A Simulation Suite for Lattice-Boltzmann based Real-Time CFD Applications Exploiting Multi-Level Parallelism on modern Multi- and Many-Core Architectures

Markus Geveler[a], Dirk Ribbrock[a,*], Sven Mallach[b], Dominik Göddeke[a]

[a]*Institut für Angewandte Mathematik (LS3), TU Dortmund, Vogelpothsweg 87, D-44227 Dortmund, Germany*
[b]*Institut für Informatik, Universität zu Köln, Pohligstr. 1, D-50969 Cologne, Germany*

Abstract

We present a software approach to hardware-oriented numerics which builds upon an augmented, previously published set of open-source libraries facilitating portable code development and optimisation on a wide range of modern computer architectures. In order to maximise efficiency, we exploit all levels of parallelism, including vectorisation within CPU cores, the Cell BE and GPUs, shared memory thread-level parallelism between cores, and parallelism between heterogeneous distributed memory resources in clusters. To evaluate and validate our approach, we implement a collection of modular building blocks for the easy and fast assembly and development of CFD applications based on the shallow water equations: We combine the Lattice-Boltzmann method with fluid-structure interaction techniques in order to achieve real-time simulations targeting interactive virtual environments. Our results demonstrate that recent multi-core CPUs outperform the Cell BE, while GPUs are significantly faster than conventional multi-threaded SSE code. In addition, we verify good scalability properties of our application on small clusters.

Keywords: HPC software development, multi- and many-core architectures, Real-time simulation, Lattice-Boltzmann methods, fluid-structure interaction

1. Introduction and Motivation

Due to the pervasive nature of computers and due to the increase in data being acquired for fast ad-hoc processing or future analysis in many different fields, applications hunger for ever larger amounts of processing and memory resources. In order to numerically solve practical problems in engineering and science, parallel systems are essential to match modern applications' high demands in terms of compute time and memory. In addition, the memory wall (bandwidth and latency increase much slower than raw compute performance), the power wall (performance per Watt, 'green computing') and the ILP wall (*instruction level parallelism*, compilers and dedicated circuits are no longer able to extract sufficient parallelism from instruction streams to saturate the arithmetic units) form a 'brick wall' [1], and performance is no longer increased by frequency scaling, but by parallelisation and specialisation. Hardware parallelism can be found on the microprocessor level in the form of multi- and many-core architectures and furthermore, in terms of data-vectorisation and on the instruction level. Multiple CPUs and accelerators like GPUs or Cell BEs can be clustered into a single processing node and these hybrid computation nodes can be clustered as well, communicating via commodity or special network interconnects, and even clusters are connected in Grid- or Cloud-computing networks. Looking at single processors, we see that commodity CPUs have up to twelve cores, the Cell processor is heterogeneous, and throughput-oriented fine-grained parallel designs like GPUs are transitioning towards becoming viable general purpose compute resources.

*Corresponding author
Email address: dirk.ribbrock@math.tu-dortmund.de (Dirk Ribbrock)

Concerning HPC software in general and especially numerical codes, programming models for fine-grained parallelism are discussed and augmented frequently. Due to sophisticated compiler support for the development and optimisation for parallel hardware being unavailable or at least unable to extract a reasonable amount of the peak performance, developers have to adapt to specific programming models for specific hardware architectures. Therefore, the need for parallel programming tools and strategies increases continuously.

The parallelisation and heterogenisation of resources imposes a major challenge to the application developer who often primarily concentrates on advancements in domain-specific methodology. But in order to extract as much performance as possible from the hardware, any software targeting parallel systems has inevitably to be designed with respect to hardware details. This necessity is ever more an economic issue, due to sophisticated applications being more and more highly interdisciplinary, requiring specialists from the mathematical, natural science, engineering and computer science domains. Consequently, software approaches that aim at hardware-abstraction become more and more important so that application programmers can (continue to) focus on domain specific methodology.

We are convinced that the design of *efficient* parallel numerical codes requires both a feasible *method*, consisting of parallel data structures and algorithmic building blocks, and their *hardware-aware implementation*, respectively.

1.1. Paper Contribution and Paper Overview

In a previous publication [2] we have described the design philosophy of the HONEI libraries which aim at facilitating and simplifying application development on different hardware architectures: Their goal is to encapsulate hardware specific tuning transparently so that application developers do not have to re-implement everything from scratch for each architecture. Obviously, this is not always possible, and a set of architecture abstractions and infrastructure is additionally provided to counteract this issue as much as possible, see section 3.2.1. To this end, the HONEI libraries strictly separate frontends (application code) from (hardware-specific) backends. We have evaluated our approach with a multigrid solver for the prototypical Poisson problem, and an explicit finite difference solver for the *shallow water equations* (SWE). In this paper, we go significantly further and apply newly developed and improved backends [3, 4] to the more challenging task of interactive fluid dynamics (CFD) using a coupled *Lattice-Boltzmann* (LBM) *fluid structure interaction* (FSI) application. We aim at exploiting the full performance of hybrid single nodes, equipped with CPUs and GPUs or the Cell BE, an important prerequisite towards efficient large-scale computations. Finally, we have examined the additional level of parallelism for distributed memory clusters [5], which by design can easily be added on top of our single-node code.

The remainder of this section is dedicated to related work in section 1.2. In section 2, we provide a detailed description of our LBM-based solver for the inhomogeneous SWE, including the modifications we applied to incorporate dry states and nontrivial bed topographies (section 2.5) as well as FSI (section 2.6). The implementation of the general framework as well as the new modular set of CFD kernels is presented in section 3 and results concerning numerical properties of the presented toolkit as well as parallel performance experiments are given in section 4. We conclude in section 5 with a summary and an outlook to future work.

1.2. Related Work

An important paradigm throughout this paper is *hardware-oriented numerics*, which is best summarised as the goal of simultaneously maximising numerical (convergence rates, advanced methodology), computational (good exploitation of the hardware) and parallel (scalability) efficiency. Many publications are concerned with this approach, we exemplary refer to Keyes and Colella et al., who survey trends towards terascale computing for a wide range of bandwidth-limited applications and conclude that only a combination of techniques from computer architecture, software engineering, numerical modelling and numerical analysis will enable a satisfactory and future-proof scale-out on the application level [6, 7]. The FEAST toolkit by Turek et al. [8] pursues hardware-oriented numerics for finite-element based discretisations applied to problems from continuum mechanics.

Many publications are concerned with real-time[1] CFD simulations based on the shallow water equations, and we exemplarily refer to Brodtkorb et al. who approach the problem with a finite difference discretisation and who utilise

[1] In the scope of this paper, we use the term *real-time* synonymously to *interactive*, i.e., results being computed with a frequency of at least 30 frames per second or higher.

GPUs in order to overcome the problem of huge simulation times in the forecast of floods [9]. They also released some very close in spirit work concerning the design of BLAS-like APIs targeting heterogeneous parallel systems [10].

The WaLBerla and VirtualFluids frameworks by Rüde et al. and Krafczyk et al. are prominent examples of parallel Lattice-Boltzmann based simulators [11, 12]. Recent improvements in the scope of our work target Lattice-Boltzmann kernels on multi-core systems [13], their GPU acceleration [14], real-time simulations and advanced Lattice Boltzmann techniques [15, 16]. Finally, Fan et al. [17] were the first to implement a Lattice-Boltzmann solver on a cluster of GPUs.

The simultaneous exploitation of different levels of parallelism is also pursued by Chorley and Walker [18], and in addition to already referenced work also by Williams et al. and Pohl et al. for the LBM [19, 20].

2. Real-time Fluid Dynamics with the LBM

2.1. Motivation

As a typical representative of a numerical problem with real-time constraints and thus arbitrarily large hardware efficiency requirements, we consider the simulation of fluid flow in a virtual environment at interactive rates. In this setting, timing constraints are imposed by the fact that time-dependent solutions have to be provided fast enough to satisfy a reasonably high frame-rate. Applications containing such computations may range from computer graphics and computer games to engineering simulations for rapid prototyping and even the 'faster-than-real-time' forecast of, e. g., tsunami waves. The functional requirements of a simulation software allowing user interaction are twofold: (1) The solver for the governing flow equations must be able to react to changes within its embedding environment, and (2) the environment itself has to be able to react to the fluid's motion.

However, due to the fact that the full numerical 3D simulation of a fluid is quite demanding in terms of computational resources, it is still a rare feature in interactive simulation software. Further problems arise with stability, since all possible states of the fluid can not be known a priori.

In the scope of this paper, we concentrate on the interaction of an incompressible laminar fluid with rigid bodies, i. e., all non-deformable objects that the fluid might interact with in the course of the simulation. The fluid solver has to fulfil both requirements (1) and (2) (see above), where we focus on the reaction of the fluid, which means, that we are especially interested in an approximation of a fluid that can flow over arbitrary geometries (in other words: that can flow anywhere in the virtual scene) and which can react on self-propelled solids moving through it.

2.2. General Approach

Two potentially mutually exclusive constraints arise for the specification of the solver described so far: On the one hand, there is the real-time constraint in order to enable interactive rates, and on the other hand, the fluid has to be simulated with good visual results, which requires a reasonable high resolution of the discretisation in use. Hence, any high-level approach, which reduces the application's demand for computational and memory resources with good (qualitative) approximation of the fluid is feasible: In many practical situations, the behaviour of a laminar fluid can be modelled by the *shallow water equations (SWE)*, e. g., for tidal flows, open water waves (such as tsunamis), dam break flows and open channel flows (such as rivers). In such cases, vertical acceleration of the fluid is negligible because the flow is dominated by horizontal movement, with its wavelength being much larger than the corresponding height. In the SWE, vertical velocity is replaced by a depth-averaged quantity, which leads to a significant simplification of the general flow equations (like the Navier-Stokes equations which are derived from general conservation and continuity laws). In the inhomogeneous SWE, source terms are employed to internalise external forces, e. g., wind shear stress and, more importantly, forces resulting from the interaction between the fluid and the *bed topography*. By employing such source terms, the two-dimensional SWE can be used for the simulation of a fluid given by its free surface in three space dimensions, which significantly reduces the computational cost and makes them a popular method for instance in (ocean-, environmental- and hydraulic) engineering.

Using the SWE as a starting point for our method, the simulation consists of the repeated application of the following steps:

1. Parameterisation of 3D rigid bodies in the scene with respect to their position relative to the fluid's surface
2. Rasterisation of their 2D representations with respect to the discretisation in use

3. **CFD solution with *Fluid Structure Interaction* (FSI) including the computation of forces acting on all solids**
4. Reaction of the solid objects on the forces, computed by a structural mechanics solver
5. Rendering and/or other further processing of the resulting data

As stated at the beginning of this section, we concentrate on the third step, starting with a given rasterisation of the solids and implementing the solid reaction by a black-box structural mechanics solver tailored to the specific scenario. The correctness can nonetheless be verified by restricting the analysis on self-propelled objects and examining the forces acting on them separately. Any resulting movement of floating or diving solids may be either ignored (e. g. applicable to any objects with no or negligible change in alignment to the artificial horizon) or translated into a three-dimensional movement by evaluating the forces acting on the boundaries with respect to the fluid's depth at specific control points.

The remainder of this section is organised as follows: In sections 2.3 and 2.4, we briefly introduce the homogeneous SWE and an associated Lattice-Boltzmann method (LBM). Section 2.5 is dedicated to extending the method to the inhomogeneous SWE. This is done on the model level by incorporating a source term in the SWE and on the discrete level, by applying a force term to the LBM and stabilising flows over arbitrary geometry allowing the dynamic drying and wetting of the scene geometry. Finally a description of the extensions for FSI is presented in section 2.6.

2.3. Approximation of the Fluid's Surface with the Shallow Water Equations

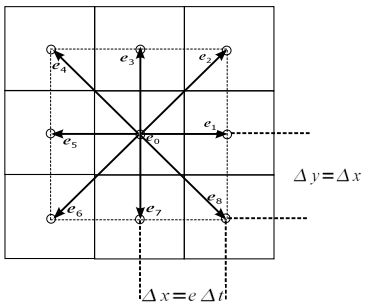
Using the Einstein summation convention (subscripts i and j are spatial indices) the two-dimensional, homogeneous shallow water equations, i. e., the problem to be solved, read in tensor form

$$\frac{\partial h}{\partial t} + \frac{\partial (hu_j)}{\partial x_j} = 0 \quad \text{and} \quad \frac{\partial hu_i}{\partial t} + \frac{\partial (hu_i u_j)}{\partial x_j} + g\frac{\partial}{\partial x_i}(\frac{h^2}{2}) = 0, \tag{1}$$

where h is the fluid depth, $\mathbf{u} = (u_1, u_2)^T$ its velocity in x- and y-direction, t is the elapsed time and g denotes the gravitational acceleration.

2.4. Lattice-Boltzmann Method for the 2D SWE

In order to solve problem (1) with some initial conditions $h(\mathbf{x}, t = 0)$, $\mathbf{u}(\mathbf{x}, t = 0)$ and a *planar* bed topography, $b(\mathbf{x}) = \text{const}$, we apply the Lattice-Boltzmann method (LBM) with a suitable equilibrium distribution to recover the SWE. In the LBM, the fluid behaviour is determined by particle populations residing at the sites of a regular grid the so-called *lattice*. The particles' movement (*streaming*) is restricted to fixed trajectories $\mathbf{e}_\alpha$ (*lattice velocities*) defined by a local neighbourhood on the lattice. We use the D2Q9 lattice, which defines the lattice velocities in the direction of the eight spatial neighbours as depicted in figure 1 and as specified by equation (2) with $e = \frac{\Delta x}{\Delta t}$ being the ratio of lattice spacing and time step.



$$\mathbf{e}_\alpha = \begin{cases} (0,0) & \alpha = 0 \\ e(\cos\frac{(\alpha-1)\pi}{4}, \sin\frac{(\alpha-1)\pi}{4}) & \alpha = 1,3,5,7 \\ \sqrt{2}e(\cos\frac{(\alpha-1)\pi}{4}, \sin\frac{(\alpha-1)\pi}{4}) & \alpha = 2,4,6,8, \end{cases} \tag{2}$$

Figure 1: Discrete phase space of the D2Q9 model

Particle behaviour is defined by the Lattice-Boltzmann equation and a corresponding *collision* operator. Here, the Lattice-Bhatnagar-Gross-Krook (LBGK) collision operator [21, 22] is used, which is a linearisation of the collision-integral around its equilibrium state with a single uniform relaxation time τ. Using this relaxation, the Lattice-Boltzmann equation can be written as

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq}),\ \alpha = 0, \ldots, 8, \tag{3}$$

where f_α is the particle distribution corresponding to the lattice-velocity $\mathbf{e}_\alpha$ and f_α^{eq} a local equilibrium distribution, which defines the actual equations that are solved. In order to recover the SWE, a suitable f_α^{eq} has to be defined for every lattice-velocity. It can be shown (see for instance Zhou [23]) that the equilibria can be written as

$$f_\alpha^{\mathrm{eq}} = \begin{cases} h(1 - \frac{5gh}{6e^2} - \frac{2}{3e^2}u_i u_i) & \alpha = 0 \\ h(\frac{gh}{6e^2} + \frac{e_{\alpha i}u_i}{3e^2} + \frac{e_{\alpha j}u_i u_j}{2e^4} - \frac{u_i u_i}{6e^2}) & \alpha = 1, 3, 5, 7 \\ h(\frac{gh}{24e^2} + \frac{e_{\alpha i}u_i}{12e^2} + \frac{e_{\alpha j}u_i u_j}{8e^4} - \frac{u_i u_i}{24e^2}) & \alpha = 2, 4, 6, 8 \end{cases} \tag{4}$$

and that the SWE can be recovered by applying Chapman-Enskog expansion on the LBGK approximation (3). Finally, macroscopic mass (fluid depth) and velocity can be obtained by

$$h(\mathbf{x}, t) = \sum_\alpha f_\alpha(\mathbf{x}, t) \quad \text{and} \quad u_i(\mathbf{x}, t) = \frac{1}{h(\mathbf{x}, t)} \sum_\alpha e_{\alpha i} f_\alpha, \tag{5}$$

respectively. We use the popular bounce-back rule as boundary conditions, where particles are reflected using opposite outgoing directions and which therefore implements no-slip boundary conditions. This leads to second order accuracy at the boundaries, when aligned with the cartesian coordinate system.[2] In the following, this basic method is used as a starting point for our more sophisticated solver, capable of dealing with complex flow scenarios, including the interaction with moving solid obstacles.

2.5. Scene Interaction I: Bed Topography and Dry-States

In order to enable simulations with an arbitrary bed topography, we apply a source term S which leads to the inhomogeneous SWE:

$$\frac{\partial h}{\partial t} + \frac{\partial (h u_j)}{\partial x_j} = 0 \quad \text{and} \quad \frac{\partial h u_i}{\partial t} + \frac{\partial (h u_i u_j)}{\partial x_j} + g \frac{\partial}{\partial x_i}(\frac{h^2}{2}) = S_i, \tag{6}$$

with

$$S_i = -g \left(h \frac{\partial b}{\partial x_i} + n_b^2 h^{-\frac{1}{3}} u_i \sqrt{u_j u_j} \right) \tag{7}$$

The slope term is defined by the sum of the partial derivatives of the bed topography, weighted by gravitational acceleration and fluid depth (b denotes the bed elevation), and we define the friction term using the Manning equation as suggested by Zhou [23], with n_b denoting the material-dependent Manning constant.

The source term can be applied as an additive term to the collision operator in the Lattice-Boltzmann equation:

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq}) + \frac{\Delta t}{6e^2} e_{\alpha i} S_i,\ \alpha = 0, \ldots, 8. \tag{8}$$

The LBM for the shallow water equations presented above can interact with the bed surface and therefore is not restricted to simple scenarios (as it would be if an equilibrium distribution function corresponding to the two-dimensional Navier-Stokes equations had been used). However, it is restricted to subcritical[3] flows, i. e., when the fluid depth is significantly greater than zero. The first extension to the method aims at allowing so-called *dry-states*,

[2] Zhou shows in the third and fourth chapters of his book, that the LBM is second order accurate for the recovered macroscopic equations when using a suitable form of the source term, see section 2.5.

[3] Usually, the term *critical flow* is associated with a Froude number being smaller than one. Throughout this paper, we use it for flows over a bed topography with possibly very small (or even zero-valued) fluid depth, as commonly done in the SWE literature.

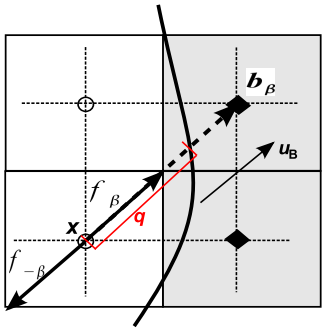


Figure 2: Modified *bounce-back* scheme for moving boundaries.

since the dynamic drying and wetting of the bed topography is a feature desired by many applications. In our approach, we define a fluid depth below a specified small threshold parameter as *dry* and set the macroscopic velocity at dry sites to zero, to avoid the division by zero in the extraction phase of the original algorithm (the evaluation of equation (5)). Secondly, local oscillations caused by critical non-zero fluid-depths are confined with a limiter approach: The usage of a MinMod-like limiter, i. e.,

$$\phi_U(x) = \max(-U, \min(U, x)) \tag{9}$$

with U defined as close as possible to the maximum fluid depth expected locally can be incorporated in the LBM extraction phase (5) to cut off the noise plateau at fluid sites close to zero:

$$u_i(\mathbf{x}, t) = \begin{cases} \frac{1}{h(\mathbf{x},t)} \sum_\alpha e_{\alpha i} f_\alpha(\mathbf{x}, t), & h(\mathbf{x}, t) > \epsilon, \\ \phi_U(\frac{1}{h(\mathbf{x},t)}) \sum_\alpha e_{\alpha i} f_\alpha(\mathbf{x}, t), & \text{otherwise.} \end{cases} \tag{10}$$

This procedure reduces large variations in fluid velocity by limiting the reciprocal of h to a positive, non critical value $U > \epsilon$ at all fluid sites, where fluid depth is critical (i.e. $h < \epsilon$), see the numerical test in section 4.1, figure 7(b).

2.6. Scene Interaction II: Moving Rigid Bodies

In order to simulate rigid bodies moving within the fluid, the method described so far is extended by three major algorithmic steps: In the first step, the forces acting on the fluid due to a moving boundary have to be determined. We use the so-called BFL-rule [24], which interpolates the momentum values for the consistency with non-zero Dirichlet boundary conditions induced by a moving boundary. The interpolation is achieved by taking values in opposite direction of the solid movement similar to the bounce-back rule, see figure 2. In the original BFL formulation, the interpolation needs four coefficients depending on the distance $q = \frac{|\mathbf{b}_\beta - \mathbf{x}|}{\Delta x}$ where $\mathbf{e}_\beta$ is the lattice-velocity approximating the direction of the solid movement, $\mathbf{b}_\beta$ the corresponding point on the solid boundary and $\mathbf{x}$ a location in the fluid (and on the lattice) in opposite direction:

$$\begin{aligned} f^{\text{temp}}_{-\beta}(\mathbf{x}, t + \Delta t) = C_1(q) f_\beta(\mathbf{x}, t) + C_2(q) f_\beta(\mathbf{x} + \mathbf{e}_{-\beta}, t) + C_3(q) f_{-\beta}(\mathbf{x}, t) + \\ C_4(q) 2\Delta x w_{-\beta} c_s^{-2} [\mathbf{u}_B(\mathbf{b}_\beta) \cdot \mathbf{e}_{-\beta}], \end{aligned} \tag{11}$$

where $c_s = 1/\sqrt{3}$ is the lattice speed of sound. In our approach, we use a piecewise linear approximation of the boundary, and set $q = \frac{1}{2}$, which reduces three of the four coefficients C_i to zero. We obtain a very simple formula[4] for the missing momentum, that still respects the moving boundary. Denoting the macroscopic velocity of the solid by $\mathbf{u}_B$, our modified boundary condition then reads:

$$f^{\text{temp}}_{-\beta}(\mathbf{x}, t + \Delta t) = 6\Delta x \, w_{-\beta} (\mathbf{u}_B(\mathbf{b}_\beta) \cdot \mathbf{e}_{-\beta}). \tag{12}$$

[4]It should be noted that this simplification increases performance but reduces the spatial approximation order to one.

The superscript temp indicates the distribution function after the collision step. In both approaches, the w_α are weights depending on the lattice, which can be set to $\frac{4}{9}$ for $\alpha = 0$, $\frac{1}{9}$ for uneven α and $\frac{1}{36}$ in the even case for our D2Q9 lattice, see Caiazzo [25].

The second major step in performing FSI is the extrapolation of missing macroscopic quantities. Moving solids imply that the lattice is in general not invariant over time: Lattice sites that belong to a solid region at time t may become fluid sites at time $t + \Delta t$. In this case, the missing quantities have to be reconstructed. We use an indirect so-called *equilibrium refill* method proposed for example by Caiazzo [25], which uses a three point-backward approximation after calculating the opposite direction of the solid's movement in order to use one-dimensional extrapolation only. We denote the extrapolated quantities with $\tilde{h}$ and $\tilde{\mathbf{u}}$ respectively, which can be written as

$$\tilde{h}(\mathbf{x}, t + \Delta t) = 3h(\mathbf{x} + \mathbf{e}_{-\beta}\Delta t, t + \Delta t) - 3h(\mathbf{x} + 2(\mathbf{e}_{-\beta}\Delta t), t + \Delta t) + h(\mathbf{x} + 3(\mathbf{e}_{-\beta}\Delta t), t + \Delta t) \tag{13}$$

and

$$\tilde{\mathbf{u}}(\mathbf{x}, t + \Delta t) = 2\Delta x \frac{\mathbf{u}_B(\mathbf{b}_\beta, t + \Delta t)}{q^2 + 3q + 2} + 2q \frac{\mathbf{u}(\mathbf{x} + \mathbf{e}_{-\beta}\Delta t, t + \Delta t)}{q + 1} - 2q \frac{\mathbf{u}(\mathbf{x} + 2(\mathbf{e}_{-\beta}\Delta t), t + \Delta t)}{q + 2} \tag{14}$$

Again, we substitute $q = \frac{1}{2}$ in equation (14) and we obtain

$$\begin{aligned} \tilde{\mathbf{u}}(\mathbf{x}, t + \Delta t) &= \tfrac{8}{15}\Delta x \mathbf{u}_B(\mathbf{b}_\beta, t + \Delta t) + \tfrac{2}{3}\mathbf{u}(\mathbf{x} + \mathbf{e}_{-\beta}\Delta t, t + \Delta t) \\ &- \tfrac{2}{5}\mathbf{u}(\mathbf{x} + 2(\mathbf{e}_{-\beta}\Delta t), t + \Delta t) \end{aligned} \tag{15}$$

for the macroscopic velocities.

Finally, the force acting on the solid due to fluid movement is determined by the *Momentum-Exchange algorithm* (MEA) [26], in order to be able to couple the method with a solid mechanics (CSM) solver. The MEA uses special distribution functions to compute the moments resulting from incoming particles and outgoing particles corresponding to a single lattice-velocity $-\beta$ at a solid (boundary) point $\mathbf{b}$:

$$f^{\mathrm{MEA}}_{-\beta}(\mathbf{b}, t) = (e_\beta)_i (f^{\mathrm{temp}}_{\beta}(\mathbf{x}, t) + f^{\mathrm{temp}}_{-\beta}(\mathbf{x}, t + \Delta t)). \tag{16}$$

The forces can be aggregated into the total force acting on $\mathbf{b}$:

$$F(\mathbf{b}, t) = \sum_\alpha f^{\mathrm{MEA}}_\alpha(\mathbf{b}, t). \tag{17}$$

3. System Design and Implementation

As motivated in the introduction, modern multi- and many-core architectures comprise multiple levels of parallelism stemming from superscalar features, SIMD instructions, multiple cores, and simultaneous multi-threading on a single chip. Though the latter implies multiple execution contexts per processor core, we –for the sake of simplicity– use the terms 'core' and 'execution unit' synonymously within this section.

Taking a look at state-of-the-art processors with their increasing numbers of cores, heterogeneities implied by the hardware design steadily surge. Either this is because of different functional capabilities, like, e. g., within the elements of the Cell BE architecture, or due to non-uniform access to on- and off-chip memory. Today, non uniform memory access (NUMA) is often encountered in multiprocessor systems in scientific and high performance computing. Here, the achieved performance considerably varies with the encountered memory access pattern [27]. At the same time, the well-known memory wall has become an ever larger limiting obstacle, as currently sustainable memory bandwidths do not suffice to serve all cores situated on a single chip simultaneously [4, 28]. In particular, this is a rigorous restriction for a wide range of so-called *memory-bound* applications performing only a few computational instructions per read/write access.

Even though the reasons vary, most challenges similarly lie in an improved exploitation of the sustained bandwidth and minimising latencies. Therefore, in order to detect and best exploit the computational potential, one needs a hardware-aware approach that automatically adapts to the underlying system. A corresponding thread management should be able to automatically investigate the systems' hardware and offer an interface comprising alternative strategies for different memory architectures, processor topologies and applications.

3.1. Hardware Backends

To address the challenges outlined so far, HONEI [2] comprises multiple *backends* that encapsulate the architecture-specific parts of execution such as task dispatching and synchronisation. In comparison to previous publications, a new multi-core backend has been implemented from scratch, the GPU backend now supports multiple devices as well as transfers between them, and a new MPI backend [3, 5] allows for the use of distributed memory systems. At the top level all backends targeting parallelism follow a similar design. For every corresponding device, e. g., a processor core, a GPU or an SPE in the Cell BE, (at least) one POSIX thread is maintained. Combined with data structures that allow recursive partitioning, this modular design enables the use of existing optimised single core operations within the device threads. Furthermore, every *frontend* (see section 3.2) that uses one of the backends to perform a task calls a function to enqueue it in the corresponding task queue and receives an object that we call a *ticket*. Tickets mainly keep track of the state of a task, i. e., whether it has been finished. Its receiver may use it to (asynchronously) wait for single tasks to be completed before continuing with further actions. Listing 1 shows enqueuing and synchronisation for a simple multi-core solver. Note that in this example `_solver_list` contains the single solver objects, that shall be executed in the different threads. Besides this global functionality, the backends specialise their implementation of the ticket concept with further information, i. e., the multi-core backend maintains the thread which in fact executes the task.

```
TicketVector tickets;
for (unsigned long i(0) ; i < _parts ; ++i)
{
    tickets.push_back(honei::mc::ThreadPool::instance()->enqueue(
                std::tr1::bind(
                    std::tr1::mem_fn(&honei::SolverLBMGrid<tags::CPU::SSE>::solve),
                    *(_solver_list.at(i))
                    )));
}
tickets.wait();
```

Listing 1: Task dispatching in a CPU LBM solver

Another global property of these backends is its consistent implementation of the pooling paradigm. Threads created by the multi-core backend, as well as SPE kernel threads in the Cell backend or threads that orchestrate the computation on a GPU, are stored for rapid reuse after they have been created.

3.1.1. Commodity x86 CPU Single- and Multi-Core Backends

In order to address the demands of a hardware-aware approach, the implemented multi-core backend combines two key concepts, namely thread affinity and an automatic investigation of the compute node's topology [4]. Because operating systems tend to schedule multiple threads to one of the available cores, especially for small tasks, or need some time slices to move threads to less loaded execution units [4], we use thread affinity to bind threads to distinct processor cores. Our abstraction also enables users or application specific kernels to easily pin single tasks to distinct cores or enforce cache-reuse by using the same core for tasks that operate on the same portion of data. Here, the ticket system provides a history which thread processed/performed a preceeding task and makes it possible to bind them to new ones. As already indicated, threads are pooled to be quickly and easily reused. This widely applied concept avoids the overhead of creation in subsequent parallel computations and allows the development of more advanced dispatch strategies aware of the hardware by, e. g., sorting available threads according to the topology of the cores they are executed on.

In order to realise the investigation of the system topology, a lot of information is collected, e. g., by using OS calls or reading from the sys-filesystem if run under Unix/Linux. This allows the multi-core backend to derive detailed information about the number of available cores, which cores are situated on the same chip, and whether the system is a non uniform memory architecture (NUMA) or not. Combined with the affinity concept, this enables an optimised selection of threads, i. e. cores, for every task that is to be dispatched. By using the corresponding interface functions, application kernels may instruct the backend to spread work over multiple chips or to keep it local to one chip in order to exploit locality, prevent remote memory access patterns or optimise bandwidth utilisation. This approach is especially beneficial when large-scale NUMA computers with a high ratio between remote and local memory access

latencies (the so-called *NUMA-factor*) are used. We call such instructions to the backend a *dispatch policy*. The backend especially offers functions like `DispatchPolicy::on_node(id)` and `DispatchPolicy::on_core(id)` to explicitly specify the target processor or core for a given task. Subsequent tasks can inherit the same dispatch constraints from former tasks by using similar routines. If, for instance, a task to be executed includes a lot of memory allocation which is consequently thread-local, these functions enable bandwidth optimisation, e. g., by alternating NUMA nodes. Similarly, if only sequentially initialised memory is accessed within a parallel section, a strategy that improves locality is more appropriate.

3.1.2. GPU Backend

The GPU backend is based on NVIDIA CUDA [29] and provides simplified access to any CUDA enabled GPU. To avoid resource conflicts and expensive context switching, a host thread is executed for every GPU device. This thread has exclusive access to the corresponding CUDA device context and thus no context switching is necessary. As stated in Section 3.1, work can be dispatched to these threads via the ticket system.

To ease the use of external device memories, a memory transfer scheduler works behind the scenes and transparently processes all necessary exchanges between the device memories and the host memory. The scheduler ensures used data in the different memories to be up to date, but postpones transfers until they are unavoidable. Since it is not possible to directly move data from one GPU's device memory into another, data needs to be transferred via the main memory which makes these transfers twice as expensive as a standard device to host memory transfer.

3.1.3. Cell Backend

The implementation of the Cell backend respects the heterogeneous design of the Cell BE and therefore consists of two instances addressing the different purposes of its execution units. On the one hand, the architecture comprises a 'controller' unit which has an instruction set similar to a PowerPC processor and is therefore called *Power Processing Unit* (PPE). On the other hand there are eight SIMD units, called *Synergistic Processing Elements* (SPE). The different processors can communicate using DMA transfers via the so-called Element Interconnect Bus.

The Cell backend implements a management infrastructure on top of `libspe2`, a library provided by IBM [30], using a steering thread on the PPE and a worker thread on each of the SPEs. The typical use-case is to create a list of SPE-tasks (called `SPEInstruction`) and hand it over to the PPE thread which will successively distribute them to the SPEs. Similarly to all other hardware-specific backends, the task-based implementation allows the user or calling applications to asynchronously wait for the completion of the instructions handed over.

As the SPEs are solely able to access their 256 kB *Local Store* (LS) directly, applications operating on large data sets have to make intensive use of DMA transfers, pulling data from the PPE's main memory and pushing the processed data back afterwards using the corresponding `libspe2`-operations. The backend supports these memory transfers by an internal mapping of *effective* and *local store* addresses w.r.t. the type of unit. As the program code to be executed by an SPE also has to be stored in its LS, the backend further provides some pre-implemented *kernels* comprising the operations already available and offers the necessary infrastructure to create new kernels and load them into SPEs.

3.2. Frontends

3.2.1. General Concepts

Application-specific functionality that exceeds basic BLAS-like operators are called *application-specific kernels* (ASK) within our framework. The application-specific kernels, as well as their numerical base operations, are implemented by means of generic template programming. That is, the respective functions may be tagged with a parameter that specifies the target architecture backend which shall execute it. As not all kernels and operations are implemented for all of the mentioned architectures, there is a fall-back mechanism that, if the demanded implementation is not available, executes its single-threaded CPU variant.

Furthermore, the architectural parameters can be combined in a meaningful manner, e. g. one may call the base operation for an operator with the template parameter associated with the multi-core backend, that also utilises the SIMD-fashion vectorised code (SSE) partitioned and threaded on multiple cores. Similarly one may issue operations on a general purpose processor and GPUs at the same time and asynchronously wait for tasks processed by the different devices. The necessary load balancing is performed by the calling application within the frontend.

A major achievement of this modular concept are hybrid applications, utilising multiple CPU cores as well as multiple GPUs. Furthermore, the floating point precision can be varied on a per-operation basis, with automatic format conversions. For instance, the LBM-solver shown in figure 3 uses two GPUs and four CPU cores to process different parts of the domain in parallel. Due to the fact that domain decomposition may be applied recursively, this solver works with three parts on the top level and further decomposes one of these parts into four sub-parts to be handed over to the CPU solvers. In terms of synchronisation, at first the four results of the `CPU::SSE` solvers are merged into one CPU result, and then this result is merged with the two `GPU::CUDA` solver results.

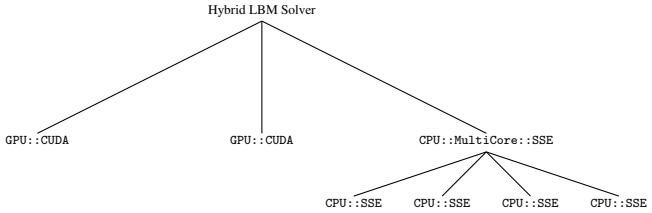


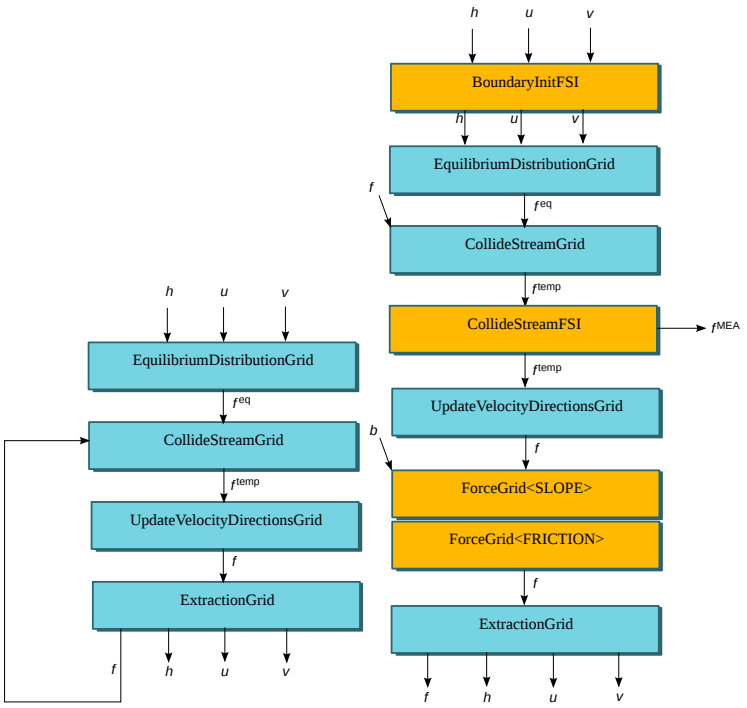
Figure 3: Example of a hybrid solver configuration

3.2.2. Lattice-Boltzmann Library

As outlined so far, multiple threads can be directly mapped to multiple parts after domain decomposition and therefore on different parts of the lattice in the LBM case. Here, we use a straightforward method to decompose the lattice, relying on the fact that the distribution functions and macroscopic scalar fields can be stored linearly in a row-major fashion and that it is therefore possible to simply cut the data at arbitrary positions [5]. This data layout imposes minimal communication between the parts using ghost cells for the synchronisation in which no additional computations take place. In addition, indirect indexing is feasible, which allows contiguous memory storage of data for lattice sites with equal boundary treatment and hence branch-free processing in the streaming-phase. Furthermore, stationary obstacles are not present in the data structures, which reduces the memory footprint. However, since the lattice is not invariant over time in the case of our FSI-capable solver, these advantages do not fully carry over to more sophisticated simulations (see below).

The frontend for LBM-based computations contains a variety of building blocks to assemble the needed functionality. Hence, there is no single 'Lattice-Boltzmann kernel' but rather solver components (application specific kernels, see above) to provide the needed features. For instance, to set up a basic LBGK method for the solution of the Navier-Stokes equations, a suitable kernel for the equilibrium distribution functions and a LBGK module that performs the collision and streaming have to be chosen, as well as kernels for the boundary conditions and the extraction phase. Together with different ASKs for several source terms, boundary conditions and stability related corrections, the FSI modules and kernels for the SWE and Navier-Stokes equations enable the application programmer to easily adjust his solver to the desired functionality and/or performance. Figure 4 exemplary depicts two solver configurations: (a) a basic solver for simple SWE-based simulations and (b) a quite sophisticated one, which includes all functionality described in section 2.

Both solvers use the base modules (with the boundary treatment denoted by `UpdateVelocityDirectionsGrid`). The advanced configuration is additionally constituted by the force term (7) (encapsulated in two distinct ASKs in order to enable friction without the slope term and vice versa), and two modules realising FSI functionality. The kernel named `BoundaryInitFSI` is explicitly employed to determine the fluid sites, which have to be initialised by using equations (13) and (15) and perform these extrapolations. The assignment of `CollideStreamFSI` is threefold: The simplified BFL-rule (12) has to be applied first and the lattice-velocity dependent forces acting on the solid, f_α^{MEA}, are computed using equation (16). The module's third major task is to perform a *backward* streaming after the standard LBGK collision and streaming in order to avoid branch divergence: Due to the fact, that the lattice is altered by fluid sites that become solid sites and vice versa in the course of the simulation, representing moving solids by means of the data structures outlined above is prohibitively expensive, since they require intense preprocessing and even reallocation. To avoid this, we employ the lattice compression technique only for stationary obstacles and use a colouring for the different sets of lattice sites needed by the FSI modules. In addition, and since the LBM library follows the paradigm of reusability of implemented functionality, `CollideStreamFSI` is employed after the



(a) Basic LBGK solver configuration (b) FSI-LBM solver configuration

Figure 4: Different LBM and SWE based solver configurations

associated standard LBGK kernel. Putting these two ideas together, the standard `CollideStreamGrid` module is executed first and stationary objects are treated as described above. Consequently, moving solids are not present in this step and incorrect streaming to solid sites occurs. However this only applies to sites at the solids' boundaries, which can be processed by the FSI kernel afterwards with absolutely no branches, if the solid boundary sites are coloured appropriately. Figure 5 exemplary outlines the backward streaming technique. Note, that the computations of the f_α^{MEA} are executed here, because they have to be applied to all *solid boundary* sites as well and that the BFL-rule can easily be evaluated here, because the *fluid boundary* sites are exactly targeted by the backward streaming.

Another issue for the implementation of the advanced functionality is the evaluation of source terms. Initial numerical experiments have forced us to abstain from using a basic scheme, i. e.,

$$S_i = S_i(\mathbf{x}, t) \tag{18}$$

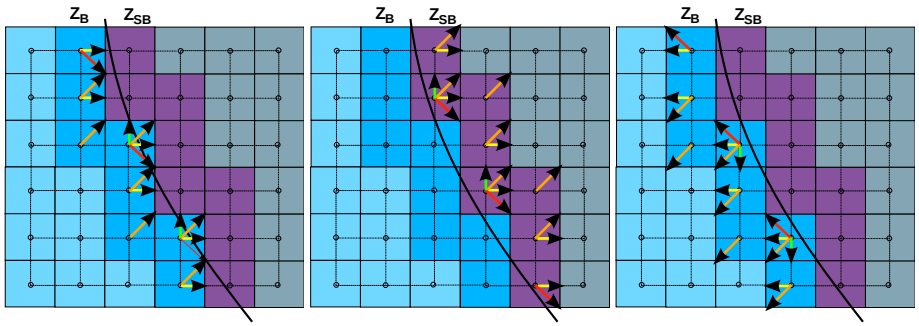
because we were able to confirm Zhou's results concerning the inability of this scheme to achieve planar steady states with the slope term [23]. Since a fully implicit (and therefore accurate) evaluation of the source terms with

$$S_i = S_i(\mathbf{x} + \frac{1}{2}\mathbf{e}_\alpha \Delta t, t + \Delta t). \tag{19}$$

is not feasible due to its inherently sequential character, we followed his proposal to employ a semi-implicit scheme, where the slope is computed at the midpoint between the lattice-site and one neighbouring lattice-site and which therefore includes the interpolation of the fluid depth [23]:

$$S_i = S_i(\mathbf{x} + \frac{1}{2}\mathbf{e}_\alpha \Delta t, t). \tag{20}$$

The evaluation of source terms as well as `BoundaryInitFSI` impose a major disadvantage compared to simple LBGK methods: Although we strongly believe the presented method to be minimally stressing the property of the LBM to be 'well suited for parallel execution' with respect to the predefined functionality constraints, *locality* in terms of accessing only direct neighbours is not preserved by it.



(a) Distribution functions before regular streaming (b) after regular streaming (c) after correction

Figure 5: Backward streaming

4. Results

In the following, we concentrate on numerical aspects of the FSI extensions in section 4.1 and parallel performance on all levels in section 4.2. The benchmarks of the solver configurations include comparisons with basic linear algebra kernels to evaluate different performance and memory characteristics. We refer to previously published work for an analysis in terms of mass conservation, a comparison with external codes and some other aspects not covered here [3, 5].

4.1. Numerical Results

Besides conservation of macroscopic quantities and comparison to state-of-the-art codes, the validation of the presented method has to include numerical properties concerning the extensions described in section 2, since, to the best of our knowledge, it is a unique (yet simple) combination of the described modified boundary treatment and extrapolations with the inhomogeneous SWE and most importantly, with a suitable mechanism to realise dry-states. Therefore, we provide results concerning three aspects; the capability of our advanced solver to deal with arbitrary bed topography, its qualitative correctness with dry-states present and finally, the correct evaluation of forces acting on rigid bodies within the fluid.

We first demonstrate the applicability of our approach with figure 6, which displays the resulting scalar field for $h + b$ and which can in principle be used to render the fluid's surface in 3D. Three typical simulations are depicted: The first row shows the full functionality of the basic solver configuration, simulating a full dam break with arbitrarily shaped stationary obstacles. In the mid row, a large tsunami wave approaches an 'island', which clearly shows the qualitative correctness of our dry-states approach: As the wavefront reaches the obstacle, it breaks and is partially reflected and parts of the dry area are flooded before the tide flows back. Finally, a moving solid with a round cross-section is moving at high speed through the water in non-cartesian direction.

In order to validate the force terms alongside with the employed numerical scheme, we compare the absolute fluid depths computed by different schemes for a standard test scenario also used by Zhou [23]: Figure 7(a) depicts the values for $h + b$ in the steady state along the centre line of a channel with an uneven bed topography and after several full dam breaks. It can be seen, that the applied semi-implicit scheme fully matches the analytical result, whereas the basic scheme is insufficient to obtain the desired planar steady state.

The same scenario is used in order to test our method's capability of coping with dry states. Figure 7(b) illustrates the results for the absolute water depth for different values of the cutoff parameter ϵ (see equation (10)): The larger the ϵ, the less accurate is the result, due to a sharply featured interface between dry and wet sites (emphasised by the vertical lines in the same color as the associated flow), which decreases the quality visually. On the other hand, oscillations occur more often with a small ϵ (e. g., on the left side of the dry area in the red and green plots). In our measurements, we find an ϵ not smaller than 0.001 percent of the characteristic fluid depth offering the best trade-off between stability and quality of the visual result. The same value has been employed in our showcase scenario in figure 6.

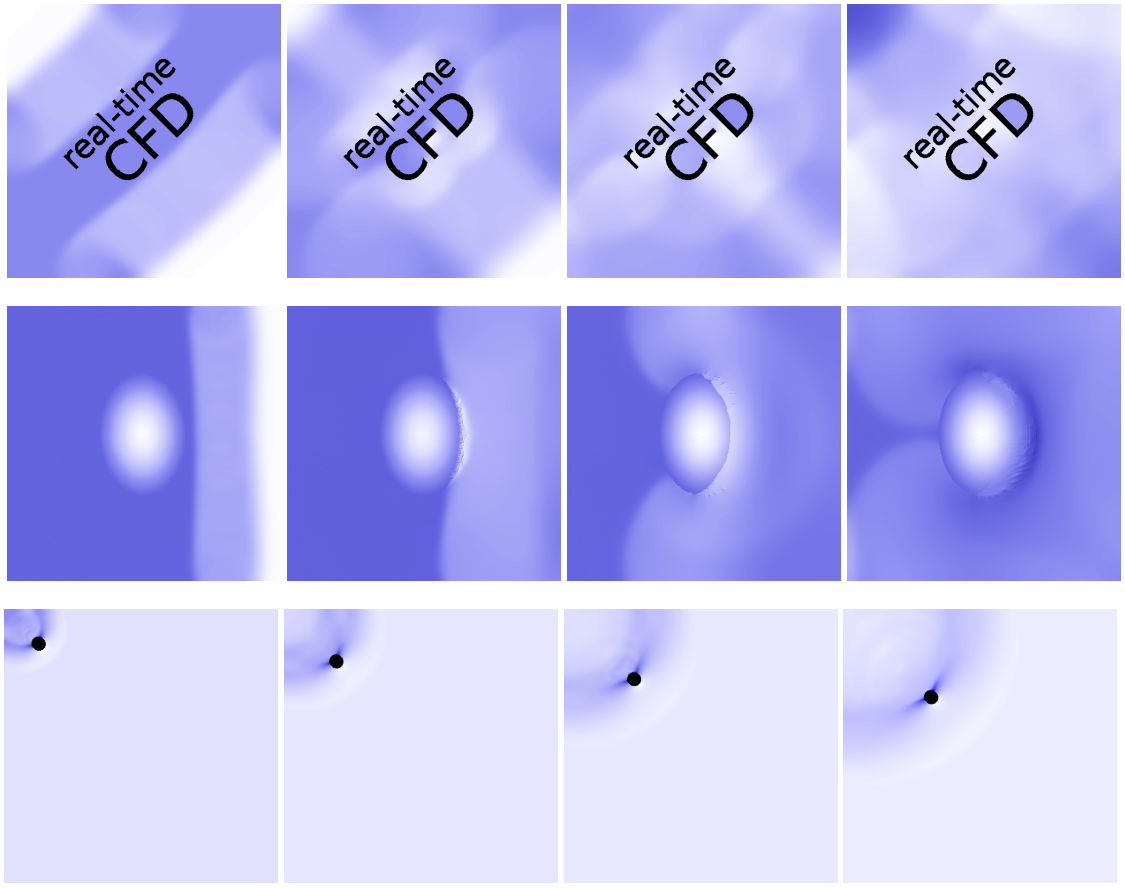
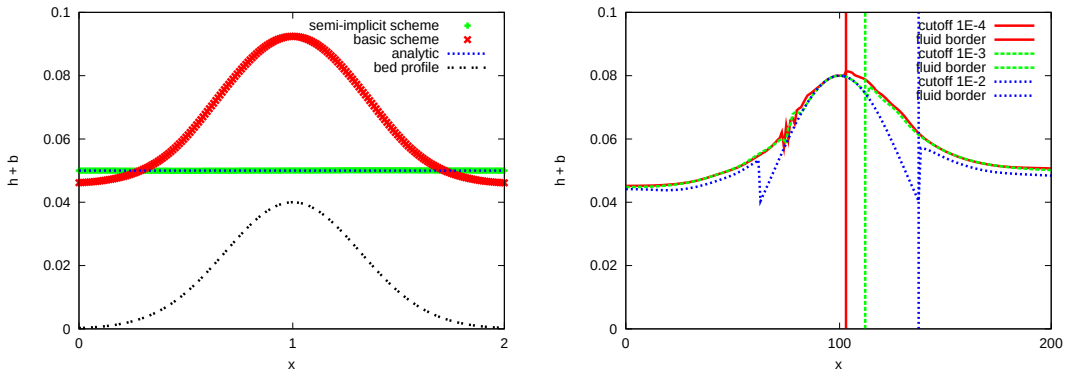


Figure 6: Real-time simulations: Top: Full dam break with arbitrary geometry; Middle row: Tsunami wave flooding an island; Bottom: FSI simulation with a moving solid moving self-propelled through the fluid.



(a) Absolute fluid depth $h + b$ in steady state with different numerical schemes.

(b) Absolute fluid depth $h + b$ during dry-states simulation with different values for ϵ

Figure 7: Numerical results for the FSI-capable LBM solver.

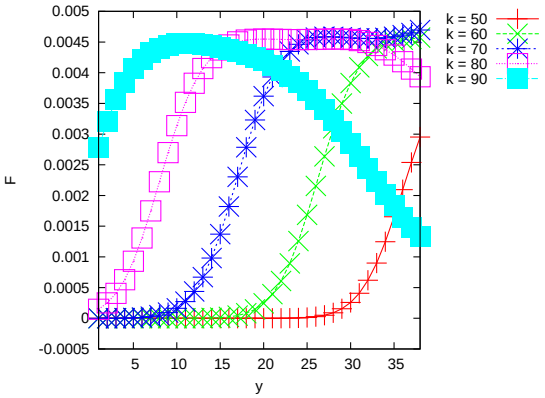


Figure 8: Numerical results for the FSI-capable LBM solver: Total force F acting on a solid at different time steps k.

Finally, the total forces acting on a rigid cuboid's boundary, computed by the Momentum Exchange Algorithm within a dam break simulation are depicted in figure 8. In this simulation, we set up a stationary cuboid in the center of a rectangular domain and initialise a flow that hits it (via a near dam break). We then measure the MEA functions at one of the solid boundary edges at different time steps. Here, we can confirm, that all computed f_α^{MEA} functions are smooth and proportional to the velocities of the flow, even at low resolutions, which enables the solver to be coupled with a more general structural mechanics solver in future work.

4.2. Parallel Performance and Real-Time Simulations

Table 1 contains the benchmark results of our basic LBM solver with a hybrid configuration using up to four CPU cores (Intel Core i7 920 quad-core) and 2 GPUs (NVIDIA GeForce GTX 285), as described in section 3.2.1. The memory bandwidth available to the quad-core CPU is not sufficient to fully serve more than one memory bound thread simultaneously [4]. Nevertheless we achieve a 2.66-fold speedup with four threads in contrast to the 1.40 speedup of the strong memory bound `saxpy` (e. g., $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$) operation. As this bottleneck does not apply to distributed memory systems, e. g. GPUs, we achieve very good speedups using two of them. Since a synchronisation effort, which means expensive memory transfers between the GPUs over the PCI bus, is necessary after each time step, the LBM solver does not reach the speedup of the `saxpy` operation but comes quite close. Finally using the CPU cores as well as the GPUs results in a moderate speedup, thwarted through multiple synchronisation efforts between the CPU threads and the GPU threads. In this configuration, the load balance is based upon the measured memory performance. Here every GPU processes 47 per cent of the whole domain and the four CPUs process only six per cent, appropriate to their computational performance.

Type	Saxpy (64^4 elements)		LBM (1500^2 elements)	
	GFlop/s	Speedup	MFLUPS	Speedup
QS22 Blade	2.4	-	22.36	-
1 CPU core	2.0	-	15	-
4 CPU cores	2.8	1.40	40	2.66
1 GPU	16.5	-	193	-
2 GPUs	32.0	1.94	348	1.80
4 CPU cores + 1 GPU			217	-
4 CPU cores + 2 GPUs			362	1.66

Table 1: Saxpy and LBM benchmarks with different hardware configurations

Scalability experiments show, that the LBM solvers built upon our system scale well on small clusters [5]. Table 2 contains the speedups for our application for increasing numbers of MPI jobs on a four-node dualcore Opteron 2214 cluster, demonstrating very good strong and almost perfect weak scaling.

(a) Strong scaling

#lattice sites	#jobs			
	2	3	4	8
250^2	1.9	2.3	2.4	4.0
500^2	2.1	3.1	4.1	5.8
1000^2	2.1	3.0	3.8	5.9
1500^2	2.0	3.0	4.0	6.4
2000^2	2.0	2.9	4.0	6.5
2400^2	2.0	3.0	4.0	6.5
2800^2	2.1	3.0	4.0	7.6

(b) Weak scaling

#jobs	#initial lattice sites	
	442^2	600^2
1	1	1
2	0.96	1.01
4	1.01	1.02
8	1.01	1.03
16	1.14	1.03

Table 2: Speedup factors for strong and weak scaling on Opteron clusters.

We conclude this section with a discussion of the real-time performance of our advanced FSI capable LBM solver on different GPUs. Figure 9 contains two plots: The execution times on a NVIDIA GeForce GTX 285 and a slightly older GeForce 8800 GTX and the associated limits for a real-time simulation. Here, we assume that a frame rate of 30fps (depicted by the vertical line in figure 9) is sufficient and that rendering or other postprocessing does not take a significant percentage of the execution time[5]. The largest lattice resolution that can be processed in real-time

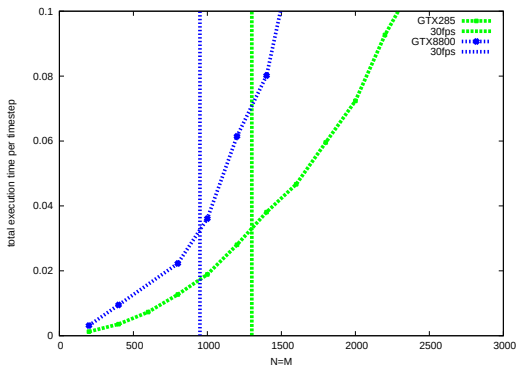


Figure 9: Total execution times of the advanced FSI-capable LBM solver on different hardware architectures.

is 1300×1300 on the GTX 285 graphics accelerator, which is approximately 1.7 million lattice sites and twice as much as can be simulated with the older 8800 card. Although this allows a reasonable visual result at interactive rates, the GPU performance decreases by a factor of approximately three compared to a basic solver. This stems from the fact, that the FSI solver does not benefit from the static lattice compaction to the same extent as the basic solver, because the domain changes in the course of the simulation. Besides the additional computational effort, the loss in performance compared to the basic solver is therefore certainly due to the increase in conditional branches in the code. It should be noted, that since all presented libraries are work in progress and all backends are optimised simultaneously, some optimisations concerning, for example coalesced memory accesses in the LBGK streaming and a performance comparison with state-of-the-art LBM codes are left for future work.

5. Summary and Future Work

We have presented a simulation suite for real-time CFD applications, based on the shallow water equations with suitable source terms and FSI based on Lattice-Boltzmann methods. The approach can be used in many different

[5]Which we believe can be achieved by rendering directly on the GPU and abstaining from additional communication between CPU and GPU. However this assumption does surely only hold true if the solid mechanics solver is implemented in such a way, too. Since we concentrated on the fluid dynamics part of the simulation up to now, applying advanced visualisation methods and coupling the system with a CSM solvers remains for future work.

fields to set up simulators featuring the simultaneous exploitation of multiple levels of parallelism in commodity and unconventional hardware architectures. We have shown that the software has reached a reasonable level of reliability and performance.

In future work, we will explore the performance of the presented hybrid solvers, exploiting all available resources on nodes equipped with CPUs and GPUs, in a MPI cluster environment. In addition, performance comparison with different state of the art approaches to solving the SWE are in progress. Concerning the numerical methods used, we will examine their capabilities of being employed in large-scale simulations associated with the forecast of tsunami waves in coastal regions.

Acknowledgements

We would like to thank Danny van Dyk and all contributors to the HONEI project. This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under the grant TU 102/22-2, and by BMBF (call: HPC Software für skalierbare Parallelrechner) in the SKALB project (01IH08003D / SKALB). Thanks to IBM Germany for access to QS22 blades.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, The landscape of parallel computing research: A view from Berkeley, Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006).

[2] D. van Dyk, M. Geveler, S. Mallach, D. Ribbrock, D. Göddeke, C. Gutwenger, HONEI: A collection of libraries for numerical computations targeting multiple processor architectures, Computer Physics Communications 180 (12) (2009) 2534–2543. doi:10.1016/j.cpc.2009.04.018.

[3] M. Geveler, D. Ribbrock, D. Göddeke, S. Turek, Lattice-Boltzmann simulation of the shallow-water equations with fluid-structure interaction on multi- and manycore processors, in: R. Keller, D. Kramer, J.-P. Weiß (Eds.), Facing the Multicore Challenge, Vol. 6310 of Lecture Notes in Computer Science, Springer, 2010, pp. 92–104.

[4] S. Mallach, C. Gutwenger, Improved scalability by using hardware-aware thread affinities, in: R. Keller, D. Kramer, J.-P. Weiß (Eds.), Facing the Multicore Challenge, Vol. 6310 of Lecture Notes in Computer Science, Springer, 2010, pp. 29–41.

[5] D. Ribbrock, M. Geveler, D. Göddeke, S. Turek, Performance and accuracy of Lattice-Boltzmann kernels on multi- and manycore architectures, in: P. M. Sloot, G. van Albada, J. J. Dongarra (Eds.), International Conference on Computational Science (ICCS'10), Vol. 1 of Procedia Computer Science, 2010, pp. 239–247. doi:10.1016/j.procs.2010.04.027.

[6] D. E. Keyes, Terascale implicit methods for partial differential equations, in: X. Feng, T. P. Schulze (Eds.), Recent Advances in Numerical Methods for Partial Differential Equations and Applications, Vol. 306 of Contemporary Mathematics, American Mathematical Society, 2002, pp. 29–84.

[7] P. Colella, T. H. Dunning Jr., W. D. Gropp, D. E. Keyes, A science–based case for large–scale simulation, Tech. rep., Office of Science, US Department of Energy, http://www.pnl.gov/scales (Jul. 2003).

[8] S. Turek, D. Göddeke, C. Becker, S. Buijssen, H. Wobker, FEAST —- realisation of hardware–oriented numerics for HPC simulations with finite elements, Concurrency and Computation: Practice and Experience 6 (2010) 2247–2265, special Issue Proceedings of ISC 2008. doi:10.1002/cpe.1584.

[9] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, O. O. Storaasli, State-of-the-art in heterogeneous computing, Scientific Programming 18 (1) (2010) 1–33. doi:10.3233/SPR-2009-0296.

[10] A. R. Brodtkorb, An asynchronous API for numerical linear algebra, Scalable Computing: Practice and Experience 9 (3) (2008) 153–163.

[11] C. Feichtinger, J. Götz, S. Donath, K. Iglberger, U. Rüde, WaLBerla: Exploiting massively parallel systems for lattice Boltzmann simulations and manycore processors, in: R. Trobec, M. Vajteršic, P. Zinterhof (Eds.), Parallel Computing – Numerics, Applications, and Trends, 2009, pp. 241–260. doi:10.1007/978-1-84882-409-6_8.

[12] S. Kühner, M. Krafczyk, Virtual fluids – An environment for integral visualisation and analysis of cad and simulation data, in: B. Girod, G. Greiner, H. Niemann, H.-P. Seidel (Eds.), Proceedings of Vision, Modeling and Visualization 2000, 2000, pp. 311–318.

[13] S. Donath, K. Iglberger, G. Wellein, T. Zeiser, A. Nitsure, U. Rüde, Performance comparison of different parallel lattice boltzmann implementations on multi-core multi-socket systems, International Journal of Computational Science and Engineering 4 (1) (2008) 3–11. doi:10.1504/IJCSE.2008.021107.

[14] J. Tölke, M. Krafczyk, TeraFLOP computing on a desktop PC with GPUs for 3D CFD, International Journal of Computational Fluid Dynamics 22 (7) (2008) 443–456. doi:10.1080/10618560802238275.

[15] N. Thürey, K. Iglberger, U. Rüde, Free surface flows with moving and deforming objects for LBM, in: L. Kobbelt, T. Kuhlen, T. Aach, R. Westermann (Eds.), Proceedings of Vision, Modeling and Visualization 2006, 2006, pp. 193–200.

[16] J. Tölke, B. Ahrenholz, J. Hegewald, M. Krafczyk, Parallel free-surface and multi-phase simulations in complex geometries using lattice Boltzmann methods, in: Third Joint HLRB and KONWIHR Result and Reviewing Workshop, 2007.

[17] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, GPU cluster for high performance computing, in: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 2004, p. 47. doi:10.1109/SC.2004.26.

[18] M. J. Chorley, D. W. Walker, Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters, Journal of Computational Science 1 (3) (2010) 168–174. doi:10.1016/j.jocs.2010.05.001.
[19] S. Williams, J. Carter, L. Oliker, J. Shalf, K. A. Yelick, Lattice Boltzmann simulation optimization on leading multicore platforms, in: IEEE International Symposium on Parallel and Distributed Processing, IEEE, 2008, pp. 1–14. doi:10.1109/IPDPS.2008.4536295.
[20] T. Pohl, N. Thürey, F. Deserno, U. Rüde, P. Lammers, G. Wellein, T. Zeiser, Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures, in: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 2004, article No. 21. doi:10.1109/SC.2004.37.
[21] F. J. Higuera, J. Jiménez, Boltzmann approach to lattice gas simulations, Europhysics Letters 9 (7) (1989) 663–668. doi:10.1209/0295-5075/9/7/009.
[22] P. L. Bhatnagar, E. P. Gross, M. Krook, A model for collision processes in gases. I. small amplitude processes in charged and neutral one-component systems, Phys. Rev. 94 (3) (1954) 511–525. doi:10.1103/PhysRev.94.511.
[23] J. G. Zhou, Lattice Boltzmann methods for shallow water flows, Springer, 2004.
[24] M. Bouzidi, M. Firdaouss, P. Lallemand, Momentum transfer of a Boltzmann-lattice fluid with boundaries, Physics of Fluids 13 (11) (2001) 3452–3459. doi:10.1063/1.1399290.
[25] A. Caiazzo, Asymptotic analysis of lattice Boltzmann method for fluid-structure interaction problems, Ph.D. thesis, Technische Universität Kaiserslautern, Scuola Normale Superiore Pisa (Feb. 2007).
[26] A. Caiazzo, M. Junk, Boundary forces in lattice Boltzmann: Analysis of momentum exchange algorithm, Computers & Mathematics with Applications 55 (7) (2008) 1415–1423. doi:10.1016/j.camwa.2007.08.004.
[27] H. Löf, S. Holmgren, Affinity-on-next-touch: Increasing the performance of an industrial PDE solver on a cc-NUMA system, in: Proceedings of the 19th Annual International Conference on Supercomputing, 2005, pp. 387–392. doi:10.1145/1088149.1088201.
[28] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms, in: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 2007, article No. 38. doi:10.1145/1362622.1362674.
[29] NVIDIA Corporation, NVIDIA CUDA programming guide version 2.3, http://www.nvidia.com/cuda (Jul. 2009).
[30] IBM Corporation, SPE runtime management library, http://www-01.ibm.com/chips/techlib/techlib.nsf/pages/main (2007).