Fakultät für Mathematik
der Technischen Universität Dortmund

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

Kommunikationsvermeidende und asynchrone Verfahren zur Lösung dünnbesetzter linearer Gleichungssysteme auf modernen Höchstleistungsrechnern

vorgelegt von

Marcel Klinger

und betreut durch

Jun.-Prof. Dr. Dominik Göddeke

August 2012

„So eine Arbeit wird eigentlich nie fertig, man muß sie für fertig erklären, wenn man nach Zeit und Umständen das möglichste getan hat.“

– Johann Wolfgang von Goethe

Inhaltsverzeichnis

Notationen **7**

1 Einleitung **11**
1.1 Motivation und Überblick 11
1.2 Struktur der Arbeit 13
1.3 Hardware-orientierte Numerik 14

2 Kommunikationsvermeidung in Krylow-Unterraum-Methoden **19**
2.1 Krylow-Unterraum-Methoden 19
2.2 Pipelined CG-Verfahren 21
2.3 Matrixpotenzen-Kernel 23
2.3.1 Graphentheoretische Notationen 26
2.3.2 Exemplarische Realisierung 30
2.3.3 Allgemeine Realisierung 35
2.4 Kommunikationsvermeidung im CG-Verfahren 36
2.4.1 CG-Verfahren mit Drei-Term-Rekursion 37
2.4.2 Verbindung mit dem Matrixpotenzen-Kernel 39
2.4.3 Entfernen der Skalarprodukte 45
2.4.4 Fazit 49
2.5 Kommunikationsvermeidung im vorkonditionierten CG-Verfahren . 50
2.5.1 Vorkonditionierung 50
2.5.2 Vorkonditioniertes CG-Verfahren mit Drei-Term-Rekursion . 52
2.6 Kommunikationsvermeidung im BiCGStab-Verfahren 59
2.7 Wahl der Basis und Stabilität 64

3 Asynchrone Verfahren **67**
3.1 Asynchrone Fixpunktiterationsverfahren 68
3.1.1 Klassische Fixpunktiterationsverfahren 68
3.1.2 Asynchrone Fixpunktiterationsverfahren 71
3.2 Block-Jacobi-Vorkonditionierung 77
3.3 FGMRES(m)-Verfahren 78
3.4 Ein asynchrones Verfahren 82
3.4.1 Synchrone Tests 84
3.4.2 Asynchrone Tests 101
3.5 Fazit und Ausblick 113

A Definitionen **117**

B Sätze **119**

Literaturverzeichnis **121**

Index **125**

Eidesstattliche Versicherung **127**

Notationen

Lineare Algebra Grundsätzlich wollen wir in dieser Arbeit der sogenannten *Householder-Notation* für lineare Algebra folgen. Dies bedeutet, wir werden skalare Größen mit griechischen und Vektoren wie auch Matrizen mit lateinischen Buchstaben bezeichnen. Für Skalare und Vektoren werden wir uns der Kleinschreibung, für Matrizen der Großschreibung bedienen. Eine Ausnahme bildet dabei die Definition einer Matrix über ihre konkreten skalaren Einträge. Diese bennenen wir in der Regel mit dem mit der Matrix assoziierten Buchstaben in Kleinschreibweise, etwa $V := (v_{i,j}) \in \mathbb{R}^{n \times m}$. Auf Vektorpfeile oder Ähnliches verzichten wir überdies gänzlich.

Algebraisch unterscheiden wir einspaltige Matrizen und Vektoren ebenfalls nicht. Entsprechend verfahren wir mit (1×1)-Matrizen: Solche Matrizen betrachten wir als skalare Größen und umgekehrt. Ob nun ein Groß- oder Kleinbuchstabe beziehungsweise ein griechischer oder lateinischer verwendet wird, ergibt sich aus dem Sachzusammenhang: Ergibt sich etwa eine vektorielle Größe aus einem Entartungsfall einer echten Matrix, werden wir auch weiter einen Großbuchstaben zur Nomenklatur heranziehen.

Sollten wir nur Teilvektoren beziehungsweise Teilmatrizen betrachten, bedienen wir uns diesbezüglich der folgenden Notationen: Benötigen wir nur einen einzelnen Eintrag eines Vektors v beziehungsweise einer Matrix V, sprechen wir diesen über die Notation $v(i) = v_i$ beziehungsweise $V(i,j) = V_{i,j}$ an. Mit $V(:,j)$ beziehungsweise $V(i,:)$ bezeichnen wir die j-te Spalte beziehungsweise i-te Zeile der Matrix V. Zu beachten ist dabei, dass im ersten Fall ein „stehender“ Vektor, also eine $(n \times 1)$-Matrix, im letzten Fall ein „liegender“ Vektor, also eine $(1 \times m)$-Matrix, resultiert, falls $V \in \mathbb{R}^{n \times m}$ gilt. Um einzelne Teilbereiche von Vektoren oder Matrizen auszuschneiden, verwenden wir eine weitere Notation: $V(i_1 : i_2, j_1 : j_2)$ bezeichnet etwa für $0 < i_1 \leq i_2 < n$ und $0 < j_1 \leq j_2 < m$ die $(i_2 - i_1 + 1 \times j_2 - j_1 + 1)$-

Teilmatrix von V, die durch Vernachlässigung aller Zeilen i mit $i < i_1$ sowie aller Zeilen i mit $i > i_2$ und aller Spalten j mit $j < j_1$ sowie aller Spalten j mit $j > j_2$ entsteht. Ferner treffen wir die Konventionen $V(i_1 : i_2, j) := V(i_1 : i_2, j : j)$ und $V(i, j_1 : j_2) := V(i : i, j_1 : j_2)$.

Für die $(n \times m)$-Matrix, die nur aus Nullen besteht, verwenden wir die Bezeichnung $0_{n,m}$. Wir treffen zudem die Definition $0_n := 0_{n,1}$. Mit e_i bezeichnen wir darüber hinaus den kanonischen Einheitsvektor, der bis auf den i-ten Eintrag nur aus Nullen besteht. Der i-te Eintrag selbst hingegen ist somit eine Eins. Die Dimension des Vektors sollte zudem jeweils aus dem Kontext hervorgehen. Mit I_n bezeichnen wir die spaltenweise aus aufsteigend indizierten Einheitsvektoren bestehende Einheitsmatrix, beginnend bei e_1: $I_n := [e_1, \ldots, e_n]$. Sollte die Dimension wieder aus dem Kontext hervorgehen, unterdrücken wir den Index n und schreiben lediglich I.

Geben wir Matrizen direkt über ihre Einträge an, verwenden wir eckige Klammern $[\cdot]$. Dabei ist es außerdem möglich, dass wir Matrizen blockweise durch bereits bekannte Matrizen angeben. Die Schreibweise $V = [M, N]$ stellt etwa die Matrix V dar, die aus den nebeneinander stehenden Matrizen M und N besteht, wobei wir voraussetzen, dass M und N die gleiche Anzahl an Zeilen besitzen. Nutzen wir in diesem Kontext unbestimmtere Symbole, etwa I für die Einheitsmatrix, gehen wir, wie bereits erwähnt, davon aus, dass die Dimension „passend“ gewählt wird und aus dem Kontext hervorgeht, etwa durch die Höhe einer nebenstehenden Matrix.

Es sei schließlich noch auf die üblichen zur Erörterung unserer Theorien notwendigen Begriffe innerhalb der linearen Algebra verwiesen. Mit dem *Bild einer Matrix* V bezeichnen wir etwa den *Spann* beziehungsweise die *lineare Hülle* ihrer Spalten, das heißt in Zeichen

$$\begin{aligned} \operatorname{image}(V) &:= \operatorname{span}\{V(:,1), \ldots, V(:,m)\} \\ &:= \{\lambda_1 V(:,1) + \ldots + \lambda_m V(:,m) \ : \ \lambda_1, \ldots, \lambda_m \in \mathbb{R}\}. \end{aligned}$$

Das *euklidische Skalarprodukt* sowie die *euklidische Norm* sind für uns ferner wie üblich für Vektoren $v, w \in \mathbb{R}^n$ definiert als

$$(v, w)_2 := \sum_{i=1}^{n} v_i w_i$$

beziehungsweise

$$||v||_2 := \sqrt{(v,v)_2}.$$

Indizierung Indizieren wir eine vektorielle Größe v beziehungsweise eine Matrix V im Subskript, also etwa v_i beziehungsweise $V_{i,j}$ sprechen wir darüber die entsprechenden Einträge an (vgl. oben). Hierbei wird es aus naheliegenden Gründen nicht zu Konflikten mit den obigen Definitionen bezüglich 0_n sowie e_i kommen. In wenigen Ausnahmefällen werden wir Indizes im Subskript jedoch auch benutzen, um eine Reihe von Vektoren zu bezeichnen, beispielsweise innerhalb einer Basis $v_1, \ldots, v_n$.

Hochgestellte in runde Klammern gesetzte Indizes markieren Iterationsschritte bei Vektoren und Skalaren, etwa $v^{(k)}$ für die Andeutung des k-ten Iterationsschritts. Um Iterationsschrittabhängigkeit bei Matrizen und Mengen auszudrücken, verwenden wir hingegen Indizes im Subskript, jedoch ohne runde Klammern, etwa V_k für die Andeutung des k-ten Iterationsschritts. Diese Schreibweisen werden wir im Kontext der Vorstellung diverser Algorithmen verwenden. Verwechslungen mit Potenzen sind bei den im Superskript indizierten Größen ausgeschlossen, da wir diese ohne die Verwendung von Klammern darstellen. Wird eine Iterierte potenziert oder transponiert, setzen wir die entsprechende Kennzeichnung in einem Supersuperskript-Niveau ohne eine zusätzliche Klammerung zu verwenden, etwa $\lambda^{(k)^2}$ für $\left(\lambda^{(k)}\right)^2$ oder $v^{(k)^\top}$ für $\left(v^{(k)}\right)^\top$.

Graph Ein *Graph* G ist für uns das geordnete Paar der Mengen V und E, also $G = (V, E)$, wobei wir die Elemente von V als *Knoten* (engl. *vertices*) und die Elemente von E als *Kanten* (engl. *edges*) bezeichnen. Ferner soll $E \subset V \times V$ gelten, das heißt wir verstehen Graphen immer als *gerichtet*. $(x, y) \in E$ meint dabei etwa die Kante von Knoten x nach Knoten y.

Wir werden in Abschnitt 2.3.1 weitere mit dieser Struktur verbundene Definitionen treffen.

Kapitel 1

Einleitung

1.1 Motivation und Überblick

In der Hardwareindustrie setzt sich ein signifikanter Anstieg an verfügbarer Rechenkraft unvermindert fort, wohingegen die erreichbare Kommunikationsgeschwindigkeit zwischen einzelnen Komponenten der Computersysteme deutlich weniger zunimmt. So verdoppelt sich derzeit etwa alle zwei Jahre die verfügbare Rechengeschwindigkeit, während Transfergeschwindigkeiten durch physikalische Schranken und das Energiebudget begrenzt sind (vgl. etwa Graham et al. [16] oder Kogge et al. [20]). Die Divergenz zwischen diesen Geschwindigkeitsbegrifflichkeiten ist gemein hin als *Memory-Wall-Problematik* bekannt. Sie erstreckt sich im Detail auf alle Ebenen einer Rechnerarchitektur, so etwa auf Zugriffe des Prozessors auf den Hauptspeicher oder aber auch auf den Datenaustausch zwischen einzelnen Rechenknoten in einem System mit verteiltem Speicher.

In seiner Konsequenz macht dieser Sachzusammenhang ein Umdenken in aktueller numerischer Algorithmik notwendig: Klassische Numerik berücksichtigt diesen Trend in der Hardware-Entwicklung nicht. Sie hat gewisse Grundvoraussetzungen an den Datenaustausch und damit verbundene Synchronisationspunkte, wie etwa globale Skalarprodukte, welche aufgrund ihres inhärenten Kommunikationsbedarfs mit Blick auf die Memory-Wall-Problematik kontraproduktiv sind.

Damit auch in Zukunft Rechnersysteme optimal genutzt werden können, müssen moderne numerische Verfahren möglichst große Anteile der Laufzeit auf voneinander unabhängige parallele Berechnungen aufwenden, während Kommunikati-

on zwischen einzelnen Rechnerkomponenten minimiert oder gar ganz unterlassen werden muss. Diese Überlegung führt auf sogenannte kommunikationsvermeidende beziehungsweise asynchrone Verfahren innerhalb der sogenannten Hardware-orientierten Numerik.

Konkret befassen wir uns aus diesem Grund mit derartigen Verfahren im Zusammenhang mit dem schnellen Lösen großer dünnbesetzter linearer Gleichungssysteme

$$Ax = b, \tag{1.1}$$

wobei A die Systemmatrix, x den Vektor der Unbekannten und b die rechte Seite darstellt. Solche Gleichungssysteme treten typischerweise in industriellen Anwendungen, insbesondere im Zusammenhang mit der Kontinuumsmechanik, auf und ergeben sich aus der Diskretisierung partieller Differentialgleichungen, etwa mittels der Finite-Differenzen- oder Finite-Elemente-Methode.

Auf der einen Seite untersucht diese Arbeit kommunikationsvermeidende Methoden, die aus klassischen Verfahren durch Umformulierungen der Algorithmik entstehen. Auf diese Weise können Kommunikationsanforderungen mitunter erheblich reduziert werden. So ist es etwa in vielen Krylow-Unterraum-Methoden, welche eine populäre Löserklasse für dünnbesetzte Gleichungssysteme bilden, möglich, Datenabhängigkeiten zwischen Matrix-Vektor- und Vektor-Vektor-Operationen zu umgehen und so die Kommunikationskosten für je s Iterationen von $\mathcal{O}(s)$ auf $1 + \mathcal{O}(1)$ zu reduzieren. Im Kern berufen wir uns hier auf die Veröffentlichungen von Carson et al. [5] und insbesondere Hoemmen [18]. Letztere bildet dabei für uns eine fundamentale Grundlage.

Auf der anderen Seite analysiert diese Arbeit Verfahren, die Kommunikation gänzlich zu unterlassen versuchen: So sind sogenannte asynchrone Block-Jacobi-Methoden in Bezug auf die reine Rechenperformance potentiell noch leistungsfähiger, jedoch ist mit ihnen ein gleichzeitiger Verlust der numerischen Qualitäten verbunden. Um abzuwägen, wie stark Kommunikation unterlassen werden kann, um die numerische Leistungsfähigkeit nicht überproportional zu verlieren, werden neben der theoretischen Analyse konkrete Implementierungen evaluiert. Wir verweisen ferner auf Veröffentlichungen der Autoren Frommer [13] sowie Frommer und Szyld [14].

1.2 Struktur der Arbeit

Zwecks einer erleichterten Lektüre wollen wir zunächst die Struktur dieser Arbeit beziehungsweise ihres Inhalts erläutern. Neben der obigen Zusammenfassung sowie einer nachstehenden Motivation des Forschungsgebiets der Hardware-orientierten Numerik gliedert sich diese Arbeit im Wesentlichen in zwei weitere Kapitel: In Kapitel 2 werden wir verschiedene kommunikationsvermeidende Verfahren durch eine Umformulierung klassischer Krylow-Unterraum-Methoden zur Lösung linearer Gleichungssysteme herleiten. Wir beginnen dabei zunächst mit einer kurzen Einführung in die Theorie solcher Verfahren (Abschnitt 2.1) und stellen exemplarisch mit dem sogenannten Pipelined CG-Verfahren eine erste an spezielle Hardware-Anforderungen angepasste Umformulierung einer Lehrbuch-Methode vor (Abschnitt 2.2). In Abschnitt 2.3 führen wir den Matrixpotenzen-Kernel ein, welcher ein elementares Bauteil unserer kommunikationsvermeidenden Methoden ist, und beschreiben ihn ausführlich. Eine erste kommunikationsvermeidende Methode, die den Matrixpotenzen-Kernel nutzt, resultiert schließlich in Abschnitt 2.4 aus dem Verfahren der konjugierten Gradienten. Da Krylow-Unterraum-Methoden in der Praxis meist mittels einer sogenannten Vorkonditionierung beschleunigt werden, erweitern wir in Abschnitt 2.5 alle Überlegungen des vorangegangenen Abschnitts auf die Verwendung solcher Vorkonditionierer und erläutern zunächst diese Technik im Allgemeinen. Einen weiteren Vertreter der Krylow-Unterraum-Methoden, das BiCGStab-Verfahren, bereiten wir in Abschnitt 2.6 auf die Verwendung des Matrixpotenzen-Kernels vor, so dass dieser ebenfalls in der Lage ist, kommunikationsvermeidend zu arbeiten. In allen Fällen werden sich unsere Bemühungen bezüglich der Verbesserung des Kommunikationsbedarfs der Verfahren jedoch nicht nur auf die Verwendung des Matrixpotenzen-Kernels beschränken: Wir werden ebenso die Vermeidung gewisser Skalarprodukte, die globaler Kommunikation bedürfen, durch Umformulierung anstreben. Einen Ausblick liefert schließlich Abschnitt 2.7. Hier gehen wir auf die Möglichkeiten der Basiswahl im Matrixpotenzen-Kernel wie auch auf entsprechende Konsequenzen ein und setzen weitere Verweise.

Kapitel 3 bildet den zweiten Teil dieser Arbeit: Es beschreibt und untersucht die Möglichkeiten Synchronisationspunkte in Verfahren zur Lösung linearer Gleichungssysteme gänzlich zu unterlassen. In Abschnitt 3.1 betrachten wir dabei

zunächst eine Verfahrensklasse, die in der Praxis aufgrund mangelnder Konvergenzeigenschaften nicht von Relevanz ist, jedoch eine hervorragende Einstiegsmöglichkeit in die Thematik bietet – die Klasse der Fixpunktiterationsverfahren. Wir werden die Möglichkeiten, diese Verfahren asynchron zu betreiben, beschreiben und eine allgemeine Konvergenzaussage beweisen. Des Weiteren führen wir vorbereitend die potentiell mächtige Block-Jacobi-Vorkonditionierungsmethode (Abschnitt 3.2) sowie eine Variante des GMRES(m)-Verfahrens ein, welche in der Lage ist flexible, sich potentiell in jedem Schritt ändernde Vorkonditionierer zu verwenden (Abschnitt 3.3). Schließlich werden wir in Abschnitt 3.4 ein asynchrones Verfahren vorstellen, dessen wesentliche Komponenten die in den beiden vorangegangenen Abschnitten vorgestellten sein werden, und dieses anhand von verschiedenen Tests evaluieren.

Die Kapitel A und B stellen schließlich den Anhang dieser Arbeit dar. Hier geben wir für die Arbeit relevante Definitionen und Sätze an, die jedoch nicht im Mittelpunkt dieser stehen. Zur weiteren Erleichterung der Lektüre steht auf Seite 125 ein Index bereit.

1.3 Hardware-orientierte Numerik

Jeder moderne Computer basiert auf einer zentralen Recheneinheit, der *CPU* (engl. *central processing unit*). Sie ist für die meisten Berechnungen der Maschine verantwortlich und befindet sich seit Anfang der 1970er Jahre in der Regel auf einem einzigen Mikrochip, was den Begriff des „Mikroprozessors“ prägte. Ein erster Prozessor dieser Art, der *Intel 4004*, erreichte Taktfrequenzen zwischen 500 und etwa 750 KHz. 1993 brachte die Firma Intel den ersten Prozessor der *Pentium-Reihe* auf den Markt. Prozessoren dieser Baureihe erreichten bereits Taktfrequenzen zwischen 60 und 300 MHz. Dieser rasante Trend setzte sich unvermindert fort, bis ab dem Jahr 2000 mit Prozessoren der *Pentium-4-Reihe* bereits Mikroprozessoren zur Verfügung standen, deren Taktfrequenz zwischen 1300 und 3800 MHz lag. Die sukzessive Erhöhung der erreichbaren Rechengeschwindigkeit hatte somit zur Folge, dass auch numerische Verfahren, die man auf diesen Rechenmaschinen implementierte, schneller wurden, ohne dass konzeptionelle Änderungen an den mathematischen Hintergründen notwendig wurden.

Etwa ab dem Jahr 2005 setzte ein Umdenken im Bereich der Hardware-Industrie

ein: Der zunehmende Energiebedarf sowie eine nicht mehr wirtschaftlich handhabbare Wärmestrahlung immer höher getakteter Prozessoren begründete die Zeit der *Mehrkern-CPU*. Man begann mehrere voneinander autarke Recheneinheiten, die jedoch gleichsam einen gemeinsam geteilten Speicher – sogenannte geteilte *Caches* – besaßen, auf einem Mikrochip zu platzieren. Dies brachte den Vorteil mit sich, dass die gesamte potentiell verfügbare Rechenkraft einer derartigen Mehrkern-CPU gesteigert werden konnte, indem die Anzahl der einzelnen Rechenkerne, nicht aber die jeweilige Taktfrequenz erhöht wurde. Mehrkern-CPU sind dabei in der Regel so konstruiert, dass alle Rechenkerne mit etwa derselben Latenz auf den gemeinsam genutzten Speicher zugreifen können. Man spricht in diesem Zusammenhang vom sogenannten *UMA* (engl. *uniform memory access*). In großen industriellen wie wissenschaftlichen Rechenanlagen, die auf vielen miteinander vernetzten Computern bestehen, bewegt man sich zudem heute noch einen Schritt weiter: Hier ist es gängiger Standard eine einzige Hauptplatine eines Computers, einem sogenannten *Clusterknoten*, mit mehreren Mehrkern-CPU zu bestücken, dabei aber jeder CPU den Zugriff auf den Speicherbereich einer jeden anderen CPU innerhalb dieses Knotens zu ermöglichen. Dies hat zur Folge, dass nicht mehr jeder Kern gleich schnell auf jeden verfügbaren Speicherbereich zugreifen kann. Die Zugriffszeit hängt dabei entscheidend davon ab, ob die Speicheranfrage in den Bereich der eigenen oder einer fremden Mehrkern-CPU zielt. Im Falle einer solchen Speicherarchitektur spricht man schließlich vom *NUMA* (engl. *non-uniform memory access*).

Diese Entwicklung innerhalb der Hardware-Industrie machte nun in ihrer Konsequenz erstmals ein Umdenken im Bereich numerischer Verfahren notwendig. Um die maximale Rechenkraft eines solchen Computers zu nutzen, müssen die einzelnen von einem Algorithmus verlangten Rechenoperationen geschickt auf die Prozessorkerne verteilt werden, so dass im Idealfall kein Leerlauf entsteht. Leider wird dieser Zusammenhang von vielen klassischen numerischen Verfahren nicht inhärent berücksichtigt. Als elementares Beispiel stellen wir uns hier etwa ein lineares Gleichungssystem $Rx = b$ mit einer oberen Dreiecksmatrix R, einem Vektor von Unbekannten x und einer rechten Seite b vor. Der klassische Lösungsvorgang sieht es hier vor, zunächst die letzte Unbekannte, dann mit dieser die vorletzte, dann die vorvorletzte Unbekannte freizustellen und so fortzufahren, bis auch die erste Unbekannte bestimmt wurde. Diesen Vorgang bezeichnen wir als *Rückwärtseinsetzen*. Möchte man diesen Prozess nun auf einer Mehrkern-CPU umsetzen, ist

dies nicht ohne komplexe Überlegungen möglich: Was soll etwa der zweite Kern einer Mehrkern-CPU berechnen, während der erste Kern zur ersten Unbekannten auflöst? Ein Berechnen der zweiten Unbekannten ist schließlich noch nicht möglich, da dazu in der Regel der Wert der ersten bekannt sein muss. In seiner Konsequenz fordert dieser Zusammenhang eine geschickte Aufteilung der notwendigen Rechenoperationen zwischen den einzelnen Rechenkernen. Wir sprechen hier von der *Parallelisierung eines Verfahrens*.

Wie eingangs erwähnt, zeichnet sich jedoch noch ein anderer Trend innerhalb der Entwicklung von Rechenhardware ab: Während die verfügbare Rechenkraft einzelner Prozessoren – unabhängig, ob durch Steigerung der Taktfrequenz oder der Anzahl von autonomen Recheneinheiten erzielt – einem ständigen Wachstum unterliegt, ist die Kommunikationsgeschwindigkeit durch physische Schranken sowie das Energiebudget beschränkt. Bei dieser Geschwindigkeit handelt es sich im Kern um einen Begriff, der durch zwei Attribute bestimmt ist: *Latenz* und *Bandbreite*. Während erstere die Verzögerungszeit betitelt, die zwischen dem Zeitpunkt einer Datenanforderung etwa aus dem Speicher und dem Eintreffen der Informationen beim Prozessor liegt, bezeichnet letzterer Begriff die Menge an Daten, die in einer bestimmten Zeit überhaupt transferiert werden kann. Diese Begriffe beziehen sich dabei nicht nur auf eine Kommunikation zwischen dem Prozessor und Speicher eines Computers, sie sind vielmehr bei jedem auftretenden Informationstransfer innerhalb einer Rechenmaschine relevant. Dies kann das Versenden von Nachrichten zwischen verschiedenen Prozessoren oder sogar Informationsaustausch zwischen vielen einzelnen Computern innerhalb eines Netzwerks bedeuten, unabhängig davon, ob eine UMA- oder NUMA-Architektur vorliegt.

Die Latenz beziehungsweise Bandbreite, die in aktueller Hardware auftritt, bleibt tatsächlich jedoch nicht konstant. Die durchschnittliche Latenz verbessert sich laut Demmel [9] jedes Jahr abhängig von der betrachteten Computerkomponente zwischen 5 und 15 Prozent. Dies steht jedoch in keinem Verhältnis zu den stetig auftretenden Verbesserungen bezüglich der Rechengeschwindigkeit aktueller Computer. Diese nimmt jedes Jahr um etwa 59 Prozent zu.

Die Laufzeit eines Programms ergibt sich nunmehr aus Hardware-Sicht im Wesentlichen aus zwei Eigenschaften der Computerarchitektur, auf welcher es ausgeführt wird: Die verfügbare Rechengeschwindigkeit sowie die Dauer der innerhalb der Hardware stattfindenden Kommunikation. Da die Divergenz zwischen diesen

beiden Merkmalen nun einem steten Wachstum unterzogen ist, werden Kommunikationszeiten, die sich aus bestehender Latenz und verfügbarer Bandbreite ergeben, zu einem zunehmendem Nadelöhr in aktueller Algorithmik. Die damit einhergehende Problematik ist gerade die bereits in Abschnitt 1.1 skizzierte Memory-Wall-Problematik.

Um aktuelle Hardware möglichst optimal zu nutzen und somit weiterhin wissenschaftlichen Fortschritt zu ermöglichen, ist es zwingend notwendig, numerische Mathematik reflektierend in Bezug auf ihre Anwendbarkeit in modernen Computersystemen zu konzipieren. Um den obigen Sachverhalten Rechnung zu tragen, dürfen mathematische Algorithmen nicht mehr nur zielgerichtet auf ihre numerische Effizienz, sondern müssen zeitgleich auch hinsichtlich ihrer Ansprüche bezüglich der Realisierbarkeit (vgl. obiges Beispiel des Rückwärtseinsetzens), des Kommunikationsbedarfs sowie ihrer effizienten Implementierbarkeit auf aktueller paralleler Hardware optimiert werden. Leider bedeutet dies oftmals ein Balancieren sich widersprechender Anforderungen: Eine optimal parallelisierte Methode etwa wird sich in der Regel nicht mehr numerisch optimal verhalten, beispielsweise hinsichtlich ihrer Konvergenzgeschwindigkeit oder Stabilität. Das Bestreben klassische Lehrbuchmethoden der Numerik den heutigen Anforderungen moderner Höchstleistungsrechner anzupassen sowie neue Verfahren, die diesen gerecht werden, zu entwickeln, dabei jedoch einen sinnvollen Kompromiss der genannten Effizienzmerkmale zu finden, begründet das Forschungsgebiet der *Hardware-orientierten Numerik*.

Kapitel 2

Kommunikationsvermeidung in Krylow-Unterraum-Methoden

2.1 Krylow-Unterraum-Methoden

Krylow-Unterraum-Verfahren sind eine populäre Verfahrensklasse zum schnellen iterativen Lösen dünnbesetzter linearer Gleichungssysteme

$$Ax = b,$$

wobei A die Systemmatrix, x den Vektor der Unbekannten und b die rechte Seite darstellt. Sie gehen auf den russischen Schiffbauingenieur und Mathematiker Alexei Nikolajewitsch Krylow (1863 – 1945) zurück, welcher die für diese Methoden essentiellen Krylow-Unterräume 1931 zur Lösung von Eigenwertproblemen definierte:

Definition 2.1 (Krylow-Unterraum)**:** Der *Krylow-Unterraum zum Iterationsindex* $k > 0$ ist definiert als

$$\mathcal{K}_k := \mathcal{K}_k(A, r^{(0)}) := \text{span} \bigcup_{i=1}^{k} \{A^{i-1} r^{(0)}\},$$

wobei $r^{(0)} = b - Ax^{(0)}$ das Ausgangsresiduum sowie $x^{(0)}$ die Ausgangsnäherung der Lösung darstellt.

Definition 2.2 (Krylow-Unterraum-Methode)**:** Eine *Krylow-Unterraum-Methode* ist ein Verfahren zur Berechnung von Näherungslösungen $x^{(k)} \in x^{(0)} + \mathcal{K}_k := \{x^{(0)} + v \mid v \in \mathcal{K}_k\}$ des linearen Gleichungssystems $Ax = b$, wobei

$$r^{(k)} := b - Ax^{(k)} \perp \mathcal{K}_k$$

für jede Iterierte $r^{(k)}$ mit $k > 0$ gelten soll. Die Orthogonalität bezieht sich hier gerade auf das euklidische Skalarprodukt.

Bekannte Vertreter dieser Verfahrensklasse sind etwa das *CG-Verfahren* (engl. *conjugate gradients method*), das *BiCGStab-Verfahren* (engl. *biconjugate gradient stabilized method*) sowie das *GMRES-Verfahren* (engl. *generalized minimal residual method*). Ersteres wird in der deutschsprachigen Literatur auch als das *Verfahren der konjugierten Gradienten* bezeichnet. Erwähnt sei, dass dieses Verfahren in seiner Konvergenztheorie explizit die Vorgabe einer symmetrischen, positiv definiten Systemmatrix A fordert. Wir werden im Folgenden von *SPD-Matrizen* sprechen.

Zur weiteren Einführung in die Theorie der Krylow-Unterraum-Verfahren verweisen wir auf das Vorlesungsskriptum von Göddeke [17, Kapitel 3.4] respektive auf das dieser Quelle zu Grunde liegende Lehrbuch von Saad [24, Kapitel 6] und geben noch exemplarisch eine gängige unvorkonditionierte (vgl. Unterkapitel 2.5) Formulierung des CG-Verfahrens in Pseudocode an, um insbesondere auch unsere Notation an einem bekannten Beispiel einzuführen.

Algorithmus 2.3 (CG-Verfahren)**:**

Eingabe: SPD-Systemmatrix A, Anfangsnäherung $x^{(1)}$, rechte Seite b

1: Setze $r^{(1)} := p^{(1)} := b - Ax^{(1)}$
2: Setze $\alpha^{(1)} := (r^{(1)}, r^{(1)})_2 = ||r^{(1)}||_2^2$
3: Schleife über $k = 1, 2, \ldots$ bis zur Konvergenz
4: $\quad v^{(k)} := Ap^{(k)}$
5: $\quad \lambda^{(k)} := \frac{\alpha^{(k)}}{(v^{(k)}, p^{(k)})_2}$
6: $\quad x^{(k+1)} := x^{(k)} + \lambda^{(k)} p^{(k)}$
7: $\quad r^{(k+1)} := r^{(k)} - \lambda^{(k)} v^{(k)}$
8: $\quad \alpha^{(k+1)} := (r^{(k+1)}, r^{(k+1)})_2 = ||r^{(k+1)}||_2^2$ und Konvergenzkontrolle
9: $\quad p^{(k+1)} := r^{(k+1)} + \frac{\alpha^{(k+1)}}{\alpha^{(k)}} p^{(k)}$
10: Ende Schleife

2.2 Pipelined CG-Verfahren

Zwecks einer ersten exemplarischen Verbesserung des obigen Algorithmus 2.3 bezüglich seiner Kommunikationseigenschaften, betrachten wir dessen Hauptschleife erneut:

$$\begin{aligned} v^{(k)} &:= Ap^{(k)} \\ \lambda^{(k)} &:= \frac{\alpha^{(k)}}{(v^{(k)}, p^{(k)})_2} \\ x^{(k+1)} &:= x^{(k)} + \lambda^{(k)} p^{(k)} \\ r^{(k+1)} &:= r^{(k)} - \lambda^{(k)} v^{(k)} \\ \alpha^{(k+1)} &:= (r^{(k+1)}, r^{(k+1)})_2 \\ \beta^{(k)} &:= \frac{\alpha^{(k+1)}}{\alpha^{(k)}} \\ p^{(k+1)} &:= r^{(k+1)} + \beta^{(k)} p^{(k)}. \end{aligned}$$

Wir haben hier im Unterschied zu Algorithmus 2.3 lediglich einen weiteren Hilfsskalar $\beta^{(k)}$ eingeführt. Hinsichtlich einer parallelen Implementierung des CG-Verfahrens ist die Anordnung der Skalarprodukte wie etwa $(r^{(k+1)}, r^{(k+1)})_2$ störend, da diese globale Kommunikation zwischen allen Prozessoren erfordern. In einem Konferenzbeitrag von Strzodka und Göddeke [27] schlagen die Autoren eine geschickte Umordnung der einzelnen Algorithmusoperationen vor, die dazu führt, dass globale Kommunikationspunkte „gebündelt“ auftreten. Konkret handelt es sich dabei um die Umordnung

$$\begin{aligned} x^{(k+1)} &:= x^{(k)} + \lambda^{(k)} p^{(k)} \\ r^{(k+1)} &:= r^{(k)} - \lambda^{(k)} v^{(k)} \\ p^{(k+1)} &:= r^{(k+1)} + \beta^{(k)} p^{(k)} \\ v^{(k)} &:= Ap^{(k)}. \end{aligned}$$

Da jedoch

$$\begin{aligned} \alpha^{(k+1)} &:= (r^{(k+1)}, r^{(k+1)})_2 \\ \beta^{(k)} &:= \frac{\alpha^{(k+1)}}{\alpha^{(k)}} \end{aligned}$$

gilt und wir daher $r^{(k+1)}$ berechnet haben müssen, um $p^{(k+1)}$ berechnen zu können, scheint diese Umformung zunächst nicht rechtfertigbar. Daher führen Strzodka und

Göddeke [27] einen weiteren Skalar

$$\sigma^{(k)} := \lambda^{(k)}(\lambda^{(k)}(v^{(k)}, v^{(k)})_2 - (p^{(k)}, v^{(k)})_2)$$

ein. Mit Hilfe der Orthogonalitätseigenschaften bezüglich der Residuen $r^{(k)}$ sowie der Suchrichtungen $p^{(k)}$ lässt sich nun zeigen, dass

$$\sigma^{(k)} = \alpha^{(k+1)} \tag{2.1}$$

gilt. Nach der Berechnung der obigen Vektoren folgt schließlich jene aller Skalarprodukte:

$$\begin{aligned}
\alpha^{(k+1)} &:= (r^{(k+1)}, r^{(k+1)})_2 \\
\lambda^{(k+1)} &:= \frac{\alpha^{(k+1)}}{(v^{(k+1)}, p^{(k+1)})_2} \\
\sigma^{(k+1)} &:= \lambda^{(k+1)}(\lambda^{(k+1)}(v^{(k+1)}, v^{(k+1)})_2 - (p^{(k+1)}, v^{(k+1)})_2) \\
\beta^{(k+1)} &:= \frac{\sigma^{(k+1)}}{\alpha^{(k+1)}}.
\end{aligned}$$

Im Vergleich zur Ausgangsversion des Verfahrens befinden sich im ersten Teil der Iteration nun keine globalen Kommunikationspunkte mehr unter der Bedingung, dass die Systemmatrix A dünnbesetzt ist, was wir aber im Rahmen dieser Arbeit vorausgesetzt haben. Des Weiteren muss nun zwar ein weiteres Skalarprodukt $(v^{(k+1)}, v^{(k+1)})_2$ berechnet werden, jedoch erringen wir den Vorteil, dass alle benötigten Skalarprodukte innerhalb einer Iteration gleichzeitig verlangt werden. Dadurch steigt das Volumen der zu kommunizierenden Daten, jedoch müssen Prozessoren insgesamt weniger auf das Eintreffen dieser warten.

Strzodka und Göddeke [27] haben in Kapitel 5 ihrer Veröffentlichung zudem dieser sogenannten *Pipelined-Variante des CG-Verfahrens* durch umfangreiche numerische Tests ähnliche Eigenschaften bezüglich Konvergenzverhalten und Stabilität wie jenen der Ausgangsformulierung attestiert.

Obiges Vorgehen führt zu einer Laufzeitoptimierung des CG-Verfahrens auf parallelen Computern, allerdings ist der Begriff „Kommunikationsvermeidender Algorithmus“ nicht exakt zutreffend – schließlich wird das eigentlich kommunizierte Datenvolumen nicht verringert, vielmehr noch wird es sogar erhöht. So sinkt die Anzahl globaler Synchronisationspunkte sowie die gemessenen Latenz überschlagsweise um 80 Prozent zu Lasten einer Steigerung der Bandbreitenanforderung von 20 Prozent. Einen tatsächlich kommunikationsvermeidenden Ansatz beschreiben wir hingegen im restlichen Teil dieses Kapitels.

2.3 Matrixpotenzen-Kernel

Innerhalb einer Iteration eines Verfahrens der Krylow-Unterraum-Klasse wird typischerweise wenigstens eine Matrix-Vektor-Multiplikation ausgeführt: Beim CG-Verfahren etwa wird für jeden Iterationsindex k das Produkt $Ap^{(k)}$ der Systemmatrix A mit einer Suchrichtung $p^{(k)}$ benötigt (vgl. Zeile 4 in Algorithmus 2.3). Dieses Produkt beziehungsweise der damit einhergehende serielle Synchronisationspunkt zwischen einzelnen Prozessoren innerhalb einer parallelen Implementierung bildet einen „Flaschenhals“ im Informationsfluss, was gerade daher besondere Relevanz hat, da obige Multiplikation in der Regel einen signifikanten Teil der Löserlaufzeit in Anspruch nimmt.

Des Weiteren sorgt auch die mit einer parallelen Berechnung von Skalarprodukten einhergehende globale Kommunikation – trete sie nun in Form von Datenaustausch über ein Netzwerk oder Speicherzugriffen auf – für suboptimale Laufzeiten (vgl. etwa Zeile 5 in Algorithmus 2.3).

Wir werden diese Problematik im Folgenden genauer erläutern, zwecks einer präzisen Nomenklatur treffen wir jedoch zunächst eine Definition:

Definition 2.4 (Kernel)**:** Als *Kernel* bezeichnen wir einen isolierten, sich wiederholenden Bereich eines numerischen Algorithmus, der einen signifikanten Teil der Laufzeit in Anspruch nimmt.

Diese Definition geht zurück auf die Dissertation von Hoemmen [18], findet sich aber ebenso in diversen Veröffentlichungen aus dem Bereich High Performance Computing. Insbesondere besteht kein Zusammenhang zu klassischen mathematischen Begrifflichkeiten, wie etwa dem Nullraum einer Abbildung.

Wir nehmen den eingangs geschilderten Sachverhalt als Beispiel. Das Produkt $Ap^{(k)}$ stellt innerhalb des CG-Algorithmus einen Kernel dar: Aufgrund der typischerweise dünnbesetzten Erscheinungsform der Systemmatrix A sprechen wir hier von einem *SpMV-Kernel* (engl. *sparse matrix-vector kernel*).

Im Prinzip basieren alle klassischen iterativen Verfahren auf der Zusammensetzung verschiedener Kernel. Als Beispiele sind hier für das CG-Verfahren neben dem SpMV-Kernel unter anderem Skalarprodukt- sowie AXPY-Kernel zu nennen. Letzterer bezeichnet dabei Operationen der Art $y := \alpha x + y$ mit Vektoren x und y sowie einem Skalar α.

Diese Kernel-Struktur bildet gerade das „Nadelöhr“ innerhalb einer parallelen Implementierung eines iterativen Verfahrens. Bei klassischen Umsetzungen der CG-Methode beschränken sich die Parallelisierungsbemühungen beispielsweise meist auf die Aufteilung der Arbeit unter den zur Verfügung stehenden Prozessoren ausschließlich innerhalb einzelner Kernel-Umgebungen. Hat ein Prozessor seine Arbeit abgeschlossen, muss er zunächst leerlaufen und warten, bis auch der letzte Prozessor seine Berechnungen abschließt, die der Kernel ihm zugetragen hat. Erst dann gilt der Kernel als abgearbeitet und die Algorithmik springt sequentiell zum nächsten Programmpunkt beziehungsweise Kernel.

Im Kontext der Memory-Wall-Problematik ergibt sich ein weiterer Missstand in Verbindung mit einer klassischen Kernel-Struktur. Da ein Kernel einen autarken Programmteil darstellt, müssen bei jedem Aufruf die notwendigen Informationen aus dem potentiell langsamen Speicher gelesen werden. Bei einem SpMV-Kernel innerhalb eines Krylow-Unterraum-Verfahrens etwa müssen die Einträge der Systemmatrix A in jeder Iteration erneut geladen und zudem die Einträge des zu multiplizierenden Vektors kommuniziert werden.

Die grundlegende Idee der Forschergruppe um Demmel et al. [10, 11] ist es nun, diese Kernel-Struktur aufzubrechen und so die mit ihr einhergehende Problematik des Flaschenhalses im Programmablauf aufzuheben sowie die potentiell langsamen Speicherzugriffe zu vermeiden, dabei jedoch nicht die fundamentalen Vorteile eines Kernels, also einer Programmsubroutine, in Bezug auf die ihm inhärenten effizienten Programmierungs- und gezielten Parallelisierungsmöglichkeiten aufzugeben. Speziell befassen sich die Forscher damit, den SpMV-Kernel innerhalb von Krylow-Unterraum-Verfahren so umzuorganisieren, dass eine gewisse Menge von benötigten Matrix-Vektor-Produkten zeitgleich berechnet werden kann. Diese Umorganisierung führt auf den Begriff des „Matrixpotenzen-Kernels“ (engl. *matrix powers kernel*). Dieser kann – wie die Autoren zeigen – so implementiert werden, dass sich die Kommunikationskosten statt wie $\mathcal{O}(s)$ in klassischen Algorithmen wie $1 + \mathcal{O}(1)$ für s Iterationen verhalten. Dazu stellen sie verschiedene parallele Algorithmen vor.

Definition 2.5 (Matrixpotenzen-Kernel)**:** Der *Matrixpotenzen-Kernel* (engl. *matrix powers kernel*) erwartet als Eingabe eine Matrix $A \in \mathbb{R}^{n \times n}$, einen Vektor

$v \in \mathbb{R}^n$ sowie eine Drei-Term-Rekursion der Form

$$p_{j+1}(Z) = a_j Z p_j(Z) - b_j p_j(Z) - c_j p_{j-1}(Z)$$

mit $a_j \neq 0$ sowie $j = 1, \ldots, s-1$. Dabei sind $p_0(Z), \ldots, p_s(Z)$ Polynome von Matrizen $Z \in \mathbb{R}^{n \times n}$. Der Index markiert den jeweiligen Grad. Der Kernel berechnet nun Vektoren $v_1, \ldots, v_{s+1}$, bestimmt durch die Beziehung $v_j = p_{j-1}(A)v$. Diese Vektoren

$$\underline{V} := [v_1, \ldots, v_{s+1}] = [p_0(A)v, \ldots, p_s(A)v]$$

bilden schließlich eine Basis des Krylow-Unterraums

$$\mathcal{K}_{s+1}(A, v) = \text{span}\{v, Av, \ldots, A^s v\}.$$

Üblicherweise wird man $p_0(Z) = 1$ setzen, so dass $v_1 = v$ folgt. Setzt man die Koeffizienten ferner auf $a_j = 1, b_j = 0, c_j = 0$ für alle $j = 1, \ldots, s$, ergeben sich für die Polynome gerade die bekannten Monome $p_j = Z^j$. Wir weisen überdies daraufhin, dass der Matrixpotenzen-Kernel die Vektoren $v_1, \ldots, v_{s+1}$ explizit als Spalten einer Matrix zurückgibt. Diese Schreibweise ist zwar für Vektorraumbasen mathematisch unüblich, bietet jedoch bei unserer späteren Abfassung der Algorithmik notationelle Vorteile.

Prozessor Bisher haben wir die Bezeichnung „Prozessor“ eher abstrakt für eine autarke Recheneinheit verwandt, die auf eine nicht näher bestimmte Weise mit ihresgleichen kommunizieren kann. Diese Nomenklatur werden wir auch im Folgenden beibehalten. Es sei jedoch darauf hingewiesen, dass Hardware-orientiert gedacht mit diesem Begriff verschiedene architektonische Modelle gemeint sein können. Grundlegend sind hier die Begriffe *Distributed Memory* und *Shared Memory*. Im ersten Fall ist mit „Prozessor“ typischerweise ein gesamter Rechenknoten mit jeweiligem privaten lokalen Speicher und mehreren *CPU* (engl. *central processing unit*), ihrerseits bestehend aus einzelnen Kernen, gemeint, während sich im Kontext des letzteren Begriffs „Prozessor“ eher auf einen einzelnen Kern einer CPU bezieht. Typischerweise nutzt man Distributed Memory eher im Bereich *grob-granularer Parallelität*, das heißt parallele Algorithmen mit eher langer Ausführungszeit und relativ seltenen Synchronisierungen. Im Gegensatz dazu

kommt Shared Memory oft in Bereichen *fein-granularer Parallelität* zum Einsatz. In solchen Bereichen muss meist häufiger synchronisiert werden, dafür jedoch mit einem geringeren Datenvolumen in jeder Kommunikation. Dieses Vorgehen findet seine Berechtigung in den Eigenschaften der einzelnen Hardwaremodelle bezüglich Latenz und Bandbreite. Diese sind bei Kommunikation innerhalb eines Computernetzwerks selbst bei Verwendung von Techniken wie Infiniband[1] um Größenordnungen größer als bei Synchronisationen innerhalb einzelner Mehrkern-CPU.

Distributed versus Shared Memory Im Rahmen dieser Arbeit werden wir uns in den theoretischen Bereichen, wie der Darstellung unserer Algorithmen, eher an einer Distributed-Memory-Architektur orientieren. Das heißt wir nutzen explizit die Begriffe „Senden“ wie „Empfangen“ als Operatoren, um Kommunikation zwischen Prozessoreinheiten zu markieren. Analog werden wir diese Bezeichnungen auch innerhalb des Fließtextes verwenden. Wir weisen an dieser Stelle jedoch daraufhin, dass sämtliche Überlegungen ohne Mühe auf die Shared-Memory-Situation übertragbar sind. Tut man dies beispielsweise im Rahmen der Algorithmik und strebt etwa eine Implementierung auf einem Multicore-Prozessor mit gemeinsam genutzten Speicher an, sind solche Kommunikationsoperatoren implizit im Abruf einzelner Variablenwerte, die von anderen Prozessen geschrieben wurden, aus diesem Speicher enthalten. In der Regel ist dieses Modell für den Programmierer sogar deutlich komfortabler.

2.3.1 Graphentheoretische Notationen

Zur präziseren Darstellung und Analyse der im Folgenden präsentierten Algorithmen zur Realisierung des Matrixpotenzen-Kernels benötigen wir einige weitere auf unsere Situation speziell zugeschnittene graphentheoretische Bezeichnungen. Eine in diesem Kontext üblicherweise sinnvolle Definition ist die des *Adjazenzgraphen*. Zu einer gegebenen Matrix $A = (a_{i,j}) \in \mathbb{R}^{n \times n}$ assoziieren wir mit jeder Spalte beziehungsweise Zeile einen Knoten des Graphen. In ihm existiert genau dann eine gerichtete Kante von Knoten i zu Knoten j, wenn $a_{i,j} \neq 0$ gilt. Berechnet man ein SpMV-Produkt der Form $y := Ax$, wird zur Berechnung von y_i unter anderem der Wert von x_j benötigt.

[1] Spezifikation zur Beschreibung einer seriellen Hochgeschwindigkeitsübertragungstechnologie

Im Kontext von Implementierungen des Matrixpotenzen-Kernels ist jedoch eine Notation vonnöten, die Abhängigkeiten innerhalb einzelner Multiplikationsebenen darstellen kann, etwa die Menge aller Indizes j, deren zugehörige Vektorkomponenten x_j zur Berechnung der Komponente z_i nach $z = A^k x$ zur Verfügung stehen müssen. Um diesem Zusammenhang Rechnung zu tragen, definieren wir einen Graph $G = G(A) = (V, E)$ nach folgender Regel: Wir bezeichnen mit $x_j^{(i)}$ die j-te Komponente des Vektors $x^{(i)} = A^i x^{(0)}$ und identifizieren ebendieses $x_j^{(i)}$ für $i = 0, \ldots, s$ und $j = 1, \ldots, n$ in eineindeutiger Weise mit einem Knoten des Graphen G. Eine gerichtete Kante in G von $x_j^{(i+1)}$ nach $x_m^{(i)}$ existiert nun genau dann, wenn $a_{j,m} \neq 0$ gilt. Die Knotenmenge von G hat also die Mächtigkeit $n(s+1)$. Da der obere Index i eines Knoten gerade die Multiplikationsebene bezeichnet, sprechen wir hier auch vom *Level des Knoten* $x_j^{(i)}$.

Unter der *Affinität* $q(x_j^{(i)})$ *eines Knotens* $x_j^{(i)}$ verstehen wir ferner die Prozessornummer $q \in \{1, \ldots, p\}$ jenes Prozessors, der zu $x_j^{(i)}$ eine gewisse Lokalität aufweist. Im Fall einer Distributed-Memory-Architektur bedeutet dies etwa, dass $x_j^{(i)}$ auf Prozessor q gespeichert ist, während in der Shared-Memory-Situation jener Prozessor q gemeint ist, welcher auf $x_j^{(i)}$ im Sinne einer geringen Latenz beziehungsweise hohen Bandbreite eine ausgezeichnete Zugriffsmöglichkeit hat. Wir setzen voraus, dass $x_j^{(0)}, \ldots, x_j^{(s)}$ die gleiche Affinität aufweisen, so dass diese lediglich von der gewählten Komponente j abhängt.

Des Weiteren bezeichne $V_q \subset V$ die Teilmenge von Knoten aus G mit Affinität q sowie $V^{(i)} \subset V$ die Teilmenge solcher Knoten aus G mit Level i. Entsprechend verstehen wir unter $V_q^{(i)} \subset V$ die Schnittmenge dieser Teilmengen.

Für eine beliebige Teilmenge $S \subset V$ stelle $R(S) \subset V$ die Menge von Knoten aus G dar, die durch einen gerichteten Pfad innerhalb von G erreicht werden können, beginnend bei einem Element aus S. Wir erlauben an dieser Stelle auch den trivialen Pfad, der über keine Kante unmittelbar an seinem Anfangsknoten endet. Daher gilt also insbesondere $S \subset R(S)$. $R(S, m) \subset R(S)$ bezeichne darüber hinaus jene gerichteten Pfade von einer Länge von höchstens m Kanten, während wir unter $R_q(S)$, $R^{(i)}(S)$ beziehungsweise $R_q^{(i)}(S)$ die entsprechenden Einschränkungen auf Affinität q, Level i beziehungsweise Affinität q sowie Level i verstehen.

Die Menge der von einem Prozessor q lokal – also ohne Kommunikation nur mit Hilfe der zu $V_q^{(0)}$ gehörenden Daten – berechenbaren Komponenten bezeichnen wir

mit $L_q \subset V$. Mit den vorangegangenen Definitionen gilt gerade der Zusammenhang

$$L_q = \{x \in V_q \mid R(x) \subset V_q\}.$$

Analog zu Obigem bezeichnen wir mit $L_q^{(i)}$ wieder die Knoten aus L_q mit Level i.

Des Weiteren bezeichnen wir mit $B_{q,r} \subset V$ eine minimale Menge von Knoten aus G, die Prozessor r an Prozessor q senden muss, damit Prozessor q die Knoten, für die er zuständig ist, berechnen kann, also gerade jene aus V_q. Für einen Knoten x gilt schließlich genau dann $x \in B_{q,r}$, wenn $x \in L_r$ gilt und ein Pfad von einem $y \in V_q$ zu x existiert, so dass x der erste Knoten dieses Pfads in L_r ist.

Beispiel Wir wollen die obigen Notationen abschließend an einem Beispiel veranschaulichen. Dazu ziehen wir exemplarisch die Matrix

$$A = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ c & 0 & d \end{bmatrix}$$

mit Nichtnull-Einträgen a, b, c, d sowie die Matrix-Vektor-Produkte $x^{(i)} = A^i x^{(0)}$ für $i = 1, 2$ heran. Dabei sei $x^{(0)} \neq 0$ ein beliebiger Ausgangsvektor.

In unserem Beispiel gilt also $s = 2$ und die Knotenmenge V des nach obigen Regeln assemblierten Graphen $G = (V, E)$ ergibt sich in dieser Situation nun als

$$V = \{x_1^{(0)}, x_2^{(0)}, x_3^{(0)}, x_1^{(2)}, x_2^{(2)}, x_3^{(2)}, x_1^{(3)}, x_2^{(3)}, x_3^{(3)}\}.$$

Eine gerichtete Kante von Knoten $x_j^{(i+1)}$ nach $x_m^{(i)}$ besteht nun, wenn A an Position j, m einen Nichtnull-Eintrag besitzt. Demnach ergibt sich die Kantenmenge

$$E = \left\{ \begin{array}{l} (x_1^{(2)}, x_1^{(1)}), (x_2^{(2)}, x_2^{(1)}), (x_3^{(2)}, x_1^{(1)}), (x_3^{(2)}, x_3^{(1)}), \\ (x_1^{(1)}, x_1^{(0)}), (x_2^{(1)}, x_2^{(0)}), (x_3^{(1)}, x_1^{(0)}), (x_3^{(1)}, x_3^{(0)}) \end{array} \right\},$$

wobei das Tupel (x, y) eine Kante von Knoten x nach Knoten y darstellt. Abbildung 2.1 zeigt den entstandenen Graphen.

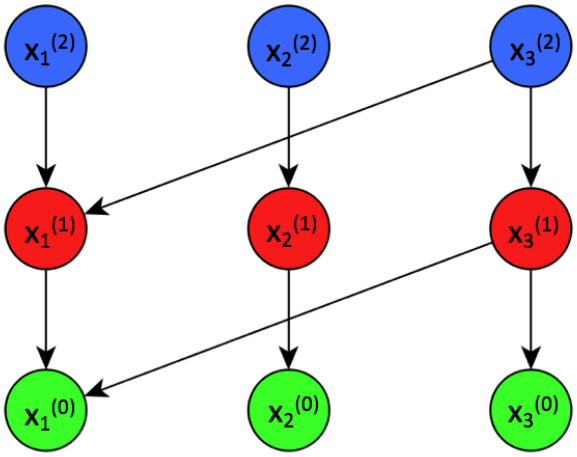

Abbildung 2.1: Graph $G = (V, E)$ zur Matrix A

Die Arbeit zur Berechnung der Matrix-Vektor-Produkte soll zur weiteren Veranschaulichung auf zwei Prozessoren aufgeteilt werden – Prozessor 1 und Prozessor 2. Für die Affinität q gilt also $q \in \{1, 2\}$. Speziell sollen etwa die ersten beiden Zeilen von $x^{(0)}$ Lokalität zu Prozessor 1 und die letzte Zeile von $x^{(0)}$ Lokalität zu Prozessor 2 aufweisen. Im Distributed-Memory-Fall würde dies etwa bedeuten, dass die ersten beiden Einträge von $x^{(0)}$ im Speicher von Prozessor 1 liegen, der letzte Eintrag aber hingegen im Speicher von Prozessor 2. Da wir eine levelunabhängige Lokalität gefordert haben, gilt nun etwa für die Menge aller Knoten, die Lokalität zu Prozessor 2 aufweisen,

$$V_2 = \{x_3^{(0)}, x_3^{(1)}, x_3^{(2)}\}.$$

Die Menge aller Knoten mit Level 1 etwa ergibt sich hingegen als

$$V^{(1)} = \{x_1^{(1)}, x_2^{(1)}, x_3^{(1)}\}.$$

Die Menge, die beide Eigenschaften kombiniert, ist ferner

$$V_2^{(1)} = V_2 \cap V^{(1)} = \{x_3^{(1)}\}.$$

Setzen wir die Menge $S := \{x_3^{(2)}\}$, ergibt sich für die Menge der durch einen gerichteten Pfad ausgehend von Knoten $x_3^{(2)}$ erreichbaren Knoten die Menge

$$R(S) = \{x_3^{(2)}, x_3^{(1)}, x_1^{(0)}, x_1^{(1)}\}.$$

Für die von Prozessor 2 lokal berechenbaren Komponenten der Matrix-Vektor-Produkte ergibt sich

$$L_2 = \{x \in V_2 \mid R(x) \subset V_q\} = \{x_3^{(0)}\}.$$

Das bedeutet also, dass in unserem Beispiel Prozessor 2 nicht in der Lage ist überhaupt kommunikationsfrei Berechnungen anzustellen. Das einzige lokal berechenbare Element ist trivialerweise $x_3^{(0)}$, welches vor Beginn der Berechnungen bereits im Speicher von Prozessor 2 vorliegt. Anschaulich leuchtet dieses Resultat sofort ein: Der Nichtnull-Eintrag c der Matrix A sorgt dafür, dass zur Berechnung jedes weiteren Knotens mit Lokalität zu Prozessor 2 der Ausgangswert $x_1^{(0)}$ zur Verfügung stehen muss, welcher aber ohne eine Synchronisierung unter den Prozessoren nicht bereitsteht.

Schließlich wollen wir noch die minimale Menge $B_{1,2}$ von Knoten aus V bestimmen, die Prozessor 2 an Prozessor 1 senden muss, damit Prozessor 1 die Knoten, für die er zuständig ist, berechnen kann. Ein Knoten $x \in V$, der sich in dieser Menge befindet, muss nun nach Konstruktion auch $x \in L_2$ erfüllen. Die Menge $B_{1,2}$ besteht also entweder einzig aus dem Element $x_3^{(0)}$ oder ist leer. Da aber kein Knoten $y \in V_1$ existiert, von welchem ein gerichteter Pfad zu $x_3^{(0)}$ ausgeht, kann die zweite Bedingung, die oben gefordert wurde, für keinen Knoten x erfüllt sein und es ergibt sich

$$B_{1,2} = \emptyset.$$

Auch dies ist anschaulich aber wieder sofort klar: Da die ersten beiden Einträge der dritten Spalte von A jeweils gleich null sind, benötigt Prozessor 1 den einzigen Ausgangswert $x_3^{(0)}$, der ihm von Prozessor 2 übermittelt werden könnte, nicht und ist somit in seinen Berechnungen ebenso wenig auf Prozessor 2 angewiesen.

2.3.2 Exemplarische Realisierung

Hoemmen [18] stellt in seiner Dissertation drei parallele Algorithmen zur Realisierung des Matrixpotenzen-Kernels vor: *PA0*, *PA1* und *PA2*. Wir möchten diese Algorithmen kurz beschreiben, Unterschiede hervorheben und im letzten Teil dieses Abschnitts das jeweilige Vorgehen anhand des Beispiels einer Systemmatrix A in Tridiagonalgestalt unter Zuhilfenahme dreier Grafiken erläutern.

PA0 bezeichnet eine naive Variante, in der s SpMV-Kernel aufgerufen werden. In jedem Schritt j wird unter Ausnutzung der Drei-Term-Rekursion $v_j = p_{j-1}(A)v$ berechnet. Dazu benötigen wir eine SpMV-Multiplikation und zwischen null und zwei AXPY-Operationen (abhängig davon, ob die Koeffizienten b_{j+1} oder c_{j+1} verschwinden). Dabei ist jeder Prozessor für die Berechnung eines Blocks an Komponenten von v_j verantwortlich. Jeder Prozessor erhält dann die zur Berechnung seiner Einträge notwendigen Komponenten von v_{j-1} und möglicherweise auch jene von v_{j-2}. Dieser Algorithmus vermeidet keinerlei Kommunikation und dient lediglich zu Vergleichszwecken.

Im Algorithmus PA1 hingegen berechnet jeder Prozessor zunächst solche Einträge der Vektoren aus $\{v_2, \ldots, v_{s+1}\}$, die keine Kommunikation mit anderen Prozessoren erfordern. Zeitgleich sendet er alle Komponenten von v_1, welche mit ihm assoziiert sind und von benachbarten Prozessoren benötigt werden. Sind alle lokal möglichen Berechnungen eines Prozessors abgeschlossen, wartet dieser, bis er die benötigten Einträge von v_1 erhält, um seinen Teil der Berechnung von $\{v_2, \ldots, v_{s+1}\}$ fertigzustellen. In diesem Algorithmus werden Daten potentiell redundant berechnet: In der Nähe von Prozessorgrenzen kann es vorkommen, dass Werte von beiden Nachbarn berechnet werden.

PA2 berechnet zunächst die kleinstredundante Menge von lokalen Werten, die von den Nachbarn benötigt werden. Dadurch sparen Nachbarprozessoren einige redundante Berechnungen potentiell ein.

Wir wollen die Algorithmen und die in ihnen bestehenden Kommunikationsabhängigkeiten zwischen einzelnen Prozessoren zur Berechnung des Matrixpotenzen-Kernels anhand der folgenden Abbildungen illustrieren (Quelle: Hoemmen [18]). Diesem Beispiel liegt dabei, wie eingangs erwähnt, eine tridiagonale Matrix A zugrunde. Für die Höhe der im Matrixpotenzen-Kernel zu berechnenden Potenzen setzen wir $s = 8$. Jede Zeile stellt dabei Einträge der Vektoren $A^j v$ für $j = 0, \ldots 8$ dar, wobei der Einfachheit halber im Matrixpotenzen-Kernel die kanonische Monombasis gesetzt wurde. Abgebildet wird zudem lediglich ein Teilausschnitt von 30 Komponenten jedes Vektors, die in zwei Partitionen – für jeweils einen Prozessor – aufgeteilt wurden. Getrennt werden diese beiden Partitionen durch eine vertikale grüne Linie. Datenabhängigkeiten zwischen einer Komponente i von $A^j v$ bestehen jeweils zu den Komponenten $i-1, i$ und $i+1$ von $A^{j-1}v$. Dies ergibt sich unmit-

telbar aus der tridiagonalen Besetzungsstruktur von A und wurde mit vertikalen beziehungsweise diagonalen Verbindungen zwischen den Kreisen markiert.

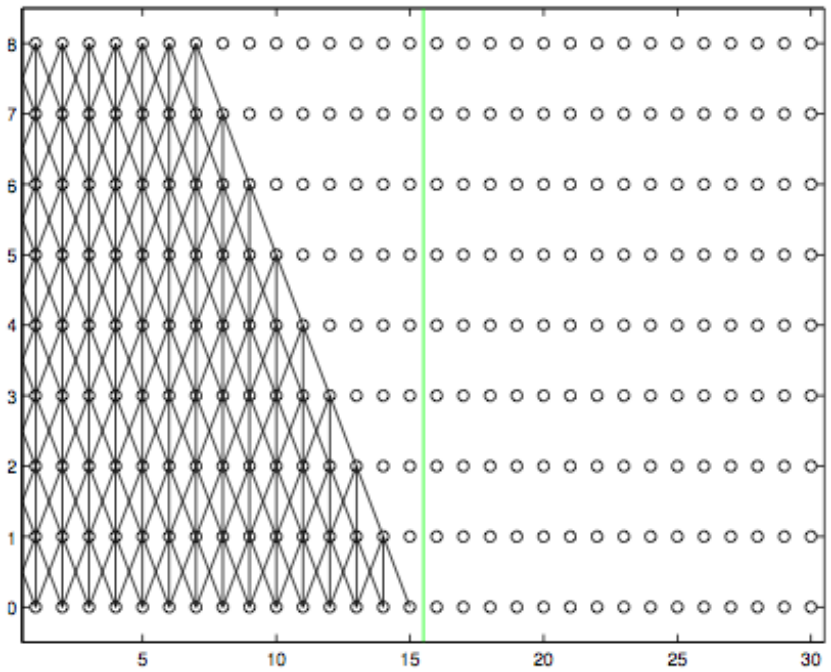

Abbildung 2.2: Lokal berechenbare Komponenten [18]

Abbildung 2.2 zeigt alle Komponenten des Ausschnitts, welche unabhängig lokal, also ohne Kommunikation mit dem benachbarten rechten Prozessor, berechnet werden können. Die restlichen Kreise ohne Verbindungen links jenseits der grünen Linie stellen jene Komponenten der Vektoren $A^j v$ dar, die explizit Informationen des rechten Nachbarprozessors benötigen, um bearbeitet werden zu können.

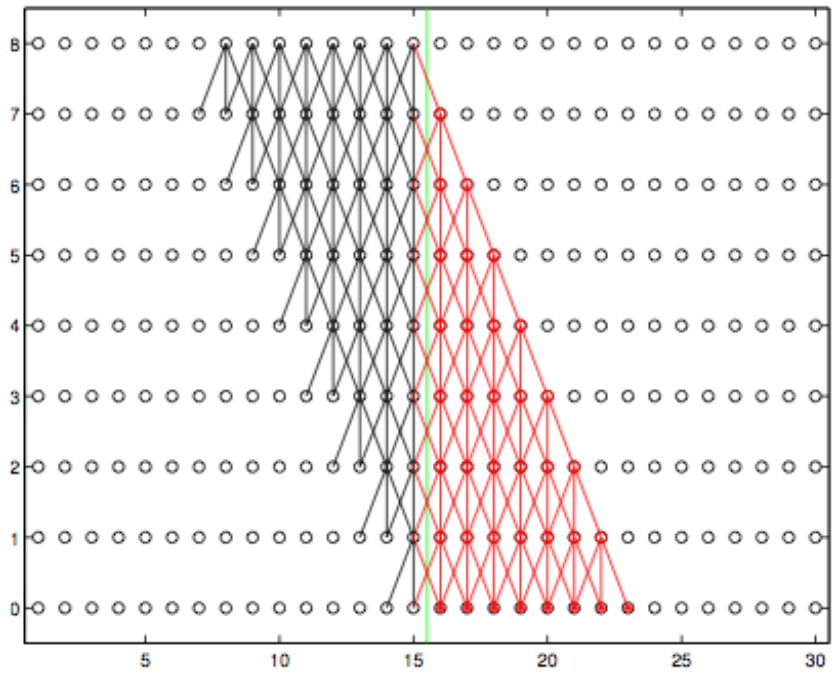

Abbildung 2.3: Abhängigkeiten zwischen benachbarten Prozessoren in PA1 [18]

Abbildung 2.3 zeigt, wie innerhalb des Algorithmus PA1 diese verbleibenden Komponenten, für welche Kommunikation mit dem rechten Nachbarprozessor vonnöten ist, berechnet werden. Datenabhängigkeiten zwischen einzelnen Komponenten entlang der Multiplikationsebenen sind erneut mit vertikalen beziehungsweise diagonalen Verbindungen zwischen den Kreisen dargestellt, jedoch sind jene Abhängigkeiten zwischen den Daten, die zuvor mit dem rechten Prozessor assoziiert wurden, rot markiert. All diese Werte hängen von den linkesten acht Komponenten von v ($s = 0$) des rechten Prozessors ab. Sie wurden daher zusätzlich mit einem roten Kreuz markiert. Sendet der rechte Prozessor diese Daten an den linken Prozessor, kann dieser alle in Abbildung 2.3 durch Kreise dargestellten Komponenten berechnen. Dabei sind Daten, die idealerweise durch den linken Prozessor berechnet würden, schwarz, und Daten, die idealerweise durch den rechten Prozessor berechnet würden, rot abgebildet. Hier wird also eine Redundanz innerhalb der Berechnungen, die PA1 leistet, illustriert: Die mit roten Kreisen verbundenen Rechnungen werden durch den linken, wie auch durch den rechten Prozessor ausgeführt.

Eine sinnvolle Setzung für die Praxis scheint $s < n/P$ zu sein, so dass ledig-

lich Daten von Nachbarprozessoren herangezogen werden müssen, statt, wie sonst möglich, von entfernteren Prozessoren. Genauer dürfte $s \ll n/P$ zusätzlich den Aufwand, der – relativ betrachtet – für redundante Berechnungen erbracht werden muss, verschwinden lassen.

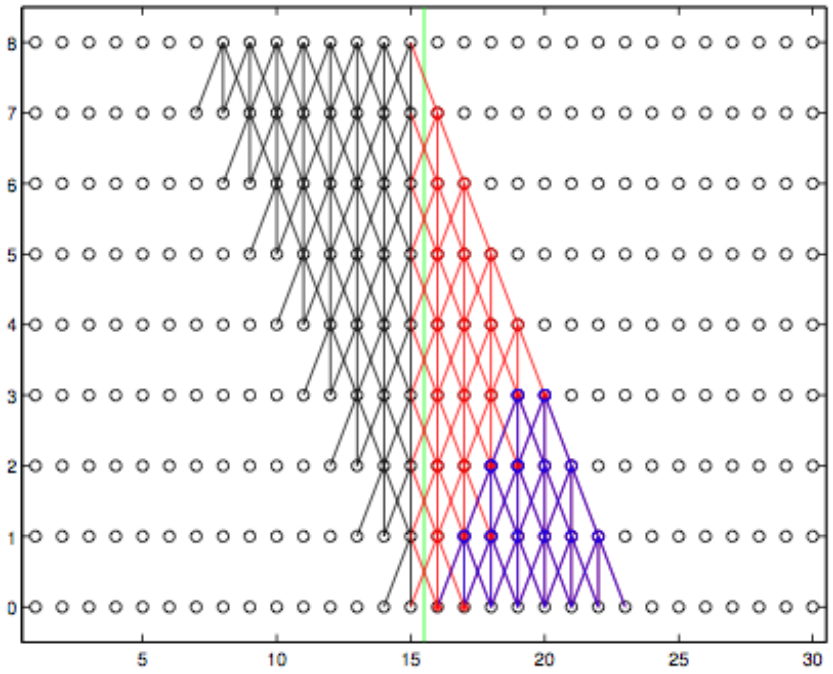

Abbildung 2.4: Abhängigkeiten zwischen benachbarten Prozessoren in PA2 [18]

Schließlich zeigt Abbildung 2.4 den in Abbildung 2.3 dargestellten Sachverhalt bezogen auf PA2 und demonstriert die damit verbundene Arbeitsweise: Die hier blau dargestellten Kreise können lokal ohne Datentransfer auf dem rechten Prozessor verarbeitet werden, wobei jene acht Kreise, die zusätzlich rote Kreuze beinhalten, dann an den linken Prozessor übertragen werden. So kann dieser dann die verbleibenden Komponenten, die mit roten oder schwarzen Linien verbundenen Kreisen assoziiert sind, berechnen. Dadurch kann die redundante Arbeit, die nun einzig auf die roten Kreise reduziert wurde, illustriert werden. Man überschlägt leicht, dass sich in diesem Fall die redundanten Berechnungen auf etwa die Hälfte im Vergleich zu PA1 verringern.

2.3.3 Allgemeine Realisierung

Mittels der in Abschnitt 2.3.1 eingeführten Notationen können wir nun die Algorithmen PA0, PA1 sowie PA2 explizit angeben. Bei der Beschreibung der Algorithmen benutzen wir übliche „Sende“- und „Empfange“-Operationen, um die Kommunikation zwischen Prozessoren zu beschreiben. Wie bereits erläutert, orientieren wir uns hier innerhalb des theoretischen Rahmens also eher an einer Distributed-Memory-Architektur. Anzumerken ist, dass in diesem Kontext der Begriff „Prozessor“ im engeren Sinne daher nun etwa speziell den Knoten eines Clusters meint.

Die Algorithmen stellen jeweils die Programmanweisungen für Prozessor q dar.

Algorithmus 2.6 (PA0)**:**

1: Schleife über $i = 1, \ldots, s$
2: Schleife über alle Prozessoren $r \neq q$
3: Sende alle $x_j^{(i-1)} \in R_q^{(i-1)}(V_r^{(i)})$ an Prozessor r
4: Ende Schleife
5: Schleife über alle Prozessoren $r \neq q$
6: Empfange alle $x_j^{(i-1)} \in R_r^{(i-1)}(V_q^{(i)})$ von Prozessor r
7: Ende Schleife
8: Berechne alle $x_j^{(i)} \in L_q^{(i)}$
9: Warte auf den Abschluss der obigen Empfangsoperationen
10: Berechne die verbleibenden $x_j^{(i)} \in V_q^{(i)} \setminus L_q^{(i)}$
11: Ende Schleife

Algorithmus 2.7 (PA1)**:**

1: Schleife über alle Prozessoren $r \neq q$
2: Sende alle $x_j^{(0)} \in R_q^{(0)}(V_r)$ an Prozessor r
3: Ende Schleife
4: Schleife über alle Prozessoren $r \neq q$
5: Empfange alle $x_j^{(0)} \in R_r^{(0)}(V_q)$ von Prozessor r
6: Ende Schleife
7: Schleife über $i = 1, \ldots, s$
8: Berechne alle $x_j^{(i)} \in L_q$ ▷ *schwarze Knoten, Abb. 2.2*
9: Ende Schleife
10: Warte auf den Abschluss der obigen Empfangsoperationen

11: Schleife über $i = 1, \ldots, s$

12: Berechne die verbleibenden $x_j^{(i)} \in R(V_q) \setminus L_q$ ▷ *schwarze und rote Knoten in Abb. 2.3*

13: Ende Schleife

Algorithmus 2.8 (PA2)**:**

1: Schleife über $i = 1, \ldots, s$

2: Berechne alle $x_j^{(i)} \in \bigcup_{r \neq q}(R(V_r) \cap L_q)$ ▷ *blaue Knoten in Abb. 2.4*

3: Ende Schleife

4: Schleife über alle Prozessoren $r \neq q$

5: Sende alle $x_j^{(0)} \in B_{r,q}$ an Prozessor r ▷ *blau-rote Knoten in Abb. 2.4*

6: Ende Schleife

7: Schleife über alle Prozessoren $r \neq q$

8: Empfange alle $x_j^{(0)} \in B_{r,q}$ von Prozessor r▷ *blau-rote Knoten in Abb. 2.4*

9: Ende Schleife

10: Schleife über $i = 1, \ldots, s$

11: Berechne alle $x_j^{(i)} \in L_q \setminus \bigcup_{r \neq q}(R(V_r) \cap L_q)$ ▷ *etwa lokal berechenbare Knoten des rechten Prozessors, jedoch ohne blaue Knoten in Abb. 2.4*

12: Ende Schleife

13: Warte auf den Abschluss der obigen Empfangsoperationen

14: Schleife über $i = 1, \ldots, s$

15: Berechne die verbleibenden $x_j^{(i)} \in R(V_q) \setminus (L_q \setminus \bigcup_{r \neq q}(R(V_q) \cap L_r))$ ▷ *etwa schwarze Knoten in Abb. 2.4*

16: Ende Schleife

2.4 Kommunikationsvermeidung im CG-Verfahren

Bislang haben wir den Matrixpotenzen-Kernel und drei zu seiner Umsetzung mögliche Algorithmen erläutert und diskutiert. In diesem Abschnitt soll nun seine konkrete Anwendung demonstriert werden. Exemplarisch ziehen wir dazu erneut das Verfahren der konjugierten Gradienten heran, welches wir in seiner gängigsten Lehrbuchformulierung bereits in Algorithmus 2.3 erfasst haben.

Konkret werden wir zunächst eine mathematisch äquivalente Umformulierung des Verfahrens der konjugierten Gradienten betrachten – das CG-Verfahren mit

Drei-Term-Rekursion. Wie wir erläutern werden, bietet diese Variante Vorteile bezüglich der Anwendbarkeit des Matrixpotenzen-Kernels, schließlich wurde dieser selbst über eine entsprechende Drei-Term-Rekursion definiert (vgl. Definition 2.5). Ferner werden wir dieses Verfahren in eine äußere und innere Schleife aufteilen. Die innere Schleife soll dabei gerade s Durchläufe umfassen, um hier später den Matrixpotenzen-Kernel einbinden zu können.

Um letztlich das Verfahren derart umzubauen, dass die zentrale Matrix-Vektor-Multiplikation nicht in jedem Schritt der inneren Schleife ausgeführt werden muss, definieren wir gewisse Koeffizientenvektoren $d^{(sk+j)}$, welche rekursiv bestimmt werden können, und bringen den Matrixpotenzen-Kernel zum Einsatz. Ein erstes Resultat dieses Abschnitts ist Algorithmus 2.12, welcher jedoch noch die Berechnung von Skalarprodukten innerhalb der inneren Iteration voraussetzt. Dies ist problematisch, da Skalarprodukten globale Kommunikationanforderungen inhärent sind. Hingegen muss die Systemmatrix A jedoch lediglich $(1 + \mathcal{O}(1))$-mal aus dem Speicher herangezogen werden, was bereits eine erste Verbesserung darstellt.

Wir werden diesen Algorithmus dahingehend erweitern, dass auch die Berechnung der Skalarprodukte umgangen wird, um so auch die mit ihnen verbundene klassische Kernelstruktur aufzubrechen. Dazu benötigen wir weitere Koeffizientenvektoren $g^{(sk+j)}$, welche sich ebenfalls über eine rekursive Berechnungsvorschrift bestimmen lassen. Aus diesem Schritt resultiert schließlich als ein finales Ergebnis Algorithmus 2.16.

2.4.1 CG-Verfahren mit Drei-Term-Rekursion

Naheliegend ist die eingangs erwähnte Idee, das CG-Verfahren um eine äußere Schleife zu ergänzen, so dass ein Durchlauf dieser Schleife s klassische Schritte in ihrem Innern ausführt. Schließlich soll auf diese Weise dann der Aufruf von s SpMV-Kernel durch die Anwendung des Matrixpotenzen-Kernel ersetzt werden.

In seiner Dissertation motiviert und beschreibt Hoemmen [18] dieses Vorgehen zunächst anhand der Standardformulierung des CG-Verfahren (Algorithmus 2.3). Dieses Vorhaben schlägt jedoch fehl, wie der Autor schildert, und liefert nur für $s = 2$ ein praktisch verwendbares Resultat. Der Ursprung dieses Fehlschlags findet sich in der inhärenten Abhängigkeit zwischen den Vektoren $r^{(k)}$ und $p^{(k)}$ in der Standardformulierung, die gerade einer Zwei-Term-Rekursion entspricht. Wir

verweisen an dieser Stelle auf diese Literatur für weitere Details.

Stattdessen verwendet Hoemmen [18] letztlich eine alternative Formulierung des Algorithmus, die sich einer Drei-Term-Rekursion bedient. Diese Formulierung ist mathematisch – also in exakter Arithmetik – äquivalent zu Algorithmus 2.3, das heißt sie liefert die gleichen Residuuen $r^{(k)}$ sowie Lösungsnäherungen $x^{(k)}$ wie die Ausgangsformulierung des CG-Verfahrens. Sie findet sich ursprünglich etwa im Lehrbuch von Saad [24] in Form von Algorithmus 6.19, enthält hier jedoch einen kleinen Fehler. Hoemmen [18] diskutiert diesen in Anhang C.1 seiner Arbeit, gibt aber auch eine Herleitung dieser Verfahrensvariante an.

Wir geben den entsprechenden Algorithmus nun an und bezeichnen dabei mit $0_{n,m}$ eine $(n \times m)$-Matrix mit allen Einträgen jeweils gleich Null. Für den Spezialfall $0_{n,1}$ ergibt sich also der Nullvektor mit n Einträgen. Der Name *CG3* geht ebenfalls auf Hoemmen [18] zurück.

Algorithmus 2.9 (CG3-Verfahren)**:**

Eingabe: SPD-Systemmatrix A, Anfangsnäherung $x^{(1)}$, rechte Seite b

1: Setze $x^{(0)} := 0_{n,1}, r^{(0)} := 0_{n,1}, r^{(1)} := b - Ax^{(1)}$
2: Schleife über $k = 1, 2, \ldots$ bis zur Konvergenz
3: $\quad w^{(k)} := Ar^{(k)}$
4: $\quad \alpha^{(k)} := (r^{(k)}, r^{(k)})_2$ und Konvergenzkontrolle
5: $\quad \nu^{(k)} := (w^{(k)}, r^{(k)})_2$
6: $\quad \gamma^{(k)} := \alpha^{(k)} / \nu^{(k)}$
7: $\quad$ Wenn $k = 1$, dann
8: $\quad\quad$ Setze $\rho^{(k)} := 1$
9: $\quad$ Sonst
10: $\quad\quad$ Setze $\rho^{(k)} := (1 - \frac{\gamma^{(k)} \alpha^{(k)}}{\gamma^{(k-1)} \alpha^{(k-1)} \rho^{(k-1)}})^{-1}$
11: $\quad$ Ende Wenn
12: $\quad x^{(k+1)} := \rho^{(k)}(x^{(k)} + \gamma^{(k)} r^{(k)}) + (1 - \rho^{(k)}) x^{(k-1)}$
13: $\quad r^{(k+1)} := \rho^{(k)}(r^{(k)} + \gamma^{(k)} w^{(k)}) + (1 - \rho^{(k)}) r^{(k-1)}$
14: Ende Schleife

In der in Algorithmus 2.9 dargestellten Variante des CG-Verfahren verschwindet der Suchrichtungsvektor $p^{(k)}$. Folglich finden sich keine störenden Abhängigkeiten zwischen den Suchrichtungen $p^{(k)}$ und dem aktuellen Residuum $r^{(k)}$. Wir

weisen überdies auf die namensgebende Drei-Term-Rekursion im Algorithmus hin: Sowohl die aktuelle Näherungslösung $x^{(k)}$ als auch das jeweilige Residuum $r^{(k)}$ hängen explizit von ihren beiden Vorgängern ab.

Um die bisher grob geschilderte Idee, s SpMV-Kernel durch je einen Aufruf des Matrixpotenzen-Kernels zu ersetzen, realisieren zu können, formulieren wir Algorithmus 2.9 derart, dass dieser über eine innere Schleife der Länge s und eine äußere Schleife iteriert. Letztere soll ausgeführt werden, bis ein geeignetes Konvergenzkriterium greift. Für die innere Schleife wollen wir dabei den Laufindex j, für die äußere Schleife den Laufindex k verwenden. Die Umformulierung ergibt sich direkt und bedarf keiner weiteren Erläuterung.

Algorithmus 2.10 (CG3 (mit innerer und äußerer Schleife))**:**

Eingabe: SPD-Systemmatrix A, Anfangsnäherung $x^{(1)}$, rechte Seite b, Schleifenlänge s

Setze $x^{(0)} := 0_{n,1}, r^{(0)} := 0_{n,1}, r^{(1)} := b - Ax^{(1)}$
Schleife über $k = 0, 1, \ldots$ bis zur Konvergenz
    Schleife über $j = 1, \ldots, s$
        $w^{(sk+j)} := Ar^{(sk+j)}$
        $\alpha^{(sk+j)} := (r^{(sk+j)}, r^{(sk+j)})_2$ und Konvergenzkontrolle
        $\nu^{(sk+j)} := (w^{(sk+j)}, r^{(sk+j)})_2$
        $\gamma^{(sk+j)} := \alpha^{(sk+j)} / \nu^{(sk+j)}$
        Wenn $sk + j = 1$, dann
            Setze $\rho^{(sk+j)} := 1$
        Sonst
            Setze $\rho^{(sk+j)} := (1 - \frac{\gamma^{(sk+j)}\alpha^{(sk+j)}}{\gamma^{(sk+j-1)}\alpha^{(sk+j-1)}\rho^{(sk+j-1)}})^{-1}$
        Ende Wenn
        $x^{(sk+j+1)} := \rho^{(sk+j)}(x^{(sk+j)} + \gamma^{(sk+j)}r^{(sk+j)}) + (1 - \rho^{(sk+j)})x^{(sk+j-1)}$
        $r^{(sk+j+1)} := \rho^{(sk+j)}(r^{(sk+j)} + \gamma^{(sk+j)}w^{(sk+j)}) + (1 - \rho^{(sk+j)})r^{(sk+j-1)}$
    Ende Schleife
Ende Schleife

2.4.2 Verbindung mit dem Matrixpotenzen-Kernel

Wir streben nun das Ziel an, den Matrixpotenzen-Kernel in Algorithmus 2.10 einzubinden, um so eine erste kommunikationsvermeidende Variante des CG-Verfahren

zu konstruieren. Zur Kennzeichnung der kommunikationsvermeidenden Natur solcher Verfahren, stellen wir jeweils das Präfix „CA“ (engl. *communication-avoiding*) der Algorithmenbezeichnung voran.

Beginnen müssen wir jedoch zunächst mit weiteren Bezeichnern sowie einigen theoretischen Überlegungen. Wir möchten zwecks einer einfachen Notation in ein Matrizenkalkül innerhalb der Beschreibung unserer Algorithmik übergehen. Wir betrachten dazu die Berechnung des Residuums innerhalb von Algorithmus 2.10

$$r^{(sk+j+1)} := \rho^{(sk+j)}(r^{(sk+j)} + \gamma^{(sk+j)} r^{(sk+j)}) + (1 - \rho^{(sk+j)}) r^{(sk+j-1)}. \tag{2.2}$$

Diese kann umformuliert werden zu

$$Ar^{(sk+j)} = \frac{1 - \rho^{(sk+j)}}{\rho^{(sk+j)}\gamma^{(sk+j)}} r^{(sk+j-1)} + \frac{1}{\gamma^{(sk+j)}} r^{(sk+j)} - \frac{1}{\rho^{(sk+j)}\gamma^{(sk+j)}} r^{(sk+j+1)}. \tag{2.3}$$

Da die Größen $\rho^{(sk+j)}$ sowie $\gamma^{(sk+j)}$ nun im Nenner eines Bruchs stehen, wollen wir diese Umformulierung kurz rechtfertigen: In exakter Arithmetik ist $\rho^{(sk+j)}$ immer definiert und nicht gleich null, solange wie Algorithmus 2.10 nicht zusammenbricht. Für weitere Details diesbezüglich verweisen wir auf den Anhang C.1 der Doktorarbeit von Hoemmen [18] beziehungsweise auf Unterkapitel 2.7 dieser Arbeit, da die Stabilität kommunikationsvermeidender Verfahren, die den Matrixpotenzen-Kernel verwenden, entscheidend von der Wahl der Basis in diesem abhängt. Bei exakter Rechnung ist die Größe

$$\gamma^{(sk+j)} = \frac{\alpha^{(sk+j)}}{\nu^{(sk+j)}} = \frac{(r^{(sk+j)}, r^{(sk+j)})_2}{(w^{(sk+j)}, r^{(sk+j)})_2} = \frac{(r^{(sk+j)}, r^{(sk+j)})_2}{(Ar^{(sk+j)}, r^{(sk+j)})_2}$$

ebenfalls ungleich null. Genauer betrachtet ist sie, solange wir die exakte Lösung x nicht erreichen und das Residuum somit nicht verschwindet, sogar echt positiv, was auf die Symmetrie und positive Definitheit der Systemmatrix A zurückgeht, welche wir für das CG-Verfahren voraussetzen mussten.

Ausgehend von der obigen Vektor-Rekursionsgleichung für das Residuum (2.3) übersetzen wir diesen Zusammenhang nun in ein Matrizenkalkül. Dazu bezeichnen wir die Matrix, welche spaltenweise aus s Residuenvektoren beginnend bei Index $sk + 1$ besteht, mit R_k, also

$$R_k := [r^{(sk+1)}, \ldots, r^{(sk+s)}]$$

für $k \in \mathbb{N}$. Fügen wir einen Residuenvektor mehr hinzu, sprechen wir von

$$\underline{R}_k := [R_k, r^{(sk+s+1)}].$$

Zur Sicherung einer kompakten Schreibweise treffen wir ferner die Definitionen

$$R_{-1} := 0_{n,s}, \quad \underline{R}_{-1} := [0_{n,s}, r^{(0)}]$$

und erinnern noch einmal daran, dass $0_{n,s}$ hier eine $(n \times s)$-Nullmatrix bezeichnet. Aus Beziehung (2.3) folgt nun der Zusammenhang

$$AR_k = \frac{1 - \rho^{(sk+1)}}{\rho^{(sk+j)}\gamma^{(sk+1)}} r^{(sk)} e_1^\top + \underline{R}_k \, \underline{T}_k \,. \tag{2.4}$$

Dabei ist $\underline{T}_k$ eine $(s+1 \times s)$-Matrix, definiert als

$$\underline{T}_k := \underline{\tilde{T}}_k \, \mathrm{diag}(\rho^{(sk+1)}\gamma^{(sk+1)}, \ldots, \rho^{(sk+s)}\gamma^{(sk+s)})^{-1}$$

mit

$$\underline{\tilde{T}}_k := \begin{bmatrix} \rho^{(sk+1)} & 1-\rho^{(sk+2)} & 0 & \ldots & 0 \\ -1 & \rho^{(sk+2)} & 1-\rho^{(sk+3)} & & \vdots \\ 0 & -1 & \ddots & \ddots & \\ \vdots & & \ddots & & 1-\rho^{(sk+s)} \\ & & & -1 & \rho^{(sk+s)} \\ 0 & & & & -1 \end{bmatrix},$$

sowie e_1 der erste kanonische Einheitsvektor passender Dimension. Obige Schreibweise beinhaltet zwar auf der Diagonalen der resultierenden Matrix $\underline{T}_k$ noch ungekürzte Größen, jedoch wird so eine kompakte Darstellung ermöglicht.

Sprechen wir von T_k, ist ferner der obere $(s \times s)$-Block von $\underline{T}_k$ gemeint, also $\underline{T}_k$ ohne ihre letzte Zeile. In Definition 2.5 haben wir für die Rückgabe des Matrixpotenzen-Kernels lediglich die Variable $\underline{V}$ verwandt. Wir wollen diese Schreibweise im Folgenden um den Index k erweitern: In diesem Kontext soll

$$\underline{V}_k := [V_k, v_{sk+s+1}] := [v_{sk+1}, \ldots, v_{sk+s+1}],$$

also mit

$$V_k := [v_{sk+1}, \ldots, v_{sk+s}],$$

gelten. Um einen Zusammenhang zwischen dem Produkt AV_k und $\underline{V}_k$ herzustellen, führen wir die *Basiswechsel-Matrix* $\underline{B}_k$ ein. Sie genügt gerade der Beziehung

$$AV_k = \underline{V}_k \, \underline{B}_k \, . \tag{2.5}$$

Ihre Erscheinung hängt einzig von der speziellen Wahl der Basis im Matrixpotenzen-Kernel ab. Hoemmen [18] gibt in Kapitel 7 seiner Arbeit Formeln für verschiedene Basen an. Im Falle der Standardbasis ergibt sich etwa die Matrix

$$\underline{B}_k = \begin{bmatrix} 0 & & & \dots & 0 \\ 1 & 0 & & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ & & & & \vdots \\ \vdots & & & \ddots & 0 \\ 0 & & \dots & 0 & 1 \end{bmatrix} = [e_2, \dots, e_{s+1}] \in \mathbb{R}^{s+1 \times s},$$

für die Konstruktion weiterer spezieller Basiswechselmatrizen $\underline{B}_k$ verweisen wir an dieser Stelle jedoch auf die oben genannte Literatur.

Wir weisen noch auf die Analogie innerhalb der Notation hin: Bei nicht unterstrichenen Größen fehlt im Vergleich zu unterstrichenen Größen jeweils eine Matrixspalte beziehungsweise -zeile.

Nachdem nun alle notwendigen Schreibweisen eingeführt sind, können wir eine erste kommunikationsvermeidende Variante des CG-Verfahrens (CA-CG) unter Ausnutzung des Matrixpotenzen-Kernels angeben. Diese ist in Algorithmus 2.12 dargestellt. Wie schon in Algorithmus 2.10 umfasst eine äußere Iteration jeweils s Iterationen des Standard-CG-Verfahrens, jedoch muss die Systemmatrix A nur $(1 + \mathcal{O}(1))$-mal aus dem Speicher gelesen werden, während es in der klassischen Variante noch s Male notwendig war.

Der wesentliche Unterschied zwischen der herkömmlichen Drei-Term-Variante und Algorithmus 2.12 ist nun, die Bestimmung eines $2s + 1$ Koeffizienten beinhaltenden Vektors $d^{(sk+j)}$ nach

$$Ar^{(sk+j)} = [R_{k-1}, \underline{V}_k] d^{(sk+j)}. \tag{2.6}$$

Dieses Vorgehen bewirkt, dass jetzt kein Matrix-Vektor-Produkt zur Bestimmung der Variable $w^{(sk+j)}$ mehr explizit zu berechnen ist: In Algorithmus 2.9 wurde diese

Größe noch nach $w^{(sk+j)} := Ar^{(sk+j)}$ berechnet, während wir nun nach Konstruktion von $d^{(sk+j)}$ die Berechnungsvorschrift

$$w^{(sk+j)} = [R_{k-1}, \underline{V_k}]d^{(sk+j)}$$

benutzen dürfen. Zwar handelt es sich hierbei nach wie vor um ein Matrix-Vektor-Produkt, jedoch besitzen die $d^{(sk+j)}$ tatsächlich nur wenige Nichtnull-Einträge, sind also dünnbesetzt. Aus diesem Grund bedeutet der obige Schritt nur eine Skalierung weniger Spalten, nicht aber die Ausführung eines vollständigen Matrix-Vektor-Produkts. Wir wollen dies in Abbildung 2.5 (Quelle: Hoemmen [18]) für $s = 5$ veranschaulichen.

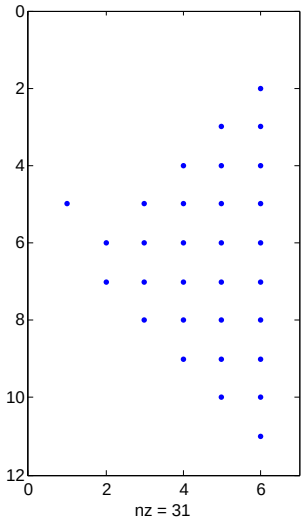

Abbildung 2.5: Besetzung der Matrix $[d^{(sk)}, \ldots, d^{(sk+s)}]$ für $s = 5$ [18]

Hier wird die sechsspaltige Matrix $[d^{(sk)}, \ldots, d^{(sk+s)}]$ dargestellt. Jeder blaue Punkt repräsentiert einen potentiellen Nichtnull-Eintrag. Wie man beobachten kann, nimmt die Anzahl der Nichtnull-Einträge mit steigendem j in $d^{(sk+j)}$ zu und erreicht ihr Maximum für $j = s$. Nach einer anfänglichen Unregelmäßigkeit erhöht sich die Anzahl der Nichtnull-Einträge in $d^{(sk+j)}$ um zwei bei jeder Erhöhung von j um eins.

Hoemmen [18] beschreibt in Unterkapitel 5.4.4 seiner Abhandlung die Herleitung einer umfassenden Rekursionsformel für die Komponenten des Vektors $d^{(sk+j)}$. Diese stellen wir ohne Beweis im folgenden Satz 2.11 zusammen und verweisen auf die zugehörige Literatur.

Satz 2.11: Für die Koeffizienten $d^{(sk+j)}$ gilt der rekursive Zusammenhang

$$d^{(sk+j)} = \begin{cases} 0_{2s+1,1} & \text{für } k=0, j=0, \\ [0_{1,s-1}, 1, 0_{1,s+1}]^\top & \text{für } k>0, j=0, \\ [0_{1,s}, \underline{B}_k(1,1), \underline{B}_k(2,1), 0_{1,s-1}]^\top & \text{für } k \geq 0, j=1, \\ \rho^{(sk+j-1)} d^{(sk+j-1)} + & \\ (1-\rho^{(sk+j-1)}) d^{(sk+j-2)} - & \\ \rho^{(sk+j-1)} \gamma^{(sk+j-1)} \cdot & \\ \begin{bmatrix} T_{k-1} d^{(sk+j-1)}(1:s) \\ -\frac{d^{(sk+j-1)}(s)}{\rho^{(sk)}\gamma^{(sk)}} e_1 \end{bmatrix} + & \text{für } k \geq 0, j=2,\ldots,s. \\ \rho^{(sk+j-1)} \gamma^{(sk+j-1)} \cdot & \\ \begin{bmatrix} 0_{s,1} \\ \underline{B}_k \, d^{(sk+j-1)}(s+1:2s) \end{bmatrix} & \end{cases} \tag{2.7}$$

Beweis: Hoemmen [18, Unterkapitel 5.4.4] □

Es folgt die erste vollständige Version des CA-CG-Algorithmus. In dieser Variante ist die explizite Berechnung von einigen Skalarprodukten jedoch noch notwendig. Wir müssen an dieser Stelle außerdem darauf hinweisen, dass der Algorithmus – so wie er in der Dissertation von Hoemmen [18] dargestellt wird – fehlerhaft ist. Innerhalb der zweiten inneren Iteration ($j=2$) der ersten äußeren Iteration ($k=0$) soll hier nach Satz 2.11 auf die nicht definierte Größe $\rho^{(sk)}\gamma^{(sk)} = \rho^{(0)}\gamma^{(0)}$ zugegriffen werden. Um dies zu umgehen, können diese Werte mit einer zu Beginn stattfindenden klassischen Iteration des CG3-Verfahrens bereitgestellt werden. Diese initiale Iteration unterdrücken wir innerhalb der Darstellung des folgenden Algorithmus 2.12 jedoch der Übersichtlichkeit halber.

Algorithmus 2.12 (CA-CG-Verfahren mit Skalarprodukten)**:**

Eingabe: SPD-Systemmatrix A, Anfangsnäherung $x^{(1)}$, rechte Seite b, Schleifenlänge s

1: Setze $x^{(0)} := 0_{n,1}, r^{(0)} := 0_{n,1}, r^{(1)} := b - Ax^{(1)}$
2: Schleife über $k = 0, 1, \ldots$ bis zur Konvergenz
3: $\quad v_{sk+1} := r^{(sk+1)}$
4: $\quad$ Berechne mittels Matrixpotenzen-Kernel: $\underline{V}_k = [v_{sk+1}, \ldots, v_{sk+s+1}]$
5: $\quad$ Schleife über $j = 1, \ldots, s$

6: Berechne den Vektor aus $2s+1$ Koeffizienten $d^{(sk+j)}$ nach $Ar^{(sk+j)} = [R_{k-1}, \underline{V_k}]d^{(sk+j)}$ mittels Satz 2.11
7: $w^{(sk+j)} := [R_{k-1}, \underline{V_k}]d^{(sk+j)}$
8: $\alpha^{(sk+j)} := (r^{(sk+j)}, r^{(sk+j)})_2$ und Konvergenzkontrolle
9: $\nu^{(sk+j)} := (w^{(sk+j)}, r^{(sk+j)})_2$
10: $\gamma^{(sk+j)} := \alpha^{(sk+j)} / \nu^{(sk+j)}$
11: Wenn $sk + j = 1$, dann
12: Setze $\rho^{(sk+j)} := 1$
13: Sonst
14: Setze $\rho^{(sk+j)} := (1 - \frac{\gamma^{(sk+j)}\alpha^{(sk+j)}}{\gamma^{(sk+j-1)}\alpha^{(sk+j-1)}\rho^{(sk+j-1)}})^{-1}$
15: Ende Wenn
16: $x^{(sk+j+1)} := \rho^{(sk+j)}(x^{(sk+j)} + \gamma^{(sk+j)}r^{(sk+j)}) + (1 - \rho^{(sk+j)})x^{(sk+j-1)}$
17: $r^{(sk+j+1)} := \rho^{(sk+j)}(r^{(sk+j)} + \gamma^{(sk+j)}w^{(sk+j)}) + (1 - \rho^{(sk+j)})r^{(sk+j-1)}$
18: Ende Schleife
19: Ende Schleife

2.4.3 Entfernen der Skalarprodukte

Schlussendlich wollen wir uns nun der Umformulierung der im obigen Algorithmus 2.12 noch enthaltenen Skalarprodukte widmen. Die Entfernung dieser ist insofern essentiell, als dass diese noch globale Kommunikation zwischen den Prozessoren notwendig machen. Befreien wir uns von ihnen, verspricht dies die Möglichkeit der vollständigen autarken Berechnung der inneren Schleife in Algorithmus 2.12 bei einer blockweisen Arbeitsaufteilung unter den Prozessoren. Wir gehen in Abschnitt 2.4.4 im Detail darauf ein. Das schließlich resultierende Verfahren wird wieder mathematisch äquivalent zum Ausgangsverfahren sein, also bei exakter Rechnung dieselbe Lösung wie Algorithmus 2.3 liefern.

Im Rahmen der folgenden Herleitung sind erneut weitere Definitionen zu treffen.

Definition 2.13 (Gram-Matrix)**:** Bezüglich der Vektoren $v_1, \ldots, v_n,\ w_1, \ldots, w_n$ ist die *Gram-Matrix* G innerhalb des Hilbertraums $\mathbb{R}^n$ definiert durch

$$G_{i,j} := (v_j, w_i)_2.$$

Zwecks einer kompakten Notation verwenden wir auch die Schreibweise

$$G = (V, W)_2 := V^\top W$$

für die Matrizen $V := [v_1, \ldots, v_n]$, $W := [w_1, \ldots, w_n]$, die die Vektoren v_j beziehungsweise w_j $(j = 1, \ldots, n)$ als Spalten enthalten. Das euklidische Skalarprodukt $(\cdot, \cdot)_2$ vererbt der Matrix G dabei unmittelbar seine Symmetrieeigenschaft für den Fall $V = W$.

Eine allgemeine Gram-Matrix kann mittels Reduktion parallel berechnet werden, etwa indem die zu summierenden Produkte der in ihr enthaltenen Skalarprodukte blockweise auf die zur Verfügung stehenden Prozessoren verteilt werden. Geschieht dies einheitlich, erhält also jeder Prozessor die gleichen Blöcke der in den jeweiligen Skalarprodukten enthaltenen Vektoren, kann auf diese Weise die bei der Kalkulation insgesamt auftretende Datenlatenz verringert werden, da so etwa in Systemen mit verteiltem Speicher die Anzahl der zu versendenden Nachrichten reduziert wird (vgl. auch Unterkapitel 2.2). Der zur Berechnung der Gram-Matrix notwendige Rechenaufwand entspricht zudem nach Konstruktion höchstens dem einer Matrix-Matrix-Multiplikation.

Wir weisen noch daraufhin, dass obige Definition von der gängigen Lehrbuchvariante abweicht, welche in der Regel die Gram-Matrix auf den Fall $V = W$ beschränkt.

Für unser Vorhaben, die Skalarprodukte in Algorithmus 2.12 zu ersetzen, benötigen wir speziell die Gram-Matrix, erzeugt ausschließlich durch die Spalten von $[R_{k-1}, \underline{V}_k]$. Hier gilt also $V = W$ (vgl. Definition 2.13). Daher setzen wir

$$G_k := ([R_{k-1}, \underline{V}_k], [R_{k-1}, \underline{V}_k])_2 = \begin{bmatrix} R_{k-1}^\top R_{k-1} & R_{k-1}^\top \underline{V}_k \\ \underline{V}_k^\top R_{k-1} & \underline{V}_k^\top \underline{V}_k \end{bmatrix}. \tag{2.8}$$

Obiger Ausdruck beinhaltet einige Möglichkeiten zu seiner effizienteren Handhabung: Da die Spalten von R_{k-1} nach Konstruktion des CG-Verfahrens orthogonal zueinander sind, handelt es sich bei dem nordwestlichen Block von G_k offensichtlich um eine Diagonalmatrix $D_{k-1} := R_{k-1}^\top R_{k-1}$. Für ihre Nichtnull-Einträge gilt

$$D_{k-1}(j, j) = (r^{(s(k-1)+j)}, r^{(s(k-1)+j)})_2 = \alpha^{(s(k-1)+j)}.$$

Wie in Algorithmus 2.12 ersichtlich, steht die Größe $\alpha^{(s(k-1)+j)}$ für jeden Durchlauf der äußeren Schleife aus der vorherigen Iteration bereits zur Verfügung und

muss nicht erneut berechnet werden. Weiterhin gilt aufgrund der Symmetrie der Gram-Matrix für den nordöstlichen sowie den südwestlichen Block von G_k der Zusammenhang

$$R_{k-1}^\top \underline{V}_k = (\underline{V}_k^\top R_{k-1})^\top.$$

Für die Gram-Matrix erhalten wir somit die Darstellung

$$G_k := \begin{bmatrix} D_{k-1} & (\underline{V}_k^\top R_{k-1})^\top \\ \underline{V}_k^\top R_{k-1} & \underline{V}_k^\top \underline{V}_k \end{bmatrix}. \tag{2.9}$$

Aufgrund ihrer bereits erörterten Eigenschaften können wir ihre Berechnung mittels zweier Reduktionen über die zwei verbleibenden nichttrivialen Blöcke $\underline{V}_k^\top R_{k-1}$ sowie $\underline{V}_k^\top \underline{V}_k$ innerhalb einer parallelen Implementierung bewerkstelligen.

Ist die Gram-Matrix berechnet, lassen sich mit ihrer Hilfe die Skalarprodukte

$$\alpha^{(sk+j)} := (r^{(sk+j)}, r^{(sk+j)})_2, \tag{2.10}$$

$$\nu^{(sk+j)} := (Ar^{(sk+j)}, r^{(sk+j)})_2 \tag{2.11}$$

in Algorithmus 2.12 ersetzen. Dazu betrachten wir erneut den im vorherigen Abschnitt eingeführten Koeffizientenvektor $d^{(sk+j)}$. Dieser wurde über Gleichung (2.6) definiert:

$$Ar^{(sk+j)} = [R_{k-1}, \underline{V}_k] d^{(sk+j)}$$

Analog definieren wir nun einen weiteren Koeffizientenvektor $g^{(sk+j)}$ über eine ähnliche Bedingung, die sich aus (2.6) durch Entfernen der Matrix A ergibt:

$$r^{(sk+j)} = [R_{k-1}, \underline{V}_k] g^{(sk+j)} \tag{2.12}$$

Mit Hilfe der Literatur (vgl. Hoemmen [18, Unterkapitel 5.4.5]) ist es wieder möglich $g^{(sk+j)}$ über eine rekursive Fallunterscheidung zu bestimmen. Diese fasst der folgende Satz zusammen.

Satz 2.14: Für die Koeffizienten $g^{(sk+j)}$ gilt der rekursive Zusammenhang

$$g^{(sk+j)} = \begin{cases} 0_{2s+1,1} & \text{für } k = 0, j = 0, \\ [0_{1,s-1}, 1, 0_{1,s+1}]^\top & \text{für } k > 0, j = 0, \\ [0_{1,s}, 1, 0_{1,s}]^\top & \text{für } k \geq 0, j = 1, \\ \rho^{(sk+j-1)} g^{(sk+j-1)} - & \\ \rho^{(sk+j-1)} \gamma^{(sk+j-1)} d^{(sk+j-1)} + & \text{für } k \geq 0, j = 2, \ldots, s. \\ (1 - \rho^{(sk+j-1)}) g^{(sk+j-2)} & \end{cases} \tag{2.13}$$

Beweis: Hoemmen [18, Unterkapitel 5.4.5] □

Über die Koeffizientenvektoren $d^{(sk+j)}$ sowie $g^{(sk+j)}$ und die Gram-Matrix G_k lassen sich nun die Skalarprodukte (2.10) und (2.11) berechnen:

Satz 2.15: Mit den Koeffizientenvektoren $d^{(sk+j)}$ sowie $g^{(sk+j)}$ und der Gram-Matrix G_k gilt

$$\alpha^{(sk+j)} = g^{(sk+j)^\top} G_k g^{(sk+j)}, \tag{2.14}$$

$$\nu^{(sk+j)} = g^{(sk+j)^\top} G_k d^{(sk+j)}. \tag{2.15}$$

Beweis: Hoemmen [18, Unterkapitel 5.4.5] □

Mit Satz 2.15 können wir den finalen CA-CG-Algorithmus nun angeben, welcher auch die Umgehung der Skalarprodukte beinhaltet. Unverändert ist hier eine kleine Korrektur notwendig: Unser Hinweis kurz vor der Angabe von Algorithmus 2.12 hat hier ebenso Gültigkeit.

Algorithmus 2.16 (CA-CG-Verfahren)**:**

Eingabe: SPD-Systemmatrix A, Anfangsnäherung $x^{(1)}$, rechte Seite b, Schleifenlänge s

1: Setze $x^{(0)} := 0_{n,1}, r^{(0)} := 0_{n,1}, R_{-1} := 0_{n,s}, r^{(1)} := b - Ax^{(1)}$
2: Schleife über $k = 0, 1, \ldots$ bis zur Konvergenz
3: $\quad v_{sk+1} := r^{(sk+1)}$
4: $\quad$ Berechne mittels Matrixpotenzen-Kernel: $\underline{V_k} = [v_{sk+1}, \ldots, v_{sk+s+1}]$
5: $\quad$ Berechne $G_k := \begin{bmatrix} D_{k-1} & (\underline{V_k}^\top R_{k-1})^\top \\ \underline{V_k}^\top R_{k-1} & \underline{V_k}^\top \underline{V_k} \end{bmatrix}$
6: $\quad$ Schleife über $j = 1, \ldots, s$
7: $\quad\quad$ Berechne den Vektor aus $2s+1$ Koeffizienten $d^{(sk+j)}$ nach $Ar^{(sk+j)} = [R_{k-1}, \underline{V_k}]d^{(sk+j)}$ mittels Satz 2.11
8: $\quad\quad$ Berechne den Vektor aus $2s+1$ Koeffizienten $g^{(sk+j)}$ nach $r^{(sk+j)} = [R_{k-1}, \underline{V_k}]g^{(sk+j)}$ mittels Satz 2.14
9: $\quad\quad w^{(sk+j)} := [R_{k-1}, \underline{V_k}]d^{(sk+j)}$
10: $\quad\quad \alpha^{(sk+j)} := g^{(sk+j)^\top} G_k g^{(sk+j)}$ und Konvergenzkontrolle
11: $\quad\quad \nu^{(sk+j)} := g^{(sk+j)^\top} G_k d^{(sk+j)}$
12: $\quad\quad \gamma^{(sk+j)} := \alpha^{(sk+j)} / \nu^{(sk+j)}$

13: Wenn $sk + j = 1$, dann
14: Setze $\rho^{(sk+j)} := 1$
15: Sonst
16: Setze $\rho^{(sk+j)} := (1 - \frac{\gamma^{(sk+j)}\alpha^{(sk+j)}}{\gamma^{(sk+j-1)}\alpha^{(sk+j-1)}\rho^{(sk+j-1)}})^{-1}$
17: Ende Wenn
18: $x^{(sk+j+1)} := \rho^{(sk+j)}(x^{(sk+j)} + \gamma^{(sk+j)}r^{(sk+j)}) + (1 - \rho^{(sk+j)})x^{(sk+j-1)}$
19: $r^{(sk+j+1)} := \rho^{(sk+j)}(r^{(sk+j)} + \gamma^{(sk+j)}w^{(sk+j)}) + (1 - \rho^{(sk+j)})r^{(sk+j-1)}$
20: Ende Schleife
21: Ende Schleife

2.4.4 Fazit

Mit Algorithmus 2.16 steht nun ein Verfahren zur Verfügung, dass in der Lage ist mittels Matrixpotenzen-Kernel Kommunikation zu vermeiden. In klassischen Formulierungen des Verfahrens der konjugierten Gradienten wird in vergleichsweise naiven Parallelisierungsstragegien typischerweise nur innerhalb einzelner algebraischen Kernel, wie solche zur Berechnung von Skalar- oder Matrix-Vektor-Produkten, parallelisiert, was jeweils zu einem globalen Synchronisationspunkt zwischen den Kernel-Aufrufen führt. In unserem neuen Verfahren ist dies gerade nicht der Fall, da innerhalb der inneren Schleife jeder Prozessor s klassische CG-Verfahrensschritte autark berechnen kann ohne kommunizieren zu müssen. Erst bei der Erhöhung von k, also in der äußeren Schleife, finden sich Kommunikationspunkte.

Zur Realisierung dieses äußeren Abschnitts haben wir in Unterkapitel 2.3.3 verschiedene Algorithmen vorgestellt, um die Berechnung der Rückgabevektoren des Matrixpotenzen-Kernels blockweise parallelisiert zu bewerkstelligen. Ebenso kann die Bereitstellung der Gram-Matrix mittels zweier blockweiser Reduktionen realisiert werden. Dies begründet sich durch ihre oben beschriebenen Eigenschaften.

Innerhalb der inneren Schleife können bei einer erneuten blockweisen Parallelisierung die Koeffizientenvektoren $d^{(sk+j)}$ und $g^{(sk+j)}$ mit Hilfe von Satz 2.11 und Satz 2.14 redundant auf jedem Prozessor berechnet werden. Dadurch kann jeder Prozessor seine Zeilen von

$$w^{(sk+j)} := [R_{k-1}, \underline{V_k}]d^{(sk+j)}$$

für alle $j = 1, \ldots, s$ ohne Kommunikation bestimmen. Ebenso ist für die Berech-

nung von

$$\alpha^{(sk+j)} := g^{(sk+j)^\top} G_k g^{(sk+j)}$$

sowie

$$\nu^{(sk+j)} := g^{(sk+j)^\top} G_k d^{(sk+j)}$$

und schließlich

$$\gamma^{(sk+j)} := \alpha^{(sk+j)} / \nu^{(sk+j)}$$

kein Informationsaustausch zwischen den Recheneinheiten notwendig. Letztlich können die neuen Iterierten der Näherungslösung $x^{(sk+j+1)}$ sowie das aktuelle Residuum $r^{(sk+j+1)}$ ebenfalls kommunikationsfrei bestimmt werden.

Erst bei Beendigung der inneren Schleife ist zur jeweiligen Ausführung des Matrixpotenzen-Kernel sowie zur Berechnung der Gram-Matrix G_k wieder eine Synchronisation notwendig. Algorithmus 2.16, dessen äußere Iteration s Iterationen seines klassischen Pendants entspricht, liest die Systemmatrix A also lediglich $(1 + \mathcal{O}(1))$-mal aus dem langsamen Speicher, während eine klassische CG-Formulierung $\mathcal{O}(s)$ Lesevorgänge für ein mathematisch äquivalentes Resultat benötigt.

2.5 Kommunikationsvermeidung im vorkonditionierten CG-Verfahren

2.5.1 Vorkonditionierung

Zur Verbesserung der spektralen Eigenschaften (vgl. Definition A.1 und A.2) des Ausgangssystems $Ax = b$ wird formal von links mit der Inversen einer regulären Matrix M derselben Dimension multipliziert:

$$M^{-1}Ax = M^{-1}b. \tag{2.16}$$

Dieses Vorgehen wird als *Linksvorkonditionierung* und M als *Linksvorkonditionierer* bezeichnet. Da wir uns in diesem Kapitel zunächst auf Linksvorkonditionierung

beschränken wollen, verwenden wir auch kurz nur die Bezeichnungen *Vorkonditionierung* beziehungsweise *Vorkonditionierer*.

Wir werden uns im späteren Verlauf dieser Arbeit auch der *Rechtsvorkonditionierung* bedienen. Hier wird formal zunächst das System

$$AM^{-1}y = b \tag{2.17}$$

gelöst. Die eigentliche Lösung des Ausgangssystems ergibt sich dann aus dem Zusammenhang

$$x = M^{-1}y. \tag{2.18}$$

Wir wollen nicht unerwähnt lassen, dass auch eine *beidseitige Vorkonditionierung* möglich ist. Da dieser Begriff im Wesentlichen von selbsterklärender Natur ist und wir im Verlauf dieser Arbeit nicht mehr darauf Bezug nehmen, verweisen wir für weitere Informationen auf einschlägige Literatur, etwa das Vorlesungsskriptum von Göddeke [17, Kapitel 4].

Klar ist, dass in allen Fällen der Effekt bezüglich der spektralen Eigenschaften des resultierenden linearen Gleichungssystems besonders groß ist, wenn die Matrix M^{-1} in einem gewissen Sinne besonders „nahe“ an der inversen Systemmatrix A^{-1} liegt.

Typischerweise wird in einer tatsächlichen Implementierung das Matrixprodukt $M^{-1}A$ niemals berechnet. Stattdessen berechnet man das Matrix-Vektor-Produkt $q := M^{-1}z$ mit einem Eingabevektor z und einem Ausgabevektor q. Ersterer wird im Falle des CG-Verfahrens in der Regel das aktuelle Residuum sein. Auf diese Weise lässt sich eine aufwendige Matrix-Matrix-Multiplikation umgehen.

Zur speziellen Konfiguration von M gibt es eine Vielzahl an Möglichkeiten. Die damit verbundenen Verfahren können jedoch in zwei grundlegende Klassen aufgeteilt werden: *implizite* und *explizite Vorkonditionierer*. Erstere geben direkt die Matrix M vor. Die Bezeichnung „implizit“ ergibt sich dann aus dem Zusammenhang, dass bei jeder Anwendung ein System

$$Mq = z$$

gelöst wird, welches schließlich äquivalent zu $q = M^{-1}z$ ist. Im Kontext der letzteren Bezeichnung berechnet das entsprechende Verfahren direkt eine zumeist dünnbesetzte Näherung M^{-1} an A^{-1}. Somit reduziert sich die Anwendung des Vorkonditionierers formal auf eine einfache Matrix-Vektor-Multiplikation. Auch für eine

Übersicht möglicher Verfahren möchten wir erneut auf das Vorlesungsskriptum von Göddeke [17, Kapitel 4] referenzieren.

Wir weisen überdies darauf hin, dass wir jedes Verfahren, welches wir um eine Vorkonditionierungsroutine erweitern, mit dem Präfix „LP“ (engl. *left-preconditioned*) ausstatten wollen.

2.5.2 Vorkonditioniertes CG-Verfahren mit Drei-Term-Rekursion

Wir werden nun die bereits in Form von Algorithmus 2.9 vorgestellte Variante des Verfahrens der konjugierten Gradienten mit Drei-Term-Rekursion (CG3) mit der Anwendung eines Vorkonditionierers verbinden. Das resultierende Verfahren nennen wir LP-CG3 und stellen es in Algorithmus 2.17 dar.

Algorithmus 2.17 (LP-CG3-Verfahren)**:**

Eingabe: SPD-Systemmatrix A, Anfangsnäherung $x^{(1)}$, rechte Seite b, Vorkonditionierer M

1: Setze $x^{(0)} := 0_{n,1}, z^{(0)} := 0_{n,1}, q^{(0)} := 0_{n,1}$
2: Setze $z^{(1)} := b - Ax^{(1)}, q^{(1)} := M^{-1}z^{(1)}$
3: Schleife über $k = 1, 2, \ldots$ bis zur Konvergenz
4: $\quad \alpha^{(k)} := (z^{(k)}, q^{(k)})_2$ und Konvergenzkontrolle
5: $\quad w^{(k)} := Aq^{(k)}$
6: $\quad v^{(k)} := M^{-1}w^{(k)}$
7: $\quad \nu^{(k)} := (w^{(k)}, q^{(k)})_2$
8: $\quad \gamma^{(k)} := \alpha^{(k)}/\nu^{(k)}$
9: $\quad$ Wenn $k = 1$, dann
10: $\quad\quad$ Setze $\rho^{(k)} := 1$
11: $\quad$ Sonst
12: $\quad\quad$ Setze $\rho^{(k)} := (1 - \frac{\gamma^{(k)}\alpha^{(k)}}{\gamma^{(k-1)}\alpha^{(k-1)}\rho^{(k-1)}})^{-1}$
13: $\quad$ Ende Wenn
14: $\quad x^{(k+1)} := \rho^{(k)}(x^{(k)} + \gamma^{(k)}q^{(k)}) + (1 - \rho^{(k)})x^{(k-1)}$
15: $\quad z^{(k+1)} := \rho^{(k)}(z^{(k)} + \gamma^{(k)}w^{(k)}) + (1 - \rho^{(k)})z^{(k-1)}$
16: $\quad q^{(k+1)} := \rho^{(k)}(q^{(k)} + \gamma^{(k)}v^{(k)}) + (1 - \rho^{(k)})q^{(k-1)}$
17: Ende Schleife

Wir weisen darauf hin, dass entgegen dem bisherigen Vorgehen das Residuum nicht mehr mit $r^{(k)}$, sondern nunmehr mit $z^{(k)}$ bezeichnet wird. Es handelt sich dabei jedoch tatsächlich um das reale Residuum. Für sein vorkonditioniertes Pendant führt der obige Algorithmus die Bezeichnung $q^{(k)}$.

Als Abbruchkriterium beziehungsweise zur Konvergenzkontrolle dient nun außerdem nicht mehr die Norm des Residuums, sondern die Mischform $\alpha^{(k)} := (z^{(k)}, q^{(k)})_2$. Diese wird gelegentlich auch als *M-Norm* bezeichnet (vgl. Hoemmen [18, Seite 239]).

Wir wollen diesen Algorithmus im Folgenden in analoger Weise zu Unterkapitel 2.4 zu einem kommunikationsvermeidenden Verfahren erweitern. Dazu überlegen wir uns sukzessive, wie die einzelnen Begrifflichkeiten zur Anwendung eines Vorkonditionierers übertragen werden müssen. Am Ende wird dann eine vorkonditionierte Variante von Algorithmus 2.16 stehen. Für dieses Verfahren verwenden wir entsprechend unserer Nomenklatur die Bezeichnung „LP-CA-CG“ und stellen es in Algorithmus 2.22 dar.

Auch dieses Verfahren wird wieder – exakte Arithmetik vorausgesetzt – mathematisch äquivalent zu seinem Ausgangsverfahren sein. Da wir ferner wieder eine innere wie äußere Schleife verwenden wollen, bedienen wir uns erneut einer doppelten Indizierung unserer Variablen, etwa $x^{(sk+j)}$ statt $x^{(k)}$ wie bisher. Für die innere Iteration werden wir dabei nach wie vor den Index j, für die äußere Iteration den Index k verwenden.

Innerhalb eines äußeren Schleifendurchlaufs des LP-CA-CG-Verfahrens muss der Matrixpotenzen-Kernel nun zweimal statt einmal ausgeführt werden. Dabei sollen Basen $\{w_{sk+1}, \ldots, w_{sk+s+1}\}$ beziehungsweise $\{v_{sk+1}, \ldots, v_{sk+s+1}\}$ von geeigneten Krylow-Unterräumen berechnet werden. Zwecks einer besseren Beschreibung des resultierenden Verfahrens schreiben wir diese wieder als Spalten innerhalb einer Matrix, verwenden für sie aber die im klassischen Sinne formal nicht vollständig korrekte Bezeichnung „Basis“. Wir wollen die benötigten Matrizen in der folgenden Definition genauer erklären.

Definition 2.18 (Linksseitige und rechtsseitige Basis)**:** Als *linksseitige* beziehungsweise *rechtsseitige Basis* $\underline{W}_k$ beziehungsweise $\underline{V}_k$ bezeichnen wir die Matrizen

$$\begin{aligned} \underline{W}_k &:= [w_{sk+1}, \ldots, w_{sk+s+1}] \text{ bzw.} \\ \underline{V}_k &:= [v_{sk+1}, \ldots, v_{sk+s+1}], \end{aligned}$$

falls

$$\begin{aligned}
\text{image}(\underline{W}_k) &= \mathcal{K}_{s+1}(AM^{-1}, w_{sk+1}) \\
&= \text{span}\{w_{sk+1}, AM^{-1}w_{sk+1}, \ldots, (AM^{-1})^s w_{sk+1}\} \text{ bzw.} \\
\text{image}(\underline{V}_k) &= \mathcal{K}_{s+1}(M^{-1}A, v_{sk+1}) \\
&= \text{span}\{v_{sk+1}, M^{-1}Av_{sk+1}, \ldots, (M^{-1}A)^s v_{sk+1}\}
\end{aligned}$$

gilt.

Als Ausgangsvektoren werden wir für diese Ausführungen des Matrixpotenzen-Kernels die Residuen $w_{sk+1} := z^{(sk+1)}$ beziehungsweise $v_{sk+1} := q^{(sk+1)}$ nutzen. Nach Konstruktion gilt dann der Zusammenhang

$$v_{sk+1} = M^{-1}w_{sk+1} \tag{2.19}$$

über alle äußeren Iterationen k.

Um erneut kompakte Notationen innerhalb des LA-CA-CG-Algorithmus zu gewährleisten, führen wir noch die Bezeichnungen

$$\begin{aligned}
Z_k &:= [z_{sk+1}, \ldots, z_{sk+s}] \text{ sowie} \\
Q_k &:= [q_{sk+1}, \ldots, q_{sk+s}]
\end{aligned}$$

für die spaltenweise aus (vorkonditionierten) Residuen bestehenden Matrizen Z_k und Q_k ein. Für die um einen Vektor erweiterten Analoga dieser Matrizen ergänzen wir die Symbole wieder um einen Unterstrich:

$$\begin{aligned}
\underline{Z}_k &:= [Z_k, z_{sk+s+1}], \\
\underline{Q}_k &:= [Q_k, q_{sk+s+1}].
\end{aligned}$$

Die $(s+1 \times s)$-Matrix $\underline{T}_k$ definieren wir analog zum unvorkonditionierten Fall im vorherigen Abschnitt. Für sie gilt

$$\underline{T}_k := \underline{\tilde{T}}_k \, \text{diag}(\rho^{(sk+1)}\gamma^{(sk+1)}, \ldots, \rho^{(sk+s)}\gamma^{(sk+s)})^{-1}$$

mit

$$\underline{\tilde{T}}_k := \begin{bmatrix} \rho^{(sk+1)} & 1-\rho^{(sk+2)} & 0 & \dots & 0 \\ -1 & \rho^{(sk+2)} & 1-\rho^{(sk+3)} & & \vdots \\ 0 & -1 & \ddots & \ddots & \\ \vdots & & \ddots & & 1-\rho^{(sk+s)} \\ & & & -1 & \rho^{(sk+s)} \\ 0 & & & & -1 \end{bmatrix}.$$

T_k bezeichne darüber hinaus erneut den oberen $(s \times s)$-Block von $\underline{T}_k$, also $\underline{T}_k$ ohne ihre letzte Zeile.

Im vorkonditionierten Fall müssen wir uns hingegen einer abgewandelten Version der Gram-Matrix G_k bedienen. Hier setzen wir

$$G_k := ([Q_{k-1}, \underline{V}_k], [Z_{k-1}, \underline{W}_k])_2 = \begin{bmatrix} Q_{k-1}^\top Z_{k-1} & Q_{k-1}^\top \underline{W}_k \\ \underline{V}_k^\top Z_{k-1} & \underline{V}_k^\top \underline{W}_k \end{bmatrix}. \tag{2.20}$$

Wir verzichten auf eine detaillierte Herleitung und verweisen daher auf die Dissertation von Hoemmen [18, Unterkapitel 4.3.4], wollen aber erneut auf die Vorteile bezüglich der Berechenbarkeit dieser Matrix hinweisen: Der nordwestliche Block degeneriert wie bisher zu einer Diagonalmatrix, welche wir auch weiterhin mit D_{k-1} bezeichnen wollen. Für diese Matrix gilt nun wieder der Zusammenhang

$$D_{k-1}(j,j) = \alpha^{(s(k-1)+j)}.$$

Ferner kann man für G_k erneut

$$\underline{V}_k^\top Z_{k-1} = (Q_{k-1}^\top \underline{W}_k)^\top$$

zeigen. Insgesamt vereinfacht sich G_k also zu

$$G_k := ([Q_{k-1}, \underline{V}_k], [Z_{k-1}, \underline{W}_k])_2 = \begin{bmatrix} D_{k-1} & Q_{k-1}^\top \underline{W}_k \\ (Q_{k-1}^\top \underline{W}_k)^\top & \underline{V}_k^\top \underline{W}_k \end{bmatrix}. \tag{2.21}$$

Da $\underline{V}_k^\top \underline{W}_k = \underline{V}_k^\top M \underline{V}_k$ gilt, ist G_k auch insgesamt symmetrisch, falls der Vorkonditionierer M symmetrisch gewählt wurde, was jedoch vom CG-Verfahren selbst vorausgesetzt wird.

Wir betrachten nun die Koeffizientenvektoren $d^{(sk+j)}$ beziehungsweise $g^{(sk+j)}$ im vorkonditionierten Fall. Diese bestehen wieder aus $2s+1$ Koeffizienten. Dabei sei nun $d^{(sk+j)}$ als der Vektor definiert, der

$$[Z_{k-1}, \underline{W}_k] d^{(sk+j)} = Aq^{(sk+j)}$$

sowie

$$[Q_{k-1}, \underline{V}_k] d^{(sk+j)} = M^{-1} Aq^{(sk+j)}$$

erfüllt. Analog definieren wir $g^{(sk+j)}$ über die Beziehungen

$$[Z_{k-1}, \underline{W}_k] g^{(sk+j)} = z^{(sk+j)}$$

und

$$[Q_{k-1}, \underline{V}_k] g^{(sk+j)} = q^{(sk+j)}.$$

Auch unter der Verwendung eines Vorkonditionierers können wir nun erneut rekursive Berechnungsvorschriften für die Koeffizientenvektoren $d^{(sk+j)}$ und $g^{(sk+j)}$ in Sätzen formulieren:

Satz 2.19: Für die Koeffizienten $d^{(sk+j)}$ gilt der rekursive Zusammenhang

$$d^{(sk+j)} = \begin{cases} [0_{s-2,1}, \frac{1-\rho^{(sk)}}{\rho^{(sk)}\gamma^{(sk)}}, \frac{1}{\gamma^{(sk)}}, \frac{-1}{\rho^{(sk)}\gamma^{(sk)}}, 0_{s,1}] & \text{für } j = 0, \\ [0_{s,1}, \underline{B}_k(:,1), \underline{B}_k(:,2), 0_{s-1,1}]^\top & \text{für } j = 1, \\ \rho^{(sk+j-1)}\gamma^{(sk+j-1)} \cdot & \\ \begin{bmatrix} -T_{k-1} & 0_{s,s} \\ 0_{s+1,s} & \frac{1}{\rho^{(sk)}\gamma^{(sk)}} e_1 e_1^\top - \underline{B}_k \end{bmatrix} d^{(sk+j-1)} - & \text{für } j = 2, \ldots, s. \\ \frac{1-\rho^{(sk+j-1)}}{d^{(sk+j-2)}} - \rho^{(sk+j-1)} d^{(sk+j-1)} & \end{cases} \tag{2.22}$$

mit $k \in \mathbb{N}_0$.

Beweis: Hoemmen [18, Unterkapitel 5.5.4] □

Satz 2.20: Für die Koeffizienten $g^{(sk+j)}$ gilt der rekursive Zusammenhang

$$g^{(sk+j)} = \begin{cases} 0_{2s+1,1} & \text{für } k=0, j=0, \\ [0_{s-1,1}, 1, 0_{s+1,1}]^\top & \text{für } k>0, j=0, \\ [0_{s,1}, 1, 0_{s,1}]^\top & \text{für } k \geq 0, j=1, \\ -\rho^{(sk+j-1)}\gamma^{(sk+j-1)}d^{(sk+j-1)} + & \\ \rho^{(sk+j-1)}g^{(sk+j-1)} + & \text{für } k \geq 0, j=2,\ldots,s. \\ (1-\rho^{(sk+j-1)})g^{(sk+j-2)} & \end{cases} \tag{2.23}$$

Beweis: Hoemmen [18, Unterkapitel 5.5.5] □

Mit Hilfe dieser Koeffizientenvektoren und der Gram-Matrix G_k kann nun die explizite Berechnung der störenden Skalarprodukte

$$\alpha^{(sk+j)} := (z^{(sk+j)}, q^{(sk+j)})_2, \tag{2.24}$$

$$\nu^{(sk+j)} := (Aq^{(sk+j)}, q^{(sk+j)})_2 \tag{2.25}$$

erneut umgangen werden:

Satz 2.21: Mit den Koeffizientenvektoren $d^{(sk+j)}$ sowie $g^{(sk+j)}$ und der Gram-Matrix G_k gilt

$$\alpha^{(sk+j)} = g^{(sk+j)^\top} G_k g^{(sk+j)}, \tag{2.26}$$

$$\nu^{(sk+j)} = g^{(sk+j)^\top} G_k d^{(sk+j)}. \tag{2.27}$$

Beweis: Hoemmen [18, Unterkapitel 5.5.6] □

Schließlich sind wir in der Lage, den LP-CA-CG-Algorithmus anzugeben. Weiterhin ist hier eine Korrektur notwendig: Unser Hinweis kurz vor der Angabe von Algorithmus 2.12 hat hier wieder Gültigkeit.

Algorithmus 2.22 (LP-CA-CG-Verfahren)**:**

Eingabe: SPD-Systemmatrix A, Anfangsnäherung $x^{(1)}$, rechte Seite b, Schleifenlänge s, Vorkonditionierer M

Setze $x^{(0)} := 0_{n,1}$

Setze $z^{(0)} := 0_{n,1}, z^{(1)} := b - Ax^{(1)}$

Setze $q^{(0)} := 0_{n,1}, q^{(1)} := M^{-1}z^{(1)}$

Schleife über $k = 0, 1, \ldots$ bis zur Konvergenz

  $v_{sk+1} := z^{(sk+1)}$

  $w_{sk+1} := q^{(sk+1)}$

  Berechne mittels Matrixpotenzen-Kernel: $\underline{V_k} = [v_{sk+1}, \ldots, v_{sk+s+1}]$

  Berechne mittels Matrixpotenzen-Kernel: $\underline{W_k} = [w_{sk+1}, \ldots, w_{sk+s+1}]$

  Berechne $G_k := \begin{bmatrix} D_{k-1} & (\underline{V_k}^\top Z_{k-1})^\top \\ \underline{V_k}^\top Z_{k-1} & \underline{V_k}^\top \underline{W_k} \end{bmatrix}$

  Schleife über $j = 1, \ldots, s$

    Berechne den Vektor aus $2s + 1$ Koeffizienten $d^{(sk+j)}$ mittels Satz 2.19

    Berechne den Vektor aus $2s + 1$ Koeffizienten $g^{(sk+j)}$ mittels Satz 2.20

    $\alpha^{(sk+j)} := {g^{(sk+j)}}^\top G_k g^{(sk+j)}$ und Konvergenzkontrolle

    $\nu^{(sk+j)} := {g^{(sk+j)}}^\top G_k d^{(sk+j)}$

    $\gamma^{(sk+j)} := \alpha^{(sk+j)} / \nu^{(sk+j)}$

    Wenn $sk + j = 1$, dann

      Setze $\rho^{(sk+j)} := 1$

    Sonst

      Setze $\rho^{(sk+j)} := (1 - \frac{\gamma^{(sk+j)}\alpha^{(sk+j)}}{\gamma^{(sk+j-1)}\alpha^{(sk+j-1)}\rho^{(sk+j-1)}})^{-1}$

    Ende Wenn

    $x^{(sk+j+1)} := \rho^{(sk+j)}(x^{(sk+j)} + \gamma^{(sk+j)}q^{(sk+j)}) + (1 - \rho^{(sk+j)})x^{(sk+j-1)}$

    $u^{(sk+j)} := [Q_{k-1}, \underline{V_k}]d^{(sk+j)}$

    $q^{(sk+j+1)} := \rho^{(sk+j)}(q^{(sk+j)} + \gamma^{(sk+j)}u^{(sk+j)}) + (1 - \rho^{(sk+j)})q^{(sk+j-1)}$

    $y^{(sk+j)} := [Z_{k-1}, \underline{W_k}]d^{(sk+j)}$

    $z^{(sk+j+1)} := \rho^{(sk+j)}(z^{(sk+j)} + \gamma^{(sk+j)}y^{(sk+j)}) + (1 - \rho^{(sk+j)})z^{(sk+j-1)}$

  Ende Schleife

Ende Schleife

2.6 Kommunikationsvermeidung im BiCGStab-Verfahren

Eine elementare Schwäche des Verfahrens der konjugierten Gradienten ist seine Einschränkung auf symmetrische, positiv definite Matrizen A, da häufig auch unsymmetrische Systemmatrizen aus diskretisierten Differentialgleichungen entstehen können. Daher ist es unumgänglich, das Verfahren von dieser Restriktion zu befreien.

Eine entsprechende Erweiterung ist der bereits erwähnte BiCGStab-Algorithmus. Dieses Verfahren entstand aus der Idee nicht nur das System $Ax = b$, sondern simultan ebenso das transponierte System $A^\top x = b$ zu betrachten, um der fehlenden Symmetrieeigenschaft des Ausgangssystems entgegenzuwirken. Im Grunde löst man beide Systeme auf klassische Weise mit dem CG-Verfahren, koppelt aber geschickt, um über das transponierte System die Näherungslösung des eigentlichen Systems möglichst gut zu verbessern. Dabei geht die Struktur des CG-Verfahrens sogar derart in die des BiCGStab-Verfahrens über, dass im Falle einer symmetrischen und positiv definiten Ausgangsmatrix A ein Schritt der BiCGStab-Methode im Wesentlichen zwei klassischen CG-Iterationen entspricht.

Unsere Bemühungen aus Kapitel 2.4 zur Kommunikationsvermeidung innerhalb des CG-Verfahrens wollen wir nun auf das BiCGStab-Verfahren übertragen und orientieren uns dabei an einem technischen Bericht von Carson et al. [5]. Da die Notation in dieser Quelle wesentlich von jener aus der Doktorarbeit von Hoemmen [18] abweicht, an der wir uns in Kapitel 2.4 orientiert haben, werden wir manche Notationen ändern, um relative Konsistenz in beide Richtungen zu wahren. Dabei müssen natürlicherweise gewisse Kompromisse – etwa in der Indizierung – eingegangen werden.

Zu Übersichtszwecken beginnen wir mit der Angabe der klassischen Lehrbuchfassung des BiCGStab-Verfahrens ohne Verwendung eines Vorkonditionierers. Diese findet sich bis auf geringe Abweichungen in der Notation etwa im Lehrbuch von Meister [21] beziehungsweise ebenso in unserer Primärquelle von Carson et al. [5].

Algorithmus 2.23 (BiCGStab-Verfahren)**:**

Eingabe: Systemmatrix A, Anfangsnäherung $x^{(1)}$, rechte Seite b

1: Setze $r^{(1)} = p^{(1)} = b - Ax^{(1)}$
2: Schleife über $k = 1, 2, \ldots$ bis zur Konvergenz
3: $\quad \alpha^{(k)} := \frac{(r^{(0)}, r^{(k)})_2}{(r^{(0)}, Ap^{(k)})_2}$
4: $\quad s^{(k)} := r^{(k)} - \alpha^{(k)} Ap^{(k)}$
5: $\quad \omega^{(k)} := \frac{(s^{(k)}, As^{(k)})_2}{(As^{(k)}, As^{(k)})_2}$
6: $\quad x^{(k+1)} := x^{(k)} + \alpha^{(k)} p^{(k)} + \omega^{(k)} s^{(k)}$
7: $\quad r^{(k+1)} := s^{(k)} - \omega^{(k)} As^{(k)}$ und Konvergenzkontrolle
8: $\quad \beta^{(k)} := \frac{\alpha^{(k)} (r^{(0)}, r^{(k+1)})_2}{\omega^{(k)} (r^{(0)}, r^{(k)})_2}$
9: $\quad p^{(k+1)} := r^{(k+1)} + \beta^{(k)} (p^{(k)} - \omega^{(k)} Ap^{(k)})$
10: Ende Schleife

Wir beginnen mit der Erörterung eines *CA-BiCGStab-Verfahrens* und entledigen uns dazu zunächst des in Algorithmus 2.23 genutzten Hilfsvektors $s^{(k)}$ und schreiben die zentrale Zwei-Term-Rekursion als

$$\begin{aligned} r^{(k+1)} &= (I - \omega^{(k)} A)(r^{(k)} - \alpha^{(k)} Ap^{(k)}), \\ p^{(k+1)} &= r^{(k+1)} + \beta^{(k)} (I - \omega^{(k)} A) p^{(k)}. \end{aligned}$$

Entsprechend erhalten wir

$$\omega^{(k)} = \frac{r^{(k)\top} Ar^{(k)} - \alpha^{(k)} r^{(k)\top} A^2 p^{(k)} - \alpha^{(k)} p^{(k)\top} A^\top Ar^{(k)} + \alpha^{(k)2} p^{(k)\top} A^\top A^2 p^{(k)}}{r^{(k)\top} A^\top Ar^{(k)} - \alpha^{(k)} r^{(k)\top} A^\top A^2 p^{(k)} - \alpha^{(k)} p^{(k)\top} (A^\top)^2 Ar^{(k)} + \alpha^{(k)2} (A^\top)^2 p^{(k)}},$$

wobei wir die enthaltenen Skalarprodukte zudem zwecks einer kompakteren Notation wieder in ein Matrizenkalkül übertragen, sie also nicht in der Form $(\cdot, \cdot)_2$ schreiben. Die Aktualisierung der Näherungslösung ergibt sich nun ferner nach

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)} + \omega^{(k)} (r^{(k)} - \alpha^{(k)} Ap^{(k)}).$$

Das BiCGStab-Verfahren erhöht die Dimension des als Suchraum verwendeten Krylow-Unterraums in jeder Iteration um zwei. Dies geschieht implizit durch die zweimalige Anwendung von A innerhalb eines Iterationsschritts. Möchten wir nun s Verfahrensschritte in analoger Weise zu unseren das CG-Verfahren betreffenden Überlegungen von Kommunikation befreien, ist es notwendig, den Matrixpotenzen-Kernel zweimals innerhalb einer äußeren Iteration anzuwenden.

Des Weiteren sollen störende Skalarprodukte, die globale Kommunikation erfordern, wieder unter Zuhilfenahme einer Gram-Matrix sowie zweier Koeffizientenvektoren entfernt werden. Um in der Notation der Originalveröffentlichung von Carson et al. [5] zu bleiben, bezeichnen wir letztere nun mit $a_m^{(sk+j)}$ respektive $b_m^{(sk+j)}$, wobei k erneut der Laufindex der äußeren und j jener der inneren Schleife ist. Der Index m im Subskript stellt dabei eine Verbindung zur jeweiligen Potenz der Systemmatrix A her. Als sinnvollen notationellen Kompromiss erachten wir es dabei, erneut zu einer Zweifachindizierung der Form $sk+j$ überzugehen, welche die genannte Originalveröffentlichung in dieser Form nicht verwendet. Genauer seien die Koeffizientenvektoren $a_m^{(sk+j)}$ und $b_m^{(sk+j)}$ so gewählt, dass sie

$$\begin{aligned} [\underline{P}_k, \underline{R}_k]\, b_m^{(sk+j)} &= A^m r^{(sk+j)}, \\ [\underline{P}_k, \underline{R}_k]\, a_m^{(sk+j)} &= A^m p^{(sk+j)} \end{aligned}$$

erfüllen. Dabei seien $\underline{P}_k := [v_{sk}, \ldots, v_{sk+s}]$ beziehungsweise $\underline{R}_k := [w_{sk}, \ldots, w_{sk+s}]$ die durch den Matrixpotenzen-Kernel berechneten Teilbasen von jeweils Länge $s+1$. Der Leser beachte an dieser Stelle die im Vergleich zu Kapitel 2.4 um Eins verschobene Indizierung.

Wir benötigen überdies erneut eine Basiswechsel-Matrix $\underline{B}$. Diese genüge

$$\underline{P}_k\, \underline{B} = [p^{(sk)}, \ldots, A^{sk+2s} p^{(sk)}]$$

beziehungsweise

$$\underline{R}_k\, \underline{B} = [r^{(sk)}, \ldots, A^{sk+2s} r^{(sk)}],$$

überführe also die im Matrixpotenzen-Kernel gewählte Basis in die kanonische Monombasis. Wir gehen dabei davon aus, dass bei der Berechnung von $\underline{P}_k$ beziehungsweise $\underline{R}_k$ mittels Matrixpotenzen-Kernel die gleiche rekursive Beziehung – also die gleiche Basis – gewählt wird.

Nun benötigen wir noch einen Vektor g_k, den wir aus naheliegenden Gründen *Gram-Vektor* nennen, sowie erneut eine Gram-Matrix G_k. Diese definieren wir nach

$$\begin{aligned} g_k &:= {r^{(0)}}^\top [\underline{P}_k, \underline{R}_k], \\ G_k &:= ([\underline{P}_k, \underline{R}_k], [\underline{P}_k, \underline{R}_k])_2 \end{aligned}$$

und sind nunmehr in der Lage, den CA-BiCGStab-Algorithmus anzugeben:

Algorithmus 2.24 (CA-BiCGStab-Verfahren)**:**

Eingabe: Systemmatrix A, Anfangsnäherung $x^{(0)}$, rechte Seite b, Schleifenlänge s

1: Setze $r^{(0)} := p^{(0)} := b - Ax^{(0)}$

2: Schleife über $k = 0, 1, \ldots$ bis zur Konvergenz

3: $\quad v_{sk} := p^{(sk)}$

4: $\quad w_{sk} := r^{(sk)}$

5: $\quad$ Berechne mittels Matrixpotenzen-Kernel: $\underline{P}_k = [v_{sk}, \ldots, v_{sk+s}]$

6: $\quad$ Berechne mittels Matrixpotenzen-Kernel: $\underline{R}_k = [w_{sk}, \ldots, w_{sk+s}]$

7: $\quad$ Berechne $g_k := r^{(0)^\top}[\underline{P}_k, \underline{R}_k]$

8: $\quad$ Berechne $G_k := ([\underline{P}_k, \underline{R}_k], [\underline{P}_k, \underline{R}_k])_2$

9: $\quad$ Setze $[b_0^{(sk+0)}, \ldots, b_s^{(sk+0)}] := \begin{bmatrix} 0_{s+1,s+1} \\ \underline{B} \end{bmatrix}$

10: $\quad$ Setze $[a_0^{(sk+0)}, \ldots, a_s^{(sk+0)}] := \begin{bmatrix} \underline{B} \\ 0_{s+1,s+1} \end{bmatrix}$

11: $\quad$ Setze $e^{(0)} := \begin{bmatrix} 0_{2s+2,1} \\ 1 \end{bmatrix}$

12: $\quad$ Schleife über $j = 0, \ldots, s-1$

13: $\quad\quad \alpha^{(sk+j)} := \frac{(g_k, b_0^{(sk+j)})_2}{(g_k, a_1^{(sk+j)})_2}$

14: $\quad\quad \omega_1^{(sk+j)} := b_0^{(sk+j)^\top} G b_1^{(sk+j)} - \alpha^{(sk+j)} b_0^{(sk+j)^\top} G a_2^{(sk+j)} - \alpha^{(sk+j)} a_1^{(sk+j)^\top} G b_1^{(sk+j)} + \alpha^{(sk+j)^2} a_1^{(sk+j)^\top} G a_2^{(sk+j)}$

15: $\quad\quad \omega_2^{(sk+j)} := b_1^{(sk+j)^\top} G b_1^{(sk+j)} - \alpha^{(sk+j)} b_1^{(sk+j)^\top} G a_2^{(sk+j)} - \alpha^{(sk+j)} a_2^{(sk+j)^\top} G b_1^{(sk+j)} + \alpha^{(sk+j)^2} a_2^{(sk+j)^\top} G a_2^{(sk+j)}$

16: $\quad\quad \omega^{(sk+j)} := \frac{\omega_1^{(sk+j)}}{\omega_2^{(sk+j)}}$

17: $\quad\quad e^{(sk+j+1)} := e^{(sk+j)} + \alpha^{(sk+j)} \begin{bmatrix} a_0^{(sk+j)} \\ 0 \end{bmatrix} + \omega^{(sk+j)} \begin{bmatrix} b_0^{(sk+j)} \\ 0 \end{bmatrix} - \alpha^{(sk+j)} \omega^{(sk+j)} \begin{bmatrix} a_1^{(sk+j)} \\ 0 \end{bmatrix}$

18: $\quad\quad$ Schleife über $i = 0, \ldots, 2(s-j-1)$

19: $\quad\quad\quad b_i^{(sk+j+1)} := b_i^{(sk+j)} - \alpha^{(sk+j)} a_{i+1}^{(sk+j)} - \omega^{(sk+j)} b_{i+1}^{(sk+j)} + \alpha^{(sk+j)} \omega^{(sk+j)} a_{i+2}^{(sk+j)}$

20: $\quad\quad$ Ende Schleife über i

21: $\quad\quad \beta^{(sk+j)} := \frac{\alpha^{(sk+j)} (g_k, b_0^{(sk+j+1)})_2}{\omega^{(sk+j)} (g_k, b_0^{(sk+j)})_2}$

22: $\quad\quad$ Schleife über $i = 0, \ldots, 2(s-j-1)$

23: $a_i^{(sk+j+1)} := b_i^{(sk+j+1)} + \beta^{(sk+j)} a_i^{(sk+j)} - \beta^{(sk+j)} \omega^{(sk+j)} a_{i+1}^{(sk+j)}$

24: Ende Schleife über i

25: Ende Schleife über j

26: $r^{(sk+s)} := [\underline{P}_k, \underline{R}_k] b_0^{(sk+s)}$ und Konvergenzkontrolle

27: $p^{(sk+s)} := [\underline{P}_k, \underline{R}_k] a_0^{(sk+s)}$

28: $x^{(sk+s)} := [\underline{P}_k, \underline{R}_k, x^{(sk)}] e^{(sk+s)}$

29: Ende Schleife über k

Bei der Angabe von Algorithmus 2.24 haben wir uns, wie bereits betont, an einem technischen Forschungsbericht von Carson et al. [5] orientiert. Aus diesem Grund weicht seine Darstellung im Vergleich zu unseren in Kapitel 2.4 abgebildeten kommunikationsvermeidenden Algorithmen, deren Notation stark an jenen innerhalb der Dissertation von Hoemmen [18] verwendeten Schreibweisen angelehnt ist, etwas ab: In Algorithmus 2.24 werden etwa in der inneren über j indizierten Schleife nur Hilfsvektoren $e^{(sk+j)}$ iteriert statt wie bisher tatsächliche Näherungslösungen $x^{(sk+j)}$. Im CA-BiCGStab-Verfahren werden Näherungslösungen $x^{(sk+s)}$ erst nach jedem vollständigen Durchlauf der inneren Schleife mit Hilfe der Vektoren $e^{(sk+s)}$ erzeugt. Analog verhält es sich mit dem Residuum $r^{(sk+s)}$ sowie der aktuellen Suchrichtung $p^{(sk+s)}$. Auch diese Größen werden lediglich in jedem Durchlauf der äußeren Schleife bestimmt. Dies geschieht unter Zuhilfenahme der in jeweils einer zweiten inneren über i indizierten Schleife bestimmten Koeffizientenvektoren $b_0^{(sk+s)}$ respektive $a_0^{(sk+s)}$. Diese zweite innere Schleife ist im Grunde nicht neu, sie verbarg sich zuvor lediglich implizit in den von uns formulierten Sätzen 2.11 und 2.14 beziehungsweise im vorkonditionierten Fall den Sätzen 2.19 und 2.20 zur Bestimmung der Koeffizientenvektoren $d^{(sk+j)}$ sowie $g^{(sk+j)}$.

Zur Folge hat dieser Zusammenhang, dass die Konvergenzkontrolle – etwa mittels $||r^{(sk+s)}||_2$ – nun in der äußeren Schleife angesiedelt ist, was zu einer durch s teilbaren Gesamtiterationszahl führt. Da s typischerweise deutlich kleiner als die erwartete Gesamtiterationszahl gewählt wird, fällt der Laufzeitverlust, der mit den etwaig über die Konvergenz hinaus berechneten Iterierten einhergeht, somit nicht ins Gewicht.

Es kann gezeigt werden, dass bei einer blockweisen Parallelisierungsstrategie zu Algorithmus 2.24 die innere Schleife (Index j) wie auch schon in unseren Überlegungen und Erörterungen zum Verfahren der konjugierten Gradienten aus Kapitel

2.4 von jedem Prozessor vollständig kommunikationsfrei berechnet werden kann. Dabei muss man jedoch wieder die redundante Berechnung von Koeffizientenvektoren – hier $b_i^{(sk+j)}$ sowie $a_i^{(sk+j)}$ – in Kauf nehmen (vgl. Carson et al. [5]).

Für die praktische Konkurrenzfähigkeit von Algorithmus 2.24 ist seine zusätzliche Kopplung mit einem Vorkonditionierer M unumgänglich. Da wir uns in Kapitel 2.5 bereits mit der Erweiterung der kommunikationsvermeidenden Variante des Verfahrens der konjugierten Gradienten um eine Vorkonditionierungsroutine beschäftigt haben, verweisen wir an dieser Stelle jedoch auf folgende Literatur: Grobe Grundideen zur Anwendung eines Vorkonditionierers auf das oben beschriebene CA-BiCGStab-Verfahren finden sich etwa in Unterkapitel 3.5 des technischen Forschungsberichts von Carson et al. [5], während allgemeinere Überlegungen zur generellen Erweiterung des Matrixpotenzen-Kernels um Vorkonditionierungsprozesse Kapitel 2.2 der Dissertation von Hoemmen [18] entnommen werden können.

2.7 Wahl der Basis und Stabilität

Mathematisch sind die bisher vorgestellten Verfahren allesamt äquivalent zu ihrer jeweiligen Ausgangsformulierung. Da jedoch in der Praxis nur mit endlicher Genauigkeit gerechnet werden kann, entstehen möglicherweise sich fortpflanzende Fehler innerhalb der Ausführung des jeweiligen Verfahrens, so dass die Konvergenz oder die Korrektheit der resultierenden Lösung (sog. *Konsistenz*) gefährdet ist. Eine entsprechende Robustheit eines Algorithmus gegenüber solchen Fehlern in den Daten bezeichnen wir als *Stabilität*.

Da alle verwendeten Ausgangsverfahren für übliche Probleme bekanntermaßen hinreichend robust gegenüber durch Rechnung in endlicher Genauigkeit verursachte Fehler sind, reduziert sich obige Fragestellung darauf, ob unsere Umformulierungen zur Kommunikationsvermeidung einen Kompromiss bezüglich dieser Stabilitätseigenschaften fordern.

Hoemmen [18] stellt in seinen Untersuchungen fest, dass die Wahl der s-Schrittbasis im Matrixpotenzen-Kernel den wesentlichsten Beitrag für resultierende Stabilitätseigenschaften liefert. Er widmet sich genaueren Erläuterungen in Kapitel 7 seiner Dissertation.

Wir wollen an dieser Stelle die dort erörterten Sachverhalte lediglich zusammenfassen: Die Wahl der Basis im Matrixpotenzen-Kernel in kanonischer Form als

Monombasis, welche man durch Wahl der Koeffizienten nach $a_j = 1, b_j = 0, c_j = 0$ erhält, ist einfach und unkompliziert, resultiert aber in unmittelbaren Konsequenzen für die Verfahrensstabilität. So stellen etwa Chronopoulos und Gear [6] Testprobleme vor, die die Konvergenz ihres s-Schritt-CG-Algorithmus für $s > 5$ unter Verwendung der Monombasis beeinträchtigen. Die Wahl der die Basis erzeugenden Polynome $p_j(Z)$ im Matrixpotenzen-Kernel als *Newton-Polynome*, das heißt

$$\begin{aligned} p_0(Z) &= 1, \\ p_1(Z) &= (Z - \vartheta_1), \\ &\vdots \\ p_s(Z) &= \prod_{i=1}^{s} (Z - \vartheta_i) \end{aligned}$$

mit Interpolationsknoten $\vartheta_1, \ldots, \vartheta_s \in \mathbb{R}$, oder *Tschebyschow-Polynome*, das heißt

$$\begin{aligned} p_0(Z) &= 1, \\ p_1(Z) &= Z, \\ p_{i+1}(Z) &= 2Zp_i(Z) - p_{i-1}(Z) \end{aligned}$$

für $i = 1, \ldots, s-1$, kann hingegen im Vergleich zur Standardbasis zu deutlichen Stabilitätsverbesserungen führen. Wir müssen an dieser Stelle noch betonen, dass wir uns bei der Angabe der Tschebyschow-Polynome der Einfachheit halber auf eine unskalierte, nicht verschobene Variante beschränkt haben. Hoemmen [18] beschreibt die zur Aufstellung der Basis verwendeten Tschebyschow-Polynome in einer allgemeineren, skalierten sowie Shift-behafteten Weise.

In Kapitel 7 seiner Doktorarbeit geht er ferner auf den Umstand ein, dass die s-Schrittbasis unter endlich genauer Gleitkommaarithmetik dazu neigt, ihre natürliche lineare Unabhängigkeit zu verlieren und somit irregulär wird. Er definiert in diesem Zusammenhang den für weitere Untersuchungen wichtigen Begriff der *Basiskonditionszahl*.

Des Weiteren beschreibt er seine Beobachtungen bezüglich einer möglichen Skalierung der im Matrixpotenzen-Kernel verwendeten Basis. Diese kann einen sinnvollen Ausweg darstellen, falls bei Vergrößerung von s die Länge der Basisvektoren schnell zunimmt oder sich verringert und dadurch die Stabilität der resultierenden Methode oder ihre Konvergenzgeschwindigkeit negativ beeinflusst wird.

Kapitel 3

Asynchrone Verfahren

Bislang haben wir im Rahmen von Kapitel 2 klassische Lehrbuchmethoden derart umgeformt, dass sie in der Lage waren, Kommunikation zwischen einzelnen Computerkomponenten zu vermeiden, um modernen Hardwarekonzepten Rechnung zu tragen. Dies geschah jeweils unter der Prämisse, solche klassischen Verfahren mathematisch exakt weiterzuentwickeln: Das von uns im Kern herangezogene Beispielverfahren, das Verfahren der konjugierten Gradienten, lieferte etwa nach seiner Umformulierung zum CA-CG-Algorithmus (Algorithmus 2.16) – exakte Arithmetik vorausgesetzt – die gleiche Lösung $x^{(k+1)}$ nach jeweils k Iterationen wie sein unverändertes Pendant.

Wir wollen in diesem Kapitel nun einen anderen Ansatz verfolgen, versuchen jedoch gleichsam Kommunikation zu vermeiden, ja gänzlich zu unterlassen. Wir werden zwar weiterhin klassische Verfahren heranziehen, welche optimiert und moderner Hardware angepasst werden sollen, jedoch geschieht dies nicht mehr unter dem Vorsatz mathematischer Exaktheit: Tatsächlich werden wir sogar strukturelle mathematische Zusammenhänge in Verfahren ignorieren, indem einzelne Verfahrensschritte eines iterativen Verfahrens miteinander „vermischt“ werden. Der fundamentale Vorsatz der „Gleichzeitigkeit“ eines Iterationsschritts in Bezug auf alle Komponenten der Lösungsnäherung wird zu Gunsten größerer Auslastung aller Prozessoren innerhalb einer parallelen Implementierung aufgegeben: Mögliche Wartezeiten eines Prozessors vor Synchronisationspunkten werden umgangen, indem dieser statt wie in klassischen Parallelisierungsansätzen nicht auf das Eintreffen aktualisierter Größen von anderen Prozessoren wartet, sondern direkt mit

seinen Berechnungen fortfährt unter Verwendung der jüngsten verfügbaren Version benötigter Größen. Durch dieses Vorgehen verschmelzen die fundamentalen Iterationsschritte miteinander und bislang klare Grenzen zwischen jenen Schritten verschwinden. Aufgrund dieser fehlenden Synchronisation zwischen einzelnen Prozessoren sprechen wir hier gerade von *asynchronen Verfahren*. Da a priori nicht genau bestimmbar ist, welcher Prozessor genau wann welche Größe verwendet, spricht man im Extremfall auch von *chaotischen Verfahren*.

Wir wollen diese prosaische Motivation im Folgenden konkretisieren und den Begriff „asynchrones Verfahren“ genauer definieren.

3.1 Asynchrone Fixpunktiterationsverfahren

Wir orientieren uns für unsere nun angestrebte Definition eines asnychronen Verfahrens an einem Lehrbuch von Frommer [13, Kapitel 11]. Eine Veröffentlichung von Frommer und Szyld [14] erweist sich ebenfalls als nützlich.

3.1.1 Klassische Fixpunktiterationsverfahren

Wir möchten zunächst klassische Fixpunktiterationsverfahren erklären, um sie schließlich als Beispiel verwenden zu können. Solche Verfahren spielen in der Praxis aufgrund ihrer mäßigen Konvergenzeigenschaften keine Rolle, dienen in einführenden Vorlesungen sowie Lehrbüchern der Numerik aus didaktischen Gründen jedoch als Routinebeispiel, was unter anderem auf ihre Einfachheit hinsichtlich der Verfahrensstruktur sowie Konvergenztheorie zurückzuführen ist. Außerdem spielen sie eine wichtige Rolle als Komponenten in komplexeren Verfahren, wie beispielsweise als sogenannte *Glätter* in *Mehrgitter-Verfahren*. Für eine kurze Einführung verwenden wir ein entsprechendes Vorlesungsskriptum von Rannacher [23].

Unverändert ist das Lösen des regulären linearen Gleichungssystems

$$Ax = b$$

unser primäres Ziel. Für die Systemmatrix A definieren wir nun die Zerlegung

$$A = D + L + R, \tag{3.1}$$

wobei D die Diagonalmatrix darstellt, welche alle Diagonalelemente von A enthält, und L beziehungsweise R die verbleibenden Einträge von A unterhalb beziehungsweise oberhalb der Diagonale beinhalten. Alle anderen Einträge von D, L und R sind gleich null. Mit dieser Setzung nehmen wir nun die folgenden Umformungen bezüglich des Ausgangssystems vor:

$$\begin{aligned} & Ax = b \\ \Leftrightarrow \quad & (D + L + R)x = b \\ \Leftrightarrow \quad & Dx + (L + R)x = b \\ \Leftrightarrow \quad & x = D^{-1}(b - (L + R)x) \\ \Leftrightarrow \quad & x = -D^{-1}(L + R)x + D^{-1}b. \end{aligned}$$

Bei diesen Umformungen setzen wir implizit voraus, dass die Diagonalmatrix D tatsächlich regulär ist. Wir statten die letzte Zeile schließlich mit einem Iterationsindex $k \in \mathbb{N}_0$ aus und erhalten die Iterationsvorschrift des klassischen *Jacobi-Iterationsverfahrens*:

$$x^{(k+1)} := \underbrace{-D^{-1}(L + R)}_{=:J} x^{(k)} + D^{-1}b. \tag{3.2}$$

Die Matrix J ist die sogenannte *Iterationsmatrix* des Jacobi-Verfahrens.

Die nun folgende Definition erweitert das Jacobi-Verfahren auf eine allgemeine Verfahrensklasse:

Definition 3.1 (Fixpunktiterationsverfahren)**:** Ein Iterationsverfahren mit der Iterationsvorschrift

$$x^{(k+1)} := Bx^{(k)} + c \tag{3.3}$$

mit einer *Iterationsmatrix* B und einem Vektor c sowie einem Startvektor $x^{(0)}$ heißt *Fixpunktiterationsverfahren.*

Ist obiges Verfahren konvergent, etwa gegen einen Vektor x, so gilt offenbar

$$x = Bx + c,$$

das heißt x ist Fixpunkt der Abbildung $g : x \mapsto Bx + c$. Dieser Zusammenhang ist für diese Verfahrensklasse gerade namensgebend.

Die Erweiterung des Jacobi-Verfahrens zu dieser Klasse legitimiert sich nun durch die Setzung $B := J$ sowie $c := D^{-1}b$. Für die Setzungen $B := H_1 := -(D + L)^{-1}R$ und $c := (D + L)^{-1}b$ ergibt sich ferner das ebenso prominente *Gauß-Seidel-Verfahren*, welches als Lösungsverfahren für lineare Gleichungssysteme jedoch ebenfalls nicht von praktischer Relevanz ist.

Der Frage, unter welchen Bedingungen ein derart konstruiertes Verfahren nun tatsächlich gegen die Lösung x eines linearen Gleichungssystems $Ax = b$ konvergiert, soll nun nachgegangen werden. Als erstes notwendiges Kriterium muss die Iterationsmatrix B sowie der Vektor c selbstverständlich so gewählt werden, dass

$$x = Bx + c \Leftrightarrow Ax = b \tag{3.4}$$

gilt. Für ein weiteres schließlich hinreichendes Kriterium ist der Begriff des *Spektralradius einer Matrix* unverzichtbar. Diesen definieren wir für eine Matrix B als

$$\mathrm{spr}(B) := \max\{|\lambda| \ : \ \lambda \in \sigma(B)\},$$

wobei $\sigma(B)$ das *Spektrum der Matrix* B, also die Menge ihrer Eigenwerte, darstellt. Vergleiche dazu auch Definition A.1 in Anhang A.

Ein hinreichendes Konvergenzkriterium für das in Definition 3.1 vorgestellte Verfahren ergibt sich damit aus folgendem Satz:

Satz 3.2 (Hinreichendes Konvergenzkriterium)**:** Die durch die Iterationsvorschrift

$$x^{(k+1)} := Bx^{(k)} + c$$

erzeugten Iterierten $x^{(j)}$, $j \in \mathbb{N}$, konvergieren genau dann für jeden beliebigen Startvektor $x^{(0)}$ gegen die Lösung x der Gleichung $x = Bx + c$, wenn

$$\mathrm{spr}(B) < 1 \tag{3.5}$$

gilt.

Beweis: Rannacher [23, Seite 192] □

Der oben referenzierte Beweis beruht in wesentlichen Zügen auf der Verwendung des Banachschen Fixpunktsatzes. Dieser befindet sich in einer recht allgemeinen Form im Anhang dieser Arbeit als Satz B.1.

3.1.2 Asynchrone Fixpunktiterationsverfahren

Eine erste Beschreibung spezieller asynchroner Verfahren möchten wir im folgenden Abschnitt unternehmen. Zu Motivationszwecken wollen wir die im vorangegangenen Unterkapitel 3.1.1 eingeführten klassischen Fixpunktiterationsverfahren zu asynchronen Verfahren erweitern. Von einer „Erweiterung“ zu sprechen ist dabei tatsächlich angebracht, da bei unserer Definition eines asynchronen Verfahrens das Ausgangsverfahren immer auch als Spezialfall enthalten sein wird, nämlich dann, wenn die Synchronisationszeitpunkte künstlich auf jene des eigentlichen Verfahrens gesetzt werden. Praktisch verliert man dann natürlich die Berechtigung von einem asynchronen Verfahren zu sprechen.

Wir betonen an dieser Stelle erneut, dass auch die Umwandlung von Fixpunktiterationsverfahren zur Lösung von linearen Gleichungssystemen in asynchrone Verfahren in der numerischen Praxis nicht konkurrenzfähig ist. Sie hat an dieser Stelle allein einen einführenden sowie didaktischen Charakter.

Im zu Beginn erwähnten Lehrbuch von Frommer [13] aus dem Jahr 1990 findet sich bereits eine Definition eines asynchronen Verfahrens – diese sogar im Kontext von Mehrprozessorrechnern mit geteiltem Speicher. Da diese Definition noch sehr speziell auf das Jacobi-Verfahren zugeschnitten ist, möchten wir die Grundnotationen im Wesentlichen aufgreifen, die Definition aber auf beliebige Fixpunktiterationsverfahren im Sinne von Definition 3.1 erweitern.

Wir betrachten erneut die Aktualisierungsvorschrift für die Lösungsnäherung $x^{(k)}$ aus dem vorangegangenen Unterkapitel. Mit Hilfe der von uns definierten Funktion $g : x \mapsto Bx + c$ gilt für diese sofort

$$x^{(k+1)} := g(x^{(k)}) = Bx^{(k)} + c. \tag{3.6}$$

Wir gehen weiterhin davon aus, dass es sich bei der Systemmatrix A um eine reguläre $(n \times n)$-Matrix handelt. Entsprechend gilt also auch $x \in \mathbb{R}^n$ und somit auch $g : \mathbb{R}^n \to \mathbb{R}^n$.

Es folgt die Angabe der Definition eines asynchronen Verfahrens.

Definition 3.3 (Asynchrones Verfahren)**:** Für $k \in \mathbb{N}$ seien Indexmengen $I_k \subset \{1, \ldots, n\}$ sowie n-Tupel $(s_1(k), \ldots s_n(k)) \in \mathbb{N}_0^n$ gegeben. Für diese Größen gelte

(a) $s_j(k) \leq k$ für $j = 1, \ldots, n$ und $k \in \mathbb{N}$,

(b) $\lim_{k\to\infty} s_j(k) = \infty$ für $j = 1, \ldots, n$ und

(c) für jedes $j \in \{1, \ldots, n\}$ enthalte die Menge $\{k \ : \ j \in I_k\}$ unendlich viele Elemente.

Dann heißt das Verfahren, welches ausgehend von einem Startvektor $x^{(0)} \in \mathbb{R}^n$ die Iterierten $x^{(k)}$ über die Vorschrift

$$x_i^{(k+1)} = \begin{cases} x_i^{(k)} & \text{für } i \notin I_k \\ \left[g\left(\left[x_1^{(s_1(k))}, \ldots, x_n^{(s_n(k))} \right]^\top \right) \right](i) & \text{für } i \in I_k \end{cases} \tag{3.7}$$

für $k \in \mathbb{N}$ berechnet, ein *zu* $(I_k, (s_1(k), \ldots, s_n(k)))$ *gehöriges asynchrones Iterationsverfahren*. Das Mengensystem I_k nennen wir dabei die *Strategie des Verfahrens* und $d_j(k) := k - s_j(k)$ die *Verzögerung der Komponente* j.

Definition 3.3 können wir uns wie folgt vorstellen: Der Iterationsindex des Ausgangsverfahrens k indiziert nun eine Reihe von Zeitpunkten $t_1 < \ldots < t_k$. Zu jedem Zeitpunkt t_{k+1} werden alle Komponenten der Lösungsnäherung $x_j^{(k+1)}$ mit $j \in I_k$ aktualisiert. Dies kann mit mehreren Prozessoren parallel erfolgen, etwa indem wieder alle Zeilen des Systems blockweise auf alle zur Verfügung stehenden Recheneinheiten verteilt werden. Bei dieser Aktualisierung werden die zu diesem Zeitpunkt aktuellsten vorangegangenen Komponenten der Lösungsnäherung verwendet, nämlich $(x_1^{(s_1(k))}), \ldots, (x_n^{(s_n(k))})$. Befindet sich eine Komponente nicht in I_k, so wird sie zum Zeitpunkt t_{k+1} auch nicht verändert.

Das n-Tupel $(s_1(k), \ldots s_n(k)) \in \mathbb{N}_0^n$ gibt also für jede Iteration k und jede Komponente der Lösungsnäherung des asynchronen Verfahrens an, wie oft diese bereits verändert wurde. Forderung (a) aus Definition 3.3 bedeutet also, dass nach k Zeitpunkten $t_1, \ldots, t_k$ eine Komponente höchstens k-mal verändert worden sein kann. Forderung (b) hingegen soll sicherstellen, dass kein Teil der Lösung vollständig stagniert: Nach theoretisch unendlich langer Laufzeit des Verfahrens soll auch jede Komponente der Lösungsnäherung unendlich oft aktualisiert worden sein. Dies stellt auch sicher, dass im Laufe des Verfahrens nicht mehr auf Werte zurückgegriffen wird, die zu alt sind. Die letzte Forderung (c) ist schließlich sehr verwandt zu (b): Damit Forderung (b) erfüllbar ist, muss jede Komponente auch in unendlich vielen Mengen I_k auftreten, also in unendlich vielen Verfahrensschritten erneuert

werden. Wie wir sehen werden, ist gerade (b) von zentraler Bedeutung für die Konvergenz asynchroner Verfahren dieser Art.

Wie wir bereits zu Beginn erwähnt haben, ergibt sich nun tatsächlich das Ausgangsverfahren als spezielle Strategie unseres asynchronen Verfahrens: Setzen wir $I_k := \{1, \ldots, n\}$ und $s_j(k) = k$ für alle $j = 1, \ldots, n$ und alle $k \in \mathbb{N}$, so wird jede Komponente zu jedem Zeitpunkt t_{k+1} genau einmal aktualisiert und somit in jedem Schritt auf die vollständig aktualisierte Lösungsnäherung zurückgegriffen.

Ähnlich wie wir bereits im vorangegangenen Kapitel in Form von Satz 3.2 ein hinreichendes Kriterium für die Konvergenz eines klassischen Fixpunktiterationsverfahrens bereitgestellt haben, wollen wir dies nun auch für das asynchrone Pendant schaffen. Dabei verwenden wir wieder das Lehrbuch von Frommer [13], welcher ein solches hinreichendes Kriterium speziell für das asynchrone Jacobi-Verfahren beweist. Dieses Resultat lässt sich aber auf beliebige asynchrone Verfahren, die aus Fixpunktiterationsverfahren entstanden sind, ohne Mühe erweitern. Das Kriterium ist in folgendem Satz zusammengefasst:

Satz 3.4 (Hinreichendes Konvergenzkriterium)**:** Die durch die Iterationsvorschrift (3.7) erzeugten Iterierten $x^{(j)}$, $j \in \mathbb{N}$, konvergieren für jeden beliebigen Startvektor $x^{(0)}$ gegen die Lösung x der Gleichung $x = Bx + c$, wenn

$$\operatorname{spr}(|B|) < 1 \tag{3.8}$$

gilt. Dabei versteht sich $|\cdot|$ als komponentenweise Betragsfunktion.

Beweis: Nach Voraussetzung gilt die Gleichung

$$x = Bx + c$$

beziehungsweise komponentenweise

$$x_i = B(i,:)x + c_i. \tag{3.9}$$

Nun sei $\varepsilon > 0$ so gewählt, dass

$$\rho_\varepsilon := \operatorname{spr}(B_\varepsilon) < 1 \tag{3.10}$$

mit der positiven Matrix

$$B_\varepsilon := |B| + \varepsilon \begin{bmatrix} 1 & \ldots & 1 \\ \vdots & & \vdots \\ 1 & \ldots & 1 \end{bmatrix} \tag{3.11}$$

gilt. Nach dem Satz von Perron-Frobenius (Satz B.2) existiert nun ein Eigenvektor u von B_ε zu ρ_ε mit $u \geq 0$ sowie $u \neq 0$. Insbesondere gilt also die Eigenwertgleichung

$$B_\varepsilon u = \rho_\varepsilon u. \tag{3.12}$$

Damit folgt

$$|B|u < |B|u + \varepsilon \begin{bmatrix} 1 & \dots & 1 \\ \vdots & & \vdots \\ 1 & \dots & 1 \end{bmatrix} u = |B_\varepsilon| u = \rho_\varepsilon u$$

beziehungsweise wieder in komponentenweiser Betrachtung

$$|B(i,:)|u < \rho_\varepsilon u_i \tag{3.13}$$

für $i = 1, \dots, n$. Wir definieren nun weiterhin ein $\alpha > 0$, so dass

$$|x^{(0)} - x| \leq \alpha u \tag{3.14}$$

gilt. Damit streben wir das Ziel der Konstruktion einer Folge $k_p : \mathbb{N}_0 \to \mathbb{N}$ mit $p \mapsto k_p$ an, die

$$|x^{(k+1)} - x| \leq \alpha \rho_\varepsilon^p u$$

für alle $k \geq k_p$ beziehungsweise komponentenweise

$$|x_i^{(k+1)} - x_i| \leq \alpha \rho_\varepsilon^p u_i \tag{3.15}$$

für $i = 1, \dots, n$ und alle $k \geq k_p$ erfüllt. Da $\rho_\varepsilon < 1$ gilt, folgt dann die Behauptung des Satzes.

Wir wollen zunächst die Gültigkeit von (3.15) für $p = 0$ und $k_0 = 0$ zeigen. Diese stellt sich als

$$|x_i^{(k+1)} - x_i| \leq \alpha u_i \tag{3.16}$$

dar für $i = 1, \dots, n$ und alle $k \geq 0$. Für $k = 0$ gilt (3.16) bereits nach Wahl von α. Wir wollen dies induktiv fortsetzen und nehmen dazu an, dass (3.16) bereits bis einschließlich zu einem bestimmten Index k_0 gilt. Für den Fall, dass $i \notin I_{k_0}$ gilt, folgt sofort

$$|x_i^{(k_0+1)} - x_i| = |x_i^{(k_0)} - x_i| \leq \alpha u_i.$$

Falls hingegen $i \in I_{k_0}$ gilt, erhalten wir

$$\begin{aligned} |x_i^{(k_0+1)} - x_i| &= |x_i^{(k_0+1)} - (B(i,:)x + c_i)| \\ &= \left| B(i,:) \left(\left[x_1^{(s_1(k_0))}, \ldots, x_n^{(s_n(k_0))} \right]^\top - x \right) \right| \\ &\leq |B(i,:)| \left| \left[x_1^{(s_1(k_0))}, \ldots, x_n^{(s_n(k_0))} \right]^\top - x \right|. \end{aligned}$$

Da nach Definition 3.3 (a) $s_i(k_0) \leq k_0$ gilt, folgt mit $|x_i^{(s_i(k_0))} - x_i| \leq \alpha u_i$ für $i = 1, \ldots, n$

$$|x_i^{(k_0+1)} - x_i| \leq |B(i,:)|\alpha u \leq \alpha \rho_\varepsilon u_i \leq \alpha u_i$$

und damit schließlich

$$|x_i^{(k+1)} - x_i| \leq \alpha u_i \tag{3.17}$$

für alle $k \geq 0$. Dies schließt den induktiven Beweis ab.

Wir nehmen nun an, dass (3.15) bereits für alle $k \geq k_p$ und für alle $p \leq p_0 \in \mathbb{N}$ mit einem gewissen Index p_0 gilt. Damit konstruieren wir ein weiteres Folgenglied k_{p_0+1} derart, dass (3.15) auch für alle $p \leq p_0 + 1$ gilt. Dazu definieren wir die Zahl

$$r := \max\{l_i \ : \ i \in \{1, \ldots, n\}\} \tag{3.18}$$

mit

$$l_i := \min\{l \in \mathbb{N} \ : \ s_i(k) \geq k_{p_0} \ \forall k \geq l\} \tag{3.19}$$

für $i = 1, \ldots, n$. Da nach Definition 3.3 (b) die Zahlen l_i existieren, existiert auch r. Für $k \geq r$ wird bei der Berechnung von $x^{(k+1)}$ nur auf Komponenten der Vektoren $x^{(j)}$ zurückgegriffen mit $j \geq k_{p_0}$. Ferner gilt

$$r > k_{p_0}, \tag{3.20}$$

denn nach Definition 3.3 (a) ist $l_i > k_p$ für $i = 1, \ldots, n$. Wir konstruieren nun das nächste Folgenglied k_{p_0+1} nach der Vorschrift

$$k_{p_0+1} := \min \left\{ k \ : \ k \geq r \wedge \bigcup_{j=r}^{k} I_j = \{1, \ldots, n\} \right\}. \tag{3.21}$$

Die Existenz von k_{p_0+1} ist dabei durch Definition 3.3 (c) gesichert. Für $j \geq k_{p_0+1}$ wurden im Vergleich zu $x^{(r)}$ nun alle Komponenten von $x^{(j)}$ mindestens einmal aktualisiert. Wir zeigen nun noch, dass (3.15) auch für $p = p_0+1$ und alle $k \geq k_{p_0+1}$ gilt: Für $k \geq k_{p_0+1}$ und $i \in \{1, \ldots, n\}$ definieren wir dazu

$$m_i := \max\{l \ : \ i \in I_l, \ l \leq k\}. \tag{3.22}$$

Anschaulich bedeutet dies, dass die i-te Komponente von $x^{(k)}$ letztmalig bei der m_i-ten Iteration erneuert wurde. Nach Konstruktion von k_{p_0+1} gilt $m_i \geq r$ für $i = 1, \ldots, n$. Insbesondere bedeutet dies nach Definition von l_i, dass

$$s_j(m_i) > k_{p_0} \tag{3.23}$$

für $i, j = 1, \ldots, n$ gilt. Für $k \geq k_{p_0+1}$ erhalten wir

$$\begin{aligned} |x_i^{(k_0+1)} - x_i| &= |x_i^{(m_i+1)} - x_i| \\ &= |x_i^{(m_i+1)} - (B(i,:)x + c_i)| \\ &= \left| B(i,:) \left(\left[x_1^{(s_1(m_i))}, \ldots, x_n^{(s_n(m_i))} \right]^\top - x \right) \right| \\ &\leq |B(i,:)| \left| \left[x_1^{(s_1(m_i))}, \ldots, x_n^{(s_n(m_i))} \right]^\top - x \right|. \end{aligned} \tag{3.24}$$

Unter Ausnutzung von (3.23) folgt aus der Induktionsannahme nun

$$|x_j^{(s_j(m_i))} - x_j| \leq \alpha \rho_\varepsilon^{p_0} u_j$$

für $j = 1, \ldots, n$ und schließlich mit Ungleichung (3.24)

$$|x_i^{(k+1)} - x_i| \leq |B(i,:)| \alpha \rho_\varepsilon^{p_0} u$$

für $i = 1, \ldots, n$. Wenden wir nun (3.13) an, erhalten wir

$$|x_i^{(k+1)} - x_i| \leq \alpha \rho_\varepsilon^{p_0+1} u_i$$

für $i = 1, \ldots, n$. Damit ist der Beweis zu Satz 3.4 abgeschlossen. □

Man beachte, dass mit obigem Satz keine Äquivalenz wie in Satz 3.2, sondern lediglich eine Folgerung gegeben ist. Frommer [13] stellt außerdem eine gewisse Umkehrung von Satz 3.4 in seinem Lehrbuch bereit, beschränkt sich hier jedoch

erneut auf das Jacobi-Verfahren. So liefert der Satz im Wesentlichen die Aussage, dass für den Fall

$$\rho(|J|) \geq 1$$

mit der Iterationsmatrix $J = -D^{-1}(L + R)$ des Jacobi-Verfahrens dieses nicht für alle Strategien I_k und Verzögerungen $d_j(k)$ konvergiert. Bei Interesse verweisen wir auf Satz 11.2.5 im entsprechenden Lehrbuch.

3.2 Block-Jacobi-Vorkonditionierung

In Unterkapitel 2.5.1 dieser Arbeit haben wir bereits den Begriff eines „Vorkonditionierers“ eingeführt.

In diesem Kapitel wollen wir einen speziellen Vorkonditionierer erklären. Wir greifen dazu eine Definition aus dem Lehrbuch von Barrett et al. [1] auf, die wir jedoch wesentlich verallgemeinern, um unseren späteren Anwendungen gerecht zu werden. Wir gehen dabei wie üblich davon aus, dass ein lineares Gleichungssystem der Form

$$Ax = b$$

mit einer Matrix $A \in \mathbb{R}^{n \times n}$ sowie einem Vektor der rechten Seite $b \in \mathbb{R}^n$ und einem Vektor von Unbekannten $x \in \mathbb{R}^n$ zu lösen beziehungsweise zu diesem Zweck vorzukonditionieren ist.

Definition 3.5 (Block-Jacobi-Vorkonditionierer)**:** Zur Menge von Indizes $S = \{1, \ldots, n\}$ sei eine Zerlegung in paarweise disjunkte Teilmengen S_j, $j = 1, \ldots, p$, $p \leq n$, gegeben. Es gelte also $S = \bigcup_{j=1}^{p} S_j$, $S_i \cap S_j = \emptyset$ für $i \neq j$ sowie $S_j \neq \emptyset$, $i, j = 1, \ldots, p$. Wir fordern ferner, dass $k < l$ für alle $k \in S_i$ und $l \in S_j$ und alle $i < j$ gilt. Schließlich seien Teilmengen $U_i \subset S_i \times S_i$ gegeben. Dann sprechen wir bei der zweidimensionalen Indexmenge

$$U := \bigcup_{j=1}^{p} U_j \tag{3.25}$$

von einer *Übernahmestrategie*. Die Matrix $M \in \mathbb{R}^{n \times n}$ mit

$$M(i,j) := \begin{cases} A(i,j) & \text{falls } \exists (i,j) \in U \\ 0 & \text{sonst} \end{cases} \tag{3.26}$$

sowie $i, j = 1, \ldots, n$ definiert den *Block-Jacobi-Vorkonditionierer zur Blockstruktur* $(S_1, \ldots, S_n)$ *und zur Übernahmestrategie* U *von* A.

M ist somit eine Matrix von blockweiser Diagonalgestalt. Für $p = 1$, $S_1 = S = \{1, \ldots, n\}$ und $U = U_1 = S_1 \times S_1$ entsteht gerade der pathologische Fall $M := A$. Ein Vorkonditionerungsschritt entspräche also formal dem Lösen des eigentlichen linearen Gleichungssystems, was keinen Gewinn in Bezug auf die zu bewältigende Problemkomplexität darstellt. Als umgekehrter Entartungsfall ergibt sich bei einer Wahl von $p = n$ die einzige legitime Partition $S_j = \{j\}$ für $j = 1, \ldots, n$. In diesem Fall entspricht M gerade dem klassischen *Jacobi-Vorkonditionierer*

$$M := \operatorname{diag}(A(1,1), \ldots, A(n,n)),$$

falls auch $U_j = S_j \times S_j$ für $j = 1, \ldots, n$ gewählt wurde. Dieser trägt seinen Namen aufgrund seiner Verbindung zum Jacobi-Fixpunktiterationsverfahren aus Unterkapitel 3.1.1. Die hier essentielle Matrix D aus der additiven Zerlegung $D + L + R$ von A gleicht nun gerade der Matrix M des Jacobi-Vorkonditionierers. Aus der Analogie zur Struktur dieser Matrix des Jacobi-Vorkonditionierers leitet sich nun gerade der Name des Block-Jacobi-Vorkonditionierers aus Definition 3.5 ab.

Der wesentliche Unterschied zwischen unserer Definition des Block-Jacobi-Vorkonditionierers und jener aus dem Lehrbuch von Barrett et al. [1] findet sich in der Möglichkeit, die Struktur der einzelnen Blöcke in unserer Definition explizit vorgeben zu können. Die Ursprungsdefinition ist hingegen auf die vollständige Übernahme aller Einträge innerhalb eines Blocks ausgelegt.

3.3 FGMRES(m)-Verfahren

In Unterkapitel 2.1 haben wir das GMRES-Verfahren bereits als einen populären Vertreter der Klasse der Krylow-Unterraum-Methoden erwähnt. Dieses Verfahren hat im Vergleich zur CG-Methode zunächst den Vorteil, dass die Systemmatrix A nicht notwendigerweise symmetrisch sowie positiv definit sein muss. Es wurde 1986 von Saad und Schultz [25] entwickelt.

Einen elementaren Nachteil des GMRES-Verfahrens – etwa im Vergleich zum CG- oder BiCGStab-Verfahren – bildet der Umstand, dass sich in jedem Iterationsschritt die Anzahl der zu speichernden Basisvektoren sowie der zu berechnenden

Skalarprodukte erhöht. Da diese Basisvektoren in der Regel vollbesetzt sind, resultiert hieraus ein extrem hoher Speicherplatzbedarf. Darüber hinaus muss aus ihnen eine Orthogonalbasis bestimmt werden, was natürlicherweise im Aufwand mit der Erhöhung der Länge der Basis zunimmt. Um dieser Problematik entgegenzuwirken, nimmt man daher in der Regel nach einer festen Zahl von Iterationen einen *Restart* vor. Dabei gehen zwar die bereits konstruierten Suchrichtungen verloren und das Verfahren startet mit der aktuellen Näherungslösung neu, jedoch wird auch der notwendige Speicherplatz reduziert. Man spricht hier im Allgemeinen vom *GMRES(m)-Verfahren*, wobei m die Anzahl von Iterationen bezeichnet, nach denen das Verfahren neugestartet werden soll.

Leider ist dieses Vorgehen auch mit einem Verlust hinsichtlich der Konvergenz des Verfahrens verbunden. Während das nicht restartbehaftete GMRES-Verfahren theoretisch als direktes Verfahren betrachtet werden kann, da es bei exakter Arithmetik nach spätestens n Iterationen zur gesuchten Lösung gelangt, ist diese Anschauung bei einem regelmäßigen Restart nicht mehr rechtfertigbar. Es liegt zwar bei der Unternahme solcher Neustarts immer noch ein monotenes Abfallen des Residuums vor, jedoch besteht die Möglichkeit, dass das Residuum stagniert. Es wurde aber für eine Reihe von Spezialfällen eine untere Schranke für die Restarthäufigkeit m nachgeweisen, für welche die Konvergenz des GMRES(m)-Verfahren gesichert ist. Meister [21] attestiert der Methode in Abschnitt 4.3.2.4 seines Lehrbuchs etwa für $m \geq 2$ Konvergenzverhalten, falls A regulär und symmetrisch ist.

Aus ebendiesem Lehrbuch stammt auch der folgende Satz, welcher die Residuumsreduktion nach m Verfahrensschritten des herkömmlichen, nicht restartbehafteten GMRES-Verfahrens beschreibt.

Satz 3.6: Sei $A \in \mathbb{R}^{n \times n}$ positiv definit und symmetrisch. Zudem sei $r^{(m)}$ der im GMRES-Verfahren ermittelte m-te Residuenvektor. Dann konvergiert das GMRES-Verfahren und es gilt

$$||r^{(m)}||_2 \leq \sqrt{\frac{\text{cond}^2(A) - 1}{cond^2(A)}}^m \; ||r^{(0)}||_2. \tag{3.27}$$

Beweis: Meister [21, Satz 4.78, Korollar 4.79] □

Da das GMRES(m)-Verfahren innerhalb eines Restartzyklus äquivalent zur Standard-Variante des Verfahrens ist, gilt Satz 3.6 somit auch für die restartbe-

haftete Methode, jedoch bezieht sich $r^{(0)}$ dann immer auf das letzte Residuum vor dem jüngsten Restart.

Da in der Praxis die Verwendung eines solchen Restarts gängiger Standard ist und wir überdies ebenso immer einen solchen vornehmen werden, weisen wir darauf hin, dass wir im Folgenden den Restartparameter m nicht immer explizit in der entsprechenden Verfahrensbezeichnung angeben werden.

In Kapitel 9.3 des Lehrbuchs von Saad [24] gibt dieser links- sowie rechtsvorkonditionierte Varianten des GMRES-Verfahrens an.

Wir verfolgen im weiteren Verlauf den Gedanken einer asynchronen Vorkonditionerung, also einer Vorkonditionerung die sich im Lösungsverfahren nicht notwendigerweise gleichzeitig auf alle Komponenten eines vorzukonditionierenden Vektors auswirkt. Da speziell nicht vorherzusagen ist, wann genau welche Teile einer Größe von der Vorkonditionierung betroffen sind, müssen wir dabei davon ausgehen, dass – formal betrachtet – die Matrix M^{-1} über die einzelnen Iterationsschritte des Verfahrens nicht konstant ist. Wir nehmen daher an, dass in jedem Schritt j eine möglicherweise unterschiedliche Matrix M_j als Vorkonditionierer verwendet wird.

Die üblichen Formulierungen der vorkonditionierten CG-, BiCGStab- sowie GMRES-Varianten sind unter diesem Umstand nicht notwendig konvergent, bedeutet eine sich ständig verändernde Matrix M_j doch, dass auch das zu lösende lineare Gleichungssystem in jedem Iterationsschritt ein zwar äquivalentes, jedoch anderes ist (vgl. Saad [24, Seite 273]).

Abhilfe schafft hier das sogenannte *FGMRES(m)-Verfahren* (engl. *flexible generalized minimal residual method*). Dieses ist auch in der Lage, *flexible Vorkonditionierer* M_j zu verwenden. Entgegen unserer vorkonditionierten Verfahren aus Kapitel 2 verwendet der nachstehende Algorithmus des FGMRES-Verfahrens dabei eine Rechtsvorkonditionierungstechnik. Er entstammt ebenfalls dem Lehrbuch von Saad [24], enthält dort aber einen Fehler: Hier befindet sich die Definition der Matrizen Z_m sowie H_m innerhalb der über j indizierten inneren Schleife. In Algorithmus 3.7 haben wir dies entsprechend korrigiert.

Algorithmus 3.7 (FGMRES(m)-Verfahren)**:**

Eingabe: Systemmatrix A, Anfangsnäherung $x^{(0)}$, rechte Seite b, Schleifenlänge m, Vorkonditionierer M_j, $j = 1, \ldots, m$

1: Setze $r^{(0)} := b - Ax^{(0)}$
2: Setze $\beta := (r^{(0)}, r^{(0)})_2 = ||r^{(0)}||_2^2$
3: Setze $v^{(0)} := \frac{r^{(0)}}{\beta}$
4: Schleife über $j = 1, \ldots, m$
5: $\quad z^{(j)} := M_j^{-1} v^{(j)}$
6: $\quad w := Az^{(j)}$
7: $\quad$ Schleife über $i = 1, \ldots, j$
8: $\quad\quad h_{i,j} := (w, v^{(i)})_2$
9: $\quad\quad w := w - h_{i,j} v^{(i)}$
10: $\quad$ Ende Schleife
11: $\quad h_{j+1,j} := ||w||_2$
12: $\quad v^{(j+1)} := \frac{w}{h_{j+1,j}}$
13: Ende Schleife
14: Setze $Z_m := [z^{(1)}, \ldots, z^{(m)}]$
15: Setze $H_m := (h_{i,j})_{1 \leq i \leq j+1, 1 \leq j \leq m}$
16: Setze $y^{(m)} := \text{argmin}_{y \in \mathbb{R}^m} ||\beta e_1 - H_m y||_2$
17: Setze $x^{(m)} := x^{(0)} + Z_m y^{(m)}$
18: Konvergenzkontrolle, falls nicht erfüllt setze $x^{(0)} := x^{(m)}$ und springe zum Anfang

Für den Fall einer konstanten Vorkonditionierung $M_j = M$ ist der obige Algorithmus mathematisch äquivalent zu seinem Ausgangsalgorithmus (Algorithmus 9.5 im Lehrbuch von Saad [24]), aus welchem er abgeleitet wurde. Beide Algorithmen unterscheiden sich im Grunde nur durch das Erlauben unterschiedlicher Vorkonditionierer in jedem Schritt j sowie eine daraus resultierende abweichende Berechnung der Näherungslösung $x^{(m)}$. Ersetzt man in Algorithmus 3.7 die Zeilen 5 sowie 6 durch $w := AM^{-1}v^{(j)}$ und berechnet die Näherungslösung in Zeile 17 nach $x^{(m)} := x^{(0)} M^{-1} V_m y^{(m)}$, wobei die Matrix V_m analog zu Z_m für die Vektoren $v^{(j)}$ mit $j = 1, \ldots, m$ definiert sei, ergibt sich wieder der Ausgangsalgorithmus.

Bei der Matrix H_m handelt es sich um eine $(m + 1 \times m)$-Matrix. Entfernen wir ihre letzte Zeile, betrachten also $H_m(1 : m, :)$, erhalten wir eine obere Hessenberg-

Matrix (vgl. Definition A.5). Wir nehmen also an, dass alle Einträge von H_m, die in Algorithmus 3.7 nicht explizit gesetzt werden, gleich null sind, die Matrix also zu Beginn des Algorithmus mit $H_m := 0_{m+1,m}$ initialisiert wird.

Wir unterdrücken ferner in einigen Größen, wie etwa dem Vektor w, die Angabe des Iterationsindex, um die Notation möglichst einfach und in Konsistenz zur Literaturvorlage zu halten. Aus diesem Grund indizieren wir die Anfangsnäherung $x^{(0)}$ außerdem abweichend vom bisherigen Konsenz innerhalb dieser Arbeit mit der Null und formulieren die äußere Schleife auf „go to"-artige Weise in der letzten Zeile des Algorithmus.

Das Minimierungsproblem (*kleinste Fehlerquadrate*)

$$y^{(m)} := \text{argmin}_{y \in \mathbb{R}^m} ||\beta e_1 - H_m y||_2 \tag{3.28}$$

in Zeile 16 des FGMRES-Verfahrens ist mit wenig Aufwand zu lösen, da $m \in \mathbb{N}$ typischerweise klein gewählt wird. Weist die Matrix H_m überdies vollen Rang auf, gilt also $\text{rank}(H_m) = m$, so lässt sich die Lösung $y^{(m)}$ von (3.28) als die eindeutige Lösung des linearen Gleichungssystems

$$H_m^\top H_m y^{(m)} = H_m^\top \beta e_1 \tag{3.29}$$

beschreiben. In diesem Fall wäre also in jedem äußeren Iterationsschritt ein quadratisches System der Dimension m zu lösen. Einen allgemeineren Ansatz liefert die Moore-Penrose-Inverse $p(H_m)$ von H_m (auch Pseudoinverse, vgl. Definition A.6). Mit ihr ergibt sich die Lösung des obigen Minimierungsproblems direkt als

$$y^{(m)} := \beta p(H_m) e_1. \tag{3.30}$$

Für eine umfassende Abhandlung über Pseudoinversen verweisen wir bei Interesse auf das Buch von Ben-Israel und Greville [3], für genaue Unterschiede zwischen dem eigentlichen GMRES-Verfahren und der obigen FGMRES-Variante, Konvergenzaussagen sowie eine präzise Herleitung verweisen wir abschließend erneut auf die genannte Literatur von Saad [24].

3.4 Ein asynchrones Verfahren

Wir wollen an dieser Stelle nun exemplarisch ein spezielles asynchrones Verfahren erzeugen, dass mit seinen Eigenschaften versucht, modernen Hardwarearchitektu-

ren gerecht zu werden. Beispielhaft haben wir in Unterkapitel 3.1 bereits Überlegungen zu asnychronen Fixpunktiterationsverfahren gesehen, wie sie Frommer [13] bereits 1990 angestellt hat. Leider sind diese Verfahren – wie erläutert – heute nur noch von didaktischem Charakter.

Des Weiteren haben wir in Unterkapitel 3.2 den Block-Jacobi-Vorkonditionierer kennengelernt, eine potentiell mächtige Vorkonditionierungstechnik. Gerade durch die Konfiguration der einzelnen Blöcke innerhalb der Matrix M ergeben sich unzählige Möglichkeiten, den Vorkonditionierer dem jeweils betrachteten Problem, also insbesondere der Besetzung der Systemmatrix A, anzupassen.

Schließlich haben wir im vorangegangenen Unterkapitel 3.3 das FGMRES-Verfahren sowie zugehörige Erläuterungen angegeben. Die besondere Stärke dieser Variante des klassischen GMRES-Verfahrens besteht gerade in der Möglichkeit, in jedem Schritt unterschiedliche Vorkonditionierungsmatrizen M_j verwenden zu können. Dies ist eine unerlässliche Eigenschaft, um unser Vorhaben umsetzen zu können, welches gerade die Ausstattung des FGMRES-Verfahren mit verschiedenen Block-Jacobi-Vorkonditionierungstechniken sein wird. Wir wollen diese Verbindung dabei in einer asynchronen Weise vornehmen, die es zulässt, dass mit Teilen bereits vorkonditionierter Größen gerechnet wird, während sich auf weitere Teile der entsprechenden Größe dieser Effekt noch nicht ausgewirkt hat. Die namensgebende flexible Natur des FGMRES-Verfahrens wird schließlich trotz dieser verwendeten Asynchronität Konvergenz sichern.

Anhand numerischer Experimente wollen wir für unterschiedliche Probleme, die aus der Diskretisierung einer partiellen Differentialgleichungen entstehen, die Leistungsfähigkeit des entstandenen Verfahrens einer eingehenden Analyse unterziehen. Wir wollen insbesondere mit der Robustheit der Konvergenzeigenschaften des Verfahrens bezüglich des Sachverhalts experimentieren, dass die Vorkonditionierung für einige Verfahrensschritte aus Teilen der vorzukonditionierenden Größen gänzlich ausfällt, in anderen Teilen jedoch vollständig unternommen wird. Dies werden wir – wie wir später genauer erläutern – künstlich induzieren.

3.4.1 Synchrone Tests

Übersicht der untersuchten Probleme

Zu Beginn wollen wir abseits jeder Asynchronität Referenzwerte des FGMRES-Verfahrens ermitteln. Ziel ist hierbei zunächst die Ermittlung eines adäquaten Restartwerts m anhand der numerischen Evaluierung dreier Beispielprobleme. Als mögliche Kandidaten für m haben wir die Zweierpotenzen $2, 4, 8, 16$ und 32 gesetzt.

Bei den Beispielproblemen handelt es sich um lineare Gleichungssysteme, die durch Diskretisierung einer partiellen Differentialgleichung mittels der Finite-Elemente-Methode entstehen. Konkret handelt es sich bei der partiellen Differentialgleichung um das sogenannte *Poisson-Problem*

$$\begin{aligned} -\Delta u &= f \text{ auf } \Omega \text{ und} \\ g &= 0 \text{ auf } \partial\Omega \end{aligned}$$

für eine offene und beschränkte Teilmenge $\Omega \subset \mathbb{R}^d$. Dieses Problem wurde mit Hilfe der Software *FEAST*[1] (*Finite Element Analysis and Solutions Tools*, vgl. Turek et al. [28, 29]) für den Spezialfall $f \equiv 1$ und $d = 2$ diskretisiert.

Die Finite-Elemente-Methode basiert nun darauf, dass an die Stelle des kontinuierlichen Gebiets Ω ein diskretes Gitter tritt, so dass die Differentialgleichung nur punktweise in den entsprechenden Gitterknoten approximativ gelöst wird. Die drei erwähnten Probleme unterscheiden sich nun gerade durch die Wahl des Gitters: Im ersten Fall (*Problem A*) wählen wir ein Einheitsquadrat, welches regulär verfeinert werden soll. Dabei wird in jedem Verfeinerungsschritt die Kantenlänge des Gitters halbiert. Zur Illustration zeigt Abbildung 3.1 Verfeinerungslevel 1, also das Ausgangsgitter. Abbildung 3.2 zeigt ferner die beiden darauffolgenden Verfeinerungsstufen 2 und 3.

[1] `http://www.feast.tu-dortmund.de/`

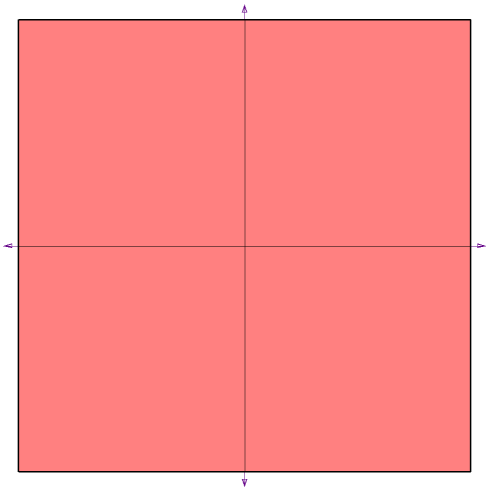

Abbildung 3.1: Verfeinerungslevel 1 (Problem A)

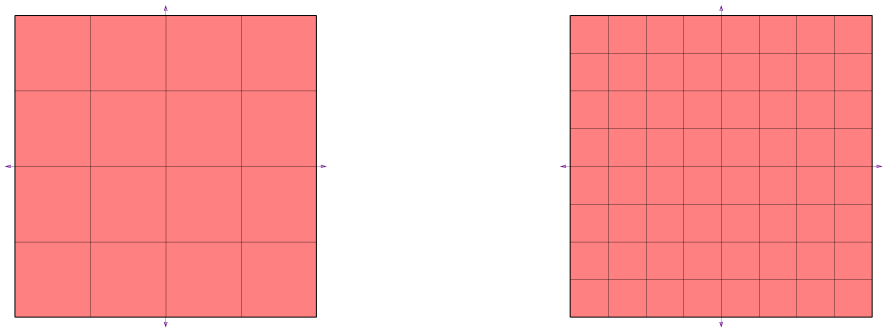

Abbildung 3.2: Verfeinerungslevel 2 und 3 (Problem A)

Im zweiten Fall (*Problem B*) verwenden wir – um unseren Löser mit größerer spektraler Komplexität zu konfrontieren – ein verzerrtes Einheitsquadrat, welches anisotrop innerhalb der südlichsten sowie östlichsten Elementschicht verfeinert wird. Dabei teilen wir die jeweils kürzeren Kanten im Verhältnis $\frac{3}{4} : \frac{1}{4}$. Alle restlichen Kanten werden auch weiterhin halbiert. Zur Verdeutlichung zeigt Abbildung 3.3 Verfeinerungsstufe 1 respektive Abbildung 3.4 die Verfeinerungsstufen 2 und 3 des entsprechenden Gitters.

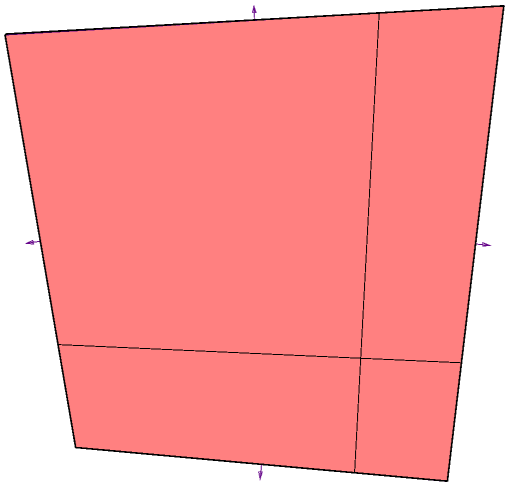

Abbildung 3.3: Verfeinerungslevel 1 (Problem B)

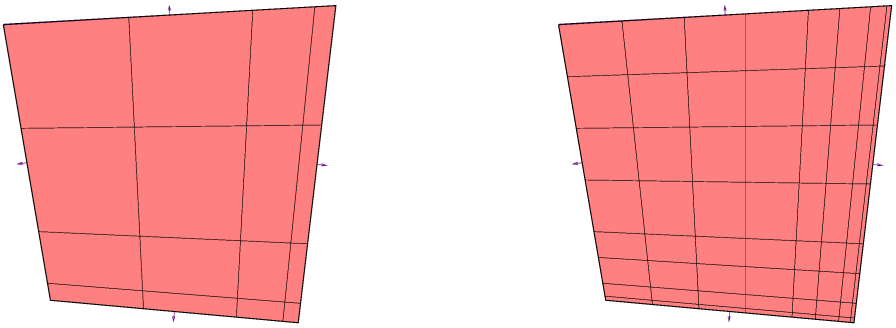

Abbildung 3.4: Verfeinerungslevel 2 und 3 (Problem B)

Im letzten Fall (*Problem C*) betrachten wir ein spezielles rechteckiges Gitter mit einer Singulärität im linken Teil. Dieses entstammt einer Veröffentlichung von Schäfer und Turek [26]. In jedem Verfeinerungsschritt werden hier alle Kantenlängen halbiert. Das Gitter ist in Abbildung 3.5 dargestellt.

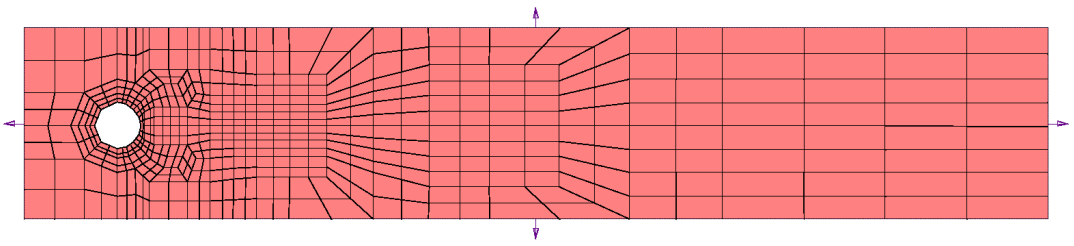

Abbildung 3.5: Verfeinerungslevel 1 (Problem C)

Alle Gitter-Abbildungen wurden mit Hilfe der Software *DeViSoR Grid 3D* erstellt, welche am Lehrstuhl für Angewandte Mathematik und Numerik (LS3) der Fakultät für Mathematik der TU Dortmund entwickelt wurde (vgl. Becker und Göddeke [2]).

Die jeweilige Nummerierung der Gitterpunkte erfolgt nach dem Prinzip des *Two-Level-Numbering*. Hier werden die Punkte des Gitters der Verfeinerungsstufe 1 zunächst lexikographisch (zeilenweise) nummeriert. Alle Punkte, die bei Erhöhung der Verfeinerungsstufe um Eins hinzukommen, werden schließlich ihrerseits lexikographisch nummeriert angehängt (vgl. Geveler et al. [15, Kapitel 2.2]).

Wichtige Basisdaten der resultierenden linearen Gleichungssysteme fassen wir in Tabelle 3.6 zusammen. Diese beinhaltet die Anzahl der Nichtnull-Einträge NNZ (engl. *number of non-zeros*) sowie die Dimension n der erzeugten quadratischen Systemmatrizen A. Ferner wird auch die Konditionszahl $\mathrm{cond}(A)$ (vgl. Definition A.2) dieser Matrizen abgebildet, welche ein wichtiges Indiz für die potentielle Problemkomplexität darstellt. Der Vollständigkeit halber umfasst die Tabelle zudem die Norm des Anfangsresiduums $r^{(0)} = b - Ax^{(0)} = b$, das heißt als Startvektor wird jeweils $x^{(0)} := 0_n$ gesetzt.

Prob.	Lev.	n	NNZ	cond(A)	$\|\|r^{(0)}\|\|_2$
A	3	81	473	13.77	1.09E-01
A	4	289	2089	52.22	5.86E-02
A	5	1089	8777	207.6	3.03E-02
A	6	4225	35977	829.99	1.54E-02
B	3	81	473	52.23	6.18E+00
B	4	289	2089	342.12	1.86E+01
B	5	1089	8777	2731.95	5.53E+01
B	6	4225	35977	22243.82	1.62E+02
C	1	572	4316	263.56	3.76E-02
C	2	2184	17992	1131.35	2.54E-02
C	3	8528	73424	4630.67	1.44E-02

Tabelle 3.6: Übersicht der definierten Probleme

Alle Matrizen weisen eine dünnbesetzte Struktur auf, wie es bei Diskretisierungen mittels Finite-Elemente-Methode die Regel ist. Exemplarisch zeigt Abbildung 3.7 die Besetzungsstruktur der Systemmatrizen A aus Problem A sowie B für den Verfeinerungsgrad 4, das heißt beide Matrizen weisen die gleiche Besetzungsstruktur auf.

Mit zwei- bis fünfstelligen Konditionszahlen sind die Probleme von moderater Komplexität. Nach Konstruktion vervierfacht sich die Anzahl der Unbekannten sowie die Dimension des Systems bei einer Erhöhung der Verfeinerungsstufe um Eins. Hieraus resultiert im Fall von Problem A und B auch eine ungefähre Vervierfachung der Konditionszahl $cond(A)$. Für Problem B nimmt die Komplexität des Systems bedingt durch die von uns konstruierte Anisotropie bei Steigerung des Verfeinerungslevel hingegen nicht linear zu.

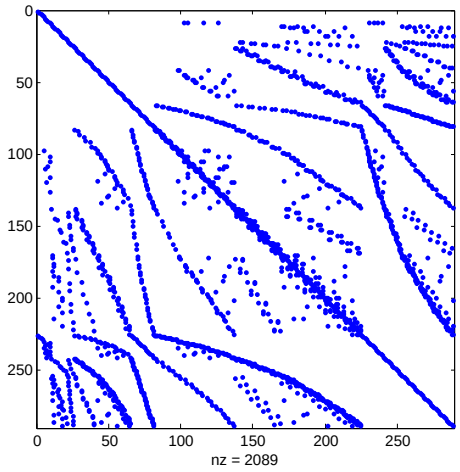

Abbildung 3.7: Besetzungsstruktur der Systemmatrix A (Problem A/B, Verfeinerungslevel 4)

Keine Vorkonditionierung

Zwecks einer ersten Übersicht lösen wir Problem A mit dem FGMRES-Algorithmus ohne die Verwendung eines Vorkonditionierers. Dieses Verfahren ergibt sich durch die Setzung von $M_j = I$ für alle $j = 1, \ldots, m$ in Algorithmus 3.7. Wir durchlaufen die äußere Iterationsschleife dabei solange, bis für das den Quotient der Residuumsnormen

$$\frac{||r^{(k)}||_2}{||r^{(0)}||_2} < 10^{-7} \tag{3.31}$$

gilt. An dieser Stelle sollte erwähnt werden, dass für diesen Quotienten oft die Bezeichnung „relatives Residuum“ verwendet wird, was für uns genau genommen nicht korrekt ist: Wir beschreiben mit dem Begriff „Residuum“ immer einen Vektor und grenzen diese Bezeichnung präzise zur Norm desselbigen ab.

Die Konvergenzkontrolle mittels Bedingung (3.31) findet in Algorithmus 3.7 nach jedem Durchlauf der äußeren Schleife statt. Die Anzahl der gesamten Iterationen bis zur Konvergenz ist daher immer ein Vielfaches der inneren Schleifenlänge m. Sie ergibt sich demnach als Produkt km der Schleifenlänge m mit der Anzahl der gesamten äußeren Iterationen k bis zur Konvergenz. Die Ergebnisse dieser ersten Simulationsrechnung sind in Tabelle 3.8 zusammengefasst. Die Tabelle beinhaltet Rechnungen für die inneren Schleifenlängen $m = 2, 4, 8, 16, 32$ für jeweils alle vor-

gestellten Größenvarianten von Problem A. Der Faktor τ beschreibt die Zunahme der Gesamtanzahl an Iterationen km bei Erhöhung des Verfeinerungslevel um Eins. Die Zahl κ bezeichnet die Konvergenzrate vom ersten bis km-ten Iterationsschritt (vgl. Definition A.3).

Lev.	n	m	k	km	τ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	2	26	52		1.03E-08	9.42E-08	0.733
4	289	2	105	210	4.04	5.86E-09	9.99E-08	0.926
5	1089	2	413	826	3.93	3.01E-09	9.94E-08	0.981
6	4225	2	1642	3284	3.98	1.53E-09	9.97E-08	0.995
3	81	4	5	20		5.29E-09	4.84E-08	0.431
4	289	4	29	116	5.8	3.75E-09	6.39E-08	0.867
5	1089	4	106	424	3.66	3.02E-09	9.97E-08	0.963
6	4225	4	415	1660	3.92	1.54E-09	9.99E-08	0.990
3	81	8	2	16		3.20E-15	2.92E-14	0.143
4	289	8	6	48	3	1.38E-09	2.35E-08	0.694
5	1089	8	30	240	5	1.81E-09	5.97E-08	0.933
6	4225	8	107	856	3.57	1.41E-09	9.14E-08	0.981
3	81	16	1	16		1.89E-16	1.73E-15	0.119
4	289	16	2	32	2	2.06E-12	3.52E-11	0.471
5	1089	16	6	96	3	1.35E-09	4.45E-08	0.838
6	4225	16	30	480	5	1.11E-09	7.20E-08	0.966
3	81	32	1	32		6.15E-16	5.62E-15	0.359
4	289	32	1	32	1	1.66E-15	2.83E-14	0.377
5	1089	32	2	64	2	2.14E-12	7.06E-11	0.694
6	4225	32	6	192	3	9.59E-10	6.24E-08	0.917

Tabelle 3.8: Übersicht der Testergebnisse ohne Vorkonditionierung, variierendes m (Problem A)

Analoge Tests sind nun in Tabelle 3.9 (Problem B) sowie Tabelle 3.10 (Problem C) zusammengefasst.

Lev.	n	m	k	km	τ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	2	58	116		6.00E-07	9.71E-08	0.870
4	289	2	403	806	6.95	1.85E-06	9.95E-08	0.980
5	1089	2	2843	5686	7.05	5.52E-06	9.98E-08	0.997
6	4225	2	19277	38554	6.78	1.61E-05	1.00E-07	1.000
3	81	4	17	68		2.72E-07	4.40E-08	0.779
4	289	4	105	420	6.18	1.69E-06	9.08E-08	0.962
5	1089	4	718	2872	6.84	5.52E-06	9.99E-08	0.994
6	4225	4	4801	19204	6.69	1.61E-05	9.99E-08	0.999
3	81	8	6	48		9.81E-08	1.59E-08	0.688
4	289	8	29	232	4.83	1.24E-06	6.67E-08	0.931
5	1089	8	180	1440	6.21	5.49E-06	9.93E-08	0.989
6	4225	8	1201	9608	6.67	1.61E-05	9.98E-08	0.998
3	81	16	2	32		2.70E-07	4.36E-08	0.589
4	289	16	9	144	4.5	6.02E-07	3.23E-08	0.887
5	1089	16	46	736	5.11	5.37E-06	9.71E-08	0.978
6	4225	16	300	4800	6.52	1.58E-05	9.80E-08	0.997
3	81	32	1	32		5.00E-12	8.08E-13	0.419
4	289	32	3	96	3	5.97E-07	3.21E-08	0.835
5	1089	32	13	416	4.33	5.20E-06	9.41E-08	0.962
6	4225	32	75	2400	5.77	1.48E-05	9.15E-08	0.993

Tabelle 3.9: Übersicht der Testergebnisse ohne Vorkonditionierung, variierendes m (Problem B)

Lev.	n	m	k	km	τ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
1	572	2	462	924		3.74E-09	9.94E-08	0.983
2	2184	2	2015	4030	4.36	2.53E-09	9.94E-08	0.996
3	8528	2	8096	16192	4.02	1.44E-09	9.99E-08	0.999
1	572	4	115	460		3.54E-09	9.41E-08	0.965
2	2184	4	498	1992	4.33	2.48E-09	9.75E-08	0.992
3	8528	4	2065	8260	4.15	1.43E-09	9.97E-08	0.998
1	572	8	30	240		3.74E-09	9.94E-08	0.935
2	2184	8	128	1024	4.27	2.40E-09	9.43E-08	0.984
3	8528	8	514	4112	4.02	1.44E-09	9.98E-08	0.996
1	572	16	9	144		3.09E-09	8.22E-08	0.893
2	2184	16	35	560	3.89	2.22E-09	8.74E-08	0.971
3	8528	16	133	2128	3.8	1.36E-09	9.47E-08	0.992
1	572	32	4	128		4.42E-10	1.18E-08	0.867
2	2184	32	11	352	2.75	1.07E-09	4.22E-08	0.953
3	8528	32	37	1184	3.36	1.23E-09	8.57E-08	0.986

Tabelle 3.10: Übersicht der Testergebnisse ohne Vorkonditionierung, variierendes m (Problem C)

Wir betrachten an dieser Stelle erneut Satz 3.6. Die Systemmatrizen aller Probleme sind symmetrisch sowie positiv definit, daher lässt sich dieser anwenden. Mit Hilfe der Daten aus Tabelle 3.6 errechnet man für den Wurzel-Faktor innerhalb des Satzes Werte innerhalb des Intervalls $[\frac{99}{100}, 1)$. Durch das Teilen von $||r^{(0)}||_2$ sowie das Ziehen der m-ten Wurzel in (3.27) wird sofort ersichtlich, dass diese Faktoren eine obere Schranke für die Konvergenzrate κ liefern. Befürchtungen, dass die Ausführung eines Restarts im Aufbau der jeweiligen Krylow-Unterraum-Sequenz zu einer deutlich höheren Konvergenzrate führen, treffen glücklicherweise nicht ein. Tatsächlich liegt die erreichte Konvergenzrate sogar oftmals unterhalb des Werts $\frac{99}{100}$, wie obige Tabellen zeigen.

Der Faktor τ der Zunahme der Iterationsanzahl bei Erhöhung der Verfeinerungsstufe um Eins liegt im Mittel bei den Problemen A und B etwa bei vier, bei Problem B etwa bei sechs. Dieser Faktor ist etwas höher im Vergleich zu ande-

ren Verfahren wie etwa dem CG-Verfahren. Hier kann man für die Diskretisierung des Poisson-Problems auf dem Einheitsquadrat (Problem A) zeigen, dass sich bei Vervierfachung der Problemgröße – also Erhöhung des Verfeinerungslevel um Eins – die Anzahl der erwarteten Iterationen bis zur Konvergenz lediglich verdoppelt (vgl. Göddeke [17, Abschnitt 3.4.3]).

Bei allen Problemen ist bei Verdoppelung der Anzahl m der Iterationen bis zum Restart eine ungefähre Halbierung der Anzahl km der gesamten Iterationsschritte zu verzeichnen. Dass ein größeres m eine Beschleunigung des Verfahrens bewirkt, ist logisch, da durch Erhöhung der maximalen Dimension der Krylow-Unterräume schließlich mehr Informationen in die Bestimmung neuer Suchrichtungen einfließen. Andererseits müssen, wie bereits erwähnt, naturgemäß mehr Vektoren zwischengespeichert werden. Da beim Sprung von $m = 16$ auf $m = 32$ nicht mehr für alle Probleme und Systemgrößen eine Halbierung der Iterationszahlen zu beobachten ist, sondern der entsprechende Wert etwas unterhalb der Zwei liegt, scheint die Wahl von $m = 16$ für weitere Experimente einen guten Kompromiss zwischen numerischer Leistungsfähigkeit sowie Speicherplatzbedarf darzustellen.

Klassische Vorkonditionierer

Um die Leistungsfähigkeit des vorkonditionierten FGMRES-Verfahrens im Vergleich zur bisher unvorkonditioniert betrachteten Variante beurteilen zu können, statten wir die Methode nun mit einfachen Standard-Vorkonditionierern aus. Hierzu legen wir erneut die additive Zerlegung der jeweiligen Systemmatrix A

$$A = D + L + R$$

zugrunde und betrachten im Folgenden das Verfahren in Kombination mit dem Jacobi-Vorkonditionierer ($M := D$) sowie dem *Gauß-Seidel-Vorkonditionierer* ($M := D + L$), jeweils für $m = 16$. Die entsprechenden Testergebnisse sind in Tabelle 3.11 (Problem A), Tabelle 3.12 (Problem B) sowie Tabelle 3.13 (Problem C) vergleichend mit dem Fall keiner Vorkonditionierung ($M := I$) dargestellt.

Lev.	n	M	k	km	τ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	I	1	16		1.89E-16	1.73E-15	0.119
4	289	I	2	32	2	2.06E-12	3.52E-11	0.471
5	1089	I	6	96	3	1.35E-09	4.45E-08	0.838
6	4225	I	30	480	5	1.11E-09	7.20E-08	0.966
3	81	D	1	16		4.23E-16	3.87E-15	0.126
4	289	D	2	32	2	2.06E-12	3.52E-11	0.471
5	1089	D	6	96	3	1.35E-09	4.45E-08	0.838
6	4225	D	30	480	5	1.11E-09	7.20E-08	0.966
3	81	$D+L$	1	16		6.20E-13	5.67E-12	0.198
4	289	$D+L$	2	32	2	1.51E-11	2.58E-10	0.502
5	1089	$D+L$	4	64	2	1.22E-10	4.04E-09	0.739
6	4225	$D+L$	15	240	3.75	4.66E-10	3.03E-08	0.930

Tabelle 3.11: Übersicht der Testergebnisse mit Vorkonditionierung für $m = 16$ (Problem A)

Lev.	n	M	k	km	τ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	I	2	32		2.70E-07	4.36E-08	0.589
4	289	I	9	144	4.5	6.02E-07	3.23E-08	0.887
5	1089	I	46	736	5.11	5.37E-06	9.71E-08	0.978
6	4225	I	300	4800	6.52	1.58E-05	9.80E-08	0.997
3	81	D	2	32		1.21E-10	1.96E-11	0.463
4	289	D	4	64	2	1.14E-07	6.11E-09	0.744
5	1089	D	8	128	2	1.92E-06	3.48E-08	0.874
6	4225	D	16	256	2	1.32E-05	8.18E-08	0.938
3	81	$D+L$	1	16		1.15E-10	1.86E-11	0.213
4	289	$D+L$	2	32	2	5.25E-09	2.82E-10	0.503
5	1089	$D+L$	4	64	2	3.96E-07	7.16E-09	0.746
6	4225	$D+L$	7	112	1.75	1.52E-05	9.43E-08	0.866

Tabelle 3.12: Übersicht der Testergebnisse mit Vorkonditionierung für $m = 16$ (Problem B)

Lev.	n	M	k	km	τ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
1	572	I	9	144		3.09E-09	8.22E-08	0.893
2	2184	I	35	560	3.89	2.22E-09	8.74E-08	0.971
3	8528	I	133	2128	3.8	1.36E-09	9.47E-08	0.992
1	572	D	8	128		8.82E-10	2.35E-08	0.872
2	2184	D	24	384	3	1.36E-09	5.34E-08	0.957
3	8528	D	63	1008	2.63	1.21E-09	8.41E-08	0.984
1	572	$D+L$	4	64		7.63E-11	2.03E-09	0.731
2	2184	$D+L$	9	144	2.25	4.56E-10	1.79E-08	0.884
3	8528	$D+L$	25	400	2.78	7.94E-10	5.52E-08	0.959

Tabelle 3.13: Übersicht der Testergebnisse mit Vorkonditionierung für $m = 16$ (Problem C)

Wir bemerken zunächst, dass die Vorkonditionierung in keinem Fall eine Verschlechterung der Iterationszahl km bewirkt. Für Problem B und C ergeben sich sogar deutliche Verbesserungen für beide verwendeten Vorkonditionierer, die sich im Fall von Problem C in einer Reduktion der Anzahl der Iterationsschritte um einen maximalen Faktor größer als fünf und im Fall von Problem B sogar um einen Faktor jenseits der 40 widerspiegeln. Bei Problem A zeigt sich hingegen nur bei Verwendung des Gauß-Seidel-Vorkonditionierers eine Verbesserung der Iterationszahl. Dadurch, dass bei Problem A keine Anisotropie im Gitter vorhanden ist, bewirkt der Jacobi-Vorkonditionierer hier im Wesentlichen lediglich eine Skalierung des Vektors, auf den die Vorkonditionierung angewandt werden soll, da bis auf die mit Randknoten assoziierten Diagonaleinträge der Matrix A alle Einträge gleich sind. Vergleicht man die Anzahl der Gesamtiterationen km zwischen der Anwendung des Jacobi-Vorkonditionierers und des Gauß-Seidel-Vorkonditionierers, zeigt sich jeweils eine Halbierung der Iterationszahl, was zumindest im Kontext von Problem A auch theoretisch untermauert ist (vgl. Göddeke [17, Abschnitt 3.3.3]).

Block-Jacobi-Vorkonditionierer

Als weitere Vorbereitung auf asynchrone Tests wollen wir nun das FGMRES-Verfahren mit verschiedenen Varianten des Block-Jacobi-Vorkonditionierers aus-

rüsten. Wir versprechen uns an dieser Stelle Aufschluss über die entsprechenden Wirkungen verschiedener Blockgrößen δ_j sowie jene unterschiedlicher Übernahmestrategien innerhalb dieser Blöcke. Unter der Blockgröße δ_j verstehen wir dabei die Mächtigkeit der Mengen S_j aus Definition 3.5, also die Dimension des entsprechenden j-ten Diagonalblocks j der Block-Diagonalmatrix M. Wir verwenden im Folgenden der Einfachheit halber ausschließlich Blöcke von einheitlicher Größe, betrachten die Blockgröße also unabhängig von j und verwenden demnach zu ihrer Kennzeichnung lediglich den Buchstaben δ ohne Indizierung. Sollte dabei diese universelle Blockgröße δ die Größe n der Systemmatrix A nicht teilen, setzen wir die Größe des letzten Blocks auf die universell gewünschte Größe δ plus den Rest den diese beim Teilen der Systemgröße n lässt. Der letzte Block besitzt also eine maximale Größe kleiner als 2δ. Bei einer gewünschten Blockgröße von $\delta = 3$ und einer Systemgröße von $n = 10$ würde unser Vorkonditionierer also aus zwei (3×3)-Blöcken sowie einem (4×4)-Block bestehen.

Eine erste Übernahmestrategie, die wir verwenden wollen, soll alle Nichtnull-Einträge der Matrix A innerhalb der ausgewählten Blockstruktur übernehmen, eine zweite soll lokal auf jeden Block die Besetzungsstruktur des Gauß-Seidel-Vorkonditionierers anwenden, also alle potentiell vorhandenen Nichtnull-Einträge oberhalb der Diagonalen löschen. Für die entsprechenden Vorkonditionierungsmatrizen verwenden wir die Bezeichnungen

$$M_{\mathrm{BJF}}(\delta) \text{ respektive } M_{\mathrm{BJGS}}(\delta).$$

Dabei stehen die Buchstaben „BJF“ respektive „BJGS“ für „Block-Jacobi Full“ respektive „Block-Jacobi Gauß-Seidel“. Sollte die Blockgröße δ ferner aus dem Kontext hervorgehen, so werden wir diese in der Bezeichnung der Vorkonditionierer gegebenenfalls unterdrücken.

Die folgenden Tabellen stellen die Ergebnisse unserer Simulationsrechnungen für beide Vorkonditionierer, die Blockgrößen $\delta = 4, \lceil\sqrt{n}\rceil, 40$ sowie alle bisher eingeführten Probleme dar. Die Werte 4 und 40 sind dabei relativ willkürlich gewählt, wohingegen $\lceil\sqrt{n}\rceil$ eine in der Problemgröße n variable Blockgröße darstellt.

Die Länge der inneren Schleife $m = 16$ wird weiterhin beibehalten. Tabelle 3.14 enthält dabei die entsprechenden Daten für Problem A, Tabelle 3.15 für Problem B und 3.16 schließlich für Problem C.

Lev.	n	M	δ	k	km	τ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	M_{BJF}	4	1	16		7.77E-09	7.10E-08	0.357
4	289	M_{BJF}	4	3	48	3	3.37E-09	5.75E-08	0.707
5	1089	M_{BJF}	4	6	96	2	2.36E-09	7.79E-08	0.843
6	4225	M_{BJF}	4	28	448	4.67	1.38E-09	8.98E-08	0.964
3	81	M_{BJF}	9	2	32		3.25E-15	2.97E-14	0.378
4	289	M_{BJF}	17	3	48	1.5	1.20E-11	2.05E-10	0.628
5	1089	M_{BJF}	33	6	96	2	1.53E-09	5.06E-08	0.839
6	4225	M_{BJF}	65	26	416	4.33	1.10E-09	7.12E-08	0.961
3	81	M_{BJF}	40	1	16		6.86E-16	6.27E-15	0.130
4	289	M_{BJF}	40	3	48	3	3.91E-11	6.67E-10	0.644
5	1089	M_{BJF}	40	6	96	2	1.97E-09	6.50E-08	0.842
6	4225	M_{BJF}	40	26	416	4.33	1.36E-09	8.85E-08	0.962
3	81	M_{BJGS}	4	1	16		8.76E-09	8.01E-08	0.360
4	289	M_{BJGS}	4	3	48	3	1.08E-10	1.84E-09	0.658
5	1089	M_{BJGS}	4	6	96	2	3.02E-09	9.99E-08	0.845
6	4225	M_{BJGS}	4	29	464	4.83	1.38E-09	8.97E-08	0.966
3	81	M_{BJGS}	9	1	16		1.09E-08	9.95E-08	0.365
4	289	M_{BJGS}	17	3	48	3	7.07E-11	1.21E-09	0.652
5	1089	M_{BJGS}	33	6	96	2	2.71E-09	8.95E-08	0.844
6	4225	M_{BJGS}	65	29	464	4.83	1.04E-09	6.76E-08	0.965
3	81	M_{BJGS}	40	1	16		3.37E-09	3.08E-08	0.339
4	289	M_{BJGS}	40	3	48	3	1.65E-10	2.81E-09	0.664
5	1089	M_{BJGS}	40	6	96	2	3.00E-09	9.90E-08	0.845
6	4225	M_{BJGS}	40	29	464	4.83	1.21E-09	7.89E-08	0.965

Tabelle 3.14: Übersicht der Testergebnisse mit Block-Jacobi-Vorkonditionierung für $m = 16$ (Problem A)

Lev.	n	M	δ	k	km	τ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	M_{BJF}	4	2	32		7.04E-11	1.14E-11	0.455
4	289	M_{BJF}	4	4	64	2	3.17E-08	1.70E-09	0.729
5	1089	M_{BJF}	4	7	112	1.75	3.40E-06	6.16E-08	0.862
6	4225	M_{BJF}	4	14	224	2	8.47E-06	5.24E-08	0.928
3	81	M_{BJF}	9	2	32		3.95E-11	6.39E-12	0.447
4	289	M_{BJF}	17	3	48	1.5	7.99E-07	4.29E-08	0.702
5	1089	M_{BJF}	33	7	112	2.33	1.47E-06	2.65E-08	0.856
6	4225	M_{BJF}	65	13	208	1.86	6.45E-06	3.99E-08	0.921
3	81	M_{BJF}	40	1	16		2.12E-07	3.43E-08	0.342
4	289	M_{BJF}	40	3	48	3	4.99E-07	2.68E-08	0.695
5	1089	M_{BJF}	40	7	112	2.33	1.03E-06	1.87E-08	0.853
6	4225	M_{BJF}	40	13	208	1.86	7.73E-06	4.79E-08	0.922
3	81	M_{BJGS}	4	2	32		9.88E-11	1.60E-11	0.460
4	289	M_{BJGS}	4	4	64	2	6.67E-08	3.58E-09	0.738
5	1089	M_{BJGS}	4	7	112	1.75	5.01E-06	9.06E-08	0.865
6	4225	M_{BJGS}	4	15	240	2.14	1.15E-05	7.10E-08	0.934
3	81	M_{BJGS}	9	2	32		7.80E-11	1.26E-11	0.456
4	289	M_{BJGS}	17	4	64	2	2.93E-08	1.58E-09	0.729
5	1089	M_{BJGS}	33	7	112	1.75	3.32E-06	6.01E-08	0.862
6	4225	M_{BJGS}	65	15	240	2.14	8.23E-06	5.10E-08	0.932
3	81	M_{BJGS}	40	2	32		1.01E-12	1.64E-13	0.399
4	289	M_{BJGS}	40	3	48	1.5	1.55E-06	8.33E-08	0.712
5	1089	M_{BJGS}	40	7	112	2.33	2.64E-06	4.77E-08	0.860
6	4225	M_{BJGS}	40	15	240	2.14	9.62E-06	5.96E-08	0.933

Tabelle 3.15: Übersicht der Testergebnisse mit Block-Jacobi-Vorkonditionierung für $m = 16$ (Problem B)

Lev.	n	M	δ	k	km	τ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
1	572	M_{BJF}	4	7	112		1.63E-09	2.64E-10	0.821
2	2184	M_{BJF}	4	21	336	3	1.42E-09	7.62E-11	0.933
3	8528	M_{BJF}	4	63	1008	3	1.16E-09	2.10E-11	0.976
1	572	M_{BJF}	24	6	96		3.20E-09	1.98E-11	0.774
2	2184	M_{BJF}	47	19	304	3.17	2.24E-09	3.63E-10	0.931
3	8528	M_{BJF}	93	60	960	3.16	1.27E-09	6.83E-11	0.976
1	572	M_{BJF}	40	6	96		3.38E-09	6.11E-11	0.783
2	2184	M_{BJF}	40	19	304	3.17	2.44E-09	1.51E-11	0.921
3	8528	M_{BJF}	40	62	992	3.26	1.21E-09	1.95E-10	0.978
1	572	M_{BJGS}	4	7	112		3.30E-09	1.77E-10	0.818
2	2184	M_{BJGS}	4	22	352	3.14	1.86E-09	3.37E-11	0.934
3	8528	M_{BJGS}	4	49	784	2.23	9.86E-10	6.10E-12	0.968
1	572	M_{BJGS}	24	7	112		9.85E-10	1.59E-10	0.818
2	2184	M_{BJGS}	47	21	336	3	1.63E-09	8.77E-11	0.933
3	8528	M_{BJGS}	93	48	768	2.29	8.67E-10	1.57E-11	0.968
1	572	M_{BJGS}	40	7	112		8.39E-10	5.19E-12	0.793
2	2184	M_{BJGS}	40	21	336	3	1.84E-09	2.97E-10	0.937
3	8528	M_{BJGS}	40	50	800	2.38	1.40E-09	7.54E-11	0.971

Tabelle 3.16: Übersicht der Testergebnisse mit Block-Jacobi-Vorkonditionierung für $m = 16$ (Problem C)

Insgesamt zeigen sich in der Anzahl der Verfahrensschritte bis zur Konvergenz nur wenig Beeinflussungen durch die Variation der Blockgröße δ des Vorkonditionierers sowie dessen selbst. Tendentiell zeichnet sich in einigen Kombinationen aus Problem und Art des Vorkonditionierers durch Erhöhung der Blockgröße δ jedoch eine Verbesserung in der Anzahl der Iterationsschritte ab, was durch eine Hinzunahme von mehr Informationen aus der Systemmatrix A in die des Vorkonditionierers zu erklären ist. So hat der BJF-Vorkonditionierer im Vergleich zum BJGS-Vorkonditionierer auf die Systemmatrizen der Probleme A und B in Bezug auf die Iterationszahl km eine positivere Wirkung. Für Problem C kehrt sich dieser Effekt jedoch gerade um.

Verhalten bei großen Blöcken

Schließlich möchten wir noch die Auswirkungen einer Vorkonditionierung mit dem BJF-Vorkonditionierer für extrem große Blöcke testen. Dazu ziehen wir jedes Problem jeweils in seiner größten von uns definierten Form heran und verwenden die Blockgrößen $\delta = \lfloor \frac{n}{16} \rfloor, \lfloor \frac{n}{8} \rfloor, \lfloor \frac{n}{4} \rfloor, \lfloor \frac{n}{2} \rfloor$. Natürlich sind derartige δ-Werte im Fall dieses Vorkonditionierers (im Gegensatz zum BJGS-Vorkonditionierer) nur bedingt praxisrelevant, jedoch ignorieren wir dies, um asymptotisch beste Fälle zu erzeugen. Die entsprechenden Versuchsauswertungen werden in Tabelle 3.17 dargestellt.

Prob.	Lev.	n	δ	k	km	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
A	6	4225	264	25	400	1.18E-09	7.65E-08	0.960
A	6	4225	528	26	416	8.57E-10	5.56E-08	0.961
A	6	4225	1056	28	448	1.17E-09	7.59E-08	0.964
A	6	4225	2112	21	336	9.53E-10	6.19E-08	0.952
B	6	4225	264	12	192	9.52E-06	5.88E-08	0.917
B	6	4225	528	12	192	1.10E-05	6.80E-08	0.918
B	6	4225	1056	12	192	7.84E-06	4.84E-08	0.916
B	6	4225	2112	9	144	5.87E-06	3.62E-08	0.888
C	3	8528	533	58	928	1.18E-09	8.18E-08	0.983
C	3	8528	1066	58	928	1.24E-09	8.59E-08	0.983
C	3	8528	2132	51	816	1.34E-09	9.33E-08	0.980
C	3	8528	4264	22	352	9.40E-10	6.53E-08	0.954

Tabelle 3.17: Übersicht der Testergebnisse mit BJF-Vorkonditionierung über große Blöcke für $m = 16$

Wie man sieht, sinken die Iterationszahlen durch Ausweitung der Blockgröße δ im Vergleich zu den bisherigen Tests mit BJF-Vorkonditionierung nur gering (vgl. Tabellen 3.14, 3.15 und 3.16). Erst bei einer Blockgröße von $\delta = \lfloor \frac{n}{2} \rfloor$ sinkt die Iterationszahl km weiter. Da derartige Blockgrößen für Probleme mit großer Dimension n jedoch nicht von praktischer Relevanz sind, schließlich müssen in der Vorkonditionierung zwei Systeme mit der Hälfte der Unbekannten des Ausgangssystems gelöst werden, legitimieren obige Testergebnisse unsere Wahl der zu untersuchenden Blockgrößen $\delta = 4, \lceil \sqrt{n} \rceil, 40$.

3.4.2 Asynchrone Tests

Nachdem wir bislang das FGMRES(m)-Verfahren ausgiebig im Fall einer synchronen Vorkonditionierung getestet haben, wollen wir uns nun der Simulation einer asynchron stattfindenden Block-Jacobi-Vorkonditionierung widmen. Dabei soll das FGMRES-Verfahren selbst also synchron arbeiten.

Wir gehen davon aus, dass die Anwendung des Vorkonditionierers parallel auf unterschiedlichen Prozessoren erfolgen würde. Dabei werden die Zuständigkeiten der einzelnen Prozessoren wieder abschnittsweise auf die einzelnen Einträge des zu berechnenden Vektors verteilt. Wir nehmen dabei der Einfachheit halber an, dass jeder Prozessor mit einem Block innerhalb des Vorkonditionierers assoziiert wird, die Aufteilung unter den Prozessoren sich insbesondere also an den Blöcken des Vorkonditionierers orientiert.

Da unsere synchronen Tests in Abschnitt 3.4.1 im Wesentlichen eine einheitliche Reaktion unserer Testprobleme auf beide evaluierten Block-Vorkonditionierer zeigen, beschränken wir uns nun aus Gründen der Kompaktheit auf den BJF-Vorkonditionierer, verwenden also in jedem Block eine vollständige Übernahmestrategie und damit eine exakte Invertierung in jedem Block.

Findet die Vorkonditionierung im FGMRES-Verfahren letztlich also asynchron statt, wird es vorkommen, dass einzelne Prozessoren den Teil der Vorkonditionierung, für den sie zuständig sind, bereits berechnet haben, während andere ihren Teil noch bearbeiten. In einem solchen Fall wird das Verfahren dann innerhalb der verzögerten Bereiche des vorzukonditionierenden Vektors mit den Daten rechnen, die seit der letzten abgeschlossenen Vorkonditionierung in diesem Block des Vektors zur Verfügung stehen. Mathematisch kommt dies der Anwendung der Einheitsmatrix I in diesem Block des Vorkonditionierers gleich, während in den nicht verspäteten Blöcken die vollständige Vorkonditionierung wie im synchronen Fall zum Tragen kommt. Solange die Verspätung dieser Blöcke weiter andauert, wird dieser Vorgang in unseren Simulationen wiederholt. Wenn schließlich die Berechnung der verspäteten Teile abgeschlossen ist, werden diese bei der aktuell vorzukonditionierenden Größe als Rückgabe eingesetzt. Auf diese Weise simulieren wir also ein verspätet eintreffendes Vorkonditionierungsergebnis, das sich mathematisch auf eine Größe aus einer bereits vergangenen Iteration bezieht.

Abbildung 3.18 illustriert dieses Vorgehen. Sie stellt bereits in der Notation des

FGMRES-Verfahrens gehalten die Vorkonditionierung der Vektoren $v^{(1)}, \ldots, v^{(9)}$ dar, wobei die Arbeit auf drei Prozessoren verteilt wird und jede Zeile jeweils den gesamten Vektor symbolisiert. Prozessor 1 und 3 liefern ihre Kalkulationen rechtzeitig, während Prozessor 2 nach dem ersten Iterationsschritt plötzlich mehr Zeit benötigt und somit nur die unvorkonditionierte Größe zur Verfügung steht. In diesem Beispiel benötigt Prozessor 2 für die verzögerte Vorkonditionierung soviel Zeit, dass die beiden anderen Prozessoren gleichzeitig in der Vorkonditionierung bis zum Ergebnis $z^{(8)}$ fortschreiten und im mittleren Block daher das Verfahren in der Vorkonditionierung lediglich die Identität verwendet. Erst für $z^{(9)}$ trifft das Ergebnis der Vorkonditionierung von $v^{(2)}$ schließlich verspätet ein. Der sich abdunkelnde Grünton deutet darauf hin, wie oft der Vorkonditionierer M auf jeden Block bereits angewandt wurde.

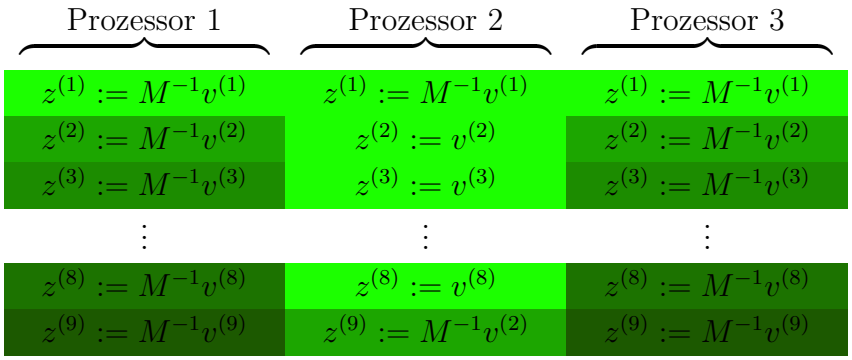

Abbildung 3.18: Darstellung einer Asynchronität in der Vorkonditionierung mit drei Blöcken im Rahmen des FGMRES-Verfahrens

Wir weisen darauf hin, dass die Bezeichnungen in obiger Abbildung der Kompaktheit halber nicht vollständig korrekt sind. Genauer müssten hier eigentlich Teilvektoren in jeder Zelle abgebildet sein, also etwa in der ersten Zelle oben links $z^{(1)}(1 : \frac{n}{3})$ statt $z^{(1)}$ für eine Dimension n, die ein Vielfaches von drei ist.

Unsere Berechnungen sind dabei nur bedingt praxistauglich: Schließlich gibt es keine Veranlassung, anzunehmen, ein Prozessor, der sich bei der Vorkonditionierung verspätet, sei in der Lage, die übrigen Operationen des FGMRES-Algorithmus synchron durchzuführen. Unser wie oben beschriebenes Vorgehen sehen wir dennoch gerechtfertigt, da in der Praxis der Vorkonditionierer in der Regel den wesentlichsten Anteil der benötigten Rechenzeit in Anspruch nimmt. Wir gehen da-

her von einem hypothetischen Computer aus, welcher asynchrone sowie synchrone Rechenknoten (Prozessoren) besitzt. In ähnlicher Form rechtfertigen Hoemmen und Heroux [19] in einem Bericht ihr dortiges Vorgehen, dies jedoch im Kontext fehlertoleranter, statt wie hier asynchroner Verfahren. Die Autoren sprechen von „zuverlässigen“ und „unzuverlässigen“ Clusterknoten (engl. *high reliability* bzw. *bulk reliability*).

Vollständiges Ausbleiben in einem Block

Zu Beginn möchten wir das Verfahren mit einem Grenzfall konfrontieren. Dieser ist gerade die Sitation, dass die Vorkonditionierung in einem einzelnen Block vollständig ausfällt, die in Abbildung 3.18 gezeigte Aktualisierung mit einem veralteten Ergebnis also niemals stattfindet. Zur Simulation dieses Sachverhalts wählen wir in den betrachteten Problemen jeweils einen mittleren Block. Genauer wählen wir jeweils den γ-ten Block mit

$$\gamma := \left\lfloor \frac{\#\text{Blöcke}}{2} \right\rfloor = \left\lfloor \frac{\lfloor \frac{n}{\delta} \rfloor}{2} \right\rfloor .$$

Die Ergebnisse der entsprechenden Tests sind in Tabelle 3.19 (Problem A), Tabelle 3.20 (Problem B) respektive Tabelle 3.21 (Problem C) abgefasst. Dabei blenden wir den bisherigen Faktor τ zu Gunsten des Faktors ξ aus. Dieser beschreibt die Zunahme der Iterationszahlen im Vergleich zur vollständig synchronen BJF-Vorkonditionierung aus Abschnitt 3.4.1 (Tabellen 3.14, 3.15 und 3.16).

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	4	2	32	2	3.55E-14	3.24E-13	0.407
4	289	4	3	48	1	1.11E-09	1.90E-08	0.690
5	1089	4	11	176	1.83	2.47E-09	8.16E-08	0.911
6	4225	4	36	576	1.29	1.34E-09	8.71E-08	0.972
3	81	9	2	32	1	1.00E-11	9.16E-11	0.486
4	289	17	4	64	1.33	5.34E-10	9.11E-09	0.749
5	1089	33	15	240	2.5	2.47E-09	8.17E-08	0.934
6	4225	65	49	784	1.88	1.38E-09	8.98E-08	0.980
3	81	40	1	16	1	6.86E-16	6.27E-15	0.130
4	289	40	4	64	1.33	2.34E-09	3.99E-08	0.766
5	1089	40	16	256	2.67	1.18E-09	3.89E-08	0.936
6	4225	40	50	800	1.92	1.24E-09	8.05E-08	0.980

Tabelle 3.19: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter vollständigem Auslassen eines mittleren Blocks γ für $m = 16$ (Problem A)

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	4	2	32	1	4.65E-10	7.53E-11	0.483
4	289	4	4	64	1	1.25E-06	6.69E-08	0.773
5	1089	4	9	144	1.29	2.54E-06	4.59E-08	0.889
6	4225	4	20	320	1.43	1.35E-05	8.34E-08	0.950
3	81	9	9	144	4.5	2.30E-08	3.72E-09	0.874
4	289	17	5	80	1.67	3.95E-07	2.12E-08	0.802
5	1089	33	19	304	2.71	2.64E-06	4.78E-08	0.946
6	4225	65	57	912	4.38	1.55E-05	9.58E-08	0.982
3	81	40	1	16	1	2.12E-07	3.43E-08	0.342
4	289	40	10	160	3.33	1.18E-06	6.35E-08	0.902
5	1089	40	21	336	3	4.01E-06	7.25E-08	0.952
6	4225	40	58	928	4.46	1.43E-05	8.84E-08	0.983

Tabelle 3.20: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter vollständigem Auslassen eines mittleren Blocks γ für $m = 16$ (Problem B)

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
1	572	4	9	144	1.29	1.18E-09	1.90E-10	0.856
2	2184	4	26	416	1.24	2.24E-09	1.20E-10	0.947
3	8528	4	90	1440	1.43	1.43E-09	2.60E-11	0.983
1	572	24	12	192	2	1.59E-09	9.83E-12	0.876
2	2184	47	30	480	1.58	2.00E-09	3.24E-10	0.955
3	8528	93	106	1696	1.77	1.30E-09	6.97E-11	0.986
1	572	40	10	160	1.67	1.55E-09	2.81E-11	0.859
2	2184	40	34	544	1.79	1.86E-09	1.15E-11	0.955
3	8528	40	100	1600	1.61	1.34E-09	2.16E-10	0.986

Tabelle 3.21: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter vollständigem Auslassen eines mittleren Blocks γ für $m = 16$ (Problem C)

Die obigen Tabellen – und insbesondere der Faktor ξ – zeigen, dass das Auslassen der Vorkonditionierung in einem Block keineswegs unbedeutend ist und dies selbst im Fall von Problem C, wo das Auslassen in einem Block bei $n = 8528$ lediglich einem Auslassen von etwa 0.19 Prozent aller Blöcke entspricht. In den extremsten Fällen wird die Anzahl der Iterationen mehr als vervierfacht. Der Einfluss des Auslassens ist natürlicherweise umso größer, je höher wir δ wählen. So zeichnen sich für $\delta = 4$ lediglich Faktoren ξ kleiner als zwei ab.

Verspätung eines Blocks

Wir zeigen nun, dass die Iterationszahlen durch ein verspätetes Eintreffen der angefragten Vorkonditionierung verglichen mit dem Totalwegfall positiv, verglichen mit der vollständig synchronen Situation jedoch weiterhin negativ beeinflusst werden. Dazu wählen wir wieder den gleichen Block γ wie bisher. Für diesen soll der Vorkonditionierer die ersten sechs Schritte der inneren Schleife synchron arbeiten, sich dann über vier Schritte verspäten und schließlich wieder sechs Iterationen synchron vorkonditionieren. Dieser Prozess findet wie in Abbildung 3.18 illustriert statt und wiederholt sich in jeder inneren Schleife. Wir bemerken noch, dass obige Zahlen so gewählt wurden, dass zwischen zwei unterschiedlichen durch einen Restart getrennten Krylow-Unterräumen keine Wechselwirkungen durch ein verspätet eintreffendes Datum entstehen können. Die Ergebnisse sind in den Tabellen 3.22 (Problem A), 3.23 (Problem B) sowie 3.24 (Problem C) abgefasst.

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	4	2	32	2	5.26E-13	4.81E-12	0.443
4	289	4	3	48	1	3.36E-10	5.73E-09	0.673
5	1089	4	8	128	1.33	2.42E-09	7.99E-08	0.880
6	4225	4	29	464	1.04	1.43E-09	9.32E-08	0.966
3	81	9	2	32	1	3.22E-12	2.94E-11	0.469
4	289	17	3	48	1	4.89E-10	8.35E-09	0.679
5	1089	33	9	144	1.5	8.17E-10	2.70E-08	0.886
6	4225	65	28	448	1.08	1.32E-09	8.61E-08	0.964
3	81	40	1	16	1	5.45E-16	4.99E-15	0.128
4	289	40	3	48	1	8.73E-10	1.49E-08	0.687
5	1089	40	9	144	1.5	1.12E-09	3.69E-08	0.888
6	4225	40	29	464	1.12	1.10E-09	7.14E-08	0.965

Tabelle 3.22: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter Verspätung eines mittleren Blocks γ für $m = 16$ (Problem A)

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	4	2	32	1	9.26E-10	1.50E-10	0.493
4	289	4	4	64	1	6.70E-07	3.60E-08	0.765
5	1089	4	8	128	1.14	3.91E-06	7.08E-08	0.879
6	4225	4	19	304	1.36	1.18E-05	7.31E-08	0.947
3	81	9	2	32	1	7.25E-09	1.17E-09	0.526
4	289	17	4	64	1.33	8.57E-07	4.60E-08	0.768
5	1089	33	10	160	1.43	5.29E-06	9.57E-08	0.904
6	4225	65	42	672	3.23	1.56E-05	9.63E-08	0.976
3	81	40	2	32	2	2.94E-10	4.76E-11	0.476
4	289	40	5	80	1.67	1.30E-07	6.97E-09	0.791
5	1089	40	10	160	1.43	5.12E-06	9.27E-08	0.904
6	4225	40	32	512	2.46	1.47E-05	9.09E-08	0.969

Tabelle 3.23: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter Verspätung eines mittleren Blocks γ für $m = 16$ (Problem B)

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
1	572	4	8	128	1.14	1.95E-09	3.15E-10	0.843
2	2184	4	24	384	1.14	1.54E-09	8.28E-11	0.941
3	8528	4	64	1024	1.02	1.25E-09	2.26E-11	0.976
1	572	24	8	128	1.33	2.04E-09	1.26E-11	0.822
2	2184	47	23	368	1.21	1.76E-09	2.85E-10	0.942
3	8528	93	79	1264	1.32	1.36E-09	7.31E-11	0.982
1	572	40	8	128	1.33	3.37E-09	6.09E-11	0.832
2	2184	40	22	352	1.16	1.90E-09	1.17E-11	0.931
3	8528	40	72	1152	1.16	1.26E-09	2.05E-10	0.981

Tabelle 3.24: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter Verspätung eines mittleren Blocks γ für $m = 16$ (Problem C)

Diese Tests zeigen nahezu ausnahmslos ein besseres Konvergenzverhalten, gemessen an der Anzahl an Iterationen km, als der von uns dargestellte Extremfall einer vollständig ausbleibenden Vorkonditionierung in einem Block. So liegen die Werte für ξ überwiegend zwischen eins und zwei mit Tendenz zur Eins. Die Verwendung einer asynchronen Vorkonditionierung scheint im Hinblick auf den mit ihr verbundenen Verlust an Konvergenzgeschwindigkeit bislang rechtfertigbar.

Verspätung eines Blocks während eines Restarts

Die letzten Tests möchten wir nun wiederholen und dabei derart anpassen, dass Wechselwirkungen zwischen den durch Restart getrennten Krylow-Unterräumen mittels der Asynchronität des γ-ten Blocks proviziert werden. Dazu soll Block γ zunächst 12-mal synchron, achtmal asynchron und schließlich erneut 12-mal synchron vorkonditioniert werden. Es ergibt sich also in dieser Testreihe ein Zyklus im asynchronen Verhalten von zwei äußeren Iterationen (die Länge 32 zweier inneren Schleifen gleicht dem Asynchronitätsmuster „12+8+12“). Wir lassen den mittleren Block γ dabei gerade achtmal verspäten, um Vergleichbarkeit mit den vorherigen Tests zu wahren. Das Asynchronitätsmuster „$12 + 8 + 12$“ produziert nun über zwei äußere Iterationen gleichviele Verspätungen wie das Muster „$6 + 4 + 6$“ der vorangegangenen Tests.

Die entsprechenden Resultate liefern Tabelle 3.25 (Problem A), Tabelle 3.26 (Problem B) sowie Tabelle 3.27 (Problem C).

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	4	2	32	2	1.16E-14	1.06E-13	0.393
4	289	4	3	48	1	5.10E-10	8.71E-09	0.679
5	1089	4	9	144	1.5	9.29E-10	3.07E-08	0.887
6	4225	4	29	464	1.04	1.33E-09	8.63E-08	0.966
3	81	9	2	32	1	2.41E-13	2.20E-12	0.432
4	289	17	3	48	1	8.44E-10	1.44E-08	0.686
5	1089	33	8	128	1.33	1.07E-09	3.55E-08	0.875
6	4225	65	30	480	1.15	1.30E-09	8.45E-08	0.967
3	81	40	1	16	1	8.17E-16	7.47E-15	0.131
4	289	40	3	48	1	3.47E-09	5.93E-08	0.707
5	1089	40	9	144	1.5	5.38E-10	1.78E-08	0.883
6	4225	40	32	512	1.23	9.55E-10	6.21E-08	0.968

Tabelle 3.25: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter Verspätung eines mittleren Blocks γ über zwei durch Restart getrennte Krylow-Unterräume für $m = 16$ (Problem A)

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	4	2	32	1	2.04E-10	3.30E-11	0.470
4	289	4	4	64	1	7.33E-07	3.94E-08	0.766
5	1089	4	8	128	1.14	3.28E-06	5.93E-08	0.878
6	4225	4	21	336	1.5	1.25E-05	7.76E-08	0.952
3	81	9	2	32	1	5.15E-10	8.34E-11	0.484
4	289	17	4	64	1.33	1.36E-06	7.32E-08	0.774
5	1089	33	10	160	1.43	3.99E-06	7.21E-08	0.902
6	4225	65	31	496	2.38	1.27E-05	7.89E-08	0.968
3	81	40	2	32	2	5.52E-12	8.93E-13	0.420
4	289	40	5	80	1.67	6.33E-08	3.40E-09	0.784
5	1089	40	10	160	1.43	3.23E-06	5.85E-08	0.901
6	4225	40	28	448	2.15	1.61E-05	9.98E-08	0.965

Tabelle 3.26: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter Verspätung eines mittleren Blocks γ über zwei durch Restart getrennte Krylow-Unterräume für $m = 16$ (Problem B)

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
1	572	4	8	128	1.14	1.75E-09	2.83E-10	0.842
2	2184	4	22	352	1.05	2.13E-09	1.14E-10	0.937
3	8528	4	73	1168	1.16	1.32E-09	2.39E-11	0.979
1	572	24	8	128	1.33	9.04E-10	5.60E-12	0.817
2	2184	47	25	400	1.32	1.94E-09	3.14E-10	0.947
3	8528	93	79	1264	1.32	1.27E-09	6.81E-11	0.982
1	572	40	8	128	1.33	6.64E-10	1.20E-11	0.822
2	2184	40	23	368	1.21	2.20E-09	1.36E-11	0.934
3	8528	40	85	1360	1.37	1.29E-09	2.09E-10	0.984

Tabelle 3.27: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter Verspätung eines mittleren Blocks γ über zwei durch Restart getrennte Krylow-Unterräume für $m = 16$ (Problem C)

Im Wesentlichen zeichnen sich in den obigen Ergebnissen nun schlechtere ξ-Werte im Vergleich zu der Variante, in der die Asynchronität wechselwirkungslos in einem Krylow-Unterraum verbleibt, ab. Dies lässt sich dadurch erklären, dass ein verspätet eintreffendes Vorkonditionierungsergebnis für einen neu aufgebauten Krylow-Unterraum weniger wertvoll ist, als für jenen, auf den es sich eigentlich bezieht.

Eine Ausnahme bilden dabei die Messungen für Problem B für große n und große δ. Hier liefert das vorherige Muster „$6+4+6$“ etwas bessere Resultate. Dies widerspricht zunächst der Intuition, liefert doch das verspätet eintreffende Datum Informationen über den falschen Krylow-Unterraum. Wir wollen uns hier jedoch lediglich auf die erste Erkenntnis beschränken und halten dieses Phänomen als Ansatz für weiterführende Tests fest.

Eine komplexere Verspätungssituation

Schließlich möchten wir unser asynchrones Verfahren mit einer etwas komplexeren Situation konfrontieren. Im Folgenden bilden wir Untersuchungen ab, bei denen jeder zehnte Block einer Verspätung unterliegen soll. Diese dauert eine bis vier Iterationen an, woraufhin ein synchrones Abarbeiten des betreffenden Blocks von fünf bis 15 Iterationsschritten folgt. Die übrigen Blöcke bleiben die gesamte Rechnung über im synchronen Modus. Die entsprechenden Testergebnisse sind in Tabelle 3.28 (Problem A), Tabelle 3.26 (Problem B) respektive 3.27 (Problem C) dargestellt.

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	4	2	32	2	6.96E-14	6.36E-13	0.416
4	289	4	4	64	1.33	5.63E-10	9.61E-09	0.749
5	1089	4	17	272	2.83	2.54E-09	8.40E-08	0.942
6	4225	4	56	896	2	1.26E-09	8.20E-08	0.982
3	81	9	2	32	1	3.25E-15	2.97E-14	0.378
4	289	17	3	48	1	5.90E-10	1.01E-08	0.681
5	1089	33	9	144	1.5	2.00E-09	6.60E-08	0.892
6	4225	65	36	576	1.38	1.12E-09	7.31E-08	0.972
3	81	40	1	16	1	6.86E-16	6.27E-15	0.130
4	289	40	3	48	1	3.91E-11	6.67E-10	0.644
5	1089	40	11	176	1.83	9.51E-10	3.14E-08	0.907
6	4225	40	41	656	1.58	1.25E-09	8.14E-08	0.975

Tabelle 3.28: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter Verspätung jedes zehnten Blocks für $m = 16$ (Problem A)

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
3	81	4	2	32	1	4.59E-10	7.43E-11	0.482
4	289	4	6	96	1.5	2.67E-07	1.44E-08	0.829
5	1089	4	43	688	6.14	4.65E-06	8.40E-08	0.977
6	4225	4	2138	34208	152.71	1.59E-05	9.87E-08	1.000
3	81	9	9	144	4.5	3.95E-11	6.39E-12	0.836
4	289	17	4	64	1.33	1.22E-06	6.54E-08	0.772
5	1089	33	12	192	1.71	2.00E-06	3.61E-08	0.915
6	4225	65	221	3536	17	1.59E-05	9.82E-08	0.995
3	81	40	1	16	1	2.12E-07	3.43E-08	0.342
4	289	40	3	48	1	4.99E-07	2.68E-08	0.695
5	1089	40	13	208	1.86	3.82E-06	6.90E-08	0.924
6	4225	40	331	5296	25.46	1.37E-05	8.45E-08	0.997

Tabelle 3.29: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter Verspätung jedes zehnten Blocks für $m = 16$ (Problem B)

Lev.	n	δ	k	km	ξ	$\|\|r^{(k)}\|\|_2$	$\frac{\|\|r^{(k)}\|\|_2}{\|\|r^{(0)}\|\|_2}$	κ
1	572	4	11	176	1.57	3.34E-09	5.41E-10	0.886
2	2184	4	45	720	2.14	2.53E-09	1.36E-10	0.969
3	8528	4	384	6144	6.1	1.42E-09	2.57E-11	0.996
1	572	24	8	128	1.33	1.33E-09	8.23E-12	0.819
2	2184	47	25	400	1.32	1.80E-09	2.91E-10	0.947
3	8528	93	151	2416	2.52	1.32E-09	7.07E-11	0.990
1	572	40	8	128	1.33	4.67E-10	8.45E-12	0.819
2	2184	40	34	544	1.79	1.81E-09	1.12E-11	0.955
3	8528	40	209	3344	3.37	1.40E-09	2.26E-10	0.993

Tabelle 3.30: Übersicht der Testergebnisse mit BJF-Vorkonditionierung unter Verspätung jedes zehnten Blocks für $m = 16$ (Problem C)

Zwar zeigen die Werte für Problem A und C insgesamt moderate ξ-Werte, jedoch ist die Zunahme in der Anzahl an Iterationen im Fall von Problem B alarmie-

rend. Zudem zeigen alle Tests ein überproportionales Anwachsen der Iterationszahl bei Steigerung der Problemgröße n.

Interessant ist zudem die Wirkung der Blockgröße δ. Bei konstantem Anteil an asynchronen Blöcken (zehn Prozent) scheinen wenige große insgesamt günstiger zu sein als viele kleine Blöcke.

Zusammenfassung

Abschließend fassen wir die wichtigsten Ergebnisse unserer Testreihen grafisch zusammen. Abbildung 3.31 zeigt die Iterationszahlen km aller Probleme bei ihrer jeweils höchsten Verfeinerungsstufe bei Verwendung des BJF-Vorkonditionierers mit $\delta = \lceil \sqrt{n} \rceil$ und $m = 16$.

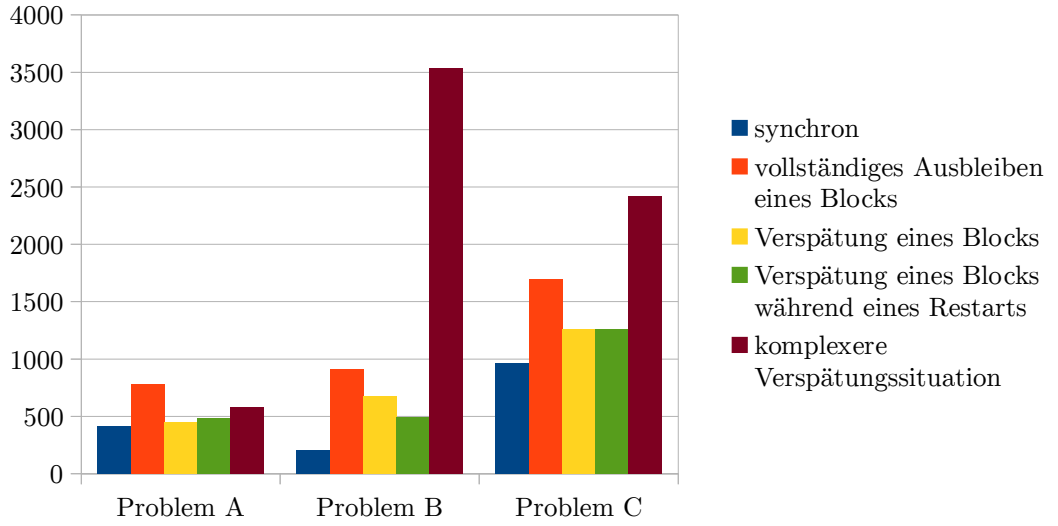

Abbildung 3.31: Vergleichendes Diagramm der Iterationszahlen km aller Probleme bei ihrer jeweils höchsten Verfeinerungsstufe bei Verwendung des BJF-Vorkonditionierers mit $\delta = \lceil \sqrt{n} \rceil$ und $m = 16$

3.5 Fazit und Ausblick

Die in den vorangegangenen Abschnitten durchgeführten Tests zeigen zum einen, dass es sich bei der Block-Jacobi-Technik tatsächlich um einen mächtigen Vorkonditionierer handelt, der durch seine natürliche Blockstruktur ideale Paralleli-

sierungseigenschaften bietet bei deutlicher spektraler Wirkung auf das zu lösende lineare Gleichungssystem.

Zum anderen attestieren sie dem FGMRES(m)-Verfahren in der Tat eine erhebliche Flexibilität in der Vorkonditionierung. Bei all unseren Tests kam es in keinem Fall zu einem Divergenzverhalten. Zwar existieren für verschiedene Problemtypen Konvergenzbeweise für hinreichend große m, jedoch ist die Konvergenz nicht notwendig gesichert, falls die unterschiedlichen Vorkonditionierer M_j zu sehr voneinander abweichen.

In Bezug auf den asynchronen Teil der Experimente muss hervorgehoben werden, dass zwar nicht in allen Fällen ein vollständig überzeugendes Löserverhalten gezeigt wurde, sich jedoch in vielen Fällen ein ξ-Wert lediglich knapp oberhalb der Eins eingestellt hat, so dass die simulierte Asynchronität in diesen Fällen keine spürbare Wirkung zeigte. Dies lässt sich insbesondere für Problem A und C in Abbildung 3.31 erkennen. Hier wird nie mehr als ein Faktor Zwei in der Erhöhung der Iterationszahl im Vergleich zur synchronen Variante des BJF-Vorkonditionierers beobachtet. Auch für Problem B sind die sich zeigenden Erhöhungen noch moderat. Lediglich die von uns getestete komplexere Verspätungssituation reißt aus, jedoch wurde hier auch ein wesentlich größeres Maß an Asynchronität simuliert.

Zur praktischen Umsetzung asynchroner Verfahren, wie das im vorherigen Kapitel vorgestellte, sind weitere Untersuchungen unumgänglich.

So muss unter anderem eine taugliche Strategie entwickelt werden, wie lang ein solches Verfahren warten darf, bis entschieden wird, dass in einem Block unvorkonditioniert weitergerechnet und das Ergebnis jener Vorkonditionierung erst später eingebunden wird.

Ferner ist auch die Untersuchung der Wirkungsweise verschiedener Blockgrößen innerhalb eines Vorkonditionierungsschritts insbesondere im Kontext von Problem B von Interesse. Hier stehen etwa Simulationen aus, wie diese der aufgrund der Verfeinerung in eine Ecke entstehenden Anisotropie optimal angepasst werden kann. Auch ist eine in unterschiedlichen Schritten j wechselnde, aber innerhalb eines Schrittes einheitliche Blockgröße denkbar. In diesem Fall muss jedoch gewährleistet sein, dass die Blockgröße verspätet eintreffender Daten zu jener des aktuellen Schritts kompatibel ist.

Die von uns durchgeführten Evaluationen spiegeln zudem nur einen geringen

Teil aller möglichen Parameterkonfigurationen wider, so dass künftig weitere Untersuchungen sinnvoll erscheinen, etwa hinsichtlich der genauen Wirkung eines Restarts während der Verspätung eines Blocks im Vergleich zu einer Verspätung, die ausschließlich innerhalb eines Krylow-Unterraums wirkt.

Wir wollen weiterhin noch bemerken, dass Collignon und van Gijzen [7] bereits ähnliche Experimente vorgenommen haben. Sie haben die Wirkungsweise einer flexiblen GMRES-Variante mit einem iterativen Block-Jacobi-Vorkonditionierer, der asynchron arbeitet, untersucht. Mathematisch bedeutet dies, dass als Vorkonditionierer ein Verfahren wie die von uns beschriebenen asynchronen Fixpunktiterationsverfahren (Abschnitt 3.1.2) genutzt wird. Dies jedoch mit dem Unterschied, dass nicht etwa wie beim Jacobi-Iterationsverfahren die Diagonale D der Matrix A genutzt wird, sondern die Matrix M_{BJF} des Block-Jacobi-Vorkonditionierers. Entsprechend muss in der Verfahrensvorschrift $L + R$ durch $A - M_{\mathrm{BJF}}$ ersetzt werden und es ergibt sich

$$x^{(k+1)} := -M_{\mathrm{BJF}}^{-1}(A - M_{\mathrm{BJF}})x^{(k)} + M_{\mathrm{BJF}}^{-1}b. \tag{3.32}$$

Der Vorkonditionierungsprozess in Zeile 5 des FGMRES-Verfahrens (Algorithmus 3.7) lautet im Falle des BJF-Vorkonditionierers

$$z^{(j)} := M_{\mathrm{BJF}}^{-1}v^{(j)}$$

für $j = 1, \ldots, m$, das heißt gesucht ist die Lösung $z^{(j)}$ des linearen Gleichungssystems

$$M_{\mathrm{BJF}}z^{(j)} = v^{(j)}. \tag{3.33}$$

Wir bezeichnen nun – abweichend von unserer bisher verwendeten Notation – mit $A_{p,q}$ den p,q-ten Block der Systemmatrix A (in Konsistenz zur Vorkonditionierungsmatrix M_{BJF} bei uniformer Blockgröße δ) und mit $z_q^{(j)}$ sowie $v_q^{(j)}$ für $p, q = 1, \ldots, \lfloor \frac{n}{\delta} \rfloor$ die entsprechenden Bereiche der verwendeten Vektoren. Um das Fixpunktiterationsverfahren (3.32) als Vorkonditionierer im FGMRES-Algorithmus zu verwenden, muss (3.33) damit gelöst werden. Bedenkt man, dass, wenn man in (3.32) mit M_{BJF} multipliziert, sich für jeden Fixpunktverfahrensschritt wiederum ein zu lösendes lineares Gleichungssystem ergibt, und formuliert dieses schließlich um, erhält man

$$A_{p,p}z_p^{(j,k+1)} = v_p^{(j)} - \sum_{q=1, q\neq p}^{\lfloor \frac{n}{\delta} \rfloor} A_{p,q}z_q^{(j,k)} \tag{3.34}$$

für $p = 1, \ldots, \lfloor \frac{n}{\delta} \rfloor$ und Iterationsschritte k. Um diese Folge von Gleichungssystemen zu lösen, wenden Collignon und van Gijzen [7] letztlich ein weiteres iteratives Verfahren an, welches bessere Eigenschaften hinsichtlich der Konvergenzgeschwindigkeit besitzt. Dabei wird auf die gleiche asynchrone Weise vorgegangen, wie wir sie im Kontext asynchroner Fixpunktiterationsverfahren in Abschnitt 3.1.2 dieser Arbeit beschrieben haben. Durch dieses Vorgehen bleibt die Asynchronität aus der Sicht des FGMRES-Lösers im Unterschied zu unserer Herangehensweise aus dem vorangegangenen Kapitel in vollständiger Verborgenheit.

Anhang A

Definitionen

Die folgenden Definitionen geben wir der Vollständigkeit halber wieder. Es handelt sich bei ihnen um allgemein anerkannte Standard-Begrifflichkeiten innerhalb der numerischen Mathematik.

Definition A.1 (Spektrum und Spektralradius)**:** Für eine Matrix $A \in \mathbb{R}^{n \times n}$ definieren wir das *Spektrum von* A als

$$\sigma(A) := \{\lambda \ : \ \lambda \text{ ist Eigenwert von } A\}$$

sowie den *Spektralradius von* A als

$$\text{spr}(A) := \rho(A) := \max\{|\lambda| \ : \ \lambda \in \sigma(A)\}.$$

Definition A.2 (Spektralnorm und -konditionszahl)**:** Für eine Matrix $A \in \mathbb{R}^{n \times n}$ definieren wir die *Spektralnorm* $||A||_2$ *von* A durch

$$||A||_2 := \max_{||x||_2 = 1} ||Ax||_2 = \sqrt{\text{spr}(A^\top A)}.$$

Damit definieren wir die *Spektralkonditionszahl* (oder kurz *Konditionszahl*) $\text{cond}(A)$ *von* A als

$$\text{cond}(A) := \text{cond}_2(A) := ||A||_2 ||A^{-1}||_2.$$

Definition A.3 (Konvergenzrate)**:** Wir bezeichnen die *Konvergenzrate eines iterativen Verfahrens nach* k *Schritten* als

$$\kappa := \kappa(k) := \left(\frac{||b - Ax^{(k)}||_2}{||b - Ax^{(0)}||_2} \right)^{\frac{1}{k}}.$$

Definition A.4 (nichtnegativ bzw. positiv)**:** Es seien $A = (a_{i,j})$ und $B = (b_{i,j})$ zwei $(n \times m)$-Matrizen. Dann sind die Relationen „$\leq$“ und „$<$“ auf $R^{n\times m}$ durch

$$\begin{aligned} A \leq B &:\Leftrightarrow a_{i,j} \leq b_{i,j} \text{ für } i = 1, \ldots, n, j = 1, \ldots, m, \\ A < B &:\Leftrightarrow a_{i,j} < b_{i,j} \text{ für } i = 1, \ldots, n, j = 1, \ldots, m \end{aligned}$$

definiert. Gilt $0_{n,m} \leq A$ beziehungsweise $0_{n,m} < A$, so nennen wir A *nichtnegativ* beziehungsweise *positiv*. Da wir einspaltige Matrizen ohnehin mit Vektoren identifizieren, induziert dies auch eine Relation auf $\mathbb{R}^n$.

Definition A.5 (Hessenberg-Matrix)**:** Eine *obere Hessenberg-Matrix* oder lediglich *Hessenberg-Matrix* ist eine quadratische Matrix $H = (h_{i,j}) \in \mathbb{R}^{n\times n}$, deren Einträge unterhalb der ersten unteren Nebendiagonalen gleich Null sind, für die also

$$H = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \cdots & h_{1,n} \\ h_{2,1} & h_{2,2} & h_{2,3} & \cdots & h_{2,n} \\ 0 & h_{3,2} & h_{3,3} & \cdots & h_{3,n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & h_{n,n-1} & h_{n,n} \end{bmatrix}$$

mit Einträgen $h_{i,j} \in \mathbb{R}$ gilt. Wir sprechen ferner von einer *unteren Hessenberg-Matrix*, falls es sich um eine quadratische Matrix handelt, deren Transponierte eine obere Hessenberg-Matrix ist.

Definition A.6 (Moore-Penrose-Inverse)**:** Die *Moore-Penrose-Inverse* oder *Pseudoinverse* $p(A) := A^+ \in \mathbb{R}^{m\times n}$ einer beliebigen Matrix $A \in \mathbb{R}^{n\times m}$ ist die eindeutige Lösung der Beziehungen

$$\begin{aligned} AA^+A &= A, \\ A^+AA^+ &= A^+, \\ (AA^+)^\top &= AA^+, \\ (A^+A)^\top &= A^+A. \end{aligned}$$

Für den Fall einer regulären Matrix A fällt diese Definition gerade mit jener der üblichen Inversen A^{-1} zusammen.

Anhang B

Sätze

Wir wollen im Folgenden Sätze wiedergeben, die zum tiefgreifenden Verständnis dieser Arbeit nicht obligatorisch sind und höchstens als technische Hilfsmittel in Beweisen genutzt werden.

Satz B.1 (Banachscher Fixpunktsatz)**:** Sei A eine abgeschlossene Teilmenge eines Banachraums, das heißt eines vollständigen normierten Vektorraums $(V, ||\cdot||)$. Die Abbildung $\Phi : A \to A$ sei eine *Kontraktion*, das heißt es gebe eine Konstante ϑ mit $0 < \vartheta < 1$, so dass

$$||\Phi(f) - \Phi(g)|| \leq \vartheta ||f - g||$$

für alle $f, g \in A$ gilt. Dann besitzt Φ genau einen *Fixpunkt*, das heißt es gibt ein eindeutig bestimmtes Element $f_* \in A$ mit

$$\Phi(f_*) = f_*.$$

Für einen beliebigen Anfangswert $f^{(0)} \in A$ konvergiert die durch $f^{(k)} := \Phi(f^{(k-1)})$ rekursiv definierte Folge $f^{(k)}$ für $k \to \infty$ $(k \in \mathbb{N})$ gegen den Fixpunkt f_*.

Beweis: Forster [12, Kapitel 8, Satz 1] □

Satz B.2 (Satz von Perron-Frobenius)**:** Gegeben sei eine nichtnegative Matrix $A \in \mathbb{R}^{n \times n}$. Dann gilt:

(a) Der betraglich größte Eigenwert von A ist positiv. Dieser Eigenwert ist also gleich dem Spektralradius $\rho(A)$.

(b) Zum Eigenwert $\rho(A)$ existiert ein nichtnegativer Eigenvektor u, das heißt es existiert ein $u \in \mathbb{R}^n$ mit $u \geq 0$ sowie $u \neq 0$, der

$$Au = \rho(A)u$$

erfüllt.

Beweis: Varga [30, Theorem 2.1, 2.7] □

Literaturverzeichnis

[1] Richard Barrett, Michael Berry, Tony F. Chan, James W. Demmel, June Donato, Jack J. Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine und Henk A. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 2. Auflage, 1994. ISBN: 9780898713282.

[2] Christian Becker und Dominik Göddeke. DeViSoR Grid. Technischer Bericht, Universität Dortmund, `http://www.feast.tu-dortmund.de/`, 2002.

[3] Adi Ben-Israel und Thomas N. E. Greville. *Generalized Inverses: Theory and Applications*. CMS Books in Mathematics. Springer, 2003. ISBN: 9780387002934.

[4] Kostas Blathras, Daniel B. Szyld und Yuan Shi. Parallel processing of linear systems using asynchronous iterative algorithms. Technischer Bericht, Temple University, Philadelphia, 1997.

[5] Erin Carson, Nicholas Knight und James Demmel. Avoiding communication in two-sided Krylov methods. Technischer Bericht UCB/EECS-2011-93, EECS Department, University of California, Berkeley, 2006.

[6] Anthony T. Chronopoulos und Charles W. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153–168, 1989. ISSN: 03770427. DOI: 10.1016/0377-0427(89)90045-9.

[7] Tijmen P. Collignon und Martin B. van Gijzen. Solving large sparse linear systems efficiently on grid computers using an asynchronous iterative method as a preconditioner, 2008.

[8] Tijmen P. Collignon und Martin B. van Gijzen. Minimizing synchronization in IDR(s). *Numerical Linear Algebra with Applications*, 18(5):805–825, 2011. DOI: 10.1002/nla.764.

[9] James Demmel. Introduction to Communication-Avoiding Algorithms. Präsentation, University of California at Berkeley, Department of Mathematics, `http://www.cs.berkeley.edu/~demmel/`, 2011.

[10] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin und Katherine A. Yelick. Avoiding communication in computing krylov subspaces. Technischer Bericht UCB/EECS-2007-123, EECS Department, University of California, Berkeley, 2007.

[11] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin und Katherine A. Yelick. Avoiding communication in sparse matrix computations. In *IPDPS*, Seiten 1–12. IEEE, 2008.

[12] Otto Forster. *Analysis 2. Differentialgleichungen im $\mathbb{R}^n$, gewöhnliche Differentialgleichungen.* Vieweg Studium. Vieweg + Teubner, 2008. ISBN: 9783834805751.

[13] Andreas Frommer. *Lösung linearer Gleichungssysteme auf Parallelrechnern.* Vieweg, 1990. ISBN: 9783528063979.

[14] Andreas Frommer und Daniel B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123(1–2):201–216, 2000. DOI: 10.1016/S0377-0427(00)00409-X.

[15] Markus Geveler, Dirk Ribbrock, Dominik Göddeke, Peter Zajac und Stefan Turek. Efficient finite element geometric multigrid solvers for unstructured grids on GPUs. In P. Ivànyi und B. Topping (Hrsg.), *Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, Seite 22. Civil-Comp Press, 2011. DOI: 10.4203/ccp.95.22.

[16] Susan L. Graham, Marc Snir und Cynthia A. Patterson. *Getting up to Speed - The Future of Supercomputing.* The National Academies Press, Washington, D.C., November 2004.

[17] Dominik Göddeke. High Performance Computing und parallele Numerik. Vorlesungsskriptum, Technische Universität Dortmund, Fakultät für Mathematik, 2012.

[18] Mark Hoemmen. *Communication-avoiding Krylov subspace methods*. Doktorarbeit, EECS Department, University of California, Berkeley, 2010.

[19] Mark Hoemmen und Michael A. Heroux. Fault-tolerant iterative methods via selective reliability. `http://www.sandia.gov/~maherou/docs/FTGMRES.pdf`, 2011.

[20] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams und Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technischer Bericht, DARPA IPTO, 2008.

[21] Andreas Meister. *Numerik linearer Gleichungssysteme*. Vieweg, 3. Auflage, 2008. ISBN: 9783528131357.

[22] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel und Katherine Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the 2009 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'09)*, Seiten 36:1–36:12, 2009. DOI: 10.1145/1654059.1654096.

[23] Rolf Rannacher. Einführung in die Numerische Mathematik (Numerik 0). Vorlesungsskriptum, Universität Heidelberg, Institut für Angewandte Mathematik, 2006.

[24] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 3. Auflage, 2003. ISBN: 9780898715347.

[25] Yousef Saad und Martin H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scien-*

tific and Statistical Computing, 7(3):856–869, 1986. ISSN: 01965204. DOI: 10.1137/0907058.

[26] Michael Schäfer und Stefan Turek. Benchmark computations of laminar flow around a cylinder. In Ernst H. Hirschel (Hrsg.), *Flow Simulation with High-Performance Computers II*, Band 52 aus *Notes on Numerical Fluid Mechanics*, Seiten 547–566. Vieweg, 1996.

[27] Robert Strzodka und Dominik Göddeke. Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, Seiten 259–270, 2006. DOI: 10.1109/FCCM.2006.57.

[28] Stefan Turek, Dominik Göddeke, Christian Becker, Sven H. M. Buijssen und Hilmar Wobker. FEAST – Realisation of hardware-oriented numerics for HPC simulations with finite elements. *Concurrency and Computation: Practice and Experience*, 22(6):2247–2265, 2010. DOI: 10.1002/cpe.1584.

[29] Stefan Turek, Dominik Göddeke, Sven H. M. Buijssen und Hilmar Wobker. Hardware-oriented multigrid finite element solvers on GPU-accelerated clusters. In Jakub Kurzak, David A. Bader und Jack J. Dongarra (Hrsg.), *Scientific Computing with Multicore and Accelerators*, Kapitel 6, Seiten 113–130. CRC Press, 2010. DOI: 10.1201/b10376-10.

[30] Richard S. Varga. *Matrix Iterative Analysis*. Springer Series in Computational Mathematics. Springer, 2009. ISBN: 9783642051548.

[31] Johann Wolfgang von Goethe. *Italienische Reise*, Band 1-2 aus *Goethe's Meisterwerke, mit Illustrationen deutscher Künstler*. G. Grote, 1870.

Index

Symbole

M-Norm 53

A

Adjazenzgraph 26
Affinität eines Knotens 27
asynchrones Verfahren 68, 72

B

Banachscher Fixpunktsatz 119
Bandbreite 16
Basis
– linksseitige 53
– rechtsseitige 53
Basiskonditionszahl 65
Basiswechsel-Matrix 42, 61
BiCGStab-Verfahren 20, 60
Block-Jacobi-Vorkonditionierung . . . 78

C

CA-BiCGStab-Verfahren 62
CA-CG-Verfahren 44, 48
CG-Verfahren 20
CG3-Verfahren 38
chaotisches Verfahren 68
CPU 25

D

Distributed Memory 25

F

FEAST 84
FGMRES(m)-Verfahren 80
Fixpunkt 119
Fixpunktiterationsverfahren 69

G

Gauß-Seidel-Verfahren 70
Gauß-Seidel-Vorkonditionierung 93
GMRES-Verfahren 20
Gram-Matrix 45
Gram-Vektor 61
Graph 9
– gerichteter 9

H

Hardware-orientierte Numerik . . 12, 17
Hessenberg-Matrix 118
– obere 118
– untere 118

J

Jacobi-Verfahren 69
Jacobi-Vorkonditionierung 78

K

Kante 9
Kernel 23
kleinste Fehlerquadrate 82
Knoten 9

Konditionszahl 117
Konsistenz 64
Kontraktion 119
Konvergenzrate 117
Krylow-Unterraum 19
Krylow-Unterraum-Methode 20

L

Latenz 16
Level eines Knotens 27
lineare Hülle 8
Linksvorkonditionierung 50
LP-CA-CG-Verfahren 58
LP-CG3-Verfahren 52

M

Matrixpotenzen-Kernel 24
Moore-Penrose-Inverse 82, 118

N

Newton-Polynome 65
nichtnegativ 118
NUMA 15

P

PA0-Algorithmus 35
PA1-Algorithmus 35
PA2-Algorithmus 36
Parallelität
– fein-granulare 26
– grob-granulare 25
Pipelined-CG-Verfahren 22
Poisson-Problem 84
positiv 118
Pseudoinverse 82

R

Rechtsvorkonditionierung 51

S

Satz von Perron-Frobenius 119
Shared Memory 25
Spann 8
SPD-Matrix 20
Spektralkonditionszahl 117
Spektralnorm 117
Spektralradius 70, 117
Spektrum 70, 117
SpMV-Kernel 23
Stabilität 64

T

Tschebyschow-Polynome 65
Two-Level-Numbering 87

U

Übernahmestrategie 77
UMA 15

V

Verf. der konjugierten Gradienten .. 20
Vorkonditionierung 51
– beidseitige 51
– explizite 51
– flexible 80
– implizite 51

Eidesstattliche Versicherung

Klinger, Marcel | 116088

Name, Vorname | Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende ~~Bachelorarbeit~~/Masterarbeit* mit dem Titel

Kommunikationsvermeidende und asynchrone Verfahren zur Lösung dünnbesetzter linearer Gleichungssysteme auf modernen Höchstleistungsrechnern

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, 21.08.2012 | _____

Ort, Datum | Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Dortmund, 21.08.2012 | _____

Ort, Datum | Unterschrift