Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Hans-Joachim Bungartz
Betreuer:	Dr.-Ing. Martin Bernreuther
begonnen am:	01.01.2004
beendet am:	02.07.2004
CR-Klassifikation:	I.6.3, D.1.3, G.1.0

Diplomarbeit Nr. 2169

Entwicklung eines modularen Softwarepaketes für reaktive massiv parallele dreidimensionale Lattice-Boltzmann Strömungsberechnungen

Fabian Kaiser

Institut für Parallele und
Verteilte Systeme
Universität Stuttgart
Fakultät Informatik,
Elektrotechnik und Informationstechnik
Universitätsstraße 38
D–70569 Stuttgart

Inhaltsverzeichnis

KAPITEL 1

Abkürzungsverzeichnis

Symbol	**Bedeutung**
$\vec{e_i}$	(diskrete) mikroskopische Teilchengeschwindigkeiten
f	Verteilungsfunktion für Teilchenaufenthalts-Wahrscheinlichkeiten
$f^{(0)}$	Gleichgewichtsverteilung
$\vec{F}$	äußere auf Teilchen einwirkende Kraft
L	charakteristische Länge
p	Druck
Re	Reynoldszahl
Re^*	Element-Reynoldszahl
ρ	Fluiddichte
τ	Relaxationszeit
t	Zeit
$\vec{u}$	makroskopische Geschwindigkeit
$\vec{\xi}$	(kontinuierliche) mikroskopische Geschwindigkeit
$\vec{x}$	Vektor der den Aufenthaltsort eines Teilchens bestimmt

KAPITEL 2

Einleitung und Zielsetzung

Ziel dieser Arbeit war die *Entwicklung eines modularen Softwarepaketes für reaktive massiv parallele dreidimensionale Lattice-Boltzmann Strömungsberechnungen*. Mit Hilfe der Simulation von Strömungen lassen sich die verschiedenartigsten Erkenntnisse für technisch relevante und vielfach noch unzureichend erforschte Zusammenhänge erlangen, um so ein tieferes Verständnis für die Materie zu erhalten. Beispiele hierfür sind Sickerwasserströmungen, der Luftfluss um ein Tragflächenprofil, aber auch Probleme der Biomechanik wie der Blutfluss im System der Kapillargefäße. Derartige Probleme sind nicht nur rein akademischer Natur, sondern spielen auch in der Industrie bzw. der industriellen Forschung eine zunehmend stärkere Rolle.

Anhand der einzelnen Bestandteile der Themenstellung soll hier einleitend auf die zu realisierende Funktionalität eingegangen werden:

Mit dem *Lattice Boltzmann*-Verfahren existiert ein Lösungsmodell für Strömungsprobleme, das sich, wenn auch nicht notwendigerweise in der dahinter stehenden Theorie, so doch in der Implementierung äußerst einfach gestaltet.
Die Forderung nach *Parallelität* wird klar, wenn man sich vor Augen führt, welch hohen Rechenaufwand schon die Simulation von kleinen Strömungsfeldern - insbesondere im dreidimensionalen Fall - fordert. Der Zusatz „massiv“ besagt dabei, dass es sich hier nicht nur um einen Einsatz des Programms auf „kleinen“ Zwei- oder Vierprozessormaschinen handelt, sondern auch um den Einsatz auf Hochleistungsrechnersystemen und Clustern.
Anhand der *reaktiven* Komponente unterscheidet sich diese Arbeit sicher am deutlichsten von den meisten anderen *Lattice Boltzmann*-Implementierungen. Ziel war es, weg zukommen vom klassischen Ablauf Preprozessor-Berechnung-Postprozessor, hin zu einem dynamischen System, in dem der Benutzer während des Simulationslaufs Änderungen an Einstellungen und/oder der Geometrie des Strömungsgebietes machen kann.
Der *modulare* Charakter einer Software zeichnet sich immer dann aus bzw. wird vermisst, wenn nach einer anfänglichen Entwicklungsphase im Laufe der Zeit Änderungen oder Erweiterungen am Quell-

code gemacht werden sollen. Je modularer eine Software ist, desto eher zeigt eine Codeänderung nur Auswirkungen in dem jeweils modifizierten Modul und berührt nicht das ganze System. Zudem erleichtert eine modulare Struktur i.d.R. das Verständnis sowie die Übersicht über das System, wenn dies auch manchmal auf Kosten der Leistungsfähigkeit in Form von Rechengeschwindigkeit geschieht.

Diese Punkte sind in dem Zusammenhang zu sehen, dass mit der vorliegenden Diplomarbeit die Grundlage für weiterführende Arbeiten in Form von Studienprojekten, Fachstudien oder weiteren Diplomarbeiten gelegt werden sollte. Ein wesentlicher Gedanke in der Implementierungsphase war daher stets, möglichst ausführlich zu dokumentieren, welche Funktion die entwickelten Klassen und Methoden jeweils besitzen. Teilweise wurde auch zugunsten des besseren Codeverständnisses und der übersichtlicheren Codestruktur die eine oder andere Optimierung bewusst nicht durchgeführt, um in ohnehin schon schwierigen Code- oder Strukturabschnitten nicht zusätzliche Hürden für das Verständnis aufzubauen, bzw. um die Modularität nicht unnötig einzuschränken.

Zum Ende der Arbeit hin zeichnete sich bereits der erste Einsatz des Systems im Rahmen eines Softwarepraktikums ab, mit dem Ziel einer Visualisierungs- und Interaktionskomponente. Damit sind die Einsatzmöglichkeiten aber nicht erschöpft: Es ist durchaus denkbar, das System auch im Rahmen eines Computational Steering Projekts einzusetzen, und anhand komplexer Hardware die Visualisierung der Daten sowie die Steuerung der Simulation durchzuführen. Voraussetzung hierfür ist dann aber in jedem Fall entsprechende Rechenleistung, da ohne sie ein interaktives Benutzen der Software wenig Sinn ergibt.

2.1. Reaktivität vs. Interaktivität

Um den in der Themenstellung verwendeten Begriff der „reaktive[n] [...] Strömungsberechnung“ gab es einige Verwirrungen, da dieser Begriff Verschiedenes vermuten lassen kann, was tatsächlich in keiner Weise in dieser Arbeit behandelt wird.
Thema dieser Arbeit war die parallele Strömungssimulation unter Einsatz der *Lattice Boltzmann*-Methode, erweitert um Komponenten, die es einem Benutzer ermöglichen sollten, während des Simulationslaufes Änderungen an bestimmten Rahmenparametern vorzunehmen. Der Begriff „reaktiv“ sollte sich dabei darauf beziehen, dass eine *Aktion* des Benutzer gemäß dem Gesetz „actio et reactio“ eine *Reaktion* des Systems bewirkt. Er wurde in Anlehnung an die von Amir Pnueli beschriebenen „reactive systems“[13] gewählt. Diese zeichnen sich gegenüber anderen Systemen dahingehend aus, dass sie nicht nach dem Start eine Dateneingabe verlangen, daraufhin die Berechnung durchführen und sich dann beenden, sondern sich in einer permanenten Interaktion mit dem Benutzer befinden. Als *Reaktive Systeme* in diesem Sinne werden häufig Steuerungen in Embedded Systems bezeichnet, aber auch Anwendungen wie Reservierungssysteme, kurz alles, was das permanente Reagieren auf einen Aktor (Mensch oder technische Komponente) erfordert und eine lange, wenn nicht (idealerweise oft) niemals endende Laufzeit hat.

Dem wurde entgegengesetzt, dass der Begriff der „reaktiven Strömungsberechnung“ doch sehr dem Terminus der „Berechnung reaktiver Strömungen“ ähnele. *Reaktive Strömungen* haben nun aber in der Tat nichts mit der vorliegenden Arbeit gemein, so dass an dieser Stelle noch einmal explizit darauf hingewiesen sei, auch wenn sich der Bedeutungskontext natürlich aus den folgenden Kapiteln ergibt.

Beides, sowohl die Verwendung des Begriffs „reaktive Berechnung“ in der Themenstellung, als auch dessen Ablehnung haben eine gewisse Berechtigung. Ihn einfach durch „interaktive Berechnung“ zu ersetzen, schien jedoch auch nicht der richtige Weg zu sein. *Interaktivität* impliziert gemeinhin etwas dialogartiges, wobei die Dialogpartner jeweils im wechselseitigen Ausschluss operieren. Genau das sollte aber nicht bezweckt werden. Vielmehr war die Intention, ein System zu entwickeln, das auf einem hohen Geschwindigkeitslevel die Simulation berechnet und gegebenenfalls auf Benutzereinwirkung dahingehend reagiert, dass die veränderten Parameter im weiteren Verlauf berücksichtigt werden. Totzeiten sollten dabei aber keine entstehen bzw. wenn nicht zu vermeiden, so doch möglichst gering gehalten werden.

Letztlich wurde relativ pragmatisch beschlossen, die Themenstellung nicht nachträglich zu ändern, sondern an dieser Stelle auf die Problematik hinzuweisen, da der Begriff des „reaktiven Systems“ hier durchaus als zutreffend angesehen wird. Die Gefahr der Verwechslung besteht in der Tat, jedoch sollte sie mit diesen Worten ausgeräumt sein. Unter diesem Gesichtspunkt werden beide Begriffe in dieser Arbeit verwendet, jeweils in dem Kontext, in welchem - nach obigen Ausführungen - entweder der Fokus auf Interaktion oder auf der Reaktivität liegt.

2.2. Themenüberblick

Die vorliegende Arbeit besitzt die folgende Struktur:

In Kapitel 3 wird eine kurze Einführung in das Thema *Lattice Boltzmann* gegeben. Es werden die Zusammenhänge zwischen grundlegenden Gleichungen der Strömungslehre und den im *Lattice Boltzmann*-Verfahren zum Einsatz kommenden aufgezeigt bzw. angedeutet.

Kapitel 4 beschreibt die Rahmenbedingungen unter denen das Zielsystem zu entwickeln war. Dieses wird in groben Zügen vorgestellt um einen Gesamteindruck zu verschaffen und das Verständnis der nachfolgenden detaillierten Beschreibungen zu erleichtern. Zudem werden softwaretechnische Gesichtspunkte angesprochen.

Kapitel 5 begründet die im Kapitel 4 angedeuteten Entwurfsschritte und beschreibt die Realisierung der einzelnen Komponenten. Zusätzlich werden Messungen zur Performance präsentiert und analysiert.

Abschließend wird in Kapitel 6 eine Zusammenfassung der Arbeit und ihrer Ergebnisse gegeben, sowie eine Liste der Probleme angesprochen, die im Rahmen dieser Arbeit nicht oder nicht abschließend bearbeitet werden konnten.

KAPITEL 3

Theorie des Lattice-Boltzmann Verfahrens

Da es sich bei dieser Arbeit weniger um eine theoretische Abhandlung über das *Lattice Boltzmann*-Verfahren handelt, sondern eher die Parallelität und Reaktivität im Vordergrund stehen, wird an dieser Stelle auf eine detaillierte Vorstellung und Herleitung des *Lattice Boltzmann*-Verfahrens verzichtet und statt dessen auf die Literatur [8][9] verwiesen. Dennoch sollen hier kurz die wesentlichen Gleichungen und ihre Beziehungen angesprochen werden, um den physikalischen und mathematischen Hintergrund darzulegen, wobei dies im Wesentlichen eine Zusammenfassung der o.g. Literatur darstellt.

3.1. Die Boltzmann-Gleichungen

Im Bereich der Strömungssimulation werden in der Regel zur Berechnung von einfachen Fluiden (Materialeigenschaften unabhängig vom Fließzustand) die Navier-Stokes Gleichungen oder deren Verwandte eingesetzt. Die einzusetzenden Ergänzungsterme für komplexere Strömungsprobleme (z.B. thermisches Mehrphasen-Mehrkomponentensystem) sind jedoch oft unbekannt bzw. müssen experimentell bestimmt werden.
Um solchen Problemen zu begegnen, erscheint ein vordergründig komplett verschiedener Ansatz überlegenswert - die Boltzmann-Gleichung. Sie beschreibt aus einer mikroskopischen Betrachtungsweise die raum-zeitliche Entwicklung von Teilchenaufenthaltswahrscheinlichkeiten. Nur „vordergründig komplett verschieden“ ist dieser Ansatz daher, weil zwar die Betrachtungsweise eine andere ist, beide Verfahren sich aber ineinander überführen lassen. Es lässt sich zeigen, dass die Boltzmann-Gleichung unter bestimmten Einschränkungen in die Navier-Stokes Gleichungen überführt werden kann, womit man für das gleiche Problem und dessen gleiche Lösung ein alternatives Lösungsmodell, zusätzlich zu den Navier-Stokes Gleichungen bekommt.
Unter bestimmten Umständen sind also Lösungen der Boltzmann-Gleichung auch Lösungen der Navier-

Stokes Gleichungen. Findet man eine Möglichkeit, die Boltzmann-Gleichung mit geringem Rechenaufwand zu lösen, so kann diese Zeitersparnis gegenüber dem aufwendigen Lösen der Navier-Stokes Gleichungen evtl. die Nachteile durch die oben angedeuteten Einschränkungen überwiegen.

Boltzmann-Gleichung

$$\frac{\partial f}{\partial t} + \vec{\xi} \cdot \frac{\partial f}{\partial \vec{x}} + \vec{F} \cdot \frac{\partial f}{\partial \vec{\xi}} = \Omega(f) \tag{3.1}$$

Grundlage dieser Überlegungen bildet, wie bereits angesprochen, die Boltzmann-Gleichung 3.1.
Die Verteilungsfunktion $f(t, \vec{x}, \vec{\xi})$ bestimmt dabei die Wahrscheinlichkeit, ein Teilchen zum Zeitpunkt t mit der Geschwindigkeit $\vec{\xi}$ am Ort $\vec{x}$ anzutreffen (und damit bei gegebener Dichte die Anzahl der Teilchen mit diesen Eigenschaften). Die linke Seite der Gleichung beschreibt die durch Bewegung der Partikel und äußere Einwirkung hervorgerufenen Änderungen der Verteilungsfunktion, wohingegen durch den Kollisionsoperator auf der rechten Seite die Interaktion der Teilchen modelliert ist.

Unter Voraussetzung eines isothermen Systems lässt sich nun das hier nicht im Detail dargestellte komplizierte Kollisionsintegral dahingehend vereinfachen, dass die Kollision durch eine sich der lokalen Maxwellverteilung annähernde Verteilungsfunktion nachgebildet wird:

BGK-Kollisionsoperator

$$\Omega = -\frac{1}{\tau} \cdot (f - f^{(0)}) \tag{3.2}$$

Damit vereinfacht sich die Boltzmann-Gleichung (3.1), wobei die Geschwindigkeit der Annäherung durch die Relaxationszeit τ geregelt wird, welche anschaulich als mittlere Zeitdauer zwischen dem Zusammenstoß zweier Teilchen betrachtet werden kann.
Dieser Ansatz geht zurück auf Bhatnagar, Gross und Krook (daher der Name „BGK“) und setzt voraus, dass kein Interesse an den Details des Kollisionsprozesses besteht, sondern nur Aussagen über das Gesamtverhalten der Teilchen gemacht werden sollen. Das Einsetzen des BGK-Operators führt bei Vernachlässigung äußerer Kräfte zur vereinfachten Boltzmann-Gleichung:

Vereinfachte Boltzmann-Gleichung

$$\frac{\partial f}{\partial t} + \vec{\xi} \cdot \frac{\partial f}{\partial \vec{x}} = -\frac{1}{\tau} \cdot (f - f^{(0)}) \tag{3.3}$$

Führt man nun eine Diskretisierung der Gleichung 3.3 bzgl.des mikroskopischen Geschwindigkeitsraumes durch, so erhält man die diskrete Boltzmann-Gleichung 3.4, in der die Verteilungsfunktion $f(t, \vec{x}, \vec{\xi})$ durch die Werte $f(t, \vec{x}, \vec{e_i})$ an i ausgewählten Kollokationspunkten dargestellt wird.

Diskrete Boltzmann-Gleichung

$$\frac{\partial f_i}{\partial t} + \vec{e_i} \cdot \frac{\partial f_i}{\partial \vec{x}} = -\frac{1}{\tau} \cdot (f_i - f_i^{(0)}) \quad |i = 0..n-1 \tag{3.4}$$

Für die weitere Diskretisierung mit Finiten Differenzen in Raum und Zeit lassen sich interessante Eigenschaften der diskreten Boltzmann-Gleichung ausnutzen, was über den Zwischenschritt 3.5

$$\frac{f_i(t+\Delta t,\vec{x})-f_i(t,\vec{x})}{\Delta t}+c\frac{f_i(t+\Delta t,\vec{x}+\vec{e_i}\Delta x)-f_i(t+\Delta t,\vec{x})}{\Delta x}=-\frac{1}{\tau}(f_i(t,\vec{x})-f_i^{(0)}(t,\vec{x})) \quad (3.5)$$

durch Wählen von $\Delta x = c\Delta t$ letztlich zur Lattice-Boltzmann-Gleichung (3.6) führt:

Lattice-Boltzmann-Gleichung

$$f_i(t+\Delta t,\vec{x}+\vec{e_i}\Delta t)=f_i(t,\vec{x})-\frac{\Delta t}{\tau}(f_i(t,\vec{x})-f_i^{(0)}(t,\vec{x})) \quad |i=0..n-1 \quad (3.6)$$

3.2. Gleichgewichtsverteilung

Für die in den Navier-Stokes Gleichungen geforderte Massen- und Impulserhaltung wären beim direkten Einsatz der Maxwellverteilung unendlich viele Kollokationspunkte nötig. Um dennoch brauchbare Ergebnisse zu erzielen, wird die Maxwellverteilung in einer Reihe bis zur zweiten Ordnung um die absolute Gleichgewichtsfunktion bezüglich der Geschwindigkeit entwickelt, was jedoch eine Beschränkung des Verfahrens auf kleine Machzahlen ($Ma \ll 1$) impliziert.

$$f_i^{(0)}(t,\vec{x})=t_p(\rho+\rho_0(\frac{e_{i\alpha}u_\alpha}{c_s^2}+\frac{u_\alpha u_\beta}{2c_s^2}(\frac{e_{i\alpha}e_{i\beta}}{c_s^2}-\delta_{\alpha\beta}))) \quad |i=0..n-1 \quad (3.7)$$

Mit
$c_s = \frac{1}{\sqrt{3}}c$, ($c$ Geschwindigkeit, die die Größenordnung der Schallgeschwindigkeit angibt)
$\delta_{\alpha\beta} = \begin{cases} 1 & \text{für } \alpha=\beta \\ 0 & \text{sonst} \end{cases}$
$\rho = \sum_i f_i(\vec{x},t)$
$\vec{u} = \sum_i f_i(\vec{x},t)\cdot\vec{e_i}$
t_p :

Modell	t_0	t_1	t_3
D2Q9	$\frac{4}{9}$	$\frac{1}{9}$	$\frac{1}{36}$
D3Q15	$\frac{2}{9}$	$\frac{1}{9}$	$\frac{1}{72}$
D3Q19	$\frac{1}{3}$	$\frac{1}{18}$	$\frac{1}{36}$

Tabelle 3.1.: Faktoren zur Berechnung von $f_i^{(0)}$, t_p abhängig vom Modell und jeweiligem i

Kollokationspunkte ($\{\vec{e_i}\}$)

Für die Wahl der Kollokationspunkte ist ausschlaggebend, dass unter den gegebenen Bedingungen die Navier-Stokes Gleichungen aus Gleichung 3.6/3.7 hergeleitet werden können, sowie die durch $\{\vec{e_i}\}$

gebildete Einzelzelle sich raumfüllend fortsetzen lässt.
Die in der Literatur am häufigsten eingesetzten Punkte werden durch die Modelle *D2Q9*, *D3Q15* und *D3Q19* beschrieben, wobei *D* die Anzahl der Dimensionen und *Q* die Anzahl der Kollokationspunkte pro Einzelzelle angibt.

D2Q9:
$$\begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{pmatrix}$$

D3Q15:
$$\begin{pmatrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \end{pmatrix}$$

D3Q19:
$$\begin{pmatrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \end{pmatrix}$$

3.3. Randbedingungen

Auch auf eine ausführliche Beschreibung der Randbedingungen soll hier verzichtet werden, da allein dieses Thema Stoff für eigenständige Arbeiten bietet. Eine knappe Übersicht über die relevanten Bedingungen sowie die Art und Weise, wie sie in der vorliegenden Arbeit realisiert wurden, soll an dieser Stelle genügen. Für detaillierte Informationen sei auf die Literatur[8] verwiesen.

Es existieren im Wesentlichen drei Arten von Randbedingungen:

Periodische Randbedingungen

Handelt es sich um Simulationsprobleme, die in einer Achsrichtung periodisch mit der Periodenlänge α sind, so kann man sich auf die Berechnung einer Periode beschränken. Die physikalischen Größen müssen dann an beiden Rändern übereinstimmen. In der vorliegenden Arbeit wird das dahingehend ausgenutzt, dass z.B. der Abfluss am jeweiligen „rechten“ Rand gleich dem Einfluss am gegenüberliegenden „linken“ Rand ist (sowie umgekehrt).

Haftrandbedingung

Diese Bedingung fordert, dass die Geschwindigkeit des Flusses gleich der Geschwindigkeit des Randes ist. Handelt es sich bei dem Rand um feste unbewegliche Elemente, so impliziert dies eine Geschwindigkeit $\vec{u} = 0$. Häufig wird als einfache Realisierung dieser Forderung der *Bounceback*-Algorithmus gewählt, bei welchem in Richtung der Wand zeigende Verteilungen an dieser antiparallel reflektiert werden. Physikalisch liegt dem die Annahme zugrunde, dass sich ein Teilchen aufgrund der Wandrauhigkeit nach dem Aufprall in einer zur Einlaufrichtung unkorrelierten Auslaufrichtung fortbewegt. Dies bewirkt im zeitlichen Mittel näherungsweise den Impuls $\vec{0}$. Allerdings ist die Realisierung von beliebigen Wandgeometrien dadurch hochgradig aufwendig, da in diesem Fall nicht davon ausgegangen werden kann, dass der Rand einer Zelle immer mit der Wand zusammenfällt. Um

diese Komplexität für die vorliegende Arbeit - die klar nicht die Optimierung von Randbedingungen als Ziel hat - zu reduzieren, wird hier stets eine Wand auf die Ränder der Normzellen abgebildet, was allerdings gewisse Verfälschungen des Ergebnis fördert.

Druck-Geschwindigkeitsrandbedingung

Sie legt fest, wie der Druck bzw. die Geschwindigkeit am Rand des Strömungsfeldes vorgegeben werden kann. Für diese Arbeit wurde eine einfache Methode gewählt, die allerdings den Nachteil hat, dass die Druckwelle das gesamte Strömungsfeld mehrere Male durchlaufen muss, bis sich brauchbare Ergebnisse abzeichnen (siehe nächster Abschnitt, „Randbedingungen setzen“)

3.4. Simulationsablauf

Der Ablauf einer Simulation gestaltet sich demnach relativ einfach:

Listing 3.1: Simulationsablauf

```
initDomain();
setInitialBoundaryConditions();
for (int i=0; i<maxTimeSteps; i++) {
  // accelerate fluid
  setBoundaryConditions(i);
  // start collision phase
  performCollision(i);
  // start propagation phase
  performPropagation(i);

  // dump current results
  if (0 == (i % postProcInterval))
    doPostProcessing(i);
}
cleanup();
```

Randbedingungen setzen

In der Phase *setBoundaryConditions* kann das Fluid durch eine simulierte äußere Krafteinwirkung beschleunigt werden. Die einfachste Möglichkeit, dies zu erreichen ist, die Verteilungen der Fluid-Zellen am Einfluss zu modifizieren. Dabei wird zu den in das Feld zeigenden Verteilungen ein Beschleunigungswert a addiert; von den aus dem Feld herausweisenden Verteilungen wird dieser Wert subtrahiert.

$a = Beschleunigungsfaktor \cdot \rho \cdot t_{p_i}$

Kollision

In der Kollisionsphase wird die Interaktion der Fluid-Teilchen untereinander bzw. mit Hindernissen berechnet. Im einfachsten Fall (siehe 3.3) wird hierbei je nach Typ der jeweiligen Zelle (Hindernis- oder Fluidzelle) ein Bounceback bzw. die Relaxation berechnet.

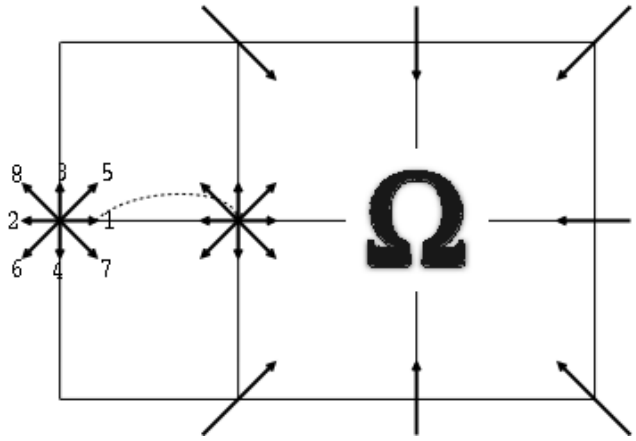

Abbildung 3.1.: Kollisionsberechnung der eingehenden Verteilungen (D2Q9)

Beim Bounceback auf Hinderniszellen werden die räumlich entgegengesetzen Verteilungen ausgetauscht, wohingegen sich die Relaxation wie folgt berechnet:

1. Berechnen der Dichte $\rho = \sum_i f_i$
2. Berechnen der Geschwindigkeit $\vec{u} = \sum_i f_i \cdot \vec{e_i}$
3. Einsetzen dieser Werte in Gleichung 3.7 und berechnen von $f_i^{(0)}$
4. Neue Verteilungen gemäß $f_i = f_i - \frac{\Delta t}{\tau}(f_i - f_i^{(0)})$ berechnen

Diese Schritte werden für jede im Strömungsgebiet enthaltene Zelle durchgeführt, wobei die Reihenfolge in der die einzelnen Zellen bearbeitet werden für die Korrektheit des Verfahren unerheblich ist, da ohnehin nur Daten der jeweiligen Zelle in die Berechnung eingehen, diese also lokal erfolgt.

Propagation

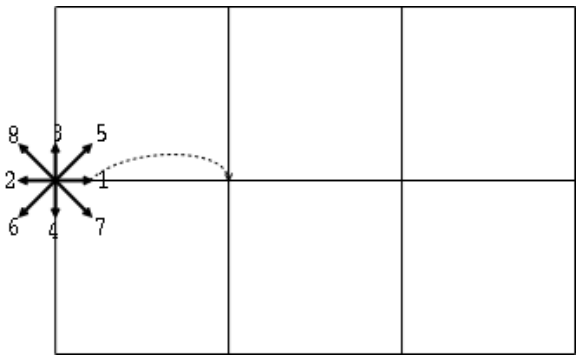

Abbildung 3.2.: Propagation der Verteilung f_1 (D2Q9)

In der Propagationsphase werden die in der Kollisionsphase neu berechneten Verteilungen f_i auf die benachbarten Zellen kopiert und überschreiben damit deren alte Werte. Das Kopieren erfolgt dabei auf diejenige Nachbarzelle, welche in Richtung der jeweiligen Verteilung liegt (f_1 wird also in Abb. 3.2 nach rechts kopiert, f_5 nach rechts oben etc.).

Hier spielt die Reihenfolge in der die Zellen bearbeitet werden eine Rolle, da durch naives Anwenden dieses Schemas z.B. in der Ordnung nach aufsteigendem X die Verteilung $f_1((1,0,0)^T)$ durch $f_1((0,0,0)^T)$ überschrieben wird bevor dieser Wert auf $f_1((2,0,0)^T)$ kopiert werden konnte. Details sind Kapitel 5.1.1 zu entnehmen.

KAPITEL 4

Konzeption und Durchführung

Nach der Themenvorstellung und der Einführung in das *Lattice Boltzmann*-Verfahren, soll hier zunächst in groben Umrissen ein Überblick über das entwickelte Gesamtsystem gegeben werden, um im Weiteren das Verständnis für die Zusammenhänge bei der Beschreibung der jeweiligen Komponenten zu erleichtern.

Strukturell wurde das System als Client-Server Software entworfen, um eine logische und räumliche Entkoppelung von Simulationskern und Visualisierungs- bzw. Steuerungskomponente zu erreichen. Dementsprechend gliedern sich die Funktionalitäten gemäß der Zuordnung zu diesen Komponenten: auf Seiten des Simulationsservers erfolgt die rechenintensive Simulation, ggf. auf mehreren Prozessoren, wohingegen die Clientseite lediglich eine Schnittstelle zur Interaktion mit dem Benutzer darstellt. Beiden Komponenten gemein ist eine Kommunikationsebene, anhand derer Daten ausgetauscht werden können.
Der Server wiederum gliedert sich in weitere zwei Untereinheiten, von denen die eine, der Rechenkern (siehe 5.1), die Berechnung der Simulation durchführt und die andere sich um Aufgaben wie Synchronisation mit dem Client kümmert, also die eigentliche Serverfunktionalität implementiert (siehe 5.3). Das Gegenstück zu Letzterer bildet auf Clientseite eine Schnittstelle, die dafür konzipiert wurde, beliebige Clientanwendungen an den Server zu binden (siehe 5.3.5). Diese Komponente abstrahiert weitestgehend von allen Fragen der Kommunikation und Synchronisation mit dem Server und bietet dem Anwendungsprogrammierer eine Highlevel-API für den Zugriff auf die Serverfunktionalität.

4.1. Rahmenbedingungen

Die Entwicklung des Systems unterlag von Beginn an einigen Rahmenbedingungen, die Einfluss auf die Struktur und Arbeitsweise des Endprodukts hatten.
Zuallererst wäre hier der Einsatz unter Unix zu nennen. Diese Forderung ergab sich aus dem Ziel,

die Simulation nicht nur auf einfachen PC-Architekturen auszuführen, sondern auch auf Hochleistungsrechnern mit ihr zu arbeiten, um zum einen die Skalierbarkeit zu testen und zum anderen auch tatsächlich Ergebnisse für komplexere Probleme in angemessener Zeit zu erhalten. Da im Bereich der Hochleistungsrechner nahezu ausschließlich auf Unix-Systemen gerechnet wird, war diese Bedingung quasi unumstößlich. Zugleich implizierte sie aber auch, dass der Code auf unterschiedlichsten Unix Derivaten und Hardwarearchitekturen lauffähig sein musste, angefangen vom Linux-PC auf dem die Entwicklung erfolgte, über Cluster bis hin zu (kombinierten) SharedMemory Systemen, aus denen heutige Hochleistungsrechner in der Regel aufgebaut sind.

Während eine derartige Bedingung zwangsläufig die Entwicklung aufwendiger macht, so hatte sie auch ihre positiven Seiten: Da für Unix Systeme deutlich mehr frei verfügbare Softwarepakete vorhanden sind als z.B. für Windows-Architekturen, lassen sich derartige Pakete legal und ohne Lizenzkosten einbinden um Entwicklungszeit zu sparen (siehe dazu auch Kapitel 5.6).

Für die parallele Berechnung auf nicht SharedMemory System war MPI[10] als Kommunikationssystem vorgegeben. Da es sich bei MPI letztlich um *den* Standard für Nachrichtenaustausch im Bereich von Hochleistungsrechner im Allgemeinen und numerischer Strömungssimulation im Besonderen handelt, war die Bedingung eher formaler Natur. Lediglich die Einschränkung auf MPI v. 1 hatte einige Konsequenzen im Bereich von Multithreading (Kapitel 5.2.4) und der Datenausgabe (Kapitel 5.4.1), warf jedoch keine unüberwindbaren Probleme auf.

Da diese Arbeit von Anfang an als Basis für nachfolgende Diplomarbeiten und Studienprojekte gedacht war, war eine der grundlegenden Anforderungen die hohe Wartbarkeit des Codes. Dies schließt nach Pressman[14] neben einer möglichst einfachen Korrektur von Fehlern auch die leichte Anpassbarkeit an neue oder erweiterte Problemstellungen ein. Um dieses Ziel erfüllen zu können, war ein softwaretechnisch strukturiertes Vorgehen nötig (siehe auch 4.2).

In diesem Zusammenhang kam während der Entwicklungsphase seitens des Betreuers die Idee auf, den entstandenen Code nicht nur für Studenten und Angehörige der Universität Stuttgart zugänglich zu machen, sondern ihn allgemein jedem Interessierten zur Verfügung zu stellen. Als Grundlage dafür bot sich die GPL[3] der *Free Software Foundation* an. Quellcode, der unter der GPL veröffentlicht wird, kann von jedermann frei und für nicht-kommerzielle Projekte auch kostenlos verwendet werden; dennoch behält der Autor das Copyright und veröffentlichte Änderungen müssen ebenfalls unter die GPL gestellt werden.
Im Hinblick auf die freie Verwendbarkeit durch Dritte war dann aber sicherzustellen, dass keine Lizenzbedingungen anderer eingesetzter Softwarepakete und Bibliotheken mit den Bedingungen der GPL kollidierten, andernfalls wäre die Nutzbarkeit wieder eingeschränkt.

4.2. Softwaretechnische Aspekte

Wie oben bereits angedeutet, hatte diese Arbeit einigen Anspruch an das softwaretechnische Vorgehen, da ihr Ergebnis als Grundlage für weitere Projekte genutzt werden sollte. Als Vorgehensweise bot sich ein klassisches Linear Sequentielles bzw. Wasserfallmodell (Abb. 4.1) an, da das Gesamtprojekt relativ überschaubar war und keine großartigen Überraschungen zu erwarten waren, die z.B. ein iteratives Anpassen an sich ändernde Spezifikationen und Entwürfe erfordert hätten.

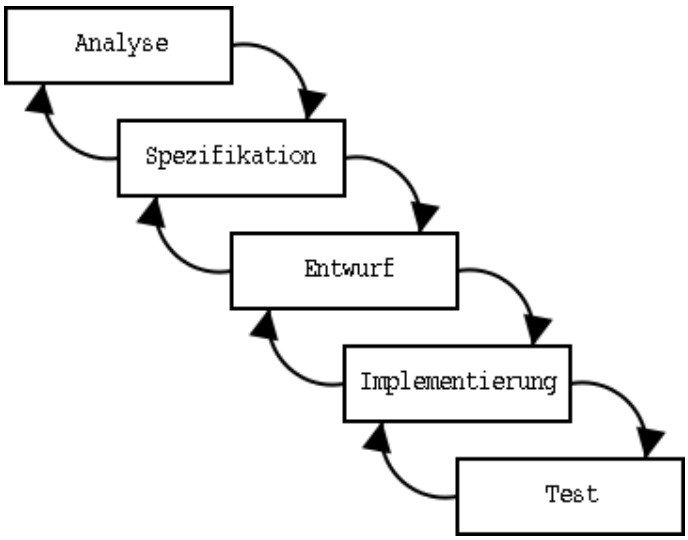

Abbildung 4.1.: Wasserfallmodell als Modell des Entwicklungsprozesses

Als von diesem Modell abweichend wurde nur die Implementationsphase geplant. Diese musste aufgrund der Anforderungen zwangsläufig iterativ ablaufen, da nicht zu erwarten war, dass durch das Implementieren eines im Vorfeld gemachten Entwurfs die rechenzeitkritischen Komponenten eine als akzeptabel anzusehenden Leistung erbringen würden. Dies wäre unrealistisch gewesen, da eben jene Komponenten stark hardwareabhängig sind und hier zu viele Unbekannte durch die hohe Komplexität der Hardware auftauchen, um diese im Vorfeld exakt analysieren und einordnen zu können (siehe dazu vor allem 5.2).

In der *Analyse*-Phase wurde hauptsächlich vorhandener Code auf die Eignung im Rahmen der Problemstellung hin gesucht und bewertet. Der von Jörg Bernsdorf entwickelte beispielhafte *Lattice Boltzmann*-Code „ANB“[1] erschien dabei als eine gute Grundlage, um sich mit dem *Lattice Boltzmann*-Verfahren vertraut zu machen. Im Rahmen eines Art Vorprojekts wurde der in Fortran geschriebene 2D-Code von ANB nach C++ übersetzt und um die Fähigkeit zur Berechnung einer dritten Dimension erweitert.

Die *Spezifikation* gestaltete sich ausnahmsweise äußerst einfach, da seitens der Aufgabenstellung/des Betreuers relativ wenige konkrete Anforderungen gestellt wurden und somit ein hohes Maß an entwerferischer und entwicklerischer Freiheit gegeben war.
Es wird an dieser Stelle auf die Wiedergabe einer gesamten *Spezifikation* verzichtet und nur deren Kernaussagen angeführt:

- Leistungsfähiger *Lattice Boltzmann* Code zur parallelen Berechnung von 3D Strömungsproblemen auf mehreren Prozessoren/Rechnerknoten
- Reaktive/Interaktive Komponente, um auf Änderungen durch den Benutzer reagieren zu können sowie Zwischenergebnisse online zu visualisieren
- Architekturunabhängig entwickelt in C++
- Hoher Dokumentationsanteil

Diese Anforderungen wurden in der darauf folgenden *Entwurfsphase* in ein Softwaredesign übertragen, das durch eine starke Modularität versucht, die Kopplung der einzelnen Programmteile so stark wie möglich zu reduzieren, um eine einfache Erweiterbarkeit bzw. Ersetzbarkeit der jeweiligen Module zu ermöglichen. Der Entwurf und seine Realisierung sind in Kapitel 5 zusammen mit der *Implementierung* dargestellt.

Die *Implementierung* gestaltete sich nun erwartungsgemäß relativ aufwendig im Vergleich zu den vorangegangenen Phasen. Zuerst wurde im sequentiellen Code mit verschiedenen Datenstrukturen experimentiert und deren Geschwindigkeitsverhalten sowie Speicherbedarf ermittelt und bewertet (Kapitel 5.1.1). Auf Basis der dort gewonnenen Erkenntnisse wurde der Code dann parallelisiert und nach diversen Modifikationen des Feinentwurfs in seine endgültige Form gebracht. Wie oben bereits angedeutet, war in der *Implementierungsphase* das iterative Vorgehen durchaus geplant und notwendig: es hat sich im Laufe der Arbeit bestätigt, dass heutige Hardware- und Betriebssystemarchitekturen derart aufwendig und komplex sind, dass sich ein Einbeziehen aller relevanten Randbedingungen kaum vernünftig realisieren lässt, weshalb man hier stark auf ein experimentelles Vorgehen angewiesen ist.

Dem *Test* wurde keine eigene Phase gewidmet, da dieser im Rahmen der Implementierung kontinuierlich durchgeführt wurde. Es ist leicht einzusehen, dass für die Optimierung eines Simulationscodes hinsichtlich höherer Geschwindigkeit viele Rechenläufe durchgeführt werden müssen und deren Ergebnisse nicht nur im Hinblick auf hohe Rechengeschwindigkeiten selbst, sondern auch auf die Korrektheit der Ergebnisse hin untersucht werden müssen. So gesehen war der Anteil an Tests relativ hoch, wenn auch abschließend keine groß angelegte Testphase mehr durchgeführt wurde.

KAPITEL 5

Realisierung der einzelnen Komponenten

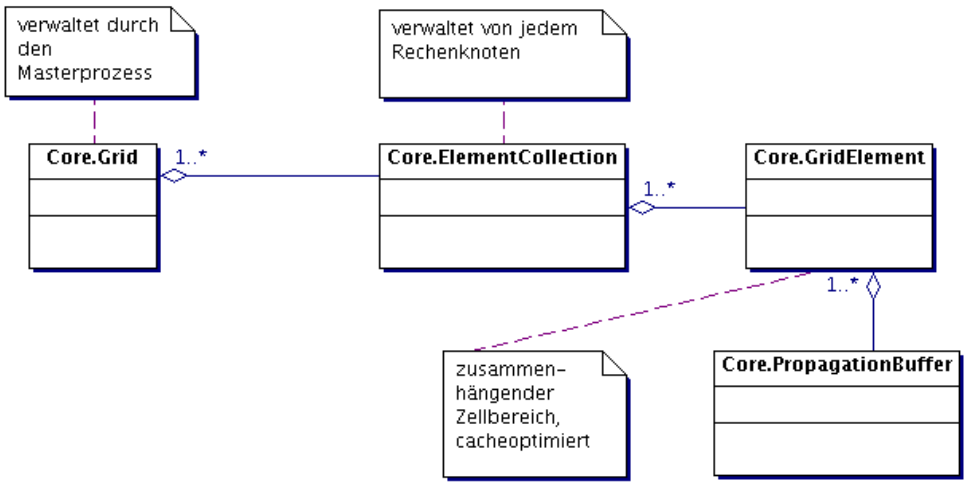

Abbildung 5.1.: Zusammenhang der parallelen Komponenten

Abb. 5.1 beschreibt einen Teil der Struktur des entworfenen Simulationsprogramms. Zuerst wird im folgenden Kapitel 5.1 die zentrale Rechenkomponente erläutert (Core.GridElement). Daraufhin wird dargelegt, wie zu Optimierungszwecken ein Strömungsgebiet in Einheiten von mehreren Teilgebieten (Core.ElementCollection) aufgeteilt und diese miteinander verbunden werden können (Core.PropagationBuffer). In Kapitel 5.2 werden dann Aspekte der Parallelisierung behandelt und nachfolgend die reaktiven/interaktiven Komponenten dargestellt.

5.1. Der Rechenkern

Zentrale Komponente einer jeden Simulationssoftware ist in der Regel eine Art Rechenkern, der losgelöst von administrativen Aufgaben wie Datenein- und ausgabe oder Kommunikationsaufgaben, die eigentliche Umsetzung der mathematisch- physikalischen Lösungsvorschriften darstellt.
Eine derartige Entkopplung ist, wo möglich, zu forcieren, da auf diese Art und Weise eine starke Modularisierung erreicht werden kann. Diese wiederum stellt eine zwingende Voraussetzung für die Erweiter- und Anpassbarkeit der Software dar. Zudem erleichtert eine strukturelle Trennung dieser Aufgaben auch deren jeweilige Optimierung, da zumindest bis zu einem gewissen Grad Änderungen einer Komponente lokal bleiben und nicht notwendigerweise in Wechselwirkung mit anderen Modulen treten, was die Analyse ihrer jeweiligen Wirkung erleichtert.

5.1.1. Datenstrukturen

Grundlegenden Einfluss auf die Leistungsfähigkeit des Rechenkerns - soll heißen auf die Geschwindigkeit, mit der die Berechnung durchgeführt wird - hat die Wahl und Implementierung der Datenstrukturen zur Speicherung des Strömungsfeldes. Das *Lattice Boltzmann*-Verfahren benötigt im Wesentlichen drei Arten von Informationen zu jeder Zelle:

- Die Teilcheninformationen für jede der i Verteilungen
- Art der Zelle (Fluid- oder Hinderniszelle)
- Information über die Speicherorte der Nachbarzellen

Entscheidend für die Berechnungsgeschwindigkeit ist dann im Wesentlichen ob die Datenstruktur es (effizient) unterstützt, die unnötige Ausführung von Operationen (z.B. Kollisionsberechnung auf einer Hinderniszelle) zu erkennen und zu verhindern, sowie die Art des Zugriffs auf die benötigten Informationen (direkte Adressierung bzw. Adressierung über eine oder mehrere Indirektionen) und der Grad der Angepasstheit an Vektor- und Cachingverhalten und der CPU. Für diese Arbeit wurden zwei Datenstrukturen, die für die oben gestellten Aufgaben in Frage kommen, implementiert und analysiert: eine Matrixstruktur sowie eine Vektorstruktur.

5.1.1.1. Matrixstruktur

Das Konzept der Matrixstruktur ergibt sich direkt aus der bildlichen Betrachtung des in Normzellen aufgeteilten Strömungsgebietes:
Die Zellen des d-dimensionalen Strömungsgebietes werden in ein d-dimensionales Array abgebildet. Dabei hat jedes Array-Element in etwa folgende Struktur:

```
typedef struct Cell {
  CellType cellType; // Hindernis, Fluid, Rand
  double f[<#Verteilungen>];
  ...
};
```

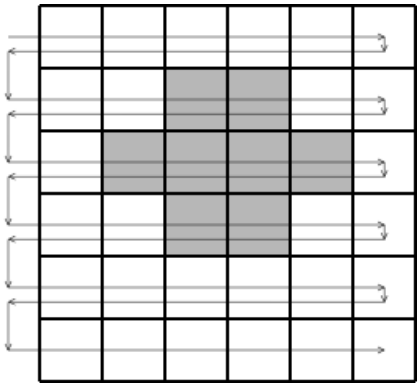

Abbildung 5.2.: Anordnung und Berechnungsreihenfolge mit der Matrix-Struktur (2D)

Der Vorteil dieser Datenstruktur liegt klar in seiner Einfachheit:

- Alle zur Berechnung benötigten Daten liegen in einem zusammenhängenden Speicherbereich, was zu einer besseren Ausnutzung der schnellen CPU-Caches führen kann, im Vergleich zu im Speicher verteilten Daten (näheres siehe 5.1.2).
- Es wird kein zusätzlicher Speicher für das Halten von in der Propagationsphase notwendigen Nachbarschaftsinformationen benötigt. Der Arrayindex einer jeden Zelle kann direkt aus ihren Koordinaten berechnet werden.
- In einer einfachen Schleife über alle Zellen kann die Kollision berechnet werden (Abb. 5.2) indem in jedem Schritt der Zeiger auf die aktuelle Zelle durch eine günstige Integer-Addition auf die nächste Zelle verschoben wird (für die Propagation gilt prinzipiell das Gleiche, Details über Unterschiede sind 5.1.2 zu entnehmen).

Allerdings birgt diese Regelmäßigkeit auch Nachteile:

- Unabhängig von der Art der Zelle (Hindernis oder Fluid) wird die gleiche Anzahl an Speicherwörtern für jede Zelle alloziert (andernfalls wäre die Berechnung der Speicherposition aus den Zellkoordinaten nicht mehr möglich). Das wiederum bedeutet, dass bei einem hohen Anteil von Hindernissen am gesamten Strömungsgebiet auch ein großer Teil des verwendeten Speichers für die Berechnung überhaupt nicht relevant ist.
- Jede der für die Berechnung irrelevanten Zellen (Zellen, die sich im Inneren eines Hindernisses befinden und keine Fluidzelle als Nachbar haben) muss in jedem Berechnungsschritt dennoch besucht werden. Ist dabei die Unterscheidung Hindernis-/Fluidzelle in der Kollisionsphase noch notwendig für die Wahl zwischen Relaxation und (hier) Bounceback, so führt eine weitere Unterscheidung zwischen Rand- und innerer Hinderniszelle zu keiner Performancesteigerung. Zwar würde sowohl in der Propagation als auch beim Bounceback etwas Rechenzeit eingespart, diese Einsparung (tatsächlich nur wenige Kopierschritte) wird aber durch die schlechtere Ausnutzung der CPU-Pipelines kompensiert. Faktisch reduziert man also die Performance des Systems entweder durch Ausführung von unnötigen aber das Ergebnis nicht verändernden Rechenschritten (Bounceback und Propagation auf inneren Hinderniszellen haben keine sichtbare Wirkung) oder durch das Brechen der CPU Pipelines wegen zusätzlichen Codeverzweigungen.

5.1.1.2. Vektorstruktur

Die Vektorstruktur versucht nun, die Nachteile der Matrixstruktur zu kompensieren. In erster Näherung bedeutet das, dass nur solche Zellen erfasst werden die tatsächlich für die Berechnung relevant sind.
Dazu werden die Zellen nicht wie in der Matrixstruktur nach ihrer räumlichen xyz-Ordnung im Speicher benachbart abgelegt, sondern sortiert nach ihrem Zelltyp - Fluid- oder Randzelle. Für innere Hinderniszellen wird dabei kein zusätzlicher Speicher benötigt.

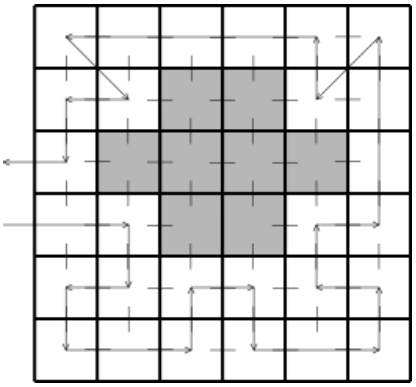

Abbildung 5.3.: Anordnung und Berechnungsreihenfolge mit der Vektor-Struktur (2D)

Die Kollisionsphase unterteilt sich nun in zwei Unterphasen:

- In einer Schleife über alle Fluidzellen wird die Relaxation berechnet
- In einer Schleife über alle Randzellen wird die Kollision mit dem Hindernis (Bounceback) berechnet

Schwieriger gestaltet sich hingegen die Propagation:
Da die Speicheradresse einer Zelle nicht mehr aus ihren Koordinaten berechnet werden kann, muss die Information wo sich eine Nachbarzelle befindet separat gespeichert werden, ansonsten kann keine Propagation stattfinden.

Wie bereits angedeutet, hat auch die Vektorstruktur sowohl Vor- als auch Nachteile:

Vorteile:

- Es wird kein Speicher für Zellen benötigt, die nicht zum Ergebnis der Berechnung beitragen (innere Hinderniszellen)
- Die Kollision kann unter hoher Ausnutzung der CPU-Pipelines gerechnet werden, da eine Unterscheidung zwischen Rand- und Fluidzelle implizit mit der Wahl der zu berechnenden Zellmenge zu Beginn des Schleifendurchlaufs getroffen wird

Nachteile:

- Der wesentliche Nachteil besteht darin, dass Nachbarschaftsinformationen nicht implizit durch die Datenstruktur gegeben sind. Dieser Mangel lässt sich auf zwei Arten beheben:

- Einführen einer Matrix, die pro Eintrag anstatt der gesamten Zelle nur einen Zeiger auf die von diesem Eintrag repräsentierte Zelle enthält. Zusätzlich wird in jeder Zelle die globale Koordinate gespeichert.
- Speichern von Zeigern auf die Nachbarzellen in jeder Zelle

Beide Ansätze erlauben es, zu jeder Zelle die Benachbarten zu berechnen, wenn auch mit unterschiedlichem Speicher- und Adressierungsaufwand.

5.1.2. Optimierung

Beide Datenstrukturen lassen gegenüber den oben skizzierten Grundprinzipien verschiedene Optimierungen zu.

5.1.2.1. Optimierung der Matrixstruktur

In beiden Strukturen stellt sich das Problem, dass bei naiver Implementierung des *Lattice Boltzmann*-Algorithmus für jede Zelle das Doppelte des Speicherplatz für die Verteilungen reserviert werden muss:

```
typedef struct Cell {
  double f[<#Verteilungen>];
  double c[<#Verteilungen>]; // <- Kopierfeld
  ...
};
```

Die Notwendigkeit dieses Kopierfeldes ergibt sich aus der Propagation. Wird hier in einer Schleife über alle Zellen jeweils die Propagation durchgeführt, so werden dabei noch nicht propagierte Werte teilweise überschrieben:

Listing 5.1: Fehlerhafte Propagation in 2D (3D erfolgt analog)

```
for (int y=0; y<dy; y++) {
  for (int x=0; x<dx; x++) {
    ...
    cell[x-1, y]->f[LEFT]  = cell[x, y]->f[LEFT];
    cell[x+1, y]->f[RIGHT] = cell[x, y]->f[RIGHT];
    ...
  }
}
```

Hier wird also in der nächsten (rechten) Zelle eine Verteilung überschrieben, bevor diese aus der nächsten in die übernächste kopiert werden konnte.

Einsparen des Kopierfeldes

Um dieses Problem zu umgehen lässt sich nun besagtes Kopierfeld einführen, was den Speicherverbrauch annähernd verdoppelt. Dabei werden alle Verteilungen statt direkt auf das eigentliche Ziel

zuerst auf das Kopierfeld geschrieben. Nach Abschluss sämtlicher Kopieroperationen erfolgt das Umsetzen auf die eigentlichen Verteilungen.
Alternativ wählt man die Strategie der richtungsabhängigen Propagation. Hier werden in jedem Schritt nur Propagationen ausgeführt, die in Richtung der bereits besuchten Zellen weisen. Um dennoch alle Propagationen auszuführen sind daher zwei Schleifen notwendig:

Listing 5.2: Korrekte Propagation in 2D (3D erfolgt analog)

```
for (int y=0; y<dy; y++) {
  for (int x=0; x<dx; x++) {
    ...
    cell[x, y-1]->f[TOP]   = cell[x, y]->f[TOP];
    cell[x-1, y]->f[LEFT] = cell[x, y]->f[LEFT];
    ...
  }
}

for (int y=dy; y-->0; ) {
  for (int x=dx; x-->0; ) {
    ...
    cell[x, y+1]->f[BOTTOM] = cell[x, y]->f[BOTTOM];
    cell[x+1, y]->f[RIGHT]   = cell[x, y]->f[RIGHT];
    ...
  }
}
```

Die zweite Schleife ließe sich einsparen wenn man statt einer aktuellen Zelle zwei Zellen gegeneinander laufen lässt (wieder von (0,0) und (dx-1, dy-1)). In Laufzeitmessungen führte dies jedoch zu schlechteren Ergebnissen, da offensichtlich mehr Cachefaults produziert werden und die Adressierung der zweiten Zelle etwas mehr Rechenaufwand erfordert. Für der Implementierung der Matrixstruktur wurde daher die richtungsabhänge Propagation mit zwei Schleifen realisiert, da diese ein gutes Verhältnis zwischen Rechenleistung und Speicherverbrauch bietet.

Propagationspuffer

Neben der Reduktion des Speicherbedarfs kann auch der Rechenaufwand gegenüber einer naiven Implementierung deutlich verringert werden.
Zuallererst wäre hier der Einsatz von Puffern an den Rändern des Strömungsgebietes zu nennen (Abb. 5.4).

Das bedeutet, dass um das eigentliche Strömungsgebiet herum eine weitere Reihe Zellen gelegt wird, die ebenfalls in die Propagation integriert werden, für die Kollisionsberechnung jedoch keine Rolle spielen. Diese „Umrandung“ wird auf den Speicher abgebildet, indem eine in jeder Dimension um zwei Elemente vergrößerte Matrix zum Einsatz kommt. Faktisch ergibt sich damit das Zellarray als

```
Cell cell[(dx+2) * (dy+2)];
```

, wobei das eigentliche Strömungsfeld auf den Bereich

```
cell[{1..dx}, {1..dy}]
```

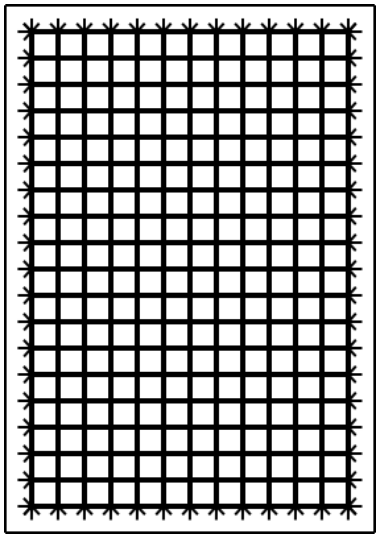

Abbildung 5.4.: Strömungsgebiet mit Propagationspuffern

abgebildet wird. Dadurch erhöht sich zwar der Aufwand für die Adressierung einer Zelle minimal, jedoch kann in der Propagationsphase jede Zelle uniform behandelt werden, ohne berücksichtigen zu müssen, ob es sich hierbei um eine Zelle am Rand des Berechnungsfeldes handelt. Es muss lediglich sicher gestellt sein, dass die nach „innen“ weisenden Randverteilung mit korrekten Daten vorbelegt sind. In der Propagation werden diese Randverteilungen dann in das Feld hineinpropagiert. Analog werden die aus dem Feld weisenden Randverteilungen durch Werte aus dem tatsächlichen Feld gefüllt. In einem zweiten, an die eigentliche Propagation anschließenden Schritt, müssen dann die so gefüllten Puffer entsprechend behandelt werden (im parallelen Fall z.B. durch Weitergabe an den Prozessor, der den benachbarten Zellblock berechnet bzw. beim Einsatz von periodischen Randbedingungen an die im Feld gegenüberliegende Seite).

Gebietszerlegung

Ferner bietet sich eine Aufteilung des gesamten Strömungsgebietes in mehrere unabhängige Teilbereiche an (Abb. 5.5).

Dabei wird jedes der n Teilgebiete als eigenständiges Gebiet berechnet. Der Überlauf an den Rändern wird dann durch den Austausch der entsprechenden Propagationspuffer mit den Nachbargebieten weitergegeben. In einem an die Propagation anschließenden Schritt werden dann die von den umliegenden Teilgebieten empfangenen Daten in das jeweilige Teilgebiet hineinpropagiert. Diese Mehrfachverarbeitung der Randdaten impliziert zwar einen höheren Rechenaufwand, ist jedoch wie Grafik Abb. 5.6 zeigt, durch die bessere Ausnutzung des CPU-Caches deutlich schneller.

5.1.2.2. Optimierung der Vektorstruktur

Wie bereits angesprochen, liegt der größte Nachteil der Vektorstruktur darin, dass Informationen über benachbarte Zellen sich nicht aus dem Datenmodell direkt ergeben, sondern zusätzlich abgespeichert werden müssen. Die oben (siehe 5.1.1.1) angesprochene zusätzliche Matrix in der - ähnlich der reinen

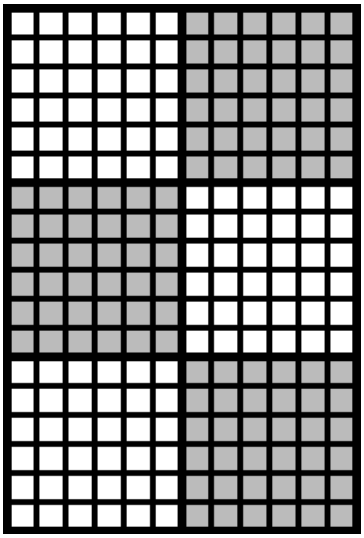

Abbildung 5.5.: Strömungsgebiet bestehend aus unabhängigen Teilgebieten

Matrixlösung - pro Feld ein Zeiger auf das repräsentierte Datenobjekt enthalten ist, ist dabei der Speicherung der Nachbarschaftsinformationen im Datenobjekt selbst vorzuziehen. Zum einen beträgt der zusätzliche Speicherbedarf nur ein Bruchteil des bei der direkten Speicherung benötigten (in 2D: $\frac{1}{4}$, in 3D $\frac{1}{6}$). Zum anderen ist die Zusatzmatrix ohnehin notwendig für eine evtl. Parallelisierung, da hier in jedem Zeitschritt der Überlauf an das benachbarte Teilgebiet übergeben werden muss und somit eine wahlfreie Adressierung der Zellen notwendig ist. Dieser wahlfreie Zugriff lässt sich im Vektormodell aber nur über eine Zusatzmatrix erzielen. Auch wird der erhöhte Aufwand für die indirekte Adressierung durch das bessere Caching kompensiert, da weniger Speicher pro Zelle benötigt wird.

Propagation auf Randzellen

Zusätzliches Optimierungspotential scheint bei der Behandlung der Propagation in den Randknoten vorhanden zu sein. Das Problem ist der Gestalt, dass ein Randknoten gemäß seiner Bezeichnung am Rand eines Hindernisses liegt und damit an mindestens einer Seite keine Fluidzelle als Nachbar hat. In der hier vorgestellten Datenstruktur hat er damit aber überhaupt keinen Nachbar, da innere Hinderniszellen überhaupt nicht erfasst werden. Bei der Propagation bringt das dann den Nachteil mit sich, dass beim Kopieren jeder Verteilung in jeder Zelle geprüft werden muss, ob ein solcher Nachbar überhaupt existiert. Diese häufigen Code-Verzweigungen verhindern einen effizienten Einsatz der CPU-Pipelines und führen somit zu einer starken Geschwindigkeitsreduktion der Berechnung. Lösen lässt sich dieses Problem in Abhängigkeit von der oben angesprochenen Speicherung der Nachbarschaftsinformation:
Speichert man die Zeiger zu den benachbarten Zellen direkt im Datenobjekt ab, so lassen sich diese Zeiger einfach anstatt auf einen definierten Null-Wert auf das Quellobjekt selbst setzten. Die Propagation aus Zelle I erfolgt dann wieder auf dieselbe Zelle I und hat somit keine Wirkung außer der, dass sie keine, die CPU-Pipeline brechende Anweisung enthält.
Speichert man hingegen die Nachbarschaftsinformationen nicht im Datenobjekt selbst sondern in einer Zusatzmatrix ab, so stellt sich dieses Problem als etwas komplizierter dar. Um dennoch auf die zusätzlichen *if*-Abfragen (um das Kopieren auf nicht vorhandene Nachbarzellen zu vermeiden) bei

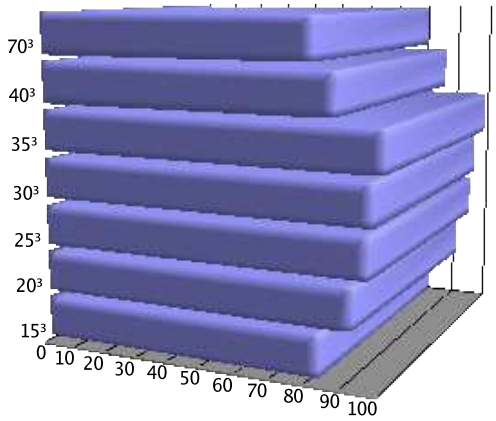

Abbildung 5.6.: Änderung des Durchsatzes über die Teilgebietgröße in %

der Propagation verzichten zu können, muss hier ein zweiter Rand eingeführt werden. D.h. zusätzlich zum eigentlichen Hindernisrand muss eine weitere Zellreihe hin zum Inneren des Hindernisses angelegt werden. Diese zusätzliche Zellreihe dient dann als „Mülleimer“, d.h. auf sie wird nur geschrieben aber nicht von ihr gelesen. Der Vorteil scheint mit der einfacheren Propagation auf der Hand zu liegen, hat sich in der Praxis allerdings nicht bestätigt: Durch den Einsatz eines zweiten Randes bricht die Leistung deutlich ein.

5.1.2.3. Vergleich der Rechenzeit

Wie aus den obigen Ausführungen ersichtlich, haben beide Datenstrukturen ihre Vor- und Nachteile, die sich sowohl im Speicher- als auch Rechenzeitverbrauch zeigen.

Erläuterung zur Abb. 5.7

In Abb. 5.7 wurde mit beiden Datenstrukturen Testläufe auf verschiedenen Strömungsgebieten durchgeführt. Die Gebiete hatten dabei folgende Eigenschaften:

a Das in Abb. 5.22 gezeigte Automobil wobei nur der Rand des Hindernisses tatsächlich aus Hinderniszellen bestand. Das Innere setzte sich aus Fluidzellen zusammen

b Wie *a*, allerdings waren hier auch die inneren Zellen Hinderniszellen

c Komplett hindernisfreies Strömungsfeld der Größe 100x50x40 Zellen

d Komplett hindernisfreies Strömungsfeld der Größe 50x50x50 Zellen

e Komplett hindernisfreies Strömungsfeld der Größe 20x20x20 Zellen

f Zufällig verteilte Hinderniszellen mit der Wahrscheinlichkeit $p_{Hindernis} = \frac{1}{2}$

g Zufällig verteilte Hinderniszellen mit der Wahrscheinlichkeit $p_{Hindernis} = \frac{1}{3}$

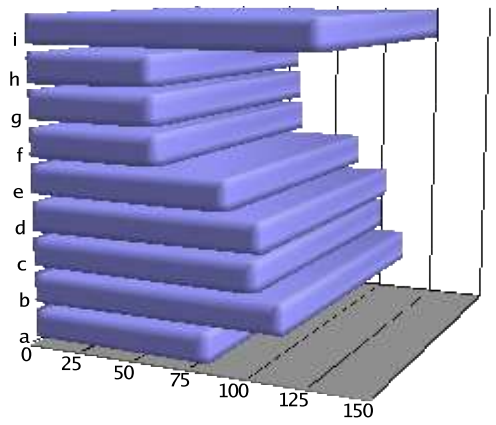

Abbildung 5.7.: Rechenzeitvergleich von Matrix- und Vektorstruktur auf verschiedenen Simulationsfeldern in % (Rechenzeit der Vektorstruktur entspricht 100%)

h Zufällig verteilte Hinderniszellen mit der Wahrscheinlichkeit $p_{Hindernis} = \frac{1}{10}$

i Kugel mit Radius 10 Zellen in einem 50x50x50 Strömungsfeld

Das Ergebnis von Testläufen ist, dass beide Datenstrukturen auf den zu ihnen „passenden“ Strömungsgebieten ein jeweils besseres Laufzeitverhalten aufweisen. Handelt es sich um ein Gebiet mit hohem Hindernisanteil bei gleichzeitig relativ wenigen Randzellen (d.h. große, zusammenhängende Hindernisse mit relativ geringer Oberfläche), so ist die Vektorstruktur klar schneller (b, i), da eben diese Hindernisse nur an ihrem Rand berechnet werden und innere Zellen komplett ignoriert werden können.
Bei Gebieten mit einem hohen Anteil an Hindernisrandzellen hingegen erreicht die Matrixstruktur durch die direkte Adressierbarkeit der Nachbarzellen, vor allem in der Propagationsphase, einen deutlich höheren Durchsatz (a, f, g, h), der den Vorteil des geringeren Rechenaufwandes durch die Selektivität der Vektorstruktur bei weitem kompensiert.
Demnach kann es keine allgemein gültige Aussage geben, welche der beiden Strukturen vorzuziehen ist. Es muss immer das zu berechnende Problem in diese Überlegungen einbezogen werden und anhand der zu erwartenden Speicher- und Rechenkomplexität entschieden werden. Als Faustregel lässt sich jedoch, wie aus den Messungen ersichtlich und oben bereits angedeutet, der Schluss ziehen, dass der zusätzliche Adressierungs- und Verwaltungsaufwand der Vektorstruktur sich nur dann amortisiert und sogar bezahlt macht, wenn das Verhältnis von Hindernis- zu Hindernisrandzellen groß ist.

Weiterführende Informationen zu Datenstrukturen in *Lattice Boltzmann* Strömungssimulationen sowie weitere Implementierungs- und Optimierungshinweise finden sich in [4] bzw. [9].

5.2. Parallelisierung

Will man reale Probleme simulieren, so benötigt man ein hohe zeitliche und räumliche Auflösung. Die Forderung der hohen räumliche Auflösung ist anschaulich, da jedes Hindernis durch Normzellen approximiert werden muss, d.h. eine Kugel z.B. immer eine treppenartige Oberfläche hat. Je höher die Auflösung, desto geringer sind jedoch die verfälschenden Einflüsse dieser Approximation. Zudem ist klar, dass ein System dann besser simuliert werden kann, wenn Mittelungen (wie z.B. bei der Kollisionsberechnung) über relativ kleinere Intervalle erfolgen, da auch hier wieder der Einfluss der Verfälschungen geringer ist. Die zeitliche Auflösung bestimmt sich, wie unten angeführt, direkt aus der räumlichen.

Aus der Konstruktion des *Lattice Boltzmann*-Verfahrens ergeben sich verschiedene Forderungen für die verwendeten Simulationsparameter um die numerische Stabilität zu erhalten und die durch die Konstruktion eingeführten Fehler zu minimieren (siehe auch 3 sowie [9]):

- die Wahl der kinematischen Viskosität $\rho > \frac{1}{1000}$ vermeidet numerische Instabilität und Oszillationen im Druck
- in den Gleichungen 3.5 und 3.6 in Kapitel 3 werden aus Stabilitätsgründen in der Regel $c = \Delta x = \Delta t = 1$ gesetzt, damit ist durch die oben angesprochene räumliche Auflösung ebenso die zeitliche Auflösung festgelegt
- die Geschwindigkeit des Flusses muss verfahrensbedingt klein sein $Ma = \frac{u_{max}}{c_s} < 0.15$. Da $|u_{max}| \ll c_s = \frac{1}{\sqrt{3}}$, ergibt sich $u_{max} \simeq 0.1$.
- Aus obigen drei Einschränkungen und der Definition der Reynoldszahl

  $Re = \frac{(Geschwindigkeit \star CharakteristischeLaenge)}{kinematischeViskositaet}$

  ergibt sich direkt die maximale Element-Reynoldszahl ($Re^\star = Re/L$) zu $Re^\star_{max} \simeq 20$

Da nun die wenigsten praxisrelevanten Probleme eine Simulation bei kleinen Reynoldszahlen erlauben, ist es zwingend, auch Simulationen mit großen Reynoldszahlen möglich zu machen.
Aus den obigen Forderungen von $Re^\star_{max} \simeq 20$ und festgelegter Geschwindigkeit und Viskosität in der Zelle, bleibt als einziger Weg zu hohen Reynoldszahlen die Steigerung der räumlichen Auflösung. Da aber eine Verfeinerung der Auflösung um das n-fache zu einer Erhöhung des Rechenaufwands um das $n^{\#Dimensionen}$ führt, gerät man hier schnell an die Grenzen der Leistungsfähigkeit eines Prozessors. Der naheliegende Weg ist somit die Parallelisierung. Zwar lässt eine Parallelisierung i.d.R. bestenfalls lineare Durchsatzsteigerung erwarten, was gegenüber der kubischen Komplexitätssteigerung im dreidimensionalen Fall nicht gleichwertig ist, dennoch wird man dies meist akzeptieren können und müssen.

5.2.1. Propagationspuffer

Wie bereits mehrfach angedeutet, ist das *Lattice Boltzmann*-Verfahren hervorragend zur Parallelisierung geeignet. Im Wesentlichen kann man das gesamte Strömungsgebiet in n Teilgebiet zerlegen und jedes dieser Teile eigenständig berechnen. Da zur Kollisionsberechnung in jeder Zelle ausschließlich lokale Daten verwendet werden, entfällt hier jeglicher Aufwand für einen Zugriff auf benachbarte

Zellen, d.h. sämtlicher notwendiger Datenaustausch geschieht in der Propagationsphase.
Aufgrund der Tatsache, dass auch in der Propagationsphase in jeder Zelle stets nur Informationen aus den direkt benachbarten Zellen benötigt werden, lässt sich auch hier ein Großteil der Berechnung mit Informationsaustausch innerhalb eines Teilgebietes durchführen. Allein am Rand des Teilgebietes ist ein Abgleich mit den Rändern der benachbarten Teilgebiete notwendig. In Kapitel 5.1.2.1 wurde bereits die Verwendung von Propagationspuffern eingeführt; diese Puffer erweisen sich hier nun wieder als hilfreich, da sie genau die Daten enthalten, die in die benachbarten Teilgebiete propagiert werden müssen (Abb. 5.8).

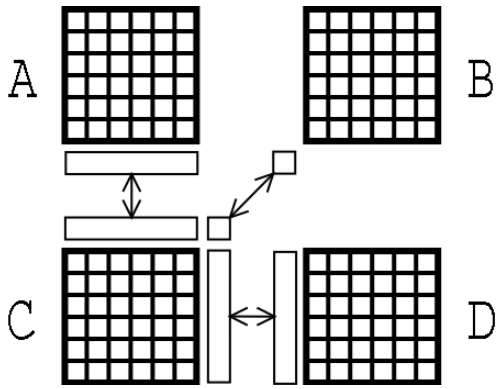

Abbildung 5.8.: Propagation der Puffer über Gebietsgrenzen hinweg

Dazu wird für jedes Nachbargebiet ein kompakter, temporärer Puffer angelegt, in den lückenlos all die Daten hineinkopiert werden, die für diesen Nachbar relevant sind. Das sind all jene Verteilungen, die die Grenzgerade (bzw. im dreidimensionalen die Grenzfläche) schneiden und aus dem Teilgebiet hinausweisen. Diese kompakten Puffer werden dann über ein Nachrichtensystem an die benachbarten Teilgebiete übertragen und dort entsprechend aus den Puffern, in einem der eigentlichen Propagation ähnlichen Schritt, in das Teilgebiet hineinpropagiert.

Besonderes Augenmerk ist dabei auf jene Verteilungen zu legen, die Teilgebiete verbinden, deren Positionen sich auf mindestens zwei Koordinatenachsen (bezogen auf das System der Teilgebiet) unterscheiden, in Abb. 5.8 also die diagonale Propagation zwischen den Teilgebieten B und C.
Um beim Hineinpropagieren jede Zelle gleich behandeln zu können, was Code-Verzweigungen reduziert, werden stets alle in das Feld zeigenden Verteilungen des Puffers propagiert. Im D2Q9-Modell bedeutet dies z.B. für den rechten Puffer des in Abb. 5.8 dargestellten Teilgebietes C, dass die Verteilungen f_2, f_6 und f_8 (also die nach links, links unten und links oben weisenden) kopiert werden. Das Problem wird deutlich, wenn man Quelle und Ziel der Kopieroperationen betrachtet: der besagte Propagationspuffer kann für die oberste Zelle keine Informationen über die Verteilung f_6 (nach links unten weisend) haben, da diese aus einem anderen Teilgebiet kommt (Teilgebiet B statt D). Bei diesem Wert handelt es sich um einen Dummy-Wert, der ausschließlich zur Code-Vereinfachung (s.o.) übermittelt wird. Daher ist sicherzustellen, dass besagte Diagonalverteilungen bzw. deren Propagationspuffer nach der Propagation der Orthogonalverteilungen verarbeitet werden. Kurzzeitig nimmt man damit zwar Inkonsistenzen durch im Allgemeinen zufällige Werte in Kauf, diese werden jedoch beim anschließenden Kopieren der Diagonalverteilungen mit den richtigen Daten überschrieben und haben somit keine Auswirkung.

5.2.2. Kommunikation und Synchronisation

Bisher wurde nur davon gesprochen, dass Daten aus den Propagationspuffern der Teilgebiete in die benachbarten Teilgebiete übermittelt werden müssen. Das „wie“ ist hier allerdings der entscheidende Punkt, da es sich bei der Kommunikation tendenziell um den Flaschenhals der Simulation handelt. Dies ist leicht ersichtlich, aufgrund der Tatsache, dass auch Hochleistungs-Kommunikationsnetzwerke wie Gigabit-Ethernet, Infiniband oder Myrinet nicht die Bandbreite bieten, mit der die CPU an den lokalen Hauptspeicher angebunden ist. Damit sind fast zwangsläufig Idle-Zeiten der CPU zu erwarten, wenn diese auf die langsamere Kommunikationshardware warten muss. Es gilt also ein System zu finden, das die vorhandene Hardware auslasten kann und dabei gleichzeitig einfach und auf die Bedürfnisse der Simulation zugeschnitten ist. Letzter Punkt ist nicht zu vernachlässigen, da ein System mit komplexem Kommunikations- und/oder Synchronisationsprotokoll auch einen entsprechend höheren Overhead erzeugt. Dieser Overhead kann sich bemerkbar machen durch verringerte Nettobandbreite und höhere CPU-Idlezeiten, da evtl. die Nachrichtenfolge implizit mehr Synchronisation beinhaltet, als in diesem Fall nötig wäre (vergleiche hier z.B. die verschiedenen Arten mit MPI eine Nachricht zu versenden). Unter Auslastung der Hardware fällt dabei auch, dass evtl. vorhandene Strukturen wie SharedMemory, das zwischen einzelnen der am Rechenverbund beteiligten Prozessoren vorhanden ist, entsprechend zur Steigerung der Performance genutzt werden kann.

Hier stellte sich die Frage, ob es zweckmäßiger wäre, ein komplett eigenes, auf das Problem zugeschnittenes Kommunikationssystem zu entwickeln, oder aber auf Standardsoftware zurückzugreifen. Beides hätte eine gewisse Berechtigung, da z.B. relative geringe Funktionalität benötigt wird und somit eine Eigenentwicklung - auch aus Overhead-Überlegungen heraus - sicherlich den ein oder anderen Vorteil in puncto Geschwindigkeit hätte bringen könne. Letztlich fiel die Wahl dennoch aus folgenden Gründen auf das Standardpaket MPI[10]

- MPI bietet eine hohe Abstraktion von der zugrunde liegenden Hardwarearchitektur und zusätzlich Unterstützung von einiger Spezialhardware wie z.B. Infiniband; deren Unterstützung hätte sich im Rahmen dieser Arbeit nicht realisieren lassen (siehe auch Anhang D)

- Es zeichnet sich durch seine weite Verbreitung aus und ist für nahezu jede Architektur erhältlich. Das war ein wichtiges Kriterium für die eingangs erwähnte Unabhängigkeit des Codes von Betriebssystem und Hardwarearchitektur

- Zusätzliche Funktionalitäten wie Logging, Synchronisation, Management von Programmstart/-Ende/-Abbruch etc. werden neben der reinen Kommunikations-API angeboten

- Es impliziert einen relativ geringen Kommunikationsoverhead

- Die Programmierschnittstelle ist relativ simpel und leicht nachvollziehbar. Auch werden keine keine zusätzlichen Compiler benötigt wie z.B. bei OpenMP

Aus diesen Gründen und insbesondere aufgrund der Verfügbarkeit für verschiedenen Plattformen (siehe 4.1) fiel die Entscheidung für den Einsatz von MPI und gegen eine Eigenentwicklung wie es z.B. bei der Client-Server-Kommunikation (siehe 5.3) der Fall war.

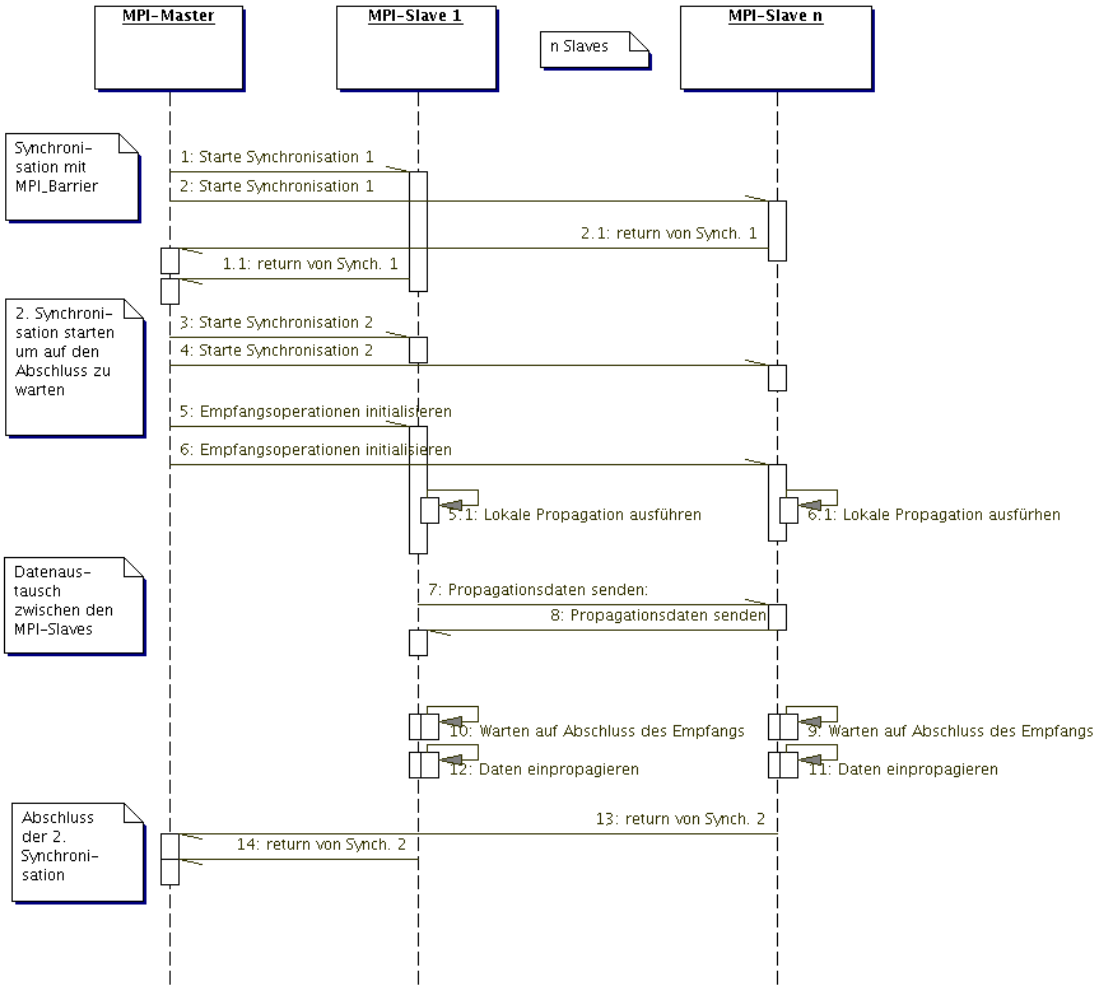

Abbildung 5.9.: Kommunikation während der Berechnung eines Schrittes

5.2.2.1. Einsatz des Kommunikationssystems

Abb. 5.9 erläutert den Ablauf der Berechnung eines Zeitschritts aus der Sicht der Kommunikation.
Am Anfang steht ein Synchronisationsschritt (*MPI_Barrier*) um sicher zu stellen, dass sich alle Teilgebiete im selben Iterationsschritt befinden. Zusätzlich werden hiermit auch definierte Zeitpunkte für ein konsistentes Auslesen des aktuellen Simulationsstatus eingeführt (siehe auch 5.3.3).
Im zweiten Schritt erfolgt das Initialisieren der asynchronen Empfangsoperationen (*MPI_Irecv*). Diese Initialisierung ist prinzipiell nicht notwendig, reduziert jedoch die MPI-interne Pufferung von Nachrichten. Wurde eine Empfangsoperation bereits initialisiert bevor die zugehörige Sendeoperation von einem zweiten Prozess gestartet wurde, so entfällt ein Teil der aufwendigeren MPI-Nachrichtenverwaltung, die ansonsten nötig wäre um die Daten bis zu ihrer Abholung durch eine später gestartete Empfangsoperation vorzuhalten[10].
In einem dritten Schritt, der Redistribution und lokalen Propagation erfolgen keine Nachrichten-

übermittlungen. Dieser Schritt ist ebenso wie die Kollisionsberechnung vollständig lokal und hat als kommunikationsrelevantes Ergebnis die gefüllten Propagationssendepuffer in jedem Teilgebiet (siehe 5.2.1).
Anschließend werden diese Puffer asynchron via *MPI_Isend* an die Prozesse der jeweiligen Nachbargebiete versandt, d.h. die Sendeoperation kehrt sofort zurück und nicht erst nach Empfang der Nachricht durch den Zielprozess. Dadurch können alle Propagationen für alle Teilgebiete eines Prozesses gestartet werden, was zum einen die Bandbreite der Kommunikationseinheit auslastet und zum anderen CPU-Idlezeiten reduziert, da einige hier notwendige Berechnungen durchgeführt werden können, während die CPU auf den Abschluss der Kommunikation wartet.
Der letzte Einsatz von MPI während eines Berechnungsschrittes erfolgt nun dahingehend, dass auf die Beendigung aller asynchronen Sende- und Empfangsoperationen gewartet wird. Dieser Schritt erfolgt im synchronen Aufruf *MPI_Wait* und erzwingt ein Blockieren des Prozesses bis jede der Übertragungen abgeschlossen ist. Dieser Schritt ist notwendig um beim folgenden Hineinkopieren der Empfangspuffer in das jeweilige Teilgebiet stets konsistente Daten als Grundlage zu haben.

Die in Kapitel 5.1.2.1 beschriebene Aufteilung des Strömungsgebietes in mehrere Teilgebiete - im parallelen Fall hier also die nochmalige Aufteilung der Teilgebiete in Untereinheiten - würde bei schemahafter Anwendung des obigen Verfahrens allerdings einen erhöhten Aufwand für MPI-Kommunikation mit sich bringen, da auch innerhalb des Teilgebietes Nachrichten zwischen benachbarten Untereinheiten ausgetauscht würden. Dies lässt sich auf zwei Arten verhindern:

- Die Empfangs- und Sendepuffer der Untereinheiten können so entworfen werden, dass sie jeweils eine Entsprechung in der anderen Menge finden. Damit erreicht man die Möglichkeit, durch einfaches Austauschen der Zeiger auf diese Puffer die Übertragung mittels MPI überflüssig zu machen.

- Ersetzen der tatsächlichen Aufteilung in Untereinheiten, d.h. deren unabhängige Anordnung als einzelne zusammenhängende Speicherbereiche, durch eine logische Einbettung in einen zusammenhängenden Speicherbereich.

Beide Ansätze bieten Vor- und Nachteile. Der Hauptnachteil der ersten Variante ist, dass für jede Untereinheit die Sende- und Empfangspuffer verwaltet werden müssen. D.h. in jedem Schritt werden diese Puffer gefüllt und nach dem Austausch in die Zielgebiete hineinpropagiert. Allerdings ist dieser Aufwand gering gegenüber der Zeiteinsparung durch das verbesserte Cacheverhalten (siehe Abb. 5.6). Im zweiten Fall könnte die Propagation ohne den Einsatz von Sende- und Empfangspuffern an den Rändern der Untereinheiten durchgeführt werden, hier ließen sich die Nachbarzellen berechnen. Da das Konzept der Aufteilung in Untereinheiten aber beibehalten werden soll (siehe 5.1.2.1), wird die Adressierung jeder einzelnen Zelle damit etwas komplexer - es muss nun zusätzlich die Position der entsprechenden Untereinheit mit eingerechnet werden. Zudem muss beim Einsatz von mehreren Threads bei der Propagation (siehe 5.2.4) darauf geachtet werden, dass keine noch zu propagierenden Daten von einem anderen Thread überschrieben werden (ähnlich dem in 5.1.2.1 beschriebenen Problem).

In dieser Arbeit wurde der zuerst genannte Ansatz realisiert, da auch im Hinblick auf eine dynamische Lastverteilung (die hier allerdings nicht implementiert wurde) dieser als der bessere erschien. Bei einer dynamischen Lastverteilung können einzelne Untereinheiten relativ einfach auf andere Prozessoren übertragen werden, wodurch eine aufwendige Neuaufteilung des gesamten Strömungsgebietes

überflüssig wird. Der zweite Ansatz gestaltet sich in der Implementierung deutlich komplexer und erfordert das Beachten einiger Besonderheiten (wie oben bereits angedeutet). Daher lässt sich schwer vorhersagen, wie hoch der Performancegewinn wäre bzw. ob es überhaupt einen gäbe. Da diese Arbeit jedoch als Grundlage für Studienprojekte und weiter Diplomarbeiten gedacht ist, mag hier Spielraum für anschließende Versuche vorhanden sein.

Eine weitere interessante Fragestellung ist, ob sich die Kommunikation nicht beschleunigen ließe, wenn man anstatt mehrerer kleiner Nachrichten weniger, dafür größere Nachrichten überträgt. Dies könnte z.B. erreicht werden, indem man Propagationen zu Teilgebieten, deren Position sich auf mindestens zwei Koordinatenachsen (bezogen auf das System der Teilgebiet) unterscheiden, leicht abändert:
Anstatt hier diagonal zu propagieren, erfolgt die Propagation in zwei aufeinanderfolgenden Schritten über die Seiten. Dadurch würden zwar letztlich mehr Daten übertragen, jedoch hätte man eine Nachricht eingespart (Abb. 5.10). Allerdings führt man damit wiederum zeitliche Abhängigkeiten zwischen den Propagationen der einzelnen Verteilungen ein, was den Aufwand für Kontrolle und evtl. Wartezeiten erhöht. Auch hier sei wieder auf evtl. folgende Arbeiten verwiesen.

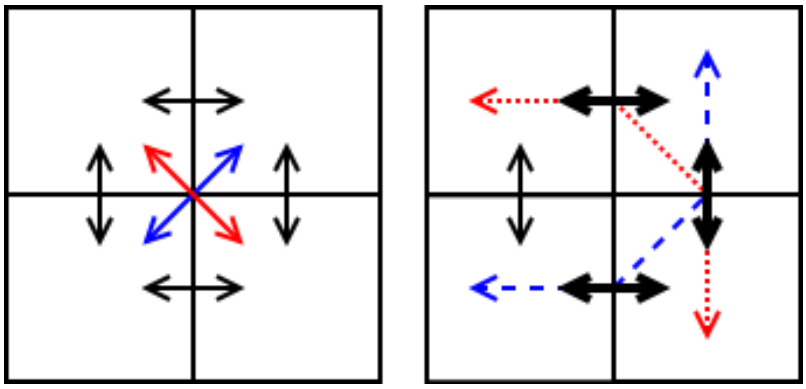

Abbildung 5.10.: Links: Standardpropagation einschl. Diagonalnachricht. Rechts: Reduktion der Nachrichtenanzahl durch Zusammenfassen der Einzelnachrichten

5.2.3. Leistung

Der Parallelisierung liegt die zentrale Forderung nach höherer Rechenleistung zu Grunde. Da jeder zusätzliche Prozessor zusätzliche finanzielle Anforderungen stellt, muss das Ziel lauten, die Parallelisierung so effizient wie möglich zu gestalten. Im Wesentlichen bedeutet das, sofern möglich, nahe an eine linear mit der Anzahl der eingesetzten Prozessoren einhergehende Durchsatzsteigerung zu gelangen.
Anhand der folgenden Konfiguration (im Weiteren mit dem Systemnamen „pclt“ bezeichnet) wurde das Leistungsverhalten des Codes beim parallelen Einsatz auf mehreren CPUs untersucht und soll hier erläutert werden:

Beschreibung der Konfiguration

- 5 Knoten (für das Rechnen wurden nur bis zu 4 verwendet, Knoten 5 diente als Master)

- 2 Intel Xeon 2.4 MHz CPUs (512 Kb Cache) je Knoten
- 1 GB DDR Hauptspeicher je Knoten
- Gigabit Ethernet für Kommunikation zwischen den Knoten
- Betriebssystem RedHat Linux 7.3 mit Kernel 2.4.24 SMP

Gemessen wurde der Durchsatz, den das Gesamtsystem erzielt, d.h. wie viele Normzellen pro Sekunde berechnet werden können. Die Angabe erfolgt dabei relativ zum Durchsatz bei Einsatz einer einzigen CPU.

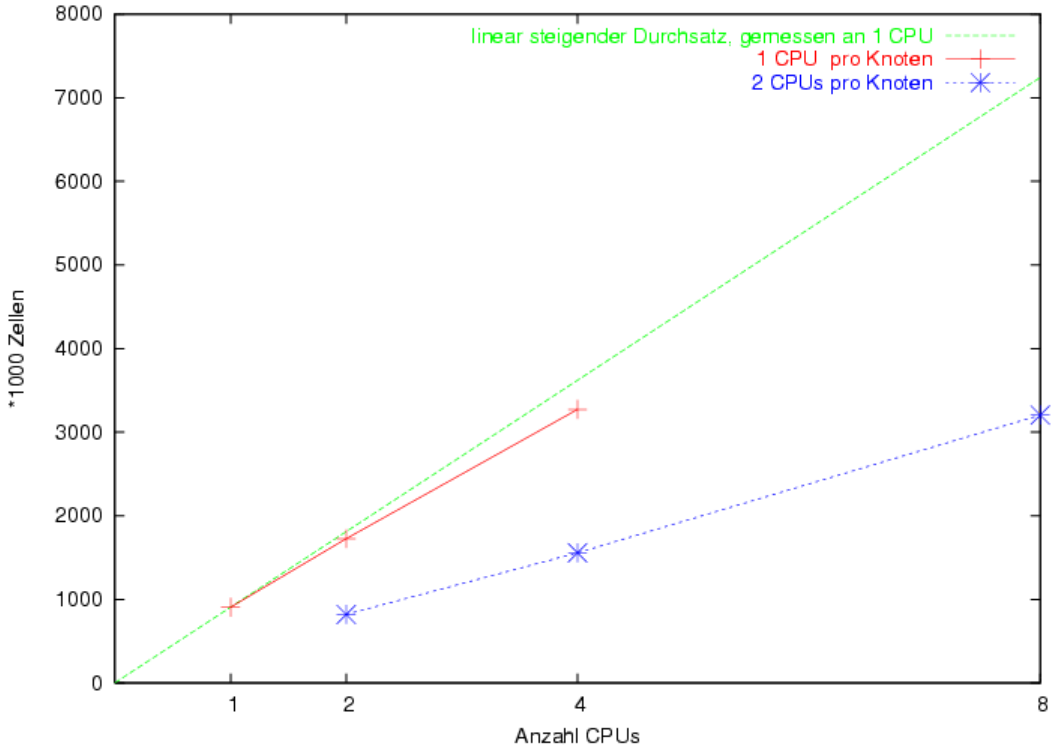

Abbildung 5.11.: Durchsatz des Systems beim Einsatz von 1,2,4,8 CPUs

Abb. 5.11 zeigt bei zwei Prozessoren einem Normwert von ca. 960k Zellen pro Sekunde eine gute Annäherung an die ideale Steigerungsrate. Ein anderes Ergebnis würde überraschen, da hier bei kluger Gebietsaufteilung auch beim Einsatz mehrerer Untereinheiten nur eine Grenzebene existiert, über die hinweg Daten im Rahmen der Propagation zwischen den Prozessen ausgetauscht werden müssen.
Bei vier Prozessoren fällt der Verlust durch Kommunikation und Synchronisation schon stärker ins Gewicht, ist aber mit ca. 10% keineswegs als schlecht zu werten. Allerdings kamen bei diesen beiden Messungen pro Knoten nur eine CPU zum Einsatz, d.h. der Rechner wurde tatsächlich nur zu maximal 50% ausgelastet.

Ein interessantes Phänomen tritt nun genau dann zu Tage, wenn auf einem Knoten beide CPUs zum Einsatz kommen.
Erwartungsgemäß sollte die Durchsatzsteigerung etwas oberhalb der oben beschriebenen Raten liegen, da anstelle der Kommunikation über Netzwerk der Datenaustausch zwischen den MPI-Prozessen

lokal erfolgen kann.
Tatsächlich verhält sich der Code aber konträr zur Annahme und reagiert auf den zweiten Prozessor mit einem deutlichen Leistungseinbruch. Faktisch erreicht ein Prozessor damit nicht einmal mehr die Hälfte des Durchsatzes und somit scheint es günstiger, nur einen Prozessor zu verwenden.

Um den Grund für dieses Verhalten zu ermitteln, wurde der Code einer detaillierten Analyse unterzogen, indem Laufzeitprofile im Ein- und Zweiprozessormodus angefertigt und ausgewertet wurden. Das Ergebnis fasst Tabelle 5.1 zusammen.

	Testlauf 1	Testlauf 2
Propagation 1	26.3s (51.1%)	65.6s (69.3%)
Propagation 2	3.3s (6.4%)	3.6s (3.8%)
Propagation 3	1.8s (3.4%)	3.9s (4.1%)
Synchronisation	2.9s (5.6%)	1.6s (1.7%)
Kollision	16.7s (32.4%)	18.1s (19.1%)
Summe	51.0s (98.9%)	92.8s (98.0%)

Tabelle 5.1.: Benötigte Rechenzeit pro Phase in Sekunden (bzw. % der Gesamtrechenzeit)

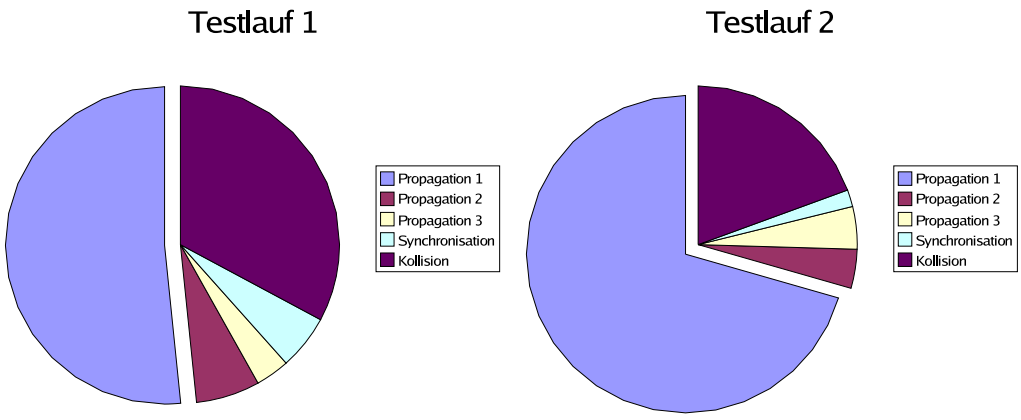

Abbildung 5.12.: Anteilig benötigte Rechenzeit je Phase

Tabelle 5.1 und Abb. 5.12 zeigen die absolut und prozentual pro Berechnungsphase verbrauchte CPU-Zeit. Die einzelnen Phasen enthalten dabei

- **Propagation 1:** Teilgebietlokale Propagation
- **Propagation 2:** Füllen und Versenden der Kommunikationspuffer
- **Propagation 3:** Einpropagieren der Empfangspuffer
- **Synchronisation:** Warten auf die Beendigung der Kommunikationsoperationen und Synchronisation am Ende eines Berechnungsschrittes
- **Kollision:** Lokale Berechnung der Kollision auf jeder Gitterzelle

Berechnet wurde dabei ein Strömungsgebiet (70x35x35 Zellen) mit 2 Teilgebieten a 35x35x35 Zellen. Testlauf 1 wurde auf zwei Knoten mit je einem Prozessor durchgeführt, Testlauf 2 auf einem Knoten

mit beiden Prozessoren.

Auffällig ist, dass nahezu jeder der o.g. Schritte im zweiten Lauf mehr Zeit als im ersten Lauf benötigt, jedoch nirgends ein so deutlicher Unterschied zu Tage tritt wie im Schritt *Propagation 1*. Einzige Ausnahme bildet der Schritt *Synchronisation*, der im Wesentlichen das Warten auf Kommunikation beinhaltet. Hier entspricht es den Erwartungen, dass die Kommunikation über das SharedMemory des Doppelprozessor-Knotens deutlich schneller ist als das im Lauf 1 verwendete Gigabit-Ethernet.
Zwar wird in der Literatur oft darauf verwiesen, dass auf Mehrprozessorsystemen ein besseres Verhalten erreicht wird, wenn ein Prozessor für Betriebssystem und anderer Hintergrunddienste reserviert wird, allerdings sind dabei keine solchen Unterschiede zu erwarten, die die hier beobachteten Differenzen erklären würden
Da wie in 5.1.2.1 beschrieben, der Schritt *Propagation 1* vollständig lokal abläuft und dort tatsächlich nichts anderes passiert, als Speicherwerte von A nach B zu kopieren, bleibt als einzig vernünftige Erklärung die limitierte Speicherbandbreite. Die starke Verdopplung der Rechenzeit in diesem Schritt lässt ebenso vermuten, dass bereits eine Prozessor die gegebene Bandbreite vollständig ausreizt und durch den zweiten Prozessor zusätzlich Kollisionen beim Zugriff auf den Speicher auftreten, was den tatsächlichen Speichertransfer dann erneut bremst.

Bandbreitenberechnung
Bei einem Durchsatz von ca. 960000 Zellen pro Sekunde mit dem D3Q19-Modell und $double$ als Variablentyp ($= 8Byte$) werden damit $960000 * 19 * 8 = 145920000 \simeq 140MB$ an Speicher pro Sekunde eingesetzt. Dieser Speicher muss komplett eingelesen und komplett neu geschrieben werden, da jeder Wert in die entsprechende Nachbarzelle propagiert wird, d.h. in der Summe werden ca. 280 MB pro Sekunde bewegt.

Anhand des folgenden Codes wurde auf den oben beschriebenen Systemen eine einfache Bandbreitenmessung durchgeführt:

Listing 5.3: Bandbreitenmessung

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>

// 20 GB
#define MB      20480
// 1 MB
#define BCOUNT (1*1024*1024)
// number of integers that form 1 MB
#define COUNT  (BCOUNT / sizeof(int))

int main(int argc, char** argv) {
  struct tms start;
  struct tms end;

  int* buffer1 = new int[COUNT];
  int* buffer2 = new int[COUNT];

  // init with random values to prevent compiler optimization
```

```
  for (int i=0; i<COUNT; i++)
    buffer1[i] = (int) (((1.0 * random()) / RAND_MAX) * COUNT);

  times(&start);
  for (int i=0; i<MB; i++) {
    for (int j=0; j<COUNT; j++)
      buffer2[COUNT-1-j] = buffer1[j];
  }
  times(&end);

  printf("used time: %f\n", (end.tms_utime - start.tms_utime) / 100.0);
}
```

Dieser Code wurde wieder auf beiden Konfigurationen ausgeführt und bestätigte die Vermutung, dass der limitierende Faktor hier nur die Speicherbandbreite sein kann:

	Testlauf 1	Testlauf 2
benötigte Zeit	35.8s	77.1s
Durchsatz	$\frac{559MB}{s}$	$\frac{259MB}{s}$

Tabelle 5.2.: Benötigte Rechenzeit für Laden und Speichern von 20 GB durch eine CPU

Nach den Ergebnissen in Tabelle 5.2, entfallen ca. 50% der Gesamtrechenzeit auf den Propagationsschritt, d.h. während der Hälfte der Rechenzeit ist die Speicherbandbreite vollständig ausgeschöpft. Fasst man die obigen Daten und Ergebnisse also zusammen, so stehen einer möglichen Spitzenleistung (gemessen am Code nach Listing *Bandbreitenmessung*, s.o.) von 559 MB * 50% $\simeq$ 280 MB ein tatsächlicher Wert von 140 MB * 2 = 280 MB (*2 wg. Load/Store) entgegen, was belegt, dass hier keine Verbesserung durch effizienteres Implementieren ohne eine konzeptionelle Änderung der Struktur zu erwarten ist.

Möglicherweise ließe sich durch ein anderes Datenmodell die Propagation beschleunigen:
Die hier vorgestellten Modelle (Matrix und Vektor, siehe 5.1.1) sind klar auf die Kollisionsphase hin optimiert. Dort zeichnen sie sich durch hohe Lokalität aller Speicherzugriffe aus, d.h. mit wenigen Speicherzugriffen werden alle relevanten Daten eines Schrittes in den CPU-Cache geladen und dort verarbeitet, das Lesen und Speichern der Daten geschieht also blockweise und nicht für jedes Datum einzeln. Gemäß der Literatur[4] ist dies gemeinhin der bessere, da schnellere Ansatz.
Für Einprozessorsysteme sowie Hochleistungsrechner, die eine deutlich höhere Speicherbandbreite aufweisen, ist dies sicherlich uneingeschränkt richtig. Es wäre jedoch zu prüfen, ob nicht durch ein Speichern der Verteilungen in jeweils einem Vektor in der Propagationsphase deutlich Rechenzeit gespart werden könnte. Dann könnte nämlich die gesamte (lokale) Propagation durch einen (im Test deutlich schnelleren) *memcpy*-Befehl bzw. durch ein einziges Zeigerverschieben pro genanntem Vektor realisiert werden (siehe Abb. 5.13).

Für die Kollisionsphase würde dies einen deutlich höheren Adressierungsaufwand bedeuten, dieser wäre jedoch evtl. gerechtfertigt wenn man die Verteilung der Rechenzeit auf die einzelnen Phasen betrachtet (Abb. 5.12).
Da die Implementierung dieses Datenmodells jedoch die Grenzen dieser Diplomarbeit gesprengt hät-

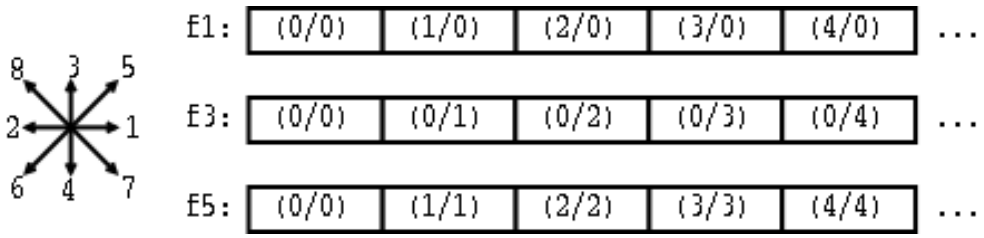

Abbildung 5.13.: Alternatives Modell: Propagationsoptimierte Verteilungsstruktur

te, sei hier auf evtl. folgende Arbeiten verwiesen.

5.2.4. Multithreading, OpenMP

Um dem oben beschriebenen Problem zu begegnen und zumindest zu einem Verhalten zu gelangen, das den Einsatz eines zweiten Prozessors nicht verbieten sondern wenigstens durch eine leichte Performancesteigerung anbieten würde, musste nach anderen Parallelisierungsansätzen gesucht werden. Da offensichtlich die Parallelisierung eines gesamten Iterationsschrittes in Form eines eigenständigen Prozesses hier nicht zum Erfolg führt, bleibt weiter nur die Parallelisierung von Teilen des Schrittes. Dazu bieten sich im Wesentlichen zwei Wege an:

- der Einsatz von OpenMP[12]
- die Aufgliederung der *Teilschritte* auf mehrere gleichzeitig ablaufende Prozesse

OpenMP bietet eine compilergestützte Möglichkeit, einzelne Codeabschnitte auf Basis von Shared-Memory zu parallelisieren. Realisiert wird dies dadurch, dass ein OpenMP-Compiler anhand von besonderen Compiler-Direktiven im Quellcode den zu parallelisierenden Code erkennt, und diesen so aufbereitet, dass er zur Laufzeit in mehreren Threads parallel ausgeführt wird. Für den Programmierer bietet dies den Vorteil, dass er sich nicht um Interna der Thread- und Multiprozess-Technik kümmern muss, sondern diese Aufgabe komplett dem Compiler überlassen kann. Nachteilig daran ist jedoch, dass OpenMP-Programme i.d.R. schwer zu debuggen sind, da der compilergenerierte Code nicht immer den eigenen Vorstellungen entsprechen muss und daher leicht falsche Annahmen über die Arbeitsweise des Codes getroffen werden.

5.2.4.1. Eigenständiger Kommunikationsthread

Diesen Nachteil der fehlenden Transparenz vermeidet man genau dann, wenn man das Multithreading selbst implementiert. Man erreicht dadurch ein 100%ige Kontrolle über die Parallelisierung und kann für Fehlersuche und Laufzeitanalysen an frei definierbaren Punkten ansetzen.
Allerdings gilt es bei einem multithreaded Code darauf zu achten, dass alle eingesetzten Bibliotheken auch threadsicher sind, d.h. durch den Einsatz von mehreren Threads nicht in ihrer Funktion beeinträchtigt werden. MPI (in der hier verwendeten Version 1) z.B. bietet diese Threadsicherheit nicht, weshalb besondere Maßnahmen ergriffen werden müssen, um gleichzeitige MPI-Aufrufe aus unterschiedlichen Threads zu unterbinden.

Ein erster Ansatz für die Parallelisierung einzelner Codeabschnitte war nun, die Kommunikation der Prozesse untereinander in einen separaten Thread auszulagern. Die Idee dahinter beruht darauf, dass auch beim Einsatz asynchroner MPI-Aufrufe die Ausführung des Programms durch das Betriebssystem immer dann durch ein Signal unterbrochen wird, wenn an der Kommunikationsschnittstelle eine Statusänderung eingetreten ist. Dabei wird dann der kommunikationsrelevante Programmcode ausgeführt (in diesem Fall Code der MPI-Bibliothek), was wiederum zur Folge hat, dass

- die CPU-Pipelines leer laufen und neu initialisiert werden müssen
- sämtliche im CPU-Cache befindliche Daten durch die Sende-/Empfangsdaten überschrieben werden

Wird nun die Kommunikation ausgelagert in einen separaten Thread, so kann der Scheduler des Betriebssystems den Kommunikationsthread beim Empfang eines solchen Signals auf dem ungenutzten Prozessor ausführen, sofern nur ein Prozess pro Knoten ausgeführt wird (und damit nur eine CPU zum Einsatz kommt). Das führt dazu, dass der Hauptthread ohne Unterbrechung weiterlaufen kann und keine zusätzlichen Cachefaults produziert werden, da der CPU-Cache jeweils pro CPU vorhanden ist und nicht unter dem gemeinsamen Zugriff beider CPUs steht (gleiches gilt für die Pipelines).

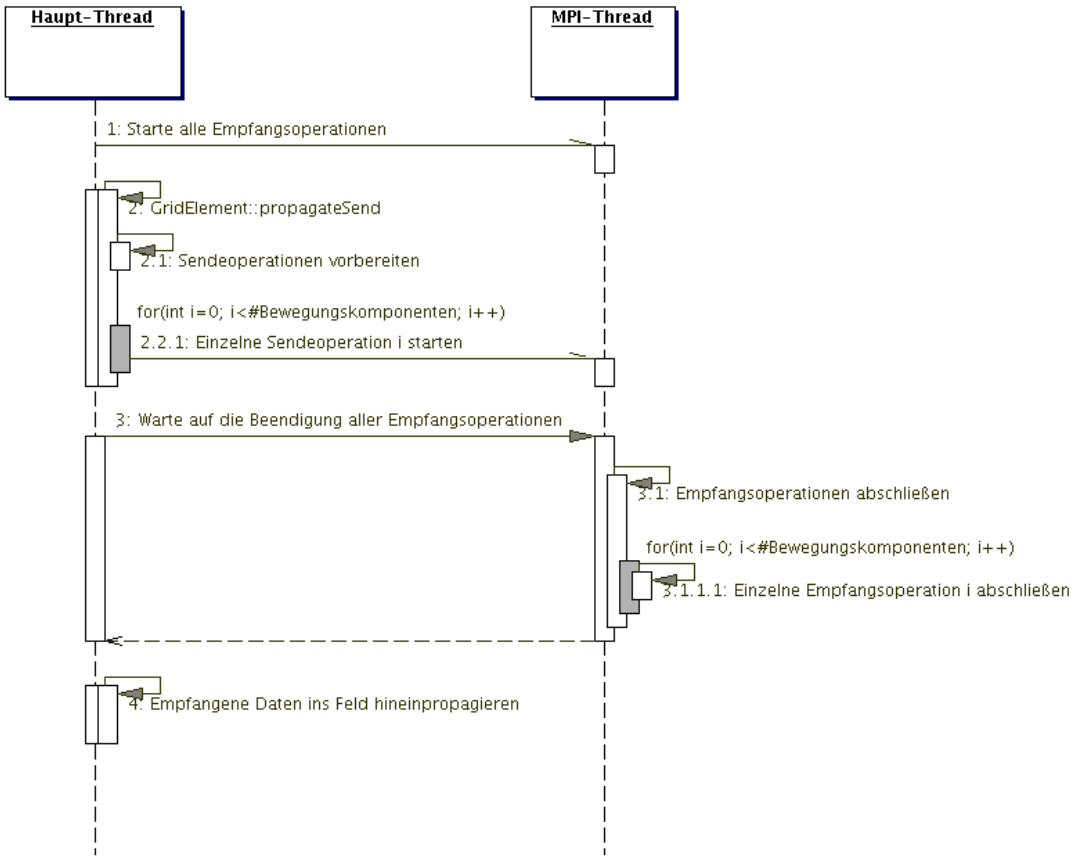

Abbildung 5.14.: Ablauf und Synchronisation von Haupt- und Kommunikationsthread

Abb. 5.14 stellt den prinzipiellen Ablauf der beiden Threads während der Propagationsphase dar. Zu Beginn der Propagationsphase aktiviert der Hauptthread den Kommunikationsthread und veranlasst diesen, alle Empfangsoperationen für dieses Teilgebiet zu starten (siehe 5.2.2.1). Dieser Aufruf erfolgt asynchron, d.h. der Hauptthread wartet nicht bis alle Empfangsoperationen gestartet wurden, sondern geht sofort in die Propagationsschleife über.
Hier werden nun in jedem der n Schleifendurchläufe aus den Propagationspuffern des Teilgebietes die jeweils relevanten Verteilungen in einen zusammenhängenden Sendepuffer kopiert und wiederum in einem asynchronen Aufruf an den Kommunikationsthread übergeben. Dieser startet nun die tatsächliche Sendeoperation (ebenfalls ein asynchrones Senden über *MPI_Isend*) und wartet danach auf die nächste Anweisung der Hauptthreads.
Nachdem in dieser Schleife die Propagation für jede der n Richtungen durchgeführt wurde, blockiert der Hauptthread so lange, bis alle Empfangsoperationen durch den Kommunikationsthread abgeschlossen wurden.

Die Synchronisation der beiden Threads erfolgt dabei über Pipes[18]. Beim Start der Simulation wird der Kommunikationsthread erzeugt und geht sofort in ein blockierendes *read* auf der Pipe über. Der Start der Empfangsoperationen wird durch den Hauptthread eingeleitet, indem dieser eine bestimmte Nachrichten-Id in die Pipe schreibt. Dadurch kehrt der Kommunikationsthread aus seiner Blockierung zurück und kann die geforderten Operationen durchführen. Nach dem gleichen Prinzip erfolgt die Aktivierung für die einzelnen Sendeoperationen und den Abschluss der Empfangsoperationen. Bei letzterer Aktion blockiert nun allerdings der Hauptthread seinerseits durch ein *read* auf der Pipe und wartet bis der Kommunikationsthread nach Beendigung der Empfangsoperationen eine entsprechende Meldung in die Pipe schreibt. Danach blockiert der Kommunikationsthread wieder durch ein *read* bis zur Propagation im nächsten Simulationsschritt.

Ergebnis
Wird auf jedem Knoten nur ein MPI-Prozess eingesetzt, so führt der Einsatz des eigenständigen Kommunikationsthreads zu einer Beschleunigung von ca.

+4%

Interessanterweise ist dies tendenziell auch auf Systemen mit nur einem Prozessor zu beobachten. Zwar fällt hier die Geschwindigkeitszunahme nicht ganz so deutlich aus wie im oben geschilderten Fall, dennoch bewegt sie sich im Bereich von ca. 1% bis 1,5%. Erklären lässt sich dies nur dadurch, dass der Scheduler den Kommunikationsthread wohl nicht immer nach Erhalt des Signals sofort aktiviert sondern oftmals wartet, bis der Hauptthread den nächsten Synchronisationspunkt erreicht. Dadurch können die einzelnen Abschnitte der Propagation (siehe Tabelle 5.1, Propagation 2) abgeschlossen werden bevor MPI-Code zur Ausführung kommt, was eine bessere Cache- und Pipelineausnutzung zur Folge hat.

5.2.4.2. Threads zur Kollisionsberechnung

Da mit der Einführung eines separaten Kommunikationsthreads der Schritt hin zum Multithreadingcode bereits getan war, lag es nahe, weitere Teile des Codes mit Hilfe von Threads zu parallelisieren. Aus Tabelle 5.1 ergeben sich die Schritte, deren Parallelisierung besonders erfolgversprechend ist:

- Propagation 2 (Füllen und Versenden der Kommunikationspuffer)
- Kollision

Die Implementierung lehnt sich an das oben vorgestellte Prinzip an und setzt pro vorhandener CPU einen Thread ein.

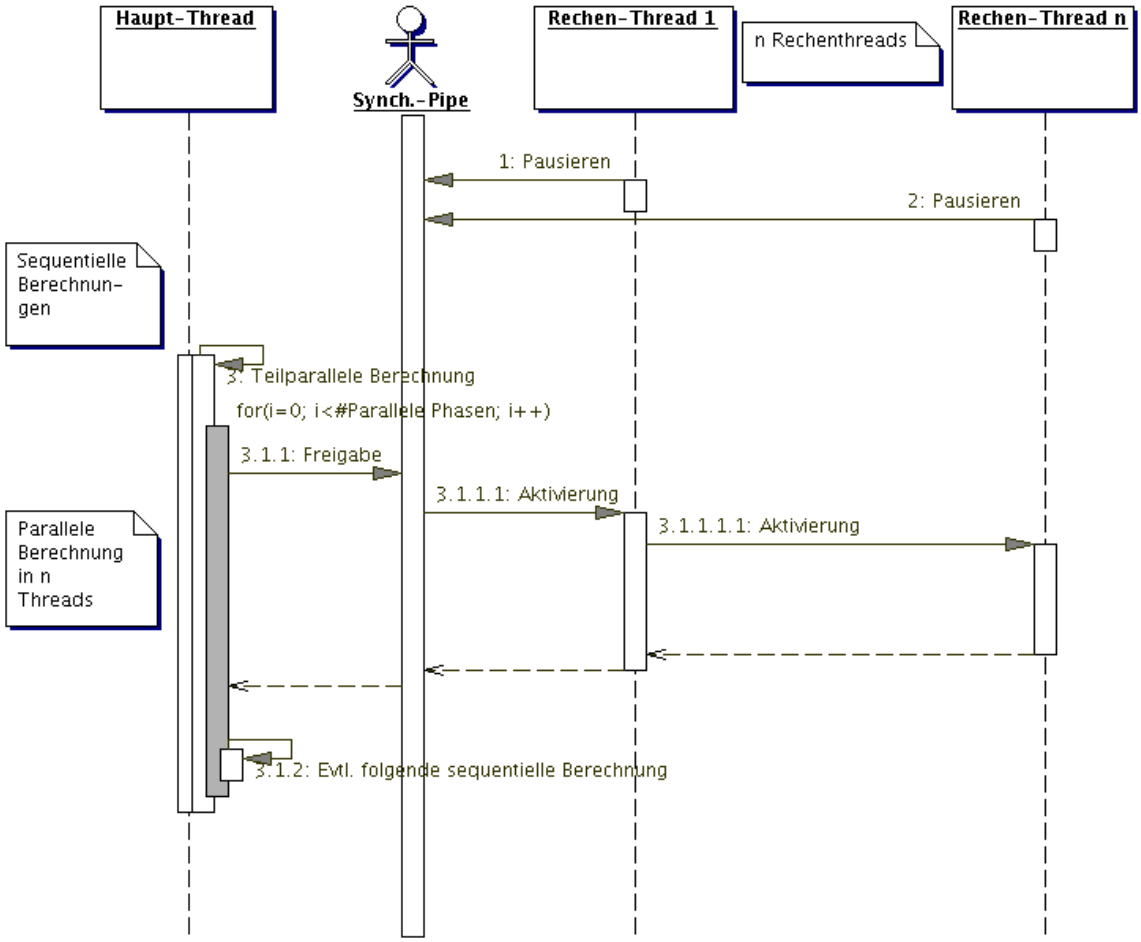

Abbildung 5.15.: Ablauf der Berechnung beim Einsatz mehrerer Rechenthreads

Ergebnis
Beim Einsatz zweier Workerthreads auf zwei CPUs lässt sich ein Geschwindigkeitszuwachs von ca.

+18%

beobachten. Wird zusätzlich der oben vorgestellte Kommunikationsthread verwendet, so steigt die Geschwindigkeit insgesamt um ca.

+23%

gegenüber der 1-Thread-Realisierung.

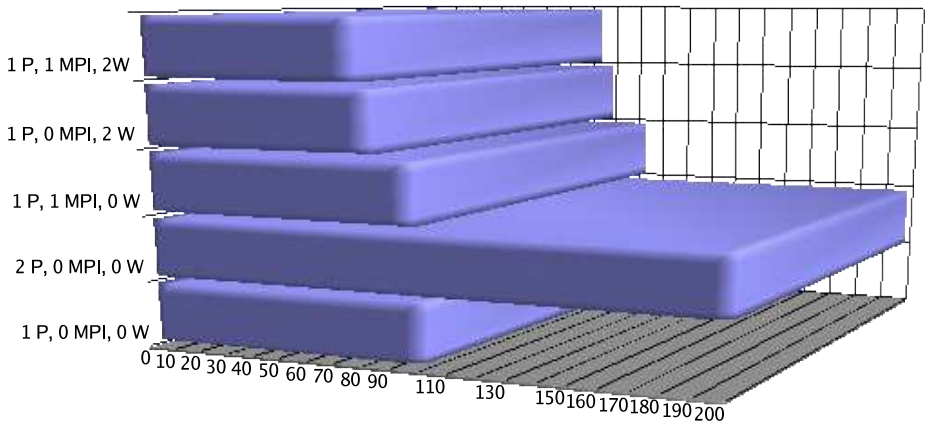

Abbildung 5.16.: Benötigte Rechenzeit in %

Abb. 5.16 zeigt das Rechenzeitverhältnis der verschiedenen oben vorgestellten Implementierungen.

n MPI $\triangleq$ n eingesetzten Kommunikationsthreads
n W $\triangleq$ n eingesetzten Workerthreads
n P $\triangleq$ n eingesetzten CPUs pro Knoten

5.3. Client/Server Architektur

Grundlage für die *reaktiven* Komponenten des Systems bildet die Client/Server Architektur.
Während der Rechenkern in aller Regel auf Hochleistungsrechnern läuft, kann vom Benutzer kaum erwartet werden, die Ergebnisse in gekühlten Maschinenräumen direkt zu betrachten. Auch eine sich an die Simulation anschließende Analyse der Ergebnisse kann nicht allein zufrieden stellend sein, da sie wiederum die reaktive Komponente vermissen lassen würde.

Es bietet sich also an, den Rechenkern um die Fähigkeiten zu erweitern, mit einem oder mehreren Clients kommunizieren zu können. Die drei wichtigsten Erweiterungen sind daher

- Funktionalität zur Rückmeldung von Berechnungsergebnissen und Systemzuständen
- Entgegennahme von Parameteränderungen wie Modifikation von Hindernissen oder Fließeigenschaften des Fluids
- Wahrung der Konsistenz sowohl beim Rückmelden von (Zwischen-) Ergebnissen, als auch konsistentes Einbringen von Änderungen in die laufende Simulation

5.3.1. Mögliche Realisierungen

Es gibt nun verschiedene Realisierungsansätze, die sich mehr oder weniger gut für die oben geschilderten Aufgaben eignen. Sie alle haben gemeinsam, dass über die zentrale Funktionalität - die Bereitstellung einer Netzwerkverbindung - eine mehr oder weniger starke Abstraktionsebene gelegt wird. Auf einige dieser Ansätze wird im Folgenden eingegangen und dargelegt, warum oder warum sie schlussendlich nicht bzw. nicht in ihrer ursprünglichen Form in das System übernommen wurden.

5.3.1.1. Verwendung von MPI

Auf den ersten Blick erscheint der Gedanke interessant, MPI nicht nur zur Kommunikation der Recheneinheiten untereinander, sondern auch zur Kommunikation mit einem Client zu verwenden. Auf diese Art und Weise erhielte man ein System mit einheitlicher Kommunikation, einheitlicher Synchronisation und hätte nicht zuletzt auch eine äußerst ressourcenschonende Möglichkeit gefunden, Daten zwischen dem Benutzer und der Berechnung auszutauschen.
Schon bei näherer Betrachtung allerdings treten diese theoretischen Vorteile in den Hintergrund, da eine Einbindung des Clients in die MPI-Architektur nur unter größeren Schwierigkeiten (wenn überhaupt) möglich ist:

- Der Client müsste in den logischen Rechnerverbund aufgenommen werden
- Damit darf die Aufteilung der Rechenaufgaben auf die einzelnen Prozessoren aber nicht mehr alleine dem MPI-System überlassen werden, da dies ansonsten den Client als Recheneinheit erkennen könnte. Dies wiederum würde aber zwangsläufig zu einer Erhöhung der Komplexität führen. Auch könnten verschiedene MPI-interne Optimierungen (wie z.B. die Erhöhung der Lokalität auf SMP- oder SHM-Maschinen durch günstige Verteilung der Recheneinheiten) unter Umständen nicht mehr so gut greifen.
- Die Kommunikation würde auf einem zwar schnellen, aber doch relativ primitiven Level ablaufen. Der Fokus von MPI liegt sicherlich auf dem Austausch zeitkritischer Nachrichten bei definierter Behandlung von Gleichzeitigkeit und der zeitlichen Folge von Ereignissen. Da es sich beim Client um eine Komponente handelt, die nicht im selben Maß zeitkritisch ist wie der Rechenkern, liegt es hier nahe, zugunsten einer höheren Abstraktion und damit einfacherem Interface den Performance-Aspekt etwas zurückzustellen.

Zusammenfassend bleibt, dass MPI zwar für die Kommunikation der Recheneinheiten untereinander hervorragend geeignet ist, die Kommunikation über den Rechnerverbund hinweg jedoch besser mit Protokollen getätigt werden sollte, die eine höhere Abstraktion erlauben und flexibler sind in den Kommunikationspartnerschaften die sie eingehen können.

5.3.1.2. HTTP

Das HTTP-Protokoll bietet letztlich nur einen Vorteil gegenüber einer Eigenentwicklung:
Es gibt fertige Bibliotheken für quasi jedes Betriebssystem- und Hardwarearchitektur.

Zwar ist das HTTP durch seine Verwendung im WWW weit verbreitet und auch leicht zu implementieren, dennoch hat es zwei gravierende Nachteile, die gegen den Einsatz dieses Protokolls sprechen:

- HTTP ist (zumindest in seiner ursprünglichen Form) verbindungslos, d.h. für jede Anfrage muss eine Verbindung zum Server hergestellt werden. Zudem ist das Halten eines globalen Zustandes seitens des Servers nur mit zusätzlichem Aufwand zu erreichen.
- Die Verarbeitung von Binärdaten erfordert eine Transformation auf ein Sieben-Bit-Encoding
- Serverinitiierte Nachrichten stellen auf Grund der Verbindungslosigkeit beim Einsatz über Firewallgrenzen hinweg unter Umständen ein Problem dar.

5.3.1.3. CORBA oder andere RPC-Systeme

Kennzeichen von RPC-Systemen wie CORBA, COM, RMI o.ä. ist, dass sie es einem Entwickler von Clientanwendungen erlauben, Dienste eines Servers zu nutzen, ohne die hierfür notwendige Kommunikation explizit zu berücksichtigen. Entsprechend hervorzuheben ist hierbei, dass der Entwickler auf entfernte (auf dem RPC-Server ausgeführte) Objekte und Methoden zugreift, als wären sie lokal verfügbar - das RPC-System kümmert sich um die Befehls- und Rückgabedatenübermittlung und garantiert (je nach Implementierung) eine geordnete Abfolge.
Erreicht wird dies durch die automatische Generierung von sog. Stubs und Skeletons, also Methoden, die im Client- resp. Servercode anstelle der eigentlichen Methoden aufgerufen werden und den Aufruf über das Netzwerk transportieren.

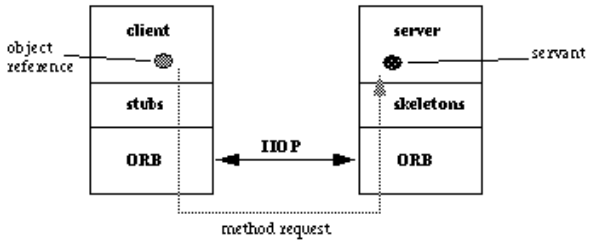

Abbildung 5.17.: Methodenaufruf in RPC-Systemen (Quelle: Object Management Group)

Prinzipiell sind solche Systeme hervorragend für den vorliegenden Anwendungsfall geeignet:

- Hohe Abstraktion durch Verwendung objektorientierter Konzepte
- Kapselung sämtlicher Netzwerkkommunikation
- Ihre Verwendung impliziert eine einfache Programmierschnittstelle an der Grenze zum Server

Der gravierende Nachteil solcher Systeme ist jedoch, dass die Bereitstellung des Frameworks durch die Systemadministration teilweise einen nicht zu unterschätzenden Aufwand erfordert bzw.für manche Architekturen schlicht unmöglich ist.

Aus diesem Grund fiel die Entscheidung für das im Rahmen dieser Diplomarbeit verwendete Kommunikationsprotokoll auf eine Eigenentwicklung, die im Folgenden skizziert werden soll.

5.3.2. Alternatives Kommunikationsprotokoll

Das hier verwendete Kommunikationsprotokoll zeichnet sich durch seine Einfachheit und Transparenz aus.
Endpunkte der Kommunikation sind ein Server sowie ein Client, die über einfache TCP/IP-Sockets miteinander verbunden sind. Die Verbindung besteht dabei (im Gegensatz zu Protokollen wie HTTP, siehe 5.3.1.2)) über den gesamten Sitzungszeitraum hinweg, was unnötiges Ab- und Wiederaufbauen des Verbindungskanals verhindert und zusätzlich serverinitiierte Absetzen von Nachrichten beim Betrieb über eine Firewall nicht behindert.

Die tatsächlich übertragenen Daten sind nun serialisierte Kommando-Objekte. Dies ist von der Konzeption ähnlich den RPC-Systemen wie CORBA/COM und wurde übernommen weil es zusätzlich zu seiner Einfachheit auch eine hohe Abstraktion auf Seiten des Programms erlaubt:

Durch Implementieren des folgenden Interfaces kann jede Klasse erweitert werden hin zu einer gültigen Kommandoklasse. Die einzig notwendige zusätzliche Funktionalität ist das Lesen/Schreiben von bzw. in einen Stream.

```
class Command {
public:
  virtual bool readFromSocket(int sock, char** data,
                              int* length) = 0;
  virtual bool writeToSocket (int sock, const char* data,
                              int length) = 0;
};
```

Dies legt nahe, sowohl im Server- als auch im Clientcode dieselbe Implementierung der Klasse zu verwenden, da somit bei einer Formatänderungen der kommunizierten Daten beide Seiten, die Schreibende und die Lesende, in einem Schritt angepasst werden können.

Auf Clientseite erfolgt nun für jedes abzusetzende Kommando nach der Parametrierung eben jene Serialisierungsphase, in der zusätzlich zu einer Nachrichten-Id und verschieden weiteren Zusatzparametern, die für eine serverseitige Rekonstruktion des Kommandos notwendigen Daten in den Kommunikationsstream geschrieben werden (*writeToSocket()*). Analog dazu wird serverseitig nach der Signalisierung eines Nachrichteneingangs über *readFromSocket()* der Inhalt des Kommandos gelesen und dieses rekonstruiert.
Die eigentliche Ausführung des Kommandos erfolgt dann durch den Server über einen *execute()*-Aufruf, wobei allerdings wie im nächsten Abschnitt beschrieben, noch eine Synchronisationsebene dazwischengeschaltet ist, um Probleme durch Inkonsistenzen zu vermeiden.

```
class Command {
public:
  virtual bool execute() = 0;
};
```

Nach Abarbeitung des Kommandos erfolgt die Rückübertragung der Ergebnisdaten. Dazu wird entsprechend der Hinübertragung *writeToSocket()* auf dem Kommando-Objekt aufgerufen woraufhin die Daten an den Client geschickt werden und dieser nach einem *readFromSocket()* die Ergebnisse rekonstruieren kann. Anzumerken ist dabei, dass die Kommunikation komplett asynchron betrieben

wird, d.h. der Client kann beliebig viele unterschiedliche Kommandos absetzen bevor er die Antwort auf das Erste zurückerhält. Die Zuordnung zum Client-Kommando erfolgt dabei über eine eindeutige Kommandokennung, die der Client zusammen mit dem Kommando erzeugt.

Diese Art der asynchronen Kommunikation bietet sich vor allem für serverinitiierte Datenübertragungen an, wie sie im Auto-Repeat-Modus (siehe 5.3.4) verwendet werden.

5.3.3. Wahrung der Konsistenz

Im vorigen Abschnitt wurde bereits angesprochen, dass ein besonderes Augenmerk auf die Wahrung der Konsistenz beim Ausführen von Kommandos gelegt werden muss, d.h. dass das Ausführen eines Kommandos das System nicht oder nur zeitlich begrenzt und kontrolliert in einen Zustand bringen darf, der undefinierte Ergebnisse produzieren könnte.
Im Folgenden wird der Begriff der *Konsistenz* für diesen Zusammenhang definiert und aufgezeigt, in welchen Fällen diese gefährdet ist. Im Anschluss daran wird dargelegt, wie die Konsistenzwahrung realisiert wurde.

Konsistenz der Simulation bedeutet in diesem Zusammenhang, dass für jede Gitterzelle C des Simulationsgebietes gilt:

$\forall C : LetzterZeitschritt_C = t = NaechsterZeitschritt_C - 1$

D.h. ein konsistenter Zustand ist dann erreicht, wenn für alle Gitterzellen der letzte abgeschlossene Zeitschritt der Schritt t und der nächste Zeitschritt $t + 1$ ist. Keine Gitterzelle befindet sich also zu diesem Zeitpunkt in einem der Berechnungsschritte Kollision/Propagation (siehe 3.4).

Während für Zustandsmeldungen wie

- Melden des Fortschritts
- Melden von Dimensionsdaten des Simulationsgebietes
- Melden des Zelltyps (Fluid-/Hinderniszelle)

Zustände, die im obigen Sinne inkonsistent sind, zugelassen werden können, so ist die Forderung der Konsistenz doch grundlegend für Zustandsmeldungen und Interaktionen wie

- Melden von berechneten Größen wie Druck oder Geschwindigkeit für einen beliebigen Ausschnitt
- Setzen oder Löschen von Hindernis-Zellen
- Erzeugen eines kompletten Speicherabbildes

Diesen Anforderungen genügt der Code wie folgt (Abb. 5.18):

Jedes Kommando ist genau einer der Dringlichkeitsklassen *Immediately, AfterCalculationStep, AfterDump, AfterFinish* zugeordnet, wobei es für jede dieser Klassen ein eigenes Modul gibt, das sich um die Ausführung der entsprechenden Kommandos kümmert.
Empfängt der Server nun ein solches Kommando, so sucht er das zugehörige Ausführungsmodul und übergibt dorthin das Kommando - dieser Schritt erfolgt ohne zeitliche Verzögerung (abgesehen von

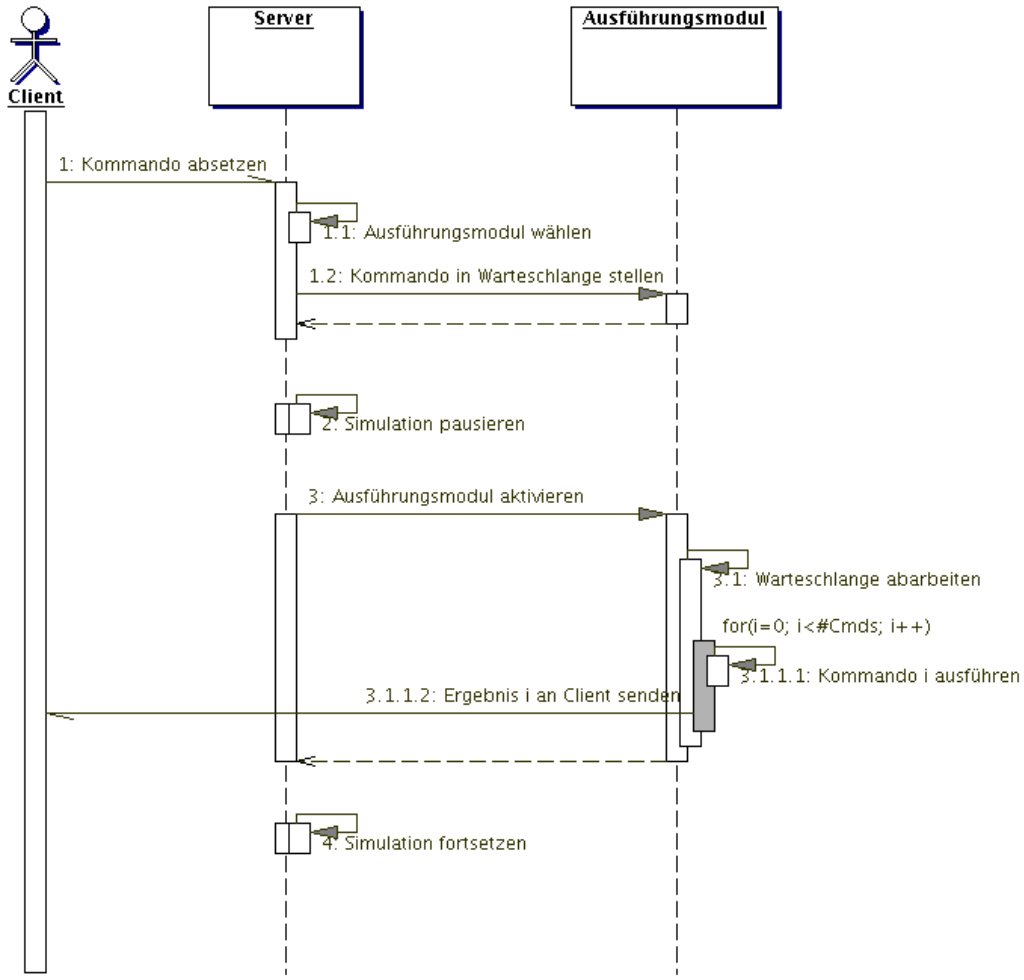

Abbildung 5.18.: Ablauf der Bearbeitung eines Kommandos

der Sperrung eines kritischen Bereichs beim Modifizieren des modulinternen Zustands).
Die Konsistenz der zurückgemeldeten Daten wird nun dadurch sicher gestellt, dass die einzelnen Ausführungsmodule erst dann zur Abarbeitung der wartenden Kommandos aktiviert werden, wenn der Zustand der Berechnung dies erlaubt. Es wird also z.B. das *AfterCalculationStep*-Modul genau dann aktiviert, wenn der aktuelle Zeitschritt für alle Gitterzellen berechnet wurde und bevor die Simulation in den nächsten Zeitschritt eintritt. Letztere wird bis zur vollständigen Abarbeitung aller wartenden *AfterCalculationStep*-Kommandos angehalten.

Server-Kommandos

Unter 5.3.2 wurde dargelegt, dass es sinnvoll sein kann, server- und clientseitig dieselbe Implementierung für die Kommando-Klasse zu verwenden. Um jedoch nicht unnötige Code-Abhängigkeiten zu implizieren, erfolgt seitens des Servers eine zusätzliche Kapselung der Kommandoobjekte, wobei diese Wrapper-Klassen dann die tatsächliche Kommando-Implementierung beinhalten.

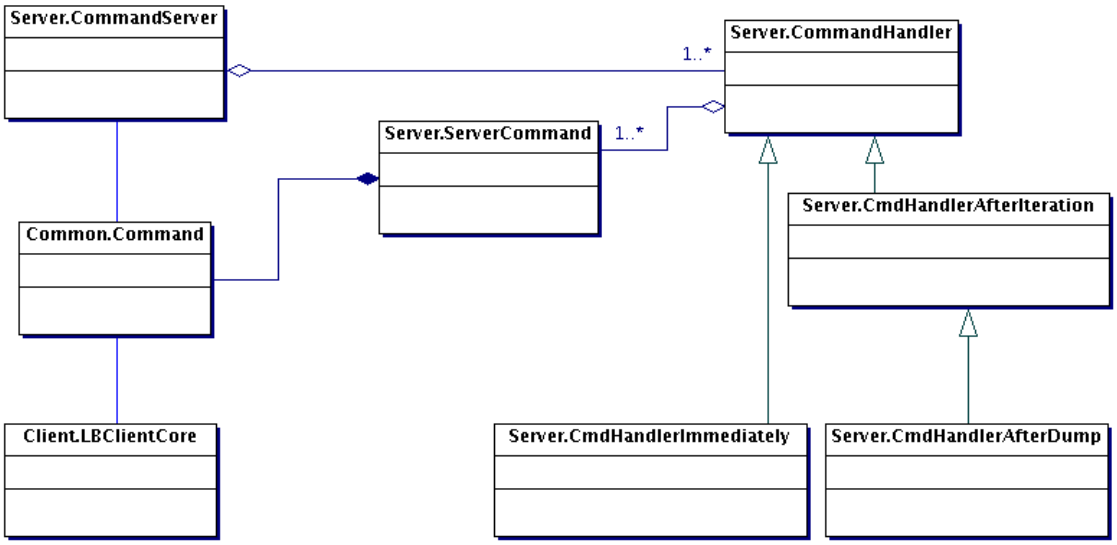

Abbildung 5.19.: Klassendiagramm der Kommando-Ebene

Abb. 5.19 verdeutlicht diesen Zusammenhang: *Common.Command* wird von beiden Teilen, dem Server sowie dem Client eingesetzt und dient der Übermittlung von Kommando- und Rückgabedaten.
Auf Seite des Servers werden *Common.Commands* in einem *Server.ServerCommand* gekapselt und gemäß der oben beschriebenen Klassifizierung an den jeweiligen *Server.CommandHandler* gebunden. Der *Server.CommandServer* aktiviert die *Server.CommandHandler* genau dann, wenn eine Ausführung der ihnen zugeordneten *Server.ServerCommands* (bzw. *Common.Commands*) als nicht konsistenzgefährdend gilt.

5.3.4. Automatische Wiederholung von Kommandos

Für Funktionen wie die Übermittlung eines Druck- oder Geschwindigkeitsabbildes ist es wünschenswert, eine intervallgesteuerte Übertragung zur Verfügung zu haben, die in definierbaren Zeitabständen eine Aktualisierung der zuletzt übertragenen Daten durchführt. Könnte dies theoretisch auch durch den Client in einer Art „Polling“ durchgeführt werden, so ist es in jedem Fall eleganter, diese ressourcenschonend auf dem Server zu realisieren.
Die Implementierung gestaltet sich derart, dass jedes Kommando-Objekt zusätzlich ein *Time-To-Live*-Feld sowie eine eventuelle Intervalllängenspezifikation (gemessen in Zeitschritten der Simulation) besitzt. Wurde nun ein *Server.ServerCommand* ausgeführt, so wird sein TTL-Feld um den jeweiligen Wert dekrementiert und auf das Erreichen der 0-Grenze erfolgt die Löschung des Kommandos aus dem Handler. Andernfalls verbleibt es dort und der Handler prüft bei seiner nächsten Aktivierung, ob der aktuelle Zeitschritt t mit dem nächsten Ausführungszeitschritt t_{cmd} zusammenfällt, also $t \ mod \ Intervalllaenge_{cmd} = 0$ ist.
Um ein sich automatisch wiederholendes Kommando abzubrechen, kann einfach das gleiche Kommando (d.h. ein Kommando der selben Klasse) mit einer TTL von 1 und einer beliebigen Intervalllänge > 0 abgesetzt werden. Da sich Kommandos der selben Klasse gegenseitig überschreiben, kann so

ein falsch abgesetztes Kommando nicht das gesamte System durch unnötigen Ressourcenverbrauch blockieren, da es durch das lediglich einmal auszuführende Abschluss- bzw. Löschkommando ersetzt wird.

5.3.5. Client-Gerüst

Auf Clientseite wird die Anbindung an diese Struktur durch die in Abb. 5.19 gezeigte Klasse *Client.LBClientCore* realisiert, die sich zur Kommunikation mit dem Server des beidseitig verfügbaren *Common.Commands* bedient. Diese Klasse ist als Grundlage für beliebige Clients konzipiert und bietet eine zusätzliche Abstraktionsebene für Clientanwendungen, in der lediglich Schnittstellen zum Aufruf der gewünschten Operationen angeboten werden, wobei sämtliche Interna wie Netzwerkkommunikation und Synchronisation vor dem Programmierer verborgen werden.
Die Klasse ist konzipiert als ein *QObject* der QT-Bibliothek[17] und fügt sich daher in deren Signal-Slot-Konzept zur asynchronen Nachrichtenübermittlung ein (siehe auch 5.4.2). Das bedeutet, dass Aufrufe an *LBClientCore* nach dem Absetzen des *Commands* an die aufrufende Instanz zurückkehren und diese beim Eintreffen der Antwort(en) über ein sog. „Signal“ darüber informiert werden und entsprechend reagieren können. Signal ist hier nicht im Sinne des ANSI C-Signalkonzepts zu verstehen, es handelt sich dabei lediglich um einen etwas vereinfachten Callback-Mechanismus, näheres ist der QT-Beschreibung zu entnehmen.
Auf diese Art und Weise lassen sich beliebige Clientanwendungen entwickeln, die ihren Fokus auf die Darstellung der Ergebnisse richten können, ohne sich mit Fragen der Kommunikation auseinandersetzen zu müssen. In Anhang C findet sich eine Liste der bisher implementierten Funktionen, sowie eine Beschreibung des Nachrichtenformats zwischen Client und Server.

5.4. Datenexport und Visualisierung

Zwar ist die Visualisierung der Simulationsergebnisse nicht explizit in den Anforderungen an diese Diplomarbeit genannt, dennoch erfordert jeder ernst gemeinte Versuch einer Implementierung des *Lattice Boltzmann*-Verfahrens, dass die resultierenden Daten wenigstens mit einfachen Mitteln visualisiert werden können. Anderenfalls würden sich Implementierung und Fehlersuche als deutlich komplexer herausstellen als sie durch die hier durchgeführte Parallelisierung ohnehin schon sind.

Bei entsprechender Hardwareausstattung bietet sich auch für komplexere Simulationsprobleme eine direkte Visualisierung der Zwischenergebnisse während des Simulationslaufes an. Durch die in Kapitel 5.3 beschriebene Aufteilung in einen Rechenkern sowie den separaten Visualisierungsclient kann dies auch dann erreicht werden, wenn der Simulationsrechner räumlich von der Visualisierungskomponente getrennt ist und lediglich eine Netzwerkverbindung zwischen beiden besteht. Die im Folgenden vorgestellte Visualisierungskomponente ist als prototypisch anzusehen und soll lediglich die Möglichkeiten demonstrieren, die mit dieser Architektur gegeben sind. Im Rahmen eines auf dieser Arbeit aufbauenden Softwarepraktikums wird ein leistungsfähigeres Tool entstehen. Denkbar wäre allerdings auch eine Anbindung an komplexere Präsentationshardware wie z.B. Powerwalls o.ä.

5.4.1. Datenexport

Der einfachste Weg um berechnete Ergebnisse anzuzeigen, führt über Standard-Visualisierungstools wie z.B. OpenDX[11]. Dazu werden nach jedem n-ten Iterationsschritt (siehe auch 5.3.3) die gewünschten Daten für das gesamte Strömungsgebiet ausgelesen und in einem einfachen Binärformat auf Festplatte gespeichert. Die vom System unterstützten Informationstypen sind dabei

- Fluidgeschwindigkeit in jeder Zelle, aufgeschlüsselt nach X, Y und Z (skalar)
- Absolutgeschwindigkeit des Fluids in jeder Zelle (vektoriell)
- Druck in jeder Zelle

Die Anzeige dieser Daten kann über beliebige Visualisierungswerkzeuge erfolgen, für die dann jeweils ein Konverter/Importer entwickelt werden muss (siehe 5.6.2).
Um die zeitliche Verzögerung beim Auslesen und Schreiben der Daten zu minimieren, wird für jedes der in Kapitel 5.1.2.1 beschriebenen i Teilgebiete bzw. Untereinheiten eine eigene Datei
$Iterationsschritt_{globalX,globalY,globalZ}$
angelegt. Das erfordert zwar einen höheren Aufwand in der Nachbearbeitung, verringert jedoch erheblich den Kommunikations-, Speicher- und Rechenzeitaufwand während der Simulation. Zwar bietet der MPI 2 Standard Unterstützung für verteiltes Schreiben in Dateien, wie jedoch in Kapitel 4.1 bereits angeführt, war eine der Rahmenbedingungen, unter denen diese Diplomarbeit durchgeführt wurde, die Verwendung von MPI 1, in welchem diese Funktionalität noch nicht zur Verfügung stand. Aber auch mit MPI 2 hätte sich die Frage gestellt, ob nicht mehrere kleine Gebietsausschnitte einfacher zu handhaben sind als eine Datendatei mit den Informationen des gesamten Strömungsgebietes, da man hier schnell in Bereiche von mehreren Dutzend bis mehreren Hundert MB gelangt.

5.4.2. Eigenständige Visualisierung

Herkömmliche Visualisierungswerzeuge (wie sie auch in Kapitel 5.6.2 angesprochen werden) sind hauptsächlich auf die Darstellung von Daten ausgelegt und optimiert. Ein wesentliches Thema dieser Arbeit war allerdings die Reaktivität bzw. Interaktivität des Systems. D.h. der Benutzer sollte während des Simulationslaufs die Möglichkeit bekommen, Hindernisdaten und weitere Simulationsparameter zu ändern. Wenn dies auch ohne eine dazwischengeschaltete Visualisierung möglich ist, so wäre es dennoch unpraktisch. Aus diesem Grund wurde eine eigens auf dieses System zugeschnittene Visualisierung implementiert, die zusätzlich zur reinen Darstellung der Simulationsdaten auch die Forderung nach Interaktivität erfüllt.

Die zentrale Eigenschaft der Visualisierungskomponente ist, dass sie sich zur Laufzeit in Form eines Clients mit dem als Server fungierenden Rechenkern verbindet, um mit diesem direkt zu kommunizieren (siehe auch 5.3). Somit hat der Benutzer die Möglichkeit, jederzeit den aktuellen Berechnungsstand eines beliebigen Ausschnittes des Strömungsgebietes zu betrachten. Das es sich bei diesem Client im Gegensatz zum Rechenkern nur um eine prototypische Implementierung handelt, ist die Betrachtung eingeschränkt auf die Ebenendicke von genau einer Normzelle, d.h. dreidimensionale Visualisierung, wie sie verschiedene ausgereifte Tools bieten, ist hier nicht möglich.
Die Oberfläche des Clients wurde unter Zuhilfenahme der freien GUI-Bibliothek QT[17] entwickelt.

Die Wahl fiel auf QT, da diese Bibliothek für mehrere Plattformen und Betriebssysteme verfügbar ist und zusätzlich für nichtkommerzielle Zwecke frei und kostenlos genutzt werden kann. Zudem besitzt sie eine leicht erlernbare und gut strukturierte objektorientierte Programmierschnittstelle.
Um einen bestimmten Ausschnitt des Strömungsgebietes auszuwählen, hat der Benutzer die Möglichkeit, anhand von Schiebereglern Position und Ausmaße des Ausschnittes festzulegen. Nach Betätigung des „GetData“-Buttons werden diese Daten an den Server übertragen woraufhin jener die angeforderten Daten zurückliefert. Der „GetData“-Button wird bewusst gefordert, da andernfalls durch das Nacheinanderausführen der Einstellungen (X, Y, Z, DX, DY, DZ) mehrere unnötige Datentransfers initiiert würden, was zum einen den Server bremst und zum anderen auch im Client Verzögerungen hervorruft.

Abb. 5.20 zeigt die Fluidgeschwindigkeit in X-Richtung für einen Ausschnitt des Strömungsfeldes um ein Auto. Um die Farbwerte zu berechnen werden, zuerst Minimum und Maximum der vom Server zurückgelieferten Geschwindigkeitsdaten berechnet, um so die Spanne der zu visualisierenden Werte zu erhalten. Alternativ könnte diese Spanne auch vom Benutzer eingegeben werden, was letztlich eine detaillierte Auflösung für relevante Wertebereiche erlauben würde. Um die Oberfläche des Clients jedoch einfach und übersichtlich zu halten, wurde hier eine automatische Berechnung favorisiert.
Diese Wertespanne dient nun dazu, jedes Datum auf einen Wert des Farbgradienten von blau über grün und gelb nach rot abzubilden. Blau bedeutet dabei „geringster Wert“ und rot entsprechend „höchster Wert“.
Die so berechneten Farbwerte werden in ein Bitmap der Größe $dx_{vis} * dy_{vis} * dz_{vis}$ eingetragen und für die Darstellung am Bildschirm auf Zielgröße skaliert. Die Zwischenstufe über das Bitmap hat den Vorteil, dass die Farbberechnung nur einmal pro Ausschnitt durchgeführt werden muss und nicht erneut bei jeder GUI-Aktualisierung. Zudem können für das Skalieren auf Darstellungsgröße effiziente Algorithmen der zugrunde liegenden Grafik-/GUI-Bibliotheken (hier XLib/QT) benutzt werden, was auch Operatoren wie Inter- und Extrapolation einschließt und somit das optische Erscheinungsbild der Visualisierung verbessert.

Auf diese Weise können Druck bzw. die Geschwindigkeiten in Richtung jeder der drei Dimensionsachsen dargestellt werden.

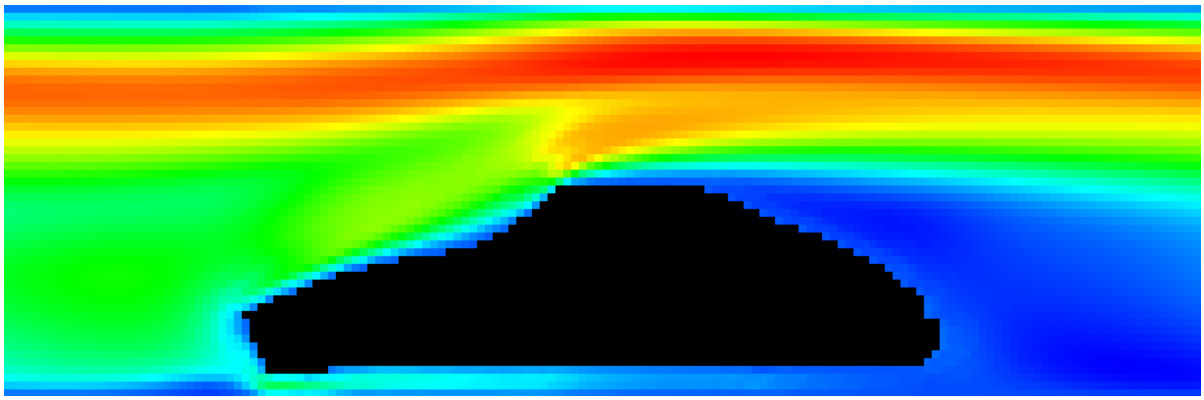

Abbildung 5.20.: Fluidgeschwindigkeit in X-Richtung, Seitenansicht

Um eine kontinuierlich fortschreitende Visualisierung zu erhalten, kann zusätzlich eine automatische Aktualisierung aktiviert werden. Dabei wird die letzte Datenanfrage in frei definierbaren Intervallen

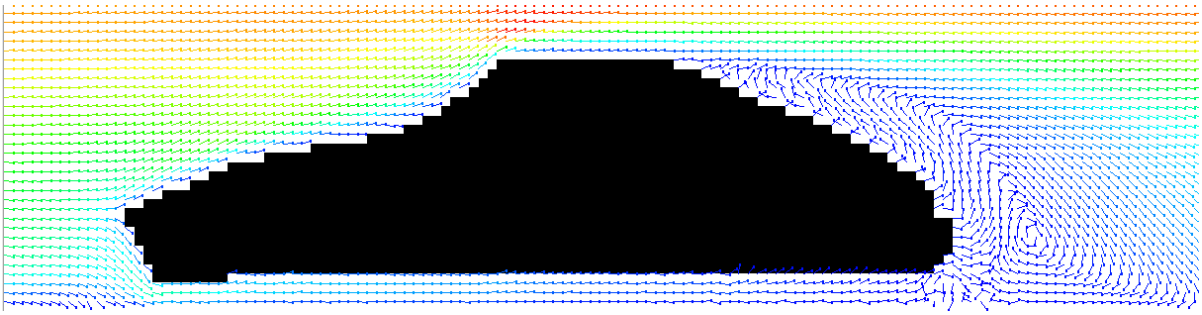

Abbildung 5.21.: Geschwindigkeitsfeld mit Anzeige der Flussrichtung, Seitenansicht

(bezogen auf Anzahl der Rechenschritte) wiederholt.
Ein detaillierte Beschreibung der Steuerung des Clients findet sich in Anhang B.

5.5. Interaktion

Zusätzlich zur oben vorgestellten reinen Visualisierung der Simulationsdaten hat der Client verschiedene Zusatzfunktionen, die eine Interaktion des Benutzers mit dem Rechenkern erlauben. Dazu gehören neben einfacher Ablaufsteuerung wie Pausieren und Fortsetzen der Simulation im Wesentlichen die Modifikation von Hindernisdaten.

Für das Hinzufügen bzw. Entfernen von Hinderniszellen stehen zwei Funktionen zur Verfügung: das Flipping einzelner Zellen direkt in der in Abb. 5.20 dargestellten Anzeige sowie das indirekte Einfügen/Löschen einfacher Geometrieelemente über einen separaten Dialog. Bei Ersterem findet der Benutzer dahingehend Unterstützung, dass sich optional die Zellgrenzen als Gitter über die Anzeige legen lassen und die logische Position der Maus in $(X, Y, Z)_{absolut}$ angegeben wird.
Beim Arbeiten mit Geometrieelementen sind derzeit die primitiven Körper Kugel und Quader unterstützt. Dabei kann der Benutzer die entsprechenden Daten (X, Y, Z, DX, DY, DZ für einen Quader bzw. Mittelpunkt und Radius für eine Kugel) eingeben und zusätzlich auswählen, welche Änderungsoperation auf den so spezifizierten Zellen durchgeführt werden soll. Zur Auswahl stehen

- Einfügen als Hinderniszellen
- Löschen aller in diesem Bereich liegenden Hinderniszellen
- Flipping des Zelltyps aller in diesem Bereich liegenden Hinderniszellen

Das Verhalten des Fluids auf diese Änderung ist allerdings nicht dem physikalischen Verhalten nachempfunden, da dies den Rahmen dieser Arbeit gesprengt hätte.
Beim Ändern einer Zelle von Fluid- auf Hinderniszelle wird lediglich deren Typ geändert, eine der Realität nachempfundene Massenverdrängung auf die umliegenden Zellen findet nicht statt.
Analog dazu wird bei der Änderung von Hindernis- auf Fluidzelle diese Zelle lediglich mit der initialen Dichte sowie der Geschwindigkeit $\vec{0}$ initialisiert.

Dies hat zur Folge, dass keine echten Bewegungen von Körpern simuliert werden können, da vor allem beim Löschen von Hinderniszellen starke Fehler eingeführt werden, die erst im Laufe des Si-

mulationsfortschritts ihre Relevanz verlieren.
Hier bietet sich also noch relativ viel Raum für Weiterentwicklungen.

5.6. Zusätzliche Komponenten

Zusätzlich zu den bisher beschriebenen Systemkomponenten sind im Rahmen dieser Arbeit noch verschiedene Zusatzprogramme entstanden, die das Arbeiten mit dem Simulationscode vereinfachen. Diese Komponenten erheben nicht den Anspruch, besonders elegant und effizient implementiert zu sein, sondern stellen vielmehr eine adhoc-Ansatz dar und sollen darum hier nur kurz vorgestellt werden.

5.6.1. GTS-Gebietsmodellierung

Mit der GTS-Gebietsmodellierung ist ein einfaches Werkzeug entstanden mit dem Ziel, die Möglichkeit zu erhalten, ein Strömungsgebiet aus primitiven Körpern wie Quadern und Kugeln aufzubauen. Dabei sollten diese Körper durch boolsche Operationen verknüpft werden können, um so auch komplexere Geometrien erzeugen zu können. Für die Beschreibung eines solchen Gebietes bot sich eine hierarchische Beschreibungssprache wie XML [20] an, da jedes Geometrieelement sowie jeder Operator leicht auf einen XML-Baum abgebildet werden können und die Gesamthierarchie somit eine vollständige Beschreibung des zu modellierenden Gebietes enthält.

```
<?xml version="1.0"?>
<!DOCTYPE scenario SYSTEM "gts.dtd">
<scenario dx="50" dy="50" dz="50">
  <collection>
    <operator name="minus">
      <!-- Umschliessender Quader -->
      <quboid>
        <x>0</x>
        <y>0</y>
        <z>0</z>
        <dx>50</dx>
        <dy>50</dy>
        <dz>50</dz>
      </quboid>
      <!-- Abzueglich eines inneren Quaders:
           In X/Y auf beiden Seiten jew. 5 Zellen Rand,
           In Z einen Durchstoss -->
      <quboid>
        <x>5</x>
        <y>5</y>
        <z>-1</z>
        <dx>40</dx>
        <dy>40</dy>
```

```
        <dz>52</dz>
      </quboid>
    </operator>
    <!-- In der Mitte des Gebietes eine Kugel mit Radius 10 Zellen -->
    <sphere>
      <x>25</x>
      <y>25</y>
      <z>25</z>
      <r>10</r>
    </sphere>
  </collection>
</scenario>
```

Dieses Beispiel erzeugt ein 50x50x50 Zellen großes Strömungsfeld, das einen 40 Zellen breiten und hohen Durchfluss besitzt und in der Mitte eine Kugel mit 20 Zellen Durchmesser enthält.

Unterstützte Körper und Operationen sind:

- Quader
- Kugel
- Zylinder
- Vereinigung
- Schnitt
- Differenz
- Affine Transformationen wie Streckung, Rotation und Translation

Die tatsächliche Berechnung der Operationen erfolgt mit Hilfe der freien Bibliothek GTS (The GNU Triangulated Surface Library [7]). Dabei handelt es sich zwar um eine Bibliothek die nicht auf die Berechnung von Volumen sondern auf die Oberflächenberechnung spezialisiert ist, für die hier gedachten Aufgaben ist sie dennoch ausreichend. Zudem können auf diese Art und Weise viele der frei verfügbaren GTS-Modelle als zusätzlicher Teil der o.g. XML-Hierarchie mit eingelesen werden und somit als Ausgangspunkt für eigene Simulationsfelder oder zu Testzwecken benutzt werden.

Bei der Transformation der XML-Daten in ein diskretes Strömungsgebiet wird nun ausgehend von den Blättern des XML-Baums jeder Knoten in ein GTS-Objekt überführt. Operator-Knoten führen dabei die entsprechenden Verknüpfungen aus und letztlich entsteht am Wurzelknoten das GTS-Objekt, welches das beschriebene Strömungsgebiet repräsentiert.

Dieses GTS-Objekt wird nun gemäß dem eingestellten Raster zweiwertig zerlegt - in Hindernis- und Fluidzellen und kann somit als Eingabe für die Simulation in eine Hindernisdatei geschrieben werden.

5.6.2. Schnittstelle zu Visualisierungswerkzeugen

Wie in Kapitel 5.4.1 beschrieben, erfolgt die Ausgabe der Berechnungsdaten in einem einfachen Binärformat, das im Wesentlichen ein 1:1 Abbild des Datenspeichers der berechneten Teilgebiete darstellt. Da für jedes Teilgebiet eine eigene Datendatei angelegt wird, war für das Betrachten der Daten ein Nachbearbeitungswerkzeug zu entwickeln, um zum einen die auf die einzelnen Dateien verteilten Daten zusammen zu führen und diese des weiteren für das Format des Visualisierungsprogramms aufzubereiten.
Dieses Procedere hat den Vorteil, dass der eigentliche Simulationscode nicht geändert werden muss wenn ein Visualisierungsprogramm gewechselt wird, sondern lediglich der Filter für das Ausgabeformat angepasst bzw. hinzugefügt werden muss.
Um die Simulationsergebnisse in das jeweilige Format zu überführen, wird dem Konverter nun lediglich das Ausgabeformat, die Nummer des Iterationsschrittes sowie der zu extrahierende Ausschnitt als Kommandozeilenparameter übergeben:

```
./transform --format=opendx --iteration=1000 --clip=0,0,0,200,50,50
```

Exemplarisch wurde dieser Filter für das freie System *OpenDX*[11] entwickelt. Da es sich bei *OpenDX* um ein flexibles und offenes Werkzeug handelt, mit dem sich schnell und ohne großen Aufwand verschiedenste Daten visualisieren lassen, fiel die Wahl auf dieses Programm. Die folgenden Bilder geben einen Eindruck

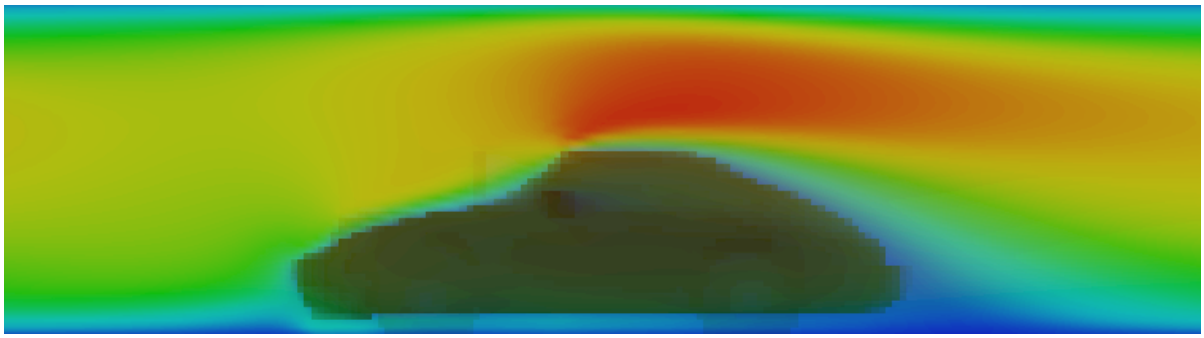

Abbildung 5.22.: Geschwindigkeit in X-Richtung, Seitenansicht

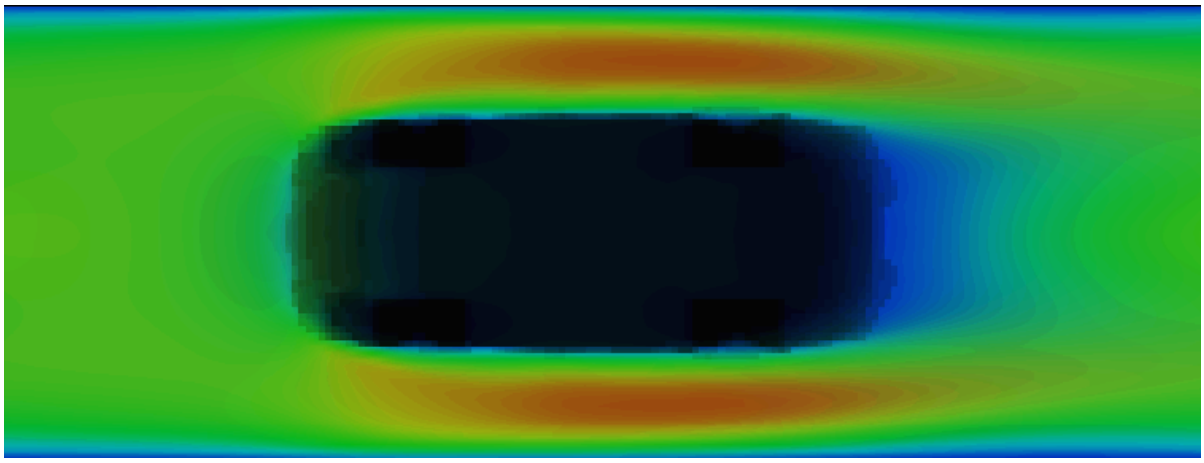

Abbildung 5.23.: Geschwindigkeit in X-Richtung, Unteransicht

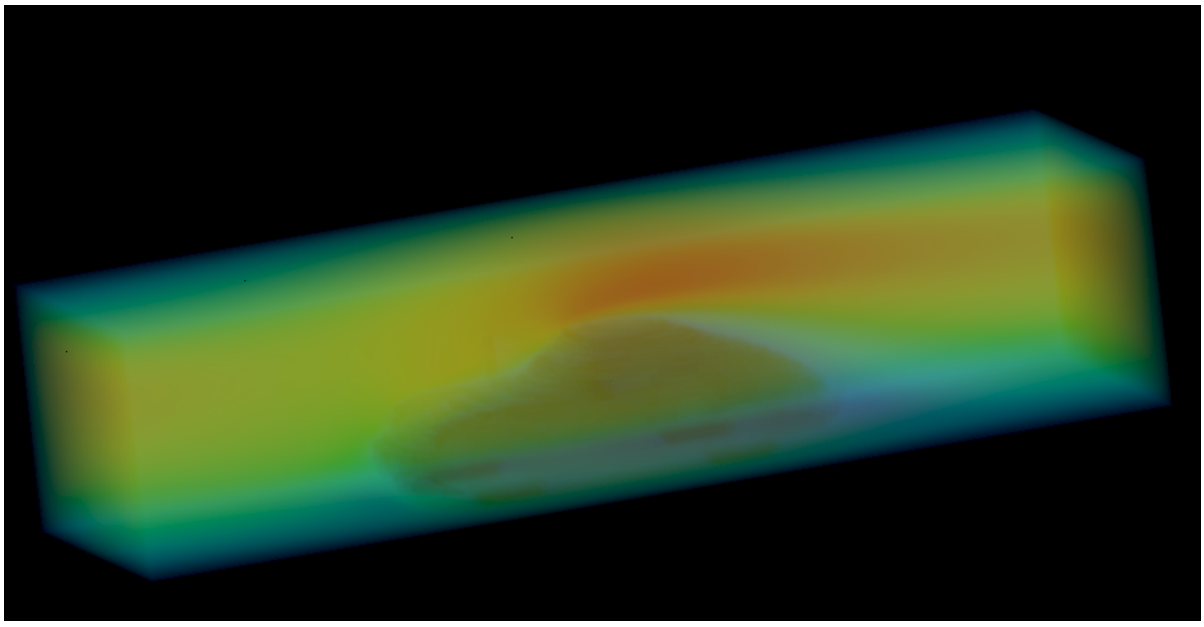

Abbildung 5.24.: Geschwindigkeit in X-Richtung, 3D Ansicht

KAPITEL 6

Zusammenfassung

Ergebnis dieser Arbeit sind zwei eng miteinander gekoppelte Systeme: ein Simulationskern zur Berechnung von Strömungsproblemen mittels des *Lattice Boltzmann* -Verfahrens, sowie ein an diesen Kern angegliedertes Kommunikations- und Interaktionssystem, welches eine Online-Steuerung der Simulation ermöglicht.
Für den Simulationskern wurden verschiedene Wege untersucht, diesen möglichst leistungsfähig zu gestalten, d.h. eine hohe Rechengeschwindigkeit zu erreichen. Ausgehend von möglichen Datenstrukturen zur Haltung der Simulationsdaten, über deren Optimierung unter hardwaretechnischen Gesichtspunkten bis hin zur Organisation der Parallelität wurden dafür Lösungsvorschläge gemacht und ihre Realisierbarkeit und Leistungsfähigkeit unter Beweis gestellt bzw. analysiert.
Ähnliches gilt für die Clientanwendung. Hier wurde eine Plattform entwickelt, die von beliebigen Clients eingesetzt werden kann und auf einem hohen Abstraktionslevel Zugriff auf die Funktionalität des Simulationsservers gewährt. Ihre Mächtigkeit und Einfachheit werden dabei in einem einfachen Visualisierungs- und Interaktionstool demonstriert.
Während auf Seiten der Client- und Kommunikationsfunkionalität die zentralen Fragen und Realisierungsansätze durchaus als geklärt und erfüllt angesehen werden können, so ist doch erwartungsgemäß die ein oder andere Frage - vor allem im Hinblick auf die Parallelität der Berechnung - nicht oder nicht abschließend geklärt. Da aber die Arbeit ohnehin nicht als eigenständigs und in sich abgeschlossenes Projekt geplant war, wurde wo nötig auf die jeweilige Problematik hingewiesen, um in evtl. nachfolgenden Arbeiten zu einer Lösung zu gelangen.
Aus dem Gesichtspunkt der Performance lässt sich zusammenfassend sagen, dass im Großen und Ganzen ein leistungsfähiges, gut parallelisierendes System entstanden ist, das auch für anspruchsvollere Clientanwendungen als die hier implementierte, eine gute Grundlage bietet.

Da der Quellcode, wie eingangs angeführt, unter der GPL veröffentlicht werden soll, findet sich dieser, zusammen mit einer ausführlichen API-Dokumentation sowie Beispielmaterial unter

`http://www.derzach.de/da/` bzw. kann via Mail an `da@derzach.de` angefordert werden.

ANHANG A

Dateiformate

Um die Simulationssoftware einsetzen zu können, ist die Kenntnis der Struktur von Ein- und Ausgabedateien unerlässlich. Im Folgenden soll daher beschrieben werden, wie die Syntax der Eingabedateien definiert ist und gemäß welcher Struktur Ausgabedateien erzeugt werden.

A.1. Eingabedateien

Für die Dateneingabe sind zwei Dateien relevant:

- **lb.obs:** Die Beschreibung des Strömungsgebietes sowie der Hindernisse
- **lb.par:** Rahmenparameter für die Berechnung

Syntax und Namensgebung sind dabei angelehnt an das *Lattice Boltzmann*-Tutorial „anb“[1]

Hindernisdatei „lb.obs“

Die Hindernisdatei setzt sich aus zwei Abschnitten zusammen, dem Kopf und dem eigentlichen Datenblock. Der Kopf besteht aus einer Zeile die für jede der drei Raumdimensionen einen Eintrag für die Anzahl der Normzellen der jeweiligen Dimension enthält. Er beschreibt damit die Ausmaße des Strömungsgebietes. Die Reihenfolge der Dimensionseinträge ist dabei **X, Y, Z** wobei **X+** die Richtung des Flusses angibt.
Der Datenteil besteht aus n Zeilen mit je einer 3D-Koordinate. Jede der n Einträge beschreibt dabei eine Hinderniszelle. D.h. prinzipiell wird jede Zelle zuerst als „Fluidzelle“ angesehen und erst durch deren Angabe in der Hindernisdatei wird ihr Typ auf „Hinderniszelle“ umgesetzt.
Eine beispielhafte Hindernisdatei für ein 50x50x50 Zellen großes Strömungsgebiet mit einen Würfel der Kantenlänge 2 als Hindernis in der Mitte würde demnach folgendermaßen aussehen:

```
50 50 50
24 24 24
24 24 25
24 25 25
24 25 24
25 24 24
25 24 25
25 25 25
25 25 24
```

Da in dieser Arbeit periodische Randbedingungen eingesetzt wurden, genügt es, pro Dimension einen Rand anzugeben. D.h. es ist nicht notwendig bei einem Strömungsgebiet mit $dy = 20$ sowohl auf $y = 0$ als auch auf $y = 19$ einen Rand zu setzen, um das Strömungsgebiet beidseitig zu begrenzen, es genügt die Angabe von $y = 0$ **oder** $y = 19$.
Zusätzlich sei erwähnt, dass in der Hindernisdatei keine Kommentarzeilen erlaubt sind.

Parameterdatei „lb.par“

Die Parameterdatei legt Eigenschaften des Fluids fest und gibt an, in welchen Intervallen Ergebnisse auf Platte geschrieben werden sollen (siehe dazu auch A.2). Die Parameterdatei kann beliebig viele Kommentarzeilen enthalten, die jeweils mit einem „#“ eingeleitet werden. Die unten genannte Zeilennummerierung bezieht sich damit auf alle Nicht-Kommentarzeilen.

1. Anzahl der Rechenschritte für die gesamte Simulation
2. Anzahl der Rechenschritte zwischen zwei Dumps
3. Dichte des Fluids
4. Beschleunigung der Fluidzellen an den Positionen $X = 0, Y = *, Z = *$
5. Kinematische Viskosität

Dichte des Fluids, Beschleunigung und kinematische Viskosität sind dabei abhängig vom gewählten Strömungsgebiet und so zu wählen, dass die in Kapitel 5.2 genannten Bedingungen für eine stabile Berechnung eingehalten werden.

A.2. Ausgabedateien

Es gibt zwei Arten von Ausgabedateien, Datendateien und eine Übersichtsdatei, die die Datendateien näher beschreibt.

Datendateien
Die Datendateien beinhalten Informationen über Geschwindigkeitsverteilungen und Druck in je einem Teil des Strömungsgebietes. Für jedes der in Kapitel 5.1.2.1 vorgestellten Untereinheiten des Strömungsgebietes wird solch eine Datei angelegt (siehe auch 5.6.2). Die Namensgebung folgt dabei der Struktur

```
lb_[Nummer des Dumps]_[Nummer des Untergebiets].dat
```

Im Vergleich zu den Eingabedateien handelt es sich hierbei um Binärdateien mit folgender Struktur:

Header:
Byte 0-3: Kleinste X-Koordinate der Zellen (32 Bit Integer in Netzwerk-Byteorder)
Byte 4-7: Kleinste Y-Koordinate der Zellen (32 Bit Integer in Netzwerk-Byteorder)
Byte 8-11: Kleinste Z-Koordinate der Zellen (32 Bit Integer in Netzwerk-Byteorder)
Byte 12-15: Anzahl der Zellen in X-Richtung (32 Bit Integer in Netzwerk-Byteorder)
Byte 16-19: Anzahl der Zellen in Y-Richtung (32 Bit Integer in Netzwerk-Byteorder)
Byte 20-23: Anzahl der Zellen in Z-Richtung (32 Bit Integer in Netzwerk-Byteorder)

Datenbereich:
Byte 24 bis (24+k*4*8): k mal vier Double Werte (8 Byte) der folgenden Struktur

```
struct Cell {
  double pressure;
  double speedX;
  double speedY;
  double speedZ;
};
```

und $k = dx_{Untereinheit} * dy_{Untereinheit} * dz_{Untereinheit}$ gemäß Byte 12 bis 23.

Die Reihenfolge der **Cell**-Einträge wird dabei gemäß folgendem Algorithmus erzeugt:

```
for (int z=0; z<dz; z++) {
  for (int y=0; y<dy; y++) {
    for (int x=0; x<dx; x++) {
      ...
    }
  }
}
```

D.h. es werden immer $dx_{Untereinheit}$ in X-Richtung benachbarte Zellen hintereinander in die Ausgabedatei geschrieben. Handelt es sich um eine Hinderniszelle, so werden die Werte „Cell.pressure“, „Cell.speedX“, „Cell.speedY“ und „Cell.speedZ“ jeweils auf „-999“ gesetzt, was ein Ausfiltern beim Postprocessing bzw. der Visualisierung ermöglicht.

Übersichtsdatei
Die Übersichtsdatei enthält in menschenlesbarer Form die Zuordnung zwischen den Nummern der einzelnen Untereinheiten und deren Position im Gesamtgebiet. Sie enthält pro Untereinheit eine Zeile mit sieben Zahlen (in Textdarstellung)

1. Nummer der Untereinheit gemäß der o.g. Dateinamen
2. Kleinste X-Koordinate der Zellen (siehe Beschreibung der Datendatei, Offset 0)
3. Kleinste Y-Koordinate der Zellen (siehe Beschreibung der Datendatei, Offset 4)
4. Kleinste Z-Koordinate der Zellen (siehe Beschreibung der Datendatei, Offset 8)
5. Anzahl der Zellen in X-Richtung (siehe Beschreibung der Datendatei, Offset 12)

6. Anzahl der Zellen in Y-Richtung (siehe Beschreibung der Datendatei, Offset 16)
7. Anzahl der Zellen in Z-Richtung (siehe Beschreibung der Datendatei, Offset 20)

Diese Daten sind redundant, da sie ebenso durch Lesen der Header aller Datendateien ermittelt werden könnten, allerdings sind diese erstens in Binärform gespeichert und damit nicht ohne weiteres menschenlesbar und zweitens muss so nicht jede Datendatei geöffnet werden um zu bestimmen, ob sie in einem gewünschten Ausschnitt des Strömungsgebietes liegt.

ANHANG B

Funktionsbeschreibung des Beispielclients

Die folgenden Abschnitte stellen eine kurze Beschreibung der Client-Anwendung dar und sollen dem Benutzer die Arbeit mit dem Programm erleichtern sowie die Funktionsweise der in Kapitel 5.3.5 vorgestellten Steuerungsschnittstelle erläutern.
Voraussetzung für den Einsatz des Clients ist, dass die Simulation auf einem über TCP/IP vom Client aus erreichbaren Rechner gestartet wurde (siehe auch 5.3).

Nach dem Start des Clients präsentiert dieser sich mit dem in Abb. B.1 gezeigten Fenster.

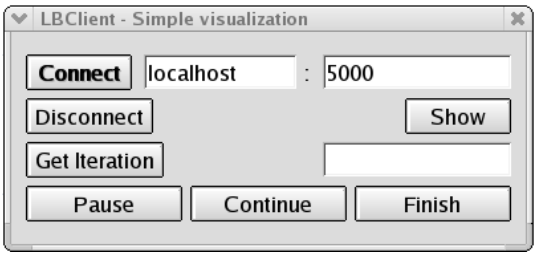

Abbildung B.1.: Hauptfenster der Client-Anwendung

Über den Button *Connect* wird eine Verbindung zum nebenstehenden Server (hier *localhost* auf TPC/IP-Port *5000*) aufgebaut.
Die drei Buttons *Pause*, *Continue* und *Finish* erlauben ein Unterbrechen und Fortführen bzw. Abbrechen der laufenden Simulation. Diese Befehle wirken direkt auf den Server (bzw. den dahinter liegenden Rechnerverbund).
Über den Button *Disconnect* kann der Client jederzeit vom Server getrennt werden, sowohl während einer laufenden Berechnung, als auch in einer Pause-Phase. Eine solche Trennung hat keinerlei Einfluss auf die serverseitige Berechnung und kann somit beliebig oft im Wechsel mit einem *Connect* erfolgen.

Ist man nur am Fortschritt der Simulation interessiert und nicht an einer Visualisierung der Ergebnisdaten, so lässt sich über den Button *Get Iteration* der aktuell berechnete Zeitschritt ermitteln. Dieser wird daraufhin im nebenstehenden Textfeld angezeigt. Ansonsten lassen sich über den Button *Show* die im im Folgenden beschriebenen Fenster öffnen.

Nach erfolgreichem Verbindungsaufbau bzw. nach Drücken des *Show*-Buttons öffnet sich das in Abb. B.2 dargestellte Fenster, in dem sich die zu visualisierenden Daten spezifizieren lassen.

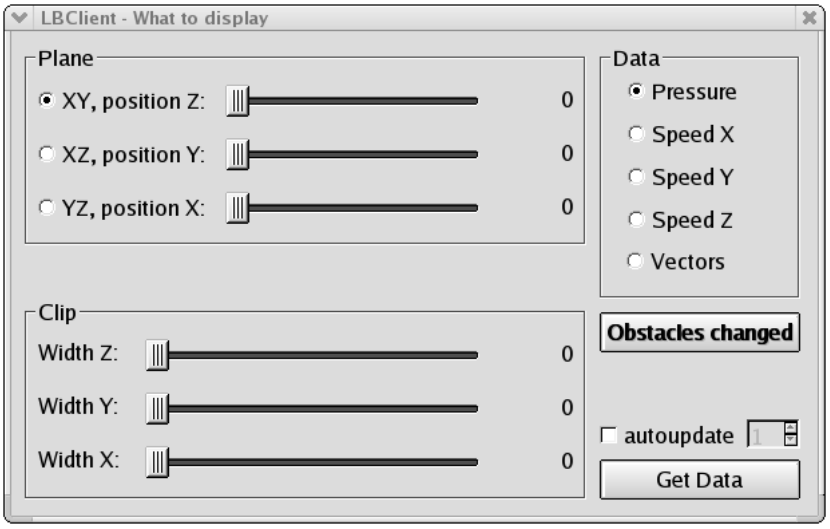

Abbildung B.2.: Auswahl des Visualisierungs-Ausschnittes

Im oberen linken Bereich finden sich drei Auswahlfelder, anhand derer die anzuzeigende Ebene ausgewählt werden kann:

- **XY**: Seitenansicht, senkrecht zur Hauptflußrichtung X+
- **XZ**: Draufsicht, senkrecht zur Hauptflußrichtung X+
- **YZ**: Durchsicht, in Hauptflußrichtung X+

Daneben sind jeweils Schieberegler angebracht, die die Position der Ebene in der fehlenden Dimension angeben.
Im unteren linken Bereich kann nun die Größe des Ausschnittes festgelegt werden. Da es sich um eine 2D-Visualisierung handelt, wird die Größenangabe für die nicht dargestellte Dimension ignoriert, d.h. wurde oben z.B. die XY-Ebene ausgewählt, so bleibt das Ändern von *Width Z* ohne Auswirkungen.

Im rechten oberen Bereich *Data* erfolgt die Auswahl des zu visualisierenden Datentyps. Zur Auswahl stehen Druck bzw. Geschwindigkeitskomponenten der drei Hauptrichtungen X,Y und Z sowie eine Vektordarstellung. Die Vektordarstellung enthält dabei zusätzlich zur farblichen Darstellung der Absolutgeschwindigkeit auch Pfeile, die die Richtung des Flusses angeben (projiziert auf die darzustellende Ebene).

Über das Selektionsfeld *autoupdate* kann ein Intervall eingestellt werden, wie oft die Visualisierung mit aktuellen Daten aufgefrischt werden soll. Die angegebene Zahl ist dabei die Anzahl von Zeitschritten, um welche die Berechnung fortlaufen soll bis eine Aktualisierung erfolgt.

Jede der beschriebenen Änderungen (Ausschnitt, Datentyp und Autoupdate) wird erst mit Aktivieren des *Get Data*-Buttons ausgeführt, was das Einstellen über die Schieberegler deutlich erleichtert.

Der *Obstacles changed*-Button erfüllt seine Aufgabe im Zusammenspiel mit dem in Abb. B.3 beschriebenen Fenster.

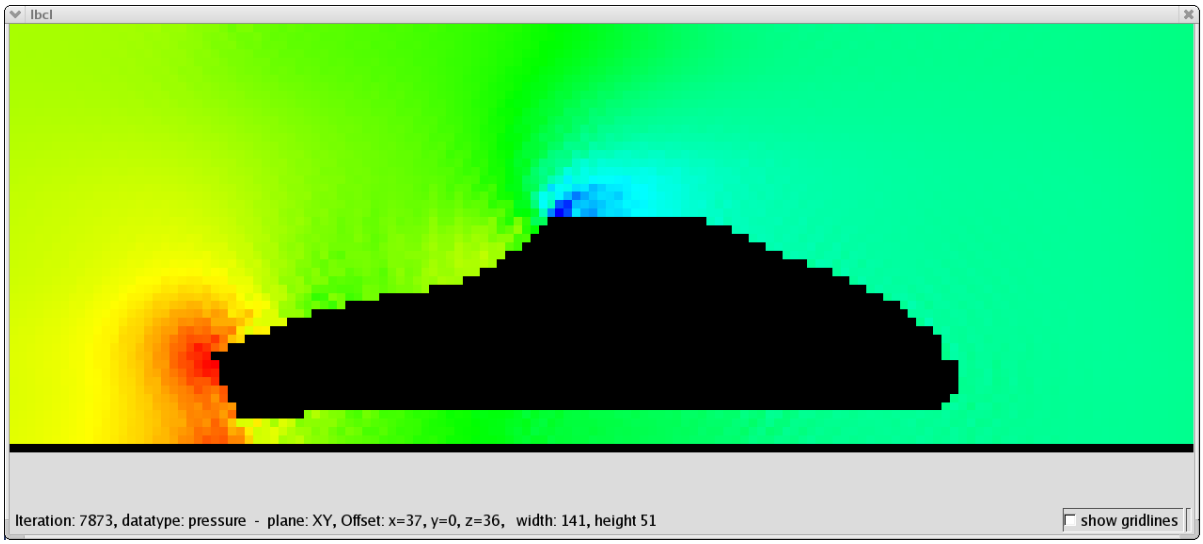

Abbildung B.3.: Visualisierung der Daten (Druck)

In diesem Fenster werden die vorher ausgewählten Daten gemäß Kapitel 5.4.2 visualisiert. Zusätzlich werden in der Statuszeile die Position und Dimension des Ausschnittes sowie der aktuelle Zeitschritt und Datentyp angezeigt.

Über den Schalter *show gridlines* können zusätzlich zur Visualisierung Gitterlinien eingeblendet werden, die die Grenzen der Normzellen anzeigen. Bewegt der Benutzer den Mauszeiger über den dargestellten Ausschnitt, so wird in der Statuszeile zudem die Position der Maus in den Koordinaten des Strömungsgebietes angezeigt.

Durch Anklicken einer Zelle kann deren Typ geändert werden. Fluidzellen werden dabei zu Hinderniszellen und Hinderniszellen zu Fluidzellen. Ähnlich wie im Fenster B.2 beschrieben, erfolgt die Änderung nicht unmittelbar, sondern erst nach Betätigen des in jenem Fenster angezeigten *Obstacles changed*-Buttons, was ein flüssigeres Modifizieren von mehreren Zellen erlaubt. Vor der Betätigung des *Obstacles changed*-Buttons werden die so modifizierten Zellen schraffiert dargestellt, um ihre vorgesehene Änderung anzuzeigen. Anschließend wird die Visualisierung automatisch aktualisiert.

Im in Abb. B.4 dargestellten Fenster lassen sich nicht nur der Typ einzelner Zellen ändern, sondern auch einfache geometrische Objekte (Kugeln und Quader) bearbeiten. Je nach gewähltem Element werden dabei im unteren Bereich die Geometriedaten spezifiziert. Beim Quader ist die Angabe der Position (x, y und z) sowie der Ausmaße (dx, dy und dz) erforderlich, bei der Kugel die Position des Mittelpunktes (x, y, und z) sowie des Radius (r).
Mit dem Auswahlfeld *Operation* kann festgelegt werden, wie das spezifizierte Element in das Strömungsfeld eingebettet werden soll. Wählt man *Add obstacles*, so wird das Element als Hindernis eingefügt, ungeachtet des Typs der bisher an dieser Stelle befindlichen Zellen. Wählt man *Remove*

obstacles, so werden alle betroffenen Zellen in Fluidzellen umgewandelt. Bei *Flip obstacles* hingegen wird der Typ jeder einzelnen der betroffenen Zellen auf genau den anderen Wert gesetzt. Hindernis-zellen werden somit zu Fluidzellen und Fluidzellen zu Hinderniszellen.
Durch Betätigen des *Ok*-Buttons werden die Änderungen an den Server übertragen und die Visualisierung automatisch aktualisiert.

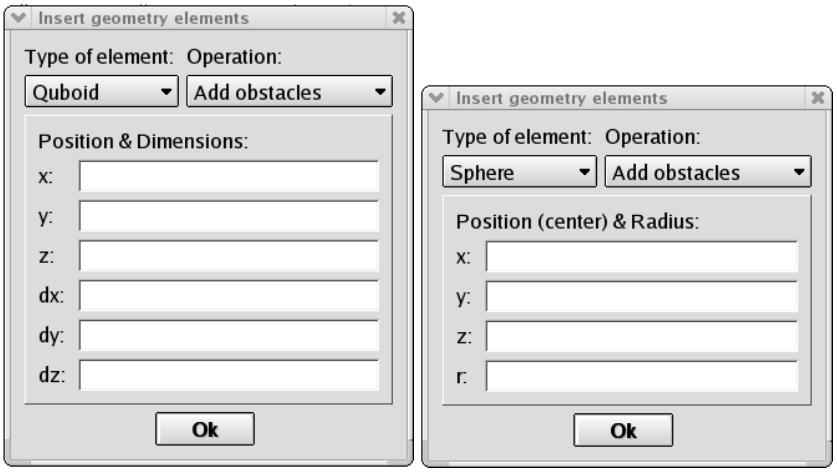

Abbildung B.4.: Einfügen/Löschen von Hindernissen; zwei verschiedene Ansichten, je nach ausgewähltem Geometrieelement (Quader/Kugel)

ANHANG C

Nachrichtenformat der Client-Server-Kommunikation

Die Nachrichtenübertragung erfolgt wie Kapitel 5.3.2 beschrieben über die beiden Seiten gemeinsame Klasse *Command*. Dabei wird die jeweilige Objektinstanz serialisiert, über die bestehende Netzwerkverbindung übertragen und auf Empfängerseite rekonstruiert. Um diese Serialisierung und anschließende Rekonstruktion zu ermöglichen, wird zusätzlich zu den eigentlichen Daten des Kommandos ein Header vorangestellt, der unter anderem Angaben für die Rekonstruktion enthält.

Kommando-Header

Byte 0-3: Kommandokennung (32 Bit Integer in Netzwerk-Byteorder)
Byte 4-7: Dringlichkeitsklasse (32 Bit Integer in Netzwerk-Byteorder)
Byte 8-11: Nachrichtensequenznummer (32 Bit Integer in Netzwerk-Byteorder)
Byte 12-15: Länge der Nutzdaten in Byte (32 Bit Integer in Netzwerk-Byteorder)
Byte 16-19: TTL (32 Bit Integer in Netzwerk-Byteorder)
Byte 20-23: Intervalllänge für evtl. Wiederholung (32 Bit Integer in Netzwerk-Byteorder)

Die Kommandokennung spezifiziert die Klasse und ist somit verantwortlich für die Rekonstruktion nach der Übertragung. Anhand der Dringlichkeitsklasse wird festgelegt, von welchem Handler das Kommando bearbeitet werden soll (Konsistenzwahrung). Da die Kommunikation asynchron erfolgt, kann anhand der Nachrichtensequenznummer eine Antwort dem jeweiligen Kommando zugeordnet werden. Sie wird automatisch beim Erzeugen eines Kommandos generiert. Das Längenfeld gibt die Anzahl der auf diesen Header folgenden Bytes an Nutzdaten an, d.h. die Gesamtlänge eines Kommandos entspricht diesem Wert zuzüglich den $6 \cdot 4 = 24$ Byte des Headers. TTL und Intervalllänge spezifizieren die automatische Wiederholung eines Kommandos.

Das Implementieren eines neuen Kommandos erfolgt dadurch, dass eine von *Command* abgeleitete Klasse erzeugt wird, die folgende virtuelle Methoden überschreibt:

```
class MyCommand : public Command {
public:
  bool readLoad(char* data, int length);
  string writeLoad();
};
```

readLoad() wird nach der Rekonstruktion des Kommandos aufgerufen und ermöglicht dem Kommando, die übertragenen Nutzdaten auszuwerten. Analog wird vor der Übertragung *writeLoad()* aufgerufen, wobei von der Kommandoklasse erwartet wird, die Nutzdaten zurückzuliefern.

Auf diese Art wurden die folgenden Kommandos implementiert:

- **CmdAdministration** Ermöglicht das Pausieren/Fortsetzen/Beenden einer Simulation und gibt den Fortschritt in Form des aktuell berechneten Zeitschritts zurück
- **CmdGetData** Liefert Geschwindigkeits- und Druckwerte für alle Zellen eines definierbaren Ausschnittes zurück
- **CmdModifyObstacles** Erlaubt das Modifizieren des Zelltyps (Hindernis- oder Fluidzelle) beliebiger Zellmengen

Zusätzlich muss eine weitere von *ServerCommand* abgeleitete Klasse erzeugt werden, die die eigentliche Ausführung des Kommandos übernimmt:

```
class MySrvCommand : public ServerCommand {
public:
  void executeInternal();
};
```

executeInternal() wird dabei vom Handler aufgerufen sobald der Systemzustand dies ohne Gefährdung der Konsistenz erlaubt. Von *ServerCommand* abgeleitete Klassen können dabei während der Ausführung von *executeInternal()* auf den Simulationskontext, das Serverinterface sowie den Handler zugreifen.

ANHANG D

Weitere Messungen

Kurz vor Abschluss dieser Arbeit ergab sich noch die Möglichkeit, den entstandenen Code auf dem institutseigenen Cluster auszuführen und seine Leistungsfähigkeit zu messen. Bei diesem Cluster handelt es sich um das folgende System:

- 64 Knoten + 1 Master
- 2 Intel Xeon 3,066 MHz CPUs (1 MB Cache) je Knoten
- 4 GB DDR Hauptspeicher je Knoten
- Infiniband für Kommunikation zwischen den Knoten
- Betriebssystem RedHat Linux 9.0 mit Kernel 2.4.21 SMP

Einige der aus diesen Messungen abgeleiteten Ergebnisse sind überraschend, da sie sich auf dem während der Entwicklung eingesetzten Cluster (siehe 5.2.3) so nicht andeuteten.

Konfiguration/#CPUs	1	2	4	8	16	32	64	128
1 CPU, 1 Prozess	1045	1019	976	999	935	789	537	-
2 CPUs, 2 Prozesse	-	748	703	663	600	455	357	212
2 CPUs, 2 Threads	-	724	669	574	576	474	409	208

Tabelle D.1.: Anzahl der berechneten Zellen pro Sekunde und CPU (in 1000) beim Einsatz von 1 bis 128 CPUs, jeweils ein bzw. zwei CPUs pro Knoten)

Tabelle D.1 stellt eine Zusammenfassung der im Folgenden beschriebenen Ergebnisse dar und dient dem übersichtlichen Vergleich.

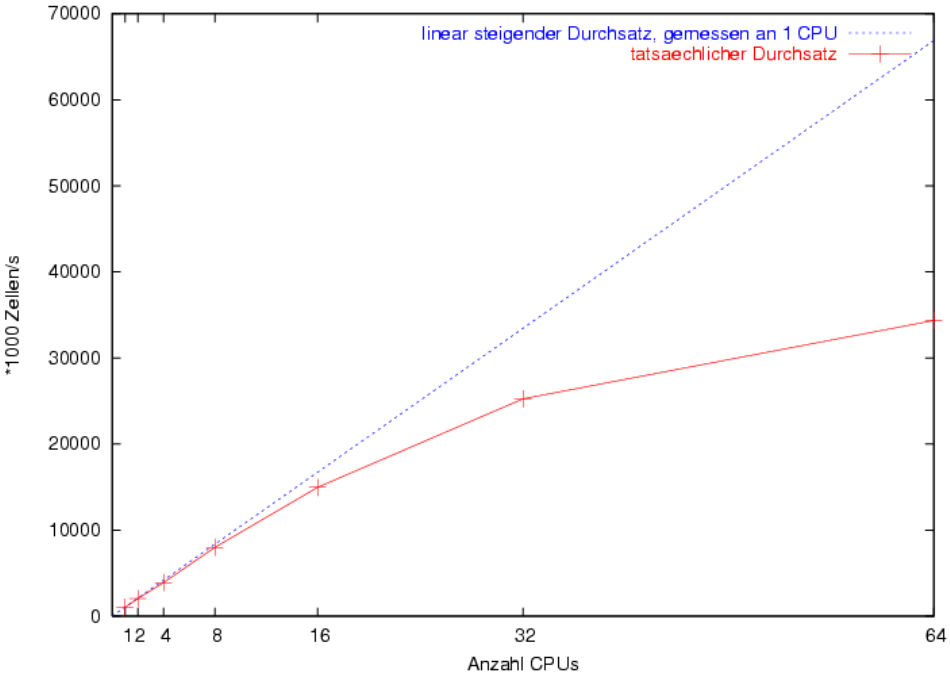

Abbildung D.1.: Anzahl berechneter Zellen pro Sekunde bei einer CPU pro Knoten

Abb. D.1 zeigt den Durchsatz bei folgender Code-Konfiguration:

- 1,2,4,8,16,32,64 CPUs
- pro Doppelprozessorknoten ein MPI-Prozess
- separater MPI-Thread (siehe 5.2.4.1)
- keine zusätzlichen Workerthreads (siehe 5.2.4.2)

Es wird deutlich, dass bei gegebener Problemgröße der Durchsatz über die Anzahl der Prozessoren relativ nahe am Optimum liegt bis 16 Prozessoren erreicht sind (jeder Prozessor erbringt dann noch ca. 90% der Spitzenleistung). Bei 32 Prozessoren fällt der Durchsatz ab auf ca. 75% und bei 64 Prozessoren gar auf etwas mehr als 50%.
Diese 50% müssen allerdings im folgenden Zusammenhang gesehen werden: Der bei der Entwicklung des Codes eingesetzte Cluster (siehe 5.2.3) konnte bei der Simulation den Master-Knoten mit einbeziehen, und dementsprechend wurde der Code geschrieben. Der für diese Messungen benutzte Cluster hingegen kann ausschließlich auf den 64 Knoten rechnen, wobei der Master-Knoten nur für administrative Zwecke genutzt wird. Da der Code allerdings auf einen Masterprozess angewiesen ist, musste dieser zusätzlich auf einem der Rechenknoten laufen, was auf jenem Knoten die für die Berechnung zur Verfügung stehende Zeit und damit den Gesamtdurchsatz merklich reduziert (da alle anderen Knoten auf den überlasteten Knoten warten müssen, um in den jeweils nächste Zeitschritt zu wechseln).
In der Messung D.1 fällt dies nicht übermäßig ins Gewicht (im Gegensatz zu den Messungen D.2 bzw. D.3), da die zweite CPU der Knoten kaum beansprucht wird, dennoch dürfte es das ein oder andere Prozent Durchsatz kosten, da alle Prozessoren auf den letzten warten müssen.

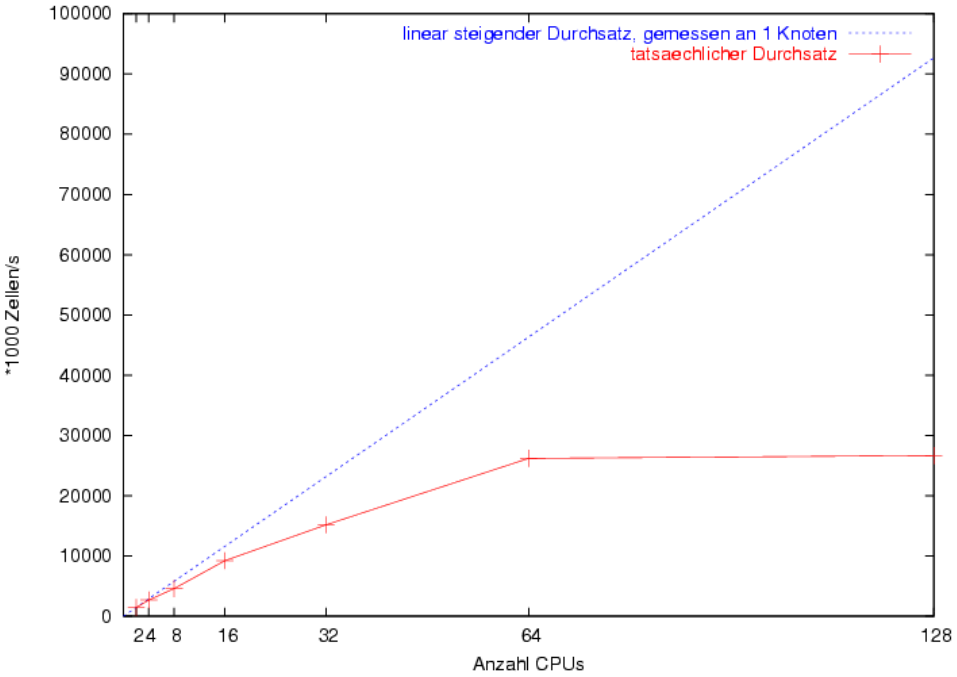

Abbildung D.2.: Anzahl berechneter Zellen pro Sekunde bei zwei CPUs pro Knoten

Abb. D.2 zeigt die Durchsatzsteigerung bei folgender Code-Konfiguration:

- 2,4,8,16,32,64,128 CPUs
- pro Doppelprozessorknoten zwei MPI-Prozesse
- jeweils separater MPI-Thread
- keine zusätzlichen Workerthreads

Schon auf dem in der Entwicklung eingesetzten Cluster hat sich gezeigt, dass zwei Prozesse auf einem Knoten keine hohe Steigerung des Durchsatzes bringen, da offensichtlich die Speicherbandbreite einen stark limitierenden Faktor darstellt (siehe 5.2.3).
Auf dem für diese Messung genutzten Cluster zeichnet sich ein ähnliches Bild, wenn auch nicht so drastisch wie in Abb. 5.11, hier liegt der Gewinn durch den zweiten Prozessor bei immerhin knapp 40%.
Die in Abb. D.2 gezeigte Steigerung des Durchsatzes liegt bis 64 Prozessoren (32 Knoten) leicht unterhalb der Steigerung beim Einsatz von nur einer CPU pro Knoten. Bei 128 eingesetzten Prozessoren jedoch kommt das unter der vorigen Messung D.1 angesprochene Problem des zusätzlichen Master-Prozesses voll zum Tragen:
Aus Gründen die mangels Zeit nicht mehr genauer untersucht werden konnten, läuft der Masterprozess auf einem Prozessor bei knapp 100%iger Auslastung (auf dem Entwicklungscluster waren es deutlich unter 2%). Das bedeutet, dass der mit dem Masterprozess zusätzlich belastete Knoten effektiv nur zu 50% für die eigentliche Berechnung zur Verfügung steht. Da aber keine Lastbalancierung vorgesehen war, konnte diese Situation nicht adäquat behandelt werden, mit dem Ergebnis, dass alle anderen Prozessoren auch nur mit ca. 50% rechnen konnten (Synchronisation). Da sich dieses Pro-

blem aber sicherlich durch den ein oder anderen Kunstgriff beheben lässt (es ist nicht einzusehen, wofür der Prozessor zu 100% ausgelastet ist, der Masterprozess stellt letztlich nur den zentralen Synchronisationspunkt), lassen sich hier vermutlich deutlich bessere Ergebnisse erzielen.

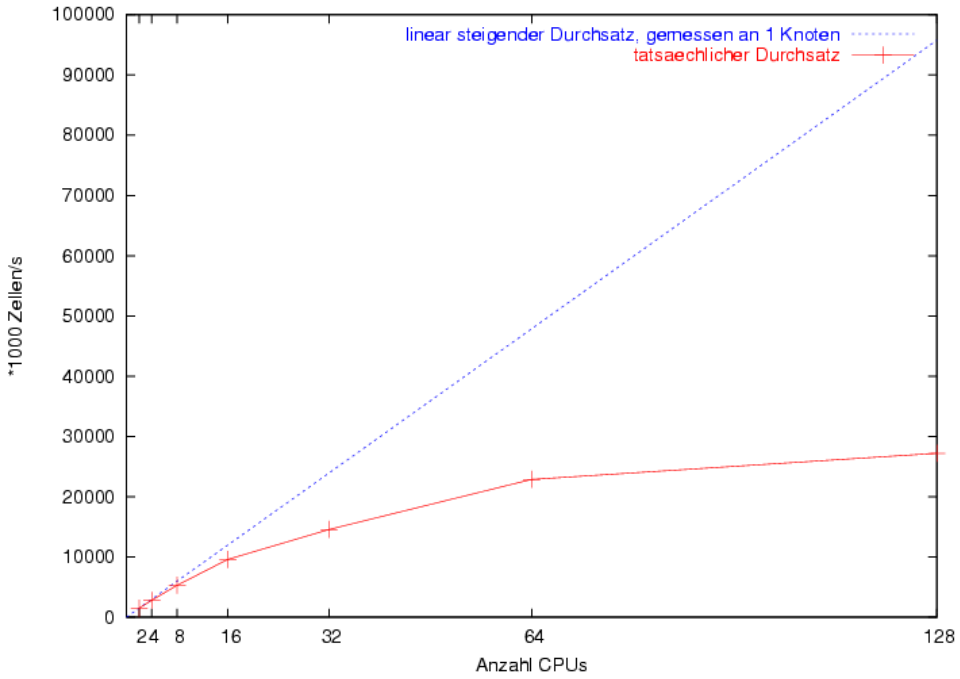

Abbildung D.3.: Anzahl berechneter Zellen pro Sekunde bei zwei CPUs, ein MPI-Prozess pro Knoten

Abb. D.3 zeigt das den Durchsatz bei folgender Code-Konfiguration:

- 2,4,8,16,32,64,128 CPUs
- pro Doppelprozessorknoten ein MPI-Prozess
- separater MPI-Thread
- ein zusätzlichen Workerthread

Hier sind zwei Tatsachen auffallend: Zum einen zeigt sich das Problem des zusätzlichen Masterprozesses beim Einsatz aller 128 Prozessoren, daher die geringe Steigerung zwischen 64 und 128 CPUs. Allerdings gibt es eine solche, wenn auch nur minimal. Das ist darauf zurückzuführen, dass in den nicht parallelisierten Phasen (siehe 5.2.4.2) der zweite Prozessor quasi nicht genutzt wird, was wiederum Rechenzeit für den Masterprozess übrig lässt.
Gravierender ist allerdings die Tatsache, dass die vorliegende Konfiguration etwas langsamer als die in Messung D.2 eingesetzte ist, was im Gegensatz zu den Messungen auf dem Entwicklungscluster steht (Abb. 5.11). Um die Gründe für dieses Verhalten zu finden, ist wohl eine detailierte Laufzeitanalyse notwendig, ohne diese kann über Ursachen nur spekuliert werden.

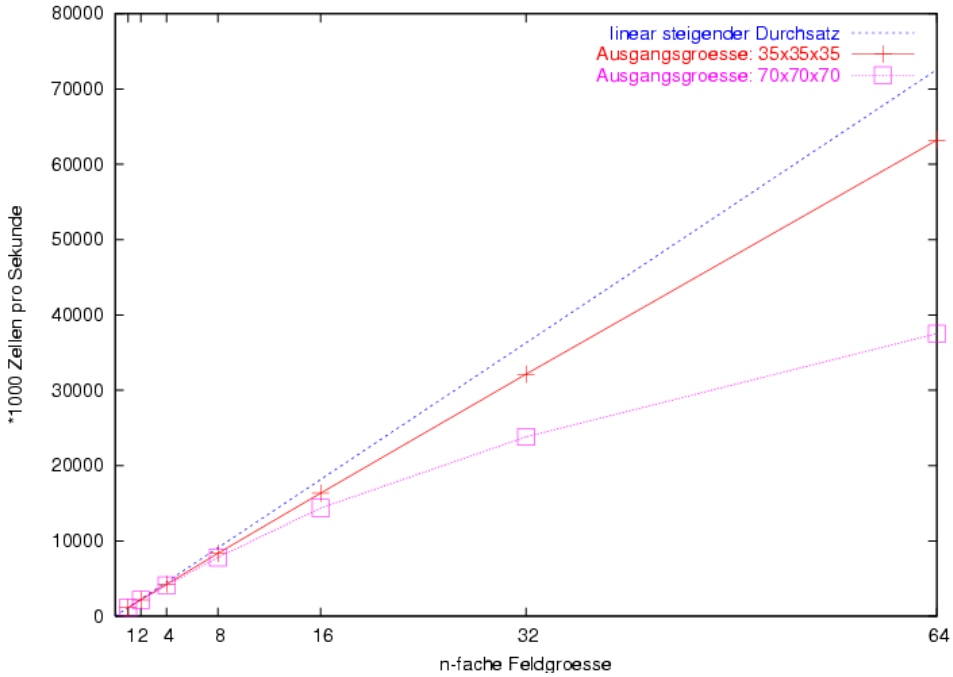

Abbildung D.4.: Skalierung über die Feldgröße

In Abb. D.4 zeigt sich das Skalieren des Codes über die Größe des Simulationsfeldes. Die eingesetzte Konfiguration entspricht dabei derjenigen aus Messung D.1, d.h. ein eingesetzter Prozessor pro Knoten.
Mit jeder Verdopplung der Prozessorenzahl wurde auch die Größe des Simulationsfeldes verdoppelt, was in der Messung 35x35x35 auch in einer nahezu linearen Steigerung des Durchsatzes resultiert (Faktor ca. $0,9$). Die zweite Messung mit größerem Ausgangsfeld (70x70x70) weicht hier jedoch sowohl von der Linearität, als auch von der prozentualen Steigerung ab.

Anhand dieser Grafik wird einmal mehr deutlich, mit welcher Vorsicht derart präsentierte Ergebnisse zu verstehen und zu verwenden sind. Mit einigem Recht hätte hier behauptet werden können, der Code skaliere quasi linear, wäre nur die 35x35x35-Messung präsentiert worden. Für reale Anwendungen ist die damit erreichbare Auflösung (bei 64 Prozessoren 140x140x140 Zellen) offensichtlich aber nicht ausreichend. Ein Schritt um in diesem Fall die Skalierbarkeit zu verbessern könnte die in Kapitel 5.2.2.1 und Abb. 5.10 angedeutete Zusammenfassung von MPI-Nachrichten sein.

Zusätzlich sollte angemerkt werden, dass die hier eingesetzte MPI-Implementierung (MPICH2 mit Unterstützung für Infiniband) verschiedentlich (hauptsächlich bei großen Strömungsgebieten) Programm-Abstürze verursachte. Inwieweit die Ursache für diese Abstürze in der MPI-Implementierung oder dem Simulationscode zu suchen ist, konnte jedoch nicht mehr geklärt werden.

Literaturverzeichnis

[1] Bernsdorf, Jörg: *anb is not best*, C&C Research Laboratories, NEC Europe Ltd. (2001), http://ccrl-nece.de/lba/

[2] Crispin, Mark R.: *INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1*, University of Washington, Network Working Group (2003)

[3] Free Software Foundation: *GNU GENERAL PUBLIC LICENSE Version 2*, http://www.gnu.org/copyleft/gpl.html

[4] Freudiger, Sören: Diplomarbeit *Effiziente Datenstrukturen für Lattice-Boltzmann Strömungssimulationen in der computergestützten Strömungsmechanik*, TU München (2001)

[5] Gergova, Milena: Bachelor's Thesis *Evaluation of Improved Boundary Conditions for the Lattice Boltzmann Approach: Investigation of the Laminar Vortex Street behind a Circular Cylinder*, Universität Erlangen-Nürnberg (2002)

[6] GNU gprof, http://www.gnu.org/software/binutils/manual/gprof-2.9.1/

[7] GTS: *The GNU Triangulated Surface Library*, http://gts.sourceforge.net/

[8] Kandhai, Drona: Dissertation *Large Scale Lattice-Boltzmann Simulations*, Universität Amsterdam (1999)

[9] Krafczyk, Manfred: Habilitationsschrift *Gitter-Boltzmann-Methoden: Von der Theorie zur Anwendung*, TU München (2001)

[10] MPI: *Message Passing Interface standard*, version 1.1 (1995), http://www-unix.mcs.anl.gov/mpi/

[11] OpenDX, http://www.opendx.org/

[12] OpenMP, http://www.openmp.org/specs/

[13] Pnueli, Armin: *Liste von Veröffentlichungen zum Thema „reactive systems“*, http://www.wisdom.weizmann.ac.il/ amir/c-and-j.html

[14] Pressman, Roger S.: *Software Engineering - A Practitioner's Approach*, McGraw-Hill (1996)

[15] Rabenseifer, Rolf: *Hybrid Parallel Programming on HPC Platforms*, Fifth European Workshop on OpenMP, Aachen (2003)

[16] Rabenseifer, Rolf: Kursmaterial *Parallel Programming Workshop*, Universität Stuttgart (2004)

[17] Trolltech, *QT 3.2*, http://www.trolltech.org

[18] Tannenbaum, Andrew S.: *Operating Systems: Design and Implementation*, Prentice Hall (1997)

[19] Tölke, Jonas: Dissertation *Gitter-Boltzmann-Verfahren zur Simulation von Zweiphasenströmungen*, TU München (2001)

[20] XML: *Extensible Markup Language (XML)*, http://www.w3.org/XML/

(WWW-Adressen Stand 29. Juni 2004)

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfaßt und nur die angegebenen Quellen benutzt zu haben.

_____

(Fabian Kaiser)