Universität Passau

Fakultät für Informatik und Mathematik


Bachelorarbeit

# Implementierung und Evaluierung konfigurierbarer Mehrgitterlöser für Multiprozessorsysteme

Autor:

Christian Kaltenecker

28. September, 2014

Betreuer:

Prof. Dr.-Ing. Sven Apel

Software Product-Line Group

Alexander Grebhahn

Software Product-Line Group

Dr. Ing. Norbert Siegmund

Software Product-Line Group

**Kaltenecker, Christian:**
*Implementierung und Evaluierung konfigurierbarer Mehrgitterlöser für Multiprozessorsysteme*
Bachelorarbeit, Universität Passau, 2014.

# Inhaltsangabe

Heutzutage werden viele physikalische Vorgänge simuliert. Angefangen von der Wettervorhersage über die Simulation der Erdmantelkonvektion bis hin zur Simulation von Schmelzvorgängen. Ein Großteil dieser Simulationen werden als partielle Differentialgleichung dargestellt und müssen hinterher gelöst werden, etwa mithilfe des Mehrgitterverfahrens. Dabei kann das Mehrgitterverfahren mit den richtigen Komponenten sehr effizient sein. In dieser Arbeit beschäftigt man sich mit der Analyse und Erweiterung verschiedener Komponenten des Mehrgitterverfahrens sowie mit der Aufstellung von Hypothesen bezüglich der erwarteten Laufzeit.

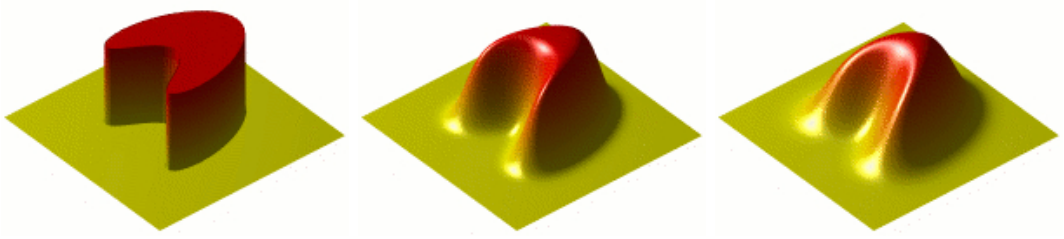
# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Quelltextverzeichnis

# 1. Einführung



(a) (b) (c)

Abbildung 1.1: Simulation der Wärmeleitungsgleichung, welches das Schmelzen eines Objektes in 3 Stadien berechnet. [2]

Simulationen haben heutzutage einen großen Einfluss auf unser Leben. So wird das morgige Wetter als auch das Klima der nächsten Dekade durch Simulationen vorhergesagt [Buc10]. Daneben spielen Simulationen auch eine immer größere Bedeutung in der Industrie. Eigenschaften von Materialien, wie zum Beispiel die Schmelzeigenschaft werden über die Wärmeleitungsgleichung simuliert (wie in Abbildung 1.1 in drei Schritten dargestellt) [HPRW04]. Die zugrundeliegende Funktionsweise vieler Simulationen ist die Lösung von partiellen Differentialgleichungen (PDG). So werden die eigentlichen Probleme, wie zum Beispiel die Berechnung der Erdmantelkonvektion, in partiellen Differentialgleichungen mathematisch ausgedrückt und durch spezielle Lösungsverfahren errechnet [Kam05]. Diese PDG enthalten beliebig viele Variablen und werden mit zunehmender Anzahl aufwendiger zu lösen. Es gibt zwei Arten von PDG: lineare und nichtlineare. Sie bestimmen, welche computergestützte Verfahren angewendet werden. Im Folgenden werden mehrere Arten von Verfahren vorgestellt, welche PDG lösen.

### Direktes Verfahren

Das direkte Verfahren löst die PDG ohne zusätzliche Änderungen an der eigentlichen PDG vorzunehmen, weswegen dies für kleinere Gleichungssysteme aufgrund

[2] http://de.wikipedia.org/wiki/Wärmeleitungsgleichung, Letzter Zugriff am 24 Juli 2014

der Schnelligkeit zu bevorzugen ist. Für eine größere Anzahl an Unbekannten wächst der Aufwand so sehr, dass andere Verfahren bezüglich der Ausführung weniger Zeit brauchen. Dies lässt sich dadurch erklären, dass andere Verfahren die PDG so umformen, dass sich das Gleichungssystem leichter lösen lässt. Dies müssen nicht zwingend äquivalente Umformungen sein, so dass bei anderen Verfahren ein Fehler mit einfließen könnte. Das direkte Verfahren ist im Vergleich zu anderen Verfahren genau. Ein Beispiel hierzu wäre das *gaußsche Eliminationsverfahren* (für $N$ Unbekannte liegt der Aufwand bei $O(N^3)$).

### Iteratives Verfahren

In der numerischen Mathematik wird mit dem iterativen Verfahren eine Methode bezeichnet, womit man sich der genauen Lösung eines Problems schrittweise annähert. Abgebrochen wird dieses Verfahren, sobald die Lösung eine gewisse Genauigkeit erreicht hat.
Inzwischen gibt es eine große Auswahl an iterativen Verfahren. Einige davon sind:

- *Jacobi-Verfahren*
- *Gauß-Seidel-Verfahren*
- ***Mehrgitterverfahren*** (der Aufwand kommt hier auf die verwendeten Algorithmen an. Es kann ein Aufwand von $O(N \log \epsilon)$ für $N$ Unbekannte und einer Konstante $\epsilon$ erzielt werden)
- *vorkonditionierte Krylow-Unterraum-Verfahren*
  - *Konjugierten Gradienten*
  - *Generalized minimal residual method (GMRES)*

Dies ist nur ein Bruchteil der Verfahren, die man zur Lösung von partiellen Differentialgleichungen verwendet. Diese Arbeit konzentriert sich auf Lösungstechniken durch das Mehrgitterverfahren, da dieses Verfahren gut skaliert und der Aufwand bei PDG mit vielen Unbekannten noch geringer ist, als bei anderen Verfahren.
Das Mehrgitterverfahren selbst ist eher für die linearen Differentialgleichungen gedacht, kann jedoch auch für nichtlineare partielle Differentialgleichungen eingesetzt werden.
Das Verfahren lässt sich zudem auf zwei weitere Arten unterteilen:

- Geometrische Mehrgitterverfahren (GMG)
- Algebraische Mehrgitterverfahren (AMG)

Bei GMG wird das PDG durch Diskretisierung in ein Gitter umgewandelt, auf welchem verschiedene geometrische Operationen ausgeführt werden. Etwa kann man die Gittergröße sowie -weite ändern und anhand dessen eine Vergröberung oder eine

Verfeinerung des Gitters erzielen. Bei komplexen Geometrien erreicht das klassische (bzw. geometrisches) Mehrgitterverfahren schnell seine Grenzen. Deshalb entstand das AMG, welches ausschließlich mit Modifikationen der Gleichungssysteme arbeitet und nicht wie das klassische Mehrgitterverfahren mit Änderungen der Gitterweite und anderen geometrischen Elementen. Auf das AMG wird in dieser Arbeit nicht mehr weiter eingegangen, da sich die in Kapitel 4 vorgestellten Frameworks und Programme nur auf das GMG beziehen.
In der restlichen Arbeit bezieht man sich immer auf geometrische Mehrgitterverfahren.

**Problemstellung: Variabilität**

Es gibt mehrere Frameworks, die eine Bibliothek von Algorithmen zur Realisierung des Mehrgitterverfahrens anbieten. Im Rahmen dieser Arbeit werden die zwei prominenten Vertreter Dune und Hypre betrachtet und in Kapitel 4 werden diese ausführlicher erklärt.

Das Mehrgitterverfahren hat einen minimalen Aufwand von $O(N \log \epsilon)$, wobei $N$ die Anzahl der Unbekannten und $\epsilon$ eine Konstante ist. Jedoch müssen dafür bestimmte Algorithmen verwendet werden, welche vom Aufwand her $O(N \log \epsilon)$ nicht überschreiten. In der Regel bedeutet ein niedriger Aufwand auch einen größeren Fehler bei der Lösung der PDG. Der praktische Erfolg einer Simulation hängt somit davon ab, die für die jeweilige Aufgabe passenden Algorithmen zu wählen.

Dabei ist die große Auswahl an Algorithmen nicht nur ein Vorteil, sondern für den Anwender auch ein Nachteil, da die Algorithmen unterschiedlich kombiniert werden können. Dadurch ist es nicht klar, welche man auswählt, um die Algorithmen bezüglich der Performance für die eigene Aufgabe zu optimieren und wie man diese Variabilität durch die Kombination verschiedener Algorithmen technisch handhabt, ohne neue Probleme hervorzurufen und von vorne anfangen zu müssen. Deshalb stützt man sich im Rahmen dieser Arbeit auf zwei Programme, HSMGP und MGS. Diese werden erweitert und anschließend anhand ihrer Resultate bezüglich Laufzeit und Eigenschaften des Mehrgitterverfahrens analysiert.

## 1.1 Zielstellung der Arbeit

Die Arbeit hat zwei Ziele: Erstens wird eine Übersicht über die bestehende Variabilität des Mehrgitterverfahrens gegeben. Dafür muss das Mehrgitterverfahren mitsamt der dafür benötigten Komponenten verstanden werden. Mit diesem Wissen wird die Variabilität der Programme Hypre und MGS analysiert. Außerdem wird eine Übersicht über die Variabilität dieser Programme durch visuelle Darstellungen gegeben. Unter Variabilität sind die Einstellungsmöglichkeiten gemeint, die man vornehmen kann, um unterschiedliche Ausführungszeiten und/oder unterschiedliche Genauigkeiten der Resultate zu erhalten. Zu den Einstellungsmöglichkeiten gehören nicht nur die Auswahl des Algorithmus zum Lösen des Mehrgitterverfahrens, sondern auch

diverse andere Optionen, wie zum Beispiel das Mitzählen und Ausgeben der Lösungsschritte.

Zweitens werden diese Programme erweitert, um deren Mächtigkeit zu erhöhen sowie um noch mehrere Eigenschaften des Mehrgitterverfahrens nutzen zu können. Ein Beispiel für eine Eigenschaft des Mehrgitterverfahrens ist die zugrundeliegende Zyklenart (darauf wird in Abschnitt 2.3.6 eingegangen). Diese Erweiterungen werden in Kapitel 4 anhand von beispielhafter Implementierung vorgestellt und erklärt.

Basierend auf der Analyse der Variabilität und deren Erweiterung ist es nun erstmals möglich auch Hypothesen bezüglich der erwarteten Laufzeit von Konfigurationen aufzustellen. Im Rahmen dieser Arbeit werden exemplarisch für ausgewählte Algorithmen solche Hypothesen aufgestellt und evaluiert, so dass nachfolgende Arbeiten hierauf aufbauen können.

## 1.2 Gliederung der Arbeit

Bisher wurde das Thema lediglich im Allgemeinen erklärt und um einen tieferen Einblick zu bekommen werden in den folgenden Kapiteln unter anderem jegliche, für das Verstehen der Programme nötige Hintergrundinformationen geliefert.

Kapitel 2 beschreibt das Mehrgitterverfahren genauer. Darunter wird die ganze Theorie des Mehgitterverfahrens erklärt. Bei der Beschreibung wird ein großer Wert auf die Stärken und Schwächen der unterschiedlichen Algorithmen gelegt, um diese nachvollziehen zu können.
Weitere Hintergrundinformationen, welche nicht direkt das Mehrgitterverfahren betrifft werden in Kapitel 3 behandelt. In diesem Kapitel geht es um die visuelle Darstellung von Erweiterungen, sogenannten Features. Dies wird im späteren Verlauf der Arbeit gebraucht, um die bisherige Variabilität mit der durch diese Arbeit erzielten Variabiltät zu vergleichen.
In Kapitel 4 werden die in dieser Arbeit verwendeten Frameworks Hypre und DUNE näher erklärt, genauso wie die Programme, denen im Rahmen dieser Arbeit noch zusätzliche Erweiterungen hinzugefügt werden. Dabei wird vor allem auch auf die Konfigurationsmöglichkeiten und auf die Implementierung der Erweiterungen an sich eingegangen.
Die Bewertung der Erweiterungen erfolgt in Kapitel 5. Dies wird hauptsächlich durch das Durchführen von Tests und durch Auswertung der Testresultate erzielt. Zudem werden in diesem Kapitel die Performanceergebnisse verschiedener Algorithmen gegenübergestellt.
Schlussendlich wird in Kapitel 6 ein Fazit aus der in dieser Arbeit vorgestellten Programmen und Erweiterungen gezogen und ein Ausblick auf weitere Anknüpfungspunkte für weitere Arbeiten gegeben.

# 2. Geometrisches Mehrgitterverfahren

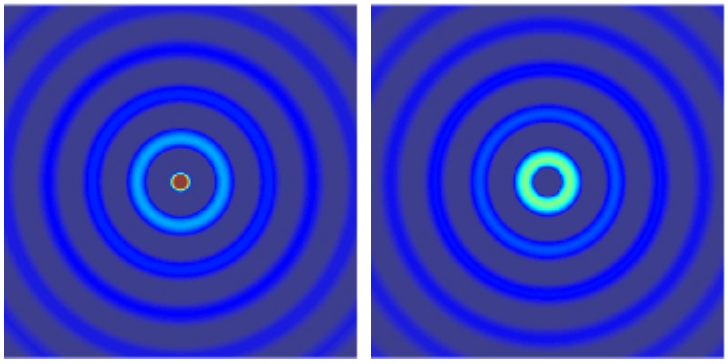
In diesem Kapitel wird das geometrische Mehrgitterverfahren, angelehnt an [TS01] von A. Schuller und U. Trottenberg, vorgestellt. Das Verständnis dieses Kapitels ist für die im späteren Verlauf vorgestellten Frameworks und Programme essentiell.

Zuerst wird das grundsätzliche Problem in Abschnitt 2.1 dargestellt für dessen Lösung man oft das Mehrgitterverfahren verwendet.
Im Anschluss werden in Abschnitt 2.2 die Grundbegriffe erläutert. Diese sind für das Verständnis des Mehrgitterverfahrens nötig.
Die einzelnen Operationen, welche im Rahmen des Mehrgitterverfahrens angewandt werden, werden in Abschnitt 2.3 näher beschrieben.
Schließlich wird das gesamte Mehrgitterverfahren in Abschnitt 2.4 als Pseudocode dargestellt und dabei noch einmal auf die einzelnen Operationen und deren Variabilität eingegangen.

## 2.1 Problemstellung

Es gibt verschiedene Arten von Problemen, welche sich mit dem Mehrgitterverfahren lösen lassen. Ursprünglich wurde das Mehrgitterverfahren für poissonartige Probleme entwickelt. Poissonartige Probleme sind Gleichungssysteme, welche auf der Poisson-Gleichung basieren, worauf im Verlauf dieses Abschnitts noch weiter eingegangen wird.
Doch auch andere zweidimensionale partielle Differentialgleichungen (im Weiteren auch abgekürzt als 2D PDG), die nicht-elliptisch sind, können in ein Mehrgitter umgewandelt und gelöst werden. Diese Gleichungen sind normalerweise von der Form: $Pu = f$ in einem zweidimensionalen Raum $\Omega \in \mathbb{R}^2$ mit dem Ausgabe-Vektor $f$, dem Differentialoperator $P$ und der gesuchten Funktion der Differentialgleichung $u$ wobei $a_{ij}, a_i$ und $a_0$ dabei die Koeffizienten sind.

$$Pu = a_{11}u_{xx} + a_{12}u_{xy} + a_1u_x + a_2u_y + a_0u$$



(a) (b)

Abbildung 2.1: Simulation der Wellengleichung. Durch die Simulation wird die Bewegung der Wellen dargestellt. Links (a) ist der Zeitpunkt dargestellt, an dem im Zentrum eine Welle entsteht. In (b) wird die gleiche Welle in einem späteren Zeitpunkt dargestellt [2].

Das bekannteste Beispiel zur obigen Gleichung ist die Poisson-Gleichung:

$$-\Delta u = -u_{xx} - u_{yy} = f$$

Dabei bezeichnet $\Delta$ den aus der Mathematik bekannten Laplace-Operator. $f$ ist dabei eine Funktion und $u$ die gesuchte Lösung. $x$ und $y$ sind die Koordinaten im zweidimensionalen Raum. Mit $u_{xy}$ wird zudem die Lösung im zweidimensionalen Raum an der Stelle $(x, y)$ bezeichnet.

Andere Beispiele zur obigen Gleichung sind:

- Die Wellen-Gleichung: $u_{xx} - u_{yy} = 0$ Diese Gleichung wird dazu verwendet, Wellen zu simulieren (siehe Abbildung 2.1).
- Die Hitze-Gleichung: $u_{xx} - u_y = 0$ Durch das Anwenden dieser Gleichung ergibt sich die Simulation des Schmelzvorgans (siehe Abbildung 1.1).

## 2.2 Grundbegriffe

Im Folgenden werden einige Grundbegriffe und Erklärungen vorgestellt, welche dabei helfen, das Mehrgitterverfahren besser zu verstehen.

### 2.2.1 Gitter

Das Gitter entsteht durch Diskretisierung aus dem Gleichungssystem, das heißt, dass die Gleichungen auf das Gitter abgebildet werden. Dafür verwendet man oft die finite Elemente und finite Differenzen Methode. Darauf wird in [Dor97] sowie

[2] http://de.wikipedia.org/wiki/Wellengleichung, Letzter Zugriff am 28 Juli 2014

[ZC06] genauer eingegangen und wird im Rahmen dieser Arbeit nicht näher behandelt, da die im Verlauf der Arbeit vorgestellten Programme die Gitter automatisch generieren.

Bei dem Gitter sind alle Überschneidungen von horizontalen und vertikalen Linien Unbekannte im ursprünglichen Gleichungssystem.
Im Mehrgitterverfahren geht es darum, eine Lösung für das Gitter zu finden, welche wiederum die Lösung für die zugrundelegende PDG ist. Manche Operationen des Mehrgitterverfahrens verursachen einen Fehler in der Lösung. Das bevorzugte Ziel ist dabei im Allgemeinen einen möglichst geringen Fehler beziehungsweise eine möglichst kleine Abweichung von der genauen Lösung.

Beim Mehrgitterverfahren ist eine ungenaue Lösung schnell gefunden. Der Fehler wird durch das wiederholte Anwenden des Mehrgitterverfahrens reduziert.
In diesem Zusammenhang wird der Begriff der Konvergenzrate eingeführt, der das schrittweise Annähern der Lösung zu einer möglichst guten Lösung bezeichnet.
Wie der Abbildung 2.2 zu entnehmen ist, werden in einem Gitter sowohl die inneren Punkte, als auch die Randpunkte berücksichtigt. Die Eigenschaft, dass man nur eine Teilmenge aller Knoten berücksichtigt, wird vor allem für Vergröberung des Gitters (oder auch Restriktion, Abschnitt 2.3.1) und die Verfeinerung (oder auch Prolongation, Abschnitt 2.3.4) bezüglich der Stempel verwendet. Die Stempel werden im Folgenden näher behandelt.

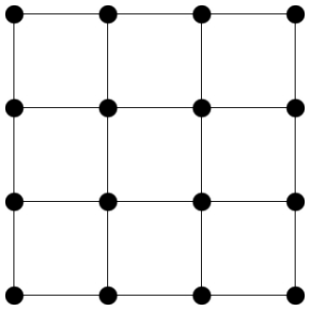
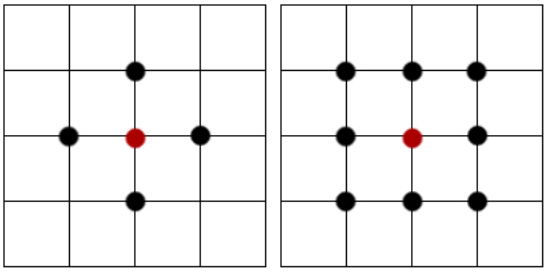


Abbildung 2.2: Ein 4x4 Gitter im zweidimensionalen Fall. Jeder Punkt, an dem sich zwei Linien schneiden, ist eine Variable. Durch dieses Gitter werden 16 Variablen dargestellt.

### 2.2.2 Stempel

Durch einen Stempel (übersetzt aus dem englischen Begriff „Stencil“) lässt sich die Relation eines Punktes zu den benachbarten Punkten beschreiben. Dieser wird zum Glätten benötigt, wobei der Stempel dabei festlegt mit welchem Gewicht die umliegenden Knoten bei der Glättung eines Punktes mit einfließen. Ein Stempel ist eine Matrix und sieht wie folgt aus:



(a) 5-Punkt-Stempel (b) 9-Punkt-Stempel

Abbildung 2.3: Grafische Darstellung von 5-Punkt (a) und 9-Punkt (b) Stempel. Der rote Punkt ist dabei der Ausgangspunkt und die restlichen Punkte sind die benachbarten Punkte, zu dem eine Beziehung besteht.

$$\begin{bmatrix} & \vdots & \vdots & \vdots & \\ \cdots & s_{-1,1} & s_{0,1} & s_{1,1} & \cdots \\ \cdots & s_{-1,0} & s_{0,0} & s_{1,0} & \cdots \\ \cdots & s_{-1,-1} & s_{0,-1} & s_{1,-1} & \cdots \\ & \vdots & \vdots & \vdots & \end{bmatrix}$$

$s_{0,0}$ ist dabei der Ausgangspunkt. Die Koeffizienten können dabei nahezu beliebig gewählt werden. Daher gibt es auch eine große Variation an Stempeln, welche durchgehend verwendet werden. Beispielsweise könnte man auch nur die Koeffizienten auf der Diagonale ($s_{-1,1}$, $s_{0,0}$, $s_{1,-1}$, $s_{1,1}$ und $s_{-1,-1}$) nehmen.
Die Stempel, die man in dieser Arbeit hauptsächlich benutzen wird, sind die 5-Punkt (five-point) und die 9-Punkt (nine-point) Stempel. Der Name leitet sich von der Anzahl der Punkte ab, die im Stempel eine von 0 verschiedene Gewichtung bekommen. Die 5- und 9-Punkt Stempel sehen wie folgt in Matrixform aus:

$$\begin{bmatrix} & s_{0,1} & \\ s_{-1,0} & s_{0,0} & s_{1,0} \\ & s_{0,-1} & \end{bmatrix}$$

und

$$\begin{bmatrix} s_{-1,1} & s_{0,1} & s_{1,1} \\ s_{-1,0} & s_{0,0} & s_{1,0} \\ s_{-1,-1} & s_{0,-1} & s_{1,-1} \end{bmatrix}$$

Dabei ist $s_{0,0}$ die Gewichtung des Punktes, welcher geglättet wird. Die restlichen Gewichtungen sind die der Nachbarn. Grafisch dargestellt würden diese beiden Stempel wie in Abbildung 2.3 aussehen.

Ein Beispiel für einen 5-Punkt-Stempel ist im Folgenden gegeben:

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}$$

Dabei wird der geglättete Punkt mit der Gewichtung 4 versehen. Alle Nachbarpunkte fließen bei der Glättung mit der Gewichtung -1 ein. Alle anderen Punkte haben die Gewichtung 0 und spielen daher keine Rolle.

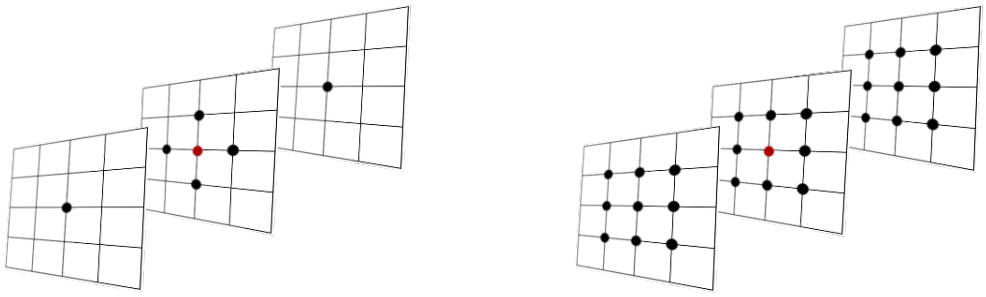
Die 5-Punkt und 9-Punkt-Stempel sind die Stempel, die wegen ihrer Einfachheit sehr oft verwendet werden. Auch in [TS01] werden hauptsächlich diese für den zweidimensionalen Raum verwendet.
Normalerweise werden der 7-Punkt und 27-Punkt Stempel im dreidimensionalen Raum genutzt. Diese sehen wie folgt aus, wobei $x, y, z$ aus $s_{x,y,z}$ für die jeweiligen Dimensionen stehen:

$$\begin{bmatrix} \\ s_{0,0,-1} \\ \end{bmatrix} \begin{bmatrix} & s_{0,1} & \\ s_{-1,0} & s_{0,0} & s_{1,0} \\ & s_{0,-1} & \end{bmatrix} \begin{bmatrix} \\ s_{0,0,1} \\ \end{bmatrix}$$

und

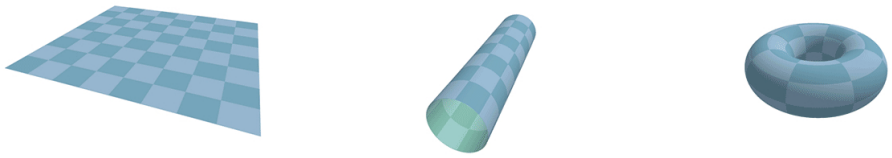
$$\begin{bmatrix} s_{-1,1,-1} & s_{0,1,-1} & s_{1,1,-1} \\ s_{-1,0,-1} & s_{0,0,-1} & s_{1,0,-1} \\ s_{-1,-1,-1} & s_{0,-1,-1} & s_{1,-1,-1} \end{bmatrix} \begin{bmatrix} s_{-1,1,0} & s_{0,1,0} & s_{1,1,0} \\ s_{-1,0,0} & s_{0,0,0} & s_{1,0,0} \\ s_{-1,-1,0} & s_{0,-1,0} & s_{1,-1,0} \end{bmatrix} \begin{bmatrix} s_{-1,1,1} & s_{0,1,1} & s_{1,1,1} \\ s_{-1,0,1} & s_{0,0,1} & s_{1,0,1} \\ s_{-1,-1,1} & s_{0,-1,1} & s_{1,-1,1} \end{bmatrix}$$

Visualisiert man diese Stempel, so sehen die beiden Stempel, der 7-Punkt-Stempel und der 27-Punkt-Stempel so aus, wie in Abbildung 2.4. K. Datta hat in [Dat09] eine noch ausführlichere Beschreibung von Stempeln.
Für Knoten am Rand gibt es sogenannte Randbedingungen. Diese werden im Folgenden näher erläutert.



(a) 7-Punkt-Stempel (b) 27-Punkt Stempel

Abbildung 2.4: Zwei Stempel, die im dreidimensionalen Raum verwendet werden. Der rote Punkt ist dabei der Punkt, welcher geglättet wird.



(a) (b) (c)

Abbildung 2.5: Visuelle Darstellung einer periodischen Randbedingung [3].

### 2.2.3 Randbedingungen

Für die Knoten am Rand muss entweder der Stempel(siehe Abschnitt 2.2.2) verändert oder dem Gitter Punkte hinzugefügt werden. Aus diesem Grund gibt es mehrere Randbehandlungsmethoden:

- *Dirichlet-Randbedingungen*: Die Dirichlet-Randbedingungen werden am häufigsten benutzt. Dabei wird direkt ein Funktionswert (beispielsweise der Wert 0) vorgegeben (siehe [Jen11]). Außerdem kann man zum Beispiel bei der Hitze-Gleichung eine Funktion vorgeben, die jedem Randpunkt eine feste Temperatur zuweist.

- *Neumann-Randbedingungen*: Bei diesem Verfahren werden Geisterpunkte angelegt. Dabei wird die Normalableitung vorgegeben (siehe [Jen11]). Beispielsweise kann man bei der Hitze-Gleichung die Funktion für Wärmeleitung als Randbedingung verwenden. Diese Art von Randbedingung ist also variabel.

- *periodische Randbedingungen*: Hier werden ebenfalls Geisterpunkte angelegt. Allerdings besteht hier der Unterschied, dass die Werte vom Rand der anderen Gitterseite genommen werden. Die Werte der linken Geisterpunkte stimmen mit den Werten der rechten Randpunkte des Gitters überein. Eine Visualisierung davon findet man in Abbildung 2.5. In dem ersten Bild der Abbildung (a) sieht man das ursprüngliche Gitter. Dann verbindet man durch die Geisterpunkte die linken Randpunkte mit den rechten Randpunkten. Das heißt, dass die linken Nachbarn der linken Randpunkte die rechten Randpunkte sind (b). Selbiges wird mit der vorderen und hinteren Seite gemacht (c). Dabei ist zu beachten, dass das Gitter dabei nicht verzerrt wird, so wie es im rechten Bild (c) dargestellt ist.

Gute Beispiele für die Verwendung in der Praxis sind in der Arbeit von A. McAdams et al. in [MST10] aufgeführt. Vor allem die Dirichlet- und Neumann-Randbedingungen werden dabei behandelt.

[3] http://de.wikipedia.org/wiki/Periodische_Randbedingung, Letzter Zugriff am 24 Juli 2014

## 2.3 Funktionsweise

Das Mehrgitterverfahren ist eine iterative Methode PDG zu lösen. Wie schon in Abschnitt 2.2.1 erwähnt, verwendet man eine Diskretisierung, um aus dem partiellen Differentialgleichungen ein äquivalentes Gitter aufzustellen. Dieses feine Gitter bildet das Ausgangsgitter für die nächsten Schritte.
Gegenüber anderen Verfahren, wie etwa dem gaußschen Eliminationsverfahren, versucht das Mehrgitterverfahren den Aufwand für das Lösen des Gitters zu minimieren. Dies wird vor allem durch Vergröberung des Gitters erzielt, da das Lösen auf dem ursprünglich sehr feinem Gitter zu hoch ist. Auf die Vergröberung wird in Abschnitt 2.3.1 näher eingegangen. Diese wird beim Mehrgitterverfahren mehrfach angewandt. Durch die mehrfache Vergröberung wird das Gitter immer gröber. Damit vermindert sich auch der Berechnungsaufwand zum Lösen des Gitters, da auch die Anzahl der Variablen erheblich gesenkt wird.

Sobald das Gitter gelöst wurde, muss das Gitter wieder verfeinert werden, um zum ursprünglichen Problem zu gelangen. Eine ganze Anwendung mit den zuvor erwähnten Glätter-, Vergröberungs-, den Lösungs- und den Verfeinerungsschritten nennt man Zyklus (vergleiche Abschnitt 2.3.6). Wichtig ist dabei noch, dass ein Zyklus mehrfach angewandt werden muss, um eine möglichst genaue Lösung zu erhalten. Das einmalige Anwenden eines Zyklus wird außerdem Iteration genannt.

### 2.3.1 Restriktion

Die Restriktion oder in dieser Arbeit auch Vergröberung genannt, bezeichnet den Prozess für das Verkleinern des Gitters. Damit ist die Restriktion eine Abbildung von einem feinen zu einem gröberen Gitter. Dadurch wird aus dem Problem des feinen Gitters ein sogenanntes „Hilfsproblem“ aufgestellt, welches weniger Variablen enthält als das Problem des feinen Gitters. Die Vergröberung wird in Abbildung 2.6 allgemein und in Abbildung 2.7 etwas detaillierter veranschaulicht. In der detaillierteren Ansicht ist zu erkennen, wie aus vier Blöcken ein Block interpoliert wird. Zuerst werden die Ecken von links nach rechts und von unten nach oben nummeriert und daraufhin die restlichen Punkte gegen dem Uhrzeigersinn. Der Ausgangspunkt ist dabei unten links.
Die Restriktion betrachtet immer vier Blöcke, welche schließlich durch die Restriktion auf einen Block abgebildet werden.
Dabei können unterschiedliche Vergröberungsoperatoren benutzt werden, welche die Vergröberung ausführen. Diese unterscheiden sich anhand ihrer Laufzeits-, Parallelitäts- und Effektivitätseigenschaft.

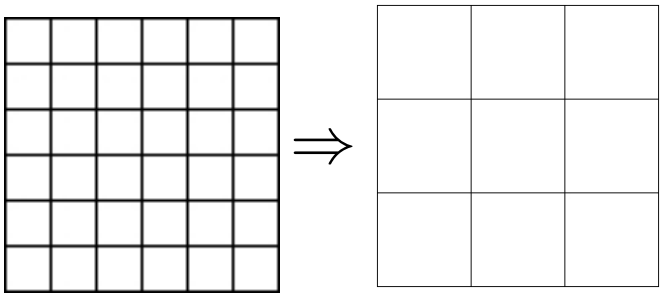


Abbildung 2.6: Eine Vergröberung von der Größe 6x6 zu 3x3.

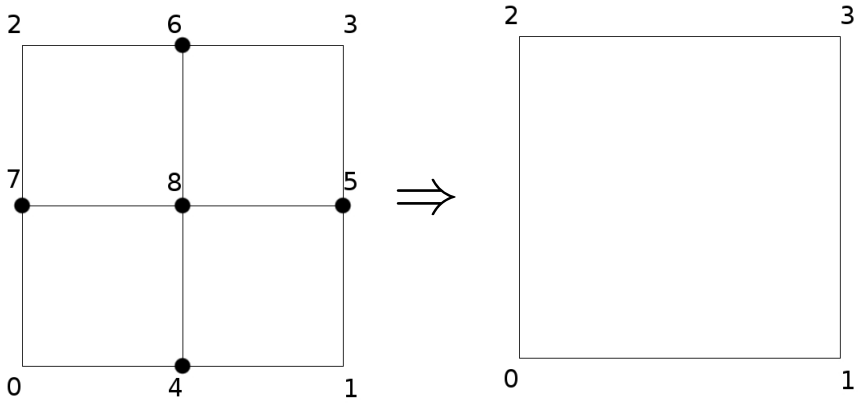



Abbildung 2.7: Die Restriktion im Detail.

### 2.3.2 Glätter

Durch die Restriktion wird ein möglicher Fehler verstärkt, denn die Vergröberung macht aus dem niederfrequenten einen hochfrequenten Fehleranteil. Niedrigfrequent bedeutet, dass der Fehler der Gitterpunkte nicht zu sehr von dem Fehler der benachbarten Gitterpunkte abweicht. Hochfrequent steht hingegen für eine hohe Fehlerabweichung zwischen Gitterpunkten. Um dem Prozess entgegen zu wirken wird ein Glättungssschritt eingeführt, der den Fehler aus dem hochfrequenten in den niederfrequenten Bereich überführt. Dabei kann ein Glätter nicht nur einmal, sondern auch mehrfach angewandt werden. Mehrfache Anwendungen des Glätters führen zu einem sehr niederfrequenten Fehleranteil (siehe Abbildung 2.8). Der Grund, weshalb der hochfrequente Fehleranteil in einen niederfrequenten Fehleranteil umgewandelt werden muss, ist der, dass man durch die Vergröberung starke Schwankungen des Fehlers nicht darstellen kann.
Des Weiteren bleibt zu beachten, dass trotz der Umwandlung des Fehlers dieser nicht zwingend verringert wird.

Beim Glätten können auch die benachbarten Gitterpunkte einen Einfluss haben. Dieser wird durch den Stempel (siehe Abschnitt 2.2.2) bestimmt.

Die Besonderheit am Mehrgitterverfahren liegt vor allem daran, dass die Konvergenzzeit (die Zeit, die im Schnitt zur Ausführung gebraucht wird) des iterativen Prozesses unabhängig von der Größe des Gitters ist. Für größer werdende Gleichungssysteme steigt sozusagen die Iterationsanzahl nicht.
Jedoch muss man auch hier bezüglich der Gesamtlaufzeit auf die verwendeten Glät-

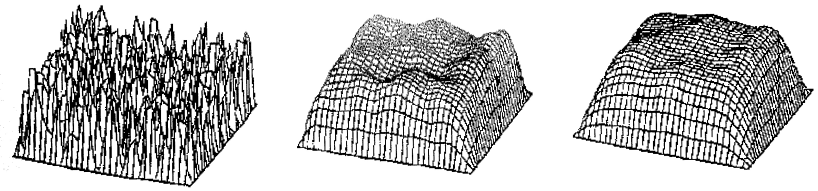
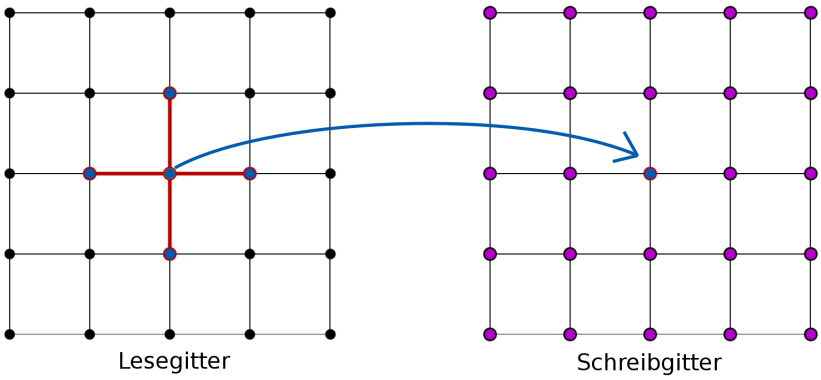


Abbildung 2.8: Der Fehler ohne Glätter (links), der Fehler mit 5 Iterationen des Gauss-Seidel-Glätters (mittig) und mit 10 Iterationen des Gauss-Seidel-Glätters (rechts) [TS01].

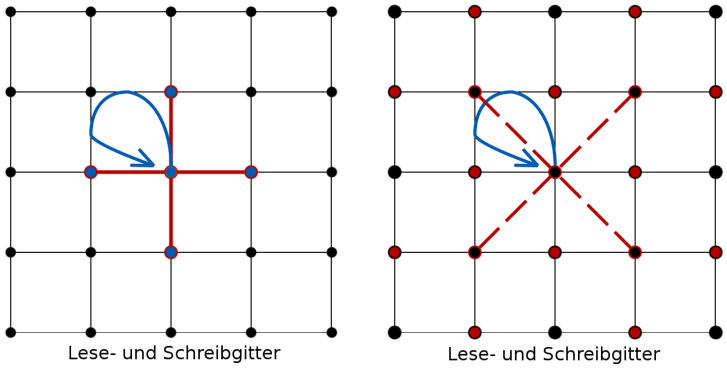
ter achten. Diese können je nach verwendeten Verfahren einen höheren oder niedrigeren Aufwand bedeuten. In der Regel braucht man bei Verfahren mit höherem Aufwand nicht so oft zu glätten, als bei Verfahren mit niedrigeren Aufwand.

Das parallele Ausführen von Glättern ist besonders im Hinblick auf die Rechenzeit von Bedeutung. Es gibt Glätter, welche voll parallel sind, also jeder Knoten auf einen Thread ausgelagert werden kann. Diese besitzen meistens nur eine schwache Glättungseigenschaft, weswegen diese öfter angewandt werden müssen.
Dass manche Glätter viel besser parallelisierbar sind als andere, liegt hauptsächlich an dem zugrundeliegenden Verfahren.
*Jacobi* ist beispielsweise ein voll parallelisierbarer Glätter, der allerdings nur schwach glättet. Das Ergebnis des Glättungsschrittes wird in einem anderen Gitter gespeichert. Dies wird für jeden Punkt gemacht, weswegen man dies leicht auf andere Threads auslagern kann. Dies wird unter anderem in Abbildung 2.9 veranschaulicht. *Jacobi* verwendet ein Lesegitter, auf dem die eigentlichen Werte gespeichert sind. Aus diesen Werten werden die Neuen berechnet und auf einem separaten Schreibgitter gespeichert.
Da für die Glätter die Stempel (siehe Abschnitt 2.2.2) verwendet werden, gibt es verschiedene Jacobi-Glätter, wie in etwa im dreidimensionalen Fall den 7-Punkt Jacobi(`jacobi7`) und den 27-Punkt Jacobi (`jacobi27`).

Der Glätter *Gauss-Seidel-LEX*(*Gauss-Seidel* Iteration mit lexikographischer Ordnung) verbessert die Glättungseigenschaft und muss deshalb nicht so oft ausgeführt werden wie etwa *Jacobi*. Allerdings können durch das Verfahren nur diagonale Gitterpunkte (maximal $\sqrt{N}$ wobei $N$ die Anzahl der Gitterpunkte) parallelisiert werden. Außerdem besitzt der Gauss-Seidel Glätter die Eigenschaft, dass man hierbei nur auf einem Gitter arbeitet. Das heißt, dass man ein Gitter zum Lesen und Schreiben der Werte benutzt. Diese neuen Werte werden noch von den benachbarten Punkten genutzt, sobald diese geglättet werden. Auch damit lässt sich die schlechte Parallelisierbarkeit erklären. Die Operationen auf dem Gitter sind unter anderem in Abbildung 2.9 veranschaulicht.
Im dreidimensionalen Fall gibt es hierfür den 7-Punkt Gauss-Seidel (`gauss`). Aufgrund der schlechten Parallelisierbarkeit entstand *Gauss-Seidel-RB* (*Gauss-Seidel* mit der rot-schwarzen (red-black) Ordnung wie in Abbildung 2.10). Dieser ist halb parallelisierbar, so dass sich $\frac{1}{2}N$ Knoten parallelisieren lassen, wobei $N$ die Anzahl




(a) Jacobi



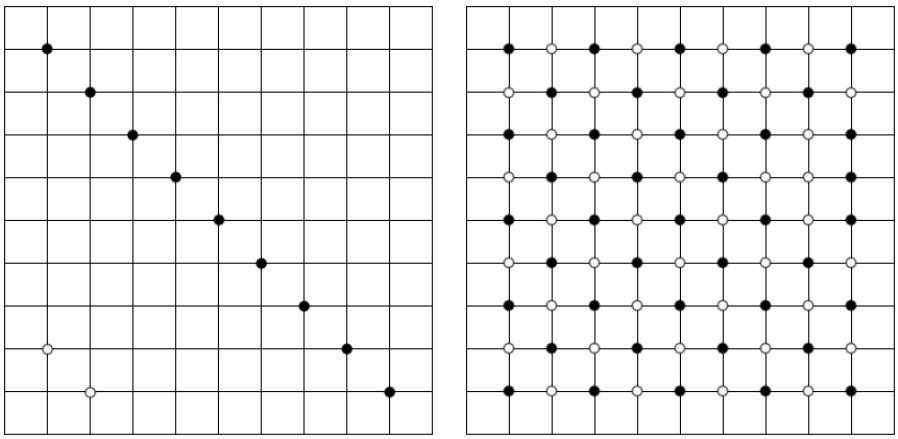

(b) Gauss-Seidel (c) Gauss-Seidel Red-Black

Abbildung 2.9: Darstellung verschiedener Glätter: (a) Jacobi-Glätter; (b) Gauss-Seidel-Glätter; (c) Gauss-Seidel Red-Black-Glätter. [Dat09]

an Gitterknoten ist. Bei diesem Glätter können in einem Schritt alle roten Punkte und im darauffolgenden Schritt alle schwarzen Punkte geglättet werden. Dabei werden nicht mehr die direkten Nachbarn genommen, wie es etwa bei *Jacobi* und *Gauss-Seidel* der Fall, sondern die Nachbarn, welche sich auf der Diagonale befinden und daher auch die gleiche Farbe haben wie der Ausgangspunkt. Auch dies wird in Abbildung 2.9 veranschaulicht.

Weitere Gauss-Seidel-Glätter lassen sich in mehr als zwei Farben darstellen lassen, wie etwa bei *parallel block multi-color Gauss-Seidel*. *Parallel block multi-color Gauss-Seidel* wurde von Mark Adams [Ada01] entwickelt. Der Glätter besitzt jedoch den Nachteil, dass der rechnerische Aufwand proportional zur Anzahl der benötigten Farben steigt [ABHT03].

Eine Analyse von `jacobi7`, `jacobi27` und `gauss` auf Mehrkern-Prozessoren wird in [RYQ11] durchgeführt. J. Holewinski et al. haben verschiedene Glätter und Verbesserungen dieser Glätter auf GPU-Architekturen analysiert [HPS12].



(a) Gauss-Seidel-LEX (b) Gauss-Seidel-RB

Abbildung 2.10: Darstellung der Parallelität von *Gauss-Seidel-LEX*(a) und von *Gauss-Seidel-RB*.

### 2.3.3 Vorkonditionierer

Durch die Vorkonditionierung werden die PDG für die Verarbeitung und für das Anwenden des Lösers vereinfacht, das heißt, dass dadurch der Aufwand erheblich vermindert wird indem das Gleichungssystem durch Umformung in ein äquivalentes System gebracht wird. Beispiele dafür sind:

- SSOR (Symmetric Successive Over-Relaxation algorithm)
- pointwise Jacobi
- elementwise block Jacobi

Die Vorkonditionierer können außerdem auch als Glätter benutzt werden, da auch diese eine Glättungseigenschaft haben und sich anhand der Glättungseigenschaft, der Laufzeit und Parallelität unterscheiden.

### 2.3.4 Prolongation

Die Prolongation ist das Gegenteil von der Restriktion(siehe Abschnitt 2.3.1). Sie ist als Abbildung von einem groben Gitter in ein feineres Gitter zu sehen. Das resultierende Problem hat wieder mehrere Variablen beziehungsweise Unbekannte als auf dem gröberen Gitter. Dies wird exemplarisch in Abbildung 2.11 dargestellt. Zuerst werden auf dem noch groben Gitter mit der linken unteren Ecke von links nach rechts und von unten nach oben die bisherigen Knoten nummeriert. Daraufhin müssen neue Punkte generiert werden, die später als Knoten genutzt werden. Diese werden gegen den Uhrzeigersinn von unten beginnend nummeriert. Nach diesem Schritt werden die neuen Zellen erzeugt. Der Vorteil der Nummerierung ist, dass sie von den nächsten Prolongations- und Restriktionsschritten übernommen und weiterhin verwendet werden kann [Alt13]. Ein Prolongationsschritt wird durch die Anwendung eines Prolongationsoperators durchgeführt. Der Prolongationsoperator wird im Folgenden auch Interpolationsoperator genannt.

Wie bei der Vergröberung besteht auch bei der Prolongation die Wahl zwischen mehreren sogenannten Interpolationsoperatoren, die sich anhand von Parallelität, Laufzeit und Effektivität unterscheiden.

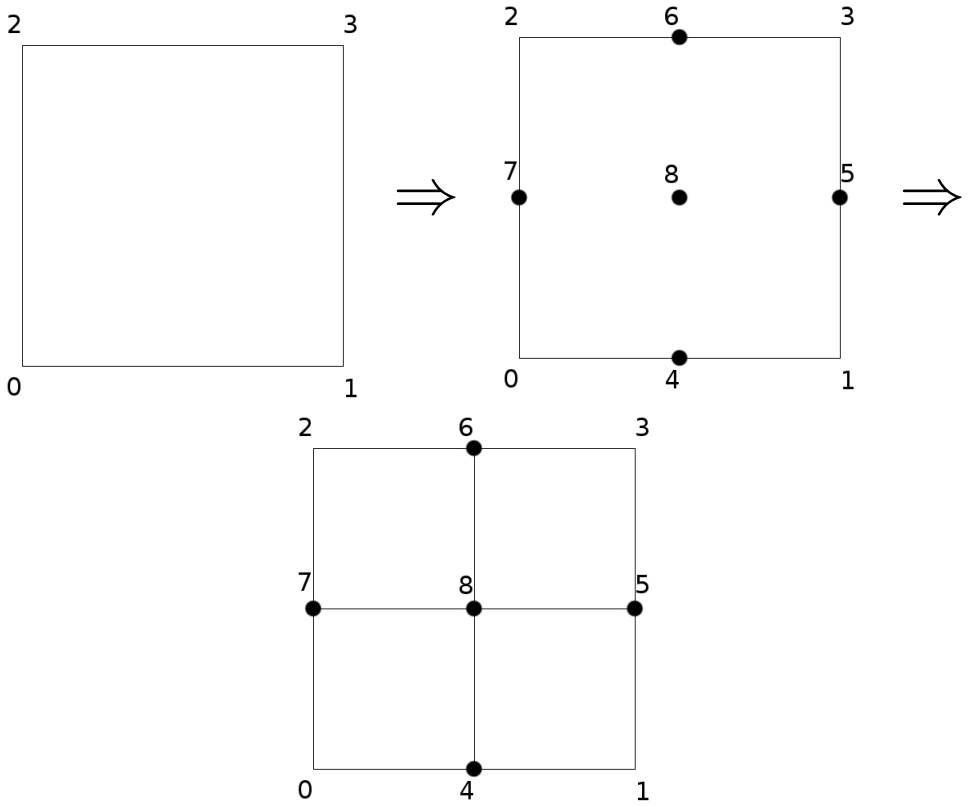



Abbildung 2.11: In dieser Darstellung sind die einzelnen Schritte der Prolongation ersichtlich.

### 2.3.5 Löser

Der Begriff „Löser“ hat in dieser Arbeit zwei Bedeutungen. Einmal steht es für das allgemeine Verfahren PDG zu lösen. Dafür gibt es verschiedene Herangehensweisen. Jedes einzelne davon hat seine eigenen Vor- und Nachteile. Diese sind bereits in Kapitel 1 zum Teil aufgeführt. In Tabelle 2.1 sind einige der Verfahren mit der Anzahl an Operationen im zweidimensionalen Raum aufgelistet.

Zum Anderen steht der Begriff Löser im Falle des iterativen Mehrgitterverfahrens für das Anwenden eines Lösers auf einen sehr groben Gitter. Oftmals wird es hier auch als „Mehrgitterlöser“ bezeichnet.

### 2.3.6 Zyklenarten

Für das Mehrgitterverfahren sind nicht nur die Glätter, die Vergröberungen und die Löser wichtig, sondern auch der angewandte Zyklus, der den ganzen Vorgang beschreibt.
Ein Zyklus fängt mit dem feinsten Gitter an, wird dann immer gröber, wobei die Glätter angewandt werden. Jedes dieser Gitter wird dabei einem Level zugeordnet,

| Verfahren | # Operationen in 2D |
|---|---|
| Gaußsche Elimination | $O(N^3)$ |
| Jacobi-Iteration | $O(N^2 \log \epsilon)$ |
| Gauss-Seidel Iteration | $O(N^2 \log \epsilon)$ |
| Sukzessives Überspannen (SOR) | $O(N^{3/2} \log \epsilon)$ |
| Conjugate gradient(CG) | $O(N^{3/2} \log \epsilon)$ |
| Geschachtelte Zerteilung | $O(N^{3/2})$ |
| ICCG | $O(N^{5/4} \log \epsilon)$ |
| ADI | $O(N \log N \log \epsilon)$ |
| Schnelle Fourier Transformation | $O(N \log N)$ |
| Buneman | $O(N \log N)$ |
| Vollständige Reduktion | $O(N)$ |
| Mehrgitter(iterativ) | $O(N \log \epsilon)$ |
| Mehrgitter(FMG) | $O(N)$ |

Tabelle 2.1: Komplexität verschiedener Löser des 2D Poisson Problems ($N$ steht für die Anzahl an Unbekannten). Die große Anzahl an Lösern ergibt sich durch die ständige Verbesserung des Verfahrens [TS01].

wobei sich das feinste Gitter im höchsten Level und das gröbste Gitter im niedrigsten Level (in der Regel gilt hier Level=0) befindet.
Auf dem gröbsten Level wird das Gitter gelöst. Daraufhin wird es verfeinert. Generell hat man jedoch die Wahl, ob man nach einer oder mehreren Prolongationen die Vergröberung wiederholt anwendet und das Gitter ein weiteres Mal löst, um den Fehler der Lösung zu verkleinern. Man unterscheidet dadurch zwischen einem V-Zyklus, einem W-Zyklus und einem F-Zyklus.
Außerdem kann man diese Zyklen auch beliebig schachteln. Dies wird oft für nichtlineare Probleme verwendet. Hierbei wird für jedes Gitterlevel ein V-Zyklus durchgeführt und hinterher hat man die Wahl zwischen V-, W- und F-Zyklen, die man beliebig verschachteln kann, um die Genauigkeit der Lösung zu beeinflussen.

### V-Zyklus

Der V-Zyklus ist der Einfachste der drei Zyklen. Dieser Zyklus fängt beim feinsten Gitter an, glättet dieses wahlweise vor (wird auch „Vorglättung“ genannt), setzt die Restriktion ein und wiederholt dies, bis es Level $k = 0$ erreicht hat. Dann wird das Gitter einmal gelöst. Sobald es gelöst wurde, kommt man durch Prolongationen wieder zurück zum feinsten Gitter, wobei auch hier die Möglichkeit besteht, das Gitter zu glätten (sogenanntes „Nachglättung“), um den Fehler weiterhin zu minimieren. Eine Veranschaulichung davon findet man in Abbildung 2.12, wobei $k$ das Gitterlevel ist. Die Überführung von $k = i$ zu $k = i + 1$ ist eine Restriktion, die Überführung von $k = i$ zu $k = i - 1$ eine Prolongation. Die ausgefüllten Punkte sind die jeweiligen Gitter, auf die je nach Einstellung ein Glätter angewandt wird. Der nicht ausgefüllte Punkt bei $k = 0$ stellt das Gitter dar, worauf der Löser angewendet wird.
Der Vorteil des V-Zyklus liegt darin, dass er leicht zu verstehen, zu implementieren und auch keinen großen Rechenaufwand beim Lösen erzeugt.
Der Nachteil ergibt sich ebenfalls aus der Einfachheit: Die Genauigkeit der Lösung

ist nicht so hoch. Dadurch wird auch eine erhöhte Zahl an Glättungsschritten gebraucht.

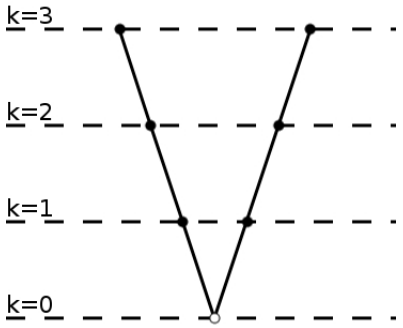



Abbildung 2.12: Der V-Zyklus.

### W-Zyklus

Der W-Zyklus hat genauso wie der V-Zyklus den Namen aus seiner Form, ist aber wesentlich komplexer als der V-Zyklus. Hierbei wird für jeden Durchlauf eines Levels der V-Zyklus zweimal ausgeführt. Dadurch wird das Gitter nicht einmal oder zweimal gelöst, sondern $2^{L-1}$ wobei $L$ das Level des feinsten Gitters bezeichnet.
Bezüglich des Post- und Pre-Smoothings gilt hier das Gleiche wie in Abschnitt 2.3.6. Der Vorteil liegt nun in der wiederholten Anwendung des Lösers und daher auch eine geringere Iterationsanzahl, wogegen der Nachteil in der Komplexität und dem deutlich erhöhten Rechenaufwand liegt. Eine visuelle Darstellung des W-Zyklus ist in Abbildung 2.13 zu sehen. Dabei steht $k$ für das Gitterlevel. Die Überführung von $k = i$ zu $k = i - 1$ ist eine Restriktion, die Überführung von $k = i$ zu $k = i + 1$ eine Prolongation. Die ausgefüllten Punkte sind die jeweiligen Gitter, auf die je nach Einstellung ein Glätter angewandt wird. Der nicht ausgefüllte Punkt bei $k = 0$ stellt das Gitter dar, worauf der Löser angewendet wird.
Außerdem kommt der W-Zyklus mit weniger Iterationen und Glättungsschritten aus, da das Gitter mehrfach gelöst wird und dadurch die Glattheitsanforderungen nicht so hoch sind wie beim V-Zyklus. Deshalb hat der W-Zyklus eine gute Robustheitseigenschaft.

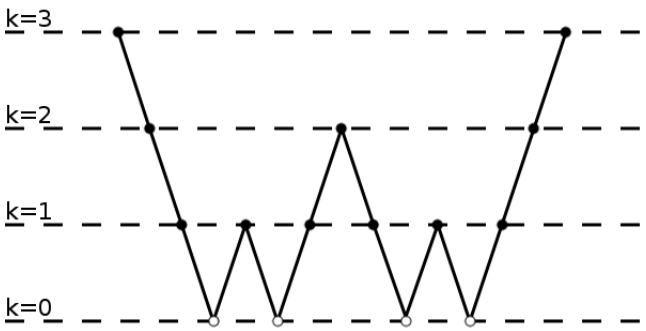



Abbildung 2.13: Der W-Zyklus.

Wenn man die Anzahl der V-Zyklen beziehungsweise der Lösungsschritte genauer betrachtet, so kommt man zu folgenden Werten:

| Level | Lösungsschritte |
|---|---|
| 2 | 1 |
| 3 | 2 |
| 4 | 4 |
| 5 | 8 |

Tatsächlich fällt einem dabei auf, dass die Anzahl der Lösungsschritte exponentiell wächst. Genauer lässt sich die Anzahl der Lösungsschritte wie folgt bestimmen:

$$Schritte = 2^{(level-2)}$$

wobei *level* die Anzahl der Levels ist.

Folgende Formel wird im Buch von Ulrich Trottenberg et al., [TS01] benutzt:

$$Schritte = 2^{(L-1)}$$

falls $L$ das Level des feinsten Gitters ist.

**F-Zyklus**

Der V-Zyklus ist ziemlich einfach, benötigt jedoch eine höhere Iterationsanzahl, als der W-Zyklus. Der W-Zyklus löst jedoch das Gitter sehr oft. Diese beiden Zyklen kann man allerdings kombinieren, um einen Zyklus zu erhalten, der weniger Iterationen benötigt, als der V-Zyklus und zudem noch annähernd so eine gute Lösung hat, wie der W-Zyklus, dafür jedoch das Gitter nicht so oft löst. Aus der Kombination der beiden Zyklen ergibt sich der sogenannte F-Zyklus. Wichtig ist bei diesem Zyklus, dass für jedes Gitterlevel einmal der V-Zyklus ausgeführt wird. In diesem Zyklus wird das Gitter daher nur $L$-mal gelöst, wobei $L$ das Level des feinsten Gitters ist. Dieser Zyklus erklärt sich am besten durch die Veranschaulichung in Abbildung 2.14, wobei $k$ für das Gitterlevel steht. Die Überführung von $k = i$ zu $k = i + 1$ ist eine Restriktion, die Überführung von $k = i$ zu $k = i - 1$ eine Prolongation. Die ausgefüllten Punkte sind die jeweiligen Gitter, auf die je nach Einstellung ein Glätter angewandt wird. Der nicht ausgefüllte Punkt bei $k = 0$ stellt das Gitter dar, worauf der Löser angewendet wird. In dieser Abbildung wurde bei $k = 4$ angefangen, um den F-Zyklus exemplarisch besser darstellen zu können.

Hierbei gestaltet sich die Anzahl der Lösungschritte wie folgt:

| Level | Lösungsschritte |
|---|---|
| 2 | 1 |
| 3 | 2 |
| 4 | 3 |
| 5 | 4 |

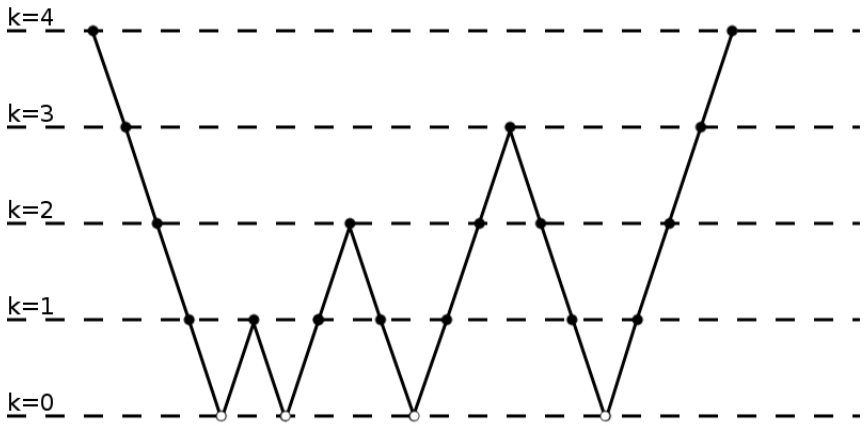



Abbildung 2.14: Der F-Zyklus.

Die Formel dafür lautet:

$$Schritte = level - 1$$

wobei das *level* die Anzahl der Levels ist.

Wie auch beim W-Zyklus, wird im Buch von Ulrich Trottenberg et al., [TS01] folgende Formel verwendet:

$$Schritte = L$$

wobei das $L$ das Level des feinsten Gitters ist.

## 2.4 Zusammenfassung

In 2.1 ist das Mehrgitterverfahren mit seinen Komponenten als Pseudocode unter der Verwendung des V-Zyklus zusammengefasst und wird hinterher Schritt für Schritt erklärt. Dadurch sollte noch einmal verdeutlicht werden, wann, wo und wie die einzelnen Komponenten beim Mehrgitterverfahren angewendet werden. Im Pseudocode wird exemplarisch die Methode `performVCycle()` für die Ausführung eines V-Zyklus verwendet. Diese kann jedoch auch durch andere Methoden ersetzt werden. Welche Änderungen dabei nötig sind, wird in Abschnitt 4.3 bei der Implementierung verschiedener Zyklen veranschaulicht.

Der erste Schritt (Zeile 1) beim Ausführen des Mehrgitterverfahrens ist das Initialisieren des Gitters(siehe Abschnitt 2.2.1). Dabei werden alle Gitterknoten mit Werten aus der Diskretisierung des PDG's oder mit zufällig bestimmten Werten belegt.
In Zeile 2 wird gezeigt, wie eine Iteration aussieht. Für jede Iteration wird dementsprechend einmal der jeweilige Zyklus (siehe Abschnitt 2.3.6) ausgeführt.

```
1  initGrid();
2  for iteration=0 to maxNumIt step iteration + 1
3    performVCycle();
4
5  function performVCycle() {
6    if(coarsest level) {
7      solve();
8    } else {
9      preSmoothing();
10     performRestriction();
11     performVCycle();
12     performProlongation();
13     postSmoothing();
14   }
15 }
```

Listing 2.1: Mehrgitterverfahren als Pseudocode

Alternativ kann man anstatt der Iterationsanzahl auch festlegen, wie genau die Lösung sein soll. In diesem Fall werden so viele Iterationen ausgeführt, bis der Fehler so klein ist, dass dieser die Bedingung erfüllt.
Von Zeile 5 bis 15 erstreckt sich der Zyklus und die jeweiligen Schritte dafür. Das Prinzip ist hier folgendes: Beim ersten Aufruf der Funktion hat man ein feines Gitter. Dann versucht man den Aufwand vom Finden der Lösung des Gitters zu minimieren, indem man das Gitter vergröbert. Ist man auf dem gröbsten Level angekommen(Zeilen 6 - 7) wird es gelöst. Dabei hat man die Auswahl, welchen Löser man verwendet. Darauf wird in Abschnitt 2.3.5 eingegangen.
Beim Vergröbern muss man immer auf den Fehleranteil achten. Deshalb wird es vor jeder Vergröberung in Zeile 9 geglättet (siehe Abschnitt 2.3.2). Dabei kann beliebig oft geglättet werden. Die Anzahl der Glättungsschritte, welche man durchführen muss damit es für die Vergröberung reicht hängt von dem verwendeten Glätter ab. Hierbei gibt es verschiedene, die sich von der Parallelität und der Glättungseigenschaft her unterscheiden. Nach dem Glätten folgt das Vergröbern (Zeile 10). Die Restriktion(oder auch Vergröberung) wird in Abschnitt 2.3.1 ausführlich erklärt. Bei der Restriktion besteht ebenfalls eine Auswahl zwischen mehreren Restriktionsoperatoren.
Nachdem es geglättet worden ist, folgt ein rekursiver Aufruf in Zeile 11 damit, falls nötig, noch weiter vergröbert und schließlich gelöst wird. Damit man am Ende auf das feinste Gitter kommt, muss daraufhin verfeinert beziehungsweise die Prolongation angewandt werden. Dies passiert in Zeile 12. In Abschnitt 2.3.4 wird darauf näher eingegangen. Damit auch bei der Prolongation der Fehler im niedrigfrequenten Bereich bleibt, muss nach der Prolongation noch einmal nachgeglättet(siehe Zeile 13) werden.
Auch hier besteht die Auswahl zwischen verschiedenen Prolongationsalgorithmen.

# 3. Visualisierung konfigurierbarer Systeme

In diesem Kapitel werden weitere Grundbegriffe behandelt, welche nötig sind, um die visuelle Darstellung der Programme und ihrer Mächtigkeit zu verstehen. Zuerst wird die Bedeutung des Begriffs Feature erklärt. Daraufhin wird beschrieben, was ein Feature Modell ist und welche Eigenschaften es hat. Schlussendlich werden Feature Diagramme beschrieben beziehungsweise die visuelle Darstellung der Programme, die im Rahmen dieser Arbeit analysiert und erweitert werden.

## 3.1 Feature

Ein Feature eines Systems beziehungsweise eines Programms ist eine Komponente des Systems, welches eine bestimmte Funktion oder eine Reihe von Funktionen erfüllt. Bei Features hat man oftmals die Wahl, ob man es aktiviert oder deaktiviert haben möchte. Außerdem kann ein Feature weitere Features bereitstellen, andere Features ausschließen oder darauf basieren. Diese Eigenschaften werden auch Beziehungen zu anderen Features genannt. Wenn ein Feature (mit A in Abbildung 3.1 bezeichnet) ein anderes Feature (mit C in Abbildung 3.1 bezeichnet) besitzt, ist es das Eltern Feature des anderen Features. Umgekehrt ist das andere Feature (mit C in Abbildung 3.1 bezeichnet) das Kind Feature.
Im Rahmen dieser Arbeit ist beispielsweise jeder einzelne Löser ein Feature.
Features können zudem auf zwei verschiedene Arten bezüglich ihrer Variabilität unterteilt werden, wie:

- *boolsche Features*: Die boolschen Features haben nur zwei Zustände: an oder aus.
- *numerische Features*: Diese Art von Features können im Gegensatz zu boolschen Features mehr als nur zwei Zustände haben. Normalerweise gibt man bei den numerischen Features einen Wert, etwa eine Zahl, ein. Da es bei der Wahl des Wertes mehrere Möglichkeiten gibt, sofern diese nicht eingegrenzt ist, gibt es für das Feature dementsprechend mehrere Zustände.

Die zuvor erwähnten Beziehungen sind im sogenannten Feature Modell beinhaltet, worauf im Folgenden näher eingegangen wird.

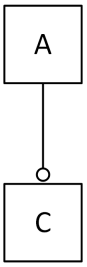



Abbildung 3.1: Die Feature Beziehung zwischen einem übergeordneten und einem untergeordneten Feature als visuelle Darstellung (siehe auch Abschnitt 3.2).

## 3.2 Feature Modell und Feature Diagramm

### Feature Modell

Das Feature Modell[1] ist als eine Art von Beschreibung eines Systems zu verstehen. Ein Programm kann verschiedene Features haben. Zwischen manchen oder sogar allen von diesen kann eine Beziehung bestehen. Außerdem kann ein Feature Modell eine detaillierte Beschreibung der Features enthalten (siehe [KCH+90]). Im Folgenden wird auf die visuelle Darstellung des Feature Modells näher eingegangen.

### Feature Diagramm

Ein Feature Diagramm ist die grafische Darstellung der Variabilität eines Systems mit ihren zugrundelegenden Komponenten beziehungsweise Features. Das Feature Diagramm ist eine baumartige Struktur, wobei der oberste Knoten die Wurzel ist. Ein Beispiel für ein Feature Diagramm ist in Abbildung 3.2 zu sehen. Der oberste Knoten in der Abbildung (mit A bezeichnet) ist dabei die Wurzel. Von A aus gibt es verschiedene Features, die einen hohlen oder einen ausgefüllten Kreis darauf haben. Der hohle Kreis bedeutet, dass das Feature optional ist. Features mit einem ausgefüllten Kreis hingegen müssen ausgewählt werden, sie sind also obligatorisch. Sollte beispielsweise das Feature C ausgewählt werden, so muss auch das Feature H genommen werden. In den Features B und D gibt es eine Variabilität. Bei B muss man genau ein Kind-Feature von B auswählen, wie E, F oder G. Dies entspricht einer Alternativ-Gruppe. Bei D hingegen können ein oder mehrere Kind-Features ausgewählt werden, was als Oder-Gruppe bezeichnet wird.
Bei numerischen Optionen ist es oft so, dass die jeweiligen Typen angegeben werden, wie etwa **u.int**, **int**, **double** oder **vector<>**. Bei **u.int** handelt es sich um **unsigned integer**, eine Integer-Zahl ohne Vorzeichenbit. Damit können sich die Werte im Bereich $0 \leq x \leq$ `<Größe von unsigned int>` befinden. **int** steht für Integer. Der Wertebereich kann sowohl positiv, als auch negativ sein. Festkommazahlen sind vom Typ **double**. Falls ein Eingabeparameter mehrere Werte verlangt, so ist es vom

[1] Oft wird mit dem Begriff Feature Modell auch das Feature Diagramm gemeint, da das Feature Diagramm die visuelle Darstellung des Feature Modells ist.

Typ **vector<>**. Um den Wertebereich noch zusätzlich einzuschränken, wird dieser meistens auch in den Feature Diagrammen mitgeliefert. Die Darstellung davon ist unterschiedlich. In dieser Arbeit wird ein Wertebereich durch $[x, ..., y]$ dargestellt, das so viel bedeutet wie „die eingegebene Zahl muss größer oder gleich als $x$ und kleiner oder gleich $y$ sein“. Eine andere Darstellung wäre $[x, ...]$. Dies bedeutet „die eingegebene Zahl muss größer oder gleich $x$ sein und darf maximal so groß wie die letzte darstellbare Zahl des jeweiligen Datentyps sein“.
In einem Feature Diagramm können auch noch weitere Eigenschaften dargestellt werden, wie in etwa Abhängigkeiten und Einschränkungen. Eine Abhängigkeit existiert in etwa zwischen H und B, als auch zwischen F und J. $\rightarrow$ steht dabei für eine Folgerung. Wenn nun H ausgewählt wird, muss auch B ausgewählt werden, jedoch nicht umgekehrt. Genau so verhält es sich auch zwischen F und J. Einschränkungen sind das Gegenteil von Abhängigkeiten. Damit schließt man die gleichzeitige Auswahl mehrerer Feature oder Werte von Features aus. In der Grafik sind beispielsweise zwei Einschränkungen bezüglich $F$, $J$ und $I$ vorhanden. Die Einschränkungen werden durch ein $\wedge$-Symbol getrennt. $\wedge$ steht dabei für „und “ und mit $\neg$ wird die Negation symbolisiert. $\neg(F \wedge J = 1) \wedge \neg(F \wedge I = 0)$ bedeutet etwa: Wenn $F$ ausgewählt ist, darf $J$ nicht der Wert 1 und $I$ nicht der Wert 0 zugewiesen werden.

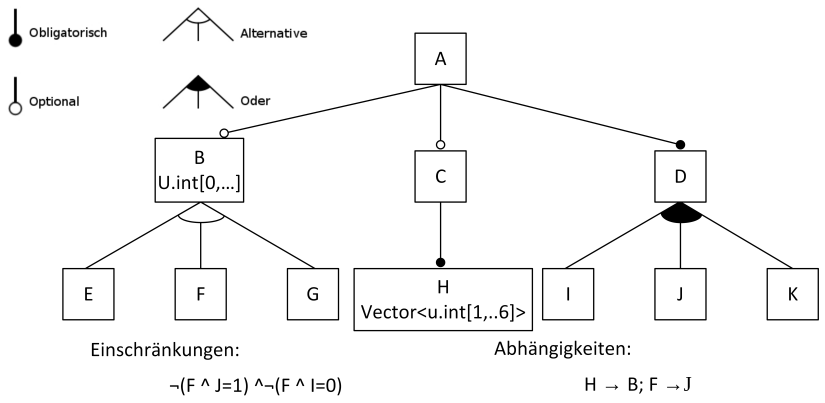



Abbildung 3.2: Ein Beispiel für ein Feature Modell.

Bei einem Feature Diagramm gibt es verschiedene Arten von Beziehungen zwischen Features. In dieser Arbeit werden folgende Beziehungen verwendet (siehe Abbildung 3.2 für deren Darstellung):

- *Obligatorisch*: Obligatorische Features beschreiben Features, die zwingend sind. Das bedeutet, sobald das Eltern Feature ausgewählt ist, muss auch das obligatorische Kind Feature ausgewählt werden.

- *Optional*: Diese Feature können entweder an oder abgeschaltet werden, wenn ihr Eltern Feature ausgewählt ist.

- *Alternative-Gruppe*: Alternative-Gruppen sind Gruppen von Features von denen genau eines ausgewählt werden muss, sobald das Eltern Feature der Gruppe gewählt ist.

- *Oder-Gruppe*: Wie bei *Alternative* kann nur eine Gruppe mit mindestens zwei Features mit der *Oder*-Verzweigung versehen werden. Dabei hat man die Wahl, ob man eines oder mehrere Features auswählt.

# 4. Programme für strukturierten Mehrgitterverfahren

In diesem Kapitel geht es um Frameworks und Programme sowie deren Erweiterungen, um die Mächtigkeit zu erhöhen. In Abschnitt 4.1 werden die Frameworks Hypre und DUNE mitsamt der Bibliotheken beschrieben, welche diese anbieten. Darauf aufbauend werden in Abschnitt 4.2 die Programme HSMGP und MGS vorgestellt, welche in Abschnitt 4.3 und in Abschnitt 4.4 bezüglich ihrer Mächtigkeit und Variabilität erweitert werden.

## 4.1 Frameworks

Im Folgenden werden zwei Frameworks vorgestellt, die Algorithmen zur Lösung von Mehrgittern anbieten. Da diese jedoch eine Bibliothek bilden, liegt es am Entwickler die entsprechenden Algorithmen so zu verknüpfen, um ein Mehrgitter möglichst effizient zu lösen.

### 4.1.1 Hypre

Hypre ist ein Framework, welches Algorithmen für Glätter und Löser bereitstellt. Ein wichtiger Aspekt ist dabei die Parallelität. Die Algorithmen in diesem Framework wurden so ausgelegt, dass sie auch auf System mit vielen Kernen (wie etwa bei Supercomputern) effizient ausgeführt werden können. Diese Algorithmen werden verschiedenen Kategorien zugeordnet, die sich voneinander durch die Form des Gitters unterscheiden, wie etwa:

- **Strukturierte Gitter System Schnittstelle (Struct)**: Diese Schnittstelle ist für Gitter geeignet, die rechteckig angeordnet sind und einen festen Stempel haben. Außerdem wird hierbei nur eine Unbekannte je Gitterpunkt unterstützt.
- **Semi-Strukturierte Gitter System Schnittstelle (SStruct)**: **SStruct** ist für die Gitter gedacht, die zum Großteil strukturiert, an manchen Stellen jedoch unstrukturiert sind. Genauer sind Gitter gemeint, deren Inhalt blockartig

und rechteckig aufgebaut ist, die jedoch in ihrer Form nicht mehr rechteckig sind. Ein sternförmiges Gitter wäre ein zulässiges Beispiel für diese Schnittstelle. Durch diese Schnittstelle werden auch mehrere Unbekannte je Punkt unterstützt.

- **Schnittstelle der finiten Elemente (FEI)**: Diese Schnittstelle ist für Gitter gedacht, deren Inhalt zudem auch unstrukturiert ist. Hier ist der Inhalt der Gitter nicht mehr rechteckig, sondern kann beliebige Formen haben.
- **Linear-Algebraische System Schnittstelle (IJ)**: **IJ** ist für Nutzer, welche ihr Problem nicht in die Kategorien **Struct**, **Sstruct** oder **FEI** passend zuordnen können. Bei der Berechnung ist mehr Aufwand nötig, als bei den anderen Schnittstellen. Der Nachteil an dieser Schnittstelle ist, dass nur allgemeine Algorithmen vorhanden sind und keine, die auf das Problem zugeschnitten sind, welche mehr Informationen brauchen.

Die Löser, die von den jeweiligen Schnittstellen bereitgestellt werden sind in Tabelle 4.1 ersichtlich.

| Löser | Struct | SStruct | FEI | IJ |
|---|---|---|---|---|
| Jacobi | X | X | | |
| SMG | X | X | | |
| PGMG | X | X | | |
| Split | | X | | |
| SysPFMG | | X | | |
| FAC | | X | | |
| Maxwell | | X | | |
| BoomerAMG | | X | X | X |
| AMS | | X | X | X |
| ADS | | X | X | X |
| MLI | | X | X | X |
| ParaSails | | X | X | X |
| Euclid | | X | X | X |
| PILUT | | X | X | X |
| PCG | X | X | X | X |
| GMRES | X | X | X | X |
| FlexGMRES | X | X | X | X |
| LGMRES | X | X | | X |
| BiCGSTAB | X | X | X | X |
| Hybrid | X | X | X | X |
| LOBPCG | X | X | | X |

Tabelle 4.1: Derzeitige Löser, welche durch die Hypre-Schnittstellen bereitgestellt werden [hyp01].

### 4.1.2 Dune

DUNE ist ein modulbasiertes Framework zum Lösen von PDG. Dieses Framework besteht aus mehreren Modulen, die sich in Kernmodule, Diskretisierungsmodule, Module für spezielle Gitter und externe Module unterteilen lassen. Dieser Aufbau wird in Abbildung 4.1 dargestellt.
Die Kernmodule sind folgende [BBD+11]:

- *dune-common*: Das Kernmodul *dune-common* enthält die Basisklassen, welche von allen DUNE-Modulen verwendet werden. Zudem werden Klassen zur Verfügung gestellt, welche für das Finden von Fehlern hilfreich sind. In diesem Modul befindet sich ebenfalls noch eine Bibliothek für dicht besetzte Matrizen und Vektoren.
- *dune-geometry*: Dieses Modul beinhaltet DUNE Referenzelemente, welche von den anderen Modulen gebraucht werden.
- *dune-grid*: Bei *dune-grid* handelt es sich um ein Kernmodul, welches unter anderem verschiedene parallele Gitter in beliebigen Größendimensionen und grafische Ausgabe ermöglicht (siehe hierzu [BBD+08b] und [BBD+08a]).
- *dune-istl*: *dune-istl* ist für den iterativen Löser von besonderer Bedeutung, da es Schnittstellen für Komponenten, wie etwa Löser und Vorkonditionierer zur Verfügung stellt (siehe hierzu auch [BB07] und [BB08]).
- *dune-localfunctions*: Dieses Modul stellt verschiedene Funktionen für unterschiedliche Formen von Referenzelementen aus *dune-geometry* zur Verfügung.

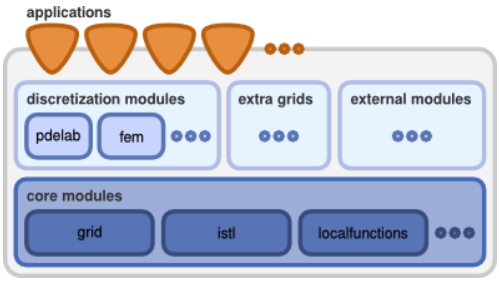



Abbildung 4.1: Design des Frameworks DUNE [BBD+11].

Das im späteren Verlauf vorgestellte Programm MGS hat als Abhängigkeit die zuvor erwähnten Kernmodule sowie zusätzlich *dune-pdelab*.
*dune-pdelab* ist ein allgemeines Diskretisierungsmodul mit einer großen Variation an Diskretisierungsmethoden, welche hauptsächlich von MGS verwendet wird, um das Gitter zu erstellen und die PDG zu lösen.

Außer diesen Modulen bietet DUNE noch andere Module an, welche jedoch in dieser Arbeit nicht benötigt und daher nicht aufgeführt werden.

## 4.2 Mehrgitterlöser

In diesem Abschnitt werden die Programme HSMGP und MGS näher beschrieben, welche jeweils verschiedene Komponenten eines Frameworks verwenden. Diese Programme werden im Verlauf dieser Arbeit noch erweitert und anschließend getestet.

### 4.2.1 HSMGP

**HSMGP** steht für *Highly Scalable Multigrid Prototype* und ist aus dem Vorhaben entstanden, Mehrgitter auch auf größeren Rechnern mit mehr als 450.000 Kernen ausführen zu können. Das Programm wurde unter anderem auf dem derzeit schnellsten Supercomputer Deutschlands, Blue Gene/Q[1] (JUQUEEN) ausgeführt. Dieser enthält 458.752 Kerne. Im Vergleich zu normalen Systemen, wie Laptops oder PC's, welche oftmals 4 Kerne haben, ist dies ein großer Unterschied und so ist es auch nicht weiter überraschend, wenn auch die Algorithmen für solche Parallelität teilweise angepasst werden müssen. Aus diesem Grund hat man sich für Algorithmen aus dem Framework Hypre entschieden (siehe Abschnitt 4.1.1). Genauer benutzt man hierbei den Löser **BoomerAMG**, um diesen mit den eigens implementierten **Conjugate Gradient (CG)**-Löser zu vergleichen. Es stellte sich tatsächlich heraus, dass **BoomerAMG** ab etwa 8.000 Kernen von der Laufzeit her schneller ist, als **CG** [KGKR13].

Das Feature Model von HSMGP wird in Abbildung 4.2 und Abbildung 4.3 dargstellt. Unter den Programmoptionen gibt es noch numerische Features (siehe Abschnitt 3.1).
Lediglich bei der Programmoption **omega** wurde der Wertebereich weggelassen. Dabei ist allerdings zu beachten, dass der Wert, welcher **omega** zugewiesen wird, größer 0 sein muss.

In HSMGP hat man zudem die Auswahl zwischen den Uhren, welche zur Zeitmessung verwendet werden, die Löser und die Glätter. Werden mehrere Glätter und/oder Löser verwendet, so verfälscht dies das Ergebnis. Deshalb ist es wichtig, bei den Definitionen nur jeweils einen Glätter und einen Löser anzugeben.

[1] http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html, letzter Zugriff: 11.08.2014

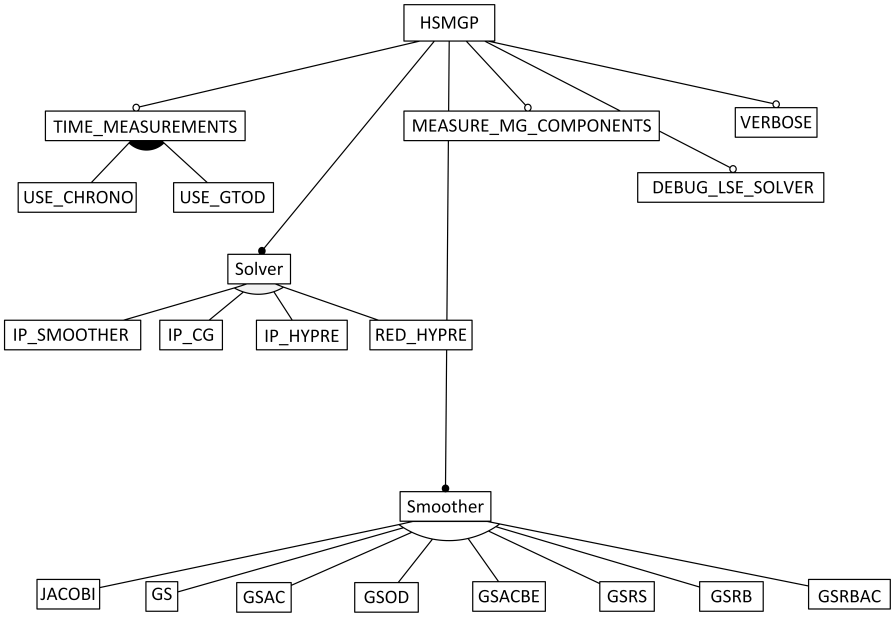



Abbildung 4.2: Das Feature Modell für die Definitionen von HSMGP

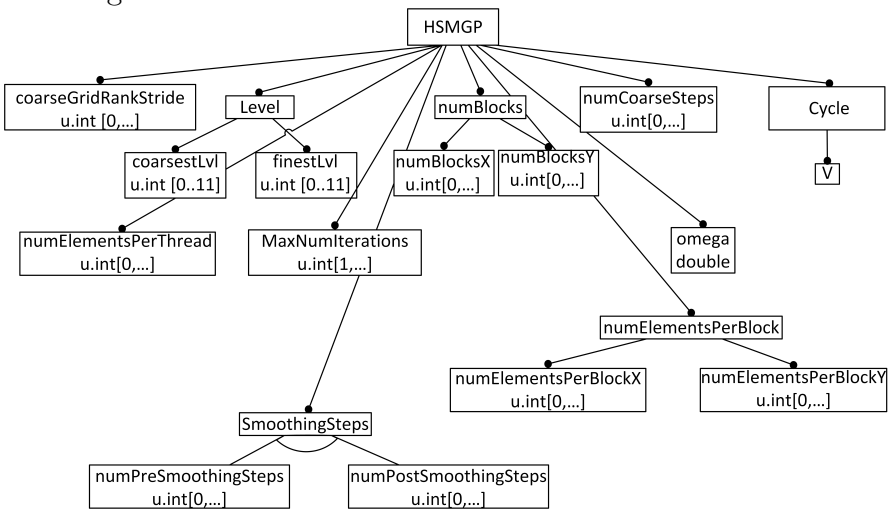



Abbildung 4.3: Das Feature Modell für die Programmoptionen von HSMGP

### 4.2.2 MGS

**MGS** ist die Abkürzung von *Multigrid Solver* und ist ein externes DUNE-Modul, welches dazu verwendet werden kann, Gitter von beliebiger Größe zu lösen. Dafür werden die in Abschnitt 4.1.2 erwähnten Module verwendet. Dieses Programm wurde implementiert, um die Mächtigkeit von DUNE zu demonstrieren. Während in HSMGP alles bis auf den Löser und den Vorkonditionierer implementiert worden ist, verwendet MGS hauptsächlich die Komponenten von DUNE, so dass hierdurch weniger Code erstellt werden musste. Dies hat zwar den Vorteil, dass aufgrund der Übersichtlichkeit sehr schnell Änderungen gemacht werden können, man jedoch bezüglich der Variabilität auf die Variabilität des Frameworks angewiesen ist. Möchte man zum Beispiel die Ausgabe modifizieren, so muss man dafür Code vom Framework umschreiben. Die Variabilität in MGS wird in Abbildung 4.4 als Feature Modell

dargestellt. Auch in diesem Feature Modell gibt es numerische Features. Diese sind vom Typ **Int**, was für **Integer** steht. Bei näherer Betrachtung des Feature Modells von MGS ist zu erkennen, dass die Variabilität im Vergleich zu HSMGP wesentlich geringer ist. Der Grund dafür ist das Fokussieren der Entwickler auf die Verwendung von DUNE-Komponenten und nicht auf die Variabilität des Programms.

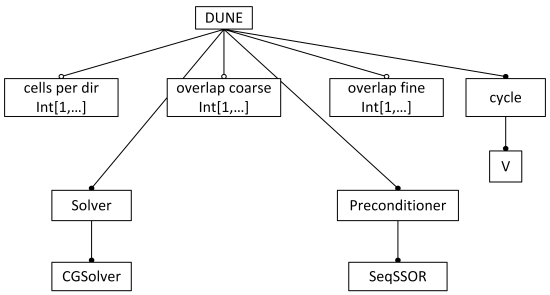



Abbildung 4.4: Das Feature Modell zu MGS.

## 4.3 HSMGP-Erweiterung

Das Programm HSMGP wurde mit einigen Erweiterungen versehen, um mehrere Eigenschaften des Mehrgitterverfahrens ausnutzen, Komponenten, wie etwa den Glätter und den Löser, austauschen zu können und die Mächtigkeit des Programms dadurch zu erhöhen. Das Feature Modell mit den Erweiterungen wird in Abbildung 4.5 und in Abbildung 4.6 dargestellt.

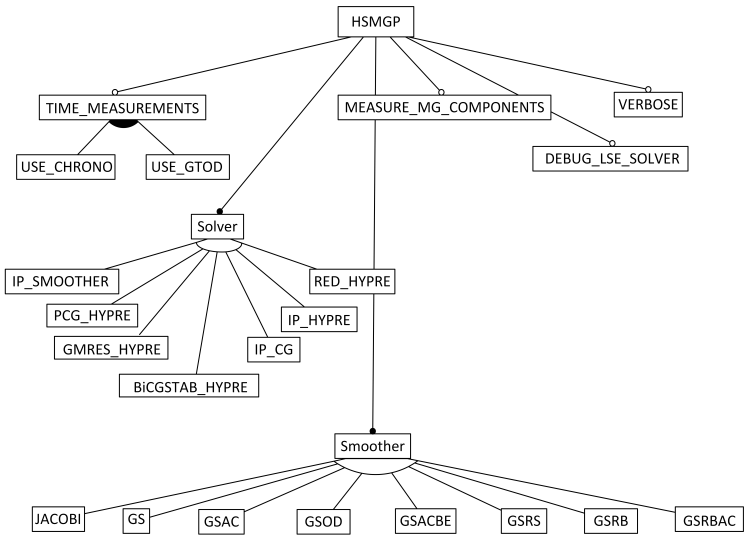



Abbildung 4.5: Das Feature Modell für die Definitionen von HSMGP mitsamt Erweiterungen

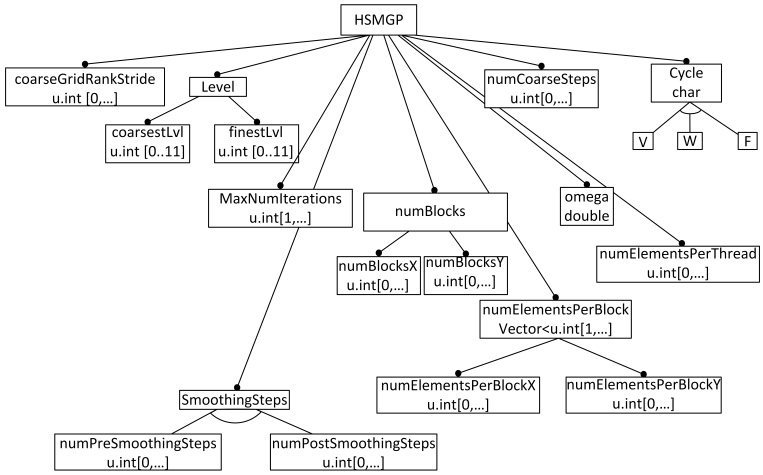



Abbildung 4.6: Das Feature Modell für die Programmoptionen von HSMGP mitsamt Erweiterungen

### 4.3.1 Zusätzliche Zyklenarten

Bisher wurde nur der V-Zyklus angeboten. Nun wurde dies noch um den W- und F-Zyklus erweitert (siehe auch Abschnitt 2.3.6). Damit hat der Nutzer die Auswahl zwischen verschiedenen Zyklen und kann so besser bestimmen, welche Konfigurationsoptionen am passendsten für die zu lösende Aufgabe sind. Durch die Verwendung des V-Zyklus braucht man mehrere Iterationen, um den Fehler so klein wie möglich zu bekommen, während man mit dem W-Zyklus nur wenige braucht, der W-Zyklus bezüglich der Laufzeit jedoch schlechter ist. Im Folgenden wird die Implementierung der Zyklenarten erläutert, um auch selber weitere Zyklen implementieren und testen zu können.

Die neu hinzugefügten Zyklen wurden um ein Array (`hasLevel`) erweitert, welches dabei hilft, festzustellen, ob in einem Level noch ein V-Zyklus gemacht werden muss oder nicht. Dadurch wurde der Einfluss bei der Berechnung so gering wie möglich gehalten. Dies sieht für den W-Zyklus im Grunde so aus:

```
1 if  (!hasLevel[level - 1]) {
2     for  (int i = 0; i < level - 1; i++) {
3       hasLevel[i] = false;
4     }
5     hasLevel[level - 1] = true;
6     performWCycle ();
7   }
```

Listing 4.1: W-Zyklus

Der W-Zyklus wird in Abschnitt 2.3.6 näher erklärt. Bei der Betrachtung von Abbildung 2.13 fällt einem auf, dass beim W-Zyklus für jedes einzelne Level der V-Zyklus doppelt ausgeführt wird. Wenn man davon ausgeht, dass $k = 3$ das feinste Gitter ist, so müssen 2 V-Zyklen für die darunterliegenden Levels ausgeführt werden, also für $k = 2$ und $k = 1$. Da $k = 0$ das gröbste Level ist, wird in diesem Level kein Zyklus mehr ausgeführt, sondern gelöst. Bei $k = 1$ muss der V-Zyklus dabei insgesamt viermal ausgeführt werden, je zweimal beim Durchlaufen des Levels.

Meine Überlegung war folgende: Man muss sich die verschiedenen Levels, welche inzwischen durchlaufen wurden, abspeichern. Zum Abspeichern wird ein Vektor (`hasLevel`) benutzt, welcher anfangs mit dem Wert `false` initialisiert wird. Immer, wenn ein höheres Level erreicht wird, werden die unteren Level zurückgesetzt, damit diese beim nächsten Aufruf den V-Zyklus erneut zweimal ausführen. Für das Beispiel in Abbildung 2.13 wird zuerst zweimal der V-Zyklus für das Level $k = 1$ durchgeführt. Sobald jedoch der V-Zyklus für $k = 2$ noch einmal ausgeführt wird, wird auch der V-Zyklus für das Level $k = 1$ erneut ausgeführt. So funktioniert auch der Codeabschnitt aus Listing 4.1. In Zeile 1 wird geprüft, ob für das derzeitige Level der V-Zyklus schon ausgeführt worden ist. Falls das der Fall ist, so wird in dem Fall auf einem höheren Level geschaut, ob man dafür einen zusätzlichen V-Zyklus ausführen kann. In diesem Vektor werden nur die inneren Level betrachtet. Dies entspräche bei 4 Levels die Levels 1 und 2 (auf Level 0 löst man und Level 3 ist das feinste Gitter). Da ein Vektor bei 0 beginnt, entspricht die Stelle 0 im Vektor **hasLevel** dem Level 1. Deswegen wird auch immer auf **level - 1** geprüft.
Wie schon erwähnt, wird zunächst der Vektor von den vorhergehenden Levels auf **false** gesetzt. Der Wert **false** steht dabei für „muss noch durchlaufen werden “ und **true** für „bereits durchlaufen“. Der Wert des eigenen Levels wird daraufhin auf **true** gesetzt, da dieser ja durchlaufen wurde (Zeile 5).
Letztendlich erfolgt ein rekursiver Aufruf für einen weiteren V-Zyklus (Zeile 6).

Der F-Zyklus ist etwas einfacher und die größte Änderung, welche nötig war, wird in Listing 4.2 dargestellt.

```
1 if  (!hasLevel[level - 1]) {
2     hasLevel[level - 1] = true;
3     performFCycle ();
4     }
```

Listing 4.2: F-Zyklus

Dieser gestaltet sich im Grunde wie der W-Zyklus. Allerdings gibt es dabei den Unterschied, dass die unteren Levels nicht mehr auf **false** gesetzt werden. Das bedeutet, dass für jedes Level genau ein V-Zyklus durchgeführt wird.

### 4.3.2 Zähler für Anwendung von Löser

Mit der Wahl der Zyklen kam auch noch ein Zähler dazu, der das Anwenden des Lösers in den jeweiligen Zyklen misst. Beim V-Zyklus sieht es beispielsweise so aus:

```
Number of solving steps: 1
```

Dies besagt, dass beim V-Zyklus nur einmal das Gitter gelöst worden ist. Während dies bei dem V-Zyklus immer nur bei 1 bleibt, verändert sich dies bei dem W- und F-Zyklus. Wie sich dies genau verhält, wird in Abschnitt 4.3.1 beschrieben.

### 4.3.3 Auswahl zusätzlicher Löser

Wie in Tabelle 4.1 zu sehen ist, bietet Hypre weitaus mehr Algorithmen an, als implementiert sind. Genutzt wurde bisher nur **BoomerAMG**.
Nun können noch weitere Löser ausgewählt werden, da auch diese verschiedene Laufzeiten und verschiedene Genauigkeiten besitzen und daher in gewissen Fällen besser oder schlechter sind als **BoomerAMG**. Wie bei den Glättern, spielt auch bei den Lösern die Parallelität eine wichtige Rolle.
Die Löser sind in Tabelle 4.2 mitsamt ihrer Präprozessordefinition aufgelistet.

| Löser | Definition |
|---|---|
| PCG | COARSE_GRID_SOLVER_PCG_HYPRE |
| GMRES | COARSE_GRID_SOLVER_GMRES_HYPRE |
| BiCGSTAB | COARSE_GRID_SOLVER_BiCGSTAB_HYPRE |

Tabelle 4.2: Die in HSMGP zusätzlich implementierten Löser.

In Kapitel 5 werden diese Löser näher betrachtet und untereinander verglichen.

### 4.3.4 Automatische Aufteilung des Gitters für Mehrprozessorsysteme

Bisher war es so, dass man die Anzahl der Blöcke auf der X- beziehungsweise Y-Achse immer von Hand eingeben musste. Nun werden diese automatisch bestimmt, falls sie nicht angegeben werden. Dabei wird möglichst eine quadratische Fläche genommen, falls die Anzahl an Prozessen eine Quadratzahl ist. Sollte dies nicht der Fall sein, so ist die Größe auf der Y-Achse fest 1 und auf der X-Achse die Anzahl an Prozessen.
Sollte allerdings die Größe für eine Achse angegeben sein und diese die Anzahl an Prozessen teilen, so wird die Größe der anderen Achse durch das Ergebnis der Teilung bestimmt. Beispielsweise wird bei einer Anzahl von Prozessen 8 und einer Größe auf der Y-Achse von 4 `numBlocksX` der Wert 2 zugewiesen.

### 4.3.5 Einfachere Eingabe der Anzahl von Elementen je Block

Um die Eingabe noch weiter zu verbessern, wurde die Option `numElementsPerBlock` hinzugefügt. Diese ist als Alternative zu den beiden Programmoptionen `numElementsPerBlockX` und `numElementsPerBlockY` zu sehen. Man kann die Anzahl an Elementen eines Blocks entweder so eingeben:
`--numElementsPerBlockX 4 --numElementsPerBlockY 4` oder
`--numElementsPerBlock 4 4` oder auch `--numElementsPerBlock 4` bei quadratischen Blöcken.

## 4.4 MGS-Erweiterung

Wie schon in Abschnitt 4.2.2 erwähnt, bietet MGS bezüglich der Variabilität nicht sehr viel Spielraum. Deshalb wurde dieses Programm um einige Eigenschaften des Mehrgitterverfahrens erweitert, wie etwa das Festlegen der Anzahl der Vor- und

Nachglättungsschritten. Auch die Zyklen V, W und F sind nun wählbar. Zudem gibt es auch eine größere Variabilität beim Löser und Vorkonditionierer. Dies wird in Abbildung 4.7 als Feature Modell dargestellt. Im Folgenden wird auf diese Erweiterungen näher eingegangen.

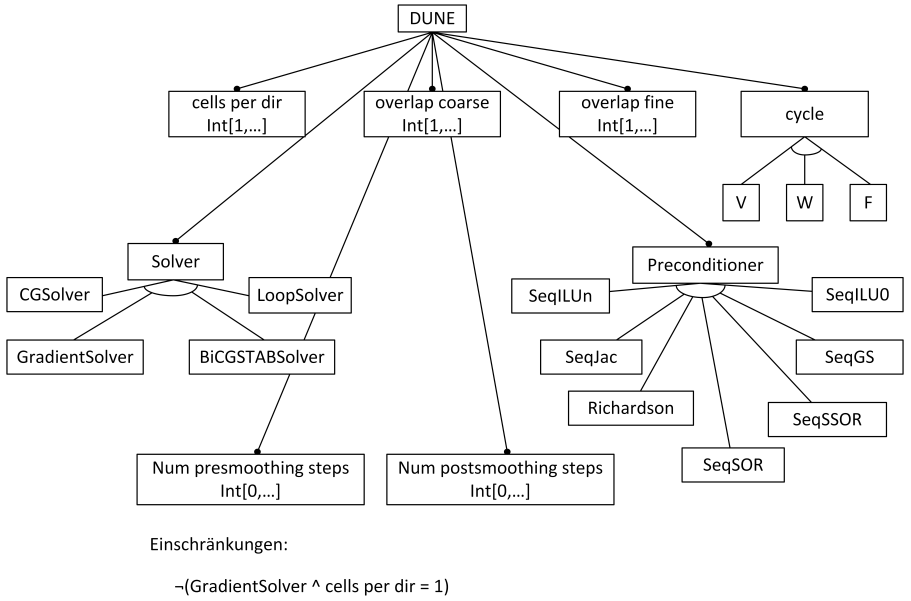



Abbildung 4.7: Das Feature Modell von MGS mit den Erweiterungen.

### 4.4.1 Anzahl an Vor- und Nachglättungsschritten

Neu ist das Konfigurieren einer bestimmten Anzahl von Vor- und Nachglättungsschritten. Vor dieser Erweiterung war die Anzahl der Vor- und Nachglättungsschritte fest. Außerdem wurden immer genauso viele Nachglättungs- wie Vorglättungsschritte ausgeführt. Durch diese Erweiterung kann nun sowohl die Anzahl der Vor-, als auch die Anzahl der Nachglättungsschritten eingegeben werden. Dabei kann es sein, dass für verschiedene Vorkonditionierer auch eine unterschiedliche Anzahl an Vor- beziehungsweise Nachglättungsschritten gebraucht werden, um eine optimale Laufzeit zu erzielen.

### 4.4.2 Zusätzliche Zyklenarten

Wie in HSMGP war es bisher in MGS der Fall, dass man lediglich einen V-Zyklus verwenden konnte. Dadurch sind mehr Iterationen nötig, als durch das Verwenden eines W- oder eines F-Zyklus. Deshalb wurden auch diese implementiert, obgleich es sich nicht so wie in HSMGP (siehe Abschnitt 4.3.1) implementieren ließ, da es sonst zu `memory-corruption`-Fehlern gekommmen ist.

In MGS gibt es einen Codeabschnitt, der in Listing 4.3 aufgeführt wird. Bisher war *gamma_* immer 1 und daher wurde für jedes Level der V-Zyklus lediglich ein einziges Mal ausgeführt.
Setzt man das *gamma_* auf 2, so entspricht es einem W-Zyklus, da für jedes Level zweimal ein V-Zyklus ausgeführt wird. Dies wird in Abbildung 4.8 exemplarisch dargestellt. Dabei wird hauptsächlich die Variable $i$ aus der For-schleife betrachtet des Codeabschnitts betrachtet.

```
1 int g = gamma_;
2 if (coarsestLevel + 1) {
3   g = 1;
4 }
5 for (int i = 0; i < g; i++) {
6   performCycle();
7 }
```

Listing 4.3: Zyklus in MGS

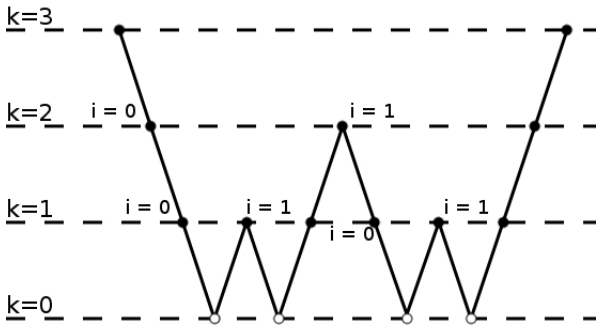



Abbildung 4.8: Der W-Zyklus in MGS.

Der F-Zyklus besitzt die Eigenschaft, dass der Löser nicht so oft wie beim W-Zyklus ausgeführt wird, jedoch öfter als beim V-Zyklus. Dies zu implementieren war etwas schwieriger, ließ sich am Ende jedoch auch mit einem weiteren Vektor *hasLevel*, ähnlich wie in Abschnitt 4.3.1. Betrachtet man nun den F-Zyklus (siehe Abbildung 4.9) bezüglich der Iterationsvariable $i$, so fällt einem auf, dass $i$ manchmal den Wert 0, als auch 1 und manchmal nur den Wert 0 annehmen kann. Bei näherer Betrachtung der Abbildung 4.9 für $k = 1$ fällt auf, dass $i$ nur beim ersten Durchlauf des Levels 0 und 1 sein kann. Ansonsten wird der V-Zyklus für $k = 1$ nur ein einziges Mal ausgeführt. So verhält es sich auch mit $k = 2$.
Damit abgespeichert werden kann, ob in einem Level $g$ schon den Wert 2 angenommen hat, wird der zuvor erwähnte Vektor *hasLevel* gebraucht. Der finale Code ist als Pseudocode in Listing 4.4 aufgeführt. Durch die hinzugefügten Zeilen (Zeilen 1 - 8) wird $g$ für jedes Level zuerst mit dem Wert 2 belegt und mit dem Wert 1 sonst. Schließlich erhalten wir mit der Änderung als Resultat den F-Zyklus wie ein Abbildung 4.9.

### 4.4.3 Weitere Löser und Vorkonditionierer

Bisher wurde im Programm immer fest *SeqSSOR* als Vorkonditionierer und *CGSolver* als Löser genommen. Damit man auch durch die Löser und Vorkonditionierer besser bestimmen kann, welche Kombination dieser beiden am besten geeignet ist, wurde auch hier die Variabilität erhöht. In Tabelle 4.3 sind die derzeitigen Löser mitsamt der Definitionen aufgeführt.

Beim GradientSolver gibt es allerdings die Einschränkung, dass dabei `cells per dir` nicht 1 sein darf, da sonst durch das Verfahren `not a number`-Werte verglichen werden und deshalb das Ergebnis welches dieses Verfahren liefert verfälscht werden kann.

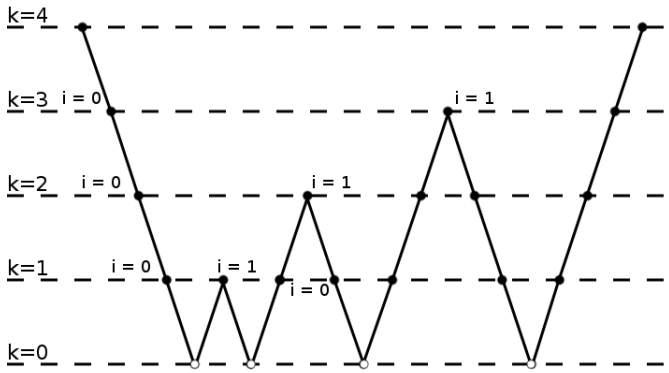



Abbildung 4.9: Der F-Zyklus in MGS.

```
1  if (F-Cycle) {
2    if (!coarsestLevel && !finestLevel && !hasLevel[currentLevel]) {
3      g = 2;
4      hasLevel[currentLevel] = true;
5    } else {
6      g = 1;
7    }
8  }
9
10 if (coarsestLevel + 1) {
11   g = 1;
12 }
13 for (int i = 0; i < g; i++) {
14   performCycle();
15 }
```

Listing 4.4: Erweiterung für den F-Zyklus in MGS

| Löser | Definition |
|---|---|
| BiCGSTABSolver | X_BiCGSTABSolver |
| CGSolver | X_CGSolver |
| GradientSolver | X_GradientSolver |
| LoopSolver | X_LoopSolver |

Tabelle 4.3: Eine Gesamtübersicht über die Löser.

Die verwendeten Vorkonditionierer sind in Tabelle 4.4 mit ihren Definitionen für den Präprozessor aufgelistet.

Die Löser und Vorkonditionierer wurden in Grebhahn et al. in [GKS+14] implementiert und durch diese Arbeit übernommen.

| Vorkonditionierer | Definition |
|---|---|
| SeqGS | X_SeqGS |
| SeqILU0 | X_SeqILU0 |
| SeqILUn | X_SeqILUn |
| SeqJac | X_SeqJac |
| SeqSOR | X_SeqSOR |
| SeqSSOR | X_SeqSSOR |
| BiCGSTABSolver | X_BiCGSTABSolver |
| LoopSolver | X_LoopSolver |

Tabelle 4.4: Eine Gesamtübersicht über die Löser.

# 5. Evaluierung

In diesem Kapitel wird das Verhalten der Erweiterungen durch sogenannte Korrektheitstests überprüft. Außerdem werden Hypothesen auf Basis der vorhergehenden Kapitel aufgestellt. Die Hypothesen beschäftigen sich mit dem Verhalten der Laufzeit und der Iterationsanzahl bei unterschiedlichen Konfigurationen. Mit diesem Wissen ist es möglich, besser abzuschätzen, welche Konfigurationen bezüglich Iterationsanzahl oder bezüglich Laufzeit am besten sind. Um zu überprüfen, ob die Hypothesen tatsächlich zutreffen, werden mehrfach Tests mit unterschiedlichen Konfigurationen ausgeführt, welche anschließend ausgewertet werden. Durch Laufzeittests werden zudem noch unterschiedliche Komponenten bezüglich ihrer Laufzeit analysiert, auf welche in dieser Arbeit noch keine Hypothesen verfasst werden. Jedoch ist keine Evaluierung aller Konfigurationen möglich, da die Anzahl an Konfigurationen durch die bisherige Variabilität und durch die Erweiterungen hoch ist. Im Folgenden werden die Anzahl an Konfigurationen für jedes der Programme bestimmt und angegeben.

## 5.1 Anzahl an möglichen Konfigurationen

In diesem Abschnitt wird die Anzahl an möglichen Konfigurationen für die Programme HSMGP und MGS berechnet. Dabei werden lediglich die Feature berücksichtigt, welche tatsächlich das Verhalten des Programms ändern. Optionen, wie etwa eine detailiertere Ausgabe sind dabei beispielsweise nicht berücksichtigt. Außerdem werden bei numerischen Konfigurationen alle Werte berücksichtigt, sofern diese nicht vom Typ **double** oder nach oben beschränkt sind. Sonst werden nur einige Werte genommen. Daher werden die Anzahl an Möglichkeiten bezüglich der Wertebereiche nach unten abgeschätzt. Darüber hinaus werden alle Programme bis zu einer maximalen Anzahl von 4 Prozessen getestet, da das in dieser Arbeit verwendete Testgerät 4 Kerne enthält.

#### HSMGP

Anhand Abbildung 4.2 und Abbildung 4.3 sind die verschiedenen Optionen ersichtlich. In Tabelle 5.1 wird die Anzahl an Konfigurationsmöglichkeiten für HSMGP

ausgerechnet, allerdings mit der Voraussetzung, dass die Gittergröße konstant ist. Zudem wurde die Einschränkung ignoriert, dass mindestens ein Vor- oder mindestens ein Nachglättungsschritt ausgewählt sein muss.

| | | |
|---|---|---|
| 7 | Anzahl an Löser | $*$ |
| 8 | Anzahl an Glätter | $*$ |
| 7 | Vorglättungsschritte | $*$ |
| 7 | Nachglättungsschritte | $*$ |
| 3 | Zyklen | $*$ |
| 100 | Levelanzahl | $*$ |
| 100 | Omega | $*$ |
| 100 | Anzahl an Elementen je Thread | $*$ |
| 4 | Anzahl an Prozessen | |
| 32928000000 | Möglichkeiten | |

Tabelle 5.1: Anzahl an Konfigurationsmöglichkeiten bei HSMGP.

Jede Konfiguration wird fünfmal getestet, um Schwankungen in den Messungen durch die Ausführung anderer Prozesse zu verkleinern. Damit müsste man das Programm 164640000000 Mal ausführen, damit der Fehler bei der Messung aller Konfigurationen klein gehalten wird. Wenn man davon ausgeht, dass eine Konfiguration in 10 Sekunden gemessen ist, wären alle Konfigurationen in 1646400000000 Sekunden gemessen. Dies wäre umgerechnet eine Dauer von 19055556 Tagen, oder 635186 Monaten. Das ist jedoch im Rahmen dieser Bachelorarbeit nicht möglich. Deshalb werden die Werte vieler Optionen bei späteren Testfolgen festgelegt.

### MGS

Die Berechnung für MGS stützt sich auf Abbildung 4.7 und erfolgt in Tabelle 5.2 unter der Annahme, dass die Gittergröße und die Überlappung (`overlap coarse` sowie `overlap fine`) bereits vorgegeben sind. Die Einschränkung, dass bei Nutzung des Lösers GradientSolver die Option `cells per dir` nicht 1 sein darf, wurde hier nicht berücksichtigt.

| | | |
|---|---|---|
| 4 | Anzahl an Löser | $*$ |
| 7 | Anzahl an Glätter | $*$ |
| 7 | Vorglättungsschritte | $*$ |
| 7 | Nachglättungsschritte | $*$ |
| 3 | Zyklen | $*$ |
| 100 | Levelanzahl | $*$ |
| 4 | Anzahl an Prozessen | |
| 1646400 | Möglichkeiten | |

Tabelle 5.2: Anzahl an Konfigurationsmöglichkeiten bei MGS.

Damit Messschwankungen verkleinert werden, ist auch hier ein mehrfaches Ausführen des Programms nötig. Bei fünffacher Ausführung je Konfiguration müsste das Programm 8232000 Mal ausgeführt werden. Wie bei HSMGP wird hier weiterhin davon ausgegangen, dass eine Konfiguration 10 Sekunden benötigt. Damit würden

alle Tests innerhalb von 82320000 Sekunden durchlaufen. Das wären wiederum 953 Tage beziehungsweise 32 Monate. Auch diese Zeit wäre noch viel zu lange, weswegen auch hier bestimmte Optionen festgelegt werden.

Im Folgenden wird die Korrektheit der Erweiterungen geprüft.

## 5.2 Korrektheit der Implementierung

In diesem Abschnitt wird die Korrektheit der Implementierung durch Tests überprüft. Dabei wird vor allem auf das spezifische Verhalten verschiedener Komponenten eingegangen, die im Rahmen dieser Arbeit hinzugefügt wurden.

**1. Die Anzahl an Lösungsschritten ist bei den Zyklen berechenbar**

Zuvor wurden in Abschnitt 2.3.6 Formeln angegeben, anhand dessen die Anzahl an Lösungsschritten für verschiedene Zyklenarten berechnet werden kann. Falls die Zyklen richtig funktionieren, gilt dies auch bei den hinzugefügten Zyklenarten von HSMGP und MGS.

Für die Tests sind alle anderen Optionen nicht von Relevanz, da die Anzahl an Lösungsschritten immer gleich ist und nur von der Anzahl an Vergröberungen abhängt. Deshalb sind diese redundant und werden hier nicht aufgelistet.

Die Ergebnisse von HSMGP sind in Abbildung 5.1 ersichtlich. Dabei können alle Werte anhand der Formel in Abschnitt 2.3.6 ausgerechnet werden. Die daraus erhaltenen Resultate stimmen mit den Ergebnissen in der Abbildung überein.

In Abbildung 5.2 sind die Ergebnisse von MGS sichtbar. Dabei wurden nur maximal 6 Vergröberungsschritte ausgeführt, da der Arbeitsspeicher für mehr Vergröberungen nicht ausreichte.

An den Abbildungen kann beobachtet werden, wie viele Löserschritte die verschiedenen Zyklen ausführen. Ebenfalls ersichtlich ist der exponentielle Wachstum des W-Zyklus im Gegensatz zum linearen Wachstum des F-Zyklus. Der V-Zyklus dagegen ist konstant.

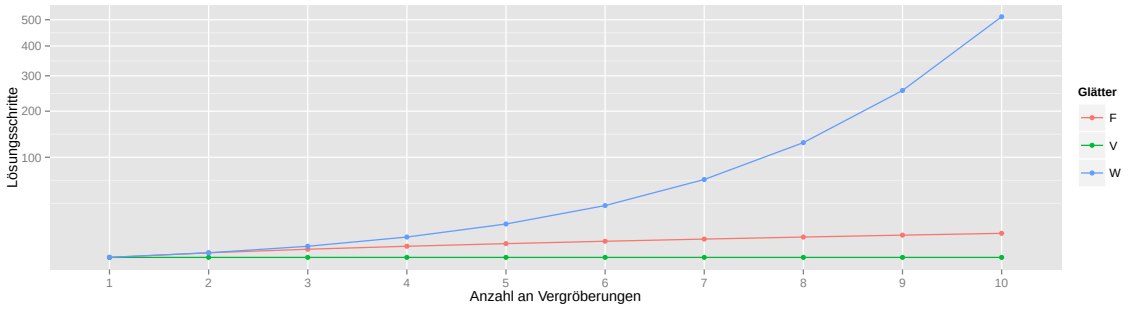



Abbildung 5.1: Anzahl an Lösungsschritten bei verschiedenen Zyklen in HSMGP.

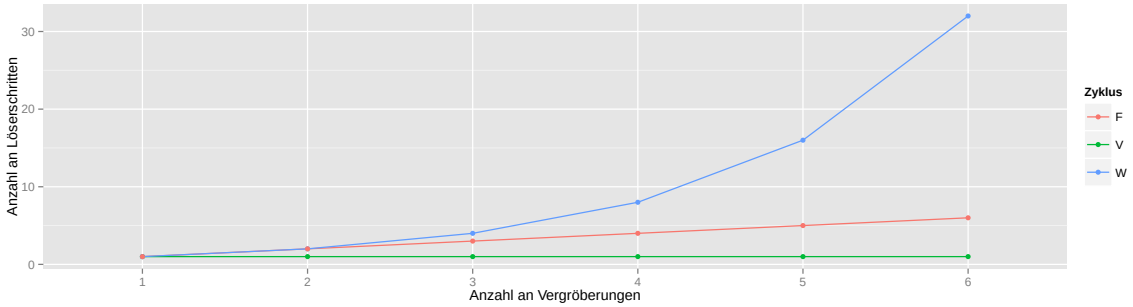



Abbildung 5.2: Anzahl an Lösungsschritten bei verschiedenen Zyklen in MGS.

**2. Die Anzahl an Iterationen bei unterschiedlichen Gittergrößen ist nahezu konstant**

In Kapitel 2 wurde außerdem erwähnt, dass die Anzahl an Iterationen nicht von der Gittergröße abhängt. Also müsste die Anzahl der Iterationen bei sich verändernder Gittergröße nahezu konstant bleiben.

Ein Test in HSMGP mit verschieden vielen Elementen je Gitter wurde mit allen Lösern ausgeführt und ist in Abbildung 5.3 zu sehen. Die dafür festgelegten Parameter sind in Tabelle 5.3 aufgelistet. Lediglich bei einer Größe von 4 Elementen (1,1 in der Abbildung) kam es bei den Tests zu einer geringen Abweichung, wobei eine Iteration mehr benötigt wurde. Eine unterschiedliche Anzahl an Iterationen tritt auf, wenn die Messgenauigkeit noch nicht die erwünschte Genauigkeit erreicht hat. In diesem Test betrug die Messgenauigkeit bei einer Größe von 4 Elementen (1,1 in der Abbildung) in der 9. Iteration den Wert `1,39296e-07`. Der erwünschte Wert war allerdings `1,243939e-07`. Da der erwünschte Wert noch nicht erreicht war, wurde noch zusätzlich eine Iteration ausgeführt. Die Abweichung ist allerdings minimal, so dass die Messung bei einer Größe von 4 Elementen vernachlässigbar ist.

In MGS wurden ebenfalls verschiedene Gittergrößen bezüglich ihrer Anzahl an Iterationen getestet. Das Ergebnis ist in Abbildung 5.4 zu sehen. Dabei wurde lediglich ein Löser getestet, da sich die anderen Löser bezüglich der Änderung der Gittergröße ebenso verhalten. Die festgelegten Optionen sind in Tabelle 5.4 aufgelistet.

Dadurch bestätigt sich, dass **die Anzahl an Iterationen bei unterschiedlichen Gittergrößen nahezu konstant bleibt**.

| Option | Wert |
|---|---|
| Prozesse | 4 |
| coarsestLvl | 0 |
| cycle | V |
| finestLvl | 2 |
| numBlocksX | 2 |
| numBlocksY | 2 |
| numPreSmoothingSteps | 2 |
| numPostSmoothingSteps | 2 |
| maxNumIterations | 100 |
| Smoother | SMOOTHER_JACOBI |

Tabelle 5.3: Die festgelegten Konfigurationsoptionen für CG und BoomerAMG.

| Option | Wert |
|---|---|
| Prozesse | 4 |
| cycle | V |
| number of refinements | 4 |
| overlapCoarse | 0 |
| overlapFine | 0 |
| Smoother | SeqSOR |
| Solver | CGSolver |

Tabelle 5.4: Die festgelegten Konfigurationsoptionen für CG und BoomerAMG.

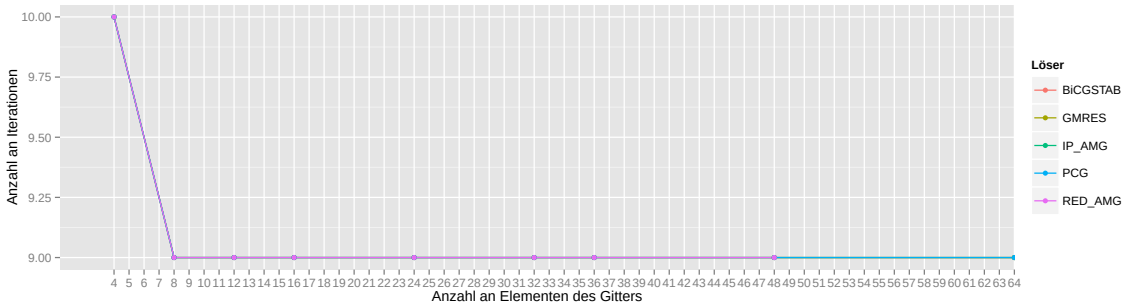



Abbildung 5.3: Anzahl an Iterationen der verschiedenen Löser für unterschiedliche Gittergröße in HSMGP.

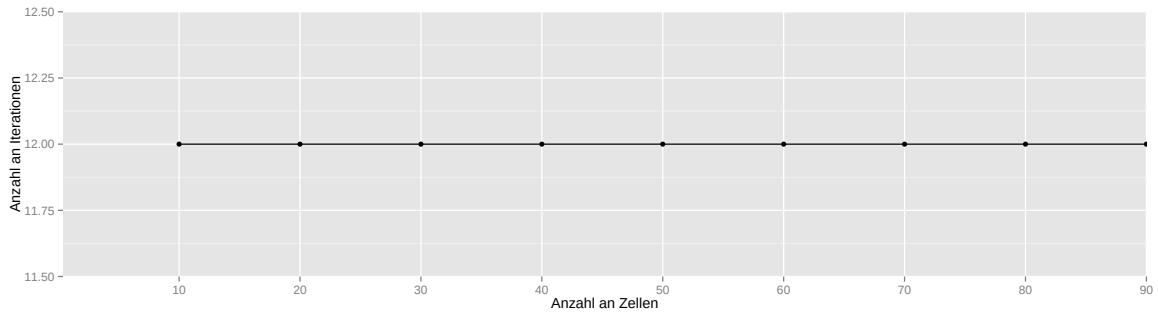



Abbildung 5.4: Anzahl an Iterationen der verschiedenen Löser für unterschiedliche Gittergröße in MGS.

## 5.3 Hypothesen

In diesem Abschnitt werden Hypothesen, also Annahmen über die verschiedenen Komponenten und die Verhaltensweise des Mehrgitterverfahrens aufgestellt. Diese werden in Abschnitt 5.4 anhand von Tests der in dieser Arbeit behandelten Programme HSMGP und MGS ausgewertet.

**1. Jacobi-Glätter braucht mindestens so viele Iterationen wie Gauss-Seidel-Glätter**

Wie in Abschnitt 2.3.2 erklärt, arbeitet der Jacobi-Glätter auf einem Lese- und Schreibgitter. Der Gauss-Seidel-Glätter arbeitet auf nur einem einzigen Gitter. Dies ist für die Glättungseigenschaft von Vorteil, da der Glätter mit den neuen Werten der schon bearbeiteten Gitterpunkten weiter rechnet. Daher braucht man bei der Ausführung des Mehrgitterverfahrens mit Gauss-Seidel auch gleich viele oder weniger Iterationen als mit dem Jacobi-Glätter.

**2. Ein Mehrgitterlöser braucht unter Verwendung eines V-Zyklus mehr Iterationen als der F- und W-Zyklus, um eine gewisse Konvergenz zu erhalten**

Das Verhalten des V-, W- und F-Zyklus ist in Abschnitt 2.3.6 erklärt worden. Diese haben demnach den Unterschied, dass beim V-Zyklus der Löser nur ein einziges Mal angewendet wird. Beim F- und beim W-Zyklus wird dies öfter gemacht. Durch das mehrfache Anwenden eines Lösers auf einem Gitter ergibt sich eine höhere Genauigkeit bei der Lösung nach einer Iteration. Infolgedessen müssten durch die höhere Genauigkeit weniger Iterationen durchgeführt werden, um die vordefinierte Konvergenz zu erreichen.

**3. CG ist bei wenigen Kernen performancemäßig besser als BoomerAMG**

Im Programm HSMGP wurde außer BoomerAMG(IP_HYPRE) noch der Löser Konjugierter-Gradient (CG) implementiert. Diese beiden Löser unterscheiden sich durch ihre Parallelität. CG hat bei einer niedrigeren Anzahl an Prozessen (weniger als 8.000) eine bessere Laufzeit als BoomerAMG. Ansonsten ist BoomerAMG performancemäßig besser. In Kuckuk et al. [KGKR13] wird dies anhand von Tests mit mehreren tausend Kernen gezeigt. Bei wenigen Kernen (bis zu 4 im Rahmen dieser Arbeit) sollte deshalb CG bezüglich der Laufzeit besser sein.

**4. Iterationsanzahl ändert sich nicht bei Wechsel von Löser**

Löser haben einen geringen Einfluss auf die Glättung und damit auch keinen oder nur einen sehr kleinen Einfluss auf die Konvergenzrate. In MGS hängt die Iterationsanzahl nur von der Konvergenzrate ab.
Dadurch müsste die Iterationsanzahl bei beiden Programmen immer gleich bleiben, sofern man lediglich den Löser wechselt.

**5. Die Anzahl an Iterationen sinkt mit zunehmender Anzahl an Vor- und Nachglättungsschritten**

Glätter sind in Abschnitt 2.3.2 näher erklärt. Diese glätten immer stärker, je öfter man diese verwendet. Damit müssten die Fehlerschwankungen immer kleiner werden und infolgedessen die Genauigkeit beim Lösen erhöht werden. Durch die Erhöhung von Vor- und Nachglättungsschritten müsste also die Anzahl an Iterationen verkleinert werden.

## 5.4 Auswertung der Hypothesen

In diesem Abschnitt werden die Hypothesen aus Abschnitt 5.3 durch Tests ausgewertet. Folgende Tests wurden auf einem Laptop mit einem Arbeitsspeicher von 16 GB und i7-4700MQ Prozessor unter Ubuntu 14.04 ausgeführt. Jeder Test wurde fünfmal ausgeführt. Die erzielten Laufzeiten in den Tests wurden hinterher wieder zusammengefasst und der Mittelwert der Ergebnisse gebildet. Für die Visualisierung wurden zudem oft die Glätter und Löser festgelegt, da das Verhalten der Glätter und Löser bezüglich der in den Hypothesen betrachteten Aspekten ähnlich und daher auch redundant ist. Diese Redundanz wurde durch das Festlegen der Glätter und Löser vermieden.

**1. Jacobi-Glätter braucht mehr Iterationen als Gauss-Seidel-Glätter**

Die Jacobi und Gauss-Seidel Glätter wurden ausschließlich in HSMGP getestet, da Jacobi und Gauss-Seidel in MGS nicht als Glätter, sondern als Vorkonditionierer vorhanden sind. Die Konfigurationsoptionen aus Tabelle 5.5 waren dabei fest gewählt.

Zuvor wurde die Hypothese aufgestellt, dass Gauss-Seidel maximal so viele Iterationen haben kann wie Jacobi. Anhand Abbildung 5.5 ist zu erkennen, dass es bei allen Tests zutrifft. Dabei ist die Einstellung der Vor- und Nachglättungsschritten in Form von $x, y$ gegeben. Die erste Zahl, $x$, steht für die Anzahl an Vorglättungsschritten und die zweite Zahl, $y$, für die Anzahl an Nachglättungsschritten. Auf der Y-Achse werden die insgesamt benötigten Iterationen angezeigt. Wie man sieht, tritt dabei oft der Fall ein, dass für Gauss-Seidel gleich viele Iterationen benötigt werden wie

| Option | Wert |
|---|---|
| Prozesse | 4 |
| coarsestLvl | 0 |
| finestLvl | 3 |
| numBlocksX | 2 |
| numBlocksY | 2 |
| numElementsPerBlockX | 8 |
| numElementsPerBlockY | 8 |
| maxNumIterations | 100 |
| cycle | V |
| Solver | IP_HYPRE |

Tabelle 5.5: Die festgelegten Konfigurationsoptionen für den Jacobi- und Gauss-Seidel-Glätter in HSMGP.

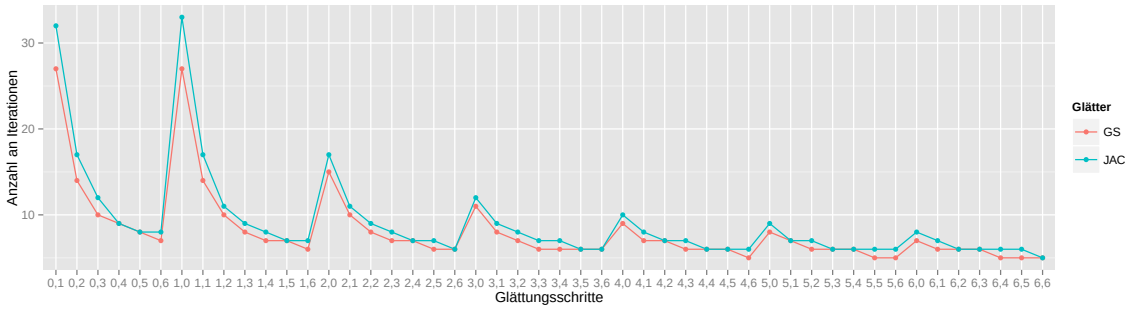



Abbildung 5.5: Die Anzahl an Iterationen von Gauss-Seidel und Jacobi mit unterschiedlichen Glättungseinstellungen.

für Jacobi. Manchmal kommt es jedoch auch vor, dass weniger Iterationen benötigt werden. Dies lässt sich durch die stärkere Glättungseigenschaft von Gauss-Seidel erklären. Ebenfalls in der Abbildung ersichtlich ist, dass GS bei den Glättungsschritten nie mehr Iterationen braucht als Jacobi. Damit wird die Hypothese **Jacobi-Glätter braucht mehr Iterationen als Gauss-Seidel-Glätter** in den Tests bestätigt.

**2. Ein Mehrgitterlöser braucht unter Verwendung eines V-Zyklus mehr Iterationen als der F- und W-Zyklus, um eine gewisse Konvergenz zu erhalten**

Der F- und W-Zyklus wurden durch die Erweiterungen zu den Programmen hinzugefügt. Diese wurden bezüglich der Anzahl an Iterationen im Rahmen dieser Arbeit getestet. In Tabelle 5.6 und Tabelle 5.7 sind die festgelegten Konfigurationsoptionen der Tests der beiden Programme angegeben.

Anhand Abbildung 5.6 ist ersichtlich, dass sich die Hypothese über die Zyklen bestätigt. Der W-Zyklus hat genauso oder weniger Iterationen für manche Optionen gebraucht. Der F-Zyklus hat sich bezüglich der Iterationen gleich verhalten wie der W-Zyklus und wird deswegen überlappt.

| Option | Wert |
|---|---|
| Prozesse | 4 |
| coarsestLvl | 0 |
| finestLvl | 2 |
| numBlocksX | 2 |
| numBlocksY | 2 |
| numElementsPerBlockX | 8 |
| numElementsPerBlockY | 8 |
| maxNumIterations | 100 |
| Smoother | SMOOTHER_JACOBI |
| Solver | IP_HYPRE |

Tabelle 5.6: Die festgelegten Konfigurationsoptionen für die Auswertung der Zyklen in HSMGP.

| Option | Wert |
|---|---|
| Prozesse | 4 |
| cells per direction | 80 |
| number of refinements | 4 |
| overlapCoarse | 0 |
| overlapFine | 0 |
| Smoother | SeqSOR |
| Solver | LoopSolver |

Tabelle 5.7: Die festgelegten Konfigurationsoptionen für die Auswertung der Zyklen in MGS.

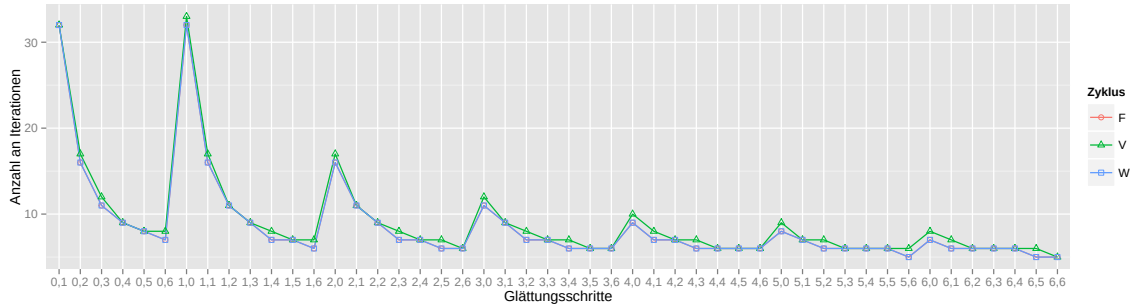



Abbildung 5.6: Iterationsanzahl der Zyklen V, W und F bei verschiedenen Konfigurationen in HSMGP.

Die Ergebnisse für MGS sind in Abbildung 5.7 visualisiert. Darin ist besser zu erkennen, dass der V-Zyklus mehr oder gleich viele Iterationen braucht, als F und W. Dies lässt sich dadurch erklären, dass das Gitter beziehungsweise das zugrundeliegende Problem anders ist, als in HSMGP. Außerdem wurde in der Abbildung erst mit mindestens einem Vor- und mindestens einem Nachglättungsschritt begonnen, da MGS ansonsten teilweise über 2000 Iterationen durchführt.

Durch die beiden Tests wurde die Hypothese bestätigt, dass **der V-Zyklus mehr oder gleich viele Iterationen braucht, als der F- und der W-Zyklus**.

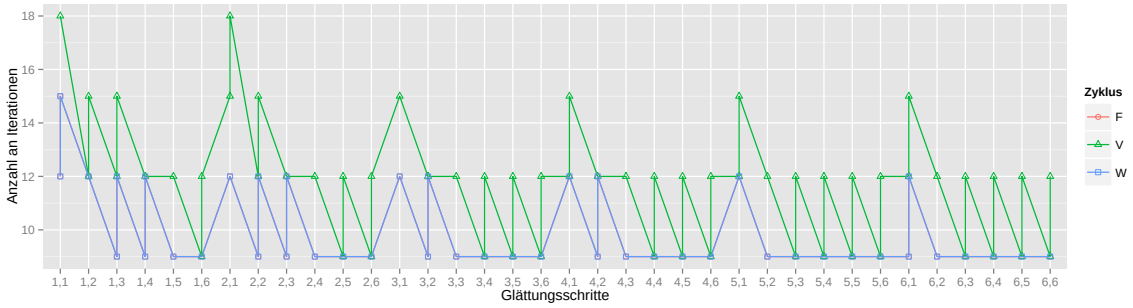



Abbildung 5.7: Iterationsanzahl der Zyklen V, W und F bei verschiedenen Konfigurationen in MGS.

**3. CG ist bei wenigen Kernen performancemäßig besser als BoomerAMG**

Die Löser CG (IP_CG) und BoomerAMG (IP_HYPRE) wurden für die Auswertung dieser Hypothese mit unterschiedlicher Anzahl an Prozessen getestet. Die dafür festgelegten Optionen sind in Tabelle 5.8 aufgelistet. Das Ergebnis des Tests ist in Abbildung 5.8 zu sehen. Hierbei fällt auf, dass BoomerAMG entgegen der Hypothese bezüglich der Laufzeit bei 1-4 Kernen wesentlich schneller ist als **CG**. Auch bei sequentieller Ausführung ist BoomerAMG besser als **IP_CG**, auch wenn der Unterschied bei nur 0,04 Sekunden Unterschied liegt. Dies widerlegt die Hypothese, dass **CG performancemäßig besser als BoomerAMG ist**, zumindest für eine Anzahl von Prozessen zwischen 1-4. Grund dafür könnte die Struktur der benutzten Hardware sein. Ein anderer vorstellbarer Grund könnte sein, dass CG zwischen 4 und 512 Prozessen besser wird als BoomerAMG. Auch bei den Tests der ursprünglichen Version (also der Version ohne Erweiterungen) wurde dieses Verhalten beobachtet.

| Option | Wert |
|---|---|
| coarsestLvl | 0 |
| finestLvl | 2 |
| numBlocksX | 2 |
| numBlocksY | 2 |
| numElementsPerBlockX | 10 |
| numElementsPerBlockY | 10 |
| numPreSmoothingSteps | 2 |
| numPostSmoothingSteps | 2 |
| maxNumIterations | 100 |
| Smoother | SMOOTHER_JACOBI |

Tabelle 5.8: Die festgelegten Konfigurationsoptionen für CG und BoomerAMG.

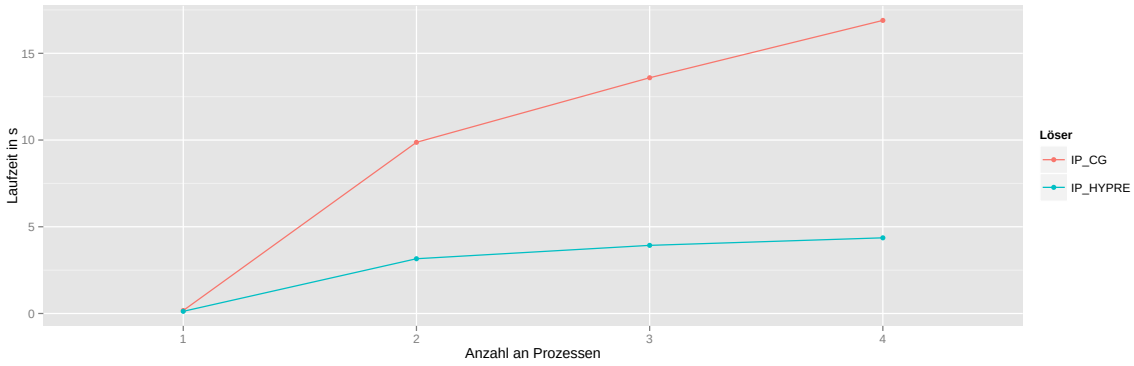



Abbildung 5.8: Durchschnittliche Laufzeit der Löser CG und BoomerAMG bei unterschiedlicher Anzahl an Prozessen.

**4. Iterationsanzahl ändert sich nicht bei Wechsel von Löser**

Um diese Hypothese zu bestätigen oder zu widerlegen muss sowohl HSMGP, als auch MGS getestet. Die dafür festgelegten Optionen sind in Tabelle 5.9 für HSMGP und in Tabelle 5.10 für MGS tabellarisch aufgelistet. Betrachtet man nun zuerst Abbildung 5.9, so scheint sich dies zu bewahrheiten. Durch die Abbildung ist ersichtlich, dass sich jeder Löser aus HSMGP bezüglich der Iterationsanzahl gleich verhält und sich dabei die einzelnen Löser überlappen.
Die Ergebnisse der Tests von MGS sind in Abbildung 5.10 visualisiert und zeigen, dass die Hypothese in MGS nicht gilt, da die verschiedenen Löser verschiedenes Verhalten bezüglich der Anzahl an Iterationen zu den jeweiligen Glättungsschritten aufweisen. Anhand der Abbildung ist zudem zu erkennen, welche Löser viele und welche Löser wenigere Iterationen brauchen. Etwa wäre dafür der CGSolver ein Löser, der sehr wenige Iterationen braucht und der LoopSolver der Löser, der die meisten Iterationen braucht.

Die Hypothese **Iterationsanzahl ändert sich nicht bei Wechsel von Löser** gilt somit nicht.

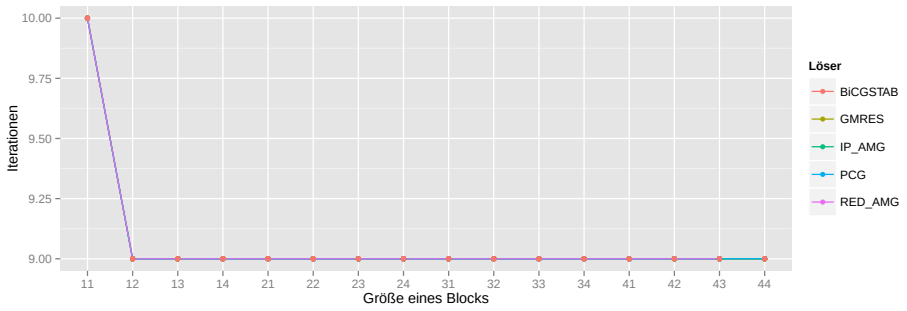



Abbildung 5.9: Iterationsanzahl verschiedener Löser bei verschiedenen Konfigurationen in MGS.

| Option | Wert |
|---|---|
| Prozesse | 4 |
| coarsestLvl | 0 |
| cycle | V |
| finestLvl | 2 |
| numBlocksX | 2 |
| numBlocksY | 2 |
| numElementsPerBlockX | 8 |
| numElementsPerBlockY | 8 |
| numPreSmoothingSteps | 2 |
| numPostSmoothingSteps | 2 |
| maxNumIterations | 100 |
| Smoother | SMOOTHER_JACOBI |

Tabelle 5.9: Die festgelegten Konfigurationsoptionen für HSMGP.

| Option | Wert |
|---|---|
| Prozesse | 4 |
| cells per direction | 80 |
| cycle | V |
| number of refinements | 4 |
| overlapCoarse | 0 |
| overlapFine | 0 |
| Smoother | SeqSOR |

Tabelle 5.10: Die festgelegten Konfigurationsoptionen für MGS.

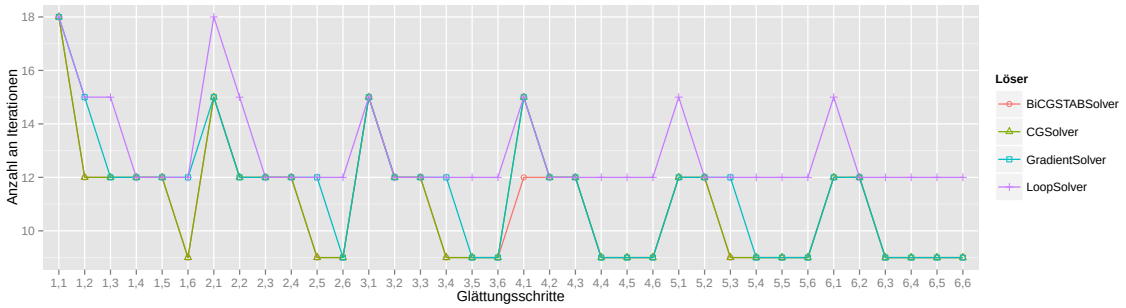



Abbildung 5.10: Iterationsanzahl verschiedener Löser bei verschiedenen Konfigurationen in MGS.

**5. Die Anzahl an Iterationen sinkt mit zunehmender Anzahl an Vor- und Nachglättungsschritten**

Damit diese Hypothese verifiziert werden kann, wurden sowohl HSMGP, als auch MGS getestet. Für die Ausführung der Tests wurden die Optionen aus Tabelle 5.11 für HSMGP und Tabelle 5.12 für MGS festgelegt.

Die Ergebnisse bezüglich HSMGP und MGS werden zur besseren Visualisierung als eine 3D-Grafik dargestellt.

In HSMGP flacht die Anzahl der Iterationen mit zunehmender Anzahl an Vor- und Nachglättungsschritten schnell ab, wie in Abbildung 5.11 (a) zu sehen ist. Da mit 0 Vor- und 0 Nachglättungsschritten die Genauigkeit nicht besser wird, werden so viele Iterationen durchgeführt wie durch die maximale Iterationsanzahl angegeben. Das wären im Fall des Tests 100 Iterationen. Damit die anderen Glättungsschritte hervorgehoben werden, wurde die Iterationsanzahl bei 0 Vor- und 0 Nachglättungsschritten auf 35 gestellt.

In MGS flacht es in Abbildung 5.11 (b) aufgrund des gewählten Glätters `SeqSOR` nicht so schnell ab und ist auch nicht symmetrisch, wie der `Jacobi`-Glätter in HSMGP. Außerdem musste dabei für jeden Vor- und Nachglättungsschritt mindestens die Zahl 1 genommen werden, da MGS ansonsten bei einigen Konfigurationen über 2000 Iterationen durchführen würde. Doch auch hier bestätigt sich, dass mit zunehmender Anzahl an Vor- und Nachglättungsschritten die Anzahl an Iterationen kleiner wird.

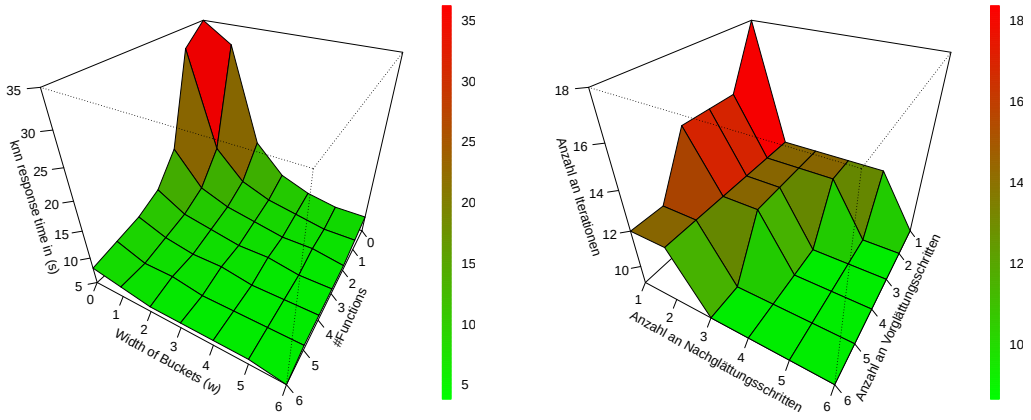
Damit ist die Hypothese **die Anzahl an Iterationen sinkt mit zunehmender Anzahl an Vor- und Nachglättungsschritten** durch die Tests in MGS und HSMGP bestätigt.

| Option | Wert |
|---|---|
| Prozesse | 4 |
| coarsestLvl | 0 |
| cycle | V |
| finestLvl | 2 |
| numBlocksX | 2 |
| numBlocksY | 2 |
| numElementsPerBlockX | 8 |
| numElementsPerBlockY | 8 |
| numPreSmoothingSteps | 2 |
| numPostSmoothingSteps | 2 |
| maxNumIterations | 100 |
| Smoother | SMOOTHER_JACOBI |
| Solver | COARSE_GRID_SOLVER_IP_HYPRE |

Tabelle 5.11: Die festgelegten Konfigurationsoptionen für HSMGP.

| Option | Wert |
|---|---|
| Prozesse | 4 |
| cells per direction | 80 |
| cycle | V |
| number of refinements | 4 |
| overlapCoarse | 0 |
| overlapFine | 0 |
| Smoother | SeqSOR |
| Solver | CGSolver |

Tabelle 5.12: Die festgelegten Konfigurationsoptionen für MGS.




(a) HSMGP (b) MGS

Abbildung 5.11: Iterationsanzahl bei unterschiedlicher Anzahl an Vor- und Nachglättungsschritten in HSMGP und MGS.

## 5.5 Laufzeittests

**Laufzeit von Jacobi und Gauss-Seidel**

Betrachtet man bei den Glättern Jacobi und Gauss-Seidel nicht nur die Anzahl an Iterationen, sondern auch die benötigte Laufzeit in Abbildung 5.13, so zeigt sich keine deutliche Verbesserung mehr zu Jacobi (siehe hierzu Abschnitt 5.4). Daraus lässt sich ableiten, dass obwohl Gauss-Seidel teilweise eine niedrigere Anzahl an Iterationen in Abbildung 5.5 benötigt, keine Aussage über die Laufzeit gemacht werden kann. Bei Betrachtung der Abbildung mit einem Vorglättungsschritt und 2 Nachglättungsschritten, fällt einem dabei auf, dass Gauss-Seidel eine Iteration weniger benötigt, laufzeitmäßig jedoch nicht besser ist als Jacobi. Nimmt man anstatt dem IP_HYPRE-Löser den CG-Löser, ergibt sich das Bild aus Abbildung 5.12, welches mit Abbildung 5.13 nur schwer vergleichbar ist. Es reicht also nicht, den Löser vom Glätter getrennt zu betrachten. Dies bedeutet, dass man, um die beste Einstellung

zu finden, alle Kombinationen von Lösern und Glättern durchgehen müsste, um die Konfiguration mit der besten Performance zu finden.

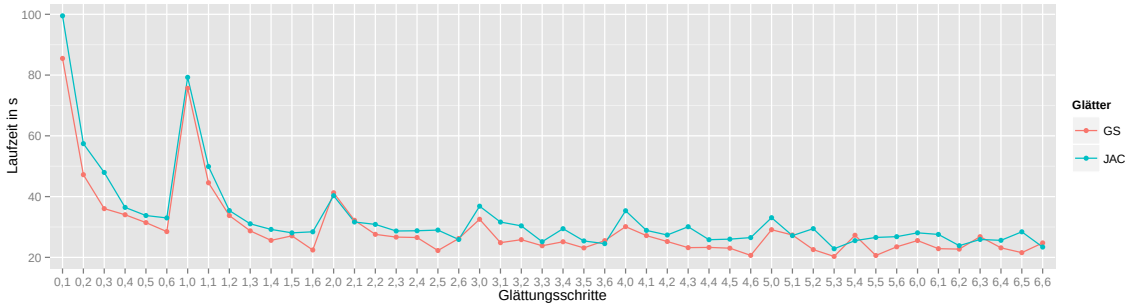



Abbildung 5.12: Die Laufzeit von Gauss-Seidel und Jacobi mit unterschiedlichen Glättungseinstellungen und mit dem Löser CG.

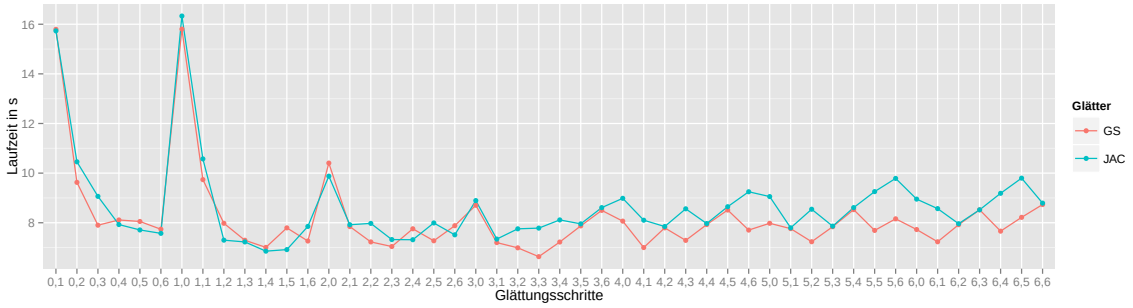



Abbildung 5.13: Die Laufzeit von Gauss-Seidel und Jacobi mit unterschiedlichen Glättungseinstellungen und mit dem IP_HYPRE-Löser.

**Laufzeit der Zyklen V, W und F**

Betrachtet man nun nicht mehr die Iterationsanzahl wie in Abbildung 5.6, sondern die Laufzeit, so wird dabei ein ganz anderes Bild wird geliefert. Durch Abbildung 5.14 ist ersichtlich, dass der F-Zyklus in Vergleich zu dem W-Zyklus bezüglich der Laufzeit besser ist. Zudem fällt auf, dass der V-Zyklus als schnellster Zyklus bewährt, obwohl dieser Zyklus manchmal oder immer mehr Iterationen braucht, als die anderen Zyklen. Der Grund dafür liegt größtenteils an der geringen Komplexität der durch das Programm betrachteten PDG. MGS hat andere Komponenten als HSMGP und liefert daher auch andere Ergebnisse für die Laufzeit der verschiedenen Zyklen. Diese sind in Abbildung 5.15 dargestellt. Dabei ist zu erkennen, dass manchmal der F-Zyklus tatsächlich bezüglich der Laufzeit besser ist als der V-Zyklus. Der F-Zyklus ist jedoch performancemäßig nie langsamer als der W-Zyklus.

Zusammengefasst kommt aus diesen Laufzeittests hervor, dass der V-Zyklus laufzeitmäßig besser ist als der F- und der W-Zyklus, obwohl der F- und der W-Zyklus weniger Iterationen brauchen als der V-Zyklus.

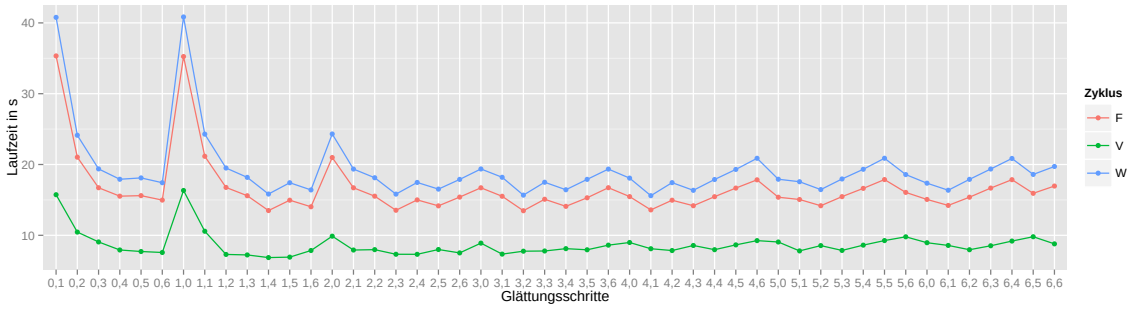



Abbildung 5.14: Laufzeit der Zyklen bei verschiedenen Konfigurationen in HSMGP.

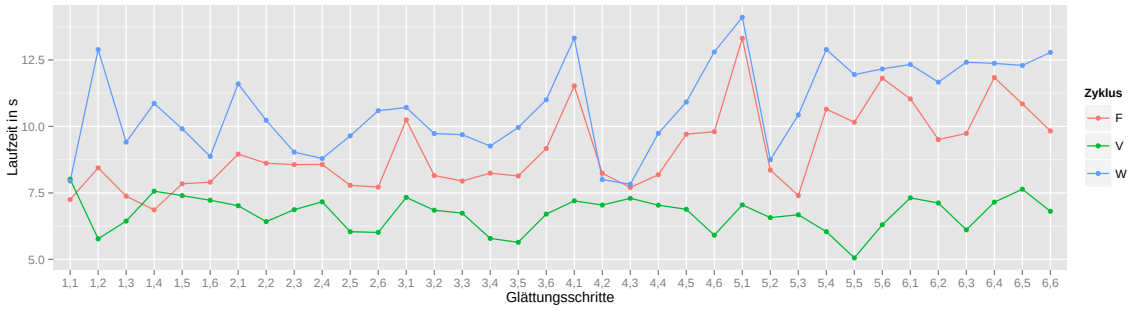



Abbildung 5.15: Laufzeit der Zyklen bei verschiedenen Konfigurationen in MGS.

# 6. Zusammenfassung und zukünftige Arbeiten

### Zusammenfassung

In dieser Arbeit wurde ein iteratives Verfahren zur Lösung von partiellen Differentialgleichungen, das Mehrgitterverfahren erklärt. Bei der Vorstellung des Verfahrens wurde auf die einzelnen Komponenten eingegangen, wie in etwa auf die Restriktion, Prolongation, den Glätter, den Löser und die verschiedenen Zyklen. Weiterhin wurden zwei Frameworks vorgestellt, welche verschiedene Algorithmen anbieten, um Mehrgitterlöser implementieren zu können. Durch die große Auswahl an Algorithmen ist es schwer Aussagen über die beste Kombination der Komponenten zu machen. Zwei Programme, HSMGP und MGS, verbinden dabei einige Komponenten der jeweiligen Frameworks und wurden im Rahmen dieser Arbeit noch zusätzlich erweitert, um noch mehr Kombinationen unterschiedlicher Komponenten testen zu können. Schließlich wurden Hypothesen aufgestellt, welche durch Tests unterschiedlicher Kombinationen ausgewertet worden sind. Durch diese Hypothesen wird überprüft, ob die in der Theorie getroffenen Aussagen auch für diese beiden Programme zutreffen.

### Zukünftige Arbeiten

Die Frameworks Hypre und DUNE bieten allerdings viel mehr Algorithmen, welche in Zukunft implementiert werden können. Es können beispielsweise noch verschiedene Restriktions- und Prolongationsoperatoren implementiert werden, welche ebenfalls einen Einfluss auf die Laufzeit und die Genauigkeit der Lösung haben. In HSMGP könnte dies gemacht werden, allerdings müsste man dafür viel mehr die angebotenen Algorithmen aus Hypre nutzen als bisher und das Programm HSMGP umbauen.
Außerdem können noch weitere Löser und Glätter hinzugefügt werden. Dafür muss MGS jedoch auf eine neuere Version des Frameworks DUNE portiert und dazu teilweise neu geschrieben werden, da mit der neueren Version des Frameworks zusätzliche Löser dazu kamen. Auch in HSMGP wäre es nötig, für weitere Glätter und

Löser einen Teil des Codes umzuschreiben.
Darüber hinaus könnten auch andere Gitterformen betrachtet werden. Im Rahmen dieser Arbeit hat man lediglich strukturierte Gitter betrachtet, jedoch gibt es auch beispielsweise unstrukturierte Gitter. Damit wäre die Unterscheidung und Performanceanalyse verschiedener Gitterformen möglich.

Eine weitere zukünftige Arbeit wäre die detailliertere Analyse der unterschiedlichen Komponenten. Im Rahmen dieser Arbeit wurden die Programme auf maximal 4 Kernen getestet, jedoch bieten HSMGP und MGS viel mehr Parallelität an. Das Ausführen der Programme auf dem derzeit schnellsten Supercomputer in Deutschland, Blue Gene/Q (JUQUEEN) wäre beispielsweise vorstellbar. Die Gittergröße wäre ein weiterer Punkt, welchen man in zukünftigen Messungen erhöhen könnte. Dazu ist jedoch mehr Arbeitsspeicher und mehr Zeit notwendig. Darüber hinaus könnte man auch die Anzahl an des maximalen Gitterlevels erhöhen. Bisher sind in HSMGP maximal 10 Vergröberungen möglich, da der Arbeitsspeicher für die Ausführung mehrerer Vergröberungen nicht ausreicht. In MGS dagegen gibt es keinen Limit außer den Arbeitsspeicher, der für die Ausführung des Programms zur Verfügung steht.

Außerdem könnte ein Framework erstellt werden, welches beide Programme, MGS und HSMGP nutzt und es ermöglicht, Algorithmen aus beiden Frameworks zu kombinieren. Ein großer Vorteil liegt darin, dass man dadurch eine größere Auswahl hat und dadurch eine noch bessere Kombination von Komponenten finden kann. Allerdings müssten hierzu auch HSMGP und MGS angepasst werden, da diese von unterschiedlicher Struktur sind und unterschiedliche Konfigurationsoptionen anbieten. Während beispielsweise in HSMGP das Gitter in Blöcke geteilt und jeder Block von einem Prozess behandelt wird, gibt man in MGS lediglich die Anzahl an Zellen in einer Richtung an. Ein weiterer Nachteil bei der Erstellung eines gemeinsamen Frameworks wäre der Datentransport von einem Programm zu dem Anderen zurück. Dies braucht zusätzlich Zeit.

# A. Anhang

## A.1 Ordnerstruktur auf der CD

Auf der CD findet man folgende Verzeichnisse:

- **DUNE**: Dies ist das Verzeichnis mit allen Modulen von DUNE inklusive MGS. Das Modul von MGS heißt `multigrid_overlapping`.
- **hypre-2.8.0b**: Dieser Ordner beinhaltet die Hypre-Bibliothek mit der Version 2.8.0b. Die Hypre-Bibliothek muss in HSMGP bei der Auswahl von Lösern und Glättern aus Hypre referenziert werden.
- **lapack-3.5.0**: Die LAPACK-Bibliothek bietet Funktionen der linearen Algebra an und wird für HSMGP gebraucht. Auch dies muss referenziert werden.
- **Poisson2D**: Poisson2D beinhaltet HSMGP.

Diese sollten am besten auf die lokale Festplatte kopiert werden und durch den Befehl `sudo chmod +rwx -R ./` werden die Rechte für die Dateien gesetzt, so dass man sowohl lesend, als auch schreibend zugreifen und manche Dateien zudem noch ausführen kann. Im Folgenden wird auf HSMGP und MGS näher eingegangen.

## A.2 HSMGP

### A.2.1 Installation von HSMGP

Um HSMGP ausführen zu können, werden einige Bibliotheken benötigt, die installiert und deren Pfad in der cmake-Datei angegeben werden müssen. Im Folgenden wird dies anhand des Betriebssystems Ubuntu, Version 14.04 näher erklärt.
Folgende Bibliotheken/Programme werden benötigt:

- **gcc, g++**: Normalerweise sind **gcc** und **g++** ebenfalls auf Ubuntu vorinstalliert. Sollte dies jedoch nicht der Fall sein, so werden diese durch

```
sudo apt-get install gcc g++
```

installiert.

- **make, cmake**: **make** ist auf Ubuntu bereits vorinstalliert. Nur cmake muss durch

  ```
  sudo apt-get install cmake
  ```

  installiert werden. Sollten hierbei Fehler wie zum Beispiel ein Fehler bezüglich des Pfades `/usr/bin/c++` auftreten, so empfiehlt sich die Ausführung von:

  ```
  sudo apt-get install build-essential
  ```

- **Boost**: Die **Boost**-Bibliothek stellt Elemente zur Verfügung, die wegen der Parallelität gebraucht werden. Außerdem wird diese Bibliothek für die Programmoptionen-Erweiterung gebraucht. **Boost** installiert man unter Ubuntu durch den Befehl:

  ```
  sudo apt-get install libboost-all-dev
  ```

  Falls beim Kompilieren `program_options` fehlen sollte, so kann man diese durch:

  ```
  sudo apt-get install libboost-program-options-dev
  ```

  installieren.

- **MPI**: Um das Programm parallel auf mehreren Prozessoren ausführen zu können, wird **MPI** benötigt. Für **MPI** gibt es mehrere unterschiedliche Bibliotheken, jedoch wurde Hypre mit **OpenMPI** programmiert. Deshalb kann es dazu kommen, dass unter Umstände bestimmte Verweise nicht richtig aufgelöst werden können, wenn man beispielsweise **MPICH** verwendet. Der Befehl lautet:

  ```
  sudo apt-get install libopenmpi-dev openmpi-bin
  ```

  Sollte es zu dem Problem kommen, dass die Prozesse sich untereinander während der Ausführung nicht kennen(mehrfache Ausgabe von „Initializing rank 0 of 1 “), so ist eine Neuinstallation von **OpenMPI** zu empfehlen.

- **Hypre, LAPACK**: Das **Hypre**-Framework wie auch die **LAPACK**-Bibliothek befinden sich in zwei verschiedenen Ordnern auf der zu dieser Arbeit gehörenden CD. Einzig und allein die Umgebungsvariablen müssen vor dem Ausführen von **make** noch gesetzt werden. Die Befehle für die Variablen `HYPRE_INCLUDE`, `HYPRE_LIB`, `LAPACK_LIB` lauten wie folgt:

  ```
  export HYPRE_INCLUDE=<Pfad zum Include-Ordner von Hypre>
  export HYPRE_LIB=<Pfad zur libHypre.a von Hypre>
  export LAPACK_LIB=<Pfad zum Lib-Ordner von LAPACK>
  ```

Im Fall, dass man die Ordner **hypre-2.8.0b** und **lapack-3.5.0** im Home-Verzeichnis hat, sehen die Befehle so aus:

```
export HYPRE_INCLUDE=~/hypre-2.8.0b/src/hypre/include
export HYPRE_LIB=~/hypre-2.8.0b/src/hypre/lib/libHypre.a
export LAPACK_LIB=~/lapack-3.5.0/
```

Sollte bei Hypre das Verzeichnis `/hypre/include` noch nicht existent sein, so muss dies erst noch erstellt werden. Dies geschieht mithilfe von:

```
cd ~/hypre-2.8.0b/src
./configure
make install
```

mit der Voraussetzung, dass sich **hypre-2.8.0b** im Home-Verzeichnis befindet. Wichtig ist dabei, dass diese Umgebungsvariablen nur gesetzt sein müssen, wenn man Löser oder Glätter, welche in HSMGP verwendet werden, wie in Tabelle A.1 benutzt.

Der `Poisson2D`-Ordner beinhaltet HSMGP. Auf diesen Ordner bezieht sich die folgende Beschreibung.

Durch die Datei `CMakeLists.txt` im Verzeichnis von HSMGP kann man Präprozessor-Anweisungen hinzufügen. Die Datei muss dafür durch ein Textbearbeitungsprogramm geöffnet werden. In Tabelle A.1 werden dabei die Definitionen aufgelistet, welche zur Auswahl stehen.

In der Datei `CMakeLists.txt` gibt es einen Abschnitt:

```
# ===== Definitions =====
add_definitions(-DVERBOSE)
```

Dadurch werden Präprozessor-Definitionen hinzugefügt und ausgegeben, damit man sich noch einmal vergewissern kann, dass die Definitionen auch gespeichert und erkannt wurden. Wichtig ist dabei, dass man vor die Optionen ein -D hängt. Für `VERBOSE` wäre der Befehl: `-DVERBOSE`. Außerdem schreibt man diese mit einem Leerzeichen getrennt hintereinander. Beispiel: `-DVERBOSE -DTIME_MEASUREMENTS` Alternativ kann auch die Skriptdatei `incDefs` verwendet werden. Dazu genügt ein Befehl wie: `./incDefs -DVERBOSE -DTIME_MEASUREMENTS` um die Definitionen `VERBOSE` und `TIME_MEASUREMENTS` zu selektieren.

Nachdem alles eingegeben und installiert worden ist, was für das Kompilieren notwendig ist, kann man nun im Verzeichnis von **Poisson2D** den Befehl:
`make all` oder kürzer auch `make` eingeben.
Hinterher muss man es nur noch mit MPI starten. Dies geschieht durch Eingabe von:
`mpirun -np <Anzahl an Prozessen> "<Pfad zum Poisson2D-Ordner>/Poisson2D" <Startparameter wie in Abschnitt A.2.2>`

| Option | Beschreibung |
|---|---|
| COARSE_GRID_SOLVER_IP_SMOOTHER | Löser „in place Glätter “ |
| COARSE_GRID_SOLVER_IP_CG | „In place CG“Löser |
| COARSE_GRID_SOLVER_IP_HYPRE | „in place Hypre/MG “Löser |
| COARSE_GRID_SOLVER_RED_HYPRE | „Reduktion-Hypre/AMG“Löser |
| DEBUG_LSE_SOLVER | Debug-Ausgabe bezüglich des LSE-Lösers |
| MEASURE_MG_COMPONENTS | Ausgabe für verschiedene Mehrgitter-Komponenten |
| SMOOTHER_JACOBI | Jacobi-Glätter |
| SMOOTHER_GS | Gauss-Seidel-Glätter |
| SMOOTHER_GSAC | Gauss-Seidel mit zusätzlicher Kommunikation |
| SMOOTHER_GSOD | Gauss-Seidel-of-Death (zusätzliche Kommunikation) |
| SMOOTHER_GSACBE | Gauss-Seidel Block Edition mit zusätzlicher Kommunikation |
| SMOOTHER_GSRS | Gauss-Seidel mit zufälliger Auswahl |
| SMOOTHER_GSRB | Gauss-Seidel-Rot-Schwarz |
| SMOOTHER_GSRBAC | Gauss-Seidel-Rot-Schwarz mit zusätzlicher Kommunikation |
| TIME_MEASUREMENTS | Zeitmessungen |
| USE_CHRONO | Verwende die `chrono`-Bibliothek von C++ |
| USE_GTOD | Verwende die `ctime`-Bibliothek von C++ |
| VERBOSE | Detailierte Ausgabe |

Tabelle A.1: Definitionen für die Datei CMakeLists.txt

### A.2.2 Konfiguration von HSMGP

Bisher musste man in **HSMGP** die Parameter nach einer fest vorgelegten Reihenfolge angeben. Jetzt können aber durch `program_options` die Programmoptionen auch in beliebiger Reihenfolge angegeben werden. Dabei wurden die Programmoptionen mit Namen versehen wie in Tabelle A.2 aufgelistet.

Durch das Nutzen von `program_options` der **Boost**-Bibliothek erfolgt die Eingabe der Startparameter durch Zuweisungen mittels key-value pair, wie in etwa:
`-maxNumIterations 4`, welches die Zahl 4 der Option `maxNumIterations` zuweist und an das Programm weitergibt. Dadurch hat man einen höheren Grad an Übersicht.
Außerdem besteht durch die `program_options`-Bibliothek die Möglichkeit Konfigurationsdateien zu definieren. Diese Datei muss im HSMGP-Ordner gespeichert sein und `settings.ini` heißen. Dabei ist zu beachten, dass diese Optionen überschrieben werden können. Das heißt, dass man die Optionen, die man in `settings.ini` festgelegt hat, nachträglich durch Eingabe der Option beim Start des Programms ändern kann.
Die Wertezuweisung in der Konfigurationsdatei geschehen anders, wie bei der Angabe der Startparameter. Bei der Konfigurationsdatei wird eine Zuweisung folgender-

| Option | Beschreibung |
|---|---|
| coarseGridRankStride | Standard: 4 |
| coarsestLvl | Das gröbste Level. Standard: 0 |
| cycle | Der auszuführende Zyklus(V,W,F)<br>Standard: V |
| finestLvl | Das feinste Level. Standard: 8 |
| help | Die Ausgabe aller Parameter und deren Beschreibung |
| maxNumIterations | Die Anzahl an Iterationen |
| numBlocksX | Anzahl an Blöcken auf der X-Achse<br>Standard: Automatisch bestimmt |
| numBlocksY | Anzahl an Blöcken auf der Y-Achse<br>Standard: Automatisch bestimmt |
| numCoarseSteps | Anzahl an Vergröberungsschritten<br>Standard: 64 |
| numElementsPerBlockX | Anzahl der Elemente eines Blocks<br>auf der X-Achse. |
| numElementsPerBlockY | Anzahl der Elemente eines Blocks<br>auf der Y-Achse. |
| numElementsPerBlock | Alternative zu numElementsPerBlockX<br>und numElementsPerBlockY<br>Hierbei kann man für quadratische Blöcke eine<br>Zahl angeben oder eben auch 2 Zahlen. |
| numLSELinesPerThread | Anzahl an LSE-Zeilen je Thread<br>Standard: 64 |
| numPreSmoothingSteps | Anzahl an Vorglättungsschritten<br>Standard: 2 |
| numPostSmoothingSteps | Anzahl an Nachglättungsschritten<br>Standard: 2 |
| omega | Der Omega-Wert. Standard: 0.8 |

Tabelle A.2: Die Startoptionen, welche sowohl in die Konfigurationsdatei, als auch beim Start des Programms in der Shell eingegeben werden können.

maßen angegeben: `<Name der Option> = <Wert der Option>`. Während man bei der Angabe als Startparameter einen Vektor so eingeben kann: `-numElementsPerBlock 4 5`, wird dies in der Konfigurationsdatei eine Eingabe, welche sich über mehrere Zeilen erstreckt:

```
numElementsPerBlock = 4
numElementsPerBlock = 5
```

Damit erzielt man das gleiche Verhalten wie bei der Angabe als Startparameter.

### A.2.3 Stoppuhr

Von Sebastian Kuckuk gab es zwei Implementierungen der Stoppuhr. Eine davon ist aus der `chrono`-Bibliothek von C++(wird durch die Definition `USE_STD_CHRONO` aktiviert), die andere war bisher linuxbasiert(wurde durch `USE_GTOD` aktiviert). Letztere

wurde nun durch eine Implementierung aus der `ctime`-Bibliothek. Diese wird nun standardmäßig benutzt, sofern nicht `USE_STD_CHRONO` als Definition festgelegt worden ist. Die Stoppuhr aus der `chrono`-Bibliothek ist genauer als die Stoppuhr aus `ctime`. Die normale Uhr aus `ctime` weicht allerdings ab, wenn man die Ergebnisse von Windows und Linux vergleicht. Dies liegt daran, dass unter Windows die Uhrzeit und unter Linux die CPU-Zeit gemessen wird. Die Intention bei dieser Änderung war vor allem, das Programm auch auf anderen Plattformen unabhängig von den Definitionen lauffähig zu machen.

### A.2.4 Verbesserte Ausgabe bei der Parametereingabe

Durch `program_options` wurde noch die Fehlerausgabe bezüglich der Parametereingabe und die Hilfe verbessert. Bei der Eingabe von `-help` oder bei der Belegung von Optionen mit falschen Werten erscheint die Hilfe, um es dem Nutzer einfacher zu gestalten, die richtigen Optionen zu suchen. Sollte einer Option ein falscher Wert zugewiesen worden sein, so ist in der Regel die letzte Ausgabe vor der Hilfe die Fehlermeldung. Ohne diese wäre es durch das Nutzen von `program_options` schwieriger, den Fehler herauszufinden.
Sollte bei der Eingabe von Werten einer Option ein anderer Typ zugewiesen werden, als den, für den die Option bestimmt ist, so wird dies durch `program_options` automatisch ausgegeben. Dies passiert beispielsweise bei der Eingabe einer Festkommazahl, obwohl eine ganze Zahl erwartet wird.

## A.3 MGS

### A.3.1 Installation von MGS

Um das Programm MGS(**M**ulti**g**rid **s**olver) ausführen zu können, werden zunächst Programme benötigt, um schließlich DUNE installieren zu können. Dies wird im Folgenden anhand des Betriebssystems Ubuntu, Version 14.04 näher erklärt.

- **gcc, g++, gfortran**: Normalerweise sind **gcc**, **g++** und **gfortran** ebenfalls auf Ubuntu vorinstalliert. Sollte dies jedoch nicht der Fall sein, so werden diese durch

  ```
  sudo apt-get install gcc g++ gfortran
  ```

  installiert.

- **make, cmake, automake, autoconf, libtool**: **make** ist auf Ubuntu bereits vorinstalliert. Nur **cmake**, **automake**, **autoconf** und **libtool** müssen für den späteren Kompiliervorgang durch

  ```
  sudo apt-get install cmake automake
  ```

  installiert werden. Sollten hierbei Fehler wie zum Beispiel ein Fehler bezüglich des Pfades `/usr/bin/c++` auftreten, so empfiehlt sich die Ausführung von:

```
sudo apt-get install build-essential
```

- **MPI**: Um das Programm parallel auf mehreren Prozessoren ausführen zu können, wird **MPI** benötigt. Für MGS wird **OpenMPI** empfohlen, da es auch für HSMGP verwendet wird und man so keine andere Bibliothek herunterladen muss. Der Befehl hierzu lautet:

  ```
  sudo apt-get install libopenmpi-dev openmpi-bin
  ```

  Sollte im späteren Verlauf bei der Konfiguration **MPI** nicht gefunden werden, so sollte **OpenMPI** manuell installiert werden.

Nach der Installation der für das Kompilieren nötigen Programme, folgt die Konfiguration des Kompiliervorgangs und das Kompilieren. Für die parallele Ausführung gibt es ein Konfigurationsparameter „–enable-parallel “, womit die erforderlichen Komponenten, wie etwa **MPI** auf dem Computer gesucht und verlinkt werden. Um die Befehle ausführen zu können, muss man sich zunächst im **DUNE**-Ordner befinden. Der Befehl lautet wie folgt:

```
sudo ./dune-common/bin/dunecontrol configure "--enable-parallel"
```

Da hierdurch alle Module entsprechend konfiguriert werden, kann dieser Vorgang etwas länger dauern. Falls man für **MGS** ein oder mehrere Präprozessordefinitionen einsetzen möchte, wurde für die leichtere Erstellung ein Shell-Script geschrieben, womit man durch einen einzigen Befehl die Definitionen hinzufügen kann. Dieses Skript ist ebenfalls auf der CD als Datei mit dem Namen `incDefs` im Ordner `src` von **MGS** vorhanden. Hierzu muss folgender Befehl ausgeführt werden:

```
./multigrid_overlapping/src/incDefs -D<Definition 1> -D<Definition 2>
```

Dabei sind `<Definition 1>` und `<Definition 2>` Platzhalter für die Präprozessordefinitionen. Zudem können auch mehr als 2 Definitionen hinzugefügt werden, indem man weitere `-D<Definition>` dem Befehl hinzufügt. Wenn man beispielsweise die Definitionen `X_CGSolver` und `X_SeqJac` an den Präprozessor weitergeben möchte lautet der Befehl `./multigrid-overlapping/src/incDefs -DX_CGSolver -DX_SeqJac`. Eine Liste aller Präprozessordefinitionen ist durch Tabelle A.3 gegeben.

Nach der Konfiguration folgt das Kompilieren. Mit dem Parameter „-j “wird die Parallelisierung des Vorgangs angestoßen. Dieser Parameter kann für die sequentielle Ausführung auch weggelassen werden. Der Befehl hierzu lautet:

```
sudo ./dune-common/bin/dunecontrol make -j
```

| Definition | Beschreibung |
|---|---|
| TIME_MEASUREMENTS | Anwendung einer Stoppuhr zur Messung der Gesamtzeit |
| X_SeqGS | Sequentieller Gauss-Seidel Vorkonditionierer |
| X_SeqSSOR | Sequentieller symmetrisch-sukzessiver Überrelaxationsalgorithmus |
| X_SeqSOR | Sequentieller sukzessiver Überrelaxationsalgorithmus |
| X_SeqILU | Sequentieller unvollständiger Links-Rechts-Zerlegungs-Vorkonditionierer |
| X_SeqILU0 | Sequentieller unvollständiger kommunikationsvermeidender Links-Rechts-Zerlegungs-Vorkonditionierer |
| X_SeqJac | Sequentieller Jacobi-Vorkonditionierer |
| X_SeqILUn | Verallgemeinerter sequentieller unvollständiger Links-Rechts-Zerlegungs-Vorkonditionierer |
| X_Richardson | Richardson-Vorkonditionierer |
| X_CGSolver | Konjugierter-Gradient-Löser |
| X_BiCGSTABSolver | Stabilisierter Bikonjugierter-Gradient-Löser |
| X_LoopSolver | Schleifen-Löser |
| X_GradientSolver | Gradient-Löser |
| USE_CHRONO | Verwende die `chrono`-Bibliothek von C++ |
| USE_GTOD | Verwende die `ctime`-Bibliothek von C++ |

Tabelle A.3: Definitionen für die Datei CMakeLists.txt

### A.3.2 Konfiguration von MGS

Sobald das Kompilieren abgeschlossen ist, kann das Programm ausgeführt werden. Der dazugehörige Befehl lautet:
`sudo ./multigrid_overlapping/src/multigrid_overlapping <cells per dir> <overlap coarse> <overlap fine> <number of refinements> <number of presmoothing steps> <number of postsmoothing steps> <cycle>`
Bei dem Befehl sind alle Tags[1] zwischen `<>` Programmoptionen, wovon einige obligatorisch sind. Diese werden in Tabelle A.4 näher erklärt.

Die Optionen `overlap coarse` und `overlap fine` sind, wie beschrieben, die Überlappung von mehreren Punkten. In Abbildung A.1 ist die normale Aufteilung eines Gitters ohne überlappende Punkte dargestellt. Erhöht man die sich überlappenden Punkte lediglich um 1, so ergibt sich eine Aufteilung wie in Abbildung A.2. Dabei wird die eigentliche Aufteilung wie in Abbildung A.1 noch beibehalten. Allerdings werden die sich überschneidenden Gitterpunkte von einem Prozess zum Anderen übertragen, damit dieser mit diesen Gitterpunkten das weitere Verfahren anwenden kann.

[1] Anhänger von zusätzlichen Informationen

| Option | Beschreibung |
|---|---|
| cells per dir | Die Größe des Gitters. Das Gitter ist dabei immer quadratisch. (Obligatorisch) |
| overlap coarse | Die sich überlappenden Punkte im gröberen Gitter. (Obligatorisch) |
| overlap fine | Die sich überlappenden Punkte im feineren Gitter. (Obligatorisch) |
| number of refinements | Anzahl an Verfeinerungen. Dadurch wird die Anzahl der Levels bestimmt. (Obligatorisch) |
| number of presmoothing steps | Anzahl an Vorglättungsschritten. Standard: 2 (Optional) |
| number of postsmoothing steps | Anzahl an Nachglättungsschritten. Standard: 2 (Optional) |
| cycle | Die Zyklenart (V, W, F). Standard: V. (Optional) |

Tabelle A.4: Definitionen für die Datei CMakeLists.txt

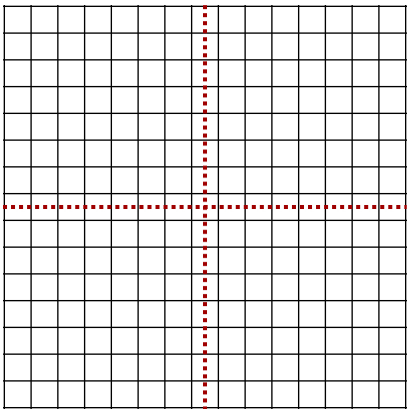


Abbildung A.1: Aufteilung eines Gitters auf 4 Prozesse ohne überlappende Punkte.

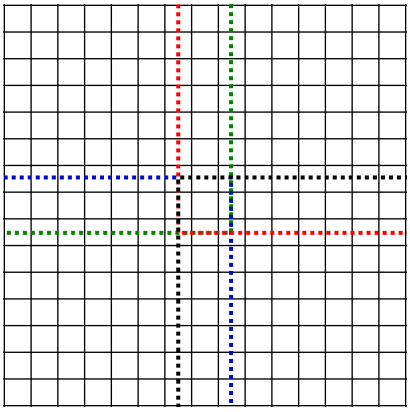


Abbildung A.2: Aufteilung des Gitters auf 4 Prozessen mit sich überlappenden Punkten.

# Literaturverzeichnis


[ABHT03] Mark F. Adams, Marian Brezina, Jonathan Hu, and Ray Tuminaro. Parallel Multigrid Smoothing: Polynomial Versus Gauss–Seidel. *Journal of Computational Physics*, 188(2):593–610, July 2003. (zitiert auf Seite 14)

[Ada01] Mark F. Adams. A Distributed Memory Unstructured Gauss-seidel Algorithm for Multigrid Smoothers. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 1–14. ACM, 2001. (zitiert auf Seite 14)

[Alt13] Mirco Altenbernd. Numerischer Vergleich nichtlinearer und linearer Mehrgitterverfahren zum Lösen von Partiellen Differentialgleichungen. Master's thesis, TU Dortmund, 2013. (zitiert auf Seite 15)

[BB07] Markus Blatt and Peter Bastian. The Iterative Solver Template Library. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors, *Proceedings of Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 666–675. Springer, 2007. (zitiert auf Seite 29)

[BB08] Markus Blatt and Peter Bastian. On the Generic Parallelisation of Iterative Solvers for the Finite Element Method. *International Journal of Computer Science Engineering*, 4(1):56–69, 2008. (zitiert auf Seite 29)

[BBD+08a] Peter Bastian, Markus Blatt, Andreas Dedner, Christian Engwer, Robert Klöfkorn, Mario Ohlberger, and Oliver Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE. *Computing*, 82(2–3):121–138, 2008. (zitiert auf Seite 29)

[BBD+08b] Peter Bastian, Markus Blatt, Andreas Dedner, Christian Engwer, Markus Ohlberger, and Oliver Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. *Computing*, 82(2–3):103–119, 2008. (zitiert auf Seite 29)

[BBD+11] Peter Bastian, Markus Blatt, Andreas Dedner, Christian Engwer, Carsten Gräser, Robert Klöfkorn, Mario Ohlberger, and Oliver Sander. DUNE Web page, 2011. http://www.dune-project.org. (zitiert auf Seite 29)

[Buc10] Sean Buckeridge. *Numerical solution of weather and climate systems*. PhD thesis, University of Bath, 2010. (zitiert auf Seite 1)

[Dat09] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, 2009. (zitiert auf Seite 9 und 14)

[Dor97] Thomas Dornseifer. *Diskretisierung allgemeiner elliptischer Differentialgleichungen in krummlinigen Koordinatensystemen auf dünnen Gittern*. PhD thesis, Universität München, 1997. (zitiert auf Seite 6)

[GKS+14] Alexander Grebhahn, Sebastian Kuckuk, Christian Schmitt, Harald Köstler, Norbert Siegmund, Sven Apel, Frank Hannig, and Jürgen Teich. Optimizing Performance of Stencil Code with SPL Conqueror. In *Proceedings of the Parallel Processing Letters*, 2014. To appear. (zitiert auf Seite 38)

[HPRW04] Müller Hannes, Nicole Piffer, Barbara Raschke, and Melanie Wogrin. Die Wärmeleitungsgleichung Projekt aus Partielle Differentialgleichungen. 2004. (zitiert auf Seite 1)

[HPS12] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320. ACM, 2012. (zitiert auf Seite 14)

[hyp01] Hypre - High Performance Preconditioners User's Manual. Technical report, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2001. (zitiert auf Seite 28)

[Jen11] Thomas Jenni. Einführung in die Finite-Elemente-Methode. 2011. (zitiert auf Seite 10)

[Kam05] Masanori Kameyama. ACuTEMan: A multigrid-based mantle convection simulation code and its optimization to the Earth Simulator. *Journal Earth Simulator*, 4:2–10, 2005. (zitiert auf Seite 1)

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC Document, 1990. (zitiert auf Seite 24)

[KGKR13] Sebastian Kuckuk, Björn Gmeiner, Harald Köstler, and Ulrich Rüde. A Generic Prototype to Benchmark Algorithms and Data Structures for Hierarchical Hybrid Grids. In *Proceedings of the PARCO*, pages 813–822, 2013. (zitiert auf Seite 30 und 47)

[MST10] Aleka A. McAdams, Eftychios Sifakis, and Joseph Teran. A Parallel Multigrid Poisson Solver for Fluids Simulation on Large Grids. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on*

*Computer Animation*, SCA '10, pages 65–74. Eurographics Association, 2010. (zitiert auf Seite 10)

[RYQ11] Shah M. Faizur Rahman, Qing Yi, and Apan Qasem. Understanding Stencil Code Performance on Multicore Architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 30:1–30:10. ACM, 2011. (zitiert auf Seite 14)

[TS01] Ulrich Trottenberg and Anton Schuller. *Multigrid*. Academic Press, Inc., 2001. (zitiert auf Seite 5, 9, 13, 17, 19 und 20)

[ZC06] Yu Zhu and Andreas C Cangellaris. *Multigrid Finite Element Methods For Electromagnetic Field Modeling*, volume 28. John Wiley & Sons, 2006. (zitiert auf Seite 7)

**Eidesstattliche Erklärung:**

Hiermit versichere ich an Eides statt, dass ich diese Bachelorarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Christian Kaltenecker

Passau, den 28. September 2014