technische universität dortmund

fakultät für mathematik

Masterarbeit

Konzepte für fehlertolerante Mehrgitterverfahren

am Lehrstuhl für Angewandte Mathematik und Numerik

der Fakultät für Mathematik, TU Dortmund

vorgelegt von

Mirco Altenbernd

betreut durch

Prof. Dr. Dominik Göddeke

und

Prof. Dr. Stefan Turek

30. Juni 2015

Inhaltsverzeichnis

1 Einleitung

Aktuelle Forschungen im Bereich des Hochleistungsrechnens befassen sich intensiv mit zukünftigen exascale-Systemen [8]. Diese zeichnen sich insbesondere dadurch aus, dass der Grad der Parallelität immer weiter steigt. Es ist zu erwarten, dass bis zu 100 000 Prozessoren Verwendung finden [17]. Dies ist einhergehend mit einer immer größer werdenden Fehlerwahrscheinlichkeit im Gesamtsystem, da die Zuverlässigkeit der einzelnen Komponenten vermutlich nicht steigt, sondern eher noch weiter sinkt. Es ist davon auszugehen, dass Fehler in komplexen Simulationen damit zur Regel werden und in den Algorithmen selbst abgefangen werden müssen. Sonst kann unter Umständen jeglicher Fortschritt beim Auftreten eines Fehlers verloren gehen. Dabei wird üblicherweise zwischen verschiedenen Fehlertypen unterschieden, welche wir in Kapitel 2.4 genauer beschreiben.

In dieser Arbeit untersuchen wir Konzepte mit denen sich verschiedene Varianten des geometrischen Mehrgitterverfahrens, gegenüber Fehlern in numerischen Operation, stabilisieren lassen. Wir betrachten das aus der Welt der dichtbesetzten Strukturen bekannten Konzept der Prüfsummen sowie einen konstruierten Mechanismus, der es ermöglicht die Ergebnisse der Glättungsphase des Verfahrens auf Tauglichkeit zu überprüfen. Die numerische Effizienz untersuchen wir dabei anhand eines konformen Finite Elemente-Ansatzes und zweidimensionalen Diffusions-Problemen, mit zum Teil auch konvektiven und reaktiven Anteilen, wobei eine Implementierung innerhalb von `FEAST` [27] durchgeführt wird.

Es ist wünschenswert, dass solche im exascale-Bereich verwendete Algorithmen Fehler erkennen und reparieren, ohne dabei im fehlerfreien Fall erheblichen Mehraufwand zu produzieren. Bisherige Analysen wurden hauptsächlich für CG-artige Verfahren durchgeführt. Für diese Verfahren konnte gezeigt werden, dass sie sogar selbststabilisierend sind [21], d.h. dass sie sich beim Auftreten von Fehlern innerhalb weniger Iterationen wieder korrigieren und weiterhin zur ursprünglichen Lösung konvergieren. Dahingegen können z.B. beim GMRES-Verfahren Fehler im Arnoldi-Algorithmus zur Divergenz oder einer falschen Lösung führen [9].

Klassische Stabilisierungsmethoden wie globales Checkpointing verursachen neben zusätzlichem Speicherbedarf einen Bedarf an Kommunikation, der, wie in Kapitel 2.3 weiter beschrieben, nicht zu vernachlässigen ist. Außerdem entsteht durch dieses Konzept auch im fehlerfreien Fall ein entsprechender Mehraufwand. Diese Nachteile überwiegen den Vorteil, entsprechende Methoden ohne großen Aufwand auf ein weites Spektrum an

Verfahren anwenden zu können. Es ist erforderlich für verschiedene Verfahren effizientere eigene Stabilisierungsmethoden zu entwickeln.

Im Bereich der Mehrgitterverfahren liegen bisher relativ wenige Resultate bzgl. dieser Problematik vor. In bisherigen Arbeiten wurde, unter Beteiligung des Autors, bereits die Auswirkung von Störungen auf das Konvergenzverhalten untersucht und dabei festgestellt, dass das Verfahren robust ist und sich beim Auftreten von Fehlern i.d.R. nach kurzer Zeit wieder stabilisiert [13]. Der dort demonstrierte Ansatz des minimalen Checkpointings erlaubt es mit wenig Kommunikationsaufwand Backups zu erstellen, die im Falle von Knotenausfällen eine lokale Restauration erlauben, ohne dass eine erhebliche Verlangsamung der Konvergenz eintritt.

Das Ziel der vorliegenden Arbeit ist es unsere Untersuchungen und Konzepte insbesondere auf die Situation der sogenannten *soft faults* [10] auszuweiten. Diese Fehlerart zeichnet sich dadurch aus, dass sie nicht den Ausfall von Rechenkomponenten oder den unmittelbaren Abbruch eines Verfahrens bewirkt, sondern nur das Ergebnis einzelner Operationen verfälscht, wodurch die Konvergenz verschlechtert werden kann. Das Ergebnis der Multiplikation von zwei Zahlen mit ähnlicher Binärdarstellung, wie z.B.

$$\begin{aligned} 1001 \times 0011 &= 011011 \quad &\widehat{=} \quad & 9 \times 3 = 27 \\ 1101 \times 0011 &= 100111 \quad &\widehat{=} \quad & 13 \times 3 = 39, \end{aligned}$$

kann sich beim Auftreten eines solchen Fehlers bereits stark unterscheiden.

Schwerpunkt Der Schwerpunkt in früheren Analysen konzentrierte sich auf das Szenario des Knotenausfalls, wohingegen wir in dieser Arbeit, wie angesprochen, insbesondere *soft faults* betrachten. Es soll vor allem die Auswirkung von Fehlern innerhalb von numerischen Operationen minimiert bzw. eliminiert werden.

Ziel ist es dabei, wie bereits erwähnt, das Mehrgitterverfahren, mit Hilfe der beiden vorgestellten Komponenten, fehlertoleranter zu machen und insbesondere die selbststabilisierende Fähigkeit weiter zu verbessern. Dabei konzentrieren wir uns bei der Analyse auf den seriellen Fall mit dem Argument, dass die Verwendung von Gebietszerlegungsverfahren (z.B. durch den ScaRC-Ansatz [16]) auch im Bereich des Hochleistungsrechnens dazu führt, dass viele kleine lokale Teilprobleme gelöst werden müssen. Bei der Gebietszerlegung wird das Rechengebiet üblicherweise in Teilgebiete zerlegt, die nur in einer kleinen Grenzschicht zwischen den Iterationen kommunizieren. Auf jedem dieser Teilgebiete arbeitet dann ein eigener Löser. Eine Stabilisierung dieser lokalen Löser führt damit unmittelbar zu einer Verbesserung des Gesamtverfahrens.

Aufbau Nach einer kurzen Erläuterung essentieller Grundlagen und der Beschreibung von Fehlertypen sowie des simulierten Fehlerszenarios in Kapitel 2, untersuchen wir in Kapitel 3 die Verwendbarkeit von Prüfsummen in der dünnbesetzten linearen Algebra und betrachten auch spezielle Varianten. Darauf folgt in Kapitel 4 die Konstruktion eines Mechanismus zur Erzeugung einer algorithmenbasierten Fehlertoleranz im Glättungsprozess des Mehrgitterverfahrens, welche es erlaubt diesen Bereich des Verfahrens fehlertoleranter zu gestalten.

Als Abschluss stellen wir in Kapitel 5 ein mögliches vollständig fehlertolerantes Mehrgitterverfahren vor, welches aus einer Kombination der zuvor analysierten Komponenten konstruiert wird. Ziel ist es dabei durch einen möglichst geringen Mehraufwand zusätzliche Toleranz zu erhalten, um auch den fehlerfreien Fall nicht zu stark zu beeinflussen.

2 Grundlegende Methoden und aktuelle Entwicklungen

Zu Beginn geben wir einen kurzen Überblick über einige Grundlagen, die im späteren Verlauf benötigt werden. Wir stellen die bei der späteren Analyse verwendete Finite Elemente-Methode (FEM) und die hieraus resultierenden Matrixstrukturen vor, welche bei der Verwendung von Prüfsummen (vgl. Kapitel 3) relevant sind. Darüber hinaus beschreiben wir die Funktionsweise des später modifizierten Mehrgitterverfahrens. Außerdem erläutern wir die aktuellen Entwicklungen im Bereich des Hochleistungsrechnens und die daraus resultierenden Erwartungen bzgl. der Fehleranfälligkeit numerischer Methoden im Hinblick auf exascale-Systeme sowie die Rolle die Verfahren mit algorithmenbasierter Fehlertoleranz in diesem Bereich spielen. Abschließend beschreiben wir gängige Fehlertypen und formulieren das von uns simulierte Fehlerszenario, das gewisse Aspekte der Entwicklung modellieren soll.

2.1 Finite Elemente Methode

Die Finite Elemente-Methode ist neben der Methode der Finite Differenzen eine der am häufigsten verwendeten Methoden zur Diskretisierung partieller Differentialgleichungen. Wir geben in diesem Kapitel einen kurzen Abriss über die Vorgehensweise und verweisen für Details auf die Arbeit von Braess [4].

Für die exemplarisch betrachtete elliptische partielle Differentialgleichung

$$\begin{aligned} Lu := -\sum_{i,k=1}^{n} \partial_i(a_{ik}\partial_k u) = f \quad &\text{in } \Omega \subset \mathbb{R}^n \\ u = g \qquad &\text{auf } \partial\Omega \end{aligned} \tag{2.1}$$

mit $f, g \in L_2(\Omega)$ liefert die Finite Elemente-Methode, unter Verwendung von Multiplikation, (partieller) Integration und dem Übergang zu diskreten Ansatz- und Testräumen, ein diskretes Problem, welches mit numerischen Methoden, wie dem Mehrgitterverfahren, gelöst werden kann.

Das kontinuierliche Ausgangsproblem (2.1) erfordert eine *klassische Lösungen* $u \in C^2(\Omega) \cap C^0(\bar{\Omega})$. Die Finite Elemente-Methode dagegen sucht eine Approximation der als *schwache Lösung* bezeichneten Funktion $u \in H_0^1(\Omega)$ (im Falle von homogen Dirichlet-Randdaten, d.h. $g \equiv 0$), die die variationelle Formulierung

$$a(u, v) =< f, v > \qquad \forall v \in H_0^1(\Omega)$$

mit

$$\begin{aligned} a(u, v) &:= \int_\Omega \sum_{i,k=1}^{n} a_{ik} \partial_i u \partial_k v \, \mathrm{d}x \\ < f, v > &:= \int_\Omega f v \, \mathrm{d}x. \end{aligned}$$

des Problems erfüllt.

Die Diskretisierung des Gebietes, d.h. eine Überlagerung von Ω durch ein diskretes Gitter Ω_h und das damit verbundene Ersetzen des kontinuierlichen Ansatz- bzw. Testraums $H_0^1(\Omega)$ durch einen endlichen Teilraum $V_h = \{\varphi_1, \ldots, \varphi_N\} \subset H_0^1(\Omega)$ liefert dann das diskrete Problem. Im einfachsten Fall bestehen diese Räume aus polynomiellen Funktionen endlichen Grades. Dies führt zu Lösungen, die im konformen Fall zwar weiterhin stetig aber nicht mehr unbedingt differenzierbar sein müssen.

Bei konformen Finite Elementen haben die verwendeten Basisfunktionen, die den Raum V_h aufspannen, dabei die Eigenschaft, jeweils an einem Freiheitsgrad den Wert Eins und an den übrigen den Wert Null zu haben. Mit anderen Worten bedeutet dies, dass

$$\sum_{i=1}^{N} \varphi_i = 1 \qquad \text{und} \qquad \varphi_i(x_j) = \delta_{ij},$$

wenn x_j die Koordinate des zu j gehörenden Freiheitsgrades ist. Insbesondere ist das Gebiet, auf dem die einzelnen Basisfunktionen ungleich Null sind, beschränkt. Eine Basisfunktion ist dabei i.d.R. auf ein bis vier Zellen von Null verschieden. Dieses Gebiet wird als Träger der Funktion bezeichnet und ist wie in Abbildung 2.1 (am Beispiel von Q_2-Finite Elementen) dargestellt von der Art, d.h. der Position, des zugeordneten Freiheitsgrades abhängig. Zwei Basisfunktionen φ_i und φ_j mit sich überschneidendem Träger zeichnen sich dabei insbesondere dadurch aus, dass

$$a(\varphi_i, \varphi_j) \neq 0.$$

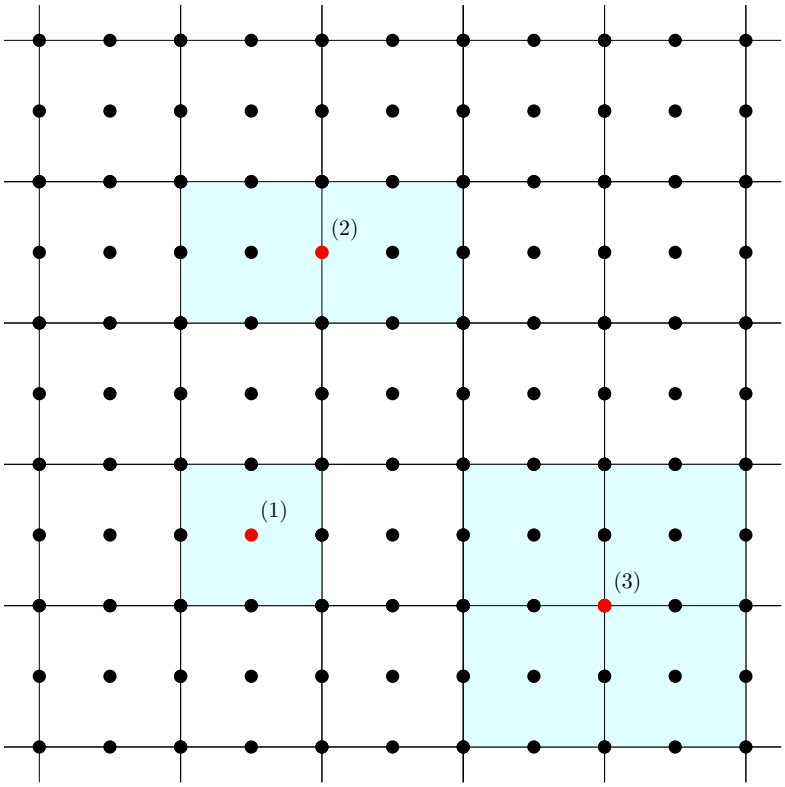

Abbildung 2.1: Grafische Darstellung der Träger (cyanfarbige Flächen) von Freiheitsgraden im Inneren (1), auf Kanten (2) bzw. auf Ecken (3) am Beispiel von konformen Finite Elementen zweiter Ordnung.

Die Finite Elemente-Methode sucht nun die diskrete Lösung u_h, die das diskrete Analogon der variationellen Formulierung

$$a_h(u_h, \varphi_j) =< f, \varphi_j >_h \qquad \forall j \in \{1, \ldots, N\}$$

mit

$$a_h(u_h, \varphi_j) := \int_{\Omega_h} \sum_{i,k=1}^{n} a_{ik} \partial_i u_h \partial_k \varphi_j \, \mathrm{d}x$$

$$< f, \varphi_j >_h := \int_{\Omega_h} f \varphi_j \, \mathrm{d}x.$$

erfüllt. Dabei lässt sich diese Lösung mittels der Elemente des Ansatzraumes V_h durch $u_h = \sum_{i=1}^{N} u_i \varphi_i$ darstellen. Mit den Eigenschaften des Integrals, und unter Einbeziehung der Randbedingungen, kann dies dann in ein reguläres lineares Gleichungssystem

$$A\mathbf{u} = b$$

mit der Matrix

$$A = \Big(a_h(\varphi_i, \varphi_j)\Big)_{i,j=1,\ldots,N},$$

sowie den Vektoren

$$\mathbf{u} = \Big(u_i\Big)_{i=1,\ldots,N} \quad \text{und} \quad b = \Big(< f, \varphi_j >_h \Big)_{j=1\ldots,N}$$

überführt werden.

Da die Träger der Basisfunktionen klein sind, ist die aus dem diskrete Operator entstehende Matrix A entsprechend dünnbesetzt. Eine Kopplung zwischen zwei Freiheitsgraden besteht nur, wenn der Schnitt der Träger nichtleer ist. Dies führt dazu, dass entsprechende Einträge in der Matrix ungleich Null sind. Dabei unterscheidet sich die Dichte der Besetzungen, d.h. die Anzahl der Einträge pro Matrixzeile, je nach Wahl des Ansatzraumes und ist insbesondere von der Ordnung des Ansatzraumes abhängig, wie wir im folgenden Abschnitt 2.1.1 näher erläutern werden. Im Zusammenhang mit Aufwandsschätzungen spielen diese Strukturen der resultierenden Matrizen eine entscheidende Rolle.

Für andersartige partielle Differentialgleichungen und Nicht-Dirichlet Randbedingungen ist ein ähnliches Vorgehen möglich. Für weitere Details verweisen wir erneut auf entsprechende Literatur [4], da diese für die durchgeführten Analysen keine entscheidende Rolle spielen.

2.1.1 Besetzungsstruktur einer Finite Elemente-Matrix

Die Matrizen eines Finite Elemente-Verfahrens sind, wie bereits angesprochen, dünnbesetzt. Für konforme Finite Elemente p-ter Ordnung (Q_p) lässt sich abhängig von der Ordnung eine durchschnittliche Anzahl an Nicht-Null-Einträgen pro Zeile angeben.

Wir beschränken uns auf einen zweidimensionalen Ansatz und betrachten zur Vereinfachung ein unendliches zweidimensionales Gitter. Dies erspart uns die gesonderte Betrachtung von Randzeilen. Somit erhalten wir eine obere Abschätzung für die durchschnittliche Anzahl M_p an Elementen pro Matrixzeile, da das Randgebiet im Regelfall im Vergleich zum Inneren des Gebiet nur einen kleinen Anteil ausmacht und zugehörige Zeilen weniger Komponenten aufweisen.

Entscheidend für die durchschnittliche Anzahl an Einträgen pro Matrixzeile ist das Verhältnis zwischen der Anzahl an Elementen und der Freiheitsgrade auf Ecken, Kanten bzw. im Inneren von Elementen. In Abbildung 2.2 ist diese Zuordnung exemplarisch für Q_2-Finite Elemente dargestellt. Es lassen sich je eine Ecke und zwei Kanten einem Element zuordnen. Die Freiheitsgrade im Inneren sind darüber hinaus trivial dem entsprechenden Element zugeordnet. Diese verschiedenen Typen von Freiheitsgraden haben unterschiedlich große Träger (vgl. Abbildung 2.1) und treten in unterschiedlicher Häufigkeit auf.

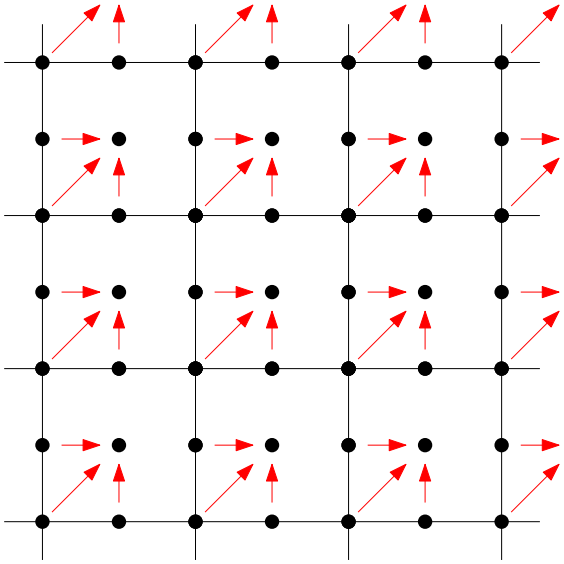

Abbildung 2.2: Zuordnung der Freiheitsgrade zu den Elementen am Beispiel eines konformen Finite Elemente-Ansatzes zweiter Ordnung, wobei die Freiheitsgrade in der Mitte der Elemente trivial dem Element selbst zuzuordnen sind.

Jedem Element lässt sich, unabhängig von der Ordnung p des Ansatzes, genau eine Ecke zuordnen und somit ein entsprechender Freiheitsgrad. Diese Freiheitsgrade haben den größtmöglichen Träger, da sie auf vier Elementen ungleich Null sind. Der Träger besteht somit aus den $(2p+1)^2$ Freiheitsgraden der anliegenden Elemente. Ab einem Finite Elemente-Ansatz der Ordnung 2 kommen zusätzlich Freiheitsgrade auf Kanten und im Inneren der Elemente hinzu. Jeder Zelle im Gitter lassen sich zwei Kanten zuordnen. Damit hat jedes Element $2(p-1)$ Freiheitsgrade auf Kanten. Der Träger dieser Freiheitsgrade besteht jeweils aus den zwei anliegenden Elementen. Demzufolge haben zugehörige Matrixzeilen $(2p+1)(p+1)$ Nicht-Null-Einträge. Schließlich hat ein Element $(p-1)^2$ Freiheitsgrade im Inneren deren Träger sich auf das Element selbst beschränkt und somit aus $(p+1)^2$ Freiheitsgraden besteht. Eine zusammenfassende Auflistung der Eigenschaften findet sich in Tabelle 2.1.

Position	Häufigkeit pro Element	Größe des Trägers
Ecke	1	$(2p+1)^2$
Kante	$2(p-1)$	$(2p+1)(p+1)$
Innen	$(p-1)^2$	$(p+1)^2$

Tabelle 2.1: Häufigkeit der unterschiedlichen Typen von Freiheitsgraden pro Element und Größe des Trägers der mit dem Freiheitsgrad verbundenen Funktion in Abhängigkeit von der Ordnung p des konformen Finite Elemente-Ansatzraumes.

Die durchschnittliche Anzahl M_p an Einträgen pro Matrixzeile bei der Verwendung von Finite Elementen p-ter Ordnung ergibt sich damit zu

$$\begin{aligned} M_p &= \frac{(2p+1)^2 + 2(p-1)(2p+1)(p+1) + (p-1)^2(p+1)^2}{1 + 2(p-1) + (p-1)^2} \\ &= \frac{p^4 + 4p^3 + 4p^2}{p^2} \\ &= (p+2)^2. \end{aligned}$$

Mit dieser Näherung ist es nun möglich den Aufwand von Matrix-Vektor-Operationen abhängig von dem verwendeten Grad der Finite Elemente-Diskretisierung abzuschätzen.

2.2 Das Mehrgitterverfahren

Das Mehrgitterverfahren [14, 26] ist ein oft verwendetes Verfahren zum Lösen dünnbesetzter linearer Gleichungssystemen und findet insbesondere im Zusammenhang mit der Finite Elemente-Methode und Poisson-artigen Differentialgleichungen seine Anwendung. Im folgenden Kapitel wollen wir kurz das Verfahren beschreiben und auf einige wichtige Aspekte und Eigenschaften eingehen. Des Weiteren stellen wir im Vergleich zum klassischen Mehrgitterverfahren, den vollständig approximierenden Ansatz (*engl.:* **F**ull **A**pproximation **S**cheme, FAS) vor. Dieser generiert zwar einen geringen Mehraufwand, bietet aber einige zusätzliche Eigenschaften, die wir später im Bereich der algorithmenbasierten Fehlertoleranz verwenden möchten.

Die Grundidee des Verfahrens besteht aus der Kombination von zwei unterschiedlichen Komponenten, die als eigenständige Löser schlecht geeignet sind, aber in Zusammenarbeit eine der stärksten Methoden zum Lösen von Poisson-artigen Problem in der numerischen Mathematik ergeben. Es ist möglich eine von der Gitterweite unabhängige Konvergenzgeschwindigkeit zu erzielen.

Wir beschränken uns im Folgenden auf das geometrische Mehrgitterverfahren und die Verwendung von Finite Elementen.

2.2.1 Klassisches Mehrgitterverfahren

Grundlegende Voraussetzung für das geometrische Mehrgitterverfahren ist eine Gitterhierarchie sowie Operatoren die den Transfer von Größen zwischen diesen Gittern ermöglichen. Diese Gitterhierarchie beginnt mit einem Grobgitter (Nummerierung: $k = 0$). Durch sukzessive Verfeinerung werden dann Gitter mit mehr Zellen generiert, bis schließlich nach L Verfeinerungen ein Feingitter ($k = L$), auf dem die Lösung gesucht wird, erzeugt ist. Die Gitterhierarchie besteht somit aus $L + 1$ Gittern. Es ist notwendig, dass auf jedem Gitter der Differentialoperator A_k assembliert wird. Eine Assemblierung der rechten Seite b ist nur auf dem feinsten Gitter erforderlich.

Zusätzlich zu den Gittern werden geeignete Transferoperatoren benötigt. Im Allgemeinen gibt es dazu mehrere Möglichkeiten. Wir verwenden im Zusammenhang mit konformen Finite Elementen für den Gittertransfer Interpolationen entsprechender Ordnung, d.h. bilineare Interpolation für Q_1 bzw. biquadratische Interpolation für Q_2. Wir bezeichnen den Transfer von einem Gitter $k-1$ auf ein feineres Gitter k (*Prolongation*) dabei durch $\mathbf{P}_{k-1}^{k}$ und den Transfer auf ein gröberes Gitter (*Restriktion*) durch $\mathbf{R}_{k}^{k-1}$. Im beschriebenen Fall gilt zusätzlich der Zusammenhang $(\mathbf{R}_{k}^{k-1})^T = \mathbf{P}_{k-1}^{k}$.

Neben der Gitterhierarchie und den zugehörigen Komponenten spielt der *Glätter* $\mathcal{S}$ eine entscheidende Rolle. Dabei handelt es sich i.d.R. um ein einfaches gedämpftes Defektkorrekturverfahren und es werden nur wenige Iterationen (*Glättungsschritte*) durchgeführt. Ziel hierbei ist es bestimmte Fehlerfrequenzen zu dämpfen. Es können durchaus auch komplexere Verfahren zur Glättung verwendet werden [18]. Entscheidend ist, dass diese die *Glättungseigenschaft* (*engl.*: smoothing property) besitzen.

Eingeleitet wird das Verfahren durch ν-maliges Vorglätten, d.h. die ν-malige Anwendung des Glätters

$$u^{(\nu)} = \mathcal{S}^{\nu}\left(u^{(0)}, b\right)$$

auf die initialisierte Näherungslösung $u^{(0)}$ und die rechte Seite b. Die Initialisierung kann dabei auf unterschiedliche Weisen erfolgen. Danach folgt auf die Berechnung des aktuellen Residuums

$$r_k = b - A_k u^{(\nu)}$$

der Transfer

$$r_{k-1} = \mathbf{R}_{k}^{k-1} r_k$$

auf das nächstgröbere Gitter. Nun wird auf diesem Gitter das Problem

$$A_{k-1} v = r_{k-1}$$

gelöst. Die Bestimmung der Lösung v erfolgt dabei abhängig vom Gitterlevel. Auf dem Grobgitter erfolgt die Berechnung direkt, wohingegen auf den anderen Gittern die Lösung näherungsweise, durch das erneute γ-fache Aufrufen eines Mehrgitterzyklus, berechnet wird. Bei dieser näherungsweisen Bestimmung erfolgt die Initialisierung des Korrekturvektors natürlicherweise durch den Nullvektor. Die Wahl von γ bestimmt die Art des Mehrgitterzyklus. Mit Hilfe des so bestimmten Korrekturvektors v lässt sich im Anschluss die Näherungslösung, nach der Prolongation auf das feinere Gitter, korrigieren

$$u^{(\nu+1)} = u^{(\nu)} + \mathbf{P}_{k-1}^{k} v.$$

Dieser Korrektur folgt, zur Eliminierung der neu entstandenen hohen Fehlerfrequenzen, eine μ-fache Anwendung des Glätters

$$u = \mathcal{S}^{\mu}\left(u^{(\nu+1)}, b\right).$$

Diese Phase wird als Nachglättung bezeichnet. Durch erneute Aufrufe des Verfahrens, mit der berechneten Lösung u als Startlösung, lässt sich eine weitere Verbesserung erzielen. Dies wird üblicherweise bis zum Erreichen eines Abbruchkriteriums, wie z.B. einer relativen oder absoluten Residuumsreduktion, wiederholt.

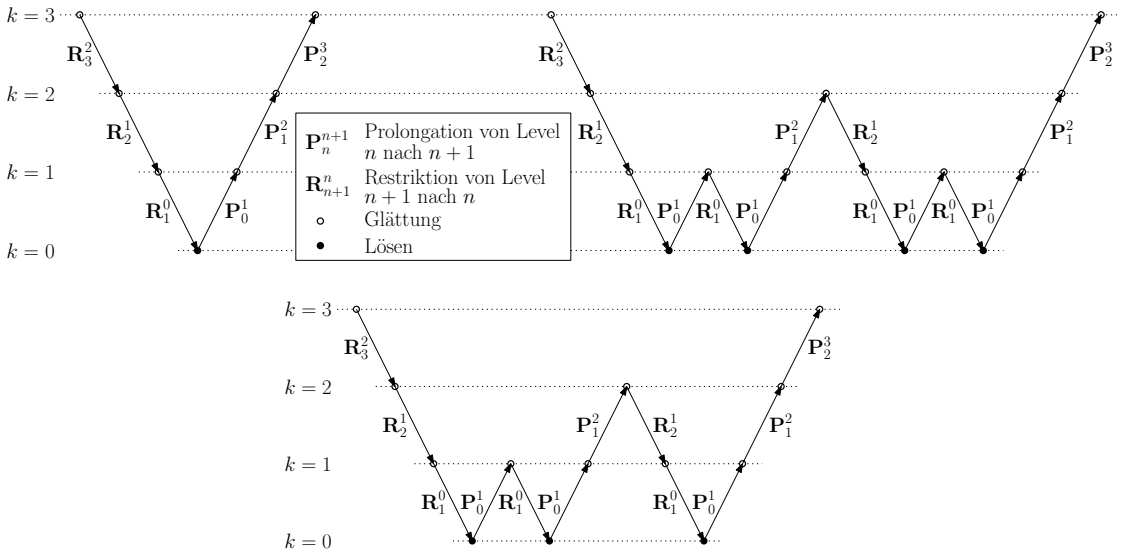

Abbildung 2.3: Grafische Darstellung des Durchlaufens der Gitterhierarchie bei einem Mehrgitterverfahren mit 4 Gitterleveln und V-, W- bzw. F-Zyklus.

Das Durchlaufen der Gitterhierarchie kann auf unterschiedliche Arten erfolgen. Üblicherweise werden dabei sogenannte V-,W- oder F-Zyklen verwendet. Diese unterscheiden sich insbesondere darin, wie oft ein Grobgitterproblem gelöst wird, d.h. wie γ gewählt wird, wobei der F-Zyklus einen Spezialfall darstellt. Es entstehen dabei Schemata, wie sie in Abbildung 2.3 dargestellt sind. Innerhalb dieser Arbeit beschränken wir uns für numerische Analysen und Untersuchungen auf den einfachen V-Zyklus ($\gamma = 1$). In Algorithmus 2.1 ist der entsprechende Pseudocode des Mehrgitterverfahrens dargestellt.

Algorithmus 2.1: Klassischer V-Zyklus

Aufruf :
$\texttt{MG}(k, \mathcal{A}, b, u^{(0)})$
Parameter :
ν - Anzahl der Vorglättungsschritte
μ - Anzahl der Nachglättungsschritte
Eingabe :
k - Gitterlevel
$\mathcal{A}$ - Menge der Systemmatrizen $\mathcal{A} = \{A_0, \ldots, A_k\}$
b - rechte Seite auf Level k
$u^{(0)}$ - initiale Lösungsapproximation auf Level k
Ausgabe :
u - Lösungsapproximation des Systems $A_k u = b$

```
1  if k = 0 then
2  |  u = A_0^{-1} b                                  // Lösen des Grobgitterproblems
3  else
4  |  u^(ν) = S^ν (u^(0), b)                          // ν-faches Vorglätten
5  |  r_k = b − A_k u^(ν)                             // Berechnung des Residuums
6  |  r_{k−1} = R_k^{k−1} r_k                         // Restriktion des Residuums
7  |  v^(0) ≡ 0                                       // Initialisierung Korrekturwert
8  |  v = MG(k − 1, A, r_{k−1}, v^(0))                // rekursives Lösen des Grobgit-
   |                                                  //   terdefektproblems
9  |  u^(ν+1) = u^(ν) + P_{k−1}^k v                    // Grobgitterkorrektur
10 |  u = S^μ (u^(ν+1), b)                            // μ-faches Nachglätten
11 end
```

2.2.2 Vollständig approximierendes Mehrgitterverfahren

Der Ansatz des vollständig approximierenden Mehrgitterverfahrens (*engl.:* **F**ull **A**pproximation **S**cheme, FAS, siehe u. a. Briggs et al. [5]) wurde für das Lösen von nichtlinearen Differentialgleichungen entworfen. In diesem Fall ist auch der diskrete Operator A von der Lösung u abhängig, so dass im Allgemeinen die Gleichheit

$$A(u+e) = Au + Ae$$

nicht erfüllt ist. Diese Eigenschaft liegt jedoch dem Konzept des klassischen Mehrgitterverfahrens zugrunde, da dort die Umformung

$$\begin{aligned} & A(u+e) = f \\ \Leftrightarrow \qquad & Ae = f - Au \end{aligned}$$

bei der Berechnung der Grobgitterkorrekturen eine entscheidende Rolle spielt, denn auf jedem gröberen Gitter wird nur ein Defektproblem gelöst. Stattdessen wird beim FAS-Ansatz die Lösung v des Grobgitterproblems

$$\begin{aligned} A_{k-1}v &= r_{k-1} \\ &= \mathbf{R}_k^{k-1}(b - A_k u^{(\nu)}) + A_{k-1}\mathbf{I}_k^{k-1}u^{(\nu)} \end{aligned}$$

ermittelt und anschließend die vorgeglättete Näherungslösung $u^{(\nu)}$ mit der prolongierten Differenz zwischen v und der restringierten Näherungslösung $v^{(0)} = \mathbf{I}_k^{k-1}u^{(\nu)}$ korrigiert. Wichtig ist hierbei, dass der Operator $\mathbf{I}$, mit dem die Näherungslösung auf das gröbere Gitter projektiert wird, nicht mit dem Restriktionsoperator übereinstimmt. Stattdessen entspricht dieser der Übernahme von Funktionswerten in den gemeinsamen Punkten der beiden Gitter. Es handelt sich somit um eine natürliche Injektion.

Falls die zugrundeliegende Differentialgleichung linear ist, gilt für die Korrektur $v - v^{(0)}$

$$\begin{aligned} A_{k-1}(v - v^{(0)}) &= A_{k-1}v - A_{k-1}v^{(0)} \\ &= \mathbf{R}_k^{k-1}(b - A_k u^{(\nu)}) + A_{k-1}\mathbf{I}_k^{k-1}u^{(\nu)} - A_{k-1}v^{(0)} \\ &= \mathbf{R}_k^{k-1}(b - A_k u^{(\nu)}) + A_{k-1}\mathbf{I}_k^{k-1}u^{(\nu)} - A_{k-1}\mathbf{I}_k^{k-1}u^{(\nu)} \\ &= \mathbf{R}_k^{k-1}(b - A_k u^{(\nu)}) \end{aligned}$$

und somit lässt sich das Problem auf das klassischen Mehrgitterverfahren reduzieren. Die Verfahren sind in diesem Fall also bis auf Formulierung und Ausführung äquivalent. Es entsteht jedoch ein Mehraufwand aufgrund von zusätzlichen Berechnungen. Dafür existiert auf jedem Gitterlevel eine Approximation der Feingitteriterierten mit verringerter Auflösung. Dies werden wir uns im späteren Verlauf bei der algorithmenbasierten Fehlertoleranz zu nutzen machen (siehe Kapitel 4). Die schematische Darstellung des Verfahrens ist in Algorithmus 2.2 beschrieben.

Es ist anzumerken, dass die Näherungslösungen, abseits des Feingitters, keine Approximationen des eigentlichen Problems auf dem gröberen Gitter darstellen. Es sind viel mehr Approximationen des Feingitterproblems mit einer verringerten Auflösung [5, Kapitel 6].

Darüber hinaus ermöglicht uns dieses Verfahren eine einfache Implementierung des bereits vorgestellten Ansatzes des *minimalen Checkpointings* [13], da bereits komprimierte Näherungslösungen vorliegen.

Algorithmus 2.2: FAS V-Zyklus

Aufruf :
`FAS_MG(`$k, \mathcal{A}, b, u^{(0)}$`)`
Parameter :
ν - Anzahl der Vorglättungsschritte
μ - Anzahl der Nachglättungsschritte
Eingabe :
k - Gitterlevel
$\mathcal{A}$ - Menge der Systemmatrizen $\mathcal{A} = \{A_0, \ldots, A_k\}$
b - rechte Seite auf Level k
$u^{(0)}$ - initiale Lösungsapproximation auf Level k
Ausgabe :
u - Lösungsapproximation des Systems $A_k u = b$

1 **if** $k = 0$ **then**
2 | $u = A_0^{-1} b$ // Lösen des Grobgitterproblems
3 **else**
4 | $u^{(\nu)} = \mathcal{S}^{\nu}\left(u^{(0)}, b\right)$ // ν-faches Vorglätten
5 | $r_k = b - A_k u^{(\nu)}$ // Berechnung des Residuums
6 | $r_{k-1} = \mathbf{R}_k^{k-1} r_k$ // Restriktion des Residuums
7 | $v^{(0)} = \mathbf{I}_k^{k-1} u^{(\nu)}$ // Restriktion der Lösung
8 | $r_{k-1} = r_{k-1} + A_{k-1} v^{(0)}$ // Berechnung der rechten Seite
9 | $v =$ `FAS_MG(`$k-1, \mathcal{A}, r_{k-1}, v^{(0)}$`)` // rekursives Lösen des Grobgitterproblems
10 | $u^{(\nu+1)} = u^{(\nu)} + \mathbf{P}_{k-1}^{k}\left(v - v^{(0)}\right)$ // Grobgitterkorrektur
11 | $u = \mathcal{S}^{\mu}\left(u^{(\nu+1)}, b\right)$ // μ-faches Nachglätten
12 **end**

2.2.3 Aufwand beim Mehrgitterverfahren

Der numerische Aufwand des Mehrgitterverfahrens lässt sich durch die Operationen auf dem Feingitter abschätzen. Detaillierte Ergebnisse dazu finden sich u. a. in Trottenberg et al. [26]. Für unsere späteren Tests relevant ist die Abschätzung für den Aufwand des V-Zyklus bei zweidimensionalen Problemstellungen. Hier lässt sich der Gesamtaufwand des Verfahrens durch

$$W = \frac{4}{3}CN$$

abschätzen, wobei N die Anzahl der Unbekannten auf dem Feingitter darstellt und C die pro Komponente durchgeführten Operationen. Es ist somit möglich, den Gesamtaufwand des Verfahrens durch den Aufwand auf dem Feingitter abzuschätzen.

Wir interessieren uns bei späteren Analysen insbesondere für das in Algorithmus 2.2 dargestellte vollständig approximierende Mehrgitterverfahren, so dass wir uns im Folgenden auf diesen Ansatz beschränken.

Für unsere Betrachtung unterteilen wir das Verfahren des Weiteren in eine Glättungs- und eine Transferphase. Die Transferphase beinhaltet dabei alle Operationen, die außerhalb des Prozesses der Glättung durchgeführt werden. Wir verwenden für den Glätter exemplarisch eine Jacobi-vorkonditionierte Richardson-Iteration - eine der einfachsten Möglichkeiten. Diese wird durch

$$x^{(i+1)} = x^{(i)} + \omega D^{-1}(b - Ax^{(i)})$$

beschrieben, wobei die Matrix D aus den Diagonaleinträgen der Matrix A besteht und ω ein Dämpfungsparameter ist. Wie in Abschnitt 2.1.1 beschrieben, enthält eine FEM-Systemmatrix pro Zeile durchschnittlich $(p+2)^2$ Einträge, wobei p der Ordnung des konformen Ansatzraumes entspricht. Damit werden bei der Anwendung der Systemmatrix auf einen Vektor $2(p+2)^2$ Operationen (Addition und Multiplikation) pro Zeile durchgeführt. Zusätzlich erfordert ein vollständiger Glättungsschritt zwei weitere Additionen und eine Multiplikation pro Komponente. Die Skalierung mit dem Dämpfungsparameter ist vernachlässigbar, da dieser in der Praxis bereits in die Diagonalmatrix D^{-1} integriert ist. Damit werden in der Glättungsphase

$$\left(2(p+2)^2 + 3\right) N$$

numerische Operationen pro Glättungsschritt durchgeführt.

Für eine detaillierte Betrachtung der Transferoperationen ist es notwendig die Besetzungsstruktur der Interpolations- und Restriktionsoperatoren $\mathbf{P}, \mathbf{R}$ und $\mathbf{I}$ zu analysieren. Der Injektionsoperator $\mathbf{I}$ entspricht, im betrachteten zwei-dimensionalen Fall bei regelmäßiger Verfeinerung, der einfachen Übernahme von jedem vierten Wert. Es entsteht eine Rechteck-Matrix, wobei in einem Viertel der Spalten eine Eins auftritt. Somit werden bei der Anwendung $\frac{1}{4}N$ Operationen ausgeführt.

Bei den Prolongations- und Restriktionsoperatoren gilt im vorliegenden Fall, wie in Kapitel 2.2 beschrieben, der Zusammenhang $\mathbf{P}^T = \mathbf{R}$, somit reicht die Betrachtung einer der Operatoren. Die durchschnittliche Anzahl der Elemente pro Zeile beläuft sich bei einer bilinearen Interpolation auf 3 und bei einer biquadratischen auf $3.\bar{6}$ Einträge. Die Bestimmung dieser Werte erfolgt ähnlich wie in Abschnitt 2.1.1. Dabei werden bei der Schätzung die Eigenschaften der konformen Finite Elemente ausgenutzt. Hier fallen die Knoten der unterschiedlichen Verfeinerungsstufen zusammen, so dass häufig nur eine einfache Übernahme der Werte notwendig ist. Der Einfachheit halber verwenden wir in beiden Fällen eine obere Abschätzung von 4 Einträgen pro Zeile.

Damit lassen sich auch die für die Transferphase benötigten numerischen Operationen berechnen. Die Berechnung des Residuums benötigt weiterhin $(2(p+2)^2+1)N$ Operationen. Das Berechnen der rechten Seite des Grobgitterproblems in Zeile 8 von Algorithmus 2.2 wird bereits auf dem nächstgröberen Gitter durchgeführt und erfolgt daher nur für $\frac{N}{4}$ Komponenten. Die Anwendung des Injektionsoperators $\mathbf{I}$ benötigt ebenfalls $\frac{N}{4}$ numerische Operationen. Des Weiteren werden in Zeile 6 und 10 die Transferoperatoren angewendet. Diese benötigen jeweils $4N$ Multiplikationen und Additionen. Darüberhinaus wird in Zeile 10 noch eine Vektoraddition auf dem aktuellen und dem nächstgröberen Gitter durchgeführt. Dies ergibt somit einen Gesamtaufwand von

$$\left(17.5 + \frac{5}{4}(2(p+2)^2+1)\right) N$$

numerischen Operationen für die Transferphase.

Abbildung 2.4 stellt den relativen Anteil der Glättungsphase am Gesamtaufwand des Mehrgitterverfahrens dar. Es ist ersichtlich, dass bei zunehmender Anzahl an Glättungsschritten der Aufwand in der Transferphase vernachlässigbar klein wird. Bereits bei 8 Glättungsschritten (z.B. 4 Vor- und 4 Nachglättungen) beträgt der Aufwand des Glätters 80% des Gesamtaufwandes. Die Auswirkung der Wahl der Ordnung der verwendeten Finite Elemente ist dabei nur geringfügig.

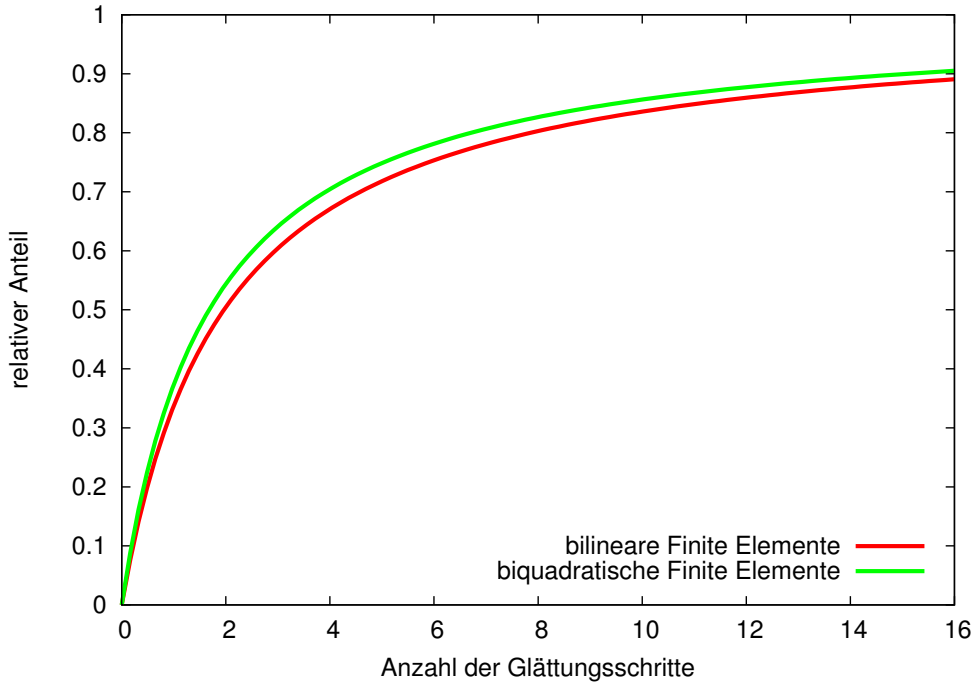

Abbildung 2.4: Relativer Anteil des Glätters am Gesamtaufwand des Mehrgitterverfahrens in Abhängigkeit der Summe der durchgeführten Glättungsschritte.

2.3 Entwicklungen im Bereich des Hochleistungsrechnens

Bisherigen Analysen zufolge ist es nicht zu erwarten, dass sich die Zuverlässigkeit von Prozessoren in naher Zukunft verbessert. Wahrscheinlicher ist sogar, dass aufgrund der verkleinerten Fertigungsprozesse [2, 7] und der damit u. a. verbunden höheren Anfälligkeit gegenüber Leckspannung, die Zuverlässigkeit sinkt. Durch diese bestenfalls gleichbleibende Zuverlässigkeit der Komponenten steigt die Fehleranfälligkeit der entstehenden Gesamtsysteme mit dem Grad der Parallelität. Bereits bei einem zeitlichen Abstand von 4 bis 50 Jahren (pro Prozessor) zwischen zwei Fehlern (*engl.:* **M**ean-**t**ime **b**etween **f**ailure, MTBF), wie ihn Kogge et al. [17] für die Zeit der exascale-Systeme erwarten, sinkt dieser für das Gesamtsystem schnell in den Bereich von wenigen Stunden ab. Dies hängt insbesondere damit zusammen, dass, wie Stearley et al. [25] demonstrieren, der MTBF des Gesamtsystems linear mit der Anzahl der Kerne zusammenhängt. Eine entsprechende Darstellung findet sich in Abbildung 2.5 für einen MTBF von 5 Jahren pro Prozessor und einer Anzahl von bis zu 200 000 Kernen. Bereits bei nur 20 000 Recheneinheiten würde der MTBF des Gesamtsystems bei unter zwei Stunden liegen.

Der erwartete Anstieg an Parallelität im Bereich des Hochleistungsrechnens erfordert demzufolge sowohl Umdenken in Bereichen der Soft- als auch Hardwareentwicklung [8, 17], denn damit verändert sich nicht nur die Struktur, sondern eben auch die Feh-

lerwahrscheinlichkeit der Systeme. Fehler innerhalb von einzelnen Simulationen werden zur Regel statt zur Ausnahme. Ziel ist es daher die Algorithmen so zu erweitern, dass sie selbstständig Fehler registrieren und sich gegebenenfalls korrigieren. Diese Art von Algorithmen bezeichnen P. Sao und R. Vuduc als selbststabilisierend [21].

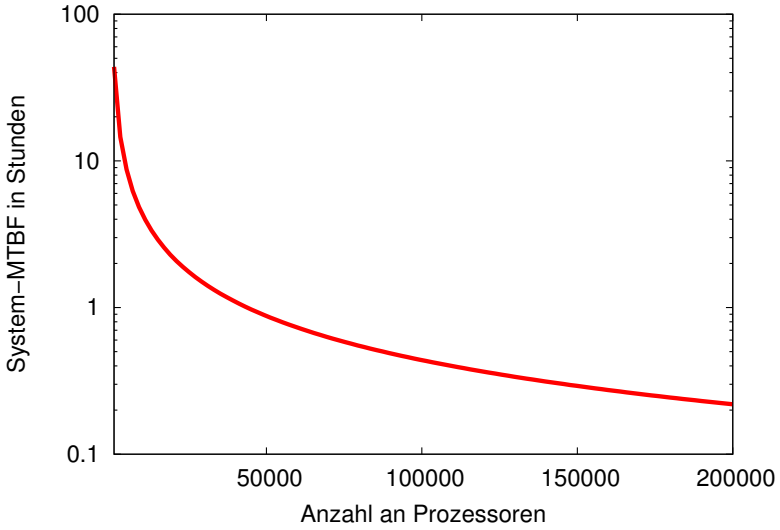

Abbildung 2.5: Veränderung der MTBF des Gesamtsystems (logarithmisch skaliert) bei zunehmender Anzahl an Prozessoren mit einem MTBF von 5 Jahren pro Prozessor.

Herkömmliche Ansätze Bisher verwendete Methoden, um die Fehlertoleranz von Verfahren zu erhöhen, sind i.d.R. globales Checkpointing (*engl.:* **C**heck**p**oint/**R**estart, CPR) oder Strategien die auf Mehrheitsentscheid basieren (z.B. ***T**riple **M**odular **R**edundancy*, TMR). Alle diese Verfahren reduzieren die zur Verfügung stehenden Ressourcen jedoch erheblich. Zusätzlich erfordern Verfahren die auf globalem Checkpointing basieren einen erheblichen Kommunikationsaufwand zwischen den Recheneinheiten, was zu weiteren Performanceverlusten führt. Insbesondere gilt dies im Bereich des Hochleistungsrechnens (*engl.*: **H**igh **P**erformance **C**omputing, HPC), da sich die Bandbreite wesentlich langsamer erhöht als die zur Verfügung stehende Rechenleistung und somit einen Flaschenhals bildet [8]. Ferreira et al. [11], Fiala et al. [12] und Stearley et al. [25] zeigen, dass die Verwendung von CPR in Kombination mit dem zunehmenden Anstieg an Recheneinheiten dazu führt, dass ein Großteil der Ressourcen und Laufzeit für das Erzeugen der Checkpoints bzw. das Restaurieren der verloren Daten benötigt wird. Beispielhafte Zahlen zu dieser Aussage liefern sie mit den in den Tabellen 2.2 und 2.3 angegebenen Werten. Es ist zu erkennen, dass bereits bei 100 000 Prozessoren, einem MTBF von 5 Jahren und einer Rechnung mit einer Laufzeit von sieben Tagen nur 35% der Zeit wirklich in die Berechnung der Lösung einfließen.

Diese Verfahren teilen darüber hinaus die Eigenschaft, dass sie Fehler erst auf einer sehr groben Ebene und unter Umständen zu einem sehr späten Zeitpunkt detektieren. Beim

#Prozessoren	Löser	Checkpointing	Neuberechnung	Neustart
100	96%	1%	3%	0%
1 000	92%	7%	1%	0%
10 000	75%	15%	6%	4%
100 000	35%	20%	10%	35%

Tabelle 2.2: Relative Verteilung der Laufzeit auf die einzelnen Teilbereiche des Algorithmus bei einem Prozess mit der Laufzeit von 168 Stunden und einem MTBF von 5 Jahren (pro Prozessor) bei ansteigender Anzahl an Prozessoren [11, 12, 25].

Laufzeit	MTBF	Löser	Checkpointing	Neuberechnung	Neustart
168 Std.	5 J.	35%	20%	10%	35%
700 Std.	5 J.	30%	18%	9%	43%
5 000 Std.	1 J.	5%	5%	5%	85%

Tabelle 2.3: Relative Verteilung der Laufzeit auf die einzelnen Teilbereiche des Algorithmus bei einem Prozess mit 100 000 Prozessoren und unterschiedlicher Laufzeit bzw. MTBF (pro Prozessor) [11, 12, 25].

globalen Checkpointing werden Fehler erst erkannt, wenn sie zu Abstürzen von Prozessen oder einem ungewöhnlichen Verhalten, wie etwa dem Auftreten von `NaN` oder `inf` innerhalb von Operationen, führen. In diesen Situationen kann nicht einmal garantiert werden, dass der letzte Checkpoint fehlerfreie Daten enthält. Auch bei der Verwendung von TMR geschieht der Abgleich der redundanten Rechnungen nur zu gewissen Zeitpunkten. Je nach Wahl dieser Abgleichzeitpunkte kann somit auch hier ein durchaus großer Verlust an Rechenzeit entstehen. Dieser Verlust ergibt sich zusätzlich zur Reduktion der verfügbaren Rechenleistung um den Faktor $\frac{2}{3}$, die durch die Redundanz verursacht wird.

Algorithmenbasierte Fehlertoleranz Eine Möglichkeit diese Probleme zu minimieren könnte die Erweiterung von Verfahren um eine sogenannte algorithmenbasierte Fehlertoleranz (*engl.:* **A**lgorithm **B**ased **F**ault-**T**olerance, ABFT, [3, 15]) sein. Ziel hierbei ist es, grundlegende Methoden der Gesamtverfahren durch inhärente Eigenschaften zu stabilisieren. Beispielsweise können Matrix-Vektor-Operationen, wie später in Kapitel 3 näher erläutert, durch Prüfsummen abgesichert werden, so dass das Resultat mit hoher Wahrscheinlichkeit richtig ist. Sollte bereits auf dieser Ebene die Korrektheit von Operationen sichergestellt werden, so sind Methoden wie CPR oder TMR weniger notwendig.

Bei der Konstruktion von Verfahren mit algorithmenbasierter Fehlertoleranz sind jedoch auch Eigenschaften der Hardware zu beachten. Li et al. [20] zeigen, dass die parallele Verwendung von ECC-Mechanismen (*engl.:* **E**rror-**c**orrecting **c**ode), einem Speicherschutz auf Hardwareebene, und algorithmenbasierter Fehlertoleranz zu einem übermäßigen Schutz führen kann. Dies gilt es zu vermeiden. Dabei ist es das Ziel, die ECC-Mechanismen auf ein Minimum zu reduzieren. Dies kann insofern ein Vorteil sein, da es im Bereich des Hochleistungsrechnens ein Ziel bleibt die sogenannte *Power-Wall*, ein Energielimit von 20 Megawatt, einzuhalten und entsprechende ECC-Verfahren einen nicht

vernachlässigbaren Energiebedarf haben [19].

Neben dem nicht vernachlässigbaren Bedarf an Energie weisen ECC-artige Schutzmechanismen auch andere Schwachstellen auf. Sie garantieren beispielsweise keinesfalls die Richtigkeit von Operationen innerhalb der ALU (*engl.:* **A**rithmetic **L**ogic **U**nit). Elliott et al. [10] zeigen, welche Auswirkungen bereits einzelne Bitflips innerhalb einer IEEE-Gleitkommaoperation haben können, wie auch das Beispiel in der Einleitung demonstriert. Dagegen ist es durch ABFT-Algorithmen möglich auch solche Fehler zu entdecken und zu reparieren.

2.4 Fehlertypen und simuliertes Fehlerszenario

Wir beschränken uns innerhalb dieser Arbeit auf stille Datenkorruption (*engl.:* **S**ilent **D**ata **C**orruption, SDC). Diese Klasse von Fehlern zeichnet sich dadurch aus, dass sie keinen Einfluss auf die Eingabegrößen hat, sondern nur die Ausgabe von Funktionen bzw. Operationen verfälscht. Beispielhaft wird, ausgehend von den Werten $a = 4$ und $b = 5$, die Berechnung der Summe

$$c = a + b = 15$$

fehlerhaft durchgeführt. Im Folgenden liegen dabei die Variablen a, b weiterhin mit den korrekten Werten vor, wohingegen der Wert c nicht korrekt ist. Er entspricht nicht, wie erwartet, der Summe der beiden Variablen. Um diese Art von Fehlern zu detektieren, ist es also erforderlich das Ergebnis entsprechender Operationen kritisch zu betrachten und zu überprüfen. Wir unterscheiden im Bereich der SDC nach Elliott et al. [9] des Weiteren zwischen den in Abbildung 2.6 dargestellten Fehlertypen. Wir vermeiden dabei zwecks Eindeutigkeit die Übersetzung der einzelnen Arten.

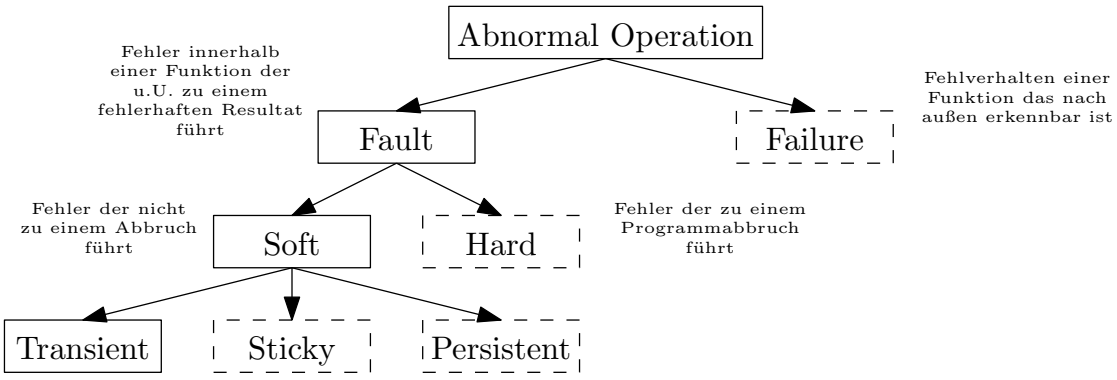

Abbildung 2.6: Darstellung der Zusammenhänge zwischen den verschiedenen Arten von möglichen Fehlertypen bei einer SDC, wobei gestrichelte Boxen Fehlertypen kennzeichnen, die wir innerhalb dieser Arbeit nicht betrachten. Darstellung und Bezeichnungen nach Elliott et al. [9].

Eine *abnormal Operation* kann sich grundlegend auf zwei Arten auswirken. Wir sprechen von einem *failure*, wenn sich der Fehler auf Nutzerebene bemerkbar macht, wie zum Beispiel durch die Divergenz eines iterativen Verfahrens von dem eigentlich Konvergenz zu erwarten ist. Führt die fehlerhafte Operation dagegen nicht zu einer unmittelbaren Auswirkung aus der Perspektive des Nutzers (z.B. die Konvergenz gegen eine falsche Lösung), so sprechen wir von einem *fault*.

Ein *fault* lässt sich dabei weiter spezifizieren. Zu einem sogenannten *hard fault* kommt es, wenn z.B. ein Teilprozess abstürzt. In diesen Situationen kann der Gesamtprozess nicht fortgesetzt werden und es kommt zu einem *failure*. Mit Methoden wie CPR oder TMR lassen sich solche Programmabbrüche durch *hard faults* jedoch unter Umständen verhindern. Die von uns im Folgenden schwerpunktmäßig betrachtete Art von Fehlern sind *soft faults*. Diese zeichnen sich dadurch aus, dass sie weder auf Nutzer- noch auf Systemebene ohne weitere Mechanismen detektiert werden können. Es ist zusätzlicher numerischer Aufwand, wie zum Beispiel durch die Verwendung von Prüfsummen (vgl. Kapitel 3), nötig um diese Fehlertypen zu erkennen. Führen diese Fehler zu falschen Lösungen, so sprechen wir außerdem von einem *silent failure*.

Darüber hinaus ist es möglich die Klasse der *soft faults*, abhängig von ihrer Beständigkeit, weiter zu unterteilen. Ist ein Modul der Hardware beschädigt, so sind die Auswirkungen dauerhaft (*persistent*). Andererseits ist es auch möglich, dass sich die Störung für einen begrenzten Zeitraum bemerkbar macht (*sticky*) und sich beispielsweise durch einen Neustart beheben lässt. In unserem Fall beschränken wir uns auf *transient soft faults*. Bei diesen *transient soft faults* handelt es sich um einmalige Störungen von Operationen.

Im Weiteren verstehen wir unter dem Begriff Störung bzw. Fehler jeweils die in diesem Kapitel als *transient soft fault* beschriebene Situation.

Simuliertes Fehlerszenario Um die erwarteten Konsequenzen der Entwicklungen im HPC-Bereich zu simulieren und mögliche Modifikationen an Algorithmen zu testen, ist es notwendig ein Fehlermodell zu verwenden. Dieses Fehlermodell kann dabei nicht alle Szenarien gleichzeitig abdecken und stellt somit nur einzelne Aspekte der erwarteten Entwicklung dar.

In unseren bisherigen Analysen zur Fehlertoleranz des Mehrgitterverfahrens [13] haben wir uns insbesondere auf das Verhalten des Verfahrens beim Ausfall von Rechenknoten konzentriert und immer angenommen, dass auftretende Fehler unmittelbar detektiert werden können. Innerhalb der vorliegenden Arbeit wollen wir nun den Schwerpunkt hin zu Fehlern innerhalb von Operationen verschieben und somit ist eine Modifikation unseres in bisherigen Arbeiten verwendeten Fehlermodells erforderlich. Wir sind insbesondere daran interessiert fehlerhafte Operationen zu detektieren und gegebenenfalls zu korrigieren. Diese bereits beschriebenen *soft faults* [10] führen i.d.R. nicht unmittelbar zu einem Abbruch der Verfahren.

Grundsätzlich erlauben wir Fehler innerhalb jeglicher numerischer Operationen, gehen aber davon aus, dass die Eingangsgrößen dabei nicht manipuliert werden und somit dass das Schreiben und Lesen aus dem Speicher fehlerfrei erfolgt. Einige vorgestellte Ansätze erlauben darüber hinaus auch Rückschlüsse auf Fehler in Daten im Speicher. So ist es z.B. durch die Verwendung der in Kapitel 3 beschriebenen Prüfsummen möglich, auf Fehler in den Operatoren zurückzuschließen.

Zur Vereinfachung beschränken wir uns bei der numerischen Analyse auf den seriellen Fall. Der Zusammenhang zu extrem parallelen Rechnungen im HPC-Bereich wird dadurch gewährleistet, dass die Verwendung von Gebietszerlegungsverfahren [24], wie sie u. a. in `FEAST` [27] verwendet werden, dazu führt, dass wenige Komponenten ein Teilproblem mit Hilfe eines eigenen lokalen Mehrgitterverfahrens lösen. Somit lassen sich die Erkenntnisse und Überlegungen auf parallele Verfahren übertragen.

Die Umsetzung der Fehlergenerierung variiert dabei von Situation zu Situation. Wir erläutern diese jeweils im Sachzusammenhang.

3 Prüfsummen

Als erstes Konzept zur Detektion von *soft faults* untersuchen wir die Verwendung von Prüfsummen. Diese finden ihre Anwendung insbesondere in der dichtbesetzten Linearen Algebra. Sie erlauben es dort die Korrektheit von Matrix- und Vektor-Operationen, bis zu einem gewissen Grad, zu garantieren ohne einen merkbaren Mehraufwand zu produzieren. Grund dafür ist, dass sich der Aufwand einer Matrix-Vektor Multiplikation bei dichtbesetzten Strukturen wie $O(N^2)$ verhält, wohingegen die Bestimmung einer Prüfsummen i.d.R. nur $O(N)$ Operationen benötigt. Huang und Abraham [15] beschreiben wie sich darüber hinaus mit Hilfe dieser Prüfsummen eine algorithmenbasierte Fehlertoleranz erzeugen lässt, die die entstandenen Fehler auch korrigieren kann.

In der Welt der dünnbesetzten Strukturen ist dies jedoch nicht ohne zusätzliche Überlegungen möglich. Hier erfordert die effektive Verwendung von Prüfsummen mehr Aufmerksamkeit. Insbesondere ist eine Lokalisierung der Fehler und die Korrektur nicht ohne Weiteres möglich.

3.1 Idee

Die Idee der Prüfsummen basiert auf der zweifachen Bestimmung eines Skalarproduktes unter Verwendung des Distributivgesetzes. Für das Skalarprodukt des Prüfvektors c mit dem Ergebnis der Matrix-Vektor-Multiplikation (`MV`-Operation) Ax gilt die Identität

$$c^T(Ax) = (c^T A)x. \tag{3.1}$$

Dabei lässt sich das Skalarprodukt $c^T A =: c_A$ einmalig bestimmen und im Zusammenhang mit der Matrix A abspeichern. c_A ist dabei der Vektor der gewichteten Spaltensummen der Matrix A. Erfolgt dann eine weitere Multiplikation eines Vektors mit der Matrix A, so lässt sich das Ergebnis im Skalarprodukt mit c mit dem Skalarprodukt $c_A^T x$ vergleichen. Stimmen beide Werte überein, so ist die Berechnung mit hoher Wahrscheinlichkeit korrekt durchgeführt worden. Sollten beide Berechnungen Fehler erzeugen, die sich gegenseitig aufheben wird dies jedoch nicht erkannt.

Ebenso lassen sich direkt erweiterte Operationen, wie die häufig verwendete Kopplung aus Matrix-Vektor-Multiplikation und einer Vektor-Addition (im Folgenden auch als `MVV`-Operation bezeichnet) durch

$$c^T(\alpha Ax + y) = \alpha(c^T A)x + c^T y$$

absichern. Auch für den Fall, dass bei der Anwendung eines Vorkonditionierers die Matrix A^{-1} nur implizit vorliegt, lässt sich durch

$$\begin{aligned} & A^{-1}x =: y \\ \Leftrightarrow \qquad & x = Ay \end{aligned}$$

die Prüfsummen-Technik mit

$$c^T x = c_A^T y$$

anwenden, d.h. grundlegend müssen bei derartigen Varianten nur die Rollen von Ergebnis- und Ausgangsvektor vertauscht werden.

Als Prüfvektor c wird in der einfachsten Variante der Einsvektor verwendet. Dadurch lassen sich Fehler in allen Komponenten detektieren. Es ist jedoch nicht möglich den Fehler genau zu lokalisieren. Dies wird erst durch mehrfache Anwendung mit unterschiedlichen Prüfvektoren ermöglicht, wie es u. a. von Sloan et al. [23] beschrieben wird. Durch eine solche Lokalisierung lassen sich dann schließlich auch Rechenfehler korrigieren. Für den Fall, dass der Prüfvektor dem Einsvektor entspricht, sprechen wir statt von gewichteten Spaltensummen nur noch von Spaltensummen und dem Spaltensummenvektor.

3.2 Numerischer Aufwand und Effizienz

Eine Matrix-Vektor-Multiplikation einer Matrix der Dimension $N \times N$ mit durchschnittlich M Komponenten pro Zeile erfordert $2MN$ Operationen (pro Eintrag eine Multiplikation und eine Addition). Die Bestimmung der beiden Skalarprodukte erfordert im allgemeinsten Fall jeweils $2N$ Operationen für beliebige Prüfvektoren, bei denen insbesondere eine Summation nicht ausreicht. Zusätzlich benötigt das einmalige Bestimmen der gewichteten Spaltensummen der Matrix, unter der Annahme, dass die Besetzungsstruktur symmetrisch ist (bei FE-Diskretisierungen üblich), $2MN$ Operationen.

Damit beläuft sich der Aufwand von n Matrix-Vektor-Multiplikationen der selben Matrix, bei gleichbleibendem Prüfvektor, auf

$$2MN + n(2MN + 4N).$$

Somit wird durch die Kontrolle ein relativer Mehraufwand von

$$\begin{aligned} & \frac{2MN + n(2MN + 4N)}{n2MN} \\ = & \frac{M(n+1)/n + 2}{M} \\ \xrightarrow{n\to\infty} \qquad & \frac{M+2}{M} = 1 + \frac{2}{M} \end{aligned}$$

erzeugt. Dies bedeutet, dass bei Q_1-Finite Elementen ($M = 9$, vgl. Abschnitt 2.1.1) ein Mehraufwand von mindestens 22% entsteht und bei Ansätzen höherer Ordnung, unter

der Annahme, dass viele Operationen mit der selben Matrix erfolgen, ein vernachlässigbarer zusätzlicher Zeitaufwand erzeugt wird (vgl. Abbildung 3.1). Werden unterschiedliche Prüfvektoren verwendet, so verschwindet der vermindernde Effekt bei steigender Iterationszahl, da die Neubestimmung der gewichteten Spaltensummen einen hohen Aufwand erzeugt und dominiert.

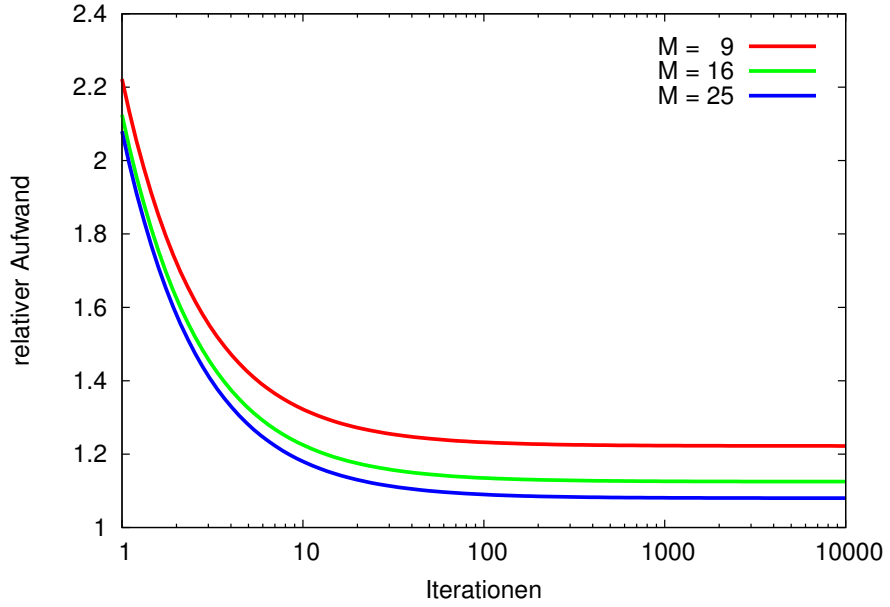

Abbildung 3.1: Relativer Aufwand der Matrix-Vektor-Multiplikation mit Prüfsummen bei festem Prüfvektor gegenüber der einfachen Variante für unterschiedlich stark besetzte Matrizen mit durchschnittlich M Nicht-Null-Einträgen pro Matrixzeile bei zunehmender Anzahl an Iterationen.

Etwas anders sieht dies bei der `MVV`-Variante aus. Die Berechnung selbst benötigt neben der Matrix-Vektor-Multiplikation zusätzlich eine Multiplikation (Skalierung mit α) und eine Addition pro Spalte. Bei der Prüfsumme wird ein zusätzliches Skalarprodukt benötigt. Die Skalierung mit der skalaren Größe α ist bei der Prüfsumme vom Aufwand her vernachlässigbar. Wir erhalten somit

$$2MN + n(2MN + 8N)$$

Operationen bei n Iterationen. Dies führt zu einem relativen Mehraufwand von

$$
\begin{aligned}
& \frac{2MN + n(2MN + 8N)}{n(2MN + 2N)} \\
=& \frac{2M(n+1) + 8n}{n(2M+2)} \\
=& \frac{2M(n+1)/n + 8}{2M+2} \\
\xrightarrow{n\to\infty} & \frac{2M+8}{2M+2} = 1 + \frac{6}{2M+2}.
\end{aligned}
$$

Hier führt die Verwendung von Q_1-Finite Elementen zu einem Mehraufwand von mindestens 30%. Der relative Mehraufwand in Abhängigkeit von der Anzahl an durchgeführten Iterationen für unterschiedliche Besetzungsstärken ist in Abbildung 3.2 dargestellt. Insgesamt erzeugt die Verwendung von Prüfsummen also bei der `MVV`-Variante einen größeren Mehraufwand als bei der einfachen Matrix-Vektor-Multiplikation.

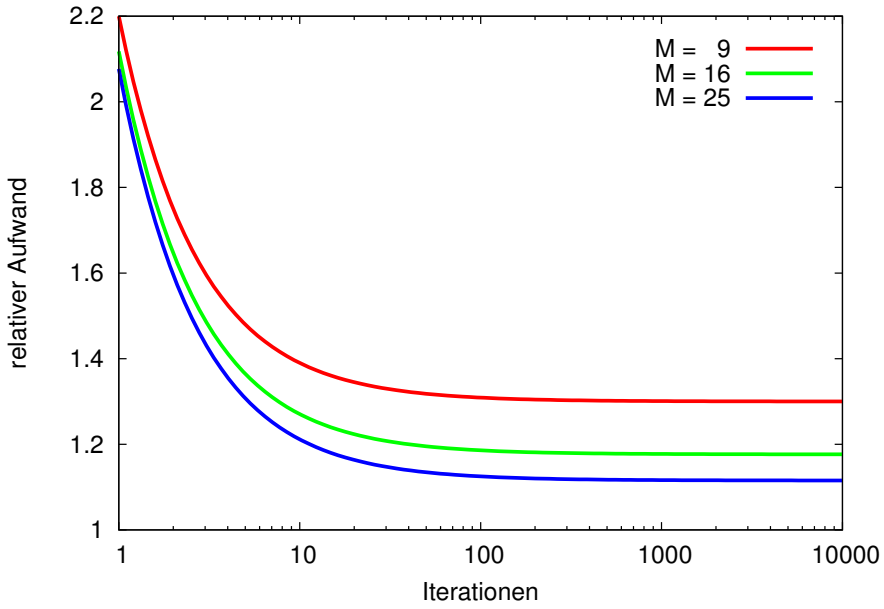

Abbildung 3.2: Relativer Aufwand der `MVV`-Variante mit Prüfsummen bei festem Prüfvektor gegenüber der einfachen Variante für unterschiedlich stark besetzte Matrizen mit durchschnittlich M Nicht-Null-Einträgen pro Matrixzeile bei zunehmender Anzahl an Iterationen.

Die Vorberechnung des gewichteten Spaltensummenvektors c_A ist bei Verwendung von einem ausgewählten Prüfvektor c sinnvoll, da die Einträge der Matrix A i.d.R. im Laufe der Anwendung konstant bleiben. Die Vorberechnung einer Vektorsumme ist dagegen nur in seltenen Fällen lohnenswert, denn die meisten verwendeten Vektorgrößen sind Veränderliche, wie z.B. der Lösungsvektor oder die Residuumsvektoren. Daher ist das

Absichern einer Vektoraddition, im Vergleich zu den algorithmisch komplexeren Matrix-Vektor-Multiplikationen, aufwändiger. Bei jeder Vektor-Vektor-Addition müssen zwei zusätzliche Skalarprodukte bestimmt werden, so dass sich der Aufwand nahezu um den Faktor drei erhöht.

In iterativen Prozessen wird häufig das Resultat der vorherigen Iteration als Eingangsgröße der Nächsten verwendet, sodass die Prüfsumme von Vektorgrößen zumindest zweimal benötigt wird. Somit kann der Mehraufwand bei effizienter Implementierung hierfür nahezu um den Faktor zwei reduziert werden. Dazu ist es jedoch notwendig sicherzustellen, dass die Prüfsummen der Vektoren fehlerfrei berechnet werden, was wiederum den Aufwand erhöht.

Nach Möglichkeit sollte eine Vektor-Vektor-Addition also mit einer Matrix-Vektor-Operation zu einer `MVV`-Operation gekoppelt werden, um den entstehenden Mehraufwand durch die aufwendigere Matrix-Multiplikation zu kaschieren.

Beim, in dieser Arbeit schwerpunktmäßig analysierten, Mehrgitterverfahren werden größtenteils `MV`- und `MVV`-Operationen durchgeführt (vgl. Algorithmus 2.1 und 2.2). Somit ist zu erwarten, dass Prüfsummen bei einer Verwendung von konformen Finite Elementen erster Ordnung zu einem Mehraufwand von mindestens 30% führen.

3.3 Lokaliserung und erweiterte Varianten

Neben der einfachen Anwendung von Prüfsummen, um Operationen auf Korrektheit zu überprüfen, gibt es erweiterte Ansätze, deren Ziel es ist den benötigten Aufwand zu reduzieren oder mit Hilfe von Lokalisierung und Reparatur eine algorithmenbasierte Fehlertoleranz zu ermöglichen.

Huang und Abraham [15] demonstrieren, wie diese Lokalisierung und Reparatur bei vollbesetzten Matrix-Matrix-Operationen möglich ist. Die Erweiterung der Matrizen um einen Spalten- und Zeilensummenvektor ermöglicht es die Korrektheit dieser zu überprüfen. Nach der Durchführung der Operation besitzt die resultierende Matrix ebenfalls einen zusätzlichen Spalten- und Reihensummenvektor. Der Vergleich dieser Summen mit den berechenbaren Summen der resultierenden Matrix gestattet es einen Fehler zu lokalisieren. Ein Fehler innerhalb der Operationen führt sowohl zu einer fehlerhaften Spalten- als auch Reihensumme, so dass der Fehler sich in der gemeinsamen Komponente befindet. Durch die Prüfsummen kann anschließend eine Reparatur erfolgen.

Dieser Ansatz ist jedoch nicht ohne Weiteres auf Matrix-Vektor-Operationen zu übertragen, die bei iterativen Prozessen sehr häufig benötigt werden. In diesem Abschnitt werden wir daher eine Möglichkeit der Lokalisierung auch für diese Operationen vorstellen und ebenso einige spezielle Wahlen für den Prüfvektor betrachten, die das Ziel haben den Aufwand in gewissen Situationen zu reduzieren.

3.3.1 Lokalisierung

Sloan et al. [23] schlagen im Falle einer positiven Detektion, zur Lokalisierung des Fehlers, die Einführung eines binären Suchbaums vor. So soll nach dem initialen Test mit einem Prüfvektor c, eine Kette aus Tests erfolgen, die den Ort des Fehlers immer weiter einschränken. Dafür wird der ursprüngliche Testvektor in eine obere und untere Hälfte unterteilt und jeweils mit Nullen aufgefüllt. Durch erneute Anwendung des Tests mit den beiden Vektoren lässt sich nun der Ort des Fehlers u. U. auf eine der Hälften einschränken. Sukzessive Anwendung dieses Prinzips erlaubt es schließlich den Fehler bis auf einzelne Komponenten zu lokalisieren (vgl. Abbildung 3.3). Es ist dabei zu beachten, dass durchaus mehrere Fehler auftreten können und daher bei einem positiven Resultat weiterhin beide Teile des vorherigen Prüfvektors getestet werden müssen. Erst wenn die Prüfsummen korrekt sind wird der entsprechende Teil des Suchbaums verworfen.

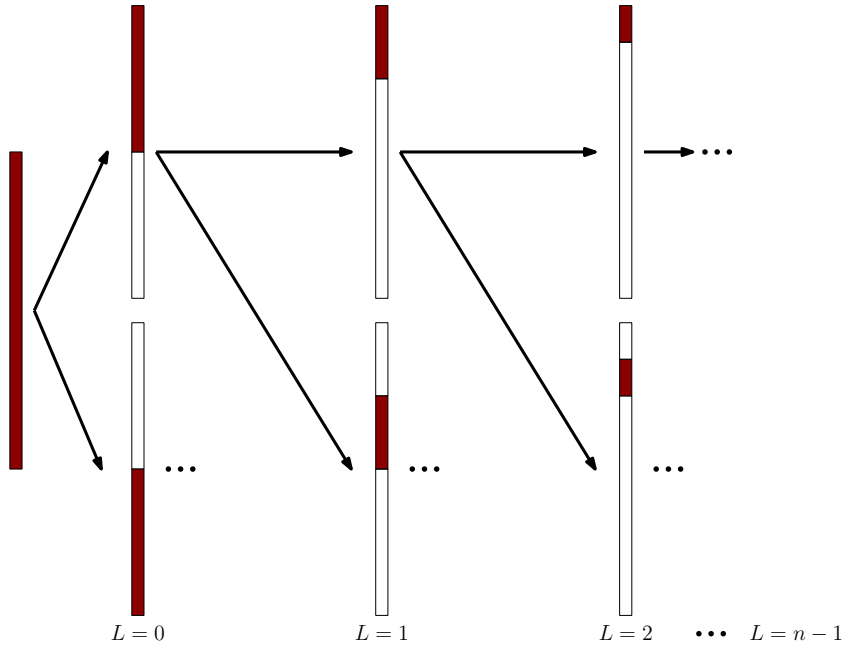

Abbildung 3.3: Ausschnitt eines binären Suchbaums der zur Lokalisierung eines Fehlers innerhalb einer Matrix-Vektor-Operation genutzt werden kann. Dabei werden die verwendeten Prüfvektoren dargestellt, wobei der rote Bereich den Komponenten entspricht, die von Null verschieden sind.

Um den Aufwand dieser Lokalisierung abzuschätzen, ist es hilfreich sich grundlegende Gedanken über die Art und Weise der Berechnung zu machen. Auch wenn nach der Aufteilung eines Prüfvektors in zwei Teile vom Prinzip her zwei neue Prüfungen erfolgen müssen, so ist es durch die Aufteilung des Vektors $c = \begin{pmatrix} a & b \end{pmatrix}^T$ und der trivialen

Umformung

$$\begin{pmatrix} a \\ b \end{pmatrix}^T (Ax) = (\begin{pmatrix} a \\ b \end{pmatrix}^T A)x$$
$$\Leftrightarrow \quad \begin{pmatrix} a \\ 0 \end{pmatrix}^T (Ax) + \begin{pmatrix} 0 \\ b \end{pmatrix}^T (Ax) = (\begin{pmatrix} a \\ 0 \end{pmatrix}^T A)x + (\begin{pmatrix} 0 \\ b \end{pmatrix}^T A)x$$

möglich, dass nur zu einem der Vektoren eine Neuberechnung erfolgen muss. Somit lässt sich für einen beliebigen Ausgangsprüfvektor $c \in \mathbb{R}^N$ mit $N = 2^n$ (d.h. beliebige Einträge sind zugelassen) der Aufwand für die Lokalisierung eines Fehlers durch

$$\begin{aligned} &\sum_{L=0}^{n-1} 2M \cdot 2^{n-1-L} + \sum_{L=0}^{n-1} 2 \cdot 2 \cdot 2^{n-1-L} \\ =&M(2^{n+1} - 2) + 2^{n+2} - 4 \\ =&2MN + 4N - 2M - 4 \end{aligned}$$

abschätzen. Die erste Summe stellt dabei die Bestimmung des Produktes $\tilde{c}^T A$ zum halbierten Prüfvektor $\tilde{c}$ dar. In der zweiten Summe stecken die Operationen, die zur Bestimmung der Skalarprodukte notwendig sind. Dabei ist zu erinnern, dass wir den allgemeinsten Fall mit beliebigen Einträgen im Prüfvektor betrachten, so dass auch eine Multiplikation pro Komponente notwendig ist.

Insgesamt erhalten wir somit einen Aufwand von ca.

$$2MN + 4N,$$

der benötigt wird, um einen Fehler zu lokalisieren.

Dieser reduziert sich im Falle der Verwendung des Einsvektors nochmals um den Faktor zwei (bei effizienter Implementierung, d.h. ausschließlich Summation statt Multiplikation und Summation). Der Aufwand ist also vergleichbar mit dem einer Matrix-Vektor-Multiplikation ($2MN$ Operationen).

Des Weiteren ist zu beachten, dass die Bestimmung der Spaltensummen $\tilde{c}^T A$ bei Matrizen im CSR-Format (*engl.:* **C**ompressed **S**parse **R**ow) entweder mit vielen Sprüngen im Speicher verbunden ist oder aber das zusätzliche Abspeichern in einem CSC-Format (*engl.:* **C**ompressed **S**parse **C**olumn) erfordert. Alternativ kann auch zusätzlich die transponierte Matrix bestimmt und verwendet werden, um Sprünge innerhalb des Speichers zu vermindern. Dies erfordert jedoch zusätzlichen Speicherplatz.

Sloan et al. [23] erreichen mit Hilfe von geschicktem Zwischenspeichern eine weitere Reduktion des Aufwandes durch die Wiederverwertung von Ergebnissen. Dies ist jedoch mit hohem Implementierungsaufwand verbunden.

Bereits die benötigte Anzahl der numerischen Operationen zur Lokalisierung ist vergleichbar mit denen einer weiteren Matrix-Vektor-Multiplikation. Zusätzlich erfordert

die vorgestellte Variante eine Struktur, die vergleichbar mit dem binären Suchbaum ist und eine optimierte Implementierung, um die Anzahl an redundanten Rechnungen zu minimieren. Im Falle eines detektierten Fehlers kann also auch die Berechnung wiederholt werden, ohne dass der Aufwand wesentlich größer ist. Die Tatsache, dass dafür der Ausgangsvektor zwischengespeichert werden muss, ist insofern gerechtfertigt, da auch zusätzliche Prüfvektoren weiteren Speicherplatz benötigen.

3.3.2 Null- bzw. Einsvorkonditionierung

Neben der Lokalisierung schlagen Sloan et al. [22] einige Möglichkeiten zur Wahl des Prüfvektors c vor. Sie bezeichnen ihre Varianten als Null- bzw. Einsvorkonditionierung. Hierbei ist es das Ziel c so zu bestimmen, dass

$$c^T A = \bar{1} \qquad \text{oder} \qquad c^T A = \bar{0}$$

gilt. Die rechten Seiten stellen in diesem Fall Vektoren mit gleichen Einträgen in jeder Komponente dar. Durch diese Wahl kann der Aufwand zur Bestimmung der Prüfsummen reduziert werden. Es ist jedoch zusätzlicher Aufwand zur Initialisierung notwendig, da die Prüfvektoren problemspezifisch ermittelt werden müssen.

Einsvorkonditionierung Die Bestimmung eines Vektors c, so dass

$$c^T A = \bar{1}$$

gilt, erfordert üblicherweise das Lösen eines Gleichungssystems desselben Schwierigkeitsgrades wie das Ausgangsproblem, dessen Operationen abgesichert werden sollen. Um dieses Problem zu vermeiden, schlagen Sloan et al. vor das Minimierungsproblem

$$c = \arg\min_{v} \|A^T v - \bar{1}\|$$

approximativ zu lösen. Bereits mit wenigen Iterationen erhalten sie, innerhalb ihrer Tests, brauchbare Resultate. Der Aufwand zur Bestimmung ist vergleichbar mit mehreren Matrix-Vektor-Multiplikationen. Daher wird der Ansatz erst interessant, wenn die Matrix häufig für Operationen verwendet wird.

Der Gewinn gegenüber eines beliebigen Prüfvektors wird bei Betrachtung der Prüfsummenberechnung

$$c^T(Ax) = (c^T A)x = \bar{1}^T x = \sum x$$

deutlich. Im Vergleich zur allgemeinen Variante (siehe Abschnitt 3.2) halbiert sich der Aufwand zur Berechnung des Prüfwertes auf der rechten Seite. Wir benötigen nicht mehr eine Multiplikation und Addition pro Komponente, sondern ausschließlich eine Addition. Auf der linken Seite ist jedoch nun auf jeden Fall je eine Multiplikation und Addition notwendig.

Einen vergleichbaren numerischen Gewinn erzielt man auch durch die Wahl von $c = \bar{1}$. Hierbei reduziert sich der Aufwand auf der linken Seite der Gleichung. Anstelle des Lösens eines Minimierungsproblems ist es nun notwendig einmalig die Spaltensummen von A zu bestimmen.

Nullvorkonditionierung Durch die Verwendung eines Vektors c mit

$$c^T A = \bar{0}$$

reduziert sich die Überprüfung einer Matrix-Vektor-Multiplikation Ax auf den einfachen Test

$$c^T(Ax) = \bar{0}.$$

Es ist also ausschließlich die Bestimmung eines Skalarproduktes (eine Multiplikation und eine Addition) notwendig. Sloan et al. [22] verwenden für den Prüfvektor c den Singulärvektor zum kleinsten Singulärwert σ. Dieser erfüllt

$$Ac = \sigma u \qquad \text{und} \qquad A^* u = \sigma c,$$

wobei A^* die transponierte konjugierte Matrix beschreibt. Mit diesem Singulärvektor gilt: $c^T A \approx \bar{0}$. Sloan et al. [22] stellen in ihren numerischen Tests fest, dass Singulärvektoren zu Singulärwerten kleiner als 10^{-6} bereits gut geeignet sind.

Die Bestimmung des Prüfvektors c erfordert in diesem Fall also die Berechnung eines möglichst kleinen Singulärwertes. Der Aufwand dafür ist mit Hilfe von geeigneten Bibliotheken (z.B. LAPACK [1]) erneut vergleichbar mit wenigen Matrix-Vektor-Multiplikationen. Der Wegfall des Skalarproduktes auf der rechten Seite ist jedoch damit verbunden, dass auf der linken Seite der Prüfsumme ein vollwertiges Skalarprodukt zu bestimmen ist.

Aufgrund der erhöhten Komplexität zur Bestimmung der Eins- bzw. Nullvorkonditionierung und dem im Vergleich dazu geringen Vorteil, werden wir uns in dieser Arbeit auf den Einsvektor als Prüfvektor beschränken.

3.4 Verifizierung des Modells

Im Folgenden wollen wir das in Kapitel 3.2 beschriebene Modell zur Aufwandsschätzung überprüfen und anhand einer Implementierung testen, da wir für das abschließend konstruierte fehlertolerante Mehrgitterverfahren (siehe Kapitel 5) auch die Prüfsummen als eine Komponente verwenden werden.

Wir verwenden bei der Implementierung das CSR-Format für dünnbesetzte Matrizen, ausschließlich den Einsvektor als Prüfvektor und somit insbesondere keine Lokalisierung. Durch die Wahl des Prüfvektors reduzieren sich einige Skalarprodukte auf Summationen, was zu einer Reduzierung des Aufwandes führen soll. Das Produkt $c^T A$, welches den

Spaltensummen entspricht, kann, nach einmaliger Berechnung, als zusätzliche Zeile der Matrix A interpretiert und gespeichert werden. Dadurch wird bei einer Matrix-Vektor-Multiplikation automatisch die Prüfsumme auf der rechten Seite von Gleichung (3.1) mitbestimmt (vgl. Abbildung 3.4) und es ist nur noch notwendig das Skalarprodukt von c und dem Ergebnisvektor zu bestimmen sowie beide Resultate zu vergleichen. Dafür ist jedoch sicherzustellen, dass die initiale Berechnung der Prüfsummen (Spaltensummen der Matrix) korrekt erfolgt ist. Dies kann beispielsweise durch eine doppelte Berechnung und einen entsprechenden Abgleich garantiert werden.

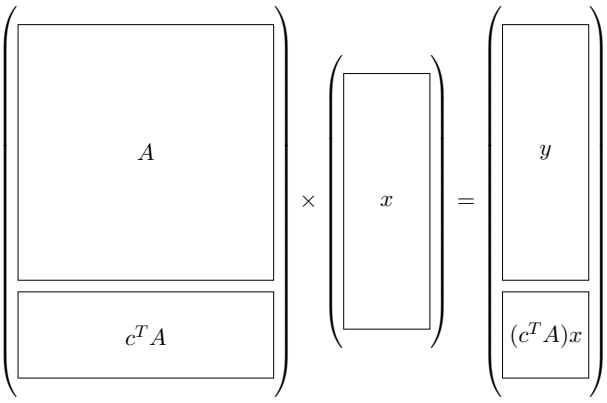

Abbildung 3.4: Modellhafte Darstellung einer Matrix-Vektor-Multiplikation mit Prüfsummen.

Zur Analyse des Modells führen wir fünf Testreihen auf einem MacBook Pro (Mitte 2012) mit einem 2,9 GHz Dual-Core Intel i7 Prozessor, 4MB L3-Cache und 8GB DDR3 RAM sowie Mac OS 10.10.2 durch. Der für den Test benötigte Speicherplatz beläuft sich auf unter 2GB, so dass alle zur Berechnung notwendigen Daten im RAM abgelegt werden können. In Tabelle 3.1 sind die Mittelwerte der Laufzeiten der Variante ohne Prüfsummen (Spalte o), der Variante mit Verwendung von Prüfsummen (Spalte m) sowie der daraus resultierende relative Mehraufwand, dargestellt. Wir testen die Implementierung an Matrizen mit einer Besetzungsstärke von 9 bzw. 16, was der Verwendung von Q_1- bzw. Q_2-Finite Elementen entspricht, wie im entsprechenden Abschnitt von Kapitel 2.1.1 gezeigt wurde. Das der Assemblierung zugrundeliegende Gitter ist dabei jeweils so gewählt, dass sich die Anzahl der Freiheitsgrade auf 4 198 401 beläuft.

Zur Veranschaulichung der Ergebnisse sind diese in Abbildung 3.5 gegenüber den vom Modell vorhergesagten Werten dargestellt. Für die untersuchten Operationen ist zu erkennen, dass die Vorhersage des Modells bei hohen Iterationszahlen eintreffen. Bei wenigen Iterationen fällt auf, dass das Modell zu optimistisch ist. Dies liegt daran, dass bei der einmaligen Berechnung von $c^T A$ neben dem reinen Aufwand der Berechnung zusätzliche Kommunikation entsteht und aufgrund der Struktur auch viele Sprünge im Speicher notwendig sind. Zu bemerken ist außerdem, dass der Mehraufwand in der Implementierung für große Iterationszahlen geringer ist als es das Modell vorhersagt. Dies liegt

	M \ #Iter	1		10		100		1000	
		o	m	o	m	o	m	o	m
MV	9	0.076	0.262	0.766	1.067	7.384	9.268	76.813	93.121
		×3.454		×1.393		×1.255		×1.212	
	16	0.132	0.441	1.256	1.687	12.653	14.560	128.690	143.704
		×3.344		×1.343		×1.154		×1.117	
MVV	9	0.083	0.260	0.775	1.132	7.929	10.095	79.151	100.321
		×3.152		×1.460		×1.273		×1.267	
	16	0.147	0.436	1.306	1.807	13.257	15.686	131.901	153.794
		×2.970		×1.383		×1.183		×1.166	

Tabelle 3.1: Zeitmessung für die Ausführung der Operationen **o**hne und **m**it Prüfsummen für eine Matrix der Dimension $4\,198\,401 \times 4\,198\,401$ und unterschiedlichen Besetzungsdichten sowie Angabe des daraus resultierenden Faktors an Mehraufwand.

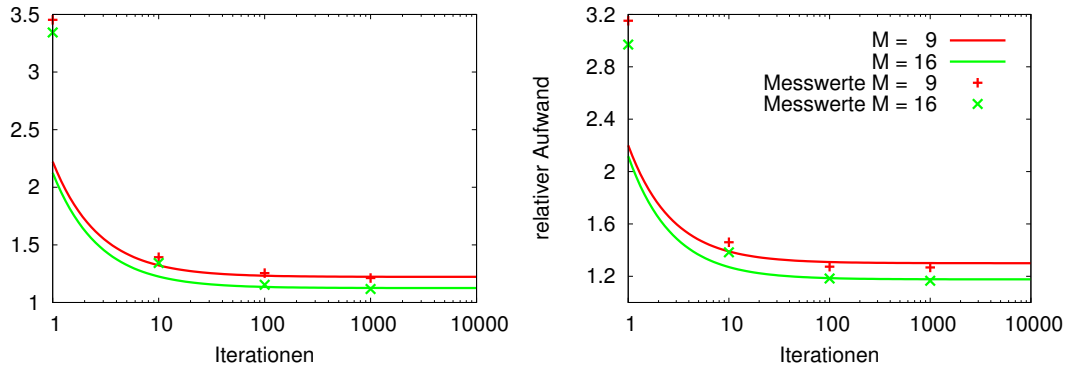

Abbildung 3.5: Vergleich des erzeugten Mehraufwands bei Modell und Implementierung für die `MV`- bzw. `MVV`-Operation (rechts) mit den Messdaten aus Tabelle 3.1.

u. a. an der Verwendung des Einsvektors als Prüfvektor. Wie einleitend angesprochen reduzieren sich dadurch einige Skalarprodukte auf die Berechnung einer Summe. Bei der Matrix-Vektor-Multiplikation wird ein Skalarprodukt entsprechend reduziert und bei der `MVV`-Operationen zwei Skalarprodukte. Dies erklärt, warum der Unterschied dort etwas stärker ausgeprägt ist.

3.5 Fazit: Prüfsummen

Wir haben gezeigt, dass die Verwendung von Prüfsummen grundsätzlich auch bei dünnbesetzten Matrizen attraktiv sein kann, sofern die entsprechenden Matrizen häufig genug verwendet werden. Es entsteht jedoch, im Gegensatz zur Welt der dichtbesetzten Matrizen, ein spürbarer Mehraufwand, der sich je nach Besetzungsdichte in einem Bereich zwischen 10% und 40% bewegt. Darüber hinaus ist es möglich den Fehler, durch eine Lokalisierung und mit erweiterten Methoden, zu beheben, wenn denn die Prüfsummen-

berechnung als fehlerfrei angenommen werden kann. Sollten Fehler in der Berechnung einer Prüfsumme auftreten, so kann jedoch eventuell eine korrekte Operation verfälscht werden.

Die Lokalisierung erfordert zusätzlich eine durchdachte Implementierung, da die mehrfache Berechnung von Spaltensummen zu Teilvektoren unter Verwendung von zwischengespeicherten Werten optimiert werden muss, um wirklich effizient zu sein. Dies ist nicht für alle Matrixstrukturen trivial möglich.

Es ist außerdem möglich spezielle Prüfvektoren zu verwenden, um gewisse Eigenschaften von Matrizen oder speziellen Gleichungen auszunutzen. Mit der Eins- bzw. Nullvorkonditionierung haben wir dafür zwei Beispiele vorgestellt.

Als Quintessenz ist festzuhalten, dass die Verwendung von Prüfsummen bei häufiger Verwendung der selben Matrizen eine einfache und in dem Fall auch verhältnismäßig günstige Möglichkeit ist, um zu überprüfen ob eine Matrix-Vektor-Operation korrekt ausgeführt wurde. Die unmittelbare Korrektur dieser Rechnung erfordert jedoch einen nicht zu vernachlässigenden zusätzlichen Aufwand. Alternativ kann bei positiver Detektion eine Neuberechnung durchgeführt werden. Dies erfordert jedoch das Zwischenspeichern der Eingangsgrößen. Diese dürfen innerhalb der Berechnung nicht verändert werden.

Darüber hinaus ist es durch die Verwendung von Prüfsummen möglich auf Fehler innerhalb der Operatoren zurückzuschließen. Sollten z.B. die Prüfsummen einer Matrix-Vektor-Multiplikation auch nach mehrmaliger Ausführung nicht übereinstimmen, so ist zu vermuten, dass der Operator fehlerhaft ist. In diesem Fall kann dann eine Neuassemblierung durchgeführt werden, gefolgt von einer erneuten Ausführung der Operation.

Eine allgemeine Schwachstelle der Prüfsummen innerhalb von iterativen Prozessen ist die Wahl der Toleranz für die Überprüfung der Gleichheit der Prüfsummen. Aufgrund von Rundungsfehler ist zu keinem Zeitpunkt davon auszugehen, dass beide Seiten der Gleichung exakt übereinstimmen. Vielmehr muss ein Parameter τ gewählt werden, so dass z.B. für die Matrix-Vektor-Multiplikation

$$|c^T(Ax) - (c^T A)x| \leq \tau$$

gilt. Diese numerische Komponente und die Tatsache, dass sich die Lösungen und Hilfsgrößen im Laufe des Lösungsprozesses nur dem Ergebnis annähern, macht eine adaptive Wahl dieses Toleranzwertes attraktiv. Zu frühen Zeitpunkten können kleine Fehler innerhalb der Matrix-Vektor-Multiplikationen durchaus toleriert werden, wohingegen im späteren Konvergenzverlauf ein exakteres Ergebnis wünschenswert ist. Die Wahl dieses Wertes ist daher nicht trivial.

Die bisherige Analyse der möglichen Varianten von Prüfsummen für dünnbesetzte Strukturen führt dazu, dass wir uns innerhalb der weiteren Arbeit mit dem Einsvektor als Prüfvektor zufriedengeben. Außerdem verwenden wir keine Lokalisierung bzw. Reparatur, sondern führen bei positiver Detektion nur eine Neuberechnung der Operation durch. Wie in Abschnitt 3.3 gezeigt, ist der Aufwand dabei durchaus vergleichbar und die Implementierung ist wesentlich einfacher zu handhaben.

4 Algorithmenbasierte Fehlertoleranz für den Glättungsprozess

Die Stabilisierung von numerischen Operationen bei dünnbesetzten Strukturen durch Prüfsummen ist, wie wir in Kapitel 3 gezeigt haben, durchaus aufwändig und die anschließende Lokalisierung und Reparatur erfordert viel Arbeit. Für den Fall, dass eine anschließende Reparatur der Fehler möglich ist, spricht man von sogenannter algorithmenbasierter Fehlertoleranz (*engl.:* **A**lgorithm **B**ased **F**ault-**T**olerance, ABFT, [3]). Die Verwendung von Prüfsummen innerhalb der Operationen der linearen Algebra ist dabei eines der niedrigsten Level auf denen dieser Ansatz erfolgen kann. In diesem Kapitel werden wir die Glättungsphase des Mehrgitterverfahrens um eine solche algorithmenbasierte Fehlertoleranz erweitern.

Obwohl das Mehrgitterverfahren eines der attraktivsten Verfahren zum Lösen von dünnbesetzten linearen Gleichungssystemen ist, existieren bisher nur wenig Forschungsergebnisse zu den Eigenschaften des Algorithmus bzgl. Fehlertoleranz. Casas et al. [6] untersuchen die Fehlertoleranz des algebraischen Mehrgitterverfahrens, insbesondere die Auswirkung von Störungen in unterschiedlichen Komponenten des Algorithmus. Sie kommen zu dem Schluss, dass das Verfahren an sich sehr robust ist und in den meisten Fällen auch beim Auftreten von Fehlern konvergiert. Außerdem ist es ihnen gelungen durch einen vergleichsweise geringen Mehraufwand die Fehleranfälligkeit durch Speicherabzugsfehler (*engl.:* segmentation faults) um 50 - 80% zu reduzieren. Auf Fehler innerhalb von numerischen Operationen gehen sie jedoch nicht weiter ein. Elliott et al. [10] analysieren die Auswirkung von Bitflips auf das Konvergenzverhalten von einfachen iterativen Methoden am Beispiel des Jacobi-Verfahrens und betrachten dabei ebenfalls die Relevanz der Position des Fehlers. Sie stellen dabei jedoch weder einen Detektions- noch Stabilisierungsalgorithmus vor.

Wir werden nun, aufbauend auf dem Ansatz des minimalen Checkpointings [13], der im Bereich des Hochleistungsrechnens insbesondere geeignet ist um den Verlust durch Knotenausfälle zu minimieren, den Mehrgitteralgorithmus gegenüber einzelnen Bitflips, auch innerhalb von Operationen, stabilisieren. Somit also den Effekt lokaler Störungen in einzelnen Komponenten reduzieren. Die Auswirkung solcher Störungen auf das Konvergenzverhalten wurde bereits in früheren Arbeiten [13] analysiert. Insbesondere führen solche Fehler nicht zwangsweise zu einem Abbruch des Prozesses, sondern machen sich erst durch Anomalien z.B. im Konvergenzverlauf bemerkbar.

4.1 Anwendung

Als Ausgangspunkt für unsere Konstruktion eines Mechanismus zur Fehlerdetektion und -reparatur verwenden wir den in Kapitel 2.2.2 vorgestellten FAS-Ansatz des Mehrgitterverfahrens (vgl. Algorithmus 2.2). Dieser Ansatz hat den Vorteil, dass auf jedem Gitterlevel eine Lösungsapproximation existiert und nicht nur ein Korrekturvektor. Damit können wir eine intuitive Lokalisierung von Fehlern konstruieren und erhalten insbesondere eine einfachere Reparatur, denn auf jedem Level liegt eine eigene Lösungsapproximation vor, wie in Abschnitt 2.2.2 beschrieben. Zusätzlich ermöglicht uns dieser Ansatz ohne großen Berechnungsaufwand das Backup einer komprimierten Lösung, um das Verfahren mit dem bereits erwähnten minimalen Checkpointing [13] zu kombinieren.

Der Hauptaufwand des Mehrgitterverfahrens steckt, wie in Abschnitt 2.2.3 untersucht, in der Glättungsphase. Hierbei werden je nach Komplexität des verwendeten Operators unterschiedlich viele Matrixoperationen durchgeführt. Die Verwendung von Prüfsummen, um alle Operationen auf Fehler zu überprüfen, ist aufwendig. Nach dem Modell aus Kapitel 3 ist ein Mehraufwand von ungefähr 30% (Aufwand einer `MVV`-Operation bei der Verwendung von linearen Finite Elementen) zu erwarten, da in der Regel innerhalb von iterativen Lösern auch einige Vektor-Vektor-Additionen und Skalarprodukt-Berechnungen erfolgen, ist sogar durchaus mit einem höheren Mehraufwand zu rechnen. Darüber hinaus erfordert die Korrektur im fehlerhaften Fall das Zwischenspeichern von weiteren Größen, damit durch eine Neuberechnung der Fehler korrigiert werden kann. Zusätzlich müsste jeder Glätter innerhalb der Implementierung u. U. angepasst werden. Wir werden daher den Glättungsoperator als *Black-Box* betrachten und im Anschluss daran, mit Eigenschaften des Mehrgitterverfahrens, das Ergebnis auf Tauglichkeit überprüfen. Für den Fall, dass das Ergebnis gewissen Eigenschaften nicht genügt, soll im Anschluss eine Reparatur erfolgen.

Die weiteren Operationen, die zur Prolongation bzw. Restriktion oder Korrektur benötigt werden, lassen sich dadurch jedoch nicht direkt absichern. Dies ist erst durch eine Kombination mit anderen Ansätzen möglich und wird in Kapitel 5 umgesetzt. Wir gehen daher bei dieser Analyse ausschließlich von Störungen im Glättungsprozess aus und betrachten auch den Grobgitterlöser als fehlerfrei, da der Aufwand verhältnismäßig gering ist und z.B. durch mehrfaches Berechnen die Richtigkeit garantiert werden kann.

Zur Herleitung eines Mechanismus werden wir getrennt voneinander die Prolongations- und Restriktionsphase betrachten, da dort durchaus unterschiedliche Situationen auftreten, die andere Methoden erfordern. Dabei beschreiben wir in Abschnitt 4.1.1 und 4.1.2 die Ideen und geben später in Kapitel 4.2 die daraus resultierenden Algorithmen an.

4.1.1 Prolongationsphase

Der Anwendung des Glättungsoperators, ausgenommen der auf dem Feingitter, innerhalb der Prolongationsphase folgt der Transfer des berechneten Korrekturvektors

$$v - v^{(0)} =: c.$$

auf das nächstfeinere Gitter. Dieser Korrekturvektor c zeichnet sich dadurch aus, dass er gegen den Nullvektor konvergiert, denn auf jedem nächstgröberem Gitter wird das Problem (vgl. Zeile 8 und 9 von Algorithmus 2.2)

$$\begin{aligned} A_{k-1}v &= \mathbf{R}_k^{k-1}(r_k) + A_{k-1}v^{(0)} \\ &= \mathbf{R}_k^{k-1}(b - A_k u^{(\nu)}) + A_{k-1}v^{(0)} \\ \Leftrightarrow \qquad A_{k-1}(v - v^{(0)}) &= \mathbf{R}_k^{k-1}(b - A_k u^{(\nu)}) \end{aligned}$$

gelöst, wobei $b - A_k u^{(\nu)}$ aufgrund der Kontraktionseigenschaft des Mehrgitterverfahrens konvergiert und somit die rechte Seite der Gleichung gegen den Nullvektor strebt. Demzufolge gilt dies auch für den Korrekturvektor c auf der linken Seite. Wir haben hiermit also eine intuitive Position, um Fehler innerhalb des Glätters zu detektieren. Für den Fall, dass eine Anomalie in diesem Konvergenzverlauf auftritt, ist eine Störung anzunehmen. Detektion und Lokalisierung werden dabei durch in vorherigen Iterationen generierte Vergleichswerte ermöglicht. Auf die Wahl dieses Wertes gehen wir in Abschnitt 4.1.3 genauer ein.

Der Vorteil des FAS-Ansatzes liegt, wie bereits erwähnt, darin, dass auf jedem Gitterlevel eine Lösungsapproximation gespeichert ist. Sollte eine fehlerhafte Komponente im Korrekturvektor c detektiert werden, so kann unter der Annahme, dass bisher aufgetretene Fehler erkannt und repariert wurden, die entsprechende Komponente aus v durch eine lokale Prolongation der Lösung des nächstgröberen Gitters ersetzt werden. Dieser Berechnung folgt schließlich eine lokale Neuberechnung des Korrekturvektors c. Einige Lösungskomponenten werden also durch ungeglättete Einträge ersetzt.

Erst nachdem der Korrekturvektor derart überprüft und eventuell repariert wurde, erfolgt die Prolongation und Aktualisierung der Lösung auf dem nächsten Level. Durch diese Kontrolle kann also die Tauglichkeit der ausgeführten Glättung überprüft werden. Ausgenommen ist dabei das Nachglätten auf dem Feingitter. Da im Anschluss keine erneute Korrektur bzw. Prolongation erfolgt, kann auf vergleichbare Weise keine Überprüfung erfolgen. Hier ist es jedoch möglich eine Überprüfung, ähnlich zur Vorgehensweise innerhalb der Restriktionsphase, durchzuführen, wie wir sie im folgenden Abschnitt beschreiben.

4.1.2 Restriktionsphase

Im Gegensatz zur Prolongationsphase wird in der Restriktionsphase kein Korrekturvektor berechnet mit Hilfe dessen eine Überprüfung der Lösung möglich ist. Wir können jedoch auch hier eine Komponente identifizieren, die uns eine ähnliche Überprüfung erlaubt.

Wir erweitern die Darstellung der Lösungsapproximationen sowie des Vektors b der rechten Seite im Folgenden um einen levelspezifischen Index k. Dies ermöglicht eine genauere Zuordnung der Komponenten.

In Zeile 5 von Algorithmus 2.2 wird, nach der Anwendung des Glättungsoperators, das aktuelle Residuum

$$r_k = b_k - A_k u_k^{(\nu)}$$

berechnet, wobei $u_k^{(\nu)}$ die vorgegelättete Lösungsapproximation darstellt. Auf dem Feingitter haben wir aufgrund der Eigenschaften des Mehrgitterverfahrens Konvergenz und asymptotisch eine monotone Reduktion der Residuumsnorm. Sonst handelt es sich bei der Größe $u_k^{(\nu)}$ um die vorgeglättete Verbesserung der restringierten Lösungsapproximation $u_k^{(0)} = \mathbf{I}_{k+1}^k u_{k+1}^{(\nu)}$ des feineren Gitters. Des Weiteren lässt sich der Vektor b_k durch

$$b_k = \mathbf{R}_{k+1}^k (b_{k+1} - A_{k+1} u_{k+1}^{(\nu)}) + A_k \mathbf{I}_{k+1}^k u_{k+1}^{(\nu)}$$

aus Komponenten des feineren Gitters darstellen. Hierbei wird ausgenutzt, dass sich der Vektor b_k aus dem restringierten Residuum des feineren Gitters und der Anwendung des dem Level zugehörigen Operators auf die restringierte Lösungsapproximation ergibt (vgl. Zeile 5-8 in Algorithmus 2.2). Damit lässt sich der Residuumsvektor schließlich in

$$\begin{aligned} r_k &= \mathbf{R}_{k+1}^k (b_{k+1} - A_{k+1} u_{k+1}^{(\nu)}) + A_k \mathbf{I}_{k+1}^k u_{k+1}^{(\nu)} - A_k u_k^{(\nu)} \\ &= \mathbf{R}_{k+1}^k (b_{k+1} - A_{k+1} u_{k+1}^{(\nu)}) + A_k (u_k^{(0)} - u_k^{(\nu)}) \end{aligned}$$

zerlegen und wir erhalten in der 2-Norm die Abschätzung

$$\|r_k\|_2 \leq \|\mathbf{R}_{k+1}^k\|_2 \|(b_{k+1} - A_{k+1} u_{k+1}^{(\nu)})\|_2 + \|A_k\|_2 \|(u_k^{(0)} - u_k^{(\nu)})\|_2.$$

Hierbei sind die Operatoren $\mathbf{R}_{k+1}^k$ und A_k per Konstruktion in der 2-Norm beschränkt. Des Weiteren konvergiert die Komponente $b_{k+1} - A_{k+1} u_{k+1}^{(\nu)}$ aufgrund der Eigenschaften des Mehrgitterverfahrens auf dem feinsten Gitter zumindest asymptotisch. Für den Fall, dass der Glätter die Kontraktionseigenschaft besitzt, konvergiert auch $u_k^{(0)} - u_k^{(\nu)}$ in der 2-Norm. Sollte keine Vorglättung erfolgen, so ist der entsprechende Term irrelevant. Damit konvergiert der Residuumsvektor r_k auch auf dem zweitfeinsten Gitter. Rekursive Anwendung dieser Argumentation liefert schließlich auf allen Gitterleveln unabhängig die Konvergenz.

Der Residuumsvektor r_k konvergiert somit, im Verlauf des iterativen Prozesses, auf jedem Level gegen den Nullvektor. Dadurch haben wir nun eine vergleichbare Situation, wie in der Prolongationsphase. Wir können anhand eines Vergleichs der Komponenten mit einem Wert aus der vorherigen Iteration überprüfen, ob die Komponenten unerwarteter Weise größer geworden sind und somit möglicherweise eine Störung aufgetreten ist.

Für den Fall, dass ein Fehler im Residuumsvektor lokalisiert wurde, ist darauf zurückzuschließen, dass auch der Lösungsvektor $u_k^{(\nu)}$ fehlerhafte Komponenten enthält, wobei

wir hier davon ausgehen, dass die Residuumsbestimmung fehlerfrei erfolgt. Mit Hilfe des Besetzungsmusters der Systemmatrix A_k lässt sich nun ein maximaler Träger ermitteln und der Fehler in $u_k^{(\nu)}$ so anhand der detektierten Störung im Residuumsvektor einschränken. Die möglicherweise fehlerhaften Komponenten können dann durch die der nicht geglätteten Lösungsapproximation $u_k^{(0)}$ ersetzt werden. Abschließend erfolgt eine neue Berechnung des Residuums.

Analoges Vorgehen ist für das Nachglätten auf dem Feingitter möglich. Hier ist es jedoch notwendig vor der Anwendung des Glätters die Lösungsapproximation explizit abzuspeichern. Dafür kann in der Implementierung der zu diesem Zeitpunkt nicht mehr relevante Vektor $v^{(0)}$ verwendet werden. Es ist kein zusätzlicher Speicherplatz notwendig.

Auf dem Feingitter erfolgt dementsprechend an zwei Stellen eine Überprüfung des Residuums, sowohl nach der Vorglättung als auch der Nachglättung, und keine direkte Überprüfung der Iterierten.

4.1.3 Detektion

In den beiden vorangegangenen Abschnitten sprechen wir von Fehlerdetektionen anhand von zuvor generierten Vergleichswerten ohne diese näher zu erläutern. Ziel ist es hierbei eine skalare Größe für die Vektoren zu erzeugen um mit minimalem Speicheraufwand eine Detektion und Lokalisierung von Anomalien, die durch Fehler entstanden sind, zu ermöglichen. In diesem Abschnitt werden wir genauer erläutern, wie dieser Vergleichswert generiert werden kann.

Eine klassische Größe zur Konvergenzkontrolle ist die 2-Norm

$$\|x\|_2 = \sqrt{\sum_{i=1}^{N} x_i^2}$$

eines N-dimensionalen Vektors. In unserem Algorithmus wollen wir, durch eine komponentenweise Überprüfung, neben der Fehlerdetektion auch eine -lokalisierung erhalten. Die 2-Norm ist dafür insofern ungeeignet, da

$$\max_{i=1,\dots,N} |x_i| \leq \|x\|_2 \leq \sqrt{N} \max_{i=1,\dots,N} |x_i| \tag{4.1}$$

und dadurch der Wert, im Vergleich zu dem maximalen Eintrag, durchaus in einem großen Spektrum liegen kann. Selbst das Auftreten von nur wenigen Komponenten, die sich in der selben Größenordnung wie der maximale Eintrag befinden, sorgt für eine große Überschätzung dieses Wertes.

Daher verwenden wir zur Generierung der Vergleichswerte anstelle der üblicherweise verwendeten 2-Norm die Maximumsnorm

$$\|x\|_\infty = \max_{i=1,\dots,N} |x_i|.$$

Aufgrund der Norm-Äquivalenz (vgl. Gleichung 4.1) gelten weiterhin alle Überlegungen zu den Konvergenzeigenschaften aus den vorherigen Kapiteln.

Wird nun in der folgenden Iteration die entsprechende Stelle im Algorithmus erreicht, so wird jede Komponente des neuen Vektors mit dem Vergleichswert abgeglichen. Wird der Vergleichswert, welcher eine obere Schranke darstellen soll, betragsmäßig überschritten, so ist davon auszugehen, dass die Komponente fehlerbehaftet ist.

Eine geeignete problemspezifische Initialisierung der Werte ist notwendig, um auch innerhalb der ersten Iteration einen Vergleich zu ermöglichen. Alternativ kann die erste Iteration auch ohne Detektion erfolgen. Es ist jedoch ratsam eine Initialisierung durchzuführen, um zumindest das Auftreten von Extremsituationen wie `NaN` oder `inf` zu erkennen.

4.2 Das modifizierte Mehrgitterverfahren

Mit Hilfe der vorgestellten Mechanismen lässt sich nun das Mehrgitterverfahren erweitern und die Stabilität bzgl. Fehlern im Glättungsprozess erhöhen. Wir stellen dazu exemplarisch in Algorithmus 4.1 den modifizierten V-Zyklus dar. Die Ergänzungen und Veränderungen gegenüber des klassischen V-Zyklus des FAS-Ansatzes (vgl. Algorithmus 2.2) sind durch rote Schriftfarbe hervorgehoben.

Die Überprüfung erfolgt an drei Stellen. In Zeile 6 wird nach der Anwendung des Glättungsoperators das Residuum kontrolliert und gegebenenfalls korrigiert. Für die Überprüfung des Korrekturvektors in Zeile 12 ist es notwendig die Grobgitterkorrektur aufzuspalten und den Korrekturvektor vor der Prolongation zwischenzuspeichern. Da die Kontrolle der Korrektur immer nach der Lösung des untergeordneten Mehrgitterproblems erfolgt, ist es auf dem Feingitter notwendig eine gesonderte Überprüfung durchzuführen (vgl. Zeile 15-17). Hier ist, wie in Kapitel 4.1 beschrieben, erneut eine Kontrolle analog zur Restriktionsphase, d.h. einem Test des Residuums, möglich.

Zur Beschreibung der Detektions- und Reparaturmechanismen sind zwei zusätzliche Funktionen notwendig. Die Detektion der fehlerhaften Komponenten erfolgt durch den Aufruf von `detect(`v, cmp`)` mit einem Vektor v und einem Vergleichswert cmp. Als Ausgabe liefert diese Funktion alle Komponenten, deren Werte betragsmäßig größer sind als der Vergleichswert oder mit `NaN` übereinstimmen, ergänzt um die Schicht der im Gitter anliegenden Freiheitsgrade, welche durch die Matrixstruktur bestimmbar sind. Des Weiteren muss der Vergleichswert zu einem Vektor v durch den Aufruf der Funktion `calc_cmp(`v`)` erzeugt werden. Diese liefert in unserem Fall, wie in Abschnitt 4.1.3 beschrieben, den betragsmäßig größten Eintrag von v.

Mit Hilfe dieser Funktionen ist es nun möglich die im modifizierten Mehrgitterverfahren verwendeten Detektions- und Reparaturmechanismen zu erläutern. Algorithmus 4.2 beschreibt diesen für die Prolongationsphase. Initial erfolgt eine Detektion der fehlerhaften Komponenten, welche ausgehend vom aktuellen Vergleichswert bestimmt werden. Die entsprechenden Indizes werden in der Menge $\mathcal{I}$ gespeichert. Für den Fall, dass $\mathcal{I} \neq \emptyset$

Algorithmus 4.1: ABFT V-Zyklus

Aufruf :
`ABFT_MG`$(k, \mathcal{A}, b, u^{(0)})$
Parameter :
ν - Anzahl der Vorglättungsschritte
μ - Anzahl der Nachglättungsschritte
L - Feingitterlevel
Eingabe :
k - Gitterlevel
$\mathcal{A}$ - Menge der Systemmatrizen $\mathcal{A} = \{A_0, \ldots, A_k\}$
b - rechte Seite auf Level k
$u^{(0)}$ - initiale Lösungsapproximation auf Level k
Ausgabe :
u - Lösungsapproximation des Systems $A_k u = b$

1 **if** $k = 0$ **then**
2 | $u = A_0^{-1} b$ // Lösen des Grobgitterproblems
3 **else**
4 | $u^{(\nu)} = \mathcal{S}^{\nu}\left(u^{(0)}, b\right)$ // ν-faches Vorglätten
5 | $r_k = b - A_k u^{(\nu)}$ // Berechnung des Residuums
6 | `detect_and_repair_res`(r_k, k) // Überprüfung auf Fehler
7 | $r_{k-1} = \mathbf{R}_k^{k-1} r_k$ // Restriktion des Residuums
8 | $v^{(0)} = \mathbf{I}_k^{k-1} u^{(\nu)}$ // Restriktion der Lösung
9 | $r_{k-1} = r_{k-1} + A_{k-1} v^{(0)}$ // Berechnung der rechten Seite
10 | $v =$ `ABFT_MG`$(k-1, \mathcal{A}, r_{k-1}, v^{(0)})$ // rekursives Lösen des Grobgitterdefektproblems
11 | $c = v - v^{(0)}$ // Berechnung der Korrektur
12 | `detect_and_repair_cor`$(c, k-1)$ // Überprüfung auf Fehler
13 | $u^{(\nu+1)} = u^{(\nu)} + \mathbf{P}_{k-1}^{k}(c)$ // Grobgitterkorrektur
14 | $u = \mathcal{S}^{\mu}\left(u^{(\nu+1)}, b\right)$ // μ-faches Nachglätten
15 | **if** $k = L$ **then**
16 | | `detect_and_repair_res`$(b - A_L u, k)$ // Überprüfung auf Fehler
17 | **end**
18 **end**

erfolgt eine Reparatur, wie sie in Zeile 2 bis 8 dargestellt wird. Die dem Korrekturvektor zugrundeliegende Näherungslösung $v^{(0)}$ wird auf das nächstgröbere Level restringiert und die Differenz zur dort bestimmten nachgeglätteten Näherungslösung als temporärer neuer Korrekturvektor $\tilde{c}$ ermittelt. Bei dieser Korrektur handelt es sich somit, nach der erneuten Prolongation, um eine auf dem Level k ungeglättete Korrektur. Im Anschluss daran werden die als fehlerhaft detektierten Komponenten des im Algorithmus eingehenden Korrekturvektors c durch die des temporären Vektors $\tilde{c}$ ersetzt. Somit handelt es sich um einen in Teilbereichen auf einem Gitterlevel ungegelätteten Korrekturvektor.

Es ist anzumerken, dass alle Operationen in Zeile 3 bis 5 in einem reduzierten System erfolgen können. Dazu ist es für einige Operationen lediglich notwendig die Indexmenge $\mathcal{I}$ um eine weitere Schicht zu vergrößern bzw. die Ausführung der Matrix-Vektor-Multiplikation auf entsprechende Zeilen zu reduzieren, da vom Vektor $\tilde{c}$ ausschließlich die Komponenten der Indexmenge $\mathcal{I}$ benötigt werden.

Im Anschluss an die Reparatur erfolgt zusätzlich zur Berechnung des neuen Vergleichswertes eine Überprüfung, ob sich dieser im Vergleich zur vorherigen Iteration reduziert hat (vgl. Zeile 11). Für den Fall, dass $tmp > 1$ ist kennzeichnet dies insbesondere, sofern das Verfahren üblicherweise monoton ist, dass die Reparatur nicht ausreichend war. Dies ist z.B. möglich, wenn die Detektion nicht unmittelbar erfolgt, sondern erst auf einem späteren Gitterlevel. Sollte dies der Fall sein, so werden die Vergleichswerte der anderen Gitterlevel im entsprechenden Teilprozess (Prolongation bzw. Restriktion) des Mehrgitterverfahrens skaliert, um im Anschluss Detektionen zu verhindern, die durch das Nichterkennen der vorher auftretenden Störung oder besonderen Eigenschaften des zugrundeliegenden Problems entstanden sind. Das Verfahren wird also in der Folge für eine Prolongations- bzw. Restriktionsphase toleranter.

Für den Fall, dass keine signifikante Störung detektiert wird, erfolgt nur eine Berechnung des neuen Vergleichswertes.

Die Detektion und Reparatur in der Restriktionsphase unterscheidet sich in einigen Komponenten und ist in Algorithmus 4.3 dargestellt. Analog zur Prolongationsphase erfolgt erneut initial eine Detektion der Komponenten deren Werte betragsmäßig größer sind als der entsprechende Vergleichswert. Für den Fall, dass keine Komponente entdeckt wird, wird auch hier nur der neue Vergleichswert berechnet (siehe Zeile 17).

Werden fehlerbehaftete Komponenten im Residuum identifiziert, so werden diese im Folgenden durch die Einträge der ungeglätteten Näherungslösung $u^{(0)}$ ersetzt. Somit entspricht $u^{(\nu)}$ u. U. wieder einer in Teilbereichen auf einem Gitterlevel nicht geglätteten Lösung. Der Korrektur folgt eine Neuberechnung des Residuums und, wie in Algorithmus 4.2, eine Nachskalierung der Vergleichswerte, falls keine Reduktion vorliegt. Erneut ist es möglich, die auftretende Matrix-Vektor-Multiplikation (siehe Zeile 6) auf den Auswirkungsbereich der reparierten Komponenten einzuschränken, um den numerischen Aufwand zu reduzieren. Abschließend ist noch einmal hervorzuheben, dass die Funktion `detect` zusätzlich die anliegenden Freiheitsgrade der Einträge liefert, die betragsmäßig größer als der Vergleichswert sind und somit auch im Falle einer Detektion mittels des

Algorithmus 4.2: Detektions- und Reparaturalgorithmus für die Prolongationsphase.

Aufruf :
`detect_and_repair_cor(`c, k`)`
Parameter :
cmp_cor_k - Vergleichswert
$\mathcal{I}$ - Indexmenge
L - Feingitterlevel
$v^{(0)}$ - zu korrigierende Lösung
u_{k-1} - Lösung vom gröberen Gitter
Eingabe :
c - Korrekturvektor
k - aktuelles Gitterlevel

1 $\mathcal{I} =$ `detect(`c`,`cmp_cor_k`)` // Detektion
2 **if** $\mathcal{I} \neq \emptyset$ **then**
3 $\quad \tilde{v} = \mathbf{I}_k^{k-1} v^{(0)}$ // Restriktion auf nächstgröberes Level
4 $\quad \tilde{c} = u_{k-1} - \tilde{v}$ // Berechnung der neuen Korrektur
5 $\quad \tilde{c} = \mathbf{P}_{k-1}^{k} \tilde{c}$ // Prolongation des neuen Korrekturvektors
6 $\quad$ **for** $i \in \mathcal{I}$ **do**
7 $\quad\quad c(i) = \tilde{c}(i)$ // Ersetzen der fehlerhaften Komponenten
8 $\quad$ **end**
9 $\quad cmp =$ `calc_cmp(`c`)` // Berechnung des neuen Vergleichswertes
10 $\quad tmp = \frac{cmp}{cmp\_cor_k}$ // Änderung des Vergleichswertes
11 $\quad$ **if** $tmp > 1$ // Neuskalierung
12 $\quad$ **then**
13 $\quad\quad$ **for** $i \in \{0, \ldots, L\} \backslash \{k\}$ **do**
14 $\quad\quad\quad cmp\_cor_k = cmp\_cor_k * 10 * tmp$
15 $\quad\quad$ **end**
16 $\quad$ **end**
17 $\quad cmp\_cor_k = cmp$ // Abspeichern des neuen Vergleichswertes
18 **else**
19 $\quad cmp\_cor_k =$ `calc_cmp(`c`)` // Abspeichern des neuen Vergleichswertes
20 **end**

Residuums alle möglicherweise für diesen Fehler verantwortlichen Komponenten des Lösungsvektors liefert. Die Hinzunahme weiterer Schichten ist eine zusätzliche Option, um möglicherweise eine großflächigere Reparatur zu erhalten. In diesem Fall reicht die explizite Detektion einer Komponente aus, um einen Patch aus bis zu 25 Komponenten zu korrigieren (Q_1).

Die verwendeten Vergleichswerte sollten, wie bereits in Abschnitt 4.1.3 beschrieben, in der Praxis initialisiert werden, da zu Beginn keine entsprechenden Größen vorliegen. Diese sollten je nach Phase unterschiedlich gewählt werden. Für die Restriktionsphase bietet sich eine Berechnung auf Basis des Startresiduums an, wohingegen in der Prolongationsphase u. U. eine Abschätzung auf Basis der Norm der rechten Seite erfolgen kann, denn mit der Wahl des Nullvektors als Startlösung $u^{(0)} = \bar{0}$ gilt für den Korrekturvektor $c = u - u^{(0)}$

$$\begin{aligned} & Ac = f \\ \Rightarrow \quad & \|c\| \leq \|A^{-1}\|\|f\| \end{aligned}$$

und somit aufgrund der Eigenschaften des Operators A

$$\|c\| \leq \|f\|.$$

Die genaue Analyse der Wahl des Startwertes vernachlässigen wir innerhalb dieser Arbeit und realisieren die Initialisierung durch die Wahl eines hinreichend großen Wertes, so dass zumindest starke Störungen detektiert werden.

Algorithmus 4.3: Detektions- und Reparaturalgorithmus für die Restriktionsphase.

Aufruf :
`detect_and_repair_res(`r, k`)`
Parameter :
cmp_res_k - Vergleichswert
$\mathcal{I}$ - Indexmenge
L - Feingitterlevel
$u^{(\nu)}$ - nachgeglättete Lösung
$u^{(0)}$ - ungeglättete Lösung
Eingabe :
r - Residuumsvektor
k - aktuelles Gitterlevel

```
1  I = detect(r,cmp_res_k)                 // Detektion
2  if I ≠ ∅ then
3  |  for i ∈ I do
4  |  |  u^(ν)(i) = u^(0)(i)               // Ersetzen der fehlerhaften Komponenten
5  |  end
6  |  r = b − A_k u^(ν)(i)                 // Neuberechnung des Residuums
7  |  cmp = calc_cmp(r)                    // Berechnung des neuen Vergleichswertes
8  |  tmp = cmp / cmp_res_k                // Änderung des Vergleichswertes
9  |  if tmp > 1                           // Neuskalierung
10 |  then
11 |  |  for i ∈ {0,...,L}\{k} do
12 |  |  |  cmp_res_k = cmp_res_k * 10 * tmp
13 |  |  end
14 |  end
15 |  cmp_res_k = cmp                      // Abspeichern des neuen Vergleichswertes
16 else
17 |  cmp_res_k = calc_cmp(r)              // Abspeichern des neuen Vergleichswertes
18 end
```

4.3 Numerische Experimente

Im Folgenden untersuchen wir die Funktionsfähigkeit des in Kapitel 4.2 beschriebenen Ansatzes für algorithmenbasierte Fehlertoleranz in der Glättungsphase des Mehrgitterverfahrens an vier beispielhaften Testproblemen. Als ersten Test betrachten wir dabei das der Konstruktion des geometrischen Mehrgitterverfahrens zugrundeliegende Poisson-Problem

$$-\Delta u = f. \qquad (\texttt{poisson})$$

Bei den weiteren Problemen handelt es sich um ein anisotropes Diffusions-Problem (`andi`), ein Diffusions-Konvektions-Problem (`dico`) und ein anisotropes Diffusions-Konvektions-Reaktions-Problem (`andicore`).

$$-\nabla \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0.15 \end{pmatrix} \nabla u = f \qquad (\texttt{andi})$$

$$-\nabla \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0.8 \end{pmatrix} \nabla u + \begin{pmatrix} 1.0 \\ 0.7 \end{pmatrix} \cdot \nabla u = f \qquad (\texttt{dico})$$

$$-\nabla \cdot \begin{pmatrix} 1.2 & -0.7 \\ -0.4 & 0.9 \end{pmatrix} \nabla u + \begin{pmatrix} 0.4 \\ -0.2 \end{pmatrix} \cdot \nabla u + 0.3u = f \qquad (\texttt{andicore})$$

Alle Probleme werden dabei auf dem Einheitsquadrat $[0,1]^2$ gelöst, wobei die rechte Seite f so konstruiert wird, dass die Lösung jeweils durch

$$u = \sin(\pi x)\sin(\pi y)$$

gegeben ist. Insbesondere erhalten wir somit durch die Vorgabe der Funktionswerte auf dem Rand einen homogenen Dirichletrand.

Für die numerische Analyse verwenden wir eine Gitterhierarchie aus 8 Gittern beginnend mit einer Schrittweite von $h = 0.25$ und anschließender Verfeinerung durch Schrittweitenhalbierung und Zwei-Level-Nummerierung. Die Nummerierung der Level beginnt dabei mit 0 auf dem Grobgitter und endet bei 7 auf dem Feingitter. Außerdem verwenden wir auf jedem Level vier Vor- und Nachglättungsschritte mit einem um den Faktor 0.7 gedämpften Jacobiverfahren, damit auch im Fall der Anisotropie eine gute Konvergenzgeschwindigkeit erreicht wird. Mit Q_1-Finite Elementen erhalten wir somit Probleme bestehend aus 263 169 Unbekannten.

Beim Mehrgitterverfahren verwenden wir den einfachen V-Zyklus und als Abbruchkriterium eine relative Residuumsreduktion um 10^{-10} in der 2-Norm. Zum Lösen des Grobgitterproblems verwenden wir das BiCGStab-Verfahren und ebenfalls eine rel. Residuumsreduktion um 10^{-10} als Abbruchkriterium.

Der Gittertransfer wird, wie bereits in Kapitel 2.2 beschrieben, durch bilineare Interpolationen bzw. natürliche Injektionen durchgeführt.

Wird nach 40 Iterationen keine Konvergenz erzielt, so gehen wir davon aus, dass das Verfahren divergiert.

Die Initialisierung der Vergleichswerte führen wir innerhalb dieses Kapitels mit dem Wert 100 durch. Dieser hat sich als hinreichend geeignet ergeben, um starke Störungen in der ersten Iteration zu detektieren ohne fälschlicherweise positive Detekionen zu bewirken.

4.3.1 Fehlergenerierung

Wie bereits angesprochen konzentrieren wir uns in diesem Kapitel auf die Detektion von Fehlern, die innerhalb der Glättungsphase geschehen. Demzufolge werden wir die Fehlergenerierung ebenso auf diese Phase beschränken. Am Ende einer jeden Anwendung des Glätters erzeugen wir mit einer Wahrscheinlichkeit von 1% einen Bitflip in einer Komponente des Lösungsvektors. Die Stelle innerhalb der Binär-Darstellung der Komponente, an der der Bitflip erzeugt wird, ist dabei ebenfalls zufällig. Alle zufälligen Ereignisse werden dabei mit Hilfe eines festen Seed-Wertes generiert, so dass die Auswirkung der Fehler untersucht werden kann. Wir vergleichen insbesondere den Konvergenzverlauf des korrigierten fehlerbehafteten Mehrgitterverfahrens mit dem klassischen Ansatz, wobei die Fehler zu den selben Zeitpunkten an den identischen Stellen erzeugt werden. Durch die möglicherweise unterschiedlichen Konvergenzverläufe ist es jedoch nicht möglich zu garantieren, dass es sich um identische Manipulationen handelt.

Es ist anzumerken, dass die Erzeugung eines Fehlers vor der letzten Anwendung des Glätters dazu führt, dass sich der Fehler ausbreitet. Die Art und Weise wie dieser sich ausbreitet ist abhängig vom verwendeten Glätter. In der Regel wird bei jedem Glätter ein Residuum berechnet, so dass die Systemmatrix auf den Lösungsvektor angewendet wird. Dies führt dazu, dass sich der Fehler mindestens auf dem gesamten Träger des Freiheitsgrades ausbreitet, was bedeutet, dass Fehler in frühen Glättungsschritten ein besonders großes Einflussgebiet haben. Das von uns verwendete Jacobi-vorkonditionierte Richardson-Verfahren führt dazu, dass der Fehler sich außer durch die Residuumsberechnung nicht weiter verbreitet. Folgende Anwendungen des Glätters streuen den Fehler jedoch weiter aus.

Um auch der unkorrigierten fehlerbehafteten Variante (dem klassischen Verfahren) die Konvergenz zu ermöglichen, wird ab dem Zeitpunkt zu dem das korrigierte Mehrgitterverfahren konvergiert ist, dort kein weiterer Fehler erzeugt. Diese Grenze ist abhängig von den generierten Fehlern und somit u. U. für jede einzelne Testsituation unterschiedlich. Diese Annahme stellt die klassische Variante besser dar, als zu erwarten ist, da im Anwendungsfall nicht davon auszugehen ist, dass ab einer bestimmten Iteration keine Fehler mehr entstehen können.

Unsere Tests sind sehr davon abhängig wie und wo Fehler erzeugt werden. Um diesen zufälligen Effekt zu verringern und aussagekräftigere Resultate zu erhalten wird jedes Testproblem 100 mal ausgeführt. In jedem Fall treten unterschiedliche Fehler auf. Dies

und die relativ hohe Fehlerwahrscheinlichkeit von 1% sollen zeigen, ob der Ansatz auch in Extremsituationen zufriedenstellend funktioniert.

4.3.2 Testreihen

In diesem Abschnitt wollen wir nun die Resultate der einzelnen Testreihen betrachten und analysieren. Wir gehen dabei nicht auf einzelne Ausführungen ein, sondern betrachten ausschließlich den gesamten Datensatz, um eine möglichst allgemeine Aussage über die Funktionsfähigkeit zu treffen. Im darauf folgenden Abschnitt 4.3.3 werden wir einige Fälle detaillierter untersuchen. Eine Auflistung einiger Statistiken zu den einzelnen Tests findet sich in Kapitel A.1 im Anhang.

Wir geben zu jeder Testreihe eines Problems die durchschnittlichen Anzahlen an Iterationszahlen der beiden Verfahren, erzeugten Fehlern, detektierten Störungen und fehlerhaften Detektionen an. Darüber hinaus stellen wir in entsprechenden Abbildungen die Verteilungen grafisch dar. Wir bezeichnen das konstruierte Verfahren im Folgenden als ABFT-Verfahren und den unmodifizierten Ansatz als klassisches Verfahren.

Das `poisson`-Problem erreicht im fehlerfreien Fall nach sieben Iterationen das Abbruchkriterium. Auch das erweiterte fehlertolerante Verfahren konvergiert nach sieben Iterationen, ohne dass einer der Detektionsmechanismen fälschlicherweise einen Fehler erkennt. Bei den Tests (siehe Tabelle A.1) handelt es sich nun um Ausführungen der Verfahren bei Entstehung von Fehlern, nach dem in Abschnitt 4.3.1 vorgestellten Prinzip.

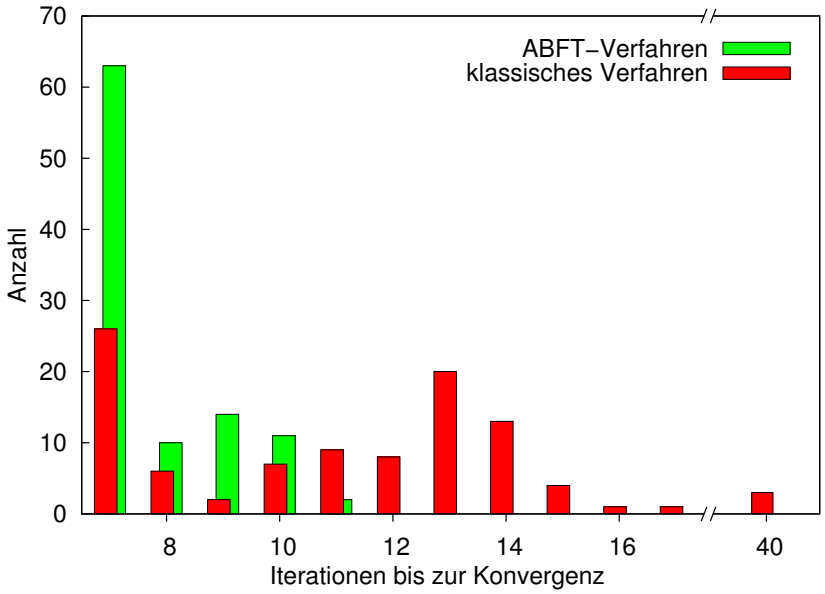

Abbildung 4.1: Anzahl der Tests zum `poisson`-Problem bei dem das Verfahren die entsprechenden Iterationszahlen bis zum Erreichen des Abbruchkriteriums benötigt.

In den hundert durchgeführten Tests werden im Schnitt 3.82 Fehler generiert. Dabei treten sowohl Situationen auf in denen nur ein einziger Fehler auftritt, als auch Situationen in denen in einem V-Zyklus gleich mehrere Fehler erzeugt werden. Die entsprechende Verteilung ist in Abbildung 4.2 (links) dargestellt. Die y-Achse entspricht der Anzahl der Tests in denen die Ereignisse mit entsprechender Häufigkeit (x-Achse) auftreten. Die erzeugten Fehler führen dazu, dass das klassische Verfahren, wenn es konvergiert, im Durchschnitt 10.82 statt 7 Iterationen bis zum Erreichen des Abbruchkriteriums benötigt. In drei Fällen divergiert das Verfahren sogar. Dies ist darauf zurückzuführen, dass durch die erzeugten Fehler im Laufe des Iterationsprozesses `NaN` oder `inf` entstehen, welche dann zur Divergenz bzw. zum Abbruch führen. In Abbildung 4.1 ist die genaue Verteilung der bis zur Konvergenz benötigten Iterationszahlen dargestellt. Dabei ist ein Verfahren besser, je weiter links sich der Großteil der ausgeführten Tests befindet. Im Gegensatz zum klassischen Verfahren konvergiert das ABFT-Verfahren immer und zwar bereits nach durchschnittlich 7.79 Iterationen.

Das fehlertolerante Verfahren detektiert dabei im Durchschnitt 1.89 Fehler. Andere Störungen sind zu den entsprechenden Zeitpunkten entweder so klein, dass sie keinen Einfluss auf das Konvergenzverhalten zeigen oder aber ihre unmittelbare Auswirkung ist so gering, dass der Detektionsmechanismus nicht greift. Sollte der Fehler dennoch große Auswirkungen haben, so führt dies in der Regel zu einer fehlerhaften Detektion im späteren Verlauf. In der untersuchten Testreihe zum `poisson`-Problem treten im Schnitt 0.61 fehlerhafte Detektionen auf (siehe Abbildung 4.2, rechts). Diese führen in der Regel lediglich zu einer überflüssigen Reparatur, was dem Verzicht einer Glättung auf einem Gitterlevel in einem kleinen Teilbereich entspricht. Der Einfluss auf die Konvergenz ist daher gering. Insbesondere führt das Nicht-Detektieren eines relevanten Fehlers in den meisten Fällen ausschließlich zu einer weiteren fehlerhaften Detektion. Im Abschnitt 4.3.3 werden wir dazu einige Fälle näher analysieren.

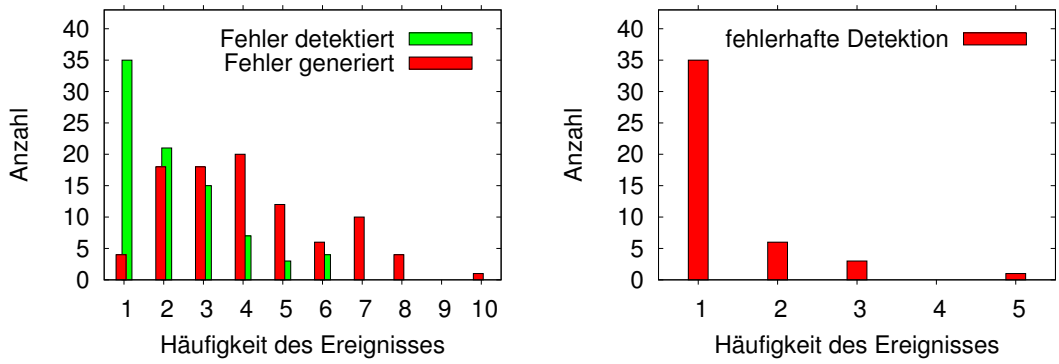

Abbildung 4.2: Absolute Anzahl der Tests (y-Achse) in denen die Ereignisse mit entsprechender Häufigkeit (x-Achse) beim `poisson`-Problem auftreten.

Bei dem bisher betrachteten `poisson`-Problem handelt es sich um das dem geometrischen Mehrgitterverfahren zugrundeliegende Modellproblem. Daher betrachten wir mit dem `andi`-Problem nun eine entsprechend schwieriger zu lösende anisotrope Variante des ersten Testproblems. Hierbei wird im fehlerfreien Fall die Konvergenz erst nach 18 Iterationen erreicht. Der fehlerfreie Fall führt auch hier zu keinen fehlerhaften Detektionen im fehlertoleranten Algorithmus. Statistiken zu den einzelnen Tests befinden sich im Anhang in Tabelle A.2.

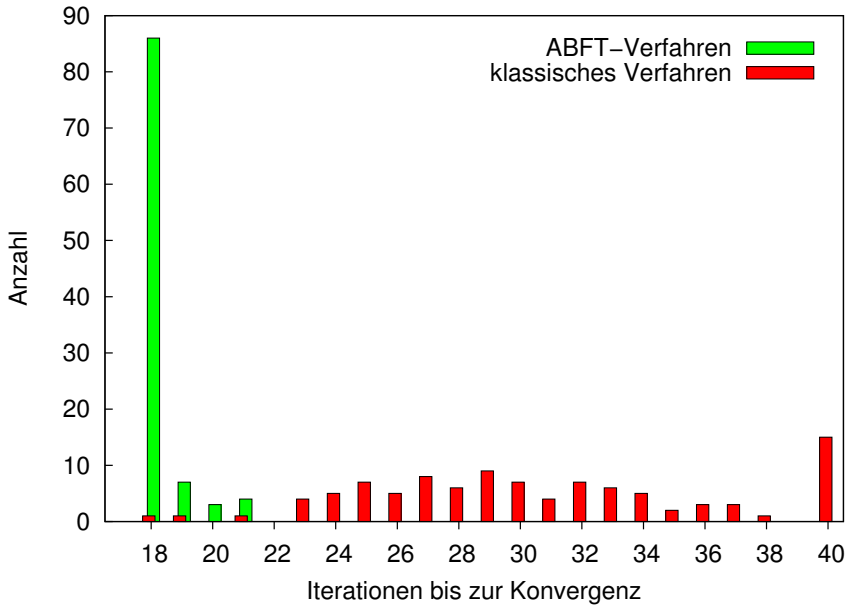

Abbildung 4.3: Anzahl der Tests zum `andi`-Problem bei dem das Verfahren die entsprechenden Iterationszahlen bis zum Erreichen des Abbruchkriteriums benötigt.

Durchschnittlich werden 10.24 Fehler erzeugt, welche beim modifizierten Verfahren im Schnitt in 4.94 positiven Detektionen (siehe Abbildung 4.4, links) resultieren. Die erzeugten Fehler führen beim klassischen Ansatz in 15 Fällen zur Divergenz und allgemein zu einem Anstieg der Iterationszahlen auf 29.09. Im Gegensatz dazu konvergiert der fehlertolerante Ansatz bereits nach 18.25 Iterationen und erreicht in den meisten Fällen das Abbruchkriterium bereits, wie im fehlerfreien Fall, nach 18 Iterationen (vgl. Abbildung 4.3). Die 0.49 fälschlicherweise positiven Detektionen verteilen sich wie in Abbildung 4.4 (rechts) dargestellt. Es treten somit weniger auf, als beim `poisson`-Problem, obwohl sowohl mehr Fehler erzeugt, als auch mehr Iterationen durchgeführt werden. Dies hängt möglicherweise mit der allgemein langsameren Konvergenz zusammen, wodurch der verwendete Detektionsmechanismus besser funktioniert, da die Vergleichswerte das erwartete Verhalten besser repräsentieren.

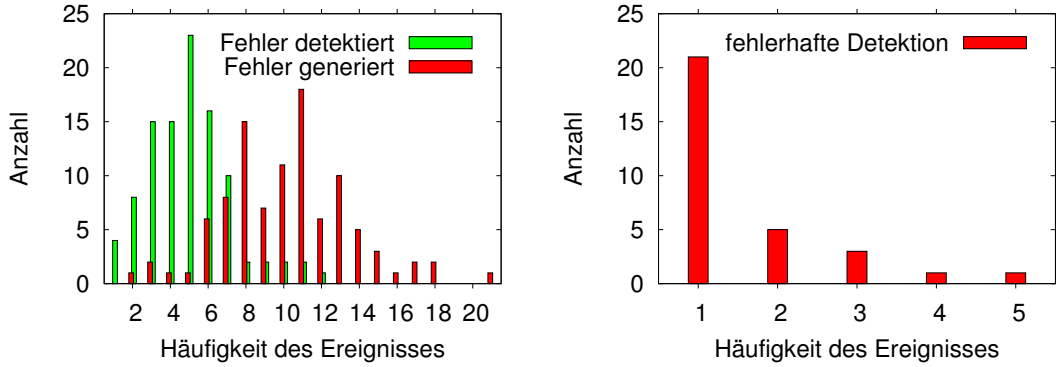

Abbildung 4.4: Absolute Anzahl der Tests (y-Achse) in denen die Ereignisse mit entsprechender Häufigkeit (x-Achse) beim `andi`-Problem auftreten.

Die dritte Testreihe nimmt Abstand vom reinen Diffusions-Problem und erweitert die Problemstellung um einen konvektiven Anteil zum beschriebenen `dico`-Problem. Dieses konvergiert im fehlerfreien Fall bereits nach 8 Iterationen und somit wieder wesentlich schneller als das `andi`-Problem. Erneut treten im ungestörten Fall keine fehlerhaften Detektionen auf und die Iterationszahl bleibt unverändert. Details zu den einzelnen Ausführungen der fehlerbehafteten Fälle finden sich in Tabelle A.3 im Anhang.

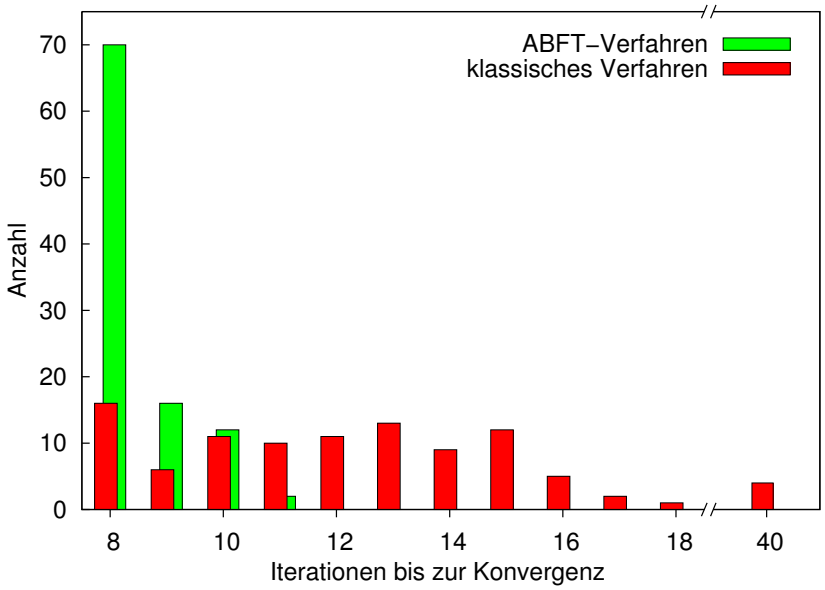

Abbildung 4.5: Anzahl der Tests zum `dico`-Problem bei dem das Verfahren die entsprechenden Iterationszahlen bis zum Erreichen des Abbruchkriteriums benötigt.

In Abbildung 4.6 (links) ist zu erkennen, dass pro Test im Schnitt 4.92 Fehler generiert

werden, wodurch das klassische Mehrgitterverfahren durchschnittlich 11.89 Iterationen (vgl. Abbildung 4.5) bis zur Konvergenz benötigt. Dabei ist weiterhin zu beachten, dass ab einer gewissen Iteration, die vom korrigierten Verfahren vorgegeben wird, keine weiteren Fehler erzeugt werden. Nichtsdestotrotz treten auch hier vier Fälle auf, in denen die erzeugten Störungen zu einer Divergenz des Verfahrens führen. Im Gegensatz dazu konvergiert das fehlertolerante Verfahren auch in dieser Testreihe immer und benötigt dafür nur 8.46 Iterationen. Dabei werden mit durchschnittlich 2.01 Detektionen, wovon gerade einmal 0.39 fehlerhaft sind (vgl. Abbildung 4.6), nicht einmal 45% aller Störungen als relevant erkannt.

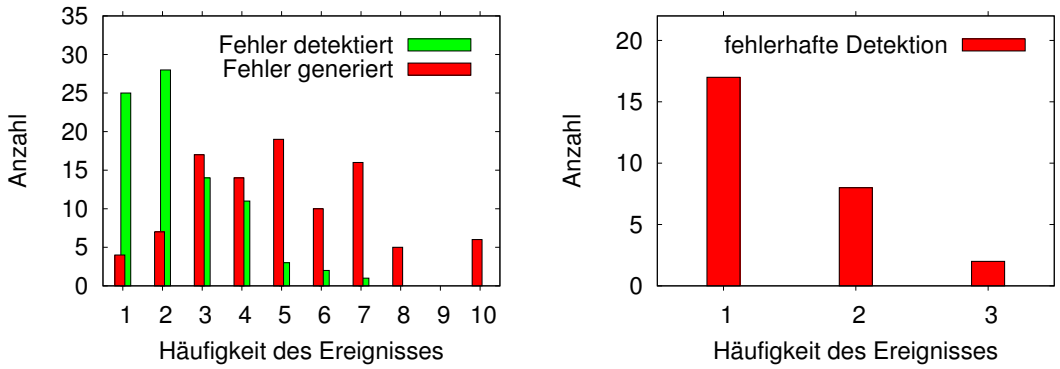

Abbildung 4.6: Absolute Anzahl der Tests (y-Achse) in denen die Ereignisse mit entsprechender Häufigkeit (x-Achse) beim `dico`-Problem auftreten.

Wir schließen unsere erste numerische Auswertung nun mit einem vollständigen anisotropen Diffusions-Konvektions-Reaktions-Problem (`andicore`) ab. Das Abbruchkriterium wird hierbei unter gewöhnlichen Umständen nach 10 Iterationen erreicht. Abermals führt die Verwendung des fehlertoleranten Algorithmus in diesem Fall zu keiner fehlerhaften Detektion.

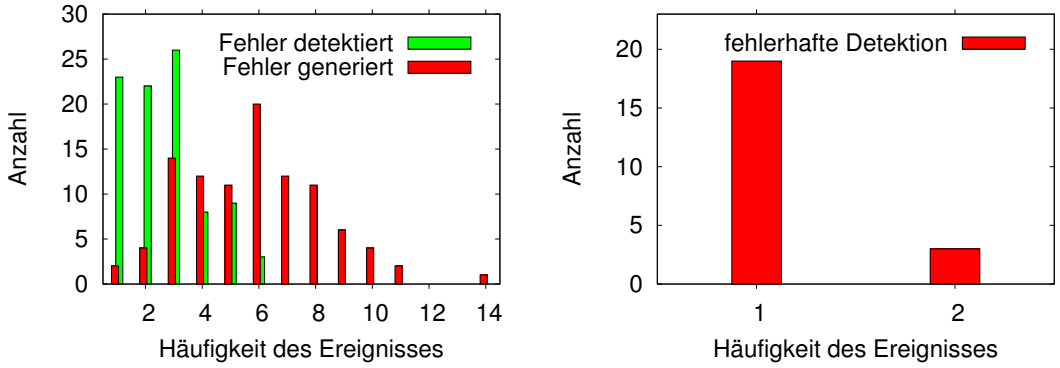

Abbildung 4.7: Absolute Anzahl der Tests (y-Achse) in denen die Ereignisse mit entsprechender Häufigkeit (x-Achse) beim `andicore`-Problem auftreten.

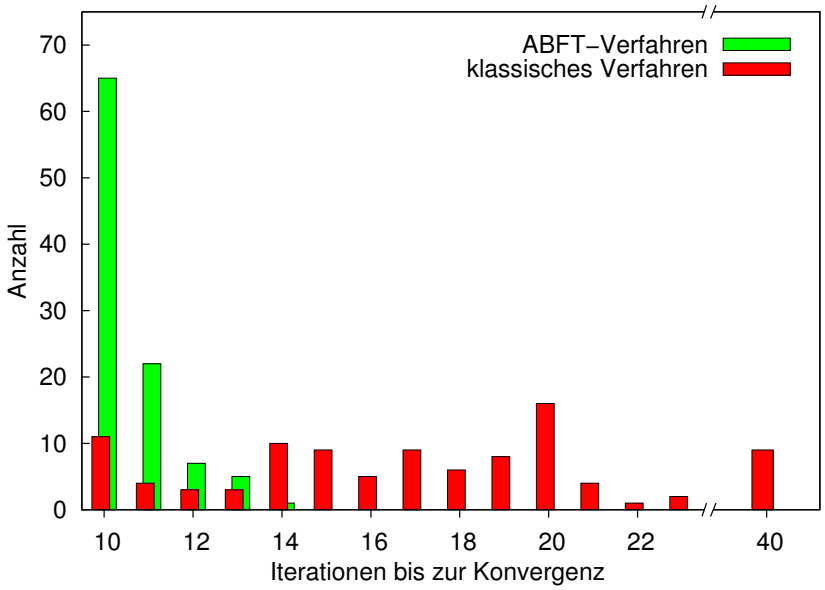

Abbildung 4.8: Anzahl der Tests zum `andicore`-Problem bei dem das Verfahren die entsprechenden Iterationszahlen bis zum Erreichen des Abbruchkriteriums benötigt.

Beim Auftreten der durchschnittlich 5.77 Fehler pro Test (siehe Tabelle A.4) divergiert das klassische Verfahren in 9 Situationen und benötigt im Falle der Konvergenz 16.14 Iterationen, um das Abbruchkriterium zu erreichen (vgl. Tabelle 4.8). Im Gegensatz dazu führt beim modifizierten Ansatz keiner der erzeugten Fehler zur Divergenz und es wird bereits nach 10.55 Iterationen das Abbruchkriterium erfüllt. Somit ist im Vergleich zum fehlerfreien Fall auch hier, gemessen an den durchgeführten Zyklen, nur ein minimaler Mehraufwand nötig. Das Verfahren identifiziert im Schnitt 2.4 der auftretenden Störungen als relevant und es treten durchschnittlich 0.25 fehlerhafte positive Detektionen auf. Die entsprechenden Verteilungen sind in Abbildung 4.7 dargestellt.

4.3.3 Detaillierte Betrachtung

Bisher haben wir allgemein die Funktionsweise des fehlertoleranten Ansatzes anhand eines größeren Datensatzes untersucht. Wir haben ausschließlich durchschnittliche Werte zu den Testproblemen betrachtet und keine genaue Betrachtung der Einflüsse der erzeugten Fehler auf die Verfahren analysiert. Dies wollen wir in diesem Abschnitt nachholen. Im Anhang (siehe Kapitel A.1) befinden sich die Auflistung der ausgeführten Tests mit den wichtigsten Statistiken die zur Untersuchung erforderlich sind. Darüber hinaus werden wir in einzelnen Fällen die erzeugten Fehler und die daraus resultierenden Auswirkungen auf das Verfahren angeben und entsprechend betrachten.

Zu jedem im Detail betrachteten Test geben wir die Konvergenzverläufe des fehlerfreien, des unkorrigierten (klassischen) sowie des modifizierten fehlertoleranten Verfahrens (ABFT-Verfahren) an und kennzeichnen Iterationen in denen mindestens ein Fehler auftritt durch Punkte. Zusätzlich erfolgt eine Auflistung der generierten Fehler mit Angabe der Iteration (**Iter**), der **Phase** (Prolongation oder Restriktion), des Gitterlevels (**Level**) und des Glättungsschrittes nachdem dieser erzeugt wird (**Step**). Darüber hinaus wird der Ausgansgswert (**Vorher**), der durch die Manipulation erzeugte Wert (**Nachher**) und die relative prozentuale Veränderung (%)

$$\frac{|\textbf{Vorher} - \textbf{Nachher}|}{|\textbf{Vorher}|}$$

angegeben. Außerdem listen wir die Anzahl der im modifizierten Verfahren detektierten (#**det**) und dementsprechend korrigierten Komponenten (#**korr**) auf. Im Falle von fehlerhaften Detektionen wird auch der in Algorithmus 4.2 und 4.3 verwendete Wert *tmp* angegeben. Dieser entspricht dem Quotienten aus neuem und altem Vergleichswert. Für den Fall das $tmp > 1$ gilt, wird, wie in den Algorithmen beschrieben, eine Skalierung der Vergleichswerte auf den weiteren Gitterleveln durchgeführt.

Im Folgenden betrachten wir nun also eine Auswahl an Tests der einzelnen Testprobleme, die uns einen Einblick in die Funktionsweise des Verfahrens geben. Dabei betrachten wir insbesondere auch Fälle in denen die Resultate des modifizierten Ansatzes nicht unseren Erwartungen entsprechen, um mögliche Verbesserungsmöglichkeiten zu erkennen.

`poisson`-Problem Eingehend betrachten wir drei Fälle aus der ersten Testreihe. In Test 013 verhindert der Reparaturmechanismus die Divergenz des Verfahrens. Ausreichend dafür ist es den in der Prolongationsphase der fünften Iteration auf Gitterlevel 7 erzeugten Fehler (vgl. Tabelle 4.1) zu korrigieren. Hier wird nach dem ersten Glättungsschritt das führende Bit in der Binärdarstellung der Komponente manipuliert. Im Folgenden führt dies durch die weiteren Anwendungen des Glättungsoperators zur Verbreitung der Störung und im unkorrigierten Algorithmus infolgedessen zum Abbruch des Verfahrens (durch das Auftreten von `NaN`). In Abbildung 4.9 ist zu erkennen, dass durch diese Reparatur der Störung der Konvergenzverlauf des fehlerfreien Falls wiederhergestellt werden kann.

Zusätzlich zu dieser Reparatur detektiert das Verfahren eine weitere Störung in der Restriktionsphase der siebten Iteration auf Gitterlevel 7 (siehe Tabelle 4.2). Der maximale Eintrag des Residuumsvektors aus der vorherigen Iteration, der als Kontrollwert verwendet wird, befindet sich in einer Größenordnung von 10^{-14}, so dass möglicherweise Rundungsfehler zu der positiven Detektion in der Komponente führen. Üblicherweise wäre bei der verwendeten doppelten Genauigkeit der iterative Prozess bereits beendet. Jedoch führt auch die Durchführung der Reparatur zu keinem nennenswerten Mehraufwand oder einer Verschlechterung der Konvergenz.

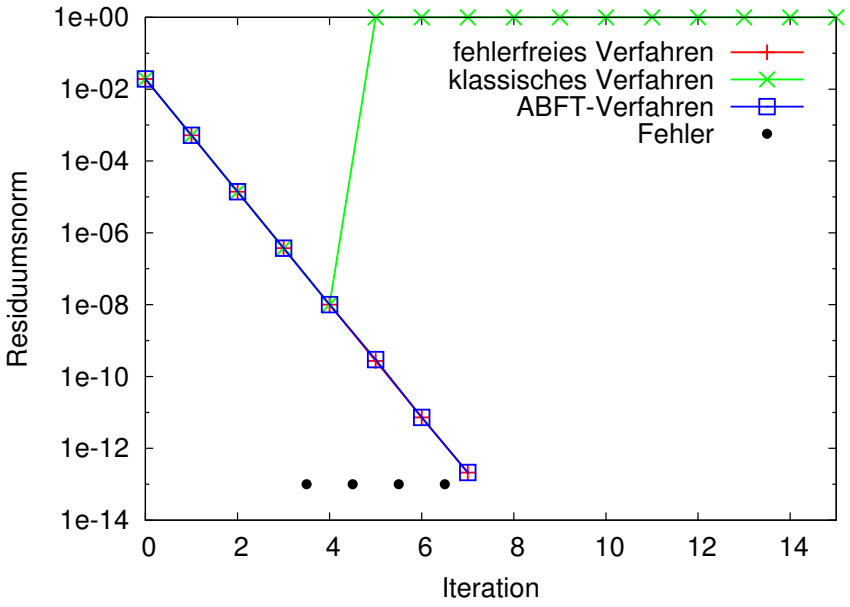

Abbildung 4.9: Konvergenzverlauf der unterschiedlichen fehlerbehafteten Mehrgitterverfahren sowie des fehlerfreien Verfahrens in Test 013 des `poisson`-Problems mit Markierung der Iterationen in denen Fehler generiert werden.

Iter	Phase	Level	Step	Vorher	Nachher	%	#det	#korr
4	Restr.	7	1	$2.815364 \cdot 10^{-2}$		$2.52 \cdot 10^{-11}$		
		2	2	$3.535481 \cdot 10^{-1}$		$1.65 \cdot 10^{-8}$		
5	Prol.	7	1	$4.332649 \cdot 10^{-1}$	$7.788773 \cdot 10^{307}$	`inf`	81	121
6	Restr.	6	1	$7.003321 \cdot 10^{-1}$		$2.60 \cdot 10^{-10}$		
7	Restr.	5	1	$8.835593 \cdot 10^{-1}$		$1.03 \cdot 10^{-10}$		
		2	4	0	$2.172924 \cdot 10^{-311}$	0		

Tabelle 4.1: Übersicht über generierte Fehler in Test 013 des `poisson`-Problems mit Angaben des Zeitpunktes und der prozentualen Auswirkung der Manipulation sowie der Anzahl der dadurch detektierten bzw. korrigierten Komponenten bei Verwendung des ABFT-Verfahrens.

Iter	Phase	Level	#det	#korr	*tmp*
7	Restr.	7	1	9	2.67

Tabelle 4.2: Übersicht über fehlerhafte Detektionen bei Verwendung des ABFT-Verfahrens in Test 013 des `poisson`-Problems mit Angabe der daraus resultierenden Anzahl an Detektionen bzw. Korrekturen sowie des für die Neuskalierung verantwortlichen *tmp*-Wertes.

Eine weitere interessante Situation tritt in Test 028 auf. Dies ist der einzige Fall innerhalb aller Testreihen in dem das korrigierte Verfahren mehr Iterationen als das unkorrigierte benötigt. Dem Konvergenzverlauf in Abbildung 4.10 ist jedoch zu entnehmen, dass dies nur einem geringfügigen Verfehlen des Abbruchkriteriums verschuldet ist.

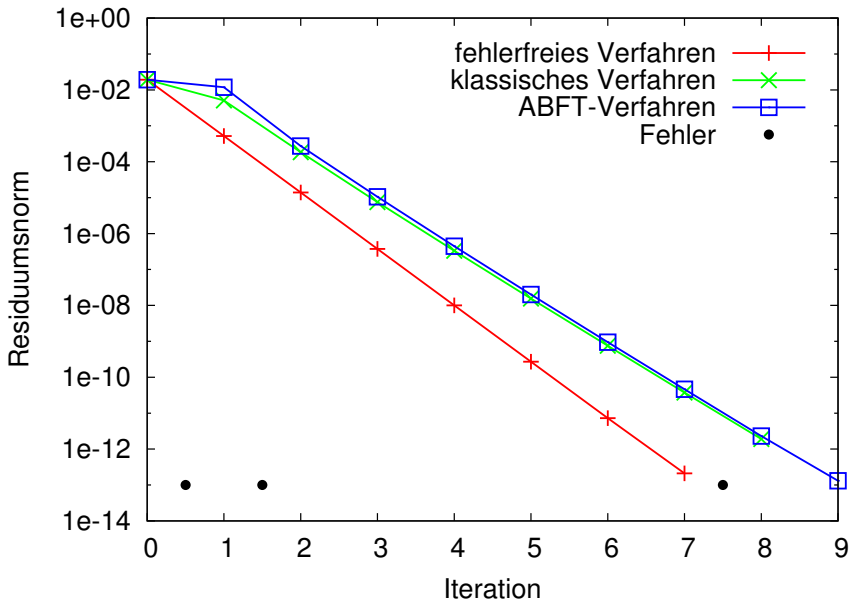

Abbildung 4.10: Konvergenzverlauf der unterschiedlichen fehlerbehafteten Mehrgitterverfahren sowie des fehlerfreien Verfahrens in Test 028 des `poisson`-Problems mit Markierung der Iterationen in denen Fehler generiert werden.

Wie bereits erwähnt, verbessert die Verwendung des fehlertoleranten Verfahrens die Konvergenzeigenschaften nicht. Dies ist darauf zurückzuführen, dass die signifikante Störung innerhalb der ersten Iteration nicht detektiert werden kann. In den Tabellen 4.3 und 4.4 ist zu erkennen, dass diese Störung in der Prolongationsphase der ersten Iteration auf Gitterlevel 6 erfolgt. Die Tatsache, dass diese Störung nicht unmittelbar erkannt wird, führt zu einer verspäteten Detektion auf dem Feingitter. Hier wird nun ein größerer Bereich korrigiert, d.h. insbesondere auf dem Feingitter nicht geglättet, was dazu führt, dass das Residuum in diesem Fall sogar schlechter ist, als in der unkorrigierten Situation.

Die fehlerhafte Detektion macht sich im Mechanismus dadurch bemerkbar, dass der Wert *tmp* größer als Eins ist. Dies bedeutet (vgl. Algorithmus 4.2 und 4.3), dass für den Vergleichswert die Monotonie nicht erfüllt ist und führt somit zu einer Skalierung der folgenden Vergleichswerte. Dadurch wird sichergestellt, dass im Folgenden alle Glättungsschritte auf den gröberen Leveln in der Restriktionsphase akzeptiert werden.

Iter	Phase	Level	Step	Vorher	Nachher	%	#det	#korr
1	Restr.	1	3	$1.494829 \cdot 10^{-1}$		$3.99 \cdot 10^{-5}$		
	Prol.	6	2	$2.977883 \cdot 10^{-1}$	$3.602883 \cdot 10^{-1}$	$2.10 \cdot 10^{1}$		
2	Restr.	1	4	$3.474228 \cdot 10^{-1}$	$3.552353 \cdot 10^{-1}$	$2.25 \cdot 10^{0}$		
8	Prol.	1	4	0	$2.529616 \cdot 10^{-321}$	0		

Tabelle 4.3: Übersicht über generierte Fehler in Test 028 des `poisson`-Problems mit Angaben des Zeitpunktes und der prozentualen Auswirkung der Manipulation sowie der Anzahl der dadurch detektierten bzw. korrigierten Komponenten bei Verwendung des ABFT-Verfahrens.

Iter	Phase	Level	#det	#korr	*tmp*
1	Prol.	7	181	249	$9.88 \cdot 10^{1}$

Tabelle 4.4: Übersicht über fehlerhafte Detektionen bei Verwendung des ABFT-Verfahrens in Test 028 des `poisson`-Problems mit Angabe der daraus resultierenden Anzahl an Detektionen bzw. Korrekturen sowie des für die Neuskalierung verantwortlichen *tmp*-Wertes.

In Test 083 treten mit fünf, innerhalb der Testreihe zum `poisson`-Problem, vergleichsweise viele fehlerhaften Detektionen auf. Die Betrachtung des Konvergenzverlaufes in Abbildung 4.11 macht dennoch deutlich, dass die Verwendung des modifizierten Algorithmus einen Vorteil darstellt. Zwar kann der Fehler innerhalb der ersten Iteration nicht detektiert werden und führt erneut zu einer fehlerhaften Detektion und Reparatur, jedoch wird der im klassischen Fall zum (Quasi-)Neustart führende Fehler in der achten Iteration korrekterweise detektiert und korrigiert, so dass nur 10 statt 15 Iterationen bis zum Erreichen des Abbruchkriteriums benötigt werden.

Ein Blick auf die erzeugten Fehler und Korrekturen in Tabelle 4.5 sowie die fehlerhaften Detektionen und ihre Auswirkung in Tabelle 4.6 erlaubt eine detailliertere Analyse. Der in der ersten Iteration erzeugte Fehler wird erneut erst verspätet auf dem Feingitter erkannt. Dies führt zu einer überflüssigen Korrektur auf dem Feingitter und demzufolge zum Ausbleiben der Glättung in einigen Komponenten. Dies erklärt auch hier die Tatsache, dass im korrigierten Fall die Residuumsnorm größer ist, als im unkorrigierten (vgl. Abbildung 4.11, Iteration 1). Die ausbleibende unmittelbare Detektion ist auf die Initialisierung des Vergleichswertes zurückzuführen. Eine Initialisierung mit dem Wert 100, wie wir sie durchführen, reicht lediglich aus, um stark ausgeprägte Störungen zu detektieren. Eine problemspezifische Wahl könnte hier Abhilfe schaffen.

Anders verhält sich die Störung in Iteration 5. Signifikant ist dabei die Manipulation in der Prolongationsphase nach dem ersten Glättungsschritt auf Level 4. Hier erfolgt eine Veränderung der Komponenten um ca. 0.24%, die nicht detektiert wird. Erst auf dem Feingitter schlägt der Detektionsmechanismus an. Infolge der späten Detektion hat sich der Fehler zu diesem Zeitpunkt bereits so weit ausgebreitet, dass 713 Komponenten als fehlerhaft erkannt und sogar 837 Komponenten repariert werden. Die Detektion löst des

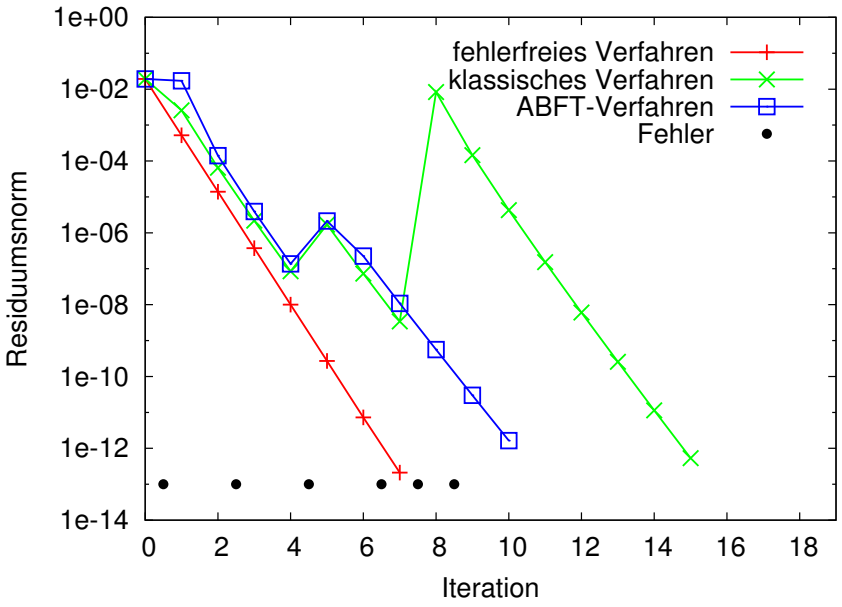

Abbildung 4.11: Konvergenzverlauf der unterschiedlichen fehlerbehafteten Mehrgitterverfahren sowie des fehlerfreien Verfahrens in Test 083 des `poisson`-Problems mit Markierung der Iterationen in denen Fehler generiert werden.

Weiteren eine Neuskalierung der Vergleichswerte des Residuumsmechanismus aus. Diese ist jedoch nicht ausreichend, um die folgende Restriktionsphase ohne fehlerhafte Folgedetektion ablaufen zu lassen, so dass auf Level 5 erneut korrigiert wird. Zu diesem Zeitpunkt hat dieser weit ausgebreitete Fehler nur Neuskalierungen innerhalb der Restriktionsphase bewirkt, wodurch in der folgenden Prolongationsphase ebenfalls zwei kleine Korrekturen durchgeführt werden. Dabei tritt die zweite Detektion auf, weil der Sprung innerhalb der Vergleichswerte bei der ersten Detektion nicht hinreichend groß für eine Neuskalierung ist. Für dieses mit einer guten Konvergenzrate konvergierende Testproblem ist es daher vermutlich ratsam bereits früher eine Skalierung durchzuführen, um das Auftreten solcher Effekte zu minimieren.

Auch wenn dies einen Schwachpunkt des Ansatzes offenbart, ist als Vorteil festzuhalten, dass sich das Verfahren schlussendlich selbst stabilisiert und konvergiert. Spätestens nach einem Zyklus sind die Folgen einer ausbleibenden Detektion behoben.

Iter	Phase	Level	Step	Vorher	Nachher	%	#det	#korr
1	Prol.	6	4	$3.161274 \cdot 10^{-1}$	$3.239399 \cdot 10^{-1}$	$2.47 \cdot 10^{0}$		
3	Prol.	5	2	$2.738351 \cdot 10^{-1}$		$1.05 \cdot 10^{-11}$		
5	Prol.	4	1	$5.039104 \cdot 10^{-2}$	$5.051311 \cdot 10^{-2}$	$2.42 \cdot 10^{-1}$		
		5	4	$6.872827 \cdot 10^{-2}$		$1.69 \cdot 10^{-7}$		
7	Restr.	7	2	$5.820273 \cdot 10^{-1}$	$2.273544 \cdot 10^{-3}$	$9.96 \cdot 10^{1}$	49	81
9	Restr.	1	1	0	$1.112537 \cdot 10^{-308}$	0		

Tabelle 4.5: Übersicht über generierte Fehler in Test 083 des `poisson`-Problems mit Angaben des Zeitpunktes und der prozentualen Auswirkung der Manipulation sowie der Anzahl der dadurch detektierten bzw. korrigierten Komponenten bei Verwendung des ABFT-Verfahrens.

Iter	Phase	Level	#det	#korr	*tmp*
1	Prol.	7	69	109	$1.73 \cdot 10^{2}$
5	Prol.	7	713	837	$1.06 \cdot 10^{1}$
6	Restr.	5	23	47	$2.58 \cdot 10^{0}$
	Prol.	2	2	12	$5.85 \cdot 10^{-1}$
		3	8	24	$5.01 \cdot 10^{0}$

Tabelle 4.6: Übersicht über fehlerhafte Detektionen bei Verwendung des ABFT-Verfahrens in Test 083 des `poisson`-Problems mit Angabe der daraus resultierenden Anzahl an Detektionen bzw. Korrekturen sowie des für die Neuskalierung verantwortlichen *tmp*-Wertes.

`andi`-Problem Beim zweiten Testproblem treten, insbesondere in Test 127, ebenfalls Situationen mit vergleichsweise vielen fehlerhaften Detektionen auf. Dort wird erneut fünfmal fälschlicherweise eine Korrektur durchgeführt. Jedoch werden, aufgrund von mehr benötigten Iterationen im Vergleich zum `poisson`-Problem (im fehlerfreien Fall 18 statt 7), insgesamt auch mehr Fehler generiert, so dass im Verhältnis seltener fälschlicherweise Fehler detektiert werden.

Trotz dieser hohen Zahl an fehlerhaften Korrekturen konvergiert das modifizierte Verfahren bereits nach 19 Iterationen. Im Vergleich dazu benötigt der klassische Ansatz mit 33 Iterationen wesentlich mehr. Bei der Betrachtung der Konvergenzverläufe in Abbildung 4.12 ist zu erkennen, dass drei signifikante Störungen auftreten. Von diesen Störungen repariert der modifizierte Ansatz ausschließlich die Störung in Iteration 3 nicht ausreichend, woraus ein Sprung in der Residuumsnorm resultiert. Es ist jedoch zu erkennen, dass in den folgenden Iteration eine verbesserte Konvergenz vorliegt, so dass sich der Konvergenzverlauf wieder dem des fehlerfreien Falles annähert. Die Störung in der dritten Iteration geschieht in der Prolongationsphase auf Level 6 nach dem ersten Glättungsschritt (vgl. Tabelle 4.7). Dieser sollte zu einer Verbreitung der Störung auf mehr als den neun detektierten Komponenten führen. Demzufolge ist die Korrektur von 31 Komponenten nicht ausreichend, so dass, wie in Tabelle 4.8 zu erkennen, auch auf dem Feingitter noch eine entsprechend starke Störung registriert wird.

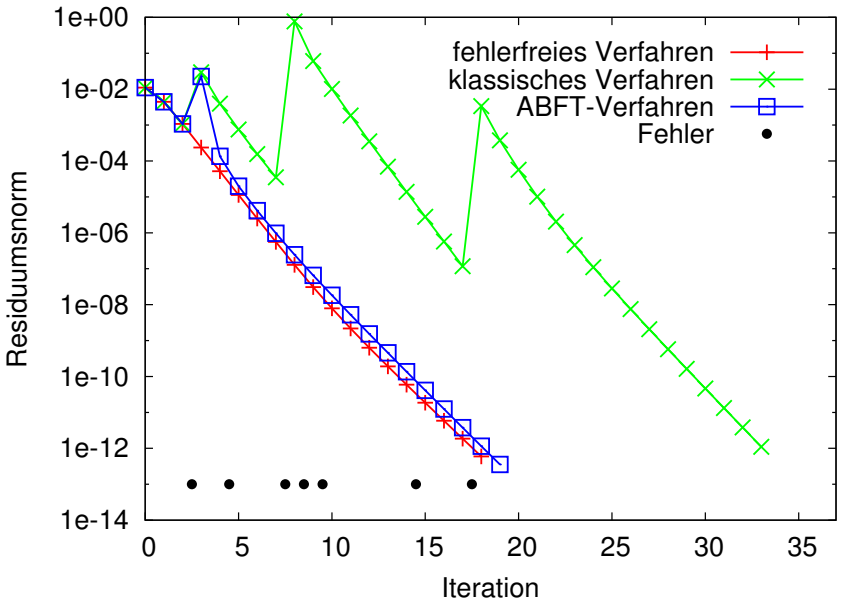

Abbildung 4.12: Konvergenzverlauf der unterschiedlichen fehlerbehafteten Mehrgitterverfahren sowie des fehlerfreien Verfahrens in Test 127 des `andi`-Problems mit Markierung der Iterationen in denen Fehler generiert werden.

Die beiden weiteren einflussreichen Störungen in Iteration 8 und 18 werden dagegen vom fehlertoleranten Ansatz unmittelbar behoben. Bei den fehlerhaften Detektionen in Iteration 10 und 11 auf dem zweitgröbsten Gitter handelt es sich um die Folgen einer nicht ausreichend großen Korrektur in Iteration 9 auf dem selben Level. Hier tritt nach der ersten Anwendung des Glätters erneut ein Fehler auf, jedoch werden nur 17 Komponenten korrigiert. Demzufolge schlägt die Detektion in den folgenden beiden Prolongationsphasen abermals an, da der Quotient *tmp* nicht ausreichend groß für eine Neuskalierung ist. Ähnliches passiert nach der Störung in der fünfzehnten Iteration.

Diese zusätzlichen fehlerhaften Detektionen (ausgenommen Iteration 3) führen jedoch zu einem nicht nennenswerten Mehraufwand, da sie auf einem entsprechend groben Gitter passieren und die Korrektur weiterhin in reduzierter Form ausschließlich für die entsprechenden Komponenten erfolgen kann. Auch wird durch diese Korrekturen das Konvergenzverhalten nicht erkennbar beeinflusst.

Iter	Phase	Level	Step	Vorher	Nachher	%	#det	#korr
3	Restr.	5	3	$4.459638 \cdot 10^{-1}$	$4.45964 \cdot 10^{-11}$	$2.14 \cdot 10^{-4}$		
			4	$8.351431 \cdot 10^{-1}$		$2.72 \cdot 10^{-11}$		
	Prol.	6	1	$6.106451 \cdot 10^{-1}$	$9.317704 \cdot 10^{-6}$	$9.99 \cdot 10^{1}$	9	31
5	Prol.	5	3	$6.554866 \cdot 10^{-1}$		$5.55 \cdot 10^{-10}$		
8	Restr.	7	3	$4.013219 \cdot 10^{-1}$		$1.81 \cdot 10^{-9}$		
	Prol.	7	3	$9.623099 \cdot 10^{-1}$	$6.014437 \cdot 10^{-2}$	$9.38 \cdot 10^{01}$	25	49
9	Restr.	4	2	$9.039921 \cdot 10^{-1}$	$9.039311 \cdot 10^{-1}$	$6.75 \cdot 10^{-3}$	49	81
	Prol.	1	2	$8.535561 \cdot 10^{-1}$	$8.535599 \cdot 10^{-1}$	$4.47 \cdot 10^{-4}$	17	45
10	Prol.	5	2	0	$6.953356 \cdot 10^{-310}$	0		
15	Prol.	1	1	$3.826846 \cdot 10^{-1}$	$3.826842 \cdot 10^{-1}$	$1.25 \cdot 10^{-4}$	28	54
18	Restr.	7	4	$1.325293 \cdot 10^{-2}$	$1.144546 \cdot 10^{-79}$	10^{2}	9	25

Tabelle 4.7: Übersicht über generierte Fehler in Test 127 des `andi`-Problems mit Angaben des Zeitpunktes und der prozentualen Auswirkung der Manipulation sowie der Anzahl der dadurch detektierten bzw. korrigierten Komponenten bei Verwendung des ABFT-Verfahrens.

Iter	Phase	Level	#det	#korr	*tmp*
3	Prol.	7	383	519	$1.69 \cdot 10^{3}$
10	Prol.	1	9	33	$6.29 \cdot 10^{-1}$
11	Prol.	1	1	9	$9.96 \cdot 10^{-1}$
16	Prol.	1	8	31	$5.33 \cdot 10^{-1}$
17	Prol.	1	5	21	$9.60 \cdot 10^{-1}$

Tabelle 4.8: Übersicht über fehlerhafte Detektionen bei Verwendung des ABFT-Verfahrens in Test 127 des `andi`-Problems mit Angabe der daraus resultierenden Anzahl an Detektionen bzw. Korrekturen sowie des für die Neuskalierung verantwortlichen *tmp*-Wertes.

`dico`-Problem Auch beim dritten Testproblem treten einige Fälle auf, in denen mehrfach fälschlicherweise Reparaturen durchgeführt werden. Exemplarisch betrachten wir hier den Test 244. Abbildung 4.13 sowie die Tabellen 4.9 und 4.10 beinhalten die zugehörigen Informationen.

In diesem Test werden nur drei Fehler erzeugt, jedoch schlägt der Detektionsmechanismus viermal an. Des Weiteren ist nur eine der Detektionen die Folge eines generierten Fehlers und die drei Übrigen sind fehlerhaft. Grund für diese fehlerhaften Detektionen ist die innerhalb der zweiten Iteration auf Gitterlevel 1 generierte Störung. Obwohl die Veränderung der Komponente groß ist, wird sie vom Algorithmus nicht erkannt. Erst auf dem Feingitter wird registriert, dass sich das Verfahren nicht wie erwartet verhält. Da zwischen der Fehlergenerierung und -erkennung entsprechend viele Gitterlevel und Anwendungen von fehlerausbreitenden Glättungsschritten liegen, wird hier eine entsprechend große Anzahl an Komponenten als fehlerhaft angenommen. Durch die große Anzahl der gestörten Komponenten und das ausschließliche Skalieren der Vergleichswerte innerhalb der Residuumstests führt die Störung in der folgenden Prolongationsphase erneut zu

zwei fehlerhaften Detektionen, die aber zu einer vergleichsweise geringfügigen Korrektur führen. Insbesondere erfolgt dort die zweite Detektion lediglich, weil der Quotient *tmp* mit 0.98 minimal zu klein für eine Nachskalierung der folgenden Vergleichswerte ist.

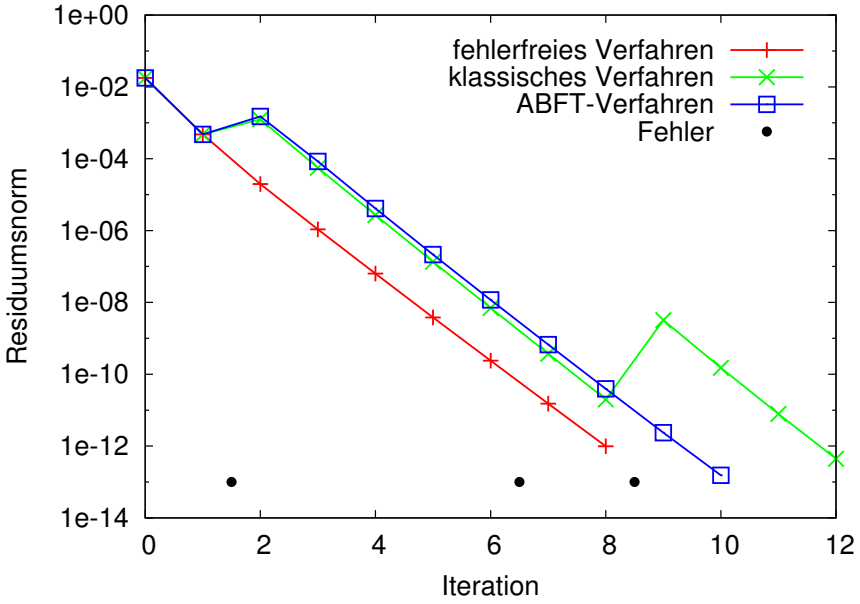

Abbildung 4.13: Konvergenzverlauf der unterschiedlichen fehlerbehafteten Mehrgitterverfahren sowie des fehlerfreien Verfahrens in Test 244 des `dico`-Problems mit Markierung der Iterationen in denen Fehler generiert werden.

Die zweite einflussreiche Störung in Iteration 9 wird vom modifizierten Verfahren wieder tadellos erkannt und führt somit zu keinem weiteren Anstieg der bis zur Konvergenz benötigten Iterationszahlen, so dass wir nichtsdestotrotz zwei Iterationen weniger benötigen als beim klassischen Verfahren.

Iter	Phase	Level	Step	Vorher	Nachher	%	#det	#korr
2	Prol.	1	3	$3.825627 \cdot 10^{-1}$	$5.837444 \cdot 10^{-6}$	$9.99 \cdot 10^{1}$		
7	Prol.	1	1	$3.535546 \cdot 10^{-1}$		$6.43 \cdot 10^{-11}$		
9	Prol.	2	4	$9.061305 \cdot 10^{-1}$	$9.061307 \cdot 10^{-1}$	$2.63 \cdot 10^{-5}$	1	9

Tabelle 4.9: Übersicht über generierte Fehler in Test 244 des `dico`-Problems mit Angaben des Zeitpunktes und der prozentualen Auswirkung der Manipulation sowie der Anzahl der dadurch detektierten bzw. korrigierten Komponenten bei Verwendung des ABFT-Verfahrens.

Iter	Phase	Level	#det	#korr	*tmp*
2	Prol.	7	26699	27561	$1.52 \cdot 10^{1}$
3	Prol.	2	1	127	$9.80 \cdot 10^{-1}$
		3	31	61	1.37

Tabelle 4.10: Übersicht über fehlerhafte Detektionen bei Verwendung des ABFT-Verfahrens in Test 244 des `dico`-Problems mit Angabe der daraus resultierenden Anzahl an Detektionen bzw. Korrekturen sowie des für die Neuskalierung verantwortlichen *tmp*-Wertes.

`andicore`-Problem Abschließend betrachten wir noch einen Test der vierten Testreihe zum anisotropen Diffusions-Konvektions-Reaktions-Problem. In dieser sticht kein Test durch eine bemerkenswert hohe Anzahl an fehlerhaften Detektionen oder durch Mehriterationen beim ABFT-Verfahren heraus. Dennoch werden auch hier Fehler induziert, die beim klassischen Verfahren zur Divergenz führen, jedoch beim modifizierten Ansatz ohne zusätzliche Iterationen konvergieren. Ein Beispiel dafür stellt Test 376 dar. Bei der Generierung von sieben Fehlern genügen zwei Reparaturen aus, um den optimalen Kovergenzverlauf wiederherzustellen. Außerdem treten keine fehlerhaften Detektionen auf.

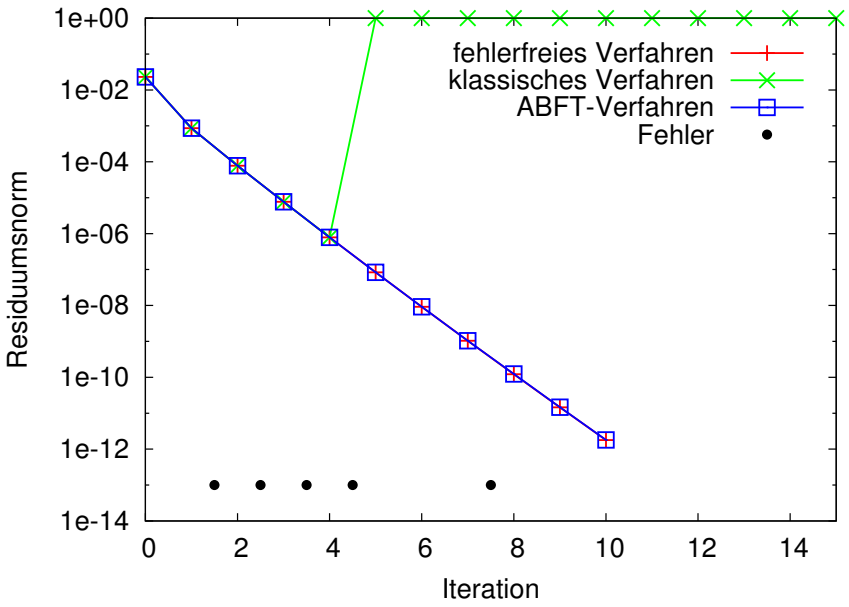

Abbildung 4.14: Konvergenzverlauf der unterschiedlichen fehlerbehafteten Mehrgitterverfahren sowie des fehlerfreien Verfahrens in Test 376 des `andicore`-Problems mit Markierung der Iterationen in denen Fehler generiert werden.

Wie bereits angesprochen divergiert das klassische Verfahren in dieser Situation. Grund dafür ist die in Iteration 5 erzeugte Störung (vgl. Tabelle 4.11 und Abbildung 4.14). Hier wird, wie in Test 013 des `poisson`-Problems, das führende Bit einer Komponente

manipuliert, so dass sich die Größenordnung des Eintrages stark verändert. Dies führt ohne Korrektur dazu, dass im Folgenden die Werte von Variablen auf `NaN` umspringen. Die Korrektur von neun Komponenten reicht jedoch aus, um das Verfahren mit der gleichen Anzahl an benötigten Iteration konvergieren zu lassen. Durch die unmittelbare Detektion des Fehlers und der Tatsache, dass die anderen Störungen nicht signifikant sind, treten außerdem keine fehlerhaften und somit überflüssigen Reparaturen auf.

Iter	Phase	Level	Step	Vorher	Nachher	%	#det	#korr
2	Restr.	3	4	$3.399437 \cdot 10^{-2}$		$3.27 \cdot 10^{-13}$		
	Prol.	3	3	$9.063188 \cdot 10^{-2}$		$5.14 \cdot 10^{-7}$		
3	Restr.	7	3	$3.675134 \cdot 10^{-1}$	$3.675096 \cdot 10^{-1}$	$1.04 \cdot 10^{-3}$	1	9
	Prol.	2	1	$3.927551 \cdot 10^{-1}$		$3.71 \cdot 10^{-9}$		
4	Prol.	1	3	0	$7.905050 \cdot 10^{-323}$	0		
5	Prol.	4	3	$4.193411 \cdot 10^{-1}$	$7.538465 \cdot 10^{307}$	`inf`	9	25
8	Prol.	5	2	$9.683673 \cdot 10^{-1}$		$4.81 \cdot 10^{-8}$		

Tabelle 4.11: Übersicht über generierte Fehler in Test 376 des `andicore`-Problems mit Angaben des Zeitpunktes und der prozentualen Auswirkung der Manipulation sowie der Anzahl der dadurch detektierten bzw. korrigierten Komponenten bei Verwendung des ABFT-Verfahrens.

4.4 Fazit: ABFT beim Mehrgitterverfahren

In diesem Kapitel haben wir einen Mechanismus vorgestellt, der es ermöglicht anhand von im Mehrgitterverfahren verwendeten Größen, Fehler innerhalb der Anwendung des Glätters zu erkennen. Außerdem ist die Detektion mit einer Lokalisierung verbunden, so dass im Anschluss daran eine Reparatur der fehlerhaften Komponenten erfolgen kann, ohne dass dafür zusätzliche Größen abgespeichert oder sogar Checkpoints erstellt werden müssen. Dies ist durch die Erzeugung einer zusätzlichen skalaren Vergleichsgröße möglich. Die Detektion selbst benötigt pro Gitterlevel an zwei Stellen den komponentenweisen Vergleich von einem Vektor mit dieser generierten Größe. Reparaturen können darüberhinaus in lokal reduzierten Systemen erfolgen, so dass der Aufwand vergleichsweise gering gehalten werden kann.

Eine erste numerische Untersuchung zeigt, dass ein Großteil der Fehler, die die Konvergenz entscheidend beeinflussen, entdeckt und repariert werden können. Weniger signifikante Fehler werden dagegen ignoriert, was einen entscheidenden Vorteil gegenüber einer einfachen Detektion mittels Prüfsummen, wie in Kapitel 3 beschrieben, darstellt. Hier ist es schwer innerhalb des iterativen Prozesses zu ermitteln, wie exakt eine Prüfsumme sein muss, um unerhebliche Fehler zu tolerieren.

Die Tests haben gezeigt, dass insbesondere Störungen die in frühen Iterationen auftreten unter Umständen nicht detektiert werden. Dies führt in einigen Fällen dazu, dass der Algorithmus erst auf einem der folgenden Gitterlevel eine Anomalie im Konvergenzverlauf

feststellt und die daraus resultierende Reparatur keine Verbesserung der Konvergenz bewirkt. In diesen Situationen erhalten wir somit sogar eine minimale Verschlechterung der Konvergenz, da dadurch in lokalen Bereichen die Glättung ausbleibt. Durch eine bessere Intialisierung der Vergleichswerte kann dies möglicherweise behoben werden.

Mit weiteren Analysen ist es darüber hinaus potentiell möglich die erzeugten Vergleichswerte allgemein zu optimieren, um eine bessere Detektion zu erhalten. Dies würde auch das Auftreten von fehlerhaften Detektionen vermindern, da diese in den meisten Fällen aus dem Nicht-Detektieren einer signifikanten Störung resultieren. Zusätzlich wäre es möglich mehrere Vergleichswerte für unterschiedliche Teilgebiete zu erzeugen, um das lokale Verhalten der Lösung besser abzubilden. Diese Optimierungen könnten insbesondere problemangepasst erfolgen.

Eine verspätete Detektion ist i.d.R. damit verbunden, dass der Quotient tmp, welcher einer Veränderung der Vergleichswerte entspricht, auch nach der Reparatur größer als Eins ist. In diesen Fällen wäre es möglich den Reparaturmechanismus adaptiv zu erweitern. Zum einen könnte die Anzahl der zu reparierenden Komponenten vergrößert werden. Damit würden Situation abgedeckt in denen nicht der gesamte Bereich detektiert wird, in dem sich die Störung ausgebreitet hat. Andererseits ließe sich auch die Korrektur so erweitern, dass in diesen Fällen mit der Näherungslösung von einem feineren, respektive gröberen Level (Restriktion bzw. Prolongation) repariert wird. Spätestens eine Reparatur vom Fein- bzw. Grobgitter sollte den Quotienten auf einen Wert kleiner gleich Eins setzen, da dies in etwa einer Nicht-Ausführung der entsprechenden Phase entspricht und somit keiner Verbesserung der Lösung in einem Teilbereich. Alternativ könnte, falls die Größe der Indexmenge $\mathcal{I}$ der zu korrigierenden Komponenten einen Wert überschreitet, auch eine Wiederholung des Zyklus ausgehend von der letzten Approximierenden des Feingitters durchgeführt werden. Diese und andere Verbesserungen sind mögliche Schwerpunkte weiterer Arbeiten.

5 Ein fehlertolerantes Mehrgitterverfahren

Im Folgenden wollen wir ein fehlertolerantes Mehrgitterverfahren konstruieren, das insbesondere im fehlerfreien Fall nicht zu viel zusätzlichen Aufwand erzeugt.

Bisher haben wir zwei Möglichkeiten vorgestellt, mit denen sich zumindest teilweise Bereiche des Algorithmus absichern lassen. Prüfsummen erlauben es dabei das Verfahren vollständig gegen Fehler abzusichern. Jedoch erzeugen sie auch im fehlerfreien Fall einen Mehraufwand von ungefähr 30% ($M = 9$, `MVV`-Operation dominant, vgl. Kapitel 3), was bei der zu erwartenden Fehlerhäufigkeit zu viel ist, da nur in seltenen Fällen wirklich signifikante Störungen auftreten, die das Verfahren zur Divergenz führen oder aber zumindest die Konvergenzgeschwindigkeit erheblich beeinflussen. Des Weiteren haben wir bei dem Mehrgitterverfahren eine algorithmenbasierte Fehlertoleranz konstruiert, die es uns ermöglicht die vom Glättungsoperator generierten Daten auf Fehler zu überprüfen und gegebenenfalls zu reparieren. Hierbei entstehen jedoch u. U. zusätzliche Iterationen und ein weiteres Problem ist die nicht erfolgende Absicherung der Operationen außerhalb der Glättung.

Wir wollen nun durch ein Zusammenspiel der beiden Komponenten ein vollständig fehlertolerantes Mehrgitterverfahren konstruieren.

5.1 Konstruktion

In Abschnitt 2.2.3 haben wir gezeigt, dass bereits bei 8 Glättungsschritten eines einfachen Glättungsoperators der Hauptaufwand des Mehrgitterverfahrens bei der Glättung liegt und die übrigen Operationen nur 20% des Aufwandes entsprechen. Zudem ist diese Verteilung unabhängig von der Anzahl der Unbekannten, da in beiden Phasen der Zusammenhang zwischen Aufwand und Anzahl der Freiheitsgrade linear ist. Die Verwendung von komplexeren Glättern, wie zum Beispiel Krylowraum-Verfahren [18], oder mehr Glättungsschritten vergrößert das Ungleichgewicht weiter, so dass der Aufwand der Transferphase im Vergleich immer geringer wird. Daher kombinieren wir den in Kapitel 4 vorgestellten Ansatz der algorithmenbasierten Fehlertoleranz für den Glätter des Mehrgitterverfahrens nun mit dem der Prüfsummen aus Kapitel 3. Alle Operationen die außerhalb des Glätters nötig sind, werden hierbei mit Prüfsummen abgesichert, wobei wir die einfachste Variante von korrigierenden Prüfsummen verwenden. Operationen,

die durch Prüfsummen abgesichert sind, werden solange wiederholt bis die Prüfsummen übereinstimmen. Den resultierenden Algorithmus bezeichnen wir als fehlertolerantes Mehrgitterverfahren (**FTMG**).

Die verwendete Variante von Prüfsummen erlaubt es darüber hinaus prinzipiell, wenn nach einer gewissen Anzahl von Versuchen keine Gleichheit erzielt wurde, so auf Fehler in den Operatoren zurückzuschließen und eine Neuassemblierung durchzuführen. Dabei muss weiterhin davon ausgegangen werden, dass zumindest die Bestimmung der Spaltensummen der Systemmatrizen innerhalb der Assemblierungsphase fehlerfrei erfolgt. Dies ist zum Beispiel durch mehrmalige Berechnungen möglich.

Wir untersuchen nun eingehend die Auswirkung der einzelnen Bestandteile unseres Algorithmus und den erwarteten Mehraufwand, der durch die einzelnen Komponenten entsteht. Außerdem werden wir den konstruierten Ansatz mit einer Variante mit ausschließlich korrigierenden Prüfsummen vergleichen und auf diese Weise die Vorteile des Ansatzes demonstrieren. Dazu werden wir im späteren Verlauf insbesondere die Laufzeit der unterschiedlichen Verfahren vergleichen.

5.1.1 Erwartete Auswirkungen auf die Laufzeit

Die einzelnen Komponenten des modifizierten Ansatzes haben unterschiedlich starken Einfluss auf die Laufzeit des Verfahrens. Wir betrachten nun exemplarisch, welche Laufzeiterhöhungen, bei der Verwendung von konformen lineare Finite Elemente (Q_1), zu erwarten sind.

Innerhalb der Transferphase finden auf jedem Gitter die in Tabelle 5.1 dargestellten Operationen statt. Dabei handelt es sich i.d.R. um die in Kapitel 3 bzgl. Prüfsummen analysierten `MV`- und `MVV`-Operationen sowie eine einfache Vektor-Vektor-Addition (`v+v`). Die Zeilenangaben beziehen sich auf Algorithmus 4.1 und der Aufwand ergibt sich aus den Betrachtungen aus Kapitel 3. Zusätzlich zu der Art der Operation geben wir noch die Besetzungsdichte M der verwendeten Matrix sowie die Anzahl der Freiheitsgrade an für die die Operation durchgeführt wird, d.h. die Dimension des eingehenden Vektors. Hieraus ergibt sich unmittelbar der erwartete Aufwand im Vergleich zur ungesicherten Operation.

Die innerhalb der Transferphase durchgeführte Restriktion der Feingitteriterierten (Algorithmus 4.1, Zeile 8) kann dabei durch eine Matrix-Vektor-Multiplikation mit einem Element pro Zeile ($M = 1$) dargestellt werden. Diese Matrix der Dimension $\frac{N}{4} \times N$ (zweidimensionales Gitter) besitzt, aufgrund der verwendeten Injektion zur Restriktion, in jeder Zeile nur eine Komponente ungleich Null. Dies führt dazu, dass der Operator lediglich auf ein Viertel der Komponenten des N-dimensionalen Vektors zugreift, wodurch sich die effektive Anzahl der bei der Operation verwendeten Komponenten auf $\frac{N}{4}$ reduziert. Zusätzlich werden die Operationen in Zeile 9 bzw. 11 bereits auf einem gröberen Gitter durchgeführt, was ebenfalls dazu führt, dass die Operationen nur mit einem $\frac{N}{4}$-dimensionalen Vektor durchgeführt werden.

Zeile	5:	`MVV`	mit	$M = 9, N$	und Aufwand	130%
Zeile	7:	`MV`	mit	$M = 4, N$	und Aufwand	150%
Zeile	8:	`MV`	mit	$M = 1, \frac{N}{4}$	und Aufwand	300%
Zeile	9:	`MVV`	mit	$M = 9, \frac{N}{4}$	und Aufwand	130%
Zeile	11:	`v+v`	mit	$\frac{N}{4}$	und Aufwand	300%
Zeile	13:	`MVV`	mit	$M = 4, N$	und Aufwand	160%

Tabelle 5.1: Operationen innerhalb der Transferphase des Mehrgitteralgorithmus 4.1 mit Angabe der Besetzungsdichte M und der Unbekannten sowie des Aufwands, der bei der Verwendung von Prüfsummen entsteht.

Die Anzahl der durchgeführten numerischen Operationen ergibt sich je nach Art der Matrix-Operation (vgl. Kapitel 3.2). Eine Matrix-Vektor-Multiplikation (`MV`) benötigt $2MN$ und eine `MVV`-Operation $(2M+1)N$ Operationen. Dahingegen erfordert die Vektor-Vektor-Addition nur die komponentenweise Addition und somit N Operationen. Die Summe der Produkte aus der Anzahl an Operationen und dem verbundenen Aufwand durch die Prüfsummen ergibt, im Verhältnis zur ungesicherten Variante, schließlich einen Mehraufwand von durchschnittlich

$$\frac{(19 \cdot 1.3 + 8 \cdot 1.5 + \frac{1}{4} \cdot 2 \cdot 3.0 + \frac{1}{4} \cdot 19 \cdot 1.3 + \frac{1}{4} \cdot 1 \cdot 3.0 + 9 \cdot 1.6)N}{(19 + 8 + 0.5 + \frac{19}{4} + \frac{1}{4} + 9)N} = 1.4343$$

pro Operation innerhalb der Transferphase. Aufgrund der Tatsache, dass die Transferphase bei der Verwendung von 8 Glättungsschritten einen Anteil von 20% am Gesamtaufwand darstellt (vgl. Kapitel 2.2.3), erhöht sich der Aufwand des vollständigen Verfahrens, im exemplarisch betrachteten Fall, nur um den Faktor

$$0.8 \cdot 1 + 0.2 \cdot 1.4343 = 1.0869.$$

Wir können somit erwarten, dass diese Modifikation der Transferphase einen Mehraufwand von weniger als 9% erzeugt. Innerhalb der Implementierung erwarten wir darüber hinaus eine geringere Aufwandserhöhung, da wir durch die Verwendung des Einsvektors als Prüfvektor dort zusätzlich Operationen einsparen (ausschließlich Summation statt Multiplikation und Summation, vgl. Kapitel 3.4). Dahingegen würde die Verwendung von Prüfsummen für das gesamte Verfahren mindestens 30% Mehraufwand produzieren. Die Durchführung von zusätzlichen Glättungsschritten oder aber komplexeren Glättungsoperatoren senkt den Einfluss der Transferphase auf die Laufzeit weiter.

Der Einfluss der ABFT-Modifikation des Glätters auf die Laufzeit ist ebenfalls gering. Diese Modifikation erfordert, ohne den Reparatur-Mechanismus, ausschließlich an zwei Stellen pro Gitterlevel und Iteration zusätzlich den Vergleich von Vektorkomponenten mit einem Vergleichswert sowie die Erzeugung dieses Wertes. Dieser Aufwand sollte in etwa vergleichbar sein mit wenigen Skalarprodukten. Insbesondere der fehlerfreie Fall ist somit nur mit einem geringen Mehraufwand verbunden. Sollte eine Reparatur notwendig sein, so müssen weitere Operationen durchgeführt werden. Diese können aber, wie in Kapitel 4.1 beschrieben, in reduzierten Systemen erfolgen.

Der Nachteil des, aus der Kombination der beiden Mechanismen entstehenden, FTMG-Ansatzes ist, dass u. U. mehr Iterationen durchgeführt werden müssen, wie wir bereits bei der Analyse in Kapitel 4 gesehen haben. Bei der Verwendung von korrigierenden Prüfsummen für das komplette Verfahren ist dagegen eine Konvergenz nach der selben Anzahl an Iterationen, wie im fehlerfreien Fall, zu erwarten. Die Auswirkungen dieser Mehriterationen werden wir im Weiteren in numerischen Experimenten analysieren.

Unter der optimistischen Annahme, dass sich der entstehende Mehraufwand bei der Verwendung von korrigierenden Prüfsummen für das gesamte Mehrgitterverfahren in der Größenordnung der dominanten `MVV`-Operationen (30% bei Q_1-Finite Elemente, vgl. Kapitel 3) befindet, beläuft sich die Laufzeitersparnis durch den FTMG-Ansatz im fehlerfreien Fall auf ca. 20%. Dies ergibt sich beim Vergleich zum erwarteten Mehraufwand des FTMG-Verfahrens, der sich auf unter 10% beläuft.

Dieser Laufzeitgewinn gegenüber einem Ansatz mit korrigierenden Prüfsummen für das gesamte Mehrgitterverfahren kann somit in Mehriterationen im fehlerbehafteten Fall investiert werden, ohne dass eine höhere Gesamtlaufzeit zu erwarten ist.

5.2 Numerische Experimente

Wir werden nun eine Reihe an numerischen Tests zum konstruierten vollständig fehlertoleranten Mehrgitterverfahren (FTMG-Verfahren) durchführen und auch den erwarteten Einfluss der einzelnen Bestandteile auf die Laufzeit, wie im vorherigen Abschnitt untersucht, numerisch bestätigen. Wie bereits angesprochen handelt es sich bei dem Verfahren um den in Algorithmus 4.1 dargestellten Ansatz, wobei zusätzlich die Operationen außerhalb der Glättungsphase durch Prüfsummen abgesichert werden. Wünschenswert ist, dass die vorgestellte Variante im Großteil der Fälle, in Bezug auf die Laufzeit, besser oder vergleichbar abschneidet, wie der vollständig mit korrigierenden Prüfsummen abgesicherte Algorithmus. Im Fall, dass keine signifikanten Störungen auftreten und das Verfahren keine zusätzlichen Iterationen benötigt, ist der FTMG-Ansatz einem Verfahren mit vollständig korrigierenden Prüfsummen bzgl. der Laufzeit überlegen, da der Aufwand durch die Tests vergleichsweise gering ist.

Die Laufzeit des Verfahrens mit ausschließlich korrigierenden Prüfsummen werden wir dabei nicht durch Experimente ermitteln, sondern gehen weiterhin davon aus, dass dies mindestens zu einer Steigerung der Laufzeit um den Faktor 1.3 führt und die selbe Anzahl an Iterationen benötigt, wie der fehlerfreie Fall. Dies ist insofern weiterhin eine optimistische Annahme, da der Faktor 1.3 bei der `MVV`-Operation erst mit einer hohen Anzahl an Anwendungen erreicht wird (vgl. Abschnitt 3.2). Außerdem vernachlässigen wir in dem Fall, dass die Korrektur es erfordert fehlerhafte Operationen zu wiederholen, um das richtige Resultat zu erhalten.

5.2.1 Fehlergenerierung

Der Schwerpunkt in diesem Kapitel liegt auf dem Laufzeitvergleich des Ansatzes im Vergleich zu herkömmlichen Verfahren. Dabei wird sowohl mit einem vollständig durch Prüfsummen gesicherten Verfahren verglichen, als auch mit dem Standard-Verfahren. Da die Funktionsweise des ABFT-Mechanismus bereits in Kapitel 4 genauer betrachtet wurde, verwenden wir für die Anwendung des Glätters das selbe Fehlermodell wie in Abschnitt 4.3.1 beschrieben. Erneut werden die Fehler, mit einer Wahrscheinlichkeit von 1% pro Glättungsschritt, anhand eines Seed-Wertes generiert und können so in allen Fällen reproduziert werden, um einen Vergleich zu ermöglichen.

Darüber hinaus lassen wir die Operationen in der Transferphase, welche mit Prüfsummen gesichert werden, mit einer 1%-igen Wahrscheinlichkeit wiederholen. Dies soll den Fall von fehlerhaften Berechnungen innerhalb dieser Operationen abdecken. Durch diese allgemeine „Fehlergenerierung“ im Bereich der Prüfsummen simulieren wir sowohl den Fall, dass die Operation fehlerhaft war, als auch den Fall, dass die Berechnung der Prüfsummen nicht fehlerfrei durchgeführt wurde. Die Auswahl der Operationen, die wiederholt werden, ist vollständig zufällig.

Damit die gemessene Laufzeit nicht zu stark von anderen Prozessen oder aber durch eine extrem hohe Anzahl an wiederholten Operationen auf den feinen Gittern beeinflusst wird, führen wir zu jedem Seed-Wert (ein Seed-Wert beschreibt weiterhin ein Fehlerszenario in der Glättungsphase, vgl. Kapitel 4.3.1) zehn Messungen durch. Die Streuung der gemessenen Zeiten liefert so ein möglichst repräsentatives Ergebnis des Einflusses der Wiederholungen der durch Prüfsummen abgesicherten Operationen auf die Laufzeit.

5.2.2 Testreihen

Wir analysieren die Laufzeit anhand von zwei Testproblemen, die wir bereits in Kapitel 4 verwendet haben. Dabei betrachten wir das `andi`-Problem

$$-\nabla \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0.15 \end{pmatrix} \nabla u = f, \qquad (\texttt{andi})$$

welches vergleichsweise langsam konvergiert und mehr Iterationen benötigt und das vollständige `andicore`-Problem

$$-\nabla \cdot \begin{pmatrix} 1.2 & -0.7 \\ -0.4 & 0.9 \end{pmatrix} \nabla u + \begin{pmatrix} 0.4 \\ -0.2 \end{pmatrix} \cdot \nabla u + 0.3u = f. \qquad (\texttt{andicore})$$

Erneut geben wir dabei die Lösung durch

$$u = \sin(\pi x) \sin(\pi y)$$

vor und setzen entsprechende Dirichlet-Randbedingungen auf dem Einheitsquadrat $[0, 1]^2$.

Die Parameter des Mehrgitteralgorithmus sind nahezu identisch mit denen aus Kapitel 4. Wir verwenden eine Gitterhierarchie aus 8 Gittern, Q_1-Finite Elemente, bilineare Interpolation für den Gittertransfer (ausgenommen ist die Restriktion der Lösungsapproximation, vgl. Abschnitt 2.2.2) sowie ein mit dem Faktor 0.7 gedämpftes Jacobivorkonditioniertes Richardson-Verfahren als Glätter. Wir führen ebenfalls erneut vier Vor- und Nachglättungen durch. Im Unterschied zu Kapitel 4.3 verwenden wir jedoch auf dem Grobgitter bereits eine Schrittweite von $h = 0.125$ und erhalten somit ungefähr eine Millionen Freiheitsgrade auf dem Feingitter. Diese Verfeinerung wird durchgeführt, um die Laufzeiten zu erhöhen und somit die Ergebnisse besser visualisieren zu können bzw. weniger anfällig gegenüber Schwankungen zu machen.

Das Abbruchkriterium der Mehrgitterverfahren liegt bei einer relativen Residuumsreduktion von 10^{-8} beim `andi`-Problem und 10^{-10} beim `andicore`-Problem. Die unterschiedlichen Abbruchkriterien werden dabei verwendet, da das `andi`-Problem in dieser Situation früher einen Bereich erreicht, in dem Rundungsfehler einen Einfluss haben (vgl. Kapitel 4.3, Test 028). Das Grobgitterproblem wird weiterhin erst bei einer relativen Residuumsreduktion von 10^{-10} abgebrochen und somit nahezu ausiteriert. Wir gehen weiterhin davon aus, dass das Grobgitterproblem ohne viel Mehraufwand fehlerfrei gelöst werden kann.

Wir untersuchen und vergleichen im Weiteren die folgenden Verfahren:

- **fehlerfreies klassisches Verfahren**
  ungestörtes Mehrgitterverfahren (Mittelwert aus zehn Ausführungen)
- **korrigierende Prüfsummen** (Schätzung)
  Verwendung von korrigierenden Prüfsummen für alle Operationen mit geschätzter Laufzeit basierend auf Resultaten aus Kapitel 3 (ca. 30% Mehraufwand zum fehlerfreien klassischen Verfahren)
- **fehlerfreies FTMG**
  vollständiger, in Abschnitt 5.1 beschriebener, fehlertoleranter Ansatz, ohne dass Fehler generiert werden (Mittelwert aus zehn Ausführungen)
- **fehlerbehaftetes FTMG**
  FTMG-Ansatz mit Generierung von Fehlern und zehn Ausführungen mit unterschiedlicher Wiederholung von durch Prüfsummen gesicherten Operationen
- **fehlerbehaftetes klassisches Verfahren**
  klassisches Verfahren unter Einfluss von Fehlern

Es ist zu erinnern, dass beim fehlerbehafteten klassischen Verfahren das Erzeugen von Störungen endet, sobald das zugehörige FTMG-Verfahren konvergiert ist. Um hier Schwankungen der Laufzeit zu eliminieren erfolgt die Berechnung der Laufzeit aus einer Skalierung der Laufzeit des fehlerfreien klassischen Verfahrens auf Basis der Iterationszahlen. Dieses liefert durch zehnmalige Ausführung eine stabilere Schätzung. Darüber hinaus werden in dem Fall keine Fehler außerhalb der Glättungsphase erzeugt, was somit einem

optimistischen Fall entspricht. Für den Fall, dass das Verfahren innerhalb von 40 Iterationen nicht konvergiert, ist davon auszugehen, dass eine Divergenz vorliegt, die z.B. durch das Auftreten von `NaN` oder `inf` im Iterationsprozess entstanden ist.

Die Initialisierung der Vergleichswerte beim FTMG-Verfahren erfolgt hier mit dem Wert 1000. Dies ist notwendig, da im Vergleich zu den Tests in Kapitel 4 die Anzahl der Freiheitsgrade gestiegen ist.

Zusätzlich zur Analyse im fehlerbehafteten Fall untersuchen wir auch die Auswirkung der einzelnen Komponenten auf die Laufzeit im fehlerfreien Fall. Dabei unterscheiden wir zwischen der ausschließlichen Verwendung von **Prüfsummen** in der Transferphase, der Verwendung des in Kapitel 4 beschriebenen **ABFT**-Detektionsmechanismus für die Glättungsphase des Mehrgitterverfahrens und des aus der Kombination der beiden Komponenten entstehenden **FTMG**-Ansatzes.

Alle Zeitmessungen erfolgen auf einem Kern eines Intel Core i7 X 980 @ 3.33 GHz. Dabei steht ausreichend Arbeitsspeicher zur Verfügung, um keine Daten auf nichtflüchtigen Speicher mit hoher Zugriffszeit übertragen zu müssen.

`andi`-Problem Die Betrachtung des Einflusses der Mechanismen auf die Laufzeit des fehlerfreien Falles beim `andi`-Problem liefert die in Tabelle 5.2 dargestellten Resultate. Dabei werden die Laufzeiten unter Verwendung der entsprechenden Modifikation in zehn Tests gemessen und mit dem klassischen Verfahren verglichen.

Es ist zu erkennen, dass der Einfluss der verwendeten Prüfsummen auf die Gesamtlaufzeit, wie erwartet, etwas geringer ist als die in Abschnitt 5.1.1 ermittelte Schätzung. Wir messen einen Anstieg von weniger als 5% statt der berechneten 9%. Dies ist, wie bereits angesprochen, u. a. auf die Wahl des Einsvektors als Prüfvektor zurückzuführen. Darüberhinaus bestätigt sich die Annahme, dass der ABFT-Detektionsmechanismus mit weniger als 3% nur einen geringen Anstieg der Laufzeit verursacht. Insgesamt erhöht sich die benötigte Zeit für den konstruierten fehlertoleranten Ansatz (FTMG) somit im fehlerfreien Fall nur um ca. 7%.

Die Ergebnisse zu den 500 durchgeführten Tests (50 unterschiedliche Fehlerszenarien im Glätter mit je 10 Ausführungen) mit Fehlergenerierung sind in Abbildung 5.1 dargestellt und werden in Tabelle A.5 um weitere Statistiken ergänzt. Wir nummerieren die Tests dabei fortlaufend ab 401, um Verwechslungen mit vorherigen Tests aus Kapitel 4 zu vermeiden. In dieser Darstellung entspricht jeder Punkt der x-Achse einer Testreihe zu einem festen Seed-Wert. Zu jedem Test werden die Laufzeiten der zehnmal wiederholten FTMG-Verfahren (fehlerbehaftetes FTMG, hellblaue Kreuze) sowie die Laufzeit, die das klassische Verfahren (fehlerbehaftetes klassisches Verfahren, schwarze Quadrate) zur Konvergenz benötigt, dargestellt. Darüber hinaus geben wir die Zeit an, die der ABFT-Mechanismus zur Detektion und Reparatur, der in dieser Situation auftretenden Fehler, benötigt (ABFT-Mechanismus, lila Balken). Wir verwenden dabei in der Implementierung nicht die Möglichkeit, diese Reparaturen in reduzierten Systemen durchzuführen, so

	Klassisch	Prüfsummen	ABFT	FTMG
	25.8008	27.0902	27.0203	27.7707
	26.0534	27.4799	26.9171	28.5103
	25.9984	27.5805	27.1677	27.7984
	26.1863	27.6174	26.7648	27.5325
	26.3278	27.3653	26.6476	27.8364
	26.5627	27.3733	26.9674	28.0592
	25.8402	27.5449	26.9738	27.9856
	26.0607	27.4814	26.4110	27.8669
	26.0631	27.5735	26.7563	28.0990
	26.6685	27.0215	26.2793	27.6304
Mittelwert	26.1562	27.4128	26.7905	27.9090
Faktor	1	1.0480	1.0243	1.0670

Tabelle 5.2: Laufzeiten der unterschiedlichen Verfahren im fehlerfreien Fall in Sekunden gemessen beim `andi`-Problem.

dass diese Zeit weiter reduziert werden kann. Die Zeit innerhalb des ABFT-Mechanismus ist dabei nur eine Komponenten die beim FTMG-Ansatz die Laufzeit erhöht. Der größte Einfluss auf die Laufzeit entsteht durch möglicherweise notwendige Mehriterationen. Es sei erinnert, dass das klassische Verfahren in diesem Fall in den meisten Situationen nur konvergieren kann, da die Erzeugung der Fehler nach der Konvergenz des FTMG-Ansatzes gestoppt wird. Es dient somit lediglich als optimistische Schätzung.

Zum Vergleich mit fehlerbehafteten Situationen stellen wir zusätzlich die Laufzeiten der Verfahren ohne Fehlergenerierung und die erwartete Laufzeit bei vollständiger Verwendung von korrigierenden Prüfsummen dar. Die rote Linie entspricht der Laufzeit des einfachen Mehrgitterverfahrens, ohne dass Störungen auftreten. Dies entspricht somit dem idealen Fall. In dieser Situation benötigt das Verfahren beim beschriebenen Problem 14 Iterationen bis zur Konvergenz. Für diesen Fall stellen wir außerdem die Laufzeit des FTMG-Ansatzes (fehlerfreies FTMG, dunkelblaue Linie) dar. Der Abstand der beiden Linien entspricht also dem Laufzeitanstieg durch unsere Modifikationen im fehlerfreien Fall. Schließlich wird durch die grüne Linie die geschätzte Laufzeit für das Verfahren mit korrigierenden Prüfsummen für alle Operationen angegeben.

Der erste Blick auf die Resultate zeigt sofort, dass das klassische Verfahren unter dem Einwirken von Fehlern viele zusätzliche Iterationen und somit Laufzeit bis zur Konvergenz benötigt. In der vorliegenden Situation konvergiert dieses Verfahren nur in 7 von 50 Fällen ohne erkennbaren Mehraufwand. Alle weiteren Szenarien produzieren immer mehr zusätzlichen Aufwand als die Verwendung von Prüfsummen, obwohl das Verfahren nur solange gestört wird bis das zugehörige FTMG-Verfahren konvergiert ist. Darüber hinaus ist schnell ersichtlich, dass es keine entscheidende Rolle spielt welche Prüfsummenoperationen wiederholt werden. In allen Fällen liegen die Laufzeiten der zehn Ausführungen der Tests zum FTMG-Verfahren nah beieinander.

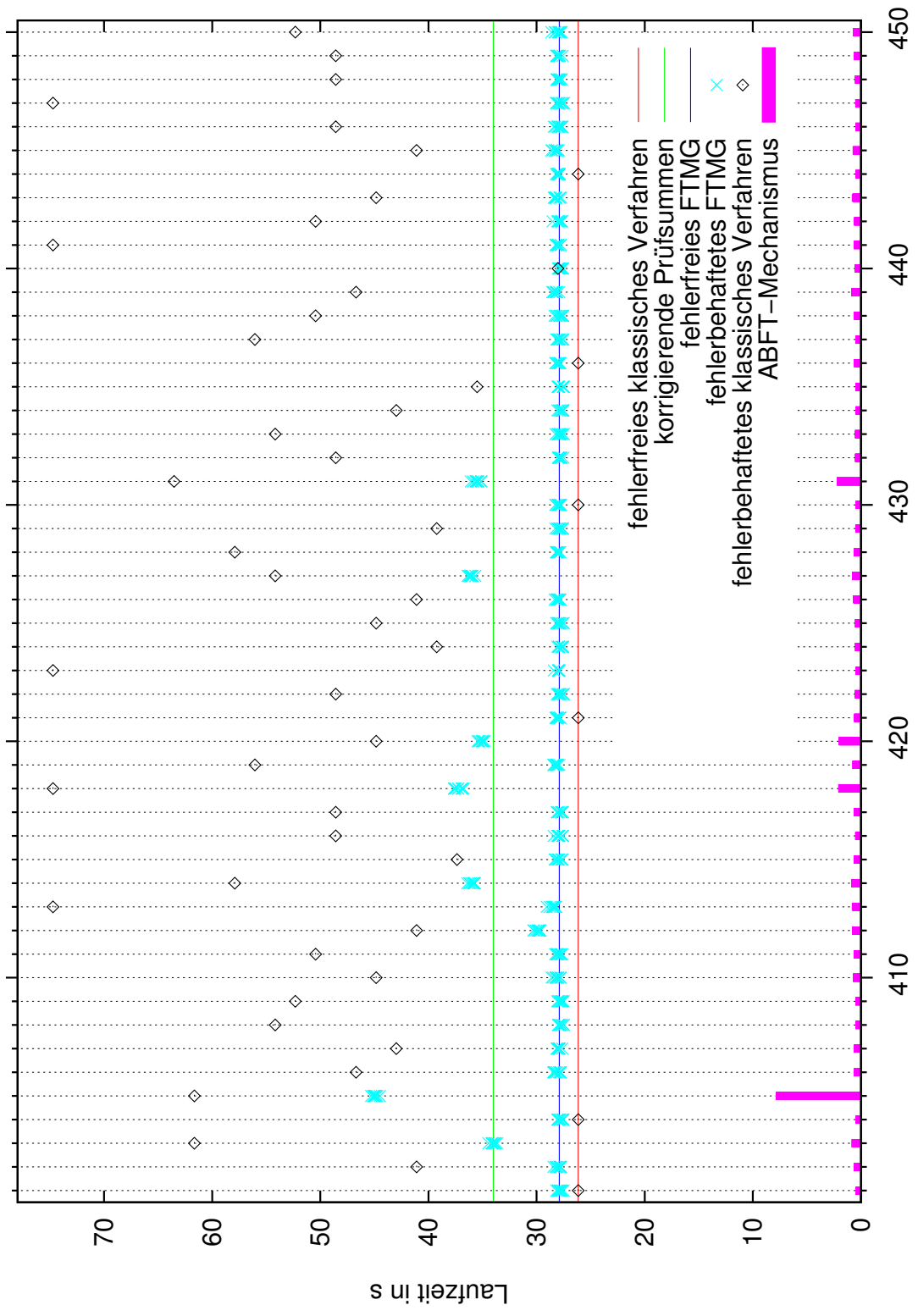

Abbildung 5.1: Laufzeiten der verschiedenen Tests zum `andi`-Problem mit zusätzlicher Angabe der Zeit, die der ABFT-Mechanismus zur Detektion bzw. Reparatur benötigt.

Weitere Auswertung der dargestellten Resultate zeigt, dass in 7 der 50 Fehlerszenarien der FTMG-Ansatz mehr Zeit als das modellierte Verfahren mit vollständig korrigierenden Prüfsummen benötigt (403, 405, 414, 418, 420, 427, 431), d.h. die zugehörigen Kreuze liegen oberhalb der grünen Linie. In diesem Zusammenhang sei erneut daran erinnert, dass unser Fehlermodell durchaus überdurchschnittlich viele Störungen erzeugt. Dies führt dazu, dass häufig, insbesondere auch mehrmals innerhalb eines Lösungsverfahrens, signifikante Störungen auftreten, die starken Einfluss auf das Konvergenzverhalten haben. Zum Einen ermöglicht diese Situation die bessere Simulation und Analyse von „worst-case“ Szenarien, zum Anderen wird dadurch jedoch auch der Eindruck des FTMG-Ansatzes verschlechtert, da wir häufiger höhere Laufzeiten erhalten. Es ist in der Regel nicht davon auszugehen, dass tatsächlich die in dieser Testreihe generierten ca. 8 Störungen innerhalb von 14 Iterationen auftreten.

Vier der Tests (405, 418, 420, 431) sind des Weiteren mit einem erkennbar höheren Aufwand beim ABFT-Mechanismus verbunden. Alle diese Tests sind mit dem in Kapitel 4, insbesondere in Abschnitt 4.3.3, detaillierter beschriebenen Phänomen zu erklären. Es treten jeweils verspätete Detektionen von Störungen auf, die sich bereits verbreitet haben. Dadurch werden entsprechend viele Komponenten als fehlerhaft detektiert, welche aber mit dem aktuellen Ansatz nicht ausreichend repariert werden können. Die große Anzahl an Komponenten führt dazu, dass bereits die Ermittlung des Trägers, mit Hilfe der Matrixstruktur, sehr aufwendig ist. Dies resultiert in dem erkennbaren Mehraufwand und als Folgerung aus der nicht möglichen Reparatur, entstehen die Mehriterationen, welche zur erhöhten Gesamtlaufzeit führen.

In den weiteren Fällen liegt die Laufzeit des fehlerbehafteten FTMG-Verfahrens nah an der des fehlerfreien Falles (blaue Linie). Hier wird durch den ABFT-Mechanismus das Konvergenzverhalten stabilisiert und Störungen ausreichend repariert. Im Vergleich dazu benötigt das klassische Verfahren in diesen Situationen häufig weitere Iterationen, um das Abbruchkriterium zu erreichen. Der klassische Ansatz divergiert dabei sogar in fünf Fällen (413, 418, 423, 441, 447), wohingegen der FTMG-Ansatz nur in Test 418 zusätzliche Iterationen durchführen muss. Hier liegt wieder ein Fall vor, in dem der ABFT-Detektionsmechanismus die Störung zu spät erkennt und folglich keine ausreichende Reparatur möglich ist. Nichtsdestotrotz kann die Konvergenz sichergestellt werden.

In sieben Szenarien (401, 404, 421, 430, 435, 440, 444) zeigen die Störungen keine Auswirkung auf die Iterationszahl und Laufzeit des klassischen Verfahrens (Quadrate sind auf der roten Linie). Auch der FTMG-Ansatz konvergiert in diesen Fällen in der Zeit, die auch für den fehlerfreien Fall, mit aktivem ABFT-Detektionsmechanismus, benötigt wird. Die teilweise durchgeführten Reparaturen zeigen dabei keine sichtbare Auswirkung auf die Laufzeit, so dass nur die üblichen 7% Mehraufwand entstehen.

Mit dem `andi`-Problem haben wir bisher einen Test betrachtet, in dem mit 14 Iterationen vergleichsweise viele Iterationen bis zum Erreichen des Abbruchkriteriums vonnöten sind. Durch die relativ hohe Zahl an Iteration entsteht ein entsprechend größerer zeitlicher Puffer zwischen dem FTMG-Ansatz und der Variante mit vollständigen Prüfsummen. Dies erlaubt es verhältnismäßig viele Mehriterationen durchzuführen, bis die zeitliche

Ersparnis des FTMG-Verfahrens aufgebraucht ist. Darüber hinaus haben wir bereits in Kapitel 4 gesehen, dass die langsame Konvergenz dem vorgestellten ABFT-Mechanismus zugute kommt. Dementsprechend werden wir im Folgenden die Untersuchungen an einem Weiteren der Testprobleme durchführen.

`andicore`-Problem Auch die Testreihe zum `andicore`-Problem beginnen wir mit der Messung des Einflusses auf die Laufzeit des fehlerfreien Falles. Die entsprechenden Resultate dazu sind in Tabelle 5.3 dargestellt.

	Klassisch	**Prüfsummen**	**ABFT**	**FTMG**
	18.6962	19.3955	19.2656	20.0691
	18.6194	19.5001	18.9261	20.2042
	18.6699	19.8495	19.3390	19.8760
	18.6049	19.5554	18.9757	19.8959
	18.5117	19.4028	19.5785	20.0326
	19.2698	19.5048	19.3469	19.9144
	18.4869	19.4926	19.5763	20.3621
	19.1226	20.3427	19.4654	20.4712
	19.1969	19.6330	18.9398	19.9921
	18.5587	19.4720	19.6908	20.0820
Mittelwert	18.7737	19.6148	19.3104	20.1900
Faktor	1	1.0448	1.0286	1.0754

Tabelle 5.3: Laufzeiten der unterschiedlichen Verfahren im fehlerfreien Fall in Sekunden gemessen beim `andicore`-Problem.

Wie bei der Testreihe zum `andi`-Problem messen wir auch hier, wie erwartet, ähnliche Auswirkungen durch die einzelnen Modifikationen auf die Gesamtlaufzeit des Verfahrens. Die Prüfsummen resultieren in einem Anstieg der Laufzeit um weniger als 5% und die ABFT-Detektionsmechanismen geben sich mit unter 3% Mehraufwand zufrieden. Insgesamt steigt somit die Laufzeit des fehlertoleranten FTMG-Ansatzes erneut um ca. 7% gegenüber dem einfachen klassischen Verfahren.

Die Testreihe zum `andicore`-Problem liefert im fehlerbehafteten Fall ebenfalls ähnliche Resultate, wie das vorher betrachtete `andi`-Problem. Statistiken zu den einzelnen Tests finden sich erneut im Anhang (siehe Tabelle A.6). Da das fehlerfreie Verfahren bei diesem Problem bereits nach zehn Iterationen das Abbruchkriterium erreicht, ist, wie in Abbildung 5.2 zu erkennen, der absolute Zeitgewinn durch die Verwendung des FTMG-Ansatzes, im Vergleich zum modellierten Verfahren mit vollständig durch korrigierende Prüfsummen abgesicherte Operationen, geringer. Nichtsdestotrotz tritt mit Test 452 nur ein Fall auf, in dem der FTMG-Ansatz mehr Zeit benötigt als das modellierte Verfahren. Dies resultiert wieder aus einem zeitlich aufwendigen ABFT-Mechanismus. Die erzeugten Fehler werden hier erneut erst im späteren Verlauf erkannt und können so nicht ausreichend repariert werden, erzeugen jedoch einen hohen Aufwand bei der spä-

teren Detektion. Trotzdem konvergiert dieser Ansatz noch schneller, als das klassische unstabilisierte Verfahren.

Mit Test 488 tritt erstmals eine Situation auf, in der der FTMG-Ansatz mehr Laufzeit benötigt als das klassische Verfahren. Beide Verfahren benötigen eine zusätzliche Iteration im Vergleich zum ungestörten Fall. Da die Störung bereits in der zweiten Iteration auftritt ist die Detektion und Reparatur durch den ABFT-Mechanismus, wie in Kapitel 4 beschrieben schwer und kann nicht ausreichend umgesetzt werden. Insbesondere wird sie in diesem Fall erneut zu spät detektiert und somit nicht erfolgreich behoben. Der Umstand, dass im Folgenden keine weitere signifikante Störung auftritt, führt schließlich dazu, dass auch das klassische Verfahren nach 11 Iterationen konvergiert.

Ebenfalls treten beim `andicore`-Problem auch Situationen auf, in denen das klassische Verfahren divergiert (478, 496). In beiden Fällen konvergiert dagegen der FTMG-Ansatz. Die benötigte Laufzeit ist dabei jeweils geringer als beim vollständigen durch Prüfsummen abgesicherten Verfahren. Insbesondere ist bei Test 478 keine zusätzliche Iteration gegenüber dem fehlerfreien Fall notwendig, um das Abbruchkriterium zu erreichen.

Wie in der vorherigen Testreihe gibt es abermals Situationen in denen trotz Störungen auch das klassische Verfahren mit der üblichen Anzahl an Iterationen konvergiert (467, 468, 469, 494, 495). Diese fünf Fälle verursachen auch beim FTMG-Ansatz keine zusätzlichen Laufzeiten und sind somit nur mit den erwarteten 7% Mehraufwand verbunden.

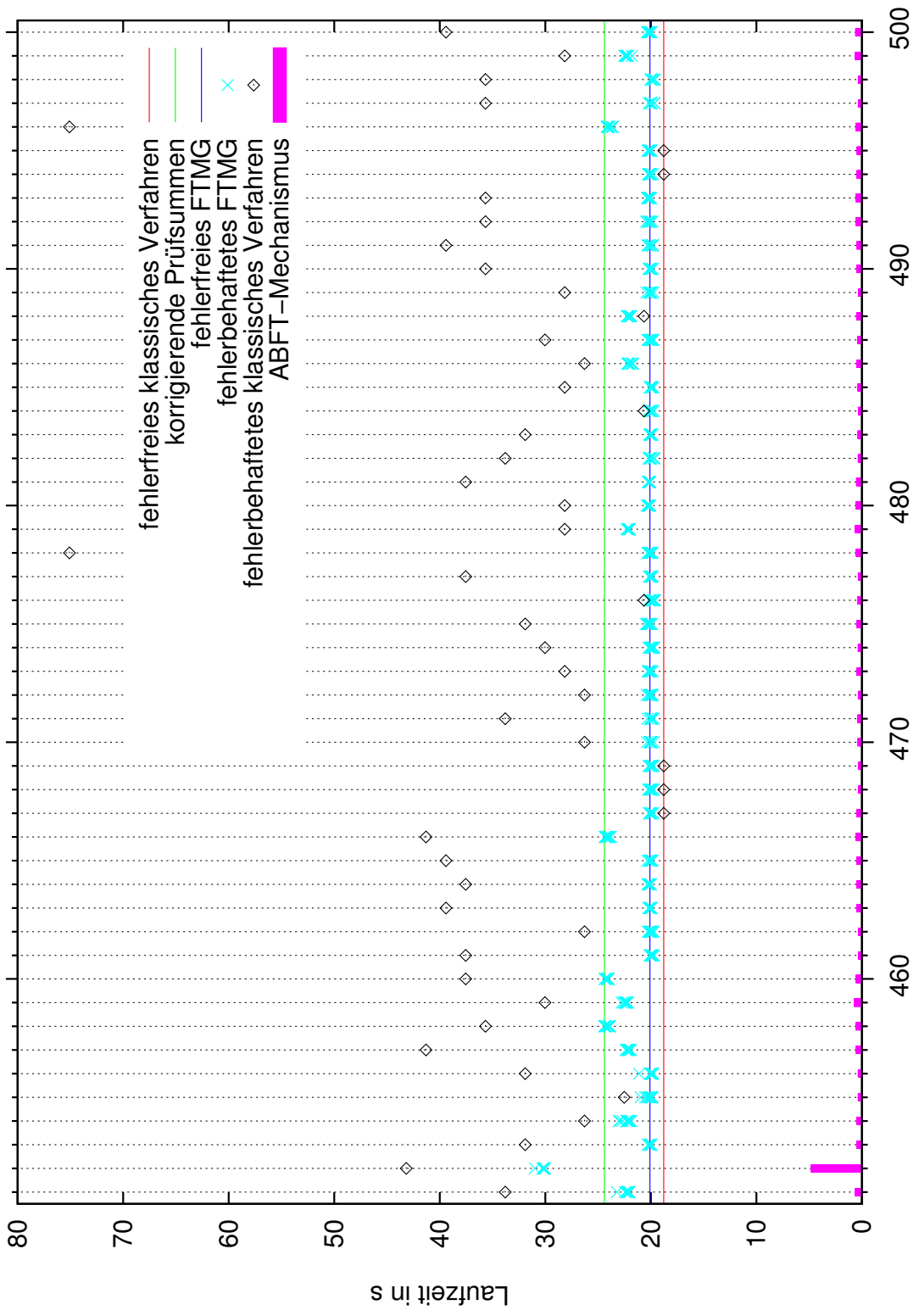

Abbildung 5.2: Laufzeiten der verschiedenen Tests zum `andicore`-Problem mit zusätzlicher Angabe der Zeit, die der ABFT-Mechanismus zur Detektion bzw. Reparatur benötigt.

5.3 Fazit: FTMG

In diesem Kapitel haben wir eine mögliche Variante eines vollständig fehlertoleranten Mehrgitterverfahrens konstruiert und anhand von numerischen Experimenten hinsichtlich Laufzeit und Konkurrenzfähigkeit untersucht. Die Analyse des verwendeten Reparaturmechanismus wurde dabei bereits in Kapitel 4 durchgeführt und spielte in diesem Kapitel nur eine untergeordnete Rolle.

Durch die Kombination der beiden in Kapitel 3 bzw. 4 vorgestellten Konzepte haben wir ein Verfahren konstruiert, das mit einem Mehraufwand von ungefähr 7% gegenüber dem klassischen Mehrgitterverfahren zusätzliche Fehlertoleranz besitzt. Im Vergleich dazu würde ein Ansatz, der das Verfahren vollständig durch Prüfsummen absichert, mit mindestens 30% zusätzlichem Aufwand verbunden sein. Die so gewonnene Laufzeit im fehlerfreien Fall tauscht das Verfahren dabei gegen unter Umständen notwendige Mehriterationen beim Auftreten von Fehlern ein. Die ausschließliche Verwendung von korrigierenden Prüfsummen für alle Operationen des Mehrgitterverfahrens würde bei jeglichen Fehlern, die unserem Fehlermodell entsprechen, dagegen dazu führen, dass das Abbruchkriterium nach der selben Anzahl an Iterationen, wie im fehlerfreien Fall, erreicht wird.

Die numerischen Tests in Kapitel 5.2 zeigen, dass der Ansatz im vorliegenden Fehlerszenario und bei den betrachteten Problemen durchaus attraktiv ist. Der Ansatz stabilisiert das Mehrgitterverfahren in vielen Fällen so, dass nicht einmal zusätzliche Iterationen durchgeführt werden müssen. In diesen Situationen führt das Verfahren zu einer Effizienzsteigerung von ungefähr 20% gegenüber einem Verfahren mit vollständiger Absicherung durch Prüfsummen.

Durch diese Effizienzsteigerung wird das Verfahren attraktiver je mehr Iterationen des Mehrgitterverfahrens durchgeführt werden. Die steigende Anzahl an Iteration ist mit einer Vergrößerung des Zeitgewinns verbunden, so dass dann beim Auftreten von Störungen, die nicht ausreichend detektiert oder repariert werden konnten, auch Mehriterationen noch innerhalb dieses Zeitfensters ausführbar sind. Innerhalb der durchgeführten numerischen Experimente traten nur wenige Probleme auf in denen dieses Budget an „freier Zeit“ nicht ausreichte.

Insgesamt ermöglicht der vorgestellte Ansatz also zusätzliche Stabilität ohne einen großen Mehraufwand im fehlerfreien Fall zu produzieren. Dieser geringe Mehraufwand ist zum aktuellen Zeitpunkt noch damit verbunden, dass in seltenen „worst-case“ Szenarien ein Anstieg der Rechenzeit in kauf genommen werden muss, wenn die Reparatur nicht hinreichend gut erfolgen kann. Dieser Nachteil lässt sich jedoch möglicherweise durch Modifikationen des ABFT-Detektionsmechanismus bzw. Erweiterungen der Reparatur, wie sie bereits in Kapitel 4.4 erläutert wurden, eliminieren oder zumindest verringern.

6 Zusammenfassung und Ausblick

In dieser Arbeit haben wir Konzepte untersucht, die es ermöglichen sollen die Fehlertoleranz des Mehrgitterverfahrens zu erhöhen. Dabei haben wir neben der Analyse des, aus der Welt der dichtbesetzten Matrizen stammenden, Ansatzes der Prüfsummen (siehe Kapitel 3) in Kapitel 4 einen Mechanismus konstruiert, der es ermöglicht die Ausgabe des Glätters zu kontrollieren und dabei zu überprüfen, ob die Daten zur Weiterverwendung geeignet sind sowie gegebenenfalls diese in einem gewissen Maße zu reparieren. Schließlich haben wir in Kapitel 5 ein Verfahren vorgestellt, das mit vergleichsweise geringem Mehraufwand das Mehrgitterverfahren gegenüber Fehlern innerhalb von numerischen Operationen (sogenannten *soft faults*, vgl. Kapitel 2.4) stabilisiert.

Die Analyse der Prüfsummen ergab, dass diese erst bei der Verwendung von hinreichend stark besetzten Matrizen und vielen Anwendungen attraktiv werden. Darüberhinaus sind komplexe Operationen, wie die Kopplung einer Matrix-Vektor-Multiplikation mit einer weiteren Vektor-Vektor-Addition, einfachen Operationen vorzuziehen. Hierbei kann der entstehende Mehraufwand durch die komplexere Operation besser überlagert werden. Außerdem haben wir erweiterte Varianten vorgestellt (vgl. Kapitel 3.3), diese Ideen jedoch im vorliegenden Zusammenhang als nicht hilfreich bewertet und verworfen.

Statt der durch Prüfsummen intuitiv möglichen Generierung einer algorithmenbasierten Fehlertoleranz beim Mehrgitterverfahren haben wir in Kapitel 4 Mechanismen entwickelt, die dafür geeignet sind zumindest die Operationen der Glättungsphase, die den Großteil des Aufwandes repräsentieren, zu überprüfen und teilweise zu reparieren. Dadurch war es uns möglich die Iterationszahlen, die beim Auftreten von Fehlern innerhalb dieser Phase, bis zur Konvergenz notwendig waren, gegenüber dem klassischen Verfahren, stark zu reduzieren.

Schließlich haben wir in Kapitel 5 ein Verfahren konstruiert, das durch Kombination der beiden Ansätze ein Mehrgitterverfahren erzeugt, welches mit geringem Mehraufwand in allen Komponenten eine zusätzliche Fehlertoleranz gegenüber dem klassischen Verfahren besitzt. In numerischen Tests haben wir dabei festgestellt, dass es, selbst in der vorliegenden Situation einer sehr hohen Fehlerwahrscheinlichkeit, meistens weniger Mehraufwand erzeugt als die Verwendung von korrigierenden Prüfsummen für alle Operationen. Der entscheidende Vorteil am konstruierten Verfahren ist jedoch die geringe Auswirkung im fehlerfreien Fall und die Toleranz von weniger signifikanten Störungen, die nicht zu Reparaturen oder Wiederholungen von Operationen führen.

Ausblick Die bisherigen Analysen des konstruierten ABFT-Verfahrens haben einige Schwachstellen offengelegt, die es noch zu verbessern gilt. Für spätere Arbeiten ist es interessant entsprechende Modifikationen, wie andere Vergleichswerte, Reparatur von feineren bzw. gröberen Gitterleveln oder das Verwerfen der Reparatur bei zu viel detektierten Komponenten, zu betrachten.

Darüber hinaus ist weiterhin die Fragestellung nach einer geeigneten Initialisierung der Vergleichswerte offen. Wir haben dazu bisher lediglich einige Ideen gesammelt, jedoch keine Tests zur Funktionsweise durchgeführt. Die bisherigen numerischen Tests haben dabei gezeigt, dass die von uns durchgeführte pauschale Initialisierung mit einem großen Wert ungeeignet ist, um Störungen innerhalb der ersten Iterationen ausreichend zu erkennen.

Zusätzlich dazu wäre es interessant die Einflüsse von komplexeren Glättern sowie die Tauglichkeit des Verfahrens bei komplexeren oder gar höherdimensionalen Problemen zu untersuchen. Insbesondere im Hinblick auf exascale-Systeme, die mittelfristig relevanter werden, gilt es dies zeitnah zu analysieren.

Literaturverzeichnis

[1] Anderson, E.; Bai, Z.; Bischof, C.; Blackford, S.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; Sorensen, D.: *LAPACK Users' Guide*. Third. Philadelphia, PA : Society for Industrial and Applied Mathematics, 1999. – ISBN 0-89871-447-8 (paperback)

[2] Blair, Vilray P.: The why and the how of harelip correction. In: *International Journal of Orthodontia, Oral Surgery and Radiography* 15 (1929), Nr. 11, S. 1112 – 1119. – URL `http://www.sciencedirect.com/science/article/pii/S0099696329904841`. – ISSN 0099-6963

[3] Bosilca, George; Delmas, Rémi; Dongarra, Jack; Langou, Julien: Algorithm-based fault tolerance applied to high performance computing. In: *Journal of Parallel and Distributed Computing* 69 (2009), April, Nr. 4, S. 410–416

[4] Braess, Dietrich: *Finite Elemente - Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*. Berlin [u.a.] : Springer, 1992. – ISBN 3-540-54794-0

[5] Briggs, William L.; Henson, Van E.; McCormick, Steve F.: *A Multigrid Tutorial: Second Edition*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 2000. – ISBN 0-89871-462-1

[6] Casas, Marc; Supinski, Bronis R. de; Bronevetsky, Greg; Schulz, Martin: Fault Resilience of the Algebraic Multi-grid Solver. In: *Proceedings of the 26th ACM International Conference on Supercomputing*. New York, NY, USA : ACM, 2012 (ICS '12), S. 91–100. – URL `http://doi.acm.org/10.1145/2304576.2304590`. – ISBN 978-1-4503-1316-2

[7] Dongarra, Jack; et al.: The International Exascale Software Project roadmap. In: *International Journal of High Performance Computing Applications* 25 (2011), Februar, Nr. 1, S. 3–60

[8] Dongarra, Jack; Hittinger, Jeffrey; Bell, John; Chacón, Luis; Falgout, Robert; Heroux, Michael; Howland, Paul; Ng, Esmond; Webster, Clayton; Wild, Stefan; Pau, Karen: Applied Mathematics Research for Exascale Computing / U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program. März 2014. – Forschungsbericht

[9] Elliott, James; Hoemmen, Mark; Mueller, Frank: Evaluating the Impact of SDC on the GMRES Iterative Solver. In: *Proceedings of the 2014 IEEE 28th In-*

ternational Parallel and Distributed Processing Symposium (IPDPS'14), Mai 2014, S. 1193–1202

[10] Elliott, James; Mueller, Frank; Stoyanov, Miroslav; Webster, Clayton: Quantifying the Impact of Single Bit Flips on Floating Point Arithmetic / North Carolina State University. März 2013 (TR-2013-2). – Forschungsbericht

[11] Ferreira, Kurt; Stearley, Jon; Laros, James H.; Oldfield, Ron; Pedretti, Kevin; Brightwell, Ron; Riesen, Rolf; Bridges, Patrick G.; Arnold, Dorian: Evaluating the Viability of Process Replication Reliability for Exascale Systems. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA : ACM, 2011 (SC '11), S. 44:1–44:12. – URL `http://doi.acm.org/10.1145/2063384.2063443`. – ISBN 978-1-4503-0771-0

[12] Fiala, David; Mueller, Frank; Engelmann, Christian; Riesen, Rolf; Ferreira, Kurt; Brightwell, Ron: Detection and correction of silent data corruption for large-scale high-performance computing. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, November 2012, S. 78:1–78:12

[13] Göddeke, Dominik; Altenbernd, Mirco; Ribbrock, Dirk: *Fault-tolerant finite-element multigrid algorithms with hierarchically compressed asynchronous checkpointing.* 2015. – eingereicht zur Veröffentlichung in Parallel Computing

[14] Hackbusch, Wolfgang: *Multi-grid methods and applications.* 2. Springer, Oktober 1985,2003

[15] Huang, Kuang-Hua; Abraham, J. A.: Algorithm-Based Fault Tolerance for Matrix Operations. In: *IEEE Trans. Comput.* 33 (1984), Juni, Nr. 6, S. 518–528. – URL `http://dx.doi.org/10.1109/TC.1984.1676475`. – ISSN 0018-9340

[16] Kilian, S.: *ScaRC: Ein verallgemeinertes Gebietszerlegungs-/Mehrgitterkonzept auf Parallelrechnern.* Logos Verlag, Berlin, Universität Dortmund, Dissertation, Januar 2001. – `http://www.logos-verlag.de/cgi-bin/buch?isbn=0092,` ISBN 978-3-8325-0092-4

[17] Kogge, Peter; Bergman, Keren; Borkar, Shekhar; Campbell, Dan; Carlson, William; Dally, William; Denneau, Monty; Franzon, Paul; Harrod, William; Hill, Kerry; Hiller, Jon; Karp, Sherman; Keckler, Stephen; Klein, Dean; Lucas, Robert; Richards, Mark; Scarpelli, Al; Scott, Steven; Snavely, Allan; Sterling, Thomas; Williams, R. S.; Yelick, Katherine: ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems / DARPA IPTO. September 2008. – Forschungsbericht

[18] Köster, M.: *Robuste Mehrgitter-Krylowraum-Techniken für FEM-Verfahren*, TU Dortmund, Fakultät für Mathematik, Lehrstuhl 3 für Angewandte Mathematik und Numerik, Diploma thesis, 2004

[19] Lammers, David: The Era of Error-Tolerant Computing. In: *IEEE Spectrum* 47 (2010), November, Nr. 11, S. 15–15

[20] Li, Dong; Chen, Zizhong; Wu, Panruo; Vetter, Jeffrey S.: Rethinking Algorithm-based Fault Tolerance with a Cooperative Software-hardware Approach. In: *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2013, S. 44:1–44:12

[21] Sao, Piyush; Vuduc, Richard: Self-stabilizing Iterative Solvers. In: *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA '13)*, 2013, S. 4:1–4:8

[22] Sloan, Joseph; Kumar, Rakesh; Bronevetsky, Greg: Algorithmic Approaches to Low Overhead Fault Detection for Sparse Linear Algebra. In: *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Washington, DC, USA : IEEE Computer Society, 2012 (DSN '12), S. 1–12. – URL `http://dl.acm.org/citation.cfm?id=2354410.2355166`. – ISBN 978-1-4673-1624-8

[23] Sloan, Joseph; Kumar, Rakesh; Bronevetsky, Greg: An Algorithmic Approach to Error Localization and Partial Recomputation for Low-overhead Fault Tolerance. In: *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Washington, DC, USA : IEEE Computer Society, 2013 (DSN '13), S. 1–12. – URL `http://dx.doi.org/10.1109/DSN.2013.6575309`. – ISBN 978-1-4673-6471-3

[24] Smith, Barry F.; Bjørstad, Petter E.; Gropp, William D.: *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, März 2004

[25] Stearley, Jon R.; Riesen, Rolf; Laros III, James H.; Ferreira, Kurt B.; Pedretti, Kevin; Oldfield, Ron A.; Brightwell, Ron: Redundant computing for exascale systems / Sandia National Laboratories. URL `http://dx.doi.org/10.2172/1011662`, December 2010 (SAND2010-8709). – Forschungsbericht

[26] Trottenberg, Ulrich; Oosterlee, Cornelius W.; Schuller, Anton: *Multigrid*. Academic Press, 2001

[27] Turek, Stefan; Göddeke, Dominik; Becker, Christian; Buijssen, Sven H.; Wobker, Hilmar: FEAST – Realisation of hardware-oriented Numerics for HPC simulations with Finite Elements. In: *Concurrency and Computation: Practice and Experience* 22 (2010), November, Nr. 6, S. 2247–2265

A Anhang

A.1 Statistiken der Tests zur algorithmenbasierten Fehlertoleranz für den Glättungsprozess

Test	Iterationen		#faults	#det	#false
	Klassisch	ABFT			
001	14	7	5	4	1
002	10	10	8	2	2
003	13	7	1	1	0
004	14	9	5	2	1
005	12	7	8	2	0
006	7	7	3	1	0
007	11	10	4	3	2
008	13	7	7	3	0
009	8	7	3	1	0
010	13	9	3	3	1
011	13	7	5	2	0
012	7	7	2	1	1
013	40	7	6	2	1
014	14	7	2	2	1
015	14	7	2	2	0
016	7	7	2	1	1
017	7	7	0	0	0
018	7	7	1	1	1
019	7	7	0	0	0
020	13	11	6	4	2
021	15	8	7	5	0
022	7	7	2	0	0
023	7	7	1	1	1
024	10	8	2	1	0
025	7	7	4	1	1
026	15	9	3	3	1
027	14	7	4	3	1
028	8	9	4	1	1
029	14	10	7	2	1
030	12	9	5	3	1
031	11	9	6	3	1
032	16	11	5	6	3
033	7	7	2	0	0
034	12	7	3	1	0

Test	Iterationen		#faults	#det	#false
	Klassisch	ABFT			
035	13	7	3	3	0
036	13	7	3	1	0
037	14	10	5	6	3
038	12	10	2	3	1
039	11	7	2	1	0
040	7	7	4	0	0
041	40	7	7	3	0
042	7	7	2	1	1
043	10	10	4	1	1
044	13	9	6	5	1
045	11	7	3	1	0
046	7	7	0	0	0
047	10	7	3	2	0
048	13	7	3	1	0
049	15	10	5	4	2
050	12	7	3	1	0
051	7	7	2	0	0
052	12	7	4	1	0
053	9	9	2	1	1
054	12	7	2	2	0
055	40	9	10	4	1
056	8	8	4	2	2
057	13	8	7	2	1
058	8	7	2	1	0
059	7	7	2	1	1
060	7	7	4	0	0
061	11	9	6	4	1
062	11	7	4	1	0
063	7	7	0	0	0
064	7	7	0	0	0
065	14	7	2	2	0
066	7	7	3	0	0
067	13	7	4	2	1
068	11	8	3	1	0

Test	Iterationen		#faults	#det	#false
	Klassisch	ABFT			
069	7	7	3	0	0
070	13	7	7	3	0
071	14	8	4	4	2
072	13	7	2	1	0
073	17	9	6	5	1
074	11	7	7	2	0
075	7	7	2	1	1
076	10	8	5	2	1
077	8	8	4	1	1
078	14	10	7	3	0
079	13	7	8	2	0
080	7	7	0	0	0
081	12	9	7	4	1
082	7	7	4	1	1
083	15	10	7	6	5
084	13	7	4	2	0
085	7	7	1	0	0
086	9	8	5	1	0
087	14	9	5	2	0
088	13	7	3	3	0
089	13	7	3	1	0
090	13	7	5	2	0
091	10	10	8	2	1
092	11	7	4	1	0
093	10	7	3	1	0
094	14	7	4	3	0
095	13	7	4	1	0
096	13	10	5	3	1
097	7	7	0	0	0
098	14	9	4	6	3
099	8	8	4	1	0
100	7	7	3	1	1

Tabelle A.1: Iterationszahlen der Verfahren zu den einzelnen fehlerbehafteten Tests des `poisson`-Problems mit Angabe der erzeugten Fehler (#**faults**), der korrekt detektierten Fehler (#**det**) sowie der fehlerhaften Detektionen (#**faults**).

Test	Iterationen		#faults	#det	#false
	Klassisch	ABFT			
101	29	18	15	6	2
102	40	18	17	6	0
103	25	18	10	3	0
104	32	18	10	6	0
105	40	18	14	4	0
106	29	19	10	6	1
107	31	18	11	3	0
108	28	18	8	5	0
109	40	18	11	7	0
110	40	18	18	12	1
111	30	18	7	5	0
112	29	18	11	5	0
113	28	18	8	5	0
114	30	18	14	5	0
115	32	20	8	6	1
116	29	18	11	8	2
117	29	18	9	3	0
118	40	21	14	7	1
119	21	18	11	3	0
120	37	18	11	7	0
121	24	18	8	7	1
122	37	21	11	6	3
123	36	19	13	7	3
124	29	18	11	5	0
125	30	18	8	1	0
126	23	18	7	2	0
127	33	19	11	11	5
128	25	18	11	3	0
129	27	18	12	5	1
130	26	18	7	3	0
131	34	18	9	7	0
132	27	18	10	5	0
133	31	18	6	2	0
134	36	18	8	5	0
135	19	18	5	1	0
136	34	18	14	5	0
137	33	18	12	6	0
138	37	19	10	4	1
139	27	18	21	10	1
140	27	18	8	4	0
141	40	18	10	7	1
142	40	18	13	5	0
143	26	18	15	5	1
144	40	18	13	9	1
145	27	18	3	1	0
146	34	19	9	6	1
147	29	18	10	5	0
148	26	18	7	2	0

Test	**Iterationen**		**#faults**	**#det**	**#false**
	Klassisch	**ABFT**			
149	26	21	8	4	3
150	26	19	6	5	1
151	33	18	7	4	0
152	33	18	12	5	0
153	23	18	10	3	0
154	33	18	11	5	0
155	18	18	2	1	0
156	25	20	10	6	1
157	32	18	6	3	0
158	24	18	3	2	0
159	32	18	8	4	0
160	24	18	13	6	1
161	40	18	10	5	0
162	34	19	11	6	2
163	35	18	9	4	1
164	30	18	11	5	0
165	32	18	9	4	0
166	27	18	11	3	0
167	25	18	7	4	0
168	32	18	16	6	0
169	24	18	6	3	0
170	40	18	8	4	0
171	31	18	13	6	0
172	25	18	9	3	0
173	29	20	11	6	1
174	40	18	18	6	0
175	24	18	11	6	1
176	40	18	13	5	0
177	29	18	8	4	0
178	31	18	12	4	0
179	36	18	8	3	0
180	23	18	8	2	0
181	40	18	7	5	0
182	34	18	12	7	0
183	28	18	13	4	0
184	40	18	7	4	0
185	32	18	8	3	0
186	25	18	8	3	0
187	38	18	15	7	0
188	33	18	13	9	2
189	30	18	12	8	0
190	30	18	17	11	0
191	30	18	6	3	1
192	23	18	11	2	0
193	35	18	6	2	0
194	27	18	11	7	1
195	28	18	10	4	2
196	27	18	4	5	1
197	40	21	13	10	4
198	28	18	13	5	0
199	25	18	9	2	0
200	28	18	14	5	0

Tabelle A.2: Iterationszahlen der Verfahren zu den einzelnen fehlerbehafteten Tests des `andi`-Problems mit Angabe der erzeugten Fehler (#**faults**), der korrekt detektierten Fehler (#**det**) sowie der fehlerhaften Detektionen (#**faults**).

Test	Iterationen		#faults	#det	#false
	Klassisch	ABFT			
201	14	8	6	4	0
202	13	8	3	1	0
203	15	9	4	2	1
204	16	10	7	7	3
205	10	9	3	1	0
206	8	8	5	0	0
207	13	9	7	4	1
208	9	8	5	2	0
209	11	8	5	2	0
210	8	8	6	0	0
211	8	8	5	0	0
212	40	8	3	2	0
213	14	10	7	4	1
214	10	8	3	1	0
215	9	8	7	2	0
216	12	8	7	2	0
217	14	9	10	6	2
218	40	8	6	2	0
219	15	8	4	1	0
220	13	8	3	1	0
221	16	9	8	2	0
222	8	8	2	0	0
223	12	8	6	1	0
224	16	9	4	4	1
225	40	8	3	2	0
226	40	11	7	4	2
227	11	9	7	4	1
228	8	8	3	0	0
229	10	8	10	2	0
230	14	8	4	3	0
231	15	8	8	3	0
232	8	8	1	1	0
233	15	8	2	2	0
234	16	10	4	4	2
235	12	8	5	2	0
236	15	9	7	5	1
237	11	8	6	2	0
238	10	8	2	1	0
239	13	9	5	3	2
240	14	8	3	1	0
241	11	8	6	2	0
242	13	8	3	1	0
243	11	8	7	1	0
244	12	10	3	4	3
245	8	8	0	0	0
246	16	9	7	4	0
247	10	8	5	2	0
248	8	8	0	0	0

Test	Iterationen		#faults	#det	#false
	Klassisch	ABFT			
249	15	8	6	1	0
250	11	9	2	2	1
251	11	10	4	3	1
252	12	8	8	1	0
253	13	8	2	1	0
254	10	10	3	2	1
255	14	8	6	3	0
256	12	8	6	2	0
257	8	8	2	0	0
258	13	9	5	3	0
259	8	8	1	0	0
260	12	8	3	1	0
261	10	8	4	1	0
262	13	8	3	2	0
263	18	10	7	3	2
264	17	10	8	5	1
265	9	8	5	1	0
266	15	8	5	2	0
267	11	8	4	1	0
268	13	8	3	1	0
269	10	8	1	1	0
270	10	8	3	2	0
271	9	8	5	2	0
272	9	9	4	1	1
273	12	8	5	1	0
274	14	8	5	3	0
275	15	8	7	3	1
276	8	8	3	0	0
277	12	8	5	1	0
278	17	11	8	4	2
279	15	8	6	4	0
280	11	8	5	3	0
281	14	10	10	6	2
282	14	10	2	2	1
283	10	10	4	2	2
284	12	8	1	1	0
285	13	9	5	3	1
286	8	8	4	0	0
287	13	8	7	3	1
288	15	8	5	2	0
289	8	8	5	0	0
290	11	8	4	2	0
291	8	8	5	0	0
292	13	8	10	3	0
293	15	9	10	5	1
294	13	8	4	2	0
295	9	9	7	0	0
296	12	8	4	2	0

Test	**Iterationen**		**#faults**	**#det**	**#false**
	Klassisch	**ABFT**			
297	8	8	7	0	0
298	10	10	10	3	1
299	15	8	3	1	0
300	8	8	7	0	0

Tabelle A.3: Iterationszahlen der Verfahren zu den einzelnen fehlerbehafteten Tests des `dico`-Problems mit Angabe der erzeugten Fehler (#**faults**), der korrekt detektierten Fehler (#**det**) sowie der fehlerhaften Detektionen (#**faults**).

Test	**Iterationen**		**#faults**	**#det**	**#false**
	Klassisch	**ABFT**			
301	17	14	3	2	1
302	11	11	5	2	1
303	21	10	6	2	0
304	16	10	7	1	0
305	15	10	4	1	0
306	20	10	6	2	0
307	14	10	3	2	0
308	10	10	3	0	0
309	19	10	7	4	0
310	17	10	7	4	0
311	14	11	6	3	1
312	40	10	8	4	0
313	19	11	9	2	1
314	16	10	4	2	0
315	17	11	5	2	0
316	20	10	6	2	0
317	13	10	8	1	0
318	20	10	6	4	0
319	10	10	5	1	0
320	20	12	6	5	1
321	20	10	5	3	0
322	18	10	6	2	0
323	18	13	8	5	1
324	23	13	14	6	2
325	10	10	2	0	0
326	19	10	4	2	0
327	17	10	10	3	0
328	19	11	7	3	0
329	14	10	4	1	0
330	14	11	6	2	0
331	20	10	6	2	0
332	20	10	5	2	0
333	13	11	5	2	0
334	20	13	11	5	1
335	10	10	0	0	0
336	21	11	8	5	1
337	19	12	8	2	1
338	12	10	4	1	0
339	15	11	8	3	0
340	40	11	6	3	0
341	10	10	4	0	0
342	10	10	3	0	0
343	14	10	6	3	0
344	20	11	10	5	0
345	18	10	4	1	0
346	10	10	3	0	0
347	40	10	6	5	0
348	15	10	2	2	0

Test	Iterationen		#faults	#det	#false
	Klassisch	ABFT			
349	23	11	9	6	1
350	20	12	8	4	1
351	19	12	8	3	1
352	15	11	5	3	1
353	15	10	4	3	0
354	15	11	7	1	0
355	17	13	10	3	1
356	19	10	3	1	0
357	12	10	3	2	0
358	19	10	2	1	0
359	16	12	5	5	2
360	13	10	7	3	0
361	14	10	9	2	0
362	18	13	7	4	1
363	20	10	7	3	0
364	14	11	7	3	1
365	20	10	3	1	0
366	11	11	9	1	1
367	18	10	4	1	0
368	22	10	2	1	0
369	15	10	3	1	0
370	20	10	10	3	0
371	16	11	6	3	0
372	17	10	4	4	0
373	40	10	9	5	0
374	14	11	6	4	0
375	40	11	8	3	0
376	40	10	7	2	0
377	15	10	6	1	0
378	10	10	1	0	0
379	21	11	5	3	0
380	40	10	6	3	0
381	10	10	6	1	0
382	11	10	9	1	0
383	20	12	8	6	2
384	18	10	3	1	0
385	10	10	7	0	0
386	14	11	6	3	0
387	40	10	6	3	0
388	17	10	3	1	0
389	15	10	1	1	0
390	20	10	8	3	0
391	12	10	4	1	0
392	10	10	6	0	0
393	11	10	4	1	0
394	21	12	7	5	1
395	14	10	3	2	0
396	17	11	5	3	1
397	40	10	3	3	0
398	16	10	5	3	0
399	20	10	11	3	0
400	17	10	3	2	0

Tabelle A.4: Iterationszahlen der Verfahren zu den einzelnen fehlerbehafteten Tests des `andicore`-Problems mit Angabe der erzeugten Fehler (#**faults**), der korrekt detektierten Fehler (#**det**) sowie der fehlerhaften Detektionen (#**faults**).

A.2 Statistiken der Tests zum fehlertoleranten Mehrgitterverfahren

Test	Iterationen		#faults	#det	#false
	Klassisch	FTMG			
401	14	14	7	0	0
402	22	14	7	5	0
403	33	17	11	5	1
404	14	14	4	1	0
405	33	19	11	7	3
406	25	14	8	4	0
407	23	14	6	4	0
408	29	14	7	4	0
409	28	14	6	3	0
410	24	14	7	3	0
411	27	14	8	5	0
412	22	15	7	5	1
413	40	14	6	4	1
414	31	18	16	9	2
415	20	14	11	5	1
416	26	14	13	3	0
417	26	14	6	3	0
418	40	18	14	6	3
419	30	14	10	8	0
420	24	17	9	6	0
421	14	14	6	4	0
422	26	14	7	6	0
423	40	14	6	2	0
424	21	14	10	7	2
425	24	14	9	4	0
426	22	14	6	4	0
427	29	18	11	7	3
428	31	14	7	6	0
429	21	14	5	2	0
430	14	14	3	0	0
431	34	17	12	8	2
432	26	14	8	3	0
433	29	14	7	5	0
434	23	14	8	4	0
435	19	14	5	3	0
436	14	14	10	1	0
437	30	14	9	4	0
438	27	14	4	3	0
439	25	14	12	9	1
440	15	14	6	3	0
441	40	14	13	8	1
442	27	14	6	4	0
443	24	14	11	8	0
444	14	14	6	0	0
445	22	14	8	5	2
446	26	14	8	5	0
447	40	14	9	5	0
448	26	14	9	3	0
449	26	14	3	3	2
450	28	14	9	6	0

Tabelle A.5: Iterationszahlen der Verfahren zu den einzelnen fehlerbehafteten Tests des `andi`-Problems mit Angabe der erzeugten Fehler (#**faults**), der korrekt detektierten Fehler (#**det**) sowie der fehlerhaften Detektionen (#**faults**).

Test	Iterationen		#faults	#det	#false
	Klassisch	FTMG			
451	18	11	7	4	1
452	23	13	9	5	1
453	17	10	4	3	0
454	14	11	5	2	1
455	12	10	2	1	0
456	17	10	4	2	0
457	22	11	5	5	2
458	19	12	11	5	1
459	16	11	5	5	1
460	20	12	11	6	1
461	20	10	9	4	0
462	14	10	4	2	0
463	21	10	7	3	0
464	20	10	7	2	0
465	21	10	5	2	0
466	22	12	8	4	1
467	10	10	3	1	1
468	10	10	5	0	0
469	10	10	0	0	0
470	14	10	6	3	0
471	18	10	7	5	0
472	14	10	8	4	0
473	15	10	7	3	0
474	16	10	4	2	0
475	17	10	7	3	1
476	11	10	3	2	0
477	20	10	5	1	0
478	40	10	7	4	0
479	15	11	7	7	1
480	15	10	4	2	0
481	20	10	6	3	0
482	18	10	8	4	0
483	17	10	5	3	0
484	11	10	6	1	0
485	15	10	4	1	0
486	14	11	5	2	1
487	16	10	11	6	0
488	11	11	6	1	1
489	15	10	6	2	0
490	19	10	6	4	0
491	21	10	6	1	0
492	19	10	4	2	0
493	19	10	4	3	0
494	10	10	4	1	0
495	10	10	5	1	0
496	40	12	5	4	2
497	19	10	2	1	0
498	19	10	5	1	0
499	15	11	4	2	1
500	21	10	9	3	0

Tabelle A.6: Iterationszahlen der Verfahren zu den einzelnen fehlerbehafteten Tests des `andicore`-Problems mit Angabe der erzeugten Fehler (#**faults**), der korrekt detektierten Fehler (#**det**) sowie der fehlerhaften Detektionen (#**faults**).

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

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

_____

_____

_____

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.

_____ _____

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:

_____ _____

Ort, Datum Unterschrift