Wie später zu sehen ist, ist ist die Nutztung von `TRI_SQ_ROW` in den meisten Situationen leicht schneller, was eventuell auf die zeilenweise Speichung der Daten in C++ zurückzuführen ist. Das grundsätzliche Verhalten beider Algorithmen ist allerdings ähnlich, weshalb die Betrachtung von `TRI_SQ_ROW` ausgereicht hätte.
Falls die Anzahl der Prozessoren bei Verwendung dieser Algorithmen keine Quadratzahl ist, wird das größtmögliche quadratische Gitter genutzt, wodurch insgesamt weniger Prozessoren zum Einsatz kommen und eine Umverteilung der Daten notwendig ist. Der Algorithmus für das quadratische Prozessorgitter geht auf den Ansatz von Hendrickson, Jessup und Smith [58] zurück, und ist mit ScaLAPACKs neuer Reduktionsroutine `PDSYNTRD` zu vergleichen, während `TRI_NORMAL` die entsprechende Version von `PDSYTRD` darstellt. Der Algorithmus für das quadratische Prozessorgitter nutzt lokale `TRMV`-Operationen (Dreiecksmatrix-Vektor-Produkt) während `TRI_NORMAL` viele `GEMV`-Operationen (Matrix-Vektor-Produkt) nutzt, um die Householderreduktion zu realisieren. Ein weiterer wesentlicher Unterschied liegt in der Kommunikation. Auf dem quadratischen Prozessorgitter werden häufiger einzelne Punkt-zu-Punkt Kommunikationen genutzt als dies für `TRI_NORMAL` der Fall ist. Bei der Tridiagonalisierung mittels `TRI_NORMAL` nutzt Elemental größtenteils globale Kommunikation. Diese Unterschiede sind in Abschnitt 6.5 über die Analyse mittels Scalasca [66] zu sehen.
Die Parameterstudie zur Routine aus Elemental ist in Abschnitt 5.3.2 zu finden.
Für beide Bibliotheken wurden zur Blockgrößenanalyse auf JUGENE folgende Referenzmatrizen mit einer festen Prozessoranzahl betrachtet:

- Matrixgröße $n = 25600$ mit 1024 Knoten mit jeweils 4 Kernen auf einem logischen Prozessorgitter von 64×64
- Matrixgröße $n = 8192$ mit 1024 Knoten mit jeweils 4 Kernen auf einem logischen Prozessorgitter von 64×64

5.3.1 ScaLAPACK - JUGENE

Die fest gewählte Blockgröße für die Verteilung und die Algorithmen, wird im Folgenden mit nb bezeichnet. In diesem Abschnitt wird die Auswirkung der Blockgröße auf die Laufzeit untersucht.
Für die beiden Referenzmatrizen wurden die Laufzeiten aller Routinen mit den verschiedenen Blockgrößen $8, 16, \ldots, 128$ gemessen.
In Abbildung 5.3.1 wird die Untersuchung der Blockgrößen für die vier ScaLAPACK Routinen dargestellt.
In Abbildungsteil (a) ist zu erkennen, dass `PDSYEV` (QR-Algorithmus) wesentlich langsamer als die anderen Routinen ist. Für diese Routine lassen sich außerdem nur sehr kleine Blockgrößen wählen, da sie sonst noch langsamer wird. Der Anstieg der



(a) Matrixgröße $n = 8192$ (isoliert)

(b) Matrixgröße $n = 25600$ (isoliert)

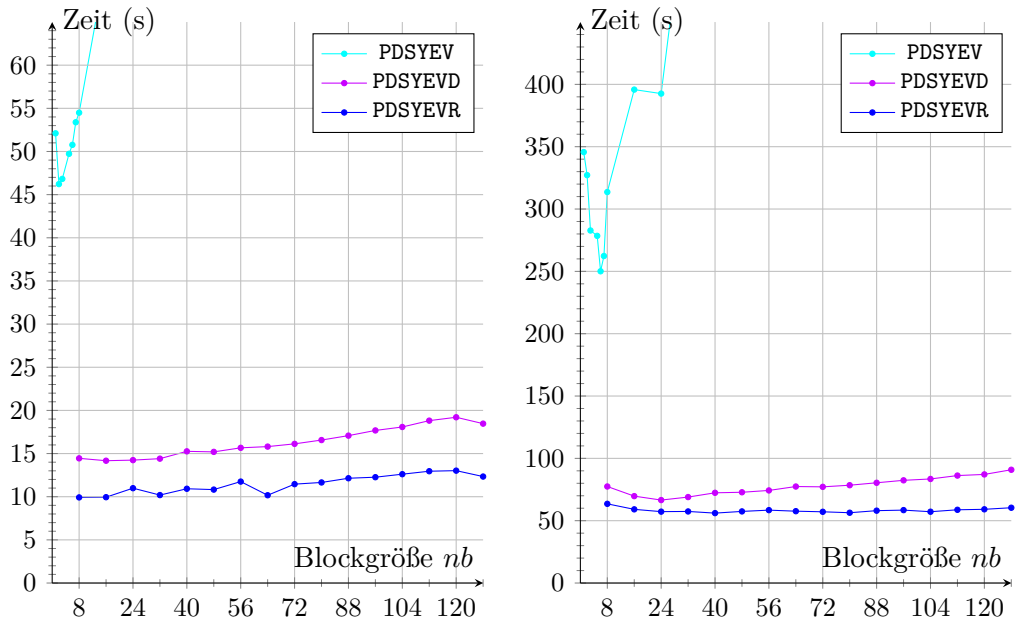
Abbildung 5.3.1: Laufzeiten der Routinen `PDSYEV`, `PDSYEVX`, `PDSYEVD` und `PDSYEVR` aus ScaLAPACK bei verschieden gewählten Blockgrößen nb auf JUGENE. Die beiden getesteten Matrizen haben jeweils isolierte Eigenwerte, welche zwischen 0 und 1 gleichverteilt sind. Die Routine `PDSYEV` ist nur mit der kleineren Matrix gemessen worden, weshalb sie im Abbildungsteil (b) nicht auftritt.

Laufzeit ist ab Blockgröße $nb = 4$ linear. Die optimale Blockgröße wäre für `PDSYEV` gleich 2. Dies liegt an der lokalen Berechnung der Eigenwerte der Tridiagonalmatrix auf jedem Prozessor, wofür die Daten in eindimensionale Spaltenblöcke umverteilt werden. Jeder Prozessor berechnet dann jeweils nb Eigenwerte. Damit alle Prozessoren an der Rechnung beteiligt sind muss die Blockgröße

$$nb \leq \frac{n}{np} = \frac{8192}{4096} = 2$$

sein. Daher ist für dieses Beispiel die Blockgröße 2 optimal.
Da die Messung für die große Matrix in Teil (b) sehr lange gedauert hätte, und die Routine `PDSYEV` nicht die Relevanz der anderen Routinen hat, sind in Teil (b) nur die drei anderen Routinen zu sehen.
Die Routine `PDSYEVX` (Bisektion & Inverse Iteration) ist für beide Matrizen mit jeder getesteten Blockgröße die schnellste der Routinen. Bei der kleinen Matrix in Teil (a) sieht man, dass die Blockgrößen 16 und 32 zu den kürzesten Zeiten führen. Bei größeren Blockgrößen steigen die Laufzeiten langsam an.
Bei der größeren Matrix in Teil (b) hat sich das Minimum der Laufzeiten etwas nach rechts verschoben, sodass die Blockgröße $nb = 48$ als optimal erscheint.

Die dritte Routine, `PDSYEVD` (Divide-and-Conquer), zeigt die Auswirkungen der Blockgrößen auf die Performance deutlicher. Für die kleine Matrix ist $nb = 16$ als deutliches Minimum zu erkennen.
In Teil (b) ist das Minimum, welches hier bei $nb = 24$ liegt noch besser zu sehen, und die Auswirkung auf die Laufzeit ist erheblich. Alleine der Unterschied zwischen Blockgröße 24 und 16 liegt bei etwa 4 Sekunden. Für größere Blöcke wird die Performance viel schlechter, sodass die Laufzeit mit einer Blockgröße von 128 etwa 35 Sekunden und damit über 43% länger als mit der optimalen Blockgröße ist.
Die letzte betrachtete Routine, `PDSYEVR` (MR^3-Algorithmus), ist wiederum stabiler, was die Laufzeiten mit den größeren Blockgrößen betrifft. Optimal erscheint $nb = 32$ bei der kleinen Matrix und $nb = 80$ bei der großen Matrix.
Hierbei ist es noch wichtig zu erwähnen, dass diese Routine mit nur 1024 statt 4096 Prozessoren lief, und die Laufzeiten in Bezug zu den anderen Routinen, hier nicht vergleichbar sind. Bei mehr als 1024 Prozessoren werden die Laufzeiten von `PDSYEVR` wesentlich länger, obwohl sie dann eigentlich kürzer werden müssten. Dies lässt vermuten, dass sich noch ein Fehler in dieser Routine befindet, da sie auch noch nicht im offiziellen Release von ScaLAPACK 1.8 enthalten ist.

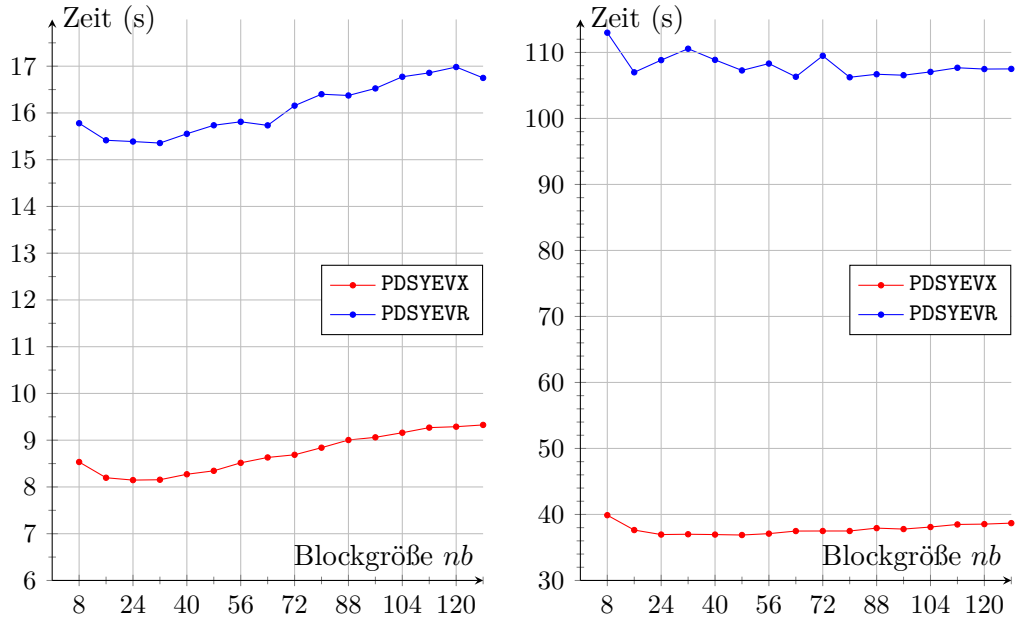


(a) Matrixgröße $n = 8192$ (geclustert)

(b) Matrixgröße $n = 25600$ (geclustert)

Abbildung 5.3.2: Laufzeiten der Routinen `PDSYEV`, `PDSYEVD` und `PDSYEVR` aus ScaLAPACK bei verschieden gewählten Blockgrößen nb auf JUGENE. Die beiden getesteten Matrizen haben jeweils $n - 1$ Eigenwerte in einem Cluster um 0 herum, und einen Eigenwert bei 1.

In Abbildung 5.3.2 wurde die Blockgrößenanalyse speziell für geclusterte Eigenwerte gemacht. Selbst bei der kleineren Matrix mit einer Dimension von 8192 und mit dem Parameter `ORFAC`= 10^{-4} kann die Routine `PDSYEVX` keine Ergebnisse liefern. Die bei Clustern notwendige Nachorthogonalisierung wird nur auf einem Prozessor ausgeführt [49]. Dies kann man auch mittels Scalasca in Abbildung 6.5.1 erkennen. Die Berechnung mit nur einem Prozessor verschlechtert zum Einen die Performance, wie man später in Teil (a) der Abbildung 6.2.3 bei kleineren Matrizen sieht, und zum anderen tritt bei größeren Matrizen das Problem auf, dass zu wenig Speicher auf dem genutzten Prozessor vorhanden ist.
Die Routine `PDSYEV` zeigt hinsichtlich der Performance keine Reaktion auf den Cluster, was beim Vergleich der Abbildung 5.3.1 mit 5.3.2 zu erkennen ist. Dies liegt am genutzten QR-Algorithmus. Das bedeutet allerdings, dass diese Routine auch bei geclusterten Eigenwerten sehr langsam und nicht konkurrenzfähig ist. Die Auswirkung der verschiedenen Blockgrößen ist ebenfalls absolut identisch zu den Matrizen mit isolierten Werten.
`PDSYEVD`, welche den Divide-and-Conquer Algorithmus nutzt, scheint bei den geclus-



(a) Matrixgröße $n = 8192$ (isoliert)

(b) Matrixgröße $n = 25600$ (isoliert)

Abbildung 5.3.3: Laufzeiten der Routinen `PDSYEVX` und `PDSYEVR` aus ScaLAPACK bei verschieden gewählten Blockgrößen nb auf JUGENE. Die beiden getesteten Matrizen haben jeweils isolierte Eigenwerte, von denen 10% berechnet wurden.

terten Eigenwerten etwa 15% schneller zu sein, als bei den isolierten. Die Auswirkung der Blockgrößen ist allerdings wieder absolut identisch, so dass sich der gleiche Kurvenverlauf mit den gleichen Minima erkennen lässt.

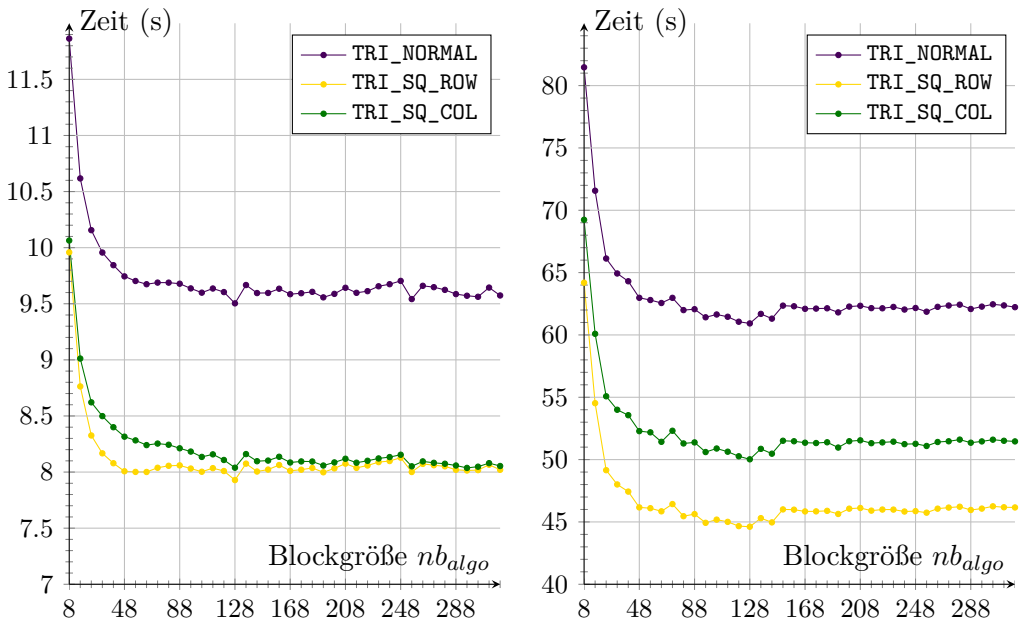
Die letzte getestete Routine `PDSYEVR` konnte mit geclusterten Matrizen mit 4096 Prozessoren gemessen werden, da hier das oben beschriebene Problem nicht auftritt. Daher ergibt sich in dem Fall kein Vergleich zu den isolierten Eigenwerten, worauf aber später bei den Ergebnissen noch eingegangen wird. Die Blockgrößen haben, insbesondere bei der großen Matrix, keine große Auswirkung auf die Performance. Bei der kleinen Matrix lässt sich $nb = 32$ als Minimum und damit optimale Blockgröße bestimmen.

Abbildung 5.3.3 zeigt die Analyse der Blockgrößen, wenn nur 10% der Eigenwerte und -vektoren berechnet werden, was in der Praxis häufig gefordert ist. Die einzigen beiden Routinen aus ScaLAPACK, bei denen diese Option vorgesehen, sind `PDSYEVX` und `PDSYEVR`. Dies liegt an den genutzten Algorithmen, wie in Abschnitt 3.3 beschrieben wurde. Daher wurden wieder die Matrizen mit den isolierten Eigenwerten betrachtet, da sonst nur die Messung von `PDSYEVR` möglich gewesen wäre.
Das Verhalten von `PDSYEVX` bei verschiedenen Blockgrößen ist identisch mit dem Verhalten aus Abbildung 5.3.1, als alle Eigenwerte berechnet wurden. Daher sind auch die optimalen Blockgrößen wieder 32 für die kleinere bzw. 48 für die große Matrix.
Die Routine `PDSYEVR` wurde wieder mit 4096 Prozessoren ausgeführt, was bei den isolierten Eigenwerten, wenn alle berechnet werden, noch zu Problemen geführt hatte. Hier scheint $nb = 32$ optimal für die kleine Matrix zu sein, was auch bei beiden vorherigen Untersuchungen mit geclusterten Eigenwerten (Abbildung 5.3.2) oder mit weniger Prozessoren (Abbildung 5.3.1) für die Matrix mit der Dimension 8192 optimal war.
Für die Matrix der Dimension 25600 ist in diesem Fall eine Blockgröße von 80 optimal, was ebenfalls mit den vorherigen Untersuchungen übereinstimmt.
Daher kann man vermuten, dass die optimale Blockgröße für diese Routine eher von der Matrixgröße abhängt, als von der Lage der Eigenwerte, der Prozessoranzahl oder der Anzahl zu berechnender Eigenwerte.
Die Genauigkeit der Lösungen wird nicht durch die Wahl der Matrixgröße beeinflusst. Alle drei in Abschnitt 5.2 definierten Normen liefern jeweils gute Ergebnisse. So ergibt die maximale Abweichung zu den Eingabewerten, Norm (5.1), Werte im Bereich von 10^{-15}. Der Test für die Korrektheit der Eigenwerte zu den -vektoren, Norm (5.2), und der Test für die Orthogonalität der Eigenvektoren, Norm (5.3), liefern jeweils Werte im Bereich von 10^{-12}.
Da die Ergebnisse immer ähnlich gut sind, wird im Folgenden, auch in Kapitel 6, nicht mehr weiter darauf eingegangen.

5.3.2 Elemental - JUGENE

Zu Beginn der Parameterstudie für Elemental auf JUGENE wurden alle Tridiagonalisierungsmöglichkeiten mit verschiedenen algorithmischen Blockgrößen für die zwei Referenzmatrizen auf einem festen Prozessorgitter getestet. Bei Elemental wurde allerdings mit den Blöcken $8, 16, \ldots, 320$ gemessen. Die lokale Blockgröße für die Level 2 BLAS Operationen nb_{symv} wird zunächst bei 48 festgehalten, und später einzeln

analysiert.



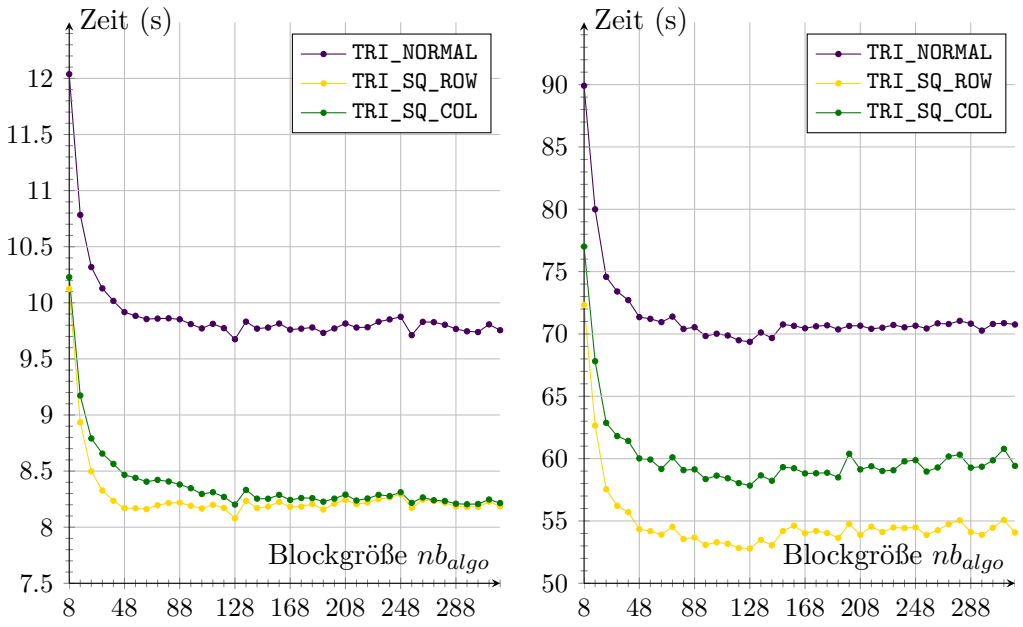
(a) Matrixgröße $n = 8192$ (isoliert)

(b) Matrixgröße $n = 25600$ (isoliert)

Abbildung 5.3.4: Laufzeiten der Routine `HermitianEig` aus Elemental mit den drei verschiedenen Tridiagonalisierungen bei verschieden gewählten Blockgrößen nb_{algo} auf JUGENE. Die beiden getesteten Matrizen haben jeweils isolierte Eigenwerte, welche zwischen 0 und 1 gleichverteilt sind.

In Abbildung 5.3.4 wird die Untersuchung der Blockgrößen für die Referenzmatrizen dargestellt. Es ist zu erkennen, dass kleine Blockgrößen in allen Fällen sehr schlecht sind, was bei ScaLAPACK mit kleinen Matrizen nicht der Fall ist. Dies liegt daran, dass bei ScaLAPACK die kleinen Blockgrößen vorteilhaft für die Verteilung der Matrizen sind und somit den Nachteil bei den Algorithmen ausgleichen. Erst bei großen Matrizen wirken sich die kleinen Blockgrößen schlecht auf die Performance aus, wie in Abbildung 5.3.1 gut zu sehen ist.
Da in Elemental die Verteilung elementweise realisiert wird, führen große algorithmische Blöcke nicht zu einer schlechten Lastverteilung. Der Vorteil größerer Blöcke für BLAS Operationen kann also genutzt werden, ohne die Lastverteilung zu beeinflussen.
Es ist also klar, dass Elemental wesentlich besser mit größeren Blockgrößen arbeiten kann, als dies bei ScaLAPACK der Fall ist. Daher wurden die Messungen auch bis zu einer Blockgröße von $nb_{algo} = 320$ durchgeführt.
Bei der kleineren Matrix in Abbildungsteil (a) erkennt man zwei deutliche lokale Minima bei den Blockgrößen 128 und 256, wobei die Performance für $nb_{algo} = 128$

am besten ist.
Wenn man die Zeiten für die große Matrix in Teil (b) betrachtet, erkennt man ebenfalls die Blockgröße 128 als optimalen Wert. Ein Minimum bei $nb = 256$ ist nur bei genauem Hinsehen zu entdecken, da dies bei der größeren Matrix nicht so stark ausgeprägt ist, wie in Abbildungsteil (a).
Insgesamt scheint die Größe 128 eine sehr gute Wahl zu sein, die auch für alle drei verschiedenen Algorithmen der Tridiagonalisierung optimal ist.



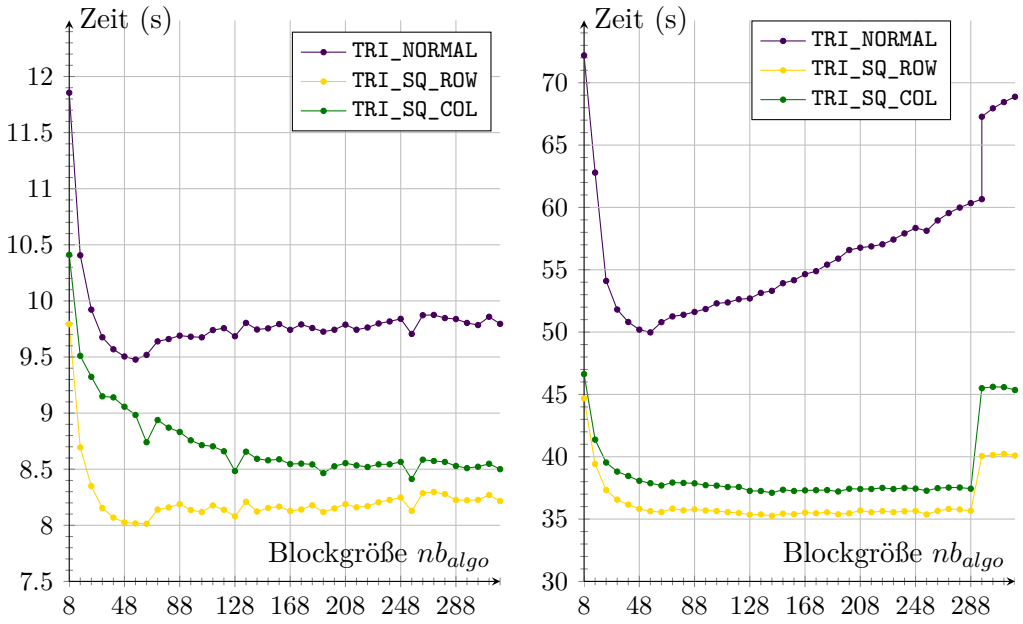
(a) Matrixgröße $n = 8192$ (geclustert)

(b) Matrixgröße $n = 25600$ (geclustert)

Abbildung 5.3.5: Laufzeiten der Routine `HermitianEig` aus Elemental mit den drei verschiedenen Tridiagonalisierungen bei verschieden gewählten Blockgrößen nb_{algo} auf JUGENE. Die beiden getesteten Matrizen haben jeweils $n - 1$ Eigenwerte in einem Cluster um 0 herum, und einen Eigenwert bei 1.

In Abbildung 5.3.5 wurde die Performance der Matrizen mit geclusterten Eigenwerten bei verschiedenen Blockgrößen untersucht.
Hierbei fallen einem nicht viele Unterschiede zu Abbildung 5.3.4 auf, außer dass die Gesamtdauer der Routinen länger ist, was an der Auflösung der Cluster liegt. Die Blockgröße 128 ist wieder in allen Fällen die optimale Wahl.
Bei der großen Matrix in Teil (b) sind noch mehrere und teilweise breitere lokale Minima, insbesondere bei `TRI_SQ_ROW` und `TRI_SQ_COL`, zu finden. Da aber insgesamt auch hier eine steigende Tendenz für die Laufzeit mit noch größeren Blöcken zu erkennen ist, wurden keine weiteren Tests mit noch größeren Blöcken ausgeführt.

Daher kann man sich insgesamt auch für die geclusterten Eigenwerte auf die Blockgröße 128 festlegen.



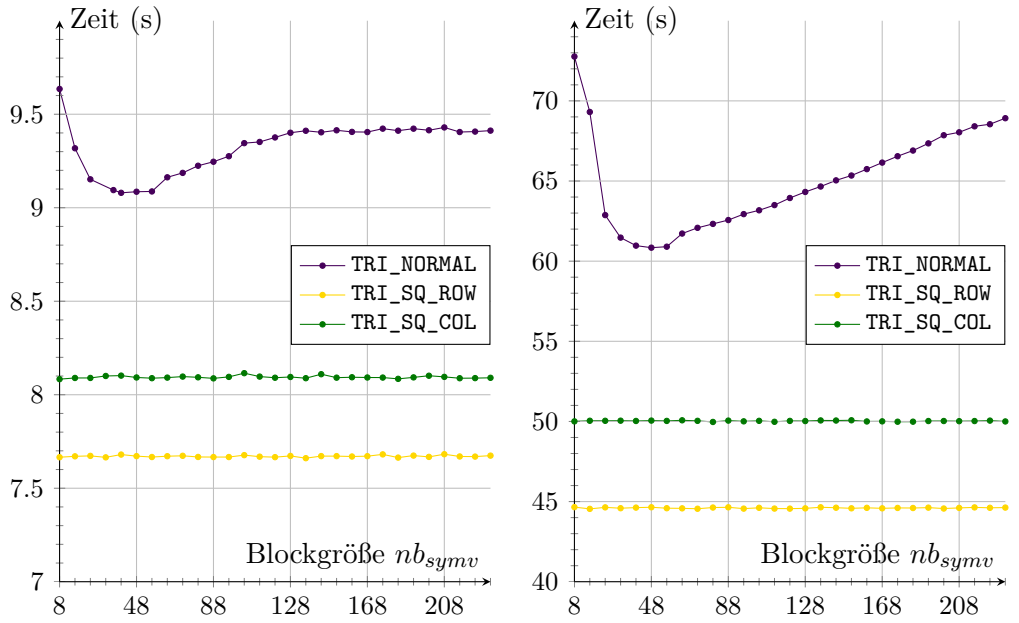
(a) Matrixgröße $n = 8192$ (isoliert)

(b) Matrixgröße $n = 25600$ (isoliert)

Abbildung 5.3.6: Laufzeiten der Routine `HermitianEig` aus Elemental mit den drei verschiedenen Tridiagonalisierungen bei verschieden gewählten Blockgrößen nb_{algo} auf JUGENE. Die beiden getesteten Matrizen haben jeweils isolierte Eigenwerte, von denen 10% berechnet werden.

Bisher wurden immer alle Eigenwerte berechnet. Nachfolgend wird die Auswirkung der Blockgrößen auf die Laufzeit getestet, wenn nur 10% der Eigenwerte und -vektoren berechnet werden. Das Ergebnis ist in Abbildung 5.3.6 dargestellt. Hier sind nun einige Unterschiede zu den vorherigen Abbildungen zu erkennen.
In Abbildungsteil (a) erkennt man für die kleinere Matrix mit `TRI_NORMAL` und `TRI_SQ_ROW` ein relativ breites lokales Minimum um die Blockgröße 56 herum. Dieses lässt sich auch als globales Minimum, und damit als optimale Blockgröße, deuten, da die Tendenz der zwei Kurven in den größeren Bereichen steigend ist. Die beste Performance für `TRI_NORMAL` wird mit $nb_{algo} = 56$ und für `TRI_SQ_ROW` mit $nb_{algo} = 64$ erreicht. Auffällig ist, dass die grundsätzliche Struktur mit den zwei lokalen Minima bei 128 und 256 auch hier ansatzweise gegeben ist.
`TRI_SQ_COL` hat dagegen einen anderen Verlauf. Hier ist eine fallende Tendenz zu erkennen, so dass eventuell auch Blöcke mit $nb > 320$ sinnvoll sein könnten. In dem getesteten Bereich von 8 bis 320 treten drei stark ausgeprägte lokale Minima bei den 2-er Potenzen $64, 128$ und 256 auf, wobei $nb_{algo} = 256$ am günstigsten ist.

Bei der großen Matrix ist für `TRI_NORMAL` ebenfalls ein deutliches Minimum bei $nb_{algo} = 56$ zu sehen, womit diese Blockgröße optimal ist. Die Performance nimmt bei Vergrößerung der Blöcke rapide ab.
Für die anderen beiden Algorithmen ist die Blockgröße 64 auch bei der großen Matrix gut. Hier sind allerdings auch größere Blöcke bis zu 288 auf einem ähnlichen bzw. teilweise sogar leicht besserem Niveau.
Mit Blockgrößen $nb \geq 296$ werden die Routinen wesentlich langsamer. Ein solcher Sprung in der Laufzeit ist für die Nutzung von `TRI_NORMAL` zwischen den Größen 296 und 304 gegeben. Wodurch dieses Verhalten in der Laufzeit begründet ist, und insbesondere warum dies nur bei der Berechnung von 10% der Eigenwerte und -vektoren auftaucht, konnte bisher nicht geklärt werden.



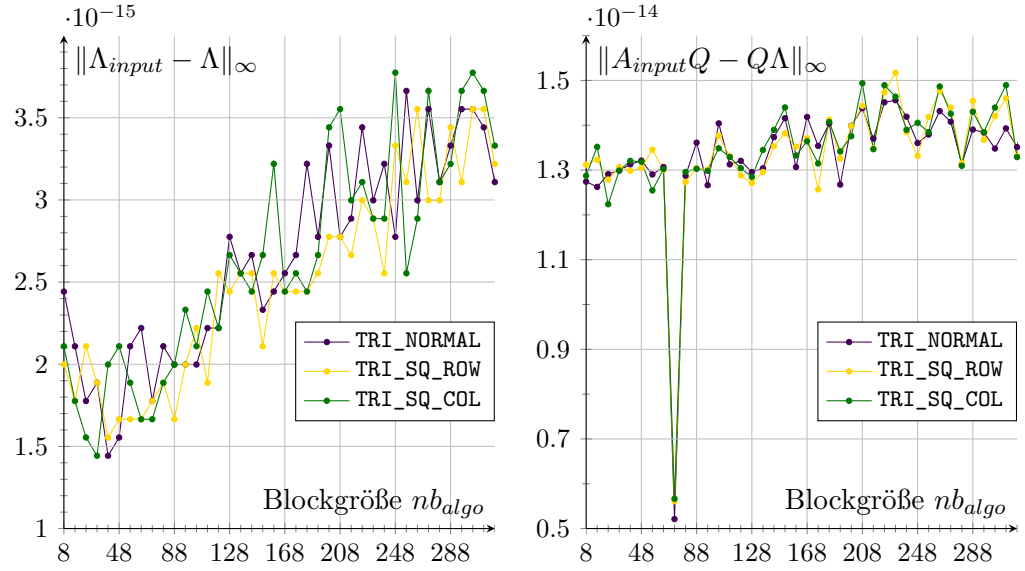
(a) Matrixgröße $n = 8192$ (isoliert)

(b) Matrixgröße $n = 25600$ (isoliert)

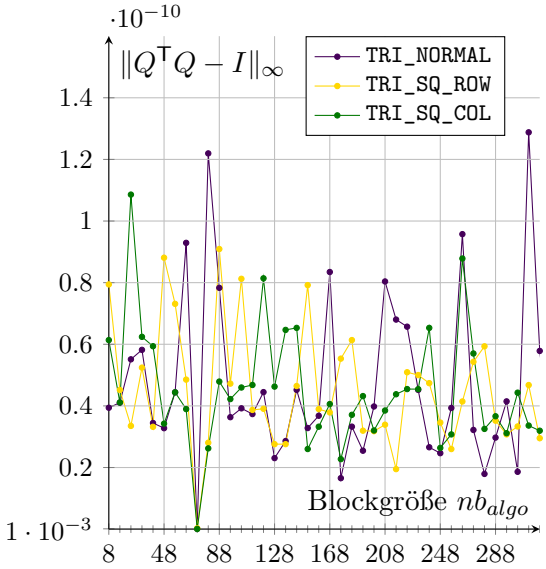
Abbildung 5.3.7: Laufzeiten der Routine `HermitianEig` aus Elemental mit den drei verschiedenen Tridiagonalisierungen bei verschieden gewählten lokalen Blockgrößen nb_{symv} auf JUGENE. Die beiden getesteten Matrizen haben jeweils isolierte Eigenwerte, welche alle berechnet werden.

In Abbildung 5.3.7 ist die Untersuchung der lokalen Blockgröße nb_{symv} für die Level 2 BLAS Operation zu sehen. Dabei wurde die algorithmische Blockgröße nb_{algo} mit 128 festgehalten, welche schon als optimal identifiziert wurde. Die betrachteten Matrizen haben isolierte Eigenwerte. Die Messungen wurden für die lokalen Blockgrößen $8, 16, \ldots, 232$ ausgeführt.
Die Auswirkungen von nb_{symv} auf die Performance sind für die Matrix mit gecluster-

ten absolut identisch zu denen mit isolierten Eigenwerten, sodass darauf hier nicht weiter eingegangen wird.



(a) Matrixgröße $n = 8192$ (isoliert), Norm 5.1 (b) Matrixgröße $n = 8192$ (isoliert), Norm 5.2



(c) Matrixgröße $n = 8192$ (isoliert), Norm 5.3

Abbildung 5.3.8: Verschiedene Normen zur Überprüfung der Ergebnisse von `HermitianEig` aus Elemental mit den drei verschiedenen Tridiagonalisierungen bei verschieden gewählten Blockgrößen nb_{algo}.

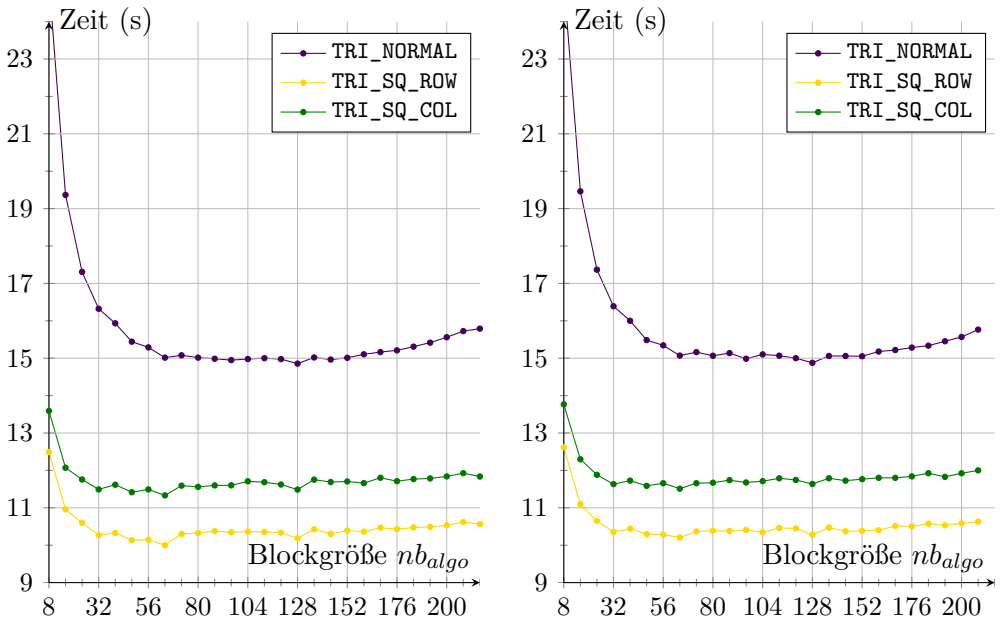
Die Blockgröße nb_{symv} beeinträchtigt die Laufzeit von `HermitianEig` erheblich, wenn für die Tridiagonalisierung `TRI_NORMAL` genutzt wird. Hier ist ein breites Minimum zu sehen, sodass die Blockgrößen $40, 48$ und 56 als optimal betrachtet werden können.
Wenn die anderen beiden Algorithmen zur Reduktion verwendet, scheint die Laufzeit gar nicht abhängig von dieser Größe zu sein. Dies liegt daran, dass das Matrix-Vektor-Produkt `GEMV` für `TRI_NORMAL` wesentlich häufiger aufgerufen wird, als dies für die anderen beiden Algorithmen der Fall ist. Hier werden nur wenige `TRMV`s statt der vielen `GEMV`s aufgerufen, wie schon erwähnt wurde und in der Scalasca Analyse in Abschnitt 6.5 zu erkennen ist.
Insgesamt kann man sich für weitere Messungen auf die Blockgröße $nb_{symv} = 48$ festlegen. Bei der Berechnung von nur 10% der Eigenwerte und -vektoren wirkt sich die Blockgröße nb_{symv} nicht auf die Performance aus.

Die Genauigkeit der Ergebnisse hängt in Elemental minimal von der gewählten algorithmischen Blockgröße nb_{algo} ab. Zur Vollständigkeit sind die Ergebnisse zu der Referenzmatrix mit der Dimension 8192 in Abbildung 5.3.8 dargestellt.
Die maximale Abweichung zwischen den generierten und den berechneten Eigenwerten ist in Abbildungsteil (a) eingezeichnet. Für `TRI_NORMAL` und `TRI_SQ_ROW` ist die Abweichung bei der Blockgröße 40 am kleinsten. Für `TRI_SQ_ROW` scheint $nb_{algo} = 32$ zu den besten Ergebnissen zu führen. Auffällig ist, dass insgesamt eher die kleineren Blockgrößen genauere Ergebnisse liefern. Diese waren für die Performance allerdings am schlechtesten.
Die Abweichung in der Spektralzerlegung wird in Teil (b) dargestellt. Hierbei sind auch eher kleinere Blockgrößen vorteilhaft, wobei die Blockgröße $nb_{algo} = 72$ im Gegensatz zu allen anderen, die mit Abstand besten Ergebnisse liefert. Alle drei Algorithmen verhalten sich dabei ähnlich.
Dies ist auch in Abbildungsteil (c) der Fall, wo die Metrik für die Orthogonalität dargestellt wird. Auch hier liefert eine Blockgröße von 72 deutlich bessere Ergebnisse, im Vergleich zu den anderen Größen.
Sowohl für den Fall, dass die Matrix geclusterte Eigenwerte hat, als für den Fall, dass nur 10% der Eigenwerte berechnet werden, sehen die Ergebnisse ähnlich aus. Die optimalen Blockgrößen für die höchste Genauigkeit liegen ebenfalls in kleineren Bereichen als dies bei den optimalen Größen für die Performance der Fall war.
Da insgesamt betrachtet, alle Normen für jeden Durchlauf sehr gute Ergebnisse liefern, wird wie auch bei ScaLAPACK im Folgenden nicht weiter auf die Korrektheit der Werte eingegangen.
Da sich diese Arbeit hauptsächlich mit der Performance der Routinen beschäftigt, werden die Blockgrößen nachfolgend so gewählt, dass die Performance und nicht die Genauigkeit der Ergebnisse optimiert wird.

5.3.3 Elemental - BG/Q Prototyp

Da der Prototyp des Blue Gene/Q Systems nicht zu jeder Zeit zur Verfügung stand, wurden die Messungen auf dieser Maschine in einem kleineren Umfang durchgeführt.

Für die Blockgrößenanalyse wurde nur die kleinere Referenzmatrix von der Dimension 8192 genutzt und auf insgesamt 32 Knoten mit jeweils 32 Prozessen, also jeweils 2 auf den 16 Kernen, gerechnet.
Da die Bibliothek ScaLAPACK auf diesem System noch nicht zur Verfügung steht, wurde nur die Routine aus Elemental gemessen. Es wurden dafür die algorithmischen Blockgrößen $8, 16, \ldots, 216$ betrachtet, da auf dem Blue Gene/Q die größeren Blöcke wieder zu schlechteren Laufzeiten führen.



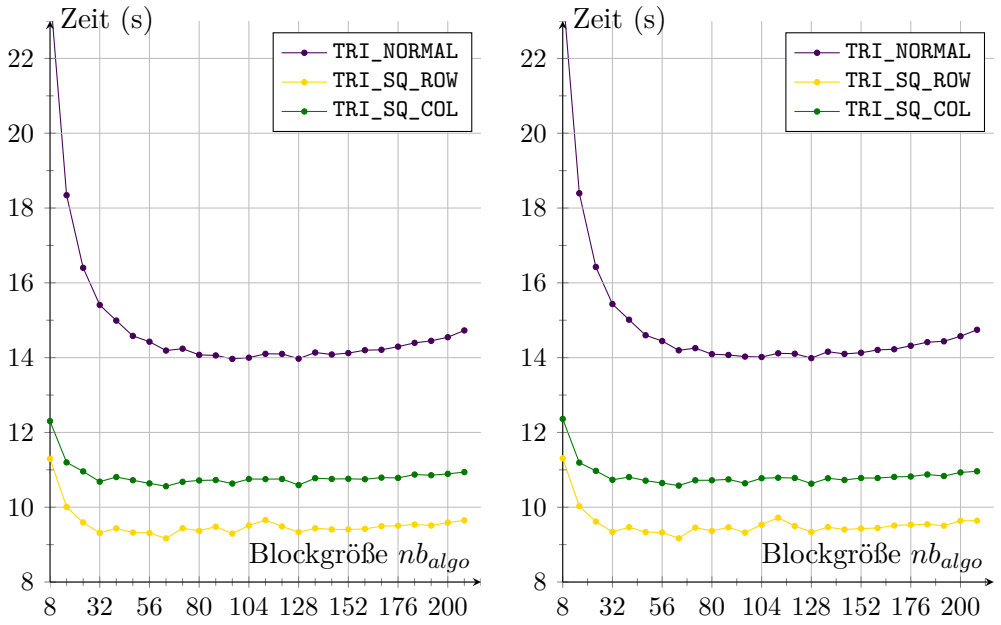
(a) Matrixgröße $n = 8192$ (isoliert)

(b) Matrixgröße $n = 8192$ (geclustert)

Abbildung 5.3.9: Laufzeiten der Routine `HermitianEig` aus Elemental mit den drei verschiedenen Tridiagonalisierungen bei verschieden gewählten Blockgrößen nb_{algo} auf dem BG/Q-System. Es wurden jeweils alle Eigenwerte und -vektoren berechnet.

In Abbildung 5.3.9 ist die Untersuchung der Blockgrößen auf dem Blue Gene/Q zu sehen, wenn alle Eigenwerte und -vektoren mit `HermitianEig` berechnet werden.
In Abbildungsteil (a) sind die Laufzeiten für Matrizen mit isolierten Eigenwerten zu sehen. Der Kurvenverlauf ist für geclusterte Eigenwerte in Abbildungsteil (b) allerdings nahezu identisch, sodass hier nicht weiter darauf eingegangen wird.
Für die beiden Algorithmen für quadratische Prozessorgitter ist die Blockgröße 64 als optimale Größe zu erkennen. Die Struktur der Kurven ist mit den lokalen Minima bei den 2-er Potenzen vergleichbar mit den Kurven, welche aus der Untersuchung auf dem JUGENE entstanden sind. Dort war die 128 optimal, welche hier ebenfalls ein lokales Minimum darstellt, welches aber nicht so stark ausgeprägt ist. Die Tendenz der Kurven ist für größere Blöcke steigend, weshalb keine Laufzeiten für noch

größere Blöcke gemessen wurden.
Falls `HermitianEig` mit `TRI_NORMAL` ausgeführt wird, ist die Blockgröße 128 wiederum als optimal anzusehen, wobei auch eine Größe von 64 eine gute Wahl darstellt.



(a) Matrixgröße $n = 8192$ (isoliert) (b) Matrixgröße $n = 8192$ (geclustert)

Abbildung 5.3.10: Laufzeiten der Routine `HermitianEig` aus Elemental mit den drei verschiedenen Tridiagonalisierungen bei verschieden gewählten Blockgrößen nb_{algo} auf dem BG/Q-System. Es wurden jeweils 10% aller Eigenwerte und -vektoren berechnet.

In Abbildung 5.3.10 ist die Blockgrößenuntersuchung für `HermitianEig` auf dem Blue Gene/Q System dargestellt, wenn nur 10% der Eigenwerte und -vektoren berechnet werden.
Dem Abbildungsteil (a) liegt dafür eine Matrix mit gleichverteilten Eigenwerten zwischen 0 und 1 zu Grunde, während Teil (b) sich mit den um 0 herum geclusterten Eigenwerten befasst.
Das Verhalten der Routine ist sehr ähnlich zu dem in Abbildung 5.3.9 dargestellten Verhalten. Die Auswirkungen der verschiedenen Blockgrößen auf die Laufzeit ist bei der Berechnung von nur 10% identisch zu der Berechnung aller Eigenwerte. Dies ist auf dem JUGENE nicht der Fall gewesen, was ein Vergleich der Abbildungen 5.3.6 und 5.3.4 zeigt.
Die Korrektheit der berechneten Eigenwerte liegt auch auf dem Blue Gene/Q System in einem sehr guten Bereich, so dass für diesen Fall hier nicht weiter darauf eingegangen wird.

Kapitel 6

Laufzeit- und Skalierungsverhalten

In diesem Kapitel werden die Testergebnisse vorgestellt. Die Messungen wurden jeweils mit den im vorherigen Kapitel als optimal bestimmten Blockgrößen ausgeführt. In Abschnitt 6.1 werden die durchgeführten Messungen aufgelistet. Die wichtigsten Ergebnisse dazu werden in den Abschnitten 6.2 und 6.3 dargestellt, wobei zunächst die Laufzeiten und im folgenden das Skalierungsverhalten aller Routinen präsentiert wird.
In Abschnitt 6.4 werden die Ergebnisse für die beiden Blue Gene-Architekturen gegenübergestellt. Der letzte Abschnitt dieses Kapitels stellt die Ergebnisse der Analyse mit Hilfe des Tools Scalasca [66] dar.

6.1 Testfälle

Zur Untersuchung der Laufzeiten und des Skalierungsverhalten aller Routinen wurden folgende Messungen durchgeführt:

JUGENE:

- 64 Knoten mit je einem Prozess, logisches (8×8)-Prozessorgitter
- 128 Knoten mit je einem Prozess, logisches (8×16)-Prozessorgitter
- 256 Knoten mit je einem Prozess, logisches (16×16)-Prozessorgitter
- 512 Knoten mit je einem Prozess, logisches (16×32)-Prozessorgitter
- 1024 Knoten mit je einem Prozess, logisches (32×32)-Prozessorgitter

Auf JUGENE wurden die vier Routinen aus ScaLAPACK und Elementals `HermitianEig` mit den drei Tridiagonalisierungen gemessen. Für alle Routinen wurden Matrizen der Dimensionen $1024, 1536, \ldots, 12288$ genutzt. Diese wurden alle jeweils sowohl mit zwischen 0 und 1 gleichverteilten Eigenwerten als auch mit einem Cluster von Eigenwerten um 0 herum erzeugt. Für beide Arten von Matrizen, mit isolierten und mit geclusterten Eigenwerten, wurde

die Performance jeder Routine gemessen.
Die Routinen, die es erlauben, nur einen Teil der Eigenwerte zu berechnen, wurden für beide Arten von Matrizen zusätzlich darauf getestet, nur 10% der Eigenwerte und -vektoren zu berechnen. Größere Matrizen sind für den Vergleich mit dem Blue Gene/Q Prototyp aufgrund des geringen Speicherplatzes auf dem Testsystem nicht geeignet, weshalb auch auf JUGENE nur Matrizen bis zur Dimension 12288 verwendet wurden.

Blue Gene/Q Prototyp:

- 32 Knoten mit je 2 Prozessen, logisches (8×8)-Prozessorgitter
- 32 Knoten mit je 4 Prozessen, logisches (8×16)-Prozessorgitter
- 32 Knoten mit je 8 Prozessen, logisches (16×16)-Prozessorgitter
- 128 Knoten mit je 4 Prozessen, logisches (16×32)-Prozessorgitter
- 128 Knoten mit je 8 Prozessen, logisches (32×32)-Prozessorgitter

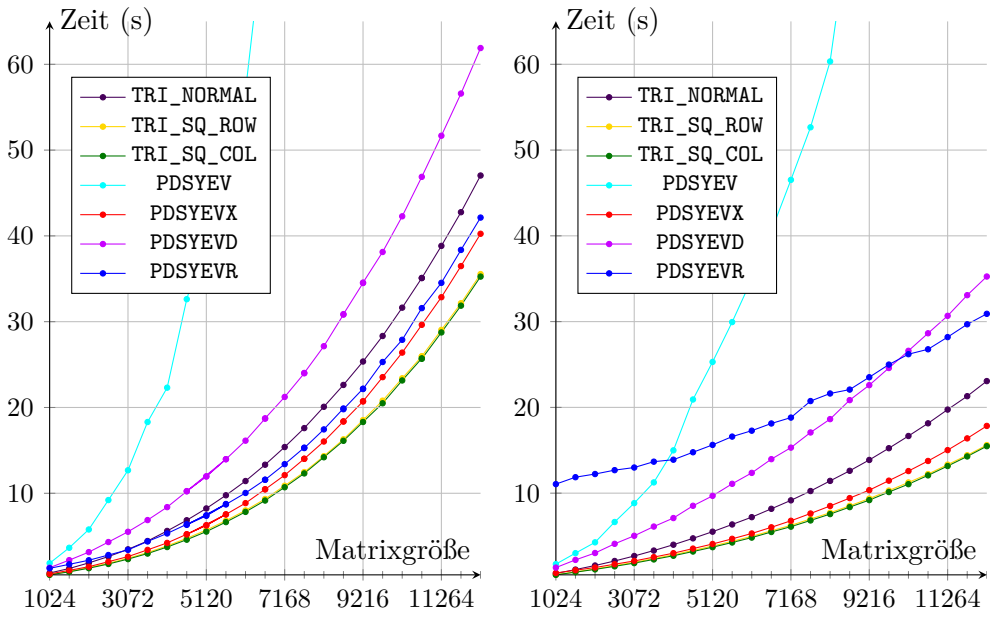
Auf dem Blue Gene/Q System wurde nur Elementals `HermitianEig` mit den drei Tridiagonalisierungen gemessen. Diese wurde jeweils für Matrizen der Dimensionen $1024, 1536, \ldots, 12288$ ausgeführt. Es wurden wieder jeweils beide Arten von Matrizen, isoliert und geclustert, betrachtet und sowohl alle als auch 10% der Eigenwerte und -vektoren berechnet. Da die Mindestanzahl der zu reservierenden Knoten auf dem Prototyp 32 beträgt, wurden für die kleineren Prozessorgitter nicht weniger Knoten verwendet. Zudem ist für eine Verwendung von jeweils 16 Prozessen pro Knoten, was sich als optimal herausstellt, der Speicherplatz bei weniger verwendeten Knoten nicht ausreichend.

Für die nichtquadratischen Prozessorgitter wurden $nb_{row} < nb_{col}$ gewählt, da diese Auswahl bei ScaLAPACK zu besseren Ergebnissen führt. In Elemental wird die Laufzeit nur minimal von dieser Wahl beeinflusst. Bei den selten und nur leicht auftretenden Differenzen in Elemental, ist $nb_{row} < nb_{col}$ auch hier die bessere Wahl. Daher werden die Tests mit $nb_{col} < nb_{row}$ hier nicht weiter betrachtet.
Zusätzlich zu dem beschriebenen Testumfang, wurden vereinzelte Tests mit veränderter Knotenanzahl durchgeführt, auf welche an dieser Stelle nicht weiter eingegangen wird. Diese Tests werden teilweise für die Auswertung mit einbezogen, und bei der Präsentation der Ergebnisse aufgegriffen.

6.2 Laufzeiten

6.2.1 JUGENE

Zunächst werden in Abbildung 6.2.1 die Laufzeiten aller getesteten Routinen für Matrizen mit gleichverteilten Eigenwerten auf quadratischen Prozessorgittern dargestellt.
Wie bei der Blockgrößenanalyse schon zu sehen war, kann ScaLAPACKs `PDSYEV`,

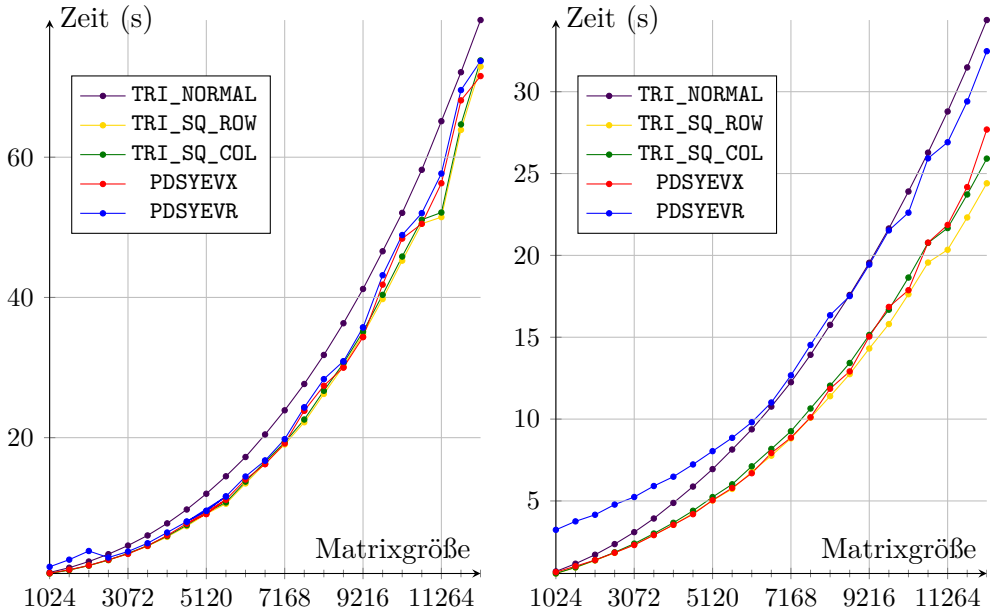


(a) 256 Prozesse, (16×16)-Gitter (b) 1024 Prozesse, (32×32)-Gitter

Abbildung 6.2.1: Laufzeiten aller Routinen für Matrizen mit isolierten Eigenwerten, welche zwischen 0 und 1 gleichverteilt generiert wurden. Elementals Routine `HermitianEig` wird für alle drei verschiedenen Algorithmen dargestellt. Die Messungen wurden auf JUGENE mit 256 für (a) bzw. 1024 Knoten für (b) ausgeführt, wobei auf jedem Knoten jeweils ein Prozessorkern genutzt wurde.

welche den QR-Algorithmus nutzt, mit keiner der anderen Routinen mithalten. Sie ist mit großem Abstand die langsamste der Routinen, was sich besonders bei größeren Matrizen auswirkt. Die meisten Routinen haben einen eher flachen Anstieg. Einzig `PDSYEV` zeigt ein $\mathcal{O}(n^3)$ Verhalten bezüglich der Matrixdimension n, was eigentlich bei allen Routinen zu erwarten wäre, da sie von der Reduktion dominiert werden.
Die schnellste Routine ScaLAPACKs ist `PDSYEVX`, welche die Bisektion und die Inverse Iteration verwendet. Insgesamt betrachtet löst Elementals `HermitianEig` das Problem am schnellsten, wenn sie für die Tridiagonalisierung einen der beiden Algorithmen `TRI_SQ_ROW` oder `TRI_SQ_COL` für quadratische Prozessorgitter nutzt. Dazu sind in diesen beiden Beispielen die Voraussetzungen optimal, da zu Beginn schon mit quadratischen Gittern gearbeitet wird, und sie daher nicht verkleinert werden müssen.
Interessant ist das Verhalten von ScaLAPACKs `PDSYEVR` in Abbildung 6.2.1. Der Verlauf sieht mit dem kleineren Gitter in Teil (a) normal aus, wogegen in Teil (b) zu erkennen ist, dass die Routine für kleine Matrizen sehr langsam ist. Sie benötigt für die Matrixdimension 1024 mit mehr Prozessoren in Teil (b) etwa 10 mal mehr Zeit als mit weniger Prozessoren, obwohl sie eigentlich schneller sein sollte.

Dieses Phänomen haben wir auch bei 4096 Prozessoren in der Blockgrößenanalyse schon beobachtet, so dass dort dann doch nur mit 1024 Prozessoren gerechnet worden ist.
Hier sehen wir nun, dass sich dieses Verhalten mit der Matrixgröße ändert, sodass bei einer Größe von 12288 die Laufzeit wie zu erwarten mit mehr Prozessoren schneller ist.
Die Routine `PDSYEVD` von ScaLAPACK ist insgesamt gesehen die zweit langsamste Routine nach `PDSYEV`. Das Verhalten der Routinen für nichtquadratische Prozessor-



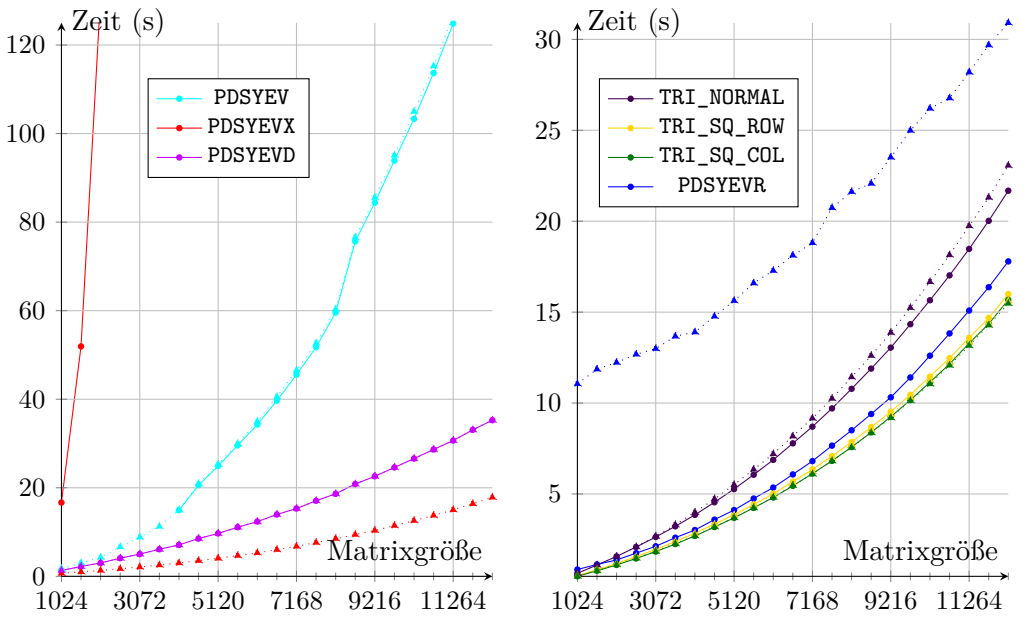
(a) 128 Prozesse, (8 × 16)-Gitter

(b) 512 Prozesse, (16 × 32)-Gitter

Abbildung 6.2.2: Laufzeiten ausgewählter Routinen für Matrizen mit isolierten Eigenwerten, welche zwischen 0 und 1 gleichverteilt generiert wurden. Elementals Routine `HermitianEig` wird für alle drei verschiedenen Algorithmen dargestellt. Die Messungen wurden auf JUGENE mit 128 für (a) bzw. 512 Knoten für (b) ausgeführt, wobei auf jedem Knoten jeweils ein Prozessorkern genutzt wurde.

gitter wird in Abbildung 6.2.2 dargestellt. Die beiden Routinen `PDSYEV` und `PDSYEVD` sind insgesamt langsamer als die übrigen Routinen. Da sie außerdem den Reduktionsalgorithmus verwenden, der auf nichtquadratischen Prozessorgittern läuft, ist ihr Verhalten auf quadratischen und nichtquadratischen Prozessorgittern identisch. Sie werden daher in Abbildung 6.2.2 nicht betrachtet.
Die zwei Routinen `PDSYEVX` und `PDSYEVR`, welche die Tridiagonalisierung mittels `PDSYNTRD` auf einem quadratischen Gitter ausführen, kommen gut mit einem nichtquadratischen Gitter zurecht. Man sieht in dieser Abbildung, dass auch `HermitianEig` mit den beiden Tridiagonalisierungsalgorithmen für quadratische Gitter immer noch besser als mit `TRI_NORMAL` ist. Wie schon zu Beginn der Parameterstudie bemerkt,

ist TRI_SQ_ROW in vielen Situationen besser als TRI_SQ_COL. Daher liegt es nahe, immer TRI_SQ_ROW für die Tridiagonalisierung zu nutzen, wenn man die Berechnung mittels HermitianEig ausführt.
Der Abstand zwischen HermitianEig mit TRI_NORM und HermitianEig mit den beiden schnelleren Tridiagonalisierungsalgorithmen ist allerdings nicht mehr so groß, wie auf einem quadratischen Gitter. Dies ist aber auch nicht verwunderlich, da für die Tridiagonalisierung von HermitianEig weniger Prozessoren genutzt werden, als eigentlich zu Verfügung stehen, da auf einem quadratischen Gitter gerechnet wird. Für 128 Prozessoren wären dies 121 in einem (11×11)-Gitter, da dies die größte darin enthaltene Quadratzahl ist. Für Teil (b) sind die Auswirkungen noch gravierender, da hier nur 484 statt der 512 verfügbaren Prozessoren genutzt werden. Außerdem erfordert die Umverteilung bei einer größeren Anzahl von Prozessoren mehr Kommunikation. Die Umverteilung auf ein quadratisches Gitter ist auch in der Analyse mit Scalasca in Abschnitt 6.5 zu sehen.



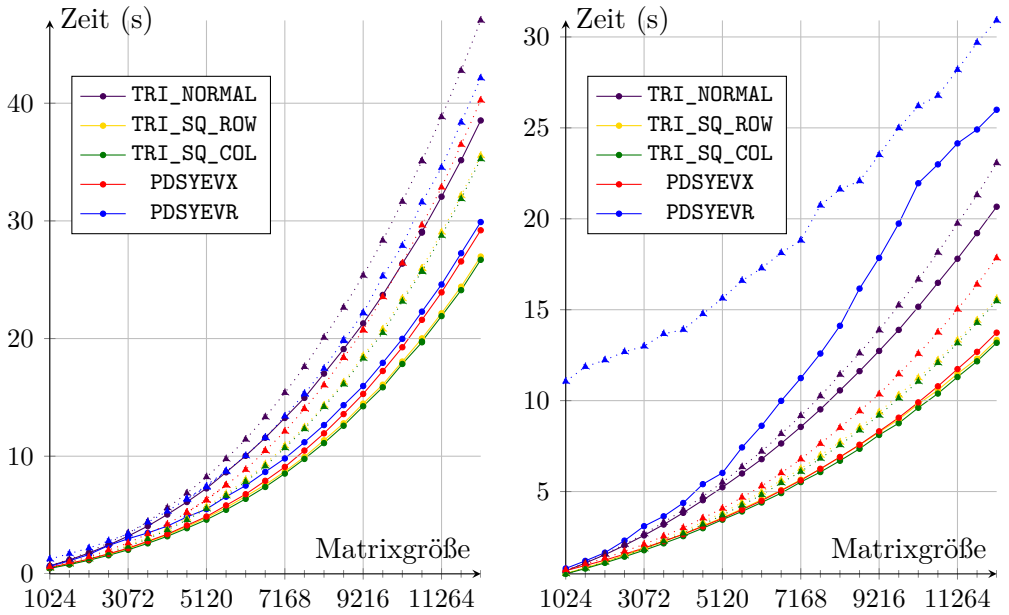
(a) 1024 Prozesse, (32×32)-Gitter

(b) 1024 Prozesse, (32×32)-Gitter

Abbildung 6.2.3: Vergleich der Laufzeiten aller Routinen bei geclusterten (durchgezogene Linien) und isolierten Eigenwerten (gestrichelten Linien). Die Messungen wurden auf JUGENE mit 1024 Knoten ausgeführt, wobei auf jedem Knoten jeweils ein Prozessorkern genutzt wurde.

In Abbildung 6.2.3 werden alle Routinen bezüglich ihres Verhaltens mit geclusterten Eigenwerten untersucht.
Die gestrichelten Linien zeigen nochmal den Kurvenverlauf aus Abbildung 6.2.1 für Matrizen mit isolierten Eigenwerten, während die durchgezogenen Linien die Lauf-

zeiten für Matrizen mit geclusterten Eigenwerten darstellen.
In Teil (a) sind die Laufzeiten der Routinen abgebildet, welche nicht auf den MR^3-Algorithmus basieren. Man erkennt am Kurvenverlauf von `PDSYEV`, dass der QR-Algorithmus für beide Arten von Matrizen die gleiche Zeit benötigt.
Die Performance der Routine `PDSYEVD` scheint ebenfalls unabhängig von der Lage der Eigenwerte zu sein, da beide Kurven übereinander liegen.
Die interessanteste Entwicklung ist für `PDSYEVX` zu beobachten. Wie bereits erwähnt,



(a) 256 Prozesse, (16 × 16)-Gitter

(b) 1024 Prozesse, (32 × 32)-Gitter

Abbildung 6.2.4: Die Laufzeiten der Eigenwertlöser bei Berechnung von 10% der Eigenwerte und -vektoren, wobei die Matrizen isolierte Eigenwerte zwischen 0 und 1 haben. Die Messungen wurden auf JUGENE mit 256 bzw. 1024 Knoten ausgeführt, wobei auf jedem Knoten jeweils ein Prozessorkern genutzt wurde. Zum Vergleich sind die Laufzeiten der Berechnung aller Eigenwerte und -vektoren als gestrichelte Kurven mit eingezeichnet.

stehen die Eigenvektoren von geclusterten Eigenwerten nach der Inversen Iteration, welche hier genutzt wird, nicht orthogonal aufeinander, weshalb eine Nachorthogonalisierung ausgeführt wird. Da diese auf nur einem Prozessor ausgeführt wird, wird diese Routine bei geclusterten Eigenwerten extrem langsam. Der letzte vorhandene Messpunkt liegt bei einer Matrixdimension von 3584 und bei einer Laufzeit von ca. 760 Sekunden. Für alle Größen darüber hinaus reicht der Hauptspeicher auf einem Knoten des JUGENEs (2 GB) nicht mehr aus.
Dabei wurde schon mit 1024 Knoten und je einem Kern, statt mit 256 Knoten und je 4 Kernen gerechnet, um viel Speicherplatz für jeden Prozessor zu ermöglichen. Dies wurde so praktiziert, da die Eingabematrix für die spätere Kontrolle der Ergebnisse

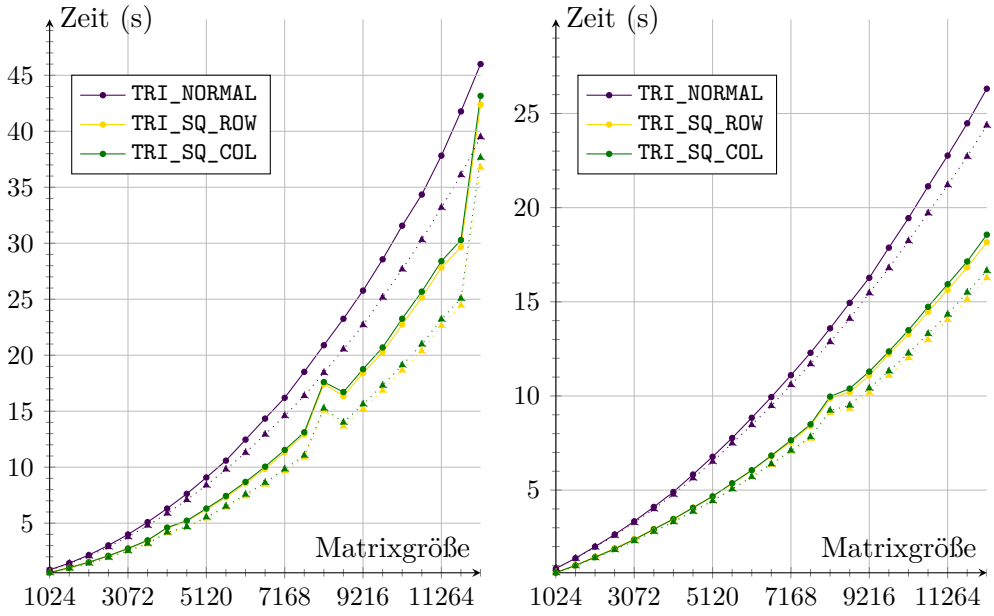
komplett abgespeichert wird, was den für die Rechnung zur Verfügung stehen Speicherplatz einschränkt.
In Teil (b) werden die drei verschiedenen Arten von `HermitianEig` aus Elemental und `PDSYEVR` aus ScaLAPACK untersucht. Die schnellsten Varianten, `HermitianEig` mit den beiden Algorithmen für quadratische Prozessorgitter, haben sich kaum verändert. Hier scheint die Lage der Eigenwerte keinen bzw. nur einen geringen Einfluss auf die Performance zu nehmen. Für den Algorithmus `TRI_NORMAL` ist bei geclusterten Eigenwerten eine leichte Verbesserung der Performance zu beobachten.
Bemerkenswert ist allerdings das Verhalten von ScaLAPACKs `PDSYEVR` zur Berechnung geclusterter Eigenwerte. Während sie zur Berechnung von isolierten Eigenwerten, insbesondere von kleinen Matrizen, relativ lange braucht, hat sie diese Probleme bei geclusterten Eigenwerten nicht. Die gesamte Performance dieser Routine verbessert sich deutlich, so dass sie fast so schnell wie die zwei quadratischen Varianten von `HermitianEig` ist.

In Abbildung 6.2.4 werden die Laufzeiten der Routinen betrachtet, die es erlauben, nur einen ausgewählten Teil der Eigenwerte und -vektoren zu berechnen. Es wurden jeweils die kleinsten Eigenwerte einer Matrix mit gleichverteilten Eigenwerten berechnet, da bei der Matrix mit geclusterten Eigenwerten wieder die Probleme mit der Nachorthogonalisierung auftreten. Wieder wurde auf JUGENE mit 256 bzw. 1024 Knoten und nur einem Prozessorkern pro Knoten gemessen, um die Vergleichbarkeit mit den Laufzeiten bei Berechnung aller Eigenwerte und -vektoren aus Abbildung 6.2.1 (gestrichelte Kurven) zu gewährleisten. Man erkennt sofort, dass, wie zu erwarten, die Berechnung von nur 10% der Eigenwerte und -vektoren schneller ist, als die Berechnung alle Eigenwerte und -vektoren.
Die beiden ScaLAPACK Routinen `PDSYEVX` und `PDSYEVR`, welche die Funktionalität bieten nur ausgewählte Eigenwerte zu berechnen, profitieren allerdings mehr davon als Elementals `HermitianEig`. Dies ist besonders deutlich bei dem größeren Prozessorgitter in Abbildungsteil (b) zu erkennen, da der Abstand zwischen den gleichfarbigen Kurven bei den beiden Routinen ScaLAPACKs am größten ist. Während Elementals Routinen mit den quadratischen Tridiagonalisierungsalgorithmen bei der Berechnung aller Eigenwerte und -vektoren noch deutlich schneller waren, ist ScaLAPACKs `PDSYEVX` hier fast gleich schnell.
Die Routine `PDSYEVR` hat die schon häufiger aufgetretenen Probleme für kleine Matrixdimensionen nicht mehr, wenn nur 10% der Eigenwerte zu berechnen sind, weshalb der Abstand hier am größten ist. Dafür scheinen große Matrizen hier zu leichten Problemen zu führen, da der Anstieg der Kurve in der ab Dimension 5120 sehr steil wird.
Insgesamt betrachtet ist Elementals `HermitianEig` mit `TRI_SQ_ROW` bzw. `TRI_SQ_COL` aber auch in diesen Tests die schnellste Routine.

6.2.2 Blue Gene/Q Prototyp

Abbildung 6.2.5 zeigt die Laufzeiten der Routine `HermitianEig` aus Elemental bei Matrizen mit isolierten Eigenwerten. Für beide Abbildungsteile wird die Routine

jeweils auf einem quadratischen Prozessorgitter ausgeführt, sodass bei `TRI_SQ_COL` und `TRI_SQ_ROW` jeweils alle zur Verfügung stehende Prozesse genutzt werden können. Wie auch auf JUGENE, beschleunigen beiden Algorithmen für quadratische Gitter die Routine deutlich gegenüber `TRI_NORMAL`.

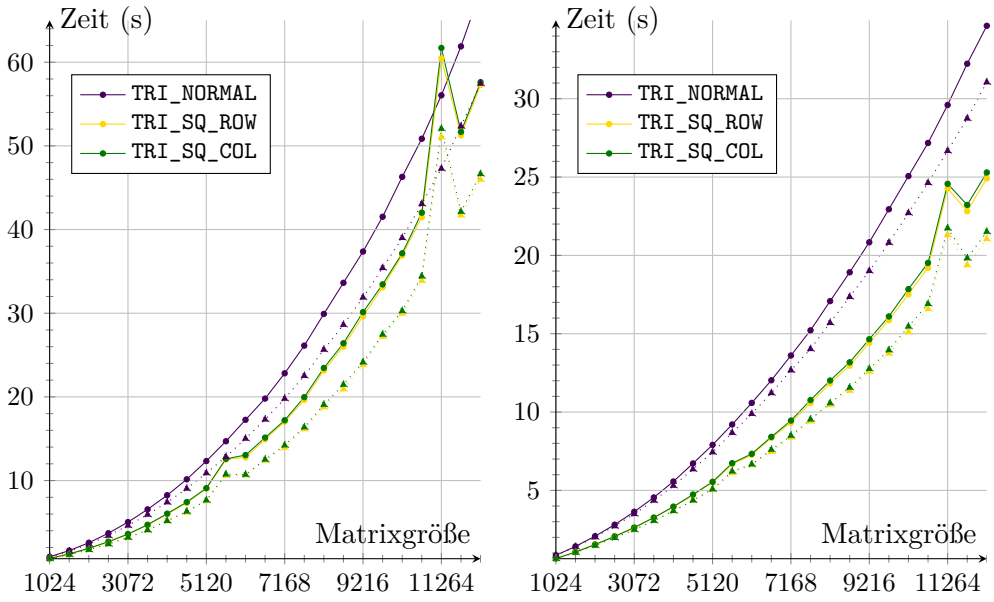


(a) 256 Prozesse, (16×16)-Gitter (b) 1024 Prozesse, (32×32)-Gitter

Abbildung 6.2.5: Laufzeiten der Routine `HermitianEig` mit den drei Tridiagonalisierungsalgorithmen für Matrizen mit isolierten Eigenwerten, welche zwischen 0 und 1 gleichverteilt generiert wurden. Die Messungen wurden auf dem Blue Gene/Q Prototyp mit 32 für (a) bzw. 128 Knoten für (b) ausgeführt, wobei auf jedem Knoten jeweils 8 Prozesse gestartet wurden. Die durchgezogene Linie stellt die Laufzeit der Berechnung aller Eigenwerte und -vektoren dar, während die gestrichelte Linie die Laufzeit der Berechnung von 10% der Eigenwerte und -vektoren markiert.

Für das größere Prozessorgitter wurden 128 Knoten mit je 8 Prozessen genutzt, da für 32 Knoten mit je 32 Prozessen der Speicher für die Matrizen ab Dimension 8704 nicht mehr ausreicht. Die durch eine veränderte Knotenanzahl entstehenden Unterschiede in der Laufzeit, werden später in Abbildung 6.3.5 untersucht.
In Abbildungsteil (a) sind jeweils drei Ausreißer in den Kurven von `HermitianEig` mit den quadratischen Algorithmen zu erkennen. Die Routine ist für die Matrixdimensionen $4096, 8192$ und 12288 deutlich langsamer als dies anhand des restlichen Kurvenverlaufs zu erwarten ist. Während diese Verzögerung bei 4096 noch sehr gering ist, wird sie für die größeren Dimensionen immer extremer.
Auf dem größeren Prozessorgitter, dargestellt in (b), liegt diese Verzögerung nur bei Dimension 8192 vor, allerdings nicht so deutlich wie auf dem kleineren Gitter.

Selbst für die Dimensionen, an denen die beschriebenen Verzögerungen auftreten, ist `HermitianEig` mit den quadratischen Algorithmen schneller als mit `TRI_NORMAL`, obwohl dort keine Verzögerungen zu erkennen sind. Da die Verzögerungen nur bei Verwendung der beiden schnelleren Algorithmen auftreten, muss die Tridiagonalisierung dafür verantwortlich sein.
Wie zu erwarten, wird die Berechnung von nur 10% der Eigenwerte (gestrichelte Linien) jeweils ein bisschen schneller ausgeführt, als die Berechnung aller Eigenwerte (durchgezogene Linien). Dass die Verzögerungen hierbei ähnlich stark sind wie bei der Berechnung aller Eigenwerte, lässt ebenfalls auf die Tridiagonalisierung als Ursache schließen, weil diese immer vollständig durchgeführt wird.



(a) 128 Prozesse, (8 × 16)-Gitter

(b) 512 Prozesse, (16 × 32)-Gitter

Abbildung 6.2.6: Laufzeiten der Routine `HermitianEig` mit den drei Tridiagonalisierungsalgorithmen für Matrizen mit isolierten Eigenwerten, welche zwischen 0 und 1 gleichverteilt generiert wurden. Die Messungen für Teil (a) wurden auf dem Blue Gene/Q Prototyp mit 32 Knoten und je 4 Prozessen ausgeführt. Für Teil (b) wurden 128 Knoten mit jeweils 4 Prozessen genutzt.

Das Verhalten der Laufzeit auf einem nichtquadratischen Prozessorgitter wird in Abbildung 6.2.6 dargestellt. Hierbei ist insbesondere das Verhalten der beiden quadratischen Varianten der Tridiagonalisierung interessant, da nicht mehr alle zur Verfügung stehenden Prozesse genutzt werden.
`TRI_SQ_COL` und `TRI_SQ_ROW` sind insgesamt betrachtet wie auf JUGENE auch auf dem Blue Gene/Q System mit nichtquadratischen Prozessorgittern die schnelleren Varianten für `HermitianEig`. Allerdings treten hierbei, wie in Abbildung 6.2.5, eben-

falls Verzögerungen bei bestimmte Matrixdimensionen auf. Auf beiden Prozessorgittern ist die Berechnung der Eigenwerte einer Matrix der Dimension 5632 und 11264 langsamer als erwartet. Diese Verzögerung ist, wie auch bei den beiden quadratischen Gittern, auf dem kleineren Prozessorgitter stärker ausgeprägt.
Für die Dimension 11264 ist `HermitianEig` mit `TRI_NORMAL` sogar schneller, da hier zum einen die starke Verzögerung bei den quadratischen Algorithmen auftritt und zum anderen nicht alle Prozessoren für die Tridiagonalisierung genutzt werden.
Das Verhalten für die Berechnung von nur einem kleinen Teil der Eigenwerte und -vektoren ist wieder gleich, wobei die Berechnung wie zu erwarten ein wenig schneller durchgeführt wird.
Die Laufzeit von `HermitianEig` zeigt auf dem Blue Gene/Q System keine Reaktion auf geclusterte Eigenwerte. Da der Kurvenverlauf absolut identisch zu dem Verlauf der Berechnung isolierter Eigenwerte ist, wird er hier nicht abgebildet.

6.3 Skalierungsverhalten

6.3.1 Performancebewertung paralleler Programme

Zur Bewertung des Skalierungsverhaltens von parallelen Programmen oder Bibliotheken wird nicht nur die parallele Laufzeit betrachtet. Es existieren weitere Metriken zur Beurteilung der parallelen Performance. Zwei häufig genutzte Begriffe sind der ***Speedup*** und die ***Effizienz***, welche nachfolgend eingeführt werden.
Der Speedup bezieht neben der parallelen Performance das Verhältnis zur Laufzeit T_{seq} des schnellsten sequentiellen Programms mit ein. Die parallele Laufzeit für p Prozessoren wird im Folgenden mit T_p bezeichnet.
Der Speedup ist durch

$$S_p := \frac{T_{seq}}{T_p} \tag{6.1}$$

definiert [67]. Gene Amdahl hat 1967 in [68] die Leistungssteigerung eines Programms mit fester Problemgröße n untersucht und daraufhin den theoretischen Speedup wie folgt definiert:

$$\hat{S}_p := \frac{T_{seq}}{fT_{seq} + \frac{(1-f)}{p}T_{seq}} \tag{6.2}$$

Dabei wird mit $0 \leq f \leq 1$ der Anteil des Programms bezeichnet, welcher nicht parallelisierbar ist. Wenn $f = 0$ wäre, würde dies zu einem idealen Speedup von $\hat{S}_p = p$ führen. Wenn nun in (6.2) die Prozessoranzahl p sehr groß wird, läuft der Speedup in eine Sättigung, da

$$\lim_{p\to\infty} \hat{S}_p = \frac{1}{f}$$

gilt und f als konstant angenommen wird. Dieser Zusammenhang wird heute als Amdahls Gesetz bezeichnet.
Die Effizienz eines parallelen Programms ist durch

$$E_p = \frac{S_p}{p}$$

definiert. Im Idealfall wäre die Effizienz gleich 1, was z. B. auf Grund des Kommunikationsoverhead nie erreicht werden kann.
1988 hat sich John L. Gustafson in [69] mit Amdahls Gesetz beschäftigt und die Problemgröße n an die Prozessoranzahl angepasst, während sie bei Amdahl fix war. In Anlehung an Amdahls Gesetz wird die Aussage, dass ein genügend großes Problem immer effizient parallelisiert werden kann, als Gustafsons Gesetz bezeichnet.
Aus diesen beiden Ansätzen sind die beiden Skalierungsbegriffe ***strong scaling*** und ***weak scaling*** entstanden. Beim ***strong scaling*** wird die Problemgröße n festgehalten, während die Problemgröße beim ***weak scaling*** auf np angepasst wird.
Die im nächsten Abschnitt vorgestellten Ergebnisse beziehen sich nicht auf den in (6.1) definierten Speedup. Da keine Messungen für entsprechende sequentielle Algorithmen vorhanden sind und Messungen der parallelen Routinen mit jeweils einem Prozessor nicht sinnvoll wären, wurde statt der sequentiellen Laufzeit T_{seq} die Laufzeit T_{64} mit 64 Prozessoren betrachtet.
Somit wird der in dieser Arbeit betrachtete Speedup durch

$$\check{S}_p := \frac{T_{64}}{T_p} \tag{6.3}$$

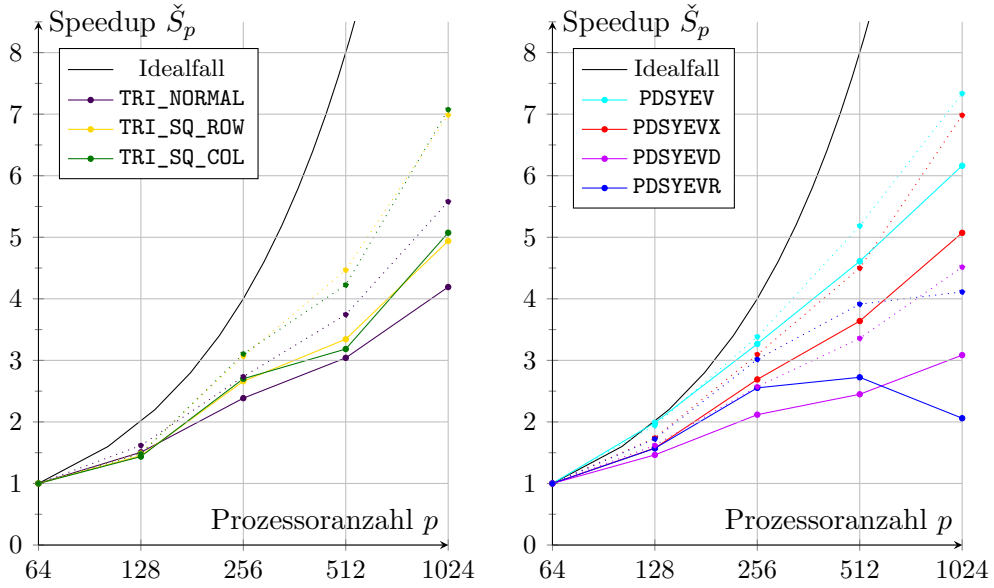
definiert, wobei ein idealer Speedup $\check{S}_p = \frac{p}{64}$ erreichen würde.
Die betrachtete Effizienz bezieht sich ebenfalls auf diesen Speedup und wird durch

$$\check{E}_p = \frac{\check{S}_p}{\frac{p}{64}} \tag{6.4}$$

bestimmt.

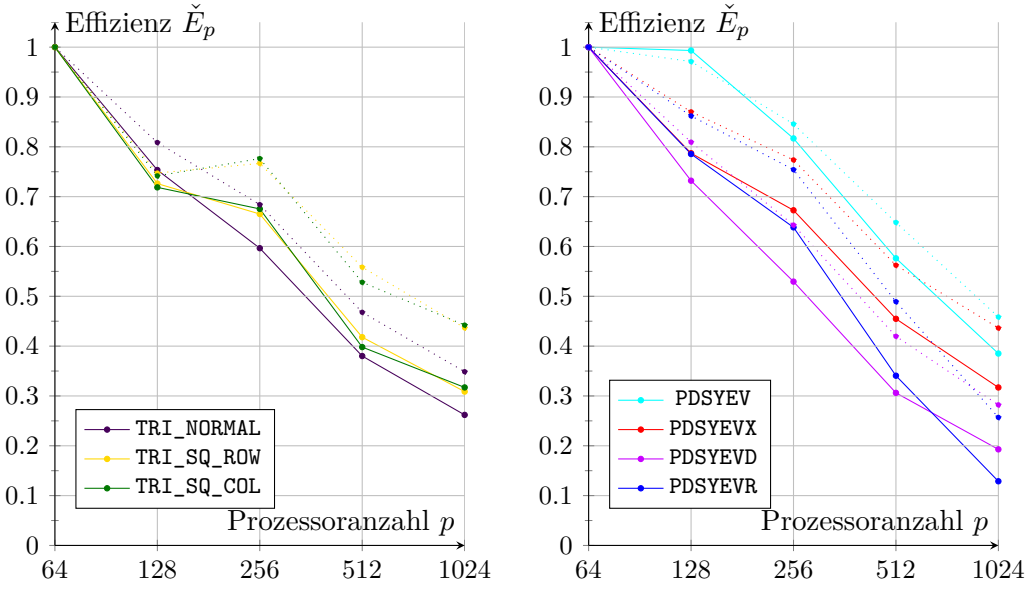
6.3.2 Ergebnisse - JUGENE

Die im Folgenden vorgestellten Ergebnisse zeigen das Skalierungsverhalten der getesteten parallelen Routinen. Dabei beziehen sich die Begriffe Speedup und Effizienz auf die in (6.3) und (6.4) vorgestellten Metriken, welche sich über die Laufzeit mit 64 Prozessoren statt über sie sequentielle Laufzeit berechnen lassen.
In Abbildung 6.3.1 ist der Speedup aller Routinen für die festen Problemgrößen $n = 8192$ und $n = 12288$ aufgetragen. Die Prozessoranzahl ist in der X-Achse logarithmisch aufgetragen.
Die Bibliothek Elemental wird in Teil (a) untersucht. Je mehr Prozessoren genutzt werden, desto weiter entfernt sich der Speedup von der Ideallinie. Dies liegt an Amdahls Gesetz, da die Problemgröße hier festgehalten wurde und der Speedup so in eine Sättigung läuft.



(a) Elemental (b) ScaLAPACK

Abbildung 6.3.1: Speedup aller Routinen bei einer Matrix mit isolierten Eigenwerten auf JUGENE. Die durchgezogenen Kurven beziehen sich auf die Matrixdimension $n = 8192$ und die gestrichelten auf $n = 12288$.

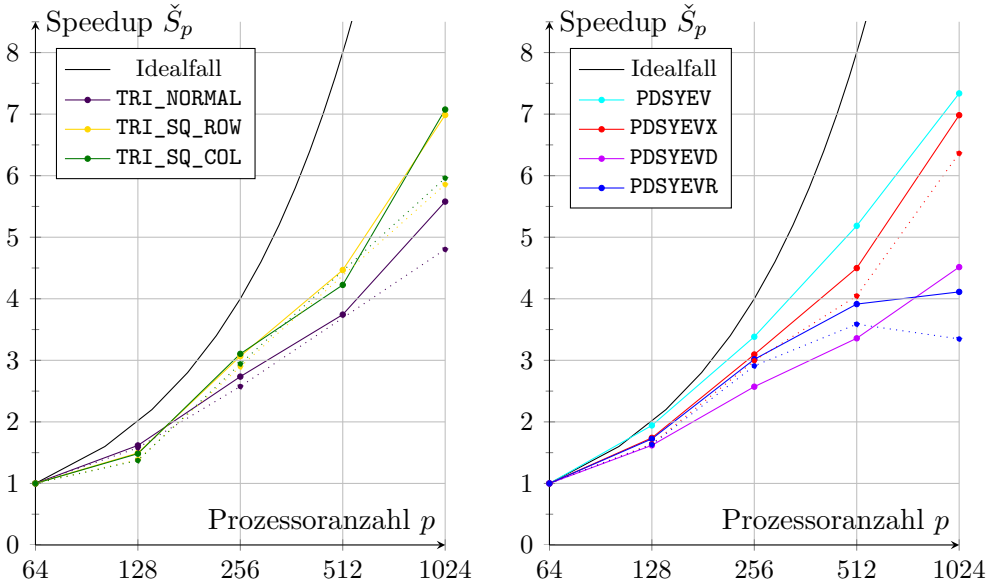


(a) Elemental (b) ScaLAPACK

Abbildung 6.3.2: Effizienz aller Routinen bei einer Matrix mit isolierten Eigenwerten auf JUGENE. Die durchgezogenen Kurven beziehen sich auf die Matrixdimension $n = 8192$ und die gestrichelten auf $n = 12288$.

Eine Vergrößerung der Matrixdimension n wirkt sich positiv auf den Speedup aus, was an den gestrichelten Kurven zu erkennen ist. Selbst die geringe Erhöhung von 8192 auf 12288 hat bei einer Prozessoranzahl von 1024 schon eine sehr große Auswirkung auf den Speedup, da die Sättigung bei der größeren Matrix erst später eintritt. Bei der kleinen Matrix kann man sehr gut erkennen, dass die Routine `HermitianEig` mit `TRI_SQ_COL` oder `TRI_SQ_ROW` bei den quadratischen Prozessorgittern einen vergleichsweise besseren Speedup erreichen. So erreichen zum Beispiel 512 Prozessoren einen kleineren Speedup als man durch Interpolation zwischen 256 und 1024 erwarten würde, da für die Tridiagonalisierung nur 484 statt 512 Prozessoren genutzt werden.

Insgesamt betrachtet, skaliert die Routine besser, wenn sie einen der beiden quadratischen Algorithmen zur Tridiagonalisierung nutzt. Da die Routine mit diesen Algorithmen neben der besten Skalierung auch die beste Laufzeit erreicht, was in Abschnitt 6.2.1 gezeigt wurde, sind die quadratischen Algorithmen immer vorzuziehen.



(a) Elemental (b) ScaLAPACK

Abbildung 6.3.3: Vergleich des Speedups zwischen den Routinen bei der Berechnung aller Eigenwerte und -vektoren (durchgezogene Linien) und der Berechnung von nur 10% der Eigenwerte und -vektoren (gestrichelte Linien) auf JUGENE. Die Matrix hat eine Dimension von $n = 12288$ und zwischen 0 und 1 gleichverteilte Eigenwerte.

In Abbildungsteil (b) werden die Routinen aus ScaLAPACK betrachtet. Der größte Speedup ist bei `PDSYEV` zu erkennen, welche von der Laufzeit allerdings die mit Abstand langsamste Routine ist. Bei einer Prozessoranzahl von 128 erreicht sie fast den Speedup 2, welcher ideal ist. Das heißt, dass mit Verdopplung der Prozessoranzahl,

die Laufzeit fast halbiert wird.
Die Routine `PDSYEVD` profitiert am wenigsten vom Einsatz vieler Prozessoren, da der Speedup immer sehr klein ist. Nur bei 1024 Prozessoren stellt sie nicht den kleinsten Speedup, sondern `PDSYEVR`. Dieser wird für die kleinere Matrix bei 1024 sogar kleiner als bei 512 Prozessoren. Dies ist das Problem, was uns bei dieser Routine nun schon häufiger aufgefallen ist. Dass der Speedup auf einmal so stark verschlechtert wird ist sehr ungewöhnlich, und selbst durch einen hohen Kommunikationsaufwand nicht zu erklären. Da die Routine aber, wie schon erwähnt, nicht im offiziellen ScaLAPACK Release implementiert ist, könnte hier auch noch ein Fehler vorliegen.

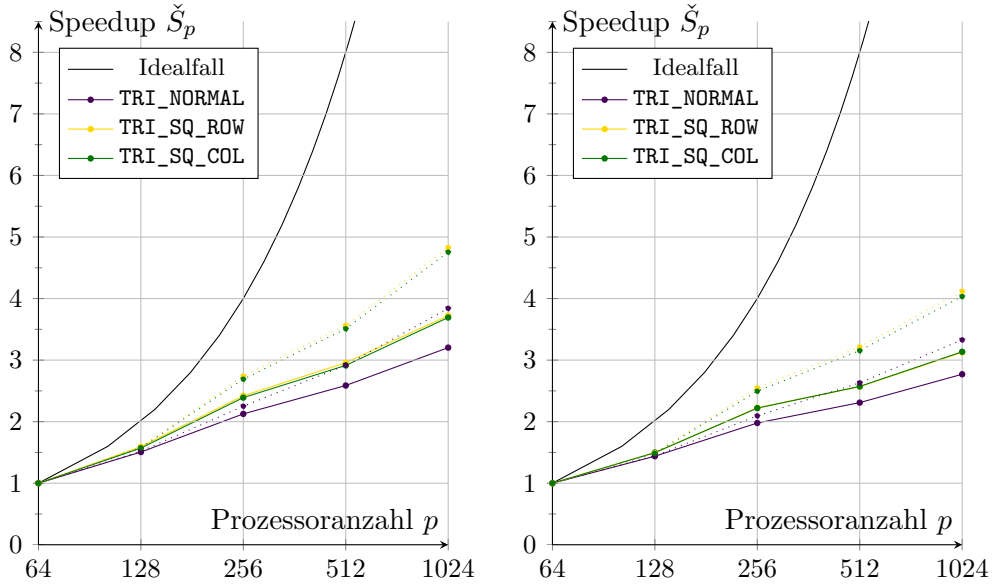
In Abbildung 6.3.2 wird die Effizienz aller Routinen dargestellt. Zur Berechnung der Effizienz wurde der Speedup genutzt, welcher in Abbildung 6.3.1 dargestellt wird. Die Ergebnisse zeigen, dass für diese kleinen Matrizen eine Prozessoranzahl von 1024 nicht sinnvoll ist, da keine der Routinen eine Effizienz von 50% oder mehr erreicht. Bei Elemental in Teil (a) ist zu sehen, dass die Effizienz bei Vergrößerung der Prozessoranzahl an einer Stelle ansteigt, was sehr ungewöhnlich ist. Dies liegt in diesem Fall wiederum an der Nutzung des quadratischen Gitters für die Tridiagonalisierung und der damit verbundenen Leerlaufzeit einiger Prozessoren.
Die Plots zur Effizienz zeigen insgesamt keine neuen Ergebnisse, da sich die Effizienz direkt aus dem Speedup berechnen lässt, und so zu den gleichen Ergebnissen führt.

In Abbildung 6.3.3 sind die Speedups zum Einen für die Berechnung aller Eigenwerte einer (12288×12288)-Matrix und zum Anderen für die Berechnung von 10% der Eigenwerte dieser Matrix dargestellt. Die gestrichelten Linien sind für `PDSYEV` und `PDSYEVD` nicht vorhanden, da diese Routinen die Berechnung von einem Teil der Eigenwerte nicht erlauben.
Der Vergleich zeigt, dass alle Routinen besser skalieren, wenn sie alle Eigenwerte und -vektoren berechnen.
Auch in diesem Beispiel wurden Matrizen mit isolierten Eigenwerten verwendet. Die Lage der Eigenwerte hat allerdings weder bei Elemental noch bei einer der Routinen aus ScaLAPACK Einfluss auf die Skalierung der Routinen, außer für `PDSYEVX`, welche für geclusterte Eigenwerte ohnehin nicht brauchbar ist. Daher werden Matrizen mit geclusterten Eigenwerten an dieser Stelle nicht behandelt.

6.3.3 Ergebnisse - BG/Q Prototyp

Abbildung 6.3.4 stellt den Speedup von `HermitianEig` auf dem Blue Gene/Q System dar. Die Routinen haben für Abbildungsteil (a) alle Eigenwerte und -vektoren berechnet. Wie auch auf JUGENE wirkt sich die Größe der Matrix stark auf den Speedup aus, wenn viele Prozesse genutzt werden, was im Vergleich zwischen den Matrixdimensionen 7680 (durchgezogene Linie) und 11776 (gestrichelte Linie) zu erkennen ist.
Die Routine skaliert auch auf dem Blue Gene/Q System deutlich besser, wenn sie einen der beiden Algorithmen für quadratische Gitter nutzt.
Für Abbildungsteil (b) wurden nur 10% aller Eigenwerte und -vektoren berechnet.

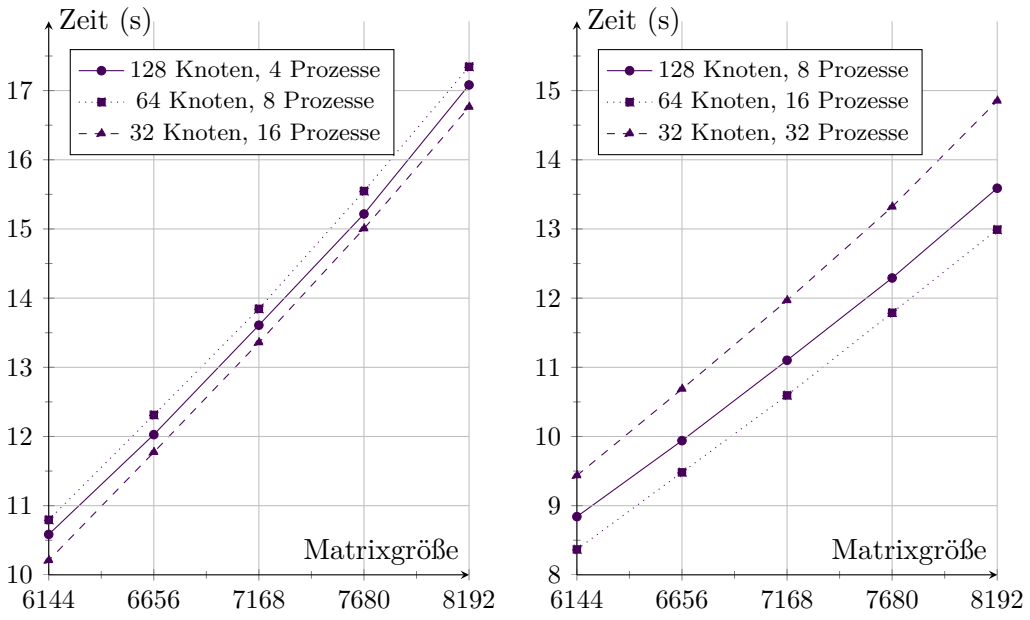
Das Verhalten des Speedups der Routine ist grundsätzlich ähnlich zu dem in Teil (a) abgebildeten Verhalten. Allerdings wird insgesamt, wie auch auf JUGENE, ein etwas kleinerer Speedup erreicht, als der Speedup bei der Berechnung aller Eigenwerte.
Auf dem JUGENE ist der Speedup größer als auf dem Blue Gene/Q Prototyp. Besonders auffällig ist dies im Bereich von 1024 Prozessen.
Die Effizienz bringt keine neuen Erkenntnisse, da es sich bei der Effizienz nur um eine andere Darstellungsform des Speedups handelt. Daher wird sie an dieser Stelle nicht mehr dargestellt.
Der Speedup bzw. die Effizienz hängt wie auf JUGENE auch auf dem Blue Gene/Q System nicht von der Lage der Eigenwerte ab, sodass Matrizen mit geclusterten Eigenwerte hier nicht gesondert betrachtet werden.



(a) alle Eigenwerte und -vektoren (b) 10% aller Eigenwerte und -vektoren

Abbildung 6.3.4: Speedup aller Routinen bei einer Matrix mit isolierten Eigenwerten auf dem BG/Q-System. Die durchgezogenen Kurven beziehen sich auf die Matrixdimension $n = 7680$ und die gestrichelten auf $n = 11776$.

In Abbildung 6.3.5 wird die Auswirkung der Knotenanzahl auf die Laufzeit dargestellt. Die Gesamtanzahl der Prozesse wurde dabei gleich gehalten.
In beiden Abbildungsteilen wurde jeweils nur die Nutzung von `TRI_NORMAL` dargestellt. Das Verhalten der Kurven untereinander ist für die anderen Tridiagonalisierungsvarianten identisch, und würde die Abbildung nur unübersichtlicher machen ohne neue Erkenntnisse zu liefern. Daher wurden `TRI_SQ_ROW` und `TRI_SQ_COL` nicht mit einbezogen.
In Abbildungsteil (a) wird die Laufzeit für ein logisches (16×32)-Gitter dargestellt. Ob für dieses Gitter $32, 64$ oder 128 Knoten genutzt werden, beeinträchtigt die Lauf-



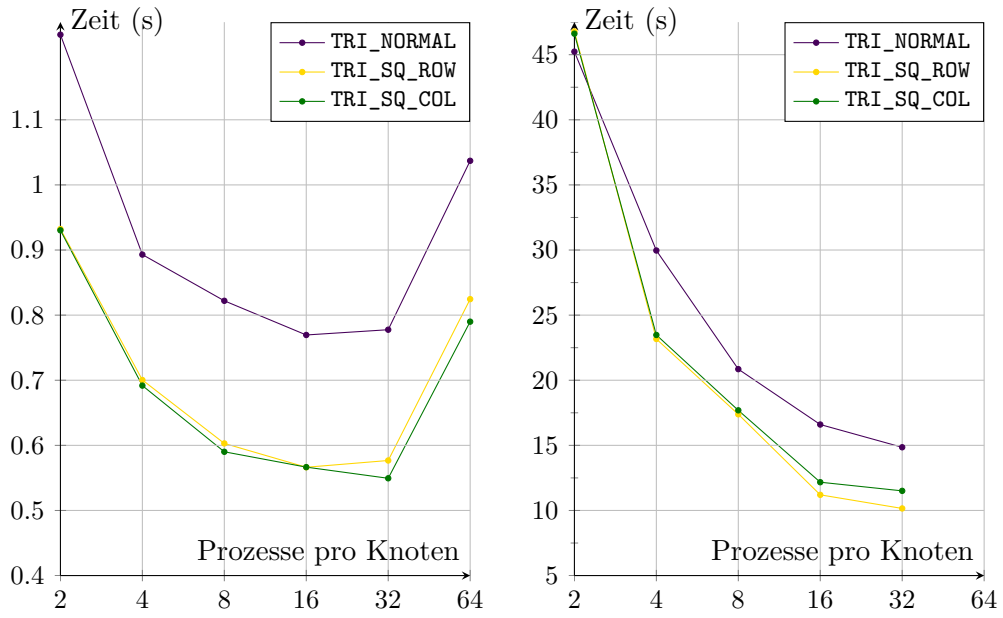
(a) 512 Prozesse, (16×32)-Gitter

(b) 1024 Prozesse, (32×32)-Gitter

Abbildung 6.3.5: Laufzeiten der Routine `HermitianEig` mit `TRI_NORMAL` bei unterschiedlicher Knotenanzahl auf dem BG/Q-System. Um die Abbildung übersichtlich zu halten, wurde nur eine Tridiagonalisierungsvariante betrachtet. Die Matrizen der Dimension 6144 bis 8192 haben jeweils isolierte Eigenwerte, welche alle berechnet wurden.

zeit leicht. Die Verwendung von 32 Knoten mit jeweils 16 Prozessen ist dabei am schnellsten.
Für Teil (b) wurde mit insgesamt 1024 Prozessen gerechnet. Hier ist die Nutzung von 64 Knoten mit 16 Prozessen die schnellste Variante.
Auffällig ist, dass in beiden Fällen die Verwendung von 16 Prozessen pro Knoten am effizientesten ist. Das heißt, dass nicht die Knotenanzahl für die unterschiedlichen Laufzeiten verantwortlich ist, sondern die Anzahl der genutzten Prozesse pro Knoten. Dies liegt daran, dass auf jedem Knoten 16 Kerne zur Berechnung vorhanden sind und daher der Einsatz von 16 Prozessen pro Knoten am effizientesten ist. Falls mehr Prozesse pro Knoten genutzt werden, wie in Teil (b) z. B. die Nutzung von 32 Prozessen auf 32 Knoten, wird die Routine deutlich langsamer, da hier pro Kern 2 Prozesse gestartet werden, und dafür SMT genutzt wird.
Die Verwendung von 64 Prozessen pro Knoten würde die Berechnung noch ineffizienter machen, da dann für insgesamt 4 Prozesse pro Kern SMT genutzt werden müsste. Der Einsatz von weniger als 16 Prozesse pro Knoten ist ebenfalls langsamer, da dann mehr Knoten verwendet werden müssen um die gleiche Prozessanzahl zu erreichen. Dies führt zu vermehrter Kommunikation zwischen den Knoten, welche sonst innerhalb eines Knotens stattgefunden hätte. Die MPI-Kommunikation

innerhalb eines Knotens ist z. B. aufgrund der Nutzung des gemeinsamen Speichers schneller als die Kommunikation über die Knotengrenzen hinaus.



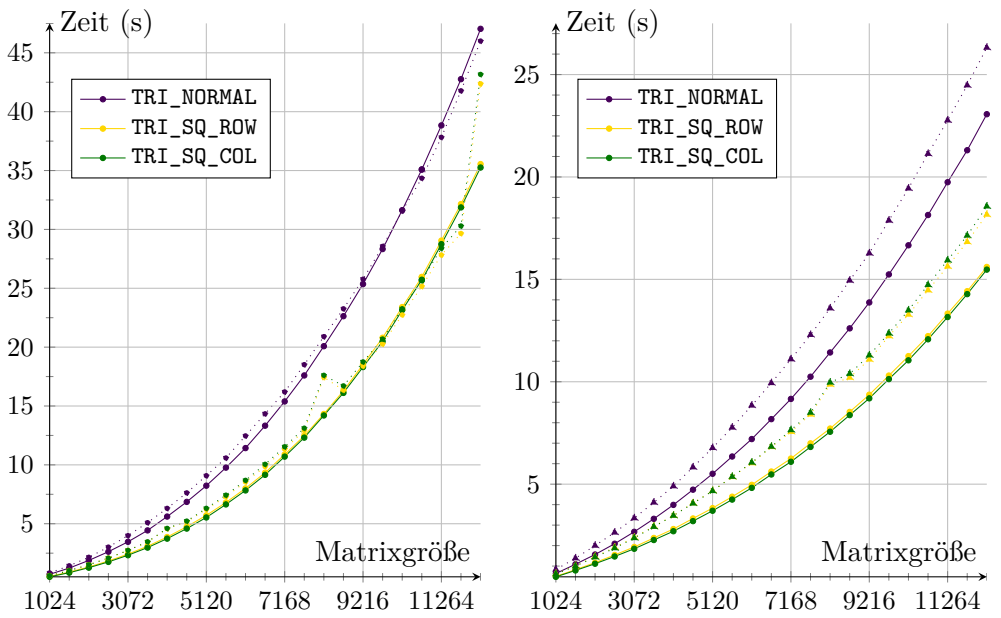
(a) Matrixdimension $n = 1024$ (b) Matrixdimension $n = 8192$

Abbildung 6.3.6: Laufzeiten der Routine `HermitianEig` für die Berechnung aller Eigenwerte und -vektoren bei einer festen Knotenanzahl von 32 auf dem BG/Q-System. Die Anzahl der verwendet Prozesse pro Knoten variiert. Die Matrizen haben jeweils isolierte Eigenwerte.

Abbildung 6.3.6 zeigt das Skalierungsverhalten für eine feste Anzahl verwendeter Knoten. Die gestarteten Prozesse pro Knoten variieren in der Abbildung.
In Abbildungsteil (a) wurde eine sehr kleine Matrix verwendet, damit ein Test mit 64 Prozessen pro Knoten möglich war. Für die Dimension von 8192 in Teil (b) war dies wegen zu geringem Speicherplatz nicht mehr möglich.
Man erkennt in Teil (a), dass die `HermitianEig` bis zu 16 Prozessoren pro Knoten relativ gut skaliert. Für 32 Prozesse wird die Routine teilweise wieder langsamer, obwohl insgesamt mehr Prozesse genutzt worden sind. Dies liegt hier, wie oben schon bemerkt, an der zweifachen Nutzung von SMT auf jedem der 16 Kerne.
Für 64, also bei einer vierfachen Verwendung von SMT, wird die Performance schon deutlich schlechter.
Die Nutzung von SMT bringt hier also keine Performanceverbesserung. Zusätzliche MPI-Prozesse bewirken bei SMT teilweise sogar das Gegenteil, sie verlangsamen die Routine.
Bei der etwas größeren Matrix, wie in Abbildungsteil (b) zu sehen ist, bringt der Einsatz von 32 Prozessen pro Knoten eine leichte Verbesserung der Performance.

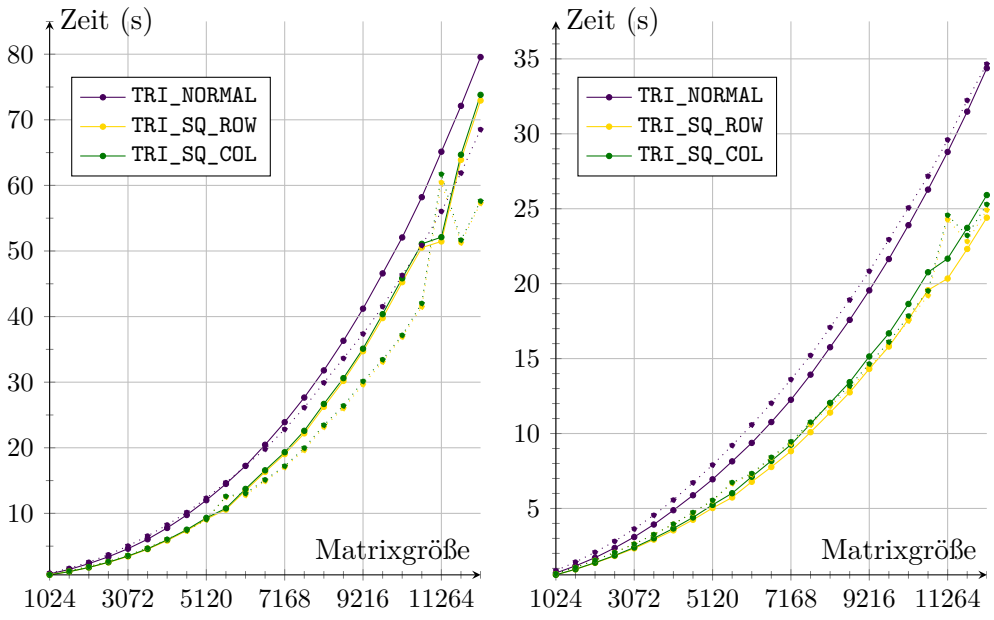
Diese Verbesserung ist allerdings nicht vergleichbar mit den Verbesserungen, welche bis zu 16 Prozessen erreicht werden.
Die Nutzung von 64 Prozessen, also von vierfachen SMT, war hier leider nicht mehr möglich, würde aber vermutlich nur eine leichte bzw. keine Performanceverbesserung bringen, da diese schon bei 32 sehr gering ist. Eventuell würde es wie in Abbildungsteil (a) sogar zu einer Verschlechterung führen.
Für deutlich größere Matrizen ist die Nutzung von SMT sicherlich sinnvoll, falls man sich schon auf eine feste Knotenanzahl festgelegt hat. Mit insgesamt 64 Prozessen pro Knoten wird man für diese feste Knotenanzahl bei sehr großen Matrizen sicherlich die maximale Geschwindigkeit erreichen. Eine Verwendung von 4 OpenMP-Threads statt der 4 MPI-Prozesse pro Kern wäre bei einer hybriden Version vermutlich noch besser. Leider ist die hybride Version der Eigenwertlöser aus Elemental noch fehlerhaft.

6.4 Vergleich BG/P - BG/Q



(a) 256 Prozesse, (16×16)-Gitter (b) 1024 Prozesse, (32×32)-Gitter

Abbildung 6.4.1: Vergleich der Laufzeiten von Elementals `HermitianEig` für die Berechnung aller Eigenwerte und -vektoren auf JUGENE (durchgezogene Linien) und dem Blue Gene/Q System (gestrichelte Linien). Alle Matrizen haben isolierte Eigenwerte, welche zwischen 0 und 1 gleichverteilt generiert wurden. Die Messungen wurden auf JUGENE mit 256 für (a) und 1024 Knoten für (b) ausgeführt. Auf dem Blue Gene/Q Prototyp wurden für (a) 32 und für (b) 128 Knoten genutzt.

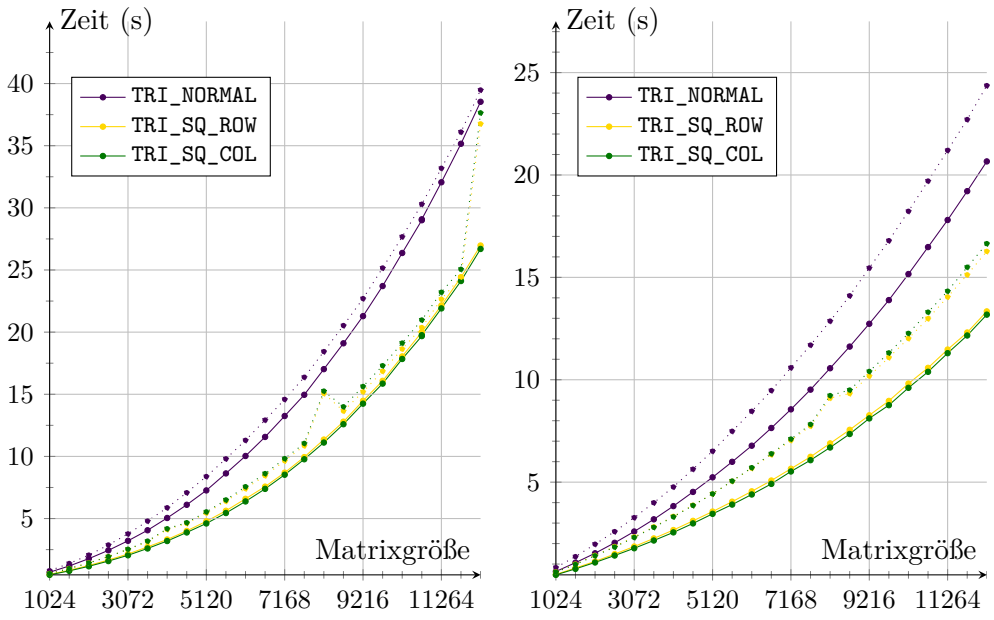


(a) 128 Prozesse, (8 × 16)-Gitter

(b) 512 Prozesse, (16 × 32)-Gitter

Abbildung 6.4.2: Vergleich der Laufzeiten von Elementals `HermitianEig` für die Berechnung aller Eigenwerte und -vektoren auf JUGENE (durchgezogene Linien) und dem Blue Gene/Q System (gestrichelte Linien). Alle Matrizen haben isolierte Eigenwerte, welche zwischen 0 und 1 gleichverteilt generiert wurden. Die Messungen wurden auf JUGENE mit 128 für (a) und 512 Knoten für (b) ausgeführt. Auf dem Blue Gene/Q Prototyp wurden für (a) 32 und für (b) 128 Knoten genutzt.

In diesem Abschnitt werden die schon jeweils einzeln betrachteten Ergebnisse vom JUGENE und von dem Blue Gene/Q Prototyp verglichen. Da ScaLAPACK auf dem Blue Gene/Q System bisher nicht zum Einsatz gekommen ist, wird in diesem Abschnitt nur `HermitianEig` mit den drei Tridiagonalisierungsalgorithmen aus Elemental betrachtet.
In Abbildung 6.4.1 werden die Ergebnisse für quadratische Gitter verglichen. Auf dem kleineren Prozessorgitter ist die Routine für große Matrizen ab Dimension 9728 jeweils auf dem Blue Gene/Q bei Verwendung der gleichen Anzahl von Kernen schneller, was in Abbildungsteil (a) zu sehen ist. Allerdings treten wieder die in Abschnitt 6.2.2 besprochenen Verzögerungen auf dem Prototyp auf.
Für kleine Matrixdimensionen ist `HermitianEig` mit allen drei Tridiagonalisierungsalgorithmen jeweils auf JUGENE schneller.
Für das größere Prozessorgitter, dargestellt in Teil (b), ist der JUGENE deutlich schneller. Anhand der Steigung der Kurven im Bereich der größten Matrixdimensionen lässt sich allerdings vermuten, dass für größere Dimensionen der Blue Gene/Q Prototyp wieder schneller wird. Die Steigung der Kurve des JUGENEs ist in diesem Bereich höher.



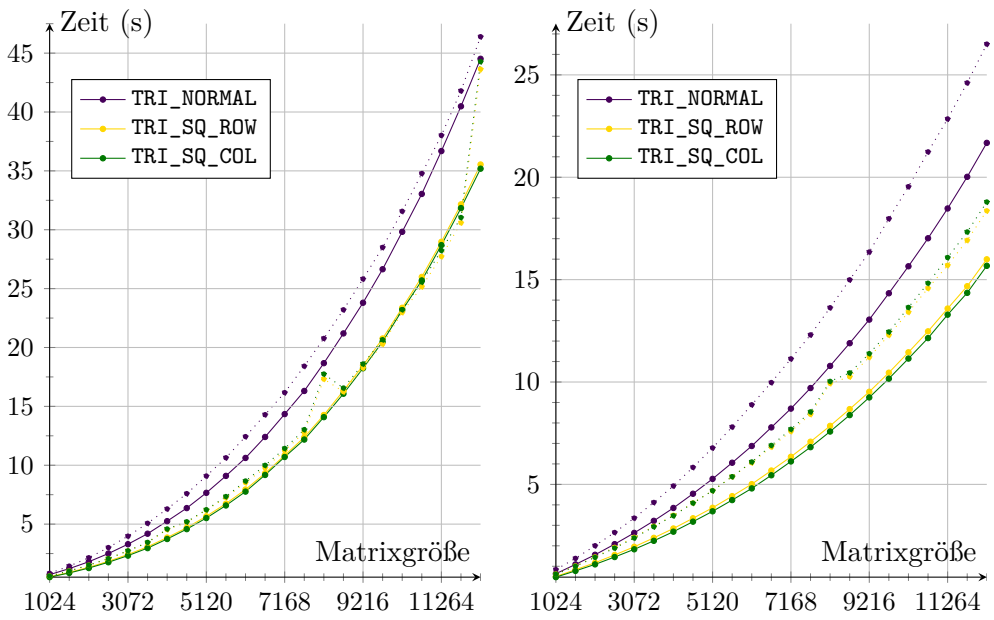
(a) 256 Prozesse, (16×16)-Gitter (b) 1024 Prozesse, (32×32)-Gitter

Abbildung 6.4.3: Vergleich der Laufzeiten von Elementals `HermitianEig` für die Berechnung von 10% der Eigenwerte und -vektoren auf JUGENE (durchgezogene Linien) und dem Blue Gene/Q System (gestrichelte Linien). Alle Matrizen haben isolierte Eigenwerte, welche zwischen 0 und 1 gleichverteilt generiert wurden. Die Messungen wurden auf JUGENE mit 256 für (a) und 1024 Knoten für (b) ausgeführt. Auf dem Blue Gene/Q Prototyp wurden für (a) 32 und für (b) 128 Knoten genutzt.

Die Abbildung 6.4.2 zeigt den Vergleich auf den verschiedenen Systemen, wobei hier für die Messungen nichtquadratische Prozessorgitter genutzt wurden. Das Verhalten der Kurven zueinander ist ähnlich zu dem in Abbildung 6.4.1 festgestellten Verhalten.
In Abbildungsteil (a) ist der Schnittpunkt der Kurven bei Dimension 6144 zu erkennen. Für kleine Dimensionen ist der JUGENE jeweils schneller. Für größere, insbesondere bei $n = 12288$, läuft `HermitianEig` auf dem Blue Gene/Q System deutlich performanter.
Für das größere Prozessorgitter in Teil (b) ist der Schnittpunkt der Kurven bis Dimension 12288 zwar noch nicht erreicht, aber die Kurven liegen in diesem Bereich schon sehr nah beieinander. Daher lässt sich vermuten, das auch hier der Blue Gene/Q für größere Dimensionen schneller wäre.
In Abbildung 6.4.1, Teil (b) war dies noch nicht so deutlich zu erkennen. Die Vermutung, dass sich die Kurven dort für noch größere Matrixdimensionen schneiden, wird durch die Abbildung 6.4.2 unterstützt.
Das Verhalten der Kurven ist in allen Abbildungen gleich. Der Schnittpunkt der

Kurven, also die Dimension, ab welcher der Blue Gene/Q Prototyp schneller als der JUGENE ist, wird durch die Nutzung eines größeren Prozessorgitter weiter nach rechts verschoben.

Zur Vollständigkeit werden in den Abbildungen 6.4.3 bzw. 6.4.4 die Vergleiche zwischen den beiden Systemen dargestellt, wobei nur 10% aller Eigenwerte berechnet werden bzw. die Eigenwerte in einem Cluster liegen. Es wurden jeweils die beiden quadratischen Gitter aus 6.4.1 betrachtet.
Man sieht grundsätzlich ein ähnliches Verhalten, wie es in allen Abbildungen zum Vergleich zwischen dem JUGENE und dem Blue Gene/Q Prototyp zu erkennen ist. Der Schnittpunkt der beiden Kurven tritt im Vergleich zu Abbildung 6.4.1 jeweils erst bei größeren Matrixdimensionen auf.



(a) 256 Prozesse, (16×16)-Gitter (b) 1024 Prozesse, (32×32)-Gitter

Abbildung 6.4.4: Vergleich der Laufzeiten von Elementals `HermitianEig` für die Berechnung aller Eigenwerte und -vektoren auf JUGENE (durchgezogene Linien) und dem Blue Gene/Q System (gestrichelte Linien). Alle Matrizen haben um 0 geclusterte Eigenwerte. Die Messungen wurden auf JUGENE mit 256 für (a) und 1024 Knoten für (b) ausgeführt. Auf dem Blue Gene/Q Prototyp wurden für (a) 32 und für (b) 128 Knoten genutzt.

6.5 Scalasca-Analyse

In diesem Abschnitt werden einige Ergebnisse präsentiert, welche mit dem Tool Scalasca [66] zur Analyse paralleler Programme erzeugt wurden.
Scalasca ist ein Tool zur Analyse von parallelen Programmen, welche MPI, OpenMP [70] oder eine Kombination aus beiden, also eine hybride Parallelisierung, verwenden. Es unterstützt die Sprachen C, C++ und Fortran und einige HPC Systeme, wie den Blue Gene/P bzw. Blue Gene/Q.
Zur Analyse muss das auszuführende Programm zuvor instrumentiert werden. Das bedeutet, dass der Code des Programms modifiziert wird um relevante Ereignisse wie Kommunikationen aufzunehmen. Statt das Programm normal zu kompilieren wird es mit `scalasca -instrument <compiler> <programm>` instrumentiert und kompiliert.
Wenn die so entstandene ausführbare Datei anschließend wie gewohnt ausgeführt wird, wird ein Verzeichnis angelegt, in dem die Informationen der Analyse gespeichert werden. Diese werden nun mit `scalasca -examine` nachbearbeitet, bevor die Ergebnisse mit dem Programm CUBE3 angezeigt werden. Bei der oben beschriebenen Instrumentierung des ausführbaren Programms werden nur die MPI Aufrufe aufgezeichnet. Um weitere Informationen über die anderen Programmkonstrukte zu erhalten, müssen die Unterroutinen, welche von Interesse sind ebenfalls beim Kompilieren instrumentiert werden.
Die Abbildungen in diesem Kapitel sind mit CUBE3 erstellt worden und zeigen jeweils einzelne Routinen oder MPI Aufrufe mit ihren Laufzeiten oder Prozessorauslastungen. Eine Beschreibung von CUBE3 ist in der Dokumentation [71] zu finden. Weitere Information zu Scalasca und der Instrumentierung sind in der Dokumentation [72] zu finden.

6.5.1 ScaLAPACK

Zur Analyse des Problems von `PDSYEVX` bei Matrizen mit geclusterten Eigenwerten wurde speziell die Routine `PDSTEIN`, welche alle Eigenvektoren mittels Inverser Iteration berechnet und gegebenenfalls nachorthogonalisiert, mit Scalasca analysiert. Abbildung 6.5.1 zeigt in der linken Spalte, in welchem Verhältnis die reine Ausführungszeit und die reine Kommunikationszeit zueinander stehen. Da $97,18\%$ der Gesamtlaufzeit in Kommunikation verbracht werden, stehen diese Zeiten in keinem guten Verhältnis zueinander. Die hier als Kommunikation angegebene Zeit besteht allerdings zum größten Teil aus Wartezeit.
In diesem Beispiel werden Eigenvektoren einer Matrix mit geclusterten Eigenwerten berechnet, daher sind die Eigenvektoren, welche mittels Inverser Iteration berechnet wurden, nicht alle orthogonal zueinander.
In der rechten Spalte wird die reine Berechnungszeit der Routine `PDSTEIN` für die 64 Prozesse farblich dargestellt. Die Farben stellen hier Prozente der Laufzeit des Prozessors mit der maximalen Laufzeit dar. Man sieht, dass der erste Prozessor der mit der längsten Berechnungszeit ist, an welchem sich die anderen orientieren, da er mit rot eingefärbt ist, was für nahezu 100% steht. Alle anderen Prozessoren liegen

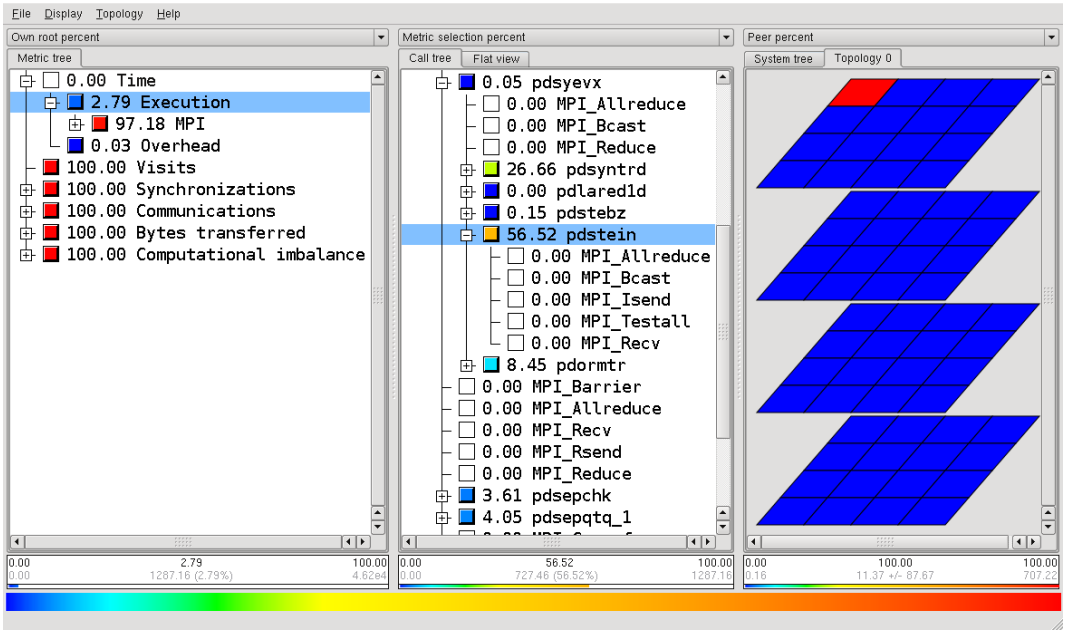


Abbildung 6.5.1: Scalasca Analyse zur Routine `PDSYEVX` bei geclusterten Eigenwerten und einer Matrixdimension von 3584. Die Routine wurde auf JUGENE mit 64 Knoten mit je einem Prozess ausgeführt. Die Laufzeit der ausgeführten Berechnungen wird dargestellt.

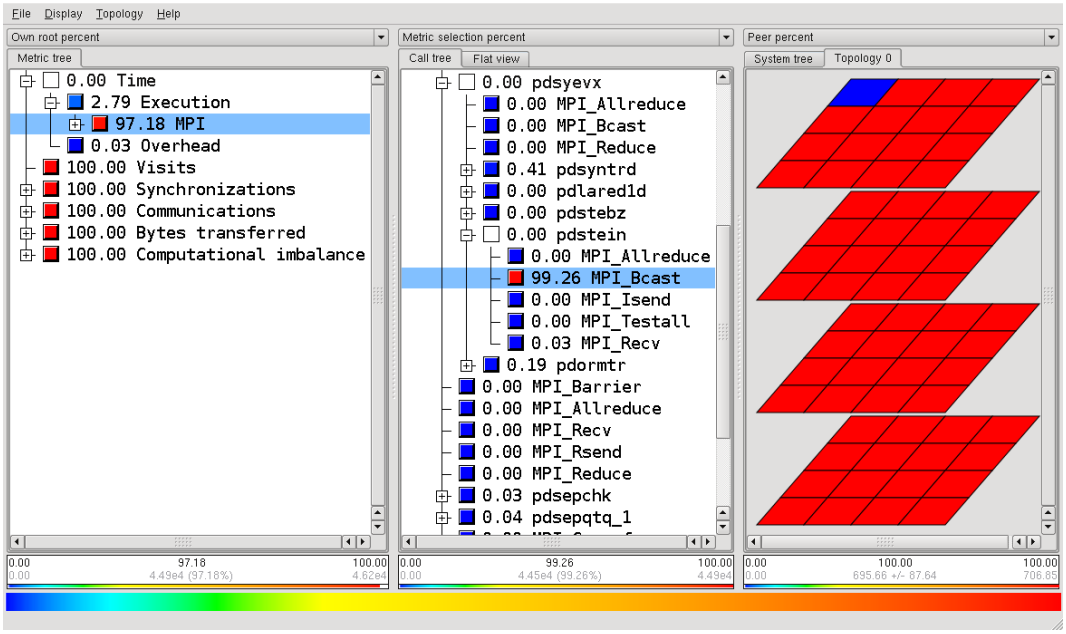


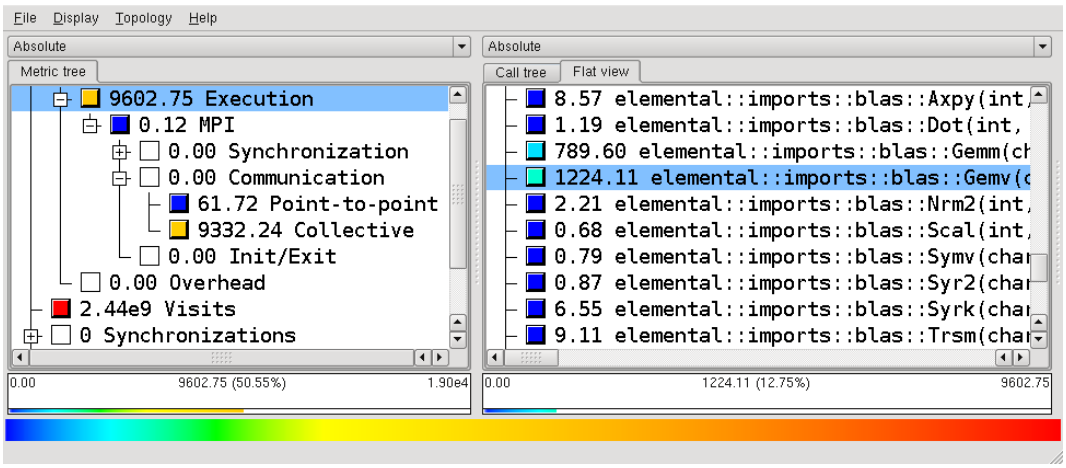
Abbildung 6.5.2: Scalasca Analyse zur Routine `PDSYEVX` bei geclusterten Eigenwerten und einer Matrixdimension von 3584. Die Routine wurde auf JUGENE mit 64 Knoten mit je einem Prozess ausgeführt. Die Zeit, welche die Kommunikation benötigt, wird dargestellt.

in einem sehr niedrigen Prozentbereich (blau), da sie im Vergleich zum ersten Prozessor fast keine Zeit für Berechnungen benötigen.
Dies zeigt, dass ein großer Teil der Routine `PDSTEIN` nur auf einem Prozess ausgeführt wird. Wie schon erwähnt, handelt es sich dabei um die Nachorthogonalisierung der Eigenvektoren.
In Abbildung 6.5.2 ist die Untersuchung der selben Routine auf die Kommunikationszeit dargestellt. Hier erkennt man, dass die Verteilung der Laufzeit für einen Broadcast genau entgegengesetzt zu der Laufzeit der Berechnungen ist.
Während der erste Prozessor die Eigenvektoren orthogonalisiert und sie danach mittels eines Broadcasts wieder auf die anderen Prozessoren verteilt, warten diese. Diese Wartezeit wird von Scalasca als Laufzeit zum Broadcast gewertet.
Die Zeit, welche für Kommunikation verwendet wird, liegt bei $97,18\%$ der Gesamtlaufzeit, weil die Laufzeiten der Prozessoren für die Darstellung in der linken Spalte aufaddiert werden.

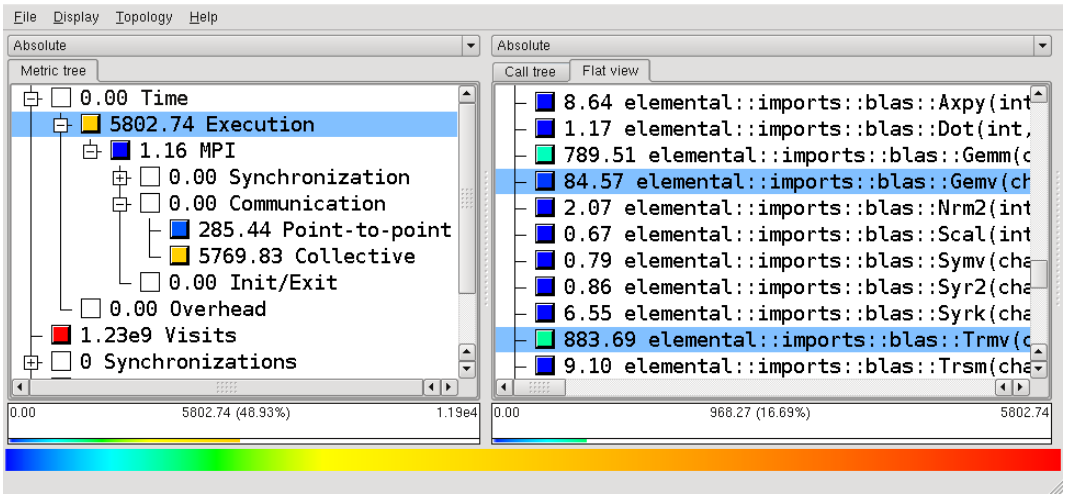
6.5.2 Elemental

Um die Unterschiede zwischen `TRI_NORMAL` und `TRI_SQ_ROW` genauer zu analysieren wurden die beiden Tridiagonalisierungsroutinen mit Scalasca instrumentiert.
Die Abbildungen 6.5.3 und 6.5.4 zeigen jeweils Ausschnitte der Scalasca Analyse für HermitianEig. Für die Abbildungsteile (a) wurde jeweils der Tridiagonalisierungsalgorithmus `TRI_NORMAL` genutzt, während für Teil (b) `TRI_SQ_ROW` angewendet wurde.
Bei der Angabe von Zeiten (Absolute) handelt es sich immer um die aufaddierte Laufzeit aller Prozessoren in Sekunden. Da für die Analyse der beiden Routinen jeweils die gleiche Matrixgröße und das gleiche Prozessorgitter verwendet wurden, lassen sich die Zeiten direkt vergleichen. Man sieht so, dass die Laufzeit für die Berechnungen durch die Benutzung des quadratischen Algorithmus von ca. 9603 auf 5803 Sekunden zurück geht.
In Abbildung 6.5.3 sind in der rechten Spalte jeweils die BLAS Aufrufe mit ihren insgesamt darauf verwendeten Zeiten aufgelistet, in der linken Spalte ist die Gesamtlaufzeit in Sekunden dargestellt. Durch einen Vergleich zwischen (a) und (b) erkennt man die wesentlichen Unterschiede zwischen beiden Tridiagonalisierungsalgorithmen. In den BLAS Aufrufen sind nur Unterschiede bei den mit blau markierten `GEMV` und `TRMV` zu erkennen. Während `TRMV` für `TRI_NORMAL` gar nicht genutzt wird, dominiert `TRMV` die Laufzeit aller BLAS Aufrufe bei der Verwendung von `TRI_SQ_ROW`. Dagegen wird durch Reduzierung der `GEMV` Aufrufe einige Zeit eingespart.
Die für die Kommunikation verwendete Zeit ist in die Kategorien Punkt-zu-Punkt und kollektive Kommunikation eingeteilt. Hier ist ebenfalls ein gravierender Unterschied zwischen den beiden Algorithmen zu erkennen. Die Zeit für kollektive Operationen geht bei Verwendung von `TRI_SQ_ROW` stark zurück, während die Zeit für Punkt-zu-Punkt Kommunikation leicht ansteigt. Dies wird mit Hilfe der Darstellung der einzelnen MPI Aufrufe in Abbildung 6.5.4 detaillierter betrachtet.
Die größten Unterschiede zwischen den beiden genutzten Algorithmen sind wieder in der rechten Spalte blau markiert. Für die kollektive Kommunikation `MPI_Allgather`

und `MPI_Bcast` wird wesentlich mehr Zeit aufgewendet, wenn mit `TRI_NORMAL` tridiagonalisiert wird. Dagegen wird bei den quadratischen Reduktionsalgorithmen deutlich mehr Zeit für `MPI_Sendrecv` aufgewendet, wobei es sich um eine Punkt-zu-Punkt Kommunikation handelt.



(a) `TRI_NORMAL`

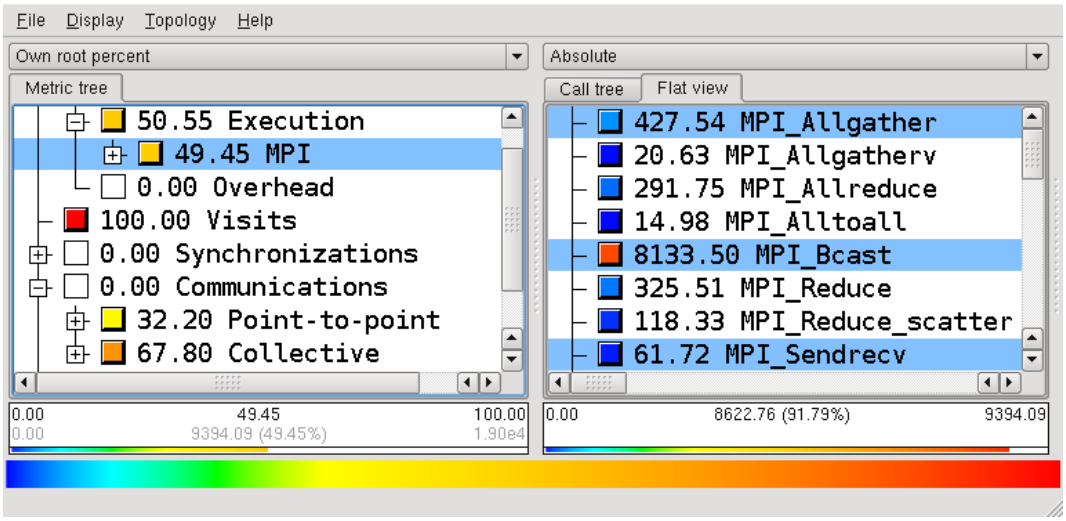


(b) `TRI_SQ_ROW`

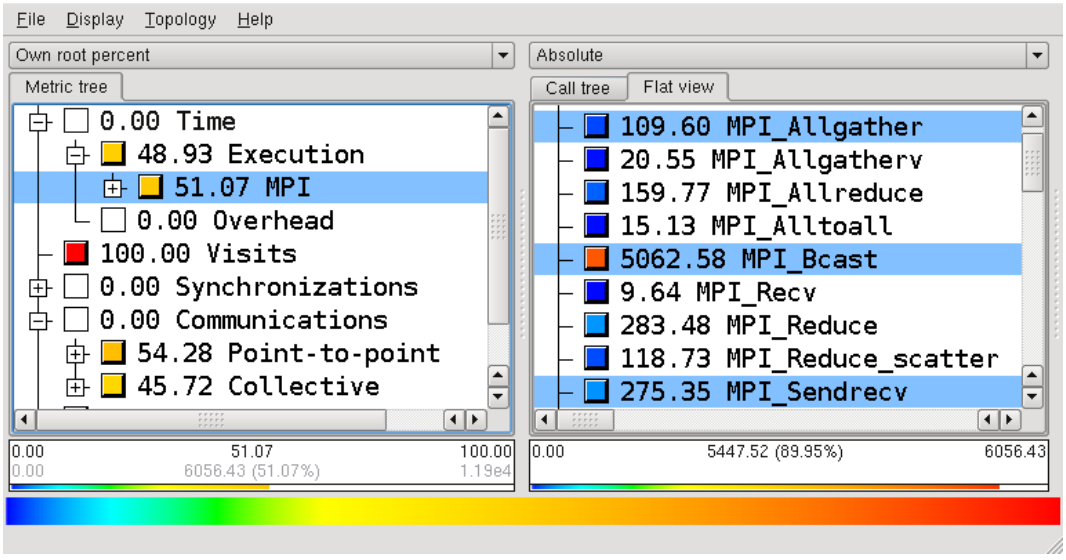
Abbildung 6.5.3: Scalasca Analyse zur Routine `HermitianEig` bei isolierten Eigenwerten und einer Matrixdimension von 8192. Die Routine wurde auf JUGENE mit 1024 Knoten mit je einem Prozess ausgeführt. Die BLAS Aufrufe werden dargestellt.

Dies zeigt auch die Darstellung in der linken Spalte, in welcher die Häufigkeit der Kommunikationen aus beiden Kategorien prozentual dargestellt wird. Dies ist nicht mit der aufgewendeten Zeit für die Kommunikation in beiden Kategorien zu verwechseln, weshalb sich Prozentwerte ergeben, die nicht zu den absoluten Werten der

Laufzeit der MPI Aufrufe in der rechten Spalte passen. Dies liegt daran, dass der Zeitaufwand für kollektive Kommunikation größer ist und der Zeitaufwand immer von der Größe des versendeten Objekts abhängt.



(a) `TRI_NORMAL`



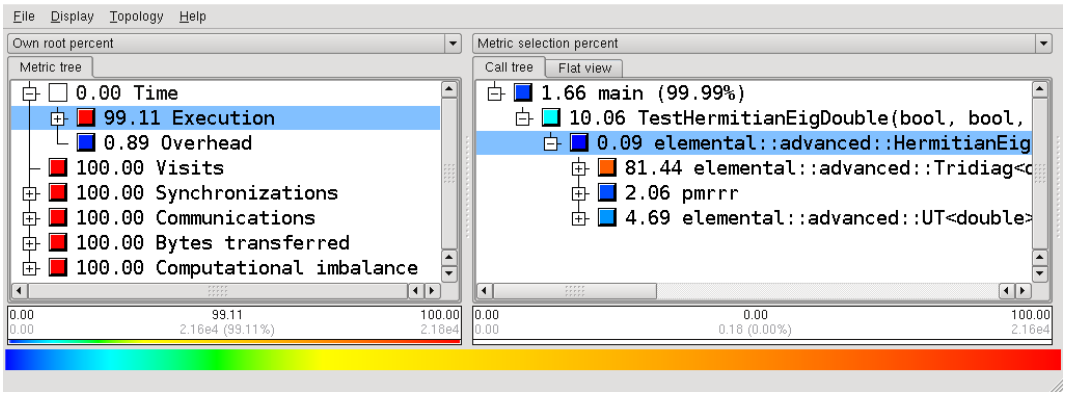
(b) `TRI_SQ_ROW`

Abbildung 6.5.4: Scalasca Analyse zur Routine `HermitianEig` bei isolierten Eigenwerten und einer Matrixdimension von 8192. Die Routine wurde auf JUGENE mit 1024 Knoten mit je einem Prozess ausgeführt. Die MPI Aufrufe werden dargestellt.

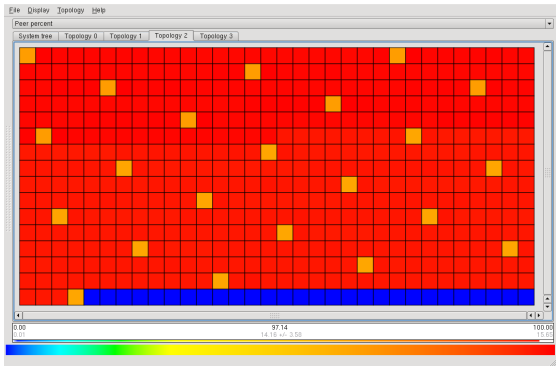
Die kollektive Kommunikation wird bei der Anwendung von `TRI_NORMAL` doppelt so häufig wie Punkt-zu-Punkt Kommunikation genutzt. Beim Gebrauch von `TRI_SQ_ROW`

hat die Punkt-zu-Punkt Kommunikation dagegen einen Anteil von fast 55% der Aufrufe.

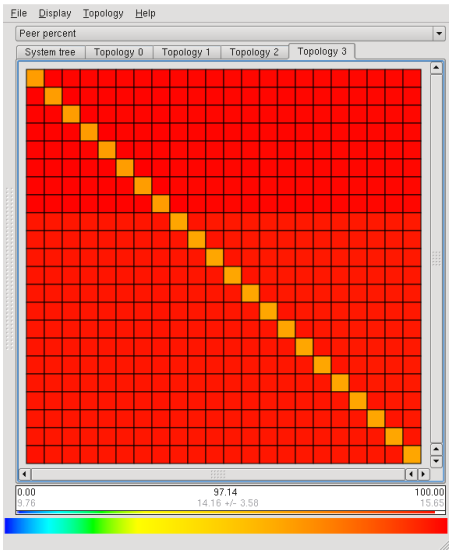
Um besser zu verstehen, was bei der Verwendung der quadratischen Reduktionsalgorithmen auf nichtquadratischen Prozessorgittern passiert, wurde `TRI_SQ_ROW` auf JUGENE mit 512 Knoten mit Scalasca analysiert. Die Ergebnisse der Routine `HermitianEig` mit `TRI_SQ_ROW` werden in Abbildung 6.5.5 dargestellt.



(a) Verhältnis der Verfahrensschritte



(b) Auslastung aller Prozessoren



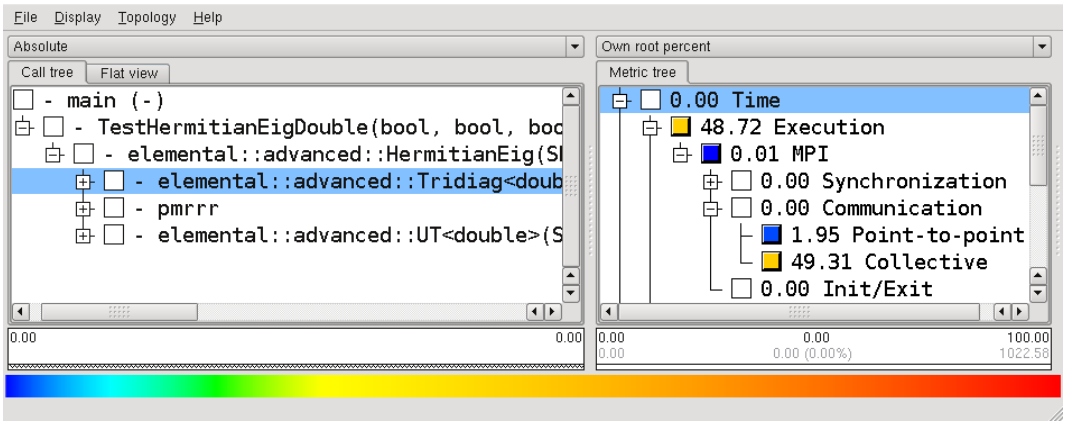
(c) Auslastung im neuen Gitter

Abbildung 6.5.5: Scalasca Analyse zur Routine `HermitianEig` mit `TRI_SQ_ROW` bei isolierten Eigenwerten und einer Matrixdimension von 8192. Die Routine wurde auf JUGENE mit 512 Knoten mit je einem Prozess, das heißt auf einem nichtquadratischen Prozessorgitter ausgeführt.

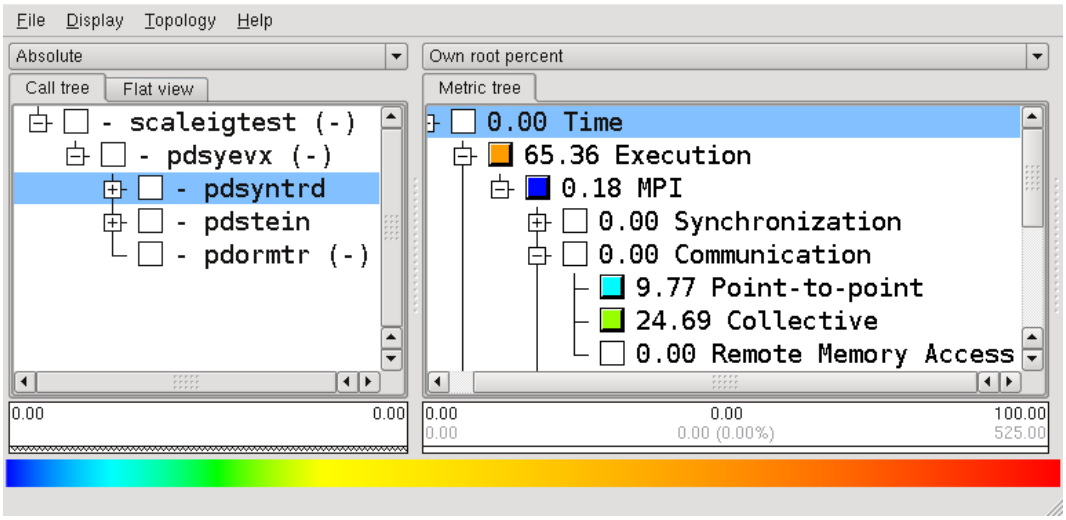
Abbildungsteil (a) zeigt das Verhältnis der einzelnen Verfahrensschritte zueinander. Hier sieht man, dass selbst mit der schnelleren Tridiagonalisierung mittels `TRI_SQ_ROW`, diese noch über 80% der gesamten Laufzeit des Testprogramms benötigt. Ohne den durch die Initialisierung und den durch die Tests auf Korrektheit der Ergebnisse entstandenen Overhead liegt der Anteil der Tridiagonalisierung bei fast 90%.
Abbildungsteil (b) zeigt die dazu gehörige Auslastung der Prozessoren für die Tridiagonalisierung. Die Auslastung bezieht sich auf die Ausführungszeit der durchgeführten Berechnungen. Die 512 Prozessoren sind in einem (16×32)-Gitter angeordnet. Die letzten 28 Prozessoren scheinen fast keine Berechnungen durchzuführen, da sie mit blau eingefärbt sind, was für fast 0% der maximalen Ausführungszeit eines Prozessors steht. In regelmäßigen Abständen von 23 Prozessoren existiert jeweils ein Prozessor, der etwas weniger im Vergleich zu allen anderen ausgelastet ist.
In Abbildungsteil (c) wird schließlich deutlich, warum dies der Fall ist. Dieser Abbildungsteil zeigt das quadratische Prozessorgitter, welches für die Tridiagonalisierung mittels `TRI_SQ_ROW` genutzt wird. Die Prozessoren, welche weniger ausgelastet sind, liegen in dem (22×22)-Gitter hier auf der Diagonalen. Diese Prozessoren wenden weniger Zeit für Berechnungen auf, sind aber dafür stärker ausgelastet, was den Kommunikationsaufwand betrifft.
Die in Teil (b) mit blau gekennzeichneten Prozessoren, sind hier nicht mehr dargestellt, da sie an der Berechnung auf diesem quadratischen Gitter nicht beteiligt sind.

6.5.3 Vergleich einer Reduktionsroutine aus ScaLAPACK und Elemental

Zuletzt soll noch untersucht werden, wie sich die unterschiedliche Datenaufteilung zwischen Elemental und ScaLAPACK bei Verwendung des gleichen Reduktionsalgorithmus auf das Verhältnis von Rechnung zu Kommunikation auswirkt.
Abbildung 6.5.6 zeigt die jeweiligen Anteile der Laufzeit, welche für Kommunikation bzw. Berechnungen verwendet werden. Die beiden schnelleren Reduktionsroutinen aus Elemental und ScaLAPACK, welche auf quadratischen Gittern laufen, werden in der Abbildung gegenübergestellt. In der rechten Spalte sind die einzelnen Anteile der verwendeten Zeit für die Tridiagonalisierung aufgelistet.
Man erkennt, dass Elemental für die Tridiagonalisierung einen größeren Kommunikationsaufwand hat als ScaLAPACKs `PDSYNRD`. Dies liegt an der elementweisen Aufteilung der Matrizen. Die einzelnen Elemente müssen häufiger durch globale Kommunikation auf anderen Prozessoren bekannt gemacht werden, weshalb besonders der Anteil der kollektiven Kommunikation hoch ist.
In ScaLAPACK liegt der Anteil der Laufzeit, die für Berechnungen verwendet wird, bei ca. 65%, während er bei der Tridiagonalisierung in Elemental bei ca. 50% liegt. In ScaLAPACK werden ganze Blöcke häufiger durch Punkt-zu-Punkt Kommunikation verschickt, sodass dieser Anteil deutlich größer als bei Elemental ist.



(a) Tridiagonalisierung mit `TRI_SQ_ROW` aus Elemental



(b) Tridiagonalisierung mit `PDSYNTRD` aus ScaLAPACK

Abbildung 6.5.6: Scalasca Analyse zur Routine `HermitianEig` mit `TRI_SQ_ROW` (a) bzw. `PDSYEVX` mit `PDSYNTRD` bei isolierten Eigenwerten und einer Matrixdimension von 3500. Die Routinen wurden auf JUGENE mit 64 Knoten mit je einem Prozess ausgeführt.

Kapitel 7

Zusammenfassung und Ausblick

Dieses Kapitel stellt den Abschluss meiner Arbeit dar und gibt einen Überblick über die im Rahmen der Untersuchungen für diese Arbeit entstandenen Ergebnisse. Anschließend wird ein Ausblick darüber gegeben, welche Untersuchungen zukünftig noch durchzuführen sind.

7.1 Zusammenfassung

Im Rahmen dieser Masterarbeit wurden die parallelen Eigenwertlöser der Bibliotheken ScaLAPACK und Elemental auf dem JUGENE-System und einem Blue Gene/Q Prototyp getestet.
Während ScaLAPACK eine zweidimensionale blockzyklische Verteilung der Matrizen auf den verteilten Speicher nutzt, verfolgt die noch in der Entwicklung befindliche Bibliothek Elemental einen elementweisen zweidimensional zyklischen Ansatz zur Verteilung der Matrizen. Dadurch erreicht man immer die bestmögliche Lastverteilung für die Prozessoren, was sonst bei größeren Blöcken nicht gegeben ist. Aufgrund der Unabhängigkeit der algorithmischen Blockgröße von der Verteilung, ist es mit Elemental dennoch möglich, Blockalgorithmen und insbesondere Level 3 BLAS Operationen mit größeren Blöcken zu verwenden.
Die Untersuchungen für die einzelnen Routinen zeigen, dass aus ScaLAPACK nur `PDSYEVX`, welche die Bisektion und Inverse Iteration nutzt, akzeptable Laufzeiten erreicht. Die Routine bekommt allerdings große Probleme, falls die zu berechnenden Eigenwerte geclustert vorliegen und die zugehörigen Eigenvektoren von Interesse sind.
Für geclusterte Eigenwerte ist `PDSYEVR`, welche den MR^3-Algorithmus nutzt, vorzuziehen. Da diese Routine noch kein offizieller Bestandteil von ScaLAPACK ist, bleibt es abzuwarten, ob die hier auftretenden Schwankungen und Probleme mit vielen Prozessoren noch beseitigt werden können, sodass diese Routine auch für isolierte Eigenwerte konkurrenzfähig zu `PDSYEVX` ist.
Die Routine, welche insgesamt in den meisten Testsituationen am besten abgeschnitten hat, ist Elementals `HermitianEig` mit dem Tridiagonalisierungsalgorithmus `TRI_SQ_ROW`. Sie nutzt ebenfalls den MR^3-Algorithmus.

Der Vergleich für `HermitianEig` auf dem JUGENE-System und dem Blue Gene/Q-System zeigt, dass die neue Blue Gene/Q-Architektur insbesondere für große Matrizen eine bessere Performance pro Rechenkern erreicht. Dies würde in einer Endversion des Systems noch besser werden, da der Prototyp aufgrund einiger Einschränkungen noch nicht die volle Leistungsfähigkeit erreicht. Insbesondere die vorläufige ESSL Version und die nicht optimierte globale Kommunikation beeinträchtigen die Performance von `HermitianEig` erheblich, da ESSL-Aufrufe und globale Kommunikation einen Großteil der verwendeten Zeit für diese Routine benötigen.
Die Performance-Analyse auf dem Prototyp des Blue Gene/Q Systems zeigt, dass die Nutzung von genau 16 Prozessen pro Knoten optimal ist, was an den 16 vorhandenen Kernen auf jedem Knoten des Systems liegt. Die Verwendung von vielfachen SMT für MPI-Prozesse verlangsamt die Routine in einigen Fällen sogar.

7.2 Ausblick

Testdurchläufe der Routinen aus ScaLAPACK zeigen, dass sie ebenfalls skalieren, wenn man statt mehr MPI Prozesse mehrere Threads nutzt, und die ESSL SMP Bibliothek statt der normalen ESSL Bibliothek verwendet. Es handelt sich dabei nicht um ein vollständig hybrides Programm, da nur die ESSL-Aufrufe hybrid genutzt werden.
Dies könnte man in Zukunft auch für Elemental, insbesondere auf einem Blue Gene/Q System testen, da dort die Nutzung von 16 MPI-Prozessen pro Knoten optimal ist. Durch zusätzliche 4 OpenMP-Threads auf jedem Kern, könnten die BLAS Operationen mit der ESSL SMP eventuell schneller ausgeführt werden. In Elemental ist zusätzlich eine vollständig hybride Version implementiert, welche für den Eigenwertlöser `HermitianEig` leider noch fehlerhaft ist. Da Elemental sich noch in der Entwicklung befindet, ist hier zukünftig eine funktionierende hybride Version von `HermitianEig` zu erwarten. Man sollte diese Version dann auch darauf untersuchen, ob nicht eventuell eine noch geringere Anzahl von MPI-Prozessen mit mehr Threads pro MPI-Prozess noch günstiger ist.
Die in C++ implementierte Bibliothek Elemental stellt ein Interface für C bereit, sodass für Fortran- und C-Programmierer die Verwendung von Elemental ermöglicht wird. Dies wird für zukünftige Verwendungen der Bibliothek sehr wichtig sein, da viele Anwendungen im Jülich Supercomputing Centre in Fortran geschrieben sind.
Im Rahmen dieser Arbeit wurden symmetrische Eigenwertprobleme gelöst. Häufig sind für die Anwendungen allerdings hermitesche Eigenwertprobleme zu lösen, sodass die Routinen in Zukunft auch für komplexe Werte getestet werden sollten, da die Performance für komplexe Matrizen schlechter wird.
Zur Zeit wird mit *Eigenvalue soLvers for Petaflop Applications* (ELPA) eine weitere Bibliothek in Fortran entwickelt, welche sich auf die Lösung von Eigenwertproblemen auf parallelen Systemen spezialisiert [73], [74]. Diese nutzt zur Tridiagonalisierung der symmetrischen Matrix unter anderem eine zweistufige Reduktionsroutine, welche die symmetrische Matrix im ersten Schritt auf eine Bandmatrix reduziert. Im zweiten Schritt wird die symmetrische Bandmatrix schließlich auf die symmetrische Tridia-

gonalgestalt gebracht. Die Rücktransformation wird dadurch aufwendiger, weshalb sich die zweistufige Tridiagonalisierung nur für die Berechnung weniger Eigenwerte und -vektoren anbietet.
Diese Bibliothek sollte in Zukunft ebenfalls auf Laufzeit- und Skalierungsverhalten getestet und mit den anderen Bibliotheken verglichen werden.

Literaturverzeichnis

[1] Niehaus, T. A. ; Suhai, S. ; Della Sala, F. ; Lugli, P. ; Elstner, M. ; Seifert, G. ; Frauenheim, Th.: Tight-binding approach to time-dependent density-functional response theory. In: *Phys. Rev. B* 63 (2001), Feb, Nr. 8, S. 085108 1

[2] Fischer, G.: *Lineare Algebra: Eine Einführung für Studienanfänger.* Vieweg+ Teubner, 2009 2

[3] Jänich, K.: *Lineare Algebra.* Springer, 2008 2

[4] Bjorck, A.: Numerics of gram-schmidt orthogonalization. In: *Linear Algebra and Its Applications* 197 (1994), S. 297–316 2.2.5, 5

[5] Parlett, B.N.: *The symmetric eigenvalue problem.* Bd. 20. Society for Industrial Mathematics, 1998. – ISBN 0898714028 1, 3.2, 3.3.2

[6] Watkins, D.S.: *Fundamentals of matrix computations.* LibreDigital, 2002. – ISBN 0471213942 3.2

[7] Householder, A.S.: *The theory of matrices in numerical analysis.* Blaisdell Pub. Co., 1964 3.2

[8] Golub, G.H. ; Van Loan, C.F.: *Matrix computations.* Johns Hopkins Univ Pr, 1996. – ISBN 0801854148 3.2, 11, 2, 5, 3.3.4

[9] Lang, B. et al.: Direct Solvers for Symmetric Eigenvalue Problems in Modern Methods and Algorithms of Quantum Chemistry. In: *J. Grotendorst (Editor), Proceedings, NIC Series Volume* Citeseer, 2000 11, 5, 3.3.3, 4

[10] Dongarra, J.J. ; Du Croz, J. ; Hammarling, S. ; Duff, I.S.: A set of level 3 basic linear algebra subprograms. In: *ACM Transactions on Mathematical Software (TOMS)* 16 (1990), Nr. 1, S. 1–17 11

[11] Bischof, C.H. ; Van Loan, C.: The WY representation for products of Householder matrices. In: *Selected Papers from the Second Conference on Parallel Processing for Scientific Computing* Society for Industrial and Applied Mathematics, 1985, S. 2–13 11

[12] Bunse, W. ; Bunse-Gerstner, A.: *Numerische lineare Algebra.* Teubner, 1985 3.3.1

[13] Francis, J.G.F.: The QR transformation a unitary analogue to the LR transformation Part 1. In: *The Computer Journal* 4 (1961), Nr. 3, S. 265. – ISSN 0010–4620 3.3.1

[14] Wilkinson, JH: Convergence of the LR, QR, and related algorithms. In: *The Computer Journal* 8 (1965), Nr. 1, S. 77. – ISSN 0010–4620 3.3.1, 5

[15] Wilkinson, J. H.: Global convergene of tridiagonal QR algorithm with origin shifts. In: *Linear Algebra and Its Applications* 1 (1968), Nr. 3, S. 409–420 5

[16] Reinsch, C. ; Bauer, FL: RationalQR transformation with Newton shift for symmetric tridiagonal matrices. In: *Numerische Mathematik* 11 (1968), Nr. 3, S. 264–272 5

[17] Horn, R.A. ; Johnson, C.R.: *Matrix analysis.* Cambridge University Press, 2005. – ISBN 0521386322 3.3.2

[18] Hwang, S.G.: Cauchy's Interlace Theorem for Eigenvalues of Hermitian Matrices. In: *American Mathematical Monthly* (2004), S. 157–159. – ISSN 0002–9890 3.3.2

[19] Yap, C.K.: *Fundamental problems of algorithmic algebra.* Bd. 54. Oxford University Press, 2000 3.3.2

[20] Stoer, J.: *Einführung in die Numerische Mathematik I. Heidelberger Taschenbücher Band 105.* 1972 3.3.2

[21] Henrici, P.: Bounds for eigenvalues of certain tridiagonal matrices. In: *Journal of the Society for Industrial and Applied Mathematics* 11 (1963), Nr. 2, S. 281–290. – ISSN 0368–4245 3.3.2

[22] Wielandt, H.: *Beiträge zur mathematischen Behandlung komplexer Eigenwertprobleme.* Aerodynamische Versuchsanstalt, 1944 12

[23] Schwarz, H.R. ; Köckler, N.: *Numerische mathematik.* Vieweg+ Teubner, 2006 12

[24] Demmel, J.W.: Applied numerical linear algebra. (1997) 5, 12

[25] Dhillon, I.S.: Current inverse iteration software can fail. In: *BIT Numerical Mathematics* 38 (1998), Nr. 4, S. 685–704 5

[26] Cuppen, JJM: A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem. In: *Numer. Math* 36 (1981), S. 177–195 3.3.3

[27] Dhillon, I.S. ; Parlett, B.N.: Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. In: *Linear Algebra and its Applications* 387 (2004), S. 1–28 3.3.4, 14

[28] Dhillon, I.S.: *A New O(n^2) Algorithm for the Symmetric Tridiagonal Eigenvalue Eigenvector Problem*, University of California, Berkeley, Diss., 1997 3.3.4, 3.3.1, 3.3.4

[29] Dhillon, I.S. ; Parlett, B.N.: Orthogonal eigenvectors and relative gaps. In: *SIAM Journal on Matrix Analysis and Applications* 25 (2004), Nr. 3, S. 858–899 3.3.4, 3.3.4, 3.3.4

[30] Fernando, K.V. ; Parlett, B.N.: Accurate singular values and differential qd algorithms. In: *Numerische Mathematik* 67 (1994), Nr. 2, S. 191–229 3.3.4

[31] Xu, W. ; Qiao, S.: A twisted factorization method for symmetric SVD of a complex symmetric tridiagonal matrix. In: *Numerical Linear Algebra with Applications* 16 (2009), Nr. 10, S. 801–815 3.3.4

[32] Dhillon, I.S. ; Parlett, B.N. ; Vömel, C.: The design and implementation of the MRRR algorithm. In: *ACM Transactions on Mathematical Software (TOMS)* 32 (2006), Nr. 4, S. 533–560 14, 4.2.1.2

[33] *IBM Research Blue Gene Project Page*. Website, Jun 2011. – Available online at `http://www.research.ibm.com/bluegene/index.html` 4

[34] *IBM - Homepage*. Website, Jun 2011. – Available online at `http://www.ibm.com/us/en/` 4

[35] Allen, F. ; Almasi, G. et al.: Blue Gene: A vision for protein science using a petaflop supercomputer. 4.1

[36] *November 2004 | TOP500 Supercomputing Sites*. Website, Jun 2011. – Available online at `http://www.top500.org/lists/2004/11` 4.1

[37] *FZJ-JSC IBM Blue Gene/P - JUGENE home page*. Website, Jun 2011. – Available online at `http://www2.fz-juelich.de/jsc/jugene` 4.1

[38] *Forschungszentrum Jülich - Homepage*. Website, Jun 2011. – Available online at `http://www.fz-juelich.de/portal/DE/Home/home_node.html` 4.1

[39] *HPCWire: Lawrence Livermore Prepares for 20 Petaflop Blue Gene/Q*. Website, Jun 2011. – Available online at `http://www.hpcwire.com/hpcwire/2009-02-03/lawrence_livermore_prepares_for_20_petaflop_blue_gene_q.html` 4.1, 4.1.2

[40] Arnold, L. ; Frings, W. ; Gutheil, I. ; Homberg, W. ; Knobloch, M. ; Krieg, S. ; Mohr, B. ; Schiller, A. ; Stephan, M. ; Sutmann, G.: BG/Q EURO Prototype DD1 at Jülich: Early Access Report / JSC & IBM Confidential. 2011. – Forschungsbericht 4.1, 4.1.2

[41] *November 2007 | TOP500 Supercomputing Sites*. Website, Jun 2011. – Available online at `http://www.top500.org/lists/2007/11` 4.1.1

[42] Sosa, C.P. ; Knudson, B.: IBM system Blue Gene solution: Blue Gene/P application development. In: *IBM Redpaper Publication* (2009) 4.1.1, B

[43] Dongarra, J.J. ; Luszczek, P. ; Petitet, A.: The LINPACK Benchmark: past, present and future. In: *Concurrency and Computation Practice and Experience* 15 (2003), Nr. 9, S. 803–820 4.1.1

[44] Dongarra, J.: The LINPACK benchmark: An explanation. In: *Supercomputing* Springer, 1988, S. 456–474 4.1.1

[45] *Lawrence Livermore National Laboratory (LLNL)*. Website, Jun 2011. – Available online at `https://www.llnl.gov/` 4.1.2

[46] *The Green500 List*. Website, Jun 2011. – Available online at `http://www.green500.org/lists/2011/06/top/list.php` 4.1.2

[47] *IBM Research | Watson Research Center*. Website, Jun 2011. – Available online at `http://www.watson.ibm.com/index.shtml` 4.1.2

[48] Budnik, T. ; Knudson, B. ; Megerian, M. ; Miller, S. ; Mundy, M. ; Stockdell, W.: Blue Gene/Q Resource Management Architecture. In: *IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10), co-located with IEEE/ACM Supercomputing* Bd. 2010, 2010 4.1.2

[49] Blackford, LS ; Cleary, A. ; Choi, J. ; d'Azevedo, E. ; Demmel, J. ; Dhillon, I. ; Dongarra, J. ; Hammarling, S. ; Henry, G. ; Petitet, A. et al.: *ScaLAPACK users' guide*. Society for Industrial Mathematics, 1997. – ISBN 0898713978 4.2, 4.2.1, 4.2.1.2, 4.2.1.3, 5.3.1

[50] *elemental - distributed-memory dense linear algebra - Google Project Hosting*. Website, Jun 2011. – Available online at `http://code.google.com/p/elemental/` 4.2

[51] Anderson, E. ; Bai, Z. ; Bischof, C.: *LAPACK Users' guide*. Society for Industrial Mathematics, 1999. – ISBN 0898714478 4.2.1.1

[52] Lawson, C. L. ; Hanson, R. J. ; Kincaid, D. R. ; Krogh, F. T.: Basic Linear Algebra Subprograms for Fortran Usage. In: *ACM Trans. Math. Softw.* 5 (1979), September, 308–323. `http://dx.doi.org/http://doi.acm.org/10.1145/355841.355847`. – DOI http://doi.acm.org/10.1145/355841.355847. – ISSN 0098–3500 4.2.1.1

[53] Agarwal, R.C. ; Gustavson, F.G. ; McComb, J. ; Schmidt, S.: Engineering and Scientific Subroutine Library Release 3 for IBM ES/3090 vector multiprocessors [Technical note]. In: *IBM Systems Journal* 28 (1989), Nr. 2, S. 345–350 4.2.1.1

[54] Forum, Message Passing I.: *MPI: A Message-Passing Interface Standard, Version 2.2*. High-Performance Computing Center Stuttgart (HLRS), 2009 4.2.1.1

[55] Al Geist, A.B. ; Dongarra, J. ; Jiang, W. ; Manchek, R. ; Sunderam, V.S.: PVM: Parallel Virtual Machine: a users' guide and tutorial for network parallel computing. (1994) 4.2.1.1

[56] Dongarra, J.J. ; Whaley, R.C.: A User's Guide to the BLACS v1. 1. (1997). – Available online at `http://www.netlib.org/blacs/lawn94.ps` 4.2.1.1

[57] Choi, J. ; Dongarra, JJ ; Ostrouchov, S. ; Petitet, A. ; Walker, D. ; Whaley, RC: LAPACK working note 100: a proposal for a set of parallel basic linear algebra subprograms. In: *Computer Science Dept. Technical Report CS-95-292, University of Tennessee, Knoxville* (1995) 4.2.1

[58] B. Hendrickson, E. J. ; Smith, C.: A parallel eigensolver for dense symmetric matrices / Sandia National Labs, Albuquerque, NM. 1996. – Forschungsbericht 4.2.1.2, 5.3

[59] Esser, R. ; Knecht, R.: Intel Paragon XP/S-Architecture and Software Enviroment. In: *Anwendungen, Architekturen, Trends, Seminar* Springer-Verlag, 1993, S. 121–141 4.2.1.2

[60] Stanley, K.S.: *Execution time of symmetric eigensolvers*, University of California, Berkeley, Diss., 1997 4.2.1.2

[61] Choi, J. ; Demmel, J. ; Dhillon, I. ; Dongarra, J. ; Ostrouchov, S. ; Petitet, A. ; Stanley, K. ; Walker, D. ; Whaley, R.C.: ScaLAPACK: a portable linear algebra library for distributed memory computers–design issues and performance. In: *Computer Physics Communications* 97 (1996), Nr. 1-2, S. 1–15 4.2.1.3

[62] *pmrrr - Distributed-memory symmetric tridiagonal eigensolver - Google Project Hosting*. Website, Jun 2011. – Available online at `http://code.google.com/p/pmrrr/` 4.2.2.2

[63] Hendrickson, B. ; Jessup, E. ; Smith, C.: Toward an efficient parallel eigensolver for dense symmetric matrices. In: *SIAM Journal on Scientific Computing* 20 (1999), Nr. 3, S. 1132–1154 4.2.2.3

[64] Strazdins, P.: Optimal load balancing techniques for block-cyclic decompositions for matrix factorization. In: *Proceedings of PDCN* 98 (1998) 4.2.2.3

[65] Poulson, J. ; Marker, B. ; Hammond, J.R. ; Romero, N.A. ; Geijn, R. van d.: Elemental: A new framework for distributed memory dense matrix computations. In: *ACM Transactions on Mathematical Software. submitted* (2010). – Available online at `http://users.ices.utexas.edu/~poulson/publications/Elemental.pdf` 4.2.2.3, 4.2.2.3

[66] Geimer, M. ; Wolf, F. ; Wylie, B.J.N. ; Ábrahám, E. ; Becker, D. ; Mohr, B.: The Scalasca performance toolset architecture. In: *Concurrency*

and Computation: Practice and Experience 22 (2010), Nr. 6, S. 702–719 5.3, 6, 6.5

[67] Rauber, T. ; Rünger, G.: *Parallel Programming: for Multicore and Cluster Systems*. Springer-Verlag New York Inc, 2010 6.3.1

[68] Amdahl, Gene M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*, ACM, 1967 (AFIPS '67 (Spring)), S. 483–485 6.3.1

[69] Gustafson, J.L.: Reevaluating Amdahl's law. In: *Communications of the ACM* 31 (1988), Nr. 5, S. 532–533 6.3.1

[70] Dagum, L. ; Menon, R.: OpenMP: an industry standard API for shared-memory programming. In: *Computational Science & Engineering, IEEE* 5 (1998), Nr. 1, S. 46–55 6.5

[71] Geimer, M. ; Wolf, F. ; Wylie, B.J.N. ; Ábrahám, E. ; Becker, D. ; Mohr, B.: *Cube 3.3 – User Guide*. Jülich Supercomputing Centre, 3 2011. – Available online at `http://www2.fz-juelich.de/jsc/datapool/scalasca/CubeGuide.pdf` 6.5

[72] Geimer, M. ; Wolf, F. ; Wylie, B.J.N. ; Ábrahám, E. ; Becker, D. ; Mohr, B.: *Scalasca 1.3 – User Guide*. Jülich Supercomputing Centre, 3 2011. – Available online at `http://www2.fz-juelich.de/jsc/datapool/scalasca/UserGuide.pdf` 6.5

[73] Auckenthaler, T. ; Blum, V. ; Bungartz, H.-J. ; Huckle, T. ; Johanni, R. ; Krämer, L. ; Lang, B. ; Lederer, H. ; Willems, P.R.: Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. In: *Parallel Computing* In Press, Corrected Proof (2011), -. `http://dx.doi.org/DOI:10.1016/j.parco.2011.05.002`. – DOI DOI: 10.1016/j.parco.2011.05.002. – ISSN 0167–8191 7.2

[74] Auckenthaler, T. ; Bungartz, H.-J. ; Huckle, T. ; Krämer, L. ; Lang, B. ; Willems, P.: Developing algorithms and software for the parallel solution of the symmetric eigenvalue problem. In: *Journal of Computational Science* In Press, Accepted Manuscript (2011), -. `http://dx.doi.org/DOI:10.1016/j.jocs.2011.05.002`. – DOI DOI: 10.1016/j.jocs.2011.05.002. – ISSN 1877–7503 7.2

Anhang A

Compiler und Software

Compiler:

- IBM XL C/C++ Advanced Edition for Blue Gene/P, V9.0
- IBM XL Fortran Advanced Edition for Blue Gene/P, V11.1
- IBM XL C/C++ for Blue Gene/Q, V12.0
- IBM XL Fortran for Blue Gene/Q, V14.1

Software:

- Elemental, V0.66
- ScaLAPACK, V1.8
- LAPACK, V3.3
- BLACS, V1.1
- ESSL, V4.4
- Scalasca, V1.3

Anhang B

Quellcode

Listing B.1: dsyemagsc.cpp

```
/*  (ID-2W(W**T))D(ID-2W(W**T)) = A = D-2W(W**T)D-2DW(W**T)+4W(W**T)DW(W**T) is computed with
     a normed random vector called work and the vector diag.
    A is a full symmetric matrix of size n, whose eigenvalues are elements of diag and its
        eigenvectors are ID-2W(W**T).
    The Matrix A ist blockcyclic distributed with blocksize 1 (ELEMENTAL).
    The array rv1 ist just a temporary working array. */
void dsyemagsc(int n,double *diag,double *work,DistMatrix<double,MC,MR>& A,double *rv1){
  const Grid& g = A.Grid();
  int i,j, iglob, jglob, iloc, jloc, ii, jj;
  int mb, nb, myrow, mycol, nprow, npcol, np, nq;
  double r;
  nb=1; //blocksizes (in ELEMENTAL always 1)
  mb=1; //
  myrow=g.MRRank(); // my row rank
  mycol=g.MCRank(); // my col rank
  npcol=g.Width();// number of procs in col
  nprow=g.Height();//number of procs in row
  np=A.LocalMatrix().Height(); //number of local rows
  nq=A.LocalMatrix().Width(); //number of local columns
  for(i=0;i<n;++i)
    rv1[i]=work[i]*work[i];
  r=imports::blas::Dot(n,diag,1,rv1,1);
  for (j=1;j<=nq;j=j+nb){
    for(jj=1;jj<=min(nb,nq-j+1);++jj){
      jloc=j-1+jj;
      jglob=(j-1)*npcol + myrow*nb + jj;
      for(i=1;i<=np;i=i+mb){
  for(ii=1;ii<=min(mb,np-i+1);++ii){
    iloc=i-1+ii;
    iglob=(i-1)*nprow + mycol*mb +ii;
    if(iglob == jglob){
      A.LocalMatrix().Set(iloc-1,jloc-1,diag[iglob-1] - 4* (diag[iglob-1]-r) * rv1[iglob-1]);
    }
    else{
      A.LocalMatrix().Set(iloc-1,jloc-1, -2*( diag[iglob-1] + diag[jglob-1] -2*r ) * work[
          iglob-1] * work[jglob-1]);
    }
  }
      }
    }
  }
}
```

Abbildungsverzeichnis