Karlsruhe Institute of Technology

Institute for Applied and Numerical Mathematics 4

Prof. Dr. Vincent Heuveline

Performance Optimization of Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries

Diploma Thesis of

Jonas Fietz

AG Numerische Simulation, Optimierung und Hochleistungsrechnen

Reviewer:	Prof. Dr. Vincent Heuveline
Second reviewer:	Prof. Dr. Peter Sanders
Advisor:	Dr. Mathias Krause
Advisor:	Dipl. Inform. Dipl. Math. Christian Schulz

December 2011

Preface

I would like to thank my two reviewers, Prof. Vincent Heuveline and Prof. Peter Sanders, for allowing me to write this thesis at both their institutes, for their advice and their support.

Also, I would like to thank my two advisors, Dr. Mathias Krause and Christian Schulz, for taking the time to discuss ideas, to make suggestions, to proofread and for their general support.

Finally, I would like to thank my proofreaders Isabel Krauss, Judith Ottich and Marlene Sauer, as well as my parents for giving me the financial support for studying in the first place.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegeben Quellen und Hilfsmittel verwendet habe.

Karlsruhe, 30.11.2011 Jonas Fietz

Contents

1. Introduction **1**

1.1. Outline 2

1.2. License 3

2. Fundamentals **5**

2.1. Mesoscopic Fluid Flow Model 5

2.2. Lattice Boltzmann Methods 8

2.2.1. Discretization of Time and Phase Space 9

2.2.2. Boundary and Initial Conditions 10

2.2.3. Implementation 11

2.3. Graphs and Graph Partitioning Algorithms 13

2.3.1. Basic Concepts and Definitions 13

2.3.2. Multi-level Graph Partitioning 15

2.3.3. Concepts Used in KaFFPa 15

2.4. Octree-based Domain Decomposition 16

2.4.1. Definition of an Octree 16

3. Analysis of Current State **19**

3.1. Original Load Balancer in OpenLB 19

3.2. Heuristic Domain Decomposition 20

4. Optimization Strategies **23**

4.1. Load Balancing Utilizing Graph Partitioning 23

4.1.1. Calculating Node and Edge Weights 24

4.1.2. Example 26

4.1.3. Possible Further Optimizations 26

4.2. Heuristic Load Balancing 28

4.3. Octree-based Domain Decomposition 29
4.3.1. Implementation 30
4.4. Shrinking Cuboids 30
4.5. Other Considered Strategies and Variations 31

5. Evaluation **33**
5.1. Benchmark Problems 33
5.1.1. Lid-Driven Cavity 33
5.1.2. Bifurcation 34
5.2. Description of Test-Machines 35
5.2.1. HP XC3000 36
5.2.2. Institutscluster 37
5.2.3. Desktop Computer 38
5.3. Results 38
5.3.1. Measurements 38
5.3.2. Graph Based Load Balancer 39
5.3.2.1. Validation of Model Assumptions for Communication 39
5.3.2.2. cavity3d 40
5.3.2.3. bifurcation 40
5.3.2.4. bifurcation-large 42
5.3.3. Heuristic Load Balancer 44
5.3.4. Octree Domain Decomposition 44
5.3.5. Shrinking Cuboids For Geometry Approximating Sparse Domain Decomposition 50

6. Conclusion **57**
6.1. Summary 57
6.2. Outlook 58

A. Appendix **59**
A.1. Open Source Library OpenLB 59
A.1.1. Project Overview 59
A.1.2. Authors 59
A.2. KaFFPa 61

B. German Abstract **63**

Bibliography **67**

List of Figures **71**

List of Tables **73**

1. Introduction

Computational fluid dynamics (CFD) as a research field has become more and more important in the last decades, as the rise in computing power enabled ever more complicated simulations. CFD help accelerate research in many different areas, as applications are manifold.

One example application is the computation of vehicular aerodynamics in the field of automotive and aeronautical engineering. Another application, in which the importance of CFD has risen tremendously in the past few years, lies in the medical realm. The function of the human respiratory system has not yet been fully understood, and its complete description is extremely complex. Due to highly complex multiphysics phenomena involving multi-scale features and ramified, complex geometries, it is considered one of the *Grand Challenges* in scientific computing today. One day, numerical simulation of fluid flows is hoped to enable surgeons to analyze possible implications prior to or even during surgery. This requires *efficient parallelization* strategies as well as taking advantage of the currently available hardware architectures with large amounts of cores in both CPUs and Graphical Processing Units (GPU) with hybrid parallelization techniques.

One of the projects that is working on simulating the respiratory tract is the *United Airways* project. The United Airways project uses the open source software *OpenLB* for its simulations of the human lungs and nose. OpenLB is a library for simulating user defined fluid flows. It has facilities for enabling users to write their own simulations with their own characteristics.

The library OpenLB implements an open source version of *lattice Boltzmann* (LB), that have been developed in the second half of the last century. These methods are

inherently parallel and therefore well suited to solve large-scale problems.

The goal of this thesis is to research, implement and test possible performance improvements for LBM on the example of OpenLB. Several possible bottle-necks are identified in advance and then analyzed, leading to two dimensions of optimization strategies.

The first area that needs improvement is the load balancer in OpenLB. The current method for load balancing is very simplistic in its design. Therefore, two different new load balancers are implemented. The first is a graph based load balancer that leans heavily on work in the field of graph theory, utilizing the graph partitioner *KaFFPa* as a library to improve load balancing. An alternative method using only heuristics instead of external software is presented as well.

These new balancers are evaluated on different geometries, of which the *bifurcation* tests are prime examples of fluid flow simulation in the human lungs or capillaries. The tests showed that the new load balancers are competitive with the traditional load balancer, improving performance perceptibly in most non-trivial cases.

But the most important effect of using these load balancers is losing restrictions for the decomposition of the geometries, enabling improvements for sparse domain decomposition. The goal here is to better fit the computational domains to the underlying geometry, leading to less wasted processing power.

Two methods are presented and evaluated in this thesis: The first is improving the original domain decomposition by *shrinking* the sub domains. The second strategy is the usage of *octrees* for geometry aware domain decomposition. This tree data structure and the spatial order it signifies allows to get a much closer fit to the different geometries tested. Results show that these are very promising approaches.

1.1. Outline

A short summary of all chapters is shown here to give the reader an overview over the different parts of this document.

In Chapter 2 the prerequisites used in this thesis will be presented. This section is mostly meant as a place to get references for readers interested in further background information or to ascertain themselves with specific terms and concepts. It will also relate the work presented to other work in te different areas.

Chapter 3 contains an analysis of the original version of OpenLB, as well as the implications of identified shortcomings.

In Chapter 4, we will describe in greater detail the designed and implemented optimization strategies, as well as discuss possible alternatives and the implications.

An evaluation of the implemented strategies then follows in Chapter 5. Here, three different benchmark problems are tested, and the improvements in performance explained and quantified. Implications for later users are discussed here as well.

Chapter 6 will then contain a recapitulation of the presented concepts and results. Additionally, a short outlook into further opportunities for improvements and research will be given.

The Appendix A describes the two used software packages – KaFFPa and OpenLB.

1.2. License

2. Fundamentals

In this chapter several of the concepts used in this thesis are introduced. They give an overview over most basics needed to understand the theories developed in the following sections. In case further clarification is needed or the reader is interested in a more in-depth look at some of the topics covered, it is proposed to refer into the cited works in the bibliography.

2.1. Mesoscopic Fluid Flow Model

On the following pages the models and methods used to examine the behavior of fluid flows are described. As this thesis deals specifically with the fluid flow simulation using Lattice Boltzmann Methods (LBM), those are described in more detail. Basic understanding of fluid dynamics results from the underlying physics and observed behaviors of fluids in controlled experiments. From those observations one can derive assumptions on different levels, which can be classified by their degree of refinement. According to [21], one can give a partial classification of those refinements – and therefore the underlying models – as such:

- Macroscopic models: Typical models at this level include Navier-Stokes, Euler or Stokes. Typical observations are macroscopic in nature such as fluid velocity, pressure, density or temperature while ignoring properties at the molecular level.

- Mesoscopic models: Typical models at this level include Liouville, Boltzmann or the BGK-Boltzmann equation. In these statistical measures such as the mean free path, mean molecular velocity or density is observed.

- Microscopic models: Modelled by molecular dynamics as they are given by the understanding of physical laws. Observations are at the molecular level such as molecular mass, velocity or extent and form. As one can imagine, these easily overwhelm even the biggest computers currently available for any interestingly sized problems (details on why this is unfeasible are given by [31, pp. 3,4]).

The general equation governing the dynamics of dilute gases was developed by Boltzmann in 1872, and is accordingly called the Boltzmann equation. It has been applied to various other fields such as electron transport in semiconductors, neutron transport or quantum liquids [10], [23].

The mathematical description of the mesoscopic model is derived from the microscopic view of a fluid via statistical measures. An overview of the underlying assumptions is given in the following paragraphs as described in [21] and [31]. Exact derivations for the model can be found in [21] or [10] as well as the derivations for the macroscopic quantities.

The fluid is modelled as a collection of N molecules in a domain $\Omega \subseteq \mathbb{R}^d$ at a certain temperature. Each molecule is then represented as a particle with the same particle diameter $\delta \in \mathbb{R}_{>0}$ and the same mass $m \in \mathbb{R}_{>0}$.

The particles interact with each other at any given time. This is known as *Brownian motion*. Particles are assumed to be free flowing, respecting Newton's laws, without being subject to any external volume forces except gravity. The path between two collisions is called *free path*, the length of the average free path in a volume over a given time interval is the *mean free path* $l_f \in \mathbb{R}_{>0}$. All collisions are supposed to be elastic, meaning that momentum and kinetic energy are conserved. Particles also are assumed to collide seldom – or interact weakly – as their assumed diameter is very small. In addition, one makes the assumption that all collisions are binary, i.e. only two particles can collide with each other at the same time.

In such a gas, each particle is perfectly described by the current position of its center $\mathbf{r} \in \Omega$ and its current velocity $\mathbf{v} \in \Xi = \mathbb{R}^d$. By convention, one uses the following terms for these spaces:

- position space $\Omega \subseteq \mathbb{R}^d$
- velocity space $\Xi = \mathbb{R}^d$
- phase space $\mu = \Omega \times \Xi \subseteq \mathbb{R}^{2d}$

Both $\mathbf{r}$ and $\mathbf{v}$ are functions of a time interval $I = [t_0, t_1] \subseteq \mathbb{R}$ into Ω and Ξ. At a certain point in time $t \in I$ the system is defined as the state of all particles $\mu^N \subseteq \mathbb{R}^{2dN}$.

Based on this microscopic model, to get to the level of a mesoscopic model one discards the information for each individual particle, and instead examines probabilities for the states of the fluid. The position $(\mathbf{r}(t), \mathbf{v}(t))_i$ of each particle in its phase space μ_i is considered as the realisation of a random variable. The phase space μ_i generates a σ-Algebra $\sigma(\mu_i)$. A σ-algebra is defined as follows: For a non-empty set Ω and $\mathcal{A} \subset \mathbb{P}(\Omega)$, where $\mathbb{P}$ is the power set of Ω. This system of subsets of Ω is then called a σ-algebra if it satisfies the three conditions [16]

1. $\Omega \in \mathcal{A}$
2. $A \in \mathcal{A} \Rightarrow A^c (:= \Omega \setminus A) \in \mathcal{A}$
3. $A_n \in \mathcal{A} (n \in \mathbb{N}) \Rightarrow \bigcup_{n=1}^{\infty} A_n \in \mathcal{A}$

So $\sigma(\mu_i)$ is then a σ-algebra with the probability $P_i^t(A_i)$ to find $(r(t), v(t))_i$ in $A_i \in \sigma(\mu_i)$ therefore giving us the probability space $(\mu_i, \sigma(\mu_i), P_i^t)$. From these probability spaces one constructs the finite discrete probability space (Θ, P_A^t) for a given $A \in \sigma(\mu)$. The variable $k \in \Theta$ is the number of particles found in A with the probability $P_A^t(k)$. One can now calculate the expected number of particles at a point in time t in a certain space A as $E_A^t = \sum_{k=1}^{N} k P_A^t(k)$.

$E_A^t/N =: P_N^t(A)$ as a function of $A \subseteq \mu$ defines the probability measure itself on the probability space consisting of phase space μ and the σ-algebra $\sigma(\mu)$.

One can derive a multitude of macroscopic quantities using these probability measures such as particle density, mass density, fluid velocity, stress tensor or pressure. For this, the density function p_N^t of P_N^t is multiplied with the number of particles N in Ω. This results in the *particle density function* f given by

$$f : \begin{cases} I \times \Omega \times \mathbb{R}^d & \to \mathbb{R}_{\geq 0} \\ (t, \mathbf{r}, \mathbf{v}) & \to f(t, \mathbf{r}, \mathbf{v}) \end{cases} \tag{2.1}$$

Using the restrictions defined above, one can then derive the numbers of molecules entering and leaving a certain subset $B \times C \subseteq \Omega$. For an external volume force Fresults in the Boltzmann equation for rigid spheres as follows

$$\left(\frac{\delta}{\delta t} + v \cdot \nabla_r + \frac{F}{m} \cdot \nabla_v \right) f = \underbrace{\int_S \int_{\substack{\mathbb{R}^d \\ n_*^+}} n_* \cdot \mathbf{g}(f' f'_* - f f_*) d\mathbf{v}_* d\mathbf{s}}_{=:J(f)}, \tag{2.2}$$

where J is the *collision operator*.

This derivation and explanation of the Boltzmann equation closely follows the one given in [21].

Due to the complex structure of the collision operator J, Bhatnagar, Gross and Krook proposed in 1954 an elegant simplification [7]. Their *BGK collision operator* is defined by

$$Q(f) := -\frac{1}{\omega}(f - M_f^{eq}) \text{ in } I \times \Omega \times \mathbb{R}^d, \tag{2.3}$$

where $M_f^{eq} := f^{eq}(n_f, \mathbf{u_f}, T_f)$, T_f is the absolute temperature and u_f is the specific velocity. M_f^eq is a specific Maxwellian distributions with the moments of f serving as suitable choices [21].

2.2. Lattice Boltzmann Methods

Originally, lattice Boltzmann methods were developed in the wake of *lattice gas cellular automata* (LGCA) methods developed by Uriel Frisch, Brosl Hasslacher and Yves Pomeau in 1986 [31]. The elegance of the LGCA methods was that they allowed to reproduce the complexity of fluid flows using an intrinsically parallel paradigm. Typically, they trace the movement of a given number of particles. One can then deduce macroscopic quantities such as pressure or velocity from the state of the whole system.

LGCAs usually choose a uniform grid, referred to as a *lattice*. An example of such a grid, as used by the Frisch-Hasslacher-Pomeau automaton, is a regular lattice with a hexagonal symmetry, in which each lattice site is surrounded by six neighbors. [31] Each particle has a certain velocity which enables it to reach exactly one of its neighbors during each *propagation* or *streaming step*. Exactly as defined before all particles have the same mass, and as they all can only reach their nearest neighbors, all of them have the same kinetic energy. This streaming step can be seen as a constant time interval, during which there are no collisions. After each streaming step a *collision step* is executed. Intuitively, this means that once the particles arrive at their destination, they collide with whatever is there. This collision of course has to fit into the same structure as before, so this is only a radical simplification of the Newtonian equations for collisions. But due to the Boolean characteristics, LGCA schemes resulted in a certain amount of statistical noise, motivating the transition to *Lattice Boltzmann* (LB) schemes where the specific number of moving particles was replaced with an average, as invented by McNamara and Zanetti [25]. Another downside of LGCA schemes was the unsuitability for high-Reynolds flows due to some characteristics of their collision operator, which prompted the development of

alternatives for these operators. The collision operator defined by BGK as given in Section 2.1 is one of those alternatives.

2.2.1. Discretization of Time and Phase Space

To begin this section, we will introduce the *characteristic length* $L \in \mathbb{R}_{>0}$ which characterizes the magnitude of the domain Ω. We also fix a *characteristic speed* $U \in \mathbb{R}_{>0}$ which stands for a typical magnitude of macroscopic flow velocity. Two typical microscopic magnitudes aret the mean free path l_f and the mean absolute thermal velocity $\bar{c}$. One can derive $\bar{c} = \sqrt{\frac{8}{\pi}RT}$ with R being the *universal gas constant* and T being the absolute temperature. The *speed of sound* is the given by $c_s = \sqrt{\gamma RT}$ with the *adiabatic exponent* γ of the same magnitude as $\bar{c}$. From this, the two following dimensionless quantities are defined:

Knudsen number:

$$Kn := l_f/L, \tag{2.4}$$

and *Mach number*:

$$Ma := U/c. \tag{2.5}$$

We also define kinematic viscosity $\nu := \frac{\pi}{8}\bar{c}l_f$, dynamic viscosity $\mu := \nu\rho$ and *Reynolds number*

$$Re := \frac{\rho UL}{\mu} = \frac{pi\gamma}{2}\rho\frac{Ma}{Kn}. \tag{2.6}$$

Definitions follow the ones given in [21].

Now let $h := l_f/\bar{c}$ be the discretization parameter. It is the shortest distance between two nodes of a uniform grid approximating the position space Ω. This grid is commonly called the lattice.

The velocity space is discretized into $q \in \mathbb{N}$ specific velocities v_i, with the space of velocities $Q := \{v_i : i = 0, 1, \ldots, q-1\}$. The velocities are chosen so that a particle with a given velocity always either stays at the current position or at a neighboring knot of the lattice at $t = t + h^2$. This results in the discretization of time space as $I_h := \{t \in I : t = t_0 + h^2k, k \in \mathbb{N}\}$.

Discretizations are usually named after the number of space dimensions d and the number of velocities q, normally using the convention of DdQq to describe a specific LB model. Typical models used are D2Q9 or D3Q19 [31]. The D3Q19 example stands for a 3-dimensional space with 19 directions. The 19 speeds are made up of the six neighboring nodes, the twelve neighbors sharing an edge and no motion at all

(see Figure 2.1). The D3Q27 model would extend the D3Q19 model to also include the speeds to the neighbors sharing a corner.

With this parameter h, one can derive the Lattice Boltzmann equation from the BGK-Boltzmann equation we saw earlier (for an example for such a derivation see [21]. For a vanishing force term $F = 0$ in $I \times \Omega$ and viscosity ν we get the lattice Boltzmann equation as follows:

$$f_i(t + h^2) - f_i(t) = -\frac{1}{3\nu + 1/2}\left(f_i(t) - M_{f_i}^{eq}(t)\right) \text{ for } t \in I_h, i = 0, 1, \cdots, q - 1. \quad (2.7)$$

This equation gives us the q different equations for the DdQq model.

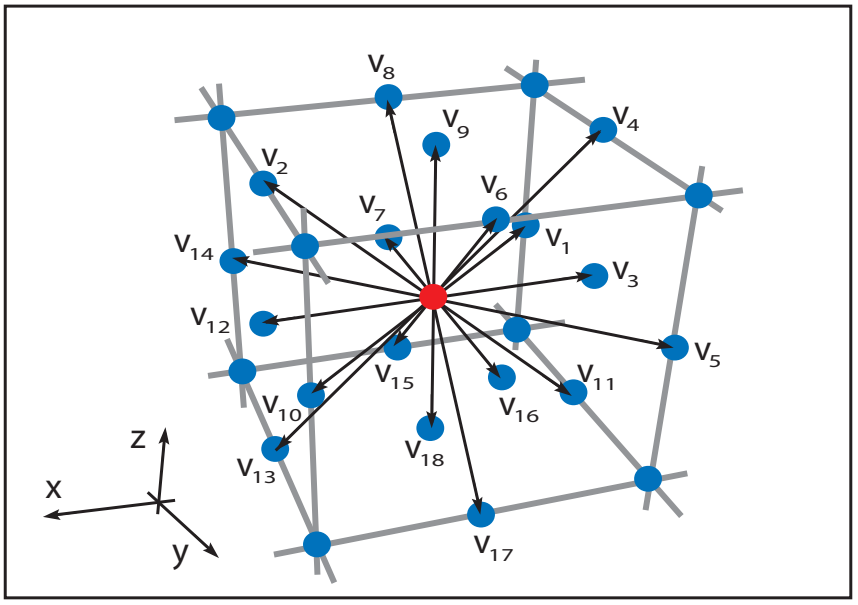

Figure 2.1.: Example for *D3Q19* model showing the 19 different speeds

2.2.2. Boundary and Initial Conditions

This section is meant to be of an exemplary character to give readers an idea of what kind of conditions are commonly used and being researched for the problems of fluid flows. One difficulty with specifying boundary conditions is that at least for fluid flows, the boundary conditions are often characterised by macroscopic properties such as pressure and velocity, often leading to feedback loops. In addition, boundary conditions do not necessarily have to be aligned to the grid coordinates, leading to complex interpolations and abstractions.

At least for solid walls three boundary conditions are commonly differentiated:

- No-Slip approximates a wall, at which there is no motion of fluid at all, therefore defining a bounce-back boundary condition.
- Free-Slip being a wall that has no mentionable frictional influence on the fluid
- Frictional-Slip boundary condition which is a mixture of the two above. An extension of this kind can then be sliding walls moving tangentially, a fairly straightforward generalization [31].

Initial conditions and more general boundary conditions are far more complicated and still an active research topic [21].

2.2.3. Implementation

This section gives an overview over some of the implementation details in the software OpenLB, as far as it is needed to understand the motivation and the implementation of the optimizations developed in this thesis.

As described above, an LBM is usually used on a regular, homogeneous lattice Ω_h. But, if needed, it may as well be executed on an inhomogeneous lattice by using a *multi-block* approach. This commonly entails dividing the lattice in several sub-blocks and implementing a suitable interface between the blocks of different resolutions. This approach has the additional benefit that the execution speed on homogeneous blocks has been shown to be higher than on unstructured grids. A multi-block approach also lends itself very well to parallelization by splitting the geometry into smaller blocks and distributing these to the specific compute nodes, thereby also reducing memory requirements and allowing parallel execution of multiple blocks.

This approach is reflected in the data structure `BlockLattice` in OpenLB. A `BlockLattice` is the representation of a rectangular set of `Cell`s. Each `Cell` holds the q variables for the velocity distribution functions f_i. The `BlockLattice`s are encapsulated in another data structure, the `SuperLattice`, representing the whole geometry.

The calculations of the LB algorithms are implemented in the `BlockLattice` in OpenLB. For these calculations all cells are evaluated iteratively and a local collision step is executed, which is then followed by the streaming step. The collision operator can be given by the user to characterize specific physical conditions for the simulation. This simple model does not encompass more complex boundary conditions, but as those are viewed as local and are only relevant in very confined areas, they are implemented as special procedures executed in a post-processing step. These steps only access the `Cell`s needed. Therefore their complexity can be ignored.

After each step, boundary `Cell`s are communicated to the neighboring `BlockLattice`s. This so far describes a very typical implementation of the LB algorithm, as it is probably present in most similar software.

Some more details are a bit more implementation dependent, so we will have a look at how the `SuperLattice` handles the structural data of the geometry. The `BlockLattice` structure itself is unaware of its position within the geometry. This information is saved in a parallel data structure named `Cuboid`. The cuboids themselves are organized in a `CuboidGeometry`, which is part of the `SuperLattice`. The communication method used in this thesis is only MPI for which the `SuperLattice` initializes a `Communicator`. A communicator is the structure defining the data exchange between different `Cuboid`s and `BlockGeometry`s. For each communication between different `BlockGeometry`s, one data array is defined, so that the overhead for MPI is kept to a minimum. In this array, all communication data is aggregated during each step. For the communication, a layer of ghost cells is created around the cuboid to buffer the data needed.

Algorithm 1 Basic steps of the parallel lattice Boltzmann algorithm for communication based parallelization with MPI [21]

Reading Input
Simulation Setup and Initialization
Time loop
for $t = t_0 \rightarrow t = t_{max}$ **do**
  Collision
  {Blocking}
  {Communicating}
  {Writing to ghost cells}
  Streaming
  Post-Processing
  Writing Output
end for

As a note to the reader, in the following chapters, Cuboid will usually be used to describe the part of a geometry consisting of the associated `Cell`s, `BlockGeometry` and `Cuboid`.

This concept also allows for implementations of hybrid parallelization [17], for example to compute certain BlockGeometries on certain kinds of compute nodes, e.g. certain kinds of graphic card clusters, for which implementations exist in other software.

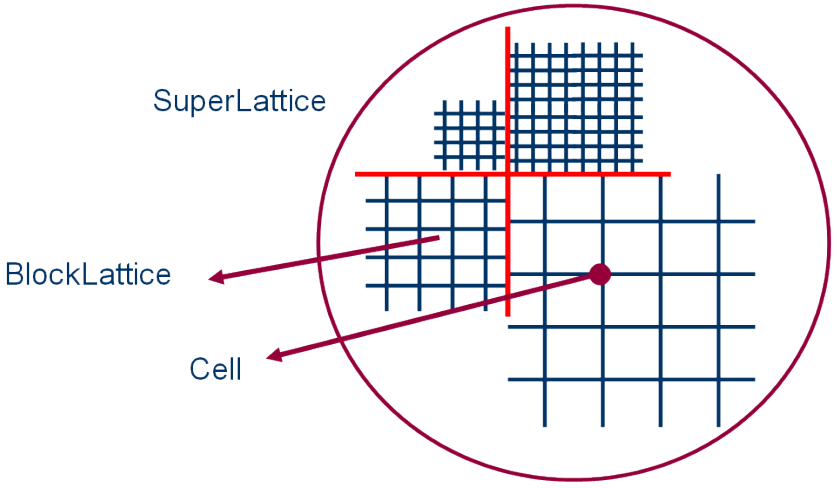

Figure 2.2.: The different levels of abstractions in OpenLB used for *hybrid parallelization* [21]

2.3. Graphs and Graph Partitioning Algorithms

Graph partitioning is a technique used in many fields such as computer science or engineering. One very good use-case of partitioning of unstructured graphs is the field of high performance computing; in this case, graphs can be used to model the underlying computation and communication, and the partition is then a load balance with reduced communication and equal sizes blocks of computation.

2.3.1. Basic Concepts and Definitions

A *directed graph* G is defined as a pair (V, E). V is a finite set of *vertices* and therefore called *vertex set*. The set E is called the *edge set*; its elements are *edges* and it is defined as a binary relation on V.

For an *undirected graph* $G = (V, E)$, E is defined as a set of unordered pairs of elements of V opposed to the ordered pairs in a directed graph. So an edge $e \in E$ is a set $\{u, v\}$, $u, v \in V$ and $u \neq v$. We follow the usual convention for edges in replacing the set notation $\{u, v\}$ with (u, v) which is equal to (v, u).

If (u, v) is an edge in a directed graph $G = (V, E)$, one says that (u, v) is *incident from* or *leaves* vertex u and is *incident to* or *enters* vertex v.

Let (u, v) be an edge in a graph $G = (V, E)$, then vertex v is said to be *adjacent* to vertex u. If the graph is not directed, the adjacency relation is symmetric, which is not necessarily the case for directed graphs.

A *matching* M in a graph G is a set of pairwise non-adjacent edges. This means

that no two edges share a common vertex.

Extending the definition for the vertices and edges to include a weight leads to a *weighted graph*. For this we will use the functions

$$\begin{aligned} c : V &\rightarrow \mathbb{N} \\ v &\rightarrow c(v) \end{aligned}$$

and

$$\begin{aligned} \omega : E &\rightarrow \mathbb{N} \\ e &\rightarrow \omega(e) \end{aligned}$$

to assign weights to vertices and edges. ω and c are then extended to sets so that $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$.

A *contraction* of an edge $e = (v_1, v_2)$ is an operation on a graph $G = (V, E)$ that combines v_1, v_2 and adds the new vertice v_n back to V to create $V' = V \cup (v_n) \backslash (v_1, v_2)$. Then, it removes the edge e from E and replaces all edges (v_1, v) and (v_2, v) with (v_n, v) for $v \in V$ to create E'. The weight $c(v_n) = c(v_1) + c(v_2)$. If replacing edges of the form (u, v_1), (u, v_2)leads to the situation of two parallel edges (u, v_n), these edges are replaced with a single edge with $\omega((u, v_n)) = \omega((u, v_1)) + \omega((u, v_2))$.

A *k-way graph partitioning problem* of a graph $G = (V, E)$ is defined as follows: With $|V| = n$, partition V in k subsets $V_1, V_2, \ldots, V_k$, such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $|V_i| = n/k$ and $\bigcup_i V_i = V$, while minimizing the *edge-cut*. For V_i, V_j, let $E_{i,j}$ be all edges (u, v) with $u \in V_i, v \in V_j$. Then the edge-cut is defined as

$$\sum_{i,j,i \neq j} \omega(E_{i,j}), \text{ with } E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}.$$

The *balancing constraint* places the demand that $\forall i \in \{1, .., k\} : c(V_i) \leq L_{max} := (1+\epsilon)c(V)/k + max_{v \in V} c(v)$ for some parameter ϵ. The last term is needed as vertexes can only be applied atomically to the partitions.

A projection back from $G' = (V', E')$ to $G = (V, E)$ is a reversal operation for a contraction. It maps a partition V_i' back to a partition V_i so that for all $v_1, v_2 \in V$ that were combined to a node $v_n \in V_i' \subseteq V'$ are assigned to the subset $V_i \subseteq V$, while all untouched nodes from V_i' are assigned one-to-one to V_i.

The definitions for the graphs, adjacency, incidence and contractions follow those given in [12, pp.1080-1084]. The description and definition of the k-way graph par-

titioning problem this way have been adapted from [20].

2.3.2. Multi-level Graph Partitioning

Currently, the two major methods for partitioning graphs are multi-level algorithms and spectral bisection and partitioning. The software *KaFFPa* (further details see Section A.2) used in this thesis utilizes multi-level algorithms, so we will focus on those.

Multi-level algorithms are generally divided in three main phases described below, each of which may be accomplished by different strategies [20]

Starting with a graph $G_0 = (V_0, E_0)$ one proceeds with:

- *Coarsening Phase*: The graph is transformed into a sequence of smaller and smaller graphs $G_1, G_2, \ldots, G_m$ such that $|V_1| > |V_2| > \cdots > |V_m|$ by identifying matchings $M \subseteq E$ and contracting the edges in M.

- *Initial Partitioning Phase*: Partition P_m of the graph $G_m = (V_m, E_m)$ is computed, observing the balancing constraint for an $\epsilon > 0$. This may be done either with another algorithm or by contracting the graph to exactly k nodes and using the trivial initial partitioning.

- *Uncoarsening Phase*: The partition P_m of G_m is projected back to G_0 going through the intermediate partitions P_i. At each projection step refinements are made. The refinement algorithms move nodes between different partitions to improve balance and cut size [28].

2.3.3. Concepts Used in KaFFPa

This section will be a brief overview of the algorithms used in KaFFPa as described by Sanders and Schulz in [28]. KaFFPa makes use of some of the methods and strategies proposed in KaPPa [19] and KaSPar [26].

Contraction: A *rating function* scores edges for contraction on local information. The score signifies how useful the contraction of the respective edge would be. Then, a *matching* algorithm tries to maximize the sum of ratings of contracted edges by evaluating the global structure of the graph. For matchings, a *Global Path Algorithm* (GPA) [24] which was chosen to compute matchings, as it has empirically shown good results [29]. The GPA scans edges in order of decreasing weight, and constructs collections of paths and lengths cycles during this analysis. Afterwards, optimal solutions are computed through dynamic programming.

Initial Partitioning: Contraction stops at $max(60k, \frac{n}{60k})$. Then KaFFPa may use *kMetis* or *Scotch* [27] for initial partitioning. For the purpose of this thesis, only Scotch was used as an initial partitioner.

Refinement: Refinement is the last step of this multi-level algorithm. After uncontracting edges to uncoarsen the current graph, different local improvement methods are applied. Two kinds of these are implemented in KaFFPa. The first kind is *Quotient graph style refinement* [19], which utilizes the quotient graph for analysis. In it, each edge yields a pair of blocks which share a non-empty boundary. On these pairs, one can apply a two-way local improvement method moving only nodes between the current two blocks.

The second kind is a *k-way local search*. This method is not restricted to only two blocks and enables the algorithm to take a more global view. It is based on the FM-algorithm from [14]. This method uses a single priority queue. This queue is initialized with the possible gains in edge-cut that can be achieved when moving a specific node to another block. Nodes can be all nodes in the partition boundary. The algorithm then repeatedly looks for the node with the highest gain, which is then moved if this move does not unbalance the graph. This is done until either the queue is empty or certain stopping criteria are reached. Then, the situation with the lowest edge-cut is used.

2.4. Octree-based Domain Decomposition

One of the new methods for domain decompositions evaluated in this thesis uses structures of octrees as their basis. This structure will be defined and explained below. Octrees allow a close fit of the cuboid structure to the underlying geometry of the problem. Octree based methods have been used with great success before in the application of finite element methods, for example in [30] or [9].

2.4.1. Definition of an Octree

An octree is a tree data structure in which each node is either a leaf node or an internal node with exactly eight children. It is a special case of tree structures for every n-dimensional space, although, due to computer graphics, the most commonly used structures are quadtrees (the 2D variant) and octrees. Octrees lend themselves very well to the partitioning of three dimensional spaces, as each node can be seen and understood as a cube in this space. If a node has children, these nodes are the natural sub cubes dividing the original cube in eight equally sized portions. An example can be found in Figure 2.3.

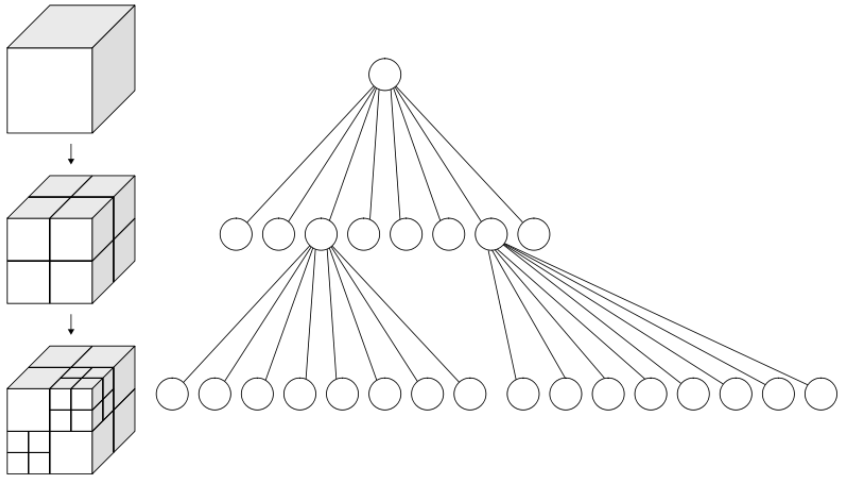

Figure 2.3.: Example Octree with Corresponding Geometry - Author: White Timerwolf, downloaded from "http://en.wikipedia.org/wiki/File:Octree2.svg" on 22.11.2011 - License Creative Commons Attribution-Share Alike 3.0 Unported

3. Analysis of Current State

This chapter will present an analysis of some of the weaker points in the OpenLB library. Then, the effects of these inefficiencies and their consequences are explained briefly.

One of the points identified is the load balancer in OpenLB. A description of its function is given below in Section 3.1. Due to its simplistic nature, the implementation of improvements in the domain decomposition were not possible. These short-comings are discussed in Section 3.2.

3.1. Original Load Balancer in OpenLB

The original load balancer in OpenLB uses a very simple approach. The underlying assumption is that the computational effort for all the cuboids it tries to balance is approximately the same. Also, it completely disregards the effects of communication between nodes and does neither calculate nor use the information. In general, we will refer to an assignment of specifig cuboids to certain processor (cores) as a load balance in some circumstances.

The algorithm works as follows: For nC cuboids created for a specific problem, each of n processors would be assigned either $\lfloor \frac{nC}{n} \rfloor$ or $\lfloor \frac{nC}{n} \rfloor + 1$ cuboids. The first $nC \ mod \ n$ processors would get one more cuboid than the others. Assignment of cuboids is linear, according to a mostly random numbering and therefore a random order of the cuboids.

Each cuboid in OpenLB consists of a certain amount of cells. Each of these cells might be empty, full or a boundary cell. The amount of work any of these kinds of cells is different; exact numbers are shown in Table 4.1. This of course leads to the situation that even same-sized cuboids might vary greatly in the amount of work

required. Therefore, the current load balancer probably creates certain amounts of imbalance in the assignment of the cuboids to CPUs, possibly leaving large chunks of computing power unused because of it. Two solutions to these problems are presented in the optimization section, a graph based load balancer and a simpler heuristic load balancer.

3.2. Heuristic Domain Decomposition

The way the traditional load balancer in OpenLB works has other downsides as well. To achieve the prerequisite for the load balancer that all cuboids have about the same amount of computational complexity – even when only fluid cells are present – OpenLB currently uses heuristics for domain decomposition. These heuristics try to keep the cuboids at about the same size, while also trying to minimize the surface. Minimizing the surface means creating cuboids that are close to cube-shaped, as those have less surface in relation to their volume and therefore need less communication between cuboids per cell. The heuristics also completely ignore the underlying geometry.

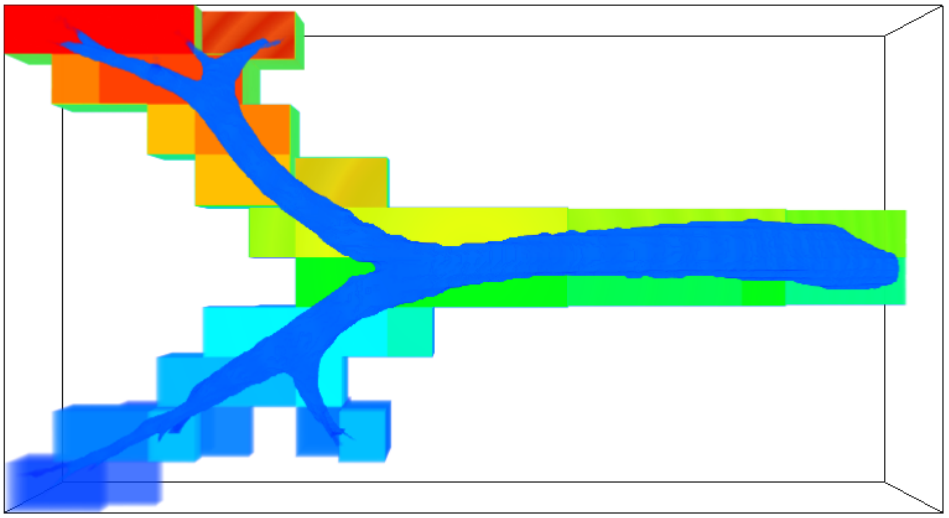

Figure 3.1.: Example for *Domain Decomposition* with the current methodology [21]. Cuboids in the finer parts of the lung are several times larger than the geometry, leading to inefficiency.

This strategy has downsides. As can be seen in the example in Figure 3.1, the cuboids created include a lot of empty cells. Especially for the finer parts of the

geometry, only small amounts of the total volume of the cuboids are really fluid cells. Therefore, following from the prior section, the volume is not an adequate measure of work required for this cuboid. Also, as the empty cells are not needed for the fluid flow simulation, computational power is wasted on these cells. This leads to the proposal of two out of a multitude of possible techniques that allow a better fit to the geometry. The first is using octrees to refine the decomposition in parts of the geometry with high complexity, that is in boundary regions. The second method is the shrinking of the original cuboids to reduce the amount of empty cells around the geometry. Both of these measures were implemented in the course of this thesis and are presented in the following chapter.

4. Optimization Strategies

In Chapter 3, we outlined some of the deficiencies in the current load balancer of OpenLB and the resulting inefficiencies in the domain decomposition, as well as its unused optimization potential. As a solution, this chapter will present two dimensions of strategies for improving the situation. First, the ideas for an improved load balancer are discussed with a graph based load balancer and a simpler load balancer version using heuristics. The second dimension of strategies are about the improvements for sparse domain decomposition.

These different strategies are described as well as the reasoning behind different design decisions and trade-offs.

4.1. Load Balancing Utilizing Graph Partitioning

When executing programs on large parallel computers, one has to assign certain work loads to each of the processors. Classically, one differentiates between static and dynamic load balancers for this task. *Static load balancers* use a priori information or estimations to give a static - hence the name - assignment of work units to, in our case, processor cores. *Dynamic load balancers* are used for algorithms with work load that variates during the run. Here, certain work units are reassigned to other processors on the fly. Examples are adaptive finite element methods or typical server applications on the internet. To extend the terminology here, normally one calls the result of a load balancing algorithm – the assignment of work units to workers – a *load balance* as well.

OpenLB and LBM in general are constant during the runtime in the amount of work required for each cuboid, so only static load balancers shall be considered. As described in the prior sections, the algorithm in OpenLB is divided in two parts.

The first part is the local collision and streaming for each cuboid or blockgeometry. Afterwards, the information is exchanged between different cuboids, that is the information of the border cells for neighboring cuboids which were assigned to different processors is transmitted.

Logically, the perfect load balancer would always achieve minimal communication while achieving perfect load balancing, i.e. each processor would need exactly the same time for the compute step of all its cuboids combined. It is obvious that this can only be the case for the most trivial situations and geometries, because as soon as there are empty cells, computing times will differ. To achieve near perfect load balancing while maintaining a minimal amount of communication, typically one uses results and algorithms from the subject of graph theory, or more exactly graph partitioning.

To utilize graph partitioning, one has to find a way to map the problem at hand onto a graph. Typically, the algorithms for graph partitioning will divide a graph in such a way that each partition has approximately the same node weight while minimizing the cut between the different partitions. It makes sense to associate the work amount or needed CPU time for each cuboid to the weight of a node for this cuboid, and to associate the needed communication between two cuboids to the edge weight of the edge between their respective nodes in the graph. Applying the graph partitioner to this graph will yield subsets of nodes, such that the subsets have approximately the same sum of node weights and therefore computing time. Because of this, the edge-cut – the inter-processor communication – will be minimized. That this relation between edge-cut and communication holds as suggested is shown in Subsection 5.3.2.1. As the problem of graph partitioning is NP-complete, this will not necessarily be the minimal communication for this specific problem and domain decomposition, but it will be good enough in general.

4.1.1. Calculating Node and Edge Weights

The mapping of the problem at hand to the graph is now clear. But one still needs to find the exact numbers for the amount of work to be done for each node and the amount of communication between two cuboids.

We shall begin with the latter. As has been described in Section 2.2, OpenLB aggregates all the communication between nodes in a single array for each communication pair. In order to do so, OpenLB finds all neighboring cells of a cuboid. These are then put in an array and are communicated in this specific order at each communication. This practice reduces the necessary communication to the information in these cells. The size of these data structures stays constant. So to get the

exact amount of communication between two cuboids, one only needs to count the amount of communicated cells. Consequently, this number is then assigned to each edge in the graph.

Calculating the work to be done for each cuboid is not as easy, as the amount of work for empty cells, boundary cells and normal fluid cells differ. As the amount of boundary cells is usually quite small, and as they are treated as an extra step at the end anyway, this special case will be ignored. They will be treated as normal fluid cells. Empty cell in this case means that the geometry at this specific point is solid or that this cell is outside of the fluid filled body being simulated. Intuitively, one would expect the empty cells to take no computation time at all, but - due to the way OpenLB is organized - streaming is still executed for these cells. As it is very possible for a cuboid to consist largely of empty cells, it is important to know how much work the empty cells take when compared to the normal fluid cells. Sadly, this is not a specific constant valid for all types of architectures. Instead, it varied in the tests performed between 1.8 and 4.5 for the differing types of computers. These differences are most likely due to the different memory and cache hierarchies and differing memory access speeds, as the streaming part of the LB algorithm is memory bound.

To get a feel for the amount of variation in the relative execution speed of full and empty cells, tests were done on the two clusters used at the university (for a description see Section 5.2) as well as on a typical desktop machine. For this, we used the cavity3d benchmark (see Subsection 5.1.1) with only fluid cells in one case, and only empty cells in the other case. For verification purposes, this was cross checked with a version of the cavity3d benchmark where the length of one side was doubled; one half was then filled and one half was left empty. The results are shown in Table 4.1. Due to the memory constraint of only 8GB, the test on the desktop computer used a smaller geometry, but the results should still be accurate enough.

	Dimension	Empty	Full	relative (rounded)
HC3	400^3	728	3260	ca. 4.5
IC1	400^3	1230	3619	ca. 2.9
Desktop	300^3	2790	5071	ca. 1.8

Table 4.1.: Execution speed of full and empty cells for cavity3d benchmark

As one can see, the differences in execution times between full and empty cuboids might vary greatly. Therefore, to calculate the work to be done for each cuboid, we

get the formula

$$work = \#fullNodes \times correctionFactor + \#emptyNodes.$$

The *correctionFactor* could as well be applied inversely to the $\#emptyNodes$.

The implementation allows the user to choose the calculation of node weights, either by volume of the cuboid or by accounting for the type of node in each cuboid. For the second case, one can also pass a correction factor that relates the amount of work for full and empty cells.

Of course this only applies for certain collision operators. As these values differ with different collision operators and different computers, the user has to calculate this through tests, if they want to take full advantage of the load balancer. These tests are just basic runs of the cavity3d benchmark, one time only filled with fluid cells and one time filled with only empty cells. These benchmark programs will be supplied with the next published version of OpenLB. These are exactly the same benchmarks as were used for calculating these numbers for this thesis.

4.1.2. Example

To illustrate the difference between the old and new balancer, compare Figure 4.2 of the graph based load balancing to the traditional load balancing in Figure 4.1.

Both figures were a test load balancing done on a cube, which was divided in 256 sub-cuboids. The load balancer had to split the workload of 256 cuboids on to 32 processors. As the workload can be evenly distributed (eight work units per processor), the deciding factor for the graph load balancer is minimizing communication. There is one color per processor and each of the sub-cubes is colored according to its assignment.

One can clearly see that the graph based load balancer tries to build segments as similar to a circle (or cube) as possible as this minimizes the surface and therefore the inter-processor communication. Clearly, the classic load balancer just hands out slices according to the arbitrary numbering, resulting in very simple layering.

4.1.3. Possible Further Optimizations

There are multiple possible optimizations and tweaks that could be done to improve the current version of the graph load balancer.

As mentioned in Section 2.3, one can influence the amount of imbalance in the node weight between different partitions with a parameter ϵ. Due to the fact that OpenLB is mostly compute speed bound, ϵ should be set as low as possible, but there might still be some room for improvement.

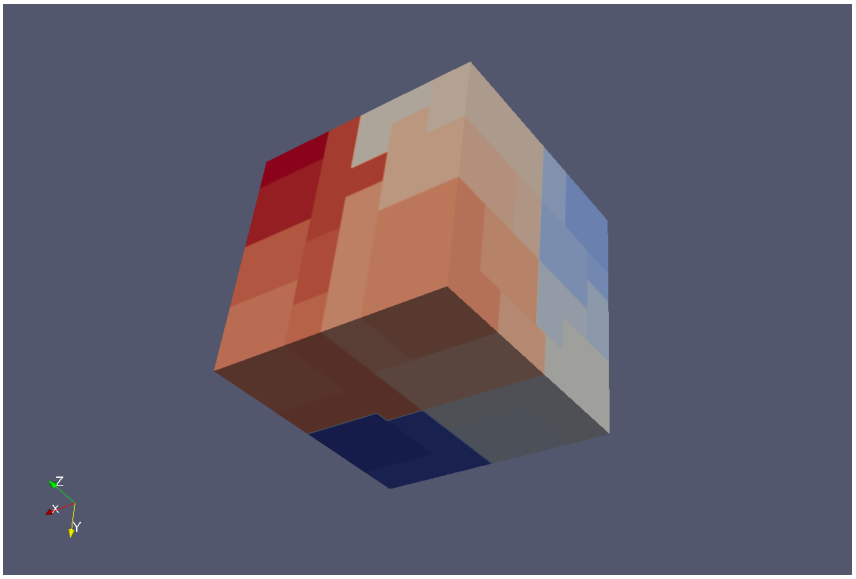

Figure 4.1.: Load balancing of cuboid consisting of 256 smaller cuboids using the old load balancer for 32 tasks

Another idea might be to execute multiple runs of the graph load balancer utilizing the topography and different access speed of clusters, as each compute node usually has multiple cores. In this case, one would first partition the complete geometry, one subset per node. Afterwards, one partitions each subset for each node into as many partitions as there are cores. The rationale behind this is that the communication between local nodes is usually faster than the communication between different cluster nodes. The downside is that in most cases, as there is overhead for communication, one tries to minimize the numbers of cuboids for a reasonable load balance. Because of this, the local load balance step might not be very efficient, as e.g. one might end up with just one big cuboid for eight processors, further complicating the situation.

One problem of this load balancing algorithm is that it only works for clusters with homogeneous nodes because the partitioner tries to create partitions of equal size. But as most clusters in use today have homogeneous computing nodes, or can be used so that only those are taken, this should not be a problem for the application. In the worst case faster nodes would be underutilized and if this should really present a problem, then solutions for creating different blocks also exist (Section 6.2).

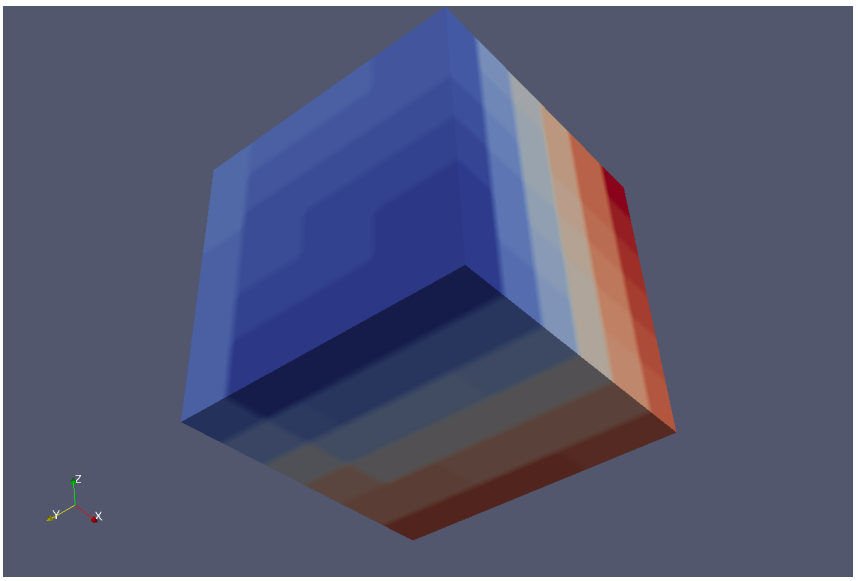

Figure 4.2.: Load Balancing of cuboid consisting of 256 smaller Cuboids with graph based load balancer for 32 tasks

4.2. Heuristic Load Balancing

The solution of the load balancing using the graph load balancer is, as simple as it may seem, quite complicated in the back end, as the algorithms used for graph partitioning can be quite complex. The used graph partitioner KaFFPa is, in general, non-deterministic, although this behavior is not exhibited as strong on smaller graphs, which the resulting graphs usually are. Remembering that communication was not accounted for in the original, simpler load balancer. Therefore, one might get the idea to implement a load balancer which tries to balance the load as best as possible while only taking the amount of computation each node has to handle into account.

The goal of this simple algorithm then should be to equalize the amount of CPU time for all nodes as much as possible. To achieve this, one can use the same methods as for the graph based load balancer to determine the CPU time each cuboid will need per iteration. These times are saved in an array $cuboidTime$ for the nC cuboids. Then, one creates an array $currentLoad$ with size $|currentLoad| = n$ for the n processors used in the cluster. At each step, one assigns the largest remaining work unit to the processor with the least load at that moment. Additionally, we use an array $taken$ of booleans to track which cuboids are already assigned. Pseudo code for the algorithm can be seen in Algorithm 2.

Algorithm 2 Assign each processor an approximately equal amount of work

```
for i = 0 → nC − 1 do
  maxTime ← −1
  maxCuboidNo ← −1
  for iC = 0 → nC − 1 do
    if taken[iC] = False and cuboidTime[iC] > maxTime then
      maxTime ← cuboidTime[iC]
      maxCuboidNo ← iC
    end if
  end for
  minLoad ← currentLoad[0]
  minLoadJ ← 0
  for j = 1 → n − 1 do
    if currentLoad[j] < minLoad then
      minLoadJ ← j
      minLoad ← currentLoad[j]
    end if
  end for
  taken[maxCuboidNo] ← True
  currentLoad[minLoadJ] ← minLoad + maxTime
end for
```

This algorithm is one usually known as *Largest Processing Time* (LPT) first. It has an upper bound of $(\frac{4}{3} - \frac{1}{3n}) \times \text{opt}$[15]. The implementation shown here is intentionally kept simple and has a complexity of $O(nC \times (nC + n))$ as the run time is very small compared to the calculcation for the fluid flow problem. This algorithm could be sped up for example through the use of *binary heaps*.

4.3. Octree-based Domain Decomposition

A very important part of load balancing is the decomposition of the complete domain in sub-domains, as one has to optimize for multiple, sometimes opposing properties of the sub-domains. As has been written before, cuboids should be as large as possible, as more cuboids mean more communication even when local. The surface of each cuboid should be minimal, as this determines the amount of communication for this cuboid. This usually implies a shape as close to a ball or circle as possible. As this is not space filling and only rectangular shapes are supported, the best shape is therefore a cube. Another point to consider is that – as described in Subsection 4.1.1 – empty cells still use some processing power. So the cuboids should be fitting to the geometry tightly.

The original partitioner used heuristics to optimize for equal size, while trying to keep the cuboids as close to cubes as possible. This was needed by the old load balancer which just handed out certain amounts of cuboids without regard for the amount of computing they needed.

This constraint was not given anymore with the new load balancer. As a consequence, other methods of partitioning the problem domain could be evaluated, and one of the algorithms used is octree partitioning.

4.3.1. Implementation

The advantage of octrees for domain decomposition lies in their ability to refine parts of the problem domain, where the underlying geometry is more complex. At these places, there are more opportunities for better fitting the cubes to the geometry than with a naive decomposition in equal parts.

The general concept starts with embedding the problem domain in a cube. As we want the boundaries to be exactly on the boundaries between the different cells being generated by the voxelizer, we use a size of $2^k * \text{voxelSize}$ for some $k \in \mathbb{N}$ as the side length of that cube.

As described above, domain decomposition should always result in cuboids that are neither too small nor too large. E.g. using the surrounding cube by itself would not be very useful for load balancing, while using single cells would create a massive overhead. So the implementation allows for limiting the smallest and the biggest size of cubes.

Having defined the mother cube now, one recursively divides the cube into smaller cubes as long as the geometry in this part is interesting. In our case that means that it contains empty cells at the same time as boundary or fluid cells. If this is not the case, for example if we are completely on the inside or outside of the geometry, then we keep the cube at this size, as long as it is smaller or equal to the maximum size. Additionally, we limit the size to the low end, not splitting further when the cubes would become smaller than the minimum size.

To conserve memory, as the mother cuboid can by quite a bit larger than the non-cube shaped underlying geometries, the cuboids are resized so that they fit into the bounding cuboid of the geometry. An exemplary decomposition can be seen in Figure 4.3.

4.4. Shrinking Cuboids

The reasoning behind this step has been explained in Subsection 4.1.1. As empty cells still need some processing time, it is best not to compute them at all. To exempt

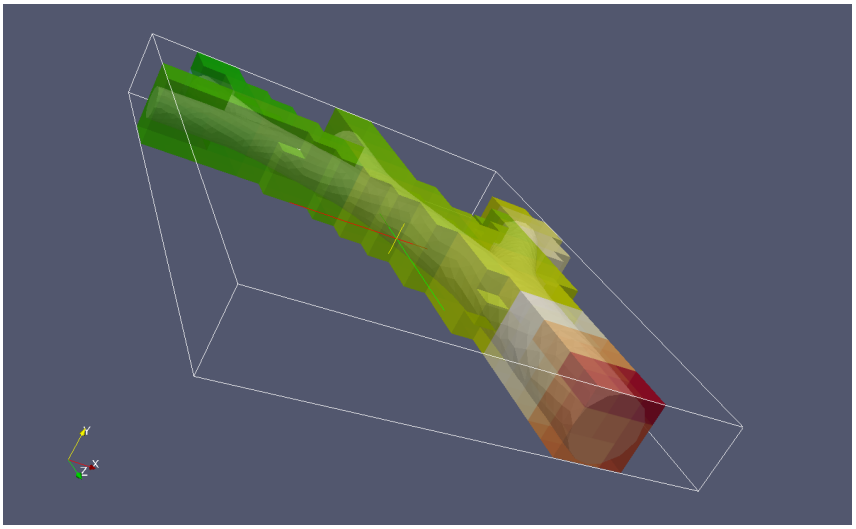

Figure 4.3.: An example octree decomposition of bifurcation-large benchmark with dimension Nx=599, Ny=92, Nz=703 and a minimum cuboid size of 64

most of these empty cells from computation, one only has to shrink the cuboids, as these determine which areas are being calculated. To find out if a cuboid can be shrunk, we just start running through each layer in all 6 directions, starting at the respective faces of the cube. For each layer, we check if it is completely empty, stopping the iteration in this direction when a full cell is found. In the next step, all empty layers are removed. This shrinking is executed for all the cuboids.

Depending on the structure of the geometry, this can be more efficient than the octree decomposition, as the cuboids stay larger but their borders still can get very close to the boundaries of the geometry.

4.5. Other Considered Strategies and Variations

Several other optimizations were considered in the course of this thesis. As described above, one problem of small cuboids is that the communication overhead is quite large. A possible fix for this might be recombining cuboids once they are assigned to a certain processor core. As in the current implementation this would only work when this results in another cuboid, this was not attempted, as this problem largely exists for the octree implementation. There, depending on the geometry, large amounts of cuboids with the minimum size are created. After applying a shrink step, the likelihood for being able to recombine two cuboids would be very small. In addition to this, the recombination would only result in a cuboid by chance as well for the old style decomposition, and is also very limited due to the shrinking step.

The possibility of using multiple levels of graph partitioning have been discussed above. But this presents some other problems, as topology information would have to be supplied to the library, complicating interaction even further for the user.

For heterogeneous computers, many different heuristics exist as this problem is in its most general form NP-complete. A comparison of different heuristics is presented by Braun et.al. in [8].

5. Evaluation

5.1. Benchmark Problems

Several benchmarks were run to evaluate the improvement due to each of the implemented optimizations. The benchmarks with which all the tests were run will be described in the following sections.

5.1.1. Lid-Driven Cavity

The *lid-driven cavity* (LDC) benchmark problem has been used considerably in research for years [11]. The reason for this lies in its simple geometry on the one hand, which makes the setup as such very simple. On the other hand, the observed physical behavior of the flow is non-trivial, especially in the 3D case. There, one can observe eddies, boundary layers and different instabilities (see Figure 5.2).

The variant chosen for benchmarking can be further described as follows: An incompressible Newtonian fluid in a cube is set into motion by a moving lid with a constant speed. The typical variants differ in the assumed direction and speed of the lid. In our cases, the speed is assumed to be one, and moving in direction of $u = (1, 0, 0)$ (see Figure 5.1). For these units, the Reynolds number is set to $Re = 1000$. At the beginning, the fluid is at rest, while the velocity is set to $u = 0$ everywhere. The fluid is then accelerated by the lid, that is moving with constant unit speed. For our tests, only 100 time steps are calculated. Although 100 steps are not enough to display any kind of interesting fluid behavior, it suffices for the purpose of speed measurement. Resulting runtimes are in the order of minutes, depending on the amount of CPUs and the chosen geometry size.

Several benchmarks were run for $N \in 100, 200, 400$, therefore with $h := 1/N$, leading to lattice sizes of 100^3, 200^3 and 400^3.

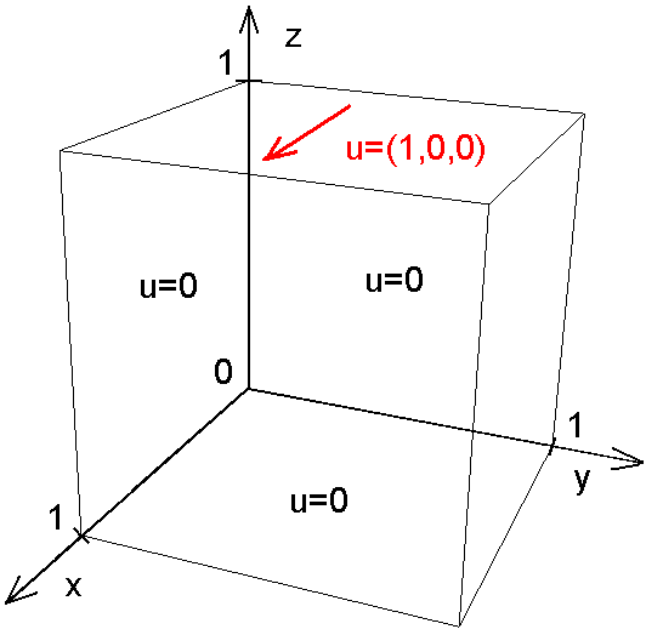

Figure 5.1.: The geometry of the *Lid-Driven-Cavity* benchmark - a cube with a lid moving in direction $(1, 0, 0)$ where all other cells are initialized to $u = 0$ at the beginning. These cells are then accelerated by the moving lid [21].

5.1.2. Bifurcation

During the past few years, one of the main applications of OpenLB at the KIT has been the simulation of flows in the human lungs and nose. The structure of the lungs is such that coming from the trachea, the lung is then divided into the two main bronchi, leading to the two halves of the lung. This first part is idealized in the geometry of the *bifurcation* benchmark, see Figure 5.3. The bronchi then divide several more times, until after some time leading to the bronchioles and after further branching leading to the alveolar sacs.

The further branching is implemented in an idealized way in the *bifurcation-large* benchmark (Figure 5.4), which has the two main bronchi, one of which is divided again into two parts. One of these parts is then divided again, finally resulting in two third level bronchi, two second level bronchi and the main bronchi.

Both problems are supplied as a *STL-file*, which is a typical file format in stereolithography CAD software. These files only describe the surface geometry and are read by a voxelizer that creates the `BlockGeometry` from the file.

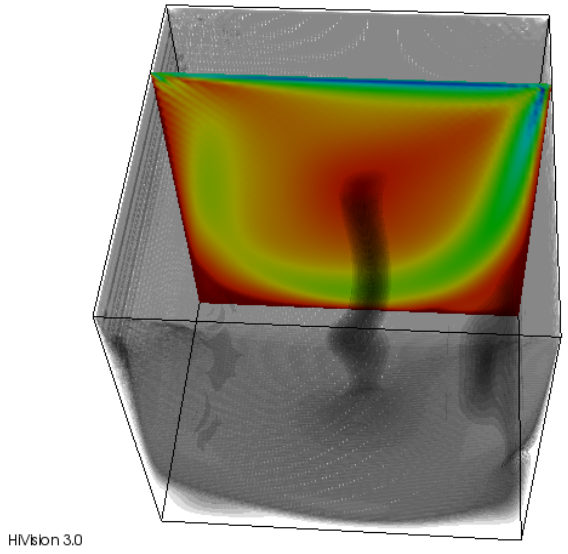

Figure 5.2.: Typical fluid behavior for *Lid-Driven-Cavity* benchmark with the development of a vortice [21].

The boundary of the idealized lung is not aligned to the directions, so it is initialized as an interpolation boundary. The trachea is initialized as a pressure boundary and the entries at the bronchi are velocity boundaries. As our benchmarks only intended to measure the performance for this type of problem, no speeds were initialized, which has no influence on running time. But to simulate the real fluid flow, one would iteratively raise the speed at the entries of the bronchi.

The bifurcation example was run for 100 iterations, the same as the cavity benchmark. As for the bifurcation-large benchmark, it was run for 300 iterations since this benchmark needed more memory. As the IC1 also has less memory, the examined geometries had to be smaller (See section 5.2). Still, with even more iterations, one achieves run-times of just over a minute for one core.

5.2. Description of Test-Machines

The benchmarks were run on two parallel computers, often referred to as clusters, which are described below. Both these clusters are located at the Steinbuch Centre for Computing (SCC) located at the KIT.

Jobs on these clusters are governed by a central batching system. Each node is

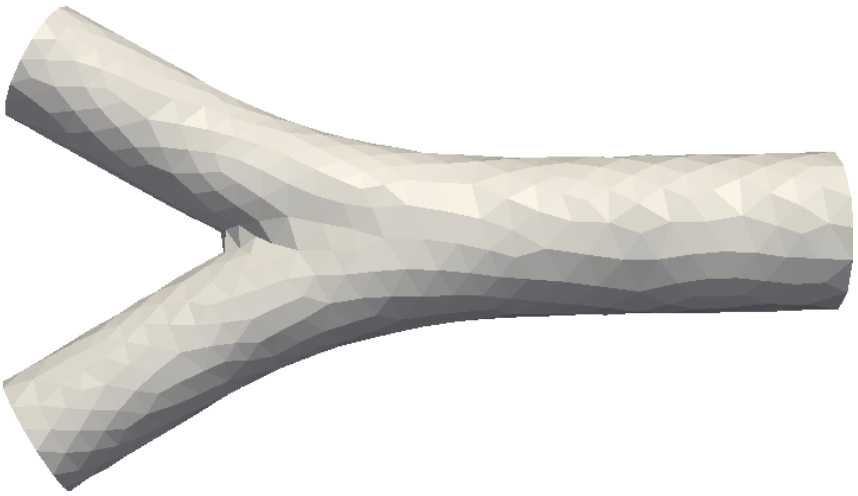

Figure 5.3.: An image of the bifurcation geometry

used exclusively during all tests for the benchmarks run. Utilizing the batching system support for a strict enforcement of maximum memory, one can assign certain numbers of processes to each node. In all the tests, memory requirements were set in such a way, that each process ran on one core and that no core was left unused, if possible. While being more conservative with the used resources, this could possibly induce certain variations in the results. One possible case might be that since lower levels of cache are often shared between cores of each processor, there might be competition between the cores leading to an increased amount of cache misses. Nevertheless, this way was chosen because adverse effects were not immediately visible and because it resembles real world usage much more closely. Generally, in the further parts of this thesis, core and processor will be used interchangeably due to this choice.

To get some more numbers, the tests for the comparison between full and empty cells were also run on the author's desktop computer.

5.2.1. HP XC3000

The HP XC3000 (HC3) is a "high performance and throughput computer" [4]. It is comprised of different kinds of nodes, of which only the compute nodes are relevant for the tests executed. There are different kinds of compute nodes, of which we only used the HP Proliant DL170h compute nodes to prevent variations because of differing architectures between runs.

Each of the 312 8-way DL170h nodes consists of two quad-core Intel Xeon processors E5540 (Nehalem architecture) which run at a clock speed of 2.53 GHz and have

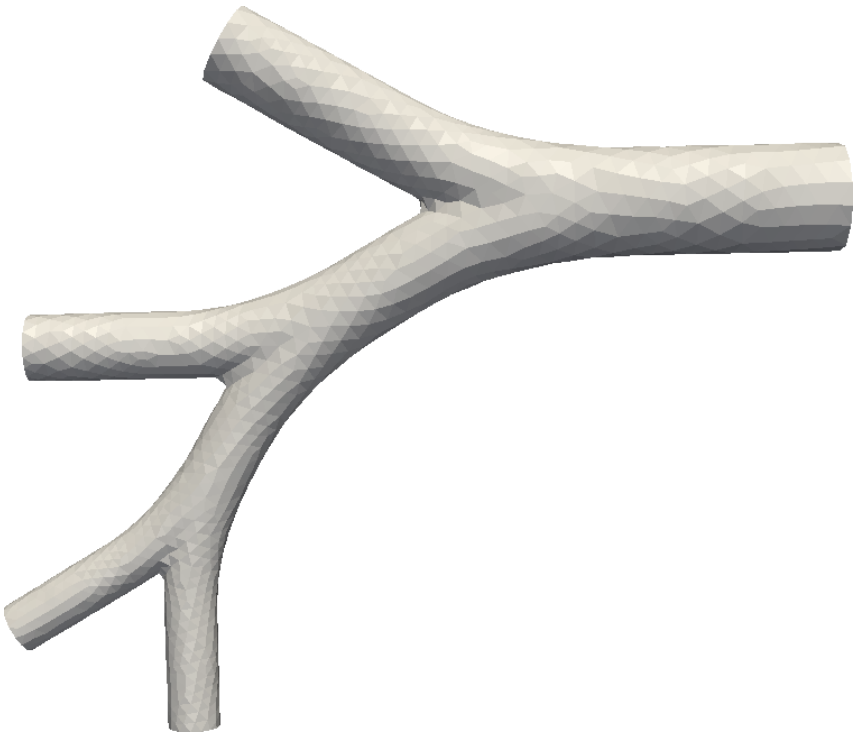

Figure 5.4.: An image of the bifurcation-large geometry

4x256 KB of level 2 cache and 8 MB level 3 cache. Every node has 24 GB of main memory as well as a local disk with 250 GB and is connected to the other nodes via an InfiniBand 4X QDR interconnect.

The InfiniBand interconnect is characterized by a very low latency of just 2 microseconds and a bandwidth node to node of more than 3100 MB/s. This architecture makes the cluster well suited for communication intensive applications utilizing lots of MPI communications [4]. The operating system running on the HC3 is Red Hat 5.8. The compiler used to compile the test programs was the GCC 4.5.3 with an optimization level of O3; the MPI version used was HP-MPI 2.3.1.

5.2.2. Institutscluster

The "Institutscluster" (IC1) is a parallel computer hosted at the SCC for different departments at the KIT. The cluster consists of the same four node types as the HC3, login nodes, compute nodes, file server nodes and administrative server nodes. As we only used the compute nodes during program execution, only those are described here.

The IC1 consists of 200 8-way Intel Xeon X5355 nodes. Each of these nodes

contains two quad-core Intel Xeon processors with a clock speed of 2.667 GHz and 2x4 MB of level-2 cache each. Every node has 16 GB of main memory as well as four local disks with 250 GB each. The nodes are connected to each other via an Infiniband 4X DDR interconnect.

The Infiniband interconnect has a latency from node to node below 2 microseconds and a point to point bandwidth of 1300 MB/s [1].

Programs on the IC1 were compiled with the GCC 4.5.3 with an optimization level of O3 and using the MPI library OpenMPI 1.5.4.

5.2.3. Desktop Computer

This personal computer consists of an Intel Core 2 Duo E6600 with two cores at a clock speed of 2.40 GHz, an Intel 82975X Memory Controller and 8 GB of DDR2 RAM.

5.3. Results

To compare different implementations of LB, in most cases one uses the measurement unit of *million fluid-lattice-site updates per second* MLUP/s, e.g. [32]. This idea can be extended to the unit MLUP/ps for *million fluid-lattice-site updates per process and second* [21]. The latter unit shall be used in all examples. For this, the amount of fluid cells N_c for each of the examples is calculated and with the run-time t_p for p processor cores, i the number of iterations, the result is given as

$$P_{LB} := 10^{-6} \frac{iN_c}{t_p p}$$

5.3.1. Measurements

As was explained in the sections about the different benchmarks, each benchmark was run for several iterations. This was 100 for the *bifurcation* and *lid-driven cavity* examples and 300 iterations for the *bifurcation-large* benchmark. The benchmarks cavity and bifurcation were run on the HC3 and the ones for bifurcation-large were run on the IC1 if not explicitly declared otherwise.

Due to the internal structure of the clusters HC3 and IC1, fluctuations in run-time can sometimes be observed that make up to 15% of the total run-time. To mitigate this effect, most tests were run five times and the best result was selected as being representative. While this does not meet the requirements for statistical significance, slowed down test cases stand out and were, if needed, verified on a case by case basis. This trade-off had to be made due to constraints in the amount of CPU time usable.

ToDo **(Begründung so in Ordnung?)**

Source	Target	$\sum$ Edge Weight	Bytes Communicated	Ratio
0	6	4 558	207 844 800	45 600
0	2	5 353	244 096 800	45 600
0	7	53	2 416 800	45 600
15	11	5 300	241 680 000	45 600
9	11	4 505	205 428 000	45 600

Table 5.1.: Communication between nodes compared to the sum of edge weights between their assigned cuboids. This shows that the communication between nodes is perfectly proportional to the assigned weights. The amount of communication for 300 iterations of the *bifurcation-large* benchmark is shown. When dividing per weight unit and iteration, one gets 152 bytes for each communicated cell. This is the amount expected for 19 doubles, which is exactly the amount of data of a cell in a D3Q19 model.

Also, all graphs in this section connect the different data points to clarify their connection. This is by no means meant to imply the possibility of interpolation but only for visual clarification.

5.3.2. Graph Based Load Balancer

The Graph Based Load Balancer (GBLB) was tested extensively on all the benchmarks, as it is also the load balancer of choice for the decomposition strategies. This section will only use the standard decomposition algorithm and validate the assumptions made in the design of the GBLB. It will also show that even with the standard decomposition, improvements in run-time can be measured.

5.3.2.1. Validation of Model Assumptions for Communication

In the Subsection 4.1.1, weights for cuboids as graph nodes and for the communication as edges in the graph had to be derived. Validation for the work of each cuboid was inherent in the calculation of the weight as we conducted measurements for the amount of calculations for empty and full cells. The validation for communication is presented in this section.

The HP-MPI built-in measurement facilities were used to measure communication between different cluster nodes. The files produced by this measurement include the exact numbers of bytes transferred between each of the cluster nodes. From this, the transfers for the setup were subtracted, as only the communication during the actual calculation is relevant. The results were then compared to the calculated weight for the edges that connected the cuboids running between the nodes. An excerpt of this

data for a test run of the bifurcation-large example with one cuboid per processor for 16 processors is shown in Table 5.1.

These results show that the calculation of edge weights is proportional to the communication. The calculated number of 45 600 is exactly the amount expected, as these tests were done on 300 iterations of the bifurcation-large example. This means that 152 bytes were communicated for each iteration and edge weight unit. 152 bytes are exactly 19 doubles on the test machines, and exactly 19 doubles are used per cell in the D3Q19 model. Therefore, the used method for calculating edge weights is correct.

5.3.2.2. cavity3d

The first test for the GBLB was the cavity3d benchmark. This benchmark was used to show some general characteristics of the LB implementation which are also exhibited for load balancing with the GBLB. This test was always run with the $k \times p$ cuboids, that is with k cuboids for each of the p processors. These tests were run for different sizes of h, resulting in cubes of the dimensions 100^3, 200^3 and 400^3 cells for the cavity3d benchmark.

As one can see from Figure 5.5, the GBLB performs at the same speed as the simplistic *traditional load balancer* (TLB) for $k = 1$. One can also see that the best performance per processor for this example is achieved with this same setting, that is with one cuboid per processor. This is due to the minimal communication within the cores and also between cores, as more cuboids always result in greater communication, even when the cuboids are later assigned to the same core.

Although higher values of k are not beneficial in this specific benchmark, one can also see that the GBLB often performs better for more cuboids per core. This is despite the fact that the assumption of the TLB that volume is equal to computation time still holds for this specific example.

Figure 5.6 shows the comparison between different sizes of the benchmark. One can see that for larger cuboids (resulting from the larger geometry), the efficiency in form of MLUP/ps is higher. This is because there are less boundary cells in relation to internal cells, resulting in relatively fewer communication, which is the slower. From this we know that bigger cuboids are the better choice, so one optimization criterion for domain decomposition should be maximizing the cuboids.

5.3.2.3. bifurcation

On the first view, this benchmark does not seem as clear as the cavity3d example. In Figure 5.7, one can see two rises of the MLUP/ps for the $k = 4$ plots and one for

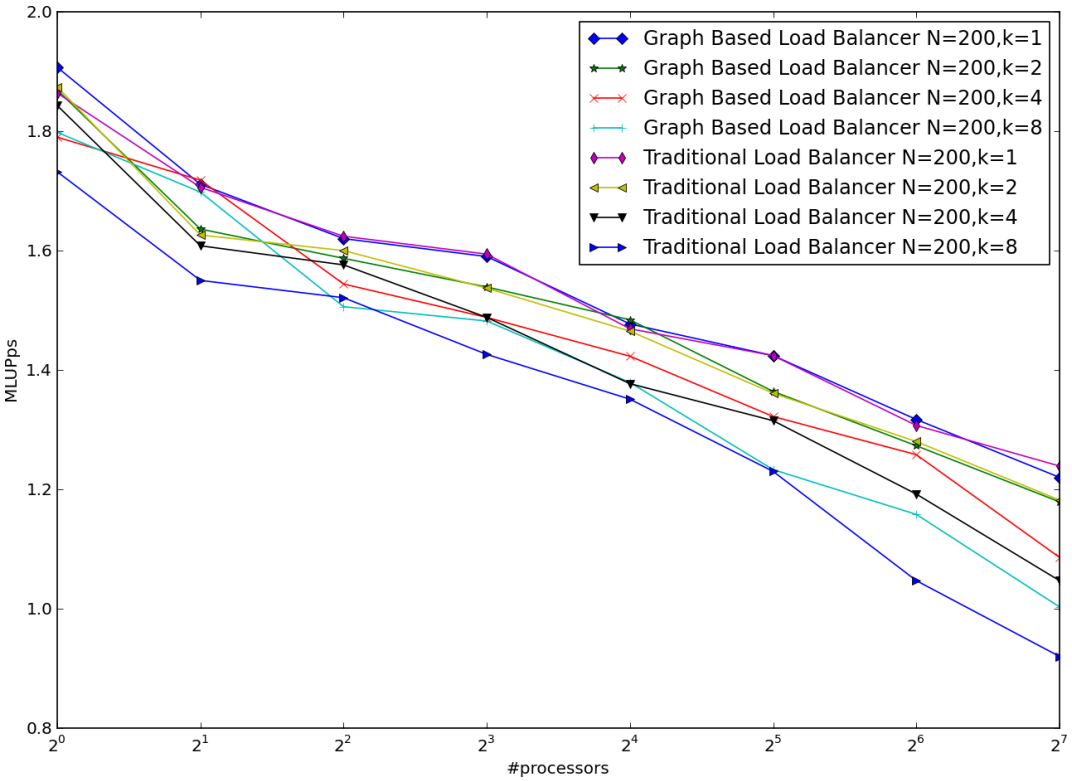

Figure 5.5.: Graph comparing *traditional* with *graph based load balancer*. k is a factor for the number of cuboids, i.e. $nC = k \times \text{numProc}$. The graph shows that the graph based load balancer is at least as good as the traditional one in nearly all cases.

$k = 1$. As we recall, in section 4, we deduced that empty cells required computing time as well. Due to this, a step implemented in the examples removes completely empty cuboids from the scheduling, which explaines the rises in computing speed when distributing on more cores and nodes. As soon as a critical amount of cuboids is generated, also completely empty cuboids are generated, which are then removed and not "wasting" cpu time any more. One can therefore already get an idea of the possible speed-ups due to *shrinking*.

Also, the $k = 1$ case, which was always faster for the cavity benchmark, is now not optimal, for nearly all cases beyond a certain amount of CPUs. The reason for this is that some cuboids are removed, and when there is only one cuboid per CPU, some CPUs will be left idle. On the other hand, for the example for 64 processes, there are only 44 cuboids left for the case $k = 1$, but there are 154 cuboids for the

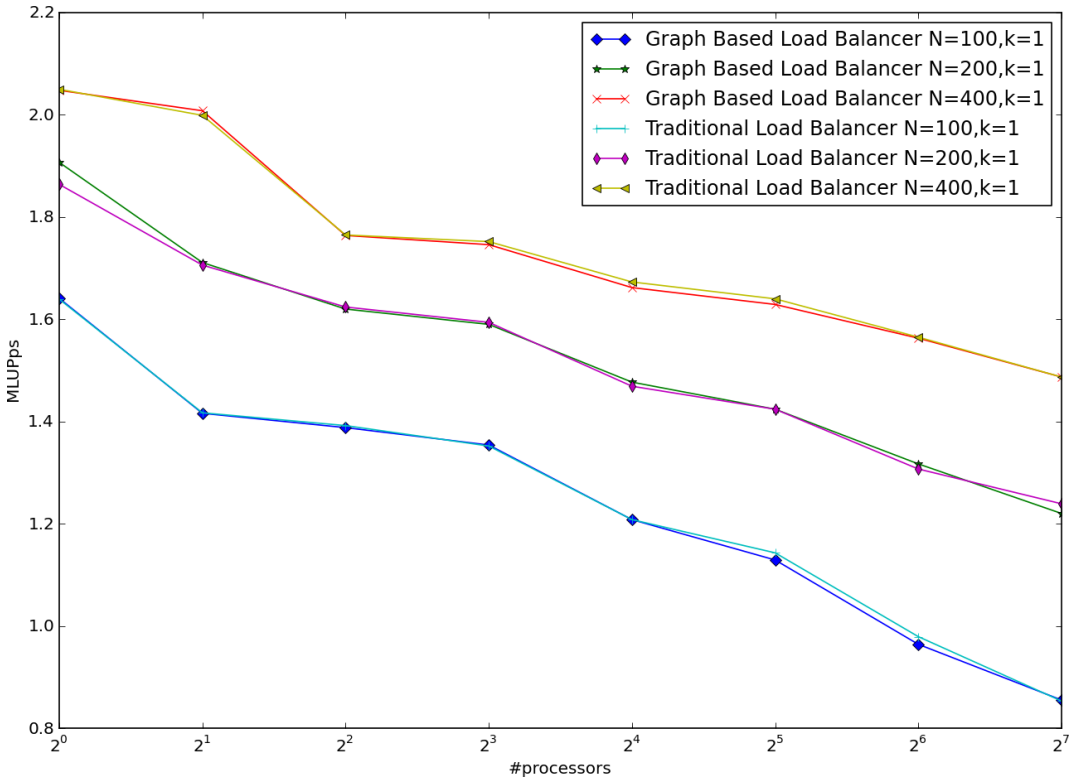

Figure 5.6.: Graph comparing *traditional* with *graph based load balancer* for different sizes of geometry, 100^3. 200^3 and 400^3 cells and one cuboid per core. Bigger cuboids result in higher MLUP/ps due to more computing compared to communicating.

case $k = 4$. In this case, it seems that the overhead of the many cuboids per CPU is as bad as leaving some CPUs completely idle. This is probably due to the fact that most cuboids are about the same size, so that each of the 64 CPUs gets at least two work units with some receiving three. As all processors have to synchronize for each iteration, this reduces the performance of all CPUs to about 2/3. Intuitively, this should have approximately the same effect as letting about 1/3 of the CPUs idle.

5.3.2.4. bifurcation-large

As we noticed for the bifurcation benchmark, small values of k are not very efficient because they allow some processors to run empty. Therefore, only higher values of k are graphed in Figure 5.8.

This example shows the initial decline of computation speed from one CPU to eight CPUs. This decline of speed has been observed before by [21] and is due to

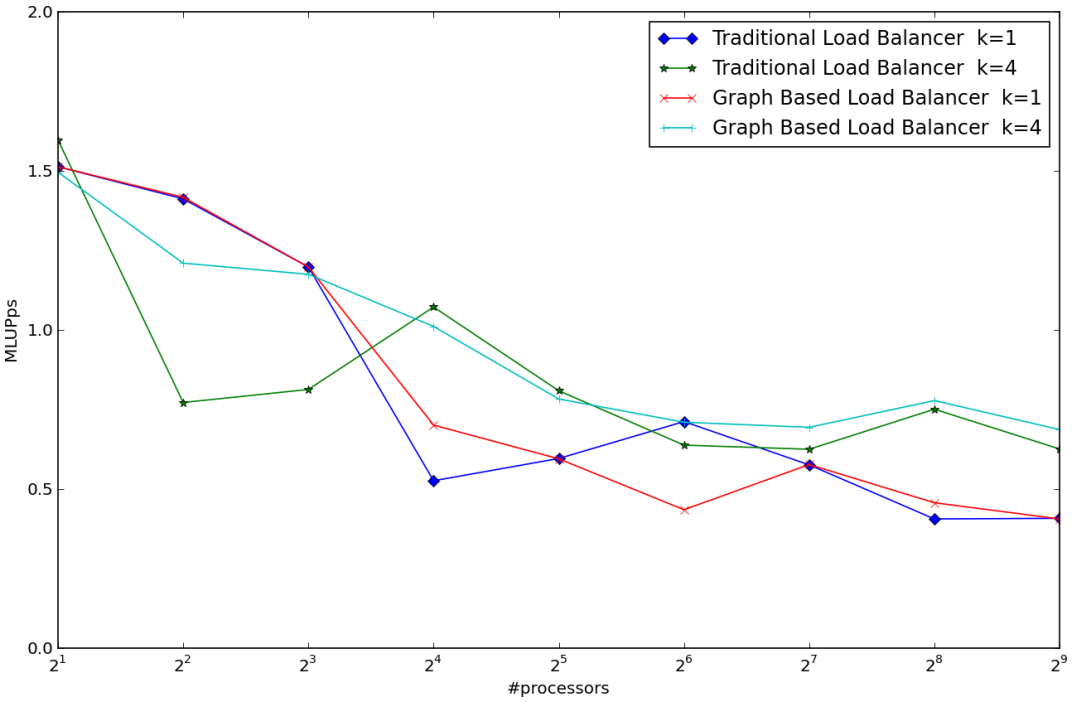

Figure 5.7.: Graph comparing *traditional* with *graph based load balancer*. This graph exhibits some typical behavior due to the removal of completely empty cuboids from the scheduling resulting in rising MLUP/ps with more processors.

the memory bound characteristics of the LB algorithm. This behavior is especially pronounced for the bifurcation-large benchmark as this example runs on the IC1. The IC1 has eight-way nodes which all share the same memory bus and so the effect is even greater as before because the HC3 has larger amounts of L2 cache for the processors as well as an L3-cache.

While one can see that the results are all not that far apart, starting in the range of approximately 128 processors the GBLB becomes more efficient by a margin. To show how much more efficient the load balancer performs for higher numbers of CPUs, see Table 5.2. The GBLB solution is about 34% faster than the solution with the TLB in the largest test with 512 CPUs.

The table 5.2 also contains the number for $k = 1, 2$. For these two values, the decomposition results in less cuboids than cpus. This combined with the maximum part of the balancing constraint results in more than one cuboid being assigned to some cpus, leading to a worse solution than the TLB. This was not optimized further,

k	MLUP/ps GBLB	MLUP/ps TLB
1	0.067	0.117
2	0.044	0.223
4	0.303	0.198
8	0.297	0.226
16	0.26	0.199
32	0.137	0.134
64	0.068	0.022

Table 5.2.: This table shows the speed of the *Graph Based Load Balancer* and the *Traditional Load Balancer* in MLUP/ps with 512 processors for the bifurcation-large benchmark. The GBLB is about 34% faster than the solution with the TLB.

because this case is not desirable anyway, as resources would be wasted with CPUs being left empty even with a perfect load balance.

5.3.3. Heuristic Load Balancer

The general results for the *graph based load balancer* have been very good. So the question was if the heuristic performs at a similar level. As computation time was limited, only certain test cases were run with the heuristic load balancer, which are presented in Table 5.3. From this table, one can see that the heuristic does not fair much better than the traditional load balancer when using the old domain decomposition. Also, the true speed up of the GBLB lies with more cores. While the optimum lies at different values of k, the absolute values are not much better of the heuristics than of the TLB.

The picture quickly becomes more clear when combining the new domain decomposition techniques with the heuristic load balancer. The results can be seen in Table 5.4. Clearly, the heuristic load balancer performs at a level in between the TLB and the GBLB. Its most important feature clearly is that it also supports the different cuboid sizes produced by the new decomposition strategies.

5.3.4. Octree Domain Decomposition

Tests for the Octree Domain Decomposition were done on the most complicated of the benchmarks, the *bifurcation-large* example. Example decompositions that utilized the octree algorithm for the bifurcation-large example can be seen in the following figures:

- For an $8 \times 8 \times 8$ minimum cuboid, see Figure 5.9

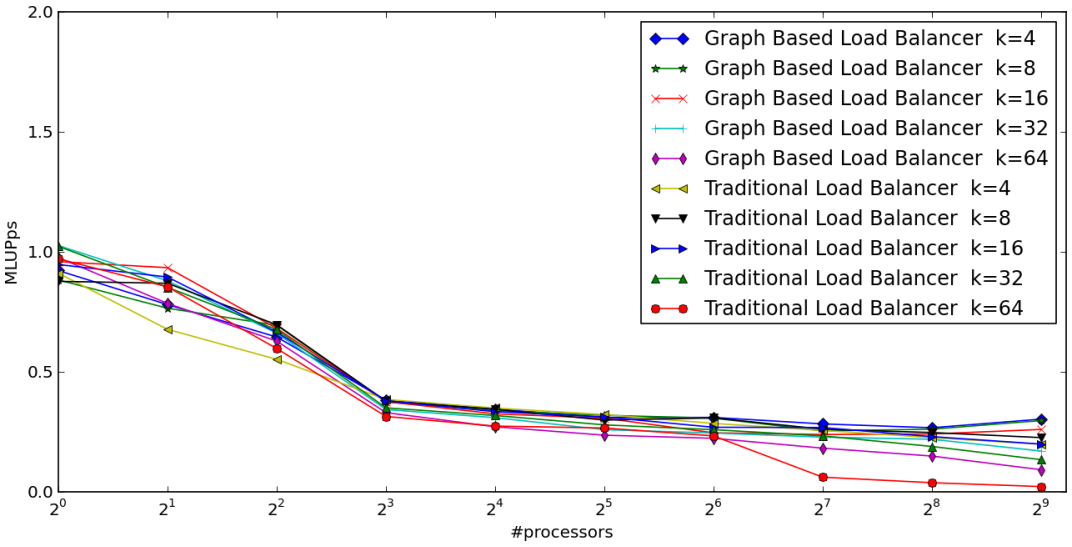

Figure 5.8.: Comparing *traditional* with *graph based load balancer* for the *bifurcation-large* test case. k is a factor for the number of cuboids. $nC = k \times$ numProc. This shows that the graph based load balancer in general performs better than the traditional one, especially for larger cases.

- For an $16 \times 16 \times 16$ minimum cuboid, see Figure 5.10
- For an $32 \times 32 \times 32$ minimum cuboid, see Figure 5.11
- For an $64 \times 64 \times 64$ minimum cuboid, see Figure 5.12

From these examples, one can see that the smaller the minimum cuboid, the better the domain decomposition approximates the underlying geometry. All these examples are conducted on a geometry of the bifurcation-large benchmark with a resolution of Nx: 428, Ny: 66 and Nz: 503.

As one can easily deduct from the pictures, the amount of cuboids generated increases the smaller the minimum cuboid. To give an overview over how many cuboids are left after removing the completely empty ones, please refer to Table 5.5. Due to the huge amount of cuboids generated for smaller minimum sizes, these minimum sizes only work with large amounts of processors. But there, they enable a very good load balancing between the different processors as work units are very small. Problematic might be that the communication increases as well with the number of cuboids, so performance probably depends on the specific underlying geometry.

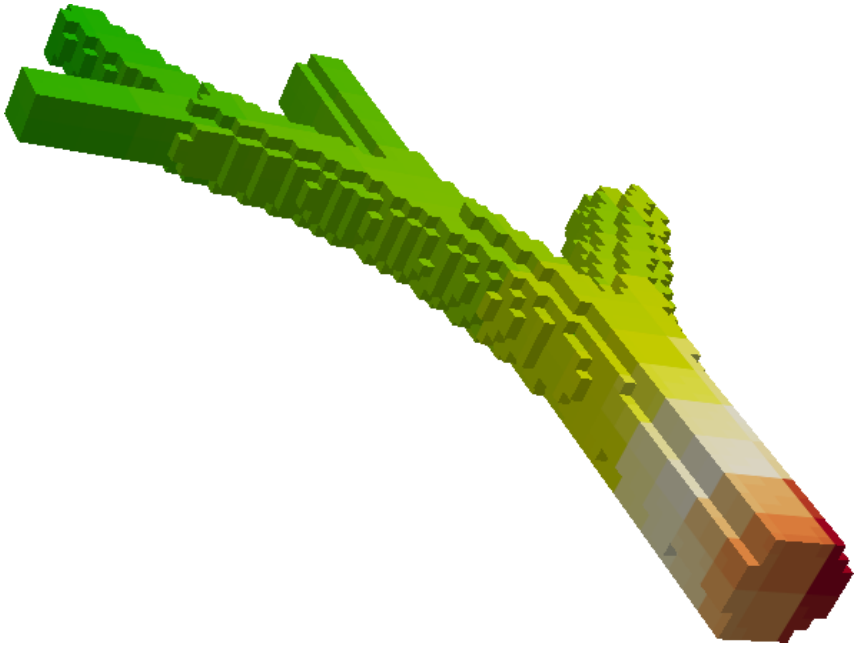

Figure 5.9.: *Octree Decomposition* of the *bifurcation-large* example with a minimum cuboid size of $8 \times 8 \times 8$.

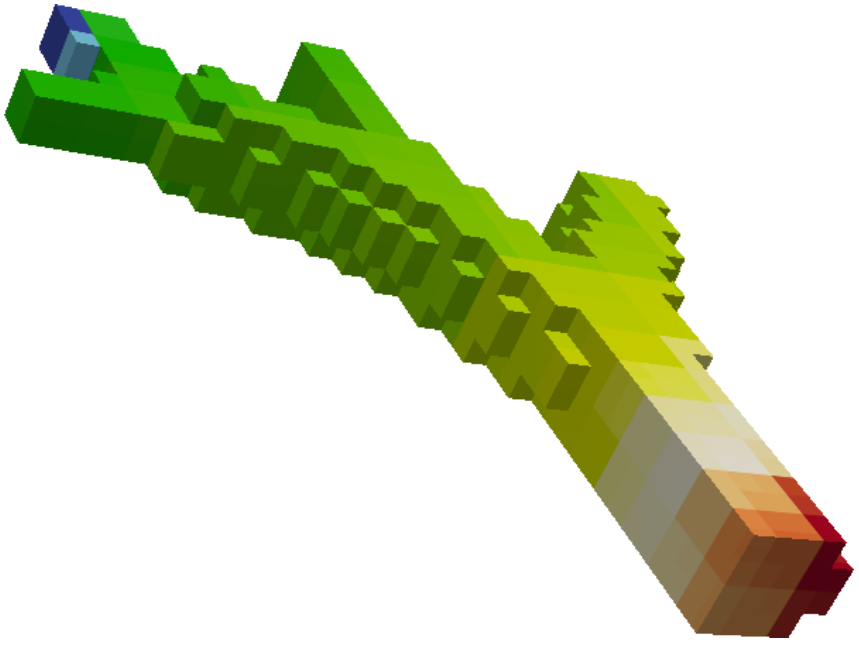

Figure 5.10.: *Octree Decomposition* of the *bifurcation-large* example with a minimum cuboid size of $16 \times 16 \times 16$.

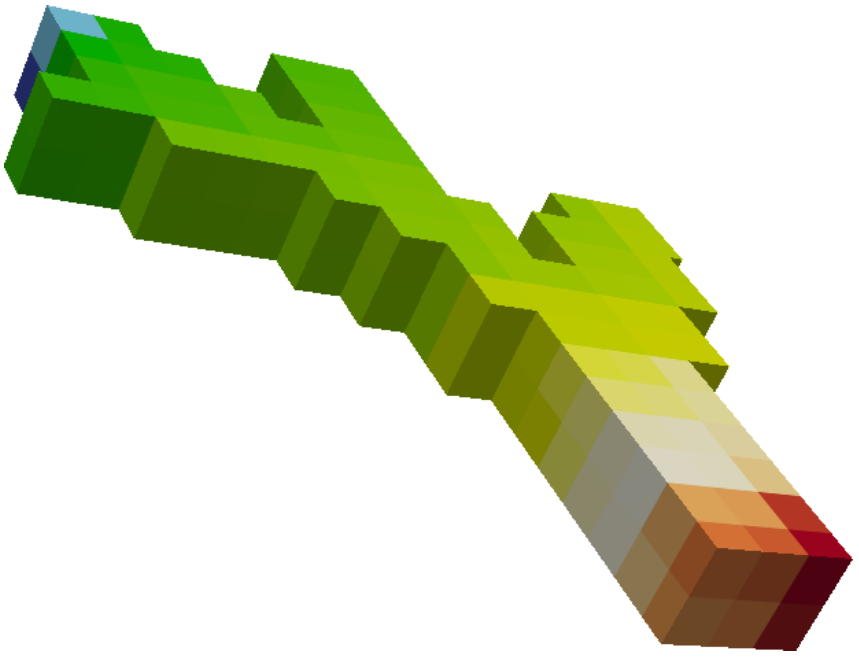

Figure 5.11.: *Octree Decomposition* of the *bifurcation-large* example with a minimum cuboid size of $32 \times 32 \times 32$.

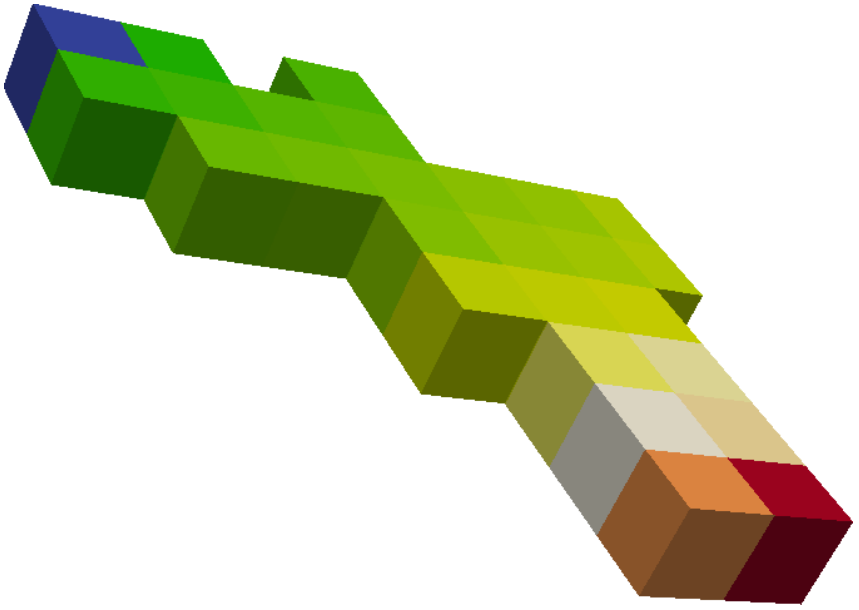

Figure 5.12.: *Octree Decomposition* of the *bifurcation-large* example with a minimum cuboid size of $64 \times 64 \times 64$.

# processors	k	Old Dec. TLB	Old Dec. GBLB	Old Dec. HLB
32	4	0.322	0.310	0.349
32	8	0.299	0.319	0.324
32	16	0.312	0.307	0.331
256	4	0.226	0.267	0.239
256	8	0.247	0.261	0.248
256	16	0.23	0.241	0.178
512	4	0.198	0.303	0.236
512	8	0.226	0.297	0.183
512	16	0.199	0.260	0.139

Table 5.3.: Comparing the speed in MLUP/ps of *Heuristic*, *Graph Based* and *Traditional Load Balancer* for the bifurcation-large benchmark with a certain subset of possible variations. All these examples use the old domain decomposition algorithm. k is the multiplicator for the number of cuboids per processor.

A bigger example shall be shown as well, discussing some more of the effects of the octree decomposition: For a larger version of the bifurcation-large benchmark – Nx: 684 Ny: 105 Nz: 802 – with a minimum cuboid size of 8 we get the amount of 11043 cuboids. These cuboids have sizes between 192 and 33792 cells. For a minimum size for cuboids of 8, that is a minimum cube of $8 \times 8 \times 8$, one would expect a minimum size of 512 cells. Due to the alignment of the mother cuboid, the volume can be significantly less. The mother cuboid is aligned, so that it shares three edges with the surrounding box of the geometry. Therefore, some cubes may be at the opposing border of the surrounding box. These will be cut off for efficiency, as they would unnecessarily waste memory. With a minimum size of 16, we get 2430 cuboids with a minimum of 2048 cells and a maximum of 33792. One can imagine that this would be a good amount of cuboids for about 512 to 1024 CPUs.

Comparing the speeds in Figure 5.13, one can observe that, most likely due to the overhead of the shear amount of cuboids, the octree decomposition is only able to compete with the traditional load balancer and domain decomposition in a few cases. But the graph also suggests that for larger numbers of CPUs, the octree might actually surpass the performance of the original decomposition and load balancer. Two values for the octree with minimum cuboid size of $c = 16$ could not be calculated due to errors

on the cluster at that time.

# processors	k	Old Dec. & TLB	Shrink & GBLB	Shrink & HLB
32	4	0.322	0.708	0.692
32	8	0.299	0.694	0.629
32	16	0.312	0.578	0.584
256	4	0.226	0.587	0.391
256	8	0.247	0.504	0.380
256	16	0.23	0.419	0.252
512	4	0.198	0.303	0.301
512	8	0.226	0.297	0.263

Table 5.4.: Comparing the speed in MLUP/ps of *Heuristic*, *Graph Based* and *Traditional Load Balancer* for the bifurcation-large benchmark with a certain subset of possible variations. All these examples use the old domain decomposition algorithm, but the GBLB and the HLB benchmarks both utilize the shrinking step in addition.

Minimum Cuboid	Number of Cuboids
8	4056
16	842
32	157
64	26
128	9
256	4

Table 5.5.: This table shows the number of cuboids generated for the bifurcation-large example with the dimensions Nx:428, Ny: 66 and Nz: 503 for each minimum cuboid size.

These observations show that more tests need to be done, especially for larger numbers of cores and maybe even bigger geometries, as those will have larger internal cuboids, leading to better performance on those. So evidence suggest that octree decomposition is best suited to very large and complex geometries. Also, one probably needs a certain amount of CPU to offset the huge numbers of small cuboids. Improvements might be possible by shrinking the cuboids even further or maybe by recombining them and then applying a shrinking step, using heuristics.

Last, some work has still be taken on by the implementor to find the right parameters for each specific fluid flow simulation.

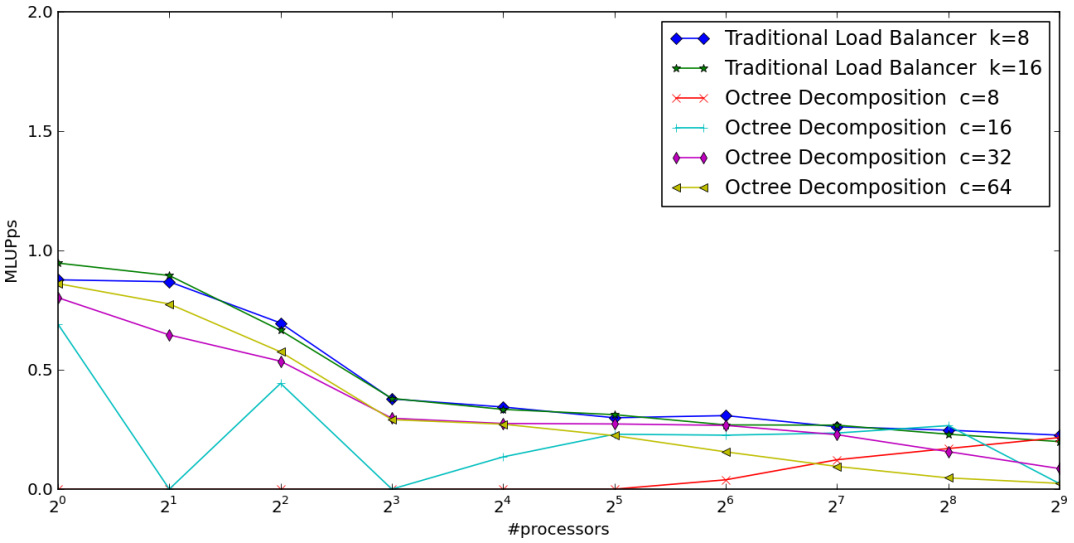

Figure 5.13.: Graph comparing *Traditional Load Balancer* with classic domain decomposition with the *Octree Decomposition* and the Graph Based Load Balancer. For the octree, c is the minimum side length of the cubes, for the traditional load balancer it is the factor for the number of cuboids in relation to the number of processors.

5.3.5. Shrinking Cuboids For Geometry Approximating Sparse Domain Decomposition

The *shrinking step* as a step to optimize the size of the cuboids showed itself to be the most promising and performance raising strategies of all, despite its seemingly simple nature. Performance for the *bifurcation-large* benchmark increased by over 100% in certain cases.

All these tests were run with the graph based load balancer. These tests included processor counts between 1 and 256 and a factor $k \in \{1, 2, 4, 8, 16, 32, 48, 64\}$ for the amount of cuboids relative to the processor count. An excerpt of the performance for the best performing traditional load balancer and the best graph based load balancer test runs with an additional shrinking step are shown in Figure 5.14. One can see that the solution with the shrinking step outperforms the traditional load balancer for all values of k. Runs for $c = 32$ were not performed for $p > 64$ as memory requirements were too tight for this test case and the trend suggests that results are not competitive. The same holds true for the test case $p = 256, k = 16$. Also, once more, we can see the effect of the removal of completely empty cuboids at work for

$p = 1, 2$, as the variants with $k >= 16$ are faster there, because they already have enough cuboids to contain completely empty ones.

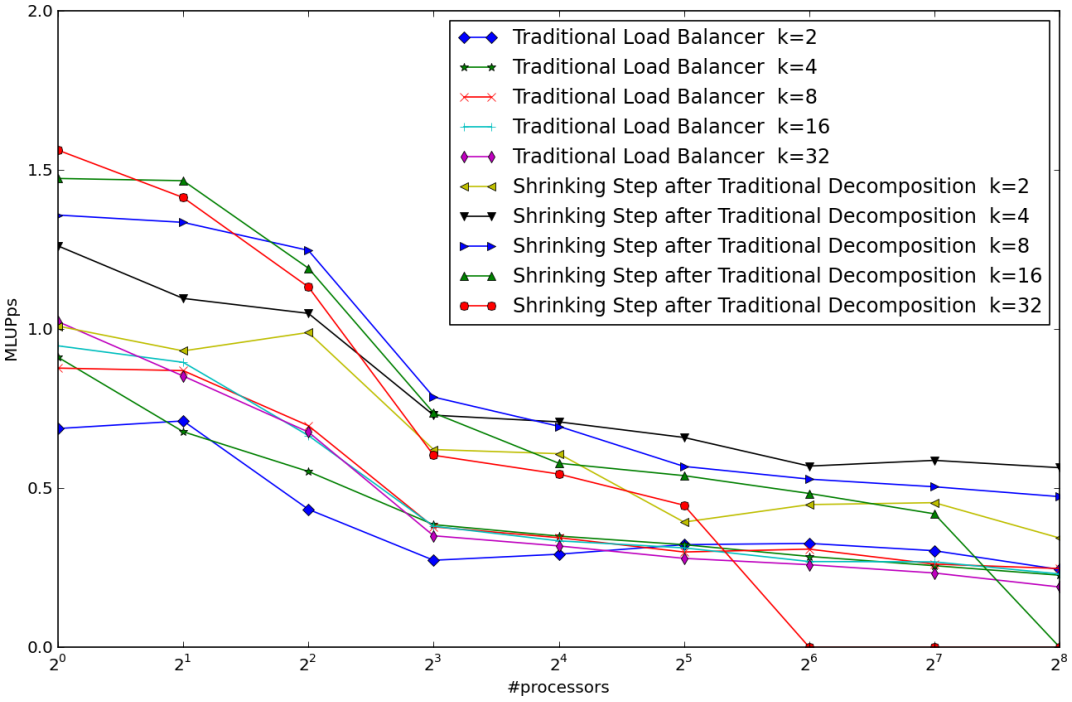

Figure 5.14.: Graph showing the improvement in run-time of the *bifurcation-large* benchmark for *shrinking* with the *graph based load balancer* compared to the *traditional load balancer* with the *traditional decomposition*. k is the factor for the number of cuboids in relation to the number of processors. Speed is given in MLUP/ps.

A numeric comparison for this test case is shown in Table 5.6. There one sees that the shrinking step always results in faster execution than just utilizing the traditional load balancer.

This effect can be attributed to multiple effects. Because empty cells are excluded from the streaming step, less streaming is required. An overview of what kind of reduction in the amount of empty cells could be achieved can be seen in Table 5.7.

As one can deduce from these numbers, the speed-up can not solely be explained by the smaller amount of empty cells. The graph load balancer certainly has its part as was showed in the prior tests. But most likely several secondary effects are at work as well. The ratio of memory accesses to CPU work shifts towards the calculation side when the empty cells are removed, as the empty cells require no

# Processors	Traditional	Shrinking	Performance Improvement
1	1.023	1.562	52.7%
2	0.895	1.466	63.8%
4	0.696	1.247	79.2%
8	0.385	0.786	104.2%
16	0.349	0.708	102.9%
32	0.322	0.659	104,7%
64	0.326	0.569	74.5%
128	0.303	0.587	93.7%
256	0.247	0.473	91.5%

Table 5.6.: Speeds in MLUP/ps for the best variant for each processor with the *traditional load balancer* and domain decomposition compared to the *shrinking step* and the *graph based load balancer*. In the last column, the speed-up with the new methods is shown.

# Processors	k	# cuboids	# cells before shrinking	# cells after shrinking
128	4	185	14115849	11665913
16	4	33	14164128	9878364
32	4	58	14205510	10717376

Table 5.7.: Comparison of the amount of cells that are computed before and after executing the *shrinking* step on a traditional decomposition. k is the multiplier for the amount of cuboids in relation to the number of processors.

computation and are only memory intensive. Because of this, the memory hierarchy is put under less strain, so the caches might work more effectively. Another effect is that the communication between cuboids assigned to different nodes is reduced as when the cuboids are shrunk, their surface area shrinks, too. Therefore, the amount of communication needed for these cuboid is reduced as well.

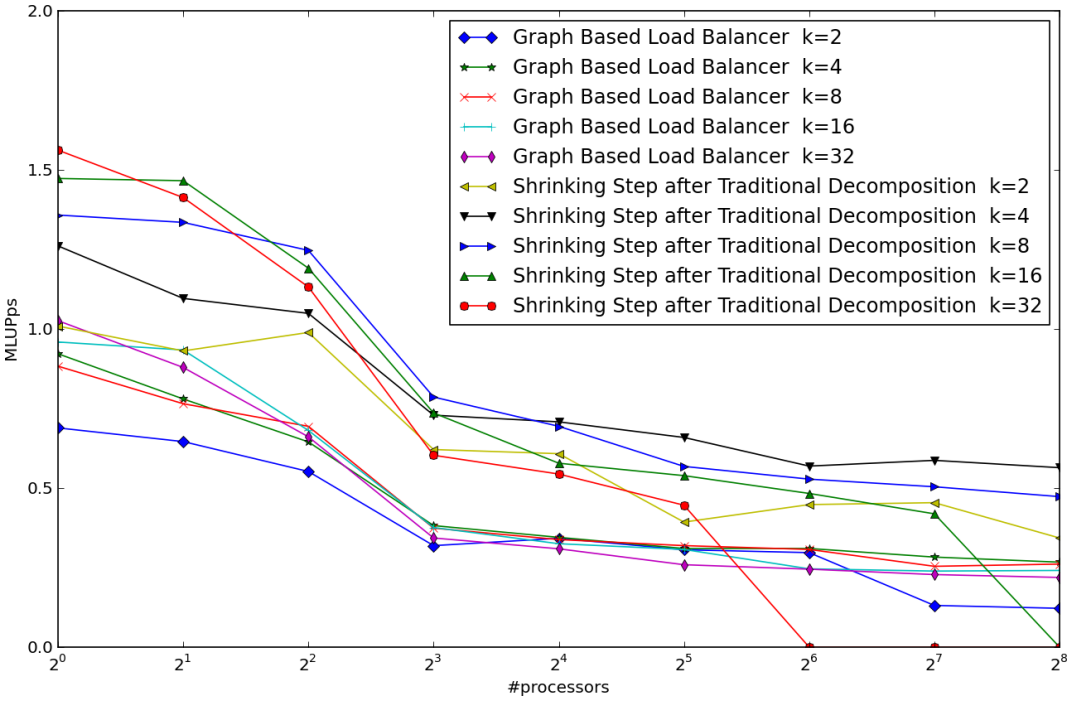

Figure 5.15.: Graph showing the improvement in run-time on the *bifurcation-large* benchmark for *shrinking* with the *graph based load balancer* compared to the graph load balancer combined with the *traditional decomposition*. k is the factor for the number of cuboids in relation to the number of processors. Speed is given in MLUP/ps.

In Figure 5.15, the results are shown that compare the graph based load balancer with the traditional decomposition to the additional shrinking step and the graph based load balancer applied. Obviously, the shrinking and GBLB combination is still even faster.

In the following figures, three example shrinking steps are shown, each with the different decomposition strategies employed. Figure 5.16 shows what the cuboid structure is reduced to for the bifurcation-large benchmark when applying the shrinking step after the octree decomposition with a minimum cuboid size of 32. Figure 5.17

shows the same situation for the minimum cuboid size of 64, and Figure 5.18 shows the result for the traditional decomposition with a starting number of 270 cuboids. These 270 cuboids are reduced to 112 due to the removal step.

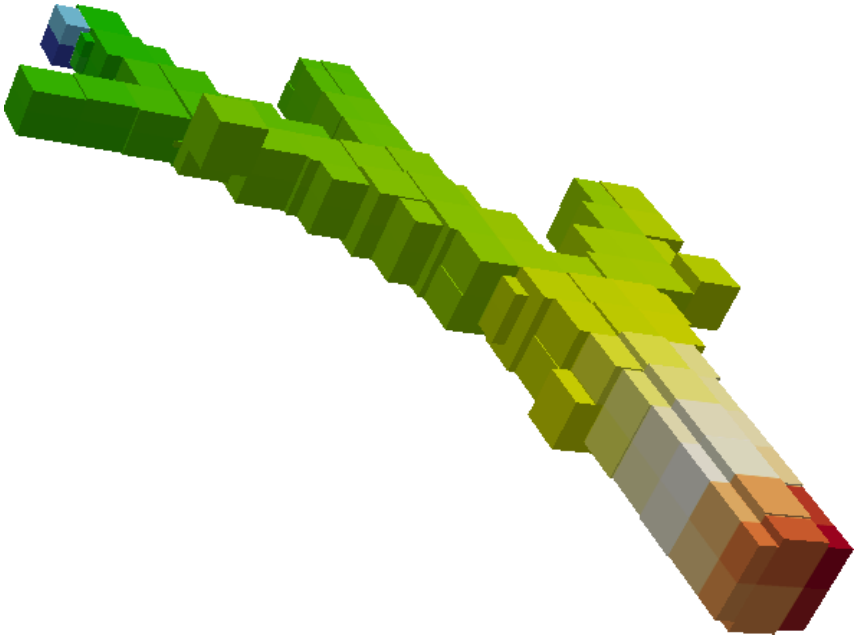

Figure 5.16.: *Octree Decomposition* of the *bifurcation-large* example with a minimum cuboid size of $32 \times 32 \times 32$ and an additional shrinking step.

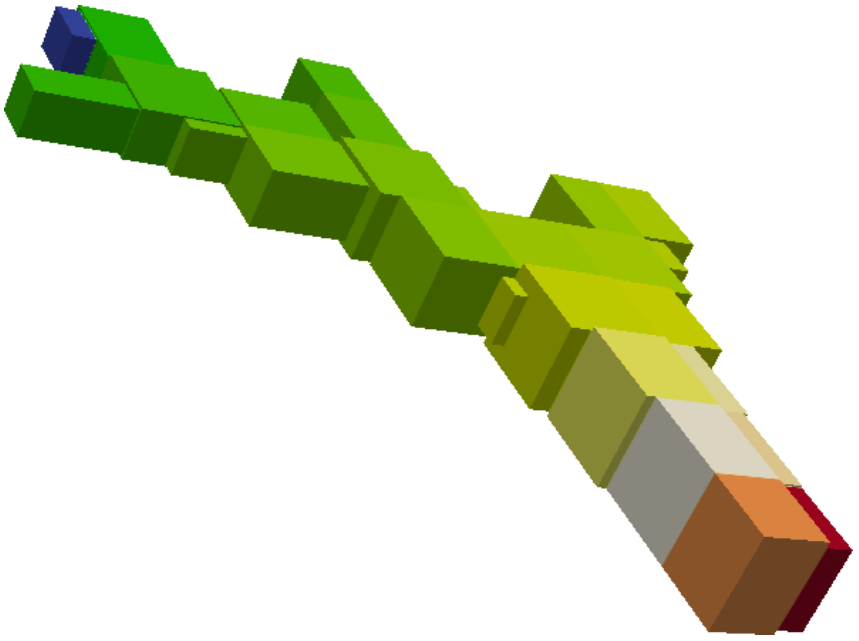

Figure 5.17.: *Octree Decomposition* of the *bifurcation-large* example with a minimum cuboid size of $64 \times 64 \times 64$ and an additional shrinking step.

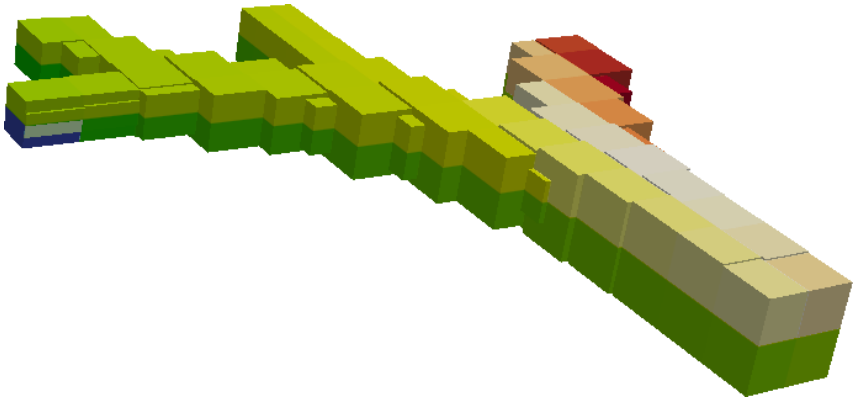

Figure 5.18.: Traditional decomposition of the *bifurcation-large*into 270 cuboids (of which 112 are non-empty) and an afterwards applied shrinking step.

6. Conclusion

6.1. Summary

In this thesis we researched potential optimizations for Lattice Boltzmann Methods on the example of the OpenLB implementation. The analysis of the current situation identifies two major areas with potential for major improvement. The first is the current load balancer in OpenLB, and the second is the simple heuristic sparse domain decomposition. The implementation of the old load balancer does not take into account the communication between cuboids. It also severely limits the potential optimizations for sparse domain decomposition. Because of this, we designed and implemented two alternative load balancers. They solve these problems, allowing us to improve the second major area: the aforementioned sparse domain decomposition. Of the multitude of different improvement strategies, we propose and evaluate these: *shrinking of cuboids* and *Octree Domain Decomposition*.

Testing these measures shows their great potential. The *graph load balancer* performs at least as well as the original load balancer for nearly all cases, while outperforming it on most non-trivial geometries. The *heuristic load balancer* does not achieve the efficiency of the graph based load balancer, but still permits to utilize some of the gains due to the domain decomposition improvements.

As for the domain decomposition strategies, the octree allows the scaling of the cuboids to the complexity of the geometry at each point. Octree decomposition creates better fitting domain decompositions, but measurements show that it results in higher overhead. This is either due to the large number of cuboids per processor or due to the non-optimal fit to the geometry – depending on the minimum size of the cuboids. Nevertheless, the results hint at a better performance with more processors.

The *shrinking* strategy is the single most performant optimization developed in this thesis. It improved performance for the real world examples by 75% to 110%.

The achieved speed-up translates directly into time, money and energy savings for research and industrial applications. It moves boundaries for the problem size and geometry size even further, providing opportunities for ever more complex simulations.

6.2. Outlook

While this thesis improves the performance by a huge amount, many possible further optimizations remain. It might be of interest to further benchmark the combination of octree domain decomposition with shrinking, as this might lead to even more efficient sparse domain decomposition strategies. Also, the tests for octrees should be run on larger problems with more CPUs as those cases seem to be promising for this approach. Additional tests with the heuristic load balancer might be interesting as well, to see how it performs for larger problems in which the communication becomes more important as cuboids grow smaller.

Also, octrees could form the basis for an implementation of different degrees of discretization in parts of the geometry, depending on the amount of detail actually needed there.

The algorithm of octrees might even be further improved by an additional recombination step if the later steps were not successful in removing parts of the cuboids. This would reduce the overhead induced by the huge amounts of small cuboids.

Further work may be done in the area of partitioning memory on the different cluster nodes. This would be beneficial for the RAM usage of all processes and would allow for even bigger simulations.

With the future inclusion of the simulation of particles in OpenLB, the load balancer could be extended to adjust to shifting load, as the number of particles in each cuboid vary its load. If this is necessary and if the amount of difference in CPU time warrant it, such an approach should be evaluated.

All these measures could help to come close to the final goal, to investigate ever new aspects of the *grand challenge*, the simulation of the human respiratory system.

A. Appendix

A.1. Open Source Library OpenLB

A.1.1. Project Overview

OpenLB is an open source library that implements LBM in a modular and object oriented fashion. The library implements a multitude of lattice Boltzmann models with C++. The project is intended to be used by application programmers and developers alike, as it is easily extendable with self-developed dynamics.

The framework is optimized for high performance computing to simulate even large problems in computational fluid dynamics. It sports support for advanced data structures to handle complex geometries and highly parallel programs. A large range of problems can be simulated with OpenLB. Examples include, but are not limited to, the air flow in the lungs, blood flow in capillaries or the analysis of particle spreading from nose sprays in the nose.

Detailed information about the project may be gotten from the project homepage [5] and the overview article [18]. Additional details about the implementation can be found in the tech reports on the homepage. Details about the hybrid-parallelisation may be found in [17]. Latt, Chopard et.al. compared several boundary conditions for LBM in [22].

A.1.2. Authors

Several people were involved in the development of OpenLB. All the names shall be listed in alphabetical order below:

Project Coordinator:

- Dr. Mathias J. Krause (Karlsruhe Institute of Technology)

Project Initiators:

- Prof. Dr. Vincent Heuveline (Karlsruhe Institute of Technology)
- Dr. Mathias J. Krause (Karlsruhe Institute of Technology)
- Dr. Jonas Latt (Ecole Polytechnique Federale de Lausanne)

Current Code Developers:

- cand. techn. math. Lukas Baron (Karlsruhe Institute of Technology)
  - Parameterization, I/O, Optimization
- cand. inform. cand. math. Jonas Fietz (Karlsruhe Institute of Technology)
  - Config file parsing based on XML
- cand. techn. math. Thomas Henn (Karlsruhe Institute of Technology)
  - Pre-processing, Voxelizer based on STL
- cand. techn. math. Jonas Kratzke (Karlsruhe Institute of Technology)
  - Alternative boundary conditions (Bouzidi), Unit converter
- Dr. Mathias J. Krause (Karlsruhe Institute of Technology)
  - OpenMP parallelization
  - Cuboid data structure for MPI parallelization
  - Makefiles

Former Code Developers:

- Dr. Jonas Latt (Ecole Polytechnique Federale de Lausanne)
  - Development of the OpenLB core
  - Integration and maintenance of added components
- M. Sc. Orestis Malaspinas (Ecole Polytechnique Federale de Lausanne)
  - Alternative boundary conditions (Inamuro, Zou/He, Nonlinear FD)
  - Alternative LB models (Entropic LB, MRT)
- Bernd Stahl (University of Geneva)
  - MultiBlock structure for MPI parallelization
  - Parallel version of (scalar or tensor-valued) data fields
  - VTK output of data
- Math. techn. Simon Zimny (German Reseach School for Simulation Sciences)
  - Pre-processing, automized setting of boundary conditions

A.2. KaFFPa

KaFFPa is an extremely fast graph partitioner developed by the *Institute of Theoretical Computer Science, Algorithmics II* at the Karlsruhe Institute of Technology by Dipl. Math. Dipl. Inform. Christian Schulz and Prof. Dr. Peter Sanders.

At the time the papers detailing the implementation were submitted, it achieved over 300 new records in the Walshaw Benchmark [6], improving on the best known partitions for those problems.

The focus of KaFFPa lies on partition quality rather than speed. It is written in C++ and will be released in the beginning of 2012.

The main techniques used in KaFFPa are flow based algorithms, localized local searches and F-cycles, which are techniques transfered from the multigrid community.

B. German Abstract

In den letzten Jahrzehnten hat der Bereich der *computergestützten numerischen Strömungsmechanik* immer mehr an Bedeutung gewonnen. Die rapide Entwicklung von immer schnelleren Rechnern leistete der Simulation komplexer werdender Geometrien weiteren Vorschub. Die Anwendungsbereiche dieser Simulationstechniken sind vielfältig.

Ein beliebtes Anwendungsbeispiel ist die Simulation der aerodynamischen Eigenschaften von Automobilen oder Flugzeugen als Ersatz für oft teure und aufwendige Windkanalversuche. Ein anderes, sehr wichtiges Beispiel, auf das sich die Forschungsarbeit am Institut für numerisches Hochleistungsrechnen in letzter Zeit unter anderem fokussiert wurde, ist die Simulation des menschlichen Atemtrakts. Seine Funktion ist bisher noch nicht in allen Einzelnheiten verstanden, da hier hochkomplexe physikalische Phänomene auf multiple Strukturskalen treffen, gepaart mit hochgradig komplexen Geometrien. Deswegen ist die Simulation der Lunge und der restlichen Atemwege immer noch eine der großen Herausforderungen der Strömungsmechanik. Die Bedeutung von Fortschritten in diesem Sektor wird klar, wenn man mögliche Anwendungen betrachtet: z.B. die auf Patienten zugeschnittene, individuelle Simulation, Analyse und Behandlung von Atemwegsrestriktionen und anderen Lungenerkrankungen; die Vorberechnung und Simulation möglicher Nebeneffekte von Operationen durch Chirurgen oder sogar vollautomatische Operationen durch Roboterchirurgen, die mit höchster Präzision arbeiten können.

Die Simulation der Lunge erfordert *effiziente Parallelisierungsstrategien* und die Ausnutzung der heute vorhandenen Hardware-Architekturen, wie massiv-parallele CPUs und GPUs durch die Nutzung von *hybriden High-Performance-Technologien.*

Eines der Projekte, die an der Simulation des Atemtrakts arbeiten, ist das Projekt *United Airways*. Das United Airways-Project nutzt die Open-Source-Software *OpenLB* für die Simulation der menschlichen Lunge und des Nasenrachenraums. OpenLB-Bibliothek erlaubt die Simulation von durch den Anwender definierten strömungsmechanischen Problemen, und gibt dem Nutzer hierfür die Möglichkeit, selbst definierte Characteristiken und physikalische Grundparameter zu wählen.

Die Bibliothek OpenLB implementiert eine Open-Source-Variante der *Lattice-Boltzmann-Methoden* (LBM), die in der zweiten Hälfte des letzten Jahrhunderst entwickelt wurden. Diese Methoden haben den Vorteil, dass sie inhärent parallel sind und sich dadurch besonders gut für große Probleme eignen.

Das Ziel dieser Arbeit ist es, mögliche Verbesserungen der Skalierung und Performanz für LBM am Beispiel von OpenLB zu untersuchen, zu implementieren und zu testen. Mehrere mögliche Schwachpunkte werden hierzu identifiziert und auf mögliche Verbesserungen analysiert, so dass hier zwei Dimensionen von Optimierungsstrategien aufgezeigt werden.

Der erste Teil von OpenLB, der Verbesserungspotential zeigt, ist der *Load-Balancer*, der Algorithmus zur Lastverteilung. Der aktuelle Load-Balancer in OpenLB beruht auf sehr vereinfachenden Annahmen und besitzt daraus resultierend eine eher einfache Struktur. Als Alternative zu dieser Implementierung wurden zwei weitere Load-Balancer implementiert. Der erste dieser beiden nutzt Ergebnisse aus der Graphentheorie, indem er die Prinzipien der Graph-Partitionierung auf das Problem der Lastverteilung anwendet. Als Graphpartitionierer für diesen Algorithmus wird die Software *KaFFPa* verwendet. Die zweite Alternative ist die Implementierung einer simplen Heuristik, die im Gegensatz zu dem originalen Load-Balancer zumindest die unterschiedlichen Umfänge der Arbeitspakete, die es zu verteilen gilt, berücksichtigt.

Diese beiden Load-Balancer werden in dieser Arbeit anhand ihrer Performance bei verschiedenen Geometrien evaluiert. Einer dieser Tests ist die Geometrie *Bifurcation-Large*. Diese ist ein typisches Beispiel für eine Teilsimulation der Lunge oder für das Bifurkations-Verhalten von Kapillaren z.B. in der Leber. Nachdem mit Hilfe von Tests die Modellannahmen der beiden neuen Load-Balancer-Varianten validiert wurden, werden die Berechnungsgeschwindigkeiten u.a. fÃ$\frac{1}{4}$r das genannte Bifurcation-Large-Beispiel evaluiert. Diese Tests zeigen, dass insbesondere der graph-basierte Load-Balancer bei den komplexen Testfällen bereits Effizienzsteigerungen von ca. 30%, d.h. Reduktionen der Laufzeit auf ca. 2/3, erreichen kann.

Ein Resultat und mitunter auch ein Grund für die Verbesserung der Lastverteilung

ist, dass die beiden neuen Load-Balancer beide wesentliche Fortschritte fÃ¼r die *Gebietszerlegung* erlauben. Dies ist der zweite große Bereich der Verbesserungsstrategien. Hier werden zwei weitere Ansätze untersuchst. Der erste ist die an die zugrundeliegende Geometrie angepasste Zerlegung des Gebiets durch *Octrees*. Diese Strategie erlaubt die Zerlegung mit unterschiedlichen Feinheiten, je nachdem wie komplex die Geometrie in dem jeweiligen Bereich ist. Neben dem Effekt durch weniger Verschnitt ermÃ¶glicht dies auch weitergehende Verfahren wie die ortsabhängige, konkrete Anpassung der Auflösung an die vorhandene Komplexität der Geometrie. Tests dieser Strategie zeigen zwar keine konkreten Verbesserungen in der Laufzeit für die getesteten Problemgrößen, aber die Trends scheinen auf eine Verbesserung bei größeren Fällen hinzudeuten.

Die zweite Methode zur Verbesserung der Gebietszerlegung ist das *Schrumpfen* der bisherigen Quaderzerlegung. Dabei werden einzelne Quader auf eine Größe geschrumpft, mit der sie nur noch Fluid- und Randzellen enthalten. Diese Methode ergab in den Tests insbesondere bei den realistischen Testfällen der *Bifurcation-Benchmarks* Geschwindigkeitszuwächse von 75% bis zu über 100% mit den dazugehörigen Lauftzeitverbesserungen der Berechnungen auf weniger als Hälfte.

Diese Resultate bedeuten natürlich für Wissenschaft, Forschung und Industrie eine immense Reduktion der investierten Zeit, des Geldes und natürlich der Energie. Alternativ erlauben sie auch eine Vergrößerung und erhöhte Komplexität der zu berechnenden Probleme bei gleichem Aufwand, mit all den damit verbundenen Chancen.

Bibliography

[1] InstitutsCluster User Guide - Version 0.93. `http://www.scc.kit.edu/downloads/sca/ugic.pdf`, 2009. [Online; accessed 11-November-2011].

[2] *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010.

[3] Creative Commons Attribution-ShareAlike 3.0 Unported License. `http://creativecommons.org/licenses/by-sa/3.0/`, 2011. [Online; accessed 25-November-2011].

[4] HP XC3000 User Guide - Version 1.07. `http://www.scc.kit.edu/scc/docs/HP-XC/ug/ug3k.pdf`, 2011. [Online; accessed 11-November-2011].

[5] OpenLB Homepage. `http://www.openlb.org`, 2011. [Online; accessed 11-November-2011].

[6] Walshaw Graph Partitioning Archive. `http://staffweb.cms.gre.ac.uk/~wc06/partition/`, 2011. [Online; accessed 29-November-2011].

[7] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. *Phys. Rev.*, 94:511–525, May 1954.

[8] T Braun. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001.

[9] Campbell, P.M., Devinve, K.D., Flaherty, J.E., Gervasio, L.G. and Teresco, J.D. Dynamic Octree Load Balancing Using Space-Filling Curves. 2003.

[10] C. Cercignani. *Theory and application of the Boltzmann equation*. Elsevier, 1975.

[11] T P Chiang, W H Sheu, and Robert R Hwang. Effect of reynolds number on the eddy structure in a lid-driven cavity. *International Journal for Numerical Methods in Fluids*, 26(5):557–579, 1998.

[12] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[13] Camil Demetrescu, editor. *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, volume 4525 of *Lecture Notes in Computer Science*. Springer, 2007.

[14] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, New York, NY, USA, 1982. ACM.

[15] R. L. Graham and R. L. Grahamt. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[16] Norbert Henze. *Stochastik II - Maß und Wahrscheinlichkeitstheorie*. 2004.

[17] V. Heuveline, M.J. Krause, and J. Latt. Towards a hybrid parallelization of lattice Boltzmann methods. *Computers & Mathematics with Applications*, 2009.

[18] V. Heuveline and J. Latt. The OpenLB project: an open source and object oriented implementation of lattice Boltzmann methods. *Int. J. Mod. Phys. C*, 18:627–634, 2007.

[19] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *IPDPS* [2], pages 1–12.

[20] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes.

[21] Mathias J. Krause. *Fluid Flow Simulation and Optimisation with Lattice Boltzmann Methods on High Performance Computers: Application to the Human Respiratory System*. Karlsruhe Institute of Technology (KIT), 2010.

[22] Jonas Latt, Bastien Chopard, Orestis Malaspinas, Michel Deville, and Andreas Michler. Straight velocity boundaries in the lattice Boltzmann method. *Phys. Rev. E*, 77:056703, 2008.

[23] R.L. Liboff. *Kinetic theory: classical, quantum, and relativistic descriptions*. Prentice Hall advanced reference series: Physical and life sciences. Prentice Hall, 1990.

[24] Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In Demetrescu [13], pages 242–255.

[25] Guy R. Mcnamara and Gianluigi Zanetti. Use of the boltzmann equation to simulate Lattice-Gas automata. *Phys. Rev. Lett.*, 61, 1988.

[26] Vitaly Osipov and Peter Sanders. n-Level Graph Partitioning. In *18th European Symposium on Algorithms (ESA)*, LNCS, 2010.

[27] F. Pellegrini. Scotch home page. `http://www.labri.fr/pelegrin/scotch`. [Online; accessed 11-November-2011].

[28] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms.

[29] Kirk Schloegel, George Karypis, Vipin Kumar, J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White, and Morgan Kaufmann. Graph partitioning for high performance scientific simulations, 2000.

[30] Shephard, M.S. and Georges, M.K. Automatic three-dimensional mesh generation by Finite Octree technique. *Int. J. Numer. Meth. Engng*, 32:709–749, 1991.

[31] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford University Press, 1st edition, 2001.

[32] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice Boltzmann kernels. *Comput. Fluids*, 35(8-9):910–919, September 2006.

List of Figures

2.1. Example for *D3Q19* model showing the 19 different speeds 10
2.2. The different levels of abstractions in OpenLB used for *hybrid parallelization* [21] 13
2.3. Example octree with Geometry 17

3.1. Example for *Domain Decomposition* with the current methodology [21]. Cuboids in the finer parts of the lung are several times larger than the geometry, leading to inefficiency. 20

4.1. Load balancing of cuboid consisting of 256 smaller cuboids using the old load balancer for 32 tasks 27
4.2. Load Balancing of cuboid consisting of 256 smaller Cuboids with graph based load balancer for 32 tasks 28
4.3. An example octree decomposition of bifurcation-large benchmark with dimension Nx=599, Ny=92, Nz=703 and a minimum cuboid size of 64 31

5.1. The geometry of the *Lid-Driven-Cavity* benchmark - a cube with a lid moving in direction $(1, 0, 0)$ where all other cells are initialized to $u = 0$ at the beginning. These cells are then accelerated by the moving lid [21]. 34
5.2. Typical fluid behavior for *Lid-Driven-Cavity* benchmark with the development of a vortice [21]. 35
5.3. An image of the bifurcation geometry 36
5.4. An image of the bifurcation-large geometry 37
5.5. Cavity3D - Comparing load balancers 41
5.6. Cavity3D - comparing load balancers - N=100,200,400 42
5.7. Bifurcation - comparing load balancers 43
5.8. Bifurcation-Large - Comparing load balancers 45

5.9. Octree Decomposition Bifurcation Large 8^3 46
5.10. Octree Decomposition Bifurcation Large 16^3 46
5.11. Octree Decomposition Bifurcation Large 32^3 47
5.12. Octree Decomposition Bifurcation Large 64^3 47
5.13. Bifurcation Large - Comparing Octree and Traditional Decomposition 50
5.14. Bifurcation Large - Improvements through Shrinking 51
5.15. Bifurcation Large - Improvements through Shrinking 53
5.16. Octree Decomposition Bifurcation Large with Shrinking 32^3 54
5.17. Octree Decomposition Bifurcation Large with Shrinking 64^3 55
5.18. Traditional decomposition and shrinking 55

List of Tables

4.1. Execution speed of full and empty cells for cavity3d benchmark . . . 25

5.1. Verification of Communication Modeling 39
5.2. Comparison bifurcation-large speed p=512 44
5.3. Heuristic Load Balancer 48
5.4. Heuristic Load Balancer and Shrinking 49
5.5. Number of Cuboids in Relation to Minimum Cuboid Size 49
5.6. Speed-comparison Traditional to Shrinking 52
5.7. Speed-comparison Traditional to Shrinking 52