FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Code Generation vs. HPC Framework

Denis Ribica

Bachelorarbeit

Code Generation vs. HPC Framework

Denis Ribica

Bachelorarbeit

Aufgabensteller:	PD Dr. Ing. habil. Harald Köstler
Betreuer:	M. Sc. Sebastian Kuckuk, M. Sc. (hons) Christoph Rettinger
Bearbeitungszeitraum:	07.12.2017 – 07.06.2018

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 6. Juni 2018 .....

Abstract

Die Intension dieser Arbeit ist es, mit Hilfe der Lattice-Boltzmann-Methode einen Vergleich zwischen der automatischen Code Generierung und dem Gebrauch eines High-Performance Computing (HPC) Frameworks zu ziehen. Die Lattice-Boltzmann-Methode wird verwendet um numerische Strömungssimulationen durchzuführen. Da die Lattice-Boltzmann-Methode einen hohen Rechenaufwand benötigt, zeigt diese Arbeit mit Hilfe dieser zwei unterschiedlichen Herangehensweisen die verschiedenen Möglichkeiten zur Umsetzung und Implementierung der Lattice-Boltzmann-Methode und vergleicht die beiden Verfahren anschließend miteinander. Zuerst werden die Grundlagen der Lattice-Boltzmann-Methode erklärt. Zur genaueren Betrachtung der Lattice-Boltzmann-Methode werden die Testfälle des Poiseuille Channel Flow und der Lid-driven-Cavity Flow gewählt und erläutert. Nach einer Einführung in die domänenspezifische Sprache (DSL) ExaSlang, wird die Implementierung der Lattice-Boltzmann-Methode und der beiden Testfälle in ExaSlang aufgezeigt. Daraufhin folgt eine Einsicht in das HPC Framework waLBerla und die Implementierung der beiden Testfälle in Verbindung mit der Lattice-Boltzmann-Methode in waLBerla. Anschließend werden die verschiedenen Lösungswege nebeneinander gestellt und ausgewertet. Hierbei steht nicht allein die Performance der beiden Methodiken im Vordergrund, sondern ein Vergleich der Bedienbarkeit, Komplexität, Flexibilität und das Aufzeigen der Vor- und Nachteile dieser beiden Methoden.

Inhaltsverzeichnis

1 Einleitung

Bei der Lösung großer numerischer Probleme, dem einfachen Verarbeiten von großen Datenmengen oder dem Modellieren und Lösen von komplexen Systemen, kann ein einzelner Rechner schnell an seine Grenzen geraten. Um das Verhalten eines Fluids zu testen, werden aus Kosten- und Zeitgründen verschiedene Strömungen und Kollisionsmodelle durch Simulationen umgesetzt. Die Bewältigung dieser Aufgabenstellung in einer absehbaren Zeit führt zu einem hohen Bedarf an Rechenleistung.
Um den immer größer werdenden Rechenaufwand gerecht zu werden, entwickelte sich mit der Zeit das High-Performance Computing (HPC). Anstatt der Nutzung eines einzelnen Rechners, welcher beim Lösen dieser komplexen Aufgaben nicht mehr ausreichend war, begann man mehrere Rechner zu einem Rechnerverbund zu vernetzen. Dieser Rechnerverbund verfügt über eine erhöhte Rechnerkapazität und kann durch parallelisierte Arbeiten große Probleme effizienter lösen. Wenn man die Hardwarestruktur des Rechnerverbundes kennt, kann man die Software an diese Hardware anpassen. So kann man für einzelne Problemstellungen ein HPC Framework entwickeln. Dieses HPC Framework beinhaltet Spezifikationen und Optimierungen, die sich nach der Hardware richten und für eine noch bessere Performance sorgen sollen. Ein HPC Framework wird allerdings nicht nur verwendet um die Parallelisierung der Arbeitsschritte zu optimieren. Meist macht es auch Sinn eine anwendungsfreundliche Umgebung für den Benutzer zu schaffen, welcher dieses HPC Framework benutzen will, ohne die genauen Kenntnisse der Hardware und einzelnen Optimierungsschritte genau zu kennen. Die Nähe des HPC Frameworks zur Hardware ist nicht nur ihr größter Vorteil, sondern auch ein großer Nachteil. So muss die Software bei einem Hardwareaustausch neu überarbeitet und an die neuen Strukturen angepasst werden. Diese Anpassungen im Code sind meist recht zeitintensiv und verlangen vom Programmierer ein genaues Verständnis der Hardware und des HPC Frameworks, um die nötigen Änderungen im Code vorzunehmen. Anschließend müssen diese Anpassungen bezüglich der Korrektheit des Codes überprüft werden und hinsichtlich der Performance erneut getestet und eventuell wieder neu angepasst werden.
Eine andere Herangehensweise, um z.B. ein konkretes Problem der Strömungsmechanik effektiver simulieren zu können, ist die Verwendung einer geeigneten domänenspezifischen Sprache (DSL). Eine DSL ist eine Sprache, die an eine explizite Problemstellung angepasst ist und welche dann innerhalb einer bestimmten Domäne gewisse Probleme lösen kann. Durch die Adaption der Sprache an die Domäne und die Problemstellung lassen sich hier viele Vereinfachungen und Optimierungen durchführen, welche auch zu einer besseren Performance führen können. So können unter anderem Datenstrukturen vereinfacht werden und Routinen programmiert werden, welche auf diesen Datenstrukturen effizienter arbeiten. Anschließend wird mittels eines Compilers aus der DSL ein ausführbarer Code generiert. In diesem Fall wird der ExaStencils Compiler verwendet. Der hierbei generierte Code ist dann zusätzlich auch noch unabhängig von der Hardware und muss bei einem Hardwareaustausch nicht neu überarbeitet, sondern nur erneut mit Hilfe des Compilers übersetzt werden. Um nun die beiden oben genannten Methodiken miteinander vergleichen zu können, wurde hier die Lattice-Boltzmann-Methode zur Hilfe genommen. Die Lattice-Boltzmann-Methode ist eine Verfahren zur Lösung und Simulation von verschiedenen Strömungen und die Grundlage des hier angestrebten Vergleichs und muss deshalb zuerst genauer betrachtet werden. Als Testfälle für die Lattice-Boltzmann-Methode werden der Poiseuille Channel Flow und der Lid-driven-Cavity Flow [4] ausgewählt und genauer erläutert. Die nächste Aufgabe, die hier zu erledigen ist, besteht darin die Lattice-Boltzmann-Methode und die Testfälle in einer geeigneten DSL, in diesem Fall

ExaSlang, zu programmieren. Anschließend kann der Code generiert, ausgeführt und die Ergebnisse zum Vergleich ausgegeben werden. Die Vorgehensweise bei der Erstellung des Codes in ExaSlang und deren Funktionsweise ist einer der Hauptbestandteile dieser Arbeit. Ein weiterer Schwerpunkt, auf den anschließend genauer eingegangen wird, ist das HPC Framework waLBerla(widely applicable lattice Boltzmann from Erlangen) [8]. Dieses Framework verfügt bereits über Spezifikationen und Möglichkeiten zur Simulation der Lattice-Botzmann Methode. Hier besteht die Aufgabe darin, die beiden Testfälle in das HPC Framework einzubauen, in dem man bereits vorhandene Funktionen nutzt und diese an die Problemstellung anpasst. Nach diesen ganzen Vorbereitungen kann die Auswertung der zwei Lösungswege erfolgen. Ziel ist es hier nicht nur die Ergebnisse und die Performance zu vergleichen, sondern auch die Bedienbarkeit, Komplexität und Flexibilität der beiden Lösungswege genauer zu betrachten.

2 Die Lattice-Boltzmann-Methode

2.1 Allgemeine Theorie

Das FHP-Modell, auch als Lattice Gas Cellular Automata (LGCA) [9] bekannt, wurde 1986 von Uriel Frisch, Brosl Hasslacher und Yves Pompeau entwickelt und gilt als Vorreiter der Lattice-Boltzmann-Methode. Hierbei wird der Raum in Zellen diskretisiert. Auf den Gitterpunkten der Zellen befinden sich die Partikel, welche das Gas simulieren sollen. Die einzelnen Partikel wiederum können sich nur in eine diskrete Richtung bewegen. Der Zustand in einer Zelle ist boolisch, das heißt in diesem Fall, dass entweder ein Partikel existiert, welches in eine bestimmte diskrete Richtung strömt oder nicht. Jeder einzelne Zeitschritt kann dann wiederum in zwei Schritte unterteilt werden. Zuerst kommt der Strömungsschritt in dem sich die Partikel entsprechend ihrer Geschwindigkeit und Richtung in eine Nachbarzelle bewegen. Im zweiten Schritt, dem Kollisionsschritt, werden die Kollisionsregeln umgesetzt. Die Kollisionsregeln gewährleisten unter anderem die Massen- und Impulserhaltung, welche in einem geschlossenen System, wie es in dieser Arbeit auch der Fall ist, gelten müssen. Der Massenerhaltungssatz besagt, dass sich die Masse nicht spürbar verändert, wenn das System über keine Quellen oder Senken verfügt. Der Impulserhaltungssatz oder auch Impulssatz stellt sicher, dass in einem geschlossenen konstanten System, der Gesamtimpuls konstant ist. Aus dem LGCA lässt sich dann die Boltzmann Gleichung ableiten. Diese geht unter bestimmten Bedingungen und Grenzwerten in die Navier-Stokes Gleichungen über.
Die Lattice-Boltzmann-Methode kann man dann als Erweiterung des LGCA sehen. Hierbei ist die grundlegendste Änderung der Austausch der boolischen Zustände der Partikel, durch Partikelverteilungsfunktionen (PDFs). Die Lattice-Boltzmann-Gleichung [3], die sich daraus nun ableiten lässt, lautet[1]:

$$f_q(\vec{x}_j + \vec{c}_q \delta t, t + \delta t) - f_q(\vec{x}_j, t) = \Omega_q(\vec{x}_j, t) \qquad q = 0, ..., (Q-1), \tag{2.1}$$

f_q ist die q-te Komponente der PDF, $\vec{x}_j$ ist der Ort der j-ten Gitterzelle, $\vec{c}_q$ ist die q-te diskrete Gittergeschwindigkeit des Modells, δt ist die Zeitschrittlänge und Ω_q ist die q-te Komponente des Kollisionsterms. Q steht für die Anzahl an diskreten Gittergeschwindigkeiten und ist abhängig von der Wahl des Models. Da in dieser Arbeit ausschließlich 2D-Simulationen verwendet wurden, wird auch nur das $d2q9$-Modell vorgestellt. Auf das $d2q9$-Modell wird in der Abb. 2.1 noch einmal genauer eingegangen.

[1] Die Notation, die in dieser Arbeit benützt wird, ist von der Bachelorarbeit von Christoph Rettinger [6] und der Dissertation von D. Bartuschat [1] übernommen.

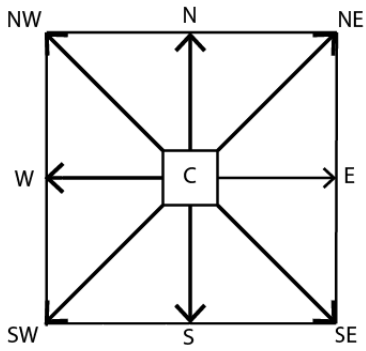

Abbildung 2.1: Das $d2q9$-Modell

In der $dDqQ$-Notation steht das D für die Dimension und das Q für die Anzahl der diskreten Geschwindigkeiten. Es gibt 4 diskrete Gittergeschwindigkeiten für die horizontalen und vertikalen Nachbarzellen sowie 4 diskrete Gittergeschwindigkeiten für die 4 diagonalen Zellen und eine diskrete Gittergeschwindigkeit im Zentrum. Die diskreten Gittergeschwindigkeiten für das $d2q9$-Modell lauten:

$$\vec{c}_q = \begin{cases} (0,0), & q \in \{C\} \\ (\pm c, 0), (0, \pm c) & q \in \{E, N, W, S\} \\ (\pm c, \pm c), & q \in \{NE, NW, SW, SE\} \end{cases} \tag{2.2}$$

c steht für die Gittergeschwindigkeit, welche sich aus der Division der Länge einer Zelle δx, durch die Dauer eines einzelnen Zeitschrittes δt ergibt. In diesem Fall setzen wir $\delta t = 1$ und $\delta x = 1$. Durch diese Vereinfachung erhält man für die Gittergeschwindigkeit $c = \frac{\delta t}{\delta x}$, sodass $c = 1$ ist.
Die Lattice-Boltzmann-Methode besteht nun darin, die Lattice-Boltzmann-Gleichung aus Gl. 2.1 in jedem Zeitschritt aufzulösen. Hierfür wird die Lattice-Boltzmann-Methode in zwei Teilschritte unterteilt, den Kollisionsschritt und den Strömungsschritt. Bevor auf die einzelnen Teilschritte genauer eingegangen werden kann, muss man zuerst die makroskopischen Eigenschaften, also die Dichte des Fluids und dessen Geschwindigkeit in x-Richtung und in y-Richtung betrachten. Diese Eigenschaften der PDFs, kann man durch die Lattice-Boltzmann-Gleichung in Gl. 2.1 als Momente der Geschwindigkeit aus den PDFs berechnen. Diese Operationen sind für jede Zelle lokal berechenbar. Für die makroskopische Dichte ϱ des Fluids in einer Zelle ergibt sich dann das folgende Moment 0-ter Ordnung:

$$\varrho(\vec{x}_j, t) = \sum_q f_q(\vec{x}_j, t) \tag{2.3}$$

Aus dem Moment 1-ter Ordnung für die Dichte kann man dann die Geschwindigkeit des Fluids einer Zelle ableiten. Das Moment 1-ter Ordnung für die Dichte ist folgendermaßen definiert:

$$\varrho(\vec{x}_j, t)\vec{u}(\vec{x}_j, t) = \sum_q f_q(\vec{x}_j, t)\vec{c}_q \tag{2.4}$$

Das Produkt aus der Dichte ϱ und der Geschwindigkeit $\vec{u}$ ergibt sich, durch das Aufsummieren der Produkte der einzelnen PDFs f_q mit den Gittergeschwindigkeiten der unterschiedlichen Richtungen.
Bei inkompressiblen Fluiden, wie sie in dieser Arbeit simuliert werden sollen, ist die Dichte ϱ nicht vom Druck abhängig. Stattdessen wird in den meisten Fällen und auch in dieser Arbeit für die Dichte $\varrho = 1$ angenommen, um ein inkompressibles Fluid zu beschreiben. Durch diese Vereinfachung kann man nun die Gl. 2.4 weiter auflösen und erhält dann für die Geschwindigkeiten in x-Richtung und y-Richtung:

$$\vec{u}(\vec{x}_j, t) = \sum_q f_q(\vec{x}_j, t)\vec{c}_q \tag{2.5}$$

Vektor $\vec{u}$ steht hierbei für die Geschwindigkeiten in x-Richtung und y-Richtung.
Im Testfall des Poiseuille Channels verändert sich die Berechnung der Geschwindigkeit durch die Einwirkung einer externen Kraft.

$$\vec{u}(\vec{x}_j, t) = \sum_q f_q(\vec{x}_j, t)\vec{c}_q + \frac{1}{2}\vec{a} \tag{2.6}$$

Die externe Krafteinwirkung wird hier durch den Vektor $\vec{a}$ dargestellt.
Nachdem die makroskopischen Eigenschaften nun bekannt sind, werfen wir einen Blick auf die zwei Teilschritte der Lattice-Boltzmann-Methode und wie diese makroskopischen Eigenschaften hierbei umgesetzt werden.

2.1.1 Kollisionsschritt

Zuerst wird der Kollisionsschritt durchgeführt. Hierbei werden die Kollisionsregeln umgesetzt. Die Formel, welche den Kollisionsschritt umsetzt lautet:

$$f_q^*(\vec{x}_j, t) = f_q(\vec{x}_j, t) + \Omega_q(\vec{x}_j, t) \tag{2.7}$$

Hier wird der Kollisionsterm $\Omega_q(\vec{x}_j, t)$ zu den PDFs $f_q(\vec{x}_j, t)$ addiert. Das aktualisierte Ergebnis zeigt die PDFs nach dem Kollisionsschritt $f_q^*(\vec{x}_j, t)$.
Der Kollisionsterm, der in der Gl. 2.7 dazu addiert wird, spielt eine entscheidende Rolle bei der Lattice-Boltzmann-Methode [2]. In dieser Arbeit wird ausschließlich der Bhatnagar–Gross–Krook (BGK) Operator als Kollisionsterm verwendet. Der BGK Operator lautet:

$$\Omega = -\frac{1}{\tau}(\vec{f} - \vec{f}^{eq}) \tag{2.8}$$

τ ist die Relaxionszeit und muss im Intervall von $[0, 2]$ liegen. Der Kollisonsterm ergibt sich

durch Multiplikation von $-\frac{1}{\tau}$ mit der Differenz aus den PDFs $\vec{f}$ und der Equilibriumsfunktion $\vec{f^{eq}}$, welche aus der Maxwell-Boltzmann-Verteilung gefolgert werden kann. In dieser Arbeit wird ausschließlich das Single-Relaxation-Time (SRT) Modell verwendet, somit gibt es nur eine Relaxationszeit. Die Relaxationszeit τ wiederum hängt wie folgt direkt mit der Viskosität v des Fluids zusammen:

$$v = \frac{1}{3}(\tau - \frac{1}{2}) \tag{2.9}$$

Die Equilibriumsfunktion selbst stellt sich dann folgendermaßen dar:

$$f_q^{eq}(\varrho, \vec{u}) = w_q \varrho(1 + \frac{1}{c_s^2}(\vec{c_q} \cdot \vec{u}) + \frac{1}{2c_s^4}(\vec{c_q} \cdot \vec{u})^2 - \frac{1}{2c_s^2}(\vec{u} \cdot \vec{u})) \tag{2.10}$$

$\vec{u}$ steht weiterhin für die Geschwindigkeit des Fluids,ϱ für die Dichte und $\vec{c_q}$ für die Gittergeschwindigkeit. Die Schallgeschwindigkeit c_s erhält man durch folgende Gleichung:

$$c_s = \frac{1}{\sqrt{3}}c \tag{2.11}$$

Die Schallgeschwindigkeit c_s ist somit abhängig von der Gittergeschwindigkeit c. Der Druck $p$wiederum hängt dann folgendermaßen direkt mit der Schallgeschwindigkeit und dem Druck zusammen:

$$p(\vec{x}_j) = c_s^2 \varrho(\vec{x}_j) \tag{2.12}$$

Die Gewichtungen w_q der q unterschiedlichen Richtungen lauten für das $d2q9$-Modell:

$$w_q = \begin{cases} \frac{4}{9}, & \text{für } q \in \{C\} \\ \frac{1}{9}, & \text{für } q \in \{E, N, W, S\} \\ \frac{1}{36}, & \text{für } q \in \{NE, NW, SW, SE\} \end{cases} \tag{2.13}$$

Um den Strom mit einer externen Kraft anzutreiben, wie es im Testfall des Poiseuille Channels auftritt, muss dem Kollisionsschritt aus Gl. 2.7 eine Kraftkomponente hinzugefügt werden:

$$f_q^* = f_q - \omega(\vec{f} - \vec{f^{eq}}) + 3w_q \varrho \vec{e}_q \cdot \vec{a} \tag{2.14}$$

$\vec{a}$ steht hierbei für den Kraftvektor, welcher auf die Zellen einwirkt. Die y-Komponente des Vektors ist null, da nur eine Krafteinwirkung in x-Richtung auftreten soll.

2.1.2 Strömungsschritt

Beim Strömungsschritt wird die Strömung der Partikel in eine Nachbarzelle umgesetzt. Betrachtet man nun die Lattice-Boltzmann-Gleichung Gl. 2.1 repräsentiert der linke Teil der Gleichung die Wanderung der Partikel, also in unserem Fall die PDFs f_q in der Gitterrichtung c_q. Aus der Gl. 2.2 kann man erkennen, dass die PDFs f_q pro Zeitschritt nur in die direkt anliegenden Nachbarzellen strömen können. Daraus ergibt sich für den Strömungsschritt folgende Gleichung:

$$f_q(\vec{x}_j + \vec{c}_q, t + \delta t) = f_q^*(\vec{x}_j, t) \tag{2.15}$$

f_q^* steht hier für die PDF nach dem Kollisionsschritt. Zur Verdeutlichung der Wanderung der PDFs folgt in Abb. 2.2 ein Beispiel dafür, in welche Nachbarzellen sich die PDFs nach einem Strömungsschritt befinden.

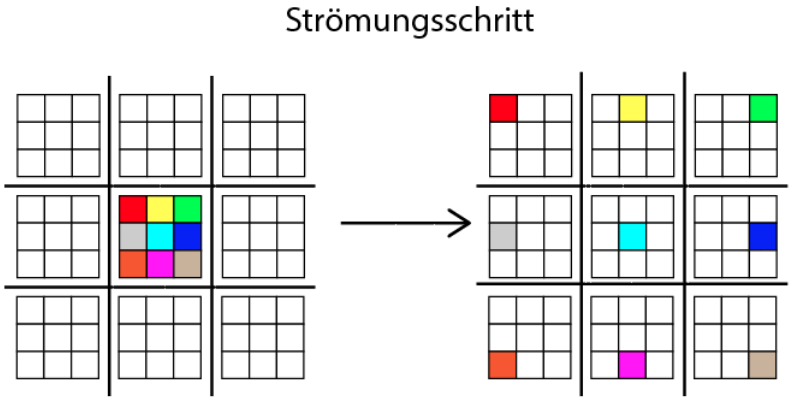

Abbildung 2.2: Strömungsschritt

2.2 Randbehandlung

Die Randbehandlung bei der Implementierung von unterschiedlichen Modellen und Testfällen ist auch bei der Lattice-Boltzmann-Methode ein entscheidender Faktor. So benötigt man bei dem Strömungsschritt, in dem die PDFs einer Zelle in die unterschiedlichen anliegenden Nachbarzellen weiter strömen, bei einem $d2q9$-Modell die 9 anliegenden Nachbarzellen. Falls wir uns jedoch neben einer der Randzellen befinden, müssen wir diese vorerst extra behandeln. Die Umsetzung der Randbehandlungen erfolgt mit Hilfe eines Ghostlayers. Die verschiedenen Testfälle benötigen allerdings nicht nur ein Ghostlayer zur Umsetzung der Lattice-Boltzmann-Methode, sondern auch ein unterschiedliches Verhalten an den Randzellen, um die verschiedenen Strömungen zu simulieren. In dieser Arbeit werden ausschließlich die zwei verschiedenen Randbehandlungen der No-Slip-Randbedingung und der periodischen Randbedingung verwendet, um die zwei Testfälle des Poiseuille Channels und der Lid-Driven-Cavity zu simulieren.

2.2.1 No-Slip-Randbedingung

Eine No-Slip-Randbedingung bedeutet, dass die Geschwindigkeit des Fluids an den Rändern gleich null ist. Die PDFs, welche direkt neben den Rändern liegen und in Richtung der Ränder zeigen, strömen gegen die Wand und werden von dieser ohne Verlust bei der Kollision reflektiert. Bei der Implementierung eines konkreten Falles bedeutet dies, dass z.B. die Partikel einer Zelle, welche in die Nordwest-Richtung fließen, im nächsten Zeitschritt in derselben Zelle in die Südost-Richtung fließen. Siehe Abb 2.3 für eine Illustration der Randbedingung.

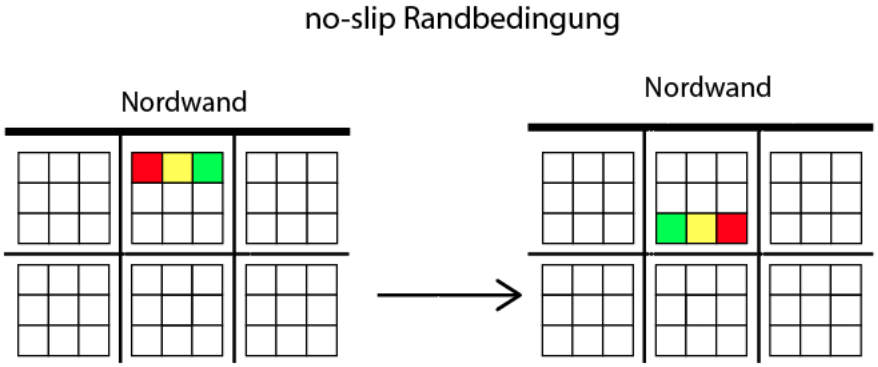

Abbildung 2.3: No-Slip-Randbedingung

2.2.2 Periodische Randbedingung

Die Periodische Randbedingung ist die zweite Randbehandlung, welche hier vorgestellt wird und kommt nur in dem Testfall des Poiseuille Channels vor. Hierbei strömen die Partikel einer Zelle, an dem Rand an dem die Periodische Randbedingung vorkommt, in die Randzelle des gegenüberliegenden Randes. Durch diesen Effekt wird eine Periodizität erzeugt. Ein konkretes Beispiel zur Verdeutlichung der periodischen Randbehandlung folgt in der Abb. 2.4.

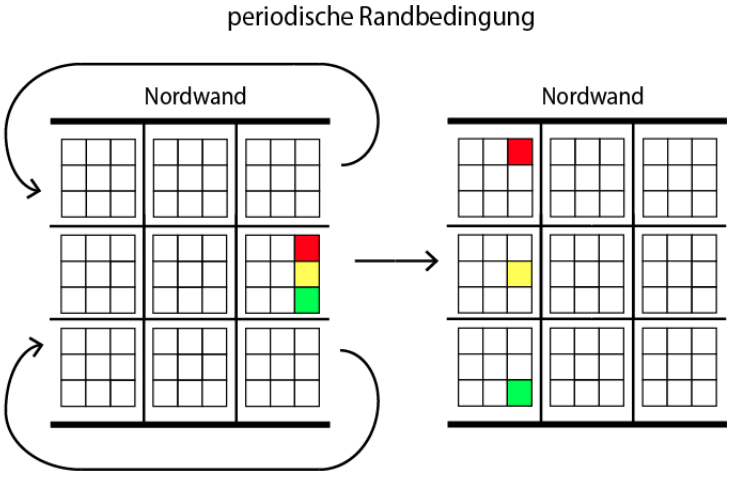

Abbildung 2.4: Periodische Randbedingung

3 Testfälle

Um die Implementierung der Lattice-Boltzmann-Methode zu validieren, benötigen wir einen Testfall. Dieser wird anschließend in ExaSlang und in waLBerla umgesetzt und simuliert. Der Testfall wird während der Ausführung graphisch überprüft und zur genaueren Auswertung in Ausgabedateien geschrieben. Der Poiseuille Channel und die Lid-Driven-Cavity werden hier als Testfälle behandelt. Da beide Testfälle den Strom unterschiedlich antreiben, interessiert uns die genaue Umsetzung dieser beiden Testfälle.

3.1 Poiseuille Channel

Da in dieser Arbeit ausschließlich 2D-Simulationen durchgeführt werden, wird der Poiseuille Channel Testfall auch nur mit zwei Dimensionen getestet. Der Poiseuille Channel ist dann ein Kanal mit einer gegebenen Höhe unendlichen Länge. Durch den Kanal fließt das Fluid in diesem Fall von Westen nach Osten. Ein Beispiel für einen Poiseuille Channel folgt in Abb. 3.1.

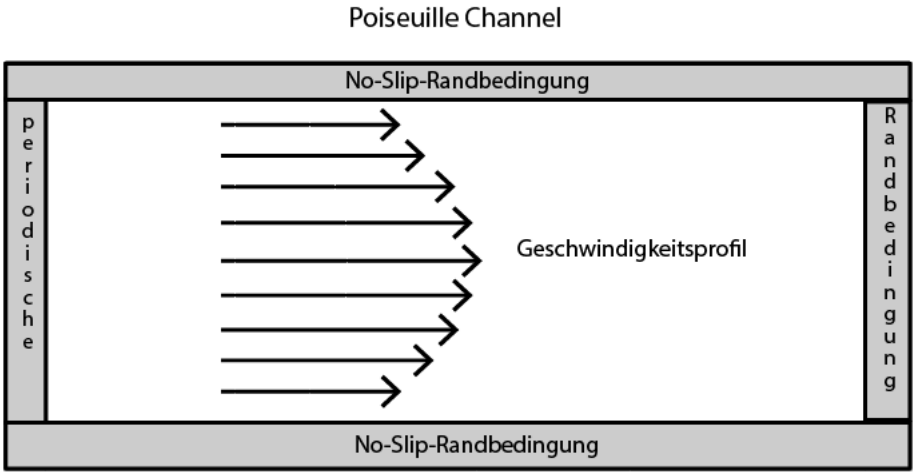

Abbildung 3.1: Poiseuille Channel

Die entscheidenden Kriterien sind die Randbehandlung, das Verfahren um den Strom anzutreiben und die Umsetzung in ExaSlang und in waLBerla. An dem nördlichen und südlichen Rand haben wir eine No-Slip-Randbedingung und am westlichen und östlichen Rand eine periodische Randbedingung. Die No-Slip-Randbedingung und die periodische Randbedingung wurden in Kapitel 2.2. bereits genauer erläutert. Um den Strom anzutreiben wird für den Poiseuille Channel eine zusätzliche externe Krafteinwirkung benötigt. Diese Krafteinwirkung wirkt in x-Richtung auf das Fluid ein und versetzt es somit in Strömung.

3.2 Lid-Driven-Cavity

Der Lid-Driven-Cavity Testfall ist eine Box gefüllt mit einer Flüssigkeit. An dem West-, Süd- und Ostrand haben wir eine No-Slip-Randbehandlung. Der Nordrand weißt dagegen eine konstante Geschwindigkeit von Westen nach Osten in der x-Richtung auf, welche die Flüssigkeit im Inneren zum Zirkulieren bringt. Ein Beispiel für den Lid-Driven-Cavity Testfall findet man nachfolgend in der Abb. 3.2.

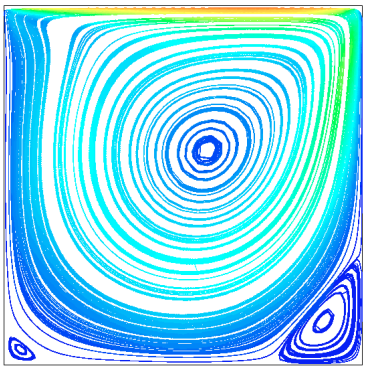

Abbildung 3.2: Lid-Driven-Cavity mit einer zirkulierenden Flüssigkeit, welche durch die anliegende konstante Geschwindigkeit am Nordrand hervorgerufen wird[3]

[3] http://www.nacad.ufrj.br/~rnelias/gallery/cavity.html

4 Code Generierung

Oft wird ein Code über Jahre von Hardwarespezialisten an die Hardware angepasst und durch das Wissen über die Hardware optimiert. Durch die vielen Spezifikationen lässt sich die Performance stark steigern. Um das notwendige Wissen über die Hardware in Erfahrung bringen zu können, dauert es allerdings oft Jahre. Wenn die Hardware ausgetauscht wird oder gleich versucht wird den Code auf einem anderem HPC Cluster zu übersetzen, muss der Code erst überarbeitet werden. Der Code muss an vielen Stellen an die neue Hardware angepasst und erneut optimiert werden. Dies wiederum erfordert ebenfalls einen großen Zeitaufwand. Eine Mögliche Lösung für dieses Problem ist die Verwendung einer DSL um den Algorithmus und die Implementation voneinander zu trennen. Mit Hilfe einer DSL ist es möglich gewisse Probleme innerhalb einer Domäne einfacher und effizienter zu lösen. So können benötigte Datenstrukturen oder Algorithmen speziell auf die Problemstellung angepasst werden. Anschließend wird mit einem DSL Compiler Programmcode generiert. Bei einem Hardwareaustausch genügt es dann mit Hilfe des Compilers den Code neu zu generieren. Durch die vielen Spezifikationen sowie der abgegrenzten Domäne wird schnell ersichtlich, was innerhalb dieser Domäne möglich ist und was erforderlich ist, um z.B. ein neues Modell innerhalb dieser Domäne einzuführen.

4.1 ExaSlang

ExaSLang ist eine DSL, welche entwickelt wurde um partielle Differentialgleichungen mittels Mehrfachgittern numerisch zu lösen. [7] Die ExaSlang Struktur ist in mehrere Layer eingeteilt. Layer 1 ist das abstrakteste Layer und dient zur Lösung von partieller Differentialgleichungen. Dieses Layer stellt ein eine kontinuieliche Domäne dar und soll die Schnittstelle für Anwender sein, welche nur wenig Programmierkenntnisse besitzen. Layer 2 ermöglicht eine Diskretisierung der Domäne und der Modelle und ist somit bereits weniger abstrakt. Im Layer 3 werden zusätzlich algorithmische Strukturen und Parameter zur Verfügung gestellt und sorgen für eine noch konkretere Darstellugn. Layer 4 ist die konkreteste Ebene. Hier können zusätzlich noch Paralleliserungen und genauere Spezifikationen im Programm durchgeführt werden. Diese Arbeit wurde ausschließlich auf der Ebene des Layer 4 umgesetzt. Eine weitere Besonderheit von ExaSlang ist die Verwendung von unterschiedlichen Leveln. Die Level helfen bei der Abbildung der Funktionalitäten und der Gitterstruktur. Mit Hilfe von Schlüsselwörten wie **finest** oder **coarsest** kann dann innerhalb der Gitterstruktur auf die unterschiedlichen Granularitäten zugregriffen werden. Die genauen Datenstrukturen , welche ExaSlang bietet und welche in dieser Arbeit benötigt werden, werden im nächsten Kapitel genauer vorgestellt. Bevor der ExaSlang Code mittels des ExaStencils Compiller [5] kompiliert werden kann, muss dieser in ExaSlang implementiert werden.

4.2 Implementierung in ExaSlang

Um die Lattice-Boltzmann-Methode in ExaSlang zu implementieren, hilft es, sich zuerst eine mögliche Implementierung in einer bereits bekannten Programmiersprache anzusehen. Da waLBerla in der Programmiersprache C++ zur Verfügung steht und auch ExaSlang C++ Code generiert, wurde hier zuerst eine Implementierung der Lattice-Boltzmann-Methode in C++ vorgenommen mit dem Ziel die notwendigen Datenstrukturen, Funktionen und andere wichtige Aspekte bei der Implementierung der Lattice-Boltzmann-Methode frühzeitig zu erkennen. Nach der Identifizierung der genauen Datenstrukturen, welche bei der Implementierung der Lattice-Boltzmann-Methode benötigt werden, erfolgt eine genauere Analyse der DSL ExaSlang, um nach Möglichkeiten zu suchen genau diese Datenstrukturen in ExaSlang umzusetzen. Hierbei werden nun auch die Unterschiede zwischen den Programmiersprachen und deren Vor- und Nachteile deutlicher. So ist zum Beispiel die Iteration über einen der 4 GhostLayer bei der Randbehandlung der Testfälle in ExaSlang deutlich leichter programmierbar und die Möglichkeit einen Indexfehler zu begehen verringert sich enorm.
Als erstes betrachten wir nun die verwendeten Datenstrukturen in ExaSlang. Als Simulationsdomäne wird ein 50x50 Gitter gewählt. Es gibt 50^2 Zellen, welche unser gesamtes Feld und jeweils die einzelnen Testfälle darstellen. Hierfür wird in ExaSlang ein Layout angelegt mit:

```
Layout ScalarCell < Real, Cell > {
        innerPoints      = [ 50, 50 ]
        duplicateLayers  = [ 0, 0 ]
        ghostLayers      = [ 1, 1 ] with communication
}
```

Programmausdruck 4.1: Layout

Wie im Programmausdruck 4.1 zu erkennen ist, wird hier die Simulationsdomäne erstellt und zum anderen ein Ghostlayer für unseren Testfall gesetzt. Durch den Ausdruck
with communication wird der Datenaustausch, welcher bei einer verteilten Parallelisierung erforderlich ist, vorbereitet. Da die Teilschritte aus der Lattice-Boltzmann-Methode auf lokalen Zellen ausgeführt werden, kann man das ganze Gitter in mehrere einzelne Gitter unterteilen und auf verschiedenen Prozessoren getrennt und parallel verarbeiten. Die verschiedenen Prozesse wiederum müssen dann miteinander kommunizieren um z.B. die berechneten Ergebnisse an den Trennstellen des Untergitter miteinander auszutauschen. Den Ghostlayer muss man sich als eine Struktur von Helferzellen vorstellen, welche unsere eigentlichen Gitterzellen umranden und für die spätere Randbehandlung benötigt werden. Wenn man sich das Feld als ein zweidimensionales Gitter vorstellt, erweitert der Ghostlayer dieses um zwei Zeilen und zwei Spalten. Am Nord- und Südrand wird jeweils eine weitere Zeile mit Zellen angefügt und am West- und Ostrand eine weitere Spalte mit Zellen. In der Abb. 4.1 wird eine mögliche Darstellung des Ghostlayers und der Innenzellen aufgezeigt.

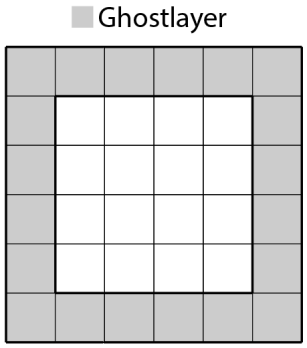

Abbildung 4.1: Ghostlayer

Nun können wir 9 Felder für die verschiedenen PDFs anlegen. Stellt man sich unser 50x50-Gitter vor, so ist jeder Punkt des Gitters eine Zelle und jede Zelle wiederum enthält 9 Felder, um die PDFs einer Zelle zu speichern. Da für spätere Berechnungen alte Werte aus dem Feld benötigt werden, benötigt man die ganze Feldstruktur ein zweites Mal, um die neuen Werte zu speichern ohne die alten Werte zu überschreiben. Als Beispiel für ein Feld kann man das Nordwestliche Feld zur Speicherung der diskreten Geschwindigkeiten betrachten, welches in ExaSlang wie folgt definiert wird:

```
Field Pdf_NW < global , ScalarCell , None >[2]
```

Programmausdruck 4.2: Felddeklaration

Mit Hilfe des in Programmausdruck 4.1 definierten Layouts wird hier nun die Feldstruktur für das nordwestliche Feld definiert. Als Randbehandlung wurde mit **None** keine spezielle Randbehandlung angegeben, da wir diese später selbst implementieren. Die Domäne auf der wir ausschließlich arbeiten ist global. Die Feldstruktur wird hier zweimal angelegt um bei den Teilschritten die noch benötigten PDFs nicht zu überschreiben, sondern in die zweite Feldstruktur zu schreiben und im Anschluss dann richtig zuzuweisen.
Für die Dichte des Fluids wir ein Feld erstellt, welches für jede Zelle einen skalaren Wert speichert. Die Geschwindigkeiten des Fluids einer Zelle werden bei jeder Berechnung temporär in einem Vektorfeld gespeichert, welche die Geschwindigkeit in x-Richtung und y-Richtung beinhaltet.
Die Gewichtung der diskreten Richtungen in einer Zelle werden in ExaSlang mit Hilfe von konstanten Variablen angelegt. Ein Beispiel für die Gewichtung der Nordwestrichtung lautet:

```
Val latticeWeight_NW : Real = 1./36.
```

Programmausdruck 4.3: Konstante Variablen

Hier wird die Gewichtung der nordwestlichen Richtung in der konstanten Variablen latticeWeight_NW als reelle Zahl mit dem Wert $\frac{1}{36}$ angelegt.
Weitere wichtige Variablen wie z.B. die Relaxationszeit τ oder die Anzahl der benötigten Zeitschritte, werden als Variablen angelegt, da man diese eventuell im generierten Code noch ändern und anpassen will.

```
Var tau : Real = 1.9
```

Programmausdruck 4.4: Variablen

Im Programmausdruck 4.4 wird die Variable τ mit dem reellen Wert 1.9 initialisiert. Neben Variablen und dem Layout benötigt man zur Umsetzung der Lattice-Boltzmann-Methode in ExaSlang Funktionen, wie es in anderen Sprachen wie C++ auch üblich ist. Da bei der Lattice-Boltzmann-Methode die Werte der PDFs den Gewichtungen entsprechen sollen, wird zu erst mit Hilfe einer eigenen Funktion diese Initialisierung folgendermaßen durchgeführt.

```
Function InitFields@finest ( ) {
        loop over Pdf_C {
                Pdf_C  = latticeWeight_C
                Pdf_N  = latticeWeight_N
                Pdf_E  = latticeWeight_E
                Pdf_S  = latticeWeight_S
                Pdf_W  = latticeWeight_W
                Pdf_NE = latticeWeight_NE
                Pdf_SE = latticeWeight_SE
                Pdf_SW = latticeWeight_SW
                Pdf_NW = latticeWeight_NW
        }
}
```

Programmausdruck 4.5: Funktionen

Die Funktion wird InitFields genannt und soll nur auf dem feinsten Level, gekennzeichnet durch @**finest**, die Initalisierung der PDFs mit den Gewichtungen der Richtung durchführen. Da wir in dieser Arbeit immer auf dem gleichen Level arbeiten, interessieren uns die restlichen Level nicht.

4.2.1 Randbehandlung

Bevor wir die zwei Teilschritte der Lattice-Boltzmann-Methode implementieren können, muss die Randbehandlung der jeweiligen Testfälle in jedem Zeitschritt durchgeführt werden. Da beide Testfälle eine unterschiedliche Randbehandlung besitzen, ist es notwendig diese Randbehandlungen genauer zu betrachten.

Randbehandlung im Testfall Poiseuille Channel

Im Testfall des Poiseuille Channels gilt es am Nord- und Südrand die No-Slip-Randbedingung und am West- und Ostrand eine periodische Randbedingung zu implementieren. Die No-Slip-Randbedingung am Nordrand sieht in ExaSlang wie folgt aus:

```
loop over Pdf_S only ghost [1, 0] on boundary{
        Pdf_SW  = Pdf_NE@[-1, -1]
        Pdf_S   = Pdf_N @[ 0, -1]
        Pdf_SE  = Pdf_NW@[ 1, -1]
}
```

Programmausdruck 4.6: No-Slip-Randbedingung am Nordrand

Um die No-Slip-Randbedingung am Nordrand durchzuführen, wird nur der Nordrand des Ghostlayers mit **only ghost** [1, 0] in einer Schleife durchlaufen und die PDFs an die entsprechenden Stellen geschrieben. Bei Unklarheiten diesbezüglich, kann die Abb. 2.3 nochmals eingesehen werden.
Die zweite Randbedingung im Testfall des Poiseuille Channels ist die periodische Bedingung. Diese ist bereits in ExaSlang umgesetzt und muss somit nur wie folgt ausgewählt werden.

```
Knowledge {
        domain_rect_periodic_x = true
}
```

Programmausdruck 4.7: Periodische Bedingung in x-Richtung

Hier wird im knowledge Block die Periodizität in X-Richtung eingestellt.

Randbehandlung im Testfall Lid-Driven-Cavity

Im Testfall Lid-Driven-Cavity gilt am West-, Ost- und Südrand die No-Slip-Randbedingung, wie sie bereits im Testfall des Poiseuille Channels implementiert worden ist. Am Nordrand kommt zusätzlich zur No-Slip-Randbedingung die konstante Randgeschwindigkeit zum Einsatz. Die Umsetzung in ExaSlang sieht dann folgendermaßen aus:

```
loop over Pdf_S only ghost [0, 1] on boundary{
        Pdf_SW = Pdf_NE@[-1,-1]-(6.0 * latticeWeight_NE * (1.0) * topVel)
        Pdf_S  = Pdf_N @[0,-1]-(6.0 * latticeWeight_N * (0.0) * topVel)
        Pdf_SE = Pdf_NW@[1,-1]-(6.0 * latticeWeight_NW * (-1.0) * topVel)
}
```

Programmausdruck 4.8: Randbehandlung am Nordrand im Testfall Lid-Driven-Cavity

$\vec{u}_w$ steht für die konstante Randgeschwindigkeit des Fluids am Nordrand und wird im Programmausdruck 4.8 durch topVel umgesetzt. Die Randgeschwindigkeit des Fluids am Nordrand beim Lid-Driven-Cavity Testfall beträgt in dieser Arbeit 0.1. Da wir nur eine Geschwindigkeit in x-Richtung besitzen, wurde im Programmausdruck 4.8 anstatt des vollständigen Richtungsvektors $\vec{c}_q$ nur der x-Wert explizit angegeben.

4.2.2 Strömungsschritt

Im Strömungsschritt muss der Transport der PDFs von einer Zelle in die angrenzenden Nachbarzellen durchgeführt werden. Der Strömungsschritt unterscheidet sich grundsätzlich weder in ExaSlang noch in waLBerla für die beiden unterschiedlichen Testfälle des Poiseuille Channels und der Lid-Driven-Cavity.

Bei der genauen Implementierung des Strömungsschritts bieten sich hier zwei verschiedene Varianten an. Die eine Möglichkeit ist, jede Zelle zu durchlaufen und den Inhalt der aktuellen Zelle in die Nachbarzellen an den entsprechenden Stellen zu schreiben. Diese Möglichkeit den Strömungsschritt umzusetzen kann man in Abb. 2.2 erkennen. Die andere Möglichkeit ist es alle Zellen zu durchlaufen und dann den Inhalt aus den anderen Zellen auszulesen und in die aktuelle Zelle zu schreiben. Bei dieser Variante kann man den Kollisionsschritt und den Strömungsschritt bei der Implementierung in einem Zug abarbeiten und den Algorithmus dadurch hinsichtlich der Performance verbessern. Deshalb wurde diese zweite Variante den Strömungsschritt zu implementieren in dieser Arbeit verwendet und wird in ExaSlang und in waLBerla verwendet. In ExaSlang sieht der Strömungsschritt für die Nordwestliche PDF dann so aus:

```
Function StreamStep@finest ( ) {
        loop over Pdf_C{
                Pdf_NW[next] = Pdf_NW@[ 1,-1]
        }
}
```

Programmausdruck 4.9: Strömungsschritt

In diesem Programmausdruck erkennt man, wie in der Schleife das ganze Gitter durchlaufen wird und aus der Nachbarzelle die Nordwest PDF ausgelesen und der Nordwest PDF der aktuellen Zelle zugewiesen wird. Da der Wert der aktuellen PDF noch gebraucht wird, wird der Wert mit Hilfe des **next** Operators in die zweite angelegte Feldstruktur geschrieben. Um den Wert aus z.B. der südöstlichen Nachbarzelle auslesen zu können, kann man mit Hilfe von @[,] auf eine bestimmte Stelle des Gitters ausgehend von der aktuellen Zelle zugreifen. Die erste Zahl in den eckigen Klammern steht hierbei für die x-Koordinate und die zweite Zahl für die y-Koordinate. So bedeutet [1,-1], dass man auf die südöstliche Nachbarzelle ausgehend von der aktuellen Zelle zugreift. Da unsere Schleife nur über unser inneres Feld und nicht über das Ghostlayer läuft, wird gewährleistet, dass man nicht auf eine Nachbarzelle zugreifen kann, welche überhaupt nicht existiert.

4.2.3 Kollisionsschritt

Im Kollisionsschritt unterscheiden sich die beiden Testfälle von einander. In einer Schleife wird zuerst das ganze Feld durchlaufen und für jede Zelle, die Dichte und die Geschwindigkeit des Fluids berechnet. Die Berechnung der Dichte, in ExaSlang als den bezeichnet, ist die Aufsummierung der 9 PDFs einer Zelle und wurde in Gl 2.3 bereits definiert. Die Gleichung lautet in ExaSlang dann:

```
den = Pdf_NW + Pdf_N + Pdf_NE + Pdf_W + Pdf_C + Pdf_E
    + Pdf_SW + Pdf_S + Pdf_SE
```

Programmausdruck 4.10: Dichte

Die Geschwindigkeit des Fluids einer Zelle, in ExaSlang als vel bezeichnet, erhält man durch die Berechnung der Gl. 2.5 in jedem Zeitschritt. In ExaSlang wird diese Gleichung wie folgt umgesetzt:

```
vel = ( Pdf_NW * latticeVel_NW + Pdf_N * latticeVel_N + Pdf_NE
    * latticeVel_NE + Pdf_W * latticeVel_W + Pdf_C
    * latticeVel_C + Pdf_E * latticeVel_E + Pdf_SW
    * latticeVel_SW + Pdf_S * latticeVel_S + Pdf_SE
    * latticeVel_SE ) / den
```

Programmausdruck 4.11: Geschwindigkeit

Als nächstes muss die Equilibriumsfunktion aus der Gl. 2.10 berechnet werden. Diese wird für jedes der 9 Felder nacheinander berechnet. So wird die Berechnung der Equilibriumsfunktion und die Speicherung der berechneten Werte in einer temporären Variable, im Folgenden in ExaSlang feq genannt, der Nordwest PDF folgendermaßen umgesetzt:

```
feq = latticeWeight_NW * den * (1.0 + 3.0 * dot(vel,latticeVel_NW)
    + 4.5 * ((dot(vel,latticeVel_NW))**2) - 1.5 * dot(u,u))
```

Programmausdruck 4.12: Equilibriumsfunktion

Mit der bereits in ExaSlang gegebenen Funktion dot(,) wird hier das Skalarprodukt von zwei Vektoren bestimmt. Nach der Berechnung der Equilibriumsfunktion für die Nordwest PDF, werden dann die Equilibriumsfunktionen für die restlichen PDFs in ExaSlang aufgestellt und berechnet.

Kollisionsschritt im Lid-Driven-Cavity Testfall

Der eigentliche Kollisionsschritt und somit die Aktualisierung der Werte wird in ExaSlang für den Lid-Driven-Cavity Testfall im konkreten Beispiel der Nordwest PDF wie folgt umgesetzt:

```
Pdf_NW = Pdf_NW - tau * ( Pdf_NW - feq )
```

Programmausdruck 4.13: Kollisionsschritt im Lid-Driven-Cavity Testfall

Hier wird der Kollisionsschritt aus der Gleichung 2.7 umgesetzt. Der neue Wert für die Nordwest PDF ergibt sich durch die Addition des Kollisionsterm aus der Gl. 2.8 mit dem dem alten Wert für die Nordwest PDF.

Kollisionsschritt im Poiseuille Channel Testfall

Für den Poiseuille Channel Testfall stellt sich die Nordwest PDF einer Zelle folgendermaßen in ExaSlang dar:

```
Pdf_NW = Pdf_NW - tau * ( Pdf_NW - feq ) + 3.0 * latticeWeight_NW
       * den * dot ( latticeVel_NW, force)
```

Programmausdruck 4.14: Kollisionsschritt im Poieusille Channel Testfall

Im Fall des Poieusille Channels wird hier der Kollisionsschritt aus dem Lid-Driven-Cavity Testfall im Programmausdruck 4.13 übernommen und zusätzlich die exterene Krafteinwirkung in x-Richtung hinzu addiert, was der Umsetzung der Gl 2.14 in ExaSlang entspricht.

Der Kraftvektor $\vec{a}$ wird in ExaSlang hier mit force bezeichnet.

4.2.4 Zeitschritt

Die einzelnen Teilschritte der Lattice-Boltzmann-Methode wurden nun bereits implementiert. Um eine Simulation des Fluids über einen bestimmten Zeitraum durchführen zu können, muss die Lattice-Boltzmann-Methode wiederholt werden. Hierfür benötigen wir nun noch eine Zeitschrittfunktion, welche alle Teilschritte der Lattice-Boltzmann-Methode wiederholt ausführt. Die Zeitschrittfunktion beinhaltet unter anderem folgende wichtige Funktionen:

```
TreatBoundaries ( )
StreamStep ( )
advance Pdf_C
advance Pdf_N
advance Pdf_E
advance Pdf_S
advance Pdf_W
advance Pdf_NE
advance Pdf_SE
advance Pdf_SW
advance Pdf_NW
CollideStep ( )
```

Programmausdruck 4.15: Zeitschritt

In jedem Zeitschritt wird die Randbehandlung mit der Funktion TreatBoundaries () und der Strömungsschritt mit der Funktion StreamStep () ausgeführt. Da wir im Strömungsschritt unsere aktuellen Werte in die zweiten Feldstrukturen der PDFs geschrieben haben, müssen wir nun die zweiten Feldstrukturen der PDFs mit **advance** mit den alten Feldstrukturen der PDFs austauschen. Dies erfolgt bei allen 9 PDFs. Nachdem nun die Randbehandlung und der Strömungsschritt durchgeführt wurden, kann der eigentliche Kollisionsschritt mit CollideStep () durchgeführt werden. Dieser Zeitschritt und somit die Lattice-Boltzmann-Methode muss dann wiederholt werden um die Simulation eines Fluids über einen gewissen Zeitraum darzustellen. In ExaSlang wird wie in anderen Programmiersprachen wie C++ üblich, ebenfalls eine Art main-Funktion benötigt. Diese ruft alle Funktionen auf, wie z.B. Initialisierungsfunktionen, die Zeitschrittfunktion und Ausgabefunktionen, welche im Programm ausgeführt werden sollen. Da die Funktionsweise zu anderen Programmiersprachen hier sehr ähnlich ist, wird auf die main-Funktion in ExaSlang und die genaue Umsetzung der Ausgabefunktionen in ExaSlang nicht weiter eingegangen.

5 High Performance Computing Framework

Um Problemstellungen mit hohen Rechneraufwand zu lösen wird meist ein großer Rechnerverbund benötigt. Neben der Hardware des Rechnerverbunds, welche die erhöhte Rechnerkapazität bietet, um den Rechenaufwand gerecht zu werden, wird häufig ein HPC Framework implementiert, das die Rahmenstruktur für eine gewisse Problemstellung bilden soll. Dieses HPC Framework gilt nicht nur als Schnittstelle zwischen der Hardware und der Software, sondern häufig auch eine Schnittstelle für den Anwender, welcher die genauen Kenntnisse über die Hardware nicht besitzt.

5.1 waLBerla

In dieser Arbeit wurde das HPC Framework waLBerla verwendet. WaLBerla bietet nicht nur eine Vielzahl an Funktionalitäten zur Simulation der Lattice-Boltzmann-Methode, sondern auch verschiedene Implementierungen der einzelnen Funktionen. Die Templatestruktur, welche waLBerla verwendet, ermöglicht es verschiedene Kollisionsmodelle, Randbehandlungen und Testfälle in waLBerla zu integrieren. Um eine genauere Übersicht über die Funktionalitäten, welche waLBerla bereitstellt zu erhalten, kann die Dokumentation von waLBerla [8] eingesehen werden.

5.2 Integration in waLBerla

In waLBerla gibt es bereits alle Datenstrukturen, die wir benötigen, um unsere Testfälle für die Lattice-Boltzmann-Methode zu implementieren. Für diese Datenstrukturen existieren verschiedene Templates. Durch die Verwendung der Templates bietet waLBerla eine größere Auswahl, um z.B. verschiedene Datenstrukturen, Modelle oder andere Testfälle leichter zu simulieren. Zuerst müssen deshalb in waLBerla die richtigen Templates ausgewählt werden, bevor die Felder und Funktionen konkret anlegen werden können. Als erstes wird das Modell der Lattice-Boltzmann-Methode gewählt:

```
typedef lbm::D2Q9< lbm::collision_model::SRT, false,
        lbm::force_model::GuoConstant > LatticeModel_T;
typedef LatticeModel_T::Stencil Stencil_T;
typedef LatticeModel_T::CommunicationStencil CommunicationStencil_T;
```

Programmausdruck 5.1: Modell der Lattice-Boltzmann-Methode

In waLBerla benötigt das Layout die Information, welches Gittermodell verwendet werden soll, um den Stencil festzulegen. Als Gittermodell wurde hier das $d2q9$-Modell gewählt. Die zweite wichtige Information die benötigt wird, ist um welches Kollisionsmodell es sich handelt. In unserem Fall handelt es sich um das bereits bekannte SRT-Modell mit nur einer Relaxationszeit. Nach dem Anlegen des Layouts sowie des benötigten Stencils und einem Stencil für die Kommunikation der verschiedenen Prozesse, muss man sich für die Feldstrukturen, welche man erstellen will, entscheiden.

```
typedef lbm::PdfField< LatticeModel_T >  PdfField_T;
typedef walberla::uint8_t     flag_t;
typedef FlagField< flag_t >  FlagField_T;
```

Programmausdruck 5.2: Felder

Hier wird ein Template PdfField_T für das 50x50 große Gitter gewählt. Später benötigt waLBerla jedoch nicht nur unser Gitter, sondern auch ein Feld FlagField, welches zur Markierung der Ränder oder andere Objekte dient. Deshalb wird im Programmausdruck 5.2 das notwendige Markierungsfeld FlagField_T definiert, um es in die Blockstruktur von waLBerla integrieren zu können.

Nachdem nun das Layout, der Stencil und die benötigten Felder bestimmt worden sind, kann man die notwendigen Parameter aus der Konfigurationsdatei einlesen und diese dann für einen expliziten Testfall auch konkret anlegen. So wird z.B. τ mit

```
auto parameters = config->getOneBlock("Parameters");
const real_t tau = parameters.getParameter< real_t >
        ("tau",real_c(1.8));
```

Programmausdruck 5.3: Parameter auslesen

eingelesen. Hier kann man erkennen, wie zuerst die Konfigurationsdatei nach einem Block mit der Beschriftung Parameters durchsucht wird. Anschließend wird im Block Parameters nach τ gesucht und dieses als konstante reelle Variable angelegt. Falls im Block Parameters in der Konfigurationsdatei kein τ zu finden ist, wird dieses mit dem Defaultwert 1.8 trotzdem erstellt. Der nächste Ausschnitt zeigt den Block Parameters in der Konfigurationsdatei, welcher für die Simulationen in dieser Arbeit verwendet worden ist:

```
"Parameters": {
    "timesteps": 100000,
    "useGui": 0,
    "tau": 1.8,
    "remainingTimeLoggerFrequency": 10,
},
```

Programmausdruck 5.4: Block Parameters in der Konfigurationsdatei

Wie im Programmausdruck 5.4 erkennbar ist, wird hier nicht nur ein Wert für τ mit angegeben, sondern auch die Anzahl der Zeitschritte mit 100000. Die restlichen benötigten Parameter werden auf die selbe Weise ausgelesen und zugewiesen.
Als nächsten Schritt erfolgt dann das Anlegen der bereits definierten Felder mit Hilfe der eingelesen Parameter.

```
LatticeModel_T latModel =
  LatticeModel_T(lbm::collision_model::SRT(tau),
  lbm::force_model::GuoConstant(force));

BlockDataID pdfFieldId = lbm::addPdfFieldToStorage
  (blocks, "pdf field", latModel, initialVelocity, real_t(1));

BlockDataID flagFieldId = field::addFlagFieldToStorage
  < FlagField_T >(blocks,"flag field");
```

Programmausdruck 5.5: Felddefinitionen

5.2.1 Randbehandlung

Die Randbehandlung in waLBerla wird ebenfalls durch Templates umgesetzt. Die zwei verschiedenen Randbedingungen der No-Slip-Randbedingung und der periodischen Randbedingung sind ebenfalls bereits in waLBerla integriert und müssen nur in das Template eingefügt werden. Das Template benötigt zur Randbehandlung das Gitter und das Flagfield, welches zur Kennzeichnung der Zellen benötigt wird. Welche Zelle als Rand deklariert werden soll und welche Randbedingung in dieser Zelle gelten soll, wird in diesem Flagfield implementiert. Da die benötigten Randbedingungen bereits in waLBerla implementiert sind, reicht es in der Konfigurationsdatei in Form des Blocks Boundaries die Randbedingungen zu deklarieren und diese dann der Funktion, welche die Randbedingungen umsetzt als Parameter zu übergeben. Wie die genaue Randbehandlung in waLBerla umgesetzt worden ist, kann im Ausdruck 5.6 nochmals genauer betrachtet werden.

```
const FlagUID fluidFlagUID( "Fluid" );
auto bCon=config->getOneBlock("Boundaries");
typedef lbm::DefaultBoundaryHandlingFactory
  <LatticeModel_T,FlagField_T>BHFactory;

BlockDataID bHId=BHFactory::addBoundaryHandlingToStorage
  (blocks,"boundary handling",flagFieldId,pdfFieldId,fluidFlagUID,
  boundariesConfig.getParameter<Vector3<real_t>>
  ("velocity0",Vector3<real_t>()),
  boundariesConfig.getParameter<Vector3<real_t>>
  ("velocity1",Vector3<real_t>()),
  boundariesConfig.getParameter< real_t >("pressure0",real_c(1.0)),
  boundariesConfig.getParameter< real_t >("pressure1",real_c(1.0)));

geometry::initBoundaryHandling<BHFactory::BoundaryHandling>
  (*blocks,bHId,boundariesConfig);

geometry::setNonBoundaryCellsToDomain
  <BHFactory::BoundaryHandling>(*blocks,bHId);
```

Programmausdruck 5.6: Randbehandlung

5.2.2 Zeitschritt

Da der Kollisionsschritt und der Strömungsschritt in waLBerla durch weitere Templates bereits implementiert worden ist, folgt die Integration dieser beiden Schritte in den Zeitschritt. WaLBerla bietet hierfür eine Zeitschrittfunktion, welcher Funktionen hinzugefügt werden können, die dann in jedem Zeitschritt ausgeführt werden. Um den Kollisionsschritt und den Strömungsschritt auszuführen, muss man dann lediglich die entsprechenden Funktionen, die den Kollisionsschritt und den Strömungsschritt durchführen, mit den richtigen Parametern und unserer angelegten Datenstruktur, in der Zeitschrittfunktion aufrufen. Zuerst wird eine Zeitschrittfunktion angelegt und die Kommunikation der unterschiedlichen Blöcke auf den verschiedenen Prozessen gewährleistet:

```
auto timeloop = make_shared<SweepTimeloop>(blocks->getBlockStorage(),
  timesteps);

blockforest::communication::UniformBufferedScheme
  < CommunicationStencil_T >communication(blocks);

communication.addPackInfo(make_shared< lbm::PdfFieldPackInfo
  < LatticeModel_T >>(pdfFieldId));
```

Programmausdruck 5.7: Kommunikation- und Zeitschritterstellung

Als nächstes wird der Zeitschrittfunktion die Informationen zur Kommunikation der Blöcke, zur Randbehandlung und anschließend der Kollisionsschritt sowie der Strömungsschritt hinzugefügt. Kollisionsschritt und Strömungsschritt werden in waLBerla allerdings in einem Schritt durchgeführt. In waLBerla sieht das wie folgt aus:

```
timeloop->add() << BeforeFunction(communication,"communication")
  << Sweep(BHFactory::BoundaryHandling::getBlockSweep(boundaryHandlingId),
  "boundary handling");

timeloop->add() << Sweep(makeSharedSweep(lbm::makeCellwiseSweep
  < LatticeModel_T, FlagField_T >(pdfFieldId, flagFieldId, fluidFlagUID)),
  "LB stream & collide");
```

Programmausdruck 5.8: Hinzufügen aller Informationen zum Zeitschritt

6 Auswertung

Nachdem nun beide Methodiken zur Implementierung der Lattice-Boltzmann-Methode ausführlich erläutert wurden, kann man die Ergebnisse miteinander vergleichen. Der Fokus bei der Auswertung liegt hierbei nicht allein auf dem Performance Vergleich zwischen beiden Herangehensweisen, sondern auf den Faktoren wie der Benutzerfreundlichkeit, Komplexität, Korrektheit und Flexibiliät. Hierbei ist es wichtig den Entstehungsprozess bei der Implementierung der Lattice-Boltzmann-Methode ebenfalls genauer zu betrachten.
Nach der Erstellung der beiden Methodiken zur Implementierung der Lattice-Boltzmann-Methode können nun die Ergebnisse miteinander verglichen werden. Beim Lid-Driven-Cavity Testfall ist das Ziel, die Zirkulierung des Fluids zu erhalten und die verschiedenen Wirbel, welche sich bei genügend Zeitschritten bilden zu erkennen. Beim Poieusille Channel sollten sich nach genügend Zeitschritten der Kanal einpendeln und das Fluid von Westen nach Osten strömen. Hierbei sollte die Geschwindigkeit in der Mitte am höchsten und an den Rändern am geringsten sein. Da beide Testfälle bekannt sind und dieses Verhalten des Fluids auftreten soll, kann man bereits durch eine erste optische Überprüfung das Verhalten der beiden Methodiken betrachten.

Validierung im Poiseuille Channel Testfall

Die folgenden Abbildungen zeigen für beide Methodiken die Geschwindigkeiten in x-Richtung im Testfall des Poiseuille Channels nach 100000 Zeitschritten und mit dem Relaxations-Parameter $\tau = 1.8$:

Abbildung 6.1: Poiseuille Channel in ExaSlang

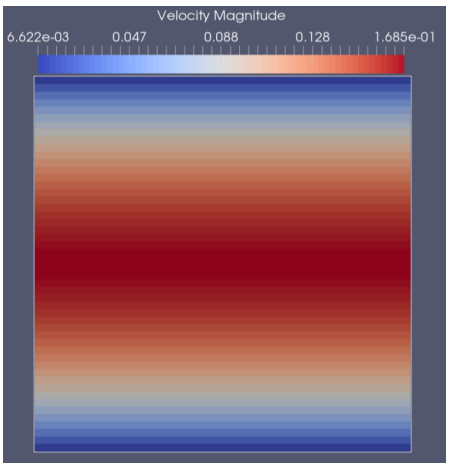

Abbildung 6.2: Poiseuille Channel in waLBerla

Zwar sind die Ergebnisbilder aus den Abb. 6.1 und Abb. 6.2 bei der Implementierung und der Validierung hilfreich und können bei der Fehlersuche bereits ein erstes Indiz sein, allerdings reichen diese zur Überprüfung der numerischen Äquivalenz nicht aus. Um die beiden Methodiken aussagekräftig zu vergleichen, wird bei beiden Methodiken im selben Zeitschritt ein Längsschnitt, parallel zur y-Achse, in der Mitte der Domäne durchgezogen. Anschließend können die Geschwindigkeiten in x-Richtung, die man durch den Längsschnitt erhält, wie folgt miteinander verglichen werden:

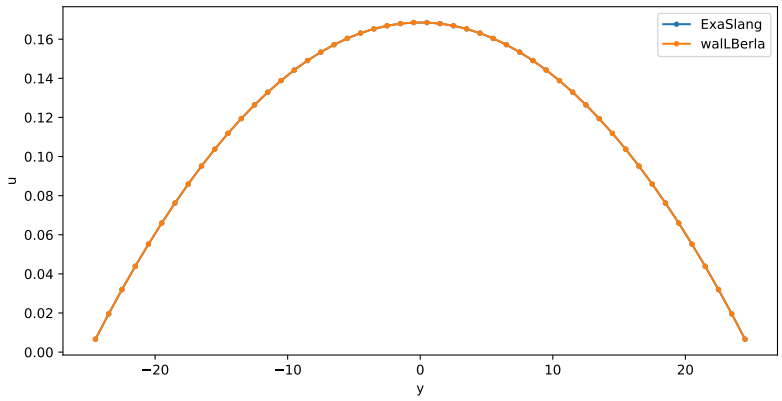

Abbildung 6.3: Geschwindigkeitsprofil im Fall des Poiseuille Channels

Das Geschwindigkeitsprofil im Fall des Poiseuille Channels zeigt die Geschwindigkeit in x-Richtung beider Methodiken. Hier steht u für die Geschwindigkeit und y für die y-Koordinate. Das Geschwindigkeitsprofil weist ein symmetrisches Geschwindigkeitsprofil und stellt eine Parabel dar. In der Mitte des Kanals ist die Geschwindigkeit am größten und an den Rändern am kleinsten. Um den genauen Unterschied beider Methodiken zu erkennen, wird in der nächsten Abbildung der relative Fehler zwischen den beiden Methodiken aufgezeigt:

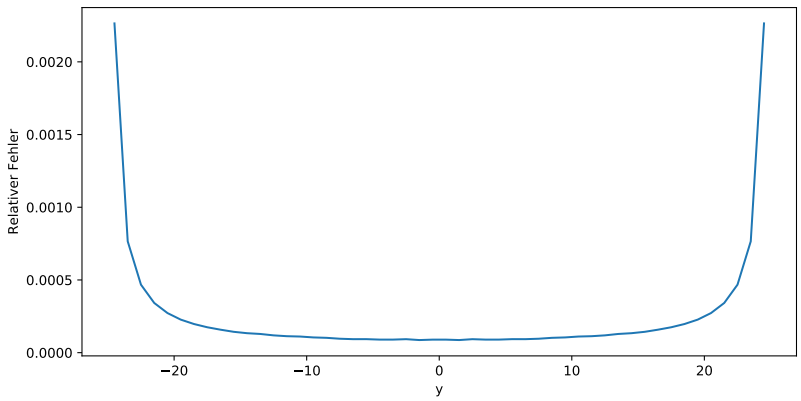

Abbildung 6.4: Relativer Fehler im Poiseuille Channels Testfall

Da waLBerla bereits mehrfach getestet und überprüft wurde, kann man aus der Abb. 6.2 schließen, dass die Diskrepanz beider Methodiken durch einen Fehler bei der Umsetzung der Lattice-Boltzmann-Methode in ExaSlang hervorgerufen wird. Durch den Verlauf des relativen Fehlers kann man erkennen, dass der Fehler an den Rändern größer wird. Dies lässt darauf schließen, dass bei der Implementierung in ExaSlang noch ein Fehler in der Randbehandlung existieren muss.

Validierung im Lid-Driven-Cavity Testfall

Für den Lid-Driven-Cavity Testfall ergeben die beiden Methodiken nach 100000 Zeitschritten und dem Relaxations-Parameter $\tau = 1.8$ folgende Abbildungen:

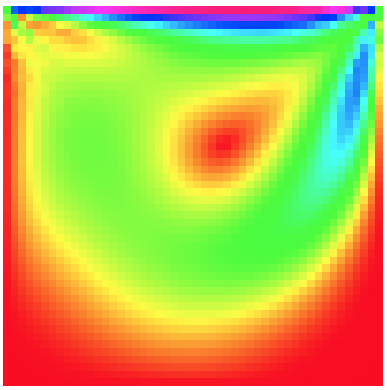

Abbildung 6.5: Lid-Driven-Cavity in ExaSlang

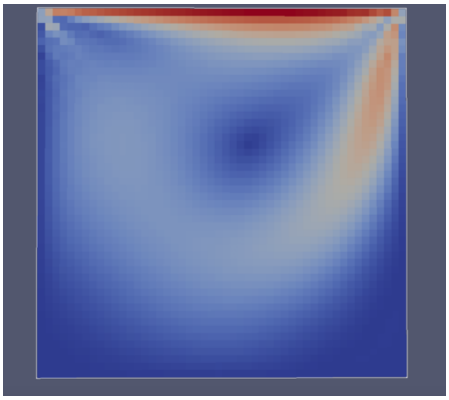

Abbildung 6.6: Lid-Driven-Cavity in waLBerla

Auch im Lid-Driven-Cavity Testfall wird mit Hilfe eines Längsschnitts durch die Mitte der Domäne das Geschwindigkeitsprofil und der relative Fehler dargestellt:

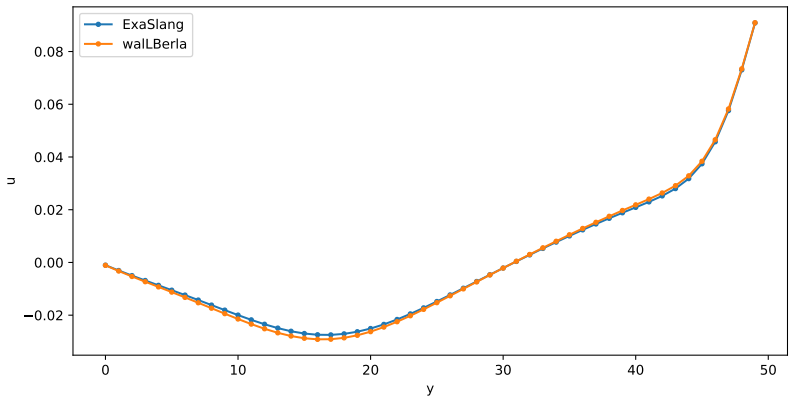

Abbildung 6.7: Geschwindigkeitsprofil im Fall des Lid-Driven-Cavity

In der Abb. 6.7 ist bereits zu erkennen, dass auch im Lid-Driven-Cavity Testfall keine numerische Äquivalenz vorliegt. Der relative Fehler in diesem Fall sieht wie folgt aus:

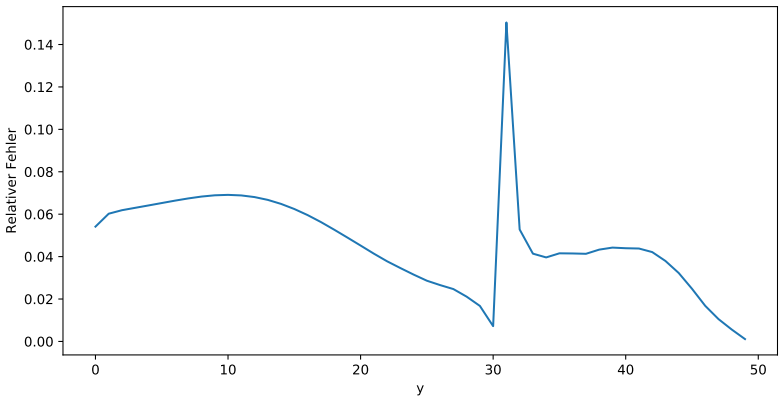

Abbildung 6.8: Relativer Fehler im Lid-Driven-Cavity Testfall

Der relative Fehler ist auch hier nicht konstant und zeigt auch für den Lid-Driven-Cavity Testfall eine Diskrepanz.
Da in beiden Testfällen die numerischen Werte für die beiden Methodiken nicht exakt gleich sind, sollte die Fehlersuche und anschließende Verbesserung des Fehlers der nächste Schritt sein. Diese Korrektur ist allerdings nicht mehr Bestandteil dieser Arbeit und muss in einem zukünftigen Projekt realisiert werden. Trotz der Diskrepanz bei der Validierung der numerischen Äquivalenz, ist es jedoch möglich Aussagen über die Flexibilät, Komplexität und die Performance beider Methodiken zu treffen.

6.1 Flexibilität

Beim Vergleich der Flexibilität beider Verfahren ergeben sich große Unterschiede und es ist hier notwendig zuerst die Flexibilität für jede Methodik einzeln zu analysieren. Bevor wir konkret die gewählte DSL ExaSlang und das HPC Framework waLBerla betrachten, werfen wir einen Blick auf die Code Generierung und das HPC Framework im Allgemeinen. Als erstes untersuchen wir die Flexibilität bei der automatischen Code Generierung.
Bei der automatischen Code Generierung ist es notwendig eine DSL zu verwenden, um gewisse Problemstellungen in einer bestimmten Domäne und unter vorher festgelegten Spezifikationen zu lösen. Durch die Einschränkung einer DLS auf eine bestimmte Domäne, bietet diese nur wenig Möglichkeiten eine Problemstellung außerhalb dieser Domäne zu behandeln. Die geringe Auswahl an verfügbaren DSLs schränkt die Möglichkeit eine DSL zur Code Generierung zu verwenden zusätzlich ein.
Die in dieser Arbeit verwendete DSL ExaSlang ist zur Lösung von numerischen Problemen mittels Mehrfachgittern konzipiert worden. ExaSlang ist somit nicht direkt zur Lösung der Lattice-Boltzmann-Methode, welche nur auf einem Gitter arbeitet, ausgelegt. Der Gebrauch von ExaSlang, trotz der womöglich erschwerten Bedingungen, zeigt die Vor- und Nachteile beim Einsatz von ExaSLang zur Implementierung der Lattice-Boltzmann-Methode. Das Ziel bei der Implementierung der Lattice-Boltzmann-Methode in ExaSlang sollte es somit auch sein, mögliche fehlende Funktionen, Datenstrukturen oder andere wichtige Funktionalitäten zu erkennen und eine mögliche Alternativlösung der Lattice-Boltzmann-Methode in ExaSlang zu finden. Bei der Implementierung der Lattice-Boltzmann-Methode stellte sich heraus, dass

ExaSlang bereits alle notwendigen Funktionen zur Erstellung der Lattice-Boltzmann-Methode in ExaSlang besitzt. Da die Lattice-Boltzmann-Methode auf einem Einfachgitter agiert und auch ein Einfachgitter Bestandteil eines Mehrfachgitters ist, sind notwendige Funktionalitäten wie Feldstrukturen, eine Gitterstruktur, Schleifen und If-Anweisungen bereits in ExaSlang integriert. Auch das Integrieren der Randbehandlung stellt durch bereits einige vordefinierte Randbehandlungen und durch eine Ghostlayerstruktur kein Hindernis dar. Soll das Modell von einem $d2q9$-Modell zu einem $d3q19$-Modell erweitert werden, kann es in ExaSlang etwas unübersichtlicher werden. Durch die Hardcodierung der einzelnen PDFs werden bei einem dreidimensionalen Modell 19 Felder für die PDFs benötigt. Da in dem Kollisionsschritt für jede PDF eine angepasste Equillibriumsfunktion und die Aktualisierung der neuen Werte einzeln durchgeführt und nicht in einer Schleife abgewickelt wird, erhöht sich nicht nur die Unübersichtlichkeit, sondern die Anfälligkeit für Flüchtigkeitsfehlern bei der Erstellung der vielen Gleichungen. Doch auch die Erweiterung der Dimensionen stellt trotz des deutlich erhöhten Codes in ExaSlang keine großen Probleme dar. Auch der Austausch des Kollisionsmodells von einem SRT-Modell in ein Multi-Relaxation Time (MRT) Modell ist in ExaSlang möglich. Die bisher benützten Feldstrukturen und Funktionen müssen in ExaSlang allerdings neu programmiert werden. Die Umstellung des Kollisionsmodells ist somit grundsätzlich möglich, allerdings ist dies recht umständlich und erfordert unter Umständen eine komplett neue Implementierung der Lattice-Boltzmann-Methode in ExaSlang mit dem neuen Kollisionsmodell. Möchte man die Strömung des Fluids mit einem Algorithmus, welcher nicht auf einer ähnlichen Gitterstruktur wie die Lattice-Boltzmann-Methode basiert, simulieren, dann kann es er Fall sein, dass ExaSlang nicht die nötige Funktionalität bietet und auch die Erweiterung der DSL zur Bewältigung der Problemstellung nicht möglich ist.
Abschließend kann man sagen, dass bei der Code Generierung die Flexibilität nur bedingt vorhanden ist. Um eine Problemstellung zu lösen benötigt man eine DSL, welche für die zu lösende Problemstellung oder eine sehr ähnliche Problemstellung innerhalb der selben Domäne entwickelt wurde. Innerhalb dieser Domäne sind Erweiterungen und Veränderungen der Problemstellung umsetzbar, doch wenn die Problemstellung zu stark von der aktuellen Domäne abweicht, wird eine andere DSL benötigt, welche meist erst für die neue Problemstellung entwickelt werden muss.

Ein HPC Framework bietet eine Rahmenstruktur um gewisse Problemstellungen auf einem Cluster zu berechnen. Das HPC Framework wird hierbei stark an die Hardware angepasst und die einzelnen Funktionalitäten so weit wie möglich parallelisiert und optimiert um die Performance zu verbessern. Die einzelnen Funktionen die ein HPC Framework bieten kann, sind hier wiederum abhängig von der Problemstellung für welche das HPC Framework entwickelt worden ist. Das HPC Framework bietet somit auch nur für eine gewisse Domäne Funktionalitäten.
Das in dieser Arbeit verwendete HPC Framework waLBerla bietet eine Rahmenstruktur zur Simulation von Fluiden mit Hilfe der Lattice-Boltzmann-Methode. Die Umsetzung von zweidimensionalen oder dreidimensionalen Modellen kann hier mittels einer Parameterübergabe in der Konfigurationsdatei umgesetzt werden. WaLBerla beinhaltet auch alle notwendigen Funktionen um die Randbehandlung der verschiedenen Testfälle umzusetzen. Durch die Umsetzung der meisten Funktionen in waLBerla in Form von Templates bietet waLBerla eine größere Vielfalt an möglichen Parametern und Funktionalitäten die umgesetzt werden können, wie ExaSlang bisher bereitstellt. Da waLBerla verschiedene Funktionen wie Blockstruk-

turen zur Parallelisierung, Speicherung der Felder jeglicher Art durch Templates und der Funktionsstruktur, welche den Einbau eigener Funktionen unterstützt, bietet, besitzt waLBerla eine größere Flexibilität. Die Integrierung eines anderen Algorithmus wie der Lattice-Boltzmann-Methode in waLBerla gestaltet sich auch in waLBerla umständlich und ist nur bedingt möglich. Um einen anderen Algorithmus in waLBerla umzusetzen benötigt man nicht nur fundierte Fachkenntnisse zum Algorithmus, sondern auch das Wissen über die Struktur und Funktionalitäten von waLBerla.
Im direkten Vergleich bietet waLBerla die höhere Flexibilität zur Erweiterung der Lattice-Boltzmann-Methode und zur Simulation von Fluiden mit Hilfe der Lattice-Boltzmann-Methode. Die größere Funktionalität und die Templatestruktur die waLBerla bereitstellt, ermöglicht eine höhere Flexibilität im Vergleich zu ExaSlang. Bei der Umsetzung eines anderen Algorithmus als der Lattice-Boltzmann-Methode zur Simulation von Fluiden kann es bei ExaSlang und bei waLBerla zu Problemen und Einschränkungen kommen.

6.2 Komplexität

Um die Komplexität bei der Code Generierung zu analysieren muss die Komplexität der DSL und die Komplexität des generierten Codes beachtet werden. Die verwendete DSL bei der Code Generierung bietet meist nur Funktionalitäten, welche benötigt werden um die Problemstellung zu bewältigen. Andere und unnötige Funktionen werden in der DSL bereits im vornherein nicht berücksichtigt und umgesetzt. Die Einschränkung auf eine gewisse Domäne und die vielen Spezifikationen definieren das Verhalten, welches die DSL aufweisen soll und schränken diese stark ein. Andere Problemstellungen mit einer DSL zu lösen, ist dann meist wegen fehlender Funktionalitäten vorerst nicht möglich. Durch diese Einschränkung auf die notwendigen Funktionalitäten bleiben viele DSLs, wie auch ExaSlang, recht übersichtlich. Das Programmieren in ExaSlangs stellt somit von der Komplexität her in ExaSlang vorerst kein Problem dar. Der generierte C++ Code, der nach dem Kompilieren unseres ExaSlang Codes generiert wird, kann jedoch sehr komplex werden. Da der ExaStencils Kompiler versucht den C++ Code so gut wie möglich zu parallelisieren und zu optimieren, wird die Performance zwar verbessert, doch der generierte Code wird deutlich unübersichtlicher und komplexer.
Die Komplexität eines HPC Frameworks ist meist sehr hoch. Das HPC Framework soll eine Rahmenstruktur bilden, mit der eine Problemstellung auf einem Cluster simuliert werden kann. Da der Anwender das nötige Wissen über die Hardware meist nicht kennt, muss das HPC Framework bereits die meisten Funktionalitäten zur Verfügung stellen. Durch die vielen Funktionalitäten die das HPC Framework bieten soll, um dem Anwender die Benutzung zu erleichtern, steigt die Komplexität bei einem HPC Framework bereits stark an. Die hohe Parallelisierung und die vielen Optimierungen, welche an die Hardware angepasst werden, führen bei einem HPC Framework ebenfalls zu sehr hoher Komplexität. Wenn der Anwender die Funktionsweise hinter den gebotenen Funktionalitäten verstehen will, muss der stark parallelisierte und optimierte Code genauer betrachtet werden. Dieser Code ist wie der generierte Code in ExaSlang meist recht unübersichtlich und komplex. Auch in waLBerla ist die Komplexität sehr hoch. WaLBerla bietet bereits viele Funktionalitäten und Möglichkeiten um unterschiedliche Modelle zu simulieren, indem nur wenige Parameter angepasst werden. Die Templatestruktur, welche hierfür benötigt wird, ist allerdings recht komplex. WaLBerla versucht dem Anwender verschiedene Möglichkeiten zu geben um unterschiedliche Simulationen mit den bereits enthaltenen Funktionen durchzuführen. Das Ergebnis der vielen Funktionalitä-

ten und der ständigen Erweiterung von waLBerla, um noch mehr Simulationen durchführen zu können, führt zu einer hohen Komplexität. WaLBerla ist also nicht nur durch die hohe Auswahl an Funktionalitäten, sondern auch durch den umfangreichen Code hinter diesen Funktionalitäten sehr unübersichtlich und schwer zu verstehen.
Da ExaSlang nicht so viele Funktionalitäten besitzt und sich deutlich eingeschränkter auf der gewählten Domäne zeigt, ist ExaSlang nicht so komplex wie waLBerla. Die vielen Funktionalitäten, der umfangreiche Code zur Parallelisierung und die Templatestruktur, welche den Einbau neuer Funktionen erleichtern soll, bewirken bei waLBerla dagegen zu einer sehr hohen Komplexität.

6.3 Benutzerfreundlichkeit

Um die Benutzerfreundlichkeit beider Methodiken miteinander zu vergleichen, müssen mehrere Faktoren betrachtet werden. Da bei der Code Generierung die DSL das wichtigste ist blicken wir direkt auf die hier verwendete DSL ExaSlang. ExaSlang bietet einige Datenstrukturen, welche die Implementierung der Lattice-Boltzmann-Methode erleichtern können. Die Feldstrukturen und die einfach Erstellung eines Layouts mit den Ghostlayern, sind ein großer Vorteil bei der Implementierung der Lattice-Boltzmann-Methode. Da die Randbehandlung mit Hilfe der Ghostlayer sehr einfach umgesetzt werden kann, kommt es hier wenig zu Indexfehlern. Auch die Kommunikation der Blöcke bei der Parallelisierung gestaltet sich in ExaSlang einfach. Einfache Zuweisungen von Werten, wie es beim Strömungsschritt der Fall ist, lassen sich in ExaSlang sehr einfach und direkt umsetzen und sind dann auch sehr verständlich. Die vielen Funktionalitäten die ExaSlang bereits enthält, erleichtern die Implementierung der Lattice-Boltzmann-Methode im Allgemeinen. Ein Problem in ExaSlang war anfangs jedoch die fehlende Dokumentation. Zwar ist die Dokumentation mittlerweile vorhanden, doch ohne eine Dokumentation am Anfang ist es schwer die Sprache zu verstehen. Wenn man die Funktionalitäten nicht kennt und deren Funktionsweise nicht nachlesen kann, kann man diese unter Umständen auch nicht verwenden. Ein weiteres Problem in ExaSlang ist die Fehleranzeige bei Kompilierfehlern. Die Fehleranzeige in der Konsole beschreibt den Fehler nur recht ungenau und die Stelle im eigentlichen Code, wo der Fehler auftritt wird auch nicht direkt angegeben. Die Fehlersuche gestaltet sich in ExaSlang dadurch recht schwer und ein großes Fachwissen über ExaSlang ist notwendig um einige Fehler deuten und beheben zu können. Bei der Fehlersuche im generierten Code, welche keine Übersetzungsfehler sind, besitzt ExaSlang auch noch einige Probleme. Der generierte Code ist meist recht unübersichtlich und dadurch, dass man ihn nicht selbst programmiert hat, sind die Fehler zunächst schwer zu deuten. Man benötigt auch hier recht viel Zeit um den generierten Code zu verstehen und dann einen möglichen Fehler zu erkennen.
In waLBerla gestaltet sich bei der Betrachtung der Benutzerfreundlichkeit ein anderes Bild. So war in ExaSlang das größte Problem noch die fehlende und nicht so ausgereifte Dokumentation und die Fehlersuche beim Übersetzen des Codes und der größte Vorteil die Einfachheit der DSL ExaSlang und die Programmierung in ExaSlang, nachdem man ein Verständnis über die Sprache erhalten hat. In waLBerla dagegen ist die Fehlersuche sehr angenehm. Da waLBerla auch auf C++ Code basiert, bekommt man eine detaillierte Fehlerangabe, welcher Fehler auftritt und an welcher Stelle. Dadurch können Fehler recht schnell behoben werden. Ein weiterer Vorteil von waLBerla ist auch dessen Interface. So bietet waLBerla neben dem C++ Code auch eine Python Schnittstelle mit der man sehr schnell Testfälle der Lattice-Boltzmann-Methode simulieren und betrachten kann. Der größte Vorteil von waLBerla ist allerdings die ausge-

prägte Dokumentation. Diese enthälte nicht nur Beschreibungen wie die Funktionen definiert sind, sondern auch viele Tutorials, die einem den Umgang mit den wichtigsten Funktionen näher bringen und zeigen wie man seine eigenen Funktionen in waLBerla einbauen kann um so z.B. unterschiedliche Testfälle zu simulieren. Dies ist durch die Templatestruktur möglich, auch wenn man hierfür zuerst die Funktionsweise von waLBerla verstanden haben muss. Das größte Problem in waLBerla ist die Komplexität. Zwar gibt es eine ausgereifte Dokumentation und Tutorials, doch durch die Vielzahl an Funktionen, muss man bei der Implementierung des eigenen Codes oft sehr lange suchen, welche Funktionen es gibt und welche unter Umständen besser geeignet sind. Die vielen Funktionalitäten werden schnell unübersichtlich und Implementierung eines Algorithmus gestaltet sich mehr als eine Suche der notwendigen Funktionen und dem Nachlesen wie diese zu verwenden sind. Durch die Templatestruktur werden viele Funktionen und auch er spätere Code zum Teil sehr unübersichtlich und erschweren das Programmieren zusätzlich.

6.4 Performance

Bei der Performance-Messung beider Methodiken gilt es viele Faktoren zu beachten, um ein aussagekräftiges Ergebnis vorweisen zu können. Da die Optimierung beider Methodiken jedoch nicht das Ziel dieser Arbeit war, wurden hier weder in ExaSlang noch in waLBerla Optimierungen vorgenommen. Die Implementierung der Lattice-Boltzmann-Methode wurde auf einer Workstation am Lehrstuhl für Systemsimulation an der Friedrich-Alexander-Universität durchgeführt. Die Performance-Messungen wurden dementsprechend auf selbiger Workstation durchgeführt. Da auf der Workstation auch noch viele andere Prozesse durchgeführt werden, ist die Performance Messung der beiden Methodiken auch von der verfügbaren Rechenleistung, welche die Workstation im Moment der Messung zur Verfügung stellt, abhängig. Nachdem beide Methodiken nicht parallelisiert getestet wurden, wird die Performance auch nur für einen Kernel getestet. Die Wahl des Kernels auf dem simuliert worden ist, wurde hierbei auch nicht berücksichtigt. Des Weiteren ist die Gittergröße mit 50x50 nur sehr klein gewählt und die Lattice-Boltzmann-Methode benötigt dadurch nur wenig Zeit. Die Erstellung aller notwendigen Datenstrukturen und Funktionen beeinflusst die Gesamtzeit bei der Durchführung des Programms jedoch. Wenn die Berechnungen somit sehr schnell durchgeführt werden und die Erstellung aller notwendigen Datenstrukturen, welche unabhängig von der Größe des Gitters immer relativ gleich lang dauern, in keinem angemessenen Verhältnis stehen, wird die Aussagekraft über die Performance auch hier stark beeinflusst. Für aussagekräftigere Performance-Messungen müsste die Gittergröße dementsprechend größer gewählt werden. Wenn man all diese Faktoren vernachlässigt, ist es für einen trotzdem von Interesse die Laufzeit der Lattice-Boltzmann-Methode beider Methodiken zu testen und diese miteinander zu vergleichen. Beide Methodiken wurden mit den selben Parametern öfter hintereinander ausgeführt. Hierbei wurden dann nur die besten Ergebnisse berücksichtigt. Des Weiteren wird hier nur die Dauer für die Randbehandlung, den Kollisionsschritt und den Strömungsschritt ausgewertet. Ausgabefunktionen, Visualisierungen und andere Funktionen, welche kein Bestandteil der Lattice-Boltzmann-Methode sind werden zeitlich nicht gemessen und beeinflussen das Ergebnis somit nicht.
Der mit ExaSlang generierte Code ergab bei mehrfachem Ausführen nur für die berücksichtigten Schritte der Randbehandlung, des Kollisionsschritts und des Strömungsschitts im Poiseuille Channel Testfall eine Laufzeit von 24,93 Sekunden und im Lid-Driven-Cavity Testfall eine Laufzeit von 23,46 Sekunden.

Die Laufzeit der Lattice-Boltzmann-Methode und somit nur die Laufzeit der 3 Teilschritte der Randbehandlung, des Kollisionsschritt und des Strömungsschritt, ergab bei der Messung in waLBerla für den Poiseuille Channel Testfall eine Laufzeit von 12,93 Sekunden und im Lid-Driven-Cavity Testfall eine Laufzeit von 12,90 Sekunden. Alle Zeiten können in der folgenden Tabelle nochmals genauer betrachten werden:

	Poiseuille Channel	Lid-Driven-Cavity
ExaSlang	24,93 s	23,46 s
waLBerla	12,93 s	12,88 s

Abbildung 6.9: Performance-Messungen

Beim Vergleich der unterschiedlichen Laufzeiten fällt auf das in beiden Methodiken die Laufzeiten für die verschiedenen Testfälle relativ gleich lang sind. Was uns jedoch eigentlich interessiert ist der Vergleich der Laufzeiten des generierten Codes mit Hilfe von ExaSlang und die Laufzeit des Codes, welcher mit Hilfe des HPC Frameworks waLBerla umgesetzt worden ist. Hier ist waLBerla in beiden Fällen um den Faktor 2 schneller. Da jedoch zu viele Faktoren diese Performance-Messungen beeinflussen und diese größtenteils bei den Messungen vernachlässigt wurden, muss dieses Ergebnis unter Vorbehalt betrachtet werden.

7 Zusammenfassung

In dieser Arbeit waren die Hauptziele die zwei verschiedenen Methodiken der Code Generierung und der Verwendung eines HPC Frameworks zur Umsetzung der Lattice-Boltzmann-Methode miteinander zu vergleichen und die jeweiligen Vor- und Nachteile genauer zu betrachten. Um die Lattice-Boltzmann-Methode durchzuführen wurden die beiden Testfälle des Poiseuille Channels und der Lid-Driven-Cavity ausgewählt. Nachdem die Lattice-Boltzmann-Methode, sowie die gewählten Testfälle genauer erläutert worden sind, galt es eben diese durch die zwei verschiedenen Herangehensweisen umzusetzen. Bereits bei der Implementierung des Codes in ExaSlang und der Integration der Testfälle in waLBerla konnte man die entscheidenden Eigenschaften beider Verfahren deutlich erkennen. Die Faktoren, welche bei der Auswertung der beiden Methodiken eine entscheidende Rolle spielen, sind die Validierung beider Verfahren, sowie der Vergleich von Flexibilität, Komplexität und der Performance.
Die erste umgesetzte Methodik ist die Implementierung der Lattice-Boltzmann-Methode mittels automatischer Code Generierung. Hierfür wurde die DSL ExaSlang ausgewählt. Bei der Implementierung in ExaSlang zeigten sich schnell die Stärken und Schwächen bei dieser Herangehensweise. ExaSlang verfügt nur über eine äußerst kurze und lückenhafte Dokumentation. Dies führt zu Problemen bei der Betrachtung der Funktionalitäten, welche ExaSlang zur Verfügung stellt. Ein weiteres Problem von ExaSlang ist die ungenaue Fehlerangabe beim Übersetzen des Codes. Die Fehleranzeigen in ExaSlang sind nicht detailliert genug. So lässt sich der Fehler nur schwer bis zur richtigen Stelle im Code zurückverfolgen und ist für Laien häufig auch nicht deutbar. Ein großer Vorteil dagegen ist die Übersichtlichkeit des ExaSlang Codes. Im Gegensatz zum HPC Framework waLBerla ist der ExaSlang Code sehr übersichtlich, kurz und verständlich. Nachdem Verstehen der Datenstrukturen und Funktionalitäten, welche ExaSlang bietet, ist die Implementierung der Lattice-Boltzmann-Methode in ExaSlang unkompliziert. Die Wahl einer DSL wie ExaSlang zur Implementierung der Lattice-Boltzmann-Methode macht aus Gründen der Performance Sinn. Eine DSL bietet durch ihre vielen Spezifikationen und ihre Auslegung um eine gewisse Problemstellung zu lösen, die Möglichkeit die Performance eines Programmes stark zu verbessern. Mögliche Optimierungen und Parallelisierungen können in ExaSlang durchgeführt werden um die Performance zu steigern. Um die Performance-Verbesserungen allerdings auch aussagekräftig zu beweisen, sind genauere Messungen notwendig, wie sie in dieser Arbeit durchgeführt worden sind.
Die zweite Methodik die verwendet wurde ist die Umsetzung der Lattice-Boltzmann-Methode mit Hilfe des HPC Frameworks waLBerla. Dieses HPC Framework ist bereits darauf ausgelegt die Lattice-Boltzmann-Methode zu simulieren. Der Fokus bei dieser Herangehensweise, war die Integration der ausgewählten Testfälle in waLBerla und das Zusammensetzen aller notwendigen Funktionen um die Lattice-Boltzmann-Methode zu simulieren. Die Schwierigkeit liegt hier darin, die vielen verschiedenen Templatefunktionen, die waLBerla bietet zu verstehen, zusammenzufügen und mit den richtigen Parametern angepasst auszuführen. Die ausführliche Dokumentation von waLBerla ist hierbei jedoch ein großer Vorteil. Das größte Problem, welches waLBerla beinhaltet sind die komplexen Datenstrukturen und der unübersichtliche Code. Die Suche nach möglichen Funktionen, welche man für die Umsetzung benötigt, sind sehr zeitaufwendig und durch die Templatestrukturen auch recht kompliziert. Beim Vergleich der Performance zeigte sich waLBerla als die deutlich bessere Wahl. Das HPC Framework war bereits unoptimiert um den Faktor zwei schneller und bietet auch deutlich mehr Möglichkeiten für Optimierungen.
Bei der abschließenden Validierung beider Methodiken mit Hilfe eines Längsschnitts durch

die Mitte der Domäne, kann man durch Betrachtung des relativen Fehlers für den Poiseuille Channel Testfall in Abb. 6.4 und des relativen Fehlers für den Lid-Driven-Cavity Testfall in Abb. 6.8 erkennen, dass die numerischen Ergebnisse bei der Implementierung in ExaSlang noch einen geringen Fehler an den Rändern ergeben. Diese gilt es in zukünftigen Projekten noch zu beheben, um die Validierung und den endgültigen Beweis zu erbringen, dass die Lattice-Boltzmann-Methode auch korrekt mittels ExaSlang umsetzbar ist.
Abschließend lässt sich jedoch sagen, dass beide Methodiken ihre Vor- und Nachteile besitzen. Die richtige Wahl zwischen den beiden unterschiedlichen Methodiken zur Umsetzung der Lattice-Boltzmann-Methode ist von vielen verschiedenen Faktoren abhängig und kann somit nicht verallgemeinert getroffen werden.

Literaturverzeichnis

[1] Dominik Bartuschat. "Direct Numerical Simulation of Particle-Laden Electrokinetic Flows on High-Performance Computers". Magisterarb. Chair for System Simulation FAU, 2016.

[2] D. d'Humières. "Generalized lattice-boltzmann equations". In: *Rarefied gas dynamics-Theory and simulations* (1992), S. 450–458.

[3] Xiaoyi He und Li-Shi Luo. "Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation". In: *Phys. Rev. E 56, 6811* (1997).

[4] S. Hou et al. "Simulation of Cavity Flow by the Lattice Boltzmann Method". In: *Journal of Computational Physics* Volumne 118, Issue 2 (1995), S. 329–347.

[5] Christian Lengauer et al. *ExaStencils: Advanced Stencil-Code Engineering*. Springer-Verlag. 2014.

[6] Christoph Rettinger. "Fluid flow simulations using the lattice Boltzmann method with multiple relaxation times". Magisterarb. Chair for System Simulation FAU, 2013.

[7] Christian Schmitt et al. "ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers". In: *IEEE Press.* 42-51 (2014).

[8] Chair for System Simulation FAU. *http://www.walberla.net/*.

[9] D. A. Wolf-Gladrow. *Lattice-gas cellular automata and lattice Boltzmann models: an introduction*. Number 1725. Springer, 2000.