Fachhochschule Aachen
Campus Jülich
Fachbereich 09
Medizintechnik und Technomathematik

Untersuchung des Rot-Schwarz Gauß-Seidel-Verfahrens für Diffusionsprobleme bezüglich Parallelisierung auf einer GPU

Bachelorarbeit

im Bachelor Studiengang
Scientific Programming

geschrieben von
My Linh Würzburger

Jülich, Oktober 2016

Selbstständigkeitserklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Name: My Linh Würzburger

.....

Unterschrift

Erstprüfer: Prof. Dr. rer. nat. Matthias Grajewski
Zweitprüfer: Anne Severt

Die Arbeit wurde angefertigt in der Forschungszentrum Jülich GmbH im Jülich Supercomputing Center (JSC).

Inhaltsverzeichnis

1. Einleitung

Die Parallelisierung des Gauß-Seidel-Verfahrens für Diffusionsprobleme auf einer GPU wird im Rahmen von JuROr, einem Projekt des Forschungszentrums Jülich, entwickelt.

1.1. Motivation für das Rot-Schwarz Gauß-Seidel Verfahren

Das Gauß-Seidel-Verfahren, auch bekannt als **Einschrittverfahren**, ist ein spezielles Splitting-Verfahren aus der numerischen Mathematik. Es handelt sich hierbei um einen iterativen Löser für lineare Gleichungssysteme (LGS). Ein weiteres bekanntes Splitting-Verfahren ist das Jacobi-Verfahren oder auch Mehrschrittverfahren, welches bisher in JuROr eingesetzt wurde.
Ziel dieser Bachelorarbeit ist es mit Hilfe des Gauß-Seidel-Verfahrens eine schnellere Konvergenz zu erreichen und durch die Variante des Rot-Schwarz Gauß-Seidel-Verfahrens die Möglichkeit der Parallelisierung auf der GPU[1] zu eröffnen. Dafür werden im Nachfolgenden (Kapitel 2) das Gauß-Seidel und das Jacobi-Verfahren vorgestellt und die Parallelisierungsmöglichkeiten für das Gauß-Seidel-Verfahren erklärt (Kapitel 3). Anschließend folgt in Kapitel 4 die praktische Umsetzung innerhalb von JuROr und eine Analyse über die Effizienz in diesem Rahmen (Kapitel 5).

1.2. Vorstellung: JuROr

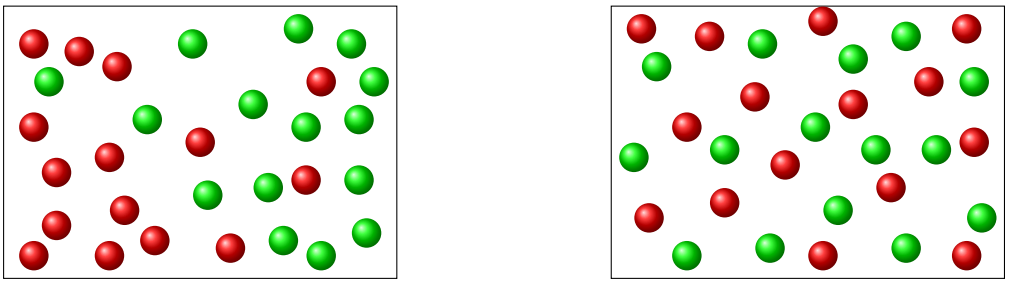
JuROr (Jülich's Real-time simulation within ORPHEUS[2]) dient der numerischen Berechung der Rauchentwicklung in komplexen Geometrien. Innerhalb dieser Software gibt es einen Löser für die Navier-Stokes-Gleichung und diesem unterstehend einen Advektions-, Diffusions- und Drucklöser. Zum Zeitpunkt der Entstehung dieser Bachelorarbeit ist der Diffusionslöser mit dem Jacobi-Verfahren implementiert worden.
Die Berechnungen werden auf die GPU ausgelagert auf Grund ihrer Spezialisierung auf parallelisierbare Aufgaben in denen sie der CPU bezüglich der Rechenleistung überlegen ist. Zusätzlich findet eine Entlastung der CPU statt und ermöglicht rechenintensive

[1] Graphics Processing Unit (z. Dt. Grafikprozessor)
[2] Projekt ORPHEUS (Optimierung der Rauchableitung und Personenführung in U-Bahnhöfen: Experimente und Simulationen) ist ein vom BMBF (Bundesministerium für Bildung und Forschung) gefördertes Projekt, dessen Ziel es ist, die Personensicherheit in unterirdischen Verkehrsanlagen im Brandfall zu verbessern. Weitere Informationen unter: `http://www.orpheus-projekt.de/` [16].

Simulationen auf Computern ohne spezielle Voraussetzungen. Zur Kompilierung auf der GPU wird der pgi-Compiler benötigt.
Die Parallelisierung findet mit OpenACC statt. OpenACC ist nutzbar für die Programmiersprachen Fortran, C und C++ und läuft auf einer Vielzahl von Graphik Prozessoren (weitere Informationen unter `www.openacc.org`).

1.3. Diffusion

Diffusion ist ein freiwillig (passiv) ablaufender physikalischer Prozess, welcher die Durchmischung mehrerer unterschiedlicher Stoffe beschreibt. In der Regel handelt es sich um gasförmige oder flüssige Stoffe. Prinzipiell findet Diffusion in allen Aggregatzuständen statt, in Festkörpern allerdings nur sehr langsam. Diffusion kann aufgrund von Konzentrations-, Druck- und Temperaturunterschieden auftreten und findet durch die molekulare Wärmebewegung bis zum Abbau des Konzentrationsgradienten (Konzentrationsunterschied) statt.
Die Brown'sche Molekularbewegung beschreibt die Bewegung von Teilchen auf einer bestimmten Strecke in eine zufällige Richtung. Bedingt durch Wechselwirkungen mit anderen Teilchen kommt es zu ständigen Richtungswechseln. Liegt ein Konzentrationsgradient vor, ist die Wahrscheinlichkeit für alle Bewegungsrichtungen nicht mehr gleich verteilt. Es entsteht ein so genannter Nettofluss, bis sich das thermodynamische Gleichgewicht einstellt, welches in der Regel eine Gleichverteilung der Konzentration der verschiedenen Teilchen ist. Ein solches System, in dem die Teilchen über das gesamte Volumen verteilt sind, hat eine höhere Entropie[1] als ein geordnetes System in dem sich die Teilchen in bestimmten Bereichen aufhalten (siehe Abbildung 1.1). Nach dem zweiten Hauptsatz der Thermodynamik, welcher das Prinzip der Irreversibilität von Prozessen[2] beschreibt, kann Entropie nicht abnehmen [Tip00].



(a) Niedrige Entropie **(b)** Hohe Entropie

Abbildung 1.1.: Entropie

[1] Entropie (griechisch von en = innerhalb und trope = Wendung, Umkehr) ist eine thermodynamische Zustandsfunktion. Umgangssprachlich mit „Maß für Unordnung“ beschrieben oder auch als „Maß für die Unkenntnis des atomaren Zustandes“.
[2] Prozesse, welche nur in einer Richtung ablaufen, wie zum Beispiel das Zerspringen eines Glases beim Aufprall.

Ein anschauliches Beispiel aus dem Alltag ist Rauch. Wenn beispielsweise eine Kerze oder ein Raucher Rauchschwaden produziert, verfliegt der Rauch nicht in einer Linie, sondern beginnt sich mit der Luft zu vermischen. Der Rauch teilt sich in verschiedene Strömungen. Das Volumen (in dem sich der Rauch verteilt) steigt somit an, während die Rauchkonzentration fällt. Äquivalent verhält sich das Beispiel mit einem Tropfen Tinte, der in ein Glas Wasser fällt. Veranschaulicht ist es in Abbildung 1.2 zu sehen. Dieser Prozess ist irreversibel, eine Umkehrung ist nicht möglich. Dies bedeutet, dass sich die Tinte nicht ohne äußere Einwirkungen vom Wasser trennen wird.

Abbildung 1.2.: Die Tinte vermischt sich langsam mit dem Wasser (Bild 2-3), bis am Ende nur noch eine homogene Mischung (Bild 4) besteht

Ein ebenfalls irreversibler Prozess ist die Wärmeleitung. Die Wärmeleitgleichung, welche auch bekannt ist als **Diffusionsgleichung**, beschreibt die zeitliche Veränderung der Temperatur oder auch der Geschwindigkeit [Wür16]. Eine Unterscheidung des Diffusions- und des Wärmeübertragungsvorgangs ist nicht nötig, da beide äquivalenten Gesetzesmäßigkeiten folgen. Die molekulare Wärmeleitung entspricht der molekularen Diffusion und die konvektive Wärmeübertragung der konvektiven Stoffübertragung. Lediglich die Wärmestrahlung besitzt kein derartiges Analogon.
Im Bereich der irreversiblen Thermodynamik werden die linearen Zusammenhänge zwischen Flüssen und ihren korrespondierenden Kräften beschrieben durch Naturgesetze wie dem ohm'schen Gesetz (Stromfluss), dem Fick'schen Gesetz (Diffusion) und dem Fourier'schen Gesetz (Wärmeleitung).
Das erste Fick'sche Gesetz beschreibt die Proportionalität zwischen dem Diffusionsstrom und dem Kozentrationsgradienten in einem ruhenden Medium:

$$\vec{J} = -D \cdot \nabla C \tag{1.3.1}$$

Hierbei ist J der Diffusionsstrom (Stoffmenge, welche in einer Zeiteinheit durch eine Flächeneinheit tritt), C die Stoffmengenkonzentration des diffundierenden Stoffes und

D der Diffusionskoeffizient. Ist D konstant, ergibt sich die Wärmeleitgleichung, dessen Analogon das Fourier'sche Gesetz (1.3.2) ist: [FK59].

$$\vec{q} = -\lambda \cdot \nabla T \tag{1.3.2}$$

q entspricht der Wärmestromdichte, λ der Wärmeleitfähigkeit und ∇T ist der Temperaturgradient. In Anbetracht der Analogie zwischen den beiden Vorgängen ist mathematisch gesehen keine Unterscheidung nötig.
Aus dem ersten Fick'schen Gesetz lässt sich das zweite Gesetz ableiten. Mit Hilfe der Kontinuitätsgleichung (Massenerhaltung)

$$\frac{\partial c}{\partial t} = -\nabla J \tag{1.3.3}$$

ergibt sich die Diffusionsgleichung:

$$\frac{\partial c}{\partial t} = \nabla \cdot (D \cdot \nabla C)\,. \tag{1.3.4}$$

Für einen konstanten Diffusionskoeffizienten D folgt:

$$\frac{\partial c}{\partial t} = D \cdot \nabla^2 C. \tag{1.3.5}$$

Es stellt die Beziehung zwischen den örtlichen und zeitlichen Konzentrationsunterschieden dar und beschreibt wie sich durch Diffusion die Konzentration mit der Zeit ändert. Wie beim ersten Fick'schen Gesetz ist mathematisch gesehen die Diffusionsgleichung identisch mit der Wärmeleitgleichung. Bei den Gleichungen handelt es sich um partielle Differentialgleichungen[1] (PDE - partial differential equation).

[1]Partielle Differentialgleichungen enthalten Ableitungen einer Funktion mit mehreren Variablen.

2. Theorie

Bei Näherungsverfahren für PDEs müssen häufig große, schwach besetzte lineare Gleichungssysteme (LGS) gelöst werden. Ein Gleichungssystem wird schwach oder dünn besetzt genannt, wenn die Matrix vorwiegend aus Nulleinträgen besteht beziehungsweise die Anzahl der Nichtnulleinträge gering ist (deutlich unter 1%) [RG15]. Oftmals werden diese wie im Englischen als **Sparse Matrix** bezeichnet (äquivalent verhält es sich mit dem Sparse Vektor). Zur Speicherung gibt es spezielle Datenformate wie CSR (Compressed Sparse Row) oder CSC (Compressed Sparse Column), welche weniger Speicherplatz benötigen.
Iterative Methoden haben bei schwach besetzten LGS wesentliche Vorteile, da zum einen die Matrix-Vektor-Multiplikation optimiert werden kann durch die Einsparung von Rechenoperationen, wenn ein Operand Null entspricht, und zum anderen das speichereffiziente Speichern der Matrix. Während direkte Löser mit jedem Schritt eine neue Koeffizientenmatrix bilden, verändert ein iteratives Verfahren die Matrix nicht. Zusätzlich ist im Regelfall die Inverse und die LU-Zerlegung einer Sparse Matrix vollbesetzt mit Ausnahme von Bandmatrizen[1].

2.1. Iterative Verfahren

Ein iteratives Verfahren wird durch seine Iterationsvorschrift beschrieben. Auf der einen Seite befinden sich die Iterierten im neuen Schritt und auf der anderen Seite die Funktionsvorschrift, welche die bereits errechneten Iterierten einbezieht. Durch mehrmalige Wiederholung der Iterationsvorschrift wird sich der Lösung genähert.
Ausgehend von einem LGS $A \cdot \vec{x} = \vec{b}$ der Form

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{2.1.1}$$

mit einer $(n \times n)$-Matrix A, der rechten Seite $\vec{b}$ und dem gesuchten Lösungsvektor $\vec{x}$,

[1] Bei einer Bandmatrix ist die Hauptdiagonale und eine bestimmte Anzahl an Nebendiagonalen besetzt. Sind zusätzlich die untere und obere Nebendiagonale besetzt, handelt es sich um eine Tridiagonalmatrix.

ergibt sich die elementweise Schreibweise

$$\sum_{j=1}^{n} a_{ij}\, x_j = b_i \qquad \text{mit } i = 1, ..., n. \tag{2.1.2}$$

Zur Lösung des LGS wird nun jede i-te Zeile nach x_i aufgelöst:

$$\sum_{j=1}^{n} a_{ij}\, x_j = b_i \tag{2.1.3}$$

$$\iff \sum_{j=1}^{i-1} a_{ij}\, x_j + a_{ii}\, x_i + \sum_{j=i+1}^{n} a_{ij}\, x_j = b_i \qquad | \; a_{ii} \neq 0 \tag{2.1.4}$$

$$\iff x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j - \sum_{j=i+1}^{n} a_{ij}\, x_j \right) \tag{2.1.5}$$

Wie bereits erwähnt wird bei einem Iterationsverfahren das neue $\vec{x}^{(k+1)}$ aus dem vorherigen, berechneten Vektor $\vec{x}^{(k)}$ errechnet, womit eine Anfangsschätzung $\vec{x}^{(0)}$ benötigt wird.

2.2. Jacobi-Verfahren

Das Jacobi-Verfahren berechnet den neuen Vektor $\vec{x}^{(k+1)}$ basierend auf den Komponenten von $\vec{x}^{(k)}$, womit sich die Vorschrift

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{(k)} \right) \qquad \text{mit } i = 1, \ldots, n; a_{ii} \neq 0 \tag{2.2.1}$$

ergibt. Alternativ lässt sich das Jacobi-Verfahren in Matrixschreibweise ausdrücken. Die Matrix A wird zerlegt in eine Diagonalmatrix D, einer unteren Dreiecksmatrix L (lower) und einer oberen Dreiecksmatrix U (upper), so dass gilt:

$$A = L + D + U\,. \tag{2.2.2}$$

Angewandt auf die Iterationsvorschrift (2.2.1), stellt die folgende Vorschrift entsprechend den ganzen Vektor $\vec{x}^{(k+1)}$ dar:

$$\vec{x}^{(k+1)} = D^{-1} \cdot \left(b - (L + U) \cdot \vec{x}^{(k)} \right) \tag{2.2.3}$$

oder alternativ

$$\vec{x}^{(k+1)} = D^{-1} \cdot \left(b - (A - D) \cdot \vec{x}^{(k)} \right). \tag{2.2.4}$$

Da die Berechnung der Komponenten eines Iterationsschrittes durch die Verwendung des vorher berechneten Vektors $\vec{x}^{k}$ voneinander unabhängig sind, eignet sich das Jacobi-Verfahren zur Parallelisierung im Gegensatz zum Gauß-Seidel-Verfahren.

2.3. Gauß-Seidel-Verfahren

Das Gauß-Seidel-Verfahren besitzt eine ähnliche Iterationsvorschrift wie das Jacobi-Verfahren mit dem Unterschied, dass die bereits berechneten Komponenten des aktuellen Iterationschrittes $\vec{x}^{(k+1)}$ anstelle der Komponenten des letzten Iterationschrittes genutzt werden, womit sich folgende Iterationsvorschrift ergibt:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{(k)} \right) \qquad \text{mit } i = 1, \ldots, n; a_{ii} \neq 0 \tag{2.3.1}$$

Für die Matrixschreibweise wird A wie in (2.2.2) zerlegt, womit sich

$$\vec{x}^{(k+1)} = D^{-1} \cdot \left(\vec{b} - L \cdot \vec{x}^{(k+1)} - U \cdot \vec{x}^{(k)} \right) \tag{2.3.2}$$

$$\iff \qquad D \cdot \vec{x}^{(k+1)} + L \cdot \vec{x}^{(k+1)} = \vec{b} - U \cdot \vec{x}^{(k)} \tag{2.3.3}$$

$$\iff \qquad \vec{x}^{(k+1)} = (D + L)^{-1} \cdot \left(\vec{b} - U \cdot \vec{x}^{(k)} \right) \tag{2.3.4}$$

ergibt. Beim Gauß-Seidel-Verfahren benötigt jeder Schritt das Ergebnis aus der aktuellen Iteration und somit ist das Verfahren inhährent sequentiell. Das Gauß-Seidel Verfahren ist ein Einzelschrittverfahren, da bei jedem Schritt die bereits neu erhaltenen Iterationswerte verwendet werden im Gegensatz zum Jacobi-Verfahren, welcher zu den Mehrschrittverfahren gehört. Diese nutzen Informationen aus den zuvor berechneten Iterationen. Das Jacobi-Verfahren besitzt somit keine Abhängigkeiten innerhalb eines Iterationschrittes und ist entsprechend parallelisierbar. Anders als das Gauß-Seidel-Verfahren, welches durch seine sukzessive Ersetzung nicht parallelisierbar ist. Um dennoch eine Parallelisierung zu ermöglichen, gibt es Varianten wie das Rot-Schwarz Gauß-Seidel Verfahren, welches in Kapitel 3 behandelt wird.

2.4. SOR-Verfahren

Das **S**uccessive **O**ver-**R**elaxation-Verfahren ist eine Weiterentwicklung des Gauß-Seidel-Verfahrens mit der Iterationsvorschrift

$$x_i^{(k+1)} = (1 - \omega) \cdot x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{(k)} \right) \tag{2.4.1}$$

mit $i = 1, \ldots, n$ und $a_{ii} \neq 0$. Der reelle Relaxationsparameter ω beeinflusst die Konvergenzeigenschaften der Iteration. Für $\omega < 1$ liegt eine Unterrelaxation und für $\omega > 1$ eine Überrelaxation vor, bei $\omega = 1$ ergibt sich das Gauß-Seidel-Verfahren. Das Verfahren ist konvergent für $\omega \in (0, 2)$ [RW12].

2.5. Allgemeine Iterationsvorschrift

Allgemein lassen sich die drei Verfahren in der Form

$$\vec{x}^{(k+1)} = M \cdot \vec{x}^{(k)} + N \cdot \vec{b} \tag{2.5.1}$$

darstellen. Mit

$$M_J = -D^{-1} \cdot (b - (L + U), \quad N_J = D^{-1} \tag{2.5.2}$$

ergibt sich das Jacobi Verfahren (2.2.3). Äquivalent dazu lassen sich M_{GS}, N_{GS} für das Gauß-Seidel-Verfahren und M_{SOR}, N_{SOR} für das SOR-Verfahren bestimmen wie in Tabelle 2.1 dargestellt.

Tabelle 2.1.: Matrizen vom Jacobi-, Gauß-Seidel-, und SOR-Verfahren für die allgemeine Iterationsvorschrift $\vec{x}^{(k+1)} = M \cdot \vec{x}^{(k)} + N \cdot \vec{b}$ mit $A = L + D + U$

Verfahren	**M**	**N**
Jacobi-Verfahren	$M_J = -D^{-1} \cdot (b - (L + U)$	$N_J = D^{-1}$
Gauß-Seidel-Verfahren	$M_{GS} = (L + D)^{-1} \cdot U$	$N_{GS} = (L + D)^{-1}$
SOR-Verfahren	$M_{SOR} = (\omega L + D)^{-1} \cdot ((\omega - 1) \cdot D + \omega U)$	$N_{SOR} = \omega \cdot (\omega L + D)^{-1}$

2.6. Konvergenz

Mit Hilfe der allgemeinen Schreibweise (2.5.1) lassen sich Konvergenzaussagen herleiten. Der Banach'sche Fixpunktsatz [RG15] besagt, dass für jeden Startvektor $\vec{x}^{(0)}$ und jede rechte Seite $\vec{b}$ die Iteration konvergiert, wenn gilt

$$\rho(M) < 1 \qquad \text{mit } \rho(M) = \max_{i=1,\dots,n} |\lambda_i(M)| \tag{2.6.1}$$

wobei $\rho(M)$ den Spektralradius[1] bezeichnet und λ_i den i-ten Eigenwert der Matrix. Davon ableiten lassen sich folgende Aussagen (siehe auch Satz von Ostriwski und Reich) [Her11]:

- Ist A symmetrisch positiv definit konvergieren das Gauß-Seidel- und das SOR-Verfahren. Letzteres lediglich für $0 < \omega < 2$
- Ist A strikt oder irreduzibel[2] diagonal dominant, dann konvergieren das Jacobi- und das Gauß-Seidel-Verfahren.

[1] Das Spektrum eines Operators ist in einer Kreisscheibe enthalten, dessen Radius dem Spektralradius entspricht.

[2] Eine Matrix ist irreduzibel (aus dem Lateinischen: ir- (in-) = un-, nicht und reducere = zurückführen), wenn sie nicht durch Permutationen auf eine untere Blockdreiecksgestalt überführt werden kann.

Auf das SOR-Verfahren wird nachfolgend nicht weiter eingegangen und es wird lediglich das Gauß-Seidel-Verfahren mit dem Jacobi-Verfaren verglichen.
Ausgehend davon, dass das Gauß-Seidel-Verfahren die bereits berechneten Komponenten mit einfließen lässt, scheint es ein verbessertes Jacobi-Verfahren zu sein und eine schnellere Konvergenz wäre zu erwarten, doch wird der Spezialfall im Satz von Stein und Rosenberg betrachtet:

Satz von Stein und Rosenberg: Gelte $A \in M_n$ mit $a_{ij} > 0$ für $i = j$ und $a_{ij} \leq 0$ für $i \neq j$ $(i, j = 1, .., n)$, dann gilt für die Verfahren Jacobi und Gauß-Seidel genau eine der folgenden Beziehungen [Hac91]:

- $\rho(M_{GS}) = \rho\,(M_J) = 0$
- $0 < \rho(M_{GS}) < \rho(M_J) < 1$
- $\rho(M_{GS}) = \rho(M_J) = 1$
- $\rho(M_{GS}) > \rho(M_J) > 1.$

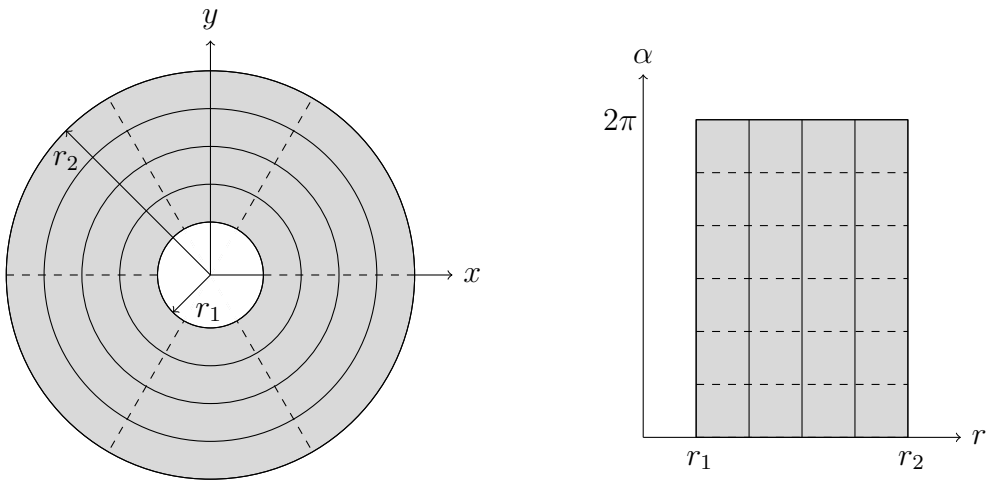
Es ist zu erkennen, dass unter den gesetzten Bedingungen das Jacobi- und das Gauß-Seidel-Verfahren gemeinsam konvergieren oder divergieren, aber das Gauß-Seidel-Verfahren nicht immer schneller konvergiert als das Jacobi-Verfahren. Für ein allgemeines A lässt sich keine explizite Aussage treffen, da es Beispiele für

- Jacobi konvergiert, Gauß-Seidel nicht
- Gaus-Seidel konvergiert, Jacobi nicht
- Jacobi konvergiert schneller als Gauß-Seidel
- Gauß-Seidel konvergiert schneller als Jacobi

gibt [RG15]. Entsprechend wird der abschließende Vergleich zwischen dem Jacobi- und dem parallelisierten Gauß-Seidel-Verfahren in Kapitel 5.2 sich nur auf das zu lösende Problem in JuROr beziehen. Für den Vergleich der beiden Verfahren werden im nachfolgenden Kapitel die Parallelisierungsmöglichkeiten des Gauß-Seidel-Verfahrens untersucht.

3. Parallelisierung des Gauß-Seidel-Verfahrens

Es gibt mehrere Möglichkeiten eine Parallelisierbarkeit des Gauß-Seidel-Verfahrens (nachfolgend als GS beziechnet) zu erreichen. Für eine dünnbesetzte Matrix A aus dem LGS $A \cdot \vec{x} = \vec{b}$ werden wie bereits erwähnt nicht alle Werte zur Berechnung benötigt. Gelte $a_{ij} = 0$ mit $j < i$, so fällt der Term $a_{ij}\, x_j^{(k+1)}$ weg. Dadurch besteht keine Abhängigkeit mehr zu dieser Komponente und $x_i^{(k+1)}$ kann ohne Kenntnis von $x_j^{(k+1)}$ berechnet werden. Äquivalent gilt es für den Fall $j > i$ mit $x_j^{(k)}$.
Dünnbesetze Matrizen entstehen zum Beispiel bei der Diskretisierung von PDEs. Im Nachfolgenden sei Ω das räumliche Gebiet, welches eine rechteckige Fläche sein soll. Für komplexere Gebiete werden spezielle Methoden benötigt, um die entsprechenden PDEs zu diskretisieren. In der numerischen Strömungssimulation können die komplexen Gebiete mit Hilfe von numerischen Methoden zur Gittererzeugung in ein Rechteck (2D) oder einen Kubus (3D) überführt werden. Ein einfaches Beispiel für eine Transformation ist die Überführung in Polarkoordinaten. Liegt zum Beispiel ein Kreisring vor, ergibt die Transformation in Polarkoordinaten ein Rechteck (vgl. Abbildung 3.1).



(a) Kartesisches Koordinatensystem **(b)** Polarkoordinatensystem

Abbildung 3.1.: Polarkoordinatentransformation bildet Kreisring 3.1a auf Rechteck 3.1b ab [Zhu94]

Zur Einführung in die Parallelisierung des GS wird exemplarisch die Poisson-Gleichung $\nabla^2 u = f$ im zweidimensionalen Raum genommen. $\vec{u}$ ist hierbei die unbekannte Funktion.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \qquad (x,y) \in \Omega \tag{3.0.1}$$

Zum Lösen dieser Gleichung wird das Gebiet Ω mit Hilfe der Finiten Differenzen Methode (FDM) auf einem $m \times n$ Gitter diskretisiert. FDM ist ein numerisches Verfahren zur Lösung von gewöhnlichen und partiellen Differentialgleichungen (DGL). Die Ableitungen werden mit Hilfe von Differenzenquotienten gelöst. In der Praxis werden verschiedene Varianten verwendet: Vorwärts- (3.0.2), Rückwärts- (3.0.3) und zentraler Differenzenquotient (3.0.4).

$$f'(x) = \frac{f(x+\Delta x) - f(x)}{\Delta x} + \mathcal{O}(\Delta x) \tag{3.0.2}$$

$$f'(x) = \frac{f(x) - f(x-\Delta x)}{\Delta x} + \mathcal{O}(\Delta x) \tag{3.0.3}$$

$$f'(x) = \frac{f(x+\Delta x) - f(x-\Delta x)}{2 \cdot \Delta x} + \mathcal{O}(\Delta x^2) \tag{3.0.4}$$

Zur Diskretisierung der Poisson-Gleichung (3.0.1) wird der zentrale Differenzenquotient genutzt. Daraus folgend ergibt sich für die inneren Punkte:

$$f_{i,j} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} \tag{3.0.5}$$

und für ein äquidistantes Gitter mit $\Delta x = \Delta y = h$ folgt:

$$f_{i,j} = \frac{u_{i,j-1} + u_{i-1,j} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1}}{h^2} \tag{3.0.6}$$

Daraus resultierend wird ein lineares System $A \cdot u = b$, der Größe $mn \times mn$, mit

$$\vec{u} = [u_{0,0}, u_{0,1}, \ldots, u_{0,n-1}, u_{1,0}, u_{1,1}, \ldots, u_{1,n-1}, \ldots u_{m-1,n-1}]^T \tag{3.0.7}$$

$$\vec{b} = h^2 \cdot [f_{0,0}, f_{0,1}, \ldots, f_{0,n-1}, f_{1,0}, f_{1,1}, \ldots, f_{1,n-1}, \ldots f_{m-1,n-1}]^T \tag{3.0.8}$$

aufgebaut.

Obgleich die in Kapitel 2 aufgeführten Iterationsvorschriften die Indizierung mit 1 beginnen, wird im folgenden bei 0 angefangen als Anpassung an die Architektur von C++ in der die Indizierung bereits bei 0 startet.
Die Besetzungsstruktur für A lässt sich für ein 5×4 Gitter wie folgt darstellen (eine schematische Darstellung der Matrix A ist in Abbildung 3.3 zu finden):

$$A = \begin{pmatrix} D & -I & & & \\ -I & D & -I & & \\ & -I & D & -I & \\ & & -I & D & -I \\ & & & -I & D \end{pmatrix} \quad \text{mit } D = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & -1 & \\ & -1 & 4 & -1 \\ & & -1 & 4 \end{pmatrix} \tag{3.0.9}$$

In D und I (Einheitsmatrix) werden alle zu berechnenden Werte zusammengefasst. Ausgehend davon, dass lediglich die Randwerte fest vorgegeben sind und es alle anderen Werte zu berechnen gilt, ergibt sich ein Gleichungssystem der Größe $(m-2)(n-2) \times (m-2)(n-2)$. Im Nachfolgenden wird das Gitter als Auschnitt eines größeren Gebietes angesehen alle Werte variabel, das heißt zu berechnen sind. Das vollständige Beispiel für ein 5×4 Gitter mit festen Randwerten ist in Anhang A zu finden.

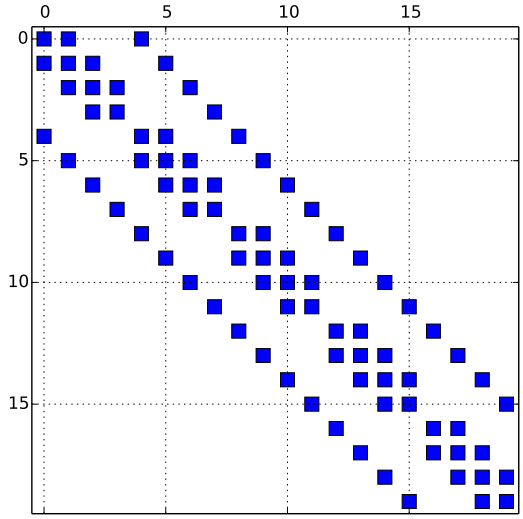
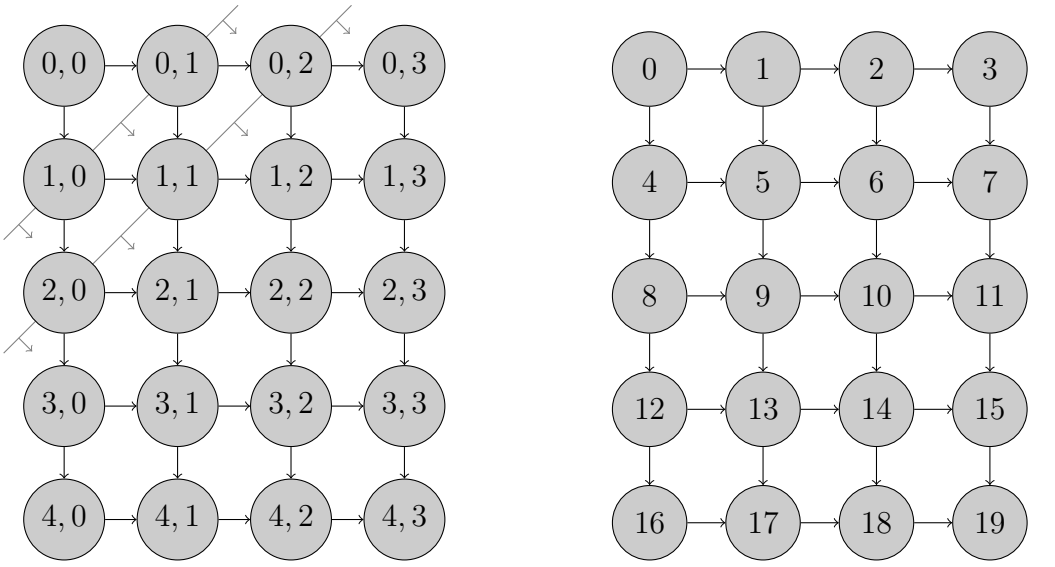


Abbildung 3.3.: Schematische Darstellung von A

3.1. Wellenfront Anordnung

Bei Betrachtung der Matrix A (siehe Abbildung 3.3), ist zu erkennen, dass nach der Berechnung von $u_{0,0}$ zusätzlich zu $u_{1,0}$ auch noch $u_{0,1}$ berechnet werden kann. Im nächsten Schritt können entsprechend $u_{1,1}$, $u_{2,0}$ und $u_{0,2}$ bestimmt werden. Bei der 2-Indizes-Anordnung handelt es sich um die Komponenten, deren Indizes die gleiche Summe haben. Die Datenabhängigkeiten sind in Abbildung 3.4 dargestellt.



(a) Nummerierung mit 2 Indizes **(b)** Lexikographische Nummerierung

Abbildung 3.4.: Abhängigkeiten der Komponenten von $\vec{u}$

Wie in Abbildung 3.4 zu sehen ist, liegen die parallel berechenbaren Punkte auf einer Diagonalen. Der Rechenprozess verläuft gleich einer Wellenfront durch die Matrix wie in Abbildung 3.4a im Hintergrund angedeutet wird. Unter der Beachtung des GS kann des Weiteren nach der Berechnung von $u_{1,0}$ und $u_{0,1}$ die nächste Iteration begonnen werden. Im Anhang B sind die Wellenfronten für verschiedene Stadien verdeutlicht [Moo07].
Durch eine Umverteilung der Daten ist es somit möglich den GS parallel laufen zu lassen. Zu beachten ist, dass zu Beginn die Anzahl der parallel berechenbaren Knoten ansteigt (exemplarisch aufgelistet in Tabelle 3.2a). Äquivalent dazu verhält sich das Ende bei dem die Anzahl der parallel berechnbaren Knoten sinkt. Für Gesamtschrittverfahren wie das Jacobi-Verfahren bietet sich die Wellenfront Ordnung auf Grund der Variation der parallel zu berechnenden Knoten nicht an. Zuerst steigt der Grad der Parallelisierung an, danach fällt er wieder. Dies wiederholt sich mit jedem Iterationsschritt (siehe Tabelle 3.2b). Load Balancing (z.Dt. Lastverteilung) ist schwer realisierbar und somit ineffizient.

Tabelle 3.1.: Anzahl der zu berechnenden Knoten in einem Zeitschritt einer 5×4 Matrix

Schritt	1	2	3	4	5	6	7	8	...
Anz. Knoten	1	2	4	6	8	9	10	10	...

(a) Gauß-Seidel-Verfahren

Schritt	1	2	3	4	5	6	7	8	...
Anz. Knoten	1	2	3	4	4	3	2	1	...

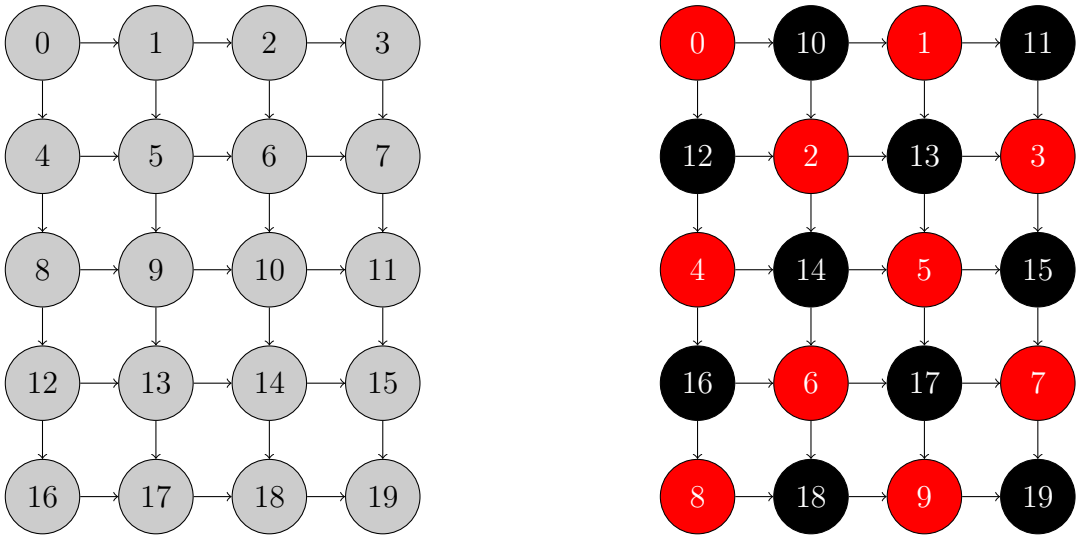
(b) Jacobi-Verfahren

3.2. Rot-Schwarz Anordnung

Aufbauend auf die Welltenfront Anordnung ergibt sich das Rot-Schwarz Gauß-Seidel Verfahren (CGS - Colored Gaus-Seidel-Verfahren). Der CGS trägt viele Namen und ist unter anderem auch als Schachbrett-Schema bekannt und im Englischen als red-black, colored oder two-colored Gauss-Seidel vertreten. Der CGS führt zu einer Umstrukturierung des LGS, gemäß der Welltenfront Anordnung, so dass keine Datenabhängigkeiten innerhalb eines Iterationsschrittes bestehen. Das bedeutet für eine Matrix A:

$$\forall\, 0 \leq i < n, 0 \leq j < i : a_{i,j} = 0 \tag{3.2.1}$$

Die Komponenten der Matrix werden in zwei disjunkte Teilmengen, Rot und Schwarz, zerlegt, so dass schwarze und rote Punkte abwechselnd zueinander liegen. Hierbei werden die Knotenpunkte, angefangen bei Rot, von 0 bis $r-1$ nummeriert und anschließend die Knotenpunkte der schwarzen Teilmenge von r bis $r+b-1$. Es gilt $r+b = n \cdot m$.
Für ein 5×4 Gitter ergeben sich entsprechend zehn rote und zehn schwarze Knotenpunkte (vgl. Abbildung 3.5b).



(a) Lexikographische Aufteilung **(b)** Rot-Schwarz Aufteilung

Abbildung 3.5.: Umstrukturierung durch Rot-Schwarz Aufteilung

Bedingt durch die Umstrukturierung von $\vec{u}$ in $\vec{u}_r$ und $\vec{u}_b$ wird die rechte Seite $\vec{b}$ in $\vec{b}_r$ und $\vec{b}_b$ unterteilt. Matrix A spaltet sich in die vier Matrizen D_r, E, D_b und F auf:

$$\begin{pmatrix} D_r & E \\ F & D_b \end{pmatrix} \cdot \begin{pmatrix} \vec{u}_r \\ \vec{u}_b \end{pmatrix} = \begin{pmatrix} \vec{b}_r \\ \vec{b}_b \end{pmatrix} \qquad \begin{matrix} D_r \in \mathbb{R}^{r \times r} & E \in \mathbb{R}^{r \times b} \\ D_b \in \mathbb{R}^{b \times b} & F \in \mathbb{R}^{b \times r} \end{matrix} \tag{3.2.2}$$

D_r und D_b enstprechen Diagonalmatrizen mit dem Wert vier auf der Hauptdiagonalen. E und F repräsentieren die Abhängigkeiten zwischen den roten und schwarzen Unbekannten,

das heißt die Zeilen der Matrix E drücken die Abhängigkeit eines roten Knotenpunktes zu schwarzen Knotenpunkten aus. Dementsprechend drücken die Zeilen der Matrix F die Abhängigkeit eines schwarzen Punktes zu den roten Punkten aus. Entspricht die Anzahl der roten der Anzahl der schwarzen Punkte, gilt $F = E^T$. Für dieses Beispiel ergibt sich für Matrix E [Zhu94]:

$$\begin{pmatrix}
-1 & & -1 & & & & & & & \\
-1 & -1 & & -1 & & & & & & \\
-1 & & -1 & -1 & -1 & & & & & \\
 & -1 & & -1 & & -1 & & & & \\
 & & -1 & & -1 & & -1 & & & \\
 & & & -1 & -1 & -1 & & -1 & & \\
 & & & & -1 & & -1 & -1 & -1 & \\
 & & & & & -1 & & -1 & & -1 \\
 & & & & & & -1 & & -1 & \\
 & & & & & & & -1 & -1 & -1
\end{pmatrix} \tag{3.2.3}$$

Aus Abbildung 3.6 ist anhand der Diagonalgestalt von D_r und D_b zu erkennen, dass kein roter Knotenpunkt von anderen roten und kein schwarzer von anderen schwarzen abhängt. Bezogen auf das GS ergibt sich folgender Iterationsschritt:

$$\begin{pmatrix} D_r & 0 \\ F & D_b \end{pmatrix} \cdot \begin{pmatrix} u_r^{(k+1)} \\ u_b^{(k+1)} \end{pmatrix} = \begin{pmatrix} b_r \\ b_b \end{pmatrix} - \begin{pmatrix} 0 & E \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} u_r^{(k)} \\ u_b^{(k)} \end{pmatrix} \tag{3.2.4}$$

In Vektorschreibweise ergibt sich für die roten Knoten:

$$D_r \cdot \vec{u}_r^{(k+1)} = \vec{b}_r - E \cdot \vec{u}_b^{(k)} \tag{3.2.5}$$

und für die schwarzen Knoten:

$$F \cdot \vec{u}_r^{(k+1)} + D_b \cdot \vec{u}_b^{(k+1)} = \vec{b}_b \tag{3.2.6}$$

Da D_r und D_b Diagonalmatrizen mit dem Wert vier auf der Hauptdiagonalen sind, können die Gleichungen (3.2.5) und (3.2.6) umgeschrieben werden in:

$$\vec{u}_r^{(k+1)} = \frac{1}{4} \cdot \left(\vec{b}_r - E \cdot \vec{u}_b^{(k)} \right) \tag{3.2.7}$$

$$\vec{u}_b^{(k+1)} = \frac{1}{4} \cdot \left(\vec{b}_b - F \cdot \vec{u}_r^{(k+1)} \right) \tag{3.2.8}$$

Alle Komponenten von $\vec{u}_r$ können simultan berechnet werden wie in (3.2.7) zu erkennen ist. Nach der Berechnung von $\vec{u}_r$ können alle Werte von $\vec{u}_b$ parallel berechnet werden [Zhu94].

Allgemein formuliert ergibt sich für reguläre Gitter in Komponentenschreibweise die Form [RR13]:

$$\left(\vec{u}_r^{(k+1)}\right)_i = \frac{1}{a_{i,i}} \cdot \left(\left(\vec{b}_r\right)_i - \sum_{j \in N(i)} a_{i,j} \cdot \left(\vec{u}_b^{(k)}\right)_j \right) \qquad i = 0, \ldots, r-1 \quad (3.2.9)$$

$$\left(\vec{u}_b^{(k+1)}\right)_i = \frac{1}{a_{i+r,i+r}} \cdot \left(\left(\vec{b}_b\right)_i - \sum_{j \in N(i)} a_{i+r,j} \cdot \left(\vec{u}_r^{(k+1)}\right)_j \right) \qquad i = 0, \ldots, b-1 \quad (3.2.10)$$

$N(i)$ steht für die Indizes der benachbarten Punkte.

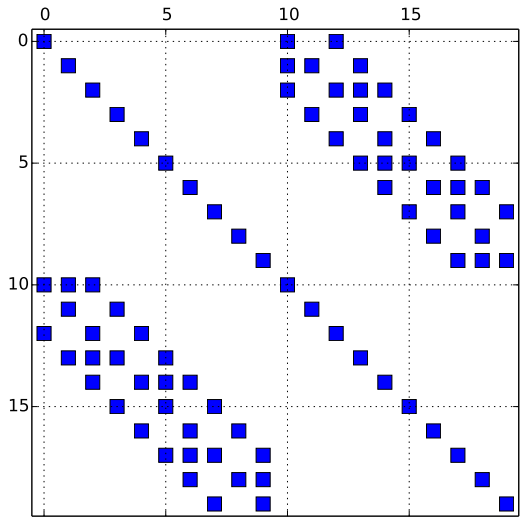


Abbildung 3.6.: Schematische Darstellung der Matrix A

Die Ergebnisse des CGS bleiben trotz Umstrukturierung unverändert, somit ist der CGS invariant. Zu erwarten ist eine schnellere Konvergenz als beim Jacobi-Verfahren, da die errechneten Iterierten des nächsten Schrittes direkt verwendet werden. Bedingt durch die Rot-Schwarz Anordnung wird nur die Hälfte des nächsten Iterationsschrittes verwendet, wodurch der CGS langsamer konvergiert als das klassische Gauß-Seidel-Verfahren, jedoch schneller als das Jacobi-Verfahren.
Zusammengefasst: Je weniger Farben der CGS verwendet, desto langsamer konvergiert es, jedoch desto mehr Parallelität. Kombiniert ist eine ähnliche Laufzeit wie die des Jacobi-Verfahren zu erwarten.

3.3. Implementierung

Gesetzt den Fall, dass $m \cdot n$ eine gerade Zahl ergibt[1], wird die Menge an Punkten in je $\frac{m \cdot n}{2}$ rote und schwarze Unbekannte unterteilt. Im ersten Schritt berechnen alle Prozesse die roten Unbekannten mit Hilfe von Gleichung (3.2.7) und im zweiten Schritt werden alle schwarzen Unbekannten errechnet durch Gleichung (3.2.8). Knotenpunkte, welche am Randgebiet zu den Punkten eines anderen Prozesses liegen, werden entsprechend an diesen Nachbarprozess gesendet, welcher sie empfängt und die alten Werte im Zuge dessen ersetzt. Der Austausch von den Randwerten eines Teilgebietes findet in jedem Schritt statt.
Zur gleichmäßigen Verteilung wird jedem Prozess ein Teilgebiet des $m \times n$ Gitter zugewiesen in dem gleich viele rote wie schwarze Knotenpunkte vorliegen.
Bezogen auf ein 4×4 Gitter werden vier Prozessoren benutzt. Entsprechend wird das Gebiet in vier Teilgebiete zerteilt wie in Abbildung 3.7 dargestellt ist. Jedem Prozess P_i mit $i = 0, \ldots 4$ werden zwei rote und zwei schwarze Punkte zugewiesen.

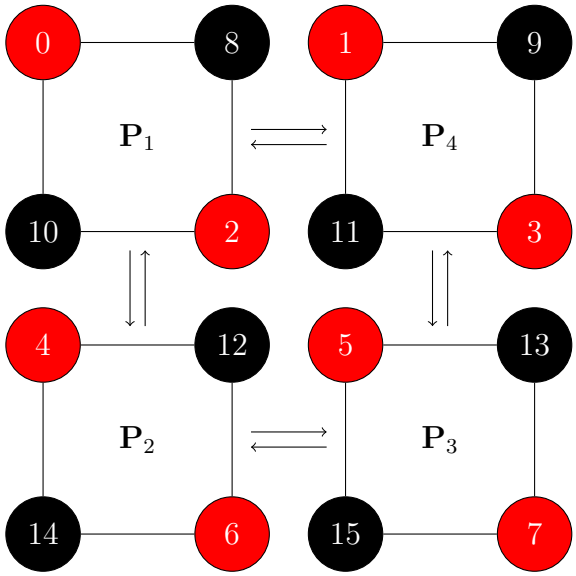


Abbildung 3.7.: Aufteilung der Arbeit auf Prozessoren

3.3.1. Distributed Memory

Liegen die Daten nicht auf getrennten Speichern, müssen nach jeder Rot beziehungsweise Schwarz Berechnung die notwendigen Werte an die Nachbarprozesse geschickt werden. P_1 schickt somit Variable u_2 an P_4 und empfängt von P_4 u_1 und von P_2 u_4. Mit den aktualisierten Rot-Werten werden die Schwarz-Werte berechnet, welche anschließend ebenfalls an die Nachbarprozesse geschickt werden. Parallel werden aktualisierte Werte von eben jenen Nachbarprozessen empfangen.

[1] Ergibt $m \cdot n$ eine ungerade Zahl ist eine Gleichverteilung der Daten nicht möglich

Der Algorithmus mit automatisierter Iterationszahlbestimmung über eine Residuumsgrenze *tol* und eine maximale Iterationszahl *iter_max* lässt sich wie folgt beschreiben:

Solange die L_2-Norm des Residuums ($r = A \cdot u - b$) größer als *tol* und die Anzahl der Iterationen kleiner *iter_max* ist:

- Berechne alle roten Knotenpunkte
- Sende die Werte am Rande zu einem Nachbarprozess zu diesem
- Empfange die vom Nachbarprozess gesendeten Werte
- Berechne alle schwarzen Knotenpunkte
- Sende die Werte am Rande zu einem Nachbarprozess zu diesem
- Empfange die vom Nachbarprozess gesendeten Werte
- Berechne das Residuum

3.3.2. Shared Memory

Handelt es sich um ein System mit geteiltem Speicher, fallen die Punkte in denen aktualisierte Werte gesendet beziehungsweise empfangen werden müssen weg. Stattdessen muss eine Barriere eingefügt werden, um zu gewährleisten, dass jeder Prozess mit den aktuellen Werten rechnet. Prinzipiell sollte es nicht zu bemerkenswerten Verzögerungen kommen, da der Aufwand gleich verteilt ist.
Der Algorithmus lässt sich unter denselben Bedingungen wie beim Distributed Memory wie folgt beschreiben:

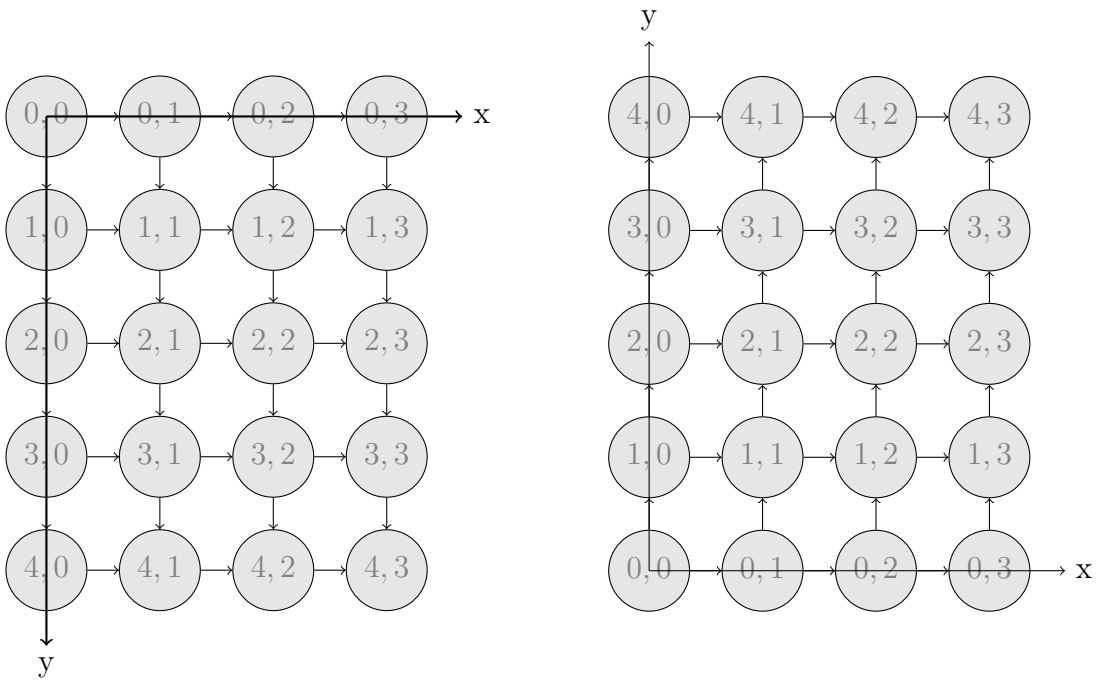
Solange die L_2-Norm des Residuums größer als *tol* und die Anzahl der Iterationen kleiner *iter_max* ist:

- Berechne alle roten Knotenpunkte
- [Barriere] Warte auf alle Prozesse
- Berechne alle schwarzen Knotenpunkte
- [Barriere] Warte auf alle Prozesse
- Berechne das Residuum

Die Parallelisierung findet auf einer GPU statt, somit ein liegt ein Shared Memory System vor. Für die Einbindung in JuROr werden noch Details umgeändert, um eine optimale Implementierung zu gewährleisten, welche im folgenden Kapitel erläutert werden.

4. Realisierung

Der Einfachheithalber und aus Übersichtlichkeitsgründen wurde der CGS im zweidimensionalen auf einem äquidistanten Gitter erklärt. In JuROr wird das Diffusionsproblem jedoch auf einem nicht-äquidistanten Gitter im dreidimensionalen Raum angewandt. Desweitern wird das benutzte Koordinatensystem entlang der x-Achse gespiegelt (vgl. Abbildung 4.1). Der Algorithmus bleibt derselbe, auch wenn eine Dimension dazu kommt und die Abstände nicht mehr gleich sind. Alle Veränderungen werden im Folgenden beschrieben:



(a) Anordung in den vorherigen Kapiteln **(b)** Anordnung in JuROr

Abbildung 4.1.: Änderung der Ausrichtung des Kooridnatensystem

4.1. Änderungen von 2D nach 3D

Anstelle der zweidimensionalen Poisson-Gleichung (3.0.1) wird die dreidimensionale Poisson-Gleichung gewählt, wodurch ein weiterer Summand mit Δz und die Komponente k für die z-Achse hinzukommt.

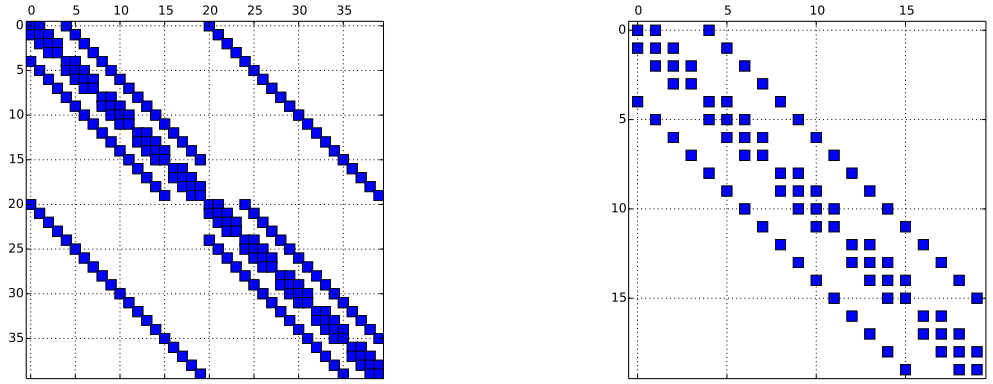
$$f_{i,j,k} = \frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{(\Delta x)^2} + \frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{(\Delta y)^2} + \frac{u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}}{(\Delta z)^2} \tag{4.1.1}$$

Diskretisiert wird diese Gleichung auf einem $m \times n \times o$ Gitter. Im LGS $A \cdot \vec{u} = \vec{b}$ wird die fehlende Komponente k hinzugefügt.

$$u = [u_{0,0,0}, \ldots, u_{0,n-1,0}, \ldots, u_{m-1,n-1,0}, \ldots u_{m-1,n-1,o-1}]^T \tag{4.1.2}$$

$$b = [f_{0,0,0}, \ldots, f_{0,n-1,0}, \ldots, f_{m-1,n-1,0}, \ldots f_{m-1,n-1,o-1}]^T \tag{4.1.3}$$

In Abbildung 4.2 ist die schematische Darstellung der Poisson-Gleichung im Zwei- und im Dreidimensionalen aufgetragen. Der zweidimensionale Fall entspricht dem der in Kapitel 3 besprochen wurde. Dies entspricht der Gittergröße 5×4. Für den dreidimensionalen Fall wurde lediglich eine Ebene hinzu genommen, sodass ein $5 \times 4 \times 2$ Gitter vorliegt. Wird Abbildung 4.2a mit Abbildung 4.2b verglichen, fällt auf, dass der zweidimensionale Fall dem oberen linken beziehungsweise dem unteren rechten Viertel des dreidimensionalen Falls entspricht. Dies ist bedingt dadurch, dass sich die Beziehung der Punkte auf der x-y-Ebene nicht verändert hat, jedoch jeder Punkt zusätzlich noch einen Nachbarn auf der z-Ebene bekommen hat, deren Verknüpfung sich in den anderen beiden Vierteln wiederspiegelt.



(a) Dreidimensionaler Raum: $5 \times 4 \times 2$ Gitter **(b)** Zweidimensionaler Raum: 5×4 Gitter

Abbildung 4.2.: Diskrete Poisson-Gleichung in 2D und 3D

Im Nachfolgenden werden die Diskretisierung mit Hilfe der FDM und die Anordnung des CGS genutzt, um eine Formel zur Berechnung der einzelnen Komponenten für Rechnen auf einem Computer aufzustellen.

4.2. Bestimmung der Formel für einen Rechenschritt

Als Ausgangsgleichung wird die homogene Diffusionsgleichung genommen:

$$\partial_t \vec{u} = \nu \nabla^2 \vec{u} \tag{4.2.1}$$

Mit Hilfe des impliziten Euler-Schrittes für die Zeitdiskretisierung wird die Gleichung umgeformt. Hierbei ist I die Einheitsmatrix und n der Zeitschritt.

$$\frac{\vec{u}^{(n+1)} - \vec{u}^{(n)}}{\Delta t} = \nu \nabla^2 \vec{u}^{(n+1)} \tag{4.2.2}$$

$$\Leftrightarrow \quad \vec{u}^{(n+1)} - \nu \Delta t \nabla^2 \vec{u}^{(n+1)} = \vec{u}^{(n)} \tag{4.2.3}$$

$$\Leftrightarrow \quad \underbrace{\left(I - \nu \Delta t \nabla^2\right)}_{A} \cdot \underbrace{\vec{u}^{(n+1)}}_{\vec{x}} = \underbrace{\vec{u}^{(n)}}_{\vec{b}} \tag{4.2.4}$$

Gemäß $\nabla^2 = \partial_x + \partial_y + \partial_z$ ergibt sich durch die Diskretisierung mit dem zentralen Differenzenquotienten für die komponentenweise Darstellung an der Stelle (i, j, k):

$$\begin{aligned} u_{i,j,k}^{(n+1)} - \nu \Delta t \Bigg(& \frac{u_{i+1,j,k}^{(n+1)} - 2u_{i,j,k}^{(n+1)} + u_{i-1,j,k}^{(n+1)}}{\Delta x^2} + \frac{u_{i,j+1,k}^{(n+1)} - 2u_{i,j,k}^{(n+1)} + u_{i,j-1,k}^{(n+1)}}{\Delta y^2} \\ & + \frac{u_{i,j,k+1}^{(n+1)} - 2u_{i,j,k}^{(n+1)} + u_{i,j,k-1}^{(n+1)}}{\Delta z^2} \Bigg) = u_{i,j,k}^{(n)} \end{aligned} \tag{4.2.5}$$

$$\begin{aligned} \Leftrightarrow \underbrace{u_{i,j,k}^{(n)}}_{b_{i,j,k}^{(n)}} = & \underbrace{\left[1 + 2 \cdot \nu \Delta t \left(\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2} \right)\right]}_{\beta^{-1}} \cdot u_{i,j,k}^{(n+1)} \\ & - \underbrace{\frac{\nu \Delta t}{(\Delta x)^2}}_{\alpha_x} \left(u_{i+1,j,k}^{(n+1)} + u_{i-1,j,k}^{(n+1)} \right) \\ & - \underbrace{\frac{\nu \Delta t}{(\Delta y)^2}}_{\alpha_y} \left(u_{i,j+1,k}^{(n+1)} + u_{i,j-1,k}^{(n+1)} \right) \\ & - \underbrace{\frac{\nu \Delta t}{(\Delta z)^2}}_{\alpha_z} \left(u_{i,j,k+1}^{(n+1)} + u_{i,j,k-1}^{(n+1)} \right) \end{aligned} \tag{4.2.6}$$

Mit l für den Iterationsschritt ergibt sich:

$$\begin{aligned}\left[u_{i,j,k}^{(n+1)}\right]^{(l+1)} = \beta \cdot \Big[b_{i,j,k}^{(n)} &+ \alpha_x \cdot \left(\left[u_{i+1,j,k}^{(n+1)}\right]^{(l)} + \left[u_{i-1,j,k}^{(n+1)}\right]^{(l)} \right) \\ &+ \alpha_y \cdot \left(\left[u_{i,j+1,k}^{(n+1)}\right]^{(l)} + \left[u_{i,j-1,k}^{(n+1)}\right]^{(l)} \right) \\ &+ \alpha_z \cdot \left(\left[u_{i,j,k+1}^{(n+1)}\right]^{(l)} + \left[u_{i,j,k-1}^{(n+1)}\right]^{(l)} \right) \Big]\end{aligned} \tag{4.2.7}$$

Die Formel 4.2.7 entspricht der komponentenweisen Berechnung im Jacobi-Verfahren. Für den CGS ist die Unterscheidung zwischen roten und schwarzen Punkten nötig. Für die roten Punkte gilt 4.2.7, während für die schwarzen Punke (l) durch $(l+1)$ ersetzt werden müsste, da die schwarzen Punkte die bereits berechneten roten Punkte verwenden.
Programmiertechnisch spielt es jedoch keine Rolle, ob der zu berechnende Punkt der Menge der roten oder schwarzen Punkte angehört. Aufgrund der Unabhängigkeit der beiden Mengen untereinander, kann die Matrix während des Iterationsschrittes selbst neu beschrieben werden. Das heißt im ersten Schritt indem die roten Punkte berechnet werden, wird auf die schwarzen Punkte der Matrix A zu gegriffen und das errechnete Ergebnis direkt wieder in Matrix A geschrieben. Dadurch ist keine Unterscheidung zwischen bereits berechneter und alter Wert nötig. Das Jacobi-Verfahren hingegen benötigt eine Eingangsmatrix von denen es die alten Punkte bezieht und eine Ausgangsmatrix in denen es die berechneten Punkte schreibt. Ansonsten ist es nicht möglich zwischen den Werten des vorherigen und des aktuellen Iterationsschrittes zu unterscheiden. Für den CGS muss lediglich unterschieden werden, wann welche Punkte berechnet werden. Das heißt zuerst werden alle roten Punkte und danach alle schwarzen Punkte berechnet. Nachfolgend wird der Algorithmus näher beschrieben.

4.3. Realisierung in JuROr

4.3.1. Programmkonzept

Der Rot-Schwarz Gauß-Seidel wird als Löser für Diffusionsprobleme in JuROr implementiert. Die Klasse, genannt `ColoredGaussSeidel.cpp`, implementiert das Interface `DiffusionI`.

```
1  class DiffusionI {
2  public:
3     DiffusionI();
4     virtual ~DiffusionI();
5     virtual void diffuse(Field* out, const Field* in, const Field* b, bool sync = true)=0;
6  };
```

Listing 4.3.1: Interface DiffusionI.h

`Field` ist eine in JuROr erstellte Datenstruktur, welche im Groben die Aufgaben eines Arrays erfüllt mit spezifischen Anforderungen auf das jeweilige Problem (Diffusion, Druck oder Advektion). `Field` wird auch zur Speicherung der Parameter verwendet. Die Variable `sync` ist lediglich für die parallelisierte Version auf der GPU von Bedeutung (siehe Kapitel 4.3.3).

Der Diffusionslöser ist einer von vier Lösern an denen momentan gearbeitet wird. Wie im Klassendiagram 4.3 angedeutet, implementiert jeder Löser sein entsprechendes Interface, womit der Austausch des Lösers lediglich in der einzulesenden XML-Datei 4.3.2 (Zeile 13) vorgenommen wird. Desweitern werden in der Datei die Parameter für das zu berechnende Gebiet und die physischen Parameter angegeben. Dies ermöglicht eine Änderung der Eingabeparameter ohne das Programm erneut kompilieren zu müssen.

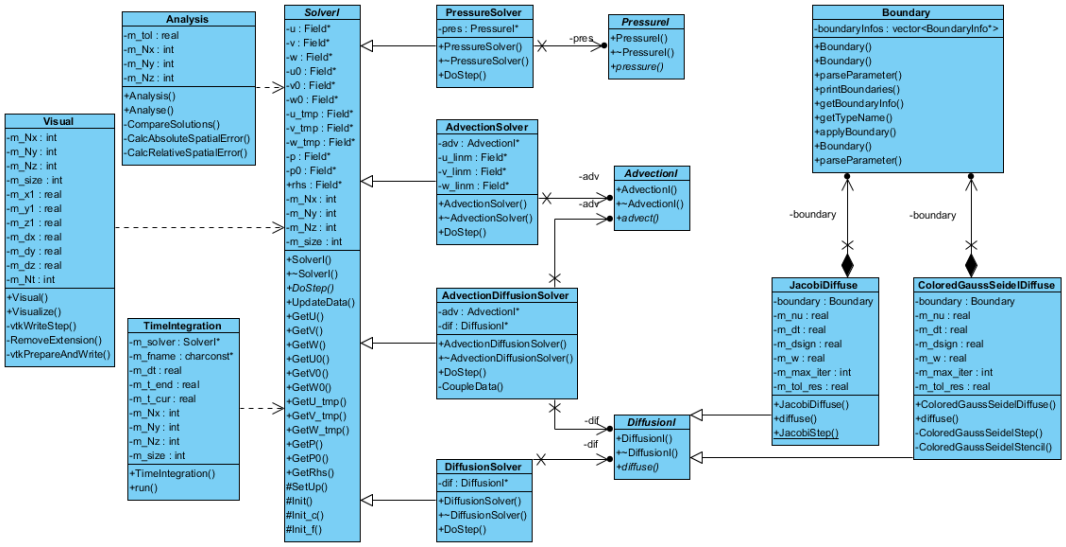


Abbildung 4.3.: Klassenstruktur, der bisher vorhandenden Löser von JuROr. Jeder Löser (gemäß des Interfaces) kann analysiert werden mit Hilfe der Klassen `Analysis` (Vergleich mit analytischer Lösung, wenn vorhanden), `TimeIntegration` (Steuerung der Zeitschleife) und `Visual` (Erzeugung von vtk-Dateien [1]).

[1] Das Visualization Toolkit (vtk) ist eine kostenlose open-source Software für 3D Computer Graphiken und Visualisierung.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <JuROr>
3    <xml_filename>Diffusion.xml</xml_filename>

4    <physical_parameters>
5      <t_end> 1.0 </t_end>    <!-- simulation end time -->
6      <dt> 0.0125 </dt>       <!-- time stepping, caution: CFL-condition dt < 0.5*dx^2/nu -->
7      <nu> 0.001 </nu>        <!-- physical diffusion -->
8      <alpha> 0. </alpha>     <!-- gravitational force -->
9      <beta> 0. </beta>       <!-- buoancy force -->
10     <kappa> 0. </kappa>     <!-- thermal diffusion -->
11   </physical_parameters>

12   <solver description = "DiffusionSolver" >
13     <diffusion type = "ColoredGaussSeidel" field = "u,v,w">//field im Sinne des Zielfeldes
14         <max_iter> 100 </max_iter>  <!-- max number of iterations -->
15         <tol_res> 1e-07 </tol_res>  <!-- tolerance for residuum/ convergence -->
16     </diffusion>
17     <solution available = "yes">
18         <tol> 1e-03 </tol>    <!-- tolerance for further tests -->
19     </solution>
20   </solver>

21   <domain_parameters>
22     <x1> 0. </x1>         <!-- computational domain -->
23     <x2> 2.0 </x2>
24     <y1> 0. </y1>
25     <y2> 2.0 </y2>
26     <z1> 0. </z1>
27     <z2> 2.0 </z2>
28     <Nx> 22 </Nx>         <!-- grid resolution (number of cells incl. ghost cells!, number of faces
 ↪   Nx+1, inner Nx-2, index: all 0...Nx-1, inner: 1...Nx-2) -->
29     <Ny> 22 </Ny>
30     <Nz> 22 </Nz>
31   </domain_parameters>

32   <boundaries>
33     <boundary field="u,v,w" patch="front,back,left,right,top,bottom" type="dirichlet" value="0.0" />
34   </boundaries>

35   <initial_conditions usr_fct = "ExpSinusProd" > <!-- product of exponential and sinuses
 ↪   exp*sin*sin*sin -->
36     <l> 2 </l>                                  <!-- wavelength -->
37   </initial_conditions>
38 </JuROr>
```

Listing 4.3.2: XML-Datei mit Eingabeparametern für JuROr: Festlegung der physischen Parameter (Z. 4-11) und des Lösers (Z.13) und den Abbruchkriterien (Z. 14-15). Liegt eine analytische Lösung vor, wird dies zusammen mit der Abweichungstoleranz in Zeile 17 bzw. 18 angegeben. Parameter, welche das Gebiet oder auch Feld betreffen, werden in den Zeilen 22-30 eingetragen. Rand- und Anfangsbedingungen werden in den Zeile 32-37 festgelegt.

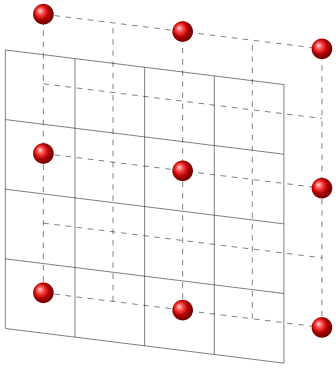
4.3.2. Realisierung des Algorithmus

Die Bestimmung der Menge der roten bzw. schwarzen Punkte wird mit Hilfe von jeweils zehn for-Schleifen realisiert. Beispielhaft werden diese in Quellcode 4.4 für die roten Punkte gezeigt. Der entsprechende Quellcode findet sich unter Anhang C.2.1.

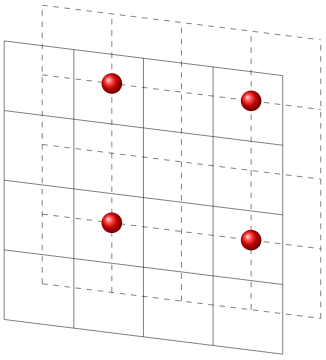
```
1   //red
2   for (int k = 1; k < nz - 1; k += 2) {
3       for (int j = 1; j < ny - 1; j += 2) {
4           for (int i = 1; i < nx - 1; i += 2) {
5               ColoredGaussSeidelStencil(i, j, k, ...);
6           }
7       }
8       for (int j = 2; j < ny - 1; j += 2) {
9           for (int i = 2; i < nx - 1; i += 2) {
10              ColoredGaussSeidelStencil(i, j, k, ...);
11          }
12      }
13  }

14  for (int k = 2; k < nz - 1; k += 2) {
15      for (int j = 1; j < ny - 1; j += 2) {
16          for (int i = 2; i < nx - 1; i += 2) {
17              ColoredGaussSeidelStencil(i, j, k, ...);
18          }
19      }
20      for (int j = 2; j < ny - 1; j += 2) {
21          for (int i = 1; i < nx - 1; i += 2) {
22              ColoredGaussSeidelStencil(i, j, k, ...);
23          }
24      }
25  }
```

Abbildung 4.4.: Bestimmung der Teilmenge der roten Punkte, wobei `nx`, `ny` und `nz` für die Anzahl der Punkte auf der jeweilige Koordinatenachse stehen. Die Methode `ColoredGaussSeidelStencil` berechnet den Wert für die Position i, j, k. Die vollständige Methode ist im Anhang C.3.1 zu finden.

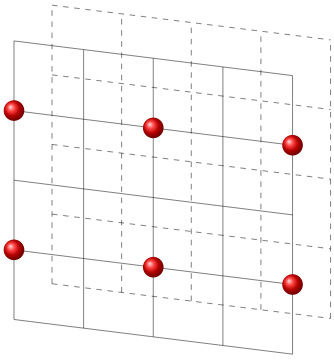
Die Zeilen 2-13 bestimmen die roten Punkte auf jeder Ebene, aufgespannt durch x und y, mit ungerader z-Koordinate. Umgekehrt bestimmen die Zeilen 14-25 die roten Punkte auf den Ebenen mit gerader z-Koordinate. Insbesondere ermitteln die for-Schleife der Zeilen 3-7 die Punkte mit gerader x- und y-Koordinate (vgl. Abbildung 4.5a) und die for-Schleife der Zeilen 8-12 die Punkte mit ungerader x- und y-Koordinate (vgl. Abbildung 4.5b). Äquivalent verhalten sich die Zeilen 15-19 (vgl. Abbildung 4.5c) und die Zeilen 20-24 (vgl. Abbildung 4.5d). Gemeinsam bilden sie die Menge aller roten Punkte auf dem Gitter (vgl. Abbildung 4.5e). Kombiniert mit den schwarzen Punkten, ergibt sich das komplette Gitter (vlg. Abbildung 4.5f). In diesem Fall gilt für alle roten Punkte: Die Summe ihrer Koordinaten ist ungerade. Entgegengesetzt entspricht die Summe der Koordinaten der schwarzen Punkte einer geraden Zahl.
Die zur Parallelisierung benutzten Pragmas werden nachfolgend vorgestellt.



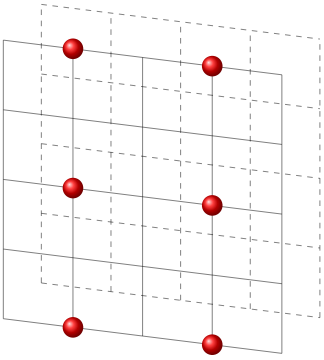
(a) Verbildlichung von 4.4: Zeilen 3-7



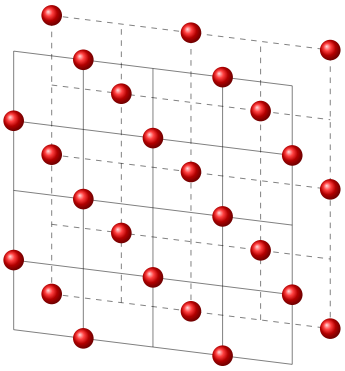
(b) Verbildlichung von 4.4: Zeilen 8-12



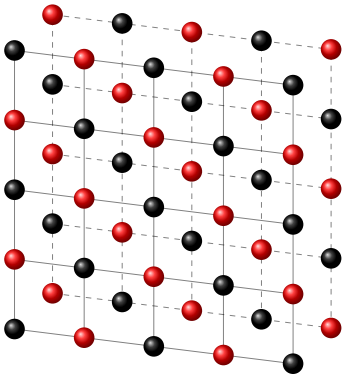
(c) Verbildlichung von 4.4: Zeilen 15-19



(d) Verbildlichung von 4.4: Zeilen 20-24



(e) Verbildlichung von 4.4



(f) Verbildlichung vom gesamten ColoredGaussSeidelStep C.2.1

Abbildung 4.5.: Bildliche Darstellung des Algorithmus 4.4 in seinen Einzelteilen

4.3.3. Parallelisierung

Der zuvor beschriebene Algorithmus wurde mit Hilfe von OpenACC parallelisiert. OpenACC wird ähnlich wie OpenMP über Compiler-Pragmas gesteuert. Serieller Code wird parallelisiert indem bestimmte Direktiven vorangestellt werden. Die Parallelisierung ist sowohl auf der GPU als auch der CPU nutzbar. Für eine serielle Ausführung wird das Programm auf einen Kern beschränkt, welches den selben Effekt hat wie den nicht parallelisierten Code zu nutzen. Ein Compiler ohne OpenACC Unterstützung wird weiterhin ein korrektes, nicht parallelisiertes Programm erzeugen. Der Portierungsaufwand ist somit vergleichsweise klein, jedoch führt eine compilergesteuerte Parallelisierung nicht immer zur bestmöglichen Parallelisierung.
Innerhalb der Klasse `ColoredGaussSeidel` wurden die folgenden Direktiven benutzt:

#pragma acc data ist ein Daten-Konstrukt, welches eine Region definiert in der Daten für den Accelerator zugänglich sind.

present ist ein Daten-Ausdruck und besagt, dass die Daten dem Accelerator bereits vorliegen. Die entsprechende Kopie wird gesucht und benutzt.

#pragma acc kernels ist ein Kernel-Konstrukt, welcher eine oder mehrere Schleifen einschließt und diese auf dem Accelerator ausführt.

async ermöglicht einen asynchronen Datenaustausch zwischen Host und Accelerator. Sie steht in der Regel am Ende eines Befehls. Zum Beispiel: `#pragma acc kernels present (out, d_out[:bsize], b, ` (Aus ColoredGaussSeidelStep Zeile 9 C.2.1)

#pragma acc wait ist eine sogenannte „Wait-Directive“, welche den Host zwingt auf die Fertigstellung der asynchronen Accelerator Aktivitäten zu warten. Ohne zusätzliche Angaben wird auf alle ausstehenden Aktivitäten gewartet.

#pragma acc loop ist ein Schleifen-Konstrukt, welches für die darauf folgende und alle in ihr verschachtelten Schleifen gilt. Darauf folgt ein Ausdruck wie beispielsweise *independent*, welche die Art der Ausführung auf dem Accelerator bestimmt.

independent besagt, dass die Schleifeniterationen datenunabhängig sind und parallel ausgeführt werden können.

collapse(n) beschränkt die assoziierte Direktive auf die folgenden, verschachtelten `n` for-Schleifen.

Weitere Direktiven finden sich im „Quick Reference Guide“ von OpenACC.
Mit den vorgestellten Pragmas und dem Algorithmus wurde der CGS in JuROR umgesetzt. Die Methoden der Klasse ColoredGaussSeidel findet sich im Anhang C. Die durch den Compiler umgesetzte Parallelisierung wurde mit Hilfe des NVIDIA Visual Profilers (nvprof) überprüft (vgl. Abbildung 4.6).

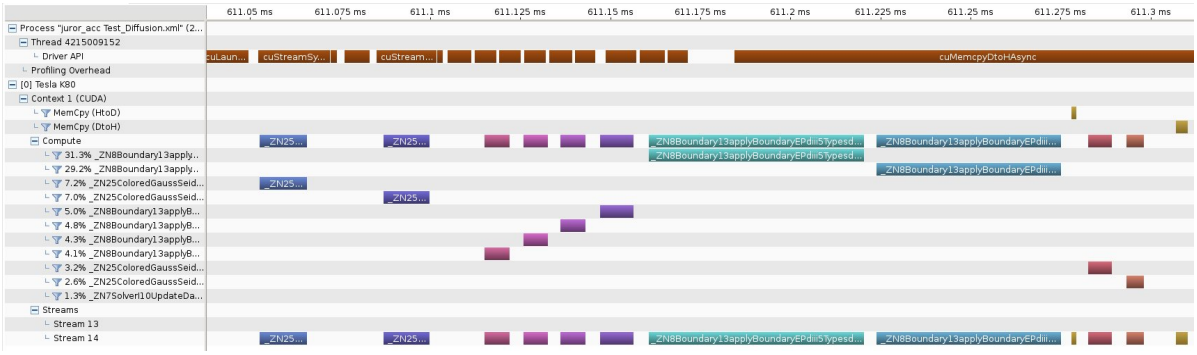


Abbildung 4.6.: Ausschnitt aus nvprof: Darstellung aus einem Zeitschleifendurchgang

In Abbildung 4.6 dargestellt ist ein Zeitschleifendurchgang. In diesem Zusammenhang sind nur die dritte und die vierte Zeile unter der Kategorie *Compute* (7.2% _ZN25ColoredGaussSeid... und 7.0% _ZN25ColoredGaussSeid...) interessant. Ersteres repräsentiert die Berechnung der roten Punkte und zweiteres entsprechend die Berechnung der schwarzen Punkte. Anschließend folgen Zuweisung für die Randwerte (die äußeren Punkte). Aus der blockweisen Darstellung ist zu schließen, dass die roten beziehungsweise schwarzen Punkte jeweils parallel berechnet werden. Die Lücke zwischen den beiden Rechenschritten entspricht der Zeit, indem ein neuer Kernel bereit gestellt wird für die Berechnung der schwarzen Punkte. Momentan nehmen die Boundary-Methoden den Hauptteil der Laufzeit ein. Für die nachfolgenden Vergleiche bezüglich der Performanz des CGS und des Jacobi-Verfahren spielt dies eine untergeordnete Rolle, da beide Verfahren auf dieselben Boundary-Methoden zurück greifen. Im Nachfolgenden wird der Algorithmus weiterhin auf seine Richtigkeit, seine Konvergenz und seine Performanz geprüft.

5. Analyse

5.1. Bewertung

5.1.1. Verifizierung

Die Verifizierung der Daten findet durch die Analysis-Klasse in JuROr statt. Es wird ein fiktives Problem geschaffen, dessen analytische Lösung bekannt ist und mit der numerisch errechneten Lösung verglichen. Unterschreitet die L_20Norm der Differenz zwischen numerischer und analytischer Lösung die in der XML-Datei angegeben Toleranzgrenze, so gilt der Test als bestanden. Dies dient vornehmlich der Unterstützung des Entwicklers, um zu testen, ob der Algorithmus die richtigen Ergebnisse liefert. Wie in der XML-Datei abzulesen ist, gibt es zu den im Rahmen dieser Bachelorarbeit genutzten Testfällen eine analytische Lösung (`ExpSinusProd` vgl. Abbildung 5.1).

$$e^{-kt} \cdot sin(-kt) \cdot sin(l\pi(x1 + (i - 0.5)dx)) \cdot sin(l\pi(y1 + (j - 0.5)dy)) \cdot sin(l\pi(z1 + (k - 0.5)dz)) \tag{5.1.1}$$

Diese ist in der Klasse Functions.cpp definiert:

```
1  void ExpSinusProd(Field* out, real t){
2    int Nx = out->GetNx();
3    int Ny = out->GetNy();
4    int Nz = out->GetNz();
5    real x1 = out->Getx1();
6    real y1 = out->Gety1();
7    real z1 = out->Getz1();
8    real dx = out->Getdx();
9    real dy = out->Getdy();
10   real dz = out->Getdz();
11   auto params = Parameters::getInstance();
12   real nu = params->getReal("physical_parameters/nu");
13   real l = params->getReal("initial_conditions/l");
14   real kpinu = 3 * l * l * M_PI * M_PI * nu;
15   FOR_EACH_CELL(Nx, Ny, Nz) // boundary is initialized as well
16     out->data[IX(i, j, k, Nx, Ny)] = exp(-kpinu * t) * sin(l * M_PI * xi(i, x1, dx)) * sin(l * M_PI *
     ↪   yj(j, y1, dy)) * sin(l * M_PI * zk(k, z1, dz));
17   END_FOR
18 }
```

Abbildung 5.1.: Ausschnitt aus der Klasse Functions.cpp

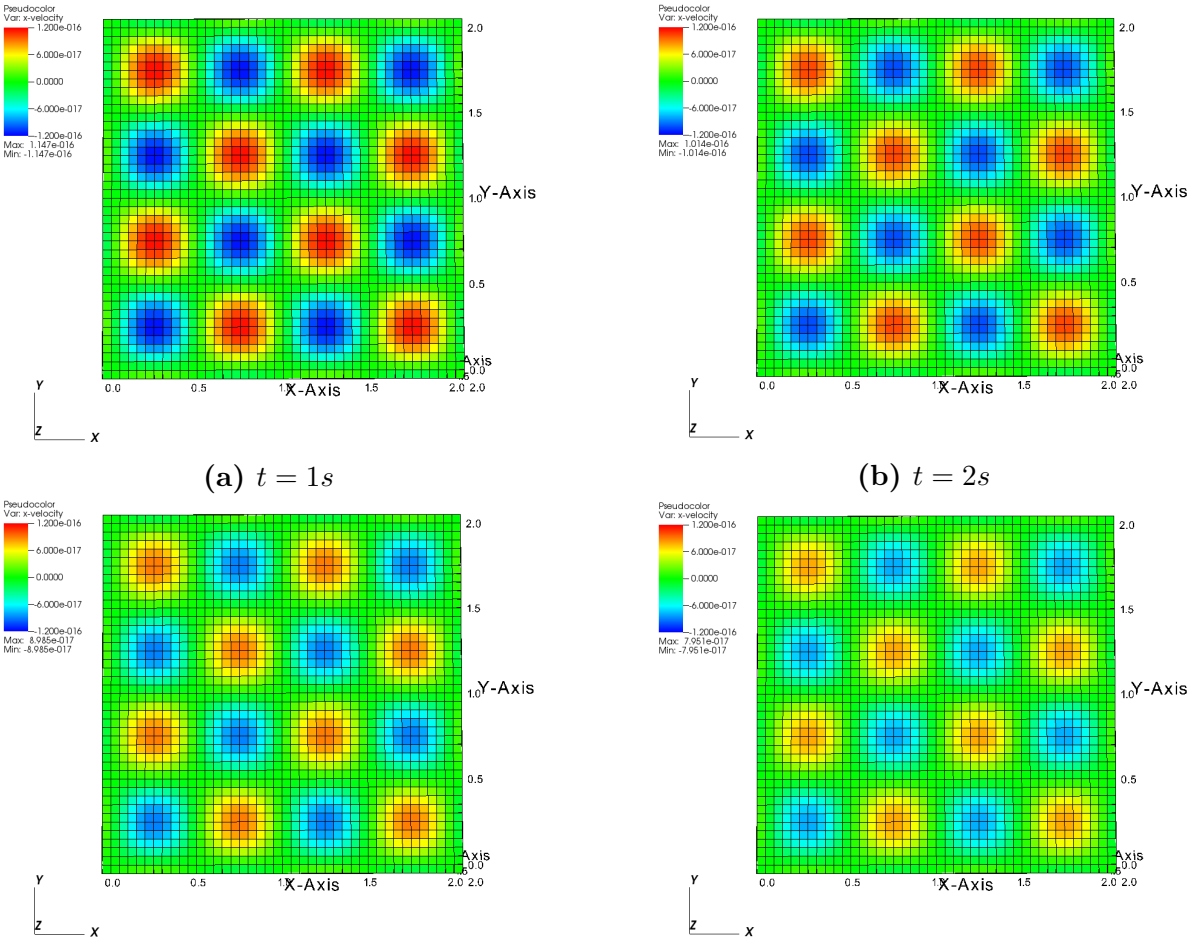
`M_PI` ist eine Konstante aus der Datei math.h und repräsentiert π. `x_i( )`, `y_j( )`, `z_k( )` und `IX( )` sind Methoden der GlobalMacrosTypes.h Datei:

```
1 #define IX(i,j,k,Nx,Ny) ((i) + (Nx)*(j) + (Nx)*(Ny)*(k)) // row-major index for one dimensional arrays
   ↪   (i=0..Nx-1 columns, j=0..Ny-1 rows, k = 0...Nz-1)

2 #define xi(i,x,dx) ((x) + ((i)-0.5)*(dx)) // physical xcoords at midpoints calculated by index
   ↪   i=0..Nx-1
3 #define yj(j,y,dy) ((y) + ((j)-0.5)*(dy)) // physical ycoords at midpoints calculated by index
   ↪   j=0..Ny-1
4 #define zk(k,z,dz) ((z) + ((k)-0.5)*(dz)) // physical ycoords at midpoints calculated by index
   ↪   j=0..Ny-1
```

Abbildung 5.2.: Ausschnitt aus der GlobalMacrosTypes.h Datei

5.1.2. Visualisierung

Die qualitative Bewertung findet mittels Visualisierung statt. Die Klasse `Visual` schreibt vtk-Dateien heraus, welche mit Hilfe von Visualierungstools wie VisIt oder auch ParaView die numerische Lösung abbilden kann. Dies ermöglicht einen visuelle Bewertung der Ergebnisse.



(a) $t = 1s$ **(b)** $t = 2s$

(c) $t = 3s$ **(d)** $t = 4s$

Abbildung 5.3.: Visuelle Darstellung des Diffusionsproblems zu verschiedenen Zeitpunkten

In Abbildung 5.3 sind für vier Zeitschritte die errechneten Ergebnisse aufgetragen. Zur Veranschaulichung wurde lediglich die xy-Ebene visualisiert. Zu erwarten ist eine Dämpfung der Amplitude bei gleichbleibender Fläche. Die Dämpfung der Amplitude lässt sich gut erkennen anhand der Farben und die gleichbleibende Fläche zeigt sich in der gleichbleibenden Kreisgröße. Das betrachtete Beispiel entspricht den Erwartungen und mit Hilfe der Analysis Klasse wurde der entwickelte Algorithmus für eine Toleranzgrenze von $\varepsilon = 10^{-3}$ als korrekt bewertet.

5.2. Konvergenz

5.2.1. Konvergenz im Raum

Die quantitative Bewertung des CGS wurde mit Hilfe der L_2-Norm, auch bekannt als die euklidische Norm, erstellt. Für $u_{i,j,k}$ als Wert der exakten und $\tilde{u}_{i,j,k}$ als Wert der numerischen Lösung ergibt sich folgende Formel für die Fehlerberechnung zum festen Zeitpunkt t normiert im Ort:

$$\varepsilon(t) := \sqrt{\frac{1}{(Nx-2)(Ny-2)(Nz-2)} \sum_{i=1}^{Nx-2} \sum_{j=1}^{Ny-2} \sum_{k=1}^{Nz-2} (u_{i,j,k} - \tilde{u}_{i,j,k})^2(t)}. \tag{5.2.1}$$

Die L_2-Norm ist Bestandteil der Analysis-Klasse, welche bei Angabe einer analytischen Lösung in der XML-Datei am Ende der Berechnung den Fehler mit ausgibt.
Zur Untersuchung wurden diverse Gittergrößen beziehungsweise Schrittweiten gewählt, beginnend bei $Nx = 12$ bis zu $Nx = 128$ mit $Ny = Nz = 22$ bei einem konstanten Δt zum Zeitpunkt $t = t_{end}$.

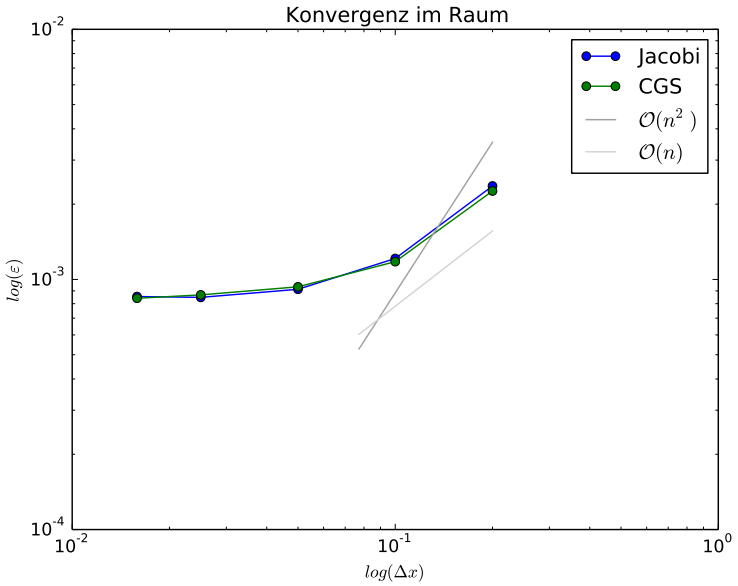


Abbildung 5.4.: Konvergenzuntersuchung

Die zu erwartende Konvergenzordnung entspricht $p = 2$. Allerdings wie in Abbildung 5.4 zu

erkennen ist, stimmt dies nicht mit den Ergebnissen überein. Es handet sich eher um eine Konvergenzordnung von $p = 1$. Um einen Fehler bei der Implementierung auszuschließen, wurde für dieselbe Problemstellung das Jacobi-Verfahren aufgetragen. Der Graph vom Jacobi-Verfahren verläuft ähnlich wie der des CGS. Dies kann bedingt sein durch den Fehler, der im Zeitschritt gemacht wird. Alternativ kann es an der Floating Point Arithmetik liegen. Nach [Drz+14] bietet sich eine Verkleinerung von Δx nur an, wenn folgendes gilt:

$$\Delta x \geq \sqrt{\epsilon} \tag{5.2.2}$$

Hierbei ist $\epsilon = 10^{-16}$ die Floating Point Genauigkeit und für die finite Differenzen der 2. Ordnung gilt somit:

$$\Delta x \geq 10^{-5} \tag{5.2.3}$$

Die Gleichung ist in diesem Fall erfüllt obgleich bei den ersten beiden Punkten aus Abbildung 5.4 mit dem Auge bereits keine Änderung zu erkennen ist. Dies kann an dem verwendeten Datentyp liegen. In JuROr wird mit dem Datentyp double gerechnet, welcher über 15 Nachkommastellen verfügt ($\epsilon = 10^{-16}$). Für eine höhere Genauigkeit beziehungsweise um den Fehler im Raum bestimmen zu können, müsste mit einem anderen Datentyp gearbeitet werden. Es gibt eine Bibliothek namens GMP (GNU multiple precision library), welche für diesen Zweck eingebunden werden kann.
Eine andere Möglichkeit bietet die getrennte Betrachtung der Löser. Mit einer einfachen Laplace-Gleichung könnte die räumliche Diskretisierung getrennt von der zeitlichen untersucht werden. Damit ließe sich ein Fehler durch die zeitliche Diskretisierung ausschließen.
Eine schnellere Konvergenz konnte somit nicht nachgewiesen werden, weshalb im Folgenden zusätzlich das Residuum aufgetragen gegen die Iterationsanzahl betrachtet wird.

5.2.2. Anzahl der Iterationen

Für diese Untersuchung wurde anstelle der implementierten automatischen Iterationszahlbestimmung über eine Residuumsbestimmung und Toleranzgrenze die maximale Anzahl an Iterationen manuell fest gesetzt. Zur Analyse wurde ein Gitter der Größe $496 \times 522 \times 522$ genommen. Wie in Abbildung 5.5 zu erkennen ist, hat der CGS ein kleineres Residuum als das Jacobi-Verfahren bei der gleichen Iterationsanzahl. Das bedeutet der CGS konvergiert schneller als das Jacobi-Verfahren. Wird die Problemstellung mit dem Abbruchkriterium $res > tol\_res$ durchlaufen, braucht der CGS durchschnittlich 22 Iterationen und das Jacobi-Verfahren 48. Daraus lässt sich schließen, dass bei wenigen Iterationen der CGS eine bessere Näherung berechnet als das Jacobi-Verfahren. Dies bezieht sich jedoch lediglich auf diese Problemstellung, da bei gröberen Gittern meist das Jacobi-Verfahren weniger Iterationen benötigt als der CGS.
Nach [Dem96] gilt jedoch, dass der CGS in Kombination mit dem SOR-Verfahren in jedem Fall schneller konvergiert als das Jacobi-Verfahren. In JuROr ist die Gewichtung des SOR-Verfahrens bereits implementiert. Für die Berechnung wurde $\omega = 1$ gewählt, womit es sich um den CGS ohne Gewichtung handelt. Wie in Kapitel 2 vorgestellt, konvergiert das SOR-Verfahren für $0 < \omega < 2$. Jedoch lohnen sich laut [Dem96] keine Werte kleiner eins, womit sich eine Parameteranalyse für $1 < \omega < 2$ anbietet, da ohne die Überrelaxation eine schnellere Konvergenz nicht in jedem Fall gegeben ist und durch eine Laufzeit der gleichen Größenordnung der CGS keinen Vorteil gegenüber dem Jacobi-Verfahren hat. Diese Parameteranalyse wird im weiteren Verlauf in

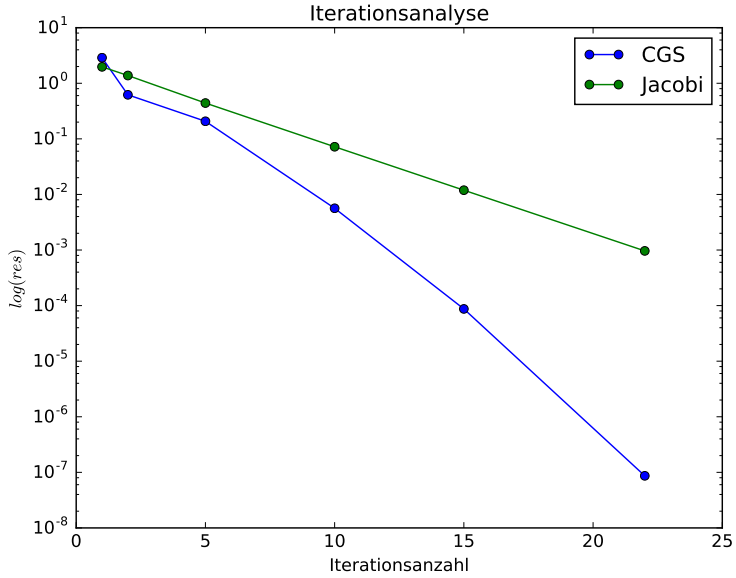


Abbildung 5.5.: Anzahl der Iterationen

Kooperation mit dem Forschungszentrum Jülich fortgeführt.

Zusammengefasst ergab die Konvergenzanalyse eine Ordung $p = 1$ bei der Konvergenz im Ort, jedoch zeigt die Iterationsanalyse, dass mit Hilfe der Gewichtung des SOR-Algorithmus eine weitere Optimierung möglich ist.

5.3. Performanz

Im Nachfolgenden wird die Leistungsfähigkeit des CGS abgeschätzt, indem ein Vergleich zwischen seriellem und parallelisiertem Code und ein Vergleich zwischen dem Jacobi-Verfahren und dem CGS angesetzt wird. Das erwartete beziehungsweise erwünschte Ergebnis ist, dass der parallelisierte CGS laufzeitmäßig schneller ist als seine serielle Version und parallelisiert in derselben Größenordnung wie das parallelisierte Jacobi-Verfahren fällt.

5.3.1. Vergleich: Serieller und parallelisierter Code

Verglichen wird der serielle, der multicore und der GPU-parallelisierte Code. Ausgeführt wird er auf dem Supercomputer JURECA (Details zu JURECA in Anhang D) des Forschungszentrums Jülich. Der serielle Code und die multicore Version laufen auf einem Intel Xeon E5-2680 v3 Haswell CPU und die GPU-parallelisierte Version, nachfolgend als GPU Version, läuft auf einem Chip der NVIDIA K80 GPU.

Bei der Parallelisierung sind mehrere Probleme aufgetreten von denen das gravierendste in 5.6a abgebildet wurde. Mit der Parallelisierung des CGS fällt der CUPS (cell updates per seconds 5.3.1) Wert auf zirka die Hälfte im Vergleich zum seriellen Code (je höher der CUPS Wert,

desto performanter).

$$CUPS = (Nx-2)\cdot(Ny-2)\cdot(Nz-2)\cdot\frac{(\frac{t_{end}}{dt})}{runtime} \tag{5.3.1}$$

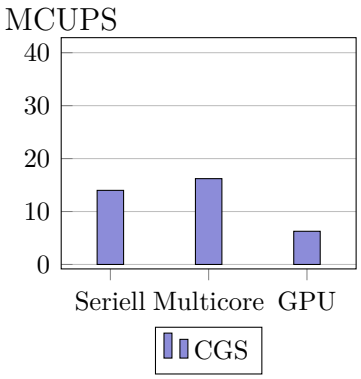
Laufzeitmäßig bedeutet das eine Verlangsamung. Dieser Überhang ließ sich nicht durch die Portierung auf die GPU erklären, da beim Jacobi Verfahren die gleiche Datenmenge übertragen wird. Des Weiteren lag es auch nicht an einer zu kleinen Problemgröße. Eine Verfeinerung des Gitters bewirkte nur noch eine größere Differenz zwischen den beiden Versionen.
Das Problem ließ sich letztlich beim Lesen und Schreiben auf das Datenarray lokalisieren. Bisher wurde auf die Daten mit Hilfe des Indexoperators `[ ]` zugegriffen.

```
Field *out;
d_out = out->data;
d_out[IX(i, j, k, nx, ny)]
```

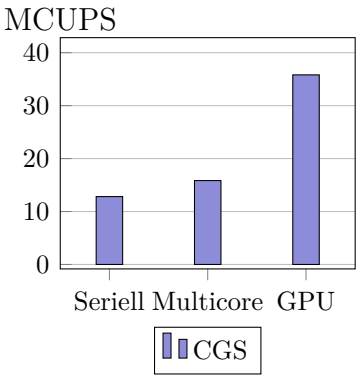
Durch die Nutzung des Indexoperators wird beim Schreibzugriff das gesamte Array gesperrt. Das bedeutet, dass die Schreibzugriffe seriell ausgeführt werden. Zusätzlich können in diesem Zeitraum keine Lesezugriffe von statten gehen. Die Sperre lässt sich umgehen, indem direkt mit Hilfe von Pointern auf die entsprechende Stelle im Array zugegriffen wird. Die Pointer werden berechnet, indem der Index auf den Anfangspointer des Arrays addiert wird. Die Schreibweise mit `d_out[value]` und `*(d_out + value)` sind äquivalent, das heißt sie greifen auf dasselbe Element vom Datenarray von Field zurück.

```
*(d_out + IX(i, j, k, nx, ny))
```

Durch die Berechnung der Pointer ändert sich nun die GPU Version nennenswert wie in Abbildung 5.6b zu erkennen ist. Mit dem Zugriff über die Pointer ist die GPU Version des CGS nun schneller als die serielle und die multicore Version, das bedeutet, dass sich die Parallelisierung an sich lohnt.



(a) Performanz bei Zugriff auf das Datenarray mit dem Indexoperator `[ ]`



(b) Performanz bei Zugriff auf das Datenarray mit Pointern

Abbildung 5.6.: Performanzvergleich zwischen zwei verschiedenen Zugriffsarten: Links wurde mit Hilfe des Indexoperators auf das Datenarray zugegriffen, rechts mit Hilfe von Pointern. Gittergröße: $522 \times 522 \times 3$

5.3.2. Vergleich: Jacobi-Verfahren und CGS

Zur Abschätzung, ob das GPU-parallelisierte CGS und das GPU-parallelisierte Jacobi-Verfahren läufzeitmäßig äquivalent sind, werden die beiden Algorithmen für verschieden feine Gitter getestet. Für jede Einstellung wurden fünf Durchläufe gestartet von denen das arithmetische Mittel gebildet werden sollte, um einen Durchschnittswert zu bekommen. Dabei fiel auf, dass es regelmäßig Werte gab, die nicht zu den anderen Werten passen (vgl. Abbildung 5.7). Für die gröberen Gitter (512 und 1024) traten keine Unregelmäßigkeiten auf, erst bei der Gittergröße $2048 \times 2048 \times 3$ kam es zu Verlangsamungen bis zu einem Faktor von 15.

45929	221992	169117	878694
45944	221975	169287	3502109
45938	222032	46226	223030
45978	222029	46223	223064
45974	222021	46259	223056
(a) $1024 \times 1024 \times 3$ (Jacobi)	**(b)** $1024 \times 1024 \times 3$ (CGS)	**(c)** $2048 \times 2048 \times 3$ (Jacobi)	**(d)** $2048 \times 2048 \times 3$ (CGS)

Abbildung 5.7.: Verschiedene Laufzeitwerte für den CGS und das Jacobi Verfahren

Diese Unregelmässigkeiten können daher kommen, dass die clock frequency der GPU nicht einheitlich hoch gesetzt wurde. Bei den gröberen Gittern kann dies auf Grund geringerer Rechenintensivität nicht ins Gewicht gefallen sein. Zu erwarten ist eine kleinere Streuung der Werte, jedoch wie sich zeigte, steht das nicht zwingend im Zusammenhang, da sich an der Streuung kaum etwas verändert hat (vgl. Abbildung 5.8). Allerdings hat der CGS eine Beschleunigung von einem Faktor größer 3 bekommen. Ein konkreter Grund für die Reduzierung der Laufzeit ließ sich nicht finden, allerdings besteht die Möglichkeit, dass während eines Jobs auf der JURECA Ressourcen entzogen beziehungsweise mehr Ressourcen zugesprochen wurden (je nach dem auf wie viele Jobs die GPU geteilt werden muss).

46106	222939	168025	874740
46127	3500914	168130	3492113
46138	60070	168060	59885
46143	59983	168084	59818
46104	59965	168120	59776
(a) $1024 \times 1024 \times 3$ (Jacobi)	**(b)** $1024 \times 1024 \times 3$ (CGS)	**(c)** $2048 \times 2048 \times 3$ (Jacobi)	**(d)** $2048 \times 2048 \times 3$ (CGS)

Abbildung 5.8.: Verschiedene Laufzeitwerte für den CGS und das Jacobi Verfahren mit festgesetzter Frequenz

Die Streuung der Werte erschwert den Vergleich zwischen den beiden Verfahren. In Abbildung 5.9 wurden die Werte mit der höchsten Auftrittswahrscheinlichkeit mit Festlegung der Frequenz aufgetragen, das arithmetische Mittel gebildet und graphisch dargestellt.
Das Jacobi-Verfahren ist dem CGS in allen Gittergrößen laufzeitmäßig überlegen. Allerdings gab es durchaus auch Fälle in denen das Jacobi-Verfahren deutlich langsamer war. Die Laufzeit für

die vier Gittergrößen ist annähernd konstant, weshalb es nahe liegt, dass die Problemgröße nicht groß beziehungsweise nicht komplex genug ist. Größere Gitter als $6000 \times 6000 \times 3$ wurden nicht mehr getestet auf Grund von Speicherallozierungsfehlern. Es besteht zudem die Möglichkeit, dass sich bei einer anderen Problemstellung die Verhältnisse von dem CGS und dem Jacobi Verfahren verändern.
Lediglich bezogen auf diese Problemstellung scheint der Jacobi Löser schneller und vor allem stabiler zu sein. Der CGS hatte eine größere Streuung bei den Laufzeitwerten und darunter auch die größeren Werte. Um einen angemessen Vergleich führen zu können, müsste als Grundlage ein anderer Rechner genommen werden bei dem die Ressourcen für die Berechnung fest eingestellt werden können (dies war bei JURECA nicht möglich). Zusätzlich sollten verschiedene Problemstellungen untersucht werden. Diese sollten eine unterschiedliche Anzahl an Iterationen benötigen (für Variation der Komplexität) und für diverse Gittergrößen getestet werden.

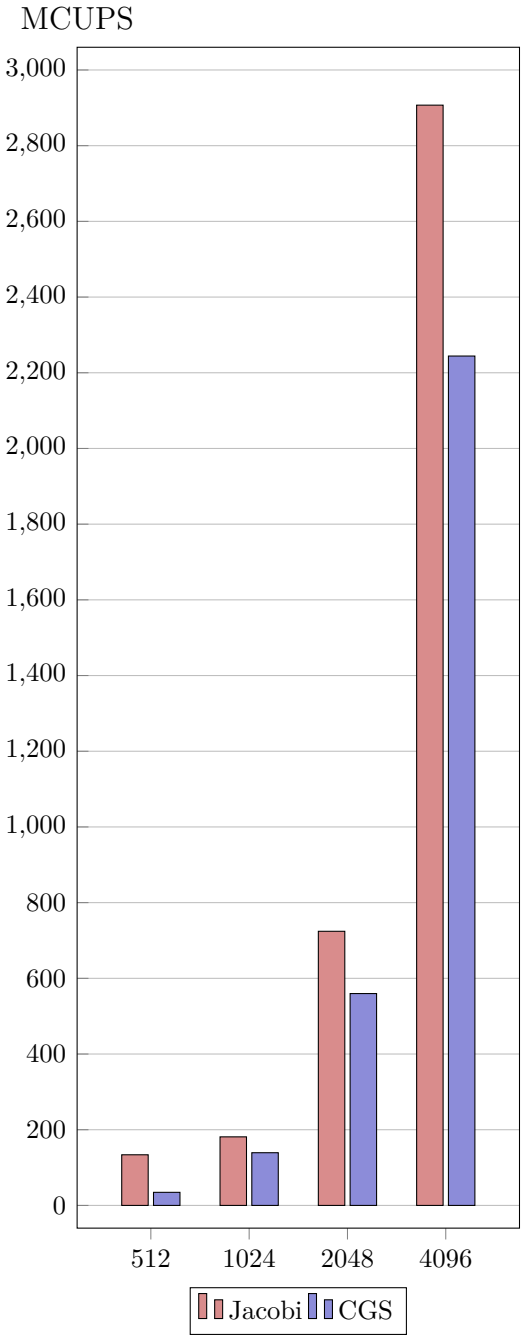


Abbildung 5.9.: Performanzvergleich zwischen dem CGS und dem Jacobi Verfahren für verschieden große Gitter. Nz = 3, Nx = Ny entspricht der Achsenbeschriftung.

6. Zusammenfassung

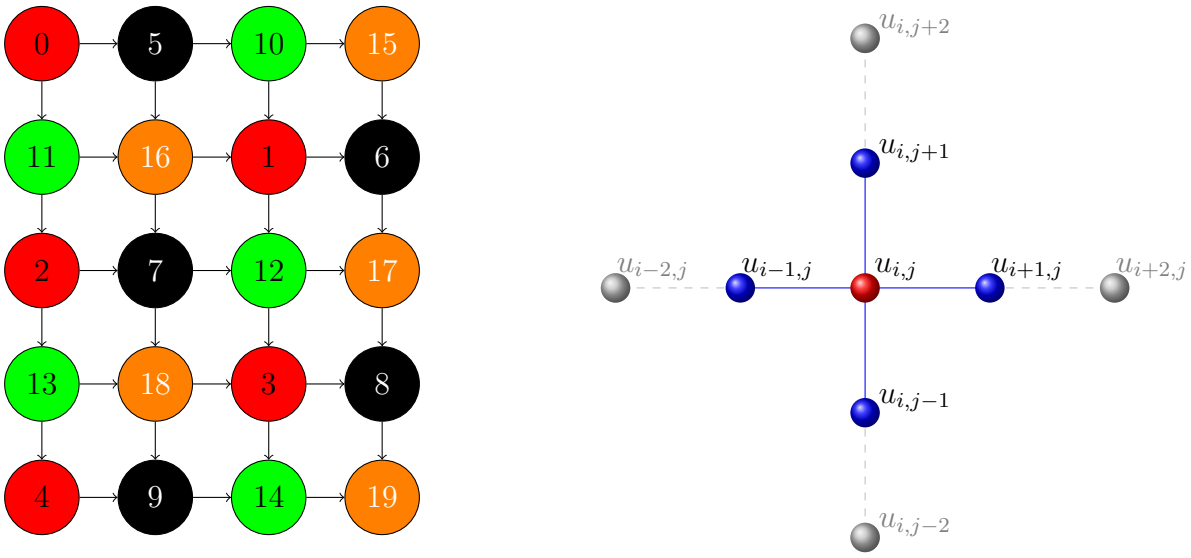
Es sollte eine Beschleunigung der Konvergenz bei der Berechnung von Diffusionsproblemen erreicht werden, indem das Jacobi durch das Rot-Schwarz Gauss-Seidel-Verfahren ersetzt wird. Im Rahmen der Bachelorarbeit war es möglich eine schnellere Konvergenz als das Jacobi-Verfahren zu erreichen, jedoch auf Kosten der Laufzeit. Nach [Dem96] müsste der CGS schneller konvergieren als das Jacobi-Verfahren bei gleicher Laufzeit, dies ließ sich jedoch weder bestätigen noch widerlegen auf Grund der großen Streuung bei der Laufzeitmessung.
Die Gewichtung des CGS wurde in dieser Arbeit nicht betrachtet und ist ein wichtiger Faktor bei der Beschleunigung der Konvergenz. Als Teil von JuROr wird die Parameteranalyse für ω weiter verfolgt.
Ebenfalls eine Vertiefung benötigt der Umgang mit OpenACC, da das vorhandende Wissen zu oberfächlich war, um eine optimale Parallelisierung zu ermöglichen. Mehrfach konnte eine Beschleunigung des CGS erreicht werden in dem lediglich andere Compiler Direktiven genutzt wurden. Zusätlich fanden sich im Laufe dieser Arbeit einige Stellen in JuROr, welche Optimierungsbedarf benötigen beziehungsweise deren Optimierung eine Beschleunigung des gesamten Programmes herbei führen würden. Dies erfordert jedoch tiefer greifende Änderungen, da diese (wie zum Beispiel die Änderung der Field Klasse oder Ooptimierung der Boundary-Methoden) die anderen Löser (Advektion und Druck) beeinflussen.
Zusammenfassend bietet die Parallelisierung des CGS innerhalb von JuROr noch viele nicht beachtete Möglichkeiten, welche sich zu vertiefen lohnen. Der CGS unterlag dem Jacobi-Verfahren im direkten Vergleich bei der Performanz, doch auf Grund der Optimierungsmöglichkeiten, welcher der NVIDIA Profiler aufgezeigt hat, bietet sich weitere Arbeit an dem Verfahren an. Erreicht werden konnte zumindest der Nachweis der schnelleren Konvergenz bei der Iterationsanalyse, welches positive Auswirkungen für das Multigrid Verfahren hat, in der die Druckgleichung mit einer festgelegten Anzahl an Iterationen gelöst wird.

7. Ausblick

7.1. Neun-Punkt Stern

In dieser Arbeit wurde der Fünf-Punkt Stencil (im deutschen geläufig als Fünf-Punkt Stern) für die Diskretisierung im zweidimensionalen Raum genutzt (analog der Sieben-Punkt Stencil im dreidimensionalen). Für eine höhere Ordnung kann der Neun-Punkt Stencil benutzt werden, welcher in Abbildung 7.1b abgebildet ist. Im Dreidimensionalen handelt es sich um einen 13-Punkt Stencil.

Daraus folgend ergibt sich eine Koeffizientenmatrix mit größerer Bandbreite. Auf Grund des größeren Stencil (Punkte der gleichen Farbe sind nicht mehr unabhängig von einander) senkt sich der Grad der Parallelisierung. In Folge desssen erhöht sich die Laufzeit, aber auch die Genauigkeit, welche eine Beschleunigung der Konvergenz mti sich führt. Als Gegenmaßnahme der Senkung der Parallelität können mehrere Farben genutzt werden.



(a) Vier-Farben Gauß-Seidel: Parallelisierung durch Neuanordnung mit vier Farben exemplarisch auf einem 5×4 Gitter

(b) Der Fünf-Punkt Stern (in Farbe dargestellt) mit Erweiterung auf den Neun-Punkt Stern (in grau)

Abbildung 7.1.: Erweiterungsmöglichkeiten der verwendeten Methoden

7.2. Vier-Farben Gauß-Seidel (Mehrfarben-Gauß-Seidel)

Anstatt sich wie in dieser Arbeit auf zwei Farben zu beschränken, können mehrere Farben genutzt werden. Ein weiterer bekannter Algorithmus der Mehrfarben-Technik ist der Vier-Farben Gauß-Seidel, für den die Farben rot, schwarz, grün und orange genommen werden. Sind die Knoten im Schema von Abbildung 7.1a gefärbt worden, kann ein Gauß-Seidel Schritt gemäß Gleichung (7.2.1) berechnet werden [SB89]:

$$x_{i,c}^{(k+1)} = T\left(x_{j<i,\hat{c}}^{(k+1)},\, x_{(j>i,\hat{c})}^{(k)}\right) \qquad \hat{c}, c = R, B, G, O;\ \hat{c} \neq c;\ i = 1, \ldots N/4 \tag{7.2.1}$$

c beziehungsweise $\hat{c}$ repräsentieren die benutzten Farben und N ist die Menge der zu berechnenden Parameter. T präsentiert den Übergangsoperator, welcher die zwei nachfolgenden Iterationslevel verbindet.
Der Vier-Farben Gauß-Seidel bietet sich beispielsweise für einen 13-Punkt Stern an oder für spezielle Stencils wie den 19-Punkt Stencil. Für diesen wird jedoch eine andere Farbreihenfolge der Knoten benötigt wie Abbildung 7.2 zu erkennen ist. Dieser Stencil ist auch mit der rot-schwarz Ordnung realisierbar. Näheres findet sich in [Zha98] und [GZ00].

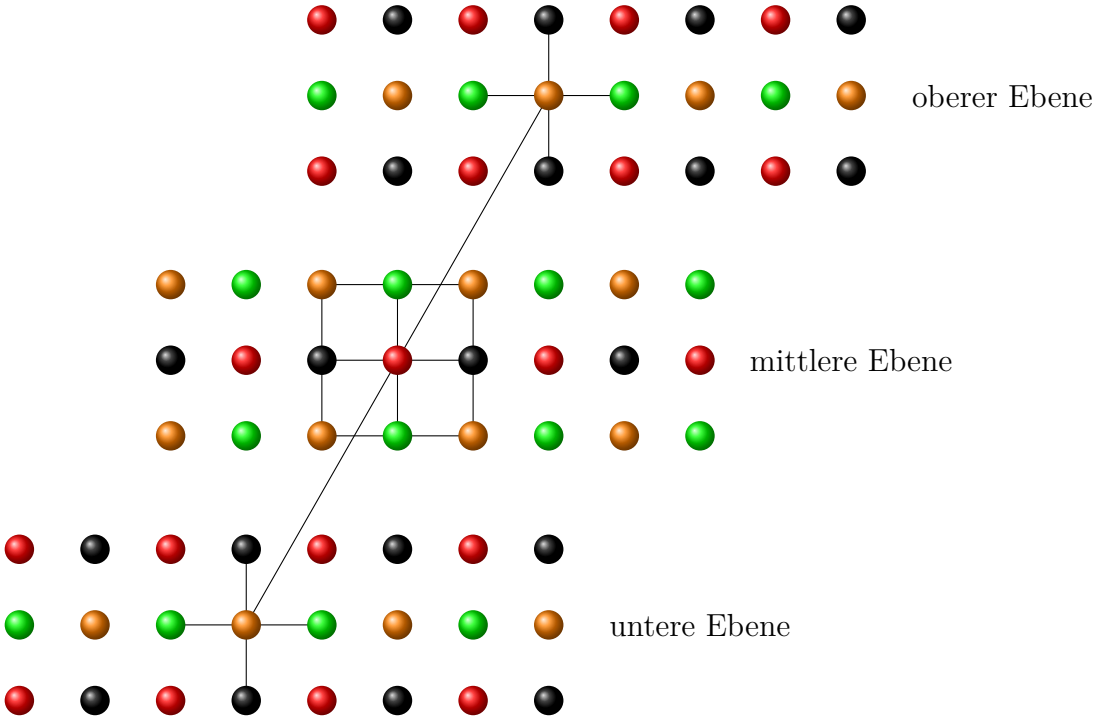


Abbildung 7.2.: 19-Punkt Stern mit vier Farben Gauß-Seidel [GZ00]

7.3. Explizite Parallelisierung

OpenACC war auf Grund des geringen Aufwandes leicht in JuROr einzubauen, jedoch besteht die Möglichkeit, dass bei expliziter Parallelprogrammierung ein besseres Ergebnis erreicht werden kann. Dies wäre mit wesentlich mehr Aufwand verbunden und würde sich vermutlich

nicht rentieren. Auf Grund der Einfachheit des Quellcodes (vorwiegend for-Schleifen) liegt es Nahe, dass der Compiler die optimale Lösung gefunden hat.

7.4. Höhere Ordnung im Zeitschritt

Momentan wird der Schritt in der Zeit mit Hilfe der impliziten Euler-Methode (Rückwärtsdifferenz) berechnet. Der implizite Euler besitzt die Ordnung $p = 1$. Bevor der Fehler im Raum minimiert wird, bietet es sich an, zuerst den Fehler in der Zeit zu minimieren. Ansonsten würde der Fehler im Raum zwar geringer werden, jedoch auch dessen Anteil am Gesamtfehler. Somit muss für eine höhere Genauigkeit sowohl der Fehler im Raum als auch in der Zeit verringert werden. Dies kann zum Beispiel durch die Nutzung der Runge-Kutta-Verfahren realisiert werden.

Anhang

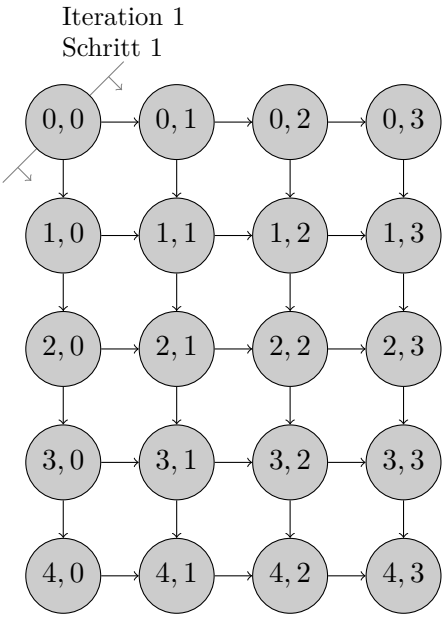
A. Beispiel für die Poisson Gleichung auf einem zweidimensionalen Gitter mit vorgegebenen Werten

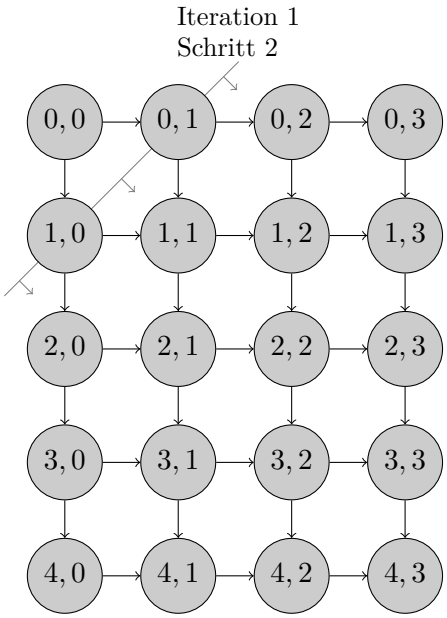
Für ein 5×4 Gitter in dem alle Randwerte bereits bekannt sind, ergibt sich für $A \cdot \vec{u} = \vec{b}$:

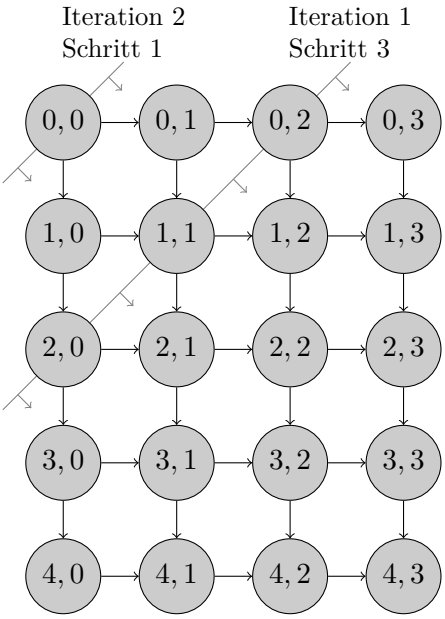
$$\begin{pmatrix} 4 & -1 & 0 & 0 & -1 & 0 \\ -1 & 4 & -1 & 0 & 0 & -1 \\ 0 & -1 & 4 & -1 & 0 & 0 \\ 0 & 0 & -1 & 4 & -1 & 0 \\ -1 & 0 & 0 & -1 & 4 & -1 \\ 0 & -1 & 0 & 0 & -1 & 4 \end{pmatrix} \cdot \begin{pmatrix} u_{1,1} \\ u_{1,2} \\ u_{2,1} \\ u_{2.2} \\ u_{3,1} \\ u_{3,2} \end{pmatrix} = \begin{pmatrix} -h^2 \cdot f_{1,1} + u_{0,1} + u_{1,0} \\ -h^2 \cdot f_{1,2} + u_{0,2} + u_{1,3} \\ -h^2 \cdot f_{2,1} + u_{2,0} \\ -h^2 \cdot f_{2,2} + u_{2,3} \\ -h^2 \cdot f_{3,1} + u_{3,0} + u_{4,1} \\ -h^2 \cdot f_{3,2} + u_{3,3} + u_{4,2} \end{pmatrix} \tag{A.0.1}$$

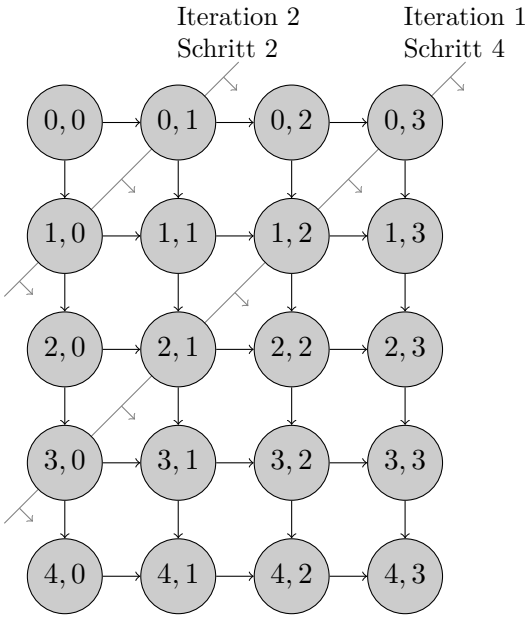
Die festgelegten Werte werden auf die rechte Seite der Nachbarknoten addiert, welche innere Knoten sind. Knoten wie $u_{0,0}, u_{0,3}, u_{4,0}$ und $u_{4,3}$ sind somit nicht Bestandteil des LGS.

B. Wellenfronten des Gauß-Seidel Algorithmus









C. Quellcode

C.1. ColoredGaussSeidel Methode: diffuse

```
1  void ColoredGaussSeidelDiffuse::diffuse(Field *out, Field *in, const Field *b, bool sync) {

2    auto bsize = out->GetSize();
3    Types type = out->GetType();
4    auto d_out = out->data;
5    auto d_b = b->data;

6    #pragma acc data present(out, d_out[:bsize], b, d_b[:bsize])
7    {
8            /* [...] intializing of variables for GPU  */
9       int it = 0;
10      real sum;
11      real res = 1.;

12      while (res > tol_res && it < max_it) {

13         ColoredGaussSeidelStep(out, b, alphaX, alphaY, alphaZ, beta, dsign, w, false);
14         this->boundary.applyBoundary(d_out, Nx, Ny, Nz, type, dx, dy, dz, false);
15         sum = 0;

16         #pragma acc kernels present(out, d_out[:bsize], b, d_b[:bsize]) async
17         FOR_EACH_INNER_CELL_ACC(Nx, Ny, Nz)
18            res = (- alphaX * (d_out[IX(i + 1,j,k,Nx,Ny)] + d_out[IX(i - 1,j,k,Nx,Ny)])\
19                   - alphaY * (d_out[IX(i,j + 1,k,Nx,Ny)] + d_out[IX(i,j - 1,k,Nx,Ny)])\
20                   - alphaZ * (d_out[IX(i,j,k + 1,Nx,Ny)] + d_out[IX(i,j,k - 1,Nx,Ny)])\
21                   +  rbeta *  d_out[IX(i, j, k, Nx, Ny)] -   d_b[IX(i, j, k, Nx, Ny)]);
22            sum += res * res;
23         END_FOR_ACC
24         #pragma acc wait

25         res = sqrt(sum);
26         it++;
27      }

28      if ( sync ) {
29         #pragma acc wait
30      }

31      #ifndef PROFILING
32         cout << "Number of iterations:" << it << endl;
33         cout << "Colored Gauss-Seidel ||res|| = " << res << "\n";
34      #endif
35   }
36 }
```

Listing C.1.1: Quellcode ColoredGaussSeidel.cpp – Ausschnitt der Methode diffuse(...): `FOR_EACH_INNER_CELL_ACC`, `END_FOR_ACC` und `IX(...)` sind benutzerdefinierte Macros der Klasse GlobalMacroTypes.h (siehe C.1.2)

```
1  #define FOR_EACH_INNER_CELL(Nx, Ny, Nz) for (int k=1; k<Nz-1; k++) {\
2          for (int j=1; j<Ny-1; j++) {\
3                  for (int i=1; i<Nx-1; i++) {
4  #define END_FOR                  }}}
5  #define IX(i,j,k,Nx,Ny) ((i) + (Nx)*(j) + (Nx)*(Ny)*(k)) // row-major index for
   ↪  one dimensional arrays (i=0..Nx-1 columns, j=0..Ny-1 rows, k = 0...Nz-1)
```

Listing C.1.2: Quellcode GlobalMacroTypes.h – Ausschnitt

C.2. ColoredGaussSeidel Methode: ColoredGaussSeidelStep

```
1  void ColoredGaussSeidelDiffuse::ColoredGaussSeidelStep(Field *out, const Field *b,
   ↪  const real alphaX, const real alphaY, const real alphaZ, const real beta,
   ↪  const real dsign, const real w, bool sync) {
2    int nx = out->GetNx();
3    int ny = out->GetNy();
4    int nz = out->GetNz();
5    int bsize = out->GetSize();

6    auto d_out = out->data;
7    auto d_b = b->data;

8    //red
9    #pragma acc kernels present(out, d_out[:bsize], b, d_b[:bsize]) async
10   {
11     #pragma acc loop independent collapse(3)
12     for (int k = 1; k < nz - 1; k += 2) {
13       for (int j = 1; j < ny - 1; j += 2) {
14         for (int i = 1; i < nx - 1; i += 2) {
15           ColoredGaussSeidelStencil(i, j, k, out, b, alphaX, alphaY, alphaZ, dsign,
   ↪  beta, w, nx, ny);
16         }
17       }

18       for (int j = 2; j < ny - 1; j += 2) {
19         for (int i = 2; i < nx - 1; i += 2) {
20           ColoredGaussSeidelStencil(i, j, k, out, b, alphaX, alphaY, alphaZ, dsign,
   ↪  beta, w, nx, ny);
21         }
22       }
23     }

24     #pragma acc loop independent collapse(3)
25     for (int k = 2; k < nz - 1; k += 2) {
26       for (int j = 1; j < ny - 1; j += 2) {
27         for (int i = 2; i < nx - 1; i += 2) {
28           ColoredGaussSeidelStencil(i, j, k, out, b, alphaX, alphaY, alphaZ, dsign,
   ↪  beta, w, nx, ny);
29         }
30       }

31       for (int j = 2; j < ny - 1; j += 2) {
32         for (int i = 1; i < nx - 1; i += 2) {
33           ColoredGaussSeidelStencil(i, j, k, out, b, alphaX, alphaY, alphaZ, dsign,
   ↪  beta, w, nx, ny);
34         }
35       }
36     }
37   }

38   #pragma acc wait
```

```
39    //black
40    #pragma acc kernels present(out, d_out[:bsize], b, d_b[:bsize]) async
41    {
42      #pragma acc loop independent collapse(3)
43      for (int k = 1; k < nz - 1; k += 2) {
44        for (int j = 1; j < ny - 1; j += 2) {
45          for (int i = 2; i < nx - 1; i += 2) {
46            ColoredGaussSeidelStencil(i, j, k, out, b, alphaX, alphaY, alphaZ, dsign,
      ↪  beta, w, nx, ny);
47          }
48        }

49        for (int j = 2; j < ny - 1; j += 2) {
50          for (int i = 1; i < nx - 1; i += 2) {
51            ColoredGaussSeidelStencil(i, j, k, out, b, alphaX, alphaY, alphaZ, dsign,
      ↪  beta, w, nx, ny);
52          }
53        }
54      }

55      #pragma acc loop independent collapse(3)
56      for (int k = 2; k < nz - 1; k += 2) {
57        for (int j = 1; j < ny - 1; j += 2) {
58          for (int i = 1; i < nx - 1; i += 2) {
59            ColoredGaussSeidelStencil(i, j, k, out, b, alphaX, alphaY, alphaZ, dsign,
      ↪  beta, w, nx, ny);
60          }
61        }

62        for (int j = 2; j < ny - 1; j += 2) {
63          for (int i = 2; i < nx - 1; i += 2) {
64            ColoredGaussSeidelStencil(i, j, k, out, b, alphaX, alphaY, alphaZ, dsign,
      ↪  beta, w, nx, ny);
65          }
66        }
67      }
68    }

69    #pragma acc wait
70  }
```

Listing C.2.1: Quellcode ColoredGaussSeidel.cpp – Methode ColoredGaussSeidelStencil

C.3. ColoredGaussSeidel Methode: ColoredGaussSeidelStencil

```
1  void ColoredGaussSeidelDiffuse::ColoredGaussSeidelStencil(int i, int j, int k,
   ↪  Field *out, const Field *b, const real alphaX, const real alphaY, const real
   ↪  alphaZ, const real dsign, const real beta, const real w, const int nx, const
   ↪  int ny) {

2    auto d_out = out->data;
3    auto d_b = b->data;

4    real d_out_x  = d_out[IX(i + 1, j, k, nx, ny)];
5    real d_out_x2 = d_out[IX(i - 1, j, k, nx, ny)];
6    real d_out_y  = d_out[IX(i, j - 1, k, nx, ny)];
7    real d_out_y2 = d_out[IX(i, j - 1, k, nx, ny)];
8    real d_out_z  = d_out[IX(i, j, k + 1, nx, ny)];
9    real d_out_z2 = d_out[IX(i, j, k - 1, nx, ny)];

10   real d_b = d_b[IX(i, j, k, nx, ny)];

11   real out_h = beta * (dsign * d_b
12       + alphaX * (d_out_x + d_out_x2)
13       + alphaY * (d_out_y + d_out_y2)
14       + alphaZ * (d_out_z + d_out_z2));

15   d_out[IX(i, j, k, nx, ny)] = (1 - w) * d_out[IX(i, j, k, nx, ny)] + w * out_h;
16 }
```

Listing C.3.1: Quellcode ColoredGaussSeidel.cpp – Methode ColoredGaussSeidelStencil

```
1  void ColoredGaussSeidelDiffuse::ColoredGaussSeidelStencil(int i, int j, int k,
   ↪  real *out, real *b, const real alphaX, const real alphaY, const real alphaZ,
   ↪  const real dsign, const real beta, const real w, const int nx, const int ny) {

2    real d_out_x  = *(out + IX(i + 1, j, k, nx, ny));
3    real d_out_x2 = *(out + IX(i - 1, j, k, nx, ny));
4    real d_out_y  = *(out + IX(i, j + 1, k, nx, ny));
5    real d_out_y2 = *(out + IX(i, j - 1, k, nx, ny));
6    real d_out_z  = *(out + IX(i, j, k + 1, nx, ny));
7    real d_out_z2 = *(out + IX(i, j, k - 1, nx, ny));

8    real d_b = *(b + IX(i, j, k, nx, ny));

9    real r_out = *(out + IX(i, j, k, nx, ny));

10   real out_h = beta * (dsign * d_b
11         + alphaX * (d_out_x + d_out_x2)
12         + alphaY * (d_out_y + d_out_y2)
13         + alphaZ * (d_out_z + d_out_z2));

14   *(out + IX(i, j, k, nx, ny)) = (1 - w) * r_out + w * out_h;
15 }
```

Listing C.3.2: Quellcode ColoredGaussSeidel.cpp – Methode ColoredGaussSeidelStencil mit Pointern

D. JURECA

JURECA (Jülich Research on Exascale Cluster Architectures) ist ein Supercomputer am Forschungzentrum Jülich, welcher im November 2015 in Betrieb genommen wurde. Es ist das Nachfolgemodell von JUROPA und gehört zur derzeit höchsten Leistungsklasse: der Petaflop-Klasse. Dies entspricht einer Rechenleistung von 2,2 Billiarden Operationen pro Sekunde.

D.1. Hardware Charakteristiken

1872 Rechenknoten

- Zwei Intel Xeon E5-2680 v3 Haswell CPUs pro Knoten
  - 2 x 12 cores, 2.5 GHz
  - Intel Hyperthreading Technology (Simultanes Multithreading)
  - AVX 2.0 ISA Erweiterung
- 75 Rechenknoten mit zwei NVIDIA K80 GPUs (vier Stück pro Knoten)
  - 2 x 4992 CUDA Kernen
  - 2 x 24 GiB GDDR5 Speicher

Entnommen von `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/Configuration/Configuration_node.html` (Stand: 19.07.2016)

Literatur

Artikel

[GZ00] Murli M. Gupta und Jun Zhang. „High accuracy multigrid solution of the 3D convection-diffusion equation“. In: *Applied Mathematics and Computation* 113 (2000), S. 249–274.

[SB89] Sauro Succi und Maurizio Benassi. „A Four-Color Parallel Algorithm for the Solution of a Two-Dimension Advection-Diffusion“. In: *Journal of Scientific Computing* 4 (1989), S. 61–70.

[Zha98] Jun Zhang. „Fast and High Accuracy Multigrid Solution of the Three Dimensional Poisson Equation“. In: *Journal of Comutational Physics* 143 (1998), S. 449–461.

Bücher

[FK59] D. A. Frank-Kamenetzki. *Stoff- und Wärmeübertragung in der chemischen Kinetik*. Springer Verlag, 1959. ISBN: 978-3-662-13055-1.

[Hac91] Wolfgang Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme*. Vieweg+Teubner Verlag, 1991. ISBN: 978-3-519-12372-9.

[Her11] M. Hermann. *Numerische Mathematik*. De Gruyter, 2011. ISBN: 978-3-486-71970-3.

[RR13] Thomas Rauber und Gundula Rünger. *Parallele und verteilte Programmierung*. Springer-Verlag, 2013. ISBN: 978-3-662-09196-8.

[Tip00] Paul A. Tipler. *Physik*. 3. Aufl. Heidelberg Berlin Oxford: Spektrum Akademischer Verlag, 2000.

[Zhu94] Jianping Zhu. *Solving Partial Differential Equations on Parallel Computers*. World Scientific, 1994. ISBN: 978-9-810-21578-1.

Skripte

[Dem96] Jim Demmel. „Applications of Parallel Computers“. 1996. URL: `https://people.eecs.berkeley.edu/~demmel/cs267/` (besucht am 20. Okt. 2016).

[Drz+14] Kevin Drzycimski, Lukas Arnold, Andreas Meunders, Benjamin Schröder und Corinna Trettin. „Fire Simulation. Exercise 5 - Computational Fluid Dynamics“. Skript der Universität Wuppertal. 2014.

[Moo07] Nial Moore. „Numerisches Lösen von partiellen Differenzialgleichungen“. 2007. URL: https://www.wi1.uni-muenster.de/pi/lehre/ss07/SeminarPP/ausarbeitungen/12-PDGL.pdf (besucht am 18. Apr. 2016).

[RG15] Martin Reißel und Matthias Grajewski. „Numerik 1“. Skript der FH Aachen. 24. März 2015.

[RW12] Thomas Richter und Thomas Wick. „Einführung in die Numerische Mathematik“. Skript der Universität Heidelberg. 30. Okt. 2012. URL: http://numerik.uni-hd.de/~lehre/SS12/numerik0/gesamt.pdf (besucht am 15. Apr. 2016).

[Wür16] My Linh Würzburger. „Untersuchung numerischer Methoden zur Wärmeleitgleichung“. Seminararbeit an der FH Aachen. 20. Jan. 2016.

Online

[16] Forschungszentrum Jülich. *ORPHEUS*. 2015. URL: http://www.orpheus-projekt.de (besucht am 24. Juni 2016).

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis