High Performance Simulation of Free Surface Flows Using the Lattice Boltzmann Method

Hochleistungssimulation von Strömungen mit freien Oberflächen basierend auf der Lattice-Boltzmann-Methode

Der Technischen Fakultät der
Universität Erlangen-Nürnberg
zur Erlangung des Grades

DOKTOR–INGENIEUR

vorgelegt von

DIPL.-PHYS. THOMAS POHL

Erlangen, 2008

Als Dissertation genehmigt von

der Technischen Fakultät der

Universität Erlangen-Nürnberg

Tag der Einreichung:	26. November 2007
Tag der Promotion:	19. Juni 2008
Dekan:	Prof. Dr. J. Huber
Berichterstatter:	Prof. Dr. U. Rüde
	Prof. Dr. H.-J. Bungartz

Abstract

Continuing increases in computing power open the discipline of computer simulation to more and more complex applications. One example is the simulation of free surface flows. Thies, Körner et al. proposed a model to simulate such flows in the context of material science. However, their model is limited to two spatial dimensions. One major topic of the work presented in this thesis is the extension of their model to 3D. Various modifications to the algorithms are described in detail. In particular, the calculation of the surface curvature required a completely different approach which is now based on a local triangulation of the fluid/gas interface.

As the second major topic of this thesis, various aspects of high performance computing are discussed on the basis of the implementation of the 3D model. An example is the optimization of the spatial and temporal data locality which is difficult due to the complexity of this model. Furthermore, different optimization strategies for parallel computing are discussed in detail. The application of the presented performance optimizations is not limited to this specific model implementation, but can be generalized to a wide range of complex simulation codes.

Acknowledgments

First of all, I would like to thank my advisor Prof. Dr. Ulrich Rüde (System Simulation Group, University of Erlangen-Nuremberg) for his support, countless stimulating discussions, and, finally, for his patience. Furthermore, I wish to thank Dr. Carolin Körner (Institute of Science and Technology of Metals, University of Erlangen-Nuremberg) and Thomas Zeiser (Regional Computing Center Erlangen). Without their friendly willingness to share their expertise about the lattice Boltzmann method and computational fluid dynamics in general, this work would not have been possible. I also thank Prof. Dr. Hans-Joachim Bungartz for reviewing this thesis.

During the work on this thesis, a small group of scientists working on lattice Boltzmann related topics began to form at the System Simulation Group. I would like to thank the members of this group, in particular, Klaus Iglberger, Stefan Donath, and Christian Feichtinger for many constructive discussions.

Thanks go out to the ensemble of proofreaders: Uwe Fabricius, Christoph Freundl, Ben Bergen, Stefan Donath, and Frank Ortmeier (Institute for Software Engineering, University of Augsburg).

As a computer scientist working in the field of high performance computing, I worked on several supercomputers and computer clusters. The system operators did a great job in supporting my work. Thus, I would like to thank Frank Deserno (System Simulation Group, University of Erlangen-Nuremberg), Dr. Gerhard Wellein and Dr. Georg Hager (Regional Computing Center Erlangen), and all the other system operators for their help. Furthermore, I wish to thank the open source community for providing all these tools that we take for granted in our daily work.

Finally, I express my deepest gratitude to my wife Tina, my parents and grandparents for their loving support, and their trust and patience over all these years.

Tom Pohl

Contents

We are what we think. All that we are arises with our thoughts. With our thoughts, we make the world.

Buddha (Siddhartha Gautama)

Chapter 1

Introduction

1.1 Scope

The work presented in this thesis is split into three major parts:

- Extension of a model for the simulation of free surface flows, based on the work of Thies, Körner et al. [Thi05, KTH+05].
- Development of an efficient parallel implementation for this extended model.
- Analysis of various high performance computing aspects, based on this implementation.

While the first part is located in the area of computational fluid dynamics and physics, the two remaining parts concentrate on the field of computer science. The topic of high performance computing is a central part of the entire thesis and is examined in detail in the last part.

1.2 Motivation

The simulation of fluid dynamics using computers is a well-established field of research. Due to its practical importance and its wide field of application, various methods have been devised to solve the fundamental Navier-Stokes equations. In 1986, Frisch, Hasslacher and Pomeau set the cornerstone for a new class of methods, the lattice-gas cellular automata [FHP86]. In the macroscopic limit, this billiard-like game resembles the Navier-Stokes equations without using a single floating-point operation. The disadvantages of this model gave rise to the development of lattice Boltzmann models, featuring a higher flexibility compared to their discrete precursor. In the following years, the approach gained momentum and is now being applied in a variety of fields, such as, magnetohydrodynamics and multiphase flows.

Two characteristics make lattice Boltzmann methods especially attractive to both scientists and engineers: First, the basic model is comparably easy to implement. Even more importantly, the model lends itself to extensions, such as, modelling free surface flows. Thies, Körner et al. developed such an extension for simulating the evolution of so-called metal foams [Thi05, KTH+05]. In simple terms, the behavior of such melts is similar to that of soap bubbles. The complex structure of these foams and their ever-changing geometry make them an ideal application for the lattice Boltzmann method. The model by Thies et al. provides an unparalleled insight into the rheology and morphology of metal foams, however, it is restricted to two spatial dimension.

When extending this model to 3D, it becomes obvious that some of the employed algorithms are no longer applicable. So, the first goal in the course of this thesis is to replace some of the methods by alternative approaches, which allow for the extension to three spatial dimensions.

Another problem of the initial implementation is that the enlarged simulation domain in 3D would lead to a prohibitive increase in the required computing time. The discipline of high performance computing studies various ways to increase the computational performance of a code, such as, data access optimizations or parallel computing. However, the investigation and application of these techniques is often limited to comparably simple codes. Therefore, the second major challenge of this work is to transfer these methods to the complex computational kernel of this free surface flow model.

1.3 Outline

In Chapter 2, the description for the model of free surface flows, based on the lattice Boltzmann method, is given. Both the underlying model by Thies, Körner et al., and the extension to 3D and required modifications, developed in the course of this thesis, are described.

Chapter 3 outlines the implementation of the model. The model data and the basic sequential program flow are briefly discussed. Three typical simulation setups, which are used in the following sections, are introduced. In addition to these qualitative tests, we investigate a quantitative validation experiment in detail.

In Chapter 4, we discuss several aspects of high performance computing, while applying different optimization techniques to the model implementation. Two major topics are the optimization of data access, and the realization of parallel computing.

Two selected projects of related work are presented in Chapter 5: The precursor to the model implementation, without the free surface extension, has been contributed to the SPEC benchmark suite CPU2006. The second topic is the integration of an LBM implementation into a grid computing framework, based on the programming language Java.

In Chapter 6, final conclusions are presented. As an outlook, we discuss the future

development of the model for free surface flows and its implementation in Chapter 7.

Chapter 2

Model Description

This chapter gives a detailed description of a model to simulate free surface flows in 3D using the lattice Boltzmann method. This model is based on the work of Thies, Körner et al. [Thi05, KTH+05] who developed the according 2D model. The description does not cover the derivations of all equations. Instead, the essential results are presented and pointers to the corresponding literature are given. Where appropriate, methods are described in pseudocode notation.

2.1 Overview

The presented model for simulation free surface flows can be split up into two major parts as depicted in Fig. 2.1, the fluid phase and the gas phase model.

2.1.1 Gas Phase Model

Since the flow dynamics of the gas phase are neglected due to its low density compared to the fluid phase density, the gas phase model just stores the initial volume and the current volume for a set of separated gas volumes, so-called bubbles (see Fig. 2.1, right). The current gas pressure in the bubble linearly depends on the quotient of both values. The coalescence of two bubbles is treated correctly (see Sec. 2.4), whereas the segmentation of one bubble to several separate bubbles is not modeled (see Sec. 2.6.1).

2.1.2 Fluid Phase Model

The fluid phase model is more complex. It is based on a spatial discretization of the computational domain. At any given time, each of the resulting grid points, so-called cells, belongs to exactly one of four different cell types (see Fig. 2.1, left):

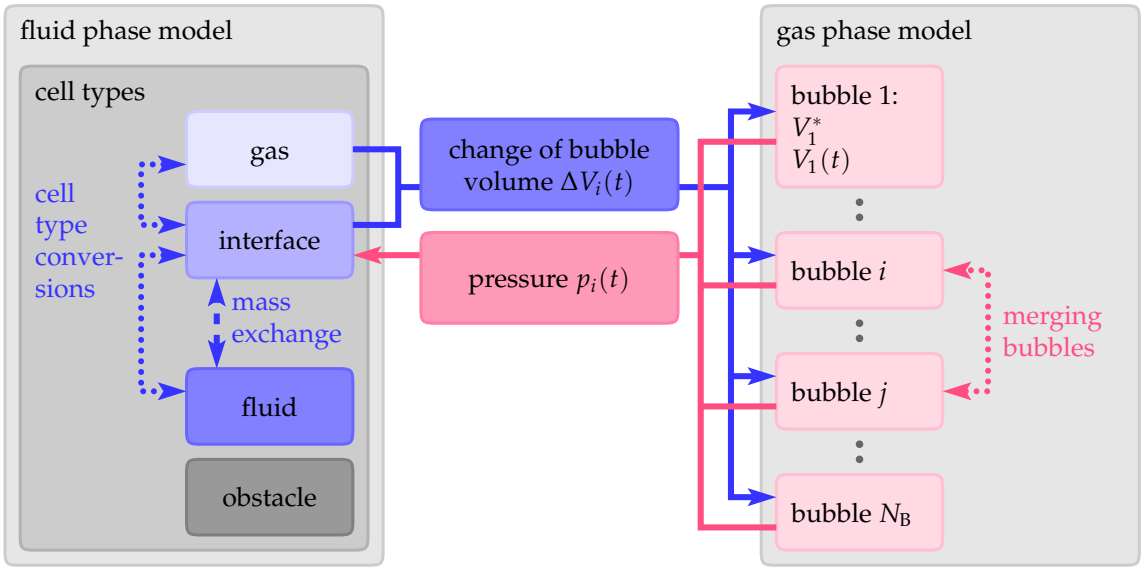


Figure 2.1: Overview of the model for free surface flows.

gas cells: No computations are necessary for this cell type, since the flow dynamics of the gas phase are neglected.

interface cells: Because of the employed algorithms, it is necessary to separate fluid and gas cells with so-called interface cells. Details about this requirement are given in Sec. 2.2.

fluid cells: These cells are handled with an ordinary fluid dynamics code using the lattice Boltzmann method (LBM).

obstacle cells: This is the only cell type which does not change in time. It is used to model fixed domain boundaries and obstacles within the computational domain.

To denote all cells of a certain type, we use the four sets $\mathcal{C}^{\mathrm{O}}$, $\mathcal{C}^{\mathrm{F}}$, $\mathcal{C}^{\mathrm{G}}$, and $\mathcal{C}^{\mathrm{I}}$ with the first letter of the cell type as an index. A typical setup of the discretized computational domain for 2D is shown in Fig. 2.2. The strict separation of fluid and gas cells by the interface cells becomes obvious. The extension of this concept to 3D is straightforward.

Each cell is described by a set of state variables. The type of the cell determines which state variables are defined as listed in Tab. 2.1. Below is a brief description of these state variables and their use within the model. Further details will be given in the appropriate sections:

distribution functions F_i: This set of values is used in the lattice Boltzmann method to model the fluid dynamics.

bubble index b: Cells which represent some part of a bubble, store an index which uniquely identifies the bubble.

cell types	fluid	interface	gas	obstacle
distribution functions F_i	•	•	-	-
bubble index b	-	•	•	-
fluid fraction φ	-	•	-	•
mass m	-	•	-	-
surface point $\boldsymbol{p}$	-	•	-	-
surface normal $\boldsymbol{n}$	-	•	-	-

Table 2.1: The cell type determines which state variables are defined for each cell. A bullet (•) means that the variable is defined; a dash (-) means that the variable is undefined.

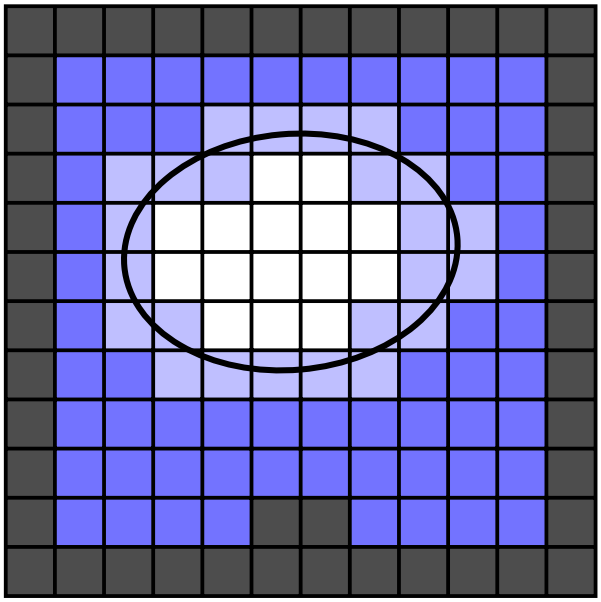


Figure 2.2: An example of a computational domain in 2D: The fluid cells (*dark blue*) are surrounded by obstacle cells (*dark gray*). Inside the fluid lies a bubble consisting of the inner gas cells (*white*) and interface cells (*light blue*).

fluid fraction φ: To track the fluid/gas interface on a subgrid scale, each interface cell features a so-called fluid fraction which is defined as the portion of the cell's volume filled with fluid.

mass m: The mass m of an interface cell is used for bookkeeping to guarantee local mass conservation. For fluid cells, the mass is stored implicitly in the distribution functions as described in Sec. 2.2.1. The mass of the gas phase is neglected.

surface point $\boldsymbol{p}$ and surface normal $\boldsymbol{n}$: Both values are required to calculate the surface curvature for interface cells.

Because of the movement of the fluid, mass is exchanged between fluid and interface cells (see Sec. 2.2.6). Thus, the fluid fractions change and an interface cell might become completely filled or emptied which causes its conversion to a fluid or gas cell, respectively. Consequently, neighboring fluid and/or gas cells have to be converted to interface cells in order to fulfill the requirement that fluid cells are not adjacent to gas cells (see Sec. 2.4).

The movement of the fluid might also induce the change of a bubble volume which is the only possibility for the fluid phase model to influence the gas phase model. In return, the gas phase model determines the gas pressure in each bubble which influences the interface cells of the fluid phase model.

Apart from the gas pressure, the behavior of interface cells also depends on the surface curvature which is a result of the geometric shape of the interface layer (see Sec. 2.5).

In the context of the fluid phase model, each of the bubbles, mentioned in the gas phase model, is a a set of gas and interface cells with the same bubble index.

2.2 Augmented Lattice Boltzmann Method for Free Surface Flows

This part of the model contains the typical LBM equations and the extensions which are required to deal with free surface flows.

2.2.1 Discretization of the Boltzmann Equation

The following section gives only a brief description of the basic ideas of the LB method. The starting point is the Boltzmann equation

$$\frac{\partial f}{\partial t} + v\boldsymbol{\nabla} f = Q \quad , \tag{2.1}$$

which describes the evolution in time of the particle distribution function $f = f(\boldsymbol{x}, \boldsymbol{v}, t)$ with the spatial position $\boldsymbol{x}$, the velocity $\boldsymbol{v}$, and the time t [Bol72, Har04]. According to this equation, the evolution of f depends on the inner product of the velocity $\boldsymbol{v}$ and the gradient of f with respect to spatial dimensions $\boldsymbol{\nabla} f$, and on the collision operator Q. Informally speaking, Q describes the interaction of particles when they collide.

Bhatnagar et al. [BGK54] and Welander [Wel54] proposed a simple linear collision operator

$$Q = -\frac{1}{\tau}\left(f - f^0\right) \tag{2.2}$$

based on the assumption that each collision changes the distribution function f by an amount proportional to the deviation of f from the Maxwellian distribution function

$$f^0 = f^0\big(\rho(\boldsymbol{x}, t), \boldsymbol{v}(\boldsymbol{x}, t)\big)$$

with ρ as the local density. The relaxation time τ describes how fast a disturbed distribution function f converges towards f^0. The simplification of using just one relaxation time degrades the accuracy of this so-called BGK model as investigated in [PLM06]. A more sophisticated model for the collision operator will be presented in Sec. 2.2.2. With the collision operator Q of Eq. 2.2, Eq. 2.1 becomes

$$\frac{\partial f}{\partial t} + \boldsymbol{v}\boldsymbol{\nabla} f = -\frac{1}{\tau}\left(f - f^0\right) \quad .$$

The next step is the transition from the continuous to a discretized model with respect to space, velocity and time. For this purpose, we introduce a finite set of velocities $\{\boldsymbol{v}_i\}$, $0 \leq i \leq n$. Accordingly, the distribution function f is now also limited to n discrete values which gives us the discrete Boltzmann equation

$$\frac{\partial f_i}{\partial t} + \boldsymbol{v}_i\boldsymbol{\nabla} f_i = -\frac{1}{\tau}\left(f_i - f_i^0\right) \quad .$$

This formula can be non-dimensionalized by introducing a reference length scale L_r, a reference speed V_r, a reference density η_r, and the characteristic time between particle collisions t_r

$$\frac{\partial F_i}{\partial \hat{t}} + \boldsymbol{c}_i\hat{\boldsymbol{\nabla}} F_i = -\frac{1}{\hat{\tau}\epsilon}\left(F_i - F_i^0\right) \quad ,$$

with

$$\begin{aligned} F_i &:= f_i/\eta_\mathrm{r} \\ F_i^0 &:= f_i^0/\eta_\mathrm{r} \\ \hat{t} &:= tV_\mathrm{r}/L_\mathrm{r} \\ \boldsymbol{c}_i &:= \boldsymbol{v}_i/V_\mathrm{r} \\ \hat{\boldsymbol{\nabla}} &:= L_\mathrm{r}\boldsymbol{\nabla} \\ \hat{\tau} &:= \tau/t_\mathrm{r} \\ \epsilon &:= t_\mathrm{r}V_\mathrm{r}/L_\mathrm{r} \quad . \end{aligned}$$

The parameter ϵ can be interpreted either as the Knudsen number, i.e., the ratio of the mean free path to the characteristic length or as the ratio of the collision time to the flow time. Details can be found in [WG00].

To simplify the presentation of the formulas, we return to the familiar variables t, $\boldsymbol{\nabla}$, and τ:

$$\frac{\partial F_i}{\partial t} + \boldsymbol{c}_i\boldsymbol{\nabla} F_i = -\frac{1}{\tau}\left(F_i - F_i^0\right) \quad .$$

In the final step, we approximate the derivatives with respect to the spatial dimensions $\boldsymbol{\nabla} F_i$ and the time $\frac{\partial F_i}{\partial t}$ by first-oder difference schemes. The grid spacing Δx, the time step Δt, and the lattice velocities $\boldsymbol{c}_i$ are chosen such that $\Delta x/\Delta t = \boldsymbol{c}_i$. This means that the particles corresponding to the lattice velocities $\boldsymbol{c}_i$ cover exactly the chosen grid spacing Δx in

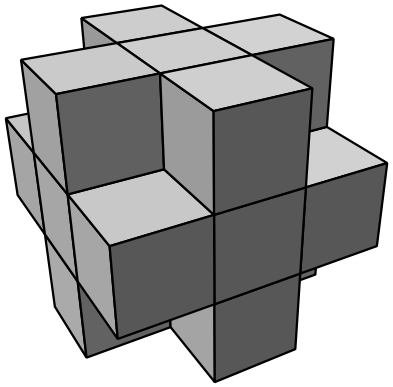


Figure 2.3: Each cube belongs to the set of 18 neighboring cells $\mathcal{N}_{x,y,z}$ around the (hidden) central cell located at $\langle x, y, z \rangle$ for the D3Q19 model.

each time step Δt. This results in the lattice Boltzmann equation for the BGK approximation

$$F_i\left(\boldsymbol{x} + \boldsymbol{c}_i \Delta t, t + \Delta t\right) - F_i\left(\boldsymbol{x}, t\right) = -\frac{1}{\tau}\Big[F_i\left(\boldsymbol{x}, t\right) - F_i^0\big(\rho(\boldsymbol{x}, t), \boldsymbol{v}(\boldsymbol{x}, t)\big)\Big] \quad . \tag{2.3}$$

For practical reasons, the application of this formula is often divided into two separate steps, the stream operation

$$F_i'\left(\boldsymbol{x}, t\right) = F_i\left(\boldsymbol{x} - \boldsymbol{c}_i \Delta t, t - \Delta t\right) \tag{2.4}$$

and the collide operation

$$F_i\left(\boldsymbol{x}, t\right) = F_i'\left(\boldsymbol{x}, t\right) - \frac{1}{\tau}\Big[F_i'\left(\boldsymbol{x}, t\right) - F_i^0\big(\rho(\boldsymbol{x}, t), \boldsymbol{v}(\boldsymbol{x}, t)\big)\Big] \tag{2.5}$$

with the intermediate distribution functions F_i'. While the ordering of these two steps marginally influences the results, the steady-state solutions are equal [WG00]. For this implementation, the stream/collide sequence was chosen.

Since we will use the so-called D3Q19 model (see Sec. 2.2.3), the notion of "neighboring cells" refers to the eighteen cells which exchange distribution functions with a central cell as depicted in Fig. 2.3. This set of 18 cells around a central cell located at $\langle x, y, z \rangle$ will be denoted as $\mathcal{N}_{x,y,z}$.

While the stream operation (Eq. 2.4, Fig. 2.4) describes the propagation of the distribution functions, i.e., the propagation of information in the system, the collision operation (Eq. 2.5) determines for each grid point (also called fluid cell) how the distribution functions evolve because of particle interactions.

The macroscopic flow quantities as the mass density $\rho(\boldsymbol{x}, t)$ and the momentum density $\boldsymbol{j}(\boldsymbol{x}, t)$ can be derived from the distribution functions as

$$\begin{aligned} \rho(\boldsymbol{x}, t) &:= \sum_i F_i\left(\boldsymbol{x}, t\right) \quad , \\ \boldsymbol{j}(\boldsymbol{x}, t) &:= \rho(\boldsymbol{x}, t)\boldsymbol{v}(\boldsymbol{x}, t) = \sum_i \boldsymbol{c}_i F_i\left(\boldsymbol{x}, t\right) \quad . \end{aligned} \tag{2.6}$$

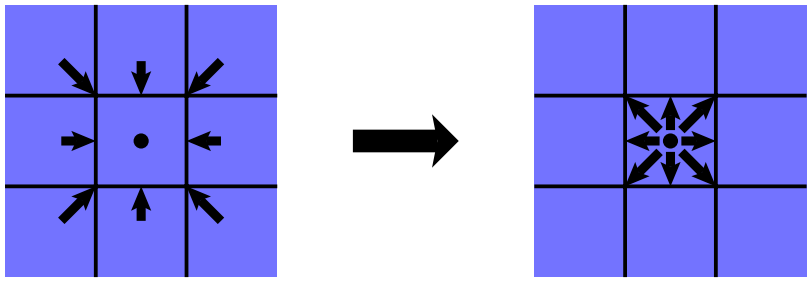


Figure 2.4: Stream operation for the D2Q9 model: For each cell (*blue squares*), the stream operation collects the distribution function (*black arrows and dot*) of neighboring cells pointing towards the center cell and stores them in the center cell.

Another important macroscopic value is the kinematic viscosity ν. It is found [Suc01] to be

$$\nu = \frac{c^2}{3}\left(\tau - \frac{\Delta t}{2}\right) \quad . \tag{2.7}$$

A high value for the viscosity corresponds to a viscous fluid (e.g., honey), whereas a low value characterizes thin fluids (e.g., water).

2.2.2 Two Relaxation Time Model

The single relaxation time approximation is the most important reason for several artifacts of the BGK approximation. Some improved schemes have been devised in the past, most importantly the intricate multi-relaxation time model [dGK+02] in order to mitigate these artifacts. A trade-off between complexity and computational effort on the one side, and simplicity and correctness on the other side, is the two relaxation time (TRT) model which is described in [Gin05]. This section briefly presents the TRT model without giving its derivation.

In the following equations, symmetric quantities are marked with a bar ($\bar{}$); antisymmetric quantities are marked with a hat ($\hat{}$).

First, we define a new set of symmetric and antisymmetric distribution functions as

$$\bar{F}_i = \frac{F_i + F_{\bar{\imath}}}{2} \quad ; \quad \hat{F}_i = \frac{F_i - F_{\bar{\imath}}}{2} \quad ,$$

where $F_{\bar{\imath}}$ is the distribution function which corresponds to the lattice velocity $\boldsymbol{c}_{\bar{\imath}}$ pointing in the opposite direction compared to $\boldsymbol{c}_i$:

$$\boldsymbol{c}_{\bar{\imath}} = -\boldsymbol{c}_i \quad .$$

In the special case of the center cell with $\boldsymbol{c}_C = \langle 0, 0, 0 \rangle$, these quantities are defined as $\bar{F}_C = F_C$ and $\hat{F}_C = 0$. Similarly, two new sets of equilibrium distribution functions are

defined as

$$\bar{F}_i^0 = \frac{F_i^0 + F_{\bar{i}}^0}{2} \quad ; \quad \hat{F}_i^0 = \frac{F_i^0 - F_{\bar{i}}^0}{2} \quad .$$

For the TRT model, the collision operator changes to

$$F_i' = F_i - \lambda_e \left(\bar{F}_i - \bar{F}_i^0 \right) - \lambda_0 \left(\hat{F}_i - \hat{F}_i^0 \right) \quad ,$$

with $\lambda_e = \tau^{-1}$, $\lambda_0 = \left(\frac{1}{2} + \frac{3}{16\tau - 8} \right)^{-1}$.

2.2.3 The D3Q19 Model

Several different 3D LBM models have been developed which differ in the quantity of discrete lattice velocities $\boldsymbol{c}_i$. The most prominent ones are the so-called D3Q15, D3Q19, and D3Q27 models with 15, 19, and 27 lattice velocities, respectively. Mei et al. showed [MSYL00] that the D3Q19 model features the best compromise in terms of accuracy and stability on the one side, and numerical efficiency on the other side. In this model, the 19 lattice velocities are defined as

$$\boldsymbol{c}_i := \begin{cases} \langle 0,0,0 \rangle & \text{for} \quad i = C \\ \langle \pm c,0,0 \rangle & \text{for} \quad i = E,W \\ \langle 0,\pm c,0 \rangle & \text{for} \quad i = N,S \\ \langle 0,0,\pm c \rangle & \text{for} \quad i = T,B \\ \langle \pm c,\pm c,0 \rangle & \text{for} \quad i = NE,NW,SE,SW \\ \langle \pm c,0,\pm c \rangle & \text{for} \quad i = ET,EB,WT,WB \\ \langle 0,\pm c,\pm c \rangle & \text{for} \quad i = NT,NB,ST,SB \end{cases} ,$$

which is depicted in Fig. 2.5. We choose $c = 1$ for simplicity.

The Maxwellian distribution functions F_i^0 only depend on the local values of density $\rho(\boldsymbol{x},t)$ and velocity $\boldsymbol{v}(\boldsymbol{x},t)$. For the D3Q19 model, their values are computed as

$$F_i^0(\rho,\boldsymbol{v}) = \omega_i \rho(\boldsymbol{x},\boldsymbol{t}) \left[1 - \frac{3}{2} \frac{\boldsymbol{v}(\boldsymbol{x},t)^2}{c^2} \right] \quad ; \quad \omega_i = \frac{1}{3}$$

for $i = \mathrm{C}$,

$$F_i^0(\rho,\boldsymbol{v}) = \omega_i \rho(\boldsymbol{x},\boldsymbol{t}) \left[1 + 3\frac{\boldsymbol{c}_i \cdot \boldsymbol{v}(\boldsymbol{x},t)}{c^2} + \frac{9}{2} \frac{(\boldsymbol{c}_i \cdot \boldsymbol{v}(\boldsymbol{x},t))^2}{c^4} - \frac{3}{2} \frac{\boldsymbol{v}(\boldsymbol{x},t)^2}{c^2} \right] \quad ; \quad \omega_i = \frac{1}{18}$$

for $i =$ N, S, E, W, T, B, and

$$F_i^0(\rho,\boldsymbol{v}) = \omega_i \rho(\boldsymbol{x},\boldsymbol{t}) \left[1 + 3\frac{\boldsymbol{c}_i \cdot \boldsymbol{v}(\boldsymbol{x},t)}{c^2} + \frac{9}{2} \frac{(\boldsymbol{c}_i \cdot \boldsymbol{v}(\boldsymbol{x},t))^2}{c^4} - \frac{3}{2} \frac{\boldsymbol{v}(\boldsymbol{x},t)^2}{c^2} \right] \quad ; \quad \omega_i = \frac{1}{36}$$

for $i =$ NE, NW, SE, SW, ET, EB, WT, WB, NT, NB, ST, SB. A detailed derivation of these equations is given in [WG00].

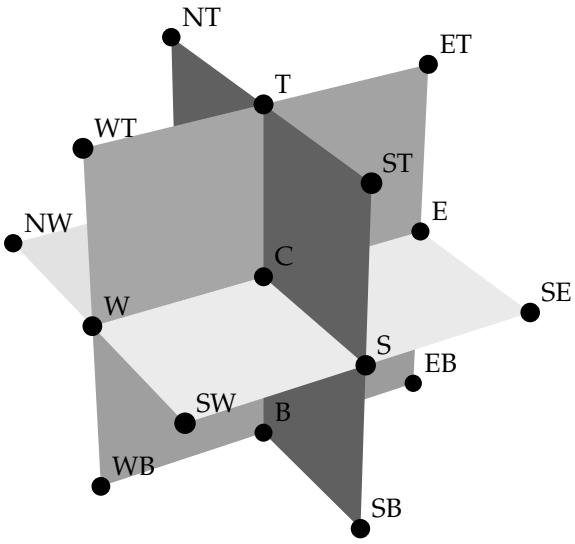


Figure 2.5: The 19 discrete lattice velocities $\boldsymbol{c}_i$ for the D3Q19 model. The letters stand for East and West ($x = \pm 1$), North and South ($y = \pm 1$), and Top and Bottom ($z = \pm 1$).

2.2.4 Boundary Conditions for Obstacle Cells

This implementation of the LB method uses the so-called no-slip boundary conditions for obstacle cells which is realized by a bounce-back scheme. No-slip means that the mean velocity at the border of an obstacle is zero. The characteristics of the bounce-back scheme will be described in the following.

As mentioned before, we define a lattice velocity $\boldsymbol{c}_{\bar{i}}$ pointing in the opposite direction compared to $\boldsymbol{c}_i$

$$\boldsymbol{c}_{\bar{i}} = -\boldsymbol{c}_i \quad .$$

During the stream operation described by Eq. 2.4, the distribution functions F_i of each cell are shifted along their associated lattice velocities $\boldsymbol{c}_i$ to the neighboring cells.

If a cell at the location $\boldsymbol{x}$ has an obstacle neighbor in the direction $\boldsymbol{c}_i$, it will not receive a distribution function from there, i.e., the distribution function $F'_{\bar{i}}(x, y, z)$ would be undetermined, because obstacle cells do not participate in the stream/collide operations. Instead, the distribution function $F_i(\boldsymbol{x})$, which would be sent to the obstacle cell, is assigned to $F'_{\bar{i}}(\boldsymbol{x})$:

$$F'_{\bar{i}}(\boldsymbol{x}) = F_i(\boldsymbol{x}) \quad \text{for} \quad (\boldsymbol{x} + \boldsymbol{c}_i) \in \mathcal{C}^{\mathrm{O}} \quad .$$

The simple physical interpretation is the reflection of particles bouncing into an obstacle as shown in Fig. 2.6.

Inamuro et al. showed in [IYO95] that this method results in a no-slip boundary condition.

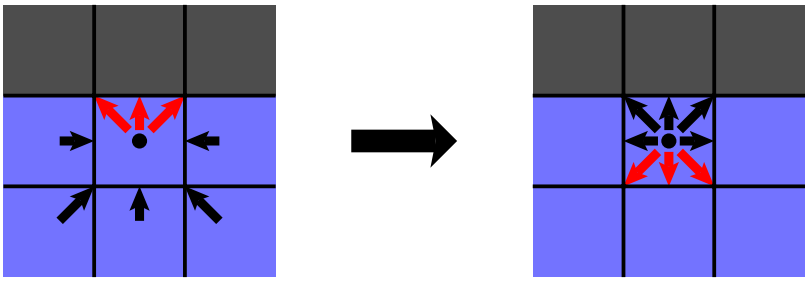


Figure 2.6: If obstacle cells (*gray squares*) prevent the reading of distribution functions during the stream operation, the distribution functions of the center cell which are pointing towards obstacle cells (*red arrows*) are used with reversed directions.

2.2.5 External Forces

The influence of an external force $\boldsymbol{S}$ can be integrated into the LBM scheme in many different ways. A detailed comparison can be found in [BG00]. In the course of this thesis, several methods have been implemented and tested to incorporate the effects of gravity. Because the influence on the results was negligible small, the computationally most efficient method was chosen which is described in [GA94, HZLD97].

This method only requires an additional term in the collision operation of Eq. 2.5

$$F_i(\boldsymbol{x},t) = F_i'(\boldsymbol{x},t) - \frac{1}{\tau}\Big[F_i'(\boldsymbol{x},t) - F_i^0\big(\rho(\boldsymbol{x},t),\boldsymbol{v}(\boldsymbol{x},t)\big)\Big] + \omega_i\rho(\boldsymbol{x},t)(\boldsymbol{S}\cdot\boldsymbol{c}_i) \quad . \tag{2.8}$$

In the special case of interface cells, the force term has to be weighted with the fluid fraction

$$F_i(\boldsymbol{x},t) = F_i'(\boldsymbol{x},t) - \frac{1}{\tau}\Big[F_i'(\boldsymbol{x},t) - F_i^0\big(\rho(\boldsymbol{x},t),\boldsymbol{v}(\boldsymbol{x},t)\big)\Big] + \omega_i\rho(\boldsymbol{x},t)\varphi(\boldsymbol{x},t)(\boldsymbol{S}\cdot\boldsymbol{c}_i) \quad . \tag{2.9}$$

similar to the volume of fluid method described in [GS02, GS03].

For the described model of free surface flows, an external force will be used to implement the influence of gravity.

2.2.6 Free Surface Model and Fluid Advection

The extension to model free surfaces used in this thesis is based on the work of Thies, Körner et al. [Thi05, KTH+05]. It is very similar to the volume of fluid methods where an additional variable, the fluid fraction φ, specifies the portion of the cell's volume filled with fluid. A value of 1 means that the interface cell is completely filled with fluid, 0 means that it is empty.

Apart from the fluid fraction, the fluid mass of a cell $m(\boldsymbol{x},t)$ is introduced as an additional state variable. Both new state variables and the density $\rho(\boldsymbol{x},t)$ are related by

$$m(\boldsymbol{x},t) = \rho(\boldsymbol{x},t)\cdot\varphi(\boldsymbol{x},t) \quad . \tag{2.10}$$

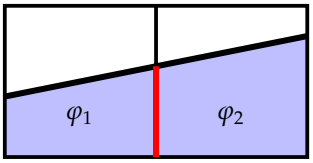


Figure 2.7: Geometric interpretation of the weighting factor in Eq. 2.11 for the mass exchange between two interface cells. Mass can only be exchanged inside the fluid phase (*blue area*) at the marked interface (*red line*). The length of this line is proportional to the mean fluid fraction of both cells φ_1 and φ_2.

Per definition, the fluid fraction and the mass are set to 0 for gas cells and to 1 for fluid and obstacle cells. The fluid fraction assigned to obstacle cells could be used to influence the contact angle at the fluid/obstacle interface to model a wetting or non-wetting behavior. Similar techniques are applied in the context of the volume of fluid method [RRL01, HM04]. However, this topic is beyond the scope of this thesis.

2.2.6.1 Fluid Advection

The movement of the fluid and the gas phase leads to a mass exchange among the cells. For fluid cells, this is modeled by the streaming of the distribution functions F_i. The mass exchange $\Delta m_i(\boldsymbol{x}, t)$ between an interface cell at the position $\boldsymbol{x}$ and its neighboring cell at $\boldsymbol{x} + \boldsymbol{c}_i$ depends on the type of the neighboring cell and is given as

$$\Delta m_i(\boldsymbol{x}, t) = \begin{cases} 0 & \text{for} \quad (\boldsymbol{x} + \boldsymbol{c}_i) \in (\mathcal{C}^{\mathrm{G}} \cup \mathcal{C}^{\mathrm{O}}) \\ F_{\bar{\imath}}(\boldsymbol{x} + \boldsymbol{c}_i, t) - F_i(\boldsymbol{x}, t) & \text{for} \quad (\boldsymbol{x} + \boldsymbol{c}_i) \in \mathcal{C}^{\mathrm{F}} \\ \frac{1}{2}\left[\varphi(\boldsymbol{x}, t) + \varphi(\boldsymbol{x} + \boldsymbol{c}_i, t)\right]\left[F_{\bar{\imath}}(\boldsymbol{x} + \boldsymbol{c}_i, t) - F_i(\boldsymbol{x}, t)\right] & \text{for} \quad (\boldsymbol{x} + \boldsymbol{c}_i) \in \mathcal{C}^{\mathrm{I}} \end{cases} . \tag{2.11}$$

No mass transfer is possible between interface and gas/obstacle cells. The mass transfer between fluid and interface cells is the difference of the particles that are streaming into the interface cell $F_{\bar{\imath}}(\boldsymbol{x} + \boldsymbol{c}_i, t)$ and the particles that are streaming out to the neighboring fluid cell $F_i(\boldsymbol{x}, t)$. For the mass exchange among interface cells, the same value as above is used weighted with the mean fluid fraction of both cells. The geometric interpretation of this factor for 2D is given in Fig. 2.7. The weighting factor is also used for 3D, although the geometric interpretation fails for neighboring cells which only share an edge. It is important to note that local mass conservation is given in Eq. 2.11, because mass that is lost in one cell is received by a neighboring cell and vice versa:

$$\Delta m_i(\boldsymbol{x}, t) = -\Delta m_{\bar{\imath}}(\boldsymbol{x} + \boldsymbol{c}_i, t) \quad .$$

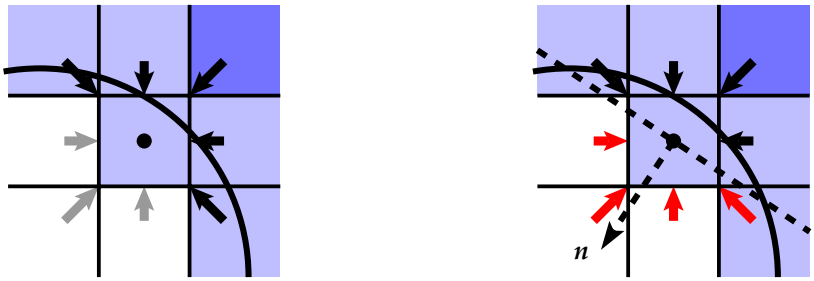


Figure 2.8: Instead of the three missing distribution functions (*left image, gray arrows*) from gas cells, four distribution functions are reconstructed (*right image, red arrows*) depending on the surface normal $\boldsymbol{n}$ (*right image, dashed arrow*).

The temporal evolution of the mass of an interface cell is given by the sum of all mass exchanges with neighboring cells

$$m(\boldsymbol{x}, t + \Delta t) = m(\boldsymbol{x}, t) + \sum_i \Delta m_i(\boldsymbol{x}, t) \quad .$$

2.2.6.2 Reconstruction of Missing Distribution Functions

Just as obstacle cells, gas cells do not have distribution functions as state variables. So, for the stream operation the same problem concerning missing distribution functions arises as for the obstacle cells. This is the reason for the introduction of interface cells. They ensure that missing distribution functions are reconstructed for the streaming operation and that distribution functions are not directly advected from a fluid cell to a gas cell.

The reconstruction of the missing distribution functions has to fulfill two boundary conditions. First, the velocity of the fluid and the gas phase have to be equal at the interface. Second, the known force performed by the gas has to be balanced by the force exerted by the fluid. Both requirements are met by the reconstruction method proposed by Körner et al. in [KTH+05]

$$F_i'(\boldsymbol{x} - \boldsymbol{c}_i, t) = F_i^0(\rho_G, \boldsymbol{v}) + F_{\bar{i}}^0(\rho_G, \boldsymbol{v}) - F_{\bar{i}}'(\boldsymbol{x}, t) \quad , \quad \boldsymbol{x} \in \mathcal{C}^I \wedge \left[(\boldsymbol{x} - \boldsymbol{c}_i) \in \mathcal{C}^G \vee \boldsymbol{c}_i \cdot \boldsymbol{n}(\boldsymbol{x}, t) < 0\right]$$

with the gas density ρ_G. The normal vector of the interface $\boldsymbol{n}(\boldsymbol{x}, t)$ will be defined in Sec. 2.5.1. With this method, not only the missing distribution functions from neighboring gas cells are reconstructed, but also distribution functions whose associated lattice velocity is pointing in the opposite direction of the interface normal $\boldsymbol{n}(\boldsymbol{x}, t)$ as shown in Fig. 2.8. A detailed derivation of the formula for the reconstruction is given in [KTH+05].

The gas density is influenced by two parameters: the gas pressure p_G which is equal for any point inside the bubble and the surface curvature $\kappa(\boldsymbol{x}, t)$ which depends on the local shape of the surface

$$\rho_G = 3p_G + 6\sigma_S\kappa(\boldsymbol{x}, t) \tag{2.12}$$

with the surface tension σ_S. By modifying the gas pressure for each cell according to the local curvature, the forces resulting from the surface tension are included. The calculation of the surface curvature κ will be described in Sec. 2.5. The handling of gas bubbles and the calculation of the gas pressure p_G is described in the next section.

2.3 Handling of Gas Volumes

Each separated volume of gas is treated as a so-called bubble. Each bubble b is characterized by its initial volume V_b^* and its current volume $V_b(t)$. The bubble $b = 0$ which is used to model the surrounding atmosphere with a constant pressure is treated specially. Its initial volume V_0^* is updated to be equal to $V_0(t)$ in each time step.

Assuming that each bubble initially starts with the atmosphere pressure, the normalized gas pressure, which is required in Eq. 2.12, is calculated as

$$p_G = \frac{V_b^*}{V_b(t)} \quad .$$

While the coalescence of bubbles is handled correctly as it is described in Sec. 2.4.4.2, the segmentation of a bubble is not covered in this model (see Sec. 2.6.1).

2.4 Cell Conversion

As mentioned before, the movement of the fluid results in a mass transfer among cells which can cause the conversion of cell types. Since different cell types feature different sets of state variables, the missing state variables for the new cell type have to be set. Depending on the initial cell type, certain conversion rules have to be applied.

2.4.1 General Conversion Rules

The basic conversion rules are shown in Fig. 2.9. A direct conversion from gas to fluid cells or vice versa is neither possible nor realistic, because the mass transfer between neighboring cells per time step is below 1/10 of the cell volume for any reasonable simulation setup.

The conversion of a cell does not only depend on the local fluid fraction φ, but also on the conversion of neighboring cells. This is necessary to ensure that fluid and gas cells are separated by a contiguous layer of interface cell. Therefore, if an interface cell converts to a fluid (gas) cell, neighboring gas (fluid) cells have to be converted to interface cells. In the case of contradicting conversion requirements, precedence has to be given to the separation of fluid and gas cells.

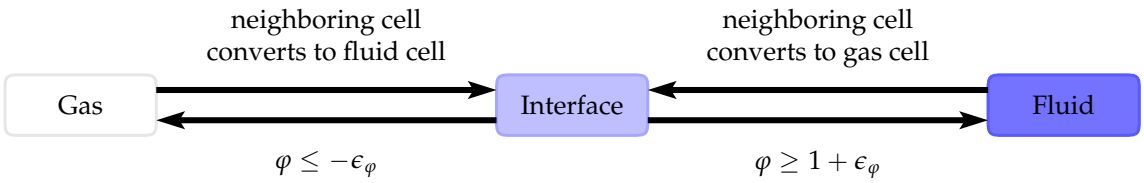


Figure 2.9: The conversion of a cell is triggered by the fluid fraction φ reaching a certain limit and by the conversion of neighboring cells. This might result in conflicting conversion requirements which has to be resolved.

The limit of the fluid fraction φ is not exactly 0.0 or 1.0 for a conversion to a gas or a fluid cell, respectively. Instead, the fluid fraction has to exceed an additional threshold of ϵ_φ in order to trigger a conversion. This prevents the unwanted effect of fast oscillations between cell types. For all performed simulations, ϵ_φ was heuristically chosen as 0.01.

2.4.2 Conversion of Interface Cells

Interface cells can either be converted to fluid or gas cells, depending on which limit their value for the fluid fraction reached. When mass is transfered among cells according to Eq. 2.11, it is not guaranteed that the resulting fluid fraction

$$\varphi(\boldsymbol{x}, t + \Delta t) = \varphi(\boldsymbol{x}, t) + \frac{\Delta m(\boldsymbol{x}, t + \Delta t)}{\rho(\boldsymbol{x}, t + \Delta t)}$$

is within the interval $[0, 1]$. If an interface cell located at $\boldsymbol{x}_\mathrm{c}$ would simply be converted to a gas cell ($\varphi = 0$) or a fluid cell($\varphi = 1$), this excess or missing mass

$$m_\mathrm{excess}(\boldsymbol{x}_\mathrm{c}, t) = \begin{cases} m(\boldsymbol{x}_\mathrm{c}, t) - 1 & \text{for transformation to fluid cell} \\ m(\boldsymbol{x}_\mathrm{c}, t) & \text{for transformation to gas cell} \end{cases}$$

would be lost. Though this single violation of mass conservation might be negligible small, this situation occurs frequently in the course of a simulation run and cannot be neglected. Therefore, this excess or missing mass is distributed equally among neighboring interface cells to preserve local and global mass conservation

$$m(\boldsymbol{x}, t + \Delta t) = m(\boldsymbol{x}, t) + \frac{m_\mathrm{excess}(\boldsymbol{x}_\mathrm{c}, t)}{\left|\mathcal{N}(\boldsymbol{x}_\mathrm{c}) \cap \mathcal{C}^\mathrm{I}\right|} \quad ; \quad \forall\, \boldsymbol{x} \in \left(\mathcal{N}(\boldsymbol{x}_\mathrm{c}) \cap \mathcal{C}^\mathrm{I}\right) \quad ,$$

where $|.|$ denotes the number of cells in the specified set of cells. If there are no neighboring interface cells, the excess mass cannot be distributed and is lost. In the course of the performed simulations, this situation hardly ever occurred.

2.4.3 Conversion of Gas Cells

If a cell is converted from a gas to an interface cell, a new set of distribution functions has to be created for this cell. Neighboring fluid or interface cells which are denoted as the set $\mathcal{R}$ are used to calculate a mean value for the density and the macroscopic velocity:

$$\begin{aligned} \mathcal{R} &= \mathcal{N}(\boldsymbol{x}_\mathrm{c}) \cap (\mathcal{C}^\mathrm{I} \cup \mathcal{C}^\mathrm{F}) \quad , \\ \bar{\rho}(\boldsymbol{x}_\mathrm{c}, t) &= \frac{1}{|\mathcal{R}|} \sum_{x \in \mathcal{R}} \rho(\boldsymbol{x}, t) \quad , \\ \bar{\boldsymbol{v}}(\boldsymbol{x}_\mathrm{c}, t) &= \frac{1}{|\mathcal{R}|} \sum_{x \in \mathcal{R}} \boldsymbol{v}(\boldsymbol{x}, t) \quad , \\ F_i(\boldsymbol{x}_\mathbf{c}, t) &= F_i^0\big(\bar{\rho}(\boldsymbol{x}_\mathrm{c}, t), \bar{\boldsymbol{v}}(\boldsymbol{x}_\mathrm{c}, t)\big) \quad . \end{aligned}$$

The other missing state variables belonging to an interface cell, such as, fluid fraction and mass are both set to 0.0, since the interface cell initially contains no fluid.

2.4.4 Conversion of Fluid Cells

2.4.4.1 General Conversion

Fluid cells can only be converted to interface cells. Two missing state variables can easily be determined: The fluid fraction φ is set to 1.0, since the cell is initially completely filled. Thus, the mass m and the density ρ have the same value (see Eq. 2.10) which can be determined from the distribution functions using Eq. 2.6.

Another state variable that has to be reconstructed is the bubble index. To set this local bubble index $b(\boldsymbol{x}_\mathrm{c}, t)$, all neighboring interface or fluid cells are tested for their bubble index:

$$\mathcal{B} = \left\{ b(\boldsymbol{x}, t) \;\middle|\; \boldsymbol{x} \in \left[\mathcal{N}(\boldsymbol{x}_\mathrm{c}) \cap (\mathcal{C}^\mathrm{I} \cup \mathcal{C}^\mathrm{F}) \right] \right\} \quad . \tag{2.13}$$

If only one individual bubble index can be found ($|\mathcal{B}| = 1$), the local bubble index is set to this value. If more than one unique bubble index is found, it is assumed that all the bubbles in the set $\mathcal{B}$ have coalesced.

2.4.4.2 Coalescence of Bubbles

The coalescence of two or more bubbles consists of several steps. The bubble information—namely the initial bubble volume V^* and the current bubble volume $V(t)$—of all bubbles in the set $\mathcal{B}$ as defined in Eq. 2.13 are subsumed in the bubble with the minimum bubble index in $\mathcal{B}$:

$$\begin{aligned} V^*_{\min(\mathcal{B})} &= \sum_{b \in \mathcal{B}} V^*_b \\ V_{\min(\mathcal{B})}(t) &= \sum_{b \in \mathcal{B}} V_b(t) \end{aligned}$$

Furthermore, the bubble indices of all fluid or interface cells that are included in $\mathcal{B}$ have to be set to $\min(\mathcal{B})$.

2.5 Calculating the Surface Curvature

The method to calculate the surface curvature for a certain cell described in [Thi05] requires information from all nearest and next-nearest neighbor cells. For the 3D case, this corresponds to a neighborhood of $5 \times 5 \times 5$ cells. Since the efficient parallelization of the algorithm is a major goal of this thesis, a different method has been developed and implemented which splits the calculation of the surface curvature into three parts:

In the first step, the surface normal is calculated for each interface cell. This step only requires information from the nearest neighbor cells. Second, a surface point is reconstructed, based on the local surface normal and fill fraction. Finally, the surface curvature is calculated from the surface points of the local and the nearest neighbor cells.

If all three steps are combined, the calculation of the surface curvature for a certain interface cell also requires the information of a $5 \times 5 \times 5$ neighborhood, but since these steps are divided, intermediate communication makes it possible to use a single cell layer for communication.

The three steps will be discussed in detail in the following sections. For a simpler notification, we now switch from the continuum notation for variables, e.g., $\varphi(\boldsymbol{x})$, to a notation with discrete indices, e.g., $\varphi_{x,y,z}$.

2.5.1 Calculating the Surface Normals

In the first step, an approximate surface normal is calculated by treating the fluid fraction φ of each cell as a scalar field and calculating its gradient as described in [PY92, PS92]. Since the fluid fractions are clipped to an interval of $[-\epsilon_{\text{vol}}, 1+\epsilon_{\text{vol}}]$ the resulting normal vector must be considered as a rough approximation for the real normal vector. This so-called Parker-Youngs (PY) approximation calculates the normal vector $\boldsymbol{n}_{x,y,z}$ as

$$\begin{aligned} \boldsymbol{n}'_{x,y,z} &= \sum_{dx,dy,dz \,\in\, [-1,0,+1]} -\omega_{dx,dy,dz}\ \varphi_{x+dx,y+dy,z+dz}\ \langle dx, dy, dz \rangle \\ \boldsymbol{n}_{x,y,z} &= \frac{\boldsymbol{n}'_{x,y,z}}{\left\| \boldsymbol{n}'_{x,y,z} \right\|} \end{aligned}$$

with

$$\omega_{dx,dy,dz} = \begin{cases} 1 & \text{for} \quad \|\langle dx, dy, dz \rangle\| = 1 \\ 2 & \text{for} \quad \|\langle dx, dy, dz \rangle\| = \sqrt{2} \\ 4 & \text{for} \quad \|\langle dx, dy, dz \rangle\| = \sqrt{3} \end{cases} .$$

This normal vector is defined to point away from the fluid towards the gas phase.

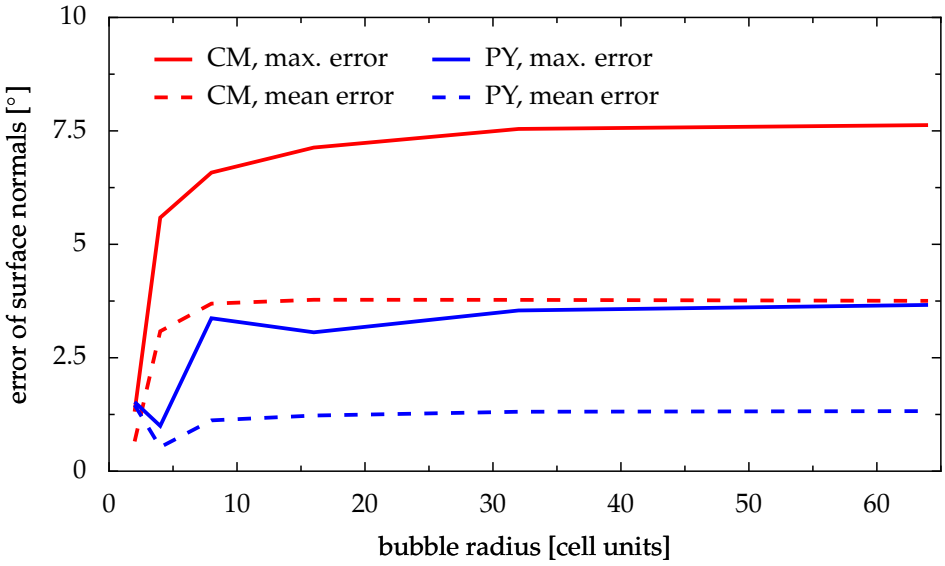


Figure 2.10: Comparison of the Center of Mass (CM) method and the Parker-Youngs (PY) approximation: The graph shows the maximum and mean error of the calculated surface normals for different radii of a spherical bubble.

Another method for approximating surface normals which has been devised in the context of volume of fluid methods is the so-called Center of Mass (CM) method by Pilliod et. al [PP04]. It is a simpler variant of the PY approximation, since the coefficients $\omega_{dx,dy,dz}$ are all equal to 1. Figure 2.10 show a comparison of both algorithms with respect to the maximum and mean error of the calculated surface normals for different radii for a spherical bubble. Obviously, the PY approximation is superior to the simpler CM approximation. Due to these results, the PY approximation was used in this thesis.

2.5.2 Calculating the Surface Points

The next step towards the calculation of the surface curvature is the reconstruction of a surface point $\boldsymbol{p}_{x,y,z}$ for each interface cell. For simplification, the small part of the surface fraction passing through a cell is assumed to be planar. This hypothetical surface plane $\mathcal{A}$ must fulfill two requirements:

1. The plane's surface normal must be equal to $\boldsymbol{n}_{x,y,z}$.
2. The plane bisects the cell into two volumes, corresponding to the fluid and gas fraction of the cell. This fluid volume $V_{x,y,z}$ must be equal to the known fluid fraction $\varphi_{x,y,z}$.

In the following explanations, we always refer to an individual interface cell and thus neglect the index x, y, z. The center of this representative cell is located at $\langle 0.5, 0.5, 0.5 \rangle$; the size of the cell is $\langle 1, 1, 1 \rangle$.

Algorithm 2.1 Calculate the plane offset.

procedure CALCULATEPLANEOFFSET($\boldsymbol{n}, \varphi$)
  if $\varphi > \frac{1}{2}$ **then** ▷ exploit the symmetry of the problem
    $\varphi' \leftarrow 1 - \varphi$ ▷ restrict φ to the interval $[0, \frac{1}{2}]$
  else
    $\varphi' \leftarrow \varphi$
  end if

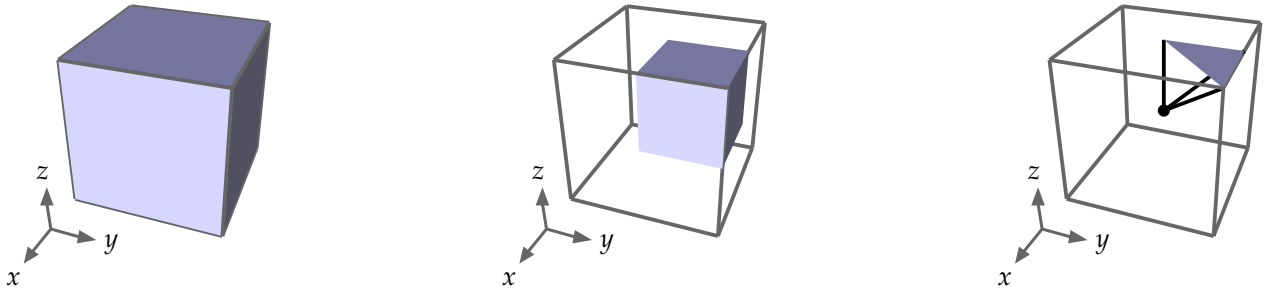
  $a_{\min} \leftarrow -\sqrt{3/4}$ ▷ the lowest possible value
  $a_{\max} \leftarrow 0$
  for $i \leftarrow 1, N_{\text{bisection}}$ **do**
    $a \leftarrow \frac{1}{2}(a_{\min} + a_{\max})$
    $V \leftarrow$ CALCULATEFLUIDVOLUME($\boldsymbol{n}, a$)
    if $V > \varphi'$ **then**
      $a_{\max} \leftarrow a$ ▷ volume too large, so lower upper bound to mean offset a
    else
      $a_{\min} \leftarrow a$ ▷ volume too small, so raise lower bound to mean offset a
    end if
  end for
  $a \leftarrow \frac{1}{2}(a_{\min} + a_{\max})$

  if $\varphi > \frac{1}{2}$ **then**
    return $-a$
  else
    return a
  end if
end procedure

To characterize the position of the plane $\mathcal{A}$, we introduce an offset parameter a such that

$$\boldsymbol{p} = \langle 0.5, 0.5, 0.5 \rangle + a\,\boldsymbol{n} \quad ; \quad \boldsymbol{p} \in \mathcal{A} \quad . \tag{2.14}$$

The direct calculation of the offset a is difficult, because a closed-form equation for $a(\boldsymbol{n}, V)$ cannot be given. Similar to an approach described in [RK98], the inverse problem $V(\boldsymbol{n}, a)$ is solved instead using a bisection algorithm as described in Alg. 2.1. This iterative algorithm makes use of a procedure CALCULATEFLUIDVOLUME which is intricate to implement for 3D. In the simpler 2D case, a closed-form equation for $V(\boldsymbol{n}, a)$ can be found in [SZ99].



(a) no restriction

(b) $n_x, n_y, n_z \geq 0$

(c) $n_x, n_y, n_z \geq 0$ and $n_x \leq n_y \leq n_z$

Figure 2.11: The colored surface in each cube represents the area, where a normal vector $\boldsymbol{n} = \langle n_x, n_y, n_z \rangle$ might pierce the surface of the cell for three different restrictions with respect to $\boldsymbol{n}$.

The CALCULATEPLANEOFFSET procedure returns the offset value a which is required in Eq. 2.14 to calculate the surface point $\boldsymbol{p}$. Under the assumption of a correct normal vector $\boldsymbol{n}$, the maximum error $e_{\max}(\boldsymbol{p})$ of $\boldsymbol{p}$ depends on the number of bisections (see Alg. 2.1):

$$e_{\max}(\boldsymbol{p}) = \frac{\sqrt{3}}{2^{2+N_{\text{bisection}}}} \quad .$$

All results presented in this thesis have been obtained with $N_{\text{bisection}}$ set to 10 which corresponds to $e_{\max}(\boldsymbol{p}) = 0.0004$.

In the following section, the CALCULATEFLUIDVOLUME procedure will be described.

2.5.3 Calculating the Fluid Volume

Calculating the fluid volume for any given normal vector and any fluid fraction value is a tedious task, since many different geometric situations have to be taken into account. To simplify the calculation, the high degree of symmetry immanent to the cubic cell can be exploited by reducing the possible plane orientations to a minimal set of cases. In Fig. 2.11 the colored surface in each cube represents the area, where a normal vector $\boldsymbol{n} = \langle n_x, n_y, n_z \rangle$ originating from the center of the cube passes through a cube face for three different restrictions with respect to $\boldsymbol{n}$:

- Without any restriction, each point on the cell's surface can be reached (see Fig. 2.11a).
- If all three components of $\boldsymbol{n}$ are set to their absolute value ($n_x, n_y, n_z \geq 0$), the reachable area is reduced to $\frac{1}{8}$ of the previous area (see Fig. 2.11b).
- If the three components of $\boldsymbol{n}$ are also sorted ($n_x \leq n_y \leq n_z$), the resulting area is just $\frac{1}{48}$ (see Fig. 2.11c) compared to the non-restricted case.

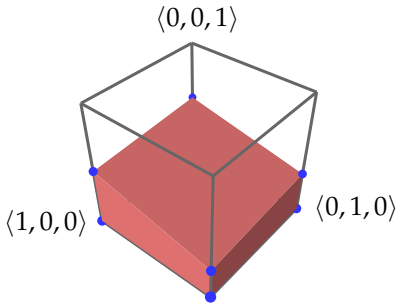


Figure 2.12: Special case for calculating the fluid volume where all four vertices $v_{0,0,0}$, $v_{1,0,0}$, $v_{0,1,0}$, and $v_{1,1,0}$ are inside the fluid domain.

The reduction of the reachable area is directly related to a reduction in cases that have to be considered in the CALCULATEFLUIDVOLUME procedure. It is essential to note that the mapping of the normal vector to this reduced set of cases does not influence the result of the volume calculation in the CALCULATEFLUIDVOLUME procedure.

For the following description of the CALCULATEFLUIDVOLUME procedure, we introduce two notations: $v_{x,y,z}$ with $x, y, z \in \{0, 1\}$ denotes one of the eight vertices of the cell at the position $\langle x, y, z \rangle$ (e.g., $v_{0,1,0}$). Furthermore, $e_{x,y,z}$ denotes one of the twelve edges of the cell. To specify an edge, two of the variables are replaced by actual coordinates (e.g., $e_{1,0,z}$ corresponds to the edge stretching from $\langle 1, 0, 0 \rangle$ to $\langle 1, 0, 1 \rangle$; $e_{0,y,0}$ corresponds to the edge stretching from $\langle 0, 0, 0 \rangle$ to $\langle 0, 1, 0 \rangle$). If an edge $e_{x,y,z}$ is cut by the interface plane, its value is defined as the value of the free coordinate (e.g., if the edge $e_{0,y,1}$ is cut exactly in the middle, its value is 0.5).

First, we consider a special case that is both simple to detect and to handle (see Fig. 2.12): If the four vertices $v_{0,0,0}$, $v_{1,0,0}$, $v_{0,1,0}$, and $v_{1,1,0}$ are included in the fluid domain[1], the volume V of the resulting frustum of a cube can be calculated as [BSM05]

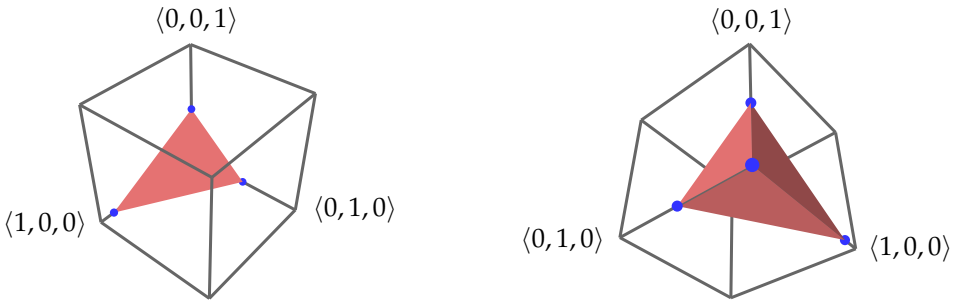
$$V = \frac{1}{4} \left(e_{0,0,z} + e_{1,0,z} + e_{0,1,z} + e_{1,1,z} \right) \quad .$$

In all other cases (i.e., $v_{1,1,0}$ is not included in the volume), we calculate the total fluid volume V by dividing it into at most four subvolumes, each with a pyramidal shape:

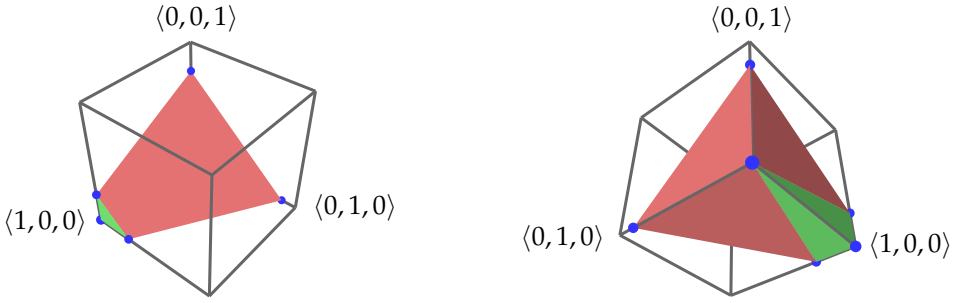
$$V = V_{0,0,0} + V_{1,0,0} + V_{0,1,0} + V_{0,0,1} \tag{2.15}$$

Figure 2.13 shows the different subvolumes with a constant normal vector and four different offset values a. The indices of the four subvolumes correspond to the vertex that has to be included for the subvolume to be considered. Because of the restricted normal vector, the vertex $v_{0,0,0}$ is always included in the volume and therefore the corresponding volume (red volume in Fig. 2.13) is always considered. Its base area $\mathcal{B}$ is made up of all

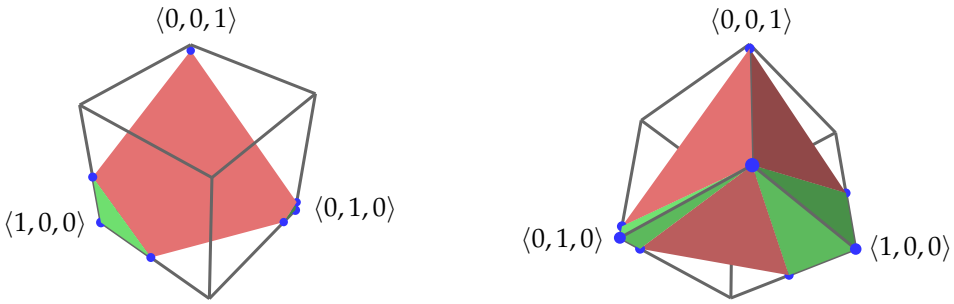
[1] Due to the limitations to the normal vector, the fact that the vertex $v_{1,1,0}$ is inside the volume implies that the other three mentioned vertices are also included.



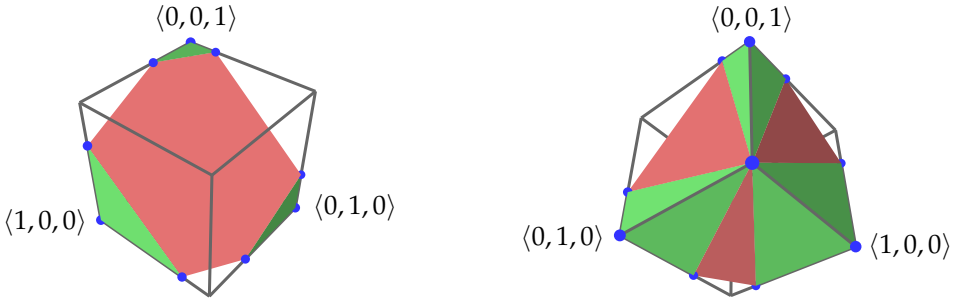
(a) $a = -0.52$



(b) $a = -0.30$



(c) $a = -0.17$



(d) $a = 0.00$

Figure 2.13: Fluid volume V with a constant normal vector $\boldsymbol{n} = \langle 0.37, 0.61, 0.70 \rangle$ and four different offset values a. For calculating the fluid volume, it is divided into at most four subvolumes (*red and green volumes*). The diagrams to the left and to the right show the same setup from opposing points of view.

points where the surface plane $\mathcal{A}$ cuts a cell edge. In the example in Fig. 2.13, the base plane $\mathcal{B}$ is delimited by three intersection points for $a = -0.52$, and six intersection points for $a = 0$. With the known area $\|\mathcal{B}\|$ of the base plane $\mathcal{B}$ and the height $\boldsymbol{n} \cdot \boldsymbol{p}$ of the pyramid, the volume $V_{0,0,0}$ is given by

$$V_{0,0,0} = \tfrac{1}{3} \|\mathcal{B}\| (\boldsymbol{n} \cdot \boldsymbol{p}) \quad . \tag{2.16}$$

The potential three other subvolumes (green volumes in Fig. 2.13) are calculated with the same formula for pyramidal shapes as

$$\begin{aligned} V_{1,0,0} &= \tfrac{1}{6}\, e_{1,y,0}\, e_{1,0,z} \\ V_{0,1,0} &= \tfrac{1}{6}\, e_{x,1,0}\, e_{0,1,z} \quad . \\ V_{0,0,1} &= \tfrac{1}{6}\, e_{x,0,1}\, e_{0,y,1} \end{aligned} \tag{2.17}$$

Finally, the total fluid volume V is gained by using Eq. 2.15 to sum up all subvolumes from Eqs. 2.16 and 2.17. This concludes the description of the CALCULATEFLUIDVOLUME procedure.

2.5.4 Calculating the Surface Curvature

Most algorithms found in literature to calculate the surface curvature (e.g., [LP82, LC87, MBF92, CS92]) require polyhedral or even triangulated surface meshes. The generation of such a mesh, based on the available surface points, would be time-consuming, since it requires global operations on the distributed data sets of the surface points in each time step. Instead, a variant of an algorithm described in [Tau95] is performed on a local triangulation to calculate the mean curvature $\bar{\kappa}_{x,y,z}$ at each surface point $\boldsymbol{p}_{x,y,z}$. This procedure consists of three parts:

1. selecting an appropriate subset of neighboring surface points
2. local triangulation of the chosen surface points
3. calculation of the surface curvature

2.5.4.1 Selecting a Subset of Neighboring Surface Points

Two aspects influence the choice of the surface points for constructing a local triangulation around a central surface point $\mathcal{P}_{x,y,z}$. On the one hand, only surface points from adjacent cells are considered to allow for optimal parallel processing. On the other hand, the resulting mesh should not contain degenerated triangles[2]. To consider both requirements, the

[2]The exact definition of a degenerated triangle depends on the context. Here, we mean long triangles with a small area.

adjacent surface point has to fulfill several restrictions.

$$\mathcal{P}^1_{x,y,z} = \Big\{ (x+dx, y+dy, z+dz) \;\Big|\; dx, dy, dz \in \{-1, 0, +1\} \;\wedge \tag{2.18}$$

$$\mathcal{C}_{x+dx,y+dy,z+dz} \in \mathcal{C}^{\mathrm{I}} \;\wedge \tag{2.19}$$

$$0.8 \leq \left\| \boldsymbol{p}_{x,y,z} - \boldsymbol{p}_{x+dx,y+dy,z+dz} \right\| \leq 1.8 \Big\} \tag{2.20}$$

Equation 2.18 only allows surface points from neighboring cells, while Eq. 2.19 restricts the selection to interface cells. Equation 2.20 excludes interface cells with surface points that are too close or too far away from the central surface point $\boldsymbol{p}_{x,y,z}$. The optimal distance limits have been found heuristically and result in a reasonable number of considered interface cells.

It is still possible that two surface points in $\mathcal{P}^1_{x,y,z}$ lie in a similar direction relative to the central surface point which would result in degenerated triangles in the mesh. In set $\mathcal{P}^2_{x,y,z}$ this situation is avoided by removing the surface point that is closer to the central surface point:

$$\mathcal{P}^2_{x,y,z} = \Big\{ k \in \mathcal{P}^1_{x,y,z} \;\Big|\; \forall\, m \in \mathcal{P}^1_{x,y,z} \;:\; \measuredangle(\boldsymbol{p}_k - \boldsymbol{p}_{x,y,z}, \boldsymbol{p}_m - \boldsymbol{p}_{x,y,z}) > 30^\circ \;\vee \left\| \boldsymbol{p}_{x,y,z} - \boldsymbol{p}_k \right\| > \left\| \boldsymbol{p}_{x,y,z} - \boldsymbol{p}_m \right\| \Big\}$$

Again, the minimum angle of 30° between two neighboring surface points was chosen heuristically. Similar results are obtained when the other surface point (further away from the central surface point) is removed. Still, the former gives slightly more accurate results for the curvature approximation.

2.5.4.2 Local Triangulation of the Chosen Surface Points

As a first step, a mean normal $\bar{\boldsymbol{n}}_{x,y,z}$ is calculated, based on set $\mathcal{P}^2_{x,y,z}$:

$$\bar{\boldsymbol{n}}'_{x,y,z} = \boldsymbol{n}_{x,y,z} + \sum_{(x',y',z') \,\in\, \mathcal{P}^2_{x,y,z}} \boldsymbol{n}_{x',y',z'} \quad ; \quad \bar{\boldsymbol{n}}_{x,y,z} = \frac{\bar{\boldsymbol{n}}'_{x,y,z}}{\left\| \bar{\boldsymbol{n}}'_{x,y,z} \right\|}$$

The positions of the surface points in $\mathcal{P}^2_{x,y,z}$ are now projected onto a plane $\mathcal{D}$ which is perpendicular to the mean normal $\bar{\boldsymbol{n}}'_{x,y,z}$. Starting with an arbitrary neighboring surface point, the surface points are enumerated counter-clockwise, as seen against the direction of $\bar{\boldsymbol{n}}_{x,y,z}$ (see Fig. 2.14).

Now, all requirements are met to apply the algorithm described in Appendix A. The outcome of this algorithm is an approximation for the curvature $\kappa_{x,y,z}$.

2.5.5 Results for the Curvature Calculation

To test the behavior of the described procedure, we calculate the results for spherical bubbles with two different radii: 4 and 16 cell units (cu). Because of the symmetry of the

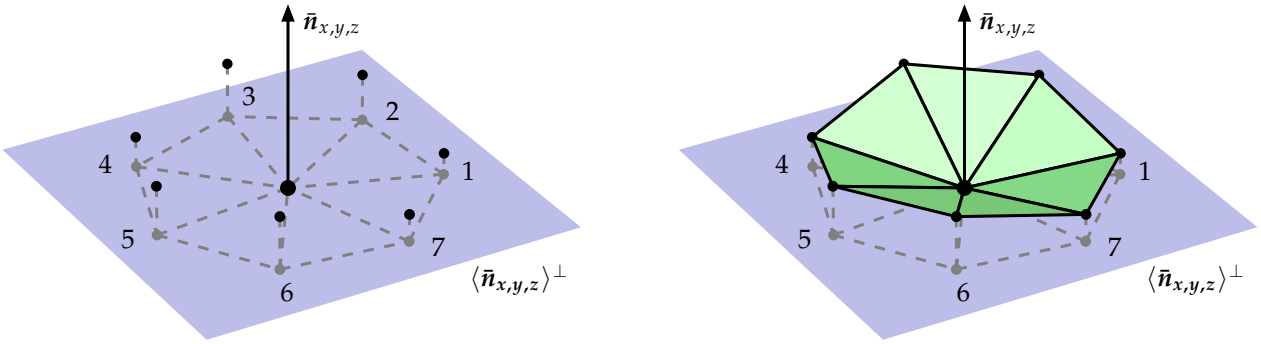


Figure 2.14: The neighboring surface points (*black dots*) are projected onto a plane $\langle \bar{n}_{x,y,z} \rangle^{\perp}$. The surface points are then enumerated according to their projected position on this plane. This results in an umbrella-shaped set of faces surrounding the center point.

bubble and the applied algorithm, the Figures 2.15 and 2.16 only display $\frac{1}{8}$ of the entire bubble surface in polar diagrams. We distinguish between the gradient normal $\boldsymbol{n}$ and the mesh normal $\boldsymbol{N}$ which is defined in Appendix A. For each radius, the five polar diagrams show

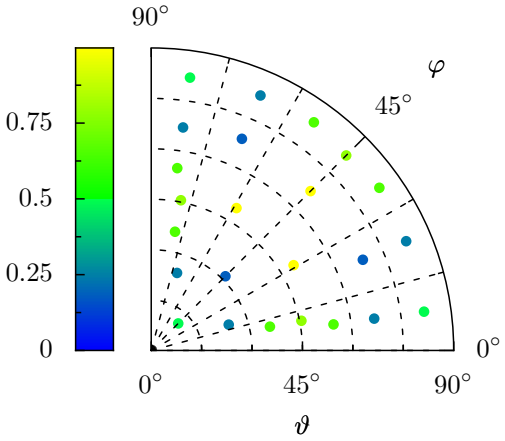
- the error of the gradient normal $e(\boldsymbol{n}) = \angle(\boldsymbol{n}_{\text{calc}}, \boldsymbol{n}_{\text{corr}})$,
- the distance from the center of the bubble to the surface point d,
- the error of the mesh normal $e(\boldsymbol{N}) = \angle(\boldsymbol{N}_{\text{calc}}, \boldsymbol{N}_{\text{corr}})$,
- the number of the neighboring vertices used for the local triangulation,
- the resulting surface curvature κ.

Table 2.2 summarizes these five values for four different spherical bubbles with the radii 4, 8, 12, and 16 cu.

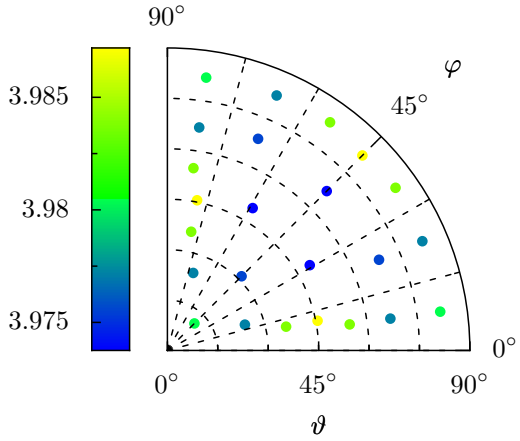
Except for the smallest bubble radius, the maximum error of the gradient normal $e(\boldsymbol{n})$ is not strongly influenced by the bubble radius with a value of about 3°. As can be seen in Figs. 2.15a and 2.16a, the error is lower for interface points close to $\vartheta = i \cdot 45°$ and $\varphi = j \cdot 45°$ ($i, j \in [0, 1, 2]$) which is an effect of the PY approximation.

The accuracy of the distance d of the interface points from the center of the bubble is increasing with the radius of the bubble. Since the position of the interface points is calculated from the gradient normal $\boldsymbol{n}$ and the fluid fraction φ of a cell, the error pattern in Figs. 2.15b and 2.16b is similar to the pattern of the gradient normal error.

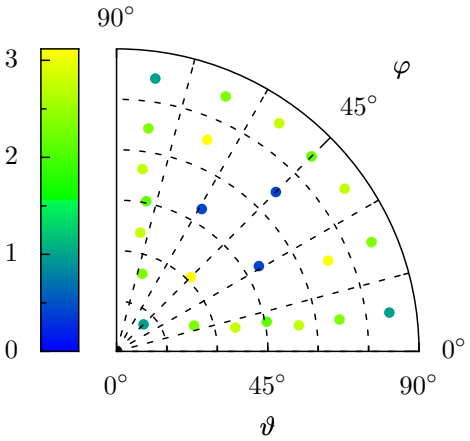
The number of neighboring interface points which are used for the local triangulation lies between 7 and 12 neighbors for all bubble radii with an average of about 8 neighbors.



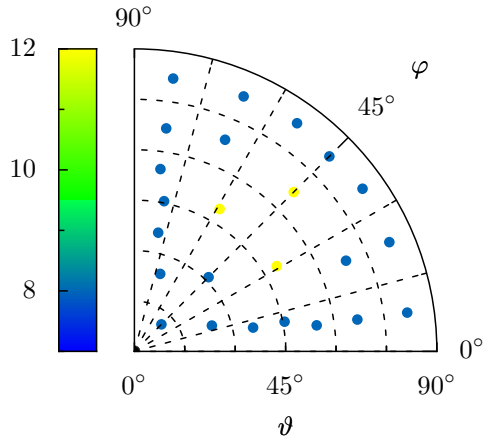
(a) Error of the gradient normal $e(\boldsymbol{n})$[°]



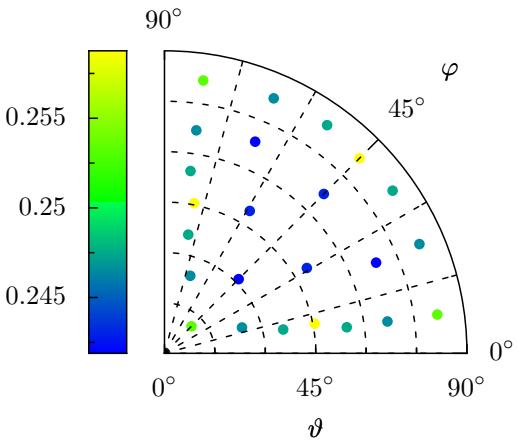
(b) Distance from the center of the bubble to the surface point d [cu]



(c) Error of the mesh normal $e(\boldsymbol{N})$[°]

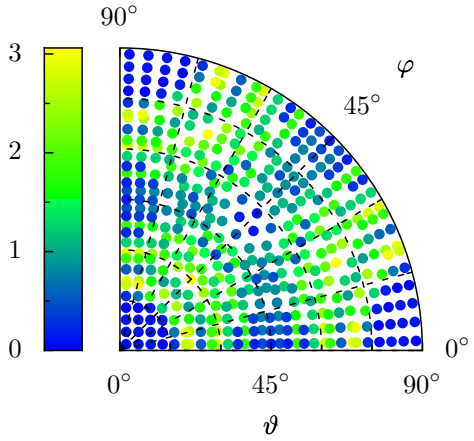


(d) Number of neighboring vertices used for the local triangulation

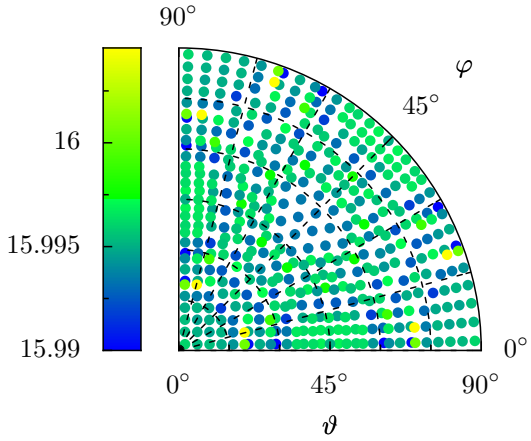


(e) Surface curvature κ [1/cu]

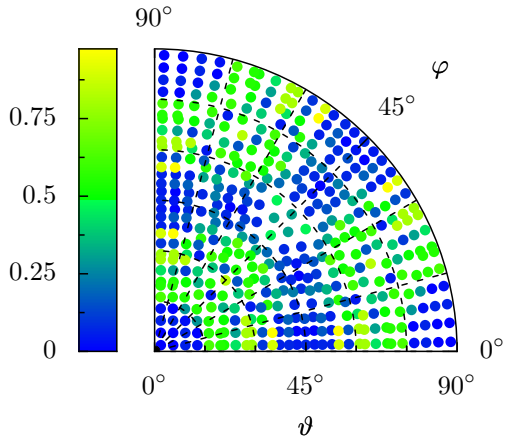
Figure 2.15: Polar plots of characteristic values for the calculation of the surface curvature. Each dot represents an interface point for a spherical bubble with radius 4 cu which results in a curvature of 0.250 $\frac{1}{\text{cu}}$.



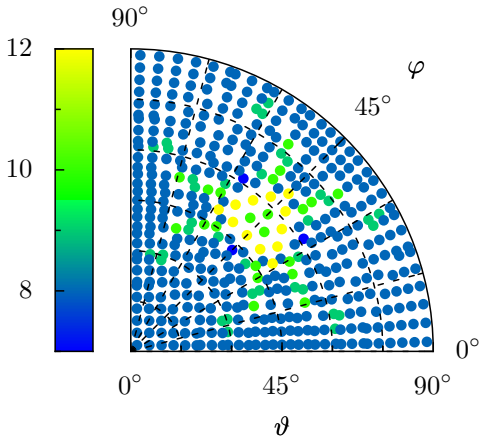
(a) Error of the gradient normal $e(\boldsymbol{n})[^\circ]$



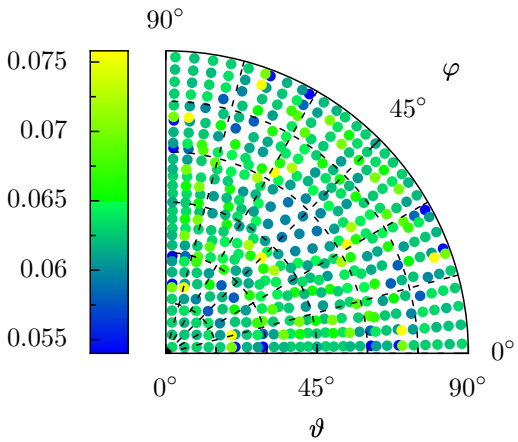
(b) Distance from the center of the bubble to the surface point d [cu]



(c) Error of the mesh normal $e(\boldsymbol{N})[^\circ]$



(d) Number of neighboring vertices used for the local triangulation



(e) Surface curvature κ [1/cu]

Figure 2.16: Polar plots of characteristic values for the calculation of the surface curvature. Each dot represents an interface point for a spherical bubble with radius 16 cu which results in a curvature of approx. 0.063 $\frac{1}{\text{cu}}$.

	min. value	max. value	mean value	std. deviation
bubble radius: 4 cu				
$e(\boldsymbol{n}_\mathrm{g})[^\circ]$	$1.81 \cdot 10^{-1}$	$9.98 \cdot 10^{-1}$	$5.30 \cdot 10^{-1}$	—
$e(\boldsymbol{n}_\mathrm{m})[^\circ]$	$4.16 \cdot 10^{-1}$	3.12	2.10	—
d [cu]	3.97	3.99	3.98	$4.51 \cdot 10^{-3}$
# neighbors	8	12	8.50	—
κ [1/cu]	$2.42 \cdot 10^{-1}$	$2.59 \cdot 10^{-1}$	$2.48 \cdot 10^{-1}$	$5.26 \cdot 10^{-3}$
bubble radius: 8 cu				
$e(\boldsymbol{n}_\mathrm{g})[^\circ]$	$5.53 \cdot 10^{-2}$	3.37	1.12	—
$e(\boldsymbol{n}_\mathrm{m})[^\circ]$	0.00	1.66	$7.44 \cdot 10^{-1}$	—
d [cu]	7.99	8.00	7.99	$3.23 \cdot 10^{-3}$
# neighbors	7	11	8.24	—
κ [1/cu]	$1.18 \cdot 10^{-1}$	$1.37 \cdot 10^{-1}$	$1.25 \cdot 10^{-1}$	$4.74 \cdot 10^{-3}$
bubble radius: 12 cu				
$e(\boldsymbol{n}_\mathrm{g})[^\circ]$	$8.58 \cdot 10^{-2}$	3.45	1.21	—
$e(\boldsymbol{n}_\mathrm{m})[^\circ]$	0.00	1.34	$4.52 \cdot 10^{-1}$	—
d [cu]	$1.20 \cdot 10^{1}$	$1.20 \cdot 10^{1}$	$1.20 \cdot 10^{1}$	$1.94 \cdot 10^{-3}$
# neighbors	8	12	8.28	—
κ [1/cu]	$7.84 \cdot 10^{-2}$	$9.36 \cdot 10^{-2}$	$8.39 \cdot 10^{-2}$	$2.72 \cdot 10^{-3}$
bubble radius: 16 cu				
$e(\boldsymbol{n}_\mathrm{g})[^\circ]$	0.00	3.06	1.23	—
$e(\boldsymbol{n}_\mathrm{m})[^\circ]$	0.00	$9.75 \cdot 10^{-1}$	$3.29 \cdot 10^{-1}$	—
d [cu]	$1.60 \cdot 10^{1}$	$1.60 \cdot 10^{1}$	$1.60 \cdot 10^{1}$	$2.11 \cdot 10^{-3}$
# neighbors	7	12	8.28	—
κ [1/cu]	$5.40 \cdot 10^{-2}$	$7.58 \cdot 10^{-2}$	$6.33 \cdot 10^{-2}$	$3.51 \cdot 10^{-3}$

Table 2.2: Characteristic values for the calculation of the surface curvature for four spheres with different radii.

In contrast to the gradient normal error, the mesh normal error $e(\boldsymbol{N})$ strongly depends on the bubble radius. With increasing bubble radius, both the maximum and the mean mesh normal error decrease reciprocally. For the bubble radius of 16 cu, the mean (maximum) mesh normal error is only 27% (31%) of the mean (maximum) grid normal error. The reason for this difference in the accuracy of the two normal values is the higher accuracy of the positions of the interface points which are used to calculated the grid normal $\boldsymbol{N}$.

The error of the resulting surface curvature κ, however, is not decreasing reciprocally with increasing bubble radius. The standard deviation for κ is almost constant, i.e., for

small curvatures the relative errors are comparably high. A small curvature results in a low surface tension, which means that the influence of the error will only slightly disturb the simulation results. The spatial pattern of the curvature shown in Figs. 2.15e and 2.16e is closely related to the error pattern of the gradient normal error. As expected, the deviation from the correct curvature is biggest, where areas with a small mesh normal error (blue dots in Figs. 2.15c and 2.16c) border an area with a high mesh normal error (yellow dots in Figs. 2.15c and 2.16c).

2.6 Restrictions

Just like any other model, the model described in this thesis has restrictions which are not obvious from the model description. This section describes the two most important limitations.

2.6.1 Bubble Segmentation

As described in Sec. 2.4.4.2, the coalescence of two bubbles is handled correctly. The opposite effect, i.e., the segmentation of one bubble to several separate bubbles is more difficult to address algorithmically. The detection of a separation would involve the topological scanning of all bubble volumes similar to a graphical filling algorithm. From an algorithmic point of view this is a feasible approach even for partitioned domains in the context of parallel computing, but the required time for this procedure would be unacceptably high, since it needs to be performed in each time step.

Because many interesting applications can be simulated which do not depend on the feature of bubble segmentation, this issue was not addressed in this thesis.

2.6.2 Maximum Fluid Velocity

The equivalence of the Boltzmann equations and the Navier-Stokes equations can be shown using a Chapman-Enskog analysis (for details see [WG00]). From this analysis, the restrictions arises that the equivalence is only given for small velocities of the fluid compared to the sound velocity c_s which is $\frac{1}{3}$ for the D3Q19 model. With the dimensionless variables, the restriction reads as

$$\|\boldsymbol{v}\| \ll c_s^2 = \frac{1}{9} \quad .$$

Depending on the simulation setup, higher velocities can be reached. Although this does not necessarily lead to unphysical behavior, the simulation setup should be chosen to avoid this situation.

To circumvent this problem one can simply increase the spatial resolution which inherently increases the time resolution. This lowers the fluid velocities for a physically equivalent simulation setup. Another way to prevent this conditions was proposed by Thürey et al. in [TPR+06]. This method uses adaptive parameterization to decrease the numerical velocity while preserving the physical interpretation.

Chapter 3

Implementation

In 2002, the implementation of the previously described model started. The code will be called FSLBM in the following, which stands for free surface LBM. At this time, the major target architecture for FSLBM was the former German high performance computing (HPC) system Hitachi SR8000-F1 with an overall performance of 2 TFlop/s (see Sec. 4.1.2). This strongly influenced the choice of the programming language, since the C++ compiler available for this system did not support some important features for automatic parallelization at this time (see Sec. 4.1.2). For that reason, the programming language C was chosen as a compromise between a modern programming language and achievable application performance.

3.1 Overview

In the following overview, the term module (or code module) will be used for an entity of code which encapsulates a specific functionality similar to a class in C++ but without the strict definition as in the programming language Modula.

Figure 3.1 depicts the most important code modules and their interactions. For each module, the major tasks are listed below.

Simulation Steering: This module contains the parsing and checking of the command line arguments. It triggers the initialization of the other modules, because the mechanism of constructors as known from C++ is not available in C. Apart from that, it contains the central time loop.

Numerics Module: This module encloses all numerical routines for handling the fluid and its surface. In particular the calculation of the surface normals, the surface points, the surface curvature, the lattice Boltzmann method, and the cell conversions are handled here.

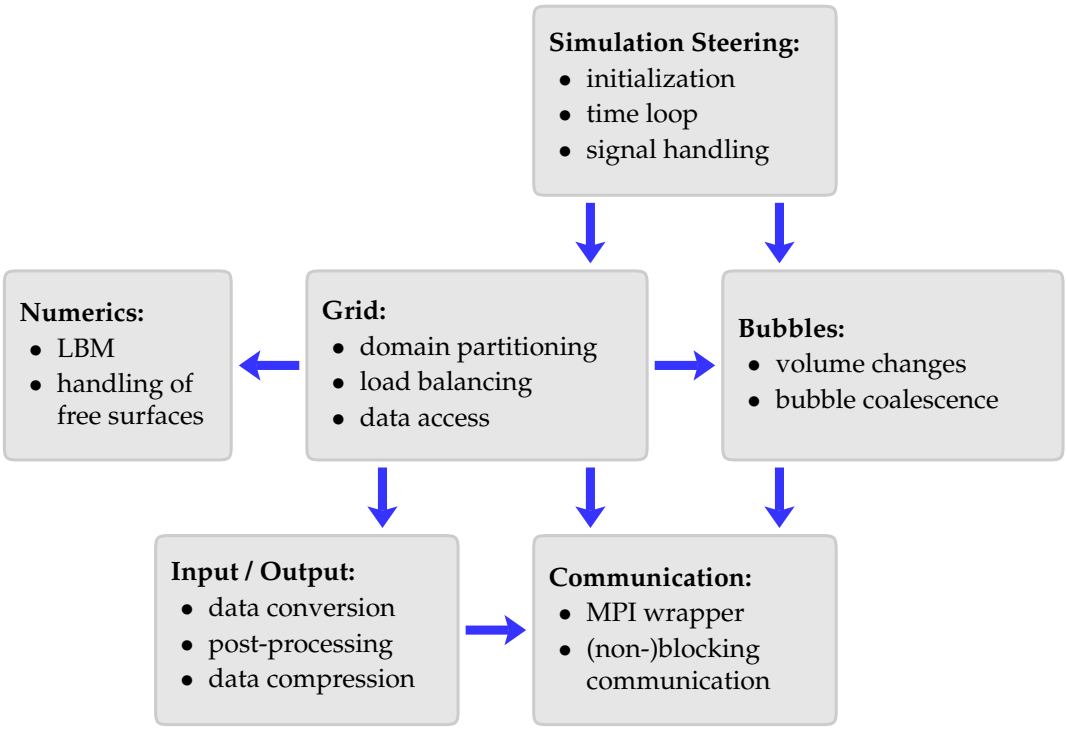


Figure 3.1: Schematic view of the most important code modules and their interactions. Apart from these, some auxiliary modules exist, e.g., for performance measurements or the initialization of the fluid and obstacle geometries.

Grid Module: This module plays the most central role in the simulation code. Among its tasks are the domain partitioning, triggering of communication with neighboring domains, the detection and leveling of load imbalances, thread handling, and the regulation of the access to the cell data.

Bubble Module: For each bubble, this module stores its initial and current volume. Furthermore, the volume changes are gathered and synchronized in each time step among the participating grid domains.

Input/Output Module: Several reasons make it necessary to pre- and post-process data, e.g., (de)compression of data, little/big-endian issues, or for visualization.

Communication Module: The main intention of this module is to hide the specifics of the underlying communication library; in this case the Message Passing Interface[1] (MPI) library. If a different communication library was to be used, it would only influence the implementation of the communication module.

Some of the implementation details described here will be revisited in Chapter 4 regarding the aspects of high performance computing.

[1] See `http://www.mpi-forum.org/` for official specifications and more information.

cell types	fluid	interface	gas	obstacle	data type
flags	•	•	•	•	`long int`
distribution functions f_i	•	•	-	-	`double[19]`
bubble index b	-	•	•	-	`long int`
fluid fraction φ	-	•	-	•	`double`
mass m	-	•	-	-	`double`
surface point $\boldsymbol{p}$	-	•	-	-	`double[3]`
surface normal $\boldsymbol{n}$	-	•	-	-	`double[3]`

Table 3.1: Each cell type only uses a subset of the available cell data. A bullet (•) means that the data is used and contains valid data; a dash (-) means that the data is unused. The memory space for each piece of data is allocated regardless whether the data is used or not. The corresponding C data type is given in the right column.

3.2 Model Data

The required data for the described model can be separated into two major parts: the state variables for the set of cells in the grid module and the state variables of the bubbles in the bubble module. The basic handling of both data will be described in the following two sections.

3.2.1 Cell Data

Each cell features a set of state variables depending on the current cell type as listed in Tab. 2.1. All listed state variables are stored for each cell type, although some state variables are not required for certain cell types. If only the required state variables were stored for each cell type, the conversion of a cell type would involve the re-allocation of memory for this cell, since the required memory space would be different.

Table 3.1 lists all state variables and their corresponding C data type. Some of the state variables, e.g., the surface point $\boldsymbol{p}$ are arrays consisting of several entities of the same data type. The flag state variable uses individual bits to store several different pieces of status information, e.g., the cell type or the conversion to a different cell type. The two used data types `long int` and `double` both use eight bytes of memory. Since 29 of these data types are required for an entire cell, the overall memory size amounts to 232 bytes (B) for each cell.

Some of the methods operating on the state variables require the data for two time steps, e.g., the streaming operation of the LBM. Therefore, two sets of state variables have to be stored for all cells. This doubles the memory requirements. Taking this into account, about 2.3 million cells can be stored in one GB of memory. This corresponds to a cubical

computational domain of about 132 cells. From these considerations it becomes obvious that the described model and its implementation consume large amounts of memory even for comparably small computational domains.

Many implementations of the LBM store the cell data in an array structure with the data belonging to one cell arranged contiguously in memory. This is canonical for a plain LB method, since all data is of the same type — `double` for the C language. With the free surface extension, however, different data types have to be stored for one cell. To achieve this, the `union` construct can be used as shown in Alg. 3.1. It allows to address the same data in memory as different data types; in our case as a `double` or `long int`. Two arrays of this `Entry` type are allocated to store all state variables of a cell as entries for the entire computational domain of the size `SIZE_X` $\times$ `SIZE_Y` $\times$ `SIZE_Z`. Compared to an array of structures, the `union` construct allows for an easy reordering of the cell data entries, since all entries have the same type `Entry`.

Algorithm 3.1 Using the `union` construct to unify data access.

```
typedef union {
  double d;
  long int i;
} Entry;

Entry sourceCellData[SIZE_Z][SIZE_Y][SIZE_X][29],
      destCellData  [SIZE_Z][SIZE_Y][SIZE_X][29];
```

Since the elements addressed with the right-most array index are located contiguously in memory for C, all data for one cell is grouped. The next indices correspond to the x, y, and z dimension of the computational domain. In Sec. 4.4 other possible memory layouts and their impact on computational performance will be discussed.

3.2.2 Gas Bubble Data

As described in Sec. 2.3, each gas bubble b is described by two state variables: its initial volume V_b^* and current volume V_b. Internally, two more variables are stored for each bubble: the volume change ΔV_b in the current time step and the coalescence index M_b.

During the processing of a time step, the bubble volume V_b should be constant to ensure consistent results. Therefore, the volume changes for a bubble are kept separately in ΔV_b until the time step is completely calculated. Then the bubble volumes are update according to the volume change:

$$V_b(t+1) = V_b(t) + \Delta V_b \quad . \tag{3.1}$$

The merge index M_b is used to store the potential coalescence of the bubble b with another bubble as described in Sec. 2.4.4.2. Before each time step, all coalescence indices M_b are set to an invalid value. If coalescence occurs of bubble b_i with bubble b_j, the coalescence index of the bubble with the higher index is checked. If it is invalid, it is set to the lower of both indices:

$$M_{\max(b_i,b_j)} = \min(b_i, b_j) \quad .$$

If $M_{\max(b_i,b_j)}$ is already set to a valid value b_k because of a previous coalescence in the same time step, the coalescence indices of the two bubbles with the highest index are set to the lowest involved bubble index:

$$M_b = \min(b_i, b_j, b_k) \quad ; \quad b \in \{b_i, b_j, b_k\} \setminus \min(b_i, b_j, b_k)$$

This procedure ensures that the bubble data is always merged in the bubble with the lower bubble index which keeps the array of bubble data compact. Furthermore, it allows for any possible coalescence scenarios.

All four variables are stored as an array which contains all bubble data. More details about the handling of this data structure in the context of parallel computing is given in Sec. 4.5.3.

3.3 Basic Sequential Program Flow

To simplify the description of the simulation code, we first concentrate on the sequential version without parallel computing. In Sec. 4.5.2.3, the aspects of the program flow concerning the parallelization will be discussed.

First, we define several expressions which will be used in the following sections:

Sweep: During a sweep, all cells in the grid are processed according to a given algorithm. An important requirement for a sweep is that the result must be independent of the ordering in which the cells are processed. This guarantees that intrinsic symmetries in the simulation setup are preserved. Furthermore, it allows for optimization techniques, such as, loop blocking (see Sec. 4.4.5) without influencing the simulation results.

Active Cell (AC): The AC is the cell which is currently being processed during a sweep. Each cell becomes the AC exactly once in the course of a complete sweep.

Active Cell Neighbor (ACN): The ACNs are the 26 adjacent cells and the AC itself in a $3 \times 3 \times 3$ environment. Compared to the neighborhood $\mathcal{N}_{x,y,z}$ as depicted in Fig. 2.3, the ACNs also include the eight cells at the corners.

To ensure the requirement for a sweep that simulation result must not depend on the order in which the cells become the AC, two requirements concerning the data access are postulated:

1. If a certain type of state variable (e.g., the fluid fraction) is read from one of the two available grids (one for time step t and one for $t + 1$) during a sweep, writing the same type of state variable to the same grid is forbidden in the same sweep.

2. State variables may only be read from ACNs and may only be altered for the AC.

The first requirement alone would be sufficient to ensure the requirement for a sweep. The second requirement restricts the location of data access. Its use will become apparent in Sec. 4.5 about parallel computing.

These two requirements make it necessary to break up the algorithms employed for the described model into several sweeps. In total, six sweeps are used:

1. The surface normals are calculated for all interface cells, based on their fluid fractions.

2. The stream and collide operation are performed and the missing distribution functions are reconstructed. This also involves the calculation of the surface curvature and the mass transfer between the cells.

3. The fluid advection might have triggered interface cell conversions to gas or fluid cells. These conversions are prepared in this sweep while resolving conflicting cell conversions.

4. This sweep ensures that the layer of interface cells is still intact by converting gas or fluid cells to interface cells.

5. Cell conversions make it necessary to distribute missing or excess mass to neighboring interface cells. Therefore, the number of available interface cells has to be calculated.

6. All scheduled cell conversions are finally performed including the distribution of missing or excess mass. The volume changes for the gas bubbles are calculated.

After the complete set of sweeps has been applied, the bubble data is updated as described in Sec. 3.2.2. The simple program flow is outlined in Alg. 3.2.

3.4 Validation Experiments

As already stated, the main focus of this thesis is on the high performance aspects of a complex simulation code. Nevertheless, it is important and instructive to evaluate the model qualitatively and quantitatively with respect to its physical behavior and correctness.

Algorithm 3.2 Basic sequential program flow: For each time step, a number of sweeps has to be performed.

```
1: initialize
2: for t ← t_start, t_end do
3:     for s ← 1, N_sweep do
4:         perform sweep s
5:     end for
6:     update bubble data
7: end for
8: finalize
```

3.4.1 Qualitative Validation

For a qualitative validation, three typical simulation setups will be presented below. Depending on the topic at hand, the more appropriate one will be used for demonstration in the following sections.

3.4.1.1 Breaking Dam

The setup of a breaking dam represents the class of simulations with only a few individual gas bubbles (exactly one in this case). Only a quarter of the entire computational domain is filled with fluid. Furthermore, the fluid is changing its shape quickly. Figure 3.2 shows a sequence of ray-traced images for this setup. As expected, the block of fluid is pulled downwards by the gravitational force.

Initially, about 3% of the cells are obstacle cells, 24% are fluid cells, 0.5% are interface cells, and 73% are gas cells. In the course of the simulation, the share of interface cells increases, because the fluid surface grows significantly.

3.4.1.2 Set of Rising Bubbles

In contrast to the previous setup, the set of rising bubbles features many individual gas bubbles. The computational domain is almost completely filled and the overall movement of the fluid is low. Figure 3.3 displays a sequence of ray-traced images for this setup. Due to the buoyant force, the set of initially 161 spherical bubbles with different radii rises as expected. After 50000 time steps, 102 bubbles are left.

Although the total number of cells for this setup is almost equal to the setup described before, the overall runtime is much higher, since the domain is almost completely filled with fluid (95%) or interface cells (1%). The remainder of the domain is filled with 3% obstacle cells and 2% gas cells. These rates change only slightly during the simulation.

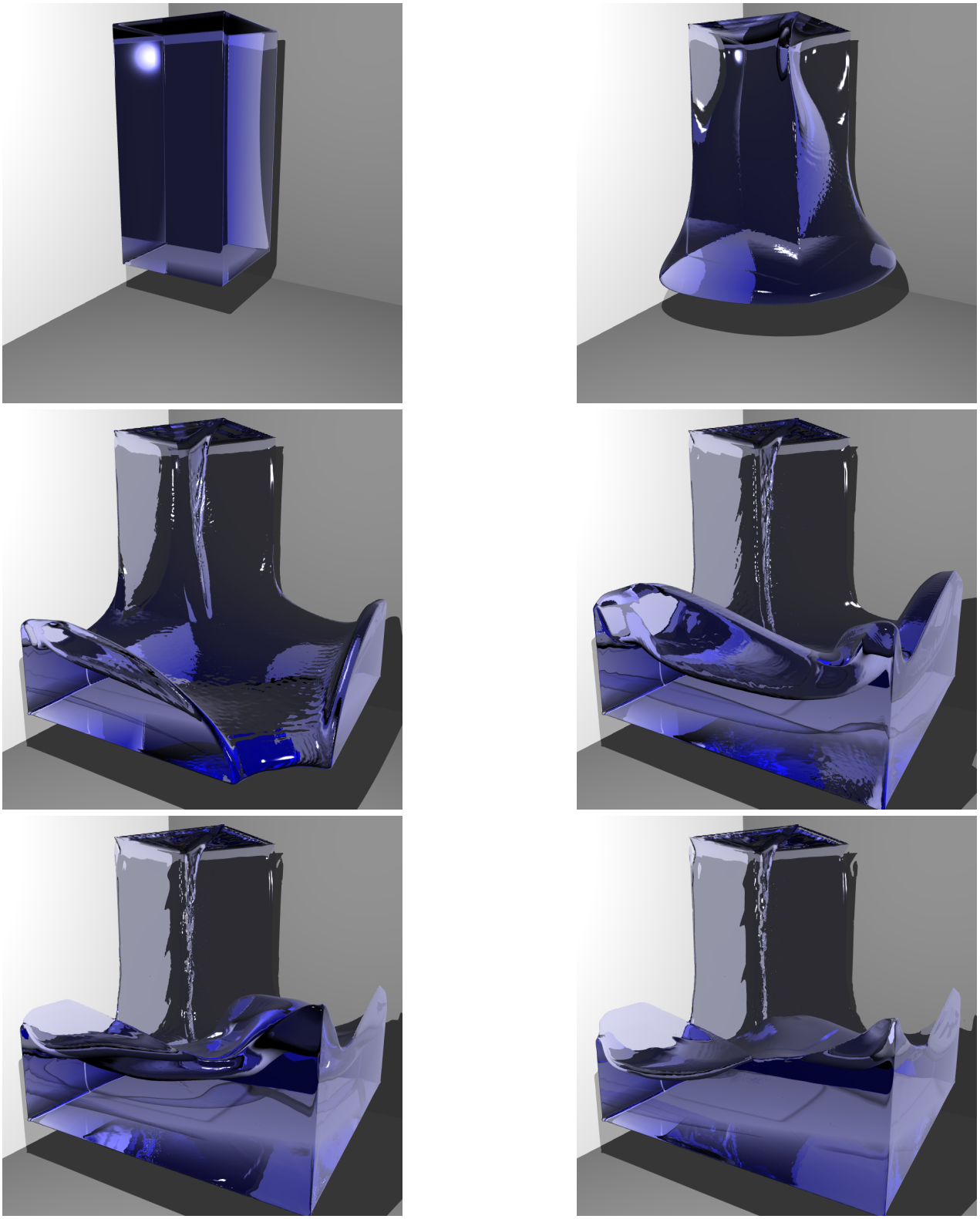


Figure 3.2: Simulation result of the breaking dam setup (domain size: 200^3; 50000 time steps)

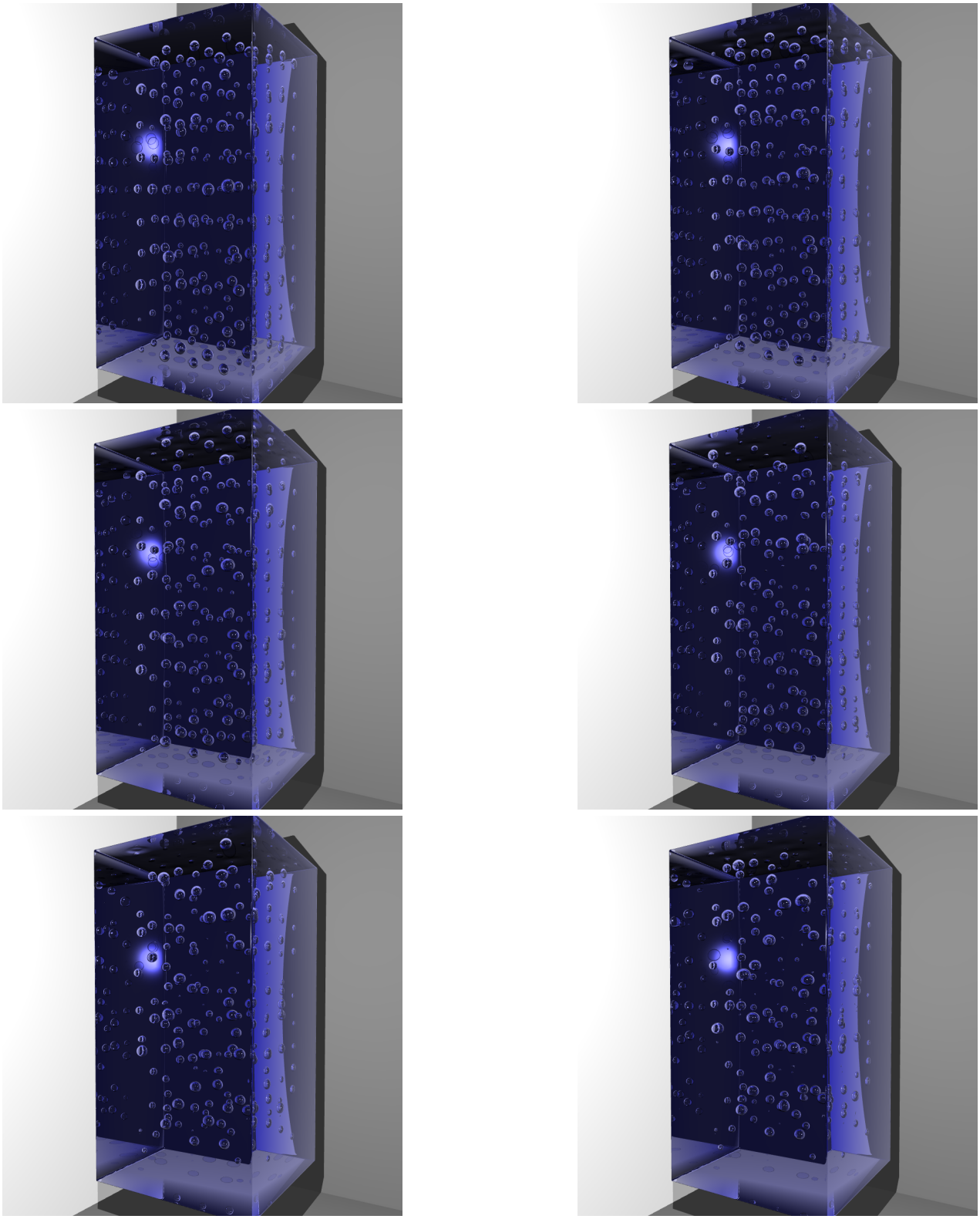


Figure 3.3: Simulation result of the set of rising bubbles (domain size: $160 \times 160 \times 312$; 50000 time steps)

3.4.1.3 Falling Drop

In the falling drop setup, a drop is falling into a pool of fluid as depicted in Fig. 3.4. The computational domain is divided into an upper half with a low count of fluid or interface cells and a lower half with a high number of these cell types. As we will discuss later, this imbalance of cell types deteriorates the performance for parallel computing.

3.4.2 Quantitative Validation: Single Rising Bubble

3.4.2.1 Description of the Setup

To quantitatively evaluate the model and its implementation, a simple simulation setup is used: a single spherical bubble rising in a tank of fluid as described in [HB76]. The exact geometric setup is shown in Fig. 3.5. Originally, the base of the tank has a circular base which leads to a cylindrical symmetry of the simulation setup. For simplicity reasons, the simulation setup used in this work has a quadratic base.

The walls of the tank are modelled with no-slip boundary conditions which is different from the reference simulation by Buwa et al. [BGD05]. To investigate the influence of the boundary conditions, the same simulation has been performed with a doubled tank diameter. The results show only insignificant differences. Thus, it can be concluded that the boundary conditions do not influence the final result of the simulation.

The bubble starts to rise in a quiescent fluid with a bubble pressure that is in equilibrium with the hydrostatic pressure of the surrounding fluid. The physical fluid properties of the employed mineral oil are:

- density $\rho = 875.5\ \mathrm{kg/m^3}$,
- dynamic viscosity $\mu = 0.118\ \mathrm{Pa \cdot s}$,
- surface tension $\sigma = 32.2 \cdot 10^{-3}\ \mathrm{N/m}$,
- at ambient (isothermal) conditions constant in the entire domain.

The dynamic viscosity μ and the kinetic viscosity ν, which is often used in the LBM context, only differ in a scaling with the density ρ of the fluid: $\nu = \frac{\mu}{\rho}$. The remaining physical parameter is the gravitational constant $g = 9.81\ \frac{\mathrm{m}}{\mathrm{s}^2}$ acting as an external force.

3.4.2.2 Parameterization

Based on these physical parameters, the following simulation parameters have to be chosen:

- spatial resolution Δx

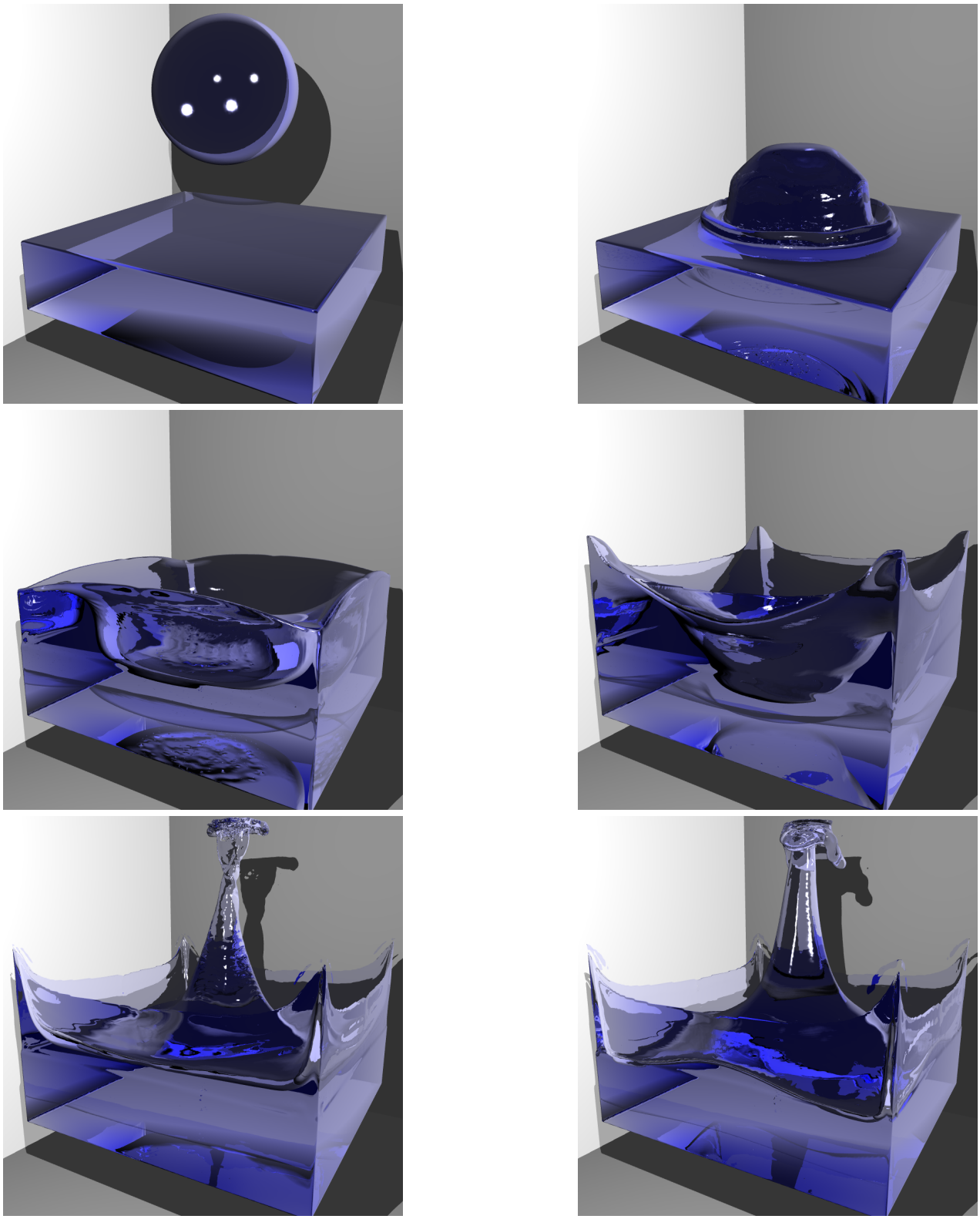


Figure 3.4: Simulation result of the falling drop (domain size: 200^3; 50000 time steps)

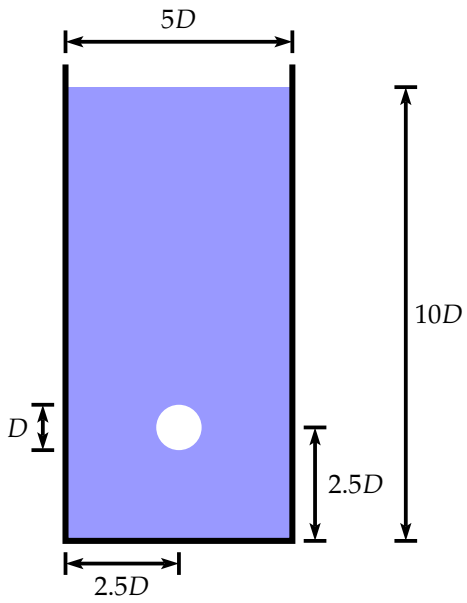


Figure 3.5: Geometrical setup of the rising bubble experiment. The parameter D is set to 1.215 cm.

- time step Δt
- relaxation time τ (see Eq. 2.3)
- surface tension σ_S (see Eq. 2.12)
- gravitational constant g_S (as external force $\boldsymbol{S}$ in Eqs. 2.8 and 2.9)

The spatial resolution Δx should be small enough to capture the geometrical details of the given setup. In our case Δx is set to 0.6075 mm. Thus, the bubble diameter is covered with 20 cells and the entire fluid domain has a size of $100 \times 100 \times 200$ cells.

The choice of the remaining simulation parameters is limited by several restrictions. Since the stability of the LB method is only given for a relaxation time τ higher than 0.5 [WG00], we choose a value of $\frac{1}{1.95} \approx 0.513$. By rewriting Eq. 2.7 for the kinetic viscosity using the simulation parameters, we get

$$\nu_S = \frac{1}{6}(2\tau - 1) \quad . \tag{3.2}$$

Considering the unit of the kinetic viscosity ($\frac{m^2}{s}$), its value in physical and grid units are related as

$$\frac{\nu}{\nu_S} = \frac{\Delta x^2}{\Delta t} \quad . \tag{3.3}$$

By combining Eqs. 3.2 and 3.3, the required time step Δt can be calculated in order to simulate a fluid with the prescribed kinetic viscosity ν:

$$\Delta t = \Delta x^2 \frac{\nu_S}{\nu} = \frac{\Delta x^2 (2\tau - 1)}{6\nu} \quad . \tag{3.4}$$

In most cases, one wants to simulate the behavior of a system setup for a certain duration in real-time T. Let N_t be the number of time steps which are required to cover the duration T: $N_t = \frac{T}{\Delta t}$. With the help of Eq. 3.4, we can predict the influence of the spatial resolution Δt on the number of required time steps N_t and therefore on the required computation time. If we halved Δx, the number of required time steps would increase by a factor of four. In combination with the increase of the number of cells by a factor of $2^3 = 8$, the computation time would be 32 times higher than for the original spatial resolution. It is therefore desirable to keep the spatial resolution as low as possible.

Similar to the previous simulation parameters, the gravitational constant can be calculated as

$$g_S = g\frac{\Delta t^2}{\Delta x} \quad .$$

The remaining parameter, the surface tension σ with the unit $\frac{\mathrm{N}}{\mathrm{m}} = \frac{\mathrm{kg}}{\mathrm{s}^2}$, requires some special considerations, because it also depends on the mass unit "kg". To rescale the mass unit, we consider the real weight m of a cube with the size Δx^3 which is filled with a fluid with the density ρ. The same volume in the simulation is the size of one cell with a weight defined to be equal to 1. We therefore need to rescale the weight unit "kg" with a factor of $\rho\Delta x^3$

$$\sigma_S = \sigma\frac{\Delta t^2}{\rho\Delta x^3} \quad .$$

For the interpretation of the final simulation results, the result data has to be rescaled to the real physical dimensions. For this purpose, the same equations in this section are used.

3.4.2.3 Results

Several other models and simulation codes have been validated with a similar simulation setup. One of them is a 2D simulation exploiting the cylindrical symmetry of the setup performed by Buwa et al. [BGD05]. Their simulation results have been used as reference results, because they closely resemble the original experimental results.

First, we compare the general movement of the bubble with the reference results in Fig. 3.6. As we can see, the velocities coincide well at the beginning. The peak in the rise velocity at $t = 0.025$ s is not as high as in the reference simulation. The terminal rise velocity which is reached at about $t = 0.2$ s, is equal to 0.20 $\frac{\mathrm{m}}{\mathrm{s}}$ for the LBM-based model. Compared to the value of the reference simulation and the experiment (0.215 $\frac{\mathrm{m}}{\mathrm{s}}$), it is about 7% lower.

To investigate this deviation, it is instructive to investigate the shape of the bubble at different time steps as depicted in Fig. 3.7. On the left, the shapes from the reference simulation are shown. Compared to the shapes of the LBM-based simulation on the right, it becomes obvious that the skirt formation, i.e., the concave deformation on the lower border

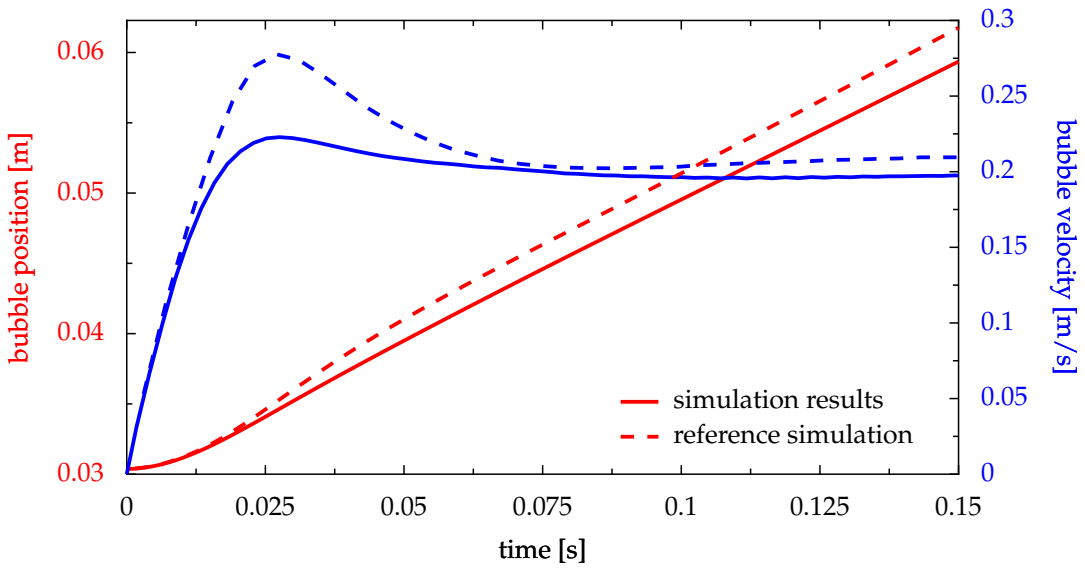


Figure 3.6: Comparison of the velocity and the position of the bubble with a reference simulation performed by Buwa et. al [BGD05].

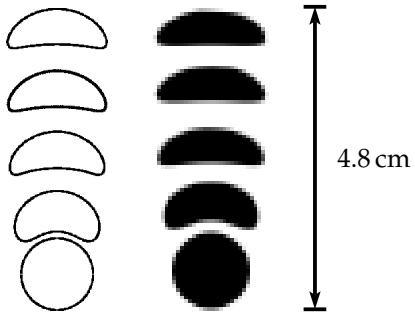


Figure 3.7: Simulated shape of the bubble after 0 s, 0.05 s, 0.1 s, 0.15 s, and 0.2 s. The diameter of the lower-most bubble is 1.215 cm.

of the bubble, is not as pronounced for the later time steps as in the reference simulation. The reason for this behavior is due to inaccuracies of the curvature calculation.

In Fig. 3.8 the streamlines in a central cut plane are visualized using the so-called line integral convolution (LIC) method. In contrast to the original description of the LIC method in [CL93], the length of the streamlines does not correspond to the velocity. Instead, the length is equal at all points to better visualize small velocities. The absolute value for the velocity is represented by different colors. The rising bubble induces a vortex left and right to the bubble. In 3D the vortex has a toroidal shape. As expected, the velocity of the fluid directly above and beneath the bubble is equal to the rise velocity of about 0.2 $\frac{\mathrm{m}}{\mathrm{s}}$.

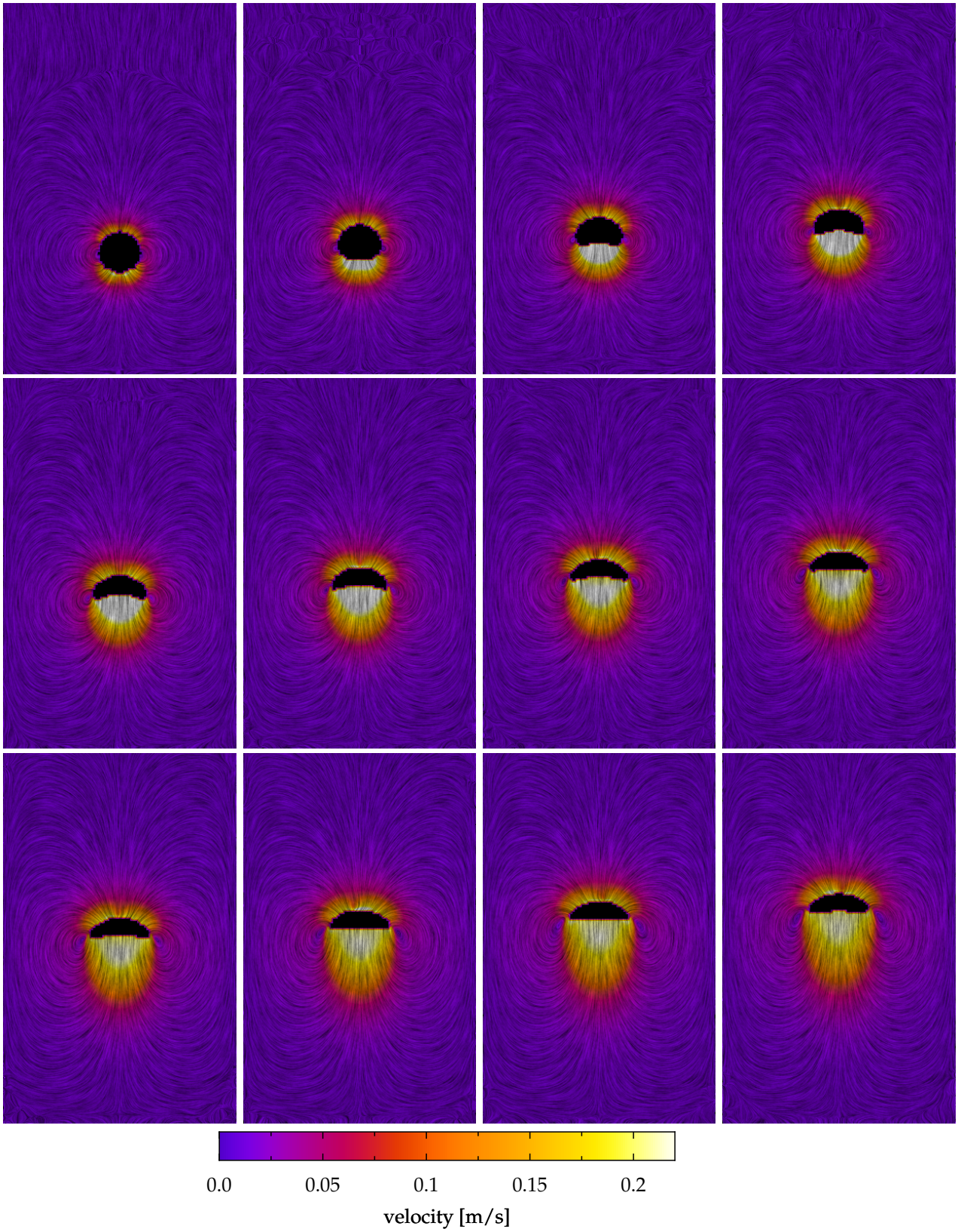


Figure 3.8: Visualization using a line integral convolution (LIC) [CL93] of the streamlines of the fluid. The colors correspond to the velocity of the fluid. 1000 time steps (11.7 ms in real-time) passed between each image.

3.4.3 Conclusions

According to the two qualitative evaluation setups, the model behaves in an expected and physically valid manner. Although the quantitative results deviate slightly from the experiment and other simulation results, it is still applicable to practically relevant problems. The detailed comparison with a reference simulation indicates that the reason for the deviation is likely to be found in the calculation of the surface curvature.

Chapter 4

Aspects of High Performance Computing

In the following section, various aspects of high performance computing (HPC) will be discussed, based on the described model for free surface flows in 3D.

4.1 Employed High Performance Computing Platforms

Two different HPC platforms have been used extensively: a cluster architecture using the AMD Opteron CPU and the former German national supercomputer Hitachi SR8000-F1. Both will briefly be characterized in the next two sections.

4.1.1 Opteron Cluster

With the increasing performance of CPUs, the access to memory became the bottleneck limiting the overall performance of a computer [BGK96, Wil01]. The introduction of multi-core CPUs multiplied the severity of this problem, because in general all CPUs[1] have to share one single bus to access memory. One way to solve this issue was the introduction of so-called non-uniform memory access (NUMA) architectures, i.e., each CPU gets a dedicated path to access a block of private memory. Nevertheless, a CPU can access the data located in the memory block of a different CPU, but this involves the usage of special software or hardware which significantly slows down the access. From a programmer's point of view, the access to local memory or remote memory is completely transparent except for the different bandwidth and latency.

The Opteron cluster located at the System Simulation Group at the University of Erlangen-Nuremberg, Germany, is a typical representative of such a NUMA architecture. This cluster consists of eight nodes with four AMD Opteron CPUs and nine nodes with two CPUs running at 2.2 GHz. All nodes use a Gigabit network with a latency of about 50 μs

[1] We will use the term CPU also for the cores of a multi-core CPU.

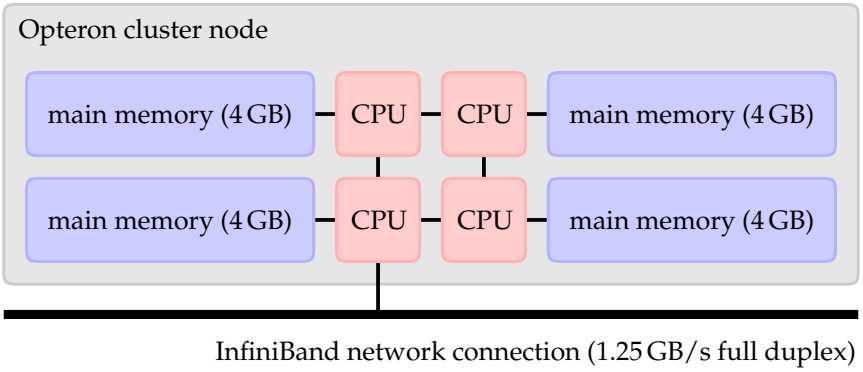


Figure 4.1: Schematic view of a computing node of the Opteron cluster which consists of eight quad-core nodes and nine dual-core nodes.

and a bandwidth of 0.12 GB/s. Additionally, the quad nodes are connected via InfiniBand, a high performance network with a latency of 5 μs and a theoretical bandwidth of 1.25 GB/s. Except for special tests, the InfiniBand network of the quad nodes was used for performance measurements.

Figure 4.1 depicts the schematic view of one quad node of the Opteron cluster. Each CPU is connected to a dedicated block of 4 GB main memory and to two other CPUs via HyperTransport links. This ring network allows each CPUs to access the memory of all other CPUs with at most two intermediate network nodes (INN). Only one CPU is directly connected to the InfiniBand network with a HyperTransport link. All other CPUs have to use the ring network first to access the InfiniBand network which results in a reduced network bandwidth.

The ping-ping benchmark [NN97] shall be used here to characterize the network of this cluster: Two messages of size S are sent simultaneously forth and back N times between each possible pair of CPUs in a network; one pair at a time. The required time T is measured for each pair. The total time T consists of two parts

$$T = N\,(T_\mathrm{L} + T_\mathrm{T}) \quad ,$$

with T_L as the time it takes to initiate the communication, called latency, and T_T as the time required for the actual transmission of the message. The quotient of the total size of transmitted messages and T_T is defined to be the bandwidth B:

$$B = \frac{N \cdot S}{T_\mathrm{T}}$$

If only a few large messages are transmitted, the latency T_L can be neglected and the network bandwidth B can be approximated as $B \approx \frac{N \cdot S}{T}$. To measure the latency, small or empty messages are transmitted many times. In this case, the transmission time T_T can be neglected and the latency is $T_\mathrm{L} = \frac{T}{N}$.

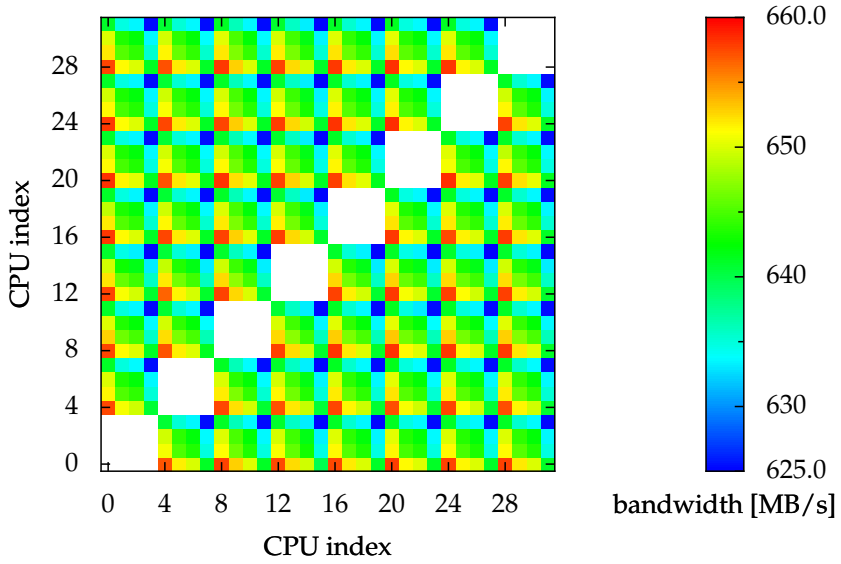


Figure 4.2: Result of the ping-ping benchmark for the Opteron cluster. The intra-node network bandwidth varies between 880 and 1000 MB/s and is not shown in this diagram.

For measuring the network bandwidth of the Opteron cluster, a message size of 512 MB was used. The results for the inter-node bandwidth shown in Fig. 4.2 exhibits a regular pattern due to the asymmetrical access of the CPUs to the InfiniBand network. Each INN between a CPU and the InfiniBand network reduces the network bandwidth by about 18 MB/s. Therefore, three groups of network bandwidths can be identified: 660 MB/s (no INN), 642 MB/s (one INN), 625 MB/s (two INNs).

It seems advisable, to select the CPU with the direct InfiniBand connection for inter-node communicating. To measure the influence of this effect on the performance of the simulation code, two parallel simulation runs were performed. In the first run, only CPUs with the highest network bandwidth were used; for the second run, the CPUs with the lowest network bandwidth were chosen. The difference in the performance was below the normal measurement fluctuations. Therefore, the effect can be neglected for further considerations

4.1.2 Hitachi SR8000-F1

Unlike many other supercomputers, the Hitachi SR8000-F1 is not a classical vector computer, although it inherits some of the typical vector processing features. Its RISC (reduced instruction set computer) CPUs [SKH+99], which are enhanced IBM PowerPC processors running at 375 MHz and a theoretical peak performance of 1.5 GFlop/s, are grouped in so-called nodes. Each node consists of nine CPUs of which eight can be used in application codes. Among other minor tasks, the remaining processor handles I/O operations. These

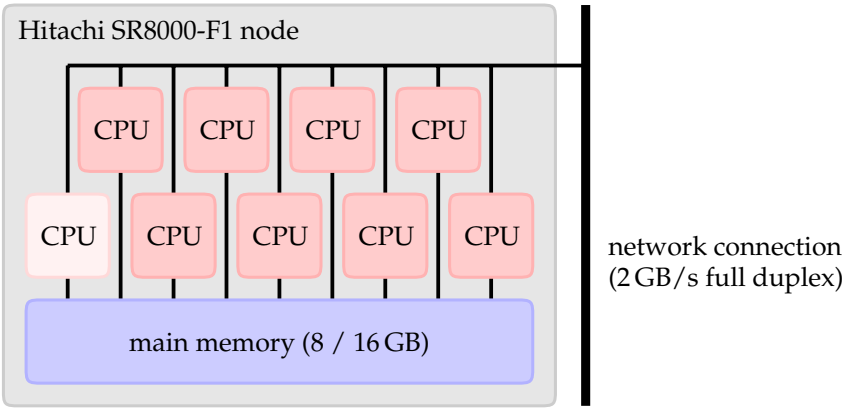


Figure 4.3: Schematic view of a computing node of the Hitachi SR8000-F1 which contains 168 of these nodes. Only eight of the nine CPUs are available for computing.

nine CPUs can access the local node memory of 8 or 16 GB with 32 GB/s as depicted in Fig. 4.3. The nodes are connected with a 3D crossbar providing a network bandwidth of 2 GB/s in full duplex mode [SMK03]. The performance evaluation runs were executed on the Hitachi SR8000-F1 at the Leibniz Computing Center in Munich, Germany. This machine has 168 nodes and delivers a peak performance of 2.0 TFlop/s. It held rank 127 in the TOP500[2] list of June 2004 and has been replaced 2006 by its successor, a SGI Altix 4700 with a peak performance of more than 62 TFlop/s.

A major difference between the architecture of the Hitachi SR8000-F1 and a normal RISC-based supercomputer is the extension to the CPU's instruction set. One important enhancement is the increase of the floating point registers to 160 which can be put to good use especially in scientific codes. Further extensions are special prefetch and preload instructions which both retrieve data from main memory and store it in the cache or in a floating-point register, respectively. These instructions are issued by the compiler at appropriate points in the code to hide memory latency (see Sec. 4.4.1 for further details about the memory hierarchy). Therefore, the compiler needs to know the memory access pattern of the code during compilation. If the code does not allow for such an automatic memory access analysis, the application performance degrades drastically. In this sense, these RISC CPUs behave similar to vector CPUs which also rely on a simple and predictable data flow in the application.

The prefetch/preload mechanisms in combination with the extensive software pipelining done by the compiler are called pseudo vector processing (PVP) as they are able to provide an uninterrupted stream of data from main memory to the processor circumventing the penalties of memory latency.

[2]The web page `http://top500.org/` lists the 500 most powerful HPC systems currently in operation.

For parallelization issues several different software libraries have been provided by Hitachi. If treated as an MPP (massively parallel processing/processors) both intra- and inter-node communication can be handled with the standard message passing interface (MPI). For intra-node communication there are two alternatives which exploit the benefits of the shared memory on a node more efficiently: the standardized OpenMP interface version 1.0[3], and the proprietary COMPAS (cooperative micro-processors in a single address space) which allows for easy thread parallelization of appropriate loops and sections.

4.2 Measuring Performance

Typically, the performance of simulation codes is measured in the amount of floating-point operations that have been performed in a certain period of time. This leads to the well-known unit of one million floating-point operations per second, abbreviated as MFlop/s [Smi88]. Reducing the complex notion of performance to a single number of course has its limitations [DMV87]. A higher MFlop/s rate does not necessarily mean that the awaited results of a simulation are available faster compared to another implementation with a lower MFlop/s rate. Redundant or ineffective computations might increase the MFlop/s rate while prolonging the overall runtime.

In the LBM community, another performance indicator is commonly used. It is defined as million cell (also known as lattice site) updates per second, abbreviated as MLup/s. This value strongly depends on the details of the implemented LB model: 2D or 3D domain, number of distribution functions per cell, approximation of the collision operator, etc. For the user of an LBM application, this value is very handy, since the overall runtime for a given domain size and the number of time steps can easily be computed.

A comparison of different LBM implementations, based solely on any of these performance values, is difficult. In any case, the specific LBM model and implementation details have to be considered. For evaluating the effectiveness of performance optimizations applied to one specific LBM implementation, measuring the MLup/s should be preferred, since it directly relates to the runtime of the simulation.

4.3 Code Characterization Based on the Instruction Mix

4.3.1 Introduction

Quite early, computer scientists realized that the so-called instruction mix of a code, i.e., the number of instruction grouped in certain categories, is essential for a first characterization

[3]More details about OpenMP can be found on the OpenMP homepage `http://www.openmp.org/`.

of a code. Interesting categories of instructions are for example floating-point or integer operations, read or write access to the memory, and conditional branches.

In the beginning, researchers performed a static analysis of the instruction mix with tools that examined each instruction in a binary executable file [Gib70]. It is of course difficult to predict the performance of a code, based on this static analysis, because simulation codes typically feature a small computational kernel which is nested in several layers of loops. The total amount of floating-point operations in the code might therefore be low, but during runtime these operations are performed very often.

With the increasing complexity of CPUs, more sophisticated means to monitor its activity have been implemented. On modern CPUs it is now possible to retrieve information about the type of the executed instructions. This offers the possibility to measure the executed instruction mix dynamically. Since this low-level measurement requires detailed knowledge about the internals of each CPU, libraries have been implemented to abstract from the CPU-specific details to a high-level application interface.

The most sophisticated library which supports a broad range of CPUs is the Performance Application Programming Interface[4] (PAPI). It allows for configuring and reading the performance counters of a CPU. Depending on the architecture, these counters can be set up to measure different types of events like the total number of instructions or the number of floating-point operations performed. Unfortunately, the AMD Opteron CPU which has been used, offers only a limited set of measurable events.

4.3.2 Results

Table 4.1 summarizes the measured instruction mix for a plain LBM implementation from the SPEC benchmark suite (see Sec. 5.1 for more details) and the FSLBM implementation. Both were tested with a completely filled domain of size $200 \times 200 \times 200$. The latter was additionally tested with the breaking dam setup described in Sec. 3.4.1.1. The instructions have been categorized in three groups: floating-point operations, conditional branches, and the remaining instructions which mainly consist of memory access operations and logical and mathematical operations with integer values. Furthermore, the number of instructions has been divided by the number of fluid and interface cells. The handling of the other two cell types does not require any computations.

For the SPEC code, the floating-point operations constitute 77% of the total number of instructions. One of the three conditional branches is the test whether a cell is an obstacle or fluid—the only two cell types present in a regular LBM code. The remaining two conditional branches result from the nested loops enclosing the streaming and collision operations. The group of unspecified instructions contributes only one quarter of the performed instructions.

[4] Further information can be found on the PAPI homepage `http://icl.cs.utk.edu/papi/`.

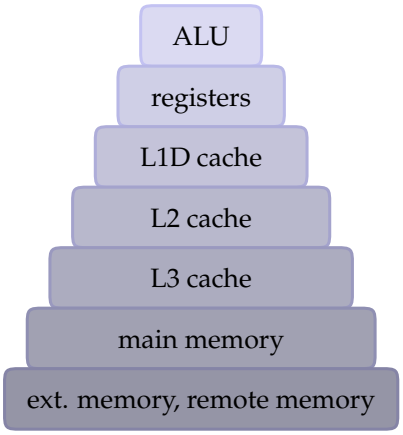
number of instructions per fluid or interface cell	SPEC code `470.lbm`		FSLBM filled setup		FSLBM breaking dam	
floating-point	251	(77%)	386	(42%)	491	(28%)
branches	3	(1%)	66	(7%)	169	(10%)
unspecified	72	(22%)	464	(51%)	1083	(62%)
total	326	(100%)	915	(100%)	1743	(100%)
MFlop/s	652		645		496	
MLup/s	2.68		1.72		4.16	

Table 4.1: Comparison of the instruction mix for the SPEC implementation and two different setups for the FSLBM implementation on an Opteron CPU.

The picture changes drastically for the FSLBM code: Even without any interface cells (filled setup), the number of unspecified instructions surpasses the number of floating-point operations due to the more complex collision operator and the required cell type tests. For the breaking dam setup, the percentage of floating-point operations drops to 28%, because the handling of interface cells involves several tests and operations which use only a few floating-point operations.

Another important aspect is the rise of conditional branches from 1% for the SPEC code to 7% for the filled setup of the FSLBM code. Modern all-purpose CPUs, such as, the Intel Pentium 4 or the AMD Opteron feature several techniques, e.g., speculative execution, branch predictors, and out-of-order execution [HP06] to circumvent the performance degradation inflicted by conditional branches. Their effectiveness is obvious from the MFlop/s performance which drops only slightly from 652 (SPEC code) to 645 MFlop/s (FSLBM, breaking dam setup).

Special-purpose architectures, such as, vector CPUs which are often used in the HPC context, suffer severely from the increase of conditional branches. They are optimized for small computational kernels with simple and predictable flows for both the instruction and the data stream. Although the Hitachi SR8000-F1 is not a classical vector architecture, its vector-like processing concept makes it vulnerable for the performance degradation of conditional branches. Here, the performance of a single PowerPC CPU drops from 2.5 (SPEC code) to 0.5 MFlop/s (FSLBM, filled setup). It can be deduced that the Hitachi SR8000-F1 and other vector architectures are not well suited for this kind of complex simulation codes.



memory type	capacity	bandwidth	latency
L1D cache	16 KB	16 GB/s	1 cycle
L2 cache	256 KB	32 GB/s	$>$ 5 cycles
L3 cache	3 MB	32 GB/s	$>$ 12 cycles
main memory	2 GB	6 GB/s	$>$ 100 cycles

Figure 4.4: Typical memory hierarchy for cache-based architectures. The data refers to the Intel Itanium 2 architecture [Int02]. The abbreviation ALU stands for the arithmetic and logic unit of a CPU.

4.4 Data Layout and Access Optimizations

4.4.1 Introduction

In order to mitigate the effects of the growing gap between theoretically available processor speed and main memory performance, today's computer architectures are typically based on hierarchical memory designs, involving CPU registers, several levels of cache memories (caches), and main memory [Han98]. Remote main memory and external memory (e.g., hard disk) can be considered as the slowest components in any memory hierarchy. Figure 4.4 illustrates the memory architecture of a common high performance workstation. The AMD Opteron CPU has only two levels of cache. Detailed data about their performance is not available.

Efficient execution in terms of work units per second can only be obtained if the codes exploit the underlying memory design. This is particularly true for numerically intensive codes, such as, an LBM implementation. Unfortunately, current compilers cannot perform highly sophisticated code transformations automatically. Much of this optimization effort is therefore left to the programmer [GH01].

Generally speaking, efficient parallelization and cache performance tuning can both be interpreted as data locality optimizations. The underlying idea is to keep the data to be processed as close as possible to the corresponding ALU. From this viewpoint, cache optimizations form an extension of classical parallelization efforts.

Research has shown that the cache utilization of iterative algorithms for the numerical solution of systems of linear equations can be improved significantly by applying suitable combinations of data layout optimizations and data access optimizations [BDQ98]. The idea behind these techniques is to enhance the spatial locality as well as the temporal

locality of the code.

Although the LB method is not an iterative algorithm, its notion of time steps exhibit the same characteristics for the data layout and access. Thus, the same optimizations are applicable.

4.4.2 Combining Data Layout and Loop Ordering

Accessing main memory is very costly compared to even the lowest cache level. Hence, it is essential to choose a memory layout for the implementation of FSLBM which allows to exploit the benefits of hierarchical memory architectures.

As described in Sec. 3.2.1, the cell data is stored in two arrays, one for time step t and another one for $t + 1$. Several layouts are possible for such an array depending on the ordering of the four array indices. The ordering of the three indices corresponding to the three spatial dimensions is irrelevant, since the coordinates are treated equally in the FSLBM implementation. The remaining index for the cell entry, however, is treated differently. Its position relative to the other indices is therefore essential for the performance of the data layout. The four different possibilities for placing the cell entry index are listed in Alg. 4.1.

Algorithm 4.1 Four different array index orderings for the cell data.

```
typedef union {
  double d;
  long int i;
} Entry;

Entry ZYXE    [SIZE_Z]    [SIZE_Y]    [SIZE_X][29],
      ZYEX    [SIZE_Z]    [SIZE_Y][29][SIZE_X]    ,
      ZEYX    [SIZE_Z][29][SIZE_Y]    [SIZE_X]    ,
      EZYX[29][SIZE_Z]    [SIZE_Y]    [SIZE_X]    ;
```

Most high-level programming languages use the so-called row major ordering[5], i.e., if the right-most index is incremented, the next memory location of the array is accessed. The ZYXE ordering is possibly the most commonly used ordering for LBM codes, since it is very intuitive: All entries for each cell are located contiguously in memory. Current research [WZDH06] suggests, however, that other layouts might be advantageous.

Just as important as the data layout is the data access pattern. In our case, the ordering in which the cells become the AC does not influence the simulation result. Thus, we can

[5]The most famous exception to this rule is the programming language Fortran which uses column major ordering.

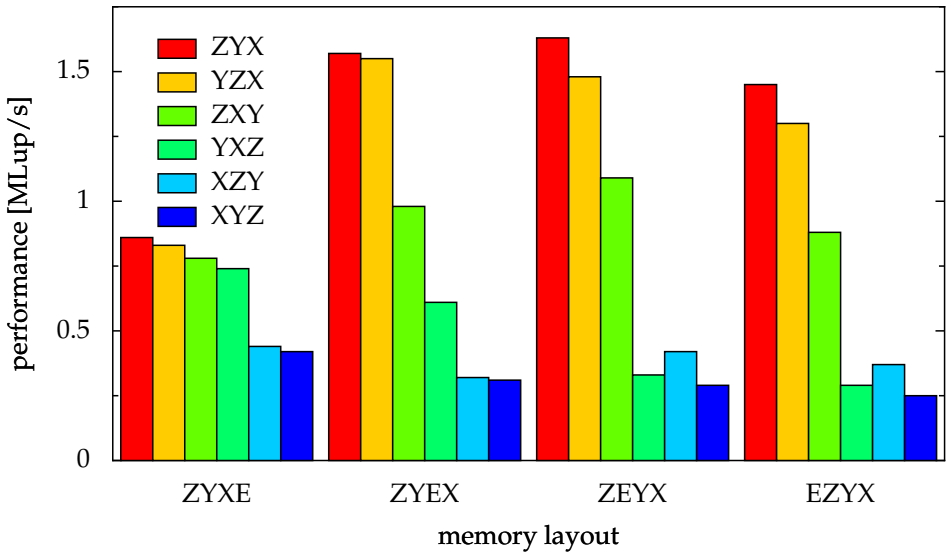


Figure 4.5: Performance of the simulation code for four different memory layouts and six different loop orderings (filled setup). The different colors correspond to the different loop orderings.

freely choose one of the six canonical orderings which is used in every sweep. As an example, the loop ordering ZYX is outlined in Alg. 4.2.

Algorithm 4.2 One of the six possible loop orderings. This ordering is called ZYX.

```
for (z = 0; z < SIZE_Z; z++) {
  for (y = 0; y < SIZE_Y; y++) {
    for (x = 0; x < SIZE_X; x++) {
      /* perform calculations for sweep */
    }
  }
}
```

The performance for each combination of data layout and loop ordering is shown in Fig. 4.5 for the filled setup. As expected, the ZYX loop ordering (*red bars*) is fastest, because the strides for the memory access are smallest in this case. The slowest loop ordering XYZ has exactly the opposite ordering as the data array. Generally speaking, the right-most spatial array index x is most important for the performance of a loop ordering. More interesting is the comparison of the different memory layouts. The canonical ZYXE layout is slowest for the most efficient loop ordering ZYX (only 53% compared to the best performance). The performance of the other three layouts is significantly better with ZEYX as the most successful.

To explain these differences, PAPI was used to measure the events of L1 and L2 cache

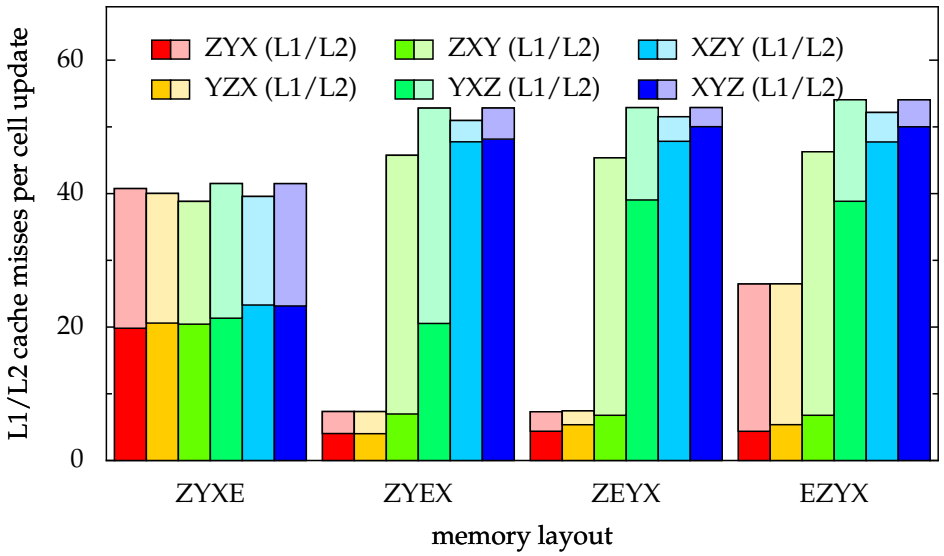


Figure 4.6: Number of L1/L2 cache misses per cell update (only fluid or interface cells are counted) for different memory layout and loop orderings (filled setup).

misses. The results in Fig. 4.6 reflect the findings of the previous performance measurements. With an increasing number of L1 or L2 cache misses, the performance degrades as expected. The impact of L2 cache misses on the performance is stronger than for L1 cache misses. Based on this observation, it can be concluded that the difference in access latency between L2 cache and main memory is significantly higher than the difference between L1 and L2 cache. This is in accordance with the experience and available specifications for other CPUs. Reliable data on cache and memory timings for the AMD Opteron CPU are not available.

The equivalent performance and cache misses measurements for the breaking dam setup qualitatively gives the same results. However, the number of cache misses per cell update is about one third compared to the completely filled setup because the gas cells do not require any computations.

4.4.3 Interaction of Data Access Patterns and Cache

In order to explain the unexpected results of the performance for the different data layouts, the interaction of a simplified sweep as listed in Alg. 4.3 with a cut down model of a cache is instructive. The sweep features a memory access pattern which is common for many sweeps in the actual code. It reads data from the 18 neighbors of the AC and stores the result into the AC itself. In order to be able to visualize the data access, the dimension of the domain and the number of state variables (entries) for each cell are kept to a minimum which still exhibits all relevant effects.

The model of the cache consists of just one cache level. Data is assumed to reside in

Algorithm 4.3 Simplified sweep with a typical access pattern for the ZYXE data layout.

```
#define SIZE_X 5
#define SIZE_Y 5
#define SIZE_Z 5
#define SIZE_E 2

Entry c[SIZE_Z][SIZE_Y][SIZE_X][SIZE_E];

void sweep(void) {
  int x, y, z, dx, dy, dz;

  /* nested sweep loops */
  for (z = 1; z < SIZE_Z; z++) {
    for (y = 1; y < SIZE_Y; y++) {
      for (x = 1; x < SIZE_X; x++) {

        /* typical access pattern of a simplified sweep */
        for (dz = -1; dz <= 1; dz++) {
          for (dy = -1; dy <= 1; dy++) {
            for (dx = -1; dx <= 1; dx++) {
              /* skip corners of the 3x3x3 cube */
              if (dx*dy*dz == 0) {
                /* read access */
                readFrom(&( c[z+dz][y+dy][x+dx][0] ));
              }
        }}}
        /* write access */
        writeTo(&( c[z][y][x][0] ));
  }}}
}
```

employed data layout	ZYXE	ZYEX	ZEYX	EZYX
relative number of cache misses	48.3%	0.8%	0.4%	0.4%

Table 4.2: Results for the simplified cache model for a domain size of 60^3 and 29 cell entries.

the cache, if it has been read or written during the update of the current or previous cell. Data accessed before the previous cell update is considered to be flushed from cache. More complex cache characteristics, e.g., size or associativity are not modeled.

A comparison of the data access patterns for the four different data layouts is depicted in Fig. 4.7. The abscissa corresponds to the number of updated cells during the sweep. The ordinate is equivalent to the relative position of the data in memory with the origin located at the beginning of the grid data array. Every dot stands for a read or write access to the corresponding memory location. Black and gray dots represent cache misses and cache hits, respectively. It becomes obvious that the ZYXE layout suffers from the separation of cell entries with the same index. The relative number of cache misses decreases the more the cell entry index E moves to the left which is the position of the slowest-moving array index. Although this simplified model cannot explain why the ZEYX data layout shows the best performance, the general tendency is well reproduced. The quantitative results for a larger domain size of $60 \times 60 \times 60$ with the correct number of 29 cell entries are given in Tab. 4.2.

These results for the optimal memory layout agree with the findings by Donath et al. [Don04].

4.4.4 Grid Compression

In the case of the regular LB method without the free surface extension, a data layout has been devised by J. Wilke and the author of this thesis [PKW+03, WPKR03] which almost halves the memory requirements. Consequently, the spatial locality of memory access is improved. The method has also been extended to 3D [Igl03]. This so-called grid compression will be explained below.

For a standard LBM, the streaming operation requires data only from neighboring cells as depicted in Fig. 2.4. Instead of storing the data of a streaming operation (e.g., from cell $(1,1)$) in a separate grid, the data is stored in the same grid, but shifted by one column and one row towards the lower left corner (e.g., to cell $(0,0)$). If this is done in a lexicographic order starting at cell $(1,1)$, the original data of the blue cells for time step t has been shifted to the red cells for time step $t+1$ as shown in Fig. 4.8. The shifting ensures that no data is overwritten which is required for the streaming operation of another cell. The streaming operation for the next time step has to start at the upper right corner with a shifting in the

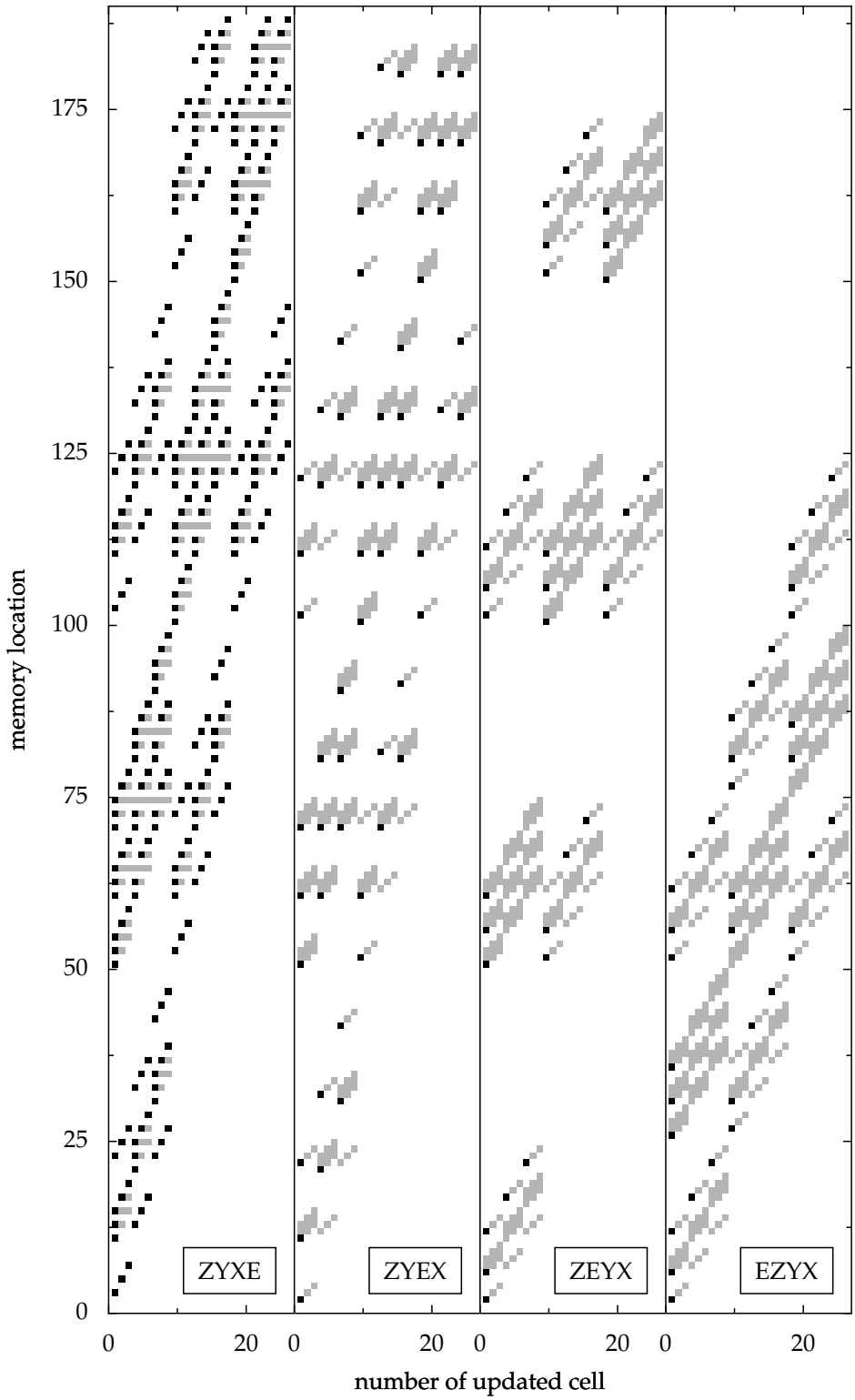


Figure 4.7: Comparison of the data access patterns for the four different data layouts.

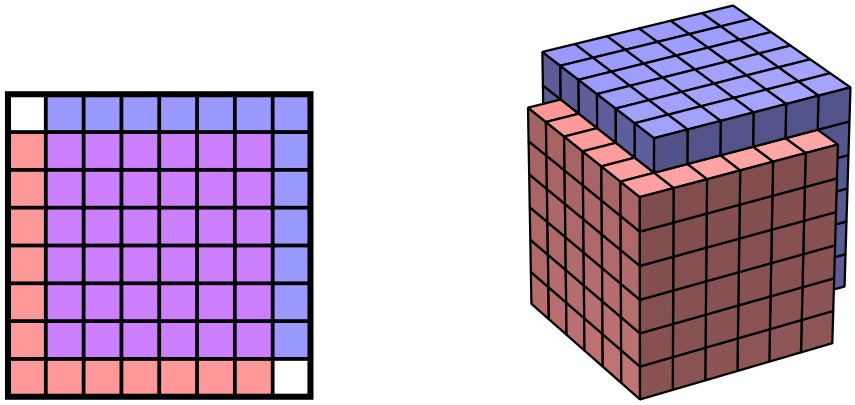


Figure 4.8: Grid compression technique for 2D (*left*) and 3D (*right*) domains. In one time step, data is read from the blue cells and written to the red cells. In the following time step, the roles are swapped and so forth. The data from the magenta cells which are hidden in the 3D case, is both read and written in all time steps.

opposite direction. With grid compression, the ordering of the cell updates is restricted to the described lexicographic ordering. The extension to the 3D case is straightforward.

Both in 2D and 3D, the application of this method almost halves the memory requirements for the cell data. Details about the performance impact of grid compression can be found in [PKW+03, WPKR03, Igl03].

In principle, this technique could also be applied to the FSLBM implementation. Yet it would be necessary to use more than just two overlayed grids due to the higher number of sweeps over the grid. A similar method to enable the combined use of grid compression and loop blocking is described in [Wil03] for 2D. The larger number of required overlayed grids limits the potential gain in performance. Thus, grid compression was not implemented for the FSLBM implementation.

4.4.5 Data Access Optimizations by Loop Blocking

Data access optimizations change the order in which the data are referenced in the course of a sweep, while respecting all data dependencies. The access transformations for the implementation of the FSLBM code are based on the loop blocking (loop tiling) technique. The idea behind this general approach is to divide the iteration space of a loop or a loop nest into blocks and to perform as much computational work as possible on each individual block before moving on to the next block. Blocking an individual loop means replacing it by an outer loop which controls the movement of the block, and an inner loop which traverses the block itself. If the size of the blocks is chosen appropriately, loop blocking can significantly enhance cache utilization and thus yield substantial application speedups [AK01, GH01, KW03].

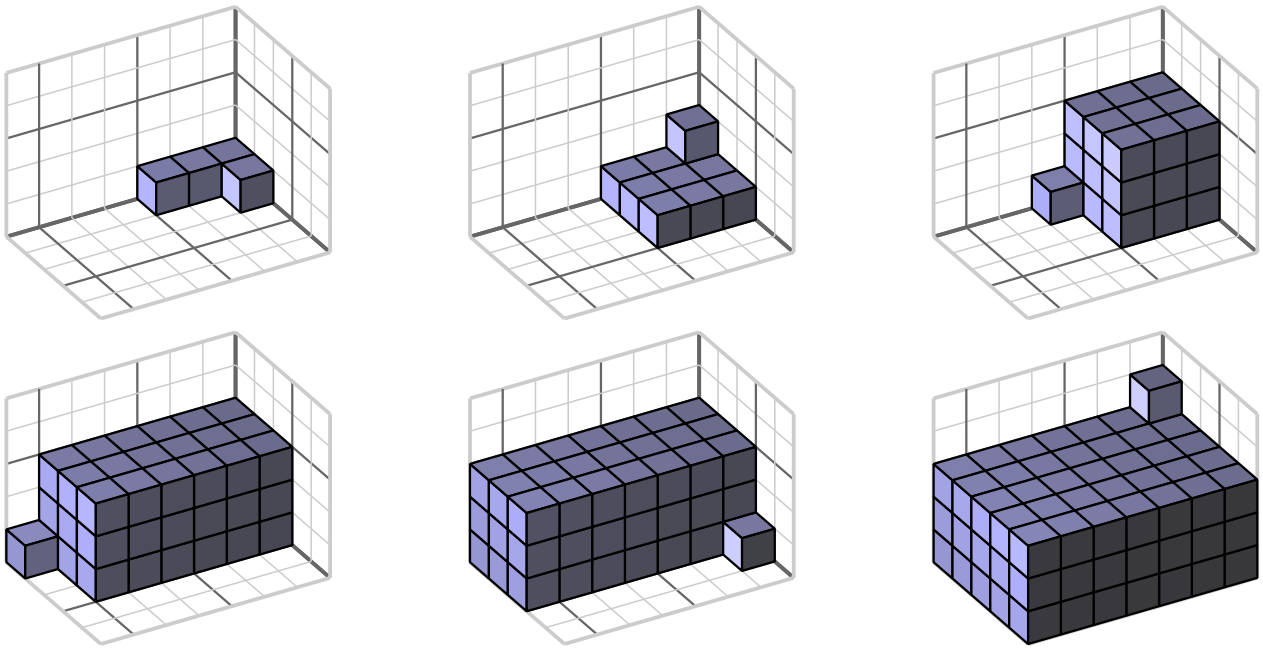


Figure 4.9: Different time steps depicting the updated cells for a sweep with 3-way loop blocking. The blocking size is $3 \times 3 \times 3$ for the spatial coordinates.

Figure 4.9 illustrates the ordering for the cell updates for a 3-way loop blocking in 3D, i.e., all three spatial array indices are blocked. The block size in this example is 3^3. After updating the first three cells, the sweep interrupts its proceeding along the x axis and fills the 3^3 block first (upper three images). Analogously, the next blocks are filled. Special care has to be taken at the end of the domain if the grid size is not divisible by the block size.

If several iterations or time steps need to be calculated, the time loop enclosing the spatial loops can also be blocked. The research carried out by Wilke, Iglberger, Kowarschik, and the author of this thesis in fact showed that this procedure delivers the highest performance benefit in comparison to the blocking of the spatial loops [PKW+03]. However, this time loop blocking cannot be applied to the FSLBM implementation, since a new time steps requires the updated pressure of the gas bubbles. For calculating the new pressure, the current time step has to be completed.

For simple computational kernels, appropriate block sizes can be calculated by taking the size of the cache into account. The FSLBM implementation with its sweeps is too complex for such an approach. Instead, the optimal block size values are found by scanning a field of reasonable block sizes for the best performance. The scanning was performed for the ZEYX ordering, the filled simulation setup and block sizes ranging from 1 (no blocking) to 32 for each of the three spatial array indices. As an example, a cut through the search space in the xz plane is displayed in Fig. 4.10. The fastest of the $32^3 = 32768$ variants is the completely unblocked version with 1.74 MLup/s. The next best performance of 1.73 MLup/s can be achieved with a blocking size of $(23, 1, 2)$, $(25, 1, 2)$, $(29, 1, 2)$, and

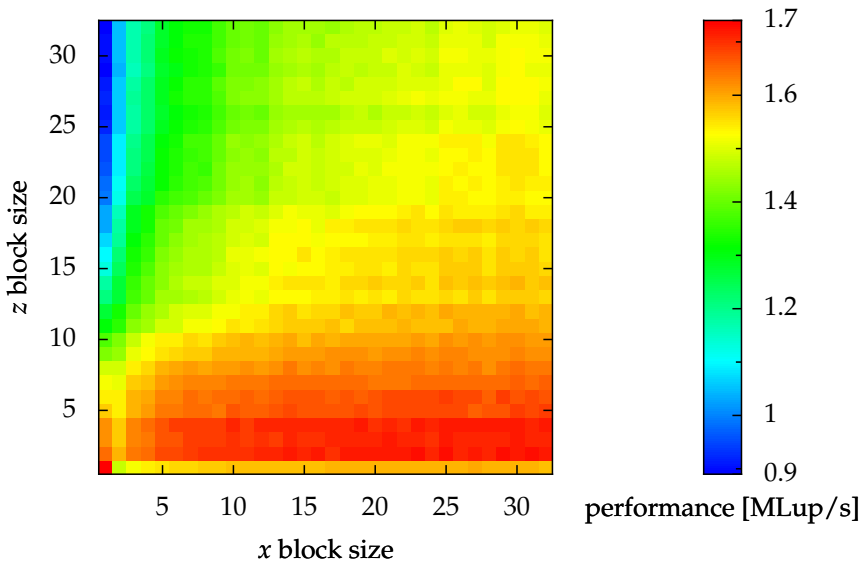


Figure 4.10: Cut surface through the search space for the optimal block size in the xz plane. Loop blocking in the y direction is not applied.

$(31, 1, 2)$ for the block size in (x, y, z) direction. It can be deduced that for the FSLBM implementation, the blocking in y direction is disadvantageous. For all of the four mentioned block size configuration, the number of cache misses is about 3% (L1) and 4% (L2) lower compared to the unblocked version. Yet the overhead of the additional loops required for the loop blocking obviously outbalances this minor data locality improvement.

4.4.6 Results

By using the best combination for the data layout and the loop ordering, the performance could be improved from 0.86 MFlop/s for the canonical data layout to 1.63 MFlop/s for the ZEYX layout which is an increase of 90%. As we will see in the next section about parallel computing, the chosen data layout ZEYX features interesting characteristics which can also be exploited to reduce the size of the communication messages.

The application of loop blocking to a regular LBM is able to boost the performance significantly. In particular, the blocking of the time loop is most rewarding, yet also most intricate [PKW+03, WPKR03, Igl03]. For the larger data sets required for the FSLBM code, however, blocking is not able to improve the performance. Due to the calculation of the gas pressure for bubbles which requires a completed sweep over the entire domain, the blocking of the time loops is not possible.

4.5 Parallel Computing

Today, high performance computing virtually is a synonym for parallel computing. Therefore, we will investigate several topics of parallel computing based on the FSLBM implementation.

In the first part, a brief introduction to parallel computing and the measurement of parallel performance is given. The second and third section describe the method of parallelization for the cell and the gas bubble data, respectively.

4.5.1 Introduction

The goal of parallel computing is the increase of computational performance by distributing the workload on many CPUs. This performance increase is often measured as the speedup S for a parallelized code which is defined to be

$$S = \frac{T_1}{T_N} \tag{4.1}$$

with T_1 and T_N as the runtime for the sequential code and the parallel code on N CPUs, respectively [MSM05]. Ideally, the speedup would be N if the code was using N CPUs[6]. Some fraction of the code might not be parallelizable. Hence, the sequential runtime T_1 consists of two parts,

$$T_1 = s + p \quad , \tag{4.2}$$

a part s which can only be run sequentially and a part p which can be parallelized. Since only the parallelizable part is affected by the number of employed CPUs, the runtime T_N for the parallel code is

$$T_N = s + \frac{p}{N} \quad . \tag{4.3}$$

By inserting Eqs. 4.2 and 4.3 into Eq. 4.1, Amdahl's Law [Amd67] is gained

$$S = \frac{T_1}{s + \frac{p}{N}} \quad . \tag{4.4}$$

The values for s and p are normally not known a priori for a parallel code. By performing a least squares fit [SK06] of Eq. 4.3 to a number of measured runtimes with different number of CPUs, these values can be approximated. In the following, we will frequently use this method to characterize the parallel performance of the FSLBM implementation.

The evaluation of the scalability of a code, i.e., how much a code benefits from an increase of employed CPUs, is often split into two application scenarios [WA04]. When the simulation parameters, such as, domain size and the required time steps are kept fixed,

[6] Due to cache effects, the speedup can be even higher than N. Except for some pathological examples, this is hardly ever experienced for production codes.

the runtime depends on number of employed CPUs as given by Amdahl's Law in Eq. 4.4. This scenario is called speed-up or strong scaling. In the second application scenario, the number of CPUs is increased while refining the resolution of the simulation domain, such that the wall clock computing time remains constant for linear scalability. For most models, the spatial and the temporal resolution are not coupled and can be chosen separately to a certain extent, i.e., if the resolution is doubled in one spatial dimension, the runtime will also double. In the case of a 3D domain, a doubled spatial resolution in all three dimension would ideally require eight times more CPUs to keep the required wall clock computing time constant. This application scenario is called scale-up or weak scaling.

For the LBM model, however, the spatial and the temporal resolution are inherently coupled as explained in Sec. 2.2.1. Thus, the temporal resolution increases if the spatial resolution is doubled in the case of the weak scaling performance. To retain the same physical setup (e.g., fluid viscosity, simulated time period), the number of required cell updates would be 32 times higher than for the original spatial resolution (see Sec. 3.4.2.2). Instead of eight times more CPUs for a doubled spatial resolution and a constant wall clock computing time, 32 times more CPUs would be required for measuring the weak scaling performance. The subdomain size for each CPU would be reduced to one fourth of the original subdomain size. This contradicts the initial idea of the weak scaling performance. Therefore, only the strong scaling performance is investigated in the following.

4.5.2 Handling of Cell Data

4.5.2.1 Domain Partitioning

The most common way to distribute the workload on several CPUs is the method of domain partitioning. For this purpose, the entire computational domain is divided into several subdomains. Each subdomain is assigned to a processing unit (PU) which can be a single CPU or a group of CPUs accessing the same memory. In general, every PU can only access the local memory where the data for the subdomain is stored. If a PU only had to access data in its own subdomain, no communication would be necessary. Yet the streaming operation of the LBM and several other operations of the free surface extension need to access data from neighboring cells which might be located in adjacent subdomains. Assuming a distributed-memory environment, this cannot be accomplished without explicit communication.

The restrictions for cell data access formulated in Sec. 3.3 allow for a simple parallelization strategy. As described above, the cell data is divided into several subdomains and is distributed to several PUs as depicted in Fig. 4.11. The PUs perform the sweeps over their subdomain in parallel updating all dark blue cells. The pale blue cells at the interface to neighboring subdomains, the so-called halo cells, replicate the data of the first layer of cells of the neighboring subdomain. These halo cells are necessary, since the update of the

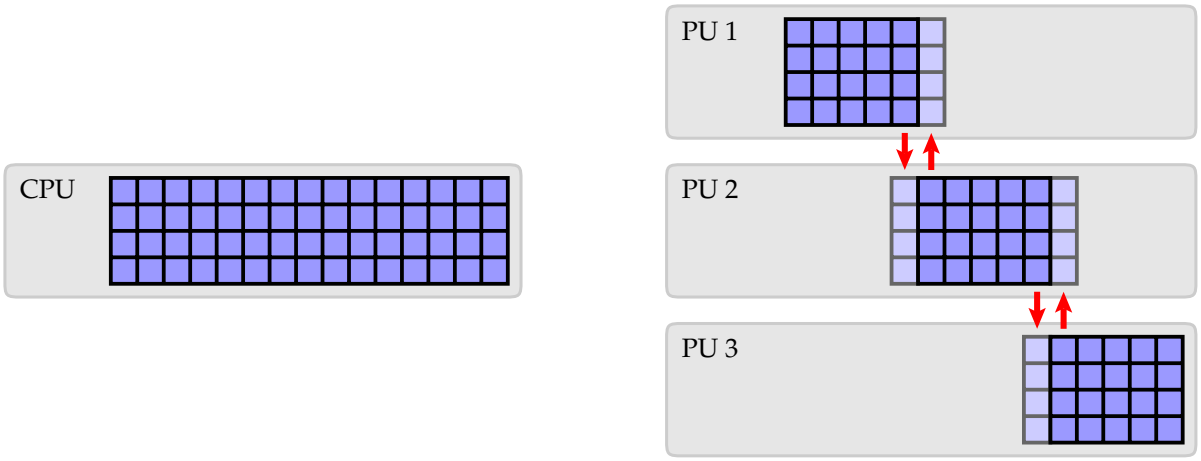


Figure 4.11: Instead of just one CPU (*left*), several PUs can be used, each working only on a subdomain (*right*). After each sweep, communication (*red arrows*) is necessary to update the halos (*pale cells*) which hold the data of neighboring domains.

dark blue cells relies on the data of neighboring cells. One layer of halo cells is sufficient, because the data required for an update of the AC may only be read from ACNs.

By definition, the data of the halo cells has to be identical to the data of the associated original cells. This also implies that any change made to the halo cells would have to be reflected in the original cells. Due to the second restriction in Sec. 3.3, however, this is not allowed. The synchronization of the halo cells and the originating cells has to be considered only from original to halo cells. Thus, after each sweep, the data from the original cells is copied into the halo cells using explicit communication.

In Fig. 4.11, the domain is partitioned only in one dimension. The domain could also be partitioned in two or three dimensions (for computational domains in 3D) which would lower the number of cells to be communicated. Yet this comes at a price [KPR+06]:

- The number of communication partners and thus the number of required messages increases significantly.
- Before sending a message, its data has to be merged in a contiguous block of memory. This strains both the CPU and the memory bus. After a message has been received, its data has to be copied back to the correct memory locations. It will be shown that the latter can be avoided for a 1D domain partitioning and an appropriate data layout.

Figure 4.12 shows an example for a two dimensional domain partitioning. The communication with "diagonal" neighbors could be avoided with a method described in [Moh00]. However, this would require that the communication is done consecutively for the horizontal and vertical direction which increases the total communication time. Therefore, this

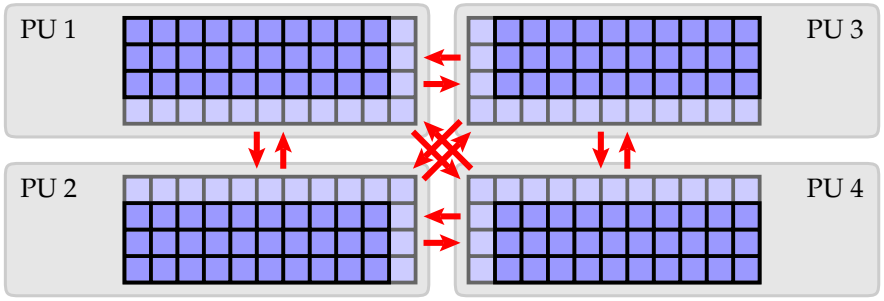


Figure 4.12: Two-dimensional domain partitioning involves more communication partners and requires a larger number of messages.

method has not been applied and only one dimensional partitioning is implemented in the FSLBM code.

4.5.2.2 Optimizing the Message Size

As stated in the previous section, the data of a message has to be merged to a contiguous block of memory before it can be sent. This task can either be done by the application code or be left to the MPI library. In any case, this process adds an additional load on both the CPU and the memory bus and increases the total runtime. After receiving a message, the reverse process has to be performed.

For the combination of one dimensional domain partitioning and the chosen data layout ZEYX (see Sec. 4.4) both packing and unpacking steps are redundant and can be skipped. The data of all cells which have to be copied to adjacent halo cells already lie contiguously in memory and can be sent directly from the original memory location as shown in Fig. 4.13. In contrast to the previous figures, each cube represents a single cell entry. For simplicity reasons, the number of entries per cell has been reduced from 29 to three entries, each displayed with a different color (red, green, and blue).

Since the x and y coordinates are the right-most array indices (fastest-moving indices), xy planes for a cell entry are located contiguously in memory. The next array index corresponds to the cell entry. Thus, a stack of three xy planes contains the entire data for all cells in a xy plane. With the remaining array index z, one of these layer triplets is selected. Since the data of all halo cells in neighboring subdomains is located contiguously in memory, it can be sent directly saving the time for composing the message data in a separate buffer.

The communication can be further optimized. In Fig. 4.13 all cell entries (i.e., all three layers) of the cells required for a halo update are selected for communication. However, during a sweep, only a subset of cell entries is altered, e.g., the sweep for calculating the surface normals only changes the position of the surface point and the normal vector which amounts to six cell entries. Two different ways of reducing the message size can therefore be applied:

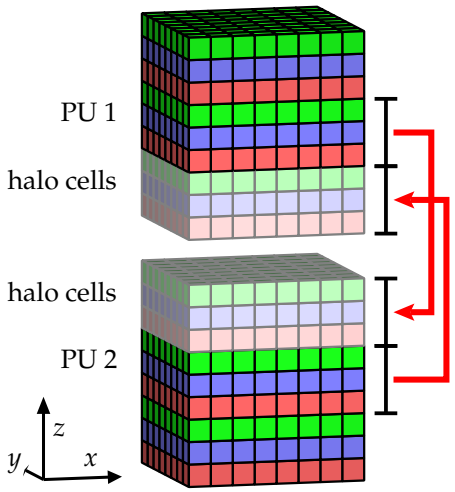


Figure 4.13: Memory layout for the ZEYX index ordering. Each color corresponds to one of the cell entries (for simplicity reasons reduced from 29 to 3). Contiguous blocks of memory can directly be sent to the halo cells (*pale cells*).

- Only the layers of altered cell entries are communicated (e.g., the green and the red cell layers). If several non-contiguous cell entries are involved, several messages have to be exchanged.
- A minimal range of cell entries is selected which encompasses all altered cell entries. The advantage of just one message per halo update is preserved, but unaltered cell entries might be contained in this range (e.g., the blue cells are also sent although only the green and red cells have been altered).

By optimizing the ordering of the cell entries, the latter variant could be implemented with only two redundant layers being sent in the six sweeps of a complete time step. Thus, the message size required for one time step could be reduced by 68%.

In Fig. 4.14 the performance of the set of rising bubbles setup is compared for communicating all cell entries and just the altered cells. The quotient of the parallelizable and the total runtime $\frac{p}{T_1} = \frac{p}{s+p}$ (see Eq. 4.2) can be used as an indicator about the scalability of a parallel code. The closer it is to 100%, the better the performance of the code scales with the number of employed CPUs. Since the message size mainly contributes to the sequential runtime s, the scalability factor increases from 96.5% to 98.7% with the reduction of the message size.

4.5.2.3 Hiding the Communication Overhead

In the basic parallel program flow as outlined in Alg. 4.4, the exchange of the halo data is performed after each sweep. The MPI routine `MPI_Sendrecv` both sends the boundary layer to neighboring PUs and receives the halo update. According to the MPI standard, this

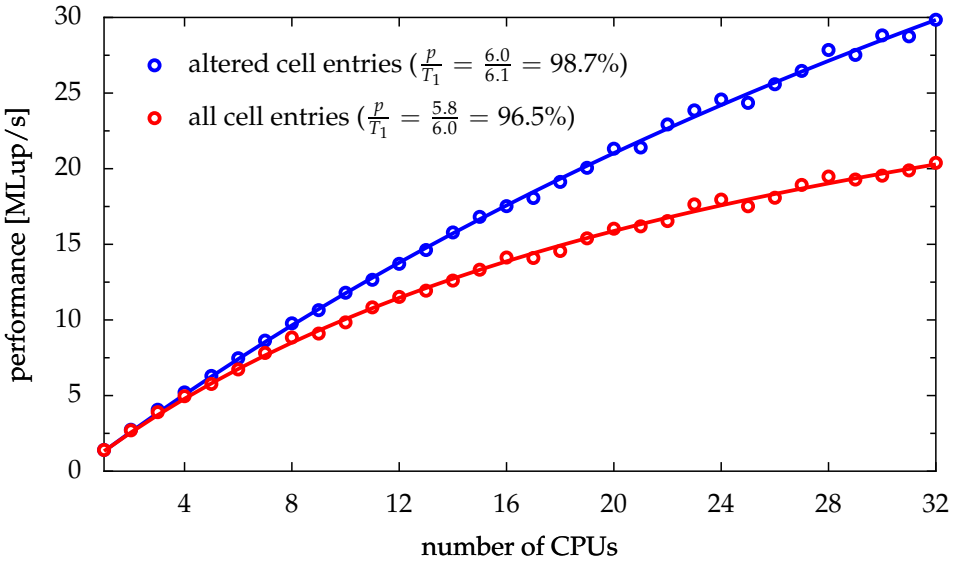


Figure 4.14: Comparison of the performance when communicating all cell entries or just the altered cell entries (set of rising bubbles setup).

routine is a blocking communication call, i.e., the completion of this routine guarantees that the communication has been completed. For simplicity, we do not distinguish between one and two communication partners and restrict the presentation to a single MPI call.

Algorithm 4.4 Basic parallel program flow.

```
1: for t ← t_start, t_end do                          ▷ time loop
2:     for s ← 1, N_sweep do                          ▷ sweep loop
3:         perform sweep s for entire subdomain
4:         MPI_Sendrecv                               ▷ update halos
5:     end for
6: end for
```

As we have seen, the influence of the sequential runtime s on the scalability of a code is significant. The sequential runtime should therefore be kept as low as possible. Communication has to be performed by all involved CPUs and therefore contributes to the sequential runtime. This so-called communication overhead has soon been identified as a potential subject for optimization by both scientists implementing simulation codes and the developers of communication libraries. One common way to mitigate the effect of the communication overhead is to "hide" it behind the computations of the simulation code. For this purpose, the MPI standard offers a set of so-called non-blocking communication calls which are denoted with the prefix "I" (immediate).

For hiding the time spent with communication, each sweep is divided into two parts: First, the boundary layers are updated. These can now be exchanged with the adjacent

Algorithm 4.5 Parallel program flow with hidden communication as suggested by the MPI standard.

```
1: for t ← t_start, t_end do                                   ▷ time loop
2:     for s ← 1, N_sweep do                                   ▷ sweep loop
3:         MPI_Irecv                                           ▷ initiate receiving
4:         perform sweep s for upper and lower cell layer
5:         MPI_Isend                                           ▷ initiate sending
6:         perform sweep s for the rest of the subdomain
7:         MPI_Waitall          ▷ wait for completion of receiving and sending
8:     end for
9: end for
```

subdomains while the sweep is completed for the rest of the subdomain. The altered program flow is shown in Alg. 4.5.

The influence of this modification strongly depends on the employed MPI implementation. The standard implementation used in this thesis, if not stated otherwise, is the commercial SCALI MPI. An older but still widely used open-source implementation is MPICH. The performance of both libraries for the standard implementation using blocking communication and for the implementation with hidden communication using non-blocking communication is compared in Fig. 4.15.

The SCALI library is highly optimized for the underlying InfiniBand network. The handling of the comparably small message sizes only marginally increases the runtime and the achievable improvement of the scalability by hiding the communication is therefore low. Interestingly, the performance with pure intra-node communication (one to four CPUs) does not deteriorate with the transition to inter-node communication (five and more CPUs).

For MPICH, only the results for the non-blocking variant are plotted, since the blocking variant exhibits virtually the same performance. Investigations show, that the communication only progresses inside of MPI calls. The time required for the completion of a communication is just shifted from the blocking MPI call `MPI_Sendrecv` to `MPI_Waitall`. This behavior is in accordance with the MPI standard which does not claim that the communication must be performed in a separate thread. Compared to the scalability of SCALI, MPICH performs significantly worse, since communication cannot be hidden by using non-blocking MPI calls. Furthermore, MPICH is not optimized for the InfiniBand network. As for SCALI, the performance of MPICH for intra-node and inter-node communication is comparable.

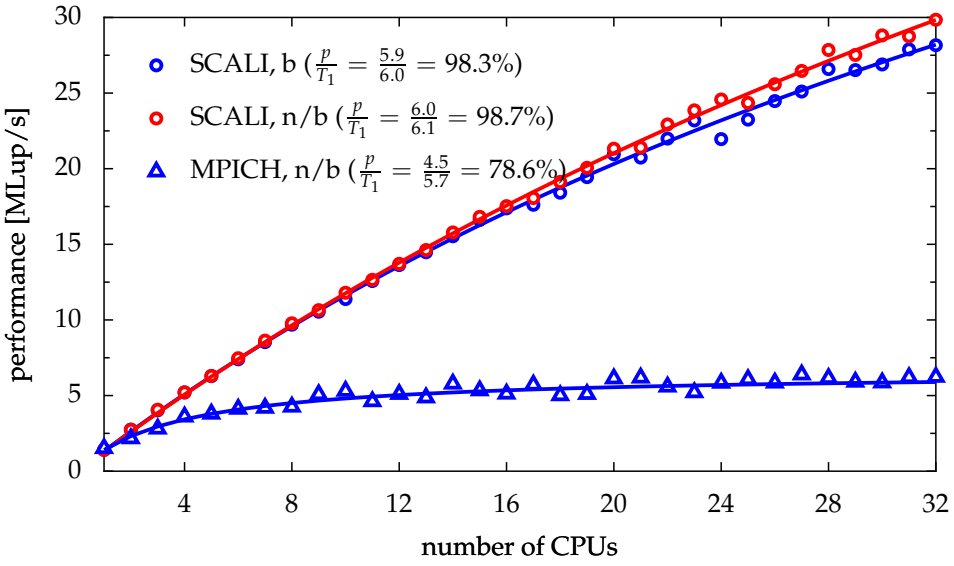


Figure 4.15: The performance of two different MPI implementations (SCALI MPI and MPICH) are compared for the standard implementation using blocking communication (b) and for the implementation with hidden communication using non-blocking communication (n/b).

4.5.2.4 Using Threads for Parallel Computing

In contrast to different processes running on a CPU, threads belonging to the same process share the same context. In particular, they all have access to the same memory space. Therefore, threads lend themselves well to distributing work on several PUs on shared-memory architectures [But97]. For this purpose, several computing threads are created which perform the computations for different parts of the computational domain, while the master thread is responsible for synchronizing the computing threads.

The restrictions for data access described in Sec. 3.3 do not only allow for an arbitrary ordering of a sequential cell update but also for parallel cell updates. The use of halo cells or explicit communication is not necessary in this case, since the data for the entire computational domain is logically stored as one block of memory as depicted in Fig. 4.16.

The lack of explicit communication makes this approach very attractive for parallel computing on shared-memory architectures. Yet the NUMA architecture of the most modern clusters adds some complexity to this method which is discussed below.

Thread Placement: The operating system running on each node is responsible for the scheduling, i.e., it decides which process or thread is running at a certain time on each of the PUs. By default, the operating system is free to move a thread from one PU to another. This is useful, since the programmer normally does not know the exact number of PUs and their current process load. For our purpose, however, it is vital to associate each

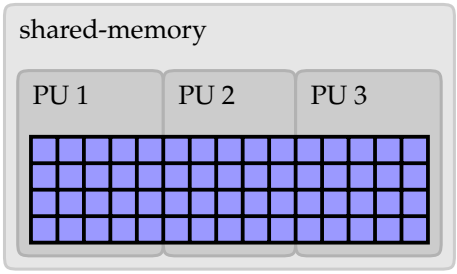


Figure 4.16: Parallelization for a shared-memory architecture. Each PU is assigned to some part of the computational domain. The use of halo cells and explicit communication is not necessary.

computational thread with exactly one PU, as we will see in the next paragraph about memory placement. This technique is known as thread pinning. For Linux, each thread can be pinned to a different PU by calling `sched_setaffinity`. Other operating systems offer a similar functionality.

Memory Placement: As described in Sec. 4.1.1 for the Opteron cluster, the memory of one cluster node is divided into four parts which are assigned to each PU. Processes or threads running on one PU are able to access data in any block of memory, but the access to non-local memory is slower. Therefore, care has to be taken that the majority of memory accesses refers to local memory.

For most operating systems, memory is organized in so-called memory pages [HP06]. A large block of memory which is contiguous from the programmer's and application's point of view, physically consists of several memory pages which are commonly 64 KB large. Each memory page can be located anywhere in memory or even on hard disk.

A common strategy for placing these memory pages is the so-called first touch page placement [TK98]: When a process or thread allocates memory, the decision where to physically place the data is deferred until the data is first written. This decision is made for each individual memory page and not for the entire block of memory. From a programmer's point of view, the procedure of allocating memory is split into two steps: First, one thread allocates the entire block of memory which will later be used by the ensemble of threads. Second, each thread initializes its part of memory by writing to it. Considering the PU where the process or thread is currently running, the operating system will try to map the data access to a local memory page. If the pool of free local memory pages is depleted, non-local memory pages are used instead.

Several modern operating systems offer the possibility of page migration, i.e., memory pages can be moved from the associated memory block of one PU to a different memory block without any change to the logical view of the memory. Linux supports page migration since release 2.6.1, but it does not automatically move memory pages from non-local

to local memory blocks if a thread is rescheduled to a different CPU. Therefore, the combination of thread pinning and allocation of local memory pages by the first touch page placement is essential for an optimal and reproducible computational performance.

If the threads are not pinned to one specific PU, identical simulation runs can deviate by up to 15% with respect to their overall runtime, because threads have been rescheduled to other PUs after the memory placement. This effect is not restricted to the use of threads, but was also observed for parallel simulation runs using MPI.

Synchronization of Threads: Care has to be taken that all threads complete their work for a certain sweep, before the next sweep can be started. This is realized by the use of so-called mutexes and condition variables [But97]. Although the general concept of thread synchronization is simple, the implementation is intricate and error-prone. A detailed description is beyond the scope of this thesis.

The resulting performance of this parallelization technique will be discussed in the next section where it will be compared with pure MPI and hybrid parallelization approaches.

4.5.2.5 Hybrid Parallelization using MPI and Threads

As we have seen, the parallelization with threads is attractive but limited to shared-memory architectures, such as, one node of a cluster computer. If more nodes should be used, a hybrid approach combining MPI and threads can be applied [Rab03]. In addition to synchronizing the computing threads, the master thread now also takes over the task of exchanging the halo cells with neighboring subdomains.

In this context, the difference of blocking and non-blocking communication vanishes, because the computations are done in separate threads and are therefore intrinsically overlapping with the communication performed in the master thread. A comparison of the performance for pure MPI and hybrid parallelization using SCALI and MPICH is shown in Fig. 4.17. Furthermore, the sequential and parallel computing time s and p for one time step as approximated by a least-squares fit with Amdahl's Law can be found in Tab. 4.3.

With the SCALI library, the introduction of threads slightly decreases the scalability. This has been expected, since SCALI internally uses threads to hide non-blocking communication calls. The additional usage of threads in the hybrid approach only adds some computational overhead without improving the runtime. Since the MPICH library does not use threads internally, it benefits significantly from the hybrid approach. Yet the performance is still not competitive to the performance with SCALI, because of a higher latency and lower bandwidth for communication.

The performance of the FSLBM implementation using only threads is equal to the parallel implementation using the SCALI library for up to four CPUs. It can be deduced that the SCALI implementation is able to hide any intra-node communication overhead.

	s [s]	p [s]	T_1 [s]	$\frac{p}{T_1}$
SCALI, pure MPI	0.08	5.98	6.06	98.7%
SCALI, hybrid	0.12	5.65	5.77	97.9%
MPICH, pure MPI	1.22	4.45	5.68	78.4%
MPICH, hybrid	0.69	5.11	5.80	88.1%
pure threads	0.08	5.61	5.69	98.7%

Table 4.3: Sequential s and parallelizable computing time p for one time step as approximated by a least-squares fit with Amdahl's Law (Eq. 4.4) for the set of rising bubbles setup.

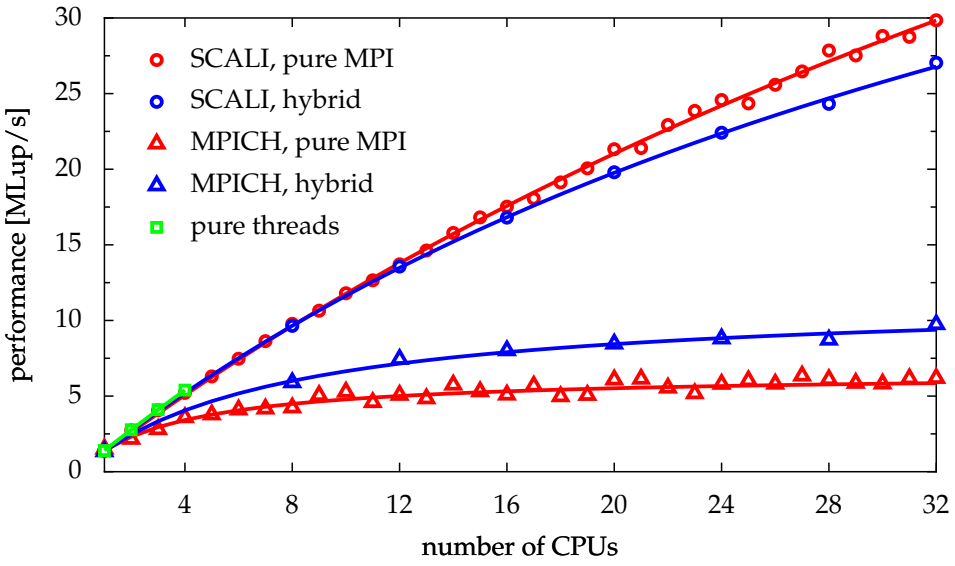


Figure 4.17: Comparison of the performance for pure MPI and hybrid parallelization using SCALI and MPICH (set of rising bubbles setup).

4.5.2.6 Dynamic Load Balancing

When using a domain partitioning method to distribute the workload on several PUs, a decision has to be made about the domain size for each PU. For a standard LBM implementation the required time to update a cell is constant in time and space. Therefore, an equal workload is achieved by assigning an equal amount of cells to each PU. Unfortunately, this does not hold for the LBM model with the free surface extension due to the different amount of time required for the update of different cell types. Furthermore, the cell types change in the course of a simulation because of the movements of the fluid. Accordingly, the workload shifts from one PU to another. Since a new sweep can only be started after all PUs have completed the current sweep and halos cells have been updated, the overall performance is dictated by the slowest PU.

Dynamic load balancing is able to compensate such load imbalances by redistributing

the load evenly on all PUs. Several different types of dynamic load balancing have been devised [XL97]. For the FSLBM implementation, the diffusive load balancing has been implemented. It is one representative of the so-called nearest-neighbor load balancing algorithms.

Its key features nicely fit the characteristics of the FSLBM implementation:

- If necessary, computational load is shifted from one PU to an adjacent PU. In contrast to a global load balancing scheme, where the new domain partitioning might be completely different, this local method is easier to implement and still fast enough to react on load changes due to the movement of the fluid which does not exceed $\frac{1}{10}$ cell per time step.
- In the diffusive load balancing approach, the decision for transferring load is only based on the load of neighboring PUs. Therefore, global communication can be avoided. The required load data can be sent piggyback with the data for the gas bubbles which will be described in the next section.

For implementing the diffusive load balancing method, the time required for a complete time step $t_i (1 \leq i \leq N)$ is measured for all N PUs in each time step. These values are exchanged with the neighboring PUs. Each PU i is now able to calculate

$$\begin{aligned} \delta t_i^{i-1} &= t_i - t_{i-1} \\ \delta t_i^{i+1} &= t_i - t_{i+1} \\ \delta t_i^{\min} &= \min(\delta t_i^{i-1}, \delta t_i^{i+1}) \\ \delta t_i^{\max} &= \max(\delta t_i^{i-1}, \delta t_i^{i+1}) \end{aligned}$$

with the missing values for the subdomain at both ends defined as $t_0 = t_1$ and $t_N = t_{N+1}$.

If the maximum load imbalance $\delta t_i^{\max}$ is larger than a threshold δt, one work unit is sent to the neighboring domain with the lower computational load depending on t_{i-1} and t_{i+1}. If the minimum load imbalance $\delta t_i^{\min}$ is smaller than $-\delta t$, one work unit is received from the neighboring domain with the higher computational load depending on t_{i-1} and t_{i+1}. In the unlikely event of equal loads for both neighbors, the decision is postponed until the next time step. This process guarantees that the number of work units does not increase for the PU with the highest load which is dominating the overall runtime.

In order to be able to receive additional layers of cells, all PUs allocate a predefined amount of additional memory. If these additional layers are depleted, no additional load can be accepted by the PU. Furthermore, each PU must host at least one layer of cells, since the method of halo exchanges would fail otherwise.

A work unit is the smallest amount of workload that can be shifted among PUs. The canonical and possibly easiest interpretation of a work unit is one layer of cells in the xy plane. The corresponding data is located contiguously in memory and can therefore be

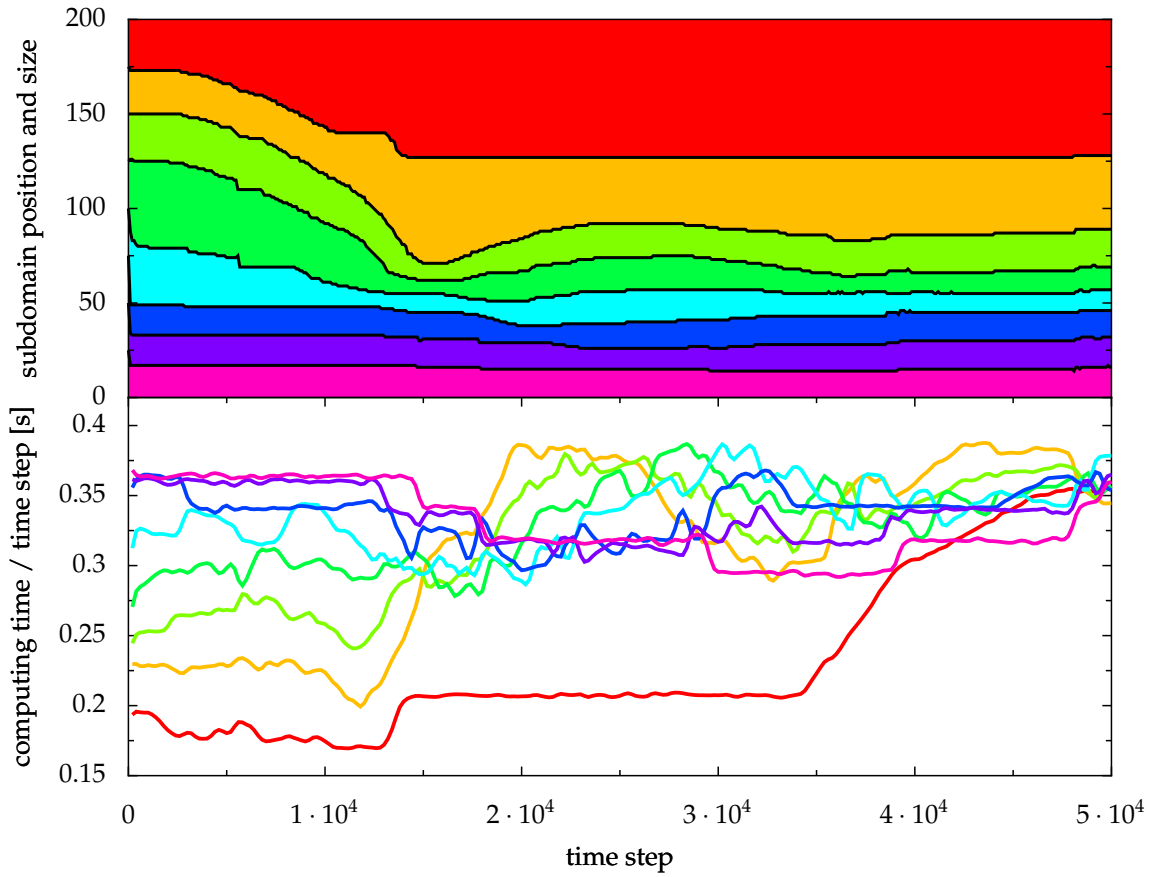


Figure 4.18: Evolution of the subdomain size and the computing time per time step for the falling drop setup using dynamic load balancing. Each color represents one of the eight CPUs.

transferred with a single MPI call. Using more than one layer of cells as a work unit is feasible, but also unnecessarily coarse with respect to the speed of load changes. For the FSLBM implementation, one layer of cells is used.

The threshold δt for sending or receiving a work unit is essential for the success of load balancing. If this value is chosen too low, slight oscillations in the computation time can trigger unnecessary load transactions which induce communication overhead and longer runtimes. With a larger threshold δt, the load balancing tolerates larger load imbalances which reduces the advantages of the method.

As an example, the falling drop setup has been calculated using the described diffusive load balancing method. The evolution of the subdomain sizes and the computing time per time step is depicted in Fig. 4.18. Each color corresponds to one of the CPUs. For simplicity reasons, only eight CPUs were used. In Figure 3.4, a snapshot of the setup is printed for every 10000 time steps.

Initially, all subdomains start with the same size. In the first few time steps, the green subdomain grows rapidly, since it is lying between the falling drop and the pool below and

contains mostly gas cells. While the drop is falling, the upper-most red subdomain has less work to do and grows. After about 15000 time steps, the drop has fallen into the pool. Since most fluid and interface cells are now located in the lower half of the domain, the six lower subdomain have the highest workload and therefore have a smaller size compared to the upper subdomains. The domain sizes mirror the movement of the fluid.

With the described method of load balancing, the performance for this setup could be improved from 10.7 to 15.4 MLup/s which is a significant increase of 44%. For the threshold δt, a value of 0.08 s was used, since the observed runtime oscillations were in the range 0.05 s. Other setups with a more homogeneous mass distribution still benefit from load balancing but show a smaller increase.

Currently, the combined use of load balancing and threads is not advisable. As described in Sec. 4.5.2.4, common operating systems do not offer the functionality of automatic page migration which would be necessary to shift load from one thread and one associated block of memory to another. With the increasing popularity of NUMA-based architectures, one can expect this functionality to be implemented in the near future.

4.5.3 Handling of Gas Bubble Data

For the sequential implementation, the handling of gas bubble data has been described in Sec. 3.2.2. The basic idea of the sequential code is to accumulate the volume changes for each bubble b in ΔV_b. The coalescence of bubbles is stored in the coalescence index M_b. After all sweeps in a time step have been applied, the volume changes are added to the bubble volumes stored in V_b (Eq. 3.1) and the coalescence of bubbles according to M_b is handled as described in Sec. 3.2.2.

For the parallel implementation, it has to be considered that a single bubble can be present in each subdomain. Furthermore, the bubble data only consists of two values: the initial and current bubble volume V_b^* and V_b. Even for a few hundred bubbles the data set is so small that it is feasible and most convenient to replicate this data for each PU.

During a time step, each PU proceeds exactly as the sequential implementation by accumulating volume changes in a local copy of ΔV_b and storing the coalescence of bubbles in a local copy of M_b. Each PU now has a separate set of data for ΔV_b and M_b. The merging of these data sets can be done in several different ways. Two different methods are implemented for the FSLBM code. Their general communication pattern is sketched in Fig. 4.19:

- Each PU sends its bubble update data to all other PUs. There, it is merged with the local bubble update data and applied to the bubble data. The communication pattern resembles an all-to-all communication.

- The left-most and right-most PU each start a “chain letter” by sending their bubble

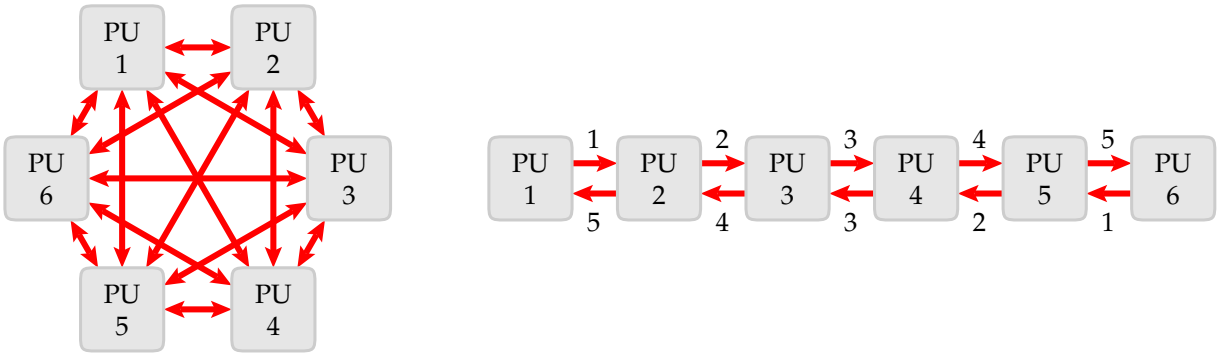


Figure 4.19: Two different methods for merging the bubble update data. Left: This global communication requires a large number of communications. Right: Less communications are necessary, but they have to be performed sequentially.

update data to their right and left neighbor, respectively. As soon as a PU receives a message either from its left or right neighbor, it adds its own bubble update data to the message and sends it to a neighbor preserving the direction of the message. When both "chain letters" have reached the other end of the line of PUs, all PUs have three packages of bubble update data: both packages traveling left and right and its own package. As the final step, all three packages are merged with the local bubble data.

A priori, it is not obvious which method performs better. Two aspects have to be considered: the computational effort for merging the bubble update data and the different communication patterns.

For N PUs, N packages of update data have to be merged using the first method. The second method requires only five of these data merging operations: two for preparing the messages and three for applying the three packages of update data. In general, the computational effort for both methods can be neglected even for several hundred gas bubbles compared to the time required for communication.

The second aspect are the different communication patterns. The first method requires $N(N-1)$ messages which can in principle be transferred in parallel. The second method only requires $2(N-1)$ messages. However, these messages cannot be sent simultaneously.

A comparison of the runtime for the set of rising bubbles plotted in Fig. 4.20 reveals only a minor difference. With an increasing number of CPUs, the advantage in performance of the second method grows up to 1.2% for 32 CPUs. The overall performance strongly depends on the underlying network performance and topology. In general, the second method should be preferred especially for larger number of CPUs.

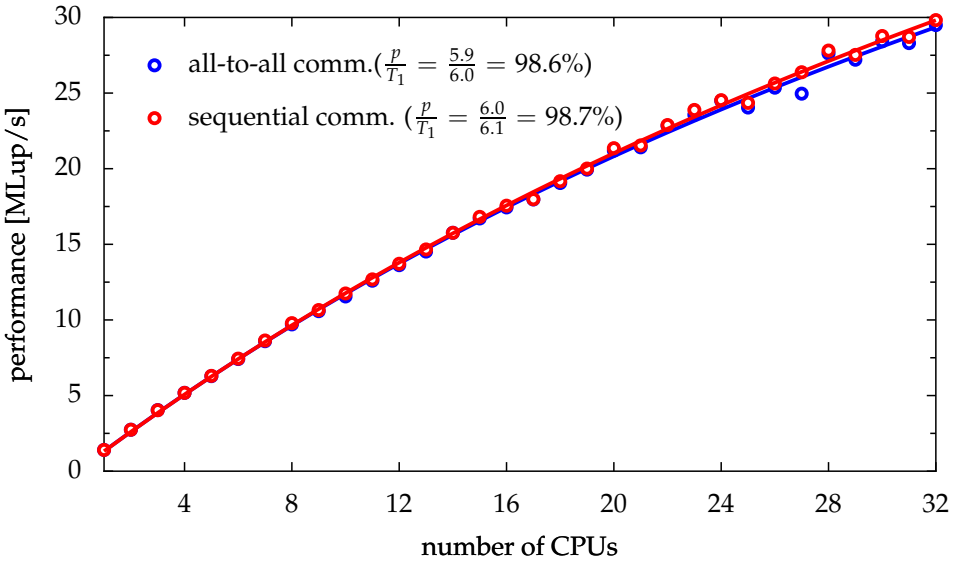


Figure 4.20: Comparison of the performance for the two communication patterns for the gas bubble data (set of rising bubbles setup).

4.5.4 Performance Results on the Hitachi SR8000-F1

The default method for intra-node parallelization on the Hitachi SR8000-F1 is to use the proprietary COMPAS library. It offers comparable features as the OpenMP library and can easily be used by adding so-called pragmas to the source code. These pragmas tell the compiler whether a certain code construct, such as, a loop can be computed in parallel on the eight available PowerPC CPUs of one node.

For the SPEC code, a single PowerPC of the Hitachi SR8000-F1 has a performance of about 2.5 MLup/s. Compared to modern commodity CPUs, this is about two to three times slower. Using COMPAS and MPI for intra-node and inter-node parallelization, respectively, the parallel performance of the SPEC code for various domain sizes was tested. The combination of using threads for the shared-memory and MPI for distributed-memory parallelization is very similar to the hybrid approach described in Sec. 4.5.2.5.

Due to the sophisticated network architecture, the Hitachi SR8000-F1 exhibits a good scaling performance for up to 512 CPUs as shown in Fig. 4.21. The largest domain size comprises about $1.1 \cdot 10^9$ cells which require 370 GB of memory in total. For 512 CPUs, still 75% of the scaled single-node performance can be achieved.

If the same combination of COMPAS and MPI is used for parallelizing the FSLBM code, the performance degrades dramatically as shown in Fig. 4.22. Two main reasons can be identified: Due to the more complicated data dependencies, the compiler is no longer able to distribute the computations on the eight CPUs using the COMPAS framework. This reduces the performance of one node virtually to the performance of a single PowerPC CPU. The extensive compiler log files support this assumption. The second reason for the perfor-

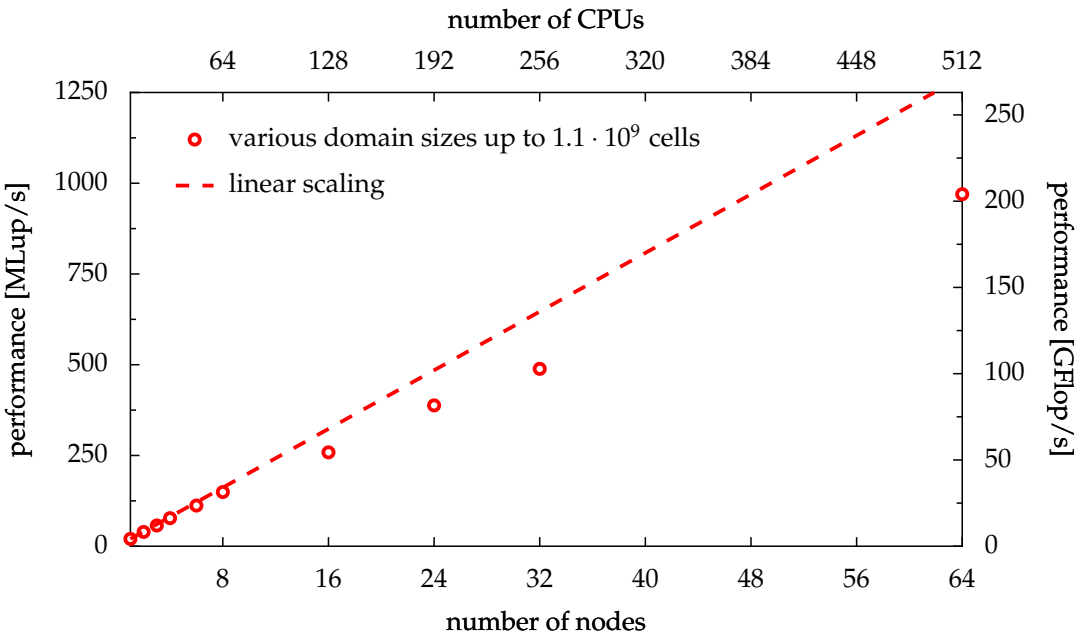


Figure 4.21: Parallel performance of the SPEC code on the Hitachi SR8000-F1 for various domain sizes up to $1.1 \cdot 10^9$ cells.

mance degradation is the higher number of conditional branches as already investigated in Sec. 4.3 discussing the instruction mix [PTD+04].

To circumvent the first problem, COMPAS was replaced by the explicit usage of threads as in the hybrid approach described in Sec. 4.5.2.5. Yet an incompatibility in the thread library of the Hitachi SR8000-F1 with the FSLBM code caused severe instabilities and impeded consistent and reproducible performance measurements. Instead, the performance of the FSLBM code using plain MPI parallelization is shown in Fig. 4.22. The performance is about eight times higher compared to the implementation using COMPAS, since all eight CPUs are now used again.

In general, the pseudo-vector architecture of the Hitachi SR8000-F1 does not lend itself well to the FSLBM which features complex data dependencies and a flow of execution which is hard to predict.

4.6 Debugging Techniques

In the context of HPC, the aspect of code debugging has to be reconsidered. Tools and methods which are commonly used for the development of standard software cannot be scaled to large-scale simulation codes running simultaneously on several hundreds of CPUs and handling terabytes of data:

- Debuggers with an intuitive graphical front-end like `ddd` are often unavailable on

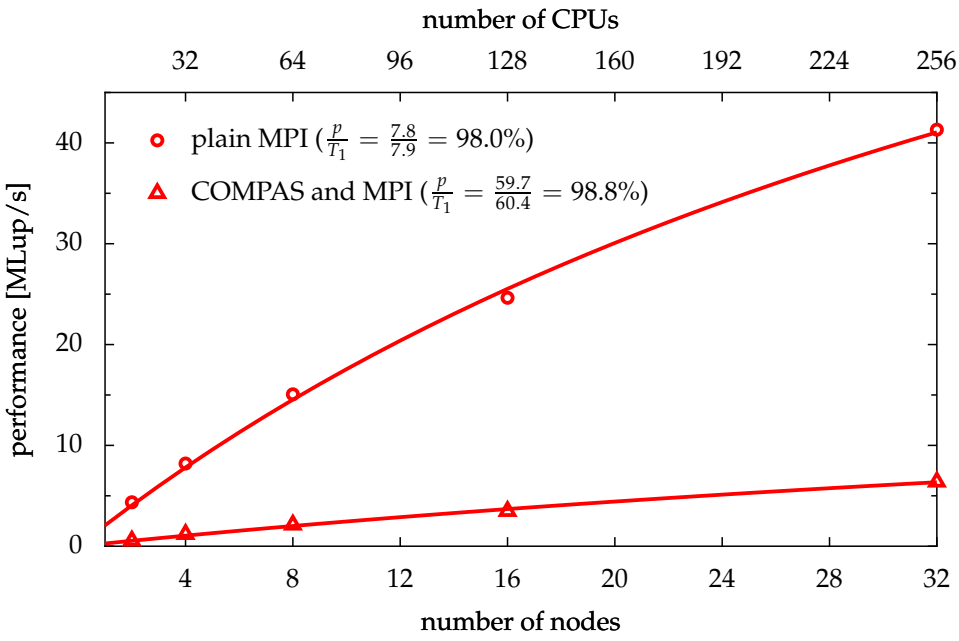


Figure 4.22: Comparison of the performance on the Hitachi SR8000-F1 using plain MPI and COMPAS parallelization (rising bubble setup with a domain size of $204 \times 204 \times 402$).

HPC architectures or cannot be used, because access to the system is limited to a text console, such as, the secure shell `ssh`.

- Console-based debuggers cannot be used to supervise several hundreds of CPUs involved in a single simulation run.
- Even dedicated HPC debuggers which can handle the number of involved CPUs, do not help to spot one erroneous data access among billions of correct ones.
- Storing the current simulation data to hard disk for further off-line investigations is hardly an option, because the required memory for large-scale simulations exceeds the user's hard disk quota.

In the following sections, several methods will be described which are implemented in the FSLBM simulation code. Neither of them is original in its idea, but the adaption to the requirements of HPC is essential.

4.6.1 Using Log Files

Still the most common way to debug a code during the development phase is to output messages about critical states of the code to a log file. For parallel applications, it is vital to know which process has issued a certain log message. For this purpose, a special logging

module automatically adds the MPI process number to each output. Furthermore, the messages from all tasks are collected in a single log file. Most MPI implementations do this automatically by redirecting the standard output and the error output stream to individual files, but the ordering of the messages in the log files might not correspond to the actual sequence of issue due to buffering of the output.

The typical runtime for a simulation code is in the order of a few hours up to several days. During the unattended runtime, a critical condition in the code can trigger the output of debugging messages in each iteration or time step. The resulting log file can grow rapidly and finally consume all available disk space or the assigned disk quota. To avoid this critical situation, the simulation stops automatically if a predefined number of log messages has been issued.

4.6.2 Checkpointing

An erroneous behavior of a code which has to be investigated, can occur after several hours of computation. For revisiting this critical behavior and for testing bug fixes, it is necessary to re-run the simulation several times. To avoid the long computation time before the critical time step is reached, so-called checkpointing [KT86] is implemented in the FSLBM code. Checkpointing means that the complete status of the code, i.e., the cell and the gas bubble data can be stored to hard disk after a certain number of time steps or in intervals. These checkpointing files can be loaded to continue the computation from the stored time step.

For parallel applications on distributed-memory architectures, the data can either be stored in parallel using functions provided by a MPI 2 library, or by having the processes write their data to a file sequentially. The latter is implemented for the FSLBM code. As stated before, the available disk space is often too small to hold the checkpointing data for large simulation setups. Therefore, the data is compressed on-the-fly using the `gzip` library. Since the cell data often features large domains with repetitive values, compression ratios up to 10:1 can be achieved.

4.6.3 Monitoring Data Access

Using pointers and arrays in C is very common but also quite error-prone, because boundary checks for the array indices are not performed by the run-time system. If the erroneous data access refers to a memory location which may not be accessed by the application, a segmentation fault is signaled and the application is halted. It is left to the programmer to find the location in the code which caused this error. If the erroneous data access happens to access an allowed memory location, the error is even harder to spot, since the code continues its computations, but might produce inconsistent results.

Algorithm 4.6 Using macros to select a debug or production run.

```
#define DEBUG 0

#if DEBUG == 1
#  define GET_VAL(x, y, z, e) getValue(x, y, z, e, __FILE__, __LINE__)
#else
#  define GET_VAL(x, y, z, e) data[z][e][y][x]
#endif

Entry data[SIZE_Z][29][SIZE_Y][SIZE_X];

void performSweep(void) {
  /* ... */
  v = GET_VAL(x, y, z, e);
  /* ... */
}
```

To detect such erroneous data accesses, each access to cell data could be encapsulated in function calls. This access function does a boundary check for the array indices and returns the requested cell data. For a write access, the function additionally checks if the value is within a certain range, depending on the type of state variable. For example, the distribution functions must not be set to negative or much larger values than 1.

The overhead induced by the use of such an access function prolongs the runtime of the FSLBM code by about a factor of four which is unacceptable for long simulation runs. Thus, each data access does not directly call the access function, but is implemented as a so-called macro. During compilation, the pre-processor of the compiler does a simple textual replacement of the macro with a predefined text. By changing this predefined text, the macro either directly accesses the data resulting in an undiminished performance, or calls the data access function as outlined in Alg. 4.6. The behavior of the `GET_VAL` macro is specified by the value of the `DEBUG` macro.

The call of the `getValue` function makes use of two other macros `__FILE__` and `__LINE__` which are automatically set by the pre-processor. They contain the file name of the source code and the line number, respectively. Thus, the `getValue` function can write a detailed error log specifying the source file and the line number of the erroneous data access.

The actual realization of this mechanism in the FSLBM code also encapsulates the type of data layout for the cell data. Switching from one data layout to another as done for the comparison in Sec. 4.4 only requires the change of a single value.

Furthermore, the data access restrictions described in Sec. 3.3 are also enforced using

this mechanism. For each sweep, it can be specified which state variables may be read or written. In debugging mode, the access function then checks the access to each state variable. If an access violates the current access settings, an error message is issued with the corresponding source file and line of code.

4.6.4 Online Data Visualization

The complex interactions of the different cell types in the course of a simulation run are difficult to investigate if the output is restricted to plain text. A more intuitive visualization of the computational domain with the possibility of direct interaction is realized as described below.

4.6.4.1 Communicating with the Simulation

For security reasons, the access to most HPC systems is limited to a textual connection based on the secure shell (`ssh`). Furthermore, only front-end nodes may be accessed directly. The computing nodes are normally blocked from direct access by the user. Using the special file system `sshfs`, it is possible to mount the remote file system of the HPC system on a local machine without any modifications to the `ssh` server of the remote site. Since the remote file system is also available on the computing nodes, the communication with the running simulation can be established using data files.

A special interactive mode of the simulation can be triggered in several ways. Most batch systems on HPC systems provide the command `qsig` to send a signal to a running code. If a certain signal is sent to the FSLBM code, it enters the interactive mode after completing the current time step. The same signal can be sent by the simulation itself when a critical situation occurs during the simulation run.

In the interactive mode, the simulation waits until it finds a predefined file in the working directory. The file can contain the coordinates of cells or commands to trigger different actions, such as, performing another time step or continuing the computation until another signal is raised. The process with the lowest index parses this file. If it finds cell coordinates, it writes all state variables for this cell into a predefined result file. If a different process hosts the cell data, the data is provided by this process.

4.6.4.2 Visualizing the Computational Domain

On the local machine a small script written in Python called EndoscoPy is used to interact with the simulation running on the HPC system and for visualizing the results as shown in Fig. 4.23.

To retrieve the cell information from the simulation waiting in interactive mode, the script creates a file with the required cell coordinates. The size and the position of the

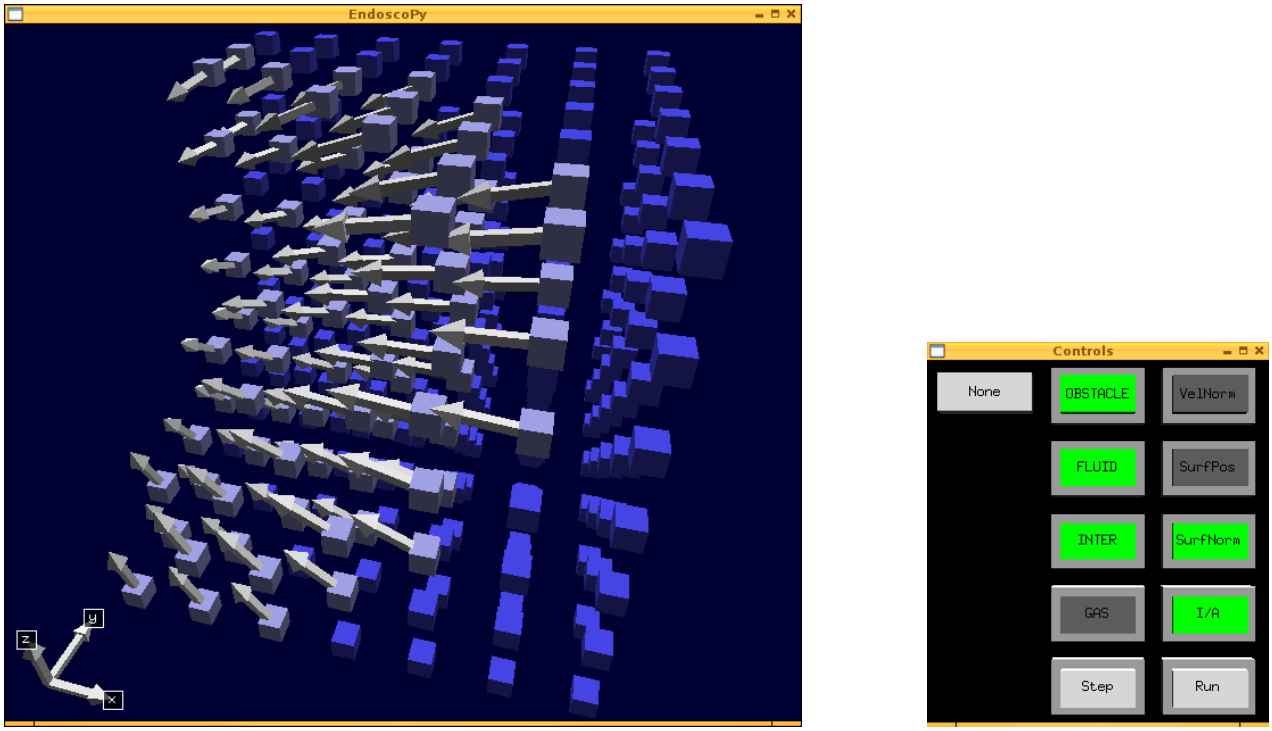


Figure 4.23: Screenshot of the EndoscoPy script. Only fluid cells (*dark blue cubes*) and interface cells (*light blue cubes*) are selected for display. In addition, the surface normals (*white arrows*) are shown. The right window contains the control elements.

displayed region can be altered using the keyboard. After receiving the answer in another file, the script displays this part of the computational domain. Each cell type can be turned on or off for visualization. Additionally, the fluid velocity, the surface points, and the surface normals can be displayed. By clicking on one of the cubes, all state variables of the corresponding cell are printed for further investigation. The cells can also be color shaded according to different state variables, such as, velocity or mass.

A considerable advantage of the EndoscoPy script is the possibility to inspect large-scale simulation runs which are distributed on many CPUs using a local computer with modest hardware requirements. By performing single time steps, the evolution of the simulation can be precisely studied. This feature was vital for the development of the simulation algorithms.

Chapter 5

Related Work

5.1 Contribution to the SPEC Benchmark Suite CPU2006

The performance of computer architectures is often measured using synthetic benchmark codes, such as, the LINPACK benchmark [DLP03]. While they are useful to investigate the performance of certain aspects of an architecture, it is difficult to predict the overall performance for scientific or application codes.

Another approach to measure performance is the use of real-life applications which are representative for certain fields of high performance computing such as computational fluid dynamics, molecular dynamics, or quantum chemistry. The benchmark suite compiled by the Standard Performance Evaluation Corporation[1] (SPEC) takes this approach. This group periodically releases a suite of real-life applications from both research and industry. A simplified version of the simulation code described in this thesis is included in the release of the so-called CPU2006 benchmark under the acronym `470.lbm`[2]. This benchmark implementation does not contain the free surface extensions, uses the BGK approximation for the collision operator, and can only be run sequentially.

In cooperation with the SPEC group, the `470.lbm` code was tested on more than 30 different computing platforms and various operating systems to ensure compatibility and consistency of the computed results. Phansalkar et al. studied the redundancy and the application balance in the SPEC CPU2006 benchmark suite [PJJ07]. According to a clustering of all 17 floating-point benchmark codes based on certain performance characteristics, the `470.lbm` code belongs to a representative subset of six benchmarks. It can therefore be expected that the `470.lbm` benchmark will have a significant influence on future compiler development.

[1] More information about the Standard Performance Evaluation Corporation can be found on their homepage `http://www.spec.org/`.

[2] Details about this benchmark are available on `http://www.spec.org/cpu2006/Docs/470.lbm.html`.

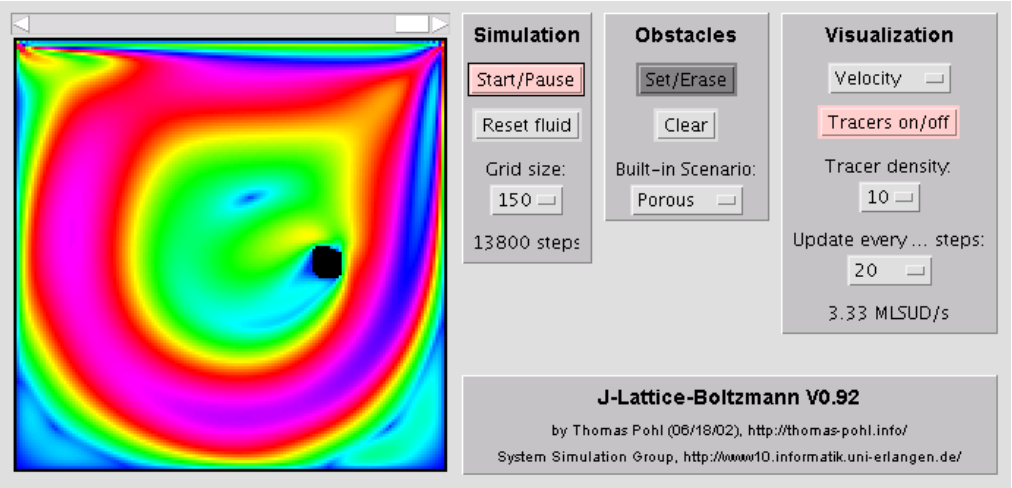


Figure 5.1: Screenshot of the interactive LBM implementation written in the Java programming language. The color corresponds to the velocity of the fluid.

5.2 Lattice Boltzmann Implementation in Java

5.2.1 Demonstration Code

For demonstration and educational purposes, an LBM simulation without the free surface extension has been implemented in the Java[3] programming language. The application is limited to a small computational domain in 2D, but can be run in a standard web browser if the Java runtime environment is installed[4].

As shown in the screenshot in Fig. 5.1, the application features various possibilities to visualize the computed results interactively. By clicking into the computational domain, obstacle cells can be added or removed. The simulation instantaneously reacts to the altered domain with an adapted flow field. The performance of the application reaches up to 5 MLup/s on modern CPUs. Several researchers in the field of fluid dynamics use this Java implementation for comparing and visualizing the results of their LBM implementations.

5.2.2 Integration into a Grid Computing Framework

De Sterck and Markel implemented a prototype framework for grid computing of scientific computing problems that require intertask communication [SMPR03]. The taskspaces framework is characterized by three major design choices:

- decentralization provided by an underlying tuple space concept,

[3]Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

[4]The Java LBM implementation is available on `http://thomas-pohl.info/work/lba.html`.

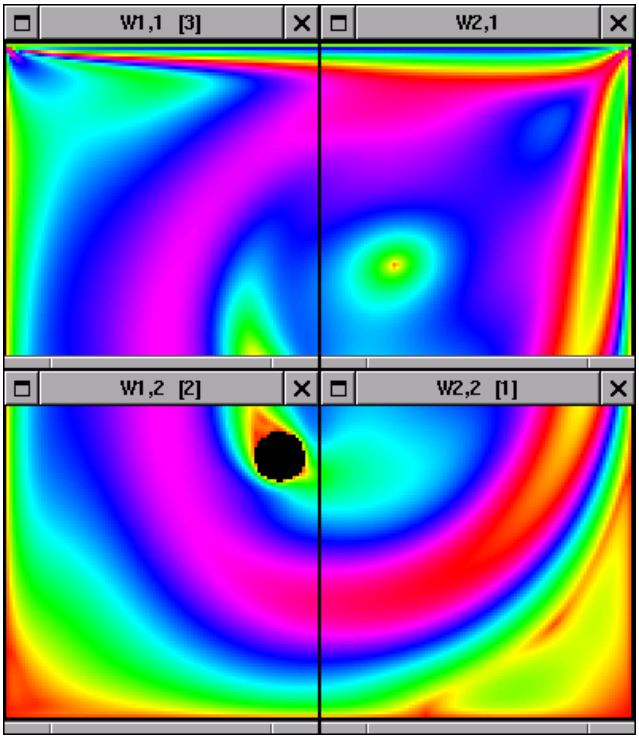


Figure 5.2: Parallel LBM implementation running on four different computers. The communication is based on the grid computing framework implemented by De Sterck and Markel [SMPR03]. The color corresponds to the velocity of the fluid.

- enhanced direct communication between tasks by means of a communication tuple space distributed over the worker hosts,
- object orientation and platform independence realized by implementation in Java.

The tuple space concept is similar to the coordination language Linda [CG89]. A distinctive feature of this grid computing framework is that instead of a working task which is tailored for just one specific application, a generic working task is started on all participating computers. The worker then downloads the Java byte code of the specific application and starts its computations.

As a proof of concept, the author of this thesis implemented a parallel LBM application, based on this grid computing framework and the LBM implementation described in the previous section. Figure 5.2 shows a parallel run of the LBM implementation on four computers. The computational domain can be divided into subdomains in two dimensions. Obstacles can be added or removed interactively and influence the flow field in all four subdomains.

Due to the early stage of development of the grid computing framework, the performance of the parallel implementation using four computers was lower than the sequential performance.

Chapter 6

Conclusions

Based on the 2D model for free surface flows by Thies et al., an extension of the model to 3D has been developed. For this purpose, several methods had to be replaced, because they could not be applied to a 3D computational domain. In particular the method to calculate the surface curvature required a major redesign. It is now based on a local triangulation of the surface points. This local mesh is then used as an input for a method described by Taubin [Tau95] to calculate the mean surface curvature.

This model and its implementation has been tested with several qualitative and a quantitative validation experiments. The qualitative setups exhibit the expected physical behavior. For the quantitative test, the setup of a rising bubble has been chosen as an experimentally and theoretically investigated and well-documented experiment. The results of the model for this test agree well with results from a reference simulation and also experimental results. Slight differences in the shape of the bubble lead to a deviation of the terminal rise velocity, which is about 7% lower than for the experimental result.

In the second part of this work, the model implementation has been analyzed and optimized in several different ways to improve the sequential and parallel performance. Based on the measured instruction mix, the huge gap in the performance on different high performance architectures can be explained. The flexible design of the implementation allowed for testing different data layouts. Hence, an optimal data layout has been found which achieves about twice the performance compared to the canonical data layout.

For standard lattice Boltzmann applications, further data access optimizations based on loop blocking have been developed. Furthermore, a method has been devised which almost halves the required space of memory. Due to the larger data set and a more complex computational kernel of the extended LBM implementation, no performance improvement could be achieved by loop blocking.

Two different methods have been implemented for parallel computing. For shared-memory architectures, the efficient use of threads has been demonstrated. The parallelization for distributed-memory architectures is based on the MPI standard. It has been shown

that a hybrid approach, using both threads and MPI for parallelization, can be beneficial for certain MPI libraries, which do not use threads internally for communication. By hiding the communication behind the computations, the good parallel scalability of the implementation can be demonstrated. As a final optimization, the efficient implementation of dynamic load balancing has been described. Depending on the simulation setup, the performance can be improved by up to 44%.

Finally, several debugging techniques, which have been adapted to the special requirements of high performance computing, have been presented.

Chapter 7

Future Work

7.1 Consideration of the Disjoining Pressure

An interesting result in the work of Thies [Thi05] was the influence of the so-called disjoining pressure on the formation of metal foams. The disjoining pressure is a repulsive force acting between two or more fluid/gas interfaces. With a decreasing distance of both interfaces, the force increases. For the investigated aluminum foams, this force presumably results from a network of oxides residing on the fluid/gas interface. Thies' simulations showed that the formation of aluminum metal foams strongly depends on the presence of this force.

To be able to simulate the formation of foams with the FSLBM implementation, it is necessary to add this force to the 3D model. The major problem of this extension is the range of the disjoining pressure. Although its effective range could be restricted to about ten cells without losing its physical meaning, its presence would still induce problems in the context of parallel computing.

The canonical way to solve this problem would require an increase of the number of halo layers at the borders of subdomains up to the desired range of the disjoining pressure. It would then be possible to calculate the distance from each interface cell to other interface cells belonging to a different gas bubble. However, the increased message size for a halo update would have a strong impact on the parallel performance of the implementation. In addition, the search for the nearest interface cell belonging to a different gas bubble significantly adds to this performance degradation.

A more promising approach is implemented prototypically in FSLBM, but has not yet been tested as thoroughly as the other features. This implementation of the disjoining pressure exploits the fact that the interface and therefore the position of the interface cells is moving slowly with a maximum speed of about $1/10$ cell per time step. Hence, it is possible to propagate the information about the distance to other interface cells in a diffusive manner. Interface cells act as the source for the distance information; by definition, they

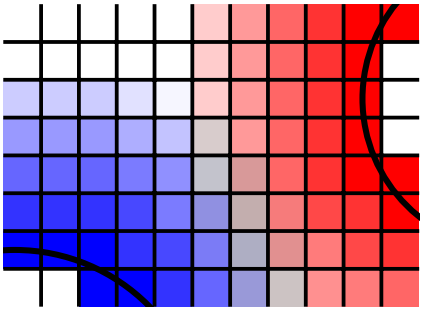


Figure 7.1: Distance map for two bubbles which are indicated by two arcs. The shades of color in each cell represent the distance to the closest interface cells. For each fluid cell, the two closest bubbles are stored.

have the distance zero to their associated bubble. In additional state variables, each fluid cell stores the index of the two closest bubbles and the approximate distances which are calculated as follows: During an additional sweep, every fluid cell searches for the smallest distance value among its direct neighbors. If this minimum distance plus the distance between both cells is smaller than the stored distance value, this value is saved as the new minimum distance value. This "diffusion" of the distance value can be stopped if it is larger than a certain cut-off distance. The procedure is also done for the second nearest bubble.

After a few time steps, a distance map has developed as depicted in Fig. 7.1. The two different bubbles are only indicated with two arcs. The shades of color of the fluid cells between both bubbles correspond to the distance to both bubbles. The resulting distances are only approximations to the shortest distance from a fluid cell to a bubble. Nevertheless, they might be used to calculate the force exerted by the disjoining pressure. This approach could be easily integrated into the framework of the FSLBM implementation. In the context of parallel computing, the same communication patterns can be used and a single halo layer is still sufficient.

7.2 Improving the Calculation of the Surface Curvature

The quantitative experiment with the setup of the rising bubble showed that the terminal rise velocity is very sensitive to the shape of the bubble. This shape, in turn, is strongly influenced by the algorithm that calculates the surface curvature. The largest improvement for the accuracy of the calculations can therefore be achieved by improving the approximation of the surface curvature.

The algorithm used in the FSLBM implementation is a local method which enables efficient parallel computing by requiring only a small neighborhood of cells. If the code is mainly used on shared-memory architectures, this advantage is of minor relevance and

other non-local approaches become applicable which can consider a larger number of neighboring cells.

7.3 Further Modifications to the Model

At the beginning of the development of the model and its implementation, one important criterion for the correctness of both was the conservation of symmetry, i.e., an initially symmetric setup must stay symmetric for the entire length of the simulation run.

In the course of this work, it turned out that this cannot be accomplished even if all applied methods theoretically conserve symmetry which is the case for the FSLBM implementation. The root cause for this unexpected behavior can be traced to the so-called numerical cancellation. An example is the summation of a number of floating-point values which is done in various places of the FSLBM implementation. Depending on the ordering of these values, the results at symmetric positions of the computational domain can differ by a small amount. This effect is well-known and results from the storing format of floating-point values [SK06].

Taking the effect of numerical cancellation into account, one could argue that the result should be symmetric up to some tolerance value ϵ. Yet even for comparably large values for ϵ, the results are not symmetric. The reason for this behavior is an inherent feature of the model: Based on the quasi-continuous value of the fluid fraction φ, discrete actions are triggered, such as, the conversion of cells. If the fluid fraction is close to the threshold for a cell conversion, tiny fluctuations cause the conversion at one location and prevent the same action at the symmetric position. It is obvious that different cell types automatically induce further deviations from the symmetric setup.

If a strict conservation of symmetry is not required, this behavior can be tolerated. The model tends to mitigate the observed asymmetries for all calculated setups.

It is not clear, how this inherent problem of the model can be solved. Instead of discrete events and decisions, based on the value of quasi-continuous values, the behavior of the cells could be coupled to these pivotal values to ensure a smooth transition from one state (e.g., cell type) to another. However, this would require a major redesign of the entire model.

Appendix A

Estimating the Curvature from a Triangulated Mesh in 3D

In contrast to the 2D case, where a single scalar is sufficient to entirely describe the curvature, the 3D case requires a tensor of curvature. This section presents the important parts of an algorithm described in [Tau95] to approximate the tensor of curvature from a triangulated mesh in 3D.

As a first step, we setup some definitions and important relations. The tensor of curvature is defined as the mapping $\boldsymbol{p} \mapsto \kappa_{\boldsymbol{p}}$ which assigns each point $\boldsymbol{p}$ of the surface S to the function that measures the directional curvature $\kappa_{\boldsymbol{p}}(\boldsymbol{T})$ of S at the point $\boldsymbol{p}$ in the direction of the unit length tangent vector $\boldsymbol{T}$ at $\boldsymbol{p}$. The directional curvature $\kappa_p(T)$ is defined by the identity $x''(0) = \kappa_{\boldsymbol{p}}(\boldsymbol{T})\boldsymbol{N}$, where $\boldsymbol{N}$ is the unit length normal vector of the surface S at the point $\boldsymbol{p}$, and $x(s)$ is a normal section to S at $\boldsymbol{p}$ parameterized by the arc length s. This normal section is defined to fulfill $x(0) = \boldsymbol{p}$ and $x'(0) = \boldsymbol{T}$.

According to [dC76, Tho79], the directional curvature function $\kappa_{\boldsymbol{p}}$ is a quadratic form which satisfies the identity

$$\kappa_{\boldsymbol{p}}(\boldsymbol{T}) = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}^t \begin{pmatrix} \kappa_{\boldsymbol{p}}^{11} & \kappa_{\boldsymbol{p}}^{12} \\ \kappa_{\boldsymbol{p}}^{21} & \kappa_{\boldsymbol{p}}^{22} \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \quad ,$$

where $\boldsymbol{T} = t_1\boldsymbol{T}_1 + t_2\boldsymbol{T}_2$ is a tangent vector to S at $\boldsymbol{p}$, and $\{\boldsymbol{T}_1, \boldsymbol{T}_2\}$ is an orthogonal basis of the tangent space to S at $\boldsymbol{p}$. The directional curvatures $\kappa_{\boldsymbol{p}}(\boldsymbol{T}_1)$ and $\kappa_{\boldsymbol{p}}(\boldsymbol{T}_2)$ are equal to the diagonal entries κ_{11} and κ_{22}, respectively. The vectors $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ are called principal directions of S at $\boldsymbol{p}$ if the off-diagonal entries κ_{12} and κ_{21} are both equal to zero. In this case, the corresponding directional curvatures are the principal curvatures, which we will denote as κ_1 and κ_2, instead of κ_{11} and κ_{22}.

Since the normal vector $\boldsymbol{N}$ is perpendicular to the principal directions $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$, we can construct an orthogonal basis $\{\boldsymbol{N}, \boldsymbol{T}_1, \boldsymbol{T}_2\}$ which depends on the point $\boldsymbol{p}$. The definition of

the directional curvature can be extended to arbitrary directions $\boldsymbol{T} = n\boldsymbol{N} + t_1\boldsymbol{T}_1 + t_2\boldsymbol{T}_2$ as

$$\kappa_{\boldsymbol{p}}(\boldsymbol{T}) = \begin{pmatrix} n \\ t_1 \\ t_2 \end{pmatrix}^t \begin{pmatrix} 0 & 0 & 0 \\ 0 & \kappa_{\boldsymbol{p}}^1 & 0 \\ 0 & 0 & \kappa_{\boldsymbol{p}}^2 \end{pmatrix} \begin{pmatrix} n \\ t_1 \\ t_2 \end{pmatrix} \quad .$$

For $n = 0$, $\boldsymbol{T}$ is again a tangent vector to S at $\boldsymbol{p}$. Transformed to an arbitrary system of orthogonal vectors $\{\boldsymbol{U}_1, \boldsymbol{U}_2, \boldsymbol{U}_3\}$, the directional curvature for a vector $\boldsymbol{T} = u_1\boldsymbol{U}_1 + u_2\boldsymbol{U}_2 + u_3\boldsymbol{U}_3$ is

$$\kappa_{\boldsymbol{P}}(T) = \boldsymbol{u}^t \, K_{\boldsymbol{p}} \, \boldsymbol{u} \quad , \tag{A.1}$$

where $u = \langle u_1, u_2, u_3 \rangle$, and $K_{\boldsymbol{p}}$ is a symmetric matrix with 0 and the two principal curvatures $\kappa_{\boldsymbol{p}}^1$ and $\kappa_{\boldsymbol{p}}^2$ as eigenvalues. Since the basis $\{\boldsymbol{U}_1, \boldsymbol{U}_2, \boldsymbol{U}_3\}$ can be chosen freely, we will set it to the same Cartesian coordinate system that is used to specify the coordinate of points on the surface, independently of the point $\boldsymbol{p}$ on S.

To obtain the principal curvatures and the principal directions from the matrix $K_{\boldsymbol{p}}$, it has to be restricted by a Householder Transformation [GL96] to the tangent plane $\langle \boldsymbol{N} \rangle^\perp$ of S at $\boldsymbol{p}$. The eigenvalues and the eigenvectors of the resulting 2×2 matrix are equal to the principal curvatures and the principal directions.

For approximating the directional curvature $\kappa_{\boldsymbol{p}}(\boldsymbol{T})$ for a unit length vector $\boldsymbol{T}$, tangent to S at $\boldsymbol{p}$, we consider a smooth curve $s(\boldsymbol{x})$ parameterized by arc length and contained in S such that $\boldsymbol{x}'(0) = \boldsymbol{T}$. This implies that $\boldsymbol{x}''(0) = \kappa_{\boldsymbol{p}}(\boldsymbol{T})\boldsymbol{N}$. The Taylor expansion of $\boldsymbol{x}(s)$ up to second order

$$\begin{aligned} \boldsymbol{x}(s) &= \boldsymbol{x}(0) + \boldsymbol{x}'(0)s + \frac{1}{2}\boldsymbol{x}''(0)s^2 + \mathcal{O}(s^3) \\ &= \boldsymbol{p} + \boldsymbol{T}s + \frac{1}{2}\kappa_{\boldsymbol{p}}(\boldsymbol{T})\boldsymbol{N}s^2 + \mathcal{O}(s^3) \end{aligned}$$

shows that

$$2\boldsymbol{N}^t \left(\boldsymbol{x}(s) - \boldsymbol{p}\right) = \kappa_{\boldsymbol{p}}(\boldsymbol{T})s^2 + \mathcal{O}(s^3)$$

and

$$\|\boldsymbol{x}(s) - \boldsymbol{p}\|^2 = s^2 + \mathcal{O}(s^3) \quad .$$

Dividing the previous two equations gives us

$$\frac{2\boldsymbol{N}^t \left(\boldsymbol{x}(s) - \boldsymbol{p}\right)}{\|\boldsymbol{x}(s) - \boldsymbol{p}\|^2} = \kappa_{\boldsymbol{p}}(\boldsymbol{T}) + \mathcal{O}(s) \quad .$$

The directional curvature is therefore equal to the limit

$$\kappa_{\boldsymbol{p}}(\boldsymbol{T}) = \lim_{s \to 0} \frac{2N^t \left(\boldsymbol{x}(s) - \boldsymbol{p}\right)}{\|\boldsymbol{x}(s) - \boldsymbol{p}\|^2} \quad . \tag{A.2}$$

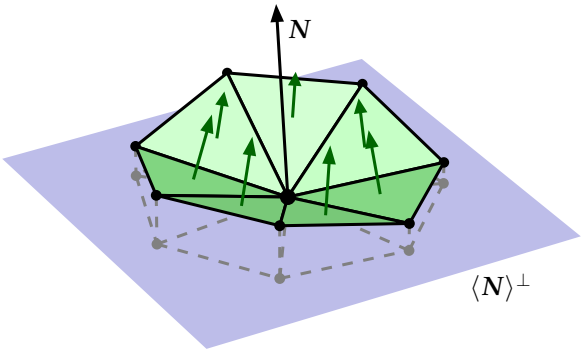


Figure A.1: The center point $\boldsymbol{p}$ is surrounded by a set of neighbor points P. Together, they form a set of faces F (*green triangles*). Each face f_k has a corresponding normal vector $\boldsymbol{N}_{f_k}$ (*green arrows*) which is used to calculate a weighted mean normal vector $\boldsymbol{N}$ (*black arrow*).

For what follows, we assume that we want to calculate the mean curvature for a central point $\boldsymbol{p}$ surrounded by a set of N_P neighboring points

$$P = \{\boldsymbol{p}_i : 1 \leq i \leq N_P\} \quad .$$

The central point $\boldsymbol{p}$ together with the neighboring points define a set of faces

$$F = \left\{ f_k : f_k = \left(\boldsymbol{p}_k, \boldsymbol{p}_{(k+1) \bmod N_P}, \boldsymbol{p} \right) \right\}$$

with $|f_k|$ being the surface area of the face f_k. The unit length normal vector for each face is denoted as $\boldsymbol{N}_{f_k}$. The geometric situations is depicted in Fig. A.1.

First, we define the normal vector at the point $\boldsymbol{p}$ as a weighted sum of the normals of the set of faces F

$$\boldsymbol{N}' = \sum_{f_k \in F} |f_k| \boldsymbol{N}_{f_k} \quad ; \quad \boldsymbol{N} = \frac{\boldsymbol{N}'}{\|\boldsymbol{N}'\|} \quad .$$

In the next step, we calculate an approximation M for the matrix K from Eq. A.1

$$M = \sum_{\boldsymbol{p}_i \in P} \lambda_i \kappa_i \boldsymbol{T}_i \boldsymbol{T}_i^t \quad .$$

It is shown in [Tau95] that the matrices M and K have the same eigenvectors. $\boldsymbol{T}_i$ is the normalized projection of the vector $\boldsymbol{p}_i - \boldsymbol{p}$ onto the tangent plane $\langle \boldsymbol{N} \rangle^\perp$:

$$\boldsymbol{T}_i' = \left(I - \boldsymbol{N}\boldsymbol{N}^t \right) (\boldsymbol{p}_i - \boldsymbol{p}) \quad ; \quad \boldsymbol{T}_i = \frac{\boldsymbol{T}_i'}{\|\boldsymbol{T}_i'\|}$$

The directional curvature $\boldsymbol{T}_i$ is approximated using Eq. A.2 under the assumption that the two points $\boldsymbol{p}_i$ and $\boldsymbol{p}$ are positioned sufficiently close to each other:

$$\kappa_i = \frac{2\boldsymbol{N}^t (\boldsymbol{p}_i - \boldsymbol{p})}{\|\boldsymbol{p}_i - \boldsymbol{p}\|^2} \quad .$$

The weighting factor λ_i is proportional to the sum of the surface areas of the two triangles that contain the point $\boldsymbol{p}_i$.

$$\lambda_i' = \left|f_{(i-1+N_P) \bmod N_P}\right| + \left|f_i\right| \quad ; \quad \lambda_i = \frac{\lambda_i'}{\sum_{\boldsymbol{p}_i \in P} \lambda_i'} \quad \Rightarrow \quad \sum_{\boldsymbol{p}_i \in P} \lambda_i = 1$$

If we were to calculate the individual eigenvalues and eigenvectors, we would have to follow the procedure described in [Tau95]. For our purpose, which is the calculation of the surface tension, we are only interested in the mean curvature $\kappa = \frac{1}{2}(\kappa_1 + \kappa_2)$. This simply is the trace of the matrix M:

$$\kappa = \mathrm{tr}(M)$$

Compared to the procedure in [Tau95] which requires the use of a Householder transformation, the computational effort for calculating the trace of M is significantly lower.

Appendix B

German Parts

B.1 Inhaltsverzeichnis

B.2 Zusammenfassung

Die anhaltende Steigerung der Rechenleistung eröffnet der Disziplin der Computersimulation immere komplexere Anwendungsbereiche. Ein Beispiel dafür ist die Simulation von Strömungen mit freien Oberflächen. Thies, Körner et al. haben ein Modell vorgeschlagen, um solche Strömungen im Rahmen der Materialwissenschaften zu simulieren. Ihr Modell ist jedoch auf zwei Raumdimensionen beschränkt. Ein Hauptthema der hier vorgestellten Arbeit, ist die Erweiterung ihres Modells auf 3D. Die verschiedenen Modifikationen an den Algorithmen werden detailliert beschrieben. Insbesondere die Berechnung der Oberflächenkrümmung musste durch einen völlig neuen Ansatz ausgetauscht werden, der auf der lokalen Triangulierung der Flüssig-Gas-Grenzfläche basiert.

Das zweite Hauptthema dieser Arbeit sind verschiedenste Aspekte des Hochleistungsrechnens, die anhand der 3D-Modellimplementierung diskutiert werden. Ein Beispiel dafür ist die Optimierung der räumlichen und temporalen Datenlokalität, was auf Grund des komplexen Modells schwierig ist. Darüberhinaus werden verschiedene Optimierungsstrategien zur parallelen Berechnung beleuchtet. Die Anwendung der vorgestellten Leistungsoptimierungen ist nicht auf die spezifische Modellimplementierung beschränkt, sondern kann auf ein breites Spektrum komplexer Simulationsprogramme verallgemeinert werden.

B.3 Einleitung

B.3.1 Umfang der Arbeit

Die hier vorgestellte Arbeit ist in drei Bereiche unterteilt:

- Erweiterung des Modells zur Simulation von Strömungen mit freien Oberflächen basierend auf der Arbeit von Thies, Körner et al. [Thi05, KTH+05].

- Entwicklung einer effizienten parallelen Implementierung dieses erweiterten Modells.

- Analyse verschiedener Aspekte des Hochleistungsrechnens dieser Implementierung.

Der erste Teil der Arbeit liegt im Gebiet der numerischen Strömungssimulation und der Physik, während der zweite Teil hauptsächlich im Bereich der Informatik angesiedelt ist. Der Aspekt des Hochleistungsrechnens ist ein zentrales Thema der gesamten Arbeit und wird im letzten Teil detailliert beleuchtet.

B.3.2 Motivation

Die numerische Strömungsmechanik hat sich zu einem etablierten Forschungszweig entwickelt. Aufgrund ihrer praktischen Relevanz und der breiten Anwendbarkeit wurden verschiedenste Methoden entwickelt, um die zugrunde liegenden Navier-Stokes-Gleichungen zu lösen. 1986 legten Frisch, Hasslacher und Pomeau den Grundstein für eine neue Klasse von Lösungsmethoden, die zellulären Automaten für "Gitter-Gas" [FHP86]. Im makroskopischen Grenzfall bildet dieses dem Billard-Spiel ähnlichen Verfahren die Navier-Stokes-Gleichungen nach, ohne Fließkomma-Arithmetik zu verwenden. Die Nachteile dieser Methodik waren der Anlass für die Entwicklung der Lattice-Boltzmann-Verfahren, die sich unter anderem durch eine höhere Flexibilität von ihrem Vorgänger auszeichnen. In den folgenden Jahren fand dieser Ansatz immer weitere Verbreitung in den verschiedensten Anwendungsbereichen wie z.B. der Magnetohydrodynamik und den Mehrphasenströmungen.

Zwei Charakteristika der Lattice-Boltzmann-Methode machen dieses Verfahren sowohl für Wissenschaftler als auch Anwender besonders attraktiv: Zuallererst ist das grundlegende Modell vergleichsweise einfach zu implementieren. Noch wichtiger allerdings ist, dass dieses Verfahren sich einfach erweitern lässt, beispielsweise um freie Oberflächen zu simulieren. Thies, Körner et al. entwickelten eine solche Erweiterung zur Simulation des Verhaltens von sog. Metallschäumen [Thi05, KTH$^+$05]. Einfach ausgedrückt verhalten sich solche Schäume ähnlich wie Seifenschaum. Die komplexe Struktur und die sich ständig ändernde Geometrie solcher Schäume sind ein ideales Anwendungsszenario für die Lattice-Boltzmann-Methode. Das Modell von Thies et al. ermöglicht beispiellose Einblicke in die Rheologie und Morphologie solcher Metallschäume, ist allerdings auf zwei Raumdimensionen beschränkt.

Bei der Erweiterung den Modells auf drei Raumdimensionen wird klar, dass ein Teil der verwendeten Algorithmen nicht mehr anwendbar sind. Daher besteht der erste Teil der hier vorgestellten Arbeit darin, alternative Methoden anzuwenden um die Simulation in drei Raumdimensionen zu ermöglichen.

Ein weiteres Problem der ursprünglichen Implementierung ist, dass der größere Rechenaufwand in 3D die Berechnungsdauer inakzeptabel erhöhen würde. Aus dem Bereich des Hochleistungsrechnens sind verschiedenste Techniken bekannt, um die Rechenleistung eines Programms zu verbessern, wie z.B. die Optimierung des Datenzugriffs oder paralleles Rechnen. Allerdings bleibt die Anwendung und Untersuchung solcher Methoden oftmals auf vergleichsweise simple Berechnungsroutinen beschränkt. Daher ist der Transfer dieser Techniken auf solch einen komplexen Berechnungskern die zweite große Herausforderung dieser Arbeit.

B.3.3 Gliederung

Kapitel 2 beschreibt das Modell zur Simulation von Strömungen mit freien Oberflächen basierend auf der Lattice-Boltzmann-Methode. Es werden sowohl das zugrunde liegende Modell von Thies, Körner et al., als auch die Erweiterung auf 3D und die dazu notwendigen Modifikation, die im Rahmen dieser Arbeit entwickelt wurden, dargestellt.

In Kapitel 3 wird die Implementierung des Modells skizziert. Die Modelldaten und der grundlegende sequentielle Ablauf des Simulationscodes werden erläutert. Es folgt eine Beschreibung von drei Simulationsszenarien, die in den folgenden Kapiteln häufige Verwendung finden. Zusätzlich zu diesen qualitativen Tests, wird abschließend ein quantitativer Test detailliert dargestellt.

In Kapitel 4 werden verschiedenste Aspekte des Hochleistungsrechnens anhand der Modellimplementierung beleuchtet. Dabei sind die zwei Hauptthemen die Optimierung des Datenzugriffs und die Realisierung der parallelen Verarbeitung.

Zwei verwandte Arbeitsbereiche werden in Kapitel 5 beschrieben: Ein Vorgänger der hier vorgestellten Implementierung, ohne die Erweiterung zur Behandlung freier Oberflächen, wurde in die SPEC-Benchmark-Sammlung CPU2006 aufgenommen. Das zweite Thema ist die Integration einer LBM-Implementierung in ein Grid-Computing-Gerüst basierend auf der Programmiersprache Java.

Abschließend werden in Kapitel 6 die Folgerungen aus dieser Arbeit präsentiert. Ein Ausblick in Kapitel 7 zeigt weitere Entwicklungs- und Verbesserungsmöglichkeiten des Modells und dessen Implementierung auf.

Appendix C

Curriculum Vitae

General Information

Date of Birth	March 18, 1974
Place of Birth	Füssen, Germany
eMail	`thomas.pohl@gmail.com`
WWW	`http://thomas-pohl.info/`

Education

11/2000 – 2006	PhD student, System Simulation Group (Informatik 10), Department of Computer Science, University of Erlangen-Nuremberg, Germany
11/1994 – 08/2000	Student, University of Augsburg, Germany, major: physics, minor: computer science, 08/10/2000: graduation as *Diplom-Physiker (Dipl.-Phys.)*
07/1993	*Allgemeine Hochschulreife (Abitur)*

Employment

12/2006 –	Siemens AG, Siemens Medical Solutions, Forchheim, Germany
11/2005 – 10/2006	Regional Computing Center (RRZE), University of Erlangen-Nuremberg, Germany
12/2000 – 07/2005	Research Assistant, System Simulation Group (Informatik 10), Department of Computer Science, University of Erlangen-Nuremberg, Germany

Index

Bibliography

[AK01] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2001.

[Amd67] G.M. Amdahl. Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of the AFIPS Conference*, volume 30, pages 483–485, April 1967.

[BDQ98] F. Bassetti, K. Davis, and D. Quinlan. Temporal Locality Optimizations for Stencil Operations within Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures. In *Proceedings of the Int. Conf. on Parallel and Distributed Computing and Systems*, pages 145–153, October 1998.

[BG00] J.M. Buick and C.A. Greated. Gravity in a Lattice Boltzmann Model. *Phys. Rev. E (3)*, 61(5):5307–5320, May 2000.

[BGD05] V. Buwa, D. Gerlach, and F. Durst. Regimes of Bubble Formation on Submerged Orifices. *Phys. Rev. Lett.*, April 2005.

[BGK54] P. Bhatnagar, E.P. Gross, and M.K. Krook. A Model for Collision Processes in Gases - I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Phys. Rev. Lett.*, 94(3):511–525, 1954.

[BGK96] D. Burger, J.R. Goodman, and A. Kägi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd Annual Int. symposium on Computer Architecture*, pages 78–89, 1996.

[Bol72] L. Boltzmann. Weitere Studien über das Wärmegleichgewicht unter Gasmolekülen. *Sitzungsber. Akad. Wiss. Wien*, 66:275–370, 1872.

[BSM05] I.N. Bronstein, K.A. Semendjajew, and G. Musiol. *Taschenbuch der Mathematik*. Verlag Harry Deutsch, 2005.

[But97] D.R. Butenhof. *Programming with POSIX Threads*. Professional Computing Series. Addison-Wesley, May 1997.

[CG89] N. Carriero and D. Gelernter. Linda in Context. *Comm. ACM*, 32(4):444–458, April 1989.

[CL93] B. Cabral and L.C. Leedom. Imaging Vector Fields Using Line Integral Convolution. In *Proceedings of SIGGRAPH '93*, pages 263–270, 1993.

[CS92] X. Chen and F. Schmitt. Intrinsic Surface Properties from Surface Triangulation. In *Proceedings of the European Conference on Computer Vision*, pages 739–743, May 1992.

[dC76] M.P. do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[dGK+02] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.-S. Luo. Multiple-Relaxation-Time Lattice Boltzmann Models in Three Dimensions. *Philos. Trans. Roy. Soc. London Ser. A*, 360(1792):437–451, March 2002.

[DLP03] J.J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present and Future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, July 2003.

[DMV87] J.J. Dongarra, J. Martin, and J. Vorlton. Computer Benchmarking: Paths and Pitfalls. *IEEE Spectrum*, 24(7):38–43, July 1987.

[Don04] S. Donath. On Optimized Implementations of the Lattice Boltzmann Method on Contemporary Architectures. Bachelor thesis, Universität Erlangen, Germany, 2004.

[FHP86] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-Gas Automata for the Navier-Stokes Equation. *Phys. Rev. Lett.*, 56(14):1505–1508, 1986.

[GA94] I. Ginzburg and P.M. Adler. Boundary Flow Condition Analysis for the Three-Dimensional Lattice Boltzmann Model. *J. Physique II*, 4:191–214, 1994.

[GH01] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM Publ., 2001.

[Gib70] J.C. Gibson. The Gibson Mix. Technical Report 00.2043, IBM Systems Development Division, Poughkeepsie, NY, 1970.

[Gin05] I. Ginzburg. Generic Boundary Conditions for Lattice Boltzmann Models and their Application to Advection and Anisotropic Dispersion Equations. *Adv. in Water Res.*, 28(11):1196–1216, November 2005.

[GL96] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Press, Baltimore, 1996.

[GS02] I. Ginzburg and K. Steiner. A Free Surface Lattice Boltzmann Method for Modelling the Filling of Expanding Cavities by Bingham Fluids. *Philos. Trans. Roy. Soc. London Ser. A*, 360:453–466, 2002.

[GS03] I. Ginzburg and K. Steiner. Lattice Boltzmann Model for Free-Surface Flow and its Application to Filling Process in Casting. *J. Comput. Phys.*, 185:61–99, 2003.

[Han98] J. Handy. *The Cache Memory Book*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, February 1998.

[Har04] S. Harris. *An Introduction to the Theory of the Boltzmann Equation*. Dover Publications, 2004.

[HB76] J.G. Hnat and J.D. Buckmaster. Spherical Cap Bubbles and Skirt Formation. *Phys. Fluids*, 19(2):182–194, February 1976.

[HM04] H. Huang and P. Meakin. Volume-Of-Fluid Simulation of Two-Phase Incompressible Fluid Flow Through Interconnected Pores and Fracture Networks. *AGU Fall Meeting Abstracts*, December 2004.

[HP06] J.H. Hennessy and D.A. Patterson. *Computer Architecture - A Quantitative Approach*. Academic Press, 2006.

[HZLD97] X. He, Q. Zou, L.-S. Luo, and M. Dembo. Analytic Solutions and Analysis on Non-Slip Boundary Condition for the Lattice Boltzmann BGK Model. *J. Statist. Phys.*, 87:115–136, 1997.

[Igl03] K. Iglberger. Performance Analysis and Optimization of the Lattice Boltzmann Method in 3D. Bachelor thesis, Universität Erlangen, Germany, 2003.

[Int02] Intel Corporation. *Intel Itanium 2 Processor Reference Manual*, June 2002. Document Number: 251110-001.

[IYO95] T. Inamuro, M. Yoshino, and F. Ogino. A Non-Slip Boundary Condition for Lattice Boltzmann Simulations. *Phys. Fluids*, 7(12):2928–2930, 1995. Erratum 8(4):1124, 1996.

[KPR+06] C. Körner, T. Pohl, U. Rüde, N. Thürey, and T. Zeiser. *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, chapter Parallel Lattice Boltzmann Methods for CFD Applications, pages 439–465. Springer-Verlag, 2006.

[KT86] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery in Distributed Systems. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, pages 1150–1158. IEEE Press, 1986.

[KTH+05] C. Körner, M. Thies, T. Hofmann, N. Thürey, and U. Rüde. Lattice Boltzmann Model for Free Surface Flow for Modeling Foaming. *J. Statist. Phys.*, 121(1):179–196, October 2005.

[KW03] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In P. Sanders, U. Meyer, and J. Sibeyn, editors, *Proceedings of the GI-Dagstuhl Forschungsseminar on Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, pages 213–232. Springer-Verlag, 2003.

[LC87] W. Lorenson and H. Cline. Marching Cubes: A High Resolution 3D Surface Reconstruction Algorithm. In *Proceedings of SIGGRAPH*, pages 163–169, 1987.

[LP82] C. Lin and M.J. Perry. Shape Description Using Surface Triangulation. In *Proceedings of the IEEE Workshop on Computer Vision: Representation and Control*, pages 38–43, 1982.

[MBF92] O. Monga, S. Benayoun, and O. Faugeras. From Partial Derivatives of 3D Density Images to Ridge Lines. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 354–359, 1992.

[Moh00] M. Mohr. Low Communication Parallel Multigrid: A Fine Level Approach. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Proceedings of the 6th International Euro-Par Conference*, volume 1900 of *Lecture Notes in Computer Science*, pages 806–814. Springer-Verlag, 2000.

[MSM05] T.G. Mattson, B.A. Sanders, and B.L. Massingill. *Patterns for Parallel Computing*. Addison-Wesley, 2005.

[MSYL00] R. Mei, W. Shyy, D. Yu, and L.-S. Luo. Lattice Boltzmann Method for 3-D Flows with Curved Boundary. *J. Comput. Phys.*, 161(2):680–699, July 2000.

[NN97] N. Nupairoj and L.M. Ni. Performance Metrics and Measurement Techniques of Collective Communication Services. In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 212–226, 1997.

[PJJ07] A. Phansalkar, A. Joshi, and L.K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In D.M. Tullsen and B. Calder, editors, *ISCA*, pages 412–423. ACM Press, 2007.

[PKW+03] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde. Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes. *Parallel Process. Lett.*, 13(4):549–560, 2003.

[PLM06] C. Pan, L.-S. Luo, and C.T. Miller. An Evaluation of Lattice Boltzmann Schemes for Porous Medium Flow Simulation. *Comput. & Fluids*, 35(8-9):898–909, September 2006.

[PP04] J.E. Pilliod and E.G. Puckett. Second-Order Accurate Volume-of-Fluid Algorithms for Tracking Material Interfaces. *J. Comput. Phys.*, 199(2):465–502, 2004.

[PS92] E.G. Puckett and J.S. Saltzmann. A 3-D Adaptive Mesh Refinement Algorithm for Multimaterial Gas Dynamics. *Physica D*, 60:84–104, 1992.

[PTD+04] T. Pohl, N. Thürey, F. Deserno, U. Rüde, P. Lammers, G. Wellein, and T. Zeiser. Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 21. IEEE Computer Society, 2004.

[PY92] B.J. Parker and D.L. Youngs. Two and Three Dimensional Eulerian Simulation of Fluid Flow with Material Interfaces. Technical Report 01/92, UK Atomic Weapons Establishment, Aldermaston, Berkshire, February 1992.

[Rab03] R. Rabenseifner. Hybrid Parallel Programming on HPC Platforms. In *Proceedings of the Fifth European Workshop on OpenMP*, pages 185–194, September 2003.

[RK98] W.J. Rider and D.B. Kothe. Reconstructing Volume Tracking. *J. Comput. Phys.*, 141:112–152, 1998.

[RRL01] M. Renardy, Y. Renardy, and J. Li. Numerical Simulation of Moving Contact Line Problems Using a Volume-Of-Fluid Method. *J. Comput. Phys.*, 171(1):243–263, 2001.

[SK06] H.R. Schwarz and N. Köckler. *Numerische Mathematik*. Teubner, 2006.

[SKH+99] K. Shimada, T. Kawashimo, M. Hanawa, R. Yamagata, and E. Kamada. A Superscalar RISC Processor with 160 FPRs for Large Scale Scientific Processing. In *Proceedings of Int. Conf. on Computer Design*, pages 279–280, 1999.

[Smi88] J.E. Smith. Characterizing Computer Performance with a Single Number. *Comm. ACM*, 31(10):1202–1206, October 1988.

[SMK03] H.D. Simon, C.W. McCurdy, and W.T.C Kramer. Creating Science-Driven Computer Architecture: A New Path to Scientific Leadership, 2003. URL: `http://www.nersc.gov/news/HECRTF-V4-2003.pdf`.

[SMPR03] H. De Sterck, R.S. Markel, T. Pohl, and U. Rüde. A Lightweight Java Taskspaces Framework for Scientific Computing on Computational Grids. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1024–1030. ACM Press, 2003.

[Suc01] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Numerical Mathematics and Scientific Computation. Oxford University Press, 2001.

[SZ99] R. Scardovelli and S. Zaleski. Direct Numerical Simulation of Free-Surface and Interfacial Flow. *Annu. Rev. Fluid Mech*, 31:567–603, 1999.

[Tau95] G. Taubin. Estimating the Tensor of Curvature of a Surface from a Polyhedral Approximation. In *Proceedings of the Fifth International Conference on Computer Vision*, pages 902–905, 1995.

[Thi05] M. Thies. *Lattice Boltzmann Modeling with Free Surfaces Applied to Formation of Metal Foams*. PhD thesis, Universität Erlangen, Germany, Germany, 2005.

[Tho79] J.A. Thorpe. *Elementary Topics in Differential Geometry*. Springer-Verlag, Berlin, Heidelberg, 1979.

[TK98] S.A.M. Talbot and P.H.J. Kelly. *High Performance Computing Systems and Applications*, chapter Stable Performance for ccNUMA Using First Touch Page Placement and Reactive Proxies. Kluwer Academic Publishers Group, May 1998.

[TPR+06] N. Thürey, T. Pohl, U. Rüde, M. Öchsner, and C. Körner. Optimization and Stabilization of LBM Free Surface Flow Simulations using Adaptive Parameterization. *Comput. & Fluids*, 35(8-9):934–939, September 2006.

[WA04] B. Wilkinson and M. Allen. *Parallel Programming*. Prentice-Hall, 2004.

[Wel54] P. Welander. On the Temperature Jump in a Rarefied Gas. *Ark. f. Fysik*, 7(44):507–553, 1954.

[WG00] D.A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Number 1725 in Lecture Notes in Mathematics. Springer-Verlag, 2000.

[Wil01] M.V. Wilkes. The Memory Gap and the Future of High Performance Memories. *ACM SIGARCH Comput Architecture News*, 29(1):2–7, March 2001.

[Wil03] J. Wilke. Cache Optimizations for the Lattice Boltzmann Method in 2D. Seminar paper, Universität Erlangen, Germany, 2003.

[WPKR03] J. Wilke, T. Pohl, M. Kowarschik, and U. Rüde. Cache Performance Optimizations for Parallel Lattice Boltzmann Codes in 2D. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 441–450. Springer-Verlag, 2003.

[WZDH06] G. Wellein, T. Zeiser, S. Donath, and G. Hager. On the Single Processor Performance of Simple Lattice Boltzmann Kernels. *Comput. & Fluids*, 35:910–919, 2006.

[XL97] C. Xu and F.C.M. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers Group, Dordrecht, The Netherlands, 1997.