Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Lattice-Boltzmann Simulationen auf mehreren GPUs

Benjamin Kurz

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. habil. Miriam Mehl
Betreuer/in:	Dipl.-Inf. Michael Lahnert
Beginn am:	7. November 2017
Beendet am:	7. Mai 2018

Kurzfassung

In dieser Arbeit geht es um die Implementierung der Lattice-Boltzmann Methode auf mehreren Grafikkarten. Die Lattice-Boltzmann Methode ist eine bekannte und beliebte Methode um hydrodynamische Simulationen zu berechnen. Dabei wird der Raum durch ein Gitter diskretisiert und mittels einer Verteilungsdichtefunktion wird mit Populationen, die sich auf den Gittern bewegen, gerechnet. Als Grundlage dient dafür das Software Paket *ESPResSo*, das durch die vorgestellte Implementierung erweitert wird, und die Bibliothek *P4EST*, die für die Erstellung und Verwaltung eines Gitters, das auf Oktalbäumen basiert, genutzt wird. Das durch *P4EST* erstellte Gitter wird zusätzlich in Patches verfeinert, die dann auf den Grafikkarten durch einen CUDA-Code parallel bearbeitet werden. Die vorgestellte Implementierung nutzt das Message Passing Interface und ist dafür ausgerichtet auch auf großen Computern zu laufen und eine gute Skalierung zu erreichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Weiche Materie und komplexe Flüssigkeiten sind gegenwärtig von Interesse für viele wissenschaftliche Arbeiten. Die weiche Materie deckt dabei einen großen Bereich ab. So fallen Flüssigkristalle, wie sie in LCD Bildschirmen Verwendung finden und eine eigene Industrie hervorgebracht haben, unter die weiche Materie. Auch verschiedene Polyelektrolyte, Gele, Elastomere und Tenside gehören dazu. In der Biologie und der Medizin ist die weiche Materie auch vertreten zum Beispiel durch Blut, Zellen oder Protein-Strukturen. Das Verhalten dieser Materie lässt sich nicht einfach beschreiben, da die Interaktionen auf verschiedenen Zeit- und Längenskalen verlaufen. So reichen die Wechselwirkungen von der mikroskopischen Skala, bei der die Materie durch Atome und Moleküle repräsentiert wird und über Kräfte miteinander wechselwirken, bis hin zur makroskopischen Skala, bei der die Materie als Kontinuum wahrgenommen wird. Algorithmen, welche die Phänomene zwischen der mikroskopischen und makroskopischen Skala betrachten, werden als mesoskopische Algorithmen bezeichnet und sind ausschlaggebend für die Eigenschaften der weichen Materie. So kann eine Flüssigkeit ihre Eigenschaft stark verändern durch die Zugabe einer winzigen Menge eines Tensids, wie sie in Wasch- und Spülmitteln vorkommt.

Die Beschreibung all der dafür verantwortlichen Eigenschaften wie die Thermodynamik, elektrokinetische und rheoligische Effekte, so wie Phasenübergänge und Instabilitäten ist auf analytischem Wege nicht zu lösen oder bedient sich vieler Vereinfachungen [Sch08]. Deshalb ist es wichtig, aussagekräftige und zuverlässige Simulationsmethoden zu finden, um die weiche Materie auch außerhalb von Experimenten verstehen zu können. Der Vorteil von Simulationen ist, dass sie ohne äußere Störungen durchführbar sind und beliebig reproduzierbar. Kritisch ist dabei nur die Komplexität der Simulation und ihre Laufzeit. Durch bessere Hardware, die zum Teil auch eine große Parallelität bei der Ausführung von Code erlaubt, und dafür optimierte Algorithmen, wird die Simulation aber immer wichtiger in der Forschung und zählt inzwischen auch als dritte Säule der Wissenschaft neben dem Experiment und der Analytik [DGFK07]. In dieser Arbeit wird eine Implementierung der Lattice-Boltzmann Methode (LBM) vorgestellt, ein mesoskopischer Algorithmus, der genau für dieses Aufgabe geeignet ist.

In Kapitel 2 wird erläutert, worauf die Implementierung aufbaut. Einmal ist dies das Softwarepaket *ESPResSo* [ALK+13; LAMH06], das eine Vielzahl an Simulationen für den Bereich der weichen Materie und Molekulardynamik anbietet und durch die vorgestellte Implementierung erweitert wird. Zum anderen die *P4EST*-Bibliothek [BWG11], die eine skalierbare, adaptive Gitterstruktur auf Basis von Oktalbäumen bereitstellt, die Last auf verschiedene Prozesse verteilen kann und Nachbarschaftsbeziehungen effizient speichert. In Kapitel 3 geht es um die Herleitung der Boltzmann-Gleichung und ihre Diskretisierung zur Lattice-Boltzmann Methode. Begonnen wird dabei in der mikroskopischen Skala. Durch ein Verfahren namens *coarse-graining* werden den Gleichungen *unwichtige* Freiheitsgrade entzogen um dafür größere Zeit- und Längenabschnitte zu betrachten. Dadurch lässt sich die Boltzmann-Gleichung herleiten, die dann durch verschiedene Maßnahmen diskretisiert wird und mithilfe eines Gitters die Lattice-Boltzmann Methode ergibt.

In Kapitel 4 wird dann die eigentliche Implementierung vorgestellt. Diese basiert auf der adaptiven Lattice-Boltzmann Methode von [LABM16; LBH+16] und wird angepasst, so dass sie mithilfe von CUDA [Cor18b] parallelisiert berechnet werden kann. Dafür wird eine durch *P4EST* generierte Zelle nochmals durch ein reguläres Gitter in weitere Zellen unterteilt. Diese Zellen ergeben zusammen eine *Payload*, die dann an eine Grafikkarte geschickt wird und da als eine Einheit parallel verarbeitet wird. Dies funktioniert, da bei der LBM die meisten Berechnungen lokal, also unabhängig von der restlichen Simulation, stattfindet und für die übrigen Berechnungen nur der Datenaustausch mit den direkt benachbarten Zellen von Bedeutung ist. Die dafür notwendige Änderungen und die Hindernisse werden in diesem Kapitel beschrieben.

In Kapitel 5 wird überprüft, ob der Algorithmus die richtige Werte liefert anhand von bekannten und überschaubaren Simulationen. Dabei findet zum Teil ein Vergleich mit der analytischen Lösung als auch mit der adaptiven CPU-Implementierung statt. Darauf wird untersucht wie gut die Implementierung skaliert, wenn die Problemgröße anwächst oder mehrere Grafikkarten und CPU-Kerne benutzt werden.

Schließlich findet in Kapitel 6 eine Zusammenfassung statt und es wird ein Ausblick auf weiterführende Arbeiten zu der Implementierung gegeben.

2 Benutzte Software Pakete

Dieses Kapitel hat das Ziel, die zugrunde liegenden Software-Pakete bzw. Bibliotheken vorzustellen und einige wichtige Funktionalitäten wie die Baumstruktur der *P4EST*-Bibliothek zu erläutern. Die in der Arbeit vorgestellte Implementierung erweitert die Simulationssoftware *ESPResSo* (Extensible Simulation Package for Research on Soft Matter Systems) um eine Möglichkeit auf mehreren Grafikkarten in einem Computer oder in ganzen Clustern mit CUDA-fähigen Beschleunigerkarten die LBM zu berechnen. Für die Verwaltung des benötigten Gitters wird die *P4EST*-Bibliothek genutzt. Diese eignet sich ausgezeichnet für hohe Parallelität und erlaubt für die Zukunft eine Erweiterung auf eine adaptive Gitterstruktur, wie sie schon in einer CPU-Version für *ESPResSo* benutzt wird.

2.1 Extensible Simulation Package for Research on Soft Matter Systems

ESPResSo (Extensible Simulation Package for Research on Soft Matter Systems) ist ein open-source Softwarepaket unter *GPL*, das für numerische Molekulardynamik-Simulationen und Monte-Carlo Simulationen in einer parallelen Umgebung entwickelt worden ist [ALK+13; LAMH06]. Bei Monte-Carlo Simulationen versucht man analytisch schwierige Probleme mithilfe von vielen Zufallsexperimenten numerisch zu lösen. Unter Soft Matter versteht man hier Objekte, die sich weder wie eine reguläre Flüssigkeit verhalten noch wie solide Materie. Dazu gehören unter anderem Polymere, Flüssigkristalle, Gase, Kolloide, Polyelektrolyde, Ferrofluide oder Gele. Das *ESPResSo*-Projekt startete im Jahr 2001 und hat sich seitdem stetig weiterentwickelt und an Funktionalität hinzugewonnen. Dabei blieb der Fokus auf Geschwindigkeit und hoher Parallelität, so dass nun die meisten Simulationen sehr hoch skaliert werden können. So kommt die Software sowohl auf Heimcomputern, als auch auf Supercomputern zum Einsatz. Für die Verteilung und Kommunikation der Simulation auf verschiedenen Prozessoren nutzt *ESPResSo* das *Message Passing Interface* (MPI). Inzwischen wurden die *Core*-Funktionen erweitert und decken zB. auch die Elektrostatik oder Magnetostatik ab.

ESPResSo ist unterteilt in effiziente und optimierte Algorithmen für die einzelnen Simulationen und einem *TCL*- oder *Python*-Interpreter um diese Simulationen zu steuern. Die im Laufe der Arbeit behandelte LBM (Kapitel 3 auf Seite 23) gehört zu der Hydrodynamik und weicher Materie. Sie wurde auf unterschiedliche Weise für *ESPResSo* implementiert. 2008 wurde eine CPU-Implementation, die reguläre Gitter nutzt, vorgestellt [Sch08]. 2012 wurde es um eine GPU-Implementierung erweitert, die mit einer einzelnen GPU auf regulären Gittern arbeitet [RA12]. 2016 erschien eine auf *P4EST aufbauende Implementierung* [BWG11]. Diese nutzt ein dynamisch adaptives Gitter [LABM16; LBH+16]. 2017 wurde die adaptive LBM mit einer Partikel-Simulation gekoppelt [Bru17], basierend auf der Arbeit von [DL09].

2.2 Baum basierte Gitter (Octrees, Quadtrees) durch *P4EST*

Parallel adaptive mesh refinement on Forests of Octrees (*P4EST*) ist eine open-source (LGPL v2.1) Bibliothek, welche dynamisch eine Menge an adaptiven Quad- oder Oktalbäumen (Abschnitt 2.2.1) verwaltet [BWG11]. Der Begriff *Forest* kommt daher, da *P4EST* mit mehrere Bäumen arbeiten kann. *P4EST* wurde mit dem Ziel entwickelt, schnell und unter Ausnutzung von Parallelität und möglichst minimalen Datenaustausch *Forests* zu erstellen, diese dynamisch anzupassen, die aus den Bäumen resultierende Last gleichmäßig auf die Rechenressourcen zu verteilen und die Kommunikation zwischen den einzelnen Blättern zu bewerkstelligen. Dabei wurde demonstriert, dass *P4EST* auf über 450.000 Kerne mit MPI skaliert [IBWG15] und sogar in Programmen, die auf Supercomputern mit über 3.000.000 Hardware Threads zum Einsatz kommt (Blue Gene Q Mira) [MKM+15]. Der *Forest* besitzt eine statische Makrostruktur, die auf allen Prozessoren geteilt wird. Dabei wird das Simulationsgebiet Ω diskretisiert über einen oder mehrere disjunkte Oktalbäume $\bigcap_i o_i = \emptyset$, die dann auf mehrere Prozesse partitioniert werden können und zusammen durch Vereinigung $\bigcup_i o_i = \Omega$ die Makrostruktur ergeben. Die Mikrostruktur, also die Partitionierung der Bäume und die Lastverteilung zwischen den Prozessoren, ist hingegen dynamisch und wird parallel berechnet.

Die vorgestellte Implementierung unterstützt adaptive Gitter, hat aber keinen adaptiven Zeitschritt implementiert und nutzt daher ein durch *P4EST* erstelltes reguläres Gitter. Aus diesem Grund wird im restlichen Teil des Kapitels nur am Rande auf die adaptiven Eigenschaften von *P4EST* eingegangen. Die Beschreibung orientiert sich dabei primär an den Ausführungen von [BWG11].

2.2.1 Oktalbäume

Eine Datenstruktur, die häufig für dreidimensionale Daten benutzt wird, ist der Oktalbaum (engl. *octree*). Dies ist eine Baumstruktur, bei der jeder Knoten entweder keine oder 8 Nachfolger besitzt. Die Nachfolger, auch *Kinder* (engl. *child*) genannt, sind dann sogenannte Oktanten. Jeder Oktalbaum hat genau eine Wurzel, welche der Startpunkt des Baumes ist. Jeder Knoten hat einen Vorgänger, auch *Eltern* (engl. *parents*) genannt, während die möglichen Nachfolger eines Knotens untereinander Geschwister (engl. *siblings*) sind. Eine Ausnahme davon ist die Wurzel. Sie ist der Beginn des Baumes und hat keinen Vorgänger. Die Tiefe eines Knotens oder Blattes wird auch Level (l) genannt, wobei der Wurzel Level 0 zugewiesen wird.

Oktalbäume sind eine Struktur um einen dreidimensionalen Raum abzubilden. Die Wurzel entspricht dabei dem gesamten Raum Ω, häufig einem Würfel, während eine Unterteilung in Oktanten den Raum in 8 gleichgroße Teile aufspaltet. Hat der Raum eine andere Form, kann eine geometrische Transformation nötig sein. Dabei wird die Länge des Oktantens in jeder Dimension gegenüber der Länge des Raumes des Vorgängers halbiert. Wegen dieser gleichmäßigen Aufteilung sagt man auch, dass Oktalbäume eine natürliche Struktur für den dreidimensionalen Raum sind. Einzelne Oktanten können rekursiv in 8 feinere Oktanten aufgeteilt werden und so kann der Raum in ein adaptives Gitter unterteilt werden. Für ein maximales Level $l_{\max}$ existieren also höchstens $8^{l_{\max}}$ Blätter. Für genau $8^{l_{\max}}$ Blätter, liegen diese alle auf einem Level und es wird ein reguläres Gitter erhalten.

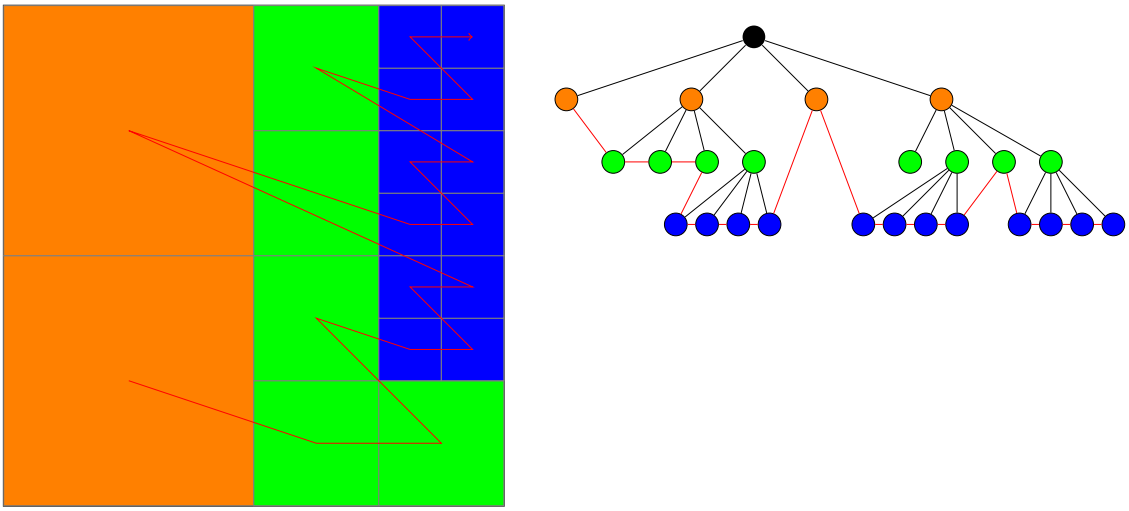


Abbildung 2.1: Das adaptive Gitter (links) wird durch den Quadtree (rechts) repräsentiert. Die rote *Morton*-Kurve entspricht dabei der Speicherreihenfolge der Blätter im Quadtree.

Für den zwei- und eindimensionalen Fall gibt es analog dazu den Quadtree und den Binärbaum, bei denen es jeweils 4 oder 2 Kinder gibt. Damit lassen sich dann die dazugehörenden Gebiete im selben Verfahren unterteilen. Das Prinzip lässt sich auch auf n-dimensionale Daten erweitern, wobei dann jeder Knoten entweder 0 oder 2^n Kinder besitzt.

P4EST erstellt mehrere solcher Oktalbäume, diese ergeben zusammen den *Forest*.

2.2.2 Koodierung des *Forests*

Das Ziel der Kodierung des *Forests* ist es die Oktalbäume möglichst kompakt abzuspeichern, so dass die gewünschten Informationen schnell/mit minimaler Zugriffszeit und wenig Kommunikations-Overhead extrahiert werden können. Dies geschieht unter anderem durch die *Morton*-Kodierung und die raumfüllende *Morton*-Kurve [Mor66], die im folgenden näher erklärt werden.

In *P4EST* werden die Blätter eines einzelnen Oktalbaumes jeweils in einem eigenen Array gespeichert (Abbildung 2.1). Dadurch muss nicht zu jedem Oktanten der dazugehörige Oktalbaum mit gespeichert werden. Als Array-Eintrag steht dann unter anderem die jeweilige *Morton*-Kodierung des Blattes, aus der man die Position bestimmen kann. Bei der Gebietszerlegung eines Oktalbaumes wird mit jedem weiteren Level die Seitenlänge der Oktanten halbiert gegenüber dem Vorgänger. Wird die Länge der Blätter h_l, die vom jeweiligen Level l abhängt, bei maximalen Level $l_{\max}$ auf $h_{\max} = 1$ gesetzt. Dann hat die Wurzel des Oktalbaumes die Seitenlänge $h_0 = 2^{l_{\max}}$ und allgemein gilt für die Länge eines Oktanten mit Level l:

$$h_{level} = 2^{(l_{\max} - l)} \tag{2.1}$$

Intuitiv würde man dies eventuell gerade andersrum definieren und $h_0 = 1$ setzen, in *P4EST* hat man sich aber bewusst für den gezeigten Weg entschieden, um die Zahlen als Integer speichern zu können und dadurch mögliche Gleitkomma-Rundungsfehler auszuschließen.

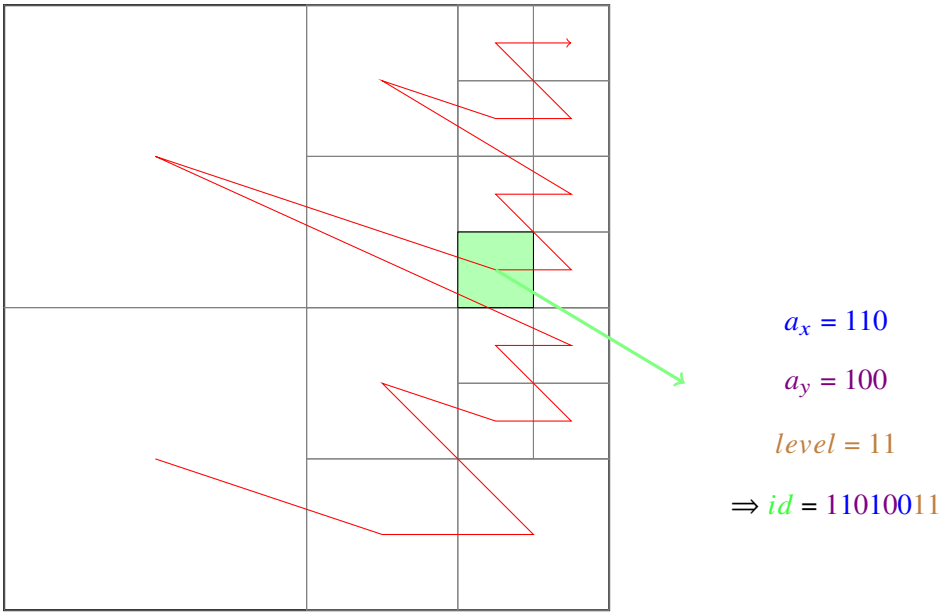


Abbildung 2.2: Mittels *Morton*-Kodierung erhält der grün eingefärbte Quadrant seine id. Die Oktanten *ankern* unten links.

Gespeichert wird nun von jedem Blatt die Position des linken unteren Ecks im Koordinatensystem des Oktalbaumes. Nun können die Koordinaten eines Blattes für jede Dimension als Vielfaches von $h_{\max}$ beziehungsweise 1 angegeben werden. Dadurch lässt sich mit der *Morton*-Kodierung eine eindeutige ID bestimmen. Da die linke untere Ecke der Blätter im Koordinatensystem gespeichert wird, beträgt die größtmögliche Koordinate, die ein Blatt erreichen kann, in einer Dimension $2^{l_{\max}} - 1 = h_0 - 1$, was sich mit der Anzahl von $l_{\max}$ Bits abspeichern lässt. Für ein Blatt erhält man für jede Dimension eine $l_{\max}$ lange Bitfolge $a[l_{\max}]$ für die Position im Raum. Diese Bitfolgen werden kombiniert zu einer neuen Bitfolge z. Im zweidimensionalen Fall wird jeweils von vorne nach hinten das Bit der 1.-Dimension an das Bit der 2.-Dimension angehängt, so dass sich $z = a_y[0]a_x[0]a_y[1]a_x[1]\ldots a_y[l_{\max}]a_x[l_{\max}]$ ergibt. Durch diese Kodierung wird die Position eindeutig gespeichert. Die noch fehlende Größe des Oktanten lässt sich durch das Level eindeutig bestimmen. Des Weiteren wird an z das Level als Binärzahl angegeben, dadurch ergibt sich eine eindeutige ID, mit der die Position und Größe bestimmt werden kann (Abbildung 2.2). Die benötigte Anzahl an Bits für die ID berechnet sich aus:

$$\begin{aligned} id &= z + l_{bit} \\ id &= dim \cdot l_{\max} + \lfloor \log_2(l_{\max}) \rfloor \end{aligned}$$

Wobei dim die Anzahl der Dimensionen und l_{bit} die Anzahl der benötigten Bits für $l_{\max}$ ist.

Bei *P4EST* beträgt das maximale Level 19 [BWG11; IBWG15]. Dadurch werden $id = 3 \cdot 19 + 5 = 62$ Bits benötigt um die ID zu speichern, welche durch einen 64 Bit langen Integer dargestellt werden kann. Mit zusätzlichen Informationen werden insgesamt für die Speicherung eines einzelnen Oktanten 24 Bytes in *P4EST* benötigt.

Das Durchlaufen des Gebietes nach der *Morton*-Kurve (Abbildung 2.1 auf Seite 17), die wegen ihrer Form auch *Z-Kurve* genannt wird, entspricht einer Tiefensuche auf dem Oktalbaum und der gespeicherten Reihenfolgen der Elemente auf dem Array

2.2.3 Ghost-Layer und Nachbarschaftsbeziehungen

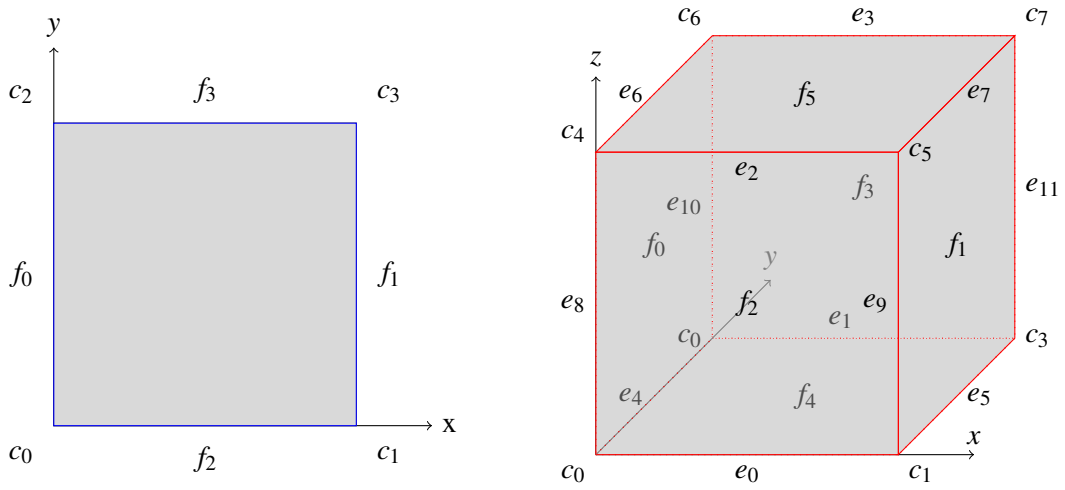


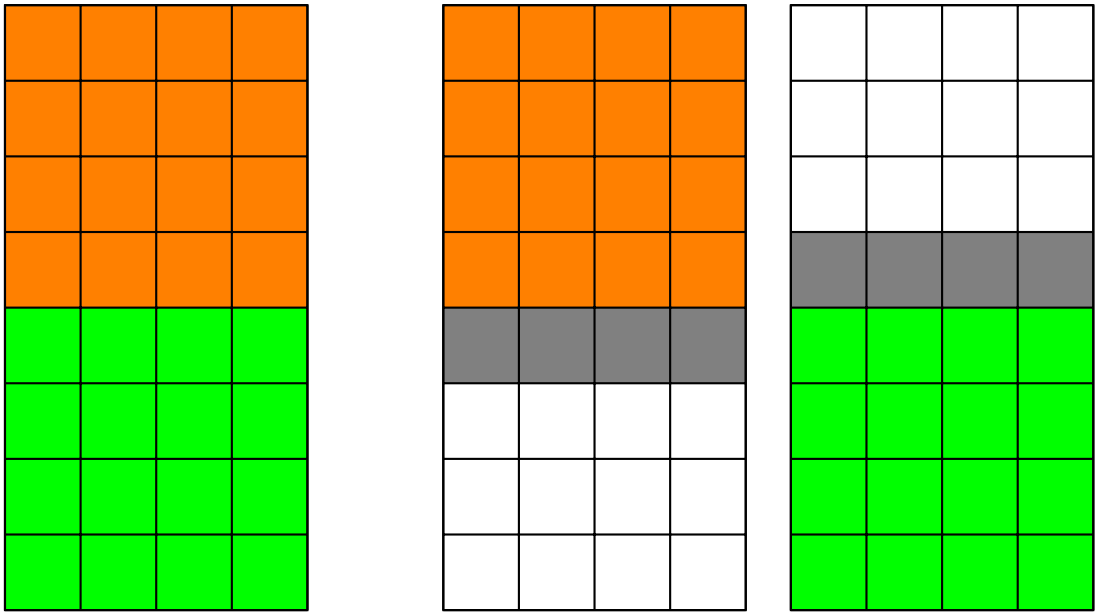
Abbildung 2.3: Bezeichung der Ecken c_i, der Flächen f_i und Kanten e_i für einen *P4EST* Quad- und Oktalbaum.

Bei Algorithmen, die auf Gittern basieren, ist es entscheidend, schnell und problemlos Daten zwischen Gitternachbarn auszutauschen. Dafür ist es wichtig die Nachbarn schnell zu finden, was bei einem regulären Oktalbaum und der *Morton*-Kodierung effizient durch Indexrechnungen möglich ist. Entspricht der Oktalbaum keinen regulären Gitter, muss eine Binärsuche durchgeführt werden:

Jedem Prozess wird eine Partition eines Oktalbaumes zugeteilt. Um zu vermeiden, dass innerhalb eines Simulationsschrittes Daten über die Prozesse hinaus ausgetauscht werden müssen, wird für diese ein sogenannter *Ghost-Layer* angelegt (Abbildung 2.4 auf der nächsten Seite). Hierbei wird um das Gebiet, welches der Prozess abdeckt, eine weitere Schicht an Blättern von benachbarten Prozessen hinzugefügt, so dass für alle direkten Nachbarabfragen der Blätter innerhalb des Prozesses keine weitere Kommunikation mit anderen Prozessen erforderlich ist. Dabei gilt es zu beachten, dass es sich bei den Blättern, die sich im Ghost befinden, bezüglich ihrer Größe und Nutzdaten um eine Duplizierung von bereit vorhanden Blättern handelt. Dadurch steigt der Speicherverbrauch und eventuell auch der Rechenbedarf.

Häufig werden in Simulationen bestimmte Nachbarn gesucht und es ist auch von Interesse, ob der Nachbar an eine Ecke, Fläche oder Kante angrenzt. Bei *P4EST* werden bei der Nachbarsuche aus diesem Grund die Verhältnisse zweier Oktanten zueinander direkt bestimmt. Ein Oktant hat dabei 8 Ecken, 6 Flächen und 12 Kanten, ein Quadrant 4 Ecken und 4 Kanten (bei *P4EST* werden diese Kanten auch faces genannt) (Abbildung 2.3).

Um nun einen Nachbarn zu finden, wird die *Morton*-Kodierung (Abbildung 2.2 auf der vorherigen Seite) betrachtet. Dabei kann auf jede Dimension der Koordinaten die Länge h_{level} (2.1) addiert oder subtrahiert werden. Wird die x-Koordinate um h_{level} erhöht, wird der Nachbar zu f_1 gefunden,



(a) Ein Gebiet wird in 2 Quadtrees (orange und grün) aufgeteilt.

(b) Der orangene Quadtree mit Ghostlayer (grau)

(c) Der grüne Quadtree mit Ghostlayer (grau)

Abbildung 2.4: Ein einfaches Beispiel für einen Ghost-Layer

wird die x-Koordinate und die z-Koordinate um jeweils h_{level} verringert, wird der Nachbar zu c_0 gefunden (Abbildung 2.3 auf der vorherigen Seite). Durch die adaptive Gitterstruktur kann es sein, wenn kein reguläres Gitter vorliegt, dass ein so berechneter Nachbar nicht existiert. Hat der Oktalbaum eine *2:1 Balance* (Abschnitt 2.2.4) ergeben sich 2 mögliche Fälle. Der Nachbar hat entweder ein um 1 größeres oder kleineres Level, was sich durch die id eindeutig bestimmen lässt. Existiert die Koordinate, aber das Level passt nicht, so hat der Nachbar ein höheres Level, existiert keine id zu den Koordinaten, hat der Nachbar ein niedrigeres Level. In beiden Fällen lässt sich weiterhin die id eindeutig bestimmen. *P4EST* stellt für die Nachbarsuche verschiedene Funktionen bereit, wie einen Iterator, der über die einzelnen Blätter iteriert. Diese Funktionen erlauben eine unkomplizierte und schnelle Implementierung.

2.2.4 Weitere Eigenschaften von *P4EST*

P4EST verteilt, falls gewünscht, die Last optimal auf verschiedenen Prozessen. Zusätzlich können die Bäume dynamisch adaptiv angepasst werden, um so zu jedem Zeitschritt eine geeignete Auflösung zu finden. Gleichzeitig kann *P4EST* eine *2:1 Balance* in den Oktalbäumen einhalten. Das bedeutet, dass für alle Zellen die Nachbarzellen maximal 1 Level höher oder niedriger sein dürfen, damit zwischen Nachbarn maximal ein Größenunterschied im Verhältnis von $2:1$ besteht.

Die aktuelle Implementierung nutzt ein reguläres Gitter, folglich muss nur einmal das Simulationsgebiet Ω in Bäume aufgeteilt und diese möglichst balanciert an die einzelnen Prozesse partitioniert werden. Dabei ist es wichtig, die notwendige Kommunikation zwischen den Prozessoren gering zu halten. Daraus resultiert, dass der *Ghost-Layer* möglichst klein ausfallen sollte, da in vielen Algorithmen nur dort in jedem Simulationsschritt Daten ausgetauscht werden müssen. Danach

sollte die Partitionierung statisch sein und sich nicht mehr verändern. Aus diesen Gründen wird auf die Algorithmen, die primär bei dynamisch adaptiven Gittern benötigt werden, nicht weiter eingegangen.

3 Die Lattice-Boltzmann Methode

In diesem Kapitel wird die Herleitung der Boltzmann-Gleichung beschrieben und wie sich daraus die LBM entwickelte, wie sie später in der Implementierung verwendet wird. Das folgende Kapitel basiert auf den Arbeiten von [Suc13], [Wol00], [Sch08] und [LBH+16].

3.1 Grundlegende Physik hinter der Boltzmann-Gleichung

3.1.1 *Newton*-Gleichung

Bewegen sich N Moleküle mit ihrer jeweiligen Masse m in einem Raum mit Volumen V, so lässt sich dieses System unter der Bedingung, dass nur jeweils 2 Moleküle gleichzeitig durch harte und elastische Stöße in Wechselwirkung (ideales Gas) stehen, durch die klassische *Newton*-Gleichungen beschreiben

$$\frac{\mathrm{d}\vec{x}_i(t)}{\mathrm{d}t} = \frac{\vec{p}_i(t)}{m}, \tag{3.1}$$

$$\frac{\mathrm{d}\vec{p}_i(t)}{\mathrm{d}t} = \vec{F}_i(t),\ i = 1, \ldots, N, \tag{3.2}$$

wobei t die Zeit, $\vec{x}_i$ die zeitabhängige Partikelposition, $\vec{p}_i \equiv m\vec{v}_i$ der zeitabhängige Impuls und $\vec{F}_i$ die zeitabhängige Kraft, die auf ein Partikel i wirkt, sind.

Die Kraft setzt sich zusammen aus den intermolekularen Kräften und möglichen externen Kräften wie der Gravitation oder elektrischen Feldern. Werden einzelne molekulare oder atomare Interaktionen ausgewertet, wird die sogenannte *mikroskopische*-Struktur betrachtet. Das Problem ist die sehr große Anzahl N der zu betrachtenden Moleküle bei Flüssigkeiten oder weicher Materie, welche meistens in der Größenordnung der Avogadro-Konstante $N_A \approx 6,02 \cdot 10^{23}$ liegt. Simulationen mit dieser Anzahl an Teilchen sind selbst für die aktuellen Supercomputer mit über 10^{15} Gleitkommaoperationen pro Sekunde nicht durchzuführen und übersteigen deren Speicherplatz.

3.1.2 *Navier-Stokes*-Gleichungen

Die Hydrodynamik ist ein Teilgebiet der Strömungslehre, das sich mit den Bewegungen und Kräften von Fluiden beschäftigt. Hydrodynamische Interaktionen werden durch ein Strömungsfeld vermittelt und können mit den *Navier-Stokes*-Gleichungen beschrieben werden. Wird davon ausgegangen, dass die Temperatur überall gleich ist, lauten die Gleichungen

$$\frac{\partial}{\partial t}\rho + \frac{\partial}{\partial \boldsymbol{r}}\rho\boldsymbol{u} = 0, \tag{3.3}$$

$$\frac{\partial}{\partial t}\rho\boldsymbol{u} + \frac{\partial}{\partial \boldsymbol{r}} \cdot (\rho\boldsymbol{u} \otimes \boldsymbol{u}) = -\frac{\partial}{\partial \boldsymbol{r}}p + \frac{\partial}{\partial \boldsymbol{r}} \cdot \sigma + g, \tag{3.4}$$

Dabei ist ρ die Dichte des Fluids, $\rho\boldsymbol{u} = \boldsymbol{j}$ die Impulsdichte, σ der deviatorische Spannungstensor, der wiederum von der Viskosität abhängt, und g eine externe Kraft (zum Beispiel die Schwerkraft). Diese Gleichung beschreibt das Fluid als *Kontinuum* auf größeren Zeit- und Längen-Skalen. Dabei wird nicht die Interaktion einzelner Atome betrachtet, sondern aus der Dichte ρ und der Impulsdichte $\boldsymbol{j}$ werden die Eigenschaften bestimmt. Dies nennt man eine makroskopische Beschreibung. Die *Navier-Stokes*-Gleichungen sind sehr allgemein und gelten für viele Fluide, auch wenn diese sich in den mikroskopischen Abläufen unterscheiden können.

Gleichung (3.3) entspricht dabei dem Massenerhalt in einem kontinuierlichem Fluid und Gleichung (3.4) entspricht dabei dem *newtonschen* Gesetz $\vec{F} = m\vec{a}$ [Suc13]. Die mikroskopischen Einzelheiten werden dabei durch die Transportkoeffizienten subsumiert und tauchen in den makroskopischen Gleichungen so nicht mehr auf. Dadurch werden der Gleichung überflüssige Freiheitsgrade entzogen. Dies bedeutet, dass verschiedene Modelle, die aus dem mikroskopischen Bereich hergeleitet werden können, in der Strömungsmechanik Verwendung finden können, aber auf makroskopischer Ebene müssen sie sich an den *Navier-Stokes*-Gleichungen und der Wirklichkeit messen lassen. Die Entwicklung solcher Modelle, die alle relevanten Größen erhalten, während überflüssige Freiheitsgrade vernachlässigt werden, nennt man *coarse-graining*.

3.1.3 Die *Boltzmann*-Gleichung

Durch *coarse-graining* kann die *Boltzmann*-Gleichung hergeleitet werden. Den Anfang bilden dabei die *newtonschen Gleichungen* (3.1) und (3.2). Bei diesen werden noch alle Interaktionen der Partikel einzeln berechnet.

Wird nun eine Vergröberung durch Liouvilles Theorie vorgenommen, werden nicht einzelne Teilchen, sondern eine Verteilungsdichtefunktion $f(\boldsymbol{r}, \boldsymbol{v}, t)$ betrachtet, welche die Wahrscheinlichkeit angibt ein Partikel mit Geschwindigkeit $\boldsymbol{v}$ am Ort $\boldsymbol{r}$ zum Zeitpunkt t zu treffen. Übertragen auf die *newtonsche Gleichungen* ergibt sich eine neue probabilistische Beschreibung

$$\left(\frac{\partial}{\partial t} + \boldsymbol{v} \cdot \frac{\partial}{\partial \boldsymbol{r}} + \frac{\boldsymbol{F}}{m} \cdot \frac{\partial}{\partial \boldsymbol{v}}\right) f(\boldsymbol{r}, \boldsymbol{v}, t) = 0, \tag{3.5}$$

die Liouville Gleichung. Hierbei wird aber nur beschrieben, wie sich die Partikel entlang ihrer Bahn ausbreiten ohne Beachtung der mikroskopischen Interaktionen der einzelnen Partikel. Dafür wird nun ein Kollisionsterm $C_{coll}(f)$ eingeführt. Daraus ergibt sich die Gleichung

$$\left(\frac{\partial}{\partial t} + \boldsymbol{v} \cdot \frac{\partial}{\partial \boldsymbol{r}} + \frac{\boldsymbol{F}}{m} \cdot \frac{\partial}{\partial \boldsymbol{v}}\right) f(\boldsymbol{r}, \boldsymbol{v}, t) = C_{coll}(f), \tag{3.6}$$

wobei der Kollisionsterm beschreibt, wie sich Verteilungen durch mikroskopische Interaktionen der Partikel ändern.

Dabei beschreibt C_{coll} an sich die Zwei-Teilchen-Verteilungsfunktion. Diese gibt Aufschluss darüber wie hoch die Wahrscheinlichkeit ist zur gleichen Zeit ein Partikel (1) am Ort $\boldsymbol{r_1}$ mit Geschwindigkeit $\boldsymbol{v_1}$ und ein Partikel (2) am Ort $\boldsymbol{r_2}$ mit Geschwindigkeit $\boldsymbol{v_2}$ zu finden. Die Zwei-Teilchen-Verteilungsfunktion ist aber abhängig von der Drei-Teilchen-Verteilungsfunktion, welche wiederum abhängig ist von der Vier-Teilchen-Verteilungsfunktion und so weiter. Diese unendliche Hierarchie von Gleichungen wird als BBGKY-Hierarchie bezeichnet und ergibt sich aus den 6 vollen Freiheitsgraden der Verteilungsfunktion im Phasenraum (3 Dimensionen für den Impuls, 3 für den Raum) der Liouville-Gleichung. Wird das Modell aber vereinfacht durch die Annahme, dass sich die Partikel wie Punkte verhalten und die Kollisionen untereinander binär und unkorreliert seien, während sich die Partikel die meiste Zeit über auf freien Bahnen ungestört voneinander bewegen, ergibt sich die gesuchte *Boltzmann*-Gleichung. Diese Annahme nennt sich molekulares Chaos oder auch Stoßzahlansatz. Die *Boltzmann*-Gleichung lautet dann wie folgt [Suc13]

$$\left(\frac{\partial}{\partial t} + \boldsymbol{v} \cdot \frac{\partial}{\partial \boldsymbol{r}} + \frac{\boldsymbol{F}}{m} \cdot \frac{\partial}{\partial \boldsymbol{v}}\right) f(\boldsymbol{r}, \boldsymbol{v}, t) = \int (f(\boldsymbol{r}, \boldsymbol{v}_1\prime, t) f(\boldsymbol{r}, \boldsymbol{v}_2\prime, t) - f(\boldsymbol{r}, \boldsymbol{v}_2, t) f(\boldsymbol{r}, \boldsymbol{v}_1, t)) \boldsymbol{g} \sigma(\boldsymbol{g}, \Omega) d\Omega \vec{p}_2, \quad (3.7)$$

dabei ist $\boldsymbol{g} = \boldsymbol{v_1} - \boldsymbol{v_2}$ die relative Geschwindigkeit vor der binären Kollision, $\boldsymbol{v}_1\prime$ und $\boldsymbol{v}_2\prime$ die Geschwindigkeiten nach der Kollision, Ω der Streuwinkel und $\sigma(\boldsymbol{g}, \Omega)$ der differentielle Wirkungsquerschnitt.

Die Verteilungsfunktion beschreibt durch ihre Momente auch messbare makroskopische Größen, wie die Massendichte (ρ, die eigentlich ein *nulltes*-Moment ist), die Impulsdichte ($\rho \boldsymbol{u}$) und die Energiedichte (ρe)

$$m \int f(\boldsymbol{r}, \boldsymbol{v}, t) d\boldsymbol{v} = \rho(\boldsymbol{r}, t), \quad (3.8)$$

$$m \int \boldsymbol{v} f(\boldsymbol{r}, \boldsymbol{v}, t) d\boldsymbol{v} = \rho \boldsymbol{u}(\boldsymbol{r}, t), \quad (3.9)$$

$$m \int \frac{|\boldsymbol{v}|^2}{2} f(\boldsymbol{r}, \boldsymbol{v}, t) d\boldsymbol{v} = \rho e(\boldsymbol{r}, t). \quad (3.10)$$

3.1.4 Gleichgewichtsverteilungen

Für hydrodynamische Interaktionen spielt die *lokale Gleichgewichtsverteilung* $f^{eq}(\boldsymbol{v})$ eine entscheidende Rolle. Gibt es keine Krafteinwirkungen von außen, wird eine *lokale Gleichgewichtsverteilung* erreicht, die unabhängig von $\boldsymbol{r}$ und t ist, wenn sich der Kollisionsterm aufhebt:

$$C_{coll}(f) = 0 \quad (3.11)$$

Um dies zu erfüllen, muss nach dem Kollisionsterm aus (3.7)

$$f(\boldsymbol{r}, \boldsymbol{v}_1\prime, t) f(\boldsymbol{r}, \boldsymbol{v}_2\prime, t) = f(\boldsymbol{r}, \boldsymbol{v}_2, t) f(\boldsymbol{r}, \boldsymbol{v}_1, t) \tag{3.12}$$

sein, was auch ein *detailliertes Gleichgewicht* genannt wird [Suc13]. Dies bedeutet, dass sich die Kollisionen dynamisch ausgleichen und so zu jeder Kollision eine inverse Kollision stattfindet.

Aus dem *detailiertem Gleichgewicht* folgt, dass der Logarithmus von $f(\boldsymbol{r}, \boldsymbol{v}, t)$ eine additive Invariante ist

$$\ln f(\boldsymbol{r}, \boldsymbol{v}_1\prime, t) + \ln f(\boldsymbol{r}, \boldsymbol{v}_2\prime, t) = \ln f(\boldsymbol{r}, \boldsymbol{v}_2, t) + \ln f(\boldsymbol{r}, \boldsymbol{v}_1, t) \tag{3.13}$$

und daher innerhalb eines thermodynamischen Gleichgewichts $\ln(f(\boldsymbol{r}, \boldsymbol{v}, t)$ nur aus Kollisionsinvarianten $\boldsymbol{I}(\boldsymbol{v}) = [1, m\boldsymbol{v}, m\boldsymbol{v}^2/2]$ (Zahl, Moment, Energieerhaltung) bestehen darf.

Dies ergibt die Linearkombination

$$\ln f(\boldsymbol{r}, \boldsymbol{v}_1\prime, t) = A + \boldsymbol{B}\boldsymbol{v} + \frac{1}{2} C\boldsymbol{v}^2, \tag{3.14}$$

in die nun die makroskopische Massendichte ρ (3.8), Impulsdichte $\rho\boldsymbol{v}$ (3.9)und Energiedichte ρe (3.10) eingesetzt werden können.

Daraus ergibt sich die *Maxwell-Boltzmann-Gleichgewichtsverteilung*

$$f^{eq} = \left(\frac{m}{2\pi k_B T}\right)^{\frac{3}{2}} \frac{\rho}{m} \exp\left(-\frac{m(\boldsymbol{v} - \boldsymbol{u})^2}{2k_B T}\right), \tag{3.15}$$

wobei k_B die Boltzmann Konstante ist.

Um diese Gleichung analytisch oder auch numerisch besser lösen zu können, werden weitere Vereinfachungen vorgenommen. So wird angenommen, dass die eigentliche Verteilungsdichtefunktion nur leicht von der lokalen Gleichgewichtsverteilung abweicht. Dies ergibt

$$f = f^{eq} + f^{neq}, \tag{3.16}$$

dabei ist f^{neq} die Abweichung zur Gleichgewichtsverteilung. Bei Werten um das Gleichgewicht kann der Kollisionsterm dadurch linearisiert werden. Zusätzlich wird dieser durch den BGK (Bhatnagar-Gross-Krook)-Kollisionsoperator vereinfacht [BGK54], bei dem die zugrunde legende Physik weitgehend erhalten bleibt:

$$C_{BGK}(f) = -\frac{f - f^{eq}}{\tau_{rel}} = -\frac{f^{neq}}{\tau rel} \tag{3.17}$$

Hierbei ist τ_{rel} eine zeit abhängige Einheit, die angibt wie lange die Partikel brauchen um den Gleichgewichtszustand zu erreichen. Der BGK Kollisionsoperator vereinfacht diesen (eigentlich nur schwer zu berechnenden) τ_{rel}-Term durch eine Konstante.

Um den Übergang von mikroskopisch zu makroskopisch abzuschließen, lässt sich Mithilfe der Chapman-Enskog-Erweiterung beweisen, dass die *Boltzmann*-Gleichung die *Navier-Stokes*-Gleichungen approximieren [Li15].

3.2 Diskretisierung der *Boltzmann*-Gleichung

Um eine numerische Lösung für die Boltzmann-Gleichung zu erhalten, muss der 7-dimensionale Phasenraum diskretisiert werden. Ohne fremde Krafteinwirkungen und mit dem BGK Kollisionsoperator lautet die Gleichung

$$\frac{\partial}{\partial t} f + \boldsymbol{v} \frac{\partial}{\partial \boldsymbol{r}} f = -\frac{f - f^{eq}}{\tau_{rel}}, \tag{3.18}$$

die Bedeutung der Variablen bleibt dabei unverändert zum vorherigen Abschnitt. Zuerst wird die Raumgeschwindigkeit durch eine Gauss-Hermite Quadratur ersetzt [Sch08]. Für diskrete Geschwindigkeiten c_i lautet die Gleichung dann

$$\frac{\partial}{\partial t} f_i + \boldsymbol{c}_i \frac{\partial}{\partial \boldsymbol{r}} f_i = -\frac{f - f^{eq}}{\tau_{rel}}, \tag{3.19}$$

wobei f_i nun nicht mehr von $\boldsymbol{v}$ abhängig ist, sondern nur noch von $\boldsymbol{r}$ und t. Die Gleichung ergibt dadurch Verteilungsfunktionen f_i mit der zugehörigen Geschwindigkeit c_i.

Die *Maxwell-Boltzmann-Gleichgewichtsverteilung* kann ebenso durch ein hermitesches Polynom zweiter Ordnung erweitert werden

$$f_i^{eq} = \omega_i \rho \left(1 + \frac{\boldsymbol{u} \cdot \boldsymbol{c}_i}{c_s^2} + \frac{(\boldsymbol{u} \cdot \boldsymbol{c}_i)^2}{2c_s^4} - \frac{|\boldsymbol{u}|^2}{2c_s^2} \right) \tag{3.20}$$

mit einem Koeffizienten ω_i und der Schallgeschwindigkeit c_s. Die makroskopische Beschreibung der Hydrodynamik ändert sich dadurch nicht und die Massendichte ρ, die Impulsdichte $\rho\boldsymbol{u}$ und der Energie-Impuls-Tensor Π lassen sich aus den mikroskopischen Populationen beschreiben als

$$\rho = \sum_i f_i, \tag{3.21}$$

$$\rho \boldsymbol{u} = \sum_i f_i \boldsymbol{c}_i, \tag{3.22}$$

$$\Pi = \sum_i f_i \boldsymbol{c}_i \otimes \boldsymbol{c}_i, \tag{3.23}$$

und bilden somit eine Summe über die einzelne Verteilungsfunktionen f_i.

Um Raum und Zeit zu diskretisieren, werden die schon diskretisierten Geschwindigkeiten $\boldsymbol{c}_i$ durch gewöhnliche Differentialgleichung erster Ordnung ersetzt. Für kleine Zeitschritte kann die gewöhnliche Differentialgleichung durch ein Integral approximiert werden, wodurch man die durch BGK-Operator angenäherte *Lattice-Boltzmann*-Gleichung erhält

$$f_i(\boldsymbol{r} + \tau \boldsymbol{c}_i, t + \tau) = f_i(\boldsymbol{r}, t) + \frac{1}{\tau_{rel}}(f_i(\boldsymbol{r}, t) - f_i^{eq}(\boldsymbol{r}, t)), \tag{3.24}$$

mit dem Zeitschritt τ, wobei anstelle des BGK-Operators auch andere Kollisionsoperatoren genutzt werden können. Die Geschwindigkeiten $\boldsymbol{c}_i$ werden so gewählt, dass pro Zeiteinheit die Partikel nur ein benachbartes Gitterfeld erreichen können.

3.3 Die Lattice-Boltzmann-Methode

Um die diskretisierte *Boltzmann Gleichung* numerisch lösen zu können, muss sie noch mit einer passenden Gitterstruktur gekoppelt werden. Wie wichtig das Finden eines solchen Gitters ist, zeigt sich in der historischen Entwicklung der *Lattice-Gas*-Automaten, aus denen sich die LBM entwickelte.

3.3.1 Gittermodelle

Ein Lattice Gas Automata (LGA) ist ein zellulärer Automat und wird zur Simulation von Fluiden benutzt, dabei wird ein diskretes mikroskopisches Modell verwendet. Es gibt verschiedene Modelle an LGAs. Motiviert wurde die Entwicklung aus der Beobachtung, dass sich makroskopische Eigenschaften bei Gasen und Flüssigkeiten trotz unterschiedlichen mikroskopischen Verhaltens kaum unterscheiden. Die Partikel bewegen sich dabei auf den Gitterpunkten entlang der Gitterkanten und mit einer Geschwindigkeit, dass pro Zeitschritt die Partikel den nächsten Gitterpunkt erreichen. Auf jedem Gitterpunkt kann sich zu jeder ausgehende Kante jeweils ein oder kein Partikel befinden. Weiterhin haben alle Partikel die gleiche Masse. Wenn sich die Partikel treffen kommt es zu einer Kollision oder auch Streuung, bei der Masse und Impuls erhalten bleiben. Einige Modelle können von diesen grundsätzlichen Eigenschaften abweichen. Eines der ersten Modelle, das sogenannte HPP Modell [Suc13], benutzt ein quadratisches Gitter mit jeweils 4 Nachbarn und kann dadurch die *Navier-Stokes*-Gleichungen nicht nachbilden, weil es keine ausreichende Rotationsinvarianz besitzt. Das FHP Modell [FHP86] umgeht das Problem durch ein dreieckiges Gitter mit 6 Nachbarn und kann dadurch im zweidimensionalen die *Navier-Stokes*-Gleichungen reproduzieren.

Für die LBM gibt es eine Reihe von Gittermodellen, wie das D2Q9 Gitter oder das D3Q19 Gitter, das in *ESPResSo* verwendet wird. Die Zahl nach dem D steht dabei für die Dimension des Gitters und der Eintrag nach dem Q für die Anzahl diskreter Geschwindigkeitsvektoren.

Für das D3Q19 Modell werden die einzelnen $\boldsymbol{c}_i$ so gewichtet

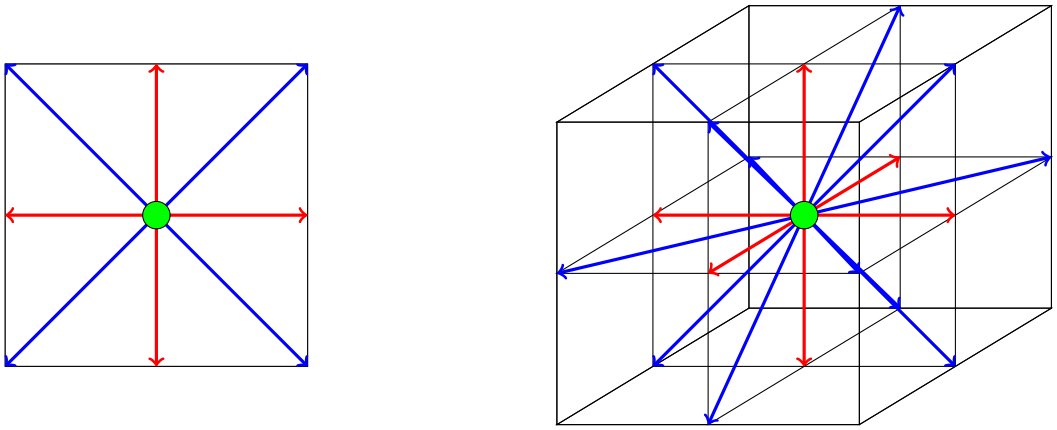


Abbildung 3.1: Auf der linken Seite ist das D2Q9 Modell abgebildet mit seinen 9 Geschwindigkeitsvektoren, 4 Vektoren auf die Ecken, 4 Vektoren auf die Kante und 1 Vektor auf sich selbst. Auf der rechten Seite ist das D3Q19 Modell abgebildet. Dieses hat einen Vektor auf sich selbst (c_0), 6 Vektoren auf die Flächen ($c_{1..6}$) und 12 Vektoren auf die Kanten ($c_{7..18}$).

$$\omega_0 = \frac{1}{3}, \tag{3.25}$$

$$\omega_{1..6} = \frac{1}{18}, \tag{3.26}$$

$$\omega_{7..18} = \frac{1}{36}. \tag{3.27}$$

3.3.2 Aufteilung in Kollisions- und Strömungsschritt

Die LBM (3.24) kann in 2 Schritte aufgeteilt werden. Zuerst wird lokal für jede einzelne Zelle ein Kollisionsschritt durchgeführt, bei dem die vorhandenen Partikelpopulationen durch Relaxation zu ihrem lokalen Gleichgewicht übergehen.

Im zweiten Schritt werden die entspannten Populationen gemäß ihrem Geschwindigkeitsvektor an die Nachbarzellen übergeben, wo diese dann wieder für den ersten Schritt verwendet werden.

Dies sieht dann so aus:

$$f_i^*(\boldsymbol{r}, t) = f_i(\boldsymbol{r}, t) + \frac{1}{\tau_{rel}}(f_i(\boldsymbol{r}, t) - f_i^{eq}(\boldsymbol{r}, t)), \tag{3.28}$$

$$f_i(\boldsymbol{r} + \tau \boldsymbol{c}_i, t + \tau) = f_i^*(\boldsymbol{r}, t) \tag{3.29}$$

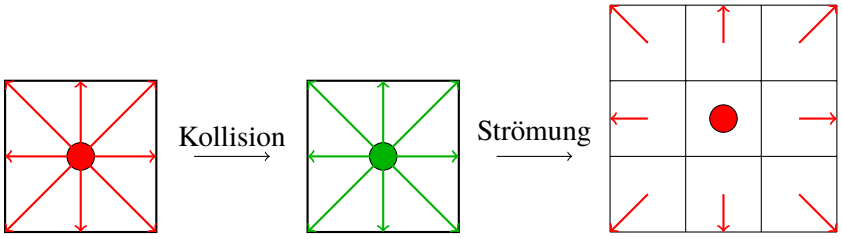
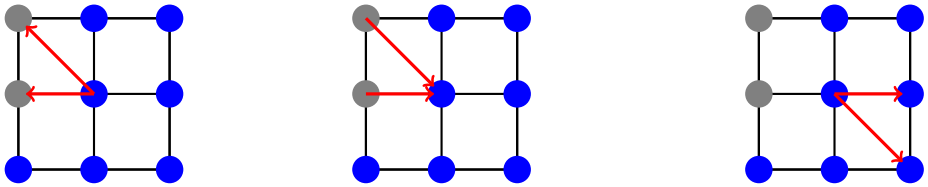


Abbildung 3.2: Ein LBM Schritt mit einem D2Q9 Gitter. Im Kollisionsschritt wird das Gleichgewicht für die Populationen und ihre Geschwindigkeitsvektoren berechnet. Im Strömungsschritt wandern die entspannten Populationen dann zu den entsprechenden Nachbarpunkten.

3.3.3 Randbedingungen

Damit Simulationen richtig funktionieren können, benötigen sie ein Gebiet in dem sie ablaufen. Zusätzlich werden Begrenzungen wie Wände benötigt. Daher ist es wichtig simulieren zu können, wie sich Fluide verhalten, wenn sie auf einen Festkörper treffen. Das Fluid darf nicht in den Rand und in der Regel auch nicht in einen Festkörper hineinströmen (Ausnahme: poröses Material) und trotzdem sollte die makroskopische Struktur sich gemäß den Beobachtungen verhalten. Dafür werden Regeln aufgestellt, die sogenannten Randbedingungen.

Es gibt eine ganze Reihe verschiedener Randbedingungen, in der Implementierung werden dabei *no-slip bounce-back* Randbedingungen genutzt. Dabei wird zuerst überprüft, ob ein Gitterpunkt zu einem soliden Gegenstand wie einer Wand gehört. Ist dies der Fall, werden dort alle Fluid-Geschwindigkeiten auf 0 gesetzt und der Gitterpunkt ist *trocken (dry)*, daher auch der Name *no-slip*, die nicht davon betroffenen Gitterpunkte werden als *wet* bezeichnet. Weil auch sich bewegende Wände berücksichtigt werden, muss dieser Schritt vor jedem Iterationsschritt ausgeführt werden. Treffen im Strömungsschritt Partikelpopulationen auf einen soliden Gitterpunkt, so prallen diese wie in Abbildung 3.3 ab.



(a) Populationen treffen auf die *dry nodes*

(b) Und prallen an ihnen unter gleichem Winkel zurück.

(c) Richtungen der Populationen vor dem nächsten Kollisionsschritt.

Abbildung 3.3: *no-slip bounce-back* Randbedingungen mit blauen *wet nodes* und grauen *dry nodes* in zeitlicher Abfolge

4 Implementierung

Die Aufgabe war es eine weitere Variante der Lattice Boltzmann Methode für das Softwarepaket *ESPResSo* zu implementieren, die mehrere Grafikkarten (*graphics processing units* (GPUs)), für die Berechnung nutzen kann. Dabei wurde sich an der vorhandenen Implementierung, die den Simulationsraum mittels adaptiven Gittern diskretisiert, orientiert. Die jetzige Implementierung unterstützt dabei noch ein adaptives Gitter.

Wie in Kapitel 3.3.2 erwähnt, besteht ein Zeitschritt aus zwei verschiedenen Schritten, die gemeinsam Integrationsschritt genannt werden:

1. Der Kollisionsschritt. Dieser ist der rechenintensive Schritt und läuft komplett lokal ab, dadurch lässt sich dieser Schritt perfekt parallelisieren.
2. Der Strömungsschritt. Hier werden Daten nur von den direkten Nachbarn benötigt. Das Ziel ist es also eine Aufteilung zu finden, die es erlaubt schnell und unkompliziert auf direkte Nachbarn zuzugreifen.

Diese beiden Schritte sollten mithilfe von GPUs beschleunigt werden.

In diesem Kapitel wird zuerst auf die Besonderheiten von CUDA und GPUs eingegangen und wie die vorhanden Strukturen erweitert oder modifiziert werden müssen, damit diese für eine parallele Verarbeitung für GPUs geeignet sind. Danach wird ein Simulationsablauf betrachtet und anhand dessen einen grober Überblick über die Implementierung gegeben. Im weiteren Verlauf wird dann auf einzelne Herausforderungen genauer eingegangen.

4.1 CUDA und GPUs

CPUs wurden entwickelt mit dem Ziel serielle Algorithmen möglichst schnell abarbeiten zu können. Verbesserungen fanden bei CPUs lange hauptsächlich dadurch statt, dass die Fertigungsmethoden besser und kleiner wurden, so dass mehr Platz für zusätzliche Transistoren vorhanden war. Dies wurde dann genutzt, um die IPC (instruction per clock) und den Takt, mit dem die CPU Rechenschritte ausführt, zu erhöhen. Dieses Vorgehen stößt aber immer mehr an seine Grenzen und deckt nicht mehr den Rechenbedarf. Daher wurden Multi-Core Architekturen eingeführt, bei denen eine CPU mehrere Recheneinheiten bekommt. Zusätzlich gibt es ein Verfahren Namens SIMD (Single instruction, multiple data), bei dem die Datenparallelität ausgenutzt wird um zum Beispiel bei der Addition zweier Vektoren in einer Instruktion mehrere Rechenschritte auf einmal auszuführen. Für Intel und AMD Prozessoren heißen diese Instruktionen Advanced Vector Extensions (AVX). Diese beide Änderungen machen moderne CPUs deutlich schneller bei vielen parallelen Algorithmen, ihre Wurzeln liegen aber nach wie vor bei der schnellen Ausführung von seriellen Algorithmen.

GPUs haben einen anderen Ursprung. Sie wurden konzipiert um für Computerspiele Berechnungen zur Grafik zu machen. In der Computergrafik werden dabei meistens direkt mit tausenden Objekten oder Bildpunkten gleichzeitig gearbeitet, die dann in Echtzeit ein Bild erzeugen sollen. Dies erforderte schon früh eine spezielle Rechenarchitektur, welche die Berechnungen parallel abarbeitet. Die theoretische Rechenleistung überstieg bei hoher Datenparallelität schon bald die Leistung vergleichbarer CPUs [Cor18b].

Der Grafikkartenhersteller *Nvidia* veröffentlichte 2007 eine erste Version von *CUDA*, eine Architektur für parallele Berechnungen mit Grafikkarten über Spielegrafik hinaus. *Nvidia* stellt aber nicht nur die CUDA Architektur, sondern auch die CUDA Plattform und das CUDA Toolkit, die gemeinsam alles liefern, um für Grafikkarten optimierte Programme zu erstellen. CUDA wird seitdem in vielen Anwendungen genutzt, die sich parallel gut verarbeiten lassen, wie Bild- und Videobearbeitung, numerische Strömungssimulationen, seismische Analysen und vielen mehr. Dabei werden Programmiersprachen durch einige C und C++ Erweiterungen ergänzt mit denen man relativ einfach parallele Aufrufe auf der GPU starten kann. Diese werden *Kernel*s genannt und lassen sich über

```
1  NameKernel<<<N, M>>>(A,B,C);
```

aufrufen, wobei N und M Integer sind und A,B und C exemplarisch für beliebige Eingabeparameter stehen. N bestimmt dabei die Zugehörigkeit zu einem Block und M die Zugehörigkeit zu einem Thread. Insgesamt wird die Funktion $N \cdot M$ fach ausgeführt. Anstelle von 2 Integern können auch dreidimensionale Vektoren benutzt werden, dann bekommt jeder Aufruf eine eigene BlockIdx.x, BlockIdx.y und BlockIdx.z, ebenso für die ThreadIdx. Diese IDs sind im Aufruf bekannt und werden im allgemeinen genutzt um die richtigen Daten zu allokieren.

Moderne GPUs besitzen bis zu 5120 CUDA-Cores (Titan V, Tesla V100), also Recheneinheiten, die eine noch deutlich größere Anzahl an Threads gleichzeitig bearbeiten können. Zwischen 8 (Tesla-Architektur) und 192 (Kepler-Architektur) CUDA-Cores werden zusammen in einem *Streaming Multiprocessor* (SM) vereint. Jeder SM besitzt seinen eigenen Speicher und einen Anteil an *Shared Memory*, über den Daten mit anderen SMs ausgetauscht werden können. Ein SM kann auf den neueren CUDA-Architekturen bis zu 32 Blöcke bearbeiten und insgesamt bis zu 2048 Threads gleichzeitig verwalten. Ein einzelner Block darf dabei maximal 1024 Threads auf einmal besitzen. Ein ähnliches Modell wie SIMD nutzen auch Grafikkarten, das sogenannte Single instruction, multiple threads (SIMT) Modell. So werden immer Einheiten von 32 Threads auf einmal ausgeführt, Warps genannt [Cor18b].

Wird ein Kernel aufgerufen, wird dieser N mal mit einer individuellen BlockIdx an die SMs aufgeteilt. Auf der SM wird für jeden Block der Kernel M mal als Thread ausgeführt. Um eine GPU optimal zu nutzen, sollte jede SM mit genügend Threads versorgt werden. Eine gute Threadgröße pro Block liegt dabei bei 256 [Cor18a].

Bei der Implementierung wird das Simulationsgebiet durch *P4EST* in Oktalbäume unterteilt, die dann in einzelne Oktanten verfeinert werden, sodass ein (reguläres) Gitter entsteht. Werden diese Oktanten auf die GPU kopiert, lässt sich dort nicht feststellen, wer dessen Nachbarn sind. Ohne diese Information lässt sich kein Strömungsschritt auf der GPU realisieren. Um die benötigte Datenparallelität zu schaffen, wird eine Diskretisierungshierarchie [BCMT13; CB17] eingeführt. Jede vorhandene *P4EST*-Zelle wird nochmals in ein reguläres Gitter mit n Zellen pro Dimension unterteilt, dies ergibt den sogenannten Patch. Um zwischen Patches keine Kommunikation im

Strömungsschritt zu benötigen, wird für diese Patches ein jeweiliger *Halo* (Abbildung 4.1) erstellt, dieser entspricht einem zelllokalen *Ghost-Layer* (Abschnitt 2.2.3 auf Seite 19), und legt eine Schicht mit Breite l/n, also einer Zelle des Patches, um den gesamten Patch herum, wobei l hier die Länge einer *P4EST*-Zelle ist. Der Patch mit seinem Halo entspricht dann der Last, die auf der GPU unabhängig von den anderen gelöst werden kann und wird in einer *Payload*-Datenstruktur gespeichert. Diese besitzt dann $(n + 2)^3$, wobei n die Anzahl pro Dimension ist, erweitert um die Halo-Schicht, Patchzellen. Die Größe n wird auch Patchsize genannt und kann variabel festgelegt werden.

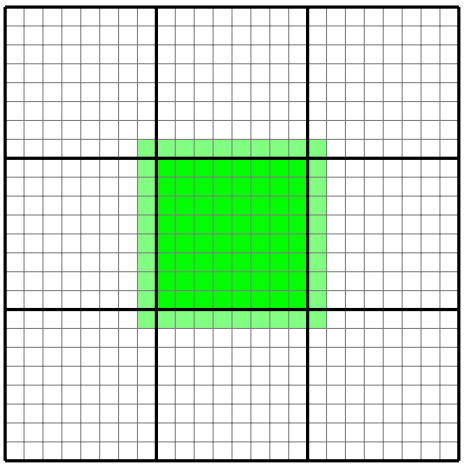


Abbildung 4.1: Patchzellen mit ihrem Halo. Die schwarz umrahmten Gebiete entsprechen dabei jeweils einem *P4EST* Oktanten.

Jeder Payload wird einer BlockID zugewiesen und die einzelnen Zellen in der Payload werden einer ThreadIdx.x, ThreadIdx.y und ThreadIdx.z zugewiesen (Listing 4.1 auf der nächsten Seite). Durch inkrementieren und dekrementieren der einzelnen threadIdxs lassen sich die Nachbarn finden. Für 512 *P4EST*-Zellen und einer Patchsize von 8, ergibt dies $512 \cdot 10^3 = 512000$ Kollisionsaufrufe und $512 \cdot 8^3 = 262144$ Strömungsaufrufe.

Dieses Verfahren hat aber nicht nur Vorteile. So müssen zwischen jedem Integrationsschritt die Daten zwischen der GPU und CPU ausgetauscht werden um die Halos der Patches zu bestimmen, da verschiedene Payloads nicht ihre Nachbarn kennen und nicht einfach über die Blockgrenzen hinaus kommunizieren können. Außerdem werden die Daten der Halos dupliziert, diese Zellen sind dann auf mehreren Payloads vorhanden. Für kleine n bedeutet dies, dass ein Vielfaches der Daten gespeichert und kopiert wird und auch ein Vielfaches an Kollisionen im Kollisionsschritt berechnet werden muss. Daher ist es natürlich wichtig ein möglichst gutes n zu wählen, dies wird gesondert in Abschnitt 4.3.2 auf Seite 40 behandelt.

4.2 Simulationsablauf

Eine Simulation in *ESPResSo* startet, indem das Programm in der Konsole mit einem Simulationsscript gestartet wird

```
1  mpirun -n #jobs ./Espresso ./PathToScript.tcl #parameter,
```

```
#Size of P4EST Octants on used level
dim3 blocks_per_grid(local_num_real_quadrants_level[level]);
#Size of Threads for collision kernel
dim3 threads_per_block(LBADAPT_PATCHSIZE_HALO,
                       LBADAPT_PATCHSIZE_HALO,
                       LBADAPT_PATCHSIZE_HALO);

#call of collision kernel with it's necessary arguments
lbadapt_gpu_collide<<<blocks_per_grid, threads_per_block>>>(
                                            dev_local_real_quadrants[level],
                                            level, h_max, d_lbpar, d_d3q19_modebase,
                                            d_d3q19_w);

#Size of Threads for streaming kernel
dim3 threads_per_block(LBADAPT_PATCHSIZE,
                       LBADAPT_PATCHSIZE,
                       LBADAPT_PATCHSIZE);

#call of bounce-back and streaming kernel with it's necessary arguments
lbadapt_gpu_bounce_back_and_stream<<<blocks_per_grid, threads_per_block>>>(
                                            dev_local_real_quadrants[level],
                                            h_max, d_lb_boundaries, d_lbpar,
                                            d_d3q19_lattice, d_d3q19_w);
```

Listing 4.1: Vereinfachte Form der Kernelaufrufe für den Integrationsschritt

dabei können optional noch Parameter für das Script übergeben werden oder die Ausführung durch MPI spezifiziert werden. Dieses Script gibt dabei wichtige Simulationsgrößen vor. In diesem Abschnitt wird anhand eines einfachen Simulationsscriptes, das eine Hagen-Poiseuille-Simulation ausführt, verständlich gemacht, wie ein allgemeines Script funktioniert und was parallel in *ESPResSo* geschieht.

Ein Script lässt sich grob in 3 Teile aufteilen

1. Ein- und Ausgabe Parameter
2. Initialisierung der Simulation
3. Ausführen der Simulationsschritte

4.2.1 Ein- und Ausgabe Parameter

In diesem Teil werden die durch die Konsole eingegebene Parameter an das Script übergeben und der Ort für die Ausgabe festgelegt (Listing 4.2 auf der nächsten Seite). Die Parameter werden beim Aufruf durch die Kommandozeile einzeln und nur durch ein Leerzeichen getrennt hinter dem Pfad zum Script eingegeben. Dabei ist es möglich Standardwerte vorzugeben, die nur dann überschrieben werden, wenn sie beim Aufruf durch die Konsole angegeben werden. In einem ausführlichen Script lässt sich genau festlegen, welchen Werte einzelnen Eingabeparametern zugewiesen werden je nach Anzahl der optionalen Eingabeparameter. Für die Ausgabe der Simulationsergebnisse muss ein Ordner gewählt werden und ein Schema, wie die einzelnen Ausgabedateien genannt werden sollen. Tcl erlaubt es anhand des Server- und dortigem Nutzernamen ein Verzeichnis auszuwählen. Die

```
# Copyright (C) 2013,2014 The ESPResSo project
#
# This file is part of ESPResSo.

# read refinement level, steps, write_output and patchsize from command line or use default settings
if { $argc == 4 } {
    set max_level [lindex $argv 0]
    set steps [lindex $argv 1]
    set write_output [lindex $argv 2]
    set patchsize [lindex $argv 3]
} else {
    set max_level 4
    set steps 512
    set write_output 1
    set patchsize 16
}
set agrid_max [expr 1./pow(2, $max_level)]


# set output directories
set folder ""
set filename ${folder}poiseuille
```

Listing 4.2: Erster Teil eines für die Implementierung angepassten simplen *poiseuille*-Simulationsscriptes

Ausgaben werden im VTK-Format (Visualization Toolkit) gespeichert. VTK ist eine open-source Bibliothek, die gut geeignet ist für die Visualisierung von dreidimensionalen Simulationsdaten. Die ausgegebene VTK Dateien können dann mit verschiedene Programmen wie Paraview ausgewertet und visualisiert werden.

Wird nun der Code aus Listing 4.2 mit den Parametern 4, 1000, 1 und 16 aufgerufen, so wird das $\max_{level}$ auf 4 gesetzt, 1000 Simulationsschritte ausgeführt, Output geschrieben und die Patchsize auf 16 gesetzt. Dadurch ergibt sich eine Maschenweite

$$h = \left(\frac{Box}{2^{\max_{level}} \cdot Patchsize}\right) = \left(\frac{1}{(2^4 \cdot 16)}\right) = 0.001953125,$$

wobei in dem Beispiel von einer Box, die dem Simulationsgebiet entspricht (Abschnitt 4.2.2), mit Seitenlängen 1 ausgegangen wird. Der Output wird in den Ordner, von dem aus das Programm gestartet wird, angelegt.

4.2.2 Simulationsinitialisierung

Im zweiten Teil wird die Simulation initialisiert. Dabei wird das Simulationsgebiet definiert und das Verhalten der Teilchen spezifiziert (Listing 4.3 auf Seite 37). Als Simulationsgebiet kann eine Menge verschiedener geometrischer Objekte mit verschiedener Größe angegeben werden, wie beispielsweise Quader oder Kugeln. Daraus ergibt sich eine dreidimensionale Boxsize. Auch kann definiert werden, wie sich die Simulation am Rand verhält. Im Beispielscript (Listing 4.3 auf Seite 37) wird in jede Dimension ein periodischer Rand definiert, dadurch wird beispielsweise in

Y-Richtung aus $y_{\max} + 1 = y_o$. Um aus einer adaptiven Gitterstruktur eine reguläre Struktur zu machen, wird das $\min_{level}$ und $\max_{level}$ auf die gleiche Größe gebracht. Danach wird der Zeitschritt relativ zur Molekular-Dynamik gesetzt und die Eigenschaften des Fluids mit Daten wie der Dichte, Viskosität, Scherung und Reibung. Thermische Schwankungen sind für adaptive Gitter noch nicht in *ESPResSo* implementiert und werden daher auf 0 gesetzt. Damit sind die notwendigen Bedingungen gesetzt. Optional können zum Beispiel externe Kräfte auf die Simulation wirken oder Boundaries definiert werden. Für mögliche Boundaries stehen auch wieder verschiedene geometrische Objekte zur Verfügung. Boundaries können Eigenschaften haben, so können sie sich bewegen oder mit einer Geschwindigkeit initialisiert werden und dadurch zu einem Ein- beziehungsweise Ausfluss werden.

Mit diesen Angaben wird das Gebiet in gleich große Gebiete aufgeteilt, und an die durch MPI verwaltete Jobs verteilt. Die Oktalbäume werden dann rekursiv bis auf Stufe $\max_{level}$ und darauf nochmals in die einzelnen Patches unterteilt.

Im nächsten Schritt wird mit einem aus *P4EST* zur Verfügung gestellten Iterator [LBH+16] über alle Payloads iteriert und die Patchzellen auf 0 initialisiert und abgefragt, ob zum Beispiel eine externe Kraft gesetzt ist und wie diese auf die Zellen wirkt. Ebenso wird für jede Patchzelle überprüft, ob sie zu einer Boundary gehört. Ist dies der Fall, wird sie als *dry-node* markiert und verliert alle Populationen.

Am Ende dieser Operationen erhält man den Ausgangszustand für die Integrationsschritte, bei denen dann die LBM ausgeführt wird. Wenn sich an der Konfiguration nichts mehr ändert, ist dieser Initialisierungsprozess pro Simulation nur einmal zu durchlaufen. Im Codebeispiel (Listing 4.3 auf der nächsten Seite) wird dieser Zustand am Ende noch gespeichert und in VTK-Dateien exportiert, falls dies durch den *write_output* Parameter gesetzt worden ist. Dabei wird von jedem Prozess ein file geschrieben, dass dann in Programmen wie Paraview wieder zu einem Ganzen vereint werden kann.

4.2.3 Ausführen der Simulationsschritte

Während die ersten zwei Teile durch *ESPResSo* und *P4EST* zum Großteil schon gegeben waren, wurde die meiste Arbeit der Implementierung in die Integrationsschritte gesteckt. Die Integrationsschritte werden mit dem Befehl *integrate AnzahlSchritte* gestartet. Im Codebeispiel Listing 4.4 auf Seite 38 wird eine Zeitmessung initialisiert, welche die Zeit von *start* bis *stop* misst und damit die ganze Dauer der Integration umfasst. Bei großen Simulationen gibt es mehrere Gründe eine Schleife, wie sie im Codebeispiel verwendet wird, zu nutzen:

- Es lässt sich genau definieren, wann eine Ausgabe gemacht wird. Bei einer hohen Anzahl von Zellen dauert es lange bis sich Störungen oder Kräfte einmal über das komplette Feld ausgebreitet haben, noch länger dauert es bis sich ein Gleichgewicht eingestellt hat, falls eines existiert. Dies liegt daran, dass Populationen pro Zeitschritt nur bis zu ihren Nachbarzellen strömen können. Da die Ausgabe zu einem einzelnen Zeitpunkt die Größe von einem Gigabyte problemlos übersteigen kann und jede Ausgabe auch eine Menge Zeit benötigt und nicht von der GPU beschleunigt wird, sollte überlegt werden, zu welchen Zeitschritten eine Ausgabe wirklich notwendig ist.

```
# Start initializing the simulation

# setup simulation domain
set box_x 2
set box_y 1
set box_z 1
setmd box_l $box_x $box_y $box_z
setmd periodicity 1 1 1

lbadapt_set_min_level $max_level
lbadapt_set_max_level $max_level
# set simulation length
set dt 0.01
setmd time_step $dt
#Variable must be set to prevent error, but isn't used
setmd skin 0.000003
# set the fluid
lbfluid agrid $agrid_max dens 0.008 visc 0.032 tau [expr 1*$dt] friction 0.5
# disable fluctuations
thermostat lb 0.
# setup a channel
lbboundary wall dist [expr $agrid_max / $patchsize] normal 0 0 1
lbboundary wall dist [expr ($agrid_max /$patchsize) - $box_z] normal 0 0 -1
# set inflow/outflow boundaries
lbboundary wall dist [expr $agrid_max / $patchsize] normal 1 0 0 velocity 0.1 0 0
lbboundary wall dist [expr ($agrid_max / $patchsize) - $box_x] normal -1 0 0 velocity 0.1 0 0

if {[expr $write_output == 1]} {
    lbfluid print vtk boundary ${filename}_boundary.vtk
}
```

Listing 4.3: Zweiter Teil eines für die Implementierung angepassten simplen *poiseuille*-Simulationsscriptes

- Um besser abschätzen zu können, wie lange die Simulation noch dauert, kann eine Ausgabe nach einer gewissen Anzahl von Iterationen hilfreich sein.

Mittels *put*-Befehl kann eine Ausgabe auf der Konsole erzeugt werden. Im Beispiel wird am Ende eine letzte Ausgabe geschrieben und die benötigte Zeit ausgegeben.

Während eines Simulationsschrittes können je nach aktivierter Module eine ganze Reihe von Simulationen vorgenommen werden. Es wird davon ausgegangen, dass nur das hier vorgestellte LBM Modul aktiv ist.

Bei jedem Integrationsvorgang laufen dann folgende Schritte ab:

1. Die Halos der Payloads werden auf der CPU aktualisiert. Hier wird wieder auf jedem Prozessor mit dem *P4EST* Iterator über alle Payloads iteriert. Der Iterator bestimmt den benachbarten Payload, von diesem werden dann die passenden Patchzellen in den Halo der Ursprung Payload geschrieben. In Abschnitt 4.3.3 auf Seite 41 wird auf die Funktionsweise genauer eingegangen.
2. Im nächsten Schritt werden die Payloads auf die GPU kopiert.

```
1  ############################################################################
2  # Perform integration steps                                                #
3  ############################################################################
4  set itermax $steps
5  set output_step 100
6  set start_integration [clock microseconds]
7
8  for {set i 0} {$i < $itermax} {incr i} {
9      puts "Performing integration step $i of $itermax"
10     integrate 1
11     if {1 == $write_output &&
12         0 == $i % $output_step } {
13         lbfluid print vtk velocity ${filename}_vel_${i}.vtk
14     }
15 }
16 integrate [expr int($itermax)]
17
18 set stop_integration [clock microseconds]
19
20 if {[expr $write_output == 1]} {
21     lbfluid print vtk velocity ${filename}_velocity_${itermax}.vtk
22 }
23
24 puts "simulation run time: [expr {$stop_integration - $start_integration}]"
25 puts "for $steps iteration steps."
```

Listing 4.4: Dritter Teil eines für die Implementierung angepassten simplen *poiseuille*-Simulationsscriptes

3. Nun wird auf der GPU die rechenintensive Kollision, wie in Kapitel 3 auf Seite 23 behandelt, ausgewertet. Dafür werden zuerst die Erhaltungsgrößen ρ,$\rho\boldsymbol{u}$ und Π berechnet. Daraus lässt sich das lokale Gleichgewicht berechnen, zu dem sich, abhängig von der Viskosität, die einzelnen Populationen hinbewegen. Auf diese lässt man nun mögliche externe Kräfte wirken.
4. Auf die Kollision folgt der Strömungsschritt. Hier iteriert jede Zelle über seine Nachbarn und prüft, ob diese als Boundary markiert ist. Ist dies der Fall, wird die Population mit umgekehrten Geschwindigkeitsvektor in derselben Zelle gespeichert (bounce-back), ansonsten wird von der Nachbarzelle die zur Zelle zeigende Population übernommen.
5. Die Daten werden nun zurück von der GPU auf die CPU kopiert.
6. Die einzelnen Prozessoren tauschen ihren *Ghost-Layer* aus, dies geschieht wieder über *P4EST* und MPI.

Die Ergebnisse der Simulation lassen sich dann mit geeigneten Programmen auslesen.

4.3 Ausgewählte Themen und Probleme

4.3.1 Speicheraustausch

Ein generelles Problem bei dieser Implementierung ist, dass in jedem Schritt Daten zwischen der CPU und GPU ausgetauscht werden müssen um die Halos zu beschreiben und evtl. eine Ausgabe zu generieren. Einmal dauert jeder Kopiervorgang wertvolle Zeit und nimmt dabei einen großen Anteil der Gesamtdauer eines Integrationsschrittes ein. So konnte die Zeit einer Integration um ungefähr 20% beschleunigt werden, indem die Größe der Payloads um circa 33% verringert wurde. Für den Kollisionsschritt werden zusätzlich zu den Populationen verschiedene Momente berechnet, diese werden im Strömungsschritt nicht mehr benötigt. Anderseits werden im Strömungsschritt für die parallele Bearbeitung ein weiterer Satz an Populationen benötigt, die den Zustand nach der Strömung wiedergeben. In der Payload-Datenstruktur wurden beide Werte separat gespeichert, inzwischen wird ein Array für beide Daten benutzt. Anderseits hat die CPU nichts zu berechnen, während auf der Grafikkarte ein Integrationsschritt ausgeführt wird, und umgekehrt ist die GPU im Leerlauf, während die Halos für die nächste Integration befüllt werden (Abbildung 4.2).

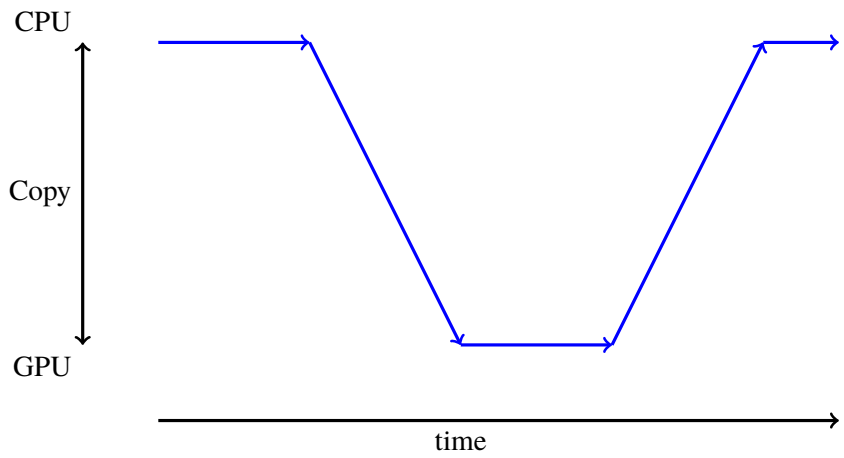


Abbildung 4.2: Verlauf der Daten einer Simulation: Es ist immer nur der Teil aktiv, auf dem sich die Daten (blaue Linie) befinden

Der Speicher ist allgemein ein wichtiges Thema, weil die auf der Grafikkarte verfügbare Menge im allgemeinen deutlich geringer ausfällt als auf der CPU. Bei Serversystemen mit Beschleunigerkarten, derzeit hauptsächlich GPUs, werden häufig Systeme genutzt, die 1-2 CPUs benutzen und mit bis zu 4GPUs arbeiten. Prominente Beispiele sind der Piz Daint [20116] und der TSUBAME 3.0 [Ser17]. Ersterer belegt aktuell den 3. Platz in der top500 [TOP18b] und erreicht den Großteil seiner Rechenleistung aus 5320 XC50 Compute Nodes mit jeweils 12 CPU-Kernen, 64GB Ram und einer NVIDIA Tesla P100 mit 16 GB Speicher. Damit ist er der größte Supercomputer mit dieser Architektur. Zweiterer belegt aktuell den 6. Platz in den green top500 [TOP18a] mit 540 SGI ICE XA Nodes mit jeweils 28 CPU-Kernen, 256GB Ram und 4 NVIDIA Tesla P100 mit jeweils 16 GB Speicher. Der TSUBAME 3.0 ist der effizienteste Supercomputer mit dieser Architektur, der gleichzeitig auch in den top500 sehr gut abschneidet. Die CPUs haben hier jeweils das 4-fache an Speicher verglichen mit den GPUs.

Patchsize	Anzahl Zellen ohne Halo	Anzahl Zellen mit Halo	Redundanz
1	1	27	27
2	8	64	8
4	64	216	3.375
8	512	1000	1.953
16	4096	5832	1.424
32	32768	39304	1.199

Tabelle 4.1: Daten- und Kollisionsdopplung in Abhängigkeit der Patchsize.

Theoretisch sind für CPU Nodes Konfigurationen mit über einem Terabyte Hauptspeicher möglich, bei GPUs sieht die Situation anders aus. Dieses Jahr (2018) wurden einzelne GPUs mit bis zu 32 Gigabyte Speicher veröffentlicht, die meistens Modelle besitzen aber deutlich weniger. Dadurch kann es vorkommen, dass im Hauptspeicher noch eine Menge Platz ist, aber der Grafikkartenspeicher ausgereizt ist. Da eine einzelne Server-CPUs heutzutage auch bis zu 32 Kernen haben, kommt hier ein weiteres Ungleichgewicht zustande. Wird eine Simulation über MPI gestartet, so sucht sich jeder Prozess eine möglichst freie GPU. Da in der Regel deutlich weniger GPUs vorhanden sind als CPU-Kerne, kann eine Grafikkarte für mehrere CPU-Prozesse gleichzeitig als Beschleunigerkarte dienen. Dies verringert die Dauer der Initialisierung des Gitters und die Dauer der Beschreibung der Halos. Zusätzlich wird eine gewisse Nebenläufigkeit erreicht, weil, während ein Prozess mit dem Kopiervorgang abgeschlossen hat, die Grafikkarte anfangen kann diese Payloads zu berechnen, während weitere noch kopiert werden. Dadurch wird die Simulation auch merkbar durch zusätzliche CPU-Prozesse beschleunigt, limitiert wird das Ganze nur durch den Speicher der GPUs und mit der Bandbreite, mit der sie an die CPUs angeschlossen sind.

4.3.2 Patchsize

Wie in Abschnitt 4.1 auf Seite 31 erwähnt, spielt n, die Anzahl der Unterteilungen des Patches pro Dimension, auch Patchsize genannt, eine entscheidende Rolle. Wird im Extremfall die Patchsize auf 1 gesetzt, so entspricht eine Patchzelle einer *P4EST*-Zelle, nur wird in jeder Payload die Nachbarn der Patchzelle ebenfalls gespeichert. So enthält eine Payload später 27 Zellen, für die auch alle eine Kollision berechnet wird. Der Speicherbedarf wächst dabei auch um das 27-fache an.

Das Ziel ist es folglich n möglichst groß zu wählen und gleichzeitig einen für die Berechnungen auf der GPU *guten* Wert zu finden. Theoretisch ist für n eine beliebige natürliche Zahl denkbar, es wurde sich aber auf Potenzen von 2 konzentriert. Um auch bei kleineren Simulationen eine sinnvolle hierarchische Unterteilung zu erhalten und trotzdem relativ wenig Daten duplizieren zu müssen, wurde sich für eine standardmäßige Patchsize von 16 entschieden. Jeder Payload wird auf der GPU einem Block zugewiesen. In jedem Block haben alle Threads auf den gemeinsamen Speicher Zugriff und jeder Block kann bis zu 1024 Threads gleichzeitig bearbeiten. Mit einer Patchsize von 16 gibt es insgesamt 5832 Patchzellen (Tabelle 4.1), diese können durch das Threadlimit nicht gleichzeitig bearbeiten werden. Dadurch wird für die Bearbeitung der Patch nochmals in 8 weitere Schritte unterteilt, die dann nacheinander aufgerufen werden. Vor dem Aufruf des Ströumgs- und Kollisionskernels wird dafür ein Offset berechnet. Nach dem Aufruf des Kernels wird über die Block-Id der Pointer auf die richtige Payload gesetzt. Die Anzahl der Blöcke im Aufruf entspricht

also der Anzahl von Payloads, die auf dem Prozess abhängig vom Level laufen. Die einzelne Patchzelle wird über die Thread-IDs, die hier dreidimensional sind, und den Offset bestimmt. Pro Aufruf gibt es für eine Patchsize von 16 im Kollisionsschritt $\left(\frac{16+2}{2}\right)^3 = 729$ Threads und im Strömungsschritt entsprechend $\left(\frac{16}{2}\right)^3 = 512$ Threads.

4.3.3 Populate Halos

Auf der CPU werden während den Integrationsschritten, wenn nach diesem Schritt keine Ausgabe geschrieben wird, nur die Halos befüllt. Einerseits geschieht dies seriell, aber durch den *P4EST*-Iterator relativ schnell. Dabei wird über alle Payloads iteriert. Durch das D3Q19 Gitter sind jeweils die Nachbarn mit angrenzenden Flächen und Kanten interessant, Nachbarn mit Ecken sind hingegen egal. Über diese 19 Nachbarn wird iteriert. Dabei grenzen die ersten 6 Nachbarn immer an die Flächen f_0 bis f_5 an und die Nachbarn 7-19 Grenzen an die Kanten e_0 bis e_{11} (Abbildung 2.3 auf Seite 19). Da die Halos sich immer auf einem Oktalbaum und seinem *Ghost-Layer* befinden, haben die Nachbarn auch immer den gleichen Ankerpunkt. Für jeden der 18 Nachbarn lässt sich daher genau sagen, welche Patchzellen benötigt werden.

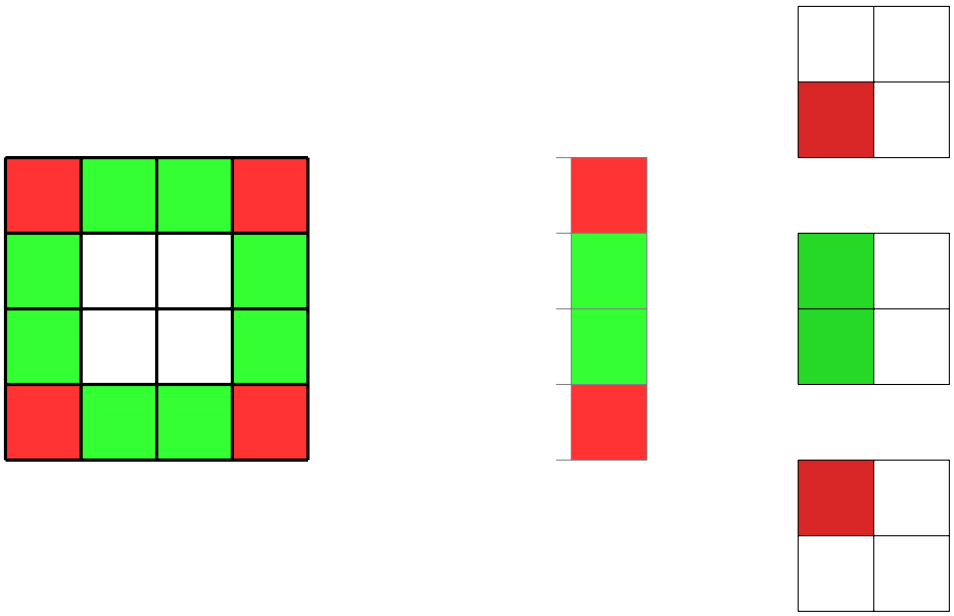


Abbildung 4.3: zweidimensionaler D2Q9 Halo wird durch seine Nachbarzellen bevölkert.

Durch den Halo haben für eine Patchsize von 16 die Zellen in jeder Dimension einen Wert zwischen 0 und 17, die *inneren* Daten von 1 bis 16. Beim Schreiben des Halos wird immer aus dem *inneren* einer anderen Zelle in den Halo der behandelten Zelle geschrieben (Abbildung 4.3). Dafür wird ein write_offset, ein read_offset und ein iter_max für die Dimensionen X, Y und Z angelegt. Für den Fall f_0 sind dies alle Zellen mit X Koordinate 16, während Y und Z von 1 bis 16 durchlaufen. Diese werden in den Halo mit X Koordinate 0, während Y und Z wieder von 1 bis 16 durchlaufen, geschrieben (Listing 4.5 auf der nächsten Seite).

```
if (0 <= dir_p4est && dir_p4est < P8EST_FACES) {
                                // for faces:
                                // The face is orthogonal to the direction it is associated with.
                                // That means for populating the halo of the patch we have to
                                // iterate over the other two indices, keeping the original
                                // direction constant.
                                iter_max_x = iter_max_y = iter_max_z = LBADAPT_PATCHSIZE;
                                r_offset_x = r_offset_y = r_offset_z = 1;
                                w_offset_x = w_offset_y = w_offset_z = 1;
                                if (4 == (dir_p4est & 4)) {
                                  iter_max_z = 1;
                                  r_offset_z = (dir_p4est % 2 == 0 ? LBADAPT_PATCHSIZE : 1);
                                  w_offset_z = (dir_p4est % 2 == 0 ? 0 : LBADAPT_PATCHSIZE + 1);
                                } else if (2 == (dir_p4est & 2)) {
                                  iter_max_y = 1;
                                  r_offset_y = (dir_p4est % 2 == 0 ? LBADAPT_PATCHSIZE : 1);
                                  w_offset_y = (dir_p4est % 2 == 0 ? 0 : LBADAPT_PATCHSIZE + 1);
                                } else {
                                  iter_max_x = 1;
                                  r_offset_x = (dir_p4est % 2 == 0 ? LBADAPT_PATCHSIZE : 1);
                                  w_offset_x = (dir_p4est % 2 == 0 ? 0 : LBADAPT_PATCHSIZE + 1);
                                }
```

Listing 4.5: Bestimmung der Offsets für die 6 Nachbarflächen

```
// perform the actual data replication
        for (int patch_z = 0; patch_z < iter_max_z; ++patch_z) {
                for (int patch_y = 0; patch_y < iter_max_y; ++patch_y) {
                        for (int patch_x = 0; patch_x < iter_max_x; ++patch_x) {
                                memcpy(&data->patch[w_offset_x + patch_x]
                                                   [w_offset_y + patch_y]
                                                   [w_offset_z + patch_z],
                                                   &neighbor_data->patch
                                                   [r_offset_x + patch_x]
                                                   [r_offset_y + patch_y]
                                                   [r_offset_z + patch_z],
                                                   sizeof(lbadapt_patch_cell_t));
                        }
                }
        }
```

Listing 4.6: Die eigentliche Datenreplikation, bei der die Halos beschrieben werden

Bei den Kanten funktioniert das analog. Hier wird zwischen den 12 Kanten unterschieden und dann nur über eine Dimension iteriert.

Beim eigentlichen Kopiervorgang wird dann so oft wie berechnet über die Positionen iteriert und die Zellen an die passende Stelle kopiert (Listing 4.6).

5 Analyse, Anwendung und Auswertung

5.1 Validierung

Durch die Diskretisierung, Randbedingungen oder Rundungen kann die Simulation die Realität nie ganz genau abbilden. Diese kleine Fehler können sich aufsummieren und Folgefehler produzieren, welche die Simulation stark von der Realität abweichen lässt. Daher ist es wichtig Tests durchzuführen um festzustellen, ob sich die Simulation mit Experimenten oder analytischen Lösungen deckt beziehungsweise um wie viel sie davon abweicht. Dafür werden drei gut bekannte Szenarien betrachtet.

5.1.1 Couette-Strömung

Bei der Couette-Strömung wird ein Fluid zwischen zwei endlosen und zueinander parallelen Platten beobachtet. Die Platten haben den Abstand h zueinander. Dabei ist eine Platte, sei es die untere, stationär und die andere Platte, sei es die obere, bewegt sich mit konstanter Geschwindigkeit $\boldsymbol{V}$ in eine Richtung, sei dies x, so dass der Abstand zwischen den Platten konstant bleibt. Das Fluid wird durch die sich bewegende obere Platte *mitgerissen*. Dabei werden zuerst nur die oberen Schichten bewegt und langsam breitet sich die Kraft über die komplette Höhe h aus und nähert sich einem theoretischen Gleichgewichtszustand an. Das Gleichgewicht lässt sich, wenn der Druck vernachlässigt wird, einfach mit den *Navier-Stokes*-Gleichungen lösen [Ach90]:

$$\frac{d^2\boldsymbol{v}}{dy^2} \tag{5.1}$$

Dabei sei y der Einheitsvektor der y-Dimension und $\boldsymbol{v}(y)$ die Geschwindigkeitsverteilung abhängig von der Höhe y. Dann ist

$$\boldsymbol{v}(y) = \boldsymbol{V}\frac{y}{h}, \tag{5.2}$$

Trägt man die Geschwindigkeit abhängig von der Höhe auf, wird eine Gerade erhalten mit Geschwindigkeit 0 an der unteren Platte und Geschwindigkeit $\boldsymbol{V}$ an der oberen Platte. Bei der Annäherung des Gleichgewichts durchläuft der Aufbau ein bestimmtes Muster, erst werden nur die oberen Schichten stark beschleunigt, während die unteren Schichten beinahe ruhen, langsam verteilt sich die Kraft und das Geschwindigkeitsprofil nähert sich der Geraden an. Nach ungefähr $t \approx \frac{h^2}{\boldsymbol{V}}$ Schritten wird eine gute Näherung der Gerade erhalten. Diesen typischen zeitlichen Verlauf lässt sich in der Simulation beobachten (Abbildung 5.1 auf der nächsten Seite, Abbildung 5.2 auf Seite 45 und Abbildung 5.3 auf Seite 46). Werden die Simulationen mit der schon vorhandenen adaptiven Implementierung verglichen, indem in beiden Simulationen durch die Eingabeparameter

das gleiche Gitter erzeugt wird, sind die Ergebnisse bis auf einen minimalen Fehler identisch. Dieser entsteht, da die Implementierung auf der GPUs nur mit einer einfachen Genauigkeit (32 Bit float) rechnet, die Ausgabe ist aber in doppelter Genauigkeit (64 Bit double).

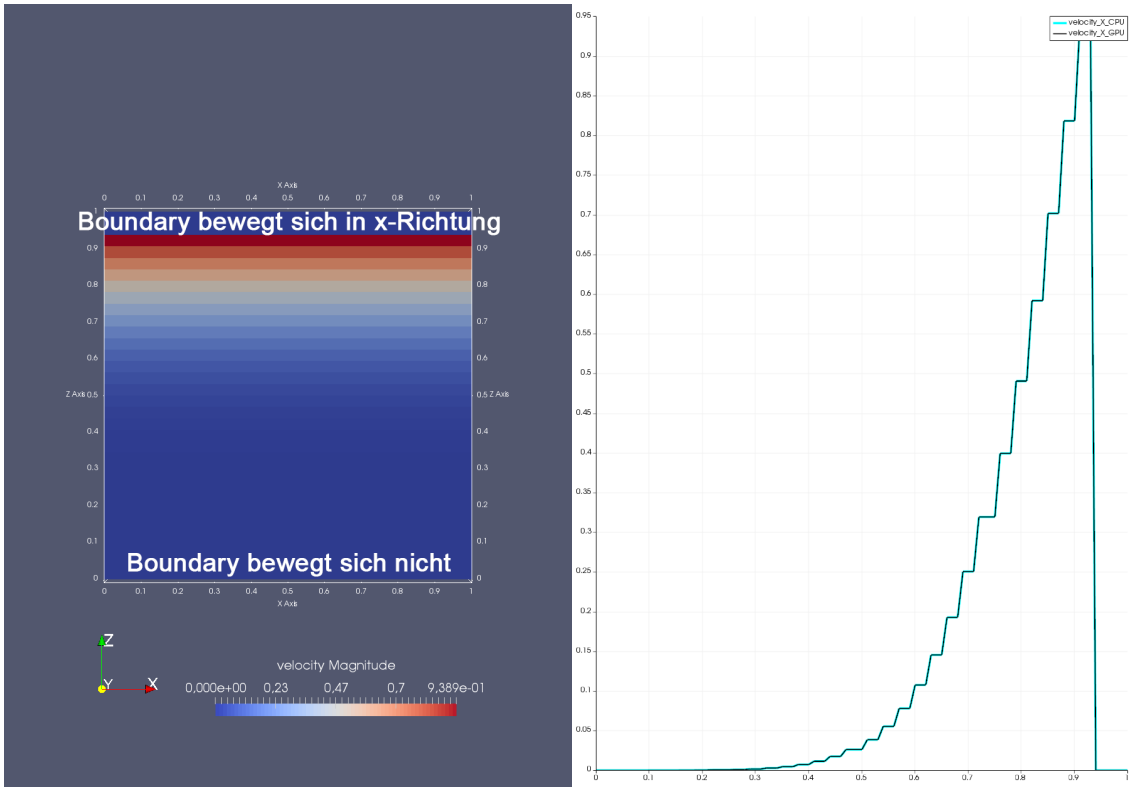


Abbildung 5.1: Eine Couette-Simulation nach 128 Schritten mit 32 Zellen pro Dimension und $V = 1$. Links ist eine Ansicht zwischen den Platten, rechts das Geschwindigkeitsprofil der CPU- und GPU-Implementierung abhängig von der Höhe der Z-Koordinate. Diese sind identisch. Nach 128 Schritten ist die Simulation noch weit von einer Geraden entfernt.

5.1.2 Poiseuille-Strömung

Die Poiseuille-Strömung, auch unter Hagen-Poiseuille-Gesetz bekannt, beschreibt wie ein newtonsches Fluid durch ein Kanal fließt [Kir10]. Dabei entsteht eine typische Geschwindigkeitsverteilung durch den Mittelpunkt, der einer Parabel entspricht und von dem Aufbau der Röhre und der Viskosität des Fluids abhängt. Diese Simulation wird auf zwei unterschiedliche Arten durchgeführt. Dabei wird die Simulation durch eine periodisches Gebiet und keine Boundaries in Y-Richtung auf 2 Dimensionen *runter gebrochen*. Einmal wird das Fluid in einem quadratischen Rohr einer externen Kraft ausgesetzt, die das Fluid durch die Röhre bewegt (Abbildung 5.4 auf Seite 47).

In der anderen Simulation werden in einem Rohr Ein- und Ausflussgrenzen definiert, welche das Fluid in Bewegung setzen (Abbildung 5.5 auf Seite 48). In beiden Fällen wird eine typische parabelförmige Geschwindigkeitsverteilung erreicht. Die Ausgabe deckt sich wieder mit der CPU-Implementierung.

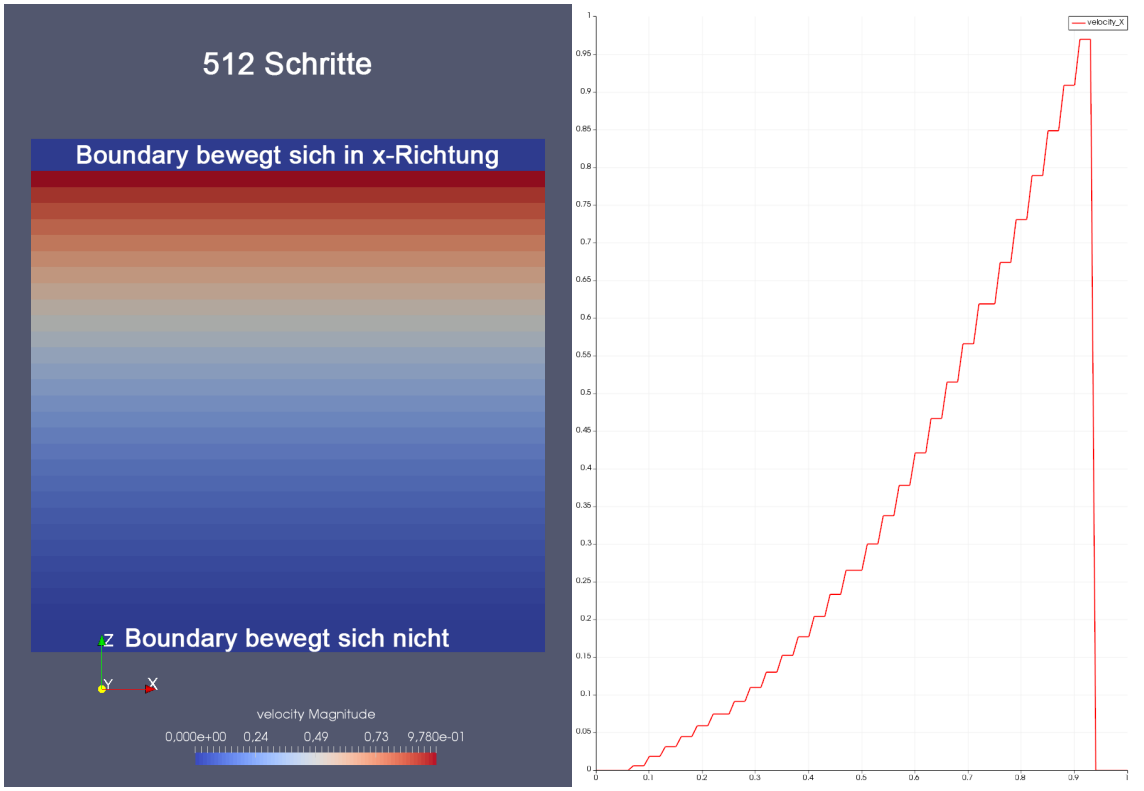


Abbildung 5.2: Eine Couette-Simulation nach 256 Schritten mit 32 Zellen pro Dimension und $V = 1$. Links ist eine Ansicht zwischen den Platten, rechts das Geschwindigkeitsprofil abhängig von der Höhe der Z-Koordinate. Das Profil nähert sich langsam einer Geraden an.

5.1.3 Driven-Cavity

Bei der Driven-Cavity-Simulation wird ein ein Quader untersucht, in dem sich Flüssigkeit befindet. Wie bei der Couette-Strömung bewegt sich eine Wand und erzeugt dadurch eine Strömung, nur diesmal wird die Strömung durch *Seitenwände* begrenzt. Um den dreidimensionalen Fall auf zwei Dimensionen zu reduzieren, wird die Y-Dimension periodisch und besitzt dort keine Grenzen. Dadurch wirken keine Kräfte in diese Dimension und die Geschwindigkeiten in Y-Richtung betragen alle 0. Es wird eine typische Driven-Cavity Strömung erhalten [BS06], wie in Abbildung 5.6 auf Seite 49 und Abbildung 5.7 auf Seite 50 zu erkennen ist. Auch hier decken sich die Ergebnisse mit der CPU-Implementierung.

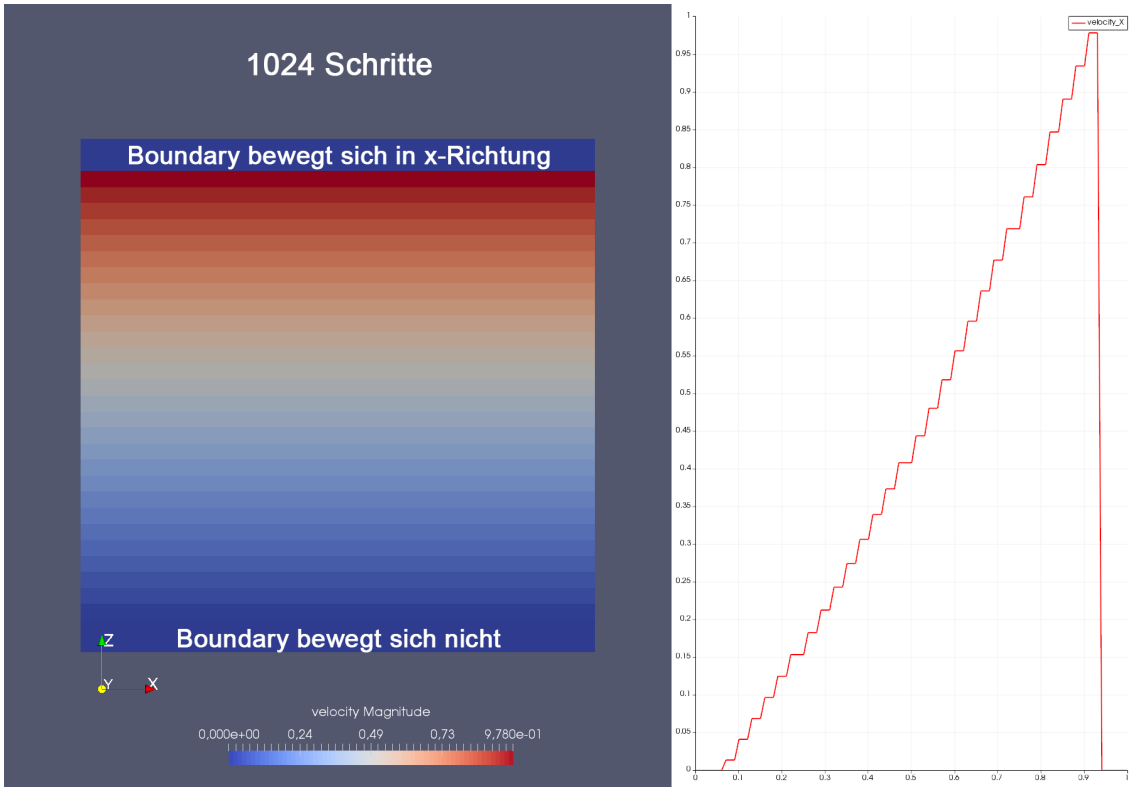


Abbildung 5.3: Eine Couette-Simulation nach 1024 Schritten mit 32 Zellen pro Dimension und $\boldsymbol{V} = 1$. Links ist eine Ansicht zwischen den Platten, rechts das Geschwindigkeitsprofil abhängig von der Höhe der Z-Koordinate. Das Geschwindigkeitsprofil ergibt fast eine Gerade.

5.2 Geschwindigkeit und Skalierung

5.2.1 Die Wahl der Patchsize

In Kapitel 4.3.2 wurde die Bedeutung der Patchsize bezüglich der Datenredundanz erläutert. In diesem Teil wird überprüft, ob die theoretischen Überlegungen sich mit den realen Werten bestätigen lassen. Dafür wird ein Test durchgeführt. Als System wird ein PC mit einer Intel i7-6700k (4 Kerne) CPU, 32 Gigabyte DDR4 Arbeitsspeicher, einer Samsung 960 EVO M.2 SSD und einer Nvidia GTX 980 ti Grafikkarte benutzt. Dabei wird für verschiedene Patchsizes eine Poiseuille-Simulation mit insgesamt $8^6 = 262144$ Patch Zellen und jeweils 20 Schritten durchgeführt. Um die Ergebnisse besser miteinander vergleichen zu können und ohne Einschränkungen bei der Gebietsaufteilung berechnen zu können, werden nur Patchsizes der Größe 1 und Zweierpotenzen bis 16 getestet. Für Patchsize 1 entspricht dies 8^6 *P4EST*-Zellen und folglich ein reguläres Gitter auf Level 6, für die doppelte Patchsize wird jeweils das *P4EST*-Level um eine Stufe reduziert. Dabei zeigt sich, dass die Geschwindigkeitsunterschiede groß ausfallen (Abbildung 5.8 auf Seite 51) und die Simulation mit Patchsize 16 über 32 mal beschleunigt wird (Abbildung 5.9 auf Seite 52), für die Beschleunigung wird auch das englische Wort *Speedup* benutzt.

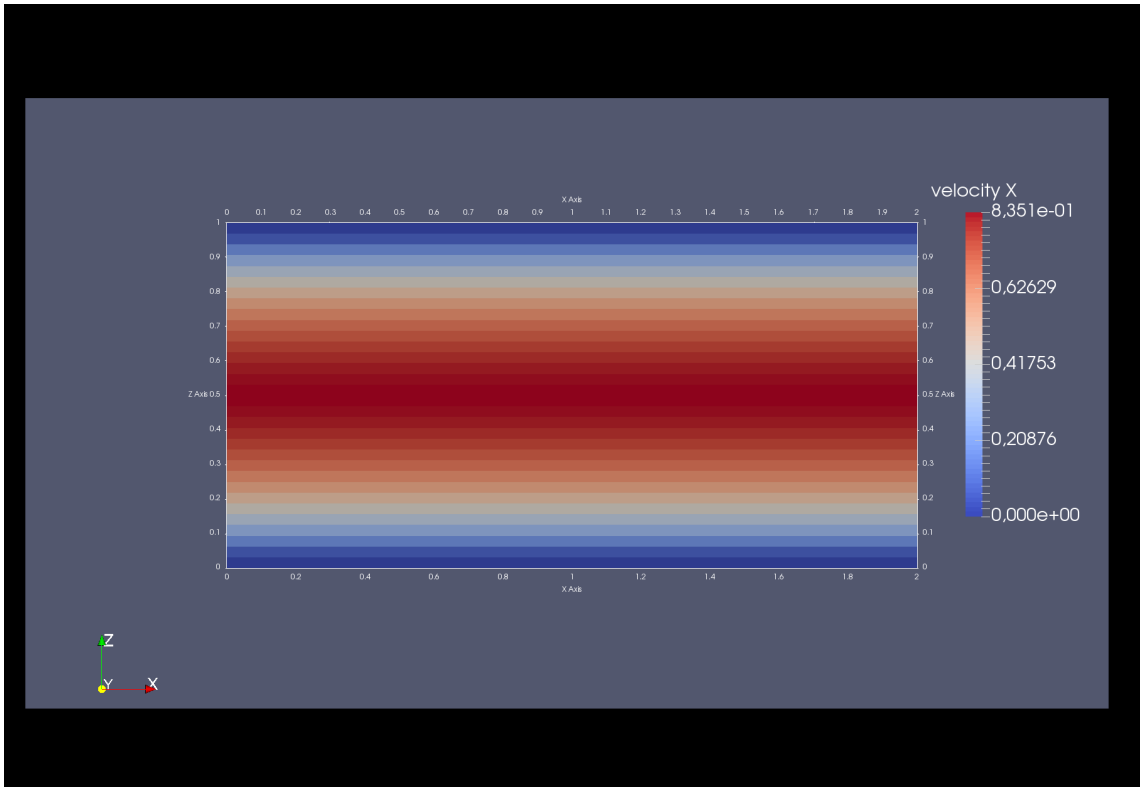


Abbildung 5.4: Eine Poiseuille-Strömung-Simulation mit einer externen Kraft in X-Richtung nach 128 Schritten. Die typische Poiseuille-Geschwindigkeitsverteilung wird erhalten.

Durch dieses Ergebnis wird in allen weiteren Simulationen daher von einer Patchsize von 16 ausgegangen. Nicht berücksichtigt ist in diesem Vergleich, dass durch die geringere Datenredundanz zusätzlicher Speicher zur Verfügung steht, der es ermöglichen kann, dass ein weiterer MPI Job der Grafikkarte zugewiesen werden kann ohne in das Speicherlimit zu laufen.

5.2.2 Skalierungen mit einer Grafikkarte

Bevor Tests mit mehreren Grafikkarten durchgeführt werden, wird zuerst ein Test mit einer Grafikkarten und mehren MPI-Prozessen untersucht. Dafür wird wieder das System mit einer Intel I7-6700k (4Kerne) CPU, 32 Gigabyte DDR4 Arbeitsspeicher, einer Samsung 960 EVO M.2 SSD und einer Nvidia GTX 980ti Grafikkarte benutzt. Zuerst wurde die Dauer einer Ausgabe aller Daten gemessen. Die Ausgabe der vtk-Dateien wird komplett auf der CPU berechnet. Durch das hinzufügen weiterer Prozesse wird die Last unter diesen aufgeteilt und es sollte relativ gut mit der Anzahl der Prozesse skalieren. Im durchgeführten Test konnte bei 4 benutzen Kernen ein Speedup von ≈ 3.6 erreicht werden (Abbildung 5.10 auf Seite 53). Auch durch 8 Jobs mit der Ausnutzung von Hyperthreading konnte gegenüber 4 Jobs ein spürbarer Vorteil erreicht werden. Das Schreiben der Ausgabe kann gerade mit wenig Prozessen pro GPU ein Vielfaches eines Integrationsschrittes benötigen. Für einen Job und eines GPU konnten in dem Versuchsaufbau ≈ 84 Integrationsschritte in der Zeit für eine Ausgabe berechnet werden.

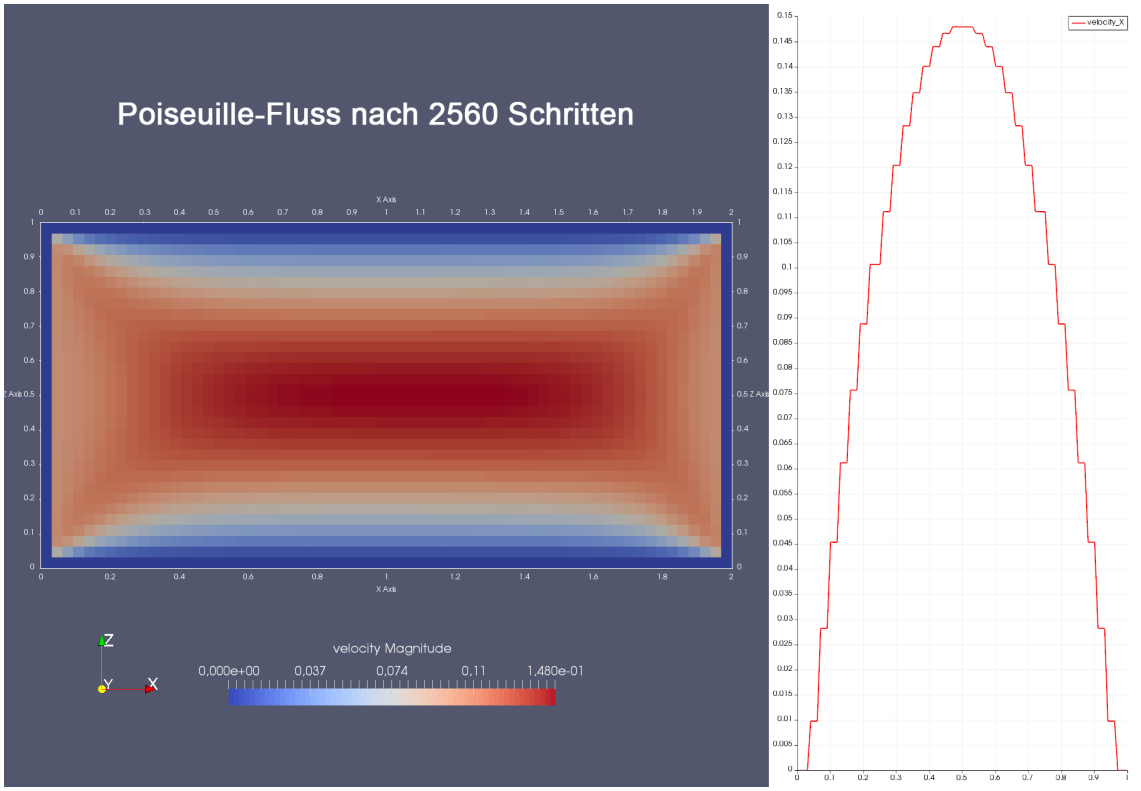


Abbildung 5.5: Eine Poiseuille-Strömung-Simulation mit Zu- und Abflüssen am linken beziehungsweise rechtem Rand nach 2560 Schritten. Das Fluid fließt in x-Richtung vom Zufluss in den Ausfluss. In der Mitte der Simulation bildet sich eine parabelförmige Geschwindigkeitsverteilung, die auf der rechten Seite abgebildet ist. Diese Simulation braucht eine gewisse Zeit, bis die Geschwindigkeiten einigermaßen stabil sind.

Bei den Integrationsschritten wird nur der populate_halos Algorithmus auf der CPU ausgeführt. Gleichzeitig erlaubt das Nutzen mehrerer Prozesse eine gewisse Asynchronität beim Kopieren der Daten zwischen den Geräten. Bei 4 Prozessen konnte im Versuch der beste Wert ermittelt werden mit einem SpeedUp von etwas über 30% (Abbildung 5.11 auf Seite 53).

5.2.3 Schwache Skalierung

Bei der schwachen Skalierung wird untersucht, wie sich die Laufzeit verhält, wenn die Problemgröße proportional zu der verwendeten Hardware wächst. Bei einer Verdopplung der Zellen werden folglich die doppelte Anzahl an CPUs und GPUs benutzt. Das Ziel ist es eine möglichst konstante Laufzeit zu erhalten. Um möglichst gut vergleichbare Ergebnisse zu erhalten, werden nur Zweierpotenzen an Jobs untersucht. Dabei wird als Simulationsgebiet mit einem Würfel angefangen, der bei jeder Verdopplung an Jobs sich in eine Dimension verdoppelt. Nach 3 solchen Schritten oder bei der achtfachen Anzahl an Jobs wird dann wieder ein Würfel erhalten mit den doppelten Seitenlängen und den achtfachen an Zellen. Als Grundlage für diesen Test wird ein System mit 2 Xeon CPUs

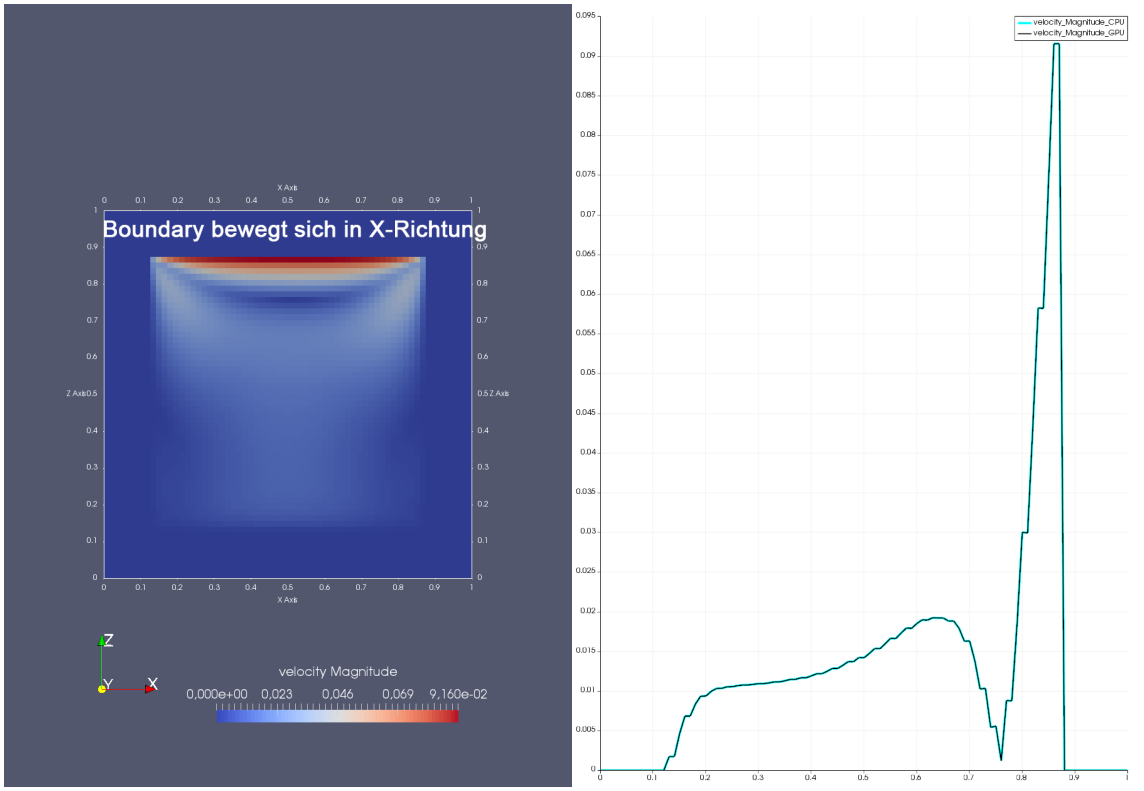


Abbildung 5.6: Links eine Driven-Cavity Simulation nach 256 Schritten mit ihren absoluten Geschwindigkeiten. Rechts sind die Geschwindigkeiten entlang der Mitte der Z-Achse aufgetragen und werden verglichen mit der CPU-Implementierung. Dabei sind keine Abweichungen erkennbar

mit insgesamt 40 Kernen, 786 Gigabyte Arbeitsspeicher und 8 Nvidia GTX 1080 ti Grafikkarten genutzt. Dadurch sind leider nur aussagekräftige Erkenntnisse zu gewinnen für eine Anzahl von bis zu acht Jobs.

Die Effizienz nimmt relativ stark ab (Abbildung 5.12 auf Seite 54). Obwohl der Aufbau ähnlich ist wie die CPU Variante, fällt die schwache Skalierung nicht so gut aus. Wie in Abschnitt 4.3.1 auf Seite 39 erwähnt, konnte die Geschwindigkeit des Algorithmus um 25% gesteigert werden, indem die Größe der Kopiervorgänge um 33% verringert wurde. Die Geschwindigkeit, mit der kopiert wird, ist folglich ein wichtiger Faktor bei der Gesamtperformance. Die meisten GPUs, auch die hier verwendete GTX 1080 ti, können derzeit mit 16 PCI Express 3.0 lanes mit der CPU verbunden werden. Dies erlaubt eine theoretische Datenübertragungsrate zwischen der CPU und der GPU von 15,754 Gigabyte pro Sekunde. Eine einzelne Xeon CPU hat aber nur 48 PCI Express 3.0 lanes, weshalb nicht alle 4 Grafikkarten mit den vollen 16 lanes verbunden werden können. Die genaue Konfiguration ist dabei unbekannt. Es wäre daher denkbar, dass ein Teil des Einbruchs der Effizienz mit der Limitierung der Speicherbandbreite zu tun hat. Um diesen Zusammenhang aber mit Sicherheit bestätigen zu können, wären weitere Tests auf unterschiedlichen Systemen notwendig.

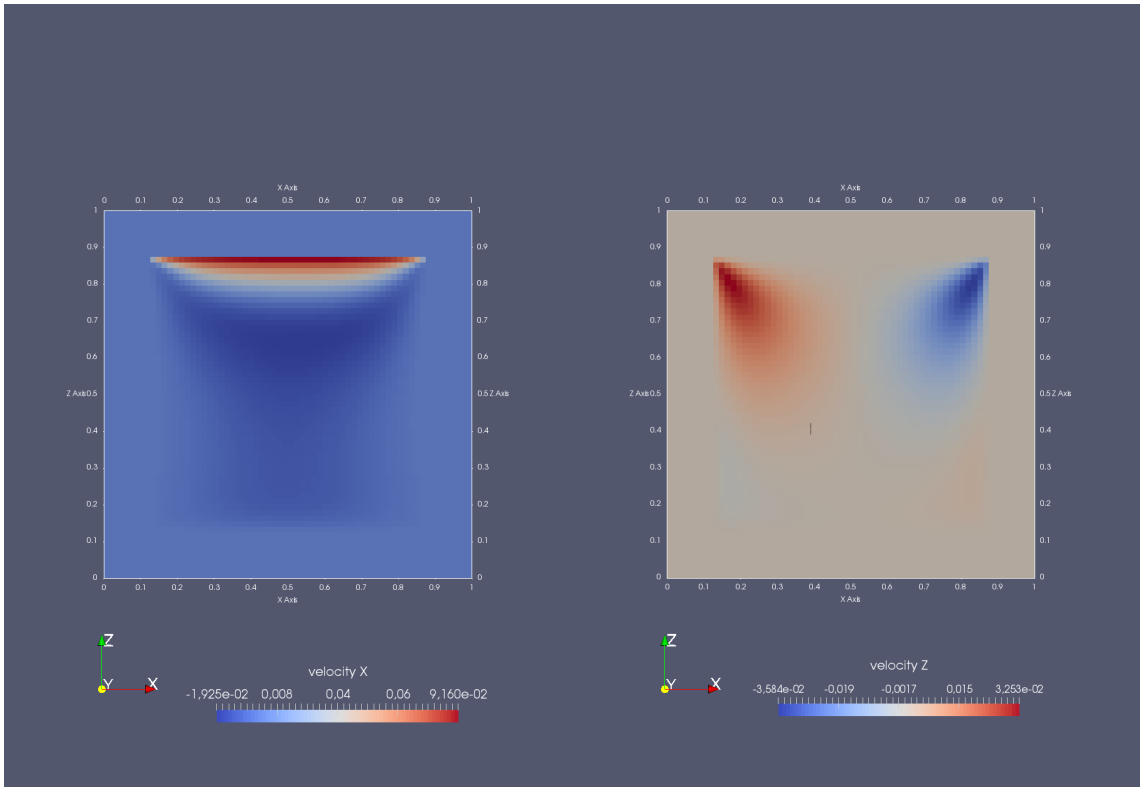


Abbildung 5.7: Links die Geschwindigkeiten entlang der X-Achse und rechts entlang der Z-Achse der Driven-Cavity Simulation Abbildung 5.6 auf der vorherigen Seite. Das Fluid bewegt sich am bewegenden Deckel entlang, wird dann durch die Begrenzung nach unten gelenkt und fließt dort über eine große Distanz verteilt langsam zurück, von wo es nach oben getrieben wird. Dies ergibt einen Kreislauf.

5.2.4 Starke Skalierung

Bei der starken Skalierung wird untersucht, wie sich die Laufzeit verhält, wenn die Problemgröße konstant bleibt und die verwendete Hardware ansteigt. Die Anzahl der Zellen bleibt konstant, während sich bei mehr Hardware die Simulationszeit verringern sollte. Das Ziel ist es möglichst bei einer Verdopplung der Hardware die Hälfte an Laufzeit zu benötigen. Durch die Art der Implementierung ist klar, dass für kleine Probleme dies nicht erreicht werden kann, da genügend Payloads vorhanden sein müssen um die SMs zu befüllen um eine hohe Auslastung der GPUs zu erreichen. Das System für die Tests ist das gleiche wie bei der schwachen Skalierung, auch wird wieder das Poiseuille-Script genutzt und die Integrationsschritte betrachtet.

In diesem Szenario ist der Einbruch der Effizienz noch größer als bei der schwachen Skalierung (Abbildung 5.13 auf Seite 55 und Abbildung 5.14 auf Seite 56). Dies ist zu erwarten, da durch die Aufteilung der Simulation die einzelnen Teilgebiete relativ klein werden und pro Prozess bei 32 Jobs nur noch 128 Payloads bearbeitet werden. Ein Aufruf mit 64 Jobs (Hyperthreading) wurde unternommen, die Simulation kann aber nicht davon profitieren, was sich auch deckt mit ähnlichen Versuchen der adaptiven CPU Implementierung.

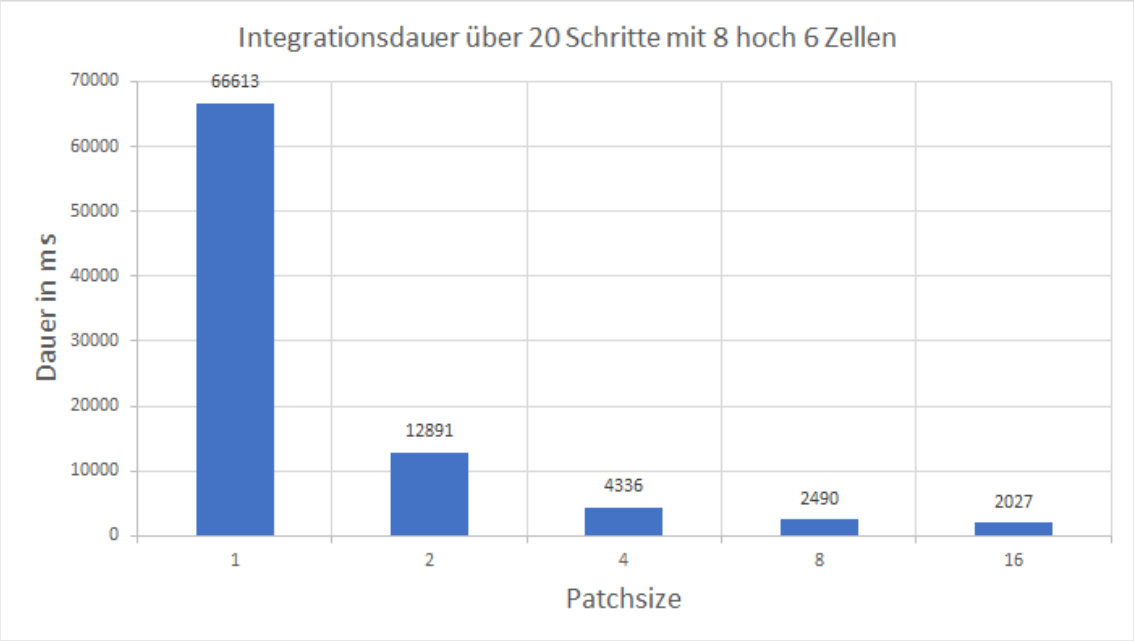


Abbildung 5.8: Vergleich der Integrationsdauer in Millisekunden über 20 Schritte mit 262144 Zellen. Dabei wurde ein CPU Kern und eine GPU verwendet.

5.2.5 Zwischenfazit

Für bis zu 8 Grafikkarte wird eine Skalierungseffizienz zwischen 65% (strong scaling) und 78% (weak scaling) erhalten. Leider lässt sich durch die vermutete Bandbreitenlimitierung nicht sinnvoll vorhersagen, wie sich die Implementierung auf anderen Systemen verhält. So wäre zum Beispiel ein Test auf einem Server mit einer AMD-EPYC CPU interessant. Diese hat 128 PCI Express lanes und kann dadurch theoretisch ganze 8 Grafikkarten pro CPU mit vollen 16 lanes mit der CPU verbinden. Wird diese Limitierung vernachlässigt, sollte die Skalierung auf mehrere nodes ähnlich verlaufen wie bei der adaptiven CPU-Implementierung, da die Baumstruktur über *P4EST* vergleichbar ist. Auch hier wären weitere Untersuchungen in Zukunft interessant. Von einem Plus an Kernen profitiert die Simulation ebenfalls. Bis zu einem Verhältnis von 4 Kernen pro Grafikkarte konnte eine Beschleunigung der Simulation erreicht werden. Bei den mit 4.2 Ghz taktendem Intel i7 6700k und der Grafikkarte Nvidia GTX 980 ti beträgt die Beschleunigung etwas über 30%. Bei den Intel Xeon Kernen, die nur mit 2,4 Ghz takten und mit der Nvidia GTX 1080 ti eine stärkere Grafikkarte zur Verfügung haben, ist die Beschleunigung sogar noch größer und beträgt bei den Tests zur starken Skalierung über 70%.

5.3 Vergleich mit anderen Implementierungen in *ESPResSo*

Vergleicht man die Laufzeiten der adaptiven CPU- und GPU-Implementierung, so hat sich der Aufwand gelohnt und ein Integrationsschritt auf der Grafikkarte wird deutlich schneller ausgeführt. Für die ausgeführten Simulationen der starken und schwachen Skalierung wird bei einer Grafikkarte und einem Kern ein Speedup von über 10 erhalten. Das genaue Plus an Leistung hängt dabei im

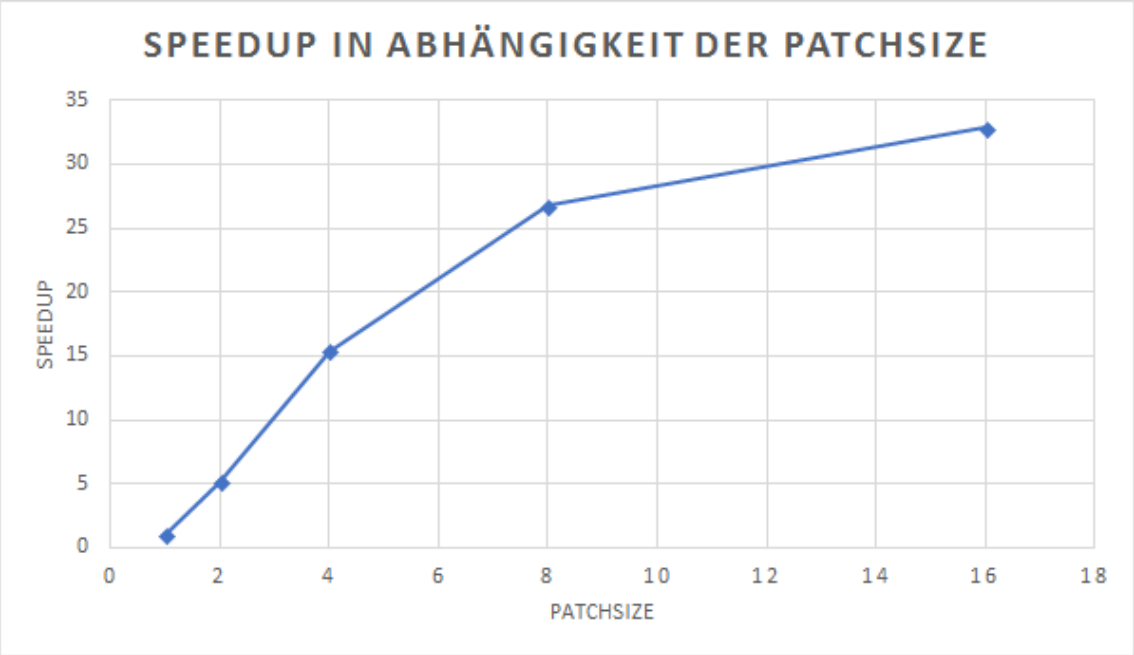


Abbildung 5.9: Untersuchung des Speedups bei einer Simulation über 20 Schritte mit 262144 Zellen anhand der Patchsize. Dabei wurde ein CPU Kern und eine GPU verwendet.

allgemeinen natürlich stark von der Problemgröße, der verfügbaren CPU und GPU ab. Die CPU-Implementierung skaliert bei der starken Skalierung deutlich besser, was damit zusammenhängen wird, dass die CPU auch bei einer kleinen Anzahl an Zellen ihre Last noch gut verteilen kann (Abbildung 5.15 auf Seite 56). Bei der schwachen Skalierung bleibt die GPU-Implementierung bis zu 8 Jobs um beinahe 10 mal schneller, da sich die Last auf der GPU kaum verändert (Abbildung 5.16 auf Seite 57).

Zusammenfassend lässt sich sagen, dass auf dem Server, der für die Tests benutzt worden ist, die GPU-Implementierung zu jedem Zeitpunkt schneller war als die CPU-Implementierung bei gleicher Anzahl an Jobs (Abbildung 5.17 auf Seite 57). Mit ungefähr 4 Kernen und 4 Grafikkarten wurde die Leistung übertroffen, welche die CPU-Implementierung mit 32 Kernen erreichte. Wie sich der Unterschied auf anderen Systemen oder auf mehreren Nodes auf einem Großrechner verhalten, lässt sich durch die Versuche nicht vorhersagen. Die CPU-Implementierung ist allerdings flexibler was die Größe der Simulation angeht. So skaliert sie auch sehr gut bei einer kleinen Anzahl Zellen und wird nicht so schnell durch den Speicher limitiert.

Im Vergleich zu der GPU-Implementierung von Roehm [RA12], die nur eine GPU unterstützt, ist diese Version deutlich langsamer, da bei einer GPU nicht bei jedem Integrationsschritt Daten über die CPU ausgetauscht werden müssen. Und gerade der Datenaustausch ist zum aktuellen Stand der zeitintensivste Schritt. Dafür kann die vorgestellte Implementierung auf mehreren Nodes laufen und erlaubt größere Domains zu simulieren, da die Daten auf mehrere GPUs verteilt wird.

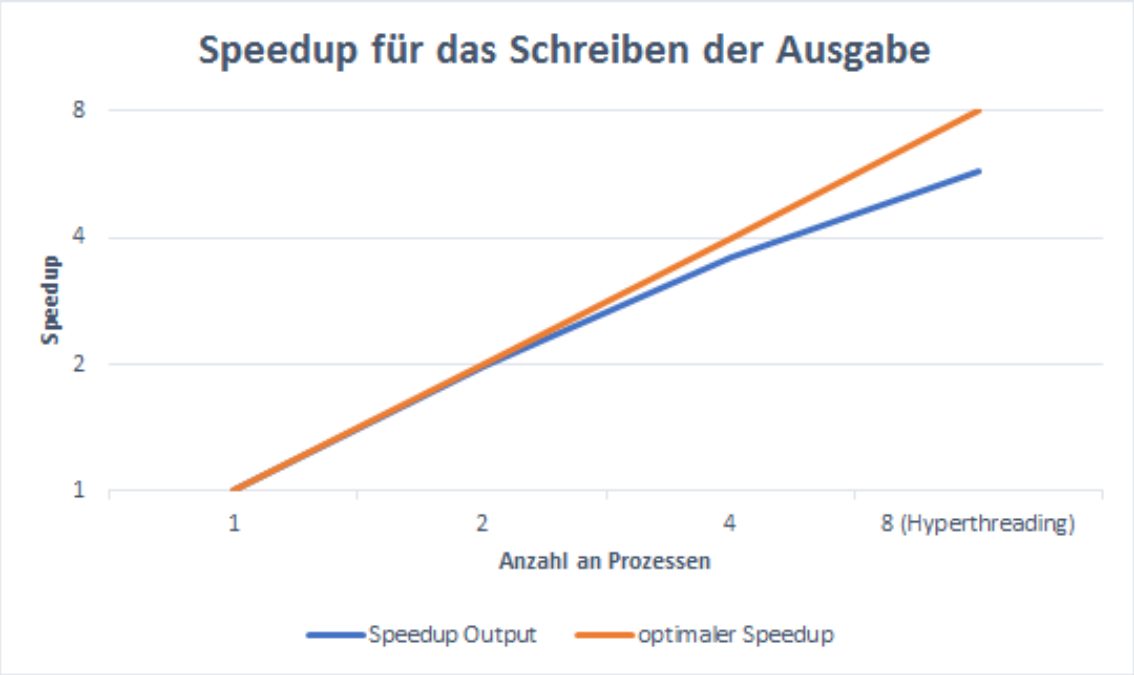


Abbildung 5.10: Speedup der Datenausgabe in Abhängigkeit der Prozessanzahl. Bei der Simulation wurden 2097152 beziehungsweise 8^7 Zellen bearbeitet. Die blaue Kurve entspricht den gemessenen Werte, die orangene Kurve gibt einn theoretische perfekten Speedup an.

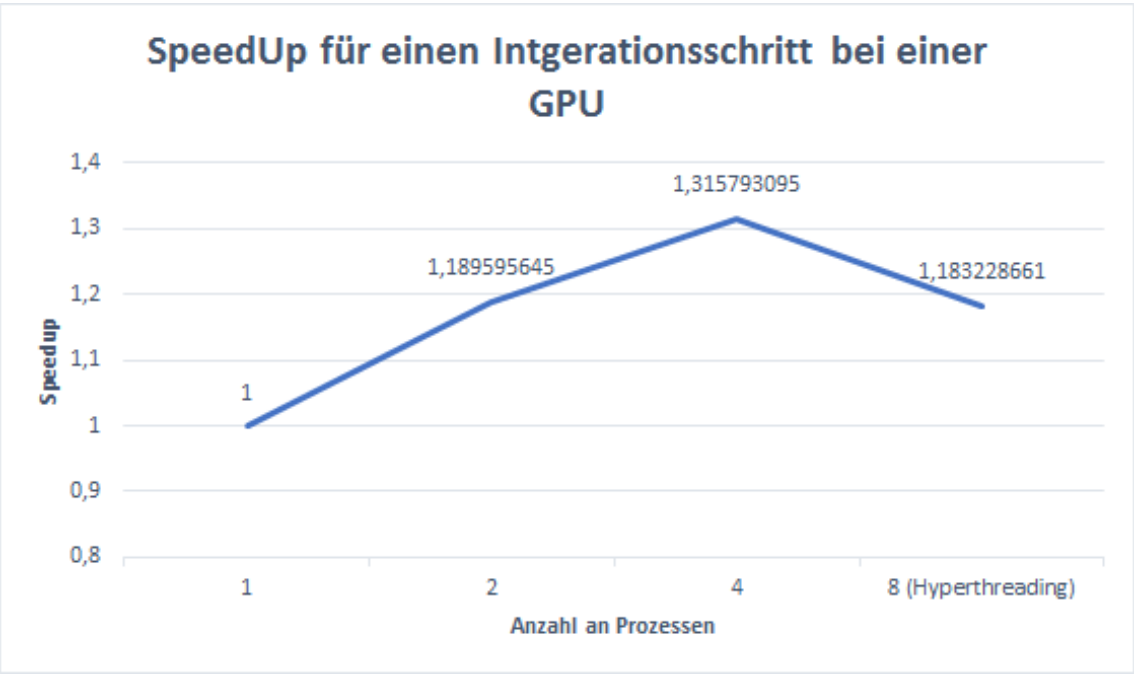


Abbildung 5.11: Speedup mit einer Grafikkarte in Abhängigkeit von der Anzahl MPI-Prozessen. Bei 4 Prozessen wird das Maximum erreicht, danach geht Leistung verloren.

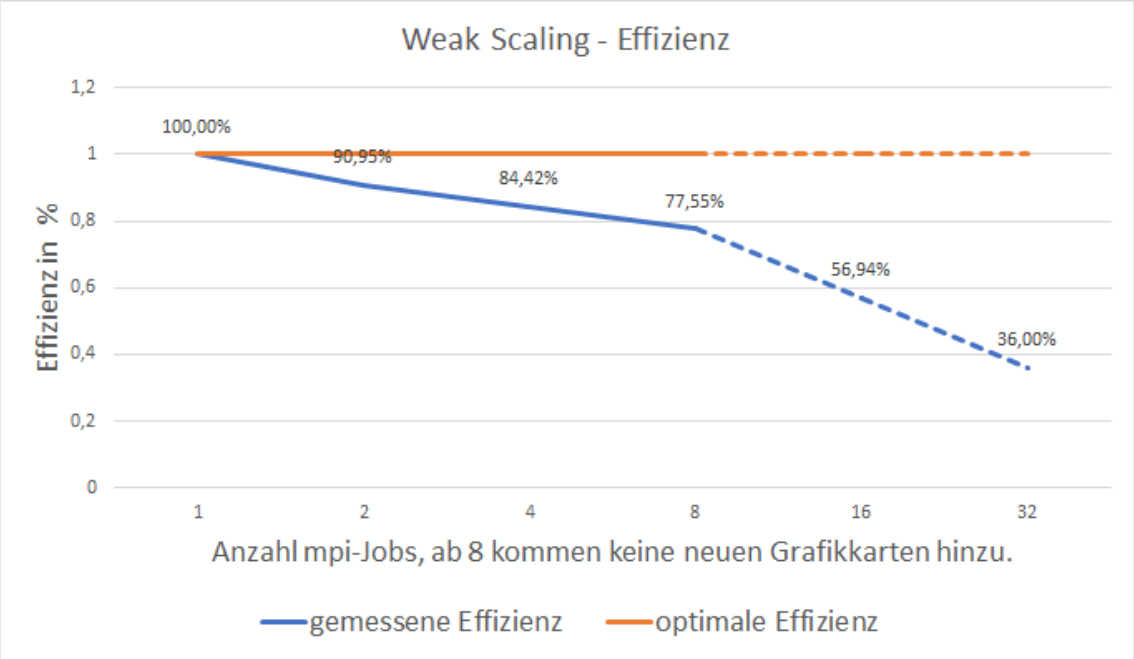


Abbildung 5.12: Simulation mit verschiedener Anzahl an MPI-Jobs und bis zu 8 Grafikkarten. Dabei wächst die Simulation von 2^{23} bis 2^{28} Zellen an. Nachdem keine weiteren Grafikkarten mehr hinzu kommen, nimmt die Effizienz rapide ab.

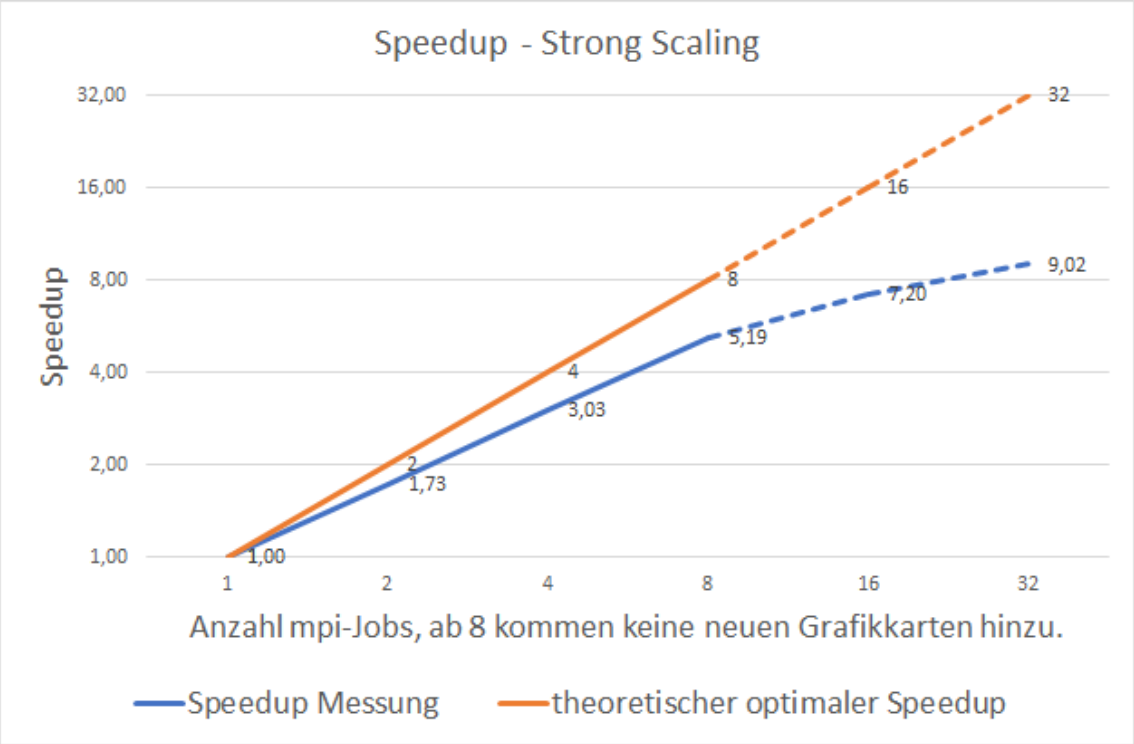


Abbildung 5.13: Simulation mit verschiedener Anzahl an MPI-Jobs und bis zu 8 Grafikkarten. Die Simulationen berechnet 2^{24} Zellen. Nachdem keine weiteren Grafikkarten mehr hinzu kommen, nimmt der Speedup rapide ab.

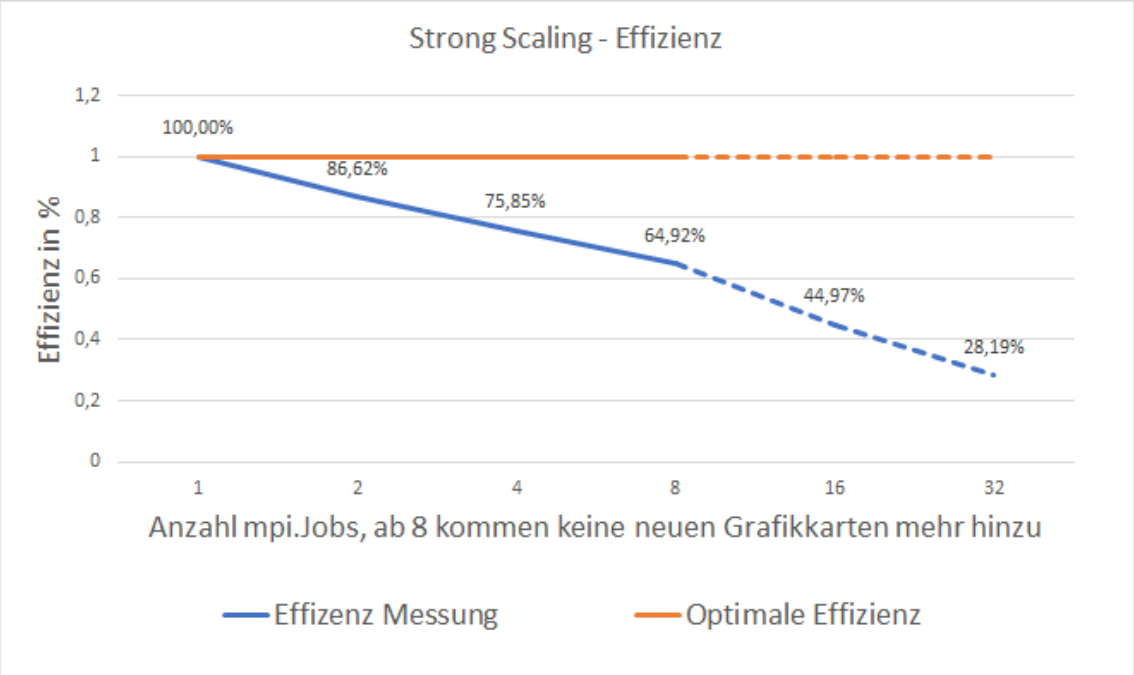


Abbildung 5.14: Simulation mit verschiedener Anzahl an MPI-Jobs und bis zu 8 Grafikkarten. Die Simulationen berechnet 2^{24} Zellen. Nachdem keine weiteren Grafikkarten mehr hinzu kommen, nimmt die Effizienz rapide ab.

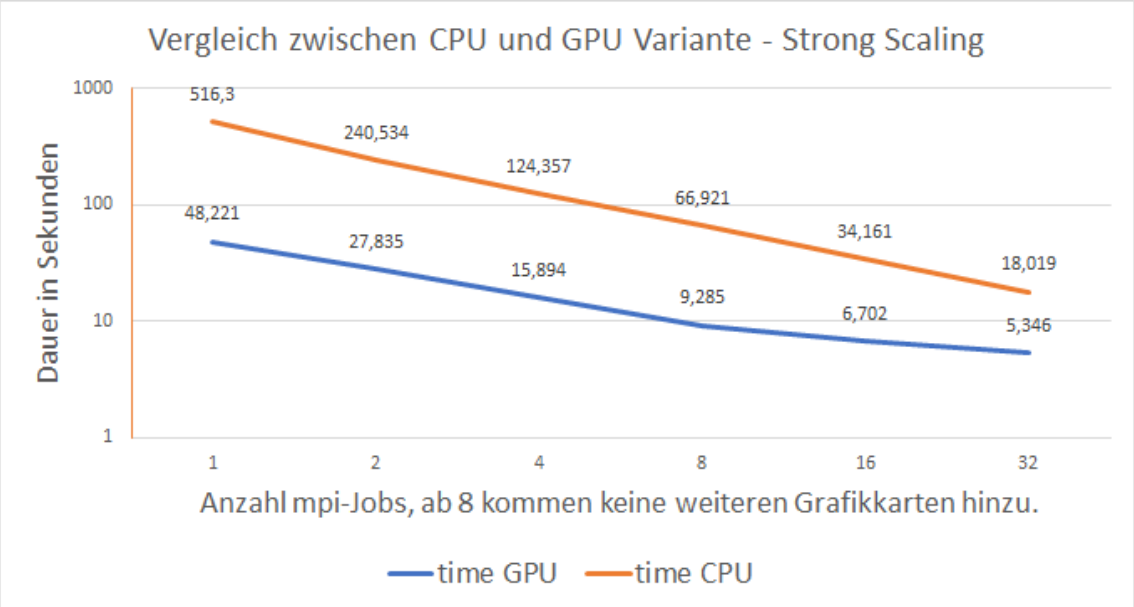


Abbildung 5.15: Vergleich der Dauer von 20 Simulationsschritten bei einer konstanten Zellanzahl von 2^24 in Abhängigkeit der MPI-Jobs. Bei der GPU-Implementierung werden maximal 8 Grafikkarten genutzt. Bis zu 8 Jobs ist die Berechnung fast 10 mal schneller, bei 32 Jobs beträgt der Unterschied noch Faktor 3.4.

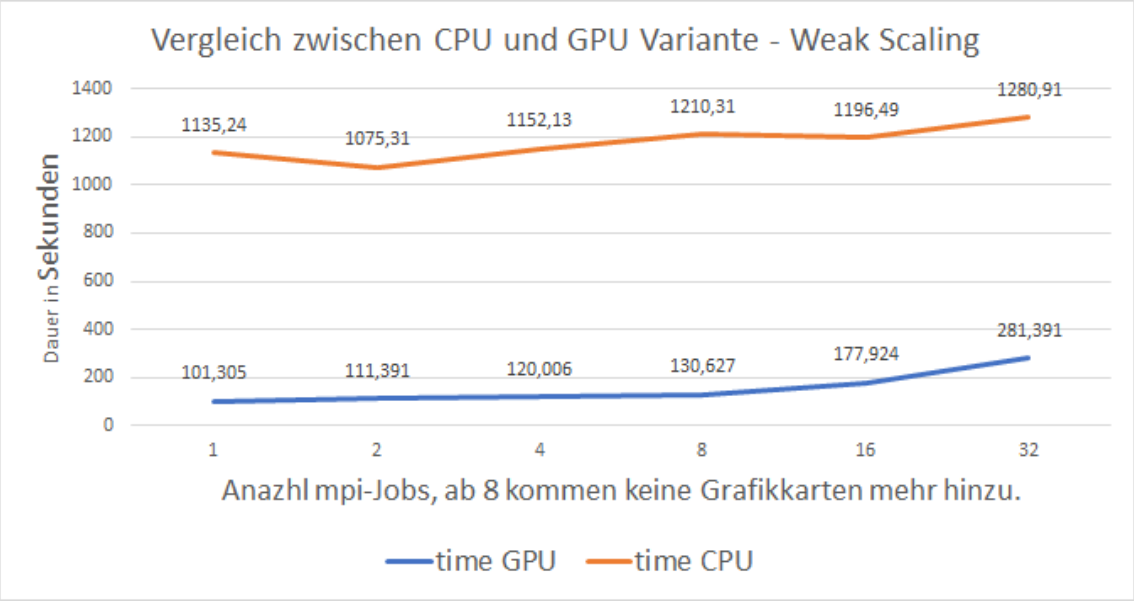


Abbildung 5.16: Vergleich der Dauer von 20 Simulationsschritten bei einer wachsenden Anzahl von Zellen proportional zu der Anzahl an MPI-Jobs. Bei der GPU-Implementierung werden maximal 8 Grafikkarten genutzt. Bei einem Job ist die Berechnung über 11 mal schneller, bei 32 Jobs beträgt der Unterschied noch Faktor 4.6.

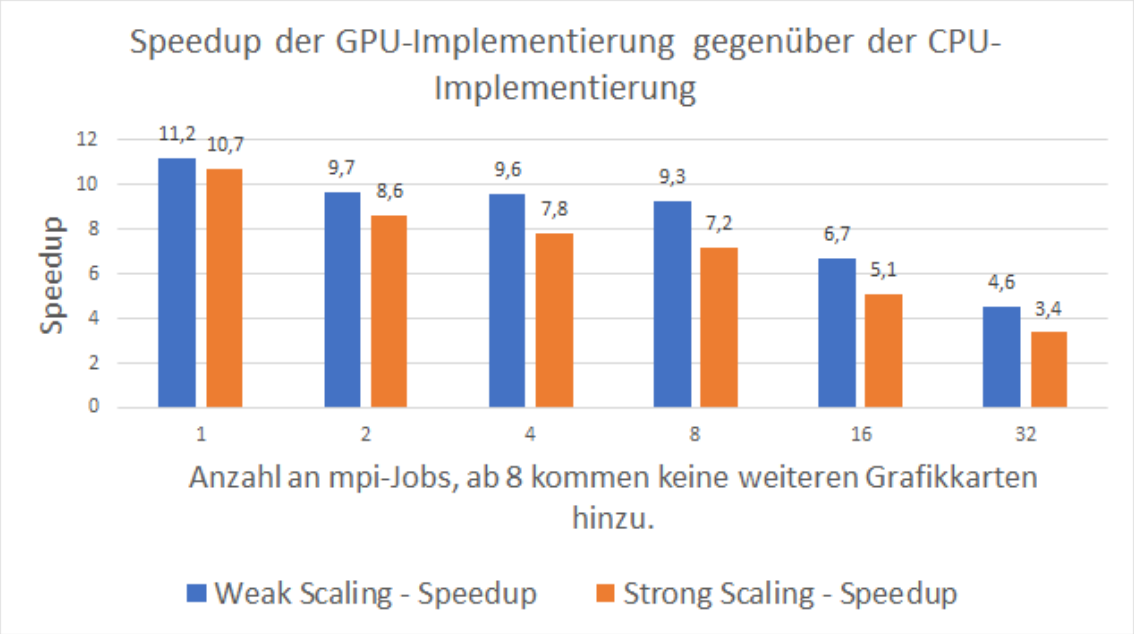


Abbildung 5.17: Speedup der GPU-Implementierung gegenüber der CPU-Implementierung bezüglich starker und schwacher Skalierung in Abhängigkeit von der Anzahl an MPI-Jobs.

6 Zusammenfassung und Ausblick

In dieser Arbeit wird eine Implementierung der Lattice-Boltzmann Methode für mehrere Grafikkarten vorgestellt, die das Softwarepaket *ESPResSo* erweitert. Diese basiert auf der vorhandenen Implementierung von [LABM16; LBH+16] und verwendet die Bibliothek *P4EST* [BWG11] um mit Oktalbäumen ein Gitter zu erstellen. Dabei liegt der Fokus auf der parallelen Berechnung mit Grafikkarten. Für diesen Zweck werden einige Anpassungen an die vorhanden Strukturen gemacht.

Das Programm unterteilt zuerst das Simulationsgebiet in mehrere Oktalbäume, so dass rekursiv ein reguläres Gitter mit der gewünschten Tiefe entsteht. Anhand dieser Oktalbäume kann die Last auf mehrere Prozesse verteilt werden. Die Konstruktion des so entstandenen regulären Gitters, die Nachbarschaftsbeziehungen und die Kommunikation zwischen den verschiedenen Prozessen werden durch die *P4EST* Bibliothek, wie in Kapitel 2.2, beschrieben gesteuert. In Kapitel 3 wird erläutert wie die Boltzmann-Gleichung (Kapitel 3.1) zustande kommt und wie die LBM diskretisiert (Kapitel 3.2) wird. Ergebnisse daraus sind, dass ein *D3Q19*-Gitter (Kapitel 3.3.1) verwendet wird und sich die LBM in 2 Schritte (Kapitel 3.3.2) aufteilen lässt. Bei den 2 Schritten handelt es sich um einen lokalen Kollisionsschritt, der sich optimal eignet parallel ausgeführt zu werden, und einen Strömungsschritt, der nur von den direkten Nachbarn abhängig ist und der sich mit den benutzten no-slip bounce back Randbedingungen kombinieren lässt (Kapitel 3.3.3). Um diesen Strömungsschritt gut zu parallelisieren und auf einer GPU zu berechnen, müssen auf der Grafikkarte die Nachbarschaftsbeziehungen bekannt sein. Dafür werden die einzelnen Oktanten, die *P4EST* erstellt, nochmals in mehrere Zellen erteilt, diese werden Patch genannt. Um diesen Patch wird von den benachbarten Patches eine weitere Schicht dieser unterteilten Zellen gelegt, den sogenannten Halo. Der Patch ergibt zusammen mit seinem Halo eine Einheit, die sogenannte *Payload*. Auf der Payload sind die Nachbarschaftsbeziehungen der einzelnen Zellen durch eine reguläres Gitter bekannt und so kann der Kollisions- und Strömungsschritt auf der GPU ausgeführt werden, bevor die Payloads zurück auf die CPU kopiert werden. Dort werden die Halos anhand der Nachbarzellen wieder neu beschreiben.

Bei einer mit der Implementierung durchgeführten Simulation wird diese, wie in allen *ESPResSo*-Simulationen, mit einem Script über die Konsole gestartet. Dabei können verschiedene Parameter übergeben werden. In dem Simulationsscript wird die Domain mit ihren Boundaries, Partikeleigenschaften und möglichen externen Kräften festgelegt. Anhand dessen wird ein Gitter erstellt und die Last zwischen den gestarteten Prozessen verteilt. Ebenso wird im Script festgelegt, wie oft kollidiert und geströmt wird und wann beziehungsweise ob Daten ausgegeben werden. Ein Kollisions- und Strömungsschritt ergibt zusammen einen Integrationsschritt. Dieser wird dann auf der GPU ausgeführt. Die Ausgabe der Daten findet auf der CPU statt und liefert vtk Dateien, die mit Programmen wie Paraview analysiert und visualisiert werden können.

In Kapitel 5.1 wird anhand bekannter Probleme gezeigt, dass die implementierte LBM die gewünschten Ergebnisse liefert und sich dadurch für die vorgesehenen Simulationen eignet. Danach wird die Skalierung untersucht und die Implementierung mit der adaptiven CPU Implementierung verglichen (Kapitel 5.2 - 5.3). Dabei wird festgestellt, dass die GPU Variante um ungefähr einen Faktor 10 schneller ist, aber dieser Vorsprung auf der getesteten Umgebung mit wachsender Skalierung nachlässt, bis, bei einigermaßen voller Ausnutzung der Umgebung, der Vorsprung auf ungefähr einen Faktor 4 schrumpft. Dies liegt hauptsächlich daran, dass auf der Umgebung nur 8 GPUs vorhanden sind und mit wachsender Anzahl an Jobs diese sich eine GPU teilen müssen.

Ausblick

Das Programm kann auf unterschiedliche Art und Weise erweitert und optimiert werden. Es hat sich herausgestellt, dass die Kopiervorgänge zwischen CPU und GPU ein stark limitierender Faktor sind. Hier lassen sich verschiedene Optimierungen durchführen. Einerseits kann die Kopierfunktion aufgeteilt und asynchron ausgeführt werden, anderseits kann die Datenstruktur optimiert werden und nur die wirklich benötigten Daten für den Halo-Austausch transferiert werden. Anderseits lässt sich eine Kopplung mit anderen Simulationen wie der von [Bru17] vorgestellten Partikelsimulation umsetzen. Durch *P4EST* beherrscht die Implementierung ein adaptives Gitter, nutzt es aber nicht. Mit einigen Ergänzungen wie dem adaptiven Integrationsschritt lässt sich die komplette Implementierung mit relativ geringem Aufwand vollständig adaptiv dynamisch machen. Eventuell könnte es auch sinnvoll sein, die Implementierung aufzuteilen in einen regulären Fall, bei dem die Patches maximiert werden und ihre Halos vergrößert, so dass nur alle n-Schritte Daten mit der CPU und anderen Nachbarn ausgetauscht werden müssen, und einen adaptiv dynamischen Fall, der die jetzige Struktur beibehält und den Datenaustausch minimiert durch eine optimierte Kopierfunktion.

Literaturverzeichnis

[20116] C. 2018. *Piz Daint, one of the most powerful supercomputers in the world*. 2016. URL: https://www.cscs.ch/computers/piz-daint/ (zitiert auf S. 39).

[Ach90] D. Acheson. *Elementary Fluid Dynamics*. Oxford Applied Mathematics and Computing Science Series. Clarendon Press, 1990. ISBN: 9780198596790. URL: https://books.google.de/books?id=IGfDBAAAQBAJ (zitiert auf S. 43).

[ALK+13] A. Arnold, O. Lenz, S. Kesselheim, R. Weeber, F. Fahrenberger, D. Roehm, P. Košovan, C. Holm. „ESPResSo 3.1 — Molecular Dynamics Software for Coarse-Grained Models“. In: *Meshfree Methods for Partial Differential Equations VI*. Hrsg. von M. Griebel, M. A. Schweitzer. Bd. 89. Lecture Notes in Computational Science and Engineering. Springer, 2013, S. 1–23. DOI: 10.1007/978-3-642-32979-1_1. URL: http://www.springer.com/mathematics/computational+science+%26+engineering/book/978-3-642-32978-4 (zitiert auf S. 13, 15).

[BCMT13] C. Burstedde, D. A. Calhoun, K. T. Mandli, A. R. Terrel. „ForestClaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws“. In: *CoRR* abs/1308.1472 (2013). arXiv: 1308.1472. URL: http://arxiv.org/abs/1308.1472 (zitiert auf S. 32).

[BGK54] P. L. Bhatnagar, E. P. Gross, M. Krook. „A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems“. In: *Phys. Rev.* 94 (3 Mai 1954), S. 511–525. DOI: 10.1103/PhysRev.94.511. URL: https://link.aps.org/doi/10.1103/PhysRev.94.511 (zitiert auf S. 26).

[Bru17] M. Brunn. „Coupling of particle simulation and lattice boltzmann background flow on adaptive grids“. Master's thesis. Universit¨at Stuttgart, 2017 (zitiert auf S. 15, 60).

[BS06] C.-H. Bruneau, M. Saad. „The 2D lid-driven cavity problem revisited“. In: *Computers & Fluids* 35.3 (2006), S. 326–348. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2004.12.004. URL: http://www.sciencedirect.com/science/article/pii/S0045793005000368 (zitiert auf S. 45).

[BWG11] C. Burstedde, L. C. Wilcox, O. Ghattas. „P4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees“. In: *SIAM Journal on Scientific Computing* 33.3 (2011), S. 1103–1133. DOI: 10.1137/100791634 (zitiert auf S. 13, 15, 16, 18, 59).

[CB17] D. A. Calhoun, C. Burstedde. „ForestClaw: A parallel algorithm for patch-based adaptive mesh refinement on a forest of quadtrees“. In: *CoRR* abs/1703.03116 (2017). arXiv: 1703.03116. URL: http://arxiv.org/abs/1703.03116 (zitiert auf S. 32).

[Cor18a] N. Corporation. *CUDA C Best Practices Guide*. 2018. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (zitiert auf S. 32).

[Cor18b] N. Corporation. *NVIDIA CUDA C Programming Guide*. Version 9.1.85. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 2018. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (zitiert auf S. 14, 32).

[DGFK07] J. Dongarra, D. Gannon, G. Fox, K. Kennedy. „The Impact of Multicore on Computational Science Software“. In: 3 (Jan. 2007) (zitiert auf S. 13).

[DL09] B. Dünweg, A. J. C. Ladd. „Lattice Boltzmann Simulations of Soft Matter Systems“. In: *Advanced Computer Simulation Approaches for Soft Matter Sciences III*. Hrsg. von C. Holm, K. Kremer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 89–166. ISBN: 978-3-540-87706-6. DOI: 10.1007/978-3-540-87706-6_2. URL: https://doi.org/10.1007/978-3-540-87706-6_2 (zitiert auf S. 15).

[FHP86] U. Frisch, B. Hasslacher, Y. Pomeau. „Lattice-Gas Automata for the Navier-Stokes Equation“. In: *Phys. Rev. Lett.* 56 (14 Apr. 1986), S. 1505–1508. DOI: 10.1103/PhysRevLett.56.1505. URL: https://link.aps.org/doi/10.1103/PhysRevLett.56.1505 (zitiert auf S. 28).

[IBWG15] T. Isaac, C. Burstedde, L. C. Wilcox, O. Ghattas. „Recursive algorithms for distributed forests of octrees“. In: *SIAM Journal on Scientific Computing* 37.5 (2015), S. C497–C531. DOI: 10.1137/140970963 (zitiert auf S. 16, 18).

[Kir10] B. Kirby. *Micro- and Nanoscale Fluid Mechanics: Transport in Microfluidic Devices*. Cambridge University Press, 2010. ISBN: 9781139489836. URL: https://books.google.de/books?id=y7PB9f5zmU4C (zitiert auf S. 44).

[LABM16] M. Lahnert, T. Aoki, C. Burstedde, M. Mehl. „Minimally-Invasive Integration of P4est in Espresso for Adaptive Lattice-Boltzmann“. Deutsch. In: *The 30th Computational Fluid Dynamics Symposium ; Tokyo, Japan, December 12–14, 2016*. Japan Society of Fluid Mechanics, Dezember 2016, S. 1–7. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2016-57&engl=0 (zitiert auf S. 14, 15, 59).

[LAMH06] H. Limbach, A. Arnold, B. Mann, C. Holm. „ESPResSo—an extensible simulation package for research on soft matter systems“. In: *Computer Physics Communications* 174.9 (2006), S. 704–727. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2005.10.005. URL: http://www.sciencedirect.com/science/article/pii/S001046550500576X (zitiert auf S. 13, 15).

[LBH+16] M. Lahnert, C. Burstedde, C. Holm, M. Mehl, G. Rempfer, F. Weik. „Towards Lattice-Boltzmann on Dynamically Adaptive Grids – Minimally-Invasive Grid Exchange in ESPResSo“. Englisch. In: *ECCOMAS Congress 2016, VII European Congress on Computational Methods in Applied Sciences and Engineering*. Hrsg. von M. Papadrakakis, V. Papadopoulos, G. Stefanou, V. Plevris. ECCOMAS, Juni 2016, S. 1–25. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2016-20&engl=0 (zitiert auf S. 14, 15, 23, 36, 59).

[Li15] J. Li. *Appendix: Chapman-Enskog Expansion in the Lattice Boltzmann Method*. Techn. Ber. arXiv:1512.02599. Comments: Complete and general derivation with discussions. Dez. 2015. URL: https://cds.cern.ch/record/2112003 (zitiert auf S. 27).

[MKM+15] A. Müller, M. A. Kopera, S. Marras, L. C. Wilcox, T. Isaac, F. X. Giraldo. „Strong Scaling for Numerical Weather Prediction at Petascale with the Atmospheric Model NUMA“. In: *CoRR* abs/1511.01561 (2015). arXiv: 1511.01561. URL: http://arxiv.org/abs/1511.01561 (zitiert auf S. 16).

[Mor66] G. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966. URL: https://books.google.de/books?id=9FFdHAAACAAJ (zitiert auf S. 17).

[RA12] D. Roehm, A. Arnold. „Lattice Boltzmann simulations on GPUs with ESPResSo“. In: *The European Physical Journal Special Topics* 210.1 (Aug. 2012), S. 89–100. ISSN: 1951-6401. DOI: 10.1140/epjst/e2012-01639-6. URL: https://doi.org/10.1140/epjst/e2012-01639-6 (zitiert auf S. 15, 52).

[Sch08] U. D. Schiller. „Thermal fluctuations and boundary conditions in the lattice Boltzmann method.“ Diss. Johannes Gutenberg-Universität, Mainz, 2008 (zitiert auf S. 13, 15, 23, 27).

[Ser17] T. C. Services. *TSUBAME3.0*. 2017. URL: http://www.t3.gsic.titech.ac.jp/en/hardware (zitiert auf S. 39).

[Suc13] S. Succi. *The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond*. OUP Oxford, 2013. ISBN: 9780199679249. URL: https://books.google.de/books?id=x9KwNAEACAAJ (zitiert auf S. 23–26, 28).

[TOP18a] TOP500.org. *Green500 List for November 2017*. 2018. URL: https://www.top500.org/green500/lists/2017/11/ (zitiert auf S. 39).

[TOP18b] TOP500.org. *Top500 List - November 2017*. 2018. URL: https://www.top500.org/list/2017/11/ (zitiert auf S. 39).

[Wol00] D. A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models. An Introduction*. Springer-Verlag Berlin Heidelberg, 2000. ISBN: 978-3-540-66973-9. DOI: 10.1007/b72010 (zitiert auf S. 23).

Alle URLs wurden zuletzt am 04. 05. 2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift