Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries

Jonas Fietz[2], Mathias J. Krause[2], Christian Schulz[1], Peter Sanders[1], and Vincent Heuveline[2]

[1] Karlsruhe Institute of Technology (KIT), Institute for Theoretical Informatics, Algorithmics II

[2] Karlsruhe Institute of Technology (KIT), Engineering Mathematics and Computing Lab (EMCL)

Abstract. Computational fluid dynamics (CFD) have become more and more important in the last decades, accelerating research in many different areas for a variety of applications. In this paper, we present an optimized hybrid parallelization strategy capable of solving large-scale fluid flow problems on complex computational domains. The approach relies on the combination of lattice Boltzmann methods (LBM) for the fluid flow simulation, octree data structures for a sparse block-wise representation and decomposition of the geometry as well as graph partitioning methods optimizing load balance and communication costs. The approach is realized in the framework of the open source library OpenLB and evaluated for the simulation of respiration in a subpart of a human lung. The efficiency gains are discussed by comparing the results of the full optimized approach with those of more simpler ones realized prior.

Keywords: Computational Fluid Dynamics, Numerical Simulation, Lattice Boltzmann Method, Parallelization, Graph Partitioning, High Performance Computing, Human Lungs, Domain Decomposition

1 Introduction

The importance of *computational fluid dynamics* (CFD) for medical applications have risen tremendously in the past few years. For example, the function of the human respiratory system has not yet been fully understood, and its complete description can be considered byzantine. Due to highly intricate multi-physics 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. Widely automated preprocessing as well as efficient numerical methods are both necessary conditions for enabling real-time simulations.

In the last decades, *lattice Boltzmann methods* (LBM) have evolved into a mature tool in CFD and related topics in the landscape of both commercial and academic software. The simplicity of the core algorithms as well as the locality properties resulting from the underlying kinetic approach lead to methods which are very attractive in the context of parallel computing and high performance computing [7,8,13]. In this context, it is of great importance to take advantage of nowadays available hardware architectures like Graphic Processing Units (GPUs), multi-core processors and especially hybrid high performance technologies that blur the line of separation between architectures with shared and distributed memory. A concept to use LBM dedicated for hybrid platforms has been described before in [3]. It relies on spatial domain decomposition where each domain represents a basic block entity which is solved on a symmetric multi-processing (SMP) system. The regularity of the data structure of each block allows a highly optimized implementation dedicated to the particular SMP hardware. Load balancing is achieved by assigning the same number of equally-sized blocks to each of the available SMP nodes. This concept has been extended and applied for fluid flows simulation on complex geometries [5].

The goal of this work is to optimize the hybrid parallelization approach for LBM simulations on complex geometries. The basic idea is to drop the equally-sized block constraint thereby enabling a sparse representation of the computational domain. Therefore, two domain decomposition strategies are proposed as well as the application of graph based load balancing techniques to the load distribution problem for LBM. The first domain decomposition strategy is a heuristic, which we further improve by a shrinking step. The second strategy is a geometry aware decomposition using octrees. This results in a sparse domain decomposition with larger computational domains. Both of these strategies require sophisticated load balancing. We propose a graph partitioning based approach optimizing the load and minimizing communication costs. While graph based load balancing has been done before by [1], we propose to apply this not on a fluid cell level but at block level. Finally, we evaluate the presented measures on a subset of the human lung, showing performance improvements for all of them.

2 Lattice Boltzmann Fluid Flow Simulations

The here considered subclass of *lattice Boltzmann methods* (LBM) enable to simulate the dynamics of *incompressible Newtonian fluids* which is usually described macroscopically by an initial value problem governed by a *Navier-Stokes equation*. Instead of directly computing the quantities of interests, which are the fluid velocity $\boldsymbol{u} = \boldsymbol{u}(t, \boldsymbol{r})$ and fluid pressure $p = p(t, \boldsymbol{r})$ where $\boldsymbol{r} \in \Omega \subseteq \mathbb{R}^d$ and $t \in I = [t_0, t_1) \subseteq \mathbb{R}_{\geq 0}$, a lattice Boltzmann (LB) numerical model simulates the dynamics of particle distribution functions $f = f(t, \boldsymbol{r}, \boldsymbol{v})$ in a phase space $\Omega \times \mathbb{R}^d$ with position $\boldsymbol{r} \in \Omega$ and particle velocity $\boldsymbol{v} \in \mathbb{R}^d$. The continuous transient phase space is replaced by a discrete space with a spacing of $\delta r = h$ for the positions, a set of $q \in \mathbb{N}$ vectors $\boldsymbol{v_i} \in \mathcal{O}(h^{-1})$ for the velocities and a spacing of $\delta t = h^2$ for time. The resulting discrete phase space is called the lattice and is labeled with the term $DdQq$. To reflect the discretization of the velocity space, the continuous distribution function f is replaced by a set of q distribution functions f_i $(q = 0, 1, ..., q - 1)$, representing an average value of f in the vicinity of the velocity $\boldsymbol{v}_i$. Detailed derivations of various LBM can be found in the literature, e.g. in [11].The iterative process in an LB algorithm can be written in two steps as follows, the *collision step* (1) and the *streaming step* (2):

$$\widetilde{f}_i(t,\boldsymbol{r}) = f_i(t,\boldsymbol{r}) - \frac{1}{3\nu + 1/2}\left(f_i(t,\boldsymbol{r}) - M^{eq}_{f_i}(t,\boldsymbol{r})\right) \;, \tag{1}$$

$$f_i(t+h^2, \boldsymbol{r}+h^2\boldsymbol{v_i}) = \widetilde{f}_i(t,\boldsymbol{r}) \tag{2}$$

for $i = 0,..,q-1$. $M^{eq}_{f_i}(t,\boldsymbol{r}) := \frac{w_i}{w}\rho_{f_i}\left(1 + 3h^2\ \boldsymbol{v_i}\cdot\boldsymbol{u_{f_i}} - \frac{3}{2}h^2\boldsymbol{u}^2_{f_i} + \frac{9}{2}h^4\left(\boldsymbol{v_i}\cdot\boldsymbol{u_{f_i}}\right)^2\right)$ is a discretized *Maxwell distribution* with moments ρ and $\boldsymbol{u}$ which are given according to $\rho := \sum_{i=0}^{q-1} f_i$ and $\rho\boldsymbol{u} := \sum_{i=0}^{q-1} \boldsymbol{v_i} f_i$. The variable $\boldsymbol{u}$ is the discrete fluid velocity and ρ the discrete mass density. The kinematic fluid viscosity is ν which is assumed to be given, and the terms w_i/w, $\boldsymbol{v_i}h$ $(i = 0,1,...,q-1)$ are model dependent constants. The discrete fluid velocity $\boldsymbol{u}$ and the discrete mass density ρ can be related to the solution of a macroscopic initial value problem governed by an incompressible Navier-Stokes equation as shown by Junk and Klar [4].

3 Domain Decomposition for Hybrid Parallelization

The most time demanding steps in LB simulations are usually the collision (1) and the streaming (2) operations. Since the collision step is purely local and the streaming step only requires data of the neighboring nodes, parallelization has mostly been done by domain partitioning [7,8,13]. To take advantage of hybrid architectures, a multi-block approach is used [3]: the computational domain is partitioned into sub-grids with possibly different levels of resolution, and the interface between those sub-grids is handled appropriately. This leads to implementations that are both elegant and efficient since the execution on a set of regular blocks is much faster compared to an unstructured grid representation of the whole geometry. For complex domains a multi-block approach also yields sparse memory consumption. Furthermore, it encourages a particularly efficient form of data parallelism, in which an array is cut into regular pieces. This is a good mapping to hybrid architectures.

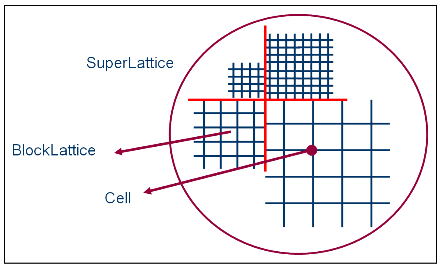


Fig. 1: Data structures used in OpenLB: `BlockLattices` consist of `Cells` and make up a `SuperLattice` enabling higher level software constructs.

In *OpenLB*, the basic data-structure is a `BlockLattice` representing a regular array of `Cells`. In each `Cell`, the q variables for the storage of the discrete velocity distribution functions f_i, $(i = 0,1,...,q-1)$ are contiguous in memory. Required memory is allocated only once since no temporary memory is needed in the applied algorithm. This data structure is encapsulated by a higher level, object-oriented layer. The purpose of this layer is to handle groups of `BlockLattices`, and to build higher level software constructs in a relatively transparent way. Those constructs are called `SuperLattices` and include multi-block, grid refined lattices as well as parallel lattices.

3.1 Heuristic Domain Decomposition with Shrinking Step

In this section, we describe our heuristic domain partitioning strategy. We further improve this by shrinking each of the partitions, so that it achieves a closer fit to the underlying geometry.

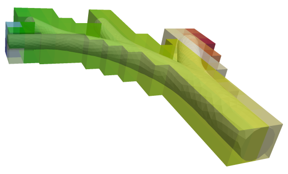
The hybrid parallelization strategy proposes to partition the data of a considered discrete position space Ω_h, which is a uniform mesh with spacing $h > 0$, according to their geometrical origin into $n \in \mathbb{N}$ disjoint, preferably cube-shaped sub-lattices Ω_h^k $(k = 0, 1, ..., n-1)$ of almost similar sizes. This becomes feasible by extending Ω_h to a cuboid-shaped lattice $\widetilde{\Omega}_h$ through the introduction of ghost cells. Then, $\widetilde{\Omega}_h$ is split into $m \in \mathbb{N}$ disjoint, cuboid-shaped extended sub-lattices $\widetilde{\Omega}_h^l$ $(l = 0, 1, ..., m-1)$ of approximately similar size and as cube-shaped as possible. Afterwards, all extended sub-lattices $\widetilde{\Omega}_h^l$ which consist solely of ghost cells are neglected. The number of the remaining extended sub-lattices $\widetilde{\Omega}_h^l$ $(l_0, l_1, ..., l_{n-1})$ defines n. Finally, for each $k \in \{0, 1, \ldots, n-1\}$ one gets the Ω_h^k as a subspace of $\widetilde{\Omega}_h^{l_k}$ by neglecting the existing ghost cells.

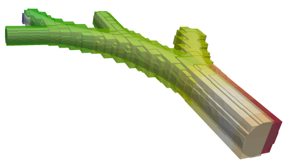
For the number $p \in \mathbb{N}$ of available processing units (PUs) of a considered hybrid high performance computer, an even load balance will be assured for complex geometries in particular if the domain Ω_h is partitioned into a sufficiently large number $n \in \mathbb{N}$ of sub-lattices. Then, several of the sub-lattices Ω_h^k $(k = 0, 1, ..., n-1)$ can be assigned to each of the available PUs. To find a good value for n, we introduce a factor k for the amount of sub-lattices with the relation $n = p \times k$. This factor can be adjusted for a specific problem by evaluating run-times for a few hundred time steps to achieve better performance.

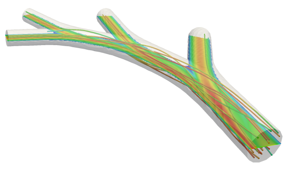
After removing all empty cuboids, we then optimize the fit of the cuboids to the underlying geometry. To find out if a cuboid can be shrunk, we start running through each layer in all 6 directions beginning at the respective faces of the cube. For each layer, we check if it is completely empty and stopping 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 cuboids. Note that the same shrinking step can also be applied to the above mentioned octree domain decomposition and works in exactly the same way. An example decomposition can be seen in Fig. 2a.

3.2 Sparse Octree Domain Decomposition with Shrinking

A key part of load balancing is the decomposition of the complete domain in sub-domains. Here, one has to optimize for multiple, sometimes opposing properties of the sub-domains. As more cuboids mean more communication, cuboids should be as large as possible. The surface of each cuboid should be minimal, as this


(a) Heuristic Decomposition without additional steps


(b) Octree Decomposition with additional shrinking step


(c) Stream lines and a cut plane of velocity distribution.

determines the amount of communication for this cuboid. This usually implies a shape as close to a ball as possible. As ball-shaped objects are not space filling, and due to the current implementation in our library supporting only rectangular shapes, the optimal shape is a cube.

The streaming step is executed without respect for the underlying geometry information. Therefore, even non-fluid cells use some processing power. Because of this, a tight fit of the domain decomposition with respect to the specific geometry is desirable. This is where octrees come into play to adjust the size of cubes depending on the geometry.

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, we use a size of $2^l \times \delta r$ for some $l \in \mathbb{N}$ 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 cube sizes.

Having defined the root cube now, one recursively divides the cube into smaller cubes as long as the geometry in this part is interesting. In our case this 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, 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. This minimum size can be defined as the side length of the minimum cube, a number c. The shrinking procedure can be applied to the octree domain decomposition as well (combination is abbreviated as *ODS*). An example decomposition can be seen in Fig. 2b.

4 Load Balancing

As we explained in the introduction of Section 3, the most commonly used approach to load balancing LBM is based on an even decomposition of the computational domain. This section describes our alternative approach using techniques from graph partitioning.

4.1 Graph Partitioning using KaFFPa

Our parallelization strategy employs the graph partitioning framework KaFFPa [9]. We shortly describe the graph partitioning problem. and introduce notations used. Consider an undirected graph $G = (V, E, c, \omega)$ with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. Given a number $k \in \mathbb{N}$ (in our case the number of processors) the graph partitioning problem demands to partition V into *blocks* of nodes $V_1,\ldots,V_k$ such that $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. A *balancing constraint* demands that all blocks have roughly equal size, i.e. the maximum load of a processing element is bounded. The *objective* is to minimize the total *cut*, i.e. the sum of the weight of the edges that run between blocks. We have tested and shown that the edge cut models the communication very well, because the edge weights correlate linearly with the amount of communication between two cuboids [2]. For more details on graph partitioning with KaFFPa we refer the reader to [9].

4.2 Graph-based Parallelization Strategy for LBM

As described in the prior sections, the LBM algorithm is divided into two parts. The first part is the local collision and streaming for each cuboid. Afterwards, the information is exchanged between different cuboids, i.e. transmitting the information of the border cells for neighboring cuboids assigned to different processors. 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 map this problem to graph partitioning, we associate the work amount or needed CPU time for each cuboid with the weight of a node for this cuboid, and associate the needed communication between two cuboids with 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. The edge-cut – the inter-processor communication – will be minimized. 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.

4.3 Determining Node and Edge Weights

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

We begin with the latter. The edge weight is either the byte count or the number of cells to be communicated. This information is often already present, as every implementation of LBM has to find the border cells that need to be communicated, anyway. The case is not as simple for non-symmetrical communication between different cuboids. One can either choose the maximum or the sum of both parts as the edge weight. Since the data transfer between two cuboids is serial in our implementation – i.e. we first transfer in one direction, then in the opposite – we pick the sum.

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 differs. As the number of boundary cells is usually quite small, and as they are treated as an extra step, this special case is ignored; they are assumed to be normal fluid cells. Empty cell in our case are either cells in a solid area or that this cell is outside of the fluid filled body being simulated. While the collision step is not executed for the empty cells, the streaming still is. As it is very possible for a cuboid to consist largely of empty cells, it is important to know how much processing time the empty cells use when compared to the normal fluid cells. For this we introduce a factor χ. We measured χ for several different architectures. Unfortunately, it is not a specific constant valid even for the limited types we tested. Instead, it varies from 1.8 to 4.5 [2] for the differing machines used in our preliminary work. These dispartities are most likely due to the different memory and cache hierarchies and resulting diverse memory access speeds, as the streaming part of the LB algorithm is memory bound. To calculate the work to be done for each cuboid, using the symbols for the work ω, for the number of fluid cells n_f, and for the number of empty, non-fluid cells n_e, we get the formula $\omega = n_f + \chi n_e$.

In the end, the graph load balancer is now able to balance the work load to a set number of processing nodes or cores and to find a solution for a certain load imbalance with accordingly small communication overhead.

5 Experiments

The aim of this section is to illustrate the effectiveness of the presented options considering a practical problem with an underlying complex geometry, namely the expiration in a human lung. The geometry we use is a subset of the bronchi of the lungs, with bifurcation of the bronchi to the third level (see Fig. 2a). The air for the simulation is assumed to be at normal conditions ($1013hPa$, 20°C), i.e. $\rho = 1.225kg/m^3$ and its kinematic viscosity is $\nu = 1.4 \times 10^{-5}m^2/s$. The outflow region is set at the trachea with a pressure boundary condition with constant pressure of $1013hPa$. The inflow regions are the bronchioles. There, a velocity boundary condition is set as a Poiseuille distribution with a maximum speed of $1m/s$. The characteristical length is set to $2cm$, which is the diameter of the trachea. With a characteristical speed of $1m/s$, we get a Reynolds number of around 1400. To solve the problems numerically, we use a D3Q19 LB model with the pressure and velocity boundary conditions as proposed by Skordos [10]. No-slip conditions for the walls are realized as a bounce-back boundary. For the LB simulation, we set the Mach number to 0.05 and δr to 3.91×10^{-4}. We obtain the dimensions of $402 \times 54 \times 343$ cells, with about 1.06 million filled cells, i.e. a fill grade of approximately 14.5%.

Table 1: Comparison of balancers with 512 processors. The best value of k is emphasized.

k	DBLB	GBLB
1	0.117	0.067
2	0.223	0.044
4	0.198	**0.303**
8	**0.226**	0.297
16	0.199	0.260
32	0.134	0.137
64	0.022	0.068

All benchmarks were run on a cluster at the Karlsruhe Institute of Technology. It consists of 200 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 with 16 GB of RAM.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. 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.

To compare the performance of LB, in most cases one uses the measurement unit of *million fluid-lattice-site updates per second* MLUP/s, e.g. [12]. This idea can be extended to the unit MLUP/ps for *million fluid-lattice-site updates per process and second* [6]. The latter unit is used in all examples. One calculates the amount of fluid cells N_c for each of the examples. With the run-time t_p for p processor cores, the number i of iterations, the result is given as $P_{LB} := 10^{-6} \frac{iN_c}{t_p p}$

5.1 Decomposition vs. Graph Based Load Balancing

The first comparison is between the *Decomposition Based Load Balancer* (DBLB) and the new *Graph Based Load Balancer* (GBLB). Small values of k are not very efficient, because they allow some processors to run empty. Therefore, only higher values of k are shown in Fig. 2.

This benchmark shows a steep initial decline of computation speed when scaling from one to eight computing cores. This is due to the memory bound characteristic of the LBM algorithm and the limited amount of faster caches on the target architecture and its shared memory buses.

While one can see that the results are 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 Tab. 1. For 512 cores, the GBLB solution only takes about two-thirds of the execution time of the DBLB one.

5.2 Effects of Using the Shrinking Step

The *shrinking step* as a step to optimize the size of the cuboids showed itself to be the most effective strategy of all, despite its seemingly simple nature. Performance for the test benchmark increased by over 100% in certain cases (see Tab. 2). All test cases with shrinking were run with the graph based load balancer. These tests included core counts between 1 and 256 and the factor $k \in \{2^0, \ldots, 2^6\}$. An excerpt of the performance for the best decomposition based

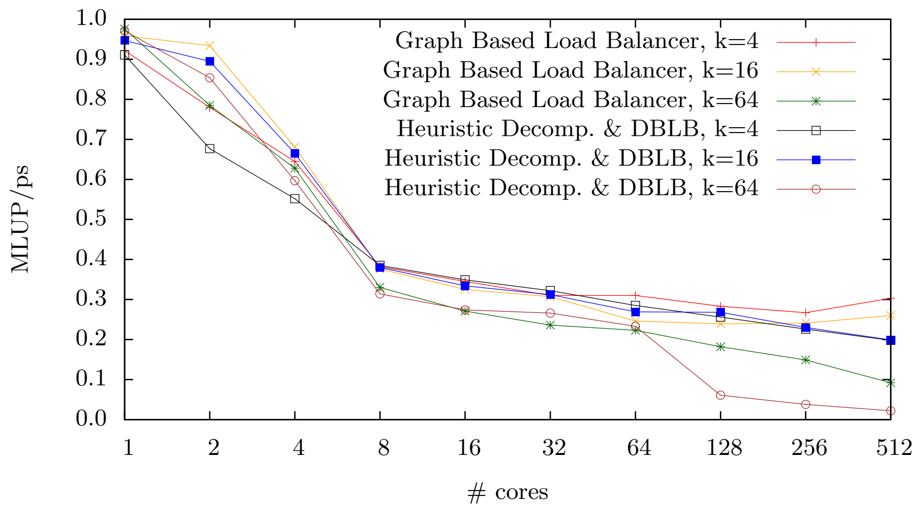


Fig. 2: Comparison of the DBLB and the GBLB for certain k. For larger numbers of cores, one can see the performance improvement of using the GBLB without any change to the decomposition algorithm.

load balancer and the best graph based load balancer test runs with an additional shrinking step are shown in Fig. 3. One can see that the solution with the shrinking step and the GBLB outperforms the DBLB for all values of k, respectively. This speed-up can be attributed to multiple effects. Because empty cells are excluded from the streaming step, less streaming is required. One can get an impression of the possible reduction in the amount of empty cells from Tab. 3. 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 shown 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, as empty cells are removed, because the empty cells require no computation and are mainly memory intensive. Hence, the memory hierarchy is put under less strain, so the caches 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.

Table 2: Speeds in MLUP/ps for the *best* variant for each processor with DBLB compared to the *shrinking step* and GBLB.

# Cores	DBLB	Shrinking	Speed-up
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 3: Comparison of amount of cells that are computed before and after executing the *shrinking* step on a heuristically decompositioned geometry.

# Cuboids Before Remove	# Cuboids	# Cells Before Shrinking	# Cells After Shrinking
64	33	3 776 144	1 628 975
128	58	3 303 225	1 577 444
512	185	2 659 763	1 410 635

5.3 Octree Domain Decomposition

For smaller number of cores the octree decomposition combined with the graph based load balancer turns out not to improve performance significantly over the original approach. This is due to the amount of cuboids generated by this approach which creates inefficiencies for small numbers of cores and small minimum cuboids. It is only when combined with the shrinking step that performance improves significantly, although not all across the board. This is because the sizing of the minimum cuboid is too coarse, as it is restricted to powers of two. In certain cases, this might lead to too many or to not enough cuboids for efficient load balancing, exactly the situation where the factor k for the heuristic decomposition shows its strengths. Another detrimental effect can also be due to the specific structure of the tested geometry. Because the diameter of the bronchi is small, the middle coordinates have to align perfectly to get bigger cuboids with the octree decomposition. Yet in certain situations, the GBLB & ODS is the fastest solution (see Tab. 4).

Table 4: Comparing DBLB to GBLB with heuristic decomposition (HD) and to GBLB with octree decomposition and shrinking step (ODS) for a randomly chosen example subset. Performance varies depending on the number of cuboids, but GBLB solutions always achieve a speed-up.

# Cores	k	HD & DBLB	HD& GBLB	c	GBLB & ODS
32	4	**0.322**	0.310	16	0.243
32	8	0.299	**0.319**	32	**0.421**
32	16	0.312	0.307	64	0.357
256	4	0.226	**0.267**	8	0.147
256	8	**0.247**	0.261	16	**0.319**
256	16	0.230	0.241	32	0.152
512	4	0.198	**0.303**	8	0.058
512	8	**0.226**	0.297	16	**0.264**
512	16	0.199	0.260	32	0.077

6 Conclusions

We have given a successful example for a general technique that will become more and more important in the simulation of unstructured systems: Use partitioning of weighted graphs to do high level load balancing of computational grids where each node represents a regular grid that can be handled efficiently by modern hardware.

Specifically, we examined potential optimizations for Lattice Boltzmann Methods on the example of the OpenLB implementation. We identified two areas with potential for major improvement. First, the current load balancer, and second

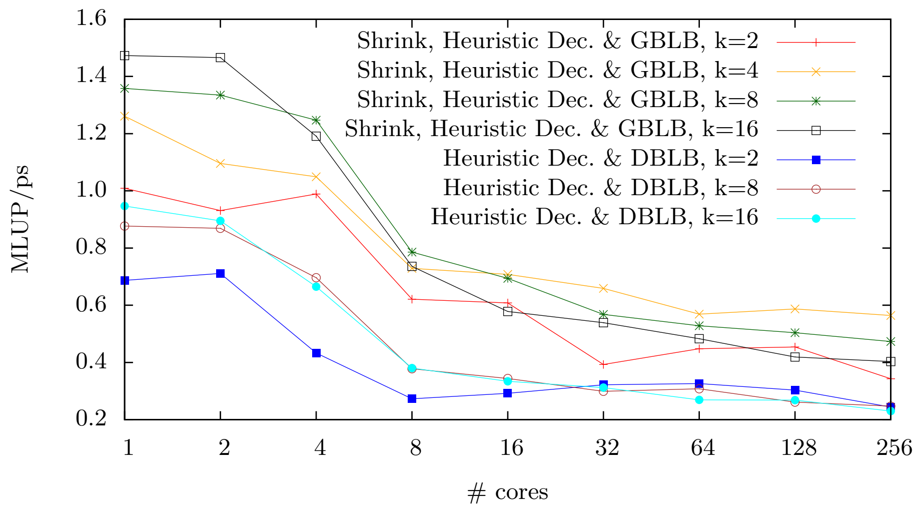


Fig. 3: Comparing the DBLB to the heuristic domain decomposition with the additional shrinking step and GBLB. One can see that performance approximately doubles with the new shrinking and GBLB solution.

the simple heuristic sparse domain decomposition. The decomposition based load balancer *only* equalizes the computational complexity and limits potential optimizations for sparse domain decomposition. Therefore, we designed and implemented two alternatives which additionally allow us to improve the sparse domain decomposition.

Of the multitude of different improvement strategies, we propose and evaluate these: *shrinking of cuboids* and *Octree Domain Decomposition*. 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 *decomposition based load balancer* (DBLB) 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 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 sometimes results in higher overhead. Nevertheless, the results hint at a better performance with more processors. The *shrinking* strategy improves performance for the real world example from 75% up to 105%. Further improvements are expected to be made by combining other measures and fine-tuning settings. 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.

References

1. Mauro Bisson, Massimo Bernaschi, Simone Melchionna, Sauro Succi, and Efthimios Kaxiras. Multiscale hemodynamics using GPU clusters. *Communications in Computational Physics*, 2011.
2. Jonas Fietz. Performance Optimization of Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries. Diplomarbeit, Karlsruhe Institute of Technology (KIT), Department of Mathematics, December 2011.
3. V. Heuveline, M.J. Krause, and J. Latt. Towards a Hybrid Parallelization of Lattice Boltzmann Methods. *Computers & Mathematics with Applications*, 58:1071–1080, 2009.
4. Michael Junk and Axel Klar. Discretizations for the Incompressible Navier-Stokes Equations Based on the Lattice Boltzmann Method. *SIAM J. Sci. Comput.*, 22(1):1–19, 2000.
5. Mathias Krause, Thomas Gengenbach, and Vincent Heuveline. Hybrid Parallel Simulations of Fluid Flows in Complex Geometries: Application to the Human Lungs. In *Euro-Par 2010 Parallel Processing Workshops.*
6. 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.
7. Federico Massaioli and Giorgio Amati. Achieving high performance in a LBM code using OpenMP. *Unknown.*
8. Pohl, T., Deserno, F., Thurey, N., Rude, U., Lammers, P., Wellein, G., Zeiser, T. Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures. In *Supercomputing 2004, Proceedings of the ACM/IEEE SC2004 Conference*, page 21, 2004.
9. P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. *19th European Symposium on Algorithms*, 2011.
10. P. Skordos. Initial and boundary conditions for the Lattice Boltzmann Method. *Phys. Rev. E*, 48(6):4823–4842, 1993.
11. M. C. Sukop and D. T. Thorne. *Lattice Boltzmann modeling.* Springer, 2006.
12. 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.
13. T. Zeiser, J. Götz, and M. Stürmer. On performance and accuracy of lattice Boltzmann approaches for single phase flow in porous media: A toy became an accepted tool - how to maintain its features despite more and mor complex (physical) models and changing trends in high performance computing!? In N. Shokina et al. M. Resch, Y. Shokin, editor, *Proceedings of 3rd Russian-German Workshop on High Performance Computing, Novosibirsk, July 2007, Springer*, 2008.