FORSCHUNGSZENTRUM JÜLICH GmbH KFA

Zentralinstitut für Angewandte Mathematik

Iterative Verfahren für dünnbesetzte Matrizen zur Lösung technischer Probleme auf massiv-parallelen Systemen

Achim Basermann

Berichte des Forschungszentrums Jülich ; 3015
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3015
D 82 (Diss. RWTH Aachen)

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
D-52425 Jülich · Bundesrepublik Deutschland
Telefon: 02461/61-6102 · Telefax: 02461/61-6103 · Telex: 833556-70 kfa d

Iterative Verfahren für dünnbesetzte Matrizen zur Lösung technischer Probleme auf massiv-parallelen Systemen

Achim Basermann

// Abstract

Iterative algorithms to solve systems of equations or eigenvalue problems with sparse coefficient matrix play an important role in numerical methods for solving discretized partial differential equations. The large size of many technical or physical applications in this area results in the need for the execution of iterative solvers on massively parallel systems. When iterative solvers are parallelized on a multiprocessor system with distributed memory, the data distribution and the communication scheme – depending on the data structures used for sparse matrices – are of the greatest importance for an efficient execution. Here, data distribution and communication schemes are presented that are based on the analysis of the indices of the non-zero matrix elements. Performance tests, using the conjugate gradient method with preconditioning, the QMR and the TFQMR algorithm to solve systems of equations, and the Lanczos method for the symmetric eigenvalue problem, were carried out on a PARAGON XP/S 10 with 140 processors. The parallel variants of the algorithms show good scaling behavior for matrices with different sparsity patterns stemming from real finite element applications.

Kurzfassung

Viele Problemstellungen der Natur- und Ingenieurwissenschaften werden durch partielle Differentialgleichungen beschrieben. Bei der Diskretisierung ergeben sich Gleichungssysteme oder Eigenwertprobleme mit dünnbesetzter Koeffizientenmatrix, die mit iterativen Verfahren effizient gelöst werden können. Heutige große technische Anwendungen erfordern den Einsatz massiv-paralleler Systeme zur Lösung derartiger Probleme, da diese Rechner sowohl über den notwendigen Speicherplatz als auch die erforderliche Rechenleistung verfügen. Auf Parallelrechnern mit verteiltem Speicher ist in Abhängigkeit von den verwendeten Datenstrukturen zur komprimierten Speicherung der Koeffizientenmatrix insbesondere das Datenverteilungs- und Kommunikationsmodell für die effiziente Ausführung iterativer Verfahren entscheidend. Hier werden Strategien zur Verteilung der Daten und Kommunikationsschemata vorgestellt, die auf der automatischen Analyse der Besetzungsstruktur der Matrix beruhen. Zur Lösung von linearen Gleichungssystemen werden die Methode der konjugierten Gradienten mit Vorkonditionierung sowie die Verfahren QMR und TFQMR verwendet. Zur Lösung des reell symmetrischen Eigenwertproblems wird ein Lanczos-Verfahren vorgestellt. Das Zeitverhalten der entwickelten parallelen Varianten dieser Verfahren wurde auf dem massiv-parallelen System mit verteiltem Speicher PARAGON XP/S 10 mit bis zu 140 Prozessoren untersucht. Die parallelen Algorithmen zeigen gute Skalierungseigenschaften für Matrizen unterschiedlicher Besetzungsstruktur, die aus realen Finite-Element-Modellen stammen.

Inhalt

1 Einführung **1**

2 Iterative Verfahren **5**

2.1 Grundlagen 5
2.2 Die Methode der konjugierten Gradienten 7
2.2.1 Algorithmus 7
2.2.2 Konvergenz 10
2.2.3 Vorkonditionierung 11
2.3 Verfahren für Gleichungssysteme mit unsymmetrischer Matrix . . 16
2.3.1 Das QMR-Verfahren 17
2.3.2 Das TFQMR-Verfahren 19
2.4 Ein Lanczos-Verfahren für das symmetrische Eigenwertproblem . . 23
2.4.1 Tridiagonalisierung 23
2.4.2 Lösung des tridiagonalen Eigenwertproblems 25

3 Speichertechniken für dünnbesetzte Matrizen **29**

3.1 Diskretisierungsgitter und Besetzungsmuster der Matrix 30
3.2 Datenstrukturen 31
3.2.1 Unstrukturierte Matrizen 31
3.2.2 Blockstrukturierte Matrizen 36
3.2.3 Bandstrukturierte Matrizen 37
3.2.4 Profil-Matrizen 42
3.3 Die Matrix-Vektor-Multiplikation 44
3.3.1 CRS-Format 44
3.3.2 SICRS-Format 45
3.3.3 ITPACK-Format 46

4 Parallelverarbeitung **49**

4.1 Parallelrechner 49
4.1.1 Systeme mit gemeinsamem Speicher 49
4.1.2 Systeme mit verteiltem Speicher 50
4.1.3 Der Rechner INTEL PARAGON XP/S 10 50
4.2 Parallele Algorithmen 51

4.2.1 Charakterisierung 51
4.2.2 Leistungsaspekte 52
4.2.3 Sprachkonzepte 52

5 Parallelisierungsstrategien für iterative Verfahren 55
5.1 Überblick über aktuelle Methoden 56
5.1.1 Zerlegung des Diskretisierungsgitters 56
5.1.2 Zerlegung der Matrix 57
5.1.3 Software-Pakete 58
5.1.4 Parallelisierende Compiler 59
5.2 Entwickelte Verfahren 59
5.3 Datenverteilung 60
5.4 Kommunikationsschemata 65
5.4.1 Analyse der Indizes 65
5.4.2 Zeilenweise Matrix-Vektor-Multiplikation 69
5.4.3 Spaltenweise Matrix-Vektor-Multiplikation 77
5.4.4 Kopplung des zeilen- und spaltenweisen Verfahrens 80
5.5 Integration in eine funktionale Programmiersprache 82

6 Ergebnisse 87
6.1 Verwendete Matrizen 87
6.1.1 Symmetrische Matrizen 88
6.1.2 Unsymmetrische Matrizen 91
6.2 Matrix-Vektor-Multiplikation 93
6.2.1 Speichertechniken 93
6.2.2 Kommunikationsschemata 98
6.2.3 Überlappung 99
6.2.4 Bandbreitenreduktion 99
6.2.5 Skalierungseigenschaften 101
6.3 Iterative Methoden 102
6.3.1 Startwerte 103
6.3.2 Das CG-Verfahren 104
6.3.3 Die Verfahren QMR und TFQMR 117
6.3.4 Die Lanczos-Methode 123
6.4 Algorithmische Skelette in einer funktionalen Sprache 128
6.4.1 Vorverarbeitung 129
6.4.2 Matrix-Vektor-Multiplikation 129

7 Zusammenfassung und Ausblick 131

Literaturverzeichnis 135

A Symbole und Abkürzungen 147

Abbildungsverzeichnis

1.1 FE-Diskretisierungsgitter und dünnbesetzte Koeffizientenmatrix . 2

3.1 FE-Gitter aus Quadern 30
3.2 Datenstruktur des CRS-Formats 31
3.3 Ein Beispiel für die CRS-Speichertechnik 32
3.4 Ein Beispiel für die CCS-Speichertechnik 33
3.5 Datenstruktur des SICRS-Formats 34
3.6 Ein Beispiel für die SICRS-Speichertechnik 35
3.7 Datenstruktur des BCRS-Formats 36
3.8 Datenstruktur des DIA-Formats 37
3.9 Ein Beispiel für die DIA-Speichertechnik 38
3.10 Datenstruktur des Streifen-Formats 39
3.11 Ein Beispiel für die Streifen-Speichertechnik 39
3.12 Datenstruktur des ITPACK-Formats 40
3.13 Ein Beispiel für die ITPACK-Speichertechnik 41
3.14 Datenstruktur des modifizierten ITPACK-Formats 41
3.15 Datenstruktur des JDS-Formats 42
3.16 Ein Beispiel für die JDS-Speichertechnik 42
3.17 Struktur einer unsymmetrischen Profil-Matrix 43
3.18 Die Matrix-Vektor-Multiplikation im CRS-Format 45
3.19 Multiplikation mit der Transponierten, CCS-Format 45
3.20 Die Matrix-Vektor-Multiplikation im SICRS-Format 46
3.21 Matrix-Vektor-Multiplikation, ITPACK-Format, spaltenorientiert 46
3.22 Matrix-Vektor-Multiplikation, ITPACK-Format, zeilenorientiert . 47

5.1 CRS-Speicherschema 63
5.2 Datenverteilung, Kriterium „Gleich viele Zeilen“ 64
5.3 Datenverteilung, Kriterium „Gleich viele Nichtnull-Elemente“ . . 64
5.4 Datenverteilung für $\xi = 16$ und $s = 2$ 64
5.5 Ordnen der Spaltenindizes 67
5.6 Datenstruktur zur Analyse der Spaltenindizes 67
5.7 Analyse der Spaltenindizes 68
5.8 Datenstruktur für das Senden 68
5.9 Speicherung der Indizes für das Senden 68

5.10 Umordnung in Blöcke 69
5.11 Verteilte Datenstruktur bei blockweiser Sortierung 70
5.12 Datenaustausch bei Block-Sortierung 70
5.13 Ablauf eines parallelen iterativen Verfahrens 72
5.14 Zeilenweise Matrix-Vektor-Multiplikation bei Block-Sortierung . . 73
5.15 Datenverteilung vor und nach der Neuverteilung der Blöcke 75
5.16 Verteilte Datenstruktur bei Neuverteilung der Blöcke 76
5.17 Datenaustausch bei Neuverteilung der Blöcke 76
5.18 Zeilenweises Verfahren bei Neuverteilung der Blöcke 78
5.19 Datenaustausch der spaltenweisen Matrix-Vektor-Multiplikation . 80
5.20 Kopplung der unabhängigen Matrix-Vektor-Multiplikationen . . . 81
5.21 Umordnung in zwei Blöcke und Index-Transformation 84
5.22 Ein funktionales Programm 85

6.1 Besetzungsmuster 92
6.2 Ausführungszeiten beim ITPACK-Schema 94
6.3 Vergleich der Ausführungszeiten beim ITPACK- und CRS-Schema 94
6.4 Vergleich der Ausführungszeiten beim CRS- und SICRS-Schema . 96
6.5 Ausführungszeiten für verschiedene Kommunikationsschemata . . 98
6.6 Einfluß der Überlappung 99
6.7 Speedup-Werte ohne und mit Bandbreitenreduktion 100
6.8 Speedup-Werte ohne und mit Bandbreitenreduktion 101
6.9 Speedup-Verhalten der Matrix-Vektor-Multiplikation 102
6.10 Speedup-Verhalten der Matrix-Vektor-Multiplikation 103
6.11 Ausführungszeiten für verschiedene CG-Verfahren 106
6.12 Datenverteilung, CG-Verfahren 107
6.13 Datenverteilung, CG-Verfahren 107
6.14 Einfluß der Überlappung, CG-Verfahren 109
6.15 Speedup-Verhalten des CG-Verfahrens 110
6.16 Speedup-Verhalten des CG-Verfahrens 111
6.17 Speedup-Verhalten ohne und mit Kommunikationsprozessor . . . 111
6.18 Ausführungszeiten ohne und mit Vorkonditionierung 113
6.19 Ausführungszeiten ohne und mit Vorkonditionierung 113
6.20 Speedup-Verhalten bei Vorkonditionierung 114
6.21 Speedup-Verhalten bei Vorkonditionierung 115
6.22 Ausführungszeiten für Vorverarbeitung und CG-Verfahren 116
6.23 Datenverteilung, QMR-Methode 118
6.24 Datenverteilung, TFQMR-Methode 119
6.25 Ausführungszeiten für die Methoden QMR, TFQMR und CG . . 120
6.26 QMR-Methode ohne und mit Kopplung 121
6.27 Speedup-Werte des QMR-Verfahrens 122
6.28 Speedup-Werte des TFQMR-Verfahrens 122
6.29 Ausführungszeiten für verschiedene Lanczos-Methoden 124

6.30 Datenverteilung, Lanczos-Tridiagonalisierung 125
6.31 Ausführungszeiten der gesamten Lanczos-Methode 126
6.32 Ausführungszeiten der gesamten Lanczos-Methode 126
6.33 Ausführungszeiten der gesamten Lanczos-Methode 127
6.34 Speedup-Verhalten der Lanczos-Methode 128
6.35 Speedup-Werte des Skeletts zur Vorverarbeitung 129
6.36 Speedup-Werte des Skeletts zur Matrix-Vektor-Multiplikation . . 130

Tabellenverzeichnis

5.1 Näherungen des Parameters der Datenverteilung 62

6.1 Kenngrößen der Matrizen **ICG2D** und **ICG3D** 88
6.2 Kenngrößen der Matrix **ISR** 88
6.3 Kenngrößen der Matrix **PRESS** 89
6.4 Kenngrößen der Matrizen **BCSSTK17** und **BCSSTK18** 90
6.5 Kenngrößen der Matrix **LAPLACE** 90
6.6 Brandbreiten ohne und mit RCM-Umordnung 91
6.7 Kenngrößen der Matrizen **ORSREG1** und **WATT1** 93
6.8 Speicherbedarf bei verschiedenen Formaten der Matrizen 97
6.9 Iterationsschritte des CG-Verfahrens 105
6.10 Iterationsschritte der Verfahren QMR, TFQMR und CG 117
6.11 Anzahl der ermittelten Eigenwerte 123

Kapitel 1

Einführung

Viele Problemstellungen der Natur- und Ingenieurwissenschaften werden durch Differentialgleichungen beschrieben. Da eine geschlossene Lösung für dieses mathematische Modell häufig nicht angegeben werden kann, wird das Problem diskretisiert und mit numerischen Methoden approximativ gelöst.

Bei der Diskretisierung gewöhnlicher oder partieller Differentialgleichungen ergeben sich je nach Diskretisierungsverfahren Gleichungssysteme oder Eigenwertprobleme mit dünnbesetzten Koeffizientenmatrizen unterschiedlicher Besetzungsstruktur. Bei FE-Verfahren (*Finite Element*) sind die entstehenden Matrizen weitgehend unstrukturiert, während Differenzenverfahren im allgemeinen zu Matrizen mit regulärem Besetzungsmuster führen, z. B. mit Band- oder Blockstruktur.

Eigenwertprobleme treten bei der Analyse elastischer Strukturen auf [42] [81] [118]. Ein weiteres Anwendungsgebiet ist die Stabilitätsanalyse dynamischer Systeme, z. B. elektrischer Netzwerke. Ferner sind große Eigenwertprobleme in der Chemie zu finden, insbesondere in der Molekulardynamik und der Quantenchemie.

Eine Anwendung der FE-Methode ist die Modellierung des Strömungsverlaufs an Tragflächen. Abbildung 1.1 zeigt das Diskretisierungsgitter eines zweidimensionalen Modells und das Besetzungsmuster der zugehörigen Koeffizientenmatrix bei einer vorgegebenen Numerierung der Gitterpunkte [9]. Das Modell stammt aus einer Untersuchung des Research Institute for Applications of Computer Science der NASA. Das Strömungsproblem wird durch nichtlineare partielle Differentialgleichungen beschrieben; berechnet werden u. a. der hydrodynamische Druck und zwei Komponenten der Strömungsgeschwindigkeit.

Das Diskretisierungsgitter in Abbildung 1.1 ist aus Dreieckselementen aufgebaut. Die Eckpunkte der Dreiecke bilden die Gitterpunkte; insgesamt sind 4253 Gitterpunkte vorhanden. Zum Tragflächenquerschnitt hin wird das Gitter feiner, da hier eine hohe Genauigkeit der Berechnungen erforderlich ist. Jeder Schritt der nichtlinearen Iteration des FE-Modells erfordert die Lösung eines linearen Gleichungssystems mit dünnbesetzter Koeffizientenmatrix. Die Verteilung der

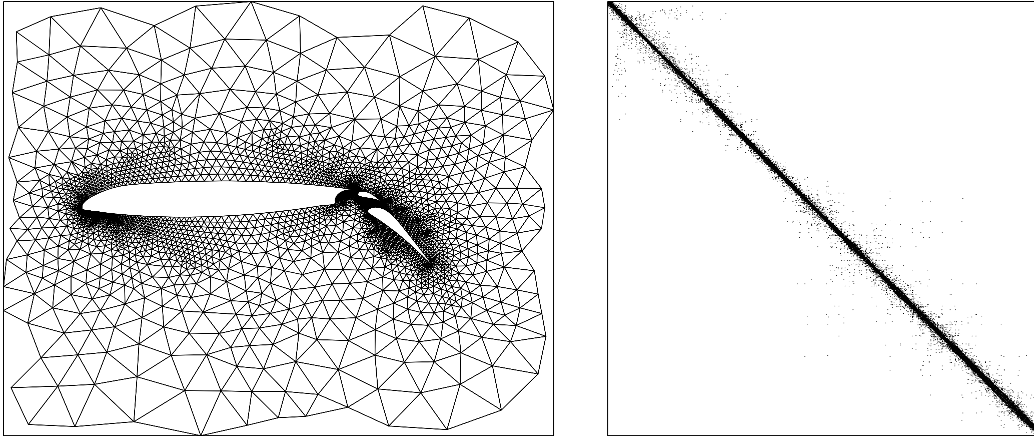


Abbildung 1.1: Links: FE-Diskretisierungsgitter. Rechts: Dünnbesetzte Koeffizientenmatrix.

Nichtnull-Elemente der Matrix ist in Abbildung 1.1 rechts dargestellt. Die Matrix besitzt die Ordnung 4253 und insgesamt 28831 Nichtnull-Elemente. Der Besetzungsgrad beträgt somit lediglich 0.16%.

Eine Matrix kann als Beschreibung der linearen Abhängigkeiten zwischen den Komponenten eines mathematischen Modells betrachtet werden. Ein Modell aus n Bestandteilen führt zu einer $n \times n$ Matrix. In Abbildung 1.1 entsprechen die unbekannten Größen an den Gitterpunkten den Komponenten des FE-Modells. Die Lösung eines linearen Gleichungssystems mit vollbesetzter Matrix erfordert auf einem Rechner ca. n^2 Speicherworte und eine Rechenzeit, die proportional zu n^3 ist. Bei vollbesetzter Matrix erreichen Modelle mit einigen tausend Komponenten bereits die Grenzen herkömmlicher Rechnersysteme.

In vielen großen Modellen ist eine einzelne Komponente jedoch nur von wenigen anderen Komponenten abhängig. In Abbildung 1.1 gehen in die Berechnungen pro Gitterpunkt lediglich die Werte an den Punkten ein, die mit dem betrachteten Punkt direkt durch eine Dreiecksseite verbunden sind. Pro Punkt besteht eine Verbindung zu drei bis neun anderen Punkten. Modelle mit dieser Eigenschaft und die resultierenden Matrizen werden dünnbesetzt (*sparse*) genannt; der bei weitem überwiegende Teil der Matrixelemente besitzt den Wert Null. J. H. Wilkinson, ein Begründer der modernen numerischen linearen Algebra, sieht als Merkmal einer dünnbesetzten Matrix die Eigenschaft [9]: „It has enough zeros that it pays to take advantage of them.“ Die Nutzung der Besetzungsstruktur der Matrix und die Effizienz von Algorithmen für dünnbesetzte Matrizen auf einem Rechnersystem sind eng miteinander verknüpft. Durch die Vermeidung von Operationen mit Null-Elementen und die Speicherung nur oder hauptsächlich der Nichtnull-Elemente werden sowohl Speicherplatz als auch Rechenzeit gespart. Die in dieser Arbeit betrachteten iterativen Verfahren zur

Lösung von Gleichungssystemen und Eigenwertproblemen begünstigen die Nutzung der dünnbesetzten Struktur der Matrix; Speicherbedarf und Operationen der Algorithmen sind proportional zur Anzahl der Nichtnull-Elemente.

Da in heutigen großen technischen Anwendungen Modelle mit Millionen von Komponenten erforderlich sind, werden die Grenzen herkömmlicher Rechner sowohl an Speicherplatz als auch an Rechenleistung überschritten. Massiv-parallele Rechnersysteme mit verteiltem Speicher hingegen ermöglichen die Modellierung derartiger Probleme; Speicherplatz und Rechenleistung dieser Parallelrechner skalieren mit der Anzahl der Prozessoren. Die parallele Bearbeitung von iterativen Algorithmen für dünnbesetzte Matrizen auf massiv-parallelen Systemen ist eine große Herausforderung heutiger Forschung.

Die Hauptarbeit in iterativen Verfahren zur Lösung von linearen Gleichungssystemen oder Eigenwertproblemen besteht in der Berechnung von Matrix-Vektor-Produkten und Vektor-Vektor-Operationen; gewöhnlich ist die Matrix-Vektor-Multiplikation die aufwendigste Operation. Bei der Durchführung dieser Operation wird der Zugriff auf den Vektor durch die Besetzungsstruktur und die Speichertechnik der Matrix bestimmt.

Auf Parallelrechnern mit verteiltem Speicher ist in Abhängigkeit von den verwendeten Datenstrukturen insbesondere das Datenverteilungs- und Kommunikationsschema für die effiziente Ausführung iterativer Verfahren entscheidend. In den im Rahmen dieser Arbeit vorgestellten Untersuchungen werden Strategien zur Datenverteilung und Kommunikationsschemata vorgeschlagen, die auf der Analyse der Besetzungsstruktur der Matrix beruhen. Zur Balancierung der Rechenlast berücksichtigt das Datenverteilungsschema neben den Matrix-Vektor-Multiplikationen auch die übrigen Operationen der iterativen Verfahren. Ferner werden verschiedene Methoden zur Umordnung der Matrixelemente beschrieben, die die überlappte Durchführung von lokalen Rechenoperationen und der Übermittlung nicht-lokaler Daten ermöglichen. Das Ziel ist die Reduzierung von Wartezeiten. In diesem Zusammenhang wird auf unterschiedliche Techniken zur komprimierten Speicherung der Matrixelemente eingegangen.

Datenverteilung, Kommunikationsschema und Umordung der Matrixelemente werden in einem Vorverarbeitungsschritt ermittelt. Sie werden in jedem Schritt eines iterativen Verfahrens genutzt. Darüber hinaus sind diese Schemata solange gültig, bis sich die Besetzungsstruktur der Matrix ändert; sie können z.B. in jedem Zeitschritt eines zeitabhängigen Modells oder in jedem Schritt eines nichtlinearen Problems, das durch Linearisierung gelöst wird, verwendet werden.

Zur Lösung von linearen Gleichungssystemen mit symmetrischer positiv definiter Matrix wird die Methode der konjugierten Gradienten (*CG*) betrachtet [77]. Die Skalierung mit der Diagonalen und die polynomiale Vorkonditionierung werden zur Konvergenzbeschleunigung des Verfahrens eingesetzt [11] [104]. Für Gleichungssysteme mit unsymmetrischer Matrix werden die Methoden QMR (*Quasi-Minimal Residual*) [68] und TFQMR (*Transpose-Free Quasi-Minimal Residual*) [66] verwendet. Zur Lösung des reell symmetrischen Eigenwertproblems

wird ein Lanczos-Verfahren vorgestellt [44] [70] [105].

Das Zeitverhalten der entwickelten parallelen Verfahren wird auf dem massiv-parallelen System mit verteiltem Speicher PARAGON XP/S 10 der Forschungszentrum Jülich GmbH untersucht. Die verwendeten dünnbesetzten Matrizen stammen u.a. aus Projekten des Forschungszentrums.

Am Institut für Chemie und Dynamik der Geosphäre wird das Verhalten von Schadstoffen in geologischen Systemen mit einem FE-Modell simuliert [140] [145]. In diesem Modell aus dem Bereich der Umwelttechnik erfordert die Berechnung des Wasserflusses die Lösung symmetrischer positiv definiter Gleichungssysteme, während im Stofftransportmodell Gleichungssysteme mit unsymmetrischer Matrix auftreten. Die Integration der in dieser Arbeit untersuchten Verfahren ist ein wichtiger Schritt zur erfolgreichen Durchführung dieses Projekts.

Am Institut für Sicherheitsforschung und Reaktortechnik der Forschungszentrum Jülich GmbH werden u.a. Spannungen in Materialien infolge von Erwärmung untersucht. Der Einbau des entwickelten parallelen CG-Verfahrens in das verwendete FE-Modell [8] ist Gegenstand einer Zusammenarbeit.

Am Institut für Festkörperforschung des Forschungszentrums sollen die entwickelten Methoden zur Lösung des symmetrischen Eigenwertproblems im Bereich der Molekulardynamik eingesetzt werden. Erste Untersuchungen zur Eignung der Verfahren sind durchgeführt.

In Zusammenarbeit mit dem Lehrstuhl für Informatik II der RWTH Aachen wird die Integration einiger Methoden für dünnbesetzte Matrizen in den Sprachumfang einer funktionalen Programmiersprache untersucht. Ziel dieser Arbeiten ist es, die Programmierung paralleler Algorithmen für dünnbesetzte Matrizen auf hohem Abstraktionsniveau zu ermöglichen.

Im weiteren ist die Arbeit wie folgt aufgebaut. In Kapitel 2 werden iterative Verfahren zur Lösung von Gleichungssystemen und Eigenwertproblemen vorgestellt. Anschließend wird in Kapitel 3 auf Speichertechniken für dünnbesetzte Matrizen eingegangen, bevor in Kapitel 4 grundlegende Begriffe der Parallelverarbeitung erläutert werden. In Kapitel 5 werden dann die im Rahmen dieser Arbeit entwickelten Parallelisierungsstrategien für iterative Verfahren beschrieben und in Kapitel 6 numerische sowie Performance-Ergebnisse auf dem massiv-parallelen System PARAGON XP/S 10 präsentiert. Den Abschluß bildet Kapitel 7 mit einer kurzen Zusammenfassung der Resultate dieser Arbeit und einem Ausblick auf weitere mögliche Einsatzgebiete der entwickelten Methoden.

Kapitel 2

Iterative Verfahren

In diesem Kapitel werden iterative Lösungsmethoden für lineare Gleichungssysteme und Eigenwertprobleme beschrieben, die in technischen Anwendungen häufig eingesetzt werden. Die Verfahren sind insbesondere für große dünnbesetzte Matrizen geeignet. Nach einer kurzen Beschreibung grundsätzlicher Eigenschaften iterativer Verfahren wird die Methode der konjugierten Gradienten (*CG*) zur Lösung von linearen Gleichungssystemen mit symmetrischer positiv definiter Koeffizientenmatrix vorgestellt [77] [104]. Für dieses Verfahren werden zwei Methoden zur Vorkonditionierung betrachtet, die Skalierung mit der Diagonalen [104] und die polynomiale Vorkonditionierung [11]. Anschließend werden die Verfahren QMR (*Quasi-Minimal Residual*) [68] und TFQMR (*Transpose-Free Quasi-Minimal Residual*) [66] beschrieben, die zur Lösung von linearen Gleichungssystemen mit regulärer unsymmetrischer Koeffizientenmatrix geeignet sind. Zur Lösung des reellen symmetrischen Eigenwertproblems wird schließlich ein Lanczos-Verfahren [44] [70] [105] vorgestellt.

2.1 Grundlagen

Zur Lösung von linearen Gleichungssystemen $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ mit $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ werden direkte und iterative Verfahren verwendet. Klassische direkte Methoden wie die LU-Faktorisierung und die Cholesky-Zerlegung sind u. a. in [70] beschrieben. Das letztere Verfahren ist für symmetrische positiv definite Koeffizientenmatrizen $\boldsymbol{A}$ geeignet; beide Verfahren beruhen auf der Gauß-Elimination. In technischen Anwendungen werden direkte Methoden gewöhnlich bei dichtbesetzten oder bandstrukturierten Koeffizientenmatrizen eingesetzt. Ist die Koeffizientenmatrix dünnbesetzt und unregelmäßig strukturiert, so führen direkte Methoden in der Regel zu einem Auffüllen (*Fill-in*) der mit Null besetzten Stellen der Matrix mit Nichtnull-Elementen. Spezielle Techniken, die Fill-in bei direkten Methoden reduzieren, sind in [53] beschrieben.

Im Gegensatz zu den direkten Methoden verwenden iterative Verfahren le-

diglich die von Null verschiedenen Elemente der Matrix, da auf die Matrix im wesentlichen bei der Berechnung von Matrix-Vektor-Produkten zugegriffen wird. Dies führt zu minimalem Speicherbedarf. Klassische iterative Verfahren sind die Jacobi-, die Gauß-Seidel-Iteration, das SOR-Verfahren (*Successive Over-Relaxation*) und die semi-iterative Tschebyscheff-Methode [70] [104]. Ein historischer Überblick über die Entwicklung iterativer Methoden wird in [146] gegeben. Iterative Verfahren generieren eine Folge von Näherungslösungen $\{x^{(i)}\}$. Ein wichtiges Bewertungskriterium für iterative Methoden ist die Konvergenzgeschwindigkeit, d. h. wie schnell die Iterierten $x^{(i)}$ gegen die exakte Lösung x^* konvergieren.

In den folgenden Abschnitten werden die Verfahren CG, QMR und TFQMR beschrieben, denen Optimierungsansätze zugrunde liegen. Die Lösung des linearen Gleichungssystems wird dabei in eine äquivalente Minimierungsaufgabe überführt. Die Bestimmung des Minimums erfolgt iterativ nach der Vorschrift

$$x^{(i+1)} = x^{(i)} + \gamma_i d^{(i)} \tag{2.1}$$

mit der Schrittweite γ_i und der Suchrichtung $d^{(i)}$.

Die Lösung des reellen symmetrischen Eigenwertproblems geschieht gewöhnlich in zwei Schritten. Zunächst wird die $n \times n$ Matrix $\boldsymbol{A}$ auf Tridiagonalgestalt reduziert, dann werden die Eigenwerte der Tridiagonalmatrix ermittelt. Das in dieser Arbeit betrachtete Lanczos-Verfahren erzeugt eine Folge von Tridiagonalmatrizen, deren Eigenwerte Näherungen für Eigenwerte der ursprünglichen Matrix sind. Die Lanczos-Tridiagonalisierung und das CG-Verfahren sind theoretisch eng verknüpft [44] [105]. In besonderem Maße eignet sich die Lanczos-Methode für große dünnbesetzte Matrizen mit unregelmäßiger Besetzungsstruktur, da Zugriffe auf die Matrixelemente lediglich bei der Berechnung von Matrix-Vektor-Produkten auftreten. Andere übliche Verfahren wie die Householder- oder die Givens-Transformation hingegen zerstören die Besetzungsstruktur; im Verlauf der Reduktion entstehen gewöhnlich große dichtbesetzte Matrizen [70]. Die letzteren Verfahren werden daher meist bei Eigenwertproblemen mit dichtbesetzten Matrizen oder Bandmatrizen verwendet.

Sowohl die Verfahren CG, QMR und TFQMR als auch die Lanczos-Methode zur Lösung des reellen symmetrischen Eigenwertproblems gehören der Klasse der Krylov-Teilraum-Verfahren an. Diese Methoden lassen sich durch das Aufspannen derjenigen Teilräume beschreiben, die die Iterationsvektoren enthalten. Der i-te von $z \in \mathbb{R}^n$ und $\boldsymbol{B} \in \mathbb{R}^{n \times n}$ erzeugte Krylov-Teilraum ist durch

$$\mathcal{K}(\boldsymbol{B}, z, i) = \operatorname{span}\left\{z, \boldsymbol{B}z, \boldsymbol{B}^2 z, \ldots, \boldsymbol{B}^{i-1} z\right\}$$

gegeben.

Für die Iterierten $x^{(i)} \in \mathbb{R}^n$ der betrachteten Verfahren zur Lösung linearer Gleichungssysteme gilt bei vorgegebenem Startvektor $x^{(0)}$

$$x^{(i)} \in x^{(0)} + \mathcal{K}(\boldsymbol{A}, g^{(0)}, i),$$

wobei $g^{(0)} = \boldsymbol{A}x^{(0)} - b$ das zum Startvektor gehörige Residuum ist [66] [68] [70].

Für die Vektoren $q^{(j)}$, $j = 1, \ldots, i$, der Lanczos-Methode (s. 2.4.1) ergibt sich durch Optimierung des Rayleigh-Quotienten [70] [105]

$$\text{span}\left\{q^{(1)}, \ldots, q^{(i)}\right\} = \mathcal{K}(\boldsymbol{A}, q^{(1)}, i).$$

Die Methode zur Ermittlung einer Basis für die Krylov-Teilräume charakterisiert ein Krylov-Teilraum-Verfahren. Die Herleitung der Basen für die hier betrachteten Iterationen ist in [66], [68], [70] und [105] ausführlich beschrieben.

2.2 Die Methode der konjugierten Gradienten

Hestenes und Stiefel [77] verwendeten die Methode der konjugierten Gradienten 1952 erstmals zur Lösung von Gleichungssystemen $\boldsymbol{A}x = b$. Das von ihnen entwickelte Verfahren setzt symmetrische positiv definite Koeffizientenmatrizen voraus. Matrizen mit dieser Eigenschaft besitzen positive Eigenwerte und erfüllen ferner $\boldsymbol{A} = \boldsymbol{A}^T$. Aufgrund des günstigen Konvergenzverhaltens wird die Methode heutzutage in vielen technischen Anwendungen eingesetzt, insbesondere in der Modellierung durch FE-Verfahren. In die im einführenden Kapitel erwähnten FE-Modelle aus der Umwelttechnik und der Strukturmechanik wurde das CG-Verfahren jeweils erfolgreich integriert. Es erwies sich als den bisher verwendeten SOR- und Cholesky-Verfahren deutlich überlegen.

Die theoretischen Eigenschaften des CG-Verfahrens werden in [73] ausführlich behandelt. Kurzbeschreibungen sind in [13], [70] und [104] zu finden. Ein Überblick über die Artikel, die in den ersten 25 Jahren seit der Entwicklung der Methode erschienen sind, wird in [69] gegeben.

2.2.1 Algorithmus

Der Algorithmus des CG-Verfahrens basiert auf der Lösung einer quadratischen Optimierungsaufgabe. Die Lösung des linearen Gleichungssystems $\boldsymbol{A}x = b$ wird in die äquivalente freie Optimierungsaufgabe, das lokale und gleichzeitig globale Minimum der Funktion

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

zu finden, überführt. Dieses Optimierungsproblem wird durch Einführung einer Schrittweite und einer Suchrichtung nach (2.1) approximativ gelöst. Da der negative Gradient von $F(x)$ die Richtung des steilsten Abstiegs angibt, wird zur Bestimmung der Suchrichtung $d^{(i)}$ das negative Residuum $-g^{(i)} = b - \boldsymbol{A}x^{(i)}$ der i-ten Stufe verwendet. Zusätzlich werden $d^{(0)}, \ldots, d^{(i)}$ paarweise $\boldsymbol{A}$-konjugiert, d. h. es wird dafür gesorgt, daß

$$d^{(j)^T} \boldsymbol{A} d^{(k)} = 0 \quad \forall\ 0 \leq j < k \leq i$$

Algorithmus 2.1. Das ursprüngliche CG-Verfahren

Wähle $x^{(0)} \in I\!R^n$ *beliebig*

$$\begin{aligned} g^{(0)} &= \boldsymbol{A}x^{(0)} - b \\ d^{(0)} &= -g^{(0)} \end{aligned}$$

$i = 0, 1, \ldots$

$$\begin{aligned} \gamma_i &= \frac{g^{(i)T} g^{(i)}}{d^{(i)T} \boldsymbol{A} d^{(i)}} \\ x^{(i+1)} &= x^{(i)} + \gamma_i d^{(i)} \\ g^{(i+1)} &= g^{(i)} + \gamma_i \boldsymbol{A} d^{(i)} \\ \delta_i &= \frac{g^{(i+1)T} g^{(i+1)}}{g^{(i)T} g^{(i)}} \\ d^{(i+1)} &= -g^{(i+1)} + \delta_i d^{(i)} \end{aligned}$$

bis $\|g^{(i+1)}\|_2 \leq \epsilon_{\mathrm{r}}$

mit $d^{(0)}, \ldots, d^{(i)} \in \mathbb{R}^n \backslash \{0\}$ erfüllt ist. Die Bezeichnung *Methode der konjugierten Gradienten* (*Conjugate Gradients*) ist durch dieses Vorgehen begründet. Die auf diese Weise ermittelten Suchrichtungen und die zugehörigen Residuen sind linear unabhängig; daher sind in exakter Arithmetik höchstens n CG-Schritte erforderlich, um die exakte Lösung x^* zu erhalten. Diese Eigenschaft kennzeichnet das CG-Verfahren als direkte Methode. Aufgrund von Rundungsfehlern ist es jedoch möglich, daß mehr als n Schritte benötigt werden; in endlicher Arithmetik verhält sich das CG-Verfahren wie eine iterative Methode. Bei technischen Anwendungen sind gewöhnlich weit weniger als n Schritte erforderlich, um eine gute Approximation von x^* zu erhalten.

In Algorithmus 2.1 ist der Verlauf des ursprünglichen CG-Verfahrens von Hestenes und Stiefel dargestellt. In jedem Iterationsschritt werden die Vektoren $x^{(i)}$, $g^{(i)}$ und $d^{(i)}$ gebildet. $x^{(i)}$ ist die Näherung des Lösungsvektors, $g^{(i)}$ das Residuum; $d^{(i)}$ gibt die Richtung an, in der die aktuelle Näherung des Lösungsvektors bestimmt wird.

Bei den meisten dünnbesetzten Matrizen aus technischen Anwendungen besteht die Hauptarbeit jedes Iterationsschritts in der Berechnung des Matrix-Vektor-Produkts $\boldsymbol{A}d^{(i)}$. Ferner werden zwei Skalarprodukte und drei Vektoradditionen berechnet. Der Wert des dritten Skalarprodukts kann aus dem vorherigen Schritt übernommen werden.

Die Iteration wird abgebrochen, wenn die euklidische Norm des Residuums kleiner oder gleich der vorgegebenen Genauigkeit $\epsilon_{\mathbf{r}}$ ist. Ein anderes Abbruchkriterium, das die Maximumnorm der Differenz der letzten beiden Näherungen des Lösungsvektors verwendet, läßt sich folgendermaßen bestimmen:

$$\|x^{(i+1)} - x^{(i)}\|_\infty = \max_{j=1,\ldots,n} |x_j^{(i+1)} - x_j^{(i)}| \leq \epsilon_{\mathbf{m}}.$$

Die maximale skalierte absolute Differenz (2.2) der Komponenten der letzten beiden Näherungen des Lösungsvektors kann zur Ermittlung eines weiteren Abbruchkriteriums genutzt werden:

$$\max_{j=1,\ldots,n} 2\frac{|x_j^{(i+1)} - x_j^{(i)}|}{|x_j^{(i+1)}| + |x_j^{(i)}|} \leq \epsilon_{\mathbf{s}}. \tag{2.2}$$

In dem Fall $x_j^{(i+1)} = 0$ und $x_j^{(i)} = 0$ wird die skalierte absolute Differenz der entsprechenden Komponenten zu 0 gesetzt. Die Wahl eines geeigneten Abbruchkriteriums hängt vom jeweiligen Anwendungsfall ab. Abbruchkriterien für iterative Verfahren werden ausführlich in [17] diskutiert.

Aykanat et al. [14] schlugen 1990 ein modifiziertes CG-Verfahren vor, das günstigere Parallelisierungseigenschaften als die ursprüngliche Methode besitzt. Das modifizierte CG-Verfahren unterscheidet sich vom ursprünglichen im wesentlichen dadurch, daß in Algorithmus 2.2 die Konstanten γ_i und δ_i und damit alle Skalarprodukte zu Beginn jedes Iterationsschritts berechnet werden. Bei paralleler Bearbeitung jedes Schritts auf einem Parallelrechner mit verteiltem Speicher können daher die lokalen Werte der Skalarprodukte zur Bestimmung der globalen Werte in einer Nachricht pro Prozessor zusammengefaßt werden [14].

Bei numerischen Tests mit einigen großen dünnbesetzten Matrizen, die im Rahmen dieser Arbeit durchgeführt wurden, erwies sich Algorithmus 2.2 im Vergleich zu Algorithmus 2.1 als weniger robust, d. h. Rundungsfehler beeinflußten das Konvergenzverhalten des modifizierten Verfahrens in größerem Maße als das der ursprünglichen Methode. Dieser Nachteil wird vermieden, wenn das Skalarprodukt ${g^{(i)}}^T g^{(i)}$ in jedem Iterationsschritt neu berechnet und nicht der Wert von $\delta_{i-1}\, {g^{(i-1)}}^T g^{(i-1)}$ aus dem vorhergehenden Schritt verwendet wird. Dies erfordert die Berechnung eines zusätzlichen Skalarprodukts zu Beginn jedes Iterationsschritts, wirkt sich jedoch nicht auf die günstigeren Parallelisierungseigenschaften von Algorithmus 2.2 gegenüber Algorithmus 2.1 aus. Mit dieser zusätzlichen Berechnung zeigt das modifizierte CG-Verfahren für alle untersuchten Matrizen das gleiche Konvergenzverhalten wie die ursprüngliche Methode.

Weitere Untersuchungen zur Einsparung von Synchronisationspunkten in der CG-Iteration und in verwandten Verfahren sind in [41], [49], [58], [101] und [113] zu finden.

Algorithmus 2.2. Das modifizierte CG-Verfahren

Wähle $x^{(0)} \in I\!R^n$ *beliebig*

$$\begin{aligned} g^{(0)} &= \boldsymbol{A}x^{(0)} - b \\ d^{(0)} &= -g^{(0)} \end{aligned}$$

$i = 0, 1, \ldots$

$$\begin{aligned} \gamma_i &= \frac{g^{(i)^T} g^{(i)}}{d^{(i)^T} \boldsymbol{A} d^{(i)}} \\ \delta_i &= \frac{\gamma_i (\boldsymbol{A}d^{(i)})^T \boldsymbol{A}d^{(i)}}{d^{(i)^T} \boldsymbol{A}d^{(i)}} - 1 \\ g^{(i+1)^T} g^{(i+1)} &= \delta_i \, g^{(i)^T} g^{(i)} \\ x^{(i+1)} &= x^{(i)} + \gamma_i d^{(i)} \\ g^{(i+1)} &= g^{(i)} + \gamma_i \boldsymbol{A} d^{(i)} \\ d^{(i+1)} &= -g^{(i+1)} + \delta_i d^{(i)} \end{aligned}$$

bis $\|g^{(i+1)}\|_2 \leq \epsilon_{\mathrm{r}}$

2.2.2 Konvergenz

Das Konvergenzverhalten iterativer Methoden ist schwer vorherzusagen; es kann sich von Anwendungsfall zu Anwendungsfall deutlich unterscheiden. Gewöhnlich können jedoch nützliche Schranken ermittelt werden.

Die Fehlerschranken für die Methode der konjugierten Gradienten werden unter Verwendung der Konditionszahl κ_2 der Koeffizientenmatrix $\boldsymbol{A}$ bezüglich der Spektralnorm angegeben [17] [104]. Für symmetrische positiv definite Matrizen ist κ_2 durch

$$\kappa_2(\boldsymbol{A}) = \|\boldsymbol{A}\|_2 \|\boldsymbol{A}^{-1}\|_2 = \frac{\lambda_{\max}(\boldsymbol{A})}{\lambda_{\min}(\boldsymbol{A})}$$

gegeben. Hierbei sind $\lambda_{\max}(\boldsymbol{A})$ und $\lambda_{\min}(\boldsymbol{A})$ der größte und kleinste Eigenwert der Matrix $\boldsymbol{A}$. Mit

$$\eta = \frac{\sqrt{\kappa_2(\boldsymbol{A})} - 1}{\sqrt{\kappa_2(\boldsymbol{A})} + 1}$$

läßt sich die Konvergenzrate des CG-Verfahrens wie folgt abschätzen:

$$\|x^{(i)} - x^*\|_2 \leq 2\,\eta^i \sqrt{\kappa_2(\boldsymbol{A})}\, \|x^{(0)} - x^*\|_2. \tag{2.3}$$

Für $\kappa_2(\boldsymbol{A}) = 1$ ist $\eta = 0$, und für $\kappa_2(\boldsymbol{A}) \to \infty$ gilt $\eta \to 1$, d. h. je größer $\kappa_2(\boldsymbol{A})$ ist, desto mehr nähert sich η dem Grenzwert 1 und desto langsamer konvergiert i. a. das Verfahren. Die Abschätzung (2.3) ist sinnvoll, wenn $\boldsymbol{A}$ viele verschiedene Eigenwerte besitzt, die breit gestreut sind. Sind die Eigenwerte von $\boldsymbol{A}$ überwiegend *geclustert*, d. h. Eigenwerte häufen sich bei bestimmten Werten, so kann sich das Konvergenzverhalten des Verfahrens von dem in (2.3) beschriebenen deutlich unterscheiden. Wenn z. B. $\boldsymbol{A}$ lediglich k mit $k \ll n$ verschiedene Eigenwerte besitzt, so führt das Verfahren in exakter Arithmetik nach höchstens k Schritten zur exakten Lösung. Bei geclusterten Eigenwerten wird meist schnelle Konvergenz beobachtet [56]. Entscheidend für das Konvergenzverhalten der Methode der konjugierten Gradienten ist also die Eigenwertverteilung.

Für den Fehler der Iterierten gilt zusätzlich die strenge Monotonie [104]

$$\|x^{(j)} - x^*\|_2 < \|x^{(i)} - x^*\|_2 \quad \forall\, j > i,\ x^{(j)} \neq x^*.$$

Weitere Ergebnisse bezüglich des Konvergenzverhaltens der CG-Iteration sind in [133] zu finden.

2.2.3 Vorkonditionierung

In 2.2.2 wurde gezeigt, daß die Konvergenzrate des CG-Verfahrens von den Eigenschaften des Spektrums der Koeffizientenmatrix abhängt. Durch eine geeignete Vorkonditionierung kann das ursprüngliche Gleichungssystem in ein äquivalentes System mit gleicher Lösung, aber mit verbesserten spektralen Eigenschaften überführt werden. Als *Vorkonditionierer* wird eine Matrix bezeichnet, die diese Transformation leistet.

Das Prinzip der Vorkonditionierung besteht darin, eine Matrix $\boldsymbol{M}$ zu finden, die die Koeffizientenmatrix $\boldsymbol{A}$ geeignet approximiert. Das CG-Verfahren wird dann auf das transformierte System

$$\boldsymbol{M}^{-1}\boldsymbol{A}x = \boldsymbol{M}^{-1}b \tag{2.4}$$

angewendet. Die transformierte Koeffizientenmatrix $\boldsymbol{M}^{-1}\boldsymbol{A}$ muß dabei symmetrisch positiv definit sein und sollte günstigere spektrale Eigenschaften als $\boldsymbol{A}$ besitzen. Ferner ist $\boldsymbol{M}$ so zu wählen, daß ein Gleichungssystem mit der Koeffizientenmatrix $\boldsymbol{M}$ leichter als ein Gleichungssystem mit der Koeffizientenmatrix $\boldsymbol{A}$ gelöst werden kann. Eine andere Möglichkeit besteht darin, eine Matrix $\boldsymbol{M}^{-1}$ zu finden, die $\boldsymbol{A}^{-1}$ approximiert. In diesem Fall wird gewöhnlich ein zusätzliches Matrix-Vektor-Produkt mit $\boldsymbol{M}^{-1}$ pro Iterationsschritt durchgeführt. Die unten beschriebene polynomiale Vorkonditionierung gehört dieser Klasse an.

In der Praxis wird der Vorkonditionierer $\boldsymbol{M}$ meist in $\boldsymbol{M} = \boldsymbol{M}_1\boldsymbol{M}_2$ zerlegt und das Gleichungssystem in

$$\underbrace{\boldsymbol{M}_1^{-1}\boldsymbol{A}\boldsymbol{M}_2^{-1}}_{\tilde{\boldsymbol{A}}}\underbrace{(\boldsymbol{M}_2 x)}_{\tilde{x}} = \underbrace{\boldsymbol{M}_1^{-1}b}_{\tilde{b}} \tag{2.5}$$

überführt. Der entscheidende Vorteil dieser Transformation besteht darin, daß die Matrix $\boldsymbol{M}_1^{-1}\boldsymbol{A}\boldsymbol{M}_2^{-1}$ symmetrisch positiv definit ist, wenn $\boldsymbol{A}$ symmetrisch positiv definit ist und $\boldsymbol{M}_1 = \boldsymbol{M}_2^T$ gewählt wird. In diesem Fall liegt eine Kongruenztransformation vor. Die Zerlegung $\boldsymbol{M} = \boldsymbol{M}_1\boldsymbol{M}_1^T$ ist genau dann möglich, wenn $\boldsymbol{M}$ symmetrisch positiv definit ist. Das vorkonditionierte System $\tilde{\boldsymbol{A}}\tilde{x} = \tilde{b}$ wird mit dem CG-Verfahren gelöst. Der Lösungsvektor des ursprünglichen Systems ist durch $x = \boldsymbol{M}_2^{-1}\tilde{x}$ gegeben. Bei der unten beschriebenen Skalierung mit der Diagonalen wird auf diese Weise vorgegangen.

Der Beschleunigung der Konvergenz des CG-Verfahrens steht der Aufwand zur Konstruktion und Anwendung des Vorkonditionierers gegenüber. Ziel der Vorkonditionierung bei einer realen Implementierung ist sowohl die Einsparung von Rechenzeit als auch die Erhöhung der Stabilität des Verfahrens. Ferner stellt sich die Frage nach der Eignung der Methode zur Vorkonditionierung auf Vektor- oder Parallelrechnern, da einige traditionelle Verfahren wie die unvollständige Zerlegung sequentiellen Charakter haben.

In dieser Arbeit werden die Skalierung mit der Diagonalen [104] und die adaptive polynomiale Vorkonditionierung mit Tschebyscheff-Polynomen [11] [12] [121] betrachtet. Beide Verfahren sind für Vektor- und Parallelrechner geeignet, da der zusätzliche Aufwand im wesentlichen in Matrix-Vektor-Operationen besteht.

Weitere übliche Verfahren zur Vorkonditionierung sind die Jacobi-Vorkonditionierung, die SSOR-Vorkonditionierung (*Symmetric Successive Over-Relaxation*) und die vielen Varianten der unvollständigen Faktorisierung [17] [70] [104] [138].

Skalierung mit der Diagonalen

Die Skalierung mit der Diagonalen ist ein einfaches Verfahren zur Vorkonditionierung, das auf der Transformation aus (2.5) beruht. Hier wird

$$\boldsymbol{M}_1 = \boldsymbol{M}_2 = \boldsymbol{D}$$

gewählt. Die Elemente der Diagonalmatrix $\boldsymbol{D}$ werden aus den Diagonalelementen der Matrix $\boldsymbol{A}$ berechnet:

$$d_{jj} = \sqrt{a_{jj}}, \quad j = 1, \ldots, n.$$

Das transformierte Gleichungssystem

$$\underbrace{\boldsymbol{D}^{-1}\boldsymbol{A}\boldsymbol{D}^{-1}}_{\tilde{\boldsymbol{A}}}\underbrace{(\boldsymbol{D}x)}_{\tilde{x}} = \underbrace{\boldsymbol{D}^{-1}b}_{\tilde{b}}$$

wird mit dem CG-Verfahren gelöst; die Lösung des ursprünglichen Systems wird durch

$$x = \boldsymbol{D}^{-1}\tilde{x}$$

ermittelt. Der Aufwand dieser Methode zur Vorkonditionierung ist gering, da lediglich Multiplikationen mit einer Diagonalmatrix zur Transformation des Gleichungssystems erforderlich sind. In vielen Anwendungen, insbesondere in den im einführenden Kapitel erwähnten FE-Modellen aus der Umwelttechnik und der Strukturmechanik, führt die Skalierung mit der Diagonalen zu einer deutlichen Konvergenzbeschleunigung und wegen des geringen Aufwands auf einem Rechnersystem zu einer beträchtlichen Reduktion der Rechenzeit (s. auch [107]).

Polynomiale Vorkonditionierung

Die polynomiale Vorkonditionierung basiert auf der Transformation aus (2.4). Die Matrix $\boldsymbol{M}^{-1}$ ist bei dieser Methode als Matrixpolynom vom Grad m in $\boldsymbol{A}$ darstellbar:

$$\boldsymbol{M}^{-1} = \mathcal{C}(\boldsymbol{A}) = \sum_{j=0}^{m} \nu_j \boldsymbol{A}^j. \tag{2.6}$$

Das Matrixpolynom $\mathcal{C}(\boldsymbol{A})$ muß derart gewählt werden, daß die transformierte Koeffizientenmatrix $\mathcal{C}(\boldsymbol{A})\boldsymbol{A}$ symmetrisch positiv definit ist. Ziel der polynomialen Vorkonditionierung ist, daß

$$\kappa_2(\mathcal{C}(\boldsymbol{A})\boldsymbol{A}) \ll \kappa_2(\boldsymbol{A})$$

gilt bzw. daß die spektralen Eigenschaften von $\mathcal{C}(\boldsymbol{A})\boldsymbol{A}$ deutlich günstiger als die von $\boldsymbol{A}$ sind.

Die Eigenwertverteilung der transformierten Koeffizientenmatrix ist optimal, wenn $\mathcal{C}(\boldsymbol{A})\boldsymbol{A} = \boldsymbol{I}$ gilt. $\boldsymbol{I}$ bezeichnet die Einheitsmatrix. Dies führt zu dem Minimierungsproblem bezüglich der Spektralnorm

$$\min_{\mathcal{C} \in P_m} \|\boldsymbol{I} - \underbrace{\mathcal{C}(\boldsymbol{A})\boldsymbol{A}}_{\mathcal{P}(\boldsymbol{A})}\|_2 \tag{2.7}$$

mit P_m als der Menge aller Polynome vom Grad m. Bezogen auf die Eigenwerte λ_l, $l = 1, \ldots, n$, von $\boldsymbol{A}$ läßt sich (2.7) als Minimax-Problem formulieren:

$$\min_{\mathcal{C} \in P_m} \left\{ \max_{\lambda_l} |1 - \underbrace{\mathcal{C}(\lambda_l) \cdot \lambda_l}_{\mathcal{P}(\lambda_l)}| \right\}.$$

$\mathcal{P}$ ist ein Polynom vom Grad $k = m + 1$ und erfüllt $\mathcal{P}(0) = 0$. Gesucht wird also ein Polynom $\mathcal{P}$, das alle Eigenwerte von $\boldsymbol{A}$ möglichst nahe zu 1 abbildet.

Da die Eigenwerte von $\boldsymbol{A}$ gewöhnlich nicht bekannt sind, wird folgendes Ersatzproblem mit $0 < \alpha \leq \lambda_{\min}(\boldsymbol{A}) \leq \lambda_{\max}(\boldsymbol{A}) \leq \beta$, $\alpha \neq \beta$ und $\mathcal{P}(0) = 0$ betrachtet:

$$\min_{\mathcal{P} \in P_k} \left\{ \max_{\lambda \in [\alpha, \beta]} |1 - \mathcal{P}(\lambda)| \right\}. \tag{2.8}$$

Algorithmus 2.3. Die Tschebyscheff-Iteration

$$\boldsymbol{y} = \mathbf{CHEBY}(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{z}, \boldsymbol{A}, \boldsymbol{k})$$

$$\begin{aligned}
\theta &= \frac{\alpha+\beta}{2} \\
\sigma &= \frac{\alpha+\beta}{\beta-\alpha} \\
u^{(0)} &= 0 \\
u^{(1)} &= \frac{z}{\theta} \\
\rho_1 &= 2 \\
j &= 2, \ldots, k \\
&\qquad \rho_j = \frac{4}{4-\sigma^2\rho_{j-1}} \\
&\qquad u^{(j)} = \rho_j \left(u^{(j-1)} - u^{(j-2)} + \frac{z - \boldsymbol{A}u^{(j-1)}}{\theta} \right) + u^{(j-2)} \\
y &= u^{(k)}
\end{aligned}$$

Das Minimax-Problem (2.8) zu einem vorgegebenen Polynomgrad k kann mit Hilfe der Tschebyscheff-Polynome gelöst werden [11] [121]. Die Eigenschaften der Tschebyscheff-Polynome sind in [50] und [110] ausführlich beschrieben.

Bereits (2.6) ist zu entnehmen, daß anstelle der Operation $\boldsymbol{A}d^{(i)}$ im CG-Verfahren ohne Vorkonditionierung die Operation $\mathcal{C}(\boldsymbol{A})\boldsymbol{A}d^{(i)} = \sum_{j=0}^{m} \nu_j \boldsymbol{A}^{j+1} d^{(i)}$ im CG-Verfahren mit polynomialer Vorkonditionierung durchzuführen ist. Dies erfordert die Berechnung von $m+1$ Matrix-Vektor-Produkten pro Iterationsschritt. Bei Verwendung der Tschebyscheff-Polynome werden die Koeffizienten ν_j des Matrixpolynoms nicht explizit benötigt; sie werden implizit in der Tschebyscheff-Iteration ermittelt.

Der Algorithmus der Tschebyscheff-Iteration, der im CG-Verfahren mit polynomialer Tschebyscheff-Vorkonditionierung Verwendung findet, ist in Algorithmus 2.3 dargestellt [50] [121]. Die Tschebyscheff-Iteration berechnet die Operation $y = \mathcal{C}(\boldsymbol{A})z$, die in der CG-Iteration mit polynomialer Tschebyscheff-Vorkonditionierung auftritt. In der Tschebyscheff-Iteration werden im wesentlichen m Matrix-Vektor-Produkte und $3m$ Vektoradditionen durchgeführt.

Algorithmus 2.4 zeigt den Algorithmus des modifizierten CG-Verfahrens mit polynomialer Tschebyscheff-Vorkonditionierung. In Algorithmus 2.4 werden pro

Algorithmus 2.4. Das modifizierte CG-Verfahren mit polynomialer Tschebyscheff-Vorkonditionierung

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

$$\begin{aligned}\hat{g}^{(0)} &= \boldsymbol{A}x^{(0)} - b\\ g^{(0)} &= \mathrm{CHEBY}(\alpha, \beta, \hat{g}^{(0)}, \boldsymbol{A}, m+1)\\ d^{(0)} &= -g^{(0)}\end{aligned}$$

$i = 0, 1, \ldots$

$$\begin{aligned}w^{(i)} &= \mathrm{CHEBY}(\alpha, \beta, d^{(i)}, \boldsymbol{A}, m+1)\\ \gamma_i &= \frac{{g^{(i)}}^T g^{(i)}}{{d^{(i)}}^T \boldsymbol{A}w^{(i)}}\\ \delta_i &= \frac{\gamma_i(\boldsymbol{A}w^{(i)})^T \boldsymbol{A}w^{(i)}}{{d^{(i)}}^T \boldsymbol{A}w^{(i)}} - 1\\ {g^{(i+1)}}^T g^{(i+1)} &= \delta_i\, {g^{(i)}}^T g^{(i)}\\ x^{(i+1)} &= x^{(i)} + \gamma_i d^{(i)}\\ g^{(i+1)} &= g^{(i)} + \gamma_i \boldsymbol{A}w^{(i)}\\ d^{(i+1)} &= -g^{(i+1)} + \delta_i d^{(i)}\end{aligned}$$

bis $\|g^{(i+1)}\|_2 \leq \epsilon_{\mathrm{r}}$

Iterationsschritt insgesamt $k = m + 1$ Matrix-Vektor-Produkte, drei Skalarprodukte und $3 + 3m$ Vektoradditionen berechnet. Neben dem Grad m des Matrixpolynoms $\mathcal{C}(\boldsymbol{A})$ müssen Schätzwerte für die extremen Eigenwerte von $\boldsymbol{A}$ vorgegeben werden; dies ist bereits der Formulierung des Minimax-Problems (2.8) zu entnehmen. Bei ungeradem Polynomgrad m ist bei positiven Schätzwerten für die extremen Eigenwerte, die nur einen Teil des gesamten Spektrums einschließen, nicht gewährleistet, daß die transformierte Koeffizientenmatrix symmetrisch positiv definit ist. Daher wird der Polynomgrad auf gerade m eingeschränkt [11] [121].

Bei der adaptiven polynomialen Vorkonditionierung mit Tschebyscheff-Polynomen werden die Schätzwerte für die extremen Eigenwerte von $\boldsymbol{A}$ adaptiv ermittelt, d. h. die Eigenwertschätzungen werden im Verlauf der Iteration verbessert [11] [12] [121]. Grundlage des adaptiven Verfahrens ist die theoretische Verknüpfung des CG-Verfahrens und der Lanczos-Methode zur Lösung des reellen symmetrischen Eigenwertproblems. Durch diesen Zusammenhang ist es möglich, aus den Iterationsparametern γ_i und δ_i Approximationen für die Eigenwerte der

aktuellen Matrix $\mathcal{P}(\boldsymbol{A})$ zu ermitteln (s. auch 2.4.1). Aus diesen Schätzwerten lassen sich Näherungen für die Eigenwerte von $\boldsymbol{A}$ bestimmen, die zur Aktualisierung des Intervalls $[\alpha, \beta]$ benötigt werden. Nach einer vorgegebenen Anzahl von Iterationsschritten werden die Eigenwertschätzungen zyklisch berechnet; anschließend wird überprüft, ob diese Werte ein Intervall liefern, das mehr Eigenwerte von $\boldsymbol{A}$ als das bisherige enthält. Die initialen Werte für α und β werden ermittelt, indem einige Schritte des CG-Verfahrens ohne Vorkonditionierung durchgeführt werden. Aus den Iterationsparametern werden dann erste Approximationen für die extremen Eigenwerte von $\boldsymbol{A}$ gewonnen [121].

Bei polynomialer Vorkonditionierung mit Tschebyscheff-Polynomen kann die Anzahl der Iterationsschritte des CG-Verfahrens bei dem Polynomgrad m höchstens um den Faktor $\frac{1}{m+1}$ reduziert werden [17]. Dies bedeutet, daß zwar keine Matrix-Vektor-Produkte eingespart werden, jedoch die Anzahl der Skalarprodukte sinkt. Insbesondere auf Parallelrechnern mit verteiltem Speicher kann die Einsparung von Skalarprodukten zu einem Effizienzgewinn führen.

Untersuchungen zu CG-Verfahren mit polynomialer Vorkonditionierung mit Least-Squares Polynomen sind in [84] und [114] zu finden. Least-Squares Polynome erfordern lediglich die Bestimmung einer oberen Schranke für das Spektrum der Koeffizientenmatrix, die z. B. in einfacher Weise durch das Gershgorin-Theorem ermittelt werden kann. Der Vergleich von Tschebyscheff- und Least-Squares Polynomen zur polynomialen Vorkonditionierung in [12] zeigt jedoch keinen entscheidenden Vorteil für eine der beiden Methoden.

2.3 Verfahren für Gleichungssysteme mit unsymmetrischer Matrix

In diesem Abschnitt werden zwei Verfahren beschrieben, die zur Lösung von Gleichungssystemen mit regulärer unsymmetrischer Koeffizientenmatrix geeignet sind. Die Verfahren QMR und TFQMR, die hier betrachtet werden, wurden um 1990 von Freund und Nachtigal entwickelt [66] [68] und sind derzeit ein aktuelles Forschungsthema [37] [67] [103]. Insbesondere interessiert der Einsatz dieser Methoden in technischen Anwendungen. In dem im einführenden Kapitel erwähnten FE-Modell aus der Umwelttechnik führt die Lösung der Transportgleichung zu einem Gleichungssystem mit regulärer unsymmetrischer Koeffizientenmatrix [140] [145]. Die Integration eines der beiden Verfahren QMR oder TFQMR in dieses Transportmodell ist vorgesehen.

Sowohl die QMR- als auch die TFQMR-Methode beruhen auf dem Ansatz der *quasi-minimalen Residuen.* Statt der Norm des Residuums wird in jedem Iterationsschritt lediglich ein Faktor des Residuums minimiert. Der Aufwand des zu lösenden Minimierungsproblems wird dadurch deutlich geringer.

Gegenüber anderen üblichen iterativen Lösungsverfahren für unsymmetrische

Algorithmus 2.5. Die QMR-Initialisierung

Initialisiere QMR

$$\begin{aligned}
&\textit{Wähle } x^{(0)} \in \mathbb{R}^n \textit{ beliebig}\\
g^{(0)} &= \boldsymbol{A}x^{(0)} - b\\
\tilde{q}^{(1)} &= -g^{(0)}\\
&\textit{Löse } \boldsymbol{M}_1 y = \tilde{q}^{(1)}\\
\rho_1 &= \|y\|_2\\
&\textit{Wähle } \tilde{w}^{(1)} \textit{ beliebig, z. B. } \tilde{w}^{(1)} = -g^{(0)}\\
&\textit{Löse } \boldsymbol{M}_2^T t = \tilde{w}^{(1)}\\
\xi_1 &= \|t\|_2\\
\gamma_0 &= 1\\
\eta_0 &= -1
\end{aligned}$$

Systeme wie BiCG (*Biconjugate Gradient*) [92], BiCGSTAB (*Biconjugate Gradient Stabilized*) [139] — einer verbesserten Variante von BiCG — und CGS (*Conjugate Gradient Squared*) [132] zeichnet die Methoden QMR und TFQMR ein deutlich günstigeres Konvergenzverhalten aus [37] [66] [68]. Freund und Nachtigal stellen in [68] Fehlerschranken für das QMR-Verfahren vor und weisen nach, daß die Konvergenzgeschwindigkeit von QMR annähernd der von GMRES (*Generalized Minimal Residual*) [120] entspricht. Erste Ansätze zur Abschätzung des Fehlerverhaltens von QMR und TFQMR sind in [65] zu finden.

In der Praxis werden iterative Lösungsmethoden für unsymmetrische Systeme meist in Verbindung mit effizienten Verfahren zur Vorkonditionierung verwendet [115]. Einige Methoden zur Vorkonditionierung, die sich für die Verfahren QMR und TFQMR eignen, sind in [17], [68] und [129] beschrieben.

2.3.1 Das QMR-Verfahren

Das QMR-Verfahren ist eine iterative Methode auf Polynombasis [68]. Es basiert auf dem klassischen unsymmetrischen Lanczos-Algorithmus [70], der zur Erzeugung der Basisvektoren der Krylov-Teilräume verwendet wird. Die Lanczos-Methode wird mit dem Ansatz der quasi-minimalen Residuen kombiniert.

Der Lanczos-Algorithmus erzeugt die Matrix $\boldsymbol{Q}^{(i)} \in \mathbb{R}^{n\times i}$ mit

$$\boldsymbol{Q}^{(i)} = \left[q^{(1)}q^{(2)}\cdots q^{(i)}\right],$$

Algorithmus 2.6. Die Matrix-Vektor-Produkte von QMR

Berechne die Matrix-Vektor-Produkte von QMR

$$\begin{aligned}
\varepsilon_i &= v^{(i)^T} \boldsymbol{A} p^{(i)} \\
\beta_i &= \frac{\varepsilon_i}{\delta_i} \\
\tilde{q}^{(i+1)} &= \boldsymbol{A} p^{(i)} - \beta_i q^{(i)} \\
&\textit{Löse } \boldsymbol{M}_1 y = \tilde{q}^{(i+1)} \\
\rho_{i+1} &= \|y\|_2 \\
\tilde{w}^{(i+1)} &= \boldsymbol{A}^T v^{(i)} - \beta_i w^{(i)} \\
&\textit{Löse } \boldsymbol{M}_2^T t = \tilde{w}^{(i+1)} \\
\xi_{i+1} &= \|t\|_2 \\
\vartheta_i &= \frac{\rho_{i+1}}{\gamma_{i-1}|\beta_i|} \\
\gamma_i &= \frac{1}{\sqrt{1+\vartheta_i^2}} \\
\eta_i &= \frac{-\eta_{i-1}\rho_i\gamma_i^2}{\beta_i\gamma_{i-1}^2}
\end{aligned}$$

deren Spalten die Lanczos-Vektoren $q^{(j)}$, $j = 1, \ldots, i$, bilden und die obere Hessenberg-Matrix $\boldsymbol{H}^{(i)} \in \mathbb{R}^{(i+1)\times i}$ [68]. Mit Hilfe dieser Matrizen und dem ersten Einheitsvektor $e^{(i+1)} \in \mathbb{R}^{i+1}$ ist das Residuum im i-ten Iterationsschritt von QMR in der Form

$$g^{(i)} = \boldsymbol{Q}^{(i+1)}(\boldsymbol{H}^{(i)} z - e^{(i+1)}) \quad \text{für ein} \quad z \in \mathbb{R}^i \tag{2.9}$$

darstellbar. In jedem Iterationsschritt wird das Minimierungsproblem

$$\|\boldsymbol{H}^{(i)} z^{(i)} - e^{(i+1)}\|_2 = \min_{z\in\mathbb{R}^i} \|\boldsymbol{H}^{(i)} z - e^{(i+1)}\|_2$$

gelöst und mit der Lösung $z^{(i)}$ die QMR-Iterierte

$$x^{(i)} = x^{(0)} + \boldsymbol{Q}^{(i)} z^{(i)}$$

bestimmt. Das QMR-Verfahren minimiert also lediglich den rechten Faktor des Residuums aus (2.9).

Der Algorithmus der QMR-Methode mit Vorkonditionierung ist in den Algorithmen 2.5–2.7 dargestellt [17] [37]. Der oben zur Erläuterung des Minimierungsproblems eingeführte Vektor $z^{(i)}$ kommt in der Iteration nicht explizit vor; er wird implizit zur Berechnung von $d^{(i)}$ (s. Algorithmus 2.7) verwendet. Als Vorkonditionierer wird $\boldsymbol{M} = \boldsymbol{M}_1\boldsymbol{M}_2$ gewählt, so daß das urspüngliche Gleichungssystem wie in (2.5) transformiert wird. Im QMR-Algorithmus tritt der Vorkonditionierer ausschließlich in der folgenden Form auf:

$$\begin{aligned} &\textit{Löse}\ \ \boldsymbol{M}_1^T\tilde{t} = t \quad \text{und} \quad \boldsymbol{M}_1 y = \tilde{q}^{(i+1)} \quad \text{sowie} \\ &\textit{Löse}\ \ \boldsymbol{M}_2\tilde{y} = y \quad \text{und} \quad \boldsymbol{M}_2^T t = \tilde{w}^{(i+1)}. \end{aligned}$$

Die Initialisierungen von QMR sind in Algorithmus 2.5 dargestellt. In jedem Iterationsschritt des QMR-Verfahrens werden zwei Matrix-Vektor-Produkte berechnet. Der Teil der Methode, in dem diese beiden Operationen durchgeführt werden, ist in Algorithmus 2.6 enthalten. Die Matrix-Vektor-Produkte $\boldsymbol{A}p^{(i)}$ und $\boldsymbol{A}^T v^{(i)}$ können unabhängig voneinander bestimmt werden, da das Ergebnis von $\boldsymbol{A}p^{(i)}$ nicht in die Berechnung des Vektors $v^{(i)}$ eingeht. Dies kann insbesondere bei der Implementierung des Verfahrens auf einem Parallelrechner mit verteiltem Speicher genutzt werden. Auf diese Eigenschaft wird in einem späteren Kapitel eingegangen. Algorithmus 2.7 zeigt den übrigen Teil der QMR-Iteration. Insgesamt werden ein Matrix-Vektor-Produkt mit der Koeffizientenmatrix $\boldsymbol{A}$, ein Matrix-Vektor-Produkt mit der Transponierten $\boldsymbol{A}^T$, fünf Skalarprodukte bzw. Reduktionen und 12 Vektoradditionen bzw. -skalierungen pro Iterationsschritt durchgeführt. Die Operationen für eine eventuelle Vorkonditionierung sind hier nicht berücksichtigt; die Aussagen bezüglich der Anzahl der Operationen pro Iterationsschritt gelten für den trivialen Vorkonditionierer $\boldsymbol{M} = \boldsymbol{I}$.

2.3.2 Das TFQMR-Verfahren

Wie QMR ist das TFQMR-Verfahren eine iterative Methode auf Polynombasis, die von Freund entwickelt wurde [66]. Im Gegensatz zu QMR ist die TFQMR-Iteration *transpositionsfrei*, d. h. Matrix-Vektor-Produkte mit der Transponierten der Koeffizientenmatrix treten nicht auf. Der Herleitung von TFQMR liegt der CGS-Algorithmus [132] zugrunde. Im CGS-Verfahren wird die Iterierte in der Form

$$x^{(i)} = x^{(i-1)} + \delta_{i-1}(t^{(i-1)} + h^{(i)})$$

mit den beiden Suchrichtungen $t^{(i-1)}$ und $h^{(i)}$ gebildet. Während CGS diese Suchrichtungen lediglich als Linearkombination zur Berechnung der aktuellen Iterierten verwendet, ermittelt TFQMR zu jeder Suchrichtung eine eigene Iterierte [66].

Die Residuen beider Iterierten $x^{(j)}$, $j = 2i-1, 2i$, können in der Form

$$g^{(j)} = \boldsymbol{W}^{(j+1)}(\boldsymbol{H}^{(j)}z - f^{(j+1)}) \quad \text{für ein} \quad z \in \mathbb{R}^j \tag{2.10}$$

Algorithmus 2.7. Das QMR-Verfahren

Initialisiere QMR

$i = 1, 2, \ldots$

$$\begin{aligned} q^{(i)} &= \frac{\tilde{q}^{(i)}}{\rho_i} \\ y &= \frac{y}{\rho_i} \\ w^{(i)} &= \frac{\tilde{w}^{(i)}}{\xi_i} \\ t &= \frac{t}{\xi_i} \\ \delta_i &= t^T y \end{aligned}$$

$$\begin{aligned} \textit{Löse } \boldsymbol{M}_2 \tilde{y} &= y \\ \textit{Löse } \boldsymbol{M}_1^T \tilde{t} &= t \end{aligned}$$

$$\textit{Für } i = 1 : \begin{cases} p^{(1)} = \tilde{y} \\ v^{(1)} = \tilde{t} \end{cases}$$

$$\textit{Für } i > 1 : \begin{cases} p^{(i)} = \tilde{y} - (\xi_i \delta_i / \varepsilon_{i-1}) p^{(i-1)} \\ v^{(i)} = \tilde{t} - (\rho_i \delta_i / \varepsilon_{i-1}) v^{(i-1)} \end{cases}$$

Berechne die Matrix-Vektor-Produkte von QMR

$$\textit{Für } i = 1 : \begin{cases} d^{(1)} = \eta_1 p^{(1)} \\ s^{(1)} = \eta_1 \boldsymbol{A} p^{(1)} \end{cases}$$

$$\textit{Für } i > 1 : \begin{cases} d^{(i)} = \eta_i p^{(i)} + (\vartheta_{i-1} \gamma_i)^2 d^{(i-1)} \\ s^{(i)} = \eta_i \boldsymbol{A} p^{(i)} + (\vartheta_{i-1} \gamma_i)^2 s^{(i-1)} \end{cases}$$

$$\begin{aligned} x^{(i)} &= x^{(i-1)} + d^{(i)} \\ g^{(i)} &= g^{(i-1)} + s^{(i)} \end{aligned}$$

bis $\|g^{(i)}\|_2 \leq \epsilon_{\mathrm{r}}$

Algorithmus 2.8. Die TFQMR-Initialisierung

Initialisiere TFQMR

Wähle $x^{(0)} \in I\!R^n$ *beliebig*

$$\begin{aligned}
g^{(0)} &= \boldsymbol{A}x^{(0)} - b \\
y^{(1)} &= -g^{(0)} \\
w^{(1)} &= -g^{(0)} \\
v^{(0)} &= \boldsymbol{A}y^{(1)} \\
d^{(0)} &= 0 \\
\tau_0 &= \|g^{(0)}\|_2 \\
\vartheta_0 &= 0 \\
\eta_0 &= 0
\end{aligned}$$

Wähle $\tilde{g}^{(0)}$ *so, daß* $\rho_0 = -\tilde{g}^{(0)^T} g^{(0)} \neq 0$

dargestellt werden. Die Matrix $\boldsymbol{W}^{(j+1)} \in \mathbb{R}^{n\times(j+1)}$, die Matrix $\boldsymbol{H}^{(j)} \in \mathbb{R}^{(j+1)\times j}$, die den Rang j besitzt und der Vektor $f^{(j+1)} \in \mathbb{R}^{j+1}$ sind durch die Herleitung aus dem CGS-Algorithmus gegeben [37] [66]. Wie beim QMR-Verfahren wird in jedem Iterationsschritt der TFQMR-Methode lediglich der rechte Faktor der Residuen aus (2.10) minimiert:

$$\|\boldsymbol{H}^{(j)} z^{(j)} - f^{(j+1)}\|_2 = \min_{z\in\mathbb{R}^j} \|\boldsymbol{H}^{(j)} z - f^{(j+1)}\|_2.$$

Die Algorithmen 2.8 und 2.9 zeigen die TFQMR-Iteration [37]. Während in Algorithmus 2.8 die Initialisierungen aufgeführt sind, ist in Algorithmus 2.9 die Iteration des TFQMR-Verfahrens dargestellt. Der oben eingeführte Vektor $z^{(j)}$ wird implizit zur Berechnung der Iterierten $x^{(j)}$ (s. Algorithmus 2.9) verwendet.

In jedem Iterationsschritt i von Algorithmus 2.9 werden die zwei Iterierten $x^{(2i-1)}$ und $x^{(2i)}$ ermittelt. Die Iteration wird beendet, wenn ein vorgegebenes Abbruchkriterium erfüllt ist. Das Residuum wird im TFQMR-Verfahren nicht explizit berechnet. Wird der Wert des Residuums für die Überprüfung des Abbruchkriteriums benötigt, können zusätzlich die Operationen $\boldsymbol{A}x^{(2i-1)} - b$ und $\boldsymbol{A}x^{(2i)} - b$ durchgeführt werden. Dies kann z. B. zyklisch nach einer vorgegebenen Anzahl von Iterationsschritten geschehen, um den zusätzlichen Aufwand zu reduzieren. Abgesehen von eventuellen Operationen zur Bestimmung des Residuums werden in jedem Iterationsschritt i des TFQMR-Verfahrens die zwei Matrix-

Algorithmus 2.9. Das TFQMR-Verfahren

Initialisiere TFQMR

$i = 1, 2, \ldots$

$$\begin{aligned}
\sigma_{i-1} &= \tilde{g}^{(0)^T} v^{(i-1)} \\
\alpha_{i-1} &= \frac{\rho_{i-1}}{\sigma_{i-1}} \\
y^{(2i)} &= y^{(2i-1)} - \alpha_{i-1} v^{(i-1)}
\end{aligned}$$

$$\textit{Für } j = 2i-1, 2i : \left\{ \begin{aligned}
w^{(j+1)} &= w^{(j)} - \alpha_{i-1} \boldsymbol{A} y^{(j)} \\
\vartheta_j &= \frac{\|w^{(j+1)}\|_2}{\tau_{j-1}} \\
\gamma_j &= \frac{1}{\sqrt{1+\vartheta_j^2}} \\
\tau_j &= \tau_{j-1} \vartheta_j \gamma_j \\
\eta_j &= \gamma_j^2 \alpha_{i-1} \\
d^{(j)} &= y^{(j)} + \frac{\vartheta_{j-1}^2 \eta_{j-1}}{\alpha_{i-1}} d^{(j-1)} \\
x^{(j)} &= x^{(j-1)} + \eta_j d^{(j)}
\end{aligned} \right.$$

$$\begin{aligned}
\rho_i &= \tilde{g}^{(0)^T} w^{(2i+1)} \\
\beta_i &= \frac{\rho_i}{\rho_{i-1}} \\
y^{(2i+1)} &= w^{(2i+1)} + \beta_i y^{(2i)} \\
v^{(i)} &= \boldsymbol{A} y^{(2i+1)} + \beta_i \left(\boldsymbol{A} y^{(2i)} + \beta_i v^{(i-1)} \right)
\end{aligned}$$

bis $x^{(2i-1)}$ *oder* $x^{(2i)}$ *konvergieren*

Vektor-Produkte $\boldsymbol{A}y^{(2i)}$ und $\boldsymbol{A}y^{(2i+1)}$, vier Skalarprodukte bzw. Reduktionen und zehn Vektoradditionen berechnet.

2.4 Ein Lanczos-Verfahren für das symmetrische Eigenwertproblem

In vielen technischen Aufgabenstellungen wie Lärmschutzberechnungen oder mechanischen Schwingungsproblemen [42] [81] [118] treten reelle symmetrische Eigenwertprobleme

$$\boldsymbol{A}z = \lambda z \tag{2.11}$$

auf. Die Lösungen des Problems (2.11) mit der reellen symmetrischen Matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ sind die Paare $(\lambda_k, z^{\{k\}})$, $k = 1, 2, \ldots, n$, mit den sogenannten Eigenwerten $\lambda_k \in \mathbb{R}$ und Eigenvektoren $z^{\{k\}} \in \mathbb{R}^n$ der Matrix.

Die Lanczos-Methode ist ein Standard-Verfahren zur Bestimmung einiger extremer Eigenwerte und Eigenvektoren von großen dünnbesetzten Matrizen [44] [45] [70] [105] [142] [143]; gerade die extremen Eigenwerte und Eigenvektoren sind in vielen technischen und physikalischen Anwendungen zu ermitteln. Das Lanczos-Verfahren erzeugt iterativ eine Folge von Tridiagonalmatrizen. Die Eigenwerte dieser Tridiagonalmatrizen approximieren die Eigenwerte der ursprünglichen Matrix.

2.4.1 Tridiagonalisierung

In Algorithmus 2.10 ist eine Variante der Lanczos-Tridiagonalisierung dargestellt, die in [15] verwendet wurde.

Algorithmus 2.10. Die Lanczos-Tridiagonalisierung

Wähle $q^{(1)}$ *mit* $\|q^{(1)}\|_2 = 1$ *beliebig und setze* $q^{(0)} = 0$, $\beta_1 = 0$

$i = 1, 2, \ldots$

$$\begin{aligned}
\alpha_i &= q^{(i)T} \boldsymbol{A} q^{(i)} \\
r^{(i)} &= \boldsymbol{A} q^{(i)} - \beta_i q^{(i-1)} - \alpha_i q^{(i)} \\
\beta_{i+1} &= \|r^{(i)}\|_2 \\
q^{(i+1)} &= \frac{r^{(i)}}{\beta_{i+1}}
\end{aligned}$$

In der Lanczos-Iteration werden eine Vektor-Folge $\{q^{(i)}\}_{i=1,2,...}$ mit $q^{(i)} \in \mathbb{R}^n$ und eine Folge von symmetrischen $i \times i$ Tridiagonalmatrizen $\boldsymbol{T}_i$, $i = 1, 2, \ldots$, generiert. Die Vektoren $q^{(i)}$ sind orthonormal und werden Lanczos-Vektoren genannt; die symmetrischen Tridiagonalmatrizen $\boldsymbol{T}_i$ heißen Lanczos-Matrizen. Die Matrizen $\boldsymbol{T}_i$ haben die Gestalt

$$\boldsymbol{T}_i = \begin{pmatrix} \alpha_1 & \beta_2 & 0 & \cdots & 0 \\ \beta_2 & \alpha_2 & \beta_3 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \beta_{i-1} & \alpha_{i-1} & \beta_i \\ 0 & \cdots & 0 & \beta_i & \alpha_i \end{pmatrix}$$

mit den Elementen der Hauptdiagonalen $t_{jj} = \alpha_j$, $j = 1, \ldots, i$, und den Elementen der Nebendiagonalen $t_{j-1,j} = t_{j,j-1} = \beta_j$, $j = 2, \ldots, i$. Ferner bezeichnet $\|r^{(i)}\|_2 = \left(r^{(i)^T} r^{(i)}\right)^{1/2}$ in Algorithmus 2.10 die euklidische Norm des Residuumsvektors $r^{(i)} \in \mathbb{R}^n$. Bei den meisten dünnbesetzten Matrizen aus technischen Aufgabenstellungen liegt die Hauptarbeit jedes Iterationsschritts in der Berechnung des Matrix-Vektor-Produkts $\boldsymbol{A}q^{(i)}$. Neben dieser Operation sind pro Iterationsschritt ein Skalarprodukt, eine Reduktion und drei Vektoradditionen bzw. -skalierungen durchzuführen.

In endlicher Arithmetik verlieren die Lanczos-Vektoren $q^{(i)}$ aufgrund von Rundungsfehlern mit steigender Anzahl der Iterationsschritte ihre paarweise Orthogonalität. Um eine ausreichende numerische Genauigkeit zu wahren, kann die Lanczos-Methode um einen aufwendigen Reorthogonalisierungsschritt erweitert werden [44] [105]. Wird die Lanczos-Iteration jedoch ausreichend lange durchgeführt, so werden gewöhnlich gute Näherungen aller Eigenwerte der Matrix erzeugt [15] [44]. Diese Eigenschaft des Lanczos-Verfahrens führt dazu, daß Rundungsfehler die Bestimmung von Eigenwerten lediglich verzögern und nicht verhindern. Daher wird die Reorthogonalisierung der Lanczos-Vektoren im Rahmen dieser Arbeit nicht betrachtet.

Die Lanczos-Methode zur Lösung des symmetrischen Eigenwertproblems und das CG-Verfahren zur Lösung von symmetrischen positiv definiten linearen Gleichungssystemen können wechselseitig auseinander hergeleitet werden [17] [44] [105]. Insbesondere sind die Elemente der Tridiagonalmatrizen $\boldsymbol{T}_i$ der Lanczos-Tridiagonalisierung durch die Iterationsparameter γ_i und δ_i des CG-Verfahrens ohne Vorkonditionierung darstellbar:

$$\begin{aligned} \alpha_1 &= \frac{1}{\gamma_1}, \\ \alpha_j &= \frac{1}{\gamma_j} + \delta_{j-1}\frac{1}{\gamma_{j-1}}, \quad j = 2, \ldots, i, \\ \beta_j &= -\sqrt{\delta_{j-1}}\frac{1}{\gamma_{j-1}}, \quad j = 2, \ldots, i. \end{aligned}$$

Diese Eigenschaft wird in der in 2.2.3 beschriebenen adaptiven Vorkonditionierung mit Tschebyscheff-Polynomen des CG-Verfahrens genutzt, um Schätzwerte für die extremen Eigenwerte der Koeffizientenmatrix zu erhalten.

Algorithmus 2.11 zeigt den Verlauf einer modifizierten Variante der Lanczos-Tridiagonalisierung aus [87]. Gegenüber Algorithmus 2.10 besitzt dieses Verfahren günstigere Parallelisierungseigenschaften.

Algorithmus 2.11. Die modifizierte Lanczos-Tridiagonalisierung

Wähle $r^{(0)}$ *mit* $r^{(0)} \neq 0$ *beliebig und setze* $q^{(0)} = 0$

$i = 1, 2, \ldots$

$$\begin{aligned}
\beta_i &= \|r^{(i-1)}\|_2 \\
\alpha_i &= \frac{r^{(i-1)^T} \boldsymbol{A} r^{(i-1)}}{r^{(i-1)^T} r^{(i-1)}} \\
q^{(i)} &= \frac{r^{(i-1)}}{\beta_i} \\
r^{(i)} &= \frac{\boldsymbol{A} r^{(i-1)}}{\beta_i} - \beta_i q^{(i-1)} - \alpha_i q^{(i)}
\end{aligned}$$

Im Gegensatz zu Algorithmus 2.10 können in der modifizierten Lanczos-Tridiagonalisierung alle Skalarprodukte zu Beginn jedes Iterationsschritts berechnet werden. Für die Implementierung der Methode auf einem Parallelrechner mit verteiltem Speicher ergeben sich die gleichen Vorteile wie bei dem modifizierten CG-Verfahren aus Algorithmus 2.2. Hinsichtlich der Stabilität der Methode zeigen Algorithmus 2.10 und Algorithmus 2.11 bei den im Rahmen dieser Arbeit untersuchten Matrizen keine Unterschiede.

2.4.2 Lösung des tridiagonalen Eigenwertproblems

Nach einer vorgegebenen Anzahl von Iterationsschritten der Lanczos-Tridiagonalisierung werden die Eigenwerte der aktuellen Tridiagonalmatrix berechnet und überprüft, welche dieser Eigenwerte ausreichende Näherungen für Eigenwerte der ursprünglichen Matrix $\boldsymbol{A}$ sind. Dies wird wiederholt, bis eine vorbestimmte Anzahl von Eigenwerten der Matrix $\boldsymbol{A}$ in vorgegebener Genauigkeit ermittelt ist.

Die Bestimmung der Eigenwerte der Tridiagonalmatrizen geschieht durch ein Bisektionsverfahren, das auf dem im Vorfeld dieser Arbeit entwickelten parallelen Algorithmus ALLEV (*All Eigenvalues*) [18] [28] basiert. Das Verfahren verwendet die Sturmsche Kette. Der Algorithmus ALLEV ermittelt die Eigenwerte in zwei Phasen: Isolation und Extraktion. Zunächst werden durch Bisektion Intervalle

bestimmt, die genau einen Eigenwert enthalten. Anschließend werden die Eigenwerte in vorgegebener Genauigkeit extrahiert. Zu diesem Zweck wird ein Verfahren zur Nullstellensuche mit superlinearer Konvergenz, das Pegasus-Verfahren [52], verwendet. Die Parallelisierungsstrategie des Algorithmus ALLEV auf einem Parallelrechner mit verteiltem Speicher besteht darin, Intervalle, die Eigenwerte enthalten, auf die einzelnen Prozessoren des Rechnersystems zu verteilen und dort unabhängig voneinander auszuwerten. Das genaue Vorgehen ist in [28] beschrieben.

Ein Standard-Verfahren zur Lösung des tridiagonalen Eigenwertproblems ist die QR-Faktorisierung [70]. Gegenüber dieser Methode besitzt das Verfahren ALLEV deutlich günstigere Parallelisierungseigenschaften. Zusätzlich erwies sich der letztere Algorithmus auf einem Prozessor eines CRAY-Vektorrechners als ca. doppelt so schnell wie eine optimierte Bibliotheksversion des QR-Verfahrens bei sogar höherer Genauigkeit von ALLEV [28].

Für die Verwendung in der hier betrachteten Lanczos-Methode wurde das Verfahren dahingehend modifiziert, alle Eigenwerte in einem vorgegebenen Intervall zu finden. Darüber hinaus wurden Kriterien integriert, um zu entscheiden, welche Eigenwerte der aktuellen Tridiagonalmatrix ausreichende Näherungen von Eigenwerten der Matrix $\boldsymbol{A}$ sind.

Für den Eigenwert μ_j von $\boldsymbol{T}_i$ und den zugehörigen Eigenwert λ_k von $\boldsymbol{A}$ gilt die folgende Fehlerabschätzung [15] [105]:

$$|\mu_j - \lambda_k| \leq \beta_{i+1}|y_i^{\{j\}}|.$$

$y_i^{\{j\}}$ bezeichnet die i-te Komponente des Eigenvektors $y^{\{j\}}$ des Paars $(\mu_j, y^{\{j\}})$ von $\boldsymbol{T}_i$. In dem entwickelten parallelen Verfahren LANSP (*Lanczos Sparse*) wird ein Eigenwert μ_j von $\boldsymbol{T}_i$ als genügend genaue Approximation des Eigenwerts λ_k von $\boldsymbol{A}$ akzeptiert, wenn

$$\beta_{i+1}|y_i^{\{j\}}| \leq \epsilon_t|\mu_j| \tag{2.12}$$

für eine vorgegebene Toleranz ϵ_t gilt. Die Gültigkeit von (2.12) muß jedoch lediglich für eine Teilmenge der Eigenwerte von $\boldsymbol{T}_i$ überprüft werden [44]. Zur Bestimmung dieser Teilmenge werden die Eigenwerte von $\boldsymbol{T}_i$ klassifiziert; die einzelnen Klassen werden wie folgt durch einen entsprechenden *MP-Wert* gekennzeichnet [15] [44]:

1. Unechte (*spurious*) Eigenwerte ($MP = 0$). Sei $\boldsymbol{S}_{i-1}$ die Tridiagonalmatrix, die sich durch Streichen der ersten Zeile und Spalte von $\boldsymbol{T}_i$ ergibt. Ein Eigenwert von $\boldsymbol{T}_i$ wird als unecht definiert, wenn er ein numerisch einfacher Eigenwert von $\boldsymbol{T}_i$ und gleichzeitig ein Eigenwert von $\boldsymbol{S}_{i-1}$ ist. Unechte Eigenwerte sind häufig sehr schlechte Näherungen für Eigenwerte von $\boldsymbol{A}$.

2. Vielfache Eigenwerte ($MP > 1$). *Cluster von Eigenwerten*, d. h. Eigenwerte, die extrem nah beieinander liegen, werden a priori als ausreichende Näherungen für Eigenwerte von $\boldsymbol{A}$ angenommen.

3. Nicht-isolierte einfache Eigenwerte ($MP = -1$). Liegt irgendeiner der übrigen Eigenwerte dicht an einem unechten Eigenwert (*dicht* bezieht sich hier auf eine gröbere Toleranz als die oben verwendete), so wird dieser Eigenwert mit -1 gekennzeichnet und als genügend genau angenommen.

4. Einfache Eigenwerte ($MP = 1$). Lediglich für die übriggebliebenen Eigenwerte wird die Gültigkeit von Kriterium (2.12) überprüft. Zur Berechnung der zu diesen Eigenwerten korrespondierenden Eigenvektoren wird die LAPACK-Routine DSTEIN [10] verwendet.

Der parallele Algorithmus LANSP erlaubt die Lösung der folgenden Eigenwertprobleme mit dünnbesetzter Matrix:

1. Berechnung aller Eigenwerte,

2. Berechnung aller Eigenwerte in einem vorgegebenen Intervall,

3. Berechnung einer vorbestimmten Anzahl von Eigenwerten,

4. Berechnung einer vorbestimmten Anzahl von Eigenwerten in einem vorgegebenen Intervall.

In der aktuellen Version von LANSP werden die zu den Eigenwerten der Matrix $\boldsymbol{A}$ korrespondierenden Eigenvektoren nicht ermittelt. Dies ist jedoch durch eine geringfügige Modifikation des Algorithmus möglich, da die Bestimmung der zu den Eigenwerten der Tridiagonalmatrizen $\boldsymbol{T}_i$ korrespondierenden Eigenvektoren bereits integriert ist. Zu einem beliebigen Eigenvektor $y^{\{j\}}$ von $\boldsymbol{T}_i$ ist

$$u^{\{j\}} = \sum_{k=1}^{i} y_k^{\{j\}} q^{(k)}$$

eine Näherung eines Eigenvektors der Matrix $\boldsymbol{A}$, d. h. das Paar $(\mu_j, u^{\{j\}})$ approximiert das Paar $(\lambda_k, z^{\{k\}})$ von $\boldsymbol{A}$. Das entsprechende Residuum ist durch

$$\tilde{r}^{\{j\}} = \boldsymbol{A}u^{\{j\}} - \mu_j u^{\{j\}}$$

gegeben. Zur Bestimmung der Eigenvektoren von $\boldsymbol{A}$ müssen die Lanczos-Vektoren $q^{(i)}$ entweder in einem sekundären Speichermedium abgelegt oder neu berechnet werden. Im letzteren Fall können die Werte von α_i und β_i, die als Elemente der Tridiagonalmatrizen abgespeichert sind, wiederverwendet werden.

Kapitel 3

Speichertechniken für dünnbesetzte Matrizen

Speichertechniken für große dünnbesetzte Matrizen sind von der Besetzungsstruktur der Matrix — und damit von dem speziellen Anwendungsproblem —, dem betrachteten Algorithmus und der Architektur des verwendeten Rechners abhängig [90]. In der Literatur sind verschiedene Varianten zur Speicherung dünnbesetzter Matrizen zu finden [17] [53] [89] [108] [117] [118] [123] [128] [137] [149]. Zur Darstellung der Datenstrukturen wird im folgenden die Strukturart *Satz* (*Record*) in ähnlicher Schreibweise wie in der Programmiersprache PASCAL verwendet [83] [93].

Große dünnbesetzte Matrizen aus technischen oder physikalischen Problemstellungen zeichnen sich durch einen geringen Prozentsatz an Nichtnull-Elementen aus. Speichertechniken für dünnbesetzte Matrizen nutzen diesen Sachverhalt, indem im wesentlichen die Nichtnull-Elemente in von der Besetzungsstruktur der Matrix abhängigen Datenstrukturen abgelegt werden. Einerseits wird auf diese Weise der Speicherbedarf für die Matrix deutlich vermindert. So erfordert eine dünnbesetzte Koeffizientenmatrix aus dem erwähnten FE-Modell der Strukturmechanik mit 25222 Zeilen und 3856386 Nichtnull-Elementen bei Speicherung aller Matrixelemente einen Speicherplatz von 4.7 Gigabytes (2^{30} Bytes), im CDS-Format (*Compressed Diagonal Storage*) dagegen von 668 Megabytes (2^{20} Bytes) und im CRS-Format (*Compressed Row Storage*) von lediglich 44 Megabytes. Andererseits wird der Zugriff auf die Matrixelemente in gepackter Speichertechnik aufwendig; indirekte Adressierung ist erforderlich.

In diesem Kapitel wird zunächst kurz auf den Zusammenhang zwischen Diskretisierungsgitter und Besetzungsstruktur der Koeffizientenmatrix in FE-Modellen eingegangen. Anschließend wird ein Überblick über Datenstrukturen für dünnbesetzte Matrizen gegeben. Schließlich wird die Matrix-Vektor-Multiplikation für die Speicherschemata beschrieben, die in den weiteren Untersuchungen dieser Arbeit betrachtet werden.

3.1 Diskretisierungsgitter und Besetzungsmuster der Matrix

In FE-Modellen ist die maximale Anzahl der Nichtnull-Elemente in einer Zeile der Koeffizientenmatrizen durch die Geometrie des Diskretisierungsgitters und die Art der Elemente, aus denen sich das Gitter zusammensetzt, bestimmt.

Verwendet man z. B. ein dreidimensionales Gitter aus Quadern und legt Knotenpunkte in die Ecken jedes Quaders, so beträgt die maximale Anzahl der nächsten Nachbarn jedes Knotens 26. Abbildung 3.1 veranschaulicht dies anhand eines FE-Gitters aus acht Quadern. Unten rechts ist einer der acht Quader hervorgehoben. Der Mittelpunkt des Gitters hat in der mittleren Ebene acht sowie in der unteren und oberen Ebene je neun nächste Nachbarn, insgesamt also 26. Jede Zeile der Koeffizientenmatrix enthält dann maximal 27 Elemente, die ungleich Null sind. Die Anzahl der Zeilen ist durch die Anzahl der Knotenpunkte des Gitters gegeben und ist gewöhnlich um Größenordnungen höher als die Anzahl der Nichtnull-Elemente in einer Zeile. Weniger als 26 nächste Nachbarn haben lediglich die Randpunkte des Gitters.

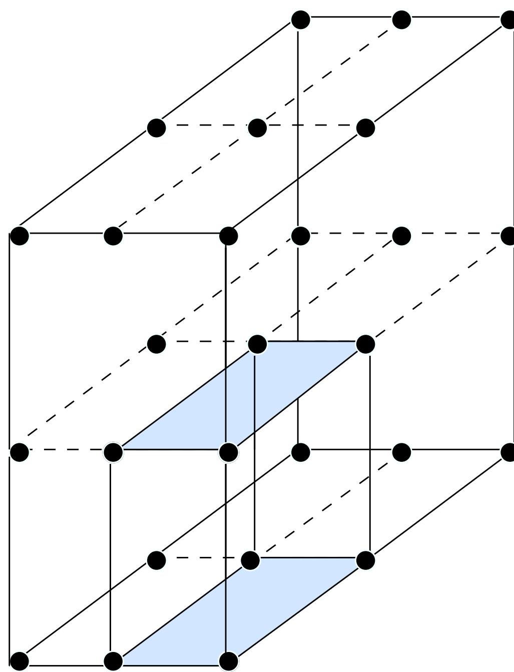


Abbildung 3.1: FE-Gitter aus Quadern

Mehr als 26 nächste Nachbarn können in Diskretisierungsgittern auftreten, die aus komplizierteren Elementen, z. B. Oktaedern, aufgebaut sind. Ferner sind bei der FE-Diskretisierung verschiedene Elemente, Gitterverfeinerungen und die Erhöhung der Anzahl der Knoten pro Element möglich. Weiterhin kann die An-

zahl der Freiheitsgrade pro Element ansteigen, wenn neben den drei Raumrichtungen z. B. noch Rotationsachsen vorhanden sind. Daher variiert die Anzahl der Nichtnull-Elemente pro Zeile bei unregelmäßigen Gittern in der Regel beträchtlich.

3.2 Datenstrukturen

Geeignete Speichertechniken für dünnbesetzte Matrizen berücksichtigen deren Besetzungsstruktur. Dadurch wird zum einen Speicherplatz gespart, zum anderen kann auf die Matrixelemente in möglichst einfacher Weise zugegriffen werden. Im folgenden werden Datenstrukturen für unstrukturierte, blockstrukturierte und bandstrukturierte Matrizen vorgestellt.

3.2.1 Unstrukturierte Matrizen

Matrizen mit unstrukturierter Besetzung stellen den allgemeinen Fall dar. Die Nichtnull-Elemente können sich an beliebigen Positionen innerhalb der Matrix befinden.

Das CRS-Format

Im CRS-Format (*Compressed Row Storage*) werden nacheinander die Nichtnull-Elemente der Matrix zeilenweise in aufeinanderfolgenden Speicherplätzen abgelegt. Wert und Position der Nichtnull-Elemente sind in drei eindimensionalen Feldern mit unterschiedlichem Typ und unterschiedlicher Länge gespeichert.

Abbildung 3.2 zeigt die Datenstruktur einer $n \times n$ Matrix mit e reellen Nichtnull-Elementen.

```
MATRIX = record
    value   : array [1..e] of REAL
    col_ind : array [1..e] of INTEGER
    row_ptr : array [1..n+1] of INTEGER
end record
```

Abbildung 3.2: Datenstruktur des CRS-Formats

Das Feld `value` vom Typ REAL enthält die e Nichtnull-Elemente der Matrix $\boldsymbol{A}$, das Feld `col_ind` die entsprechenden Spaltenindizes. Im dritten Feld `row_ptr` des Typs INTEGER und der Länge $n+1$ ist die Position des Beginns jeder Zeile in den Feldern `value` bzw. `col_ind` abgelegt. Das letzte Element `row_ptr`$(n+1)$ enthält den Eintrag $e+1$, so daß die Anzahl der Nichtnull-Elemente der Zeile i, $i = 1, \ldots, n$, durch `row_ptr`$(i+1)-$`row_ptr`(i) gegeben ist.

Ist das Matrixelement a_{ij} mit $i, j \in \{1, \ldots, n\}$ im Feld `value` an Position $m \in \{1, \ldots, e\}$ gespeichert, d. h.

$$a_{ij} = \texttt{value}(m),$$

so gilt für den Spaltenindex j dieses Matrixelements

$$j = \texttt{col_ind}(m)$$

und für den Zeilenindex i

$$i = \max_{1 \leq k \leq n} \{k \,|\, \texttt{row_ptr}(k) \leq m\}. \tag{3.1}$$

Das Prinzip der CRS-Speichertechnik wird im folgenden anhand der Matrix (3.2) der Ordnung $n = 8$ mit $e = 17$ Nichtnull-Elementen verdeutlicht.

$$\boldsymbol{A} = \begin{pmatrix} \mathbf{20} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{30} & \mathbf{9} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{40} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{10} & \mathbf{50} & 0 & \mathbf{14} & \mathbf{12} & 0 \\ 0 & 0 & 0 & \mathbf{11} & \mathbf{60} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{14} & 0 & \mathbf{70} & \mathbf{15} & 0 \\ 0 & 0 & 0 & 0 & \mathbf{17} & 0 & \mathbf{80} & 0 \\ 0 & 0 & 0 & \mathbf{18} & 0 & 0 & 0 & \mathbf{90} \end{pmatrix} \tag{3.2}$$

In Abbildung 3.3 ist die Speicherung dieser Matrix im CRS-Format dargestellt.

`value:`

20	9	30	40	10	50	14	12	11	60	15	14	70	17	80	18	90

`col_ind:`

1	3	2	3	3	4	6	7	4	5	7	4	6	5	7	4	8
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

`row_ptr:`

1	2	4	5	9	11	14	16	18

Abbildung 3.3: Ein Beispiel für die CRS-Speichertechnik

Die kleinen Zahlen unter dem Feld `col_ind` geben die Feldindizes der Felder `value` und `col_ind` an. Das Feld `row_ptr` verweist auf diejenigen dieser Indizes, die der Position des ersten Nichtnull-Elements einer Zeile in `value` und `col_ind` entsprechen. Zur Verdeutlichung sind die Elemente der einzelnen Zeilen in `value` und `col_ind` durch doppelte Querstriche getrennt. Innerhalb einer Zeile können die Matrixelemente eine beliebige Reihenfolge besitzen. Nach der Assemblierung

der Koeffizientenmatrix in FE-Modellen sind die Nichtnull-Elemente einer Zeile im Speicherschema gewöhnlich nicht nach aufsteigendem Spaltenindex geordnet.

Analog zum CRS-Format ist das CCS-Schema (*Compressed Column Storage*) definiert, das auch als *Harwell-Boeing Sparse Matrix Format* bekannt ist [54]. Im CCS-Format wird die Matrix spaltenweise abgelegt. Das CCS-Speicherschema der Matrix $\boldsymbol{A}$ entspricht dem CRS-Speicherschema der Matrix $\boldsymbol{A}^T$.

Im CCS-Format ist die Matrix in den Feldern `value`, `row_ind` und `col_ptr` abgelegt. `row_ind` enthält die Zeilenindizes der Nichtnull-Elemente, `col_ptr` verweist auf die Position des Beginns jeder Spalte in den Feldern `value` bzw. `row_ind`. Das CCS-Speicherschema für die Matrix (3.2) ist in Abbildung 3.4 dargestellt.

`value:`

20	30	10	40	9	50	14	11	18	17	60	14	70	15	80	12	90

`row_ind:`

1	2	4	3	2	4	6	5	8	7	5	4	6	6	7	4	8
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

`col_ptr:`

1	2	3	6	10	12	14	17	18

Abbildung 3.4: Ein Beispiel für die CCS-Speichertechnik

Eine Variante des CRS-Formats ist das MSR-Schema (*Modified Sparse Row*) [118]. Im MSR-Format werden Wert und Position der Nichtnull-Elemente in zwei eindimensionalen Feldern der Länge $e+1$ abgelegt. Das erste Feld vom Typ REAL enthält auf den ersten n Positionen die Werte der Diagonalelemente und auf Position $n+1$ einen beliebigen Wert. Die übrigen Elemente sind die Werte der restlichen Nichtnull-Elemente, die zeilenweise nacheinander gespeichert werden. Auf den ersten $n+1$ Positionen des zweiten Felds vom Typ INTEGER sind die Werte abgelegt, die den Elementen des Felds `row_ptr` des CRS-Formats entsprechen. Auf den übrigen Positionen folgen die den Werten des ersten Felds zugehörigen Spaltenindizes. Das MSR-Format erlaubt den Zugriff auf die Diagonalelemente ohne indirekte Adressierung. Für Algorithmen, die einen häufigen Zugriff auf die Diagonalelemente erfordern, ist diese Speichertechnik von Vorteil. Ein Beispiel ist die Skalierung mit der Diagonalen aus 2.2.3. Ein MSC-Format (*Modified Sparse Column*) für die spaltenweise Abspeicherung dünnbesetzter Matrizen läßt sich analog definieren.

Ist die Matrix symmetrisch, so ist lediglich die Speicherung der oberen bzw. unteren Dreiecksmatrix erforderlich. Speicherschemata, die dies berücksichtigen, lassen sich entsprechend den CRS- bzw. CCS- und MSR- bzw. MSC-Formaten formulieren. Diese Formate erfordern lediglich die Speicherung von $\frac{e+n}{2}$ Werten der Nichtnull-Elemente und von $\frac{e+n}{2}$ zugehörigen Spalten- bzw. Zeilenindizes.

Zusätzlich wird die Position des jeweiligen Zeilen- bzw. Spaltenbeginns in einem Feld der Länge $n+1$ abgelegt. Dem Vorteil des geringeren Speicherbedarfs gegenüber den oben genannten Schemata steht der Nachteil des komplizierteren Zugriffs auf die Daten gegenüber [17].

Das SICRS-Format

Das SICRS-Format (*Successive Index Compressed Row Storage*) stellt eine im Rahmen dieser Arbeit entwickelte Variante des CRS-Schemas dar. Bei den untersuchten Matrizen aus FE-Modellen wurde beobachtet, daß in den meisten Zeilen der Matrix eine Reihe von Nichtnull-Elementen aufeinanderfolgende Spaltenindizes besitzen. Dem SICRS-Schema liegt die folgende Idee zugrunde. Von k aufeinanderfolgenden Spaltenindizes einer Zeile wird lediglich der niedrigste Index und die Anzahl der aufeinanderfolgenden Indizes, also k, gespeichert. Dies kann sowohl zu einer Reduzierung des Speicherbedarfs für die Spaltenindizes als auch zu einem vereinfachten Zugriff auf die Matrixelemente bei der Matrix-Vektor-Multiplikation führen. Auf die Matrix-Vektor-Multiplikation im SICRS-Format wird in 3.3.2 eingegangen. Speicherplatz für die Spaltenindizes wird gespart, wenn

$$s_{\mathbf{mean}} = \frac{2e}{\tilde{e}} > 2$$

mit $\tilde{e}$ als der Anzahl der im SICRS-Schema gespeicherten niedrigsten Indizes einschließlich der zugehörigen k-Werte erfüllt ist. $s_{\mathbf{mean}}$ bezeichnet die mittlere Anzahl aufeinanderfolgender Spaltenindizes pro Nichtnull-Element und Zeile.

In Abbildung 3.5 ist die Datenstruktur für das SICRS-Format dargestellt.

MATRIX = **record**
  `value` : **array** $[1..e]$ **of** REAL
  `col_ind` : **array** $[1..\tilde{e}]$ **of** INTEGER
  `row_ptr` : **array** $[1..n+1]$ **of** INTEGER
end record

Abbildung 3.5: Datenstruktur des SICRS-Formats

Im Gegensatz zum CRS-Schema verweisen die Elemente des Felds `row_ptr` lediglich auf die Position des Beginns der Zeilen im Feld `col_ind` und nicht auf die Position des Beginns der Zeilen im Feld `value`. Die Anzahl der Elemente in den Feldern `col_ind` und `value` ist gewöhnlich unterschiedlich. Die Position des Zeilenbeginns der i-ten Zeile `row_val`(i) im Feld `value` ist durch

$$\texttt{row_val}(i) = 1 + \sum_{j=2,4,6,\ldots}^{\texttt{row_ptr}(i)} \texttt{col_ind}(j)$$

gegeben.

Abbildung 3.6 veranschaulicht das Prinzip der SICRS-Speichertechnik anhand der Matrix (3.3) mit der Ordnung $n = 8$, $e = 22$ Nichtnull-Elementen und der mittleren Anzahl aufeinanderfolgender Spaltenindizes pro Nichtnull-Element und Zeile $s_{\mathbf{mean}} = 2.2$.

$$\boldsymbol{B} = \begin{pmatrix} \mathbf{20} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{30} & \mathbf{9} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{40} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{10} & \mathbf{50} & \mathbf{11} & \mathbf{14} & \mathbf{12} & \mathbf{18} \\ 0 & 0 & 0 & \mathbf{11} & \mathbf{60} & 0 & 0 & \mathbf{17} \\ 0 & \mathbf{14} & 0 & 0 & 0 & \mathbf{70} & \mathbf{15} & 0 \\ 0 & 0 & 0 & \mathbf{12} & \mathbf{17} & \mathbf{15} & \mathbf{80} & \mathbf{18} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{90} \end{pmatrix} \tag{3.3}$$

Jedem Spaltenindex im Feld `col_ind` ist ein k-Wert zugeordnet, der zur Unterscheidung fett gedruckt ist. Dieser Wert gibt die Anzahl der aufeinanderfolgenden Indizes an.

`value:`

20	30	9	40	10	...	18	11	60	17	14	70	15	12	...	18	90
1	2	3	4	5	...	10	11	12	13	14	15	16	17	...	21	22

`col_ind:`

1	**1**	2	**2**	3	**1**	3	**6**	4	**2**	8	**1**	2	**1**	6	**2**	4	**5**	8	**1**
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

`row_ptr:`

1	3	5	7	9	13	17	19	21

Abbildung 3.6: Ein Beispiel für die SICRS-Speichertechnik

Die zu Beginn des Kapitels genannte Koeffizientenmatrix aus der Strukturmechanik erfordert im CRS-Format einen Speicherplatz von ca. 44 Megabytes. Die mittlere Anzahl aufeinanderfolgender Spaltenindizes pro Nichtnull-Element und Zeile dieser Matrix beträgt $s_{\mathbf{mean}} = 10.2$. Verwendet man das SICRS-Schema zur Speicherung der Matrix, so wird der Speicherbedarf gegenüber dem CRS-Format auf ca. 32 Megabytes reduziert.

Unter Verwendung des hier dargestellten Prinzips lassen sich analoge Speicherschemata für die spaltenweise Abspeicherung dünnbesetzter Matrizen formulieren. Ferner können die Modifikationen des MSR- bzw. MSC-Formats und die Symmetrie der Matrix berücksichtigt werden.

3.2.2 Blockstrukturierte Matrizen

Blockstrukturierte Matrizen setzen sich aus vollbesetzten oder annähernd vollbesetzten quadratischen Blöcken von Nichtnull-Elementen zusammen. Derartige Besetzungsmuster entstehen typischerweise bei der Diskretisierung partieller Differentialgleichungen, wenn pro Gitterpunkt mehrere Freiheitsgrade existieren.

Das BCRS-Format

Das BCRS-Format (*Block Compressed Row Storage*) ist eine Variante des CRS-Schemas, die das Blockmuster blockstrukturierter Matrizen berücksichtigt. Anstelle der Nichtnull-Elemente im CRS-Format werden in der BCRS-Speichertechnik quadratische Blöcke von Nichtnull-Elementen abgelegt, die wie vollbesetzte Matrizen behandelt werden.

Ist n_b die Dimension und e_b die Anzahl dieser *Nichtnull-Blöcke* in der $n \times n$ Matrix $\boldsymbol{A}$, so werden insgesamt $e_\mathrm{b} n_\mathrm{b}^2$ Matrixelemente gespeichert. Die Block-Dimension n_d von $\boldsymbol{A}$ ist als $n_\mathrm{d} = n/n_\mathrm{b}$ definiert.

Die Datenstruktur des BCRS-Formats ist in Abbildung 3.7 dargestellt.

MATRIX = **record**
    `value` : **array** $[1..e_\mathrm{b}, 1..n_\mathrm{b}, 1..n_\mathrm{b}]$ **of** REAL
    `col_ind` : **array** $[1..e_\mathrm{b}]$ **of** INTEGER
    `row_blk` : **array** $[1..n_\mathrm{d}+1]$ **of** INTEGER
end record

Abbildung 3.7: Datenstruktur des BCRS-Formats

Im Feld `value` sind die Nichtnull-Blöcke in der Reihenfolge der Block-Zeilen ähnlich der zeilenweisen Speicherung der Nichtnull-Elemente im CRS-Format abgelegt. Das Feld `col_ind` des BCRS-Schemas enthält die entsprechenden Spaltenindizes in der ursprünglichen Matrix $\boldsymbol{A}$ des $(1,1)$-Elements der Nichtnull-Blöcke. Das $(1,1)$-Element ist das erste Element der ersten Zeile und der ersten Spalte eines Blocks. Die Elemente des Felds `row_blk` weisen auf die Position des Beginns der Block-Zeilen in `value` und `col_ind`.

Ist n_b groß, kann durch das BCRS-Format gegenüber dem CRS-Schema signifikant Speicherplatz und Aufwand für indirekte Adressierung gespart werden. Zum einen enthalten die Felder `col_ind` und `row_blk` deutlich weniger Elemente als die entsprechenden Felder des CRS-Formats, zum anderen sind die Nichtnull-Blöcke als vollbesetzte Matrizen abgelegt. Sind die Nichtnull-Blöcke nicht vollbesetzt, so sind im Feld `value` des BCRS-Schemas mehr Elemente als im entsprechenden Feld des CRS-Formats gespeichert. Gewöhnlich enthalten die Blöcke blockstrukturierter Matrizen jedoch nur wenige Null-Elemente.

Analog zum BCRS-Format läßt sich ein BCCS-Format (*Block Compressed*

Column Storage) zur Speicherung in der Reihenfolge der Blockspalten formulieren.

3.2.3 Bandstrukturierte Matrizen

Die Nichtnull-Elemente einer bandstrukturierten Matrix sind im wesentlichen entlang einer kleinen Anzahl von Diagonalen in der Matrix angeordnet. Matrizen mit Bandstruktur treten bei der FE- oder FD-Diskretisierung (*Finite Difference*) mit regelmäßigen Gittern auf. Neben den Nichtnull-Elementen werden in Schemata zur Speicherung bandstrukturierter Matrizen gewöhnlich einige Null-Elemente abgelegt.

Das DIA-Format

Das DIA-Format (*Diagonal Format*) verwendet zur Speicherung von Bandmatrizen die Datenstruktur aus Abbildung 3.8 [117] [118].

MATRIX = **record**
  `value` : **array** $[1..n, 1..n_{\text{dia}}]$ **of** REAL
  `offset` : **array** $[1..n_{\text{dia}}]$ **of** INTEGER
end record

Abbildung 3.8: Datenstruktur des DIA-Formats

n_{dia} ist die Anzahl der Diagonalen in der Matrix, die Nichtnull-Elemente enthalten. In den Spalten des Felds `value` sind die Werte der Elemente dieser Diagonalen gespeichert; die Elemente des Felds `offset` geben die Position der Diagonalen bezogen auf die Hauptdiagonale der Matrix an.

In Abbildung 3.9 ist das DIA-Schema für die Matrix (3.4) dargestellt.

$$\boldsymbol{A} = \begin{pmatrix} \mathbf{20} & 0 & \mathbf{7} & 0 & 0 & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{30} & 0 & \mathbf{9} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{6} & \mathbf{40} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{7} & \mathbf{50} & 0 & \mathbf{14} & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{60} & 0 & \mathbf{2} & 0 \\ \mathbf{19} & 0 & 0 & 0 & \mathbf{14} & \mathbf{70} & 0 & \mathbf{8} \\ 0 & \mathbf{17} & 0 & 0 & 0 & \mathbf{4} & \mathbf{80} & 0 \\ 0 & 0 & \mathbf{11} & 0 & 0 & 0 & \mathbf{10} & \mathbf{90} \end{pmatrix} \tag{3.4}$$

Alle Spalten von `value` außer der Spalte mit den Elementen der Hauptdiagonalen enthalten weniger als n Elemente. In jeder Spalte von `value` sind ab der Zeile Elemente der Matrix gespeichert, die in der Matrix der Zeile des ersten Elements der zugehörigen Diagonalen entspricht. Alle Werte vor dieser Zeile existieren nicht und sind in den Spalten von `value` durch „*“ gekennzeichnet.

value:

*	*	20	7
*	1	30	9
*	6	40	0
*	7	50	14
*	0	60	2
19	14	70	8
17	4	80	*
11	10	90	*

offset:

-5	-1	0	2

Abbildung 3.9: Ein Beispiel für die DIA-Speichertechnik

Der erste Wert -5 des Felds `offset` z. B. bedeutet, daß die erste Spalte von `value` die Werte der fünften Diagonalen unterhalb der Hauptdiagonalen enthält.

Ein dem DIA-Format ähnliches Schema ist das CDS-Format (*Compressed Diagonal Storage*) [17] [51]. In dieser Speichertechnik werden alle Diagonalen des Bereichs der Matrix gespeichert, der durch die letzten Diagonalen unterhalb und oberhalb der Hauptdiagonalen, die Nichtnull-Elemente enthalten, eingegrenzt ist. Das CDS-Format ist für Matrizen geeignet, in denen dieser Bereich annähernd vollbesetzt ist.

McBryan und Van de Velde verwenden ein verteiltes DIA- bzw. CDS-Datenformat für Parallelrechner mit Hypercube-Topologie [97].

Eine Verallgemeinerung des DIA- bzw. CDS-Formats für unregelmäßig besetzte Matrizen, die *Streifenspeicherung*, wurde in [60], [98], [99] und [100] hinsichtlich der Eignung für Vektor- und Parallelrechner untersucht.

Ein Streifen S in einer $n \times n$ Matrix $\boldsymbol{A}$ ist definiert als die Menge der Paare $S = \{(i, \sigma(i)) \,|\, i \in J \subseteq J_n\}$ mit $J_n = \{1, \ldots, n\}$. $(i, \sigma(i))$ gibt den Zeilen- und Spaltenindex eines Elements von S in $\boldsymbol{A}$ an. Ferner ist σ streng monoton wachsend, d. h. es gilt

$$[(i, \sigma(i)) \in S\,,\ (j, \sigma(j)) \in S\,] \Longrightarrow [i < j \Longrightarrow \sigma(i) < \sigma(j)].$$

Zwei Streifen $S_1 = \{(i, \sigma_1(i))\}$ und $S_2 = \{(j, \sigma_2(j))\}$ sind durch

$$S_1 < S_2 \Longleftrightarrow \forall (i, \sigma_1(i)) \in S_1,\ (j, \sigma_2(j)) \in S_2\ :\ [i \leq j \Longrightarrow \sigma_1(i) < \sigma_2(j)]$$

geordnet. Diese Eigenschaft wird als *Non-Intersection-Property* bezeichnet.

Zur Speicherung der Streifen sind zwei zweidimensionale Felder

$$\texttt{value}(i,k) = \begin{cases} a_{i,\sigma_k(i)} & \text{für } (i, \sigma_k(i)) \in S_k \\ 0 & \text{sonst} \end{cases}$$

und

$$\texttt{sigma}(i,k) = \begin{cases} \sigma_k(i) & \text{für } (i, \sigma_k(i)) \in S_k \\ 0 & \text{sonst} \end{cases}$$

erforderlich. Eine mögliche Datenstruktur für das Streifen-Format ist in Abbildung 3.10 dargestellt.

```
MATRIX = record
        value : array [1..n, 1..n_stripes] of REAL
        sigma : array [1..n, 1..n_stripes] of INTEGER
end record
```

Abbildung 3.10: Datenstruktur des Streifen-Formats

n_{stripes} ist die Anzahl der Streifen in der Matrix. Für dieselbe Matrix können verschiedene Streifenstrukturen konstruiert werden. Günstig ist eine Streifenstruktur mit minimaler Streifenanzahl.

Abbildung 3.11 zeigt ein Beispiel für die Streifenspeicherung anhand der Matrix (3.5).

$$\boldsymbol{A} = \begin{pmatrix} \mathbf{20}_3 & \mathbf{7}_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{1}_2 & \mathbf{30}_3 & \mathbf{9}_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{6}_2 & \mathbf{40}_3 & 0 & 0 & \mathbf{14}_4 & 0 & 0 \\ 0 & 0 & \mathbf{7}_2 & \mathbf{50}_3 & 0 & 0 & 0 & 0 \\ \mathbf{19}_1 & 0 & 0 & 0 & \mathbf{60}_3 & 0 & \mathbf{2}_4 & 0 \\ 0 & 0 & 0 & \mathbf{14}_2 & 0 & \mathbf{70}_3 & 0 & \mathbf{8}_4 \\ 0 & \mathbf{17}_1 & 0 & 0 & \mathbf{4}_2 & 0 & \mathbf{80}_3 & 0 \\ 0 & 0 & 0 & \mathbf{11}_1 & 0 & \mathbf{10}_2 & 0 & \mathbf{90}_3 \end{pmatrix} \tag{3.5}$$

Die Matrix wird in Form von vier Streifen gespeichert. Der tiefgestellte Index in Matrix (3.5) bezeichnet die Streifennummer.

`value:`

S_1	S_2	S_3	S_4
0	0	20	7
0	1	30	9
0	6	40	14
0	7	50	0
19	0	60	2
0	14	70	8
17	4	80	0
11	10	90	0

`sigma:`

S_1	S_2	S_3	S_4
0	0	1	2
0	1	2	3
0	2	3	6
0	3	4	0
1	0	5	7
0	4	6	8
2	5	7	0
4	6	8	0

Abbildung 3.11: Ein Beispiel für die Streifen-Speichertechnik

Ist die Non-Intersection-Property gegeben, so wird ein Matrix-Vektor-Produkt der Form $y = \boldsymbol{A}x$ bei Streifenspeicherung der Matrix wie folgt berechnet. Jedes Element $a_{i,\sigma_k(i)}$ von $\boldsymbol{A}$ im Streifen S_k wird mit $x_{\sigma_k(i)}$ multipliziert und das Produkt jeweils zu y_i addiert.

Das JDS-Format

Das JDS-Format (*Jagged Diagonal Storage*) wird in [116] für die Implementierung iterativer Methoden auf Vektor- und Parallelrechnern genutzt. Eine vereinfachte Form des JDS-Schemas ist das ITPACK-Format, das auch ELLPACK-ITPACK- oder Purdue-Format genannt wird [17] [118].

Beim ITPACK-Schema werden die Nichtnull-Elemente jeder Zeile nach links verschoben. Dies ist in (3.6) veranschaulicht.

$$\begin{pmatrix} \mathbf{20} & 0 & 0 & \mathbf{17} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{30} & \mathbf{9} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{40} & 0 & \mathbf{7} & 0 & \mathbf{13} & 0 \\ 0 & 0 & \mathbf{10} & \mathbf{50} & 0 & \mathbf{14} & \mathbf{12} & 0 \\ 0 & 0 & 0 & \mathbf{11} & \mathbf{60} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{14} & 0 & \mathbf{70} & \mathbf{15} & 0 \\ 0 & 0 & 0 & 0 & \mathbf{17} & 0 & \mathbf{80} & 0 \\ 0 & 0 & 0 & \mathbf{18} & 0 & 0 & 0 & \mathbf{90} \end{pmatrix} \longrightarrow \begin{pmatrix} \mathbf{20} & \mathbf{17} & 0 & 0 \\ \mathbf{30} & \mathbf{9} & 0 & 0 \\ \mathbf{40} & \mathbf{7} & \mathbf{13} & 0 \\ \mathbf{10} & \mathbf{50} & \mathbf{14} & \mathbf{12} \\ \mathbf{11} & \mathbf{60} & 0 & 0 \\ \mathbf{14} & \mathbf{70} & \mathbf{15} & 0 \\ \mathbf{17} & \mathbf{80} & 0 & 0 \\ \mathbf{18} & \mathbf{90} & 0 & 0 \end{pmatrix} \tag{3.6}$$

z_{max} bezeichnet die maximale Anzahl von Nichtnull-Elementen, die in einer Zeile der Matrix auftritt. Alle Zeilen, die weniger als z_{max} Nichtnull-Elemente aufweisen, werden rechts mit Nullen aufgefüllt.

Abbildung 3.12 zeigt eine mögliche Datenstruktur des ITPACK-Formats. Die zusammengeschobenen Zeilen der Matrix werden in zwei zweidimensionalen Feldern gespeichert. `value` enthält die Werte der Nichtnull-Elemente, `col_ind` die zugehörigen Spaltenindizes. Der Zeilenindex eines Elements im Feld `value` entspricht dem Zeilenindex des Elements in der ursprünglichen Matrix.

```
MATRIX = record
        value   : array [1..n, 1..z_max] of REAL
        col_ind : array [1..n, 1..z_max] of INTEGER
end record
```

Abbildung 3.12: Datenstruktur des ITPACK-Formats

Die Speicherung der Matrix (3.6) im ITPACK-Format ist in Abbildung 3.13 dargestellt.

In Abbildung 3.13 sind die zusammengeschobenen Zeilen der Matrix in den Zeilen von `value` und `col_ind` gespeichert. Ebenso ist es möglich, die komprimierten Zeilen der Matrix in den Spalten von `value` und `col_ind` abzulegen und zusätzlich ein Feld `num_el` mitzuführen, dessen Elemente jeweils die Anzahl der Nichtnull-Elemente pro Zeile der Matrix angeben. Dieses modifizierte ITPACK-Schema kann gegenüber dem ursprünglichen von Vorteil sein, wenn in `value` und `col_ind` sehr viele Nullen auftreten (s. auch 3.3.3). Abbildung 3.14 zeigt die Datenstruktur des modifizierten ITPACK-Formats.

value:

20	17	0	0
30	9	0	0
40	7	13	0
10	50	14	12
11	60	0	0
14	70	15	0
17	80	0	0
18	90	0	0

col_ind:

1	4	0	0
2	3	0	0
3	5	7	0
3	4	6	7
4	5	0	0
4	6	7	0
5	7	0	0
4	8	0	0

Abbildung 3.13: Ein Beispiel für die ITPACK-Speichertechnik

MATRIX = **record**
  `value` : **array** $[1..z_{\max}, 1..n]$ **of** REAL
  `col_ind` : **array** $[1..z_{\max}, 1..n]$ **of** INTEGER
  `num_el` : **array** $[1..n]$ **of** INTEGER
end record

Abbildung 3.14: Datenstruktur des modifizierten ITPACK-Formats

Eine Kombination des ITPACK- und des CRS-Formats wird von Fernandes und Girdinio in [61] vorgeschlagen. Bis zu einer mittleren Anzahl von Nichtnull-Elementen pro Zeile werden die Matrixelemente im ITPACK-Format abgelegt; die übrigen Nichtnull-Elemente — bei vielen dünnbesetzten Matrizen aus Diskretisierungsansätzen betrifft dies nur einige Zeilen — werden in einem CRS-ähnlichen Schema gespeichert.

Das eigentliche JDS-Format hat gegenüber dem ITPACK-Format den Vorteil, daß keine Null-Elemente gespeichert werden. Die komprimierten Zeilen der Matrix (3.6) werden zunächst nach der Anzahl der Nichtnull-Elemente pro Zeile in absteigender Reihenfolge geordnet. Diese Permutation ist in (3.7) dargestellt.

$$\begin{pmatrix} \mathbf{20} & \mathbf{17} & 0 & 0 \\ \mathbf{30} & \mathbf{9} & 0 & 0 \\ \mathbf{40} & \mathbf{7} & \mathbf{13} & 0 \\ \mathbf{10} & \mathbf{50} & \mathbf{14} & \mathbf{12} \\ \mathbf{11} & \mathbf{60} & 0 & 0 \\ \mathbf{14} & \mathbf{70} & \mathbf{15} & 0 \\ \mathbf{17} & \mathbf{80} & 0 & 0 \\ \mathbf{18} & \mathbf{90} & 0 & 0 \end{pmatrix} \longrightarrow \begin{pmatrix} \mathbf{10} & \mathbf{50} & \mathbf{14} & \mathbf{12} \\ \mathbf{40} & \mathbf{7} & \mathbf{13} & 0 \\ \mathbf{14} & \mathbf{70} & \mathbf{15} & 0 \\ \mathbf{20} & \mathbf{17} & 0 & 0 \\ \mathbf{11} & \mathbf{60} & 0 & 0 \\ \mathbf{30} & \mathbf{9} & 0 & 0 \\ \mathbf{17} & \mathbf{80} & 0 & 0 \\ \mathbf{18} & \mathbf{90} & 0 & 0 \end{pmatrix} \tag{3.7}$$

Anschließend werden lediglich die Nichtnull-Elemente der komprimierten und permutierten Matrix spaltenweise in dem eindimensionalen Feld `value` abgelegt. Die Anzahl dieser Spalten, der *Jagged Diagonals*, entspricht der Anzahl der Nichtnull-Elemente in der ersten Zeile der komprimierten und permutierten Matrix. Die zugehörigen Spaltenindizes werden in dem Feld `col_ind` gespeichert. Die Ele-

mente eines dritten Felds `jd_ptr` verweisen auf den Beginn jeder Jagged Diagonal in `value` und `col_ind`. Ein viertes Feld `perm` enthält die Information, wie die ursprüngliche Reihenfolge der Zeilen wiederhergestellt wird. Abbildung 3.15 zeigt die Datenstruktur des JDS-Schemas.

MATRIX = **record**
  `value` : **array** $[1..e]$ **of** REAL
  `col_ind` : **array** $[1..e]$ **of** INTEGER
  `jd_ptr` : **array** $[1..z_{\max}+1]$ **of** INTEGER
  `perm` : **array** $[1..n]$ **of** INTEGER
end record

Abbildung 3.15: Datenstruktur des JDS-Formats

In Abbildung 3.16 ist die Speicherung der Matrix (3.6) im JDS-Format dargestellt.

`value:`

10	40	14	20	11	30	17	18	50	7	70	...	90	14	13	15	12

`col_ind:`

3	3	4	1	4	2	5	4	4	5	6	...	8	6	7	7	7
1	2	3	4	5	6	7	8	9	10	11	...	16	17	18	19	20

`jd_ptr:`

1	9	17	20	21

`perm:`

4	3	6	1	5	2	7	8

Abbildung 3.16: Ein Beispiel für die JDS-Speichertechnik

Die kleinen Indizes geben die Positionen in `value` und `col_ind` an, auf die das Feld `jd_ptr` verweist. Das letzte Element des Felds `jd_ptr` enthält den Wert $e+1$. Die Zahl 4 an Position 1 des Felds `perm` z. B. bedeutet, daß die erste Zeile der permutierten Matrix der vierten Zeile der ursprünglichen Matrix entspricht.

3.2.4 Profil-Matrizen

Bei Profil-Matrizen, die auch Skyline- oder variable Bandmatrizen genannt werden [53], besitzen die Nichtnull-Elemente einer Zeile bzw. Spalte zum überwiegenden Teil aufeinanderfolgende Spalten- bzw. Zeilenindizes und sind um die Diagonalelemente angeordnet. Zwischen dem ersten und letzten Nichtnull-Element einer Zeile bzw. Spalte befinden sich keine oder nur wenige Null-Elemente. Die Struktur einer unsymmetrischen Profil-Matrix ist in Abbildung 3.17 dargestellt.

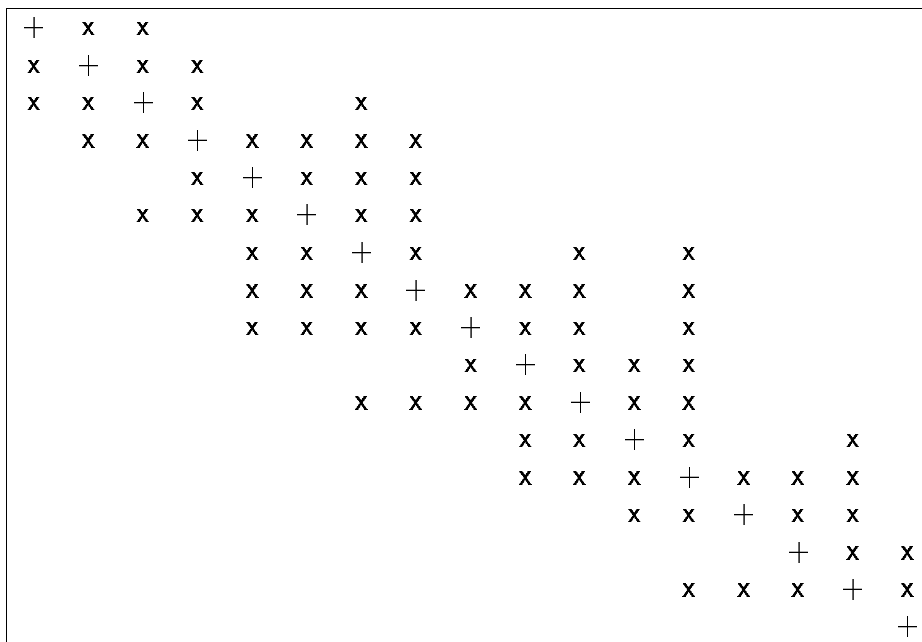


Abbildung 3.17: Struktur einer unsymmetrischen Profil-Matrix

Das SKS-Format

Datenformate für Profil-Matrizen sind hauptsächlich für direkte Methoden zur Lösung von linearen Gleichungssystemen von Bedeutung [53]. Der Vorteil dieser Matrizen liegt darin, daß die Struktur der Matrix während der Gauß-Elimination erhalten bleibt. Da in dieser Arbeit iterative Verfahren betrachtet werden, werden die SKS-Schemata (*Skyline Storage*) nur kurz skizziert.

Bei symmetrischen Profil-Matrizen ist es lediglich erforderlich, die untere Dreiecksmatrix zu speichern. Im SKS-Format für symmetrische Matrizen werden die Werte des ersten Nichtnull-Elements bis zum Diagonalelement zeilenweise in einem eindimensionalen Feld `value` vom Typ REAL abgelegt. Die Elemente eines zweiten Felds `row_ptr` vom Typ INTEGER verweisen auf die Position des Beginns jeder Zeile in `value`. Die Spaltenindizes der Nichtnull-Elemente sind durch die Position der Diagonalelemente gegeben und brauchen nicht gespeichert zu werden. Analog zur zeilenweisen Speicherung kann die untere Dreiecksmatrix auch spaltenweise abgelegt werden.

Eine unsymmetrische Matrix wie die Matrix in Abbildung 3.17 kann folgendermaßen gespeichert werden. Die untere Dreiecksmatrix wird im zeilenorientierten SKS-Format abgelegt, die obere Dreiecksmatrix im spaltenorientierten SKS-Format. Diese beiden getrennten Unterstrukturen können auf verschiedene Weise miteinander verknüpft werden. In [117] wird vorgeschlagen, jede Zeile der unteren Dreiecksmatrix und jede Spalte der oberen Dreiecksmatrix kontinuierlich in einem eindimensionalen Feld `value` vom Typ REAL abzulegen. In diesem Fall

ist ein zusätzliches Feld vom Typ INTEGER erforderlich, dessen Elemente auf die Position der Diagonalelemente verweisen und damit gleichzeitig die Elemente der unteren Dreiecksmatrix von den Elementen der oberen Dreiecksmatrix trennen.

3.3 Die Matrix-Vektor-Multiplikation

In den iterativen Methoden aus Kapitel 2 treten Matrix-Vektor-Produkte der Form

$$y = \boldsymbol{A}x \quad \text{und} \quad y = \boldsymbol{A}^T x$$

auf. Da die Hauptarbeit in jedem Iterationsschritt gewöhnlich in der Berechnung dieser Matrix-Vektor-Produkte besteht, sind effiziente Algorithmen für die Matrix-Vektor-Multiplikation in Abhängigkeit von der Speichertechnik für die dünnbesetzte Matrix erforderlich.

Im folgenden werden Algorithmen der Matrix-Vektor-Multiplikation für die Speicherschemata, die in den weiteren Untersuchungen dieser Arbeit verwendet werden, beschrieben. Zunächst wird die Matrix-Vektor-Multiplikation im CRS- bzw. CCS-Format betrachtet, da diese Schemata für allgemeine dünnbesetzte Matrizen geeignet sind. Das CRS- bzw. CCS-Format für symmetrische Matrizen wird nicht berücksichtigt, da einerseits der Zugriff auf die Matrix bei der Matrix-Vektor-Multiplikation gegenüber den allgemeineren Schemata aufwendiger wird [17] und andererseits die entwickelten Methoden für dünnbesetzte Matrizen dieser Arbeit nicht auf den symmetrischen Fall beschränkt sein sollen. Ferner führt das symmetrische Schema gegenüber dem allgemeinen Format auf einem Parallelrechner mit verteiltem Speicher gewöhnlich zu zusätzlichem Datenaustausch zwischen den einzelnen Prozessoren, wenn die Matrix zeilen- bzw. spaltenweise auf die Prozessoren verteilt und die Berechnung des Matrix-Vektor-Produkts zeilen- bzw. spaltenweise durchgeführt wird. Auf Parallelrechnern wird das allgemeine CRS- bzw. CCS-Format u. a. in [57], [76] und [86] verwendet. Neben diesen Schemata wird die Matrix-Vektor-Multiplikation im SICRS-Format und ITPACK-Format beschrieben. Im betrachteten FE-Modell aus der Umwelttechnik sind die Koeffizientenmatrizen in letzterem Schema abgelegt [145].

3.3.1 CRS-Format

Der Ablauf der Matrix-Vektor-Multiplikation $y = \boldsymbol{A}x$ im CRS-Format ist in Abbildung 3.18 als Pseudocode dargestellt. Dem Algorithmus liegt die Datenstruktur aus Abbildung 3.2 zugrunde.

Da in der inneren Schleife nacheinander auf die Nichtnull-Elemente jeder Zeile zugegriffen wird, ist die aufwendige Bestimmung des Zeilenindex nach (3.1) nicht erforderlich. Der Vektor x des Matrix-Vektor-Produkts wird über das Spaltenindex-Feld `col_ind` indirekt adressiert. Da lediglich die Nichtnull-Elemente der Matrix in die Berechnung eingehen, beträgt die Komplexität der Matrix-

```
for i = 1,...,n do
    y(i) = 0
    for j = row_ptr(i),...,row_ptr(i+1) - 1 do
        y(i) = y(i) + value(j) · x(col_ind(j))
    end for
end for
```

Abbildung 3.18: Die Matrix-Vektor-Multiplikation im CRS-Format

Vektor-Multiplikation $\mathcal{O}(e)$ gegenüber $\mathcal{O}(n^2)$ bei vollständiger Speicherung der Koeffizientenmatrix.

Abbildung 3.19 zeigt den Pseudocode der Matrix-Vektor-Multiplikation für $y = \boldsymbol{A}^T x$. Ist $\boldsymbol{A}$ im CRS-Format gespeichert, so liegt die Transponierte $\boldsymbol{A}^T$ im CCS-Format vor. Daher kann bei Verwendung der Algorithmen aus den Abbildungen 3.18 und 3.19 für das Matrix-Vektor-Produkt mit $\boldsymbol{A}$ und mit $\boldsymbol{A}^T$ dieselbe Datenstruktur genutzt werden. `col_ind` enthält die Zeilenindizes der Nichtnull-Elemente von $\boldsymbol{A}^T$; `row_ptr` verweist auf die Position des Beginns jeder Spalte von $\boldsymbol{A}^T$ in `value` und `col_ind`. Bei der spaltenweisen Matrix-Vektor-Multiplikation $y = \boldsymbol{A}^T x$ wird der Ergebnisvektor y über `col_ind` indirekt adressiert.

```
for i = 1,...,n do
    y(i) = 0
end for
for i = 1,...,n do
    for j = row_ptr(i),...,row_ptr(i+1) - 1 do
        y(col_ind(j)) = y(col_ind(j)) + value(j) · x(i)
    end for
end for
```

Abbildung 3.19: Die Matrix-Vektor-Multiplikation mit der Transponierten im CCS-Format

3.3.2 SICRS-Format

Der Pseudocode der Matrix-Vektor-Multiplikation für $y = \boldsymbol{A}x$ im SICRS-Format ist in Abbildung 3.20 dargestellt. Im Algorithmus wird die Datenstruktur aus Abbildung 3.5 verwendet. `pos_val` gibt die `col_ind`(j) zugehörige Position im Feld `value` an. In der Schleife mit dem Laufindex j wird j nach jedem Durchlauf um 2 erhöht. In der innersten Schleife sind `pos_val` und `col_ind`(j) konstant, so daß nacheinander auf aufeinanderfolgende Elemente des Felds `value` und des Vektors x zugegriffen wird. Für hohe Werte von s_{mean} kann dies für eine Implementierung auf einem Rechner mit Cache-Speicher und Pipeline-Verarbeitung von Vorteil sein.

```
pos_val = 1
for i = 1,...,n do
    y(i) = 0
    for j = row_ptr(i),...,row_ptr(i+1) - 1 step 2 do
        for k = 0,...,col_ind(j+1) - 1 do
            y(i) = y(i) + value(pos_val + k) · x(k + col_ind(j))
        end for
        pos_val = pos_val + col_ind(j+1)
    end for
end for
```

Abbildung 3.20: Die Matrix-Vektor-Multiplikation im SICRS-Format

3.3.3 ITPACK-Format

Im Gegensatz zu den oben beschriebenen CRS-Schemata sind die Nichtnull-Elemente der Matrix im ITPACK-Format in zwei zweidimensionalen Feldern gespeichert. Abbildung 3.21 zeigt den Pseudocode der Matrix-Vektor-Multiplikation $y = \boldsymbol{A}x$ in letzterem Format bei spaltenweisem Zugriff auf die Felder `value` und `col_ind`. Dem Algorithmus liegt die Datenstruktur aus Abbildung 3.12 zugrunde.

```
for i = 1,...,n do
    y(i) = 0
end for
for j = 1,...,z_max do
    for i = 1,...,n do
        y(i) = y(i) + value(i,j) · x(col_ind(i,j))
    end for
end for
```

Abbildung 3.21: Die Matrix-Vektor-Multiplikation im ITPACK-Format, spaltenorientierter Zugriff

Die große Länge n der Vektoren und Matrixspalten in der inneren Schleife ist insbesondere für eine Implementierung auf Vektorrechnern günstig [106]. Auf den Vektor x wird über das Feld `col_ind` mit indirekter Adressierung zugegriffen.

Im Pseudocode aus Abbildung 3.22 wird das Matrix-Vektor Produkt $y = \boldsymbol{A}x$ zeilenweise berechnet. Im Algorithmus wird die Datenstruktur aus Abbildung 3.14 verwendet.

Gegenüber dem spaltenweisen Vorgehen werden gewöhnlich Operationen gespart, da pro Zeile lediglich die Nichtnull-Elemente in die Berechnung eingehen. Die Anzahl der Nichtnull-Elemente pro Zeile beschränkt jeweils die Anzahl der Durchläufe der inneren Schleife, die bei großen dünnbesetzten Matrizen deut-

```
for i = 1,...,n do
    y(i) = 0
    for j = 1,...,num_el(i) do
        y(i) = y(i) + value(j,i) · x(col_ind(j,i))
    end for
end for
```

Abbildung 3.22: Die Matrix-Vektor-Multiplikation im ITPACK-Format, zeilenorientierter Zugriff

lich kleiner als n ist. Auf Vektorrechnern kann dies gegenüber dem oben beschriebenen Algorithmus ein Nachteil sein. Auf anderen Rechnerarchitekturen — z. B. massiv-parallelen Systemen mit Standardprozessoren — kann sich dagegen das Einsparen von Operationen günstig auswirken.

Kapitel 4

Parallelverarbeitung

An dieser Stelle werden Konzepte und Begriffe der Parallelverarbeitung erläutert, die in dieser Arbeit verwendet werden. Zunächst wird kurz auf Eigenschaften heutiger Parallelrechner eingegangen; insbesondere wird das massiv-parallele System INTEL PARAGON XP/S 10 beschrieben. Anschließend werden Merkmale und Sprachkonzepte für parallele Algorithmen vorgestellt.

4.1 Parallelrechner

Kriterien zur Klassifizierung von Parallelrechnern sind in [63] und [79] beschrieben. In der Klasse der MIMD-Rechner (*Multiple Instruction Stream Multiple Data Stream*) werden Systeme mit gemeinsamem und mit verteiltem Speicher unterschieden. Im folgenden werden Eigenschaften dieser heutzutage verbreiteten Systeme erläutert.

4.1.1 Systeme mit gemeinsamem Speicher

Parallelrechner mit gemeinsamem Speicher (*Shared Memory)* verfügen über einen einzigen physikalischen Hauptspeicher mit globalem Adreßraum. Jeder Prozessor eines derartigen Systems kann auf jedes Datum im gemeinsamen Speicher zugreifen. Informationen zwischen den einzelnen Prozessoren werden über gemeinsame Variablen im Hauptspeicher ausgetauscht. Da nur eine begrenzte Anzahl an Verbindungspfaden von den Prozessoren zum gemeinsamen Speicher besteht, ist die Anzahl der Prozessoren von Shared-Memory-Systemen beschränkt; ab einer bestimmten Prozessorzahl erhöht sich die Leistung des Systems nicht mehr.

Die CRAY Y-MP8/864 der Forschungszentrum Jülich GmbH ist ein MIMD-Rechner mit gemeinsamem Speicher. Das System besteht aus acht Prozessoren und einem Hauptspeicher von 64 Megaworten zu je 64 Bit. Die einzelnen Prozessoren sind über den gemeinsamen Speicher und gemeinsame Register gekoppelt. Nähere Einzelheiten zu Shared-Memory-Rechnern der Firma CRAY sind in [33]

und [79] zu finden.

4.1.2 Systeme mit verteiltem Speicher

Im Gegensatz zu Shared-Memory-Systemen besitzen die Prozessoren eines Parallelrechners mit verteiltem Speicher (*Distributed Memory*) eigene lokale Speicher; ein globaler physikalischer Adreßraum ist nicht vorhanden. Kommunikation zwischen den Prozessoren geschieht durch Nachrichten, die über das Verbindungsnetzwerk der Prozessoren ausgetauscht werden. Das zugehörige Programmiermodell wird *Message-Passing* genannt. Da Message-Passing auf einer niedrigen Abstraktionsebene stattfindet, ist die Programmierung von Distributed-Memory-Rechnern gegenüber Systemen mit gemeinsamem Speicher erschwert. Neuere Programmiermodelle für Parallelrechner mit verteiltem Speicher gehen von einem gemeinsamen logischen Hauptspeicher (*Virtually Shared Memory*) der Prozessoren mit globalem logischen Adreßraum aus [74] [96]. Unter diesen Modellen können Distributed-Memory-Systeme in ähnlicher Weise wie Parallelrechner mit gemeinsamem Speicher programmiert werden.

Die Rechenleistung von Systemen mit verteiltem Speicher skaliert mit der Anzahl der Prozessoren, jedoch ist der Aufwand für den Datenaustausch zwischen den Prozessoren groß. Daher sind leistungsfähige Verbindungsnetzwerke erforderlich. Übliche Netzwerk-Topologien besitzen Hypercube-, Baum-, Ring-, oder Gitter-Struktur [34] [112]. In heutiger Zeit existieren Distributed-Memory-Rechner mit bis zu einigen tausend Prozessoren. Diese Systeme werden durch den Begriff *massiv-parallel* charakterisiert.

4.1.3 Der Rechner INTEL PARAGON XP/S 10

Die Performance-Untersuchungen der im Rahmen dieser Arbeit entwickelten parallelen Algorithmen wurden auf dem Rechner INTEL PARAGON XP/S 10 der Forschungszentrum Jülich GmbH durchgeführt. Im folgenden werden einige Merkmale dieses Parallelrechners beschrieben. Einzelheiten zur System- und Prozessorarchitektur sind in [30], [34] und [102] zu finden.

Der PARAGON ist ein massiv-paralleles System mit verteiltem Speicher. Sein Vorgänger ist der INTEL iPSC/860, ein Parallelrechner mit Hypercube-Topologie [31] [32]. Gegenüber diesem System weist der PARAGON ein Verbindungsnetzwerk höherer Bandbreite und eine verbesserte Prozessorleistung auf.

Der PARAGON besitzt in seiner größten ausgelieferten Ausbaustufe 1984 Prozessoren [34], die durch ein Netzwerk in Form eines zweidimensionalen Gitters miteinander verbunden sind. Die maximale Bandbreite des Netzwerks beträgt 200 Megabytes/s. Neben dem Rechenprozessor, der im wesentlichen arithmetische und logische Operationen durchführt, ist jeweils ein Kommunikationsprozessor vorhanden. Dieser übernimmt die aufwendigen Protokoll- und Verwaltungsarbeiten für den Datenaustausch. Beide Prozessoren sind vom Typ i860 XP.

Jedem solchen Prozessor-Paar ist ein lokaler Hauptspeicher von 32 Megabytes zugeordnet.

Der RISC-Prozessor i860 XP (*Reduced Instruction Set Computer*) besitzt eine maximal erreichbare Rechenleistung von 75 MFLOPS in 64-Bit-Arithmetik und arbeitet mit einer Taktfrequenz von 50 MHz. Dies entspricht einer Taktzeit von 20 ns. Die Zugriffszeit auf den lokalen Speicher beträgt 60 ns.

Das am Zentralinstitut für Angewandte Mathematik der Forschungszentrum Jülich GmbH installierte System PARAGON XP/S 10 verfügt über 140 Prozessoren mit einer maximal erreichbaren Rechenleistung von 10.5 GFLOPS. Unter dem Betriebssystem PARAGON OSF/1 der Open Software Foundation, Release 1.1, wurde auf diesem Rechner eine Bandbreite des Netzwerks von ca. 35 Megabytes/s und eine Latenz-Zeit (*Startup Time*) von ca. 90 μs für den Aufbau einer Netzwerkverbindung und weitere Vorbereitungen zum Nachrichtenaustausch gemessen. In dieser Version des Betriebssystems war der Kommunikationsprozessor nicht aktiviert. Der Einsatz des Kommunikationsprozessors unter OSF/1, Release 1.2, führte zu einer Erhöhung der real verfügbaren Bandbreite auf ca. 75 Megabytes/s und zu einer Reduzierung der Latenz-Zeit auf ca. 50 μs.

4.2 Parallele Algorithmen

Die Verfügbarkeit massiv-paralleler Systeme erfordert die Entwicklung neuer Strategien und Methoden zur Lösung mathematisch-technischer Probleme. Parallele Algorithmen werden einerseits hinsichtlich der Eignung für bestimmte Rechnerarchitekturen klassifiziert, andererseits ist eine Unterscheidung nach der inhärenten Parallelität des zu lösenden Problems möglich [39] [80] [82] [91]. Im folgenden wird kurz auf Merkmale und Leistungsaspekte paralleler Algorithmen eingegangen; anschließend werden Sprachkonzepte zur Formulierung paralleler Verfahren vorgestellt.

4.2.1 Charakterisierung

Werden in einem parallelen Algorithmus gleiche Operationen auf unterschiedlichen Daten ausgeführt, liegt *Datenparallelismus* vor. Die in dieser Arbeit entwickelten Parallelisierungsstrategien nutzen den Datenparallelismus der betrachteten iterativen Verfahren. Eine andere Art des Parallelismus ist der *Funktionsparallelismus*, bei dem den Prozessoren eines Parallelrechners zur Lösung eines Problems Teilprozesse mit unterschiedlicher Funktion zugeteilt werden.

Der Grad des Parallelismus eines Verfahrens wird durch die *Daten-* und *Modulgranularität* bestimmt. Die Datengranularität beschreibt die maximal mögliche Aufteilung eines Datensatzes in Segmente, die parallel bearbeitet werden können. Demgegenüber gibt die Modulgranularität den Anteil der Operationen eines Algorithmus an, die unabhängig auf verschiedenen Prozessoren ausgeführt werden

können. Daher ist die Modulgranularität ein Maß für die Häufigkeit der erforderlichen Synchronisation, die die korrekte Reihenfolge von Anweisungen sicherstellt.

Ein paralleler Algorithmus wird als *statisch* bezeichnet, wenn Prozeßaufteilung und notwendige Synchronisation vor der Ausführung festliegen. Die in dieser Arbeit entwickelten parallelen Methoden gehören dieser Klasse an. Bei *dynamischen* Algorithmen werden Aufteilung der Prozesse und Synchronisationsbedarf zur Laufzeit entschieden.

Für Datenaufteilung und Kommunikationsschema eines parallelen Algorithmus sind *Datenabhängigkeiten* und das Zugriffsmuster auf die verwendeten *Datenstrukturen* entscheidend. Ziel der Datenaufteilung und des Kommunikationsschemas ist der Lastausgleich der Prozessoren (*Load Balancing*). Bei den im Rahmen dieser Arbeit untersuchten Parallelisierungsstrategien werden Datenverteilung und Kommunikationsschema zur Laufzeit ermittelt. Nach Bestimmung der Datenverteilung, die durch Berücksichtigung der Operationen des Algorithmus für die gleichmäßige Verteilung der Rechenlast sorgt, wird das Kommunikationsschema durch Analyse der Datenabhängigkeiten festgelegt.

4.2.2 Leistungsaspekte

Ein Maß für die Güte eines parallelen Algorithmus ist der *Speedup*. Der Speedup ergibt sich aus dem Verhältnis der Ausführungszeit für das schnellste sequentielle Verfahren zur Lösung eines Problems und der Zeit, die ein paralleler Algorithmus zur Lösung desselben Problems benötigt. Die Anzahl der eingesetzten Prozessoren bildet eine obere Schranke für den Speedup. Als *Effizienz* einer parallelen Methode wird der Quotient aus Speedup und Prozessorzahl bezeichnet. Die Speedup-Definition aus dem Gesetz von Amdahl und der skalierte Speedup von Gustafson [72], die den sequentiellen Anteil eines parallelen Algorithmus berücksichtigen, werden in dieser Arbeit nicht verwendet.

Ein weiteres Kriterium zur Bewertung paralleler Algorithmen ist die Komplexität, die den Aufwand eines Verfahrens z. B. bezüglich Operationen, Rechenzeit, Speicherplatz oder Kommunikation in Abhängigkeit von der Problemgröße beschreibt. Auf die Komplexität von Algorithmen der linearen Algebra wird u. a. in [80] eingegangen.

4.2.3 Sprachkonzepte

Mit der Verbreitung paralleler Systeme gewinnt die Entwicklung von Programmiersprachen zur Formulierung paralleler Algorithmen an Bedeutung. Einige Aspekte imperativer und funktionaler Sprachen für Parallelrechner werden im folgenden erläutert.

Imperative Programmiersprachen

Zur Beschreibung paralleler Algorithmen in imperativen Programmiersprachen wurden einerseits traditionelle Sprachen, z. B. FORTRAN, um parallele Konstrukte erweitert [1] [64] [148], andererseits wurden parallele Sprachen wie CSP (*Communicating Sequential Processes*), OCCAM und Linda entwickelt [40] [78] [134]. Die Programmierung von Parallelrechnern in imperativen Sprachen ist aufwendig, da Datenverteilung, Austausch von Daten und Synchronisation vom Programmierer spezifiziert werden müssen. Ferner sind nur wenige Werkzeuge zur Unterstützung der Programmentwicklung verfügbar.

Funktionale Programmiersprachen

Im Gegensatz zu imperativen Sprachen basieren funktionale Programmiersprachen nicht auf der Verwendung von Variablen und der Zuweisung von Werten, das Grundelement dieser Sprachen ist die Funktion [62] [135]. Nicht der Lösungsweg eines Problems, sondern das Problem selbst wird in mathematischem Sinne beschrieben. Funktionale Sprachen sind seiteneffektfrei, d. h. die Auswertungsreihenfolge unabhängiger Teilausdrücke ist beliebig. Durch die funktionale Ablaufstruktur sind diese Sprachen implizit parallel; die Programmierung von Parallelrechnern ist auf hohem Abstraktionsniveau möglich. Datenverteilung, Austausch von Daten und Synchronisation sind Aufgaben des Compilers und des Laufzeitsystems.

Kapitel 5

Parallelisierungsstrategien für iterative Verfahren

In diesem Kapitel werden Parallelisierungsstrategien für iterative Verfahren mit großen dünnbesetzten Matrizen auf massiv-parallelen Rechnersystemen mit verteiltem Speicher beschrieben. Insbesondere werden effiziente Varianten der verteilten Matrix-Vektor-Multiplikation vorgestellt, da die Hauptarbeit in jedem Iterationsschritt der betrachteten iterativen Methoden gewöhnlich in der Berechnung dieser Operation besteht.

Parallelisiert man jeden Iterationsschritt eines iterativen Verfahrens auf einem Parallelrechner mit verteiltem Speicher, so müssen zunächst die Matrix-, Vektor- und skalaren Daten geeignet auf die einzelnen Prozessoren verteilt werden. Anschließend wird ein Kommunikationsschema ermittelt, das den erforderlichen Datenaustausch der Prozessoren beschreibt. Kriterien für Datenverteilung und Kommunikationsschema hinsichtlich der effizienten Parallelisierung einer iterativen Methode sind einerseits die gleichmäßige Aufteilung der Rechenlast und andererseits die Minimierung des Aufwands für Kommunikation. Die im Rahmen dieser Arbeit entwickelten Schemata für Datenverteilung und Kommunikation werden in einem Vorverarbeitungsschritt ermittelt; sie werden in jedem Iterationsschritt eines iterativen Verfahrens genutzt. Da zur Bestimmung dieser Schemata lediglich die symbolische Struktur der dünnbesetzten Matrix analysiert wird, können sie in Diskretisierungsverfahren darüber hinaus solange verwendet werden, bis sich das Diskretisierungsgitter ändert. In FE-Modellen z. B. können die Schemata in jedem Zeitschritt eines zeitabhängigen Problems oder in jedem iterativen Schritt eines nichtlinearen Problems, das durch Linearisierung gelöst wird, genutzt werden. Da in einigen technischen Anwendungen jedoch Änderungen des Diskretisierungsgitters auftreten [124], wird auch der Vorverarbeitungsschritt möglichst effizient durchgeführt.

Die im folgenden beschriebenen Methoden werden anhand der CRS-Speichertechnik für allgemeine dünnbesetzte Matrizen erläutert. Bei den zusätzlich betrachteten Formaten SICRS und ITPACK ist analog vorzugehen. Ferner können

das CRS-, das SICRS- und das ITPACK-Format bei den untersuchten Parallelisierungsstrategien ohne Kommunikation wechselseitig ineinander überführt werden. Darüber hinaus lassen sich die Prinzipien der Datenverteilung und des Kommunikationsschemas auf weitere Datenformate für dünnbesetzte Matrizen anwenden.

Zunächst wird ein kurzer Überblick über aktuelle Methoden zur Parallelisierung iterativer Verfahren mit dünnbesetzter Matrix gegeben und anschließend der Unterschied zu den Ansätzen dieser Arbeit herausgestellt. Danach werden Strategien zur Datenverteilung und unterschiedliche Kommunikationsschemata beschrieben. Schließlich wird auf die Integration einiger der entwickelten Methoden für dünnbesetzte Matrizen in eine funktionale Programmiersprache eingegangen.

5.1 Überblick über aktuelle Methoden

Im folgenden werden derzeitige Verfahren zur Parallelisierung iterativer Methoden mit dünnbesetzten Matrizen auf Parallelrechnern mit verteiltem Speicher beschrieben. Insbesondere wird auf die Datenverteilung und das Kommunikationsschema eingegangen.

Auf Mehrprozessorrechnern mit gemeinsamem Speicher entfällt das Problem der Datenverteilung. Bei iterativen Methoden mit dünnbesetzten Matrizen bearbeitet jeder Prozessor häufig Segmente der Matrix mit gleich vielen aufeinanderfolgenden Zeilen [126] oder mit gleich vielen Blöcken bei blockstrukturierten Matrizen [101]. Zugriffskonflikte auf den gemeinsamen Speicher bei der Matrix-Vektor-Multiplikation reduzieren die Effizienz der Verfahren [101] [126].

5.1.1 Zerlegung des Diskretisierungsgitters

Ein Ansatz zur Parallelisierung von Diskretisierungsverfahren, die gewöhnlich die Lösung von Gleichungssystemen oder Eigenwertproblemen erfordern, ist die Zerlegung des Diskretisierungsgitters. Algorithmen zur Partitionierung beliebiger Gitter sind in [75], [127] und [130] beschrieben. Probleme, die bei der Gitterzerlegung auftreten, sind zum einen der Aufwand für die Partitionierung, der häufig beträchtlich ist, zum anderen der Lastausgleich, wenn die Geometrie des Gitters eine Zerlegung bedingt, die nicht zur gleichmäßigen Verteilung der Rechenlast auf die einzelnen Prozessoren eines Parallelrechners führt. Ferner ist die Zerlegung in eine beliebige Anzahl von Teilgebieten für beliebige Prozessorzahlen bei unstrukturierten Gittern schwierig.

Die Zerlegung des Gitters in Teilgebiete führt zu einer Aufteilung der globalen Koeffizientenmatrix und der übrigen Daten. Das Kommunikationsschema ist durch den Austausch der Randwerte an den Schnittstellen der Teilgebiete gegeben. In [128] wird eine verteilte Datenstruktur vorgeschlagen, in der die Werte für

Randpunkte von den Werten für innere Punkte eines Teilgebiets getrennt sind. Die Berechnungen für die inneren Punkte erfordern keine Kommunikation.

5.1.2 Zerlegung der Matrix

Ein anderer Ansatz zur Parallelisierung iterativer Verfahren mit dünnbesetzter Koeffizientenmatrix ist die unmittelbare Aufteilung der Matrix-Daten auf die Prozessoren eines Rechnersystems. Ausgangspunkt ist die Besetzungsstruktur der Matrix. Häufig werden die Matrixelemente zeilen- bzw. spaltenweise oder in Blöcken auf die einzelnen Prozessoren verteilt. Diese Methode ist ein allgemeiner Ansatz, der auch für Probleme mit dünnbesetzten Matrizen geeignet ist, die nicht aus Diskretisierungsverfahren stammen.

In [57] werden verteilte Datenstrukturen, die auf dem CRS-Format basieren, für Operationen aus dem Bereich der linearen Algebra mit dünnbesetzten Matrizen vorgeschlagen. Insbesondere wird die Matrix-Vektor-Multiplikation betrachtet. Wird einem Prozessor die k-te Komponente des Vektors zugeteilt, so erhält dieser alle komprimierten Zeilen der Matrix $\boldsymbol{A}$, in denen $a_{ij} \neq 0$ mit $i = k$ oder $j = k$ auftreten. Bei dieser Verteilung werden viele Zeilen mehrfach gespeichert. Ferner ist der Lastausgleich der Prozessoren bezüglich Rechnung und Kommunikation für unstrukturierte Probleme schwierig.

Lewis et al. vergleichen in [94] und [95] verschiedene Datenverteilungen für die Matrix-Vektor-Multiplikation mit dünnbesetzter Matrix hinsichtlich der Effizienz dieser Operation auf Parallelrechnern mit verteiltem Speicher. Jedem Prozessor werden gleich viele Zeilen, gleich viele Spalten oder gleich viele Blöcke von Matrixelementen zugeteilt. Den untersuchten Blockverteilungen liegt eine zweidimensionale Zerlegung der Matrix zugrunde. Diese Verteilungen zeigen das günstigste Kommunikationsverhalten.

In [35] und [36] werden kartesische Verteilungen dünnbesetzter Matrizen für die iterative Lösung unsymmetrischer Gleichungssysteme $\boldsymbol{A}x = b$ auf Transputer-Systemen untersucht. Insbesondere werden quadratische Verteilungen betrachtet. Sind $p = q \cdot q$ Prozessoren durch ein Netzwerk mit quadratischer Gitter-Topologie verbunden, so werden die Nichtnull-Element $a_{i+1,j+1}$, $0 \leq i, j < n$, den Prozessoren $(i \,\mathbf{mod}\, q,\ j \,\mathbf{mod}\, q)$ zugeteilt; die Vektorkomponenten x_{i+1} erhalten die Prozessoren $((i \,\mathbf{div}\, q) \,\mathbf{mod}\, q,\ i \,\mathbf{mod}\, q)$. **mod** bezeichnet die Modulo-Operation und **div** die ganzzahlige Division ohne Rest. Pro Prozessor werden die Matrix-Daten in dem CRS- bzw. CCS-Format ähnlichen Strukturen abgelegt. Bei der Berechnung der im Lösungsverfahren auftretenden Matrix-Vektor-Produkte müssen bei dieser Verteilung sowohl Komponenten des Vektors als auch des Ergebnisvektors dieser Operation ausgetauscht werden.

Die Untersuchungen in [147] zur effizienten Berechnung der Matrix-Vektor-Multiplikation basieren auf der zeilenweisen Verteilung der Matrix-Daten. Zur Speicherung der Nichtnull-Elemente wird das CRS-Schema verwendet. Zunächst erhält jeder Prozessor gleich viele, aufeinanderfolgende Zeilen der Matrix. An-

schließend wird in einem Vorverarbeitungsschritt das Kommunikationsschema für diese Verteilung ermittelt. Um den Kommunikationsaufwand zu vermindern, werden Heuristiken für die Neuverteilung der Matrix zur Laufzeit vorgeschlagen. Hierzu werden iterativ die äußeren Zeilen der Zeilen-Blöcke, die den Prozessoren zugeordnet sind, zwischen je zwei Prozessoren ausgetauscht und geprüft, ob das Kommunikationsverhalten günstiger wird.

5.1.3 Software-Pakete

In jüngster Zeit werden Methoden zur parallelen Bearbeitung von Problemen mit dünnbesetzten Matrizen auch in Software-Paketen berücksichtigt. Diese Pakete bieten Routinen zur parallelen Lösung von Diskretisierungsproblemen mit unstrukturierten Gittern an; insbesondere sind Methoden zur parallelen iterativen Lösung von linearen Gleichungssystemen und Eigenwertproblemen integriert.

Die PARTI-Routinen (*Parallel Automated Runtime Toolkit at ICASE*) unterstützen die effiziente Programmierung irregulärer Probleme mit unstrukturierten Gittern [47] [48]. Die Routinen werden u.a. in CFD-Programmen (*Computational Fluid Dynamics*) und Verfahren zur Lösung von linearen Gleichungssystemen mit dünnbesetzter Koeffizientenmatrix eingesetzt. Bei Verwendung der PARTI-Routinen zur Parallelisierung eines sequentiellen Programms ist es Aufgabe des Benutzers, eine geeignete Datenverteilung für ein spezielles Problem vorzugeben. Dazu kann z.B. eine günstige Zerlegung des Diskretisierungsgitters betrachtet werden. Zur Verteilung der Datenstrukturen des sequentiellen Programms und zum Zugriff auf die verteilten Daten sind komfortable Routinen vorhanden. Diese Routinen leisten u.a. die Umrechnung von globalen Indizes im sequentiellen Verfahren auf lokale Indizes der verteilten Datenstrukturen und unterstützen den Austausch von Daten zwischen den einzelnen Prozessoren. Die Informationen über Verteilung und Adressierung der Daten sind in umfangreichen Tabellen abgelegt, die zusätzlichen Speicherplatz erfordern.

Software-Bibliotheken, die parallelisierte iterative Verfahren enthalten, werden in [86], [128] und [129] vorgestellt. Weitere Software-Pakete sind in [46], [71] und [131] beschrieben. Darüber hinaus existieren Initiativen zur Parallelisierung sequentieller Software-Bibliotheken für iterative Verfahren [88] [109] [119]. In den meisten Bibliotheksverfahren ist jedoch die Berechnung der Matrix-Vektor-Produkte nicht spezifiziert, sondern lediglich als Aufruf eines Unterprogramms, das der Benutzer implementieren muß, vorhanden. Der Grund ist die große Anzahl unterschiedlicher Speichertechniken und Modifikationen dieser Schemata für dünnbesetzte Matrizen. Ein Standard existiert bis heute nicht. Einige Bibliotheken enthalten Implementierungen für parallele Matrix-Vektor-Produkte in weit verbreiteten Formaten wie dem CRS- und dem ITPACK-Schema.

5.1.4 Parallelisierende Compiler

Die automatische Parallelisierung sequentieller Programme, die irreguläre Strukturen wie dünnbesetzte Matrizen enthalten, durch parallelisierende Compiler kann durch Spracherweiterungen imperativer Programmiersprachen unterstützt werden. Insbesondere sind Erweiterungen zur Datenverteilung für irreguläre Strukturen und zum Zugriff auf die verteilten Daten erforderlich. Die PARTI-Routinen wurden u. a. für die Verwendung in parallelisierenden Compilern auf Parallelrechnern mit verteiltem Speicher entwickelt [48] [144]. Ferner existieren Initiativen zur Integration derartiger Spracherweiterungen in HPF (*High Performance FORTRAN*), FORTRAN D und Vienna FORTRAN [1] [64] [148].

5.2 Entwickelte Verfahren

Bei den im Rahmen dieser Arbeit entwickelten parallelen Verfahren zur Lösung von Gleichungssystemen und Eigenwertproblemen werden Datenverteilung und Kommunikationsschema in einem Vorverarbeitungsschritt ermittelt. Ausgangspunkt ist die Analyse der Besetzungsstruktur der Matrix. Die Methoden sind für allgemeine dünnbesetzte Matrizen geeignet.

Die Datenverteilung geschieht automatisch nach dem Kriterium aus 5.3 und berücksichtigt neben der Berechnung von Matrix-Vektor-Produkten auch alle übrigen Vektor-Operationen des iterativen Verfahrens. Die Verteilung braucht nicht abhängig von der spezifischen Struktur der Matrix oder des Diskretisierungsgitters vorgegeben zu werden. Jedem Prozessor werden aufeinanderfolgende, vollständige Zeilen der Matrix zugeteilt, jedoch können die Prozessoren eine unterschiedliche Anzahl von Zeilen erhalten.

Zur Reduzierung von Wartezeiten werden im Kommunikationsschema Rechnung und Datenaustausch überlappend durchgeführt, d. h. während sich angeforderte Daten auf dem Verbindungsnetzwerk der Prozessoren befinden, wird mit lokalen Daten gerechnet. Zu diesem Zweck wird eine verteilte Datenstruktur verwendet, in der die Matrix-Daten abhängig vom Kommunikationsschema in Blöcken angeordnet werden (s. 5.4).

Darüber hinaus wird die Neuverteilung der Matrix-Daten zur Laufzeit unter Berücksichtigung des Kommunikationsschemas untersucht. Dabei werden Segmente von Zeilen der Matrix ausgetauscht.

Ferner wird ein Kommunikationsschema für unabhängige Matrix-Vektor-Produkte der Form $\boldsymbol{A}s$ und $\boldsymbol{A}^T t$ betrachtet. Operationen dieser Art treten z. B. im QMR-Verfahren aus 2.3.1 und im BiCG-Algorithmus [92] auf. Das entwickelte Schema reduziert den Kommunikationsaufwand durch Kopplung des Datenaustauschs der beiden Operationen.

Die entwickelten Methoden für parallele Operationen mit dünnbesetzten Matrizen eignen sich für den Einsatz in parallelen Software-Paketen. Darüber hinaus

ist die Verwendung als Spracherweiterungen in imperativen Programmiersprachen für Parallelrechner möglich.

In Zusammenarbeit mit dem Lehrstuhl für Informatik II der RWTH Aachen wurden einige Methoden zur parallelen Matrix-Vektor-Multiplikation als algorithmische Skelette in eine funktionale Programmiersprache integriert [135]. Ein Vorteil der funktionalen Sprachen gegenüber den imperativen Sprachen ist die implizite Parallelität (s. auch 4.2.3).

Untersuchungen hinsichtlich der Effizienz der im Rahmen dieser Arbeit entwickelten iterativen Verfahren auf Parallelrechnern mit verteiltem Speicher sind in [19]–[27], [29], [38] und [122] beschrieben.

5.3 Datenverteilung

Zur Parallelisierung eines iterativen Verfahrens auf einem Parallelrechner mit verteiltem Speicher müssen zunächst alle Daten des Algorithmus geeignet auf die einzelnen Prozessoren verteilt werden. Neben der Verteilung der skalaren und Vektor-Daten ist die Verteilung der Elemente der dünnbesetzten Matrix entscheidend für die Effizienz der Methode. Bei den im Rahmen dieser Arbeit betrachteten Verteilungen werden jedem Prozessor aufeinanderfolgende, vollständige Zeilen bzw. Spalten der Matrix zugeordnet. Die Segmentierung der Vektor-Daten entspricht der Zeilen-Aufteilung der Matrix. Besitzt ein Prozessor die i-te Zeile der Matrix, so erhält er auch die i-te Komponente aller Vektoren. Skalare Daten werden jedem Prozessor zugeteilt, so daß skalare Berechnungen auf jedem Prozessor lokal durchgeführt werden können.

Die Verteilung der Elemente einer dünnbesetzten Matrix kann nach unterschiedlichen Kriterien erfolgen. Jeder Prozessor kann z. B. ungefähr gleich viele Zeilen oder annähernd gleich viele Nichtnull-Elemente der Matrix erhalten. Ziel des Kriteriums, das hier betrachtet wird, ist die gleichmäßige Aufteilung sowohl der Matrix-Vektor- als auch aller übrigen Vektor-Operationen eines iterativen Verfahrens auf die einzelnen Prozessoren. Im folgenden wird die Aufteilung einer Matrix der Ordnung n mit e Nichtnull-Elementen auf p Prozessoren nach diesem Kriterium beschrieben.

Beginnend bei Prozessor 0 erhalten die Prozessoren k, $k = 0, 1, \ldots, p-1$, mit aufsteigender Prozessornummer n_k aufeinanderfolgende Zeilen mit insgesamt e_k Nichtnull-Elementen, so daß

$$n = \sum_{k=0}^{p-1} n_k \quad \text{und} \quad e = \sum_{k=0}^{p-1} e_k$$

gilt. Die Nummer der ersten Zeile g_k der Matrix, die Prozessor k zugewiesen wird, ist durch

$$g_k = 1 + \sum_{i=0}^{k-1} n_i$$

gegeben. Bezeichnet ferner z_i die Anzahl der Nichtnull-Elemente der Zeile i mit $e = \sum_{i=1}^{n} z_i$, so wird die Anzahl der Nichtnull-Elemente e_k des Prozessors k wie folgt berechnet:

$$e_k(g_k, n_k) = \sum_{i=g_k}^{g_k+n_k-1} z_i.$$

Zur gleichmäßigen Verteilung der Operationen eines iterativen Verfahrens müssen die Werte n_k geeignet bestimmt werden. Die Anzahl der Operationen eines Matrix-Vektor-Produkts ist proportional zur Anzahl e der Nichtnull-Elemente der Matrix; alle übrigen Vektor-Operationen sind proportional zur Zeilenanzahl n. Werden pro Iterationsschritt s Matrix-Vektor-Produkte berechnet, so beträgt der Gesamtaufwand $c_1 s\, e + c_2 n + c_3$ mit den Zeitkonstanten c_1, c_2 und c_3. Die Konstante c_3 beschreibt den Aufwand für skalare Operationen und wird in den folgenden Überlegungen, die sich auf große Matrizen beziehen, vernachlässigt. Der Anteil der Operationen des Prozessors k am Gesamtaufwand eines Iterationsschritts ist durch

$$\frac{s\, e_k + \xi n_k}{s\, e + \xi n} \quad \text{mit} \quad \xi \in \mathbb{R} \tag{5.1}$$

gegeben. Der Parameter ξ ist einerseits ein Maß für die Anzahl der Vektor-Vektor-Operationen, die zusätzlich zu den Matrix-Vektor-Multiplikationen in jedem Iterationsschritt durchgeführt werden. Andererseits berücksichtigt ξ die Ausführungszeiten von arithmetischen, logischen und Speicher-Operationen auf dem verwendeten Prozessor. Aus diesem Grund ist ξ sowohl von dem Algorithmus des iterativen Verfahrens als auch von der Architektur des verwendeten Prozessors abhängig. Die Rechenlast ist gleichmäßig auf die einzelnen Prozessoren verteilt, wenn jeder Prozessor den p-ten Teil aller Operationen durchführt. Daher wird die Verteilung der Zeilen der Matrix durch das Kriterium

$$n_k = \begin{cases} \displaystyle\min_{1 \le t \le n-g_k+1} \left\{ t \,\middle|\, \frac{s\, e_k(t) + \xi t}{s\, e + \xi n} \ge \frac{1}{p} \right\} & \text{für} \quad k = 0, 1, \ldots, q \\ \displaystyle n - \sum_{i=0}^{q} n_i & \text{für} \quad k = q+1 \\ 0 & \text{für} \quad k = q+2, \ldots, p-1 \end{cases} \tag{5.2}$$

festgelegt. Die Ungleichung in (5.2) kann für die Prozessoren 0 bis q erfüllt werden. Sind weitere Zeilen vorhanden, erhält diese Prozessor $q+1$. Die Prozessoren $q+2$ bis $p-1$ erhalten keine Daten. Der letztere Fall tritt auf, wenn kleine Probleme mit vielen Prozessoren berechnet werden. Bei großen dünnbesetzten Matrizen gilt gewöhnlich $q = p-1$ oder $q+1 = p-1$.

Der Anteil der Matrix-Vektor-Produkte am Gesamtaufwand eines Iterationsschritts ergibt sich mit (5.1) zu

$$a_{\mathrm{MVP}} \approx \frac{s\, e}{s\, e + \xi n} = \frac{1}{1 + \xi/(s\, z_{\mathrm{mean}})} \tag{5.3}$$

mit $z_{\mathrm{mean}} = e/n$ als der mittleren Anzahl von Nichtnull-Elementen pro Zeile der Matrix. Darüber hinaus ist durch (5.3) ein Mittel gegeben, ξ für einen bestimmten Algorithmus und einen bestimmten Prozessortyp zu messen. Wird a_{MVP} durch Zeitmessungen ermittelt, so kann ξ durch

$$\xi \approx (\frac{1}{a_{\mathrm{MVP}}} - 1) s\, z_{\mathrm{mean}}$$

näherungsweise bestimmt werden. Auf den INTEL-Prozessoren i860 XR des iPSC/860 und i860 XP des PARAGON ergeben Zeitmessungen die in Tabelle 5.1 zusammengefaßten Näherungen von ξ für die betrachteten iterativen Verfahren.

Iterative Methode	s	ξ
CG	1	≈ 8
CG mit pol. Vorkond.	$m+1$	$\approx \xi_{\mathrm{CG}}$
QMR	2	≈ 16
TFQMR	2	≈ 13
Lanczos-Tridiagonal.	1	≈ 2

Tabelle 5.1: Näherungen des Parameters der Datenverteilung für die Prozessoren i860 XR und i860 XP

Ein hoher Wert von ξ weist auf eine große Anzahl von Vektor-Operationen hin, die pro Iterationsschritt zusätzlich zu den Operationen für die Matrix-Vektor-Produkte durchgeführt werden. Da im CG-Verfahren mit polynomialer Vorkonditionierung bei Verwendung eines Polynoms vom Grad m gegenüber der unvorkonditionierten Methode im wesentlichen m zusätzliche Matrix-Vektor-Produkte berechnet werden, ist eine Näherung für ξ durch ξ_{CG} mit ξ_{CG} als dem Wert für die CG-Iteration gegeben. Je dominanter der Aufwand für die Berechnung des Matrix-Vektor-Produkts im CG-Verfahren ist, desto höher ist die Güte dieser Näherung.

Schwankungen in den Werten für ξ bei verschiedenen dünnbesetzten Matrizen können durch ein unterschiedliches Speicherzugriffsverhalten auftreten. Bei kleinen Matrizen, die z. B. vollständig in den Prozessor-Cache passen, kann ein anderes Speicherzugriffsverhalten als bei großen Matrizen vorliegen. Ebenso führen regelmäßig strukturierte Matrizen wie Diagonal- oder Tridiagonalmatrizen zu einem günstigeren Speicherzugriffsverhalten als unregelmäßig besetzte Matrizen. Bei den im Rahmen dieser Arbeit betrachteten großen unregelmäßig besetzten Matrizen wurden jedoch lediglich geringfügige Schwankungen in den Werten von ξ bei gleichem Prozessortyp und Algorithmus für unterschiedliche Matrizen beobachtet.

Durch spezielle Werte des Parameters ξ können die Daten gemäß den beiden anderen oben erwähnten Kriterien verteilt werden. Bei $\xi = 0$ werden jedem Prozessor annähernd gleich viele Nichtnull-Elemente zugeteilt. Im Grenzfall $\xi \to \infty$ erhält jeder Prozessor ungefähr gleich viele Zeilen.

Im folgenden wird die Datenverteilung gemäß Kriterium (5.2) anhand der Matrix (5.4) veranschaulicht. Die Matrix ist in Abbildung 5.1 im CRS-Schema gespeichert. Die Elemente des Felds `col_ind` werden für unterschiedliche Werte des Parameters ξ auf vier Prozessoren verteilt. Die Verteilung des Felds `value` erfolgt analog. Durch Analyse des Felds `row_ptr` kann jeder Prozessor parallel ermitteln, welcher Zeilen-Bereich welchen Prozessoren zugeordnet wird, da alle Informationen zur Auswertung von (5.2) im Feld `row_ptr` abgelegt sind.

$$\boldsymbol{A} = \begin{pmatrix} \mathbf{20} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{30} & 0 & 0 & \mathbf{9} & 0 & 0 & 0 \\ 0 & \mathbf{7} & \mathbf{40} & \mathbf{10} & 0 & 0 & 0 & 0 \\ \mathbf{14} & 0 & \mathbf{10} & \mathbf{50} & \mathbf{11} & 0 & \mathbf{13} & \mathbf{18} \\ 0 & 0 & 0 & \mathbf{11} & \mathbf{60} & 0 & \mathbf{19} & 0 \\ 0 & 0 & 0 & \mathbf{14} & 0 & \mathbf{70} & \mathbf{16} & 0 \\ \mathbf{12} & 0 & \mathbf{17} & 0 & 0 & \mathbf{15} & \mathbf{80} & 0 \\ 0 & 0 & 0 & \mathbf{18} & 0 & 0 & 0 & \mathbf{90} \end{pmatrix} \tag{5.4}$$

`value:`

20	9	30	40	7	10	10	50	14	13	11	18	11	60	19
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

16	14	70	17	12	80	15	18	90
16	17	18	19	20	21	22	23	24

`col_ind:`

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

7	4	6	3	1	7	6	4	8
16	17	18	19	20	21	22	23	24

`row_ptr:`

1	2	4	7	13	16	19	23	25

Abbildung 5.1: CRS-Speicherschema

In Abbildung 5.2 ist der Grenzfall $\xi \to \infty$ dargestellt. Jeder Prozessor erhält gleich viele Zeilen.

Abbildung 5.3 zeigt die Verteilung für $\xi = 0$. Jedem Prozessor werden gleich viele Nichtnull-Elemente zugewiesen.

In Abbildung 5.4 ist das Feld `col_ind` für $\xi = 16$ und $s = 2$ auf vier Prozessoren verteilt.

Für die drei Werte von ξ ergeben sich in den Abbildungen 5.2 bis 5.4 drei unterschiedliche Verteilungen der Zeilen der Matrix.

col_ind
Prozessor 0: | 1 ‖ 5 | 2 |

col_ind
Prozessor 1: | 3 | 2 | 4 ‖ 3 | 4 | 1 | 7 | 5 | 8 |

col_ind
Prozessor 2: | 4 | 5 | 7 ‖ 7 | 4 | 6 |

col_ind
Prozessor 3: | 3 | 1 | 7 | 6 ‖ 4 | 8 |

Abbildung 5.2: Datenverteilung nach dem Kriterium „Gleich viele Zeilen“ ($\xi \to \infty$)

col_ind
Prozessor 0: | 1 ‖ 5 | 2 ‖ 3 | 2 | 4 |

col_ind
Prozessor 1: | 3 | 4 | 1 | 7 | 5 | 8 |

col_ind
Prozessor 2: | 4 | 5 | 7 ‖ 7 | 4 | 6 |

col_ind
Prozessor 3: | 3 | 1 | 7 | 6 ‖ 4 | 8 |

Abbildung 5.3: Datenverteilung nach dem Kriterium „Gleich viele Nicht-null-Elemente“ ($\xi = 0$)

col_ind
Prozessor 0: | 1 ‖ 5 | 2 ‖ 3 | 2 | 4 |

col_ind
Prozessor 1: | 3 | 4 | 1 | 7 | 5 | 8 ‖ 4 | 5 | 7 |

col_ind
Prozessor 2: | 7 | 4 | 6 ‖ 3 | 1 | 7 | 6 |

col_ind
Prozessor 3: | 4 | 8 |

Abbildung 5.4: Datenverteilung für $\xi = 16$ und $s = 2$

5.4 Kommunikationsschemata

Auf einem Parallelrechner mit verteiltem Speicher erfordert die Berechnung der Matrix-Vektor-Produkte Kommunikation, da die Prozessoren lediglich Segmente der Vektoren besitzen. Im folgenden werden Kommunikationsschemata für die zeilen- und spaltenweise Matrix-Vektor-Multiplikation beschrieben, denen die oben betrachtete Datenverteilung zugrundeliegt.

Neben der Matrix-Vektor-Multiplikation werden in jedem Iterationsschritt einer iterativen Methode skalare Operationen, Vektoradditionen und Skalarprodukte berechnet. Skalare Berechnungen werden auf jedem Prozessor lokal durchgeführt; Linearkombinationen von Vektoren können bei der betrachteten Datenverteilung ohne Datenaustausch ermittelt werden. Zur Berechnung von Skalarprodukten hingegen ist globale Synchronisation erforderlich. Jeder Knoten k berechnet zunächst den lokalen Anteil l_k des Skalarprodukts $x^T y$

$$l_k = \sum_{i=g_k}^{g_k+n_k-1} x_i y_i.$$

Anschließend werden die lokalen Werte aller Prozessoren summiert:

$$x^T y = \sum_{k=0}^{p-1} l_k.$$

Bei einer hohen Prozessorzahl ist die Synchronisation aller Prozessoren aufwendig. Die in dieser Arbeit verfolgte Strategie zur Reduzierung des Aufwands für globale Synchronisation ist die Minimierung der Synchronisationspunkte. Zu diesem Zweck wurden in Kapitel 2 modifizierte Algorithmen des CG-Verfahrens und der Lanczos-Tridiagonalisierung eingeführt [14] [49] [87]. In [136] sind ähnliche Ansätze für iterative Lösungsverfahren unsymmetrischer Gleichungssysteme enthalten. Ferner kann die Gesamtzahl der Synchronisationspunkte des CG-Verfahrens durch Vorkonditionierung reduziert werden. Gleiches gilt für die Verwendung einer geeigneten Vorkonditionierung für Methoden zur Lösung unsymmetrischer Gleichungssysteme.

5.4.1 Analyse der Indizes

Durch Analyse der Zeilen- bzw. Spaltenindizes der Nichtnull-Elemente der Matrix kann ermittelt werden, welche Daten mit welchen Prozessoren zur Berechnung der Matrix-Vektor-Produkte ausgetauscht werden müssen. Die Indizes der Nichtnull-Elemente sind durch die symbolische Struktur der Matrix bestimmt; die Werte der Elemente sind zur Ermittlung des Kommunikationsschemas unerheblich. Die Indizes der lokalen Zeilen bzw. Spalten werden parallel auf allen Prozessoren analysiert; dies soll möglichst effizient durchgeführt werden. Im folgenden wird die Analyse der Zeilenindizes für die zeilenweise Matrix-Vektor-Multiplikation ausgehend von der Datenverteilung aus Abbildung 5.4 beschrieben.

Die Kommunikation für diese Operation geschieht über die Komponenten des Vektors des Matrix-Vektor-Produkts. Bei der spaltenweisen Matrix-Vektor-Multiplikation wird die gleiche Analyse durchgeführt. Lediglich die Ergebnisse sind auf andere Weise zu interpretieren, da hier Komponenten des Ergebnisvektors des Matrix-Vektor-Produkts ausgetauscht werden.

Bei der zeilenweisen Matrix-Vektor-Multiplikation $y = \boldsymbol{A}x$ führt die Operation des Prozessors k

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = g_k, \ldots, g_k + n_k - 1, \tag{5.5}$$

zu einem Zugriff auf Komponenten des Vektors x, die anderen Prozessoren zugeteilt sind. Bei einer dünnbesetzten Matrix $\boldsymbol{A}$ brauchen bei dieser Operation lediglich die Nichtnull-Elemente einer Zeile berücksichtigt zu werden. Die Spaltenindizes der Nichtnull-Elemente geben an, auf welche Komponenten des Vektors x zugegriffen wird (s. auch Abbildung 3.18). Ist auf jedem Prozessor die Zeilen-Verteilung der Matrix bekannt, kann ermittelt werden, welchem Prozessor die jeweilige Komponente zugeordnet ist.

Um die Analyse der Spaltenindizes zu erleichtern, werden zunächst alle Spaltenindizes einer Zeile im Speicherschema der Größe nach in aufsteigender Reihenfolge geordnet. Entsprechend der Reihenfolge der Spaltenindizes werden die Werte der Nichtnull-Elemente sortiert. Bei der betrachteten Verteilung der Zeilen der Matrix sind dann pro Zeile die Elemente, die zu einem Zugriff auf Vektorkomponenten eines bestimmten Prozessors führen, hintereinander abgelegt. Zusätzlich kann die Umordnung das Speicherzugriffsverhalten einer Implementierung der Matrix-Vektor-Multiplikation auf einem Rechner günstig beeinflussen, da pro Zeile bei der Operation (5.5) auf Komponenten von x mit aufsteigendem Index zugegriffen wird. Abbildung 5.5 zeigt das Speicherschema nach der Umordnung ausgehend von der Darstellung in Abbildung 5.4. Zusätzlich ist die Verteilung des Vektors x auf die Prozessoren angegeben. $x^{[k]}$ bezeichnet das Segment des Vektors x, das Prozessor k zugeteilt ist.

Anschließend wird auf jedem Prozessor durch Analyse der Spaltenindizes jeder Zeile ermittelt, welche Komponenten des Vektors von anderen Prozessoren angefordert werden müssen. Zur Analyse und Speicherung dieser Information wird die Datenstruktur aus Abbildung 5.6 verwendet.

$p_{\mathrm{from},k}$ bezeichnet die Anzahl der Prozessoren, von denen auf Prozessor k Daten empfangen werden; $n_{\max}$ ist die maximale Anzahl der Zeilen, die einem der Prozessoren h, $h = 0, \ldots, p-1$, zugeordnet sind. In einer Spalte des Felds `from_proc` sind die Indizes der Vektorkomponenten eines bestimmten Prozessors, die von Prozessor k benötigt werden, abgelegt. Das Feld `proc_id` enthält die jeweilige Prozessornummer und das Feld `num_ind` die Anzahl der Indizes. Zur Analyse der Spaltenindizes wird `from_proc` mit einem ungültigen Wert vorbesetzt. Während der Analyse werden die Indizes, die zu einem Zugriff auf die Vektorkomponenten anderer Prozessoren führen, in den entsprechenden Spalten

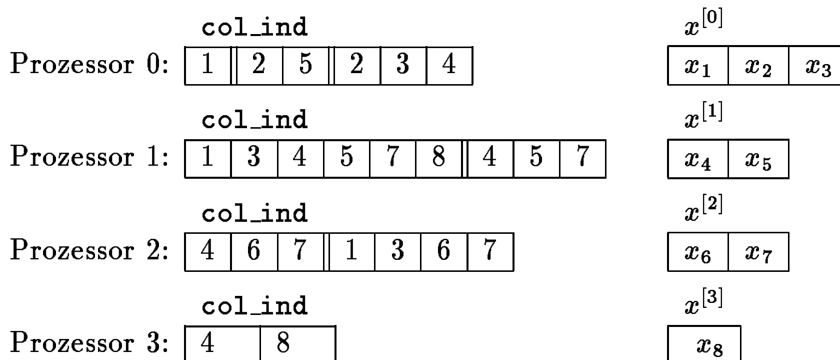


Abbildung 5.5: Ordnen der Spaltenindizes

RECEIVE = **record**
  `from_proc` : **array** $[1..n_{\max}, 1..p_{\text{from},k}]$ **of** INTEGER
  `proc_id` : **array** $[1..p_{\text{from},k}]$ **of** INTEGER
  `num_ind` : **array** $[1..p_{\text{from},k}]$ **of** INTEGER
end record

Abbildung 5.6: Datenstruktur zur Analyse der Spaltenindizes

eingetragen. Führt ein Spaltenindex j auf Prozessor k zu einem Zugriff auf die Vektorkomponente j von Prozessor m, so wird j in der zugehörigen Spalte von `from_proc` an Position $j{-}g_m{+}1$ gespeichert. Nach der Analyse werden alle Indizes der Vektorkomponenten, die benötigt werden, zum jeweiligen Spaltenbeginn hin verschoben und die Anzahl der Indizes pro Spalte ermittelt. Damit ist festgelegt, welche Daten auf jedem Prozessor von anderen Prozessoren empfangen werden. In Abbildung 5.7 ist die Analyse der Spaltenindizes für die Daten von Prozessor 1 aus Abbildung 5.5 dargestellt. „*“ bezeichnet einen ungültigen Wert. Auf Prozessor 1 werden von Prozessor 0 die Vektorkomponenten 1 und 3, von Prozessor 2 die Komponente 7 und von Prozessor 3 die Komponente 8 benötigt.

Nachdem auf jedem Prozessor bekannt ist, welche Daten von anderen Prozessoren benötigt werden, wird diese Information an die Prozessoren, auf denen die benötigten Daten lokal vorhanden sind, weitergegeben. Anschließend liegt auf jedem Prozessor fest, welche lokalen Daten an andere Prozessoren gesendet werden müssen. Die Indizes dieser lokalen Vektorkomponenten werden in der Datenstruktur aus Abbildung 5.8 abgelegt.

$p_{\text{to},k}$ ist die Anzahl der Prozessoren, an die von Prozessor k Daten gesendet werden. Im Feld `to_proc` sind die Indizes der Vektorkomponenten, die übermittelt werden, analog zur Datenstruktur aus Abbildung 5.6 gespeichert. Abbildung 5.9 zeigt die Speicherung dieser Indizes für Prozessor 1 aus Abbildung 5.5. Die Vektorkomponenten 4 und 5 von Prozessor 1 müssen an Prozessor 0 gesendet werden, die Komponente 4 wird von den Prozessoren 2 und 3 benötigt.

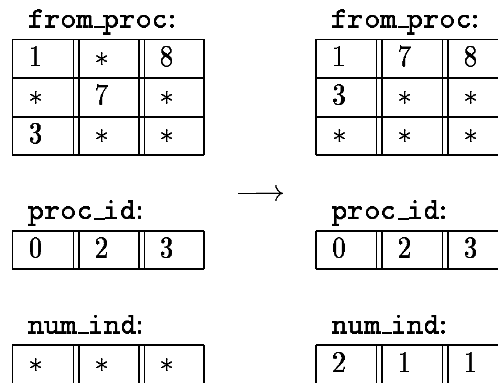


Abbildung 5.7: Analyse der Spaltenindizes, Prozessor 1

SEND = **record**
    to_proc : **array** $[1..n_{\max}, 1..p_{\mathrm{to},k}]$ **of** INTEGER
    proc_id : **array** $[1..p_{\mathrm{to},k}]$ **of** INTEGER
    num_ind : **array** $[1..p_{\mathrm{to},k}]$ **of** INTEGER
end record

Abbildung 5.8: Datenstruktur zur Speicherung der Indizes der Komponenten, die gesendet werden

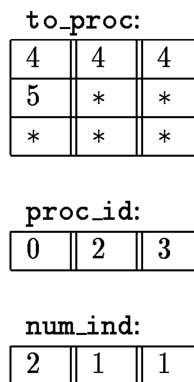


Abbildung 5.9: Speicherung der Indizes der Komponenten, die gesendet werden, Prozessor 1

5.4.2 Zeilenweise Matrix-Vektor-Multiplikation

Zur effizienten Durchführung der Matrix-Vektor-Multiplikation auf einem Parallelrechner mit verteiltem Speicher werden die Matrix-Felder umgeordnet. Ziel der Umordnung ist die Reduzierung von Wartezeiten durch Überlappung von lokalen Berechnungen und Transferzeiten für erforderliche nicht-lokale Daten. Im folgenden wird zunächst ein Schema zur Umordnung der Matrix-Daten in Blöcke für die zeilenweise Matrix-Vektor-Multiplikation vorgestellt. Anschließend wird die Neuverteilung dieser Blöcke auf die Prozessoren des Rechnersystems zur Laufzeit beschrieben. Beide Umordnungen führen zu einem unterschiedlichen Ablauf und Kommunikationsschema der Matrix-Vektor-Multiplikation.

Umordnung der Matrix-Daten in Blöcke

Auf Prozessor k werden nach der Analyse der Spaltenindizes die Daten im Speicherschema der dünnbesetzten Matrix derart sortiert, daß die Daten, die zu einem Zugriff auf Vektorkomponenten von Prozessor h führen, hintereinander abgelegt sind. Die Elemente dieses Blocks h sind zeilenweise und pro Zeile nach aufsteigendem Spaltenindex geordnet. Block k ist der erste Block im Speicherschema von Prozessor k und enthält die Daten, die zu einem Zugriff auf lokale Vektorkomponenten führen.

Abbildung 5.10 zeigt das Prinzip der Umordnung ausgehend von Abbildung 5.5 für das Spaltenindex-Feld von Prozessor 1. Die Indizes, die zu einem Zugriff auf nicht-lokale Daten führen, sind fett gedruckt. Unter den Blöcken ist die Blocknummer angegeben.

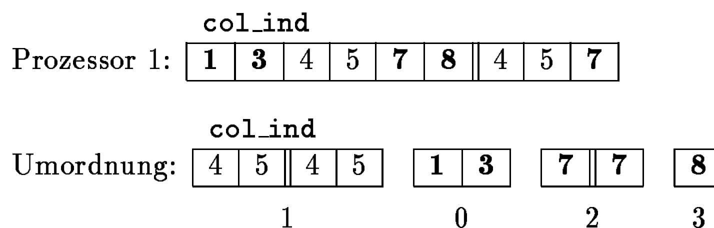


Abbildung 5.10: Umordnung in Blöcke

Die Elemente von Block 1 führen bei der Matrix-Vektor-Multiplikation zu einem Zugriff auf lokale Vektorkomponenten, während die Daten der Blöcke 0, 2 und 3 einen Zugriff auf Vektorkomponenten der Prozessoren 0, 2 und 3 in dieser Reihenfolge erfordern.

In Abbildung 5.11 ist die Datenstruktur zur Speicherung der blockweise sortierten Matrix-Daten von Prozessor k, $k = 0, \ldots, p-1$, ausgehend vom CRS-Schema dargestellt.

In den Feldern `value` und `col_ind` sind die Matrix-Daten von Prozessor k in der Reihenfolge der Blöcke abgelegt. b_k bezeichnet die Anzahl der Blöcke und

```
MATRIX = record
      value     : array [1..e_k] of REAL
      col_ind   : array [1..e_k] of INTEGER
      row_ptr   : array [1..n_k+1, 1..b_k] of INTEGER
      block_id  : array [1..b_k] of INTEGER
end record
```

Abbildung 5.11: Verteilte Datenstruktur bei blockweiser Sortierung

entspricht $p_{\mathrm{from},k} + 1$. Die Elemente der Spalten des Felds `row_ptr` verweisen pro Block auf die Position des Beginns einer Zeile in `value` und `col_ind`. Im Feld `block_id` ist die Nummer der Blöcke gespeichert. Dabei gilt `block_id`$(1) = k$ und `block_id`$(i) =$ `proc_id`$(i-1)$, $i = 2, \ldots, b_k$ mit `proc_id` aus der Datenstruktur RECEIVE in Abbildung 5.6. Enthält eine Zeile i eines Blocks h keine Elemente, so wird `row_ptr`$(i+1, h) =$ `row_ptr`(i, h) gesetzt. Da bei der blockweisen Matrix-Vektor-Multiplikation pro Block die gleiche Zeilen-Adressierung wie bei der Matrix-Vektor-Multiplikation im CRS-Schema aus Abbildung 3.18 verwendet wird, werden dann für Zeile i keine Operationen durchgeführt; die Berechnung wird mit den Elementen der Zeile $i+1$ fortgesetzt.

Abbildung 5.12 zeigt den gesamten Datenaustausch der zeilenweisen Matrix-Vektor-Multiplikation für das oben betrachtete Beispiel.

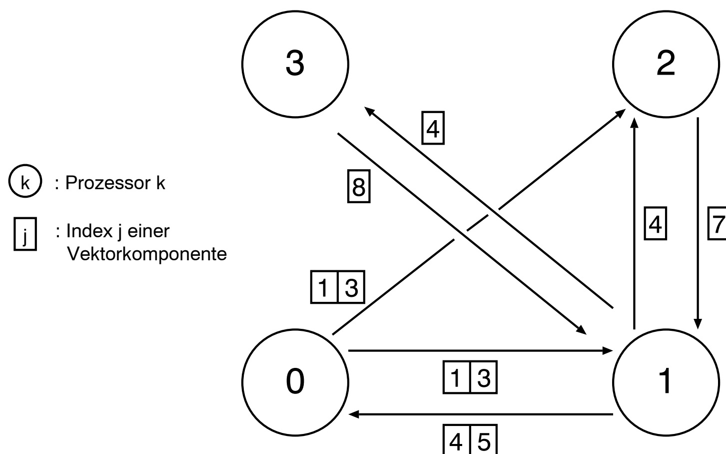


Abbildung 5.12: Datenaustausch der zeilenweisen Matrix-Vektor-Multiplikation, Umordnung in Blöcke

Auf Prozessor 1 z. B. werden die Vektorkomponenten 1 und 3 von Prozessor 0, die Komponente 7 von Prozessor 2 und die Komponente 8 von Prozessor 3 empfangen. Andererseits werden von Prozessor 1 die Komponenten 4 und 5 an Prozessor 0 sowie die Komponente 4 an Prozessor 2 und an Prozessor 3 gesendet. Durch die entwickelte Block-Sortierung kann auf die Daten, die zu

einem Zugriff auf die Vektorkomponenten eines bestimmten Prozessors führen, in einfacher Weise zugriffen werden. Während z. B. die Vektorkomponente 7 von Prozessor 2 an Prozessor 1 übermittelt wird, können auf Prozessor 1 Berechnungen mit den lokalen Vektorkomponenten 4 und 5 sowie den Matrix-Daten aus Block 1 durchgeführt werden. Die Übertragung nicht-lokaler Daten und lokale Berechnungen werden überlappt. Ist ferner die Komponente 7 auf Prozessor 1 empfangen worden, kann die Matrix-Vektor-Multiplikation unmittelbar durch Zugriff auf die Daten von Block 2 fortgesetzt werden. Diese Berechnungen werden möglicherweise überlappend mit der Übertragung der Komponenten 1 und 3 von Prozessor 0 sowie der Komponente 8 von Prozessor 3 durchgeführt.

Im folgenden wird der Ablauf der parallelen zeilenweisen Matrix-Vektor-Multiplikation mit Block-Sortierung ausführlich erläutert. Zunächst wird die Einbettung in ein iteratives Verfahren beschrieben. Der Ablauf der im Rahmen dieser Arbeit entwickelten parallelen iterativen Methoden ist in Abbildung 5.13 dargestellt.

Zu Beginn wertet jeder Prozessor k Kriterium (5.2) zur Bestimmung der Datenverteilung aus. Die dazu erforderlichen Informationen sind z. B. im Feld `row_ptr` des CRS-Schemas enthalten. Anschließend werden die den einzelnen Prozessoren zugeteilten Datensätze parallel in den privaten Speicher der jeweiligen Prozessoren übertragen. Das Kommunikationsschema für die Matrix-Vektor-Multiplikation wird in einem Vorverarbeitungsschritt ermittelt. Nach der Analyse der Indizes der Nichtnull-Elemente und der Umordnung der Matrix-Daten liegt das Schema des Datenaustauschs für die folgende Iteration fest.

In jedem Iterationsschritt einer iterativen Methode werden skalare Operationen und Vektoradditionen bzw. -skalierungen durchgeführt. Diese Operationen erfordern keine Kommunikation. Im Gegensatz dazu führen Skalarprodukte bzw. die Bestimmung einer Norm zu globaler Synchronisation. Erst wenn der globale Wert einer derartigen Operation auf jedem Prozessor vorliegt, werden die Berechnungen fortgesetzt. Die Matrix-Vektor-Multiplikation ist gewöhnlich die aufwendigste Operation jedes Iterationsschritts. Der Ablauf wird unten erläutert. Neben diesen Operationen werden Abbruchkriterien überprüft. Sind diese erfüllt, ist die Iteration beendet. Zur Auswertung eines Abbruchkriteriums ist häufig die Bestimmung einer zusätzlichen Norm und daher ein weiterer Synchronisationspunkt notwendig. Bei Kriterium (2.2) für das CG-Verfahren z. B. muß ein globales Maximum ermittelt werden. Bei Kriterien wie (2.2) kann jedoch die Kommunikation zur Normbildung mit dem Datenaustausch für Skalarprodukte oder Reduktionen, deren Berechnung die Methode erfordert, gekoppelt werden. Werden zur Überprüfung des Kriteriums in jedem Iterationsschritt nicht die aktuellsten Iterierten, sondern die der beiden vorhergehenden Schritte verwendet, so kann die erforderliche Norm an einer beliebigen Stelle im aktuellen Schritt gebildet werden. Geschieht dies zusammen mit Skalarprodukten oder Reduktionen, so ist zusätzliche Synchronisation nicht notwendig. Gegenüber dem ursprünglichen Verfahren wird ein weiterer Iterationsschritt durchgeführt; jedoch wird in jedem Schritt ein

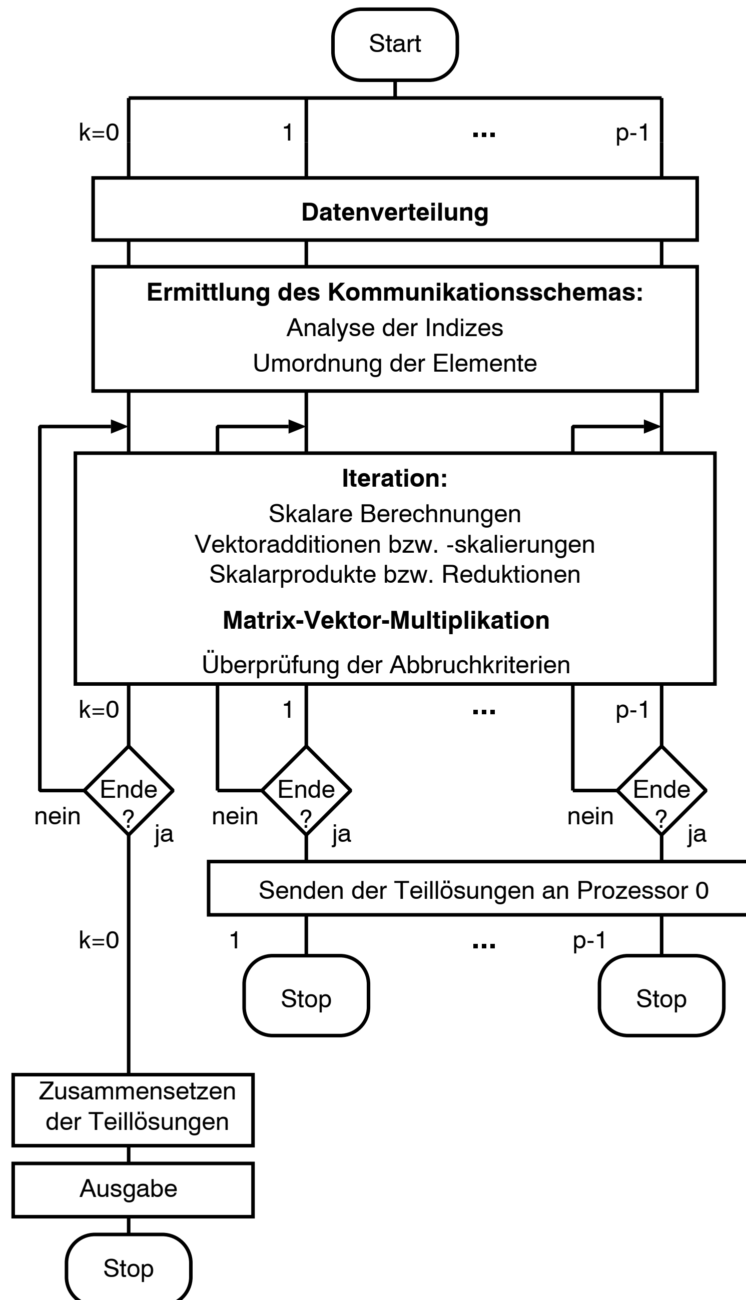


Abbildung 5.13: Ablauf eines iterativen Verfahrens auf einem Parallelrechner mit verteiltem Speicher

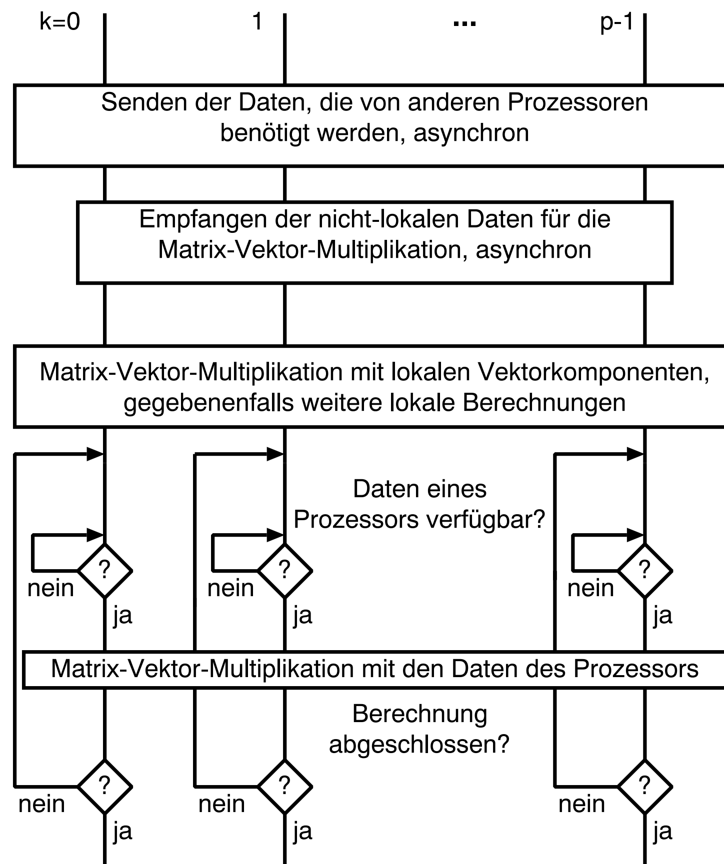


Abbildung 5.14: Ablauf der parallelen zeilenweisen Matrix-Vektor-Multiplikation mit Block-Sortierung

Synchronisationspunkt gespart. In der Lanczos-Methode erfordert die Auswertung der Abbruchkriterien die Lösung eines tridiagonalen Eigenwertproblems. In 2.4.2 wird ein paralleles Lösungsverfahren für dieses Problem beschrieben.

Nach Abbruch der Iteration wird die Gesamtlösung auf Prozessor 0 aus den Teillösungen der einzelnen Prozessoren zusammengesetzt und ausgegeben.

Abbildung 5.14 zeigt den Ablauf der parallelen zeilenweisen Matrix-Vektor-Multiplikation mit Block-Sortierung.

Zunächst werden auf jedem Prozessor die Vektorkomponenten, die andere Prozessoren benötigen, asynchron gesendet. Unmittelbar nach Aufsetzen der asynchronen Empfangsroutinen zum Empfang nicht-lokaler Komponenten wird der Teil der Matrix-Vektor-Multiplikation berechnet, der lediglich den Zugriff auf lokale Vektorkomponenten erfordert. Wenn es der Algorithmus der iterativen Methode zuläßt, werden weitere lokale Berechnungen des Iterationsschritts an dieser Stelle durchgeführt. Anschließend wird gewartet, bis die Daten irgendeines Prozessors angekommen sind und die Matrix-Vektor-Multiplikation mit diesen Wer-

ten fortgesetzt. Danach werden die Daten weiterer Prozessoren erwartet, bis die Matrix-Vektor-Multiplikation abgeschlossen ist. Rechnung und Kommunikation überlappen; während sich angeforderte Daten auf dem Netzwerk befinden, wird mit lokalen oder bereits angekommenen Daten anderer Prozessoren gerechnet.

Neuverteilung der Blöcke

Ausgangspunkt zur Ermittlung eines weiteren Kommunikationsschemas der zeilenweisen Matrix-Vektor-Multiplikation ist die oben beschriebene Block-Sortierung der Daten. Zur Laufzeit werden diese Daten-Blöcke neu auf die einzelnen Prozessoren verteilt. Jedem Prozessor werden die Blöcke zugewiesen, die lediglich zu einem Zugriff auf lokale Komponenten des Vektors der Matrix-Vektor-Multiplikation auf dem Prozessor führen. Das Ziel der Neuverteilung der Blöcke ist eine erhöhte Anzahl lokaler Berechnungen auf den einzelnen Prozessoren.

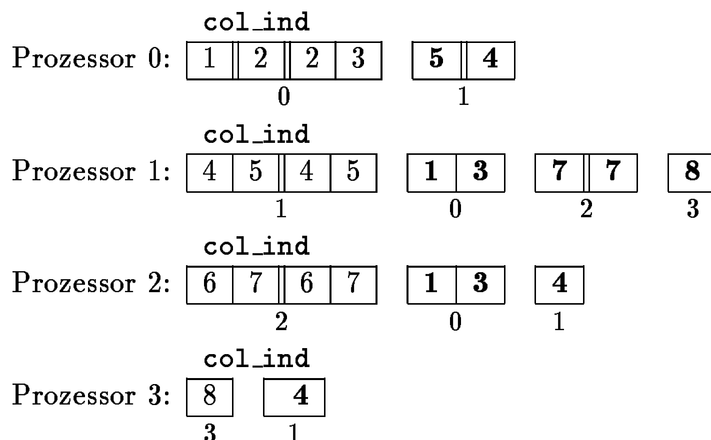
Nach der Neuverteilung der Blöcke werden auf jedem Prozessor k die Teilergebnisse $y^{[k,l]}$ des Ergebnisvektors y der Matrix-Vektor-Multiplikation $y = \boldsymbol{A}x$ berechnet. Der Index l gibt die Nummer des Prozessors an, dem das zugehörige Segment des Ergebnisvektors y zugeteilt ist. Das Teilergebnis $y^{[k,k]}$ ist das lokale Teilergebnis des Prozessors k; die übrigen Teilergebnisse $y^{[k,l]}$ mit $l \neq k$ werden nach der Berechnung auf Prozessor k an die Prozessoren l, $l \neq k$, gesendet. Nach dem Empfang der Daten auf den jeweiligen Prozessoren werden die Teilergebnisse zu den entsprechenden Komponenten des lokalen Ergebnisses addiert.

Abbildung 5.15 zeigt die Neuverteilung der Blöcke ausgehend von der Block-Sortierung der Daten für das oben betrachtete Beispiel. Unter den neu verteilten Blöcken ist die Blocknummer angegeben. Der erste Index bezeichnet die Nummer des Prozessors, auf dem der Block lokal vorhanden ist; der zweite Index entspricht der Nummer des Prozessors, von dem der Block stammt. An den letzteren Prozessor wird das Teilergebnis der Matrix-Vektor-Multiplikation, das mit den Daten dieses Blocks bestimmt wird, gesendet.

Auf Prozessor 1 z. B. wird mit den Daten des ersten Blocks das lokale Ergebnis $y^{[1,1]}$ berechnet. Mit den Daten des zweiten, dritten und vierten Blocks werden die Teilergebnisse $y^{[1,0]}$, $y^{[1,2]}$ und $y^{[1,3]}$ der Prozessoren 0, 2 und 3 bestimmt.

Die Neuverteilung der Blöcke bedingt Änderungen in den Datenstrukturen RECEIVE und SEND aus den Abbildungen 5.6 und 5.8. Die Elemente des Felds `num_ind` in der Datenstruktur SEND enthalten nach der Neuverteilung der Daten die Anzahl der nicht leeren Zeilen der Blöcke k, l mit $l \neq k$. Diese Zahl entspricht der Anzahl der Komponenten der Teilergebnisse von Prozessor l, die mit den Daten aus Block k, l auf Prozessor k berechnet werden. In den Spalten des Felds `to_proc` sind anstelle der Indizes der Vektorkomponenten die Indizes der Komponenten der Teilergebnisse hintereinander abgelegt. Die Nummer der Prozessoren, an die Teilergebnisse gesendet werden, ist in `proc_id` gespeichert und entspricht dem Index l der Blöcke k, l mit $l \neq k$. In der Datenstruktur RECEIVE sind die entsprechenden Daten der Blöcke von Prozessor k abgelegt, die bei der Neuver-

Block-Sortierung



Neuverteilung der Blöcke

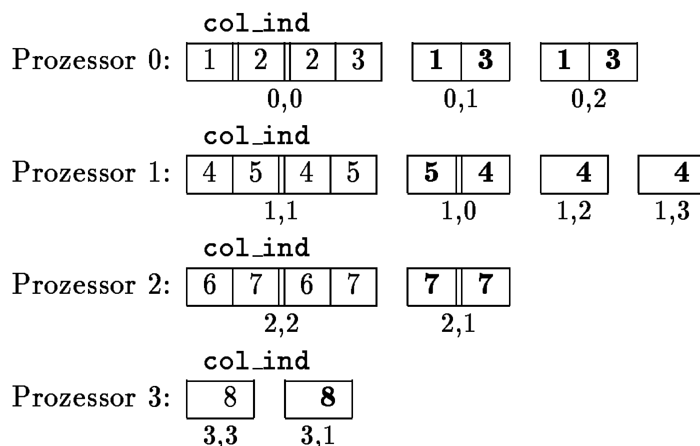


Abbildung 5.15: Datenverteilung vor und nach der Neuverteilung der Blöcke

teilung anderen Prozessoren zugewiesen werden. Die Anzahl der Komponenten der Teilergebnisse, die Indizes dieser Komponenten und die Nummer der Prozessoren, von denen auf Prozessor k Teilergebnisse empfangen werden, sind in den Feldern `num_ind`, `from_proc` und `proc_id` gespeichert. Das Kommunikationsschema der Matrix-Vektor-Multiplikation nach der Neuverteilung der Blöcke erfordert den Austausch von Daten mit denselben Prozessoren wie das oben beschriebenen Schema ohne die Neuverteilung der Blöcke.

Zur Speicherung der Matrix-Daten von Prozessor k, $k = 0,\ldots,p-1$, bei Neuverteilung der Blöcke wird die Datenstruktur aus Abbildung 5.16 verwendet. Der Datenstruktur liegt das CRS-Schema zugrunde; die Daten sind ähnlich wie

in Abbildung 5.11 strukturiert.

MATRIX = **record**
  `value` : **array** $[1..e_k^{\text{neu}}]$ **of** REAL
  `col_ind` : **array** $[1..e_k^{\text{neu}}]$ **of** INTEGER
  `row_ptr` : **array** $[1..n_{\max}+1, 1..b_k^{\text{neu}}]$ **of** INTEGER
  `block_id` : **array** $[1..b_k^{\text{neu}}]$ **of** INTEGER
end record

Abbildung 5.16: Verteilte Datenstruktur bei Neuverteilung der Blöcke

Durch die Neuverteilung der Blöcke ändert sich gewöhnlich sowohl die Anzahl der Nichtnull-Elemente als auch die Anzahl der Daten-Blöcke pro Prozessor. Ferner kann ein Block von Prozessor k nun Daten aus $n_{\max}$ Zeilen enthalten. Im Feld `block_id` ist die Nummer l des Blocks k, l gespeichert. Durch Verwendung der Informationen, die in den Feldern `num_ind` und `to_proc` der Datenstruktur SEND abgelegt sind, wird bei der Matrix-Vektor-Multiplikation nur auf nicht leere Zeilen der Blöcke k, l mit $l \neq k$ zugegriffen. Diese Informationen werden bei der Ermittlung der Position der Zeilen in `value` und `col_ind` durch das Feld `row_ptr` genutzt.

In Abbildung 5.17 ist das Schema des Datenaustauschs für die Datenverteilung aus 5.15 nach der Neuverteilung der Blöcke dargestellt. Ausgetauscht werden die Teilergebnisse $y^{[k,l]}$ mit $k \neq l$, die nach dem Empfang auf Prozessor l zu den zugehörigen Komponenten des lokalen Ergebnisses $y^{[l,l]}$ addiert werden.

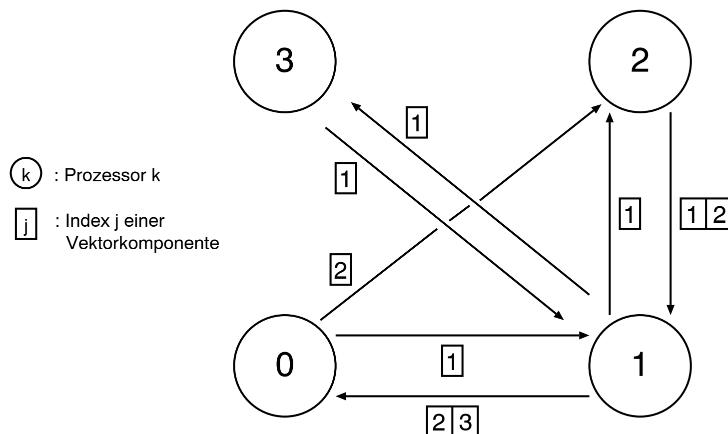


Abbildung 5.17: Datenaustausch der zeilenweisen Matrix-Vektor-Multiplikation, Neuverteilung der Blöcke

Von Prozessor 1 z. B. werden an Prozessor 0 zwei Werte gesendet. Auf Prozessor 0 werden diese Werte und die zweite bzw. dritte Komponente des lokalen Ergebnisses $y^{[0,0]}$ summiert. Andererseits wird Prozessor 1 von Prozessor 0 ein Wert übermittelt, der zur ersten Komponente von $y^{[1,1]}$ addiert wird.

Abbildung 5.18 zeigt den Ablauf der parallelen zeilenweisen Matrix-Vektor-Multiplikation bei Neuverteilung der Blöcke.

Zunächst werden auf jedem Prozessor asynchrone Routinen zum Empfang aller erforderlichen Teilergebnisse anderer Prozessoren ausgeführt. Anschließend werden die Teile des Ergebnisvektors berechnet, die anderen Prozessoren übermittelt werden. Die Berechnung wird pro Daten-Block durchgeführt; nach der Ermittlung eines Teilergebnisses werden die Daten asynchron an den entsprechenden Prozessor gesendet. Danach werden alle lokalen Operationen des aktuellen Iterationsschritts der iterativen Methode durchgeführt, die an dieser Stelle berechnet werden können, insbesondere die Berechnung des lokalen Teilergebnisses der Matrix-Vektor-Multiplikation. Anschließend wird gewartet, bis die Werte irgendeines Prozessors angekommen sind, die dann zum lokalen Ergebnisvektor addiert werden. Schließlich werden die Daten weiterer Prozessoren erwartet, bis die Matrix-Vektor-Multiplikation abgeschlossen ist. Rechnung und Kommunikation werden überlappend durchgeführt.

Da Teile des Ergebnisvektors ausgetauscht werden, erfolgt der größte Teil der Berechnungen für die parallele Matrix-Vektor-Multiplikation lokal auf den einzelnen Prozessoren. Lediglich die Summation der Teilergebnisse geschieht nach dem Erhalt der nicht-lokalen Daten. Da die Teilergebnisse jedoch zuerst berechnet werden müssen, bevor sie übermittelt werden können, ist der Grad der Überlappung von Rechnung und Kommunikation geringer als bei der Matrix-Vektor-Multiplikation nach Abbildung 5.14. Ferner kann nach der Neuverteilung der Blöcke die Rechenlast ungleichmäßig verteilt sein; einige Prozessoren können mehr oder größere Daten-Blöcke als andere besitzen. Das hier betrachtete Kommunikationsschema läßt jedoch beliebige Datenverteilungen zu; jedem Prozessor können beliebige Teile von beliebigen, auch nicht aufeinanderfolgenden Zeilen zugeteilt werden. Berücksichtigt man die Numerierung der Knoten des Diskretisierungsgitters einer speziellen FE-Anwendung, so lassen sich für dieses Schema günstige Datenverteilungen finden. Die hier dargestellten Datenverteilungs- und Kommunikationsschemata hingegen erfordern kein Wissen über das spezielle Problem. Datenverteilung und Kommunikationsschema werden automatisch durch Analyse der Spaltenindizes der Nichtnull-Elemente der Matrix ermittelt.

5.4.3 Spaltenweise Matrix-Vektor-Multiplikation

Für die spaltenweise Matrix-Vektor-Multiplikation wird die spaltenweise Speicherung der Matrix-Daten — z. B. im CCS-Format — vorausgesetzt. Die Verteilung der Matrix-Spalten erfolgt analog zur in 5.3 beschriebenen zeilenweisen Aufteilung der Matrix-Daten. n_k bezeichnet dann die Anzahl der Spalten und g_k die Nummer der ersten Spalte des Prozessors k. Die Segmentierung der Vektor-Daten entspricht der Aufteilung der Matrix-Spalten.

Bei der parallelen spaltenweisen Matrix-Vektor-Multiplikation $y = \boldsymbol{A}x$ werden Teilergebnisse des Ergebnisvektors y zwischen den einzelnen Prozessoren ausge-

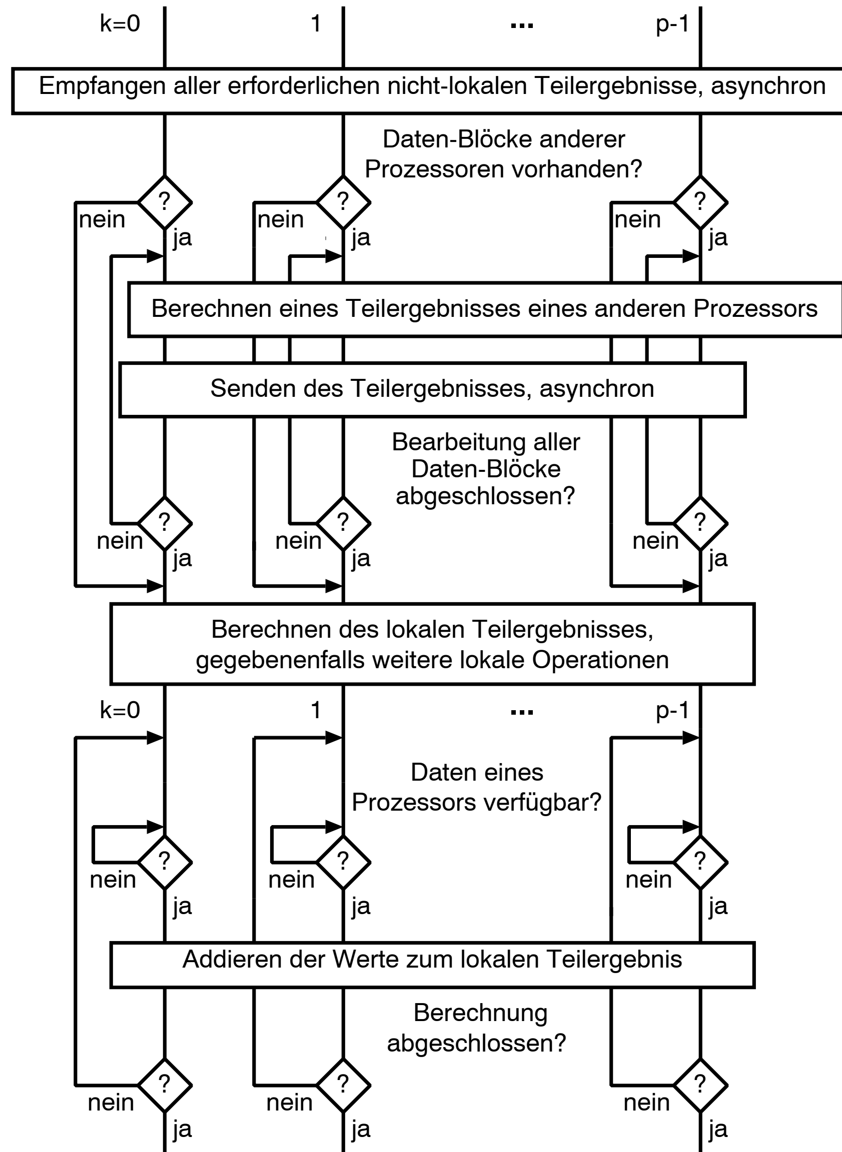


Abbildung 5.18: Ablauf der parallelen zeilenweisen Matrix-Vektor-Multiplikation bei Neuverteilung der Blöcke

tauscht. Während der Operation

$$y_i = \sum_{j=g_k}^{g_k+n_k-1} a_{ij}x_j, \quad i = 1, \ldots, n,$$

auf Prozessor k werden Teilergebnisse von Komponenten des Ergebnisvektors y berechnet, die anderen Prozessoren zugeteilt sind.

Die Analyse der Zeilenindizes der Nichtnull-Elemente der Spalten wird wie in 5.4.1 beschrieben durchgeführt. Da jedoch Teilergebnisse ausgetauscht werden, wechseln die Datenstrukturen aus 5.4.1 die Funktion. Die Elemente der Struktur RECEIVE geben an, welche Daten an welche Prozessoren gesendet werden, während die Datenstruktur SEND die Information enthält, welche Daten von welchen Prozessoren empfangen werden.

Die Umordnung der Matrix-Daten pro Prozessor wird analog zur Block-Sortierung für die zeilenweise Matrix-Vektor-Multiplikation durchgeführt. Bei der spaltenweisen Matrix-Vektor-Multiplikation sind die Elemente der Blöcke spaltenweise mit aufsteigendem Zeilenindex pro Spalte geordnet. Auf Prozessor k wird mit den Elementen des Blocks h mit $h \neq k$ ein Teilergebnis von Prozessor h berechnet. Zur Speicherung der Blöcke wird eine ähnliche Datenstruktur wie in Abbildung 5.11 verwendet; die Speicherung der Daten in dieser Struktur ist spaltenorientiert.

Abbildung 5.19 zeigt das Daten-Transferschema der parallelen spaltenweisen Matrix-Vektor-Multiplikation für die Verteilung der Transponierten der Matrix (5.4) auf vier Prozessoren nach Kriterium (5.2) mit $\xi = 16$ und $s = 2$. Da die Speicherung der Transponierten der Matrix im CCS-Format der Speicherung der Matrix im CRS-Schema entspricht, erfolgen alle Zwischenschritte bis einschließlich der Block-Sortierung der Matrix-Daten wie oben für das zeilenweise Verfahren beschrieben. Gegenüber Abbildung 5.12 geschieht der Datenaustausch in umgekehrter Richtung.

Von Prozessor 1 z. B. werden an Prozessor 0 Teilergebnisse der Komponenten 1 und 3 des Ergebnisvektors y gesendet und dort zur ersten und dritten Komponente des lokalen Vektors $y^{[0]}$ addiert. Auf Prozessor 1 werden andererseits von Prozessor 0 Teilergebnisse der Komponenten 4 und 5 von y empfangen. Diese Werte und die lokalen Komponenten $y_1^{[1]}$ und $y_2^{[1]}$ werden summiert.

Die Strategie für die Überlappung von Rechnung und Kommunikation bei der spaltenweisen Matrix-Vektor-Multiplikation ist die gleiche wie beim zeilenweisen Verfahren nach Neuverteilung der Blöcke. Der Ablauf der spaltenweisen Methode entspricht dem Schema aus Abbildung 5.18. Teilergebnisse werden zunächst berechnet und dann gesendet. Gegenüber dem zeilenweisen Verfahren aus Abbildung 5.14 ist dies ein Nachteil, da der Grad der Überlappung von Rechnung und Kommunikation geringer ist. Die Anzahl der lokalen Operationen, bevor mit nicht-lokalen Daten gerechnet wird, ist jedoch höher, da nach dem Empfang nicht-lokaler Werte lediglich eine Summation von Vektorkomponenten

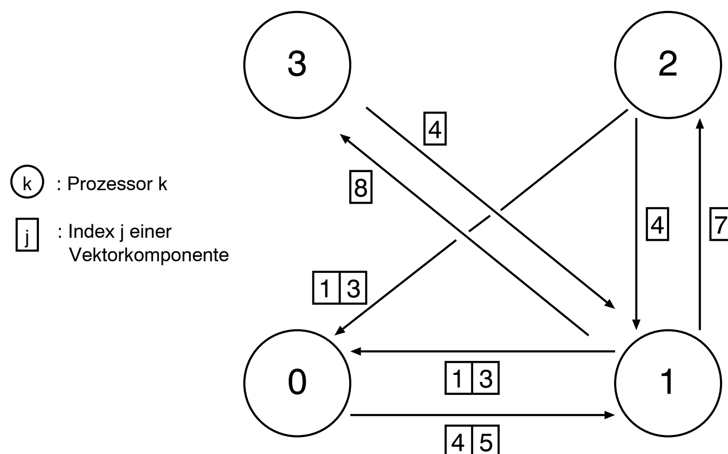


Abbildung 5.19: Datenaustausch der spaltenweisen Matrix-Vektor-Multiplikation, Umordnung in Blöcke

durchgeführt wird. Gegenüber der zeilenweisen Methode mit Neuverteilung der Blöcke hat das spaltenweise Verfahren den Vorteil, daß die gleichmäßige Verteilung der Rechenlast nach (5.2) gegeben ist und nicht durch Neuverteilung der Daten beeinflußt wird.

5.4.4 Kopplung des zeilen- und spaltenweisen Verfahrens

In den iterativen Methoden QMR aus 2.3.1 und BiCG [92] werden in jedem Iterationsschritt zwei voneinander unabhängige Matrix-Vektor-Multiplikationen der Form $y = \boldsymbol{A}s$ und $z = \boldsymbol{A}^T t$ durchgeführt. Ist die Matrix $\boldsymbol{A}$ z.B. im CRS-Format gespeichert, so kann für die Operation $y = \boldsymbol{A}s$ das zeilenweise Verfahren mit Block-Sortierung nach Abbildung 5.14 und für die Operation $z = \boldsymbol{A}^T t$ der oben beschriebene spaltenweise Algorithmus verwendet werden. Dieses Vorgehen erlaubt für beide Operationen die Verwendung derselben Datenstruktur.

Ferner kann der Datenaustausch für beide Verfahren gekoppelt werden, d. h. auf Prozessor k, $k = 0, \ldots, p-1$, werden die Daten, die auf Prozessor h zur Berechnung der lokalen Segmente $y^{[h]}$ und $z^{[h]}$ benötigt werden, in einer Nachricht zusammengefaßt. Dies ist möglich, wenn von Prozessor k sowohl Daten zur Berechnung von $y^{[h]}$ als auch von $z^{[h]}$ an Prozessor h gesendet werden. Durch die Kopplung wird sowohl die Anzahl der Nachrichten für beide Operationen verringert als auch der Grad der Überlappung von Rechnung und Kommunikation erhöht. Während sich angeforderte Daten auf dem Verbindungsnetzwerk der Prozessoren befinden, können die lokalen Berechnungen beider Operationen durchgeführt werden.

In Abbildung 5.20 ist der Ablauf der gekoppelten Matrix-Vektor-Multiplika-

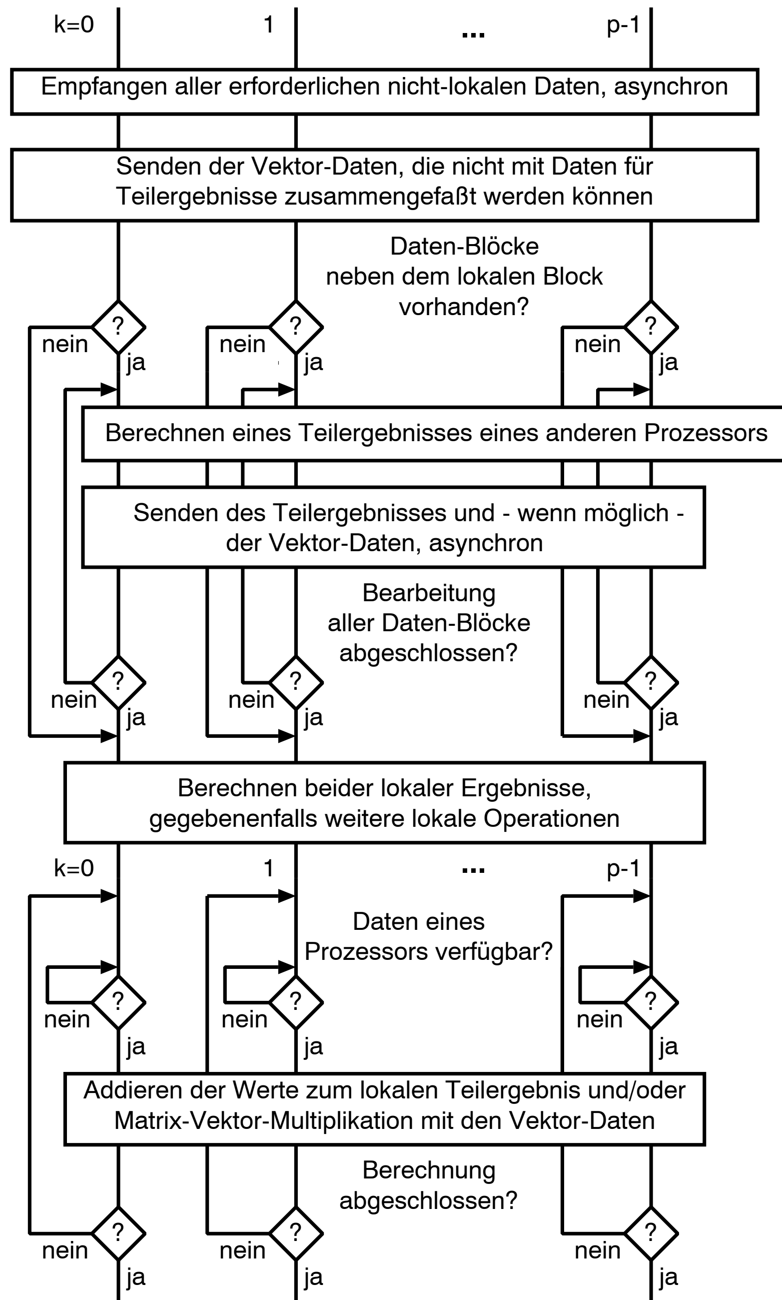


Abbildung 5.20: Kopplung der parallelen zeilen- und spaltenweisen Matrix-Vektor-Multiplikation

tion für die Operationen $y = \boldsymbol{A}s$ und $z = \boldsymbol{A}^T t$ dargestellt.

Zunächst werden auf jedem Prozessor asynchrone Routinen zum Empfang aller erforderlichen nicht-lokalen Daten ausgeführt. Dies können Komponenten von s, Teilergebnisse von Komponenten von z oder in einer Nachricht zusammengefaßt Komponenten von s und Teilergebnisse von Komponenten von z sein. Anschließend werden die Vektor-Daten von s, die nicht zusammen mit einem Teilergebnis von z verschickt werden können, asynchron gesendet. Danach werden die Teile des Ergebnisvektors z berechnet, die anderen Prozessoren übermittelt werden. Nach Bestimmung der Teilergebnisse von z werden diese entweder mit Vektor-Daten von s in einer Nachricht zusammengefaßt oder einzeln gesendet. In beiden Fällen erfolgt die Kommunikation asynchron. Anschließend wird das lokale Teilergebnis der spaltenweisen Matrix-Vektor-Multiplikation bestimmt, und die lokalen Berechnungen für die zeilenweise Matrix-Vektor-Multiplikation werden durchgeführt. Gegebenenfalls können weitere Operationen des aktuellen Iterationsschritts der iterativen Methode an dieser Stelle berechnet werden. Anschließend wird gewartet, bis die Werte irgendeines Prozessors angekommen sind. Sind dies Vektor-Daten, so wird die zeilenweise Matrix-Vektor-Multiplikation mit diesen Daten fortgesetzt. Wird ein Teilergebnis der spaltenweisen Matrix-Vektor-Multiplikation empfangen, wird dieses zu den entsprechenden Komponenten des lokalen Teilergebnisses addiert. Kommen zusammengefaßt Daten für beide Operationen an, werden nacheinander beide Berechnungen mit diesen Daten durchgeführt. Schließlich werden die Daten weiterer Prozessoren erwartet, bis die zeilenweise und die spaltenweise Matrix-Vektor-Multiplikation abgeschlossen sind. Transferzeiten nicht-lokaler Daten werden durch lokale Berechnungen für beide Operationen überlappt.

5.5 Integration in eine funktionale Programmiersprache

Die Programmierung von massiv-parallelen Rechnersystemen mit verteiltem Speicher in imperativen Programmiersprachen ist aufwendig, insbesondere bei irregulären Datenstrukturen wie Speicherschemata für dünnbesetzte Matrizen. Datenverteilung, Austausch von Daten und Synchronisation müssen explizit im Programm formuliert werden. Ferner müssen Verklemmungen (*Deadlocks*) vermieden werden.

Funktionale Sprachen hingegen besitzen eine implizite Parallelität (s. auch 4.2.3). Sie sind seiteneffektfrei, d. h. die Auswertungsreihenfolge unabhängiger Teilausdrücke ist irrelevant für das Gesamtergebnis. Diese Eigenschaften ermöglichen die Programmierung von Parallelrechnern auf hohem Abstraktionsniveau. Die Realisierung der Datenverteilung, des Austauschs von Daten und der Synchronisation übernehmen Compiler und Laufzeitsystem; Verklemmungen sind

nicht möglich [62]. Gegenüber parallelen imperativen Implementierungen besitzen funktionale Programme jedoch meist ein schlechtes Laufzeitverhalten. Häufig werden die Probleme des Lastausgleichs und der Kommunikation von Compiler und Laufzeitsystem unzureichend gelöst. Ferner werden Datenstrukturen wie z. B. Felder, die in imperativen Programmiersprachen u. a. zur effizienten Verwaltung von Matrizen genutzt werden, kaum unterstützt. Aufgrund der Forderung nach Seiteneffektfreiheit ist *Update in Place* nicht möglich, d. h. die Elemente einer Datenstruktur dürfen nicht im Speicher geändert werden.

Zur Lösung dieser Probleme wird in [135] die Integration *algorithmischer Skelette* in eine funktionale Programmiersprache untersucht. Algorithmische Skelette sind Funktionen höherer Ordnung, die einen parallelen Algorithmus oder eine Parallelisierungsstrategie realisieren. Diese Funktionen sind nach außen seiteneffektfrei, können intern jedoch seiteneffektbehaftet sein. Daher können Datenstrukturen innerhalb eines Skeletts wie in imperativen Sprachen üblich verwaltet werden. Die Integration algorithmischer Skelette steigert die Effizienz funktionaler Implementierungen unter Beibehaltung des hohen Abstraktionsniveaus funktionaler Sprachen.

In Zusammenarbeit mit dem Lehrstuhl für Informatik II der RWTH Aachen wurden einige Methoden für dünnbesetzte Matrizen als algorithmische Skelette in den Sprachumfang einer funktionalen Sprache integriert, da Operationen mit dünnbesetzten Matrizen in vielen technischen Anwendungen auftreten. Im folgenden werden diese Methoden beschrieben, und schließlich wird ein paralleles funktionales Programm für die Matrix-Vektor-Multiplikation vorgestellt.

Als algorithmische Skelette sind die Datenverteilung nach Kriterium (5.2), die Ermittlung des Kommunikationsschemas mit Sortierung der Matrix-Daten und die parallele zeilenweise Matrix-Vektor-Multiplikation realisiert. Die Speicherung der Matrix-Daten basiert auf dem CRS-Schema. Zur Verteilung der Matrix-Daten wird $\xi = 0$ verwendet, da nur die Matrix-Vektor-Multiplikation betrachtet wird. Für die Matrix (5.4) liegt dann bei vier Prozessoren die Verteilung aus Abbildung 5.3 vor. Zur Ermittlung des Kommunikationsschemas werden die Matrix-Daten pro Zeile nach aufsteigendem Spaltenindex sortiert und die Spaltenindizes nach der Methode aus 5.4.1 analysiert.

Zur weiteren Sortierung der Daten wird ein vereinfachtes Schema verwendet. Lediglich zwei Blöcke werden gebildet. Der erste Block enthält die Daten, die zu einem Zugriff auf lokale Komponenten des Vektors führen. Die Elemente des zweiten Blocks erfordern einen Zugriff auf nicht-lokale Komponenten. Ferner werden die Spaltenindizes des zweiten Blocks auf die Komponenten einer Datenstruktur `non_local` bezogen, in der die nicht-lokalen Vektorkomponenten nach dem Empfang mit aufsteigendem Index abgelegt werden. Das Feld `non_local` von Prozessor k hat die Länge $l_{\mathrm{nl},k}$ mit $l_{\mathrm{nl},k}$ als der Anzahl der erforderlichen nicht-lokalen Komponenten und den Typ REAL. Die Index-Transformation und die Verwendung des Felds `non_local` lassen die Speicherung der nicht-lokalen Vektorkomponenten unmittelbar hintereinander zu. Dadurch ist der erforderli-

che Speicherplatz für die nicht-lokalen Daten minimal und der Zugriff auf diese Daten während der Matrix-Vektor-Multiplikation erleichtert.

Abbildung 5.21 zeigt das Schema der Umordnung und der Index-Transformation für das Feld `col_ind` von Prozessor 1 ausgehend von der Datenverteilung aus Abbildung 5.3. Die Daten sind bereits nach aufsteigendem Spaltenindex geordnet. Unter den Elementen des zweiten Blocks sind die Nummern der Prozessoren angegeben, auf deren Vektorkomponenten bei der Matrix-Vektor-Multiplikation zugegriffen wird. Die Prozessornummern unter dem Feld `non_local` bezeichnen den Prozessor, auf dem die an Prozessor 1 übermittelten Komponenten lokal vorhanden sind. Über den Elementen von `non_local` sind die Indizes des Felds dargestellt, auf die die Elemente des zweiten Blocks von `col_ind` bei der Index-Transformation bezogen werden.

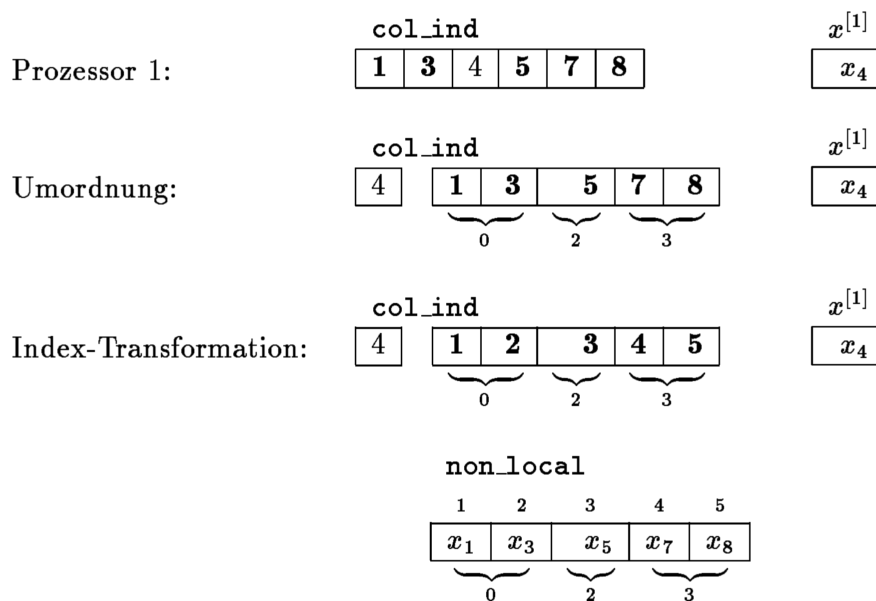


Abbildung 5.21: Umordnung in zwei Blöcke und Index-Transformation

Die parallele zeilenweise Matrix-Vektor-Multiplikation bei vereinfachter Sortierung der Matrix-Daten verläuft ähnlich wie das in Abbildung 5.14 dargestellte Verfahren mit Block-Sortierung. Nach der Berechnung aller lokalen Operationen wird jedoch gewartet, bis alle erforderlichen nicht-lokalen Daten angekommen sind. Die Werte dieser Vektorkomponenten werden im Feld `non_local` nach aufsteigendem Index der Komponenten sortiert abgelegt. Anschließend wird die Matrix-Vektor-Multiplikation unter Verwendung der Elemente des zweiten Blocks der Matrix-Daten und der Elemente des Felds `non_local` abgeschlossen.

Ein Programm in der funktionalen Programmiersprache FPS zur Berechnung der parallelen zeilenweisen Matrix-Vektor-Multiplikation ist in Abbildung 5.22 dargestellt [135].

```
EXTERN sk_load_sparse_matrix:  REAL → (SMATRIX*INT);
EXTERN sk_compute_communication_scheme:  SMATRIX → SMATRIX;
EXTERN sk_generate_dependent_vector:  SMATRIX → REAL → (ARRAY*INT);
EXTERN sk_sparse_mvp:  SMATRIX → ARRAY → ARRAY → ARRAY;

EVAL LETSEQ
   (smat1,ord) = sk_load_sparse_matrix 0.0;
   smat2 = sk_compute_communication_scheme smat1;
   (vec1,dim1) = sk_generate_dependent_vector smat2 1.0;
   (vec2,dim2) = sk_generate_dependent_vector smat2 0.0;
   result = sk_sparse_mvp smat2 vec1 vec2
IN result
END.
```

Abbildung 5.22: Ein funktionales Programm für die parallele Matrix-Vektor-Multiplikation mit dünnbesetzter Matrix

Im Programm sind zunächst die erforderlichen Deklarationen für die einzelnen Skelette angegeben. Als erstes ist der Typ der Eingabeparameter des Skeletts genannt, dann der Typ des Ergebnisses. Das Skelett `sk_load_sparse_matrix` z. B. hat einen Eingabeparameter vom Typ `REAL`; das Ergebnis ist ein Wert vom Typ `SMATRIX` und ein Wert vom Typ `INT`. `INT` ist der Typ für ganze Zahlen, `SMATRIX` der Typ für dünnbesetzte Matrizen. Ferner bezeichnet `ARRAY` ein ein- oder zweidimensionales Feld. Die folgende Code-Sequenz wird nacheinander bearbeitet. Dies ist durch die Anweisung `EVAL LETSEQ` sichergestellt. Ohne diese Anweisung würden die einzelnen Ausdrücke in beliebiger Reihenfolge ausgewertet. Das erste Skelett verteilt eine dünnbesetzte Matrix `smat1` der Ordnung `ord` auf alle zur Verfügung stehenden Prozessoren. Die Matrix-Daten werden aus dem Speicher gelesen; Eingabeparameter ist der Parameter ξ der Datenverteilung (5.2) mit dem Wert 0.0 zur gleichmäßigen Verteilung der Nichtnull-Elemente der Matrix. Das nächste Skelett bestimmt parallel das Kommunikationsschema und sortiert die Matrix-Daten gemäß Abbildung 5.21. Anschließend wird ein Test-Vektor für die Matrix-Vektor-Multiplikation erzeugt; alle Komponenten besitzen den Werte 1.0. Danach wird der Ergebnisvektor angelegt und mit dem Wert 0.0 vorbesetzt. Über den Eingabeparameter `smat2` wird sichergestellt, daß die Dimension beider Vektoren der Ordnung der Matrix entspricht. Schließlich wird die Matrix-Vektor-Multiplikation parallel durchgeführt und das Ergebnis dem Vektor `result` zugewiesen.

Das Programm aus Abbildung 5.22 nutzt eine beliebige, vor Programmstart zur Verfügung stehende Anzahl von Prozessoren. Speicherschema der Matrix, Datenverteilung, Kommunikation und parallele Berechnung sind für den Programmierer transparent. Für den Einsatz der parallelen Matrix-Vektor-Multiplikation in einem iterativen Verfahren läßt sich die Datenverteilung über den

Eingabeparameter ξ steuern.

Eine sinnvolle Erweiterung der in eine funktionale Sprache integrierten Methoden ist die Berücksichtigung des Besetzungsmusters der Matrix. In diesem Fall können als Funktion der Struktur der Matrix unterschiedliche Speichertechniken und Methoden der Matrix-Vektor-Multiplikation verwendet werden (s. Kapitel 3). Kann die Struktur und Besetzung der Matrix automatisch analysiert werden, ist für den Programmierer transparent, ob Operationen mit vollbesetzten, regelmäßig dünnbesetzten oder unstrukturiert dünnbesetzten Matrizen durchgeführt werden. Andernfalls kann der Programmierer Informationen über das Besetzungsmuster der Matrix eines speziellen Problems vorgeben; vom Compiler bzw. Laufzeitsystem können dann ein geeignetes Speicherschema und davon abhängig die Methode der Matrix-Vektor-Multiplikation gewählt werden.

Kapitel 6

Ergebnisse

Numerische und Performance-Untersuchungen der entwickelten parallelen Verfahren wurden auf dem massiv-parallelen System mit verteiltem Speicher PARAGON XP/S 10 der Forschungszentrum Jülich GmbH durchgeführt. Die Programme wurden mit dem PARAGON FORTRAN Compiler, Version 4.5, unter Verwendung der Optionen *-O4 -Knoieee* übersetzt [4] [5]. Die Ausführungszeiten der Programme wurden unter dem Betriebssystem PARAGON OSF/1, Release 1.2, gemessen, das den Einsatz des Kommunikationsprozessors ermöglicht [7]. Einige wenige Messungen, die unter dem Betriebssystem PARAGON OSF/1, Release 1.1, ohne Einsatz des Kommunikationsprozessors durchgeführt wurden, sind besonders gekennzeichnet [6].

Im folgenden werden zunächst Eigenschaften der in den Untersuchungen verwendeten Matrizen beschrieben und anschließend Performance-Ergebnisse verschiedener Methoden zur parallelen Matrix-Vektor-Multiplikation vorgestellt. Danach wird auf numerische Eigenschaften, Zeitverhalten und Skalierung der entwickelten parallelen iterativen Algorithmen zur Lösung von Gleichungssystemen und Eigenwertproblemen eingegangen. Schließlich werden Ergebnisse zur Performance algorithmischer Skelette für die parallele Matrix-Vektor-Multiplikation mit dünnbesetzter Matrix in einer funktionalen Programmiersprache präsentiert. Die letzteren Untersuchungen wurden auf einem Transputer-System des Rechenzentrums der RWTH Aachen durchgeführt [59].

6.1 Verwendete Matrizen

Zur Untersuchung der entwickelten parallelen Methoden für dünnbesetzte Matrizen werden symmetrische und unsymmetrische Matrizen aus technischen und physikalischen Problemstellungen verwendet. Einige Eigenschaften dieser Matrizen werden im folgenden beschrieben.

6.1.1 Symmetrische Matrizen

Matrizen aus der Umwelttechnik

Die Matrizen **ICG2D** und **ICG3D** stammen aus einem FE-Modell des Instituts für Chemie und Dynamik der Geosphäre (ICG4) der Forschungszentrum Jülich GmbH. Mit Hilfe dieses FE-Modells wird die Ausbreitung von Schadstoffen in geologischen Systemen simuliert [140] [145]. Beide Matrizen sind symmetrisch positiv definit. In Tabelle 6.1 sind Kenngrößen beider Matrizen angegeben.

	ICG2D	**ICG3D**
Ordnung	17368	49392
Nichtnull-Elemente	304000	1242814
Besetzungsgrad	0.1%	0.05%
$z_{\mathbf{max}}$	18	27
$z_{\mathbf{mean}}$	17.5	25.2
Konditionszahl	$\approx 10^3$	$\approx 10^3$

Tabelle 6.1: Kenngrößen der Matrizen **ICG2D** und **ICG3D**

Bei beiden Matrizen liegt die mittlere Anzahl von Nichtnull-Elementen pro Zeile $z_{\mathbf{mean}}$ nahe an der maximalen Anzahl $z_{\mathbf{max}}$, da das Diskretisierungsgitter des FE-Modells regelmäßig strukturiert ist. Auf das Besetzungsmuster der Matrizen **ICG2D** und **ICG3D** wird unten eingegangen. Zur Approximation der Konditionszahl wurde der entwickelte parallele Lanczos-Algorithmus verwendet (s. 2.4).

Matrix aus der Strukturmechanik

Im Institut für Sicherheitsforschung und Reaktortechnik der Forschungszentrum Jülich GmbH werden u. a. thermische Spannungen in Materialien mit Hilfe von FE-Modellen untersucht. Aus diesem Bereich stammt die symmetrische positiv definite Matrix **ISR**. Kenngrößen der Matrix sind in Tabelle 6.2 zusammengefaßt.

	ISR
Ordnung	25222
Nichtnull-Elemente	3856386
Besetzungsgrad	0.6%
$z_{\mathbf{max}}$	485
$z_{\mathbf{mean}}$	152.9
Konditionszahl	$\approx 10^5$

Tabelle 6.2: Kenngrößen der Matrix **ISR**

Gegenüber den Matrizen aus der Umwelttechnik besitzt die Matrix **ISR** einen deutlich höheren Besetzungsgrad. Ferner unterscheiden sich die maximale und die mittlere Anzahl der Nichtnull-Elemente beträchtlich; das Diskretisierungsgitter weist eine unregelmäßige Struktur auf. Das Besetzungsmuster der Matrix **ISR** ist in einem späteren Abschnitt dargestellt.

Matrix aus der Automobilindustrie

Verformungsvorgänge beim Pressen von Motorhauben können mit Hilfe der Methode der finiten Elemente beschrieben werden. Die symmetrische positiv definite Matrix **PRESS** entsteht bei der Modellierung dieses Problems aus der Automobilindustrie. Tabelle 6.3 zeigt die Kenngrößen dieser Matrix.

	PRESS
Ordnung	13860
Nichtnull-Elemente	661010
Besetzungsgrad	0.3%
$z_{\max}$	49
z_{mean}	47.7
Konditionszahl	$\approx 10^9$

Tabelle 6.3: Kenngrößen der Matrix **PRESS**

Im Vergleich zu den oben beschriebenen Matrizen besitzt die Matrix **PRESS** eine deutlich höhere Konditionszahl.

Harwell-Boeing Sparse Matrix Collection

Die *Harwell-Boeing Sparse Matrix Collection* ist eine Standardsammlung dünnbesetzter Matrizen aus unterschiedlichen Anwendungsbereichen [54] [55]. In der Literatur werden Matrizen aus dieser Sammlung häufig als Testbeispiele für iterative Verfahren zur Lösung von Gleichungssystemen und Eigenwertproblemen verwendet.

Die in dieser Arbeit verwendeten Matrizen **BCSSTK17** und **BCSSTK18** stammen aus FE-Modellen zur Statikberechnung. Beide Matrizen sind symmetrisch positiv definit. Kenngrößen dieser Matrizen sind in Tabelle 6.4 aufgeführt.

Beide Matrizen weisen hohe Konditionszahlen und deutlich unterschiedliche Werte für $z_{\max}$ und z_{mean} auf.

Matrix aus der Diskretisierung der Laplace-Gleichung

Die Matrix $\boldsymbol{A}$ in (6.1) ist ein Testbeispiel aus [44] und entsteht bei der Diskretisierung der Laplace-Gleichung. Sie wird im folgenden mit **LAPLACE** bezeichnet.

	BCSSTK17	**BCSSTK18**
Ordnung	10974	11948
Nichtnull-Elemente	428650	149090
Besetzungsgrad	0.4%	0.1%
$z_{\mathbf{max}}$	150	49
$z_{\mathbf{mean}}$	39.1	12.5
Konditionszahl	$\approx 10^7$	$\approx 10^{10}$

Tabelle 6.4: Kenngrößen der Matrizen **BCSSTK17** und **BCSSTK18**

$$\boldsymbol{A} = \begin{pmatrix} 1 & -\frac{1}{4} & -\frac{1}{4} & 0 & 0 & 0 & 0 & \cdots & 0 \\ -\frac{1}{4} & 1 & 0 & -\frac{1}{4} & 0 & 0 & 0 & \cdots & 0 \\ -\frac{1}{4} & 0 & 1 & -\frac{1}{4} & -\frac{1}{4} & 0 & 0 & \cdots & 0 \\ 0 & -\frac{1}{4} & -\frac{1}{4} & 1 & 0 & -\frac{1}{4} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -\frac{1}{4} & 0 & 1 & -\frac{1}{4} & -\frac{1}{4} & 0 \\ 0 & \cdots & 0 & 0 & -\frac{1}{4} & -\frac{1}{4} & 1 & 0 & -\frac{1}{4} \\ 0 & \cdots & 0 & 0 & 0 & -\frac{1}{4} & 0 & 1 & -\frac{1}{4} \\ 0 & \cdots & 0 & 0 & 0 & 0 & -\frac{1}{4} & -\frac{1}{4} & 1 \end{pmatrix} \tag{6.1}$$

Die Eigenwerte der Matrix sind durch

$$\lambda(i,j) = 1 - \frac{1}{2}\cos\frac{\pi i}{3} - \frac{1}{2}\cos\frac{\pi j}{c+1}$$

mit $1 \le i \le 2$, $1 \le j \le c$ und $n = 2c$ gegeben. Kenngrößen der Matrix in Abhängigkeit von der Ordnung n mit $n > 4$ sind in Tabelle 6.5 zusammengefaßt.

	LAPLACE
Ordnung	n
Nichtnull-Elemente	$4n - 4$
Besetzungsgrad in %	$100 \cdot (4n-4)/n^2$
$z_{\mathbf{max}}$	4
$z_{\mathbf{mean}}$	$4 - 4/n$
Konditionszahl	$(5 - 2\cos\frac{\pi c}{c+1})/(3 - 2\cos\frac{\pi}{c+1})$

Tabelle 6.5: Kenngrößen der Matrix **LAPLACE**

Bandbreitenreduktion

Die Reduktion der Bandbreite der Matrix kann den Kommunikationsaufwand für die parallele Matrix-Vektor-Multiplikation verringern. Bei allen beschriebenen

Methoden führt eine kleinere Bandbreite der Matrix u. U. zu kleineren Nachrichtenlängen oder sogar zu einem Datenaustausch mit weniger Prozessoren. Eine Diagonalmatrix erfordert keine Kommunikation. Je mehr das Besetzungsmuster der Matrix dem einer Diagonalmatrix ähnelt, desto günstiger ist das Kommunikationsverhalten der vorgestellten Verfahren zur parallelen Matrix-Vektor-Multiplikation.

In dieser Arbeit wird das Verfahren *Reverse Cuthill-McKee* (RCM) zur Umordnung der Matrix verwendet [125]. Die Permutation der Zeilen und Spalten der Matrix nach diesem Verfahren ist eine Ähnlichkeitstransformation; die Eigenwerte der Matrix ändern sich nicht. In FE-Modellen wird das RCM-Schema häufig zur Assemblierung der Koeffizientenmatrix genutzt. Die Numerierung der Gitterpunkte nach der RCM-Methode führt meist zu einer Matrix mit minimaler Bandbreite. Wenn sich das Diskretisierungsgitter nicht ändert, muß das Verfahren in einem FE-Modell lediglich einmal durchgeführt werden. Die Vorteile der Umordnung hingegen können z. B. in jedem Zeitschritt eines zeitabhängigen Problems genutzt werden.

Abbildung 6.1 zeigt die Besetzungsmuster der Matrizen **ICG2D**, **ICG3D** und **ISR** mit und ohne Bandbreitenreduktion nach dem RCM-Verfahren.

Die Matrizen **ICG2D** und **ICG3D** besitzen eine regelmäßige Besetzungsstruktur, während die Matrix **ISR** deutlich unregelmäßiger besetzt ist. Bei den Matrizen **ICG2D** und **ICG3D** wird die Bandbreite durch die RCM-Umordnung beträchtlich reduziert. Die Bandbreite der Matrix **ISR** hingegen ändert sich nur geringfügig. Bei allen drei Matrizen ändert sich das Besetzungsmuster durch das RCM-Verfahren deutlich. In Tabelle 6.6 sind die Bandbreiten der drei Matrizen vor und nach der RCM-Umordnung angegeben. Im folgenden werden die Matrizen mit Bandbreitenreduktion durch **ICG2D-RCM**, **ICG3D-RCM** und **ISR-RCM** bezeichnet.

	Bandbreite	
Matrix	Ohne RCM	Mit RCM
ICG2D	9020	106
ICG3D	2375	1303
ISR	3474	2989

Tabelle 6.6: Brandbreiten ohne und mit RCM-Umordnung

6.1.2 Unsymmetrische Matrizen

Zur Untersuchung der Verfahren QMR und TFQMR werden die regulären unsymmetrischen Matrizen **WATT1** und **ORSREG1** aus der Harwell-Boeing Sparse Matrix Collection verwendet. Die Matrix **ORSREG1** stammt aus einem dreidimensionalen Diskretisierungsproblem der Erdölförderung, die Matrix **WATT1**

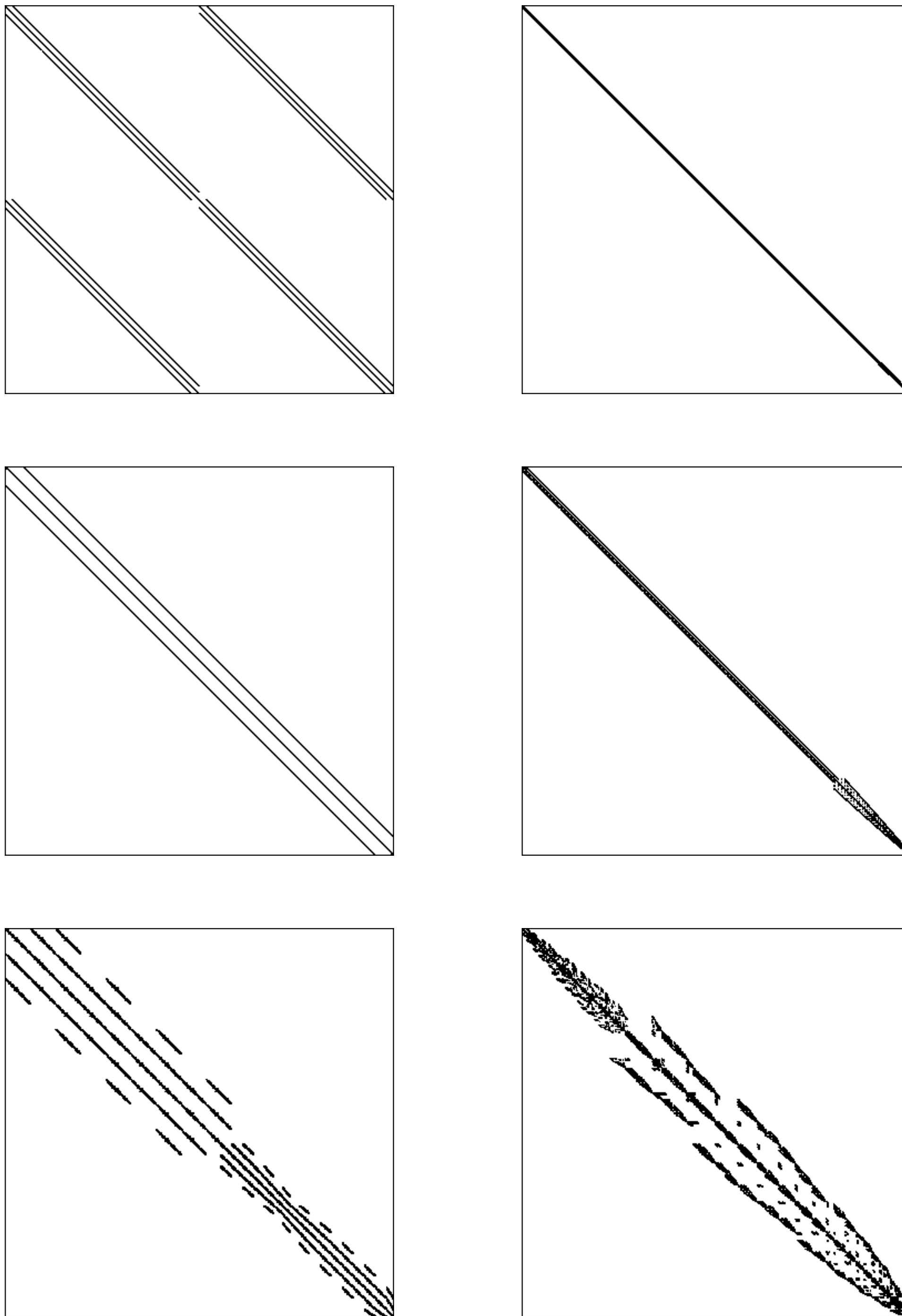


Abbildung 6.1: Links: Besetzungsmuster der Matrizen **ICG2D** (oben), **ICG3D** (Mitte) und **ISR** (unten). Rechts: die gleichen Matrizen mit Bandbreitenreduktion.

aus dem Bereich der Petrolchemie. Kenngrößen beider Matrizen sind in Tabelle 6.7 zusammengefaßt.

	ORSREG1	**WATT1**
Ordnung	2205	1856
Nichtnull-Elemente	14133	11360
Besetzungsgrad	0.3%	0.3%
z_{max}	7	7
z_{mean}	6.4	6.1

Tabelle 6.7: Kenngrößen der Matrizen **ORSREG1** und **WATT1**

Die Harwell-Boeing Sparse Matrix Collection enthält keine großen unsymmetrischen Matrizen. Daher werden für Performance-Untersuchungen der Verfahren QMR und TFQMR neben den Matrizen **ORSREG1** und **WATT1** die großen symmetrischen Matrizen **ICG3D** und **ISR** verwendet.

6.2 Matrix-Vektor-Multiplikation

Die parallele Matrix-Vektor-Multiplikation bildet den Kern der entwickelten iterativen Verfahren. Im folgenden werden das Zeitverhalten und die Skalierungseigenschaften der vorgestellten Methoden zur Matrix-Vektor-Multiplikation mit dünnbesetzter Matrix auf einem Parallelrechner mit verteiltem Speicher beschrieben.

6.2.1 Speichertechniken

Die Ausführungszeit der Matrix-Vektor-Multiplikation auf einem Rechnersystem hängt vom Speicherschema der dünnbesetzten Matrix ab. In den folgenden Untersuchungen wird die Speicherung der Matrix im ITPACK-, im CRS- und im SICRS-Format betrachtet.

In 3.3.3 sind die spalten- und zeilenweise Variante der Matrix-Vektor-Multiplikation bei Verwendung des ITPACK-Schemas beschrieben. Abbildung 6.2 zeigt die Ausführungszeiten der spalten- und zeilenweisen Matrix-Vektor-Multiplikation auf einem PARAGON-Prozessor für die Matrizen **ICG2D** und **BCSSTK18**. Das spaltenweise Verfahren entspricht der Methode aus Abbildung 3.21; der zeilenweise Algorithmus, der pro Zeile nur die Nichtnull-Elemente berücksichtigt, ist in Abbildung 3.22 dargestellt. Das mittlere, zeilenweise Verfahren in Abbildung 6.2 unterscheidet sich von der letzteren Methode darin, daß pro Zeile auch gespeicherte Null-Elemente berücksichtigt werden. In der inneren Schleife in Abbildung 3.22 ist `num_el`(i) durch z_{max} ersetzt.

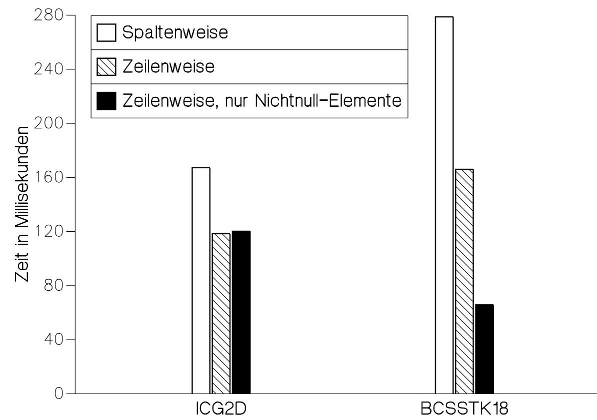


Abbildung 6.2: ITPACK-Schema: zeilen- und spaltenweise Matrix-Vektor-Multiplikation, 1 Prozessor

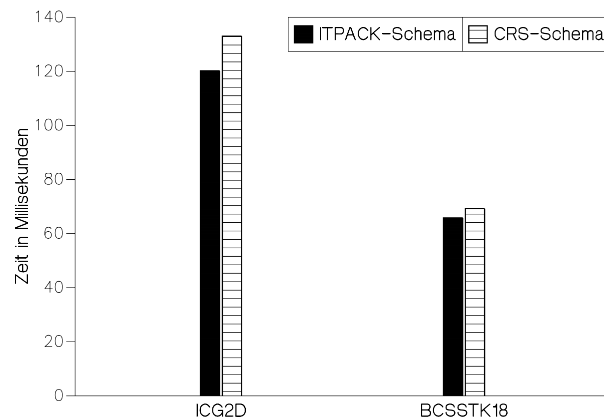


Abbildung 6.3: Vergleich der Algorithmen zur Matrix-Vektor-Multiplikation im ITPACK- und CRS-Format, 1 Prozessor

Die Ausführungszeiten der zeilenweisen Varianten in Abbildung 6.2 sind gegenüber denen der spaltenweisen Methode beträchtlich reduziert. Beim zeilenweisen Verfahren mit Berücksichtigung gespeicherter Null-Elemente führt die deutlich kleinere Länge der Vektoren in der inneren Schleife im Vergleich zur spaltenweisen Variante zu höherer Performance. Auf dem RISC-Prozessor i860 XP mit Cache-Speicher des PARAGON sind bei indirekter Adressierung kurze Vektoren günstiger als lange Vektoren. Auf Vektorrechnern hingegen sind auch bei indirekter Adressierung gewöhnlich lange Vektoren von Vorteil. Für die Matrix **ICG2D** ergab eine Vergleichsmessung auf einem Prozessor der CRAY Y-MP8/864 der Forschungszentrum Jülich GmbH eine um den Faktor 8.3 schnellere Ausführungszeit der spaltenweisen Methode gegenüber der Zeit der zeilenweisen Varianten. Die Messungen weisen darauf hin, daß auf RISC-Prozessoren mit kleinem Cache die zeilenweisen Methoden vorteilhaft sind, während auf Vektorrechnern das spaltenweise Verfahren geeignet ist. Die zeilenweise Variante, die pro Zeile nur die Nichtnull-Elemente berücksichtigt, spart gegenüber beiden anderen Verfahren gewöhnlich Operationen. Für die Matrix **ICG2D** ergeben sich bei beiden zeilenweisen Methoden ungefähr gleiche Ausführungszeiten, da die mittlere Anzahl der Nichtnull-Elemente nahe an der maximalen Anzahl liegt (s. Tabelle 6.1). Bei der Matrix **BCSSTK18** unterscheiden sich die Werte für z_{max} und z_{mean} in Tabelle 6.4 beträchtlich. Daher führt das zeilenweise Verfahren, das nur die Nichtnull-Elemente in die Rechnung einbezieht, zur deutlich kürzesten Ausführungszeit.

In Abbildung 6.3 werden die Ausführungszeiten der Matrix-Vektor-Multiplikation bei Speicherung der Matrix im ITPACK- und im CRS-Schema für die Matrizen **ICG2D** und **BCSSTK18** auf einem PARAGON-Prozessor verglichen. Die Matrix-Vektor-Multiplikation im ITPACK-Format wird zeilenweise ohne Berücksichtigung gespeicherter Null-Elemente durchgeführt; zur Berechnung dieser Operation im CRS-Format wird die zeilenweise Methode aus Abbildung 3.18 verwendet.

Die Ausführungszeiten der Verfahren unterscheiden sich bei beiden Matrizen kaum. Die Zeiten bei Verwendung des CRS-Schemas sind geringfügig höher, da zur Adressierung ein drittes Feld erforderlich ist (s. Abbildung 3.18). Der Speicherbedarf beider Formate ist für die Matrix **BCSSTK18** deutlich verschieden. Unter der Voraussetzung, daß zur Speicherung eines Werts vom Typ REAL 8 Bytes und eines Werts vom Typ INTEGER 4 Bytes notwendig sind, erfordert diese Matrix im ITPACK-Schema einen Speicherplatz von 6.7 Megabytes, im CRS-Schema dagegen von lediglich 1.8 Megabytes. Der Grund ist eine große Differenz der Werte für z_{max} und z_{mean}. Für die Matrix **ICG2D** unterscheidet sich der Speicherbedarf beider Schemata kaum. Im ITPACK-Format werden 3.6 Megabytes, im CRS-Format 3.5 Megabytes benötigt. z_{max} und z_{mean} sind bei dieser Matrix ungefähr gleich.

In Abbildung 6.4 sind die Ausführungszeiten für die Matrix-Vektor-Multiplikation im CRS- und im SICRS-Format für die Matrizen **BCSSTK18**, **PRESS**, **ICG3D** und **ISR** dargestellt. Die Zeiten für die Matrizen **BCSSTK18** und

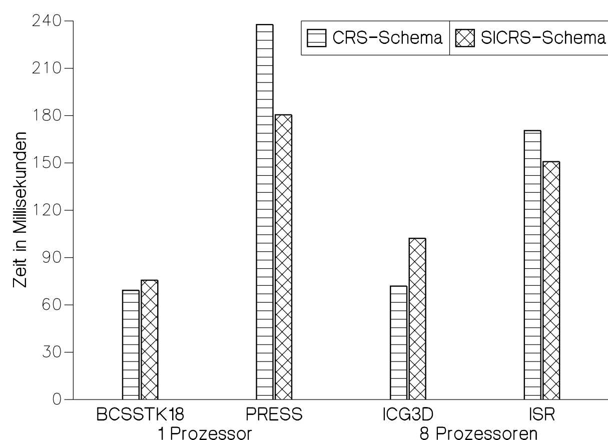


Abbildung 6.4: Vergleich der Algorithmen zur Matrix-Vektor-Multiplikation im CRS- und SICRS-Format

PRESS wurden auf einem Prozessor gemessen, die Zeiten für die Matrizen **ICG3D** und **ISR** auf acht Prozessoren. Für die Matrix-Vektor-Multiplikation im SICRS-Format wird der Algorithmus aus Abbildung 3.20 verwendet.

Für die Matrizen **PRESS** und **ISR** ergeben sich durch Verwendung des SICRS-Schemas gegenüber der Matrix-Vektor-Multiplikation im CRS-Schema um 24% bzw. um 12% reduzierte Ausführungszeiten. Der Grund ist eine hohe Anzahl von Nichtnull-Elementen pro Zeile mit aufeinanderfolgenden Spaltenindizes bei beiden Matrizen, so daß sich das Einsparen von Operationen mit indirekter Adressierung in der innersten Schleife des Algorithmus aus Abbildung 3.20 günstig auswirkt. Der Wert für die mittlere Anzahl von Nichtnull-Elementen mit aufeinanderfolgenden Spaltenindizes pro Element und Zeile beträgt $s_{\mathbf{mean},1} = 15.7$ für die Matrix **PRESS** und $s_{\mathbf{mean},8} = 10.2$ für die Matrix **ISR**. Der Index bezeichnet die Anzahl der Prozessoren, auf denen die Werte für $s_{\mathbf{mean}}$ bestimmt wurden. Durch die Umordnung der Matrixdaten in Blöcke (s. 5.4) kann sich der Wert für $s_{\mathbf{mean}}$ mit zunehmender Prozessorzahl verringern. Gewöhnlich sind die Unterschiede jedoch gering, da den Prozessoren aufeinanderfolgende Komponenten des Vektors der Matrix-Vektor-Multiplikation zugeteilt sind.

Für die Matrizen **BCSSTK18** und **ICG3D** sind die Ausführungszeiten der Matrix-Vektor-Multiplikation im CRS-Format um 9% bzw. um 30% kürzer als die der Operation im SICRS-Format. Die Werte für $s_{\mathbf{mean}}$ sind niedrig; sie betragen $s_{\mathbf{mean},1} = 1.2$ für die Matrix **BCSSTK18** und $s_{\mathbf{mean},8} = 2.9$ für die Matrix **ICG3D**. Bei diesen Matrizen werden bei Verwendung des SICRS-Schemas verglichen mit der Matrix-Vektor-Multiplikation im CRS-Format lediglich wenige Operationen mit indirekter Adressierung eingespart. Der zusätzliche Aufwand

zur Adressierung der Nichtnull-Elemente mit aufeinanderfolgenden Spaltenindizes pro Zeile im SICRS-Format überwiegt.

Bei niedrigen Werten von s_{mean} führt die Verwendung des SICRS-Schemas gegenüber der Speicherung im CRS-Schema zu höheren Ausführungszeiten, während sich bei hohen Werten ein günstigeres Zeitverhalten ergibt. Insbesondere kann das SICRS-Format auch für die zeilenweise Matrix-Vektor-Multiplikation mit vollbesetzter Matrix verwendet werden. Der zusätzliche Speicherbedarf gegenüber der üblichen Speicherung vollbesetzter Matrizen in einem zweidimensionalen Feld sind $3n + 1$ Werte von Typ INTEGER (s. auch 3.2.1). Bei großen Matrizen ist eine in etwa gleiche Performance wie bei Verwendung des zeilenweisen Algorithmus für vollbesetzte Matrizen zu erwarten. Das SICRS-Schema ist damit eine Speichertechnik, die sich sowohl für dünnbesetzte als auch für vollbesetzte Matrizen eignet.

In Tabelle 6.8 sind der Speicherbedarf der betrachteten Matrizen in den Formaten ITPACK, CRS und SICRS sowie die Werte für s_{mean} zusammengefaßt. Die Zahl in Klammern hinter einem Wert von s_{mean} gibt an, auf wievielen Prozessoren der Wert bestimmt wurde.

	Speicherbedarf in Megabytes			
	ITPACK	CRS	SICRS	s_{mean}
ICG2D	3.6	3.5	3.2	3.0 (1)
ICG2D-RCM	3.6	3.5	2.8	5.8 (1)
ICG3D	15.3	14.4	12.9	2.9 (4)
ICG3D-RCM	15.3	14.4	13.3	2.6 (4)
ISR	140.0	44.2	32.4	10.2 (8)
ISR-RCM	140.0	44.2	34.9	5.5 (8)
PRESS	7.8	7.6	5.4	15.7 (1)
BCSSTK17	18.8	4.9	4.0	5.0 (1)
BCSSTK18	6.7	1.8	2.1	1.2 (1)
LAPLACE	$48n/2^{20}$	$(52n-44)/2^{20}$	$(52n-44)/2^{20}$	2

Tabelle 6.8: Speicherbedarf bei verschiedenen Formaten der Matrizen

Für $s_{\mathrm{mean}} > 2$ ist der Speicherbedarf bei Verwendung des SICRS-Formats gegenüber der Speicherung der Matrix im CRS-Schema reduziert. Deutliche Einsparungen ergeben sich bei den Matrizen **ISR** und **PRESS**. Lediglich die Matrix **BCSSTK18** mit $s_{\mathrm{mean},1} = 1.2$ erfordert bei Speicherung im SICRS-Format mehr Speicherplatz als bei Verwendung des CRS-Schemas. Ferner ist Tabelle 6.8 zu entnehmen, daß die Permutation der Matrix zur Bandbreitenminimierung Einfluß auf den Wert von s_{mean} hat. Bei den Matrizen **ICG3D** und **ISR** sinkt der Wert durch die RCM-Umordnung, während s_{mean} bei der Matrix **ICG2D** ansteigt.

In allen folgenden Untersuchungen wird von der Speicherung der dünnbesetzten Matrix im CRS-Schema ausgegangen, da der Speicherbedarf gering und

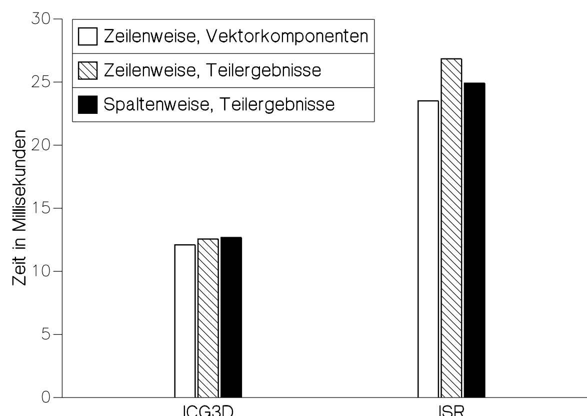


Abbildung 6.5: Ausführungszeiten der Matrix-Vektor-Multiplikation bei verschiedenen Kommunikationsschemata, 64 Prozessoren

das Zeitverhalten der Matrix-Vektor-Multiplikation für alle betrachteten Formate ähnlich ist. Das CRS-Schema wird in vielen FE-Modellen verwendet. Sämtliche Ergebnisse sind auf die beiden anderen betrachteten Formate ITPACK und SICRS übertragbar. Programme und Daten der untersuchten iterativen Methoden erfordern für die Matrizen **LAPLACE** mit $n = 100000$, **ICG3D**, **ISR** und **LAPLACE** mit $n = 1000000$ den Speicherplatz von mehr als einem, zwei, vier bzw. acht Prozessoren. Die Messungen für diese Matrizen wurden auf mindestens zwei, vier, acht bzw. 16 Prozessoren durchgeführt. Bis zu dieser Anzahl von Prozessoren wurde jeweils linearer Speedup angenommen.

6.2.2 Kommunikationsschemata

In 5.4 sind verschiedene Kommunikationsschemata zur parallelen zeilen- und spaltenweisen Matrix-Vektor-Multiplikation beschrieben. Abbildung 6.5 zeigt die Ausführungszeiten der drei vorgestellten Methoden auf 64 Prozessoren für die Matrizen **ICG3D** und **ISR**.

Für die Matrix **ICG3D** ergeben alle drei Schemata in etwa gleiche Ausführungszeiten, da die Matrix regelmäßig besetzt ist. Im Fall der Matrix **ISR** ist die Ausführungszeit der zweiten Methode deutlich gegenüber dem ersten Verfahren erhöht. Bei dem unregelmäßigen Besetzungsmuster dieser Matrix führt die Neuverteilung der Daten-Blöcke im zweiten Schema zu einer ungleichmäßigen Auslastung der Prozessoren. Ferner ist der Grad der Überlappung von lokalen Berechnungen und Kommunikation gegenüber der ersten Methode geringer. Der Anstieg der Ausführungszeit beim dritten, spaltenweisen Verfahren verglichen

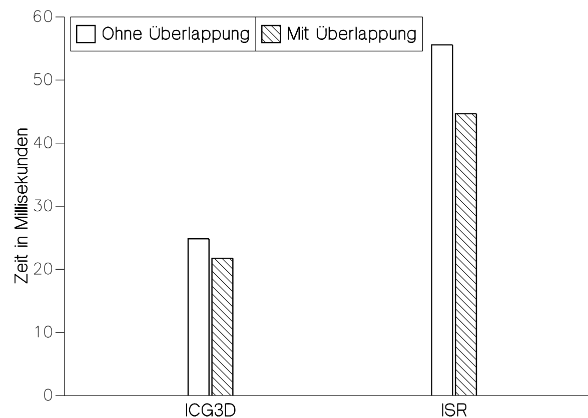


Abbildung 6.6: Einfluß der Überlappung von Rechnung und Kommunikation, 32 Prozessoren

mit der Zeit der ersten Methode ist deutlich kleiner. Hier wirkt sich lediglich der geringere Grad der Überlappung von lokalen Berechnungen und Kommunikation gegenüber dem ersten Schema auf das Zeitverhalten aus; die gleichmäßige Verteilung der Rechenlast auf die Prozessoren wird nicht beeinflußt. In allen folgenden Untersuchungen wird das erste, zeilenweise Kommunikationsschema ohne Neuverteilung der Matrix-Daten verwendet, da sich für die betrachteten Matrizen die kürzesten Ausführungszeiten ergeben.

6.2.3 Überlappung

In Abbildung 6.6 sind die Ausführungszeiten der parallelen zeilenweisen Matrix-Vektor-Multiplikation ohne und mit Überlappung von Rechnung und Kommunikation dargestellt. Durch die Überlappung wird die Ausführungszeit der Operation für die Matrix **ICG3D** um 12% reduziert; für die Matrix **ISR** ergibt sich eine um 20% verringerte Ausführungszeit. Der größere Einfluß der Überlappung bei der Matrix **ISR** ist durch die deutlich höhere Anzahl der Nichtnull-Elemente gegenüber der Matrix **ICG3D** zu erklären. Während sich angeforderte Daten auf dem Netzwerk befinden, können daher beträchtlich mehr lokale Operationen durchgeführt werden.

6.2.4 Bandbreitenreduktion

Abbildung 6.7 zeigt die Speedup-Werte der parallelen Matrix-Vektor-Multiplikation ohne und mit Bandbreitenreduktion nach dem RCM-Verfahren für die

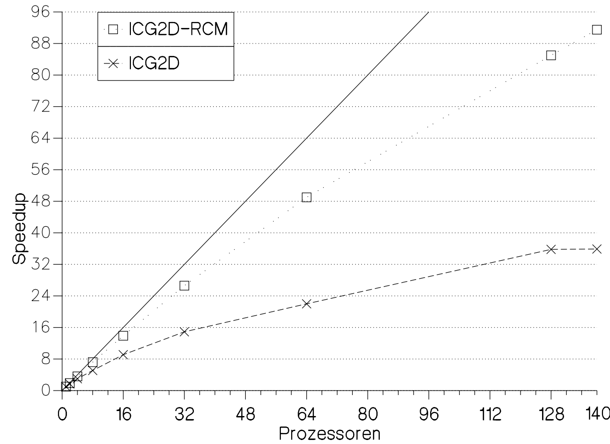


Abbildung 6.7: Speedup-Werte ohne und mit Bandbreitenreduktion, Matrix **ICG2D**

Matrix **ICG2D** auf bis zu 140 Prozessoren. Die durchgezogene Linie begrenzt den maximal erreichbaren Speedup.

Die Speedup-Werte sind für die Matrix mit reduzierter Bandbreite signifikant höher als für die ursprüngliche Matrix. Da die Bandbreite um den Faktor 85 verringert wird (s. Tabelle 6.6) und sich zusätzlich der Wert für $s_{\mathbf{mean}}$ fast verdoppelt (s. Tabelle 6.8), ist für die parallele Matrix-Vektor-Multiplikation deutlich weniger Kommunikation erforderlich. Auf 140 Prozessoren wird mit Bandbreitenreduktion ein Speedup von 91.5 bei einer Ausführungszeit von 1.38 ms erreicht, während ohne die RCM-Permutation der Speedup lediglich 35.9 bei einer Ausführungszeit von 3.70 ms beträgt.

In Abbildung 6.8 sind die entsprechenden Speedup-Werte für die Matrizen **ICG3D** und **ISR** dargestellt.

Für die Matrix **ISR** unterscheiden sich die Speedup-Werte ohne und mit Bandbreitenreduktion kaum, da die RCM-Umordnung lediglich zu einer Verringerung der Bandbreite um 14% führt. Für die Matrix **ICG3D-RCM** sind die Speedup-Werte bis 64 Prozessoren höher als für die ursprüngliche Matrix, während sie bei 128 und 140 Prozessoren unter den erreichten Werten für die Matrix **ICG3D** liegen. Der Grund für die zunächst höheren Werte ist die Reduktion der Bandbreite um 45%. Zusätzlich hat sich durch die Bandbreitenreduktion die Anzahl der Elemente mit aufeinanderfolgenden Spaltenindizes geändert. Die kleineren Werte von $s_{\mathbf{mean}}$ in Tabelle 6.8 für die Matrizen **ICG3D-RCM** und **ISR-RCM** weisen darauf hin. Dies kann gerade bei einer großen Anzahl von Prozessoren — jedem Prozessor ist lediglich ein kleines Segment des Vektors zugeordnet — zu einem ungünstigeren Kommunikationsverhalten führen. Ein

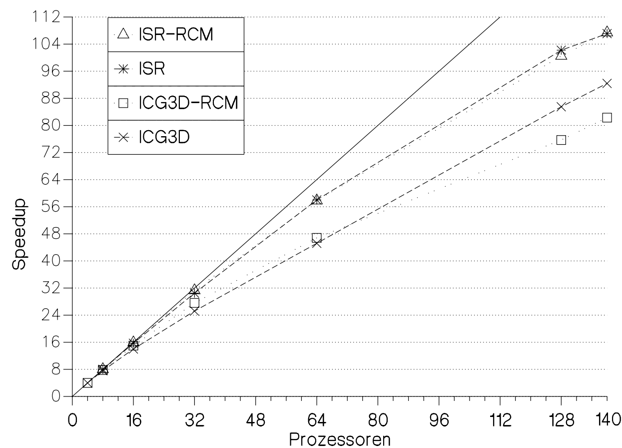


Abbildung 6.8: Speedup-Werte ohne und mit Bandbreitenreduktion, Matrizen **ICG3D** und **ISR**

erhöhter Datenaustausch wirkt sich bei einer hohen Prozessorzahl deutlich auf das Speedup-Verhalten aus, da der Rechenaufwand pro Prozessor mit steigender Anzahl der eingesetzten Prozessoren sinkt. Auf die Skalierungseigenschaften der parallelen Matrix-Vektor-Multiplikation bei unterschiedlichen Problemgrößen wird unten eingegangen.

6.2.5 Skalierungseigenschaften

In Abbildung 6.9 ist das Speedup-Verhalten verschiedener Matrizen aus technischen Anwendungen bei von unten nach oben steigender Problemgröße veranschaulicht. Die entscheidende Problemgröße dünnbesetzter Matrizen in iterativen Verfahren ist nicht die Ordnung n der Matrix, sondern die Anzahl der Nichtnull-Elemente e, da nur die Nichtnull-Elemente in die Rechnung eingehen. Daher liegt bei der Matrix **ISR** die höchste Problemgröße vor (vergl. 6.1).

Bei hoher Prozessorzahl ist der Anstieg der Speedup-Werte mit wachsender Problemgröße deutlich zu erkennen. Für die Matrizen **BCSSTK18**, **ICG2D**, **BCSSTK17**, **PRESS**, **ICG3D** und **ISR** — die Anzahl der Nichtnull-Elemente der Matrizen steigt in dieser Reihenfolge — werden auf 140 Prozessoren Speedup-Werte von 23.7, 35.9, 55.5, 86.1, 92.4 bzw. 107.1 erreicht; die Ausführungszeiten betragen 2.92 ms, 3.70 ms, 2.87 ms, 3.30 ms, 5.92 ms bzw. 12.73 ms. Zu beachten ist, daß sich dieses Speedup-Verhalten für Matrizen mit unterschiedlicher Besetzungsstruktur ergibt. Die entwickelten Methoden zur parallelen Matrix-Vektor-Multiplikation lassen damit eine gute Skalierung für Matrizen aus verschiedensten Anwendungsbereichen erwarten.

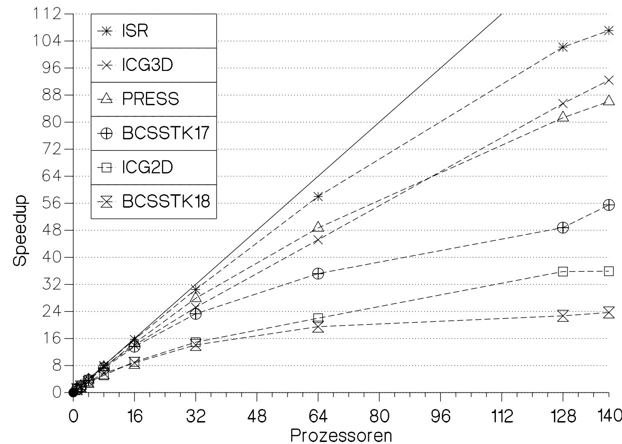


Abbildung 6.9: Speedup-Verhalten der Matrix-Vektor-Multiplikation für Matrizen aus technischen Anwendungen mit steigender Problemgröße

Abbildung 6.10 zeigt das Speedup-Verhalten für die Matrix **LAPLACE** mit n = 1000, 10000, 100000 und 1000000. Diese Matrizen besitzen die gleiche Besetzungsstruktur; die Problemgröße kann über die Ordnung vorgegeben werden (s. Tabelle 6.5).

Die Speedup-Werte der Matrizen steigen mit der Problemgröße. Bei gleicher Prozessorzahl sinkt der Anteil der Kommunikation am Gesamtaufwand der Matrix-Vektor-Multiplikation mit zunehmender Problemgröße, während sich der Anteil der Rechenoperationen erhöht. Für die größte Matrix wird annähernd linearer Speedup bis 140 Prozessoren erreicht. Bei 140 Prozessoren beträgt der Speedup 137.4 bei einer Ausführungszeit von 24.39 ms. Das Speedup-Verhalten in Abbildung 6.10 weist auf eine hohe Effizienz der entwickelten Verfahren selbst bei großer Prozessorzahl hin, wenn die Problemgröße hoch ist.

6.3 Iterative Methoden

Im folgenden werden numerische und Performance-Ergebnisse paralleler iterativer Methoden zur Lösung von Gleichungssystemen und Eigenwertproblemen vorgestellt. Zunächst wird auf CG-Verfahren mit Vorkonditionierung eingegangen; anschließend werden parallele Varianten der Verfahren QMR und TFQMR beschrieben. Zum Schluß werden Ergebnisse einer parallelen Lanczos-Methode zur Lösung des symmetrischen Eigenwertproblems präsentiert.

Speedup-Werte, die zu den iterativen Verfahren angegeben werden, sind auf einen Iterationsschritt der jeweiligen Methode bezogen, da jeder Schritt parallel

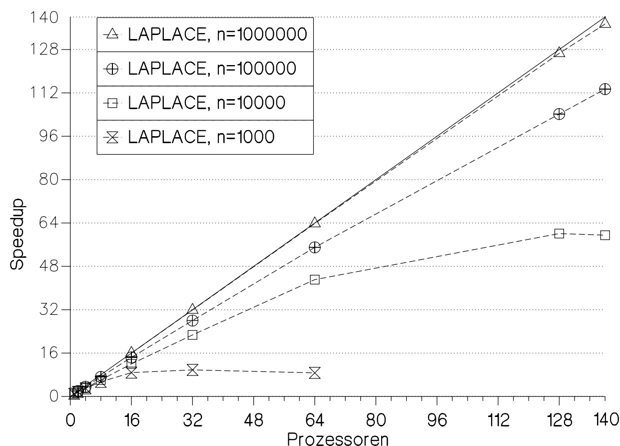


Abbildung 6.10: Speedup-Verhalten der Matrix-Vektor-Multiplikation für die Matrizen **LAPLACE**

durchgeführt wird. Bei einigen Matrizen schwankt die Anzahl der Iterationsschritte der Verfahren aus numerischen Gründen geringfügig, wenn die Anzahl der verwendeten Prozessoren variiert wird. Hierbei wurde sowohl Ansteigen als auch Sinken der Anzahl der Schritte beobachtet. Diese Schwankungen gehen bei der obigen Vorgehensweise nicht in die Speedup-Berechnung ein.

6.3.1 Startwerte

Die in Kapitel 2 beschriebenen iterativen Methoden benötigen zu Beginn der Iteration geeignete Startwerte. Sind Startwerte durch die Problemstellung vorgegeben — z. B. durch das FE-Modell —, so werden diese Werte verwendet.

Ist dies nicht der Fall, so wird der Startvektor $x^{(0)}$ der CG-Verfahren durch

$$x_j^{(0)} = \frac{b_j}{a_{jj}}, \quad j = 1, \ldots, n,$$

berechnet mit $a_{jj} \neq 0$, $j = 1, \ldots, n$, als den Diagonalelementen der Koeffizientenmatrix des Gleichungssystems $\boldsymbol{A}x = b$. Für die Verfahren QMR und TFQMR wird in diesem Fall als Startvektor $x^{(0)}$ der Null-Vektor gewählt.

Die QMR-Methode erfordert zusätzlich die Wahl eines Vektors $\tilde{w}^{(1)}$. Dieser Vektor wird wie in Algorithmus 2.5 vorgeschlagen zu $-g^{(0)}$ gesetzt. Im TFQMR-Verfahren wird ein beliebiger Vektor $\tilde{g}^{(0)}$ benötigt, der $\rho_0 = -\tilde{g}^{(0)^T} g^{(0)} \neq 0$ erfüllt (s. Algorithmus 2.8). Eine sinnvolle Wahl für diesen Vektor ist $g^{(0)}$. In diesem Fall ist die obige Bedingung erfüllt, außer das Start-Residuum $g^{(0)}$ hat den

Wert 0. Letzteres bedeutet, daß der Startvektor der exakten Lösung entspricht; eine Iteration ist nicht erforderlich.

Bei einigen Matrizen, die in Untersuchungen iterativer Verfahren zur Lösung von Gleichungssystemen $\boldsymbol{A}x = b$ verwendet werden, ist keine rechte Seite b vorgegeben. Dies trifft für die Matrix **LAPLACE** und die Matrizen aus der Harwell-Boeing Sparse Matrix Collection zu. In diesem Fall werden die Komponenten der rechten Seite zu

$$b_j = \sum_{i=1}^{n} a_{ji}, \quad j = 1, \ldots, n,$$

gesetzt, so daß die exakte Lösung des Systems der Vektor $x^* = (1, \ldots, 1)^T$ ist.

Für die zweite Variante der Lanczos-Tridiagonalisierung (s. Algorithmus 2.11) wird als Startvektor $r^{(0)}$

$$r_j^{(0)} = \frac{j}{n}, \quad j = 1, \ldots, n,$$

gewählt. Ein entsprechender Startvektor $q^{(1)}$ für Algorithmus 2.10, die erste Variante der Lanczos-Tridiagonalisierung, wird durch $q^{(1)} = r^{(0)}/({r^{(0)}}^T r^{(0)})^{1/2}$ berechnet. Für die Komponenten dieses Vektors gilt:

$$q_j^{(1)} = \frac{\sqrt{6}j}{\sqrt{n(n+1)(2n+1)}}, \quad j = 1, \ldots, n.$$

Alle Startvektoren können parallel ermittelt werden und sind unabhängig von der Anzahl der verwendeten Prozessoren.

6.3.2 Das CG-Verfahren

Parallelisierungseigenschaften des CG-Verfahrens wurden ohne und mit Vorkonditionierung der Methode untersucht. Zur Vorkonditionierung wurden die Skalierung mit der Diagonalen und die polynomiale Vorkonditionierung verwendet. In allen Untersuchungen wurde die Iteration abgebrochen, wenn das Kriterium (2.2) mit $\epsilon_s = 10^{-5}$ erfüllt war.

Numerische Resultate

In Tabelle 6.9 ist die Anzahl der Iterationsschritte des CG-Verfahrens ohne und mit Vorkonditionierung für die betrachteten symmetrischen positiv definiten Matrizen angegeben. Für die polynomiale Vorkonditionierung ist zusätzlich der Polynomgrad m aufgeführt. Die polynomiale Vorkonditionierung wurde in allen Untersuchungen zusammen mit der Skalierung unter Verwendung der Diagonalen durchgeführt, d. h. die Methode wurde auf das skalierte Gleichungssystem angewendet. Die Matrix **LAPLACE** wurde mit der Ordnung $n = 1000$, 10000, 100000 und 1000000 verwendet.

	Vorkonditionierung			
	Keine	Skalierung	Polynomiale	m
ICG2D	157	110	49	2
ICG3D	390	84	39	2
ISR	1444	658	266	2
PRESS	49273	33406	4436	8
BCSSTK17	22333	3007	667	4
BCSSTK18	-	2034	465	4
LAPLACE	14	14	11	2

Tabelle 6.9: Anzahl der Iterationsschritte des CG-Verfahrens ohne und mit Vorkonditionierung

Mit Vorkonditionierung sind deutlich weniger Iterationsschritte zum Erreichen des Abruchkriteriums erforderlich als beim unvorkonditionierten CG-Verfahren. Bereits die Skalierung mit der Diagonalen reduziert die Anzahl der Schritte beträchtlich. Durch polynomiale Vorkonditionierung wird die Iterationszahl abhängig vom Grad des Polynoms weiter verringert. Als Polynomgrad sind die Werte von m gewählt, die zu den kürzesten Ausführungszeiten des Verfahrens auf 32 Prozessoren führen. Zu diesem Zweck wurden Polynomgrade von 2 bis 10 untersucht. Gewöhnlich ist der Polynomgrad 2 eine geeignete Wahl. Das Zeitverhalten des CG-Verfahrens ohne und mit Vorkonditionierung wird unten beschrieben.

Tabelle 6.9 ist ferner zu entnehmen, daß die Anzahl der Iterationsschritte mit der Konditionszahl der Matrix steigt (vergl. 2.2.2). Zusätzlich können Rundungsfehler die Konvergenz beeinflussen. Bei den Matrizen **PRESS** und **BCSSTK17** liegen die Iterationszahlen aufgrund von Rundungsfehlern oberhalb der theoretischen Grenze n. Eine akzeptable Approximation der Lösung wird trotzdem erreicht. Bei der Matrix **BCSSTK18** erfüllt das CG-Verfahren ohne Vorkonditionierung innerhalb von 50000 Schritten das vorgegebene Abbruchkriterium nicht. Mit Vorkonditionierung wird jedoch eine Approximation der Lösung gefunden. Vorkonditionierung führt zu einem günstigeren Konvergenzverhalten. Das Verfahren wird numerisch stabiler; der Einfluß von Rundungsfehlern sinkt.

Für die Matrix **LAPLACE** ergeben sich bei allen gewählten Werten von n die gleichen Iterationszahlen; die Konditionszahl beträgt in allen Fällen ca. 7. Die Skalierung mit der Diagonalen reduziert die Anzahl der Schritte nicht, da die Diagonalelemente bereits den Wert 1 haben. Durch polynomiale Vorkonditionierung sinkt die Iterationszahl um drei Schritte. Da die Matrix **LAPLACE** bereits gut konditioniert ist, wirkt sich die Vorkonditionierung kaum auf die Konvergenz der CG-Iteration aus.

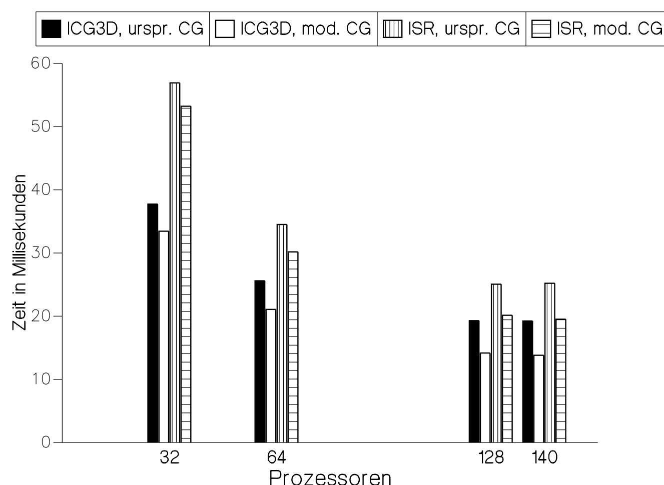


Abbildung 6.11: Ausführungszeiten des ursprünglichen und des modifizierten CG-Verfahrens

Vergleich der Varianten

In Abbildung 6.11 sind die Ausführungszeiten des ursprünglichen und des modifizierten CG-Verfahrens (s. die Algorithmen 2.1 und 2.2) auf 32 bis 140 Prozessoren für die Matrizen **ICG3D** und **ISR** dargestellt. Die Messungen wurden unter dem Betriebssystem PARAGON OSF/1, Release 1.1, durchgeführt.

Mit steigender Anzahl der Prozessoren verringern sich die Ausführungszeiten des modifizierten Verfahrens deutlich gegenüber denen der ursprünglichen Methode. Auf 140 Prozessoren wird die Ausführungszeit für die Matrix **ICG3D** um 28% reduziert; für die Matrix **ISR** ergibt sich ein Zeitgewinn von 23%. Der Grund ist die Einsparung globaler Synchronisation im modifizierten Verfahren gegenüber der ursprünglichen Methode (s. auch 2.2.1). Allen folgenden Untersuchungen liegt das modifizierte CG-Verfahren zugrunde.

Datenverteilung

In den Abbildungen 6.12 und 6.13 sind die Ausführungszeiten des CG-Verfahrens pro Iterationsschritt auf 64 Prozessoren bei unterschiedlicher Datenverteilung für die Matrizen **ICG3D** und **ISR** bzw. **BCSSTK17** und **BCSSTK18** veranschaulicht. Verglichen werden die Zeiten für die Werte $\xi = 10^{14}$, $\xi = 0$ und $\xi = 8$ des Parameters der Datenverteilung. Im ersten Fall werden jedem Prozessor möglichst gleich viele Zeilen zugewiesen, im zweiten sind auf jedem Prozessor ungefähr gleich viele Nichtnull-Elemente vorhanden. Bei $\xi = 8$ sind die Matrix- und Vektor-Daten derart auf die einzelnen Prozessoren verteilt, daß auf jedem Pro-

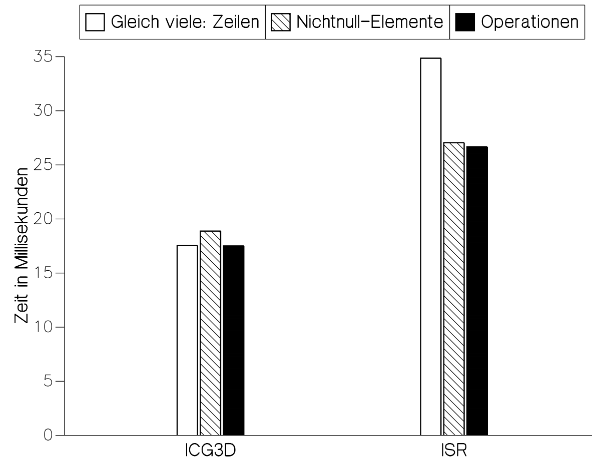


Abbildung 6.12: Ausführungszeiten des CG-Verfahrens pro Iterationsschritt für die Matrizen **ICG3D** und **ISR** bei unterschiedlicher Datenverteilung, 64 Prozessoren

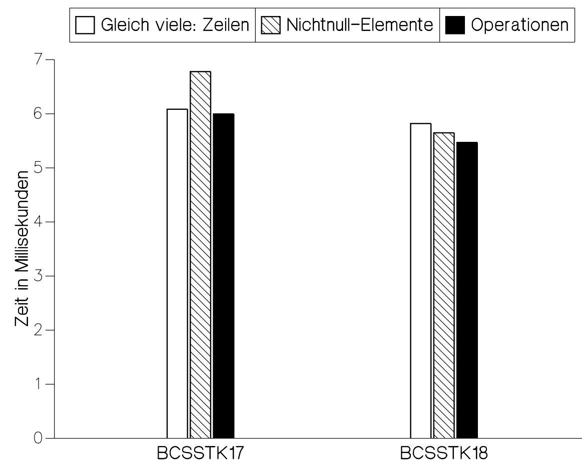


Abbildung 6.13: Ausführungszeiten des CG-Verfahrens pro Iterationsschritt für die Matrizen **BCSSTK17** und **BCSSTK18** bei unterschiedlicher Datenverteilung, 64 Prozessoren

zessor möglichst gleich viele Operationen der CG-Iteration durchgeführt werden (s. auch 5.3, Tabelle 5.1).

Für die regelmäßig strukturierte Matrix **ICG3D** in Abbildung 6.12 führen alle drei Datenverteilungen zu ungefähr gleichen Ausführungszeiten. Bei $\xi = 0$ sind die Prozessoren geringfügig ungleichmäßiger ausgelastet als bei den anderen beiden Parameterwerten. Deutlichere Unterschiede sind für die unregelmäßig besetzte Matrix **ISR** gegeben. Für diese Matrix sind die Ausführungszeiten bei $\xi = 0$ und $\xi = 8$ gegenüber der Zeit bei $\xi = 10^{14}$ um 23% reduziert. Die beiden ersteren Zeiten sind etwa gleich groß, da nach (5.3) der Anteil der Matrix-Vektor-Multiplikation am Rechenaufwand eines Iterationsschritts 95% beträgt. Bei beiden Matrizen führt die Datenverteilung mit $\xi = 8$ zur günstigsten Auslastung der Prozessoren.

Wie Abbildung 6.13 zeigt, ergibt sich für die Matrix **BCSSTK17** bei $\xi = 0$ die ungünstigste Auslastung der Prozessoren, während für die Matrix **BCSSTK18** die Ausführungszeit bei $\xi = 10^{14}$ am höchsten ist. Wieder führt die Datenverteilung mit $\xi = 8$ zum günstigsten Zeitverhalten. Für die Matrix **BCSSTK17** ist die Ausführungszeit bei $\xi = 8$ gegenüber der Zeit bei $\xi = 0$ um 12% reduziert; für die Matrix **BCSSTK18** ergibt sich im Vergleich zur Zeit bei $\xi = 10^{14}$ eine Reduzierung um 6%.

Durch Verwendung des für das CG-Verfahren charakteristischen Parameterwerts — dieser hängt zusätzlich von der Prozessorarchitektur ab —, wird für verschieden strukturierte Matrizen eine gleichmäßige Auslastung der Prozessoren erreicht. Insbesondere ist die Datenverteilung für den charakteristischen Wert günstiger, als jedem Prozessor gleich viele Zeilen oder Nichtnull-Elemente zuzuteilen. In allen folgenden Untersuchungen wird zur Ermittlung der Datenverteilung $\xi = 8$ verwendet.

Überlappung

Abbildung 6.14 zeigt den Einfluß der Überlappung von Rechnung und Kommunikation auf das Zeitverhalten des CG-Verfahrens für die Matrizen **ICG3D** und **ISR**. Die Ausführungszeiten pro Iterationsschritt wurden auf 32 Prozessoren des iPSC/860 und des PARAGON unter dem Betriebssystem OSF/1, Release 1.1 bzw. Release 1.2, gemessen. Auf dem iPSC/860 wurde der FORTRAN Compiler der Portland Group, Inc., Release 4.0, verwendet [2]; die Messungen wurden unter dem Betriebssystem NX/2 durchgeführt [3].

Auf dem iPSC/860 werden die Ausführungszeiten durch Überlappung von Rechnung und Kommunikation am deutlichsten beeinflußt. Bei beiden Matrizen sinken die Zeiten durch die Überlappung um ca. 20%. Auf dem PARAGON sind die Unterschiede geringer, da die Bandbreite des Verbindungsnetzwerks der Prozessoren um ein Vielfaches höher ist; die Transferzeiten sind bei gleicher Datenmenge deutlich kürzer als auf dem iPSC/860. Ferner sind die Ausführungszeiten auf den PARAGON-Prozessoren i860 XP durch die höhere Taktrate ge-

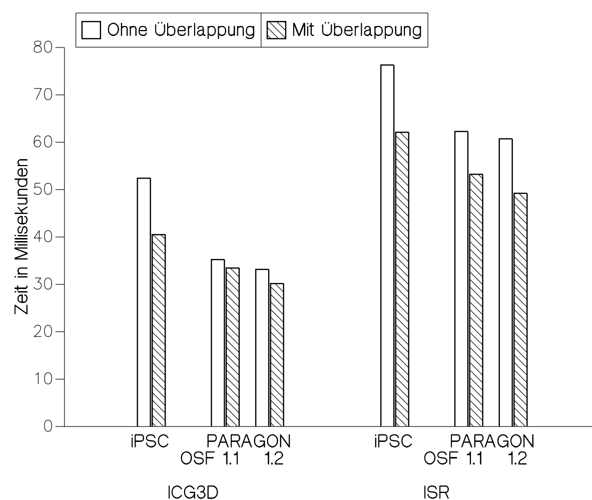


Abbildung 6.14: Einfluß der Überlappung auf das Zeitverhalten, 32 Prozessoren

genüber den iPSC/860-Prozessoren i860 XR reduziert. Unter dem Betriebssystem PARAGON OSF/1, Release 1.1, werden die Ausführungszeiten durch die Überlappung bei der Matrix **ICG3D** um 5% und bei der Matrix **ISR** um 15% verkürzt, während unter dem Betriebssystem PARAGON OSF/1, Release 1.2, eine Reduzierung um 9% bzw. 19% erreicht wird. Gegenüber dem Release 1.1 ist der Einfluß der Überlappung höher, da unter dem Release 1.2 der Kommunikationsprozessor die Abwicklung des Datenaustauschs übernimmt. Der Rechenprozessor ist von der Bearbeitung der Kommunikationsprotokolle freigestellt und kann unmittelbar nach Aufsetzen einer asynchronen Kommunikationsanweisung Operationen mit lokalen Daten ohne weitere Unterbrechungen durchführen. Die größere Auswirkung der Überlappung auf das Zeitverhalten bei der Matrix **ISR** kommt zustande, da bei dieser Matrix größere Datenmengen als bei der Matrix **ICG3D** ausgetauscht werden und damit die Transferzeiten nicht-lokaler Daten höher sind. Im Vergleich zur Matrix **ICG3D** ist die Matrix **ISR** dichter besetzt.

Skalierungseigenschaften

In Abbildung 6.15 sind Speedup-Werte des CG-Verfahrens auf bis zu 140 Prozessoren für Matrizen aus technischen Anwendungen dargestellt.

Bei hohen Prozessorzahlen steigen die Speedup-Werte deutlich mit der Anzahl der Nichtnull-Elemente der Matrizen. Speedup-Verluste kommen einerseits durch den Kommunikationsaufwand während der Matrix-Vektor-Multiplikation und andererseits durch globale Synchronisation zur Bestimmung von Skalarprodukten und Normen zustande. Jeder Iterationsschritt des modifizierten CG-Verfahrens

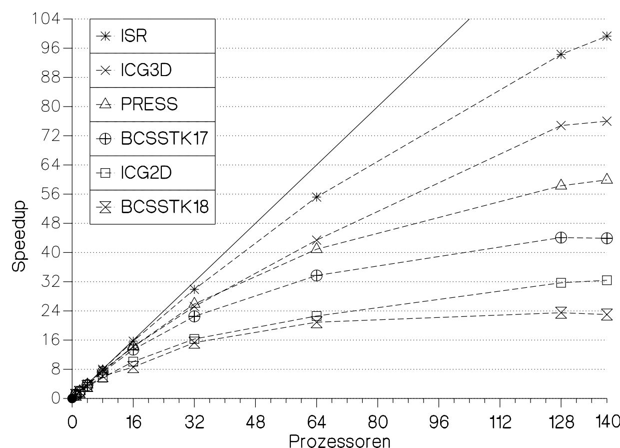


Abbildung 6.15: Speedup-Verhalten des CG-Verfahrens für Matrizen aus technischen Anwendungen mit steigender Problemgröße

enthält einen Synchronisationspunkt. Der zeitliche Aufwand für globale Synchronisation wächst mit der Anzahl der Prozessoren. Da mit steigender Prozessorzahl zusätzlich der Rechenaufwand sinkt, erhöht sich der Einfluß des Synchronisationsaufwands auf das Speedup-Verhalten. Daher liegen die Speedup-Werte in Abbildung 6.15 unter denen aus Abbildung 6.9 für die Matrix-Vektor-Multiplikation. Auf 140 Prozessoren werden für die Matrizen **BCSSTK18**, **ICG2D**, **BCSSTK17**, **PRESS**, **ICG3D** und **ISR** Speedup-Werte von 23.0, 32.4, 43.9, 59.9, 76.0 und 99.3 erreicht; die zugehörigen Ausführungszeiten bei Skalierung der Matrix betragen 9.9 s, 0.70 s, 13.8 s, 160.3 s, 0.84 s und 9.7 s.

Abbildung 6.16 veranschaulicht das Speedup-Verhalten des CG-Verfahrens für die Matrix **LAPLACE** mit $n = 1000$, 10000, 100000 und 1000000. Für die Matrizen der Ordnung $n = 1000$ und $n = 10000$ wurden Speedup-Werte bis 64 Prozessoren ermittelt, für die beiden größeren Matrizen bis 140 Prozessoren.

Da die Matrix-Vektor-Multiplikation der Matrizen aufgrund der Besetzungsstruktur nach (5.2) lediglich einen Anteil von ca. 33% am Rechenaufwand eines Iterationsschritts hat, ist der Einfluß des Synchronisationsaufwands auf das Speedup-Verhalten groß. Daher liegen die Speedup-Werte für $n = 1000$, 10000 und 100000 deutlich unter denen aus Abbildung 6.10 für die Matrix-Vektor-Multiplikation. Bei fester Prozessorzahl steigt der Rechenaufwand mit der Problemgröße, während der Synchronisationsaufwand unverändert bleibt. Bei $n = 1000000$ ist der Anteil des Synchronisationsaufwands an der Gesamtzeit bereits deutlich vermindert; auf 140 Prozessoren wird ein Speedup von 127.0 bei einer Ausführungszeit von 0.81 s erreicht.

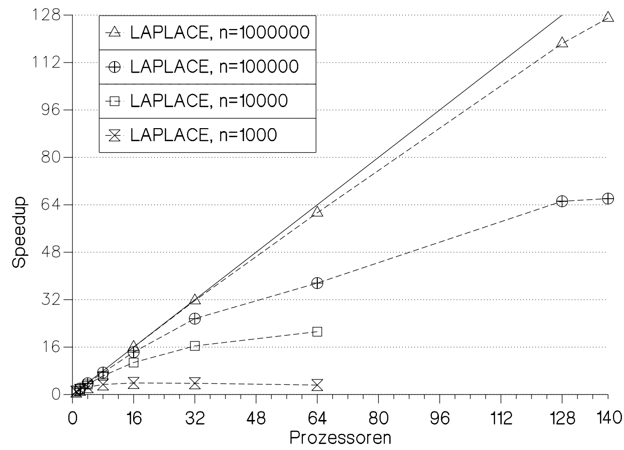


Abbildung 6.16: Speedup-Verhalten des CG-Verfahrens für die Matrizen **LAPLACE**

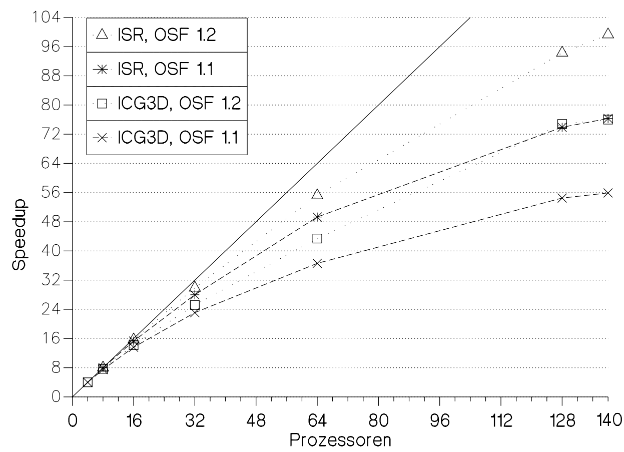


Abbildung 6.17: Speedup-Verhalten ohne und mit Einsatz des Kommunikationsprozessors

Die Reduzierung der Anzahl der Synchronisationspunkte und die Verwendung effizienter Methoden zur parallelen Matrix-Vektor-Multiplikation mit dünnbesetzter Matrix führt zu hohen Speedup-Werten des CG-Verfahrens bei unterschiedlich strukturierten Matrizen. Das Speedup-Verhalten aus Abbildung 6.16 läßt bei höheren Problemgrößen auch auf mehr als 140 Prozessoren hohe Effizienz erwarten.

In Abbildung 6.17 werden die erreichten Speedup-Werte des CG-Verfahrens unter dem Betriebssystem PARAGON 0SF/1, Release 1.1 bzw. Release 1.2, für die Matrizen **ICG3D** und **ISR** verglichen.

Bei beiden Matrizen liegen die unter 0SF/1, Release 1.2, erreichten Speedup-Werte bei hohen Prozessorzahlen deutlich über den Werten, die unter dem Release 1.1 gemessen wurden. Der Grund ist ein bedeutend günstigeres Kommunikationsverhalten durch den unter OSF/1, Release 1.2, eingesetzten Kommunikationsprozessor. Gegenüber dem Release 1.1 ist die Bandbreite des Verbindungsnetzwerks verdoppelt, während die Startup-Zeiten für Kommunikation um den Faktor 2 verkürzt sind. Auf 140 Prozessoren beträgt der Speedup mit Einsatz des Kommunikationsprozessors für die Matrizen **ICG3D** und **ISR** 76.0 bzw. 99.3; unter dem Release 1.1 wurde lediglich ein Speedup von 55.9 bzw. 76.3 gemessen.

Vorkonditionierung

Die Abbildungen 6.18 und 6.19 zeigen die Ausführungszeiten des CG-Verfahrens auf 140 Prozessoren ohne und mit Vorkonditionierung für die Matrizen **ICG3D** und **ISR** bzw. **BCSSTK17** und **PRESS**. Zur Vorkonditionierung werden die Skalierung mit der Diagonalen und die polynomiale Vorkonditionierung in Verbindung mit der vorherigen Skalierung der Matrix verwendet.

Gegenüber dem unvorkonditionierten CG-Verfahren führen beide Methoden mit Vorkonditionierung zu einer deutlichen Reduzierung der Ausführungszeit. Gründe sind einerseits beträchtlich geringere Iterationszahlen (s. Tabelle 6.9) und im Fall der polynomialen Vorkonditionierung mit Skalierung zusätzlich ein günstigeres Speedup-Verhalten, das unten erläutert wird. Bei den Matrizen **ICG3D**, **ISR**, **BCSSTK17** und **PRESS** verkürzen sich bei Verwendung der Skalierung mit der Diagonalen die Ausführungszeiten bereits um den Faktor 4.3, 2.2, 7.4 bzw. 1.5. Der zeitliche Aufwand für diese Vorkonditionierung ist von der Größenordnung des Aufwands für einen Iterationsschritt des CG-Verfahrens. Bei hohen Iterationszahlen ist er vernachlässigbar. Daher verhält sich die Verkürzung der Ausführungszeiten wie die Reduzierung der Anzahl der Iterationsschritte.

Zusätzliche polynomiale Vorkonditionierung führt bei den Matrizen **ICG3D** und **ISR** zu keiner weiteren Verbesserung; die Ausführungszeiten sind im Vergleich zu den Zeiten bei alleiniger Skalierung sogar erhöht. Zwar sinken die Iterationszahlen bei zusätzlicher polynomialer Vorkonditionierung weiter (s. Tabelle 6.9), jedoch steigt insgesamt die Anzahl der Matrix-Vektor-Multiplikationen gegenüber dem Verfahren mit Skalierung (s. auch 2.2.3). Eingespart werden Ska-

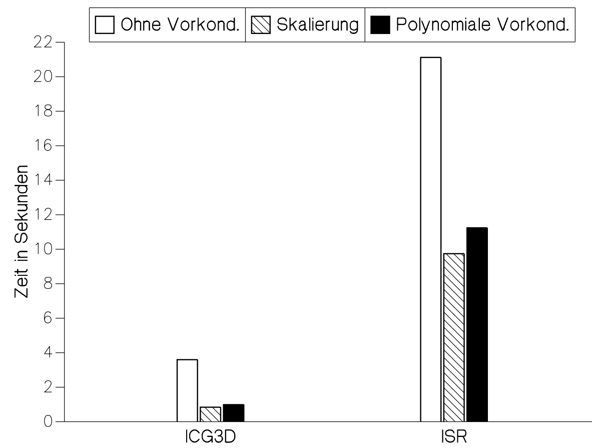


Abbildung 6.18: Ausführungszeiten des CG-Verfahrens ohne und mit Vorkonditionierung für die Matrizen **ICG3D** und **ISR**, 140 Prozessoren

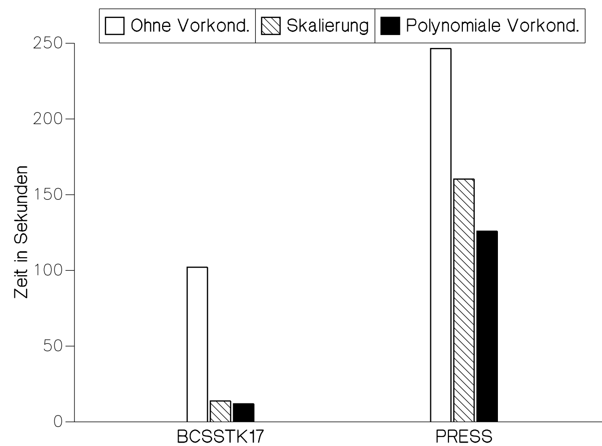


Abbildung 6.19: Ausführungszeiten des CG-Verfahrens ohne und mit Vorkonditionierung für die Matrizen **BCSSTK17** und **PRESS**, 140 Prozessoren

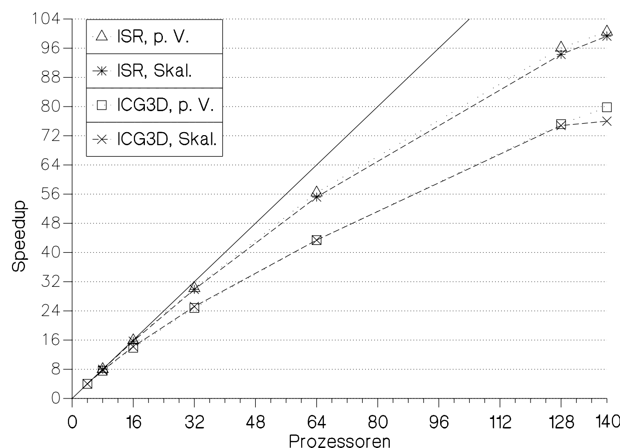


Abbildung 6.20: Speedup-Verhalten des CG-Verfahrens mit Skalierung und mit zusätzlicher polynomialer Vorkonditionierung für die Matrizen **ICG3D** und **ISR**

larprodukte — und damit globale Synchronisation — sowie Vektoradditionen und -skalierungen. Da bei beiden Matrizen der Anteil der Matrix-Vektor-Multiplikation am Rechenaufwand eines Iterationsschritts nach (5.3) mit 76% und 95% hoch ist, wird das Zeitverhalten durch das Einsparen von Iterationsschritten nicht verbessert. Bei den Matrizen **BCSSTK17** und **PRESS** hingegen werden die Ausführungszeiten des CG-Verfahrens durch polynomiale Vorkonditionierung weiter reduziert. Gegenüber der Skalierung werden um 14% bzw. 21% kürzere Zeiten erreicht. Auch bei diesen Matrizen liegt nach (5.3) mit 83% bzw. 86% ein hoher Anteil der Matrix-Vektor-Multiplikation am gesamten Rechenaufwand eines Iterationsschritts vor, jedoch ist die Anzahl der Nichtnull-Elemente deutlich kleiner als bei den oben betrachteten Matrizen. Daher ist der Einfluß des Synchronisationsaufwands bei hoher Prozessorzahl auf das Zeitverhalten des CG-Verfahrens mit Skalierung bei den Matrizen **BCSSTK17** und **PRESS** größer. Mit polynomialer Vorkonditionierung werden pro Iterationsschritt fünf bzw. neun Matrix-Vektor-Multiplikationen bei niedrigerer Iterationszahl gegenüber alleiniger Skalierung durchgeführt. Dadurch sinkt der Anteil des Synchronisationsaufwands am Gesamtaufwand eines Schritts.

Der letztere Zusammenhang wird im folgenden anhand des Speedup-Verhaltens des CG-Verfahrens mit Skalierung und mit zusätzlicher polynomialer Vorkonditionierung in den Abbildungen 6.20 und 6.21 verdeutlicht.

Bei allen vier Matrizen liegen die Speedup-Werte mit polynomialer Vorkonditionierung über den Werten für das CG-Verfahren mit Skalierung, wenn eine hohe Anzahl von Prozessoren verwendet wird. Mit polynomialer Vorkonditionierung

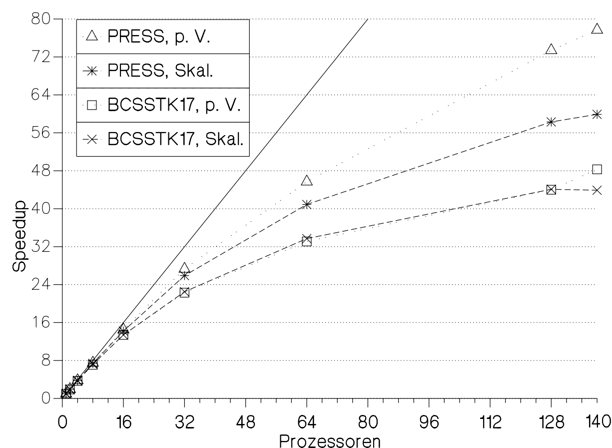


Abbildung 6.21: Speedup-Verhalten des CG-Verfahrens mit Skalierung und mit zusätzlicher polynomialer Vorkonditionierung für die Matrizen **BCSSTK17** und **PRESS**

werden auf 140 Prozessoren für die Matrizen **ICG3D**, **ISR**, **BCSSTK17** und **PRESS** Speedup-Werte von 79.8, 100.5, 48.3 bzw. 77.7 erreicht, während die entsprechenden Werte bei Skalierung der Matrix mit 76.0, 99.3, 43.9 bzw. 59.9 niedriger sind. Der geringere Anteil des Synchronisationsaufwands an der Gesamtzeit eines Iterationsschritts bei polynomialer Vorkonditionierung führt zu einem günstigeren Speedup-Verhalten. Je höher der Grad m des Polynoms gewählt wird, desto mehr nähern sich die Speedup-Werte des CG-Verfahrens mit polynomialer Vorkonditionierung den Werten der Matrix-Vektor-Multiplikation aus Abbildung 6.9, da die Matrix-Vektor-Multiplikation das Zeitverhalten in steigendem Maße bestimmt.

Mit zunehmendem Grad steigt jedoch insgesamt der Aufwand für die Matrix-Vektor-Multiplikationen, wenn die Anzahl der Iterationsschritte des CG-Verfahrens mit Skalierung durch polynomiale Vorkonditionierung um weniger als den Faktor $m+1$ reduziert wird (s. auch 2.2.3). Daher kann das Zeitverhalten bei hohem Polynomgrad insbesondere bei kleiner Prozessorzahl ungünstiger sein. Bei großer Prozessorzahl kann dagegen ein hoher Grad durch die bessere Effizienz der Methode die Reduzierung der Ausführungszeiten bewirken. Für die Matrix **PRESS** z. B. beträgt die Ausführungszeit des CG-Verfahrens mit polynomialer Vorkonditionierung auf einem Prozessor 9992 s, während mit Skalierung lediglich 9366 s erforderlich sind. Auf 140 Prozessoren hingegen erreicht die Methode bei polynomialer Vorkonditionierung mit 125.9 s gegenüber 160.3 s bei Skalierung die deutlich kürzere Ausführungszeit. Ein besonders günstiges Zeitverhalten bei polynomialer Vorkonditionierung ergibt sich bei der Matrix **BCSSTK17**. Das

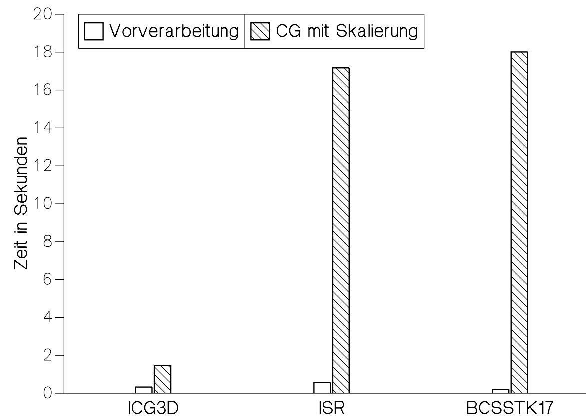


Abbildung 6.22: Vergleich der Ausführungszeiten für die Vorverarbeitung und das CG-Verfahren mit Skalierung, 64 Prozessoren

Verfahren führt sowohl auf einem Prozessor mit 575 s gegenüber 608 s bei alleiniger Skalierung als auch auf 140 Prozessoren mit 11.9 s gegenüber 13.8 s zu kürzeren Ausführungszeiten. Durch parallele Bearbeitung auf 140 Prozessoren und polynomiale Vorkonditionierung in Verbindung mit der Skalierung der Matrix wird die Ausführungszeit des CG-Verfahrens ohne Vorkonditionierung auf einem Prozessor für die Matrix **PRESS** von 4.0 h um den Faktor 114 auf 2.1 min und für die Matrix **BCSSTK17** von 1.25 h um den Faktor 379 auf 11.9 s gesenkt.

Vorverarbeitung

In Abbildung 6.22 werden die Ausführungszeiten des Vorverarbeitungsschritts zur Ermittlung des Kommunikationsschemas (s. auch Abbildung 5.13) und des CG-Verfahrens mit Skalierung für die Matrizen **ICG3D**, **ISR** und **BCSSTK17** auf 64 Prozessoren gegenübergestellt. Die Vorverarbeitung wird wie die iterative Methode parallel durchgeführt. Bei Verwendung der entwickelten parallelen iterativen Verfahren in FE-Modellen ist die Ermittlung des Kommunikationsschemas gewöhnlich nur einmal erforderlich; der zeitliche Aufwand kann vernachlässigt werden. Lediglich bei Änderung des Diskretisierungsgitters muß das Schema neu berechnet werden. Wird die iterative Methode jedoch losgelöst vom Diskretisierungsproblem zur Lösung einzelner Gleichungssysteme oder Eigenwertprobleme betrachtet — z. B. als Bibliotheksroutine —, so muß der zeitliche Aufwand für die Vorverarbeitung berücksichtigt werden.

Bei allen drei Matrizen ist die Ausführungszeit des iterativen Verfahrens um

ein Vielfaches höher als der zeitliche Aufwand der Vorverarbeitung. Während die Zeit zur Ermittlung des Kommunikationsschemas bei der Matrix **ICG3D** 22% der Ausführungszeit der iterativen Methode beträgt, ist der Aufwand bei den Matrizen **ISR** und **BCSSTK17** mit 3% bzw. 1% der Zeit des CG-Verfahrens vernachlässigbar. Für die nicht dargestellte Matrix **PRESS** ergibt sich ein Aufwand für die Vorverarbeitung von 0.1% der Ausführungszeit der iterativen Methode. Dies weist darauf hin, daß bei hohen Iterationszahlen des Verfahrens der Aufwand für die Vorverarbeitung nicht berücksichtigt werden muß. Die Ermittlung des Kommunikationsschemas wird effizient durchgeführt, so daß die entwickelten iterativen Methoden auch zur Lösung von einzelnen Gleichungssystemen oder Eigenwertproblemen geeignet sind. Bei den im weiteren betrachteten Verfahren QMR und TFQMR sowie der Lanczos-Methode ist der Aufwand der Vorverarbeitung im Vergleich zur Ausführungszeit der Iteration in der Regel geringer als beim CG-Verfahren, da der Rechenaufwand pro Iterationsschritt höher ist.

6.3.3 Die Verfahren QMR und TFQMR

Im folgenden werden Parallelisierungseigenschaften des QMR- und des TFQMR-Verfahrens zur Lösung von Gleichungssystemen mit regulärer Matrix beschrieben. Zum Abbruch der Iteration wurde in allen Untersuchungen der Wert des Residuums überprüft. Das Kriterium (2.2) ist für die Methoden QMR und TFQMR aufgrund des Fehlerverhaltens nicht geeignet [37] [43].

Numerische Resultate

In Tabelle 6.10 sind die Iterationszahlen der Methoden QMR und TFQMR für die Matrizen **WATT1**, **ORSREG1**, **ICG3D** und **ISR** zusammengefaßt.

	Iterationszahl		
	QMR	TFQMR	CG
WATT1	10	10	-
ORSREG1	463	-	-
ICG3D	671	440	707
ISR	2155	-	2170

Tabelle 6.10: Anzahl der Iterationsschritte der Verfahren QMR, TFQMR und CG

Die Iteration wurde abgebrochen, sobald die Norm des Residuums den Wert 10^{-5} unterschritt. Für die symmetrischen Matrizen **ICG3D** und **ISR** ist die Anzahl der Iterationsschritte des CG-Verfahrens ohne Vorkonditionierung bei diesem Abbruchkriterium hinzugefügt. Beim TFQMR-Verfahren wurde das Residuum alle zehn Iterationsschritte berechnet und überprüft (s. auch 2.3.2). Die Iterationszahl für die Matrizen **WATT1** und **ORSREG1** wurde auf einem Prozessor

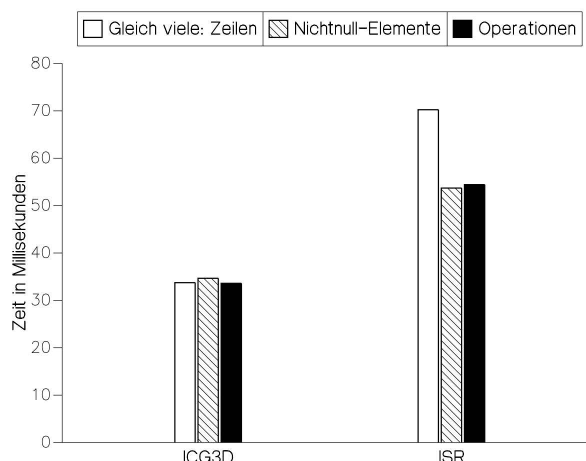


Abbildung 6.23: Ausführungszeiten der QMR-Methode pro Iterationsschritt bei unterschiedlicher Datenverteilung, 64 Prozessoren

bestimmt, die Anzahl der Schritte für die Matrizen **ICG3D** und **ISR** auf vier bzw. acht Prozessoren.

Das TFQMR-Verfahren erfüllte das Abbruchkriterium für die Matrizen **ISR** und **ORSREG1** innerhalb von 5000 Schritten nicht. Daher sind für diese Matrizen keine Iterationszahlen angegeben. Bei den Matrizen **ICG3D** und **ISR** unterscheiden sich die Iterationszahlen für die Methoden QMR und CG kaum. Für das QMR-Verfahren wurde Konvergenz auch bei höherer Genauigkeit des Residuums festgestellt, bei der die TFQMR-Methode lediglich für die Matrix **WATT1** konvergierte. Dies weist auf ein günstigeres Konvergenz- und Fehlerverhalten des QMR-Verfahrens gegenüber der TFQMR-Methode hin (vergl. [37]).

Datenverteilung

In den Abbildungen 6.23 und 6.24 sind die Ausführungszeiten der Verfahren QMR und TFQMR pro Iterationsschritt auf 64 Prozessoren bei Verteilung der Daten gemäß Kriterium (5.2) mit $\xi = 10^{14}$, $\xi = 0$ und $\xi = 16$ (QMR) bzw. $\xi = 13$ (TFQMR) für die Matrizen **ICG3D** und **ISR** dargestellt (s. auch 5.3, Tabelle 5.1).

Wie beim CG-Verfahren führen auch bei den Methoden QMR und TFQMR die charakteristischen Werte des Parameters der Datenverteilung zu einer günstigen Lastverteilung. Für die unregelmäßig besetzte Matrix **ISR** ist die Ausführungszeit des QMR-Verfahrens bei der Datenverteilung nach dem Kriterium „Gleich viele Operationen“ ($\xi = 16$) gegenüber der Zeit bei Verteilung der Daten gemäß dem Kriterium „Gleich viele Zeilen“ ($\xi = 10^{14}$) um 23% reduziert. Bei der

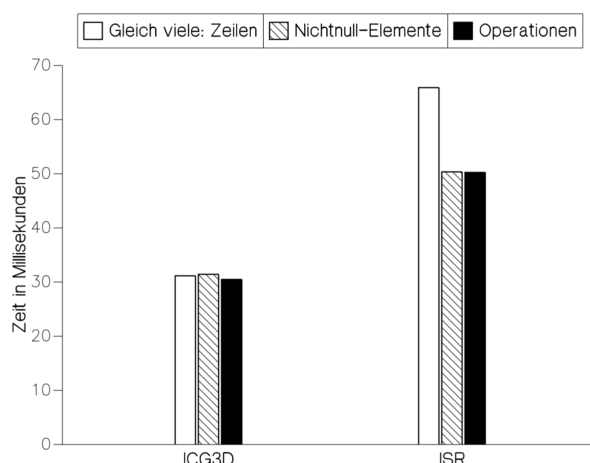


Abbildung 6.24: Ausführungszeiten der TFQMR-Methode pro Iterationsschritt bei unterschiedlicher Datenverteilung, 64 Prozessoren

TFQMR-Methode ergibt sich für $\xi = 13$ verglichen mit der Zeit für $\xi = 10^{14}$ eine um 24% verkürzte Ausführungszeit. Für die Matrix **ISR** ergeben sich bei beiden Verfahren annähernd gleiche Ausführungszeiten für die charakteristischen Werte von ξ und für $\xi = 0$, da der operationale Aufwand für die Matrix-Vektor-Multiplikationen das Zeitverhalten im wesentlichen bestimmt. Für die Matrix **ICG3D** mit regelmäßigem Besetzungsmuster unterscheiden sich die Ausführungszeiten beider Verfahren bei allen drei Datenverteilungen kaum.

Vergleich der Ausführungszeiten

In Abbildung 6.25 werden die Ausführungszeiten pro Iterationsschritt der Verfahren QMR und TFQMR für die Matrizen **WATT1** und **ORSREG1** auf einem Prozessor sowie **ICG3D** und **ISR** auf 32 Prozessoren verglichen. Für die symmetrischen Matrizen sind die entsprechenden Zeiten der CG-Iteration ergänzt.

Die Ausführungszeiten der TFQMR-Methode sind gegenüber den Zeiten des QMR-Verfahrens reduziert; dies wird im wesentlichen durch eine kleinere Anzahl von Vektor-Vektor-Operationen pro Iterationsschritt als in der QMR-Iteration verursacht. Der Unterschied ist jedoch gering. Dagegen sind die Ausführungszeiten des CG-Verfahrens pro Schritt nur ca. halb so groß wie die Zeiten der QMR- und TFQMR-Methode, da in jedem Iterationsschritt lediglich eine Matrix-Vektor-Multiplikation gegenüber zwei in der QMR bzw. TFQMR-Iteration und zusätzlich weniger Vektor-Vektor-Operationen durchgeführt werden. Bei gleicher Anzahl der Iterationsschritte ist der operationale Aufwand der Verfahren QMR und TFQMR ca. doppelt so hoch wie der Aufwand der CG-Iteration.

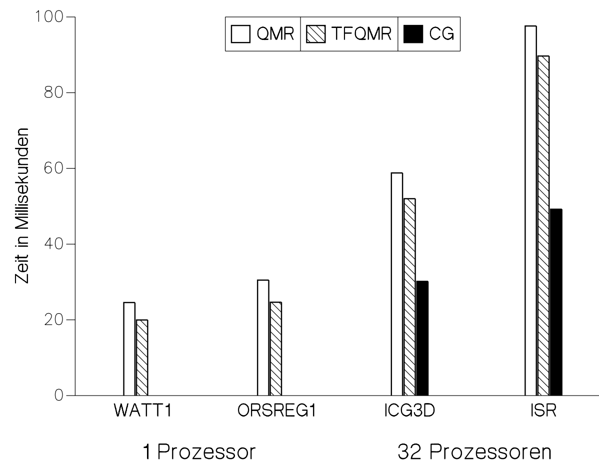


Abbildung 6.25: Ausführungszeiten pro Iterationsschritt der Methoden QMR, TFQMR und CG

Kopplung

In den obigen Untersuchungen wurde das QMR-Verfahren ohne Kopplung des Datenaustauschs für die beiden voneinander unabhängigen Matrix-Vektor-Multiplikationen mit $\boldsymbol{A}$ und $\boldsymbol{A}^T$ verwendet (s. auch 5.4.4). Abbildung 6.26 zeigt die Ausführungszeiten der QMR-Methode ohne und mit Kopplung der Kommunikation für beide Operationen. Die Zeiten für die Matrizen **WATT1** und **ORSREG1** wurden auf 32 Prozessoren gemessen, die Zeiten für die Matrizen **ICG3D** und **ISR** auf 140 Prozessoren.

Bei allen vier Matrizen sind die Ausführungszeiten des Verfahrens mit Kopplung kürzer als die Zeiten der Methode ohne Kopplung. Durch die Kopplung des Datenaustauschs für die beiden unabhängigen Matrix-Vektor-Multiplikationen eines Iterationsschritts wird der Kommunikationsaufwand für diese Operationen reduziert. Für die großen Matrizen **ICG3D** und **ISR** wirkt sich dies nur geringfügig auf die Gesamtzeit aus; die Rechenoperationen und der Aufwand für globale Synchronisation bestimmen im wesentlichen das Zeitverhalten. Pro Iterationsschritt der QMR- wie der TFQMR-Methode sind drei Synchronisationspunkte erforderlich. Bei den kleineren Matrizen **WATT1** und **ORSREG1** ist der Anteil des Kommunikationsaufwands für die Matrix-Vektor-Multiplikationen an der Gesamtzeit aufgrund des geringen Rechenaufwands größer. Hier führt die Kopplung der Kommunikation für die Matrix-Vektor-Multiplikationen mit $\boldsymbol{A}$ und $\boldsymbol{A}^T$ zu um 11% bzw. um 14% kürzeren Ausführungszeiten. In den folgenden Untersuchungen wird das QMR-Verfahren mit Kopplung verwendet.

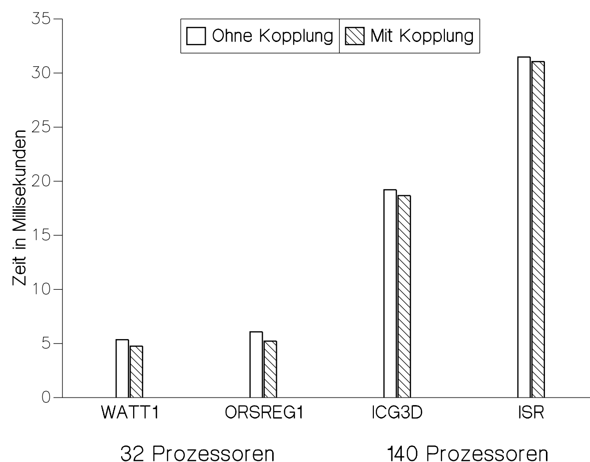


Abbildung 6.26: Ausführungszeiten pro Iterationsschritt der QMR-Methode ohne und mit Kopplung

Skalierungseigenschaften

In den Abbildungen 6.27 und 6.28 sind die Speedup-Werte der Verfahren QMR bzw. TFQMR für die Matrizen **WATT1** und **ORSREG1** auf bis zu 32 Prozessoren sowie für die Matrizen **ICG3D** und **ISR** auf bis zu 140 Prozessoren dargestellt.

Bei beiden Verfahren werden für die Matrizen **WATT1** und **ORSREG1** aufgrund der kleinen Problemgröße lediglich geringe Speedup-Werte erreicht. Auf 32 Prozessoren betragen die Speedup-Werte 5.2 bzw. 5.7 für die QMR-Methode und 3.9 bzw. 4.1 für das TFQMR-Verfahren. Die höheren Werte der QMR-Iteration kommen durch die Kopplung der Kommunikation für die Matrix-Vektor-Multiplikationen mit $\boldsymbol{A}$ und $\boldsymbol{A}^T$ sowie eine größere Anzahl von Vektor-Vektor-Operationen pro Iterationsschritt zustande. Für die großen Matrizen **ICG3D** und **ISR** ist das Speedup-Verhalten beider Verfahren ähnlich. Auf 140 Prozessoren werden für die QMR-Methode Speedup-Werte von 73.1 bzw. 89.3 und für das TFQMR-Verfahren von 66.4 bzw. 83.8 erreicht.

Durch Änderung der Reihenfolge der Operationen konnte die Anzahl der Synchronisationspunkte pro Iterationsschritt bei beiden Verfahren auf drei reduziert werden. Darüber hinausgehende Ansätze zur Verminderung des Synchronisationsaufwands für die iterativen Methoden GCR (*Generalized Conjugate Residual*) und Omin (*Orthomin*) zur Lösung unsymmetrischer Gleichungssysteme, die u. U. auf die Verfahren QMR und TFQMR übertragen werden können, sind in [136] zu finden. Ferner kann die Gesamtzahl der Synchronisationspunkte durch Vorkonditionierung der Verfahren verringert werden.

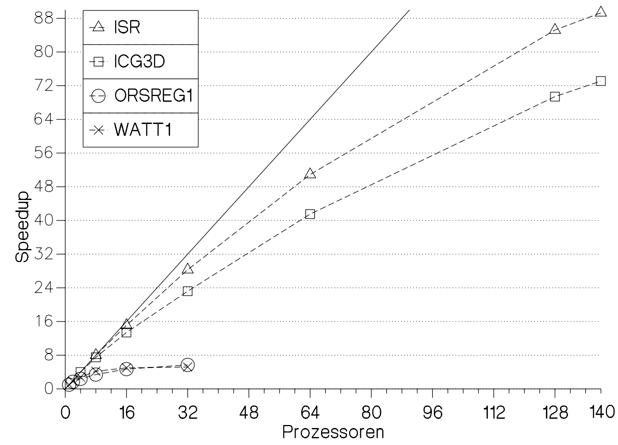


Abbildung 6.27: Speedup-Werte des QMR-Verfahrens

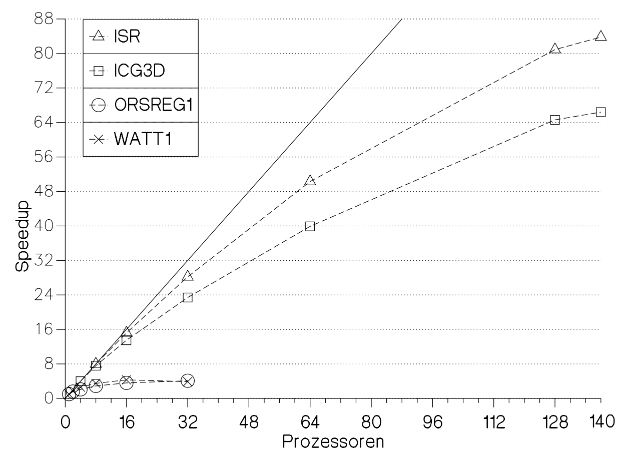


Abbildung 6.28: Speedup-Werte des TFQMR-Verfahrens

6.3.4 Die Lanczos-Methode

Die entwickelte parallele Lanczos-Methode zur Lösung des symmetrischen Eigenwertproblems erzeugt zunächst aus der ursprünglichen Matrix eine Folge von Tridiagonalmatrizen mit steigender Ordnung (s. 2.4). Nach einer vorgegebenen Anzahl von Iterationsschritten der Tridiagonalisierung werden die Eigenwerte der aktuellen Tridiagonalmatrix bestimmt und überprüft, welche Eigenwerte ausreichende Approximationen der Eigenwerte der ursprünglichen Matrix sind.

In allen folgenden Untersuchungen wurde die Lanczos-Iteration abgebrochen, wenn mindestens 100 Eigenwerte der ursprünglichen Matrix ermittelt waren. Die Eigenwerte der Tridiagonalmatrizen wurden mit einer Genauigkeit von 11 Dezimalstellen berechnet; das Abbruchkriterium ist in [28] beschrieben. Einfache Eigenwerte der Tridiagonalmatrizen, die Näherungen für Eigenwerte der ursprünglichen Matrix sind und mit einer Genauigkeit von neun Dezimalstellen mit unechten Eigenwerten übereinstimmen, werden als nicht-isolierte einfache Eigenwerte markiert. Die Toleranz ϵ_t aus (2.12) wurde zu 10^{-6} gesetzt.

Numerische Resultate

In Tabelle 6.11 ist die Anzahl der ermittelten Eigenwerte nach einem vorgegebenen Zyklus von Schritten der Lanczos-Tridiagonalisierung für die Matrizen **ISR-RCM**, **LAPLACE** mit $n = 1000000$ und **ICG3D-RCM** angegeben. Bei den Matrizen **ISR-RCM** und **LAPLACE** wurden die Eigenwerte der Tridiagonalmatrix alle 1000 Schritte bestimmt, bei der Matrix **ICG3D-RCM** alle 10000 Schritte. Die Messung wurde auf 140 Prozessoren durchgeführt.

	Schritte	EW	Schritte	EW	Schritte	EW
ISR-RCM	1000	50	2000	136	-	-
LAPLACE	1000	72	2000	104	-	-
ICG3D-RCM	10000	2	20000	37	30000	449

Tabelle 6.11: Anzahl der ermittelten Eigenwerte nach einem vorgegebenen Zyklus

Für die Matrizen **ISR-RCM** und **LAPLACE** sind zwei Zyklen erforderlich, um mindestens 100 Eigenwerte der Matrix zu bestimmen, während dieses Kriterium für die Matrix **ICG3D-RCM** nach drei Zyklen erfüllt ist.

Vergleich der Varianten

In Abbildung 6.29 sind die Ausführungszeiten der zwei in 2.4.1 vorgestellten Varianten der Lanczos-Tridiagonalisierung auf 32 bis 140 Prozessoren für die Matrizen **ICG3D** und **ISR** gegenübergestellt. Die Zeiten wurden unter dem Betriebssystem PARAGON OSF/1, Release 1.1, gemessen.

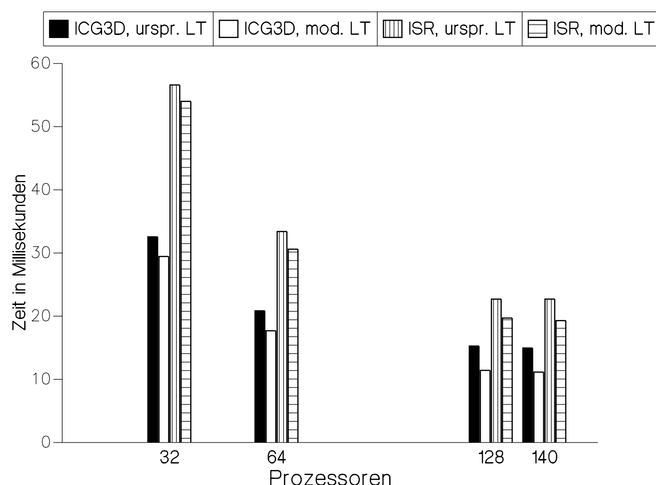


Abbildung 6.29: Ausführungszeiten der ursprünglichen und der modifizierten Lanczos-Tridiagonalisierung

Wie beim CG-Verfahren in Abbildung 6.11 sind die Ausführungszeiten des modifizierten Verfahrens bei hoher Prozessorzahl deutlich gegenüber denen der ursprünglichen Methode reduziert, da globale Synchronisation eingespart wird. Für die Matrix **ICG3D** ergibt sich auf 140 Prozessoren eine um 25% kürzere Ausführungszeit des modifizierten Verfahrens; für die Matrix **ISR** beträgt die Reduzierung der Zeit 15%. In allen folgenden Untersuchungen wird daher die modifizierte Methode verwendet.

Datenverteilung

Abbildung 6.30 zeigt die Ausführungszeiten der Lanczos-Tridiagonalisierung pro Iterationsschritt auf 64 Prozessoren bei Verteilung der Daten gemäß (5.2) mit $\xi = 10^{14}$, $\xi = 0$ und dem charakteristischen Wert $\xi = 2$ (s. Tabelle 5.1) für die Matrizen **ICG3D**, **ICG3D-RCM**, **ISR** und **ISR-RCM**.

Die günstigste Auslastung der Prozessoren wird für die Datenverteilung nach den Kriterien „Gleich viele Nichtnull-Elemente“ ($\xi = 0$) und „Gleich viele Operationen“ ($\xi = 2$) erreicht. Die Ausführungszeiten für diese Parameterwerte unterscheiden sich kaum, da einerseits in der Lanczos-Tridiagonalisierung nur wenige Vektor-Vektor-Operationen durchgeführt werden — der niedrige charakteristische Wert $\xi = 2$ weist darauf hin — und andererseits der Anteil der Matrix-Vektor-Multiplikation am gesamten Rechenaufwand eines Iterationsschritts für die betrachteten Matrizen hoch ist. Für die regelmäßig besetzten Matrizen **ICG3D** und **ICG3D-RCM** ergeben alle drei Datenverteilungen ein annähernd gleiches Zeitverhalten, während für die unregelmäßiger strukturierten Matrizen

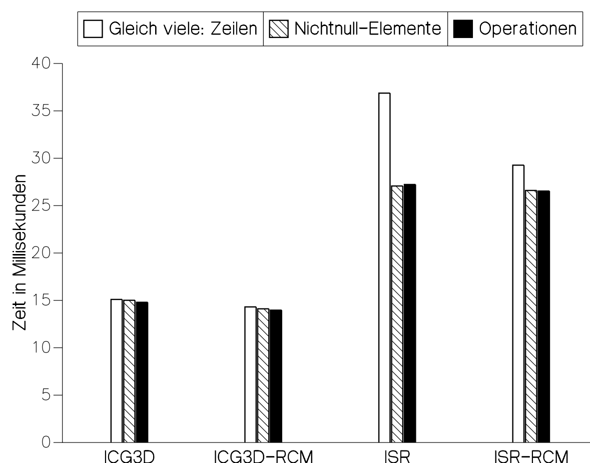


Abbildung 6.30: Ausführungszeiten der Lanczos-Tridiagonalisierung pro Iterationsschritt bei unterschiedlicher Datenverteilung, 64 Prozessoren

ISR und **ISR-RCM** deutlich erhöhte Zeiten bei der Verteilung nach dem Kriterium „Gleich viele Zeilen" ($\xi = 10^{14}$) auftreten. Gegenüber diesen Zeiten sind die Ausführungszeiten bei der Datenverteilung mit dem charakteristischen Wert um 26% bzw. 9% reduziert. Für die Matrix **ISR-RCM** ist die Zeitdifferenz kleiner als für die Matrix **ISR**, da die RCM-Permutation der Elemente dieser Matrix bei $\xi = 10^{14}$ zu einer gleichmäßigeren Verteilung der Zeilen mit vielen und wenigen Nichtnull-Elementen auf die Prozessoren führt (s. auch Abbildung 6.1).

Ausführungszeiten

In den Abbildungen 6.31 und 6.32 sind die Ausführungszeiten der Lanczos-Tridiagonalisierung, des Verfahrens zur Bestimmung der Eigenwerte der Tridiagonalmatrizen und der gesamten Lanczos-Methode auf 32 bis 140 Prozessoren für die Matrizen **ISR-RCM** bzw. **LAPLACE** mit $n = 1000000$ dargestellt.

Bei beiden Matrizen ist der Anteil des Verfahrens zur Bestimmung der Eigenwerte der Tridiagonalmatrizen an der gesamten Ausführungszeit der Lanczos-Methode gering, da die Ordnung der Tridiagonalmatrizen nach beiden Zyklen mit $n = 1000$ bzw. $n = 2000$ klein ist. Die Zeit für die Tridiagonalisierung dominiert das Zeitverhalten. Auf 140 Prozessoren beträgt der Anteil des Verfahrens zur Lösung des tridiagonalen Eigenwertproblems für die Matrizen **ISR-RCM** und **LAPLACE** an der Gesamtzeit von 34.9 s bzw. 78.0 s lediglich 9% bzw. 6%.

Während der zeitliche Aufwand der Tridiagonalisierung in jedem Zyklus gleich ist, steigt der Aufwand zur Lösung des tridiagonalen Eigenwertproblems mit der Anzahl der Zyklen. Die Komplexität des Verfahrens zur Bestimmung der Ei-

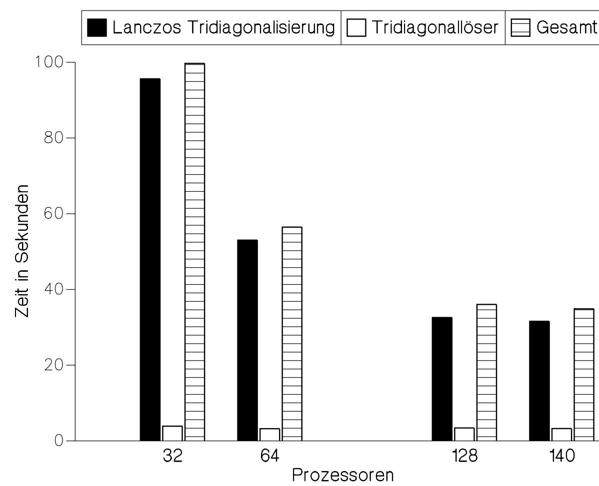


Abbildung 6.31: Ausführungszeiten der gesamten Lanczos-Methode, Matrix **ISR-RCM**

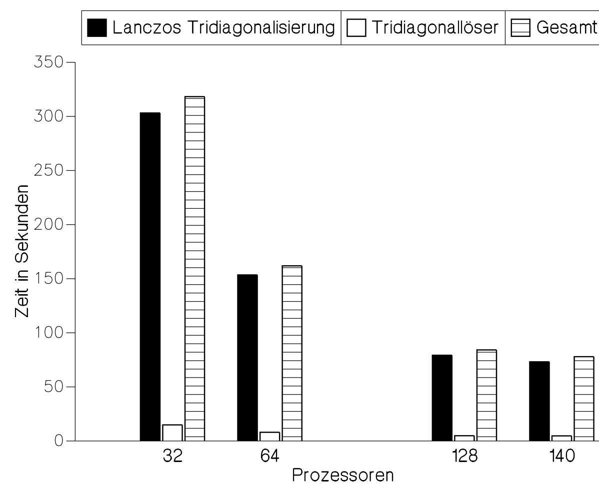


Abbildung 6.32: Ausführungszeiten der gesamten Lanczos-Methode, Matrix **LAPLACE**

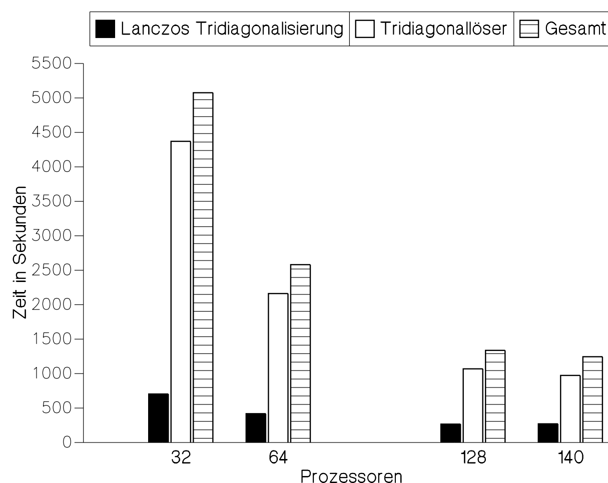


Abbildung 6.33: Ausführungszeiten der gesamten Lanczos-Methode, Matrix **ICG3D-RCM**

genwerte der Tridiagonalmatrizen ist $\mathcal{O}(n^2)$. Im Fall der Matrix **ICG3D-RCM** dominiert der Aufwand zur Lösung des tridiagonalen Eigenwertproblems das Zeitverhalten der Lanczos Methode. Abbildung 6.33 zeigt die Ausführungszeiten für diese Matrix.

Auf 140 Prozessoren beträgt der Anteil des Verfahrens zur Bestimmung der Eigenwerte an der Gesamtzeit von 1245 s 78%. Durch parallele Bearbeitung der Lanczos-Methode wird die Ausführungszeit von 10.4 h (vier Prozessoren) auf 20.7 min (140 Prozessoren) verkürzt.

Skalierungseigenschaften

Abbildung 6.34 veranschaulicht das Speedup-Verhalten der Lanczos-Methode für die Matrizen **ICG3D-RCM**, **ISR-RCM** und **LAPLACE** auf bis zu 140 Prozessoren.

Für die Matrizen **ISR-RCM** und **LAPLACE** bestimmt im wesentlichen die Lanczos-Tridiagonalisierung das Speedup-Verhalten, während die Speedup-Werte für die Matrix **ICG3D-RCM** überwiegend auf dem Zeitverhalten des Verfahrens zur Lösung des tridiagonalen Eigenwertproblems beruhen. Die Speedup-Werte für die Matrix **ICG3D-RCM** liegen bei hoher Prozessorzahl deutlich über den entsprechenden Werten der Matrix **ISR-RCM**, da die Extraktionsphase des Verfahrens zur Bestimmung der Eigenwerte der Tridiagonalmatrizen, die das Zeitverhalten bei der Matrix **ICG3D-RCM** dominiert, vollständig parallel ohne Datenaustausch durchgeführt wird. Auf 140 Prozessoren wird für die Matrix **ISR-RCM** ein Speedup von 86.2, für die Matrix **ICG3D-RCM** von 120.2 und

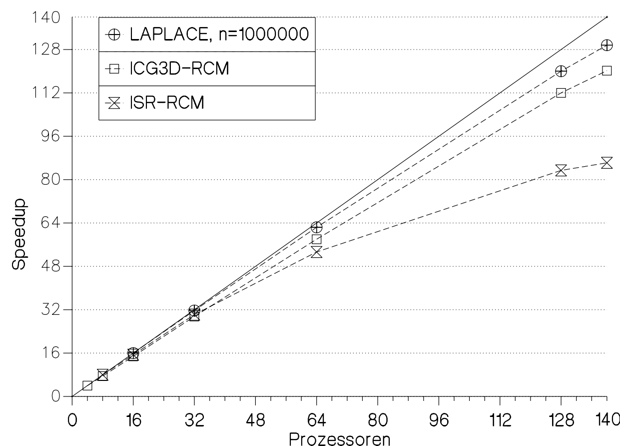


Abbildung 6.34: Speedup-Verhalten der Lanczos-Methode

für die Matrix **LAPLACE** von 129.6 erreicht. Das Speedup-Verhalten in Abbildung 6.34 zeigt, daß sowohl die entwickelte parallele Lanczos-Tridiagonalisierung als auch das Verfahren zur Lösung des tridiagonalen Eigenwertproblems günstige Skalierungseigenschaften auf massiv-parallelen Systemen mit verteiltem Speicher besitzen. Die Kombination beider Verfahren, deren Parallelisierungsstrategie verschieden ist, zur parallelen Lösung des symmetrischen Eigenwertproblems führt zu hoher Effizienz der gesamten Methode.

6.4 Algorithmische Skelette in einer funktionalen Sprache

Die Skalierungseigenschaften der in eine funktionale Programmiersprache integrierten algorithmischen Skelette zur Matrix-Vektor-Multiplikation mit dünnbesetzter Matrix (s. 5.5) wurden auf einem Transputer-System des Rechenzentrums der RWTH Aachen getestet. Die Untersuchung wurde in Zusammenarbeit mit dem Lehrstuhl für Informatik II der RWTH Aachen durchgeführt. Die im folgenden präsentierten Meßergebnisse sind [135] entnommen. Weitere Einzelheiten über die Ergebnisse der Kooperation sowie die Meßumgebung sind ebenfalls in dieser Arbeit beschrieben.

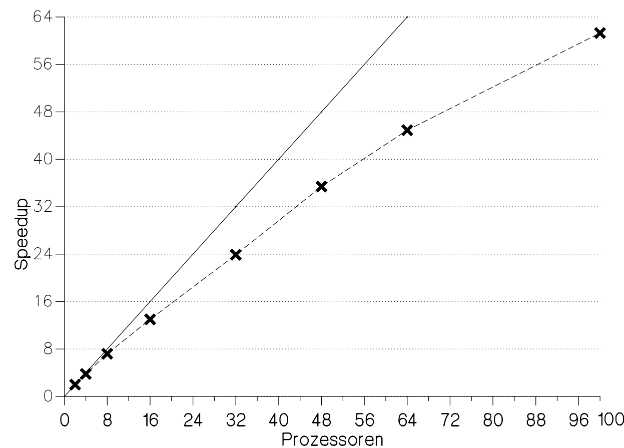


Abbildung 6.35: Speedup-Verhalten des Vorverarbeitungsschritts zur Ermittlung des Kommunikationsschemas, Matrix **ICG2D-RCM**

6.4.1 Vorverarbeitung

Abbildung 6.35 zeigt das Speedup-Verhalten des Vorverarbeitungsschritts zur Ermittlung des Kommunikationsschemas der Matrix-Vektor-Multiplikation auf bis zu 100 Prozessoren. Alle Untersuchungen wurden unter Verwendung der Matrix **ICG2D-RCM** durchgeführt, die auf dem Transputer-System den Speicherplatz von mindestens zwei Prozessoren benötigt. Auf zwei Prozessoren wurde ein Speedup von 2.0 angenommen. Zur gleichmäßigen Aufteilung der Operationen der Matrix-Vektor-Multiplikation auf die Prozessoren wurde das Kriterium (5.2) mit $\xi = 0$ verwendet. Zur Bestimmung der Speedup-Werte wurden die Ausführungszeiten des Skeletts `sk_compute_communication_scheme` (s. Abbildung 5.22) gemessen.

Speedup-Verluste sind auf den Datenaustausch zur Festlegung des Schemas, welche lokalen Daten der einzelnen Prozessoren an andere Prozessoren gesendet werden, zurückzuführen. Auf 100 Prozessoren wird ein Speedup von 61.3 erreicht.

6.4.2 Matrix-Vektor-Multiplikation

Das Speedup-Verhalten des algorithmischen Skeletts `sk_sparse_mvp` zur parallelen Matrix-Vektor-Multiplikation mit dünnbesetzter Matrix ist in Abbildung 6.36 für die Matrix **ICG2D-RCM** dargestellt.

Die Messung der Ausführungszeiten ergibt akzeptable Speedup-Werte von 35.3 und 47.0 auf 64 bzw. 100 Prozessoren. Bei größeren Matrizen ist eine höhere Effizienz zu erwarten. Die auf dem Transputer-System erreichten Speedup-Werte

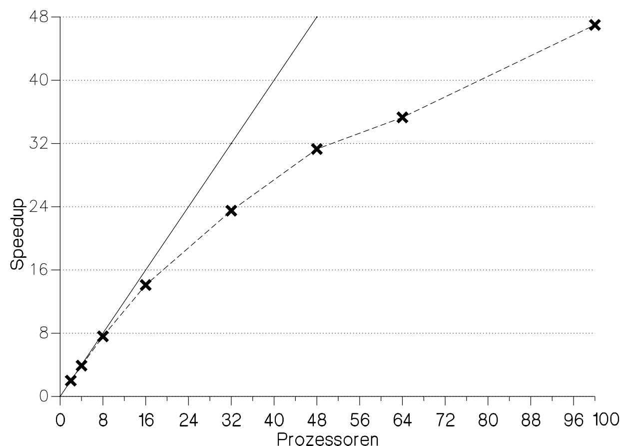


Abbildung 6.36: Speedup-Verhalten der parallelen Matrix-Vektor-Multiplikation, Matrix **ICG2D-RCM**

liegen bei hoher Prozessorzahl unter den auf dem PARAGON gemessenen Werten (s. Abbildung 6.9). Einerseits besitzt der PARAGON gegenüber dem Transputer-System ein leistungsfähigeres Verbindungsnetzwerk der Prozessoren — dies führt zu einem günstigeren Kommunikationsverhalten —, andererseits wurde auf dem Transputer-System ein vereinfachtes Verfahren zur parallelen Matrix-Vektor-Multiplikation mit geringerem Überlappungsgrad implementiert. Letzteres kann gegenüber der auf dem PARAGON verwendeten Methode geringfügig höhere Wartezeiten verursachen. Die Ausführungszeit auf zwei Prozessoren beträgt auf dem Transputer-System 1.41 s, während die Zeit auf zwei Prozessoren des PARAGON mit 0.066 s deutlich kürzer ist. Die um den Faktor 21 niedrigere Ausführungszeit kommt im wesentlichen durch die deutlich höhere Rechenleistung des PARAGON-Prozessors i860 XP gegenüber dem T800 des Transputer-Systems [59] zustande.

Die in eine funktionale Programmiersprache integrierten algorithmischen Skelette zur parallelen Matrix-Vektor-Multiplikation sind als effiziente Bausteine für iterative Methoden in technischen Problemstellungen verwendbar. Parallele iterative Verfahren zur Lösung von Gleichungssystemen oder Eigenwertproblemen können mit Hilfe dieser Skelette in einfacher Weise programmiert werden.

Kapitel 7

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden parallele iterative Verfahren für dünnbesetzte Matrizen zur Lösung von linearen Gleichungssystemen und Eigenwertproblemen entwickelt. Die Methoden zeigen gute Skalierungseigenschaften auf massiv-parallelen Systemen für Matrizen unterschiedlicher Besetzungsstruktur aus technischen Anwendungen.

Zur komprimierten Speicherung dünnbesetzter Matrizen wurden in FE-Modellen übliche Techniken untersucht. Das entwickelte SICRS-Schema — eine Modifikation des CRS-Formats — berücksichtigt pro Zeile aufeinanderfolgende Spaltenindizes der Nichtnull-Elemente. Dadurch kann in vielen Fällen sowohl Speicherplatz gespart als auch indirekte Adressierung bei der Matrix-Vektor-Multiplikation vermieden werden.

Zur Verteilung der Daten eines iterativen Verfahrens auf die einzelnen Prozessoren eines Parallelrechners mit verteiltem Speicher wurde ein Schema entwickelt, das zur Balancierung der Rechenlast führt. Die Verteilung wird durch einen Parameter gesteuert, der sowohl die Operationen der iterativen Methode als auch die Prozessorarchitektur des Parallelrechners berücksichtigt. Das Schema der Datenverteilung wird automatisch durch Analyse der Besetzungsstruktur der Matrix ermittelt.

Für die Matrix-Vektor-Multiplikation mit dünnbesetzter Matrix wurden verschiedene Kommunikationsschemata untersucht, die auf der Analyse des Besetzungsmusters der Matrix beruhen. Durch Umordnung der Matrixelemente wird die überlappte Durchführung von lokalen Berechnungen und Transferzeiten nichtlokaler Daten unterstützt, um Wartezeiten zu reduzieren. Mit den entwickelten Methoden zur parallelen Matrix-Vektor-Multiplikation stehen effiziente Routinen zur Verfügung, die leicht in iterative Methoden integriert werden können.

Die Ermittlung der Datenverteilung und des Kommunikationsschemas sowie die Umordnung der Matrix-Elemente werden parallel in einem Vorverarbeitungsschritt durchgeführt. Verglichen mit der Ausführungszeit der iterativen Methode ist der Aufwand für die Vorverarbeitung gewöhnlich gering. Die Resultate der Vorverarbeitung sind in FE-Modellen solange verwendbar, bis sich das Beset-

zungsmuster der Matrix ändert.

Die Anzahl der Synchronisationspunkte in den entwickelten parallelen iterativen Verfahren wurde durch Änderung der Reihenfolge der Operationen der Iteration und durch die Verwendung modifizierter Verfahren reduziert. Im CG-Verfahren und der Lanczos-Tridiagonalisierung ist lediglich an einer Stelle globale Synchronisation erforderlich, in den Methoden QMR und TFQMR an drei Stellen.

Durch polynomiale Vorkonditionierung in Verbindung mit der Skalierung der Matrix konnte die Konvergenz des CG-Verfahrens deutlich beschleunigt und das Fehlerverhalten der Methode verbessert werden. Zusätzlich sinkt durch polynomiale Vorkonditionierung der Anteil des Synchronisationsaufwands am Gesamtaufwand eines Iterationsschritts; die Effizienz der Methode bei hoher Prozessorzahl steigt gegenüber dem Verfahren mit alleiniger Skalierung der Matrix.

Die Methoden QMR und TFQMR zeigen ein ähnliches Skalierungsverhalten wie das CG-Verfahren. Durch Kopplung des Datenaustauschs für die unabhängigen Matrix-Vektor-Multiplikationen mit der Koeffizientenmatrix und ihrer Transponierten wird die Anzahl der Nachrichten in der QMR-Iteration reduziert.

In dem entwickelten parallelen Lanczos-Verfahren zur Lösung des reell symmetrischen Eigenwertproblems sind zwei Algorithmen mit unterschiedlichen Parallelisierungsstrategien kombiniert, die Tridiagonalisierung und der Algorithmus zur Lösung des entstehenden tridiagonalen Eigenwertproblems. Die Verknüpfung beider Methoden führt zu einer hohen Effizienz des gesamten Verfahrens.

Die entwickelten parallelen CG-Verfahren wurden erfolgreich in FE-Modelle aus der Umwelttechnik und der Strukturmechanik der Forschungszentrum Jülich GmbH integriert. Darüber hinaus weisen erste Untersuchungen zum Einbau des QMR-Verfahrens in das Stofftransportmodell des Projekts aus der Umwelttechnik und zur Verwendung der Methoden für Eigenwertprobleme in der Molekulardynamik auf günstige Eigenschaften der Algorithmen in diesen Anwendungsbereichen hin.

Ferner wurden in Zusammenarbeit mit dem Lehrstuhl für Informatik II der RWTH Aachen einige Methoden zur parallelen Matrix-Vektor-Multiplikation mit dünnbesetzter Matrix als algorithmische Skelette in eine funktionale Programmiersprache integriert. Die algorithmischen Skelette sind als effiziente Bausteine für iterative Methoden in technischen Problemstellungen verwendbar und ermöglichen eine Programmierung auf hohem Abstraktionsniveau.

Ziel weiterer Untersuchungen ist die Konvergenzbeschleunigung der Verfahren QMR und TFQMR durch Vorkonditionierung, die auch zu einer Verringerung des Synchronisationsaufwands führen kann. Ferner können zur weiteren Reduzierung der Anzahl der Synchronisationspunkte jedes Iterationsschritts u. U. Ansätze aus Mehrschrittiterationsverfahren genutzt werden [136].

Die in dieser Arbeit vorgestellten Methoden sind auch zur Parallelisierung iterativer Verfahren für unsymmetrische oder verallgemeinerte symmetrische Eigenwertprobleme geeignet. Zur Lösung dieser Probleme können entsprechende

Lanczos-Algorithmen verwendet werden [16] [85].

Weiterhin ist die Kombination der beschriebenen Parallelisierungsstrategien mit einer Schwarz-Methode zur Gebietszerlegung möglich. Bei Schwarz-Verfahren wird das Diskretisierungsgitter — z. B. eines FE-Modells — in Teilgebiete zerlegt, auf denen jeweils die zugrundeliegende Differentialgleichung gelöst wird [111]. Zur Bestimmung der globalen Lösung werden iterativ Randwerte zwischen den Teilgebieten ausgetauscht. Schwarz-Verfahren bieten gute Parallelisierungseigenschaften bei gewöhnlich vernachlässigbarem Kommunikationsaufwand, jedoch sinkt die Konvergenzgeschwindigkeit der Verfahren bei zu vielen Teilgebieten rapide. Eine Zerlegung in wenige Teilgebiete hingegen führt in der Regel zu akzeptabler Konvergenz. Auf einem Parallelrechner mit hierarchischer Topologie kann jeweils ein Teilgebiet Gruppen von Prozessoren zugeteilt werden. Für die parallelen Berechnungen auf den Teilgebieten können die Methoden dieser Arbeit verwendet werden, während die globale Lösung des Problems durch eine Schwarz-Iteration ermittelt wird. Die Eignung eines parallelen Schwarz-Verfahrens für das betrachtete FE-Modell aus der Umwelttechnik wurde bereits untersucht [141].

Eine sinnvolle Erweiterung der in eine funktionale Sprache integrierten Methoden für dünnbesetzte Matrizen ist die automatische Auswahl des Speicherschemas und des Verfahrens der Matrix-Vektor-Multiplikation in Abhängigkeit von der Besetzungsstruktur der Matrix. Diese Auswahl kann anhand von Kennwerten der Matrix, die Eigenschaften des Besetzungsmusters angeben, gesteuert werden.

Literaturverzeichnis

[1] High Performance Fortran Language Specification (version 1.0): High Performance Fortran Forum. Center for Research on Parallel Computation, Rice University, Rice 1993.

[2] Intel Supercomputer Systems Division, Beaverton, Oregon. *iPSC/860 Fortran Compiler User's Guide*, January 1993. Order Number: 312131-003.

[3] Intel Supercomputer Systems Division, Beaverton, Oregon. *iPSC/860 System User's Guide*, March 1992. Order Number: 312312-001.

[4] Intel Supercomputer Systems Division, Beaverton, Oregon. *Paragon Fortran Compiler User's Guide*, March 1994. Order Number: 312491-002.

[5] Intel Supercomputer Systems Division, Beaverton, Oregon. *Paragon Fortran Compiler, Release 4.5, Software Product Release Notes*, March 1994. Order Number: 313012-001.

[6] Intel Supercomputer Systems Division, Beaverton, Oregon. *Paragon User's Guide*, October 1993. Order Number: 312489-002.

[7] Intel Supercomputer Systems Division, Beaverton, Oregon. *Paragon User's Guide*, June 1994. Order Number: 312489-003.

[8] SMART, Benutzerhandbücher. Institut für Statik und Dynamik der Luft- und Raumfahrtkonstruktionen der Universität Stuttgart. *ISD-Berichte*, 1976–1992.

[9] The Math Works Inc., Natick, MA. *MATLAB User's Guide*, August 1992.

[10] E. Anderson et al. *LAPACK user's guide.* SIAM, Philadelphia, 1992.

[11] S.F. Ashby. Minimax polynomial preconditioning for Hermitian linear systems. *SIAM J. Matrix Anal. Appl.*, 12:766–789, 1991.

[12] S.F. Ashby, T.A. Manteuffel, J.S. Otto. A comparison of adaptive Chebyshev and least squares polynomial preconditioning for Hermitian positive definite linear systems. *SIAM J. Sci. Statist. Comput.*, 13:1–29, 1992.

[13] O. Axelsson, A. Barker. *Finite Element Solution of Boundary Value Problems. Theory and Computation.* Academic Press, Orlando, Fl., 1984.

[14] C. Aykanat, F. Özgüner, D.S. Scott. Vectorization and parallelization of the conjugate gradient algorithm on hypercube-connected vector processors. *Microprocessing and Microprogramming*, 29:67–82, 1990.

[15] V.A. Barker, C. Yingqun. LANSYM: A Fortran subroutine for computing eigensolutions of symmetric sparse matrices on the Connection Machine. Report NI-92-12, Institute for Numerical Analysis, Technical University of Denmark, Lyngby, December 1992.

[16] V.A. Barker, C. Yingqun. LANUSM: A Fortran subroutine for computing eigenvalues of unsymmetric sparse matrices on the Connection Machine. Report NI-93-03, Institute for Numerical Analysis, Technical University of Denmark, Lyngby, May 1993.

[17] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, 1993.

[18] A. Basermann. Multisektionsverfahren zur Bestimmung der Eigenwerte von Tridiagonalmatrizen auf CRAY-Multiprozessorrechnern. Bericht Jül-2427, Forschungszentrum Jülich GmbH, ZAM, Jülich, Januar 1991.

[19] A. Basermann. Operationen mit dünnbesetzten Matrizen auf Parallelrechnern. In Forschungsprojekte des Graduiertenkollegs "Informatik und Technik", Aachener Informatik-Berichte 92-16, 1992.

[20] A. Basermann. Ein Kommunikationsschema für parallele CG-Verfahren zur Lösung von Gleichungssystemen mit dünnbesetzter Koeffizientenmatrix aus FE-Anwendungen. *Mitteilungen — Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen*, Nr. 11:62–69, 1993.

[21] A. Basermann. Datenverteilungs- und Kommunikationsmodelle für parallele CG-Verfahren zur Lösung von Gleichungssystemen mit dünnbesetzter Koeffizientenmatrix aus FE-Anwendungen. In Graduiertenkolleg Informatik und Technik, Aachener Informatik-Berichte 93-7, 1993.

[22] A. Basermann. Conjugate gradients parallelized on the hypercube. *International Journal of Modern Physics C*, Vol. 4, No. 6:1295–1306, 1993.

[23] A. Basermann. Data distribution and communication schemes for solving sparse linear systems of equations from FE applications by parallel CG methods. Informatik-Bericht 94/1:155–176, 1994, TU Clausthal, Institut für

Informatik, Proceedings of the Workshop on Parallel Processing, September 20–24, 1993, Lessach, Austria.

[24] A. Basermann. Parallelizing iterative solvers for sparse systems of equations and eigenproblems on distributed-memory machines. Extended abstract, in T. Manteuffel and S. McCormick, editors, *Proceedings of the Colorado Conference on Iterative Methods*, volume 2, University of Colorado, Breckenridge, April 1994.

[25] A. Basermann. Parallelizing iterative solvers for sparse systems of equations and eigenproblems on distributed memory machines. Interner Bericht KFA-ZAM-IB-9411, 22 Seiten, Jülich, Mai 1994. Submitted to SIAM Journal on Scientific Computing.

[26] A. Basermann. Parallel sparse matrix computations in iterative solvers on distributed memory machines. To appear in the Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, February 15-17, 1995.

[27] A. Basermann, C. Schelthoff, P. Weidner. Optimized kernels for the solution of partial differential equations on distributed memory machines. APPARC PaA3b Deliverable, ESPRIT BRA III Contract # 6634, 1994.

[28] A. Basermann, P. Weidner. A parallel algorithm for determining all eigenvalues of large real symmetric tridiagonal matrices. *Parallel Computing*, 18:1129–1141, 1992.

[29] A. Basermann, P. Weidner, P.C. Hansen, T. Ostromsky, Z. Zlatev. Reordering of sparse matrices for parallel processing. APPARC PaA3a Deliverable, ESPRIT BRA III Contract # 6634, Report UNIC-94-03, Lyngby, Denmark, February 1994.

[30] R. Berrendorf, H.C. Burg, U. Detert, R. Esser, M. Gerndt, R. Knecht. Intel Paragon XP/S — architecture, software environment, and performance. Interner Bericht KFA-ZAM-IB-9409, Jülich, Mai 1994.

[31] R. Berrendorf, U. Detert, J. Docter, U. Ehrhart, M. Gerndt, I. Gutheil, R. Knecht. Massively parallel computing in a production environment iPSC/860 installation at KFA Jülich. Interner Bericht KFA-ZAM-IB-9304, Jülich, Februar 1993.

[32] R. Berrendorf, J. Helin. Evaluating the basic performance of the Intel iPSC/860 parallel computer. *Concurrency — Practice and Experience*, 4(3):223–240, 1992.

[33] T. Bönniger, R. Esser, D. Krekel. CRAY Y-MP, NEC SX-3 und Siemens VP-S — Vergleichende Darstellung von Höchstleistungsrechnern. *Wirtschaftsinform.*, 3:273–286, 1990.

[34] T. Bönniger, R. Esser, D. Krekel. CM-5, KSR1, Paragon XP/S: A comparative description of massively parallel computers on the basis of a catalog of classifying characteristics. Interner Bericht KFA-ZAM-IB-9320, Jülich, Oktober 1993.

[35] R.H. Bisseling. *Parallel Iterative Solution of Sparse Linear Systems on a Transputer Network.* In A.E. Fincham and B. Ford, editors, Parallel Computation Proceedings IMA Conference on Parallel Computation, Oxford, UK, September 1991, Oxford University Press, Oxford, 1993.

[36] R.H. Bisseling, W.F. McColl. Scientific computing on bulk synchronous parallel architectures. Preprint nr. 836, University Utrecht, Department of Mathematics, 1993.

[37] M. Bücker. Parallelisierung der QMR-Methode zur Lösung linearer Gleichungssysteme. Bericht Jül-2955, Forschungszentrum Jülich GmbH, ZAM, Jülich, August 1994.

[38] M. Bücker, A. Basermann. A comparison of QMR, CGS and TFQMR on a distributed memory machine. Interner Bericht KFA-ZAM-IB-9412, Jülich, Mai 1994. Submitted to SIAM Journal on Scientific Computing.

[39] H.C. Burg. Anwendungen kombinatorischer Versuchspläne in der Parallelverarbeitung. Bericht Jül-2813, Forschungszentrum Jülich GmbH, ZAM, Jülich, September 1993.

[40] N. Carriero, D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[41] A. Chronopoulos, C. Gear. s-step iterative methods for symmetric linear systems. *J. Comput. Appl. Math.*, 25:153–168, 1989.

[42] L. Collatz. *Eigenwertaufgaben mit technischen Anwendungen.* Akademische Verlagsgesellschaft Geest & Portig K.-G., Leipzig, 1963.

[43] J.K. Cullum. Peaks, plateaus, numerical instabilities, and achievable accuracy in Galerkin and norm minimizing procedures for solving $Ax = b$. In T. Manteuffel and S. McCormick, editors, *Proceedings of the Colorado Conference on Iterative Methods*, volume 1, University of Colorado, Breckenridge, April 1994.

[44] J.K. Cullum, R.A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations.* Volume I: Theory, Birkhäuser, Boston Basel Stuttgart, 1985.

[45] J.K. Cullum, R.A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations.* Volume II: Programs, Birkhäuser, Boston Basel Stuttgart, 1985.

[46] R.D. da Cunha, T. Hoppkins. PIM 1.0: The parallel iterative methods package for systems of linear equations user's guide. Technical Report, University of Kent, Canterbury, 1994.

[47] R. Das, D.J. Mavriplis, J. Saltz, S. Gupta, R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. ICASE Report No. 92-12, Hampton, 1992.

[48] S. Das, J. Saltz, H. Berryman. A manual for PARTI runtime primitives. ICASE Interim Report 17, Hampton, 1991.

[49] E. D'Azevedo, C. Romine. Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors. Tech. Report ORNL/TM-12192, Oak Ridge National Lab, Oak Ridge, TN, 1992.

[50] P. Deuflhard, A. Hohmann. *Numerische Mathematik.* de Gruyter, 1991.

[51] J. Dongarra, C. Moler, J. Bunch, G. Stewart. *LINPACK User's Guide.* SIAM, Philadelphia, 1979.

[52] M. Dowell, P. Jarratt. The "PEGASUS" method for computing the root of an equation. *BIT*, 12:503–508, 1972.

[53] I.S. Duff, A.M. Erisman, J.K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.

[54] I.S. Duff, R.G. Grimes, J.G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Soft.*, 15(1):1–14, 1989.

[55] I.S. Duff, R.G. Grimes, J.G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection, release I. Technical Report TR/PA/92/86, CERFACS, Toulouse Cedex, October 1992.

[56] M. Eiermann. Preconditioners for iterative methods: An overview. *EASI-Workshop on Scientific Computing*, Braunschweig, November 1991.

[57] V. Eijkhout. LAPACK working note 50: Distributed sparse data structures for linear algebra operations. Tech. Report CS 92-169, Computer Science Department, University of Tennessee, Knoxville, TN, 1992.

[58] V. Eijkhout. LAPACK working note 51: Qualitative properties of the conjugate gradient and Lanczos methods in a matrix framework. Tech. Report CS 92-170, Computer Science Department, University of Tennessee, Knoxville, TN, 1992.

[59] D. Ellison. *Understanding Occam and the Transputer.* Sigma Press, 1991.

[60] N. Feistl. Das Verfahren der konjugierten Gradienten auf Vektorrechnern. Diplomarbeit, Ludwig-Maximilians-Universität, München, Juli 1990.

[61] P. Fernandes, P. Girdinio. A new storage scheme for an efficient implementation of the sparse matrix-vector product. *Parallel Computing*, 12:327–333, 1989.

[62] A.J. Field, P.G. Harrison. *Functional Programming.* Addison-Wesly Publishing Companny, 1988.

[63] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[64] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, W. Wu. Fortran D language specification. Technical Report COMP TR90079, Department of Computer Science, Rice University, Houston 1991.

[65] R.W. Freund. Quasi-Kernel Polynomials and Convergence Results for Quasi-Minimal Residual Iterations. In D. Braess and L.L. Shumaker, editors, *Numerical Methods of Approximation Theory*, volume 9, pages 77–95, Birkhäuser, Basel, 1992.

[66] R.W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM Journal on Scientific Computing*, 14(2):470–482, 1993.

[67] R.W. Freund. Block quasi-minimal residual iterations for non-Hermitian linear systems. In T. Manteuffel and S. McCormick, editors, *Proceedings of the Colorado Conference on Iterative Methods*, volume 2, University of Colorado, Breckenridge, April 1994.

[68] R.W. Freund, N.M. Nachtigal. QMR: A quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–339, 1991.

[69] G.H. Golub, D. O'Leary. Some history of the conjugate gradient and Lanczos methods. *SIAM Rev.*, 31:50–102, 1989.

[70] G.H. Golub, C.F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, second edition, 1989.

[71] W. Gropp, B. Smith. Simplified linear equation solvers users manual. Technical Report ANL-93/8-REV 1, Argonne National Laboratory, Argonne, 1993.

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

[73] W. Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme.* Teubner Studienbücher, Stuttgart, 1991.

[74] H. Hellwagner. A survey of virtually shared memory schemes. TUM-I9056, SFB-Bericht Nr.342/33/90 A, TU München, 1990.

[75] B. Hendrickson, R. Leland. A user's guide to the graph partitioning code Chaco. Technical Report SAND93-2339, Sandia National Laboratories, Albuquerque, NM, 1993.

[76] M.A. Heroux, P. Vu, C. Yang. A parallel preconditioned conjugate gradient package for solving sparse linear systems on a Cray Y-MP. *Applied Numerical Mathematics*, 8:93–115, 1991.

[77] M.R. Hestenes, E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[78] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.

[79] R.W. Hockney, C.R. Jesshope. *Parallel Computers 2.* Adam Hilger, Bristol, second edition, 1988.

[80] F. Hoßfeld. *Parallele Algorithmen.* Informatik Fachberichte 64, Springer, Heidelberg, 1983.

[81] T.J.R. Hughes. *The Finite Element Method.* Prentice-Hall, Englewood Cliffs, 1987.

[82] L.H. Jamieson. Characterizing Parallel Algorithms. In L.H. Jamieson, D.B. Gannon, R.J. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 66–93, MIT Press, Cambridge, MA, 1987.

[83] K. Jensen, N. Wirth. *Pascal, User Manual and Report.* Springer, New York Heidelberg Berlin, 1978.

[84] O. Johnson, C. Micchelli, G. Paul. Polynomial preconditioning for conjugate gradient calculations. *SIAM J. Numer. Anal.*, 20:363–376, 1983.

[85] M.T. Jones, M.L. Patrick. The Lanczos algorithm for the generalized symmetric eigenproblem on shared-memory architectures. *Applied Numerical Mathematics*, 12:377–389, 1993.

[86] W. Joubert. PCG: A software package for the iterative solution of linear systems on scalar, vector and parallel computers. In T. Manteuffel and S. McCormick, editors, *Proceedings of the Colorado Conference on Iterative Methods*, volume 1, University of Colorado, Breckenridge, April 1994.

[87] S.K. Kim, A.T. Chronopoulos. A class of Lanczos-like algorithms implemented on parallel computers. *Parallel Computing*, 17:763–778, 1991.

[88] D.R. Kincaid, L.J. Hayes, D.M. Young. ITPACK project: Past, present, and future. In T. Manteuffel and S. McCormick, editors, *Proceedings of the Colorado Conference on Iterative Methods*, volume 1, University of Colorado, Breckenridge, April 1994.

[89] C.P. Kruskal, L. Rudolph, M. Snir. Techniques for parallel manipulation of sparse matrices. *Theoretical Computer Science*, 64:135–157, 1989.

[90] D.J. Kuck, D.D. Gajski. Parallel Processing of Sparse Structures. In J.S. Kowalik, editor, *High-Speed Computation*, NATO ASI Series, volume F7, pages 229–244, Springer, Berlin Heidelberg, 1984.

[91] H.T. Kung. The Structure of Parallel Algorithms. In M.C. Yovits, editor, *Advances in Computers*, volume 19, pages 65–112, Academic Press, New York, 1980.

[92] C. Lanczos. Solutions of systems of linear equations by minimized iterations. *Journal of Research of the National Bureau of Standards*, 49:33–53, 1952.

[93] O. Lange, G. Stegemann. *Datenstrukturen und Speichertechniken.* Vieweg, Braunschweig Wiesbaden, 2. Auflage, 1987.

[94] J.G. Lewis, D.G. Payne, R.A. van de Geijn. Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers. *Proceedings of the Intel Supercomputer User's Group*, 1993 Annual North America User's Conference, October 3–6, St. Louis, Missouri, 1993.

[95] J.G. Lewis, R.A. van de Geijn. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. *Proceedings of Supercomputing '93*, Portland, OR, November 15–19, 1993.

[96] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* Ph.D. Thesis, Yale Univ., 1986.

[97] O.A. McBryan, E.F. Van de Velde. Matrix and vector operations on hypercube parallel processors. *Parallel Computing*, 5:117–125, 1987.

[98] R. Melhem. Toward efficient implementation of preconditioned conjugate gradient methods on vector supercomputers. *Internat. J. Supercomput. Appls.*, 1:77–98, 1987.

[99] R. Melhem. Determination of stripe structures for finite element matrices. *SIAM J. Numer. Anal.*, 24(6):1419–1433, 1987.

[100] R. Melhem. Parallel solution of linear systems with striped sparse matrices. *Parallel Computing*, 6:165–184, 1988.

[101] G. Meurant. Multitasking the conjugate gradient method on the CRAY X-MP/48. *Parallel Computing*, 5:267–280, 1987.

[102] A. Mlynski-Wiese. Leistungsuntersuchung des iPSC/860 RISC-Knoten-Prozessors: Architekturanalyse und Programmoptimierung. Bericht Jül-2766, Forschungszentrum Jülich GmbH, ZAM, Jülich, Mai 1993.

[103] N.M. Nachtigal. A look-ahead variant of TFQMR. In T. Manteuffel and S. McCormick, editors, *Proceedings of the Colorado Conference on Iterative Methods*, volume 2, University of Colorado, Breckenridge, April 1994.

[104] J.M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, New York London, 1988.

[105] B.N. Parlett. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, 1980.

[106] A. Peters. Sparse matrix vector multiplication techniques on the IBM 3090 VF. *Parallel Computing*, 17:1409–1424, 1991.

[107] G. Pini, G. Gambolati. Is a simple diagonal scaling the best preconditioner for conjugate gradients on supercomputers? *Adv. Water Resources*, 13:147–153, 1990.

[108] S. Pissanetsky. *Sparse Matrix Technology.* Academic Press, London Orlando, 1984.

[109] C. Pommerell, R. Rühl. Migration of vectorized iterative solvers to distributed memory architectures. In T. Manteuffel and S. McCormick, editors, *Proceedings of the Colorado Conference on Iterative Methods*, volume 1, University of Colorado, Breckenridge, April 1994.

[110] T.J. Rivlin. *The Chebyshev Polynomials.* John Wiley, 1974.

[111] G. Rodrigue. Inner/outer iterative methods and numerical Schwarz algorithms. *Parallel Computing*, 2:205–218, 1985.

[112] S. Rödder. Aufbau und Eigenschaften von Hypercube-Rechnern und Untersuchung von Datentransferalgorithmen. Bericht Jül-2428, Forschungszentrum Jülich GmbH, ZAM, Jülich, Januar 1991.

[113] Y. Saad. Practical use of some Krylov subspace methods for solving indefinite and nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 5:203–228, 1984.

[114] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM J. Sci. Statist. Comput.*, 6:865–881, 1985.

[115] Y. Saad. Preconditioning techniques for indefinite and nonsymmetric linear systems. *J. Comput. Appl. Math.*, 24:89–105, 1988.

[116] Y. Saad. Krylov subspace methods on supercomputers. *SIAM J. Sci. Statist. Comput.*, 10:1200–1232, 1989.

[117] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[118] Y. Saad. *Numerical Methods for Large Eigenvalue Problems.* Manchester University Press, Manchester, 1992.

[119] Y. Saad. P_SPARSLIB: A parallel sparse iterative solution package. In T. Manteuffel and S. McCormick, editors, *Proceedings of the Colorado Conference on Iterative Methods*, volume 1, University of Colorado, Breckenridge, April 1994.

[120] Y. Saad, M.H. Schulz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[121] C. Schelthoff. Vergleich von parallelen Verfahren zur Vorkonditionierung für die Methode der konjugierten Gradienten. Bericht Jül-2913, Forschungszentrum Jülich GmbH, ZAM, Jülich, März 1994.

[122] C. Schelthoff, A. Basermann. Polynomial preconditioning for the conjugate gradient method on massively parallel systems. Interner Bericht KFA-ZAM-IB-9423, Jülich, Oktober 1994. Erscheint in Informatik-Berichte, TU Clausthal, Institut für Informatik, Proceedings of the Workshop on Parallel Processing, September 25–Oktober 1, 1994, Lessach, Austria.

[123] U. Schendel. *Sparse-Matrizen.* R. Oldenbourg Verlag, München Wien, 1. Auflage, 1977.

[124] R. Schneiders. *Remeshing-Algorithmen für dreidimensionale Finite-Element-Simulationen von Umformprozessen.* Dissertation, RWTH Aachen, Verlag der Augustinus Buchhandlung, Aachener Beiträge zur Informatik, Band 4, 1993.

[125] H.R. Schwarz. *FORTRAN-Programme zur Methode der finiten Elemente.* B. G. Teubner, Stuttgart, 1981.

[126] M.K. Seager. Parallelizing conjugate gradient for the CRAY X-MP. *Parallel Computing*, 3:35–47, 1986.

[127] J.N. Shadid, S.A. Hutchinson, H.K. Moffat. Parallel performance of a preconditioned CG solver for unstructured finite element applications. In T. Manteuffel and S. McCormick, editors, *Proceedings of the Colorado Conference on Iterative Methods*, volume 2, University of Colorado, Breckenridge, April 1994.

[128] J.N. Shadid, R.S. Tuminaro. Sparse iterative algorithm software for large-scale MIMD machines: An initial discussion and implementation. *Concurrency: Practice and Experience*, 4(6):481–497, 1992.

[129] J.N. Shadid, R.S. Tuminaro. A comparison of preconditioned nonsymmetric Krylov methods on a large-scale MIMD machine. *SIAM Journal on Scientific Computing*, 15(2):440–459, 1994.

[130] H.D. Simon. Partitioning of unstructured problems for parallel processing. Report RNR-91-008, NASA Ames Research Center, Moffett Field, 1991.

[131] B. Smith, W. Gropp. Portable, parallel, reusable Krylov space codes. In T. Manteuffel and S. McCormick, editors, *Proceedings of the Colorado Conference on Iterative Methods*, volume 1, University of Colorado, Breckenridge, April 1994.

[132] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 10(1):36–52, 1989.

[133] A. van der Sluis, H. van der Vorst. The rate of convergence of conjugate gradients. *Numer. Math.*, 48:543–560, 1986.

[134] R. Steinmetz. *OCCAM 2 — Die Programmiersprache für parallele Verarbeitung*. Dr. Hüthig Verlag, Heidelberg, 2. Auflage, 1988.

[135] H. Stoltze. *Implementierung einer parallelen funktionalen Sprache mit algorithmischen Skeletten zur Lösung mathematisch-technischer Probleme*. Dissertation , RWTH Aachen, Verlag Shaker, Aachen 1995.

[136] C.D. Swanson, A.T. Chronopoulos. Parallel iterative s-step methods for unsymmetric linear systems. University of Minnesota Supercomputer Institute Research Report UMSI 94/105, Minneapolis, Minnesota, 1994.

[137] R.P. Tewarson. *Sparse Matrices*. Academic Press, New York London, 1973.

[138] H.A. van der Vorst. High performance preconditioning. *Siam J. Sci. Stat. Comput.*, 10:1174–1185, 1989.

[139] H.A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *Siam J. Sci. Stat. Comput.*, 13(2):631–644, 1992.

[140] H. Vereecken, G. Lindenmayr, A. Kuhr, D.H. Welte, A. Basermann. Numerical modeling of field scale transport in heterogeneous variably saturated porous media. KFA/ICG-4 Internal Report No. 500393, Jülich, January 1993.

[141] H. Vereecken, O. Neuendorf, G. Lindenmayr, A. Basermann. A parallel Schwarz domain decomposition method for the numerical solution of transient water flow in heterogeneous porous media. Submitted to Parallel Computing.

[142] J.H. Wilkinson. *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, 1965.

[143] J.H. Wilkinson, C. Reinsch. *Handbook for Automatic Computation.* Volume II: Linear Algebra, Springer-Verlag, Berlin Heidelberg New York, 1971.

[144] J. Wu, J. Saltz, H. Berryman, S. Hiranandani. Distributed memory compiler design for sparse problems. Technical Report TR91-13, ICASE, Hampton, 1991.

[145] G.T. Yeh. 3DFEMWATER: A three-dimensional finite element model of water flow through saturated-unsaturated media. ORNL-6386, Oak Ridge National Laboratory, 1987.

[146] D.M. Young. A historical overview of iterative methods. *Computer Physics Communications*, 53:1–17, 1989.

[147] L.H. Ziantz, C.C. Özturan, B.K. Szymanski. *Run-time Optimization of Sparse Matrix-Vector Multiplication on SIMD Machines.* In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, PARLE '94: Parallel Architectures and Languages Europe, Springer, 1994.

[148] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald. Vienna Fortran — a language specification version 1.1. Technical Report Interim 21, ICASE, NASA, Hampton, 1992.

[149] Z. Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic Publishers, Dordrecht, 1991.

Anhang A

Symbole und Abkürzungen

Symbole

Allgemeine Bezeichnungen

$\mathbb{R}$	Die reellen Zahlen
$\boldsymbol{A}$	Matrix: Fettgedruckte, große lateinische Buchstaben bezeichnen Matrizen.
a_{ij}	Matrixelement der i-ten Zeile und der j-ten Spalte
n	Ordnung einer Matrix
$\boldsymbol{A}^T$	Transponierte der Matrix $\boldsymbol{A}$
$\boldsymbol{I}$	Einheitsmatrix
$\\|\boldsymbol{A}\\|_2$	Spektralnorm der Matrix $\boldsymbol{A}$
κ_2	Konditionszahl einer Matrix bezüglich der Spektralnorm
$\lambda_{\min}, \lambda_{\max}$	Kleinster bzw. größter Eigenwert einer Matrix
$\mathcal{C}(\boldsymbol{A}), \mathcal{P}(\boldsymbol{A})$	Matrixpolynome
P_m	Menge aller Polynome vom Grad m
x	Vektor: Vektoren werden durch kleine lateinische Buchstaben dargestellt.
x_j	j-te Komponente des Vektors x
$\\|x\\|_2$	Euklidische Norm des Vektors x
$\\|x\\|_\infty$	Maximumnorm des Vektors x
$e^{(k)}$	Erster Einheitsvektor aus $\mathbb{R}^k$; $e^{(k)} = (1, 0, \ldots, 0)^T$
$\alpha, \beta, \gamma, \delta, \ldots$	Skalare: Kleine griechische Buchstaben bezeichnen Skalare.

$\mathcal{O}(\ldots)$	Komplexitätsklasse
mod	Modulo-Operation
div	Ganzzahlige Division ohne Rest

Iterative Verfahren

b	Rechte Seite des Gleichungssystems $\boldsymbol{A}x = b$
$x^{(i)}$	Näherungslösung des Gleichungssystems $\boldsymbol{A}x = b$ im i-ten Iterationsschritt
$\{x^{(i)}\}$	Folge von Näherungslösungen
x^*	Exakte Lösung von $\boldsymbol{A}x = b$
$g^{(i)}$	Residuum der i-ten Iteration; $g^{(i)} = \boldsymbol{A}x^{(i)} - b$
$\alpha_i, \beta_i, \gamma_i, \delta_i, \ldots$	Skalare im i-ten Iterationsschritt
$(\lambda_k, z^{\{k\}})$	Paar aus Eigenwert λ_k und Eigenvektor $z^{\{k\}}$; eine Lösung des Eigenwertproblems $\boldsymbol{A}z = \lambda z$
$\boldsymbol{T}_i$	Tridiagonalmatrix der i-ten Iteration
$\boldsymbol{S}_{i-1}$	Tridiagonalmatrix der i-ten Iteration, die sich durch Streichen der ersten Zeile und Spalte von $\boldsymbol{T}_i$ ergibt
$(\mu_k, y^{\{k\}})$	Paar aus Eigenwert μ_k und Eigenvektor $y^{\{k\}}$ der Matrix $\boldsymbol{T}_i$
$\mathcal{K}(\boldsymbol{B}, z, i)$	Der i-te von $z \in \mathbb{R}^n$ und $\boldsymbol{B} \in \mathbb{R}^{n \times n}$ erzeugte Krylov-Teilraum
$\text{span}\left\{q^{(1)}, q^{(2)}, \ldots, q^{(i)}\right\}$	Menge aller Linearkombinationen der Vektoren $q^{(1)}, q^{(2)}, \ldots, q^{(i)}$
$\left[q^{(1)} q^{(2)} \cdots q^{(i)}\right]$	Matrix mit den Spaltenvektoren $q^{(1)}, q^{(2)}, \ldots, q^{(i)}$

Kenngrößen dünnbesetzter Matrizen

e	Anzahl der Nichtnull-Elemente
$z_{\mathbf{max}}$	Maximale Anzahl von Nichtnull-Elementen pro Zeile
$z_{\mathbf{mean}}$	Mittlere Anzahl von Nichtnull-Elementen pro Zeile; $z_{\mathbf{mean}} = e/n$
z_i	Anzahl der Nichtnull-Elemente in Zeile i
$s_{\mathbf{mean}}$	Mittlere Anzahl aufeinanderfolgender Spaltenindizes pro Nichtnull-Element und Zeile
$\tilde{e}$	Anzahl der im SICRS-Schema gespeicherten Index-Werte

e_{b}	Anzahl der Nichtnull-Blöcke im BCRS-Format
n_{b}	Dimension eines Nichtnull-Blocks
n_{d}	Block-Dimension der Matrix; $n_{\mathrm{d}} = n/n_{\mathrm{b}}$
n_{dia}	Anzahl der Diagonalen der Matrix, die Nichtnull-Elemente enthalten (DIA-Format)
n_{stripes}	Anzahl der abgelegten Streifen einer Matrix bei Streifenspeicherung
S_k	k-ter gespeicherter Streifen
$(i, \sigma_k(i))$	Zeilen- und Spaltenindex eines Elements von S_k

Parallele Methoden

p	Anzahl der Prozessoren
e_k	Anzahl der Nichtnull-Elemente von Prozessor k
g_k	Nummer der ersten Zeile (Spalte), die Prozessor k zugewiesen wird
n_k	Anzahl der Zeilen (Spalten) von Prozessor k
$n_{\max}$	Maximale Anzahl der Zeilen, die einem der Prozessoren zugeordnet sind
ξ	Parameter der Datenverteilung
a_{MVP}	Anteil der Matrix-Vektor-Multiplikationen am Gesamtaufwand eines Schritts eines iterativen Verfahrens
l_k	Lokaler Anteil von Prozessor k an der Berechnung eines Skalarprodukts
$x^{[k]}$	Segment des Vektors x einer Matrix-Vektor-Multiplikation, das Prozessor k zugeteilt ist
$y^{[k]}, z^{[k]}$	Segmente des Ergebnisvektors einer Matrix-Vektor-Multiplikation, die Prozessor k besitzt
$y^{[k,l]}$	Teilergebnis einer Matrix-Vektor-Multiplikation nach Neuverteilung der Blöcke, das auf Prozessor k berechnet wird und an Prozessor l gesendet werden muß
$p_{\mathrm{from},k}$	Anzahl der Prozessoren, von denen auf Prozessor k Daten empfangen werden
$p_{\mathrm{to},k}$	Anzahl der Prozessoren, an die von Prozessor k Daten gesendet werden

b_k	Anzahl der Blöcke von Prozessor k bei Blocksortierung
b_k^{neu}	Anzahl der Blöcke von Prozessor k nach der Neuverteilung der Blöcke
e_k^{neu}	Anzahl der Nichnull-Elemente von Prozessor k nach der Neuverteilung der Blöcke
$l_{\mathrm{nl},k}$	Anzahl der auf Prozessor k benötigten nichtlokalen Komponenten des Vektors der Matrix-Vektor-Multiplikation
$s_{\mathrm{mean},p}$	Mittlere Anzahl aufeinanderfolgender Spaltenindizes pro Nichnull-Element und Zeile bei p Prozessoren und Blocksortierung

Abkürzungen

ALLEV	All Eigenvalues
BCCS	Block Compressed Column Storage
BCRS	Block Compressed Row Storage
BiCG	Biconjugate Gradient
BiCGSTAB	Biconjugate Gradient Stabilized
CCS	Compressed Column Storage
CDS	Compressed Diagonal Storage
CFD	Computational Fluid Dynamics
CG	Conjugate Gradients
CGS	Conjugate Gradient Squared
CRS	Compressed Row Storage
CSP	Communicating Sequential Processes
DIA	Diagonal Format
FD	Finite Difference
FE	Finite Element
GCR	Generalized Conjugate Residual
GFLOPS	Giga Floating Point Operations per Second
GMRES	Generalized Minimal Residual
HPF	High Performance FORTRAN
JDS	Jagged Diagonal Storage
LANSP	Lanczos Sparse
MFLOPS	Mega Floating Point Operations per Second
MIMD	Multiple Instruction Stream Multiple Data Stream
MSC	Modified Sparse Column
MSR	Modified Sparse Row
OMIN	Orthomin
PARTI	Parallel Automated Runtime Toolkit at ICASE
QMR	Quasi-Minimal Residual
RCM	Reverse Cuthill-McKee
RISC	Reduced Instruction Set Computer
SICRS	Successive Index Compressed Row Storage
SKS	Skyline Storage
SOR	Successive Over-Relaxation
SSOR	Symmetric Successive Over-Relaxation
TFQMR	Transpose-Free Quasi-Minimal Residual

Dank

Die vorliegende Arbeit entstand im Zentralinstitut für Angewandte Mathematik (ZAM) der Forschungszentrum Jülich GmbH (KFA). Dem Institutsdirektor des ZAM und Inhaber des Lehrstuhls für technische Informatik und Computerwissenschaften an der RWTH Aachen, Herrn Prof. Dr. F. Hoßfeld, danke ich für die Betreuung dieser Arbeit.

Herrn Prof. Dr. K. Indermark, dem Inhaber des Lehrstuhls für Informatik II an der RWTH Aachen, gilt mein Dank für die Übernahme des Korreferats. Bei seinem Mitarbeiter Herrn H. Stoltze bedanke ich mich für eine ergiebige Zusammenarbeit hinsichtlich funktionaler Programmierung.

Herrn Dr. P. Weidner, dem Leiter der Abteilung Mathematik des ZAM, danke ich für hilfreiche Anregungen und Ratschläge im Verlauf der Arbeit.

Den Herren C. Schelthoff und M. Bücker gilt mein Dank für ihre Unterstützung und für viele nutzbringende Diskussionen. Bei meiner Bürokollegin Frau R. Zimmermann bedanke ich mich für zahlreiche Hilfen, insbesondere bei der Verwendung mathematischer Software.

Ferner danke ich allen Mitarbeitern des ZAM, die durch ihre Hilfsbereitschaft auf verschiedene Weise zum Gelingen dieser Arbeit beigetragen haben. Den Herren H. Bast von der Firma INTEL und R. Vogelsang von der Firma CRAY, die im ZAM arbeiten, danke ich für die Unterstützung bei der Rechnernutzung und für den Austausch von Erfahrungen.

Bei den Mitarbeitern des Instituts für Chemie und Dynamik der Geosphäre, des Instituts für Sicherheitsforschung und Reaktortechnik sowie des Instituts für Festkörperforschung der Forschungszentrum Jülich GmbH bedanke ich mich für ihre Kooperationsbereitschaft.

Weiterer Dank gilt meinen Kollegen aus dem Graduiertenkolleg "Informatik und Technik" der RWTH Aachen für die interessanten Diskussionen in der Arbeitsgemeinschaft an jedem Mittwochmorgen und in zahlreichen Veranstaltungen. Dem Vorsitzenden des Kollegs, Herrn Prof. Dr. O. Spaniol, danke ich für die erhaltene Unterstützung.

Besonders dankbar bin ich meiner Frau Andrea, die mich im Verlauf und während der Ausarbeitung dieser Arbeit tatkräftig unterstützt hat.