Berechnung der Singulärwertzerlegung von Matrizen auf Parallelrechnern

Wissenschaftliche Arbeit im Fach Mathematik
vorgelegt von

Peter Dörr

19. Oktober 1999

Peter Dörr

Berechnung der Singulärwertzerlegung von Matrizen auf Parallelrechnern

Wissenschaftliche Arbeit im Fach Mathematik

Hiermit versichere ich an Eides statt, daß ich diese Arbeit noch keinem Prüfungsamt vorgelegt habe, daß nur die angegebenen Quellen benutzt wurden, und daß fremdes Gedankengut nach bestem Wissen als solches kenntlich gemacht wurde.

Saarbrücken, im November 1995

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Seit den siebziger Jahren konnte die Computerindustrie die Leistungsfähigkeit ihrer Produkte etwa alle vier Jahre verdoppeln. Dabei bediente man sich zunehmend spezialisierterer Methoden, um weitere Leistungssteigerungen zu erreichen. Diese Spezialisierung orientierte sich an den am häufigsten auftretenden Operationen, deren Beschleunigung die Gesamtleistung am wirkungsvollsten anhob. Wenn ein Programm sich ausschließlich beschleunigter Operationen bedient, erreicht ein moderner Computer seine Maximalleistung. Diese liegt bei heutigen Supercomputern im GigaFlop-Bereich (mehr als eine Milliarde Rechenoperationen pro Sekunde). Besteht ein Programm hingegen vornehmlich aus weniger schnellen Anweisungen, dann kann die Gesamtleistung auf wenige Prozent des Machbaren zusammenbrechen. Dies gilt besonders für Parallelrechner. So werden mehrere miteinander verbundene Computer genannt, die gemeinsam ein Problem bearbeiten können. Alle modernen Rechenanlagen erfordern speziell an die jeweilige Architektur angepaßte Algorithmen, um ihre volle Leistungsfähigkeit zu entfalten. Wer heute ein effizientes Programm schreiben möchte, ein Programm, das die Möglichkeiten eines Rechners optimal ausnutzt, muß mit einem hohen Optimierungsaufwand rechnen. Dies wird zum Teil zwar durch komfortable Entwicklungswerkzeuge erleichtert, doch das letzte Quentchen Leistung läßt sich meist nur mit intimen Maschinenkenntnissen und Programmierung auf Prozessorebene aus einem Rechner herausholen. Dieser Problematik begegnen die Hersteller, indem sie optimierte Routinen für wichtige mathematische Algorithmen mit den Rechnern ausliefern. Im Bereich Lineare Algebra haben sich die BLAS-Routinen [Dongarra u.a.] als Standard für elementare Vektor- und Matrixoperationen etabliert. Auf dieser Sammlung bauen viele Bibliotheken für komplexere Anwendungen auf, die bis auf wenige kommerzielle Ausnahmen über das Internet frei zugänglich sind. Eine solche Bibliothek ist LAPACK [Dongarra u.a.]. Ähnlich den BLAS Routinen scheint sich LAPACK zum De-facto-Standard für Eigenwertprobleme und (vollbesetzte) lineare Gleichungssysteme zu entwickeln. Mathematische Bibliotheken sind vorwiegend in der Programmiersprache FORTRAN geschrieben. Sie sind darauf ausgerichtet, auf Ein-Prozessor-Computern moderner Architektur gute Effizienz zu liefern. Auf Parallelrechner sind sie nur bedingt übertragbar. Die vorliegende Arbeit hat eine der komplexesten Anwendungen, die in der LAPACK-Sammlung enthalten ist, zum Gegenstand: Die Singulärwertzerlegung von Matrizen. Sie bot sich als Studienobjekt zur Parallelrechnerprogrammierung aus zwei Gründen an:

1) Die entsprechende LAPACK-Routine besteht aus zwei Teilen, von denen nur der erstere durch Einbinden parallelisierter BLAS-Funktionen gute Effizienz auf Parallelrechnern erreicht. Der zweite Teil wird zum Beispiel in der von Silicon-Graphics für den Parallelcomputer Challenge bereitgestellten LAPACK Bibliothek nur auf einem Prozessor ausgeführt. Da bei großen Matrizen der Aufwand für den zweiten Teil überwiegt, läuft das parallelisierte Programm nur wenig schneller als das Ein-Prozessor-Programm ab.

2) Die Singulärwertzerlegung einer Matrix liefert vielfältige Informationen über deren Eigenschaften. Der Aufwand ihrer Berechnung ist aber so hoch, daß die Kapazitäten gewöhnlicher Workstations bei großen Matrizen nicht mehr ausreichen.

Dabei spielt die Effizienz - letztlich die Ablaufgeschwindigkeit - eines Programmes eine nachgeordnete Rolle. Die Brauchbarkeit einer Routine wird in erster Linie durch die Güte der berechneten Ergebnisse bestimmt, denn Funktionsbibliotheken wie LAPACK sind dazu gedacht, in größeren Projekten Teilaufgaben zu übernehmen. Ein Anwender muß sich also auf die Ergebnisse verlassen können. Unter einem „guten" Programm verstehen wir eines, das effizient ist und Ergebnisse optimaler Genauigkeit liefert. Diesem Ziel habe ich mich mit einer Funktion anzunähern versucht, die die LAPACK-Routine SGESVD zur Berechnung der Singulärwertzerlegung ersetzt. Dabei habe ich regen Gebrauch anderer LAPACK-Funktionen gemacht, die im Gegensatz zu SGESVD auch auf Parallelrechnern gute Effizienz liefern. Auf den folgenden beiden Parallelrechnern wurde der Algorithmus implementiert:

- Silicon Graphics Challenge (gemeinsamer Speicher, 4 Prozessoren)
- Intel IPSC860 (verteilter Speicher, 32 Prozessoren)

Eine alltagstaugliche Version ist für die SGI Challenge entstanden, während die Implementierung für den IPSC860 experimentellen Charakter besitzt und nur eine eingeschränkte Funktionalität bietet.

Kommen wir nun zu der Frage, was die Singulärwertzerlegung einer Matrix eigentlich ist. Ohne Beschränkung der Allgemeinheit sei A eine $m \times n$-Matrix ($m \geq n$) mit $rang(A) = r$. Die Singulärwertzerlegung von A definieren wir als

$$A = USV^T,$$

mit $U^TU = V^TV = I_n$ und $S = diag(\sigma_1, \dots, \sigma_n)$, $\sigma_i > 0$ für $1 \leq i \leq r$, $\sigma_i = 0$ für $i \geq r+1$. Die ersten r Spaltenvektoren der orthogonalen Matrizen U und V definieren die orthonormierten Eigenvektoren zu den r von Null verschiedenen Eigenwerten von

AA^T bzw. A^TA. Die Singulärwerte von A sind die Diagonalelemente von S, die gerade die positiven Wurzeln der Eigenwerte von A^TA sind. Die Singulärwertzerlegung einer Matrix liefert uns wichtige Informationen über die Stuktur derselben:

i) Rang-Eigenschaft:

$$rang(A) = r,\ \mathcal{N}(A) = span(v_{r+1}, \dots, v_n)\ und$$

$$\mathcal{R}(A) = span(u_1, \dots, u_r),\ \ wobei$$
$$U = [u_1, \dots, u_m]\ und\ V = [v_1, \dots, v_n].$$

ii) Dyadische Zerlegung:
$$A = \sum_{i=1}^{r} u_i \sigma_i v_i^T.$$

iii) Normen:
$$\|A\|_F^2 = \sigma_1^2 + \ldots + \sigma_r^2\ \ und\ \ \|A\|_2^2 = \sigma_1.$$

iv) Matrixapproximation:

$$Sei\ \ A_k := \sum_{i=1}^{k} u_i \sigma_i v_i^T\ \ mit\ \ k < r.\ \ Dann\ gilt$$

$$\min_{rang(B)=k} \|A - B\|_F^2 = \|A - A_k\|_F^2 = \sigma_{k+1}.$$

Die letztgenannte Eigenschaft, die anzeigt, daß A_k die beste Rang-k-Approximation (im least squares Sinne) an die Matrix A ist, bildet die Grundlage zum Einsatz der Singulärwertzerlegung zur Datenkomprimierung. Sind die Singulärwerte in absteigender Reihenfolge sortiert, $\sigma_i \geq \sigma_{i+1}$, dann können wir die Matrix mit Hilfe der dyadischen Zerlegung als eine Summe von Rang-1-Matrizen aufschreiben, deren Summanden immer weniger zur Summe beitragen. Lassen wir die letzten Summanden fort, dann bleiben also die wesentlichen Informationen erhalten.

Die Rang-Eigenschaft ist ein wichtiger Aspekt der Singulärwertzerlegung. Durch sie erhalten wir einen Zugang zu Lösungsmethoden für schlecht gestellte Probleme. Sei $A : X \to Y$ mit topologischen Räumen X und Y. Das Problem (A, X, Y) heißt gut gestellt, wenn

i) $Af = g$ für jedes $g \in Y$ eine Lösung besitzt

ii) diese Lösung eindeutig ist

iii) die Lösung stetig von den Daten abhängt.

Ist eine der Bedingungen nicht erfüllt, so nennen wir das Problem schlecht gestellt. Auf ein solches trifft man zum Beispiel bei der Computertomographie [vgl. Louis 1989]. Dort ist A ein linearer Operator, dessen Diskretisierung auf eine Matrix führt. In den Komponenten des Vektors g stehen die Meßdaten, in denen des Lösungsvektors f die Bilddaten, die auf dem Computermonitor visualisiert werden können.
Bei schlecht gestellten Problemen können wir keine Inverse zu A angeben, die uns die Lösung $f = A^{-1}g$ liefert. Statt dessen definiert man die verallgemeinerte Inverse $A^\dagger$, die uns die verallgemeinerte Lösung $f^\dagger = A^\dagger g$ liefert. Sie ist eindeutig bestimmt durch die Forderung, daß $f^\dagger$ das Funktional $J(f) = \|Af - g\|_2$ minimiert, und unter allen f, die J minimieren, die kleinste Norm besitzt. Wenn A eine $m \times n$-Matrix ist, dann erhalten wir die verallgemeinerte Inverse als

$$A^\dagger = \sum_{i=1}^{r} \sigma_i^{-1} v_i u_i^T,$$

und die verallgemeinerte Lösung

$$f^\dagger = A^\dagger g = \sum_{i=1}^{r} \sigma_i^{-1} < u_i, g > v_i.$$

Als zusätzliche Schwierigkeit wollen wir zulassen, daß die rechte Seite g (Meßdaten) mit Fehlern in der Größenordnung ϵ behaftet sind: $\|g^\epsilon - g\|_2 \leq \epsilon$. Eine Lösung von $Af = g^\epsilon$ sollte die verallgemeinerte Lösung $f^\dagger$ möglichst gut approximieren. Das Problem besteht nun darin, daß g und damit $f^\dagger$ nicht bekannt sind. Wie soll man dann $f^\dagger$ approximieren? Betrachten wir die Möglichkeit, $A^\dagger$ durch

$$A_k^\dagger = \sum_{i=1}^{k} \sigma_i^{-1} v_i u_i^T, \quad k \leq r$$

zu ersetzen. Dann gilt

$$\|A_k^\dagger g^\epsilon - f^\dagger\|_2 \leq \|A_k^\dagger (g^\epsilon - g)\|_2 + \|(A_k^\dagger - A^\dagger) g\|_2.$$

Den ersten Teil des Fehlers bezeichnen wir als Datenfehler. Dieser ist gleich Null, wenn keine Störungen vorliegen: $\epsilon = 0$. Im allgemeinen ist er umso größer, je größer wir k wählen. Daß das Problem schlecht gestellt ist, bewirkt, daß die Singulärwerte der Matrix schnell fallen. Betrachten wir

$$A^\dagger (g^\epsilon - g) = \sum_{i=1}^{r} \sigma_i^{-1} < u_i, g^\epsilon - g > v_i.$$

Die Störungen in den Anteilen der kleinen Singulärwerte werden durch die Multiplikation mit dem entsprechend großen Kehrwert des Singulärwertes verstärkt. Im Vergleich dazu wirken sich die Störungen in den Anteilen zu großen Singulärwerten wenig aus. Günstigerweise entsprechen die Anteile zu kleinen Singulärwerten den hochfrequenten

Anteilen (Details) der Lösung. Die wesentlichen Informationen bleiben also erhalten, wenn wir die Anteile zu den kleinen Singulärwerten verwerfen.

Für den zweiten Teil des Fehlers, den Regularisierungsfehler, gilt:

$$\|(A_k^\dagger - A^\dagger)g\|_2 = \| \sum_{i=k+1}^{r} \sigma_i^{-1} < u_i, g > v_i\|_2$$

Hier sollte k möglichst groß gewählt werden, um den Fehler gering zu halten. Mit der Wahl von k haben wir die Möglichkeit erhalten, zwischen Daten- und Regularisierungsfehler abzuwägen. Ohne zusätzliche Kenntnisse über den Fehler oder f bzw. $f^\dagger$ haben wir keine Möglichkeit, den besten Parameter k zu bestimmen. Beim Beispiel Computertomographie kann unser Auge solche Zusatzinformationen einbringen. Hier könnte anhand der visuell ermittelten Bildqualität der optimale Parameter k „eingestellt" werden.

Speziell zur Lösung schlecht gestellter Probleme wird in den folgenden Kapiteln ein Algorithmus und ein Programm für Parallelrechner entwickelt. Das Problem lautet:

Gesucht sind die ersten k Singulärtripel (u_i, σ_i, v_i) einer vollbesetzten, unstrukturierten und „großen" Matrix $A \in \mathbb{R}^{m \times n}$.

Im zweiten Kapitel werden mathematische Voraussetzungen und Algorithmen zur Lösung dieses Problems diskutiert. Das Problem der Berechnung der Singulärwertzerlegung wird auf das symmetrische Eigenwertproblem zurückgeführt. Damit lassen sich aus den reichhaltigen vorhandenen Algorithmen zur Eigenwertberechnung solche auswählen, die sich gut zur Berechnung der Singulärwertzerlegung eignen und die insbesondere auf Parallelrechnern gute Effizienz versprechen. Diesem Thema widmet sich das dritte Kapitel, in dem die heute gängigen Computerarchitekturen sowie zwei Parallelrechner, vorgestellt werden. Anhand von Laufzeitmessungen werden die für die Realisierung numerischer Algorithmen wichtigen technischen Voraussetzungen untersucht und daraus Forderungen abgeleitet, die ein Algorithmus erfüllen muß, wenn er effizient ablaufen soll. Im letzten Kapitel werden die vorherigen Erkenntnisse in Programmen zur Berechnung der Singulärwertzerlegung verdichtet.

Kapitel 2

Mathematische Grundlagen und Algorithmen

2.1 Die Singulärwertzerlegung von Matrizen

2.1.1 Definition und Eigenschaften

In diesem Abschnitt werden die mathematischen Hilfsmittel aus der linearen Algebra bereitgestellt, die zum Verständnis der Algorithmen zur Berechnung der Singulärwertzerlegung benötigt werden. Bis auf die Herleitung der Singulärwertzerlegung, die Matrizen mit komplexwertigen Einträgen voraussetzt, beschränkt sich die Darstellung auf den Körper der reellen Zahlen. Die Singulärwertzerlegung wird auf das symmetrische Eigenwertproblem zurückgeführt, für dessen Lösung es eine Reihe verschiedener Verfahren gibt. Das gängigste dieser Verfahren - der QR-Algorithmus - wird ausführlich diskutiert, um die typischen Methoden bei der Realisierung von Eigenwertalgorithmen zu veranschaulichen. Anschließend werden alternative Verfahren vorgestellt, die sich auch für die Implementierung auf Parallelrechnern eignen.

Definition 2.1.1 *Sei* $B \in \mathbb{C}^{n \times n}$.

i) Ein $\lambda \in \mathbb{C}$ *heißt Eigenwert von* B, *wenn es ein* $v \in \mathbb{C}^n$ *mit* $v \neq 0$ *gibt, so daß gilt*

$$Bv = \lambda v.$$

v *heißt Eigenvektor zum Eigenwert* λ.

ii) Das Polynom

$$p_B(\mu) := det(B - \mu I_n)$$

wird als das charakteristische Polynom der Matrix B *bezeichnet,* I_n *steht für die Einheitsmatrix in* $\mathbb{C}^{n \times n}$.

iii) *Die Menge* $Eig(B,\lambda) := \{v \in \mathbb{C}^n : Bv = \lambda v\}$ *heißt Eigenraum von* B *zum Eigenwert* λ.

iv) *Seien* $\{E, (\ ,\ )\}$ *und* $\{F, <\ ,\ >\}$ *endlich dimensionale* $\mathbb{C}$ *Vektorräume versehen mit den Skalarprodukten* $(\ ,\ )$ *und* $<\ ,\ >$. $f : E \longrightarrow F$ *sei eine lineare Abbildung. Wir definieren die adjungierte Abbildung* f^* *zu* f *durch:*

$$< f(x), y >= (x, f^*(y)), \quad \textit{für alle}\, x \in E,\ y \in F$$

Eine lineare Abbildung $f : E \longrightarrow E$ *heißt selbstadjungiert, wenn für alle* $x, y \in E$ *gilt*

$$< f(x), y >= (x, f(y)).$$

Anmerkung 2.1.2 Sei $B \in \mathbb{C}^{n\times n}$. Dann gelten die folgenden Aussagen:

i) Die Nullstellen des charakteristischen Polynoms $p_B(\mu)$ sind die Eigenwerte von B.

ii) Ist E ein endlich dimensionaler $\mathbb{C}$-Vektorraum mit Skalarprodukt $<\ ,\ >$ und f eine selbstadjungierte lineare Abbildung auf E, dann besitzt E eine Orthonormalbasis aus Eigenvektoren von f. Sind v_i, $1 \le i \le n$, die normierten Eigenvektoren von f, das heißt $\|v_i\| = (< v_i, v_i >)^{1/2} = 1$, dann gilt

$$< v_i, v_j >= \delta_{i,j}, \quad 1 \le i, j \le n, \textit{ wobei}$$

$$\delta_{i,j} := \begin{cases} 1 & : \quad i = j \\ 0 & : \quad i \ne j \end{cases}$$

Sind $\lambda_1, ..., \lambda_r$ die paarweise verschiedenen Eigenwerte von f, so ist

$$\mathbb{C}^n = Eig(f, \lambda_1) \oplus ... \oplus Eig(f, \lambda_r),$$

insbesondere gilt für $\lambda_i \ne \lambda_j \quad Eig(f, \lambda_i) \perp Eig(f, \lambda_j)$.

iii) Ist $E = \mathbb{C}^n$ und B die zu f gehörige Matrix bezüglich einer Orthonormalbasis des $\mathbb{C}^n$, so ist $B^* = \bar{B}^T$ die zu B konjugierte Matrix.

Satz 2.1.3 (Existenz der Singulärwertzerlegung) *Sei* $A \in \mathbb{C}^{m\times n}$ *, dann existieren die Matrizen* $U \in \mathbb{C}^{m\times m}$ *,* $S \in \mathbb{C}^{m\times n}$ *und* $V \in \mathbb{C}^{n\times n}$*, so daß*

$$A = USV^* \tag{2.1}$$

mit $U^*U = I_m$ *,* $V^*V = I_n$ *und* $S = diag(\sigma_1, \ldots, \sigma_n)$.

Beweis : [nach Louis 1989, S.147-148] Wir betrachten die Matrix A^*A. Es gilt

$$< A^*Ax, y >=< Ax, Ay >=< x, A^*Ay >$$

für alle $x, y \in \mathbb{C}^n$. Also ist A^*A selbstadjungiert, und es existiert demnach eine Orthonormalbasis von $\mathbb{C}^n$ bestehend aus Eigenvektoren von A^*A. Weiterhin ergibt sich für alle x $\in \mathbb{C}^n$

$$x^*A^*Ax = \|Ax\|_2 \geq 0$$

und für alle Eigenpaare (λ, v) von A^*A

$$\lambda < v, v >=< v, \lambda v >=< v, A^*Av >=< A^*Av, v >=< \lambda v, v >= \bar{\lambda} < v, v > \;.$$

A^*A ist somit positiv semidefinit und besitzt nur nichtnegative, reelle Eigenwerte. Es seien (λ_i, v_i), $1 \leq i \leq n$, die Eigenpaare der Matrix A^*A mit normierten Eigenvektoren v_i. Die Eigenwerte seien so numeriert, daß $\lambda_1 \geq ... \geq \lambda_r > \lambda_{r+1} = ... = \lambda_n = 0$. Definieren wir $\sigma_i = (\lambda_i(A^*A))^{1/2}$, $1 \leq i \leq n$, als die Wurzeln der Eigenwerte von A^*A und

$$u_i = \sigma_i^{-1} Av_i, \;\; 1 \leq i \leq r,$$

dann sind $(\lambda_i, u_i), 1 \leq i \leq r$ normierte Eigenpaare von AA^* zu den von Null verschiedenen Eigenwerten λ_1 , ... , λ_r, denn für $\sigma_i \neq 0$ gilt

$$\begin{aligned} AA^*u_i &= \sigma_i^{-1}(AA^*)Av_i \\ &= \frac{\lambda_i}{\sigma_i} Av_i \\ &= \lambda_i u_i \end{aligned}$$

und

$$\begin{aligned} u_i^* u_j &= \frac{1}{\sigma_i \sigma_j}(Av_i)^*(Av_j) \\ &= \frac{1}{\sigma_i \sigma_j} v_i^* A^* A v_j \\ &= \frac{\sigma_j^2}{\sigma_i \sigma_j} v_i^* v_j \\ &= \delta_{i,j}. \end{aligned}$$

Die Eigenvektoren zu dem $(n-r)$-fachen Eigenwert 0 von A^*A sind nicht eindeutig bestimmt. Die Vektorfamilie $(v_i)_{1 \leq i \leq r}$ läßt sich allerdings durch Hinzunahme von $n-r$ orthonormalen Lösungen von $A^*Ax = 0$ zu einer Orthonormalbasis des $\mathbb{C}^n$ erweitern. Analog können wir eine Orthonormalbasis des $\mathbb{C}^m$ durch Erweitern der Vektorfamilie $(u_i)_{1 \leq i \leq r}$ um $m-r$ Vektoren erhalten. Betrachten wir ein beliebiges Element $x \in \mathbb{C}^n$, dann gilt

$$Ax = A \sum_{i=1}^{n} < x, v_i > v_i = \sum_{i=1}^{n} (\sigma_i < x, v_i >) u_i = \sum_{i=1}^{r} (\sigma_i < x, v_i >) u_i. \tag{2.2}$$

Wir erhalten

$$\mathcal{R}(A) = span(u_1, ..., u_r), \quad \mathcal{N}(A) = span(v_{r+1}, ..., v_n)$$

und analog

$$\mathcal{R}(A^*) = span(v_1, ..., v_r), \quad \mathcal{N}(A^*) = span(u_{r+1}, ..., u_m).$$

Da mit $1 \leq i \leq m$ und $1 \leq j \leq n$

$$u_i^* A v_j = \sigma_j u_i^* u_j = \sigma_j \delta_{i,j}$$

gilt, ergibt sich für die Matrizen $U = [u_1, ..., u_m]$ und $V = [v_1, ..., v_n]$

$$U^* A V = diag(\sigma_1, ..., \sigma_n).$$

Da die Spaltenvektoren von U und V jeweils orthonormal sind, gilt $U^{-1} = U^*$ und $V^{-1} = V^*$, und wir erhalten

$$A = U S V^*.$$

□

Definition 2.1.4 (Singulärwertzerlegung) *Sei $A \in \mathbb{C}^{m \times n}$ eine beliebige Matrix. Ohne Beschränkung der Allgemeinheit legen wir $m \geq n$ fest. Eine Zerlegung*

$$A = U S V^* = \begin{bmatrix} u_1 & \dots & u_m \end{bmatrix} \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} \begin{bmatrix} v_1 \dots v_n \end{bmatrix}^*$$

mit $U^*U = I_m$, $V^*V = I_n$ und $\Sigma = diag(\sigma_i)$ heißt Singulärwertzerlegung von A. Die Spalten der Matrizen U und V heißen linke bzw. rechte Singulärvektoren, die σ_i , $1 \leq i \leq n$, sind die Singulärwerte von A.
Wir numerieren die Singulärwerte von A, so daß gilt

$$\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r > \sigma_{r+1} = ... = \sigma_n = 0$$

Die betragsmäßig größten Singulärwerte nennen wir große Singulärwerte, und entsprechen heißen die betragsmäßig kleinsten, aber von Null verschiedenen Singulärwerte kleine Singulärwerte. Wir bezeichnen das Tripel (u_i, σ_i, v_i) als das i-te Singulärtripel der Matrix A.
Zur Vereinfachung der Notation legen wir fest, daß mit „Singulärvektoren“ stets auf eins normierte Eigenvektoren von A^*A bzw. AA^* gemeint sind.

Die Singulärwertzerlegung ist nicht eindeutig:

i) Zwar sind die Singulärvektoren durch die Forderung $V^*V = I_n$ normiert, doch ist mit v auch $-v$ normierter Singulärvektor.

ii) Im Falle eines mehrfachen Singulärwertes σ_k ist die Wahl einer Orthonormalbasis entweder von $Eig(A^*A, \sigma_k)$ oder $Eig(AA^*, \sigma_k)$ beliebig.

Da die im folgenden vorgestellten Algorithmen zur Berechnung der Singulärwertzerlegung reelle Matrizen voraussetzen, werden wir uns von nun an auf Matrizen $A \in \mathbb{R}^{m\times n}$ beschränken. Prinzipiell gelten die meisten Aussagen auch für komplexe Matrizen, insbesondere läßt sich die Berechnung der Singulärwertzerlegung einer komplexen Matrix $B \in \mathbb{C}^{m\times n}$ auf den reellen Fall mit einer Matrix $\tilde{B} \in \mathbb{R}^{2m\times 2n}$ zurückführen: Spalten wir B in Real- und Imaginärteil auf, $B = W + iY$, dann können wir die Singulärwertzerlegung von B berechnen als die Singulärwertzerlegung der reellwertigen Matrix

$$\tilde{B} = \begin{bmatrix} W & -Y \\ Y & W \end{bmatrix}.$$

Die Singulärvektoren von $\tilde{B}$ setzen sich dann aus den Real- und Imaginärteilen der Singulärvektoren von B zusammen. Diese Vorgehensweise ist angesichts des kubisch wachsenden Aufwands der Berechnung der Singulärwertzerlegung allerdings nur bedingt praktikabel.

Bevor wir uns nun der eigentlichen Berechnung der Singulärwertzerlegung zuwenden können, brauchen wir noch ein weiteres Hilfsmittel: die Matrixnorm.Wir definieren

i) die p-Norm: $$\|A\|_p = \sup_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p}, \quad p \in \{1, 2, \infty\}$$

ii) die Frobenius Norm: $$\|A\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2}$$

Korollar 2.1.5 *Sei $A \in \mathbb{R}^{m\times n}$, $rang(A) = r$, und $A = USV^T$ die Singulärwertzerlegung von A. Dann gilt:*

i) $\|A\|_2 = \sigma_1$

ii) $\|A\|_F = \sum_{k=1}^{r} \sigma_k^2$

iii) Sei $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T, k \leq r$. Dann ist A_k die beste Rang k Approximation an A im least squares Sinne:

$$\min_{rang(B)=k} \|A - B\|_2 = \|A - A_k\|_2 = \sigma_{k+1}$$

BEWEIS :

i) Sei $x \in \mathbb{R}^n$

$$\begin{aligned}
\frac{x^T A^T A x}{x^T x} &= \frac{x^T (VV^T) A^T A (VV^T) x}{x^T (VV^T) x} \\
&= \frac{(x^T V) S^2 (V^T x)}{(x^T V)(V^T x)}, \quad \text{substituiere } y := V^T x \\
&= \frac{y^T S^2 y}{y^T y} \\
&= \frac{\sum_{i=1}^{n} \sigma_i^2 y_i^2}{\sum_{i=1}^{n} y_i^2} \\
&\leq \sigma_1^2.
\end{aligned}$$

Es ist

$$\|A\|_2^2 = \sup_{x \neq 0} \left(\frac{\|Ax\|_2}{\|x\|_2} \right)^2 = \frac{x^T A^T A x}{x^T x} \leq \sigma_1^2;$$

wählen wir speziell $x = v_1$, als den zu σ_1 gehörigen rechten Singulärvektor, dann erhalten wir:

$$\frac{v_1^T A^T A v_1}{v_1^T v_1} = \sigma_1^2$$

und damit

$$\|A\|_2 = \sigma_1$$

ii) Für die Einträge $a_{i,j}$ von $A \in \mathbb{R}^{m \times n}$ gilt

$$a_{i,j} = \sum_{i=1}^{r} \sigma_k u_{k,i} v_{k,j},$$

und demnach für die Frobeniusnorm von A:

$$\begin{aligned}
\|A\|_F^2 &= \sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j}^2 \\
&= \sum_{k=1}^{r} \sigma_k^2 \sum_{i=1}^{m} u_{k,i}^2 \sum_{j=1}^{m} v_{k,j}^2 \\
&= \sum_{k=1}^{r} \sigma_k^2 \|u_k\|^2 \|v_k\|^2 \\
&= \sum_{k=1}^{r} \sigma_k^2
\end{aligned}$$

iii) [Golub Van Loan 1983, S. 19] Es gilt $\sigma_{max}(A - A_k) = \sigma_{k+1}$. Nach (i) ist $\|A - A_k\|_2 = \sigma_{k+1}$. Sei $B \in \mathbb{R}^{m\times n}$ beliebig mit $rang(B) = k$, und $\{x_1, ..., x_{n-k}\}$ eine Orthonormalbasis des $\mathcal{N}(B)$, dann gilt für

$$z \in span(x_1, ..., x_{n-k}) \cap span(v_1, ..., v_{k+1}) \ (\neq \{\}), \text{ mit } \|z\|_2 = 1$$

$$Bz = 0 \text{ und } Az = \sum_{i=1}^{k+1} \sigma_i < v_i, z > u_i,$$

$$\|A - B\|_2^2 \geq \|(A - B)z\|_2^2 = \|Az\|_2^2 = \sum_{i=1}^{k+1} \sigma_i^2 < v_i, z >^2 \geq \sigma_{k+1}^2$$

$\square$

Abkürzend schreiben wir im folgenden $\|.\|$ für die 2-Norm.

2.1.2 Äquivalente Eigenwertprobleme

Wie wir bereits beim Beweis der Existenz der Singulärwertzerlegung gesehen haben, sind die Singulärtripel (u_i, σ_i, v_i) definiert durch:

$$A^T A v_i = \sigma_i^2 v_i \quad und \quad A v_i = \sigma_i u_i$$

Es erscheint daher naheliegend, die Singulärwerte und -vektoren als die Eigenpaare von $A^T A$ zu berechnen. Die explizite Berechnung des Matrixprodukts $A^T A$ kann allerdings zu ernsthaften Störungen in den kleinen Singulärwerten führen. Eine Möglichkeit, dieses Problem zu umgehen, ist durch den folgenden Satz gegeben:

Satz 2.1.6 *Sei $A \in \mathbb{R}^{m\times n}, rang(A) = r \leq n$, $A = USV^T$ die Singulärwertzerlegung von A und*

$$\tilde{A} := \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$$

Ist λ ein positiver Eigenwert von $\tilde{A}$ und $z = (v_1, ..., v_n, u_1, ..., u_m)^T$ zugehöriger Eigenvektor der Norm $\sqrt{2}$ dann ist λ Singulärwert von A, und u,v sind die zugehörigen normierten Singulärvektoren.

Beweis : [Cullum, Willoughby 1985, S.179] Sei $A = USV^T$ die Singulärwertzerlegung von A,

$$V = [v_1, ..., v_n],\ U = [u_1, ..., u_m] \ und$$

$$S = \begin{bmatrix} \Sigma \\ 0 \end{bmatrix},\ \Sigma \in \mathbb{R}^{n\times n}$$

Wir definieren:

$$U_1 := [u_1, ..., u_n],\ U_2 := [u_{n+1}, ..., u_m],$$

$$Z = \frac{1}{\sqrt{2}} \begin{bmatrix} U_1 & U_1 & \sqrt{2}U_2 \\ V & -V & 0 \end{bmatrix}.$$

Dann gilt

$$\begin{aligned} Z^T Z &= \frac{1}{2} \begin{bmatrix} U_1^T U_1 + V^T V & U_1^T U_1 - V^T V & \sqrt{2}U_1^T U_2 \\ U_1^T U_1 - V^T V & U_1^T U_1 + V^T V & \sqrt{2}U_1^T U_2 \\ \sqrt{2}U_2^T U_1 & \sqrt{2}U_2^T U_1 & 2U_2^T U_2 \end{bmatrix} \\ &= I, \end{aligned}$$

also ist Z orthogonal, und wir erhalten für

$$\begin{aligned} \tilde{A}Z &= \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} U_1 & U_1 & \sqrt{2}U_2 \\ V & -V & 0 \end{bmatrix} \\ &= \frac{1}{\sqrt{2}} \begin{bmatrix} \Sigma U_1 & -\Sigma U_1 & 0 \\ \Sigma V & \Sigma V & 0 \end{bmatrix} \\ &= \frac{1}{\sqrt{2}} \begin{bmatrix} U_1 & U_1 & \sqrt{2}U_2 \\ V & -V & 0 \end{bmatrix} \underbrace{\begin{bmatrix} \Sigma & 0 & 0 \\ 0 & -\Sigma & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{=:\Lambda} \\ &= Z\,\Lambda. \end{aligned}$$

Demzufolge erhalten wir für die Singulärtripel (u_i, σ_i, v_i) von A, daß $(\sigma_i, \frac{1}{\sqrt{2}} \begin{pmatrix} u \\ v \end{pmatrix})$ und $(-\sigma_i, \frac{1}{\sqrt{2}} \begin{pmatrix} u \\ -v \end{pmatrix})$ Eigenpaare von $\tilde{A}$ sind.

□

2.2 Reduktion einer vollbesetzten Matrix auf Bidiagonalgestalt

Wie wir im vorigen Abschnitt gesehen haben, läßt sich die Berechnung der Singulärwertzerlegung auf das symmetrische Eigenwertproblem zurückführen. Da die Verfahren, mit denen sich Eigenwerte stabil und effizient berechnen lassen, iterative Verfahren sind, die pro Iterationsschritt aufwendige Operationen auf der Matrix A ausführen, wird die Martix A erst in eine möglichst einfach strukturierte, möglichst dünnbesetzte Form überführt, für die zum Beispiel eine QR-Zerlegung oder die Auswertung des charakteristischen Polynoms mit geringem Aufwand berechnet werden kann. Hierbei nutzen wir aus, daß die Eigenwerte einer Matrix invariant bezüglich sogenannter Ähnlichkeitstransformationen sind.

Definition 2.2.1 *Sei $A \in \mathbb{R}^{n\times n}$ und F eine beliebige nichtsinguläre $n \times n$ Matrix. Dann bezeichnet man die Abbildung:*

$$A \mapsto F^{-1}AF$$

als eine Ähnlichkeitstransformation.

Ist (λ, x) ein Eigenpaar der Matrix A, d.h. $Ax = \lambda x$ und $y := F^{-1}x$, dann folgt aus

$$F^{-1}AFy = F^{-1}Ax = \lambda F^{-1}x = \lambda y, \quad y \neq 0,$$

daß y Eigenvektor von $F^{-1}AF$ zum selben Eigenwert λ ist. Daher besitzen ähnliche Matrizen dieselben Eigenwerte.

Lemma 2.2.2 *Seien $X \in \mathbb{R}^{m\times m}, Y \in \mathbb{R}^{n\times n}$ orthogonale Matrizen, d.h,*

$$X^TX = XX^T = I_m, \; Y^TY = YY^T = I_n,$$

und $A \in \mathbb{R}^{m\times n}$. Definieren wir

$$B := X^TAY,$$

dann sind A^TA und B^TB ähnliche Matrizen.

BEWEIS :

$$B^TB = (X^TAY)^TX^TAY = Y^TA^T(XX^T)AY = Y^TA^TAY$$

Da Y regulär ist und $Y^{-1} = Y^T$, folgt die Behauptung.

□

Im folgenden werden wir sehen, wie geeignete orthogonale Matrizen konstruiert werden können, um eine vollbesetzte Matrix ähnlich auf Bidiagonalgestalt zu transformieren.

Definition 2.2.3 *Sei $v \in \mathbb{R}^n, v \neq 0$. Eine Matrix $P \in \mathbb{R}^{n\times n}$ der Form*

$$P = I - 2\frac{vv^T}{v^Tv}$$

heißt Householdermatrix.

Eine Householdermatrix P ist offensichtlich symmetrisch, und da

$$\begin{aligned} PP^T &= P^TP \\ &= (I - 2\frac{vv^T}{v^Tv})^2 \\ &= I - 4\frac{vv^T}{v^Tv} + 4\frac{vv^Tvv^T}{v^Tvv^Tv} \\ &= I, \end{aligned}$$

ist P auch orthogonal.
Mittels Householdermatrizen ist es auf recht einfache Art und Weise möglich, einen gegebenen Vektor $x \in \mathbb{R}^n$ auf $span(z)$, $z \in \mathbb{R}^n$ abzubilden. Interessant ist insbesondere der Fall $z = e_1$, wobei e_1 der erste Spaltenvektor der Einheitsmatrix im $\mathbb{R}^{n \times n}$ ist. Gesucht ist also ein Vektor $v \in \mathbb{R}^n$, so daß

$$Px = (I - 2\frac{vv^T}{v^Tv})x = x - 2\frac{v^Tx}{v^Tv}v \in span(e_1).$$

Dies impliziert $v \in span(x, e_1)$. Machen wir den Ansatz $v = x + \alpha e_1$, so ergibt sich:

$$v^Tx = x^Tx + \alpha x_1, \text{ und } v^Tv = x^Tx + 2\alpha x_1 + \alpha^2.$$

Einsetzen liefert:

$$\begin{aligned} Px &= x - 2\underbrace{\frac{x^Tx + \alpha x_1}{x^Tx + 2\alpha x_1 + \alpha^2}}_{c}(x + \alpha e_1) \\ &= (1 - 2c)x + 2c\alpha e_1 \end{aligned}$$

Es gilt $Px \in span(e_1)$, wenn $1 - 2c = 0$, d.h.

$$\begin{array}{rcccc} & 1 - 2\frac{x^Tx + \alpha x_1}{x^Tx + 2\alpha x_1 + \alpha^2} & = & 0 \\ \Leftrightarrow & x^Tx + 2\alpha x_1 + \alpha^2 & = & 2(x^Tx + \alpha x_1) \\ \Leftrightarrow & \alpha^2 & = & x^Tx \\ \Leftrightarrow & |\alpha| & = & \|x\|_2 \end{array}$$

Damit können wir die gewünschte Householdermatrix berechnen. Bei der praktischen Umsetzung sind aber noch einige wichtige Details zu beachten:

i) Ist der Vektor x nahe einem Vielfachen von e_1, dann ist die Norm von $v = x - sign(x_1)\|x\|_2 e_1$ sehr kleine, und es ist ein großer relativer Fehler in $\frac{2}{v^Tv}$ zu erwarten, wenn wir mit endlicher Genauigkeit rechnen. Daher ist die Wahl von

$$v = x + sign(x_1)\|x\|_2 e_1$$

numerisch günstiger.

ii) Bei der Berechnung von v^Tv kann es sowohl zu Unter- als auch Überlauf kommen (vgl. 2.3.1). Dieses Problem läßt sich aber einfach dadurch umgehen, daß wir zur Berechnung von v den Vektor $\tilde{x} = x/\|x\|_\infty$ statt x heranziehen. Zwar erhalten wir dadurch einen anderen Vektor v, die Householdermatrix bleibt jedoch identisch.

iii) Zur Anwendung einer Householdermatrix auf einen Vektor ist es nicht notwendig, die zugehörige Matrix explizit auszurechnen. Die Speicherung des sogenannten Reflexionsvektors“v und einer Konstanten β ist ausreichend:

$$Py = y + \beta < v, y > v, \;\; P \in \mathbb{R}^{n \times n} \text{ Householdermatrix}, \; y \in \mathbb{R}^n,$$

mit

$$\begin{aligned}
\beta &= \frac{2}{v^T v} \\
&= \frac{2}{(x+\alpha e_1)^T(x+\alpha e_1)}, \qquad \alpha = sign(x_1)\|x\| \\
&= \frac{2}{\alpha^2 + 2\alpha x_1 + \alpha^2} \\
&= \frac{1}{\alpha(x_1+\alpha)}
\end{aligned}$$

Somit kann die Anwendung der Householdermatrix $P \in \mathbb{R}^{n\times n}$ auf einen Vektor $y \in \mathbb{R}^n$ mit lediglich $4n$ Operationen berechnet werden.

iv) Ist ein Vektor $x \in \mathbb{R}^m$ gegeben, dessen Komponenten $x_{l+1}, ..., x_{l+n}$ durch eine Householdermatrix auf Null abgebildet werden sollen, so kann dies mit der Matrix

$$\tilde{P} = \begin{bmatrix} I_{l-1} & 0 & 0 \\ 0 & P & 0 \\ 0 & 0 & I_{m-n-l+1} \end{bmatrix}$$

erreicht werden, wenn P die Householdermatrix ist, die den Vektor $(x_l, ..., x_{l+n})^T$ auf ein Vielfaches des Einheitsvektors $e_1 \in \mathbb{R}^n$ abbildet.

v) Soll das Produkt einer Houserholdermatrix $P \in \mathbb{R}^{n\times n}$ mit einer Matrix $A \in \mathbb{R}^{n\times l}$ berechnet werden, so läßt sich auf modernen Computern eine deutliche Effizienzsteigerung erreichen, wenn der Algorithmus mittels Matrix-Vektor-Produkten statt Skalarprodukten implementiert wird:

$$PA = P[a_1, ..., a_l] = [a_1 + \beta < v, a_1 > v \ , \ ... \ , \ a_l + \beta < v, a_l > v]$$

Die Skalarprodukte $< v, a_i >$ lassen sich simultan berechnen als $g = A^T v$. Dann ist

$$PA = A + \beta g v^T$$

Algorithmus 2.2.4 Gegeben sei eine Matrix $A \in \mathbb{R}^{m\times n}$, $A = [a_1, ..., a_n]$. Dieser Algorithmus berechnet einen Reflexionsvektor v, so daß die zugehörige Householdermatrix H den ersten Spaltenvektor a_1 auf ein Vielfaches des Einheitsvektors $e_1 \in \mathbb{R}^n$ abbildet. Anschließend wird die Matrix A mit dem Produkt HA überschrieben.

$$\begin{aligned}
a_1 &:= \frac{a_1}{\|a_1\|_\infty} \\
\alpha &= sign(a_{1,1})\|a_1\|_2 \\
\beta &= \frac{1}{\alpha(a_{1,1}+\alpha)} \\
v &= a_1 + \alpha e_1
\end{aligned}$$

$$\begin{array}{rcl}
a_1 & := & \alpha e_1 \\
g & = & \beta [a_2, ..., a_n]^T v \\
{[a_2, ..., a_n]} & := & [a_2, ..., a_n] + (v_2, ..., v_m)^T g^T
\end{array}$$

Bemessen wir den Aufwand für Division, Wurzelziehen, und Vergleiche mit jeweils einer Operation, so erhalten wir für den obigen Algorithmus $4mn$ Operationen.

Eine weitere Effizienzsteigerung gegenüber Algorithmus (2.2.4) kann durch eine Block-Formulierung des Householderverfahrens erreicht werden, wenn der größte Teil der Rechnungen als Matrix-Matrix-Multiplikationen realisiert werden kann. Gegeben sei eine Matrix $A \in \mathbb{R}^{m \times n}$ mit einer Blockaufteilung

$$A = \begin{bmatrix} A^{1,1} & \dots & A^{1,N} \\ \vdots & & \vdots \\ A^{M,1} & \dots & A^{M,N} \end{bmatrix}.$$

Wir berechnen nun eine orthogonale Matrix Q so daß

$$Q^T A = \begin{bmatrix} R^{1,1} & A^{1,2} & \dots & A^{1,N} \\ 0 & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ 0 & A^{M,2} & \dots & A^{M,N} \end{bmatrix}.$$

Bischof und Van Loan [1987] haben die „WY-Darstellung" zur Berechnung und Anwendung von Q vorgeschlagen. Die Idee ist hierbei, eine QR-Zerlegung von $[A^{1,1}, \dots, A^{M,1}]^T, \quad A^{i,j} \in \mathbb{R}^{l \times k}$ mittels Householderreflexionen $P_1, \dots, P_{k-1}$ zu berechnen:

$$\begin{array}{rcl}
Q_i & = & P_i \cdot \dots \cdot P_1, \quad 1 \leq i \leq k-1 \\
Q & = & Q_{k-1}.
\end{array}$$

Die Matrizen Q_i lassen sich dann in der Form

$$Q_i = I + W_i Y_i^T, \qquad i = 1, \dots, k-1 \tag{2.3}$$

aufschreiben, mit $m \times i$ Matrizen W_i und Y_i: Ist $P_i = I + \beta_i v_i v_i^T$, dann setzen wir $W_1 = \beta_1 v_1$ und $Y_1 = v_1$, und erhalten $Q_i, \quad 2 \leq i \leq N-1$ durch eine Zweitermrekursion:

$$\begin{array}{rcl}
Q_i & = & Q_{i-1} P_i \\
& = & Q_{i-1}(I + u_i v_i^T) \\
& = & Q_{i-1} + Q_{i-1} u_i v_i^T \\
& = & I + W_{i-1} Y_{i-1}^T + Q_{i-1} u_i v_i^T \\
& = & I + \underbrace{[W_{i-1}, Q_{i-1} u_i]}_{W_i} \underbrace{[Y_{i-1}, v_i]}_{Y_i}^T.
\end{array}$$

Der Rechenaufwand bei der Anwendung eines Block-Reflektors $Q = Q_{k-1}$ auf eine Blockmatrix $A^{i,j}$ ist wesentlich bestimmt durch Matrix-Matrix Multiplikationen:

$$Q^T A^{i,j} = A^{i,j} + Y_{N-1}(W_{N-1}^T A^{i,j}),$$

Der Aufwand, in Operationen gemessen, ist gegenüber Algotithmus (2.2.4) etwa um einen Faktor $(1 + 2/N)$ größer; wir sollten also N möglichst klein wählen. Eine zu geringe Blockgröße bedeutet aber geringere Effizienz (vgl. Kap. 4). Die Blockgröße muß also für jeden Rechner optimiert werden.

Wir können den Algorithmus (2.2.4), wie auch die Blockvariante nach (2.3) zur Berechnung einer Familie von Matrizen $A = A_0, ..., A_{n-1}$ einsetzen, deren letztes Element eine Bidiagonalmatrix ist. Zur Vereinfachung der Notation beschränken wir uns auf die klassische Householder-Methode. Bezeichnen wir mit $\mathcal{H}_{k,l}^{i,j}(A)$ die Anwendung von (2.2.4) auf die Untermatrix

$$\begin{bmatrix} a_{i,j} & \cdots & \cdots & a_{i,l} \\ \vdots & & & \vdots \\ \vdots & & & \vdots \\ a_{k,j} & \cdots & \cdots & a_{k,l} \end{bmatrix},$$

dann erhalten wir für

$$\begin{aligned} \tilde{A}_1 &= \mathcal{H}_{m,n}^{1,1}(A) \\ &= \begin{bmatrix} * & \cdots & \cdots & * \\ 0 & * & \cdots & * \\ \vdots & \vdots & & \vdots \\ 0 & * & \cdots & * \end{bmatrix} \end{aligned}$$

und

$$\begin{aligned} A_1 &= \mathcal{H}_{n,m}^{2,1}(\tilde{A}_1^T) \\ &= \begin{bmatrix} * & * & 0 & 0 \\ 0 & * & \cdots & * \\ \vdots & \vdots & & \vdots \\ 0 & * & \cdots & * \end{bmatrix} \end{aligned}$$

eine Matrix, deren erste Zeile und Spalte bereits auf Bidiagonalgestalt transformiert sind. Wenden wir dieses Schema sukzessive auf die rechten Untermatrizen an, die noch nicht auf Bidiagonalgestalt transformiert sind, so erhalten wir

$$A_{n-1} = \left[\begin{array}{ccccc} * & * & & & \\ & * & * & & \\ & & \ddots & \ddots & \\ & & & \ddots & * \\ & & & & * \\ \hline 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 \end{array}\right] := \left[\begin{array}{c} B \\ 0 \end{array}\right],$$

wenn

$$\begin{aligned} A_s &= (\mathcal{H}_{n,m}^{s+1,s}(\mathcal{H}_{m,n}^{s,s}(A_{s-1})^T))^T, \ 1 \leq s < n-1, \\ A_{n-1} &= \mathcal{H}_{m,n}^{n-1,n-1}(A_{n-2}). \end{aligned}$$

Da die Anwendung des Algorithmus (2.2.4) auf die Matrix A der Multiplikation einer Householdermatrix P mit A entspricht, können wir schreiben:

$$\left[\begin{array}{c} B \\ 0 \end{array}\right] = \underbrace{P_{n-1}^T P_{n-2}^T \ldots . P_1^T}_{:=P^T} \ A \ \underbrace{Q_1 Q_2 \ldots Q_{n-2}}_{:=Q}$$

Das Produkt orthogonaler Matrizen ist offensichtlich wieder eine orthogonale Matrix. Demzufolge haben nach Lemma (2.2.2) die Matrizen A und A_{n-1} die gleichen Singulärwerte.
Sind $A = USV^T$ und $A_{n-1} = XZY^T$ die Singulärwertzerlegungen von A und A_{n-1}, dann lassen sich die Singulärvektoren von A aus denen von $A_{n-1} = P^T AQ$ folgendermaßen berechnen:

$$u_i = Px_i, \ v_i = Qy_i.$$

Bei der praktischen Durchführung können wir uns darauf beschränken, die Singulärtripel zu den von Null verschiedenen Singulärwerten zu berechnen. Die restlichen Singulärvektoren lassen sich durch Orthogonalisierung bestimmen: Sei $A \in \mathbb{R}^{m \times n}, m \geq n, rang(A) = r$. Initialisieren wir $m - r$ Vektoren $u_r, \ldots, u_m$ mit linear unabhängigen Zufallsvektoren und wenden ein Orthogonalisierungsverfahren auf die Vektoren $u_1, \ldots, u_m$ an, so erhalten wir wegen $\mathbb{R}^m = span(u_1, \ldots, u_r) \oplus span(u_{r+1}, \ldots, u_m)$ die zu $\sigma = 0$ gehörenden Singulärvektoren.

Eine weitere Möglichkeit, eine vollbesetzte Matrix mit Hilfe orthogonaler Matrizen auf Bidiagonalgestalt zu transformieren, besteht in der Anwendung sogenannter Givens-Rotationen.

Definition 2.2.5 *Als Givens-Rotationen bezeichnet man Matrizen der Form*

$$\Omega(i,j,\phi) := \begin{array}{c} \left[\begin{array}{ccccc} I & & & & \\ & c & & s & \\ & & I & & \\ & -s & & c & \\ & & & & I \end{array}\right] \end{array} \begin{array}{c} \\ \leftarrow i \\ \\ \leftarrow j \\ \\ \end{array}$$

wobei I jeweils die Einheitsmatrix passender Dimension ist und

$$c = \cos\phi, \quad s = sin\phi.$$

Anmerkung 2.2.6 Sei $A \in \mathbb{R}^{m\times n}$ und $\Omega(i,j,\phi) \in \mathbb{R}^{m\times m}$ eine Givensrotation. Dann gelten die folgenden Aussagen:

i) Geometrisch beschreibt $\Omega(i,j,\phi)$ eine Drehung in der (i,j)-Ebene. Multiplizieren wir $\Omega(i,j,\phi)$ von links mit der Matrix A, so ist das Produkt mit der Matrix A bis auf die Zeilen i und j identisch. Multiplizieren wir $\Omega(i,j,\phi)$ von rechts mit der Matrix A, so werden nur die Spalten i und j verändert. Wenden wir $\Omega(i,j,\phi)$ auf einen Vektor $x \in \mathbb{R}^m$ an, so folgt für $i < j$:

$$\Omega(i,j,\phi)x = (x_1,\ldots,x_{i-1}, cx_i + sx_j, x_{i+1},\ldots,x_{j-1}, -sx_i + cx_j, x_{j+1},\ldots,x_m)^T.$$

ii) Durch geeignete Wahl von ϕ bzw. von c und s, können wir nun einzelne Elemente des Vektors x eliminieren. Wir können uns dabei ohne Einschränkung auf den zweidimensionalen Fall beschränken:
Gesucht sind $c, s \in \mathbb{R}$, so daß $c^2 + s^2 = 1$, und

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{pmatrix} x_i \\ x_j \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}.$$

Lösen des Gleichungssystems ergibt

$$s = \frac{x_j}{r}, \quad c = \frac{x_i}{r}, \quad r^2 = x_i^2 + x_j^2.$$

Um bei der Berechnung von $x_i^2 + x_j^2$ Exponentenüberlauf zu vermeiden, geht man bei der Bestimmung von c und s folgendermaßen vor:

falls $|x_j| \geq |x_i|$, dann setze

$$t = \frac{x_i}{x_j}, \quad s = \frac{1}{\sqrt{1+t^2}}, \quad c = st$$

sonst

$$t = \frac{x_j}{x_i}, \quad c = \frac{1}{\sqrt{1+t^2}}, \quad s = ct.$$

Ist $x \in \mathbb{R}^m$ und $\Omega(i, i+1, *), 1 \leq i \leq n-1$ die Givensrotation, die die Komponente $i+1$ eines gegebenen Vektors eliminiert, dann ist

$$\Omega(1,2,*)\Omega(2,3,*)\cdots\Omega(m-1,m,*)x$$

ein Vielfaches des ersten Einheitsvektors $e_1 \in \mathbb{R}^m$. Da Givens-Rotationen offensichtlich orthogonale Matrizen sind, können wir ähnlich (2.2.4) einen Algorithmus zur Bidiagonalisierung einer vollbesetzten Matrix konstruieren. Aufgrund der besseren Effizienz des auf Householdermatrizen basierenden Verfahrens, ist dies allerdings wenig sinnvoll. Wenn die zu transformierende Matrix nur dünn besetzt ist, dann kann der Einsatz von Givens-Rotationen vorteilhaft sein. Wollen wir beispielsweise eine Bandmatrix transformieren, so wird durch die Anwendung einer Householdertransformation die Bandstruktur zerstört, während durch Anwendung einer geeigneten Familie von Givensrotationen die Struktur erhalten werden kann. Je nach Bandbreite reduziert sich hierdurch die Gesamtzahl der auszuführenden Operationen erheblich.

2.3 Stabilitäts- und Genauigkeitsfragen

Ein qualitativ hochwertiges Programm zur Eigenwertberechnung liefert Ergebnisse, die bis auf einen Fehler in der Größenordnung der Rechengenauigkeit mit den exakten Ergebnissen übereinstimmen. Als exakte Ergebnisse werden in diesem Zusammenhang Eigenwerte und Eigenvektoren gesehen, die mit nicht endlicher Genauigkeit berechnet und abgespeichert werden. Ein Computer repräsentiert reelle Zahlen durch eine feste, endliche Anzahl von Dezimalstellen. Das Abspeichern einer Matrix mit reellen Einträgen bewirkt also schon eine Störung der Matrixelemente in der Größenordnung der Rechengenauigkeit des Computers. Daß sich diese und andere Störungen im Laufe der Berechnungen nicht wesentlich verstärken, muß das Ziel bei der Entwicklung eines guten Programmes sein.

2.3.1 Rundungsfehler

Bei heutigen Digitalrechnern ist die normalisierte Fließkommadarstellung zur Speicherung reeller Zahlen die Regel. Hierbei bedient man sich einer Exponentialschreibweise zur Basis b : $x = \alpha b^e$, $x \in \mathbb{R}$. Aufgrund der begrenzten Speicherkapazität können für α und e jeweils nur Ziffernfolgen mit fester Länge t und s im Speicher abgelegt werden. Demzufolge ist auch nur eine endliche Teilmenge FL der reellen Zahlen im Computer darstellbar. Für $x \in \mathbb{R}$ sei $fl(x) \in FL$ die zugehörige Computerzahl, die wir wie folgt ermitteln: Schreibe x in Exponentialdarstellung $x = \alpha b^e$, und zwar so, daß α genau eine von Null verschiedene Vorkommastelle besitzt - vorausgesetzt $x \neq 0$. Dann runde α auf t Stellen. Eine Rundung des Exponenten macht natürlich keinen Sinn. Wenn Zahlen mit zu großem Exponenten gespeichert werden sollen, hält der Computer das laufende

Programm an und meldet einen Fehler - wir sagen das Programm bricht zusammen. Je nach Rechner ist die Handhabung zu kleiner (negativer) Exponenten unterschiedlich; zumeist wird das Programm nicht unterbrochen, statt dessen wird die Zahl als NAN (not a number) markiert und im weiteren Ablauf wie Null behandelt. Im Falle zu großer Exponenten sprechen wir von (Exponenten-) Überlauf, bei zu kleinem Exponenten von Unterlauf.

Reale Rechner arbeiten mit einer Zahlendarstellung zur Basis 2. Zu Analysezwecken können wir aber auch die gewohntere Darstellung zur Basis 10 heranziehen. Der relative Fehler beim Abspeichern einer reellen Zahl x ist dann

$$\left|\frac{fl(x) - x}{x}\right| \leq \frac{5 \cdot 10^{-(t+1)}}{|\alpha|} \leq 5 \cdot 10^{-t}$$

wegen der Normierung $|\alpha| \geq 10^{-1}$. Die Zahl $eps := 5 \cdot 10^{-t}$ heißt die (relative) Maschinengenauigkeit. Auch bei der Auswertung der elementaren Operationen $(+, -, \star, \div)$ kommt es zu Rundungsfehlern, für die jeweils gilt:

$$fl(x \circ y) = (x \circ y)(1 + \epsilon), \quad |\epsilon| \leq eps$$

Bei der Rundungsfehleranalyse zusammengesetzter Operationen gibt es zwei Strategien: Wir betrachten einen Algorithmus f, der die Eingabedaten x auf die Ergebnisse y abbildet.

i) Bei der Vorwärtsanalyse wird versucht, eine Fehlerschranke in der Form $\|y - f(x)\| \leq g(eps, x)$ anzugeben. Hierbei ist y das exakte Ergebnis.

ii) Bei der Rückwärtsanalyse wird die Menge der Eingabedaten, die zum gleichen Ergebnis führen können betrachtet. Typischerweise erhält man Fehlerschranken der Art:
Ist $f(x) = f(\tilde{x}) = y$ dann ist $\|x - \tilde{x}\| \leq h(eps, x)$

Die Angabe möglichst exakter Fehlerschranken ist dem Altmeister der Fehlerrechnung J. H. Wilkinson zufolge nur von eingeschränkter Bedeutung [nach Golub, Van Loan, 1983, S. 36]. Besonders Vorwärtsfehleranalysen erweisen sich in der Praxis meist als zu pessimistisch. Ziel der Analyse sollte es sein, mögliche Instabilitäten herauszustellen und gegebenenfalls Auswege anzubieten.

2.3.2 Fehlerabschätzungen

Wir entfernen uns nun wieder von den technisch bedingten Rechenfehlern und untersuchen nun, wie sich allgemein Störungen in den Eingangsdaten (der Matrix) auf die Singulärwerte und -vektoren auswirken. Hierbei können wir uns auf das äquivalente symmetrische Eigenwertproblem zurückziehen.

Satz 2.3.1 *Seien B und E reelle $n \times n$ Matrizen; B sei symmetrisch und λ sei Eigenwert der Matrix $B + \epsilon\, E$, $\epsilon \in \mathbb{R}$. Dann gilt:*

$$\min_{1 \le i \le n} (\lambda_i(B) - \lambda) \le \epsilon\, \|E\|.$$

Beweis : [Wilkinson 1965, S. 87] Seien (λ_i, v_i) die normierten Eigenpaare von B und $V = [v_1, \ldots, v_n]$, dann können wir B darstellen als

$$B = V \Lambda V^T \text{ mit } \Lambda = diag(\lambda_i).$$

Da λ Eigenwert von $B + \epsilon\, E$ ist, gilt

$$det(B + \epsilon\, E - \lambda I) = 0.$$

Weiterhin ist

$$\begin{aligned} V^T(B + \epsilon\, E - \lambda I)V &= diag(\lambda_i - \lambda) + \epsilon\, V^T E V \\ &= diag(\lambda_i - \lambda)(I + \epsilon\, diag(\lambda_i - \lambda)^{-1} V^T E V). \end{aligned}$$

Die Determinante der rechten Seite ist weiterhin Null. Wenn $\lambda \neq \lambda_i$ für $1 \le i \le n$, dann muß gelten

$$det(I + \epsilon\, diag(\lambda_i - \lambda)^{-1} V^T E V) = 0.$$

Das ist aber nur möglich, wenn

$$\begin{array}{rcl} & \|\epsilon\, diag(\lambda_i - \lambda)^{-1} V^T E V\| & \ge & 1 \\ \Rightarrow & \epsilon\, \|diag(\lambda_i - \lambda)^{-1}\| \|E\| & \ge & 1 \\ \Leftrightarrow & \epsilon\, \|E\| & \ge & \min_i(|\lambda_i - \lambda|). \end{array}$$

□

Störungen in der Matrix in der Größenordnung von ϵ bewirken also nur ebensolche in den Eigenwerten. Wir werden sehen, daß sich die Singulärwerte einer Matrix A berechnen lassen, ohne das Produkt $A^T A$ explizit auszurechnen. Die obige Aussage gilt dann auch für die Singulärwerte.

Bei der Berechnung der Eigenvektoren symmetrischer Matrizen ist die Situation etwas delikater:

Satz 2.3.2 *Sei $A \in \mathbb{R}^{n \times n}$ symmetrisch, und $y \in \mathbb{R}^n$ ein beliebiger, aber normierter Vektor. Wir definieren $\rho := y^T A y$; $\lambda = \lambda_k$ der Eigenwert von A, der ρ am nächsten liegt, und v als den zu λ gehörigen Eigenvektor. Weiterhin sei*

$$\gamma = \min_{i \neq k} |\lambda_i(A) - \rho|$$

und $\psi = \angle(y, v)$. Dann gilt:

$$|\sin \psi| \le \underbrace{\|Ay - \rho y\|}_{=: r(y)} / \gamma.$$

$r(y)$ können wir als Maß für die Güte eines approximierten Eigenvektors interpretieren. Wenn y Eigenvektor von A ist, dann gilt $r(y) = 0$. Da der Rayleigh-Quotient $\rho(y)$ sowie A stetig sind, ist auch $r(y)$ stetig. Trotzdem kann der Winkel ψ im Vergleich zu $r(y)$ beliebig groß werden, wenn γ klein ist. Wenn γ, also der Abstand benachbarter Eigenwerte, klein ist, sprechen wir von wenig oder kaum separierten Eigenwerten. In der Praxis entstehen diese auch durch Rundungsfehler, wenn es sich ursprünglich um mehrfache Eigenwerte handelte.

Beweis : [nach Parlett 1980, S. 222] Sei

$$y = v\cos\psi + w\sin\psi \quad \text{mit } \|w\| = 1, w \perp v, \text{ und } w \in span(v, y) \tag{2.4}$$

Es ist

$$\begin{aligned} r(y) &= (A-\rho I)v\cos\psi + (A-\rho I)w\sin\psi \\ &= v(\lambda-\rho)\cos\psi + (A-\rho I)w\sin\psi, \end{aligned}$$

denn $Av = \lambda v$. Es gilt $v^T(A-\rho I)w = 0$, und daher

$$\|r(y)\|^2 = (\lambda-\rho)^2\cos^2\psi + \|(A-\rho I)w\|^2\sin^2\psi$$

$$\Rightarrow \sin^2\psi \geq \frac{\|r(y)\|^2}{\|(A-\rho I)w\|^2}$$

Der Nenner wird minimal, wenn wir für w den normierten Eigenvektor zum kleinsten Eigenwert von $(A-\rho I)$ einsetzen. w hat aber nach (2.4) keine Anteile in dessen Richtung, so daß der Nenner minimal wird, wenn wir für w den Eigenvektor zum zweitkleinsten Eigenwert einsetzen. Damit ist die Behauptung bewiesen.

$\square$

2.3.3 Numerische Eigenschaften orthogonaler Matrizen

Der Einsatz orthogonaler Matrizen zur Reduktion einer vollbesetzten Matrix $A \in \mathbb{R}^{m\times n}$ auf Bidiagonalgestalt wurde in (2.2) vorgeschlagen, ohne seine Unbedenklichkeit in Bezug auf Rundungsfehler zu überprüfen. Dies werden wir nun nachholen.

Nach (2.2.2) können wir uns auf den Fall symmetrischer Matrizen zurückziehen.

Satz 2.3.3 *Sei $A \in \mathbb{R}^{n\times n}$ symmetrisch und A_j, $1 \leq j \leq k$, eine Familie ähnlicher Matrizen mit*

$$\begin{aligned} A_1 &= A \\ A_j &= P_j^T A_{j-1} P_j, \ 1 \leq j \leq k. \end{aligned}$$

P_j seien orthogonale Matrizen, die von A_{j-1} abhängen. Dann ist A_k ähnlich zu einer Matrix $A + M$ mit $\|M\|_F \leq nkeps\|A\|_F$.

BEWEIS : [nach Parlett 1980, S.94-96, und Wilkinson 1965] Bei der praktischen Berechnung einer Familie der angegebenen Art treten Rundungsfehler sowohl bei der Berechnung von P_j auf, als auch bei der Auswertung von $P_j^T A_{j-1} P_j$: Es sei

$$P_j = G_j + F_j,$$

wobei G_j eine unbekannte orthogonale Matrix ist, und F_j die Fehlermatrix. $\bar{W}_j$ sei der Fehler bei der Auswertung der Transformation

$$A_j = P_j^T A_{j-1} P_j + \bar{W}_j.$$

Setzen wir $P_j = G_j + F_j$ ein, so ist

$$\begin{aligned} A_j &= (G_j + F_j)^T A_{j-1}(G_j + F_j) + \bar{W}_j \\ &= G_j^T A_{j-1} G_j + W_j \end{aligned}$$

mit

$$W_j = \bar{W}_j + G_j^T A_{j-1} F_j + F_J^T A_{j-1} G_j + F_J^T A_{j-1} F_j. \tag{2.5}$$

Die Matrix W_j steht also für den Gesamtfehler im j-ten Schritt. Betrachten wir das letzte Familienelement

$$\begin{aligned} A_k &= G_k^T A_{k-1} G_k + W_k \\ &= G_k^T (G_{k-1}^T A_{k-2} G_{k-1} + W_{k-1}) G_k + W_k \\ &= \cdots \\ &= \mathrm{J}_0^T A_0 J_0 + \sum_{j=1}^{k} J_j^T W_j J_j \end{aligned}$$

mit $J_j = G_{j+1} \cdots G_k, \quad J_k = I$. Wir können nun schreiben

$$A_k = J_0^T (A_0 + M) J_0$$

mit

$$M = J_0 (\sum_{j=1}^{k} J_j^T W_j J_j) J_0^T.$$

Damit haben wir gezeigt, daß die mit Rundungsfehlern berechnete Matrix A_k exakt ähnlich zu einer Matrix $A + M$ ist. Da G_j und damit J_j orthogonale Matrizen sind, haben wir

$$\|M\|_F \leq \sum_{j=1}^{k} \|W_j\|_F.$$

Wir benutzen nun (2.5) und die Orthogonalität von G zur Abschätzung

$$\|W_j\|_F \leq \|\bar{W}_j\|_F + \|A_{j-1}\|_F (2\|F_j\|_F + \|F\|_F^2).$$

An dieser Stelle brauchen wir nun doch eine Aussage über den Fehler bei der Berechnung von P_j. Zum Beispiel hat Wilkinson [1965, S.152-162] den Fehler in praktisch berechneten Householdermatrizen mit $10eps$ angegeben. Diese Schranke hängt aber von der jeweiligen technischen und algorithmischen Realisierung ab. Wir gehen daher davon aus, daß die Matrizen P_j mit einem geringen Fehler in der Größenordnung der Rechengenauigkeit berechnet werden können,

$$\|F_j\|_F \leq c \ eps, \quad 1 \leq j \leq k,$$

und daß der Fehler der Auswertung von $P_j^T A_j P_j$ beschränkt ist durch:

$$\|\bar{W}_j\|_F \leq n \ eps \ \|A_j\|_F.$$

Insgesamt erhalten wir unter Vernachlässigung von Termen $\mathcal{O}\ (eps^2)$ und der Annahme $\|A_j\|_F \leq (1+eps)\|A_{j-1}\|_F$

$$\begin{aligned}
\|M\|_F &\leq \sum_{j=1}^{k} \|W_j\|_F \\
&\leq \sum_{j=1}^{k} \|\bar{W}_j\|_F + \|A_{j-1}\|_F \left(2\|F_j\|_F + \|F_j\|_F^2\right) \\
&\doteq \sum_{j=1}^{k} n \ eps \ \|A_{j-1}\|_F + 2 \, c \, eps \|A_{j-1}\|_F \\
&\dot{\leq} \ n \ k \ eps \ \|A\|_F, \qquad \text{für } 2c \leq n.
\end{aligned}$$

□

In der Praxis erweist sich diese Fehlerschätzung als viel zu pessimistisch. Die eigentliche Aussage des Satzes ist es, daß die durch Rundungsfehler verursachten Störungen die Ähnlichkeit der Matrizen A_j, $j = 1, 2, \ldots$, nicht wesentlich beeinträchtigen. Nach (2.3.3) hängt der zu erwartende Fehler allerdings von der Problemgröße n ab. Bei großen Matrizen ($n > 500$) und einfacher Rechengenauigkeit kann $\|M\|_F$ also durchaus bedenklich groß werden. Eine wenig aufwendige Möglichkeit, den Fehler gering zu halten, ist dann gegeben, wenn wir beim Aufstellen von Matrizen bzw. Vektoren die Möglichkeit haben, die Anordnung der Einträge zu beeinflussen. Bei der Rückwärtsanalyse des Skalarproduktes $fl(< x, y >) =< \bar{x}, y >$ ergibt sich für den relativen Fehler in der i-ten Komponente [Parlett 1980,S29-30]

$$\frac{|\bar{x}_i - x_i|}{|x_i|} \approx (i+1)eps,$$

wenn die Produkte $x_i y_i$ in aufsteigender Reihenfolge aufsummiert werden. Sofern die Möglichkeit gegeben ist, ist es demnach numerisch günstiger, die Elemente der Vektoren bzw. Matrizen so anzuordnen, daß $|x_i| \leq |x_{i+1}|$.

2.4 Der QR-Algorithmus

Definition 2.4.1 *Sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische Matrix.*

i) Eine Zerlegung $A = QR$ mit einer orthogonalen Matrix Q und einer rechten oberen Dreiecksmatrix R heißt QR-Zerlegung von A.

ii) Eine Folge von Matrizen A_i, $A_1 = A$ und

$$\begin{aligned} A_i - \mu_i I &= Q_i R_i \\ A_{i+1} &= R_i Q_i + \mu_i I, \end{aligned}$$

nennen wir die (geshiftete) QR-Folge zu A. Die μ_i heißen Shift.

Zur Berechnung der QR-Zerlegung können sowohl Householdermatrizen als auch Givensrotationen eingesetzt werden. Wir werden sehen, daß die QR-Folge unter gewissen Voraussetzungen gegen eine Diagonalmatrix konvergiert, deren Diagonaleinträge gerade die Eigenwerte von A sind. Bei der Untersuchung der Konvergenzeigenschaften des QR-Verfahrens erweist sich dessen Verwandtschaft zu den Vektoriterationen als hilfreich.

Satz 2.4.2 *Gegeben sei eine symmetrische Matrix $A \in \mathbb{R}^{n\times n}$ und ein $x_0 \in \mathbb{R}^n$ mit $< x_0, v_1 > \neq 0$, $< x_0, v_n > \neq 0$; (λ_i, v_i) seien die Eigenpaare zu A, $\lambda_1 > \ldots > \lambda_n > 0$.*

i) Potenzmethode: Die Folge

$$x_{i+1} = \frac{y_{i+1}}{\|y_{i+1}\|}, \text{ mit } y_{i+1} = Ax_i,$$

konvergiert gegen einen normierten Eigenvektor von A zum Eigenwert λ_1.

ii) Inverse Vektoriteration: Die Folge

$$x_{i+1} = \frac{y_{i+1}}{\|y_{i+1}\|}, \text{ mit } x_i = Ay_{i+1},$$

konvergiert gegen einen normierten Eigenvektor von A zum Eigenwert λ_n.

Beweis : [Parlett 1980, S.61-62] Wir betrachten die orthogonale Zerlegung

$$x_i = v_1 \cos\phi_i + w_i \sin\phi_i,$$

$$\|v_1\| = \|w_i\| = 1, \quad v_1 \perp w_i, \quad i = 1, 2, \ldots \text{ und } \phi_i = \angle(x_i, v_1)$$

i) Anwendung der Matrix A auf diese Zerlegung ergibt

$$Ax_i = \lambda_1 v_1 \cos\phi_i + \left(\frac{Au_i}{\|Au_i\|}\right)\|Au_i\| \sin\phi_i. \tag{2.6}$$

Da $u_i \perp v_1$, ist auch $Au_i \perp v_1$. Koeffizientenvergleich liefert

$$u_{i+1} = \frac{Au_i}{\|Au_i\|}.$$

x_{i+1} ist ein Vielfaches von Ax_i und somit (2.6) eine orthogonale Zerlegung von x_{i+1}, und es gilt

$$\tan\phi_{i+1} = \frac{\|Au_i\| \sin\phi_i}{\lambda_1 \cos\phi_i}.$$

Sei $A^\perp$ die Einschränkung von A auf $v_1^\perp$, dann erhalten wir die Schranke

$$\|Au_i\| = \|A^\perp u_i\| \leq \|A^\perp\| = \lambda_2.$$

Wir interessieren uns für das Verhalten von ϕ_i für $i \to \infty$. Hierzu betrachten wir den Quotienten

$$\rho = \frac{\tan\phi_{i+1}}{\tan\phi_i} = \frac{\|Au_i\|}{\lambda_1} \leq \frac{\lambda_2}{\lambda_1} < 1.$$

Für $0 \neq |\tan\phi_0| < \infty$ und $k \to \infty$ gilt dann

$$|\phi_i| \leq |\tan\phi_i| \leq \left(\frac{\lambda_2}{\lambda_1}\right)^i \tan\phi_0 \to 0$$

Die Konvergenzgeschwindigkeit wird also wesentlich durch das Problem bestimmt. Lediglich durch die Wahl eines guten Startvektors kann diese noch verbessert werden.

ii) Die inverse Vektoriteration ist offensichtlich die Anwendung der Potenzmethode auf A^{-1}. Da der größte Eigenwert von A^{-1} gerade der Kehrwert des kleinsten Eigenwertes von A ist, ist die Behauptung bewiesen. Der Konvergenzfaktor ist hier durch die kleinen Eigenwerte bestimmt: Sei $\phi = \angle(v_n, x_i)$

$$\rho = \frac{\tan\phi_{i+1}}{\tan\phi_i} = \frac{\|Au_i\|}{\lambda_n} \leq \frac{\lambda_n}{\lambda_{n-1}} < 1.$$

□

Im Gegensatz zur Potenzmethode können wir mit der Inversen Vektoriteration die Eigenvektoren zu allen Eigenwerten bestimmen. Hierzu ersetzen wir die Matrix A in der Iterationsvorschrift durch $(A - \mu I)$. Ist μ eine Näherung an den Eigenwert λ_i, d.h.

$$|\lambda_j - \mu| > |\lambda_i - \mu|, \quad \text{für alle } i \neq j,$$

dann erhalten wir den folgenden Konvergenzfaktor:

$$\rho_I = \frac{|\lambda_i - \mu|}{\nu}, \qquad \text{mit } \nu = \min_{j \neq i} |\lambda_j - \mu|.$$

Vergleichen wir den Aufwand der beiden Iterationen, so scheint die Potenzmethode, die pro Iteration im wesentlichen mit einer Matrix-Vektor-Multiplikation auskommt, gegenüber der Inversen Iteration, die in jedem Iterationsschritt das Lösen eines linearen Gleichungssystems erfordert, im Vorteil zu sein. Handelt es sich aber um eine dünnbesetzte Matrix, so können wir das Gleichungssystem ebenfalls mit $\mathcal{O}(n)$ Operationen lösen. An dem Konvergenzfaktor ρ_I erkennen wir, daß die Inverse Vektoriteration umso schneller konvergiert, je besser die Näherung μ den Eigenwert λ_i approximiert. Wir haben damit die Möglichkeit, die Konvergenz des Verfahrens relativ unabhängig vom Problem (der Matrix) zu steuern. Diese Möglichkeit ist bei der Potenzmethode nicht gegeben. Die Zahl ν ist der ausschließlich durch das Problem bestimmte Anteil am Konvergenzfaktor. Wenn ν sehr klein wird, brauchen wir sehr gute Näherungen μ, um den Eigenwert λ_i berechnen zu können. Die Zahl ν mißt den Abstand zum nächstgelegenen Eigenwert zu λ_i; wir können also festhalten, daß die Inverse Vektoriteration umso schneller konvergiert, je besser die Eigenwerte separiert sind.

Bevor wir den Zusammenhang zwischen QR-Folge und Vektoriteration erkennen können, brauchen wir noch die folgenden Hilfssätze:

Lemma 2.4.3 *Sei* (A_i, μ_i) $\; i = 1, 2, \ldots$ *eine QR-Folge zur symmetrischen Matrix* A *mit den Shiftparametern* μ_i*, dann gilt:*

i) Für alle $i = 1, 2, \cdots$ *gilt:* A_i *und* A *sind ähnliche Matrizen.*

ii) Alle A_i *sind symmetrisch, und falls* A *tridiagonal ist, dann sind es auch alle* A_i*.*

iii) Sei $L_i = Q_1 \cdots Q_i, \qquad M_i = R_i \cdots R_1,$*, dann ist*

$$L_i A_{i+1} = A L_i.$$

BEWEIS :

i)

$$\begin{aligned} A_{i+1} &= R_i Q_i + \mu_i I \\ &= Q_i^T Q_i R_i Q_i + \mu_i I \\ &= Q_i^T (A_i - \mu_i) Q_i + \mu_i I \\ &= Q_i^T A_i Q_i \end{aligned}$$

ii) Sei ohne Beschränkung $\mu_i = 0$, $i = 1, 2, \ldots$, dann folgt folgt für $A_{i+1} = R_i Q_i$ aus

$$A_{i+1}^T = A_{i+1}^T Q_i^T Q_i = Q_i^T R_i^T Q_i^T Q_i = Q_i^T A_i Q_i = A_{i+1}$$

die erste Behauptung. Betrachten wir nun die QR-Folge T_i, $i = 1, 2, \ldots$, zu einer Tridiagonalmatrix $T = T_1$. Die folgende schematische Darstellung veranschaulicht den Schritt von T_i nach T_{i+1}, wenn die QR-Zerlegung mit Givensrotationen realisiert wird:

$$Q_i^T T_i = R_i = \begin{bmatrix} * & * & * & & & \\ & * & * & * & & \\ & & \ddots & \ddots & \ddots & \\ & & & * & * & * \\ & & & & * & * \\ & & & & & * \end{bmatrix}.$$

Dabei ist Q_i das Produkt von $n-1$ Givensrotationen $Q_i = \Omega_{n-1,n} \cdot \ldots \cdot \Omega_{1,2}$. T_{i+1} hat dann die folgende Form:

$$T_{i+1} = R_i Q_i = \begin{bmatrix} * & * & \oplus & & & \\ + & * & * & \oplus & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & + & * & * & \oplus \\ & & & + & * & * \\ & & & & + & * \end{bmatrix}.$$

Da T_{i+1} symmetrisch ist, sind die Einträge $\oplus$ Nullen, und es folgt die zweite Behauptung.

iii) Nach (i) ist

$$\begin{aligned} A_{k+1} &= Q_k^T \cdots Q_1^T \, A \, Q_1 \cdots Q_k \\ &= L_k^T A L_k \end{aligned}$$

$$\Rightarrow L_k A_{k+1} = A L_k$$

$\square$

Satz 2.4.4 *Sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische Matrix mit den Eigenpaaren (v_i, λ_i), $\lambda_1 > \ldots > \lambda_n$, und A_i eine QR-Folge von A zu den Shiftparametern μ_i. Die Vektorfolgen x_i zur Potenzmethode und z_i zur Inversen Vektoriteration zu den gleichen Shiftparametern μ_i sei gegeben durch*

$$\gamma_i x_{i+1} = (A - \mu_i I) x_i \quad \|x_i\| = 1 \textit{ für } i = 1, 2, \ldots$$

$$(A - \mu_i I) z_{i+1} = \tau_i z_i \quad \|z_i\| = 1 \textit{ für } i = 1, 2, \ldots$$

Ist kein Shiftparameter Eigenwert von A und $x_1 = e_1, \quad z_1 = e_n$, dann gilt für $i \geq 1$ mit den Bezeichnungen aus (2.4.3):

$$x_i = L_{i-1}e_1$$

$$z_i = L_{i-1}e_n.$$

Beweis : [Parlett, 1980, S. 144] Es ist

$$\begin{array}{lccc} & A_ie_1 & = & L_{i-1}^T A L_{i-1}e_1 \\ \Leftrightarrow & L_{i-1}A_ie_1 & = & AL_{i-1}e_1 \\ \Leftrightarrow & L_{i-1}(Q_iR_i + \mu_i I)e_1 & = & L_{i-1}A_ie_1 \\ \Leftrightarrow & L_{i-1}Q_iR_ie_1 & = & L_{i-1}(A_i - \mu_i I)e_1 \\ \Leftrightarrow & L_iR_ie_1 & = & (A - \mu_i I)L_{i-1}e_1. \end{array}$$

Da R_i eine rechte obere Dreiecksmatrix ist, gilt andererseits:

$$L_i(R_ie_1) = L_i(r_{1,1}^i, 0, \ldots, 0)^T = r_{1,1}^i L_ie_1.$$

Wir setzen die beiden Ausdrücke für $L_iR_ie_1$ gleich:

$$r_{1,1}^i L_ie_1 = (A - \mu_i I)L_{i-1}e_1$$

und definieren nun $x_1 = L_0e_1 = e_1, \quad x_i = L_{i-1}e_1$ und $\tau_i = r_{1,1}^i$ Dann bilden die $x_i, i = 1, 2, \ldots$, gerade die Vektorfolge zur Potenzmethode mit Shift μ_i:

$$\tau_i x_i = (A - \mu_i I)x_{i-1}$$

und

$$\|x_i\| = \|L_{i-1}e_1\| = \|e_1\| = 1.$$

Zum Beweis der zweiten Behauptung transponieren wir die Gleichung (3) und multiplizieren von links und rechts L_{i-1} und L_i:

$$\begin{array}{rcl} L_{i-1} \cdot | \quad R_i^T L_i^T & = & L_{i-1}^T (A - \mu_i I)^T \quad | \cdot L_i \\ L_{i-1}R_i^T & = & (A - \mu_i I)L_i \quad | \cdot e_n \\ r_{n,n}L_{i-1}e_n & = & (A - \mu_i I)L_ie_n. \end{array}$$

Dann erhalten wir für $z_1 = L_0e_n = e_n$ und $z_i = L_ie_n$ die Inverse Vektoriteration zum Startwert e_n.

□

Satz 2.4.5 *Sei $A \in \mathbb{R}^{n \times n}$ eine symmetrische Matrix mit QR-Folge A_i zu den Shiftparametern $\mu_i = 0$, $i = 1, 2, \ldots$. (v_i, λ_i) seien die Eigenpaare von A mit $\lambda_1 > \ldots > \lambda_n$. Dann gilt*

$$\lim_{k \to \infty} A_i = diag(\lambda_1, \ldots, \lambda_n)$$

Beweis : [Parlett, 1980] Sei x_i, $i = 1, 2, \ldots$ die Vektorfolge zur Potenzmethode ohne Shift zum Startwert $x_1 = e_1$. Nach (2.4.2) gilt

$$x_i = v_1 + \mathcal{O}(\left(\frac{\lambda_2}{\lambda_1}\right)^i)$$

Mit den Bezeichnungen aus (2.4.2) erhalten wir

$$\begin{aligned} e_1 &= L_{i-1}(L_{i-1}^T e_1) \\ &= L_{i-1}x_i \\ &= L_{i-1}v_1 + \mathcal{O}(\left(\frac{\lambda_2}{\lambda_1}\right)^i), \end{aligned}$$

und mit (2.4.4) ergibt sich:

$$\begin{aligned} A_i e_1 &= L_{i-1}^T A L_{i-1} e_1 \\ &= L_{i-1}^T A x_i \\ &= L_{i-1}^T A(v_1 + \mathcal{O}(\left(\frac{\lambda_2}{\lambda_1}\right)^i)) \\ &= \lambda_1 L_{i-1} v_1 + \mathcal{O}(\left(\frac{\lambda_2}{\lambda_1}\right)^i) \\ &= \lambda_1 e_1 + \mathcal{O}(\left(\frac{\lambda_2}{\lambda_1}\right)^i). \end{aligned}$$

Analog erhalten wir über die Vektorfolge zur Inversen Vektoriteration

$$A_i e_n = \lambda_n e_n + \mathcal{O}(\left(\frac{\lambda_n}{\lambda_{n-1}}\right)^i).$$

Sei nun $\bar{A}$ der Grenzwert der Folge (A_i), dann ist die letzte Spalte von $\bar{A}$ gerade $\lambda_n e_n$. Da alle A_i, $i = 1, 2, \ldots,$ symmetrisch sind, können wir (2.4.4) auf die Untermatrix von $\bar{A}$ anwenden, die durch Streichen der letzten Spalte und der letzten Zeile entsteht. Die Behauptung folgt dann induktiv.

□

2.5 Praktische Durchführung der QR-Iteration

Das QR-Verfahren zeichnet sich besonders durch seine Stabilität aus. Betrachten wir allerdings den Aufwand - eine QR-Zerlegung pro Iterationsschritt - so scheint die reale Verwendbarkeit des Verfahrens in Frage gestellt, schließlich bedeutet die Berechnung der QR-Zerlegung einer $n \times n$ Matrix einen Aufwand von $\mathcal{O}(n^3)$ Operationen. Dieser Nachteil läßt sich aber dadurch kompensieren, daß wir die Matrix mittels Householdertransformationen auf Tridiagonalgestalt transformieren. Eine QR-Zerlegung einer Tridiagonalmatrix kann dann mittels Givensrotationen mit $\mathcal{O}(n)$ Operationen berechnet werden. Ein weiterer Nachteil besteht in der nur linearen Konvergenz des Verfahrens. Im vorigen Abschnitt wurde jedoch eine Möglichkeit vorgestellt, Abhilfe zu schaffen: Durch Verwendung geeigneter Shiftstrategien kann sogar eine kubische Konvergenzrate erreicht werden. Bevor wir uns der Wahl von Shiftparametern zuwenden, sollten noch einige Besonderheiten von Tridiagonalmatrizen vorgestellt werden, die zeigen, daß die bisherige Einschränkung auf Matrizen mit verschiedenen Eigenwerten keine Begrenzung der Verwendbarkeit des QR-Algorithmus darstellt. Tatsächlich konvergiert die QR-Folge für Matrizen mit mehrfachen Eigenwerten nicht gegen eine Diagonalmatrix, sondern gegen eine Blockdiagonalmatrix, deren Blockgrößen den Vielfachheiten der zugehörigen Eigenwerte entsprechen. Im Falle mehrfacher Eigenwerte kann aber nachgewiesen werden [Parlett 1980, S.124], daß es mindestens ein Nebendiagonalelement $f_i = 0$ gibt. Ist k der maximale Index mit $f_k = 0$, dann zerfällt unser Eigenwertproblem in zwei Teilprobleme:

$$T - \lambda I = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} - \lambda I.$$

Wir können die Eigenwerte von T unabhängig als die Eigenwerte von T_1 und T_2 berechnen. Bei der praktischen Umsetzung geht man folgendermaßen vor: Erst wird die Nebendiagonale auf Nulleinträge hin durchsucht. Hier stellt sich die Frage, welche Zahlen wir als Null interpretieren, wenn wir mit endlicher Genauigkeit rechnen. Da die Einträge der Tridiagonalmatrix durch die Householderprozedur bereits um ein geringes Vielfaches c der Rechengenauigkeit gestört sind, können wir zwei Einträge x und y nicht mehr unterscheiden, wenn $\frac{|x-y|}{|x|} < c\, eps$. Es ist daher sinvoll, Einträge in der Größenordnung der Rechengenauigkeit als Null zu interpretieren. In der EISPACK-Bibliothek wird ein Nebendiagonalelement mit den benachbarten Diagonalelementen verglichen. Sei

$$T = \begin{bmatrix} d_1 & f_1 & & & \\ f_1 & d_2 & f_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & f_{n-1} \\ & & & f_{n-1} & d_n \end{bmatrix}.$$

Wenn für ein Nebendiagonalelement f_i gilt

$$|f_i| < eps(|d_i| + |d_{i+1}|),$$

dann wird f_i auf Null gesetzt. Ein solches Vorgehen ist auch durch die Praxis des QR-Verfahrens dringend geboten, denn im Fall sehr kleiner Einträge droht Exponentenunterlauf. Bowlder, Martin, Reinsch und Wilkinson [Wilkinson, Reinsch 1971] gingen weiterhin folgendermaßen vor: Es seien bereits l Eigenwerte konvergiert, und $k < n - l$ sei der maximale Index, für den $f_k = 0$, dann zerfällt T in drei Untermatrizen

$$T = \begin{bmatrix} T_1 & & \\ & T_2 & \\ & & \Lambda_l \end{bmatrix}$$

mit $\Lambda_l = diag(\lambda_{n-l+1}, \ldots, \lambda_n)$. Der eigentliche QR-Algorithmus wird nun auf T_2 angewandt, und nachdem Konvergenz eingetreten ist, wird die gesamte Prozedur rekursiv auf die Matrix T_1 angewandt. Hierbei wird ein Diagonalelement d_i als "konvergiert"betrachtet, wenn gilt $|f_{i-1}| < eps(|d_{i-1}| + |d_i|)$.
Zur Vereinfachung der Notationen nehmen wir wieder an, daß $T \in \mathbb{R}^{n \times n}$ eine Tridiagonalmatrix ist mit den Eigenwerten $\lambda_1 > \ldots > \lambda_n$. Nach (2.4.5) konvergiert die letzte Spalte von T beim QR-Verfahren wie die Vektorfolge bei der Inversen Vektoriteration. Bei der Wahl der Shifts ist es demnach sinnvoll, Näherungen an λ_n einzusetzen. Da die letzte Spalte von T gegen $\lambda_n e_n$ konvergiert, ist die naheliegendste Möglichkeit, als Shiftparameter μ_k im k-ten Schritt der Iteration einfach $\mu_k = T_{n,n}$ einzusetzen. Im Falle kaum separierter Eigenwerte ist die als Wilkinson-Shift bekannte Strategie vorteilhafter: Ist $\bar{\mu}$ der Eigenwert der Matrix

$$\begin{bmatrix} d_{n-1} & f_{n-1} \\ f_{n-1} & d_n \end{bmatrix},$$

der näher an d_n liegt, dann setzte $\mu_k = \bar{\mu}$. Für beide Strategien läßt sich im Fall symmetrischer Matrizen kubische Konvergenz nachweisen [Wilkinson 1965, S. 548]. Da ein explizites Addieren von $-\mu$ auf die Hauptdiagonale ernsthafte Genauigkeitsverluste mit sich bringen kann, hat sich eine implizite Shiftstrategie als vorteilhaft erwiesen, die auf dem folgenden Satz beruht:

Satz 2.5.1 *Gegeben sei eine symmetrische Tridiagonalmatrix T mit Eigenwerten $\lambda_1 > \ldots > \lambda_n$. Dann ist die QR-Folge zu T durch die jeweils ersten Spalten von Q_i, $i \geq 1$, bis auf Vorzeichen eindeutig bestimmt, d.h.*
wenn Q und J orthogonale Matrizen sind, so daß $H = Q^T T Q$ und $G = J^T T J$ wieder Tridiagonalmatrizen sind und die ersten Spalten von Q und J bis auf einen Faktor $\nu = \pm 1$ übereinstimmen, dann existieren ν_i, $1 \leq i \leq n$ mit $|\nu_i| = 1$, so daß

$$Q = diag(\nu_1, \ldots, \nu_n) J.$$

Beweis : [nach Golub, Van Loan 1983, S. 223] Sei $W := F^T Q$. Dann gilt

$$GW = GJ^T Q = J^T T Q = J^T Q Q^T T Q = WH$$

Für die i-te Spalte w_i von W gilt demnach

$$h_{i,i-1} w_i = G w_{i-1} - \sum_{j=1}^{i-1} h_{j,i-1} w_j. \tag{2.7}$$

Nehmen wir der Einfachheit halber $q_1 = f_1$ an, dann ist:

$$w_1 = J^T q_1 = e_1,$$

und induktiv ergibt sich aus (2.7), daß W eine obere Dreiecksmatrix ist. Da aber W orthogonal ist, muß gelten $W = diag(\pm 1)$.

□

Wenn wir nun zu

$$\begin{aligned} T_i - \mu I &= QR \\ T_{i+1} &:= RQ + \mu I \end{aligned}$$

eine Familie orthogonaler Matrizen $J_1, \ldots, J_{n-1}$ finden mit

$$Z := J_1 \cdot \ldots \cdot J_{n-1}, \;\; Ze_1 = J_1 e_1 = Qe_1,$$

und $Z^T T Z$ ist wieder tridiagonal, dann ist $Z^T T Z$ Nachfolger von T in einer QR-Folge.

Bei der impliziten Shift-Stategie geht man so vor, daß erst eine Givensrotation

$$J_1^T = \Omega_{2,1} = \begin{bmatrix} c & c & \\ -s & c & \\ & & I \end{bmatrix}$$

derart bestimmt wird, daß

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{pmatrix} d_1 - \mu \\ f_1 \end{pmatrix} = \begin{pmatrix} * \\ 0 \end{pmatrix}. \tag{2.8}$$

Nach Anwendung von J_1^T von links und J_1 von rechts erhalten wir eine Matrix der Form:

$$J_1^T T J_1 = \begin{bmatrix} * & * & + & & & \\ * & * & * & & & \\ + & * & * & * & & \\ & & * & * & * & \\ & & & * & * & * \\ & & & & * & * \end{bmatrix},$$

wobei an den mit + markierten Stellen Einträge verschieden von Null entstanden sind. Konstruieren wir J_2 als Givensrotation, die diesen Eintrag auf Null abbildet, aber die erste Spalte bzw. Zeile von $J_1^T T J_1$ unverändert läßt, dann kann durch sukzessives Wiederholen dieser Prozedur der neu entstandene Eintrag entlang der Diagonale in die rechte untere Ecke "gejagt"werden, bis die Matrix wieder Tridiagonalgestalt angenommen hat. Damit haben wir dann einen QR-Schritt mit Shift μ auf der Matrix T ausgeführt.

Der QR-Algorithmus wurde 1965 von Golub und Kahan zur Berechnung der Singulärwerte einer Bidiagonalmatrix vorgeschlagen. 1971 stellten Golub und Reinsch eine effiziente und robuste Algol-Implementierung vor, die bis heute nur geringfügige Änderungen erfahren hat: Wenn eine Bidiagonalmatrix $B \in \mathbb{R}^{n \times n}$ gegeben ist, dann ist $B^T B$ eine Tridiagonalmatrix, auf die der QR-Algorithmus angewandt werden könnte, um die Quadrate der Singulärwerte zu berechnen. Aber durch die explizite Berechnung des Matrixprodukts können wesentliche Informationen über die kleinen Singulärwerte verloren gehen. Führen wir einen Iterationsschritt des QR-Algorithmus auf der Matrix $B^T B$ aus, so zeigt

$$Q^T B^T B Q = (BQ)^T BQ,$$

daß es möglich ist, die Iteration allein auf B durchzuführen. Wir bestimmen wie in (2.8) die erste Givensrotation J_1 so, als wollten wir eine QR-Zerlegung von $B^T B - \mu I$ berechnen, und wenden J_1 auf B an:

$$BJ_1 = \begin{bmatrix} * & * & & & & \\ + & * & * & & & \\ & & * & * & & \\ & & & * & * & \\ & & & & * & * \\ & & & & & * \end{bmatrix}.$$

Mittels geeigneter Givensrotationen von links und rechts jagen wir dann den neu entstandenen Eintrag entlang der Diagonalen, bis die Matrix wieder Bidiagonalgestalt besitzt. Nach (2.5.1) haben wir dann einen Schritt des QR-Algorithmus auf der Matrix $B^T B$ mit Shift μ ausgeführt.

In der momentan aktuellsten Algebra Funktionensammlung LAPACK wird die implizite Null-Shift Strategie nach Demmel und Kahan [1990] eingesetzt, deren vorrangiges Ziel nicht die Steigerung der Effizienz gegenüber der Wilkinson-Shift-Strategie ist, sondern die Verbesserung der numerischen Eigenschaften. Denn bei Verwendung des impliziten Wilkinson-Shifts entspicht die erzielte Genauigkeit, insbesondere bei schlecht konditionierten Matrizen, nicht immer der theoretisch möglichen. Auch bei der Null-Shift Strategie wird ausgehend von einer Bidiagonalmatrix $B = B_0 \in \mathbb{R}^{n \times n}$ eine Folge von Matrizen $B_i, i \geq 1$, berechnet, die gegen eine Diagonalmatrix mit den Singulärwerten von B auf der Hauptdiagonalen konvergiert. Dabei berechnen sich B_{i+1} aus B_i durch abwechselndes Anwenden gewisser Givensrotationen:

$$B_{i+1} = J_{2n-1} \cdot \ldots \cdot J_2 \, B_i \, J_1 \cdot \ldots \cdot J_{2n-2}.$$

Zur Vereinfachung der Notation lassen wir nun den unteren Index von B weg. Die erste und nach (2.4.5) über den Shiftparameter entscheidende Rotation J_1 wird so gewählt, daß

$$B^{(1)} = BJ_1 = \begin{bmatrix} d_1^{(1)} & 0 & & \\ x & d_2^{(1)} & f_2 & \\ & & d_3 & f_3 \\ & & & d_4 \end{bmatrix} \qquad (n=4).$$

Mit der nächsten Rotation wird der neu entstandene Eintrag x wieder vernichtet:

$$B^{(2)} = J_2 B^{(1)} = \begin{bmatrix} d_1^{(2)} & f_1^{(2)} & y & \\ 0 & d_2^{(2)} & f_2^{(2)} & \\ & & d_3 & f_3 \\ & & & d_4 \end{bmatrix}. \tag{2.9}$$

Dabei hat der Block

$$\begin{bmatrix} f_1^{(2)} & y \\ d_2^{(2)} & f_2^{(2)} \end{bmatrix} = \begin{bmatrix} \sin\phi_2 d_2^{(1)} & \sin\phi_2 f_2 \\ \cos\phi_2 d_2^{(1)} & \cos\phi_2 f_2 \end{bmatrix}$$

den Rang eins. Multiplizieren wir eine Rotation J_3 von links, um den Eintrag y zu vernichten, dann wird daher auch der Eintrag $f_2^{(2)}$ verschwinden,

$$B^{(3)} = B^{(2)} J_3 = \begin{bmatrix} d_1^{(2)} & f_1^{(3)} & 0 & \\ 0 & d_2^{(3)} & 0 & \\ & & d_3^{(3)} & f_3 \\ & & & d_4 \end{bmatrix},$$

und wir können die Prozedur ab (2.9) auf die rechte Untermatrix der Dimension $n-1$ sukzessive anwenden, bis die Matrix wieder Bidiagonalgestalt besitzt. Damit haben wir einen Iterationsschritt des impliziten QR-Algorithmus auf B^TB bzw. BB^T mit dem Shiftparameter Null ausgeführt - daher der Name des Verfahrens. Die besseren numerischen Eigenschaften sind auf die vermehrt auftretenden Nullen zurückzuführen, die die Gesamtzahl der Operationen, insbesondere der Additionen, pro Iterationsschritt verringern.

2.6 Berechnung der Singulärwerte von Bidiagonalmatrizen mit Sturmschen Ketten.

Eine klassische Methode zur Bestimmung der Eigenwerte symmetrischer Tridiagonalmatrizen ist das Bisektionsverfahren nach Barth, Martin und Wilkinson [Wilkinson, Reinsch 1971]. Dieses Verfahren beruht auf dem Satz von Sturm:

Satz 2.6.1 *Gegeben sei ein Polynom $p_0(x), x \in \mathbb{R}$. Wir definieren $p_1(x) := p_0'(x)$, und die Polynome $p_2(x) \dots p_r(x)$ durch*

$$\begin{aligned} p_{i-1} &= q_i p_i - p_{i+1} && 1 \le i \le r-2 \\ p_{r-1} &= q_r p_r && (euklidischer Algorithmus). \end{aligned}$$

Für eine Zahl $\mu > 0$ sei $\omega(\mu)$ die Anzahl der Vorzeichenwechsel in der Kette $p_0(\mu), \ldots$ $\ldots, p_r(\mu)$, wobei Nullen ausgelassen werden. Dann ist $\omega(\mu)$ gleich der Anzahl der reellen Nullstellen von p_0 im Intervall $[0, \mu]$; mehrfache Nullstellen werden nur einfach gezählt. Die Folge $p_0, \ldots, p_r$ heißt Sturmsche Kette.

BEWEIS : zum Beispiel B.L. van der Waerden 1971, S. 248-250.

Diesen Satz kann man folgendermaßen auf das charakteristische Polynom einer symmetrischen Tridiagonalmatrix T mit Diagonalelementen $d_i, 1 \leq i \leq n$ und Nebendiagonalelementen $f_i, 1 \leq i \leq n-1$ anwenden: Zuerst wird T wie in (2.5) in unzerlegbare Teilmatrizen aufgespalten, um mehrfache Eigenwerte in den Teilmatrizen auszuschließen. Wir nehmen an, T sei bereits unzerlegbar, d.h $f_i \neq 0$ für $1 \leq i \leq n-1$.

Korollar 2.6.2 *Seien $p_0(x) = 1$, $p_1(x) = d_1 - x$, und*

$$p_i(x) = (d_i - x)p_{i-1}(x) - f_{i-1}^2 p_{i-2}(x), \quad 2 \leq i \leq n, \tag{2.10}$$

dann ist p_i, $i = n, \ldots, 0$ die Sturmsche Kette zu $p_n(x) = det(T - xI)$.

BEWEIS : Da das charakteristische Polynom $p(x)$ von T nur einfache Nullstellen besitzt, ist der größte gemeinsame Teiler von $p(x)$ und $p'(x)$ gerade eins. Die Behauptung sei für $n-1$ bereits bewiesen, dann liefert Entwickeln von T nach der letzten Spalte induktiv die Behauptung.

□

Die Dreitermrekursion (2.10) erlaubt es uns nun, die Sturmsche Kette des charakteristischen Polynoms von T zu berechnen, um so herauszufinden, wieviele Eigenwerte in einem Intervall $[0, \mu]$ liegen. Mit dem Bisektionsverfahren können wir dann die Sprungstellen von ω und damit die Eigenwerte von T berechnen. Bei der Auswertung der Dreitermrekursion in (2.10) droht bei wenig separierten Eigenwerten Unterlauf; es ist daher numerisch günstiger, die Folge

$$q_i(x) = \frac{p_i(x)}{p_{i-1}(x)}$$

auszuwerten. $\omega(x)$ ist dann gleich der Anzahl der negativen $q_i(x)$, die sich wie folgt berechnen lassen:

$$\begin{aligned} q_1(x) &= d_1 - x \\ q_i(x) &= d_i - x - \frac{f_{i-1}^2}{q_{i-1}(x)}, \qquad i = 2, 3, \ldots. \end{aligned}$$

Die Division scheint gefährlich zu sein, doch die Fehleranalyse zeigt, daß wir bei drohendem Überlauf q_{i-1} durch $|f_{i-1}|$ *eps* ersetzen dürfen. Dies entspricht nämlich der

Ersetzung von d_{i-1} durch $d_{i-1} + |f_{i-1}|eps$, also einer Änderung des Problems im Rahmen der Rechengenauigkeit. Aus dem gleichen Grund kann das Bisektionsverfahren mit oberer und unterer Schanke xo und xu abgebrochen werden, sobald

$$|xo - xu| < 2\, eps\, (|xo| + |xu|)$$

erfüllt ist. Der Vorteil dieses Verfahrens liegt in der hohen erreichbaren Genauigkeit, auch wenn die Eigenwerte wenig separiert sind. Insbesondere erlaubt es die Berechnung einzelner Eigenwerte, ohne gleich auch alle anderen berechnen zu müssen. Der Nachteil liegt in der nur linearen Konvergenz des Verfahrens. Es gibt jedoch Möglichkeiten, die Konvergenzordnung zu steigern: Differenzieren wir (2.10), dann erhalten wir explizit gegebene Dreitermrekursionen zur Bestimmung der Ableitungen des charakteristischen Polynoms, und können damit Verfahren höherer Ordung konstruieren. Sollen neben den Eigenwerten auch die zugehörigen Eigenvektoren bestimmt werden, so bietet sich eine Kombination der Bisektion mit der Inversen Vektoriteration und Rayleigh-Shift an. Hierzu werden mittels Bisektion Intervalle bestimmt, die jeweils nur einen Eigenwert enthalten. Sei $\lambda \in [xu, xo]$ und x_0 ein Startwert für die Inverse Vektoriteration. Parlett [1980, S154-157] hat für das folgende Verfahren globale Konvergenz der Ordnung 3 nachgewiesen.

$$\begin{array}{rcl}
\mu_0 & = & (x_u + xo)/2 \\
löse \quad (A - \mu_0 I)x_1 & = & x_0 \\
x_1 & := & x_1/\|x_1\| \\
für\ i = 1, 2, \ldots & & \\
\mu_i & = & x_i^T A x_i \\
löse \quad (A - \mu_i I)x_{i+1} & = & x_i \\
x_{i+1} & := & x_{i+1}/\|x_{i+1}\|
\end{array}$$

Nun soll einer der genannten Algorithmen zur Berechnung der Singulärwerte einer Bidiagonalmatrix

$$B = \begin{bmatrix} d_1 & f_1 & & \\ & \ddots & \ddots & \\ & & d_{n-1} & f_{n-1} \\ & & & d_n \end{bmatrix}$$

eingesetzt werden. Nach Satz (2.1.6) haben wir zwei Möglichkeiten die Singulärwerte von B zu berechnen:

P1) Als die Eigenwerte von $B^T B$ bzw. BB^T

P2) Als die positiven Eigenwerte von

$$\tilde{B} = \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix}$$

Um den Bisektionsalgorithmus auf das Problem P2 anwenden zu können, bedienen wir uns einer Permutationsmatrix $P \in \mathbb{R}^{2m \times 2n}$, die die Zeilen bzw. Spalten von $\tilde{B}$ so umordnet, daß

$$P\tilde{B}P^T = \begin{bmatrix} 0 & d_1 & & & & & \\ d_1 & 0 & f_1 & & & & \\ & f_1 & 0 & d_2 & & & \\ & & d_2 & 0 & f_2 & & \\ & & & f_2 & \ddots & \ddots & \\ & & & & \ddots & & d_n \\ & & & & & d_n & 0 \end{bmatrix}.$$

Die Auswertung der Sturmschen Kette vereinfacht sich zwar ein wenig durch die Tatsache, daß $d_i = 0$ für $1 \leq i \leq 2n$; trotzdem haben wir im Vergleich zu P1 fast den doppelten Aufwand. Aus numerischer Perspektive ist der Zugang über P2 jedoch günstiger. Li, Rhee und Zeng [1995] haben die folgenden Fehlerschranken angegeben:

P1) Seien $\sigma_1 > \ldots > \sigma_n$ die Singulärwerte von B und $\bar{\sigma_1} > \ldots > \bar{\sigma_n}$ die Wurzeln der Nullstellen von

$$p(x) = fl(\ det(\ fl(B^T B)\ -xI)\ ),$$

wobei die Determinante über die Sturmsche Kette ausgewertet wird. Dann gilt für den relativen Fehler

$$\frac{|\sigma_i - \bar{\sigma_i}|}{\sigma_i} \leq \frac{7}{4} \frac{\|B^T B\|_\infty}{\sigma_i^2}\, eps + \mathcal{O}(eps^2)$$

P2) Sind $\bar{\sigma_i}$ nun die mit dem Bisektionsverfahren, angewandt auf $\tilde{B}$, berechneten positiven Eigenwerte, dann gilt

$$\delta^{-1} \leq \frac{\bar{\sigma_i}}{\sigma_i} \leq \delta, \quad \text{mit}$$

$$\delta \leq 1 + 1.5(2n-1)eps + \mathcal{O}(eps^2).$$

Entscheidend ist hier, daß die Schranke im Fall P2, unabhängig vom zu berechnenden Singulärwert, sehr klein ist, während beim Zugang über P1 die Fehlerschranke das Quadrat der Singulärwerte im Nenner hat, und daher zur Berechnung der kleinen Singulärwerte ungeeignet ist. Bei großen Matrizen hingegen kann die Berechnung der großen Singulärwerte über P1 vorteilhafter sein. Li, Rhee und Zeng schlagen aus diesem Grund - aber vor allem aus Effizienzgründen - einen hybriden Algorithmus vor, bei dem die Singulärwerte σ_i mit

$$\sigma_i \geq \left(\frac{7}{12n}\|B^T B\|_\infty\right)^{1/2}$$

nach P1 berechnet werden und sonst der Zugang über P2 gewählt wird.

2.7 Die Inverse Vektoriteration

Wie wir im letzten Abschnitt sehen konnten, ermöglicht der Satz von Sturm, Verfahren zu konstruieren, mit denen die Eigenwerte symmetrischer Tridiagonalmatrizen $T \in \mathbb{R}^{n\times n}$ bis auf einen relativen Fehler in der Größenordnung der Rechengenauigkeit berechnet werden können. Die Inverse Vektoriteration ist das gängige Verfahren die zugehörigen Eigenvektoren zu bestimmen. Sei μ eine sehr gute Näherung an einen Eigenwert λ_i von T. Dann sieht ein Schritt der Inversen Vektoriteration zum Startwert x_0 folgendermaßen aus:

$$\begin{array}{ll} \text{löse das Gleichungssystem} & (T-\mu I)y_i = x_i \\ \text{und setze} & x_{i+1} = \dfrac{y_i}{\|y_i\|} \end{array} .$$

Hierbei ist das Gleichungssytem beliebig schlecht konditioniert, je nachdem, wie gut der Näherungswert μ ist; es stellt sich daher die Frage, ob die Lösung überhaupt noch sinnvolle Information enthält. Wenn wir einen direkten Löser verwenden wollen, was aus Effizienzgründen geraten ist, besteht darüber hinaus die Gefahr, daß das gewählte Verfahren wegen Über- oder Unterlauf zusammenbricht. Dieser Gefahr läßt sich ähnlich wie beim QR-Algorithmus dadurch begegnen, daß bei "gefährlichenÖperationen Matrixeinträge gegebenenfalls um geeignete Vielfache der Rechengenauigkeit geändert werden, um einen Zusammenbruch zu vermeiden. Ohne solche Sicherheitsmaßnahmen wäre das Programm praktisch nicht lauffähig. Betrachten wir nun den Fehler beim Lösen des Gleichungssystems $(T-\mu I)y_i = x_i$. Wir nehmen an, daß aufgrund von Rundungsfehlern y die folgende Gleichung erfülle [Parlett 1980, S. 65]:

$$(T-\mu I-H)y = x+f.$$

Von einem guten Löser erwarten wir, daß

$$\|H\| \ll \|T-\mu I\| \text{ und } \|f\| \ll \|x\| = 1.$$

Zur Analyse des Fehlers schreiben wir

$$(T-\mu I)y = x+g$$

mit $g = f + Hy$. Der Gesamtfehler ist dann

$$e = (T-\mu I)^{-1}g.$$

Wenn $|\mu-\lambda_i|$ hinreichend klein ist, und v der Eigenvektor zu λ_i, dann können wir eine orthogonale Zerlegung von g angeben:

$$g = (v\cos\phi + u\sin\phi)\|g\|$$

mit $\phi = \angle(g,v)$. Eine Zerlegung von e erhalten wir durch

$$\begin{aligned} e &= (T-\mu I)^{-1}g \\ &= (v\,\frac{\cos\phi}{\lambda_i-\mu} + \bar{u}\,\|(T-\mu I)^{-1}u\|\sin\phi)\quad\|g\| \end{aligned}$$

mit
$$\bar{u} = \frac{(T-\mu I)^{-1}u}{\|(T-\mu I)^{-1}u\|};$$
$\bar{u}$ ist damit wieder orthogonal zu v. Sei $\eta = \angle(v, e)$. Dann gilt
$$\begin{aligned}\tan\eta &= \|(T-\mu I)^{-1}u\|(\lambda_i - \mu)\tan\phi \\ &\leq \frac{|\lambda_i - \mu|}{\min_{j\neq i}|\lambda_j - \mu|}\tan\phi.\end{aligned}$$

Unter der Voraussetzung, daß $\tan\phi$ nicht allzugroß ist, sehen wir, daß der Fehlerwinkel η umso rascher konvergiert, je besser die Näherung μ den Eigenwert λ_i approximiert. Dabei kann die Norm des Fehlers fast beliebig groß werden, während die Richtung des verfälschten Lösungsvektors sehr gut mit der Richtung der exakten Lösung übereinstimmt. Durch die Normierung $x_{i+1} = \frac{y_i}{\|y_i\|}$ wird der Fehler in der Norm von y_i irrelevant. In der Praxis läßt sich beobachten, daß die Inverse Vektoriteration bereits nach ein bis zwei Iterationen hinreichend gute Ergebnisse liefert. Zur Praxis des Verfahrens ist weiterhin anzumerken, daß im allgemeinen die Forderung, daß der Startwert nicht orthogonal zum gesuchten Eigenvektor stehen darf, keine Probleme macht; dafür sorgen die Rundungsfehler, die hier ausnahmsweise eine positive Rolle übernehmen. Problematisch bleibt allerdings der Fall kaum separierter Eigenwerte. Wenn Näherungen für benachbarte Eigenwerte bereitstehen, werden in der LAPACK-Bibliothek die gegebenen Näherungen künstlich separiert, und zwar so, daß sich benachbarte Näherungen mindestens um $10|\lambda|eps$ unterscheiden. Erst nach dieser Vorbereitung wird die Inverse Vektoriteration gestartet. Trotzdem läßt es sich nicht ausschließen, daß die Iteration nicht konvergiert. Bei meinen Experimenten mit den entspechenden LAPACK-Routinen ist dieser Fall allerdings nie aufgetreten.
Offen bleibt noch die Frage nach einem Abbruchkriterium. Hierzu betrachten wir nun das Residuum im k-ten Schritt [vgl. Peters und Wilkinson 1979]:
$$r = Tx_k - \mu x_k = (Ty_{k-1} - \mu y_{k-1})\,\frac{1}{\|y_{k-1}\|}$$
Da $\|x_{k-1}\| = 1$, ist das Residuum r umso kleiner, je größer $\|y_{k-1}\|$ ist. Wenn $|\mu - \lambda_i| \cong eps\|T\|$ und
$$T - \mu I = V\bar{\Lambda}V^T, \quad \bar{\Lambda} = diag(\bar{\lambda_1}, \ldots, \bar{\lambda_n})$$
die Singulärwertzerlegung der symmetrischen Matrix $T - \mu I$ ist, und $x_0 = \sum_{i=1}^n \alpha_i v_i$ Startvektor der Inversen Vektoriteration, dann ergibt sich aus
$$(T-\mu I)^{-1}(\sum_{i=1}^n \alpha_i v_i) = \sum_{i=1}^n \frac{\alpha_i}{\bar{\lambda_i}} v_i,$$
daß die nächste Iterierte fast keine Anteile von v_j für $1 \leq j \leq n-1$ mehr besitzt, denn
$$\bar{\lambda_n} \cong eps\bar{\lambda_1}.$$

Mehr als eine Iteration ist daher nur dann nötig, wenn der Startvektor fast keine Anteile in Richtung von v_n besitzt. Billige, praxiserprobte Abbruchkriterien sind:

$$\begin{array}{ll} \|y\|_\infty > n^{-1/2} & \text{(EISPACK)} \\ \|y\|_\infty > 0.1 \cdot n^{-1/2} & \text{(LAPACK)} \end{array}$$

Zusammenfassend läßt sich über die Inverse Vektoriteration sagen, daß sie sehr effizient, aber relativ instabil ist. Werden aber die genannten Sicherheitsvorkehrungen getroffen, dann ist eine alltagstaugliche Implementierung möglich.

Kapitel 3

Parallelrechnerarchitekturen

Die Laufzeit eines Programmes kann auf unterschiedlichen Rechnern gleicher Maximalleistung durchaus sehr verschieden sein, je nachdem wie gut ein Algorithmus zu den technischen Gegebenheiten (Architektur) paßt. Dabei lassen sich verschiedene Typen von Computerarchitekturen ausmachen, auf denen jeweils andere Strategien nötig sind, um eine hohe Effizienz zu erreichen. Mit der Bezeichnung „modern“ fasse ich die heute typischen Architekturen zusammen, die sich von ihren Vorgängern vor allem in der Speicherstruktur unterscheiden. Der Begriff „serieller“ Rechner ist heute kaum noch zutreffend: Prozessoren der letzten Generation verfügen bereits über vielfältige Möglichkeiten, gewisse Abläufe intern parallel abzuarbeiten - auch hier sprechen wir von einem Architekturmerkmal. Doch statt umständlich von Ein- bzw. Mehr-Prozessor-Rechnern zu reden, wollen wir die Bezeichnungen serieller bzw. Parallelrechner beibehalten. Zur Klärung der wichtigsten Begriffe des „Computer-Chinesisch“ mag die folgende schematische Darstellung dienen, die veranschaulicht, was in einem Computer vor sich geht, wenn er zur Bearbeitung mathematischer Aufgaben eingesetzt wird, und welche Architekturmerkmale für numerische Anwendungen relevant sind. Daraus werden Forderungen abgeleitet, die ein Algorithmus erfüllen muß, wenn die Ressourcen des Rechners optimal genutzt werden sollen. Bei der anschließenden Vorstellung zweier Parallelrechner stellt sich heraus, daß für diesen Rechnertyp die gleichen Forderungen relevant sind wie bei seriellen Rechnern.

3.1 Funktionsweise serieller Rechenanlagen

Technisch gesehen, können wir einen Computer in zwei Teilbereiche einteilen: Die Zentraleinheit (CPU, Central Processing Unit) und die Peripherie. Die Zentraleinheit besteht in der Regel aus einem Chip, auf dem verschiedene elektronische Funktionseinheiten zum Verarbeiten (Addieren, Multiplizieren,...) von Daten (Zahlen, ...) untergebracht sind. Die Daten werden diesen Funktionseinheiten in sogenannten Registern

bereitgestellt, von denen nur eine begrenzte Zahl (üblich sind 32) auf der CPU untergebracht werden können. Komplexe Anwendungen verarbeiten mehrere Millionen Zahlen, die in der Peripherie abgelegt werden müssen und im Bedarfsfall zur Zentraleinheit transportiert werden. Wesentlicher Bestandteil der Peripherie ist der (Haupt-) Speicher: In einer Vielzahl von Chips sind elektronische Schaltungen implementiert, die die Fähigkeit besitzen, Daten zu speichern und auszugeben. Damit im Speicher abgelegte Daten wiedergefunden werden können, werden Informationseinheiten fester Größe im Speicher mit fortlaufenden Nummern (Adressen) versehen. Die Verbindung zwischen CPU und Speicher wird über den sogenannten Systembus hergestellt. Technisch gesehen ist ein Bus eine Ansammlung von n (16-256) Datenleitungen, auf denen die Daten in binärer Darstellung übermittelt werden. Hierzu werden die Leitungen für eine feste Zeiteinheit - den Bustakt - mit Strom (1) oder Nicht-Strom (0) belegt, je nach dem binären Muster, das übertragen werden soll. Auf die gleiche Weise werden auch die Daten innerhalb der CPU transportiert. Die CPU besitzt ihren eigenen, vom Bustakt unabhängigen Taktgeber, dessen Frequenz oft um ein Vielfaches höher liegt als der Bustakt. Wenn viele Daten über den Bus zur CPU transportiert werden müssen, kann es daher zu unerwünschten Wartezeiten kommen, die die Gesamtleistung des Systems drastisch mindern.
Weiterer Bestandteil der Peripherie sind Festplatten. Auch sie dienen der Datenaufbewahrung. Sie haben gegenüber dem Hauptspeicher den Vorteil, daß die gespeicherten Daten erhalten bleiben, auch wenn der Computer ausgeschaltet wird. Außerdem übertrifft die Speicherkapazität von Festplatten die des Hauptspeicher um ein Vielfaches. Die Anzahl der pro Zeiteinheit übertragbaren Daten ist bei Festplatten aber viel geringer als beim Hauptspeicher (Vergleiche Abbildung auf Seite 46).
Werden wir nun etwas konkreter: Nehmen wir als Beispiel das einfache Problem:

$$c = a + b$$

Wir interpretieren a und b als Adreßnummern im Speicher, an denen bereits Zahlenwerte abgelegt seien. Auf Maschinenebene sieht ein Programm zur Berechnung der Summe etwa wie folgt aus:

- Lade den Inhalt der Speicherstelle a in Register 1 (R1)
- Lade den Inhalt der Speicherstelle b in R2
- Berechne R1+R2 und speichere das Ergebnis in R3
- Kopiere den Inhalt von R3 in die Speicherstelle c

Die eigentliche Addition ist also nur ein Teilschritt der gesamten Operation. Entscheidend für die Ablaufgeschwindigkeit ist daher nicht nur die Addition. Die Lade- und Speicheranweisungen sind unter Umständen aufwendiger als die Rechenoperation. Wenn sich die Daten bereits in den Registern der CPU befinden und das Ergebnis nicht in den Hauptspeicher kopiert werden muß, weil es zum Beispiel für die nächste

Abbildung 3.1: Modell eines Computers mit hierarchischer Speicherarchitektur

Operation benötigt wird, dann entfällt ein Großteil der Arbeit. Nur in diesem Idealfall erreichen moderne Computer ihre Maximalleistung. Ein effizienter Algorithmus, der eine große Datenmenge verarbeitet, sollte daher so angelegt sein, daß die Daten möglichst selten zwischen CPU und Hauptspeicher transportiert werden müssen.
Ein etwas komplexeres Beispiel ist die Multiplikation $y = Ax$ einer Matrix $A = a_{i,j} \in \mathbb{R}^{n \times n}$ mit einem Vektor $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$. In einer höheren Programmiersprache können wir leicht einen Algorithmus formulieren:

$$
\begin{array}{l}
for \quad i = 1 \quad to \quad n \\
\quad y_i = 0 \\
\quad for \quad j = 1 \quad to \quad n \\
\quad \quad y_i = y_i + a_{i,j} \cdot x_j
\end{array}
$$

Nehmen wir an, daß sich alle benötigten Daten im Hauptspeicher befinden. Die Anzahl der Speichertransfers beträgt dann $2n^2 + n$. Ist n jedoch hinreichend klein, so daß die Komponenten des Vektors x und y_i in den Registern gehalten werden können, dann verringert sich die Anzahl der Speicherzugriffe auf $n^2 + 2n$. Bei den meisten, auch nicht mathematischen Algorithmen treten ähnliche Fälle häufig auf, daß nämlich jeweils Teile der insgesamt zu verarbeitenden Daten in einer Phase des Programms häufig und ansonsten gar nicht oder sehr selten benötigt werden. Je nach Ausprägung dieses Speicherverhaltens sprechen wir von hoher bzw. geringer Datenlokalität.

$$
\text{Datenlokalität} = \frac{\text{Anzahl der Rechenoperationen}}{\text{Anzahl der Speichertransfers}}
$$

Die Ablaufgeschwindigkeit von Algorithmen hoher Datenlokalität verbessert sich durch die heute üblichen Cache-Speicher. Dabei handelt es sich um schnelle Pufferspeicher

relativ geringer Kapazität, die zwischen Zentraleinheit und Systembus geschaltet sind, um die Trägheit des Hauptspeichers zu kaschieren. Sie arbeiten nach dem FIFO (First In - First Out) Prinzip. Demgemäß werden Daten, die vom Hauptspeicher zur CPU transportiert werden, zusätzlich im Cache abgelegt. Ist der Cache bereits voll belegt, dann werden die jeweils ältesten Daten verworfen, und durch die neuen überschrieben. Unter günstigen Umständen befinden sich häufig benötigte Daten im Cache, von wo aus sie schnell zur CPU transportiert werden können.
Bei der Matrix-Vektor-Multiplikation haben wir, durch das Problem bedingt, nur eine geringe Datenlokalität: Nur die Komponenten der Vektoren werden wiederholt benutzt; es stehen im günstigsten Fall $2n^2$ Rechenoperationen $n^2 + 2n$ Speichertransfers gegenüber. Eine hohe Datenlokalität kann bei der Matrix-Matrix-Multiplikation $C = AB$ erreicht werden, wenn der Cache Speicher ausreichend Platz zur Speicherung einer der beiden Matrizen bietet. Dann haben wir nämlich rund $2n^3$ Rechenoperationen gegenüber $3n^2$ Hauptspeichertransfers. Sind die Matrizen jedoch sehr groß, dann tritt folgender Fall ein: Zum Zeitpunkt, da ein Matrixeintrag $a_{i,j}$ zum wiederholten Mal von der CPU angefordert wird, sind zwischenzeitlich so viele Daten aus dem Hauptspeicher geladen worden, daß $a_{i,j}$ im Cache bereits wieder überschrieben worden ist - wir sprechen von Cache-Überlauf. Um die hohe Datenlokalität des Algorithmus gewinnbringend auszunutzen, zerlegt man das Problem in kleinere Teilprobleme, deren Datenaufkommen hinreichend gering ist, so daß der Cache Speicher optimal genutzt wird. Bei Matrix-Matrix-Multiplikationen $C = AB$ bietet sich eine Blockeinteilung der Matrizen an:

$$\begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix} = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{bmatrix}$$

Die Blockgrößen sind so zu wählen, daß Cache-Überlauf vermieden wird. Werden die Matrixprodukte A_iB_j in geeigneter Reihenfolge ausgeführt, dann ist sichergestellt, daß sich fast immer eine der Blockmatrizen im Cache befindet.
In heutigen Rechnern sind oft zwei Cache-Stufen anzutreffen: Der interne Cache, der auf der CPU untergebracht ist, und ein größerer, aber etwas langsamerer externer Cache, der zwischen CPU und Systembus geschaltet ist. Alle Speichermedien des Computers zusammen gesehen ergeben eine hierarchische Anordnung:

Speichermedium	Zugriffszeit (auf 64 Bit) in Prozessortakten (150 MHz)
Register	0
int. Cache	0
ext. Cache	2
Hauptspeicher	7 - 25
Festplatte	1000 - 10000

Die Werte für die Zugriffszeiten beziehen sich auf den Rechner „Silicon Graphics Challenge“ und sind zum Teil Schätzungen. Wie aus der Tabelle ersichtlich ist, können die

Zugriffszeiten auf Hauptspeicher und Festplatte stark variieren. Im Falle des Hauptspeichers liegt das daran, daß nie ein einzelnes Datum vom Hauptspeicher zur CPU transportiert wird, sondern immer gleich eine ganze „Cache-Line“, das sind bei der SGI Challenge 128 Byte (32 Fließkommazahlen einfacher Genauigkeit). Es wird nämlich davon ausgegangen, daß die dem angeforderten Datum im Speicher folgenden Daten für die nächsten Operationen in der CPU gebraucht werden. Die besagten 128 Byte werden im Cache abgelegt, und können von dort vergleichsweise schnell in die CPU geladen werden. Nehmen wir als Beispiel wieder die Matrix-Vektor-Multiplikation. Die Matrix sei spaltenweise abgespeichert, das heißt die Einträge $a_{i,j}$ und $a_{i+1,j}$ haben aufeinanderfolgende Speicheradressen. Wir haben zwei Möglichkeiten, die Schleifen zur Berechnung von $y = Ax$ anzuordnen:

$$
\begin{array}{ll}
(i) & for \quad i = 1 \quad to \quad n \\
 & \quad for \quad j = 1 \quad to \quad n \\
 & \quad\quad y_i = y_i + a_{i,j} x_j
\end{array}
$$

$$
\begin{array}{ll}
(ii) & for \quad j = 1 \quad to \quad n \\
 & \quad for \quad i = 1 \quad to \quad n \\
 & \quad\quad y_i = y_i + a_{i,j} x_j
\end{array}
$$

Hinsichtlich der Schleifenanordnung ist (ii) vorteilhafter, denn in der inneren Schleife wird auf Matrixelemente zugegriffen, die im Speicher aufeinanderfolgend abgelegt sind, und daher zu einem großen Teil im (internen) Cache zu finden sind.

Wenn die Cache-Line eine Länge von t Fließkommazahlen besitzt, dann haben wir in diesem Fall nur n^2/t Hauptspeicherzugriffe. Abhängig von der Datenmenge haben wir bei der Anordnung (i) bis zu n^2 Hauptspeicherzugriffe auf die Matrixeinträge.
Die folgenden Messungen wurden auf Silicon Graphics Chellenge durchgeführt. Deutlich zu erkennen ist, daß dieser Rechner darauf ausgelegt ist, aufeinanderfolgende Speicherzugriffe sehr schnell auszuführen. Die Unregelmäßigkeiten bei der ungünstigen Zugriffsreihenfolge nach Algorithmus (i) sind vermutlich auf das memory-interleaving zurückzuführen: Die Daten sind im Hauptspeicher auf viele einzelne Speicherbausteine verteilt. Nach einem Zugriff auf einen solchen Baustein benötigt dieser etwa 60 ns Zeit, bevor die nächste Anfrage bearbeitet werden kann. Um schneller auf den Speicher zugreifen zu können, werden Daten mit aufeinanderfolgenden (logischen) Adreßnummern zyklisch auf die Speicherbausteine verteilt. Bei einer Speicherzugriffsfolge wie in Algorithmus (i) werden aufeinanderfolgend Daten gelesen, deren logische Adressen sich um n unterscheiden. Ist n ein Teiler der Anzahl installierter Speicherbausteine, dann befinden sich die gelesenen Daten mit hoher Wahrscheinlichkeit in ein und demselben Baustein. Die Zugriffszeit erhöht sich. Fassen wir nun die bisherigen Ergebnisse zusammen:

i) Die Diskrepanz zwischen der Rechenleistung moderner Prozessoren und den Übertragungsraten zum Hauptspeicher kann durch die Realisierung schneller Pufferspeicher kaschiert werden. Algorithmen zur Verarbeitung großer Datenmengen

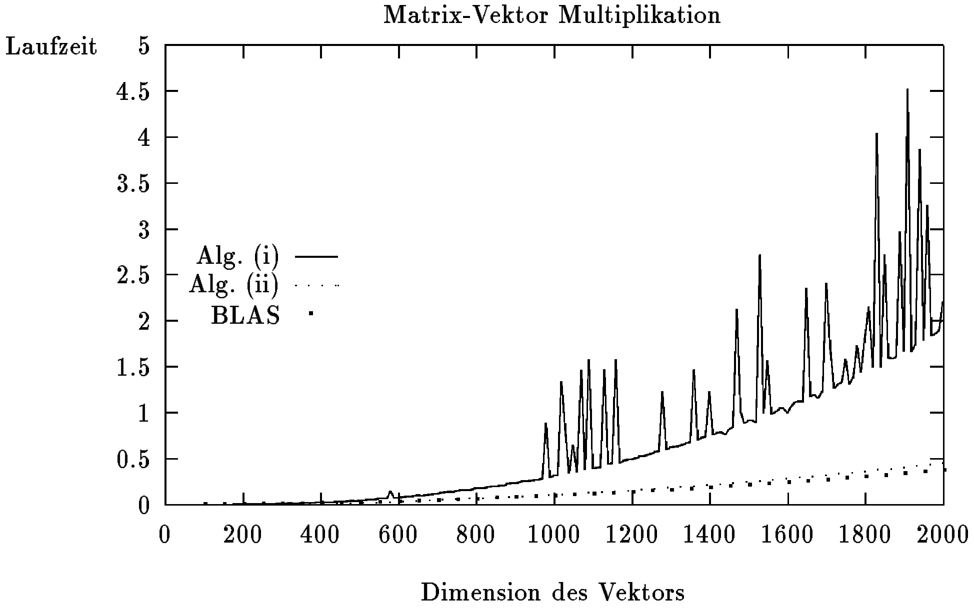

Abbildung 3.2: Matrix-Vektor-Multiplikation auf der SGI-Challenge (ein Prozessor)

müssen über eine hohe Datenlokalität verfügen, damit die Ressourcen des Computers effizient genutzt werden können. Sie sollten als Blockalgorithmen umgesetzt werden und die Anpassung der Blockgröße an die jeweilige Rechnerarchitektur erlauben.

ii) Das Bussystem der meisten Rechner erreicht seine maximale Transferrate, wenn größere Datenmengen gelesen werden, die an aufeinanderfolgenden Speicheradressen abgelegt sind. Es werden daher immer gleich mehrere Dateneinheiten übertragen und die nicht benötigten „auf Abruf“ im Cache Speicher abgelegt. Die Reihenfolge, in der auf die Daten zugegriffen wird, sollte demzufolge der Anordnung der Daten im Speicher entsprechen, um unnötigen Hauptspeicherverkehr zu vermeiden.

Bei algebraischen Algorithmen wird meist der größte Teil der Laufzeit zur Ausführung elementarer Vektor- und Matrixoperationen verwandt. Diese Operationen können mit Hilfe der BLAS-Routinen effizient ausgeführt werden, besonders dann, wenn speziell an die Hardware angepaßte Versionen dieser Routinen vorliegen. Es werden drei „level“ unterschieden:

- BLAS1: Vektor-Vektor-Operationen (z.B. Skalarprodukt)
- BLAS2: Matrix-Vektor-Operationen (z.B. Matrix-Vektor-Produkt)
- BLAS3: Matrix-Matrix-Operationen (z.B. Matrix-Matrix-Produkt)

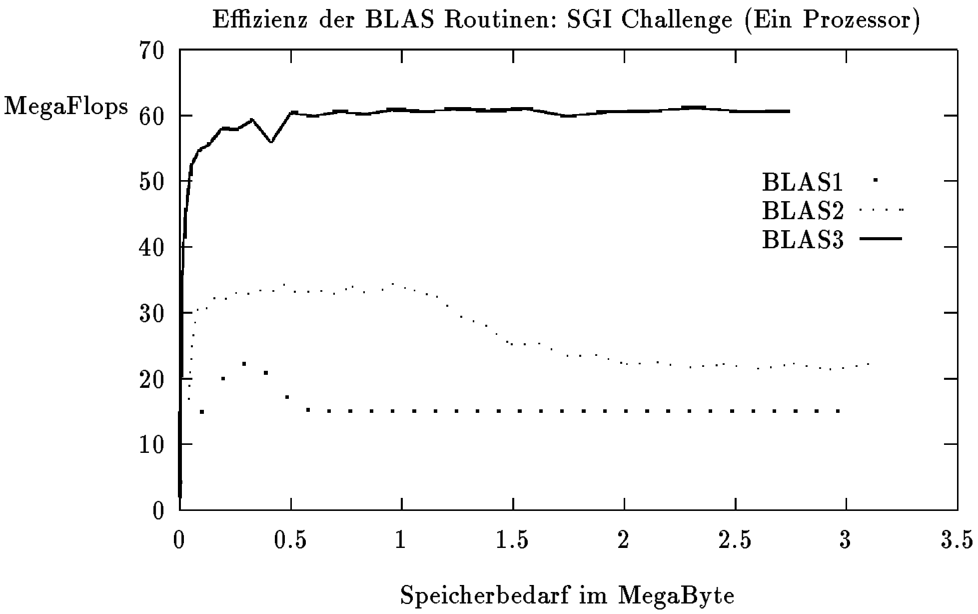

Abbildung 3.3: Effizienz der BLAS-Routinen auf der SGI-Challenge (ein Prozessor)

Der obigen Forderung nach Datenlokalität entsprechen nur die BLAS3 Operationen. Daß BLAS2 immerhin noch bessere Leistungen bringt als BLAS1, zeigt die Messung (vergleiche Abb. 3.3). Die BLAS3 Routinen erreichen fast die Maximalleistung von 75 Mega-Flops (rund 75 Mio. Fließkommaoperationen) und sind damit etwa drei bis viermal schneller als BLAS1 und BLAS2. Deutlich zu sehen ist der Einfluß des Cache: Die Effizienz von BLAS1,2 fällt bei größeren Datenmengen etwa um 30 Prozent ab. Bei den Nachfolgemodellen des hier gemessenen R4400 Prozessors, die bis zu 400 Mega-Flops leisten können, wird die Diskrepanz zwischen BLAS1/2 und BLAS3 vermutlich noch größer sein.

3.2 Prozessorinterne Parallelität

Die Leistungsfähigkeit von Prozessoren wird bestimmt durch ihre Taktfrequenz und die Anzahl der Operationen, die sie pro Taktzyklus ausführen können. Zur Verkürzung der benötigten Taktzyklen pro Operation wird das sogenannte Pipeline-Konzept angewandt. Ziel ist es, daß pro Takt und Funktionseinheit des Prozessors möglichst ein Ergebnis produziert werden kann. Hierzu werden die Rechenoperationen in Teilschritte zerlegt, und die technischen Vorkehrungen (Pipeline) getroffen, daß diese Operationen parallel abgearbeitet werden können. Nehmen wir als Beispiel die Addition zweier Vektoren x und y. In der folgenden Tabelle sind vertikal die Teilschritte der Addition aufgetragen. In horizontaler Richtung schreitet die Zeit in Taktzyklen fort.

	t=0	t=1	t=2	t=3	t=4
Eingabe	x_0, y_0	x_1, y_1	x_2, y_2	x_3, y_3	x_4, y_4
Exponenten anpassen	x_0, y_0	x_1, y_1	x_2, y_2	x_3, y_3	x_4, y_4
Addition		x_0, y_0	x_1, y_1	x_2, y_2	x_3, y_3
Normalisieren			x_0, y_0	x_1, y_1	x_2, y_2
Runden				x_0, y_0	x_1, y_1
Ausgabe					x_0, y_0

Quelle: Ehrhard 1990, S. 35

Wenn laufend neue Daten nachgeschoben werden, dann erhalten wir nach dem sechsten Takt ein Ergebnis pro Zyklus. Wenn mehrere, unabhängige Pipelines zur Verfügung stehen, können pro Takt auch mehrere Ergebnisse produziert werden - wir sprechen dann von Vektorprozessoren.
Typisch für Pipelineprozessoren ist, daß sie ihre theoretische Maximalleistung in realen Anwendungen fast nie erreichen; dies ist nämlich nur dann der Fall, wenn ständig alle Pipelines mit Daten gefüllt sind. Das heißt, daß pro Taktzyklus für jede Pipeline ein Datum zur Verfügung stehen muß, was nur möglich ist, wenn sich die Daten im schnellen Cache befinden. Wieder spielt die Datenlokalität eine entscheidende Rolle. Es können aber auch Datenabhängigkeiten auftreten, die verhindern, daß die Pipeline gefüllt werden kann. Es soll zum Beispiel die Summe $\sum_{i=1}^{n} a_i$ berechnet werden. Ein naiver Algorithmus wäre:

$$\begin{aligned} s &:= a_1 + a_2 \\ s &:= s + a_3 \\ &\dots \\ s &:= s + a_n \end{aligned}$$

Wir können die Summe $s = s + a_3$ aber erst berechnen, wenn das Ergebnis $s = a_1 + a_2$ vorliegt, das heißt, in der Pipeline kann sich immer nur ein Wertepaar befinden. Bei eingeschalteter Optimierung (Option -O) übernehmen gute Compiler die Aufhebung solcher Abhängigkeiten, was jedoch immer Leistungseinbußen gegenüber völlig unabhängigen Berechnungen mit sich bringt.
Wenn wir die zu erwartende Leistung bei einer komplexen Operation (z.B. Skalarprodukt) berechnen wollen, dann müssen wir der Pipelinestruktur Rechnung tragen. Wir gehen von einer l-stufigen Pipeline aus, ein Speicherzugriff benötige s Prozessortakte. Zur Ausführung von n einfachen, unabhängigen Operationen (z.B. Additionen) benötigen wir dann

$$t(n) = (s + l - 1 + n)$$

Prozessortakte. Die Pipeline-Effizienz ist die Anzahl der Ergebnisse pro Taktzyklus:

$$E_p(n) = \frac{n}{s + l - 1 + n}$$

Es gilt $\lim_{n\to\infty} E_p(n) = 1$, das heißt die Maximalleistung wird für große n annähernd erreicht. Wir bezeichnen mit $n_{1/2} := s + l - 1$ den Startup der betreffenden Operation. Bei modernen Rechnern treffen wir nicht nur beim Prozessor auf das Pipeline-Prinzip. Wir haben es bereits bei der Speicherverwaltung kennengelernt: Dort entspricht die Länge der cache-line der Anzahl der Pipelinestufen.

3.3 Multiprozessorsysteme

Es gibt sehr viele verschiedene Konzepte, mehrere Prozessoren in einem Computer parallel zu betreiben. Bei der Silicon Graphics Challenge zum Beispiel ist es möglich, bis zu 12 Prozessoren an den Systembus anzuschließen. Die Prozessoren teilen sich dann alle Systemressourcen wie Festplatten, Arbeitsspeicher und Cache. Aus diesem Grund ist nur eine begrenzte Anzahl an Prozessoren für diesen Typ von Parallelrechner (shared-memory) sinnvoll. Der Vorteil der Shared-Memory-Architektur liegt in der relativ einfachen Implementierung paralleler Algorithmen. Die Interprozessorkommunikation wird über den Hauptspeicher abgewickelt und ist nur wenig langsamer als „normale“ Hauptspeichertransfers. Auch hier gibt es pathologische Fälle, die dazu führen können, daß parallel ablaufende Programme kaum schneller sind als ihre seriellen Pendants:

i) Zu kurze Synchronisationszyklen: Treten im Verlauf einer Rechnung Zwischenergebnisse auf, die im folgenden allen Prozessoren bekannt sein müssen, dann sprechen wir von einem Synchronisationspunkt an dieser Stelle des Algorithmus. Wenn nicht alle beteiligten Prozessoren gleichzeitig mit ihren Berechnungen am Synchronisationspunkt ankommen, dann entstehen unerwünschte Wartezeiten. Darüberhinaus bedeutet jeder Neustart des Parallelbetriebs nach einem Synchronisationspunkt einen gewissen Initialisierungsaufwand (Startup), der mit kürzeren Synchronisationszyklen immer mehr Rechenleistung verschlingt.

ii) Ungünstige shared-memory Zugriffe: Ein Zugriff auf gemeinsam genutzten Speicher ist eine komplizierte Sache, denn es muß sichergestellt sein, daß immer auf die aktuellsten Daten zugegriffen wird. Dabei können sich gleichzeitig mehrere Kopien eines Datums in verschiedenen Speicherebenen, auf verschiedenen Prozessoren befinden. Mit dem Verwaltungsaufwand erhöht sich auch die Zugriffszeit; im ungünstigsten Fall fordert ein Prozessor ein Datum an, dessen aktuellste Kopie sich im internen Cache eines anderen Prozessors befindet.

Die Programmentwicklung auf dem Intel IPSC860 gestaltet sich wesentlich aufwendiger als auf der Challenge. Der IPSC860 ist ein Parallelrechner vom Typ distributed-memory: Alle Prozessoren sind mit eigenem Hauptspeicher ausgestattet. Einen gemeinsamen Speicher gibt es nicht, die Prozessoren kommunizieren über Verbindungsleitungen. Numerieren wir die Prozessoren mit 0 beginnend durch, dann sind alle Prozessoren, deren Nummern sich um eine Zweierpotenz unterscheiden, direkt miteinander verbunden. Der entscheidende Unterschied zu shared-memory-Computern ist, daß hier

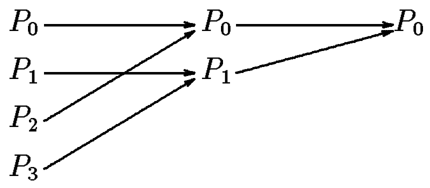

Abbildung 3.4: FAN-IN-Schema (vier Prozessoren)

die Kommunikationsbandbreite mit der Prozessorzahl wächst und daher der Rechner theoretisch mit beliebig vielen Prozessoren ausgestattet werden kann (der IPSC860 unterstützt bis zu 128 Prozessoren).
Nachrichten zwischen zwei nicht direkt verbundenen Prozessoren erreichen nur auf Umwegen ihr Ziel. Spezielle Operationen eignen sich aber besonders für diese Kommunikationsstruktur: zum Beispiel FAN-IN-Operationen. Haben wir N Prozessoren, die jeweils einen Summanden speichern, dann kann die Summe gemäß dem Kommunikationsdiagramm in Abbildung (3.4) berechnet werden. Es werden dabei immer nur direkte Kommunikationskanäle benutzt, und bereits nach $log_2 N$ Schritten liegt das Ergebnis (auf dem ersten Prozessor) vor. Soll es allen Prozessoren bekannt gemacht werden, dann ist das FAN-IN-Muster von rechts nach links zu durchlaufen. Die Firma Intel hat ihrem Parallelcomputer glücklicherweise mit einer Programmsammlung ausgestattet, mit der alle gängigen Kommunikationsaufgaben bewältigt werden können. Für die Programmentwicklung spielt diese daher nur eine untergeordnete Rolle. Für die Praxis von großer Bedeutung ist die geringe Übertragungsleistung der Verbindungen. Diese beträgt gerade einmal 2 Mega-Byte/sek, das ist etwa Festplattengeschwindigkeit. In Anbetracht der recht hohen Prozessorleistung von maximal 80 Mega-Flops ist dieser Wert sehr gering. Allerdings werden Kommunikationsaufgaben unabhängig vom Prozessor ausgeführt. Ein Algorithmus sollte also immer so angelegt sein, daß sich Berechnung und Kommunikation überlappen. Entscheidend für die Effizienz eines Programmes ist also nicht der Kommunikationsaufwand an sich, sondern die Frage, ob der Prozessor genügend „Arbeit“ hat, während er kommuniziert. Das impliziert, daß der Rechenaufwand insgesamt größer sein sollte als der Kommunikationsaufwand - auch hier ist es wieder die Datenlokalität, die über die Effizienz des Algorithmus entscheidet. Das Modell des hierarchischen Speicheraufbaus findet auch hier seine Anwendung; hinzu kommen Hierarchieebenen für den Speicher „entfernter“ Prozessoren.
Zur Leistungsbewertung von Programmen haben wir bisher den Effizienzbegriff in seiner Alltagsbedeutung benutzt. Wir wollen ihn nun doch genauer festlegen.

Definition 3.3.1 *i) Die Effizienz eines Programmes ist der prozentuale Anteil an der theoretischen Maximalleistung, die bei Ausführung des Programmes erreicht wird.*

ii) Es sei t_N die Zeit, die ein paralleles Programm benötigt, um auf einem Parallelrechner mit N Prozessoren abzulaufen. Die parallele Effizienz definieren wir als

$$E_N = \frac{t_1}{N t_N}$$

iii) Als Speedup bezeichnen wir den Faktor, um den ein paralleles Programm schneller abläuft als das serielle:

$$S_N = \frac{t_N}{t_1}$$

Bei der Analyse paralleler Algorithmen gehen wir folgendermaßen vor: Erst zerlegen wir den Algorithmus in unabhängige, parallel ausführbare Teilsschritte, zwischen denen jeweils ein Synchronisationspunkt liegt. Dann berechnen wir den Aufwand für die parallelen Rechenaufgaben zwischen je zwei Synchronisationspunkten. Seien $A_1, \ldots, A_p$ die unabhängigen Aufgaben in einem Teilschritt, und $T(A_j)$ deren Aufwand. Dann können wir

$$s = \max_{1 \leq j \leq p} T(A_j) - \min_{1 \leq j \leq p} T(A_j)$$

als Startup dieses Teilschrittes interpretieren.

Kapitel 4

Implementierungen und Laufzeitanalysen

Der Algorithmus von Golub, Kahan und Reinsch (GKR) gilt als das Verfahren der Wahl zur Berechnung der Singulärwertzerlegung vollbesetzter, unstrukturierter Matrizen. Wir nehmen es als Referenz und Ausgangspunkt zum Entwurf eines parallelen Programms. Der GKR-Algorithmus besteht aus zwei Teilen: Der Bidiagonalisierung und der QR-Iteration, deren Eignung für Parallelrechner wir anhand der folgenden Kriterien untersuchen wollen:

- Datenunabhängigkeit
- Datenlokalität

4.1 Bidiagonalisierung

Wir wollen die Anzahl der Indizes möglichst gering halten und führen daher eine vereinfachte Notation ein. Dabei können wir uns auf die klassische Householdermethode beschränken, denn die bei dem Blockalgorithmus auftretenden Abhängigkeiten bleiben im wesentlichen die gleichen. Wir erhalten die Bidiagonalmatrix $B \in \mathbb{R}^{n\times n}$ aus der Ausgangsmatrix $A \in \mathbb{R}^{m\times n}$ durch wiederholtes Anwenden von Algorithmus (2.2.4):

$$\begin{aligned} A_s &= (\mathcal{H}^{s+1,s}_{n,m}(\mathcal{H}^{s,s}_{m,n}(A_{s-1})^T))^T, \ \ 1 \le s < n-1, \\ \begin{bmatrix} B \\ 0 \end{bmatrix} &= \mathcal{H}^{n-1,n-1}_{m,n}(A_{n-2}). \end{aligned}$$

Wenden wir $\mathcal{H}^{i,j}_{m,n}$ auf die Matrix $A \in \mathbb{R}^{m\times n}$ an, so wird nur die rechte $(m-i)\times(n-j)$-Untermatrix von A verändert. Mit tiefgestellten Indizes wollen wir von nun an diese

Untermatrizen bezeichnen:

$$\mathcal{H}^{i,i}_{m,n}(A_{i-1}) = \mathcal{H}^{i,i}_{m,n}\begin{bmatrix} * & * & & \\ & * & * & \\ & & & \begin{bmatrix} A_{i-1} \end{bmatrix} \end{bmatrix} =: \begin{bmatrix} * & * & & \\ & * & * & \\ & & * & \begin{bmatrix} \bar{A}_{i-1} \end{bmatrix} \end{bmatrix}$$

$$(\mathcal{H}^{i+1,i}_{m,n}(\bar{A}^T_{i-1}))^T =: \begin{bmatrix} * & * & & \\ & * & * & \\ & & * & * \\ & & & \begin{bmatrix} A_i \end{bmatrix} \end{bmatrix}$$

Außerdem ersetzen wir

$$P_i^T := \mathcal{H}^{i,i}_{m,n}(I_m) \text{ und } Q_i = \mathcal{H}^{i,i+1}_{m,n}(I_n)$$

Die zu den P_i und Q_i gehörenden Reflexionsvektoren und Konstanten seien u_i, γ_i und v_i, α_i. Ferner bezeichnen wir mit $(A)_k$ und $(A)^l$ die k-te Spalte und l-te Zeile von A. Die Abbildung (4.1) veranschaulicht die Abhängigkeiten bei der Bidiagonalisierung:

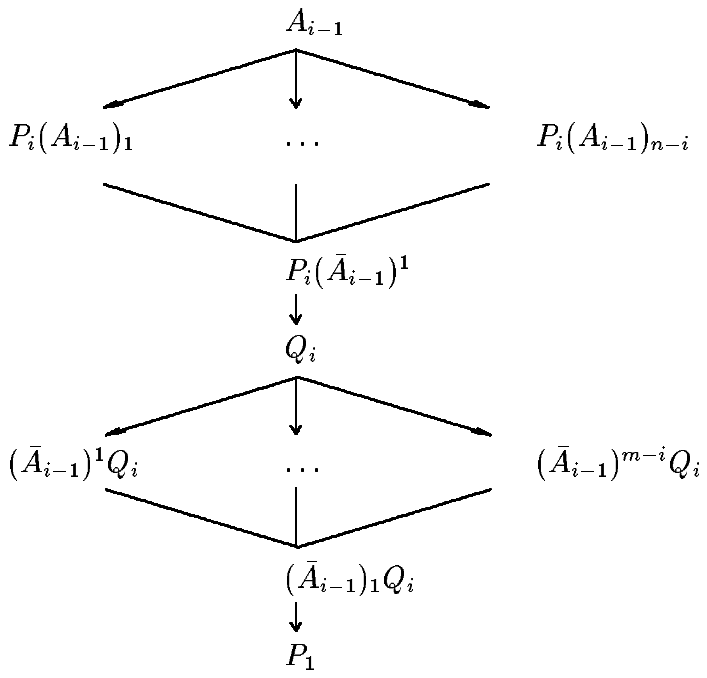

Abbildung 4.1: Sychnronisationsdiagramm: Bidiagonalisierung

In den Zeilen des Diagramms finden sich die jeweils unabhängigen Operationen, die parallel ausgeführt werden können. Betrachten wir den Schritt von A_{i-1} nach $\bar{A}_{i-1}$: Die unabhängigen Daten sind die Spalten der Matrix A_{i-1} . Zur Berechnung von Q_i wird die erste Zeile von $\bar{A}_{i-1}$ benötigt, das heißt die parallele Berechnung von $\bar{A}_{i-1}$ muß abgeschlossen sein, damit Q_i berechnet werden kann. Diese Abhängigkeit läßt sich aber leicht auflösen. Wir erinnern uns daran, daß die Anwendung des Reflektors P auf einen Vektor x mit

$$Px = x + \gamma < u, x > u$$

berechnet werden kann. Die erste Zeile von PA mit $A \in \mathbb{R}^{m \times n}$ können wir also mit

$$(PA)^1 = (a_{1,1} + \gamma < u, (A)_1 > u_1, \ldots, a_{1,n} + \gamma < u, (A)_n > u_1)$$

berechnen. Sobald die Ergebnisse der Skalarprodukte $< u, (A)_j >$ vorliegen, kann Q_i unabhängig berechnet werden. Auf die gleiche Weise lösen wir die Abhängigkeit bei der Berechnung von P_{i+1} auf. Für den gesamten Prozeß erhalten wir $2n - 3$ Synchronisationspunkte. Zwischen dem (i-1)-ten und i-ten Synchronisationspunkt liegen rund $4(m-i)(n-i)$ Rechenoperationen, die Anzahl der abhängigen Daten beträgt maximal $m - i$. Wir haben also die günstige Situation vorliegen, daß mit größeren Matrizen der Rechenaufwand um eine Potenz schneller als der Kommunikationsaufwand wächst.

Es stellt sich nun die Frage, wie die Daten auf die einzelnen Prozessoren verteilt werden sollen, um eine möglichst gleichmäßige Verteilung des Rechenaufwands zu erreichen. Bleiben wir bei der klassischen Householdermethode, dann bietet sich eine zyklische Verteilung der Spalten oder Zeilen der Matrix an. Auf diese Weise ist gesichert, daß bei wachsender Matrixgröße und konstanter Prozessorzahl der Rechenaufwand für alle Prozessoren zunehmend ausgeglichener wird. Diese Verteilung wurde auch von Köckler und Simon vorgeschlagen [Parallel Computing 1991]
Bei den weiteren Überlegungen müssen wir nun die Architekturmerkmale des betreffenden Rechners mit einbeziehen. Relativ einfach gestaltet sich die Situation auf Parallelrechnern mit gemeinsamem Speicher. Hier haben alle Prozessoren gleichschnellen Zugriff auf alle Daten. Die ja nur theoretische Datenverteilung kann also während der Laufzeit von spaltenweise zu zeilenweise wechseln.

Bei verteiltem Speicher müssen wir uns aber von Anfang an auf eine der Varianten festlegen. Wählen wir spaltenweise Verteilung, dann treten zusätzliche Abhängigkeiten bei der Berechnung von $\bar{A}_{i-1}Q_i$ auf: Wir markieren mit dem Zeichen $\tilde{}$ die Matrizen, die sich durch Aneinanderfügen der Spalten, die ein bestimmter Prozessor speichert, ergeben. Die folgende Operation wird lokal ausgeführt:

$$\tilde{g} := \tilde{\bar{A}}_{i-1}\tilde{v}.$$

$\tilde{v}$ setzt sich aus den Komponenten des Reflexionsvektors v zusammen, die den Spaltennummern von $\tilde{\bar{A}}_{i-1}$ entsprechen. Mittels FAN-IN-Schema tauschen die Prozessoren ihre Anteile $\tilde{g}$ an g aus. Die lokalen Spalten können dann gemäß

$$\tilde{A}_i = \tilde{\bar{A}}_{i-1} + g\tilde{v}^T$$

ohne weitere Kommunikation überschrieben werden - aber eben erst nachdem die Ergebnisse der FAN-IN Prozedur vorliegen. Auch hier ist die Matrix-Vektor-Formulierung der Householderprozedur derjenigen mit Skalarprodukten überlegen, denn der Startup der FAN-IN Operation würde im zweiten Fall die Kommunikationsleistung drastisch mindern.
Weiterer Kommunikationsaufwand entsteht durch die Tatsache, daß der Reflexionsvektor u_{i+1} zu P_{i+1} bis auf die erste Komponente aus einem Spaltenvektor besteht, der ja nur einem Prozessor bekannt ist. u_{i+1} muß also zu allen anderen Prozessoren gesendet werden. Da Kommunikation und Berechnungen auf dem IPSC860 unabhängige Operationen sind, ist der günstigste Zeitpunkt, u_{i+1} abzuschicken, der frühest mögliche: Der Prozessor, der im Schritt von A_{i-1} nach A_i die Berechnung von P_{i+1} übernimmt, berechnet hierzu erst $(\tilde{\bar{A}}_{i-1}Q_i)_1$ und daraus den Reflexionsvektor u_{i+1}. Diesen schickt er ab, und führt erst dann die restlichen Rechnungen zu $\tilde{\bar{A}}_{i-1}Q_i$ aus. Da der Kommunikationsaufwand im Vergleich zum Rechenaufwand weniger schnell wächst, wird bei großen Matrizen der Vektor u_{i+1} bereits bei allen Prozessoren angekommen sein, noch bevor die Berechnungen von $\bar{A}_{i-1}Q_i$ abgeschlossen sind - wir sparen einen Synchronisationspunkt.

In der Tabelle auf Seite 59 sehen wir den Algorithmus zur Bidiagonalisierung für ein System mit verteiltem Speicher. Es seien N Prozessoren verfügbar. In der linken Spalte sehen wir den i-ten Schritt des Algorithmus für einen bestimmten Prozessor, rechts stehen Rechen- bzw. Kommunikationsaufwand der einzelnen Zwischenschritte für den gesamten Algorithmus. Durch horizontale Linien sind die Synchronisationspunkte angedeutet, das heißt an diesen Stellen des Algorithmus müssen alle Prozessoren und Nachrichten angekommen sein, bevor mit den weiteren Berechnungen fortgefahren werden kann. Die Konstanten τ_r und τ_k stehen für die Rechen- und Kommunikationsleistung des betreffenden Rechners. Die Rechenleistung ist um den Faktor τ_k/τ_r größer als die Kommunikationsleistung. Legen wir die Maximalleistung zu Grunde, dann erhalten wir für den Intel IPSC860 den Quotienten:

$$\frac{\tau_k}{\tau_r} = \frac{160}{1}$$

Wir wollen nun den Gesamtaufwand als die Summe von Rechen- und Kommunikationsaufwand berechnen. Hierbei müssen wir die Möglichkeit berücksichtigen, Kommunikation mit Rechnungen zu überlappen. Betrachten wir den letzten unabhängigen Block vor der Zeile *next i*. Dort liegen zwischen dem Senden und Empfangen von $(m-i)$ Daten rund $2(m-i)(n-i)N^{-1}$ Rechenoperationen. Wenn die Berechnungen länger dauern als die Kommunikation, müssen keine Wartezeiten in Kauf genommen werden.

$$\begin{aligned} \tau_r \frac{2}{N}(m-i)(n-i) \quad &\geq \quad \tau_k log_2(N)(m-i) \\ \Leftrightarrow \quad (n-i) \quad &\geq \quad \frac{\tau_k}{\tau_r}\frac{1}{2}N\, log_2(N) \end{aligned} \tag{4.1}$$

Aufgabe	Rechenaufwand	Kommunikationsaufwand
if $q = 0$ then berechne u, γ sende u, γ if $q \neq 0$ empfange u, γ	$\tau_r\ 2m$	$\tau_k\ log_2(N)m$
for $i = 0$ to $n - 2$ $\tilde{\tilde{A}}_{i-1} = P_i \tilde{A}_{i-1}$ if $i = n - 2$ then goto MARKE $\rho =< (\tilde{\tilde{A}}_{i-1})^1, (\tilde{\tilde{A}}_{i-1})^1 >$ $\alpha' = (\sum_{alleProzessoren} \rho)^{1/2}$	$\tau_r\ (2n^2m - 0.\bar{6}n^3)/N$ $\tau_r\ n^2/N$	$\tau_k\ log_2(N)n$

if $(i+1)$ mod $(N) = q$ then
$\alpha = -sign((\bar{A}_{i-1})_{1,1})\alpha'$
sende α
endif
if $i \leq m - N + q$ then
$\tilde{g} = (\tilde{\tilde{A}}_{i-1})_{2,\ldots}\tilde{v}$
sende $\tilde{g}$
endif
if $(i+1)$ mod $(N) \neq q$ then
empfange α
endif
	$\tau_r\ (n^2m - 0.\bar{3}n^3)/N$	$\tau_k\ log_2(N)n$

$g: empfange_{vonallenProzessoren}\tilde{g}$		$\tau_k\ log_2(N)n^2(1 - 1/N)$

$(\tilde{A}_i)_1 = (\tilde{\tilde{A}}_{i-1})_1 + (g\tilde{v}^T)_1$
MARKE
if $(i+1)$ mod $(N) = q$ then
berechne u, γ
sende u, γ
endif
if $i = n - 2$ then goto next i
$(\tilde{A}_i)_{2,\ldots} = (\tilde{\tilde{A}}_{i-1})_{2,\ldots} + (g\tilde{v}^T)_{2,\ldots}$
if $(i+1)$ mod $(N) \neq q$ then
empfange u, γ
endif
	$\tau_r\ (n^2m - 0.\bar{3}n^3)/N$
	$\tau_r\ (2nm - n^2)$
	$\tau_r\ (mn^2 - 0.\bar{3}n^3)/N$
		$\tau_k\ log_2(N)(nm - 0.5n^2)$

next i		

Das heißt, daß ab einer gewissen Matrixgröße, die nicht von der Spaltenlänge abhängt, die Kommunikation für diesen Teilschritt keinen zusätzlichen Aufwand kostet. Aufgrund der spaltenweise Verteilung der Daten gelingt eine solche Konstruktion bei der

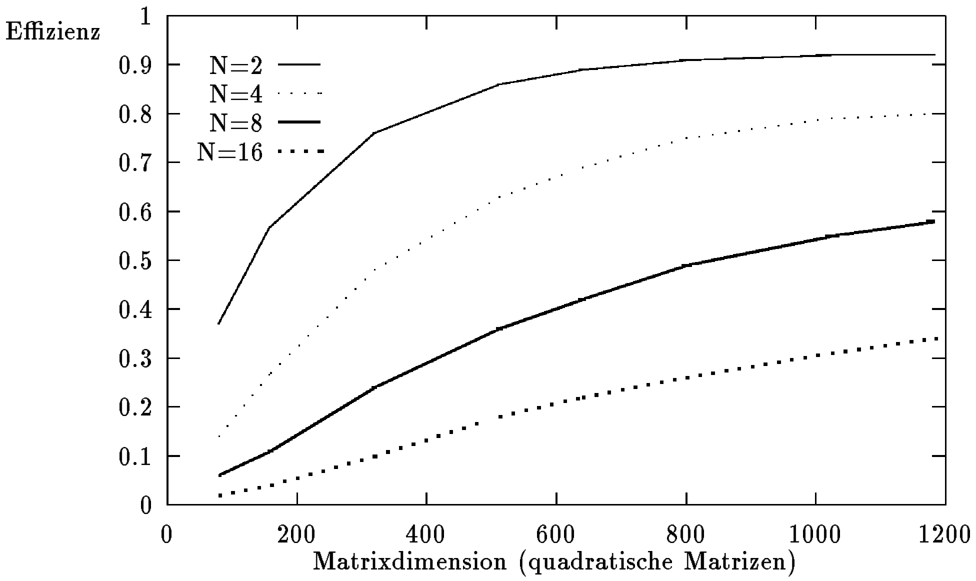

Abbildung 4.2: Parallele Effizienz: Bidiagonalisierung auf dem IPSC860

Anwendung der rechten Reflektoren $\tilde{\tilde{A}}_{i-1}Q_i$ nicht. Nur durch blockweises Berechnen von $\tilde{g}$ können Kommunikation und Rechnen überlappt werden, was lediglich einen Faktor $1/l = 1/Blockanzahl$ vor dem quadratisch wachsenden Kommunikationsaufwand bringt. Zur Vereinfachung nehmen wir an (4.1) sei für alle i erfüllt. Dann erhalten wir einen Gesamtaufwand:

$$T_{ges} = \tau_r \frac{4mn^2 - \frac{4}{3}n^3}{N} + \tau_k (log_2 N) \frac{1}{l} (n^2 - \frac{1}{N} n^2 + m + n)$$

Es ergibt sich die parallele Effizienz:

$$\begin{aligned} E_{par} &= \frac{\text{serielle Laufzeit}}{N \cdot \text{parallele Laufzeit}} \\ &= \frac{1}{1 + \frac{\tau_k}{l \cdot \tau_r}(log_2 N)\frac{n^2(1-1/N)+m+n}{4mn^2-4/3n^3}} \end{aligned}$$

Wir können hieraus ersehen, daß die Effizienz umso größer ist, je weniger Prozessoren zum Einsatz kommen und je geringer der Quotient aus Kommunikations- und Rechenleistung ist. Für einen gegebenen Rechner wächst die Effizienz mit der Matrixgröße. Die Blockgröße muß für den jeweiligen Rechner optimiert werden. Bei den Laufzeitmessungen (siehe Abb. 4.2) konnten aufgrund der geringen Speicherkapazität der Knotenrechner des IPSC860 nur relativ kleine Matrizen zum Vergleich der Effizienz herangezogen werden. Die Anzahl der Prozessoren scheint in der Praxis wesentlich stärker einzugehen, als es die Theorie nahelegt. Dabei ist zu bedenken, daß bei der theoretischen Berechnung der Effizienz, der Einfluß des Startups der Kommunikationsfunktionen vernachlässigt wurde. Außerdem dürfte in der Praxis der Quotient aus Rechen und Kommunikationsleistung größer sein, als angenommen. Aus den Meßdaten erhalten wir Werte zwischen 400 (N=2, n=1000) und 2500 (N=16, n=100).

Was die Datenlokalität angeht, so ist die Blockvariante der Householderprozedur vorteilhaft, denn sie erlaubt den Einsatz effizienter BLAS3 Routinen, während das klassische Verfahren auf BLAS2 beschränkt ist. Darüberhinaus erlaubt der Blockalgorithmus eine bessere Überlappung von Berechnungen und Kommunikation. Für die Implementierung wurde mit dem klassischen Verfahren der einfachere Weg gewählt, der sicherlich weniger effizient ist. Trotzdem wird eine brauchbare parallele Effizienz erreicht, wenn die Anzahl der Prozessoren an die Problemgröße angepaßt wird. Nehmen wir als Vergleich die von Silicon Graphics parallelisierte LAPACK-Routine für die Challenge, so zeigen die Messungen (vgl. Abb. 4.4), daß die auf dem IPSC860 erreichte parallele Effizienz bei vier Prozessoren vergleichbar ist.

4.2 Singulärwertzerlegung der Bidiagonalmatrix

Erinnern wir uns an den Algorithmus von Wilkinson, Kahan und Reinsch (2.5). Sie setzten den QR-Algorithmus mit implizitem Wilkinson-Shift zur Berechnung der Singulärwertzerlegung ein. Der neue Shift-Parameter berechnet sich aus der rechten, unteren 2×2 Matrix der aktuell Iterierten, das bedeutet, daß ein Iterationsschritt abgeschlossen sein muß, bevor der nächste gestartet werden kann. Sollen mehrere Prozessoren am Iterationsprozeß teilnehmen, so ist die Null-Shift-Strategie nach Demmel und Kahan (2.5) günstiger. Die Prozessoren können mit zeitlicher Verzögerung gestartet werden, denn die Iterationsschritte sind voneinander unabhängig. Wegen der besseren Datenlokalität empfiehlt sich dieses Vorgehen auch für serielle Rechner. Für Parallelrechner prädestiniert ist das Verfahren trotzdem nicht. Es muß nämlich sichergestellt werden, daß sich die Prozessoren nicht gegenseitig überholen. Darüberhinaus wird auf die Matrixelemente schreibend zugegriffen. Die Prozessoren müssen diese vom Cache ihres Vorläufers anfordern - sie bremsen sich gegenseitig aus. Als besonders ungünstig für Systeme mit verteiltem Speicher stellt sich das Vorgehen im dem Fall heraus, wenn während der Iteration ein Eintrag Null wird und die Matrix neu aufgesplittet werden muß. Dies geschieht nämlich mit einer geeigneten Folge von Givensrotationen, die die Matrix nach folgendem Schema aufteilt:

$$\begin{bmatrix} * & * & & & \\ & * & * & & \\ & & * & & \\ & & & * & * \\ & & & & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & & & \\ & * & * & & \\ & & & * & \\ & & & * & * \\ & & & & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & & & \\ & * & * & & \\ & & & & * \\ & & & * & * \\ & & & & * \end{bmatrix} \rightarrow \begin{bmatrix} * & * & & & \\ & * & * & & \\ & & & & \\ & & & * & * \\ & & & & * \end{bmatrix}$$

Schon bei der seriellen Effizienz weist die QR-Iteration Defizite dann auf, wenn auch die Singulärvektoren berechnet werden sollen. Hierzu werden erst die Householder-Reflektoren auf die Einheitsmatrizen angewandt, und dann werden die Givensrotationen der QR-Iteration auf diese Matrizen losgelassen. Das bedeutet aber, daß die Matrizen U und V in jedem Iterationsschritt zum Prozessor hin- und wieder zurücktransportiert werden müssen - man könnte sagen: darin besteht der größte Teil der Arbeit. Berechnen wir statt dessen die Singulärtripel mittels Bisektion und Inverser Vektoriteration, dann erhalten wir eine wesentlich bessere Effizienz, wenn wir erst die Singulärtripel der Bidiagonalmatrix berechnen und auf diese dann die Block-Reflektoren anwenden (vgl Abb. 4.3). Die Matrizen U und V werden dann nur einmal zwischen Prozessor und Hauptspeicher ausgetauscht.

Algorithmus 4.2.1 : BBI

i) Bidiagonalisierung:

$$P_{n-1}^T \cdot \ldots \cdot P_1^T A Q_1 \cdot \ldots \cdot Q_{n-2} = \begin{bmatrix} B \\ 0 \end{bmatrix}$$

ii) Berechnung der Singulärwertzerlegung von B mit Bisektion und Inverser Vektoriteration:

$$B = U_1 \Sigma V_1^T$$

iii) Berechnung der Singulärvektoren von A:

$$\begin{aligned} U &= P_{n-1} \cdot \ldots \cdot P_1 \begin{bmatrix} U_1 \\ 0 \end{bmatrix} \\ V &= Q_{n-2} \cdot \ldots \cdot Q_1 V_1 \end{aligned}$$

Bei dem vorgeschlagenen Algorithmus zur Bidiagonalisierung auf verteilten Systemen liegt die Matrix B auf allen Prozessoren vor, sobald die Bidiagonalisierung abgeschlossen ist. Die Singulärwertzerlegung der Bidiagonalmatrix kann ohne weitere Kommunikation berechnet werden, denn die Kombination Bisektion und Inverse Vektoriteration erlaubt einzelne Singulärtripel völlig unabhängig voneinander zu berechnen. Die Datenlokalität dieser Verfahren läßt aber zu wünschen übrig, denn es sind keine Blockalgorithmen. Betrachten wir das Gesamtproblem mit einem Speicherbedarf von $3n^2$ Zahlen bei quadratischen Matrizen, dann ist das Datenaufkommen von $\mathcal{O}(n)$ doch zumindest relativ gering. Auf eine Beschleunigung des Verfahrens durch Ersetzen der Bisektion durch ein modifiziertes Newton-Verfahren oder die Rayleigh-Iteration wurde verzichtet. Nach einigen Experimenten mit der Rayleigh-Iteration mußte ich feststellen, daß eine stabile Implementierung des Verfahrens den Rahmen dieser Arbeit sprengen würde. In Anbetracht des geringen Anteils von weniger als 5% ($n > 1000$), den Bisektion und Inverse Vektoriteration an der Gesamtlaufzeit haben, wäre der Aufwand der Implementierung auch kaum gerechtfertigt. Hinzu kommt, daß der Algorithmus darauf ausgelegt

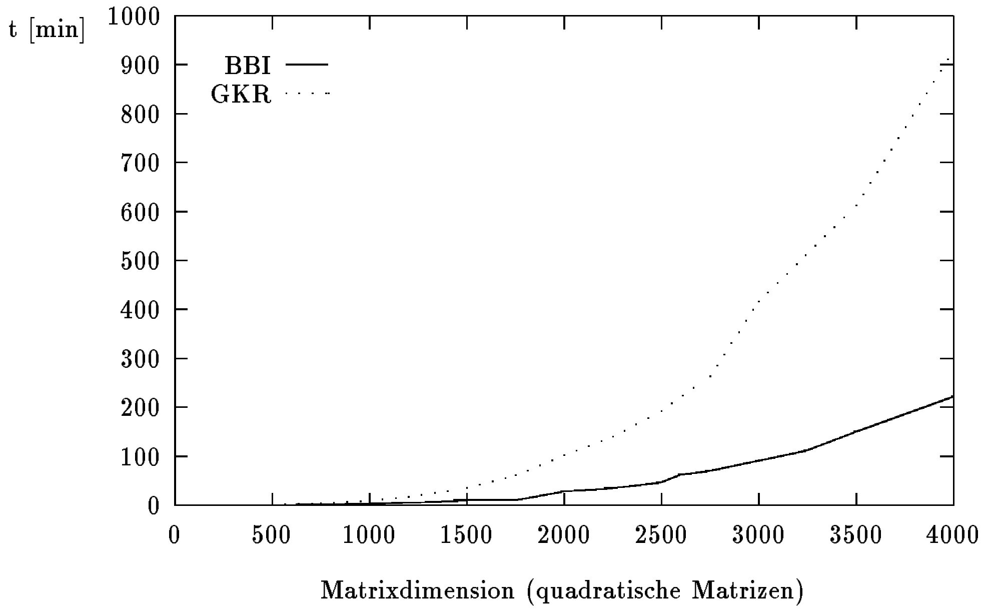

Abbildung 4.3: Laufzeitvergleich GKR - BBI, gemessen auf SGI Challenge, ein Prozessor.

ist, einige der größten Singulärtripel zu berechnen. Die Gesamtlaufzeit ist dann im wesentlichen durch die Bidiagonalisierung bestimmt.

Als letzten Schritt bleiben dann noch die Singulärvektoren der Ausgangsmatrix A zu berechnen. Wir erhalten sie analog der Bidiagonalisierung. Die Householdervektoren aus dem Bidiagonalisierungsschritt werden hier auf die Singulärvektoren der Bidiagonalmatrix angewendet. Dieser Schritt wurde für den IPSC860 nicht mehr implementiert.
Zum Vergleich des klassischen GKR- mit dem BBI-Algorithmus sollten alle Singulärtripel der Matrix

$$a_{i,j} = \frac{1}{i + 2 * j} \qquad 1 \leq i, j \leq n,$$

berechnet werden. Legen wir den Aufwand für den gesamten Algorithmus mit $5.3n^3$ fest - der doppelte Aufwand der Bidiagonalisierung - dann erhalten wir bei großen Matrizen ($n > 2000$) eine serielle Effizienz von 17.3% (BBI) bzw. 4.3% (GKR).
Für die Silicon Graphics Challenge stellt der Hersteller parallelisierte LAPACK-Routinen über das Internet zur Verfügung. Es gibt in dieser Routinensammlung Funktionen zur Bidiagonalisierung mit Block-Householder, sowie Bisektion und Inverse Vektoriteration. Die Aufgabe des zu schreibenden Programms beschränkt sich dann darauf, die Eingabewerte für die jeweiligen Funktionen entsprechend aufzubereiten. Hierzu gehört insbesondere die Anwendung der Permutationen nach (2.6). In Abbildung (4.4) sehen wir die parallele Effizienz der einzelnen Verfahren, die im BBI-Algorithmus zur An-

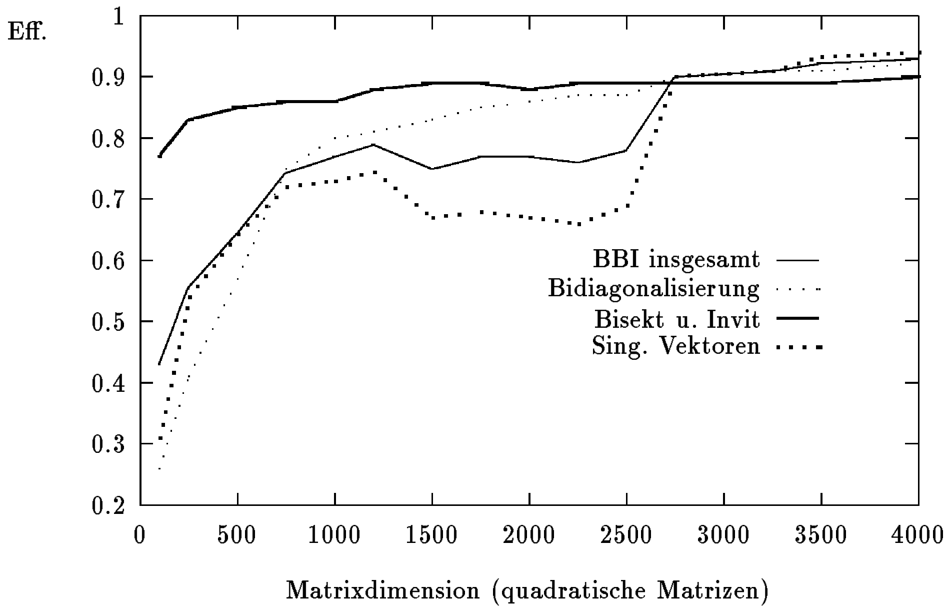

Abbildung 4.4: Parallele Effizienz einzelner Funktionen

wendung kommen. Die Rechnungen wurden auf der SGI Challenge mit 4 Prozessoren ausgeführt. Die Kombination Bisektion und Inverse Vektoriteration bringt erwartungsgemäß gute parallele Effizienz auch bei kleinen Matrizen. Doch sie überschreitet die 0.9 Marke nicht. Hierfür ist in erster Linie die Synchronisation nach Beendigung der Inversen Vektoriteration verantwortlich. Schließlich ist es wenig wahrscheinlich, daß die Prozessoren genau gleichviele Iterationsschritte ausführen müssen - es kommt zu Wartezeiten.
Bei den LAPACK-Routinen fällt das Urteil unterschiedlich aus. Die Bidiagonalisierung zeigt das typische Startupverhalten: mit größeren Matrizen wird die Effizienz immer besser. Die Anwendung der Blockreflektoren auf die Singulärvektoren der Bidiagonalmatrix kann hier nicht ganz überzeugen. Der Einbruch der Effizienz im Bereich $800 \leq n \leq 2700$ deutet auf eine wenig sorgfältige Implementierung durch den Hersteller hin. Bei sehr großen Matrizen läuft das parallele Programm auf vier Prozessoren etwa 3.6 mal schneller ab, als auf einem Prozessor. Dies ist insgesamt ein zufriedenstellendes Ergebnis, wenn man bedenkt, daß der volle Speedup (Faktor 4) praktisch nicht erreichbar ist, wenn Daten im Hauptspeicher abgelegt werden müssen. Dies ist bei den auftretenden Datenmengen von rund 180 Mega-Byte (n=4000) unumgänglich. Hinzu kommt, daß das Betriebssystem einige Prozent der Rechenleistung verbraucht, die Anwendern nicht zur Verfügung stehen. Beziehen wir die Effizienz des Verfahrens auf die maximal erreichbare Leistung der Challenge, dann erhalten wir einen Anteil von 15.5%.

Letzten Endes hat die Parallelisierung des Verfahrens für die Challenge weniger eingebracht, als die Optimierung des seriellen Verfahrens (BBI), die mit einem Faktor 4 im Vergleich zum GKR-Algorithmus zu Buche schlägt. Wegen der modernen Cache-Speicher verhalten sich serielle Rechner ähnlich wie parallel arbeitende, nur daß hier nicht Prozessoren miteinander kommunizieren, sondern der Prozessor mit der Peripherie. Verfahren, die für Parallelrechner entwickelt werden, zerlegen ein Problem in Teilprobleme, die von den einzelnen Prozessoren bearbeitet werden sollen. Wird das Datenvolumen der Teilprobleme an die Cache-Größe angepaßt, und diese nacheinander von einem Prozessor abgearbeitet, dann ist dieses Verfahren auch für serielle Rechner gut geeignet.

Die Entwicklung von Programmbibliotheken wie BLAS und LAPACK kann als die „hohe Kunst" des Algorithmendesign angesehen werden. In die genannten Bibliotheken fließen die jahrelange Erfahrung von Entwicklern und Anwendern ein, die es erst ermöglichen, ein Verfahren gegen alle denkbaren Eventualitäten abzusichern. Entsprechende Bibliotheken für Systeme mit verteiltem Speicher sind noch im Entstehen, und die Untersuchung der von Silicon Graphics parallelisierten LAPACK-Routinen für den shared-memory Rechner Challenge zeigt, daß auch für diese, vergleichsweise einfach zugängliche Architektur, die Weiterentwicklung der vorhandenen Algorithmen gewinnbringend sein kann.

Obwohl der zugrunde gelegte Aufwand für die Berechnung der Singulärwertzerlegung tatsächlich etwas höher liegen dürfte als angenommen, und quadratische Terme vernachlässigt wurden, bestätigt sich doch die Vermutung, daß die hohe Maximalleistung moderner Rechner bei realen Anwendungen bei weitem nicht erreicht wird. Das bedeutet andererseits, daß es ein hohes Optimierungspotential gibt, das nur zum Teil den Compilern[1] überlassen werden kann.

[1]Programm zur Übersetzung einer höheren Programmiersprache in Maschinencode

Anhang A

Quelltext zu BBI für SGI Challenge

```
C       BBI-Algorithmus fuer SGI Challenge
C
C       Programmiersprache: High Performance Fortran
C       Zusaetzliche Bilbliotheken: complib.sgimath_mp (BLAS u. LAPACK)
C
C
      SUBROUTINE SBBI(A,U,V,LDUV,S,M,N,K,TAUP,TAUQ,D,E,HD,HE
     & ,IWORK,IFAIL,WORK,LWORK,INFO,
     & IBLOCK,ISPLIT)

      REAL A(M,N),U(LDUV,M),V(LDUV,N),S(N)
      REAL TAUP(M),TAUQ(M),WORK(LWORK)
      REAL E(N),D(N),HD(2*N),HE(2*N)
      INTEGER M,N,K,INFO,LWORK,IWORK(N),IFAIL(N),IBLOCK(2*N),ISPLIT(2*N)
      INTEGER LDUV

C
C SBBI BERECHNET DIE K GROESSTEN SINGULEARTRIPEL DER
C MXN MATRIX A (M>=N).
C DIE SINGULAERWERTE S UND DIE ZUGEHOERIGEN VEKTOREN U,V WERDEN IN
C AUFSTEIGENDER REIHENFOLGE SORTIERT.
C
      REAL LB,UB,EPS1,ABS,VH,UH
      INTEGER KH,P,IL,IU,NOEV,P5N,KL,KU,J,J0,J1,J2,I,I0,I1,I2,L
      INTEGER P2N,N2,INDH,INDP,LINFO(10)
      INTEGER NPTR,MPTR
      REAL T1,T2,TM(2)

      EXTERNAL SSTEVX,SGEBRD,SCOPY,SSWAP,SORMBR
```

```
C NUMP: ANZAHL VERFUEGBARER PROZESSOREN

NUMP = 4

C EPS1: RELATIVE FEHLERSCHRANKE

EPS1 = 1.0E-8

C LAPACK AUFRUF: BIDIAGONALISIERUNG

CALL SGEBRD(M,N,A,M,D,E,TAUQ,TAUP,WORK,LWORK,INFO )
IF (INFO.NE.0) PRINT*,'ERROR IN DGEBRD: ',INFO

C INIT 2NX2N TRIDIAGONALMATRIX FUER JEDEN PROZESSOR ----------------

C$DOACROSS LOCAL(P,I,I1,I0,I2,P2N),
C$&    SHARE(NUMP,N,HD,HE,D,E),
C$&    MP_SCHEDTYPE=SIMPLE

DO 10 P = 0,NUMP-1
P2N      = P*2*N
DO 10 I=1,N
I1       = I*2+P2N
HE(I1-1) = D(I)
HE(I1)   = E(I)
HD(I1-1) = 0.0
HD(I1)   = 0.0
10 CONTINUE

IF (K.LT.NUMP) THEN
INDP = K-1
ELSE
INDP = NUMP-1
ENDIF

C BERECHNE AUSGEWAEHLTE EIGENWERTE/VEKTOREN VON B'*B:
C SCHLEIFE UEBER ALLE PROZESSOREN
C P: PROZESSORNUMMER
C KL,KU:  INDEXGRENZEN FUER DIESEN PROZESSOR

C$DOACROSS LOCAL(P,I,J,N2,KU,KL,KH,NPTR,MPTR,
C$&        NSPLIT,NOEV,INFO,NMODP,VH),
```

```
C$&          SHARE(NUMP,LB,UB,INDP,N,EPS1,WORK,S,HD,HE,K,U,M
C$&                IBLOCK,ISPLIT,IWORK,IFAIL,KUA,KLA,LDUV),
C$&          MP_SCHEDTYPE=SIMPLE

DO 500 P=0,INDP

NMODP = MOD(K,NUMP)
N2 = 2*N

IF (NMODP.EQ.0) THEN
        KL = P*(K/NUMP)+1
        KU = KL + K/NUMP -1
ELSE
IF (P.LT.NMODP) THEN
        KL = P*(K/NUMP+1)+1
        KU = KL + K/NUMP
ELSE
        KL = P*(K/NUMP)+1+NMODP
        KU = KL + K/NUMP -1
ENDIF
ENDIF

KH = KL
KL = N2-KU+1
KU = N2-KH+1
NPTR = N2*P+1
MPTR = 10*M*P+1

C LAPACK AUFRUF: BISEKTION UND INVERSE VEKTORITERATION

   IF (KU.GE.KL) THEN

CALL STEVX('V','I',N2,HD(NPTR),HE(NPTR),LB,UB,KL,KU,EPS1,
     & NOEV,S(KL-N),U(1,KL-N),LDUV,
     & WORK(MPTR),IWORK(5*N2*P+1),IFAIL(NPTR),INFO)

ENDIF

C SORTIEREN UND UMNUMMERIEREN DER SINGULAERTRIPEL

VH = SQRT(2.0)

DO 100 J=KL-N,KU-N
```

```
DO 200 I=1,N
WORK(MPTR+I) = U(2*I,J)*VH
WORK(MPTR+N+I) = U(2*I-1,J)*VH
200 CONTINUE

DO 300 I=1,N
U(I,J) = WORK(MPTR+I)
300 V(I,J) = WORK(MPTR+N+I)

100 CONTINUE

500 CONTINUE

IF (K.GT.N/2) THEN

C$DOACROSS LOCAL(VH,I,L), SHARE(U,V,S,N)

DO 600 I=1,N/2
CALL SSWAP(N,U(1,I),1,U(1,N-I+1),1)
CALL SSWAP(N,V(1,I),1,V(1,N-I+1),1)
DO 700 L=N+1,M
700 U(L,I) = 0.0
VH = S(I)
S(I) = S(N-I+1)
600 S(N-I+1)= VH

ELSE

C$DOACROSS LOCAL(VH,I,L), SHARE(U,V,S,N)

DO 800 I=1,K
S(I) = S(N-I+1)
CALL SCOPY(N,U(1,N-I+1),1,U(1,I),1)
CALL SCOPY(N,V(1,N-I+1),1,V(1,I),1)
DO 900 L=N+1,M
900 U(L,I) = 0.0
800 CONTINUE

ENDIF

T2=ETIME(TM)
WRITE(1,*) 'INVIT ',T2-T1
```

```
C LAPACK AUFRUF: BLOCK-REFLEKTOREN ANWENDEN
C BERECHNE U:

CALL SORMBR('Q','L', 'N', M, K, N, A, M,
     $ TAUQ, U, LDUV, WORK, LWORK, INFO)

C BERECHNE V:

CALL SORMBR('P', 'L', 'N', N, K, M, A, M,
     $ TAUP, V, LDUV, WORK, LWORK, INFO)

RETURN
END
```

Anhang B

Quelltext zu BBI für INTEL IPSC860 (Auszug)

```
/* BBI-Algorithmus fuer INTEL IPSC860 */
/* (eingeschraenkte Funktionalitaet) */
/* Auszug aus dem Quellcode  */

/* Programmiersprache C, mit Intelspez. Kommunikationsfunktionen */
/* Zusaetzliche Bibliotheken: kmath (BLAS) */

/*** Includes ***/

#include "bidiag.h"

#define maxdim 1200
#define maxdim2 2400000

/*** Globale Variablen ***/

long
N,                        /* Anzahl Prozessoren */
P,                        /* Mein Prozessor */
m,                        /* Globale Spalten u. Zeilen */
n,
s,                        /* aktuelle globale Spalte  */
L,                        /* groesster globaler Spaltenindex */
K,                        /* restliche Spalten */
IMF,                      /* Flag: Proz. speichert akt. Pivot-Spalte */
IMN,                      /* Flag: Proz. speichert naechste Piv.-Spalte*/
IMO,                      /* Flag: Prozessor ist fertig */
```

```
k,                          /* Index der akt. lokalen Spalte */
R,                          /* Anzahl der Spalten auf diesem Proz. */
INC1=1,
ind[maxdim],
noSV;

char
print_buf[100],
TRANST[1],TRANSN[1];


static double
*a,                         /* Lokale Anteile der Matrix */
u_buf[maxdim],              /* Buffer fuer Refl. Vektor */
d_buf[maxdim],
d[maxdim],                  /* Diagonale von B */
e[maxdim],                  /* Obere Diagonale von B */
ee[maxdim],                 /* sqr von e */
w[maxdim],                  /* Singulaerwerte */
rv1[maxdim],                /* buffer */
rv2[maxdim],rv3[maxdim],rv4[maxdim],rv5[maxdim],rv6[maxdim],
z[maxdim2];                 /* Rechte Singul. Vektoren */
zl[maxdim2];                 /* Linke Singul. Vektoren */

double
*u,                         /* Aktueller Householder Vektor */
alpha,Gamma;

void Compute_u(l) /**************************************************/
long l;
{
long len;
u       = &a[k*m+l];  /* zeigt auf das aktuelle Diagonalelement */
len     = m-l;
alpha   = -Sign(u[0]) * sqrt(ddot_(&len,u,&INC1,u,&INC1));
u[0]   -= alpha;
}

void Compute_v() /**************************************************/
{
register long i;
static   double buf[2],ALPHA,BETA;
static long len,MM,LDA,start,dummy;
```

```
if (IMF==1) start = k+1; else start = k;
len = R-start;
if (IMO==0)  buf[0] = ddot_(&len,&a[start*m+s],&m,&a[start*m+s],&m);
        else buf[0] = 0.0;

gdsum(buf,1,u_buf);

if (IMN==1)
        {
        alpha     = -Sign(a[k*m+s]) * sqrt(buf[0]);
        a[k*m+s] -= alpha;
        Gamma     = 1./(alpha*a[k*m+s]);
        buf[0]    = Gamma;
        buf[1]    = alpha; e[s+1]=alpha;
        isend(VECTOR_MSG_TYPE,buf,2*sizeof(double),ALL_NODES,NODE_PID);
        }

if (IMO==0)
        {
        MM = m-s-1; LDA = m;
        ALPHA = 1.;
        BETA  = 0.0;
        dummy = dgemv_(&TRANSN,&MM,&len,&ALPHA,&a[start*m+s+1],
               &LDA,&a[start*m+s],&m,&BETA,&d_buf[s+1],&INC1);
        }
        else for (i=s+1;i<m;i++) d_buf[i] = 0.0;

if (IMN==0)
        {
        crecv(VECTOR_MSG_TYPE,buf,2*sizeof(double),ALL_NODES,NODE_PID);
        Gamma  = buf[0];
        e[s+1] = buf[1];
        }
gdsum(&d_buf[s],m-s,u_buf);
}


void Apply_Ui_On_My_Columns(l1,l2) /******************************/
long  l1,l2;
{
static double ALPHA,BETA;
static long NN,MM,LDA,dummy;
```

```
NN=l2-l1; MM=m-s; LDA=m;
ALPHA = 1./(alpha*u[0]);
BETA  = 0.0;

/* GAMMA = 1/(alpha*u[0]) * Ai' * u  */

dummy = dgemv_(&TRANST,&MM,&NN,&ALPHA,&a[l1*m+s],
      &LDA,u,&INC1,&BETA,d_buf,&INC1);

NN=l2-l1; MM=m-s; LDA=m;
ALPHA = 1.0;

/* Ai = Ai + u * GAMMA'  */
dummy = dger_(&MM,&NN,&ALPHA,u,&INC1,d_buf,&INC1,&a[l1*m+s],&LDA);
}


void Apply_Vi_On_My_Rows(l1,l2) /***********************************/
long  l1,l2;
{
static long NN,MM,LDA,dummy;

/* Ai = Ai +  GAMMA * v'  */

NN=l2-l1; MM=m-s-1; LDA=m;
dummy = dger_(&MM,&NN,&Gamma,&d_buf[s+1],&INC1,
     &a[l1*m+s],&LDA,&a[l1*m+s+1],&LDA);
}

void main()  /*****************************************************/
{
long i;

Load_Global_Variables();
Load_Column_Data();

k=0;
if (P==0)
        {
        Compute_u(0);
        send(u,m);
        }
```

```
else
        {
        u = u_buf;
        receive(u,m);
        }

for (s=0;s<m-1;s++)
{
k=s/N+(s%N+N-1-P)/N;

if (s%N==P)      IMF=1; else IMF=0; /* I am first */
if ((s+1)%N==P) IMN=1; else IMN=0; /* I am next */
if (s>m-N+P-1)  IMO=1; else IMO=0; /* I am out */
d[s]=alpha;

if (IMO==0)
        {
if (IMF==1)  Apply_Ui_On_My_Columns(k+1,R);
        else Apply_Ui_On_My_Columns(k,R);}

if (s<m-2)
        {
        Compute_v(s);
        if (IMF==0) Apply_Vi_On_My_Rows(k,k+1);
        if (IMN==1)
                {
                Compute_u(s+1);
                send(u,m-s-1);
                }
        if (IMO==0) Apply_Vi_On_My_Rows(k+1,R);
        if (IMN==0) { u=u_buf;receive(u,m-s-1); }
        }
} /* end s */

if (L==m-1)
        {
        isend(10,&a[R*m-2],2*sizeof(double),ALL_NODES,NODE_PID);
        d[m-1]=a[R*m-1]; e[m-1]=a[R*m-2];
        }
else
        {
        crecv(10,d_buf,2*sizeof(double),ALL_NODES,NODE_PID);
        e[m-1]=d_buf[0]; d[m-1]=d_buf[1];
```

```
        }

/* Berechne Singulaerwertzerlegung der Bidiagonalmatrix */

Compute_my_SV(); /* incl. LAPACK-Aufruf: BISEKT. u. INVIT */
}
```

Literaturverzeichnis

[1] Anderson, E. ... [et al.]: LAPACK users' guide, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[2] Barth, W., Martin, R. S., Wilkinson, J. H.: Calculation of the eigenvalues of a symmetric tridiagonal matrix by the method of bisection, Numerische Mathematik 9, 1967, S. 386-393.

[3] Bischof, C., Van Loan, C. F.: The WY Representation for products of householder matrices, SIAM J. Sci. Stat. Computing, 8, 1987, S. 3-13.

[4] Berry, M., Sameh, A.: An overview of parallel algorithms for the singular value and symmetric eigenvalue problems, J. Comp. Appl. Math. 27, 1989, S. 191-213.

[5] Cullum, J. K., Willoughby, R. A.: Lanczos algorithms for large symmetric eigenvalue computations, Vol. I, Birkhäuser, Boston, Basel, Stuttgart, 1985.

[6] Demmel, J., Kahan, W.: Accurate singular values of bidiagonal matrices, SIAM J. Sci. Stat. Computing, 11, 1990, S. 873-912.

[7] Deuflhard, P., Hohmann, A.: Numerische Mathematik. Eine algorithmisch orientierte Einführung, de Gruyter, Berlin, 1991.

[8] Dongarra, J. J., van de Geijn, R.: Reduction to condensed form for the eigenvalue problem on distributed memory computers. Computer science Dept. Tech. Report CS-91-130, Univerity of Tennessee, Knoxville (LAPACK working note # 30), 1991.

[9] Dongarra, J. J., Sorensen, D. C.: Block reduction of matrices to condensed forms for eigenvalue computations, J. Comp. Appl. Math., 27, 1989, S. 215-227.

[10] Ehrhard, W.: Parallelrechnerstrukturen, Teubner, Stuttgart, 1990.

[11] Frommer, A.: Lösung linearer Gleichungssysteme auf Parallelrechnern, Vieweg, Braunschweig, 1990.

[12] Galles, M., Williams, E.: Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor, Silicon Graphics Computer Systems, Internet http://www.sgi.com/Technology/challenge_paper.html, 1995.

[13] Golub, G. A., Kahan, W.: Calculating the singular values and pseudo-inverse of a matrix, SIAM J. Numer. Anal., 2, 1965, S. 205-223.

[14] Golub, G. A.,Van Loan, C. F.: Matrix computations, 2nd edition, The Johns Hopkins University Press, Baltimore, 1989.

[15] Köckler, N., Simon, M.: Parallel singular value decomposition with cyclic storing, Parallel Computing, 17, 1991, S. 40-47.

[16] Li, T. Y., Rhee, N. H., Zeng, Z.: An efficient and accurate parallel algorithm for the singular value problem of bidiagonal matrices, Num. Math., 69, 1995, S. 283-301.

[17] Louis, A. K.: Inverse und schlecht gestellte Probleme, Teubner, Stuttgart, 1989.

[18] Parlett, B. N.: The symmetric eigenvalue problem, Prentice Hall, London, 1983.

[19] Stoer, J.: Numerische Mathematik 2, 3. Aufl., Springer, Berlin, 1990.

[20] Van der Waerden, B. L.: Algebra I, Springer, Berlin, 1971.

[21] Wilkinson, J. H., Reinsch, C.: Linear algebra, Springer, Berlin, 1971.

[22] Wilkinson, J. H.: The algebraic eigenvalue problem, Claredon Press, Oxford, 1965.