Institut für Parallele und Verteilte Systeme
Abteilung Simulation großer Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit

Integration einer generischen Gitterschnittstelle in ESPResSo

Daniel Eschbach

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Miriam Mehl
Betreuer/in:	Dipl.-Inf. Michael Lahnert
Beginn am:	12. September 2016
Beendet am:	14. März 2017
CR-Nummer:	I.6.3

Kurzfassung

Es wurde eine Lattice-Boltzmann-Implementierung erstellt, die auf einem adaptiven Gitter arbeitet. Gleichzeitig wurde das generische Gitterschnittstellensystem GGI dahingehend überarbeitet und präzisiert, dass es den Anforderungen dieser Lattice-Boltzmann-Implementierung gerecht wird.
Die adaptive Lattice-Boltzmann-Implementierung und das Gitterschnittstellensystem wurden in ESPResSo integriert, wobei die alte Lattice-Boltzmann-Implementierung entfernt wurde. ESPResSo vermag nun Fluidvolumina sowohl mit regulärer als auch adaptiver Auflösung zu simulieren.
Die Validierungstests ergaben, dass ESPResSo mit der neuen Lattice-Boltzmann-Implementierung für Fluidvolumina, die es mit regulärer Auflösung simuliert, dieselben Resultate errechnet wie mit der alten Implementierung.
Die Performancetests zeigten auf, dass die neue Lattice-Boltzmann-Implementierung die Simulation von Fluidvolumina mit regulärer Auflösung im Vergleich zur alten Lattice-Boltzmann-Implementierung zwar verlangsamt, aber für Fluidvolumina mit adaptiver Auflösung die erwarteten Performancevorteile realisiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

ESPResSo ist eine Software für die mesoskopische Simulation weicher Materie [LAMH06]. ESPResSo bietet eine Vielzahl von einsetzbaren Algorithmen aus dem Bereich der Molekulardynamik an. Ferner erlaubt ESPResSo die Kopplung der simulierten Partikel mit einer simulierten Hintergrundströmung. Zur Simulation einer solchen Hintergrundströmung stellt ESPResSo eine Lattice-Boltzmann-Implementierung zur Verfügung, welche die fluktuierende Navier-Stokes-Gleichung approximiert [Sch08].

Gegenwärtig wird ein reguläres kartesisches Gitter eingesetzt. Da Speicherverbrauch und Verarbeitungsaufwand kubisch mit der Gittergröße und -auflösung wachsen, stößt man schnell auf Kapazitätsprobleme, wenn man den Simulationsraum auf realistische Größen und Genauigkeiten auszudehnen sucht.

Ein adaptives Gitter würde es erlauben, Verarbeitungs- und Speicherkapazität auf die Bereiche des Simulationsraums zu konzentrieren, in denen eine hohe Genauigkeit tatsächlich erforderlich ist. Bei einer Lattice-Boltzmann-Implementierung, die mit einer Molekulardynamiksimulation gekoppelt ist, wären das z.B. die unmittelbaren Umgebungen der simulierten Partikel oder Hindernisse. Daher soll im Rahmen dieser Arbeit ein baumstrukturiertes kartesisches Gitter integriert werden, welches durch die p4est-Library implementiert wird [LBH+16] [BWG11].

Vorarbeiten

Im Rahmen von [Esc16] wurde eine *generische Gitterschnittstelle* entwickelt, die sowohl mit einer Implementierung für reguläre kartesische Gitter als auch mit einer p4est-basierten Implementierung für baumstrukturierte kartesische Gitter unterlegt werden kann. Die generische Gitterschnittstelle und ihre beiden Implementierungen für reguläre bzw. für baumstrukturierte Gitter wurden zusammengefasst zum *generischen Gitterschnittstellensystem GGI*. Auch wurde die Lattice-Boltzmann-Implementierung aus ESPResSo dahingehend entkoppelt, dass sie auf einem regulären kartesischen Gitter arbeitet, das durch GGI bereitgestellt wird [Esc16].

Aufgabenstellung

Im Rahmen dieser Arbeit soll die entkoppelte Lattice-Boltzmann-Implementierung zu einer *adaptiven Lattice-Boltzmann-Implementierung* weiterentwickelt werden, die nicht nur auf einem adaptiven kartesischen Gitter arbeitet, das GGI ihr bereitstellen kann, sondern dessen Adaptivität auch nutzt. Dazu sollen die generische Gitterschnittstelle von GGI sowie seine Schnittstellenimplementierung für baumstrukturierte kartesische Gitter vervollständigt und präzisiert werden. Die adaptive Lattice-Boltzmann-Implementierung soll dann gemeinsam mit dem generischen Gitterschnittstellensystems GGI in ESPResSo integriert werden, um die dortige Lattice-Boltzmann-Implementierung zu ersetzen.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2: umreißt das Problemumfeld, welches die Aufgabenstellung dieser Arbeit umfasst, und beschreibt die bereits durchgeführten Vorarbeiten, auf denen diese Arbeit ihre Lösung aufbauen wird.

Kapitel 3, 4, 5 und 6 beschreiben die Komponenten der Lösung dieser Arbeit: die adaptive Lattice-Boltzmann-Implementierung, das überarbeitete generische Gitterschnittstellensystem GGI und schließlich Vorgehen und Erfolgsumfang ihrer gemeinsamen Integration in ESPResSo.

Kapitel 7 misst und bewertet die integrierte Lösung dieser Arbeit.

Kapitel 8 fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Problemumfeld und Vorarbeiten

2.1 Lattice-Boltzmann

2.1.1 Die Boltzmann-Gleichung

Ein Fluidvolumen kann beschrieben werden durch eine Partikelverteilungsfunktion $f(\vec{x}, \vec{v}, t)$, welche sinngemäß zu jedem Zeitpunkt t und für jeden Raumpunkt $\vec{x}$ angibt, wie viele Partikel mit Geschwindigkeit $\vec{v}$ unterwegs sind. Statt über der Partikelgeschwindigkeit $\vec{v}$ kann f auch über dem Impuls $\vec{p}$ definiert sein [Suc01].

Die Boltzmann-Gleichung ergibt sich als:

$$\frac{\partial f}{\partial t} + \vec{v}\frac{\partial f}{\partial \vec{x}} + \frac{\vec{F}}{m}\frac{\partial f}{\partial \vec{v}} = C(f)$$

Die zeitliche Änderung $\frac{\partial f}{\partial t}$ der Partikeldichte in einem Phasenraumvolumen $d^3\vec{x}d^3\vec{v}$ ergibt sich demnach aus dem Zu-/Abfluss $\vec{v}\frac{\partial f}{\partial \vec{x}}$ von Partikeln bzgl. räumlich benachbarter Phasenraumvolumina und aus dem (kraftfeldbedingten) Zu-/Abfluss $\frac{\vec{F}}{m}\frac{\partial f}{\partial \vec{v}}$ von Partikeln bzgl. Phasenraumvolumina benachbarter Geschwindigkeiten abweichlich des Zu-/Abflusses $C(f)$ von Partikeln, deren Geschwindigkeit sich infolge von Kollisionen ändern [Suc01].

Nach dem *Stoßzahlansatz* kann der Kollisionsterm $C(f)$ angenähert werden als:

$$C(f) = \int \Big(f(\vec{x}, \vec{v}_1, t) f(\vec{x}, \vec{v}_2, t) - f(\vec{x}, \vec{v}_3, t) f(\vec{x}, \vec{v}, t) \Big) g\sigma(g, \Omega) d\Omega d\vec{v}_1 d\vec{v}_2 d\vec{v}_3$$

Der Zu-/Abfluss von Partikeln infolge von Kollisionen wird demnach modelliert als der Saldo aus allen kollidierenden Partikelpaaren mit den eingehenden Geschwindigkeiten $\vec{v}_1$, $\vec{v}_2$ und ausgehenden Geschwindigkeiten $\vec{v}_1$, $\vec{v}$ im differenziellen Wirkungsquerschnitt σ über alle Raumwinkel $d\Omega$ [Suc01].

Ein einfacherer Ansatz nach Bhatnagar, Gross und Krook modelliert den Zu-/Abfluss von Partikeln infolge von Kollisionen als die Abweichung der Partikelverteilung f von der rechnerischen Partikelverteilung f^{eq}, die dem lokalen thermodynamischen Gleichgewicht entspricht:

$$C(f) = \nu(f^{eq} - f)$$

Somit wird die lokale Partikelverteilung f zu f^{eq} hin relaxiert mit ν als Relaxierungsfaktor [BGK54].

2.1.2 Die Lattice-Boltzmann-Gleichung

Um eine numerische Lösung der Partikelverteilung f zu erreichen, wird die Boltzmann-Gleichung in Phasenraum und Zeit diskretisiert. Anstelle einer kontinuierlichen Zeitfortschreibung betrachtet man die Entwicklung der Partikelverteilung in diskreten Zeitschritten Δt. Statt kontinuierlich über dem Fluidvolumen betrachtet man die Ausprägung der Partikelverteilung auf einem Gitter von diskreten Punkten $(\vec{x}_G)$ [Esc16] [Suc01].

Ebenso betrachtet man die Partikelverteilung nicht über dem Spektrum aller möglichen Geschwindigkeiten, sondern nur für eine endliche Anzahl diskreter Geschwindigkeiten $\vec{v}_i$. Wird ein strukturiertes Rechengitter verwendet, entsprechen die Richtungen $\vec{e}_i$ der diskreten Geschwindigkeiten $\vec{v}_i$ gewöhnlich einer Auswahl aus den Richtungen, die von einem Ausgangsgitterpunkt $\vec{x}$ aus zu dessen unmmittelbaren Nachbargitterpunkten $(\vec{x}_i | \vec{x}_i \in Nachbar(\vec{x}))$ weisen. Die Beträge der diskreten Geschwindigkeiten $\vec{v}_i$ wählt man dann üblicherweise so, dass ein Partikel von einem Ausgangsgitterpunkt $\vec{x}$ aus in einem diskreten Zeitschritt Δt den Zielnachbargitterpunkt $\vec{x}_i$ erreicht. Somit ergibt sich aus dieser Wahl der diskreten Geschwindigkeiten $\vec{v}_i$ für einen Gitterpunkt $\vec{x}$ seine Nachbarschaft als $(\vec{x} + \Delta t \vec{v}_i)$ [Esc16] [Suc01].

Die verschiedenen möglichen Essembles an diskreten Geschwindigkeiten $\vec{v}_i$ werden über die D*n*Q*m*-Notation klassifiziert, welche für ein Essemble jeweils Dimensionalität und die Anzahl der verwendeten diskreten Geschwindigkeiten angibt. So arbeitet z. B. eine D2Q4-Lattice-Boltzmann-Implementierung auf einem zweidimensionalen Rechengitter und betrachtet für jeden Rechenpunkt die Partikelverteilung für vier Geschwindigkeiten, während eine D3Q27-Lattice-Boltzmann-Implementierung auf einem dreidimensionalen Rechengitter und mit 27 Geschwindigkeiten je Gitterpunkt arbeitet [Suc01].

Die Lattice-Boltzmann-Gleichung ergibt sich als:

$$f_i(\vec{x} + \vec{v}_i \Delta t, t + \Delta t) = f_i(\vec{x}, t) + \sum_k L_{ik}(f_k^{eq} - f_k) + \Psi_i$$

Die linke Gleichungsseite bildet das Strömen zwischen Gitterpunkten ab, während auf der rechten Seite der Kollisionsterm sowie der Kraftterm Ψ_i stehen, welche Kollision und Kraftfeldeinwirkung modellieren. Der Kollisionsterm ist mit einem linearen Operator L_{ik} formuliert, was es erlaubt, beliebige Linearkombinationen von $f_k^{eq} - f_k$ mit unterschiedlichen Relaxierungsfaktoren zu versehen. Ist $L_{ik} = \nu \delta_{ik}$ ergibt sich aus diesem LBGK-Kollisionsoperator wieder der Bhatnagar-Gross-Krook-Kollisionsoperator [LBH+16] [Sch08].

2.1.3 Algorithmus

In der Umsetzung wird jeder Gitterpunkt $\vec{x}$ mit je zwei Sätzen von Speicherstellen s_i und s_i^* versehen, seiner *Zeitschrittdatenstruktur*, welche für die diskreten Geschwindigkeiten $\vec{v}_i$ jeweils die Zwischenergebnisse für f_i aufnehmen d. h. seine *Zeitschrittdaten*. Zu Beginn eines Zeitschrittes von t nach $t + \Delta t$ enthalten die Speicherstellen s_i für jeden Gitterpunkt jeweils

die Partikeldichten $f_i(\vec{x}, t)$. Nach Anwendung der Lattice-Boltzmann-Gleichung enthalten die Speicherstellen s_i^* für jeden Gitterpunkt jeweils die neuberechneten Partikeldichten $f_i(\vec{x}, t + \Delta t)$. Der Zeitschritt wird abgeschlossen, indem für jeden Gitterpunkt die Inhalte von s_i und s_i^* vertauscht werden, sodass seine Zeitschrittdatenstruktur für den nächsten Zeitschritt (von $t + \Delta t$ nach $t + 2\Delta t$) wieder die Ausgangsbelegung annimmt [LBH+16].

Die Anwendung der Lattice-Boltzmann-Gleichung wird in zwei Teilberechnungen zerlegt [LBH+16]:

Kollisionsrechnung Hier werden den Speicherstellen s_i die Partikeldichten $f_i(\vec{x}, t)$ entnommen und die kollidierten Partikeldichten $f_i^*(\vec{x}, t)$ berechnet:

$$f_i^*(\vec{x}, t) = f_i(\vec{x}, t) + \sum_k L_{ik}(f_k^{eq} - f_k) + \Psi_i$$

Die berechneten kollidierten Partikeldichten $f_i^*(\vec{x})$ werden wieder in den Speicherstellen s_i abgelegt:

$$s_i(\vec{x}) := f_i^*(\vec{x}, t)$$

Strömen Hier werden den Speicherstellen s_i die kollidierten Partikeldichten $f_i^*(\vec{x}, t)$ entnommen und in die Speicherstellen $s_i^*(\vec{x}_i)$ der Nachbargitterpunkte $\vec{x}_i \in (\vec{x} + \Delta t \vec{v}_i)$ geschrieben:

$$s_i^*(\vec{x}_i) := f_i(\vec{x} + \vec{v}_i \Delta t, t + \Delta t) = f_i^*(\vec{x}, t)$$

Um eine parallele Verarbeitung zu ermöglichen, wird das Rechengitter üblicherweise in mehrere zusammenhängende Gitterabschnitte zerlegt. Jeder Gitterabschnitt wird einem Rechenknoten zugeteilt, welcher die Lattice-Boltzmann-Gleichung auf alle Gitterpunkte seines Gitterabschnittes anwendet [LBH+16].

Jeder Gitterabschnitt ist von einer Schicht zusätzlicher Gitterpunkte umgeben, der Ghostschicht. Die Gitterpunkte der Ghostschicht vertreten die randständigen Gitterpunkte benachbarter Gitterabschnitte und enthalten Kopien ihrer Zeitschrittdaten.

Für jeden Gitterpunkt $\vec{x}$ eines Gitterabschnittes, liegt somit seine Nachbarschaft $(\vec{x} + \Delta t \vec{v}_i)$ und ihre Zeitschrittdaten vollständig auf dem Rechenknoten des Gitterabschnittes vor. Die Gitterpunkte der Ghostschicht eines Gitterabschnittes werden regelmäßig durch den *Ghost-Austausch* aktualisiert, der in jeden Ghost-Gitterpunkt die Zeitschrittdaten seines entsprechenden randständigen Gitterpunktes aus dem entsprechenden benachbarten Gitterabschnitt hineinkopiert [LBH+16].

Dieses Replikationsschema erfordert eine Sonderbehandlung der Fälle, in denen für die diskrete Geschwindigkeit $\vec{v}_i$ die kollidierte Partikeldichte $f_i^*(\vec{x}, t)$ aus einem Gitterpunkt $\vec{x}$ des Gitterabschnitts in einen benachbarten Gitterpunkt $\vec{x}_i$ geströmt werden soll, der in der Ghostschicht liegt. Für diese Fälle wird stattdessen ein *inverses Strömen* ausgeführt. Dabei wird für die inverse diskrete Geschwindigkeit $\vec{v}_j = -\vec{v}_i$ die kollidierte Partikeldichte $f_j^*(\vec{x}_i, t)$ aus dem Ghost-Gitterpunkt x_i in den Gitterpunkt $\vec{x}$ geströmt. Es ist leicht einzusehen, dass das inverse

Strömen *aus* der Ghostschicht des einen Gitterabschnitts dem Strömen *in* die Ghostschicht des benachbarten Gitterabschnitts entspricht und umgekehrt [LBH+16].

Um keine erneute Kollisionsrechnung auf Ghost-Gitterpunkten ausführen zu müssen, verschiebt man den Ghost-Austausch hinter die Kollisionsrechnung: auf jedem Gitterabschnitt wird die Kollisionsrechnung durchgeführt, welche für jeden Gitterpunkt $\vec{x}$ dessen Speicherstellen s_i mit den kollidierten Partikeldichten $f_i^*(\vec{x}, t)$ belegt. Der Ghost-Austausch wird direkt im Anschluss vorgenommen und verteilt so die kollidierten Partikkeldichten auf alle Ghost-Gitterpunkte, wo sie dann für das inverse Strömen zur Verfügung stehen [Esc16].

2.2 Rechengitter

Ein Rechengitter für den Lattice-Boltzmann-Algorithmus ergibt sich aus dem geometrischen Gitter, welches das zu simulierende Fluidvolumen abdeckt. Ein geometrisches Gitter zerlegt ein Volumen überlappungsfrei in geometrische *Gitterzellen*, die definiert werden durch geometrische Gitterpunkte, die jeweils durch geometrische Gitterlinien verbunden sind.

Die im Rahmen dieser Arbeit behandelte Lattice-Boltzmann-Implementierung verfolgt eine volumetrische Formulierung der Lattice-Boltzmann-Methode d. h. sie betrachtet die Ausprägung der Partikelverteilung auf einem Rechengitter diskreter Volumina $\Delta^3 x$ und schreibt diese in diskreten Zeitschritten fort. Eine diskrete Partikeldichte $f_i(\vec{x}, t)$ umfasst also die Partikeldichte des Volumens $\Delta^3 x$ der *geometrischen Gitterzelle* an Position $\vec{x}$ für die diskrete Geschwindigkeit $\vec{v}_i$ zum Zeitpunkt t. Die Gitter*punkte* des Rechengitters entsprechen also den Gitter*zellen* des geometrischen Gitters [LBH+16] [RKDA06].

Desweiteren beschränkt sich diese Arbeit auf die Betrachtung von Gittern die dreidimensionale Würfelvolumina abdecken.

2.2.1 Reguläre kartesische Gitter

Ein reguläres kartesisches Gitter unterteilt ein Volumen vollständig in achsenparallele rechtwinklige Gitterzellen, deren Kantenlängen für alle Koordinatenachsen gleich sind. Im zweidimensionalen Fall sind diese Gitterzellen Quadrate, im dreidimensionalen Fall Würfel.

Struktur und Nachbarschaft eines regulären kartesischen Gitters sind implizit. Für eine Gitterzelle muss weder ihre Position gespeichert werden, noch welche Gitterzellen ihre jeweiligen Nachbarn sind.

Für ein reguläres kartesisches Gitter mit den Zellanzahlen (S_x, S_y, S_z) auf den drei Koordinatenachsen ergibt sich die Speicherposition $m_{\vec{x}}$ der Zeitschrittdaten einer Gitterzelle an Position $\vec{x} = (x, y, z)$ als

$$m_{\vec{x}} = x + yS_x + zS_xS_y$$

Die Speicherpositionen der Zeitschrittdaten ihrer Nachbargitterzellen $\vec{x} + \delta_{offset}$ mit $\delta_{offset} = (x_{offset}, y_{offset}, z_{offset})$ sind entsprechend

$$m_{\vec{x}+\delta_{offset}} = m_{\vec{x}} + x_{offset} + y_{offset} S_x + z_{offset} S_x S_y$$

Ebenso einfach wie die Addressierung gestaltet sich die Aufteilung eines regulären kartesischen Gitters auf mehrere Rechenknoten. Dazu wird das Gitter in rechteckige Gitterabschnitte zerlegt, indem seine Zellanzahlen (S_x, S_y, S_z) auf jeder Koordinatenachse in Intervalle $S_{x,i}, S_{y,j}, S_{z,k}$ zerlegt wird, über die dann Untergittervolumina $\forall i, j, k : S_{x,i} S_{y,j} S_{z,k}$ aufgespannt werden. Diese Untergittervolumina werden nun den einzelnen Rechenknoten zugewiesen.

2.2.2 Baumstrukturierte Gitter mit p4est

Ein baumstrukturiertes Gitter wendet ein rekursives Unterteilungsschema an, welches ein Volumen vollständig in eine Anzahl Gitterzellen unterteilt, die bereichsweise verschiedene Größen aufweisen können.

Das Unterteilungsschema sieht vor, dass jedes zu unterteilende d-dimensionale rechtwinklige Volumen mit gleichen Seitenlängen (beginnend mit dem Ausgangsvolumen) in 2^d rechtwinklige Untervolumina halber Seitenlänge unterteilt werden kann.

Im zweidimensionalen Fall wird ein quadratisches Ausgangsvolumen in Quadranten unterteilt, im dreidimensionalen Fall ein würfelförmiges Ausgangsvolumen in Oktanten. Jedes aus einer Unterteilung hervorgegangene Untervolumen kann unabhängig von den anderen Untervolumina nach demselben Unterteilungsschema weiter unterteilt werden, wodurch im Gitter Bereiche unterschiedlicher Zellengrößen entstehen können. Ein Volumen, das aus l rekursiven Unterteilungen hervorgegangen ist, hat entsprechend den Verfeinerungslevel l.

Die Baumstruktur ergibt sich, indem das Ausgangsvolumen als die Baumwurzel angesehen wird, wodurch die unterteilten Volumina den inneren Baumknoten und die nicht mehr weiter unterteilten Volumina den Blattknoten entsprechen. Die Gitterzellen entsprechen den Volumina der Blattknoten.

Anstelle einer expliziten Baumstruktur speichert p4est eine Liste aller Gitterzellen bzw. Blattknoten in Z-Ordnung d. h. entlang der raumfüllenden Z-Kurve. Im Prinzip genügte es für jede Gitterzelle in dieser Liste nur den Verfeinerungslevel l zu speichern. Raumposition und Zellgröße ließen sich für jede Gitterzelle errechnen aus ihrer Listenposition und den Verfeinerungsleveln aller Gitterzellen, die in der Liste vor ihr kommen. Ebenso könnten die Nachbarn einer Gitterzelle errechnet werden aus ihrer Listenposition und den Verfeinerungsleveln einer Anzahl von Gitterzellen, die in der Liste vor und nach ihr stehen.

In der Praxis speichert p4est für jede Gitterzelle die räumliche Position explizit. Ebenso stellt p4est die Möglichkeit bereit, in Rahmen eines zusätzlichen Gitterdurchlaufs eine Mesh-Datenstruktur zu erzeugen, die Nachbarschaftsverhältnisse explizit speichert und so im folgenden eine Nachbarschaftssuche in $O(1)$ erlaubt.

Da die Liste der Gitterzellen eine raumfüllende Z-Kurve beschreibt, entspricht jede beliebige Teilsequenz dieser Liste stets einem räumlich zusammenhängenden Gitterabschnitt. Ein baumstrukturiertes kartesisches Gitter lässt sich daher in trivialer Weise auf mehrere Rechenknoten verteilen, indem man diese Liste teilt und jedem Rechenknoten einen der Listenabschnitte zuweist.

2.3 Das generische Gitterschnittstellensystem GGI

In [Esc16] wurde mit der Entwicklung des generische Gitterschnittstellensystems GGI begonnen. GGI definiert eine generischen Gitterschnittstelle, die den Zugriff einer Anwendung auf reguläre kartesische Gitter sowie baumstrukturierte kartesische Gitter vereinheitlicht, und stellt je eine Implementierung bereit, die ein reguläres kartesische Gitter bzw. ein baumstrukturiertes kartesisches Gitter errichten, über eine Anzahl an Anwendungsprozessen verteilen und verwalten.

Eine Anwendung, die ein GGI-Gitter nutzt, kann das Gitter so einrichten, dass es für jeden Gitterpunkt eine Instanz einer anwendungsspezifischen Datenstruktur hält. GGI handhabt die Details des Transfers und der Replikation dieser anwendungsspezifischen Daten im Zuge von Gitter-Neupartitionierungen und der Bereitstellung von Ghostschichten weitgehend transparent.

Die generische Gitterschnittstelle stellt das GGI-Gitter gegenüber der Anwendung als ein *logisches* adaptives kartesisches Gitter dar, das eine maximale Verfeinerungszahl L aufweist. Die maximale Verfeinerungszahl L gibt an, wie oft ein Gitterpunkt verfeinert werden kann i. d. S. dass ein Gitterpunkt l-mal verfeinert wird, indem er einmal verfeinert wird und die dadurch aus ihm neu entstandenen Tochtergitterpunkte jeweils $(l-1)$-mal verfeinert werden.

Es ist dieses Gittermodell, das es der Schnittstelle erlaubt, sowohl verfeinerbare wie nichtverfeinerbare kartesische Gitter abzubilden. Das logische kartesische Gitter, das die generische Gitterschnittstelle gegenüber der Anwendung darstellt, wenn sie durch eine reguläre kartesische Gitterimplementierung unterlegt ist, hat folgerichtig die maximale Verfeinerungszahl $L = 0$.

Die generische Gitterschnittstelle gliedert das logische adaptive kartesische Gitter in verschiedene Gitterpunktmengen. Insbesondere verteilt die generische Gitterschnittstelle die Gitterpunkte nach ihrem jeweiligen Verfeinerungslevel l auf separate Gitterpunktmengen G_l, den *Levelpunktmengen*. Aus diesen Levelpunktmengen G_l und aus den Hilfsfunktionalitäten, die benötigt werden, wenn die Anwendung, wie im Rahmen dieser Arbeit, eine Lattice-Boltzmann-Implementierung ist, ergeben sich einige weitere Gitterpunktmengen, die Varianten oder Teilmengen der Levelpunktmengen sind.

Die generische Gitterschnittstelle definiert eine Reihe von Verarbeitungskontexten, über welche die Anwendung auf das logische adaptive kartesische Gitter zugreifen kann. Dabei ist

zu unterscheiden zwischen Verarbeitungskontexten, die sich auf das Gesamtgitter beziehen, und solchen, deren Bezug sich sich auf einzelne Gitterpunkte bzw. Gitterpunktmengen beschränkt.

Die gitterpunktbezogenen Verarbeitungskontexte folgen bislang alle dem *Iteratormodell*: Ein Iterator durchläuft eine einzelne Gitterpunktmenge punktweise, wobei er für den jeweils aktuellen Gitterpunkt Kontextinformationen und -funktionen bereitstellt, über die u. A. auf die Zeitschrittdaten und Nachbarschaft des Gitterpunktes zugegriffen werden kann. Für den Zugriff auf die Nachbarschaft eines Gitterpunktes definiert die generische Gitterschnittstelle ein Richtungsmodell, das sämtliche Raumrichtungen, in denen in einem kartesischen Gitter direkte Nachbargitterpunkte stehen können, auf eine Indexnummer $d \in [0, 26]$ abbildet.

Die gesamtgitterbezogenen Verarbeitungskontexte folgen einem Modell, das die Betriebsphasen des Gitters abbildet, die zusammen den *Gitterzyklus* bilden. Dabei definiert die generische Gitterschnittstelle Verarbeitungskontexte für vier Gitterphasen: der *Gittererrichtung*, des *Gitterdurchlaufs*, des *Ghost-Austauschs* und der *Gitteranpassung*. Jede Gitterphasen stellt über ihren Verarbeitungskontext relevante Funktionen und (gitterpunktbezogene) Verarbeitungskontexte bereit und muss explizit begonnen und beendet werden, da sich das Gitter stets nur in jeweils einer Betriebsphase befinden kann. Derzeit werden sämtliche gitterpunktbezogenen Verarbeitungskontexte durch den Verarbeitungskontext des Gitterdurchlaufs bereitgestellt.

Das generische Gitterschnittstellensystem GGI in seinem Abschlussstand in [Esc16], genügt den Anforderungen einer adaptiven Lattice-Boltzmann-Implementierung als Anwendung stellenweise noch nicht. Daher wird es im Rahmen dieser Arbeit überarbeitet: die generische Gitterschnitstelle wird präzisiert und die Schnittstellenimplementierung für baumstrukturierte kartesische Gitter wird vervollständigt.

2.4 Entkoppelte Lattice-Boltzmann-Implementierung

Ebenfalls in [Esc16] wurde von der ursprünglichen Lattice-Boltzmann-Implementierung von ESPResSo eine entkoppelte Lattice-Boltzmann-Implementierung abgeleitet, welche auf einem regulären kartesischen Gitter arbeitet, das durch GGI bereitgestellt wird.

Die entkoppelte Lattice-Boltzmann-Implementierung ist jedoch noch nicht adaptiv und nutzt daher somit noch nicht die entsprechenden Möglichkeiten von GGI. Daher wird sie im Rahmen dieser Arbeit zu einer *adaptiven Lattice-Boltzmann-Implementierung* weiterentwickelt und in ESPResSo integriert.

3 Adaptive Lattice-Boltzmann-Implementierung

Die adaptive Lattice-Boltzmann-Implementierung dieser Arbeit entstand, indem die entkoppelte Lattice-Boltzmann-Implementierung aus [Esc16] auf das in [RKDA06] beschriebene Berechnungsschema hin weiterentwickelt wurde.

Der Kollisionsoperator hingegen wurde einer anderen adaptiven jedoch nicht entkoppelten Lattice-Boltzmann-Implementierung entnommen, die im Rahmen von [LABM16] entwickelt worden ist.

Die so entstandene adaptive Lattice-Boltzmann-Implementierung basiert demnach auf Subcycling bzw. Supercycling [RKDA06]:
Lattice-Boltzmann-Zeitschritte werden nicht mehr auf dem Gesamtgitter ausgeführt.
Stattdessen werden für alle im Gitter auftretenden Verfeinerungslevel l die sich jeweils ergebenden Gitterabschnitte gleicher Verfeinerung l voneinander getrennt durchlaufen. Das Strömen zwischen Gitterabschnitten unterschiedlicher Verfeinerung wird über Schichten virtueller Gitterpunkte vermittelt, welche die Gitterabschnitte höherer Verfeinerung umgeben [RKDA06].

Für die Umsetzung dieses Berechnungsschemas standen zwei Varianten zur Auswahl:

Subcycling In der Subcycling-Variante bezieht sich die eingestellte Lattice-Boltzmann-Zeitschrittlänge τ auf den Verfeinerungslevel 0.

Eine solche Implementierung führt also auf Gitterabschnitten mit Verfeinerungslevel 0 Zeitschritte der Länge τ aus und auf Gitterabschnitten mit Verfeinerungslevel $l > 0$ 2^l Unterzeitschritte der Länge $2^{-l}\tau$.

Supercycling In der Supercycling-Variante bezieht sich die eingestellte Lattice-Boltzmann-Zeitschrittlänge τ auf die maximale Verfeinerungskapazität L.

Eine solche Implementierung führt also auf Gitterabschnitten mit Verfeinerungslevel L Zeitschritte der Länge τ und auf Gitterabschnitten mit Verfeinerungslevel $l < L$ Superzeitschritte der Länge $2^{L-l}\tau$ aus.

Die Lattice-Boltzmann-Implementierung wurde im Rahmen dieser Arbeit zunächst in der Subcycling-Variante erstellt. Unter der Annahme, dass diese sich besser für die zukünftige Kopplung an die ESPResSo-Partikelsimulation eignet, wurde sie dann auf die Supercycling-Variante umgestellt.

3.1 Teilberechnungen

Die entkoppelte Lattice-Boltzmann-Implementierung aus [Esc16] setzte einen Lattice-Boltzmann-Zeitschritt aus drei Teilberechnungen zusammen: *Kollisionsrechnung*, *Ghost-Austausch* und *Strömen*. Diese arbeiteten auf der Zeitschrittdatenstruktur der Gitterpunkte mit ihren Speicherstellen (s_i, s_i^*), berechneten aus den Ausgangspopulationen $f_i(\vec{x})$ die kollidierten Populationen $f_i^*(\vec{x})$ und strömten diese in die jeweiligen Nachbargitterpunkte.

Die adaptive Lattice-Boltzmann-Implementierung hingegen definiert sechs Teilberechnungen, die jeweils für einen Verfeinerungslevel l ausgeführt werden:

l-Kollisionsrechnung Wird auf allen Gitterpunkten $\vec{x}$ mit Verfeinerungslevel l ausgeführt.

Nimmt auf jedem behandelten Gitterpunkt $\vec{x}$ die Kollisionsrechnung vor. Die Ausgangspopulationen $f_i(\vec{x})$ werden den Speicherstellen s_i entnommen, welche anschließend mit den kollidierten Ergebnispopulationen $f_i^*(\vec{x})$ überschrieben werden.

l-Ghost-Austausch Wird auf allen Ghost-Gitterpunkten mit Verfeinerungslevel l ausgeführt.

Befüllt auf jedem Prozess die Zeitschrittdatenstrukturen der behandelten Ghost-Gitterpunkte mit den Zeitschrittdaten der entsprechenden randständigen Gitterpunkte benachbarter Gitterabschnitte. Die Zeitschrittdaten von virtuellen Gitterpunkten mit Verfeinerungslevel l werden nicht aktualisiert.

l-Host-Expansion Wird auf allen Gitterpunkten $\vec{x}$ (einschließlich Ghost-Gitterpunkten) mit Verfeinerungslevel l ausgeführt, die virtuelle Gitterpunkte $\vec{x}_j$ mit Verfeinerungslevel $(l+1)$ einbetten.

Kopiert die kollidierten Populationen $f_i^*(\vec{x})$ aus den Speicherstellen s_i jedes behandelten Gitterpunktes $\vec{x}$ in die Speicherstellen $s_{i,j}$ und $s_{i,j}^*$ seiner eingebetteten virtuellen Gitterpunkte $\vec{x}_j$.

l-Host-Reduktion Wird auf allen Gitterpunkten $\vec{x}$ (ausschließlich Ghost-Gitterpunkten) mit Verfeinerungslevel l ausgeführt, die virtuelle Gitterpunkte $\vec{x}_j$ mit Verfeinerungslevel $(l+1)$ einbetten.

Mittelt für jeden behandelten Gitterpunkt $\vec{x}$ und für jede Richtung i die Populationen in den Speicherstellen $s_{i,j}^*$ seiner virtuellen Gitterpunkte $\vec{x}_j$ und legt die jeweiligen Ergebnisse in die Speicherstellen s_i^* des behandelten Gitterpunktes ab.

l-Strömen Wird auf allen Gitterpunkten $\vec{x}$ (einschließlich virtueller Gitterpunkte) mit Verfeinerungslevel l ausgeführt.

Kopiert die kollidierten Populationen $f_i^*(\vec{x})$ aus den Speicherstellen s_i des behandelten Gitterpunktes in die Speicherstellen s_i^* seiner Nachbargitterpunkte $\vec{x}_i$. Im Falle von Nachbargitterpunkten, die Ghost-Gitterpunkte sind, wird das inverse Strömen ausgeführt.

l-Ghost-Austausch für virtuelle Gitterpunkte Wird auf allen virtuellen Gitterpunkten mit Verfeinerungslevel l ausgeführt, die durch Gitterpunkte mit Verfeinerungslevel $(l-1)$ in der Ghostschicht eingebettet werden.

Befüllt auf jedem Prozess die Zeitschrittdatenstrukturen der behandelten virtuellen Gitterpunkte mit den Zeitschrittdaten der virtuellen Gitterpunkte, die durch die entsprechenden randständigen Gitterpunkte benachbarter Gitterabschnitte eingebettet werden.

3.2 Halbschritte

Die adaptive Lattice-Boltzmann-Implementierung fasst die Teilberechnungen in zwei *Halbschritte* zusammen, die l-Eröffnung und den l-Abschluss, die jeweils den Beginn und die Fertigstellung eines (Super-)Zeitschrittes der Länge $2^{L-l}\tau$ auf einem Verfeinerungslevel l realisieren.

l-Eröffnung Ein l-Superzeitschritt auf einem Gitterabschnitt mit Verfeinerungslevel l eröffnet, indem er die l-Kollisionsrechnung, der l-Ghost-Austausch und die l-Host-Expansion in Reihe ausführt.

Die Kollisionsrechnung berechnet die kollidierten Populationen, der Ghost-Austausch macht sie prozessübergreifend für den behandelten Gitterabschnitt verfügbar und die Host-Expansion stellt sie für das Einströmen in angrenzende Gitterabschnitte mit Verfeinerungslevel $(l+1)$ im Rahmen zweier $(l+1)$-Superzeitschritte bereit.

Indem die Host-Expansion auch auf Ghost-Gitterpunkten ausgeführt wird, kann beim davor stattfindenden Ghost-Austausch die Aktualisierung der virtuellen Ghost-Gitterpunkte entfallen und so Kommunikationsvolumen eingespart werden.

l-Abschluss Ein l-Superzeitschritt auf einem Gitterabschnitt mit Verfeinerungslevel l schließt ab, indem er die l-Host-Reduktion, das l-Strömen und der l-Ghost-Austausch für virtuelle Gitterpunkte in Reihe ausführt.

Die Host-Reduktion schließt das Ausströmen von im Rahmen zweier $(l+1)$-Super-Zeitschritte kollidierten Populationen aus angrenzenden Gitterabschnitten mit Verfeinerungslevel $(l+1)$ in den behandelten Gitterabschnitt hinein ab.

Das Strömen setzt das Einströmen kollidierter Populationen aus angrenzenden Gitterabschnitten mit Verfeinerungslevel $(l-1)$ in den behandelten Gitterabschnitt hinein um, strömt innerhalb des behandelten Gitterabschnittes und stellt die Populationen bereit, die ausströmen in angrenzende Gitterabschnitte mit Verfeinerungslevel $(l-1)$.

Das Supercycling ergibt sich, indem in jedem Lattice-Boltzmann-Zeitschritt τ:

- für zunehmende Verfeinerungslevel l je die l-Eröffnung vorgenommen wird, sofern zuvor ein l-Abschluss erfolgt ist, und

- für abnehmende Verfeinerungslevel l je der l-Abschluss vorgenommen wird, sofern zuvor zwei $(l+1)$-Abschlüsse erfolgt sind

Somit wird zunächst jeder l-Superzeitschritt eröffnet und erst dann abgeschlossen, sobald zwei $(l+1)$-Superzeitschritte abgeschlossen worden sind.

Dieses Berechnungsschema wurde der Lattice-Boltzmann-Implementierung aus [LABM16] entnommen und im Rahmen dieser Arbeit stellenweise optimiert: Zum einen durch die Ausdehnung der l-Host-Expansion auf die Gitterpunkte in der Ghostschicht. Zum anderen nimmt der l-Abschluss im Gegensatz zu [LABM16] seinen Ghost-Austausch nur auf virtuellen Gitterpunkten vor und auch nur dann, wenn der l-Abschluss in der Mitte eines $(l-1)$-Superzeitschrittes liegt.

3.3 Boundaries

Die adaptive Lattice-Boltzmann-Implementierung übernimmt die No-Slip-Boundaries aus [LAMH06]. Diese Boundaries können verschiedene Geometrien aufweisen wie z. B. Ebenen, Kugeln usw.. Allen gemeinsam ist, dass sie ein Volumen *teilen* in ein Teilvolumen, das durch die Boundary verdeckt wird und in welches das simulierte Fluid weder ein- noch ausströmen kann, und in ein Restvolumen, in dem das Fluid ungehindert strömen kann.

Die Geometrien dieser Boundaries müssen diskretisiert d. h. auf das Gitter abgebildet werden. Hierfür wird für jeden Gitterpunkt seine Position relativ zu den Grenzen aller geometrischen Boundaries ermittelt. Daraus ergibt sich, dass die durch einen Gitterpunkt abgedeckte geometrische Gitterzelle entweder

1. vollständig durch eine oder mehrere Boundaries verdeckt wird oder
2. durch die Grenze wenigstens einer Boundary geschnitten wird oder
3. vollständig außerhalb aller durch Boundaries verdeckten Volumina liegen kann.

In den ersten beiden Fällen wird der betroffene Gitterpunkt der Boundary, deren Grenze jeweils am nächsten liegt, zugeordnet.

Im zweiten Fall wird außerdem der Gitterpunkt verfeinert. Für die so entstandenen Tochtergitterpunkte wird erneut die jeweilige Position relativ zu den Grenzen aller geometrischen Boundaries ermittelt. Tochtergitterpunkte, für die der zweite Fall dann immer noch eintritt, werden weiter verfeinert. Dies wird solange rekursiv fortgeführt, bis der maximale Verfeinerungslevel L erreicht ist.

Dadurch werden Boundary-Grenzen stets mit höchstmöglicher Auflösung diskretisiert. Dies ist in Abbildung 3.1 beispielhaft dargestellt.

Gitterpunkte, die Boundaries zugeordnet sind, werden bei der Berechnung von Lattice-Boltzmann-Zeitschritten übergangen und, wenn sie beim Strömen als Nachbargitterpunkte

auftreten, dem *Bounce-Back* unterzogen d. h. was aus einer Richtung d in sie einzuströmen versucht, wird für die Gegenrichtung $-d$ wieder in den jeweiligen Ausgangsgitterpunkt zurückgeschrieben [LAMH06].

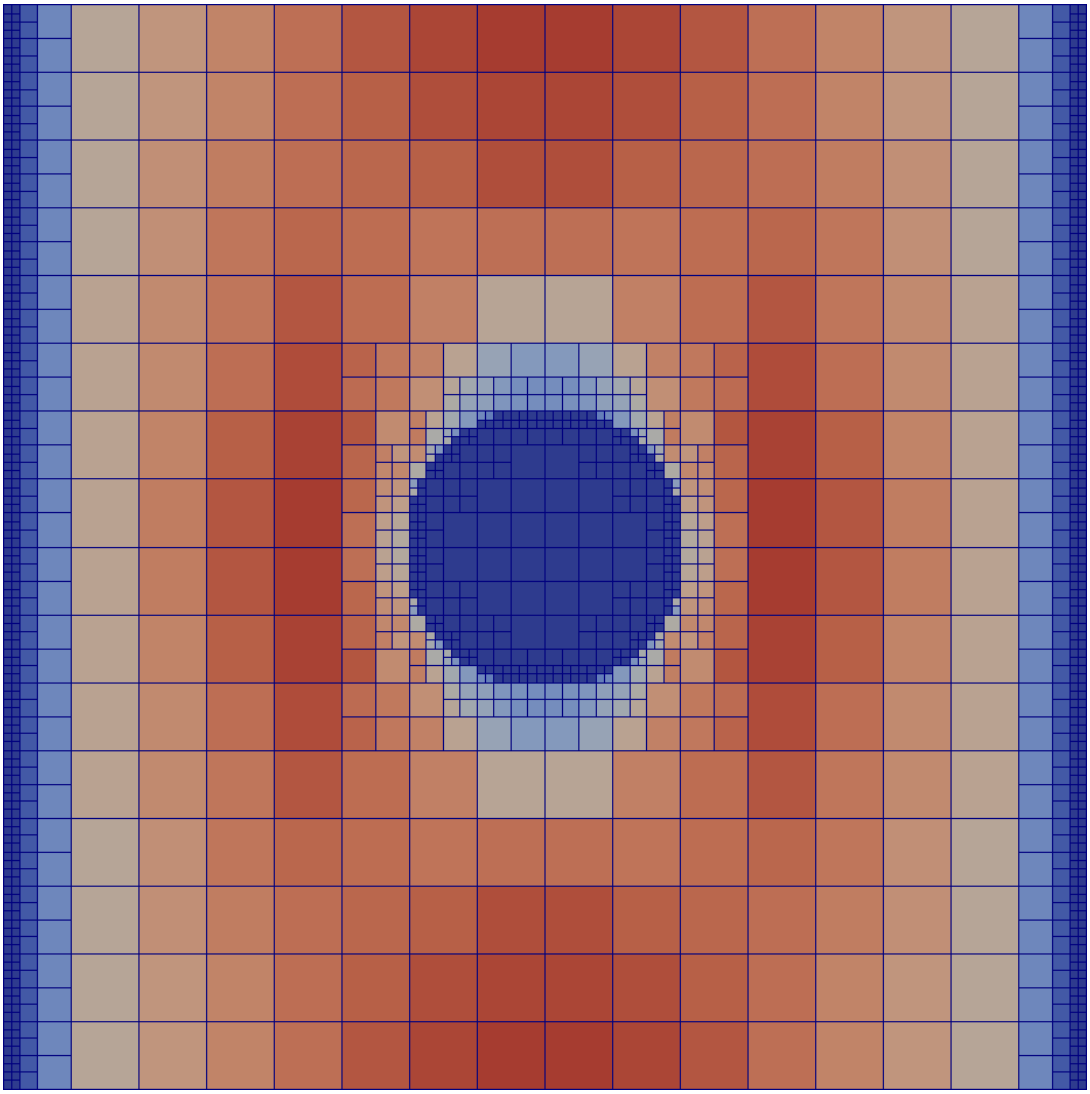

Abbildung 3.1: Schnittdarstellung einer Poiseuille-Strömung mit zusätzlicher Kugel-Boundary

4 Überarbeitung des generischen Gitterschnittstellensystems GGI

Das generische Gitterschnittstellensystem GGI umfasst neben der generischen Gitterschnittstelle zwei Schnittstellenimplementierungen, die als C++ Templateklassen ausgeführt sind: `ggi::RegularGrid`, die ein reguläres GGI-Gitter implementiert, und `ggi::P4estGrid`, die ein baumstrukturiertes GGI-Gitter implementiert [Esc16].

Der generischen Gitterschnittstelle selbst entspricht dabei keine C++ Klasse, da sie aus Performancegründen nie als Laufzeitinterface (im Sinne einer gemeinsamen Basisklasse für `ggi::RegularGrid` und `ggi::P4estGrid` mit virtuellen Methoden) vorgesehen war.

Stattdessen wird sie von `ggi::RegularGrid` und `ggi::P4estGrid` dadurch ausgeprägt, indem diese sämtliche durch die generische Gitterschnittstelle geforderte Funktionalität über gleichbenannte und -strukturierte Methoden und Datentypen bereitstellen, deren Funktionsweise und Semantik sich ebenfalls gleichen und welche die Vorgaben der generischen Gitterschnittstelle sinngemäß auf das jeweils errichtete GGI-Gitter anwenden.
So implementiert `ggi::RegularGrid` z. B. die gleichen Funktionen zur Gitterverfeinerung wie `ggi::P4estGrid`, nur eben als Stubs. Dadurch ist sichergestellt, dass zur Compilezeit einer Anwendung die zu benutzende GGI-Schnittstellenimplementierung gegen eine andere ausgetauscht werden kann und der Anwendungscode dennoch fehlerfrei kompilierbar bleibt [Esc16].

Eine solche Anwendung mit ein oder mehreren Prozessen (P_i), wie z. B. eine parallele Lattice-Boltzmann-Implementierung, nutzt ein GGI-Gitter, indem jeder Anwendungsprozess P_i eine GGI-Schnittstelleninstanz I_i von `ggi::RegularGrid` bzw. `ggi::P4estGrid` erzeugt (je nachdem welche Schnittstellenimplementierung das GGI-Gitter bereitstellen soll) und diese Schnittstelleninstanz mit einem MPI-Kommunikator initialisiert, der alle Anwendungsprozesse (P_i) umfasst.
Der Masterprozess P_0 der Anwendung stellt an seiner GGI-Schnittstelleninstanz Gitterparameter wie gewünschte Größe und geforderten Funktionsumfang ein.
Alle Anwendungsprozesse (P_i) lösen an ihren jeweiligen GGI-Schnittstelleninstanzen I_i die Gittererrichtung aus, worauf diese sich zu dem verteilten GGI-Gitter zusammenfügen.

Mit Abschluss der Gittererrichtung hält jede GGI-Schnittstelleninstanz I_i einen Abschnitt des verteilten GGI-Gitters. Jeder Anwendungsprozess P_i kann nun über seine Schnittstelleninstanz

I_i sowohl auf „seinen“ Gitterabschnitt zugreifen als auch an kollektiven Operationen auf dem verteilten Gesamtgitter teilnehmen [Esc16].

Die in [Esc16] begonnene Schnittstellenimplementierung `ggi::P4estGrid` wurde im Rahmen dieser Arbeit vervollständigt.

Ebenso wurde die generische Gitterschnittstelle teilweise restrukturiert und präzisiert, um den Anforderungen einer adaptiven Lattice-Boltzmann-Implementierung vollumfänglich gerecht zu werden. Insbesondere wurden die bereitgestellten Gitterpunktmengen präzisiert, die Gitterphasen in ihrem Funktionsumfang erweitert und die Unterstützung für virtuelle Gitterpunkte als weitere Gesamtgitterfunktion hinzugefügt.

4.1 Gitterpunktmengen und -zugriff

Die generische Gitterschnittstelle definiert eine Reihe von Gitterpunktmengen, die Gitterpunkte anhand bestimmter Eigenschaften und Rollen zusammenfassen. Jeder Anwendungsprozess P_i erhält über seine GGI-Schnittstelleninstanz I_i Zugriff auf die Gitterpunkte dieser Gitterpunktmengen, die innerhalb seines Abschnitts des verteilten GGI-Gitters liegen. Liegen von einer bestimmten Gitterpunktmenge keinerlei Gitterpunkte innerhalb des Gitterabschnitts eines Anwendungsprozesses P_i, so gibt die GGI-Schnittstelleninstanz I_i eine leere Gitterpunktmenge zurück [Esc16].

Alle Gitterpunkte Die triviale Gitterpunktmenge umfasst alle Gitterpunkte des verteilten GGI-Gitters.

Hilfsgitterpunkte wie Ghost-Gitterpunkte und virtuelle Gitterpunkte sind in dieser Gitterpunktmenge nicht enthalten.

Die adaptive Lattice-Boltzmann-Implementierung verwendet diese Gitterpunktmenge für die Ausgabe von Ergebnissen je Gitterpunkt.

Gitterpunkte mit Verfeinerungslevel l Diese Gitterpunktmenge enthält ausschließlich Gitterpunkte mit Verfeinerungslevel l.

Die Gitterpunktmenge ist für jeden im verteilten GGI-Gitter möglichen Verfeinerungslevel definiert. Hilfsgitterpunkte wie Ghost-Gitterpunkte und virtuelle Gitterpunkte sind in dieser Gitterpunktmenge nicht enthalten.

Die adaptive Lattice-Boltzmann-Implementierung verwendet diese Gitterpunktmenge für die *l-Kollisionsrechnung* und das *l-Strömen*, beide in Abschnitt 3.1 beschrieben.

Virtuelle Gitterpunkte mit Verfeinerungslevel l Diese Gitterpunktmenge enthält ausschließlich *virtuelle* Gitterpunkte mit Verfeinerungslevel l.

Die Gitterpunktmenge ist für jeden im verteilten GGI-Gitter möglichen Verfeinerungslevel definiert. Ghost-Gitterpunkte sind in dieser Gitterpunktmenge nicht enthalten.

Die adaptive Lattice-Boltzmann-Implementierung verwendet diese Gitterpunktmenge für das in Abschnitt 3.1 beschriebene *l-Strömen*.

Hostgitterpunkte mit Verfeinerungslevel l Diese Gitterpunktmenge enthält ausschließlich die Gitterpunkte mit Verfeinerungslevel l, die virtuelle Gitterpunkte einbetten.

Die Gitterpunktmenge ist für jeden im verteilten GGI-Gitter möglichen Verfeinerungslevel definiert. Ghost-Gitterpunkte sind in dieser Gitterpunktmenge nicht enthalten. Virtuelle Gitterpunkte können selbst definitionsgemäß keine weiteren virtuellen Gitterpunkte einbetten und sind somit in dieser Gitterpunktmenge auch nicht enthalten.

Die adaptive Lattice-Boltzmann-Implementierung verwendet diese Gitterpunktmenge für die *l-Host-Expansion* und die *l-Host-Reduktion*, beide in Abschnitt 3.1 beschrieben.

Hostgitterpunkte mit Verfeinerungslevel l in der Ghostschicht Diese Gitterpunktmenge enthält ausschließlich die Gitterpunkte mit Verfeinerungslevel l, die virtuelle Gitterpunkte einbetten und die in der Ghostschicht des Gitterabschnitts liegen.

Die Gitterpunktmenge ist für jeden im verteilten GGI-Gitter möglichen Verfeinerungslevel definiert. Virtuelle Gitterpunkte können selbst definitionsgemäß keine weiteren virtuellen Gitterpunkte einbetten und sind somit in dieser Gitterpunktmenge auch nicht enthalten.

Die adaptive Lattice-Boltzmann-Implementierung verwendet diese Gitterpunktmenge für die in Abschnitt 3.1 beschriebene *l-Host-Expansion*.

4.1.1 Gitterpunktbezogene Verarbeitungskontexte

In [Esc16] fand der Zugriff auf die Gitterpunkte über die Punktmengeniteratoren statt. Im Rahmen dieser Arbeit wurde dies dahingehend überarbeitet, dass dieser Zugriff nun von separaten Verarbeitungskontexten vermittelt wird.

Punkt-Akzessor Wird ein Punktmengeniterator dereferenziert, liefert er den Punkt-Akzessor zurück. Dieser ist eine Instanz von `GridPoint`, einer inneren Klasse von `ggi::RegularGrid` bzw. `ggi::P4estGrid`.

Der Punkt-Akzessor stellt Kontextinformationen und verfügbare Kontextfunktionen für den Gitterpunkt im Zugriff zur Verfügung:

Ghost-, Host- und Virtual-Flags `isGhost()`, `isHost()` und `isVirtual()` zeigen an, ob der Gitterpunkt im Zugriff in der Ghostschicht liegt, virtuelle Gitterpunkte einbettet oder selbst ein virtueller Gitterpunkt ist.

Raumposition `domainPos()` liefert die Raumposition des Gitterpunktes im Zugriff innerhalb des durch das verteilte GGI-Gitter aufgespannten Raumvolumens.

Verfeinerungslevel und Seitenlänge `level()` und `length()` liefern den Verfeinerungslevel l und die Seitenlänge h des Gitterpunktes im Zugriff.

Für einen Gitterpunkt, der einen Verfeinerungslevel l aufweist, ergibt sich die Seitenlänge immer als $h = 2^{-l}$.

Interpretiert eine Anwendung Gitterpunkte als Gitterzellen d. h. Volumina, entspräche die Raumposition der linken untern Ecke einer Gitterzelle mit der obigen Seitenlänge.
Andernfalls gibt die Seitenlänge lediglich den Abstand zu den nächstliegenden Gitterpunkten an.

Anwendungsdaten Der Zugriff auf die mit dem Gitterpunkt im Zugriff assoziierten Anwendungsdaten ist sowohl explizit mit `data()` als auch über überladene C++ Dereferenzierungsoperatoren möglich.

Verfeinerungsmarkierung und -Flag Mit `refine()` kann die Anwendung festlegen ob und wie oft der Gitterpunkt im Zugriff beim nächsten Durchlaufen der Gitteranpassung verfeinert werden soll. Ebenso kann mit `isToBeRefined()` abgefragt werden, ob der Gitterpunkt im Zugriff zur Verfeinerung markiert ist.

Liegt der Gitterpunkt im Zugriff in der Ghostschicht oder ist er ein virtueller Punkt, werden Anforderungen zu seiner Verfeinerung ignoriert.

Vergröberungsmarkierung und -Flag Mit `coarsen()` kann die Anwendung festlegen ob und wie oft der Gitterpunkt im Zugriff beim nächsten Durchlaufen der Gitteranpassung vergröbert werden soll. Ebenso kann mit `isToBeCoarsened()` abgefragt werden, ob der Gitterpunkt im Zugriff zur Vergröberung markiert ist.

Liegt der Gitterpunkt im Zugriff in der Ghostschicht oder ist er ein virtueller Punkt, werden Anforderungen zu seiner Vergröberung ignoriert.

Verfeinerungsleveldelta Das Verfeinerungsleveldelta ist die angestrebte Änderung des Verfeinerungslevels l des Gitterpunktes im Zugriff infolge der angeforderten Verfeinerungen bzw. Vergröberungen, die im nächsten Durchlauf der Gitteranpassung vollzogen werden sollen.

Das Verfeinerungsleveldelta kann mit `getLevelDelta()` abgefragt und mit `resetLevelDelta()` explizit gesetzt werden. Wie das Verfeinern und Vergröbern des GGI-Gitters im Einzelnen funktioniert, ist in Abschnitt 4.3.3 beschrieben.

Liegt der Gitterpunkt im Zugriff in der Ghostschicht oder ist er ein virtueller Punkt, werden Änderungen an seinem Verfeinerungsleveldelta ignoriert.

Verfeinerungsleveldelta für die automatische Gittervergröberung Das Verfeinerungsleveldelta für die automatische Gittervergröberung ist die Änderung des Verfeinerungslevels l des Gitterpunktes im Zugriff, das nicht durch die Anwendung sondern durch die automatische Gittervergröberung angestrebt wird.

Das Verfeinerungsleveldelta für die automatische Gittervergröberung kann mit `getAutoCoarsenLevelDelta()` abgefragt und mit `resetAutoCoarsenLevelDelta()` zurückgesetzt werden. Wie das automatische Vergröbern des GGI-Gitters im Einzelnen funktioniert, ist in Abschnitt 4.3.4 beschrieben.

Liegt der Gitterpunkt im Zugriff in der Ghostschicht oder ist er ein virtueller Punkt, wird die Zurücksetzung seines Verfeinerungsleveldelta für die automatische Gittervergröberung ignoriert.

Anwendungsdaten und Raumpositionen eingebetteter virtueller Gitterpunkte Bettet der Gitterpunkt im Zugriff virtuelle Gitterpunkte ein, kann über `virtualData()` auf deren assoziierte Anwendungsdaten zugegriffen und über `virtualDomainPos()` deren Raumposition abgefragt werden.

Die Nummerierung der eingebetteten virtuellen Gitterpunkte ist in Z-Ordnung.

Nachbarschaft `neighbourhood()` liefert den Nachbarschafts-Akzessor zurück, der weiter unten beschrieben ist.

Nachbarschafts-Akzessor Der Nachbarschafts-Akzessor ist eine Instanz von `Neighbourhood`, einer inneren Klasse von `ggi::RegularGrid` bzw. `ggi::P4estGrid`.

Er implementiert die Nachbarschaftssuche und stellt analog zum Punkt-Akzessor Kontextinformationen und verfügbare Kontextfunktionen für alle Nachbargitterpunkte gleichen Verfeinerungslevels des Gitterpunktes im Zugriff zur Verfügung:

- Ghost-, Host- und Virtual-Flags
- Raumposition
- Verfeinerungslevel und Seitenlänge
- Anwendungsdaten
- Verfeinerungsmarkierung und -Flag
- Vergröberungsmarkierung und -Flag
- Verfeinerungsleveldelta
- Verfeinerungsleveldelta für die automatische Vergröberung
- Anwendungsdaten und Raumpositionen eingebetteter virtueller Gitterpunkte

Die obigen Funktionen unterscheiden sich von ihren Pendants im Punkt-Akzessor nur dadurch, dass jeweils zusätzlich die Richtungsnummer des gewünschten Nachbarn angegeben werden muss.

Ferner lässt sich über `exists()` abfragen, ob von dem Gitterpunkt im Zugriff aus in einer bestimmten Richtung überhaupt ein Nachbargitterpunkt gleichen Verfeinerungslevels existiert.

Insgesamt hat diese Ausgliederung des Zugriffs auf Punkte und Nachbarschaften aus den Punktmengeniteratoren den Vorteil, dass auf Anwendungsseite Funktionen und Methoden definiert werden können, die einen Punkt- oder Nachbarschafts-Akzessor als Eingabe erwarten, ohne auf einen bestimmten Punktmengeniterator festgelegt zu sein.

4.2 Gitterphasen

Die in [Esc16] definierten Gitterphasen sind im Kern unverändert geblieben. Im Rahmen dieser Arbeit wurde jedoch ihre Handhabung überarbeitet und ihr Funktionsumfang stellenweise erweitert.

In [Esc16] wurde festgelegt, dass jede Gitterphase explizit begonnen und beendet werden muss. Nun erlaubt es die Gitterschnittstelle auch, eine Gitterphase - zumindest sofern keine andere Gitterphase aktiv ist - implizit zu beginnen, indem eine Funktion dieser Gitterphase aufgerufen wird.

4.2.1 Gittererrichtung

Der gesamtgitterbezogene Verarbeitungskontext dieser Gitterphase ist `setup`, ein Objektmember von `ggi::RegularGrid` bzw. `ggi::P4estGrid`. Gegenüber [Esc16] stellt er nun einige neue Funktionen bereit, mit denen das zu errichtende GGI-Gitter noch genauer spezifiziert werden kann.

So kann die Anwendung zusätzlich zu den bisherigen folgende weitere Gitterparameter festlegen:

Nachbarschaftauflösung `setConnect()` stellt die *Nachbarschaftsauflösung* ein.

GGI bestückt den Nachbarschafts-Akzessor eines Gitterpunktes mindestens mit den Gitternachbarn, die durch die eingestellte Nachbarschaftsauflösung erfasst werden.

Die Nachbarschaftsauflösung wird auch berücksichtigt beim Aufbau der Ghostschicht und der virtuellen Gitterpunktschichten. Eine kleinere Nachbarschaftsauflösung kann ihren Umfang, mithin also Speicherbedarf und Kommunikationsaufwand verringern.

Wie die Nachbarschaft definiert ist und die Nachbarschaftsauflösung im Einzelnen funktioniert, ist in Abschnitt 4.3.1 beschrieben.

Automatische Gittervergröberung `enableAutoCoarsen()` aktiviert die *automatische Gittervergöberung*.

Die automatische Gittervergöberung ist initial inaktiv und muss explizit aktiviert werden. Wie das automatische Vergröbern des GGI-Gitters im Einzelnen funktioniert, ist in Abschnitt 4.3.4 beschrieben.

Virtuelle Gitterpunkte `enableVirtuals()` aktiviert die Unterstützung für *virtuelle Gitterpunktschichten.*

Die Unterstützung für virtuelle Gitterpunktschichten ist initial inaktiv und muss explizit aktiviert werden. Wie die Unterstützung für virtuelle Gitterpunktschichten im Einzelnen funktioniert ist, in Abschnitt 4.3.2 beschrieben.

4.2.2 Gitterdurchlauf

Der gesamtgitterbezogene Verarbeitungskontext dieser Gitterphase ist `run`, ein Objektmember von `ggi::RegularGrid` bzw. `ggi::P4estGrid`. Er stellt nach wie vor die Iteratoren bereit, mit denen die Punktmengen des Gitterabschnitts des Anwendungsprozesses durchlaufen werden können.

Gegenüber [Esc16] wurde im Rahmen dieser Arbeit die Bereitstellung der Punktmengeniteratoren allerdings so überarbeitet, dass die Punktmengen durch die im C++11 Standard eingeführten „*Range-based for loop*“-Statements durchlaufen werden können.

Gitterpunktbezogene Verarbeitungskontexte

Der Gitterdurchlauf stellt folgende Punktmengeniteratoren bereit:

`GridPointIterator` Dieser Iterator durchläuft die Menge aller Gitterpunkte innerhalb des Gitterabschnitts des Anwendungsprozesses.

`LevelPointIterator` Dieser Iterator durchläuft die Menge aller Gitterpunkte mit Verfeinerungslevel l innerhalb des Gitterabschnitts des Anwendungsprozesses.

`VirtualPointIterator` Dieser Iterator durchläuft die Menge aller virtueller Gitterpunkte mit Verfeinerungslevel l innerhalb des Gitterabschnitts des Anwendungsprozesses.

`HostPointIterator` Dieser Iterator durchläuft die Menge aller Gitterpunkte mit Verfeinerungslevel l innerhalb des Gitterabschnitts des Anwendungsprozesses, die virtuelle Gitterpunkte einbetten.

`GhostHostPointIterator` Dieser Iterator durchläuft die Menge aller Ghost-Gitterpunkte mit Verfeinerungslevel l des Gitterabschnitts des Anwendungsprozesses, die virtuelle Gitterpunkte einbetten.

Nutzt die Anwendung für den Zugriff auf die Punktmengen sinnvollerweise „*Range-based for loop*“-Statements, muss sie nicht explizit mit diesen Iteratoren arbeiten.

Stattdessen ruft sie folgende Funktionen, welche `run` nun bereitstellt, und die jeweils eine Referenz auf einen (parametrisierten) Pseudo-Container mit je einer `begin()`- und einer `end()`-Methode liefern:

`gridPoints()` Liefert einen Pseudo-Container, der alle Gitterpunkte innerhalb des Gitterabschnitts des Anwendungsprozesses enthält.

`levelPoints(int level)` Liefert einen Pseudo-Container, der alle Gitterpunkte mit Verfeinerungslevel $l = level$ innerhalb des Gitterabschnitts des Anwendungsprozesses enthält.

`virtualPoints(int level)` Liefert einen Pseudo-Container, der alle virtuellen Gitterpunkte mit Verfeinerungslevel $l = level$ innerhalb des Gitterabschnitts des Anwendungsprozesses enthält.

`hostPoints(int level)` Liefert einen Pseudo-Container, der alle Hostgitterpunkte mit Verfeinerungslevel $l = level$ innerhalb des Gitterabschnitts des Anwendungsprozesses enthält.

`ghostHostPoints(int level)` Liefert einen Pseudo-Container, der alle Hostgitterpunkte mit Verfeinerungslevel $l = level$ innerhalb des Gitterabschnitts des Anwendungsprozesses enthält.

Eine Schleife wie die folgende durchliefe alle Gitterpunkte mit dem Verfeinerungslevel 2:

```
for (auto &gridPoint : grid.run.levelPoints(2)) ...
```

Die Schleifenvariable `gridPoint` wäre dann eine Referenz auf den Punkt-Akzessor (d. h. wäre vom Typ `GridPoint&`) des jeweilig referenzierten Gitterpunktes.

4.2.3 Ghost-Austausch

Beim Ghost-Austausch aktualisieren die Schnittstelleninstanzen eines GGI-Gitters die Anwendungsdaten der Gitterpunkte in ihren jeweiligen Ghostschichten, indem sie in jeden Ghost-Gitterpunkt die assoziierten Anwendungsdaten des durch ihn replizierten randständigen Gitterpunktes des in seine Richtung benachbarten Gitterabschnittes kopieren [Esc16].

In [Esc16] umfasste der Ghost-Austausch stets alle Ghost-Gitterpunkte der einzelnen Gitterabschnitte. Im Rahmen dieser Arbeit wurde er so angepasst, dass die assoziierten Anwendungsdaten zielgerichtet für einzelne Verfeinerungslevel aktualisiert werden können.

Der gesamtgitterbezogene Verarbeitungskontext dieser Gitterphase ist `exchange`, ein Objektmember von `ggi::RegularGrid` bzw. `ggi::P4estGrid`.
Dieser stellt nun folgende zusätzliche Funktionen bereit:

`levelPoints(int level)` Aktualisiert die assoziierten Anwendungsdaten aller Ghost-Gitterpunkte mit Verfeinerungslevel $l = level$.

Die adaptive Lattice-Boltzmann-Implementierung verwendet diese Funktion für den in Abschnitt 3.1 beschriebenen *l-Ghost-Austausch*.

`virtualPoints(int level)` Aktualisiert die assoziierten Anwendungsdaten aller virtuellen Ghost-Gitterpunkte mit Verfeinerungslevel $l = level$ aus.

Die adaptive Lattice-Boltzmann-Implementierung verwendet diese Funktion für den in Abschnitt 3.1 beschriebenen *l-Ghost-Austausch für virtuelle Gitterpunkte*.

4.2.4 Gitteranpassung

In [Esc16] war vorgesehen, dass in dieser Gitterphase alle akkumulierten Gitteranpassungen an allen dafür markierten Gitterpunkten durchgeführt werden. Zu diesem Zweck würde die Anwendung in einem oder innerhalb einer beliebigen Anzahl von Gitterdurchläufen Gitterpunkte auswählen und zur Verfeinerung oder Vergröberung markieren.

Im Rahmen dieser Arbeit wurde die Gitteranpassung dahingehend überarbeitet, dass sie nun auch direkte Verfeinerung oder Vergröberung des Gitters erlaubt.

Der gesamtgitterbezogene Verarbeitungskontext dieser Gitterphase ist `update`, ein Objektmember von `ggi::RegularGrid` bzw. `ggi::P4estGrid`.
Dieser stellt nun folgende zusätzliche Funktionen bereit:

`refine(std::function<bool(RefineContext &)> criterion)` Diese Funktion erwartet einen *Verfeinerungs-Callback* d. h. eine Funktion, die auf der Basis eines ihm übergebenen *Verfeinerungspunkt-Akzessors* entscheidet, ob der durch den Akzessor referenzierte Gitterpunkt verfeinert werden soll.

`refine()` durchläuft alle Gitterpunkte, ruft für jeden Gitterpunkt den Refinement-Callback auf und verfeinert gegebenenfalls den entsprechenden Gitterpunkt.

Die adaptive Lattice-Boltzmann-Implementierung verwendet diese Funktion für die in Abschnitt 3.3 beschriebene Gitterverfeinerung an Boundary-Grenzen.

`coarsen(std::function<bool(CoarsenContext &)> criterion)` Diese Funktion erwartet einen *Vergröberungs-Callback* d. h. eine Funktion, die auf der Basis eines ihm übergebenen *Punktfamilien-Akzessors* entscheidet, ob die durch den Akzessor referenzierte Gitterpunktfamilie vergröbert d. h. durch einen einzelnen gröberen Gitterpunkt ersetzt werden soll.

`coarsen()` durchläuft alle Gitterpunkte, ruft für jede Gitterpunktfamilie den Coarsen-Callback auf und vergröbert gegebenenfalls die entsprechende Gitterpunktfamilie.

Wie die akkumulierte sowie die direkte Gitteranpassung im Einzelnen funktionieren ist in Abschnitt 4.3.3 beschrieben.

Insgesamt kann die Anwendung folgende Callbacks für die Gitteranpassung registrieren:

Verfeinerungs-Callback Dieser Callback erhält einen *Verfeinerungspunkt-Akzessor* als Eingabe und liefert zurück, ob der durch den Akzessor referenzierte Gitterpunkt verfeinert werden soll oder nicht.

Der Callback wird im Rahmen einer direkten Verfeinerung für jeden Gitterpunkt einmal aufgerufen.

Vergröberungs-Callback Dieser Callback erhält einen *Punktfamilien-Akzessor* als Eingabe und liefert zurück, ob die durch den Akzessor referenzierte Gitterpunktfamilie vergröbert werden soll oder nicht.

Der Callback wird im Rahmen einer direkten Vergröberung für jede Gitterpunktfamilie einmal aufgerufen.

Ersetzungs-Callback Dieser Callback erhält einen *Ersetzungs-Akzessor* als Eingabe, der den Zugriff auf sowohl die in Ersetzung befindlichen Gitterpunkte als auch die ersetzenden Gitterpunkte vermittelt.

Der Callback wird im Rahmen von Verfeinerungen und Vergröberungen für jede dadurch vorgenommene Ersetzung von Gitterpunkten durch Familien feinerer Gitterpunkt bzw. von Gitterpunktfamilien durch einzelne gröbere Gitterpunkte aufgerufen.

Gewichtungs-Callback Dieser Callback erhält einen *Gewichtungs-Akzessor* als Eingabe, der den Zugriff auf auf den zu gewichtenden Gitterpunkt vermittelt.

Der Callback wird im Rahmen der Gitterneupartitionierung aufgerufen, mit der die Gitteranpassung abschließt.

Beginnt die Anwendung die Gitteranpassung explizit, kann sie dabei je einen Gewichtungs- oder Ersetzungs-Callback registrieren. Diese so registrierten Callbacks werden dann während der folgenden direkten Verfeinerungen und Vergröberungen verwendet. Gibt die Anwendung für eine direkte Verfeinerung oder Vergröberung explizit einen Ersetzungs-Callback an, so verdeckt dieser einen gegebenenfalls zu Beginn der Gitteranpassung registrierten Ersetzungs-Callback. Auch bzw. alternativ kann die Anwendung je einen Gewichtungs- oder Ersetzungs-Callback zum Abschluss der Gitteranpassung angeben, wobei diese gegebenenfalls zu Beginn der Gitteranpassung registrierte Gewichtungs- bzw. Ersetzungs-Callbacks verdecken.

Gitterpunktbezogene Verarbeitungskontexte

Für die verschiedenen Callbacks, welche die Anwendung für die Gitteranpassung registrieren kann, werden eine Reihe von Verarbeitungskontexten bereitgestellt:

Verfeinerungspunkt-Akzessor Der Verfeinerungs-Callback wird stets mit einem Verfeinerungspunkt-Akzessor aufgerufen.

Der Verfeinerungspunkt-Akzessor ist eine Instanz von `RefineContext`, einer inneren Klasse von `ggi::RegularGrid` bzw. `ggi::P4estGrid`.

Er stellt für den Gitterpunkt in Frage folgende Kontextinformationen und -funktionen zur Verfügung:

Raumposition `domainPos()` liefert die Raumposition des Gitterpunktes in Frage innerhalb des durch das verteilte GGI-Gitter aufgespannten Raumvolumens.

Verfeinerungslevel und Seitenlänge `level()` und `length()` liefern den Verfeinerungslevel l und die Seitenlänge des Gitterpunktes in Frage.

Anwendungsdaten Der Zugriff auf die mit dem Gitterpunkt in Frage assoziierten Anwendungsdaten ist sowohl explizit mit `data()` als auch über überladene C++ Dereferenzierungsoperatoren möglich.

Punktfamilien-Akzessor Der Vergröberungs-Callback wird stets mit einem Punktfamilien-Akzessor aufgerufen.

Der Punktfamilien-Akzessor ist eine Instanz von `CoarsenContext`, einer inneren Klasse von `ggi::RegularGrid` bzw. `ggi::P4estGrid`.
Analog zum Verfeinerungspunkt-Akzessor stellt er für eine Gitterpunktfamilie in Frage folgende Kontextinformationen und -funktionen zur Verfügung:

- Raumposition
- Verfeinerungslevel und Seitenlänge
- Anwendungsdaten

Die obigen Funktionen unterscheiden sich von ihren Pendants im Verfeinerungspunkt-Akzessor nur dadurch, dass jeweils die Nummer des gewünschten Gitterpunktfamilienmitglieds zusätzlich angegeben werden muss. Die Nummerierung der Mitglieder einer Gitterpunktfamilie ist in Z-Ordnung.

Ersetzungs-Akzessor Der Ersetzungs-Callback wird stets mit einem Ersetzungs-Akzessor aufgerufen.

Der Ersetzungs-Akzessor ist eine eine Instanz von `ReplaceContext`, einer inneren Klasse von `ggi::RegularGrid` bzw. `ggi::P4estGrid`.
Er stellt für eine anstehende Ersetzung folgende Kontextinformationen und -funktionen zur Verfügung:

Ersetzungsart `isExpand()` und `isReduce()` geben an, ob es sich um eine expandierende oder reduzierende Ersetzung handelt d. h. ob ein einzelner Gitterpunkt durch eine Familie feinerer Gitterpunkte oder eine Gitterpunktfamilie durch einen einzelnen gröberen Gitterpunkt ersetzt wird.

In Ersetzung befindliche Anwendungsdaten `currentData()` liefert die mit den in Ersetzung befindlichen Gitterpunkten assoziierten Anwendungsdaten.

Die Funktion erwartet die Nummer des in Ersetzung befindlichen Gitterpunktes. Im Falle einer expandierenden Ersetzung gibt die Anwendung hierfür 0 an, im Falle

einer reduzierenden Ersetzung die Nummer des gewünschten Gitterpunktfamilienmitglieds in Z-Ordnung.

Ersetzende Anwendungsdaten `futureData()` liefert die mit den ersetzenden Gitterpunkten assoziierten Anwendungsdaten.

Die Funktion erwartet die Nummer des ersetzenden Gitterpunktes. Im Falle einer reduzierenden Ersetzung gibt die Anwendung hierfür 0 an, im Falle einer expandierenden Ersetzung die Nummer des gewünschten Gitterpunktfamilienmitglieds in Z-Ordnung.

Gewichtungs-Akzessor Der Gewichtungs-Callback wird stets mit einem Gewichtungs-Akzessor aufgerufen.

Der Gewichtungs-Akzessor ist eine Instanz von `WeightContext`, einer inneren Klasse von `ggi::RegularGrid` bzw. `ggi::P4estGrid`.
Er stellt für einen zu gewichtenden Gitterpunkt folgende Kontextinformationen und -funktionen zur Verfügung:

Raumposition `domainPos()` liefert die Raumposition des zu gewichtenden Gitterpunktes innerhalb des durch das verteilte GGI-Gitter aufgespannten Raumvolumens.

Verfeinerungslevel und Seitenlänge `level()` und `length()` liefern den Verfeinerungslevel l und die Seitenlänge h des zu gewichtenden Gitterpunktes.

Anwendungsdaten Der Zugriff auf die mit dem zu gewichtenden Gitterpunkt assoziierten Anwendungsdaten ist sowohl explizit mit `data()` als auch über überladene C++ Dereferenzierungsoperatoren möglich.

4.3 Gesamtgitterfunktionen

Das generische Gitterschnittstellensystem GGI setzt eine Reihe von Funktionalitäten um, die, in [Esc16] stellenweise noch rudimentär ausgeführt, im Rahmen dieser Arbeit vervollständigt worden sind.

Die Unterstützung für virtuelle Gitterpunkte und die jeweiligen Funktionsweisen von Gitterpunkt-Nachbarschaften, Gitteranpassung und der automatischen Gittervergröberung werden in den folgenden Abschnitten im Einzelnen ausgeführt.

4.3.1 Gitterpunkt-Nachbarschaften

GGI macht die Nachbarschaft eines Gitterpunktes zugänglich über seinen Nachbarschafts-Akzessor.

Gitterpunkte und -zellen

GGI überlässt die geometrische Interpretation des errichteten Gitters weitgehend der Anwendung, die es nutzt. So kann die Anwendung die Gitterpunkte interpretieren als:

1. geometrische Punkte, die auf der Raumposition verortet sind, welche der Gitterpunkt-Akzessor zurückgibt, oder
2. geometrische Gitterzellen, deren Ursprung an der Raumposition verortet ist und welche die Seitenlänge aufweisen, welche der Punkt-Akzessor jeweils zurückgibt, oder
3. geometrische Punkte, die im Zentrum einer umgebenden geometrischen Gitterzelle verortet sind, und deren Raumposition der vom Punkt-Akzessor zurückgegebenen zuzüglich der halben Seitenlänge in jeder Koordinatenrichtung entspricht

Die im Rahmen dieser Arbeit überarbeitete Lattice-Boltzmann-Implementierung verfolgt die zweite Interpretation. Auch lehnen sich Definition und verwendete Begrifflichkeiten der Nachbarschaftsauflösung an diese an.

Nachbarschaftsauflösung

Betrachtet man einen Gitterpunkt als geometrische Gitterzelle, lassen sich drei Arten von direkten Nachbarn unterscheiden:

Face-Nachbarn Die Nachbargitterpunkte, deren geometrische Gitterzellen mit der des Ausgangsgitterpunktes über eine gemeinsame Fläche verbunden sind.

Edge-Nachbarn Die Nachbargitterpunkte, deren geometrische Gitterzellen mit der des Ausgangsgitterpunktes nur über eine gemeinsame Kante verbunden sind.

Corner-Nachbarn Die Nachbargitterpunkte, deren geometrische Gitterzellen mit der des Ausgangsgitterpunktes nur über einen gemeinsamen Eckpunkt verbunden sind.

Die eingestellte Nachbarschaftsauflösung legt fest, welche Arten von Nachbargitterpunkten von einem Gitterpunkt aus durch die Nachbarschaftssuche mindestens erreichbar sein sollen. Die Nachbarschaftsauflösung kann auf eine von drei Stufen eingestellt werden:

Nur Face-Nachbarn Von jedem Gitterpunkt aus sind mindestens die Nachbargitterpunkte erreichbar, die Face-Nachbarn sind.

Face-Nachbarn und Edge-Nachbarn Von jedem Gitterpunkt aus sind mindestens die Nachbargitterpunkte erreichbar, die entweder Face-Nachbarn oder Edge-Nachbarn sind.

Face-Nachbarn, Edge-Nachbarn und Corner-Nachbarn Von jedem Gitterpunkt aus sind alle direkten Nachbargitterpunkte erreichbar.

Da Gitterpunkte in der Ghostschicht nur als Nachbargitterpunkte erreichbar sind, muss die Ghostschicht auch nur solche Gitterpunkte der benachbarten Gitterabschnitte enthalten, die gemäß der eingestellten Nachbarschaftsauflösung ereichbar sein müssen. Dementsprechend kommt GGI für kleinere Nachbarschaftsauflösungen mit weniger umfangreichen Ghostschichten aus.

Für eingebettete virtuelle Gitterpunkte gilt dies entsprechend.

Nachbarschaftssuche

Die durch GGI implementierte Nachbarschaftssuche berücksichtigt nicht nur die eingestellte Nachbarschaftsauflösung sondern auch den Verfeinerungslevel des Ausgangsgitterpunktes. Sie liefert nur Nachbargitterpunkte zurück, die den gleichen Verfeinerungslevel aufweisen wie der Ausgangsgitterpunkt.

Liegen in einer Richtung mehrere feinere Gitterpunkte, gibt die Nachbarschaftssuche keinen Nachbarn für diese Richtung zurück.
Liegt in einer Richtung ein gröberer Gitterpunkt und ist die Unterstützung für virtuelle Gitterpunkt nicht aktiviert, gibt die Nachbarschaftssuche keinen Nachbarn für diese Richtung zurück.

4.3.2 Virtuelle Gitterpunkte

GGI implementiert eine Unterstützung für virtuelle Gitterpunkte.

Ist diese bei der Gittererrichtung aktiviert worden, bettet GGI automatisch virtuelle Gitterpunkte mit Verfeinerungslevel $(l + 1)$ in Gitterpunkte mit dem Verfeinerungslevel l ein, die den Rand eines Gitterbereiches mit Verfeinerungslevel $(l + 1)$ bilden.
Ein Gitterpunkt mit dem Verfeinerungslevel l liegt genau dann am Rand eines $(l + 1)$-Gitterbereiches, wenn er von wenigstens einen $(l+1)$-Gitterpunkt ausgehend über die Nachbarschaftssuche mit der eingestellten Nachbarschaftsauflösung erreichbar wäre, hätte er denselben Verfeinerungslevel $(l + 1)$.

Die Unterstützung für virtuelle Gitterpunkte erweitert die Ergebnisse der Nachbarschaftssuche für virtuelle wie nichtvirtuelle Ausgangsgitterpunkte entsprechend:
Liegt in einer Richtung ein gröberer Gitterpunkt, gibt die Nachbarschaftssuche von den in diesen eingebetteten virtuellen Gitterpunkten genau den als Nachbarn zurück, dessen geometrische Gitterzelle gemäß der eingestellten Nachbarschaftsauflösung der des Ausgangsgitterpunktes benachbart ist.

4.3.3 Gitteranpassung

Während der Gitteranpassung kann die Anwendung die Verfeinerungslevel ausgewählter Gitterpunkte verändern d. h. sie verfeinern oder vergröbern. Das GGI-Gitter sorgt dabei automatisch dafür, dass das Gitter nach Abschluss wieder in 2:1-Balance vorliegt und in etwa gleichgroßen Abschnitten auf die Anwendungsprozesse verteilt ist.

Ist die Unterstützung für virtuelle Gitterpunkte aktiviert, ermittelt GGI auch die Gitterpunkte, welche an Gitterbereiche höherer Verfeinerung angrenzen und reichert sie entsprechend mit virtuellen Gitterpunkten an.
Grundsätzlich werden bei jeder Gitteranpassung, die mit virtuellen Gitterpunkten assoziierten Anwendungsdaten verworfen, auch für die virtuellen Gitterpunkte, die unangetastet bleiben.

GGI stellt zwei Wege bereit, die Verfeinerung des Gitters zu verändern: *akkumulierte* und *direkte* Gitteranpassung.

Akkumulierte Gitteranpassung

Bei der akkumulierten Gitteranpassung wendet die Anwendung während des Gitterdurchlaufs ein Kriterium auf jeden betrachteten Gitterpunkt an, das festlegt, ob dieser verfeinert, gleich belassen oder vergröbert werden soll.

Punkt- und Nachbarschafts-Akzessor stellen jeweils entsprechende Kontextfunktionen bereit, mit denen der ausgewählte Gitterpunkt zur Verfeinerung oder Vergröberung vorgemerkt werden kann:

`refine(int times)` Diese Kontextfunktion merkt den Gitterpunkt im Zugriff dafür vor, bei der Gitteranpassung $times$-mal verfeinert zu werden.

Dabei wird ein Gitterpunkt $times$-mal verfeinert, indem er zunächst einmal verfeinert wird und alle seine Nachfolgergitterpunkte jeweils $(times - 1)$-mal verfeinert werden.

`coarsen(int times)` Diese Kontextfunktion merkt den Gitterpunkt im Zugriff dafür vor, bei der Gitteranpassung $times$-mal vergröbert zu werden.

Dabei wird ein Gitterpunkt $times$-mal vergröbert, indem er zunächst einmal vergröbert wird und sein Nachfolgergitterpunkt $(times - 1)$-mal vergröbert wird.

Da eine Verfeinerung eines Gitterpunktes zu einer Erhöhung und eine Vergröberung entsprechend zu einer Verminderung seines Verfeinerungslevels l führt, speichert GGI nur das Delta seines Verfeinerungslevels Δl als Markierung.

GGI hält daher für jeden Gitterpunkt den Δl-Wert, der angibt, um wieviel sein Verfeinerungslevel l bei der Gitteranpassung verändert werden soll. Entsprechend erhöht `refine(int times)` den Δl-Wert um $times$, während `coarsen(int times)` ihn um $times$ verringert.

Die Anwendung kann den Δl-Wert eines Gitterpunktes auch direkt abfragen und (zurück-)setzen mit den folgenden Kontextfunktionen:

- `getLevelDelta()`
- `resetLevelDelta(int levelDelta = 0)`

Das GGI-Gitter akkumuliert somit alle angesetzten Verfeinerungen und Vergröberungen in den Δl-Werten seiner Gitterpunkte, bis die Anwendung schließlich die Gitteranpassung anstößt.

Daraufhin versucht das GGI-Gitter bei allen Gitterpunkten mit $\Delta l \neq 0$ den Verfeinerungslevel l um Δl anzupassen. An allen Gitterpunkten, an denen der Verfeinerungslevel erfolgreich angepasst worden ist, sind die entsprechenden Δl-Werte nach Abschluss der Gitteranpassung wieder 0.

Im allgemeinen sind Verfeinerungen an Gitterpunkten immer erfolgreich (sofern die Verfeinerungskapazität L des Gitters noch nicht erreicht ist).

Vergröberungen allerdings erfordern es, einen Gitterpunkt mit seinen Nachbargitterpunkten zusammenzulegen. Dies kann nur gelingen, wenn der zu vergröbernde Gitterpunkt mit seinen Nachbargitterpunkten eine *Gitterpunktfamilie* bildet.

Eine Anzahl Gitterpunkte bildet genau dann eine Familie, wenn alle den gleichen Verfeinerungslevel l aufweisen und gemeinsam durch eine Verfeinerung aus einem Vorgängergitterpunkt mit Verfeinerungslevel $(l-1)$ hervorgegangen sind.

Bildet ein zu vergröbernder Gitterpunkt mit seinen Nachbargitterpunkten eine Familie, so überprüft GGI als nächstes deren Δl-Werte:
Der Gitterpunkt wird nur dann vergröbert, wenn keiner der Gitterpunkte seiner Familie zur Verfeinerung vorgesehen ist.

Direkte Gitteranpassung

Bei der direkten Gitteranpassung ruft die Anwendung die `refine()`- bzw. `coarsen()`-Funktionen der Gitteranpassung auf und übergibt diesen das anzuwendende Kriterium als Callback. Solange die Gitteranpassung aktiv ist, können die beiden Methoden beliebig oft aufgerufen werden.

Die Methoden durchlaufen alle Gitterpunkte bzw. Gitterpunktfamilien und verfeinern bzw. vergröbern entsprechend dem Rückgabewert des übergebenen Callbacks. Die Δl-Werte der betroffenen Gitterpunkte werden ignoriert.

Die Gitterpunkte bzw. die Gitterpunktfamilien werden ferner *rekursiv* durchlaufen d. h. durch Verfeinerung oder Vergröberung neu entstandene Gitterpunkte bzw. Gitterpunktfamilien werden ebenfalls durchlaufen und können so gleich weiter verfeinert bzw. vergröbert werden.

Die im Rahmen dieser Arbeit erstellte adaptive Lattice-Boltzmann-Implementierung nutzt die direkte Gitteranpassung, um das Rechengitter um Boundary-Grenzen herum zu verfeinern.

4.3.4 Automatische Gittervergröberung

GGI implementiert eine Unterstützung für eine automatische Gittervergröberung.

Da GGI beim Abschluss der Gitteranpassung eine 2:1-Rebalancierung durchführt, werden effektiv mehr Gitterpunkte verfeinert als durch die Anwendung vorgegeben.

Daher hält GGI für jeden Gitterpunkt zusätzlich zu dem Δl-Wert auch noch den $\Delta l_{AutoCoarsen}$-Wert. Für solche Gitterpunkte, die ein Δl-Wert von 0 aufweisen, aber durch die Rebalancierung verfeinert werden, mindert GGI den $\Delta l_{AutoCoarsen}$-Wert um die Anzahl der vorgenommenen Verfeinerungen.

Die Anwendung kann den $\Delta l_{AutoCoarsen}$-Wert eines Gitterpunktes über den Punkt-Akzessor abfragen und zurücksetzen:

- `getAutoCoarsenLevelDelta()`
- `resetAutoCoarsenLevelDelta()`

Der $\Delta l_{AutoCoarsen}$-Wert eines Gitterpunktes kann sich auch durch von der Anwendung vorgenommene Verfeinerungen oder Vergröberungen ändern:

- Wird ein Gitterpunkt mit $\Delta l_{AutoCoarsen} < 0$ im Rahmen der akkumulierten Gitteranpassung durch die Anwendung selbst vergröbert, erhöht GGI seinen $\Delta l_{AutoCoarsen}$-Wert um die Anzahl der vorgenommenen Vergröberungen bis der $\Delta l_{AutoCoarsen}$-Wert 0 erreicht.
- Wird ein Gitterpunkt mit $\Delta l_{AutoCoarsen} < 0$ im Rahmen der akkumulierten Gitteranpassung durch die Anwendung verfeinert, setzt GGI seinen $\Delta l_{AutoCoarsen}$-Wert auf 0 zurück.
- Bei Gitterpunkten, die im Rahmen der direkten Gitteranpassung verfeinert oder vergröbert werden, setzt GGI den $\Delta l_{AutoCoarsen}$-Wert zurück auf 0.

Ist die automatische Gittervergröberung aktiviert, versucht GGI während des Abschlusses der akkumulierten Gitteranpassung alle Gitterpunkte mit $\Delta l_{AutoCoarsen} < 0$ zusätzlich zu denen mit $\Delta l < 0$ zu vergröbern.

4.4 Erweiterungsmöglichkeiten und -grenzen der generischen Gitterschnittstelle

Die generische Gitterschnittstelle wurde in [Esc16] und im Rahmen dieser Arbeit in erster Linie daraufhin entwickelt, den Zugriff auf adaptive wie nicht-adaptive *kartesische* Gitter zu vereinheitlichen.

4.4.1 Möglichkeiten

Die generische Gitterschnittstelle abstrahiert von der konkreten Geometrie des durch sie vermittelten Gitters, indem sie die Gittergeometrie in zwei Modelle kapselt:

Richtungsmodell Die generische Gitterschnittstelle definiert die Nachbarschaft eines Gitterpunktes als Menge von Gitterpunkten, die entlang der Indizes eines *Richtungsmodells* angeordnet sind. Das Richtungsmodell wird durch die jeweilige Gitterimplementierung bereitgestellt, welche die generische Gitterschnittstelle ausprägt.

Die beiden in [Esc16] bzw. im Rahmen dieser Arbeit erstellten Schnittstellenimplementierungen stellen ein *kartesisches* Richtungsmodell bereit.
Ihr Richtungsmodell bildet alle 27 geometrischen Richtungen, in denen in einem kartesischen Gitter von einem Ausgangspunkt aus ein direkter Nachbargitterpunkt liegen kann, auf Indizes $d \in [0, 26]$ ab.

Positionsmodell Die generische Gitterschnittstelle definiert eingebettete virtuelle Gitterpunkte bzw. Mitglieder eine Gitterpunktfamilie als Menge von Gitterpunkten, die entlang der Indizes eines *Positionsmodells* angeordnet sind. Das Positionsmodell wird durch die jeweilige Gitterimplementierung bereitgestellt, welche die generische Gitterschnittstelle ausprägt.

Die beiden in [Esc16] bzw. im Rahmen dieser Arbeit erstellten Schnittstellenimplementierungen stellen ein *kartesisches* Positionsmodell bereit, das sowohl auf Gitterpunkte angewandt wird, die virtuelle Gitterpunkte einbetten, als auch auf Familien von Gitterpunkten.
Ihr Positionsmodell bildet alle 8 geometrischen Positionen, an denen innerhalb der geometrischen Gitterzelle eines Gitterpunktes bzw. innerhalb des durch eine Gitterpunktfamilie abgedeckten Volumens eingebettete virtuelle Gitterpunkte bzw. Gitterpunktfamilienmitglieder liegen können, auf Indizes $p \in [0, 7]$ in Z-Ordnung ab.

Eine nicht-kartesische Gitterimplementierung, welche die generische Gitterschnittstelle ausprägen wollte, würde einfach ein entsprechendes eigenes Richtungsmodell sowie Positionsmodell bereitstellen.

So könnte z. B. eine Gitterimplementierung, die ein Tetraedergitter bereitstellt, ein Richtungsmodell definieren, das die 4 Face-Nachbarn, 6 Kantennachbarn und 4 Ecknachbarn eines Tetraeders auf Indizes $d \in [0, 13]$ abbildet, sowie ein Positionsmodell, das die vier Sub-Tetraeder, die sich durch Halbierung der Seiten ergeben, auf Indizes $p \in [0, 4]$ abbildet.

Ebenso denkbar wäre eine Gitterimplementierung, deren Richtungsmodell sich nicht nur auf Richtungen zu direkten Gitternachbarn eines Ausgangsgitterpunktes beschränkt, sondern indirekte Gitternachbarn miteinschließt.

4.4.2 Grenzen

Die generische Gitterschnittstelle definiert Gitteradaptivität als die Möglichkeit Gitterpunkte rekursiv zu verfeinern. Unter den adaptiven Gittern ist die generische Gitterschnittstelle also auf solche beschränkt, deren Adaptivität auf einem rekursiven Unterteilungsschema beruht.

Gitteradaptivität, die z. B. auf dem Anbringen von höheraufgelösten und/oder abweichend orientierten Gitterpatches an dem Grundgitter beruht, könnte durch die generische Gitterschnittstelle nicht mehr vermittelt werden.

5 Umsetzung durch die Schnittstellenimplementierungen

5.1 Die Schnittstellenimplementierung für reguläre kartesische Gitter

Die C++ Templateklasse `ggi::RegularGrid` prägt die generische Gitterschnittstelle aus und stellt ein reguläres kartesisches Gitter bereit. Ihre Implementierung wurde bereits in [Esc16] abgeschlossen. Im Rahmen dieser Arbeit sind lediglich die Überarbeitungen an der generischen Gitterschnittstelle in `ggi::RegularGrid` nachgezogen worden.

Eine Anwendung, die `ggi::RegularGrid` nutzt, wird in jedem Anwendungsprozess P_i eine Instanz I_i von `ggi::RegularGrid` erzeugen, sie mit einem MPI-Kommunikator initialisieren, der alle Anwendungsprozesse (P_i) umfasst, Gitterparameter wie Gittergröße S usw. einstellen, und schließlich die Gittererrichtung anstoßen.

5.1.1 Datenmodell

`ggi::RegularGrid` implementiert sein Gitter als ein kartesisches Gitter von Gitterabschnitten.

Hierzu ordnet es zunächst die Anwendungsprozesse (P_i) in einem kartesischen Gitter an, dem *Prozessgitter*. Mit diesem Prozessgitter deckt `ggi::RegularGrid` das Volumen S^3 des zu errichtenden Gitters ab, wodurch jedem Anwendungsprozess P_i bzw. seiner Instanz I_i der Gitterabschnitt zufällt, der seiner Position im Prozessgitter entspricht. Dies ist in Abbildung 5.1 beispielhaft dargestellt.

5.1.2 Gittererrichtung

Die Instanzen (I_i) von `ggi::RegularGrid` führen bei der Gittererrichtung eine Reihe von lokalen wie kollektiven Operationen aus, um sich zu dem regulären Gitter mit den geforderten Gitterparametern zusammenzufügen:

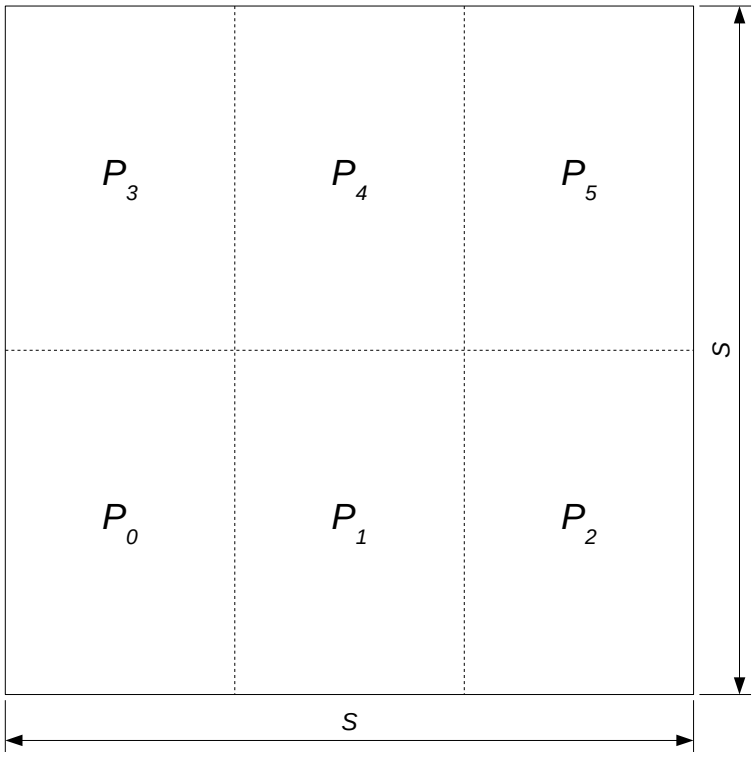

Abbildung 5.1: Kartesisches Prozessgitter für den 2D-Fall mit 6 Anwendungsprozessen, die das Volumen S^2 eines zu errichtenden Gitters abdecken.
Die Gitterabschnitte sind durch gestrichelte Linien begrenzt.

Verteilung der Gitterparameter Die Instanz I_0 des Master-Anwendungsprozesses P_0 verteilt die durch ihn an ihr eingestellten Gitterparameter über einen MPI-Broadcast an alle übrigen Instanzen.

Dadurch wird sichergestellt, dass jede Instanz I_i die Größe S des zu errichtenden Gesamtgitters kennt.

Errichtung des Prozessgitters Alle Instanzen (I_i) erzeugen in einer kollektiven Operation einen neuen MPI-Kommunikator, der alle Anwendungsprozesse (P_i) in einem kartesischen Gitter anordnet.

Jede Instanz I_i ermittelt dann die Position ihres Anwendungsprozesses P_i und seine Nachbarn in allen 27 Raumrichtungen in diesem Prozessgitter. Dadurch wird sichergestellt, dass jede Instanz I_i weiß, welchen Abschnitt des Gesamtgitters sie hält und welche Prozesse benachbarte Gitterabschnitte halten.

Zuschnitt der Gitterabschnitte Alle Instanzen (I_i) berechnen die jeweilige Größe in x-, y- und z-Richtung aller Gitterabschnitte aller Anwendungsprozesse.

Diese ergibt sich für jede Achsenrichtung aus dem ganzzahligen Quotient der Gittergröße S durch die Anzahl der Anwendungsprozesse, die das Prozessgitter auf dieser Achse anordnet. Ein etwaiger Rest r wird auf die ersten r Anwendungsprozesse auf der jeweiligen Achse umverteilt, sodass alle Gitterabschnitte ungefähr gleichgroß bleiben.

Über ihre eigene Position im Prozessgitter kennt nun jede Instanz I_i die Größe $(S_{i,x}, S_{i,y}, S_{i,z})$ ihres Gitterabschnitts.

Allokation der Gitterabschnitte Alle Instanzen (I_i) allokieren ihre jeweiligen Gitterabschnitte und umgeben sie mit einer Ghostschicht.

Hierzu allokiert jede Instanz I_i ein dreidimensionales Array von Gitterpunkten mit der Größe $(S_{i,x} + 2, S_{i,y} + 2, S_{i,z} + 2)$, sodass ihr lokaler Gitterabschnitt auf jeder Achse mit je einer zusätzlichen Gitterpunktschicht beginnt und endet.

Indizierung der Gitterabschnitte Alle Instanzen I_i indizieren ihre jeweiligen Gitterabschnitte, um sie zum einen iterierbar zu machen und zum anderen als Vorbereitung für kommende Ghost-Austausche. Hierzu zerlegen sie ihren Gitterabschnitt in ein Volumen und je eine Facette je Raumrichtung.

Das Volumen wird erzeugt als Array von Zeigern auf alle lokalen Gitterpunkte mit $(0 < x < S_{i,x} + 1, 0 < y < S_{i,y} + 1, 0 < z < S_{i,z} + 1)$; dieses Zeigerarray wird von den Punktmengeniteratoren durchlaufen.

Die Facetten werden für jede der maximal 27 Raumrichtungen, welche durch die eingestelle Nachbarschaftsauflösung abgedeckt wird, erzeugt.

Eine Facette hält zwei Arrays von Zeigern auf lokale Gitterpunkte:

Ghost-Array Dieses Array enthält Zeiger auf alle lokalen Gitterpunkte, die in dem Abschnitt der Ghostschicht liegen, welcher der Raumrichtung der Facette entspricht.

Rand-Array Dieses Array enthält Zeiger auf alle lokalen Gitterpunkte, die in dem Abschnitt des Volumenrandes liegen, welcher der Raumrichtung der Facette entspricht.

In der Facette für die Raumrichtung $(1, 0, 0)$ z. B. enthielte das Ghost-Array Zeiger auf alle Gitterpunkte mit den Positionen $(S_{i,x} + 1, 0 < y < S_{i,y} + 1, 0 < z < S_{i,z} + 1)$ und das Rand-Array Zeiger auf alle Gitterpunkte mit den Positionen $(S_{i,x}, 0 < y < S_{i,y} + 1, 0 < z < S_{i,z} + 1)$. In Abbildung 5.2 ist dies beispielhaft dargestellt.

Jede Facette enthält demnach die Liste der Ghostgitterpunkte, deren Anwendungsdaten Im Rahmen eines Ghost-Austauschs von dem Nachbarprozess in ihrer jeweiligen Raumrichtung empfangen, sowie die Liste der lokalen randständigen Gitterpunkte, deren Anwendungsdaten an denselben Nachbarprozess versandt werden müssen.

Mit Abschluss der Gittererrichtung hält jede Instanz I_i einen Gitterabschnitt korrekter Größe, umgeben mit einer Ghostschicht, der sich mit den Gitterabschnitten aller anderen Instanzen zum angeforderten verteilten regulären kartesischen Gitter zusammenfügt und bereit ist für die nächsten Gitterphasen.

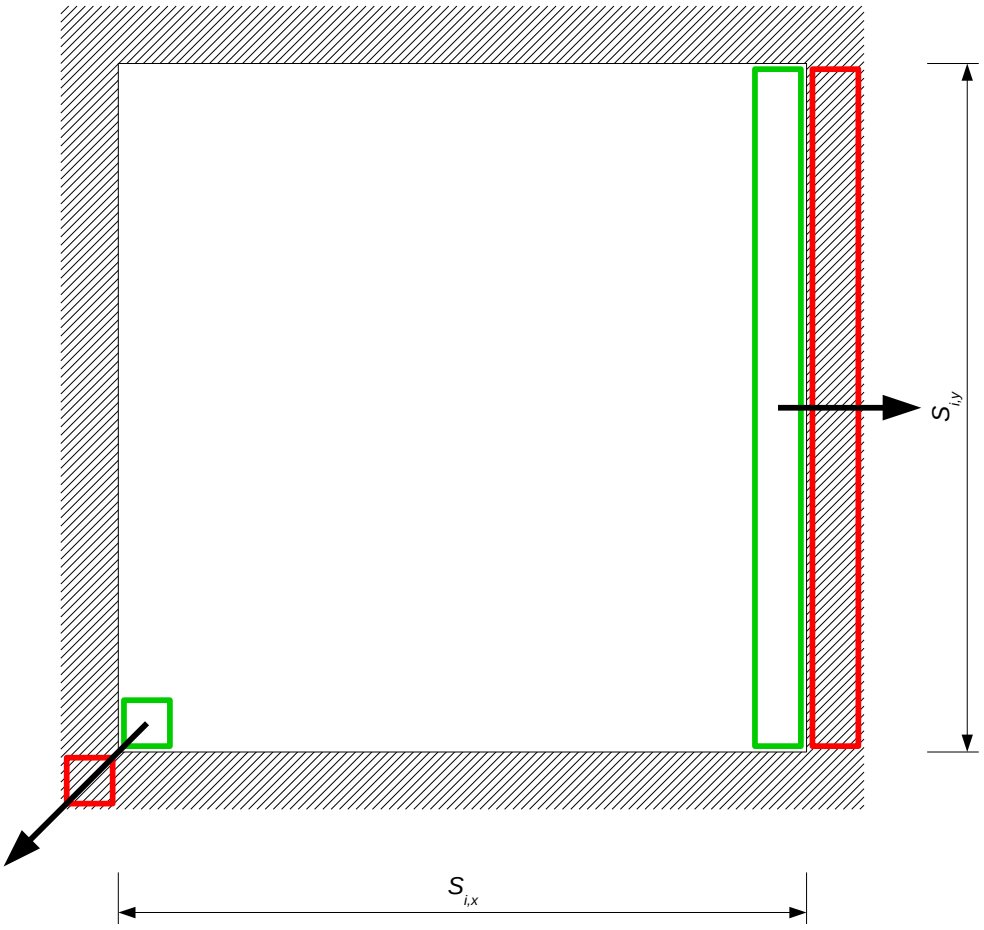

Abbildung 5.2: Die Facetten eines Gitterabschnittes für den 2D-Fall für die Raumrichtungen $(1, 0)$ und $(-1, -1)$.
Die durch die Ghost-Arrays erfassten Gitterpunkte sind rot umrandet, die durch die Rand-Arrays grün.

5.1.3 Gitterdurchlauf

Von den Punktmengeniteratoren des Gitterdurchlaufs implementiert `ggi::RegularGrid` effektiv nur den `GridPointIterator` und den `LevelPointIterator`, die beide alle Gitterpunkte im Volumen des Gitterabschnitts durchlaufen (der LevelPointIterator allerdings nur für den Verfeinerungslevel 0). Die übrigen Iteratoren sind implementiert als Stubs.

Entsprechend sind die Pseudo-Container `gridPoints()` und `levelPoints(int level)` implementiert, wobei der letztere Iteratoren für die leere Punktmenge zurückgibt für $\forall level > 0$.

Die übrigen Pseudo-Container sind als Stubs ausgeführt, die ebenfalls Iteratoren für die leere Punktmenge zurückgeben.

5.1.4 Ghost-Austausch

Beim Ghost-Austausch durchläuft jede Instanz I_i synchron mit allen anderen Instanzen jede der maximal 27 Raumrichtungen, welche durch die eingestellte Nachbarschaftsauflösung abgedeckt werden. Für jede betroffene Raumrichtung d sendet I_i die Anwendungsdaten aller Gitterpunkte, die durch das Rand-Array der Gitterfacette für d erfasst sind, an den Nachbarprozess in Richtung d. Gleichzeitig empfängt I_i die Anwendungsdaten aller Gitterpunkte, die durch das Ghost-Array der Gitterfacette für die inverse Richtung $-d$ erfasst sind, von dem Nachbarprozess in Richtung $-d$.

Mit Abschluss dieses Vorgangs sind die Anwendungsdaten aller Ghostgitterpunkte aller Gitterabschnitte auf den Stand der Anwendungsdaten der durch sie replizierten randständigen Gitterpunkte benachbarter Gitterabschnitte aktualisiert.

5.1.5 Gitteranpassung

Alle Funktionen der Gitteranpassungen sind durch `ggi::RegularGrid` als Stubs ausgeführt. Da ein reguläres Gitter keine Gitteranpassung vorsieht, werden alle entsprechenden Aufrufe ignoriert.

Um mit den Anforderungen der generischen Gitterschnittstelle konform zu gehen, liefert `ggi::RegularGrid` stets eine maximale Verfeinerungszahl L von 0 zurück.

5.1.6 Zusammenfassung

`ggi::RegularGrid` implementiert ein reguläres Gitter ohne besondere Optimierung, die zu Validierungszwecken erstellt worden ist.

5.2 Die Schnittstellenimplementierung für baumstrukturierte kartesische Gitter

Die C++ Templateklasse `ggi::P4estGrid` prägt die generische Gitterschnittstelle aus und stellt ein baumstrukturiertes kartesisches Gitter bereit. Ihre Implementierung wurde in [Esc16] begonnen und im Rahmen dieser Arbeit abgeschlossen.

5.2.1 Datenmodell

`ggi::P4estGrid` implementiert ein baumstrukturiertes Gitter mit der p4est-Library [BWG11].

Die p4est-Library erlaubt die Verwaltung eines auf mehrere Prozesse verteilten Waldes, dessen einzelne Bäume jeweils ein kubisches Raumvolumen abdecken und die an ihren Rändern miteinander verbunden sein können. Welche Baumränder mit welchen anderen verbunden sind, wird durch die p4est-Konnektivität festgelegt, einer Datenstruktur, die den Verbindungsgraph beschreibt [BWG11].

Ein verteilter p4est-Wald W wird errichtet, indem jeder Anwendungsprozess P_i eine p4est-Instanz $I_{i,p4est}$ erzeugt und mit einem MPI-Kommunikator initialisiert, der alle Anwendungsprozesse (P_i) umfasst. Jede p4est-Instanz $I_{i,p4est}$ hält dann einen Abschnitt W_i des p4est-Waldes W, umgeben mit einer Ghostschicht G_i.

Jeder p4est-Baum eines p4est-Waldes erlaubt die Unterteilung seines Raumvolumen in Oktanten und die rekursive Unterteilung dieser Oktanten in weitere, kleinere Oktanten. Auf diese Weise zerlegt jeder p4est-Baum sein Raumvolumen wie in Abbildung 5.3 beispielhaft dargestellt in eine Anzahl nicht mehr weiter unterteilter Volumina unterschiedlicher Größe, den p4est-Oktanten [BWG11].

`ggi::P4estGrid` implementiert sein Gitter als einen verteilten p4est-Wald W, dessen p4est-Oktanten die Gitterpunkte abbilden. Ein solcher p4est-Oktant speichert zunächst lediglich seine Koordinaten innerhalb des Raumvolumens seines p4est-Baums und seinen p4est-Verfeinerungslevel, der angibt, durch wieviele Unterteilungen dieser p4est-Oktant aus dem ursprünglichen Raumvolumen des p4est-Baums hervorgegangen ist.

Die p4est-Library sieht zwei Wege vor, mit den p4est-Oktanten eines p4est-Waldes W Anwendungsdaten zu assoziieren:

Interne Assoziation Bei der internen Assoziation verwaltet p4est die Anwendungsdaten für jeden p4est-Oktanten.

Die Anwendung übergibt p4est die Größe der Anwendungsdatenstruktur und p4est allokiert für jeden p4est-Oktanten in W die entsprechende Anzahl Bytes, um die Anwendungsdatenstruktur aufzunehmen. Ebenso speichert p4est für in jedem p4est-Quadranten einen Zeiger auf seine jeweilige Instanz der Anwendungsdatenstruktur ab.

Die interne Assoziation hat den Vorteil, dass p4est bei einer Neupartitionierung des p4est-Waldes die assoziierten Anwendungsdaten automatisch an die anderen Prozesse überträgt.

Externe Assoziation Bei der externen Assoziation verwaltet die Anwendung die Anwendungsdaten für jeden p4est-Oktanten getrennt von p4est.

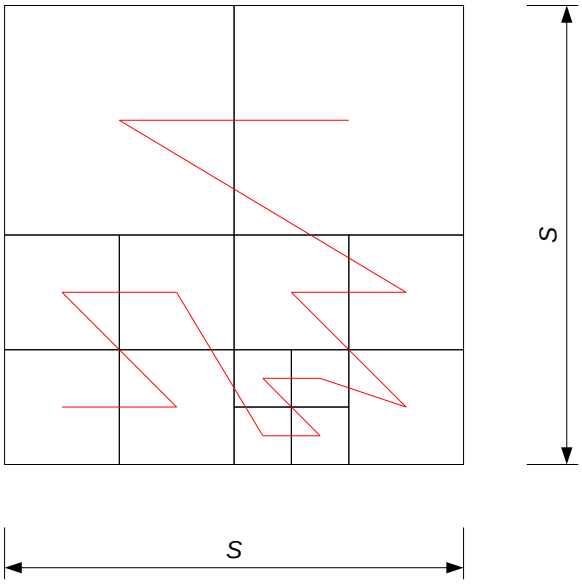

Abbildung 5.3: p4est-Wald mit einem p4est-Baum für den 2D-Fall, der das Volumen S^2 eines baumstrukturierten Gitters abdeckt.
Die Anordnung der p4est-Quadranten in Z-Ordnung ist rot eingezeichnet.

p4est definiert über den p4est-Oktanten eines p4est-Waldabschnittes W_i und ihrer Anzahl $|W_i|$ einen *lokalen Index* p mit $0 \leq p < |W_i|$. Eine Anwendung kann also die Anwendungsdaten einfach in einem Array ablegen, das parallel zu p gehalten wird.

Die externe Assoziation hat den Nachteil, dass bei einer Neupartitionierung des p4est-Waldes die extern verwalteten Anwendungsdaten aufgegeben werden müssen, da nach ihrem Abschluss Zusammensetzung und Größe jedes p4est-Waldabschnittes verändert sein und die enthaltenen p4est-Oktanten neue lokale Indizes erhalten haben können. Für eine Anwendung jedoch, die ihren p4est-Wald nur einmal partitioniert und ihre Anwendungsdaten danach initialisert, kann die externe Assoziation Performancevorteile bedeuten, da nicht mehr indirekt über Zeiger in den p4est-Oktanten auf die Anwendungsdaten zugegriffen werden muss.

Für die Anwendungsdaten, die mit p4est-Oktanten der Ghostschichten von p4est-Waldabschnitten assoziiert sind, sieht p4est ausschließlich die externe Assoziation vor.

p4est definiert über den p4est-Oktanten der Ghostschicht G_i eines p4est-Waldabschnittes W_i und ihrer Anzahl $|G_i|$ einen *Ghost-Index* g mit $0 \leq g < |G_i|$. Die Anwendung muss die Anwendungsdaten zu diesen Oktanten in einem Array ablegen, das parallel zu g gehalten wird. Da es sich bei den Oktanten in Ghostschichten sowieso um Replikate

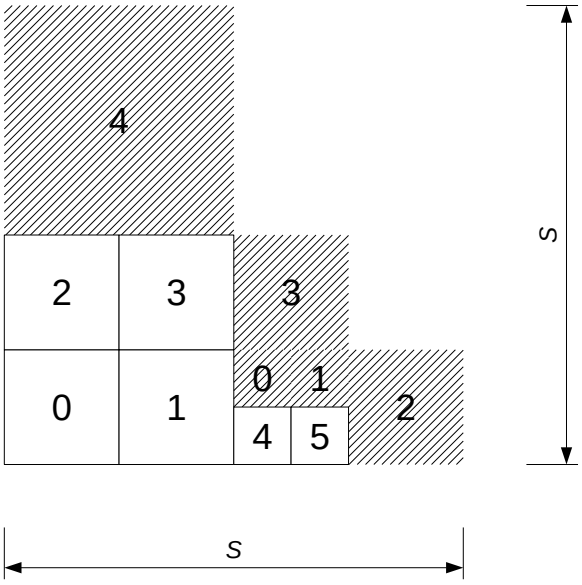

Abbildung 5.4: p4est-Waldabschnitt für den 2D-Fall, der einen Abschnitt des Volumen S^2 eines baumstrukturierten Gitters abdeckt.
Die p4est-Quadranten sind mit ihren lokalen Indizes p versehen, die p4est-Quadranten in der Ghostschicht (schraffiert gezeichnet) mit ihren Ghost-Indizes g.

ordentlicher p4est-Oktanten benachbarter p4est-Waldabschnitte handelt, entsteht der Anwendung aus der externen Assoziation ihrer Anwendungsdaten kein Nachteil.

In Abbildung 5.4 ist ein p4est-Waldabschnitt beispielhaft dargestellt.

`ggi::P4estGrid` implementiert ein dynamisch adaptives Gitter. Das hat zur Folge, dass sich der Umfang einzelner p4est-Waldabschnitte so drastisch ändern kann, dass eine Neupartitionierung notwendig wird, um den p4est-Wald wieder gleichmäßig auf alle Anwendungsprozesse zu verteilen. Daher verwendet `ggi::P4estGrid` die interne Assoziation von Anwendungsdaten.

5.2.2 Gittererrichtung

Eine Anwendung, die `ggi::P4estGrid` nutzt, wird in jedem Anwendungsprozess P_i eine Instanz I_i von `ggi::P4estGrid` erzeugen, sie mit einem MPI-Kommunikator initialisieren, der alle Anwendungsprozesse (P_i) umfasst, Gitterparameter wie Gittergröße S, maximale Verfeinerungszahl L usw. einstellen, und schließlich die Gittererrichtung anstoßen.

Die Instanzen (I_i) von `ggi::P4estGrid` führen bei der Gittererrichtung eine Reihe von lokalen wie kollektiven Operationen aus, um sich zu dem baumstrukturierten Gitter mit den geforderten Gitterparametern zusammenzufügen:

p4est-Initialisierung Jede Instanz I_i erzeugt eine p4est-Instanz $I_{i,p4est}$ und initialisiert diese mit dem MPI-Kommunikator, der die Anwendungsprozesse umfasst.

Verteilung der Gitterparameter Die Instanz I_0 des Master-Anwendungsprozesses P_0 verteilt die durch P_0 an ihr eingestellten Gitterparameter über einen MPI-Broadcast an alle übrigen Instanzen.

Dadurch wird sichergestellt, dass die p4est-Instanz $I_{i,p4est}$ jeder Instanz I_i mit den gleichen Einstellungen versehen wird und sich der verteilte p4est-Wald W ergibt, der dem geforderten verteilten baumstrukturierten kartesischen Gitter entspricht.

Ermittlung der p4est-Konnektivität Die p4est-Konnektivität des zu errichtenden p4est-Waldes sowie die initiale Verfeinerung jeden Baums dieses Waldes ergeben sich aus der Gittergröße S und der maximale Verfeinerungszahl L, die durch die Anwendung eingestellt werden.

Da das durch `ggi::P4estGrid` errichtete Gitter stets ein kubisches Raumvolumen abdeckt, verwendet `ggi::P4estGrid` stets eine p4est-„Bricks“-Konnektivität mit der Waldgröße S_W, die einen p4est-Wald aus Bäumen definiert, die zu S_W Bäumen je Achse wie die Zellen eines regulären Gitters angeordnet und miteinander verbunden sind.

`ggi::P4estGrid` errechnet die Waldgröße als S_W und die initiale p4est-Verfeinerung l_0 jeden p4est-Baums aus der Gittergröße S, der maximalen Verfeinerungszahl L und der maximalen Verfeinerungszahl eines p4est-Baumes L_{p4est}:

$$S_W := max(1, 2^{log_2(S)+L-L_{p4est}})$$

$$l_0 := log_2(S) - log_2(S_W)$$

Auf diese Weise stellt `ggi::P4estGrid` sicher, dass die p4est-Konnektivität, die von jeder Instanz I_i separat erzeugt und gehalten werden muss, nur gerade so groß ist, wie es für die Gittergröße S und die maximale Verfeinerungszahl L unbedingt erforderlich ist.

Initiale Verfeinerung und Partitionierung Alle Instanzen (I_i) stoßen die Errichtung des p4est-Waldes W mit der Waldgröße S_W an, verfeinern ihren jeweiligen Waldabschnitt W_i l_0 mal und stoßen wiederum die Partitionierung an, um den p4est-Wald in gleichgroßen Abschnitten zu verteilen. Der so errichtete p4est-Wald W enthält nun insgesamt S^3 p4est-Oktanten mit der initialen p4est-Verfeinerung l_0.

Die Gitterpunkte, die diesen p4est-Oktanten entsprechen, werden von `ggi::P4estGrid` gegenüber der Anwendung mit Verfeinerungslevel $l = 0$ aufgeführt.

Errichtung der Ghostschicht Alle Instanzen (I_i) stoßen die Errichtung der p4est-Ghostschichten G_i um alle Abschnitte W_i des verteilten p4est-Waldes W an. Im Anschluss allokiert jede Instanz die Arrays mit den Anwendungsdaten, die mit den p4est-Oktanten seiner jeweiligen Ghostschicht assoziiert sind.

Ermittlung der Mesh-Struktur Jede Instanz I_i lässt sich durch p4est die Mesh-Struktur ihres jeweiligen p4est-Waldabschnittes W_i berechnen.

p4est definiert für die Mesh-Struktur einen weiteren Index über den p4est-Oktanten des Waldabschnittes W_i einschließlich denen seiner Ghostschicht G_i, den *Mesh-Index* m. Für p4est-Oktanten in W_i ist $0 \leq m < |W_i|$, für p4est-Oktanten in G_i ist $|W_i| \leq m < |W_i| + |G_i|$.

Über den Mesh-Index m lassen sich somit sowohl p4est-Oktanten im Waldabschnitt W_i als auch solche in dessen Ghostschicht G_i addresssieren. Daher speichert die Mesh-Datenstruktur für jeden lokalen Index p jeweils für jede der Raumrichtungen d, die durch die eingestellte Nachbarschaftsauflösung abgedeckt werden:

- Anzahl und Größe der in Raumrichtung d benachbarten p4est-Oktanten, sowie
- ihre jeweiligen Mesh-Indizes $(m_{d,j})$ mit $0 \leq j < 4$.

Da `ggi::P4estGrid` für seinen p4est-Wald W eine p4est-„Bricks“-Konnektivität verwendet und zwischen den Größen seiner Oktanten stets eine 2:1-Balance aufrechterhält, kann es für jeden p4est-Oktanten für jede Raumrichtung d nur entweder einen Nachbarn doppelter Größe, einen Nachbarn gleicher Größe oder höchstens vier Nachbarn halber Größe geben.

Mit Abschluss der Gittererrichtung hält jede Instanz I_i einen Gitterabschnitt in Form eines p4est-Waldabschnittes W_i, umgeben mit einer Ghostschicht, mit berechneten Nachbarschaftsinformationen für jeden Gitterpunkt, der sich mit den Gitterabschnitten aller anderen Instanzen zum angeforderten verteilten baumstrukturierten kartesischen Gitter zusammenfügt und bereit ist für die nächsten Gitterphasen.

5.2.3 Gitterdurchlauf

Der `GridPointIterator` durchläuft alle Gitterpunkte des Gitterabschnittes, indem er alle lokalen Indizes p des Waldabschnittes W_i durchläuft, und für jede den jeweiligen p4est-Oktanten und die mit ihm assoziierten Anwendungsdaten lädt.

Die p4est-Mesh-Struktur enthält für jeden p4est-Verfeinerungslevel eine Liste mit den lokalen Indizes aller p4est-Oktanten auf diesem p4est-Verfeinerungslevel. Der `LevelPointIterator` für einen Verfeinerungslevel l durchläuft entsprechend die Indexliste aller p4est-Oktanten mit p4est-Verfeinerungslevel $(l_0 + l)$.

5.2.4 Ghost-Austausch

Da `ggi::P4estGrid` die Anwendungsdaten intern assoziiert, müssen für den Ghost-Austausch lediglich die entsprechenden p4est-Funktionen aufgerufen werden:

`p4est_ghost_exchange_data()` Führt den Ghost-Austausch für alle p4est-Oktanten und ihre assoziierten Anwendungsdaten der Ghostschicht aus.

`p4est_ghost_exchange_custom_levels()` Führt den Ghost-Austausch nur für p4est-Oktanten und ihre assoziierten Anwendungsdaten der Ghostschicht aus, die einen bestimmte p4est-Verfeinerungslevel aufweisen.

5.2.5 Gitteranpassung

`ggi::P4estGrid` zerlegt die Gitteranpassung in drei Teilvorgänge:

Direkte Gitteranpassung In diesem Vorgang rufen eine oder mehrere Instanzen I_i auf ihrem jeweiligen p4est-Waldabschnitt die p4est-Funktionen `refine()` und/oder `coarsen()` auf. Die jeweiligen Instanzen erhalten von der Anwendung Ersetzungs-, Verfeinerungs- und Vergröberungs-Callbacks, die sie in gekapselter Form an die entsprechenden p4est-Funktionen weiterreichen.

Akkumulierte Gitteranpassung In diesem Vorgang durchlaufen alle Instanzen (I_i) eine Vergröberungs-Verfeinerungs-Neupartitionierungs-Schleife.

Zunächst werden alle die p4est-Oktantenfamilien einmal vergröbert, für deren Verfeinerungslevel die Anwendung einen Delta $\Delta l < 0$ gesetzt hat bzw. für die ein solcher aus einem vorherigen Schleifendurchlauf verblieben ist, und deren Δl inkrementiert.
Dann werden alle die p4est-Oktanten einmal verfeinert, für deren Verfeinerungslevel die Anwendung einen Delta $\Delta l > 0$ gesetzt hat bzw. für die ein solcher aus einem vorherigen Schleifendurchlauf verblieben ist, und deren Δl dekrementiert.
Im Anschluss wird der p4est-Wald neupartitioniert, sodass die Gitteranpassungslast für den nächsten Schleifendurchlauf auf allen Anwendungsprozessen wieder gleich ist.

Diese Schleife wird solange durchlaufen, bis in einem Durchlauf von keiner Instanz I_i mehr eine Anpassung vorgenommen worden ist.

Rebalancierung In diesem Vorgang stoßen alle Instanzen (I_i) die p4est-Rebalancierung des p4est-Waldes an, um die 2:1-Balance wiederherzustellen, die durch die vorangegangenen Gitteranpassungen verloren gegangen sein kann. Im Anschluss wird der balancierte p4est-Wald neupartitioniert, damit jede Instanz I_i wieder einen etwa gleichgroßen Waldabschnitt hält.

Wenn die Anwendung den Abschluss der Gitteranpassung anstößt, prüft `ggi::P4estGrid` ob mindestens ein Anwendungsprozess auf seinem p4est-Waldabschnitt eine direkte Gitteranpassung vorgenommen hat.

Ist das der Fall, führt `ggi::P4estGrid` nur die Rebalancierung aus.
Andernfalls nimmt `ggi::P4estGrid` zunächst die akkumulierte Gitteranpassung vor und anschließend die Rebalancierung.

Schließlich stoßen alle Instanzen (I_i) eine Neuerrichtung der p4est-Ghostschichten um alle Abschnitte des verteilten p4est-Waldes an und reallokieren dann ihre jeweiligen Arrays mit den assoziierten Anwendungsdaten der p4est-Oktanten ihrer jeweiligen Ghostschichten.

Ferner lässt jede Instanz I_i ihre Mesh-Struktur mit den Nachbarschaftsbeziehungen durch p4est neu berechnen.

5.2.6 Umsetzung von virtuellen Gitterpunkten

Für die Umsetzung von Gitterpunkten mit Verfeinerungslevel l, die virtuelle Gitterpunkte mit Verfeinerungslevel $(l + 1)$ einbetten, standen zwei Vorgehensweisen zur Auswahl:

Sub-Oktanten-Ansatz Der einbettende Gitterpunkt und die jeweils eingebetteten virtuellen Gitterpunkte werden durch einen p4est-Quadranten mit p4est-Verfeinerungslevel $(l_0 + l)$ abgebildet, mit dem neben seinen eigenen Anwendungsdaten noch die aller eingebetteten virtuellen Gitterpunkte assoziiert werden.

Die Nachbarschaftssuche würde - ausgehend von einem Nachbargitterpunkt mit Verfeinerungslevel $(l + 1)$ - den einbettenden Gitterpunkt finden und zusätzlich ermitteln, welcher Sub-Oktant nun der eigentliche Nachbar ist und dem gesuchten virtuellen Nachbargitterpunkt entspricht, und dessen assoziierte Anwendungsdaten zurückliefern. Im Rahmen von [LABM16] wurde eine Funktion in der p4est-Library implementiert, die genau diese Suche umsetzt.

Super-Oktanten-Ansatz Der einbettende Gitterpunkt und die jeweils eingebetteten virtuellen Gitterpunkte werden durch eine Familie von p4est-Oktanten mit dem p4est-Verfeinerungslevel $(l_0 + l + 1)$ abgebildet, mit deren Mitgliedern neben den jeweils eigenen Anwendungsdaten noch die des einbettenden Gitterpunktes assoziiert werden.

Die Nachbarschaftssuche würde - ausgehend von einem Nachbargitterpunkt mit Verfeinerungslevel $(l + 1)$ - direkt den p4est-Oktanten der Oktantenfamilie finden, der dem gesuchten virtuellen Nachbargitterpunkt entspricht, und dessen assoziierte Anwendungsdaten zurückliefern.

Ausgehend von einem Nachbargitterpunkt mit Verfeinerungslevel l, fände die Nachbarschaftssuche einen der der p4est-Oktanten der Oktantenfamilie und lieferte die mit diesem zusätzlich assoziierten Anwendungsdaten des Super-Oktanten d. h. des einbettenden Gitterpunktes zurück.

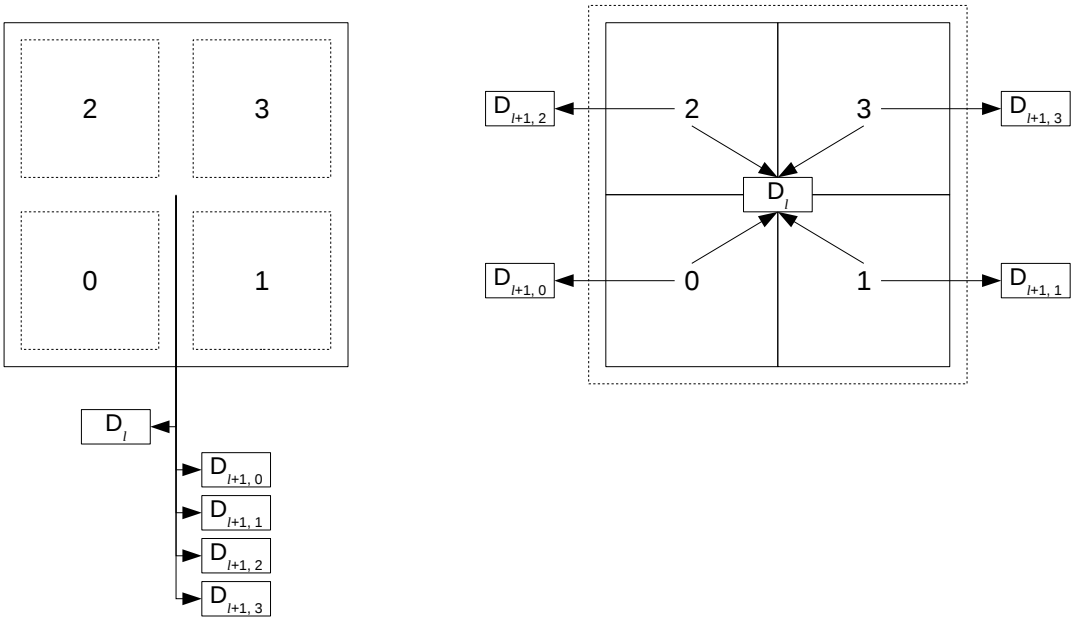

Abbildung 5.5: Links der Sub-Oktanten-Ansatz für den 2D-Fall dargestellt: ein Gitterpunkt mit Verfeinerungslevel l, der virtuelle Gitterpunkte (j) mit Verfeinerungslevel $(l+1)$ einbettet, wird abgebildet durch einen p4est-Quadranten mit p4est-Verfeinerungslevel $(l_0 + l)$, mit dem die Anwendungsdaten des Gitterpunktes D_l sowie die Anwendungsdaten $D_{l+1,j}$ der virtuellen Gitterpunkte (j) assoziiert werden.
Rechts der Super-Oktanten-Ansatz für den 2D-Fall dargestellt: ein Gitterpunkt mit Verfeinerungslevel l, der virtuelle Gitterpunkte (j) mit Verfeinerungslevel $(l+1)$ einbettet, wird abgebildet durch eine Familie von p4est-Quadranten mit p4est-Verfeinerungslevel $(l_0 + l + 1)$, mit denen jeweils die Anwendungsdaten $D_{l+1,j}$ des jeweiligen virtuellen Gitterpunktes j sowie die Anwendungsdaten D_l des einbettenden Gitterpunktes assoziiert sind.

Aufgrund der bereits geleisteten Vorarbeiten in p4est wurde im Rahmen dieser Arbeit der Sub-Oktanten-Ansatz verfolgt.

Anpassungen an der p4est-Library

Ein Problem, das beiden Ansätzen innewohnt, ist der Umstand, dass p4est es nicht erlaubt, mit einem p4est-Oktanten mehr als eine Instanz der Anwendungsdatenstruktur intern zu

assoziieren bzw. mit p4est-Oktanten Anwendungsdaten intern zu assoziieren, die für jeden p4est-Oktanten unterschiedlich groß sein können.

Eine interne Assoziation der Anwendungsdaten ist jedoch erforderlich, da `ggi::P4estGrid` den p4est-Wald jederzeit neupartitionieren können muss. Die p4est-Library allerdings dahingehend anzupassen, dass sie mit intern assoziierten Anwendungsdaten mit oktantenweise unterschiedlicher Größe an den p4est-Oktanten umgehen kann, wäre zu aufwendig gewesen. Stattdessen wurde im Rahmen dieser Arbeit ein anderer Weg gewählt.

Die Anwendungsdaten der nichtvirtuellen Gitterpunkte werden weiterhin intern mit ihren p4est-Oktanten assoziiert und können somit durch p4est im Rahmen von Neupartitionierungen und dem Aufbau der Ghostschicht berücksichtigt und gehandhabt werden.

Die Anwendungsdaten der virtuellen Gitterpunkte hingegen werden mit den Quadranten der Gitterpunkte, in die sie eingebettet sind, extern assoziiert. Bei der Gitterpartitionierung können diese extern assoziierten Anwendungsdaten somit zwar nach wie vor nicht durch p4est berücksichtigt und gehandhabt werden - weshalb eine Gitteranpassung auch stets die Anwendungsdaten aller virtuellen Gitterpunkte invalidiert - für den Ghost-Austausch hingegen ergibt sich eine Möglichkeit.

p4est stellt bereits Funktionen bereit, mit denen extern assoziierte Anwendungsdaten einem Ghost-Austausch unterzogen werden können. Daran angelehnt wurden im Rahmen dieser Arbeit Funktionen hinzugefügt, die extern assoziierte Anwendungsdaten mit *oktantenweise unterschiedlicher Größe* dem Ghost-Austausch unterziehen:

- `p4est_ghost_exchange_extra`
- `p4est_ghost_exchange_extra_levels`

Zu diesem Zweck wurde der p4est-Oktanten-Datenstruktur ein weiteres Feld hinzugefügt: `extra_data_size`. In dieses Feld kann eine Anwendung wie GGI für jeden p4est-Oktanten die Größe der mit diesem Quadranten extern assoziierten Anwendungsdaten ablegen.

Da p4est beim Aufbau der Ghostschicht die Oktanten (jedoch nicht die intern assoziierten Anwendungsdaten) überträgt, verfügen anschließend alle Anwendungsprozesse über einen konsistenten Stand bzgl. der Größe der mit ihren Ghost-Oktanten extern assoziierten Anwendungsdaten, sodass die obigen Funktionen ihren Ghost-Austausch korrekt vornehmen können.

Datenmodell

Ebenso wie für die extern assoziierten Anwendungsdaten von Ghost-Oktanten, mussten dem Datenmodell von `ggi::P4estGrid` Arrays hinzugefügt werden, welche die Anwendungsdaten eingebetteter virtueller Gitterpunkte enthalten, die extern mit den p4est-Oktanten der sie einbettenden Gitterpunkte assoziiert sind:

`m_vPoints` Dieses Array hält die Anwendungsdaten aller virtuellen Gitterpunkte einer Instanz I_i.

`m_localQuadNo_to_vPointsOffset` Dieses Array ist parallel gehalten zum lokalen Index p der p4est-Oktanten des p4est-Waldabschnitts W_i und hält für jeden p4est-Oktanten einen Offset in `m_vPoints` oder -1.

Ein p4est-Oktant, für den dieses Array einen Offset hält, bildet einen Gitterpunkt ab, der virtuelle Gitterpunkte einbettet. Das Offset gibt in diesem Fall die Stelle in `m_vPoints`, an der die Z-geordnete Sequenz der Anwendungsdaten dieser virtuellen Gitterpunkte liegt.

Ein p4est-Oktant, für den dieses Array -1 hält, bildet einen Gitterpunkt ab, der keine virtuellen Gitterpunkte einbettet.

`m_ghostQuadNo_to_vPointsOffset` Dieses Array ist parallel gehalten zum Ghost-Index g der p4est-Oktanten in der Ghostschicht G_i des p4est-Waldabschnitts W_i und hält für jeden dieser p4est-Oktanten einen Offset in `m_vPoints` oder -1.

Ein p4est-Oktant der Ghostschicht, für den dieses Array einen Offset hält, bildet einen Gitterpunkt in der Ghostschicht ab, der virtuelle Gitterpunkte einbettet. Das Offset gibt in diesem Fall die Stelle in `m_vPoints`, an der die Z-geordnete Sequenz der Anwendungsdaten dieser virtuellen Gitterpunkte liegt.

Ein p4est-Oktant der Ghostschicht, für den dieses Array -1 hält, bildet einen Gitterpunkt in der Ghostschicht ab, der keine virtuellen Gitterpunkte einbettet.

Für die Punktmengeniteratoren, die eingebettete virtuelle Gitterpunkte durchlaufen bzw. die einbettenden Gitterpunkte, wurden zwei weitere Arrays hinzugefügt:

`m_treeLevel_to_hostLocalQuadNo` Dieses Array hält für jeden p4est-Verfeinerungslevel des Waldabschnittes einer Instanz I_i ein Array mit den lokalen Indizes aller p4est-Oktanten mit diesem p4est-Verfeinerungslevel, die Gitterpunkte abbilden, die virtuelle Gitterpunkte einbetten.

`m_treeLevel_to_hostGhostQuadNo` Dieses Array hält für jeden p4est-Verfeinerungslevel des Waldabschnittes einer Instanz I_i ein Array mit den Ghost-Indizes aller p4est-Oktanten in der Ghostschicht mit diesem p4est-Verfeinerungslevel, die Gitterpunkte in der Ghostschicht abbilden, die virtuelle Gitterpunkte einbetten.

Gittererrichtung

Da ein zu errichtendes Gitter noch keine virtuellen Gitterpunkte enthalten kann, müssen hier nur Initialisierungen durchgeführt werden.

Nach Abschluss der Gittererrichtung, ist `m_vPoints` leer, halten `m_localQuadNo_to_vPointsOffset` und `m_ghostQuadNo_to_vPointsOffset` für jeden

p4est-Oktanten im Waldabschnitt W_i bzw. in der Ghostschicht G_i eine -1, und halten `m_treeLevel_to_hostLocalQuadNo` und `m_treeLevel_to_hostGhostQuadNo` für jeden p4est-Verfeinerungslevel ein leeres Array.

Gitterdurchlauf

`m_treeLevel_to_hostLocalQuadNo` enthält für jeden p4est-Verfeinerungslevel ein Array mit den lokalen Indizes aller p4est-Oktanten auf diesem p4est-Verfeinerungslevel, die Gitterpunkte abbilden, die virtuelle Gitterpunkte einbetten.
Der `HostPointIterator` für einen Verfeinerungslevel l durchläuft alle einbettenden Gitterpunkte mit diesem Verfeinerunglevel, indem er das Array für den p4est-Verfeinerungslevel $(l_0 + l)$ durchläuft.
Der `VirtualPointIterator` für einen Verfeinerungslevel l durchläuft alle virtuellen Gitterpunkte mit diesem Verfeinerungslevel, indem er für jeden Eintrag in dem Array für den p4est-Verfeinerungslevel $(l_0 + l - 1)$ einzeln dessen virtuelle Gitterpunkte durchläuft, bevor er zum nächsten Eintrag wechselt.

`m_treeLevel_to_hostGhostQuadNo` enthält für jeden p4est-Verfeinerungslevel ein Array mit den Ghost-Indizes aller p4est-Oktanten in der Ghostschicht auf diesem p4est-Verfeinerungslevel, die Gitterpunkte in der Ghostschicht abbilden, die virtuelle Gitterpunkte einbetten.
Der `GhostHostPointIterator` für einen Verfeinerungslevel l durchläuft alle einbettenden Gitterpunkte mit diesem Verfeinerunglevel, indem er das Array für den p4est-Verfeinerungslevel $(l_0 + l)$ durchläuft.

Ghost-Austausch

Da `ggi::P4estGrid` die Anwendungsdaten virtueller Gitterpunkte zwar extern assoziiert, die p4est-Library allerdings erweitert worden ist um die Möglichkeit, für jeden p4est-Oktanten extern assoziierte Anwendungsdaten oktantenweise unterschiedlicher Größe zu übertragen, müssen für den Ghost-Austausch lediglich die entsprechenden neu hinzugefügten p4est-Funktionen aufgerufen werden:

`p4est_ghost_exchange_extra()` Führt den Ghost-Austausch für die extern assoziierten Anwendungsdaten aller p4est-Oktanten der Ghostschicht aus.

`p4est_ghost_exchange_custom_levels()` Führt den Ghost-Austausch für die extern assoziierten Anwendungsdaten nur der p4est-Oktanten der Ghostschicht aus, die einen bestimmte p4est-Verfeinerungslevel aufweisen.

Gitteranpassung

Während der Gitteranpassung werden sämtliche bereits bestehenden virtuellen Gitterpunkte zunächst verworfen und dann neu erzeugt. Dies findet unmittelbar im Anschluss an die *Rebalancierung* statt:

`m_vPoints`, `m_localQuadNo_to_vPointsOffset`, `m_ghostQuadNo_to_vPointsOffset`, `m_treeLevel_to_hostLocalQuadNo` und `m_treeLevel_to_hostGhostQuadNo` werden auf denselben Stand zurückgesetzt wie nach Abschluss der Gittererrichtung.

Anschließend werden alle p4est-Oktanten des neu balancierten Waldabschnitts einschließlich derer in der Ghostschicht durchlaufen und für jeden Oktanten in allen Raumrichtungen, die durch die eingestellte Nachbarschaftsauflösung abgedeckt sind, nach Nachbaroktanten mit höherem p4est-Verfeinerungslevel gesucht.
Bei p4est-Oktanten mit wenigstens einem feineren Nachbaroktanten, wird das Feld `extra_data_size` auf einen Wert $\neq 0$ gesetzt.

Für jeden so markierten p4est-Oktanten wird in `m_vPoints` eine Z-geordnete Sequenz mit den Anwendungsdaten seiner virtuellen Gitterpunkte abgelegt. Der Offset dieser Sequenz wird für den p4est-Oktanten in `m_localQuadNo_to_vPointsOffset` bzw. `m_ghostQuadNo_to_vPointsOffset` abgelegt.

Ebenso wird für jeden markierten p4est-Oktanten sein lokaler Index p bzw. sein Ghost-Index g in das Array für seinen jeweiligen p4est-Verfeinerungslevel in `m_treeLevel_to_hostLocalQuadNo` bzw. `m_treeLevel_to_hostGhostQuadNo` eingetragen.

5.2.7 Zusammenfassung

`ggi::P4estGrid` implementiert ein baumstrukturiertes Gitter auf Basis eines p4est-Waldes mit der Optimalität, die im Zeitrahmen dieser Arbeit erzielt werden konnte.

Rückblickend eröffnen sich weitere Optimierungspotentiale:

Wiederholte Neupartitionierungen bei der Gitteranpassung `ggi::P4estGrid` führt im Zuge der *akkumulierten Gitteranpassung* und der *Rebalancierung* zahlreiche Neupartitionierungen auf den p4est-Wald aus, damit alle Instanzen (I_i) zu jedem Zeitpunkt der Gitteranpassung ungefähr diesselbe Anpassungs- und Speicherlast tragen.

Wird `ggi::P4estGrid` hauptsächlich von Anwendungen genutzt, die auf Shared-Memory-Maschinen laufen, stellt sich die Frage nach einer Begrenzung des Speicherverbrauchs einzelner Instanzen eher weniger.

Hierfür wäre zu prüfen, ob eine Neupartitionierung zum Abschluss der Gitteranpassung vielleicht Performancevorteile bietet, welche die Performancenachteile durch eine

zeitweilig ungleichmäßige Verteilung der Gitteranpassungslast über die Instanzen (I_i) überwiegt.

Umsetzung virtueller Gitterpunkte über den Super-Oktanten-Ansatz Es hat sich herausgestellt, dass die in [LABM16] hinzugefügte spezialisierte p4est-Funktion zur Nachbarschaftssuche wesentlich langsamer ist als eine Implementierung, welche die Lookup-Tabellen der Mesh-Datenstruktur nutzt.

Da die Umsetzung der virtuellen Gitterpunkte aber über den Sub-Oktanten-Ansatz erfolgt ist, muss `ggi::P4estGrid` diese langsamere Funktion verwenden, um von einem p4est-Oktanten aus auch Nachbaroktanten mit gleichem p4est-Verfeinerungslevel zu finden, die eingebettet sind in p4est-Oktanten mit niedrigerem p4est-Verfeinerungslevel.

Eine mögliche Optimierung wäre, die Funktionalität dieser p4est-Funktion in die Erzeugung der p4est-Mesh-Datenstruktur zu integrieren. Dieser würden dann weitere Felder für Lookup-Tabellen hinzugefügt, die dann eben für jeden p4est-Oktanten in jeder Raumrichtung die jeweiligen eingebetteten Nachbaroktanten speichern.

Eine radikalere Optimierung wäre, die virtuellen Gitterpunkte über den Super-Oktanten-Ansatz umzusetzen.

Dies hätte den Vorteil, dass keinerlei spezielle p4est-Funktionen für die Nachbarschaftssuche mehr erforderlich wären. Für das Auffinden von p4est-Oktanten mit gleichem p4est-Verfeinerungslevel genügte die normale Nachbarschaftssuche.

Diese fände für einen p4est-Ausgangsoktanten mit dem p4est-Verfeinerungslevel $(l_0 + l)$ und eine gegebene Raumrichtung entweder:

einen p4est-Oktanten mit p4est-Verfeinerungslevel $(l_0 + l)$ Mit diesem p4est-Oktanten können assoziiert sein:

eine Instanz der Anwendungsdatenstruktur Ein solcher p4est-Oktant wäre dann ein nichtvirtueller Nachbargitterpunkt mit gleichem Verfeinerungslevel l des Gitterpunktes, der durch den Ausgangsoktanten abgebildet wird.

zwei Instanzen der Anwendungsdatenstruktur Ein solcher p4est-Oktant wäre dann ein virtueller Nachbargitterpunkt mit gleichem Verfeinerungslevel l des Gitterpunktes, der durch den Ausgangsoktanten abgebildet wird. Die zweite Instanz der Anwendungsdatenstruktur wäre dann die des Super-Oktanten, der dem einbettenden Gitterpunkt mit Verfeinerungslevel $(l - 1)$ entspricht.

einen von mehreren p4est-Oktanten mit p4est-Verfeinerungslevel $(l_0 + l + 1)$ Mit diesem p4est-Oktanten können assoziiert sein:

eine Instanz der Anwendungsdatenstruktur Ein solcher p4est-Oktant wäre ein nichtvirtueller Nachbargitterpunkt mit Verfeinerungslevel $(l + 1)$ und würde entsprechend übergangen.

zwei Instanzen der Anwendungsdatenstruktur Ein solcher p4est-Oktant wäre dann Mitglied des Super-Oktanten, welcher den Nachbargitterpunkt mit Verfeinerungslevel l des Gitterpunktes darstellt, der durch den Ausgangsoktanten abgebildet wird. Die zweite Instanz der Anwendungsdatenstruktur enthielte dann die gewünschten mit diesem Nachbargitterpunkt assoziierten Anwendungsdaten.

einen p4est-Oktanten mit p4est-Verfeinerungslevel $(l_0 + l - 1)$ Ein solcher p4est-Oktant wäre ein nichtvirtueller Nachbargitterpunkt mit Verfeinerungslevel $(l + 1)$ und würde entsprechend übergangen.

Da der Super-Oktanten-Ansatz es erforderlich macht, mit ausgewählten p4est-Oktanten zwei Instanzen der Anwendungsdatenstruktur zu assoziieren, müsste lediglich die im Rahmen dieser Arbeit zur p4est-Library hinzugefügte Unterstützung für die externe Assoziation von Anwendungsdaten an p4est-Oktanten von oktantenweise unterschiedlicher Größe beibehalten werden.

Profiling-Ergebnissen zufolge verbringt eine Lattice-Boltzmann-Implementierung den größten Teil ihrer Rechenzeit in der Nachbarschaftssuche, die für das Strömen benötigt wird.

Eine Umsetzung virtueller Gitterpunkte, welche die schnelle Nachbarschaftssuche nutzen kann, könnte einen erheblichen Performancegewinn erzielen.

6 Integration in ESPResSo

6.1 ESPResSo

Eine ESPResSo-Simulation wird durch ein Tcl-Skript (bzw. ein Python-Skript, zumindest in den neueren Branches des ESPResSo-Projekts) definiert, welches ESPResSo beim Start übergeben wird. Über ESPResSo-spezifische Tcl-Kommandos legt das Skript Simulationsparameter wie räumliche Ausdehnung des Simulationsgebietes, enthaltene Partikel, Potentiale usw. fest. Weitere ESPResSo-spezifische Tcl-Kommandos ermöglichen es desweiteren, Simulationsfunktionen aufzurufen wie das Berechnen von Simulationszeitschritten sowie die Abfrage und Ausgabe von Simulationsergebnissen [LAMH06].

ESPResSo als Anwendung nutzt MPI-Parallelisierung, um das Simulationsgebiet und die Rechenlast unter allen ESPResSo-Prozessen aufzuteilen. Dabei bilden die ESPResSo-Prozesse eine Art Master-Slave-Architektur aus, in welcher der Master-Prozess den Tcl-Interpreter ausführt. Für jede durch das Tcl-Skript gerufene Simulationsfunktion stößt der Master-Prozess deren Ausführung auf allen Slave-Prozessen an, bevor er sie auf dem ihm zugeteilten Abschnitt des Simulationsgebietes ausführt [LAMH06].

6.2 Verarbeitungszyklus

Das Verhalten der durch ESPResSo berechneten Simulation wird durch die verwendeten Simulationsparameter bestimmt. ESPResSo sieht eine Reihe von globalen Simulationsparametern vor, die teilweise voneinander abhängen und die durch Tcl-Kommandos einzeln eingestellt werden können [LAMH06].

Der erste globale Simulationsparameter, das Prozessgitter, wird gleich beim Start von ESPResSo festgelegt, wenn sich die ESPResSo-Prozesse in einer kartesischen Kommunikationsstruktur anordnen und jeweils ihre Position in dem Prozessgitter sowie ihre Nachbarprozesse ermitteln. Auch definieren ESPResSo-Subsysteme wie z. B. das Lattice-Boltzmann-Subsystem eigene, durch Tcl-Kommandos einstellbare Simulationsparameter, die ebenfalls untereinander und/oder von globalen Simulationsparametern abhängig sein können [LAMH06].

Da aus dem Tcl-Skript heraus die meisten Simulationsparameter - ob global oder die von Subsystemen - jederzeit geändert werden können, implementiert ESPResSo eine Art ereignisgetriebenen Verarbeitungszyklus mit im Wesentlichen zwei Arten von Ereignissen:

Aufruf einer Simulationsfunktion Hier führt ESPResSo alle notwendigen Vorverarbeitungen und schließlich die gerufene Simulationsfunktion auf allen Prozessen aus.

Änderung eines Simulationsparameters Hier berechnet ESPResSo alle von den geänderten Simulationsparameter abhängigen Simulationsparameter neu. Durch unmittelbar oder mittelbar geänderte Simulationsparameter gesteuerte Subsysteme werden ganz oder teilweise neu initialisiert, wobei sogar bisher in ihnen angefallene Simulationsdaten verworfen werden können.

Die Ereignisse und ihre Behandlung sind in ESPResSo eher rudimentär in Form von C/C++ Funktionen ausgeführt und ihre Auslösung findet durch Aufrufen statt [LAMH06].

Für das Lattice-Boltzmann-Subsystem bzw. die Integration der adaptiven Lattice-Boltzmann-Implementierung sind relevant:

`on_program_start()` Wird ausgelöst unmittelbar nach der MPI-Initialisierung und bevor ESPResSo die Tcl-Verarbeitung beginnt. Die Funktion ruft dann Funktionen von ESPResSo-Subsystemen auf, die Vorinitialisierungen vornehmen bzw. Subsysteme betriebsbereit machen.

`on_integration_start()` Wird ausgelöst bevor eine Anzahl von Simulationszeitschritten berechnet werden. Die Funktion ruft dann Funktionen von ESPResSo-Subsystemen auf, um relevante Simulationsparameter auf Plausibilität zu prüfen und andere Vorbereitungen zu treffen.

`on_parameter_change()` Wird ausgelöst, wenn sich ein Simulationsparameter ändert. Die Funktion ruft dann Funktionen von ESPResSo-Subsystemen auf, um abhängige Simulationsparameter neu berechnen und Subsysteme neu zu initialisieren.

`on_lbboundary_change()` Ein durch das Lattice-Boltzmann-Subsystem hinzugefügtes Ereignis. Wird ausgelöst, wenn sich die Boundaries der Simulation ändern.

`on_boxl_change()` Wird ausgelöst, wenn sich die Größe des Simulationsgebiet ändert. Die Funktion ruft dann Funktionen von ESPResSo-Subsystemen auf, um abhängige Simulationsparameter neu berechnen und Subsysteme neu zu initialisieren.

`on_cell_structure_change()` Wird ausgelöst, wenn sich die Zellengitter, auf dem die Molekulardynamiksimulation berechnet wird, sich ändert. Die Funktion ruft dann Funktionen von ESPResSo-Subsystemen auf, um abhängige Simulationsparameter neu berechnen und Subsysteme neu zu initialisieren.

`on_temperature_change()` Wird ausgelöst, wenn sich die simulierte Temperatur ändert. Die Funktion ruft dann Funktionen von ESPResSo-Subsystemen auf, um abhängige Simulationsparameter neu berechnen.

`on_lb_params_change()` Ein durch das Lattice-Boltzmann-Subsystem hinzugefügtes Ereignis. Wird ausgelöst, wenn sich einer der subsystem-spezifischen Simulationsparameter ändert.

6.3 Das Lattice-Boltzmann-Subsystem

Das Lattice-Boltzmann-Subsystem ist in ESPResSo in zwei Varianten ausgeführt [LAMH06]. Die Integration der adaptiven Lattice-Boltzmann-Implementierung und damit zusammenhängende Arbeiten sind in der CPU-basierten Variante erfolgt. Auf die GPU-basierte Variante wurde im Rahmen dieser Arbeit nicht eingegangen. Im folgenden wird daher immer von der CPU-basierten Variante des Lattice-Boltzmann-Subsystems und ihrer Überarbeitung die Rede sein.

Das Lattice-Boltzmann-Subsystem in seiner ursprünglichen, nicht-adaptiven, Implementierung definierte einen eigenen Verarbeitungszyklus, der so in den ESPResSo-Verarbeitungszyklus eingefügt war, dass zusätzlich zu den Zeitschritten der Molekulardynamiksimulation Lattice-Boltzmann-Zeitschritte auf dem Lattice-Boltzmann-Fluid ausgeführt wurden und Kräfte zwischen simulierten Partikeln und dem Fluid ausgetauscht werden konnten [LAMH06].

Dieser Verarbeitungszyklus ist im Rahmen der Überarbeitung des Lattice-Boltzmann-Subsystems weitgehend unverändert geblieben:

Initialisierung In diesem Verarbeitungsschritt wird das Lattice-Boltzmann-Subsystem betriebsbereit gemacht, globale Variablen initialisiert usw. Dieser Verarbeitungsschritt ist implementiert in `lb_pre_init()` und wird ausgeführt im Rahmen der Ereignisbehandlung für `on_program_start()`.

Initialisierung der Lattice-Boltzmann-Simulation In diesem Verarbeitungsschritt wird die sich aus den eingestellten globalen wie Subsystem-spezifischen Simulationsparametern ergebende Lattice-Boltzmann-Simulation aufgesetzt und gestartet.

Dabei wird zunächst das Rechengitter angelegt und dann die Rechenparameter aus den Simulationsparametern berechnet. Die Rechenparameter sind Größen wie die Relaxierungsfaktoren, diverse in Gittereinheiten konvertierte Vorfaktoren usw., die vorberechnet werden können aus den Simulationsparametern, da sie zur Berechnung von Lattice-Boltzmann-Zeitschritten wiederholt benötigt werden und sich nicht ändern, sofern die Simulationsparameter gleich bleiben.

Im Anschluss wird das Lattice-Boltzmann-Fluidvolumen initialisiert. Dabei werden an allen Punkten des Rechengitters die Lattice-Boltzmann-Populationen auf die Gleichgewichtsverteilung und die lokal wirkenden Kräfte auf die externe Kraft zurückgesetzt. Auch werden die eingestellten Boundaries auf die Rechengitterpunkte abgebildet.

Ist dieser Verarbeitungsschritt abgeschlossen, kann auf der aufgesetzten Lattice-Boltzmann-Simulation der erste Lattice-Boltzmann-Zeitschritt berechnet werden.

Dieser Verarbeitungsschritt ist implementiert in `lb_init()` und wird ausgeführt im Rahmen der Ereignisbehandlung für `on_boxl_change()`, `on_cell_structure_change()` und `on_lb_params_change()`. Im letzteren Fall nur dann, wenn ein subsystem-spezifischer Simulationsparameter geändert wird, der einen kompletten Neustart der Lattice-Boltzmann-Simulation erfordert.

Neuberechnung der Rechenparameter In diesem Verarbeitungsschritt werden die Rechenparameter der Lattice-Boltzmann-Simulation erneut aus den Simulationsparametern berechnet.

Dieser Verarbeitungsschritt ist implementiert in `lb_reinit_parameters()` und wird ausgeführt im Rahmen der Ereignisbehandlung von `on_temperature_change()`, `on_parameter_change()` und `on_lb_params_change()`, wenn sich ein Simulationsparameter ändert, der keinen kompletten Neustart der Lattice-Boltzmann-Simulation erfordert.

Reinitialisierung des Fluidvolumens In diesem Verarbeitungsschritt wird das Lattice-Boltzmann-Fluidvolumen neu initialisiert.

Dieser Verarbeitungsschritt ist implementiert in `lb_reinit_fluid()` und wird ausgeführt im Rahmen der Ereignisbehandlung von `on_lb_params_change()`, wenn sich ein Simulationsparameter ändert, der eine Zurücksetzung des Fluidvolumens erfordert wie z. B. geänderte Boundaries.

Zeitschritt In diesem Verarbeitungsschritt wird ein Zeitschritt auf dem Lattice-Boltzmann-Fluid berechnet.

Dieser Verarbeitungsschritt ist implementiert in `lattice_boltzmann_update()` und wird ausgeführt, wenn ein Zeitschritt der Molekulardynamiksimulation berechnet wird.

Finalisierung In diesem Verarbeitungsschritt wird das Lattice-Boltzmann-Subsystem heruntergefahren, reservierter Speicher freigegeben usw..

Dieser Verarbeitungsschritt ist implementiert in `lb_finalize()`.

Die C/C++ Funktionen, welche in der ursprünglichen, nicht-adaptiven Lattice-Boltzmann-Implementierung die obigen Verarbeitungsschritte umsetzten, sind im Rahmen der Überarbeitung beibehalten worden. Jedoch leiten sie nur noch die Aufrufe an die eigentlichen Funktionen der adaptiven Lattice-Boltzmann-Implementierung weiter:

- `lbggi_pre_init()`
- `lbggi_init()`
- `lbggi_reinit_parameters()`
- `lbggi_reinit_fluid()`
- `lbggi_collide_stream()`
- `lbggi_finalize()`

Durch dieses Vorgehen verbleiben alle von außen aufrufbaren C/C++ Funktionen des Lattice-Boltzmann-Subsystems in einem einheitlichen Namensraum, dessen Funktionsnamen (fast) alle mit `lb_` beginnen. Den Namensraum der ursprünglichen Implementierung des Lattice-Boltzmann-Subsystems zu erhalten, schien durch den Umstand angezeigt, dass das Subsystem neben den obigen Hauptfunktionen noch eine große Anzahl weiterer C/C++ Funktionen bereitstellt, die an diversen Stellen von ESPResSo aufgerufen werden.

6.3.1 Das `lbfluid`-Kommando

Das Lattice-Boltzmann-Subsystem stellt das Tcl-Kommando `lbfluid` bereit. Das Kommando wird durch `tclcommand_lbfluid()` implementiert [LAMH06].

Mit diesem Kommando können sämtliche Simulationsparameter der Lattice-Boltzmann-Simulation eingestellt und auch ihre Ergebnisse ausgegeben werden [LAMH06].

Für jeden durch dieses Kommando einstellbaren Simulationsparameter bzw. aufrufbare Ergebnisausgabe stellt das Subsystem eine C/C++ Funktion bereit, deren Bezeichner mit `lb_lbfluid_` beginnt und die die entsprechende Verarbeitung auf allen ESPResSo-Prozessen anstößt sowie die relevanten ESPResSo-Ereignisse auslöst [LAMH06].

Im Rahmen der Überarbeitung des Lattice-Boltzmann-Subsystems sind zwei durch dieses Kommando einstellbare Simulationsparameter hinzugekommen:

Gittergröße Gibt die Anzahl Gitterpunkte des GGI-Gitters in jeder Raumrichtung an.

Wird für eine Simulation keine Gittergröße angegeben, berechnet `lb_init()` sie automatisch aus der eingestellten Größe des Simulationsgebietes und der eingestellten Maschenweite.

Maximale Verfeinerungstiefe Gibt an, bis auf welchen Verfeinerungslevel die Lattice-Boltzmann-Simulation das GGI-Gitter maximal hin verfeinern soll.

Dieser Parameter begrenzt somit effektiv die vorgenommene Gitterverfeinerung an Boundary-Grenzen.

6.3.2 Das `lbnode`-Kommando

Das Lattice-Boltzmann-Subsystem stellt das Tcl-Kommando `lbnode` bereit. Das Kommando wird durch `tclcommand_lbnode()` implementiert [LAMH06].

Mit diesem Kommando können für einzelne Gitterpunkte der Lattice-Boltzmann-Simulation Simulationsparameter eingestellt und auch ihre Ergebnisse ausgegeben werden [LAMH06].

Für jeden durch dieses Kommando einstellbaren Simulationsparameter bzw. aufrufbare Ergebnisausgabe stellt das Subsystem eine C/C++ Funktion bereit, deren Bezeichner mit `lb_lbnode_`

beginnt und die die entsprechende Verarbeitung auf allen ESPResSo-Prozessen anstößt sowie die relevanten ESPResSo-Ereignisse auslöst. Diese Funktionen benötigen für ihre Aufgabe die Möglichkeit, für eine Koordinate innerhalb des Simulationsgebiets den dazugehörigen Gitterpunkt innerhalb des GGI-Gitters der Lattice-Boltzmann-Simulation zu bestimmen [LAMH06].

Da dies in GGI nicht implementiert ist, sind alle `lb_lbnode_`-Funktionen derzeit funktionslos.

6.3.3 Partikelkopplung

Das Lattice-Boltzmann-Subsystem koppelt die Lattice-Boltzmann-Simulation an die Partikel der Molekulardynamiksimulation. Diese Kopplung wird durch `calc_particle_lattice_ia()` implementiert [LAMH06].

Diese Funktionen benötigen für ihre Aufgabe die Möglichkeit, für eine Koordinate innerhalb des Simulationsgebiets den dazugehörigen Gitterpunkt innerhalb des GGI-Gitters der Lattice-Boltzmann-Simulation zu bestimmen. Da dies in GGI nicht implementiert ist, ist `calc_particle_lattice_ia()` derzeit funktionslos.

6.3.4 Ausgabe von VTK-Dateien

Das Lattice-Boltzmann-Subsystem stellt Funktionen zur Ausgabe seiner Simulationsergebnisse bereit.

Die Ausgabe kann als Text erfolgen:

- `lb_lbfluid_print_boundary()`
- `lb_lbfluid_print_velocity()`

Sie kann aber auch im VTK-Format erfolgen:

- `lb_lbfluid_print_vtk_boundary()`
- `lb_lbfluid_print_vtk_velocity()`

Diese C/C++ Funktionen sind im Rahmen der Überarbeitung beibehalten worden. Wie weiter oben leiten sie die Aufrufe weiter an die eigentlichen Funktionen der adaptiven Lattice-Boltzmann-Implementierung:

- `lbggi_output_boundary()`
- `lbggi_vtk_output_boundary()`
- `lbggi_output_u()`

- `lbggi_vtk_output_u()`

In der ursprünglichen Implementierung ist der Master-Prozess in einer Schleife über alle Gitterpunkte der Lattice-Boltzmann-Simulation iteriert und hat für jeden Gitterpunkt einzeln den ESPResSo-Prozess ermittelt, der seinen Gitterabschnitt hält, dort die Berechnung des darzustellenden Wertes veranlasst und diesen von dort abgeholt [LAMH06].

Das hatte zwar den Vorteil, dass eine einzige Textausgabe bzw. VTK-Datei entsteht, die vom Master-Prozess geschrieben wird, aber den Nachteil, dass die Slave-Prozesse nicht mehr wirklich parallel zum Master-Prozess arbeiteten.

Die neue Implementierung geht anders vor:

1. Sie verwendet das `UNSTRUCTURED_GRID`-Format von VTK, das es erlaubt, die Gitterzellen und die in ihnen darzustellenden Werte in beliebiger Reihenfolge aufzulisten.
2. Sie berechnet auf jedem ESPResSo-Prozess die darzustellenden Werte aller Gitterpunkte des dortigen Gitterabschnittes auf einmal und überträgt diese an den Master-Prozess, der die VTK-Datei schreibt.

   Diese Übertragung von den ESPResSo-Prozessen zum Master-Prozess wird mittels der C++ Templateklassen `MpiOutgoingTransfer` und `MpiIncomingTransfer` umgesetzt, die im Rahmen dieser Arbeit erstellt wurden und eine *packetweise* Übertragung implementieren. Dadurch wird sichergestellt, dass der zusätzliche Speicherverbrauch des Master-Prozess beschränkt ist auf die Länge des Empfangspuffers für *ein* Packet der Übertragung.

Dadurch bleibt der Vorteil einer einzigen VTK-Datei gewahrt und der Nachteil mangelnder Parallelität wird zumindest abgemildert.

7 Bewertung

In [Esc16] wurde ein Performancetest der generischen Gitterschnittstelle, der Gitterimplementierung `ggi::RegularGrid` für reguläre Gitter und der entkoppelten Lattice-Boltzmann-Implementierung vorgenommen. Dabei war das zu testende System noch nicht in ESPResSo integriert, sondern lag als separates Testprogramm vor.

Im Rahmen dieser Arbeit wurde ein Validierungs- und Performancetest der generischen Gitterschnittstelle, beiden Gitterimplementierungen `ggi::RegularGrid` und `ggi::P4estGrid`, sowie der *adaptiven* Lattice-Boltzmann-Implementierung durchgeführt. Das zu testende System ist nun in ESPResSo integriert.

Sämtliche Tests wurden durchgeführt auf einer Shared-Memory-Maschine mit 72 physischen Kernen (Intel® Xeon® CPU E7-8880 @2,30 GHZ) und 512GB Hauptspeicher.

ESPResSo wurde gebaut mit gcc 5.4.0 und lief unter OpenMPI 1.10.2.

Executables

Für die Tests wurden drei verschiedene Executables von ESPResSo erstellt:

ESPResSo-Alt Dieses Executable wurde aus dem Git-Commit 7a73301 des Git-Repository `gitlab-sgs.informatik.uni-stuttgart.de:lahnerml/espresso.git` erstellt.

In der `myconfig.hpp` war im Abschnitt mit den Lattice-Boltzmann-Features folgendes gegeben:

```
#define LB
#define LB_BOUNDARIES
```

Dies erzeugt ein ESPResSo-Executable, in dem die alte Lattice-Boltzmann-Implementierung, wie sie in ESPResSo bestanden hat, bevor [Esc16] und diese Arbeit begonnen worden sind, aktiv ist.

ESPResSo-GGI-Regulär Dieses Executable wurde aus dem Git-Commit 398f2e5 desselben Git-Repositories wie oben erstellt.

In der `myconfig.hpp` war im Abschnitt mit den Lattice-Boltzmann-Features folgendes gegeben:

```
#define LB
#define LB_BOUNDARIES
#define LB_ADAPTIVE
```

Ferner wurde in `lb-ggi.hpp` der Gitterdatentyp `LbGgiGrid` definiert als:

```
typedef ggi::RegularGrid<LbGgiPoint, LbGgiData> LbGgiGrid;
```

Dies erzeugt ein ESPResSo-Executable, in dem die adaptive Lattice-Boltzmann-Implementierung aktiv ist und auf der Gitterimplementierung `ggi::RegularGrid` für reguläre Gitter arbeitet.

ESPResSo-GGI-P4est Dieses Executable wurde aus demselben Git-Commit 398f2e5 desselben Git-Repositories wie oben erstellt.

In der `myconfig.hpp` war im Abschnitt mit den Lattice-Boltzmann-Features folgendes gegeben:

```
#define LB
#define LB_BOUNDARIES
#define LB_ADAPTIVE
```

Ferner wurde in `lb-ggi.hpp` der Gitterdatentyp `LbGgiGrid` definiert als:

```
typedef ggi::P4estGrid<LbGgiPoint, LbGgiData> LbGgiGrid;
```

Dies erzeugt ein ESPResSo-Executable, in dem die adaptive Lattice-Boltzmann-Implementierung aktiv ist und auf der Gitterimplementierung `ggi::P4estGrid` für adaptive Gitter arbeitet.

7.1 Validierung

Als Ausgangspunkt für den Validierungstest dienten eine Poisseuille- und eine Couette-Strömung, beide auf einem periodischen Gitter mit Gittergröße S und ohne Verfeinerung an den Boundaries, für die I Iterationen gerechnet wurden. Für beide Strömungen wurde die Maschenweite mit 1.0, die Dichte mit 10.0, die Viskosität mit 100.0 und die Zeitschrittlänge mit 0.01 festgelegt. Die Boundaries lagen jeweils bei $x = 0.01S$ und $x = 0.99S$.

Für die Poisseuille-Strömung wurde die äußere Kraft auf $(0.0, 0.001, 0.0)$ gesetzt. Für die Couette-Strömung erhielt stattdessen die Boundary bei $x = 0.01S$ eine Geschwindigkeit von 0.01 als Randbedingung. Für beide Strömungen wurde je ein entsprechendes Tcl-Skript für ESPResSo aufgesetzt.

Die Tcl-Skripte wurden jeweils durch jede der drei ESPResSo-Executables mit der Gittergröße $S = 128$ ausgeführt, berechneten $I = 10000$ Iterationen und schrieben je eine VTK-Datei mit den Fließgeschwindigkeiten u.

Aus den VTK-Daten wurden jeweils in x-Richtung Geschwindigkeitsprofile abgetragen. Somit entstanden für jede Strömung drei Profilkurven, je eine für jedes ESPResSo-Executable, und wurden auf Deckungsgleichheit geprüft.

7.1.1 Ergebnisse

Poiseuille-Strömung

In Abbildung 7.2 sind die Profilkurven für die Poiseuille-Strömung abgetragen. Die durch *ESPResSo-Alt* erzeugte Profilkurve ist blau gezeichnet. Die jeweils durch *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* erzeugten Profilkurven sind zueinander komplett identisch und daher beide rot gezeichnet.

Es sind zwei Abweichungen erkennbar:

Rechtsversatz der roten Kurven Die jeweils durch *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* erzeugten Profilkurven sind gegenüber der durch *ESPResSo-Alt* erzeugten Referenzkurve in positiver x-Richtung versetzt.

Dies kommt dadurch zustande, dass *ESPResSo-Alt* beim Schreiben der VTK-Daten die Geschwindigkeit u für jeden Rechenpunkt auf geometrische Gitterpunkte abbildet, während *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* sie auf geometrische Gitterzellen abbilden. Dadurch ergibt sich ein Versatz von einer halben Maschenweite.

Niedrigere Höhe der roten Kurven Die jeweils durch *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* erzeugten Profilkurven erreichen im Strömungszentrum gegenüber der durch *ESPResSo-Alt* erzeugten Referenzkurve nicht diesselbe Höhe d. h. eine geringere maximale Geschwindigkeit.

Dies kommt dadurch zustande, dass *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* bei der Abbildung von Boundaries auf diskrete Gitterpunkte „aufrunden“, während *ESPResSo-GGI-Alt* „abrundet“. Hierdurch berechnen *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* die Strömung mit Boundaries, die je um eine Gitterpunktlage breiter sind, und somit in einem entsprechend schmaleren Kanal, in dem die erreichte maximale Geschwindigkeit im Strömungszentrum nach 10000 Iterationen etwas niedriger ist.

Zum Vergleich ist daher noch die Profilkurve einer durch *ESPResSo-Alt* gerechneten Poiseuille-Strömung mit den Boundaries bei $x = \frac{2}{128}S$ und $x = \frac{126}{128}S$ in grün eingezeichnet.

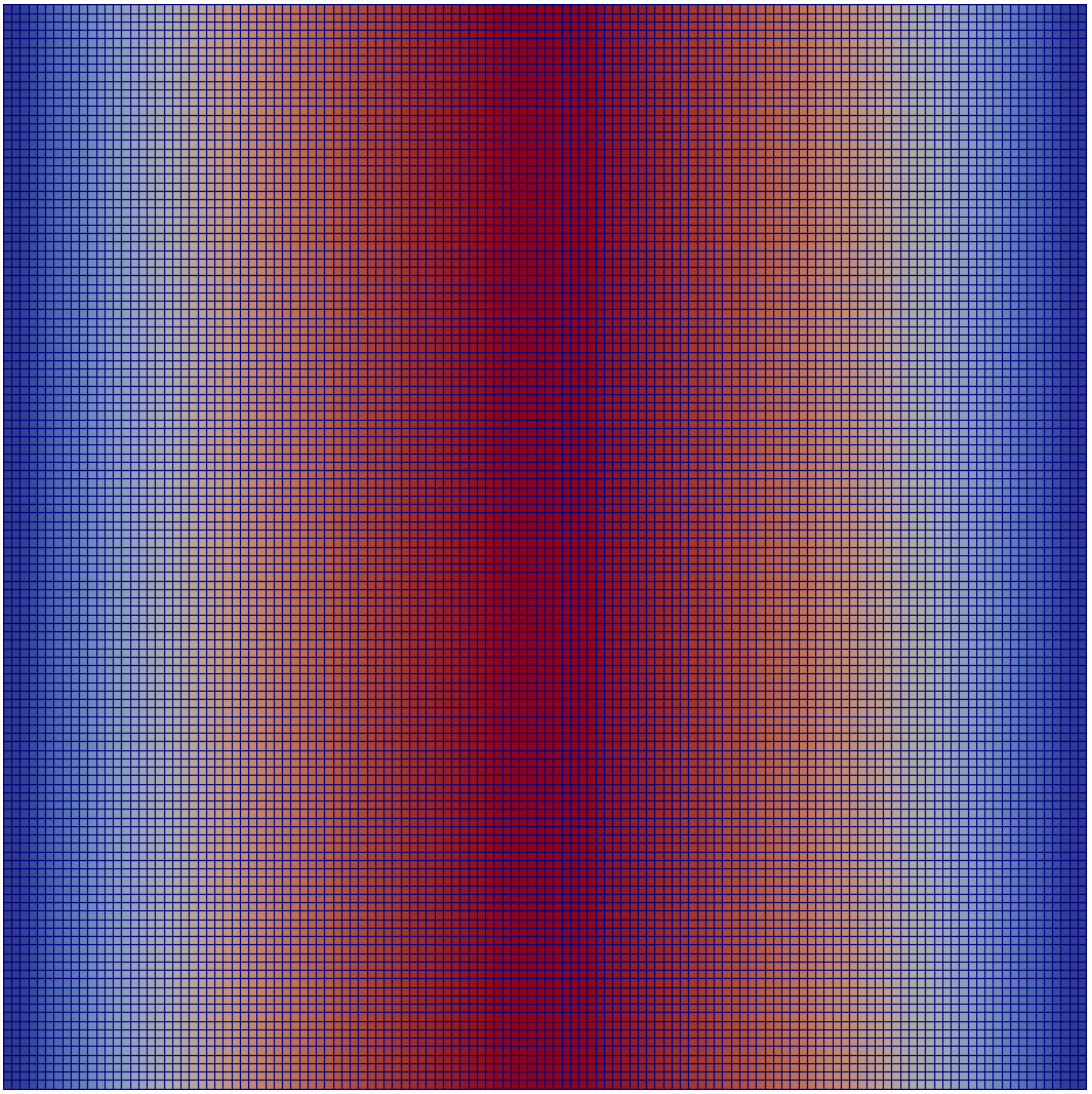

Abbildung 7.1: Schnittdarstellung einer Poiseuille-Strömung

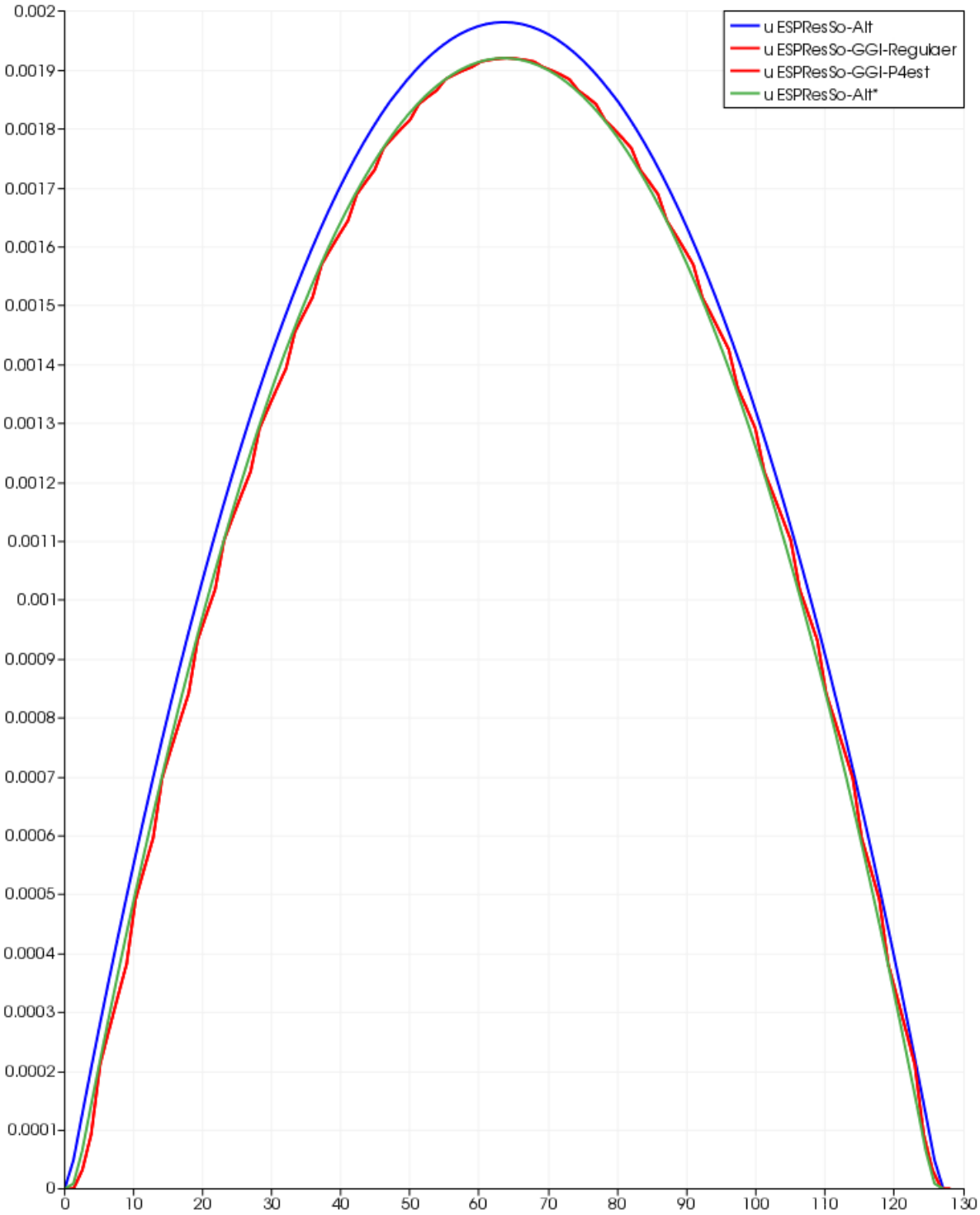

Abbildung 7.2: Profilkurven für die Poiseuille-Strömung im Vergleich

Couette-Strömung

In Abbildung 7.4 sind die Profilkurven für die Couette-Strömung abgetragen. Die durch *ESPResSo-Alt* erzeugte Profilkurve ist blau gezeichnet. Die jeweils durch *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* erzeugten Profilkurven sind zueinander komplett identisch und daher beide rot gezeichnet.

Wie bei der Poiseuille-Strömung ergibt sich ein Rechtsversatz der roten Profilkurven um eine halbe Maschenweite und die um je eine Gitterlage breiteren Boundaries. Letzteres führt dazu, dass durch den entsprechend schmaleren Kanal das Maximum der jeweils durch *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* erzeugten Profilkurven geringfügig über dem der durch *ESPResSo-Alt* erzeugten Referenzkurve liegt.

Zum Vergleich ist daher noch die Profilkurve einer durch *ESPResSo-Alt* gerechneten Couette-Strömung mit den Boundaries bei $x = \frac{2}{128}S$ und $x = \frac{126}{128}S$ in grün eingezeichnet.

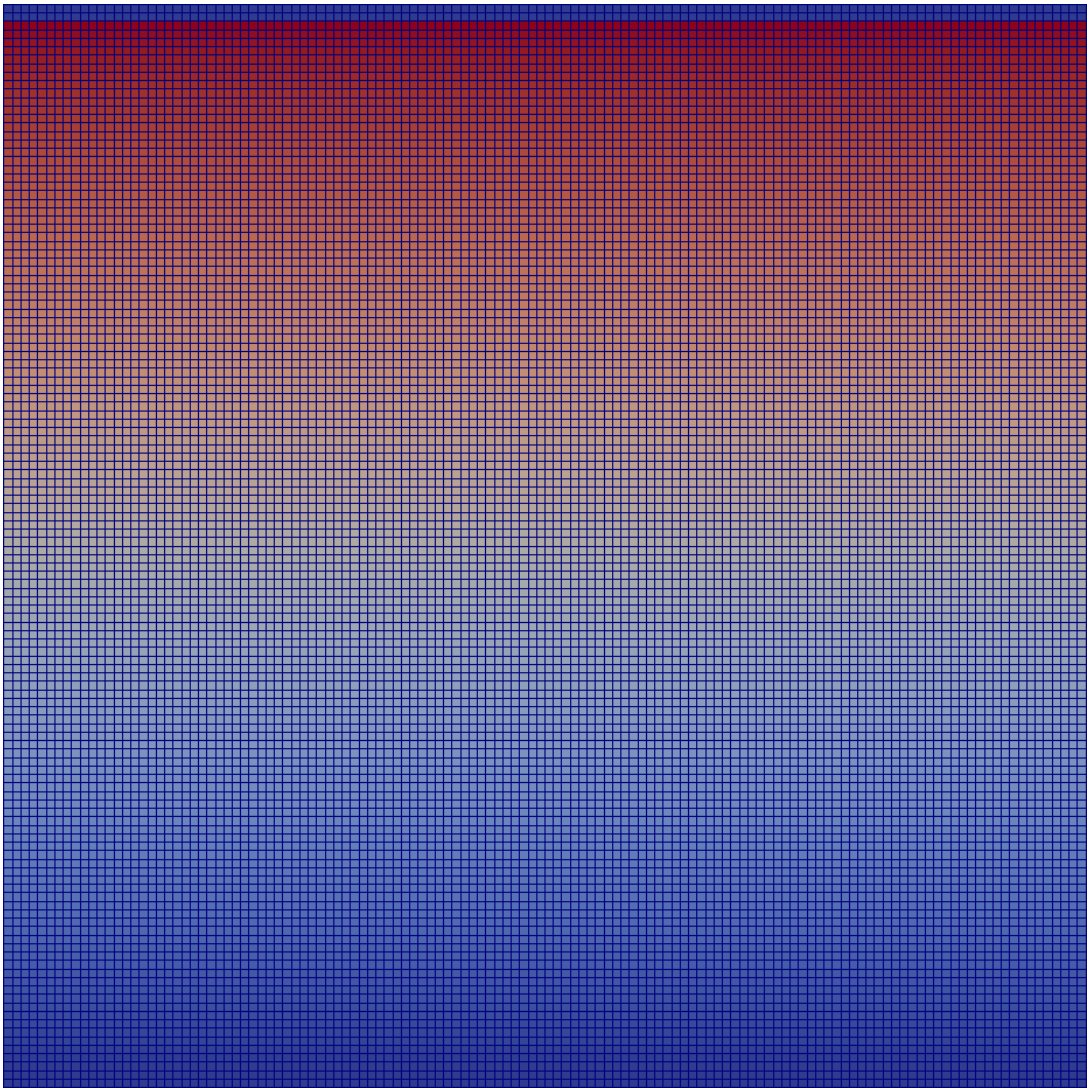

Abbildung 7.3: Schnittdarstellung einer Couette-Strömung

Abbildung 7.4: Profilkurven für die Couette-Strömung im Vergleich

Fazit

Die Abweichungen rühren daher, dass *ESPResSo-Alt* sowie *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* für das selbe Tcl-Skript geringfügig abweichende Strömungsszenarien rechnen und außerdem die Ergebnisse in leicht abweichender Form ausgeben.

7.2 Performance

Als Ausgangspunkt für den Performancetest diente eine Poiseuille-Strömung auf einem periodischen Gitter über P Kernen mit verschiedenen Gittergrößen S und einer maximalen Anzahl von Verfeinerungen L an den Boundaries, für die I Iterationen gerechnet werden. Die Boundaries lagen bei $x = 0.01S$ und $x = 0.99S$ und die äußere Kraft verlief in positiver y-Richtung. Hierzu wurde ein entsprechendes Tcl-Skript für ESPResSo aufgesetzt.

Zwei Testszenarien wurden geprüft:

Reguläres Szenario Dieses Szenario untersucht und vergleicht die Laufzeiten der drei ESPResSo-Executables, wenn sie die Poiseuille-Strömung ohne Verfeinerung an den Boundaries rechnen, sich also auf eine reguläre Auflösung beschränken.

Dabei werden die Laufzeiten von *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* jeweils mit denen von *ESPResSo-Alt* verglichen. Als jeweilige Laufzeit gemessen wird hierbei stets die verstrichene Zeit für die Ausführung von I Iterationen in μs.

Somit treten die alte, nicht-adaptive Lattice-Boltzmann-Implementierung mit ihrem regulären Gitter an gegen die neue, adaptive Lattice-Boltzmann-Implementierung, die über die generische Gitterschnittstelle ein Gitter nutzt, das im ersten Fall durch die GGI-Schnittstellenimplementierung `ggi::RegularGrid` und im zweiten Fall durch die Schnittstellenimplementierung `ggi::P4estGrid` bereitgestellt wird.

Die adaptive Lattice-Boltzmann-Implementierung entspricht für eine reguläre Auflösung - d. h. einer maximale Verfeinerung $L = 0$ - der entkoppelten Lattice-Boltzmann-Implementierung aus [Esc16]. Dadurch werden effektiv Gitterbereitstellung und -zugriff verglichen.

Das Tcl-Skript für die Poiseuille-Strömung wurde durch jede der drei ESPResSo-Executables jeweils auf $P = 1, 2, 4, 8, 16, 32, 64$ Kernen, mit Gittergrößen $S = 32, 64, 128, 256, 512$ und einer maximalen Anzahl von Verfeinerungen $L = 0$ ausgeführt, und berechnete $I = 5$ Iterationen.

Adaptives Szenario Dieses Szenario untersucht den erwarteten Nutzen aus der Gitteradaptivität, der sich ergeben müsste, wenn nur der Teil des Simulationsraums mit hoher Auflösung gerechnet werden muss, in dem dies erforderlich ist. Für die Poiseuille-Strömung ist dies die Umgebung der beiden Boundaries.

Rechnet die adaptive Lattice-Boltzmann-Implementierung auf einem adaptivem Gitter mit Größe $S = S_0$ und verfeinert die Gitterpunkte an den Boundaries $L = L_0$-mal, erziehlt sie dort eine Auflösung, für welche die nicht-adaptive Lattice-Boltzmann-Implementierung ein reguläres Gitter der Größe $S = S_0 2^{L_0}$ verwenden müsste.

Demensprechend vergleicht dieses Szenario die Laufzeiten von *ESPResSo-GGI-P4est* für eine Gittergröße $S = \frac{512}{2^L}$ jeweils mit denen von *ESPResSo-Alt* für eine Gittergröße $S = 512$. Um dem Supercycling Rechnung zu tragen, das die adaptive Lattice-Boltzmann-Implementierung durchführt, wurden die durchgeführten Iterationen I stets so gewählt, dass für jede maximale Anzahl Verfeinerungen L für die Gitterpunkte ohne Verfeinerung stets 5 Superzyklen der Länge $2^L\tau$ ausgeführt wurden.

Das Tcl-Skript für die Poiseuille-Strömung wurde durch *ESPResSo-GGI-P4est* auf $P = 16, 32, 64$ Kernen und mit den Konstellationen aus Gittergröße, maximaler Anzahl Verfeinerungen, Anzahl Iterationen $(S, L, I) = (32, 4, 80), (64, 3, 40), (128, 2, 20), (256, 1, 10)$ ausgeführt. Als Laufzeit gemessen wurde hierbei stets die verstrichene Zeit für die Ausführung der I Iterationen in μs.

Die Laufzeiten für die Ausführung des Tcl-Skripts durch *ESPResSo-Alt* auf $P = 16, 32, 64$ Kernen, mit Gittergröße $S = 512$, einer maximalen Anzahl von Verfeinerungen $L = 0$ und $I = 80, 40, 20, 10$ Iterationen wurden aus den bereits im Rahmen des regulären Testszenarios gemessenen Laufzeiten für die jeweilige Anzahl Kerne, Gittergröße, maximaler Anzahl Verfeinerungen und $I = 5$ Iterationen hochgerechnet.

7.2.1 Ergebnisse

Ergebnisse des regulären Testszenarios

In Abbildung 7.5 sind die Verhältnisse der Laufzeiten von *ESPResSo-GGI-Regulär* zu *ESPResSo-Alt* abgetragen. Von einigen Ausnahmen abgesehen liegen die Kurven für alle Gittergrößen über der 1.
ESPResSo-GGI-Regulär rechnet demnach langsamer als *ESPResSo-Alt*.

In Abbildung 7.6 sind die Verhältnisse der Laufzeiten von *ESPResSo-GGI-P4est* zu *ESPResSo-Alt* abgetragen. Die Kurven liegen für alle Gittergrößen überwiegend zwischen 1.5 und 2.5.
ESPResSo-GGI-P4est rechnet demnach z. T. erheblich langsamer als *ESPResSo-Alt*.

In den Abbildungen 7.7 und 7.8 sind die Laufzeiten *ESPResSo-GGI-Regulär* und *ESPResSo-GGI-P4est* abgetragen. Beide zeigen das erwartete Skalierungsverhalten.

Die Ausgangsdaten für die Abbildungen sind in Tabelle 7.1 aufgeführt.

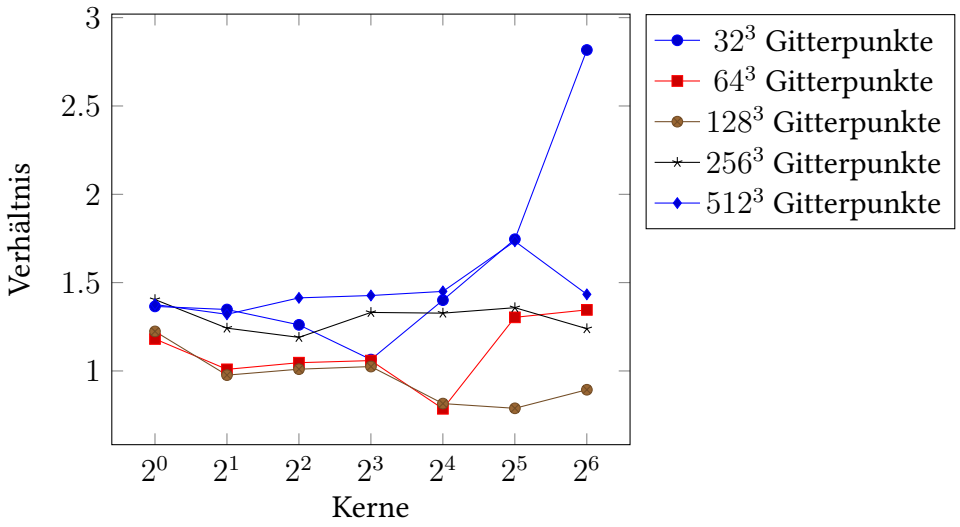

Abbildung 7.5: Verhältnis der Laufzeiten von ESPResSo-GGI-Regulär zu ESPResSo-Alt

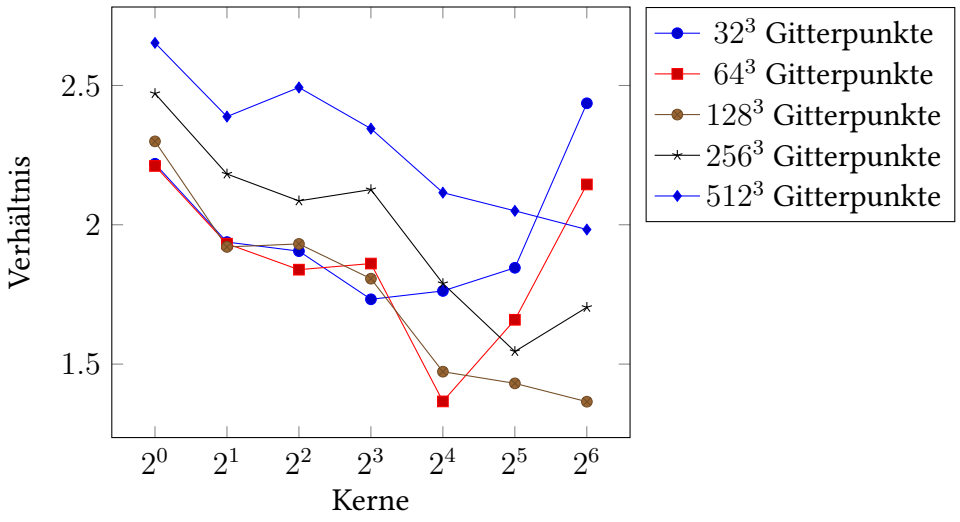

Abbildung 7.6: Verhältnis der Rechenzeiten von ESPResSo-GGI-P4est zu ESPResSo-Alt

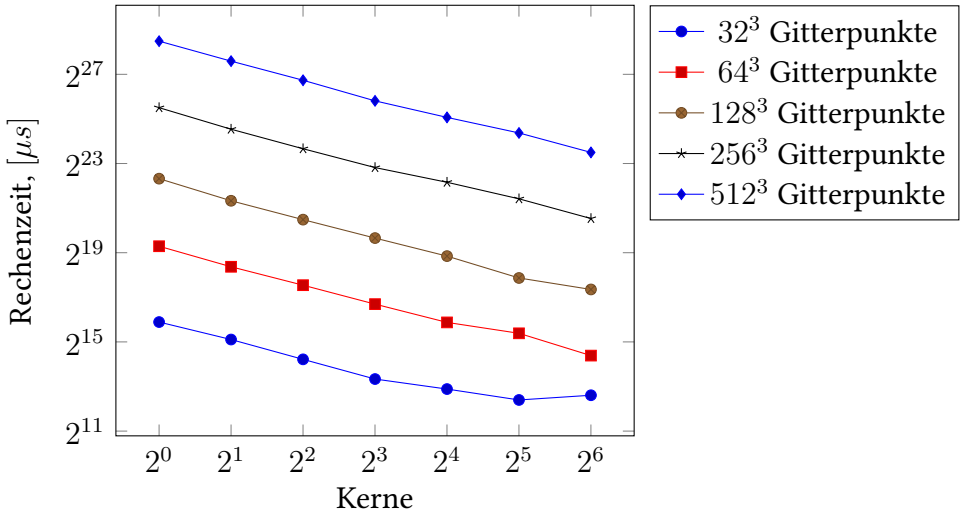

Abbildung 7.7: Rechenzeiten von ESPResSo-GGI-Regulär

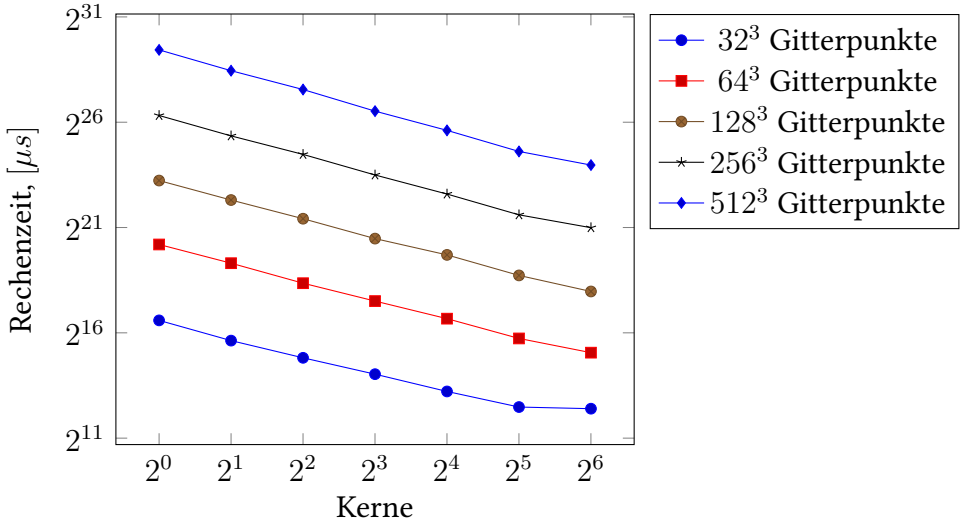

Abbildung 7.8: Rechenzeiten von ESPResSo-GGI-P4est

Ergebnisse des adaptiven Testszenarios

In Abbildung 7.10 sind die Verhältnisse der Laufzeiten von *ESPResSo-GGI-P4est* zu *ESPResSo-Alt* abgetragen. Die Kurven liegen für fast alle Gittergrößen und maximalen Verfeinerungen deutlich unter 0.1. Selbst für die Gittergröße mit minimaler Verfeinerung ist *ESPResSo-GGI-P4est* um mehr als vier Mal schneller.

Es zeigt sich erwartungsgemäß, dass sich für eine Simulation mit im Vergleich zum gesamten Simulationsgebiet kleinen Bereichen, die eine höhere Auflösung erfordern, ein deutlicher Performancevorteil aus der Möglichkeit ergibt, das Gitter in diesen Bereichen lokal zu verfeinern.

Die Ausgangsdaten für die Abbildung sind in Tabelle 7.2 aufgeführt.

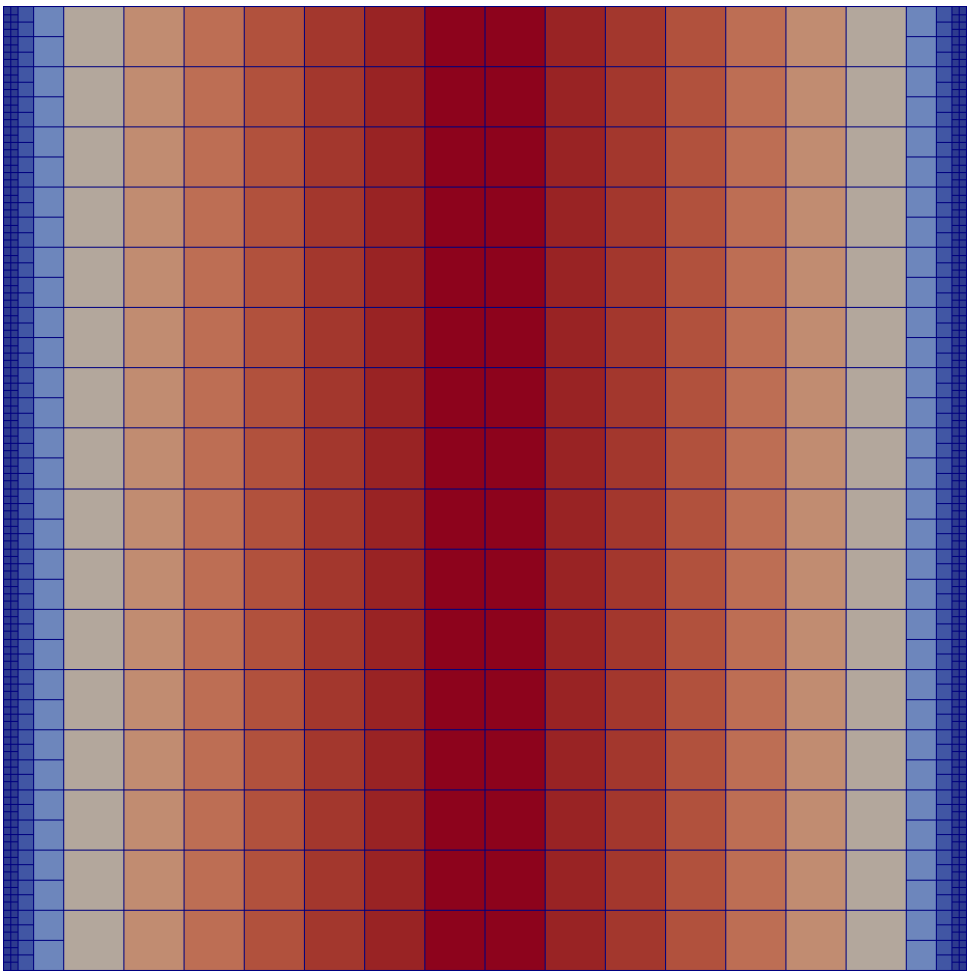

Abbildung 7.9: Schnittdarstellung einer Poiseuille-Strömung mit adaptiver Auflösung

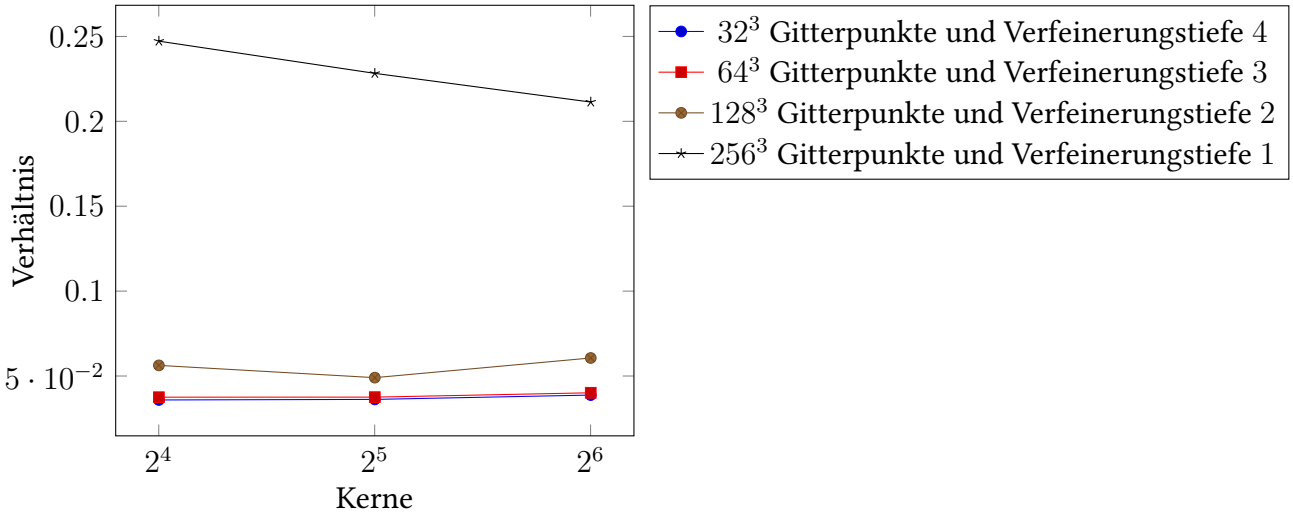

Abbildung 7.10: Verhältnis der Rechenzeiten von ESPResSo-GGI-P4est zu ESPResSo-Alt für das adaptive Szenario

Fazit

Sowohl *ESPResSo-GGI-Regulär* als auch *ESPResSo-GGI-P4est* rechnen für reguläre Auflösungen deutlich langsamer als *ESPResSo-Alt*. Der Zuwachs an Flexibilität und Gitteradaptivität durch das generische Gitterschnittstellensystem GGI erfolgt demnach um den Preis von Performanceeinbußen für solche Anwendungsszenarien, die diese Möglichkeiten nicht nutzen.

P	S	L	I	ESPResSo-Alt	ESPResSo-GGI-Regulär	ESPResSo-GGI-P4est
64	32	0	5	2213	6234	5391
64	64	0	5	15884	21375	34068
64	128	0	5	186923	167034	255163
64	256	0	5	1221389	1513602	2080994
64	512	0	5	8242367	11817564	16344384
32	32	0	5	3084	5384	5692
32	64	0	5	32790	42743	54401
32	128	0	5	302518	238553	432861
32	256	0	5	2056559	2793193	3178043
32	512	0	5	12478450	21640583	25580887
16	32	0	5	5394	7556	9507
16	64	0	5	76333	59985	104283
16	128	0	5	577389	470750	850328
16	256	0	5	3519772	4671434	6297522
16	512	0	5	24199549	35098091	51188216
8	32	0	5	9689	10315	16787
8	64	0	5	99968	105834	186028
8	128	0	5	804724	824736	1453777
8	256	0	5	5545386	7382342	11790837
8	512	0	5	41112698	58669353	96408185
4	32	0	5	15093	19031	28758
4	64	0	5	182210	190738	335059
4	128	0	5	1447921	1462480	2796367
4	256	0	5	11128307	13243499	23210774
4	512	0	5	78730572	111310535	196268985
2	32	0	5	26136	35229	50650
2	64	0	5	334969	338032	647196
2	128	0	5	2692678	2628932	5171341
2	256	0	5	19561652	24304116	42687409
2	512	0	5	152732750	201695040	364763551
1	32	0	5	44418	60648	98559
1	64	0	5	541792	639892	1197979
1	128	0	5	4270141	5226367	9818524
1	256	0	5	33807318	47469817	83540861
1	512	0	5	272704515	375318820	723433572

Tabelle 7.1: Die Parameterkonstellationen aller Durchläufe des regulären Testszenarios und die jeweils gemessenen Laufzeiten aller drei ESPResSo-Executables in $[\mu s]$

P	S	L	I	ESPResSo-GGI-P4est	ESPResSo-Alt(hochgerechnet)
64	256	1	10	3483957	16484734
64	128	2	20	1998422	32969468
64	64	3	40	2645484	65938936
64	32	4	80	5113371	131877872
32	256	1	10	5696334	24956900
32	128	2	20	2447566	49913800
32	64	3	40	3749724	99827600
32	32	4	80	7232853	199655200
16	256	1	10	11964486	48399098
16	128	2	20	5446165	96798196
16	64	3	40	7264706	193596392
16	32	4	80	13896831	387192784

Tabelle 7.2: Die Parameterkonstellationen aller Durchläufe des adaptiven Testszenarios und die gemessenen bzw. hochgerechneten Laufzeiten der beiden ESPResSo-Executables in $[\mu s]$

8 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde das in [Esc16] entwickelte *generische Gitterschnittstellensystem GGI* dahingehend überarbeitet und präzisiert, dass es den Anforderungen einer Lattice-Boltzmann-Implementierung gerecht wird, die auf einem adaptiven Gitter arbeitet. Zu einer solchen *adaptive Lattice-Boltzmann-Implementierung* wurde die *entkoppelte Lattice-Boltzmann-Implementierung* aus [Esc16] weiterentwickelt, die zwar bereits das Schnittstellensystem GGI nutzte, aber noch nicht auf einem adaptiven Gitter arbeitete.

Die adaptive Lattice-Boltzmann-Implementierung und das generische Gitterschnittstellensystem wurden in ESPResSo integriert. Die alte Lattice-Boltzmann-Implementierung von ESPResSo, die auf einem regulären Gitter arbeitete, wurde entfernt.

ESPResSo vermag nun Fluidvolumina durch die neue Lattice-Boltzmann-Implementierung mit regulärer sowie adaptiver Auflösung zu simulieren. Lediglich die mit der alten Lattice-Boltzmann-Implementierung integrierte Funktionalität zur Kopplung des simulierten Fluids mit der ESPResSo-Molekulardynamiksimulation konnte im Rahmen dieser Arbeit nicht wiederhergestellt werden.

Die Validierungstests ergaben, dass ESPResSo mit der neuen Lattice-Boltzmann-Implementierung für Fluidvolumina, die es mit regulärer Auflösung simuliert, die selben Resultate errechnet wie mit der alten Lattice-Boltzmann-Implementierung abweichlich einiger Ausgabe-Artefakte.

Die Performancetests zeigten auf, dass die neue Lattice-Boltzmann-Implementierung die Simulation von Fluidvolumina mit regulärer Auflösung im Vergleich zur alten Lattice-Boltzmann-Implementierung zwar verlangsamt, aber für Fluidvolumina mit adaptiver Auflösung die erwarteten Performancevorteile realisiert.

Ausblick

In einer Folgearbeit müsste die Fluid-Partikel-Kopplung, die zwischen der alten Lattice-Boltzmann-Implementierung und der Molekulardynamiksimulation in ESPResSo bestanden hat, auch für die neue Lattice-Boltzmann-Implementierung wiederhergestellt werden.

Die Herausforderung ergibt sich aus dem Umstand, dass ESPResSo die Molekulardynamiksimulation auf einem Linked-Cell-Gitter rechnet, das mit dem regulären Gitter der alten Lattice-Boltzmann-Implementierung auf triviale Weise in Deckung gebracht werden konnte:

Für beliebige Koordinaten innerhalb dieses Linked-Cell-Gitters, z. B. denen simulierter Partikel, konnte jeweils in $O(1)$ der nächstliegende Lattice-Boltzmann-Gitterpunkt im simulierten Fluidvolumen ermittelt und damit der Austausch von Kräften zwischen den beiden Simulationen vorgenommen werden.

Da die neue Lattice-Boltzmann-Implementierung auf einem adaptiven Gitter arbeitet, ist diese Zuordnung nicht mehr ohne weiteres in konstanter Zeit möglich, sondern vielmehr in mindestens $O(log\, n)$, wodurch die Kopplung die erzielbare Performance erheblich einschränken könnte.

Es müssten also zwei Vorgehensweisen geprüft werden:

Kopplung in $O(log\, n)$ Man nimmt den Performanceverlust in Kauf.

Hierfür müsste lediglich das generische Gitterschnittstellensystem GGI um die Möglichkeit erweitert zu werden, für gegebene Koordinaten nach den nächstliegenden GGI-Gitterpunkten zu suchen. Das Augenmerk müsste dann darauf liegen, diese Suche in garantiert $O(log\, n)$ zu implementieren.

Molekulardynamiksimulation auf einem GGI-Gitter Man nutzt für die Molekulardynamiksimulation ebenfalls ein durch GGI bereitgestelltes adaptives Gitter, das parallel zu dem GGI-Gitter der Lattice-Boltzmann-Simulation betrieben wird.

Jede Gitterzelle dieses GGI-Gitters hält eine feste Anzahl Speicherstellen für Partikel vor. Betritt ein Partikel eine Gitterzelle, deren Speicherstellen alle bereits belegt sind, verfeinert man diese Gitterzelle und verteilt die Partikeldaten auf die entstandenen Tochterzellen. Analog vergröbert man Gitterzellen, wenn genügend Partikel sie verlassen.

Hiermit könnte man auch die Rechenlast der Partikelsimulation besser auf mehrere Prozesse umzuverteilen.

Indem man Gitterzellen des GGI-Gitters der Partikel auf die würfelförmige Gitterabschnitte mit verfeinerter Substruktur, mit denen das GGI-Gitter der Lattice-Boltzmann-Simulation jeweils das selbe Simulationsvolumen abdeckt, abbildet, könnte man die maximale Anzahl der für die Kopplung zu durchsuchenden Partikel bzw. Fluidgitterpunkte auf einen konstanten Wert begrenzen. Um sicherzustellen, dass die durch die jeweiligen Gitterabschnitte abgebildeten Simulationsvolumina auf jedem Prozess zueinander deckungsgleich sind, müsste man die Partitionierung der beiden Gitter miteinander koppeln.

Hierzu müsste das generische Gitterschnittstellensystem dahingehend erweitert werden, dass würfelförmige Abschnitte eines GGI-Gitters „geschützt“ werden können i. d. S. dass sie bei einer Neupartitionierung nicht an einer Prozessgrenze aufgebrochen werden. Dies erforderte eine entsprechende Erweiterung an p4est, die es effektiv erlaubt, *Baumknoten* eines p4est-Waldes zu identifizieren und so zu markieren, dass `p4est_partition()` sie nicht aufbricht.

Es ist leicht einzusehen, dass ein solcher Baumknoten durch je einen Anfangs- und Endoktanten identifiziert werden kann. Inwiefern dies über Baumgrenzen hinweg funktionieren kann, müsste untersucht werden.

Literaturverzeichnis

[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: *Physical Review* 94.3 (1954), S. 511–525 (zitiert auf S. 13).

[BWG11] C. Burstedde, L. Wilcox, O. Ghattas. „p4est: Scalable algorithms for paralell adaptive mesh refinement on forests of octrees“. In: *SIAM Journal on Scientific Computing* 33 (Jan. 2011), S. 1103–1133 (zitiert auf S. 11, 52).

[Esc16] D. Eschbach. „Entwicklung einer generischen Gitterschnittstelle für ESPResSo“. Studienarbeit. Universität Stuttgart, IPVS/SGS, Sep. 2016 (zitiert auf S. 11, 14, 16, 18, 19, 21, 22, 27–29, 32–35, 38, 44, 47, 51, 75, 83, 91).

[LABM16] M. Lahnert, T. Aoki, C. Burstedde, M. Mehl. „Minimally-Invasive Integration of P4est in Espresso for Adaptive Lattice-Boltzmann“. In: *The 30th Computational Fluid Dynamics Symposium*. Japan Society of Fluid Mechanics, 2016 (zitiert auf S. 21, 24, 58, 64).

[LAMH06] H. Limbach, A. Arnold, B. Mann, C. Holm. „ESPResSo - an extensible simulation package for research of soft matter systems“. In: *Computer Systems Communications* 174 (Mai 2006), S. 704–727 (zitiert auf S. 11, 24, 25, 67–69, 71–73).

[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= (zitiert auf S. 11, 14–16).

[RKDA06] M. Rohde, D. Kandhai, J. J. Derksen, H. E. A. van den Akker. „A generic, mass conservative local grid refinement technique for lattice-Boltzmann schemes“. In: *International Journal for Numerical Methods in Fluids* 51.4 (2006), S. 439–468. URL: http://dx.doi.org/10.1002/fld.1140 (zitiert auf S. 16, 21).

[Sch08] U. Schiller. „Thermal fluctuations and boundary conditions in the lattice Boltzmann method“. Diss. Johannes Gutenberg University, Mainz, 2008 (zitiert auf S. 11, 14).

[Suc01] S. Succi. *The lattice boltzmann equation*. Oxford University Press, 2001 (zitiert auf S. 13, 14).

Alle URLs wurden zuletzt am 08. 03. 2017 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